Linux Rcu Documentation

The kernel development community

Jun 10, 2024

CONTENTS

CHAPTER

ONE

REVIEW CHECKLIST FOR RCU PATCHES

This document contains a checklist for producing and reviewing patches that make use of RCU. Violating any of the rules listed below will result in the same sorts of problems that leaving out a locking primitive would cause. This list is based on experiences reviewing such patches over a rather long period of time, but improvements are always welcome!

0. Is RCU being applied to a read-mostly situation? If the data structure is updated more than about 10% of the time, then you should strongly consider some other approach, unless detailed performance measurements show that RCU is nonetheless the right tool for the job. Yes, RCU does reduce read-side overhead by increasing write-side overhead, which is exactly why normal uses of RCU will do much more reading than updating.

   Another exception is where performance is not an issue, and RCU provides a simpler implementation. An example of this situation is the dynamic NMI code in the Linux 2.6 kernel, at least on architectures where NMIs are rare.

   Yet another exception is where the low real-time latency of RCU's read-side primitives is critically important.

   One final exception is where RCU readers are used to prevent the ABA problem (https://en.wikipedia.org/wiki/ABA_problem) for lockless updates. This does result in the mildly counter-intuitive situation where rcu_read_lock() and rcu_read_unlock() are used to protect updates, however, this approach can provide the same simplifications to certain types of lockless algorithms that garbage collectors do.

1. Does the update code have proper mutual exclusion?

   RCU does allow *readers* to run (almost) naked, but *writers* must still use some sort of mutual exclusion, such as:

   a. locking,

   b. atomic operations, or

   c. restricting updates to a single task.

   If you choose #b, be prepared to describe how you have handled memory barriers on weakly ordered machines (pretty much all of them -- even x86 allows later loads to be reordered to precede earlier stores), and be prepared to explain why this added complexity is worthwhile. If you choose #c, be prepared to explain how this single task does not become a major bottleneck on large systems (for example, if the task is updating information relating to itself that other tasks can read, there by definition can be no bottleneck). Note that the definition of "large" has changed significantly: Eight CPUs was "large" in the year 2000, but a hundred CPUs was unremarkable in 2017.

2. Do the RCU read-side critical sections make proper use of rcu_read_lock() and friends? These primitives are needed to prevent grace periods from ending prematurely, which could result in data being unceremoniously freed out from under your read-side code, which can greatly increase the actuarial risk of your kernel.

   As a rough rule of thumb, any dereference of an RCU-protected pointer must be covered by rcu_read_lock(), rcu_read_lock_bh(), rcu_read_lock_sched(), or by the appropriate update-side lock. Explicit disabling of preemption (preempt_disable(), for example) can serve as rcu_read_lock_sched(), but is less readable and prevents lockdep from detecting locking issues.

   Please note that you *cannot* rely on code known to be built only in non-preemptible kernels. Such code can and will break, especially in kernels built with CONFIG_PREEMPT_COUNT=y.

   Letting RCU-protected pointers "leak" out of an RCU read-side critical section is every bit as bad as letting them leak out from under a lock. Unless, of course, you have arranged some other means of protection, such as a lock or a reference count *before* letting them out of the RCU read-side critical section.

3. Does the update code tolerate concurrent accesses?

   The whole point of RCU is to permit readers to run without any locks or atomic operations. This means that readers will be running while updates are in progress. There are a number of ways to handle this concurrency, depending on the situation:

   a. Use the RCU variants of the list and hlist update primitives to add, remove, and replace elements on an RCU-protected list. Alternatively, use the other RCU-protected data structures that have been added to the Linux kernel.

      This is almost always the best approach.

   b. Proceed as in (a) above, but also maintain per-element locks (that are acquired by both readers and writers) that guard per-element state. Fields that the readers refrain from accessing can be guarded by some other lock acquired only by updaters, if desired.

      This also works quite well.

   c. Make updates appear atomic to readers. For example, pointer updates to properly aligned fields will appear atomic, as will individual atomic primitives. Sequences of operations performed under a lock will *not* appear to be atomic to RCU readers, nor will sequences of multiple atomic primitives. One alternative is to move multiple individual fields to a separate structure, thus solving the multiple-field problem by imposing an additional level of indirection.

      This can work, but is starting to get a bit tricky.

   d. Carefully order the updates and the reads so that readers see valid data at all phases of the update. This is often more difficult than it sounds, especially given modern CPUs' tendency to reorder memory references. One must usually liberally sprinkle memory-ordering operations through the code, making it difficult to understand and to test. Where it works, it is better to use things like smp_store_release() and smp_load_acquire(), but in some cases the smp_mb() full memory barrier is required.

      As noted earlier, it is usually better to group the changing data into a separate structure, so that the change may be made to appear atomic by updating a pointer to reference a new structure containing updated values.

4. Weakly ordered CPUs pose special challenges. Almost all CPUs are weakly ordered -- even x86 CPUs allow later loads to be reordered to precede earlier stores. RCU code must take all of the following measures to prevent memory-corruption problems:

   a. Readers must maintain proper ordering of their memory accesses. The rcu_dereference() primitive ensures that the CPU picks up the pointer before it picks up the data that the pointer points to. This really is necessary on Alpha CPUs.

      The rcu_dereference() primitive is also an excellent documentation aid, letting the person reading the code know exactly which pointers are protected by RCU. Please note that compilers can also reorder code, and they are becoming increasingly aggressive about doing just that. The rcu_dereference() primitive therefore also prevents destructive compiler optimizations. However, with a bit of devious creativity, it is possible to mishandle the return value from rcu_dereference(). Please see *PROPER CARE AND FEEDING OF RETURN VALUES FROM rcu_dereference()* for more information.

      The rcu_dereference() primitive is used by the various "_rcu()" list-traversal primitives, such as the list_for_each_entry_rcu(). Note that it is perfectly legal (if redundant) for update-side code to use rcu_dereference() and the "_rcu()" list-traversal primitives. This is particularly useful in code that is common to readers and updaters. However, lockdep will complain if you access rcu_dereference() outside of an RCU read-side critical section. See *RCU and lockdep checking* to learn what to do about this.

      Of course, neither rcu_dereference() nor the "_rcu()" list-traversal primitives can substitute for a good concurrency design coordinating among multiple updaters.

   b. If the list macros are being used, the list_add_tail_rcu() and list_add_rcu() primitives must be used in order to prevent weakly ordered machines from misordering structure initialization and pointer planting. Similarly, if the hlist macros are being used, the hlist_add_head_rcu() primitive is required.

   c. If the list macros are being used, the list_del_rcu() primitive must be used to keep list_del()'s pointer poisoning from inflicting toxic effects on concurrent readers. Similarly, if the hlist macros are being used, the hlist_del_rcu() primitive is required.

      The list_replace_rcu() and hlist_replace_rcu() primitives may be used to replace an old structure with a new one in their respective types of RCU-protected lists.

   d. Rules similar to (4b) and (4c) apply to the "hlist_nulls" type of RCU-protected linked lists.

   e. Updates must ensure that initialization of a given structure happens before pointers to that structure are publicized. Use the rcu_assign_pointer() primitive when publicizing a pointer to a structure that can be traversed by an RCU read-side critical section.

5. If any of call_rcu(), call_srcu(), call_rcu_tasks(), call_rcu_tasks_rude(), or call_rcu_tasks_trace() is used, the callback function may be invoked from softirq context, and in any case with bottom halves disabled. In particular, this callback function cannot block. If you need the callback to block, run that code in a workqueue handler scheduled from the callback. The queue_rcu_work() function does this for you in the case of call_rcu().

6. Since synchronize_rcu() can block, it cannot be called from any sort of irq context. The same rule applies for synchronize_srcu(), synchronize_rcu_expedited(), synchronize_srcu_expedited(), synchronize_rcu_tasks(), synchronize_rcu_tasks_rude(), and synchronize_rcu_tasks_trace().

The expedited forms of these primitives have the same semantics as the non-expedited forms, but expediting is more CPU intensive. Use of the expedited primitives should be restricted to rare configuration-change operations that would not normally be undertaken while a real-time workload is running. Note that IPI-sensitive real-time workloads can use the rcupdate.rcu_normal kernel boot parameter to completely disable expedited grace periods, though this might have performance implications.

In particular, if you find yourself invoking one of the expedited primitives repeatedly in a loop, please do everyone a favor: Restructure your code so that it batches the updates, allowing a single non-expedited primitive to cover the entire batch. This will very likely be faster than the loop containing the expedited primitive, and will be much much easier on the rest of the system, especially to real-time workloads running on the rest of the system. Alternatively, instead use asynchronous primitives such as call_rcu().

7. As of v4.20, a given kernel implements only one RCU flavor, which is RCU-sched for PREEMPTION=n and RCU-preempt for PREEMPTION=y. If the updater uses call_rcu() or synchronize_rcu(), then the corresponding readers may use: (1) rcu_read_lock() and rcu_read_unlock(), (2) any pair of primitives that disables and re-enables softirq, for example, rcu_read_lock_bh() and rcu_read_unlock_bh(), or (3) any pair of primitives that disables and re-enables preemption, for example, rcu_read_lock_sched() and rcu_read_unlock_sched(). If the updater uses synchronize_srcu() or call_srcu(), then the corresponding readers must use srcu_read_lock() and srcu_read_unlock(), and with the same srcu_struct. The rules for the expedited RCU grace-period-wait primitives are the same as for their non-expedited counterparts.

   If the updater uses call_rcu_tasks() or synchronize_rcu_tasks(), then the readers must refrain from executing voluntary context switches, that is, from blocking. If the updater uses call_rcu_tasks_trace() or synchronize_rcu_tasks_trace(), then the corresponding readers must use rcu_read_lock_trace() and rcu_read_unlock_trace(). If an updater uses call_rcu_tasks_rude() or synchronize_rcu_tasks_rude(), then the corresponding readers must use anything that disables preemption, for example, preempt_disable() and preempt_enable().

   Mixing things up will result in confusion and broken kernels, and has even resulted in an exploitable security issue. Therefore, when using non-obvious pairs of primitives, commenting is of course a must. One example of non-obvious pairing is the XDP feature in networking, which calls BPF programs from network-driver NAPI (softirq) context. BPF relies heavily on RCU protection for its data structures, but because the BPF program invocation happens entirely within a single local_bh_disable() section in a NAPI poll cycle, this usage is safe. The reason that this usage is safe is that readers can use anything that disables BH when updaters use call_rcu() or synchronize_rcu().

8. Although synchronize_rcu() is slower than is call_rcu(), it usually results in simpler code. So, unless update performance is critically important, the updaters cannot block, or the latency of synchronize_rcu() is visible from userspace, synchronize_rcu() should be used in preference to call_rcu(). Furthermore, kfree_rcu() and kvfree_rcu() usually result in even simpler code than does synchronize_rcu() without synchronize_rcu()'s multi-millisecond latency. So please take advantage of kfree_rcu()'s and kvfree_rcu()'s "fire and forget" memory-freeing capabilities where it applies.

   An especially important property of the synchronize_rcu() primitive is that it automatically self-limits: if grace periods are delayed for whatever reason, then the synchronize_rcu() primitive will correspondingly delay updates. In contrast, code using call_rcu() should explicitly limit update rate in cases where grace periods are delayed, as failing to do so can result in excessive realtime latencies or even OOM conditions.

Ways of gaining this self-limiting property when using call_rcu(), kfree_rcu(), or kvfree_rcu() include:

a. Keeping a count of the number of data-structure elements used by the RCU-protected data structure, including those waiting for a grace period to elapse. Enforce a limit on this number, stalling updates as needed to allow previously deferred frees to complete. Alternatively, limit only the number awaiting deferred free rather than the total number of elements.

   One way to stall the updates is to acquire the update-side mutex. (Don't try this with a spinlock -- other CPUs spinning on the lock could prevent the grace period from ever ending.) Another way to stall the updates is for the updates to use a wrapper function around the memory allocator, so that this wrapper function simulates OOM when there is too much memory awaiting an RCU grace period. There are of course many other variations on this theme.

b. Limiting update rate. For example, if updates occur only once per hour, then no explicit rate limiting is required, unless your system is already badly broken. Older versions of the dcache subsystem take this approach, guarding updates with a global lock, limiting their rate.

c. Trusted update -- if updates can only be done manually by superuser or some other trusted user, then it might not be necessary to automatically limit them. The theory here is that superuser already has lots of ways to crash the machine.

d. Periodically invoke rcu_barrier(), permitting a limited number of updates per grace period.

The same cautions apply to call_srcu(), call_rcu_tasks(), call_rcu_tasks_rude(), and call_rcu_tasks_trace(). This is why there is an srcu_barrier(), rcu_barrier_tasks(), rcu_barrier_tasks_rude(), and rcu_barrier_tasks_rude(), respectively.

Note that although these primitives do take action to avoid memory exhaustion when any given CPU has too many callbacks, a determined user or administrator can still exhaust memory. This is especially the case if a system with a large number of CPUs has been configured to offload all of its RCU callbacks onto a single CPU, or if the system has relatively little free memory.

9. All RCU list-traversal primitives, which include rcu_dereference(), list_for_each_entry_rcu(), and list_for_each_safe_rcu(), must be either within an RCU read-side critical section or must be protected by appropriate update-side locks. RCU read-side critical sections are delimited by rcu_read_lock() and rcu_read_unlock(), or by similar primitives such as rcu_read_lock_bh() and rcu_read_unlock_bh(), in which case the matching rcu_dereference() primitive must be used in order to keep lockdep happy, in this case, rcu_dereference_bh().

   The reason that it is permissible to use RCU list-traversal primitives when the update-side lock is held is that doing so can be quite helpful in reducing code bloat when common code is shared between readers and updaters. Additional primitives are provided for this case, as discussed in *RCU and lockdep checking*.

   One exception to this rule is when data is only ever added to the linked data structure, and is never removed during any time that readers might be accessing that structure. In such cases, READ_ONCE() may be used in place of rcu_dereference() and the read-side markers (rcu_read_lock() and rcu_read_unlock(), for example) may be omitted.

10. Conversely, if you are in an RCU read-side critical section, and you don't hold the appropriate update-side lock, you *must* use the "_rcu()" variants of the list macros. Failing to do so will break Alpha, cause aggressive compilers to generate bad code, and confuse people trying to understand your code.

11. Any lock acquired by an RCU callback must be acquired elsewhere with softirq disabled, e.g., via spin_lock_bh(). Failing to disable softirq on a given acquisition of that lock will result in deadlock as soon as the RCU softirq handler happens to run your RCU callback while interrupting that acquisition's critical section.

12. RCU callbacks can be and are executed in parallel. In many cases, the callback code simply wrappers around kfree(), so that this is not an issue (or, more accurately, to the extent that it is an issue, the memory-allocator locking handles it). However, if the callbacks do manipulate a shared data structure, they must use whatever locking or other synchronization is required to safely access and/or modify that data structure.

    Do not assume that RCU callbacks will be executed on the same CPU that executed the corresponding call_rcu() or call_srcu(). For example, if a given CPU goes offline while having an RCU callback pending, then that RCU callback will execute on some surviving CPU. (If this was not the case, a self-spawning RCU callback would prevent the victim CPU from ever going offline.) Furthermore, CPUs designated by rcu_nocbs= might well *always* have their RCU callbacks executed on some other CPUs, in fact, for some real-time workloads, this is the whole point of using the rcu_nocbs= kernel boot parameter.

    In addition, do not assume that callbacks queued in a given order will be invoked in that order, even if they all are queued on the same CPU. Furthermore, do not assume that same-CPU callbacks will be invoked serially. For example, in recent kernels, CPUs can be switched between offloaded and de-offloaded callback invocation, and while a given CPU is undergoing such a switch, its callbacks might be concurrently invoked by that CPU's softirq handler and that CPU's rcuo kthread. At such times, that CPU's callbacks might be executed both concurrently and out of order.

13. Unlike most flavors of RCU, it *is* permissible to block in an SRCU read-side critical section (demarked by srcu_read_lock() and srcu_read_unlock()), hence the "SRCU": "sleepable RCU". Please note that if you don't need to sleep in read-side critical sections, you should be using RCU rather than SRCU, because RCU is almost always faster and easier to use than is SRCU.

    Also unlike other forms of RCU, explicit initialization and cleanup is required either at build time via DEFINE_SRCU() or DEFINE_STATIC_SRCU() or at runtime via init_srcu_struct() and cleanup_srcu_struct(). These last two are passed a "struct srcu_struct" that defines the scope of a given SRCU domain. Once initialized, the srcu_struct is passed to srcu_read_lock(), srcu_read_unlock() synchronize_srcu(), synchronize_srcu_expedited(), and call_srcu(). A given synchronize_srcu() waits only for SRCU read-side critical sections governed by srcu_read_lock() and srcu_read_unlock() calls that have been passed the same srcu_struct. This property is what makes sleeping read-side critical sections tolerable -- a given subsystem delays only its own updates, not those of other subsystems using SRCU. Therefore, SRCU is less prone to OOM the system than RCU would be if RCU's read-side critical sections were permitted to sleep.

    The ability to sleep in read-side critical sections does not come for free. First, corresponding srcu_read_lock() and srcu_read_unlock() calls must be passed the same srcu_struct. Second, grace-period-detection overhead is amortized only over those updates sharing a given srcu_struct, rather than being globally amortized as they are for other forms of RCU. Therefore, SRCU should be used in preference to rw_semaphore only in extremely read-intensive situations, or in situations requiring SRCU's read-side deadlock immunity or low read-side realtime latency. You should also consider percpu_rw_semaphore when you need lightweight readers.

SRCU's expedited primitive (synchronize_srcu_expedited()) never sends IPIs to other CPUs, so it is easier on real-time workloads than is synchronize_rcu_expedited().

It is also permissible to sleep in RCU Tasks Trace read-side critical, which are delimited by rcu_read_lock_trace() and rcu_read_unlock_trace(). However, this is a specialized flavor of RCU, and you should not use it without first checking with its current users. In most cases, you should instead use SRCU.

Note that rcu_assign_pointer() relates to SRCU just as it does to other forms of RCU, but instead of rcu_dereference() you should use srcu_dereference() in order to avoid lockdep splats.

14. The whole point of call_rcu(), synchronize_rcu(), and friends is to wait until all pre-existing readers have finished before carrying out some otherwise-destructive operation. It is therefore critically important to *first* remove any path that readers can follow that could be affected by the destructive operation, and *only then* invoke call_rcu(), synchronize_rcu(), or friends.

    Because these primitives only wait for pre-existing readers, it is the caller's responsibility to guarantee that any subsequent readers will execute safely.

15. The various RCU read-side primitives do *not* necessarily contain memory barriers. You should therefore plan for the CPU and the compiler to freely reorder code into and out of RCU read-side critical sections. It is the responsibility of the RCU update-side primitives to deal with this.

    For SRCU readers, you can use smp_mb__after_srcu_read_unlock() immediately after an srcu_read_unlock() to get a full barrier.

16. Use CONFIG_PROVE_LOCKING, CONFIG_DEBUG_OBJECTS_RCU_HEAD, and the __rcu sparse checks to validate your RCU code. These can help find problems as follows:

    CONFIG_PROVE_LOCKING:
    check that accesses to RCU-protected data structures are carried out under the proper RCU read-side critical section, while holding the right combination of locks, or whatever other conditions are appropriate.

    CONFIG_DEBUG_OBJECTS_RCU_HEAD:
    check that you don't pass the same object to call_rcu() (or friends) before an RCU grace period has elapsed since the last time that you passed that same object to call_rcu() (or friends).

    __rcu sparse checks:
    tag the pointer to the RCU-protected data structure with __rcu, and sparse will warn you if you access that pointer without the services of one of the variants of rcu_dereference().

    These debugging aids can help you find problems that are otherwise extremely difficult to spot.

17. If you pass a callback function defined within a module to one of call_rcu(), call_srcu(), call_rcu_tasks(), call_rcu_tasks_rude(), or call_rcu_tasks_trace(), then it is necessary to wait for all pending callbacks to be invoked before unloading that module. Note that it is

absolutely *not* sufficient to wait for a grace period! For example, synchronize_rcu() implementation is *not* guaranteed to wait for callbacks registered on other CPUs via call_rcu(). Or even on the current CPU if that CPU recently went offline and came back online.

You instead need to use one of the barrier functions:

- call_rcu() -> rcu_barrier()
- call_srcu() -> srcu_barrier()
- call_rcu_tasks() -> rcu_barrier_tasks()
- call_rcu_tasks_rude() -> rcu_barrier_tasks_rude()
- call_rcu_tasks_trace() -> rcu_barrier_tasks_trace()

However, these barrier functions are absolutely *not* guaranteed to wait for a grace period. For example, if there are no call_rcu() callbacks queued anywhere in the system, rcu_barrier() can and will return immediately.

So if you need to wait for both a grace period and for all pre-existing callbacks, you will need to invoke both functions, with the pair depending on the flavor of RCU:

- Either synchronize_rcu() or synchronize_rcu_expedited(), together with rcu_barrier()
- Either synchronize_srcu() or synchronize_srcu_expedited(), together with and srcu_barrier()
- synchronize_rcu_tasks() and rcu_barrier_tasks()
- synchronize_tasks_rude() and rcu_barrier_tasks_rude()
- synchronize_tasks_trace() and rcu_barrier_tasks_trace()

If necessary, you can use something like workqueues to execute the requisite pair of functions concurrently.

See *RCU and Unloadable Modules* for more information.

CHAPTER TWO

RCU AND LOCKDEP CHECKING

All flavors of RCU have lockdep checking available, so that lockdep is aware of when each task enters and leaves any flavor of RCU read-side critical section. Each flavor of RCU is tracked separately (but note that this is not the case in 2.6.32 and earlier). This allows lockdep's tracking to include RCU state, which can sometimes help when debugging deadlocks and the like.

In addition, RCU provides the following primitives that check lockdep's state:

```
rcu_read_lock_held() for normal RCU.
rcu_read_lock_bh_held() for RCU-bh.
rcu_read_lock_sched_held() for RCU-sched.
rcu_read_lock_any_held() for any of normal RCU, RCU-bh, and RCU-sched.
srcu_read_lock_held() for SRCU.
rcu_read_lock_trace_held() for RCU Tasks Trace.
```

These functions are conservative, and will therefore return 1 if they aren't certain (for example, if CONFIG_DEBUG_LOCK_ALLOC is not set). This prevents things like WARN_ON(!rcu_read_lock_held()) from giving false positives when lockdep is disabled.

In addition, a separate kernel config parameter CONFIG_PROVE_RCU enables checking of rcu_dereference() primitives:

rcu_dereference(p):
Check for RCU read-side critical section.

rcu_dereference_bh(p):
Check for RCU-bh read-side critical section.

rcu_dereference_sched(p):
Check for RCU-sched read-side critical section.

srcu_dereference(p, sp):
Check for SRCU read-side critical section.

rcu_dereference_check(p, c):
Use explicit check expression "c" along with rcu_read_lock_held(). This is useful in code that is invoked by both RCU readers and updaters.

rcu_dereference_bh_check(p, c):
Use explicit check expression "c" along with rcu_read_lock_bh_held(). This is useful in code that is invoked by both RCU-bh readers and updaters.

rcu_dereference_sched_check(p, c):
Use explicit check expression "c" along with rcu_read_lock_sched_held(). This is useful in code that is invoked by both RCU-sched readers and updaters.

srcu_dereference_check(p, c):
Use explicit check expression "c" along with srcu_read_lock_held(). This is useful in code that is invoked by both SRCU readers and updaters.

rcu_dereference_raw(p):
Don't check. (Use sparingly, if at all.)

rcu_dereference_raw_check(p):
Don't do lockdep at all. (Use sparingly, if at all.)

rcu_dereference_protected(p, c):
Use explicit check expression "c", and omit all barriers and compiler constraints. This is useful when the data structure cannot change, for example, in code that is invoked only by updaters.

rcu_access_pointer(p):
Return the value of the pointer and omit all barriers, but retain the compiler constraints that prevent duplicating or coalescing. This is useful when testing the value of the pointer itself, for example, against NULL.

The rcu_dereference_check() check expression can be any boolean expression, but would normally include a lockdep expression. For a moderately ornate example, consider the following:

```
file = rcu_dereference_check(fdt->fd[fd],
                             lockdep_is_held(&files->file_lock) ||
                             atomic_read(&files->count) == 1);
```

This expression picks up the pointer "fdt->fd[fd]" in an RCU-safe manner, and, if CONFIG_PROVE_RCU is configured, verifies that this expression is used in:

1. An RCU read-side critical section (implicit), or
2. with files->file_lock held, or
3. on an unshared files_struct.

In case (1), the pointer is picked up in an RCU-safe manner for vanilla RCU read-side critical sections, in case (2) the ->file_lock prevents any change from taking place, and finally, in case (3) the current task is the only task accessing the file_struct, again preventing any change from taking place. If the above statement was invoked only from updater code, it could instead be written as follows:

```
file = rcu_dereference_protected(fdt->fd[fd],
                                 lockdep_is_held(&files->file_lock) ||
                                 atomic_read(&files->count) == 1);
```

This would verify cases #2 and #3 above, and furthermore lockdep would complain even if this was used in an RCU read-side critical section unless one of these two cases held. Because rcu_dereference_protected() omits all barriers and compiler constraints, it generates better code than do the other flavors of rcu_dereference(). On the other hand, it is illegal to use rcu_dereference_protected() if either the RCU-protected pointer or the RCU-protected data that it points to can change concurrently.

Like rcu_dereference(), when lockdep is enabled, RCU list and hlist traversal primitives check for being called from within an RCU read-side critical section. However, a lockdep expression can be passed to them as a additional optional argument. With this lockdep expression, these traversal primitives will complain only if the lockdep expression is false and they are called from outside any RCU read-side critical section.

For example, the workqueue for_each_pwq() macro is intended to be used either within an RCU read-side critical section or with wq->mutex held. It is thus implemented as follows:

```
#define for_each_pwq(pwq, wq)
        list_for_each_entry_rcu((pwq), &(wq)->pwqs, pwqs_node,
                                lock_is_held(&(wq->mutex).dep_map))
```

CHAPTER

THREE

LOCKDEP-RCU SPLAT

Lockdep-RCU was added to the Linux kernel in early 2010 (http://lwn.net/Articles/371986/). This facility checks for some common misuses of the RCU API, most notably using one of the rcu_dereference() family to access an RCU-protected pointer without the proper protection. When such misuse is detected, an lockdep-RCU splat is emitted.

The usual cause of a lockdep-RCU splat is someone accessing an RCU-protected data structure without either (1) being in the right kind of RCU read-side critical section or (2) holding the right update-side lock. This problem can therefore be serious: it might result in random memory overwriting or worse. There can of course be false positives, this being the real world and all that.

So let's look at an example RCU lockdep splat from 3.0-rc5, one that has long since been fixed:

```
=============================
WARNING: suspicious RCU usage
-----------------------------
block/cfq-iosched.c:2776 suspicious rcu_dereference_protected() usage!
```

other info that might help us debug this:

```
rcu_scheduler_active = 1, debug_locks = 0
3 locks held by scsi_scan_6/1552:
#0:  (&shost->scan_mutex){+.+.}, at: [<ffffffff8145efca>]
scsi_scan_host_selected+0x5a/0x150
#1:  (&eq->sysfs_lock){+.+.}, at: [<ffffffff812a5032>]
elevator_exit+0x22/0x60
#2:  (&(&q->__queue_lock)->rlock){-.-.}, at: [<ffffffff812b6233>]
cfq_exit_queue+0x43/0x190

stack backtrace:
Pid: 1552, comm: scsi_scan_6 Not tainted 3.0.0-rc5 #17
Call Trace:
[<ffffffff810abb9b>] lockdep_rcu_dereference+0xbb/0xc0
[<ffffffff812b6139>] __cfq_exit_single_io_context+0xe9/0x120
[<ffffffff812b626c>] cfq_exit_queue+0x7c/0x190
[<ffffffff812a5046>] elevator_exit+0x36/0x60
[<ffffffff812a802a>] blk_cleanup_queue+0x4a/0x60
[<ffffffff8145cc09>] scsi_free_queue+0x9/0x10
[<ffffffff81460944>] __scsi_remove_device+0x84/0xd0
[<ffffffff8145dca3>] scsi_probe_and_add_lun+0x353/0xb10
[<ffffffff817da069>] ? error_exit+0x29/0xb0
```

```
[<ffffffff817d98ed>] ? _raw_spin_unlock_irqrestore+0x3d/0x80
[<ffffffff8145e722>] __scsi_scan_target+0x112/0x680
[<ffffffff812c690d>] ? trace_hardirqs_off_thunk+0x3a/0x3c
[<ffffffff817da069>] ? error_exit+0x29/0xb0
[<ffffffff812bcc60>] ? kobject_del+0x40/0x40
[<ffffffff8145ed16>] scsi_scan_channel+0x86/0xb0
[<ffffffff8145f0b0>] scsi_scan_host_selected+0x140/0x150
[<ffffffff8145f149>] do_scsi_scan_host+0x89/0x90
[<ffffffff8145f170>] do_scan_async+0x20/0x160
[<ffffffff8145f150>] ? do_scsi_scan_host+0x90/0x90
[<ffffffff810975b6>] kthread+0xa6/0xb0
[<ffffffff817db154>] kernel_thread_helper+0x4/0x10
[<ffffffff81066430>] ? finish_task_switch+0x80/0x110
[<ffffffff817d9c04>] ? retint_restore_args+0xe/0xe
[<ffffffff81097510>] ? __kthread_init_worker+0x70/0x70
[<ffffffff817db150>] ? gs_change+0xb/0xb
```

Line 2776 of block/cfq-iosched.c in v3.0-rc5 is as follows:

```
if (rcu_dereference(ioc->ioc_data) == cic) {
```

This form says that it must be in a plain vanilla RCU read-side critical section, but the "other info" list above shows that this is not the case. Instead, we hold three locks, one of which might be RCU related. And maybe that lock really does protect this reference. If so, the fix is to inform RCU, perhaps by changing __cfq_exit_single_io_context() to take the struct request_queue "q" from cfq_exit_queue() as an argument, which would permit us to invoke rcu_dereference_protected as follows:

```
if (rcu_dereference_protected(ioc->ioc_data,
                              lockdep_is_held(&q->queue_lock)) == cic) {
```

With this change, there would be no lockdep-RCU splat emitted if this code was invoked either from within an RCU read-side critical section or with the ->queue_lock held. In particular, this would have suppressed the above lockdep-RCU splat because ->queue_lock is held (see #2 in the list above).

On the other hand, perhaps we really do need an RCU read-side critical section. In this case, the critical section must span the use of the return value from rcu_dereference(), or at least until there is some reference count incremented or some such. One way to handle this is to add rcu_read_lock() and rcu_read_unlock() as follows:

```
rcu_read_lock();
if (rcu_dereference(ioc->ioc_data) == cic) {
        spin_lock(&ioc->lock);
        rcu_assign_pointer(ioc->ioc_data, NULL);
        spin_unlock(&ioc->lock);
}
rcu_read_unlock();
```

With this change, the rcu_dereference() is always within an RCU read-side critical section, which again would have suppressed the above lockdep-RCU splat.

