LabSO 2025

Laboratorio Sistemi Operativi - A.A. 2024-2025

Michele Grisafi - michele.grisafi@unitn.it

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
```

Threads in C

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.

Sebbene da un punto di vista l'esecuzione di diversi thread sia sempre parallela, a causa delle politiche di scheduling delle CPU l'esecuzione può essere parallela solo se la CPU la supporta, ossia se dispone di più core.

I threads sono molto utili per eseguire funzioni bloccanti in parallelo (es. interazione con pipes)

Creare un thread

```
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 */
);
```

Quando si crea un nuovo thread, è necessario fornire un puntatore `pthread_t`, che verrà riempito con il nuovo ID generato. `attr` consente di modificare il comportamento dei thread, mentre `start_routine` serve a definire quale funzione deve essere eseguita dal thread. `arg` è un puntatore void che può essere utilizzato per passare qualsiasi argomento sia richiesto.

Esempio creazione

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

void * my_fun(void * param){
    printf("This is the thread %ld that received %d\n",
        pthread_self(),(int)param);
    //pthread_self() returns the id of the thread
    return NULL;
}
int main(void){
    pthread_t t_id;
    int arg=10;
    // We need to cast the augment to a void *.
    pthread_create(&t_id, NULL, my_fun, (void *)arg);
    printf("Executed thread with id %ld\n",t_id);
    sleep(3);
}
```

Attenzione al parametro

```c
#include <stdio.h> <pthread.h> <unistd.h> //threadStack.c
int globalVar = 3; //It is shared among all threads!
void * my_fun(void * param){
    printf("This is a thread that received %d\n", *(int *)param);
    sleep(5);
    printf("This is a thread that received %d\n", *(int *)param);
    return (void *)3;
}
int main(void){
    pthread_t t_id;
    int arg=10;
    pthread_create(&t_id, NULL, my_fun, (void *)&arg);
    sleep(1);
    arg=20; //This changes the value also for the thread!
    sleep(6);
}
```

Nel kernel...

Light-**W**eight **P**rocess:
LWP: identificativo Thread
NLWP: numero di Threads nel processo

```
$ ./threadList.out & (sleep 1 && ps -eLf)
```

```c
#include <pthread.h> //threadList.c

void * my_fun(void * param){
    sleep(2);
}
int main(void){
    pthread_t t_id;
    pthread_create(&t_id, NULL, my_fun, NULL);
    sleep(2);
}
```

Terminazione

Un nuovo thread termina in uno dei seguenti modi:

- Effettuando un return dalla funzione associata al thread, specificando un valore di ritorno.
- Chiamando la funzione `void pthread_exit(void * retval);` dal thread.
- Alla richiesta di una cancellazione da parte di un altro thread.
- Alla terminazione del programma: qualche thread chiama `exit()`, o il thread principale (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: cancel **state** e cancel **type**.

Il cancel **state** di un thread definisce **se il thread deve terminare** o meno quando una richiesta di cancellazione viene ricevuta. Il cancel **type** di un thread definisce **come il thread deve terminare**.

Sia lo stato che il tipo possono essere definiti alla creazione del thread o durante la sua esecuzione, all'interno del thread stesso.

Cambiare il thread cancel state

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

Modifica il **cancel state** del thread in esecuzione. Mentre `oldstate` viene riempito con lo stato precedente, `state` può contenere una delle seguenti macro:

- `PTHREAD_CANCEL_ENABLE`: ogni richiesta di cancellazione viene gestita a seconda del **type** del thread. Questa è la modalità default.
- `PTHREAD_CANCEL_DISABLE`: ogni richiesta di cancellazione aspetterà fino a che il cancel state del thread non diventa `PTHREAD_CANCEL_ENABLE`.

Cambiare il thread cancel type

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

Modifica il **cancel type** del thread in esecuzione. Mentre `oldtype` viene riempito con il type precedente, `type` può contenere una delle seguenti macro:

- `PTHREAD_CANCEL_DEFERRED`: la terminazione aspetta l'esecuzione di un **cancellation point**. Questa è la modalità default.
- `PTHREAD_CANCEL_ASYNCHRONOUS`: la terminazione avviene appena la richiesta viene ricevuta.

Cancellation points sono delle specifiche funzioni definite nella libreria pthread.h (list). Di solito sono system calls.

