A condition (short for "condition variable") is a synchronization device that allows threads to suspend execution and relinquish the processors until some predicate on shared data is satisfied. The basic operations on conditions are: signal the condition (when the predicate becomes true), and wait for the condition, suspending the thread execution until another thread signals the condition.

A condition variable must always be associated with a mutex, to avoid the race condition where a thread prepares to wait on a condition variable and another thread signals the condition just before the first thread actually waits on it.

`pthread_cond_init` initializes the condition variable `cond`, using the condition attributes specified in `cond_attr`. If no attributes are specified, or if `cond_attr` is `NULL`, then the default attributes are used. Variables of type `pthread_cond_t` can also be initialized statically, using the constant `PTHREAD_COND_INITIALIZER`.

`pthread_cond_signal` restarts one of the threads that are waiting on the condition variable `cond`. If no threads are waiting on `cond`, nothing happens. If several threads are waiting on `cond`, exactly one is restarted, but it is not specified which.

`pthread_cond_broadcast` restarts all the threads that are waiting on the condition variable `cond`. Nothing happens if no threads are waiting on `cond`.

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

Unlocking the mutex and suspending on the condition variable is done atomically. Thus, if all threads always acquire the mutex before signaling the condition, this guarantees that the condition cannot be signaled (and thus ignored) between the time a thread locks the mutex and the time it waits on the condition variable.

`pthread_cond_timedwait` atomically unlocks `mutex` and waits on `cond`, as `pthread_cond_wait` does, but it also bounds the duration of the wait. If `cond` has not been signaled within the amount of time specified by `abstime`, the mutex `mutex` is re-acquired and `pthread_cond_timedwait` returns the error `ETIMEDOUT`. The `abstime` parameter specifies an absolute time, with the same origin as `time` and `gettimeofday`: an `abstime` of 0 corresponds to 00:00:00 GMT, January 1, 1970.

`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.

The condition functions are not async-signal safe, and should not be called from a signal handler. Calling `pthread_cond_signal` or `pthread_cond_broadcast` from a signal handler may deadlock the calling thread.

All condition variable functions return 0 on success and a non-zero error code on error.

`pthread_cond_init` and `pthread_cond_signal` or `pthread_cond_broadcast` from a signal handler may deadlock the calling thread.

The `pthread_cond.wait` function returns the following error codes on error:

- **EINTR** - The condition variable was not signaled until the timeout specified by `abstime`
- **ETIMEDOUT** - The condition variable was not signaled until the timeout specified by `abstime`
- **BUSY** - Some threads are currently waiting on `cond`.

The `pthread_cond_destroy` function returns the following error code on error:

- **EBUSY** - Some threads are currently waiting on `cond`.

**SEE ALSO**

- `pthread_condattr_init(3)`, `pthread_mutex锁(3)`, `pthread_mutex_destroy(3)`, `pthread_mutex_unlock(3)`, `gettimeofday(2)`, `nanosleep(2)`.

**AUTHOR**

Xavier Leroy <Xavier.Leroy@inria.fr>
NAME

pthread_mutex_init, pthread_mutex_lock, pthread_mutex_trylock, pthread_mutex_unlock,
pthread_mutex_destroy — operations on mutexes

SYNOPSIS

#include <pthread.h>

pthread_mutex_t fastmutex = PTHREAD_MUTEX_INITIALIZER;

pthread_mutex_t recmutex = PTHREAD_RECURSIVE_MUTEX_INITIALIZER_NP;

pthread_mutex_t errcheckmutex = PTHREAD_ERRORCHECK_MUTEX_INITIALIZER_NP;

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

int pthread_mutex_lock(pthread_mutex_t *mutex);

int pthread_mutex_trylock(pthread_mutex_t *mutex);

int pthread_mutex_unlock(pthread_mutex_t *mutex);

int pthread_mutex_destroy(pthread_mutex_t *mutex);

DESCRIPTION

A mutex is a MUTual EXclusion device, and is 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), and locked (owned by one thread). A mutex can never be owned by two different threads simultaneously. A thread attempting to lock a mutex that is already locked by another thread is suspended until the owning thread unlocks the mutex first.

pthread_mutex_init initializes the mutex object pointed to by mutex according to the mutex attributes specified in mutexattr. If mutexattr is NULL, default attributes are used instead.

The LinuxThreads implementation supports only one mutex attribute, the mutex kind, which is either “fast”, “recursive”, or “error checking”. The kind of a mutex determines whether it can be locked again by a thread that already owns it. The default kind is “fast”. See pthread_mutexattr_init(3) for more information on mutex attributes.

Variables of type pthread_mutex_t can also be initialized statically, using the constants PTHREAD_MUTEX_INITIALIZER (for fast mutexes), PTHREAD_RECURSIVE_MUTEX_INITIALIZER_NP (for recursive mutexes), and PTHREAD_ERRORCHECK_MUTEX_INITIALIZER (for error checking mutexes).

pthread_mutex_lock locks the given mutex. If the mutex is currently unlocked, it becomes locked and owned by the calling thread, and pthread_mutex_lock returns immediately. If the mutex is already locked by another thread, pthread_mutex_lock suspends the calling thread until the mutex is unlocked.

If the mutex is already locked by the calling thread, the behavior of pthread_mutex_lock depends on the kind of the mutex. If the mutex is of the “fast” kind, the calling thread is suspended until the mutex is unlocked, thus effectively causing the calling thread to deadlock. If the mutex is of the “error checking” kind, pthread_mutex_lock returns immediately with the error code EDEADLK. If the mutex is of the “recursive” kind, 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

RETURN VALUE

pthread_mutex_init always returns 0. The other mutex functions return 0 on success and a non-zero error code on error.

ERRORS

The pthread_mutex_lock function returns the following error code on error:

EINVAL
the mutex has not been properly initialized.

EDEADLK
the mutex is already locked by the calling thread (“error checking” mutexes only).

The pthread_mutex_unlock function returns the following error code on error:

EINVAL
the mutex has not been properly initialized.

EPERM
the calling thread does not own the mutex (“error checking” mutexes only).

The pthread_mutex_destroy function returns the following error code on error:

EBUSY
the mutex is currently locked.

AUTHOR

Xavier Leroy <Xavier.Leroy@inria.fr>

SEE ALSO

pthread_mutexattr_init(3), pthread_mutexattr_setkind_np(3), pthread_cancel(3).