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

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

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.

Uso di LinuxThreads

Thread-safe functions

Functions which it is safe to call from threads originating from the same main thread.

Most libraries are not necessarily thread-safe:

- calling simultaneously two functions of the library from two threads might not work

  usual cause: internal use of global variables and the like.

- the libraries must have been compiled with `-D_REENTRANT` (see p, **??**).

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.

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!

Attributi dei thread

[Pagina intenzionalmente vuota]

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

Attributi di un thread

Possibili attributi (`xxx` in `setxxx`, `getxxx`):

- `detachstate`
- `schedpolicy`
- `schedparam`
- `inheritsched`
- `scope`

Solo il primo non riguarda lo scheduling.

`int detachstate`

valori: `PTHREAD_CREATE_JOINABLE` (default) o `PTHREAD_CREATE_DETACHED`

se il thread `t` è *joinable*, un altro thread può, con (v. p. 14) `pthread_join(t,...)`, sincronizzarsi sulla terminazione di `t` e ottenerne l'exit code;

tuttavia le risorse di `t` sono recuperate, non alla terminazione, ma solo quando viene eseguito un `pthread_join(t,...)`

Attributi di un thread (+)

Attributi per lo scheduling:

`int schedpolicy`

valori: (vedi `sched_setscheduler()` e scheduling POSIX)

`SCHED_OTHER` — scheduling standard di Linux: nessuna garanzia; è il default; gli altri sono permessi solo per processi super-user (se no, errore `ENOTSUP`)

`SCHED_FIFO` — un thread (in generale processo) in testa alla lista di scheduling gira fino a:

- completamento, blocco per I/O o rinuncia (`sched_yield()`)
- preemption da parte di processo in lista FIFO di maggior priorità

`SCHED_RR` — realtime, round-robin: come `SCHED_FIFO`, ma il thread corrente viene *preempted* dopo un quanto fisso garantito

`struct sched_param schedparam` — i suoi membri codificano i parametri della scheduling policy:

- unico struct member usato correntemente: `.sched_priority` (default `0`)

  (vedi anche, per *processi* anziché thread, `sched_setparam`)
- attributo irrilevante per policy `SCHED_OTHER`

Attributi di un thread (+)

Altri attributi per l scheduling:

`int inheritsched`

vale `PTHREAD_EXPLICIT_SCHED` o `PTHREAD_IMPLICIT_SCHED`

rispettivamente specificano se policy e parametri di scheduling sono:

- definiti dai relativi attributi (default)
- ereditati dal thread padre

`int scope`

definisce l'ambito della contesa per lo scheduling, cioè:

`PTHREAD_SCOPE_SYSTEM` (default)

`PTHREAD_SCOPE_PROCESS` - previsto dai thread standard di POSIX, ma non implementato in Linux (errore `ENOTSUP`)

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. 14)

Detach e Join

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

usato dopo l'inizializzazione per `th` in stato *joinable*, lo rende *detached*

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

Cancellazione di thread

```
#include <pthread.h>
int pthread_cancel(pthread_t cth);
```

Invia una richiesta di *cancellazione* (terminazione) a `cth`, che:

- ignora la richiesta se ha `cancelstate` `PTHREAD_CANCEL_DISABLE`
- la onora se ha `cancelstate` `PTHREAD_CANCEL_ENABLE`; l'uscita ha exit code `PTHREAD_CANCELED` per un eventuale `pthread_join(cth, thread_return)`;

La cancellazione con `pthread_cancel()` ha effetto:

- immediato se il `canceltype` di `cth` è `PTHREAD_CANCEL_ASYNCHRONOUS`
- differito fino al prossimo **cancellation point** se il `canceltype` è `PTHREAD_CANCEL_DEFERRED`

I cancellation point dei pthreads sono le chiamate a:

```
pthread_join
pthread_cond_wait, pthread_cond_timedwait
pthread_testcancel        // serve solo a questo
sem_wait
sigwait
```

NB:

- POSIX richiede che siano cancellation point anche le system call che possono bloccarsi, ma LinuxThreads non è conforme.
- Rimedio: tali system call vanno racchiuse tra due `testcancel()`.

Cancellazione di thread (+)

Stato e tipo di cancellazione si manipolano con:

```
#include <pthread.h>
int pthread_setcancelstate(int state,int *oldstate);
int pthread_setcanceltype(int type,int *oldtype);
```

dove i parametri *old* possono essere `NULL`.