Esempio cancellazione 1

```c
#include <stdio.h> <pthread.h> <unistd.h>                    //thCancel.c
void * thread1(void * param){
    pthread_setcancelstate(PTHREAD_CANCEL_DISABLE,NULL); //Change mode
    printf("Thread %ld started\n",*(pthread_t *)param); sleep(3);
    printf("Thread %ld finished\n",*(pthread_t *)param);
}
void * thread2(void * param){
    printf("Thread %ld started\n",*(pthread_t *)param); sleep(3);
    printf("Thread %ld finished\n",*(pthread_t *)param);
}
int main(void){
    pthread_t t_id1, t_id2;
    pthread_create(&t_id1, NULL, thread1, (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, thread2, (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");
}
```

Esempio cancellazione 2

```c
#include <stdio.h> <pthread.h> <unistd.h>  <string.h>   //threadCancel2.c
int tmp = 0;
void * my_fun(void * param){
    pthread_setcanceltype(*(int *)param,NULL); // Change type
    for (long unsigned i = 0; i < (0x9FFF0000); i++); //just wait
    tmp++; open("/tmp/tmp",O_RDONLY);  //Cancellation point!
}
int main(int argc, char ** argv){ //call program with 'async' or 'defer'
    pthread_t t_id1;
    int arg1 = PTHREAD_CANCEL_ASYNCHRONOUS, arg2 = PTHREAD_CANCEL_DEFERRED;
    pthread_create(&t_id1, NULL, my_fun, (void *)&arg1); sleep(1); //Create
    pthread_cancel(t_id1); sleep(5); //Cancel
    printf("Has Tmp been updated? %d\n",tmp);
    pthread_create(&t_id1, NULL, my_fun, (void *)&arg2); sleep(1); //Create
    pthread_cancel(t_id1); sleep(5); //Cancel
    printf("Has Tmp been updated? %d\n",tmp);
}
```

Aspettare un thread: join

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

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

Questa funzione 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` nella funzione associata al thread), 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, ed 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;'
}
int main(void){
    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);
}
```

Lo stack viene deallocato

Quando viene restituito un valore da un thread, se questo è un puntatore ad una variabile definita all'interno del thread (nel suo stack), c'è la possibilità che essa cessi di esistere!

```c
#include <stdio.h> <pthread.h> <unistd.h>                    //thJoinStack.c
void * my_fun(void * param){
    int val = 34;
    pthread_exit((void *)&val); //or 'return (void *) val;'
}
int main(void){
    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
    printf("Thread %ld returned '%d'\n",t_id,*(int *)retFromThread);
}
```

Thread loop creation

Quando viene restituito un valore da un thread, se questo è un puntatore ad una variabile definita all'interno del thread (nel suo stack), c'è la possibilità che essa cessi di esistere!

```c
#include <stdio.h> <pthread.h> <unistd.h>                    //thLoop.c
void * my_fun(void * param){
    printf("Thread %ld started\n",*(int *)param);
}
int main(void){
    pthread_t t_id;
    for(int i = 0; i<10; i++){
        int b = i;
        pthread_create(&t_id, NULL, my_fun, (void *)&b);
    }
    sleep(2);
}
```

Esempio join II

```c
#include <stdio.h> <pthread.h> <unistd.h> //threadJoin2.c
void * my_fun(void *param){
    printf("This is a thread that received %d\n", *(int *)param);
    return (void *)3;
}
int main(void){
    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); //A pointer to a void pointer
    printf("retval=%d\n", retval);
}
```

Esempio join III

```c
#include <stdio.h> <pthread.h> <unistd.h> //threadJoin3.c
void * my_fun(void *param){
    printf("This is a thread that received %d\n", *(int *)param);
    int i = *(int *)param * 2; //Local variable ceases to exist!
    return (void *)&i;
}
int main(void){
    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); //A pointer to a void pointer
    printf("retval=%d\n", retval);
}
```

Detach state di un thread

I thread sono creati per impostazione predefinita nello stato joinable, che consente a un altro thread di attendere la loro terminazione tramite il comando `pthread_join()`. I thread joinable non rilasciano le loro risorse alla terminazione, ma quando un thread li aspetta (salvando lo stato di uscita, come i sottoprocessi), o alla terminazione del processo stesso. Al contrario, i thread detached rilasciano le loro risorse immediatamente al termine, ma non permettono ad altri processi di aspettarli.

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

Cambiare il detach state

