POSIX - Gestione dei file

E.Mumolo, DEEI

mumolo@units.it

Gestione file

- **Gestione file: generalità**
  - Un file per essere usato deve essere aperto (open)
  - L'operazione open:
    - localizza il file nel file system attraverso il suo pathname
    - copia in memoria il descrittore del file (i-node)
    - associa al file un intero non negativo (file descriptor), che verrà usato nelle operazioni di accesso al file, invece del pathname
    - I file standard non devono essere aperti, perchè sono aperti dalla shell.
    - Sono associati ai file descriptor 0 (input), 1 (output) e 2 (error).
    - La close rende disponibile il file descriptor per ulteriori usi

- **Nota:**
  - Un file può essere aperto più volte, e quindi avere più file descriptor associati contemporaneamente

Gestione file

- **System call:**

```
int open(const char *path, int oflag, ...);
```

- apre (o crea) il file specificato da **pathname** (assoluto o relativo), secondo la modalità specificata in **oflag**
- restituisce il file descriptor con il quale ci si riferirà al file successivamente (o -1 se errore)
- **Valori di oflag**
  - **O_RDONLY** read-only (0)
  - **O_WRONLY** write-only (1)
  - **O_RDWR** read and write (2)
  - Solo una di queste costanti può essere utilizzata in oflag
  - Altre costanti (che vanno aggiunte in or ad una di queste tre) permettono di definire alcuni comportamenti particolari

Gestione file

- Altri valori di oflag:
  - **O_APPEND** append
  - **O_CREAT** creazione del file
  - **O_EXECL** con **O_CREAT**, ritorna un errore se il file esiste
  - **O_TRUNC** se il file esiste, viene svuotato
  - **O_NONBLOCK** file speciali (discusso in seguito)
  - **O_SYNC** synchronous write

  → Se si specifica **O_CREAT**, è necessario specificare anche i permessi iniziali come terzo argomento
- La maschera **O_ACCMODE** (uguale a 3) permette di isolare la modalità di accesso
  - **oflag & O_ACCMODE** può essere uguale a **O_RDONLY, O_WRONLY**, oppure **O_RDWR**

Gestione file

- **`int creat(const char *path, mode_t mode);`**
  - crea un nuovo file normale di specificato pathname, e lo apre in scrittura
  - mode specifica i permessi iniziali; l'owner è l'effective user-id del
  - processo
  - se il file esiste già, lo svuota (owner e mode restano invariati)
  - restituisce il file descriptor, o -1 se errore
- **Equivalenze:**
  - **`creat(path, mode);`**
  - **`open(path, O_WRONLY | O_CREAT | O_TRUNC, mode);`**

- **`int close(int filedes);`**
  - chiude il file descriptor **filedes**
  - restituisce l'esito dell'operazione (0 o -1)
  - Quando un processo termina, tutti i suoi file vengono comunque chiusi automaticamente

Gestione file

- **Ogni file aperto e' associato a:**
  - un "current file offset", la posizione attuale all'interno del file
  - un valore non negativo che misura il numero di byte dall'inizio del file
  - operazioni **read**/**write** leggono dalla posizione attuale e incrementano il current file offset in avanti del numero di byte letti o scritti
- **Quando viene aperto un file**
  - Il current file descriptor viene posizionato a 0…
  - a meno che l'opzione **O_APPEND** non sia specificata

Gestione file

- **`off_t lseek(int filedes, off_t offset, int whence);`**
  - sposta la posizione corrente nel file **filedes** di **offset** bytes a partire dalla posizione specificata in **whence**:
    - **SEEK_SET** dall'inizio del file
    - **SEEK_CUR** dalla posizione corrente
    - **SEEK_END** dalla fine del file
    - restituisce la posizione corrente dopo la **lseek**, o **-1** se errore
- **lseek** non effettua alcuna operazione di I/O

Gestione file

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

- legge in ***buf** una sequenza di **nbyte** byte dalla posizione corrente del file **filedes**
- aggiorna la posizione corrente
- restituisce il numero di bytes effettivamente letti, o -1 se errore
- Esistono un certo numero di casi in cui il numero di byte letti e' inferiore al numero di byte richiesti:
  - Fine di un file regolare
  - Per letture da stream provenienti dalla rete
  - Per letture da terminale
  - etc.

