Corso di Laboratorio di Sistemi Operativi
A.A. 2016–2017

Lezione 14

Ivan Scagnetto
ivan.scagnetto@uniud.it
Nicola Gigante
gigante.nicola@spes.uniud.it

Dipartimento di Scienze Matematiche, Informatiche e Fisiche
Università degli Studi di Udine

A.A. 2016–2017 - Primo Semestre - 25/11/2016

Accesso ai file (funzioni standard)

Funzioni ISO per l'accesso ai file

Abbiamo già visto alcune funzioni standard per l'accesso ai file.

```
FILE *fopen(const char *filename, const char *mode);
int fclose(FILE *file);
int fprintf(FILE *file, const char *format, ...);
int fscanf(FILE *file, const char *format, ...);
int fgetc(FILE *file);
int fputc(FILE *file, char c);
```

Ora approfondiremo l'argomento.

Gestione degli errori

Ogni interazione con il sistema operativo, compreso l'accesso ai file, può fallire per svariate ragioni. È necessario un meccanismo per riportare e accorgersi di **errori**.

Le funzioni della libreria standard per l'accesso ai file seguono un certo schema:

- Le funzioni che ritornano puntatori (es. `fopen()`), ritornano `NULL` in caso di errore.
- Le funzioni che ritornano un carattere (es. `fgetc()`), ritornano `EOF` in caso di errore.
- Le funzioni che non hanno nulla da restituire (es. `fclose()` o `fputc()`) restituiscono comunque un valore intero, che vale `EOF` se c'è stato un errore, oppure zero se non c'è nulla da segnalare.

Gestione degli errori

`ferror()` e `feof()`

Una funzione come `fgetc()` potrebbe aver restituito `EOF` anche perchè il file è effettivamente terminato.

Le funzioni `feof()` e `ferror()` servono a distinguere i due casi:

```
char c = fgetc(file);
if(c == EOF) {
  if(feof(file))
    printf("Il file e' terminato\n");
  else if(ferror(file))
    printf("La lettura ha causato un'errore\n");
}
```

Gestione degli errori

La variabile errno

Per riportare **quale** errore si è verificato, le varie funzioni impostano la variabile globale **errno**, dichiarata in <errno.h>, con un valore che rappresenta il motivo dell'errore.

```
FILE *file = fopen("/my/file", "r");
if(file == NULL) {
  if(errno == EACCES)
    fprintf(stderr, "Si e' verificato un errore di permessi\n");
  if(errno == EISDIR)
    fprintf(stderr, "Il file richiesto e' una directory\n");
}
```

Il file errno.h dichiara tutte le costanti del preprocessore corrispondenti ai possibili errori riportati dalle funzioni standard (come EACCESS o EISDIR nell'esempio).

Gestione degli errori

Le funzioni `perror()` e `strerror()`

Una stringa standard di descrizione della variabile `errno` si può ottenere tramite la funzione `strerror()`.

```
#include <stdio.h>

int main(int argc, char *argv) {
  if(argc < 2) {
    fprintf(stderr, "Specificare il nome di un file\n");
    return 1;
  }

  FILE *file = fopen(argv[1], "r");
  if(file == NULL) {
    fprintf(stderr, "%s: Impossibile aprire %s: %s\n",
            argv[0], filename, strerror(errno));
    return 2;
  }
  return 0;
}
```

La funzione `perror()` già vista in qualche esempio fa proprio questo.

Gestione degli errori

Le funzioni `perror()` e `strerror()`

Una stringa standard di descrizione della variabile `errno` si può ottenere tramite la funzione `strerror()`.

