Unreliable Guide To Locking

Paul Rusty Russell

rusty@rustcorp.com.au

Unreliable Guide To Locking

by Paul Rusty Russell

Table of Contents

List of Tables

Chapter 1. Introduction

Welcome, to Rusty's Remarkably Unreliable Guide to Kernel Locking issues. This document describes the locking systems in the Linux Kernel as we approach 2.4.

It looks like *SMP* is here to stay; so everyone hacking on the kernel these days needs to know the fundamentals of concurrency and locking for SMP.

1.1. The Problem With Concurrency

(Skip this if you know what a Race Condition is).

In a normal program, you can increment a counter like so:

```
very_important_count++;
```

This is what they would expect to happen:

Table 1-1. Expected Results

Instance 1	Instance 2
read very_important_count (5)	
add 1 (6)	
write very_important_count (6)	
	read very_important_count (6)
	add 1 (7)
	write very_important_count (7)

This is what might happen:

Table 1-2. Possible Results

Instance 1	**Instance 2**
read very_important_count (5)	
	read very_important_count (5)
add 1 (6)	
	add 1 (6)
write very_important_count (6)	
	write very_important_count (6)

This overlap, where what actually happens depends on the relative timing of multiple tasks, is called a race condition. The piece of code containing the concurrency issue is called a critical region. And especially since Linux starting running on SMP machines, they became one of the major issues in kernel design and implementation.

The solution is to recognize when these simultaneous accesses occur, and use locks to make sure that only one instance can enter the critical region at any time. There are many friendly primitives in the Linux kernel to help you do this. And then there are the unfriendly primitives, but I'll pretend they don't exist.

Chapter 2. Two Main Types of Kernel Locks: Spinlocks and Semaphores

There are two main types of kernel locks. The fundamental type is the spinlock (`include/asm/spinlock.h`), which is a very simple single-holder lock: if you can't get the spinlock, you keep trying (spinning) until you can. Spinlocks are very small and fast, and can be used anywhere.

The second type is a semaphore (`include/asm/semaphore.h`): it can have more than one holder at any time (the number decided at initialization time), although it is most commonly used as a single-holder lock (a mutex). If you can't get a semaphore, your task will put itself on the queue, and be woken up when the semaphore is released. This means the CPU will do something else while you are waiting, but there are many cases when you simply can't sleep (see Section 4.8), and so have to use a spinlock instead.

Neither type of lock is recursive: see Section 4.2.

2.1. Locks and Uniprocessor Kernels

For kernels compiled without CONFIG_SMP, spinlocks do not exist at all. This is an excellent design decision: when no-one else can run at the same time, there is no reason to have a lock at all.

You should always test your locking code with CONFIG_SMP enabled, even if you don't have an SMP test box, because it will still catch some (simple) kinds of deadlock.

Semaphores still exist, because they are required for synchronization between *user contexts*, as we will see below.

2.2. Read/Write Lock Variants

Both spinlocks and semaphores have read/write variants: rwlock_t and struct rw_semaphore. These divide users into two classes: the readers and the writers. If you are only reading the data, you can get a read lock, but to write to the data you need the write lock. Many people can hold a read lock, but a writer must be sole holder.

This means much smoother locking if your code divides up neatly along reader/writer lines. All the discussions below also apply to read/write variants.

2.3. Locking Only In User Context

If you have a data structure which is only ever accessed from user context, then you can use a simple semaphore (`linux/asm/semaphore.h`) to protect it. This is the most trivial case: you initialize the semaphore to the number of resources available (usually 1), and call `down_interruptible()` to grab the semaphore, and `up()` to release it. There is also a `down()`, which should be avoided, because it will not return if a signal is received.

Example: `linux/net/core/netfilter.c` allows registration of new `setsockopt()` and `getsockopt()` calls, with `nf_register_sockopt()`. Registration and de-registration are only done on module load and unload (and boot time, where there is no concurrency), and the list of registrations is only consulted for an unknown `setsockopt()` or `getsockopt()` system call. The `nf_sockopt_mutex` is perfect to protect this, especially since the setsockopt and getsockopt calls may well sleep.

2.4. Locking Between User Context and BHs

If a *bottom half* shares data with user context, you have two problems. Firstly, the current user context can be interrupted by a bottom half, and secondly, the critical region could be entered from another CPU. This is where `spin_lock_bh()` (`include/linux/spinlock.h`) is used. It disables bottom halves on that CPU, then grabs the lock. `spin_unlock_bh()` does the reverse.

