The new visibility of RCU processing

October 10, 2012
This article was contributed by Paul McKenney

If you run a post-3.6 Linux kernel for long enough, you will likely see a process named `rcu_sched` or `rcu_preempt` or maybe even `rcu_bh` having consumed significant CPU time. If the system goes idle and all application processes exit, these processes might well have the largest CPU consumption of all the remaining processes. It is only natural to ask "what are these processes and why are they consuming so much CPU?"

The "what" part is easy: These are new kernel threads that handle RCU grace periods, previously handled mainly in softirq context. An "RCU grace period" is a period of time after which all pre-existing RCU read-side critical sections have completed, so that if an RCU updater removes a data element from an RCU-protected data structure and then waits for an RCU grace period, it may subsequently safely carry out destructive-to-readers actions, such as freeing the data element. RCU read-side critical sections begin with `rcu_read_lock()` and end with `rcu_read_unlock()`. Updaters can wait for an RCU grace period using `synchronize_rcu()`, or they can asynchronously schedule a function to be invoked after a grace period using `call_rcu()`. RCU's read-side primitives are extremely fast and scalable, so it can be quite helpful in read-mostly situations. For more detail on RCU, see: The RCU API, 2010 Edition, What is RCU, Fundamentally?, this set of slides [PDF], and the RCU home page.

Quick Quiz 1: Why would latency be reduced by moving RCU work to a kthread? And why would anyone care about latency on huge machines?
Answer

The reason for moving RCU grace-period handling to a kernel thread was to improve real-time latency (both interrupt latency and scheduling latency) on huge systems by allowing RCU's grace-period initialization to be preempted: Without preemption, this initialization can inflict more than 200 microseconds of latency on huge systems. In addition, this change will very likely also improve RCU's energy efficiency while also simplifying the code. These potential simplifications are due to the fact that kernel threads make it easier to guarantee forward progress, avoiding hangs in cases where all CPUs are asleep and thus ignoring the current grace period, as confirmed by Paul Walmsley. But the key point here is that these kernel threads do not represent new overhead: Instead, overhead that used to be hidden in softirq context is now visible in kthread context.

Quick Quiz 2: Wow!!! If `hackbench` does a million grace periods in ten minutes, just how many does something like `rcutorture` do?
Answer

Now for "why so much CPU?", which is the question Ingo Molnar asked immediately upon seeing more than three minutes of CPU time consumed by `rcu_sched` after running a couple hours of kernel builds. The answer is that Linux makes heavy use of RCU, so much so that running `hackbench` for ten minutes can result in almost one million RCU grace periods—and more than thirty seconds of CPU time

consumed by `rcu_sched`. This works out to about thirty microseconds per grace period, which is anything but excessive, considering the amount of work that grace periods do.

As it turns out, the CPU consumption of `rcu_sched`, `rcu_preempt`, and `rcu_bh` is often roughly equal to the sum of that of the `ksoftirqd` threads. Interestingly enough, in 3.6 and earlier, some of the RCU grace-period overhead would have been charged to the `ksoftirqd` kernel threads.

But CPU overhead per grace period is only part of the story. RCU works hard to process multiple updates (e.g., `call_rcu()` or `synchronize_rcu()` invocations) with a single grace period. It is not hard to achieve more than one hundred updates per grace period, which results in a per-update overhead of only about 300 nanoseconds, which is not bad at all. Furthermore, workloads having well in excess of one thousand updates per grace period have been observed.

Of course, the per-grace-period CPU overhead does vary, and with it the per-update overhead. First, the greater the number of possible CPUs (as given at boot time by `nr_cpu_ids`), the more work RCU must do when initializing and cleaning up grace periods. This overhead grows fairly slowly, with additional work required with the addition of each set of 16 CPUs, though this number varies depending on the `CONFIG_RCU_FANOUT_LEAF` kernel configuration parameter and also on the `rcutree.rcu_fanout_leaf` kernel boot parameter.

Second, the greater the number of idle CPUs, the more work RCU must do when forcing quiescent states. Yes, the busier the system, the less work RCU needs to do! The reason for the extra work is that RCU is not permitted to disturb idle CPUs for energy-efficiency reasons. RCU must therefore probe a per-CPU data structure to read out idleness state during each grace period, likely incurring a cache miss on each such probe.

Third and finally, the overhead will vary depending on CPU clock rate, memory-system performance, virtualization overheads, and so on. All that aside, I see per-grace-period overheads ranging from 15 to 100 microseconds on the systems I use. I suspect that a system with thousands of CPUs might consume many hundreds of microseconds, or perhaps even milliseconds, of CPU time for each grace period. On the other hand, such a system might also handle a very large number of updates per grace period.

