Linux Locking Documentation

The kernel development community

Jun 10, 2024

CONTENTS

CHAPTER

ONE

LOCK TYPES AND THEIR RULES

1.1 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.

1.2 Lock categories

1.2.1 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

1.2.2 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.

1.2.3 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

1.3 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.

1.4 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.

1.5 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.

1.5.1 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.

1.6 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.

1.6.1 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.

1.7 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.

1.7.1 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.

1.7.2 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.

1.8 raw_spinlock_t and spinlock_t

1.8.1 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.

1.8.2 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.

1.8.3 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.

1.9 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.

1.9.1 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.

1.10 PREEMPT_RT caveats

1.10.1 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();
}
```

1.10.2 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.

1.10.3 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);
```

1.10.4 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 implementation enable the compiler to do the substitution transparently.

1.11 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.

CHAPTER

TWO

RUNTIME LOCKING CORRECTNESS VALIDATOR

started by Ingo Molnar <mingo@redhat.com>

additions by Arjan van de Ven <arjan@linux.intel.com>

2.1 Lock-class

The basic object the validator operates upon is a ‘class’ of locks.

A class of locks is a group of locks that are logically the same with respect to locking rules, even if the locks may have multiple (possibly tens of thousands of) instantiations. For example a lock in the inode struct is one class, while each inode has its own instantiation of that lock class.

The validator tracks the ‘usage state’ of lock-classes, and it tracks the dependencies between different lock-classes. Lock usage indicates how a lock is used with regard to its IRQ contexts, while lock dependency can be understood as lock order, where L1 -> L2 suggests that a task is attempting to acquire L2 while holding L1. From lockdep’s perspective, the two locks (L1 and L2) are not necessarily related; that dependency just means the order ever happened. The validator maintains a continuing effort to prove lock usages and dependencies are correct or the validator will shoot a splat if incorrect.

A lock-class’s behavior is constructed by its instances collectively: when the first instance of a lock-class is used after bootup the class gets registered, then all (subsequent) instances will be mapped to the class and hence their usages and dependencies will contribute to those of the class. A lock-class does not go away when a lock instance does, but it can be removed if the memory space of the lock class (static or dynamic) is reclaimed, this happens for example when a module is unloaded or a workqueue is destroyed.

2.2 State

The validator tracks lock-class usage history and divides the usage into (4 usages * n STATEs + 1) categories:

where the 4 usages can be:

- ‘ever held in STATE context’
- ‘ever held as readlock in STATE context’
- ‘ever held with STATE enabled’
- ‘ever held as readlock with STATE enabled’

where the n STATEs are coded in kernel/locking/lockdep_states.h and as of now they include:

- hardirq
- softirq

where the last 1 category is:

- 'ever used' [== !unused]

When locking rules are violated, these usage bits are presented in the locking error messages, inside curlies, with a total of 2 * n STATEs bits. A contrived example:

```
modprobe/2287 is trying to acquire lock:
 (&sio_locks[i].lock){-.-.}, at: [<c02867fd>] mutex_lock+0x21/0x24

but task is already holding lock:
 (&sio_locks[i].lock){-.-.}, at: [<c02867fd>] mutex_lock+0x21/0x24
```

For a given lock, the bit positions from left to right indicate the usage of the lock and readlock (if exists), for each of the n STATEs listed above respectively, and the character displayed at each bit position indicates:

'.'	acquired while irqs disabled and not in irq context
'-'	acquired in irq context
'+'	acquired with irqs enabled
'?'	acquired in irq context with irqs enabled.

The bits are illustrated with an example:

```
(&sio_locks[i].lock){-.-.}, at: [<c02867fd>] mutex_lock+0x21/0x24
                       ||||
                       ||| \-> softirq disabled and not in softirq context
                       || \--> acquired in softirq context
                       | \---> hardirq disabled and not in hardirq context
                        \----> acquired in hardirq context
```

For a given STATE, whether the lock is ever acquired in that STATE context and whether that STATE is enabled yields four possible cases as shown in the table below. The bit character is able to indicate which exact case is for the lock as of the reporting time.

	irq enabled	irq disabled
ever in irq	'?'	'-'
never in irq	'+'	'.'

The character '-' suggests irq is disabled because if otherwise the character '?' would have been shown instead. Similar deduction can be applied for '+' too.

Unused locks (e.g., mutexes) cannot be part of the cause of an error.

2.3 Single-lock state rules:

A lock is irq-safe means it was ever used in an irq context, while a lock is irq-unsafe means it was ever acquired with irq enabled.

A softirq-unsafe lock-class is automatically hardirq-unsafe as well. The following states must be exclusive: only one of them is allowed to be set for any lock-class based on its usage:

```
<hardirq-safe> or <hardirq-unsafe>
<softirq-safe> or <softirq-unsafe>
```

This is because if a lock can be used in irq context (irq-safe) then it cannot be ever acquired with irq enabled (irq-unsafe). Otherwise, a deadlock may happen. For example, in the scenario that after this lock was acquired but before released, if the context is interrupted this lock will be attempted to acquire twice, which creates a deadlock, referred to as lock recursion deadlock.

The validator detects and reports lock usage that violates these single-lock state rules.

2.4 Multi-lock dependency rules:

The same lock-class must not be acquired twice, because this could lead to lock recursion deadlocks.

Furthermore, two locks can not be taken in inverse order:

```
<L1> -> <L2>
<L2> -> <L1>
```

because this could lead to a deadlock - referred to as lock inversion deadlock - as attempts to acquire the two locks form a circle which could lead to the two contexts waiting for each other permanently. The validator will find such dependency circle in arbitrary complexity, i.e., there can be any other locking sequence between the acquire-lock operations; the validator will still find whether these locks can be acquired in a circular fashion.

Furthermore, the following usage based lock dependencies are not allowed between any two lock-classes:

```
<hardirq-safe>   ->  <hardirq-unsafe>
<softirq-safe>   ->  <softirq-unsafe>
```

The first rule comes from the fact that a hardirq-safe lock could be taken by a hardirq context, interrupting a hardirq-unsafe lock - and thus could result in a lock inversion deadlock. Likewise, a softirq-safe lock could be taken by an softirq context, interrupting a softirq-unsafe lock.

The above rules are enforced for any locking sequence that occurs in the kernel: when acquiring a new lock, the validator checks whether there is any rule violation between the new lock and any of the held locks.

When a lock-class changes its state, the following aspects of the above dependency rules are enforced:

- if a new hardirq-safe lock is discovered, we check whether it took any hardirq-unsafe lock in the past.

- if a new softirq-safe lock is discovered, we check whether it took any softirq-unsafe lock in the past.
- if a new hardirq-unsafe lock is discovered, we check whether any hardirq-safe lock took it in the past.
- if a new softirq-unsafe lock is discovered, we check whether any softirq-safe lock took it in the past.

(Again, we do these checks too on the basis that an interrupt context could interrupt _any_ of the irq-unsafe or hardirq-unsafe locks, which could lead to a lock inversion deadlock - even if that lock scenario did not trigger in practice yet.)

2.5 Exception: Nested data dependencies leading to nested locking

There are a few cases where the Linux kernel acquires more than one instance of the same lock-class. Such cases typically happen when there is some sort of hierarchy within objects of the same type. In these cases there is an inherent “natural” ordering between the two objects (defined by the properties of the hierarchy), and the kernel grabs the locks in this fixed order on each of the objects.

An example of such an object hierarchy that results in “nested locking” is that of a “whole disk” block-dev object and a “partition” block-dev object; the partition is “part of” the whole device and as long as one always takes the whole disk lock as a higher lock than the partition lock, the lock ordering is fully correct. The validator does not automatically detect this natural ordering, as the locking rule behind the ordering is not static.

In order to teach the validator about this correct usage model, new versions of the various locking primitives were added that allow you to specify a “nesting level”. An example call, for the block device mutex, looks like this:

```
enum bdev_bd_mutex_lock_class
{
     BD_MUTEX_NORMAL,
     BD_MUTEX_WHOLE,
     BD_MUTEX_PARTITION
};

mutex_lock_nested(&bdev->bd_contains->bd_mutex, BD_MUTEX_PARTITION);
```

In this case the locking is done on a bdev object that is known to be a partition.

The validator treats a lock that is taken in such a nested fashion as a separate (sub)class for the purposes of validation.

Note: When changing code to use the _nested() primitives, be careful and check really thoroughly that the hierarchy is correctly mapped; otherwise you can get false positives or false negatives.

2.6 Annotations

Two constructs can be used to annotate and check where and if certain locks must be held: lockdep_assert_held*(&lock) and lockdep_*pin_lock(&lock).

As the name suggests, lockdep_assert_held* family of macros assert that a particular lock is held at a certain time (and generate a WARN() otherwise). This annotation is largely used all over the kernel, e.g. kernel/sched/ core.c:

```
void update_rq_clock(struct rq *rq)
{
      s64 delta;

      lockdep_assert_held(&rq->lock);
      [...]
}
```

where holding rq->lock is required to safely update a rq's clock.

The other family of macros is lockdep_*pin_lock(), which is admittedly only used for rq->lock ATM. Despite their limited adoption these annotations generate a WARN() if the lock of interest is "accidentally" unlocked. This turns out to be especially helpful to debug code with callbacks, where an upper layer assumes a lock remains taken, but a lower layer thinks it can maybe drop and reacquire the lock ("unwittingly" introducing races). lockdep_pin_lock() returns a 'struct pin_cookie' that is then used by lockdep_unpin_lock() to check that nobody tampered with the lock, e.g. kernel/sched/sched.h:

```
static inline void rq_pin_lock(struct rq *rq, struct rq_flags *rf)
{
      rf->cookie = lockdep_pin_lock(&rq->lock);
      [...]
}

static inline void rq_unpin_lock(struct rq *rq, struct rq_flags *rf)
{
      [...]
      lockdep_unpin_lock(&rq->lock, rf->cookie);
}
```

While comments about locking requirements might provide useful information, the runtime checks performed by annotations are invaluable when debugging locking problems and they carry the same level of details when inspecting code. Always prefer annotations when in doubt!

2.7 Proof of 100% correctness:

The validator achieves perfect, mathematical 'closure' (proof of locking correctness) in the sense that for every simple, standalone single-task locking sequence that occurred at least once during the lifetime of the kernel, the validator proves it with a 100% certainty that no combination and timing of these locking sequences can cause any class of lock related deadlock.[1]

I.e. complex multi-CPU and multi-task locking scenarios do not have to occur in practice to prove a deadlock: only the simple 'component' locking chains have to occur at least once (anytime, in any task/context) for the validator to be able to prove correctness. (For example, complex deadlocks that would normally need more than 3 CPUs and a very unlikely constellation of tasks, irq-contexts and timings to occur, can be detected on a plain, lightly loaded single-CPU system as well!)

This radically decreases the complexity of locking related QA of the kernel: what has to be done during QA is to trigger as many "simple" single-task locking dependencies in the kernel as possible, at least once, to prove locking correctness - instead of having to trigger every possible combination of locking interaction between CPUs, combined with every possible hardirq and softirq nesting scenario (which is impossible to do in practice).

2.8 Performance:

The above rules require **massive** amounts of runtime checking. If we did that for every lock taken and for every irqs-enable event, it would render the system practically unusably slow. The complexity of checking is O(N^2), so even with just a few hundred lock-classes we'd have to do tens of thousands of checks for every event.

This problem is solved by checking any given 'locking scenario' (unique sequence of locks taken after each other) only once. A simple stack of held locks is maintained, and a lightweight 64-bit hash value is calculated, which hash is unique for every lock chain. The hash value, when the chain is validated for the first time, is then put into a hash table, which hash-table can be checked in a lockfree manner. If the locking chain occurs again later on, the hash table tells us that we don't have to validate the chain again.

2.9 Troubleshooting:

The validator tracks a maximum of MAX_LOCKDEP_KEYS number of lock classes. Exceeding this number will trigger the following lockdep warning:

```
(DEBUG_LOCKS_WARN_ON(id >= MAX_LOCKDEP_KEYS))
```

By default, MAX_LOCKDEP_KEYS is currently set to 8191, and typical desktop systems have less than 1,000 lock classes, so this warning normally results from lock-class leakage or failure to properly initialize locks. These two problems are illustrated below:

[1] assuming that the validator itself is 100% correct, and no other part of the system corrupts the state of the validator in any way. We also assume that all NMI/SMM paths [which could interrupt even hardirq-disabled codepaths] are correct and do not interfere with the validator. We also assume that the 64-bit 'chain hash' value is unique for every lock-chain in the system. Also, lock recursion must not be higher than 20.

1. Repeated module loading and unloading while running the validator will result in lock-class leakage. The issue here is that each load of the module will create a new set of lock classes for that module's locks, but module unloading does not remove old classes (see below discussion of reuse of lock classes for why). Therefore, if that module is loaded and unloaded repeatedly, the number of lock classes will eventually reach the maximum.

2. Using structures such as arrays that have large numbers of locks that are not explicitly initialized. For example, a hash table with 8192 buckets where each bucket has its own spinlock_t will consume 8192 lock classes -unless- each spinlock is explicitly initialized at runtime, for example, using the run-time spin_lock_init() as opposed to compile-time initializers such as __SPIN_LOCK_UNLOCKED(). Failure to properly initialize the per-bucket spinlocks would guarantee lock-class overflow. In contrast, a loop that called spin_lock_init() on each lock would place all 8192 locks into a single lock class.

   The moral of this story is that you should always explicitly initialize your locks.

One might argue that the validator should be modified to allow lock classes to be reused. However, if you are tempted to make this argument, first review the code and think through the changes that would be required, keeping in mind that the lock classes to be removed are likely to be linked into the lock-dependency graph. This turns out to be harder to do than to say.

Of course, if you do run out of lock classes, the next thing to do is to find the offending lock classes. First, the following command gives you the number of lock classes currently in use along with the maximum:

```
grep "lock-classes" /proc/lockdep_stats
```

This command produces the following output on a modest system:

```
lock-classes:                            748 [max: 8191]
```

If the number allocated (748 above) increases continually over time, then there is likely a leak. The following command can be used to identify the leaking lock classes:

```
grep "BD" /proc/lockdep
```

Run the command and save the output, then compare against the output from a later run of this command to identify the leakers. This same output can also help you find situations where runtime lock initialization has been omitted.

2.10 Recursive read locks:

The whole of the rest document tries to prove a certain type of cycle is equivalent to deadlock possibility.

There are three types of lockers: writers (i.e. exclusive lockers, like spin_lock() or write_lock()), non-recursive readers (i.e. shared lockers, like down_read()) and recursive readers (recursive shared lockers, like rcu_read_lock()). And we use the following notations of those lockers in the rest of the document:

> W or E: stands for writers (exclusive lockers). r: stands for non-recursive readers. R: stands for recursive readers. S: stands for all readers (non-recursive + recursive), as both are shared lockers. N: stands for writers and non-recursive readers, as both are not recursive.

Obviously, N is “r or W” and S is “r or R”.

Recursive readers, as their name indicates, are the lockers allowed to acquire even inside the critical section of another reader of the same lock instance, in other words, allowing nested read-side critical sections of one lock instance.

While non-recursive readers will cause a self deadlock if trying to acquire inside the critical section of another reader of the same lock instance.

The difference between recursive readers and non-recursive readers is because: recursive readers get blocked only by a write lock *holder*, while non-recursive readers could get blocked by a write lock *waiter*. Considering the follow example:

```
TASK A:                 TASK B:

read_lock(X);
                        write_lock(X);
read_lock_2(X);
```

Task A gets the reader (no matter whether recursive or non-recursive) on X via read_lock() first. And when task B tries to acquire writer on X, it will block and become a waiter for writer on X. Now if read_lock_2() is recursive readers, task A will make progress, because writer waiters don’t block recursive readers, and there is no deadlock. However, if read_lock_2() is non-recursive readers, it will get blocked by writer waiter B, and cause a self deadlock.

2.11 Block conditions on readers/writers of the same lock instance:

There are simply four block conditions:

1. Writers block other writers.
2. Readers block writers.
3. Writers block both recursive readers and non-recursive readers.
4. And readers (recursive or not) don’t block other recursive readers but may block non-recursive readers (because of the potential co-existing writer waiters)

Block condition matrix, Y means the row blocks the column, and N means otherwise.

	W	r	R
W	Y	Y	Y
r	Y	Y	N
R	Y	Y	N

(W: writers, r: non-recursive readers, R: recursive readers)

acquired recursively. Unlike non-recursive read locks, recursive read locks only get blocked by current write lock *holders* other than write lock *waiters*, for example:

```
TASK A:                 TASK B:
```

```
read_lock(X);

                        write_lock(X);

read_lock(X);
```

is not a deadlock for recursive read locks, as while the task B is waiting for the lock X, the second read_lock() doesn't need to wait because it's a recursive read lock. However if the read_lock() is non-recursive read lock, then the above case is a deadlock, because even if the write_lock() in TASK B cannot get the lock, but it can block the second read_lock() in TASK A.

Note that a lock can be a write lock (exclusive lock), a non-recursive read lock (non-recursive shared lock) or a recursive read lock (recursive shared lock), depending on the lock operations used to acquire it (more specifically, the value of the 'read' parameter for lock_acquire()). In other words, a single lock instance has three types of acquisition depending on the acquisition functions: exclusive, non-recursive read, and recursive read.

To be concise, we call that write locks and non-recursive read locks as "non-recursive" locks and recursive read locks as "recursive" locks.

Recursive locks don't block each other, while non-recursive locks do (this is even true for two non-recursive read locks). A non-recursive lock can block the corresponding recursive lock, and vice versa.

A deadlock case with recursive locks involved is as follow:

```
TASK A:                 TASK B:

read_lock(X);
                        read_lock(Y);
write_lock(Y);
                        write_lock(X);
```

Task A is waiting for task B to read_unlock() Y and task B is waiting for task A to read_unlock() X.

2.12 Dependency types and strong dependency paths:

Lock dependencies record the orders of the acquisitions of a pair of locks, and because there are 3 types for lockers, there are, in theory, 9 types of lock dependencies, but we can show that 4 types of lock dependencies are enough for deadlock detection.

For each lock dependency:

```
L1 -> L2
```

, which means lockdep has seen L1 held before L2 held in the same context at runtime. And in deadlock detection, we care whether we could get blocked on L2 with L1 held, IOW, whether there is a locker L3 that L1 blocks L3 and L2 gets blocked by L3. So we only care about 1) what L1 blocks and 2) what blocks L2. As a result, we can combine recursive readers and non-recursive readers for L1 (as they block the same types) and we can combine writers and non-recursive readers for L2 (as they get blocked by the same types).

With the above combination for simplification, there are 4 types of dependency edges in the lockdep graph:

1) **-(ER)->:**
   exclusive writer to recursive reader dependency, “X -(ER)-> Y” means X -> Y and X is a writer and Y is a recursive reader.

2) **-(EN)->:**
   exclusive writer to non-recursive locker dependency, “X -(EN)-> Y” means X -> Y and X is a writer and Y is either a writer or non-recursive reader.

3) **-(SR)->:**
   shared reader to recursive reader dependency, “X -(SR)-> Y” means X -> Y and X is a reader (recursive or not) and Y is a recursive reader.

