NAME

rmlock, **rm_init**, **rm_init_flags**, **rm_destroy**, **rm_rlock**, **rm_try_rlock**, **rm_wlock**, **rm_runlock**, **rm_wunlock**, **rm_wowned**, **rm_sleep**, **rm_assert**, **RM_SYSINIT**, **RM_SYSINIT_FLAGS**, **rms_init**, **rms_destroy**, **rms_rlock**, **rms_wlock**, **rms_runlock**, **rms_wunlock** **-** kernel reader/writer lock optimized for read-mostly access patterns

SYNOPSIS

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

void
rm_init(*struct rmlock *rm, const char *name*);

void
rm_init_flags(*struct rmlock *rm, const char *name, int opts*);

void
rm_destroy(*struct rmlock *rm*);

void
rm_rlock(*struct rmlock *rm, struct rm_priotracker* tracker*);

int
rm_try_rlock(*struct rmlock *rm, struct rm_priotracker* tracker*);

void
rm_wlock(*struct rmlock *rm*);

void
rm_runlock(*struct rmlock *rm, struct rm_priotracker* tracker*);

void
rm_wunlock(*struct rmlock *rm*);

int
rm_wowned(*const struct rmlock *rm*);

int
rm_sleep(*void *wchan, struct rmlock *rm, int priority, const char *wmesg, int timo*);

options INVARIANTS
options INVARIANT_SUPPORT
void
rm_assert(*struct rmlock *rm*, *int what*);

#include <sys/kernel.h>

RM_SYSINIT(*name*, *struct rmlock *rm*, *const char *desc*);

RM_SYSINIT_FLAGS(*name*, *struct rmlock *rm*, *const char *desc*, *int flags*);

void
rms_init(*struct rmslock *rms*, *const char *name*);

void
rms_destroy(*struct rmslock *rms*);

void
rms_rlock(*struct rmslock *rms*);

void
rms_wlock(*struct rmslock *rms*);

void
rms_runlock(*struct rmslock *rms*);

void
rms_wunlock(*struct rmslock *rms*);

DESCRIPTION

Read-mostly locks allow shared access to protected data by multiple threads, or exclusive access by a single thread. The threads with shared access are known as *readers* since they only read the protected data. A thread with exclusive access is known as a *writer* since it can modify protected data.

Read-mostly locks are designed to be efficient for locks almost exclusively used as reader locks and as such should be used for protecting data that rarely changes. Acquiring an exclusive lock after the lock has been locked for shared access is an expensive operation.

Normal read-mostly locks are similar to rwlock(9) locks and follow the same lock ordering rules as rwlock(9) locks. Read-mostly locks have full priority propagation like mutexes. Unlike rwlock(9), read-mostly locks propagate priority to both readers and writers. This is implemented via the *rm_priotracker* structure argument supplied to **rm_rlock**() and **rm_runlock**(). Readers can recurse if the lock is initialized with the RM_RECURSE option; however, writers are never allowed to recurse.

Sleeping for writers can be allowed by passing RM_SLEEPABLE to **rm_init_flags**(). It changes lock ordering rules to the same as for sx(9) locks. They do not propagate priority to writers, but they do propagate priority to readers. Note that readers are not permitted to sleep regardless of the flag.

Sleepable read-mostly locks (created with **rms_init**()) allow sleeping for both readers and writers, but don’t do priority propagation for either. They follow sx(9) lock ordering.

Macros and Functions

rm_init(*struct rmlock *rm, const char *name*)

Initialize the read-mostly lock *rm*. The *name* description is used solely for debugging purposes. This function must be called before any other operations on the lock.

rm_init_flags(*struct rmlock *rm, const char *name, int opts*)

Similar to **rm_init**(), initialize the read-mostly lock *rm* with a set of optional flags. The *opts* arguments contains one or more of the following flags:

RM_NOWITNESS	Instruct witness(4) to ignore this lock.
RM_RECURSE	Allow threads to recursively acquire shared locks for *rm*.
RM_SLEEPABLE	Create a sleepable read-mostly lock.
RM_NEW	If the kernel has been compiled with **option INVARIANTS**, **rm_init_flags**() will assert that the *rm* has not been initialized multiple times without intervening calls to **rm_destroy**() unless this option is specified.
RM_DUPOK	witness(4) should not log messages about duplicate locks being acquired.

rm_rlock(*struct rmlock *rm, struct rm_priotracker* tracker*)