```
int pthread_detach(pthread_t thread);
```

Questo comando può essere eseguito da un thread qualunque e cambia il detach state di `thread` da joinable a detached.

Ricorda che una volta cambiato lo stato non si può invertire.

Attributi di un thread

Ogni thread viene creato con gli attributi specificati nella struttura `pthread_attr_t`. Questa struttura, analogamente a quella utilizzata per gestire le maschere di segnale, è un oggetto utilizzato solo quando viene creato un thread ed è quindi indipendente da esso (se cambia, gli attributi del thread non cambiano).

La struttura deve essere inizializzata, il che imposta tutti gli attributi al loro valore predefinito. Una volta utilizzata e non più necessaria, la struct deve essere distrutta.

I vari attributi della struttura possono e devono essere modificati individualmente con alcune funzioni dedicate.

Attributi di un thread

`int pthread_attr_init(pthread_attr_t *attr)`
Inizializza la struttura con tutti gli attributi default.

`int pthread_attr_destroy(pthread_attr_t *attr)`
Distrugge la struttura.

`int pthread_attr_setXXXX(pthread_attr_t *attr, params);`
Imposta un certo attributo ad un certo valore.

`int pthread_attr_getXXXX(const pthread_attr_t *attr, params);`
Ottiene un certo attributo

xxxx 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
- `...sigmask_np(pthread_attr_t *attr,const sigset_t *sigmask);`
- `...affinity_np(...)`
- `...setguardsize(...)`
- `...inheritsched(...)`
- `...schedparam(...)`
- `...schedpolicy(...)`
- altri

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;
}
int main(void){
    pthread_t t_id; pthread_attr_t attr;
    int arg=10, detachSTate;
    pthread_attr_init(&attr);
    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); pthread_attr_destroy(&attr);
    int esito = pthread_join(t_id, (void **)&detachSTate);
    printf("Esito '%d' is different 0\n", esito);
}
```

Threads e segnali

Quando viene inviato un segnale ad un processo non si può sapere quale thread andrà a gestirlo. Per evitare problemi o comportamenti inattesi, è importante gestire correttamente le maschere dei segnali dei singoli thread con `pthread_attr_setsigmask_np(pthread_attr_t *attr,const sigset_t *sigmask)` la quale usa `*sigmask` per impostare la maschera dei segnali nella struttura `*attr`.

Tuttavia, questa funzione non è standard e potrebbe non essere implementata. Per questo motivo, l'alternativa è usare `pthread_sigmask(int how, const sigset_t *restrict set, sigset_t *restrict oset);` dal thread stesso. **NB**: in presenza di threads va usata `pthread_sigmask` e non `sigprocmask()` in tutto il programma!

Threads e segnali

Visto che alla creazione del thread viene ereditata la maschera dei segnali, per bloccare correttamente un segnale nel thread (in assenza di `pthread_attr_setsigmask_np()`) si deve:

- Bloccare tutti i segnali nel thread principale, salvando la maschera corrente
- Creare un nuovo thread, nel quale impostare una nuova maschera desiderata con `pthread_sigmask()`
- Ripristinare la maschera dei segnali salvata nel thread originale.

È poi possibile usare funzioni come `sigwait()` e `sigwaitinfo()` per gestire l'attesa di un segnale. Infine, è possibile inviare un segnale ad un thread specifico usando `int pthread_kill(pthread_t thread, int sig);`

Bloccare segnali in un thread

```
#include <stdio.h> <pthread.h> <unistd.h> //threadAttr.c
void * my_fun(void *param){
    sigset_t set;
    sigemptyset(&set);
    sigaddset(&set, SIGUSR1);
    pthread_sigmask(SIG_SETMASK, &set, NULL);
    printf("Thread changed signal mask\n");
    while(1) pause();
}
void handler(int sig, siginfo_t *si, void *uc){
    printf("Signal %d received in thread %ld\n", sig, pthread_self());
}
...
```

