Hierarchical RCU

November 4, 2008

This article was contributed by Paul McKenney

Introduction

Read-copy update (RCU) is a synchronization mechanism that was added to the Linux kernel in October of 2002. RCU improves scalability by allowing readers to execute concurrently with writers. In contrast, conventional locking primitives require that readers wait for ongoing writers and vice versa. RCU ensures coherence for read accesses by maintaining multiple versions of data structures and ensuring that they are not freed until all pre-existing read-side critical sections complete. RCU relies on efficient and scalable mechanisms for publishing and reading new versions of an object, and also for deferring the collection of old versions. These mechanisms distribute the work among read and update paths in such a way as to make read paths extremely fast. In some cases (non-preemptable kernels), RCU's read-side primitives have zero overhead.

Although Classic RCU's read-side primitives enjoy excellent performance and scalability, the update-side primitives which determine when pre-existing read-side critical sections have finished, were designed with only a few tens of CPUs in mind. Their scalability is limited by a global lock that must be acquired by each CPU at least once during each grace period. Although Classic RCU actually scales to a couple of hundred CPUs, and can be tweaked to scale to roughly a thousand CPUs (but at the expense of extending grace periods), emerging multicore systems will require it to scale better.

In addition, Classic RCU has a sub-optimal dynticks interface, with the result that Classic RCU will wake up every CPU at least once per grace period. To see the problem with this, consider a 16-CPU system that is sufficiently lightly loaded that it is keeping only four CPUs busy. In a perfect world, the remaining twelve CPUs could be put into deep sleep mode in order to conserve energy. Unfortunately, if the four busy CPUs are frequently performing RCU updates, those twelve idle CPUs will be awakened frequently, wasting significant energy. Thus, any major change to Classic RCU should also leave sleeping CPUs lie.

Both the existing and the proposed implementation have have Classic RCU semantics and identical APIs, however, the old implementation will be called “classic RCU” and the new implementation will be called “tree RCU”.

5. Towards a Greener RCU Implementation
6. State Machine
7. Use Cases
8. Testing

These sections are followed by concluding remarks and the answers to the Quick Quizzes.

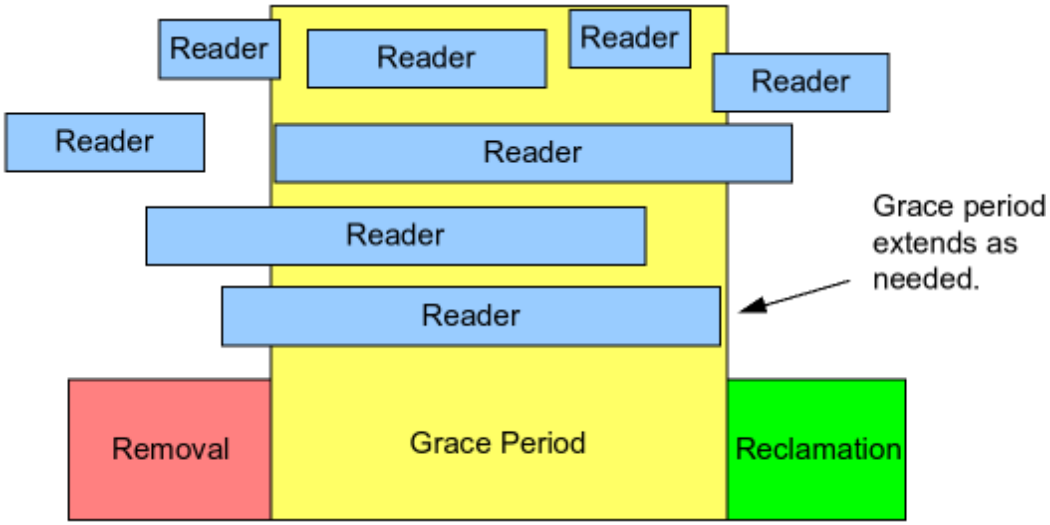
Review of RCU Fundamentals

In its most basic form, RCU is a way of waiting for things to finish. Of course, there are a great many other ways of waiting for things to finish, including reference counts, reader-writer locks, events, and so on. The great advantage of RCU is that it can wait for each of (say) 20,000 different things without having to explicitly track each and every one of them, and without having to worry about the performance degradation, scalability limitations, complex deadlock scenarios, and memory-leak hazards that are inherent in schemes using explicit tracking.

In RCU's case, the things waited on are called "RCU read-side critical sections". An RCU read-side critical section starts with an `rcu_read_lock()` primitive, and ends with a corresponding `rcu_read_unlock()` primitive. RCU read-side critical sections can be nested, and may contain pretty much any code, as long as that code does not explicitly block or sleep (although a special form of RCU called "SRCU" does permit general sleeping in SRCU read-side critical sections). If you abide by these conventions, you can use RCU to wait for *any* desired piece of code to complete.

RCU accomplishes this feat by indirectly determining when these other things have finished, as has been described elsewhere for Classic RCU and realtime RCU.

In particular, as shown in the following figure, RCU is a way of waiting for pre-existing RCU read-side critical sections to completely finish, including memory operations executed by those critical sections.

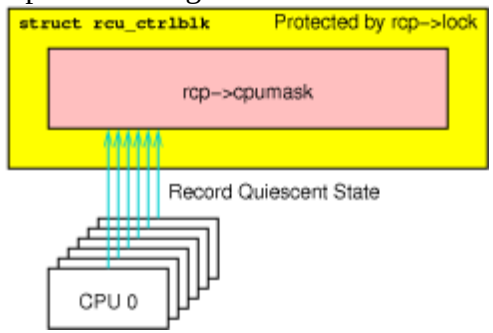


However, note that RCU read-side critical sections that begin after the beginning of a given grace period can and will extend beyond the end of that grace period.

The following section gives a very high-level view of how the Classic RCU implementation operates.

Brief Overview of Classic RCU Implementation

The key concept behind the Classic RCU implementation is that Classic RCU read-side critical sections are confined to kernel code and are not permitted to block. This means that any time a given CPU is seen either blocking, in the idle loop, or exiting the kernel, we know that all RCU read-side critical sections that were previously running on that CPU must have completed. Such states are called “quiescent states”, and after each CPU has passed through at least one quiescent state, the RCU grace period ends.



Classic RCU's most important data structure is the `rcu_ctrlblk` structure, which contains the `->cpumask` field, which contains one bit per CPU. Each CPU's bit is set to one at the beginning of each grace period, and each CPU must clear its bit after it passes through a quiescent state. Because multiple CPUs might want to clear their bits concurrently, which would corrupt the `->cpumask` field, a `->lock` spinlock is used to protect `->cpumask`, preventing any such corruption. Unfortunately, this spinlock can also suffer extreme contention if there are more than a few hundred CPUs, which might soon become quite common if multicore trends continue. Worse yet, the fact that *all* CPUs must clear their own bit means that CPUs are not permitted to sleep through a grace period, which limits Linux's ability to conserve power.

The next section lays out what we need from a new non-real-time RCU implementation.

RCU Desiderata

The list of RCU desiderata called out at LCA2005 for real-time RCU is a very good start:

1. Deferred destruction, so that an RCU grace period cannot end until all pre-existing RCU read-side critical sections have completed.
2. Reliable, so that RCU supports 24x7 operation for years at a time.
3. Callable from irq handlers.
4. Contained memory footprint, so that mechanisms exist to expedite grace periods if there are too many callbacks. (This is weakened from the LCA2005 list.)
5. Independent of memory blocks, so that RCU can work with any conceivable memory allocator.
6. Synchronization-free read side, so that only normal non-atomic instructions operating on CPU- or task-local memory are permitted. (This is strengthened from the LCA2005 list.)
7. Unconditional read-to-write upgrade, which is used in several places in the Linux kernel where the update-side lock is acquired within the RCU read-side critical section.
8. Compatible API.

