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 scalability. The existing bottom halves were reimplemented as a special form of tasklet which preserved their semantics. In Linux 2.5.40, these old-style 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 the older mechanisms to the new ones requires an SMP audit. It is also necessary to understand 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

When writing kernel code, it is common to wish to defer work until later. There are many reasons for this. One is that it is inappropriate to do too much work with a lock held. Another may be to batch work to amortise the cost. A third may be to call a sleeping function, when scheduling at that point is not allowed.

The Linux kernel offers many different facilities for postponing work until later. Bottom Halves are for deferring work from interrupt context. Timers allow work to be deferred for at least a certain length of time. Work Queues allow work to be deferred to process context.

2 Contexts and Locking

Code in the Linux kernel runs in one of three contexts: Process, Bottom-half and Interrupt. Process context executes directly on behalf of a user process. All syscalls run in process context, for example. Interrupt handlers run in interrupt context. Softirqs, tasklets and timers all run in bottom-half context.

Linux is now a fairly scalable SMP kernel. To be scalable, it is necessary to allow many parts of the system to run at the same time. Many parts of the kernel which were previously serialised by the core kernel are now allowed to run simultaneously. Because of this, driver authors will almost certainly need to use some form of locking, or expand their existing locking.

Spinlocks should normally be used to protect ac-

cess to data structures and hardware. The normal way to do this is to call `spin_lock_irqsave(lock, flags)`, which saves the current interrupt state in `flags`, disables interrupts on the local CPU and acquires the spinlock.

Under certain circumstances, it is not necessary to disable local interrupts. For example, most filesystems only access their data structures from process context and acquire their spinlocks by calling `spin_lock(lock)`. If the code is only called in interrupt context, it is also not necessary to disable interrupts as Linux will not reenter an interrupt handler.

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

The consequence of failing to disable interrupts is a potential deadlock. If the code in process context is holding a spinlock and the code in interrupt context attempts to acquire the same spinlock, it will spin forever. For this reason, it is recommended that `spin_lock_irqsave()` is always used.

One way of avoiding locking altogether is to use per-CPU data structures. If only the local CPU touches the data, then disabling interrupts (using `local_irq_disable()` or `local_irq_save(flags)`) is sufficient to ensure data structure integrity. Again, this requires a certain amount of skill to use correctly.

3 Bottom Halves

3.1 History

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. It is important not to do too much work in the interrupt handler lest new interrupts be lost and other devices starved of the opportunity to proceed. This is a common issue in Unix-like operating systems. 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.

Linux 2.2 had 18 bottom half handlers. Networking, keyboard, console, SCSI and serial all used bottom halves directly and most of the rest of the kernel used them indirectly. Timers, as well as the immediate and periodic task queues, were run as a bottom half. Only one bottom half could be run at a time.

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 handle an interrupt at the same time, the bottom half layer was singly-threaded, so the work being done in the `NET_BH` was still not distributed across all the CPUs.

The softnet work multithreaded the bottom halves. This was done by replacing the bottom halves with softirqs and tasklets. The old-style 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 that assumed it while letting the network stack run simultaneously on all CPUs.

In 2.5, the old-style bottom halves were removed with all remaining users being converted to either softirqs or tasklets. The term 'Bottom Half' is now used to refer to code that is either a softirq or a tasklet, like the `spin_lock_bh()` function mentioned above.

It is amusing 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.

3.2 Implementing softirqs

On return from handling a hard interrupt, Linux checks to see whether any of the softirqs have been raised with the `raise_softirq()` call. 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. A softirq handler will execute on the same CPU that it is raised on. Of course, it's possible that this softirq will also be raised on another CPU and may execute first on that CPU, 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. Then the timers run, then network transmit and receive softirqs are run, then the SCSI softirq is run. Finally, low-priority tasklets are run.

3.3 Tasklets

Unlike softirqs, tasklets are dynamically allocated. Also unlike softirqs, a tasklet may run on only one CPU at a time. They are more SMP-friendly than the old-style bottom halves in that other tasklets may run at the same time. Tasklets have a weaker CPU affinity than softirqs. If the tasklet has already been scheduled on a different CPU, it will not be moved to another CPU if it's still pending.

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 instead. 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 handling an exceptional situation (eg network card with an unplugged cable). If the driver needs to be sure the tasklet is not executing during the exceptional situation, it is easier to disable the tasklet than to use a global variable to indicate that the tasklet shouldn't do its work.

3.4 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, if too much work is happening in softirq context, further softirq processing is handled by ksoftirqd.

The current definition of "too much work" is when a softirq is reactivated during a softirq processing run. Some argue this is too eager and ksoftirqd activation should be reserved for higher overload situations.

ksoftirqd is implemented as a set of threads, each of which is constrained to only run on a specific CPU. They are scheduled (at a very high priority) by the normal task scheduler. This implementation has the advantage that the time spent executing the bottom halves is accounted to a system task. It is thus possible for the user to see that the machine is overloaded with interrupt processing, and maybe take remedial action.

Although the work is now being done in process context rather than bottom half context, ksoftirqd sets up an environment identical to that found in bottom half context. Specifically, it executes the softirq handlers with local interrupts enabled and bottom halves disabled locally. Code which runs as a bottom half does not need to change for ksoftirqd to run it.

3.5 Problems

There are some subtle problems with using softirqs and tasklets. Some are obvious – driver writers must be more careful with locking. Other problems 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 it can't be processed before the next one is received.

Networking is particularly vulnerable to this. Assuming the interrupt controller distributes interrupts among CPUs in a round-robin fashion (this is the default for Intel IO-APICs), worst-case behaviour can be produced by simply ping-flooding an SMP machine. Interrupts will hit each CPU in turn, raising the network receive softirq. Each CPU will then attempt to deliver its packet 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.

