LabSO

errors, fd, pipe, fifo

Errors: errno, strerror, perror

- `errno` **è una variabile globale che contiene il codice dell'ultimo errore “di sistema”**
- `strerror` **è una funzione che restituisce la “stringa” (in senso del “C”) dell'ultimo errore “di sistema” dato il codice passato come parametro (solitamente** `errno`**)**
- `perror` **è una funzione che stampa la stringa passata come parametro poi “: “ (due punti e spazio) e poi accoda il testo del messaggio dell'ultimo errore “di sistema”**

Errors: errno, strerror, perror

- **esempio:**

```
#include <stdio.h>
#include <errno.h>
#include <string.h>
extern int errno;
int main () {
   FILE * pf;
   int errnum;
   pf = fopen ("filechenonesiste.boh", "rb");
   if (pf == NULL) {
      errnum = errno;
      fprintf(stderr, "errno = %d\n", errno);
      perror("Error printed by perror");
      fprintf(stderr, "Error opening file: %s\n", strerror( errnum ));
   } else {
      fclose (pf);
   };
   return 0;
};
```

fd

- **gestione dei file tramite file descriptors:**
  - open: crea *fd* per accedere a un file
  - close: libera le risorse di un *fd*
  - read: legge da un file tramite *fd*
  - write: scrive su un file tramite *fd*
  - lseek: si posiziona in un file tramite *fd*
  - dup/dup2: duplica un *fd* (l'indice)

fd: open

- **file descriptors - open**

```
#include <fcntl.h>

#include <sys/types.h>

#include <sys/stat.h>

int open (const char* Path, int flags [, int
mode ]);
```

- **Path è il riferimento al file (assoluto se inizia con “/”)**
- **flags: `O_RDONLY` (sola lettura), `O_WRONLY` (sola scrittura), `O_RDWR` (lettura+scrittura), `O_CREAT` (crea se non esiste), `O_EXCL` (non sovrascrivere se esiste)**
- **mode: permessi (v. tabella accanto)**

```
S_IRUSR   read rights for owner
S_IWUSR   write rights for owner
S_IXUSR   exec rights for owner
S_IRGRP   read rights for group
S_IWGRP   write rights for group
S_IXGRP   exec rights for group
S_IROTH   read rights for others
S_IWOTH   write rights for others
S_IXOTH   exec rights for others
```

fd: close

- **file descriptors - close**

```
#include <fcntl.h>

int open (int fd);
```

- ***fd* è il file descriptor da “chiudere”**

fd: open/close

Esempio (chiamare con `./main` poi con `./main nomefilenonesistente` poi con `./main nomefileesistente`):

```
#include <stdlib.h>
#include <stdio.h>
#include <fcntl.h>

int main(int argc, char *argv[]) {
    int fd = open(argv[1], O_RDONLY);
    if (fd < 0) {
        perror("?");
        exit(1);
    };
    printf("opened fd = % d\n", fd);
    if (close(fd) < 0) {
        perror("?");
        exit(1);
    };
    printf("closed fd.\n");
}
```

fd: open/close

Esercizio: impostare un programma che risponda a una segnatura di parametri del tipo:

```
[--exists | --notexists] <nomefile>
```

Il programma verifica l'esistenza del file `<nomefile>`: restituisce `0` se esiste e si è passato `--exists` come primo argomento, `1` se non esiste. Passando `--notexists` i valori sono invertiti.

fd: read/write

- **`#include <fcntl.h>`**
  `size_t read (int fd, void* buf, size_t cnt);`
  - legge al file puntato da `fd` mettendo fino a `cnt` bytes in `buf` e restituisce il numero di bytes effettivamente letti (<= `cnt`) in caso di successo.
  - restituisce 0 se il file è alla fine, -1 in caso d'errore
- **`#include <fcntl.h>`**
  `size_t write (int fd, void* buf, size_t cnt);`
  - scrive sul file puntato da `fd`, `cnt` bytes a partire dalla zona di memoria definita da `buf`
  - restituisce il numero di bytes scritti in caso di successo (= `cnt` necessariamente), quelli effettivamente scritti o 0 in casi particolari (spazio ridotto, fine file), -1 in caso d'errore generico