Output:

```
$ ./programma pippo.txt # Inesistente
./programma: Impossibile aprire pippo.txt: No such file or directory
```

La posizione di lettura/scrittura

Lettura e scrittura di un file avvengono in maniera sequenziale, dall'inizio alla fine, ma è possibile spostarsi arbitrariamente con la funzione `fseek()`:

```
int fseek(FILE *file, long offset, int whence);
```

La funzione la posizione attuale a `offset` byte di distanza dalla posizione indicata da whence, che può essere:

- `SEEK_SET`: Inizio del file
- `SEEK_CUR`: Posizione attuale
- `SEEK_END`: Fine del file

Ad esempio, per tornare all'inizio del file:

```
fseek(file, 0, SEEK_SET);
```

La funzione `ftell()` restituisce la posizione attuale:

```
int ftell(FILE *file);
```

La posizione di lettura/scrittura

Esempio: conoscere a priori la lunghezza di un file

```c
#include <stdio.h>
#include <errno.h>
#include <string.h>

int main(int argc, char **argv)
{
  char *filename = argv[1];
  FILE *file = fopen(filename, "r");
  if(!file) {
    fprintf(stderr, "%s: Impossibile aprire il file %s: %s",
            argv[0], filename, strerror(errno));
    return 2;
  }

  fseek(file, 0, SEEK_END);
  long bytes = ftell(file);

  printf("Il file e' lungo %ld bytes\n", bytes);

  fclose(file);

  return 0;
}
```

Input/Output binario

La lettura e scrittura di dati testuali non è l'unico modo di accesso ai file. Specificando `"rb"` o `"wb"` come parametro `mode` di `fopen()` è possibile effettuare Input/Output di dati **binari**.

- I dati vengono scritti e letti senza intermediazioni (es. traduzioni di codifica dei caratteri). Per l'I/O binario si usano funzioni apposite.
- La lettura si effettua tramite la funzione `fread()`:

```
size_t fread(void *ptr, size_t size, size_t nitems, FILE *file);
```

  Legge `size * nitems` byte dal file e scrive nella memoria puntata da `ptr`.

- La scrittura si effettua tramite la funzione `fwrite()`:

```
size_t fwrite(void *ptr, size_t size, size_t nitems, FILE *file);
```

  Scrive sul file `size * nitems` byte dalla memoria puntata da `ptr`.

Accesso ai file (funzioni POSIX)

Funzioni POSIX per l'accesso ai file

Mentre le funzioni ISO C sono specificate dallo standard del linguaggio, le funzioni più di basso livello fornite dallo standard POSIX sono specifiche dei sistemi UNIX.

Non sono un'interfaccia completamente ridondante:

- le funzioni POSIX permettono di scrivere e leggere da fonti diverse da file regolari: pipe, socket, ecc...
- sui sistemi Unix, le funzioni POSIX sono usate per implementare le funzioni ISO. Queste ultime adottano un **buffer** interno, mentre le funzioni POSIX effettuano direttamente le relative system call.
- le funzioni POSIX permettono di gestire i **permessi** dei file, i link, e altri attributi dei file specifici di UNIX.

System call POSIX per l'accesso ai file

```
#include <unistd.h>
```

Funzione	Scopo
`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 la posizione di lettura/scrittura
`unlink()`	rimuove un file
`fcntl()`	controlla alcuni attributi associati ad un file
`chmod()`	cambia i permessi di un file

Aprire un file

La funzione `open()` serve ad aprire un file:

```
int open(const char *path, int openflags);
```

Apre il file `path`, nel modo specificato da `openflags`, che può essere una **combinazione** dei seguenti **flag**:

Funzione	Scopo
`O_RDONLY`	open for reading only
`O_WRONLY`	open for writing only
`O_RDWR`	open for reading and writing
`O_APPEND`	append on each write
`O_CREAT`	create file if it does not exist
`O_TRUNC`	truncate size to 0
`O_EXCL`	error if O_CREAT and the file exists

Restituisce un **file descriptor** che rappresenta il file aperto.

Aprire un file

Esempio

```c
#include <unistd.h>

int main(int argc, char **argv)
{
  // Apriamo in scrittura, appendendo, creando il file se non esite
  int fd = open(argv[1], O_WRONLY | O_APPEND | O_CREAT);

  char contents[13] = "Hello World\n";

  write(fd, contents, 12);

  close(fd);

  return 0;
}
```

File descriptors

La funzione `open()` restituisce un numero intero chiamato **file descriptor**.

