Sistemi Operativi

Pietro Lechthaler, Lorenzo Canciani

Sommario

Introduzione

Parametri posizionali
L'ordinamento è importante, il comando mostrerà prima gli elementi della cartella /tmp e successivamente della cartella /etc.

```
ls /tmp /etc
```

Parametri nominali (flags)
Tipicamente identificati con una lettera oppure in forma completa.

```
ls -a -l
ls -al
ls --help //mostra informazioni comando
```

Utilizzo comandi su più linee

```
ls \
> comando 1 \
> comando 2
```

Comandi fondamentali:

clear	cancellare terminale	**pwd**	path completo
ls	mostra contenuto cartella	**cd**	spostamento tra cartelle
wc	Conta parole, spazi, caratteri doc	**date**	data attuale
cat	concatenazione di file stdout/ mostrare output	**echo**	creare testo e mandare in output
alias	creazione alias di comandi	**echo > file** **echo >> file**	sovrascrive file aggiunge a file
file	informazioni su file	**chown/chmod** **chmod -R(dir e file)**	permessi e proprietari
cp **cp -r**	copiare file copiare cartella	**mv [SOURCE] [DEST]**	muovere file e cartelle
type	informazioni su singolo comando	**grep**	ricerca interno di files (evidenzia)
truncate [dim] [file]	ridurre o aumentare dimensione file	**function [NOME]** **{** **comandi** **}**	definizione funzione esecuzione: > [NOME]
read	variabile in maniera interattiva	**rm** rm -f -r	eliminare elimina ricorsivamente cartelle

Redirezionamento canali:

```
#1=output 2=error
#>[percorso] (sovrascrive ogni volta)
ls 1>/tmp/out.txt 2>/tmp/err.txt

# >>[percorso] (append - scrive in coda)
ls 1>> file.txt //appena stdout in file

#invertire i canali
ls 1>&2 2>&1

#ignora canale
ls 1>/dev/null
```

Redirezionamento input:

```
# input contenuto file
# < [percorso]
# <> [percorso] (aperto in read-write)
grep ciao < file.txt

# [comando] << [stringa] (input interattivo fino a “stringa”)
wc << stringa
> [testo in input]
> [stringa] (→ stop)
```

Variabili:

```
#creazione variabile locale
VAR=valore

#creazione variabile accessibile a tutti i sottoprocessi
export VAR=valore

#output variabile
echo $var

#output che permette di concatenare stringhe
#senza parentesi echo cercherebbe var2
echo ${var}2
>valore2
```

Variabili $$ e $?:

```
#id processo
echo $$

#codice di ritorno comando eseguito (da 0 a 255)
echo $?
```

Variabili di sistema:

SHELL	riferimento shell corrente	**PATH**	contiene tutte var ambiente
TERM	tipologia terminale corrente	**PWD**	contiene cartella corrente
PS1	contiene prompt e setup	**HOME**	cartella principale utente

Array:

```
#creazione array
lista = (“[str1]” [int2])

#output completo
echo ${lista[@]}

#output singolo elemento
#prima posizione = 0
echo ${lista[x]}

#lista indici
echo ${!lista[@]}

#concatena al primo elemento array (inserisce nel primo)
lista+=[elemento]

#inserisce nell’ultima posizione (aggiunge un elemento)
lista+=([elemento])

#dimensione array
echo ${#lista[@]}

#looping array
for i in ${!lista[@]} ;
   echo ${lista[$i]}
done

#sottolista da pos a pos+lunghezza
echo ${lista[@]:pos:lunghezza}
```

Concatenazione comandi

```
#semplice, eseguito in ordine
[cmd1] ; [cmd2]

#logica and, se cmd1 fallisce (cod. err. !=0) → cmd2 non viene eseguito
[cmd1] && [cmd2]

#logica or, se cmd1 fallisce (cod. err. !=0) → cmd2 viene eseguito
[cmd1] || [cmd2]
```

Concatenazione con piping

```
#cattura solo stdout
ls | wc -l

#cattura solo stdout e stderr
ls |& wc -l
```

Subshell (sottoambiente)

```
#cattura output comando subshell
$([comando])

#esempio
#prima viene eseguito comando dentro parentesi
#l'output viene catturato dal comando esterno che lo stampa
echo $(echo ciao)
```

>ciao

Espansione aritmetica

```
#utilizzare operatori aritmetici, evitando errori
echo $(( 1 < 7 ))
>0 → eseguito con successo (vero)
>1 → non eseguito con successo (falso)

#possibile utilizzare anche il comando test
test 1 -lt 0
echo $?
```

>1

Confronti logici

INTERI	[..]	((..))
uguale a	-eq	==
diverso da	-ne	!=
minore di **minore uguale di**	-lt -le	< <=
maggiore di **maggiore uguale di**	-gt -ge	> >=

STRINGHE	[..]	((..))
uguale a	= o ==	
diverso da	!=	
minore di **(ordine alfabetico)**	\<	<

maggiore di (ordine alfabetico)	\>	>

Nota: importante lo spazio tra parentesi e operatori

OPERATORI UNARI	
[[-f [percorso]]]	esistenza file
[[-e [percorso]]]	esistenza
[[-d [percorso]]]	esistenza cartella
[[! …]]	negazione

while (finchè valore di lunghezza diverso da 0)

```
#!/bin/bash
num_arg=$#
while [[ -n $valore ]]; do
   echo $valore
done
```

while

```
num_arg=$#
while [[ condizione ]]; do
   comando
done
```

Terminale: while [[condizione]]; do comando; done

if

```
if [ condizione ]; then
    comando
else

fi
```

Terminale: if [condizione]; then comando; else comando2; fi

for

```
for i in ${!lista[@]} ; //scorre tutti gli indici dal primo
   echo ${lista[$i]}
done
```

Terminale: for i in *; do echo $i; done

for: inversione una lista

```
ordinata=( $@ ) ; inversa=()
for i in ${!ordinata[@]}; do
    inversa=( "${ordinata[$i]}" "${inversa[$@]}" )
done
```

Bash

Esecuzione script - Opzione 1

```
bash file.sh
```

Esecuzione script - Opzione 2

```
chmod +x file.sh
./file.sh
```

All'interno del file dovrà essere presente l'hash-bang

```
#!/bin/bash
echo $BASHPID
echo $(echo $BASHPID)
```

Variabili "ambiente" bash

$@	insieme parametri passati al file	**$#**	numero parametri passati al file
$n	parametro n passato al file	**$0**	nome file/programma
$BASHPID	id shell corrente		

while con stampa parametri passati al file .sh:

```
#!/bin/bash
num_arg=$#
while [[ $1 != "" ]]; do
   echo "ARG=$1"
   shift
done
```

Viene stampato il primo valore passato come argomento, dopodichè viene spostato il secondo argomento come primo, e così via fino all'ultimo argomento. Dopo lo spostamento il valore dell'argomento precedente viene perso (consiglio: salvare in lista).

Makefile

Terminale:

```
> make [targets] [eventuali parametri]
```

Regola:

```
targets: prerequisites
   command
   command
   command
```

targets. Nome dei file, separati da spazi. tipicamente uno per regola.
commands. Serie di comandi.
prerequisites. anche loro sono dei nomi di file, separati da spazi. Se questi file esistono, vengono eseguiti i comandi del target.