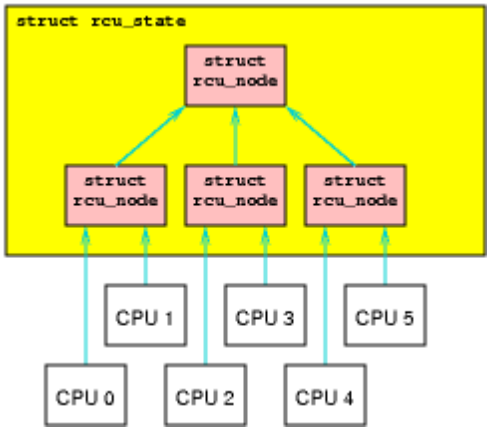
Because this is not to be a real-time RCU, the requirement for preemptable RCU read-side critical sections can be dropped. However, we need to add a few more requirements to account for changes over the past few years:

1. Scalability with extremely low internal-to-RCU lock contention. RCU must support at least 1,024 CPUs gracefully, and preferably at least 4,096.
2. Energy conservation: RCU must be able to avoid awakening low-power-state dynticks-idle CPUs, but still determine when the current grace period ends. This has been implemented in real-time RCU, but needs serious simplification.
3. RCU read-side critical sections must be permitted in NMI handlers as well as irq handlers. Note that preemptable RCU was able to avoid this requirement due to a separately implemented `synchronize_sched()`.
4. RCU must operate gracefully in face of repeated CPU-hotplug operations. This is simply carrying forward a requirement met by both classic and real-time.
5. It must be possible to wait for all previously registered RCU callbacks to complete, though this is already provided in the form of `rcu_barrier()`.
6. Detecting CPUs that are failing to respond is desirable, to assist diagnosis both of RCU and of various infinite loop bugs and hardware failures that can prevent RCU grace periods from ending.
7. Extreme expediting of RCU grace periods is desirable, so that an RCU grace period can be forced to complete within a few hundred microseconds of the last relevant RCU read-side critical second completing. However, such an operation would be expected to incur severe CPU overhead, and would be primarily useful when carrying out a long sequence of operations that each needed to wait for an RCU grace period.

The most pressing of the new requirements is the first one, scalability. The next section therefore describes how to make order-of-magnitude reductions in contention on RCU's internal locks.

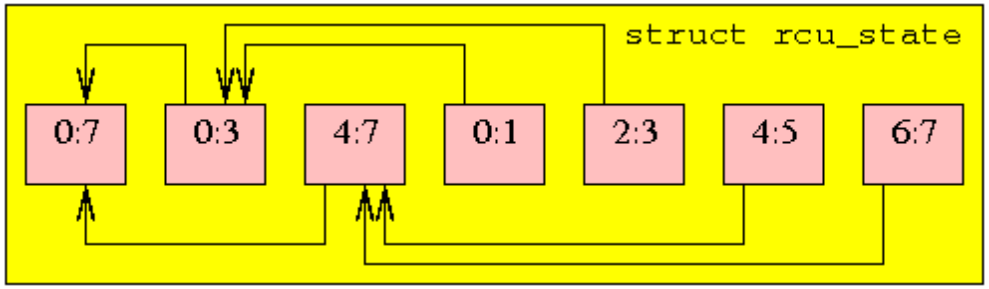
Towards a More Scalable RCU Implementation

One effective way to reduce lock contention is to create a hierarchy, as shown in the following figure. Here, each of the four `rcu_node` structures has its own lock, so that only CPUs 0 and 1 will acquire the lower left `rcu_node`'s lock, only CPUs 2 and 3 will acquire the lower middle `rcu_node`'s lock, and only CPUs 4 and 5 will acquire the lower right `rcu_node`'s lock. During any given grace period, only one of the CPUs accessing each of the lower `rcu_node` structures will access the upper `rcu_node`, namely, the last of each pair of CPUs to record a quiescent state for the corresponding grace period.



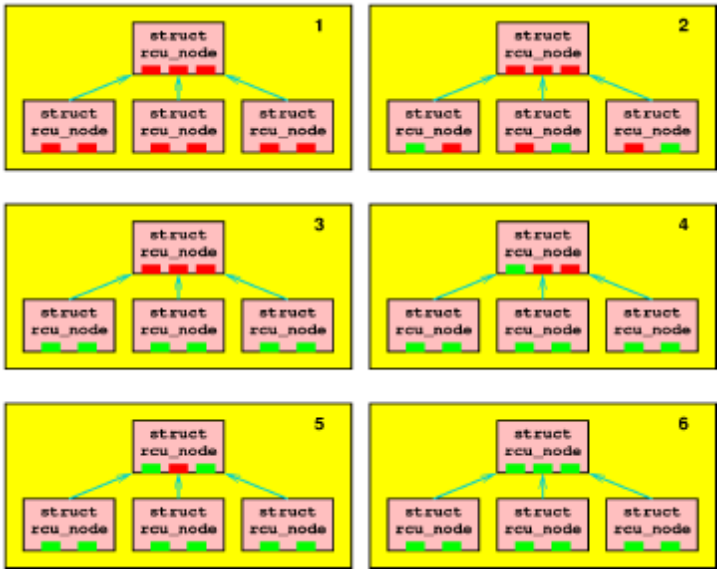
This results in a significant reduction in lock contention: instead of six CPUs contending for a single lock each grace period, we have only three for the upper `rcu_node`'s lock (a reduction of 50%) and only two for each of the lower `rcu_nodes`' locks (a reduction of 67%).

The tree of `rcu_node` structures is embedded into a linear array in the `rcu_state` structure, with the root of the tree in element zero, as shown below for an eight-CPU system with a three-level hierarchy. The arrows link a given `rcu_node` structure to its parent. Each `rcu_node` indicates the range of CPUs covered, so that the root node covers all of the CPUs, each node in the second level covers half of the CPUs, and each node in the leaf level covering a pair of CPUs. This array is allocated statically at compile time based on the value of `NR_CPUS`.

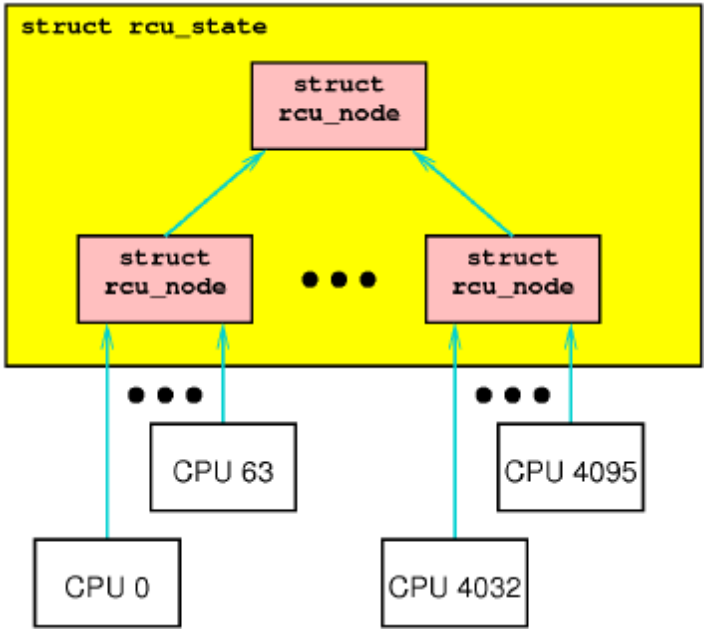


The following sequence of six figures shows how grace periods are detected. In the first figure, no CPU has yet passed through a quiescent state, as indicated by the red rectangles. Suppose that all six CPUs simultaneously try to tell RCU that they have passed through a quiescent state. Only one of each pair will be able to acquire the lock on the corresponding lower `rcu_node`, and so the second figure shows the result if the lucky CPUs are numbers 0, 3, and 5, as indicated by the green rectangles. Once these lucky CPUs have finished, then the other CPUs will acquire the lock, as shown in the third figure. Each of these CPUs will see that

they are the last in their group, and therefore all three will attempt to move to the upper `rcu_node`. Only one at a time can acquire the upper `rcu_node` structure's lock, and the fourth, fifth, and sixth figures show the sequence of states assuming that CPU 1, CPU 2, and CPU 4 acquire the lock in that order. The sixth and final figure in the group shows that all CPUs have passed through a quiescent state, so that the grace period has ended.



