SAPIENZA
UNIVERSITÀ DI ROMA

Programmazione Sistemi Multicore

Ripasso del linguaggio di programmazione C

Christian Cardia & Gabriele Saturni

cardia@di.uniroma1.it – saturni@di.uniroma1.it
Dip. Informatica, st. 317 – via Salaria 113, Roma

Multithreading in C - introduzione

- Il codice è spesso scritto in modo serializzato (o sequenziale). Cosa si intende con il termine serializzato? Ignorando il parallelismo a livello di istruzioni, il codice viene eseguito in sequenza, uno dopo l'altro in modo monolitico.
- Esistono programmi dei quali è possibile incrementare le prestazioni se esiste la possibilità di svolgere più operazioni contemporaneamente.
- Con la crescente popolarità delle macchine con multiprocessing simmetrico (dovuto in gran parte all'aumento dei processori multicore), la programmazione con thread è una competenza preziosa che vale la pena di imparare.

Multithreading in C - introduzione

- Per prima cosa ci tufferemo nel mondo dei thread dando un po' di background. Esamineremo le primitive di sincronizzazione dei thread e quindi verrà presentato un tutorial su come utilizzare i pthread POSIX.

I thread definizione

SAPIENZA
UNIVERSITÀ DI ROMA

- Un programma diventa un processo quando viene caricato nella memoria e inizia l'esecuzione. Un processo può essere eseguito da un processore o da una serie di processori.
- La descrizione del processo in memoria contiene informazioni vitali, come il program counter, che tiene traccia di quale istruzione è attualmente in esecuzione, registri, archivi variabili, handle di file, segnali e così via.
- Un thread è una sequenza di tali istruzioni all'interno di un processo che possono essere eseguite indipendentemente.

