NAME

locking - kernel synchronization primitives

DESCRIPTION

The *FreeBSD* kernel is written to run across multiple CPUs and as such provides several different synchronization primitives to allow developers to safely access and manipulate many data types.

Mutexes

Mutexes (also called "blocking mutexes") are the most commonly used synchronization primitive in the kernel. A thread acquires (locks) a mutex before accessing data shared with other threads (including interrupt threads), and releases (unlocks) it afterwards. If the mutex cannot be acquired, the thread requesting it will wait. Mutexes are adaptive by default, meaning that if the owner of a contended mutex is currently running on another CPU, then a thread attempting to acquire the mutex will spin rather than yielding the processor. Mutexes fully support priority propagation.

See mutex(9) for details.

Spin Mutexes

Spin mutexes are a variation of basic mutexes; the main difference between the two is that spin mutexes never block. Instead, they spin while waiting for the lock to be released. To avoid deadlock, a thread that holds a spin mutex must never yield its CPU. Unlike ordinary mutexes, spin mutexes disable interrupts when acquired. Since disabling interrupts can be expensive, they are generally slower to acquire and release. Spin mutexes should be used only when absolutely necessary, e.g. to protect data shared with interrupt filter code (see bus_setup_intr(9) for details), or for scheduler internals.

Mutex Pools

With most synchronization primitives, such as mutexes, the programmer must provide memory to hold the primitive. For example, a mutex may be embedded inside the structure it protects. Mutex pools provide a preallocated set of mutexes to avoid this requirement. Note that mutexes from a pool may only be used as leaf locks.

See mtx_pool(9) for details.

Reader/Writer Locks

Reader/writer 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 should only read the protected data. A thread with exclusive access is known as a *writer* since it may modify protected data.

Reader/writer locks can be treated as mutexes (see above and mutex(9)) with shared/exclusive semantics. Reader/writer locks support priority propagation like mutexes, but priority is propagated

only to an exclusive holder. This limitation comes from the fact that shared owners are anonymous.

See rwlock(9) for details.

Read-Mostly Locks

Read-mostly locks are similar to *reader/writer* locks but optimized for very infrequent write locking. *Read-mostly* locks implement full priority propagation by tracking shared owners using a caller-supplied *tracker* data structure.

See rmlock(9) for details.

Sleepable Read-Mostly Locks

Sleepable read-mostly locks are a variation on read-mostly locks. Threads holding an exclusive lock may sleep, but threads holding a shared lock may not. Priority is propagated to shared owners but not to exclusive owners.

Shared/exclusive locks

Shared/exclusive locks are similar to reader/writer locks; the main difference between them is that shared/exclusive locks may be held during unbounded sleep. Acquiring a contested shared/exclusive lock can perform an unbounded sleep. These locks do not support priority propagation.

See sx(9) for details.

Lockmanager locks

Lockmanager locks are sleepable shared/exclusive locks used mostly in VFS(9) (as a vnode(9) lock) and in the buffer cache (BUF_LOCK(9)). They have features other lock types do not have such as sleep timeouts, blocking upgrades, writer starvation avoidance, draining, and an interlock mutex, but this makes them complicated both to use and to implement; for this reason, they should be avoided.

See lock(9) for details.

Non-blocking synchronization

The kernel has two facilities, epoch(9) and smr(9), which can be used to provide read-only access to a data structure while one or more writers are concurrently modifying the data structure. Specifically, readers using epoch(9) and smr(9) to synchronize accesses do not block writers, in contrast with reader/writer locks, and they help ensure that memory freed by writers is not reused until all readers which may be accessing it have finished. Thus, they are a useful building block in the construction of lock-free data structures.

These facilities are difficult to use correctly and should be avoided in preference to traditional mutual

exclusion-based synchronization, except when performance or non-blocking guarantees are a major concern.

See epoch(9) and smr(9) for details.

Counting semaphores

Counting semaphores provide a mechanism for synchronizing access to a pool of resources. Unlike mutexes, semaphores do not have the concept of an owner, so they can be useful in situations where one thread needs to acquire a resource, and another thread needs to release it. They are largely deprecated.

See sema(9) for details.

Condition variables

Condition variables are used in conjunction with locks to wait for a condition to become true. A thread must hold the associated lock before calling one of the **cv_wait**(), functions. When a thread waits on a condition, the lock is atomically released before the thread yields the processor and reacquired before the function call returns. Condition variables may be used with blocking mutexes, reader/writer locks, read-mostly locks, and shared/exclusive locks.

See condvar(9) for details.

Sleep/Wakeup

The functions **tsleep**(), **msleep**(), **msleep_spin**(), **pause**(), **wakeup**(), and **wakeup_one**() also handle event-based thread blocking. Unlike condition variables, arbitrary addresses may be used as wait channels and a dedicated structure does not need to be allocated. However, care must be taken to ensure that wait channel addresses are unique to an event. If a thread must wait for an external event, it is put to sleep by **tsleep**(), **msleep**(), **msleep_spin**(), or **pause**(). Threads may also wait using one of the locking primitive sleep routines mtx_sleep(9), rw_sleep(9), or sx_sleep(9).