4) **-(SN)->:**
   shared reader to non-recursive locker dependency, “X -(SN)-> Y” means X -> Y and X is a reader (recursive or not) and Y is either a writer or non-recursive reader.

Note that given two locks, they may have multiple dependencies between them, for example:

```
TASK A:

read_lock(X);
write_lock(Y);
...

TASK B:

write_lock(X);
write_lock(Y);
```

, we have both X -(SN)-> Y and X -(EN)-> Y in the dependency graph.

We use -(xN)-> to represent edges that are either -(EN)-> or -(SN)->, the similar for -(Ex)->, -(xR)-> and -(Sx)->

A “path” is a series of conjunct dependency edges in the graph. And we define a “strong” path, which indicates the strong dependency throughout each dependency in the path, as the path that doesn’t have two conjunct edges (dependencies) as -(xR)-> and -(Sx)->. In other words, a “strong” path is a path from a lock walking to another through the lock dependencies, and if X -> Y -> Z is in the path (where X, Y, Z are locks), and the walk from X to Y is through a -(SR)-> or -(ER)-> dependency, the walk from Y to Z must not be through a -(SN)-> or -(SR)-> dependency.

We will see why the path is called “strong” in next section.

2.13 Recursive Read Deadlock Detection:

We now prove two things:

Lemma 1:

If there is a closed strong path (i.e. a strong circle), then there is a combination of locking sequences that causes deadlock. I.e. a strong circle is sufficient for deadlock detection.

Lemma 2:

If there is no closed strong path (i.e. strong circle), then there is no combination of locking sequences that could cause deadlock. I.e. strong circles are necessary for deadlock detection.

With these two Lemmas, we can easily say a closed strong path is both sufficient and necessary for deadlocks, therefore a closed strong path is equivalent to deadlock possibility. As a closed strong path stands for a dependency chain that could cause deadlocks, so we call it “strong”, considering there are dependency circles that won’t cause deadlocks.

Proof for sufficiency (Lemma 1):

Let’s say we have a strong circle:

```
L1 -> L2 ... -> Ln -> L1
```

, which means we have dependencies:

```
L1 -> L2
L2 -> L3
...
Ln-1 -> Ln
Ln -> L1
```

We now can construct a combination of locking sequences that cause deadlock:

Firstly let’s make one CPU/task get the L1 in L1 -> L2, and then another get the L2 in L2 -> L3, and so on. After this, all of the Lx in Lx -> Lx+1 are held by different CPU/tasks.

And then because we have L1 -> L2, so the holder of L1 is going to acquire L2 in L1 -> L2, however since L2 is already held by another CPU/task, plus L1 -> L2 and L2 -> L3 are not -(xR)-> and -(Sx)-> (the definition of strong), which means either L2 in L1 -> L2 is a non-recursive locker (blocked by anyone) or the L2 in L2 -> L3, is writer (blocking anyone), therefore the holder of L1 cannot get L2, it has to wait L2’s holder to release.

Moreover, we can have a similar conclusion for L2’s holder: it has to wait L3’s holder to release, and so on. We now can prove that Lx’s holder has to wait for Lx+1’s holder to release, and note that Ln+1 is L1, so we have a circular waiting scenario and nobody can get progress, therefore a deadlock.

Proof for necessary (Lemma 2):

Lemma 2 is equivalent to: If there is a deadlock scenario, then there must be a strong circle in the dependency graph.

According to Wikipedia[1], if there is a deadlock, then there must be a circular waiting scenario, means there are N CPU/tasks, where CPU/task P1 is waiting for a lock held by P2, and P2 is waiting for a lock held by P3, ... and Pn is waiting for a lock held by P1. Let’s name the lock Px is waiting as Lx, so since P1 is waiting for L1 and holding Ln, so we will have Ln -> L1 in

the dependency graph. Similarly, we have L1 -> L2, L2 -> L3, ..., Ln-1 -> Ln in the dependency graph, which means we have a circle:

```
Ln -> L1 -> L2 -> ... -> Ln
```

, and now let's prove the circle is strong:

For a lock Lx, Px contributes the dependency Lx-1 -> Lx and Px+1 contributes the dependency Lx -> Lx+1, and since Px is waiting for Px+1 to release Lx, so it's impossible that Lx on Px+1 is a reader and Lx on Px is a recursive reader, because readers (no matter recursive or not) don't block recursive readers, therefore Lx-1 -> Lx and Lx -> Lx+1 cannot be a -(xR)-> -(Sx)-> pair, and this is true for any lock in the circle, therefore, the circle is strong.

2.14 References:

[1]: https://en.wikipedia.org/wiki/Deadlock [2]: Shibu, K. (2009). Intro To Embedded Systems (1st ed.). Tata McGraw-Hill

CHAPTER

THREE

LOCK STATISTICS

3.1 What

As the name suggests, it provides statistics on locks.

3.2 Why

Because things like lock contention can severely impact performance.

3.3 How

Lockdep already has hooks in the lock functions and maps lock instances to lock classes. We build on that (see *Runtime locking correctness validator*). The graph below shows the relation between the lock functions and the various hooks therein:

```
      __acquire
          |
         lock _____
          |        \
          |    __contended
          |         |
          |       <wait>
          | _______/
          |/
          |
     __acquired
          |
          .
        <hold>
          .
          |
     __release
          |
       unlock

lock, unlock  - the regular lock functions
```

```
__*           - the hooks
<>            - states
```

With these hooks we provide the following statistics:

con-bounces

- number of lock contention that involved x-cpu data

contentions

- number of lock acquisitions that had to wait

wait time

min

- shortest (non-0) time we ever had to wait for a lock

max

- longest time we ever had to wait for a lock

total

- total time we spend waiting on this lock

avg

- average time spent waiting on this lock

acq-bounces

- number of lock acquisitions that involved x-cpu data

acquisitions

- number of times we took the lock

hold time

min

- shortest (non-0) time we ever held the lock

max

- longest time we ever held the lock

total

- total time this lock was held

avg

- average time this lock was held

These numbers are gathered per lock class, per read/write state (when applicable).

It also tracks 4 contention points per class. A contention point is a call site that had to wait on lock acquisition.

3.3.1 Configuration

Lock statistics are enabled via CONFIG_LOCK_STAT.

3.3.2 Usage

Enable collection of statistics:

```
# echo 1 >/proc/sys/kernel/lock_stat
```

Disable collection of statistics:

```
# echo 0 >/proc/sys/kernel/lock_stat
```

Look at the current lock statistics:

```
( line numbers not part of actual output, done for clarity in the explanation
  below )

# less /proc/lock_stat

01 lock_stat version 0.4
02-------------------------------------------------------------------------------
 ↪-------------------------------------------------------------------------------
 ↪-----------------------------------------------------------------
03                              class name    con-bounces    contentions ␣
 ↪waittime-min   waittime-max waittime-total   waittime-avg    acq-bounces ␣
 ↪acquisitions   holdtime-min   holdtime-max holdtime-total   holdtime-avg
04-------------------------------------------------------------------------------
 ↪-------------------------------------------------------------------------------
 ↪-----------------------------------------------------------------
05
06                          &mm->mmap_sem-W:            46             84 ␣
 ↪    0.26         939.10       16371.53         194.90          47291 ␣ ␣
 ↪2922365           0.16     2220301.69 17464026916.32        5975.99
07                          &mm->mmap_sem-R:            37            100 ␣
 ↪    1.31      299502.61      325629.52        3256.30         212344 ␣ ␣
 ↪34316685           0.10        7744.91    95016910.20           2.77
08                          ---------------
09                            &mm->mmap_sem             1          [
 ↪<ffffffff811502a7>] khugepaged_scan_mm_slot+0x57/0x280
10                            &mm->mmap_sem            96          [
 ↪<ffffffff815351c4>] __do_page_fault+0x1d4/0x510
11                            &mm->mmap_sem            34          [
 ↪<ffffffff81113d77>] vm_mmap_pgoff+0x87/0xd0
12                            &mm->mmap_sem            17          [
 ↪<ffffffff81127e71>] vm_munmap+0x41/0x80
13                          ---------------
14                            &mm->mmap_sem             1          [
 ↪<ffffffff81046fda>] dup_mmap+0x2a/0x3f0
15                            &mm->mmap_sem            60          [
```

```
 ↪<ffffffff81129e29>] SyS_mprotect+0xe9/0x250
16                            &mm->mmap_sem             41          [
 ↪<ffffffff815351c4>] __do_page_fault+0x1d4/0x510
17                            &mm->mmap_sem             68          [
 ↪<ffffffff81113d77>] vm_mmap_pgoff+0x87/0xd0
18
19.............................................................................
 ↪.............................................................................
 ↪..............................................................
20
21                          unix_table_lock:           110            112
 ↪    0.21          49.24         163.91           1.46          21094
 ↪ 66312          0.12         624.42       31589.81          0.48
22                          ---------------
23                          unix_table_lock             45          [
 ↪<ffffffff8150ad8e>] unix_create1+0x16e/0x1b0
24                          unix_table_lock             47          [
 ↪<ffffffff8150b111>] unix_release_sock+0x31/0x250
25                          unix_table_lock             15          [
 ↪<ffffffff8150ca37>] unix_find_other+0x117/0x230
26                          unix_table_lock              5          [
 ↪<ffffffff8150a09f>] unix_autobind+0x11f/0x1b0
27                          ---------------
28                          unix_table_lock             39          [
 ↪<ffffffff8150b111>] unix_release_sock+0x31/0x250
29                          unix_table_lock             49          [
 ↪<ffffffff8150ad8e>] unix_create1+0x16e/0x1b0
30                          unix_table_lock             20          [
 ↪<ffffffff8150ca37>] unix_find_other+0x117/0x230
31                          unix_table_lock              4          [
 ↪<ffffffff8150a09f>] unix_autobind+0x11f/0x1b0
```

This excerpt shows the first two lock class statistics. Line 01 shows the output version - each time the format changes this will be updated. Line 02-04 show the header with column descriptions. Lines 05-18 and 20-31 show the actual statistics. These statistics come in two parts; the actual stats separated by a short separator (line 08, 13) from the contention points.

Lines 09-12 show the first 4 recorded contention points (the code which tries to get the lock) and lines 14-17 show the first 4 recorded contended points (the lock holder). It is possible that the max con-bounces point is missing in the statistics.

The first lock (05-18) is a read/write lock, and shows two lines above the short separator. The contention points don't match the column descriptors, they have two: contentions and [<IP>] symbol. The second set of contention points are the points we're contending with.

The integer part of the time values is in us.

Dealing with nested locks, subclasses may appear:

```
32.............................................................................
 ↪.............................................................................
 ↪..............................................................
33
```

```
34                                &rq->lock:       13128          13128
 ↪  0.43         190.53      103881.26           7.91          97454
 ↪3453404           0.00         401.11    13224683.11           3.82
35                                ---------
36                                &rq->lock          645          [
 ↪<ffffffff8103bfc4>] task_rq_lock+0x43/0x75
37                                &rq->lock          297          [
 ↪<ffffffff8104ba65>] try_to_wake_up+0x127/0x25a
38                                &rq->lock          360          [
 ↪<ffffffff8103c4c5>] select_task_rq_fair+0x1f0/0x74a
39                                &rq->lock          428          [
 ↪<ffffffff81045f98>] scheduler_tick+0x46/0x1fb
40                                ---------
41                                &rq->lock           77          [
 ↪<ffffffff8103bfc4>] task_rq_lock+0x43/0x75
42                                &rq->lock          174          [
 ↪<ffffffff8104ba65>] try_to_wake_up+0x127/0x25a
43                                &rq->lock         4715          [
 ↪<ffffffff8103ed4b>] double_rq_lock+0x42/0x54
44                                &rq->lock          893          [
 ↪<ffffffff81340524>] schedule+0x157/0x7b8
45
46.....................................................................
 ↪.....................................................................
 ↪.............................................................
47
48                              &rq->lock/1:        1526          11488
 ↪  0.33         388.73      136294.31          11.86          21461
 ↪38404           0.00          37.93      109388.53           2.84
49                              -----------
50                              &rq->lock/1        11526          [
 ↪<ffffffff8103ed58>] double_rq_lock+0x4f/0x54
51                              -----------
52                              &rq->lock/1         5645          [
 ↪<ffffffff8103ed4b>] double_rq_lock+0x42/0x54
53                              &rq->lock/1         1224          [
 ↪<ffffffff81340524>] schedule+0x157/0x7b8
54                              &rq->lock/1         4336          [
 ↪<ffffffff8103ed58>] double_rq_lock+0x4f/0x54
55                              &rq->lock/1          181          [
 ↪<ffffffff8104ba65>] try_to_wake_up+0x127/0x25a
```

Line 48 shows statistics for the second subclass (/1) of &rq->lock class (subclass starts from 0), since in this case, as line 50 suggests, double_rq_lock actually acquires a nested lock of two spinlocks.

View the top contending locks:

```
# grep : /proc/lock_stat | head
                     clockevents_lock:       2926159        2947636
 ↪0.15       46882.81  1784540466.34         605.41        3381345
```

```
↪3879161           0.00        2260.97    53178395.68          13.71
                  tick_broadcast_lock:         346460         346717
↪0.18        2257.43    39364622.71         113.54        3642919
↪4242696           0.00        2263.79    49173646.60          11.59
               &mapping->i_mmap_mutex:         203896         203899
↪3.36      645530.05 31767507988.39      155800.21        3361776
↪8893984           0.17        2254.15    14110121.02           1.59
                             &rq->lock:         135014         136909
↪0.18         606.09      842160.68           6.15        1540728
↪10436146           0.00         728.72    17606683.41           1.69
            &(&zone->lru_lock)->rlock:          93000          94934
↪0.16          59.18      188253.78           1.98        1199912
↪3809894           0.15         391.40     3559518.81           0.93
                       tasklist_lock-W:          40667          41130
↪0.23        1189.42      428980.51          10.43         270278
↪510106            0.16         653.51     3939674.91           7.72
                       tasklist_lock-R:          21298          21305
↪0.20        1310.05      215511.12          10.12         186204
↪241258            0.14        1162.33     1179779.23           4.89
                            rcu_node_1:          47656          49022
↪0.16         635.41      193616.41           3.95         844888
↪1865423           0.00         764.26     1656226.96           0.89
    &(&dentry->d_lockref.lock)->rlock:          39791          40179
↪0.15        1302.08       88851.96           2.21        2790851
↪12527025          0.10        1910.75     3379714.27           0.27
                            rcu_node_0:          29203          30064
↪0.16         786.55     1555573.00          51.74          88963
↪244254            0.00         398.87      428872.51           1.76
```

Clear the statistics:

```
# echo 0 > /proc/lock_stat
```

CHAPTER

FOUR

KERNEL LOCK TORTURE TEST OPERATION

4.1 CONFIG_LOCK_TORTURE_TEST

The CONFIG_LOCK_TORTURE_TEST config option provides a kernel module that runs torture tests on core kernel locking primitives. The kernel module, 'locktorture', may be built after the fact on the running kernel to be tested, if desired. The tests periodically output status messages via printk(), which can be examined via the dmesg (perhaps grepping for "torture"). The test is started when the module is loaded, and stops when the module is unloaded. This program is based on how RCU is tortured, via rcutorture.

This torture test consists of creating a number of kernel threads which acquire the lock and hold it for specific amount of time, thus simulating different critical region behaviors. The amount of contention on the lock can be simulated by either enlarging this critical region hold time and/or creating more kthreads.

4.2 Module Parameters

This module has the following parameters:

4.2.1 Locktorture-specific

nwriters_stress
Number of kernel threads that will stress exclusive lock ownership (writers). The default value is twice the number of online CPUs.

nreaders_stress
Number of kernel threads that will stress shared lock ownership (readers). The default is the same amount of writer locks. If the user did not specify nwriters_stress, then both readers and writers be the amount of online CPUs.

torture_type
Type of lock to torture. By default, only spinlocks will be tortured. This module can torture the following locks, with string values as follows:

- **"lock_busted":**
  Simulates a buggy lock implementation.
- **"spin_lock":**
  spin_lock() and spin_unlock() pairs.

- **“spin_lock_irq”:**
  spin_lock_irq() and spin_unlock_irq() pairs.
- **“rw_lock”:**
  read/write lock() and unlock() rwlock pairs.
- **“rw_lock_irq”:**
  read/write lock_irq() and unlock_irq() rwlock pairs.
- **“mutex_lock”:**
  mutex_lock() and mutex_unlock() pairs.
- **“rtmutex_lock”:**
  rtmutex_lock() and rtmutex_unlock() pairs. Kernel must have CONFIG_RT_MUTEXES=y.
- **“rwsem_lock”:**
  read/write down() and up() semaphore pairs.

4.2.2 Torture-framework (RCU + locking)

shutdown_secs
The number of seconds to run the test before terminating the test and powering off the system. The default is zero, which disables test termination and system shutdown. This capability is useful for automated testing.

onoff_interval
The number of seconds between each attempt to execute a randomly selected CPU-hotplug operation. Defaults to zero, which disables CPU hotplugging. In CONFIG_HOTPLUG_CPU=n kernels, locktorture will silently refuse to do any CPU-hotplug operations regardless of what value is specified for onoff_interval.

onoff_holdoff
The number of seconds to wait until starting CPU-hotplug operations. This would normally only be used when locktorture was built into the kernel and started automatically at boot time, in which case it is useful in order to avoid confusing boot-time code with CPUs coming and going. This parameter is only useful if CONFIG_HOTPLUG_CPU is enabled.

stat_interval
Number of seconds between statistics-related printk()s. By default, locktorture will report stats every 60 seconds. Setting the interval to zero causes the statistics to be printed -only- when the module is unloaded.

stutter
The length of time to run the test before pausing for this same period of time. Defaults to “stutter=5”, so as to run and pause for (roughly) five-second intervals. Specifying “stutter=0” causes the test to run continuously without pausing.

shuffle_interval
The number of seconds to keep the test threads affinitized to a particular subset of the CPUs, defaults to 3 seconds. Used in conjunction with test_no_idle_hz.

verbose
Enable verbose debugging printing, via printk(). Enabled by default. This extra information is mostly related to high-level errors and reports from the main ‘torture’ framework.

4.3 Statistics

Statistics are printed in the following format:

```
spin_lock-torture: Writes:  Total: 93746064  Max/Min: 0/0   Fail: 0
   (A)                   (B)            (C)            (D)          (E)

(A): Lock type that is being tortured -- torture_type parameter.

(B): Number of writer lock acquisitions. If dealing with a read/write
     primitive a second "Reads" statistics line is printed.

(C): Number of times the lock was acquired.

(D): Min and max number of times threads failed to acquire the lock.

(E): true/false values if there were errors acquiring the lock. This should
     -only- be positive if there is a bug in the locking primitive's
     implementation. Otherwise a lock should never fail (i.e., spin_lock()).
     Of course, the same applies for (C), above. A dummy example of this is
     the "lock_busted" type.
```

4.4 Usage

The following script may be used to torture locks:

```
#!/bin/sh

modprobe locktorture
sleep 3600
rmmod locktorture
dmesg | grep torture:
```

The output can be manually inspected for the error flag of “!!!”. One could of course create a more elaborate script that automatically checked for such errors. The “rmmod” command forces a “SUCCESS”, “FAILURE”, or “RCU_HOTPLUG” indication to be printk()ed. The first two are self-explanatory, while the last indicates that while there were no locking failures, CPU-hotplug problems were detected.

