RCU, cond_resched(), and performance regressions

By **Jonathan Corbet**
June 24, 2014

Performance regressions are a constant problem for kernel developers. A seemingly innocent change might cause a significant performance degradation, but only for users and workloads that the original developer has no access to. Sometimes these regressions can lurk for years until the affected users update their kernels and notice that things are running more slowly. The good news is that the development community is responding with more testing aimed at detecting performance regressions. This testing found a classic example of this kind of bug in 3.16; the bug merits a look as an example of how hard it can be to keep things working optimally for a wide range of users.

The birth of a regression

The kernel's read-copy-update (RCU) mechanism enables a great deal of kernel scalability by facilitating lock-free changes to data structures and batching of cleanup operations. A fundamental aspect of RCU's operation is the detection of "quiescent states" on each processor; a quiescent state is one in which no kernel code can hold a reference to any RCU-protected data structure. Initially, quiescent states were defined as times when the processor was running in user space, but things have gotten rather more complex since then. (See LWN's lengthy list of RCU articles for lots of details on how this all works).

The kernel's full tickless mode, which is only now becoming ready for serious use, can make the detection of quiescent states more difficult. A CPU running in the tickless mode will, due to the constraints of that mode, be running a single process. If that process stays within the kernel for a long time, no quiescent states will be observed. That, in turn, prevents RCU from declaring the end of a "grace period" and running the (possibly lengthy) set of accumulated RCU callbacks. Delayed grace periods can result in excessive latencies elsewhere in the kernel or, if things go really badly, out-of-memory problems.

One might argue (as some developers did) that code that loops in the kernel in this way already has serious problems. But such situations do come about. Eric Dumazet mentioned one: a process calling `exit()` when it has thousands of sockets open. Each of those open sockets will result in structures being freed via RCU; that can lead to a long list of work to be done while that same process is still closing sockets and, thus, preventing RCU processing by looping in the kernel.

RCU developer Paul McKenney put together a solution to this problem based on a simple insight: the kernel already has a mechanism for allowing other things to happen while some sort of lengthy operation is in progress. Code that is known to be prone to long loops will, on occasion, call `cond_resched()` to give the scheduler a chance to run a higher-priority process. In the tickless situation, there will be no higher-priority process, though, so, in current kernels, `cond_resched()` does nothing of any use in the tickless mode.

But kernel code can only call `cond_resched()` in places where it can handle being scheduled out of the CPU. So it cannot be running in an atomic context and, thus, cannot hold references to any RCU-protected data structures. In other words, a call to `cond_resched()` marks a quiescent state; all that is needed is to tell RCU about it.

As it happens, `cond_resched()` is called in a lot of performance-sensitive places, so it is not possible to add a lot of overhead there. So Paul did not call into RCU to signal a quiescent state with every `cond_resched()` call; instead, that function was modified to increment a per-CPU counter and, using that counter, only call into RCU once for every 256 (by default) `cond_resched()` calls. That appeared to fix the problem with minimal overhead, so the patch was merged during the 3.16 merge window.

Soon thereafter, Dave Hansen reported that one of his benchmarks (a program which opens and closes a lot of files while doing little else) had slowed down, and that, with bisection, he had identified the `cond_resched()` change as the culprit. Interestingly, the problem is not with `cond_resched()` itself, which remained fast as intended. Instead, the change caused RCU grace periods to happen more often than before; that caused RCU callbacks to be processed in smaller batches and led to increased contention in the slab memory allocator. By changing the threshold for quiescent states from every 256 `cond_resched()` calls to a much larger number, Dave was able to get back to a 3.15 level of performance.

Fixing the problem

One might argue that the proper fix is simply to raise that threshold for all users. But doing so doesn't just restore performance; it also restores the problem that the `cond_resched()` change was intended to fix. The challenge, then, is finding a way to fix one workload's problem without penalizing other workloads.

There is an additional challenge in that some developers would like to make `cond_resched()` into a complete no-op on fully preemptable kernels. After all, if the kernel is preemptable, there should be no need to poll for conditions that would require calling into the scheduler; preemption will simply take care of that when the need arises. So fixes that depend on `cond_resched()` continuing to do something may fail on preemptable kernels in the future.

Paul's first fix took the form of a series of patches making changes in a few places. There was still a check in `cond_resched()`, but that check took a different form. The RCU core was modified to take note when a specific processor holds up the conclusion of a grace period for an excessive period of time; when that condition was detected, a per-CPU flag would be set. Then, `cond_resched()` need only check that flag and, if it is set, note the passing of a quiescent period. That change reduced the frequency of grace periods, restoring much of the lost performance.

In addition, Paul introduced a new function called `cond_resched_rcu_qs()`, otherwise known as "the slow version of `cond_resched()`". By default, it does the same thing as ordinary `cond_resched()`, but the intent is that it would continue to perform the RCU grace period check even if `cond_resched()` is changed to skip that check — or to do nothing at all. The patch changed `cond_resched()` calls to `cond_resched_rcu_qs()` in a handful of strategic places where problems have been observed in the past.