fd: read/write

- Esempio

```
#include<stdio.h>
#include<string.h>
#include<unistd.h>
#include<fcntl.h>
int main (void) {
    int fd[2];
    char buf1[12] = "hello world";
    char buf2[12];
    fd[0] = open("foobar.txt", O_WRONLY | O_CREAT, 0777);
    fd[1] = open("foobar.txt", O_RDONLY);
    printf("fd[0]=%d, fd[1]=%d\n", fd[0], fd[1]);
    write(fd[0], buf1, strlen(buf1));
    write(1, buf2, read(fd[1], buf2, 12));
    close(fd[0]);
    close(fd[1]);
    return 0;
}
```

fd: lseek

- **`#include <sys/types.h>`**
  `#include <unistd.h>`
  `off_t lseek(int fd, off_t offset, int whence);`
- **sposta il cursore dentro il file puntato da `fd` alla posizione `offset`:**
  - contanto dall'inizio se `whence=SEEK_SET`
  - relativamente alla posizione attuale se `whence=SEEK_CUR`
  - contanto dalla fine se `whence=SEEK_END`
- **in tutti i casi restituisce la posizione del cursore dall'inizio del file o -1 in caso d'errore**

fd: lseek

- **Esercizio: scrivere un programma che data una lista di file (uno o più) per ciascuno visualizza il nome e la dimensione in bytes (suggerimento: utilizzare un ciclo sugli argomenti e la funzione *lseek*)**

fd: dup/dup2

- ```
  #include <unistd.h>
  int dup(int oldfd);
  int dup2(int oldfd, int newfd);
  ```
- **le funzioni `dup` e `dup2` creano una “copia” del file descriptor `oldfd`, in modo che quello nuovo – che viene creato nel caso di `dup` come il più piccolo indice disponibile o che è `newfd` nel caso di `dup2` – punti allo stesso file. Entrambe restituiscono il “nuovo” descrittore o -1 in caso d'errore**
- **in pratica il nuovo *fd* punterà al file cui puntava il vecchio *fd***
  ```

fd: dup/dup2

- Esempio di redirezionamento

```
#include <stdlib.h>
#include <stdio.h>
#include <fcntl.h>
#include <unistd.h>
int main(int argc, char* argv[]) {
  int fdlog;
  char *cmd;
  char *args[]={NULL, NULL, NULL, NULL, NULL, NULL}; // one more to hold "cmd" too
  if (argc < 3 || argc > 7) {
    printf("?ERROR. Usage: %s <logfile> <cmd> [up to 5 <args>]\n", argv[0]);
    exit(1);
  };
  int a=0;
  printf("logfile=%s\n", argv[1]);
  printf("Command=%s\n", argv[2]);
  fdlog = open(argv[1], O_WRONLY|O_CREAT, 0777);
  dup2(fdlog, STDOUT_FILENO): dup2(fdlog, STDERR_FILENO);
  cmd=argv[2];
  args[0]=cmd;
  for (a=3; a<=argc; a++) { args[a-2]=argv[a]; };
  execvp(cmd, args);
  close(fdlog);
}
```

pipe/fifo

- **Il piping connette output di un comando a input di un altro, es.:**