Esempio:

```
some_file: other_file
    touch some_file
other_file:
    echo "nothing"

_____________ terminale___________________

>echo “nothing”
>nothing
>touch some_file
```

Variabili in make:

```
FILE = nome_preimpostato
some_file:
   echo "Look at this variable: " $(FILE)
   touch some_file

# $(files) stampa tutta la stringa files
```

Se da terminale eseguo: >make FILE=file.txt , la variabile interna al make viene sovrascritta.

Target speciali:

```
#making multiple targets
all: one two three
one:
    touch one
two:
    touch two
```

```
three:
    touch three
#remove the output of the target
clean:
    rm -f one two three
```

Wildcard *:

```
# Print out file information about every .c file
print: $(wildcard *.c)
    ls -la  $?
```

* cerca nel filesystem dei filename. Può essere usato nei target, prerequisiti o nella funzione wildcard.

Wildcard %:

```
In un target, % sostituisce qualunque stringa. In un prerequisito corrisponde alla stringa sostituita nel target.
```

Shell:

```
$(shell ...): cattura l'output di un commando shell
```

Eval:

```
$(eval ...): consente di creare nuove regole make dinamiche
```

Variabili automatiche:

$@	nome target in cui è inserito il comando	**$<**	primo prerequisito del target
$^	lista prerequisiti del target	**$(VARIABILE)**	ottiene valore variabile

Target speciali:

.PHONY

```
clean:
    rm -f some_file
    rm -f clean
.PHONY: clean
```

Previene che il make confonda il phony target con un nome.

.INTERMEDIATE

```

```

.SECONDARY

```

```

Codice C

Librerie standard:

stdio.h	FILE, EOF, stderr, stdin, stdout, fclose(), etc...
stdlib.h	atof(), atoi(), malloc(), calloc(), free(), exit(), system(), rand(), etc...
string.h	memset(), memcpy(), strncat(), strcmp(), strlen(), etc...
unistd.h	STDOUT_FILENO, read, write, fork, pipe, etc...
fcntl.h	creat, open, etc...
math.h	sin(), cos(), sqrt(), floor(), etc...

Stampare su canali standard:

```
fprintf(stdout, “testo”)
fprintf(stderr, “testo”)
```

Funzione isatty:

```
//dato un file descriptor determina se è legato ad un terminale
int isatty(int fildes);
```

ritorna 1 se il fd è un terminale, 0 altrimenti (esempio pipe).

Conversione da stringa ad intero:

```
int numero = atoi(stringa);
```

Argomenti main:

argc	numero totale argomenti passati al main	**argv**	lista di stringhe
argv[0]	nome dell’eseguibile	**argv[1][0]**	primo carattere del primo argomento

Verificare presenza parametri passati al main:

```
if(argc<2){
 //nessun parametro ricevuto
 //possibile gestione errore con return code
}else{
 //parametri ricevuti
}
```

Leggere testo inserito sul prompt:

```
int intero;
float virgola;
char c;

printf("Inserisci un carattere: ");
scanf("%c", &c);

printf("Inserisci un numero intero: ");
scanf("%d", &intero);

printf("Inserisci un numero con la virgola: ");
scanf("%f", &virgola);
```

Comparazione di stringhe

```
if(strcmp(“stringa”, “stringa2”)==0){
      //caratteri corrispondono
}else if(strcmp(“stringa”, stringa)<0){
      //stringa 1 minore di stringa 2
}else if(strcmp(“stringa”, stringa)>0){
      //stringa 1 maggiore di stringa 2
}
```

Copia di una stringa

```
char variabile[10];
strcpy(variabile, “stringa”);
```

Append di due stringhe

```
strcat(destinazione, stringa)
//”destinazionestringa”
```

Cerca carattere in stringa

```
strchr(stringa,C);
//Ritorna un puntatore alla primo carattere C nella stringa
```

Restituisce grandezza stringa

```
strlen(stringa);
```

> *Streams*

Apertura

```
#include <stdio.h>

//apertura file con permessi
FILE *ptrfile;
ptrfile=fopen(“filename.txt”, “r+”);
```

Chiusura

```
fclose(ptrfile)
```

Lettura, funzione feof(*ptrfile*):

```
while (!feof(ptrfile)){
    fscanf(ptrfile, “%s”, contenuto);
    printf(“%s\n”, contenuto);
}
```

Leggere e stampare a terminale contenuto file fino a EOF.

Scrittura, fprintf

```
fprintf(ptrfile, “contenuto da scrivere”);
```

Resettare puntatore all’inizio del file

```
rewind(ptrfile);
```

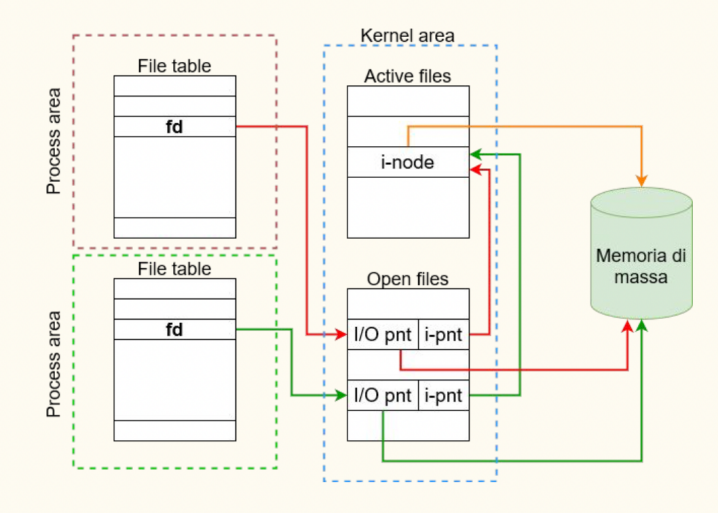
Restituire N caratteri in una stringa

```
//salva 10 caratteri dalla posizione del puntatore in stringa
fgets(stringa, 10, ptrfile);
```

Restituire prossimo carattere disponibile o EOF

```
//restituisce prossimo carattere dopo pos puntatore
char i = fgetc(stringa, 10, ptrfile);
```

> *File Descriptors*



Creazione file

```
int file = creat(“file.txt”, S_IRUSR | S_IWUSR);
```

S_IRUSR = permesso lettura ; S_IWUSR = permesso scrittura

Aprire un file

```
int file = open(“pathname”, O_CREAT | O_RDWR | O_EXCL, mode);
```

mode = quando il file non esiste, permessi di creazione: S_IRUSR | S_IWUSR

O_CREAT = se non esiste il file viene creato con permessi **mode**
O_RDWR = scrittura e lettura (default lo fa già)
O_EXCL = usato con O_CREAT, se il file esiste torna un errore.
O_RDONLY = solo lettura
O_WRONLY = solo scrittura
ritorna valore <0 se fallisce

Leggere un file

```
int bit_lettura; char stringa[10]; int num_bit=10;
do{
     bit_lettura= read(file, stringa, num_bit);
}while(bit_lettura>0);
```

```
read(fd, buffer, sizeof(buffer))
```

read ritorna il numero di caratteri letti. In un ciclo viene utilizzato per inserire nella variabile stringa il contenuto di N bit alla volta.
Importante è la posizione della testina.