Also see: Documentation/RCU/torture.rst

CHAPTER

FIVE

GENERIC MUTEX SUBSYSTEM

started by Ingo Molnar <mingo@redhat.com>

updated by Davidlohr Bueso <davidlohr@hp.com>

5.1 What are mutexes?

In the Linux kernel, mutexes refer to a particular locking primitive that enforces serialization on shared memory systems, and not only to the generic term referring to 'mutual exclusion' found in academia or similar theoretical text books. Mutexes are sleeping locks which behave similarly to binary semaphores, and were introduced in 2006[1] as an alternative to these. This new data structure provided a number of advantages, including simpler interfaces, and at that time smaller code (see Disadvantages).

[1] https://lwn.net/Articles/164802/

5.2 Implementation

Mutexes are represented by 'struct mutex', defined in include/linux/mutex.h and implemented in kernel/locking/mutex.c. These locks use an atomic variable (->owner) to keep track of the lock state during its lifetime. Field owner actually contains *struct task_struct* * to the current lock owner and it is therefore NULL if not currently owned. Since task_struct pointers are aligned to at least L1_CACHE_BYTES, low bits (3) are used to store extra state (e.g., if waiter list is non-empty). In its most basic form it also includes a wait-queue and a spinlock that serializes access to it. Furthermore, CONFIG_MUTEX_SPIN_ON_OWNER=y systems use a spinner MCS lock (->osq), described below in (ii).

When acquiring a mutex, there are three possible paths that can be taken, depending on the state of the lock:

(i) fastpath: tries to atomically acquire the lock by cmpxchg()ing the owner with the current task. This only works in the uncontended case (cmpxchg() checks against 0UL, so all 3 state bits above have to be 0). If the lock is contended it goes to the next possible path.

(ii) midpath: aka optimistic spinning, tries to spin for acquisition while the lock owner is running and there are no other tasks ready to run that have higher priority (need_resched). The rationale is that if the lock owner is running, it is likely to release the lock soon. The mutex spinners are queued up using MCS lock so that only one spinner can compete for the mutex.

The MCS lock (proposed by Mellor-Crummey and Scott) is a simple spinlock with the desirable properties of being fair and with each cpu trying to acquire the lock spinning on a local variable. It avoids expensive cacheline bouncing that common test-and-set spinlock implementations incur. An MCS-like lock is specially tailored for optimistic spinning for sleeping lock implementation. An important feature of the customized MCS lock is that it has the extra property that spinners are able to exit the MCS spinlock queue when they need to reschedule. This further helps avoid situations where MCS spinners that need to reschedule would continue waiting to spin on mutex owner, only to go directly to slowpath upon obtaining the MCS lock.

(iii) slowpath: last resort, if the lock is still unable to be acquired, the task is added to the wait-queue and sleeps until woken up by the unlock path. Under normal circumstances it blocks as TASK_UNINTERRUPTIBLE.

While formally kernel mutexes are sleepable locks, it is path (ii) that makes them more practically a hybrid type. By simply not interrupting a task and busy-waiting for a few cycles instead of immediately sleeping, the performance of this lock has been seen to significantly improve a number of workloads. Note that this technique is also used for rw-semaphores.

5.3 Semantics

The mutex subsystem checks and enforces the following rules:

- Only one task can hold the mutex at a time.
- Only the owner can unlock the mutex.
- Multiple unlocks are not permitted.
- Recursive locking/unlocking is not permitted.
- A mutex must only be initialized via the API (see below).
- A task may not exit with a mutex held.
- Memory areas where held locks reside must not be freed.
- Held mutexes must not be reinitialized.
- Mutexes may not be used in hardware or software interrupt contexts such as tasklets and timers.

These semantics are fully enforced when CONFIG DEBUG_MUTEXES is enabled. In addition, the mutex debugging code also implements a number of other features that make lock debugging easier and faster:

- Uses symbolic names of mutexes, whenever they are printed in debug output.
- Point-of-acquire tracking, symbolic lookup of function names, list of all locks held in the system, printout of them.
- Owner tracking.
- Detects self-recursing locks and prints out all relevant info.
- Detects multi-task circular deadlocks and prints out all affected locks and tasks (and only those tasks).

5.4 Interfaces

Statically define the mutex:

```
DEFINE_MUTEX(name);
```

Dynamically initialize the mutex:

```
mutex_init(mutex);
```

Acquire the mutex, uninterruptible:

```
void mutex_lock(struct mutex *lock);
void mutex_lock_nested(struct mutex *lock, unsigned int subclass);
int  mutex_trylock(struct mutex *lock);
```

Acquire the mutex, interruptible:

```
int mutex_lock_interruptible_nested(struct mutex *lock,
                                    unsigned int subclass);
int mutex_lock_interruptible(struct mutex *lock);
```

Acquire the mutex, interruptible, if dec to 0:

```
int atomic_dec_and_mutex_lock(atomic_t *cnt, struct mutex *lock);
```

Unlock the mutex:

```
void mutex_unlock(struct mutex *lock);
```

Test if the mutex is taken:

```
int mutex_is_locked(struct mutex *lock);
```

5.5 Disadvantages

Unlike its original design and purpose, 'struct mutex' is among the largest locks in the kernel. E.g: on x86-64 it is 32 bytes, where 'struct semaphore' is 24 bytes and rw_semaphore is 40 bytes. Larger structure sizes mean more CPU cache and memory footprint.

5.6 When to use mutexes

Unless the strict semantics of mutexes are unsuitable and/or the critical region prevents the lock from being shared, always prefer them to any other locking primitive.

CHAPTER

SIX

RT-MUTEX IMPLEMENTATION DESIGN

This document tries to describe the design of the rtmutex.c implementation. It doesn't describe the reasons why rtmutex.c exists. For that please see *RT-mutex subsystem with PI support*. Although this document does explain problems that happen without this code, but that is in the concept to understand what the code actually is doing.

The goal of this document is to help others understand the priority inheritance (PI) algorithm that is used, as well as reasons for the decisions that were made to implement PI in the manner that was done.

6.1 Unbounded Priority Inversion

Priority inversion is when a lower priority process executes while a higher priority process wants to run. This happens for several reasons, and most of the time it can't be helped. Anytime a high priority process wants to use a resource that a lower priority process has (a mutex for example), the high priority process must wait until the lower priority process is done with the resource. This is a priority inversion. What we want to prevent is something called unbounded priority inversion. That is when the high priority process is prevented from running by a lower priority process for an undetermined amount of time.

The classic example of unbounded priority inversion is where you have three processes, let's call them processes A, B, and C, where A is the highest priority process, C is the lowest, and B is in between. A tries to grab a lock that C owns and must wait and lets C run to release the lock. But in the meantime, B executes, and since B is of a higher priority than C, it preempts C, but by doing so, it is in fact preempting A which is a higher priority process. Now there's no way of knowing how long A will be sleeping waiting for C to release the lock, because for all we know, B is a CPU hog and will never give C a chance to release the lock. This is called unbounded priority inversion.

Here's a little ASCII art to show the problem:

```
   grab lock L1 (owned by C)
     |
A ---+
        C preempted by B
          |
C    +----+
```

```
B         +-------->
                B now keeps A from running.
```

6.2 Priority Inheritance (PI)

There are several ways to solve this issue, but other ways are out of scope for this document. Here we only discuss PI.

PI is where a process inherits the priority of another process if the other process blocks on a lock owned by the current process. To make this easier to understand, let's use the previous example, with processes A, B, and C again.

This time, when A blocks on the lock owned by C, C would inherit the priority of A. So now if B becomes runnable, it would not preempt C, since C now has the high priority of A. As soon as C releases the lock, it loses its inherited priority, and A then can continue with the resource that C had.

6.3 Terminology

Here I explain some terminology that is used in this document to help describe the design that is used to implement PI.

PI chain

- The PI chain is an ordered series of locks and processes that cause processes to inherit priorities from a previous process that is blocked on one of its locks. This is described in more detail later in this document.

mutex

- In this document, to differentiate from locks that implement PI and spin locks that are used in the PI code, from now on the PI locks will be called a mutex.

lock

- In this document from now on, I will use the term lock when referring to spin locks that are used to protect parts of the PI algorithm. These locks disable preemption for UP (when CONFIG_PREEMPT is enabled) and on SMP prevents multiple CPUs from entering critical sections simultaneously.

spin lock

- Same as lock above.

waiter

- A waiter is a struct that is stored on the stack of a blocked process. Since the scope of the waiter is within the code for a process being blocked on the mutex, it is fine to allocate the waiter on the process's stack (local variable). This structure holds a pointer to the task, as well as the mutex that the task is blocked on. It also has rbtree node structures to place the task in the waiters rbtree of a mutex as well as the pi_waiters rbtree of a mutex owner task (described below).

waiter is sometimes used in reference to the task that is waiting on a mutex. This is the same as waiter->task.

waiters

- A list of processes that are blocked on a mutex.

top waiter

- The highest priority process waiting on a specific mutex.

top pi waiter

- The highest priority process waiting on one of the mutexes that a specific process owns.

Note:
task and process are used interchangeably in this document, mostly to differentiate between two processes that are being described together.

6.4 PI chain

The PI chain is a list of processes and mutexes that may cause priority inheritance to take place. Multiple chains may converge, but a chain would never diverge, since a process can't be blocked on more than one mutex at a time.

Example:

```
Process:  A, B, C, D, E
Mutexes:  L1, L2, L3, L4

A owns: L1
        B blocked on L1
        B owns L2
               C blocked on L2
               C owns L3
                      D blocked on L3
                      D owns L4
                             E blocked on L4
```

The chain would be:

```
E->L4->D->L3->C->L2->B->L1->A
```

To show where two chains merge, we could add another process F and another mutex L5 where B owns L5 and F is blocked on mutex L5.

The chain for F would be:

```
F->L5->B->L1->A
```

Since a process may own more than one mutex, but never be blocked on more than one, the chains merge.

Here we show both chains:

```
E->L4->D->L3->C->L2-+
                    |
                    +->B->L1->A
                    |
               F->L5-+
```

For PI to work, the processes at the right end of these chains (or we may also call it the Top of the chain) must be equal to or higher in priority than the processes to the left or below in the chain.

Also since a mutex may have more than one process blocked on it, we can have multiple chains merge at mutexes. If we add another process G that is blocked on mutex L2:

```
G->L2->B->L1->A
```

And once again, to show how this can grow I will show the merging chains again:

```
E->L4->D->L3->C-+
                |
                +->L2-+
                |     |
              G-+     +->B->L1->A
                      |
                F->L5-+
```

If process G has the highest priority in the chain, then all the tasks up the chain (A and B in this example), must have their priorities increased to that of G.

6.5 Mutex Waiters Tree

Every mutex keeps track of all the waiters that are blocked on itself. The mutex has a rbtree to store these waiters by priority. This tree is protected by a spin lock that is located in the struct of the mutex. This lock is called wait_lock.

6.6 Task PI Tree

To keep track of the PI chains, each process has its own PI rbtree. This is a tree of all top waiters of the mutexes that are owned by the process. Note that this tree only holds the top waiters and not all waiters that are blocked on mutexes owned by the process.

The top of the task's PI tree is always the highest priority task that is waiting on a mutex that is owned by the task. So if the task has inherited a priority, it will always be the priority of the task that is at the top of this tree.

This tree is stored in the task structure of a process as a rbtree called pi_waiters. It is protected by a spin lock also in the task structure, called pi_lock. This lock may also be taken in interrupt context, so when locking the pi_lock, interrupts must be disabled.

6.7 Depth of the PI Chain

The maximum depth of the PI chain is not dynamic, and could actually be defined. But is very complex to figure it out, since it depends on all the nesting of mutexes. Let's look at the example where we have 3 mutexes, L1, L2, and L3, and four separate functions func1, func2, func3 and func4. The following shows a locking order of L1->L2->L3, but may not actually be directly nested that way:

```
void func1(void)
{
      mutex_lock(L1);

      /* do anything */

      mutex_unlock(L1);
}

void func2(void)
{
      mutex_lock(L1);
      mutex_lock(L2);

      /* do something */

      mutex_unlock(L2);
      mutex_unlock(L1);
}

void func3(void)
{
      mutex_lock(L2);
      mutex_lock(L3);

      /* do something else */

      mutex_unlock(L3);
      mutex_unlock(L2);
}

void func4(void)
{
      mutex_lock(L3);

      /* do something again */

      mutex_unlock(L3);
}
```

Now we add 4 processes that run each of these functions separately. Processes A, B, C, and D which run functions func1, func2, func3 and func4 respectively, and such that D runs first and A last. With D being preempted in func4 in the "do something again" area, we have a locking

that follows:

```
D owns L3
       C blocked on L3
       C owns L2
              B blocked on L2
              B owns L1
                     A blocked on L1

And thus we have the chain A->L1->B->L2->C->L3->D.
```

This gives us a PI depth of 4 (four processes), but looking at any of the functions individually, it seems as though they only have at most a locking depth of two. So, although the locking depth is defined at compile time, it still is very difficult to find the possibilities of that depth.

Now since mutexes can be defined by user-land applications, we don't want a DOS type of application that nests large amounts of mutexes to create a large PI chain, and have the code holding spin locks while looking at a large amount of data. So to prevent this, the implementation not only implements a maximum lock depth, but also only holds at most two different locks at a time, as it walks the PI chain. More about this below.

6.8 Mutex owner and flags

The mutex structure contains a pointer to the owner of the mutex. If the mutex is not owned, this owner is set to NULL. Since all architectures have the task structure on at least a two byte alignment (and if this is not true, the rtmutex.c code will be broken!), this allows for the least significant bit to be used as a flag. Bit 0 is used as the "Has Waiters" flag. It's set whenever there are waiters on a mutex.

See *RT-mutex subsystem with PI support* for further details.

6.9 cmpxchg Tricks

Some architectures implement an atomic cmpxchg (Compare and Exchange). This is used (when applicable) to keep the fast path of grabbing and releasing mutexes short.

cmpxchg is basically the following function performed atomically:

```
unsigned long _cmpxchg(unsigned long *A, unsigned long *B, unsigned long *C)
{
      unsigned long T = *A;
      if (*A == *B) {
              *A = *C;
      }
      return T;
}
#define cmpxchg(a,b,c) _cmpxchg(&a,&b,&c)
```

This is really nice to have, since it allows you to only update a variable if the variable is what you expect it to be. You know if it succeeded if the return value (the old value of A) is equal to B.

The macro rt_mutex_cmpxchg is used to try to lock and unlock mutexes. If the architecture does not support CMPXCHG, then this macro is simply set to fail every time. But if CMPXCHG is supported, then this will help out extremely to keep the fast path short.

The use of rt_mutex_cmpxchg with the flags in the owner field help optimize the system for architectures that support it. This will also be explained later in this document.

6.10 Priority adjustments

The implementation of the PI code in rtmutex.c has several places that a process must adjust its priority. With the help of the pi_waiters of a process this is rather easy to know what needs to be adjusted.

The functions implementing the task adjustments are rt_mutex_adjust_prio and rt_mutex_setprio. rt_mutex_setprio is only used in rt_mutex_adjust_prio.

rt_mutex_adjust_prio examines the priority of the task, and the highest priority process that is waiting any of mutexes owned by the task. Since the pi_waiters of a task holds an order by priority of all the top waiters of all the mutexes that the task owns, we simply need to compare the top pi waiter to its own normal/deadline priority and take the higher one. Then rt_mutex_setprio is called to adjust the priority of the task to the new priority. Note that rt_mutex_setprio is defined in kernel/sched/core.c to implement the actual change in priority.

Note:

For the “prio” field in task_struct, the lower the number, the higher the priority. A “prio” of 5 is of higher priority than a “prio” of 10.

It is interesting to note that rt_mutex_adjust_prio can either increase or decrease the priority of the task. In the case that a higher priority process has just blocked on a mutex owned by the task, rt_mutex_adjust_prio would increase/boost the task’s priority. But if a higher priority task were for some reason to leave the mutex (timeout or signal), this same function would decrease/unboost the priority of the task. That is because the pi_waiters always contains the highest priority task that is waiting on a mutex owned by the task, so we only need to compare the priority of that top pi waiter to the normal priority of the given task.

6.11 High level overview of the PI chain walk

The PI chain walk is implemented by the function rt_mutex_adjust_prio_chain.

The implementation has gone through several iterations, and has ended up with what we believe is the best. It walks the PI chain by only grabbing at most two locks at a time, and is very efficient.

The rt_mutex_adjust_prio_chain can be used either to boost or lower process priorities.

rt_mutex_adjust_prio_chain is called with a task to be checked for PI (de)boosting (the owner of a mutex that a process is blocking on), a flag to check for deadlocking, the mutex that the task owns, a pointer to a waiter that is the process’s waiter struct that is blocked on the mutex (although this parameter may be NULL for deboosting), a pointer to the mutex on which the task is blocked, and a top_task as the top waiter of the mutex.

For this explanation, I will not mention deadlock detection. This explanation will try to stay at a high level.

When this function is called, there are no locks held. That also means that the state of the owner and lock can change when entered into this function.

Before this function is called, the task has already had rt_mutex_adjust_prio performed on it. This means that the task is set to the priority that it should be at, but the rbtree nodes of the task's waiter have not been updated with the new priorities, and this task may not be in the proper locations in the pi_waiters and waiters trees that the task is blocked on. This function solves all that.

The main operation of this function is summarized by Thomas Gleixner in rtmutex.c. See the 'Chain walk basics and protection scope' comment for further details.

6.12 Taking of a mutex (The walk through)

OK, now let's take a look at the detailed walk through of what happens when taking a mutex.

The first thing that is tried is the fast taking of the mutex. This is done when we have CMPXCHG enabled (otherwise the fast taking automatically fails). Only when the owner field of the mutex is NULL can the lock be taken with the CMPXCHG and nothing else needs to be done.

If there is contention on the lock, we go about the slow path (rt_mutex_slowlock).

The slow path function is where the task's waiter structure is created on the stack. This is because the waiter structure is only needed for the scope of this function. The waiter structure holds the nodes to store the task on the waiters tree of the mutex, and if need be, the pi_waiters tree of the owner.

The wait_lock of the mutex is taken since the slow path of unlocking the mutex also takes this lock.

We then call try_to_take_rt_mutex. This is where the architecture that does not implement CMPXCHG would always grab the lock (if there's no contention).

try_to_take_rt_mutex is used every time the task tries to grab a mutex in the slow path. The first thing that is done here is an atomic setting of the "Has Waiters" flag of the mutex's owner field. By setting this flag now, the current owner of the mutex being contended for can't release the mutex without going into the slow unlock path, and it would then need to grab the wait_lock, which this code currently holds. So setting the "Has Waiters" flag forces the current owner to synchronize with this code.

The lock is taken if the following are true:

1) The lock has no owner

2) The current task is the highest priority against all other waiters of the lock

If the task succeeds to acquire the lock, then the task is set as the owner of the lock, and if the lock still has waiters, the top_waiter (highest priority task waiting on the lock) is added to this task's pi_waiters tree.

