Lock types and their rules

Introduction

The kernel provides a variety of locking primitives which can be divided into three categories:

- Sleeping locks
- CPU local locks
- Spinning locks

This document conceptually describes these lock types and provides rules for their nesting, including the rules for use under PREEMPT_RT.

Lock categories

Sleeping locks

Sleeping locks can only be acquired in preemptible task context.

Although implementations allow try_lock() from other contexts, it is necessary to carefully evaluate the safety of unlock() as well as of try_lock(). Furthermore, it is also necessary to evaluate the debugging versions of these primitives. In short, don't acquire sleeping locks from other contexts unless there is no other option.

Sleeping lock types:

- mutex
- rt_mutex
- semaphore
- rw_semaphore
- ww_mutex
- percpu_rw_semaphore

On PREEMPT_RT kernels, these lock types are converted to sleeping locks:

- local_lock
- spinlock_t
- rwlock_t

CPU local locks

- local_lock

On non-PREEMPT_RT kernels, local_lock functions are wrappers around preemption and interrupt disabling primitives. Contrary to other locking mechanisms, disabling preemption or interrupts are pure CPU local concurrency control mechanisms and not suited for inter-CPU concurrency control.

Spinning locks

- raw_spinlock_t
- bit spinlocks

On non-PREEMPT_RT kernels, these lock types are also spinning locks:

- spinlock_t
- rwlock_t

Spinning locks implicitly disable preemption and the lock / unlock functions can have suffixes which apply further protections:

_bh()	Disable / enable bottom halves (soft interrupts)
_irq()	Disable / enable interrupts
_irqsave/restore()	Save and disable / restore interrupt disabled state

Owner semantics

The aforementioned lock types except semaphores have strict owner semantics:

The context (task) that acquired the lock must release it.

rw_semaphores have a special interface which allows non-owner release for readers.

rtmutex

RT-mutexes are mutexes with support for priority inheritance (PI).

PI has limitations on non-PREEMPT_RT kernels due to preemption and interrupt disabled sections.

PI clearly cannot preempt preemption-disabled or interrupt-disabled regions of code, even on PREEMPT_RT kernels. Instead, PREEMPT_RT kernels execute most such regions of code in preemptible task context, especially interrupt handlers and soft interrupts. This conversion allows spinlock_t and rwlock_t to be implemented via RT-mutexes.

semaphore

semaphore is a counting semaphore implementation.

Semaphores are often used for both serialization and waiting, but new use cases should instead use separate serialization and wait mechanisms, such as mutexes and completions.

semaphores and PREEMPT_RT

PREEMPT_RT does not change the semaphore implementation because counting semaphores have no concept of owners, thus preventing PREEMPT_RT from providing priority inheritance for semaphores. After all, an unknown owner cannot be boosted. As a consequence, blocking on semaphores can result in priority inversion.

rw_semaphore

rw_semaphore is a multiple readers and single writer lock mechanism.

On non-PREEMPT_RT kernels the implementation is fair, thus preventing writer starvation.

rw_semaphore complies by default with the strict owner semantics, but there exist special-purpose interfaces that allow non-owner release for readers. These interfaces work independent of the kernel configuration.

rw_semaphore and PREEMPT_RT

PREEMPT_RT kernels map rw_semaphore to a separate rt_mutex-based implementation, thus changing the fairness:

> Because an rw_semaphore writer cannot grant its priority to multiple readers, a preempted low-priority reader will continue holding its lock, thus starving even high-priority writers. In contrast, because readers can grant their priority to a writer, a preempted low-priority writer will have its priority boosted until it releases the lock, thus preventing that writer from starving readers.

local_lock

local_lock provides a named scope to critical sections which are protected by disabling preemption or interrupts.