Scrivere su file

```
int fd_file = open(“file.txt”, O_CREATE | O_RDWR, S_IRUSR | S_IWUSR);
write (fd_file, “stringa\n”, strlen(“stringa\n”));
```

L’operazione di scrittura in questo caso sposta la testina dopo il contenuto inserito.

Spostamento I/O pointer

```
lseek(fd_file, offset, POSIZIONE);
lseek(file, 10, SEEK_SET); //posizione iniziale + 10
```

POSIZIONE:

- SEEK_SET, inizio + (segno) offset
- SEEK_CUR, posizione corrente + (segno) offset
- SEEK_END, fine + (segno) offset

lseek ritorna la nuova posizione della testina, ***utile con SEEK_END e offset 0 per capire la dimensione del file.***

Chiudere un file

```
close(fd_file);
```

Dimensione variabile:

```
sizeof(variabile); //o anche tipo di variabile: sizeof(int);
```

Utile nella funzione read() quando si utilizzano array di caratteri.
char buffer[500]; sizeof(buffer);

fflush:

```
fflush(stdout);
```

Importante usare il fflush se non utilizziamo “\n” poiché stamperebbe solamente alla fine dell’esecuzione del main.

Exit:

```
exit(status);
```

> Piping via Bash

Piping via bash (esempio: ls | ./main.o)

```
int main(){
   char buffer[N];
   //salva in buffer N caratteri provenienti da stdin
   fgets(buffer,sizeof(buf), stdin);
   return 0;
}
```

getchar() e putchar():

```
int main(){
while((c = getchar()) != EOF){
    //legge da stdin
    putchar(stringa) //scrive su stdout
}
```

> *System calls*

Ottenere la data:

```
time_t t_unix = time(NULL);
//è possibile scrivere anche così passando l'indirizzo
time(&t_unix);
//restituisce stringa con tempo corrente e data
ctime($t_unix);
```

Cambiare cartella corrente:

```
chdir("path");
```

Restituisce cartella dove siamo:

```
getcwd(stringa, N_BIT);
//se path maggiore di N_BIT ritorna NULL
```

Alias file descriptors

```
int fd = open("file", O_RDWR);
int fd_copy = dup(fd); //creazione alias di fd
```

Attribuisce al nuovo file descriptor, il più piccolo intero non usato.

Duplicazione file descriptors

```
dup2(oldfd, newfd);
dup2(oldfd, 1); //redirezionato stdout su oldfd
int fd = dup(oldfd); //duplica oldf in fd
```

Crea newfd come copia di oldfd, chiudendo prima newfd se e' necessario. Il vecchio e nuovo file descriptor possono essere utilizzati interscambiabilmente. E' possibile effettuare una lseek() su un file descriptor e ritorvarsi la posizione modificata su entrambi i file descriptor.

Da eseguire con sudo entrambi:
Cambiare chown()

```
fchown(fd, 0, 0);
chown("path", 0, 0);
//owner to root root
```

Permessi, chmod()

```
fchown(fd, S_IRUSR|S_IRGRP|S_IROTH);
chown("path", S_IRUSR|S_IRGRP|S_IROTH);
//cambia privilegi
```

Ottenere valore variabile d'ambiente

```
getenv("NOMEVARIABILE");
```

Impostare variabile d'ambiente

```
putenv("NOMEVARIABILE");
```

System

```
int shell = system("echo questo è un comando");
```

Restituisce un intero che equivale all'esito.
Utilizzare la sintassi di shell.

> *Exec*

---

execv (PATH, ARGOMENTI)

```
char * argv[]={"par1", "par2", NULL};
execv("/home/esercizi/./eseguibile.out", argv);
```

Se esiste ./eseguibile.out, il programma principale finisce l'esecuzione e si prosegue con ./eseguibile.out.
All'interno di ./eseguibile.out la lista dei parametri nella prima posizione non conterrà il nome dell'eseguibile ma il primo parametro.
Se non esiste l'eseguibile prosegue sul programma principale.

execvp (FILE, ARGOMENTI)

```
char * argv[]={"par1", "par2", NULL};
execvp("./eseguibile.out", argv);
```

Il parametro FILE corrisponde ad un file presente nella cartella corrente.

execve (FILE, ARGOMENTI, VARIABILI AMBIENTE)

```
int main( int argc, char *argv[], char *envp[] ){}
char * argv[]={"par1", "par2", NULL};
execve("./eseguibile.out", argv, envp);
```

Vengono passate anche le variabili d'ambiente al file ./eseguibile.out

execl (PATH, ARGOMENTO1, ARGOMENTO2, NULL)

```
execl(“./eseguibile.out”, “<argomento1>”,“<argomento2>”, NULL);
```

execlp (FILE, ARGOMENTO1, ARGOMENTO2, NULL)

```
execlp(“./eseguibile.out”, “<argomento1>”,“<argomento2>”, NULL);
```

execle (FILE, ARGOMENTO1, ARGOMENTO2, NULL, VARIABILI AMBIENTE)

```
execlp(“./eseguibile.out”, “<argomento1>”,“<argomento2>”, NULL, envp);
```

Vengono passate anche le variabili d’ambiente al file ./eseguibile.out

---

> *Forking*

Restituire PID processo attivo:

```
getpid();
```

Restituire PID processo padre:

```
getppid(); //Il figlio NON DEVE essere lasciato orfano!!
```

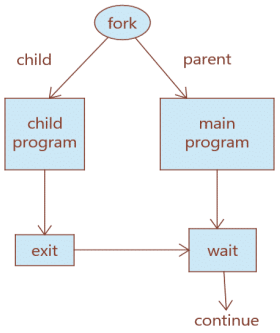
Fork:

```
int fork=fork();

if(fork==0){
   //figlio
}else{
   //padre
}
```

Restituisce -1 se c’è stato un errore.
Altrimenti:

- restituisce PID processo figlio nel caso del padre
- restituisce 0 nel caso del figlio



wait:

```
int status;
```

```
wait(&status);
while(wait(&status)>0); //aspetta i tutti i figli terminino
```

>0 → se i figli sono ancora in vita
0 → nel momento in cui tutti i figli sono terminati
-1 → errore

waitpid:

```
waitpid(OPZIONE, &status, 0);
```

OPZIONE:
- -gruppo → attende tutti i proc con gruppo “gruppo”
- -1 → attende un figlio qualunque
- 0 → stesso gruppo padre
- pid → attende pid specifico

>0 → se i figli sono ancora in vita
0 → nel momento in cui i processi sono terminati in base alla selezione
-1 → errore

Interpretazione stato:

```
waitpid(OPZIONE, &status, 0);
printf(“%d”, OPZIONE);
```

OPZIONE:
- **WEXITSTATUS(status)**: restituisce lo stato vero e proprio (ad esempio il valore usato nella “exit”)
- **WIFCONTINUED(sts)**: true se il figlio ha ricevuto un segnale SIGCONT
- **WIFEXITED(sts)**: true se il figlio è terminato normalmente
- **WIFSIGNALED(sts)**: true se il figlio è terminato a causa di un segnale non gestito
- **WIFSTOPPED(sts)**: true se il figlio è attualmente in stato di “stop”
- **WSTOPSIG(sts)**: numero del segnale che ha causato lo “stop” del figlio
- **WTERMSIG(sts)**: numero del segnale che ha causato la terminazione del figlio

