Signal and wait

```
#include <pthread.h>
int pthread_cond_wait(pthread_cond_t *cond,
                      pthread_mutex_t *mutex);

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

`pthread_cond_signal` restarts one thread waiting on `cond`:

- if no threads are waiting on `cond`, nothing happens
- if several are waiting, exactly one is restarted, (not specified which)

`pthread_cond_broadcast` restarts all threads waiting on the condition `cond`.

`pthread_cond_wait` ha la semantica descritta a p. 173,174:

- calling thread should own the mutex on entrance to this function
- `pthread_cond_wait` atomically unlocks the mutex (as per `pthread_unlock_mutex`) and waits for the condition variable cond to be signaled
- the calling thread execution is suspended and does not consume any CPU time until the condition variable is signaled.
- before returning to the calling thread, `pthread_cond_wait` re-acquires mutex (as per `pthread_lock_mutex`).

Conditions must always be associated with mutexes. According to the manual (see also p. 173):

- if a thread always acquires the mutex before signaling a condition,
- then the atomicity of unlocking/suspending in `wait`
- guarantees that the condition cannot be signaled (and thus ignored) between unlocking and suspending

Declaring, initializing and destroying conditions

```
#include <pthread.h>

pthread_cond_t cond = PTHREAD_COND_INITIALIZER;

int  pthread_cond_init(pthread_cond_t  *cond,
                       pthread_condattr_t *cond_attr);
int pthread_cond_destroy(pthread_cond_t *cond);
```

Initialization:

- `pthread_cond_init` initializes the condition variable `cond` using the attributes `cond_attr` (`NULL` to get default attributes)
- the LinuxThreads implementation supports no attributes, so effectively ignores `cond_attr`
- variables of type `pthread_cond_t` can also be initialized statically, using the constant `PTHREAD_COND_INITIALIZER`, e.g.:

```
pthread_cond_t cond = PTHREAD_COND_INITIALIZER;
```

Destruction

- `pthread_cond_destroy` destroys a condition variable, freeing the resources it might hold
- no threads must be waiting on the condition variable on entrance to `pthread_cond_destroy`
- in the LinuxThreads implementation, no resources are associated with condition variables, thus `pthread_cond_destroy` actually does nothing except checking that the condition has no waiting threads.

Condition var: uso tipico

Azioni tipiche di programmazione con thread:

1. accesso allo stato globale
2. modifica dello stato globale
3. attesa del realizzarsi di condizioni (sullo stato)
4. segnalazione che una condizione si è realizzata

Evidentemente (1,2) vanno protette da accessi concorrenti da altri thread, all'interno di sezioni critiche. P.es.:

```
...
lock(&mutex);
...
//  access/modify state
...
unlock(&mutex);
...
```

Come detto, all'interno della sezione critica, le condition variable e operazioni associate consentono:

- di attendere che altri thread realizzino una condizione (predicati) sullo stato globale
- di segnalare ad altri thread (supposti in attesa) che tale condizione di è realizzata.

Problema (v. p. 172):

- il thread che sta per attendere non può e non deve mantenere bloccata `mutex`,
- o gli altri, se rispettano la disciplina delle sezioni critiche, non potranno entrarvi
- né tantomeno realizzare la condizione attesa dal primo thread!

Condition variables

A *condition (variable)* is a synchronization device that allows threads to

- suspend execution (and relinquish the processors)
- until some *condition* (predicate) on shared data becomes true

The basic operations on conditions `cond` are:

- *signalling* `cond` (when the predicate becomes true) to whoever is waiting for it;
- *waiting* on `cond`, suspending the thread execution until another thread signals the condition.

Mutex: unlocking

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

unlocks the mutex, assumed to be owned by the calling thread, and:

- if the mutex is fast, always returns it to the unlocked state
- if it is recursive, decrements the locking count of the mutex;
  only when this gets to 0 the mutex is actually unlocked;
- if mutex is error checking, `pthread_mutex_unlock` checks that
  - the mutex is locked on entrance, and that
  - it was locked by the calling thread.

  If not, an error code is returned and the mutex remains unchanged

Thus:

- fast and recursive mutexes may get unlocked by a thread other than its owner;
- but this is non-portable behavior and must not be relied upon.

Mutex: locking

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

If the mutex is already locked by the thread calling `pthread_mutex_lock`, what happens depends on the mutex kind:

- if the mutex is fast, the calling thread is suspended until the mutex is unlocked,

  this effectively causes the calling thread to deadlock!

- if mutex is error-checking, `pthread_mutex_lock` returns immediately with error EDEADLK;

- if mutex is recursive, `pthread_mutex_lock` succeeds and returns immediately, recording the number of times the calling thread has locked the mutex.

  An equal number of `pthread_mutex_unlock` operations must be performed before a recursive mutex returns to the unlocked state.

`trylock()` behaves identically to `lock()`, except:

- it does not block the calling thread if mutex is already locked by another thread (or by the calling thread for a fast mutex)

- instead, it returns immediately, with error EBUSY

Mutex attributes: creation and destruction

```
#include <pthread.h>

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

`pthread_mutexattr_init` initializes the mutex attribute object `attr` and fills it with default values for the attributes.