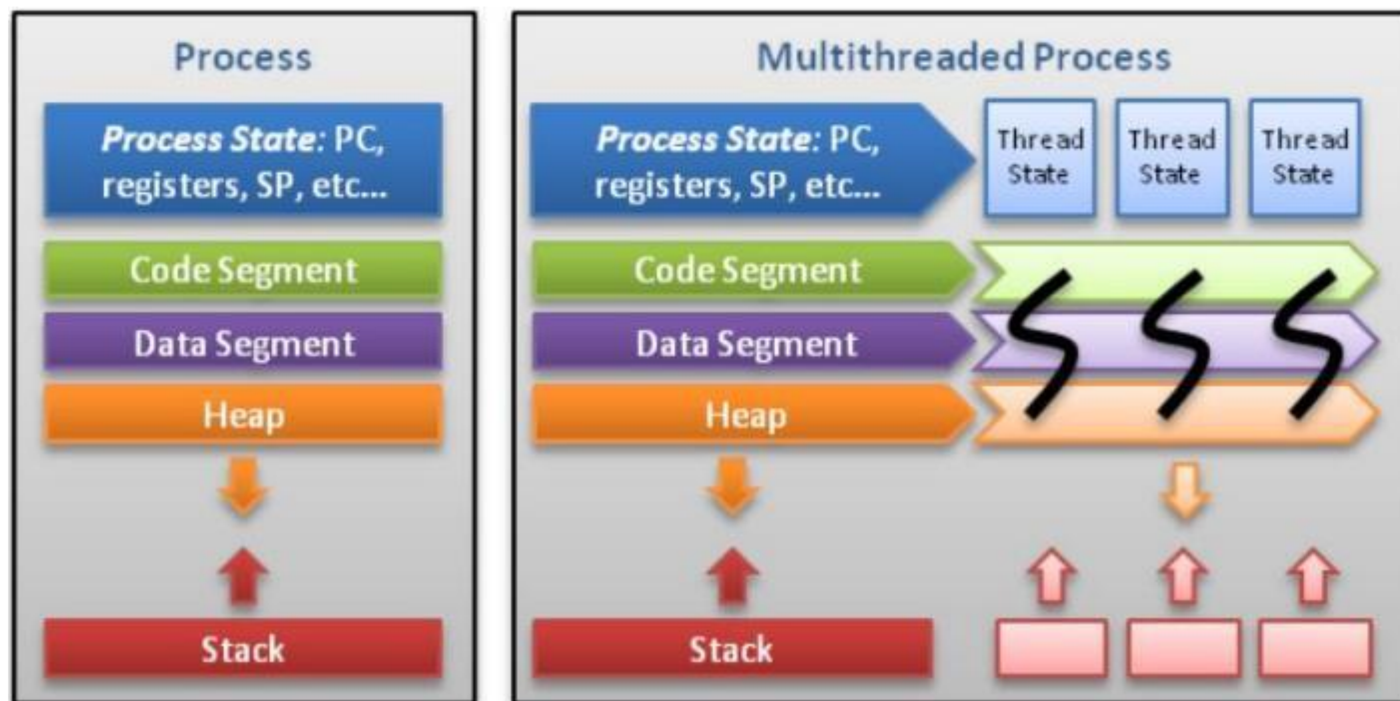
SAPIENZA
UNIVERSITÀ DI ROMA

I thread definizione

- Ogni Thread può usare tutte le variabili globali del processo, e condividere la tabella dei descrittori di file del processo.
- Ciascun thread in più potrà avere anche dei propri dati, e sicuramente avrà un proprio Stack, un proprio Program Counter ed un proprio Stato dei Registri. Dati globali ed entità del Thread (Dati,Stack,Codice,Program Counter, Stato dei Registi) rappresentano lo stato di esecuzione del singolo thread.

I thread definizione

- La figura mostra concettualmente che i thread si trovano nello stesso spazio degli indirizzi di processo, quindi molte delle informazioni presenti nella descrizione della memoria del processo possono essere condivise tra i thread.



I thread definizione

SAPIENZA
UNIVERSITÀ DI ROMA

- Alcune informazioni non possono essere replicate, come lo stack (puntatore dello stack in un'area di memoria diversa per ogni thread), registri e dati specifici del thread.
  - Queste informazioni sono necessarie per consentire la pianificazione dei thread.
- Per eseguire programmi multithread è necessario il supporto esplicito del sistema operativo. I sistemi operativi possono utilizzare meccanismi diversi per implementare il supporto del multithreading.

Legge di Adam

SAPIENZA
UNIVERSITÀ DI ROMA

- I thread possono essere utili per rendere l'esecuzione più veloce… nel giusto contesto!
- Non possiamo usare i thread per fare in contemporanea alcune parti di codice che, per sua natura, è sequenziale.
- La legge di Adam afferma che l'accelerazione di un programma dovuta alla parallelizzazione non può essere maggiore dell'inverso della parte del programma che è immutabilmente sequenziale.
  - Ad esempio, se il 50% del tuo programma non è parallelizzabile, puoi aspettarti dei miglioramenti in termini di velocità solo per la metà del codice parallelizzabile.

I thread - vantaggi

SAPIENZA
UNIVERSITÀ DI ROMA

- Visibilità dei dati globali: condivisione di oggetti semplificata.
- Più flussi di esecuzione.
- Gestione semplice di eventi asincroni (I/O per esempio)
- Comunicazioni veloci. Tutti i thread di un processo condividono lo stesso spazio di indirizzamento, quindi le comunicazioni tra thread sono più semplici delle comunicazioni tra processi.
- Context switch veloce. Nel passaggio da un thread ad un altro di uno stesso processo viene mantenuto buona parte dell'ambiente.