Bloccare segnali in un thread

```
...
int main(void){
    pthread_t t_id;
    struct sigaction sa;
    sa.sa_flags = SA_SIGINFO;
    sa.sa_sigaction = handler;
    sigaction(SIGUSR1, &sa, NULL);
    sigset_t new_set,old_set;
    sigfillset(&new_set);
    pthread_sigmask(SIG_BLOCK,&new_set, &old_set);
    pthread_create(&t_id, NULL, my_fun, NULL);
    pthread_sigmask(SIG_SETMASK, &old_set, NULL);
    sleep(1);
    printf("Sending signal to thread %ld\n", t_id);
    pthread_kill(t_id, SIGUSR1);
    sleep(1);
}
```

Mare un segnale ad un thread

Generalmente non è possibile inviare un segnale ad un thread specifico. L'unico modo è assicurarsi che solo un thread non abbia bloccato quel segnale, affinchè esso sia l'unico che andrà a gestirlo.

È tuttavia possibile inviare un segnale ad un thread da dentro il processo che ha creato il thread stesso.

```
int pthread_kill(pthread_t thread, int sig);
```

Threads vs fork

	Processi figli	**Threads**
Punto di partenza	Dopo il fork	Funzione dedicata
Condivisione	niente	Dati, codice, ciclo vitale
Attesa terminazione	wait	join

Mutex

Il problema della sincronizzazione

Quando eseguiamo un programma con più thread essi condividono alcune risorse, tra le quali le variabili globali. Se entrambi i thread accedono ad una sezione di codice condivisa ed hanno la necessità di accedervi in maniera esclusiva allora dobbiamo instaurare una sincronizzazione. I risultati, altrimenti, potrebbero essere inaspettati.

Esempio

```
#include <pthread.h> <stdlib.h> <unistd.h><stdio.h>      //syncProblem.c
pthread_t tid[2];
int counter = 0;
void *thr1(void *arg){
    counter = 1;
    printf("Thread 1 has started with counter %d\n",counter);
    for (long unsigned i = 0; i < (0x00FF0000); i++); //wait some cycles
    counter += 1;
    printf("Thread 1 expects 2 and has: %d\n", counter);
}
void *thr2(void *arg){
    counter = 10;
    printf("Thread 2 has started with counter %d\n",counter);
    for (long unsigned i = 0; i < (0xFFF0000); i++); //wait some cycles
    counter += 1;
    printf("Thread 2 expects 11 and has: %d\n", counter);
}
...
```

Esempio

```
...
void main(void){
    pthread_create(&(tid[0]), NULL, thr1,NULL);
    pthread_create(&(tid[1]), NULL, thr2,NULL);
    pthread_join(tid[0], NULL);
    pthread_join(tid[1], NULL);
}
```

Una possibile soluzione: mutex

I mutex sono dei semafori imposti ai thread. Essi possono proteggere una determinata sezione di codice, consentendo ad un thread di accedervi in maniera esclusiva fino allo sblocco del semaforo. Ogni thread che vorrà accedere alla stessa sezione di codice dovrà aspettare che il semaforo sia sbloccato, andando in sleep fino alla sua prossima schedulazione.

I mutex vanno inizializzati e poi assegnati ad una determinata sezione di codice. Il blocco e sblocco è manuale.

NB: i mutex non regolano l'accesso alla memoria o alle variabile, ma solo a porzioni di codice.

Creare e distruggere un mutex

```
int pthread_mutex_init(pthread_mutex_t *restrict mutex, const
                       pthread_mutexattr_t *restrict attr)

int pthread_mutex_destroy(pthread_mutex_t *mutex)
```

```c
#include <pthread.h>                                    //createMutex.c
pthread_mutex_t lock;
int main(void){
    pthread_mutex_init(&lock, NULL); // Create mutex with default attrs
    pthread_mutex_destroy(&lock); // Destroy mutex (it can be re-init)
}
```

Bloccaggio e sbloccaggio di un mutex

```
int pthread_mutex_lock(pthread_mutex_t *mutex)
int pthread_mutex_unlock(pthread_mutex_t *mutex)
```

Dopo essere stato creato, un mutex deve essere bloccato per essere efficace. Non appena un thread lo blocca, un altro thread deve attendere che venga sbloccato prima di procedere al suo blocco.