```
who | sort
ls | more
```

- **I processi sono eseguiti in concorrenza**
- **si utilizza un buffer:**
  - se pieno lo scrittore si sospende fino ad avere spazio libero
  - se vuoto il lettore si sospende fino ad avere dati

pipe/fifo

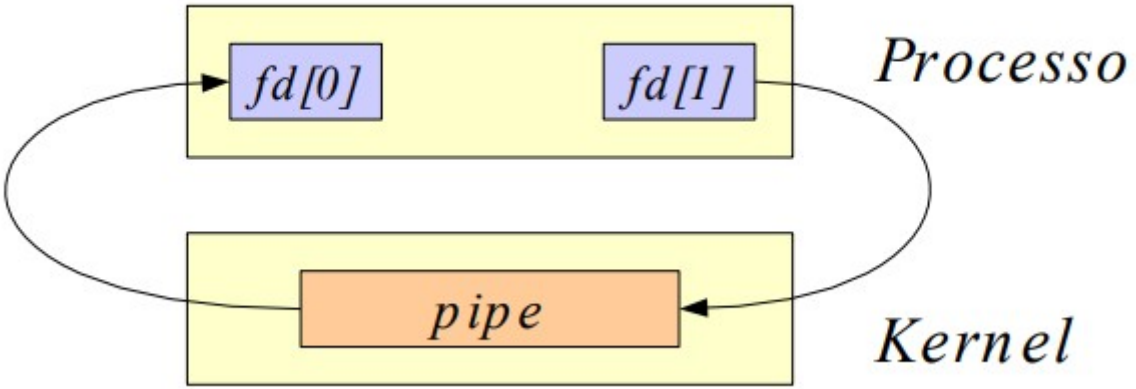
- pipe anonime (es. da shell)
- pipe con nome o “FIFO” (su file)

pipe/fifo

- **pipe anonima:**
- **“unisce” due processi (si usa per questo: si può usare solo tra processi con un antenato comune o direttamente collegati padre-figlio)**
- **è unidirezionale**
- **utilizza un buffer**
- **ha due “lati” di accesso:**
  - lato scrittura (si accede con `write()`)
  - lato lettura (si accede con `read()`)
  - si chiudono con `close()`

pipe/fifo

- **syscall:** `int pipe (int fd[2])`
- **imposta <u>due</u> file descriptor :**
  - lato lettura fd[0]
  - lato scrittura fd[1]
- **restituisce 0 se va a buon fine, -1 altrimenti**



pipe/fifo

- **leggendo con `read(fd[0], data, num)`:**
- **se l'altro lato è stato chiuso restituisce `0`**
- **se il buffer è vuoto e l'altro lato è ancora aperto il processo si sospende fino alla disponibilità dei dati o alla chiusura**
- **se si provano a leggere più bytes (`num`) di quelli disponibili sono recuperati solo quelli presenti (v. nota seguente)**
- **in caso di successo `read()` restituisce il numero di bytes effettivamente letti**

pipe/fifo

- **scrivendo con `write(fd[1], data, num)`:**
- **se l'altro lato è stato chiuso fallisce e invia un SIGPIPE allo scrittore stesso (di default lo chiude)**
- **se scrive:**
  - meno bytes di quelli che ci possono stare la scrittura è “atomica” (tutto insieme)
  - più bytes di quelli che ci possono stare non c'è garanzia di atomicità

pipe/fifo

- **tipica sequenza unidirezionale processo P1 scrittore, P2 lettore:**
  - P1 crea un `pipe()`
  - P1 esegue un `fork()` e crea P2
  - P1 chiude il lato lettura: `close(fd[0])`
  - P2 chiude il lato scrittura: `close(fd[1])`
  - P1 e P2 chiudono l'altro fd appena finito

pipe/fifo

- **tipica sequenza bidirezionale processo P1 scrittore, P2 lettore:**
  - P1 crea due `pipe()`
  - P1 esegue un `fork()` e crea P2
  - P1 chiude il lato lettura di una pipe e quello scrittura dell'altra: `close(fd[0])`
  - P2 si comporta simmetricamente
  - P1 e P2 chiudono gli altri fd appena finito

pipe/fifo

esempio unidirezionale:

```
#include <stdio.h>
#include <string.h>
#define READ 0 /* read-side of pipes */
#define WRITE 1 /* write-side of pipes */
#define MAXLEN 100 /* max length of message */
char *msg = "Content of pipe.";
int main (void) {
  int fd [2], bytesRead;
  char message [MAXLEN];
  pipe (fd); /* Create unnamed pipe */
  if (fork () > 0) { /* Parent, writer */
    close(fd[READ]); /* close other side */
    write (fd[WRITE], msg, strlen(msg)+1); /* include \0 */
    close (fd[WRITE]); /* close this side */
  } else { /* Child, reader*/
    close (fd[WRITE]); /* close other side */
    bytesRead = read (fd[READ], message, MAXLEN);
    printf ("Read %d bytes: %s\n", bytesRead, message);
    close (fd[READ]); /* close this side */
  }
}
```

pipe/fifo

Esercizio 1: impostare una comunicazione bidirezionale tra due processi con due livelli di complessità:

- alternando almeno due scambi (P1 → P2, P2 → P1, P1 → P2, P2 → P1)
- estendendo il caso a mo' di “ping-pong” fino a un messaggio convenzionale di “fine comunicazione”

pipe/fifo

- Occorre definire un “protocollo”: tipicamente avendo una lunghezza fissa nei messaggi o con un marcatore di fine messaggio, ad esempio:
  - Inviare prima la lunghezza e poi il messaggio
  - Terminare ogni messaggio con un carattere speciale (es. `NULL` o `newline`)
- Più in generale occorre definire la sequenza di messaggi attesi (scambio informazioni)

pipe/fifo

- Esempio (redirezione con piping)

