System call per l'accesso a file

Nome	Significato
`open`	apre un file in lettura e/o scrittura o crea un nuovo file
`creat`	crea un file nuovo
`close`	chiude un file precedentemente aperto
`read`	legge da un file
`write`	scrive su un file
`lseek`	sposta il puntatore di lettura/scrittura ad un byte specificato
`unlink`	rimuove un file
`remove`	rimuove un file
`fcntl`	controlla gli attributi associati ad un file

La system call `lseek`

La system call `lseek` permette l'**accesso random** ad un file, cambiando il numero del prossimo byte da leggere/scrivere.

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

Il parametro `filedes` è un descrittore di file.
Il parametro `offset` determina la nuova posizione del puntatore in lettura/scrittura.
Il parametro `start_flag` specifica da dove deve essere calcolato l'`offset`. `startflag` può assumere uno dei seguenti valori simbolici:

`SEEK_SET` (0) : `offset` è misurato dall'inizio del file
`SEEK_CUR` (1) : `offset` è misurato dalla posizione corrente del puntatore
`SEEK_END` (2) : `offset` è misurato dalla fine del file

`lseek` ritorna la nuova posizione del puntatore.

Offset validi e non validi

Il parametro `offset` può essere **negativo**, cioè sono ammessi **spostamenti all'indietro** a partire dalla posizione indicata da `start_flag`, purchè però non si vada oltre l'inizio del file.
Tentativi di spostamento prima dell'inizio del file generano un errore.

È possibile spostarsi **oltre la fine del file**.
Ovviamente non ci saranno dati da leggere in tale posizione.
Futuri accessi tramite la `read` ai byte compresi tra la vecchia fine del file e la nuova posizione danno come risultato il carattere ASCII null.

Esempio:

```
off_t newpos;
⋮
newpos = lseek(fd, (off_t)-16, SEEK_END);
```

Scrivere alla fine di un file

Vi sono due modi per scrivere alla fine di un file:

- usare `lseek` per spostarsi alla fine del file e poi scrivere:
  ```
  lseek(filedes, (off_t)0, SEEK_END);
  write(filedes, buf, BUFSIZE);
  ```
- usare `open` con il flag `O_APPEND`:
  ```
  filedes = open("nomefile", O_WRONLY | O_APPEND);
  write(filedes, buf, BUFSIZE);
  ```

Altri flag utili nell'utilizzo di `open` sono i seguenti:

- `O_RDONLY`: apre il file specificato in sola lettura.
- `O_RDWR`: apre il file specificato in lettura e scrittura.
- `O_CREAT`: crea un file con il nome specificato; con questo flag è possibile specificare come terzo argomento della `open` un numero **ottale** che rappresenta i permessi da associare al nuovo file (e.g., `0644`).
- `O_TRUNC`: tronca il file a zero.
- `O_EXCL`: flag "esclusivo"; un tipico esempio d'uso è il seguente:
  ```
  filedes = open("nomefile", O_WRONLY | O_CREAT | O_EXCL, 0644);
  ```
  che provoca un fallimento nel caso in cui il file `nomefile` esista già.

Eliminare un file

Per cancellare un file vi sono due system call a disposizione del programmatore:

```
#include <unistd.h>
int unlink(const char *pathname);

#include <stdio.h>
int remove(const char *pathname);
```

Entrambe le system call hanno un unico argomento: il pathname del file da eliminare.

Esempio:

```
remove("/tmp/tmpfile");
```

Inizialmente esisteva soltanto `unlink`, mentre `remove` è stata aggiunta in seguito come specifica dello standard ANSI C per l'eliminazione dei file regolari.