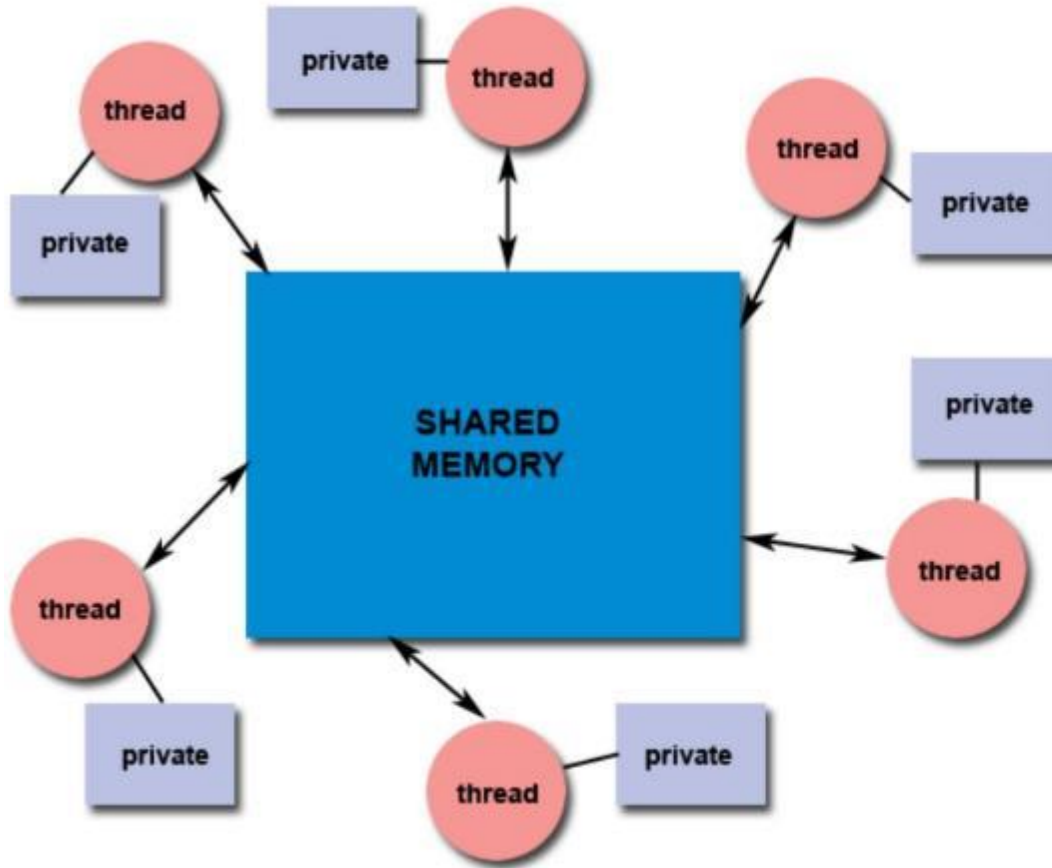
I thread - svantaggi

- Concorrenza invece di parallelismo: gestire la mutua esclusione.
- Routine di libreria devono essere rientranti (thread safe call): i thread di un programma usano il s.o. mediante system call che usano dati e tabelle di sistema dedicate al processo. Le syscall devono essere costruite in modo da poter essere utilizzate da più thread contemporaneamente evitando conflitti.

Protezione delle risorse in comune

- Abbiamo detto che i thread sono utili per svolgere più task in contemporanea. Questo generalmente crea dei problemi per la gestione delle risorse in comune.
- Infatti, i thread possono operare su dati distinti, ma spesso i thread potrebbero dover lavorare sugli stessi dati.
- Non è sicuro consentire l'accesso simultaneo a tali dati o risorse senza alcun meccanismo che definisce un protocollo per l'accesso sicuro! I thread devono essere esplicitamente istruiti a bloccare quando altri thread potrebbero potenzialmente accedere alle stesse risorse.

Protezione delle risorse in comune



Mutua esclusione

SAPIENZA
UNIVERSITÀ DI ROMA

- La mutua esclusione è il metodo con il quale viene gestito l'accesso alle risorse condivise.
- Una gestione corretta evita che due o più thread modifichino una variabile che deve essere usata da un altro thread.
- Necessità di evitare «letture sporche» → quando un valore in fase di aggiornamento viene letto contemporaneamente da un altro thread.

Mutua esclusione (mutex)

