

# Gestione dei thread

---

Ing. Giovanni Nardini - University of Pisa - All rights reserved

# Standard POSIX - pthread

Tipi e interfaccia delle funzioni definiti dallo standard POSIX → **pthread**

- Utilizzo:

Nel programma →	<b>#include &lt;pthread.h&gt;</b>	
Collegamento →	<b>gcc -pthread</b> prog.c -o prog	oppure
	<b>gcc prog.c -lpthread</b> -o prog	

- Identificatori iniziano sempre con **pthread\_**
- Tre set di funzioni
  1. Gestione dei thread
  2. Mutex
  3. Variabili condition

pthread è una libreria esterna alla libreria “standard” del C, per cui deve essere linkata esplicitamente con gcc.

La libreria standard invece viene collegata automaticamente da gcc, perché assume che il programmatore la userà sicuramente.

## Identificatore del thread

- Processo: process id (pid) `pid_t`
- Thread: thread id (tid) `pthread_t`

`#include <pthread.h>`

`pthread_t pthread_self();`  
↳ Id del thread chiamante

`int pthread_equal(pthread_t tid1, pthread_t tid2)`  
↳ !=0 se identificatori non uguali; 0 altrimenti

Univoco solo all'interno del processo, non all'interno del sistema!

Implementation-specific

Mentre `pid_t` è implementato come un intero non negativo, `pthread_t` è implementato da una struttura che può essere diversa in sistemi diversi e non dovrebbe essere trattato come un numero intero.

Per lo stesso motivo, il confronto tra due thread ID non può essere fatto con l'operatore di confronto `'=='` ma necessita di una funzione apposita.

## Creazione di un thread

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

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

0 se ok;  
!=0 in caso di errore

Conterrà l'id del nuovo thread

Attributi del thread  
Può essere NULL

Argomento della funzione  
start\_routine  
NULL se non ha argomenti

Indirizzo della funzione da  
mandare in esecuzione come  
corpo del thread

- Crea un nuovo thread del processo corrente e lo rende eseguibile
  - Viene eseguito subito?

Quando si lancia un programma, viene creato un processo composto da un thread (che esegue la funzione main).

pthread\_create può essere vista come la versione per thread della fork dei processi, con la differenza che il processo figlio creato con la fork esegue lo stesso codice del processo genitore, mentre un thread creato con pthread\_create esegue una funzione specifica indicata come argomento.

Il primo argomento della funzione è un puntatore a una locazione di memoria che, al termine della pthread\_create conterrà l'id del nuovo thread.

Il terzo argomento è il nome della funzione che sarà eseguita dal thread una volta mandato in esecuzione. Il programmatore deve aver definito tale funzione nel programma in modo che abbia un argomento di tipo void\* e che restituisca un tipo void\*.


Il quarto argomento specifica l'argomento da passare alla funzione eseguita dal thread (più dettagli nelle prossime slide)

Una volta creato, il thread è pronto per essere eseguito ed è a disposizione dello scheduler del sistema operativo, che prima o poi lo manderà in esecuzione

## Terminazione di un thread

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

```
void pthread_exit(void* rval_ptr);
```



Puntatore ad area di memoria consultabile da altri thread dello stesso processo  
Tipicamente contiene il valore di ritorno  
NULL se non ritorna alcun valore

- Termina l'esecuzione del thread da cui viene invocata
- Le risorse (stack, file descriptors) allocate dal thread vengono eliminate
- Perché non si usa `exit`?

Un thread può terminare con l'istruzione `return` nel corpo della funzione eseguita dal thread stesso, oppure con `pthread_exit`.

In entrambi i casi, può essere restituito un codice di terminazione (simile alla `exit` per i processi), sotto forma di puntatore a un area di memoria (che parlando di thread è condivisa da altri thread e può essere dunque esaminata da questi ultimi)

Un thread potrebbe anche essere terminato da un altro thread dello stesso processo, usando la funzione `pthread_cancel` (vedere man)

## Creazione e terminazione di thread - esempio

```
#include <pthread.h>
...
const int NUM_THREADS = 5

void* print_hello(void* num)
{
    pthread_t tid = pthread_self();

    printf("Hello from %d \n", (int)tid);
    pthread_exit(NULL);
}

int main()
{
    pthread_t threads[NUM_THREADS];
    int ret, t;
    for (t = 0; t < NUM_THREADS; t++)
    {
        printf("Creating thread %d \n", t);
        ret = pthread_create(&threads[t], NULL, print_hello, NULL);
        if (ret != 0)
            fprintf(stderr, "error %d: cannot create thread", ret);
    }

    sleep(1);
    return 0;
}
```

Perché?