La chiamata di sistema `fcntl`

La system call `fcntl` permette di esercitare un certo grado di controllo su file già aperti:

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

int fcntl(int filedes, int cmd, ...);
```

I parametri dal terzo in poi variano a seconda del valore dell'argomento `cmd`. L'utilizzo più comune di `fcntl` si ha quando `cmd` assume i seguenti valori:

- `F_GETFL`: fa in modo che `fcntl` restituisca il valore corrente dei flag di stato (come specificati nella `open`).
- `F_SETFL`: imposta i flag di stato in accordo al valore del terzo parametro. Esempio:

```
if(fcntl(filedes, F_SETFL, O_APPEND) == -1)
  printf("fcntl error\n");
```

Esempio d'uso di `fcntl`

```
#include <fcntl.h>

int filestatus(int filedes) {
  int arg1;

  if((arg1 = fcntl(filedes, F_GETFL)) == -1) {
    printf("filestatus failed\n");
    return -1;
  }

  printf("File descriptor %d", filedes);

  switch(arg1 & O_ACCMODE) {
    case O_WRONLY:
      printf("write only");
      break;
```

Esempio d'uso di `fcntl` (...continua)

```
    case O_RDWR:
      print("read write");
      break;
    case O_RDONLY:
      print("read only");
      break;
    default:
      print("No such mode");
      break;
  }

  if(arg1 & O_APPEND)
    printf(" - append flag set");

  printf("\n");
  return 0;
}
```

dove `O_ACCMODE` è una maschera appositamente definita in `<fcntl.h>`.

stat e fstat

Le informazioni e le proprietà dei file (dispositivo del file, numero di inode, tipo del file, numero di link, UID, GID, dimensione in byte, data ultimo accesso/ultima modifica, informazioni sui blocchi che contengono il file) sono contenute negli inode. Le chiamate di sistema `stat` e `fstat` permettono di accedere in lettura alle informazioni e proprietà associate ad un file:

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

int stat(const char *pathname, struct stat *buf);

int fstat(int filedes, struct stat *buf);
```

L'unica differenza fra le due system call consiste nel fatto che, mentre `stat` prende come primo argomento un pathname, `fstat` opera su un descrittore di file. Quindi `fstat` può essere utilizzata soltanto su file già aperti tramite la `open`.

La struttura stat

`stat` è una struttura definita in `<sys/stat.h>` che comprende i seguenti componenti (i tipi sono definiti in `<sys/types.h>`):

Tipo	Nome	Descrizione
`dev_t`	`st_dev`	logical device
`ino_t`	`st_ino`	inode number
`mode_t`	`st_mode`	tipo di file e permessi
`nlink_t`	`st_nlink`	numero di link non simbolici
`uid_t`	`st_uid`	UID
`gid_t`	`st_gid`	GID
`dev_t`	`st_rdev`	membro usato quando il file rappresenta un device
`off_t`	`st_size`	dimensione **logica** 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 dell'ultima modifica alle informazioni della struttura `stat`
`long`	`st_blksize`	dimensione del blocco per il file
`long`	`st_blocks`	numero di blocchi allocati per il file

Esempio (I)

Il programma `lookout.c`, data una lista di nomi di file, controlla ogni minuto se un file è stato modificato. Nel caso ciò avvenga termina l'esecuzione stampando un messaggio che informa l'utente dell'evento.

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

#define MFILE 10

void cmp(const char *, time_t);
struct stat sb;

main(int argc, char **argv) {
  int j;
  time_t last_time[MFILE+1];
```

Esempio (II)

```
if(argc<2) {
  fprintf(stderr, "uso: lookout file1 file2 ...\n");
  exit(1);
}

if(--argc>MFILE) {
  fprintf(stderr, "lookout: troppi file\n");
  exit(1);
}

for(j=1; j<=argc; j++) {
  if(stat(argv[j], &sb) == -1) {
    fprintf(stderr, "lookout: errore nell'accesso al file %s\n", argv[j]);
    exit(1);
  }

  last_time[j] = sb.st_mtime;
}
```

Esempio (III)

```
  for(;;) {
    for(j=1; j<=argc; j++)
      cmp(argv[j], last_time[j]);

    sleep(60);
  }

}

void cmp(const char *name, time_t last) {

  if(stat(name, &sb) == -1 || sb.st_mtime != last) {
    fprintf(stderr, "lookout: il file %s e' stato modificato\n", name);
    exit(0);
  }