SAPIENZA
UNIVERSITÀ DI ROMA

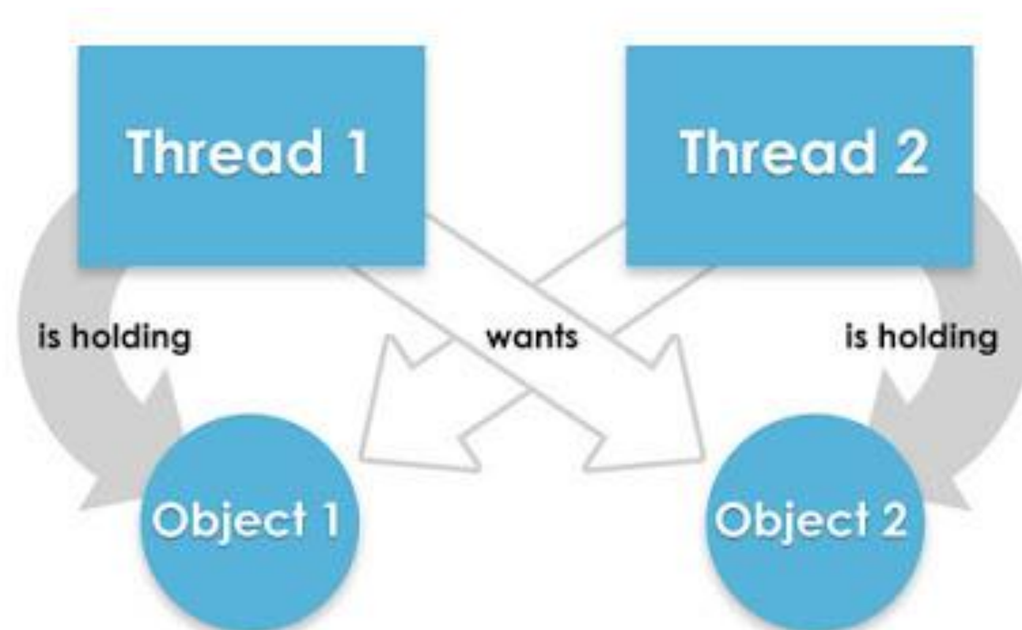
- Permette al programmatore di definire un protocollo per l'accesso controllato e serializzato alle risorse.
- Logicamente possiamo vedere questa tecnica come un lucchetto che viene attaccato momentaneamente ad una risorsa per impedirne l'accesso simultaneo.
- Per accedere alla risorsa il thread deve prima ottenere il lucchetto e poi usarla senza preoccuparsi che altri thread possano accedervi. Quando il processo ha terminato con quella risorsa la rilascia e, con essa, anche il lucchetto.
- In tal modo si ottiene un accesso serializzato alla risorsa.
- Una criticità di tale protocollo consiste nella gestione della **sezione critica** → codice che gestisce il "lucchetto".
- Necessità di minimizzare tale tempo per massimizzare le prestazioni.

Possibili problematiche

MakeAGIF.com

Possibili problematiche

- **DEADLOCK**
- Due o più thread si bloccano a vicenda



Gestire i deadlock

SAPIENZA
UNIVERSITÀ DI ROMA

- Esistono diverse macro strategie per gestire potenziali deadlock
- **Avoidance →** possibile solo se il sistema è capace di mantenere delle informazioni sulle risorse disponibili nel sistema e sulle risorse che ogni thread può potenzialmente richiedere → difficile perché non è sempre possibile chi richiederà una risorsa con anticipo
- **Detection →** riconoscere i deadlock e gestirli →chiudere i thread in stallo fino a quando non si risolve
- **Ignorarli →** Se i deadlock si verificano raramente quando accadono si potrebbe semplicemente lasciarli accadere e riavviare se necessario, piuttosto che incorrere in costanti costi generali e di prestazioni del sistema associati alla prevenzione o al rilevamento dei deadlock. Questo è l'approccio adottato da Windows e UNIX.