If the lock is not taken by try_to_take_rt_mutex(), then the task_blocks_on_rt_mutex() function is called. This will add the task to the lock's waiter tree and propagate the pi chain of the lock as well as the lock's owner's pi_waiters tree. This is described in the next section.

6.13 Task blocks on mutex

The accounting of a mutex and process is done with the waiter structure of the process. The “task” field is set to the process, and the “lock” field to the mutex. The rbtree node of waiter are initialized to the processes current priority.

Since the wait_lock was taken at the entry of the slow lock, we can safely add the waiter to the task waiter tree. If the current process is the highest priority process currently waiting on this mutex, then we remove the previous top waiter process (if it exists) from the pi_waiters of the owner, and add the current process to that tree. Since the pi_waiter of the owner has changed, we call rt_mutex_adjust_prio on the owner to see if the owner should adjust its priority accordingly.

If the owner is also blocked on a lock, and had its pi_waiters changed (or deadlock checking is on), we unlock the wait_lock of the mutex and go ahead and run rt_mutex_adjust_prio_chain on the owner, as described earlier.

Now all locks are released, and if the current process is still blocked on a mutex (waiter “task” field is not NULL), then we go to sleep (call schedule).

6.14 Waking up in the loop

The task can then wake up for a couple of reasons:

1) The previous lock owner released the lock, and the task now is top_waiter

2) we received a signal or timeout

In both cases, the task will try again to acquire the lock. If it does, then it will take itself off the waiters tree and set itself back to the TASK_RUNNING state.

In first case, if the lock was acquired by another task before this task could get the lock, then it will go back to sleep and wait to be woken again.

The second case is only applicable for tasks that are grabbing a mutex that can wake up before getting the lock, either due to a signal or a timeout (i.e. rt_mutex_timed_futex_lock()). When woken, it will try to take the lock again, if it succeeds, then the task will return with the lock held, otherwise it will return with -EINTR if the task was woken by a signal, or -ETIMEDOUT if it timed out.

6.15 Unlocking the Mutex

The unlocking of a mutex also has a fast path for those architectures with CMPXCHG. Since the taking of a mutex on contention always sets the “Has Waiters” flag of the mutex’s owner, we use this to know if we need to take the slow path when unlocking the mutex. If the mutex doesn’t have any waiters, the owner field of the mutex would equal the current process and the mutex can be unlocked by just replacing the owner field with NULL.

If the owner field has the “Has Waiters” bit set (or CMPXCHG is not available), the slow unlock path is taken.

The first thing done in the slow unlock path is to take the wait_lock of the mutex. This synchronizes the locking and unlocking of the mutex.

A check is made to see if the mutex has waiters or not. On architectures that do not have CMPXCHG, this is the location that the owner of the mutex will determine if a waiter needs to be awoken or not. On architectures that do have CMPXCHG, that check is done in the fast path, but it is still needed in the slow path too. If a waiter of a mutex woke up because of a signal or timeout between the time the owner failed the fast path CMPXCHG check and the grabbing of the wait_lock, the mutex may not have any waiters, thus the owner still needs to make this check. If there are no waiters then the mutex owner field is set to NULL, the wait_lock is released and nothing more is needed.

If there are waiters, then we need to wake one up.

On the wake up code, the pi_lock of the current owner is taken. The top waiter of the lock is found and removed from the waiters tree of the mutex as well as the pi_waiters tree of the current owner. The “Has Waiters” bit is marked to prevent lower priority tasks from stealing the lock.

Finally we unlock the pi_lock of the pending owner and wake it up.

6.16 Contact

For updates on this document, please email Steven Rostedt <rostedt@goodmis.org>

6.17 Credits

Author: Steven Rostedt <rostedt@goodmis.org>

Updated: Alex Shi <alex.shi@linaro.org> - 7/6/2017

Original Reviewers:
Ingo Molnar, Thomas Gleixner, Thomas Duetsch, and Randy Dunlap

Update (7/6/2017) Reviewers: Steven Rostedt and Sebastian Siewior

6.18 Updates

This document was originally written for 2.6.17-rc3-mm1 was updated on 4.12

CHAPTER

SEVEN

RT-MUTEX SUBSYSTEM WITH PI SUPPORT

RT-mutexes with priority inheritance are used to support PI-futexes, which enable pthread_mutex_t priority inheritance attributes (PTHREAD_PRIO_INHERIT). [See *Lightweight PI-futexes* for more details about PI-futexes.]

This technology was developed in the -rt tree and streamlined for pthread_mutex support.

7.1 Basic principles:

RT-mutexes extend the semantics of simple mutexes by the priority inheritance protocol.

A low priority owner of a rt-mutex inherits the priority of a higher priority waiter until the rt-mutex is released. If the temporarily boosted owner blocks on a rt-mutex itself it propagates the priority boosting to the owner of the other rt_mutex it gets blocked on. The priority boosting is immediately removed once the rt_mutex has been unlocked.

This approach allows us to shorten the block of high-prio tasks on mutexes which protect shared resources. Priority inheritance is not a magic bullet for poorly designed applications, but it allows well-designed applications to use userspace locks in critical parts of an high priority thread, without losing determinism.

The enqueueing of the waiters into the rtmutex waiter tree is done in priority order. For same priorities FIFO order is chosen. For each rtmutex, only the top priority waiter is enqueued into the owner's priority waiters tree. This tree too queues in priority order. Whenever the top priority waiter of a task changes (for example it timed out or got a signal), the priority of the owner task is readjusted. The priority enqueueing is handled by "pi_waiters".

RT-mutexes are optimized for fastpath operations and have no internal locking overhead when locking an uncontended mutex or unlocking a mutex without waiters. The optimized fastpath operations require cmpxchg support. [If that is not available then the rt-mutex internal spinlock is used]

The state of the rt-mutex is tracked via the owner field of the rt-mutex structure:

lock->owner holds the task_struct pointer of the owner. Bit 0 is used to keep track of the "lock has waiters" state:

owner	bit0	Notes
NULL	0	lock is free (fast acquire possible)
NULL	1	lock is free and has waiters and the top waiter is going to take the lock[1]
taskpointer	0	lock is held (fast release possible)
taskpointer	1	lock is held and has waiters[2]

The fast atomic compare exchange based acquire and release is only possible when bit 0 of lock->owner is 0.

BTW, there is still technically a “Pending Owner”, it’s just not called that anymore. The pending owner happens to be the top_waiter of a lock that has no owner and has been woken up to grab the lock.

[1] It also can be a transitional state when grabbing the lock with ->wait_lock is held. To prevent any fast path cmpxchg to the lock, we need to set the bit0 before looking at the lock, and the owner may be NULL in this small time, hence this can be a transitional state.

[2] There is a small time when bit 0 is set but there are no waiters. This can happen when grabbing the lock in the slow path. To prevent a cmpxchg of the owner releasing the lock, we need to set this bit before looking at the lock.

CHAPTER

EIGHT

SEQUENCE COUNTERS AND SEQUENTIAL LOCKS

8.1 Introduction

Sequence counters are a reader-writer consistency mechanism with lockless readers (read-only retry loops), and no writer starvation. They are used for data that's rarely written to (e.g. system time), where the reader wants a consistent set of information and is willing to retry if that information changes.

A data set is consistent when the sequence count at the beginning of the read side critical section is even and the same sequence count value is read again at the end of the critical section. The data in the set must be copied out inside the read side critical section. If the sequence count has changed between the start and the end of the critical section, the reader must retry.

Writers increment the sequence count at the start and the end of their critical section. After starting the critical section the sequence count is odd and indicates to the readers that an update is in progress. At the end of the write side critical section the sequence count becomes even again which lets readers make progress.

A sequence counter write side critical section must never be preempted or interrupted by read side sections. Otherwise the reader will spin for the entire scheduler tick due to the odd sequence count value and the interrupted writer. If that reader belongs to a real-time scheduling class, it can spin forever and the kernel will livelock.

This mechanism cannot be used if the protected data contains pointers, as the writer can invalidate a pointer that the reader is following.

8.2 Sequence counters (`seqcount_t`)

This is the raw counting mechanism, which does not protect against multiple writers. Write side critical sections must thus be serialized by an external lock.

If the write serialization primitive is not implicitly disabling preemption, preemption must be explicitly disabled before entering the write side section. If the read section can be invoked from hardirq or softirq contexts, interrupts or bottom halves must also be respectively disabled before entering the write section.

If it's desired to automatically handle the sequence counter requirements of writer serialization and non-preemptibility, use *Sequential locks (seqlock_t)* instead.

Initialization:

```
/* dynamic */
seqcount_t foo_seqcount;
seqcount_init(&foo_seqcount);

/* static */
static seqcount_t foo_seqcount = SEQCNT_ZERO(foo_seqcount);

/* C99 struct init */
struct {
        .seq   = SEQCNT_ZERO(foo.seq),
} foo;
```

Write path:

```
/* Serialized context with disabled preemption */

write_seqcount_begin(&foo_seqcount);

/* ... [[write-side critical section]] ... */

write_seqcount_end(&foo_seqcount);
```

Read path:

```
do {
        seq = read_seqcount_begin(&foo_seqcount);

        /* ... [[read-side critical section]] ... */

} while (read_seqcount_retry(&foo_seqcount, seq));
```

8.2.1 Sequence counters with associated locks (`seqcount_LOCKNAME_t`)

As discussed at *Sequence counters (seqcount_t)*, sequence count write side critical sections must be serialized and non-preemptible. This variant of sequence counters associate the lock used for writer serialization at initialization time, which enables lockdep to validate that the write side critical sections are properly serialized.

This lock association is a NOOP if lockdep is disabled and has neither storage nor runtime overhead. If lockdep is enabled, the lock pointer is stored in struct seqcount and lockdep's "lock is held" assertions are injected at the beginning of the write side critical section to validate that it is properly protected.

For lock types which do not implicitly disable preemption, preemption protection is enforced in the write side function.

The following sequence counters with associated locks are defined:

- `seqcount_spinlock_t`
- `seqcount_raw_spinlock_t`
- `seqcount_rwlock_t`

- seqcount_mutex_t
- seqcount_ww_mutex_t

The sequence counter read and write APIs can take either a plain seqcount_t or any of the seqcount_LOCKNAME_t variants above.

Initialization (replace “LOCKNAME” with one of the supported locks):

```
/* dynamic */
seqcount_LOCKNAME_t foo_seqcount;
seqcount_LOCKNAME_init(&foo_seqcount, &lock);

/* static */
static seqcount_LOCKNAME_t foo_seqcount =
        SEQCNT_LOCKNAME_ZERO(foo_seqcount, &lock);

/* C99 struct init */
struct {
        .seq   = SEQCNT_LOCKNAME_ZERO(foo.seq, &lock),
} foo;
```

Write path: same as in *Sequence counters (seqcount_t)*, while running from a context with the associated write serialization lock acquired.

Read path: same as in *Sequence counters (seqcount_t)*.

8.2.2 Latch sequence counters (seqcount_latch_t)

Latch sequence counters are a multiversion concurrency control mechanism where the embedded seqcount_t counter even/odd value is used to switch between two copies of protected data. This allows the sequence counter read path to safely interrupt its own write side critical section.

Use seqcount_latch_t when the write side sections cannot be protected from interruption by readers. This is typically the case when the read side can be invoked from NMI handlers.

Check `raw_write_seqcount_latch()` for more information.

8.3 Sequential locks (seqlock_t)

This contains the *Sequence counters (seqcount_t)* mechanism earlier discussed, plus an embedded spinlock for writer serialization and non-preemptibility.

If the read side section can be invoked from hardirq or softirq context, use the write side function variants which disable interrupts or bottom halves respectively.

Initialization:

```
/* dynamic */
seqlock_t foo_seqlock;
seqlock_init(&foo_seqlock);

/* static */
```

```
static DEFINE_SEQLOCK(foo_seqlock);

/* C99 struct init */
struct {
        .seql   = __SEQLOCK_UNLOCKED(foo.seql)
} foo;
```

Write path:

```
write_seqlock(&foo_seqlock);

/* ... [[write-side critical section]] ... */

write_sequnlock(&foo_seqlock);
```

Read path, three categories:

1. Normal Sequence readers which never block a writer but they must retry if a writer is in progress by detecting change in the sequence number. Writers do not wait for a sequence reader:

   ```
   do {
           seq = read_seqbegin(&foo_seqlock);

           /* ... [[read-side critical section]] ... */

   } while (read_seqretry(&foo_seqlock, seq));
   ```

2. Locking readers which will wait if a writer or another locking reader is in progress. A locking reader in progress will also block a writer from entering its critical section. This read lock is exclusive. Unlike rwlock_t, only one locking reader can acquire it:

   ```
   read_seqlock_excl(&foo_seqlock);

   /* ... [[read-side critical section]] ... */

   read_sequnlock_excl(&foo_seqlock);
   ```

3. Conditional lockless reader (as in 1), or locking reader (as in 2), according to a passed marker. This is used to avoid lockless readers starvation (too much retry loops) in case of a sharp spike in write activity. First, a lockless read is tried (even marker passed). If that trial fails (odd sequence counter is returned, which is used as the next iteration marker), the lockless read is transformed to a full locking read and no retry loop is necessary:

   ```
   /* marker; even initialization */
   int seq = 0;
   do {
           read_seqbegin_or_lock(&foo_seqlock, &seq);

           /* ... [[read-side critical section]] ... */

   } while (need_seqretry(&foo_seqlock, seq));
   ```

```
done_seqretry(&foo_seqlock, seq);
```

8.4 API documentation

seqcount_init

seqcount_init (s)

runtime initializer for seqcount_t

Parameters

s
Pointer to the seqcount_t instance

SEQCNT_ZERO

SEQCNT_ZERO (name)

static initializer for seqcount_t

Parameters

name
Name of the seqcount_t instance

__read_seqcount_begin

__read_seqcount_begin (s)

begin a seqcount_t read section w/o barrier

Parameters

s
Pointer to seqcount_t or any of the seqcount_LOCKNAME_t variants

Description

__read_seqcount_begin is like read_seqcount_begin, but has no smp_rmb() barrier. Callers should ensure that smp_rmb() or equivalent ordering is provided before actually loading any of the variables that are to be protected in this critical section.

Use carefully, only in critical code, and comment how the barrier is provided.

Return

count to be passed to *read_seqcount_retry()*

raw_read_seqcount_begin

raw_read_seqcount_begin (s)

begin a seqcount_t read section w/o lockdep

Parameters

s
Pointer to seqcount_t or any of the seqcount_LOCKNAME_t variants

Return

count to be passed to *read_seqcount_retry()*

read_seqcount_begin

read_seqcount_begin (s)

begin a seqcount_t read critical section

Parameters

s
Pointer to seqcount_t or any of the seqcount_LOCKNAME_t variants

Return

count to be passed to *read_seqcount_retry()*

raw_read_seqcount

raw_read_seqcount (s)

read the raw seqcount_t counter value

Parameters

s
Pointer to seqcount_t or any of the seqcount_LOCKNAME_t variants

Description

raw_read_seqcount opens a read critical section of the given seqcount_t, without any lockdep checking, and without checking or masking the sequence counter LSB. Calling code is responsible for handling that.

Return

count to be passed to *read_seqcount_retry()*

raw_seqcount_begin

raw_seqcount_begin (s)

begin a seqcount_t read critical section w/o lockdep and w/o counter stabilization

Parameters

s
Pointer to seqcount_t or any of the seqcount_LOCKNAME_t variants

Description

raw_seqcount_begin opens a read critical section of the given seqcount_t. Unlike *read_seqcount_begin()*, this function will not wait for the count to stabilize. If a writer is active when it begins, it will fail the *read_seqcount_retry()* at the end of the read critical section instead of stabilizing at the beginning of it.

Use this only in special kernel hot paths where the read section is small and has a high probability of success through other external means. It will save a single branching instruction.

Return

count to be passed to *read_seqcount_retry()*

__read_seqcount_retry

```
__read_seqcount_retry (s, start)
```

end a seqcount_t read section w/o barrier

Parameters

s
Pointer to seqcount_t or any of the seqcount_LOCKNAME_t variants

start
count, from *read_seqcount_begin()*

Description

__read_seqcount_retry is like read_seqcount_retry, but has no smp_rmb() barrier. Callers should ensure that smp_rmb() or equivalent ordering is provided before actually loading any of the variables that are to be protected in this critical section.

Use carefully, only in critical code, and comment how the barrier is provided.

Return

true if a read section retry is required, else false

read_seqcount_retry

```
read_seqcount_retry (s, start)
```

end a seqcount_t read critical section

Parameters

s
Pointer to seqcount_t or any of the seqcount_LOCKNAME_t variants

start
count, from *read_seqcount_begin()*

Description

read_seqcount_retry closes the read critical section of given seqcount_t. If the critical section was invalid, it must be ignored (and typically retried).

Return

true if a read section retry is required, else false

raw_write_seqcount_begin

```
raw_write_seqcount_begin (s)
```

start a seqcount_t write section w/o lockdep

Parameters

s
Pointer to seqcount_t or any of the seqcount_LOCKNAME_t variants

Context

check *write_seqcount_begin()*

raw_write_seqcount_end

raw_write_seqcount_end (s)

end a seqcount_t write section w/o lockdep

Parameters

s
Pointer to seqcount_t or any of the seqcount_LOCKNAME_t variants

Context

check *write_seqcount_end()*

write_seqcount_begin_nested

write_seqcount_begin_nested (s, subclass)

start a seqcount_t write section with custom lockdep nesting level

Parameters

s
Pointer to seqcount_t or any of the seqcount_LOCKNAME_t variants

subclass
lockdep nesting level

Description

See *Runtime locking correctness validator*

Context

check *write_seqcount_begin()*

write_seqcount_begin

write_seqcount_begin (s)

start a seqcount_t write side critical section

Parameters

s
Pointer to seqcount_t or any of the seqcount_LOCKNAME_t variants

Context

sequence counter write side sections must be serialized and non-preemptible. Preemption will be automatically disabled if and only if the seqcount write serialization lock is associated, and preemptible. If readers can be invoked from hardirq or softirq context, interrupts or bottom halves must be respectively disabled.

write_seqcount_end

write_seqcount_end (s)

end a seqcount_t write side critical section

Parameters

s
Pointer to seqcount_t or any of the seqcount_LOCKNAME_t variants

Context

Preemption will be automatically re-enabled if and only if the seqcount write serialization lock is associated, and preemptible.

raw_write_seqcount_barrier

raw_write_seqcount_barrier (s)

do a seqcount_t write barrier

Parameters

s
Pointer to seqcount_t or any of the seqcount_LOCKNAME_t variants

Description

This can be used to provide an ordering guarantee instead of the usual consistency guarantee. It is one wmb cheaper, because it can collapse the two back-to-back wmb()s.

