Pthread

Posix standard → estende il linguaggio C con primitive che ci permettono di esprimere la concorrenza → in C non ci sono nativamente!!!

Le dichiarazioni delle primitve si trovano in alcune librerie come `sched.h`, `pthread.h` e `semaphore.h`. Quando si compila il codice bisogna aggiungere il parametro `-lpthread`.

Librerie da importare

```
#include <stdio.h>
#include <semaphore.h>
#include <pthread.h>
#include <unistd.h>
#include <stdlib.h>
#include <sched.h> //in caso di policy e priorità
```

Thread

Corpo di un Thread

```
void *my_thread(void *arg) { ... }
```

Creazione di un Thread

```
pthread_attr_t myattr;
pthread_t t1;
pthread_attr_init(&myattr);
int err;
err = pthread_create(&t1, &myattr, body, (void *)"value");
pthread_attr_destroy(&myattr);
phtread_join(t1, NULL);
//////////////////////
int pthread_create( pthread_t *ID, //tipo che contiene l'id del thread
                                   pthread_attr_t *attr, //tipo che cont
                    void *(*body)(void *), //raramente ci interessa
                      void * arg ); //argomento da dare in pasto al thread
```

Parametri di un Thread

```
int pthread_attr_init(pthread_attr_t *attr);
int pthread_attr_destroy(pthread_attr_t *attr);
```

Terminazione di un Thread

```
void pthread_exit(void *return_value); //
```

ID di un Thread

Ogni thread ha un ID unico.

```
//Per sapere qual'è l'ID di un thread:
pthread_t pthread_self(void);
//Per sapere se due thread sono gli stessi:
int pthread_equal(  pthread_t thread1,
                    pthread_t thread2  );
```

Join di un Thread

Un thread può aspettare la terminazione di un altro thread.

```
int pthread_join(pthread_t th, void **thread_return);
// valore di ritorno del thread che termina
// tutte le risorse vengono riallocate
```

Thread Detached → thread che non deve essere joinato

```
pthread_attr_setdetachstate(&myattr, PTHREAD_CREATE_DETACHED)
pthread_detach()
```

Pthread scheduling

Due strategie di scheduling che si possono impostare negli attributi:

- **SCHED_FIFO →** tra task a stessa priorità prende il primo ad essere arrivato, lo esegue e non viene cambiato finchè non termina.
- **SCHED_RR →** il task va avanti fino a che non finisce o fino a che non consuma il suo quanto di tempo, in questo caso avviene un cambio di contesto e viene eseguito un altro task con la stessa priorità.
- **SCHED_OTHER →** Altre.

Ci sono **32 priorità** (da 0-basso a 31-alto), ognuna ha una coda dove altri processi con la stessa priorità attendono. Va prima chi ha la priorità più alta.

Priorità di un Thread

```
int pthread_attr_setschedpolicy
            (pthread_attr_t *a, int policy);
```

```
// policy -> SCHED_FIFO, SCHED_RR o SCHED_OTHER

int pthread_attr_setschedparam
            (pthread_attr_t *attr,
                        const struct sched_param *param);
// priorità di un particolare task param.sched_priority
```

Cancellazione di un Thread

```
int pthread_cancel(pthread_t thread);
// killare un thread di cui ti passo l'ID
```

- **Deferred cancellation** → quando arriva una kill request il thread non muore subito, ma muore solo quando raggiunge un **cancellation point** (`sem_wait`, `pthread_cond_wait`, `printf` e tutte le primitive di I/O primitives).
- **Asynchronous cancellation** → quando arriva una kill request il thread muore subito.

```
int pthread_setcancelstate(int state,int *oldstate);
int pthread_setcanceltype(int type, int *oldtype);
int pthread_cleanup_push(void (*routine)(void *),
                                                void *arg);
int pthread_cleanup_pop(int execute);
```

Semafori

Inizializzare un semaforo

```
int sem_init(sem_t *sem, int pshared, unsigned int value);
// pshared è 0 se il semaforo non è condiviso tra processi
```

Distruggere un semaforo

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

WAIT di un semaforo

```
int sem_wait(sem_t *sem); // bloccante
int sem_trywait(sem_t *sem); // non bloccante
```

POST di un semaforo

```
int sem_post(sem_t *sem);
// Incrementa il counter del semaforo
```

```
// Sveglia un thread in attesa
```

Ritornare il contatore del semaforo

```
int sem_getvalue(sem_t *sem,int *val);
// Ritorna il contatore del semaforo
```

Inizializzazione attributo di un mutex

```
pthread_mutexattr_t attr
int pthread_mutexattr_init(pthread_mutexattr_t *attr);
```

Distruggere un attributo di un mutex

```
int pthread_mutexattr_destroy(pthread_mutexattr_t *attr);
```