`ssize_t write(int filedes, const void *buf, size_t nbyte);`

- scrive da ***buf** una sequenza di **nbyte** byte dalla posizione corrente del file **filedes**
- aggiorna la posizione corrente
- restituisce il numero di bytes effettivamente scritti, o -1 se errore

Esempi

- **SEEK**

```
#include  <sys/types.h>
#include  <unistd.h>
int main(void)
{
   if (lseek(STDIN_FILENO, 0, SEEK_CUR) == -1) printf("cannot seek\n");
   else printf("seek OK\n");
   exit(0);
}
```

- **CAT**

```
#include <unistd.h>
#define BUFFSIZE        8192
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");
                exit(1);
        }
   if (n < 0) { perror("read error"); exit(1); }

   exit(0);
}
```

```
/* copia di file. Tiene conto delle scritture incomplete */
#include <stdio.h>
#include <errno.h>
#include <fcntl.h>
#define BUFSIZE 512
void copyok(char *from, char *to);

main()
{
   char nome1[10], nome2[10];
   printf("nomi file? "); scanf("%s %s",nome1,nome2);
   printf("nomi letti: %s , %s\n", nome1,nome2);
   copyok(nome1,nome2);
}
void copyok(char *from, char *to)
{
        int fromfd, tofd, nread, nwrite, n;
        char buf[BUFSIZE];

        if((fromfd = open(from,0)) == -1)  syserr("from");
        if ((tofd = creat(to,0666)) == -1) syserr("to");
        while ((nread = read(fromfd, buf, sizeof(buf))) != 0) {
           if (nread == -1) syserr("read");
           nwrite = 0;
           do{
                if ((n=write(tofd,&buf[nwrite], nread - nwrite)) == -1) syserr("write");
                nwrite += n;
           } while (nwrite < nread);
        }
        if(close(fromfd) == -1 || close(tofd) == -1) syserr("close");
}
```

Gestione file

BUFFSIZE	User CPU (seconds)	System CPU (seconds)	Clock time (seconds)	# loops
1	23.8	397.9	423.4	1468802
2	12.3	202.0	215.2	734401
4	6.1	100.6	107.2	367201
8	3.0	50.7	54.0	183601
16	1.5	25.3	27.0	91801
32	0.7	12.8	13.7	45901
64	0.3	6.6	7.0	22950
128	0.2	3.3	3.6	11475
256	0.1	1.8	1.9	5738
512	0.0	1.0	1.1	2869
1024	0.0	0.6	0.6	1435
2048	0.0	0.4	0.4	718
4096	0.0	0.4	0.4	359
8192	0.0	0.3	0.3	180
16384	0.0	0.3	0.3	90
32768	0.0	0.3	0.3	45
65536	0.0	0.3	0.3	23
131072	0.0	0.3	0.3	12

Bufferizzazione a livello utente

```
#include "boolean.h"
#include "stream.h"
#include <stdio.h>
#include <errno.h>
#include <fcntl.h>
void copybuffered(char *from, char *to);
void timestart();
void timestop(char *msg);
main()
{
   char nome1[10], nome2[10];
   printf("nomi file? "); scanf("%s %s",nome1,nome2);
   printf("nomi letti: %s , %s\n", nome1,nome2);
   copy2(nome1,nome2);
}
void copy2(char *from, char *to)
{
        STREAM *stfrom, *stto;
        int c; extern int errno;
        if((stfrom = Sopen(from,"r")) == NULL)  syserr(from);
        if((stto = Sopen(to,"w")) == NULL) syserr(to);
        timestart();
           while((c=Sgetc(stfrom)) != -1) if(!Sputc(stto,c)) syserr("Sputc");
        timestop("tempi di copy2");
        if(errno != 0) syserr("Sgetc");
        if (!Sclose(stfrom) || !Sclose(stto)) syserr("Sclose");
}
```

User buffer STREAM

```
#include <fcntl.h>
#include <stdio.h>
#include "boolean.h"
#include "stream.h"
extern int errno;
STREAM *Sopen(char *path,char *dir)
{
        STREAM *z;
        int fd, flags; char *malloc();
        switch(dir[0]){
        case 'r':
                flags = O_RDONLY;
                break;
        case 'w':
                flags = O_WRONLY | O_CREAT | O_TRUNC;
                break;
        default:
                errno = SEINVAL;
                return(NULL);
        }
        if((fd=open(path,flags,0666)) == -1) return(NULL);
        if((z = (STREAM *)malloc(sizeof(STREAM))) == NULL) {
                    errno = SENOMEM; return(NULL);
        }
        z->fd = fd; z->dir = dir[0]; z->total = z->next = 0;
        return(z);
}
```

STREAM (cont.)