Note that writes surrounding the barrier should be declared atomic (e.g. via WRITE_ONCE): a) to ensure the writes become visible to other threads atomically, avoiding compiler optimizations; b) to document which writes are meant to propagate to the reader critical section. This is necessary because neither writes before and after the barrier are enclosed in a seq-writer critical section that would ensure readers are aware of ongoing writes:

```
seqcount_t seq;
bool X = true, Y = false;

void read(void)
{
        bool x, y;

        do {
                int s = read_seqcount_begin(&seq);

                x = X; y = Y;

        } while (read_seqcount_retry(&seq, s));

        BUG_ON(!x && !y);
}

void write(void)
{
        WRITE_ONCE(Y, true);

        raw_write_seqcount_barrier(seq);

        WRITE_ONCE(X, false);
}
```

write_seqcount_invalidate

write_seqcount_invalidate (s)

invalidate in-progress seqcount_t read side operations

Parameters

s
Pointer to seqcount_t or any of the seqcount_LOCKNAME_t variants

Description

After write_seqcount_invalidate, no seqcount_t read side operations will complete successfully and see data older than this.

SEQCNT_LATCH_ZERO

SEQCNT_LATCH_ZERO (seq_name)

static initializer for seqcount_latch_t

Parameters

seq_name
Name of the seqcount_latch_t instance

seqcount_latch_init

seqcount_latch_init (s)

runtime initializer for seqcount_latch_t

Parameters

s
Pointer to the seqcount_latch_t instance

unsigned **raw_read_seqcount_latch**(const seqcount_latch_t *s)
pick even/odd latch data copy

Parameters

const seqcount_latch_t *s
Pointer to seqcount_latch_t

Description

See *raw_write_seqcount_latch()* for details and a full reader/writer usage example.

Return

sequence counter raw value. Use the lowest bit as an index for picking which data copy to read. The full counter must then be checked with *raw_read_seqcount_latch_retry()*.

int **raw_read_seqcount_latch_retry**(const seqcount_latch_t *s, unsigned start)
end a seqcount_latch_t read section

Parameters

const seqcount_latch_t *s
Pointer to seqcount_latch_t

unsigned start
count, from *raw_read_seqcount_latch()*

Return

true if a read section retry is required, else false

void **raw_write_seqcount_latch**(seqcount_latch_t *s)
redirect latch readers to even/odd copy

Parameters

seqcount_latch_t *s
Pointer to seqcount_latch_t

Description

The latch technique is a multiversion concurrency control method that allows queries during non-atomic modifications. If you can guarantee queries never interrupt the modification -- e.g. the concurrency is strictly between CPUs -- you most likely do not need this.

Where the traditional RCU/lockless data structures rely on atomic modifications to ensure queries observe either the old or the new state the latch allows the same for non-atomic updates. The trade-off is doubling the cost of storage; we have to maintain two copies of the entire data structure.

Very simply put: we first modify one copy and then the other. This ensures there is always one copy in a stable state, ready to give us an answer.

The basic form is a data structure like:

```
struct latch_struct {
        seqcount_latch_t        seq;
        struct data_struct      data[2];
};
```

Where a modification, which is assumed to be externally serialized, does the following:

```
void latch_modify(struct latch_struct *latch, ...)
{
        smp_wmb();      // Ensure that the last data[1] update is visible
        latch->seq.sequence++;
        smp_wmb();      // Ensure that the seqcount update is visible

        modify(latch->data[0], ...);

        smp_wmb();      // Ensure that the data[0] update is visible
        latch->seq.sequence++;
        smp_wmb();      // Ensure that the seqcount update is visible

        modify(latch->data[1], ...);
}
```

The query will have a form like:

```
struct entry *latch_query(struct latch_struct *latch, ...)
{
```

```
        struct entry *entry;
        unsigned seq, idx;

        do {
                seq = raw_read_seqcount_latch(&latch->seq);

                idx = seq & 0x01;
                entry = data_query(latch->data[idx], ...);

        // This includes needed smp_rmb()
        } while (raw_read_seqcount_latch_retry(&latch->seq, seq));

        return entry;
}
```

So during the modification, queries are first redirected to data[1]. Then we modify data[0]. When that is complete, we redirect queries back to data[0] and we can modify data[1].

NOTE2:

When data is a dynamic data structure; one should use regular RCU patterns to manage the lifetimes of the objects within.

NOTE

The non-requirement for atomic modifications does _NOT_ include the publishing of new entries in the case where data is a dynamic data structure.

An iteration might start in data[0] and get suspended long enough to miss an entire modification sequence, once it resumes it might observe the new entry.

seqlock_init

`seqlock_init (sl)`

dynamic initializer for seqlock_t

Parameters

sl
Pointer to the seqlock_t instance

DEFINE_SEQLOCK

`DEFINE_SEQLOCK (sl)`

Define a statically allocated seqlock_t

Parameters

sl
Name of the seqlock_t instance

unsigned **read_seqbegin**(const seqlock_t *sl)
start a seqlock_t read side critical section

Parameters

const seqlock_t *sl
Pointer to seqlock_t

Return

count, to be passed to *read_seqretry()*

unsigned **read_seqretry**(const seqlock_t *sl, unsigned start)
end a seqlock_t read side section

Parameters

const seqlock_t *sl
Pointer to seqlock_t

unsigned start
count, from *read_seqbegin()*

Description

read_seqretry closes the read side critical section of given seqlock_t. If the critical section was invalid, it must be ignored (and typically retried).

Return

true if a read section retry is required, else false

void **write_seqlock**(seqlock_t *sl)
start a seqlock_t write side critical section

Parameters

seqlock_t *sl
Pointer to seqlock_t

Description

write_seqlock opens a write side critical section for the given seqlock_t. It also implicitly acquires the spinlock_t embedded inside that sequential lock. All seqlock_t write side sections are thus automatically serialized and non-preemptible.

Context

if the seqlock_t read section, or other write side critical sections, can be invoked from hardirq or softirq contexts, use the _irqsave or _bh variants of this function instead.

void **write_sequnlock**(seqlock_t *sl)
end a seqlock_t write side critical section

Parameters

seqlock_t *sl
Pointer to seqlock_t

Description

write_sequnlock closes the (serialized and non-preemptible) write side critical section of given seqlock_t.

void **write_seqlock_bh**(seqlock_t *sl)
start a softirqs-disabled seqlock_t write section

Parameters

seqlock_t *sl
Pointer to seqlock_t

Description

_bh variant of *write_seqlock()*. Use only if the read side section, or other write side sections, can be invoked from softirq contexts.

void **write_sequnlock_bh**(seqlock_t *sl)
end a softirqs-disabled seqlock_t write section

Parameters

seqlock_t *sl
Pointer to seqlock_t

Description

write_sequnlock_bh closes the serialized, non-preemptible, and softirqs-disabled, seqlock_t write side critical section opened with *write_seqlock_bh()*.

void **write_seqlock_irq**(seqlock_t *sl)
start a non-interruptible seqlock_t write section

Parameters

seqlock_t *sl
Pointer to seqlock_t

Description

_irq variant of *write_seqlock()*. Use only if the read side section, or other write sections, can be invoked from hardirq contexts.

void **write_sequnlock_irq**(seqlock_t *sl)
end a non-interruptible seqlock_t write section

Parameters

seqlock_t *sl
Pointer to seqlock_t

Description

write_sequnlock_irq closes the serialized and non-interruptible seqlock_t write side section opened with *write_seqlock_irq()*.

write_seqlock_irqsave

write_seqlock_irqsave (lock, flags)
start a non-interruptible seqlock_t write section

Parameters

lock
Pointer to seqlock_t

flags
Stack-allocated storage for saving caller's local interrupt state, to be passed to *write_sequnlock_irqrestore()*.

Description

_irqsave variant of *write_seqlock()*. Use it only if the read side section, or other write sections, can be invoked from hardirq context.

void **write_sequnlock_irqrestore**(seqlock_t *sl, unsigned long flags)
end non-interruptible seqlock_t write section

Parameters

seqlock_t *sl
Pointer to seqlock_t

unsigned long flags
Caller's saved interrupt state, from *write_seqlock_irqsave()*

Description

write_sequnlock_irqrestore closes the serialized and non-interruptible seqlock_t write section previously opened with *write_seqlock_irqsave()*.

void **read_seqlock_excl**(seqlock_t *sl)
begin a seqlock_t locking reader section

Parameters

seqlock_t *sl
Pointer to seqlock_t

Description

read_seqlock_excl opens a seqlock_t locking reader critical section. A locking reader exclusively locks out *both* other writers *and* other locking readers, but it does not update the embedded sequence number.

Locking readers act like a normal spin_lock()/spin_unlock().

The opened read section must be closed with *read_sequnlock_excl()*.

Context

if the seqlock_t write section, *or other read sections*, can be invoked from hardirq or softirq contexts, use the _irqsave or _bh variant of this function instead.

void **read_sequnlock_excl**(seqlock_t *sl)
end a seqlock_t locking reader critical section

Parameters

seqlock_t *sl
Pointer to seqlock_t

void **read_seqlock_excl_bh**(seqlock_t *sl)
start a seqlock_t locking reader section with softirqs disabled

Parameters

seqlock_t *sl
Pointer to seqlock_t

Description

_bh variant of *read_seqlock_excl()*. Use this variant only if the seqlock_t write side section, *or other read sections*, can be invoked from softirq contexts.

void **read_sequnlock_excl_bh**(seqlock_t *sl)
stop a seqlock_t softirq-disabled locking reader section

Parameters

seqlock_t *sl
Pointer to seqlock_t

void **read_seqlock_excl_irq**(seqlock_t *sl)
start a non-interruptible seqlock_t locking reader section

Parameters

seqlock_t *sl
Pointer to seqlock_t

Description

_irq variant of *read_seqlock_excl()*. Use this only if the seqlock_t write side section, *or other read sections*, can be invoked from a hardirq context.

void **read_sequnlock_excl_irq**(seqlock_t *sl)
end an interrupts-disabled seqlock_t locking reader section

Parameters

seqlock_t *sl
Pointer to seqlock_t

read_seqlock_excl_irqsave

`read_seqlock_excl_irqsave (lock, flags)`

start a non-interruptible seqlock_t locking reader section

Parameters

lock
Pointer to seqlock_t

flags
Stack-allocated storage for saving caller's local interrupt state, to be passed to *read_sequnlock_excl_irqrestore()*.

Description

_irqsave variant of *read_seqlock_excl()*. Use this only if the seqlock_t write side section, *or other read sections*, can be invoked from a hardirq context.

void **read_sequnlock_excl_irqrestore**(seqlock_t *sl, unsigned long flags)
end non-interruptible seqlock_t locking reader section

Parameters

seqlock_t *sl
Pointer to seqlock_t

unsigned long flags
Caller saved interrupt state, from *read_seqlock_excl_irqsave()*

void **read_seqbegin_or_lock**(seqlock_t *lock, int *seq)
begin a seqlock_t lockless or locking reader

Parameters

seqlock_t *lock
Pointer to seqlock_t

int *seq
Marker and return parameter. If the passed value is even, the reader will become a *lockless* seqlock_t reader as in *read_seqbegin()*. If the passed value is odd, the reader will become a *locking* reader as in *read_seqlock_excl()*. In the first call to this function, the caller *must* initialize and pass an even value to **seq**; this way, a lockless read can be optimistically tried first.

Description

read_seqbegin_or_lock is an API designed to optimistically try a normal lockless seqlock_t read section first. If an odd counter is found, the lockless read trial has failed, and the next read iteration transforms itself into a full seqlock_t locking reader.

This is typically used to avoid seqlock_t lockless readers starvation (too much retry loops) in the case of a sharp spike in write side activity.

Check *Sequence counters and sequential locks* for template example code.

Context

if the seqlock_t write section, *or other read sections*, can be invoked from hardirq or softirq contexts, use the _irqsave or _bh variant of this function instead.

Return

the encountered sequence counter value, through the **seq** parameter, which is overloaded as a return parameter. This returned value must be checked with *need_seqretry()*. If the read section need to be retried, this returned value must also be passed as the **seq** parameter of the next *read_seqbegin_or_lock()* iteration.

int **need_seqretry**(seqlock_t *lock, int seq)
validate seqlock_t “locking or lockless” read section

Parameters

seqlock_t *lock
Pointer to seqlock_t

int seq
sequence count, from *read_seqbegin_or_lock()*

Return

true if a read section retry is required, false otherwise

void **done_seqretry**(seqlock_t *lock, int seq)
end seqlock_t “locking or lockless” reader section

Parameters

seqlock_t *lock
Pointer to seqlock_t

int seq
count, from *read_seqbegin_or_lock()*

Description

done_seqretry finishes the seqlock_t read side critical section started with *read_seqbegin_or_lock()* and validated by *need_seqretry()*.

unsigned long **read_seqbegin_or_lock_irqsave**(seqlock_t *lock, int *seq)
begin a seqlock_t lockless reader, or a non-interruptible locking reader

Parameters

seqlock_t *lock
Pointer to seqlock_t

int *seq
Marker and return parameter. Check *read_seqbegin_or_lock()*.

Description

This is the _irqsave variant of *read_seqbegin_or_lock()*. Use it only if the seqlock_t write section, *or other read sections*, can be invoked from hardirq context.

Note

Interrupts will be disabled only for "locking reader" mode.

Return

1. The saved local interrupts state in case of a locking reader, to be passed to *done_seqretry_irqrestore()*.
2. The encountered sequence counter value, returned through **seq** overloaded as a return parameter. Check *read_seqbegin_or_lock()*.

void **done_seqretry_irqrestore**(seqlock_t *lock, int seq, unsigned long flags)
end a seqlock_t lockless reader, or a non-interruptible locking reader section

Parameters

seqlock_t *lock
Pointer to seqlock_t

int seq
Count, from *read_seqbegin_or_lock_irqsave()*

unsigned long flags
Caller's saved local interrupt state in case of a locking reader, also from *read_seqbegin_or_lock_irqsave()*

Description

This is the _irqrestore variant of *done_seqretry()*. The read section must've been opened with *read_seqbegin_or_lock_irqsave()*, and validated by *need_seqretry()*.

CHAPTER

NINE

LOCKING LESSONS

9.1 Lesson 1: Spin locks

The most basic primitive for locking is spinlock:

```
static DEFINE_SPINLOCK(xxx_lock);

      unsigned long flags;

      spin_lock_irqsave(&xxx_lock, flags);
      ... critical section here ..
      spin_unlock_irqrestore(&xxx_lock, flags);
```

The above is always safe. It will disable interrupts _locally_, but the spinlock itself will guarantee the global lock, so it will guarantee that there is only one thread-of-control within the region(s) protected by that lock. This works well even under UP also, so the code does _not_ need to worry about UP vs SMP issues: the spinlocks work correctly under both.

> NOTE! Implications of spin_locks for memory are further described in:
>
> Documentation/memory-barriers.txt
>
> (5) ACQUIRE operations.
>
> (6) RELEASE operations.

The above is usually pretty simple (you usually need and want only one spinlock for most things - using more than one spinlock can make things a lot more complex and even slower and is usually worth it only for sequences that you **know** need to be split up: avoid it at all cost if you aren't sure).

This is really the only really hard part about spinlocks: once you start using spinlocks they tend to expand to areas you might not have noticed before, because you have to make sure the spinlocks correctly protect the shared data structures **everywhere** they are used. The spinlocks are most easily added to places that are completely independent of other code (for example, internal driver data structures that nobody else ever touches).

> NOTE! The spin-lock is safe only when you **also** use the lock itself to do locking across CPU's, which implies that EVERYTHING that touches a shared variable has to agree about the spinlock they want to use.

9.2 Lesson 2: reader-writer spinlocks.

If your data accesses have a very natural pattern where you usually tend to mostly read from the shared variables, the reader-writer locks (rw_lock) versions of the spinlocks are sometimes useful. They allow multiple readers to be in the same critical region at once, but if somebody wants to change the variables it has to get an exclusive write lock.

> NOTE! reader-writer locks require more atomic memory operations than simple spinlocks. Unless the reader critical section is long, you are better off just using spinlocks.

The routines look the same as above:

```
rwlock_t xxx_lock = __RW_LOCK_UNLOCKED(xxx_lock);

     unsigned long flags;

     read_lock_irqsave(&xxx_lock, flags);
     .. critical section that only reads the info ...
     read_unlock_irqrestore(&xxx_lock, flags);

     write_lock_irqsave(&xxx_lock, flags);
     .. read and write exclusive access to the info ...
     write_unlock_irqrestore(&xxx_lock, flags);
```

The above kind of lock may be useful for complex data structures like linked lists, especially searching for entries without changing the list itself. The read lock allows many concurrent readers. Anything that **changes** the list will have to get the write lock.

> NOTE! RCU is better for list traversal, but requires careful attention to design detail (see Documentation/RCU/listRCU.rst).

Also, you cannot "upgrade" a read-lock to a write-lock, so if you at _any_ time need to do any changes (even if you don't do it every time), you have to get the write-lock at the very beginning.

> NOTE! We are working hard to remove reader-writer spinlocks in most cases, so please don't add a new one without consensus. (Instead, see Documentation/RCU/rcu.rst for complete information.)

9.3 Lesson 3: spinlocks revisited.

The single spin-lock primitives above are by no means the only ones. They are the most safe ones, and the ones that work under all circumstances, but partly **because** they are safe they are also fairly slow. They are slower than they'd need to be, because they do have to disable interrupts (which is just a single instruction on a x86, but it's an expensive one - and on other architectures it can be worse).

If you have a case where you have to protect a data structure across several CPU's and you want to use spinlocks you can potentially use cheaper versions of the spinlocks. IFF you know that the spinlocks are never used in interrupt handlers, you can use the non-irq versions:

```
spin_lock(&lock);
...
spin_unlock(&lock);
```

(and the equivalent read-write versions too, of course). The spinlock will guarantee the same kind of exclusive access, and it will be much faster. This is useful if you know that the data in question is only ever manipulated from a “process context”, ie no interrupts involved.

The reasons you mustn’t use these versions if you have interrupts that play with the spinlock is that you can get deadlocks:

```
spin_lock(&lock);
...
        <- interrupt comes in:
                spin_lock(&lock);
```

where an interrupt tries to lock an already locked variable. This is ok if the other interrupt happens on another CPU, but it is _not_ ok if the interrupt happens on the same CPU that already holds the lock, because the lock will obviously never be released (because the interrupt is waiting for the lock, and the lock-holder is interrupted by the interrupt and will not continue until the interrupt has been processed).

(This is also the reason why the irq-versions of the spinlocks only need to disable the _local_ interrupts - it’s ok to use spinlocks in interrupts on other CPU’s, because an interrupt on another CPU doesn’t interrupt the CPU that holds the lock, so the lock-holder can continue and eventually releases the lock).

Linus

9.4 Reference information:

For dynamic initialization, use spin_lock_init() or rwlock_init() as appropriate:

```
spinlock_t xxx_lock;
rwlock_t xxx_rw_lock;

static int __init xxx_init(void)
{
     spin_lock_init(&xxx_lock);
     rwlock_init(&xxx_rw_lock);
     ...
}

module_init(xxx_init);
```

For static initialization, use DEFINE_SPINLOCK() / DEFINE_RWLOCK() or __SPIN_LOCK_UNLOCKED() / __RW_LOCK_UNLOCKED() as appropriate.

CHAPTER

TEN

WOUND/WAIT DEADLOCK-PROOF MUTEX DESIGN

Please read *Generic Mutex Subsystem* first, as it applies to wait/wound mutexes too.