`pthread_mutexattr_destroy` destroys the mutex attribute object `attr`, which must not be reused until it is reinitialized (it does nothing in the LinuxThreads implementation).

LinuxThreads supports only one mutex attribute, the mutex kind, which can be:

- `PTHREAD_MUTEX_FAST_NP` for “fast” mutexes
- `PTHREAD_MUTEX_RECURSIVE_NP` for “recursive” mutexes
- `PTHREAD_MUTEX_ERRORCHECK_NP` for “error checking” mutexes

These constants are used for the 2nd argument in the functions:

```
int pthread_mutexattr_setkind_np(pthread_mutexattr_t *attr,
                                 int kind);
int pthread_mutexattr_getkind_np(
        const pthread_mutexattr_t *attr, int *kind);
```

The alternative to this “dynamic” initialization is the static style one:

```
pthread_mutex_t fast = PTHREAD_MUTEX_INITIALIZER;
pthread_mutex_t rec  = PTHREAD_RECURSIVE_MUTEX_INITIALIZER_NP;
pthread_mutex_t errck= PTHREAD_ERRORCHECK_MUTEX_INITIALIZER_NP;
```

All the `_np`/`_NP` suffixes indicate a non-portable extension to POSIX.

Mutex: creation and destruction

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

initializes mutex `*mutex` according to attributes `*mutexattr` (set to `NULL` for default attributes).

LinuxThreads supports only one mutex attribute, the *mutex kind*, which can be

- *fast* (default)
- *recursive*
- *error checking*

Mutex kind determines whether a mutex can be locked again by a thread that already owns it (see p. 163).

Mutexes can also be initialized statically:

```
pthread_mutex_t fast = PTHREAD_MUTEX_INITIALIZER;
pthread_mutex_t rec  = PTHREAD_RECURSIVE_MUTEX_INITIALIZER_NP;
pthread_mutex_t errck= PTHREAD_ERRORCHECK_MUTEX_INITIALIZER_NP;


int pthread_mutex_destroy(pthread_mutex_t *mutex);
```

destroys a mutex object, which must be unlocked, freeing the resources it might hold.

In LinuxThreads, no resources are associated with mutexes thus this function does nothing except checking the mutex is unlocked. If so, error `EBUSY` is returned.

Mutex

A mutex is a MUTual EXclusion device, useful for:

- protecting shared data structures from concurrent modifications, and
- implementing critical sections and monitors.

A mutex has two possible states:

- unlocked (not owned by any thread)
- locked (owned by one thread)

A thread attempting to lock a mutex already locked by another thread is suspended until the *owning thread* unlocks the mutex.

Note that only the owning thread may (or should) unlock the mutex; see p. 164 for unlockings by other threads;

A shared global variable `x` can be protected by a mutex as follows:

```
int x;
pthread_mutex_t mut = PTHREAD_MUTEX_INITIALIZER;
```

All accesses and modifications to `x` should be bracketed as follows:

```
pthread_mutex_lock(&mut);
/* operate on x */
pthread_mutex_unlock(&mut);
```

Detach e Join

```
#include <pthread.h>
int pthread_detach(pthread_t th);
```

scopo: usato dopo l'inizializzazione per `th` in stato *joinable*;

se `th` è già *detached*, ha effetto nullo e dà errore `EINVAL`.

```
#include <pthread.h>
int pthread_join(pthread_t th,void **thread_return);
```

sospende il thread chiamante finché il codice eseguito da `th` ritorna o `th` viene *cancellato*.

Se `thread_return` non è `NULL`, nella locazione a cui punta andrà:

- il `(void * val)` restituito dal thread `th`, con `pthread_exit(val)`
- o `PTHREAD_CANCELED` se `th` viene *cancellato*

NB: when a joinable thread terminates:

- its memory resources (thread descriptor and stack) are not deallocated until another thread performs `pthread_join` on it.
- therefore, to avoid memory leaks, `pthread_join` must be called once for each joinable thread created

At most one thread can wait for the termination of a given thread.

Errori, con return value non zero:

ESRCH — no thread `th` could be found

EINVAL — `th` is not joinable, or another thread is already blocked on `pthread_join(th,...)`

EDEADLK — `th` is the calling thread

Operazioni semplici sui thread

```
#include <pthread.h>
pthread_t pthread_self(void);
```

Restituisce il thread corrente (quello che la chiama)

```
#include <pthread.h>
int pthread_equal(pthread_t thread1,pthread_t thread2);
```

```
#include <pthread.h>
void pthread_exit(void *retval);
```

N.B.: questa funzione non ritorna (né restituisce valori)!

Gli altri thread hanno accesso a `retval` via `pthread_join()`
(v. p. 148)

Inizializzazione degli attributi

```
#include <pthread.h>
int pthread_attr_init(pthread_attr_t *attr);
```

Serve a inizializzare un oggetto `pthread_attr_t`, da passare poi come 2o arg a `pthread_create()`.