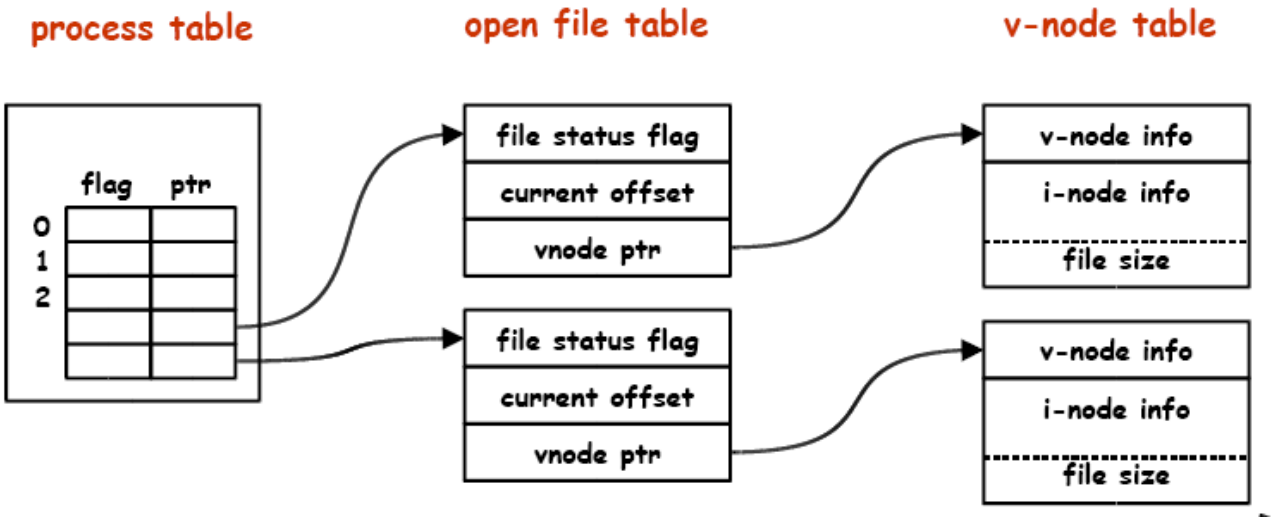
```
static BOOLEAN readbuf(STREAM *z)
{
        switch(z->total = read(z->fd, z->buf, sizeof(z->buf))) {
        case -1:
                return(FALSE);
        case 0:
                errno = 0;
                return(FALSE);
        default:
                z->next = 0;
                return(TRUE);
        }
}
static BOOLEAN writebuf(STREAM *z)
{
        int n,total;
        total=0;
        while(total < z->next){
                if((n=write(z->fd,&z->buf[total],z->next - total))==-1)
                        return(FALSE);
                total += n;
        }
        z->next = 0;
        return(TRUE);
}
```

STREAM (cont.)

```
int Sgetc(STREAM *z)
{
        int c;

        if(z->next >= z->total && !readbuf(z)) return(-1);
        return(z->buf[z->next++] & 0377);
}

BOOLEAN Sputc(STREAM *z, char c)
{
        z->buf[z->next++] = c;
        if (z->next >= sizeof(z->buf)) return(writebuf(z));
        return(TRUE);
}

BOOLEAN Sclose(STREAM *z)
{
        int fd;

        if(z->dir == 'w' && !writebuf(z)) return(FALSE);
        fd = z->fd;
        free(z);
        return(close(fd) != -1);
}
```