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. 21 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);
```

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. 20).

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 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: 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: 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.

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.

Condition var e predicati

Nozione di predicato: funzione booleana dello *stato* del thread che lo valuta.

Stato di un thread:

- variabili globali (rispetto a tutti i thread)
- variabili locali al thread
- contatore di programma di qualche (altro) thread

Data una condition variable `cond`

- `cond` *non* è un predicato
- `cond` *non* è associata automaticamente a un predicato
- logicamente, `cond` *dovrebbe* esserlo (`wait(cond)` serve ad aspettare il realizzarsi di tale predicato)
- l'associazione viene stabilita dal programmatore, con lo schema:

```
if (!pred) wait(&cond);
```

La dipendenza di un predicato da un contatore di programma di task non si può esprimere; ⇒se tale dipendenza è l'unica, può capitare:

- `wait(&cond)` non protetto da `if(!pred)`
  p.es. `broad.c` (p. 45) contiene `wait` protette e non
- ma `wait()` sono bloccanti, quindi, se non protette, sono **potenziali pericoli** (v. p. 24)

⇒ può convenire "inventare" variabili che:

- rispecchino il contatore di programma di un task
- consentendo di costruire sempre un predicato per proteggere una `wait()`

Condition var: precauzioni su `wait`

Caratteristica semantica di `signal`/`wait`:

a. `signal(cond)` non viene memorizzata, cioè:

b. se `signal(cond)` non trova thread bloccati su `wait(cond)`, non accade nulla

c. il primo `wait(cond)` successivo a un caso (b) blocca comunque

⇒Avvertenze per il programmatore:

1. prevedere `signal` che sveglino i thread in attesa su `wait`
2. inoltre, far sì che i `wait` di un thread siano *seguiti* da appropriati `signal` di un altro thread
3. evitare che, inavvertitamente, un `signal` di un thread preceda il `wait` di un thread che dovrebbe svegliare
4. preferibilmente, non chiamare `wait` senza la certezza che ve ne sia bisogno

   (non farlo aumenterebbe gli oneri (1,2))
5. preferibilmente, proteggere i `wait` con opportuni `if`

   (questo disciplina a rispettare (1,2) ed evitare (4))

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. 28):

- il thread che sta per attendere con `wait()` 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!

Monitor

Condition variable e mutex permettono di realizzare facilmente un *monitor*, cioè:

- un *tipo di dato astratto* o *ADT* (= classe Java/C++, a parte l'ereditarietà)

  ⇒Un ADT è un gruppo di strutture dati *incapsulate* e accessibili solo mediante un insieme prescritto di operazioni.

- dotato di operazioni *atomiche*, cioè per cui vale la cosiddetta proprietà di *serializzabilità*:
  - se i thread 1,2 eseguono rispettivamente le operazioni di monitor `f1()` e `f2()`
  - comunque si intreccino le azioni di 1 e 2,
  - l'"effetto netto" equivale ad eseguire prima `f1()` e poi `f2()`, oppure il contrario
  - nessun altro "effetto" (sullo stato globale) può verificarsi

Quindi, dal punto di vista dell'uso dei monitor:

- il programmatore di applicazioni multi-threaded può di fatto ignorare il multithreading, se usa i monitor!

⇒ i monitor sono un (potente) strumento di astrazione

Dal punto di vista dei meccanismi:

- lo strumento per assicurare atomicità/serializzabilità delle operazioni dei monitor è racchiuderle in una sezione critica
- il codice delle operazioni usa `wait`/`signal` e condition variables per gestire eventuali attese di condizioni

Monitor: esempio

Un monitor è costituito da:

- tipo di dato `account_t`
- operazioni

```
int getAccVal(account_t * a);
void addAcc(account_t * a, int delta);
```

Il codice del monitor potrebbe essere:

```
typedef struct {
   int val;
   pthread_mutex_t mutex;
} account_t;

int getAccVal(account_t * a)
{
   int v;

   lock(&(a->mutex));
   v = a->val;
   unlock(&(a->mutex));
   return(v);
}

void addAcc(account_t * a, int delta)
{
   int v;

   lock(&(a->mutex));
   v = a->val;
   v += delta;
   a->val = v;
   unlock(&(a->mutex));
}
```

Se due thread eseguissero “quasi contemporaneamente” `addAcc(a,10)`, con `a->val` pari a 100:

- ma `addAcc` omettesse il `lock` iniziale su `a->mutex`,
- alla fine `a->val` potrebbe valere `110`,
- mentre la serializzabilità prescrive `120` come valore finale!

Per un’attesa di condizioni, v. `synqueue`/`synqueue1`, p. 51/53.

Attesa all'interno di sezioni critiche

Come detto (p. 25), l'accesso a variabili globali da parte di un thread va protetto all'interno di una sezione critica:

```
...
lock(&mutex);      // ingresso sez. critica
...
//  access/modify state
...
unlock(&mutex);    // uscita sez. critica
...
```

Quindi, se, mentre un thread è all'interno di una sezione:

- deve attendere il realizzarsi di una condizione,
- esso deve prima sbloccare la sezione (anche perché altri thread possano realizzare la condizione)

Tentativo:

```
...
lock(&mutex);      // ingresso sez. critica
...
unlock(&mutex);
wait(&cond);
...
unlock(&mutex);    // uscita sez. critica
```

E in genere, il programmatore desidera una sincronizzazione di questo tipo, sulle sezioni critiche:

```
Thread 1                Thread 2
...                     ...
lock(&mutex);           ...
...                     lock(&mutex) //  si blocca
...
unlock(&mutex);         //  si sblocca
wait(&cond);            ...
                        //  rende vera la cond
                        signal(&cond);
```

- ma niente assicura che `wait` di 1 preceda davvero `signal` di 2, come nello scenario sopra!
- e se `signal` precede `wait`, si perde ⇒problema (cf. p. 24)!

Condition e mutex

Il problema di p. 28 è che, nel primo thread, `unlock` e `wait` dovrebbero essere tutt'uno:

```
   Thread 1                Thread 2
   ...                     ...
   lock(&mutex);           ...
   ...                     lock(&mutex) //  si blocca
   ...
<< unlock(&mutex);         //  si sblocca
   wait(&cond);            ...
>>                         //  rende vera la cond
                           signal(&cond);
```

Ciò impedirebbe che `signal` si possa inserire tra `unlock` e `wait`.

Un'altra questione: ammesso che tutto fili liscio, dopo questo frammento chi è in sezione critica?

Soluzioni:

- si introduce `wait(&cond,&mutex)` che:
  1. sblocca `mutex` e, atomicamente, attende su `cond`,
  2. fino a quando un thread non segnala `cond`;
  3. infine, cerca di riacquistare `mutex`
- il thread che esegue `signal` deve, prima o poi, sbloccare `mutex` (o, dato (3), l'altro non uscirà da `wait`!)

```
Thread 1                          Thread 2
...                               ...
lock(&mutex);                     ...
...                               ...
...                               lock(&mutex) //  si blocca
...
wait(&cond,&mutex);------->                    //  si sblocca
                                  ...
                                  ...          //  rende vera la cond
                       <-------   signal(&cond);
                                  ...
                       <-------   unlock(&mutex);
//  ora si sblocca
```

Condition e mutex (+)

```
Thread 1                          Thread 2
...                               ...
lock(&mutex);                     ...
...                               ...
...                               lock(&mutex) //  si blocca
...
wait(&cond,&mutex);------->                    //  si sblocca
                                  ...
                                  ...          //  rende vera la cond
                       <-------   signal(&cond);
                                  ...
                       <-------   unlock(&mutex);
//  ora si sblocca
```

Osservazioni:

- `signal` di thread 2 non basta a svegliare il thread 1 da `wait`: **occorre anche** `unlock`, perché `wait` ritorna solo dopo che il thread ha riacquistato il lock

  NB: quest'aspetto di `wait` è introdotto a p. 29; per il `man`, v. p. 33.

- non è facile far precedere il lock del thread 2 da quello del thread 1, come nello schedule sopra

  tuttavia, secondo il suggerimento della `man` page per le condition, realizzare questo schedule assicura che `wait` preceda `signal`

  un possibile approccio è illustrato nell'esempio `broad.c` (p. 45)

- `signal` potrebbe essere sostituito da `broadcast`:
  - equivale a `signal` spedito a *tutti* i thread fermi su una `wait` su `cond` (e una qualsiasi mutex)
  - se altri thread come 1 hanno fatto `wait` sulla stessa mutex,
    - l'`unlock` del thread 2 ne riattiverà di fatto solo uno
    - gli altri dovranno aspettare ulteriori `unlock` (p.es. da parte dei thread man mano riattivati);

  questi schemi sono illustrati nell'esempio `broad.c` (p. 45)

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 attribute initialization

Since the LinuxThreads implementation supports no attributes for conditions, the following functions are included only for compliance with the POSIX standard, but do nothing.

```
#include <pthread.h>

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

In principle,

- `pthread_condattr_init` initializes the condition attribute object `attr` and fills it with default values for the attributes
- `pthread_condattr_destroy` destroys a condition attribute object, which must not be reused until it is reinitialized.

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. 29,30:

- 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. 29):

- 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

Timed wait

```
#include <pthread.h>

int pthread_cond_timedwait(pthread_cond_t *cond,
                           pthread_mutex_t *mutex,
                           const struct timespec *abstime);
```

Function `pthread_cond_timedwait`:

- atomically unlocks `mutex` and waits on `cond`, as `pthread_cond_wait` does,
- but it also bounds the duration of the wait
- i.e. if `cond` has not been signaled within the amount of time specified by `abstime`:
  - `mutex` is re-acquired
  - `pthread_cond_timedwait` returns the error `ETIMEDOUT`

The `abstime` parameter specifies an absolute time:

- the time origin is as for `time(2)` and `gettimeofday(2)`
- i.e. an abstime of `0` corresponds to 00:00:00 GMT, January 1, 1970.

