LabSO 2022

Laboratorio Sistemi Operativi - A.A. 2021-2022

dr. Andrea Naimoli	Informatica LT andrea.naimoli@unitn.it
dr. Michele Grisafi	Ingegneria informatica, delle comunicazioni ed elettronica (LT) michele.grisafi@unitn.it

Nota sugli “snippet” di codice

Alcuni esempi di codice possono essere semplificati, ad esempio omettendo il blocco principale con la funzione `main` *(che andrebbe aggiunto) oppure elencando alcune o tutte le librerie da includere tutte su una riga o insieme (per cui invece occorre trascrivere correttamente le direttive* `#include` *secondo la sintassi corretta) o altre semplificazioni analoghe. In questi casi occorre sistemare il codice perché possa essere correttamente compilato e poi eseguito.*

Queues

Message queues

Una coda di messaggi, message queue, è una lista concatenata memorizzata all'interno del kernel ed identificata con un ID (un intero positivo univoco), chiamato **queue identifier**.

Questo ID viene condiviso tra i processi interessati, e viene generato attraverso una chiave univoca.

Una coda deve essere innanzitutto generata in maniera analoga ad una FIFO, impostando dei permessi. Ad una coda esistente si possono aggiungere o recuperare messaggi tipicamente in modalità "autosincrona": la lettura attende la presenza di un messaggio, la scrittura attende che via sia spazio disponibile. Questi comportamenti possono però essere configurati.

Queue identifier e queue key

Quanto trattiamo di message queue, abbiamo due identificativi:

- **Key**
- **Queue identifier**

Key: intero che identifica un insieme di risorse condivisibili nel kernel, come semafori, memoria condivisa e **code**. Questa chiave univoca deve essere nota a più processi, e viene usata per ottenere il queue identifier.

Queue identifier: id univoco della coda, generato dal kernel ed associato ad una specifica *key*. Questo ID viene usato per interagire con la coda.

Creazione coda

`int msgget(key_t key, int msgflg)`

Restituisce l'identificativo di una coda basandosi sulla chiave "key" e sui flags:

- `IPC_CREAT`: crea una coda se non esiste già, altrimenti restituisce l'identificativo di quella già esistente;
- `IPC_EXCL`: (da usare con il precedente) fallisce se coda già esistente;
- `0xxx`: permessi per accedere alla coda, analogo a quello che si può usare nel file system. In alternativa si possono usare S_IRUSR etc.

```
#include <sys/types.h> <sys/ipc.h> <sys/msg.h>                    //msgget.c
key_t queueKey = 56; //Unique key
int queueId = msgget(queueKey, 0777 | IPC_CREAT | IPC_EXCL);
```

Ottenere chiave univoca

`key_t ftok(const char *path, int id)`

Restituisce una chiave basandosi sul *path* (una cartella o un file), esistente ed accessibile nel file-system, e sull'id numerico. La chiave dovrebbe essere univoca e sempre la stessa per ogni coppia <path,id> in ogni istante sullo stesso sistema.

Un metodo d'uso, per evitare possibili conflitti, potrebbe essere generare un path (es. un file) temporaneo univoco, usarlo, eventualmente rimuoverlo, ed usare l'id per rappresentare diverse "categorie" di code, a mo' di indice.

```
#include <sys/ipc.h>                                    //ftok.c
key_t queue1Key = ftok("/tmp/unique", 1);
key_t queue2Key = ftok("/tmp/unique", 2); ...
```