This works perfectly for *UP* as well: the spin lock vanishes, and this macro simply becomes `local_bh_disable()` (`include/asm/softirq.h`), which protects you from the bottom half being run.

2.5. Locking Between User Context and Tasklets/Soft IRQs

This is exactly the same as above, because `local_bh_disable()` actually disables all softirqs and *tasklets* on that CPU as well. It should really be called 'local_softirq_disable()', but the name has been preserved for historical reasons. Similarly, `spin_lock_bh()` would now be called spin_lock_softirq() in a perfect world.

2.6. Locking Between Bottom Halves

Sometimes a bottom half might want to share data with another bottom half (especially remember that timers are run off a bottom half).

2.6.1. The Same BH

Since a bottom half is never run on two CPUs at once, you don't need to worry about your bottom half being run twice at once, even on SMP.

2.6.2. Different BHs

Since only one bottom half ever runs at a time once, you don't need to worry about race conditions with other bottom halves. Beware that things might change under you, however, if someone changes your bottom half to a tasklet. If you want to make your code future-proof, pretend you're already running from a tasklet (see below), and doing the extra locking. Of course, if it's five years before that happens, you're gonna look like a damn fool.

2.7. Locking Between Tasklets

Sometimes a tasklet might want to share data with another tasklet, or a bottom half.

2.7.1. The Same Tasklet

Since a tasklet is never run on two CPUs at once, you don't need to worry about your tasklet being reentrant (running twice at once), even on SMP.

2.7.2. Different Tasklets

If another tasklet (or bottom half, such as a timer) wants to share data with your tasklet, you will both need to use `spin_lock()` and `spin_unlock()` calls. `spin_lock_bh()` is unnecessary here, as you are already in a tasklet, and none will be run on the same CPU.

2.8. Locking Between Softirqs

Often a *softirq* might want to share data with itself, a tasklet, or a bottom half.

2.8.1. The Same Softirq

The same softirq can run on the other CPUs: you can use a per-CPU array (see Section 4.3) for better performance. If you're going so far as to use a softirq, you probably care about scalable performance enough to justify the extra complexity.

You'll need to use `spin_lock()` and `spin_unlock()` for shared data.

2.8.2. Different Softirqs

You'll need to use `spin_lock()` and `spin_unlock()` for shared data, whether it be a timer (which can be running on a different CPU), bottom half, tasklet or the same or another softirq.

Chapter 3. Hard IRQ Context

Hardware interrupts usually communicate with a bottom half, tasklet or softirq. Frequently this involves putting work in a queue, which the BH/softirq will take out.

3.1. Locking Between Hard IRQ and Softirqs/Tasklets/BHs

If a hardware irq handler shares data with a softirq, you have two concerns. Firstly, the softirq processing can be interrupted by a hardware interrupt, and secondly, the critical region could be entered by a hardware interrupt on another CPU. This is where `spin_lock_irq()` is used. It is defined to disable interrupts on that cpu, then grab the lock. `spin_unlock_irq()` does the reverse.

This works perfectly for UP as well: the spin lock vanishes, and this macro simply becomes `local_irq_disable()` (`include/asm/smp.h`), which protects you from the softirq/tasklet/BH being run.

`spin_lock_irqsave()` (`include/linux/spinlock.h`) is a variant which saves whether interrupts were on or off in a flags word, which is passed to `spin_lock_irqrestore()`. This means that the same code can be used inside an hard irq handler (where interrupts are already off) and in softirqs (where the irq disabling is required).

Chapter 4. Common Techniques

This section lists some common dilemmas and the standard solutions used in the Linux kernel code. If you use these, people will find your code simpler to understand.

If I could give you one piece of advice: never sleep with anyone crazier than yourself. But if I had to give you advice on locking: *keep it simple*.

Lock data, not code.

Be reluctant to introduce new locks.

Strangely enough, this is the exact reverse of my advice when you *have* slept with someone crazier than yourself.

4.1. No Writers in Interrupt Context

There is a fairly common case where an interrupt handler needs access to a critical region, but does not need write access. In this case, you do not need to use `read_lock_irq()`, but only `read_lock()` everywhere (since if an interrupt occurs, the irq handler will only try to grab a read lock, which won't deadlock). You will still need to use `write_lock_irq()`.

