## NAME

**mutex**, **mtx_init**, **mtx_destroy**, **mtx_lock**, **mtx_lock_spin**, **mtx_lock_flags**, **mtx_lock_spin_flags**, **mtx_trylock**, **mtx_trylock_flags**, **mtx_trylock_spin**, **mtx_trylock_spin_flags**, **mtx_unlock**, **mtx_unlock_spin**, **mtx_unlock_flags**, **mtx_unlock_spin_flags**, **mtx_sleep**, **mtx_initialized**, **mtx_owned**, **mtx_recursed**, **mtx_assert**, **MTX_SYSINIT** **-** kernel synchronization primitives

## SYNOPSIS

**#include <sys/param.h>**
**#include <sys/lock.h>**
**#include <sys/mutex.h>**

*void*
**mtx_init**(*struct mtx *mutex*, *const char *name*, *const char *type*, *int opts*);

*void*
**mtx_destroy**(*struct mtx *mutex*);

*void*
**mtx_lock**(*struct mtx *mutex*);

*void*
**mtx_lock_spin**(*struct mtx *mutex*);

*void*
**mtx_lock_flags**(*struct mtx *mutex*, *int flags*);

*void*
**mtx_lock_spin_flags**(*struct mtx *mutex*, *int flags*);

*int*
**mtx_trylock**(*struct mtx *mutex*);

*int*
**mtx_trylock_flags**(*struct mtx *mutex*, *int flags*);

*int*
**mtx_trylock_spin**(*struct mtx *mutex*);

*int*
**mtx_trylock_spin_flags**(*struct mtx *mutex*, *int flags*);

*void*
**mtx_unlock**(*struct mtx *mutex*);

*void*
**mtx_unlock_spin**(*struct mtx *mutex*);

*void*
**mtx_unlock_flags**(*struct mtx *mutex*, *int flags*);

*void*
**mtx_unlock_spin_flags**(*struct mtx *mutex*, *int flags*);

*int*
**mtx_sleep**(*void *chan*, *struct mtx *mtx*, *int priority*, *const char *wmesg*, *int timo*);

*int*
**mtx_initialized**(*const struct mtx *mutex*);

*int*
**mtx_owned**(*const struct mtx *mutex*);

*int*
**mtx_recursed**(*const struct mtx *mutex*);

**options INVARIANTS**
**options INVARIANT_SUPPORT**
*void*
**mtx_assert**(*const struct mtx *mutex*, *int what*);

**#include <sys/kernel.h>**

**MTX_SYSINIT**(*name*, *struct mtx *mtx*, *const char *description*, *int opts*);

## DESCRIPTION

Mutexes are the most basic and primary method of thread synchronization. The major design considerations for mutexes are:

1. Acquiring and releasing uncontested mutexes should be as cheap as possible.

2. They must have the information and storage space to support priority propagation.

3. A thread must be able to recursively acquire a mutex, provided that the mutex is initialized to support recursion.

There are currently two flavors of mutexes, those that context switch when they block and those that do not.

By default, MTX_DEF mutexes will context switch when they are already held. As an optimization, they may spin for some amount of time before context switching. It is important to remember that since a thread may be preempted at any time, the possible context switch introduced by acquiring a mutex is guaranteed to not break anything that is not already broken.

Mutexes which do not context switch are MTX_SPIN mutexes. These should only be used to protect data shared with primary interrupt code. This includes interrupt filters and low level scheduling code. In all architectures both acquiring and releasing of a uncontested spin mutex is more expensive than the same operation on a non-spin mutex. In order to protect an interrupt service routine from blocking against itself all interrupts are either blocked or deferred on a processor while holding a spin lock. It is permissible to hold multiple spin mutexes.

Once a spin mutex has been acquired it is not permissible to acquire a blocking mutex.

The storage needed to implement a mutex is provided by a *struct mtx*. In general this should be treated as an opaque object and referenced only with the mutex primitives.

