Laboratorio - system call C e comandi Shell

▼ Indice

- **access()** → *6*. Per verificare i diritti di un utente di accedere a un file
- **alarm()** → Imposta un timer
- **chmod()** → *4*. Modifica i bit di protezione
- **chown()** → *4*. Cambia il proprietario di un file
- **chdir()** → *6*. Per cambiare direttorio
- **close()** → *4*. Chiude un file aperto
- **closedir()** → *6*. Chiunde un direttorio
- **creat()** → Crea un file
- **dup()** → Duplica un elemento della tabella dei file aperti di processo.
- **execl()** → Esegue un eseguibile
- **execlp()** → Esegue un eseguibile
- **execve()** → Esegue un eseguibile
- **exit()** → Uscita volontaria da un processo
- **kill()** → Forza un segnale a un processo
- **link()** → *6*. Per aggiungere un link a un file esistente
- **lseek()** → *4*. Sposta il puntatore I/O
- **fork()** → Duplica un processo
- **getpid()** → Restituisce il PID del processo corrente
- **getppid()** → Restituisce il PID del processo padre
- **mkdir()** → *6*. Crea un direttorio
- **open()** → *4*. Apre un file
- **opendir()** → *6*. Apre un direttorio
- **pause()** → Attende un segnale
- **perror()** → Interpreta gli errori delle system calls
- **pipe()** → *5*. Consente la comunicazione tra processi
- **read()** → *4*. Legge un file
- **readdir()** → *6*. Legge un direttorio
- **receive()** → Consente la comunicazione tra più processi
- **reply()** → Consente la comunicazione tra più processi
- **send()** → Consente la comunicazione tra più processi
- **signal()** → Associa ad ogni segnale il rispettivo handler
- **sleep()** → Provoca la sospensione del processo
- **stat()** → *6*. Per leggere gli attributi di un file.
- **unlink()** → *6*. Per decrementare il numero di link del file
- **wait()** → Aspettare la terminazione di un figlio.
- **write()** → *4*. Scrive un file.

To-Do deadlock Java

Template Fedele Penna

Filesystem

/ → Root
/bin → File binari dei comandi essenziali
/sbin → File binari dei comandi di sistema essenziali
/home → Home degli utenti
/var → Dati variabili
/boot → File per operazioni di boot (avvio) della macchina
/dev → File dispositivi
/etc → File di configurazione
/lib → Shared libraries e moduli del kernel
/media → Mount point per media rimovibili
/mnt → Mount point per operazioni di mount temporanee di FS
/opt → Software applicativi
/tmp → File temporane

Comandi Linux

▼ Metacaratteri

> *****
>
> Una qualunque stringa di zero o più caratteri in un **nome di un file**.

> **?**
>
> Indica un qualunque carattere in un **nome di file**.

> **[zfc]**
>
> Indica un qualunque carattere in un **nome di file** compreso tra quelli nell'insieme. Si può esprimere intervalli di valori (*Esempio* **[a-d]**).

> #
>
> Commento fino alla fine della linea.

> \
>
> **Escape** segnala di non interpretare il carattere successivo come speciale.

```
$ ls [q-s]*
```

```
$ ls ese*
```

```
$ ls [a-p,1-7]*[c,f,d]?
```

```
$ cat esempio.txt > out\*.txt
```

```
$ ls *\**
```

▼ Controlli booleani

- `-lt` → Less than
- `-le` → Less equal
- `-gt` → Great than
- `-ge` → Great equal
- `-ne` → Not equal
- `-e` → Equal
- `-f` → Indica se una variabile è un file
- `-d` → Indica se una variabile è una directory

▼ $

Consente di richiamare il valore di una variabile

```
#!/bin/bash
#file somma.sh
A=5
B=8
echo A=$A, B=$B
C=expr $A + $B
echo C=$C
D=`expr $A + $B`
echo D=$D
```

Alcune variabili notevoli sono predefinite

```
#!/bin/bash
#file hello
echo hello $1 and $2!

Eseguo da terminale:
bash-2.05:~$ ./hello Anna Luca
hello Anna and Luca!
```

→ **$*** rappresenta l'insieme di tutte le variabili posizionali che corrispondono agli argomenti del comando.

→ **$#** numero di argomenti passati (**$0** escluso).

→ **$$** id numerico del processo in esecuzione.

→ **$?** valore (int) restituito dall'ultimo comando eseguito.

```
#!/bin/bash
#file printpid
echo Il mio pid: $$

Eseguo da terminale:
bash-2.05:~$ echo $$
2001
bash-2.05:~$ echo $$
2001
bash-2.05:~$ ./printpid
Il mio pid: 956
bash-2.05:~$ ./printpid
Il mio pid: 958
# Attenzione, cambia perchè ogni volta che il programma
# viene eseguito ha un pid diverso.
```

▼ ``

Indicano alla shell di valutare la stringa racchiusa tra ` come un comando. Esso vine eseguito e sostiutito con il suo output.

▼ <

Ridirezione in input

```
$ Comando < F
```

l'input del comando viene acquisito dal file F (invece che dal dispositivo di standard input) [vale solo per comandi «filtro»: grep, more, wc ecc.]

```
$ grep main < hello.c
```

```
sort < file > file2
```

▼ >

Crea un file vuoto.

Può essere utilizzato come comando di *reindirezzamento*:

```
$ Comando > F
```

L'output del Comando viene scritto nel file F
(e non sul dispositivo di standard output)

```
$ ls –l p* > pippo
```

▼ >>

Reindirizzamento in append

```
$ Comando >> F
```

l’output del comando viene aggiunto in coda al contenuto del file F (invece che sul dispositivo di standard output)

```
$ ps >> pippo
```

▼ |

L’operatore | consente di realizzare «pipeline» di comandi:

```
$ Comando1 | Comando2 | Comando3
```

l’output di Comando1 viene ridiretto nell’input di Comando2;

l’output di Comando2 viene ridiretto nell’input di Comando3.

```
$ cat hello.c | grep printf | wc -l
```

```
$ who | wc –l
```

```
$ ls -l | grep ^d | rev | cut -d' ' -f1 | rev
```

▼ ~

Utilizzato per indicare la $HOME del percorso corrente

```
output=~/"$1".log
```

▼ ./<filename>[&]

Esegue il file.

Se è presente la ‘&’ il programma viene avviato in backround e viene restituito il pid del processo.

▼ awk '<pattern> { action }' [<file>...]

Comando che permette la ricerca di testo ed esecuzioni di azioni:

```
$ awk '/str/ { print }' file1
```

```
$ awk '/^In/ { print }' file1
```

```
$ awk '/^In/ { print $2 }' file1
```

```
$ awk –F';' '{ print $4}' file1
```

```
$ ls –la | awk '{ print $3"***"$1}'

output
studente***drwxr-xr-x
studente***drwxr-xr--
studente***-rw-r--r--
```

▼ date

Stampa a video data e ora attuali.

▼ pwd

Stampa a video la cartella corrente.

▼ cat <filename>

Mostra a video il contenuto di un file.

▼ cd ..

Sposta l’utente nella cartella superiore.

▼ cd [dir]

Change directory, sposta l’utente nel percorso <dir>.