---

> Signal

Variabili automatiche:

SIGALARM termina	chiamata di allarme viene utilizzata per impostare un timer che genera il segnale SIGALRM dopo il periodo di	**SIGQUIT**	terminal quit

	timeout.		
SIGCHILD **ignora**	quando figlio termina	**SIGCONT** **ignora**	istruzione programma continuare
SIGKILL **termina**	generato quando deve interrompersi il processo	**SIGSTOP** **termina**	programma deve interrompersi
SIGUSER1/2 **termina**	segnale utente	**SIGINT** **termina**	shell interrompa il processo corrente, CTRL+C
SIGTERM **termina**		**SIGSYM** **termina**	termina con salvataggio di file da analizzare
S **termina**	programma deve interrompersi		

Ignorare un segnale:

```
signal(SEGNALE, SIG_IGN);
```

Signal restituisce -1 se c'è qualche problema, altrimenti restituisce un riferimento all'handler precedentemente assegnato al segnale:

- NULL se handler default
- 1 se era SIG_IGN
- indirizzo handler personalizzato

Handler default segnale:

```
signal(SEGNALE, SIG_DFL);
```

Handler personalizzato segnale:

```
signal(SEGNALE, handler);
```

Handler personalizzato più segnali:

```
void handler(int SEGNALE){
  if(SEGNALE == SIGINT) //comando
  else if(SEGNALE == SIGSTOP) //comando
}
```

Generare dei segnali:

```
kill(pid, SEGNALE);
```

pid:

- >0, inviato al processo con PID=pid
- =0, segnale inviato ogni processo dello stesso gruppo

- =-1 ad ogni processo possibile
- <-1 segnale ad ogni processo del gruppo |pid|

kill restituisce 0 se segnale è inviato, -1 in caso di errore

Generare dei segnali via bash:

```
> kill -l //per vedere il numero di segnale che mi interessa
> kill -NUMERO PID //pid processo interessato
```

Programmare un alarm

```
alarm(5);
```

Programma un allarme che verrà generato dopo N secondi.
Restituisce il numero di secondi che mancano all'**allarme precedente**.
E' possibile settare solo un allarm, quindi quando vado a definire un altro alarm(N), risetto quello precedentemente inizializzato.

Sleep potrebbe resettare degli alarm predefiniti.

"Risvegliare" un programma in pausa:

```
int main(){
  signal (SIGCONT, handler);
  signal (SIGUSR1, handler);
  pause();
}
```

Un programma in pausa continua alla ricezione di un segnale, da prassi si usa il SIGCONT (segnale neutro).

Creazione maschera segnali/lista sig_set:

```
sigset_t maschera;
```

Operazioni principali sui sigset:

```
//aggiungere tutti i segnali
sigfillset(&maschera);

//rimuovere tutti i segnali
sigemptyset(&maschera);

//aggiungere segnale specifico
sigaddset(&maschera, SEGNALE);

//rimuovere segnale specifico
sigdelset(&maschera, SEGNALE);

//check presenza nella maschera: ritorna 1 se c'è, 0 altrimenti.
sigismember(&maschera, SEGNALE);
```

Aggiornare maschera dopo operazioni:

```
sigpromasck(MODALITA, &maschera, &vecchia_maschera);
```

MODALITA:
- SIG_BLOCK → segnali aggiunti alla maschera
- SIG_UNBLOCK → segnali rimossi dalla maschera
- SIG_SETMASK → maschera diventa la maschera

&vecchia_maschera:
- NULL: non vogliamo tenere traccia di come era la maschera prima dell'aggiornamento
- se non nullo, viene salvata vecchia maschera.

Riempire sigset con i segnali pendenti (ricevuti ma non gestiti):

```
sigset_t pendenti;
sigpending(&pendenti);
```

> *Sigaction*

```
struct sigaction {
      void (*sa_handler)(int); //gestore segnale
      void (*sa_sigaction)(int, siginfo_t *, void *); //gestore segnale
      addizionale
      sigset_t sa_mask; //segnali addizionali da bloccare
      int sa_flags;
}
```

Definizione sigaction:

```
struct sigaction sa;
```

Associazione handler a sigaction se non utilizzo informazioni:

```
sa.sa_handler = funzione_handler;
```

Svuotare maschera segnali bloccati, sa_mask:

```
sigemptyset(&sa.sa_mask);
```

Assegnare struttura ad un particolare segnale:

```
sigaction(SEGNALE, &sa, NULL);
```

Definizione sa_sigaction handler (quando utilizzo informazioni):

```
void funzione_handler(int signo, siginfo_t * info, void *empty){
}

//main se utilizzo informazioni
sa.sa_sigaction = funzione_handler;
```

Aggiungere flags

```
sa.sa_flags |= SA_SIGINFO;
```

Ottenere info nell'handler su chi ha mandato il segnale:

```
info->si_pid
```

Bloccare segnale

```
sigprocmask(SIG_BLOCK, &sa.sa_mask, NULL);
```

Sbloccare segnale

```
sigprocmask(SIG_UNBLOCK, &sa.sa_mask, NULL);
```

Ignorare segnale

```
sigprocmask(SIG_IGN, &sa.sa_mask, NULL);
```

Segnali pending

```
sigpending(&sa.sa_mask);
if(sigismember(&sa.sa_mask, SIGTERM)){
   printf("SIGTERM pending\n");
   sigprocmask(SIG_UNBLOCK, &sa.sa_mask, NULL);
}
```

> Process Group

Nel set e get dei gruppi, lo 0 equivale a se stesso.

Settare il process group id:

```
setpgid(0, getpid());
setpgid(processi[i], processi[0]);
//setpgid(pid, gruppo);
```

Get del gruppo processo:

```
getpgid(0);
getpgid(processo);
```

Mandare segnale a gruppi processo:

```
kill(-group1, SEGNALE);
```

Wait gruppi processo:

```
while(waitpid(-group1,NULL,0)>0);
```

> *Errors in C*

Dichiarazione variabile:

```
extern int errno;
```

A questa variabile viene assegnato un valore nel momento in cui si presenta un errore in una funzione.

Stampare errore:

```
perror("Errore");
> Errore : [stringa errore]
```

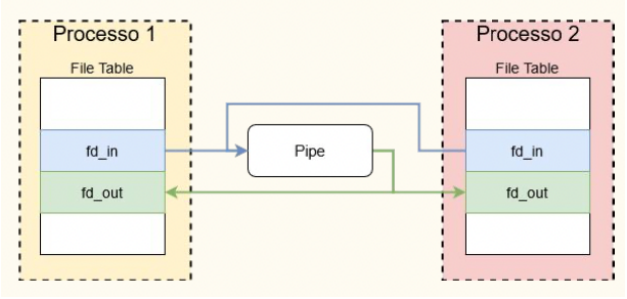
```
fprintf(stderr, "Errore: %s\n", strerror(errno));
> Errore : [stringa errore]
```

Error ha anche un valore quando ha successo, utile per il debug.