Gestione dell'oggetto-attributi `pthread_attr_t a`:

- si distrugge con `pthread_attr_destroy(&a)` (ma `a` si può sempre ri-inizializzare)
- si può *utilizzare* per creare thread, anche multipli, con `pthread_create(t,&a,...)` multipli
- si può *distruggere* o *modificare* senza influenzare i thread nella cui creazione è intervenuto

Il singolo attributo `xxx` si può modificare/leggere con:

- `pthread_attr_setxxx(pthread_attr_t *attr, xxx_t newxxx)` dove `newxxx` è il valore da dare all'attributo `xxx`
- `pthread_attr_getxxx(pthread_attr_t *attr, xxx_t * curxxx)` dove `curxxx` punta al valore corrente di `xxx`

Gli attributi per cui sta `xxx` sono:

- detached/joinable state
- scheduling parameters
- scheduling policy
- inherited scheduling parameters
- scope (della contesa per lo scheduling)

Subito dopo l'inizializzazione, gli attributi hanno dei valori di default.

Errore comune, per le funzioni `setxxx` che restituiscono non zero:

`EINVAL` il valore richiesto per l'attributo non è valido

Terminazione dei thread

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

Il nuovo thread che esegue `start_routine` può terminare:

- esplicitamente, con `pthread_exit()` o
- implicitamente, ritornando dalla funzione `start_routine`
  ciò equivale per il thread a chiamare `pthread_exit()`, con exit code pari al valore restituito da `start_routine`
- **NB** `exit()`, chiamato da qualsiasi thread, termina il **processo**, quindi tutti i thread!

Creazione di thread

Creazione di un thread, eseguito concorrentemente rispetto al thread chiamante:

```
#include <pthread.h>
int pthread_create( pthread_t * thread, pthread_attr_t * attr,
                    void * (*start_routine)(void *),
                    void * arg );
```

dove:

`thread` punta all'identificatore del thread creato

`attr` punta agli attributi richiesti per il nuovo thread o `NULL` per attributi di default, cioè:

- thread *joinable*
- default scheduling policy

`start_routine` (puntatore alla) funzione che il thread eseguirà

`arg` — argomento da passare a `start_routine`

Errori, se `pthread_create()` restituisce non zero: `EAGAIN` se:

- not enough resources to create a process for new thread
- more than `PTHREAD_THREADS_MAX` threads already active

NB: i thread sono in realtà implementati come processi.

Alternative implementation models

There are basically two other models.

- *many-to-one*: a user-level scheduler context-switches among threads entirely in user code;

  viewed from the kernel, there is only one process running.

- *many-to-many* (most commercial Unix systems e.g. Solaris);

  combines both kernel-level and user-level scheduling: for each multithreaded process:

  ⋆ several kernel-level threads run concurrently

  ⋆ each executes a user-level scheduler that selects among actual user threads.

“Many-to-one” evaluation:

+ context switches on mutex/condition blocking operations are quick (do not go through the kernel)

− but, user-level scheduling does/can not take advantage of multiprocessors, and

− requires tricks to handle blocking I/O operations properly

− hence problems wrt functionality, performance, and/or robustness

“Many-to-many” evaluation:

+ combines the advantages of both the ”many-to-one” and the ”one-to-one” model

+ avoids the worst-case behaviors of both models – especially on kernels where context switches are expensive

− pretty complex to implement

− requires kernel support which Linux does not provide

LinuxThreads: Implementazione

LinuxThreads follows the so-called *one-to-one* model:

- each thread is actually a separate process in the kernel;
- the original process that spawns the first thread is a thread too;
- an extra process acts as a “thread manager” (asleep most of the time).

The kernel scheduler takes care of scheduling the threads, just like it schedules regular processes.

Threads are created with the Linux `clone()` system call, a variation of `fork()` whereby new process and parent share: memory space, file descriptors, and signal handlers.

According to Posix, threads should share all this, and *pid* and *parent pid*, but the latter two are impossible for the current `clone` version.

Advantages of the “one-to-one” model include:

- minimal overhead on CPU-intensive multiprocessing (with about one thread per processor);
- minimal overhead on I/O operations;
- a simple and robust implementation (the kernel scheduler does most of the hard work).

Main disadvantage:

- more expensive context switches arise on mutex/condition blocking operations, which must go through the kernel
- but context switches in the Linux kernel are pretty efficient; using `clone()` vs. `fork()` helps in this respect

I Thread: la libreria LinuxThreads

È quasi conforme allo standard POSIX 1003.1c per i thread.

Altra documentazione:

- `/usr/share/doc/glibc-*/README.threads`
- `/usr/share/doc/glibc-*/FAQ-threads.html`, anche per link utili e tutorial
- `/usr/share/doc/glibc-*/examples.threads/`
- `http://mas.cs.umass.edu/~wagner/threads_html` per un tutorial