Quando si richiama il blocco, un thread attende che il mutex sia libero e poi lo blocca.

```c
#include <pthread.h>

pthread_mutex_t lock;
void * thread(void *){
    pthread_mutex_lock(&lock);
    /* Protected section of code */
    pthread_mutex_unlock(&lock);
}
```

Controllare lo stato di un mutex

`int pthread_mutex_trylock(pthread_mutex_t *mutex)`

Sebbene non sia possibile controllare lo stato di un mutex senza bloccarlo, è possibile provare a bloccarlo senza però mettersi in attesa qualora fosse già bloccato.

```
#include <pthread.h>
pthread_mutex_t lock;
void * thread(void *){
    int lockResult = pthread_mutex_trylock(&lock);
    if(lockResult == 0) pthread_mutex_unlock(&lock);
    if(lockResult == EBUSY) printf("Mutex was already locked");

}
```

Example 1/2

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

pthread_mutex_t lock;
pthread_t tid[2];
int counter = 0;

void* thr1(void* arg){
    pthread_mutex_lock(&lock);
    counter = 1;
    printf("Thread 1 has started with counter %d\n",counter);
    for (long unsigned i = 0; i < (0x00FF0000); i++);
    counter += 1;
    printf("Thread 1 expects 2 and has: %d\n", counter);
    pthread_mutex_unlock(&lock);
}...
```

Example 2/2

```
...
void * thr2(void* arg){
    pthread_mutex_lock(&lock);
    counter = 10;
    printf("Thread 2 has started with counter %d\n",counter);
    for (long unsigned i = 0; i < (0xFFF0000); i++);
    counter += 1;
    printf("Thread 2 expects 11 and has: %d\n", counter);
    pthread_mutex_unlock(&lock);
}
int main(void){
    pthread_mutex_init(&lock, NULL);
    pthread_create(&(tid[0]), NULL, thr1,NULL);
    pthread_create(&(tid[1]), NULL, thr2,NULL);
    pthread_join(tid[0], NULL);
    pthread_join(tid[1], NULL);
    pthread_mutex_destroy(&lock);
}
```

Tipi di mutex

	Blocca quando già bloccato	Sblocca quando bloccato da altri	Sblocca se già sbloccato
PTHREAD_MUTEX_NORMAL	deadlock	undefined	undefined
PTHREAD_MUTEX_ERRORCHECK	returns error	returns error	returns error
PTHREAD_MUTEX_RECURSIVE	Lock count++ so that it requires the same number of unlocks	returns error	returns error
PTHREAD_MUTEX_DEFAULT	undefined	undefined	undefined

Example 1/2

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

pthread_mutex_t lock;
pthread_t tid[2];
int counter = 0;

void* thr1(void* arg){
    pthread_mutex_lock(&lock);
    pthread_mutex_lock(&lock);
    counter = 1;
    printf("Thread 1 has started with counter %d\n",counter);
    for (long unsigned i = 0; i < (0x00FF0000); i++);
    counter += 1;
    pthread_mutex_unlock(&lock);
    printf("Thread 1 expects 2 and has: %d\n", counter);
    pthread_mutex_unlock(&lock);
}...
```

Example 2/2

```
...
void* thr2(void* arg){
    pthread_mutex_lock(&lock); pthread_mutex_lock(&lock);
    counter = 10;
    printf("Thread 2 has started with counter %d\n",counter);
    for (long unsigned i = 0; i < (0xFFF0000); i++);
    counter += 1;
    pthread_mutex_unlock(&lock);
    printf("Thread 2 expects 11 and has: %d\n", counter);
}
void main(){
    pthread_mutexattr_t attr;
    pthread_mutexattr_settype(&attr,PTHREAD_MUTEX_RECURSIVE);
    pthread_mutex_init(&lock, &attr);
    pthread_create(&(tid[0]), NULL, thr1,NULL);
    pthread_create(&(tid[1]), NULL, thr2,NULL);
    pthread_join(tid[0], NULL); pthread_join(tid[1], NULL);
    pthread_mutex_destroy(&lock);
}
```

CONCLUSIONI

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).

I “MUTEX” sono un metodo semplice ma efficace per eseguire sezioni critiche in processi multithread.

È importante limitare al massimo la sezione critica utilizzando lock/unlock per la porzione di codice più piccola possibile, e solo se assolutamente necessario (per esempio quando possono capitare accessi concorrenti).