Files, File I/O

Franco Maria Nardini

UNIX file types

- UNIX non richiede una struttura *interna* del file. Dal punto di vista del sistema operativo c'è un solo tipo di file.
- Struttura e interpretazione sono demandati al software.
- Alcuni file *speciali* esistono: directory

UNIX file types

- Tipi di file
  - *regular file*: file comune, listati con un “-“ davanti
  - *directory*: file speciale più comune
  - *symbolic link*: reference ad un altro file. ricorda nulla? :)
  - *named pipe*: comunicazioni inter-processo (più avanti)
  - *socket*: comunicazioni inter-processo (più avanti)
  - *device file*: tutto è un file, anche un disco (dischi possono essere entrambe)
    - *device a caratteri*: unbuffered, accesso diretto, blocchi fixed size, allineati
    - *device a blocchi*: buffered, blocchi any size, non allineati
  - *door*: comunicazioni inter-processo (client-server)

File Descriptors

- Un *file descriptor* (o *file handle*) è un intero non negativo e piccolo che identifica un file al kernel.

- `stdin` è 0, `stdout` è 1, `stderr` è 2

- Evitare l'uso di numeri magici!

- POSIX-compliant constants: `STDIN_FILENO`, `STDOUT_FILENO`, `STDERR_FILENO`

- *File descriptor table* per ogni processo.

Standard I/O

- Unbuffered I/O: fatto tutto con cinque funzioni: **`open(2)`**, **`close(2)`**, **`lseek(2)`**, **`read(2)`**, **`write(2)`**.
  - system call al kernel.
    - no ISO C, POSIX.1 and Single UNIX Specs.
- I processi possono voler condividere risorse!
  - condivisione file tramite *file descriptors*
  - *atomicità delle operazioni*

In origine…

- c'era **`creat(2)`**

```
#include <fcntl.h>

int creat(const char *pathname, mode_t mode);
```

Returns: file descriptor if OK, -1 on error

Ora...

- `creat(2)` deprecata da `open(2)`

```
#include <fcntl.h>

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

Returns: file descriptor if OK, -1 on error

In dettaglio…

- *oflag* deve essere uno (e solo uno) tra:
  - O_RDONLY
  - O_WRONLY
  - O_RDWR
- assieme possono essere usati (in OR):
  - O_APPEND: per appendere
  - O_CREAT: crea il file se non esiste
  - O_EXCL: genera errore se O_CREAT e il file esiste (atomica)
  - O_TRUNC: se file esiste e aperto correttamente in O_WRONLY e O_RDWR, setta lunghezza a 0.

Varianti di `open(2)`

```
#include <fcntl.h>

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

Returns: file descriptor if OK, -1 on error

- **`openat()`** è usata per gestire pathname da differenti *working directory*.

close(2)

```
#include <unistd.h>

int close(int fd);
```

Returns: 0 if OK, -1 on error

- la chiusura rilascia la *lock* su un *file descriptor*. (dettagli più avanti).

- il kernel ci aiuta! *file descriptors* non esplicitamente chiusi vengono chiusi alla terminazione del processo.

    - evitare “leaks”, gestire la chiusura appropriatamente (visibilità).

Esempio

```
$ wget http://hpc.isti.cnr.it/~nardini/siselab/02/openex.c

$ more openex.c

$ cc -Wall -o myopen openex.c
```

read(2)

```
#include <unistd.h>

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

Returns: number of bytes read, 0 if end of file, -1 on error

- Dichiarazione?

- **`read()`** inizia a leggere dall'offset corrente e incrementa fino a *nbytes*
- Alcuni casi di lettura minore dei bytes richiesti:
    - EOF incontrato prima delle *nbytes* letture
    - Letture da terminale una riga alla volta
    - Letture da rete (ritardi dovuti a buffering)
    - Arriva un segnale

write(2)

```
#include <unistd.h>

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

Returns: number of bytes written if OK, -1 on error

- Dichiarazione?

- **`write()`** ritorna *nbytes* oppure si è verificato un errore

- per i *regular file*: **`write()`** inizia a scrivere dall'offset corrente. eccezioni? :)

  - O_APPEND! qual'è l'offset in questo caso?

- alla fine, l'offset è aggiornato di *nbytes*

Esempio

```
$ wget http://hpc.isti.cnr.it/~nardini/siselab/02/rwex.c

$ more rwex.c

$ cc -Wall -o myrw rwex.c
```

lseek(2)

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

off_t lseek(int filedes, off_t offset, int whence );
```

Returns: new file offset if OK, -1 on error

lseek(2)

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

off_t lseek(int filedes, off_t offset, int whence );
```

Returns: new file offset if OK, -1 on error

- *whence* determina come l'offset e' usato
  - SEEK_SET bytes dall'inizio del file
  - SEEK_CUR bytes dalla posizione corrente
  - SEEK_END bytes dalla fine del file

lseek(2)

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

off_t lseek(int filedes, off_t offset, int whence );
```

Returns: new file offset if OK, -1 on error

- tutto ciò comporta:
  - **`lseek`** di un offset negativo
  - **`lseek`** di 0 bytes dalla posizione corrente
  - **`lseek`** superando la fine del file
- Non tutto è seekabile: pipe, FIFO, socket

Esempio

```
$ wget http://hpc.isti.cnr.it/~nardini/siselab/02/hole.c
```

```
$ more hole.c
```

```
$ cc -Wall -o myhole ./hole.c
```

```
$ ./myhole
```

Homework - 1

- Scrivere un programma C che, dato il file dei primi 100 numeri, lo legge, ordina i numeri in senso crescente e li stampa a video ordinati.
  - usa stdin, stdout con **unbuffered I/O**.

```
$ wget http://hpc.isti.cnr.it/~nardini/siselab_anl/01/primi100numeri.txt
```

Esempio

```
$ wget http://hpc.isti.cnr.it/~nardini/siselab/02/lseek.c

$ more lseek.c

$ cc -Wall -o myseek ./lseek.c

$ ./myseek < ./lseek.c

seek OK

$ cat ./lseek.c | ./myseek

cannot seek
```