I'll Do It Later: Softirqs, Tasklets, Bottom Halves, Task Queues, Work Queues and Timers

Matthew Wilcox
Hewlett-Packard Company
matthew.wilcox@hp.com

Abstract

An interrupt is a signal to a device driver that there is work to be done. However, if the driver does too much work in the interrupt handler, system responsiveness will be degraded. The standard way to avoid this problem (until Linux 2.3.42) was to use a bottom half or a task queue to schedule some work to do later. These handlers are run with interrupts enabled and lengthy processing has less impact on system response.

The work done for softnet introduced two new facilities for deferring work until later: softirqs and tasklets. They were introduced in order to achieve better SMP scalabiity. The existing bottom halves were reimplemented as a special form of tasklet which preserved their semantics. In Linux 2.5.40, these bottom halves were removed; and in 2.5.41, task queues were replaced with a new abstraction: work queues.

This paper discusses the differences and relationships between softirqs, tasklets, work queues and timers. The rules for using them are laid out, along with some guidelines for choosing when to use which feature.

Converting a driver from using the older mechanisms to the new ones requires SMP auditing and understanding the interactions between the various driver entry points. Accordingly, there is a brief review of the basic locking primitives, followed by a more detailed examination of the additional locking primitives which were introduced with the softirqs and tasklets.

1 Introduction

Low interrupt latency is extremely important to any operating system. It is a factor in desktop responsiveness and it is even more important in network loads. Since Linux disables interrupts while an interrupt handler is running, it is important to not do too much work in the interrupt handler. This is a common issue in Unix-like operating systems, and the standard approach is to split interrupt routines into a 'top half' which receives the hardware interrupt and a 'bottom half' which does the lengthy processing.

In Linux 2.2, a lot of different parts of the kernel were run as a bottom half. Networking, keyboard, console, SCSI and serial all used bottom halves directly, and most of the rest of the kernel did it indirectly. Timers were run as a bottom half, as were the immediate and periodic task queues.

It is amusing to note that when Ted Ts'o first implemented bottom halves for Linux, he called them Softirqs. Linus said he'd never accept softirqs so Ted changed the name to Bottom Halves and Linus accepted it.

2 The softirq framework

2.1 Motivation

In April 1999, Mindcraft published a benchmark [Mindcraft] which pointed out some weaknesses in Linux's networking performance on a 4-way SMP machine. As a result, Alexey Kuznetsov and Dave Miller multithreaded the network stack. They soon realised that this was not enough. The problem was that although each CPU could be handling an interrupt at the same time, the bottom half layer was singly-threaded, so all the time-consuming work was still not distributed across all the CPUs.

The remaining step was to multithread the bottom halves. They chose to do this by introducing a couple of new layers below the bottom half – softirqs and tasklets. Bottom halves were reimplemented as a set of tasklets which executed with a special spinlock held. This preserved the single-threaded nature of the bottom half for those drivers which assumed it while letting the network stack run simultaneously on all CPUs.

2.2 Implementing softirqs

On return from handling a hard interrupt, Linux would run the bottom halves. This was changed to run the softirqs instead. There are a fixed number of softirqs and they are run in priority order. It is possible to add new softirqs, but it's necessary to have them approved and added to the list.

Softirqs have strong CPU affinity. When `raise_softirq()` is called, that softirq will run on this processor. Of course, it's possible that another processor will also have this softirq raised and will arrive first, but all current softirqs have per-cpu data so they don't interfere with each other at all.

Linux 2.5.48 defines 6 softirqs. The highest priority softirq runs the high priority tasklets. After this, we run the timers, then trigger the network transmit and receive softirqs, then service the SCSI softirqs. Finally, low-priority tasklets are run.

2.3 Tasklets

Tasklets are dynamically allocated and have a weaker CPU affinity. If the tasklet has already been scheduled on a different CPU, it will not be moved to another CPU if it's still pending. Tasklets are not synchronised with respect to each other, however they will not be reentered.

Device drivers should normally use a tasklet to defer work until later by using the `tasklet_schedule()` interface. If the tasklet should be run more urgently than networking, scsi, timers or anything else, they should use the `tasklet_hi_schedule()` interface. This is intended for low-latency tasks which are critical for interactive feel – for example, the keyboard driver.

Tasklets may also be enabled and disabled. This is useful when the driver is in an exceptional situation (eg network card with an unplugged cable) which needs to be sure the tasklet is not executing, but taking a lock inside the tasklet would be too expensive.