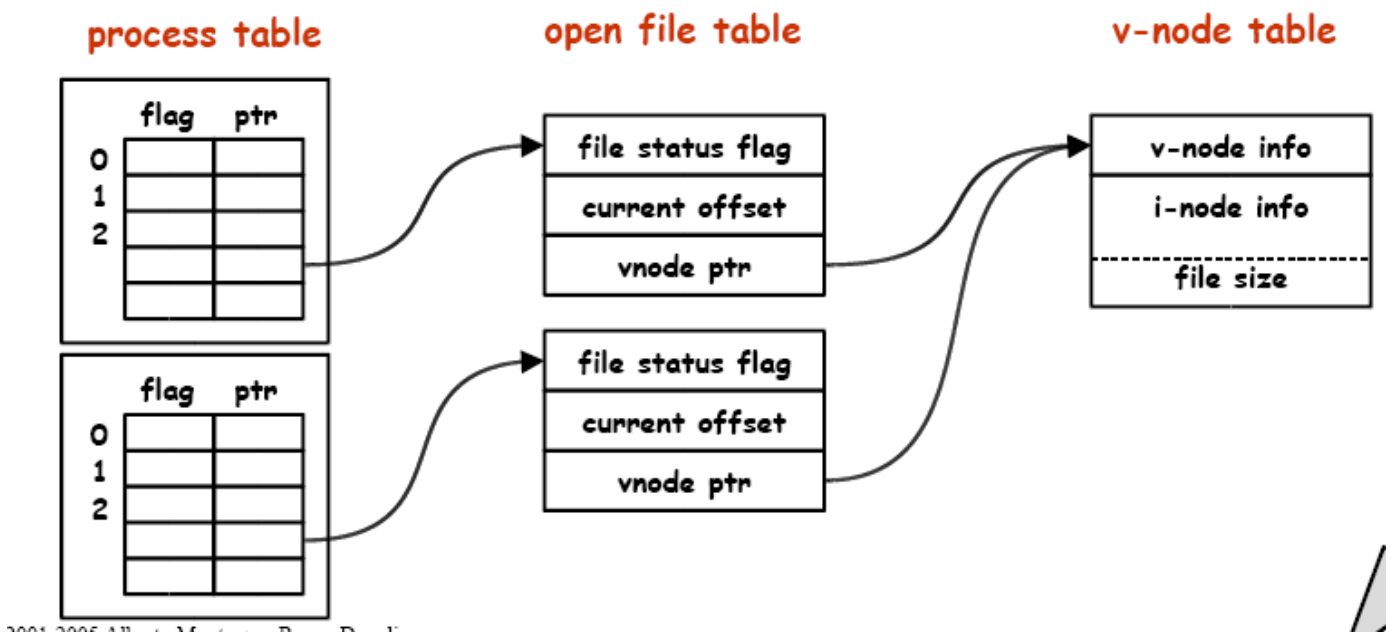
Condivisione file

- **Tabelle coinvolte: User File Descriptor Table, File Table, Inode table**
- Esempio: un processo che apre due file distinti



Condivisione file

- Esempio: due processi che aprono lo stesso file



Qualche precisazione

- Alla conclusione di ogni write
  - il current offset nella file table entry viene incrementato
  - e il current offset è maggiore della dimensione del file nella v-node table entry, questa viene incrementata
- se il file è aperto con il flag O_APPEND
  - un flag corrispondente è settato nella file table entry
  - ad ogni write, il current offset viene prima posto uguale alla dimensione del file nella v-node table entry
- lseek modifica unicamente il current offset nella file table entry corrispondente
- E' possibile che più file descriptor entry siano associate a una singola file table entry
  - tramite la funzione dup, all'interno dello stesso processo
  - tramite fork, tra due processi diversi
- E' interessante notare che esistono due tipi di flag:
  - alcuni sono associati alla file descriptor entry, e quindi sono particolari del processo
  - altri sono associati alla file table entry, e quindi possono essere condivisi fra più processi
  - Esiste la possibilità di modificare questi flag (funzione fcntl)
- E' importante notare che questo sistema di strutture dati condivise può portare a problemi di concorrenza

Duplicazione descrittori

- **Funzione dup**
  - Seleziona il più basso file descriptor libero della tabella dei file descriptor
  - Assegna la nuova file descriptor entry al file descriptor selezionato
  - Ritorna il file descriptor selezionato
- **Funzione dup2**
- Con **dup2**, specifichiamo il valore del nuovo descrittore come argomento **filedes2**
- Se **filedes2** è già aperto, viene chiuso e sostituito con il descrittore duplicato
- Ritorna il file descriptor selezionato