Advanced network cards implement a feature called interrupt mitigation. Instead of interrupting

the CPU for each packet received, they queue packets in their on-card RAM and only generate an interrupt when a sufficient number of packets have arrived. The NAPI work, done by Jamal Hadi Salim, Alexey Kuznetsov and Thomas Olsson, simulates this in the OS.

When the network card driver receives a packet, it calls `disable_irq()` before passing the packet to the network stack's receive softirq. After the network stack has processed the packet, it asks the driver whether any more packets have arrived in the meantime. If none have, the driver calls `enable_irq()`. Otherwise, the network stack processes the new packets and leaves the network card's interrupt disabled. This effectively leaves the card in polling mode, and prevents any card from consuming too much of the system's resources.

4 Timers

A timer is another way of scheduling work to do later. Like a tasklet, a `timer_list` contains a function pointer and a data pointer to pass to that function. The main difference is that, as their name implies, their execution is delayed for a specified period of time. If the system is under load, the timer may not trigger at exactly the requested time, but it will wait at least as long as specified.

4.1 History

Originally there was an array of 32 timers. Like a softirq today, special permission was needed to get one. They were used for everything from SCSI, networking and the floppy driver to the 387 coprocessor, the QIC-02 tape driver and assorted drivers for old CD-ROMs.

Even by Linux 2.0, this was found to be insufficient 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 originally run from their own bottom half. The softnet work did not change this, so only one timer could run at a time. Timers were also serialised with 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 be run on multiple CPUs simultaneously. This was initially done with a per-CPU tasklet for timers, but 2.5.48 simplified this to use softirqs directly.

Any code which uses timers in 2.5 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.

4.2 Usage

The dynamic timers have always been controlled by the following interfaces: `add_timer()`, `del_timer()` and `mod_timer()`. 2.4 added the `del_timer_sync()` interface, which guarantees that when it returns, the timer is no longer running. 2.5.45 adds `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 the timer handler is permitted so much latitude in what it may do. It may `kfree()` the `timer_list` (or the struct embedding the `timer_list`). It may add, modify or delete the timer.

Many drivers assume that after calling `del_timer()`, they can free the `timer_list`, exit the module or shut down a device safely. On an SMP system, the timer can potentially still be running on another CPU after the `del_timer()` call returns. If the timer handler re-adds the timer after it has been deleted, it will continue to run indefinitely.

The `del_timer_sync()` function waits until the timer is no longer running on any CPU before it returns. Unfortunately, it can deadlock if the code that called `del_timer_sync()` is holding a lock which the timer handler routine needs to exit. Converting drivers to use this interface is an ongoing project.

Many users of timers are still unsafe in the 2.5 kernel, and a comprehensive audit is required. Fortu-

nately, most of the unsafe uses occur in module exit paths, so are only hit rarely. But the consequences of hitting the `del_timer()` race are catastrophic – the kernel will probably panic.

5 Task and Work Queues

Task queues were originally designed to replace the old-style 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. Also like tasklets and timers, they consist of a function pointer and a data argument to pass to that function.

Despite their name, they are not related to tasks (as in 'threads, tasks and processes'), which is partly why they were renamed to work queues in 2.5. This distinction was even more blurry in Linux 2.2 as there was a `tq_scheduler` which was run from the scheduler.

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 for using `tq_timer` was fine, but `tq_immediate`'s interface 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 some drivers actually missed 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, and 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 affects system performance negatively.

A feature introduced to the 2.4 kernel was a task queue that would be run by keventd in process context. The interface was much more sensible, involving a single call to `schedule_task()`. Various parts of the kernel had their own private task queues which would run when 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 once used `tq_immediate` should normally 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 to keventd, which executes it in process context. Interrupts and bottom halves are both enabled while the work queues are being run. Functions called from a work queue may call blocking operations, but this is discouraged as it prevents other users from running.

Work queues may be created and destroyed dynamically, normally on module init and exit. This is intended to provide a replacement for the private task queues that were available before. Creating a new work queue with `create_workqueue()` starts a new kernel thread per CPU. Work may then be scheduled to this work queue with the `queue_work()` and `queue_delayed_work()` interfaces.

Note that the routines for scheduling work to be performed by keventd are available to everyone, but the routines for using custom work queues are only available to modules which are distributed under a GPL-compatible licence. They should be used with discretion, anyway, as creating a thread for each processor quickly leads to a very large number of threads being created on 64-CPU systems.

6 SCSI

In Linux 2.5.22, the SCSI subsystem was still using an old-style 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 with it.

Linux 2.5.23 introduced the SCSI 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 hosts.

Linux 2.5.25 replaced the SCSI tasklet with the `SCSI_SOFTIRQ`. This made it possible to service SCSI interrupts on all CPUs simultaneously. It was not necessary to audit all the SCSI drivers 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 use one tasklet per host.

This would queue all work for one card to the same CPU in a similar way to how NAPI ties work from a network card to a CPU. Then there would be no contention between multiple CPUs attempting to access the same card. This optimisation is not being considered for 2.5, but it's on the long-term to-do list.

7 Acknowledgements

I'd like to thank those who did the work on the softirq/tasklet framework: Alexey Kuznetsov, Dave Miller, Ingo Molnar and Jamal Hadi Salim. Also the SCSI guys: James Bottomley and Douglas Gilbert for harassing me to do the work on `SCSI_SOFTIRQ` and taking my work into their tree.

For review of early versions of this paper and the talk, I'd like to thank Danielle Wilcox, Andrew Morton, Martin Petersen and the whole of Ottawa Canada Linux Users Group.

And finally, I'd also like to thank my employer Hewlett-Packard for funding me to speak at Linux.Conf.Au.

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).