▼ chmod <mode><filename>

Consente di cambiare i permessi di protezione associati a un file.

```
:~$ ls -l sera
-rw-rw—r-- 1 daniela staff ... sera

:~$ chmod 0666 sera ([6]8 = [110]2)
:~$ ls -l sera
-rw-rw-rw- 1 daniela staff ... sera

:~$ chmod a-w,u=rw sera
:~$ ls -l sera
-rw-r--r-- 1 daniela staff ... sera
```

▼ cp <filename><newfile>

Copy, copia un file

▼ cut [-options]<filename>

Seleziona colonne da file.

Output: video.

▼ echo […]

Permette di visualizzare a terminale.

▼ expr […]

Consente di valutare un espressione numerica (altrimenti essa verrebbe considerata come una stringa)

```
utente~$ expr 1 + 3
4
```

▼ gcc <file.c> -o <file exe>

Consente di compilare il file c in un eseguibile.

▼ grep <string>[<filename>]

Ricerca di una stringa in file.

▼ kill [-opz…]<pid>

Termina il processo identificato con PID “process”.

▼ [-opz…]

-[errorID] → Termina il processo attraverso il segnale

-l → Lista tutti i segnali possibili

▼ ls [-opz…][file/dir]

Conste di visualizzare i nomi contenuti in un direttorio, opzioni:

▼ [-opz…]

-F → *classify*, aggiunge al termine del nome del file un carattere che ne indica il tipo:

***** → eseguibile.

/ → direttorio.

@ → link simbolico.

| → FIFO.

= → socket.

-l → *long format*, stampa a video più informazioni.

-a → *all files*, lista completa (anche dei file che iniziano per ‘.’)

-t → *time*, lista in ordine dell’ultima modifica.

-u → lista in ordine per data dell’ultimo accesso.

▼ [file/dir]

Nome del direttorio o elenco dei file di cui fare il listing.

E’ possbile usare metacaratteri:

***** → Numero indefinito di caratteri.

[1-4] → Tutti i numeri da 1 a 4.

▼ man

Manuale che permette di visualizzare informazioni su un comando Linux

▼ more <filename>

Visualizza un file per videate

▼ mkdir <nomedir>

Make dir, crea una directory

▼ mv <filename><newfile>

Move, sposta un file da un direttorio a un altro.

▼ passwd

Permette all’utente di modificare la propria password.

▼ ps

Mostra i processi attualmente in esecuzione.

▼ pwd

Indica la directory in cui ci si trova.

```
recursive_command="`pwd`/$dir_name/recursive.sh"
```

▼ rev <filename>

Inverte l’ordine delle righe del file:

Output: video.

▼ rm <filename>

Remove, elimina un file

▼ rmdir

Remove directory, rimuove un direttorio (deve essere vuoto).

▼ set

Permette di vedere le variabili di ambiente e i valori loro associati.

```
$ set

Output:
BASH=/usr/bin/bash
HOME=/space/home/wwwlia/www
PATH=/usr/local/bin:/usr/bin:/bin
PPID=7497
PWD=/home/Staff/AnnaC
SHELL=/usr/bin/bash
TERM=xterm
UID=1015
USER=anna
```

Nel caso ci siano argomenti **set** riassegna gli argomenti **$1** .. **$n**

```
#!/bin/bash
#file hello
echo hello $1, $2 and $3!
set Pluto Pippo Paperino
echo hello $1, $2 and $3!

Eseguo da terminale
bash-2.05:~$ ./hello Daniela Luca Paolo
hello Daniela, Luca and Paolo!
hello Pluto, Pippo and Paperino!
```

▼ shift

Fa scorrere tutti gli argomenti verso sinistra in un file bash.

```
#!/bin/bash
#file hello
echo hello $1, $2 and $3!
shift
echo hello $1, $2 and $3!


Eseguo da terminale:
bash-2.05:~$ ./hello Daniela Luca Paolo
hello Daniela, Luca and Paolo!
hello Luca, Paolo and !
```

▼ sort [<filename>…]

Ordina alfabeticamente le righe:

Input: lista di file.

Output: video.

▼ startx

Accede alla macchina dalla Shell.

▼ tee <filename>

Scrive l’input sia su file che su canale output

▼ top

Visualizza tutti i processi (’Q’ per uscire).

▼ wc [-lwc]<filename>

Conteggio di righe, parole e caratteri

▼ who

Stampa a video informazioni sugli ultimi accessi.

▼ whoami

Stampa a video l’utente attuale.

Costrutti programmazione shell utili

▼ Controllo degli argomenti

```
# Controllo che sia un intero
if [[ $1 = *[!0-9]* ]]; then
  echo "$1 non è un intero positivo" 1>&2
```

```
  exit 1
fi
```

```
# Controllo che ci siano esattamente 4 argomenti
if [[ $# -ne 4 ]]; then
  echo "Numero di argomenti diverso da 4"
  exit 1
fi
```

```
# Controllo che ci siano almeno 4 argomenti
if [[ $# -lt 4 ]]; then
  echo "Numero di argomenti diverso da 4"
  exit 1
fi
```

```
# Controllo che $dir sia una directory
if ! [[ -d "$dir" ]]; then
  echo "La directory dir non è una directory"
  exit 1
fi
```

```
# Controllo che $f sia una directory
if ! [[ -f "$f" ]]; then
  echo "Il file f non è un file"
  exit 1
fi
```

```
# Controllo che $1 abbia un pattern .XXX
if ! [[ "$1" = .??? ]]; then
  echo "Il primo argomento non segue il pattern corretto"
  exit 1
fi
```

▼ Switch case per capire se un path è assoluto o relativo

```
case "$0" in
    # La directory inizia per / Path assoluto.
/*)
    dir_name=`dirname $0`
    recursive_command="$dir_name"/rec_shell.sh
    ;;
*/*)
    # La directory non inizia per slash, ma ha uno slash al suo interno.
    # Path relativo.
    dir_name=`dirname $0`
    recursive_command="`pwd`/$dir_name/rec_shell.sh"
    ;;
*)
    # Path né assoluto nè relativo, il comando deve essere nel $PATH
    # Comando nel path
    recursive_command=rec_shell.sh
    ;;
esac
```

▼ Iterare su tutti i file in una cartella

```
for f in *; do
  ...
```

```
done
```

```
# Iterare i file con una determinata estensione $1

for f in *$1; do
  ...
done
```

▼ Contare le ricorrenze di una parola all’interno di un file

```
var="`grep -s -o "parola" "prova.txt" | ls -l`"
```

▼ Stampare solo un campo di ls

```
ls -la | awk '{print $1}'
```

Comandi C

▼ Include

```
#include <fcntl.h>
#include <stdio.h>
#include <unistd.h>
#include <signal.h>
#include <stdlib.h>
#include <string.h>
#include <sys/wait.h>


// Include più rare
#include <sys/errno.h>  // Per interpretare gli errori [perror()]
#include <sys/types.f>  // Usato da readdir()
#include <sys/stat.h>   // Per leggere gli attributi di un file [stat()]
#include <dirent.h>     // Per operazioni con i direttori
#include <time.h>
```

▼ Wait child

```
void wait_child() {
  int pid_terminated,status;
  pid_terminated=wait(&status);
    if(WIFEXITED(status))
      printf("\nPADRE: terminazione volontaria del figlio %d con stato %d\n",pid_terminated,WEXITSTATUS(status));
    else if(WIFSIGNALED(status))
      printf("\nPADRE: terminazione involontaria del figlio %d a causa del segnale %d\n",pid_terminated,WTERMSIG(status));
}
```

▼ Controlli

Una stringa d’ingresso rappresenti un path assoluto

```
if (argv[1][0]!='/'){
    printf("Il primo argomento deve essere un nome assoluto di file\n");
    exit(-2);
  }
```

Un numero sia un intero positivo

```
if ( n <= 0  ){
        printf("Il secondo argomento deve essere un intero positivo\n");
        exit(-3);
    }
```

▼ access() → *6*. Per verificare i diritti di un utente di accedere a un file

```
int access (char * pathname,int amode);
```

→ `patmode` rappresenta il nome del file

→ `amode` esprime il diritto da verificare e può essere:

- `04` → read access
- `02` → write access
- `01` → execute access
- `00` → existence

La funzione restituisce il valore `0` in caso di successo (diritto verificato), altrimenti `-1`.