Gestione errori:

```
exit(errno);
//o in alternativa una funzione, con switch per gestione errori
```

> *Pipe anonime*



Creazione pipe:

```
#define SCRITTURA 1 #define LETTURA 0
int fd[2];
int esito = pipe(fd);
```

fd[0] lettura ; fd[1] scrittura

Scrittura nella pipe:

```
write(fd[1], “stringa”, LUNGHEZZA_STRINGA);
```

Write restituisce il numero di bytes scritti. Nel caso la parte di lettura fosse chiusa viene inviato un segnale SIGPIPE allo scrittore (default handler → quit).
Consiglio:

```
if(signo==SIGPIPE){
        printf("Errore la pipe è chiusa in lettura!\n");
}
```

E’ possibile non far fallire l’invio di messaggi più grandi del buffer utilizzando: fcntl(fd[1], F_SETFL, O_NDELAY).

Leggere della pipe:

```
char buffer[DIM];
int c = read(fd[0], &buffer, NUM_CARATTERI);
printf(“%s (%d)”, buffer, c);
```

Read ritorna:
- il numero di caratteri che ha letto se funziona correttamente
- 0 se il lato scrittura è stato chiuso ed il buffer è vuoto

Nel caso il buffer fosse vuoto ma il lato scrittura è aperto il processo si sospende fino alla disponibilità dei dati o alla chiusura.

chiudere scrittura o lettura:

```
close(fd[1]);
close(fd[0]);
```

Comunicazione unidirezionale broadcast:

```
int main(int argc, char **argv){

    int fd[4][2];
    printf("[PADRE] %d\n", getpid());
    for(int i=0; i<4; i++){
        pipe(fd[i]);
        int figlio=fork();
        if(figlio==0){
            close(fd[i][1]); //chiusura scrittura
            char buffer[100];
            int c = read(fd[i][0], &buffer, 4);
            printf("[%d] %s (%d)\n", getpid(), buffer, c);
            exit(0);
        }else{
            write(fd[i][1], "ciao", strlen("ciao"));
        }
    }
```

```
return 0;
}
```

In questo caso abbiamo una comunicazione unidirezionale (tipo broadcast) tra padre e figli, tutti ricevono lo stesso messaggio. In questo caso la read non “blocca” tutti i figli nello stesso momento, perchè esiste una pipe per ogni figlio.

Comunicazione bidirezionale :

```
#define LETTURA 0
#define SCRITTURA 1

int main(int argc, char **argv){

    int fd_1[4][2]; //figlio legge
    int fd_2[4][2];  //padre legge
    printf("[PADRE] %d\n", getpid());

    for(int i=0; i<4; i++){
        pipe(fd_1[i]);
        pipe(fd_2[i]);
        int figlio=fork();
        if(figlio==0){
            close(fd_2[i][LETTURA]);
            close(fd_1[i][SCRITTURA]);
            char pid[10];
            sprintf(pid, "%d", getpid());
            //scrive pid su fd_2
            write(fd_2[i][1], pid, strlen(pid));

            char buffer[100];
            int c = read(fd_1[i][LETTURA], &buffer, 4);
            printf("[%d] %s (%d)\n", getpid(), buffer, c);
            exit(0);
        }else{
            close(fd_2[i][SCRITTURA]);
            close(fd_1[i][LETTURA]);
            write(fd_1[i][SCRITTURA], "ciao", strlen("ciao"));
        }
    }

    for(int i=0; i<4; i++){
        char buffer[100];
        int c = read(fd_2[i][LETTURA], &buffer, 8);
        printf("Padre: [%d] %s (%d)\n", getpid(), buffer, c);
    }

return 0;
}
```

In questo caso abbiamo una comunicazione bidirezionale tra padre e figli, tutti possono ricevere e scrivere messaggi.

> *Pipe con nome / FIFO*

utili anche per sincronizzare più processi: read/open bloccante (attenzione al contenuto del file, può essere già scritto).

Creazione fifo:

```
mkfifo(nome_fifo, S_IRUSR|S_IWUSR);
```

S_IRUSR|S_IWUSR nel caso non esistesse. E' possibile utilizzare un file già esistente. Ritorna 0 se va a buon fine, -1 se esiste.

Apertura fifo in modalità Read-only:

```
int fd;
fd = open(nome_fifo, O_RDONLY);
```

Il processo rimane bloccato finché la fifo non viene aperta da entrambi i lati.

Apertura fifo in modalità write-only:

```
int fd;
fd = open(nome_fifo, O_WRONLY);
```

Chiusura lato scrittura/lettura:

```
close(fd);
```

Scrittura fifo:

```
write(fd, "stringa", strlen("stringa"));
```

Se il file non esiste la pipe è vuota dopo la creazione, se esiste già ed è stata utilizzata il contenuto è l'ultimo messaggio scritto.
Il contenuto non viene consumato con la read, la write riporta la testina all'inizio e viene scritto il contenuto. write(ciao); write(a) → "aiao"

Lettura fifo:

```
read(fd, stringa, sizeof(stringa));
```

La lettura non consuma il contenuto, soltanto la riscrittura lo modifica.

> *Queues*

Struct messaggio:

```
 struct messaggio{
    long mtype; //tipo messaggio, identifica la classe
    char mtext[LUNGHEZZA_MESSAGGIO]; //campo testuale
} messaggio;
```

Struct dettagli coda (non vanno implementate ma soltanto dichiarate):

```
struct msqid_ds mod;
struct ipc_perm mod2;
```

Struct info coda:

```
struct msqid_ds {
    struct ipc_perm msg_perm; /* Ownership and permissions */
    time_t msg_stime; /* Time of last msgsnd(2) */
    time_t msg_rtime; /* Time of last msgrcv(2) */
    time_t msg_ctime; // tempo ultima modifica da msgctl
    unsigned long msg_cbytes; //bytes utilizzati nella coda
    msgqnum_t msq_qnum; //num messaggi
    msglen_t msg_qbytes; //max num bytes coda
    pid_t msg_lspid; //pid ultimo send
    pid_t msg_lrpid; //pid ultimo recev
}
```

```
struct ipc_perm {
    key_t __key;    /* Key supplied to msgget(2) */
    uid_t uid;      /* Effective UID of owner */
    gid_t gid;      /* Effective GID of owner */
    uid_t cuid;     /* Effective UID of creator */
    gid_t cgid;     /* Effective GID of creator */
    unsigned short mode;    /* Permissions */
    unsigned short __seq;   /* Sequence number */
};
```

Set tipo messaggio:

```
messaggio.mtype = 20;
```

Set messaggio:

```
strcpy(messaggio.mtext,”stringa”);
```

Ottenere chiave univoca:

```
key_t chiave_univoca = ftok(“/tmp/unique”, 1);
```

Parametri:

- path cartella o file esistente ed accessibile
- intero

Restituisce una chiave univoca basandosi su questi due parametri.
Se fallisce, ritorna -1.
Altrimenti: key_t chiave = N;

Creazione coda:

```
key_t chiave_coda = ftok(“path”,1);
int id_coda = msgget(chiave_coda, 0777 | IPC_CREAT | IPC_EXCL);
```

Etichette:

- 0777 → permessi, equivale a tutti
- IPC_CREAT → creazione qualora non esistesse
- IPC_EXCL → viene utilizzato unicamente insieme a IP_CREAT: fa fallire la richiesta qualora esistesse già la coda.