Esempi

Thread: hello world!

```
/* hello.c */

#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>
#include <unistd.h>

void print_message_function( void *ptr );

int main()
{
    pthread_t thread1, thread2;
    char *message1 = "Hello\n";
    char *message2 = "World\n";

    pthread_create( &thread1, NULL,
        (void*)&print_message_function, (void*) message1);


    pthread_create(&thread2, NULL,
        (void*)&print_message_function, (void*) message2);

// exit(0);    // Bug! May prevent threads to complete!
    pthread_join(thread1, NULL);
    pthread_join(thread2, NULL);
    return(0);
}

void print_message_function( void *ptr )
{
    char *message;
    message = (char *) ptr;
    printf("%s ", message);
// sleep(10);
}
```

Thread: argomenti e terminazione “pulita”

```
/* passargs.c */

/* Mostra come passare un argomento (logicamente piu‘ arg.) *
 * alla funzione eseguita da un thread.                      *
 *                                                           *
 * Illustra anche un trabocchetto: il main deve aspettare    *
 * i thread figli o uscire con pthread_exit();               *
 * se no la sua uscita e‘ un exit() implicito o esplicito    *
 * che fa terminare il processo che contiene i thread!       *
 * Si perde, p.es., la stdout!                               *
 * Viceversa sembra che i thread possano chiamare sleep(),   *
 * mentre altre implementazioni di POSIX threads lo          *
 * sconsigliano                                              */

#include <stdio.h>
#include <pthread.h>
#include <unistd.h>

struct thrd_arg {
    char * name;
    int val;
};

void * prnval(void * x)
{
    int i, arg;
    char * thrd_name;

    thrd_name = ((struct thrd_arg *) x)->name;
    arg = ((struct thrd_arg *) x)->val;

    for (i = 0; i < 8 ; i++) {
        printf("%s, i=%d, arg=%d\n",
               thrd_name, i, arg);
        sleep(1 + arg%2);   // attese diverse secondo arg
    }

    return NULL;
}


/* passargs.c … */
```

```
/* passargs.c (cont.) */

int main(int argc,char *argv[])
{
    pthread_t t1, t2;
    struct thrd_arg x1, x2;

    x1.name =       "Thread 1"; x1.val = 5;
    x2.name = "\t\t\tThread 2"; x2.val = 4;
    pthread_create(&t1, NULL, &prnval, (void *) &x1);
    pthread_create(&t2, NULL, &prnval, (void *) &x2);
// pthread_join(t1,NULL);
// pthread_join(t2,NULL);
    pthread_exit(NULL);      // in alternativa a 2 righe prec.
                                    // in mancanza, errore! perchï¿½ return 0;      // qu
turn ï¿½ un exit() !
}
```

Thread: sezioni critiche e mutex

```
/* thrdmutex.c */

/* due thread condividono una risorsa sincronizzandosi con
 * una mutex                                              */

#include <stdio.h>
#include <unistd.h>
#include <pthread.h>

pthread_mutex_t mutex;
int global_i = 0;
int accumulator;

void * incr(void * x)
{
    for (;;) {
        pthread_mutex_lock(&mutex);   // if critical section unlocked
        accumulator = global_i;       // accumulator may be printed
        accumulator++;                // greater than global_i
        usleep(1000L);
        global_i = accumulator;
//    printf("global_i = %d   accumulator = %d ",
//        global_i, accumulator);
        pthread_mutex_unlock(&mutex);
    }
}

void * print(void * x)
{
    for (;;) {
        pthread_mutex_lock(&mutex);
        printf("global_i = %d\taccumulator = %d\n",
               global_i, accumulator);
        usleep(3000L);
        pthread_mutex_unlock(&mutex);
    }
}

int main(int argc,char *argv[])
{
    pthread_t incr_id, print_id;

    pthread_mutex_init(&mutex, NULL);  // NULL: default mutex attrs
    pthread_create(&incr_id, NULL, incr, NULL);
    pthread_create(&print_id, NULL, print, NULL);
    pthread_join(incr_id, NULL);
    pthread_join(print_id, NULL);
    return 0;
}
```

Wait su una condizione: da man

```
/* manex1.c */

/* Example in man page for pthread_cond, slightly adjusted  *
 *                                                          */

#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <limits.h>
#include <pthread.h>

#define MAXCONS 4      // no more than will fit on a terminal line

int x,y;
pthread_mutex_t mut = PTHREAD_MUTEX_INITIALIZER;
pthread_cond_t cond = PTHREAD_COND_INITIALIZER;

void thrdPrint(int thrdindx, char * msg)
{
    int i;

    for (i = 0; i < thrdindx; i++)
        printf(".    ");
    printf(msg, thrdindx);
}

void * producer(void * arg)
{
    for (;;) {
        pthread_mutex_lock(&mut);
        thrdPrint(0, "<\n");
        x = rand(); y = rand();

        if (x > y) {
            pthread_cond_broadcast(&cond);
            thrdPrint(0, "xGTy\n");
        }
        thrdPrint(0, ">\n");
        pthread_mutex_unlock(&mut);

        sleep (1 + rand()%6);
    }
}

/* manex1.c … */
```

