## LABORATORIO SISTEMI OPERATIVI: PIPING

- **ERROR**:
  - **Librerie**: errno.h e string.h
  - **errno**:
    - **Variabile d'errore globale** che **contiene l'ultimo errore** che si è verificato (inteso come **valore numerico**).
    - Eseguendo una syscall, questa **può generare un errore** o restituire un valore d'uscita; questo valore di ritorno viene salvato nella variabile **errno**.
    - Ogni syscall eseguita, salva il suo valore di ritorno nella variabile **errno**, **sovrascrivendo quello precedentemente salvato**.

  - **strerror(int errno)**:
    - Funzione che prende come **argomento un codice d'errore** e **ritorna una stringa** (vettore di char) **descrittiva che riguarda l'errore** che si è generato.
    - Viene usata per **stampare a video un messaggio d'errore testuale associato all'errore** numero passato come parametro.
    - Partendo da un codice numerico, riusciamo a stampare un messaggio testuale d'errore.
    - La **lista di messaggi** è disponibile all'interno del **vettore globale sys_errlist** (si accede via indice al messaggio contenuto).

  - **perror(char *msg)**:
    - Funzione che **accetta come argomento una stringa** e **stampa a video**:
      **msg + ":" + <messaggio d'errore testuale>**
    - **Errore stampato è equivalente a quello che si ottiene con strerror()**.

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

int main(){
  int fp=open("pippo", "r"); //apro un file che non esiste
  printf("BEGIN\n\n");
  printf("ERRNO + STRERROR:\n");
  printf("Last error: %d, %s\n", errno, strerror(errno));
  printf("\n");
  printf("PERROR:\n");
  perror("?Error"); //di default viene aggiunto un 'a capo'
  printf("\nEND\n\n");
  return 0;
}
```

```
root@LABSO:/tmp/error# ./err
BEGIN

ERRNO+$STRETTOR:
Last error: 22, Invalid argument

PERROR:
?Error: Invalid argument

END

root@LABSO:/tmp/error#
```

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

int main(){
  int fp = open("pippo1.txt", "r"); //apro un file che non esiste
  printf("BEGIN\n\n");
  printf("ERRNO + STRERROR:\n");
  printf("Last error: %d, %s\n", errno, strerror(errno));
  printf("\n");
  printf("PERROR:\n");
  perror("?Error"); //di default viene aggiunto un 'a capo'
  printf("\nEND\n\n");
  return 0;
}
```

```
root@LABSO:/tmp/error# ./err
BEGIN

ERRNO + STRERROR:
Last error: 2, No such file or directory

PERROR:
?Error: No such file or directory

END

root@LABSO:/tmp/error#
```