▼ alarm() → Imposta un timer

```
unsigned int alarm(unsigned int N)
```

La funzione **alarm()** non sospende il processo, ma invia dopo `N` secondi un segnale di `SIGALARM` l'azione di default associata a questo segnale è la *terminazione del processo*.

Ritorna 0 se non vi erano time-out impostati in precedenza, altrimenti ritorna il numero di secondi che mancavano al timer precedente.

▼ chmod() → *4*. Modifica i bit di protezione

```
int chmod (char *pathname, char *newmode);
```

→ `pathmode` è il nome del file.

→ `newmode` contiene i nuovi diritti.

▼ chown() → *4*. Cambia il proprietario di un file

```
int chown(char *pathname, int owner, int group);
```

→ `pathname` è il nome del file

→ `owner` è l'uid del nuovo proprietario

→ `group` è il gid del gruppo

Cambia proprietario/gruppo del file, può essere eseguita solo dall'utente root

▼ chdir() → *6*. Per cambiare direttorio

```
int chdir (char *nomedir);
```

Equivalente a cd.

→ `nomedir` è il nome del direttorio in cui entrare

Restituisce `0` in caso di successo, altrimenti restituisce `-1` in caso di fallimento.

▼ close() → *4*. Chiude un file aperto

```
int close(int fd);
```

→ `fd` è il file descriptor del file da chiudere.

Restituisce l'esito della operazione, 0 in caso di successo, <0 in caso di insuccesso.

▼ closedir() → *6*. Chiunde un direttorio

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

Effettua la chiusura del direttorio riferito dal puntatore `dir`.

Ritorna `0` in caso di successo, `-1` altrimenti.

```
#include <stdlib.h>
#include <sys/types.h>
#include <dirent.h>
#include <fcntl.h>
void miols(char name[]){
  DIR *dir; struct dirent * dd;
  char buff[80];
  dir = opendir (name);
  while ((dd = readdir(dir)) != NULL){
    sprintf(buff, "%s\n", dd->d_name);
    write(1, buff, strlen(buff));
  }
  closedir (dir);
  return;
}

main (int argc, char **argv){
  if (argc <= 1){
    printf("Errore\n");
    exit(1);
  }
  miols(argv[1]);
  exit(0);
}
```

▼ creat() → Crea un file

```
int creat(char nomefile[], int mode);
```

→ `nome file` è il nome del file (relativo o assoluto) da creare.

→ `mode` specifica i 12 bit di protezione del nuovo file

Il valore restituito è il file descriptor associato al file creato o -1 nel caso di errore.

Se la **creat()** ha successo, il file viene aperto in scrittura e il pointer posizionato sul primo elemento.

▼ dup() → Duplica un elemento della tabella dei file aperti di processo.

```
int dup(int fd);
```

→ `fd` è il *file descriptor* da duplicare.

La chiamata a **dup()** copia l'elemento **fd** della tabella dei file aperti nella prima posizione libera della tabella.

Restituisc il nuovo *file descriptor* o `-1` in caso di errore.

```
main(){
    int pid, fd[2]; char msg[3]="bye";
    pipe(fd);
    pid=fork();
    if (!pid){ /* processo figlio */
        close(fd[0]); /* chiusura lato lettura della pipe */
        close(1); /* chiudo disp. stdout*/
        dup(fd[1]); /* ridirigo stdout sul lato di
                       scrittura della pipe */
        close(fd[1]); /* elim. seconda copia lato scritt.*/
        write(1,msg, sizeof(msg)); /*scrivo su pipe*/
        close(1);
    }else{ /*processo padre*/
          close(fd[1]); /* chiusura lato scrittura della pipe */
          read(fd[0], msg, 3);
          close(fd[0]);
    }
}
/*
  In caso di una write() normale il processo di reindirizzamento
  può sembrare inutile, ma esso si rivela estremamente utile quando nel
  codice è presente una exec() che lancia un comando Linux che scrive di
  default su  stdio (ls, grep, etc...).
*/
```

▼ execl() → Esegue un eseguibile

```
#include <unistd.h>
int execl(char *pathname, char *arg0, ..., char argN, (char)0);
```

→ `pathname` è il nome dell'eseguibile da caricare

→ `arg0` è il nome del programma (argv[0])

→ `arg1, ..., argN` sono gli argomenti da passare al programma

→ `(char*)0` è il puntatore nullo che termina la lista

In assenza di errori, execl è una chiamata senza ritorno, se la funzione **execl()** ritorna un valore, significa che la chiamata è fallita, pertanto il processo contimna ad eseguire il codice iniziale.

```
pid = fork();
if (pid == 0){ /* figlio */
  printf("Figlio: esecuzione di ls\n");
  execl("/bin/ls", "ls", "-l", (char *)0);
  perror("Errore in execl");
  exit(1);
}
if (pid > 0){ /* padre */
  ...
  printf("Padre ....\n");
  exit(0);
}
if (pid < 0){ /* fork fallita */
  print("Errore in fork\n");
  exit(1);
}
```

▼ execlp() → Esegue un eseguibile

```
#include <unistd.h>
int execlp(char *filename, char *arg0, ..., char argN, (char)0);
```

→ `filename` è il nome dell'eseguibile da caricare

→ `arg0` è il nome del programma (argv[0])

→ `arg1, ..., argN` sono gli argomenti da passare al programma

→ `(char*)0` è il puntatore nullo che termina la lista

In assenza di errori, execl è una chiamata senza ritorno, se la funzione **execlp()** ritorna un valore, significa che la chiamata è fallita, pertanto il processo continua ad eseguire il codice iniziale.

```
#include <stdio.h>
#include <string.h>
#define DIM 20
#define MAXP 10
typedef char stringa[80];
typedef stringa strvett[DIM];
strvett vstr;
void gest_stato(int S, int pid);
void figlio(int i);

main(int argc, char** argv){
  int pid[MAXP],ncom, stato, i;
  ncom=argc-1;
  for(i=0; i<ncom; i++){
    strcpy(vstr[i], argv[i+1]); /* vstr[0]=argv[1], ecc. */
  }
  for(i=0; i<ncom; i++)
    if ((pid[i]=fork())==0)
      figlio(i);
  for(i=0; i<ncom; i++){
    pid[i]=wait(&stato);
    gest_stato(stato,pid[i]);
  }
}

void figlio(int i){
  printf("\nProcesso %d per comando %s",getpid(), vstr[i]);
  execlp(vstr[i], vstr[i], (char *)0);
  perror("\n exec fallita: ");
  exit(-1);
}

int gest_stato(int S, int pid){
  printf("terminato processo figlio n.%d", pid);
  if ((char)S==0)
    printf("term. volontaria con stato %d", S>>8);
  else{
    printf("terminazione involontaria per segnale %d: MUOIO!\n", (char)S);
  exit(1);
  }
}
```

▼ execve() → Esegue un eseguibile

```
#include <unistd.h>
int execve(char *pathname, char *argV[], char * env[]);
```

→ `pathname` è il nome assoluto o relativo dell'eseguibile da caricare.

