NAME

lockinit, **lockdestroy**, **lockmgr**, **lockmgr_args**, **lockmgr_args_rw**, **lockmgr_disown**, **lockmgr_printinfo**, **lockmgr_recursed**, **lockmgr_rw**, **lockstatus**, **lockmgr_assert** - lockmgr family of functions

SYNOPSIS

#include <sys/types.h>
#include <sys/lock.h>
#include <sys/lockmgr.h>

void
lockinit(*struct lock *lkp*, *int prio*, *const char *wmesg*, *int timo*, *int flags*);

void
lockdestroy(*struct lock *lkp*);

int
lockmgr(*struct lock *lkp*, *u_int flags*, *struct mtx *ilk*);

int
lockmgr_args(*struct lock *lkp*, *u_int flags*, *struct mtx *ilk*, *const char *wmesg*, *int prio*, *int timo*);

int
lockmgr_args_rw(*struct lock *lkp*, *u_int flags*, *struct rwlock *ilk*, *const char *wmesg*, *int prio*, *int timo*);

void
lockmgr_disown(*struct lock *lkp*);

void
lockmgr_printinfo(*const struct lock *lkp*);

int
lockmgr_recursed(*const struct lock *lkp*);

int
lockmgr_rw(*struct lock *lkp*, *u_int flags*, *struct rwlock *ilk*);

int
lockstatus(*const struct lock *lkp*);

options INVARIANTS

options INVARIANT_SUPPORT
void
lockmgr_assert(*const struct lock *lkp, int what*);

DESCRIPTION

The **lockinit**() function is used to initialize a lock. It must be called before any operation can be performed on a lock. Its arguments are:

lkp	A pointer to the lock to initialize.
prio	The priority passed to sleep(9).
wmesg	The lock message. This is used for both debugging output and sleep(9).
timo	The timeout value passed to sleep(9).
flags	The flags the lock is to be initialized with:

LK_CANRECURSE	Allow recursive exclusive locks.
LK_NOPROFILE	Disable lock profiling for this lock.
LK_NOSHARE	Allow exclusive locks only.
LK_NOWITNESS	Instruct witness(4) to ignore this lock.
LK_NODUP	witness(4) should log messages about duplicate locks being acquired.
LK_QUIET	Disable ktr(4) logging for this lock.
LK_TIMELOCK	Use *timo* during a sleep; otherwise, 0 is used.

The **lockdestroy**() function is used to destroy a lock, and while it is called in a number of places in the kernel, it currently does nothing.

The **lockmgr**() and **lockmgr_rw**() functions handle general locking functionality within the kernel, including support for shared and exclusive locks, and recursion. **lockmgr**() and **lockmgr_rw**() are also able to upgrade and downgrade locks.

Their arguments are:

lkp A pointer to the lock to manipulate.

flags Flags indicating what action is to be taken.

LK_SHARED	Acquire a shared lock. If an exclusive lock is currently held, EDEADLK will be returned.
LK_EXCLUSIVE	Acquire an exclusive lock. If an exclusive lock is already held, and LK_CANRECURSE is not set, the system will panic(9).
LK_DOWNGRADE	Downgrade exclusive lock to a shared lock. Downgrading a shared lock is not permitted. If an exclusive lock has been recursed, the system will panic(9).
LK_UPGRADE	Upgrade a shared lock to an exclusive lock. If this call fails, the shared lock is lost, even if the LK_NOWAIT flag is specified. During the upgrade, the shared lock could be temporarily dropped. Attempts to upgrade an exclusive lock will cause a panic(9).
LK_TRYUPGRADE	Try to upgrade a shared lock to an exclusive lock. The failure to upgrade does not result in the dropping of the shared lock ownership.
LK_RELEASE	Release the lock. Releasing a lock that is not held can cause a panic(9).
LK_DRAIN	Wait for all activity on the lock to end, then mark it decommissioned. This is used before freeing a lock that is part of a piece of memory that is about to be freed. (As documented in <*sys/lockmgr.h*>.)
LK_SLEEPFAIL	Fail if operation has slept.
LK_NOWAIT	Do not allow the call to sleep. This can be used to test the lock.
LK_NOWITNESS	Skip the witness(4) checks for this instance.
LK_CANRECURSE	Allow recursion on an exclusive lock. For every lock there must be a release.
LK_INTERLOCK	Unlock the interlock (which should be locked already).
LK_NODDLKTREAT	Normally, **lockmgr**() postpones serving further shared requests for shared-

locked lock if there is exclusive waiter, to avoid exclusive lock starvation. But, if the thread requesting the shared lock already owns a shared lockmgr lock, the request is granted even in presence of the parallel exclusive lock request, which is done to avoid deadlocks with recursive shared acquisition.

