## 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** *errchkmutex* **= 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 attributes, 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_NP** (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 performed before the mutex returns to the unlocked state.

**pthread_mutex_trylock** behaves identically to **pthread_mutex_lock**, except that it does not block the calling thread if the mutex is already locked by another thread (or by the calling thread in the case of a ‘‘fast’’ mutex). Instead, **pthread_mutex_trylock** returns immediately with the error code **EBUSY**.

**pthread_mutex_unlock** unlocks the given mutex. The mutex is assumed to be locked and owned by the calling thread on entrance to **pthread_mutex_unlock**. If the mutex is of the ‘‘fast’’ kind, **pthread_mutex_unlock** always returns it to the unlocked state. If it is of the ‘‘recursive’’ kind, it decrements the locking count of the mutex (number of **pthread_mutex_lock** operations performed on it by the calling thread), and only when this count reaches zero is the mutex actually unlocked.

On ‘‘error checking’’ mutexes, **pthread_mutex_unlock** actually checks at run-time that the mutex is locked on entrance, and that it was locked by the same thread that is now calling **pthread_mutex_unlock**. If these conditions are not met, an error code is returned and the mutex remains unchanged. ‘‘Fast’’ and ‘‘recursive’’ mutexes perform no such checks, thus allowing a locked mutex to be unlocked by a thread other than its owner. This is non-portable behavior and must not be relied upon.

**pthread_mutex_destroy** destroys a mutex object, freeing the resources it might hold. The mutex must be unlocked on entrance. In the LinuxThreads implementation, no resources are associated with mutex objects, thus **pthread_mutex_destroy** actually does nothing except checking that the mutex is unlocked.

## CANCELLATION

None of the mutex functions is a cancellation point, not even **pthread_mutex_lock**, in spite of the fact that it can suspend a thread for arbitrary durations. This way, the status of mutexes at cancellation points is predictable, allowing cancellation handlers to unlock precisely those mutexes that need to be unlocked before the thread stops executing. Consequently, threads using deferred cancellation should never hold a mutex for extended periods of time.

## ASYNC-SIGNAL SAFETY

The mutex functions are not async-signal safe. What this means is that they should not be called from a signal handler. In particular, calling **pthread_mutex_lock** or **pthread_mutex_unlock** from a signal handler may deadlock the calling thread.

## 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_trylock** function returns the following error codes on error:

**EBUSY**
the mutex could not be acquired because it was currently locked.

**EINVAL**
the mutex has not been properly initialized.

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).

## EXAMPLE

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 by calls to **pthread_mutex_lock** and **pthread_mutex_unlock** as follows:

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