On non-PREEMPT_RT kernels local_lock operations map to the preemption and interrupt disabling and enabling primitives:

local_lock(&llock)	preempt_disable()
local_unlock(&llock)	preempt_enable()
local_lock_irq(&llock)	local_irq_disable()
local_unlock_irq(&llock)	local_irq_enable()
local_lock_irqsave(&llock)	local_irq_save()
local_unlock_irqrestore(&llock)	local_irq_restore()

The named scope of local_lock has two advantages over the regular primitives:

- The lock name allows static analysis and is also a clear documentation of the protection scope while the regular primitives are scopeless and opaque.
- If lockdep is enabled the local_lock gains a lockmap which allows to validate the correctness of the protection. This can detect cases where e.g. a function using preempt_disable() as protection mechanism is invoked from interrupt or soft-interrupt context. Aside of that lockdep_assert_held(&llock) works as with any other locking primitive.

local_lock and PREEMPT_RT

PREEMPT_RT kernels map local_lock to a per-CPU spinlock_t, thus changing semantics:

- All spinlock_t changes also apply to local_lock.

local_lock usage

local_lock should be used in situations where disabling preemption or interrupts is the appropriate form of concurrency control to protect per-CPU data structures on a non PREEMPT_RT kernel.

local_lock is not suitable to protect against preemption or interrupts on a PREEMPT_RT kernel due to the PREEMPT_RT specific spinlock_t semantics.

raw_spinlock_t and spinlock_t

raw_spinlock_t

raw_spinlock_t is a strict spinning lock implementation in all kernels, including PREEMPT_RT kernels. Use raw_spinlock_t only in real critical core code, low-level interrupt handling and places where disabling preemption or interrupts is required, for example, to safely access hardware state. raw_spinlock_t can sometimes also be used when the critical section is tiny, thus avoiding RT-mutex overhead.

spinlock_t

The semantics of spinlock_t change with the state of PREEMPT_RT.

On a non-PREEMPT_RT kernel spinlock_t is mapped to raw_spinlock_t and has exactly the same semantics.

spinlock_t and PREEMPT_RT

On a PREEMPT_RT kernel spinlock_t is mapped to a separate implementation based on rt_mutex which changes the semantics:

- Preemption is not disabled.
- The hard interrupt related suffixes for spin_lock / spin_unlock operations (_irq, _irqsave / _irqrestore) do not affect the CPU's interrupt disabled state.
- The soft interrupt related suffix (_bh()) still disables softirq handlers.

  Non-PREEMPT_RT kernels disable preemption to get this effect.

  PREEMPT_RT kernels use a per-CPU lock for serialization which keeps preemption enabled. The lock disables softirq handlers and also prevents reentrancy due to task preemption.

PREEMPT_RT kernels preserve all other spinlock_t semantics:

- Tasks holding a spinlock_t do not migrate. Non-PREEMPT_RT kernels avoid migration by disabling preemption. PREEMPT_RT kernels instead disable migration, which ensures that pointers to per-CPU variables remain valid even if the task is preempted.
- Task state is preserved across spinlock acquisition, ensuring that the task-state rules apply to all kernel configurations. Non-PREEMPT_RT kernels leave task state untouched. However, PREEMPT_RT must change task state if the task blocks during acquisition. Therefore, it saves the current task state before blocking and the corresponding lock wakeup restores it, as shown below:

```
task->state = TASK_INTERRUPTIBLE
 lock()
   block()
     task->saved_state = task->state
     task->state = TASK_UNINTERRUPTIBLE
     schedule()
                                    lock wakeup
                                      task->state = task->saved_state
```

  Other types of wakeups would normally unconditionally set the task state to RUNNING, but that does not work here because the task must remain blocked until the lock becomes available. Therefore, when a non-lock wakeup attempts to awaken a task blocked waiting for a spinlock, it instead sets the saved state to RUNNING. Then, when the lock acquisition completes, the lock wakeup sets the task state to the saved state, in this case setting it to RUNNING:

```
task->state = TASK_INTERRUPTIBLE
 lock()
   block()
     task->saved_state = task->state
     task->state = TASK_UNINTERRUPTIBLE
     schedule()
                                    non lock wakeup
                                      task->saved_state = TASK_RUNNING
```

```
lock wakeup
  task->state = task->saved_state
```

This ensures that the real wakeup cannot be lost.

rwlock_t

rwlock_t is a multiple readers and single writer lock mechanism.