- In quale ordine vengono eseguiti i thread?
  - Ordine di creazione dei thread **non ha alcuna relazione** con l'ordine in cui vengono eseguiti
  - pthreads schedulati dal kernel in maniera arbitraria e non predicibile



### Corretta gestione dei thread ID:

- si crea una variabile pthread\_t (o un array di pthread\_t come in questo esempio)
- si passa l'indirizzo di tale variabile (o di uno degli elementi dell'array) alla pthread\_create.
- la pthread\_create scrive in quell'indirizzo il valore del thread ID.

## Passaggio di parametri – esempio errato

```
#include <pthread.h>
...
const int NUM_THREADS = 5

void* print_hello(void* num)
{
    int num_par = *((int*)num);

    printf("Hello from %d \n", num_par);

    pthread_exit(NULL);
}

int main()
{
    pthread_t threads[NUM_THREADS];
    int ret, t;
    for (t = 0; t < NUM_THREADS; t++)
    {
        printf("Creating thread %d \n", t);
        ret = pthread_create(&threads[t], NULL, print_hello, (void*)&t);
        if (ret != 0)
            fprintf(stderr, "error %d: cannot create thread", ret);
    }
    sleep(1);
    return 0;
}
```

Un pthread inizia la sua esecuzione da una funzione che ha un parametro di tipo `void*` (puntatore)



Non usare un parametro che deve essere modificato da uno dei thread (compreso il main)

Dato che al thread viene passato un puntatore come argomento, sia il thread “genitore” che il thread “figlio” hanno accesso alla **stessa** locazione di memoria ed entrambi possono modificarla (!!!)

In questo esempio si passa al thread la variabile `t`, che però viene aggiornata ad ogni iterazione del ciclo `for`. La modifica si riflette sul comportamento del thread, dato che questo accede alla stessa variabile `t`.

## Passaggio di parametri – esempio corretto

```
#include <pthread.h>
...
const int NUM_THREADS = 5

void* print_hello(void* num)
{
    int num_par = *((int*)num);

    printf("Hello from %d \n", num_par);

    pthread_exit(NULL);
}

int main()
{
    pthread_t threads[NUM_THREADS];
    int ids[NUM_THREADS];
    int ret, t;
    for (t = 0; t < NUM_THREADS; t++) {
        printf("Creating thread %d \n", t);
        ids[t] = t;
        ret = pthread_create(&threads[t], NULL, print_hello,
                             (void*)&ids[t]);

        if (ret != 0)
            fprintf(stderr, "error %d: cannot create thread", ret);
    }
    sleep(1);
    return 0;
}
```

**Soluzione: per ogni thread, creare una struttura dati «privata»**

- Ogni parametro è acceduto solo dal thread a cui è stato passato

In questo modo, `t` viene salvato dentro `ids[t]`.

Anche se alla prossima iterazione del ciclo `for` la variabile `t` viene aggiornata, `ids[t]` non cambia.



## Passaggio di due o più parametri

- Si deve usare una **struttura**

```
#include <pthread.h>
...
const int NUM_THREADS = 5

struct st {
    int par1;
    int par2;
};

void* print_nums(void* par)
{
    int id = ((struct st*)par)->par1;
    int num = ((struct st*)par)->par2;
    printf("id:%d, num:%d\n", id, num);
    pthread_exit(NULL);
}

int main()
{
    pthread_t threads[NUM_THREADS];
    struct st pars[NUM_THREADS];
    int ret, t;
    for (t = 0; t < NUM_THREADS; t++)
    {
        pars[t].par1 = t;
        pars[t].par2 = t + 100;
        printf("Creating thread %d \n", t);
        ret = pthread_create(&threads[t], NULL, print_nums,
                             (void*)&pars[t]);
        if (ret != 0)
            fprintf(stderr, "error %d: cannot create thread", ret);
    }
    sleep(1);
    return 0;
}
```

La funzione che viene eseguita dal thread deve avere un solo argomento. L'argomento può essere a sua volta l'indirizzo di una struttura, la quale può essere definita a piacimento dal programmatore. In questo esempio la struttura ha due campi interi, per cui al thread possono essere passati due parametri di tipo intero.

