Doc

man7 documentazione

Librerie

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

#include <unistd.h>
#include <fcntl.h>
#include <errno.h>
#include <signal.h>
#include <pthread.h>

#include <sys/wait.h>
#include <sys/ipc.h>
#include <sys/msg.h>
#include <sys/stat.h>
#include <sys/types.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

```
remove("file_path"); //cancella 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_RDWD;
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
other -> parent

```
getpid(); // own pid
getppid(); // parent pid

//aspetta tutti i figli
while(wait(NULL)>0);

//aspettare pid
waitpid(pid, NULL, 0); //pid == -1 -> un figlio qualisisi
```

Segnali

Handler

```
struct sigaction sa;
sa.sa_sigaction = HANDLER_FUNZ;
sa.sa_flags = SA_SIGINFO;
//se si resetta dopo il primo signale ->
//    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(queueueID, 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)
```

Threads

per compilare bisogna aggiungere flag `-pthread`

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

Creazione

```
pthread_t threadId;
pthread_create(&threadId, attr, funz_da_eseguire, (void*) &args)
```

Terminazione

5 possibilità

```
pthread_exit(retVal);

return retVal;

pthread_cancel(threadId);

exit()

*main_thread_finishes*
```

Cancellazione di un thread

un thread puà inviare una richiesta per cancellare un altro thread.
altro thread risponderà in base a **state** e **type**

- state = se thread deve terminare quando riceve richiesta
- type = come deve termina

```
pthread_setcancelstate(int state, int* oldstate);
//state = PTHREAD_CANCEL_ENABLE (default)
//state = PTHREAD_CANCEL_DISABLE

pthread_setcanceltype(int type, int* oldtype);
//type = PTHREAD_CANCEL_DEFERRED (default)
    // terminazione aspetta cancellation point
    // https://man7.org/linux/man-pages/man7/pthreads.7.html

//type = PTHREAD_CANCEL_ASYNCHRONOUS
    // terminazione appena riceve richiesta
```

Aspettare thread | Join

thread può essere aspettato da massimo un altro thread

```
void* retVal; //deve essere void*
pthread_join(threadId, &retVal);
```

Non tutti thread possono essere aspettati

```
pthread_detach(threadId); //makes thread unjoinable
```

Attributi

```
pthread_attr_t attr;
pthread_attr_init(&attr); //default init
pthread_attr_setdetachstate(&attr, PTHREAD_CREATE_DETACHED);
pthread_attr_getdetachstate(&attr, &detachSTate);
// ...
pthread_create(&t_id, &attr, my_fun, NULL); //attach attr to thread
pthread_attr_destroy(&attr); //Destructor
```

Testing from bash

- trap signals `trap "<comando_bash>" <segnale>` + `Ctrl+C`
- vedere PID terminale `echo $$`
- mandare messaggio su fifo/pipe `echo -n "msg" > file_fifo`
- controllare se file è una fifo `ls -lp` e guardare per `prw--------`
- lista code `ipcs -q`