→ `argV` è il vettore degli argomenti del programma da eseguire

→ `env` è il vettore delle variabili di ambienti da sostituire all'ambiente del processo (contiene stringhe del tipo "VARIABILE=valore")

```
char *env[]={"USER=paolo",
             "PATH=/home/paolo/d1",
             (char *)0};
char *argv[]={"ls", "-l", "pippo", (char *)0};
int main()
{
  int pid, status;
  pid=fork();
  if (pid==0){
    execve("/bin/ls", argv, env);
    perror("exec fallita a causa dell'errore:");
    exit(1);
  }
  else if (pid >0){
    pid=wait(&status); /* gestione dello stato.. */
  }
  else
    perror(" fork fallita a causa dell'errore:");
}
```

▼ exit() → Uscita volontaria da un processo

```
#include <sys/wait.h>
void exit(int status);
```

La funzione **exit()** prevede un parametro *status* mediante il quale il processo comunica al padre **informazioni sul suo stato di terminazione**.

→ Se il processo che termina ha figli in esecuzione il processo init adotta i figli dopo la terminazione di stato.

→ Se il processo termina che il padre ne rilevi lo stato di terminazione con la system call **wait()**, il processo passa nello stato zombie.

```
#include <sys/wait.h>
main(){
  int pid, status;
  pid=fork();
  if (pid==0){
    printf("figlio");
    exit(0);
  }
  else{
    pid = wait(&status);
    printf("terminato processo figlio n.%d", pid);
    if ((char)status==0)
      printf("term. volontaria con stato %d", status>>8);
    else
      printf("terminazione involontaria per segnale %d\n", (char)status);
  }
}
```

```
#include <sys/wait.h>
#define N 100
int main(){
  int pid[N], status, i, k;
  for (i=0; i<N; i++){
    pid[i]=fork();
    if (pid[i]==0){
      printf("figlio: il mio pid è: %d", getpid());
      exit(0);
    }
  }
  for (i=0; i<N; i++){ /* attesa di tutti i figli */
    k=wait(&status);
    if (WIFEXITED(status))
      printf("Term. volontaria di %d con stato %d\n", k,WEXITSTATUS(status));
    else if (WIFSIGNALED(status))
      printf("term. Involontaria di %d per segnale %d\n",k, WTERMSIG(status));
}
```

▼ kill() → Forza un segnale a un processo

```
#include <signal.h>
int kill(int pid, int sig)
```

→ `sig` è l'intero (o il nome simbolico) che individua il segnale da gestire

→ `pid` specifica il destinatario del segnale

→ `pid` > 0: l'intero è il pid dell'unico processo destinatario.

→ `pid` = 0: il segnale è spedito a tutti i processi del gruppo del mittente.

→ `pid` < -1: il segnale è spedito a tutti i processi con groupId uguale al valore assoluto di pid

→ `pid` = -1: Possibilità non specificata negli standard Posix

```
#include <stdio.h>
#include <signal.h>

int cont=0;

void handler(int signo){
  printf ("Proc. %d: ricevuti n. %d segnali %d\n",
  getpid(),cont++, signo);
}

main (){
  int pid;
  signal(SIGUSR1, handler);
  pid = fork();
  if (pid == 0) /* figlio */
    for (;;);
  else /* padre */
    for(;;) kill(pid, SIGUSR1);
}
```

▼ link() → *6*. Per aggiungere un link a un file esistente

```
int link(char *oldname, char * newname);
```

→ `oldname` è il nome del file esistente

→ `newname` è il nome associato al nuovo link

Incrementa il numero dei link associati al file, aggiorna il direttorio, ritorna `0` in caso di successo, `-1` in caso di fallimento.

Fallisce se:

- `oldname` non esiste
- `newname` esiste già
- `oldname` e `newname` appartengono a file system diversi.

```
main (int argc,char ** argv){
  if (argc != 3){
    printf ("Sintassi errata\n"); exit(1);
  }
  if (link(argv[1], argv[2]) < 0){
    perror ("Errore link"); exit(1);
  }
  if (unlink(argv[1]) < 0){
    perror("Errore unlink"); exit(1);
  }
```

```
  exit(0);
}
```

▼ lseek() → *4.* Sposta il puntatore I/O

```
#include <fcntl.h>
lseek(int fd, int offset,int origine);
```

→ `fd` è il file decriptor del file

→ `offset` è lo spostamento in byte rispetto all'origine

→ `origine` può valere:

- `0`: inizio file (`SEEK_SET`)
- `1`: inizio file (`SEEK_CUR`)
- `2`: inizio file (`SEEK_END`)

```
#include <fcntl.h>
main(){
  int fd,n; char buf[100];
  if(fd=open("/home/miofile",O_RDWR)<0){
    ...;
    lseek(fd,-3,2); /* posizionamento sul terz'ultimo byte del
                        file */
  }
  ...
}
```

▼ fork() → Duplica un processo

```
#include <unistd.h>
int fork(void);
```

Duplica un processo, restituisce al padre il PID del figlio, al figlio restituisce 0.

Esempio codice

```
#include <stdio.h>
main(){
  int pid;
  pid=fork(); //forketta hihihihihihi
  if (pid==0){ /* codice figlio */
    printf("Sono il figlio ! (pid: %d)\n" , getpid());
  }
  else if (pid>0){ /* codice padre */
    printf("Sono il padre: pid di mio figlio: %d\n", pid);
  }
  else printf("Creazione fallita!");
}
```

▼ getpid() → Restituisce il PID del processo corrente

```
int getpid();
```

Restituisce il PID del processo che la chiama.

▼ getppid() → Restituisce il PID del processo padre

```
int getppid();
```

Restituisce il PID del processo padre.

▼ mkdir() → *6*. Crea un direttorio

```
int mkdir (char *pathname, int mode);
```

→ `pathname` è il nome del direttorio da creare.

→ `mode` esprime i bit di protezione.

Restituisce `0` in caso di successo, altrimenti un valore negativo.

Crea e inizializza un direttorio con il nome e i diritti specificati (***N.B.*** vengono sempre creati i file . (link al direttorio corrente) e .. (link al direttorio del padre))

▼ open() → *4*. Apre un file

```
#include <fcntl.h>
int open(char nomefile[],int flag, [int mode]);
```

→ `nomefile` è il nome del file, relativo o assoluto.

→ `flag` esprime il modo di accesso, definiti in `<fcntl.h>`, ad esempio:

  → `O_RDONLY` [=0] per l'accesso in lettura.

  → `O_WRONLY` [=1] per l'accesso in scrittura.

  → `O_APPEND` [=2] per l'accesso in scrittura con la modalità *append*, aggiunge senza rimuovere il contenuto precedente.

A questi è possibile abbinare tramite il connettore `|` altre modalità:

  → `O_CREAT` Se il file non esiste viene creato.