Le code sono persistenti.
mssget ritorna il valore dell’id coda, se fallisce -1.
Se non esiste e non c’è IPC_CREAT fallisce.

Rimuovere una coda:

```
 msgctl(id_coda, IPC_RMID, NULL);
```

Ritorna -1 se fallisce. Elimina e svuota la coda (se riaperta è vuota).

Inviare messaggi:

```
int esito = msgsnd(id_coda, &messaggio, sizeof(messaggio.mtext),0);
```

FLAG:

- = 0 → resta in attesa finché non c’è spazio **(bloccante)**
- = IPC_NOWAIT → la chiamata fallisce (-1) in assenza di spazio (sizeof(messaggio) + msg.cbytes <= msg.qbytes).

Ricevere messaggi:

```
esito = msgrcv(id_coda, &messaggio, sizeof(messaggio.mtext), N_TYPE,
FLAG);
```

N_TYPE:

- **= n**, FLAG=0 → primo messaggio coda con argomento N
- **= n**, FLAG=020000(MSG_EXCEPT) → primo messaggio disponibile coda con argomento diverso da N
- **= 0**, primo messaggio della coda di qualsiasi argomento.
- **= -n**: primo messaggio il cui tipo argomento è <= n

FLAG:

- =0 → chiamata fallisce (-1) se non c’è abbastanza spazio
- =MSG_NOERROR = messaggio troncato (persa parte eccesso).
- =IPC_NOWAIT = fallire se non ci sono messaggi, **altrimenti chiamata bloccante.**

Ritorna numero caratteri ricevuti, altrimenti -1 se fallisce.

Settare dettaglio della coda:

```
mod.ELEMENTO = VALORE;
```

mod → struct msqid_ds mod;

Get dettaglio della coda:

```
printf(“%d”, mod.ELEMENTO);
```

mod → struct msqid_ds mod;

Modificare dettagli della coda:

```
msgctl(id_coda, COMANDO, &mod);
```

COMANDO:

- IPC_STAT: recupera informazioni da kernel e salva in mod.
- IPC_SET: imposta parametri modificati a seconda di mod.
- IPC_RMID: rimuove immediatamente la coda
- IPC_INFO: recupera informazioni generali sui limiti delle code nel sistema
- MSG_INFO: come IPC_INFO ma con informazioni differenti
- MSG_STAT: come IPC_STAT ma con informazioni differenti

> *Threads*

Compilazione:

```
> gcc -o program main.c -pthread
```

Creazione thread:

```
void *funzione(void *param){
    printf("This is a thread that received %d\n", *(int *)param);
    return (void*)3; //valore arbitrario
    //pthread_exit((void *)valore);
}

int main(){
    pthread_t id_thread; int arg=10; //arg param che passo al thread
    pthread_create(&id_thread, NULL, funzione, (void *)&arg);
    printf("Executed thread with id %ld\n",t_id);
}
```

Terminazione thread:

- valore di ritorno funzione_thread o pthread_exit(void *ritorno);
- quando viene cancellato
- quando thread chiama exit();
- main ritorna un valore (termina tutti i thread);

Cancellazione thread:

```
void *funzione_thread(void *param){
    //Change mode
    pthread_setcanceltype(PTHREAD_CANCEL_DEFERRED, NULL);
```

```
    pthread_setcanceltype(PTHREAD_CANCEL_ASYNCHRONOUS, NULL);
    pthread_setcancelstate(PTHREAD_CANCEL_DISABLE, NULL);
    // **** sezione di codice che non può essere interrotta ****
    pthread_setcancelstate(PTHREAD_CANCEL_ENABLE, NULL);
    //Change type


}

int main(){
    pthread_cancel(id_thread);
}
```

La cancellazione dipende da due parametri MODE e TYPE:

MODE (default=ENABLE):

- DISABLE → non permette la cancellazione
- ENABLE → permette la cancellazione

TYPE (default=DEFERRED):

- DEFERRED → permette la cancellazione soltanto nei cancellation point.
- ASYNCHRONOUS → permette la cancellazione in qualsiasi parte

Aspettare un thread:

```
int main(){
void *ritorno;
int ritorno;
    pthread_join(id_thread, (**void)&ritorno)
    pthread_join(id_thread, &ritorno)
}
```

Un thread può essere aspettato da un solo thread.

Attributi di un thread:

```
pthread_attr_t attributi;
int detachstate;
sigset_t mask;

pthread_attr_setdetachstate(&attributi,PTHREAD_CREATE_DETACHED);
pthread_attr_getdetachstate(&attributi ,&detachstate);

pthread_attr_setsigmask_np(&attributi, &mask);


pthread_create(&id_thread; &attributi, funzione, (void *)&arg);
```

setdetachstate (default=JOINABLE):

- DETACH = non permette join
- JOINABLE = permette join
- In esecuzione:
    - DETACH to JOINABLE → non si può fare

- JOINABLE to DETACH → si può fare → pthread_detach(id_thread)

pthread_attr_setsigmask_np: associa ad un thread una maschera di segnali

---

> Mutex

Da utilizzare quando vogliamo proteggere una variabile o porzione di codice condiviso da più thread.

Dichiarazione mutex (variabile globale):

```
pthread_mutex_t lock;
```

Inizializzazione:

```
pthread_mutex_init(&lock, NULL); //senza attributi
```

Set attributi mutex:

```
pthread_mutexattr_t attr;
pthread_mutex_init(&lock, &attr);
pthread_mutexattr_settype(&attr, TYPE);
```

TYPE:
- PTHREAD_MUTEX_DEFAULT
- PTHREAD_MUTEX_NORMAL
- PTHREAD_MUTEX_ERRORCHECK
- PTHREAD_MUTEX_RECURSIVE

Distruggere:

```
pthread_mutex_destroy(&lock);
```

Utilizzo in funzione thread:

```
pthread_mutex_lock(&lock);
 // —--- codice protetto —---
pthread_mutex_unlock(&lock);
```

Cheat Sheet

Alias bash:

```
alias somma='somma="0";valore="0";read valore; while [[ -n $valore ]];
do somma=$(( $somma + $valore )); read valore; done; echo "somma is
$somma"'
```

Importante: apici e spazi.
-n $valore controlla che non sia vuoto.
-z $valore controlla che sia vuoto.