In the above sequence, there were never more than three CPUs contending for any one lock, in happy contrast to Classic RCU, where all six CPUs might contend. However, even more dramatic reductions in lock contention are possible with larger numbers of CPUs. Consider a hierarchy of `rcu_node` structures, with 64 lower structures and 64*64=4,096 CPUs, as shown in the following figure.



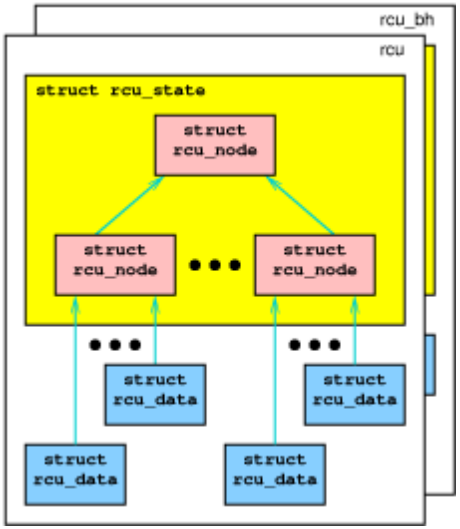
Here each of the lower `rcu_node` structures' locks are acquired by 64 CPUs, a 64-times reduction from the 4,096 CPUs that would acquire Classic RCU's single global lock. Similarly, during a given grace period, only one CPU from each of the lower `rcu_node` structures will acquire the upper `rcu_node` structure's lock, which is again a 64x reduction from the contention level that would be experienced by Classic RCU running on a 4,096-CPU system.

Quick Quiz 1: Wait a minute! With all those new locks, how do you avoid deadlock?

Quick Quiz 2: Why stop at a 64-times reduction? Why not go for a few orders of magnitude instead?

Quick Quiz 3: But I don't care about McKenney's lame excuses in the answer to Quick Quiz 2!!! I want to get the number of CPUs contending on a single lock down to something reasonable, like sixteen or so!!!

The implementation maintains some per-CPU data, such as lists of RCU callbacks, organized into `rcu_data` structures. In addition, rcu (as in `call_rcu()`) and rcu_bh (as in `call_rcu_bh()`) each maintain their own hierarchy, as shown in the following figure.



Quick Quiz 4: OK, so what is the story with the colors?

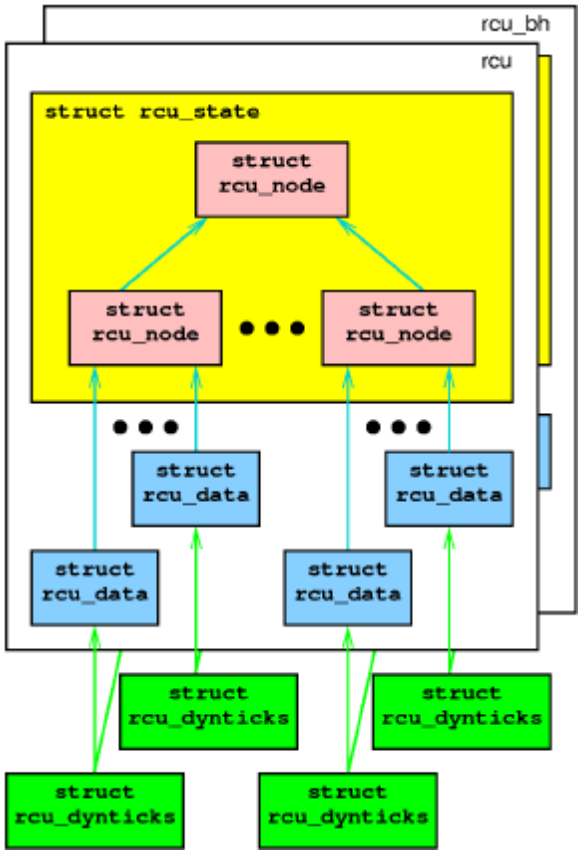
The next section discusses energy conservation.

Towards a Greener RCU Implementation

As noted earlier, an important goal of this effort is to leave sleeping CPUs lie in order to promote energy conservation. In contrast, classic RCU will happily awaken each and every sleeping CPU at least once per grace period in some cases, which is suboptimal in the case where a small number of CPUs are busy doing RCU updates and the majority of the CPUs are mostly idle. This situation occurs frequently in systems sized for peak loads, and we need to be able to accommodate it gracefully. Furthermore, we need to fix a long-standing bug in Classic RCU where a dynticks-idle CPU servicing an interrupt containing a long-running RCU read-side critical section will fail to prevent an RCU grace period from ending.

Quick Quiz 5: Given such an egregious bug, why does Linux run at all?

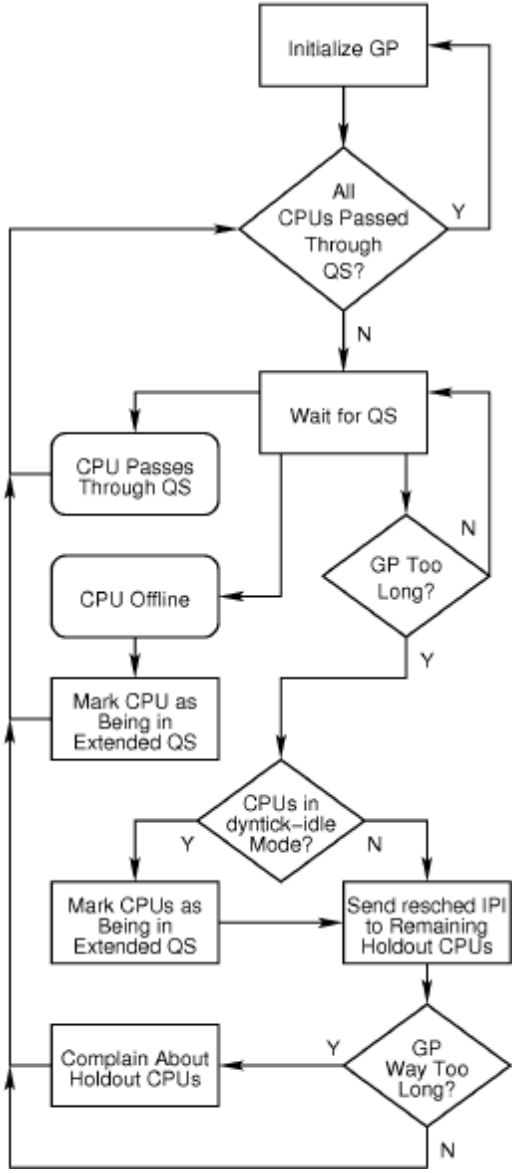
This is accomplished by requiring that all CPUs manipulate counters located in a per-CPU `rcu_dynticks` structure. Loosely speaking, these counters have even-numbered values when the corresponding CPU is in dynticks idle mode, and have odd-numbered values otherwise. RCU thus needs to wait for quiescent states only for those CPUs whose `rcu_dynticks` counters are odd, and need not wake up sleeping CPUs, whose counters will be even. As shown in the following diagram, each per-CPU `rcu_dynticks` is shared by the “rcu” and “rcu_bh” implementations.



The following section presents a high-level view of the RCU state machine.