10.1 Motivation for WW-Mutexes

GPU's do operations that commonly involve many buffers. Those buffers can be shared across contexts/processes, exist in different memory domains (for example VRAM vs system memory), and so on. And with PRIME / dmabuf, they can even be shared across devices. So there are a handful of situations where the driver needs to wait for buffers to become ready. If you think about this in terms of waiting on a buffer mutex for it to become available, this presents a problem because there is no way to guarantee that buffers appear in a execbuf/batch in the same order in all contexts. That is directly under control of userspace, and a result of the sequence of GL calls that an application makes. Which results in the potential for deadlock. The problem gets more complex when you consider that the kernel may need to migrate the buffer(s) into VRAM before the GPU operates on the buffer(s), which may in turn require evicting some other buffers (and you don't want to evict other buffers which are already queued up to the GPU), but for a simplified understanding of the problem you can ignore this.

The algorithm that the TTM graphics subsystem came up with for dealing with this problem is quite simple. For each group of buffers (execbuf) that need to be locked, the caller would be assigned a unique reservation id/ticket, from a global counter. In case of deadlock while locking all the buffers associated with a execbuf, the one with the lowest reservation ticket (i.e. the oldest task) wins, and the one with the higher reservation id (i.e. the younger task) unlocks all of the buffers that it has already locked, and then tries again.

In the RDBMS literature, a reservation ticket is associated with a transaction. and the deadlock handling approach is called Wait-Die. The name is based on the actions of a locking thread when it encounters an already locked mutex. If the transaction holding the lock is younger, the locking transaction waits. If the transaction holding the lock is older, the locking transaction backs off and dies. Hence Wait-Die. There is also another algorithm called Wound-Wait: If the transaction holding the lock is younger, the locking transaction wounds the transaction holding the lock, requesting it to die. If the transaction holding the lock is older, it waits for the other transaction. Hence Wound-Wait. The two algorithms are both fair in that a transaction will eventually succeed. However, the Wound-Wait algorithm is typically stated to generate fewer backoffs compared to Wait-Die, but is, on the other hand, associated with more work than Wait-Die when recovering from a backoff. Wound-Wait is also a preemptive algorithm in that transactions are wounded by other transactions, and that requires a reliable way to pick up the wounded condition and preempt the running transaction. Note that this is not the same as process preemption. A Wound-Wait transaction is considered preempted when it dies (returning -EDEADLK) following a wound.

10.2 Concepts

Compared to normal mutexes two additional concepts/objects show up in the lock interface for w/w mutexes:

Acquire context: To ensure eventual forward progress it is important that a task trying to acquire locks doesn't grab a new reservation id, but keeps the one it acquired when starting the lock acquisition. This ticket is stored in the acquire context. Furthermore the acquire context keeps track of debugging state to catch w/w mutex interface abuse. An acquire context is representing a transaction.

W/w class: In contrast to normal mutexes the lock class needs to be explicit for w/w mutexes, since it is required to initialize the acquire context. The lock class also specifies what algorithm to use, Wound-Wait or Wait-Die.

Furthermore there are three different class of w/w lock acquire functions:

- Normal lock acquisition with a context, using ww_mutex_lock.
- Slowpath lock acquisition on the contending lock, used by the task that just killed its transaction after having dropped all already acquired locks. These functions have the _slow postfix.

  From a simple semantics point-of-view the _slow functions are not strictly required, since simply calling the normal ww_mutex_lock functions on the contending lock (after having dropped all other already acquired locks) will work correctly. After all if no other ww mutex has been acquired yet there's no deadlock potential and hence the ww_mutex_lock call will block and not prematurely return -EDEADLK. The advantage of the _slow functions is in interface safety:

  - ww_mutex_lock has a __must_check int return type, whereas ww_mutex_lock_slow has a void return type. Note that since ww mutex code needs loops/retries anyway the __must_check doesn't result in spurious warnings, even though the very first lock operation can never fail.
  - When full debugging is enabled ww_mutex_lock_slow checks that all acquired ww mutex have been released (preventing deadlocks) and makes sure that we block on the contending lock (preventing spinning through the -EDEADLK slowpath until the contended lock can be acquired).
- Functions to only acquire a single w/w mutex, which results in the exact same semantics as a normal mutex. This is done by calling ww_mutex_lock with a NULL context.

  Again this is not strictly required. But often you only want to acquire a single lock in which case it's pointless to set up an acquire context (and so better to avoid grabbing a deadlock avoidance ticket).

Of course, all the usual variants for handling wake-ups due to signals are also provided.

10.3 Usage

The algorithm (Wait-Die vs Wound-Wait) is chosen by using either DEFINE_WW_CLASS() (Wound-Wait) or DEFINE_WD_CLASS() (Wait-Die) As a rough rule of thumb, use Wound-Wait iff you expect the number of simultaneous competing transactions to be typically small, and you want to reduce the number of rollbacks.

Three different ways to acquire locks within the same w/w class. Common definitions for methods #1 and #2:

```
static DEFINE_WW_CLASS(ww_class);

struct obj {
      struct ww_mutex lock;
      /* obj data */
};

struct obj_entry {
      struct list_head head;
      struct obj *obj;
};
```

Method 1, using a list in execbuf->buffers that's not allowed to be reordered. This is useful if a list of required objects is already tracked somewhere. Furthermore the lock helper can use propagate the -EALREADY return code back to the caller as a signal that an object is twice on the list. This is useful if the list is constructed from userspace input and the ABI requires userspace to not have duplicate entries (e.g. for a gpu commandbuffer submission ioctl):

```
int lock_objs(struct list_head *list, struct ww_acquire_ctx *ctx)
{
      struct obj *res_obj = NULL;
      struct obj_entry *contended_entry = NULL;
      struct obj_entry *entry;

      ww_acquire_init(ctx, &ww_class);

retry:
      list_for_each_entry (entry, list, head) {
              if (entry->obj == res_obj) {
                      res_obj = NULL;
                      continue;
              }
              ret = ww_mutex_lock(&entry->obj->lock, ctx);
              if (ret < 0) {
                      contended_entry = entry;
                      goto err;
              }
      }

      ww_acquire_done(ctx);
      return 0;
```

```
err:
      list_for_each_entry_continue_reverse (entry, list, head)
              ww_mutex_unlock(&entry->obj->lock);

      if (res_obj)
              ww_mutex_unlock(&res_obj->lock);

      if (ret == -EDEADLK) {
              /* we lost out in a seqno race, lock and retry.. */
              ww_mutex_lock_slow(&contended_entry->obj->lock, ctx);
              res_obj = contended_entry->obj;
              goto retry;
      }
      ww_acquire_fini(ctx);

      return ret;
}
```

Method 2, using a list in execbuf->buffers that can be reordered. Same semantics of duplicate entry detection using -EALREADY as method 1 above. But the list-reordering allows for a bit more idiomatic code:

```
int lock_objs(struct list_head *list, struct ww_acquire_ctx *ctx)
{
      struct obj_entry *entry, *entry2;

      ww_acquire_init(ctx, &ww_class);

      list_for_each_entry (entry, list, head) {
              ret = ww_mutex_lock(&entry->obj->lock, ctx);
              if (ret < 0) {
                      entry2 = entry;

                      list_for_each_entry_continue_reverse (entry2, list, head)
                              ww_mutex_unlock(&entry2->obj->lock);

                      if (ret != -EDEADLK) {
                              ww_acquire_fini(ctx);
                              return ret;
                      }

                      /* we lost out in a seqno race, lock and retry.. */
                      ww_mutex_lock_slow(&entry->obj->lock, ctx);

                      /*
                       * Move buf to head of the list, this will point
                       * buf->next to the first unlocked entry,
                       * restarting the for loop.
                       */
                      list_del(&entry->head);
                      list_add(&entry->head, list);
```

```
                }
        }

        ww_acquire_done(ctx);
        return 0;
}
```

Unlocking works the same way for both methods #1 and #2:

```
void unlock_objs(struct list_head *list, struct ww_acquire_ctx *ctx)
{
        struct obj_entry *entry;

        list_for_each_entry (entry, list, head)
                ww_mutex_unlock(&entry->obj->lock);

        ww_acquire_fini(ctx);
}
```

Method 3 is useful if the list of objects is constructed ad-hoc and not upfront, e.g. when adjusting edges in a graph where each node has its own ww_mutex lock, and edges can only be changed when holding the locks of all involved nodes. w/w mutexes are a natural fit for such a case for two reasons:

- They can handle lock-acquisition in any order which allows us to start walking a graph from a starting point and then iteratively discovering new edges and locking down the nodes those edges connect to.
- Due to the -EALREADY return code signalling that a given objects is already held there's no need for additional book-keeping to break cycles in the graph or keep track off which looks are already held (when using more than one node as a starting point).

Note that this approach differs in two important ways from the above methods:

- Since the list of objects is dynamically constructed (and might very well be different when retrying due to hitting the -EDEADLK die condition) there's no need to keep any object on a persistent list when it's not locked. We can therefore move the list_head into the object itself.
- On the other hand the dynamic object list construction also means that the -EALREADY return code can't be propagated.

Note also that methods #1 and #2 and method #3 can be combined, e.g. to first lock a list of starting nodes (passed in from userspace) using one of the above methods. And then lock any additional objects affected by the operations using method #3 below. The backoff/retry procedure will be a bit more involved, since when the dynamic locking step hits -EDEADLK we also need to unlock all the objects acquired with the fixed list. But the w/w mutex debug checks will catch any interface misuse for these cases.

Also, method 3 can't fail the lock acquisition step since it doesn't return -EALREADY. Of course this would be different when using the _interruptible variants, but that's outside of the scope of these examples here:

```
struct obj {
        struct ww_mutex ww_mutex;
```

```
	struct list_head locked_list;
};

static DEFINE_WW_CLASS(ww_class);

void __unlock_objs(struct list_head *list)
{
	struct obj *entry, *temp;

	list_for_each_entry_safe (entry, temp, list, locked_list) {
		/* need to do that before unlocking, since only the current lock␣
↪holder is
		allowed to use object */
		list_del(&entry->locked_list);
		ww_mutex_unlock(entry->ww_mutex)
	}
}

void lock_objs(struct list_head *list, struct ww_acquire_ctx *ctx)
{
	struct obj *obj;

	ww_acquire_init(ctx, &ww_class);

retry:
	/* re-init loop start state */
	loop {
		/* magic code which walks over a graph and decides which objects
		 * to lock */

		ret = ww_mutex_lock(obj->ww_mutex, ctx);
		if (ret == -EALREADY) {
			/* we have that one already, get to the next object */
			continue;
		}
		if (ret == -EDEADLK) {
			__unlock_objs(list);

			ww_mutex_lock_slow(obj, ctx);
			list_add(&entry->locked_list, list);
			goto retry;
		}

		/* locked a new object, add it to the list */
		list_add_tail(&entry->locked_list, list);
	}

	ww_acquire_done(ctx);
	return 0;
}
```

```
void unlock_objs(struct list_head *list, struct ww_acquire_ctx *ctx)
{
      __unlock_objs(list);
      ww_acquire_fini(ctx);
}
```

Method 4: Only lock one single objects. In that case deadlock detection and prevention is obviously overkill, since with grabbing just one lock you can't produce a deadlock within just one class. To simplify this case the w/w mutex api can be used with a NULL context.

10.4 Implementation Details

10.4.1 Design:

ww_mutex currently encapsulates a struct mutex, this means no extra overhead for normal mutex locks, which are far more common. As such there is only a small increase in code size if wait/wound mutexes are not used.

We maintain the following invariants for the wait list:

(1) Waiters with an acquire context are sorted by stamp order; waiters without an acquire context are interspersed in FIFO order.

(2) For Wait-Die, among waiters with contexts, only the first one can have other locks acquired already (ctx->acquired > 0). Note that this waiter may come after other waiters without contexts in the list.

The Wound-Wait preemption is implemented with a lazy-preemption scheme: The wounded status of the transaction is checked only when there is contention for a new lock and hence a true chance of deadlock. In that situation, if the transaction is wounded, it backs off, clears the wounded status and retries. A great benefit of implementing preemption in this way is that the wounded transaction can identify a contending lock to wait for before restarting the transaction. Just blindly restarting the transaction would likely make the transaction end up in a situation where it would have to back off again.

In general, not much contention is expected. The locks are typically used to serialize access to resources for devices, and optimization focus should therefore be directed towards the uncontended cases.

10.4.2 Lockdep:

Special care has been taken to warn for as many cases of api abuse as possible. Some common api abuses will be caught with CONFIG_DEBUG_MUTEXES, but CONFIG_PROVE_LOCKING is recommended.

Some of the errors which will be warned about:

- Forgetting to call ww_acquire_fini or ww_acquire_init.
- Attempting to lock more mutexes after ww_acquire_done.

- Attempting to lock the wrong mutex after -EDEADLK and unlocking all mutexes.
- Attempting to lock the right mutex after -EDEADLK, before unlocking all mutexes.
- Calling ww_mutex_lock_slow before -EDEADLK was returned.
- Unlocking mutexes with the wrong unlock function.
- Calling one of the ww_acquire_* twice on the same context.
- Using a different ww_class for the mutex than for the ww_acquire_ctx.
- Normal lockdep errors that can result in deadlocks.

Some of the lockdep errors that can result in deadlocks:

- Calling ww_acquire_init to initialize a second ww_acquire_ctx before having called ww_acquire_fini on the first.
- 'normal' deadlocks that can occur.

FIXME:

Update this section once we have the TASK_DEADLOCK task state flag magic implemented.

CHAPTER

ELEVEN

PROPER LOCKING UNDER A PREEMPTIBLE KERNEL: KEEPING KERNEL CODE PREEMPT-SAFE

Author
Robert Love <rml@tech9.net>

11.1 Introduction

A preemptible kernel creates new locking issues. The issues are the same as those under SMP: concurrency and reentrancy. Thankfully, the Linux preemptible kernel model leverages existing SMP locking mechanisms. Thus, the kernel requires explicit additional locking for very few additional situations.

This document is for all kernel hackers. Developing code in the kernel requires protecting these situations.

11.1.1 RULE #1: Per-CPU data structures need explicit protection

Two similar problems arise. An example code snippet:

```
struct this_needs_locking tux[NR_CPUS];
tux[smp_processor_id()] = some_value;
/* task is preempted here... */
something = tux[smp_processor_id()];
```

First, since the data is per-CPU, it may not have explicit SMP locking, but require it otherwise. Second, when a preempted task is finally rescheduled, the previous value of smp_processor_id may not equal the current. You must protect these situations by disabling preemption around them.

You can also use put_cpu() and get_cpu(), which will disable preemption.

11.1.2 RULE #2: CPU state must be protected.

Under preemption, the state of the CPU must be protected. This is arch- dependent, but includes CPU structures and state not preserved over a context switch. For example, on x86, entering and exiting FPU mode is now a critical section that must occur while preemption is disabled. Think what would happen if the kernel is executing a floating-point instruction and is then preempted. Remember, the kernel does not save FPU state except for user tasks. Therefore, upon preemption, the FPU registers will be sold to the lowest bidder. Thus, preemption must be disabled around such regions.

Note, some FPU functions are already explicitly preempt safe. For example, kernel_fpu_begin and kernel_fpu_end will disable and enable preemption.

11.1.3 RULE #3: Lock acquire and release must be performed by same task

A lock acquired in one task must be released by the same task. This means you can't do oddball things like acquire a lock and go off to play while another task releases it. If you want to do something like this, acquire and release the task in the same code path and have the caller wait on an event by the other task.

11.2 Solution

Data protection under preemption is achieved by disabling preemption for the duration of the critical region.

```
preempt_enable()              decrement the preempt counter
preempt_disable()             increment the preempt counter
preempt_enable_no_resched()   decrement, but do not immediately preempt
preempt_check_resched()       if needed, reschedule
preempt_count()               return the preempt counter
```

The functions are nestable. In other words, you can call preempt_disable n-times in a code path, and preemption will not be reenabled until the n-th call to preempt_enable. The preempt statements define to nothing if preemption is not enabled.

Note that you do not need to explicitly prevent preemption if you are holding any locks or interrupts are disabled, since preemption is implicitly disabled in those cases.

But keep in mind that 'irqs disabled' is a fundamentally unsafe way of disabling preemption - any cond_resched() or cond_resched_lock() might trigger a reschedule if the preempt count is 0. A simple printk() might trigger a reschedule. So use this implicit preemption-disabling property only if you know that the affected codepath does not do any of this. Best policy is to use this only for small, atomic code that you wrote and which calls no complex functions.

Example:

```
cpucache_t *cc; /* this is per-CPU */
preempt_disable();
cc = cc_data(searchp);
if (cc && cc->avail) {
        __free_block(searchp, cc_entry(cc), cc->avail);
```

```
        cc->avail = 0;
}
preempt_enable();
return 0;
```

Notice how the preemption statements must encompass every reference of the critical variables. Another example:

```
int buf[NR_CPUS];
set_cpu_val(buf);
if (buf[smp_processor_id()] == -1) printf(KERN_INFO "wee!\n");
spin_lock(&buf_lock);
/* ... */
```

This code is not preempt-safe, but see how easily we can fix it by simply moving the spin_lock up two lines.

11.3 Preventing preemption using interrupt disabling

It is possible to prevent a preemption event using local_irq_disable and local_irq_save. Note, when doing so, you must be very careful to not cause an event that would set need_resched and result in a preemption check. When in doubt, rely on locking or explicit preemption disabling.

Note in 2.5 interrupt disabling is now only per-CPU (e.g. local).

An additional concern is proper usage of local_irq_disable and local_irq_save. These may be used to protect from preemption, however, on exit, if preemption may be enabled, a test to see if preemption is required should be done. If these are called from the spin_lock and read/write lock macros, the right thing is done. They may also be called within a spin-lock protected region, however, if they are ever called outside of this context, a test for preemption should be made. Do note that calls from interrupt context or bottom half/ tasklets are also protected by preemption locks and so may use the versions which do not check preemption.

CHAPTER

TWELVE

LIGHTWEIGHT PI-FUTEXES

We are calling them lightweight for 3 reasons:

- in the user-space fastpath a PI-enabled futex involves no kernel work (or any other PI complexity) at all. No registration, no extra kernel calls - just pure fast atomic ops in userspace.
- even in the slowpath, the system call and scheduling pattern is very similar to normal futexes.
- the in-kernel PI implementation is streamlined around the mutex abstraction, with strict rules that keep the implementation relatively simple: only a single owner may own a lock (i.e. no read-write lock support), only the owner may unlock a lock, no recursive locking, etc.

12.1 Priority Inheritance - why?

The short reply: user-space PI helps achieving/improving determinism for user-space applications. In the best-case, it can help achieve determinism and well-bound latencies. Even in the worst-case, PI will improve the statistical distribution of locking related application delays.

12.2 The longer reply

Firstly, sharing locks between multiple tasks is a common programming technique that often cannot be replaced with lockless algorithms. As we can see it in the kernel [which is a quite complex program in itself], lockless structures are rather the exception than the norm - the current ratio of lockless vs. locky code for shared data structures is somewhere between 1:10 and 1:100. Lockless is hard, and the complexity of lockless algorithms often endangers to ability to do robust reviews of said code. I.e. critical RT apps often choose lock structures to protect critical data structures, instead of lockless algorithms. Furthermore, there are cases (like shared hardware, or other resource limits) where lockless access is mathematically impossible.