  → `O_TRUNC` La lunghezza viene troncata a 0.

→ `mode` è un parametro richiesto soltanto se l'apertura determina la creazione del file, in tal caso specifica i bit di protezione (codifica ottale).

open() restituisce il file descriptor associato al file nel caso abbia successo, `-1` in caso di errore.

```
#include <fcntl.h>
...
main(){
  int fd1, fd2, fd3;
  fd1=open("/home/anna/ff.txt", O_RDONLY);
  if (fd1<0) perror("open fallita");
  ...
  fd2=open("f2.new",O_WRONLY);
  if (fd2<0){
    perror("open in scrittura fallita:");
    fd2=open("f2.new",O_WRONLY|O_CREAT, 0777);
    /* è equivalente a:
    fd2=creat("f2.new", 0777); */}
  /*OMOGENEITA`: apertura dispositivo di output:*/
  fd3=open("/dev/prn", O_WRONLY);
  ...
}
```

▼ opendir() → *6*. Apre un direttorio

```
#include <dirent.h>
DIR *opendir (char *nomedir);
```

→ `nomedir` è il nome del direttorio da aprire (deve essere necessariamente una directory)

Restituisce un valore di tipo DIR (puntatore a direttorio).

```
#include <stdlib.h>
#include <sys/types.h>
#include <dirent.h>
#include <fcntl.h>
void miols(char name[]){
  DIR *dir; struct dirent * dd;
  char buff[80];
  dir = opendir (name);
  while ((dd = readdir(dir)) != NULL){
    sprintf(buff, "%s\n", dd->d_name);
    write(1, buff, strlen(buff));
  }
  closedir (dir);
  return;
}


main (int argc, char **argv){
  if (argc <= 1){
    printf("Errore\n");
    exit(1);
  }
  miols(argv[1]);
  exit(0);
}
```

▼ pause() → Attende un segnale

```
int pause(void)
```

Sospende il processo fino alla ricezione di un segnale.

Per convenzione restituisce il valore -1 nel momento in cui riceve un qualunque segnale.

```
int ntimes = 0;

void handler(int signo){
  printf ("Processo %d ricevuto #%d volte il segnale %d\n",
  getpid(), ++ntimes, signo);
}

main (){
  int pid, ppid;
  signal(SIGUSR1, handler);
  if ((pid = fork()) < 0) /* fork fallita */
    exit(1);
  else if (pid == 0){ /* figlio*/
    ppid = getppid(); /* PID del padre */
    for (;;){
      printf("FIGLIO %d\n", getpid());
      sleep(1);
      kill(ppid, SIGUSR1);
      pause();}
  }
  else /* padre */
    for(;;){ /* ciclo infinito */
      printf("PADRE %d\n", getpid());
      pause();
      sleep(1);
      kill(pid, SIGUSR1);
    }
}
```

▼ perror() → Interpreta gli errori delle system calls

In caso di fallimento ogni system call ritorna un valore negetivo.

In aggiunta UNIX prevede la variabile globale di sistema **errno** alla quale il kernel assegna il codice di errore generato dall'ultima system call eseguita. Per interpretarne il valore è possibile usare **perror()**:

perror([str]) stampa la stringa passata come parametro seguita dalla descrizione testuale del codice di errore contenuto in **errno**.

La corrispondenza codice - descrizione è contenuta in **<sys/errno.h>**

```
#include <stdio.h>
#include <stdlib.h>
#include <sys/errno.h>
#define N 10

int main()
{
  int pid[N], status, i, k;
  for (i=0; i<N; i++)
  {
    pid[i]=fork();
    if (pid[i]==0){
      printf("figlio: il mio pid è: %d\n", getpid());
      exit(0);
    }
  }
  while(1){ /* attesa figli */
    k=wait(&status);
    if (k<0) {
      perror("wait fallita");
      exit(1);
    }
    else
      printf("Terminato figlio %d \n", k);
  }
}
```

▼ pipe() → *5*. Consente la comunicazione tra processi

```
int pipe(int fd[2]);
```

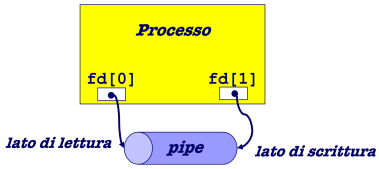
→ `fd` è il puntatore a un vettore di 2 file descriptor che verranno inizializzati dalla system call in caso di successo

→ `fd[0]` rappresenta il lato di lettura della pipe

→ `fd[1]` rappresenta il lato di scrittura della pipe

pipe() restituisce un valore negativo in caso di fallimento, 0 se ha successo.

Si può accedere alla pipe mediante le system call di accesso a file: **read()** (***vedi read()***) e ***write()*** (***vedi write()***).



Rappresentazione grafica di una *pipe*

```
main(){
  int pid;
  char msg[]=“ciao babbo”;
  int fd[2];
  pipe(fd); //Creazione della pipe
  pid=fork();
  if (pid==0){/* figlio */
    close(fd[0]);
    write(fd[1], msg, 10);
    ...
  }
  else{ /* padre */
    close(fd[1]);
    read(fd[0], msg, 10);
    ...
  }
}
```

Ogni processo può chiudere un estremo della pipe con una **close()** (*vedi close()*):

Se un processo P:

- tenta una lettura da una pipe vuota il cui lato di scrittura è effettivamente chiuso: **read()** ritorna `0`
- tenta una scrittura da una pipe il cui lato di lettura è effettivamente chiuso: **write()** ritorna `-1`, ed il segnale `SIGPIPE` viene inviato a P (broken pipe).

```
/* Sintassi: progr N
padre(destinatario) e figlio(mittente) si scambiano una
sequenza di messaggi di dimensione (DIM) costante;
la lunghezza della sequenza non e` nota a priori;
il destinatario decide di interrompere la sequenza di
scambi di messaggi dopo N secondi */
#include <stdio.h>
#include <signal.h>
#define DIM 10
/*Definizione di funzioni e variabili globali*/
int fd[2];
void fine(int signo);
void timeout(int signo);

main(int argc, char **argv){
  int pid, N;
  char messaggio[DIM]="ciao ciao ";
  if (argc!=2){
    printf("Errore di sintassi\n");
    exit(1);
  }
  N=atoi(argv[1]);
  pipe(fd);
  pid=fork();
  if (pid==0){ /* figlio mittente */
    signal(SIGPIPE, fine);
    close(fd[0]); // Viene chiuso il lato di lettura per il figlio.
    for(;;)
      write(fd[1], messaggio, DIM);
  }
  else if (pid>0){ /* padre */
    signal(SIGALRM, timeout);
    close(fd[1]); /* Viene chiuso il lato di scrittura per il padre */
    alarm(N);     /* Viene avviato il conto alla rovescia */
    for(;;){
      read(fd[0], messaggio, DIM);
      write(1, messaggio, DIM);
    }
  }
}/* fine main */


/* definizione degli handler dei segnali */
void timeout(int signo){
```

```
  int stato;
  close(fd[0]); /* chiusura effett. del lato di lettura*/
  wait(&stato);
  if ((char)stato!=0)
    printf("Term. inv. figlio (segnale %d)\n", (char)stato);
  else printf("Term. Vol. Figlio (stato %d)\n", stato>>8);
  exit(0);
}

void fine(int signo)
{
  close(fd[1]);
  exit(0);
}
```

▼ read() → *4.* Legge un file

```
int read(int fd,char *buf,int n);
```

→ `fd` è il file descriptor del file.