Esempio creazione

```
<sys/types.h><sys/ipc.h> <sys/msg.h><stdio.h><fcntl.h>  //ipcCreation.c
void main(){
    remove("/tmp/unique"); //Remove file
    key_t queue1Key = ftok("/tmp/unique", 1); //Get unique key → fail
    creat("/tmp/unique", 0777); //Create file
    queue1Key = ftok("/tmp/unique", 1); //Get unique key → ok
    int queueId = msgget(queue1Key ,0777 | IPC_CREAT); //Create queue → ok
    queueId = msgget(queue1Key , 0777); //Get queue → ok
    msgctl(queue1Key,IPC_RMID,NULL);  //Remove non existing queue → fail
    msgctl(queueId,IPC_RMID,NULL); //Remove queue → ok
    queueId = msgget(queue1Key , 0777); //Get non existing queue → fail
    queueId = msgget(queue1Key , 0777 | IPC_CREAT); //Create queue → ok
    queueId = msgget(queue1Key , 0777 | IPC_CREAT); //Get queue → ok
    queueId = msgget(queue1Key , 0777 | IPC_CREAT | IPC_EXCL); /* Create
                                    already existing queue -> fail */
}
```

Le queue sono persistenti

```c
#include <sys/ipc.h> <stdio.h> <sys/msg.h>          //persistent.c

void main(){
    key_t queue1Key = ftok("/tmp/unique", 1);
    int queueId = msgget(queue1Key , 0777 | IPC_CREAT | IPC_EXCL);
    perror("Error:");
}
```

Se eseguiamo questo programma dopo aver eseguito il precedente "*ipcCreation.c*" verrà generato un errore dato che la coda esiste già ed abbiamo usato il flag IPC_EXCL!

Da bash possiamo usare il comando `ipcs` per avere informazioni sulle queues.

Comunicazione - 1

```
struct msg_buffer{
    long mtype;
    char mtext[100];
} message;
```

Ogni messaggio inserito nella coda ha:

- Un tipo, categoria, etc... (intero > 0)
- Una grandezza non negativa
- Un payload, un insieme di dati (bytes) di lunghezza corretta

Al contrario delle FIFO, i messaggi in una coda possono essere recuperati anche sulla base del tipo e non solo del loro ordine “assoluto” di arrivo. Così come i files, le code sono delle strutture persistenti che continuano ad esistere, assieme ai messaggi in esse salvati, anche alla terminazione del processo che le ha create. L’eliminazione deve essere esplicita.

Comunicazione - 2

Il payload del messaggio non deve essere necessariamente un campo testuale: può essere una qualsiasi struttura dati. Un messaggio può anche esistere senza payload.

```
typedef struct book{
    char title[10];
    char description[100];
    unsigned short chapters;
} Book;

struct msg_buffer{
    long mtype;
    Book mtext;
} message_rcv;

struct msg_empty{
    long mtype;
} message_empty;
```

Inviare messaggi

```
int msgsnd(int msqid, const void *msgp, size_t msgsz, int msgflg);
```

Aggiunge una copia del messaggio puntato da `msgp`, con un payload di dimensione `msgsz`, alla coda identificata da `msqid`. Il messaggio viene inserito immediatamente se c'è abbastanza spazio disponibile, altrimenti la chiamata si blocca fino a che abbastanza spazio diventa disponibile. Se *`msgflg`* è IPC_NOWAIT allora la chiamata fallisce in assenza di spazio.

NB: `msgsz` è la grandezza del payload del messaggio, non del messaggio intero (che contiene anche il tipo)! Per esempio, `sizeof(msgp.mtext)`

Ricevere messaggi - 1

```
ssize_t msgrcv(int msqid,void *msgp,size_t msgsz,long msgtyp,int msgflg)
```

Rimuove un messaggio dalla coda `msqid` e lo salva nel buffer `msgp`. `msgsz` specifica la lunghezza massima del payload del messaggio (per esempio *`mtext`* della struttura `msgp`). Se il payload ha una lunghezza maggiore e `msgflg` è `MSG_NOERROR` allora il payload viene troncato (viene persa la parte in eccesso), se `MSG_NOERROR` non è specificato allora il payload non viene eliminato e la chiamata fallisce.

Se non sono presenti messaggi, la chiamata si blocca in loro attesa. Il flag IPC_NOWAIT fa fallire la syscall se non sono presenti messaggi.