## Valori di error:

| | | |
|---|---|---|
| 1 | EPERM | Operation not permitted |
| 2 | ENOENT | No such file or directory |
| 3 | ESRCH | No such process |
| 4 | EINTR | Interrupted system call |
| 5 | EIO | Input/output error |
| 6 | ENXIO | No such device or address |
| 7 | E2BIG | Argument list too long |
| 8 | ENOEXEC | Exec format error |
| 9 | EBADF | Bad file descriptor |
| 10 | ECHILD | No child processes |
| 11 | EAGAIN | Resource temporarily unavailable |
| 11 | EWOULDBLOCK | Resource temporarily unavailable |
| 12 | ENOMEM | Cannot allocate memory |
| 13 | EACCES | Permission denied |
| 14 | EFAULT | Bad address |
| 15 | ENOTBLK | Block device required |
| 16 | EBUSY | Device or resource busy |
| 17 | EEXIST | File exists |
| 18 | EXDEV | Invalid cross-device link |
| 19 | ENODEV | No such device |
| 20 | ENOTDIR | Not a directory |
| 21 | EISDIR | Is a directory |
| 22 | EINVAL | Invalid argument |
| 23 | ENFILE | Too many open files in system |
| 24 | EMFILE | Too many open files |
| 25 | ENOTTY | Inappropriate ioctl for device |
| 26 | ETXTBSY | Text file busy |
| 27 | EFBIG | File too large |
| 28 | ENOSPC | No space left on device |
| 29 | ESPIPE | Illegal seek |
| 30 | EROFS | Read-only file system |
| 31 | EMLINK | Too many links |
| 32 | EPIPE | Broken pipe |
| 33 | EDOM | Numerical argument out of domain |
| 34 | ERANGE | Numerical result out of range |
| 35 | EDEADLK | Resource deadlock avoided |
| 35 | EDEADLOCK | Resource deadlock avoided |
| 36 | ENAMETOOLONG | File name too long |
| 37 | ENOLCK | No locks available |
| 38 | ENOSYS | Function not implemented |
| 39 | ENOTEMPTY | Directory not empty |
| 40 | ELOOP | Too many levels of symbolic links |
| 42 | ENOMSG | No message of desired type |
| 43 | EIDRM | Identifier removed |
| 44 | ECHRNG | Channel number out of range |
| 45 | EL2NSYNC | Level 2 not synchronized |
| 46 | EL3HLT | Level 3 halted |
| 47 | EL3RST | Level 3 reset |
| 48 | ELNRNG | Link number out of range |
| 49 | EUNATCH | Protocol driver not attached |
| 50 | ENOCSI | No CSI structure available |
| 51 | EL2HLT | Level 2 halted |
| 52 | EBADE | Invalid exchange |
| 53 | EBADR | Invalid request descriptor |
| 54 | EXFULL | Exchange full |
| 55 | ENOANO | No anode |
| 56 | EBADRQC | Invalid request code |
| 57 | EBADSLT | Invalid slot |
| 59 | EBFONT | Bad font file format |
| 60 | ENOSTR | Device not a stream |
| 61 | ENODATA | No data available |
| 62 | ETIME | Timer expired |
| 63 | ENOSR | Out of streams resources |
| 64 | ENONET | Machine is not on the network |
| 65 | ENOPKG | Package not installed |
| 66 | EREMOTE | Object is remote |
| 67 | ENOLINK | Link has been severed |
| 68 | EADV | Advertise error |
| 69 | ESRMNT | Srmount error |
| 70 | ECOMM | Communication error on send |
| 71 | EPROTO | Protocol error |
| 72 | EMULTIHOP | Multihop attempted |
| 73 | EDOTDOT | RFS specific error |
| 74 | EBADMSG | Bad message |
| 75 | EOVERFLOW | Value too large for defined data type |
| 76 | ENOTUNIQ | Name not unique on network |
| 77 | EBADFD | File descriptor in bad state |
| 78 | EREMCHG | Remote address changed |
| 79 | ELIBACC | Can not access a needed shared library |
| 80 | ELIBBAD | Accessing a corrupted shared library |
| 81 | ELIBSCN | .lib section in a.out corrupted |
| 82 | ELIBMAX | Attempting to link in too many shared libraries |
| 83 | ELIBEXEC | Cannot exec a shared library directly |
| 84 | EILSEQ | Invalid or incomplete multibyte or wide character |
| 85 | ERESTART | Interrupted system call should be restarted |
| 86 | ESTRPIPE | Streams pipe error |
| 87 | EUSERS | Too many users |
| 88 | ENOTSOCK | Socket operation on non-socket |
| 89 | EDESTADDRREQ | Destination address required |
| 90 | EMSGSIZE | Message too long |
| 91 | EPROTOTYPE | Protocol wrong type for socket |
| 92 | ENOPROTOOPT | Protocol not available |
| 93 | EPROTONOSUPPORT | Protocol not supported |
| 94 | ESOCKTNOSUPPORT | Socket type not supported |
| 95 | ENOTSUP | Operation not supported |
| 95 | EOPNOTSUPP | Operation not supported |
| 96 | EPFNOSUPPORT | Protocol family not supported |
| 97 | EAFNOSUPPORT | Address family not supported by protocol |
| 98 | EADDRINUSE | Address already in use |
| 99 | EADDRNOTAVAIL | Cannot assign requested address |
| 100 | ENETDOWN | Network is down |
| 101 | ENETUNREACH | Network is unreachable |
| 102 | ENETRESET | Network dropped connection on reset |
| 103 | ECONNABORTED | Software caused connection abort |
| 104 | ECONNRESET | Connection reset by peer |
| 105 | ENOBUFS | No buffer space available |
| 106 | EISCONN | Transport endpoint is already connected |
| 107 | ENOTCONN | Transport endpoint is not connected |
| 108 | ESHUTDOWN | Cannot send after transport endpoint shutdow |
| 109 | ETOOMANYREFS | Too many references: cannot splice |
| 110 | ETIMEDOUT | Connection timed out |
| 111 | ECONNREFUSED | Connection refused |
| 112 | EHOSTDOWN | Host is down |
| 113 | EHOSTUNREACH | No route to host |
| 114 | EALREADY | Operation already in progress |
| 115 | EINPROGRESS | Operation now in progress |
| 116 | ESTALE | Stale file handle |
| 117 | EUCLEAN | Structure needs cleaning |
| 118 | ENOTNAM | Not a XENIX named type file |
| 119 | ENAVAIL | No XENIX semaphores available |
| 120 | EISNAM | Is a named type file |
| 121 | EREMOTEIO | Remote I/O error |
| 122 | EDQUOT | Disk quota exceeded |
| 123 | ENOMEDIUM | No medium found |
| 124 | EMEDIUMTYPE | Wrong medium type |
| 125 | ECANCELED | Operation canceled |
| 126 | ENOKEY | Required key not available |
| 127 | EKEYEXPIRED | Key has expired |
| 128 | EKEYREVOKED | Key has been revoked |
| 129 | EKEYREJECTED | Key was rejected by service |
| 130 | EOWNERDEAD | Owner died |
| 131 | ENOTRECOVERABLE | State not recoverable |
| 132 | ERFKILL | Operation not possible due to RF-kill |
| 133 | EHWPOISON | Memory page has hardware error |