→ `buf` è l'area in cui trasferire i byte letti.

→ `n` è il numero di caratteri da leggere.

In caso di successo resituisce un intero positico (≤`n`)che rappresenta il numero di caratteri effettivamente letti

Il carattere *End-Of-File* marca la fine del file.

Se la lettura ha succoesso:

→ L' I/O pointer viene spostato avanti di `n` bytes.

```
#include <fcntl.h>
main(){
  int fd,n;
  char buf[10];
  if((fd=open("/home/miofile",O_RDONLY))<0){
    perror("errore di apertura:");
    exit(-1);
  }
  while ((n=read(fd, buf,10))>0)
    write(1,buf,n); /*scrittura su stdout */
  close(fd);
}
```

```
#include <fcntl.h>
#include <stdio.h>
#define BUFDIM 1000
#define perm 0777
main (int argc, char **argv){
  int status;
  int infile, outfile, nread;
  char buffer[BUFDIM];
  if (argc != 3){
    printf (" errore \n"); exit (1);
  }
  if ((infile=open(argv[1], O_RDONLY)) <0){
    perror("apertura sorgente: ");
    exit(1);
  }
  if ((outfile=creat(argv[2], perm )) <0){
    perror("apertura destinazione:");
    close (infile); exit(1);
  }
  while((nread=read(infile, buffer, BUFDIM)) >0 ){
    if(write(outfile, buffer, nread)< nread){
      close(infile);
      close(outfile);
      exit(1);
```

```
    }
  }
  close(infile);
  close(outfile);
  exit(0);
}
```

▼ readdir() → *6.* Legge un direttorio

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

→ `dir` è il puntatore al direttorio da leggere (aperto attraverso `opendir()`)

La funzione restituisce:

→ `NULL` in caso di insuccesso.

→ Un puntatore (diverso da `NULL`) se la lettura ha avuto successo.

`descr` punta a una struttura di tipo `dirent`, dichiarata in `dirent.h`.

▼ Struttra di tipo `dirent`

```
struct dirent {
  long d_ino; /* i-number */
  off_t d_off; /* offset del prossimo */
  unsigned short d_reclen; /* lunghezza del record */
  unsigned short d_namelen; /* lunghezza del nome */
  char *d_name; /* nome del file */
}
```

```
#include <stdlib.h>
#include <sys/types.h>
#include <dirent.h>
#include <fcntl.h>
void miols(char name[]){
  DIR *dir; struct dirent * dd;
  char buff[80];
  dir = opendir (name);
  while ((dd = readdir(dir)) != NULL){
    sprintf(buff, "%s\n", dd->d_name);
    write(1, buff, strlen(buff));
  }
  closedir (dir);
  return;
}

main (int argc, char **argv){
  if (argc <= 1){
    printf("Errore\n");
    exit(1);
  }
  miols(argv[1]);
  exit(0);
}
```

▼ receive() → Consente la comunicazione tra più processi

▼ Consumazione simmetrica

Il destinatario fa il naming esplicito del mittente

Processo del produttore P

```
pid C =....;
main(){
```

Processo del consumatore C

```
pid P=....;
main(){
```

```
  msg M;
  do{
    produco(&M);
    ...
    send(C, M);
  }while(!fine);
}
```

```
  msg M;
  do{
    receive(P, &M);
    ...
    consumo(M);
  }while(!fine);
}
```

▼ **Comunzazione asimmetrica**

I*l destinatario non è obbligato a conoscere l’identificatore del mittente: la variabile id raccoglie l’identificatore del mittente. (es. Modello client-server)*

Processo del produttore P

```
pid C =....;
main(){
  msg M;
  do{
    produco(&M);
    ...
    send(C, M);
  }while(!fine);
}
```

Processo del consumatore C

```
id P=....;
main(){
  msg M;
  do{
    receive(&id, &M);
    ...
    consumo(M);
  }while(!fine);
}
```

▼ reply() → Consente la comunicazione tra più processi

```
reply(P, answ)
```

▼ send() → Consente la comunicazione tra più processi

Il destinatario può essere specificato in più modi:

→ Comunicazione diretta: al messaggio viene associato l’identificator del processo destinatario (naming esplicito).

```
send(Proc, msg)
```

→ Comunicazione indiretta: il messaggio viene indirizzato a una mailbox (contenitore di messaggi) dalla quale il destinatario preleverà il messaggio.

```
send(Mailbox, msg)
```

▼ **Consumazione simmetrica**

Il destinatario fa il naming esplicito del mittente

Processo del produttore P

```
pid C =....;
main(){
  msg M;
  do{
    produco(&M);
    ...
    send(C, M);
  }while(!fine);
}
```

Processo del consumatore C

```
pid P=....;
main(){
  msg M;
  do{
    receive(P, &M);
    ...
    consumo(M);
  }while(!fine);
}
```

▼ **Comunzazione asimmetrica**

Il destinatario non è obbligato a conoscere l'identificatore del mittente: la variabile id raccoglie l'identificatore del mittente. (es. Modello client-server)

Processo del produttore P

```
pid C =....;
main(){
  msg M;
  do{
    produco(&M);
    ...
    send(C, M);
  }while(!fine);
}
```

Processo del consumatore C

```
id P=....;
main(){
  msg M;
  do{
    receive(&id, &M);
    ...
    consumo(M);
  }while(!fine);
}
```

▼ signal() → Associa ad ogni segnale il rispettivo handler

```
#include <signal.h>
void (* signal(int sig, void (*handler)()))(int);
```

Deve essere messa all'inizio del main, assegna ad ogni segnale l'handler (una funzione).

```
#include <signal.h>
void gestore(int);
...
main(){
  ...
  signal(SIGUSR1, gestore); /*SIGUSR1 gestito */
  ...
  signal(SIGUSR1, SIG_DFL); /*USR1 torna a default */
  signal(SIGKILL, SIG_IGN); /*errore! SIGKILL non è ignorabile */
  ...
}
```

```
/* file segnali1.c */
#include <signal.h>

void handler(int);

main(){
  if (signal(SIGUSR1, handler)==SIG_ERR)
    perror("prima signal non riuscita\n");
  if (signal(SIGUSR2, handler)==SIG_ERR)
    perror("seconda signal non riuscita\n");
  for (;;);
}

void handler (int signum){
  if (signum==SIGUSR1) printf("ricevuto sigusr1\n");
  else if (signum==SIGUSR2) printf("ricevuto sigusr2\n");
}
```

Un figlio eredita le stesse signal del padre, possono essere usate nuove funzioni **signal()** per sovrascrivere quelle precedenti.

Al posto dell'handle è possibile utilizzare:

→ **SIG_DFL**: segnale di default.

→ **SIG_IGN**: ignora il segnale.

▼ sleep() → Provoca la sospensione del processo

```
unsigned int sleep(unsigned int N)
```

Mette in pausa il processo per `N` secondi, il processo è comunque sensibile ai segnali, e, nel caso ne riceva uno, interrompe la funzione di **sleep()**.

```
#include <signal.h>

void stampa(int signo){
  printf("risvegliato dal segnale %d !!\n", signo);
}

main(){
  int k;
  signal(SIGUSR1, stampa);
  k=sleep(1000);
  printf("Mancavano ancora %d sec..\n", k);
  exit(0);
}
```

▼ stat() → *6*. Per leggere gli attributi di un file.

```
#include <sys/stat.h>
int stat(const char *path, struct stat *buf);
```

→ `path` rappresetna il nome del file.

