# POSIX Threads

Corso di Laurea in Ingegneria Informatica

Università degli Studi di Napoli Federico II

Anno Accademico 2024/2025, Canale San Giovanni

# Riferimenti

- Dispensa “Programmazione dei POSIX Thread”
  http://www.llnl.gov/computing/tutorials/pthreads/

# PThreads

- Breve storia:
  - Diverse librerie proprietarie per la gestione dei thread implementate da diversi produttori di hardware
  - Forte eterogeneità tra le varie librerie

  - Bisogno di un interfaccia standard
  - Per i sistemi UNIX tale interfaccia fu specificata nel 1995 con lo standard IEEE POSIX 1003.1c (PThreads)

# PThreads

- I PThreads sono stati definiti come un insieme **di tipi e procedure** implementate in C
- Sono composti da un file **pthread.h** e una **libreria dinamica** (parametro **-pthread**)

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

```
$ gcc -c main.c -o main.o
$ gcc -c procedure.c -o procedure.o
$ gcc -pthread main.o procedure.o -o eseguibile
```

# Perché PThreads?

- Principalmente, per il guadagno in **prestazioni**:

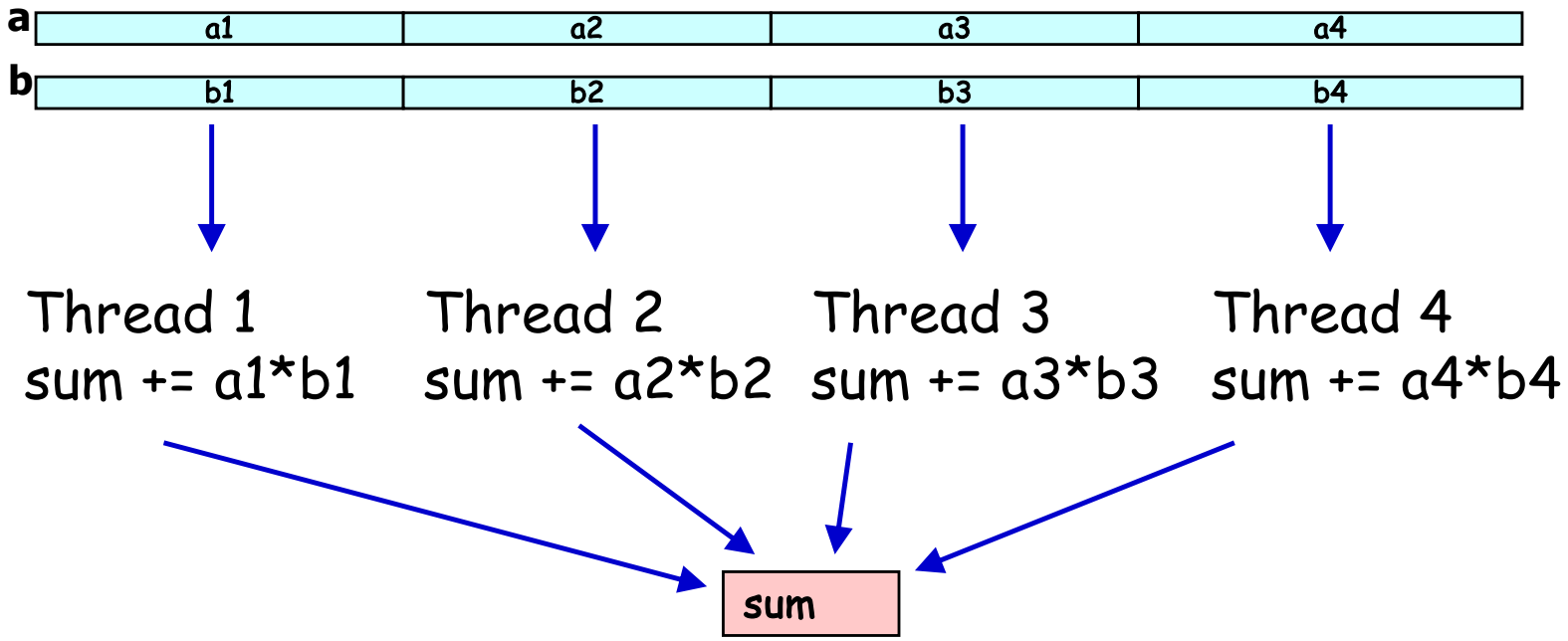
| Platform | fork() | pthread_create() |
|---|---|---|
| IBM 375 MHz POWER3 | 61.94 | 7.46 |
| IBM 1.5 GHz POWER4 | 44.08 | 1.49 |
| IBM 1.9 GHz POWER5 p5-575 | 50.66 | 1.13 |
| INTEL 2.4 GHz Xeon | 23.81 | 1.70 |
| INTEL 1.4 GHz Itanium 2 | 23.61 | 2.10 |

