Software interrupts and realtime

By **Jonathan Corbet**
October 17, 2012

The Linux kernel's software interrupt ("softirq") mechanism is a bit of a strange beast. It is an obscure holdover from the earliest days of Linux and a mechanism that few kernel developers ever deal with directly. Yet it is at the core of much of the kernel's most important processing. Occasionally softirqs make their presence known in undesired ways; it is not surprising that the kernel's frequent problem child — the realtime preemption patch set — has often run afoul of them. Recent versions of that patch set embody a new approach to the software interrupt problem that merits a look.

A softirq introduction

In the announcement for the 3.6.1-rt1 patch set, Thomas Gleixner described software interrupts this way:

> First of all, it's a conglomerate of mostly unrelated jobs, which run in the context of a randomly chosen victim w/o the ability to put any control on them.

The softirq mechanism is meant to handle processing that is almost — but not quite — as important as the handling of hardware interrupts. Softirqs run at a high priority (though with an interesting exception, described below), but with hardware interrupts enabled. They thus will normally preempt any work except the response to a "real" hardware interrupt.

Once upon a time, there were 32 hardwired software interrupt vectors, one assigned to each device driver or related task. Drivers have, for the most part, been detached from software interrupts for a long time — they still use softirqs, but that access has been laundered through intermediate APIs like tasklets and timers. In current kernels there are ten softirq vectors defined; two for tasklet processing, two for networking, two for the block layer, two for timers, and one each for the scheduler and read-copy-update processing. The kernel maintains a per-CPU bitmask indicating which softirqs need processing at any given time. So, for example, when a kernel subsystem calls `tasklet_schedule()`, the `TASKLET_SOFTIRQ` bit is set on the corresponding CPU and, when softirqs are processed, the tasklet will be run.

There are two places where software interrupts can "fire" and preempt the current thread. One of them is at the end of the processing for a hardware interrupt; it is common for interrupt handlers to raise softirqs, so it makes sense (for latency and optimal cache use) to process them as soon as hardware interrupts can be re-enabled. The other possibility is anytime that kernel code re-enables softirq processing (via a call to functions like `local_bh_enable()` or `spin_unlock_bh()`). The end result is that the accumulated softirq work (which can be substantial) is executed in the context of whichever process happens to be running at the wrong time; that is the "randomly chosen victim" aspect that Thomas was talking about.

Readers who have looked at the process mix on their systems may be wondering where the `ksoftirqd` processes fit into the picture. These processes exist to offload softirq processing when the load gets too heavy.

If the regular, inline softirq processing code loops ten times and still finds more softirqs to process (because they continue to be raised), it will wake the appropriate `ksoftirqd` process (there is one per CPU) and exit; that process will eventually be scheduled and pick up running softirq handlers. `Ksoftirqd` will also be poked if a softirq is raised outside of (hardware or software) interrupt context; that is necessary because, otherwise, an arbitrary amount of time might pass before softirqs are processed again. In older kernels, the `ksoftirqd` processes ran at the lowest possible priority, meaning that softirq processing was, depending on where it is being run, either the highest priority or the lowest priority work on the system. Since 2.6.23, `ksoftirqd` runs at normal user-level priority by default.

Softirqs in the realtime setting

On normal systems, the softirq mechanism works well enough that there has not been much motivation to change it, though, as described in "The new visibility of RCU processing," read-copy-update work has been moved into its own helper threads for the 3.7 kernel. In the realtime world, though, the concept of forcing arbitrary processes to do random work tends to be unpopular, so the realtime patches have traditionally pushed all softirq processing into separate threads, each with its own priority. That allowed, for example, the priority of network softirq handling to be raised on systems where networking needed realtime response; conversely, it could be lowered on systems where response to network events was less critical.

Starting with the 3.0 realtime patch set, though, that capability went away. It worked less well with the new approach to per-CPU data adopted then, and, as Thomas said, the per-softirq threads posed configuration problems:

> It's extremely hard to get the parameters right for a RT system in general. Adding something which is obscure as soft interrupts to the system designers todo list is a bad idea.

So, in 3.0, softirq handling looked very similar to how things are done in the mainline kernel. That improved the code and increased performance on untuned systems (by eliminating the context switch to the softirq thread), but took away the ability to finely tweak things for those who were inclined to do so. And realtime developers tend to be highly inclined to do just that. The result, naturally, is that some users complained about the changes.

In response, in 3.6.1-rt1, the handling of softirqs has changed again. Now, when a thread raises a softirq, the specific interrupt in question (network receive processing, say) is remembered by the kernel. As soon as the thread exits the context where software interrupts are disabled, that one softirq (and no others) will be run. That has the effect of minimizing softirq latency (since softirqs are run as soon as possible); just as importantly, it also ties processing of softirqs to the processes that generate them. A process raising networking softirqs will not be bogged down processing some other process's timers. That keeps the work local, avoids nondeterministic behavior caused by running another process's softirqs, and causes softirq processing to naturally run with the priority of the process creating the work in the first place.

There is an exception, of course: softirqs raised in hardware interrupt context cannot be handled in this way. There is no general way to associate a hardware interrupt with a specific thread, so it is not possible to force the responsible thread to do the necessary processing. The answer in this case is to just hand those softirqs to the `ksoftirqd` process and be done with it.

A logical next step, hinted at by Thomas, is to move from an environment where all softirqs are disabled to one where only specific softirqs are. Most code that disables softirq handling is only concerned with one specific handler; all the others could be allowed to run as usual. Going further, he adds: "the nicest solution would be to get rid of them completely." The elimination of the softirq mechanism has been on the "todo" list for a long time, but nobody has, yet, felt the pain strongly enough to actually do that work.