→ `buf` è il puntator ea una struttura di tipo `stat`, nella quale vengono restituiti gli attributi del file (definito nell'header file `<sys/stat.h>`)

▼ Struttura `stat`

N.B. La struttura seguente non è definita nello stesso modo in ogni sistema.

```
struct stat {
  dev_t st_dev; /* ID of device containing file */
  ino_t st_ino; /* i-number */
  mode_t st_mode; /* protection & file type */
  nlink_t st_nlink; /* number of hard links */
  uid_t st_uid; /* user ID of owner */
  gid_t st_gid; /* group ID of owner */
  dev_t st_rdev; /* device ID (if special file) */
  off_t st_size; /* total size, in bytes */
  blksize_t st_blksize; /* blocksize for file system I/O */
  blkcnt_t st_blocks; /* number of blocks allocated */
  time_t st_atime; /* time of last access */
  time_t st_mtime; /* time of last modification */
  time_t st_ctime; /* time of last status change */
};
```

Per interpretare i valori di `st_mode` sono presenti delle macro in `<sys/stat.h>`:

- `S_ISREG(mode)`: è un file regolare? (flag `S_IFREG`)
- `S_ISDIR(mode)`: è una directory? (flag `S_IFDIR`)
- `S_ISCHR(mode)`: è un dispositivo a caratteri (file speciale)? (flag `S_IFCHR`)
- `S_ISBLK(mode)`: è un dispositivo a blocchi (file speciale)? (flag `S_IFBLK`)

```
/* Invocazione: provastat nomefile */
#include <sys/types.h>
#include <sys/stat.h>
#include <time.h>
#include <stdio.h>
#include <stdlib.h>
main(int argc, char *argv[]){
```

```
  struct stat sb;
  if (argc != 2) {
    fprintf(stderr, "Usage: %s <pathname>\n", argv[0]);
    exit(1);
  }
  if (stat(argv[1], &sb) == -1) {
    perror("stat");
    exit(1);
  }
  printf("Tipo del file:\t");
  if (S_ISREG(sb.st_mode)) printf("file ordinario\n");
  if (S_ISBLK(sb.st_mode)) printf("block device\n");
  if (S_ISCHR(sb.st_mode)) printf("character device\n");
  if (S_ISDIR(sb.st_mode)) printf("directory\n");
  printf("I-number:\t%ld\n", (long) sb.st_ino);
  printf("Mode:\t%lo (octal)\n", (unsigned long) sb.st_mode);
  printf("numero di link:\t%ld\n", (long) sb.st_nlink);
  printf("Proprietario:\tUID=%ld GID=%ld\n", (long) sb.st_uid,
  (long) sb.st_gid);
  printf("I/O block size:\t %ld bytes\n", (long) sb.st_blksize);
  printf("dimensione del file:\t%ld bytes\n", (long) sb.st_size);
  printf("Blocchi allocati: \t%ld\n”,(long ) sb.st_blocks);
  exit(0);
}
```

```
$ ./provastat pippo.txt
Tipo del file: file ordinario
I-number: 13900906
Mode:0644 (octal)
numero di link:1
Proprietario: UID=503 GID=503
I/O block size: 4096 bytes
dimensione del file:1040 bytes
Blocchi allocati: 8
```

▼ unlink() → *6*. Per decrementare il numero di link del file

```
int unlink(char *name);
```

→ `name` è il nome del file

Ritorna 0 se OK altrimenti `-1`.

▼ wait() → Aspettare la terminazione di un figlio.

```
#include<sys/wait.h>
int wait(int *status);
```

Il padre può rilevare lo stato di terminazione attraverso la system call **wait().**

Il parametro è l’indirizzo della variavile in cui viene memorizzato lo stato di terminazione del figlio, **wait()** restituisce il pid del processo terminato.

Il processo che la chiama può avere figli in esecuzione:

→ Se tutti i figli non sono ancora terminati il processo si sospende in attesa della terminazione del primo di essi.

→ Se almeno un figlio F è già terminato ed il suo stato non è ancora stato rilevato (F zombie), **wait()** ritorna immediatamente con il suo stato di terminazione (nella variabile status).

→ Se non esiste neanche un figlio, **wait()** non è sospensiva e ritorna un codice di errore (< 0).

Variabile *status* 16bit:

→ Se il byte meno significativo di status è zero il più significativo rappresenta lo stato di terminazione (terminazione volontaria).

→ In caso contrario il byte meno significativo di status descrive il segnale che ha terminato il figlio (terminazione involontaria).

```
main(){
  int pid, status;
  pid=fork();
  if (pid==0){
    printf(“figlio”);
    exit(0);
  }
  else{
    pid = wait(&status);
    printf(“terminato processo figlio n.%d”, pid);
    if ((char)status==0)
      printf(“term. volontaria con stato %d”, status>>8);
    else
      printf(“terminazione involontaria per segnale %d\n”, (char)status);
  }
}
```

```
#include <sys/wait.h>
#define N 100
int main(){
  int pid[N], status, i, k;
  for (i=0; i<N; i++){
    pid[i]=fork();
    if (pid[i]==0){
      printf(“figlio: il mio pid è: %d”, getpid());
      exit(0);
    }
  }
  for (i=0; i<N; i++){ /* attesa di tutti i figli */
    k=wait(&status);
    if (WIFEXITED(status))
      printf(“Term. volontaria di %d con stato %d\n”, k,WEXITSTATUS(status));
    else if (WIFSIGNALED(status))
      printf(“term. Involontaria di %d per segnale %d\n”,k, WTERMSIG(status));
}
```

▼ write() → *4.* Scrive un file.

```
int write(int fd,char *buf,int n);
```

→ `fd` è il file descriptor del file

→ `buf` è l’area da cui trasferire i byte scritti

→ `n` è il numero di caratteri da scrivere

Il caso di successo restituisce un intero positivo (≤ `n`)che rappresenta il numero di caratteri effettivamente scritti.

```
#include <fcntl.h>
main(){
  int fd,n;
  char buf[10];
  if((fd=open(“/home/miofile”,O_RDONLY))<0){
    perror(“errore di apertura:”);
    exit(-1);
  }
  while ((n=read(fd, buf,10))>0)
    write(1,buf,n); /*scrittura su stdout */
```

```
  close(fd);
}
```

```
#include <fcntl.h>
#include <stdio.h>
#define BUFDIM 1000
#define perm 0777
main (int argc, char **argv){
  int status;
  int infile, outfile, nread;
  char buffer[BUFDIM];
  if (argc != 3){
    printf (" errore \n"); exit (1);
  }
  if ((infile=open(argv[1], O_RDONLY)) <0){
    perror("apertura sorgente: ");
    exit(1);
  }
  if ((outfile=creat(argv[2], perm )) <0){
    perror("apertura destinazione:");
    close (infile); exit(1);
  }
  while((nread=read(infile, buffer, BUFDIM)) >0 ){
    if(write(outfile, buffer, nread)< nread){
      close(infile);
      close(outfile);
      exit(1);
    }
  }
  close(infile);
  close(outfile);
  exit(0);
}
```

To-Do deadlock Java

In caso si verifichi un deadlock nel programma Java controllare le seguentin cose:

- ***`lock.unlock()`***

  Ogni tanto può capitare di bloccare un processo facendo `lock.lock()` e di dimenticarsi di sloccarlo, generando così uno stallo del programma.