Modifica descrittori

```
int fcntl(int filedes, int cmd, … /* int arg */)
```

- La funzione fcntl può cambiare le proprietà di un file aperto
  - Argomento 1: è il descrittore del file su cui operare
  - Argomento 2: è il comando da eseguire
  - Argomento 3: quando presente, parametro del comando;
    - in generale, un valore intero
    - nel caso di record locking, un puntatore
- Comandi:
  - duplicazione di file descriptor (**F_DUPFD**)
  - get/set file descriptor flag (**F_GETFD, F_SETFD**)
  - get/set file status flag (**F_GETFL, F_SETFL**)
  - get/set async. I/O ownership (**F_GETOWN, F_SETOWN**)
  - get/set record locks (**F_GETLK, F_SETLK, F_SETLKW**)

Modifica descrittori

`int fcntl(int filedes, F_DUPFD, int min)`

- Duplica il file descriptor specificato da **filedes**
- Ritorna il nuovo file descriptor
- Il file descriptor scelto è uguale al valore più basso corrispondente ad un file descriptor non aperto e che sia maggiore o uguale a **min**

`int fcntl(int filedes, F_GETFD)`

- Ritorna i file descriptor flag associati a **filedes**
- Attualmente, è definito un solo file descriptor flag:
- Se **FD_CLOEXEC** è true, il file descriptor viene chiuso eseguendo una **exec**

Modifica descrittori

`int fcntl(int filedes, F_SETFD, int newflag)`

- Modifica i file descriptor flag associati a **filedes**, utilizzando il terzo argomento come nuovo insieme di flag

`int fcntl(int filedes, F_GETFL)`

- Ritorna i file status flag associati a **filedes**
- I file status flag sono quelli utilizzati nella funzione **open**
- Per determinare la modalità di accesso, è possibile utilizzare la maschera **ACC_MODE**
- Per determinare gli altri flag, è possibile utilizzare le costanti definite (**O_APPEND, O_NONBLOCK, O_SYNC**)

Modifica descrittori

`int fcntl(int filedes, F_SETFL, int newflag)`

- Modifica i file status flag associati a **filedes** con il valore specificato in **newflag**
- I soli valori che possono essere modificati sono **O_APPEND,O_NONBLOCK, O_SYNC**; l'access mode deve rimanere inalterato

- **Gli altri comandi di fcntl verranno trattati in seguito**
  - Quando introduciamo il concetto di segnale
  - Quando introduciamo il concetto di locking

- Il codice proposto prende un argomento singolo da linea di comando
- (che specifica un descrittore di file) e stampa una descrizione delle flag
- del file per quel descrittore

```
// Questo programma prende un argomento singolo da linea di comando (che specifica un
// descrittore di file) e stampa una descrizione delle flag del file per quel descrittore
#include  <sys/types.h>
#include  <fcntl.h>
#include  <unistd.h>
Int main(int argc, char *argv[])
{
  int   accmode, val;
  if (argc != 2) {
    printf("usage: a.out <descriptor#>");
    exit(1);
  }
  if ( (val = fcntl(atoi(argv[1]), F_GETFL)) < 0) {
    perror("fcntl error for fd %d", atoi(argv[1]));
    exit(1);
  }

  accmode = val & O_ACCMODE;
  if     (accmode == O_RDONLY)  printf("read only");
  else if (accmode == O_WRONLY) printf("write only");
  else if (accmode == O_RDWR)   printf("read write");
  else printf("unknown access mode");

  if (val & O_APPEND)     printf(", append");
  if (val & O_NONBLOCK)   printf(", nonblocking");
#if !defined(_POSIX_SOURCE) && defined(O_SYNC)
  if (val & O_SYNC)     printf(", synchronous writes");
#endif
  putchar('\n');
  exit(0);
}
```

Modifica descrittori

- **Altro esempio d'uso di fcntl:**
  - La funzione **set_fl** mette a 1 i flag specificati
  - La funzione richiede prima i flag correnti, utilizza la maschera con OR, e salva di nuovi i flag

```
void set_fl(int fd, int flags)
{
   int val;
   if ( (val = fcntl(fd, F_GETFL, 0)) < 0)
     err_sys("fcntl F_GETFL error");
   val |= flags; /* turn on flags */
   if (fcntl(fd, F_SETFL, val) < 0)
     err_sys("fcntl F_SETFL error");
}
```

Altre funzioni d'accesso a parametri I/O

`int iocntl(int filedes, int request, …)`

- **Categorie di comandi:** disk labels I/O, file I/O, magnetic tapes I/O, socket I/O, terminal I/O