Ricevere messaggi - 2

```
ssize_t msgrcv(int msqid,void *msgp,size_t msgsz,long msgtyp,int msgflg)
```

A seconda di `msgtyp` viene recuperato il messaggio:

- `msgtyp = 0`: primo messaggio della coda (FIFO)
- `msgtyp > 0`: primo messaggio di tipo `msgtyp`, o primo messaggio di tipo diverso da `msgtyp` se `MSG_EXCEPT` è impostato come flag
- `msgtyp < 0`: primo messaggio il cui tipo T è `min(T ≤ |msgtyp|)`

Esempio comunicazione - 1

```c
#include <sys/types.h><sys/ipc.h><sys/msg.h><string.h><stdio.h>  //ipc.c
struct msg_buffer{
    long mtype;
    char mtext[100];
} msgp, msgp2; //Two different message buffers
int main(void){
    msgp.mtype = 20;
    strcpy(msgp.mtext,"This is a message");
    key_t queue1Key = ftok("/tmp/unique", 1);
    int queueId = msgget(queue1Key , 0777 | IPC_CREAT | IPC_EXCL);
    int esito = msgsnd(queueId , &msgp, sizeof(msgp.mtext),0);
    esito = msgrcv(queueId , &msgp2, sizeof(msgp2.mtext),20,0);
    printf("Received %s\n",msgp2.mtext);
}
```

Esempio comunicazione - 2

```
//ipcBook.c
#include <stdio.h> <sys/types.h> <sys/ipc.h> <sys/msg.h> <string.h>

typedef struct book{
    char title[10];
    char description[200];
    short chapters;
} Book;

struct msg_buffer{
    long mtype;
    Book mtext;
} msgp_snd, msgp_rcv; //Two different message buffers
```

Esempio comunicazione - 2

```
    …
int main(void){
    msgp_snd.mtype = 20;
    strcpy(msgp_snd.mtext.title,"Title");
    strcpy(msgp_snd.mtext.description,"This is a description");
    msgp_snd.mtext.chapters = 17;
    key_t queue1Key = ftok("/tmp/unique", 1);
    int queueId = msgget(queue1Key , 0777 | IPC_CREAT);
    int esito = msgsnd(queueId , &msgp_snd, sizeof(msgp_snd.mtext),0);
    esito = msgrcv(queueId , &msgp_rcv, sizeof(msgp_rcv.mtext),20,0);
    printf("Received: %s %s %d\n",msgp_rcv.mtext.title,
        msgp_rcv.mtext.description, msgp_rcv.mtext.chapters);
}
```

Esempio comunicazione - 3

```
#include <sys/types.h>
#include <stdio.h>
#include <stdlib.h>
#include <sys/ipc.h>
#include <sys/msg.h>
#include <string.h> //ipcType.c

struct msg_buffer{
     long mtype;
     char mtext[100];
} msgp_snd,msgp_rcv; //Two different message buffers
…
```

Esempio comunicazione - 3

```
int main(int argc, char ** argv){
     int to_fetch = atoi(argv[1]); //Input to decide which msg to get
     key_t queue1Key = ftok("/tmp/unique", 1);
     int queueId = msgget(queue1Key , 0777 | IPC_CREAT);
     msgp_snd.mtype = 20;
     strcpy(msgp_snd.mtext,"A message of type 20");
     int esito = msgsnd(queueId , &msgp_snd, sizeof(msgp_snd.mtext),0);
     msgp_snd.mtype = 10; //Re-use the same message
     strcpy(msgp_snd.mtext,"Another message of type 10");
     esito = msgsnd(queueId , &msgp_snd, sizeof(msgp_snd.mtext),0);
     esito = msgrcv(queueId , &msgp_rcv, sizeof(msgp_rcv.mtext),to_fetch,0);
     printf("Received: %s\n",msgp_rcv.mtext);
}
```