50000 creazioni di processi/thread;
i tempi sono riportati in secondi

- La **comunicazione inter-thread** è molto più efficiente e semplice da usare rispetto al caso inter-processo

# Esempio: prodotto scalare

- Si vuole calcolare il **prodotto scalare** di **due vettori di numeri reali**
- I vettori possono essere condivisi tra più thread
- Ogni thread calcola il prodotto scalare tra una parte dei due vettori, ed aggiorna una variabile condivisa contenente il risultato



# PThread APIs

- **Gestione dei Thread**: creazione, distruzione e join di thread
- **Gestione dei Mutex**: creazione, distruzione, lock e unlock di variabili di mutua esclusione (mutex) per la gestione di sezioni critiche
- **Gestione delle Condition Variables**: creazione, distruzione, wait e signal su variabili condition definite dal programmatore

# Creazione di un Thread

```
pthread_create(id, attr, start_routine, arg)
```

- Crea un nuovo thread e lo mette in esecuzione
  - `id` (output): tipo `pthread_t *`, è un identificatore del thread creato
  - `attr` (input): tipo `pthread_attr_t`, imposta gli attributi del thread
  - `start_routine` (input): puntatore alla funzione che verrà eseguita dal nuovo thread, da definire come:

    ```
    void * nome_funzione(void *)
    ```

    nel paramentro andrà passato "`nome_funzione`" (senza parentesi)
  - `arg` (input): è un **puntatore passato come parametro di ingresso** alla starting routine (ne sarà fatto casting a `void *`)

# Terminazione di un Thread

- Un thread può **terminare** per diversi motivi:
  - La starting routine termina la sua esecuzione
  - Il thread chiama la `pthread_exit()`
  - Il thread è cancellato da un altro thread con `pthread_cancel()`
  - L'intero processo termina

- **`pthread_exit(status)`**
  - Usata per terminare un thread esplicitamente
  - Se usata nel programma principale (che potrebbe terminare prima di tutti i thread), gli altri thread continueranno ad eseguire
  - È buona norma usarla in tutti i thread
  - **`status`** (input): indica lo stato di uscita del thread

# Un esempio

```c
#include <pthread.h>
#include <stdio.h>
#define NUM_THREADS     5

void *PrintHello(void * p) // start routine
{
   printf("\n%d: Hello World!\n", (int)p);
   pthread_exit(NULL); // terminazione thread "figlio"
}

int main (int argc, char *argv[])
{
   pthread_t threads[NUM_THREADS];

   int rc;
   for(int i=0; i<NUM_THREADS; i++){
      printf("Creating thread %d\n", i);

      rc = pthread_create(&threads[i], NULL, PrintHello, (void *)i);
      if (rc!=0){
         printf("ERROR; return code from pthread_create() is %d\n", rc);
         exit(-1);
      }
   }

   pthread_exit(NULL); // terminazione del thread "padre"
}
```

Questa funzione sarà eseguita da tanti thread "figli", deve avere "void *" sia in ingresso sia in uscita