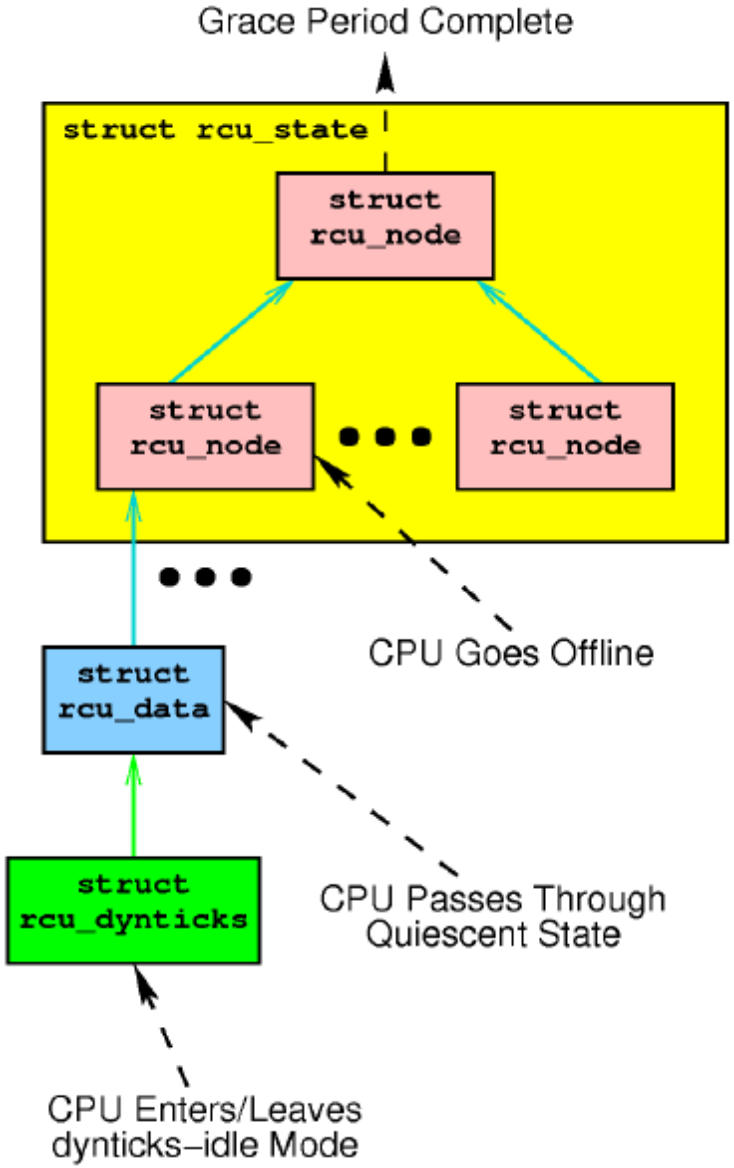
State Machine

At a sufficiently high level, Linux-kernel RCU implementations can be thought of as high-level state machines as shown in the following schematic:



The common-case path through this state machine on a busy system goes through the two uppermost loops, initializing at the beginning of each grace period (GP), waiting for quiescent states (QS), and noting when each CPU passes through its first quiescent state for a given grace period. On such a system, quiescent states will occur on each context switch, or, for CPUs that are either idle or executing user-mode code, each scheduling-clock interrupt. CPU-hotplug events will take the state machine through the “CPU Offline” box, while the presence of “holdout” CPUs that fail to pass through quiescent states quickly enough will exercise the path through the “Send resched IPIs to Holdout CPUs” box. RCU implementations that avoid unnecessarily awakening dyntick-idle CPUs will mark those CPUs as being in an extended quiescent state, taking the “Y” branch out of the “CPUs in dyntick-idle Mode?” decision diamond (but note that CPUs in dyntick-idle mode will *not* be sent resched IPIs). Finally, if `CONFIG_RCU_CPU_STALL_DETECTOR` is enabled, truly excessive delays in reaching quiescent states will exercise the “Complain About Holdout CPUs” path.

The events in the above state schematic interact with different data structures, as shown below:



However, the state schematic does not directly translate into C code for any of the RCU implementations. Instead, these implementations are coded as an event-driven system within the kernel. Therefore, the following section describes some “use cases”, or ways in which the RCU algorithm traverses the above state schematic as well as the relevant data structures.

Use Cases

This section gives an overview of several “use cases” within the RCU implementation, listing the data structures touched and the functions invoked. The use cases are as follows:

1. Start a new grace period.

2. Pass through a quiescent state.
3. Announce a quiescent state to RCU.
4. Enter and leave dynticks idle mode.
5. Interrupt from dynticks idle mode.
6. NMI from dynticks idle mode.
7. Note that a CPU is in dynticks idle mode.
8. Offline a CPU.
9. Online a CPU.
10. Detect a too-long grace period.

Each of these use cases is described in the following sections.

Start a New Grace Period

The `rcu_start_gp()` function starts a new grace period. This function is invoked when a CPU having callbacks waiting for a grace period notices that no grace period is in progress.

The `rcu_start_gp()` function updates state in the `rcu_state` and `rcu_data` structures to note the newly started grace period, acquires the `->onoff` lock (and disables irqs) to exclude any concurrent CPU-hotplug operations, sets the bits in all of the `rcu_node` structures to indicate that all CPUs (including this one) must pass through a quiescent state, and finally releases the `->onoff` lock.

The bit-setting operation is carried out in two phases. First, the non-leaf `rcu_node` structures' bits are set without holding any additional locks, and then finally each leaf `rcu_node` structure's bits are set in turn while holding that structure's `->lock`.

Quick Quiz 6: But what happens if a CPU tries to report going through a quiescent state (by clearing its bit) before the bit-setting CPU has finished?

Quick Quiz 7: And what happens if *all* CPUs try to report going through a quiescent state before the bit-setting CPU has finished, thus ending the new grace period before it starts?

Pass Through a Quiescent State

The rcu and rcu_bh flavors of RCU have different sets of quiescent states. Quiescent states for rcu are context switch, idle (either dynticks or the idle loop), and user-mode execution, while quiescent states for rcu_bh are any code outside of softirq with interrupts enabled. Note that an quiescent state for rcu is also a quiescent state for rcu_bh. Quiescent states for rcu are recorded by invoking `rcu_qsctr_inc()`, while quiescent states for rcu_bh are recorded by invoking `rcu_bh_qsctr_inc()`. These two functions record their state in the current CPU's `rcu_data` structure.

These functions are invoked from the scheduler, from `__do_softirq()`, and from `rcu_check_callbacks()`. This latter function is invoked from the scheduling-clock interrupt, and analyzes state to determine whether this interrupt occurred within a quiescent state, invoking `rcu_qsctr_inc()` and/or `rcu_bh_qsctr_inc()`, as appropriate. It also raises `RCU_SOFTIRQ`, which results in `rcu_process_callbacks()` being invoked on the current CPU at some later time from softirq context.

Announce a Quiescent State to RCU

The afore-mentioned `rcu_process_callbacks()` function has several duties:

1. Determining when to take measures to end an over-long grace period (via `force_quiescent_state()`).
2. Taking appropriate action when some other CPU detected the end of a grace period (via `rcu_process_gp_end()`). “Appropriate action“ includes advancing this CPU's callbacks and recording the new grace period. This same function updates state in response to some other CPU starting a new grace period.
3. Reporting the current CPU's quiescent states to the core RCU mechanism (via `rcu_check_quiescent_state()`, which in turn invokes `cpu_quiet()`). This of course might mark the end of the current grace period.
4. Starting a new grace period if there is no grace period in progress and this CPU has RCU callbacks still waiting for a grace period (via `cpu_needs_another_gp()` and `rcu_start_gp()`).
5. Invoking any of this CPU's callbacks whose grace period has ended (via `rcu_do_batch()`).

These interactions are carefully orchestrated in order to avoid buggy behavior such as reporting a quiescent state from the previous grace period against the current grace period.

Enter and Leave Dynticks Idle Mode

The scheduler invokes `rcu_enter_nohz()` to enter dynticks-idle mode, and invokes `rcu_exit_nohz()` to exit it. The `rcu_enter_nohz()` function increments a per-CPU `dynticks_nesting` variable and also a per-CPU `dynticks` counter, the latter of which which must then have an even-numbered value. The `rcu_exit_nohz()` function decrements this same per-CPU `dynticks_nesting` variable, and again increments the per-CPU `dynticks` counter, the latter of which must then have an odd-numbered value.

The `dynticks` counter can be sampled by other CPUs. If the value is even, the first CPU is in an extended quiescent state. Similarly, if the counter value changes during a given grace period, the first CPU must have been in an extended quiescent state at some point during the grace period. However, there is another `dynticks_nmi` per-CPU variable that must also be sampled, as will be discussed below.

