# 4' Esercitazione

https://politecnicomilano.webex.com/meet/gianenrico.conti

Gian Enrico Conti

Threads

Esercizi:

- creazione, exit e join
- passaggio si argomenti generici mediante void*
- ritorno di valori mediante void*
- problemi legati al ritorno di valori per puntatore a variabili automatiche
- sincronizzazione con semafori binari
- sezioni critiche e mutex

# Outline

- **Threads**
  - Creazione – *create*
  - Attesa della terminazione – *join*
  - Sincronizzazione
    - Semafori
    - Mutex

# Threads - Creazione (1)

- **In LINUX/UNIX/POSIX**

```
include <pthread.h>
```

- **Il comportamento del thread è determinato da una funzione, ("body")**
- **Molto generica"**
  - **Riceve puntatore a void**
  - **Ritorna un puntatore a void**
  - **Cast necessari (sia nella f. che nel main())**

```
void* thread_function(void* param){
                    …
}
```

# Threads - Creazione (2)

- **Ogni thread ha un thread ID di tipo pthread_t:**

```
pthread_t thread_ID;
```

- **Per creare un thread si usa pthread_create, con quattro argomenti:**

```
int pthread_create( pthread_t *thread,
                    const pthread_attr_t *attr,
                    void *(*start_routine)(void*),
                    void *arg);
```

  - Il primo argomento è un puntatore ad una variabile di tipo pthread_t che conterrà l'ID del thread
  - Il secondo argomento è un puntatore ad un oggetto di tipo thread_attribute; se NULL il thread viene creato con i suoi parametri di default
  - Il terzo argomento è un puntatore alla thread function;
  - Il quarto argomento di tipo `void*` è il parametro da passare alla thread function.

# Thread – Creazione (3)

- **Esempio:**

```c
//EX4_es01
#include <stdio.h>
#include <pthread.h>

void * thread_func (void * p){
    printf("This is the thread\n");
    return NULL;
}

int main(){
    pthread_t t;
    int i;
    pthread_create(&t, NULL, &thread_func,NULL);
    for(i=0;i<10000; i++){
        //do something
    }
}
```

- **proviamo..**

# Thread – Creazione (3)

- Su alcune "distro":

```
test.c:(.text+0x7a): undefined reference to `pthread_create'
test.c:(.text+0xc1): undefined reference to `pthread_join'
```

- Manca la lib (NON header..):

```
gcc -Wall -pthread test.c -o test
```

- rilanciamo..

# Thread – Creazione (3)

- **NON appare il messaggio.. perche main finisce**

  **"temp FIX": (...** (seguono dettagli..)

```c
#include <stdio.h>
#include <pthread.h>

void * thread_func (void * p){
    printf("This is the thread\n");
    return NULL;
}

int main(){
    pthread_t t;
    int i;
    pthread_create(&t, NULL, &thread_func,NULL);
    for(i=0;i<10000; i++){
        //do something
    }
    scanf("%d", &i); // keep main alive..
}
```

# Thread – simmetria con PID

- **Ogni Thread ha un id**
- **Per conoscere proprio id**

```
pthread_t pthread_self(void);
```

```
(Esempio piu' avanti.)
```

# Thread vs Process..

1) thread + leggeri
2) I "worker" difficoltoso scambio dati fra processi
3) processi "isolati"
4) context switching +pesante x processi
5) comunicazione inter-task

# Thread vs Process

Threads share a common address space, thereby avoiding a lot of the inefficiencies of multiple processes.

- The kernel does not need to make a new independent copy of the process memory space, file descriptors, etc. This saves a lot of CPU time, making thread creation ten to a hundred times faster than a new process creation. Because of this, you can use a whole bunch of threads and not worry about the CPU and memory overhead incurred. This means you can generally create threads whenever it makes sense in your program.
- Less time to terminate a thread than a process.
- Context switching between threads is much faster then context switching between processes (context switching means that the system switches from running one thread or process, to running another thread or process)
- Less communication overheads -- communicating between the threads of one process is simple because the threads share the address space. Data produced by one thread is immediately available to all the other threads.

On the other hand, because threads in a group all use the same memory space, if one of them corrupts the contents of its memory, other threads might suffer as well. With processes, the operating system normally protects processes from one another, and thus if one corrupts its own memory space, other processes won't suffer.

# Thread perche' si usano

- UI non bloccata da operazioni lunghe / rete
- sfruttano i core della CPU

...