Inseriremo gli identificatori dei thread in variabili **pthread_t**

Nome della starting routine

Valore passato a "PrintHello"

# Un esempio

nota: stiamo passando "int" invece che un puntatore (casting), andrebbe evitato

```
#include <pthread.h>
#include <stdio.h>
#define NUM_THREADS     5

void *PrintHello(void * p) // start routine
{
   printf("\n%d: Hello World!\n", (int)p);
   pthread_exit(NULL); // terminazione thread "figlio"
}

int main (int argc, char *argv[])
{
   pthread_t threads[NUM_THREADS];

   int rc;
   for(int i=0; i<NUM_THREADS; i++){
      printf("Creating thread %d\n", i);

      rc = pthread_create(&threads[i], NULL, PrintHello, (void *)i);
      if (rc!=0){
         printf("ERROR; return code from pthread_create() is %d\n", rc);
         exit(-1);
      }
   }

   pthread_exit(NULL); // terminazione del thread "padre"
}
```

# Passaggio di parametri

- La `pthread_create` può passare un **singolo argomento** di tipo `void *`
- Per passare **più di un argomento** al thread, occorre definire una **struct**:

```
struct dati {
    int dato1;
    char dato2;
};
```

- Nel thread padre, occorre **allocare sullo heap (malloc)** una istanza della struct, e **passarne il puntatore** al thread figlio:

```
struct dati *d = (struct dati *)malloc(sizeof(struct dati));
d->dato1=10;
d->dato2='c';
pthread_create(&id, NULL, start_func, (void *) d);
```

il puntatore "struct dati *" viene convertito in "puntatore void *"
(è un tipo di puntatore generico, non dereferenziabile)

# Passaggio di parametri

- Per usare correttamente la struttura dati nel thread figlio, occorre fare il **casting inverso** del puntatore "**void ***":

```
void * start_func( void * p ) {
    struct dati* dati = (struct dati *) p;
    ...
    printf("dato1=%d, dato2=%c", dati->dato1, dati->dato2);
}
```

- Esempio di uso **non corretto** del puntatore:

```
void * start_func( void * p ) {

    ...
    // NON COMPILA! Il puntatore "p" (void*) non ha i campi "dato1"/"dato2"
    printf("dato1=%d, dato2=%c", p->dato1, p->dato2);
}
```