`fsync()`

- effettua l'operazione di "flush" dei dati bufferizzati dal s.o. per il file descriptor fd, ovvero li scrive sul disco o sul dispositivo sottostante
- **Motivazioni**
  - il gestore del file system può mantenere i dati in buffer in memoria per diversi secondi, prima di scriverli su disco
  - per ragioni efficienza
  - ritorna 0 in caso di successo, -1 altrimenti

Funzione select

```
int select(int n, fd_set *readfds, fd_set *writefds,
           fd_set *exceptfds, struct timeval *timeout);
```

- permette ad un processo di aspettare contemporaneamente su file descriptor multipli, con un timeout opzionale

- **Parametri di input**
  - *fds sono insiemi di file descriptor, realizzati tramite mappe di bit
  - n è la dimensione massima di questi insiemi
  - timeval è il timeout opzionale

- **Parametri di output**
  - ritorna il numero di file descriptor che sono pronti per operazioni di I/O immediate
  - modifica gli insiemi di wait descriptor, ponendo a 1 i bit relativi ai file descriptor pronti per l'I/O

File – funzioni atomiche

- **Ogni processo possiede una file descriptor entry ed una file table entry "personale"**
- **La v-node table è condivisa**
- **Si consideri il seguente interleaving tra i due processi:**
  - inizialmente, la dimensione del file sia 1500
  - processo 1 esegue **lseek** e pone il suo current offset a 1500
  - processo 2 esegue **lseek** e pone il suo current offset a 1500
  - processo 2 esegue **write**; scrive i byte 1500-1599 e pone la dimensione del file a 1600
  - processo 1 esegue **write**; sovrascrive i byte 1500-1599 e pone la dimensione del file a 1600
  - **Soluzione: utilizzate open con O_APPEND (write atomica)**

File – funzioni atomiche

- Si consideri un singolo processo che voglia creare un file, riportando un messaggio di errore se il file è già esistente
- Nelle prime versioni di UNIX, i flag **O_CREAT** and **O_EXCL** non esistevano
- Il codice poteva essere scritto nel modo seguente:

```
if ( ( fd = open(pathname, O_WRONLY) ) < 0)
   if (errno == ENOENT) {
        if ( (fd = creat(pathname, mode) < 0) err_sys("creat error");
        }
   else
        err_sys("open error");
```

- Questo funziona quando abbiamo un solo processo che accede al file: **Cosa succede se abbiamo due (o più) processi?**
- Cosa succede se il file viene creato da un altro processo tra l'operazione **open** e l'operazione **creat**, e questo processo scrive delle informazioni su questo file?
  - I dati vengono persi, perché sovrascritti
  - Il problema è evitato dalla system call open con i flag **O_CREAT** e **O_EXCL**

Altre chiamate di sistema per il file system

- **Informazioni sui file**

```
#include <sys/types.h>
#include <sys/stat.h>
int stat(const char *pathname, struct stat *buf);
int fstat(int filedes, struct stat* buf);
int lstat(const char *pathname, struct stat *buf);
```

- **Le tre funzioni ritornano un struttura stat contenente informazioni sul file**
  - **stat** identifica il file tramite un pathname
  - **fstat** identifica un file aperto tramite il suo descrittore
  - **lstat**, se applicato ad un link simbolico, ritorna informazioni sul link simbolico, non sul file linkato

Altre chiamate di sistema per il file system

- Il secondo argomento delle funzioni stat è un puntatore ad una struttura di informazioni sul file specificato

```
struct stat {
   mode_t st_mode;         // File type & mode
   ino_t st_ino;           // i-node number
   dev_t st_dev;           // device number (file system)
   dev_t st_rdev;          // device n. for special files
   nlink_t st_nlink;       // number of links
   uit_t uid;              // user ID of owner
   gid_t gid;              // group ID of owner
   off_t st_size;          // size in bytes, for reg. files
   time_t st_atime;        // time of last access
   time_t st_mtime;        // time of last modif.
   time_t st_ctime;        // time of last status change
   long st_blksize;        // best I/O block size
   long st_blocks;         // number of 512-byte blocks
}
```

```
#include  <sys/types.h>
#include  <sys/stat.h>
#include  <unistd.h>
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";
    else if (S_ISBLK(buf.st_mode))  ptr = "block special";
    else if (S_ISFIFO(buf.st_mode)) ptr = "fifo";
#ifdef  S_ISLNK
    else if (S_ISLNK(buf.st_mode))  ptr = "symbolic link";
#endif
#ifdef  S_ISSOCK
    else if (S_ISSOCK(buf.st_mode)) ptr = "socket";
#endif
    else        ptr = "** unknown mode **";
    printf("%s\n", ptr);
  }
  exit(0);
}
```