## Join tra thread

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

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

0 se ok;  
!=0 in caso di errore

Id del thread di cui attendere la terminazione

Puntatore ad area di memoria che contiene  
il valore di ritorno del thread specificato

- Il thread chiamante si blocca in attesa della terminazione del thread specificato
  - Quando ritorna o invoca la funzione `pthread_exit()`
- Il thread chiamante può ottenere lo stato del thread che termina
- Forma elementare di sincronizzazione tra thread

Simile al concetto di `waitpid()` per i processi.

Possiamo esaminare il codice di terminazione del thread (ovvero, l'argomento passato a `pthread_exit()` nel thread), oppure passare `NULL` se non ci interessa.

## Join tra thread – esempio

```
#include <pthread.h>
...
const int NUM_THREADS = 5

void* print_hello(void* num)
{
    int num_par = *((int*)num);
    printf("Hello %d \n", num_par);
    pthread_exit((void*)1);
}

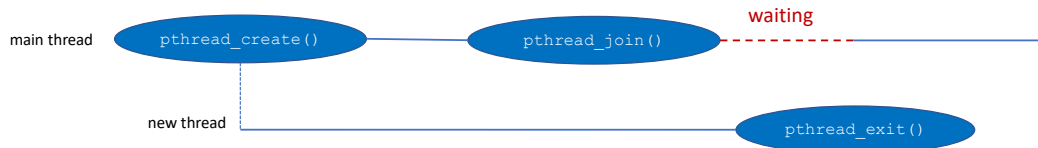
int main() {
    pthread_t threads[NUM_THREADS];
    int ids[NUM_THREADS], ret, t;
    void* status;
    for (t = 0; t<NUM_THREADS; t++) {
        printf("Creating thread %d \n", t);
        ret = pthread_create(&threads[t], NULL, print_hello,
                           (void*)&ids[t]);

        if (ret != 0)
            fprintf(stderr, "error %d: cannot create thread", ret);
    }
    for (t = 0; t<NUM_THREADS; t++) {
        ret = pthread_join(threads[t], &status);
        if (ret != 0) {
            fprintf(stderr, "error %d: cannot join", ret);
            exit(EXIT_FAILURE);
        }
        printf("Joined with thread %d, status=%d\n", t, (long)status);
    }
    return 0;
}
```

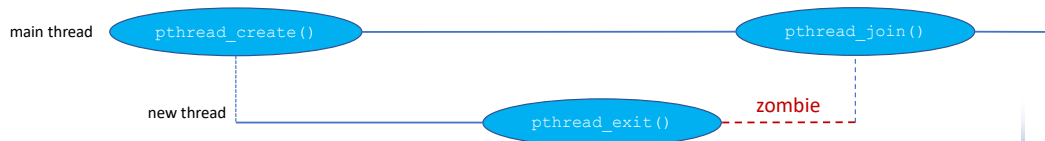
Nell'esempio il thread principale attende, a turno, la terminazione di tutti i thread precedentemente creati e ne recupera lo stato di terminazione. Notare che l'argomento di `pthread_exit` deve essere convertito esplicitamente a un tipo `void*`

## Joinable thread

- Di default, ogni nuovo thread è **joinable**
  - Altri thread possono effettuare una `pthread_join()` e attendere che il thread termini



- Quando un thread joinable termina, entra nello stato **zombie**
  - Le risorse (incluso lo stato di terminazione) non vengono rilasciate finché un altro thread esegue la `pthread_join()` su esso



## Detached thread

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

```
int pthread_detach(pthread_t tid);
```

0 se ok;  
!=0 in caso di errore

Id del thread di cui non serve attendere la terminazione

- Fa diventare il thread **detached**
- Quando un thread detached termina, le risorse vengono subito rilasciate
  - Utile quando si sa che non ci sarà bisogno di sincronizzarsi con quel thread
- Se un altro thread esegue una `pthread_join()` su un thread detached, il comportamento del sistema è indefinito

Se nessun thread esegue la `pthread_join` e il processo (eventualmente di lunga durata) crea tanti thread potrei terminare le risorse del sistema che rimangono allocate inutilmente a thread zombie.