But in this particular case, we don't actually dereference the pointer returned from rcu_dereference(). Instead, that pointer is just compared to the cic pointer, which means that the rcu_dereference() can be replaced by rcu_access_pointer() as follows:

```
if (rcu_access_pointer(ioc->ioc_data) == cic) {
```

Because it is legal to invoke rcu_access_pointer() without protection, this change would also suppress the above lockdep-RCU splat.

CHAPTER

FOUR

RCU AND UNLOADABLE MODULES

[Originally published in LWN Jan. 14, 2007: http://lwn.net/Articles/217484/]

RCU updaters sometimes use call_rcu() to initiate an asynchronous wait for a grace period to elapse. This primitive takes a pointer to an rcu_head struct placed within the RCU-protected data structure and another pointer to a function that may be invoked later to free that structure. Code to delete an element p from the linked list from IRQ context might then be as follows:

```
list_del_rcu(p);
call_rcu(&p->rcu, p_callback);
```

Since call_rcu() never blocks, this code can safely be used from within IRQ context. The function p_callback() might be defined as follows:

```
static void p_callback(struct rcu_head *rp)
{
        struct pstruct *p = container_of(rp, struct pstruct, rcu);

        kfree(p);
}
```

4.1 Unloading Modules That Use call_rcu()

But what if the p_callback() function is defined in an unloadable module?

If we unload the module while some RCU callbacks are pending, the CPUs executing these callbacks are going to be severely disappointed when they are later invoked, as fancifully depicted at http://lwn.net/images/ns/kernel/rcu-drop.jpg.

We could try placing a synchronize_rcu() in the module-exit code path, but this is not sufficient. Although synchronize_rcu() does wait for a grace period to elapse, it does not wait for the callbacks to complete.

One might be tempted to try several back-to-back synchronize_rcu() calls, but this is still not guaranteed to work. If there is a very heavy RCU-callback load, then some of the callbacks might be deferred in order to allow other processing to proceed. For but one example, such deferral is required in realtime kernels in order to avoid excessive scheduling latencies.

4.2 rcu_barrier()

This situation can be handled by the rcu_barrier() primitive. Rather than waiting for a grace period to elapse, rcu_barrier() waits for all outstanding RCU callbacks to complete. Please note that rcu_barrier() does **not** imply synchronize_rcu(), in particular, if there are no RCU callbacks queued anywhere, rcu_barrier() is within its rights to return immediately, without waiting for anything, let alone a grace period.

Pseudo-code using rcu_barrier() is as follows:

1. Prevent any new RCU callbacks from being posted.
2. Execute rcu_barrier().
3. Allow the module to be unloaded.

There is also an srcu_barrier() function for SRCU, and you of course must match the flavor of srcu_barrier() with that of call_srcu(). If your module uses multiple srcu_struct structures, then it must also use multiple invocations of srcu_barrier() when unloading that module. For example, if it uses call_rcu(), call_srcu() on srcu_struct_1, and call_srcu() on srcu_struct_2, then the following three lines of code will be required when unloading:

```
rcu_barrier();
srcu_barrier(&srcu_struct_1);
srcu_barrier(&srcu_struct_2);
```

If latency is of the essence, workqueues could be used to run these three functions concurrently.

An ancient version of the rcutorture module makes use of rcu_barrier() in its exit function as follows:

```
static void
rcu_torture_cleanup(void)
{
  int i;

  fullstop = 1;
  if (shuffler_task != NULL) {
    VERBOSE_PRINTK_STRING("Stopping rcu_torture_shuffle task");
    kthread_stop(shuffler_task);
  }
  shuffler_task = NULL;

  if (writer_task != NULL) {
    VERBOSE_PRINTK_STRING("Stopping rcu_torture_writer task");
    kthread_stop(writer_task);
  }
  writer_task = NULL;

  if (reader_tasks != NULL) {
    for (i = 0; i < nrealreaders; i++) {
      if (reader_tasks[i] != NULL) {
        VERBOSE_PRINTK_STRING(
          "Stopping rcu_torture_reader task");
```

```
24          kthread_stop(reader_tasks[i]);
25        }
26        reader_tasks[i] = NULL;
27      }
28      kfree(reader_tasks);
29      reader_tasks = NULL;
30    }
31    rcu_torture_current = NULL;
32
33    if (fakewriter_tasks != NULL) {
34      for (i = 0; i < nfakewriters; i++) {
35        if (fakewriter_tasks[i] != NULL) {
36          VERBOSE_PRINTK_STRING(
37            "Stopping rcu_torture_fakewriter task");
38          kthread_stop(fakewriter_tasks[i]);
39        }
40        fakewriter_tasks[i] = NULL;
41      }
42      kfree(fakewriter_tasks);
43      fakewriter_tasks = NULL;
44    }
45
46    if (stats_task != NULL) {
47      VERBOSE_PRINTK_STRING("Stopping rcu_torture_stats task");
48      kthread_stop(stats_task);
49    }
50    stats_task = NULL;
51
52    /* Wait for all RCU callbacks to fire. */
53    rcu_barrier();
54
55    rcu_torture_stats_print(); /* -After- the stats thread is stopped! */
56
57    if (cur_ops->cleanup != NULL)
58      cur_ops->cleanup();
59    if (atomic_read(&n_rcu_torture_error))
60      rcu_torture_print_module_parms("End of test: FAILURE");
61    else
62      rcu_torture_print_module_parms("End of test: SUCCESS");
63  }
```

Line 6 sets a global variable that prevents any RCU callbacks from re-posting themselves. This will not be necessary in most cases, since RCU callbacks rarely include calls to call_rcu(). However, the rcutorture module is an exception to this rule, and therefore needs to set this global variable.

Lines 7-50 stop all the kernel tasks associated with the rcutorture module. Therefore, once execution reaches line 53, no more rcutorture RCU callbacks will be posted. The rcu_barrier() call on line 53 waits for any pre-existing callbacks to complete.

Then lines 55-62 print status and do operation-specific cleanup, and then return, permitting the module-unload operation to be completed.

Quick Quiz #1:

Is there any other situation where rcu_barrier() might be required?

Answer to Quick Quiz #1

Your module might have additional complications. For example, if your module invokes call_rcu() from timers, you will need to first refrain from posting new timers, cancel (or wait for) all the already-posted timers, and only then invoke rcu_barrier() to wait for any remaining RCU callbacks to complete.

Of course, if your module uses call_rcu(), you will need to invoke rcu_barrier() before unloading. Similarly, if your module uses call_srcu(), you will need to invoke srcu_barrier() before unloading, and on the same srcu_struct structure. If your module uses call_rcu() **and** call_srcu(), then (as noted above) you will need to invoke rcu_barrier() **and** srcu_barrier().

4.3 Implementing rcu_barrier()

Dipankar Sarma's implementation of rcu_barrier() makes use of the fact that RCU callbacks are never reordered once queued on one of the per-CPU queues. His implementation queues an RCU callback on each of the per-CPU callback queues, and then waits until they have all started executing, at which point, all earlier RCU callbacks are guaranteed to have completed.

The original code for rcu_barrier() was roughly as follows:

```
 1  void rcu_barrier(void)
 2  {
 3    BUG_ON(in_interrupt());
 4    /* Take cpucontrol mutex to protect against CPU hotplug */
 5    mutex_lock(&rcu_barrier_mutex);
 6    init_completion(&rcu_barrier_completion);
 7    atomic_set(&rcu_barrier_cpu_count, 1);
 8    on_each_cpu(rcu_barrier_func, NULL, 0, 1);
 9    if (atomic_dec_and_test(&rcu_barrier_cpu_count))
10      complete(&rcu_barrier_completion);
11    wait_for_completion(&rcu_barrier_completion);
12    mutex_unlock(&rcu_barrier_mutex);
13  }
```

Line 3 verifies that the caller is in process context, and lines 5 and 12 use rcu_barrier_mutex to ensure that only one rcu_barrier() is using the global completion and counters at a time, which are initialized on lines 6 and 7. Line 8 causes each CPU to invoke rcu_barrier_func(), which is shown below. Note that the final "1" in on_each_cpu()'s argument list ensures that all the calls to rcu_barrier_func() will have completed before on_each_cpu() returns. Line 9 removes the initial count from rcu_barrier_cpu_count, and if this count is now zero, line 10 finalizes the completion, which prevents line 11 from blocking. Either way, line 11 then waits (if needed) for the completion.

Quick Quiz #2:

Why doesn't line 8 initialize rcu_barrier_cpu_count to zero, thereby avoiding the need for lines 9 and 10?

Answer to Quick Quiz #2

This code was rewritten in 2008 and several times thereafter, but this still gives the general idea.

The rcu_barrier_func() runs on each CPU, where it invokes call_rcu() to post an RCU callback, as follows:

```
 1  static void rcu_barrier_func(void *notused)
 2  {
 3    int cpu = smp_processor_id();
 4    struct rcu_data *rdp = &per_cpu(rcu_data, cpu);
 5    struct rcu_head *head;
 6
 7    head = &rdp->barrier;
 8    atomic_inc(&rcu_barrier_cpu_count);
 9    call_rcu(head, rcu_barrier_callback);
10  }
```

Lines 3 and 4 locate RCU's internal per-CPU rcu_data structure, which contains the struct rcu_head that needed for the later call to call_rcu(). Line 7 picks up a pointer to this struct rcu_head, and line 8 increments the global counter. This counter will later be decremented by the callback. Line 9 then registers the rcu_barrier_callback() on the current CPU's queue.

The rcu_barrier_callback() function simply atomically decrements the rcu_barrier_cpu_count variable and finalizes the completion when it reaches zero, as follows:

```
static void rcu_barrier_callback(struct rcu_head *notused)
{
  if (atomic_dec_and_test(&rcu_barrier_cpu_count))
    complete(&rcu_barrier_completion);
}
```

Quick Quiz #3:

What happens if CPU 0's rcu_barrier_func() executes immediately (thus incrementing rcu_barrier_cpu_count to the value one), but the other CPU's rcu_barrier_func() invocations are delayed for a full grace period? Couldn't this result in rcu_barrier() returning prematurely?

Answer to Quick Quiz #3

The current rcu_barrier() implementation is more complex, due to the need to avoid disturbing idle CPUs (especially on battery-powered systems) and the need to minimally disturb non-idle CPUs in real-time systems. In addition, a great many optimizations have been applied. However, the code above illustrates the concepts.

4.4 rcu_barrier() Summary

The rcu_barrier() primitive is used relatively infrequently, since most code using RCU is in the core kernel rather than in modules. However, if you are using RCU from an unloadable module, you need to use rcu_barrier() so that your module may be safely unloaded.

4.5 Answers to Quick Quizzes

Quick Quiz #1:

Is there any other situation where rcu_barrier() might be required?

Answer:

Interestingly enough, rcu_barrier() was not originally implemented for module unloading. Nikita Danilov was using RCU in a filesystem, which resulted in a similar situation at filesystem-unmount time. Dipankar Sarma coded up rcu_barrier() in response, so that Nikita could invoke it during the filesystem-unmount process.

Much later, yours truly hit the RCU module-unload problem when implementing rcutorture, and found that rcu_barrier() solves this problem as well.

Back to Quick Quiz #1

Quick Quiz #2:

Why doesn't line 8 initialize rcu_barrier_cpu_count to zero, thereby avoiding the need for lines 9 and 10?

Answer:

Suppose that the on_each_cpu() function shown on line 8 was delayed, so that CPU 0's rcu_barrier_func() executed and the corresponding grace period elapsed, all before CPU 1's rcu_barrier_func() started executing. This would result in rcu_barrier_cpu_count being decremented to zero, so that line 11's wait_for_completion() would return immediately, failing to wait for CPU 1's callbacks to be invoked.

Note that this was not a problem when the rcu_barrier() code was first added back in 2005. This is because on_each_cpu() disables preemption, which acted as an RCU read-side critical section, thus preventing CPU 0's grace period from completing until on_each_cpu() had dealt with all of the CPUs. However, with the advent of preemptible RCU, rcu_barrier() no longer waited on nonpreemptible regions of code in preemptible kernels, that being the job of the new rcu_barrier_sched() function.

However, with the RCU flavor consolidation around v4.20, this possibility was once again ruled out, because the consolidated RCU once again waits on nonpreemptible regions of code.

Nevertheless, that extra count might still be a good idea. Relying on these sort of accidents of implementation can result in later surprise bugs when the implementation changes.

Back to Quick Quiz #2

Quick Quiz #3:

What happens if CPU 0's rcu_barrier_func() executes immediately (thus incrementing rcu_barrier_cpu_count to the value one), but the other CPU's rcu_barrier_func() invocations are delayed for a full grace period? Couldn't this result in rcu_barrier() returning prematurely?

Answer:

This cannot happen. The reason is that on_each_cpu() has its last argument, the wait flag, set to "1". This flag is passed through to smp_call_function() and further to smp_call_function_on_cpu(), causing this latter to spin until the cross-CPU invocation of rcu_barrier_func() has completed. This by itself would prevent a grace period from completing on non-CONFIG_PREEMPTION kernels, since each CPU must undergo a context switch (or other quiescent state) before the grace period can complete. However, this is of no use in CONFIG_PREEMPTION kernels.

Therefore, on_each_cpu() disables preemption across its call to smp_call_function() and also across the local call to rcu_barrier_func(). Because recent RCU implementations treat preemption-disabled regions of code as RCU read-side critical sections, this prevents grace periods from completing. This means that all CPUs have executed rcu_barrier_func() before the first rcu_barrier_callback() can possibly execute, in turn preventing rcu_barrier_cpu_count from prematurely reaching zero.

But if on_each_cpu() ever decides to forgo disabling preemption, as might well happen due to real-time latency considerations, initializing rcu_barrier_cpu_count to one will save the day.

Back to Quick Quiz #3

CHAPTER

FIVE

PROPER CARE AND FEEDING OF RETURN VALUES FROM RCU_DEREFERENCE()

Most of the time, you can use values from rcu_dereference() or one of the similar primitives without worries. Dereferencing (prefix "*"), field selection ("->"), assignment ("="), address-of ("&"), addition and subtraction of constants, and casts all work quite naturally and safely.

It is nevertheless possible to get into trouble with other operations. Follow these rules to keep your RCU code working properly:

- You must use one of the rcu_dereference() family of primitives to load an RCU-protected pointer, otherwise CONFIG_PROVE_RCU will complain. Worse yet, your code can see random memory-corruption bugs due to games that compilers and DEC Alpha can play. Without one of the rcu_dereference() primitives, compilers can reload the value, and won't your code have fun with two different values for a single pointer! Without rcu_dereference(), DEC Alpha can load a pointer, dereference that pointer, and return data preceding initialization that preceded the store of the pointer. (As noted later, in recent kernels READ_ONCE() also prevents DEC Alpha from playing these tricks.)

  In addition, the volatile cast in rcu_dereference() prevents the compiler from deducing the resulting pointer value. Please see the section entitled "EXAMPLE WHERE THE COMPILER KNOWS TOO MUCH" for an example where the compiler can in fact deduce the exact value of the pointer, and thus cause misordering.

- In the special case where data is added but is never removed while readers are accessing the structure, READ_ONCE() may be used instead of rcu_dereference(). In this case, use of READ_ONCE() takes on the role of the lockless_dereference() primitive that was removed in v4.15.

- You are only permitted to use rcu_dereference() on pointer values. The compiler simply knows too much about integral values to trust it to carry dependencies through integer operations. There are a very few exceptions, namely that you can temporarily cast the pointer to uintptr_t in order to:

  - Set bits and clear bits down in the must-be-zero low-order bits of that pointer. This clearly means that the pointer must have alignment constraints, for example, this does *not* work in general for char* pointers.

  - XOR bits to translate pointers, as is done in some classic buddy-allocator algorithms.

  It is important to cast the value back to pointer before doing much of anything else with it.

- Avoid cancellation when using the "+" and "-" infix arithmetic operators. For example, for a given variable "x", avoid "(x-(uintptr_t)x)" for char* pointers. The compiler is within its rights to substitute zero for this sort of expression, so that subsequent accesses no longer depend on the rcu_dereference(), again possibly resulting in bugs due to misordering.

Of course, if “p” is a pointer from rcu_dereference(), and “a” and “b” are integers that happen to be equal, the expression “p+a-b” is safe because its value still necessarily depends on the rcu_dereference(), thus maintaining proper ordering.

- If you are using RCU to protect JITed functions, so that the “()” function-invocation operator is applied to a value obtained (directly or indirectly) from rcu_dereference(), you may need to interact directly with the hardware to flush instruction caches. This issue arises on some systems when a newly JITed function is using the same memory that was used by an earlier JITed function.
- Do not use the results from relational operators (“==”, “!=”, “>”, “>=”, “<”, or “<=”) when dereferencing. For example, the following (quite strange) code is buggy:

```
int *p;
int *q;

...

p = rcu_dereference(gp)
q = &global_q;
q += p > &oom_p;
r1 = *q;  /* BUGGY!!! */
```

  As before, the reason this is buggy is that relational operators are often compiled using branches. And as before, although weak-memory machines such as ARM or PowerPC do order stores after such branches, but can speculate loads, which can again result in misordering bugs.

- Be very careful about comparing pointers obtained from rcu_dereference() against non-NULL values. As Linus Torvalds explained, if the two pointers are equal, the compiler could substitute the pointer you are comparing against for the pointer obtained from rcu_dereference(). For example:

```
p = rcu_dereference(gp);
if (p == &default_struct)
        do_default(p->a);
```

  Because the compiler now knows that the value of “p” is exactly the address of the variable “default_struct”, it is free to transform this code into the following:

```
p = rcu_dereference(gp);
if (p == &default_struct)
        do_default(default_struct.a);
```

  On ARM and Power hardware, the load from “default_struct.a” can now be speculated, such that it might happen before the rcu_dereference(). This could result in bugs due to misordering.

  However, comparisons are OK in the following cases:

  - The comparison was against the NULL pointer. If the compiler knows that the pointer is NULL, you had better not be dereferencing it anyway. If the comparison is nonequal, the compiler is none the wiser. Therefore, it is safe to compare pointers from rcu_dereference() against NULL pointers.

- The pointer is never dereferenced after being compared. Since there are no subsequent dereferences, the compiler cannot use anything it learned from the comparison to reorder the non-existent subsequent dereferences. This sort of comparison occurs frequently when scanning RCU-protected circular linked lists.

  Note that if the pointer comparison is done outside of an RCU read-side critical section, and the pointer is never dereferenced, rcu_access_pointer() should be used in place of rcu_dereference(). In most cases, it is best to avoid accidental dereferences by testing the rcu_access_pointer() return value directly, without assigning it to a variable.

  Within an RCU read-side critical section, there is little reason to use rcu_access_pointer().

- The comparison is against a pointer that references memory that was initialized "a long time ago." The reason this is safe is that even if misordering occurs, the misordering will not affect the accesses that follow the comparison. So exactly how long ago is "a long time ago"? Here are some possibilities:

  * Compile time.
  * Boot time.
  * Module-init time for module code.
  * Prior to kthread creation for kthread code.
  * During some prior acquisition of the lock that we now hold.
  * Before mod_timer() time for a timer handler.

  There are many other possibilities involving the Linux kernel's wide array of primitives that cause code to be invoked at a later time.

- The pointer being compared against also came from rcu_dereference(). In this case, both pointers depend on one rcu_dereference() or another, so you get proper ordering either way.

  That said, this situation can make certain RCU usage bugs more likely to happen. Which can be a good thing, at least if they happen during testing. An example of such an RCU usage bug is shown in the section titled "EXAMPLE OF AMPLIFIED RCU-USAGE BUG".

- All of the accesses following the comparison are stores, so that a control dependency preserves the needed ordering. That said, it is easy to get control dependencies wrong. Please see the "CONTROL DEPENDENCIES" section of Documentation/memory-barriers.txt for more details.

- The pointers are not equal *and* the compiler does not have enough information to deduce the value of the pointer. Note that the volatile cast in rcu_dereference() will normally prevent the compiler from knowing too much.

  However, please note that if the compiler knows that the pointer takes on only one of two values, a not-equal comparison will provide exactly the information that the compiler needs to deduce the value of the pointer.

- Disable any value-speculation optimizations that your compiler might provide, especially if you are making use of feedback-based optimizations that take data collected from prior runs. Such value-speculation optimizations reorder operations by design.

  There is one exception to this rule: Value-speculation optimizations that leverage the branch-prediction hardware are safe on strongly ordered systems (such as x86), but not on weakly ordered systems (such as ARM or Power). Choose your compiler command-line options wisely!

5.1 EXAMPLE OF AMPLIFIED RCU-USAGE BUG

Because updaters can run concurrently with RCU readers, RCU readers can see stale and/or inconsistent values. If RCU readers need fresh or consistent values, which they sometimes do, they need to take proper precautions. To see this, consider the following code fragment:

```
struct foo {
        int a;
        int b;
        int c;
};
struct foo *gp1;
struct foo *gp2;

void updater(void)
{
        struct foo *p;

        p = kmalloc(...);
        if (p == NULL)
                deal_with_it();
        p->a = 42;  /* Each field in its own cache line. */
        p->b = 43;
        p->c = 44;
        rcu_assign_pointer(gp1, p);
        p->b = 143;
        p->c = 144;
        rcu_assign_pointer(gp2, p);
}

void reader(void)
{
        struct foo *p;
        struct foo *q;
        int r1, r2;

        rcu_read_lock();
        p = rcu_dereference(gp2);
        if (p == NULL)
                return;
        r1 = p->b;  /* Guaranteed to get 143. */
```

```
        q = rcu_dereference(gp1);  /* Guaranteed non-NULL. */
        if (p == q) {
                /* The compiler decides that q->c is same as p->c. */
                r2 = p->c; /* Could get 44 on weakly order system. */
        } else {
                r2 = p->c - r1; /* Unconditional access to p->c. */
        }
        rcu_read_unlock();
        do_something_with(r1, r2);
}
```

You might be surprised that the outcome (r1 == 143 && r2 == 44) is possible, but you should not be. After all, the updater might have been invoked a second time between the time reader() loaded into "r1" and the time that it loaded into "r2". The fact that this same result can occur due to some reordering from the compiler and CPUs is beside the point.

But suppose that the reader needs a consistent view?

Then one approach is to use locking, for example, as follows:

```
struct foo {
        int a;
        int b;
        int c;
        spinlock_t lock;
};
struct foo *gp1;
struct foo *gp2;

void updater(void)
{
        struct foo *p;

        p = kmalloc(...);
        if (p == NULL)
                deal_with_it();
        spin_lock(&p->lock);
        p->a = 42;  /* Each field in its own cache line. */
        p->b = 43;
        p->c = 44;
        spin_unlock(&p->lock);
        rcu_assign_pointer(gp1, p);
        spin_lock(&p->lock);
        p->b = 143;
        p->c = 144;
        spin_unlock(&p->lock);
        rcu_assign_pointer(gp2, p);
}

void reader(void)
{
        struct foo *p;
```

```
        struct foo *q;
        int r1, r2;

        rcu_read_lock();
        p = rcu_dereference(gp2);
        if (p == NULL)
                return;
        spin_lock(&p->lock);
        r1 = p->b;  /* Guaranteed to get 143. */
        q = rcu_dereference(gp1);  /* Guaranteed non-NULL. */
        if (p == q) {
                /* The compiler decides that q->c is same as p->c. */
                r2 = p->c; /* Locking guarantees r2 == 144. */
        } else {
                spin_lock(&q->lock);
                r2 = q->c - r1;
                spin_unlock(&q->lock);
        }
        rcu_read_unlock();
        spin_unlock(&p->lock);
        do_something_with(r1, r2);
}
```

As always, use the right tool for the job!

5.2 EXAMPLE WHERE THE COMPILER KNOWS TOO MUCH

If a pointer obtained from rcu_dereference() compares not-equal to some other pointer, the compiler normally has no clue what the value of the first pointer might be. This lack of knowledge prevents the compiler from carrying out optimizations that otherwise might destroy the ordering guarantees that RCU depends on. And the volatile cast in rcu_dereference() should prevent the compiler from guessing the value.

But without rcu_dereference(), the compiler knows more than you might expect. Consider the following code fragment:

```
struct foo {
        int a;
        int b;
};
static struct foo variable1;
static struct foo variable2;
static struct foo *gp = &variable1;

void updater(void)
{
        initialize_foo(&variable2);
        rcu_assign_pointer(gp, &variable2);
        /*
         * The above is the only store to gp in this translation unit,
```

```
         * and the address of gp is not exported in any way.
         */
}

int reader(void)
{
        struct foo *p;

        p = gp;
        barrier();
        if (p == &variable1)
                return p->a; /* Must be variable1.a. */
        else
                return p->b; /* Must be variable2.b. */
}
```

Because the compiler can see all stores to "gp", it knows that the only possible values of "gp" are "variable1" on the one hand and "variable2" on the other. The comparison in reader() therefore tells the compiler the exact value of "p" even in the not-equals case. This allows the compiler to make the return values independent of the load from "gp", in turn destroying the ordering between this load and the loads of the return values. This can result in "p->b" returning pre-initialization garbage values on weakly ordered systems.

In short, rcu_dereference() is *not* optional when you are going to dereference the resulting pointer.

5.3 WHICH MEMBER OF THE rcu_dereference() FAMILY SHOULD YOU USE?

First, please avoid using rcu_dereference_raw() and also please avoid using rcu_dereference_check() and rcu_dereference_protected() with a second argument with a constant value of 1 (or true, for that matter). With that caution out of the way, here is some guidance for which member of the rcu_dereference() to use in various situations:

1. If the access needs to be within an RCU read-side critical section, use rcu_dereference(). With the new consolidated RCU flavors, an RCU read-side critical section is entered using rcu_read_lock(), anything that disables bottom halves, anything that disables interrupts, or anything that disables preemption.
2. If the access might be within an RCU read-side critical section on the one hand, or protected by (say) my_lock on the other, use rcu_dereference_check(), for example:

   ```
   p1 = rcu_dereference_check(p->rcu_protected_pointer,
                              lockdep_is_held(&my_lock));
   ```

3. If the access might be within an RCU read-side critical section on the one hand, or protected by either my_lock or your_lock on the other, again use rcu_dereference_check(), for example:

```
p1 = rcu_dereference_check(p->rcu_protected_pointer,
                           lockdep_is_held(&my_lock) ||
                           lockdep_is_held(&your_lock));
```

4. If the access is on the update side, so that it is always protected by my_lock, use rcu_dereference_protected():

   ```
   p1 = rcu_dereference_protected(p->rcu_protected_pointer,
                                  lockdep_is_held(&my_lock));
   ```

   This can be extended to handle multiple locks as in #3 above, and both can be extended to check other conditions as well.

5. If the protection is supplied by the caller, and is thus unknown to this code, that is the rare case when rcu_dereference_raw() is appropriate. In addition, rcu_dereference_raw() might be appropriate when the lockdep expression would be excessively complex, except that a better approach in that case might be to take a long hard look at your synchronization design. Still, there are data-locking cases where any one of a very large number of locks or reference counters suffices to protect the pointer, so rcu_dereference_raw() does have its place.

   However, its place is probably quite a bit smaller than one might expect given the number of uses in the current kernel. Ditto for its synonym, rcu_dereference_check(… , 1), and its close relative, rcu_dereference_protected(… , 1).

5.4 SPARSE CHECKING OF RCU-PROTECTED POINTERS

The sparse static-analysis tool checks for non-RCU access to RCU-protected pointers, which can result in "interesting" bugs due to compiler optimizations involving invented loads and perhaps also load tearing. For example, suppose someone mistakenly does something like this:

```
p = q->rcu_protected_pointer;
do_something_with(p->a);
do_something_else_with(p->b);
```

If register pressure is high, the compiler might optimize "p" out of existence, transforming the code to something like this:

```
do_something_with(q->rcu_protected_pointer->a);
do_something_else_with(q->rcu_protected_pointer->b);
```

This could fatally disappoint your code if q->rcu_protected_pointer changed in the meantime. Nor is this a theoretical problem: Exactly this sort of bug cost Paul E. McKenney (and several of his innocent colleagues) a three-day weekend back in the early 1990s.

Load tearing could of course result in dereferencing a mashup of a pair of pointers, which also might fatally disappoint your code.

These problems could have been avoided simply by making the code instead read as follows:

```
p = rcu_dereference(q->rcu_protected_pointer);
do_something_with(p->a);
do_something_else_with(p->b);
```

Unfortunately, these sorts of bugs can be extremely hard to spot during review. This is where the sparse tool comes into play, along with the "__rcu" marker. If you mark a pointer declaration, whether in a structure or as a formal parameter, with "__rcu", which tells sparse to complain if this pointer is accessed directly. It will also cause sparse to complain if a pointer not marked with "__rcu" is accessed using rcu_dereference() and friends. For example, ->rcu_protected_pointer might be declared as follows:

```
struct foo __rcu *rcu_protected_pointer;
```

Use of "__rcu" is opt-in. If you choose not to use it, then you should ignore the sparse warnings.

CHAPTER

SIX

WHAT IS RCU? -- "READ, COPY, UPDATE"

Please note that the "What is RCU?" LWN series is an excellent place to start learning about RCU:

1. What is RCU, Fundamentally? https://lwn.net/Articles/262464/
2. What is RCU? Part 2: Usage https://lwn.net/Articles/263130/
3. RCU part 3: the RCU API https://lwn.net/Articles/264090/
4. The RCU API, 2010 Edition https://lwn.net/Articles/418853/
   2010 Big API Table https://lwn.net/Articles/419086/
5. The RCU API, 2014 Edition https://lwn.net/Articles/609904/
   2014 Big API Table https://lwn.net/Articles/609973/
6. The RCU API, 2019 Edition https://lwn.net/Articles/777036/
   2019 Big API Table https://lwn.net/Articles/777165/

For those preferring video:

1. Unraveling RCU Mysteries: Fundamentals
https://www.linuxfoundation.org/webinars/unraveling-rcu-usage-mysteries
2. Unraveling RCU Mysteries: Additional Use Cases https://www.linuxfoundation.org/webinars/unraveling-rcu-usage-mysteries-additional-use-cases

What is RCU?

RCU is a synchronization mechanism that was added to the Linux kernel during the 2.5 development effort that is optimized for read-mostly situations. Although RCU is actually quite simple, making effective use of it requires you to think differently about your code. Another part of the problem is the mistaken assumption that there is "one true way" to describe and to use RCU. Instead, the experience has been that different people must take different paths to arrive at an understanding of RCU, depending on their experiences and use cases. This document provides several different paths, as follows:

People who prefer starting with a conceptual overview should focus on Section 1, though most readers will profit by reading this section at some point. People who prefer to start with an API that they can then experiment with should focus on Section 2. People who prefer to start with example uses should focus on Sections 3 and 4. People who need to understand the RCU implementation should focus on Section 5, then dive into the kernel source code. People who reason best by analogy should focus on Section 6. Section 7 serves as an index to the docbook API documentation, and Section 8 is the traditional answer key.