Interrupt from Dynticks Idle Mode

Interrupts from dynticks idle mode are handled by `rcu_irq_enter()` and `rcu_irq_exit()`. The `rcu_irq_enter()` function increments the per-CPU `dynticks_nesting` variable, and, if the prior value was zero, also increments the `dynticks` per-CPU variable (which must then have an odd-numbered value).

The `rcu_irq_exit()` function decrements the per-CPU `dynticks_nesting` variable, and, if the new value is zero, also increments the `dynticks` per-CPU variable (which must then have an even-numbered value).

Note that entering an irq handler exits dynticks idle mode and vice versa. This enter/exit anti-correspondence can cause much confusion. You have been warned.

NMI from Dynticks Idle Mode

NMIs from dynticks idle mode are handled by `rcu_nmi_enter()` and `rcu_nmi_exit()`. These functions both increment the `dynticks_nmi` counter, but only if the aforementioned `dynticks` counter is even. In other words, NMI's refrain from manipulating the `dynticks_nmi` counter if the NMI occurred in non-dynticks-idle mode or within an interrupt handler.

The only difference between these two functions is the error checks, as `rcu_nmi_enter()` must leave the `dynticks_nmi` counter with an odd value, and `rcu_nmi_exit()` must leave this counter with an even value.

Note That a CPU is in Dynticks Idle Mode

The `force_quiescent_state()` function implements a two-phase state machine. In the first phase (`RCU_SAVE_DYNTICK`), the `dyntick_save_progress_counter()` function scans the CPUs that have not yet reported a quiescent state, recording their per-CPU `dynticks` and `dynticks_nmi` counters. If these counters both have even-numbered values, then the corresponding CPU is in dynticks-idle state, which is therefore noted as an extended quiescent state (reported via `cpu_quiet_msk()`). In the second phase (`RCU_FORCE_QS`), the `rcu_implicit_dynticks_qs()` function again scans the CPUs that have not yet reported a quiescent state (either explicitly or implicitly during the `RCU_SAVE_DYNTICK` phase), again checking the per-CPU `dynticks` and `dynticks_nmi` counters. If each of these has either changed in value or is now even, then the corresponding CPU has either passed through or is now in dynticks idle, which as before is noted as an extended quiescent state.

If `rcu_implicit_dynticks_qs()` finds that a given CPU has neither been in dynticks idle mode nor reported a quiescent state, it invokes `rcu_implicit_offline_qs()`, which checks to see if that CPU is offline, which is also reported as an extended quiescent state. If the CPU is online, then `rcu_implicit_offline_qs()` sends it a reschedule IPI in an attempt to remind it of its duty to report a quiescent state to RCU.

Note that `force_quiescent_state()` does not directly invoke either `dyntick_save_progress_counter()` or `rcu_implicit_dynticks_qs()`, instead passing these functions to an intervening `rcu_process_dyntick()` function that abstracts out the common code involved in scanning the CPUs and reporting extended quiescent states.

Quick Quiz 8: And what happens if one CPU comes out of dyntick-idle mode and then passed through a quiescent state just as another CPU notices that the first CPU was in dyntick-idle mode? Couldn't they both attempt to report a quiescent state at the same time, resulting in confusion?

Quick Quiz 9: But what if *all* the CPUs end up in dyntick-idle mode? Wouldn't that prevent the current RCU grace period from ever ending?

Quick Quiz 10: Given that `force_quiescent_state()` is a two-phase state machine, don't we have double the scheduling latency due to scanning all the CPUs?

Offline a CPU

CPU-offline events cause `rcu_cpu_notify()` to invoke `rcu_offline_cpu()`, which in turn invokes `__rcu_offline_cpu()` on both the rcu and the rcu_bh instances of the data structures. This function clears the outgoing CPU's bits so that future grace periods will not expect this CPU to announce quiescent states, and further invokes `cpu_quiet()` in order to announce the offline-induced extended quiescent state. This work is performed with the global `->onofflock` held in order to prevent interference with concurrent grace-period initialization.

Quick Quiz 11: But the other reason to hold `->onofflock` is to prevent multiple concurrent online/offline operations, right?

Online a CPU

CPU-online events cause `rcu_cpu_notify()` to invoke `rcu_online_cpu()`, which initializes the incoming CPU's dynticks state, and then invokes `rcu_init_percpu_data()` to initialize the incoming CPU's `rcu_data` structure, and also to set this CPU's bits (again protected by the global `->onofflock`) so that future grace periods will wait for a quiescent state from this CPU. Finally, `rcu_online_cpu()` sets up the RCU softirq vector for this CPU.

Quick Quiz 12: Given all these acquisitions of the global `->onofflock`, won't there be horrible lock contention when running with thousands of CPUs?

Detect a Too-Long Grace Period

When the CONFIG_RCU_CPU_STALL_DETECTOR kernel parameter is specified, the record_gp_stall_check_time() function records the time and also a timestamp set three seconds into the future. If the current grace period still has not ended by that time, the check_cpu_stall() function will check for the culprit, invoking print_cpu_stall() if the current CPU is the holdout, or print_other_cpu_stall() if it is some other CPU. A two-jiffies offset helps ensure that CPUs report on themselves when possible, taking advantage of the fact that a CPU can normally do a better job of tracing its own stack than it can tracing some other CPU's stack.

Testing

RCU is fundamental synchronization code, so any failure of RCU results in random, difficult-to-debug memory corruption. It is therefore extremely important that RCU be *highly* reliable. Some of this reliability stems from careful design, but at the end of the day we must also rely on heavy stress testing, otherwise known as torture.

Fortunately, although there has been some debate as to exactly what populations are covered by the provisions of the Geneva Convention, it is still the case that it does not apply to software. Therefore, it is still legal to torture your software. In fact, it is strongly encouraged, because if you don't torture your software, it will end up torturing *you* by crashing at the most inconvenient times imaginable.

Therefore, we torture RCU quite vigorously using the rcutorture module.

However, it is not sufficient to torture the common-case uses of RCU. It is also necessary to torture it in unusual situations, for example, when concurrently onlining and offlining CPUs and when CPUs are concurrently entering and exiting dynticks idle mode. I use a script to online and offline CPUs, and use the test_no_idle_hz module parameter to rcutorture to stress-test dynticks idle mode. Just to be fully paranoid, I sometimes run a kernbench workload in parallel as well. Ten hours of this sort of torture on a 128-way machine seems sufficient to shake out most bugs.

Even this is not the complete story. As Alexey Dobriyan and Nick Piggin demonstrated in early 2008, it is also necessary to torture RCU with all relevant combinations of kernel parameters. The relevant kernel parameters may be identified using yet another script, and are as follows:

1. CONFIG_CLASSIC_RCU: Classic RCU.
2. CONFIG_PREEMPT_RCU: Preemptable (real-time) RCU.
3. CONFIG_TREE_RCU: Classic RCU for huge SMP systems.
4. CONFIG_RCU_FANOUT: Number of children for each rcu_node.
5. CONFIG_RCU_FANOUT_EXACT: Balance the rcu_node tree.
6. CONFIG_HOTPLUG_CPU: Allow CPUs to be offlined and onlined.
7. CONFIG_NO_HZ: Enable dyntick-idle mode.
8. CONFIG_SMP: Enable multi-CPU operation.
9. CONFIG_RCU_CPU_STALL_DETECTOR: Enable RCU to detect when CPUs go on extended quiescent-state vacations.
10. CONFIG_RCU_TRACE: Generate RCU trace files in debugfs.

We ignore the CONFIG_DEBUG_LOCK_ALLOC configuration variable under the perhaps-naive assumption that hierarchical RCU could not have broken lockdep. There are still 10 configuration variables, which would result in 1,024 combinations if they were independent boolean variables. Fortunately the first three are mutually exclusive, which reduces the number of combinations down to 384, but CONFIG_RCU_FANOUT can take on values from 2 to 64, increasing the number of combinations to 12,096. This is an infeasible number of combinations.

One key observation is that only CONFIG_NO_HZ and CONFIG_PREEMPT can be expected to have changed behavior if either CONFIG_CLASSIC_RCU or CONFIG_PREEMPT_RCU are in effect, as only these portions of the two pre-existing RCU implementations were changed during this effort. This cuts out almost two thirds of the possible combinations.

