Four short stories about preempt_count()

By **Jonathan Corbet**
September 18, 2020

The discussion started out as a straightforward patch set from Thomas Gleixner making a minor change to how preemption counting is handled. The resulting discussion quickly spread out to cover a number of issues relevant to core-kernel development in surprisingly few messages; each of those topics merits a quick look, starting with how the preemption counter itself works. Sometimes a simple count turns out to not be as simple as it seems.

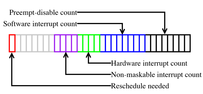
preempt_count()

In a multitasking system like Linux, no thread of execution is guaranteed exclusive access to the processor for as long as it would like to run; the kernel can (almost) always preempt a running thread in favor of one that has a higher priority. That new thread might be a different process, but it might also be a hardware interrupt or other outside event. In order to properly coordinate the running of all of a system's tasks, the kernel must keep track of the current execution state, including anything that has been preempted or which might prevent a thread from being preempted.

One piece of the infrastructure for that tracking is the preemption counter that is stored with every task in the system. That counter is accessed via the `preempt_count()` function which, in its generic form, looks like this:

```
    static __always_inline int preempt_count(void)
    {
        return READ_ONCE(current_thread_info()->preempt_count);
    }
```

The purpose of this counter is to describe the current state of whatever thread has been running, whether it can be preempted, and whether it is allowed to sleep. To do so, it must track a number of different states, so it is split into several sub-fields:



The least-significant byte tracks the nesting of `preempt_disable()` calls — the number of times that preemption has been disabled so far. The next three fields track the number of times the running thread has been interrupted by software, hardware, and non-maskable interrupts; they are all probably oversized for the number of interruptions that is likely to ever happen in real execution, but bits are not in short supply here. Finally, the most-significant bit indicates whether the kernel has decided that the current process needs to be scheduled out at the first opportunity.

A look at this value tells the kernel a fair amount about what is going on at any given time. For example, any non-zero value for `preempt_count` indicates that the current thread cannot be preempted by the scheduler; either preemption has been disabled explicitly or the CPU is currently servicing some sort of interrupt. In the same way, a non-zero value indicates that the current thread cannot sleep, since it is running in a context that must be allowed to run to completion. The "reschedule needed" bit tells the kernel that there is a higher-priority process that should be given the CPU at the first opportunity. This bit cannot be set unless `preempt_count` is non-zero; otherwise the kernel would have simply preempted the process rather than setting the bit and waiting.

Code throughout the kernel uses `preempt_count` to make decisions about which actions are possible at any given time. That, as it turns out, can be a bit of a problem for a few reasons.

Misleading counts

It is worth noting that `preempt_disable()` only applies when a thread is running within the kernel; user-space code is always preemptible. In the distant past, the kernel did not support preemption of kernel-space code at all; when that feature was added (as a way of improving latency), it was also made configurable. There are, as a result, still systems out there that are running without kernel preemption at all; it is a configuration that might make sense for some throughput-oriented workloads.

If kernel code cannot be preempted, there is little value in tracking calls to `preempt_disable()`; preemption is always disabled. So the kernel doesn't waste its time maintaining that information; in such kernels, the preempt-disable count portion of `preempt_count` is always zero. The `preemptible()` function will always return false, since the kernel is indeed not preemptible. It all seems to make sense.

There are some problems that result from this behavior, though. One is that functions like in_atomic(), which indicates whether the kernel is currently running in atomic context, do not behave in the same way. On a kernel with preemption configured in, calling `preempt_disable()` will cause `in_atomic()` to return true; if preemption is configured out, `preempt_disable()` is a no-op and `in_atomic()` will return false. This can cause `in_atomic()` to return false when, for example, spinlocks are held — a situation that is indeed an atomic context.

Gleixner, in his patch set, points out some other problems that result from this inconsistency and says that it is a problem overall:

> The lack of such indicators which work on all kernel configurations is a constant source of trouble because developers either do not understand the implications or try to work around this inconsistency in weird ways.

His solution is to remove the conditional compilation for the preemption-disable tracking, causing that counter to be maintained even in kernels that do not support kernel preemption. There is a cost in terms of increased execution time and code size on machines running those configurations, but Gleixner says that his benchmark testing "did not reveal any measurable impact" from the change.

Linus Torvalds was not convinced about the value of this change, noting that the code generation for spinlocks is indeed better when preemption is not possible. Gleixner reiterated that the effect is not measurable, and Torvalds conceded that the patch set does make the code simpler and "has its very clear charm".