- ***Aggiornamenti delle code***

  All'uscita di un `while` le code devono essere aggiornate diminuendo la lista dei thread in attesa e aumentando la lista dei thread che stanno usando la risorsa. Attenzione a modificare le liste giuste!

- ***Le condizioni del*** ***`while`***

  Se, controllati i punti precedenti il programma continua a stallare, è bene dare un'occhiata alle condizioni del while. Ricordarsi che tutte le condizioni **devono essere false affinchè il programma prosegua**.

- ***Condizioni delle*** ***`signal()`***

  Infine come ultima cose si provi a controllare le condizioni e l'ordine delle signal che vengono fatte all'uscita di un thread dalla risorsa. Si ricordi che va usato `signal()` quando è necessario risvegliare **un solo thread**, mentre va usato `signalAll()` quando è necessario svegliare un numero indefinito dei thread dalla coda.

- ***Attenzione agli indici!***

  Nel monitor spesso si lavora con array di interi o condizioni, è facile sbagliare e dimenticarsi di sostituire un indice e aumentare o diminuire valori che non dovrebbero essere modificati in quel punto.

Se le cose non funzionano è consigliato qundi di andare a fare un *double-check* sui frammenti di codice che riferiscono ad array, in particolare a quelli copiati e incollati!

- ***`printStatus()`***

  In ogni caso è consigliato fare una funzione privata `printStatus()` nel monitor che stampa lo stato (numero di thread in attesa e numero di thread che stanno utilizzando la risorsa) a ogni modifica. Questo permette di avere una visione diretta su ciò che accade nella risorsa condivisa del multithread.

  Si ricordi di stampare lo stato **dopo** aver aumentato/diminuito i valori delle variabili.

- ***Come debuggare?***

  Oltre al `printStatus()` può essere utile iniziare a testare il programma con un numero di Thread molto ridotto (una decina circa). Se il programma termina correttamente si provi ad aumentare progressivamente i thread, altrimenti si cerchi di riguardare i possibili errori elencati nei punti precedenti.

Template Fedele Penna

Il seguente codice - template contiene tutte le funzioni di base utili ai fini del superamento della parte di System Call all'esame.

A cura di **Fedele Penna** (*fedele.penna@studio.unibo.it*)

```
#include <fcntl.h>
#include <stdio.h>
#include <unistd.h>
#include <signal.h>
#include <stdlib.h>
#include <string.h>
#include <sys/wait.h>
#include <ctype.h>
#include <string.h>
#include <sys/stat.h>

/*
    file con funzioni per l'esame di Sistemi Operativi
    @author fedele.penna@studio.unibo.it
*/


void wait_child();

int pp[2]; //pipe

/*
    *** DICHIARAZIONI FUNZIONI DI CONTROLLO DATI ***
*/
void checkArgs(int argc, int rightArgs);
int checkNumber(char str[], int condition);
int isValidAbsoluteFilePath(const char* path);
int isAbsolutePath(const char* path);
int doesFileExist(const char* path);
int isChar(const char* argument); //a-z A-Z


/*
    ****************************************************
    MAIN
    ****************************************************
*/

int main(int argc, char* argv[]){
    int n_arg = 3; //inserire numero di argomenti aspettati
    checkArgs(argc, n_arg); //controllo argc
    printf("%d\n", checkNumber(argv[1], 1)); //ricorda la condizione
```

```
    int pid[2];
    for(int i=0; i<2; i++){
        pid[i]=fork();
        if(pid[i]<0){
            perror("Errore fork\n");
            exit(1);
        } else if(pid[i]==0){
            if(i==0){ //P1

            }
            else{ //P2

            }
        }
    }


}

/*
    *** BLOCCO FUNZIONI DI SYSTEM CALL E ESAME  ***
*/

void wait_child() {
    int pid_terminated,status;
    pid_terminated=wait(&status);
    if(WIFEXITED(status))
        printf("\nP0: terminazione volontaria del figlio %d con stato %d\n",
                pid_terminated, WEXITSTATUS(status));
    else if(WIFSIGNALED(status))
        printf("\nP0: terminazione involontaria del figlio %d a causa del segnale %d\n",
                pid_terminated,WTERMSIG(status));
}


/*
    *** BLOCCO FUNZIONI DI CONTROLLO DATI ***
*/



void checkArgs(int argc, int rightArgs){
    if(argc!=rightArgs){
        perror("Errore numero argomenti\n");
        exit(EXIT_FAILURE);
    }
}

/*      controlla se l'INTERA stringa è composta da numeri
        restituisce il numero
        (opzionale) su quel numero fa la condition
    0 -> nessuna condizione
    1 -> positivo compreso zero
    2 -> positivo
    -1 -> negativo
    -2 -> negativo compreso zero
*/
int checkNumber(char str[], int condition){
    int num=0;

    //disabilitare per numeri in ingresso negativi
    //oppure imporre i = 1 ma attenzione!
    for (int i = 0; str[i] != '\0'; i++) {
        if (!isdigit(str[i])) {
            fprintf(stderr, "Non è un numero!\n");
            exit(EXIT_FAILURE);
             // Il carattere corrente non è un numero
        }
    }

    num=atoi(str);
    switch (condition){
            case 0: break;
            case 1: if(num<0) { perror("Il numero non è positivo\n"); exit(-2); } break;
            case 2: if(num<=0) { perror("Il numero non è positivo\n"); exit(-2); } break;
            case -1: if(num>0) { perror("Il numero non è negativo\n"); exit(-2); } break;
            case -2: if(num>=0) { perror("Il numero non è negativo\n"); exit(-2); } break;
            default: perror("Condition errata\n");

    }
    return num;
```

```
}



int isAbsolutePath(const char* path) {
    if (path == NULL) {
        return 0; // La stringa è nulla
    }

    char resolvedPath[4096];
    if (realpath(path, resolvedPath) != NULL) {
        printf("%s è assoluto\n", path);
        return 1; // Il percorso è assoluto
    }

    return 0; // Il percorso non è assoluto
}

int doesFileExist(const char* path) {
    struct stat fileStat;
    if (stat(path, &fileStat) == 0 && S_ISREG(fileStat.st_mode)) {
        return 1; // Il file esiste ed è un file regolare
    }
    printf("Il file cercato non esiste\n");
    return 0; // Il file non esiste o non è un file regolare
}

int isValidAbsoluteFilePath(const char* path) {
    if (!isAbsolutePath(path)) {
        return 0; // Il percorso non è assoluto
    }

    if (!doesFileExist(path)) {
        return 0; // Il file non esiste
    }

    return 1; // Il percorso è un path assoluto valido con un file esistente
}

int isChar(const char* argument) {
    size_t length = strlen(argument);
    if (length != 1) {
        return 0; // La lunghezza non è 1, quindi non rappresenta un char
    }

    char firstChar = argument[0];
    if (!isalpha(firstChar)) {
        return 0; // Il primo carattere non è una lettera dell'alfabeto a-z A-Z, quindi non rappresenta un carattere
    }

    return 1; // La lunghezza è 1 e il primo carattere è una lettera, quindi rappresenta un carattere
}
```