Media players (such as Jack) are an example of reasonable application design with multiple tasks (with multiple priority levels) sharing short-held locks: for example, a highprio audio playback thread is combined with medium-prio construct-audio-data threads and low-prio display-colory-stuff threads. Add video and decoding to the mix and we've got even more priority levels.

So once we accept that synchronization objects (locks) are an unavoidable fact of life, and once we accept that multi-task userspace apps have a very fair expectation of being able to use locks,

we've got to think about how to offer the option of a deterministic locking implementation to user-space.

Most of the technical counter-arguments against doing priority inheritance only apply to kernel-space locks. But user-space locks are different, there we cannot disable interrupts or make the task non-preemptible in a critical section, so the 'use spinlocks' argument does not apply (user-space spinlocks have the same priority inversion problems as other user-space locking constructs). Fact is, pretty much the only technique that currently enables good determinism for userspace locks (such as futex-based pthread mutexes) is priority inheritance:

Currently (without PI), if a high-prio and a low-prio task shares a lock [this is a quite common scenario for most non-trivial RT applications], even if all critical sections are coded carefully to be deterministic (i.e. all critical sections are short in duration and only execute a limited number of instructions), the kernel cannot guarantee any deterministic execution of the high-prio task: any medium-priority task could preempt the low-prio task while it holds the shared lock and executes the critical section, and could delay it indefinitely.

12.3 Implementation

As mentioned before, the userspace fastpath of PI-enabled pthread mutexes involves no kernel work at all - they behave quite similarly to normal futex-based locks: a 0 value means unlocked, and a value==TID means locked. (This is the same method as used by list-based robust futexes.) Userspace uses atomic ops to lock/unlock these mutexes without entering the kernel.

To handle the slowpath, we have added two new futex ops:

- FUTEX_LOCK_PI
- FUTEX_UNLOCK_PI

If the lock-acquire fastpath fails, [i.e. an atomic transition from 0 to TID fails], then FUTEX_LOCK_PI is called. The kernel does all the remaining work: if there is no futex-queue attached to the futex address yet then the code looks up the task that owns the futex [it has put its own TID into the futex value], and attaches a 'PI state' structure to the futex-queue. The pi_state includes an rt-mutex, which is a PI-aware, kernel-based synchronization object. The 'other' task is made the owner of the rt-mutex, and the FUTEX_WAITERS bit is atomically set in the futex value. Then this task tries to lock the rt-mutex, on which it blocks. Once it returns, it has the mutex acquired, and it sets the futex value to its own TID and returns. Userspace has no other work to perform - it now owns the lock, and futex value contains FUTEX_WAITERS|TID.

If the unlock side fastpath succeeds, [i.e. userspace manages to do a TID -> 0 atomic transition of the futex value], then no kernel work is triggered.

If the unlock fastpath fails (because the FUTEX_WAITERS bit is set), then FUTEX_UNLOCK_PI is called, and the kernel unlocks the futex on the behalf of userspace - and it also unlocks the attached pi_state->rt_mutex and thus wakes up any potential waiters.

Note that under this approach, contrary to previous PI-futex approaches, there is no prior 'registration' of a PI-futex. [which is not quite possible anyway, due to existing ABI properties of pthread mutexes.]

Also, under this scheme, 'robustness' and 'PI' are two orthogonal properties of futexes, and all four combinations are possible: futex, robust-futex, PI-futex, robust+PI-futex.

More details about priority inheritance can be found in *RT-mutex subsystem with PI support*.

CHAPTER

THIRTEEN

FUTEX REQUEUE PI

Requeueing of tasks from a non-PI futex to a PI futex requires special handling in order to ensure the underlying rt_mutex is never left without an owner if it has waiters; doing so would break the PI boosting logic [see *RT-mutex implementation design*] For the purposes of brevity, this action will be referred to as "requeue_pi" throughout this document. Priority inheritance is abbreviated throughout as "PI".

13.1 Motivation

Without requeue_pi, the glibc implementation of pthread_cond_broadcast() must resort to waking all the tasks waiting on a pthread_condvar and letting them try to sort out which task gets to run first in classic thundering-herd formation. An ideal implementation would wake the highest-priority waiter, and leave the rest to the natural wakeup inherent in unlocking the mutex associated with the condvar.

Consider the simplified glibc calls:

```
/* caller must lock mutex */
pthread_cond_wait(cond, mutex)
{
        lock(cond->__data.__lock);
        unlock(mutex);
        do {
        unlock(cond->__data.__lock);
        futex_wait(cond->__data.__futex);
        lock(cond->__data.__lock);
        } while(...)
        unlock(cond->__data.__lock);
        lock(mutex);
}

pthread_cond_broadcast(cond)
{
        lock(cond->__data.__lock);
        unlock(cond->__data.__lock);
        futex_requeue(cond->data.__futex, cond->mutex);
}
```

Once pthread_cond_broadcast() requeues the tasks, the cond->mutex has waiters. Note that pthread_cond_wait() attempts to lock the mutex only after it has returned to user space. This

will leave the underlying rt_mutex with waiters, and no owner, breaking the previously mentioned PI-boosting algorithms.

In order to support PI-aware pthread_condvar's, the kernel needs to be able to requeue tasks to PI futexes. This support implies that upon a successful futex_wait system call, the caller would return to user space already holding the PI futex. The glibc implementation would be modified as follows:

```
/* caller must lock mutex */
pthread_cond_wait_pi(cond, mutex)
{
        lock(cond->__data.__lock);
        unlock(mutex);
        do {
        unlock(cond->__data.__lock);
        futex_wait_requeue_pi(cond->__data.__futex);
        lock(cond->__data.__lock);
        } while(...)
        unlock(cond->__data.__lock);
        /* the kernel acquired the mutex for us */
}

pthread_cond_broadcast_pi(cond)
{
        lock(cond->__data.__lock);
        unlock(cond->__data.__lock);
        futex_requeue_pi(cond->data.__futex, cond->mutex);
}
```

The actual glibc implementation will likely test for PI and make the necessary changes inside the existing calls rather than creating new calls for the PI cases. Similar changes are needed for pthread_cond_timedwait() and pthread_cond_signal().

13.2 Implementation

In order to ensure the rt_mutex has an owner if it has waiters, it is necessary for both the requeue code, as well as the waiting code, to be able to acquire the rt_mutex before returning to user space. The requeue code cannot simply wake the waiter and leave it to acquire the rt_mutex as it would open a race window between the requeue call returning to user space and the waiter waking and starting to run. This is especially true in the uncontended case.

The solution involves two new rt_mutex helper routines, rt_mutex_start_proxy_lock() and rt_mutex_finish_proxy_lock(), which allow the requeue code to acquire an uncontended rt_mutex on behalf of the waiter and to enqueue the waiter on a contended rt_mutex. Two new system calls provide the kernel<->user interface to requeue_pi: FUTEX_WAIT_REQUEUE_PI and FUTEX_CMP_REQUEUE_PI.

FUTEX_WAIT_REQUEUE_PI is called by the waiter (pthread_cond_wait() and pthread_cond_timedwait()) to block on the initial futex and wait to be requeued to a PI-aware futex. The implementation is the result of a high-speed collision between futex_wait() and futex_lock_pi(), with some extra logic to check for the additional wake-up scenarios.

FUTEX_CMP_REQUEUE_PI is called by the waker (pthread_cond_broadcast() and pthread_cond_signal()) to requeue and possibly wake the waiting tasks. Internally, this system call is still handled by futex_requeue (by passing requeue_pi=1). Before requeueing, futex_requeue() attempts to acquire the requeue target PI futex on behalf of the top waiter. If it can, this waiter is woken. futex_requeue() then proceeds to requeue the remaining nr_wake+nr_requeue tasks to the PI futex, calling rt_mutex_start_proxy_lock() prior to each requeue to prepare the task as a waiter on the underlying rt_mutex. It is possible that the lock can be acquired at this stage as well, if so, the next waiter is woken to finish the acquisition of the lock.

FUTEX_CMP_REQUEUE_PI accepts nr_wake and nr_requeue as arguments, but their sum is all that really matters. futex_requeue() will wake or requeue up to nr_wake + nr_requeue tasks. It will wake only as many tasks as it can acquire the lock for, which in the majority of cases should be 0 as good programming practice dictates that the caller of either pthread_cond_broadcast() or pthread_cond_signal() acquire the mutex prior to making the call. FUTEX_CMP_REQUEUE_PI requires that nr_wake=1. nr_requeue should be INT_MAX for broadcast and 0 for signal.

CHAPTER

FOURTEEN

HARDWARE SPINLOCK FRAMEWORK

14.1 Introduction

Hardware spinlock modules provide hardware assistance for synchronization and mutual exclusion between heterogeneous processors and those not operating under a single, shared operating system.

For example, OMAP4 has dual Cortex-A9, dual Cortex-M3 and a C64x+ DSP, each of which is running a different Operating System (the master, A9, is usually running Linux and the slave processors, the M3 and the DSP, are running some flavor of RTOS).

A generic hwspinlock framework allows platform-independent drivers to use the hwspinlock device in order to access data structures that are shared between remote processors, that otherwise have no alternative mechanism to accomplish synchronization and mutual exclusion operations.

This is necessary, for example, for Inter-processor communications: on OMAP4, cpu-intensive multimedia tasks are offloaded by the host to the remote M3 and/or C64x+ slave processors (by an IPC subsystem called Syslink).

To achieve fast message-based communications, a minimal kernel support is needed to deliver messages arriving from a remote processor to the appropriate user process.

This communication is based on simple data structures that is shared between the remote processors, and access to it is synchronized using the hwspinlock module (remote processor directly places new messages in this shared data structure).

A common hwspinlock interface makes it possible to have generic, platform- independent, drivers.

14.2 User API

```
struct hwspinlock *hwspin_lock_request(void);
```

Dynamically assign an hwspinlock and return its address, or NULL in case an unused hwspinlock isn't available. Users of this API will usually want to communicate the lock's id to the remote core before it can be used to achieve synchronization.

Should be called from a process context (might sleep).

```
struct hwspinlock *hwspin_lock_request_specific(unsigned int id);
```

Assign a specific hwspinlock id and return its address, or NULL if that hwspinlock is already in use. Usually board code will be calling this function in order to reserve specific hwspinlock ids for predefined purposes.

Should be called from a process context (might sleep).

```
int of_hwspin_lock_get_id(struct device_node *np, int index);
```

Retrieve the global lock id for an OF phandle-based specific lock. This function provides a means for DT users of a hwspinlock module to get the global lock id of a specific hwspinlock, so that it can be requested using the normal hwspin_lock_request_specific() API.

The function returns a lock id number on success, -EPROBE_DEFER if the hwspinlock device is not yet registered with the core, or other error values.

Should be called from a process context (might sleep).

```
int hwspin_lock_free(struct hwspinlock *hwlock);
```

Free a previously-assigned hwspinlock; returns 0 on success, or an appropriate error code on failure (e.g. -EINVAL if the hwspinlock is already free).

Should be called from a process context (might sleep).

```
int hwspin_lock_timeout(struct hwspinlock *hwlock, unsigned int timeout);
```

Lock a previously-assigned hwspinlock with a timeout limit (specified in msecs). If the hwspinlock is already taken, the function will busy loop waiting for it to be released, but give up when the timeout elapses. Upon a successful return from this function, preemption is disabled so the caller must not sleep, and is advised to release the hwspinlock as soon as possible, in order to minimize remote cores polling on the hardware interconnect.

Returns 0 when successful and an appropriate error code otherwise (most notably -ETIMEDOUT if the hwspinlock is still busy after timeout msecs). The function will never sleep.

```
int hwspin_lock_timeout_irq(struct hwspinlock *hwlock, unsigned int timeout);
```

Lock a previously-assigned hwspinlock with a timeout limit (specified in msecs). If the hwspinlock is already taken, the function will busy loop waiting for it to be released, but give up when the timeout elapses. Upon a successful return from this function, preemption and the local interrupts are disabled, so the caller must not sleep, and is advised to release the hwspinlock as soon as possible.

Returns 0 when successful and an appropriate error code otherwise (most notably -ETIMEDOUT if the hwspinlock is still busy after timeout msecs). The function will never sleep.

```
int hwspin_lock_timeout_irqsave(struct hwspinlock *hwlock, unsigned int to,
                                unsigned long *flags);
```

Lock a previously-assigned hwspinlock with a timeout limit (specified in msecs). If the hwspinlock is already taken, the function will busy loop waiting for it to be released, but give up when the timeout elapses. Upon a successful return from this function, preemption is disabled, local interrupts are disabled and their previous state is saved at the given flags placeholder. The caller must not sleep, and is advised to release the hwspinlock as soon as possible.

Returns 0 when successful and an appropriate error code otherwise (most notably -ETIMEDOUT if the hwspinlock is still busy after timeout msecs).

The function will never sleep.

```
int hwspin_lock_timeout_raw(struct hwspinlock *hwlock, unsigned int timeout);
```

Lock a previously-assigned hwspinlock with a timeout limit (specified in msecs). If the hwspinlock is already taken, the function will busy loop waiting for it to be released, but give up when the timeout elapses.

Caution: User must protect the routine of getting hardware lock with mutex or spinlock to avoid dead-lock, that will let user can do some time-consuming or sleepable operations under the hardware lock.

Returns 0 when successful and an appropriate error code otherwise (most notably -ETIMEDOUT if the hwspinlock is still busy after timeout msecs).

The function will never sleep.

```
int hwspin_lock_timeout_in_atomic(struct hwspinlock *hwlock, unsigned int to);
```

Lock a previously-assigned hwspinlock with a timeout limit (specified in msecs). If the hwspinlock is already taken, the function will busy loop waiting for it to be released, but give up when the timeout elapses.

This function shall be called only from an atomic context and the timeout value shall not exceed a few msecs.

Returns 0 when successful and an appropriate error code otherwise (most notably -ETIMEDOUT if the hwspinlock is still busy after timeout msecs).

The function will never sleep.

```
int hwspin_trylock(struct hwspinlock *hwlock);
```

Attempt to lock a previously-assigned hwspinlock, but immediately fail if it is already taken.

Upon a successful return from this function, preemption is disabled so caller must not sleep, and is advised to release the hwspinlock as soon as possible, in order to minimize remote cores polling on the hardware interconnect.

Returns 0 on success and an appropriate error code otherwise (most notably -EBUSY if the hwspinlock was already taken). The function will never sleep.

```
int hwspin_trylock_irq(struct hwspinlock *hwlock);
```

Attempt to lock a previously-assigned hwspinlock, but immediately fail if it is already taken.

Upon a successful return from this function, preemption and the local interrupts are disabled so caller must not sleep, and is advised to release the hwspinlock as soon as possible.

Returns 0 on success and an appropriate error code otherwise (most notably -EBUSY if the hwspinlock was already taken).

The function will never sleep.

```
int hwspin_trylock_irqsave(struct hwspinlock *hwlock, unsigned long *flags);
```

Attempt to lock a previously-assigned hwspinlock, but immediately fail if it is already taken.

Upon a successful return from this function, preemption is disabled, the local interrupts are disabled and their previous state is saved at the given flags placeholder. The caller must not sleep, and is advised to release the hwspinlock as soon as possible.

Returns 0 on success and an appropriate error code otherwise (most notably -EBUSY if the hwspinlock was already taken). The function will never sleep.

```
int hwspin_trylock_raw(struct hwspinlock *hwlock);
```

Attempt to lock a previously-assigned hwspinlock, but immediately fail if it is already taken.

Caution: User must protect the routine of getting hardware lock with mutex or spinlock to avoid dead-lock, that will let user can do some time-consuming or sleepable operations under the hardware lock.

Returns 0 on success and an appropriate error code otherwise (most notably -EBUSY if the hwspinlock was already taken). The function will never sleep.

```
int hwspin_trylock_in_atomic(struct hwspinlock *hwlock);
```

Attempt to lock a previously-assigned hwspinlock, but immediately fail if it is already taken.

This function shall be called only from an atomic context.

Returns 0 on success and an appropriate error code otherwise (most notably -EBUSY if the hwspinlock was already taken). The function will never sleep.

```
void hwspin_unlock(struct hwspinlock *hwlock);
```

Unlock a previously-locked hwspinlock. Always succeed, and can be called from any context (the function never sleeps).

Note: code should **never** unlock an hwspinlock which is already unlocked (there is no protection against this).

```
void hwspin_unlock_irq(struct hwspinlock *hwlock);
```

Unlock a previously-locked hwspinlock and enable local interrupts. The caller should **never** unlock an hwspinlock which is already unlocked.

Doing so is considered a bug (there is no protection against this). Upon a successful return from this function, preemption and local interrupts are enabled. This function will never sleep.

```
void
hwspin_unlock_irqrestore(struct hwspinlock *hwlock, unsigned long *flags);
```

Unlock a previously-locked hwspinlock.

The caller should **never** unlock an hwspinlock which is already unlocked. Doing so is considered a bug (there is no protection against this). Upon a successful return from this function,

preemption is reenabled, and the state of the local interrupts is restored to the state saved at the given flags. This function will never sleep.

```
void hwspin_unlock_raw(struct hwspinlock *hwlock);
```

Unlock a previously-locked hwspinlock.

The caller should **never** unlock an hwspinlock which is already unlocked. Doing so is considered a bug (there is no protection against this). This function will never sleep.

```
void hwspin_unlock_in_atomic(struct hwspinlock *hwlock);
```

Unlock a previously-locked hwspinlock.

The caller should **never** unlock an hwspinlock which is already unlocked. Doing so is considered a bug (there is no protection against this). This function will never sleep.

```
int hwspin_lock_get_id(struct hwspinlock *hwlock);
```

Retrieve id number of a given hwspinlock. This is needed when an hwspinlock is dynamically assigned: before it can be used to achieve mutual exclusion with a remote cpu, the id number should be communicated to the remote task with which we want to synchronize.

Returns the hwspinlock id number, or -EINVAL if hwlock is null.

14.3 Typical usage

```
#include <linux/hwspinlock.h>
#include <linux/err.h>

int hwspinlock_example1(void)
{
        struct hwspinlock *hwlock;
        int ret;

        /* dynamically assign a hwspinlock */
        hwlock = hwspin_lock_request();
        if (!hwlock)
                ...

        id = hwspin_lock_get_id(hwlock);
        /* probably need to communicate id to a remote processor now */

        /* take the lock, spin for 1 sec if it's already taken */
        ret = hwspin_lock_timeout(hwlock, 1000);
        if (ret)
                ...

        /*
        * we took the lock, do our thing now, but do NOT sleep
        */
```

```
        /* release the lock */
        hwspin_unlock(hwlock);

        /* free the lock */
        ret = hwspin_lock_free(hwlock);
        if (ret)
                ...

        return ret;
}

int hwspinlock_example2(void)
{
        struct hwspinlock *hwlock;
        int ret;

        /*
        * assign a specific hwspinlock id - this should be called early
        * by board init code.
        */
        hwlock = hwspin_lock_request_specific(PREDEFINED_LOCK_ID);
        if (!hwlock)
                ...

        /* try to take it, but don't spin on it */
        ret = hwspin_trylock(hwlock);
        if (!ret) {
                pr_info("lock is already taken\n");
                return -EBUSY;
        }

        /*
        * we took the lock, do our thing now, but do NOT sleep
        */

        /* release the lock */
        hwspin_unlock(hwlock);

        /* free the lock */
        ret = hwspin_lock_free(hwlock);
        if (ret)
                ...

        return ret;
}
```

14.4 API for implementors

```
int hwspin_lock_register(struct hwspinlock_device *bank, struct device *dev,
             const struct hwspinlock_ops *ops, int base_id, int num_locks);
```

To be called from the underlying platform-specific implementation, in order to register a new hwspinlock device (which is usually a bank of numerous locks). Should be called from a process context (this function might sleep).