# Concorrenza

- I thread appartenenti a uno stesso processo condividono la stessa memoria
  - Possono accedere alle stesse variabili

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

int var;

void* decr_var(void* num)
{
    if (var > 0)
        var--;
    pthread_exit(NULL);
}

int main()
{
    pthread_t t1, t2;

    var = 1;
    pthread_create(&t1, NULL, decr_var, NULL);
    pthread_create(&t2, NULL, decr_var, NULL);

    pthread_join(t1, NULL);
    pthread_join(t2, NULL);

    printf("var = %d\n", var);
    return 0;
}
```



**Problema di consistenza quando un thread può modificare una variabile**

Da questo programma ci si aspetterebbe che al termine dell'esecuzione la variabile "var" valesse 0, dato che i thread decrementano il suo valore solo quando essa è maggiore di zero.

## Concorrenza

T1:	T2:
if (var > 0)	
var--;	
	if (var > 0)
	var--;



T1:	T2:
if (var > 0)	
	if (var > 0)
var--;	
	var--;



Nel primo caso T1 viene eseguito completamente prima di T2  
Nel secondo caso T1 e T2 vengono eseguiti alternatamente.

# Mutex

- Meccanismo che permette di accedere a una sezione di codice «critica» a un thread alla volta - **Mutual exclusion**
- Semaforo binario



Non si può accedere alla sezione critica



Si può accedere alla sezione critica

```
<istruzione 1>  
<istruzione 2>  
<istruzione 3>
```

```
<istruzione critica 4>  
<istruzione critica 5>
```

```
<istruzione 6>  
<istruzione 7>
```



Il thread acquisisce il lock



Il thread rilascia il lock



Altri thread devono  
attendere prima di  
eseguire la sezione critica



## Mutex - esempio



Gestione dei thread

Ing. Giovanni Nardini - University of Pisa - All rights reserved

204

Si circonda la sezione critica con istruzioni per acquisire/rilasciare il diritto esclusivo sulla sezione stessa.

Nell'esempio, T2 va in esecuzione dopo che T1 ha eseguito l'istruzione `if (var > 0)`, ma fallisce l'acquisizione del lock per cui si blocca.

## Creazione di una variabile mutex

- Variabile di tipo `pthread_mutex_t`
- Può essere inizializzata
  - Staticamente, assegnandole un valore costante
  - Dinamicamente, invocando una funzione apposita

La libreria `pthread` fornisce un tipo di dato `pthread_mutex_t` per creare variabili per gestire la mutua esclusione.

Contiene almeno un intero che indica se il thread è libero o meno, e una coda dei thread bloccati (e degli attributi).

Inizializzare il mutex significa marcarlo come “libero”. L’inizializzazione statica si può usare solo quando il mutex è allocato nello stack, mentre se viene allocato dinamicamente nello heap (tramite una `malloc`) deve essere inizializzato dinamicamente.

Generalmente, un mutex viene allocato nello heap quando deve proteggere una risorsa allocata anch’essa nella memoria dinamica. Per esempio, abbiamo un array di `N` risorse da proteggere con un mutex ciascuna, con `N` non noto a tempo di compilazione (ad esempio, viene passato da linea di comando). L’array viene dunque allocato dinamicamente con una `malloc`, e lo stesso deve essere fatto per gli `N` mutex corrispondenti.

## Inizializzazione statica

```
#include <pthread.h>

pthread_mutex_t mymutex = PTHREAD_MUTEX_INITIALIZER;

int main()
{
    ...
}
```

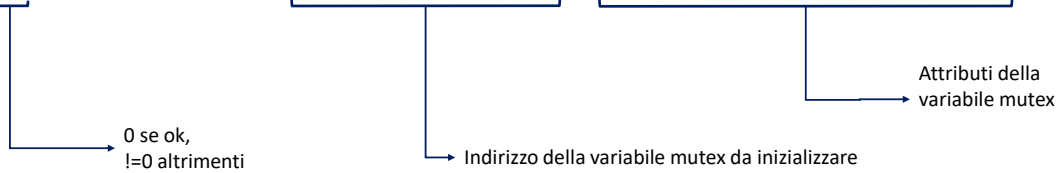
- `PTHREAD_MUTEX_INITIALIZER` è una costante definita dalla libreria `pthread.h`

Dovendo essere acceduto da diversi thread, la variabile mutex deve a sua volta essere una variabile condivisa che tipicamente si definisce come variabile globale del programma.

## Inizializzazione dinamica

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