Similar logic applies to locking between softirqs/tasklets/BHs which never need a write lock, and user context: `read_lock()` and `write_lock_bh()`.

4.2. Deadlock: Simple and Advanced

There is a coding bug where a piece of code tries to grab a spinlock twice: it will spin forever, waiting for the lock to be released (spinlocks, rwlocks and semaphores are not recursive in Linux). This is trivial to diagnose: not a stay-up-five-nights-talk-to-fluffy-code-bunnies kind of problem.

For a slightly more complex case, imagine you have a region shared by a bottom half and user context. If you use a `spin_lock()` call to protect it, it is possible that the user context will be interrupted by the bottom half while it holds the lock, and the bottom half will then spin forever trying to get the same lock.

Both of these are called deadlock, and as shown above, it can occur even with a single CPU (although not on UP compiles, since spinlocks vanish on kernel compiles with CONFIG_SMP=n. You'll still get data corruption in the second example).

This complete lockup is easy to diagnose: on SMP boxes the watchdog timer or compiling with DEBUG_SPINLOCKS set (`include/linux/spinlock.h`) will show this up immediately when it happens.

A more complex problem is the so-called 'deadly embrace', involving two or more locks. Say you have a hash table: each entry in the table is a spinlock, and a chain of hashed objects. Inside a softirq handler, you sometimes want to alter an object from one place in the hash to another: you grab the spinlock of the old hash chain and the spinlock of the new hash chain, and delete the object from the old one, and insert it in the new one.

There are two problems here. First, if your code ever tries to move the object to the same chain, it will deadlock with itself as it tries to lock it twice. Secondly, if the same softirq on another CPU is trying to move another object in the reverse direction, the following could happen:

Table 4-1. Consequences

CPU 1	CPU 2
Grab lock A -> OK	Grab lock B -> OK
Grab lock B -> spin	Grab lock A -> spin

The two CPUs will spin forever, waiting for the other to give up their lock. It will look, smell, and feel like a crash.

4.2.1. Preventing Deadlock

Textbooks will tell you that if you always lock in the same order, you will never get this kind of deadlock. Practice will tell you that this approach doesn't scale: when I create a new lock, I don't understand enough of the kernel to figure out where in the 5000 lock hierarchy it will fit.

The best locks are encapsulated: they never get exposed in headers, and are never held around calls to non-trivial functions outside the same file. You can read through this code and see that it will never deadlock, because it never tries to grab another lock while it has that one. People using your code don't even need to know you are using a lock.

A classic problem here is when you provide callbacks or hooks: if you call these with the lock held, you risk simple deadlock, or a deadly embrace (who knows what the callback will do?). Remember, the other programmers are out to get you, so don't do this.

4.2.2. Overzealous Prevention Of Deadlocks

Deadlocks are problematic, but not as bad as data corruption. Code which grabs a read lock, searches a list, fails to find what it wants, drops the read lock, grabs a write lock and inserts the object has a race condition.

If you don't see why, please stay the fuck away from my code.

4.3. Per-CPU Data

A great technique for avoiding locking which is used fairly widely is to duplicate information for each CPU. For example, if you wanted to keep a count of a common condition, you could use a spin lock and a single counter. Nice and simple.

If that was too slow [it's probably not], you could instead use a counter for each CPU [don't], then none of them need an exclusive lock [you're wasting your time here]. To make sure the CPUs don't have to synchronize caches all the time, align the counters to cache boundaries by appending '__cacheline_aligned' to the declaration (`include/linux/cache.h`). [Can't you think of anything better to do?]

They will need a read lock to access their own counters, however. That way you can use a write lock to grant exclusive access to all of them at once, to tally them up.

4.4. Big Reader Locks

A classic example of per-CPU information is Ingo's 'big reader' locks (`linux/include/brlock.h`). These use the Per-CPU Data techniques described above to create a lock which is very fast to get a read lock, but agonizingly slow for a write lock.

Fortunately, there are a limited number of these locks available: you have to go through a strict interview process to get one.

4.5. Avoiding Locks: Read And Write Ordering

Sometimes it is possible to avoid locking. Consider the following case from the 2.2 firewall code, which inserted an element into a single linked list in user context:

```
new->next = i->next;
i->next = new;
```

Here the author (Alan Cox, who knows what he's doing) assumes that the pointer assignments are atomic. This is important, because networking packets would traverse this list on bottom halves without a lock. Depending on their exact timing, they would either see the new element in the list with a valid *`next`* pointer, or it would not be in the list yet. A lock is still required against other CPUs inserting or deleting from the list, of course.

