

# Librerie

---

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <fcntl.h>
#include <sys/wait.h>
#include <sys/ipc.h>
#include <sys/msg.h>
```

# Bash

---

```
[[ ! -e $PATH ]] #check if PATH exists

for ((i = 0; i < 10; i++)); do
    echo ${i}
done
```

# MAKEFILE

---

```
SHELL := /bin/bash
.DEFAULT_GOAL := build
FILE="NONE"

build:
    @if [[ -f $(FILE) ]]; then\
        echo "file esiste";\
    else\
        echo "File non esiste";\
    fi\
```

# FILE

---

## file stream

```
FILE* file = fopen(path, mode);
fclose(file);
feof(file);
```

```
/**lettura**  
fgetc(file);  
fgets((char*) salvaqui, dim, file);  
  
/**scrittura**  
fputc(singleChar, file);  
fprintf(file, str);
```

## file descriptors

```
//flags = O_RDONLY | O_WRONLY | O_RDWR;  
int fd = open(path, flags, S_IRUSR);  
close(fd);  
feof(fd);  
  
/**lettura**  
read(fd, (char*) salvaqui, dim);  
  
/**scrittura**  
write(fd, str, dim);  
  
/** duplicazione **  
dup2(oldFd, newFD);  
//usato per reindirizzare stdout, ...  
dup2(fd, stdout);
```

## Fork

---

0 -> child 1 -> parent

```
getppid(); //parent pid  
  
//aspetta tutti i figli  
while(wait(NULL)>0);  
  
//aspettare pid  
waitpid(pid, NULL, 0); //pid == -1 -> un figlio qualsiasi
```

## Segnali

---

handler

```

struct sigaction sa;
sa.sa_handler = HANDLER_FUNZ;
sa.sa_flags = SA_SIGINFO;
//se si resetta dopo il primo segnale ->
//    sa.sa_flags = SA_RESETHAND | SA_SIGINFO;
sigemptyset(&sa.sa_mask);
//se si vuole bloccare un signal ->
//    sigaddset(&sa.sa_mask, SIGNAL_TO_MASK);
sigaction(SIGNAL_TO_HANDLE,&sa,NULL);

void HANDLER_FUNZ(int signo, siginfo_t * info, void * empty){
    //mittente = info->si_pid;
    //singal recieved = signo
}

```

## invio segnali

```

int kill(pid, signal);

// 0 = inviati
// -1 = errore
//^^ si usa per verificare se PID esiste ^^

```

## Pipes

---

```

//creazione anonima
int pipefd[2] = {fdLettura, fdScrittura};
pipe(pipefd);

//creazione con nome/FIFO
mkfifo(filePath, S_IRUSR|S_IWUSR);
//poi si usa come file normale
int fd = open(filePath, O_RDONLY);

//lettura
read(fd[0], salvaQui, dim);

//scrittura
write(fd[1], str, dim);

```

## unidirezionale

- P1 crea una pipe()
- P1 esegue un fork() e crea P2

- P1 chiude il lato lettura: `close(fd[0])`
- P2 chiude il lato scrittura: `close(fd[1])`
- P1 e P2 chiudono l'altro fd appena finiscono di comunicare.

bidirezionale

2 unidirezionali

## Queues

---

due identificativi, **key** e **queue identifier**.

da **key**, ottengo **queue identifier**.

**key** generata da utente

```
/** creazione **
flags = 0777 | IPC_CREAT | IPC_EXCL
int queueuID = msgget(key, flags);
//key univoca, basata su PATH.
//coppia <path, id> dovrebbe dare stessa key.
int key = ftok(path, id);

//messaggio della coda ha forma:
struct msg_buffer{
    long mtype;
    // customizzabile
    char mtext[100];
    //può essere anche struttura dati:
    Book mtext;
} message;

/** lettura **
msgrcv(queueuID, salvaQui, size, msgtyp, flags);
//msgtyp = 0 -> primo msg (FIFO)
//      > 0 -> primo msg di tipo msgtyp
//      < 0 -> primo msg il cui tipo T é min(T <= |msgtyp|)

/** scrittura **
msgsnd(queueuID, str, dim, flags)
```