- **FILE DESCRIPTOR**:
  - **Operazioni base**:
    - **open/close/read/write**

  - **Operazioni avanzate**:
    - **lseek**:
      - Permette di **posizionare l'indice di lettura/scrittura** del file in un punto specificato.

    - **int dup(int oldfd)**:
      - Accetta come argomento un **unico parametro intero** che è un **riferimento ad un file descriptor**.
      - Una volta eseguita, **si ottiene un nuovo indice che punta allo stesso file** (avremo due indici nella tabella dei file descriptor che referenziano lo stesso file).

    - **Int dup2(int oldfd, int newfd)**:
      - Accetta come argomento **due interi**:
        - **oldfd** è il **riferimento al file descriptor del file che si vuole duplicare**.
        - **newfd** è il **nuovo riferimento che si vuole usare per riferirsi al file**.

      - Permette anche una **sovrascrittura**, newfd potrebbe essere un indice che referenzia un altro file, ma usando la funzione dup2(), questo viene sovrascritto con l'indice passato.

    - **dup()** e **dup2()** sono **funzioni che duplicano il file descritor** e sono utili quando si eseguono funzioni come exec e la fork, per far si che i nuovi processi ereditino queste modifiche alla tabella dei descrittori di file.

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

int main(int argc, char *argv[]){
  int fdlog; //descrittore file
  char *cmd;
  char *args[] = {NULL, NULL, NULL, NULL, NULL, NULL};

  //Ogni processo ha 3 canali (input, output, error)
  //Tramite la dup() possiamo alterare il comportamento di questi canali

  //param: nome del file, il file dove salvare i dati, il comando da eseguire e fino a 5 arg
  if(argc < 3 || argc > 7){
    printf("?.ERROR. Usate: %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]);

  //per eseguirlo: ./dup /tmp/log.txt ls -l /tmp/esempio
  fdlog = open(argv[1], O_WRONLY|O_CREAT, 0777); //apre il file in sola scrittura, se non esiste lo crea

  //modifico il canale 1, facendo si che il canale 1 non punti piu' al video, ma al file appena aperto
  dup2(fdlog, STDOUT_FILENO); //STDOUT_FILENO macro per il canale stdout (1)
  dup2(fdlog, STDERR_FILENO); //STDERR_FILENO macro per il canale error (3)
  //i canali 1 e 2 vengono modificati (rimango esistenti) ma semplicemente puntano a fdlog (file)

  cmd = argv[2];
  //argomenti da passare alla exec
  args[0] = cmd; //il primo e' il comando stesso

  //prendo gli argomenti passati da riga di terminale e li salvo in args
  for(a=3; a<argc; a++){
    args[a-2] = argv[a];
  };

