

# Introduzione all'utilizzo di Pthreads

**Nicola Drago**  
**Nicola Dall'Ora**  
**Anno 2023-24**

# Programma

- Introduzione
- Libreria thread

# Programma

## Fonti:

- POSIX Thread Programming,  
[http://www.lnl.gov/computing/tutorials/workshops/workshop/pt  
hreads/MAIN.html](http://www.lnl.gov/computing/tutorials/workshops/workshop/pt<br/>hreads/MAIN.html)
- "Pthreads Programming".  
B. Nichols et al.  
O'Reilly and Associates.
- "Threads Primer".  
B. Lewis and D. Berg.  
Prentice Hall
- "Programming With POSIX Threads".  
D. Butenhof.  
Addison Wesley

# Introduzione: Thread vs. processi

- ✓ Processo = unità di possesso di risorse
- ✓ Creato dal S.O.

## **Possiede:**

- ✓ Process ID, process group ID, user ID, and group ID
- ✓ Ambiente
- ✓ Directory di lavoro
- ✓ Text
- ✓ Registri
- ✓ Stack
- ✓ Heap
- ✓ Descrittori di file
- ✓ Signal handler
- ✓ Meccanismi di IPC (code di messaggi, pipe, semafori, memoria condivisa)

# Introduzione: Thread vs. processi

- ✓ Thread = unità di esecuzione
- ✓ Esiste all'interno di un processo

## **Possiede:**

- ✓ Stack pointer
- ✓ Registri
- ✓ Proprietà di scheduling (algoritmo, priorità)
- ✓ Set di segnali pendenti o bloccati
- ✓ Dati specifici ad un thread

# Thread

- ✓ Thread multipli eseguono all'interno dello stesso spazio di indirizzamento:

Conseguenze:

- ✓ Modifiche effettuate da un thread ad una risorsa condivisa sono visibili da tutti gli altri thread
  - ✓ Possibile lettura e scrittura sulla stessa locazione di memoria
  - ✓ E' necessaria esplicita sincronizzazione da parte del programmatore
- ✓ Thread sono paritetici (no gerarchia)

# Perchè thread?

- ✓ Motivo essenziale: prestazioni!
- ✓ Costo rispetto alla creazione di un processo molto inferiore
- ✓ Costo di IPC molto inferiore
- ✓ Aumento del parallelismo nelle applicazioni

# Quali thread?

- ✓ Numerose implementazioni da parte dei produttori di HW/SW
- ✓ Implementazione standard: POSIX thread (pthread)
  - ✓ Definiti come API in C
  - ✓ Implementati tramite:
    - ✓ `pthread.h` header/include file
    - ✓ `libpthread.a`
- ✓ In Linux:
  - ✓ `gcc <file.c> -lpthread`
  - ✓ `<file.c>` deve includere `pthread.h`



# Scrittura di programmi multithreaded

Per sfruttare i vantaggi dei thread, un programma deve poter essere organizzato in una serie di task indipendenti che possono essere eseguiti in modo concorrente

## Esempi:

- ✓ procedure che possono essere sovrapposte nel tempo o eseguite in un qualunque ordine sono candidate per essere eseguite in due thread distinti
- ✓ Procedure che si bloccano per attese potenzialmente lunghe
- ✓ Procedure che devono rispondere ad eventi asincroni
- ✓ Procedure che sono più/meno importanti di altre

# Scrittura di programmi multithreaded

## Modelli tipici di programmi multithreaded:

- Manager/worker: un thread manager assegna operazioni ad altri thread (worker)  
Tipicamente il manager gestisce gli input e distribuisce il resto del lavoro
- Pipeline: un task viene spezzato in una serie di sottooperazioni (implementate da thread distinti) da eseguire in serie e in modo concorrente
- Peer: simile al modello manager/worker model
- Dopo che il thread principale crea gli altri, partecipa al lavoro

# Pthread API

Tre categorie di funzioni

- Gestione thread
- Sincronizzazione (mutex) tra thread
- Comunicazione tra thread

<b>Routine prefix</b>	<b>Functional Group</b>
<b>pthread_</b>	Thread miscellaneous subroutines
<b>pthread_attr</b>	Thread attributes objects
<b>pthread_mutex</b>	Mutexes
<b>pthread_mutexattr</b>	Mutexes attributes objects
<b>pthread_cond</b>	Condition variables
<b>pthread_condattr</b>	Condition attributes objects
<b>pthread_key</b>	Thread-specific data key