This solution worked, but it left some developers unhappy. For those who are trying to get the most performance out of their CPUs, any overhead in a function like `cond_resched()` is too much. So Paul came up with a different approach that requires no checks in `cond_resched()` at all. Instead, when the RCU core notices that a CPU has held up the grace period for too long, it sends an inter-processor interrupt (IPI) to that processor. That IPI will be delivered when the target processor is not running in atomic context; it is, thus, another good time to note a quiescent state.

This solution might be surprising at first glance: IPIs are expensive and, thus, are not normally seen as the way to improve scalability. But this approach has two advantages: it removes the monitoring overhead from the performance-sensitive CPUs, and the IPIs only happen when a problem has been detected. So, most of the time, it should have no impact on CPUs running in the tickless mode at all. It would thus appear that this solution is preferable, and that this particular performance regression has been solved.

How good is good enough?

At least, it would appear that way if it weren't for the fact that Dave still observes a slowdown, though it is much smaller than it was before. The solution is, thus, not perfect, but Paul is inclined to declare victory on this one anyway:

> So, given that short grace periods help other workloads (I have the scars to prove it), and given that the patch fixes some real problems, and given that the large number for rcutree.jiffies_till_sched_qs got us within 3%, shouldn't we consider this issue closed?

Dave still isn't entirely happy with the situation; he noted that the regression is closer to 10% with the default settings, and said "This change of existing behavior removes some of the benefits that my system gets out of RCU." Paul responded that he is "not at all interested in that micro-benchmark becoming the kernel's straightjacket" and sent in a pull request including the second version of the fix. If there are any real-world workloads that are adversely affected by this change, he suggested, there are a number of ways to tune the system to mitigate the problem.

Regardless of whether this issue is truly closed or not, this regression demonstrates some of the hazards of kernel development on contemporary systems. Scalability pressures lead to complex code trying to ensure that everything happens at the right time with minimal overhead. But it will never be possible for a developer to test with all possible workloads, so there will often be one that shows a surprising performance regression in response to a change. Fixing one workload may well penalize another; making changes that do not hurt any workloads may be close to impossible. But, given enough testing and attention to the problems revealed by the tests, most problems can hopefully be found and corrected before they affect production users.

Index entries for this article
Kernel Performance regressions
Kernel Read-copy-update

(Log in to post comments)

RCU, cond_resched(), and performance regressions
Posted Jun 26, 2014 18:57 UTC (Thu) by **PaulMcKenney** (subscriber, #9624) [Link]

The last patch did seem to grow on Dave over time: “At this point, I'm satisfied with how e552592e is dealing with the original regression.”

So here is hoping that this particular regression is well and truly put to bed. ;–)

Reply to this comment

RCU, cond_resched(), and performance regressions
Posted Jun 26, 2014 23:46 UTC (Thu) by **gerdesj** (subscriber, #5446) [Link]

Benchmarks are great but they must have some practicable application to reality. I will grant that a synthetic exercise that concentrates attention on peculiarities might be a good tool but could be the tail wagging the dog.

What kind of workload would: "... opens and closes a lot of files while doing little else ..."?

Cheers
Jon

Reply to this comment

RCU, cond_resched(), and performance regressions
Posted Jun 27, 2014 2:50 UTC (Fri) by **mathstuf** (subscriber, #69389) [Link]

> What kind of workload would: "... opens and closes a lot of files while doing little else ..."?

Virus scanner (though this probably reads quite a bit too). File indexer (small, targeted reads for things like ID3 and EXIF tags). Emacs (I kid, I kid). Nothing else comes to mind at the moment.

Reply to this comment

RCU, cond_resched(), and performance regressions
Posted Jun 27, 2014 16:51 UTC (Fri) by **hansendc** (subscriber, #7363) [Link]

Jon,

Any real-world workload is a mix of the things we measure in a microbenchmark like this. The microbenchmark just breaks the workload down in to constituent pieces so that the pieces can be measured more easily.

Almost any Linux system does lots of opens and closes. On my system, one instance of 'top' can do thousands of them a second. Everyone should care about how fast these kinds of very common operations are, even if they can't measure the overhead when they get slower by a small amount.

Reply to this comment

RCU, cond_resched(), and performance regressions
Posted Jun 27, 2014 18:01 UTC (Fri) by **PaulMcKenney** (subscriber, #9624) [Link]

And one reason for doing this is that there might be a series of small changes, each of which provides (say) either a 0.5% improvement or a 0.5% degradation. Measuring these changes one at a time against a more realistic heavyweight application-based benchmark might show no measurable change for each, and taken together, their overall effect might be to cancel each other, thus providing no measurable change in performance.

In contrast, if you have a small tight benchmark, you might be able to sort the changes that improve performance from those that degrade performance. Of course, you should follow up by measuring the collection of changes that improved performance on a more realistic benchmark. After all, sometimes small changes interact in surprising ways.