Altre chiamate di sistema per il file system

- ***set-user-ID e set-group-ID*:**
  - in **st_mode**, esiste un flag (*set-user-ID*) che fa in modo che quando questo file viene eseguito, *effective user id* == **st_uid**
  - in **st_mode**, esiste un flag (*set-group-ID*) che fa in modo che quando questo file viene eseguito, *effective group id* == ***st_gid***

- **E' possibile utilizzare questi due bit per risolvere il problema dell'accesso al file passwd:**
  - l'owner del comando **passwd** è **root**
  - quando **passwd** viene eseguito, il suo *effective user id* è uguale a **root**
  - il comando può allora modificare in scrittura il file **/etc/passwd**

Altre chiamate di sistema per il file system

- Costanti per accedere ai diritti di lettura e scrittura contenuti in st_mode
  - **S_ISUID** set-user-ID
  - **S_ISGID** set-group-ID
  - **S_IRUSR** accesso in lettura, owner
  - **S_IWUSR** accesso in scrittura, owner
  - **S_IXUSR** accesso in esecuzione, owner
  - **S_IRGRP** accesso in lettura, gruppo
  - **S_IWGRP** accesso in scrittura, gruppo
  - **S_IXGRP** accesso in esecuzione, gruppo
  - **S_IROTH** accesso in lettura, altri
  - **S_IWOTH** accesso in scrittura, altri
  - **S_IXOTH** accesso in esecuzione, altri

Altre chiamate di sistema per il file system

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

- Quando si accede ad un file, vengono utilizzati effective uid e effective gid
- A volte, può essere necessario verificare l'accessibilità in base a real uid e real gid
- Per fare questo, si utilizza access
- mode è un maschera ottenuta tramite bitwise or delle seguenti costanti:
  - R_OK test per read permission
  - W_OK test per write permission
  - X_OK test per execute permission

Altre chiamate di sistema per il file system

```
mode_t umask(mode_t cmask);
```

- Cambia la maschera utilizzata per la creazione dei file; ritorna la maschera precedente;
- Per formare la maschera, si utilizzano le costanti S_IRUSR, S_IWUSR viste in precedenza
- Funzionamento:
  - La maschera viene utilizzata tutte le volte che un processo crea un nuovo file
  - Tutti i bit che sono accesi nella maschera, verranno spenti nell'access mode del file creato

Altre chiamate di sistema per il file system

```
int chmod (const char* path, mode_t mode);
int fchmod (int filedes, mode_t mode);
```

- Cambia i diritti di un file specificato dal pathname (chmod) o di un file aperto (fchmod)
- Per cambiare i diritti di un file, l'effective uid del processo deve essere uguale all'owner del file oppure deve essere uguale a root

```
int chown(char* pathname, uid_t owner, gid_t group);
int fchown(int filedes, uid_t owner, gid_t group);
int lchown(char* pathname, uid_t owner, gid_t group);
```

- Queste tre funzioni cambiano lo user id e il group id di un file
  - Nella prima, il file è specificato come pathname
  - Nella seconda, il file aperto è specificato da un file descriptor
  - Nella terza, si cambia il possessore del link simbolico, non del file stesso
- Restrizioni:
  - In alcuni sistemi, solo il superuser può cambiare l'owner di un file (per evitare problemi di quota)
  - Costante POSIX_CHOWN_RESTRICTED definisce se tale restrizione è in vigore

Hard link

- **Hard link per directory:**
- ogni directory *dir* ha un numero di hard link maggiore uguale a 2:
  - Uno perché *dir* possiede un link nella sua directory genitore
  - Uno perché *dir* possiede un'entry "." che punta a se stessa
- ogni sottodirectory di *dir* aggiunge un hard link:
  - Uno perché la sottodirectory possiede un'entry ".." che punta alla directory genitore

Hard link

```
int link(char* oldpath, char* newpath);
```

- crea un nuovo link ad un file esistente
- operazione atomica: aggiunge la directory entry e aumenta il numero di link per l'inode identificato da **oldpath**