  //sostituisce l'immagine del processo in esecuzione, con quello del comando che viene eseguito
  //la logica viene cambiata, ma eredita delle informazioni (come la tabella di file descriptor)
  execvp(cmd, args);
  close(fdlog);
  return 0;
}
```

- **PIPING**:
  - Esempio:
    - **ls | more**:
      - Questi due comandi vengono eseguiti su **due processi diversi in concorrenza**.
      - **Viene settato un buffer** tra i due processi:
        - **ls è un comando che genera dati** e di default questi dati vengono inviati nel buffer del canale stdout (canale 1) **ma tramite il comando di pipe '|'**, questi dati non vengono mandati nel canale standard ma **vengono inviati nel buffer che si è creato tra i due processi** (il suo canale d'uscita viene "dirottato").
        - Questo stesso **buffer è dato in lettura al comando more** che **prende i dati generati da ls in input**, gli elabora e a sua volta genera altri dati.
        - Questi **dati** però **vengono mandati** sul buffer standard del **canale stdout** (non ci sono altre indicazioni dopo il comando more).

      - Il questo **buffer** (che è stato settato tra i processi):
        - Se il **processo** che prova ad **inserire dati**, trova il **buffer pieno**, **interrompe la sua esecuzione e aspetta che si liberi spazio** prima di andare avanti.
        - Se il **processo** che prova a **leggere i dati**, trova il **buffer vuoto**, **interrompe la sua esecuzione e aspetta che vengano generati i dati**.

      - È presente una sorta di **sincronizzazione** tra i processi.

  - **PIPING ANONIMO**:
    - Vengono generati dei buffer che non hanno un nome, possono essere riferiti solo dai processi che stanno interagendo in quel momento (come nell'esempio precedente).
    - Collega due processi che hanno un antenato in comune (nell'esempio precedente, l'antenato in comune è la shell).
    - Al buffer sono associati **2 lati**:
      - **Lato di scrittura** (ls scrive nel buffer).
      - **Lato di lettura** (more legge i dati).

    - La **comunicazione** standard è di tipo **unidirezionale**:
      - **P1 crea la pipe** eseguendo la chiamata di sistema (genera due descrittori).
      - **P1 esegue una fork** che genera un **nuovo processo P2** che, **ereditando** la **tabella dei descrittori di file**, **eredita** anche i **riferimenti** di questi due descrittori particolari che sono collegati tra loro.
      - **P1 chiude uno dei due descrittori** (lato scrittura o lettura) con la close().
      - **P2 chiude l'altro descrittore**.
      - Ora **P1 e P2 condividono il buffer**, uno in scrittura e uno in lettura.
      - Quando hanno terminato la comunicazione, chiudono i restanti descrittori.

    - Se si volesse avere una **comunicazione bidirezionale**:
      - **P1 crea 2 pipe** (otteniamo 4 descrittori)
      - **P1 esegue una fork** che genera un nuovo processo P2, che ereditando la tabella dei descrittori di file, eredita anche i riferimenti di questi quattro descrittori.
      - **P1 chiude** da una pipe il **lato di scrittura** e **dall'altra** pipe **quello** di **lettura**.
      - **P2 simmetricamente esegue la stessa operazione** chiudendo gli altri lati.
      - In questo modo **P1 e P2 possono entrambi leggere e scrivere usando** però 2 **buffer diversi**.
      - Quando hanno terminato la comunicazione, chiudono i descrittori.

- Tramite la chiamata di sistema **pipe(…) è possibile creare il buffer** di comunicazione:
  **int pipe(int fd[2]);** //funzione si aspetta un vettore di due interi

  - Con la chiamata **pipe(fd)**, **fd** che è un **vettore di due interi**, **viene inizializzato** con due valori (**due indici della tabella del descrittore di file**) che rappresentano:
    - **fd[0]**: lato **lettura**
    - **fd[1]**: lato **scrittura**

- Anziché creare con la open(…) dei canali di comunicazione sul file system (creazione e uso di file per la comunicazione), **possiamo utilizzare la funzione pipe(…) per creare due descrittori di file** (generati in contemporanea dal sistema) **collegati tra loro** che **fanno riferimento alle due estremità del buffer** che viene creato.
- È come se avessimo **creato due riferimenti ad un file** con la particolarità che questi due riferimenti, **puntano allo stesso contenuto** (buffer) **su due punti diversi** (i riferimenti rappresentano il lato di scrittura o di lettura).

- PER LA LETTURA:
  - **read(fd[0], data, num)**:
    - **fd[0]**: **descrittore** del **lato della lettura** dal buffer.
    - **data**: variabile dove **salvare i dati** letti.
    - **num**: **quantità** di **dati da leggere** dal buffer.

  - Se il **lato di scrittura è stato chiuso**, la funzione **restituisce 0**.
  - Se il **lato di scrittura è aperto** ma il **buffer è vuoto**, la **read(…)** aspetta i dati **sospendendo il processo** (o lo termina).
  - Se il **lato di scrittura è aperto** e **il buffer contiene dati**, **leggo i dati disponibili** (se voglio leggere una quantità di dati maggiore rispetto a quella presente del buffer non ho errori, ma possono rilevarlo in quanto la funzione **read(…) restituisce il numero di elementi letti**).

- PER LA SCRITTURA:
  - **write(fd[1], data, num)**:
    - **fd[1]**: **descrittore** del **lato di scrittura** sul buffer.
    - **data**: variabile dove **salvare i dati** letti.
    - **num**: **quantità** di **dati da leggere** dal buffer.

  - Se il **lato di lettura è stato** chiuso, la funzione **write(…) fallisce e invia un segnale di SIGPIPE al processo stesso** (operazione che di default termina il processo, ma possiamo sovrascrivere l'handler di questo segnale).
  - Se il **lato di lettura è aperto** ma il **buffer è pieno**, la **funzione sospende il processo aspettando che si liberi spazio** nel buffer.
  - Se il **lato di lettura è aperto** e il **buffer non è pieno**, **posso scrivere i dati** (se la dimensione dei dati che voglio scrivere è inferiore alla dimensione del buffer o dello spazio libero rimasto, l'operazione è atomica; altrimenti può darsi che l'operazione non sia atomica, viene eseguita in più cicli).

- È necessario **definire un protocollo di comunicazione** tra i processi per evitare problemi di sincronizzazione.

Esempio 1:

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

#define READ 0          //lato lettura pipe
#define WRITE 1         //lato scrittura pipe
#define MAXLEN 100

char *msg = "Content of pipe.";

int main(){
  int fd[2]; //contiene i riferimenti all'estremita della pipe
  int bytesRead; //byte letti
  char message [MAXLEN]; //buffer

  ////inizializza il vettore fd con due interi creando due descrittori che sono intercorrelati tra loro
  pipe(fd); //devo controllare che sia andato tutto bene
  if(fork() > 0){ //PADRE scrive dati
    close(fd[READ]); //chiudo lato lettura del buffer
    write(fd[WRITE], msg, strlen(msg)+1); //strlen() non tiene conto del terminatore di stringa
    printf("PARENT: Written data.\n");
    close(fd[WRITE]);
  }else{ //FIGLIO legge dati
    close(fd[WRITE]); //chiudo lato scrittura del buffer
    bytesRead = read(fd[READ], message, MAXLEN);
    printf("CHILD: Read %d bytes: %s\n", bytesRead, message);
    close(fd[READ]);
  }
}
```