# Thread perche' si usano: network

Esempio da:

https://www.cs.dartmouth.edu/~campbell/cs50/echoServer.c

Parti salienti:

```
listen(listenfd, LISTENQ);

 printf("%s\n","Server running...waiting for connections.");

 for ( ; ; ) {

  clilen = sizeof(cliaddr);
  connfd = accept(listenfd, (struct sockaddr *) &cliaddr, &clilen);
  printf("%s\n","Received request...");

  while ( (n = recv(connfd, buf, MAXLINE,0)) > 0)  {
   printf("%s","String received from and resent to the client:");
   puts(buf);
   send(connfd, buf, n, 0); // send back...
  }
```

**NON gestisce client multipli.. in piu accept && recv sono bloccante.**

# Thread – Terminazione e attesa (1)

- **Un thread termina con la funzione** `pthread_exit` **o con la normale** `return`

```
void pthread_exit(void *value_ptr);
```

Il parametro passato alla pthread_exit, opportunamente castato a void*, è il return value del thread

- **Una chiamata a** `exit(int)` **all'interno del thread causa la terminazione del processo padre e di conseguenza di tutti gli altri thread**

- **Se il processo padre termina prima di uno dei suoi thread possono nascere problemi in quanto la memoria cui tali thread fanno accesso viene deallocata e tali thread vengono terminati insieme al padre**

# Thread – Terminazione e attesa (2)

- **Per prevenire tale effetto, nonché per attendere un thread all'interno di un altro thread, si usa la funzione**

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

Il primo parametro è il thread ID da attendere,
il secondo è un puntatore a puntatore a void che prende il valore di uscita del thread

Se non serve, 2' param usare NULL

# Thread – Terminazione e attesa (3)

- **Un thread non dovrebbe mai attendere se stesso, per evitare tale circostanza è opportuno controllare il proprio thread ID attraverso la funzione pthread_self**

- **Per controllare l'uguaglianza di due thread ID si usa la funzione pthread_equal**

```
int pthread_equal(pthread_t t1, pthread_t t2);
```

# Thread – Terminazione e attesa

- **Esempio: (EX4_es02.c)**

```c
//EX4_es02
#include <stdio.h>
#include <pthread.h>

void * thread_func (void * p){
    // cast back:
    long n = (long)p;
    printf("This is the thread\n got: %ld\n", n);
    return NULL;
}

int main(){
    pthread_t t1,t2;
    long t_num=1;
    pthread_create(&t1, NULL, &thread_func,(void*)t_num);
    t_num=2;
    pthread_create(&t2, NULL, &thread_func,(void*)t_num);

    pthread_join(t1,(void*)&t_num);
    printf("Received thread %ld\n",t_num);
    pthread_join(t2, (void*)&t_num);
    printf("Received thread %ld\n",t_num);
    return 0;
}
```

- NOTARE cosa stampa.... Provare a ritornare il DOPPIO...

# Thread – Terminazione e attesa

- **Esempio: (EX4_es02_B.c)**

```
void * thread_func (void * p){
    // cast back:
    long n = (long)p;
    printf("This is the thread\n got: %ld\n", n);
    return (void*)(n*2);    // was: return NULL;
}

int main(){
    pthread_t t1,t2;
    long t_num=1;
    pthread_create(&t1, NULL, &thread_func,(void*)t_num);
    t_num=2;
    pthread_create(&t2, NULL, &thread_func,(void*)t_num);

    long ris1, ris2;
    pthread_join(t1,(void*)&ris1);
    printf("Received thread %ld\n",ris1);
    pthread_join(t2, (void*)&ris2);
    printf("Received thread %ld\n",ris2);
    return 0;
}
```

# Thread – Terminazione e attesa (2)

- **Esempio: (EX4_es02_C.c)**

```c
#include <stdio.h>
#include <pthread.h>

void * thread_func (void * p){
    // cast back:
    long n = (long)p;
    printf("thread got: %ld\n", n);
    return (void*)(n*2);    // was: return NULL;
}

#define MAX_THREADS 10

int main(){
    pthread_t t[MAX_THREADS];
    long t_num=0;
    for (t_num=0; t_num<MAX_THREADS; t_num++)
        pthread_create(&t[t_num], NULL, &thread_func,(void*)t_num);

    for (t_num=0; t_num<MAX_THREADS; t_num++){
        long ris;
        pthread_join(t[t_num],(void*)&ris);
        printf("Received from thread %ld\n",ris);
    }

    return 0;
}
```