So, start with the section that makes the most sense to you and your preferred method of learning. If you need to know everything about everything, feel free to read the whole thing -- but if you are really that type of person, you have perused the source code and will therefore never need this document anyway. ;-)

6.1 1. RCU OVERVIEW

The basic idea behind RCU is to split updates into “removal” and “reclamation” phases. The removal phase removes references to data items within a data structure (possibly by replacing them with references to new versions of these data items), and can run concurrently with readers. The reason that it is safe to run the removal phase concurrently with readers is the semantics of modern CPUs guarantee that readers will see either the old or the new version of the data structure rather than a partially updated reference. The reclamation phase does the work of reclaiming (e.g., freeing) the data items removed from the data structure during the removal phase. Because reclaiming data items can disrupt any readers concurrently referencing those data items, the reclamation phase must not start until readers no longer hold references to those data items.

Splitting the update into removal and reclamation phases permits the updater to perform the removal phase immediately, and to defer the reclamation phase until all readers active during the removal phase have completed, either by blocking until they finish or by registering a callback that is invoked after they finish. Only readers that are active during the removal phase need be considered, because any reader starting after the removal phase will be unable to gain a reference to the removed data items, and therefore cannot be disrupted by the reclamation phase.

So the typical RCU update sequence goes something like the following:

a. Remove pointers to a data structure, so that subsequent readers cannot gain a reference to it.

b. Wait for all previous readers to complete their RCU read-side critical sections.

c. At this point, there cannot be any readers who hold references to the data structure, so it now may safely be reclaimed (e.g., kfree()d).

Step (b) above is the key idea underlying RCU's deferred destruction. The ability to wait until all readers are done allows RCU readers to use much lighter-weight synchronization, in some cases, absolutely no synchronization at all. In contrast, in more conventional lock-based schemes, readers must use heavy-weight synchronization in order to prevent an updater from deleting the data structure out from under them. This is because lock-based updaters typically update data items in place, and must therefore exclude readers. In contrast, RCU-based updaters typically take advantage of the fact that writes to single aligned pointers are atomic on modern CPUs, allowing atomic insertion, removal, and replacement of data items in a linked structure without disrupting readers. Concurrent RCU readers can then continue accessing the old versions, and can dispense with the atomic operations, memory barriers, and communications cache misses that are so expensive on present-day SMP computer systems, even in absence of lock contention.

In the three-step procedure shown above, the updater is performing both the removal and the reclamation step, but it is often helpful for an entirely different thread to do the reclamation, as is in fact the case in the Linux kernel's directory-entry cache (dcache). Even if the same thread performs both the update step (step (a) above) and the reclamation step (step (c) above), it is often helpful to think of them separately. For example, RCU readers and updaters need not communicate at all, but RCU provides implicit low-overhead communication between readers and reclaimers, namely, in step (b) above.

So how the heck can a reclaimer tell when a reader is done, given that readers are not doing any sort of synchronization operations??? Read on to learn about how RCU's API makes this easy.

6.2 2. WHAT IS RCU'S CORE API?

The core RCU API is quite small:

a. rcu_read_lock()

b. rcu_read_unlock()

c. synchronize_rcu() / call_rcu()

d. rcu_assign_pointer()

e. rcu_dereference()

There are many other members of the RCU API, but the rest can be expressed in terms of these five, though most implementations instead express synchronize_rcu() in terms of the call_rcu() callback API.

The five core RCU APIs are described below, the other 18 will be enumerated later. See the kernel docbook documentation for more info, or look directly at the function header comments.

6.2.1 rcu_read_lock()

void rcu_read_lock(void);

This temporal primitive is used by a reader to inform the reclaimer that the reader is entering an RCU read-side critical section. It is illegal to block while in an RCU read-side critical section, though kernels built with CONFIG_PREEMPT_RCU can preempt RCU read-side critical sections. Any RCU-protected data structure accessed during an RCU read-side critical section is guaranteed to remain unreclaimed for the full duration of that critical section. Reference counts may be used in conjunction with RCU to maintain longer-term references to data structures.

6.2.2 rcu_read_unlock()

void rcu_read_unlock(void);

This temporal primitives is used by a reader to inform the reclaimer that the reader is exiting an RCU read-side critical section. Note that RCU read-side critical sections may be nested and/or overlapping.

6.2.3 synchronize_rcu()

void synchronize_rcu(void);

This temporal primitive marks the end of updater code and the beginning of reclaimer code. It does this by blocking until all pre-existing RCU read-side critical sections on all CPUs have completed. Note that synchronize_rcu() will **not** necessarily wait for any subsequent RCU read-side critical sections to complete. For example, consider the following sequence of events:

```
         CPU 0                  CPU 1                 CPU 2
    ----------------- ------------------------- ---------------
1.  rcu_read_lock()
2.                    enters synchronize_rcu()
3.                                               rcu_read_lock()
4.  rcu_read_unlock()
5.                     exits synchronize_rcu()
6.                                              rcu_read_unlock()
```

To reiterate, synchronize_rcu() waits only for ongoing RCU read-side critical sections to complete, not necessarily for any that begin after synchronize_rcu() is invoked.

Of course, synchronize_rcu() does not necessarily return **immediately** after the last pre-existing RCU read-side critical section completes. For one thing, there might well be scheduling delays. For another thing, many RCU implementations process requests in batches in order to improve efficiencies, which can further delay synchronize_rcu().

Since synchronize_rcu() is the API that must figure out when readers are done, its implementation is key to RCU. For RCU to be useful in all but the most read-intensive situations, synchronize_rcu()'s overhead must also be quite small.

The call_rcu() API is an asynchronous callback form of synchronize_rcu(), and is described in more detail in a later section. Instead of blocking, it registers a function and argument which are invoked after all ongoing RCU read-side critical sections have completed. This callback variant is particularly useful in situations where it is illegal to block or where update-side performance is critically important.

However, the call_rcu() API should not be used lightly, as use of the synchronize_rcu() API generally results in simpler code. In addition, the synchronize_rcu() API has the nice property of automatically limiting update rate should grace periods be delayed. This property results in system resilience in face of denial-of-service attacks. Code using call_rcu() should limit update rate in order to gain this same sort of resilience. See *Review Checklist for RCU Patches* for some approaches to limiting the update rate.

6.2.4 rcu_assign_pointer()

void rcu_assign_pointer(p, typeof(p) v);

Yes, rcu_assign_pointer() **is** implemented as a macro, though it would be cool to be able to declare a function in this manner. (Compiler experts will no doubt disagree.)

The updater uses this spatial macro to assign a new value to an RCU-protected pointer, in order to safely communicate the change in value from the updater to the reader. This is a spatial (as opposed to temporal) macro. It does not evaluate to an rvalue, but it does execute any memory-barrier instructions required for a given CPU architecture. Its ordering properties are that of a store-release operation.

Perhaps just as important, it serves to document (1) which pointers are protected by RCU and (2) the point at which a given structure becomes accessible to other CPUs. That said, rcu_assign_pointer() is most frequently used indirectly, via the _rcu list-manipulation primitives such as list_add_rcu().

6.2.5 rcu_dereference()

typeof(p) rcu_dereference(p);

Like rcu_assign_pointer(), rcu_dereference() must be implemented as a macro.

The reader uses the spatial rcu_dereference() macro to fetch an RCU-protected pointer, which returns a value that may then be safely dereferenced. Note that rcu_dereference() does not actually dereference the pointer, instead, it protects the pointer for later dereferencing. It also executes any needed memory-barrier instructions for a given CPU architecture. Currently, only Alpha needs memory barriers within rcu_dereference() -- on other CPUs, it compiles to a volatile load.

Common coding practice uses rcu_dereference() to copy an RCU-protected pointer to a local variable, then dereferences this local variable, for example as follows:

```
p = rcu_dereference(head.next);
return p->data;
```

However, in this case, one could just as easily combine these into one statement:

```
return rcu_dereference(head.next)->data;
```

If you are going to be fetching multiple fields from the RCU-protected structure, using the local variable is of course preferred. Repeated rcu_dereference() calls look ugly, do not guarantee that the same pointer will be returned if an update happened while in the critical section, and incur unnecessary overhead on Alpha CPUs.

Note that the value returned by rcu_dereference() is valid only within the enclosing RCU read-side critical section[1]. For example, the following is **not** legal:

```
rcu_read_lock();
p = rcu_dereference(head.next);
rcu_read_unlock();
x = p->address; /* BUG!!! */
rcu_read_lock();
y = p->data;    /* BUG!!! */
rcu_read_unlock();
```

Holding a reference from one RCU read-side critical section to another is just as illegal as holding a reference from one lock-based critical section to another! Similarly, using a reference outside of the critical section in which it was acquired is just as illegal as doing so with normal locking.

As with rcu_assign_pointer(), an important function of rcu_dereference() is to document which pointers are protected by RCU, in particular, flagging a pointer that is subject to changing at any time, including immediately after the rcu_dereference(). And, again like rcu_assign_pointer(), rcu_dereference() is typically used indirectly, via the _rcu list-manipulation primitives, such as list_for_each_entry_rcu()[2].

The following diagram shows how each API communicates among the reader, updater, and reclaimer.

```
rcu_assign_pointer()
                        +--------+
+---------------------->| reader |---------+
|                       +--------+         |
|                            |             |
|                            |             | Protect:
|                            |             | rcu_read_lock()
|                            |             | rcu_read_unlock()
|        rcu_dereference()   |             |
+---------+                  |             |
| updater |<-----------------+             |
```

[1] The variant rcu_dereference_protected() can be used outside of an RCU read-side critical section as long as the usage is protected by locks acquired by the update-side code. This variant avoids the lockdep warning that would happen when using (for example) rcu_dereference() without rcu_read_lock() protection. Using rcu_dereference_protected() also has the advantage of permitting compiler optimizations that rcu_dereference() must prohibit. The rcu_dereference_protected() variant takes a lockdep expression to indicate which locks must be acquired by the caller. If the indicated protection is not provided, a lockdep splat is emitted. See *A Tour Through RCU's Requirements* and the API's code comments for more details and example usage.

[2] If the list_for_each_entry_rcu() instance might be used by update-side code as well as by RCU readers, then an additional lockdep expression can be added to its list of arguments. For example, given an additional "lock_is_held(&mylock)" argument, the RCU lockdep code would complain only if this instance was invoked outside of an RCU read-side critical section and without the protection of mylock.

```
+---------+                                 V
|                                     +-----------+
+----------------------------------->| reclaimer |
                                      +-----------+
  Defer:
  synchronize_rcu() & call_rcu()
```

The RCU infrastructure observes the temporal sequence of rcu_read_lock(), rcu_read_unlock(), synchronize_rcu(), and call_rcu() invocations in order to determine when (1) synchronize_rcu() invocations may return to their callers and (2) call_rcu() callbacks may be invoked. Efficient implementations of the RCU infrastructure make heavy use of batching in order to amortize their overhead over many uses of the corresponding APIs. The rcu_assign_pointer() and rcu_dereference() invocations communicate spatial changes via stores to and loads from the RCU-protected pointer in question.

There are at least three flavors of RCU usage in the Linux kernel. The diagram above shows the most common one. On the updater side, the rcu_assign_pointer(), synchronize_rcu() and call_rcu() primitives used are the same for all three flavors. However for protection (on the reader side), the primitives used vary depending on the flavor:

a. rcu_read_lock() / rcu_read_unlock() rcu_dereference()

b. rcu_read_lock_bh() / rcu_read_unlock_bh() local_bh_disable() / local_bh_enable() rcu_dereference_bh()

c. rcu_read_lock_sched() / rcu_read_unlock_sched() preempt_disable() / preempt_enable() local_irq_save() / local_irq_restore() hardirq enter / hardirq exit NMI enter / NMI exit rcu_dereference_sched()

These three flavors are used as follows:

a. RCU applied to normal data structures.

b. RCU applied to networking data structures that may be subjected to remote denial-of-service attacks.

c. RCU applied to scheduler and interrupt/NMI-handler tasks.

Again, most uses will be of (a). The (b) and (c) cases are important for specialized uses, but are relatively uncommon. The SRCU, RCU-Tasks, RCU-Tasks-Rude, and RCU-Tasks-Trace have similar relationships among their assorted primitives.

6.3 3. WHAT ARE SOME EXAMPLE USES OF CORE RCU API?

This section shows a simple use of the core RCU API to protect a global pointer to a dynamically allocated structure. More-typical uses of RCU may be found in *Using RCU to Protect Read-Mostly Linked Lists*, arrayRCU.rst, and *Using RCU to Protect Dynamic NMI Handlers*.

```
struct foo {
        int a;
        char b;
        long c;
};
DEFINE_SPINLOCK(foo_mutex);
```

```
struct foo __rcu *gbl_foo;

/*
 * Create a new struct foo that is the same as the one currently
 * pointed to by gbl_foo, except that field "a" is replaced
 * with "new_a".  Points gbl_foo to the new structure, and
 * frees up the old structure after a grace period.
 *
 * Uses rcu_assign_pointer() to ensure that concurrent readers
 * see the initialized version of the new structure.
 *
 * Uses synchronize_rcu() to ensure that any readers that might
 * have references to the old structure complete before freeing
 * the old structure.
 */
void foo_update_a(int new_a)
{
        struct foo *new_fp;
        struct foo *old_fp;

        new_fp = kmalloc(sizeof(*new_fp), GFP_KERNEL);
        spin_lock(&foo_mutex);
        old_fp = rcu_dereference_protected(gbl_foo, lockdep_is_held(&foo_
↪mutex));
        *new_fp = *old_fp;
        new_fp->a = new_a;
        rcu_assign_pointer(gbl_foo, new_fp);
        spin_unlock(&foo_mutex);
        synchronize_rcu();
        kfree(old_fp);
}

/*
 * Return the value of field "a" of the current gbl_foo
 * structure.  Use rcu_read_lock() and rcu_read_unlock()
 * to ensure that the structure does not get deleted out
 * from under us, and use rcu_dereference() to ensure that
 * we see the initialized version of the structure (important
 * for DEC Alpha and for people reading the code).
 */
int foo_get_a(void)
{
        int retval;

        rcu_read_lock();
        retval = rcu_dereference(gbl_foo)->a;
        rcu_read_unlock();
        return retval;
}
```

So, to sum up:

- Use rcu_read_lock() and rcu_read_unlock() to guard RCU read-side critical sections.
- Within an RCU read-side critical section, use rcu_dereference() to dereference RCU-protected pointers.
- Use some solid design (such as locks or semaphores) to keep concurrent updates from interfering with each other.
- Use rcu_assign_pointer() to update an RCU-protected pointer. This primitive protects concurrent readers from the updater, **not** concurrent updates from each other! You therefore still need to use locking (or something similar) to keep concurrent rcu_assign_pointer() primitives from interfering with each other.
- Use synchronize_rcu() **after** removing a data element from an RCU-protected data structure, but **before** reclaiming/freeing the data element, in order to wait for the completion of all RCU read-side critical sections that might be referencing that data item.

See *Review Checklist for RCU Patches* for additional rules to follow when using RCU. And again, more-typical uses of RCU may be found in *Using RCU to Protect Read-Mostly Linked Lists*, arrayRCU.rst, and *Using RCU to Protect Dynamic NMI Handlers*.

6.4 4. WHAT IF MY UPDATING THREAD CANNOT BLOCK?

In the example above, foo_update_a() blocks until a grace period elapses. This is quite simple, but in some cases one cannot afford to wait so long -- there might be other high-priority work to be done.

In such cases, one uses call_rcu() rather than synchronize_rcu(). The call_rcu() API is as follows:

```
void call_rcu(struct rcu_head *head, rcu_callback_t func);
```

This function invokes func(head) after a grace period has elapsed. This invocation might happen from either softirq or process context, so the function is not permitted to block. The foo struct needs to have an rcu_head structure added, perhaps as follows:

```
struct foo {
        int a;
        char b;
        long c;
        struct rcu_head rcu;
};
```

The foo_update_a() function might then be written as follows:

```
/*
 * Create a new struct foo that is the same as the one currently
 * pointed to by gbl_foo, except that field "a" is replaced
 * with "new_a".  Points gbl_foo to the new structure, and
 * frees up the old structure after a grace period.
 *
 * Uses rcu_assign_pointer() to ensure that concurrent readers
 * see the initialized version of the new structure.
```

```
 *
 * Uses call_rcu() to ensure that any readers that might have
 * references to the old structure complete before freeing the
 * old structure.
 */
void foo_update_a(int new_a)
{
        struct foo *new_fp;
        struct foo *old_fp;

        new_fp = kmalloc(sizeof(*new_fp), GFP_KERNEL);
        spin_lock(&foo_mutex);
        old_fp = rcu_dereference_protected(gbl_foo, lockdep_is_held(&foo_
→mutex));
        *new_fp = *old_fp;
        new_fp->a = new_a;
        rcu_assign_pointer(gbl_foo, new_fp);
        spin_unlock(&foo_mutex);
        call_rcu(&old_fp->rcu, foo_reclaim);
}
```

The foo_reclaim() function might appear as follows:

```
void foo_reclaim(struct rcu_head *rp)
{
        struct foo *fp = container_of(rp, struct foo, rcu);

        foo_cleanup(fp->a);

        kfree(fp);
}
```

The container_of() primitive is a macro that, given a pointer into a struct, the type of the struct, and the pointed-to field within the struct, returns a pointer to the beginning of the struct.

The use of call_rcu() permits the caller of foo_update_a() to immediately regain control, without needing to worry further about the old version of the newly updated element. It also clearly shows the RCU distinction between updater, namely foo_update_a(), and reclaimer, namely foo_reclaim().

The summary of advice is the same as for the previous section, except that we are now using call_rcu() rather than synchronize_rcu():

- Use call_rcu() **after** removing a data element from an RCU-protected data structure in order to register a callback function that will be invoked after the completion of all RCU read-side critical sections that might be referencing that data item.

If the callback for call_rcu() is not doing anything more than calling kfree() on the structure, you can use kfree_rcu() instead of call_rcu() to avoid having to write your own callback:

```
kfree_rcu(old_fp, rcu);
```

If the occasional sleep is permitted, the single-argument form may be used, omitting the rcu_head structure from struct foo.

> kfree_rcu_mightsleep(old_fp);

This variant almost never blocks, but might do so by invoking synchronize_rcu() in response to memory-allocation failure.

Again, see *Review Checklist for RCU Patches* for additional rules governing the use of RCU.

6.5 5. WHAT ARE SOME SIMPLE IMPLEMENTATIONS OF RCU?

One of the nice things about RCU is that it has extremely simple "toy" implementations that are a good first step towards understanding the production-quality implementations in the Linux kernel. This section presents two such "toy" implementations of RCU, one that is implemented in terms of familiar locking primitives, and another that more closely resembles "classic" RCU. Both are way too simple for real-world use, lacking both functionality and performance. However, they are useful in getting a feel for how RCU works. See kernel/rcu/update.c for a production-quality implementation, and see:

> https://docs.google.com/document/d/1X0lThx8OK0ZgLMqVoXiR4ZrGURHrXK6NyLRbeXe3Xac/edit

for papers describing the Linux kernel RCU implementation. The OLS'01 and OLS'02 papers are a good introduction, and the dissertation provides more details on the current implementation as of early 2004.

6.5.1 5A. "TOY" IMPLEMENTATION #1: LOCKING

This section presents a "toy" RCU implementation that is based on familiar locking primitives. Its overhead makes it a non-starter for real-life use, as does its lack of scalability. It is also unsuitable for realtime use, since it allows scheduling latency to "bleed" from one read-side critical section to another. It also assumes recursive reader-writer locks: If you try this with non-recursive locks, and you allow nested rcu_read_lock() calls, you can deadlock.

However, it is probably the easiest implementation to relate to, so is a good starting point.

It is extremely simple:

```
static DEFINE_RWLOCK(rcu_gp_mutex);

void rcu_read_lock(void)
{
        read_lock(&rcu_gp_mutex);
}

void rcu_read_unlock(void)
{
        read_unlock(&rcu_gp_mutex);
}

void synchronize_rcu(void)
{
        write_lock(&rcu_gp_mutex);
```

```
        smp_mb__after_spinlock();
        write_unlock(&rcu_gp_mutex);
}
```

[You can ignore rcu_assign_pointer() and rcu_dereference() without missing much. But here are simplified versions anyway. And whatever you do, don't forget about them when submitting patches making use of RCU!]:

```
#define rcu_assign_pointer(p, v) \
({ \
        smp_store_release(&(p), (v)); \
})

#define rcu_dereference(p) \
({ \
        typeof(p) _________p1 = READ_ONCE(p); \
        (_________p1); \
})
```

The rcu_read_lock() and rcu_read_unlock() primitive read-acquire and release a global reader-writer lock. The synchronize_rcu() primitive write-acquires this same lock, then releases it. This means that once synchronize_rcu() exits, all RCU read-side critical sections that were in progress before synchronize_rcu() was called are guaranteed to have completed -- there is no way that synchronize_rcu() would have been able to write-acquire the lock otherwise. The smp_mb__after_spinlock() promotes synchronize_rcu() to a full memory barrier in compliance with the "Memory-Barrier Guarantees" listed in:

> *A Tour Through RCU's Requirements*

It is possible to nest rcu_read_lock(), since reader-writer locks may be recursively acquired. Note also that rcu_read_lock() is immune from deadlock (an important property of RCU). The reason for this is that the only thing that can block rcu_read_lock() is a synchronize_rcu(). But synchronize_rcu() does not acquire any locks while holding rcu_gp_mutex, so there can be no deadlock cycle.

Quick Quiz #1:
: Why is this argument naive? How could a deadlock occur when using this algorithm in a real-world Linux kernel? How could this deadlock be avoided?

Answers to Quick Quiz

6.5.2 5B. "TOY" EXAMPLE #2: CLASSIC RCU

This section presents a "toy" RCU implementation that is based on "classic RCU". It is also short on performance (but only for updates) and on features such as hotplug CPU and the ability to run in CONFIG_PREEMPTION kernels. The definitions of rcu_dereference() and rcu_assign_pointer() are the same as those shown in the preceding section, so they are omitted.

```
void rcu_read_lock(void) { }

void rcu_read_unlock(void) { }
```

```
void synchronize_rcu(void)
{
        int cpu;

        for_each_possible_cpu(cpu)
                run_on(cpu);
}
```

Note that rcu_read_lock() and rcu_read_unlock() do absolutely nothing. This is the great strength of classic RCU in a non-preemptive kernel: read-side overhead is precisely zero, at least on non-Alpha CPUs. And there is absolutely no way that rcu_read_lock() can possibly participate in a deadlock cycle!

The implementation of synchronize_rcu() simply schedules itself on each CPU in turn. The run_on() primitive can be implemented straightforwardly in terms of the sched_setaffinity() primitive. Of course, a somewhat less "toy" implementation would restore the affinity upon completion rather than just leaving all tasks running on the last CPU, but when I said "toy", I meant **toy**!

So how the heck is this supposed to work???

Remember that it is illegal to block while in an RCU read-side critical section. Therefore, if a given CPU executes a context switch, we know that it must have completed all preceding RCU read-side critical sections. Once **all** CPUs have executed a context switch, then **all** preceding RCU read-side critical sections will have completed.

So, suppose that we remove a data item from its structure and then invoke synchronize_rcu(). Once synchronize_rcu() returns, we are guaranteed that there are no RCU read-side critical sections holding a reference to that data item, so we can safely reclaim it.

Quick Quiz #2:
Give an example where Classic RCU's read-side overhead is **negative**.

Answers to Quick Quiz

Quick Quiz #3:
If it is illegal to block in an RCU read-side critical section, what the heck do you do in CONFIG_PREEMPT_RT, where normal spinlocks can block???

Answers to Quick Quiz

6.6 6. ANALOGY WITH READER-WRITER LOCKING

Although RCU can be used in many different ways, a very common use of RCU is analogous to reader-writer locking. The following unified diff shows how closely related RCU and reader-writer locking can be.

```
@@ -5,5 +5,5 @@ struct el {
        int data;
        /* Other data fields */
 };
-rwlock_t listmutex;
+spinlock_t listmutex;
```

```
 struct el head;

@@ -13,15 +14,15 @@
        struct list_head *lp;
        struct el *p;

-       read_lock(&listmutex);
-       list_for_each_entry(p, head, lp) {
+       rcu_read_lock();
+       list_for_each_entry_rcu(p, head, lp) {
                if (p->key == key) {
                        *result = p->data;
-                       read_unlock(&listmutex);
+                       rcu_read_unlock();
                        return 1;
                }
        }
-       read_unlock(&listmutex);
+       rcu_read_unlock();
        return 0;
 }

@@ -29,15 +30,16 @@
 {
        struct el *p;

-       write_lock(&listmutex);
+       spin_lock(&listmutex);
        list_for_each_entry(p, head, lp) {
                if (p->key == key) {
-                       list_del(&p->list);
-                       write_unlock(&listmutex);
+                       list_del_rcu(&p->list);
+                       spin_unlock(&listmutex);
+                       synchronize_rcu();
                        kfree(p);
                        return 1;
                }
        }
-       write_unlock(&listmutex);
+       spin_unlock(&listmutex);
        return 0;
 }
```

Or, for those who prefer a side-by-side listing:

```
struct el {                          struct el {
  struct list_head list;               struct list_head list;
  long key;                            long key;
  spinlock_t mutex;                    spinlock_t mutex;
  int data;                            int data;
```

```
  /* Other data fields */
};
rwlock_t listmutex;
struct el head;
```

```
  /* Other data fields */
};
spinlock_t listmutex;
struct el head;
```

```
int search(long key, int *result)
{
  struct list_head *lp;
  struct el *p;

  read_lock(&listmutex);
  list_for_each_entry(p, head, lp) {
    if (p->key == key) {
      *result = p->data;
      read_unlock(&listmutex);
      return 1;
    }
  }
  read_unlock(&listmutex);
  return 0;
}
```

```
int search(long key, int *result)
{
  struct list_head *lp;
  struct el *p;

  rcu_read_lock();
  list_for_each_entry_rcu(p, head, lp) {
    if (p->key == key) {
      *result = p->data;
      rcu_read_unlock();
      return 1;
    }
  }
  rcu_read_unlock();
  return 0;
}
```

```
int delete(long key)
{
  struct el *p;

  write_lock(&listmutex);
  list_for_each_entry(p, head, lp) {
    if (p->key == key) {
      list_del(&p->list);
      write_unlock(&listmutex);

      kfree(p);
      return 1;
    }
  }
  write_unlock(&listmutex);
  return 0;
}
```

```
int delete(long key)
{
  struct el *p;

  spin_lock(&listmutex);
  list_for_each_entry(p, head, lp) {
    if (p->key == key) {
      list_del_rcu(&p->list);
      spin_unlock(&listmutex);
      synchronize_rcu();
      kfree(p);
      return 1;
    }
  }
  spin_unlock(&listmutex);
  return 0;
}
```

Either way, the differences are quite small. Read-side locking moves to rcu_read_lock() and rcu_read_unlock, update-side locking moves from a reader-writer lock to a simple spinlock, and a synchronize_rcu() precedes the kfree().

However, there is one potential catch: the read-side and update-side critical sections can now run concurrently. In many cases, this will not be a problem, but it is necessary to check carefully regardless. For example, if multiple independent list updates must be seen as a single atomic update, converting to RCU will require special care.

Also, the presence of synchronize_rcu() means that the RCU version of delete() can now block.

If this is a problem, there is a callback-based mechanism that never blocks, namely call_rcu() or kfree_rcu(), that can be used in place of synchronize_rcu().

6.7 7. ANALOGY WITH REFERENCE COUNTING

The reader-writer analogy (illustrated by the previous section) is not always the best way to think about using RCU. Another helpful analogy considers RCU an effective reference count on everything which is protected by RCU.

A reference count typically does not prevent the referenced object's values from changing, but does prevent changes to type -- particularly the gross change of type that happens when that object's memory is freed and re-allocated for some other purpose. Once a type-safe reference to the object is obtained, some other mechanism is needed to ensure consistent access to the data in the object. This could involve taking a spinlock, but with RCU the typical approach is to perform reads with SMP-aware operations such as smp_load_acquire(), to perform updates with atomic read-modify-write operations, and to provide the necessary ordering. RCU provides a number of support functions that embed the required operations and ordering, such as the list_for_each_entry_rcu() macro used in the previous section.

A more focused view of the reference counting behavior is that, between rcu_read_lock() and rcu_read_unlock(), any reference taken with rcu_dereference() on a pointer marked as `__rcu` can be treated as though a reference-count on that object has been temporarily increased. This prevents the object from changing type. Exactly what this means will depend on normal expectations of objects of that type, but it typically includes that spinlocks can still be safely locked, normal reference counters can be safely manipulated, and `__rcu` pointers can be safely dereferenced.

Some operations that one might expect to see on an object for which an RCU reference is held include:

- Copying out data that is guaranteed to be stable by the object's type.
- Using kref_get_unless_zero() or similar to get a longer-term reference. This may fail of course.
- Acquiring a spinlock in the object, and checking if the object still is the expected object and if so, manipulating it freely.

The understanding that RCU provides a reference that only prevents a change of type is particularly visible with objects allocated from a slab cache marked `SLAB_TYPESAFE_BY_RCU`. RCU operations may yield a reference to an object from such a cache that has been concurrently freed and the memory reallocated to a completely different object, though of the same type. In this case RCU doesn't even protect the identity of the object from changing, only its type. So the object found may not be the one expected, but it will be one where it is safe to take a reference (and then potentially acquiring a spinlock), allowing subsequent code to check whether the identity matches expectations. It is tempting to simply acquire the spinlock without first taking the reference, but unfortunately any spinlock in a `SLAB_TYPESAFE_BY_RCU` object must be initialized after each and every call to kmem_cache_alloc(), which renders reference-free spinlock acquisition completely unsafe. Therefore, when using `SLAB_TYPESAFE_BY_RCU`, make proper use of a reference counter. (Those willing to use a kmem_cache constructor may also use locking, including cache-friendly sequence locking.)

With traditional reference counting -- such as that implemented by the kref library in Linux -- there is typically code that runs when the last reference to an object is dropped. With kref, this is the function passed to kref_put(). When RCU is being used, such finalization code must not be run until all __rcu pointers referencing the object have been updated, and then a grace period has passed. Every remaining globally visible pointer to the object must be considered to be a potential counted reference, and the finalization code is typically run using call_rcu() only after all those pointers have been changed.

To see how to choose between these two analogies -- of RCU as a reader-writer lock and RCU as a reference counting system -- it is useful to reflect on the scale of the thing being protected. The reader-writer lock analogy looks at larger multi-part objects such as a linked list and shows how RCU can facilitate concurrency while elements are added to, and removed from, the list. The reference-count analogy looks at the individual objects and looks at how they can be accessed safely within whatever whole they are a part of.

6.8 8. FULL LIST OF RCU APIs

The RCU APIs are documented in docbook-format header comments in the Linux-kernel source code, but it helps to have a full list of the APIs, since there does not appear to be a way to categorize them in docbook. Here is the list, by category.