Primitive di sincronizzazione

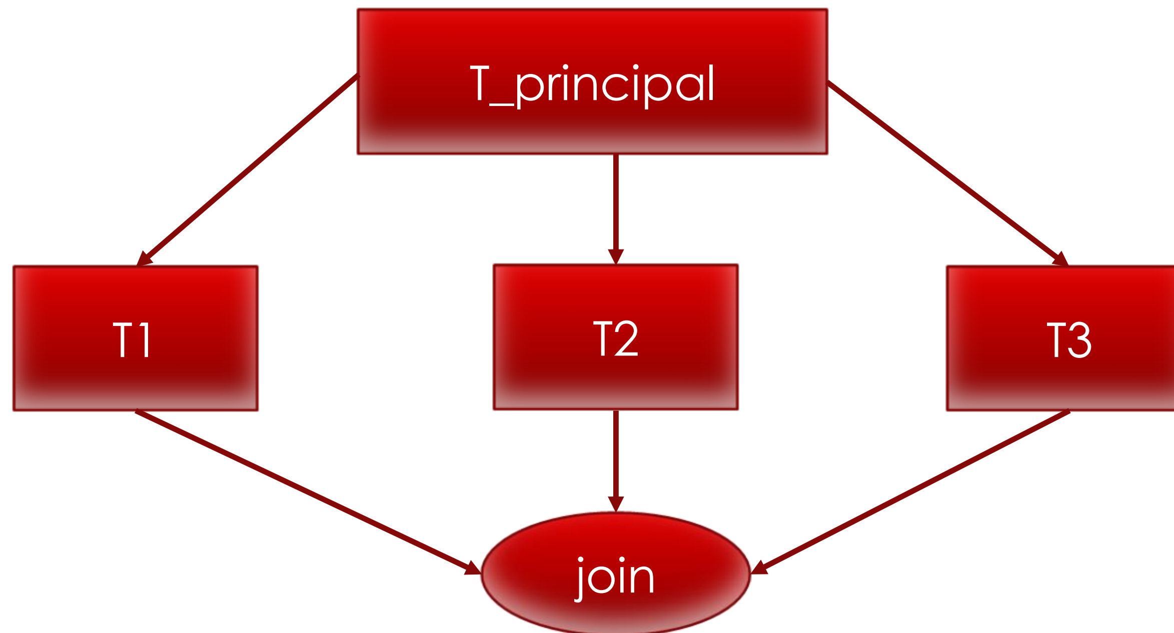
SAPIENZA
UNIVERSITÀ DI ROMA

- La mutua esclusione è modo per sincronizzare l'accesso alle risorse condivise ma non è l'unico.

- Esistono altri meccanismi disponibili non solo per coordinare l'accesso alle risorse, ma anche per sincronizzare i thread.

Primitive di sincronizzazione - JOIN

- Il thread principale genera thread «secondari» per gestire attività parallele per unirli successivamente al thread principale dopo aver completato le rispettive attività.



primitive di sincronizzazione – variabili condizionate

- Le variabili di condizione consentono ai thread di sincronizzarsi con un valore di una risorsa condivisa.
- In genere, le variabili di condizione vengono utilizzate come sistema di notifica tra i thread.
- Ad esempio, un contatore che una volta raggiunto un certo conteggio viene usato attivare un thread. Il thread (o più) si attiverà quando il contatore raggiunge il valore della variabile condizionale.
- Tale meccanismo risulta utile per la comunicazione e sincronizzazione tra i vari thread.

primitive di sincronizzazione – variabili condizionate

SAPIENZA
UNIVERSITÀ DI ROMA

4 total threads

green = active
red = sleeping
purple = woken

```
lock(mutex_x);
++x;
if (x == condition to wake) {
  unlock(mutex_x);
  signal(cond_var);
} else {
  unlock(mutex_x);
}
```

signal to wake

Sleeping threads waiting for signal to wake

a thread woken

```
lock(mutex_x);
while (x != condition to wake) {
  sleep(cond_var, mutex_x);
}
unlock(mutex_x);
/* woken - do useful things */
```

primitive di sincronizzazione - barriere

SAPIENZA
UNIVERSITÀ DI ROMA

- Le barriere sono un metodo per sincronizzare un insieme di thread in un determinato momento.
- Tutti i thread partecipanti devono attendere il momento nel quale ogni thread chiama una determinata funzione (denominata come funzione barriera).
- Questo, in sostanza, blocca tutti i thread che fino a quando il thread più lento non raggiunge un determinato punto d'esecuzione.

primitive di sincronizzazione- spinlocks

- Si basa sul concetto di attesa attiva.
- Un thread controlla periodicamente se può ottenere un lucchetto per bloccare per una risorsa.
- Il vantaggio è che il processo rimane sempre attivo e non entra in modalità sleep-wait. → incremento in termini di velocità
- Lo svataggio principale è che tale approccio è molto oneroso.

primitive di sincronizzazione - semafori

SAPIENZA
UNIVERSITÀ DI ROMA

primitive di sincronizzazione - semafori

- I semafori sono un disponibili in due versioni: binaria e conteggio.
- I semafori binari si comportano in modo molto simile ai semplici mutex → usati per consentire o negare un accesso a una risorsa.
- I semafori a conteggio → permettono un determinato numero di accessi alla risorsa.
  - Il conteggio dei semafori può essere inizializzato su qualsiasi valore arbitrario che dovrebbe dipendere da quante risorse sono disponibili per quel particolare dato condiviso. Molti thread possono ottenere il blocco contemporaneamente fino al raggiungimento del limite.
- I semafori sono i più comuni nella programmazione multi-thread (ovvero vengono generalmente utilizzati come primitiva di sincronizzazione tra processi).