- **Al run si noti la sequenza di stampa di ritorno... join aspetta su un ID in ordine...**

# Thread - Accesso ai dati (EX4_es03)

```c
//EX4_es03

#include <stdio.h>
#include <pthread.h>

#define MAX 100000
static long shared = 0;

void * thread_func (void * p){
    int i;
    long n = (long)p;
    for (i=0; i<MAX; i++){
        shared+=n;
    }
    return NULL;
}

int main(){
    pthread_t t1,t2;
    long t_num=1;
    pthread_create(&t1, NULL, &thread_func,(void*)t_num);
    t_num=2;
    pthread_create(&t2, NULL, &thread_func,(void*)t_num);
    pthread_join(t1,NULL);
    pthread_join(t2, NULL);

    printf("SHARED: %ld\n", shared);

    return 0;
}
```

- Cosa mi aspetto?

# Thread - Accesso ai dati (EX4_es03)

- Dovrei avere 100'000 *1 + 100'000*2 = 30'000
- Ad ogni run diverso..
-

# Thread - Sincronizzazione

```
void * thread_func (void * p){
    int i;
    long n = (long)p;
    for (i=0; i<MAX; i++){
        shared+=n;
    }
    return NULL;
}
```

- **A livello registri:**

  `shared+=n;` non e' atomica

  A) read di *shared* dalla RAM in R1
  B) add di R1 e n (ris. pes. In R1)
  C) write di R1 indietro nella RAM...

  MA lo scheduler interrompe....

# Thread - Sincronizzazione

- **Poiché i thread vengono schedulati dal sistema operativo in maniera non prevedibile, è opportuno utilizzare opportuni meccanismi di sincronizzazione per evitare *race condition* nell'utilizzo di dati condivisi**

**Esempio di race condition:**

```
void * thread_func (void * p){
    int i;
    long n = (long)p;
    for (i=0; i<MAX; i++){
        shared+=n;
    }
    return NULL;
}
```



A run-time, il sistema operativo in questo punto interrompe l'esecuzione di t1 e passa a t2...

# Thread - Sincronizzazione

**Bad case:**

- **A livello registri:**

  `shared+=n; non e' atomica`

```
T1
A) read di shared dalla RAM in R1
--
--
B) add di R1 e n (ris. pes. In R1)
--
C) write di R1 indietro nella RAM (1)
```

```
T2
--
A) read di shared dalla RAM in R1
B) add di R1 e n (ris. pes. In R1)
--
C) write di R1 indietro nella RAM... (2)
```

QUINDI shared vale 1!

# Thread - Sincronizzazione: MUTEX

**Un mutex, (MUTual EXclusion lock), è una primitiva di sincronizzazione che fa leva su un concetto molte semplice:**

- ***Solo un thread alla volta può detenere il lock sul mutex: se qualche altro thread tenta di effettuare il lock viene messo in attesa e bloccato finche il thread che detiene il lock non lo rilascia attraverso un'operazione di unlock.***

- **Per creare un mutex si dichiara una v. del tipo pthread_mutex_t, tale variabile detiene l'identificativo del mutex**
- **Per inizializzare il mutex si utilizza la funzione pthread_mutex_init, che prende come 1' argomento la variabile di tipo pthread_mutex_t e come 2' argomento una variabile di tipo mutex attribute (se il secondo argomento è posto a NULL il mutex viene inizializzato con gli attributi di default)**
- **In alternativa per inizializzare un mutex si può assegnare alla variabile di tipo pthread_mutex_t il valore speciale PTHREAD_MUTEX_INITIALIZER**

```
/*primo metodo*/
pthread_mutex_t mutex;
pthread_mutex_init (&mutex, NULL);

/*secondo metodo*/
pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;
```

# Thread - Sincronizzazione

## Mutex

- Su un mutex sono possibili due operazioni fondamentali: l'operazione di lock e l'operazione di unlock, tramite le seguenti funzioni:

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

- È inoltre possibile un'altra operazione, la trylock, che non è bloccante:

```
int pthread_mutex_trylock(pthread_mutex_t *mutex);
```