Settare il protocollo di un mutex

```
int pthread_mutexattr_setprotocol
  (pthread_mutexattr_t *attr, int protocol);
//protocol = PTHREAD_PRIO_NONE, PTHREAD_PRIO_INHERIT,
//           PTHREAD_PRIO_PROTECT
```

Settare la priorità di un mutex

```
int pthread_mutexattr_setprioceiling
  (pthread_mutexattr_t *attr, int pceiling);
```

Inizializzazione di un mutex

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

Distruzione di un mutex

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

Lock e Unlock di un mutex

```
int pthread_mutex_lock(pthread_mutex_t *m);
int pthread_mutex_trylock(pthread_mutex_t *m);
//non è bloccante in caso non riesca a trovare
```

```
//il mutex libero
int pthread_mutex_unlock(pthread_mutex_t *m);
```

Condition Variables

Inizializzazione attributi condition variable

```
pthread_condattr_t attr;
int pthread_condattr_init(pthread_condattr_t *attr);
```

Distruzione attributi condition variable

```
int pthread_condattr_destroy(pthread_condattr_t *attr);
```

Inizializzazione condition variable

```
int pthread_cond_init (pthread_cond_t *cond,
                       const pthread_condattr_t *attr)
```

Distruzione condition variable

```
int pthread_cond_destroy(pthread_cond_t *cond)
```

WAIT condition variable

```
int pthread_cond_wait(pthread_cond_t *cond,
                                      pthread_mutex_t *mutex);
```

POST e Broadcast condition variable

```
int pthread_cond_signal(pthread_cond_t *cond);
int pthread_cond_broadcast(pthread_cond_t *cond);
```

Esempi

Private semaphore solution 1

```
// Start
void f1(struct myresource_t *r){
    sem_wait(&r->mutex);
    if <condition> {
        <resource allocation to i>
        sem_post(&r->priv[i]);
```

```
// Finish
void f2(struct myresource_t *r) {
    sem_wait(&r->mutex);
    <release the resource>
    if <wake up someone> {
        int i = <process to wake up>
```

```
    }
    else {
        <record that i is suspended
    }
    sem_post(&r->mutex);
    sem_wait(&r->priv[i]);
}
```

```
        <resource allocation to i>
        <record that i is no more
         suspended>
        sem_post(&r->priv[i]);
    }
    sem_post(&r->mutex);
}
```

Private semaphore solution 2, Token passing

```
// Start
void f1(struct myresource_t *r) {
    sem_wait(&r->mutex);
    if <not condition> {
        <record that i is suspended>
        sem_post(&r->mutex);
        sem_wait(&r->priv[i]);
        <record that i has been woke
         up>
    }
    <resource allocation to i>
    sem_post(&r->mutex);
}
```

```
// End
void f2(struct myresource_t *r) {
    sem_wait(&r->mutex);
    <release the resource>
    if <wake up someone> {
        int i = <process to wake up>
        sem_post(&r->priv[i]); }
    else {
        sem_post(&r->mutex);
    }
}
```

Condition variable e mutex

```
// Start
void f1(struct myresource_t *r){
    pthread_mutex_lock(&r->mutex);
    while(!<condition>){
        <record that i is suspended>
        pthread_cond_wait(&r->s_p[i]

        <record that i has been woke
         up>
    }
    <resource allocation>
    pthread_mutex_unlock(&r->mutex);
}
```

```
// End
void f2(struct myresource_t *r) {
    pthread_mutex_lock(&r->mutex);
    <update resource counter and fla
    // Wake up one or more thread.
    pthread_cond_signal(&r->s_p);
    pthread_mutex_unlock(&r->mutex);
}
```

Join multiple thread nel main

```
<initialize manager>
pthread_attr_t myattr;
pthread_t t1[N];
pthread_attr_init(&myattr);
```

```
for(int i = 0; i < N; i++){
    // In this case i pass a index as argument.
    pthread_create(&t1[i], &myattr, <t_function>, i);
}
for(int i = 0; i < N; i++){
    void *tmp;
    pthread_join(t1[i], &tmp);
}
pthread_attr_destroy(&myattr);
```

Detached thread in main

```
<initialize manager>
pthread_attr_t myattr;
pthread_t t1;
pthread_attr_init(&myattr);
pthread_attr_setdetachstate(&myattr, PTHREAD_CREATE_DETACHED);
pthread_create(&t1, &myattr, <t_function>, (void *)"");
pthread_attr_destroy(&myattr);
```

Attesa randomica

```
void pausa(){
    struct timespec t;
    t.tv_sec = 0;
    t.tv_nsec = (rand()%10+1)*1000000;
    // Nano second.
    nanosleep(&t,NULL);
}
```