# Passaggio di parametri

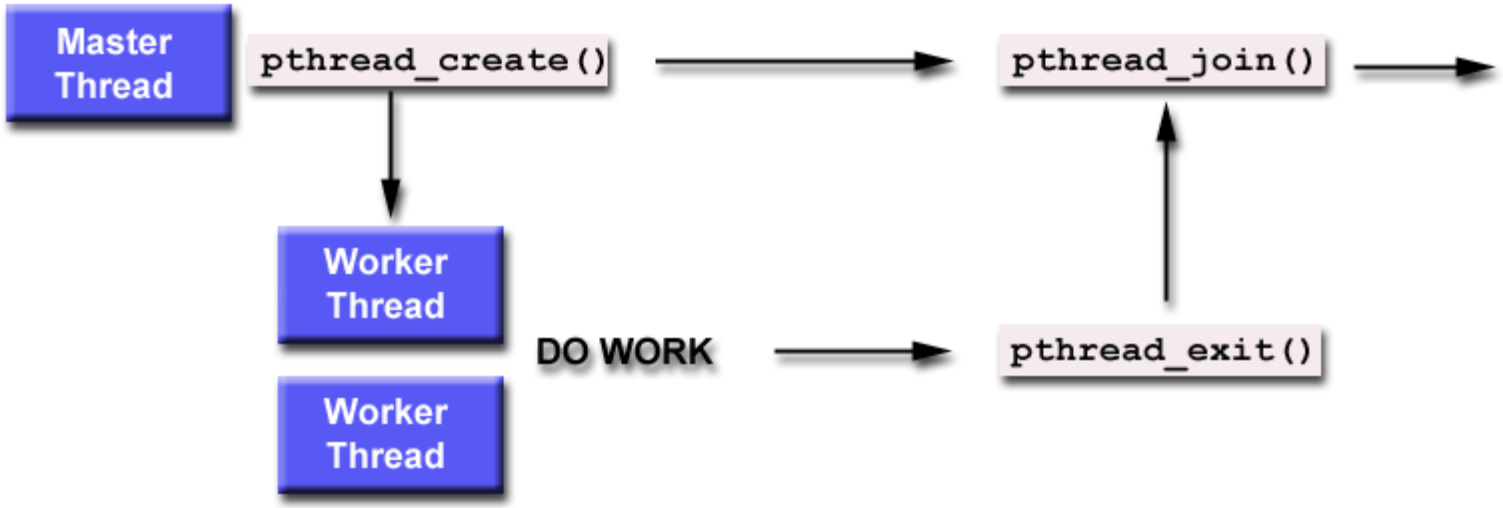
- È importante **utilizzare l'area heap** per condividere dati fra thread
- È invece **scorretto utilizzare l'area stack** per condividere dati (è dedicato alle variabili automatiche, private del thread)
- Esempio di passaggio **non corretto** tramite area stack:

```
// Thread padre
struct dati d; // var. automatica (stack)
d.dato1 = 3;
d.dato2 = 'c';
pthread_create(..., &d);  // non corretto
```

```
// Thread figlio
void * funzione(void *p) {

        ...usa p...     // non corretto, il figlio accede allo
                        // stack del thread padre!
```

# Join




- L'operazione di JOIN permette di sincronizzare un thread padre con uno o più thread figli
- La chiamata **`pthread_join(threadId, status)`** blocca il chiamante finché il thread `threadId` specificato non termina

# Joinable Threads

- `pthread_join()` pone il thread padre in attesa della terminazione di un thread figlio

```
…
pthread_create(&id, ....., start_r, (void *) data);
…
pthread_join(id, NULL);
```

Variabile di tipo `pthread_t`
(la stessa usata in `pthread_create`)

Con **NULL**, il padre "rinuncia" a ricevere dati di uscita dal thread figlio

# Joinable Threads

- Un thread deve essere dichiarato come "joinable" affinché su di esso si possa effettuare l'operazione di join

```
pthread_attr_t attr;
pthread_attr_init(&attr);
pthread_attr_setdetachstate(&attr, PTHREAD_CREATE_JOINABLE);
…
pthread_create(&id, &attr, start_r, (void *) data);
…
pthread_join(id, NULL);
```

La variabile contiene uno o più attributi per configurare il funzionamento del thread

# Joinable Threads

- `pthread_join()` consente di ricavare lo stato di uscita del thread (`status`) passato dalla `pthread_exit()`

```
struct dati_uscita * status;




…
pthread_create(&id, ....., start_r, (void *) data);
…
pthread_join(id, (void **) &status);
```

La join raccoglie un puntatore, che il thread figlio fornisce mediante `pthread_exit(void *)`

# Restituzione di dati

- Il thread figlio può **passare dei dati in uscita al thread padre**
- È possibile usare la stessa tecnica dei parametri di ingresso, tramite una **struct** sulla **area heap**
- Non è corretto usare l'area stack del thread figlio, poiché verrà distrutta all'uscita del thread
- Esempio di passaggio di parametri di uscita:

```
struct dati_uscita {
        int risultato;
};

void* figlio(void*) {

        struct dati_uscita * status = malloc(...);
        status->risultato=...;

        pthread_exit(status);
}
```

```
// Thread padre
struct dati_uscita * status;
pthread_join(..., &status);
        // puntatore-di-puntatore

int ris = status->risultato;
```