Reply to this comment

RCU, cond_resched(), and performance regressions
Posted Jun 27, 2014 19:42 UTC (Fri) by **dlang** (guest, #313) [Link]

While everything you say is correct, it's also important to keep in mind that microbenchmarks tend to stress the system in ways that are different than normal use.

In this case, it's not that the individual opens and closes get slower, it's that when there are too many of them happening at once they end up getting slower.

so while a microbenchmark may show a 3% penalty, it's very possible that a real-world task that did 1/10 as many opens/closes (because it's doing real work in between) would not see a 0.3% penalty, but no measurable penalty

Reply to this comment

RCU, cond_resched(), and performance regressions
Posted Jun 27, 2014 19:54 UTC (Fri) by **PaulMcKenney** (subscriber, #9624) [Link]

Agreed. And the points you make are exactly why I said that you should follow up microbenchmark tests with tests on a more realistic benchmark.

Reply to this comment

RCU, cond_resched(), and performance regressions
Posted Jun 27, 2014 16:55 UTC (Fri) by **hansendc** (subscriber, #7363) [Link]

Yeah, I'm fairly happy with it at this point. On my system, things got better when scaling from 1->80 CPUs, and then marginally worse (and maybe noisier) from 81->160. But, that 81->160 range is where we fall over to using the hyperthreads and where tasks are not bound 1:1 with CPUs, so the scheduler comes in to play more. So, it's naturally noisier in that range. Also, the hyperthreads almost never scale as well as physical CPU cores, so scrutinizing scalability in this range isn't as interesting.

Reply to this comment

Alternatives to RCU?
Posted Jul 1, 2014 16:51 UTC (Tue) by **geertj** (subscriber, #4116) [Link]

Are there any alternatives to RCU? The implementation seems to be complex, and the heuristics to run RCU callbacks every X cond_resched() calls seems quite arbitrary. Also the fact that its run time is amortized makes it hard to analyze it seems to me.

Is there no way to get the code performant with just locks or with atomic instructions? Also are other kernels like FreeBSD using it?

I don't have anything personal against RCU I'm just genuinely curious.

Reply to this comment

Alternatives to RCU?
Posted Jul 1, 2014 20:34 UTC (Tue) by **dlang** (guest, #313) [Link]

Atomic instructions and locks are very slow.

RCU is being used because it allows software to avoid using them and still be correct.

Everywhere that RCU is being used or considered previously used Atomic instructions and locks, and it's being used because it's so much faster.

this is a case where doing things the 'simple' way limits it's performance, RCU looks at the problem a different way and comes at the problem sideways to get the desired result with much better performance, but unfortunantly at the cost of complexity.

RCU actually takes very little run time, it takes a lot of RAM to hold all the extra copies of the data (and all the complexity boils down to figuring out when these extra copies aren't needed any longer)

Reply to this comment

Alternatives to RCU?
Posted Jul 2, 2014 0:10 UTC (Wed) by **zlynx** (subscriber, #2285) [Link]

Think of RCU as MVCC in a database.

Older database engines just locked tables for updates. That was reliable but slow.

When MVCC came along it made databases *so much faster*.

It may be complicated but you wouldn't want to go back.

Reply to this comment

Alternatives to RCU?
Posted Jul 3, 2014 10:14 UTC (Thu) by **Wol** (subscriber, #4433) [Link]

> Older database engines just locked tables for updates.

What older databases? Do you mean RELATIONAL databases?

While I've had FILE locks available for ever, they're almost never used. RECORD (ie the equivalent of rows) locking has been available and used pretty much since day 1 (day 1 being 1 jan 1967 iirc :-)

Cheers,
Wol

Reply to this comment

Alternatives to RCU?
Posted Jul 3, 2014 10:45 UTC (Thu) by **zlynx** (subscriber, #2285) [Link]

Yes relational databases. I'm thinking something on a PC. I worked with some dbase, Alpha 4, Paradox, BTrieve and Access so it must have been one or more of those.

Anyway, perhaps it was using row level locking. Which made no difference because the queries I was writing needed that row, so if it was being updated it still brought all the other queries to a complete halt.

And it seems to me that the indexes got locked during writes too.

Reply to this comment

Alternatives to RCU?
Posted Jul 3, 2014 19:14 UTC (Thu) by **Cyberax** (★ **supporter** ★, #52523) [Link]

Yep, you're still in the stone age. Modern relational databases do not lock tables or records (unless explicitly requested) but use MVCC.

Reply to this comment

Alternatives to RCU?
Posted Jul 3, 2014 20:18 UTC (Thu) by **zlynx** (subscriber, #2285) [Link]

Was that a response to me? When in my original post I called out how much better MVCC is?

I am not in the Stone Age. I left MS Access behind *long ago*. 1998, I think.

Reply to this comment

Alternatives to RCU?
Posted Jul 3, 2014 20:23 UTC (Thu) by **Cyberax** (★ **supporter** ★, #52523) [Link]

No, that's a response to Wol who is a known fan of Pick database here.

Reply to this comment