POSIX PTHREAD

SAPIENZA
UNIVERSITÀ DI ROMA

- Ora che abbiamo una buona base di concetti sui thread, parliamo di una particolare implementazione di threading la **pthreads POSIX**.

- La libreria pthread può essere trovata su quasi tutti i moderni sistemi operativi compatibili con POSIX (e anche sotto Windows, vedi pthreads-win32).

POSIX PTHREAD

SAPIENZA
UNIVERSITÀ DI ROMA

- Prima di iniziare, è necessario eseguire alcuni passaggi

1. Aggiungere la libreria pthrread → #include <pthread.h> al codice sorgente.

2. Se usate gcc, potete semplicemente specificare -pthread che imposterà tutte le definizioni e le librerie time-link appropriate. Su altri compilatori, potrebbe essere necessario definire _REENTRANT e collegarsi a -lpthread.

- Opzionale: alcuni compilatori potrebbero richiedere la definizione di _POSIX_PTHREAD_SEMANTICS per alcune chiamate di funzione come sigwait ().

POSIX PTHREAD - creazione

- Un thread è rappresentato dal tipo **pthread_t**. Per creare un thread, è disponibile la seguente funzione:

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

- **pthread_t *thread:** è un puntatore ad un identificatore di thread in cui verrà scritto l'identificatore del thread creato.
- **pthread_attr_t *attr:** attributi da applicare al thread
- **void *(*start_routine)(void *):** è il nome (indirizzo) della procedura da fare eseguire dal thread. Deve avere come unico argomento un puntatore.
- **void *arg:** argomenti da passare alla funzione start_routine

POSIX PTHREAD

SAPIENZA
UNIVERSITÀ DI ROMA

- Vediamo altre funzioni importanti:

```
1  void pthread_exit(void *value_ptr);
2  int pthread_join(pthread_t thread, void **value_ptr);
3
```

- **pthread_exit(void *value_ptr):** termina l'esecuzione del thread da cui viene chiamata, immagazzina il valore puntato da value_ptr, restituendolo ad un altro thread che attende la sua fine. Il sistema libera le risorse allocate al thread.

- **pthread_join(pthread_t thread, void **value_ptr):** sospende il thread chiamante in attesa di una terminazione corretta del thread specificato come primo argomento (pthread_t thread) con un dato facoltativo * value_ptr passato dalla chiamata del thread di terminazione a pthread_exit ()

POSIX PTHREAD

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

#define NUM_THREADS 10

/* create thread argument struct for thr_func() */
typedef struct _thread_data_t {
  int tid;
  double stuff;
} thread_data_t;

/* thread function */
void *thr_func(void *arg) {
  thread_data_t *data = (thread_data_t *)arg;

  printf("hello from thr_func, thread id: %d\n", data->tid);

  pthread_exit(NULL);
}
```

POSIX PTHREAD

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

#define NUM_THREADS 10

/* create thread argument struct for thr_func() */
typedef struct _thread_data_t {
  int tid;
  double stuff;
} thread_data_t;

/* thread function */
void *thr_func(void *arg) {
  thread_data_t *data = (thread_data_t *)arg;

  printf("hello from thr_func, thread id: %d\n", data->tid);

  pthread_exit(NULL);
}
```

Caricamento delle librerie

POSIX PTHREAD

SAPIENZA
UNIVERSITÀ DI ROMA

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

#define NUM_THREADS 10

/* create thread argument struct for thr_func() */
typedef struct _thread_data_t {
  int tid;
  double stuff;
} thread_data_t;

/* thread function */
void *thr_func(void *arg) {
  thread_data_t *data = (thread_data_t *)arg;

  printf("hello from thr_func, thread id: %d\n", data->tid);

  pthread_exit(NULL);
}
```

Definiamo un numero di thread che verranno creati. Costante con il costrutto #define

POSIX PTHREAD

SAPIENZA
UNIVERSITÀ DI ROMA

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

#define NUM_THREADS 10

/* create thread argument struct for thr_func() */
typedef struct _thread_data_t {
  int tid;
  double stuff;
} thread_data_t;

/* thread function */
void *thr_func(void *arg) {
  thread_data_t *data = (thread_data_t *)arg;

  printf("hello from thr_func, thread id: %d\n", data->tid);

  pthread_exit(NULL);
}
```