Furthermore, not all of the possible values of CONFIG_RCU_FANOUT produce significantly different results, in fact only a few cases really need to be tested separately:

1. Single-node “tree”.
2. Two-level balanced tree.
3. Three-level balanced tree.
4. Autobalanced tree, where CONFIG_RCU_FANOUT specifies an unbalanced tree, but such that it is auto-balanced in absence of CONFIG_RCU_FANOUT_EXACT.
5. Unbalanced tree.

Looking further, CONFIG_HOTPLUG_CPU makes sense only given CONFIG_SMP, and CONFIG_RCU_CPU_STALL_DETECTOR is independent, and really only needs to be tested once (though someone even more paranoid than am I might decide to test it both with and without CONFIG_SMP). Similarly, CONFIG_RCU_TRACE need only be tested once, but the truly paranoid (such as myself) will choose to run it both with and without CONFIG_NO_HZ. This allows us to obtain excellent coverage of RCU with only 15 test cases. All test cases specify the following configuration parameters in order to run rcutorture and so that CONFIG_HOTPLUG_CPU=n actually takes effect:

```
CONFIG_RCU_TORTURE_TEST=m
CONFIG_MODULE_UNLOAD=y
CONFIG_SUSPEND=n
CONFIG_HIBERNATION=n
```

The 15 test cases are as follows:

1. Force single-node “tree” for small systems:

```
CONFIG_NR_CPUS=8
CONFIG_RCU_FANOUT=8
CONFIG_RCU_FANOUT_EXACT=n
CONFIG_RCU_TRACE=y
```

2. Force two-level tree for large systems:

```
CONFIG_NR_CPUS=8
CONFIG_RCU_FANOUT=4
CONFIG_RCU_FANOUT_EXACT=n
CONFIG_RCU_TRACE=n
```

3. Force three-level tree for huge systems:

```
CONFIG_NR_CPUS=8
CONFIG_RCU_FANOUT=2
CONFIG_RCU_FANOUT_EXACT=n
CONFIG_RCU_TRACE=y
```

4. Test autobalancing to a balanced tree:

```
CONFIG_NR_CPUS=8
CONFIG_RCU_FANOUT=6
```

```
CONFIG_RCU_FANOUT_EXACT=n
CONFIG_RCU_TRACE=y
```

5. Test unbalanced tree:

```
CONFIG_NR_CPUS=8
CONFIG_RCU_FANOUT=6
CONFIG_RCU_FANOUT_EXACT=y
CONFIG_RCU_CPU_STALL_DETECTOR=y
CONFIG_RCU_TRACE=y
```

6. Disable CPU-stall detection:

```
CONFIG_SMP=y
CONFIG_NO_HZ=y
CONFIG_RCU_CPU_STALL_DETECTOR=n
CONFIG_HOTPLUG_CPU=y
CONFIG_RCU_TRACE=y
```

7. Disable CPU-stall detection and dyntick idle mode:

```
CONFIG_SMP=y
CONFIG_NO_HZ=n
CONFIG_RCU_CPU_STALL_DETECTOR=n
CONFIG_HOTPLUG_CPU=y
CONFIG_RCU_TRACE=y
```

8. Disable CPU-stall detection and CPU hotplug:

```
CONFIG_SMP=y
CONFIG_NO_HZ=y
CONFIG_RCU_CPU_STALL_DETECTOR=n
CONFIG_HOTPLUG_CPU=n
CONFIG_RCU_TRACE=y
```

9. Disable CPU-stall detection, dyntick idle mode, and CPU hotplug:

```
CONFIG_SMP=y
CONFIG_NO_HZ=n
CONFIG_RCU_CPU_STALL_DETECTOR=n
CONFIG_HOTPLUG_CPU=n
CONFIG_RCU_TRACE=y
```

10. Disable SMP, CPU-stall detection, dyntick idle mode, and CPU hotplug:

```
CONFIG_SMP=n
CONFIG_NO_HZ=n
```

```
CONFIG_RCU_CPU_STALL_DETECTOR=n
CONFIG_HOTPLUG_CPU=n
CONFIG_RCU_TRACE=y
```

This combination located a number of compiler warnings.

11. Disable SMP and CPU hotplug:

```
CONFIG_SMP=n
CONFIG_NO_HZ=n
CONFIG_RCU_CPU_STALL_DETECTOR=n
CONFIG_HOTPLUG_CPU=n
CONFIG_RCU_TRACE=y
```

12. Test Classic RCU with dynticks idle but without preemption:

```
CONFIG_NO_HZ=y
CONFIG_PREEMPT=n
CONFIG_RCU_TRACE=y
```

13. Test Classic RCU with preemption but without dynticks idle:

```
CONFIG_NO_HZ=n
CONFIG_PREEMPT=y
CONFIG_RCU_TRACE=y
```

14. Test Preemptable RCU with dynticks idle:

```
CONFIG_NO_HZ=y
CONFIG_PREEMPT=y
CONFIG_RCU_TRACE=y
```

15. Test Preemptable RCU without dynticks idle:

```
CONFIG_NO_HZ=n
CONFIG_PREEMPT=y
CONFIG_RCU_TRACE=y
```

For a large change that affects RCU core code, one should run rcutorture for each of the above combinations, and concurrently with CPU offlining and onlining for cases with `CONFIG_HOTPLUG_CPU`. For small changes, it may suffice to run kernbench in each case. Of course, if the change is confined to a particular subset of the configuration parameters, it may be possible to reduce the number of test cases.

Torturing software: the Geneva Convention does not (yet) prohibit it, and I strongly recommend it!!!

Conclusion

This hierarchical implementation of RCU reduces lock contention, avoids unnecessarily awakening dyntick-idle sleeping CPUs, while helping to debug Linux's hotplug-CPU code paths. This implementation is designed to handle single systems with thousands of CPUs, and on 64-bit systems has an architectural limitation of a quarter million CPUs, a limit I expect to be sufficient for at least the next few years.

This RCU implementation of course has some limitations:

1. The `force_quiescent_state()` can scan the full set of CPUs with irqs disabled. This would be fatal in a real-time implementation of RCU, so if hierarchy ever needs to be introduced to preemptable RCU, some other approach will be required. It is possible that it will be problematic on 4,096-CPU systems, but actual testing on such systems is required to prove this one way or the other.

   On busy systems, the `force_quiescent_state()` scan would not be expected to happen, as CPUs should pass through quiescent states within three jiffies of the start of a quiescent state. On semi-busy systems, only the CPUs in dynticks-idle mode throughout would need to be scanned. In some cases, for example when a dynticks-idle CPU is handling an interrupt during a scan, subsequent scans are required. However, each such scan is performed separately, so scheduling latency is degraded by the overhead of only one such scan.

   If this scan proves problematic, one straightforward solution would be to do the scan incrementally. This would increase code complexity slightly and would also increase the time required to end a grace period, but would nonetheless be a likely solution.

2. The `rcu_node` hierarchy is created at compile time, and is therefore sized for the worst-case `NR_CPUS` number of CPUs. However, even for 4,096 CPUs, the `rcu_node` hierarchy consumes only 65 cache lines on a 64-bit machine (and just you try accommodating 4,096 CPUs on a 32-bit machine!). Of course, a kernel built with `NR_CPUS=4096` running on a 16-CPU machine would use a two-level tree when a single-node tree would work just fine. Although this configuration would incur added locking overhead, this does not affect hot-path read-side code, so should not be a problem in practice.

3. This patch does increase kernel text and data somewhat: the old Classic RCU implementation consumes 1,757 bytes of kernel text and 456 bytes of kernel data for a total of 2,213 bytes, while the new hierarchical RCU implementation consumes 4,006 bytes of kernel text and 624 bytes of kernel data for a total of 4,630 bytes on a `NR_CPUS=4` system. This is a non-problem even for most embedded systems, which often come with hundreds of megabytes of main memory. However, if this is a problem for tiny embedded systems, it may be necessary to provide both “scale up” and “scale down” implementations of RCU.