In conclusion, the `rcu_sched`, `rcu_preempt`, and `rcu_bh` CPU overheads should not be anything to worry about. They do not represent new overhead inflicted on post-3.6 kernels, but rather better accounting of the same overhead that RCU has been incurring all along.

Quick Quiz 3: Now that all of the RCU overhead is appearing on the `rcu_sched`, `rcu_preempt`, and `rcu_bh` kernel threads, we should be able to more easily identify that overhead and optimize RCU, right?
Answer

Acknowledgments

I owe thanks to Ingo Molnar for first noting this issue and the need to let the community know about it. We all owe a debt of gratitude to Steve Dobbelstein, Stephen Rothwell, Jon Corbet, and Paul Walmsley for their help in making this article human-readable. I am grateful to Jim Wasko for his support of this effort.

Answers to Quick Quizzes

Quick Quiz 1: Why would latency be reduced by moving RCU work to a kthread? And why would anyone care about latency on huge machines?

Answer: Moving work from softirq to a kthread allows that work to be more easily preempted, and this preemption reduces scheduling latency. Low scheduling latency is of course important in real-time applications, but it also helps reduce OS jitter. Low OS jitter is critically important to certain types of high-performance-computing (HPC) workloads, which is the type of workload that tends to be run on huge systems.

Back to Quick Quiz 1.

Quick Quiz 2: Wow!!! If `hackbench` does a million grace periods in ten minutes, just how many does something like `rcutorture` do?

Answer: Actually, `rcutorture` tortures RCU in many different ways, including overly long read-side critical sections, transitions to and from idle, and CPU hotplug operations. Thus, a typical `rcutorture` run would probably "only" do about 100,000 grace periods in a ten-minute interval.

In short, the grace-period rate can vary widely depending on your hardware, kernel configuration, and workload.

Back to Quick Quiz 2.

Quick Quiz 3: Now that all of the RCU overhead is appearing on the `rcu_sched`, `rcu_preempt`, and `rcu_bh` kernel threads, we should be able to more easily identify that overhead and optimize RCU, right?

Answer: Yes and no. Yes, it is easier to optimize that which can be easily measured. But no, not all of RCU's overhead appears on the `rcu_sched`, `rcu_preempt`, and `rcu_bh` kernel threads. Some of it still appears on the `ksoftirqd` kernel threads, and some of it is spread over other tasks.

Still, yes, the greater visibility should be helpful.

Back to Quick Quiz 3.

The new visibility of RCU processing
Posted Oct 11, 2012 17:33 UTC (Thu) by **kjp** (subscriber, #39639) [Link]

Why are there three threads?

> **The new visibility of RCU processing**
> Posted Oct 11, 2012 18:14 UTC (Thu) by **PaulMcKenney** (subscriber, #9624) [Link]
>
> There is one thread for each flavor of RCU, which means three on CONFIG_PREEMPT=y kernels (rcu_bh, rcu_preempt, and rcu_sched) and two on CONFIG_PREEMPT=n kernels (rcu_bh and rcu_sched, with rcu_preempt mapping onto rcu_sched).
>
> In theory, a single thread could handle all the RCU flavors, but when I tried this the code got quite ugly.

Might this bug be related?
Posted Oct 17, 2012 12:56 UTC (Wed) by **DOT** (guest, #58786) [Link]

Might this bug be related to the RCU work that is now being done? The computer appears to hang for a some time and it spews messages like this:

```
INFO: rcu_sched detected stalls on CPUs/tasks: { 1} (detected by 0, t=15603 jiffies)
```

> **Might this bug be related?**
> Posted Oct 18, 2012 14:01 UTC (Thu) by **PaulMcKenney** (subscriber, #9624) [Link]
>
> Given that the bug seems to have happened on a 3.2 kernel, I suspect that the kthreading is not responsible. ;-)

> But here are some commits that have fixed RCU-related CPU stall warning issues in reverse chronological order:
>
> ```
> c96ea7cf: rcu: Avoid spurious RCU CPU stall warnings *
> c8020a67: rcu: Protect rcu_node accesses during CPU stall warnings
> 5fd4dc06: rcu: Avoid rcu_print_detail_task_stall_rnp() segfault
> a10d206e: rcu: Fix day-one dyntick-idle stall-warning bug *
> 1c17e4d4: rcu: Prevent uninitialized string in RCU CPU stall info
> 13cfcca0: rcu: Set RCU CPU stall times via sysfs
> 2d1dc9a6: rcu: Remove redundant rcu_cpu_stall_suppress declaration
> ```
>
> The two commits marked with "*" are the most likely to address this bug.