Modificare la coda

```
int msgctl(int msqid, int cmd, struct msqid_ds *buf);
```

Modifica la coda identificata da `msqid` secondo i comandi `cmd`, riempiendo buf con informazioni sulla coda (ad esempio tempo di ultima scrittura, di ultima lettura, numero messaggi nella coda, etc...). Valori di `cmd` possono essere:

- `IPC_STAT`: recupera informazioni da kernel
- `IPC_SET`: imposta alcuni parametri a seconda di `buf`
- `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

https://man7.org/linux/man-pages/man2/msgctl.2.html

msqid_ds structure

```
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; //Time of creation or last modification by msgctl
    unsigned long   msg_cbytes; /* # of bytes in queue */
    msgqnum_t       msg_qnum;   /* # of messages in queue */
    msglen_t        msg_qbytes; /* Maximum # of bytes in queue */
    pid_t           msg_lspid;  /* PID of last msgsnd(2) */
    pid_t           msg_lrpid;  /* PID of last msgrcv(2) */
};
```

ipc_perm structure

```
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 */
};
```

Esempio modifica coda

```
int main(void){
    struct msqid_ds mod;
    int esito = open("/tmp/unique",O_CREAT,0777);
    key_t queue1Key = ftok("/tmp/unique", 1);
    int queueId = msgget(queue1Key , IPC_CREAT | S_IRWXU  );
    msgctl(queueId,IPC_RMID,NULL);
    queueId = msgget(queue1Key , IPC_CREAT | S_IRWXU  );
    esito = msgctl(queueId,IPC_STAT,&mod); //Get info on queue
    printf("Current permission on queue: %d\n",mod.msg_perm.mode);
    mod.msg_perm.mode = 0000;
    esito = msgctl(queueId,IPC_SET,&mod);  //Modify queue
    printf("Current permission on queue: %d\n\n",mod.msg_perm.mode);
}
```

Esempio 4

```
#include <sys/types.h><sys/ipc.h><sys/msg.h><string.h><stdio.h><unistd.h><wait.h>
//ipc4.c
struct msg_buffer{
    long mtype;
    char mtext[100];
} msgpSND,msgpRCV;
void main(){
    struct msqid_ds mod;
    msgpSND.mtype = 1;
    strcpy(msgpSND.mtext,"This is a message from sender");
    key_t queue1Key = ftok("/tmp/unique", 1);
    int queueId = msgget(queue1Key , 0777 | IPC_CREAT);
    msgctl(queueId,IPC_RMID,NULL); //Remove queue if exists
    queueId = msgget(queue1Key , 0777 | IPC_CREAT); //Create queue
```

```
msgsnd(queueId, &msgpSND, sizeof(msgpSND.mtext),0); //Send msg
msgsnd(queueId, &msgpSND, sizeof(msgpSND.mtext),0); //Send msg
msgsnd(queueId, &msgpSND, sizeof(msgpSND.mtext),0); //Send msg

msgctl(queueId,IPC_STAT,&mod); //Modify queue
printf("Msg in queue: %ld\nCurrent max bytes in queue: %ld\n\n",
        mod.msg_qnum, mod.msg_qbytes);

mod.msg_qbytes = 200; //Change buf to modify queue bytes
msgctl(queueId,IPC_SET,&mod);  //Apply modification

printf("Msg in queue: %ld --> same number\nCurrent max bytes in
        queue: %ld\n\n",mod.msg_qnum, mod.msg_qbytes);

if( fork() != 0 ){ //Parent keep on writing on the queue
    printf("[SND] Sending 4th message with a full queue...\n");
    msgsnd(queueId, &msgpSND, sizeof(msgpSND.mtext),0); //Send msg
    printf("[SND] msg sent\n");
```

```
…
        printf("[SND] Sending 5th message with IPC_NOWAIT\n");
        if(msgsnd(queueId, &msgpSND, sizeof(msgpSND.mtext),IPC_NOWAIT )
            == -1){ //Send msg
            perror("Queue is full --> Error");
        }
    } else { // Child keeps reading the queue every 3 seconds
        sleep(3); msgrcv(queueId, &msgpRCV, sizeof(msgpRCV.mtext),1,0);
        printf("[Reader] Received msg 1 with msg '%s'\n",msgpRCV.mtext);
        sleep(3); msgrcv(queueId, &msgpRCV, sizeof(msgpRCV.mtext),1,0);
        printf("[Reader] Received msg 2 with msg '%s'\n",msgpRCV.mtext);
        sleep(3); msgrcv(queueId, &msgpRCV, sizeof(msgpRCV.mtext),1,0);
        printf("[Reader] Received msg 3 with msg '%s'\n",msgpRCV.mtext);
        sleep(3); msgrcv(queueId, &msgpRCV, sizeof(msgpRCV.mtext),1,0);
        printf("[Reader] Received msg 4 with msg '%s'\n",msgpRCV.mtext);
```

```
…


        sleep(3);
        if(msgrcv(queueId, &msgpRCV, sizeof(msgpRCV.mtext),1,IPC_NOWAIT)
            == -1){
            perror("Queue is empty --> Error");
        }else{
            printf("[Reader] Received msg 5 with msg '%s'\n",
                    msgpRCV.mtext);
        }
    }
    while(wait(NULL)>0);
}
```