Returns 0 on success, or appropriate error code on failure.

```
int hwspin_lock_unregister(struct hwspinlock_device *bank);
```

To be called from the underlying vendor-specific implementation, in order to unregister an hwspinlock device (which is usually a bank of numerous locks).

Should be called from a process context (this function might sleep).

Returns the address of hwspinlock on success, or NULL on error (e.g. if the hwspinlock is still in use).

14.5 Important structs

struct hwspinlock_device is a device which usually contains a bank of hardware locks. It is registered by the underlying hwspinlock implementation using the hwspin_lock_register() API.

```
/**
* struct hwspinlock_device - a device which usually spans numerous hwspinlocks
* @dev: underlying device, will be used to invoke runtime PM api
* @ops: platform-specific hwspinlock handlers
* @base_id: id index of the first lock in this device
* @num_locks: number of locks in this device
* @lock: dynamically allocated array of 'struct hwspinlock'
*/
struct hwspinlock_device {
        struct device *dev;
        const struct hwspinlock_ops *ops;
        int base_id;
        int num_locks;
        struct hwspinlock lock[0];
};
```

struct hwspinlock_device contains an array of hwspinlock structs, each of which represents a single hardware lock:

```
/**
* struct hwspinlock - this struct represents a single hwspinlock instance
* @bank: the hwspinlock_device structure which owns this lock
* @lock: initialized and used by hwspinlock core
* @priv: private data, owned by the underlying platform-specific hwspinlock drv
*/
```

```
struct hwspinlock {
        struct hwspinlock_device *bank;
        spinlock_t lock;
        void *priv;
};
```

When registering a bank of locks, the hwspinlock driver only needs to set the priv members of the locks. The rest of the members are set and initialized by the hwspinlock core itself.

14.6 Implementation callbacks

There are three possible callbacks defined in 'struct hwspinlock_ops':

```
struct hwspinlock_ops {
        int (*trylock)(struct hwspinlock *lock);
        void (*unlock)(struct hwspinlock *lock);
        void (*relax)(struct hwspinlock *lock);
};
```

The first two callbacks are mandatory:

The ->trylock() callback should make a single attempt to take the lock, and return 0 on failure and 1 on success. This callback may **not** sleep.

The ->unlock() callback releases the lock. It always succeed, and it, too, may **not** sleep.

The ->relax() callback is optional. It is called by hwspinlock core while spinning on a lock, and can be used by the underlying implementation to force a delay between two successive invocations of ->trylock(). It may **not** sleep.

CHAPTER

FIFTEEN

PERCPU RW SEMAPHORES

Percpu rw semaphores is a new read-write semaphore design that is optimized for locking for reading.

The problem with traditional read-write semaphores is that when multiple cores take the lock for reading, the cache line containing the semaphore is bouncing between L1 caches of the cores, causing performance degradation.

Locking for reading is very fast, it uses RCU and it avoids any atomic instruction in the lock and unlock path. On the other hand, locking for writing is very expensive, it calls synchronize_rcu() that can take hundreds of milliseconds.

The lock is declared with “struct percpu_rw_semaphore” type. The lock is initialized percpu_init_rwsem, it returns 0 on success and -ENOMEM on allocation failure. The lock must be freed with percpu_free_rwsem to avoid memory leak.

The lock is locked for read with percpu_down_read, percpu_up_read and for write with percpu_down_write, percpu_up_write.

The idea of using RCU for optimized rw-lock was introduced by Eric Dumazet <eric.dumazet@gmail.com>. The code was written by Mikulas Patocka <mpatocka@redhat.com>

CHAPTER

SIXTEEN

A DESCRIPTION OF WHAT ROBUST FUTEXES ARE

Started by
Ingo Molnar <mingo@redhat.com>

16.1 Background

what are robust futexes? To answer that, we first need to understand what futexes are: normal futexes are special types of locks that in the noncontended case can be acquired/released from userspace without having to enter the kernel.

A futex is in essence a user-space address, e.g. a 32-bit lock variable field. If userspace notices contention (the lock is already owned and someone else wants to grab it too) then the lock is marked with a value that says "there's a waiter pending", and the sys_futex(FUTEX_WAIT) syscall is used to wait for the other guy to release it. The kernel creates a 'futex queue' internally, so that it can later on match up the waiter with the waker - without them having to know about each other. When the owner thread releases the futex, it notices (via the variable value) that there were waiter(s) pending, and does the sys_futex(FUTEX_WAKE) syscall to wake them up. Once all waiters have taken and released the lock, the futex is again back to 'uncontended' state, and there's no in-kernel state associated with it. The kernel completely forgets that there ever was a futex at that address. This method makes futexes very lightweight and scalable.

"Robustness" is about dealing with crashes while holding a lock: if a process exits prematurely while holding a pthread_mutex_t lock that is also shared with some other process (e.g. yum segfaults while holding a pthread_mutex_t, or yum is kill -9-ed), then waiters for that lock need to be notified that the last owner of the lock exited in some irregular way.

To solve such types of problems, "robust mutex" userspace APIs were created: pthread_mutex_lock() returns an error value if the owner exits prematurely - and the new owner can decide whether the data protected by the lock can be recovered safely.

There is a big conceptual problem with futex based mutexes though: it is the kernel that destroys the owner task (e.g. due to a SEGFAULT), but the kernel cannot help with the cleanup: if there is no 'futex queue' (and in most cases there is none, futexes being fast lightweight locks) then the kernel has no information to clean up after the held lock! Userspace has no chance to clean up after the lock either - userspace is the one that crashes, so it has no opportunity to clean up. Catch-22.

In practice, when e.g. yum is kill -9-ed (or segfaults), a system reboot is needed to release that futex based lock. This is one of the leading bugreports against yum.

To solve this problem, the traditional approach was to extend the vma (virtual memory area descriptor) concept to have a notion of 'pending robust futexes attached to this

area'. This approach requires 3 new syscall variants to sys_futex(): FUTEX_REGISTER, FUTEX_DEREGISTER and FUTEX_RECOVER. At do_exit() time, all vmas are searched to see whether they have a robust_head set. This approach has two fundamental problems left:

- it has quite complex locking and race scenarios. The vma-based approach had been pending for years, but they are still not completely reliable.
- they have to scan _every_ vma at sys_exit() time, per thread!

The second disadvantage is a real killer: pthread_exit() takes around 1 microsecond on Linux, but with thousands (or tens of thousands) of vmas every pthread_exit() takes a millisecond or more, also totally destroying the CPU's L1 and L2 caches!

This is very much noticeable even for normal process sys_exit_group() calls: the kernel has to do the vma scanning unconditionally! (this is because the kernel has no knowledge about how many robust futexes there are to be cleaned up, because a robust futex might have been registered in another task, and the futex variable might have been simply mmap()-ed into this process's address space).

This huge overhead forced the creation of CONFIG_FUTEX_ROBUST so that normal kernels can turn it off, but worse than that: the overhead makes robust futexes impractical for any type of generic Linux distribution.

So something had to be done.

16.2 New approach to robust futexes

At the heart of this new approach there is a per-thread private list of robust locks that userspace is holding (maintained by glibc) - which userspace list is registered with the kernel via a new syscall [this registration happens at most once per thread lifetime]. At do_exit() time, the kernel checks this user-space list: are there any robust futex locks to be cleaned up?

In the common case, at do_exit() time, there is no list registered, so the cost of robust futexes is just a simple current->robust_list != NULL comparison. If the thread has registered a list, then normally the list is empty. If the thread/process crashed or terminated in some incorrect way then the list might be non-empty: in this case the kernel carefully walks the list [not trusting it], and marks all locks that are owned by this thread with the FUTEX_OWNER_DIED bit, and wakes up one waiter (if any).

The list is guaranteed to be private and per-thread at do_exit() time, so it can be accessed by the kernel in a lockless way.

There is one race possible though: since adding to and removing from the list is done after the futex is acquired by glibc, there is a few instructions window for the thread (or process) to die there, leaving the futex hung. To protect against this possibility, userspace (glibc) also maintains a simple per-thread 'list_op_pending' field, to allow the kernel to clean up if the thread dies after acquiring the lock, but just before it could have added itself to the list. Glibc sets this list_op_pending field before it tries to acquire the futex, and clears it after the list-add (or list-remove) has finished.

That's all that is needed - all the rest of robust-futex cleanup is done in userspace [just like with the previous patches].

Ulrich Drepper has implemented the necessary glibc support for this new mechanism, which fully enables robust mutexes.

Key differences of this userspace-list based approach, compared to the vma based method:

- it's much, much faster: at thread exit time, there's no need to loop over every vma (!), which the VM-based method has to do. Only a very simple 'is the list empty' op is done.
- no VM changes are needed - 'struct address_space' is left alone.
- no registration of individual locks is needed: robust mutexes don't need any extra per-lock syscalls. Robust mutexes thus become a very lightweight primitive - so they don't force the application designer to do a hard choice between performance and robustness - robust mutexes are just as fast.
- no per-lock kernel allocation happens.
- no resource limits are needed.
- no kernel-space recovery call (FUTEX_RECOVER) is needed.
- the implementation and the locking is "obvious", and there are no interactions with the VM.

16.3 Performance

I have benchmarked the time needed for the kernel to process a list of 1 million (!) held locks, using the new method [on a 2GHz CPU]:

- with FUTEX_WAIT set [contended mutex]: 130 msecs
- without FUTEX_WAIT set [uncontended mutex]: 30 msecs

I have also measured an approach where glibc does the lock notification [which it currently does for !pshared robust mutexes], and that took 256 msecs - clearly slower, due to the 1 million FUTEX_WAKE syscalls userspace had to do.

(1 million held locks are unheard of - we expect at most a handful of locks to be held at a time. Nevertheless it's nice to know that this approach scales nicely.)

16.4 Implementation details

The patch adds two new syscalls: one to register the userspace list, and one to query the registered list pointer:

```
asmlinkage long
sys_set_robust_list(struct robust_list_head __user *head,
                    size_t len);

asmlinkage long
sys_get_robust_list(int pid, struct robust_list_head __user **head_ptr,
                    size_t __user *len_ptr);
```

List registration is very fast: the pointer is simply stored in current->robust_list. [Note that in the future, if robust futexes become widespread, we could extend sys_clone() to register a robust-list head for new threads, without the need of another syscall.]

So there is virtually zero overhead for tasks not using robust futexes, and even for robust futex users, there is only one extra syscall per thread lifetime, and the cleanup operation, if it happens, is fast and straightforward. The kernel doesn't have any internal distinction between robust and normal futexes.

If a futex is found to be held at exit time, the kernel sets the following bit of the futex word:

```
#define FUTEX_OWNER_DIED        0x40000000
```

and wakes up the next futex waiter (if any). User-space does the rest of the cleanup.

Otherwise, robust futexes are acquired by glibc by putting the TID into the futex field atomically. Waiters set the FUTEX_WAITERS bit:

```
#define FUTEX_WAITERS           0x80000000
```

and the remaining bits are for the TID.

16.5 Testing, architecture support

I've tested the new syscalls on x86 and x86_64, and have made sure the parsing of the userspace list is robust [;-)] even if the list is deliberately corrupted.

i386 and x86_64 syscalls are wired up at the moment, and Ulrich has tested the new glibc code (on x86_64 and i386), and it works for his robust-mutex testcases.

All other architectures should build just fine too - but they won't have the new syscalls yet.

Architectures need to implement the new futex_atomic_cmpxchg_inatomic() inline function before writing up the syscalls.

CHAPTER

SEVENTEEN

THE ROBUST FUTEX ABI

Author
Started by Paul Jackson <pj@sgi.com>

Robust_futexes provide a mechanism that is used in addition to normal futexes, for kernel assist of cleanup of held locks on task exit.

The interesting data as to what futexes a thread is holding is kept on a linked list in user space, where it can be updated efficiently as locks are taken and dropped, without kernel intervention. The only additional kernel intervention required for robust_futexes above and beyond what is required for futexes is:

1) a one time call, per thread, to tell the kernel where its list of held robust_futexes begins, and
2) internal kernel code at exit, to handle any listed locks held by the exiting thread.

The existing normal futexes already provide a “Fast Userspace Locking” mechanism, which handles uncontested locking without needing a system call, and handles contested locking by maintaining a list of waiting threads in the kernel. Options on the sys_futex(2) system call support waiting on a particular futex, and waking up the next waiter on a particular futex.

For robust_futexes to work, the user code (typically in a library such as glibc linked with the application) has to manage and place the necessary list elements exactly as the kernel expects them. If it fails to do so, then improperly listed locks will not be cleaned up on exit, probably causing deadlock or other such failure of the other threads waiting on the same locks.

A thread that anticipates possibly using robust_futexes should first issue the system call:

```
asmlinkage long
sys_set_robust_list(struct robust_list_head __user *head, size_t len);
```

The pointer ‘head’ points to a structure in the threads address space consisting of three words. Each word is 32 bits on 32 bit arch’s, or 64 bits on 64 bit arch’s, and local byte order. Each thread should have its own thread private ‘head’.

If a thread is running in 32 bit compatibility mode on a 64 native arch kernel, then it can actually have two such structures - one using 32 bit words for 32 bit compatibility mode, and one using 64 bit words for 64 bit native mode. The kernel, if it is a 64 bit kernel supporting 32 bit compatibility mode, will attempt to process both lists on each task exit, if the corresponding sys_set_robust_list() call has been made to setup that list.

> The first word in the memory structure at ‘head’ contains a pointer to a single linked list of ‘lock entries’, one per lock, as described below. If the list is empty, the pointer will point to itself, ‘head’. The last ‘lock entry’ points back to the ‘head’.

The second word, called ‘offset’, specifies the offset from the address of the associated ‘lock entry’, plus or minus, of what will be called the ‘lock word’, from that ‘lock entry’. The ‘lock word’ is always a 32 bit word, unlike the other words above. The ‘lock word’ holds 2 flag bits in the upper 2 bits, and the thread id (TID) of the thread holding the lock in the bottom 30 bits. See further below for a description of the flag bits.

The third word, called ‘list_op_pending’, contains transient copy of the address of the ‘lock entry’, during list insertion and removal, and is needed to correctly resolve races should a thread exit while in the middle of a locking or unlocking operation.

Each ‘lock entry’ on the single linked list starting at ‘head’ consists of just a single word, pointing to the next ‘lock entry’, or back to ‘head’ if there are no more entries. In addition, nearby to each ‘lock entry’, at an offset from the ‘lock entry’ specified by the ‘offset’ word, is one ‘lock word’.

The ‘lock word’ is always 32 bits, and is intended to be the same 32 bit lock variable used by the futex mechanism, in conjunction with robust_futexes. The kernel will only be able to wakeup the next thread waiting for a lock on a threads exit if that next thread used the futex mechanism to register the address of that ‘lock word’ with the kernel.

For each futex lock currently held by a thread, if it wants this robust_futex support for exit cleanup of that lock, it should have one ‘lock entry’ on this list, with its associated ‘lock word’ at the specified ‘offset’. Should a thread die while holding any such locks, the kernel will walk this list, mark any such locks with a bit indicating their holder died, and wakeup the next thread waiting for that lock using the futex mechanism.

When a thread has invoked the above system call to indicate it anticipates using robust_futexes, the kernel stores the passed in ‘head’ pointer for that task. The task may retrieve that value later on by using the system call:

```
asmlinkage long
sys_get_robust_list(int pid, struct robust_list_head __user **head_ptr,
                    size_t __user *len_ptr);
```

It is anticipated that threads will use robust_futexes embedded in larger, user level locking structures, one per lock. The kernel robust_futex mechanism doesn’t care what else is in that structure, so long as the ‘offset’ to the ‘lock word’ is the same for all robust_futexes used by that thread. The thread should link those locks it currently holds using the ‘lock entry’ pointers. It may also have other links between the locks, such as the reverse side of a double linked list, but that doesn’t matter to the kernel.

By keeping its locks linked this way, on a list starting with a ‘head’ pointer known to the kernel, the kernel can provide to a thread the essential service available for robust_futexes, which is to help clean up locks held at the time of (a perhaps unexpectedly) exit.

Actual locking and unlocking, during normal operations, is handled entirely by user level code in the contending threads, and by the existing futex mechanism to wait for, and wakeup, locks. The kernels only essential involvement in robust_futexes is to remember where the list ‘head’ is, and to walk the list on thread exit, handling locks still held by the departing thread, as described below.

There may exist thousands of futex lock structures in a threads shared memory, on various data structures, at a given point in time. Only those lock structures for locks currently held by that thread should be on that thread’s robust_futex linked lock list a given time.

A given futex lock structure in a user shared memory region may be held at different times by

any of the threads with access to that region. The thread currently holding such a lock, if any, is marked with the threads TID in the lower 30 bits of the ‘lock word’.

When adding or removing a lock from its list of held locks, in order for the kernel to correctly handle lock cleanup regardless of when the task exits (perhaps it gets an unexpected signal 9 in the middle of manipulating this list), the user code must observe the following protocol on ‘lock entry’ insertion and removal:

On insertion:

1) set the ‘list_op_pending’ word to the address of the ‘lock entry’ to be inserted,
2) acquire the futex lock,
3) add the lock entry, with its thread id (TID) in the bottom 30 bits of the ‘lock word’, to the linked list starting at ‘head’, and
4) clear the ‘list_op_pending’ word.

On removal:

1) set the ‘list_op_pending’ word to the address of the ‘lock entry’ to be removed,
2) remove the lock entry for this lock from the ‘head’ list,
3) release the futex lock, and
4) clear the ‘lock_op_pending’ word.

On exit, the kernel will consider the address stored in ‘list_op_pending’ and the address of each ‘lock word’ found by walking the list starting at ‘head’. For each such address, if the bottom 30 bits of the ‘lock word’ at offset ‘offset’ from that address equals the exiting threads TID, then the kernel will do two things:

1) if bit 31 (0x80000000) is set in that word, then attempt a futex wakeup on that address, which will waken the next thread that has used to the futex mechanism to wait on that address, and
2) atomically set bit 30 (0x40000000) in the ‘lock word’.

In the above, bit 31 was set by futex waiters on that lock to indicate they were waiting, and bit 30 is set by the kernel to indicate that the lock owner died holding the lock.

The kernel exit code will silently stop scanning the list further if at any point:

1) the ‘head’ pointer or an subsequent linked list pointer is not a valid address of a user space word
2) the calculated location of the ‘lock word’ (address plus ‘offset’) is not the valid address of a 32 bit user space word
3) if the list contains more than 1 million (subject to future kernel configuration changes) elements.

When the kernel sees a list entry whose ‘lock word’ doesn’t have the current threads TID in the lower 30 bits, it does nothing with that entry, and goes on to the next entry.

INDEX

Symbols

D

N

R

S

W