/* **manex1.c** (cont.) */

```
void * consumer(void * arg)
{
    int nc;     // this is the consumer/thread number

    nc = *((int * ) arg);

    for (;;) {
        pthread_mutex_lock(&mut);
        thrdPrint(nc, "<\n");
        while (x <= y) {
            thrdPrint(nc, ">\n");
            pthread_cond_wait(&cond, &mut);
            thrdPrint(nc, "<\n");
        }
        thrdPrint(nc, "xGTy\n");
        thrdPrint(nc, ">\n");
        pthread_mutex_unlock(&mut);

        sleep (nc + rand()%nc);
    }
}

int main(int argc, char *argv[])
{
    pthread_t thrd[1+MAXCONS];
    int i;
    int indx[1+MAXCONS];

    srand((unsigned int) time(NULL));

    for (i = 0; i <= MAXCONS; i++)   // indx[i] is used to pass
        indx[i] = i;                 // to each thread its index
                                     // 0=prod, 1...MAXCONS = cons
    pthread_create(&thrd[0],  NULL, producer,
                   (void *) &indx[0]);
    for (i = 1; i <= MAXCONS; i++)
        pthread_create(&thrd[i], NULL, consumer,
                       (void *) &indx[i]);

    pthread_exit(NULL);
    exit(0);
}
```

Timed wait: da man

```
/* manex2.c */

/* Example in man page for pthread_cond, slightly adjusted *
 *                                                         */

#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <limits.h>
#include <pthread.h>
#include <sys/time.h>
#include <errno.h>      // for ETIMEDOUT

#define MAXCONS 4      // no more than will fit on a terminal line

int x,y;
pthread_mutex_t mut = PTHREAD_MUTEX_INITIALIZER;
pthread_cond_t cond = PTHREAD_COND_INITIALIZER;

void thrdPrint(int thrdindx, char * msg)
{
    int i;

    for (i = 0; i < thrdindx; i++)
        printf(".    ");
    printf(msg, thrdindx);
}

void * producer(void * arg)
{
    for (;;) {
        pthread_mutex_lock(&mut);
        thrdPrint(0, "<\n");
        x = rand(); y = rand();

        if (x > y) {
            pthread_cond_broadcast(&cond);
            thrdPrint(0, "xGTy\n");
        }
        thrdPrint(0, ">\n");
        pthread_mutex_unlock(&mut);

        sleep (1 + rand()%6);
    }
}

/* manex2.c … */
```

/* **manex2.c** (cont.) */

```
void * consumer(void * arg)
{
    int nc;     // this is the consumer/thread number

    struct timeval now;
    struct timespec timeout;
    int retcode;

    nc = *((int * ) arg);

    for (;;) {
        pthread_mutex_lock(&mut);
        thrdPrint(nc, "<\n");

        gettimeofday(&now, NULL);
        timeout.tv_sec = now.tv_sec + rand()%6;
        timeout.tv_nsec = now.tv_usec * 1000;

        retcode = 0;
        while ( x <= y && retcode != ETIMEDOUT ) {
            thrdPrint(nc, ">\n");
            retcode =
               pthread_cond_timedwait(&cond, &mut, &timeout);
            thrdPrint(nc, "<\n");
        }
        if (retcode == ETIMEDOUT)
            thrdPrint(nc, "tout\n");
        else
            thrdPrint(nc, "xGTy\n");
        thrdPrint(nc, ">\n");
        pthread_mutex_unlock(&mut);

        sleep (nc + rand()%nc);
    }
}
```

/* **manex2.c** … */

/* **manex2.c** (cont.) */

```
int main(int argc, char *argv[])
{
    pthread_t thrd[1+MAXCONS];
    int i;
    int indx[1+MAXCONS];

    srand((unsigned int) time(NULL));

    for (i = 0; i <= MAXCONS; i++)  // indx[i] is used to pass
        indx[i] = i;                // to each thread its index
                                    // 0=prod, 1...MAXCONS = cons
    pthread_create(&thrd[0],  NULL, producer, &indx[0]);
    for (i = 1; i <= MAXCONS; i++)
        pthread_create(&thrd[i], NULL, consumer,
                       (void *) &indx[i]);

    pthread_exit(NULL);
    exit(0);
}
```

Thread: attenzione a broadcast

```
/* broad.c */

/* A very fine point: broadcast awakes all threads waiting *
 * on a cond, but if all of them made wait(&cond,&mutex)   *
 * for the same mutex, only one of them will grab the      *
 * mutex, the others will block again, as though they      *
 * tried lock(&mutex)                                      */

/* see also comments on interaction between locking        *
 * mutexes and waiting on condition variables              */

#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <limits.h>
#include <pthread.h>

#define MAXCONS 100

int cons_n, cons_wait_n;
int prod_time;

pthread_mutex_t mutex;
pthread_cond_t data_rdy, all_cons_wait;

void thrdPrint(int thrdindx, char * msg)
{
    int i;

    for (i = 0; i < thrdindx; i++)
        printf(".    ");
    printf(msg, thrdindx);
}

/* broad.c … */
```