2.4 ksoftirqd

When the machine is under heavy interrupt load, it is possible for the cpu to spend all its time servicing interrupts and softirqs without making forward progress. To prevent this from saturating the machine, and so the sysadmin can detect that this situation is occurring, if too much work is happening in softirq context further softirq processing is handled by a per-cpu kernel thread called ksoftirqd. The current definition of "too much work" is a softirq being reactivated during a run at processing softirqs. Some argue this is too eager and ksoftirqd activation should be reserved for higher overload situations.

Although the work is now being done in process context rather than interrupt context, ksoftirqd still executes this code with local interrupts enabled and bottom halves disabled locally. Code which runs as a bottom half does not need to change in order to be run by ksoftirqd.

2.5 Problems

There are some subtle problems with using softirqs and tasklets. Some are obvious – driver writers muct be more careful with locking. Others are less obvious. For example, it's a great thing to be able to take interrupts on all CPUs simultaneously, but there's no point in taking an interrupt if you can't perform the work necessary to process it before receiving the next one.

Networking is particularly prone to this. Assuming the interrupt controller distributes interrupts among CPUs in a round-robin fashion (this is the default for Intel IO-APICs), we can produce worst-case behaviour by simply ping-flooding an SMP machine.

Interrupts will hit each CPU in turn, each will trigger the network receive softirq on the local CPU, then they will each try to deliver their packets into the networking stack. Even if the CPUs don't spend all their time spinning on locks waiting for each other to exit critical regions, they steal cachelines from each other and waste time that way.

The solution to this is to make use of the tasklets. Allocate one tasklet per network interface, then queue work for the tasklet to do from the softirq. This binds all the work from a network card to the first processor to get an interrupt during an interrupt storm. The other processors will service the interrupt and softirq and then go back to doing other work. This technique was designed as part of the NAPI framework by Jamal Hadi Salim and Alexey Kuznetsov.

3 Timers

Originally there was an array of 32 timers, Similarly to softirqs today, you needed special permission to get one. They were used for everything from SCSI, Net and Floppy to the 387 coprocessor, the QIC-02 tape driver and assorted old CD-ROM devices.

Even by 2.0, we had outgrown this and there was a "new and improved" dynamic timer interface. Nevertheless, the old timers persisted into 2.2 and were finally removed from 2.4 by Andrew Morton.

Timers were run from their own bottom half originally. The softnet work did not change this, so timers still ran serialised with respect to each other, other bottom halves and as a special case they were serialised with respect to network protocols which had not yet been converted to the softnet framework.

This changed in 2.5.40 when bottom halves were removed. The exclusion with other bottom halves and old network protocols was removed, and timers could now be run on multiple CPUs simultaneously. This was initially done with a per-cpu tasklet for timers, but 2.5.48 simplified this to directly use softirqs.

Any code which uses timers needs to be audited to make sure that it does not race with other timers accessing the same data, or with other asynchronous events such as softirqs or tasklets.

The dynamic timers have always been controlled by the interfaces `add_timer()`, `del_timer()` and `mod_timer()`. 2.4 extended the interface with `del_timer_sync()` which guarantees that the timer is no longer running by the time it returns. 2.5.45 augments the interface with `add_timer_on()` which allows a timer to be added on a different CPU.

Drivers have traditionally had trouble using timers in a safe and race-free way. Partly, this is because timers are permitted so much latitude in what they can do. They can (`kfree()` the `timer_list` (or the struct embedding the `timer_list`). They can add, modify or delete the timer. This can race with some other part of the kernel calling `del_timer()`, and then assuming it can free the timer, exit the module or shut down a device safely.

In fact, the timer can potentially still be running, and may even continue to run indefinitely. If the timer handler re-adds the timer after the deletion occurs, it will run again after `del_timer()` has returned. `del_timer_sync()` waits until the timer is no longer running on any CPU before it returns. Unfortunately, it can deadlock if the code which calls `del_timer_sync()` is holding a lock which the timer handler routine needs to exit.

Many users of timers are still unsafe in the 2.5 kernel, and a comprehensive audit is required. Fortunately, most of these places are in module exit paths, so are only hit rarely. But the consequences when they are can be catastrophic – the kernel will probably panic.

4 Task and Work queues

Task queues were originally designed as a replacement for the old bottom halves. When they were integrated into the kernel, they did not replace bottom halves but were used as an adjunct to them. Like tasklets and the new-style timers, they were dynamically allocated.

One bottom half was dedicated to running the `tq_timer` task queue, and another was dedicated to running the `tq_immediate` task queue. The interface to using `tq_timer` was fine, but using `tq_immediate` was awful.