Esempio 2:

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

#define READ 0
#define WRITE 1

// ./piperedir ls cat == ls | cat

int main(int argc, char *argv[]){
  int fd[2];

  pipe(fd);

  if(fork() > 0){
    close(fd[READ]);
    //quando il comando verra' eseguito, mandera' il suo output al descrittore e non a video
    dup2(fd[WRITE], 1); //dirotto il canale di output
    close(fd[WRITE]);
    execlp(argv[1], argv[1], NULL);
    perror("?Connect"); //viene visualizzato solo la exec non va a buon fine
  }else{
    close(fd[WRITE]);
    //mando i dati in input al descrittore
    dup2(fd[READ], 0); //dirotto il canale di input 0 con il descrittore
    close(fd[READ]);
    execlp(argv[2], argv[2], NULL);
    perror("?Connect"); //viene visualizzato solo la exec non va a buon fine
  };
}
```

- **PIPING NAMED (FIFO):**
  - Librerie: sys/types.h e sys/stat.h
  - **Creano un buffer che ha una denominazione** propria ed è gestito mediante dei **file particolari** salvati sul file system.
    - Sono **file persistenti**.
  - **Qualunque processo che conosca il riferimento di questi file speciali può accedervi** e usarli (se ha i permessi).
  - **Creazione** mediante la funzione:

    **int mkfifo(const char *pathname, mode_t mode);** //path del file, permessi

- Esempio:

**Processo READER**

```c
//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/pipe/pipe3/myfifo";
   //crea un file particolare su disco
   mkfifo(myfifo, 0666); //mkfifo(<pathname>, <permission>)
   char str1[80], str2[80];
   while(1){
     //open() fa da sincronizzatore
     fd = open(myfifo, O_RDONLY); //open FIFO for read only
     read (fd, str1, 80); //read from FIFO
     printf("User1: %s\n", str1); //print message 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); //write and close
     close(fd);
   };
   return 0;
}
```

**Processo WRITER**

```c
//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/pipe/pipe3/myfifo";
   //crea un file particolare su disco
   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, 80); //read from FIFO
     printf("User1: %s\n", str1); //print message and close
     close(fd);
   };
   return 0;
}
```