# Thread - Sincronizzazione

- **Tipi di mutex:**
  - *fast mutex*: modalità di **default,** è sempre bloccante, anche se lo stesso thread chiama in sequenza due lock sullo stesso mutex ,in tal caso si ha un **deadlock** irrisolvibile
  - *recursive*: se il thread che detiene il lock effettua altre operazioni di lock, l'operazione non risulta bloccante e pertanto non si ha deadlock
  - *error checking*: ritorna un errore nel caso in cui il thread che detiene il lock su un mutex tenti di effettuare una nuova operazione di lock in sequenza
  - Noi utilizzeremo di default i fast mutex, per settare la tipologia recursive o error checking è opportuno inizializzare convenientemente gli attributi mutex

# Thread - Sincronizzazione

## Mutex – Esempio

```
#include <pthread.h>
int me;
pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;

void* thread_func (void * arg){
      … /*do something*/
      pthread_mutex_lock(&mutex);
      me= (int)arg;
      …/*do something*/
      printf("My ID: %d", me);
      pthread_mutex_unlock(&mutex);
}

int main(){
    pthread_t t1,t2;
    int t_num;
    t_num=1;
    pthread_create(&t1, NULL, &thread_func,(void*)t_num);
    t_num=2;
    pthread_create(&t2, NULL, &thread_func, (void*)t_num);
    pthread_join(t1,NULL);
    pthread_join(t2,NULL);
    return 0;
}
```

## In questo modo viene evitata la race condition

# Thread - Sincronizzazione: EX4_es03 FIX

```c
//EX4_es03_FIX_with_mutex.c

#include <stdio.h>
#include <pthread.h>

#define MAX 10000
static long shared = 0;
static pthread_mutex_t mutex= PTHREAD_MUTEX_INITIALIZER;

void * thread_func (void * p){
    int i;
    pthread_mutex_lock(&mutex);
    long n = (long)p;
    for (i=0; i<MAX; i++){
        shared+=n;
    }
    pthread_mutex_unlock(&mutex);
    return NULL;
}

int main(){
    pthread_t t1,t2;
    long t_num=1;
    pthread_create(&t1, NULL, &thread_func,(void*)t_num);
    t_num=2;
    pthread_create(&t2, NULL, &thread_func,(void*)t_num);
    pthread_join(t1,NULL);
    pthread_join(t2, NULL);

    printf("SHARED: %ld\n", shared);
    return 0;
}
```

In questo modo viene evitata la race condition... si puo' fare **meglio**?

# Thread - Sincronizzazione: EX4_es03 FIX ottimizzato

```c
//EX4_es03_FIX_with_mutex.c

#include <stdio.h>
#include <pthread.h>

#define MAX 10000
static long shared = 0;
static pthread_mutex_t mutex= PTHREAD_MUTEX_INITIALIZER;

void * thread_func (void * p){
    int i;
    long n = (long)p;
    for (i=0; i<MAX; i++){
        pthread_mutex_lock(&mutex);
        shared+=n;
        pthread_mutex_unlock(&mutex);
    }
   return NULL;
}

int main(){
    pthread_t t1,t2;
    long t_num=1;
    pthread_create(&t1, NULL, &thread_func,(void*)t_num);
    t_num=2;
    pthread_create(&t2, NULL, &thread_func,(void*)t_num);
    pthread_join(t1,NULL);
    pthread_join(t2, NULL);

    printf("SHARED: %ld\n", shared);
    return 0;
}
```

Limitare il piu possibile AMPIE zone di codice *sotto* mutex.

# Thread - Sincronizzazione

## Mutex – Deadlock

- Occorre evitare situazioni di deadlock distribuito, ovvero un intreccio delle condizioni di attesa che rende impossibile il proseguire del flusso di esecuzione di due o più thread, bloccandoli perennemente.

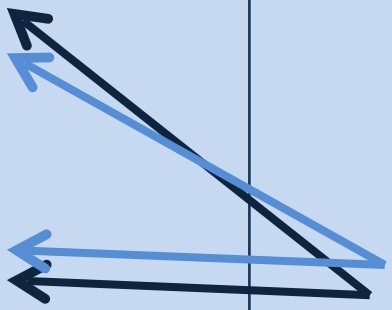
**Esempio:**

```
#include <pthread.h>
pthread_mutex_t mutex1=
PTHREAD_MUTEX_INITIALIZER;
pthread_mutex_t mutex2=
PTHREAD_MUTEX_INITIALIZER;

void* thread_func_1 (void *){
      … /*do something*/
      pthread_mutex_lock (&mutex1);
      pthread_mutex_lock (&mutex2);
      …
}

void* thread_func_2 (void *){
      … /*do something*/
      pthread_mutex_lock (&mutex2);
      pthread_mutex_lock (&mutex1);
      …
}
```

```
int main(){
    pthread_t t1,t2;
    pthread_create(&t1, NULL, &thread_func_1,NULL);
    pthread_create(&t2, NULL, &thread_func_2,NULL);
    pthread_join(t1,NULL);
    pthread_join(t2,NULL);
    return 0;
}
```