RCU list traversal:

```
list_entry_rcu
list_entry_lockless
list_first_entry_rcu
list_next_rcu
list_for_each_entry_rcu
list_for_each_entry_continue_rcu
list_for_each_entry_from_rcu
list_first_or_null_rcu
list_next_or_null_rcu
hlist_first_rcu
hlist_next_rcu
hlist_pprev_rcu
hlist_for_each_entry_rcu
hlist_for_each_entry_rcu_bh
hlist_for_each_entry_from_rcu
hlist_for_each_entry_continue_rcu
hlist_for_each_entry_continue_rcu_bh
hlist_nulls_first_rcu
hlist_nulls_for_each_entry_rcu
hlist_bl_first_rcu
hlist_bl_for_each_entry_rcu
```

RCU pointer/list update:

```
rcu_assign_pointer
list_add_rcu
list_add_tail_rcu
list_del_rcu
list_replace_rcu
hlist_add_behind_rcu
```

```
hlist_add_before_rcu
hlist_add_head_rcu
hlist_add_tail_rcu
hlist_del_rcu
hlist_del_init_rcu
hlist_replace_rcu
list_splice_init_rcu
list_splice_tail_init_rcu
hlist_nulls_del_init_rcu
hlist_nulls_del_rcu
hlist_nulls_add_head_rcu
hlist_bl_add_head_rcu
hlist_bl_del_init_rcu
hlist_bl_del_rcu
hlist_bl_set_first_rcu
```

RCU:

```
Critical sections       Grace period            Barrier

rcu_read_lock           synchronize_net         rcu_barrier
rcu_read_unlock         synchronize_rcu
rcu_dereference         synchronize_rcu_expedited
rcu_read_lock_held      call_rcu
rcu_dereference_check   kfree_rcu
rcu_dereference_protected
```

bh:

```
Critical sections       Grace period            Barrier

rcu_read_lock_bh        call_rcu                rcu_barrier
rcu_read_unlock_bh      synchronize_rcu
[local_bh_disable]      synchronize_rcu_expedited
[and friends]
rcu_dereference_bh
rcu_dereference_bh_check
rcu_dereference_bh_protected
rcu_read_lock_bh_held
```

sched:

```
Critical sections       Grace period            Barrier

rcu_read_lock_sched     call_rcu                rcu_barrier
rcu_read_unlock_sched   synchronize_rcu
[preempt_disable]       synchronize_rcu_expedited
[and friends]
rcu_read_lock_sched_notrace
rcu_read_unlock_sched_notrace
rcu_dereference_sched
```

```
rcu_dereference_sched_check
rcu_dereference_sched_protected
rcu_read_lock_sched_held
```

RCU-Tasks:

```
Critical sections       Grace period            Barrier

N/A                     call_rcu_tasks          rcu_barrier_tasks
                        synchronize_rcu_tasks
```

RCU-Tasks-Rude:

```
Critical sections       Grace period            Barrier

N/A                     call_rcu_tasks_rude     rcu_barrier_tasks_rude
                        synchronize_rcu_tasks_rude
```

RCU-Tasks-Trace:

```
Critical sections       Grace period            Barrier

rcu_read_lock_trace     call_rcu_tasks_trace    rcu_barrier_tasks_trace
rcu_read_unlock_trace   synchronize_rcu_tasks_trace
```

SRCU:

```
Critical sections       Grace period            Barrier

srcu_read_lock          call_srcu               srcu_barrier
srcu_read_unlock        synchronize_srcu
srcu_dereference        synchronize_srcu_expedited
srcu_dereference_check
srcu_read_lock_held
```

SRCU: Initialization/cleanup:

```
DEFINE_SRCU
DEFINE_STATIC_SRCU
init_srcu_struct
cleanup_srcu_struct
```

All: lockdep-checked RCU utility APIs:

```
RCU_LOCKDEP_WARN
rcu_sleep_check
```

All: Unchecked RCU-protected pointer access:

```
rcu_dereference_raw
```

All: Unchecked RCU-protected pointer access with dereferencing prohibited:

```
rcu_access_pointer
```

See the comment headers in the source code (or the docbook generated from them) for more information.

However, given that there are no fewer than four families of RCU APIs in the Linux kernel, how do you choose which one to use? The following list can be helpful:

a. Will readers need to block? If so, you need SRCU.

b. Will readers need to block and are you doing tracing, for example, ftrace or BPF? If so, you need RCU-tasks, RCU-tasks-rude, and/or RCU-tasks-trace.

c. What about the -rt patchset? If readers would need to block in an non-rt kernel, you need SRCU. If readers would block when acquiring spinlocks in a -rt kernel, but not in a non-rt kernel, SRCU is not necessary. (The -rt patchset turns spinlocks into sleeplocks, hence this distinction.)

d. Do you need to treat NMI handlers, hardirq handlers, and code segments with preemption disabled (whether via preempt_disable(), local_irq_save(), local_bh_disable(), or some other mechanism) as if they were explicit RCU readers? If so, RCU-sched readers are the only choice that will work for you, but since about v4.20 you use can use the vanilla RCU update primitives.

e. Do you need RCU grace periods to complete even in the face of softirq monopolization of one or more of the CPUs? For example, is your code subject to network-based denial-of-service attacks? If so, you should disable softirq across your readers, for example, by using rcu_read_lock_bh(). Since about v4.20 you use can use the vanilla RCU update primitives.

f. Is your workload too update-intensive for normal use of RCU, but inappropriate for other synchronization mechanisms? If so, consider SLAB_TYPESAFE_BY_RCU (which was originally named SLAB_DESTROY_BY_RCU). But please be careful!

g. Do you need read-side critical sections that are respected even on CPUs that are deep in the idle loop, during entry to or exit from user-mode execution, or on an offlined CPU? If so, SRCU and RCU Tasks Trace are the only choices that will work for you, with SRCU being strongly preferred in almost all cases.

h. Otherwise, use RCU.

Of course, this all assumes that you have determined that RCU is in fact the right tool for your job.

6.9 9. ANSWERS TO QUICK QUIZZES

Quick Quiz #1:
Why is this argument naive? How could a deadlock occur when using this algorithm in a real-world Linux kernel? [Referring to the lock-based “toy” RCU algorithm.]

Answer:
Consider the following sequence of events:

1. CPU 0 acquires some unrelated lock, call it “problematic_lock”, disabling irq via spin_lock_irqsave().
2. CPU 1 enters synchronize_rcu(), write-acquiring rcu_gp_mutex.

3. CPU 0 enters rcu_read_lock(), but must wait because CPU 1 holds rcu_gp_mutex.
4. CPU 1 is interrupted, and the irq handler attempts to acquire problematic_lock.

The system is now deadlocked.

One way to avoid this deadlock is to use an approach like that of CONFIG_PREEMPT_RT, where all normal spinlocks become blocking locks, and all irq handlers execute in the context of special tasks. In this case, in step 4 above, the irq handler would block, allowing CPU 1 to release rcu_gp_mutex, avoiding the deadlock.

Even in the absence of deadlock, this RCU implementation allows latency to "bleed" from readers to other readers through synchronize_rcu(). To see this, consider task A in an RCU read-side critical section (thus read-holding rcu_gp_mutex), task B blocked attempting to write-acquire rcu_gp_mutex, and task C blocked in rcu_read_lock() attempting to read_acquire rcu_gp_mutex. Task A's RCU read-side latency is holding up task C, albeit indirectly via task B.

Realtime RCU implementations therefore use a counter-based approach where tasks in RCU read-side critical sections cannot be blocked by tasks executing synchronize_rcu().

Back to Quick Quiz #1

Quick Quiz #2:
Give an example where Classic RCU's read-side overhead is **negative**.

Answer:
Imagine a single-CPU system with a non-CONFIG_PREEMPTION kernel where a routing table is used by process-context code, but can be updated by irq-context code (for example, by an "ICMP REDIRECT" packet). The usual way of handling this would be to have the process-context code disable interrupts while searching the routing table. Use of RCU allows such interrupt-disabling to be dispensed with. Thus, without RCU, you pay the cost of disabling interrupts, and with RCU you don't.

One can argue that the overhead of RCU in this case is negative with respect to the single-CPU interrupt-disabling approach. Others might argue that the overhead of RCU is merely zero, and that replacing the positive overhead of the interrupt-disabling scheme with the zero-overhead RCU scheme does not constitute negative overhead.

In real life, of course, things are more complex. But even the theoretical possibility of negative overhead for a synchronization primitive is a bit unexpected. ;-)

Back to Quick Quiz #2

Quick Quiz #3:
If it is illegal to block in an RCU read-side critical section, what the heck do you do in CONFIG_PREEMPT_RT, where normal spinlocks can block???

Answer:
Just as CONFIG_PREEMPT_RT permits preemption of spinlock critical sections, it permits preemption of RCU read-side critical sections. It also permits spinlocks blocking while in RCU read-side critical sections.

Why the apparent inconsistency? Because it is possible to use priority boosting to keep the RCU grace periods short if need be (for example, if running short of memory). In contrast, if blocking waiting for (say) network reception, there is no way to know what should be boosted. Especially given that the process we need to boost might well be a human being who just went out for a pizza or something. And although a computer-operated cattle prod might arouse serious interest, it might also provoke serious objections. Besides, how does the computer know what pizza parlor the human being went to???

Back to Quick Quiz #3

ACKNOWLEDGEMENTS

My thanks to the people who helped make this human-readable, including Jon Walpole, Josh Triplett, Serge Hallyn, Suzanne Wood, and Alan Stern.

For more information, see http://www.rdrop.com/users/paulmck/RCU.

CHAPTER

SEVEN

RCU CONCEPTS

The basic idea behind RCU (read-copy update) is to split destructive operations into two parts, one that prevents anyone from seeing the data item being destroyed, and one that actually carries out the destruction. A "grace period" must elapse between the two parts, and this grace period must be long enough that any readers accessing the item being deleted have since dropped their references. For example, an RCU-protected deletion from a linked list would first remove the item from the list, wait for a grace period to elapse, then free the element. See *Using RCU to Protect Read-Mostly Linked Lists* for more information on using RCU with linked lists.

7.1 Frequently Asked Questions

- Why would anyone want to use RCU?

  The advantage of RCU's two-part approach is that RCU readers need not acquire any locks, perform any atomic instructions, write to shared memory, or (on CPUs other than Alpha) execute any memory barriers. The fact that these operations are quite expensive on modern CPUs is what gives RCU its performance advantages in read-mostly situations. The fact that RCU readers need not acquire locks can also greatly simplify deadlock-avoidance code.

- How can the updater tell when a grace period has completed if the RCU readers give no indication when they are done?

  Just as with spinlocks, RCU readers are not permitted to block, switch to user-mode execution, or enter the idle loop. Therefore, as soon as a CPU is seen passing through any of these three states, we know that that CPU has exited any previous RCU read-side critical sections. So, if we remove an item from a linked list, and then wait until all CPUs have switched context, executed in user mode, or executed in the idle loop, we can safely free up that item.

  Preemptible variants of RCU (CONFIG_PREEMPT_RCU) get the same effect, but require that the readers manipulate CPU-local counters. These counters allow limited types of blocking within RCU read-side critical sections. SRCU also uses CPU-local counters, and permits general blocking within RCU read-side critical sections. These variants of RCU detect grace periods by sampling these counters.

- If I am running on a uniprocessor kernel, which can only do one thing at a time, why should I wait for a grace period?

  See *RCU on Uniprocessor Systems* for more information.

- How can I see where RCU is currently used in the Linux kernel?

Search for "rcu_read_lock", "rcu_read_unlock", "call_rcu", "rcu_read_lock_bh", "rcu_read_unlock_bh", "srcu_read_lock", "srcu_read_unlock", "synchronize_rcu", "synchronize_net", "synchronize_srcu", and the other RCU primitives. Or grab one of the cscope databases from:

(http://www.rdrop.com/users/paulmck/RCU/linuxusage/rculocktab.html).

- What guidelines should I follow when writing code that uses RCU?

  See *Review Checklist for RCU Patches*.

- Why the name "RCU"?

  "RCU" stands for "read-copy update". *Using RCU to Protect Read-Mostly Linked Lists* has more information on where this name came from, search for "read-copy update" to find it.

- I hear that RCU is patented? What is with that?

  Yes, it is. There are several known patents related to RCU, search for the string "Patent" in Documentation/RCU/RTFP.txt to find them. Of these, one was allowed to lapse by the assignee, and the others have been contributed to the Linux kernel under GPL. Many (but not all) have long since expired. There are now also LGPL implementations of user-level RCU available (https://liburcu.org/).

- I hear that RCU needs work in order to support realtime kernels?

  Realtime-friendly RCU are enabled via the CONFIG_PREEMPTION kernel configuration parameter.

- Where can I find more information on RCU?

  See the Documentation/RCU/RTFP.txt file. Or point your browser at (https://docs.google.com/document/d/1X0lThx8OK0ZgLMqVoXiR4ZrGURHrXK6NyLRbeXe3Xac/edit) or (https://docs.google.com/document/d/1GCdQC8SDbb54W1shjEXqGZ0Rq8a6kIeYutdSIajfpLA/edit?usp=sharing).

CHAPTER

EIGHT

USING RCU HLIST_NULLS TO PROTECT LIST AND OBJECTS

This section describes how to use hlist_nulls to protect read-mostly linked lists and objects using SLAB_TYPESAFE_BY_RCU allocations.

Please read the basics in *Using RCU to Protect Read-Mostly Linked Lists*.

8.1 Using 'nulls'

Using special makers (called 'nulls') is a convenient way to solve following problem.

Without 'nulls', a typical RCU linked list managing objects which are allocated with SLAB_TYPESAFE_BY_RCU kmem_cache can use the following algorithms. Following examples assume 'obj' is a pointer to such objects, which is having below type.

```
struct object {
  struct hlist_node obj_node;
  atomic_t refcnt;
  unsigned int key;
};
```

8.1.1 1) Lookup algorithm

```
begin:
rcu_read_lock();
obj = lockless_lookup(key);
if (obj) {
  if (!try_get_ref(obj)) { // might fail for free objects
    rcu_read_unlock();
    goto begin;
  }
  /*
  * Because a writer could delete object, and a writer could
  * reuse these object before the RCU grace period, we
  * must check key after getting the reference on object
  */
  if (obj->key != key) { // not the object we expected
    put_ref(obj);
    rcu_read_unlock();
    goto begin;
```

```
  }
}
rcu_read_unlock();
```

Beware that lockless_lookup(key) cannot use traditional hlist_for_each_entry_rcu() but a version with an additional memory barrier (smp_rmb())

```
lockless_lookup(key)
{
  struct hlist_node *node, *next;
  for (pos = rcu_dereference((head)->first);
       pos && ({ next = pos->next; smp_rmb(); prefetch(next); 1; }) &&
       ({ obj = hlist_entry(pos, typeof(*obj), obj_node); 1; });
       pos = rcu_dereference(next))
    if (obj->key == key)
      return obj;
  return NULL;
}
```

And note the traditional hlist_for_each_entry_rcu() misses this smp_rmb():

```
struct hlist_node *node;
for (pos = rcu_dereference((head)->first);
     pos && ({ prefetch(pos->next); 1; }) &&
     ({ obj = hlist_entry(pos, typeof(*obj), obj_node); 1; });
     pos = rcu_dereference(pos->next))
  if (obj->key == key)
    return obj;
return NULL;
```

Quoting Corey Minyard:

```
"If the object is moved from one list to another list in-between the
time the hash is calculated and the next field is accessed, and the
object has moved to the end of a new list, the traversal will not
complete properly on the list it should have, since the object will
be on the end of the new list and there's not a way to tell it's on a
new list and restart the list traversal. I think that this can be
solved by pre-fetching the "next" field (with proper barriers) before
checking the key."
```

8.1.2 2) Insertion algorithm

We need to make sure a reader cannot read the new 'obj->obj_node.next' value and previous value of 'obj->key'. Otherwise, an item could be deleted from a chain, and inserted into another chain. If new chain was empty before the move, 'next' pointer is NULL, and lockless reader can not detect the fact that it missed following items in original chain.

```
/*
 * Please note that new inserts are done at the head of list,
```

```
 * not in the middle or end.
 */
obj = kmem_cache_alloc(...);
lock_chain(); // typically a spin_lock()
obj->key = key;
atomic_set_release(&obj->refcnt, 1); // key before refcnt
hlist_add_head_rcu(&obj->obj_node, list);
unlock_chain(); // typically a spin_unlock()
```

8.1.3 3) Removal algorithm

Nothing special here, we can use a standard RCU hlist deletion. But thanks to SLAB_TYPESAFE_BY_RCU, beware a deleted object can be reused very very fast (before the end of RCU grace period)

```
if (put_last_reference_on(obj) {
  lock_chain(); // typically a spin_lock()
  hlist_del_init_rcu(&obj->obj_node);
  unlock_chain(); // typically a spin_unlock()
  kmem_cache_free(cachep, obj);
}
```

8.2 Avoiding extra smp_rmb()

With hlist_nulls we can avoid extra smp_rmb() in lockless_lookup().

For example, if we choose to store the slot number as the 'nulls' end-of-list marker for each slot of the hash table, we can detect a race (some writer did a delete and/or a move of an object to another chain) checking the final 'nulls' value if the lookup met the end of chain. If final 'nulls' value is not the slot number, then we must restart the lookup at the beginning. If the object was moved to the same chain, then the reader doesn't care: It might occasionally scan the list again without harm.

Note that using hlist_nulls means the type of 'obj_node' field of 'struct object' becomes 'struct hlist_nulls_node'.

8.2.1 1) lookup algorithm

```
head = &table[slot];
begin:
rcu_read_lock();
hlist_nulls_for_each_entry_rcu(obj, node, head, obj_node) {
  if (obj->key == key) {
    if (!try_get_ref(obj)) { // might fail for free objects
      rcu_read_unlock();
      goto begin;
```

```
    }
    if (obj->key != key) { // not the object we expected
      put_ref(obj);
      rcu_read_unlock();
      goto begin;
    }
    goto out;
  }
}

// If the nulls value we got at the end of this lookup is
// not the expected one, we must restart lookup.
// We probably met an item that was moved to another chain.
if (get_nulls_value(node) != slot) {
  put_ref(obj);
  rcu_read_unlock();
  goto begin;
}
obj = NULL;

out:
rcu_read_unlock();
```

8.2.2 2) Insert algorithm

Same to the above one, but uses hlist_nulls_add_head_rcu() instead of hlist_add_head_rcu().

```
/*
 * Please note that new inserts are done at the head of list,
 * not in the middle or end.
 */
obj = kmem_cache_alloc(cachep);
lock_chain(); // typically a spin_lock()
obj->key = key;
atomic_set_release(&obj->refcnt, 1); // key before refcnt
/*
 * insert obj in RCU way (readers might be traversing chain)
 */
hlist_nulls_add_head_rcu(&obj->obj_node, list);
unlock_chain(); // typically a spin_unlock()
```

CHAPTER

NINE

REFERENCE-COUNT DESIGN FOR ELEMENTS OF LISTS/ARRAYS PROTECTED BY RCU

Please note that the percpu-ref feature is likely your first stop if you need to combine reference counts and RCU. Please see include/linux/percpu-refcount.h for more information. However, in those unusual cases where percpu-ref would consume too much memory, please read on.

Reference counting on elements of lists which are protected by traditional reader/writer spinlocks or semaphores are straightforward:

CODE LISTING A:

```
1.                                      2.
add()                                   search_and_reference()
{                                       {
    alloc_object                            read_lock(&list_lock);
    ...                                     search_for_element
    atomic_set(&el->rc, 1);                 atomic_inc(&el->rc);
    write_lock(&list_lock);                  ...
    add_element                             read_unlock(&list_lock);
    ...                                     ...
    write_unlock(&list_lock);           }
}

3.                                      4.
release_referenced()                    delete()
{                                       {
    ...                                     write_lock(&list_lock);
    if(atomic_dec_and_test(&el->rc))        ...
        kfree(el);
    ...                                     remove_element
}                                           write_unlock(&list_lock);
                                            ...
                                            if (atomic_dec_and_test(&el->rc))
                                                kfree(el);
                                            ...
                                        }
```

If this list/array is made lock free using RCU as in changing the write_lock() in add() and delete() to spin_lock() and changing read_lock() in search_and_reference() to rcu_read_lock(), the atomic_inc() in search_and_reference() could potentially hold reference to an element which has already been deleted from the list/array. Use atomic_inc_not_zero() in this scenario as follows:

CODE LISTING B:

```
1.                                          2.
add()                                       search_and_reference()
{                                           {
    alloc_object                                rcu_read_lock();
    ...                                         search_for_element
    atomic_set(&el->rc, 1);                     if (!atomic_inc_not_zero(&el->rc))
 ↪{
    spin_lock(&list_lock);                          rcu_read_unlock();
                                                    return FAIL;
    add_element                                 }
    ...                                         ...
    spin_unlock(&list_lock);                    rcu_read_unlock();
}                                           }
3.                                          4.
release_referenced()                        delete()
{                                           {
    ...                                         spin_lock(&list_lock);
    if (atomic_dec_and_test(&el->rc))           ...
        call_rcu(&el->head, el_free);           remove_element
    ...                                         spin_unlock(&list_lock);
}                                               ...
                                                if (atomic_dec_and_test(&el->rc))
                                                    call_rcu(&el->head, el_free);
                                                ...
                                            }
```

Sometimes, a reference to the element needs to be obtained in the update (write) stream. In such cases, atomic_inc_not_zero() might be overkill, since we hold the update-side spinlock. One might instead use atomic_inc() in such cases.

It is not always convenient to deal with "FAIL" in the search_and_reference() code path. In such cases, the atomic_dec_and_test() may be moved from delete() to el_free() as follows:

CODE LISTING C:

```
1.                                          2.
add()                                       search_and_reference()
{                                           {
    alloc_object                                rcu_read_lock();
    ...                                         search_for_element
    atomic_set(&el->rc, 1);                     atomic_inc(&el->rc);
    spin_lock(&list_lock);                      ...

    add_element                                 rcu_read_unlock();
    ...                                     }
    spin_unlock(&list_lock);                4.
}                                           delete()
3.                                          {
```

```
release_referenced()                        spin_lock(&list_lock);
{                                           ...
    ...                                     remove_element
    if (atomic_dec_and_test(&el->rc))       spin_unlock(&list_lock);
        kfree(el);                          ...
    ...                                     call_rcu(&el->head, el_free);
}                                           ...
5.                                      }
void el_free(struct rcu_head *rhp)
{
    release_referenced();
}
```

The key point is that the initial reference added by add() is not removed until after a grace period has elapsed following removal. This means that search_and_reference() cannot find this element, which means that the value of el->rc cannot increase. Thus, once it reaches zero, there are no readers that can or ever will be able to reference the element. The element can therefore safely be freed. This in turn guarantees that if any reader finds the element, that reader may safely acquire a reference without checking the value of the reference counter.

A clear advantage of the RCU-based pattern in listing C over the one in listing B is that any call to search_and_reference() that locates a given object will succeed in obtaining a reference to that object, even given a concurrent invocation of delete() for that same object. Similarly, a clear advantage of both listings B and C over listing A is that a call to delete() is not delayed even if there are an arbitrarily large number of calls to search_and_reference() searching for the same object that delete() was invoked on. Instead, all that is delayed is the eventual invocation of kfree(), which is usually not a problem on modern computer systems, even the small ones.

In cases where delete() can sleep, synchronize_rcu() can be called from delete(), so that el_free() can be subsumed into delete as follows:

```
4.
delete()
{
    spin_lock(&list_lock);
    ...
    remove_element
    spin_unlock(&list_lock);
    ...
    synchronize_rcu();
    if (atomic_dec_and_test(&el->rc))
        kfree(el);
    ...
}
```

As additional examples in the kernel, the pattern in listing C is used by reference counting of struct pid, while the pattern in listing B is used by struct posix_acl.

CHAPTER

TEN

RCU TORTURE TEST OPERATION

10.1 CONFIG_RCU_TORTURE_TEST

The CONFIG_RCU_TORTURE_TEST config option is available for all RCU implementations. It creates an rcutorture kernel module that can be loaded to run a torture test. The test periodically outputs status messages via printk(), which can be examined via the dmesg command (perhaps grepping for "torture"). The test is started when the module is loaded, and stops when the module is unloaded.

Module parameters are prefixed by "rcutorture." in Documentation/admin-guide/kernel-parameters.txt.

10.2 Output

The statistics output is as follows:

```
rcu-torture:--- Start of test: nreaders=16 nfakewriters=4 stat_interval=30␣
↪verbose=0 test_no_idle_hz=1 shuffle_interval=3 stutter=5 irqreader=1 fqs_
↪duration=0 fqs_holdoff=0 fqs_stutter=3 test_boost=1/0 test_boost_interval=7␣
↪test_boost_duration=4
rcu-torture: rtc:           (null) ver: 155441 tfle: 0 rta: 155441 rtaf: 8884␣
↪rtf: 155440 rtmbe: 0 rtbe: 0 rtbke: 0 rtbre: 0 rtbf: 0 rtb: 0 nt: 3055767
rcu-torture: Reader Pipe:  727860534 34213 0 0 0 0 0 0 0 0 0
rcu-torture: Reader Batch:  727877838 17003 0 0 0 0 0 0 0 0 0
rcu-torture: Free-Block Circulation:  155440 155440 155440 155440 155440␣
↪155440 155440 155440 155440 155440 0
rcu-torture:--- End of test: SUCCESS: nreaders=16 nfakewriters=4 stat_
↪interval=30 verbose=0 test_no_idle_hz=1 shuffle_interval=3 stutter=5␣
↪irqreader=1 fqs_duration=0 fqs_holdoff=0 fqs_stutter=3 test_boost=1/0 test_
↪boost_interval=7 test_boost_duration=4
```

The command "dmesg | grep torture:" will extract this information on most systems. On more esoteric configurations, it may be necessary to use other commands to access the output of the printk()s used by the RCU torture test. The printk()s use KERN_ALERT, so they should be evident. ;-)

The first and last lines show the rcutorture module parameters, and the last line shows either "SUCCESS" or "FAILURE", based on rcutorture's automatic determination as to whether RCU operated correctly.

The entries are as follows:

- “rtc”: The hexadecimal address of the structure currently visible to readers.
- “ver”: The number of times since boot that the RCU writer task has changed the structure visible to readers.
- “tfle”: If non-zero, indicates that the “torture freelist” containing structures to be placed into the “rtc” area is empty. This condition is important, since it can fool you into thinking that RCU is working when it is not. :-/
- “rta”: Number of structures allocated from the torture freelist.
- “rtaf”: Number of allocations from the torture freelist that have failed due to the list being empty. It is not unusual for this to be non-zero, but it is bad for it to be a large fraction of the value indicated by “rta”.
- “rtf”: Number of frees into the torture freelist.
- “rtmbe”: A non-zero value indicates that rcutorture believes that rcu_assign_pointer() and rcu_dereference() are not working correctly. This value should be zero.
- “rtbe”: A non-zero value indicates that one of the rcu_barrier() family of functions is not working correctly.
- “rtbke”: rcutorture was unable to create the real-time kthreads used to force RCU priority inversion. This value should be zero.
- “rtbre”: Although rcutorture successfully created the kthreads used to force RCU priority inversion, it was unable to set them to the real-time priority level of 1. This value should be zero.
- “rtbf”: The number of times that RCU priority boosting failed to resolve RCU priority inversion.
- “rtb”: The number of times that rcutorture attempted to force an RCU priority inversion condition. If you are testing RCU priority boosting via the “test_boost” module parameter, this value should be non-zero.
- “nt”: The number of times rcutorture ran RCU read-side code from within a timer handler. This value should be non-zero only if you specified the “irqreader” module parameter.
- “Reader Pipe”: Histogram of “ages” of structures seen by readers. If any entries past the first two are non-zero, RCU is broken. And rcutorture prints the error flag string “!!!” to make sure you notice. The age of a newly allocated structure is zero, it becomes one when removed from reader visibility, and is incremented once per grace period subsequently -- and is freed after passing through (RCU_TORTURE_PIPE_LEN-2) grace periods.

  The output displayed above was taken from a correctly working RCU. If you want to see what it looks like when broken, break it yourself. ;-)
- “Reader Batch”: Another histogram of “ages” of structures seen by readers, but in terms of counter flips (or batches) rather than in terms of grace periods. The legal number of non-zero entries is again two. The reason for this separate view is that it is sometimes easier to get the third entry to show up in the “Reader Batch” list than in the “Reader Pipe” list.
- “Free-Block Circulation”: Shows the number of torture structures that have reached a given point in the pipeline. The first element should closely correspond to the number of structures allocated, the second to the number that have been removed from reader view,

and all but the last remaining to the corresponding number of passes through a grace period. The last entry should be zero, as it is only incremented if a torture structure's counter somehow gets incremented farther than it should.

Different implementations of RCU can provide implementation-specific additional information. For example, Tree SRCU provides the following additional line:

```
srcud-torture: Tree SRCU per-CPU(idx=0): 0(35,-21) 1(-4,24) 2(1,1) 3(-26,20)␣
↪4(28,-47) 5(-9,4) 6(-10,14) 7(-14,11) T(1,6)
```

This line shows the per-CPU counter state, in this case for Tree SRCU using a dynamically allocated srcu_struct (hence "srcud-" rather than "srcu-"). The numbers in parentheses are the values of the "old" and "current" counters for the corresponding CPU. The "idx" value maps the "old" and "current" values to the underlying array, and is useful for debugging. The final "T" entry contains the totals of the counters.

10.3 Usage on Specific Kernel Builds

It is sometimes desirable to torture RCU on a specific kernel build, for example, when preparing to put that kernel build into production. In that case, the kernel should be built with CONFIG_RCU_TORTURE_TEST=m so that the test can be started using modprobe and terminated using rmmod.

For example, the following script may be used to torture RCU:

```
#!/bin/sh

modprobe rcutorture
sleep 3600
rmmod rcutorture
dmesg | grep torture:
```

The output can be manually inspected for the error flag of "!!!". One could of course create a more elaborate script that automatically checked for such errors. The "rmmod" command forces a "SUCCESS", "FAILURE", or "RCU_HOTPLUG" indication to be printk()ed. The first two are self-explanatory, while the last indicates that while there were no RCU failures, CPU-hotplug problems were detected.

10.4 Usage on Mainline Kernels

When using rcutorture to test changes to RCU itself, it is often necessary to build a number of kernels in order to test that change across a broad range of combinations of the relevant Kconfig options and of the relevant kernel boot parameters. In this situation, use of modprobe and rmmod can be quite time-consuming and error-prone.

Therefore, the tools/testing/selftests/rcutorture/bin/kvm.sh script is available for mainline testing for x86, arm64, and powerpc. By default, it will run the series of tests specified by tools/testing/selftests/rcutorture/configs/rcu/CFLIST, with each test running for 30 minutes within a guest OS using a minimal userspace supplied by an automatically generated initrd. After the tests are complete, the resulting build products and console output are analyzed for errors and the results of the runs are summarized.

On larger systems, rcutorture testing can be accelerated by passing the --cpus argument to kvm.sh. For example, on a 64-CPU system, "--cpus 43" would use up to 43 CPUs to run tests concurrently, which as of v5.4 would complete all the scenarios in two batches, reducing the time to complete from about eight hours to about one hour (not counting the time to build the sixteen kernels). The "--dryrun sched" argument will not run tests, but rather tell you how the tests would be scheduled into batches. This can be useful when working out how many CPUs to specify in the --cpus argument.