Of course, the writes *must* be in this order, otherwise the new element appears in the list with an invalid *`next`* pointer, and any other CPU iterating at the wrong time will jump through it into garbage. Because modern CPUs reorder, Alan's code actually read as follows:

```
new->next = i->next;
wmb();
i->next = new;
```

The `wmb()` is a write memory barrier (`include/asm/system.h`): neither the compiler nor the CPU will allow any writes to memory after the `wmb()` to be visible to other hardware before any of the writes before the `wmb()`.

As i386 does not do write reordering, this bug would never show up on that platform. On other SMP platforms, however, it will.

There is also `rmb()` for read ordering: to ensure any previous variable reads occur before following reads. The simple `mb()` macro combines both `rmb()` and `wmb()`.

Some atomic operations are defined to act as a memory barrier (ie. as per the `mb()` macro, but if in doubt, be explicit. Also, spinlock operations act as partial barriers: operations after gaining a spinlock will never be moved to precede the `spin_lock()` call, and operations before releasing a spinlock will never be moved after the `spin_unlock()` call.

4.6. Avoiding Locks: Atomic Operations

There are a number of atomic operations defined in `include/asm/atomic.h`: these are guaranteed to be seen atomically from all CPUs in the system, thus avoiding races. If your shared data consists of a single counter, say, these operations might be simpler than using spinlocks (although for anything non-trivial using spinlocks is clearer).

Note that the atomic operations are defined to act as both read and write barriers on all platforms.

4.7. Protecting A Collection of Objects: Reference Counts

Locking a collection of objects is fairly easy: you get a single spinlock, and you make sure you grab it before searching, adding or deleting an object.

The purpose of this lock is not to protect the individual objects: you might have a separate lock inside each one for that. It is to protect the *data structure containing the objects* from race conditions. Often the same lock is used to protect the contents of all the objects as well, for simplicity, but they are inherently orthogonal (and many other big words designed to confuse).

Changing this to a read-write lock will often help markedly if reads are far more common that writes. If not, there is another approach you can use to reduce the time the lock is held: reference counts.

In this approach, an object has an owner, who sets the reference count to one. Whenever you get a pointer to the object, you increment the reference count (a 'get' operation). Whenever you relinquish a pointer, you decrement the reference count (a 'put' operation). When the owner wants to destroy it, they mark it dead, and do a put.

Whoever drops the reference count to zero (usually implemented with `atomic_dec_and_test()`) actually cleans up and frees the object.

This means that you are guaranteed that the object won't vanish underneath you, even though you no longer have a lock for the collection.

Here's some skeleton code:

```
void create_foo(struct foo *x)
{
        atomic_set(&x->use, 1);
        spin_lock_bh(&list_lock);
        ... insert in list ...
        spin_unlock_bh(&list_lock);
}

struct foo *get_foo(int desc)
{
        struct foo *ret;

        spin_lock_bh(&list_lock);
        ... find in list ...
```

```
        if (ret) atomic_inc(&ret->use);
        spin_unlock_bh(&list_lock);

        return ret;
}

void put_foo(struct foo *x)
{
        if (atomic_dec_and_test(&x->use))
                kfree(foo);
}

void destroy_foo(struct foo *x)
{
        spin_lock_bh(&list_lock);
        ... remove from list ...
        spin_unlock_bh(&list_lock);

        put_foo(x);
}
```

4.7.1. Macros To Help You

There are a set of debugging macros tucked inside `include/linux/netfilter_ipv4/lockhelp.h` and `listhelp.h`: these are very useful for ensuring that locks are held in the right places to protect infrastructure.

4.8. Things Which Sleep

You can never call the following routines while holding a spinlock, as they may sleep. This also means you need to be in user context.

- Accesses to *userspace*:

  - `copy_from_user()`
  - `copy_to_user()`
  - `get_user()`
  - `put_user()`

- `kmalloc(GFP_KERNEL)`
- `down_interruptible()` and `down()`

  There is a `down_trylock()` which can be used inside interrupt context, as it will not sleep. `up()` will also never sleep.

`printk()` can be called in *any* context, interestingly enough.

4.9. The Fucked Up Sparc

Alan Cox says "the irq disable/enable is in the register window on a sparc". Andi Kleen says "when you do restore_flags in a different function you mess up all the register windows".