Using preempt_count

Torvalds's larger complaint, though, was about code that uses `preempt_count` to change its behavior depending on the context. Such code, he said, is "always simply fundamentally wrong". Code that changes its behavior depending on the context should have that context passed in as a parameter, he said, so that callers know what to expect. Thus, the `GFP_ATOMIC` flag to the memory-allocation functions is acceptable, but changing behavior based on the return value from `in_atomic()` is not.

To an extent, there is general agreement with this position. Gleixner's patch posting included a section with future plans to audit and fix callers of functions like `in_atomic()` where, he says, "the number of buggy users is clearly the vast majority". Daniel Vetter added that, in his experience, "code that tries to cleverly adjust its behaviour depending upon the context it's running in is harder to understand and blows up in more interesting ways".

Paul McKenney, instead, argued that some code has to be able to operate properly in different contexts; the alternative would be an explosion of the API:

> Now perhaps you like the idea of call_rcu() for schedulable contexts, call_rcu_nosched() when preemption is disabled, call_rcu_irqs_are_disabled() when interrupts are disabled, call_rcu_raw_atomic() from contexts where (for example) raw spinlocks are held, and so on. However, from what I can see, most people instead consistently prefer that the RCU API instead be consolidated.

In response, Torvalds clarified that he sees core-kernel code as having different requirements than the rest. Core code has to deal with multiple contexts and should always do the right thing; code in drivers, instead, should not be changing its behavior based on its view of the context.

No hard conclusions were reached in this branch of the discussion. It does seem likely, though, that code with context-dependent behavior will be looked at more closely in the future.

Questioning high memory

One example of questionable use of `preempt_count`, in the crypto code, was pointed out early in the discussion by Gleixner; it changes a memory allocation mode in strange ways if it thinks that it's not currently preemptible. After some discussion, it turned out, according to Ard Biesheuvel, that the real purpose had been to avoid using `kmap_atomic()` if possible.

For those who are not immediately familiar with `kmap_atomic()`, a look at this article on high memory might be helpful. In short: 32-bit machines can only map a limited amount of memory into the kernel's address space; that amount is a little under 1GB on most architectures and configurations. Any memory that is not directly mapped is deemed "high memory"; any page in high memory must be explicitly (and temporarily) mapped into the kernel before the kernel can access its contents. The functions `kmap()` and `kmap_atomic()` exist to perform this mapping.

There are a few differences between those two functions, starting with the fact that only `kmap_atomic()` is callable in atomic context. Beyond that, though, `kmap_atomic()` is more efficient and is thus seen as being strongly preferable in any situation where it can be used, regardless of whether the caller was running in atomic context before the call (the CPU will always be running in atomic context while the mapping is in place). As Biesheuvel pointed out, though, the documentation doesn't reflect this preference and encourages the use of `kmap()` instead, so that is what he did.

There is another reason to prefer `kmap()`, he added; a call to `kmap_atomic()` disables preemption even on 64-bit architectures, where high memory does not exist and no temporary mapping need be made. Using it would have resulted in much of the WireGuard VPN code running with preemption disabled, entirely unnecessarily. Torvalds pointed out that there is a reason for this behavior: it is there to cause code to fail on 64-bit machines if it does things that would not work on 32-bit machines where high memory does exist. It is essentially a debugging aid that is making up for the fact that few developers run on 32-bit machines.

One way to optimize `kmap_atomic()` on 64-bit systems, Gleixner said, would be to make `kmap_atomic()` sections be preemptible — no longer atomic, in other words. This approach has been taken in the realtime kernels, he said, and "it's not that horrible". The cost would be to make `kmap_atomic()` a little slower on systems where high memory is in use.

That, it seems, is a cost that the development community is increasingly willing to pay; Torvalds replied that he would like to start removing `kmap()` support entirely. 32-Bit systems will be around for some time yet, but they are increasingly unlikely to be used in situations where lots of memory is needed. Or, as Torvalds put it: "It's not that 32-bit is irrelevant, it's that 32-bit with large amounts of memory is irrelevant". Every time that the cost of supporting high memory (which adds a significant amount of complexity to the memory-management subsystem) makes itself felt, the desire to take it out grows.