}
```

Directory

Le **directory** unix sono **file**.

Molte system call per i file ordinari possono essere utilizzate per le directory. E.g. `open`, `read`, `fstat`, `close`.

Tuttavia le directory non possono essere create con `open`, `creat`.

Esiste un insieme di system call speciali per le directory.

Una directory è rappresentata in memoria da una **tabella**, con una entry per ogni file nella directory.

inode	name
⋮	⋮

Ogni entry è una struttura di tipo `dirent` definita in `<dirent.h>`:

```
ino_t d_ino;
char d_name[ ];
```

Il primo campo contiene l'inode del file, il secondo il nome del file. Se `d_ino`=0, allora lo slot è libero.

Creazione e apertura di una directory

Creazione di una directory:

```
#include <sys/types.h>
#include <sys/stat.h>
int mkdir (const char *pathname, mode_t mode)
```

La system call `mkdir` restituisce 0 o -1 a seconda che termini con successo o meno. Al momento della creazione con `mkdir`, i link . e .. vengono inseriti nella tabella.

Apertura di una directory:

```
#include <sys/types.h>
#include <dirent.h>
DIR *opendir (const char *dirname);
```

La system call `opendir` ritorna un puntatore di tipo `DIR` oppure il null pointer, in caso di fallimento.

Lettura, riposizionamento, chiusura di una directory

Lettura di una directory:

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

La system call `readdir` restituisce un puntatore alla struttura `dirent` dove è stata copiata la prossima entry nella tabella.

Riposizionamento:

```
void rewinddir (DIR *dirptr);
```

Chiusura:

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

Esempio: lettura di una directory

```
#include <dirent.h>

int my_ls (const char *name)
{
  struct dirent *d;
  DIR *dp;

  /* apertura della directory */
  if ((dp=opendir(name)) == NULL)
     return (-1);

  /* stampa dei nomi dei file contenuti nella directory */
  while (d = readdir(dp))
  {
     if (d->d_ino != 0)
        printf("%s\n", d->d_name);
  }

  closedir(dp);
  return(0);
}
```

Tipo di un file

Come si è visto nella lezione precedente, la system call `stat` serve per accedere ad una struttura dati contenente gli attributi di un file.

L'attributo `st_mode` contiene il **file mode**, cioè una sequenza di bit ottenuti facendo l'OR bit a bit della stringa che esprime i permessi al file e una costante che determina il tipo del file (regolare, directory, speciale, etc.).

Per scoprire se un file è una directory, si può usare la macro `S_ISDIR`:

```
/* buf e' il puntatore restituito da stat */
if (S_ISDIR (buf.st_mode))
    printf("It is a directory\n");
  else
    printf("It is not a directory\n");
```

Cambiamento della directory corrente

La directory corrente di un processo è quella in cui il processo è eseguito. Tuttavia un processo può cambiare la sua directory corrente con la system call

```
#include <unistd.h>
int chdir (const char *path);
```

dove `path` è il pathname della nuova directory corrente. Il cambiamento si applica solo al processo chiamante.

Attraversamento dell'albero di una directory

La system call `ftw` consente di eseguire un'operazione `func` su tutti i file nella gerarchia della directory `path`:

```
#include <ftw.h>
int ftw (const char *path, int (*func)(), int depth);
```

Il parametro `depth` controlla il numero di file descriptor usati da `ftw`. Più grande è il valore di `depth`, meno directory devono essere riaperte, incrementando la velocità di esecuzione.

`func` è una funzione definita dall'utente, che viene passata alla routine `ftw` come puntatore a funzione. Ad ogni chiamata, `func` viene chiamata con tre argomenti: una stringa contenente il nome del file a cui `func` si applica, un puntatore ad una struttura `stat` con i dati del file, un codice intero. Il prototipo di `func` deve perciò essere:

```
int func (const char *name, const struct stat *sptr, int type);
```

L'argomento `type` contiene uno dei seguenti valori (definiti in `<ftw.h>`), che descrivono il file oggetto:

`FTW_F`	l'oggetto è un file
`FTW_D`	l'oggetto è una directory
`FTW_DNR`	l'oggetto è una directory che non può essere letta
`FTW_SL`	l'oggetto è un link simbolico
`FTW_NS`	l'oggetto non è un link simbolico e su di esso `stat` fallisce