/* **broad.c** (cont.) */

```
void * producer(void * x)
{
    thrdPrint(0, "makingData\n");
    sleep(prod_time);   // take some time to produce data
    thrdPrint(0, "dataMade\n");

    pthread_mutex_lock(&mutex);
    thrdPrint(0, "<\n");

    if (cons_wait_n < cons_n) {
        thrdPrint(0, ">waitAllCons\n");
        pthread_cond_wait(&all_cons_wait, &mutex);// wait, until all
        thrdPrint(0, "<waitEnded\n");              // consumers block
    }
    thrdPrint(0, "broadcastAllCons\n");      // consumers are blocked
    pthread_cond_broadcast(&data_rdy);       // so awake them

// thrdPrint("sleep-LetConsWait", 0); sleep(5);
// comment in previous line or comment out unlock() below, and
// consumers will stay on wait() for 5 secs or FOREVER, because,
// although producer broadcast them, they don't get the mutex

    thrdPrint(0, ">\n");
    pthread_mutex_unlock(&mutex);

    thrdPrint(0, "bye\n");
    return NULL;
}
```

/* **broad.c** ... */

/* **broad.c** (cont.) */

```
void * consumer(void * x)
{
    int nc;     // this is the consumer/thread number

    nc = *((int * ) x);

    pthread_mutex_lock(&mutex);
    thrdPrint(nc, "<\n");

    if (++cons_wait_n == cons_n) {       // last consumer going to wait
        pthread_cond_signal(&all_cons_wait);           // so signal this
        thrdPrint(nc, "signalAllConsWait\n");
    }
    thrdPrint(nc, ">waitData\n");
    pthread_cond_wait(&data_rdy, &mutex);         // wait for data
    thrdPrint(nc, "<waitEnded\n");

    thrdPrint(nc, ">\n");
    pthread_mutex_unlock(&mutex);

   // comment previous line out and all consumers but one will get
   // stuck on cond_wait(&prod,&mutex) despite producer's cond_broadcast();
   // trouble is they don't get mutex from the fastest consumer

    thrdPrint(nc, "bye\n");
    return NULL;
}
```

/* **broad.c** ... */

/* **broad.c** (cont.) */

```
int main(int argc, char *argv[])
{
    pthread_t thrd[1+MAXCONS];
    int i;
    int indx[1+MAXCONS];

    if ( argc != 3 ||
         (cons_n = atoi(argv[1])) > MAXCONS ) {
        printf("Usage: %s #_consumers prod_time (<= %d)\n",
               argv[0], MAXCONS);
        exit(1);
    }

    prod_time = atoi(argv[2]);
    pthread_mutex_init(&mutex, NULL);
    cons_wait_n = 0;    // no consumer waiting yet
    pthread_cond_init(&all_cons_wait, NULL);
    pthread_cond_init(&data_rdy, NULL);

    for (i = 0; i <= cons_n; i++)    // indx[i] is used to pass
        indx[i] = i;                 // to each thread its index
                                     // 0=prod, 1...MAXCONS = cons
    pthread_create(&thrd[0],  NULL, producer, &indx[0]);
    for (i = 1; i <= cons_n; i++)
        pthread_create(&thrd[i], NULL, consumer,
                       (void *) &indx[i]);

    pthread_exit(NULL);
    exit(0);
}
```

Thread: producer/consumer

```
/* thrdprodcons.c */

/* Thread producer-consumer */

#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <limits.h>
#include <pthread.h>

#include "synqueue.h"

#define MAXCONS 4

syn_queue_t queue;
int qsize, cons_n;

void * producer(void * x)
{
    long n, i = 0;

    show_str(0, "prod", "created");
    for (;;) {
        n = rand() % (cons_n * qsize) + 1;
        while (n-- > 0)
            q_put(&queue, i++);
        sleep(1);
    }
}

void * consumer(void * x)
{
    long v, n;
    int nr;

    nr = *((int * ) x);
    show_str(nr, "cons", "created");
    for (;;) {
        n = rand() % (2*qsize) + 1;
        while (n-- > 0)
            q_get(&queue, &v, nr);
        sleep(1);
    }
}

/* thrdprodcons.c … */
```