Lock *rm* as a reader using *tracker* to track read owners of a lock for priority propagation. This data structure is only used internally by **rmlock** and must persist until **rm_runlock**() has been called. This data structure can be allocated on the stack since readers cannot sleep. If any thread holds this lock exclusively, the current thread blocks, and its priority is propagated to the exclusive holder. If the lock was initialized with the RM_RECURSE option the **rm_rlock**() function can be called when the current thread has already acquired reader access on *rm*.

rm_try_rlock(*struct rmlock *rm*, *struct rm_priotracker* tracker*)

Try to lock *rm* as a reader. **rm_try_rlock**() will return 0 if the lock cannot be acquired immediately; otherwise, the lock will be acquired and a non-zero value will be returned. Note that **rm_try_rlock**() may fail even while the lock is not currently held by a writer. If the lock was initialized with the RM_RECURSE option, **rm_try_rlock**() will succeed if the current thread has already acquired reader access.

rm_wlock(*struct rmlock *rm*)

Lock *rm* as a writer. If there are any shared owners of the lock, the current thread blocks. The **rm_wlock**() function cannot be called recursively.

rm_runlock(*struct rmlock *rm*, *struct rm_priotracker* tracker*)

This function releases a shared lock previously acquired by **rm_rlock**(). The *tracker* argument must match the *tracker* argument used for acquiring the shared lock

rm_wunlock(*struct rmlock *rm*)

This function releases an exclusive lock previously acquired by **rm_wlock**().

rm_destroy(*struct rmlock *rm*)

This functions destroys a lock previously initialized with **rm_init**(). The *rm* lock must be unlocked.

rm_wowned(*const struct rmlock *rm*)

This function returns a non-zero value if the current thread owns an exclusive lock on *rm*.

rm_sleep(*void *wchan*, *struct rmlock *rm*, *int priority*, *const char *wmesg*, *int timo*)

This function atomically releases *rm* while waiting for an event. The *rm* lock must be exclusively locked. For more details on the parameters to this function, see sleep(9).

rm_assert(*struct rmlock *rm*, *int what*)

This function asserts that the *rm* lock is in the state specified by *what*. 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 base assertions are supported:

RA_LOCKED	Assert that current thread holds either a shared or exclusive lock of *rm*.
RA_RLOCKED	Assert that current thread holds a shared lock of *rm*.
RA_WLOCKED	Assert that current thread holds an exclusive lock of *rm*.

RA_UNLOCKED	Assert that current thread holds neither a shared nor exclusive lock of *rm*.

In addition, one of the following optional flags may be specified with RA_LOCKED, RA_RLOCKED, or RA_WLOCKED:

RA_RECURSED	Assert that the current thread holds a recursive lock of *rm*.
RA_NOTRECURSED	Assert that the current thread does not hold a recursive lock of *rm*.

rms_init(*struct rmslock *rms*, *const char *name*)

Initialize the sleepable read-mostly lock *rms*. The *name* description is used as *wmesg* parameter to the msleep(9) routine. This function must be called before any other operations on the lock.

rms_rlock(*struct rmlock *rm*)

Lock *rms* as a reader. If any thread holds this lock exclusively, the current thread blocks.

rms_wlock(*struct rmslock *rms*)

Lock *rms* as a writer. If the lock is already taken, the current thread blocks. The **rms_wlock**() function cannot be called recursively.

rms_runlock(*struct rmslock *rms*)

This function releases a shared lock previously acquired by **rms_rlock**().

rms_wunlock(*struct rmslock *rms*)

This function releases an exclusive lock previously acquired by **rms_wlock**().

rms_destroy(*struct rmslock *rms*)

This functions destroys a lock previously initialized with **rms_init**(). The *rms* lock must be unlocked.

SEE ALSO

locking(9), mutex(9), panic(9), rwlock(9), sema(9), sleep(9), sx(9)

HISTORY

These functions appeared in FreeBSD 7.0.

AUTHORS

The **rmlock** facility was written by Stephan Uphoff. This manual page was written by Gleb Smirnoff for rwlock and modified to reflect rmlock by Stephan Uphoff.

BUGS

The **rmlock** implementation is currently not optimized for single processor systems.

rm_try_rlock() can fail transiently even when there is no writer, while another reader updates the state on the local CPU.

The **rmlock** implementation uses a single per CPU list shared by all rmlocks in the system. If rmlocks become popular, hashing to multiple per CPU queues may be needed to speed up the writer lock process.