# Creazione e distruzione di Mutex

- **`pthread_mutex_t`**
  - Una struttura dati "opaca" (il programmatore non deve conoscerne il contenuto)
  - Rappresenta un oggetto-mutex
- **`pthread_mutex_init (mutex,attr)`**
  - Crea un nuovo mutex e lo inizializza come “sbloccato” (unlocked)
  - **`mutex`** (output): puntatore di tipo `pthread_mutex_t`
  - **`attr`** (input): per impostare gli attributi del mutex (può essere NULL)
- **`pthread_mutex_destroy (mutex)`**
  - Disattiva un mutex

Gestione dei Mutex

# lock e unlock

- **`pthread_mutex_lock (mutex)`**
  - Un thread invoca la lock su un mutex per acquisire l'accesso in mutua esclusione alla sezione critica relativa al mutex
  - Se il mutex è già acquisito da un altro thread, il chiamante si blocca in attesa di un unlock
- **`pthread_mutex_unlock (mutex)`**
  - Un thread invoca la unlock su un mutex per rilasciare la sezione critica, e per consentire quindi l'accesso ad un altro thread precedentemente bloccato.
- **`pthread_mutex_trylock (mutex)`**
  - Analoga alla lock, ma non bloccante. Se il mutex è già acquisito, ritorna immediatamente, con un codice di errore EBUSY.

# Esempio: Contatore condiviso

```c
#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>
#define NUM_THREADS     2

struct counter {
    pthread_mutex_t mutex;
    int valore;
};

void *Counter(void * x);  // incrementa il valore 100000 volte

int main (int argc, char *argv[]) {
   pthread_t threads[NUM_THREADS];

   struct counter * p = malloc(sizeof(struct counter));

   p->valore = 0;
   pthread_mutex_init(&p->mutex, NULL);

   for(int i=0; i<NUM_THREADS; i++)
      pthread_create(&threads[i], NULL, Counter, (void *)p);

   for(int i=0; i<NUM_THREADS; i++)
     pthread_join(threads[i], NULL);

    printf("Valore del contatore = %d\n", p->valore);
}
```

```c
void *Counter(void * x) {
   struct counter * p = x;
   for(int i=0; i<100000; i++) {
       pthread_mutex_lock(&p->mutex);
       p->valore++;
       pthread_mutex_unlock(&p->mutex);
   }
   pthread_exit(NULL);
}
```

# Condition Variables (CV)

- La cooperazione tra thread avviene mediante *condition variable* (CV)
- Vanno sempre usate in abbinamento con un mutex, realizzando un costrutto Monitor di tipo *signal and continue*

Gestione delle Condition Variables

# Creazione e Distruzione

- **`pthread_cond_t`**
  - Una struttura dati "opaca" (il programmatore non deve conoscerne il contenuto)
  - Rappresenta un oggetto-varcondition
- **`pthread_cond_init (condition,attr)`**
  - Inizializza la var. condition per l'uso
  - **`condition`** (output): puntatore di tipo `pthread_cond_t`
  - **`attr`** (input): per settare gli attributi della CV (può essere NULL)
- **`pthread_cond_destroy (condition)`**
  - Disattiva la CV che non serve più

Gestione delle Condition Variables

# Wait e Signal

- **`pthread_cond_wait (condition,mutex)`**
  - Blocca il thread chiamante finché non si invoca la cond_signal
  - La primitiva richiede in ingresso **anche il mutex del monitor**
  - Il mutex viene automaticamente rilasciato al momento della chiamata, e riacquisito alla riattivazione del thread
- **`pthread_cond_signal (condition)`**
  - Serve a risvegliare un thread precedentemente bloccato sulla CV
  - Semantica signal-and-continue

# Wait e Signal

- **`pthread_cond_broadcast (cond,mutex)`**
  - Riattiva **tutti i thread bloccati** su una CV (**signal_all**)
  - I thread accedono comunque **uno alla volta** al monitor