The parameter *chan* is an arbitrary address that uniquely identifies the event on which the thread is being put to sleep. All threads sleeping on a single *chan* are woken up later by **wakeup**() (often called from inside an interrupt routine) to indicate that the event the thread was blocking on has occurred.

Several of the sleep functions including **msleep**(), **msleep_spin**(), and the locking primitive sleep routines specify an additional lock parameter. The lock will be released before sleeping and reacquired before the sleep routine returns. If *priority* includes the PDROP flag, then the lock will not be reacquired before returning. The lock is used to ensure that a condition can be checked atomically, and that the current thread can be suspended without missing a change to the condition or an associated wakeup. In addition, all of the sleep routines will fully drop the *Giant* mutex (even if recursed) while the thread is suspended and will reacquire the *Giant* mutex (restoring any recursion) before the function

returns.

The **pause**() function is a special sleep function that waits for a specified amount of time to pass before the thread resumes execution. This sleep cannot be terminated early by either an explicit **wakeup**() or a signal.

See sleep(9) for details.

Giant

Giant is a special mutex used to protect data structures that do not yet have their own locks. Since it provides semantics akin to the old spl(9) interface, Giant has special characteristics:

1. It is recursive.

2. Drivers can request that Giant be locked around them by not marking themselves MPSAFE. Note that infrastructure to do this is slowly going away as non-MPSAFE drivers either became properly locked or disappear.

3. Giant must be locked before other non-sleepable locks.

4. Giant is dropped during unbounded sleeps and reacquired after wakeup.

5. There are places in the kernel that drop Giant and pick it back up again. Sleep locks will do this before sleeping. Parts of the network or VM code may do this as well. This means that you cannot count on Giant keeping other code from running if your code sleeps, even if you want it to.

INTERACTIONS

The primitives can interact and have a number of rules regarding how they can and can not be combined. Many of these rules are checked by witness(4).

Bounded vs. Unbounded Sleep

In a bounded sleep (also referred to as "blocking") the only resource needed to resume execution of a thread is CPU time for the owner of a lock that the thread is waiting to acquire. In an unbounded sleep (often referred to as simply "sleeping") a thread waits for an external event or for a condition to become true. In particular, a dependency chain of threads in bounded sleeps should always make forward progress, since there is always CPU time available. This requires that no thread in a bounded sleep is waiting for a lock held by a thread in an unbounded sleep. To avoid priority inversions, a thread in a bounded sleep lends its priority to the owner of the lock that it is waiting for.

The following primitives perform bounded sleeps: mutexes, reader/writer locks and read-mostly locks.

The following primitives perform unbounded sleeps: sleepable read-mostly locks, shared/exclusive locks, lockmanager locks, counting semaphores, condition variables, and sleep/wakeup.

General Principles

- It is an error to do any operation that could result in yielding the processor while holding a spin mutex.

- It is an error to do any operation that could result in unbounded sleep while holding any primitive from the 'bounded sleep' group. For example, it is an error to try to acquire a shared/exclusive lock while holding a mutex, or to try to allocate memory with M_WAITOK while holding a reader/writer lock.

  Note that the lock passed to one of the **sleep**() or **cv_wait**() functions is dropped before the thread enters the unbounded sleep and does not violate this rule.

- It is an error to do any operation that could result in yielding of the processor when running inside an interrupt filter.

- It is an error to do any operation that could result in unbounded sleep when running inside an interrupt thread.

Interaction table

The following table shows what you can and can not do while holding one of the locking primitives discussed. Note that "sleep" includes **sema_wait**(), **sema_timedwait**(), any of the **cv_wait**() functions, and any of the **sleep**() functions.

You want:	spin mtx	mutex/rw	rmlock	sleep rm	sx/lk
You have:	--------	--------	------	--------	------
spin mtx	ok	no	no	no	no
mutex/rw	ok	ok	ok	no	no
rmlock	ok	ok	ok	no	no
sleep rm	ok	ok	ok	ok-2	ok-2
sx	ok	ok	ok	ok	ok
lockmgr	ok	ok	ok	ok	ok

1* There are calls that atomically release this primitive when going to sleep and reacquire it on wakeup (mtx_sleep**(), **rw_sleep**(), **msleep_spin**(), etc.).

**2* These cases are only allowed while holding a write lock on a sleepable read-mostly lock.

3* Though one can sleep while holding this lock, one can also use a **sleep() function to atomically release this primitive when going to sleep and reacquire it on wakeup.

Note that non-blocking try operations on locks are always permitted.

Context mode table

The next table shows what can be used in different contexts. At this time this is a rather easy to remember table.

Context:	spin mtx	mutex/rw	rmlock	sleep rm	sx/lk
interrupt filter:	ok	no	no	no	no
interrupt thread:	ok	ok	ok	no	no
callout:	ok	ok	ok	no	no
direct callout:	ok	no	no	no	no
system call:	ok	ok	ok	ok	ok

SEE ALSO

lockstat(1), witness(4), atomic(9), BUS_SETUP_INTR(9), callout(9), condvar(9), epoch(9), lock(9), LOCK_PROFILING(9), mtx_pool(9), mutex(9), rmlock(9), rwlock(9), sema(9), sleep(9), smr(9), sx(9)

BUGS

There are too many locking primitives to choose from.