Esecuzione Docker:

```
docker run -ti --rm --name="labOS" --privileged  -v /:/host -v
"$(pwd):/home/labOS" --hostname "labOS" --workdir /home/labOS
ubuntu:20.04 /bin/bash
```

```
apt-get update && apt-get install -y nano build-essential
```

Eseguire un altro terminale docker

```
docker exec -ti <nome_container> bash
```

GCC

```
gcc -E <sorgente.c> -o <preProcessed.ii|.i>
gcc -S <preProcessed.i|.ii> -o <assembly.asm|.s>
gcc -c <assembly.asm|.s> -o <objectFile.obj|.o>
gcc <objectFile.obj|.o> -o <executable.out>
```

Makefile

```
> make → ./program
> make NAME=ciao → ./ciao
@ non scrive la regola

NAME = program
all:
	@echo "Compilo..."
	@gcc *.c -o $(NAME)
clean:
	rm *.out
.PHONY: clean
```

Mandare segnali da terminale | Lista segnali | Terminare tutti processi:

```
kill -SEGNALE PID
kill -l
pkill <program>
```

Codici Permessi:
chmod [UGO] file → UGO = user | group | other
In C aggiungere uno 0 davanti a [UGO] → “0777”

Decimale	**Permessi**
7	lettura, scrittura ed esecuzione
6	lettura e scrittura
5	lettura ed esecuzione
4	solo lettura
3	scrittura ed esecuzione
2	solo scrittura
1	solo esecuzione
0	nessuno

Include:

```
#include <string.h>
#include <stdio.h>
#include <stdlib.h>
#include <fcntl.h>
#include <sys/stat.h>
#include <sys/types.h>
#include <unistd.h>
#include <signal.h>
#include <pthread.h>
#include <errno.h>
#include <sys/wait.h>
#include <sys/ipc.h>
#include <sys/msg.h>
#include <stdbool.h>
#include <ctype.h>
```

Check numero argomenti:

```
narg= argc-1;
if(narg>NUMEROMAX) quit(11);
```

Parsing argomenti (consigliato quando non importa ordine opzioni):
in getopt() l’ultimo parametro rappresenta la stringa degli argomenti accettati. Se dopo un argomento poniamo ‘:’ l’opzione richiede un argomento.

```
bool primo false; bool secondo = false;
while((opt = getopt(argc, argv, ":cl")) != -1 && i < MAX_ARG_OPT ){
             switch(opt)
             {
                    case 'c':
                if(!primo){ options[0]='c'; primo=true; break; }
                if(primo && !secondo){ options[1]='c'; secondo=true; break; }
                    case 'l':
                if(!primo) { options[0]='l'; primo=true; break; }
                if(primo && !secondo){ options[1]='l'; secondo=true; break; }
             }
        i++;
       }
    //filepath
    for(; optind < argc; optind++){
        strcpy(filename, argv[optind]);
        break;
}
 for (index = optind; index < argc; index++)
    printf ("Non-option argument %s\n", argv[index]);
  return 0;
}
```

Parsing senza getopt():

```
char opt[2];
char filename[50];

int parseargs(int nargs, char **cargs)
{
    int curarg = 1; //argv[0] == ./program !!
    char arg[30];
    int oi = 0;
    bool optc = false;
    bool optl = false;
    bool optfn = false;
    while (curarg < nargs){
        strcpy(arg, cargs[curarg]);
        if (strcmp(arg, "-c") == 0 && !optc){
            opt[oi++] = 'c';
            optc = false;
        } //setto se c
        if (strcmp(arg, "-l") == 0 && !optc){
            opt[oi++] = 'l';
            optc = false;
        } //setto se l
        if (strcmp(arg, "-c") != 0 && strcmp(arg, "-l") != 0 && arg[0] == '-')
            exit(1);
        if (strcmp(arg, "-c") != 0 && strcmp(arg, "-l") != 0 && arg[0] != '-'){
            if (!optfn){
                strcpy(filename, arg);
```

```
                optfn = true;
            }else exit(2);
        };
        if (oi > 2) exit(3);
        curarg++;
    };
printf("opt0 --> %c\n", opt[0]); //stampa di controllo
printf("opt1 --> %c\n", opt[1]); //stampa di controllo
printf("filename --> %s\n", filename); //stampa di controllo
    return 0;
}
```

Check piping o direct

```
if(isatty(fileno(stdin))){
   direct();
}else{
   piping();
}

int direct(){
    char c;
    if(strcmp(filename, "")==0) { if(IGNORE_BLANK_INPUT) return 0; else quit
(3); }
    content = fopen(filename, "r");
    if(content==NULL) quit(8);
    while(content != NULL){
        c= fgetc(content);
        if(feof(content)) break;
        nc++;
        if(c=='\n') nl++;
    }
    if(content!=NULL) fclose(content);
    return 0;
}

int piping(){
    char c;
    while(read(STDIN_FILENO, &c, 1)){
        nc++;
        if(c=='\n') nl++;
    }
    return 0;
}
```

Errori:

```
extern int errno;
```

```
void main() {
    FILE * pf = fopen ("nonesiste.boh", "r");
    if (pf == NULL) {
        fprintf(stderr, "errno: %d\n", errno);
        fprintf(stderr, "strerror: %s\n", strerror(errno));
        perror("Error printed by perror");
    }else {
        fclose (pf);
    }
}
```

Exit Code (con messaggi personalizzati):

```
void quit(int err) {
    char msg[100];
      switch (err) {
            case  0 : strcpy(msg, "");                            break;
            case  2 : strcpy(msg, "?Wrong number of arguments"); break;
            case  3 : strcpy(msg, "?Wrong argument 'n'");        break;
            case  4 : strcpy(msg, "?'path' too long");           break;
            case  5 : strcpy(msg, "?Error in path");             break;
            case  6 : strcpy(msg, "?Fork error");                break;
            case  7 : strcpy(msg, "?Error creating txt");        break;
            case  8 : strcpy(msg, "?Error writing log");         break;
            case  9 : strcpy(msg, "?Error in queue");            break;
            case 10 : strcpy(msg, "?Global quit");               break;
            default: strcpy(msg, "?Generic error");              break;
      };
      if (err>0) { //errori su stderr
        fprintf(stderr, "%s", msg);
    } else { //successo su stdout
        printf("%s\n", msg);
    };
      fflush(stderr); fflush(stdout);
      exit(err);
};
```

Exit Code (con messaggi standard):

```
int main(int argc, char **argv){
        FILE * pf = fopen ("nonesiste.boh", "r");
        if(pf == -1) quit(3);
        if (pf == NULL) {
            //intero
            fprintf(stderr, "errno: %d\n", errno);
            //stringa relativa all'intero
            fprintf(stderr, "strerror: %s\n", strerror(errno));
            //stampa direttamente la stringa
            perror("Ciao");
        }else {
            fclose (pf);
```

```
        }
}
```

Stream:

```
int main(int argc, char **argv)
{
    FILE *ptr;
    ptr = fopen("test.txt", "w+");
    char stringa[DIM];
    sprintf(stringa, "contenuto da scrivere");
    fprintf(ptr, "%s\n", stringa);
    rewind(ptr); //ptr to the begin of the file
    char c;
    c = fgetc(ptr);
    while (c != EOF)
        {
            printf("%c", c);
            c = fgetc(ptr);
        }

    fclose(ptr);
    return 0;
}
```

File Descriptors:

```
int main(int argc, char **argv)
{
    int fd = open("test.txt", O_CREAT | O_RDWR, S_IRUSR | S_IWUSR);
    char stringa[DIM];
    sprintf(stringa, "contenuto da scrivere");
    int scritto = write(fd, stringa, strlen(stringa));
    if (scritto > 0)
    {
        lseek(fd, 0, SEEK_SET);
        int canread;
        char c;
        canread = read(fd, &c, 1);
        while (canread > 0)
        {
            printf("%c", c);
            canread = read(fd, &c, 1);
        }
    }

    return 0;
}
```