The nature of the realtime patch set has often been that its users feel the pain of mainline kernel shortcomings before the rest of us do. That has caused a great many mainline fixes and improvements to come from the realtime community. Perhaps that will eventually happen again for softirqs. For the time being, though, realtime users have an improved softirq mechanism that should give the desired results without the need for difficult low-level tuning. Naturally, Thomas is looking for people to test this change and report back on how well it works with their workloads.

(Log in to post comments)

Software interrupts and realtime
Posted Oct 17, 2012 19:22 UTC (Wed) by **fhuberts** (subscriber, #64683) [Link]

I really love solutions that in hindsight look obvious and easy...

Reply to this comment

Software interrupts and realtime
Posted Oct 17, 2012 21:14 UTC (Wed) by **sorpigal** (subscriber, #36106) [Link]

They're the stuff patents are made of... I hope someone friendly is paying attention.

Reply to this comment

Software interrupts and realtime
Posted Oct 18, 2012 1:45 UTC (Thu) by **xi** (subscriber, #70063) [Link]

Always supported phasing out softirq. By reducing number of possible kernel contexts there could be performance benefits too.

Reply to this comment

Why not just "cheat" and devote a whole core to the RT process?
Posted Oct 20, 2012 18:52 UTC (Sat) by **Richard_J_Neill** (subscriber, #23093) [Link]

On a multicore box, wouldn't it be easier just to dedicate a whole CPU core, 100% of the time to a particular process? In many of the RT use-cases I can think of, there's one task (often a single-threaded C-program) that needs to be able to run at top priority without interruption, while the entire rest of the OS, userspace and GUI could happily fit into the remaining cores.

Maybe I'm oversimplifying this, and it's certainly a bit wasteful (and won't work well for embedded), but for many common cases, such as low-latency audio processing, or avoiding dropouts, or data-acquistion, it would work just fine!

RT is hard when you have a mostly busy CPU (especially single-core), and multiple tasks, which might be relatively lightweight, require their small slice of CPU with hard-constraints on timing. But often, this isn't the case: we have just one critical task, and the system is mostly idle.

Reply to this comment

Why not just "cheat" and devote a whole core to the RT process?
Posted Oct 20, 2012 19:09 UTC (Sat) by **dlang** (guest, #313) [Link]

because it's really rare that your real time processing only needs to do computation on data it already has. Usually you need to do other things besides computation (like I/O of the audio, disk, etc)

At that point you are interacting with the rest of the system and you need to worry about delays and locking in the rest of the kernel.

Reply to this comment

Why not just "cheat" and devote a whole core to the RT process?
Posted Oct 29, 2012 18:20 UTC (Mon) by **cbf123** (guest, #74020) [Link]

Intel's networking fastpath basically just throws power consumption out the window and dedicates entire cpu cores to spinning on the network devices.

100% cpu usage, constantly, but you get really low-latency networking!

A somewhat less intrusive method is to direct only the interrupts you care about to the "isolated" cpu while leaving all the rest to be handled as normal.

Reply to this comment

Why not just "cheat" and devote a whole core to the RT process?
Posted Oct 31, 2012 19:32 UTC (Wed) by **XTF** (guest, #83255) [Link]

> On a multicore box, wouldn't it be easier just to dedicate a whole CPU core, 100% of the time to a particular process?

Doesn't this (kinda) happen automatically, if priorities are set right?

Reply to this comment

Software interrupts and realtime
Posted Oct 26, 2012 21:21 UTC (Fri) by **ParadoxUncreated** (guest, #87037) [Link]

Remember it`s not always about feeling pain, but about praising God.
Excellence and brilliance will usually do that.

Peace Be With You.

Reply to this comment

Software interrupts and realtime
Posted Oct 28, 2012 17:21 UTC (Sun) by **nix** (subscriber, #2304) [Link]

This theospammer has it so very wrong. Realtime developers spend so very much more time cursing than praising anything.

Reply to this comment

Software interrupts and realtime
Posted Oct 30, 2012 9:19 UTC (Tue) by **ernest** (guest, #2355) [Link]

Wow! I though just a moment I witnessed the creation of a new word there: Theospammer. It completely describes the person writing the silly, inappropriate and mostly, at least to me, very surprising comment above.
Really! I mean I read the theospam about ten times before I realized it really could only be that, and not some very deep thoughtful and witty comment.

Of course Theospammer wasn't a new word at all, apparently I don't read discussion groups enough. Still, what a beautiful word!

Ernest.

Reply to this comment

Software interrupts and realtime
Posted Oct 30, 2012 10:37 UTC (Tue) by **nix** (subscriber, #2304) [Link]

Well, I'd never seen the word before either. It's a fairly obvious portmanteau.

Reply to this comment

Example of syscall that trig a soft irq.
Posted Nov 2, 2012 9:11 UTC (Fri) by **polch** (guest, #87593) [Link]

"Now, when a thread raises a softirq, the specific interrupt in question (network receive processing, say) is remembered by the kernel"

Could you provide an example of syscll that trig a softirq ? For the network, for instance, i suppose that only calls that provide data to the kernel trig a soft irq (send, write, etc) ? But, i hardly imagine what kind of notification can trig a receive processing.

Regards.

Reply to this comment

Software interrupts and realtime
Posted Mar 5, 2013 15:58 UTC (Tue) by **ajith.adapa** (guest, #89100) [Link]

I am really confused with the usage of spin_lock_bh and spin_lock .. A simpler article would really help NOVICE to understand the various contexts available in Linux and how to protect data when executing in those contexts

Reply to this comment

Software interrupts and realtime
Posted May 18, 2014 4:55 UTC (Sun) by **a0273324@ti.com** (guest, #94150) [Link]

https://www.kernel.org/pub/linux/kernel/people/rusty/kern...

Reply to this comment