Not all changes require that all scenarios be run. For example, a change to Tree SRCU might run only the SRCU-N and SRCU-P scenarios using the --configs argument to kvm.sh as follows: "--configs 'SRCU-N SRCU-P'". Large systems can run multiple copies of of the full set of scenarios, for example, a system with 448 hardware threads can run five instances of the full set concurrently. To make this happen:

```
kvm.sh --cpus 448 --configs '5*CFLIST'
```

Alternatively, such a system can run 56 concurrent instances of a single eight-CPU scenario:

```
kvm.sh --cpus 448 --configs '56*TREE04'
```

Or 28 concurrent instances of each of two eight-CPU scenarios:

```
kvm.sh --cpus 448 --configs '28*TREE03 28*TREE04'
```

Of course, each concurrent instance will use memory, which can be limited using the --memory argument, which defaults to 512M. Small values for memory may require disabling the callback-flooding tests using the --bootargs parameter discussed below.

Sometimes additional debugging is useful, and in such cases the --kconfig parameter to kvm.sh may be used, for example, `--kconfig 'CONFIG_RCU_EQS_DEBUG=y'`. In addition, there are the --gdb, --kasan, and --kcsan parameters. Note that --gdb limits you to one scenario per kvm.sh run and requires that you have another window open from which to run `gdb` as instructed by the script.

Kernel boot arguments can also be supplied, for example, to control rcutorture's module parameters. For example, to test a change to RCU's CPU stall-warning code, use "--bootargs 'rcutorture.stall_cpu=30'". This will of course result in the scripting reporting a failure, namely the resulting RCU CPU stall warning. As noted above, reducing memory may require disabling rcutorture's callback-flooding tests:

```
kvm.sh --cpus 448 --configs '56*TREE04' --memory 128M \
        --bootargs 'rcutorture.fwd_progress=0'
```

Sometimes all that is needed is a full set of kernel builds. This is what the --buildonly parameter does.

The --duration parameter can override the default run time of 30 minutes. For example, `--duration 2d` would run for two days, `--duration 3h` would run for three hours, `--duration 5m` would run for five minutes, and `--duration 45s` would run for 45 seconds. This last can be useful for tracking down rare boot-time failures.

Finally, the --trust-make parameter allows each kernel build to reuse what it can from the previous kernel build. Please note that without the --trust-make parameter, your tags files may be demolished.

There are additional more arcane arguments that are documented in the source code of the kvm.sh script.

If a run contains failures, the number of buildtime and runtime failures is listed at the end of the kvm.sh output, which you really should redirect to a file. The build products and console output of each run is kept in tools/testing/selftests/rcutorture/res in timestamped directories. A given directory can be supplied to kvm-find-errors.sh in order to have it cycle you through summaries of errors and full error logs. For example:

```
tools/testing/selftests/rcutorture/bin/kvm-find-errors.sh \
        tools/testing/selftests/rcutorture/res/2020.01.20-15.54.23
```

However, it is often more convenient to access the files directly. Files pertaining to all scenarios in a run reside in the top-level directory (2020.01.20-15.54.23 in the example above), while per-scenario files reside in a subdirectory named after the scenario (for example, "TREE04"). If a given scenario ran more than once (as in "--configs '56*TREE04'" above), the directories corresponding to the second and subsequent runs of that scenario include a sequence number, for example, "TREE04.2", "TREE04.3", and so on.

The most frequently used file in the top-level directory is testid.txt. If the test ran in a git repository, then this file contains the commit that was tested and any uncommitted changes in diff format.

The most frequently used files in each per-scenario-run directory are:

.config:
This file contains the Kconfig options.

Make.out:
This contains build output for a specific scenario.

console.log:
This contains the console output for a specific scenario. This file may be examined once the kernel has booted, but it might not exist if the build failed.

vmlinux:
This contains the kernel, which can be useful with tools like objdump and gdb.

A number of additional files are available, but are less frequently used. Many are intended for debugging of rcutorture itself or of its scripting.

As of v5.4, a successful run with the default set of scenarios produces the following summary at the end of the run on a 12-CPU system:

```
SRCU-N ------- 804233 GPs (148.932/s) [srcu: g10008272 f0x0 ]
SRCU-P ------- 202320 GPs (37.4667/s) [srcud: g1809476 f0x0 ]
SRCU-t ------- 1122086 GPs (207.794/s) [srcu: g0 f0x0 ]
SRCU-u ------- 1111285 GPs (205.794/s) [srcud: g1 f0x0 ]
TASKS01 ------- 19666 GPs (3.64185/s) [tasks: g0 f0x0 ]
TASKS02 ------- 20541 GPs (3.80389/s) [tasks: g0 f0x0 ]
TASKS03 ------- 19416 GPs (3.59556/s) [tasks: g0 f0x0 ]
TINY01 ------- 836134 GPs (154.84/s) [rcu: g0 f0x0 ] n_max_cbs: 34198
TINY02 ------- 850371 GPs (157.476/s) [rcu: g0 f0x0 ] n_max_cbs: 2631
TREE01 ------- 162625 GPs (30.1157/s) [rcu: g1124169 f0x0 ]
TREE02 ------- 333003 GPs (61.6672/s) [rcu: g2647753 f0x0 ] n_max_cbs: 35844
TREE03 ------- 306623 GPs (56.782/s) [rcu: g2975325 f0x0 ] n_max_cbs: 1496497
```

```
CPU count limited from 16 to 12
TREE04 ------- 246149 GPs (45.5831/s) [rcu: g1695737 f0x0 ] n_max_cbs: 434961
TREE05 ------- 314603 GPs (58.2598/s) [rcu: g2257741 f0x2 ] n_max_cbs: 193997
TREE07 ------- 167347 GPs (30.9902/s) [rcu: g1079021 f0x0 ] n_max_cbs: 478732
CPU count limited from 16 to 12
TREE09 ------- 752238 GPs (139.303/s) [rcu: g13075057 f0x0 ] n_max_cbs: 99011
```

10.5 Repeated Runs

Suppose that you are chasing down a rare boot-time failure. Although you could use kvm.sh, doing so will rebuild the kernel on each run. If you need (say) 1,000 runs to have confidence that you have fixed the bug, these pointless rebuilds can become extremely annoying.

This is why kvm-again.sh exists.

Suppose that a previous kvm.sh run left its output in this directory:

```
tools/testing/selftests/rcutorture/res/2022.11.03-11.26.28
```

Then this run can be re-run without rebuilding as follow:

> kvm-again.sh tools/testing/selftests/rcutorture/res/2022.11.03-11.26.28

A few of the original run's kvm.sh parameters may be overridden, perhaps most notably --duration and --bootargs. For example:

```
kvm-again.sh tools/testing/selftests/rcutorture/res/2022.11.03-11.26.28 \
        --duration 45s
```

would re-run the previous test, but for only 45 seconds, thus facilitating tracking down the aforementioned rare boot-time failure.

10.6 Distributed Runs

Although kvm.sh is quite useful, its testing is confined to a single system. It is not all that hard to use your favorite framework to cause (say) 5 instances of kvm.sh to run on your 5 systems, but this will very likely unnecessarily rebuild kernels. In addition, manually distributing the desired rcutorture scenarios across the available systems can be painstaking and error-prone.

And this is why the kvm-remote.sh script exists.

If you the following command works:

```
ssh system0 date
```

and if it also works for system1, system2, system3, system4, and system5, and all of these systems have 64 CPUs, you can type:

```
kvm-remote.sh "system0 system1 system2 system3 system4 system5" \
        --cpus 64 --duration 8h --configs "5*CFLIST"
```

This will build each default scenario's kernel on the local system, then spread each of five instances of each scenario over the systems listed, running each scenario for eight hours. At the end of the runs, the results will be gathered, recorded, and printed. Most of the parameters that kvm.sh will accept can be passed to kvm-remote.sh, but the list of systems must come first.

The kvm.sh `--dryrun scenarios` argument is useful for working out how many scenarios may be run in one batch across a group of systems.

You can also re-run a previous remote run in a manner similar to kvm.sh:

> **kvm-remote.sh "system0 system1 system2 system3 system4 system5"**
> tools/testing/selftests/rcutorture/res/2022.11.03-11.26.28-remote --duration 24h

In this case, most of the kvm-again.sh parameters may be supplied following the pathname of the old run-results directory.

CHAPTER

ELEVEN

USING RCU'S CPU STALL DETECTOR

This document first discusses what sorts of issues RCU's CPU stall detector can locate, and then discusses kernel parameters and Kconfig options that can be used to fine-tune the detector's operation. Finally, this document explains the stall detector's "splat" format.

11.1 What Causes RCU CPU Stall Warnings?

So your kernel printed an RCU CPU stall warning. The next question is "What caused it?" The following problems can result in RCU CPU stall warnings:

- A CPU looping in an RCU read-side critical section.
- A CPU looping with interrupts disabled.
- A CPU looping with preemption disabled.
- A CPU looping with bottom halves disabled.
- For !CONFIG_PREEMPTION kernels, a CPU looping anywhere in the kernel without potentially invoking schedule(). If the looping in the kernel is really expected and desirable behavior, you might need to add some calls to cond_resched().
- Booting Linux using a console connection that is too slow to keep up with the boot-time console-message rate. For example, a 115Kbaud serial console can be *way* too slow to keep up with boot-time message rates, and will frequently result in RCU CPU stall warning messages. Especially if you have added debug printk()s.
- Anything that prevents RCU's grace-period kthreads from running. This can result in the "All QSes seen" console-log message. This message will include information on when the kthread last ran and how often it should be expected to run. It can also result in the `rcu_.*kthread starved for` console-log message, which will include additional debugging information.
- A CPU-bound real-time task in a CONFIG_PREEMPTION kernel, which might happen to preempt a low-priority task in the middle of an RCU read-side critical section. This is especially damaging if that low-priority task is not permitted to run on any other CPU, in which case the next RCU grace period can never complete, which will eventually cause the system to run out of memory and hang. While the system is in the process of running itself out of memory, you might see stall-warning messages.
- A CPU-bound real-time task in a CONFIG_PREEMPT_RT kernel that is running at a higher priority than the RCU softirq threads. This will prevent RCU callbacks from ever being invoked, and in a CONFIG_PREEMPT_RCU kernel will further prevent RCU grace periods

from ever completing. Either way, the system will eventually run out of memory and hang. In the CONFIG_PREEMPT_RCU case, you might see stall-warning messages.

You can use the rcutree.kthread_prio kernel boot parameter to increase the scheduling priority of RCU's kthreads, which can help avoid this problem. However, please note that doing this can increase your system's context-switch rate and thus degrade performance.

- A periodic interrupt whose handler takes longer than the time interval between successive pairs of interrupts. This can prevent RCU's kthreads and softirq handlers from running. Note that certain high-overhead debugging options, for example the function_graph tracer, can result in interrupt handler taking considerably longer than normal, which can in turn result in RCU CPU stall warnings.
- Testing a workload on a fast system, tuning the stall-warning timeout down to just barely avoid RCU CPU stall warnings, and then running the same workload with the same stall-warning timeout on a slow system. Note that thermal throttling and on-demand governors can cause a single system to be sometimes fast and sometimes slow!
- A hardware or software issue shuts off the scheduler-clock interrupt on a CPU that is not in dyntick-idle mode. This problem really has happened, and seems to be most likely to result in RCU CPU stall warnings for CONFIG_NO_HZ_COMMON=n kernels.
- A hardware or software issue that prevents time-based wakeups from occurring. These issues can range from misconfigured or buggy timer hardware through bugs in the interrupt or exception path (whether hardware, firmware, or software) through bugs in Linux's timer subsystem through bugs in the scheduler, and, yes, even including bugs in RCU itself. It can also result in the `rcu_.*timer wakeup didn't happen for` console-log message, which will include additional debugging information.
- A low-level kernel issue that either fails to invoke one of the variants of rcu_eqs_enter(true), rcu_eqs_exit(true), ct_idle_enter(), ct_idle_exit(), ct_irq_enter(), or ct_irq_exit() on the one hand, or that invokes one of them too many times on the other. Historically, the most frequent issue has been an omission of either irq_enter() or irq_exit(), which in turn invoke ct_irq_enter() or ct_irq_exit(), respectively. Building your kernel with CONFIG_RCU_EQS_DEBUG=y can help track down these types of issues, which sometimes arise in architecture-specific code.
- A bug in the RCU implementation.
- A hardware failure. This is quite unlikely, but is not at all uncommon in large datacenter. In one memorable case some decades back, a CPU failed in a running system, becoming unresponsive, but not causing an immediate crash. This resulted in a series of RCU CPU stall warnings, eventually leading the realization that the CPU had failed.

The RCU, RCU-sched, RCU-tasks, and RCU-tasks-trace implementations have CPU stall warning. Note that SRCU does *not* have CPU stall warnings. Please note that RCU only detects CPU stalls when there is a grace period in progress. No grace period, no CPU stall warnings.

To diagnose the cause of the stall, inspect the stack traces. The offending function will usually be near the top of the stack. If you have a series of stall warnings from a single extended stall, comparing the stack traces can often help determine where the stall is occurring, which will usually be in the function nearest the top of that portion of the stack which remains the same from trace to trace. If you can reliably trigger the stall, ftrace can be quite helpful.

RCU bugs can often be debugged with the help of CONFIG_RCU_TRACE and with RCU's event tracing. For information on RCU's event tracing, see include/trace/events/rcu.h.

11.2 Fine-Tuning the RCU CPU Stall Detector

The rcuupdate.rcu_cpu_stall_suppress module parameter disables RCU's CPU stall detector, which detects conditions that unduly delay RCU grace periods. This module parameter enables CPU stall detection by default, but may be overridden via boot-time parameter or at runtime via sysfs. The stall detector's idea of what constitutes "unduly delayed" is controlled by a set of kernel configuration variables and cpp macros:

11.2.1 CONFIG_RCU_CPU_STALL_TIMEOUT

This kernel configuration parameter defines the period of time that RCU will wait from the beginning of a grace period until it issues an RCU CPU stall warning. This time period is normally 21 seconds.

This configuration parameter may be changed at runtime via the /sys/module/rcupdate/parameters/rcu_cpu_stall_timeout, however this parameter is checked only at the beginning of a cycle. So if you are 10 seconds into a 40-second stall, setting this sysfs parameter to (say) five will shorten the timeout for the *next* stall, or the following warning for the current stall (assuming the stall lasts long enough). It will not affect the timing of the next warning for the current stall.

Stall-warning messages may be enabled and disabled completely via /sys/module/rcupdate/parameters/rcu_cpu_stall_suppress.

11.2.2 CONFIG_RCU_EXP_CPU_STALL_TIMEOUT

Same as the CONFIG_RCU_CPU_STALL_TIMEOUT parameter but only for the expedited grace period. This parameter defines the period of time that RCU will wait from the beginning of an expedited grace period until it issues an RCU CPU stall warning. This time period is normally 20 milliseconds on Android devices. A zero value causes the CONFIG_RCU_CPU_STALL_TIMEOUT value to be used, after conversion to milliseconds.

This configuration parameter may be changed at runtime via the /sys/module/rcupdate/parameters/rcu_exp_cpu_stall_timeout, however this parameter is checked only at the beginning of a cycle. If you are in a current stall cycle, setting it to a new value will change the timeout for the -next- stall.

Stall-warning messages may be enabled and disabled completely via /sys/module/rcupdate/parameters/rcu_cpu_stall_suppress.

11.2.3 RCU_STALL_DELAY_DELTA

Although the lockdep facility is extremely useful, it does add some overhead. Therefore, under CONFIG_PROVE_RCU, the RCU_STALL_DELAY_DELTA macro allows five extra seconds before giving an RCU CPU stall warning message. (This is a cpp macro, not a kernel configuration parameter.)

11.2.4 RCU_STALL_RAT_DELAY

The CPU stall detector tries to make the offending CPU print its own warnings, as this often gives better-quality stack traces. However, if the offending CPU does not detect its own stall in the number of jiffies specified by RCU_STALL_RAT_DELAY, then some other CPU will complain. This delay is normally set to two jiffies. (This is a cpp macro, not a kernel configuration parameter.)

11.2.5 rcupdate.rcu_task_stall_timeout

This boot/sysfs parameter controls the RCU-tasks and RCU-tasks-trace stall warning intervals. A value of zero or less suppresses RCU-tasks stall warnings. A positive value sets the stall-warning interval in seconds. An RCU-tasks stall warning starts with the line:

INFO: rcu_tasks detected stalls on tasks:

And continues with the output of sched_show_task() for each task stalling the current RCU-tasks grace period.

An RCU-tasks-trace stall warning starts (and continues) similarly:

INFO: rcu_tasks_trace detected stalls on tasks

11.3 Interpreting RCU's CPU Stall-Detector "Splats"

For non-RCU-tasks flavors of RCU, when a CPU detects that some other CPU is stalling, it will print a message similar to the following:

```
INFO: rcu_sched detected stalls on CPUs/tasks:
2-...: (3 GPs behind) idle=06c/0/0 softirq=1453/1455 fqs=0
16-...: (0 ticks this GP) idle=81c/0/0 softirq=764/764 fqs=0
(detected by 32, t=2603 jiffies, g=7075, q=625)
```

This message indicates that CPU 32 detected that CPUs 2 and 16 were both causing stalls, and that the stall was affecting RCU-sched. This message will normally be followed by stack dumps for each CPU. Please note that PREEMPT_RCU builds can be stalled by tasks as well as by CPUs, and that the tasks will be indicated by PID, for example, "P3421". It is even possible for an rcu_state stall to be caused by both CPUs *and* tasks, in which case the offending CPUs and tasks will all be called out in the list. In some cases, CPUs will detect themselves stalling, which will result in a self-detected stall.

CPU 2's "(3 GPs behind)" indicates that this CPU has not interacted with the RCU core for the past three grace periods. In contrast, CPU 16's "(0 ticks this GP)" indicates that this CPU has not taken any scheduling-clock interrupts during the current stalled grace period.

The "idle=" portion of the message prints the dyntick-idle state. The hex number before the first "/" is the low-order 12 bits of the dynticks counter, which will have an even-numbered value if the CPU is in dyntick-idle mode and an odd-numbered value otherwise. The hex number between the two "/"s is the value of the nesting, which will be a small non-negative number if in the idle loop (as shown above) and a very large positive number otherwise. The number following the final "/" is the NMI nesting, which will be a small non-negative number.

The “softirq=” portion of the message tracks the number of RCU softirq handlers that the stalled CPU has executed. The number before the “/” is the number that had executed since boot at the time that this CPU last noted the beginning of a grace period, which might be the current (stalled) grace period, or it might be some earlier grace period (for example, if the CPU might have been in dyntick-idle mode for an extended time period). The number after the “/” is the number that have executed since boot until the current time. If this latter number stays constant across repeated stall-warning messages, it is possible that RCU’s softirq handlers are no longer able to execute on this CPU. This can happen if the stalled CPU is spinning with interrupts are disabled, or, in -rt kernels, if a high-priority process is starving RCU’s softirq handler.

The “fqs=” shows the number of force-quiescent-state idle/offline detection passes that the grace-period kthread has made across this CPU since the last time that this CPU noted the beginning of a grace period.

The “detected by” line indicates which CPU detected the stall (in this case, CPU 32), how many jiffies have elapsed since the start of the grace period (in this case 2603), the grace-period sequence number (7075), and an estimate of the total number of RCU callbacks queued across all CPUs (625 in this case).

If the grace period ends just as the stall warning starts printing, there will be a spurious stall-warning message, which will include the following:

```
INFO: Stall ended before state dump start
```

This is rare, but does happen from time to time in real life. It is also possible for a zero-jiffy stall to be flagged in this case, depending on how the stall warning and the grace-period initialization happen to interact. Please note that it is not possible to entirely eliminate this sort of false positive without resorting to things like stop_machine(), which is overkill for this sort of problem.

If all CPUs and tasks have passed through quiescent states, but the grace period has nevertheless failed to end, the stall-warning splat will include something like the following:

```
All QSes seen, last rcu_preempt kthread activity 23807 (4297905177-4297881370),
↪ jiffies_till_next_fqs=3, root ->qsmask 0x0
```

The “23807” indicates that it has been more than 23 thousand jiffies since the grace-period kthread ran. The “jiffies_till_next_fqs” indicates how frequently that kthread should run, giving the number of jiffies between force-quiescent-state scans, in this case three, which is way less than 23807. Finally, the root rcu_node structure’s ->qsmask field is printed, which will normally be zero.

If the relevant grace-period kthread has been unable to run prior to the stall warning, as was the case in the “All QSes seen” line above, the following additional line is printed:

```
rcu_sched kthread starved for 23807 jiffies! g7075 f0x0 RCU_GP_WAIT_FQS(3) ->
↪state=0x1 ->cpu=5
Unless rcu_sched kthread gets sufficient CPU time, OOM is now expected␣
↪behavior.
```

Starving the grace-period kthreads of CPU time can of course result in RCU CPU stall warnings even when all CPUs and tasks have passed through the required quiescent states. The “g” number shows the current grace-period sequence number, the “f” precedes the ->gp_flags command to the grace-period kthread, the “RCU_GP_WAIT_FQS” indicates that the kthread is waiting for a short timeout, the "state" precedes value of the task_struct ->state field, and the "cpu" indicates that the grace-period kthread last ran on CPU 5.

If the relevant grace-period kthread does not wake from FQS wait in a reasonable time, then the following additional line is printed:

```
kthread timer wakeup didn't happen for 23804 jiffies! g7076 f0x0 RCU_GP_WAIT_
↪FQS(5) ->state=0x402
```

The "23804" indicates that kthread's timer expired more than 23 thousand jiffies ago. The rest of the line has meaning similar to the kthread starvation case.

Additionally, the following line is printed:

```
Possible timer handling issue on cpu=4 timer-softirq=11142
```

Here "cpu" indicates that the grace-period kthread last ran on CPU 4, where it queued the fqs timer. The number following the "timer-softirq" is the current `TIMER_SOFTIRQ` count on cpu 4. If this value does not change on successive RCU CPU stall warnings, there is further reason to suspect a timer problem.

These messages are usually followed by stack dumps of the CPUs and tasks involved in the stall. These stack traces can help you locate the cause of the stall, keeping in mind that the CPU detecting the stall will have an interrupt frame that is mainly devoted to detecting the stall.

11.4 Multiple Warnings From One Stall

If a stall lasts long enough, multiple stall-warning messages will be printed for it. The second and subsequent messages are printed at longer intervals, so that the time between (say) the first and second message will be about three times the interval between the beginning of the stall and the first message. It can be helpful to compare the stack dumps for the different messages for the same stalled grace period.

11.5 Stall Warnings for Expedited Grace Periods

If an expedited grace period detects a stall, it will place a message like the following in dmesg:

```
INFO: rcu_sched detected expedited stalls on CPUs/tasks: { 7-... } 21119␣
↪jiffies s: 73 root: 0x2/.
```

This indicates that CPU 7 has failed to respond to a reschedule IPI. The three periods (".") following the CPU number indicate that the CPU is online (otherwise the first period would instead have been "O"), that the CPU was online at the beginning of the expedited grace period (otherwise the second period would have instead been "o"), and that the CPU has been online at least once since boot (otherwise, the third period would instead have been "N"). The number before the "jiffies" indicates that the expedited grace period has been going on for 21,119 jiffies. The number following the "s:" indicates that the expedited grace-period sequence counter is 73. The fact that this last value is odd indicates that an expedited grace period is in flight. The number following "root:" is a bitmask that indicates which children of the root rcu_node structure correspond to CPUs and/or tasks that are blocking the current expedited grace period.

If the tree had more than one level, additional hex numbers would be printed for the states of the other rcu_node structures in the tree.

As with normal grace periods, PREEMPT_RCU builds can be stalled by tasks as well as by CPUs, and that the tasks will be indicated by PID, for example, "P3421".

It is entirely possible to see stall warnings from normal and from expedited grace periods at about the same time during the same run.

11.6 RCU_CPU_STALL_CPUTIME

In kernels built with CONFIG_RCU_CPU_STALL_CPUTIME=y or booted with rcupdate.rcu_cpu_stall_cputime=1, the following additional information is supplied with each RCU CPU stall warning:

```
rcu:          hardirqs   softirqs   csw/system
rcu:  number:      624         45            0
rcu: cputime:       69          1         2425   ==> 2500(ms)
```

These statistics are collected during the sampling period. The values in row "number:" are the number of hard interrupts, number of soft interrupts, and number of context switches on the stalled CPU. The first three values in row "cputime:" indicate the CPU time in milliseconds consumed by hard interrupts, soft interrupts, and tasks on the stalled CPU. The last number is the measurement interval, again in milliseconds. Because user-mode tasks normally do not cause RCU CPU stalls, these tasks are typically kernel tasks, which is why only the system CPU time are considered.

The sampling period is shown as follows:

```
|<------------first timeout---------->|<-----second timeout----->|
|<--half timeout-->|<--half timeout-->|                          |
|                  |<--first period-->|                          |
|                  |<-----------second sampling period---------->|
|                  |                  |                          |
           snapshot time point    1st-stall                  2nd-stall
```

The following describes four typical scenarios:

1. A CPU looping with interrupts disabled.

   ```
   rcu:          hardirqs   softirqs   csw/system
   rcu:  number:        0          0            0
   rcu: cputime:        0          0            0   ==> 2500(ms)
   ```

   Because interrupts have been disabled throughout the measurement interval, there are no interrupts and no context switches. Furthermore, because CPU time consumption was measured using interrupt handlers, the system CPU consumption is misleadingly measured as zero. This scenario will normally also have "(0 ticks this GP)" printed on this CPU's summary line.

2. A CPU looping with bottom halves disabled.

   This is similar to the previous example, but with non-zero number of and CPU time consumed by hard interrupts, along with non-zero CPU time consumed by in-kernel execution:

```
rcu:          hardirqs   softirqs   csw/system
rcu:  number:      624          0            0
rcu: cputime:       49          0         2446   ==> 2500(ms)
```

The fact that there are zero softirqs gives a hint that these were disabled, perhaps via local_bh_disable(). It is of course possible that there were no softirqs, perhaps because all events that would result in softirq execution are confined to other CPUs. In this case, the diagnosis should continue as shown in the next example.

3. A CPU looping with preemption disabled.

   Here, only the number of context switches is zero:

```
rcu:          hardirqs   softirqs   csw/system
rcu:  number:      624         45            0
rcu: cputime:       69          1         2425   ==> 2500(ms)
```

   This situation hints that the stalled CPU was looping with preemption disabled.

4. No looping, but massive hard and soft interrupts.

```
rcu:          hardirqs   softirqs   csw/system
rcu:  number:       xx         xx            0
rcu: cputime:       xx         xx            0   ==> 2500(ms)
```

   Here, the number and CPU time of hard interrupts are all non-zero, but the number of context switches and the in-kernel CPU time consumed are zero. The number and cputime of soft interrupts will usually be non-zero, but could be zero, for example, if the CPU was spinning within a single hard interrupt handler.

   If this type of RCU CPU stall warning can be reproduced, you can narrow it down by looking at /proc/interrupts or by writing code to trace each interrupt, for example, by referring to show_interrupts().

CHAPTER

TWELVE

USING RCU TO PROTECT READ-MOSTLY LINKED LISTS

One of the most common uses of RCU is protecting read-mostly linked lists (`struct list_head` in list.h). One big advantage of this approach is that all of the required memory ordering is provided by the list macros. This document describes several list-based RCU use cases.

12.1 Example 1: Read-mostly list: Deferred Destruction

A widely used usecase for RCU lists in the kernel is lockless iteration over all processes in the system. `task_struct::tasks` represents the list node that links all the processes. The list can be traversed in parallel to any list additions or removals.

The traversal of the list is done using `for_each_process()` which is defined by the 2 macros:

```
#define next_task(p) \
        list_entry_rcu((p)->tasks.next, struct task_struct, tasks)

#define for_each_process(p) \
        for (p = &init_task ; (p = next_task(p)) != &init_task ; )
```

The code traversing the list of all processes typically looks like:

```
rcu_read_lock();
for_each_process(p) {
        /* Do something with p */
}
rcu_read_unlock();
```

The simplified and heavily inlined code for removing a process from a task list is:

```
void release_task(struct task_struct *p)
{
        write_lock(&tasklist_lock);
        list_del_rcu(&p->tasks);
        write_unlock(&tasklist_lock);
        call_rcu(&p->rcu, delayed_put_task_struct);
}
```

When a process exits, `release_task()` calls `list_del_rcu(&p->tasks)` via __exit_signal() and __unhash_process() under `tasklist_lock` writer lock protection. The list_del_rcu() invocation removes the task from the list of all tasks. The `tasklist_lock` prevents concurrent list additions/removals from corrupting the list. Readers using `for_each_process()` are not protected with the `tasklist_lock`. To prevent readers from noticing changes in the list pointers, the `task_struct` object is freed only after one or more grace periods elapse, with the help of call_rcu(), which is invoked via put_task_struct_rcu_user(). This deferring of destruction ensures that any readers traversing the list will see valid `p->tasks.next` pointers and deletion/freeing can happen in parallel with traversal of the list. This pattern is also called an **existence lock**, since RCU refrains from invoking the delayed_put_task_struct() callback function until all existing readers finish, which guarantees that the `task_struct` object in question will remain in existence until after the completion of all RCU readers that might possibly have a reference to that object.

12.2 Example 2: Read-Side Action Taken Outside of Lock: No In-Place Updates

Some reader-writer locking use cases compute a value while holding the read-side lock, but continue to use that value after that lock is released. These use cases are often good candidates for conversion to RCU. One prominent example involves network packet routing. Because the packet-routing data tracks the state of equipment outside of the computer, it will at times contain stale data. Therefore, once the route has been computed, there is no need to hold the routing table static during transmission of the packet. After all, you can hold the routing table static all you want, but that won't keep the external Internet from changing, and it is the state of the external Internet that really matters. In addition, routing entries are typically added or deleted, rather than being modified in place. This is a rare example of the finite speed of light and the non-zero size of atoms actually helping make synchronization be lighter weight.

A straightforward example of this type of RCU use case may be found in the system-call auditing support. For example, a reader-writer locked implementation of `audit_filter_task()` might be as follows:

```
static enum audit_state audit_filter_task(struct task_struct *tsk, char **key)
{
        struct audit_entry *e;
        enum audit_state   state;

        read_lock(&auditsc_lock);
        /* Note: audit_filter_mutex held by caller. */
        list_for_each_entry(e, &audit_tsklist, list) {
                if (audit_filter_rules(tsk, &e->rule, NULL, &state)) {
                        if (state == AUDIT_STATE_RECORD)
                                *key = kstrdup(e->rule.filterkey, GFP_ATOMIC);
                        read_unlock(&auditsc_lock);
                        return state;
                }
        }
        read_unlock(&auditsc_lock);
        return AUDIT_BUILD_CONTEXT;
}
```

Here the list is searched under the lock, but the lock is dropped before the corresponding value is returned. By the time that this value is acted on, the list may well have been modified. This makes sense, since if you are turning auditing off, it is OK to audit a few extra system calls.