This hierarchical RCU implementation should nevertheless be a vast improvement over Classic RCU for machines with hundreds of CPUs. After all, Classic RCU was designed for systems with only 16-32 CPUs.

At some point, it may be necessary to also apply hierarchy to the preemptable RCU implementation. This will be challenging due to the modular arithmetic used on the per-CPU counter pairs, but should be doable.

Acknowledgements

I am indebted to Manfred Spraul for ideas, review comments, bugs spotted, as well as some good healthy competition, to Josh Triplett, Ingo Molnar, Peter Zijlstra, Mathieu Desnoyers, Lai Jiangshan, Andi Kleen, Andy Whitcroft, Gautham Shenoy, and Andrew Morton for review comments, and to Thomas Gleixner for much help with timer issues. I am thankful to Jon M. Tollefson, Tim Pepper, Andrew Theurer, Jose R. Santos, Andy Whitcroft, Darrick Wong, Nishanth Aravamudan, Anton Blanchard, and Nathan Lynch for keeping machines alive despite my (ab)use for this project. We all owe thanks to Peter Zijlstra, Gautham Shenoy, Lai Jiangshan, and Manfred Spraul for helping (in some cases unwittingly) render this document at least partially human readable. Finally, I am grateful to Kathy Bennett for her support of this effort.

This work represents the view of the authors and does not necessarily represent the view of IBM.

Linux is a registered trademark of Linus Torvalds.

Other company, product, and service names may be trademarks or service marks of others.

Answers to Quick Quizzes

Quick Quiz 1: Wait a minute! With all those new locks, how do you avoid deadlock?

Answer: Deadlock is avoided by never holding more than one of the `rcu_node` structures' locks at a given time. This algorithm uses two more locks, one to prevent CPU hotplug operations from running concurrently with grace-period advancement (`onofflock`) and another to permit only one CPU at a time from forcing a quiescent state to end quickly (`fqslock`). These are subject to a locking hierarchy, so that `fqslock` must be acquired before `onofflock`, which in turn must be acquired before any of the `rcu_node` structures' locks.

Also, as a practical matter, refusing to ever hold more than one of the `rcu_node` locks means that it is unnecessary to track which ones are held. Such tracking would be painful as well as unnecessary.

Back to Quick Quiz 1.

Quick Quiz 2: Why stop at a 64-times reduction? Why not go for a few orders of magnitude instead?

Answer: RCU works with no problems on systems with a few hundred CPUs, so allowing 64 CPUs to contend on a single lock leaves plenty of headroom. Keep in mind that these locks are acquired quite rarely, as each CPU will check in about one time per grace period, and grace periods extend for milliseconds.

Back to Quick Quiz 2.

Quick Quiz 3: But I don't care about McKenney's lame excuses in the answer to Quick Quiz 2!!! I want to get the number of CPUs contending on a single lock down to something reasonable, like sixteen or so!!!

Answer: OK, have it your way, then!!! Set `CONFIG_RCU_FANOUT=16` and (for `NR_CPUS=4096`) you will get a three-level hierarchy with with 256 `rcu_node` structures at the lowest level, 16 `rcu_node` structures as intermediate nodes, and a single root-level `rcu_node`. The penalty you will pay is that more `rcu_node` structures will need to be scanned when checking to see which CPUs need help completing their quiescent states (256 instead of only 64).

Back to Quick Quiz 3.

Quick Quiz 4: OK, so what is the story with the colors?

Answer: Data structures analogous to `rcu_state` (including `rcu_ctrlblk`) are yellow, those containing the bitmaps used to determine when CPUs have checked in are pink, and the per-CPU `rcu_data` structures are blue. Later on, we will see that data structures used to conserve energy (such as `rcu_dynticks`) will be green.

Back to Quick Quiz 4.

Quick Quiz 5: Given such an egregious bug, why does Linux run at all?

Answer: Because the Linux kernel contains device drivers that are (relatively) well behaved. Few if any of them spin in RCU read-side critical sections for the many milliseconds that would be required to provoke this bug. The bug nevertheless does need to be fixed, and this variant of RCU does fix it.

Back to Quick Quiz 5.

Quick Quiz 6: But what happens if a CPU tries to report going through a quiescent state (by clearing its bit) before the bit-setting CPU has finished?

Answer: There are three cases to consider here:

1. A CPU corresponding to a non-yet-initialized leaf `rcu_node` structure tries to report a quiescent state. This CPU will see its bit already cleared, so will give up on reporting its quiescent state. Some later quiescent state will serve for the new grace period.
2. A CPU corresponding to a leaf `rcu_node` structure that is currently being initialized tries to report a quiescent state. This CPU will see that the `rcu_node` structure's `->lock` is held, so will spin until it is released. But once the lock is released, the `rcu_node` structure will have been initialized, reducing to the following case.
3. A CPU corresponding to a leaf `rcu_node` that has already been initialized tries to report a quiescent state. This CPU will find its bit set, and will therefore clear it. If it is the last CPU for that leaf node, it will move up to the next level of the hierarchy. However, this CPU cannot possibly be the last CPU in the system to report a quiescent state, given that the CPU doing the initialization cannot yet have checked in.

So, in all three cases, the potential race is resolved correctly.

Back to Quick Quiz 6.

Quick Quiz 7: And what happens if *all* CPUs try to report going through a quiescent state before the bit-setting CPU has finished, thus ending the new grace period before it starts?

Answer: The bit-setting CPU cannot pass through a quiescent state during initialization, as it has irqs disabled. Its bits therefore remain non-zero, preventing the grace period from ending until the data structure has been fully initialized.

Back to Quick Quiz 7.

Quick Quiz 8: And what happens if one CPU comes out of dyntick-idle mode and then passed through a quiescent state just as another CPU notices that the first CPU was in dyntick-idle mode? Couldn't they both attempt to report a quiescent state at the same time, resulting in confusion?

Answer: They will both attempt to acquire the lock on the same leaf `rcu_node` structure. The first one to acquire the lock will report the quiescent state and clear the appropriate bit, and the second one to acquire the lock will see that this bit has already been cleared.

Back to Quick Quiz 8.

Quick Quiz 9: But what if *all* the CPUs end up in dyntick-idle mode? Wouldn't that prevent the current RCU grace period from ever ending?

Answer: Indeed it will! However, CPUs that have RCU callbacks are not permitted to enter dyntick-idle mode, so the only way that *all* the CPUs could possibly end up in dyntick-idle mode would be if there were absolutely no RCU callbacks in the system. And if there are no RCU callbacks in the system, then there is no need for the RCU grace period to end. In fact, there is no need for the RCU grace period to even *start*.

RCU will restart if some irq handler does a `call_rcu()`, which will cause an RCU callback to appear on the corresponding CPU, which will force that CPU out of dyntick-idle mode, which will in turn permit the current RCU grace period to come to an end.

Back to Quick Quiz 9.

Quick Quiz 10: Given that `force_quiescent_state()` is a two-phase state machine, don't we have double the scheduling latency due to scanning all the CPUs?

Answer: Ah, but the two phases will not execute back-to-back on the same CPU. Therefore, the scheduling-latency hit of the two-phase algorithm is no different than that of a single-phase algorithm. If the scheduling latency becomes a problem, one approach would be to recode the state machine to scan the CPUs incrementally. But first show me a problem in the real world, *then* I will consider fixing it!

Back to Quick Quiz 10.

Quick Quiz 11: But the other reason to hold `->onofflock` is to prevent multiple concurrent online/offline operations, right?

Answer: Actually, no! The CPU-hotplug code's synchronization design prevents multiple concurrent CPU online/offline operations, so only one CPU online/offline operation can be executing at any given time. Therefore, the only purpose of `->onofflock` is to prevent a CPU online or offline operation from running concurrently with grace-period initialization.

Back to Quick Quiz 11.

Quick Quiz 12: Given all these acquisitions of the global `->onofflock`, won't there be horrible lock contention when running with thousands of CPUs?

Answer: Actually, there can be only three acquisitions of this lock per grace period, and each grace period lasts many milliseconds. One of the acquisitions is by the CPU initializing for the current grace period, and the other two onlining and offlining some CPU. These latter two cannot run concurrently due to the CPU-hotplug locking, so at most two CPUs can be contending for this lock at any given time.