That said, nobody will be removing high-memory support right away. But a change that penalizes high-memory systems in favor of the systems that are being deployed now, such as making `kmap_atomic()` no longer be atomic, is increasingly likely to be accepted. Meanwhile, the other issues around `preempt_count` remain mostly unresolved, but it seems likely that, in the end, changes that bring correctness and reduce complexity will probably win out.

Index entries for this article

Kernel	kmap_atomic()
Kernel	Memory management/High memory
Kernel	Preemption

Four short stories about preempt_count()
Posted Sep 19, 2020 0:40 UTC (Sat) by **ms-tg** (subscriber, #89231) [Link]

Great, we'll-written and easy to follow! Thank you

Reply to this comment

Four short stories about preempt_count()
Posted Sep 19, 2020 8:36 UTC (Sat) by **darwi** (subscriber, #131202) [Link]

As Daniel succinctly stated, using in_interrupt(), in_irq(), or preemptible() in device drivers is either a symptom of horrible locking design or laziness. Usually it is both.

Reply to this comment

Four short stories about preempt_count()
Posted Sep 19, 2020 14:59 UTC (Sat) by **bluez_1134** (guest, #141537) [Link]

Would love to see more of these "stories" that explains the rationale behind core kernel design and implementation details in a very approachable manner. Really helpful for kernel newbies like me. :)

Reply to this comment

Four short stories about preempt_count()
Posted Sep 19, 2020 17:31 UTC (Sat) by **willy** (subscriber, #9762) [Link]

The story continues with Thomas posting kmap_temporary() and Linus liking everything but the name:

https://lore.kernel.org/linux-mm/20200919091751.011116649...

Reply to this comment

Four short stories about preempt_count()
Posted Sep 19, 2020 19:17 UTC (Sat) by **nevets** (subscriber, #11875) [Link]

Small correction about the *"reschedule needed"* bit. At least on x86. It starts out set, and is cleared when we need a schedule.

Why?

Because if an interrupt comes in and detects a schedule is needed but sees that the other bits in *preempt_count* are set, it clears this bit and returns.

Back on the task that is running, when it does a preempt_enable() the code there can do a dec and test on *preempt_count*. If it is zero, then it should call the scheduler.

```
  if (dec_and_test(preempt_count))
      schedule();
```

This lets us do a single test in preempt_enable() to know if it should schedule or not. Otherwise we need to do:

```
 if (dec_and_test(preempt_count) && need_resched_set())
   schedule();
```

Reply to this comment

Four short stories about preempt_count()
Posted Sep 19, 2020 19:22 UTC (Sat) by **nevets** (subscriber, #11875) [Link]

See `__preempt_count_dec_and_test()` in `arch/x86/include/asm/preempt.h` for details.

Reply to this comment

Four short stories about preempt_count()
Posted Sep 20, 2020 10:13 UTC (Sun) by **rustylife** (subscriber, #102864) [Link]

great article mr @corbet, tyvm!

Reply to this comment

Four short stories about preempt_count()
Posted Sep 25, 2020 13:01 UTC (Fri) by **flussence** (subscriber, #85566) [Link]

I do wish 32-bit systems with a quantity of memory appropriate for running Windows XP through 7 weren't considered a weird niche that requires highmem voodoo until the end of time. Is that really the best option?

Reply to this comment

Four short stories about preempt_count()
Posted Oct 7, 2020 16:26 UTC (Wed) by **zlynx** (subscriber, #2285) [Link]

As I remember things from the Old Days, Linux is limited to about 1 GB of RAM in 32 bit modes. Unless you use that voodoo.

I remember needing to do kernel rebuilds or picking the correct Redhat kernels to get 3 GB userspace.

Reply to this comment

Four short stories about preempt_count()
Posted Oct 11, 2020 15:34 UTC (Sun) by **mcortese** (guest, #52099) [Link]

32-bit systems suitable for old Windows versions are also suitable for old Linux versions. Newer Linux versions may well assume they are a weird niche.

Reply to this comment

Four short stories about preempt_count()
Posted Oct 14, 2020 9:49 UTC (Wed) by **flussence** (subscriber, #85566) [Link]

The problem is those old versions did too. Memory between 896MB–4GB has always been a kludge under x86 Linux.

Reply to this comment

Four short stories about preempt_count()
Posted Nov 24, 2020 8:55 UTC (Tue) by **cyph** (guest, #50776) [Link]

Why does preempt_count need to be maintained for every thread? Since a non-preemptible thread can never be scheduled away, wouldn't it be better to track preempt_count per-CPU instead?

Reply to this comment