Unreliable Guide To Locking

Paul Rusty Russell

rusty@rustcorp.com.au

Unreliable Guide To Locking

by Paul Rusty Russell

Copyright ' 2000 by Paul Russell

This documentation is free software; you can redistribute it and/or modify it under the terms of the GNU General Public License as published by the Free Software Foundation; either version 2 of the License, or (at your option) any later version.

This program is distributed in the hope that it will be useful, but WITHOUT ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU General Public License for more details.

You should have received a copy of the GNU General Public License along with this program; if not, write to the Free Software Foundation, Inc., 59 Temple Place, Suite 330, Boston, MA 02111-1307 USA

For more details see the le COPYING in the source distribution of Linux.

Table of Contents

1. Introduction 1
1.1. The Problem With Concurrency 1
2. Two Main Types of Kernel Locks: Spinlocks and Semaphores 3
2.1. Locks and Uniprocessor Kernels 3
2.2. Read/Write Lock Variants 3
2.3. Locking Only In User Context 3
2.4. Locking Between User Context and BHs 4
2.5. Locking Between User Context and Tasklets/Soft IRQs 4
2.6. Locking Between Bottom Halves 4
2.6.1. The Same BH 5
2.6.2. Different BHs 5
2.7. Locking Between Tasklets 5
2.7.1. The Same Tasklet 5
2.7.2. Different Tasklets 5
2.8. Locking Between Softirqs 6
2.8.1. The Same Softirq 6
2.8.2. Different Softirqs 6
3. Hard IRQ Context 7
3.1. Locking Between Hard IRQ and Softirqs/Tasklets/BHs 7
4. Common Techniques 8
4.1. No Writers in Interrupt Context 8
4.2. Deadlock: Simple and Advanced 8
4.2.1. Preventing Deadlock 9
4.2.2. Overzealous Prevention Of Deadlocks 10
4.3. Per-CPU Data 10
4.4. Big Reader Locks 10
4.5. Avoiding Locks: Read And Write Ordering 10
4.6. Avoiding Locks: Atomic Operations 11
4.7. Protecting A Collection of Objects: Reference Counts 12
4.7.1. Macros To Help You 13
4.8. Things Which Sleep 13
4.9. The Fucked Up Sparc 14
4.10. Racing Timers: A Kernel Pastime 14
5. Further reading 16
6. Thanks 17
Glossary 18

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 gure out where in the 5000 lock hierarchy it will t.

The best locks are encapsulated: they never get exposed in headers, and are never held around calls to non-trivial functions outside the same le. 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 nd 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 rewall 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 de ned 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 de ned 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 de ned 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.

```
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 nally 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, aming, commenting.

Thanks to the cabal for having no in uence 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.