Lock contention on `->onofflock` should therefore be no problem, even on systems with thousands of CPUs.

Back to Quick Quiz 12.

Index entries for this article

Kernel	Read-copy-update
GuestArticles	McKenney, Paul E.

Hierarchical RCU

Posted Nov 6, 2008 3:15 UTC (Thu) by **kev009** (guest, #43906) [Link]

This was an incredible write up, rich in technical data yet still compelling and informative to read. Many thanks!

PS: you should consider authoring a book on kernel development.. :)

Reply to this comment

Hierarchical RCU

Posted Nov 6, 2008 23:48 UTC (Thu) by **PaulMcKenney** (subscriber, #9624) [Link]

Glad you liked it, and thank you for the encouragement. But... Be careful what you wish for. :-)

Reply to this comment

Hierarchical RCU

Posted Nov 6, 2008 8:54 UTC (Thu) by **kleptog** (subscriber, #1183) [Link]

Agreed. This was a very readable explanation of RCU. Keep these sorts of articles coming.

Reply to this comment

Hierarchical RCU

Posted Nov 6, 2008 11:19 UTC (Thu) by **melo@simplicidade.org** (subscriber, #4380) [Link]

This type or articles are a reason to keep paying for LWN.

Thanks,

Reply to this comment

Hierarchical RCU - state schematic

Posted Nov 6, 2008 18:49 UTC (Thu) by **ds2horner** (subscriber, #13438) [Link]

In state machine schematic : "Mark CPU as being in Extended QS" from "CPU Offline" immediately returns to checking if "all cpus passed though QS".

However the "Mark CPU as being in Extended QS" for a CPU in dyntick-idle still gets rescheduled.

Is a dyntick-idle CPU actually a "hold out"?
I thought it was in Extended QS and no further checking/scheduling was required.

So is the arrow to "Send Resched IPI ..." incorrect, or should this be another Quick (or perhaps no so) Quiz?

Reply to this comment

Hierarchical RCU - state schematic

Posted Nov 6, 2008 19:49 UTC (Thu) by **PaulMcKenney** (subscriber, #9624) [Link]

The trick with that part of the diagram is that the code is actually dealing with groups of CPUs. So a given group of CPUs might have some that are in dyntick-idle state and others that have somehow avoided passing through a quiescent state, despite the fact that they are online and running. We send a reschedule IPI only to these latter CPUs, not to the dyntick-idle CPUs.

Now that you mention it, this might indeed be a good quick-quiz candidate. :-)

Reply to this comment

Hierarchical RCU - state schematic

Posted Nov 7, 2008 1:55 UTC (Fri) by **ds2horner** (subscriber, #13438) [Link]

I think the leveling confusion in the diagram starts with the exit from the "wait for QS" state.

"CPU passes through QS" and "CPU offline" are considered singleton events, but the "GP too long" is considered a bulk event with all blocking CPUs being addressed.

This, and the explanation of the initiation of the check for the dynaticks state, clarified for me (indeed corrected a misconception I had) that the dyntick counters are not captured at the beginning of each GP, but only after a "time out" (a reasonable optimization for a low occurrence event).

Speaking of the time out -
in "Detect a Too-Long Grace Period" the "record_gp_stall_check_time() function records the time and also a timestamp set three seconds into the future." timeframe seemed excessive. I believe jiffies was meant (not seconds) which would be consistent with the later reference "A two-jiffies offset helps ensure that CPUs report on themselves when possible".

Each article clarifies details I missed before.

Reply to this comment

Hierarchical RCU - state schematic

Posted Nov 7, 2008 5:52 UTC (Fri) by **PaulMcKenney** (subscriber, #9624) [Link]

Glad it helped!

There are indeed two levels of tardiness. Reschedule IPIs are sent to hold-out CPUs after three jiffies, which, in absence of kernel bugs, should end the grace period. These reschedule IPIs are considered "normal" rather than "errors".

There is a separate "error" path enabled by CONFIG_RCU_CPU_STALL_DETECTOR that does in fact have a three-second timeout (recently upped to 10 seconds based on the fact that three-second stalls appear to be present in boot-up code). There is also a three-second-and-two-jiffies timeout as part of this "error" path so that CPUs will normally report on themselves rather than being reported on by others, given that self-reported stack traces are usually more reliable.

Reply to this comment

Hierarchical RCU - unification suggestion

Posted Nov 7, 2008 15:50 UTC (Fri) by **ds2horner** (subscriber, #13438) [Link]

Thank you again.

And now that I hope I believe I understand the process adequately I have a suggestion.

Why not unify the CPU hotplug and "dyntick suspend" status tracking by using the same counter as dyntick for both?

If the CPU offline / online advanced the (now renamed rcu_cpu_awake (with the LBS matching CPU state)) counter, the even / odd state will satisfy both scenarios.

And I am not advocating this solely for the purpose of reducing the state variables and reducing code (if not complexity):
Conceptually, an offline CPU (a powered down engine which cannot receive interrupts) is equivalent to a dyntick CPU (powered down state) that receives no interrupts in the grace period.

I realize there are, of course, other reasons to track online/offline CPUs, that the offline check is immediately available.

However, the unification would allow a simpler "no off line" option to the this intended "dyntick replacement" to CLASSIC RCU.
And a simpler "no NO_HZ" option that would be useful for virtual CPUs.

I know - "show me the code!".

OK reading your posts
"v6 scalable classic RCU implementation" "Tue, 23 Sep 2008 16:53:40 -0700"
indicate the magnitude of code that is present to handle a CPU offline activity.
Concerns like moving outstanding rcu work from the "forced" offline CPU to the current CPU, make the offlining distinct.

However, it appears to be performed (at least in this version of the code) just before sending the tardy CPUs the IPI reschedule. Which, back to the schematic, makes it part of the "GP too long" path?

And reading your further post "rcu-state" "Mon, 27 Oct 2008 20:52:01 +0100"
It looks like this version is using a unifying indicator "rcu_cpumode" - still with 2 distinct substates, that are separate from RCU_CPUMODE_PERIODIC which requires the notifications each Grace Period.

How does one keep up?

Reply to this comment

Hierarchical RCU - unification suggestion

Posted Nov 8, 2008 1:03 UTC (Sat) by **PaulMcKenney** (subscriber, #9624) [Link]

Looks like I should have had yet another Quick Quiz on unifying dyntick-idle and offline detection.

First, it might well turn out to be the right thing to do. However, the challenges include the following:

1. As you say,
2. As you say, although CPUs are not allowed to go into dyntick-idle state while they have RCU callbacks pending, CPUs -are- allowed to go offline in this state. This means that the code to move their callbacks is still requires. We cannot simply let them silently go offline.
3. Present-day systems often run with NR_CPUS much larger than the actual number of CPUs, so the unified approach could waste time scanning CPUs that never will exist. (There are workarounds for this.)
4. CPUs get onlined one at a time, so RCU needs to handle onlining.
5. A busy system with offlined CPUs would always take three ticks to figure out that the offlined CPUs were never going to respond.
6. Switching into and out of dyntick-idle state can happen extremely frequently, so we cannot treat a dyntick-idle CPU as if it was offline due to the high overhead of offlining CPUs.

Under normal circumstances, offline CPUs are not included in the bitmasks indicating which CPUs need to be waited on, so that normally RCU never waits on offline CPUs. However, there are race conditions that can occur in the online and offline processes that can result in RCU believing that a given CPU is online when it is in fact offline. Therefore, if RCU sees that the grace period is extending longer than expected (jiffies, not seconds), it will check to see if some of the CPUs that it is waiting on are offline. This situation corrects itself after a few grace periods: RCU will get back in sync with which CPUs really are offline. So the offline-CPU checks invoked from force_quiescent_state() are only there to handle rare race conditions. Again, under normal circumstances, RCU never waits on offline CPUs.

At this point, when the code is still just a patch, and therefore subject to change, the only way I can see to keep up is to ask questions. Which you are doing. :-)

Reply to this comment