Non-PREEMPT_RT kernels implement rwlock_t as a spinning lock and the suffix rules of spinlock_t apply accordingly. The implementation is fair, thus preventing writer starvation.

rwlock_t and PREEMPT_RT

PREEMPT_RT kernels map rwlock_t to a separate rt_mutex-based implementation, thus changing semantics:

- All the spinlock_t changes also apply to rwlock_t.
- Because an rwlock_t writer cannot grant its priority to multiple readers, a preempted low-priority reader will continue holding its lock, thus starving even high-priority writers. In contrast, because readers can grant their priority to a writer, a preempted low-priority writer will have its priority boosted until it releases the lock, thus preventing that writer from starving readers.

PREEMPT_RT caveats

local_lock on RT

The mapping of local_lock to spinlock_t on PREEMPT_RT kernels has a few implications. For example, on a non-PREEMPT_RT kernel the following code sequence works as expected:

```
local_lock_irq(&local_lock);
raw_spin_lock(&lock);
```

and is fully equivalent to:

```
raw_spin_lock_irq(&lock);
```

On a PREEMPT_RT kernel this code sequence breaks because local_lock_irq() is mapped to a per-CPU spinlock_t which neither disables interrupts nor preemption. The following code sequence works perfectly correct on both PREEMPT_RT and non-PREEMPT_RT kernels:

```
local_lock_irq(&local_lock);
spin_lock(&lock);
```

Another caveat with local locks is that each local_lock has a specific protection scope. So the following substitution is wrong:

```
func1()
{
  local_irq_save(flags);    -> local_lock_irqsave(&local_lock_1, flags);
  func3();
  local_irq_restore(flags); -> local_unlock_irqrestore(&local_lock_1, flags);
}

func2()
{
  local_irq_save(flags);    -> local_lock_irqsave(&local_lock_2, flags);
  func3();
  local_irq_restore(flags); -> local_unlock_irqrestore(&local_lock_2, flags);
}

func3()
{
  lockdep_assert_irqs_disabled();
  access_protected_data();
}
```

On a non-PREEMPT_RT kernel this works correctly, but on a PREEMPT_RT kernel local_lock_1 and local_lock_2 are distinct and cannot serialize the callers of func3(). Also the lockdep assert will trigger on a PREEMPT_RT kernel because local_lock_irqsave() does not disable interrupts due to the PREEMPT_RT-specific semantics of spinlock_t. The correct substitution is:

```
func1()
{
  local_irq_save(flags);    -> local_lock_irqsave(&local_lock, flags);
  func3();
  local_irq_restore(flags); -> local_unlock_irqrestore(&local_lock, flags);
```

```
}

func2()
{
  local_irq_save(flags);    -> local_lock_irqsave(&local_lock, flags);
  func3();
  local_irq_restore(flags); -> local_unlock_irqrestore(&local_lock, flags);
}

func3()
{
  lockdep_assert_held(&local_lock);
  access_protected_data();
}
```

spinlock_t and rwlock_t

The changes in spinlock_t and rwlock_t semantics on PREEMPT_RT kernels have a few implications. For example, on a non-PREEMPT_RT kernel the following code sequence works as expected:

```
local_irq_disable();
spin_lock(&lock);
```

and is fully equivalent to:

```
spin_lock_irq(&lock);
```

Same applies to rwlock_t and the _irqsave() suffix variants.

On PREEMPT_RT kernel this code sequence breaks because RT-mutex requires a fully preemptible context. Instead, use spin_lock_irq() or spin_lock_irqsave() and their unlock counterparts. In cases where the interrupt disabling and locking must remain separate, PREEMPT_RT offers a local_lock mechanism. Acquiring the local_lock pins the task to a CPU, allowing things like per-CPU interrupt disabled locks to be acquired. However, this approach should be used only where absolutely necessary.