Definiamo la struct contenente i dati del thread

POSIX PTHREAD

SAPIENZA
UNIVERSITÀ DI ROMA

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

#define NUM_THREADS 10

/* create thread argument struct for thr_func() */
typedef struct _thread_data_t {
  int tid;
  double stuff;
} thread_data_t;
```

Funzione che viene eseguita dal thread

```
/* thread function */
void *thr_func(void *arg) {
  thread_data_t *data = (thread_data_t *)arg;

  printf("hello from thr_func, thread id: %d\n", data->tid);

  pthread_exit(NULL);
}
```

POSIX PTHREAD

```
int main(int argc, char **argv) {
  pthread_t thr[NUM_THREADS];
  int i, rc;
  /* create a thread_data_t argument array */
  thread_data_t thr_data[NUM_THREADS];

  /* create threads */
  for (i = 0; i < NUM_THREADS; ++i) {
    thr_data[i].tid = i;
    if ((rc = pthread_create(&thr[i], NULL, thr_func, &thr_data[i]))) {
      fprintf(stderr, "error: pthread_create, rc: %d\n", rc);
      return EXIT_FAILURE;
    }
  }
  /* block until all threads complete */
  for (i = 0; i < NUM_THREADS; ++i) {
    pthread_join(thr[i], NULL);
  }

  return EXIT_SUCCESS;
}
```

POSIX PTHREAD

SAPIENZA
UNIVERSITÀ DI ROMA

```
int main(int argc, char **argv) {
  pthread_t thr[NUM_THREADS];
  int i, rc;
  /* create a thread_data_t argument array */
  thread_data_t thr_data[NUM_THREADS];

  /* create threads */
  for (i = 0; i < NUM_THREADS; ++i) {
    thr_data[i].tid = i;
    if ((rc = pthread_create(&thr[i], NULL, thr_func, &thr_data[i]))) {
      fprintf(stderr, "error: pthread_create, rc: %d\n", rc);
      return EXIT_FAILURE;
    }
  }
  /* block until all threads complete */
  for (i = 0; i < NUM_THREADS; ++i) {
    pthread_join(thr[i], NULL);
  }

  return EXIT_SUCCESS;
}
```

Nome del thread

POSIX PTHREAD

SAPIENZA
UNIVERSITÀ DI ROMA

```
int main(int argc, char **argv) {
  pthread_t thr[NUM_THREADS];
  int i, rc;
  /* create a thread_data_t argument array */
  thread_data_t thr_data[NUM_THREADS];

  /* create threads */
  for (i = 0; i < NUM_THREADS; ++i) {
    thr_data[i].tid = i;
    if ((rc = pthread_create(&thr[i], NULL, thr_func, &thr_data[i]))) {
      fprintf(stderr, "error: pthread_create, rc: %d\n", rc);
      return EXIT_FAILURE;
    }
  }
  /* block until all threads complete */
  for (i = 0; i < NUM_THREADS; ++i) {
    pthread_join(thr[i], NULL);
  }

  return EXIT_SUCCESS;
}
```

Puntatore alla funzione che il thread deve eseguire

Valore attributo posto a null per ottenere comportamento di default

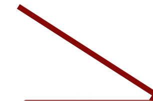
POSIX PTHREAD

```
int main(int argc, char **argv) {
  pthread_t thr[NUM_THREADS];
  int i, rc;
  /* create a thread_data_t argument array */
  thread_data_t thr_data[NUM_THREADS];

  /* create threads */
  for (i = 0; i < NUM_THREADS; ++i) {
    thr_data[i].tid = i;
    if ((rc = pthread_create(&thr[i], NULL, thr_func, &thr_data[i]))) {
      fprintf(stderr, "error: pthread_create, rc: %d\n", rc);
      return EXIT_FAILURE;
    }
  }
  /* block until all threads complete */
  for (i = 0; i < NUM_THREADS; ++i) {
    pthread_join(thr[i], NULL);
  }

  return EXIT_SUCCESS;
}
```

Argomenti passati alla funzione



ESERCIZI