System (SHELL su C):

```
int main(int argc, char **argv)
{
    int rimuoviCartella = system("rm -f -r casa");
    if (rimuoviCartella == 0)
    {
        int creaCartella = system("mkdir casa");
        if (creaCartella == 0)
        {
            printf("Cartella creata\n");
        }
        return 0;
    }
}
```

Sigaction

```
//parametri info e empty solo se uso informazioni
static void handler(int signo, siginfo_t* info, void* empty){
    switch (signo) {
            case SIGUSR1:
                printf("SIGUSR1 da sigaction da [PID]: %i\n",info->si_pid );
            break;
            case SIGUSR2:
                printf("SIGUSR2 da sigaction da [PID]: %i\n",info->si_pid );
            break;
    }
}

int main(void)
{
    printf("[PID] %i\n", getpid()); fflush(stdout);
    struct sigaction sa; //struct sigaction
    sa.sa_sigaction = handler; //utilizzo informazioni
    //sa.sa_handler = handler; //non utilizzo informazioni
    sigemptyset(&sa.sa_mask);   //svuotare maschera
    sa.sa_flags |= SA_SIGINFO;  //flag per informazioni
    sigaction(SIGUSR1, &sa, NULL); //segnale gestito in handler
    sigaction(SIGUSR2, &sa, NULL);
    sigaddset(&sa.sa_mask,SIGTERM); //segnale aggiunto a maschera
    sigdelset(&sa.sa_mask,SIGTERM); //segnale rimosso a maschera

    sigprocmask(SIG_BLOCK, &sa.sa_mask, NULL); //bloccare segnali della maschera
    //sigprocmask(SIG_IGN, &sa.sa_mask, NULL); //ignorare segnali della maschera

    kill(getpid(),SIGTERM);
    sigpending(&sa.sa_mask); //inserisce nella maschera se un segnale è pending
```

```
o no
    if(sigismember(&sa.sa_mask, SIGTERM)){ //contollo segnale è pending
(ricevuto e in attesa)
        printf("SIGTERM pending\n");
        sigprocmask(SIG_UNBLOCK, &sa.sa_mask, NULL); //sblocca segnali della
maschera ed esegue quelli bloccati che ha ricevuto prima
    }



    while(1);
    return 0;
}
```

Concatenare un intero ad una stringa:

```
int intero = 50;
char stringa_dest[MAX_LENG]= “il numero concatenato è = ” ;
snprintf(stringa_dest, MAX_LENG, "%d\n", intero);

sprintf(stringa, “%d”, intero);
```

Creare N figli

```
for(int i=0; i<N_FIGLI; i++){
    f=fork();
    if(f==0){
       child(); //esecuzione figlio int child();
    }else{
       array[i]=f; //salvare pid figli
    }
}
```

Controllare se carattere <u>non</u> è un carattere speciale

```
if(isalnum(stringa[0])){};   //libreria <ctype.h>
if(isdigit(comando[i]) != 0){}; //controlla se è una cifra

o

if(ch >= 'A' && ch <= 'Z'){
  printf("'%c' is uppercase alphabet.", ch);
}
else if(ch >= 'a' && ch <= 'z')
{
  printf("'%c' is lowercase alphabet.", ch);
}else{
}
```

Redirezionamento stdout e reset

```
int main(){
    int file=open("txt.txt", O_CREAT | O_RDWR, S_IRUSR | S_IWUSR);
    int a = dup(1);
    int b = dup2(file ,1);
    system("ls");
    close(file);
    int c = dup2(a, 1);
    system("ls");
}
```

Verificare se processo è stato ucciso da segnale non gestito

```
int status;
wait(&status) //qualsiasi altro tipo di wait con parametro status
switch(WTERMSIG(status)){
        case SIGUSR1: //segnale gestito
            printf("Kill da SIGUSR1\n");
            break;
        default: //segnale non gestito
            printf("Ucciso da segnale non gestito %d\n", WTERMSIG(status));
            break;
}
```

Creare o eliminare file e cartelle

```
creat(“path”, PERMESSI); //file
remove(“path”);
mkdir(“path” , PERMESSI); //cartella
```

path → posizione corrente
/path → percorso assoluto

Queues (code):

```
struct messaggio{
    long mtype; //tipo messaggio, identifica la classe
    char mtext[1]; //campo testuale
} messaggio_padre;

struct msqid_ds mod; //struct dettagli

int main(int argc, char **argv){
    key_t coda_key = ftok("nome.file", 2);
    int coda_id = msgget(coda_key, 0777 | IPC_CREAT);
    msgctl(coda_id, IPC_RMID, NULL); //cancello e svuoto
    coda_id = msgget(coda_key, 0777 | IPC_CREAT);

    int processo=fork();
    if(processo==0){
        printf("[FIGLIO] = %d\n", getpid());
        struct messaggio{
```

```
            long mtype; //tipo messaggio, identifica la classe
            char mtext[10]; //campo testuale
        } messaggio;
        strcpy(messaggio.mtext,"Ciao"); //set testo
        messaggio.mtype=10; //set argomento
        msgctl(coda_id, IPC_STAT, &mod); //recupero valori coda
        mod.msg_qbytes=19; //cambio max bytes coda
        msgctl(coda_id, IPC_SET, &mod); //attuo modifiche coda
        int esito = msgsnd(coda_id, &messaggio, sizeof(messaggio.mtext),
IPC_NOWAIT); //invio
        printf("Esito1: %d\n\n", esito);

    }else{
        sleep(1);
        //ricezione
        int esito = msgrcv(coda_id, &messaggio_padre,
sizeof(messaggio_padre.mtext), 0, MSG_NOERROR);
        printf("(%d) %s\n", esito, messaggio_padre.mtext);
    }
}
```

Thread

```
#define READ 0
#define WRITE 1

int fd[2];

void *funzioneThread(void *param)
{
    char buffer[100];
    read(fd[READ], buffer, sizeof(buffer));
    printf("Letto da thread --> %s\n", buffer);
    return (void *)3; //valore arbitrario
}

int main(int argc, char **argv)
{

    pthread_t id_thread;
    pipe(fd);
    int numero = 0;
    pthread_create(&id_thread, NULL, funzioneThread, (void *)&numero);
    char buffer[100];
    strcpy(buffer, "ciao mamma");
    rite(fd[WRITE], buffer, sizeof(buffer));
    pthread_join(id_thread, NULL);
    /*
    Se mettessi qui la write resterebbe bloccato
    Il thread aspetta qualcosa nella pipe
```

```
    Il main aspetta il thread
    */

    return 0;
}
```

Colorare testo terminale

```
#define RED "\033[0;31m"
#define GREEN "\033[0;32m"
#define YELLOW "\033[0;33m"
#define BLUE "\033[0;34m"
#define MAGENTA "\033[0;35m"
#define CYANO "\033[0;36m"
#define WHITE "\033[0;37m"
#define DF "\033[0m"

printf(RED "testo da scrivere\n" DF);
```

Controllare se processo è eseguito in foreground

```
> ./main.o & → foreground
int foreground(){
    int fg=0;
    if (getpgrp() == tcgetpgrp(STDOUT_FILENO)) fg=1;
    return fg;
}
```