This means that RCU can be easily applied to the read side, as follows:

```
static enum audit_state audit_filter_task(struct task_struct *tsk, char **key)
{
        struct audit_entry *e;
        enum audit_state   state;

        rcu_read_lock();
        /* Note: audit_filter_mutex held by caller. */
        list_for_each_entry_rcu(e, &audit_tsklist, list) {
                if (audit_filter_rules(tsk, &e->rule, NULL, &state)) {
                        if (state == AUDIT_STATE_RECORD)
                                *key = kstrdup(e->rule.filterkey, GFP_ATOMIC);
                        rcu_read_unlock();
                        return state;
                }
        }
        rcu_read_unlock();
        return AUDIT_BUILD_CONTEXT;
}
```

The read_lock() and read_unlock() calls have become rcu_read_lock() and rcu_read_unlock(), respectively, and the list_for_each_entry() has become list_for_each_entry_rcu(). The **_rcu()** list-traversal primitives add READ_ONCE() and diagnostic checks for incorrect use outside of an RCU read-side critical section.

The changes to the update side are also straightforward. A reader-writer lock might be used as follows for deletion and insertion in these simplified versions of audit_del_rule() and audit_add_rule():

```
static inline int audit_del_rule(struct audit_rule *rule,
                                 struct list_head *list)
{
        struct audit_entry *e;

        write_lock(&auditsc_lock);
        list_for_each_entry(e, list, list) {
                if (!audit_compare_rule(rule, &e->rule)) {
                        list_del(&e->list);
                        write_unlock(&auditsc_lock);
                        return 0;
                }
        }
        write_unlock(&auditsc_lock);
        return -EFAULT;         /* No matching rule */
}

static inline int audit_add_rule(struct audit_entry *entry,
                                 struct list_head *list)
{
```

```
        write_lock(&auditsc_lock);
        if (entry->rule.flags & AUDIT_PREPEND) {
                entry->rule.flags &= ~AUDIT_PREPEND;
                list_add(&entry->list, list);
        } else {
                list_add_tail(&entry->list, list);
        }
        write_unlock(&auditsc_lock);
        return 0;
}
```

Following are the RCU equivalents for these two functions:

```
static inline int audit_del_rule(struct audit_rule *rule,
                                 struct list_head *list)
{
        struct audit_entry *e;

        /* No need to use the _rcu iterator here, since this is the only
         * deletion routine. */
        list_for_each_entry(e, list, list) {
                if (!audit_compare_rule(rule, &e->rule)) {
                        list_del_rcu(&e->list);
                        call_rcu(&e->rcu, audit_free_rule);
                        return 0;
                }
        }
        return -EFAULT;         /* No matching rule */
}

static inline int audit_add_rule(struct audit_entry *entry,
                                 struct list_head *list)
{
        if (entry->rule.flags & AUDIT_PREPEND) {
                entry->rule.flags &= ~AUDIT_PREPEND;
                list_add_rcu(&entry->list, list);
        } else {
                list_add_tail_rcu(&entry->list, list);
        }
        return 0;
}
```

Normally, the write_lock() and write_unlock() would be replaced by a spin_lock() and a spin_unlock(). But in this case, all callers hold `audit_filter_mutex`, so no additional locking is required. The auditsc_lock can therefore be eliminated, since use of RCU eliminates the need for writers to exclude readers.

The list_del(), list_add(), and list_add_tail() primitives have been replaced by list_del_rcu(), list_add_rcu(), and list_add_tail_rcu(). The **_rcu()** list-manipulation primitives add memory barriers that are needed on weakly ordered CPUs. The list_del_rcu() primitive omits the pointer poisoning debug-assist code that would otherwise cause concurrent readers to fail spectacularly.

So, when readers can tolerate stale data and when entries are either added or deleted, without in-place modification, it is very easy to use RCU!

12.3 Example 3: Handling In-Place Updates

The system-call auditing code does not update auditing rules in place. However, if it did, the reader-writer-locked code to do so might look as follows (assuming only `field_count` is updated, otherwise, the added fields would need to be filled in):

```
static inline int audit_upd_rule(struct audit_rule *rule,
                                 struct list_head *list,
                                 __u32 newaction,
                                 __u32 newfield_count)
{
        struct audit_entry *e;
        struct audit_entry *ne;

        write_lock(&auditsc_lock);
        /* Note: audit_filter_mutex held by caller. */
        list_for_each_entry(e, list, list) {
                if (!audit_compare_rule(rule, &e->rule)) {
                        e->rule.action = newaction;
                        e->rule.field_count = newfield_count;
                        write_unlock(&auditsc_lock);
                        return 0;
                }
        }
        write_unlock(&auditsc_lock);
        return -EFAULT;         /* No matching rule */
}
```

The RCU version creates a copy, updates the copy, then replaces the old entry with the newly updated entry. This sequence of actions, allowing concurrent reads while making a copy to perform an update, is what gives RCU (*read-copy update*) its name.

The RCU version of audit_upd_rule() is as follows:

```
static inline int audit_upd_rule(struct audit_rule *rule,
                                 struct list_head *list,
                                 __u32 newaction,
                                 __u32 newfield_count)
{
        struct audit_entry *e;
        struct audit_entry *ne;

        list_for_each_entry(e, list, list) {
                if (!audit_compare_rule(rule, &e->rule)) {
                        ne = kmalloc(sizeof(*entry), GFP_ATOMIC);
                        if (ne == NULL)
```

```
                                return -ENOMEM;
                        audit_copy_rule(&ne->rule, &e->rule);
                        ne->rule.action = newaction;
                        ne->rule.field_count = newfield_count;
                        list_replace_rcu(&e->list, &ne->list);
                        call_rcu(&e->rcu, audit_free_rule);
                        return 0;
                }
        }
        return -EFAULT;         /* No matching rule */
}
```

Again, this assumes that the caller holds `audit_filter_mutex`. Normally, the writer lock would become a spinlock in this sort of code.

The update_lsm_rule() does something very similar, for those who would prefer to look at real Linux-kernel code.

Another use of this pattern can be found in the openswitch driver's *connection tracking table* code in `ct_limit_set()`. The table holds connection tracking entries and has a limit on the maximum entries. There is one such table per-zone and hence one *limit* per zone. The zones are mapped to their limits through a hashtable using an RCU-managed hlist for the hash chains. When a new limit is set, a new limit object is allocated and `ct_limit_set()` is called to replace the old limit object with the new one using list_replace_rcu(). The old limit object is then freed after a grace period using kfree_rcu().

12.4 Example 4: Eliminating Stale Data

The auditing example above tolerates stale data, as do most algorithms that are tracking external state. After all, given there is a delay from the time the external state changes before Linux becomes aware of the change, and so as noted earlier, a small quantity of additional RCU-induced staleness is generally not a problem.

However, there are many examples where stale data cannot be tolerated. One example in the Linux kernel is the System V IPC (see the shm_lock() function in ipc/shm.c). This code checks a *deleted* flag under a per-entry spinlock, and, if the *deleted* flag is set, pretends that the entry does not exist. For this to be helpful, the search function must return holding the per-entry spinlock, as shm_lock() does in fact do.

Quick Quiz:

For the deleted-flag technique to be helpful, why is it necessary to hold the per-entry lock while returning from the search function?

Answer to Quick Quiz

If the system-call audit module were to ever need to reject stale data, one way to accomplish this would be to add a `deleted` flag and a `lock` spinlock to the `audit_entry` structure, and modify audit_filter_task() as follows:

```
static enum audit_state audit_filter_task(struct task_struct *tsk)
{
        struct audit_entry *e;
```

```
        enum audit_state   state;

        rcu_read_lock();
        list_for_each_entry_rcu(e, &audit_tsklist, list) {
                if (audit_filter_rules(tsk, &e->rule, NULL, &state)) {
                        spin_lock(&e->lock);
                        if (e->deleted) {
                                spin_unlock(&e->lock);
                                rcu_read_unlock();
                                return AUDIT_BUILD_CONTEXT;
                        }
                        rcu_read_unlock();
                        if (state == AUDIT_STATE_RECORD)
                                *key = kstrdup(e->rule.filterkey, GFP_ATOMIC);
                        return state;
                }
        }
        rcu_read_unlock();
        return AUDIT_BUILD_CONTEXT;
}
```

The `audit_del_rule()` function would need to set the `deleted` flag under the spinlock as follows:

```
static inline int audit_del_rule(struct audit_rule *rule,
                                 struct list_head *list)
{
        struct audit_entry *e;

        /* No need to use the _rcu iterator here, since this
         * is the only deletion routine. */
        list_for_each_entry(e, list, list) {
                if (!audit_compare_rule(rule, &e->rule)) {
                        spin_lock(&e->lock);
                        list_del_rcu(&e->list);
                        e->deleted = 1;
                        spin_unlock(&e->lock);
                        call_rcu(&e->rcu, audit_free_rule);
                        return 0;
                }
        }
        return -EFAULT;         /* No matching rule */
}
```

This too assumes that the caller holds `audit_filter_mutex`.

Note that this example assumes that entries are only added and deleted. Additional mechanism is required to deal correctly with the update-in-place performed by audit_upd_rule(). For one thing, audit_upd_rule() would need to hold the locks of both the old `audit_entry` and its replacement while executing the list_replace_rcu().

12.5 Example 5: Skipping Stale Objects

For some use cases, reader performance can be improved by skipping stale objects during read-side list traversal, where stale objects are those that will be removed and destroyed after one or more grace periods. One such example can be found in the timerfd subsystem. When a `CLOCK_REALTIME` clock is reprogrammed (for example due to setting of the system time) then all programmed `timerfds` that depend on this clock get triggered and processes waiting on them are awakened in advance of their scheduled expiry. To facilitate this, all such timers are added to an RCU-managed `cancel_list` when they are setup in `timerfd_setup_cancel()`:

```
static void timerfd_setup_cancel(struct timerfd_ctx *ctx, int flags)
{
        spin_lock(&ctx->cancel_lock);
        if ((ctx->clockid == CLOCK_REALTIME ||
             ctx->clockid == CLOCK_REALTIME_ALARM) &&
            (flags & TFD_TIMER_ABSTIME) && (flags & TFD_TIMER_CANCEL_ON_SET)) {
                if (!ctx->might_cancel) {
                        ctx->might_cancel = true;
                        spin_lock(&cancel_lock);
                        list_add_rcu(&ctx->clist, &cancel_list);
                        spin_unlock(&cancel_lock);
                }
        } else {
                __timerfd_remove_cancel(ctx);
        }
        spin_unlock(&ctx->cancel_lock);
}
```

When a timerfd is freed (fd is closed), then the `might_cancel` flag of the timerfd object is cleared, the object removed from the `cancel_list` and destroyed, as shown in this simplified and inlined version of timerfd_release():

```
int timerfd_release(struct inode *inode, struct file *file)
{
        struct timerfd_ctx *ctx = file->private_data;

        spin_lock(&ctx->cancel_lock);
        if (ctx->might_cancel) {
                ctx->might_cancel = false;
                spin_lock(&cancel_lock);
                list_del_rcu(&ctx->clist);
                spin_unlock(&cancel_lock);
        }
        spin_unlock(&ctx->cancel_lock);

        if (isalarm(ctx))
                alarm_cancel(&ctx->t.alarm);
        else
                hrtimer_cancel(&ctx->t.tmr);
        kfree_rcu(ctx, rcu);
        return 0;
```

```
}
```

If the `CLOCK_REALTIME` clock is set, for example by a time server, the hrtimer framework calls `timerfd_clock_was_set()` which walks the `cancel_list` and wakes up processes waiting on the timerfd. While iterating the `cancel_list`, the `might_cancel` flag is consulted to skip stale objects:

```
void timerfd_clock_was_set(void)
{
        ktime_t moffs = ktime_mono_to_real(0);
        struct timerfd_ctx *ctx;
        unsigned long flags;

        rcu_read_lock();
        list_for_each_entry_rcu(ctx, &cancel_list, clist) {
                if (!ctx->might_cancel)
                        continue;
                spin_lock_irqsave(&ctx->wqh.lock, flags);
                if (ctx->moffs != moffs) {
                        ctx->moffs = KTIME_MAX;
                        ctx->ticks++;
                        wake_up_locked_poll(&ctx->wqh, EPOLLIN);
                }
                spin_unlock_irqrestore(&ctx->wqh.lock, flags);
        }
        rcu_read_unlock();
}
```

The key point is that because RCU-protected traversal of the `cancel_list` happens concurrently with object addition and removal, sometimes the traversal can access an object that has been removed from the list. In this example, a flag is used to skip such objects.

12.6 Summary

Read-mostly list-based data structures that can tolerate stale data are the most amenable to use of RCU. The simplest case is where entries are either added or deleted from the data structure (or atomically modified in place), but non-atomic in-place modifications can be handled by making a copy, updating the copy, then replacing the original with the copy. If stale data cannot be tolerated, then a *deleted* flag may be used in conjunction with a per-entry spinlock in order to allow the search function to reject newly deleted data.

Answer to Quick Quiz:

> For the deleted-flag technique to be helpful, why is it necessary to hold the per-entry lock while returning from the search function?
>
> If the search function drops the per-entry lock before returning, then the caller will be processing stale data in any case. If it is really OK to be processing stale data, then you don't need a *deleted* flag. If processing stale data really is a problem, then you need to hold the per-entry lock across all of the code that uses the value that was returned.

Back to Quick Quiz

CHAPTER

THIRTEEN

USING RCU TO PROTECT DYNAMIC NMI HANDLERS

Although RCU is usually used to protect read-mostly data structures, it is possible to use RCU to provide dynamic non-maskable interrupt handlers, as well as dynamic irq handlers. This document describes how to do this, drawing loosely from Zwane Mwaikambo's NMI-timer work in an old version of "arch/x86/kernel/traps.c".

The relevant pieces of code are listed below, each followed by a brief explanation:

```
static int dummy_nmi_callback(struct pt_regs *regs, int cpu)
{
        return 0;
}
```

The dummy_nmi_callback() function is a "dummy" NMI handler that does nothing, but returns zero, thus saying that it did nothing, allowing the NMI handler to take the default machine-specific action:

```
static nmi_callback_t nmi_callback = dummy_nmi_callback;
```

This nmi_callback variable is a global function pointer to the current NMI handler:

```
void do_nmi(struct pt_regs * regs, long error_code)
{
        int cpu;

        nmi_enter();

        cpu = smp_processor_id();
        ++nmi_count(cpu);

        if (!rcu_dereference_sched(nmi_callback)(regs, cpu))
                default_do_nmi(regs);

        nmi_exit();
}
```

The do_nmi() function processes each NMI. It first disables preemption in the same way that a hardware irq would, then increments the per-CPU count of NMIs. It then invokes the NMI handler stored in the nmi_callback function pointer. If this handler returns zero, do_nmi() invokes the default_do_nmi() function to handle a machine-specific NMI. Finally, preemption is restored.

In theory, rcu_dereference_sched() is not needed, since this code runs only on i386, which in theory does not need rcu_dereference_sched() anyway. However, in practice it is a good documentation aid, particularly for anyone attempting to do something similar on Alpha or on systems with aggressive optimizing compilers.

Quick Quiz:

Why might the rcu_dereference_sched() be necessary on Alpha, given that the code referenced by the pointer is read-only?

Answer to Quick Quiz

Back to the discussion of NMI and RCU:

```
void set_nmi_callback(nmi_callback_t callback)
{
        rcu_assign_pointer(nmi_callback, callback);
}
```

The set_nmi_callback() function registers an NMI handler. Note that any data that is to be used by the callback must be initialized up -before- the call to set_nmi_callback(). On architectures that do not order writes, the rcu_assign_pointer() ensures that the NMI handler sees the initialized values:

```
void unset_nmi_callback(void)
{
        rcu_assign_pointer(nmi_callback, dummy_nmi_callback);
}
```

This function unregisters an NMI handler, restoring the original dummy_nmi_handler(). However, there may well be an NMI handler currently executing on some other CPU. We therefore cannot free up any data structures used by the old NMI handler until execution of it completes on all other CPUs.

One way to accomplish this is via synchronize_rcu(), perhaps as follows:

```
unset_nmi_callback();
synchronize_rcu();
kfree(my_nmi_data);
```

This works because (as of v4.20) synchronize_rcu() blocks until all CPUs complete any preemption-disabled segments of code that they were executing. Since NMI handlers disable preemption, synchronize_rcu() is guaranteed not to return until all ongoing NMI handlers exit. It is therefore safe to free up the handler's data as soon as synchronize_rcu() returns.

Important note: for this to work, the architecture in question must invoke nmi_enter() and nmi_exit() on NMI entry and exit, respectively.

Answer to Quick Quiz:

Why might the rcu_dereference_sched() be necessary on Alpha, given that the code referenced by the pointer is read-only?

The caller to set_nmi_callback() might well have initialized some data that is to be used by the new NMI handler. In this case, the rcu_dereference_sched() would be needed, because otherwise a CPU that received an NMI just after the new handler was set might see the pointer to the new NMI handler, but the old pre-initialized version of the handler's data.

This same sad story can happen on other CPUs when using a compiler with aggressive pointer-value speculation optimizations. (But please don't!)

More important, the rcu_dereference_sched() makes it clear to someone reading the code that the pointer is being protected by RCU-sched.

CHAPTER

FOURTEEN

RCU ON UNIPROCESSOR SYSTEMS

A common misconception is that, on UP systems, the call_rcu() primitive may immediately invoke its function. The basis of this misconception is that since there is only one CPU, it should not be necessary to wait for anything else to get done, since there are no other CPUs for anything else to be happening on. Although this approach will *sort of* work a surprising amount of the time, it is a very bad idea in general. This document presents three examples that demonstrate exactly how bad an idea this is.

14.1 Example 1: softirq Suicide

Suppose that an RCU-based algorithm scans a linked list containing elements A, B, and C in process context, and can delete elements from this same list in softirq context. Suppose that the process-context scan is referencing element B when it is interrupted by softirq processing, which deletes element B, and then invokes call_rcu() to free element B after a grace period.

Now, if call_rcu() were to directly invoke its arguments, then upon return from softirq, the list scan would find itself referencing a newly freed element B. This situation can greatly decrease the life expectancy of your kernel.

This same problem can occur if call_rcu() is invoked from a hardware interrupt handler.

14.2 Example 2: Function-Call Fatality

Of course, one could avert the suicide described in the preceding example by having call_rcu() directly invoke its arguments only if it was called from process context. However, this can fail in a similar manner.

Suppose that an RCU-based algorithm again scans a linked list containing elements A, B, and C in process context, but that it invokes a function on each element as it is scanned. Suppose further that this function deletes element B from the list, then passes it to call_rcu() for deferred freeing. This may be a bit unconventional, but it is perfectly legal RCU usage, since call_rcu() must wait for a grace period to elapse. Therefore, in this case, allowing call_rcu() to immediately invoke its arguments would cause it to fail to make the fundamental guarantee underlying RCU, namely that call_rcu() defers invoking its arguments until all RCU read-side critical sections currently executing have completed.

Quick Quiz #1:

Why is it *not* legal to invoke synchronize_rcu() in this case?

Answers to Quick Quiz

14.3 Example 3: Death by Deadlock

Suppose that call_rcu() is invoked while holding a lock, and that the callback function must acquire this same lock. In this case, if call_rcu() were to directly invoke the callback, the result would be self-deadlock *even if* this invocation occurred from a later call_rcu() invocation a full grace period later.

In some cases, it would possible to restructure to code so that the call_rcu() is delayed until after the lock is released. However, there are cases where this can be quite ugly:

1. If a number of items need to be passed to call_rcu() within the same critical section, then the code would need to create a list of them, then traverse the list once the lock was released.
2. In some cases, the lock will be held across some kernel API, so that delaying the call_rcu() until the lock is released requires that the data item be passed up via a common API. It is far better to guarantee that callbacks are invoked with no locks held than to have to modify such APIs to allow arbitrary data items to be passed back up through them.

If call_rcu() directly invokes the callback, painful locking restrictions or API changes would be required.

Quick Quiz #2:
What locking restriction must RCU callbacks respect?

Answers to Quick Quiz

It is important to note that userspace RCU implementations *do* permit call_rcu() to directly invoke callbacks, but only if a full grace period has elapsed since those callbacks were queued. This is the case because some userspace environments are extremely constrained. Nevertheless, people writing userspace RCU implementations are strongly encouraged to avoid invoking callbacks from call_rcu(), thus obtaining the deadlock-avoidance benefits called out above.

14.4 Summary

Permitting call_rcu() to immediately invoke its arguments breaks RCU, even on a UP system. So do not do it! Even on a UP system, the RCU infrastructure *must* respect grace periods, and *must* invoke callbacks from a known environment in which no locks are held.

Note that it *is* safe for synchronize_rcu() to return immediately on UP systems, including PREEMPT SMP builds running on UP systems.

Quick Quiz #3:
Why can't synchronize_rcu() return immediately on UP systems running preemptible RCU?

Answer to Quick Quiz #1:
Why is it *not* legal to invoke synchronize_rcu() in this case?

Because the calling function is scanning an RCU-protected linked list, and is therefore within an RCU read-side critical section. Therefore, the called function has been invoked within an RCU read-side critical section, and is not permitted to block.

Answer to Quick Quiz #2:
What locking restriction must RCU callbacks respect?

Any lock that is acquired within an RCU callback must be acquired elsewhere using an _bh variant of the spinlock primitive. For example, if "mylock" is acquired by an RCU callback, then a process-context acquisition of this lock must use something like spin_lock_bh() to acquire the lock. Please note that it is also OK to use _irq variants of spinlocks, for example, spin_lock_irqsave().

If the process-context code were to simply use spin_lock(), then, since RCU callbacks can be invoked from softirq context, the callback might be called from a softirq that interrupted the process-context critical section. This would result in self-deadlock.

This restriction might seem gratuitous, since very few RCU callbacks acquire locks directly. However, a great many RCU callbacks do acquire locks *indirectly*, for example, via the kfree() primitive.

Answer to Quick Quiz #3:

Why can't synchronize_rcu() return immediately on UP systems running preemptible RCU?

Because some other task might have been preempted in the middle of an RCU read-side critical section. If synchronize_rcu() simply immediately returned, it would prematurely signal the end of the grace period, which would come as a nasty shock to that other thread when it started running again.

CHAPTER

FIFTEEN

A TOUR THROUGH TREE_RCU'S GRACE-PERIOD MEMORY ORDERING

August 8, 2017

This article was contributed by Paul E. McKenney

15.1 Introduction

This document gives a rough visual overview of how Tree RCU's grace-period memory ordering guarantee is provided.

15.2 What Is Tree RCU's Grace Period Memory Ordering Guarantee?

RCU grace periods provide extremely strong memory-ordering guarantees for non-idle non-offline code. Any code that happens after the end of a given RCU grace period is guaranteed to see the effects of all accesses prior to the beginning of that grace period that are within RCU read-side critical sections. Similarly, any code that happens before the beginning of a given RCU grace period is guaranteed to not see the effects of all accesses following the end of that grace period that are within RCU read-side critical sections.

Note well that RCU-sched read-side critical sections include any region of code for which preemption is disabled. Given that each individual machine instruction can be thought of as an extremely small region of preemption-disabled code, one can think of `synchronize_rcu()` as `smp_mb()` on steroids.

RCU updaters use this guarantee by splitting their updates into two phases, one of which is executed before the grace period and the other of which is executed after the grace period. In the most common use case, phase one removes an element from a linked RCU-protected data structure, and phase two frees that element. For this to work, any readers that have witnessed state prior to the phase-one update (in the common case, removal) must not witness state following the phase-two update (in the common case, freeing).

The RCU implementation provides this guarantee using a network of lock-based critical sections, memory barriers, and per-CPU processing, as is described in the following sections.

15.3 Tree RCU Grace Period Memory Ordering Building Blocks

The workhorse for RCU's grace-period memory ordering is the critical section for the rcu_node structure's ->lock. These critical sections use helper functions for lock acquisition, including raw_spin_lock_rcu_node(), raw_spin_lock_irq_rcu_node(), and raw_spin_lock_irqsave_rcu_node(). Their lock-release counterparts are raw_spin_unlock_rcu_node(), raw_spin_unlock_irq_rcu_node(), and raw_spin_unlock_irqrestore_rcu_node(), respectively. For completeness, a raw_spin_trylock_rcu_node() is also provided. The key point is that the lock-acquisition functions, including raw_spin_trylock_rcu_node(), all invoke smp_mb__after_unlock_lock() immediately after successful acquisition of the lock.

Therefore, for any given rcu_node structure, any access happening before one of the above lock-release functions will be seen by all CPUs as happening before any access happening after a later one of the above lock-acquisition functions. Furthermore, any access happening before one of the above lock-release function on any given CPU will be seen by all CPUs as happening before any access happening after a later one of the above lock-acquisition functions executing on that same CPU, even if the lock-release and lock-acquisition functions are operating on different rcu_node structures. Tree RCU uses these two ordering guarantees to form an ordering network among all CPUs that were in any way involved in the grace period, including any CPUs that came online or went offline during the grace period in question.

The following litmus test exhibits the ordering effects of these lock-acquisition and lock-release functions:

```
int x, y, z;

void task0(void)
{
  raw_spin_lock_rcu_node(rnp);
  WRITE_ONCE(x, 1);
  r1 = READ_ONCE(y);
  raw_spin_unlock_rcu_node(rnp);
}

void task1(void)
{
  raw_spin_lock_rcu_node(rnp);
  WRITE_ONCE(y, 1);
  r2 = READ_ONCE(z);
  raw_spin_unlock_rcu_node(rnp);
}

void task2(void)
{
  WRITE_ONCE(z, 1);
  smp_mb();
  r3 = READ_ONCE(x);
}

WARN_ON(r1 == 0 && r2 == 0 && r3 == 0);
```

The `WARN_ON()` is evaluated at "the end of time", after all changes have propagated throughout the system. Without the `smp_mb__after_unlock_lock()` provided by the acquisition functions, this `WARN_ON()` could trigger, for example on PowerPC. The `smp_mb__after_unlock_lock()` invocations prevent this `WARN_ON()` from triggering.

Quick Quiz:

But the chain of rcu_node-structure lock acquisitions guarantees that new readers will see all of the updater's pre-grace-period accesses and also guarantees that the updater's post-grace-period accesses will see all of the old reader's accesses. So why do we need all of those calls to smp_mb__after_unlock_lock()?

Answer:

Because we must provide ordering for RCU's polling grace-period primitives, for example, get_state_synchronize_rcu() and poll_state_synchronize_rcu(). Consider this code:

```
CPU 0                                     CPU 1
----                                      ----
WRITE_ONCE(X, 1)                          WRITE_ONCE(Y, 1)
g = get_state_synchronize_rcu()           smp_mb()
while (!poll_state_synchronize_rcu(g))    r1 = READ_ONCE(X)
        continue;
r0 = READ_ONCE(Y)
```

RCU guarantees that the outcome r0 == 0 && r1 == 0 will not happen, even if CPU 1 is in an RCU extended quiescent state (idle or offline) and thus won't interact directly with the RCU core processing at all.

This approach must be extended to include idle CPUs, which need RCU's grace-period memory ordering guarantee to extend to any RCU read-side critical sections preceding and following the current idle sojourn. This case is handled by calls to the strongly ordered `atomic_add_return()` read-modify-write atomic operation that is invoked within `rcu_dynticks_eqs_enter()` at idle-entry time and within `rcu_dynticks_eqs_exit()` at idle-exit time. The grace-period kthread invokes `rcu_dynticks_snap()` and `rcu_dynticks_in_eqs_since()` (both of which invoke an `atomic_add_return()` of zero) to detect idle CPUs.

Quick Quiz:

But what about CPUs that remain offline for the entire grace period?

Answer:

Such CPUs will be offline at the beginning of the grace period, so the grace period won't expect quiescent states from them. Races between grace-period start and CPU-hotplug operations are mediated by the CPU's leaf `rcu_node` structure's `->lock` as described above.

The approach must be extended to handle one final case, that of waking a task blocked in `synchronize_rcu()`. This task might be affined to a CPU that is not yet aware that the grace period has ended, and thus might not yet be subject to the grace period's memory ordering. Therefore, there is an `smp_mb()` after the return from `wait_for_completion()` in the `synchronize_rcu()` code path.

Quick Quiz:

What? Where??? I don't see any `smp_mb()` after the return from `wait_for_completion()`!!!

Answer:

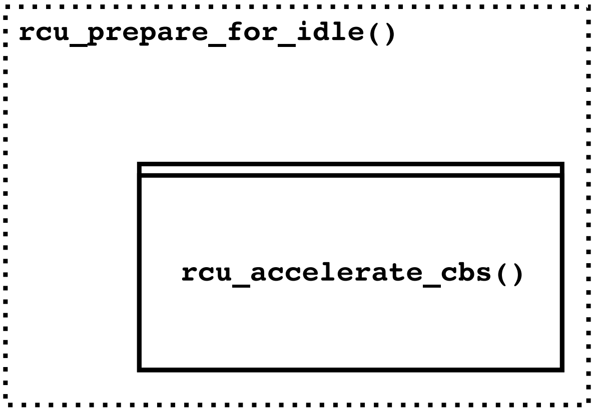
That would be because I spotted the need for that `smp_mb()` during the creation of this documentation, and it is therefore unlikely to hit mainline before v4.14. Kudos to Lance Roy, Will Deacon, Peter Zijlstra, and Jonathan Cameron for asking questions that sensitized me to the rather elaborate sequence of events that demonstrate the need for this memory barrier.

Tree RCU's grace--period memory-ordering guarantees rely most heavily on the `rcu_node` structure's `->lock` field, so much so that it is necessary to abbreviate this pattern in the diagrams in the next section. For example, consider the `rcu_prepare_for_idle()` function shown below, which is one of several functions that enforce ordering of newly arrived RCU callbacks against future grace periods:

```
static void rcu_prepare_for_idle(void)
{
  bool needwake;
  struct rcu_data *rdp = this_cpu_ptr(&rcu_data);
  struct rcu_node *rnp;
  int tne;

  lockdep_assert_irqs_disabled();
  if (rcu_rdp_is_offloaded(rdp))
    return;

  /* Handle nohz enablement switches conservatively. */
  tne = READ_ONCE(tick_nohz_active);
  if (tne != rdp->tick_nohz_enabled_snap) {
    if (!rcu_segcblist_empty(&rdp->cblist))
      invoke_rcu_core(); /* force nohz to see update. */
    rdp->tick_nohz_enabled_snap = tne;
    return;
  }
  if (!tne)
    return;

  /*
   * If we have not yet accelerated this jiffy, accelerate all
   * callbacks on this CPU.
  */
  if (rdp->last_accelerate == jiffies)
    return;
  rdp->last_accelerate = jiffies;
  if (rcu_segcblist_pend_cbs(&rdp->cblist)) {
    rnp = rdp->mynode;
    raw_spin_lock_rcu_node(rnp); /* irqs already disabled. */
    needwake = rcu_accelerate_cbs(rnp, rdp);
    raw_spin_unlock_rcu_node(rnp); /* irqs remain disabled. */
    if (needwake)
      rcu_gp_kthread_wake();
```

```
37   }
38 }
```

But the only part of `rcu_prepare_for_idle()` that really matters for this discussion are lines 32–34. We will therefore abbreviate this function as follows:



The box represents the `rcu_node` structure's `->lock` critical section, with the double line on top representing the additional `smp_mb__after_unlock_lock()`.