To defer execution from interrupt context, the driver had to call `queue_task(tq_immediate, &my_tqueue)` and then `mark_bh(BH_IMMEDIATE)`. This exposed internal implementation details to drivers and it was not unheard of for a driver to miss the call to `mark_bh`, causing their processing to be delayed until some other process marked the bottom-half ready to run.

Another well-known task queue was `tq_disk`. It was run whenever some random part of the kernel felt like calling `run_task_queue(tq_disk)`. Instructions for using various interfaces said to call it, either before or after. This interface was removed in 2.5, replaced with `blk_run_queues()`. Unfortunately, it's still called in all the same places, and both failing to call it and calling it too frequently has bad effects on system performance.

A feature introduced to the 2.4 kernel was a task queue that would be run by keventd at process context. The interface was much more sensible, involving a single call to `schedule_task()`. Various parts of the kernel had their own custom task queue which would be run as appropriate. For example, reiserfs used a task queue for its journal commit thread.

2.5.41 replaced the task queue with the work queue. Drivers which used to use `tq_immediate` are encouraged to switch to tasklets. Users of `tq_timer` should use timers directly. If these interfaces are inappropriate, the `schedule_work` and `schedule_delayed_work` interfaces may be used. These interfaces queue the work on keventd and it is then run in process context. Interrupts and bottom halves are both enabled while the work is being done. The work being done may include blocking operations, but this is discouraged as it prevents other users of these interfaces from running on this CPU.

For parts of the kernel which used their own task queue, work queues may be created and destroyed dynamically, normally on module init and exit. A new workqueue creates a kernel thread per CPU, and work may then be scheduled to it via the `queue_work` and `queue_delayed_work` interfaces.

Note that the routines for scheduling work on keventd are available to everyone, but the routines for using custom task queues are only available to modules which are licensed under a GPL-compatible license.

5 Contexts and Locking

There are now three contexts available for code in the Linux kernel. Process context is kernel code executing directly on behalf of a user process. All syscalls are in process context, for example. Interrupt handlers run in interrupt context. Softirqs, tasklets and timers all run in bottom-half context.

Since the point of the softirq framework was to improve SMP scalability, code from these entry points may run simultaneously on all CPUs. To protect access to data structures and hardware, you should use a spinlock. If you are in process context or bottom half context, you should use `spin_lock_irq()` to disable interrupts before you acquire the spinlock. This prevents receiving an interrupt on the same CPU which is holding the spinlock and deadlocking by trying to acquire the same spinlock twice.

Code running in interrupt context is running with interrupts disabled so should use a plain `spin_lock()`. If you have code in, for example, a common utility function which may be called from any context, use `spin_lock_irqsave()` which saves the current interrupt state in `flags`.

If a data structure is accessed only from process and bottom half context, you can optimise `spin_lock_irq()` to `spin_lock_bh()`. This allows interrupts to come in while you're holding the spinlock but doesn't allow bottom halves to run on exit from the interrupt routine; they will be deferred until the `spin_unlock_bh()`.

6 SCSI

In Linux 2.5.22, SCSI was still using a single-threaded bottom half. It seemed inappropriate that parts of the SCSI system were still serialised against random other parts of the kernel which could not possibly interact.

Linux 2.5.23 switched SCSI to use a tasklet. This was accepted and removed a source of contention on the `global_bh_lock`. As I investigated the softirq/tasklet framework, it became clear that there was really no reason to use a single tasklet for all SCSI cards.

Linux 2.5.25 replaced the tasklet with the `SCSI_SOFTIRQ`. This made it possible to service SCSI interrupts on all CPUs simultaneously. It was possible without major driver auditing because each request queue and each host is already locked by the SCSI midlayer.

SCSI does not suffer from the same kind of interrupt storms as networking. For one thing, it's not possible for an arbitrary user on the Internet to generate an interrupt on your scsi card, and SCSI cards typically do much more work per interrupt than a network card does. Nevertheless, it may be worth changing SCSI to a more NAPI-like framework so that if multiple interrupts come in on multiple CPUs for the same card, the work is batched onto one CPU rather than having multiple CPUs contend for the same locks. This optimisation is not being considered for 2.5, but it's on the long-term todo list.

7 Acknowledgements

References

[Mindcraft] Mindcraft Web and File Server Comparison: Microsoft Windows NT Server 4.0 and Red Hat Linux 5.2 Upgraded to the Linux 2.2.2 Kernel, `http://www.mindcraft.com/whitepapers/nts4rhlinux.html`, (1999).