The **mtx_init**() function must be used to initialize a mutex before it can be passed to any of the other mutex functions. The *name* option is used to identify the lock in debugging output etc. The *type* option is used by the witness code to classify a mutex when doing checks of lock ordering. If *type* is NULL, *name* is used in its place. The pointer passed in as *name* and *type* is saved rather than the data it points to. The data pointed to must remain stable until the mutex is destroyed. The *opts* argument is used to set the type of mutex. It may contain either MTX_DEF or MTX_SPIN but not both. If the kernel has been compiled with **option INVARIANTS**, **mtx_init**() will assert that the *mutex* has not been initialized multiple times without intervening calls to **mtx_destroy**() unless the MTX_NEW option is specified. See below for additional initialization options.

The **mtx_lock**() function acquires a MTX_DEF mutual exclusion lock on behalf of the currently running kernel thread. If another kernel thread is holding the mutex, the caller will be disconnected from the CPU until the mutex is available (i.e., it will block).

The **mtx_lock_spin**() function acquires a MTX_SPIN mutual exclusion lock on behalf of the currently running kernel thread. If another kernel thread is holding the mutex, the caller will spin until the mutex becomes available. Interrupts are disabled during the spin and remain disabled following the acquiring of the lock.

It is possible for the same thread to recursively acquire a mutex with no ill effects, provided that the MTX_RECURSE bit was passed to **mtx_init**() during the initialization of the mutex.

The **mtx_lock_flags**() and **mtx_lock_spin_flags**() functions acquire a MTX_DEF or MTX_SPIN lock, respectively, and also accept a *flags* argument. In both cases, the only flags presently available for lock acquires are MTX_QUIET and MTX_RECURSE. If the MTX_QUIET bit is turned on in the *flags* argument, then if KTR_LOCK tracing is being done, it will be silenced during the lock acquire. If the MTX_RECURSE bit is turned on in the *flags* argument, then the mutex can be acquired recursively.

The **mtx_trylock**() and **mtx_trylock_spin**() functions attempt to acquire a MTX_DEF or MTX_SPIN mutex, respectively, pointed to by *mutex*. If the mutex cannot be immediately acquired, the functions will return 0, otherwise the mutex will be acquired and a non-zero value will be returned.

The **mtx_trylock_flags**() and **mtx_trylock_spin_flags**() functions have the same behavior as **mtx_trylock**() and **mtx_trylock_spin**() respectively, but should be used when the caller desires to pass in a *flags* value. Presently, the only valid value in the **mtx_trylock**() and **mtx_trylock_spin**() cases is MTX_QUIET, and its effects are identical to those described for **mtx_lock**() above.

The **mtx_unlock**() function releases a MTX_DEF mutual exclusion lock. The current thread may be preempted if a higher priority thread is waiting for the mutex.

The **mtx_unlock_spin**() function releases a MTX_SPIN mutual exclusion lock.

The **mtx_unlock_flags**() and **mtx_unlock_spin_flags**() functions behave in exactly the same way as do the standard mutex unlock routines above, while also allowing a *flags* argument which may specify MTX_QUIET. The behavior of MTX_QUIET is identical to its behavior in the mutex lock routines.

The **mtx_destroy**() function is used to destroy *mutex* so the data associated with it may be freed or otherwise overwritten. Any mutex which is destroyed must previously have been initialized with **mtx_init**(). It is permissible to have a single hold count on a mutex when it is destroyed. It is not permissible to hold the mutex recursively, or have another thread blocked on the mutex when it is destroyed.

The **mtx_sleep**() function is used to atomically release *mtx* while waiting for an event. For more details on the parameters to this function, see sleep(9).

The **mtx_initialized**() function returns non-zero if *mutex* has been initialized and zero otherwise.

The **mtx_owned**() function returns non-zero if the current thread holds *mutex*. If the current thread does not hold *mutex* zero is returned.

The **mtx_recursed**() function returns non-zero if the *mutex* is recursed. This check should only be made if the running thread already owns *mutex*.

The **mtx_assert**() function allows assertions specified in *what* to be made about *mutex*. If the assertions are not true and the kernel is compiled with **options INVARIANTS** and **options INVARIANT_SUPPORT**, the kernel will panic. Currently the following assertions are supported:

| | |
|---|---|
| MA_OWNED | Assert that the current thread holds the mutex pointed to by the first argument. |
| MA_NOTOWNED | Assert that the current thread does not hold the mutex pointed to by the first argument. |
| MA_RECURSED | Assert that the current thread has recursed on the mutex pointed to by the first argument. This assertion is only valid in conjunction with MA_OWNED. |
| MA_NOTRECURSED | Assert that the current thread has not recursed on the mutex pointed to by the first argument. This assertion is only valid in conjunction with MA_OWNED. |

The **MTX_SYSINIT**() macro is used to generate a call to the **mtx_sysinit**() routine at system startup in order to initialize a given mutex lock. The parameters are the same as **mtx_init**() but with an additional argument, *name*, that is used in generating unique variable names for the related structures associated with the lock and the sysinit routine.