```
int pthread_mutex_init(pthread_mutex_t *mutex, const pthread_mutexattr_t *mattr )
```



- Inizializza il mutex passato come primo argomento
- L'argomento `*mattr` può essere `NULL` per impostare attributi di default

## Inizializzazione dinamica

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

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

0 se ok,  
!=0 altrimenti

Indirizzo della variabile mutex da distruggere

- Quando il mutex non serve più, deve essere deallocato
- Si usa solo se il mutex era stato inizializzato dinamicamente
- Fallisce se mutex è occupato → restituisce `EBUSY`

## Inizializzazione di un mutex - esempio

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

int var;
pthread_mutex_t mymutex;

void* decr_var(void* num)
{
    <lock mutex>
    if (var > 0)
        var--;
    <unlock mutex>
    pthread_exit(NULL);
}

int main()
{
    pthread_t t1, t2;

    var = 1;
    pthread_mutex_init(&mymutex, NULL);

    pthread_create(&t1, NULL, decr_var, NULL);
    pthread_create(&t2, NULL, decr_var, NULL);

    pthread_join(t1, NULL);
    pthread_join(t2, NULL);

    printf("var = %d\n", var);

    pthread_mutex_destroy(&mymutex);

    return 0;
}
```

## Acquisizione di un mutex

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

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

0 se ok,  
!=0 altrimenti

Indirizzo della variabile mutex da acquisire

- Se il mutex è libero, il thread chiamante lo acquisisce
- Se il mutex è occupato, il thread chiamante si blocca
  - Il suo ID viene inserito nella coda dei thread bloccati della variabile mutex

## Rilascio di un mutex

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

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

0 se ok,  
!=0 altrimenti

Indirizzo della variabile mutex da rilasciare

- Il mutex torna disponibile per l'acquisizione da parte di eventuali altri thread
- Se ci sono thread in attesa, uno di loro acquisisce il mutex e può eseguire le istruzioni nella sezione critica
  - Eventuali altri thread in attesa, rimangono bloccati nella coda

I thread eventualmente bloccati sulla `pthread_mutex_lock` vengono **tutti** svegliati e concorrono per acquisire il mutex. Solo uno di loro ci riesce, mentre gli altri si bloccano di nuovo (e il loro ID viene re-inserito nella coda dei thread bloccati interna alla variabile mutex)



## Acquisizione e rilascio di un mutex - esempio

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

int var;
pthread_mutex_t mymutex;

void* decr_var(void* num)
{
    pthread_mutex_lock(&mymutex);
    if (var > 0)
        var--;
    pthread_mutex_unlock(&mymutex);
    pthread_exit(NULL);
}

int main()
{
    pthread_t t1, t2;

    var = 1;
    pthread_mutex_init(&mymutex, NULL);

    pthread_create(&t1, NULL, decr_var, NULL);
    pthread_create(&t2, NULL, decr_var, NULL);

    pthread_join(t1, NULL);
    pthread_join(t2, NULL);

    printf("var = %d\n", var);

    pthread_mutex_destroy(&mymutex);