- errori:
  - **oldpath** non esiste
  - **newpath** esiste già
  - solo root può creare un hard link ad una directory (per evitare di creare loop, che possono causare problemi)
  - **oldpath** e **newpath** appartengono a file system diversi
- utilizzato dal comando **ln**

Hard link

```
int unlink(char* path);
int remove(char* path);
```

- rimuove un hard link per il file specificato da path
- operazione atomica: rimuove la directory entry e decrementa di uno il numero di link del file
- errori:
  - path non esiste
  - un utente diverso da root cerca di fare unlink su una directory
- utilizzato dal comando **rm**
- un file può essere effettivamente cancellato dall'hard disk quando il suo conteggio raggiunge 0

- **Unlink: cosa succede quando un file è aperto?**
  - Il file viene rimosso dalla directory
  - Il file non viene rimosso dal file system fino alla chiusura
  - Quando il file viene chiuso, il sistema controlla se il numero di hard link è sceso a zero; in tal caso rimuove il file

Altre chiamate di sistema per il file system

```
int rename(char* oldpath, char* newpath);
```

- Cambia il nome di un file da oldpath a newpath
- Casi:
  - se **oldpath** specifica un file regolare, **newpath** non può essere una directory esistente
  - se **oldpath** specifica un file regolare e **newpath** è un file regolare
  - esistente, questo viene rimosso e sostituito
  - sono necessarie write permission su entrambe le directory
  - se **oldpath** specifica una directory, **newpath** non può essere un file regolare esistente
  - se **oldpath** specifica una directory, **newpath** non può essere una directory non vuota

Link simbolici

- file speciali che contengono il pathname assoluto di un altro file
- E' un puntatore indiretto ad un file (a differenza degli hard link che sono puntatori diretti agli inode)
- Introdotto per superare le limitazioni degli hard link:
  - hard link possibili solo fra file nello stesso file system
  - hard link a directory possibili solo al superuser
- attenzione: un link simbolico può introdurre loop
  - i programmi che analizzano il file system devono essere in grado di gestire questi loop (esempio in seguito)

Link simbolici

```
int symlink(char* oldpath, char* newpath);
```

- crea una nuova directory entry **newpath** con un link simbolico che punta al file specificato da **oldpath**
- nota:
  - **oldpath** e **newpath** non devono risiedere necessariamente nello stesso file system

```
int readlink(char* path, char* buf, int size);
```

- poiché la open segue il link simbolico, abbiamo bisogno di una system call per ottenere il contenuto del link simbolico
- questa system call copia in **buf** il valore del link simbolico

Tempi

- **Tre valori temporali sono memorizzati nella struttura stat**
  - **st_atime** (opzione **–u** in **ls**)
    - Ultimo tempo di accesso
    - read, creat/open (when creating a new file), utime
  - **st_mtime** (default in **ls**)
    - Ultimo tempo di modifica del contenuto
    - write, creat/open, truncate, utime
  - **st_ctime** (opzione **–c** in **ls**)
    - Ultimo cambiamento nello stato (nell'inode)
    - chmod, chown, creat, mkdir, open, remove, rename, truncate, link, unlink, utime, write
- Attenzione ai cambiamenti su contenuto, inode e accesso per quanto riguarda le directory
- Per modificare st_ctime

```
int utime(char* pathname, struct utimbuf *times);
struct utimbuf {
    time_t actime;
    time_t modtime;
}
```

Directory

`int mkdir(char* path, mode_t);`

- Crea una nuova directory vuota dal path specificato
- Modalità di accesso:
  - I permessi specificati da mode_t vengono modificati dalla maschera specificata da umask
  - E' necessario specificare i diritti di esecuzione (search)

`int rmdir(char* path);`

- Rimuove la directory vuota specificata da path
- Se il link count della directory scende a zero, la directory viene effettivamente rimossa; altrimenti si rimuove la directory

Directory

- **Ogni processo** ha una directory corrente a partire dalla quale vengono fatte le ricerche per i pathname relativi
- **System call:**

```
int chdir(char* path);
int fchdir(int filedes);
```

- Cambia la directory corrente associata al processo, utilizzando un pathname oppure un file descriptor
- **Funzione:**

```
char* getcwd(char* buf, size_t size);
```

- Legge la directory corrente e riporta il pathname assoluto nel buffer specificato da **buf** e di dimensione **size**