### The Default Mutex Type

Most kernel code should use the default lock type, MTX_DEF. The default lock type will allow the thread to be disconnected from the CPU if the lock is already held by another thread. The implementation may treat the lock as a short term spin lock under some circumstances. However, it is always safe to use these forms of locks in an interrupt thread without fear of deadlock against an interrupted thread on the same CPU.

### The Spin Mutex Type

A MTX_SPIN mutex will not relinquish the CPU when it cannot immediately get the requested lock, but will loop, waiting for the mutex to be released by another CPU. This could result in deadlock if another thread interrupted the thread which held a mutex and then tried to acquire the mutex. For this reason spin locks disable all interrupts on the local CPU.

Spin locks are fairly specialized locks that are intended to be held for very short periods of time. Their primary purpose is to protect portions of the code that implement other synchronization primitives such as default mutexes, thread scheduling, and interrupt threads.

## Initialization Options

The options passed in the *opts* argument of **mtx_init**() specify the mutex type. One of the MTX_DEF or MTX_SPIN options is required and only one of those two options may be specified. The possibilities are:

| | |
|---|---|
| MTX_DEF | Default mutexes will always allow the current thread to be suspended to avoid deadlock conditions against interrupt threads. The implementation of this lock type may spin for a while before suspending the current thread. |
| MTX_SPIN | Spin mutexes will never relinquish the CPU. All interrupts are disabled on the local CPU while any spin lock is held. |
| MTX_RECURSE | Specifies that the initialized mutex is allowed to recurse. This bit must be present if the mutex is permitted to recurse.<br><br>Note that neither **mtx_trylock**() nor **mtx_trylock_spin**() support recursion; that is, attempting to acquire an already-owned mutex fails. |
| MTX_QUIET | Do not log any mutex operations for this lock. |
| MTX_NOWITNESS | Instruct witness(4) to ignore this lock. |
| MTX_DUPOK | Witness should not log messages about duplicate locks being acquired. |
| MTX_NOPROFILE | Do not profile this lock. |
| MTX_NEW | Do not check for double-init. |

## Lock and Unlock Flags

The flags passed to the **mtx_lock_flags**(), **mtx_lock_spin_flags**(), **mtx_unlock_flags**(), and **mtx_unlock_spin_flags**() functions provide some basic options to the caller, and are often used only under special circumstances to modify lock or unlock behavior. Standard locking and unlocking should be performed with the **mtx_lock**(), **mtx_lock_spin**(), **mtx_unlock**(), and **mtx_unlock_spin**() functions. Only if a flag is required should the corresponding flags-accepting routines be used.

Options that modify mutex behavior:

| | |
|---|---|
| MTX_QUIET | This option is used to quiet logging messages during individual mutex operations. This can be used to trim superfluous logging messages for debugging purposes. |

### Giant

If *Giant* must be acquired, it must be acquired prior to acquiring other mutexes. Put another way: it is impossible to acquire *Giant* non-recursively while holding another mutex. It is possible to acquire other mutexes while holding *Giant*, and it is possible to acquire *Giant* recursively while holding other mutexes.

### Sleeping

Sleeping while holding a mutex (except for *Giant*) is never safe and should be avoided. There are numerous assertions which will fail if this is attempted.

### Functions Which Access Memory in Userspace

No mutexes should be held (except for *Giant*) across functions which access memory in userspace, such as copyin(9), copyout(9), uiomove(9), fuword(9), etc. No locks are needed when calling these functions.

## SEE ALSO

condvar(9), LOCK_PROFILING(9), locking(9), mtx_pool(9), panic(9), rwlock(9), sema(9), sleep(9), sx(9)

## HISTORY

These functions appeared in BSD/OS 4.1 and FreeBSD 5.0. The **mtx_trylock_spin**() function was added in FreeBSD 11.1.