    return 0;
}
```

Ricorda: lock e unlock prendono come argomento l'indirizzo della variabile mutex (notare &)

## Acquisizione non bloccante di un mutex

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

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

0 se si ottiene il lock,  
EBUSY altrimenti

Indirizzo della variabile mutex da acquisire

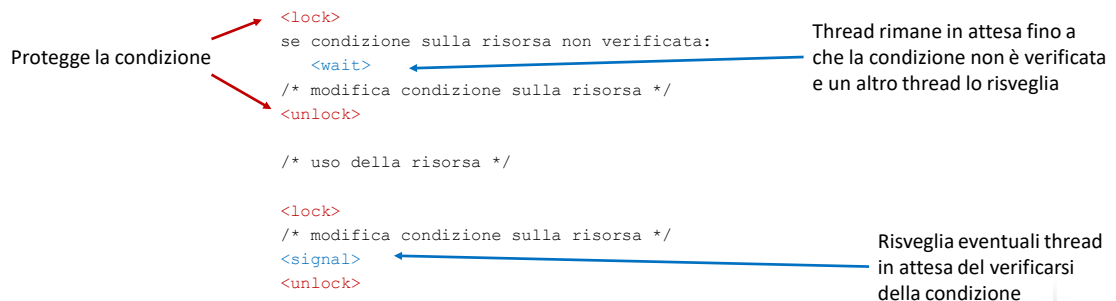
- Se il mutex è stato acquisito da un altro thread, il thread chiamante non si blocca
- Si usa testando il valore di ritorno in un'istruzione condizionale

### Versione non bloccante della pthread\_mutex\_lock

Quando restituisce EBUSY non dobbiamo eseguire la sezione critica, per questo si usa testando il valore di ritorno in un'istruzione condizionale (if, while, ...)

## Variabili condition

- Meccanismo per permettere ai thread di eseguire una sezione di codice **quando una specifica condizione è verificata**
- Deve essere sempre associata a un mutex per testare la condizione in mutua esclusione



È un meccanismo di sincronizzazione aggiuntivo, che si usa per scopi diversi dai mutex. La condizione è viene definita dal programmatore, in base alla logica richiesta dal programma. Ad esempio:

- si vuole imporre che un thread A esegua la sezione di codice prima di un thread B
- si vuole imporre che la sezione di codice possa essere eseguita in contemporanea da al più 3 thread.

Non è necessario che la sezione di codice in questione sia eseguita in mutua esclusione.

È necessario, invece, che la condizione sia verificata in mutua esclusione. Nella slide, <lock> e <unlock> circondano il controllo della condizione, non l'uso della risorsa.

## Inizializzazione statica

```
#include <pthread.h>

pthread_cond_t mycond = PTHREAD_COND_INITIALIZER;