15.3.1 Tree RCU Grace Period Memory Ordering Components

Tree RCU's grace-period memory-ordering guarantee is provided by a number of RCU components:

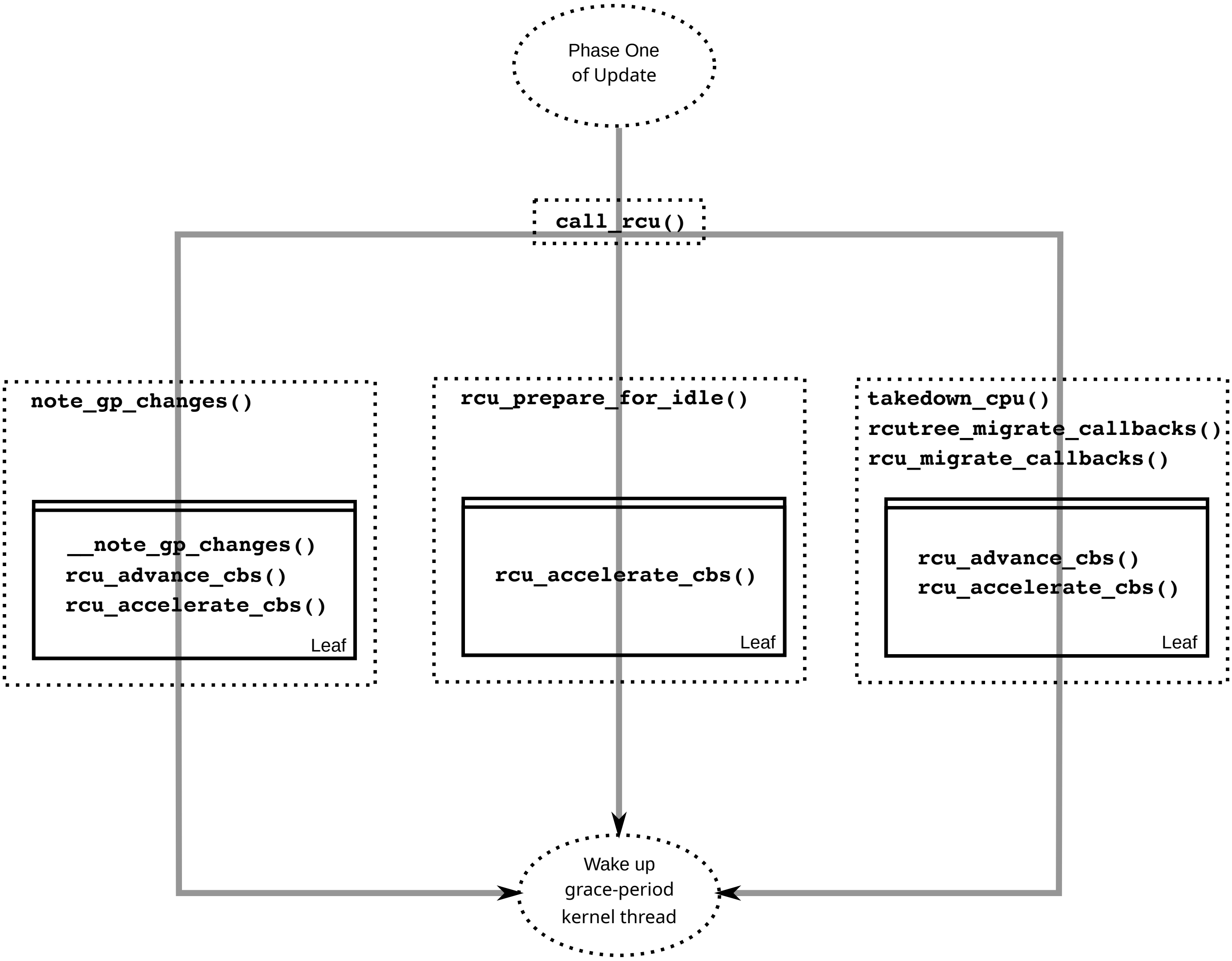
1. *Callback Registry*
2. *Grace-Period Initialization*
3. *Self-Reported Quiescent States*
4. *Dynamic Tick Interface*
5. *CPU-Hotplug Interface*
6. *Forcing Quiescent States*
7. *Grace-Period Cleanup*
8. *Callback Invocation*

Each of the following section looks at the corresponding component in detail.

Callback Registry

If RCU's grace-period guarantee is to mean anything at all, any access that happens before a given invocation of `call_rcu()` must also happen before the corresponding grace period. The implementation of this portion of RCU's grace period guarantee is shown in the following figure:

Because `call_rcu()` normally acts only on CPU-local state, it provides no ordering guarantees, either for itself or for phase one of the update (which again will usually be removal of an element from an RCU-protected data structure). It simply enqueues the `rcu_head` structure


Phase One
of Update
call_rcu()
note_gp_changes()
__note_gp_changes()
rcu_advance_cbs()
rcu_accelerate_cbs()
Leaf
rcu_prepare_for_idle()
rcu_accelerate_cbs()
Leaf
takedown_cpu()
rcutree_migrate_callbacks()
rcu_migrate_callbacks()
rcu_advance_cbs()
rcu_accelerate_cbs()
Leaf
Wake up
grace-period
kernel thread

on a per-CPU list, which cannot become associated with a grace period until a later call to `rcu_accelerate_cbs()`, as shown in the diagram above.

One set of code paths shown on the left invokes `rcu_accelerate_cbs()` via `note_gp_changes()`, either directly from `call_rcu()` (if the current CPU is inundated with queued `rcu_head` structures) or more likely from an `RCU_SOFTIRQ` handler. Another code path in the middle is taken only in kernels built with `CONFIG_RCU_FAST_NO_HZ=y`, which invokes `rcu_accelerate_cbs()` via `rcu_prepare_for_idle()`. The final code path on the right is taken only in kernels built with `CONFIG_HOTPLUG_CPU=y`, which invokes `rcu_accelerate_cbs()` via `rcu_advance_cbs()`, `rcu_migrate_callbacks`, `rcutree_migrate_callbacks()`, and `takedown_cpu()`, which in turn is invoked on a surviving CPU after the outgoing CPU has been completely offlined.

There are a few other code paths within grace-period processing that opportunistically invoke `rcu_accelerate_cbs()`. However, either way, all of the CPU's recently queued `rcu_head` structures are associated with a future grace-period number under the protection of the CPU's lead `rcu_node` structure's `->lock`. In all cases, there is full ordering against any prior critical section for that same `rcu_node` structure's `->lock`, and also full ordering against any of the current task's or CPU's prior critical sections for any `rcu_node` structure's `->lock`.

The next section will show how this ordering ensures that any accesses prior to the `call_rcu()` (particularly including phase one of the update) happen before the start of the corresponding grace period.

Quick Quiz:
But what about `synchronize_rcu()`?
Answer:
The `synchronize_rcu()` passes `call_rcu()` to `wait_rcu_gp()`, which invokes it. So either way, it eventually comes down to `call_rcu()`.

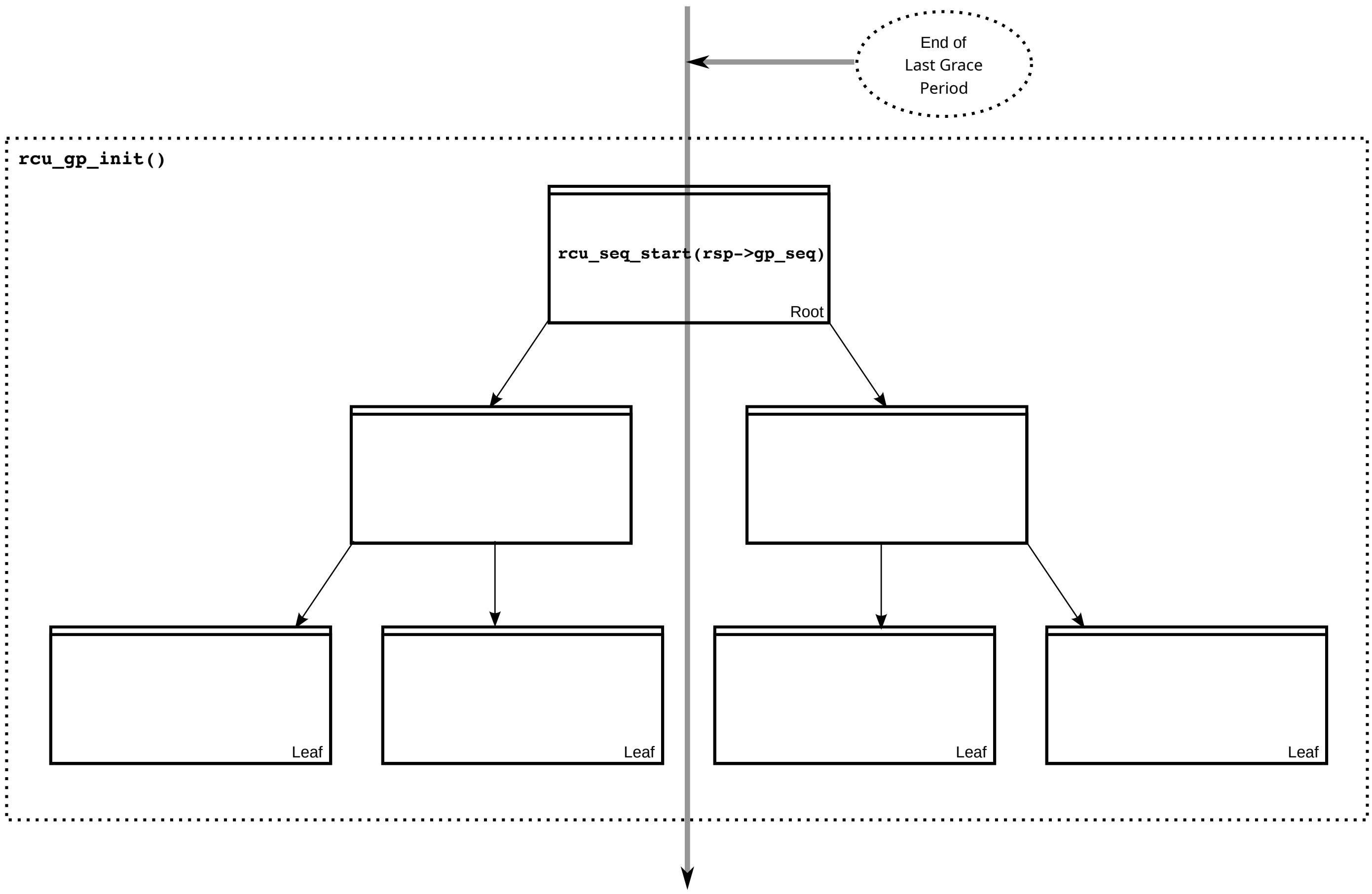
Grace-Period Initialization

Grace-period initialization is carried out by the grace-period kernel thread, which makes several passes over the `rcu_node` tree within the `rcu_gp_init()` function. This means that showing the full flow of ordering through the grace-period computation will require duplicating this tree. If you find this confusing, please note that the state of the `rcu_node` changes over time, just like Heraclitus's river. However, to keep the `rcu_node` river tractable, the grace-period kernel thread's traversals are presented in multiple parts, starting in this section with the various phases of grace-period initialization.

The first ordering-related grace-period initialization action is to advance the `rcu_state` structure's `->gp_seq` grace-period-number counter, as shown below:

The actual increment is carried out using `smp_store_release()`, which helps reject false-positive RCU CPU stall detection. Note that only the root `rcu_node` structure is touched.

The first pass through the `rcu_node` tree updates bitmasks based on CPUs having come online or gone offline since the start of the previous grace period. In the common case where the number of online CPUs for this `rcu_node` structure has not transitioned to or from zero, this pass will scan only the leaf `rcu_node` structures. However, if the number of online CPUs for a given leaf `rcu_node` structure has transitioned from zero, `rcu_init_new_rnp()` will be invoked for the first incoming CPU. Similarly, if the number of online CPUs for a given leaf `rcu_node` structure has transitioned to zero, `rcu_cleanup_dead_rnp()` will be invoked for the last outgoing CPU.



The diagram below shows the path of ordering if the leftmost `rcu_node` structure onlines its first CPU and if the next `rcu_node` structure has no online CPUs (or, alternatively if the leftmost `rcu_node` structure offlines its last CPU and if the next `rcu_node` structure has no online CPUs).

The final `rcu_gp_init()` pass through the `rcu_node` tree traverses breadth-first, setting each `rcu_node` structure's `->gp_seq` field to the newly advanced value from the `rcu_state` structure, as shown in the following diagram.

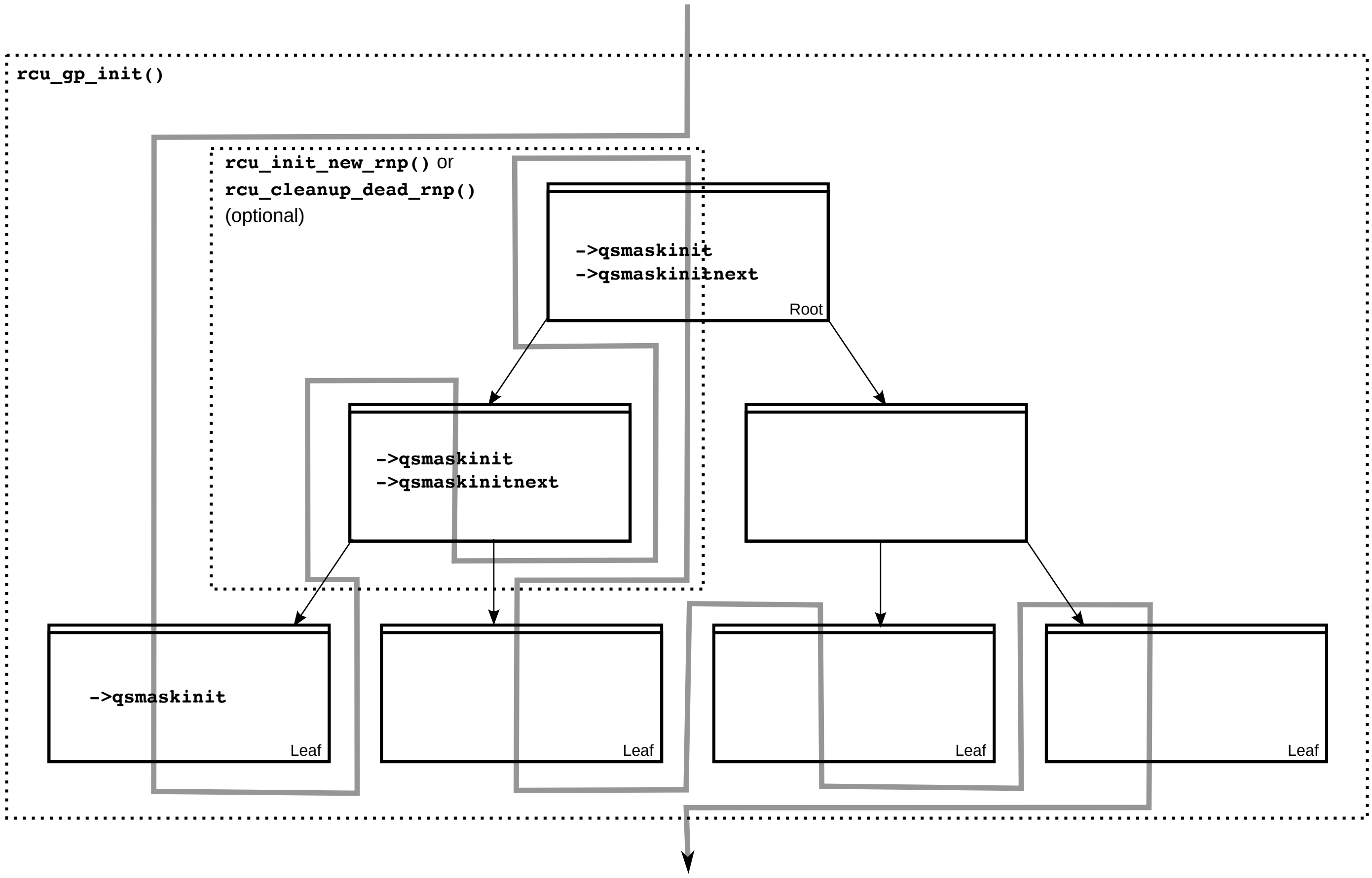
This change will also cause each CPU's next call to `__note_gp_changes()` to notice that a new grace period has started, as described in the next section. But because the grace-period kthread started the grace period at the root (with the advancing of the `rcu_state` structure's `->gp_seq` field) before setting each leaf `rcu_node` structure's `->gp_seq` field, each CPU's observation of the start of the grace period will happen after the actual start of the grace period.

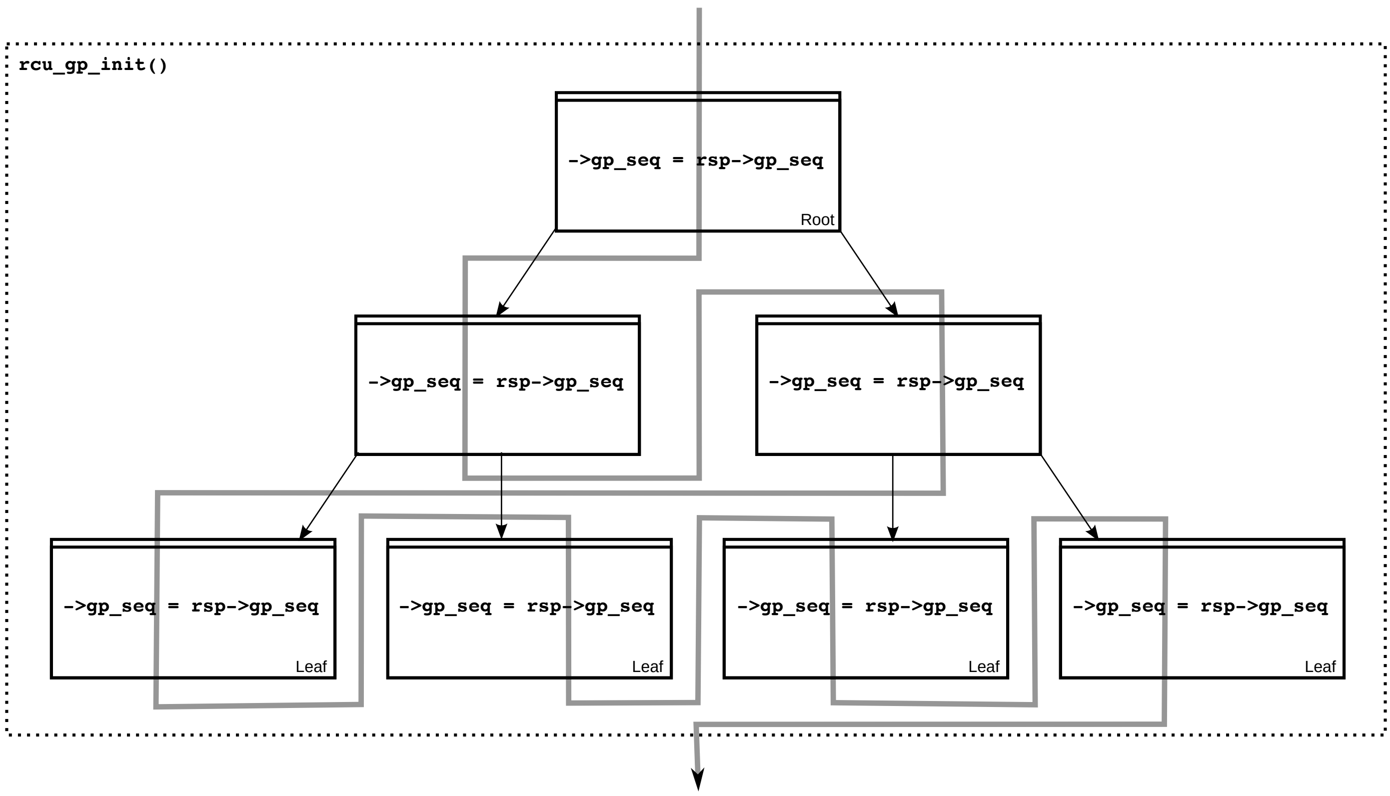
Quick Quiz:

But what about the CPU that started the grace period? Why wouldn't it see the start of the grace period right when it started that grace period?

Answer:

In some deep philosophical and overly anthromorphized sense, yes, the CPU starting the grace period is immediately aware of having done so. However, if we instead assume that RCU is not self-aware, then even the CPU starting the grace period does not really become aware of the start of this grace period until its first call to `__note_gp_changes()`. On the other hand, this CPU potentially gets early notification because it invokes `__note_gp_changes()` during its last `rcu_gp_init()` pass through its leaf `rcu_node` structure.


rcu_gp_init()
rcu_init_new_rnp() or
rcu_cleanup_dead_rnp()
(optional)
->qsmaskinit
->qsmaskinitnext
Root
->qsmaskinit
->qsmaskinitnext
->qsmaskinit
Leaf
Leaf
Leaf
Leaf


rcu_gp_init()
->gp_seq = rsp->gp_seq
Root
->gp_seq = rsp->gp_seq
->gp_seq = rsp->gp_seq
->gp_seq = rsp->gp_seq
->gp_seq = rsp->gp_seq
->gp_seq = rsp->gp_seq
->gp_seq = rsp->gp_seq
Leaf
Leaf
Leaf
Leaf

Self-Reported Quiescent States

When all entities that might block the grace period have reported quiescent states (or as described in a later section, had quiescent states reported on their behalf), the grace period can end. Online non-idle CPUs report their own quiescent states, as shown in the following diagram:

This is for the last CPU to report a quiescent state, which signals the end of the grace period. Earlier quiescent states would push up the `rcu_node` tree only until they encountered an `rcu_node` structure that is waiting for additional quiescent states. However, ordering is nevertheless preserved because some later quiescent state will acquire that `rcu_node` structure's `->lock`.

Any number of events can lead up to a CPU invoking `note_gp_changes` (or alternatively, directly invoking `__note_gp_changes()`), at which point that CPU will notice the start of a new grace period while holding its leaf `rcu_node` lock. Therefore, all execution shown in this diagram happens after the start of the grace period. In addition, this CPU will consider any RCU read-side critical section that started before the invocation of `__note_gp_changes()` to have started before the grace period, and thus a critical section that the grace period must wait on.

Quick Quiz:

But a RCU read-side critical section might have started after the beginning of the grace period (the advancing of `->gp_seq` from earlier), so why should the grace period wait on such a critical section?

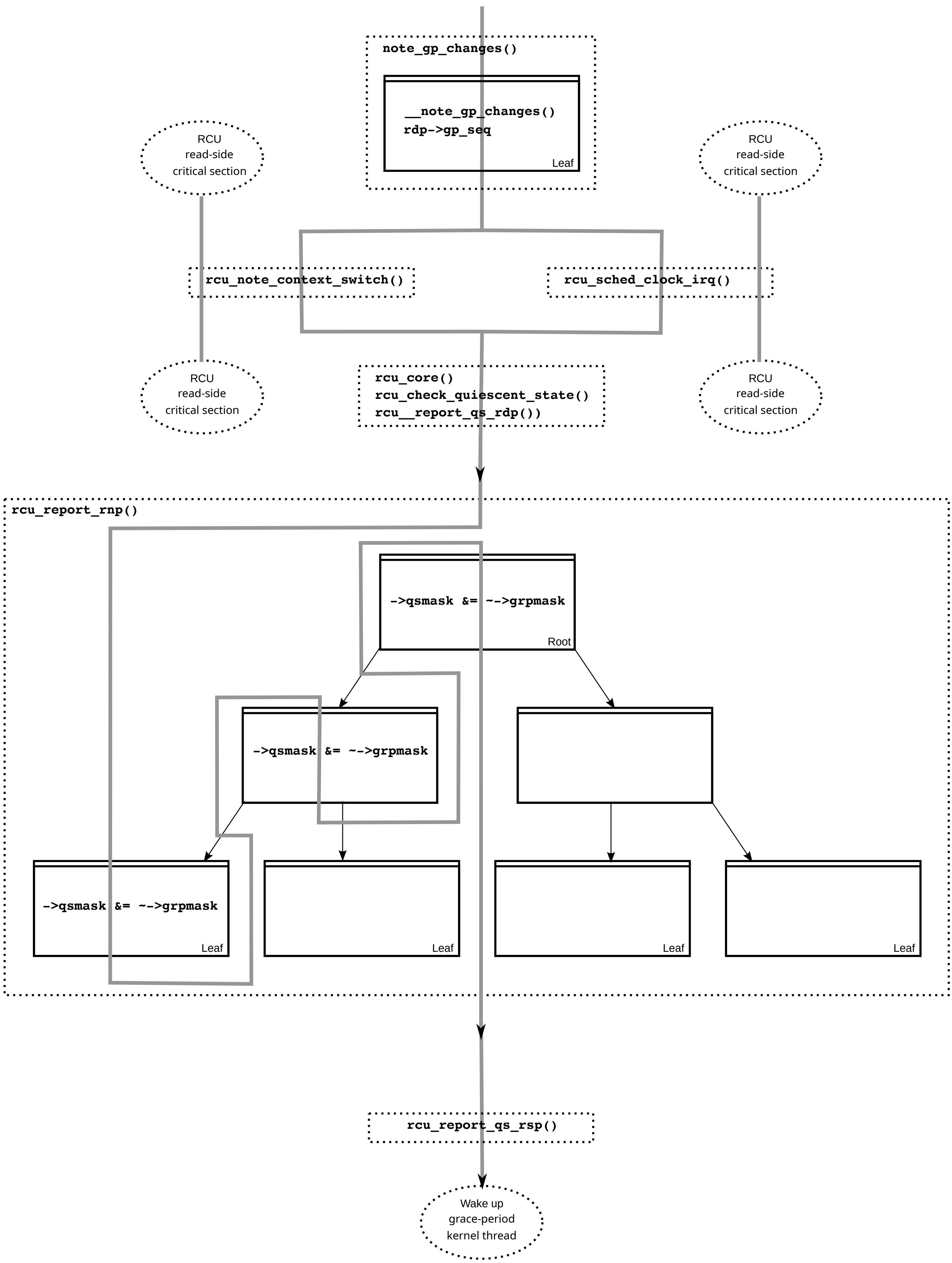
Answer:

It is indeed not necessary for the grace period to wait on such a critical section. However, it is permissible to wait on it. And it is furthermore important to wait on it, as this lazy approach is far more scalable than a "big bang" all-at-once grace-period start could possibly be.

If the CPU does a context switch, a quiescent state will be noted by `rcu_note_context_switch()` on the left. On the other hand, if the CPU takes a scheduler-clock interrupt while executing in usermode, a quiescent state will be noted by `rcu_sched_clock_irq()` on the right. Either way, the passage through a quiescent state will be noted in a per-CPU variable.

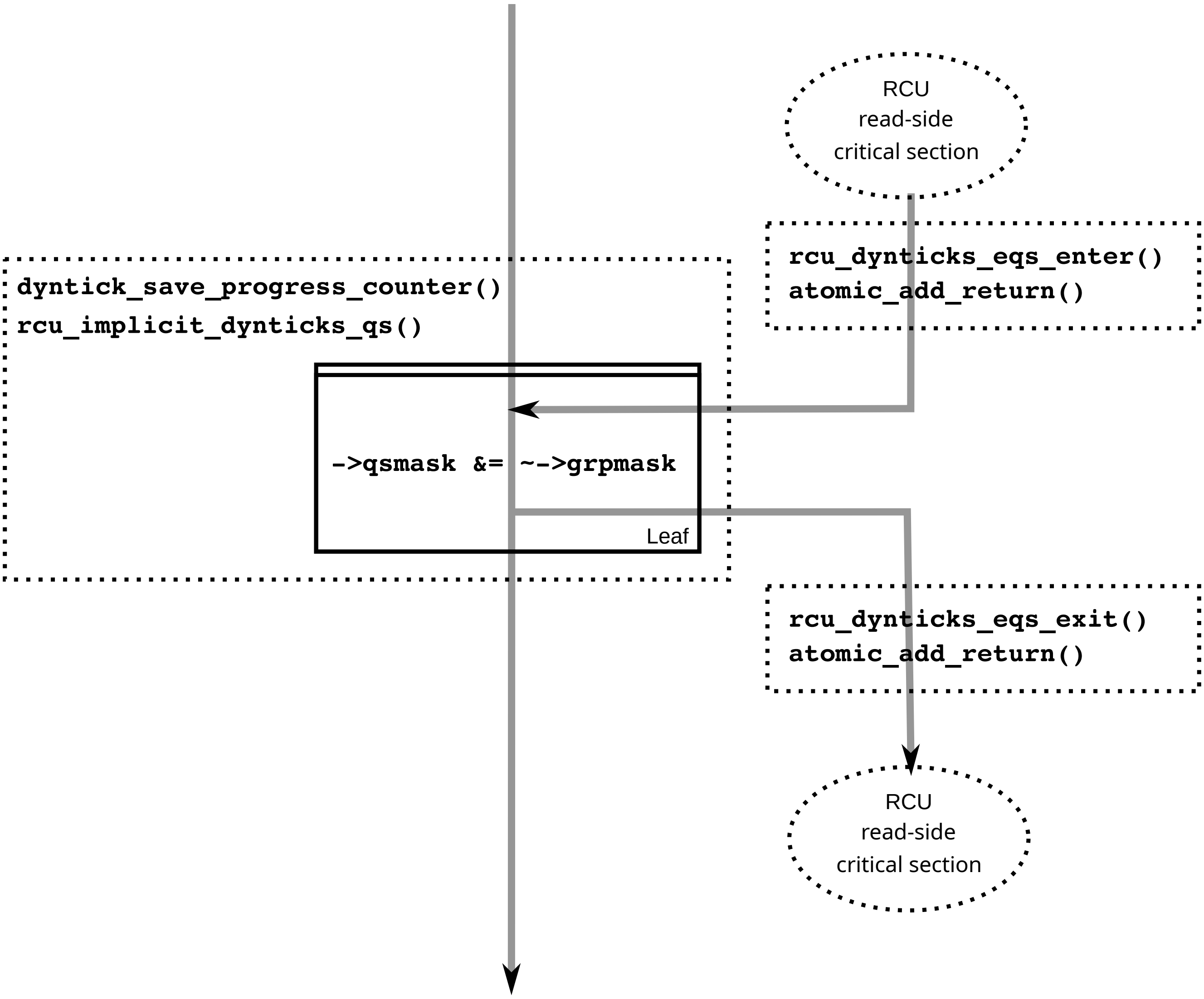
The next time an `RCU_SOFTIRQ` handler executes on this CPU (for example, after the next scheduler-clock interrupt), `rcu_core()` will invoke `rcu_check_quiescent_state()`, which will notice the recorded quiescent state, and invoke `rcu_report_qs_rdp()`. If `rcu_report_qs_rdp()` verifies that the quiescent state really does apply to the current grace period, it invokes `rcu_report_rnp()` which traverses up the `rcu_node` tree as shown at the bottom of the diagram, clearing bits from each `rcu_node` structure's `->qsmask` field, and propagating up the tree when the result is zero.