```
#include <stdio.h>
#define READ 0
#define WRITE 1
/* redirige lo stdout di cmd1 sullo stdin di cmd2 */
int main (int argc, char *argv []) {
    int fd [2];
    pipe (fd); /* Create an unamed pipe */
    if (fork () != 0) { /* Parent, writer */
        close (fd[READ]); /* Close unused end */
        dup2 (fd[WRITE], 1); /* Duplicate used end to stdout */
        close (fd[WRITE]); /* Close original used end */
        execlp (argv[1], argv[1], NULL); /* Execute writer program */
        perror ("connect"); /* Should never execute */
    } else { /* Child, reader */
        close (fd[WRITE]); /* Close unused end */
        dup2 (fd[READ], 0); /* Duplicate used end to stdin */
        close (fd[READ]); /* Close original used end */
        execlp (argv[2], argv[2], NULL); /* Execute reader program */
        perror ("connect"); /* Should never execute */
    }
}
```

pipe/fifo

- pipe con nome o FIFO
- sono gestiti con file speciali
- non ci sono vincoli di gerarchia tra i processi: per usarli basta (avendo i permessi) conoscerne il nome
- sono persistenti (fino a che siano rimossi)
- ```
  #include <sys/types.h>
  #include <sys/stat.h>
  int mkfifo(const char *pathname, mode_t mode);
  ```
- essendo oggetti nel file-system si possono usare le funzioni “normali” per i files
  ```

pipe/fifo

- Esempio “chat” writer/reader con FIFO

```
// WRITER
#include <stdio.h>
#include <string.h>
#include <fcntl.h>
#include <sys/stat.h>
#include <sys/types.h>
#include <unistd.h>
int main() {
    int fd;
    char *myfifo = "/tmp/myfifo";
    mkfifo(myfifo, 0666); // mkfifo(<pathname>, <permission>)
    char str1[80], str2[80];
    while (1) {
        fd = open(myfifo, O_WRONLY);     // Open FIFO for write only
        fgets(str2, 80, stdin);          // input from user, maxlen=80
        write(fd, str2, strlen(str2)+1); // write and close
        close(fd);
        fd = open(myfifo, O_RDONLY);     // Open FIFO for Read only
        read(fd, str1, sizeof(str1));    // Read from FIFO
        printf("User2: %s\n", str1);     // print message
        close(fd);
    };
    return 0;
}
```

```
// READER
#include <stdio.h>
#include <string.h>
#include <fcntl.h>
#include <sys/stat.h>
#include <sys/types.h>
#include <unistd.h>
int main() {
    int fd;
    char *myfifo = "/tmp/myfifo";
    mkfifo(myfifo, 0666); // mkfifo(<pathname>, <permission>)
    char str1[80], str2[80];
    while (1) {
          fd = open(myfifo, O_RDONLY); // Open FIFO for read only
        read(fd, str1, 80);          // read from FIFO
        printf("User1: %s\n", str1); // write and close
                close(fd);
        fd = open(myfifo, O_WRONLY); // Open FIFO for write only
        fgets(str2, 80, stdin);      // input from user, maxlen=80
        write(fd, str2, strlen(str2)+1);
        close(fd);
    };
    return 0;
};
```

pipe/fifo

- Esercizio: modificare l'esempio precedente applicando una o più delle seguenti varianti:
  - definire un “prompt” di input
  - consentire l'immissione “asincrona”
  - interpretare alcuni comandi se l'input inizia con un carattere speciale (ad esempio con “`/`”, per esempio: `/quit` per chiudere il processo)