Threads

Threads

I thread sono singole sequenze di esecuzione all'interno di un processo, aventi alcune delle proprietà dei processi. I threads non sono indipendenti tra loro e condividono il codice, i dati e le risorse del sistema assegnate al processo di appartenenza. Come ogni singolo processo, i threads hanno alcuni elementi indipendenti, come lo stack, il PC ed i registri del sistema.

La creazione di threads consente un parallelismo delle operazioni in maniera rapida e semplificata. Context switch tra threads è rapido, così come la loro creazione e terminazione. Inoltre, la comunicazione tra threads è molto veloce.

Per la compilazione è necessario aggiungere il flag `-pthread`, ad esempio:

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

Creazione

In C i thread corrispondono a delle funzioni eseguite in parallelo al codice principale. Ogni thread è identificato da un ID e può essere gestito come un processo figlio, con funzioni che attendono la sua terminazione.

```
int pthread_create(
    pthread_t *restrict thread, /* Thread ID */
    const pthread_attr_t *restrict attr, /* Attributes */
    void *(*start_routine)(void *), /* Function to be executed */
    void *restrict arg /* Parameters to above function */
);
```

Esempio creazione

```c
#include <stdio.h> <pthread.h> <unistd.h> //threadCreate.c

void *my_fun(void *param){
    printf("This is a thread that received %d\n", *(int *)param);
    return (void*)3;
}
void main(){
    pthread_t t_id;
    int arg=10;
    pthread_create(&t_id, NULL, my_fun, (void *)&arg);
    printf("Executed thread with id %ld\n",t_id);
    sleep(3);
}
```

Terminazione

Un nuovo thread termina in uno dei seguenti modi:

- Chiamando la funzione `void pthread_exit(void *retval);` dall'interno del thread con un valore di ritorno (per un eventuale uso nel "join")
- Ritorna (con un `return`) dalla funziona associata al thread specificando un valore di ritorno.
- Viene cancellato
- Qualche thread chiama *exit()*, o il thread che esegue *main()* ritorna dallo stesso, terminando così tutti i threads.