/* **thrdprodcons.c** (cont.) */

```
int main(int argc, char *argv[])
{
    pthread_t thrd[1+MAXCONS];
    int i;
    int indx[1+MAXCONS]; // indx[0] unused

    if ( argc != 3 ||
         (cons_n = atoi(argv[2])) > MAXCONS ) {
        printf("Usage: %s qsize #_consumers (<= %d)\n",
               argv[0], MAXCONS);
        exit(1);
    }

    q_init(&queue, qsize = atoi(argv[1]));
    srand((unsigned int) time(NULL));
    for (i = 1; i <= cons_n; i++) {
        indx[i] = i;
        pthread_create(&thrd[i], NULL, consumer,
                       (void *) &indx[i]);
    }
    pthread_create(&thrd[0],  NULL, producer, NULL);
    pthread_exit(NULL);

    exit(0);
}
```

Monitor coda: broadcast

```
/* synqueue.h */

#include "show.h"

typedef struct {
    int num, size;
    int head, tail;
    long * data;
    pthread_mutex_t mutex;
    pthread_cond_t for_space, for_data;
    int n_wait_sp, n_wait_dt;
} syn_queue_t;

int q_init(syn_queue_t * q, int size);
int q_put(syn_queue_t * q, long val);
int q_get(syn_queue_t * q, long * pval, int which_thrd);
void q_free(syn_queue_t * q);
```

```
/* synqueue.c */

/* ADT queue, synchronized nel senso del Java, cioe'        *
 * con operazioni atomiche. Si tratta quindi di un monitor. *
 *                                                          *
 * Funzionamento:                                           *
 * 1. chi non trova dati, fa wait su for_data               *
 * 2. chi produce un dato, sveglia (broadcast) chi è in wait*
 * 3. quindi chi si sveglia da wait potrebbe non trovare    *
 *    dati, se un altro svegliato li prende prima, e deve   *
 *    percio' riprovare (1) (con un while)                  */

#include <stdlib.h>
#include <pthread.h>
#include "synqueue.h"

int q_init(syn_queue_t * q, int size)
{
    q->num = q->head = q->tail = 0;
    q->size = size;
    q->data = (long *) malloc(sizeof(long) * size);
    pthread_mutex_init(&q->mutex, NULL);
    pthread_cond_init(&q->for_space, NULL);
    pthread_cond_init(&q->for_data, NULL);
    return (q->data == NULL) ? 1 : 0;
}
```

/* **synqueue.c** … */

/* **synqueue.c** (cont.) */

```
void q_free(syn_queue_t * q)
{
  if (q->data != NULL)
    free(q->data);
}

int q_get(syn_queue_t * q, long * val, int which_thrd)
{
    pthread_mutex_lock(&q->mutex);
    while (q->num == 0)    // try to change while/if -$¿$ error!
        pthread_cond_wait(&q->for_data, &q->mutex);
    *val = q->data[q->tail];
    show_val(which_thrd, "get", *val);
    q->tail = (q->tail + 1) % q->size;
    q->num--;
    pthread_cond_broadcast(&q->for_space);
    pthread_mutex_unlock(&q->mutex);
    return 0;
}

int q_put(syn_queue_t * q, long val)
{
    pthread_mutex_lock(&q->mutex);
    while (q->num == q->size)
        pthread_cond_wait(&q->for_space, &q->mutex);
    show_val(0, "put", val);
    q->data[q->head] = val;
    q->head = (q->head + 1) % q->size;
    q->num++;
    pthread_cond_broadcast(&q->for_data);
    pthread_mutex_unlock(&q->mutex);
    return 0;
}
```

Monitor coda: signal singolo

```
/* synqueue1.c */

/* ADT queue, synchronized nel senso del Java, cioe'        *
 * con operazioni atomiche. Si tratta quindi di un monitor. *
 */

#include <stdlib.h>
#include <pthread.h>

#include "synqueue.h"

int q_init(syn_queue_t * q, int size)
{
    q->num = q->head = q->tail = 0;
    q->size = size;
    q->data = (long *) malloc(sizeof(long) * size);
    pthread_mutex_init(&q->mutex, NULL);
    pthread_cond_init(&q->for_space, NULL);
    pthread_cond_init(&q->for_data, NULL);
    q->n_wait_dt = q->n_wait_sp = 0;
    return (q->data == NULL) ? 1 : 0;
}

void q_free(syn_queue_t * q)
{
  if (q->data != NULL)
    free(q->data);
}

/* synqueue1.c … */
```