# Creazione di thread

**int pthread\_create(thread, attr, start\_routine, arg)**

- fornisce un thread ID (TID) nella variabile thread
  - ✓ Passaggio per indirizzo
  - ✓ Necessario controllo sul valore corretto del valore
- attr: usato per settare attributi del thread:
  - ✓ NULL= valori di default
  - ✓ Per altre modalità vedi dopo
- start\_routine: la funzione C che verrà eseguita dal thread
  - ✓ Ha un unico argomento (coincide con il parametro arg)
  - ✓ Se necessari più parametri
    - ✓ Costruire una struct con i parametri desiderati
    - ✓ Passare un puntatore alla struct (cast a (void\*))

# Creazione di thread

**int pthread\_create(thread, attr, start\_routine, arg)**

- arg: un singolo argomento da passare a start\_routine
  - ✓ Di tipo (void\*)
- Ritorna un codice di errore
  - ✓ 0 = ok
  - ✓ Diverso da 0 = errore

# Terminazione di thread

## **int pthread\_exit (status)**

- Chiude – termina un thread
- status: valore passato come argomento (v. dopo)
  - ✓ Tipicamente = NULL
- **NON** chiude eventuali file aperti

## **Modi alternativi per terminare un thread**

- exit() sul processo
- Thread ritorna da thread corrispondente al main()

# Terminazione di thread

## NOTE:

- Se il `main()` termina prima dei thread che ha creato, ed esce con `pthread_exit()`, le altre threads continueranno l'esecuzione. In caso contrario (no `pthread_exit()`) i thread vengono terminati insieme al processo
- `pthread_exit()` non chiude i file:  
File aperti dentro i thread rimarranno aperti dopo la terminazione dei thread

# Esempio

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

void *PrintHello(void *threadid)
{
    printf("\n%d: Hello World!\n", threadid);
    pthread_exit(NULL);
}
```



# Esempio

```
int main()
{
    pthread_t threads[NUM_THREADS];
    int rc, t;
    for(t=0;t<NUM_THREADS;t++){
        printf("Creating thread %d\n", t);
        rc = pthread_create(&threads[t], NULL, PrintHello,
            (void *)t);
        if (rc){
            printf("ERROR; return code from pthread_create()
is %d\n", rc);
            exit(-1);
        }
    }
    pthread_exit(NULL);
}
```

# Esempio: parametri

Il puntatore (void \*) consente di passare più argomenti (simile a malloc)

```
struct thread_data {  
    int thread_id;  
    int sum;  
    char *message;  
};  
struct thread_data thread_buf[NUM_THREADS];
```

# Esempio: parametri

```
void *PrintHello(void *thread_arg) {
    struct thread_data *my_data;
    ...
    my_data = (struct thread_data *) thread_arg;
    taskid = my_data->thread_id;
    sum = my_data->sum;
    hello_msg = my_data->message;
    ...
}

int main() {
    ...
    thread_buf[t].thread_id = t;
    thread_buf[t].sum = sum;
    thread_buf[t].message = messages[t];
    rc =
        pthread_create(&threads[t], NULL, PrintHello, (void*)
            &thread_buf[t]);
}
```

# Identificazione di thread

## **int pthread\_self ()**

- ritorna al thread chiamante l'unico ID che il sistema assegna

## **int pthread\_equal (thread1,thread2)**

- Confronta due thread ID. Se diversi ritorna 0, altrimenti un valore diverso da 0

# “join” di thread

Il “join” di thread è uno dei modi per realizzare la sincronizzazione tra thread

## **pthread\_join (tid,status)**

- Blocca il thread chiamante fino a che il thread specificato da **tid** termina
- La terminazione del thread desiderato può essere ottenuta tramite il parametro **status** di pthread\_exit()

# Join/detach di thread

Quando un thread viene creato, il campo *attr* definisce:

- Se se ne può fare il join (joinable)
- Se NON se ne può fare il join (detached)

Funzioni:

```
int pthread_attr_init (attr)
int pthread_attr_setdetachstate (attr,detachstate)
int pthread_attr_getdetachstate (attr,detachstate)
int pthread_attr_destroy (attr)
int pthread_detach (threadid,status)
```

# Join/detach di thread

## Procedura:

1. Dichiarazione di una variabile di tipo `pthread_attr_t`
2. Inizializzazione della variabile con la funzione **`pthread_attr_init()`**
3. Settaggio dell'attributo "detached status" con la funzione **`pthread_attr_setdetachstate()`**
4. Cancellazione delle risorse usate dall'attributo con la funzione **`pthread_attr_destroy()`**

La funzione **`pthread_detach()`** permette di fare il detach esplicito di una thread anche se era stata creata come joinable!

# Esempio

```
#include <pthread.h>
#include <stdio.h>
#define NUM_THREADS      3

void *BusyWork(void *null)
{
    int i;
    double result=0.0;
    for (i=0; i<100000; i++)
    {
        result = result/2 + (double)random();
    }
    printf("Thread result = %d\n",result);
    pthread_exit((void *) 0);
}
```



# Esempio

