NAME

sx, **sx_init**, **sx_init_flags**, **sx_destroy**, **sx_slock**, **sx_xlock**, **sx_slock_sig**, **sx_xlock_sig**, **sx_try_slock**, **sx_try_xlock**, **sx_sunlock**, **sx_xunlock**, **sx_unlock**, **sx_try_upgrade**, **sx_downgrade**, **sx_sleep**, **sx_xholder**, **sx_xlocked**, **sx_assert**, **SX_SYSINIT**, **SX_SYSINIT_FLAGS** - kernel shared/exclusive lock

SYNOPSIS

#include <sys/param.h>
#include <sys/lock.h>
#include <sys/sx.h>

void
sx_init(*struct sx *sx, const char *description*);

void
sx_init_flags(*struct sx *sx, const char *description, int opts*);

void
sx_destroy(*struct sx *sx*);

void
sx_slock(*struct sx *sx*);

void
sx_xlock(*struct sx *sx*);

int
sx_slock_sig(*struct sx *sx*);

int
sx_xlock_sig(*struct sx *sx*);

int
sx_try_slock(*struct sx *sx*);

int
sx_try_xlock(*struct sx *sx*);

void
sx_sunlock(*struct sx *sx*);

void
sx_xunlock(*struct sx *sx*);

void
sx_unlock(*struct sx *sx*);

int
sx_try_upgrade(*struct sx *sx*);

void
sx_downgrade(*struct sx *sx*);

int
sx_sleep(*void *chan, struct sx *sx, int priority, const char *wmesg, int timo*);

*struct thread **
sx_xholder(*struct sx *sx*);

int
sx_xlocked(*const struct sx *sx*);

options INVARIANTS
options INVARIANT_SUPPORT
void
sx_assert(*const struct sx *sx, int what*);

#include <sys/kernel.h>

SX_SYSINIT(*name, struct sx *sx, const char *desc*);

SX_SYSINIT_FLAGS(*name, struct sx *sx, const char *desc, int flags*);

DESCRIPTION

Shared/exclusive locks are used to protect data that are read far more often than they are written. Shared/exclusive locks do not implement priority propagation like mutexes and reader/writer locks to prevent priority inversions, so shared/exclusive locks should be used prudently.

Shared/exclusive locks are created with either **sx_init**() or **sx_init_flags**() where *sx* is a pointer to space for a *struct sx*, and *description* is a pointer to a null-terminated character string that describes the shared/exclusive lock. The *opts* argument to **sx_init_flags**() specifies a set of optional flags to alter the

behavior of *sx*. It contains one or more of the following flags:

SX_DUPOK Witness should not log messages about duplicate locks being acquired.

SX_NOWITNESS Instruct witness(4) to ignore this lock.

SX_NOPROFILE Do not profile this lock.

SX_RECURSE Allow threads to recursively acquire exclusive locks for *sx*.

SX_QUIET Do not log any operations for this lock via ktr(4).

SX_NEW If the kernel has been compiled with **options INVARIANTS**, **sx_init**() will assert that the *sx* has not been initialized multiple times without intervening calls to **sx_destroy**() unless this option is specified.

Shared/exclusive locks are destroyed with **sx_destroy**(). The lock *sx* must not be locked by any thread when it is destroyed.

Threads acquire and release a shared lock by calling **sx_slock**(), **sx_slock_sig**() or **sx_try_slock**() and **sx_sunlock**() or **sx_unlock**(). Threads acquire and release an exclusive lock by calling **sx_xlock**(), **sx_xlock_sig**() or **sx_try_xlock**() and **sx_xunlock**() or **sx_unlock**(). A thread can attempt to upgrade a currently held shared lock to an exclusive lock by calling **sx_try_upgrade**(). A thread that has an exclusive lock can downgrade it to a shared lock by calling **sx_downgrade**().

sx_try_slock() and **sx_try_xlock**() will return 0 if the shared/exclusive lock cannot be acquired immediately; otherwise the shared/exclusive lock will be acquired and a non-zero value will be returned.

sx_try_upgrade() will return 0 if the shared lock cannot be upgraded to an exclusive lock immediately; otherwise the exclusive lock will be acquired and a non-zero value will be returned.

sx_slock_sig() and **sx_xlock_sig**() do the same as their normal versions but performing an interruptible sleep. They return a non-zero value if the sleep has been interrupted by a signal or an interrupt, otherwise 0.

A thread can atomically release a shared/exclusive lock while waiting for an event by calling **sx_sleep**(). For more details on the parameters to this function, see sleep(9).

When compiled with **options INVARIANTS** and **options INVARIANT_SUPPORT**, the **sx_assert**() function tests *sx* for the assertions specified in *what*, and panics if they are not met. One of the

following assertions must be specified:

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

SA_SLOCKED Assert that the current thread has a shared lock on the *sx* lock pointed to by the first argument.

SA_XLOCKED Assert that the current thread has an exclusive lock on the *sx* lock pointed to by the first argument.

SA_UNLOCKED Assert that the current thread has no lock on the *sx* lock pointed to by the first argument.

In addition, one of the following optional assertions may be included with either an SA_LOCKED, SA_SLOCKED, or SA_XLOCKED assertion:

SA_RECURSED Assert that the current thread has a recursed lock on *sx*.

SA_NOTRECURSED Assert that the current thread does not have a recursed lock on *sx*.

sx_xholder() will return a pointer to the thread which currently holds an exclusive lock on *sx*. If no thread holds an exclusive lock on *sx*, then NULL is returned instead.

sx_xlocked() will return non-zero if the current thread holds the exclusive lock; otherwise, it will return zero.

For ease of programming, **sx_unlock**() is provided as a macro frontend to the respective functions, **sx_sunlock**() and **sx_xunlock**(). Algorithms that are aware of what state the lock is in should use either of the two specific functions for a minor performance benefit.

The **SX_SYSINIT**() macro is used to generate a call to the **sx_sysinit**() routine at system startup in order to initialize a given *sx* lock. The parameters are the same as **sx_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 **SX_SYSINIT_FLAGS**() macro can similarly be used to initialize a given *sx* lock using **sx_init_flags**().

A thread may not hold both a shared lock and an exclusive lock on the same lock simultaneously; attempting to do so will result in deadlock.

CONTEXT

A thread may hold a shared or exclusive lock on an **sx** lock while sleeping. As a result, an **sx** lock may not be acquired while holding a mutex. Otherwise, if one thread slept while holding an **sx** lock while another thread blocked on the same **sx** lock after acquiring a mutex, then the second thread would effectively end up sleeping while holding a mutex, which is not allowed.

SEE ALSO

lock(9), locking(9), mutex(9), panic(9), rwlock(9), sema(9)

BUGS

A kernel without WITNESS cannot assert whether the current thread does or does not hold a shared lock. SA_LOCKED and SA_SLOCKED can only assert that *any* thread holds a shared lock. They cannot ensure that the current thread holds a shared lock. Further, SA_UNLOCKED can only assert that the current thread does not hold an exclusive lock.