/* **synqueue1.c** (cont.) */

```
int q_get(syn_queue_t * q, long * val, int which_thrd)
{
    pthread_mutex_lock(&q->mutex);
    if (q->num == 0) {
        q->n_wait_dt++;
        pthread_cond_wait(&q->for_data, &q->mutex);
    }
    *val = q->data[q->tail];
    show_val(which_thrd, "get", *val);
    q->tail = (q->tail + 1) % q->size;
    if (  (q->num-- == q->size)
        &&(q->n_wait_sp > 0) ) {
        q->n_wait_sp--;
        pthread_cond_signal(&q->for_space);
    }                                   // unlock also needed after signal
    pthread_mutex_unlock(&q->mutex);    //  or threads blocked on wait
    return 0;                           //  will not proceed
}

int q_put(syn_queue_t * q, long val)
{
    pthread_mutex_lock(&q->mutex);
    if (q->num == q->size) {
        q->n_wait_sp++;
        pthread_cond_wait(&q->for_space, &q->mutex);
    }
    q->data[q->head] = val;
    q->head = (q->head + 1) % q->size;
    show_val(0, "put", val);
    if (  (q->num++ == 0)
        &&(q->n_wait_dt > 0) ) {
        q->n_wait_dt--;
        pthread_cond_signal(&q->for_data);
    }                                   // unlock also needed after signal
    pthread_mutex_unlock(&q->mutex);    //  or threads blocked on wait
    return 0;                           //  will not proceed
}
```

Thread: dining philosophers

```
/* philosophers.c */

/* Dining Philosophers */

#include <stdio.h>
#include <pthread.h>
#include <stdlib.h>
#include <unistd.h>
#include <errno.h>

#define N   5
#define LEFT_n  n
#define RIGHT_n (n + 1) % N

#define THINK   0
#define HUNGRY  1
#define EAT     2
#define PRINT_ST(state) \
    ( (state == THINK)  ? "Think" : \
      (state == EAT)    ? "Eat" : \
      (state == HUNGRY) ? "Hungry" : \
                          "no such state")


pthread_mutex_t theFork[N], mutex;
pthread_cond_t state_changed;
int state[N];

void setPhState(int phil, int st)
{
    pthread_mutex_lock(&mutex);
    state[phil] = st;
    pthread_mutex_unlock(&mutex);
    // signal show_state thread
    pthread_cond_broadcast(&state_changed);
// pthread_cond_signal(&state_changed);      // should work too
}

void think(int n)
{
    setPhState(n,THINK);
#ifndef debug
    sleep(1+(int)(3.0*rand() / (RAND_MAX+1.0)));
#endif
}

/* philosophers.c ... */
```

/* **philosophers.c** (cont.) */

```
void eat(int n)
{
    setPhState(n,EAT);
#ifndef debug
    sleep(1+(int)(3.0*rand() / (RAND_MAX+1.0)));
#endif
}

void take_forks(int n)
// Get both forks for philosopher n (block until both available)
{
    setPhState(n,HUNGRY);
    for (;;) {
        pthread_mutex_lock(&theFork[LEFT_n]);
        if (pthread_mutex_trylock(&theFork[RIGHT_n]) != EBUSY)
            break;
        pthread_mutex_unlock(&theFork[LEFT_n]);

        pthread_mutex_lock(&theFork[RIGHT_n]);
        if (pthread_mutex_trylock(&theFork[LEFT_n]) != EBUSY)
            break;
        pthread_mutex_unlock(&theFork[RIGHT_n]);
    }
}

void put_forks(int n)
{
    setPhState(n, THINK);
// previous line is necessary, even though this will be done also
// when main loop restarts; otherwise, the unlocks below could set a
// neighbour to state EAT, and show_state() would display this phil's
// state as EAT
    pthread_mutex_unlock(&theFork[RIGHT_n]);
    pthread_mutex_unlock(&theFork[LEFT_n]);
}
```

/* **philosophers.c** ... */

/* **philosophers.c** (cont.) */

```
void * philosopher(void * x)
{
    int n = (int)x;
    for (;;) {
        think(n);
        take_forks(n);
        eat(n);
        put_forks(n);
    }
}

void * show_state(void * x)
{
    int n;
#ifdef debug
    int err, times = 0;
#endif

    for (n = 0; n < N; n++) {
        printf("%2d        ", n);
    }
    printf("\n");

    for (;;) {
        pthread_mutex_lock(&mutex);
        pthread_cond_wait(&state_changed, &mutex);
        for (n = 0; n < N; n++) {
#ifndef debug
            printf("%-10s", PRINT_ST(state[n]));
        }
        printf("\n");
#else
            if (state[n] == EAT && state[RIGHT_n] == EAT)
                    printf("Error at iter. %d\n", times);
        }
        ++times;
        if (times % 2000 == 0)
            printf("Iter %d\n", times);
#endif
        pthread_mutex_unlock(&mutex);
    }
}
```

/* **philosophers.c** ... */

/* **philosophers.c** (cont.) */

```
int main(int argc,char *argv[])
{
    pthread_t tid[N+1];
    int i;

    pthread_mutex_init(&mutex, NULL);
    pthread_cond_init(&state_changed, NULL);
    for (i = 0; i < N; i++)
        pthread_mutex_init(&theFork[i], NULL);

    srand((unsigned int) time(NULL));

    for (i = 0; i < N; i++)
        pthread_create(&tid[i], NULL, philosopher, (void *) i);
    pthread_create(&tid[N], NULL, show_state, NULL);
    pthread_join(tid[N], NULL);
    exit(0);
}
```