```
int main()
{
    pthread_t  thread[NUM_THREADS] ;
    pthread_attr_t attr;                (1)
    int rc, t, status;

    /* Initialize and set thread detached attribute
    */

    pthread_attr_init(&attr);           (2)
    pthread_attr_setdetachstate(&attr,
        PTHREAD_CREATE_JOINABLE);      (3)
```

# Esempio

```
for (t=0; t<NUM_THREADS; t++)
{
    printf("Creating thread %d\n", t);
    rc = pthread_create(&thread[t], &attr, BusyWork, NULL);
    if (rc)
    {
        printf("ERROR; return code from pthread_create() is %d\n", rc);
        exit(-1);
    }
}
```

# Esempio

```
/* Free attribute and wait for the other threads */
pthread_attr_destroy(&attr);           (4)
for (t=0; t<NUM_THREADS; t++)
{
    rc = pthread_join(thread[t], (void **)&status);
    if (rc)
    {
        printf("ERROR return code from pthread_join() is %d\n", rc);
        exit(-1);
    }
    printf("Completed join with thread %d status= %d\n", t, status);
}
pthread_exit(NULL);
}
```

# thread-safe

"Thread-safe" significa che il codice può essere chiamato da più thread senza risultati distruttivi.

Non richiede che il codice venga eseguito in modo efficiente in più thread, ma solo che possa funzionare **in modo sicuro in più thread**.

La maggior parte delle funzioni esistenti può essere resa thread-safe utilizzando gli strumenti forniti da Pthreads:

- mutex
- variabili di condizione
- dati specifici del thread

# Fonti ed esempi approfonditi

- Documentazione e testi:
- [https://github.com/freebendy/ben-books/tree/master/POSIX\\_Thread](https://github.com/freebendy/ben-books/tree/master/POSIX_Thread)
- Esempi di codice (citati anche nei testi):
- [https://github.com/snikulov/prog\\_posix\\_threads](https://github.com/snikulov/prog_posix_threads)

# Mutex

Meccanismo base per l'accesso protetto ad una risorsa condivisa

Un solo thread (alla volta) può fare il lock di un mutex

Gli altri thread in competizione sono messi in attesa

Variabili mutex:

- Tipo `pthread_mutex_t`
- Devono essere inizializzate prima dell'uso
  - ✓ Staticamente all'atto della dichiarazione  
Es: `pthread_mutex_t m = PTHREAD_MUTEX_INITIALIZER;`
  - ✓ Dinamicamente con opportuna funzione

# Mutex

## Funzioni:

```
int pthread_mutex_init (mutex, attr)
int pthread_mutex_destroy (mutex)
int pthread_mutexattr_init (attr)
int pthread_mutexattr_destroy (attr)
```

# Mutex

Il parametro ***mutex*** rappresenta l'indirizzo del mutex che si inizializza/distrugge

Il parametro ***attr*** viene usato per definire delle proprietà di un mutex

- Default = NULL
- Attributi definiti per usi avanzati
- `pthread_mutexattr_init()` e `pthread_mutexattr_destroy()` servono per creare/distruggere attributi mutex

Tipica invocazione:

```
pthread_mutex_t s;  
...  
pthread_mutex_init(&s, NULL);
```



# Mutex

In realtà, gli attributi determinano il tipo di **mutex**

- Fast mutex (**default**)
- Recursive mutex
- Error checking mutex

Il tipo di mutex ne determina il comportamento rispetto alle operazioni di lock/unlock

Per creare tipi diversi da quelli di default sono previste opportune costanti di inizializzazione

## Esempi:

```
pthread_mutex_t m =  
    PTHREAD_RECURSIVE_MUTEX_INITIALIZER_NP;  
pthread_mutex_t m =  
    PTHREAD_ERRORCHECK_MUTEX_INITIALIZER_NP;  
    (Cfr. pthread_mutex_t m = PTHREAD_MUTEX_INITIALIZER)
```

# Lock/unlock di mutex

Funzioni:

- **int pthread\_mutex\_lock (mutex)**
  - ✓ Acquisisce un lock attraverso la variabile mutex  
Se il mutex è lockato da un altro thread, la chiamata si blocca fino a che il lock è disponibile (fast mutex)
  - ✓ Altri tipi permettono multiple "lock"
- **int pthread\_mutex\_trylock (mutex)**
  - ✓ Tenta di lockare un mutex.  
Se il mutex è lockato da un altro thread, ritorna immediatamente con un codice di errore "busy"
  - ✓ In Linux: codice errore EBUSY
- **int pthread\_mutex\_unlock (mutex)**
  - ✓ Sblocca il lock posseduto dal thread chiamante (fast mutex)
  - ✓ Altri tipi permettono di associare "unlock" a lock multipli