A typical scenario is protection of per-CPU variables in thread context:

```
struct foo *p = get_cpu_ptr(&var1);

spin_lock(&p->lock);
p->count += this_cpu_read(var2);
```

This is correct code on a non-PREEMPT_RT kernel, but on a PREEMPT_RT kernel this breaks. The PREEMPT_RT-specific change of spinlock_t semantics does not allow to acquire p->lock because get_cpu_ptr() implicitly disables preemption. The following substitution works on both kernels:

```
struct foo *p;

migrate_disable();
p = this_cpu_ptr(&var1);
spin_lock(&p->lock);
p->count += this_cpu_read(var2);
```

migrate_disable() ensures that the task is pinned on the current CPU which in turn guarantees that the per-CPU access to var1 and var2 are staying on the same CPU while the task remains preemptible.

The migrate_disable() substitution is not valid for the following scenario:

```
func()
{
  struct foo *p;

  migrate_disable();
  p = this_cpu_ptr(&var1);
  p->val = func2();
```

This breaks because migrate_disable() does not protect against reentrancy from a preempting task. A correct substitution for this case is:

```
func()
{
  struct foo *p;

  local_lock(&foo_lock);
  p = this_cpu_ptr(&var1);
  p->val = func2();
```

On a non-PREEMPT_RT kernel this protects against reentrancy by disabling preemption. On a PREEMPT_RT kernel this is achieved by acquiring the underlying per-CPU spinlock.

raw_spinlock_t on RT

Acquiring a raw_spinlock_t disables preemption and possibly also interrupts, so the critical section must avoid acquiring a regular spinlock_t or rwlock_t, for example, the critical section must avoid allocating memory. Thus, on a non-PREEMPT_RT kernel the following code works perfectly:

```
raw_spin_lock(&lock);
p = kmalloc(sizeof(*p), GFP_ATOMIC);
```

But this code fails on PREEMPT_RT kernels because the memory allocator is fully preemptible and therefore cannot be invoked from truly atomic contexts. However, it is perfectly fine to invoke the memory allocator while holding normal non-raw spinlocks because they do not disable preemption on PREEMPT_RT kernels:

```
spin_lock(&lock);
p = kmalloc(sizeof(*p), GFP_ATOMIC);
```

bit spinlocks

PREEMPT_RT cannot substitute bit spinlocks because a single bit is too small to accommodate an RT-mutex. Therefore, the semantics of bit spinlocks are preserved on PREEMPT_RT kernels, so that the raw_spinlock_t caveats also apply to bit spinlocks.

Some bit spinlocks are replaced with regular spinlock_t for PREEMPT_RT using conditional (#ifdef'ed) code changes at the usage site. In contrast, usage-site changes are not needed for the spinlock_t substitution. Instead, conditionals in header files and the core locking implemementation enable the compiler to do the substitution transparently.

Lock type nesting rules

The most basic rules are:

- Lock types of the same lock category (sleeping, CPU local, spinning) can nest arbitrarily as long as they respect the general lock ordering rules to prevent deadlocks.
- Sleeping lock types cannot nest inside CPU local and spinning lock types.
- CPU local and spinning lock types can nest inside sleeping lock types.
- Spinning lock types can nest inside all lock types

These constraints apply both in PREEMPT_RT and otherwise.

The fact that PREEMPT_RT changes the lock category of spinlock_t and rwlock_t from spinning to sleeping and substitutes local_lock with a per-CPU spinlock_t means that they cannot be acquired while holding a raw spinlock. This results in the following nesting ordering:

1. Sleeping locks
2. spinlock_t, rwlock_t, local_lock
3. raw_spinlock_t and bit spinlocks

Lockdep will complain if these constraints are violated, both in PREEMPT_RT and otherwise.