- Rappresenta il file aperto
- Non solo veri file possono essere associati a file descriptor, ma anche altro, come pipes, socket, ecc...
- In Unix, "tutto è un file" è un motto ricorrente, quindi i file descriptor compaiono dappertutto.
- Tre file descriptor sono già aperti all'inizio del programma: lo standard input (0), standard output (1) e standard error(2):

```
write(1, "Hello world\n", 12);
```

Lettura e scrittura

Lettura e scrittura avvengono tramite le funzioni `read()` e `write()`, usate in modo analogo a `fread()` e `fwrite()`.

```
ssize_t read(int fd, void *buffer, size_t nbytes);
```

Legge `nbytes` byte dal file descriptor `fd`, li scrive sulla memoria puntata da `buffer`, e restituisce il numero di byte letti, o -1 se avviene un errore.

```
ssize_t write(int fd, void *buffer, size_t nbytes);
```

Scrive `nbytes` byte dalla memoria puntata da `buffer` sul file descriptor `fd`, e restituisce il numero di byte scritti, o -1 se avviene un errore.

Le chiamate `stat` e `fstat`

Le chiamate di sistema `stat` e `fstat` permettono di accedere in lettura alle informazioni e proprietà associate ad un 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):

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

int stat(const char *pathname, struct stat *out);
int fstat(int filedes, struct stat *out);
```

La funzione `stat()` prende come primo argomento un pathname, mentre `fstat()` opera su un descrittore di un file già aperto.

Il risultato viene scritto in una struttura di tipo `struct stat` puntata dal parametro `out`

La struttura stat

stat è una struttura definita in <sys/stat.h> che definita in questo modo:

```
struct stat {
  dev_t st_dev;     // device id
  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;    // device type
  off_t st_size;    // dimensione del file
  time_t st_atime;  // tempo di ultimo accesso
  time_t st_mtime;  // tempo di ultima modifica
  time_t st_ctime;  // tempo di creazione
  long st_blksize;  // dimensione del blocco
  long st_blocks;   // numero di blocchi
};
```

La funzione `stat()`

Esempio

Il programma `lookout.c`, data una lista di nomi di file, controlla ogni 5 secondi se un file è stato modificato. Nel caso ciò avvenga termina l'esecuzione stampando un messaggio che informa l'utente dell'evento.

La funzione `stat()`

Esempio

La chiamata a `stat` ci dà informazioni sul file.

```
struct stat sb;
if(stat(file, &sb) == -1) {
  fprintf(stderr, "%s: errore nell'accesso al file %s: %s\n",
          argv[0], file, strerror(errno));
  return 1;
}

time_t mtime = sb.st_mtime;
```

La funzione `stat()`

Esempio

Ogni 5 secondi confrontiamo il valore di `st_mtime` con quello che avevamo salvato.

```
while(1) {
  if(stat(file, &sb) == -1) {
    fprintf(stderr, "%s: errore nell'accesso al file %s: %s\n",
            argv[0], file, strerror(errno));
    return 1;
  }

  if(sb.st_mtime != mtime) {
    printf("Il file %s e' stato modificato\n", file);
    return 0;
  }
  sleep(5);
}
```

Esercizi

Esercizi

Per venerdì 2 dicembre

1. Si scriva un programma che legga in modalità binaria dieci numeri interi dal file `/dev/random`, e li stampi sullo schermo, utilizzando funzioni ISO per l'accesso ai file.
2. Si riscriva lo stesso programma utilizzando funzioni POSIX per l'accesso ai file.

Esercizi (2)

Per venerdì 2 dicembre

3. Si scriva un programma che conti le modifiche ad un file (specificato come primo argomento sulla riga di comando) nell'arco di un intervallo di tempo specificato in secondi come secondo argomento sulla linea di comando.
   Suggerimento: la chiamata `time(NULL)` restituisce un valore dello stesso tipo di `stat->st_time`. Si tratta del numero di secondi strascorsi dalla mezzanotte di capodanno del 1970.