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 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

This is what might happen:

Table 1-2. Possible Results

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), and so have to use a spinlock instead.

Neither type of lock is recursive: see .

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 , 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 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 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 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 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) 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