Se il sistema operativo schedula t1 fino al lock di mutex1, poi passa l'esecuzione a t2 che blocca mutex2, poi ritorna a t1 si avrà un deadlock: t1 non potrà predere il lock su mutex2 e t2 non potrà prendere più il lock su mutex1 in quanto entrambi sono già bloccati. Tale condizione di attesa durerà indefinitamente.

Una buona prassi per evitare deadlock di questo tipo è quella di richiedere i lock, nonché di rilasciarli, nello stesso ordine in ogni thread.

# Thread - Sincronizzazione

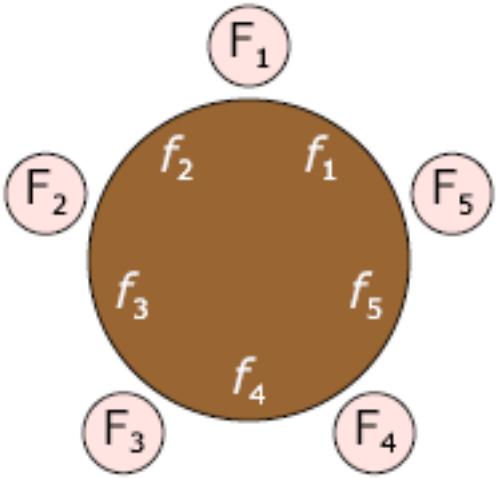
## Mutex – Deadlock

- **Occorre evitare situazioni di deadlock distribuito, ovvero un intreccio delle condizioni di attesa che rende impossibile il proseguire del flusso di esecuzione di due o più thread, bloccandoli perennemente.**

**Vale non solo x mutex.**

**In letteratura:**

**https://it.wikipedia.org/wiki/Problema_dei_filosofi_a_cena**



# Thread - Sincronizzazione

## Semafori

- **Un semaforo è un meccanismo di sincronizzazione basato su un contatore**
- **Se è maggiore di zero, i thread proseguono**
- **Se il contatore è zero, i thread si bloccano finché il contatore non viene incrementato ad un valore positivo.**
- **Su un semaforo sono possibili due operazioni fondamentali:**
  - *Wait*: questa operazione decrementa di uno il contatore del semaforo. Se il contatore è a zero si blocca nell'attesa che il contatore venga incrementato. Una volta che il contatore è incrementato, la wait si risveglia e decrementa di uno il contatore.
  - *Post*: incrementa di uno il contatore del semaforo. Se il contatore era a zero, oltre ad incrementare il contatore, la post risveglia uno dei thread che erano rimasti bloccati sulla wait.

# Thread - Sincronizzazione

## Semafori (2)

- **Per utilizzare i semafori occorre includere la header: <semaphore.h>**
- **Per creare un semaforo si dichiara una variabile di tipo sem_t, poi si invoca la funzione sem_init:**

La funzione **int sem_init(sem_t **sem*, int *pshared*, unsigned int *value*)**, prende come primo parametro un puntatore alla variabile sem_t, come secondo parametro sempre zero e come terzo parametro il valore a cui inizializzare il contatore del semaforo

```
#include <semaphore.h>
sem_t semaforo;
sem_init(&semaforo, 0, 5); /*inizializza il semaforo con un valore iniziale pari a 5*/
```

# Thread - Sincronizzazione

## Semafori

- **Sui semafori sono possibili, come detto operazioni di wait e di post, con il significato descritto nella slide precedente.**

```
int sem_wait(sem_t *sem);
int sem_trywait(sem_t *sem);
int sem_post(sem_t *sem);
```

La trywait non è bloccante, nel caso il contatore sia zero va avanti, senza decrementare il contatore.