# Monitor con PThreads

- In PThreads, un monitor può essere ottenuto combinando **un mutex** e una o più **variabili condition**
- Tipicamente, inseriremo tutto in una **struct**
- Allocata nello **heap** e condivisa fra i thread

```
struct MyMonitor {
    ... // es., un buffer
    ...
    pthread_mutex_t mutex;
    pthread_cond_t cv1;
    pthread_cond_t cv2;
};
```

# Esempio di sincronizzazione

```
void metodo1(struct MyMonitor * p) {
    pthread_mutex_lock(&p->mutex);      // entra nel monitor
    ...
    while( !condizione ) {                          // il processo ha acquisito l'accesso
        pthread_cond_wait(&p->condvar, &p->mutex);  // al monitor, ed attende che la
    }                                               // condizione diventi vera prima di
    ...                                             // continuare

    pthread_mutex_unlock(&p->mutex);    // esce dal monitor
}
```

```
void metodo2(struct MyMonitor * p) {
    pthread_mutex_lock(&p->mutex);      // entra nel monitor
    ...
    // supponendo che la condizione diventi vera in questo punto del programma...

    pthread_cond_signal(&p->condvar);   // riattiva eventuali thread in attesa, ma
                                        // continua ad avere il possesso del monitor
    ...
    pthread_mutex_unlock(&p->mutex);    // esce dal monitor
 }
```

# Esempio di sincronizzazione

```
void metodo1(struct MyMonitor * p) {
    pthread_mutex_lock(&p->mutex);      // entra nel monitor
    ...
    while( !condizione ) {                          // il processo ha acquisito l'accesso
        pthread_cond_wait(&p->condvar, &p->mutex);  // al monitor, ed attende che la
    }                                               // condizione diventi vera prima di
    ...                                             // continuare

    pthread_mutex_unlock(&p->mutex);
}
```

La semantica del monitor in PThreads è (sempre!) di tipo **signal-and-continue**

```
void metodo2(struct MyMonitor * p)
    pthread_mutex_lock(&p->mutex);
    ...
    // supponendo che la condizione diventi vera in questo punto del programma...

    pthread_cond_signal(&p->condvar);   // riattiva eventuali thread in attesa, ma
                                        // continua ad avere il possesso del monitor
    ...
    pthread_mutex_unlock(&p->mutex);    // esce dal monitor
 }
```

# Esempio di sincronizzazione

```
void metodo1(struct MyMonitor * p) {
    pthread_mutex_lock(&p->mutex);      // entra nel monitor
    ...
    while( !condizione ) {                          // il processo ha acquisito l'accesso
        pthread_cond_wait(&p->condvar, &p->mutex);  // al monitor, ed attende che la
    }                                               // condizione diventi vera prima di
    ...                                             // continuare

    pthread_mutex_unlock(&p->m
}
```

```
void metodo2(struct MyMonitor
    pthread_mutex_lock(&p->mut
    ...
    // supponendo che la condi

    pthread_cond_signal(&p->co

    ...
    pthread_mutex_unlock(&p->mutex);    // esce dal monitor
 }
```

La cond_wait prende in ingresso il mutex del monitor, che sarà

- **rilasciato automaticamente** alla sospensione del thread
- **riacquisito automaticamente** (con eventuale attesa) alla riattivazione

# Monitor con PThreads

| | Multi-processo | Multi-thread |
|---|---|---|
| **Creazione di un flusso** | `fork()` | `pthread_create()` |
| **Condivisione di dati** | `shmget() + shmat()` | `malloc(), dati globali` |
| **Attesa dei figli** | `wait()` | `pthread_join()` |
| **Ingresso monitor** | `enter_monitor()` | `pthread_mutex_lock()` |
| **Sospensione su var. condition** | `wait_condition()` | `pthread_cond_wait()` |
| **Attivazione su var. condition** | `signal_condition()` | `pthread_cond_signal()` |
| **Uscita monitor** | `leave_monitor()` | `pthread_mutex_unlock()` |