The LK_NODDLKTREAT flag can only be used by code which requests shared non-recursive lock. The flag allows exclusive requests to preempt the current shared request even if the current thread owns shared locks. This is safe since shared lock is guaranteed to not recurse, and is used when thread is known to held unrelated shared locks, to not cause unnecessary starvation. An example is vp locking in VFS lookup(9), when dvp is already locked.

ilk An interlock mutex for controlling group access to the lock. If LK_INTERLOCK is specified, **lockmgr**() and **lockmgr_rw**() assume *ilk* is currently owned and not recursed, and will return it unlocked. See mtx_assert(9).

The **lockmgr_args**() and **lockmgr_args_rw**() function work like **lockmgr**() and **lockmgr_rw**() but accepting a *wmesg*, *timo* and *prio* on a per-instance basis. The specified values will override the default ones, but this can still be used passing, respectively, LK_WMESG_DEFAULT, LK_PRIO_DEFAULT and LK_TIMO_DEFAULT.

The **lockmgr_disown**() function switches the owner from the current thread to be LK_KERNPROC, if the lock is already held.

The **lockmgr_printinfo**() function prints debugging information about the lock. It is used primarily by VOP_PRINT(9) functions.

The **lockmgr_recursed**() function returns true if the lock is recursed, 0 otherwise.

The **lockstatus**() function returns the status of the lock in relation to the current thread.

When compiled with **options INVARIANTS** and **options INVARIANT_SUPPORT**, the **lockmgr_assert**() function tests *lkp* for the assertions specified in *what*, and panics if they are not met. One of the following assertions must be specified:

KA_LOCKED Assert that the current thread has either a shared or an exclusive lock on the *lkp* lock pointed to by the first argument.

KA_SLOCKED	Assert that the current thread has a shared lock on the *lkp* lock pointed to by the first argument.
KA_XLOCKED	Assert that the current thread has an exclusive lock on the *lkp* lock pointed to by the first argument.
KA_UNLOCKED	Assert that the current thread has no lock on the *lkp* lock pointed to by the first argument.

In addition, one of the following optional assertions can be used with either an KA_LOCKED, KA_SLOCKED, or KA_XLOCKED assertion:

KA_RECURSED	Assert that the current thread has a recursed lock on *lkp*.
KA_NOTRECURSED	Assert that the current thread does not have a recursed lock on *lkp*.

RETURN VALUES

The **lockmgr**() and **lockmgr_rw**() functions return 0 on success and non-zero on failure.

The **lockstatus**() function returns:

LK_EXCLUSIVE	An exclusive lock is held by the current thread.
LK_EXCLOTHER	An exclusive lock is held by someone other than the current thread.
LK_SHARED	A shared lock is held.
0	The lock is not held by anyone.

ERRORS

lockmgr() and **lockmgr_rw**() fail if:

[EBUSY]	LK_FORCEUPGRADE was requested and another thread had already requested a lock upgrade.
[EBUSY]	LK_NOWAIT was set, and a sleep would have been required, or LK_TRYUPGRADE operation was not able to upgrade the lock.
[ENOLCK]	LK_SLEEPFAIL was set and **lockmgr**() or **lockmgr_rw**() did sleep.

[EINTR]	PCATCH was set in the lock priority, and a signal was delivered during a sleep. Note the ERESTART error below.
[ERESTART]	PCATCH was set in the lock priority, a signal was delivered during a sleep, and the system call is to be restarted.
[EWOULDBLOCK]	a non-zero timeout was given, and the timeout expired.

LOCKS

If LK_INTERLOCK is passed in the *flags* argument to **lockmgr**() or **lockmgr_rw**(), the *ilk* must be held prior to calling **lockmgr**() or **lockmgr_rw**(), and will be returned unlocked.

Upgrade attempts that fail result in the loss of the lock that is currently held. Also, it is invalid to upgrade an exclusive lock, and a panic(9) will be the result of trying.

SEE ALSO

condvar(9), locking(9), mtx_assert(9), mutex(9), panic(9), rwlock(9), sleep(9), sx(9), VOP_PRINT(9)

AUTHORS

This manual page was written by Chad David <*davidc@acns.ab.ca*>.