- **Infine, quando un semaforo non serve più, occorre deallocarlo per mezzo della funzione sem_destroy**

```
int sem_destroy(sem_t *sem);
```

- **Di seguito un tipico esempio di produttore-consumatore…**

# Thread - Sincronizzazione

## Semafori – Esempio

```
#include <pthread.h>
#include <semaphore.h>
pthread_mutex_t mutex=PTHREAD_MUTEX_INITIALIZER;
sem_t sem;
int a; /*this should not be negative*/

void* producer (void *){
      pthread_mutex_lock(&mutex);
      a++;
      sem_post(&sem);
      pthread_mutex_unlock(&mutex);
}

void* consumer (void *){
      … /*do something*/
      sem_wait(&sem);
      pthread_mutex_lock(&mutex);
      a--;
      /*do something*/
      …
     pthread_mutex_unlock(&mutex);
}
```

```
int main(){
   pthread_t p[3], c[3];
   int i;
   sem_init(&sem,0, 0);
   for(i=0;i<3;i++)
    pthread_create(&p[i], NULL, &producer,NULL);
   for(i=0;i<3;i++)
    pthread_create(&c[i], NULL, &consumer,NULL);
   /*do something for sometime*/
   …
   for(i=0;i<3;i++)
    pthread_join(c[i],NULL);
   for(i=0;i<3;i++)
    pthread_join(p[i],NULL);

   sem_destroy(&sem);

   return 0;
}
```

# Thread - passaggio parametri: struct/Array

**A) non e' possible cambiare il prototipo della thread f.**

**B) si usa Cast**

```c
//EX4_es04
#include <stdio.h>
#include <pthread.h>

typedef struct MyPoint{
    int x,y;
}MyPoint;


void * thread_func_Arr (void * p){
    // cast back to string
    char * s = (char*)p;
    printf("got: %s\n", s);
    return NULL;
}
void * thread_func_Struct (void * p){
    // cast back:
    MyPoint * pp = (MyPoint*)p;
    printf( "got: %d %d \n", pp->x, pp->y);
    return NULL;
}

int main(){
    pthread_t t1,t2;
    char msg[] = "HELLO";
    pthread_create(&t1, NULL, &thread_func_Arr,(void*)msg);

    MyPoint p = {10,20};
    pthread_create(&t2, NULL, &thread_func_Struct,(void*)&p);

    pthread_join(t1,NULL);
    pthread_join(t2,NULL);
    return 0;
}
```

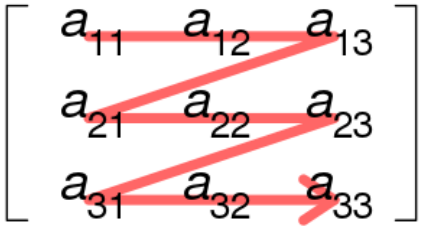
# Thread - core/CPU: suddividere carico:

Es matrice molto grande di cui voglio p.es. trovare il massimo:

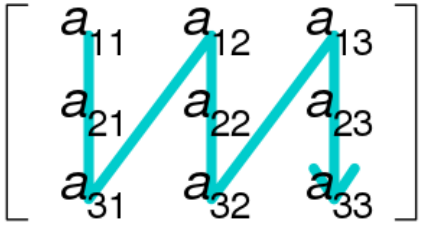
- "Spezzo" in sotto matrici
- alla "join" cerco il MAX dai max parziali

* Come "spezzo"?
* Quante "fette"?

Row-major order




Column-major order



# Thread - core/CPU: sched_getcpu

```c
// LINUX ONLY
#define _GNU_SOURCE
#include <assert.h>
#include <sched.h>
#include <stdbool.h>
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <pthread.h>

void * thread_func (void * p){
    int cpu = sched_getcpu();
    pthread_t t_id = pthread_self();
    printf("Th id %d  - on CPU: %d\n",t_id, cpu);
    return NULL;
}


int  main(void){
    pthread_t t1,t2;
    long t_num=1;
    pthread_create(&t1, NULL, &thread_func,(void*)t_num);
    t_num=2;
    pthread_create(&t2, NULL, &thread_func,(void*)t_num);
    pthread_join(t1,NULL);
    pthread_join(t2, NULL);
    printf("DONE!\n");
}
```

# Thread - Approfondimenti

- **I concetti affrontati sono da considerarsi base per quanto riguarda i thread**

- **Per approfondimenti si consiglia il seguente libro, al capitolo 4:**

  Mitchell, M., Oldham, J., and Samuel, A., Advanced Linux Programming. Boston, MA: New Riders, 2001.