So never pass the flags word set by `spin_lock_irqsave()` and brethren to another function (unless it's declared inline. Usually no-one does this, but now you've been warned. Dave Miller can never do anything in a straightforward manner (I can say that, because I have pictures of him and a certain PowerPC maintainer in a compromising position).

4.10. Racing Timers: A Kernel Pastime

Timers can produce their own special problems with races. Consider a collection of objects (list, hash, etc) where each object has a timer which is due to destroy it.

If you want to destroy the entire collection (say on module removal), you might do the following:

```
/* THIS CODE BAD BAD BAD BAD: IF IT WAS ANY WORSE IT WOULD USE
   HUNGARIAN NOTATION */
spin_lock_bh(&list_lock);

while (list) {
        struct foo *next = list->next;
        del_timer(&list->timer);
        kfree(list);
        list = next;
}

spin_unlock_bh(&list_lock);
```

Sooner or later, this will crash on SMP, because a timer can have just gone off before the `spin_lock_bh()`, and it will only get the lock after we `spin_unlock_bh()`, and then try to free the element (which has already been freed!).

This can be avoided by checking the result of `del_timer()`: if it returns 1, the timer has been deleted. If 0, it means (in this case) that it is currently running, so we can do:

```
retry:
        spin_lock_bh(&list_lock);

        while (list) {
                struct foo *next = list->next;
                if (!del_timer(&list->timer)) {
                        /* Give timer a chance to delete this */
                        spin_unlock_bh(&list_lock);
                        goto retry;
                }
                kfree(list);
                list = next;
        }

        spin_unlock_bh(&list_lock);
```

Another common problem is deleting timers which restart themselves (by calling add_timer() at the end of their timer function). Because this is a fairly common case which is prone to races, you can put a call to timer_exit() at the very end of your timer function, and user del_timer_sync() (include/linux/timer.h) to handle this case. It returns the number of times the timer had to be deleted before we finally stopped it from adding itself back in.

Chapter 5. Further reading

- `Documentation/spinlocks.txt`: Linus Torvalds' spinlocking tutorial in the kernel sources.
- Unix Systems for Modern Architectures: Symmetric Multiprocessing and Caching for Kernel Programmers:

  Curt Schimmel's very good introduction to kernel level locking (not written for Linux, but nearly everything applies). The book is expensive, but really worth every penny to understand SMP locking. [ISBN: 0201633388]

Chapter 6. Thanks

Thanks to Telsa Gwynne for DocBooking, neatening and adding style.

Thanks to Martin Pool, Philipp Rumpf, Stephen Rothwell, Paul Mackerras, Ruedi Aschwanden, Alan Cox, Manfred Spraul and Tim Waugh for proofreading, correcting, flaming, commenting.

Thanks to the cabal for having no influence on this document.

Glossary

bh

Bottom Half: for historical reasons, functions with '_bh' in them often now refer to any software interrupt, e.g. `spin_lock_bh()` blocks any software interrupt on the current CPU. Bottom halves are deprecated, and will eventually be replaced by tasklets. Only one bottom half will be running at any time.

Hardware Interrupt / Hardware IRQ

Hardware interrupt request. `in_irq()` returns true in a hardware interrupt handler (it also returns true when interrupts are blocked).

Interrupt Context

Not user context: processing a hardware irq or software irq. Indicated by the `in_interrupt()` macro returning true (although it also returns true when interrupts or BHs are blocked).

SMP

Symmetric Multi-Processor: kernels compiled for multiple-CPU machines. (CONFIG_SMP=y).

softirq

Strictly speaking, one of up to 32 enumerated software interrupts which can run on multiple CPUs at once. Sometimes used to refer to tasklets and bottom halves as well (ie. all software interrupts).

Software Interrupt / Software IRQ

Software interrupt handler. `in_irq()` returns false; `in_softirq()` returns true.

Tasklets, softirqs and bottom halves all fall into the category of 'software interrupts'.

tasklet

A dynamically-registrable software interrupt, which is guaranteed to only run on one CPU at a time.

UP

Uni-Processor: Non-SMP. (CONFIG_SMP=n).

User Context

The kernel executing on behalf of a particular process or kernel thread (given by the `current()` macro.) Not to be confused with userspace. Can be interrupted by software or hardware interrupts.

Userspace

A process executing its own code outside the kernel.