Cancellazione di un thread

```
int pthread_cancel(pthread_t thread);
```

Invia una **richiesta** di cancellazione al thread specificato, il quale reagirà (come e quando) a seconda di due suoi attributi: **state** e **type**. **State** può essere *enabled* (default) o *disabled*: se *disabled* la richiesta rimarrà in attesa fino a che diventa *enabled*, se *enabled* la cancellazione avverrà a seconda di **type**. **Type** può essere *deferred* (default) o *asynchronous*: attende la chiamata di un *cancellation point* o termina in qualsiasi momento, rispettivamente. "Cancellation points" sono funzioni definite nella libreria pthread.h (*). **State** e **type** possono essere modificati (dall'interno del thread):

```
int pthread_setcancelstate(int state, int *oldstate);
```

con state = `PTHREAD_CANCEL_DISABLE o PTHREAD_CANCEL_ENABLE`

```
int pthread_setcanceltype(int type, int *oldtype);
```

Con type = `PTHREAD_CANCEL_DEFERRED o PTHREAD_CANCEL_ASYNCHRONOUS`

(*) https://man7.org/linux/man-pages/man7/pthreads.7.html

Esempio cancellazione

```
#include <stdio.h> <pthread.h> <unistd.h>                    //thCancel.c
int i = 1;
void * my_fun(void * param){
    if(i--) pthread_setcancelstate(PTHREAD_CANCEL_DISABLE,NULL); //Change mode
    pthread_setcanceltype(PTHREAD_CANCEL_ASYNCHRONOUS,NULL); //Change type
    printf("Thread %ld started\n",*(pthread_t *)param); sleep(3);
    printf("Thread %ld finished\n",*(pthread_t *)param);
}
void main(){
    pthread_t t_id1, t_id2;
    pthread_create(&t_id1, NULL, my_fun, (void *)&t_id1); sleep(1); //Create
    pthread_cancel(t_id1); //Cancel
    printf("Sent cancellation request for thread %ld\n",t_id1);
    pthread_create(&t_id2, NULL, my_fun, (void *)&t_id2); sleep(1); //Create
    pthread_cancel(t_id2); //Cancel
    printf("Sent cancellation request for thread %ld\n",t_id2);
    sleep(5); printf("Terminating program\n");
}
```

Aspettare un thread

Un processo (thread) che avvia un nuovo thread può aspettare la sua terminazione mediante la funzione:

```
int pthread_join(pthread_t thread, void **retval);
```

Che ritorna quando il thread identificato da *thread* termina, o subito se il thread è già terminato. Se il valore di ritorno del thread non è nullo (parametro di *pthread_exit* o di *return*), esso viene salvato nella variabile puntata da *retval*. Se il thread era stato cancellato, retval è riempito con PTHREAD_CANCELED.

Solo se il thread è joinable può essere aspettato! Un thread può essere aspettato da al massimo un thread!

Esempio join I

```c
#include <stdio.h> <pthread.h> <unistd.h>                    //thJoin.c
void * my_fun(void * param){
    printf("Thread %ld started\n",*(pthread_t *)param); sleep(3);
    char * str = "Returned string";
    pthread_exit((void *)str); //or 'return (void *) str;'
}
void main(){
    pthread_t t_id;
    void * retFromThread; //This must be a pointer to void!
    pthread_create(&t_id, NULL, my_fun, (void *)&t_id); //Create
    pthread_join(t_id,&retFromThread); // wait thread
    // We must cast the returned value!
    printf("Thread %ld returned '%s'\n",t_id,(char *)retFromThread);
}
```

Esempio join II

```c
#include <stdio.h> <pthread.h> <unistd.h>
//threadJoin.c (v. esempio threadCreate.c)

void *my_fun(void *param){
    printf("This is a thread that received %d\n", *(int *)param);
    return (void*)3;
}
void main(){
    pthread_t t_id;
    int arg=10, retval;
    pthread_create(&t_id, NULL, my_fun, (void *)&arg);
    printf("Executed thread with id %ld\n",t_id);
    sleep(3);
    pthread_join(t_id, (void **)&retval);
    printf("retval=%d\n", retval);
}
```