int main()
{
    ...
}
```

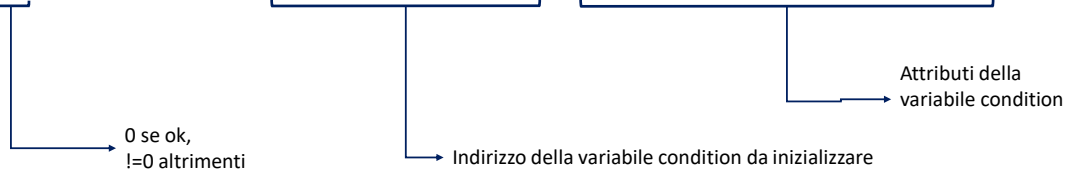
- `PTHREAD_COND_INITIALIZER` è una costante definita dalla libreria `pthread.h`

Variabile di tipo `pthread_cond_t`, che si inizializza analogamente a una variabile `mutex` (staticamente o dinamicamente, secondo le stesse considerazioni fatte per i `mutex`)

## Inizializzazione dinamica

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

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



- Inizializza la variabile condition passata come primo argomento
- L'argomento `*cattr` può essere `NULL` per impostare attributi di default

## Inizializzazione dinamica

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

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

0 se ok,  
!=0 altrimenti

Indirizzo della variabile condition da distruggere

- Quando la variabile condition non serve più, deve essere deallocato
- Solo se inizializzata dinamicamente
- Fallisce se un thread è in attesa sulla variabile condition → restituisce `EBUSY`

## Attesa su una variabile condition

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

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

0 se ok,  
!=0 altrimenti

Indirizzo della variabile  
condition su cui attendere

Indirizzo della variabile mutex  
associata alla condition

- Si invoca quando si vuole che il thread si blocchi, a seguito della (non) verifica della condizione associata
- Il mutex viene rilasciato
  - Evita deadlock
- Quando la funzione ritorna correttamente (al risveglio), il mutex viene nuovamente acquisito

A differenza della `pthread_mutex_lock`, questa funzione è **sempre** bloccante.

Dato che la condizione deve essere stata controllata acquisendo un mutex, il thread rilascia il mutex automaticamente quando si blocca con `pthread_cond_wait`. Se non lo facesse, altri thread non potrebbero a loro volta controllare (e modificare) la condizione, causando un deadlock.

## Risveglio da una variabile condition

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

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

0 se ok,  
!=0 altrimenti

Indirizzo della variabile condition da  
cui un thread deve essere risvegliato

- Uno dei thread bloccati sulla condizione viene risvegliato
  - Non è possibile decidere quale
- Deve essere invocata solo dopo aver modificato la condizione associata alla variabile
  - Il mutex deve essere stato acquisito
- Il mutex associato alla variabile condition deve essere rilasciato **esplicitamente**

Ha senso invocare la `pthread_cond_signal` dopo che la condizione è stata modificata, per dare modo ai thread bloccati sulla wait di testare nuovamente la condizione una volta che verranno risvegliati e trovarla verificata.

A differenza della `pthread_cond_wait`, **non** rilascia il mutex automaticamente.



## Risveglio da una variabile condition

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

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

0 se ok,  
!=0 altrimenti

Indirizzo della variabile condition da  
cui un thread deve essere risvegliato

- Risveglia tutti i thread in attesa
- Se tutti i thread si risvegliano, come si garantisce la mutua esclusione?

Quando un thread viene risvegliato (ritorna dalla `pthread_cond_wait`) riacquisisce il lock.

## Variabili condition - esempio

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

int presente = 0;
char* msg;

pthread_mutex_t mymutex = PTHREAD_MUTEX_INITIALIZER;
pthread_cond_t mycond = PTHREAD_COND_INITIALIZER;

void* read_message(void* arg)
{
    pthread_mutex_lock(&mymutex);
    if (presente == 0)
        pthread_cond_wait(&mycond, &mymutex);
    presente--;
    pthread_mutex_unlock(&mymutex);
    printf("letto %s\n", msg);
    pthread_exit(NULL);
}

void* write_message(void* arg) {
    msg = (char*)malloc(strlen("Ciao Mondo")+1);
    strcpy(msg, "Ciao Mondo");
    pthread_mutex_lock(&mymutex);
    presente = 1;
    pthread_mutex_unlock(&mymutex);
    pthread_cond_signal(&mycond);
    pthread_exit(NULL);
}

int main() {
    pthread_t t1, t2;

    pthread_create(&t1, NULL, read_message, NULL);
    pthread_create(&t2, NULL, write_message, NULL);

    pthread_join(t1, NULL);
    pthread_join(t2, NULL);

    return 0;
}
```

In questo esempio, si vuole imporre che il thread “lettore” esegua **dopo** il thread “scrittore” (che alloca la memoria e scrive un messaggio).  
La variabile “presente” definisce la condizione.

## Variabili condition - esempio

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

pthread_mutex_t mymutex = PTHREAD_MUTEX_INITIALIZER;
pthread_cond_t mycond = PTHREAD_COND_INITIALIZER;
unsigned int resources = 0;

void* get_res(void* arg) {
    int amount = *(int*)arg;
    pthread_mutex_lock(&mymutex);
    while (amount > resources) {
        pthread_cond_wait(&mycond, &mymutex);
    }
    resources -= amount;
    pthread_mutex_unlock(&mymutex);
}

void* add_res(void* arg) {
    int amount = *(int*)arg;
    pthread_mutex_lock(&mymutex);
    resources += (int)amount;
    pthread_cond_broadcast(&mycond);
    pthread_mutex_unlock(&mymutex);
}

int main() {
    pthread_t gt[3], at[3];
    unsigned int n=2, m=2;
    unsigned int i;
    for (i = 0; i<3; i++)
        pthread_create(&at[i], NULL, add_res, (void*)&n);
    for (i = 0; i<3; i++)
        pthread_create(&gt[i], NULL, get_res, (void*)&m);
    for (i = 0; i<3; i++) {
        pthread_join(at[i], NULL);
        pthread_join(gt[i], NULL);
    }
    return 0;
}
```

In questo esempio ciascun thread “add\_res” aggiunge un numero di risorse a un pool, mentre ciascun thread “get\_res” preleva un numero di risorse dal pool, se ce ne sono abbastanza.

E’ necessario il risveglio broadcast in quanto dopo l’aggiunta delle risorse al pool, quest’ultimo potrebbe contenere un numero di risorse sufficiente a servire le richieste di più di un thread bloccato.