Esempio join

```
#include <stdio.h> <pthread.h> <unistd.h>                    //thJoin.c
void * my_fun(void * param){ sleep(2);}
void * my_fun2(void * param){
    if(pthread_join( *(pthread_t *) param,NULL) != 0) printf("Error\n");
}
void main(){
    pthread_t t_id,t_id2;
    pthread_create(&t_id, NULL, my_fun, NULL); //Create
    pthread_create(&t_id2, NULL, my_fun, (void *)&t_id); //Create
    pthread_join(t_id,NULL); // wait thread
    sleep(1);
    perror();
}
```

Attributi di un thread

Ogni thread viene creato con degli attributi specificati nella struttura *pthread_attr_t*. Questa struttura, analogamente alla struttura usata per gestire le maschere dei segnali, è un oggetto usato solo alla creazione di un thread, ed è poi indipendente dallo stesso (se cambia, gli attributi del thread non cambiano). La struttura va inizializzata con `int pthread_attr_init(pthread_attr_t *attr);` che imposta tutti gli attributi al loro valore di default. Una volta usata e non più necessaria, la struttura va distrutta con `int pthread_attr_destroy(pthread_attr_t *attr);`

I vari attributi della struct possono, e devono, essere modificati singolarmente con le seguenti funzioni:

```
int pthread_attr_setxxxx(pthread_attr_t *attr, params);
int pthread_attr_getxxxx(const pthread_attr_t *attr, params);
```

Attributi di un thread

- `…detachstate(pthread_attr_t *attr, int detachstate)`
  - `PTHREAD_CREATE_DETACHED` → non può essere aspettato
  - `PTHREAD_CREATE_JOINABLE` → default, può essere aspettato
  - Può essere cambiato durante l'esecuzione con
    `int pthread_detach(pthread_t thread);`
- `...sigmask_np(pthread_attr_t *attr,const sigset_t *sigmask);`
- `...affinity_np(...)`
- `...setguardsize(...)`
- `...inheritsched(...)`
- `...schedparam(...)`
- `...schedpolicy(...)`
- altri

Detached e joinable threads

I threads vengono creati di default nello stato joinable, il che consente ad un altro thread di attendere la loro terminazione attraverso il comando *pthread_join*. I thread joinable rilasciano le proprie risorse non alla terminazione ma quando un thread fa il join con loro (salvando lo stato di uscita) (così come i sottoprocessi), oppure alla terminazione del processo. Contrariamente, i thread in stato detached liberano le loro risorse immediatamente una volta terminati, ma non consentono ad altri processi di fare il “join”.

NB: un thread detached non può diventare joinable durante la sua esecuzione, mentre il contrario è possibile.

Esempio attributi

```c
#include <stdio.h> <pthread.h> <unistd.h> //threadAttr.c
void *my_fun(void *param){
    printf("This is a thread that received %d\n", *(int *)param);
    return (void*)3;
}
void main(){
    pthread_t t_id; pthread_attr_t attr;
    int arg=10, detachSTate;
    pthread_attr_setdetachstate(&attr,PTHREAD_CREATE_DETACHED); //Set detached
    pthread_attr_getdetachstate(&attr,&detachSTate); //Get detach state
    if(detachSTate == PTHREAD_CREATE_DETACHED) printf("Detached\n");
    pthread_create(&t_id, &attr, my_fun, (void *)&arg);
    printf("Executed thread with id %ld\n",t_id);
    pthread_attr_setdetachstate(&attr,PTHREAD_CREATE_JOINABLE); //Inneffective
    sleep(3);
    int esito = pthread_join(t_id, (void **)&detachSTate);
    printf("Esito '%d' is different 0\n", esito);
}
```

CONCLUSIONI

QUEUES: le code sono un metodo di comunicazione comodo per inviare e ricevere informazioni anche “complesse” tra processi generici.

THREADS: i “thread” sono una sorta di “processi leggeri” che permettono di eseguire funzioni “in concorrenza” in modo più semplice rispetto alla generazioni di processi veri e propri (forking).