Note that traversal passes upwards out of a given `rcu_node` structure only if the current CPU is reporting the last quiescent state for the subtree headed by that `rcu_node` structure. A key point is that if a CPU's traversal stops at a given `rcu_node` structure, then there will be a later traversal by another CPU (or perhaps the same one) that proceeds upwards from that point, and the `rcu_node` `->lock` guarantees that the first CPU's quiescent state happens before the remainder of the second CPU's traversal. Applying this line of thought repeatedly shows that all CPUs' quiescent states happen before the last CPU traverses through the root `rcu_node` structure, the "last CPU" being the one that clears the last bit in the root `rcu_node` structure's `->qsmask` field.


note_gp_changes()
__note_gp_changes()
rdp->gp_seq
Leaf
RCU
read-side
critical section
RCU
read-side
critical section
rcu_note_context_switch()
rcu_sched_clock_irq()
RCU
read-side
critical section
rcu_core()
rcu_check_quiescent_state()
rcu__report_qs_rdp())
RCU
read-side
critical section
rcu_report_rnp()
->qsmask &= ~->grpmask
Root
->qsmask &= ~->grpmask
->qsmask &= ~->grpmask
Leaf
Leaf
Leaf
Leaf
rcu_report_qs_rsp()
Wake up
grace-period
kernel thread

Dynamic Tick Interface

Due to energy-efficiency considerations, RCU is forbidden from disturbing idle CPUs. CPUs are therefore required to notify RCU when entering or leaving idle state, which they do via fully ordered value-returning atomic operations on a per-CPU variable. The ordering effects are as shown below:

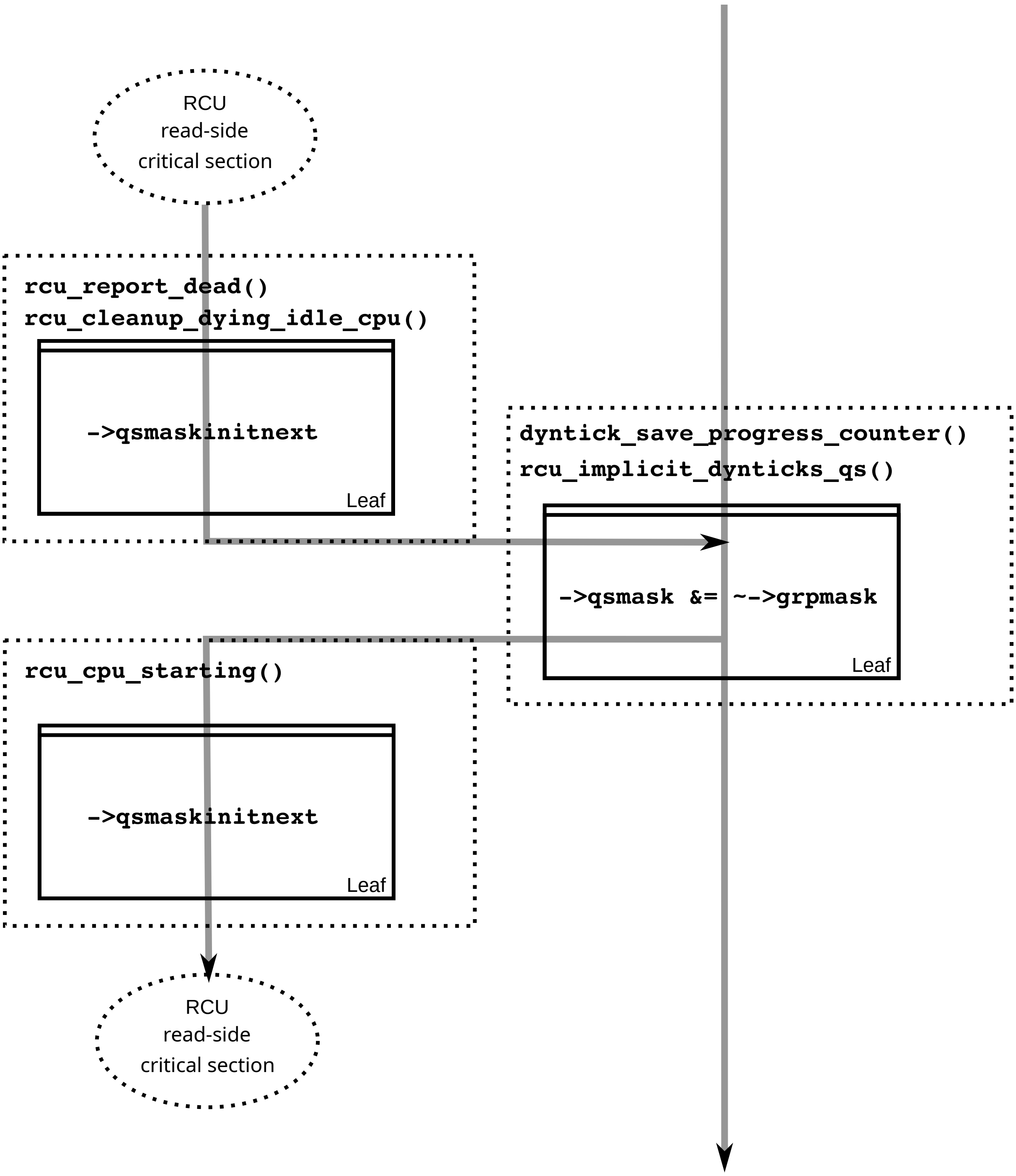


The RCU grace-period kernel thread samples the per-CPU idleness variable while holding the corresponding CPU's leaf `rcu_node` structure's `->lock`. This means that any RCU read-side critical sections that precede the idle period (the oval near the top of the diagram above) will happen before the end of the current grace period. Similarly, the beginning of the current grace period will happen before any RCU read-side critical sections that follow the idle period (the oval near the bottom of the diagram above).

Plumbing this into the full grace-period execution is described *below*.

CPU-Hotplug Interface

RCU is also forbidden from disturbing offline CPUs, which might well be powered off and removed from the system completely. CPUs are therefore required to notify RCU of their comings and goings as part of the corresponding CPU hotplug operations. The ordering effects are shown below:

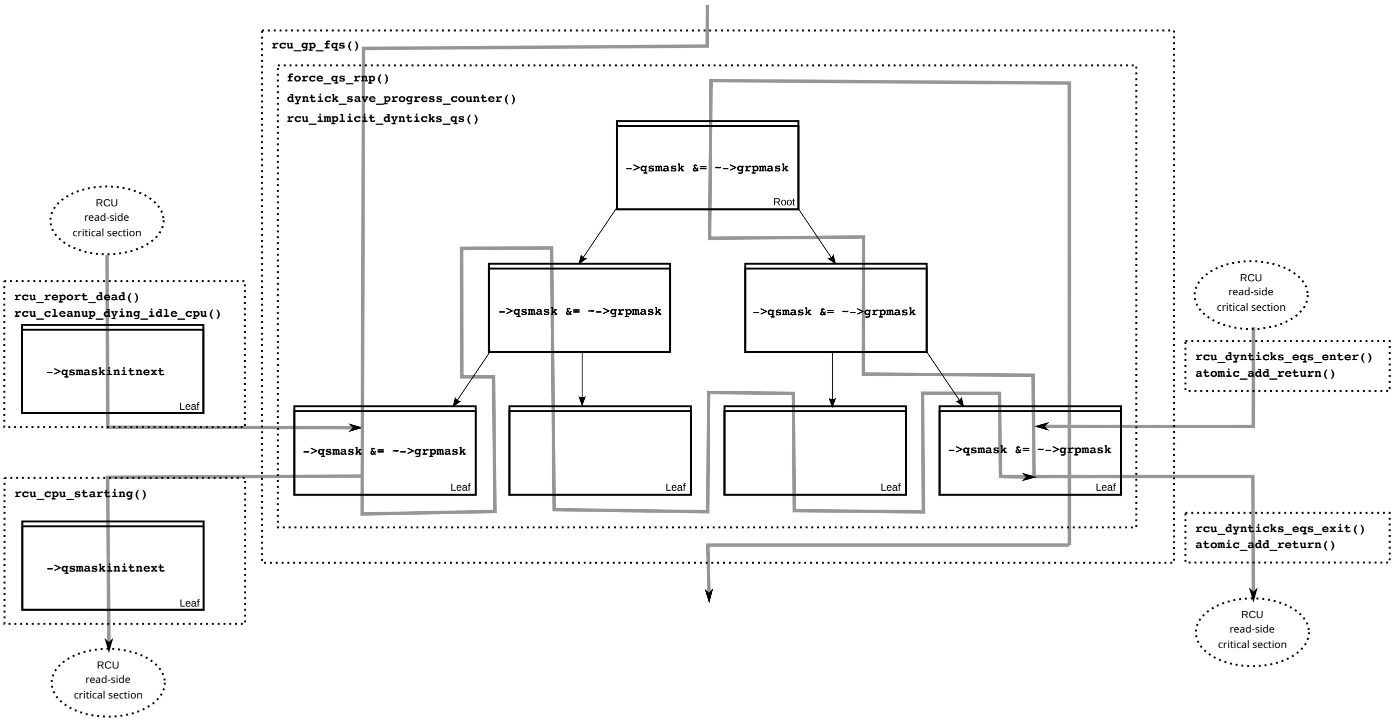


Because CPU hotplug operations are much less frequent than idle transitions, they are heavier weight, and thus acquire the CPU's leaf `rcu_node` structure's `->lock` and update this structure's `->qsmaskinitnext`. The RCU grace-period kernel thread samples this mask to detect CPUs having gone offline since the beginning of this grace period.

Plumbing this into the full grace-period execution is described *below*.

Forcing Quiescent States

As noted above, idle and offline CPUs cannot report their own quiescent states, and therefore the grace-period kernel thread must do the reporting on their behalf. This process is called "forcing quiescent states", it is repeated every few jiffies, and its ordering effects are shown below:



Each pass of quiescent state forcing is guaranteed to traverse the leaf `rcu_node` structures, and if there are no new quiescent states due to recently idled and/or offlined CPUs, then only the leaves are traversed. However, if there is a newly offlined CPU as illustrated on the left or a newly idled CPU as illustrated on the right, the corresponding quiescent state will be driven up towards the root. As with self-reported quiescent states, the upwards driving stops once it reaches an `rcu_node` structure that has quiescent states outstanding from other CPUs.

Quick Quiz:
The leftmost drive to root stopped before it reached the root `rcu_node` structure, which means that there are still CPUs subordinate to that structure on which the current grace period is waiting. Given that, how is it possible that the rightmost drive to root ended the grace period?
Answer:
Good analysis! It is in fact impossible in the absence of bugs in RCU. But this diagram is complex enough as it is, so simplicity overrode accuracy. You can think of it as poetic license, or you can think of it as misdirection that is resolved in the *stitched-together diagram*.

Grace-Period Cleanup

Grace-period cleanup first scans the `rcu_node` tree breadth-first advancing all the `->gp_seq` fields, then it advances the `rcu_state` structure's `->gp_seq` field. The ordering effects are shown below:

As indicated by the oval at the bottom of the diagram, once grace-period cleanup is complete, the next grace period can begin.

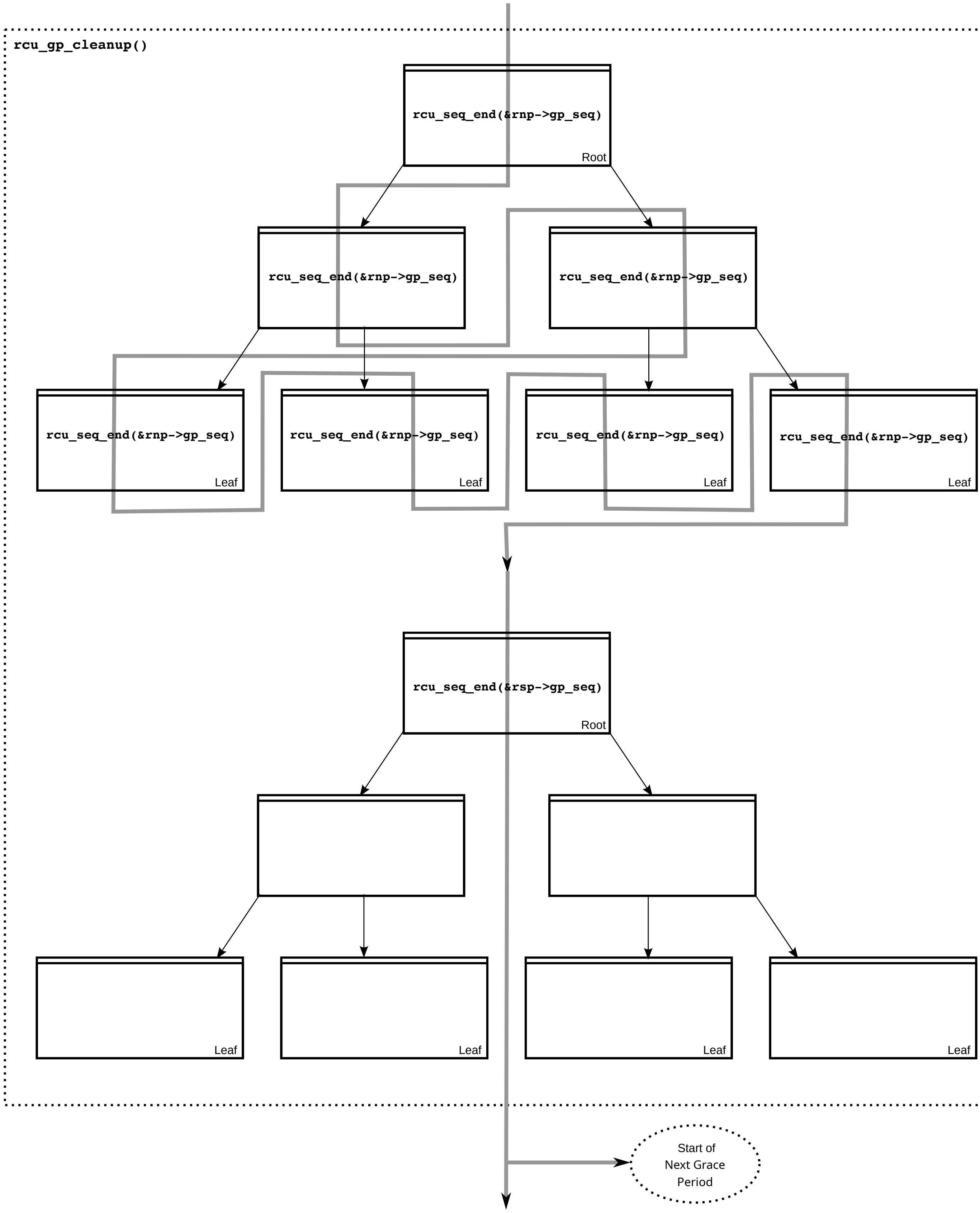
Quick Quiz:
But when precisely does the grace period end?
Answer:
There is no useful single point at which the grace period can be said to end. The earliest reasonable candidate is as soon as the last CPU has reported its quiescent state, but it may be some milliseconds before RCU becomes aware of this. The latest reasonable candidate is once the `rcu_state` structure's `->gp_seq` field has been updated, but it is quite possible that some CPUs have already completed phase two of their updates by that time. In short, if you are going to work with RCU, you need to learn to embrace uncertainty.

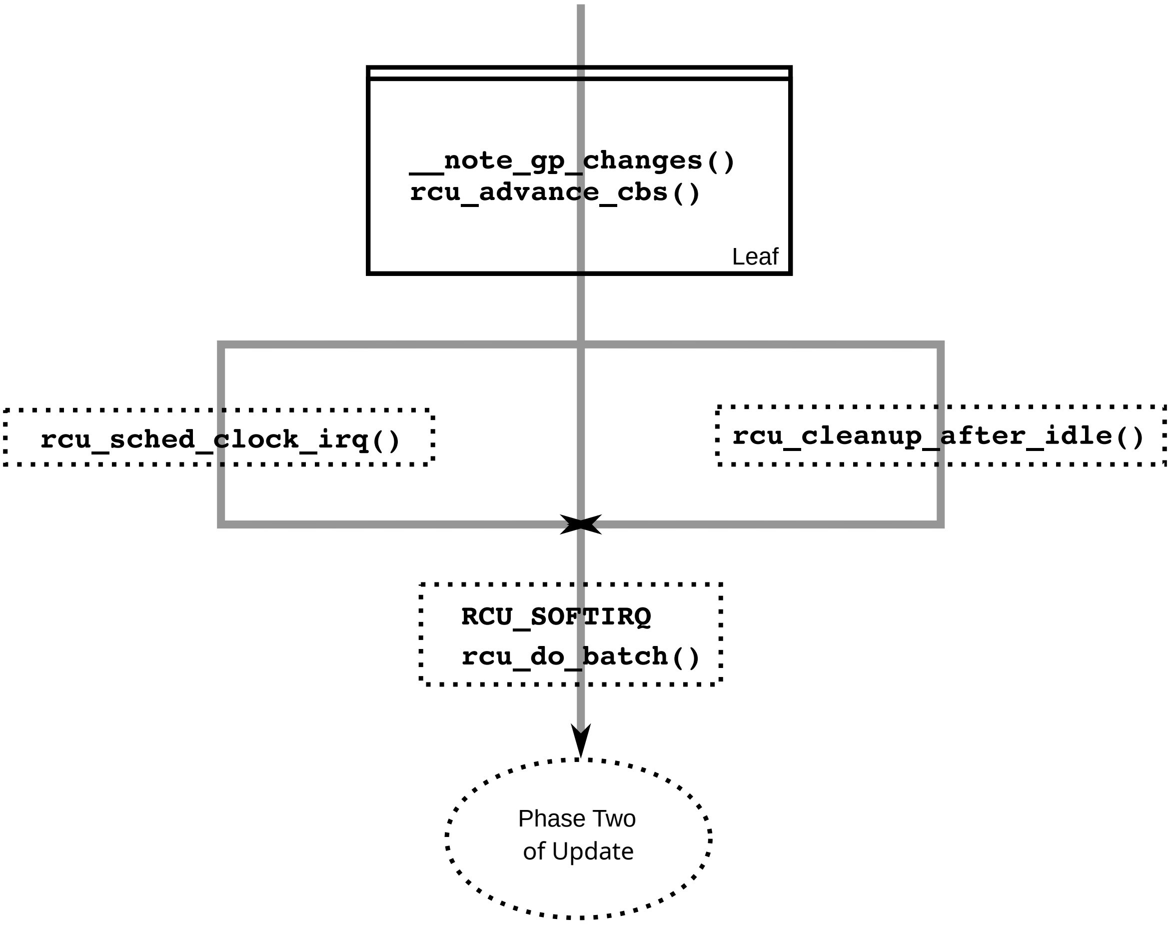
Callback Invocation

Once a given CPU's leaf `rcu_node` structure's `->gp_seq` field has been updated, that CPU can begin invoking its RCU callbacks that were waiting for this grace period to end. These callbacks are identified by `rcu_advance_cbs()`, which is usually invoked by `__note_gp_changes()`. As shown in the diagram below, this invocation can be triggered by the scheduling-clock interrupt (`rcu_sched_clock_irq()` on the left) or by idle entry (`rcu_cleanup_after_idle()` on the right, but only for kernels build with `CONFIG_RCU_FAST_NO_HZ=y`). Either way, `RCU_SOFTIRQ` is raised, which results in `rcu_do_batch()` invoking the callbacks, which in turn allows those callbacks to carry out (either directly or indirectly via wakeup) the needed phase-two processing for each update.

Please note that callback invocation can also be prompted by any number of corner-case code paths, for example, when a CPU notes that it has excessive numbers of callbacks queued. In all cases, the CPU acquires its leaf `rcu_node` structure's `->lock` before invoking callbacks, which preserves the required ordering against the newly completed grace period.

However, if the callback function communicates to other CPUs, for example, doing a wakeup, then it is that function's responsibility to maintain ordering. For example, if the callback function wakes up a task that runs on some other CPU, proper ordering must in place in both the callback function and the task being awakened. To see why this is important, consider the top half of the *grace-period cleanup* diagram. The callback might be running on a CPU corresponding to the leftmost leaf `rcu_node` structure, and awaken a task that is to run on a CPU corresponding to the rightmost leaf `rcu_node` structure, and the grace-period kernel thread might not yet have reached the rightmost leaf. In this case, the grace period's memory ordering might not yet have reached that CPU, so again the callback function and the awakened task must supply proper ordering.


rcu_gp_cleanup()
rcu_seq_end(&rnp->gp_seq)
Root
rcu_seq_end(&rnp->gp_seq)
rcu_seq_end(&rnp->gp_seq)
rcu_seq_end(&rnp->gp_seq)
Leaf
rcu_seq_end(&rnp->gp_seq)
Leaf
rcu_seq_end(&rnp->gp_seq)
Leaf
rcu_seq_end(&rnp->gp_seq)
Leaf
rcu_seq_end(&rsp->gp_seq)
Root
Leaf
Leaf
Leaf
Leaf
Start of
Next Grace
Period


__note_gp_changes()
rcu_advance_cbs()
Leaf
rcu_sched_clock_irq()
rcu_cleanup_after_idle()
RCU_SOFTIRQ
rcu_do_batch()
Phase Two
of Update

15.3.2 Putting It All Together

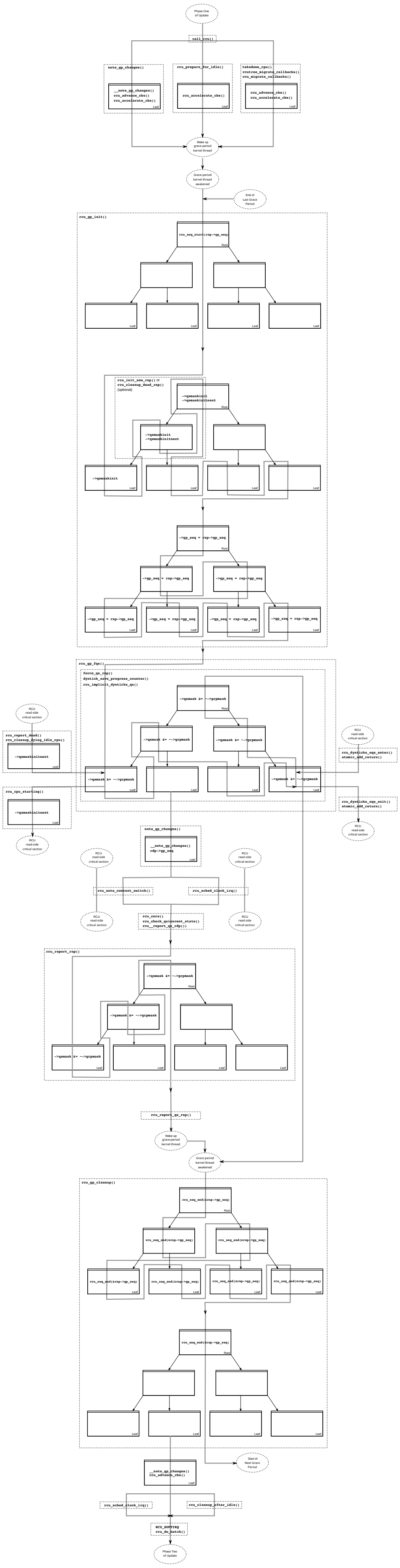
A stitched-together diagram is here:

15.3.3 Legal Statement

This work represents the view of the author 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.



CHAPTER

SIXTEEN

A TOUR THROUGH TREE_RCU'S EXPEDITED GRACE PERIODS

16.1 Introduction

This document describes RCU's expedited grace periods. Unlike RCU's normal grace periods, which accept long latencies to attain high efficiency and minimal disturbance, expedited grace periods accept lower efficiency and significant disturbance to attain shorter latencies.

There are two flavors of RCU (RCU-preempt and RCU-sched), with an earlier third RCU-bh flavor having been implemented in terms of the other two. Each of the two implementations is covered in its own section.

16.2 Expedited Grace Period Design

The expedited RCU grace periods cannot be accused of being subtle, given that they for all intents and purposes hammer every CPU that has not yet provided a quiescent state for the current expedited grace period. The one saving grace is that the hammer has grown a bit smaller over time: The old call to `try_stop_cpus()` has been replaced with a set of calls to `smp_call_function_single()`, each of which results in an IPI to the target CPU. The corresponding handler function checks the CPU's state, motivating a faster quiescent state where possible, and triggering a report of that quiescent state. As always for RCU, once everything has spent some time in a quiescent state, the expedited grace period has completed.

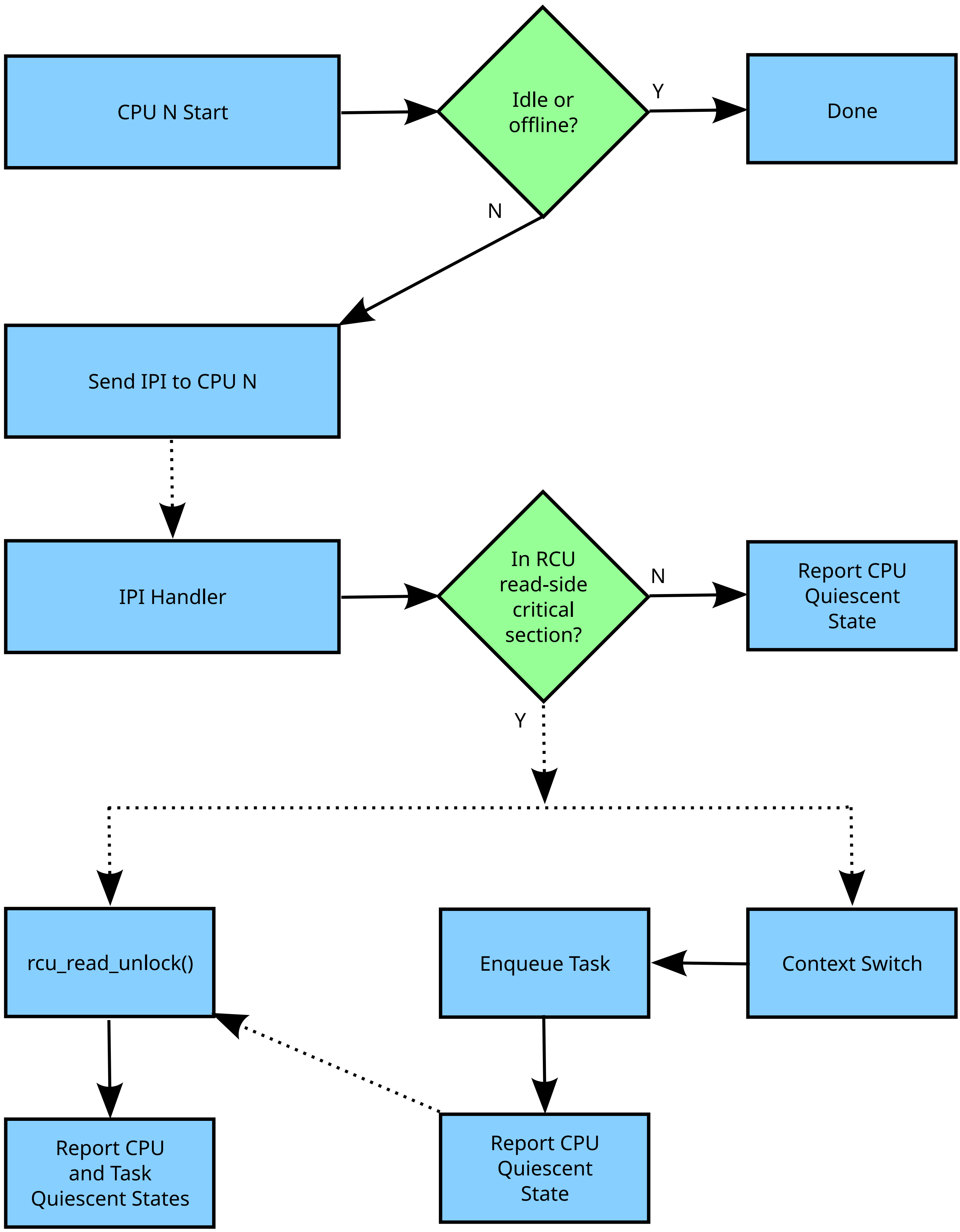
The details of the `smp_call_function_single()` handler's operation depend on the RCU flavor, as described in the following sections.

16.3 RCU-preempt Expedited Grace Periods

`CONFIG_PREEMPTION=y` kernels implement RCU-preempt. The overall flow of the handling of a given CPU by an RCU-preempt expedited grace period is shown in the following diagram:

The solid arrows denote direct action, for example, a function call. The dotted arrows denote indirect action, for example, an IPI or a state that is reached after some time.

If a given CPU is offline or idle, `synchronize_rcu_expedited()` will ignore it because idle and offline CPUs are already residing in quiescent states. Otherwise, the expedited grace period will use `smp_call_function_single()` to send the CPU an IPI, which is handled by `rcu_exp_handler()`.


CPU N Start
Idle or offline?
Y
Done
N
Send IPI to CPU N
IPI Handler
In RCU read-side critical section?
N
Report CPU Quiescent State
Y
rcu_read_unlock()
Enqueue Task
Context Switch
Report CPU and Task Quiescent States
Report CPU Quiescent State

However, because this is preemptible RCU, `rcu_exp_handler()` can check to see if the CPU is currently running in an RCU read-side critical section. If not, the handler can immediately report a quiescent state. Otherwise, it sets flags so that the outermost `rcu_read_unlock()` invocation will provide the needed quiescent-state report. This flag-setting avoids the previous forced preemption of all CPUs that might have RCU read-side critical sections. In addition, this flag-setting is done so as to avoid increasing the overhead of the common-case fastpath through the scheduler.

Again because this is preemptible RCU, an RCU read-side critical section can be preempted. When that happens, RCU will enqueue the task, which will the continue to block the current expedited grace period until it resumes and finds its outermost `rcu_read_unlock()`. The CPU will report a quiescent state just after enqueuing the task because the CPU is no longer blocking the grace period. It is instead the preempted task doing the blocking. The list of blocked tasks is managed by `rcu_preempt_ctxt_queue()`, which is called from `rcu_preempt_note_context_switch()`, which in turn is called from `rcu_note_context_switch()`, which in turn is called from the scheduler.

Quick Quiz:
Why not just have the expedited grace period check the state of all the CPUs? After all, that would avoid all those real-time-unfriendly IPIs.
Answer:
Because we want the RCU read-side critical sections to run fast, which means no memory barriers. Therefore, it is not possible to safely check the state from some other CPU. And even if it was possible to safely check the state, it would still be necessary to IPI the CPU to safely interact with the upcoming `rcu_read_unlock()` invocation, which means that the remote state testing would not help the worst-case latency that real-time applications care about. One way to prevent your real-time application from getting hit with these IPIs is to build your kernel with `CONFIG_NO_HZ_FULL=y`. RCU would then perceive the CPU running your application as being idle, and it would be able to safely detect that state without needing to IPI the CPU.

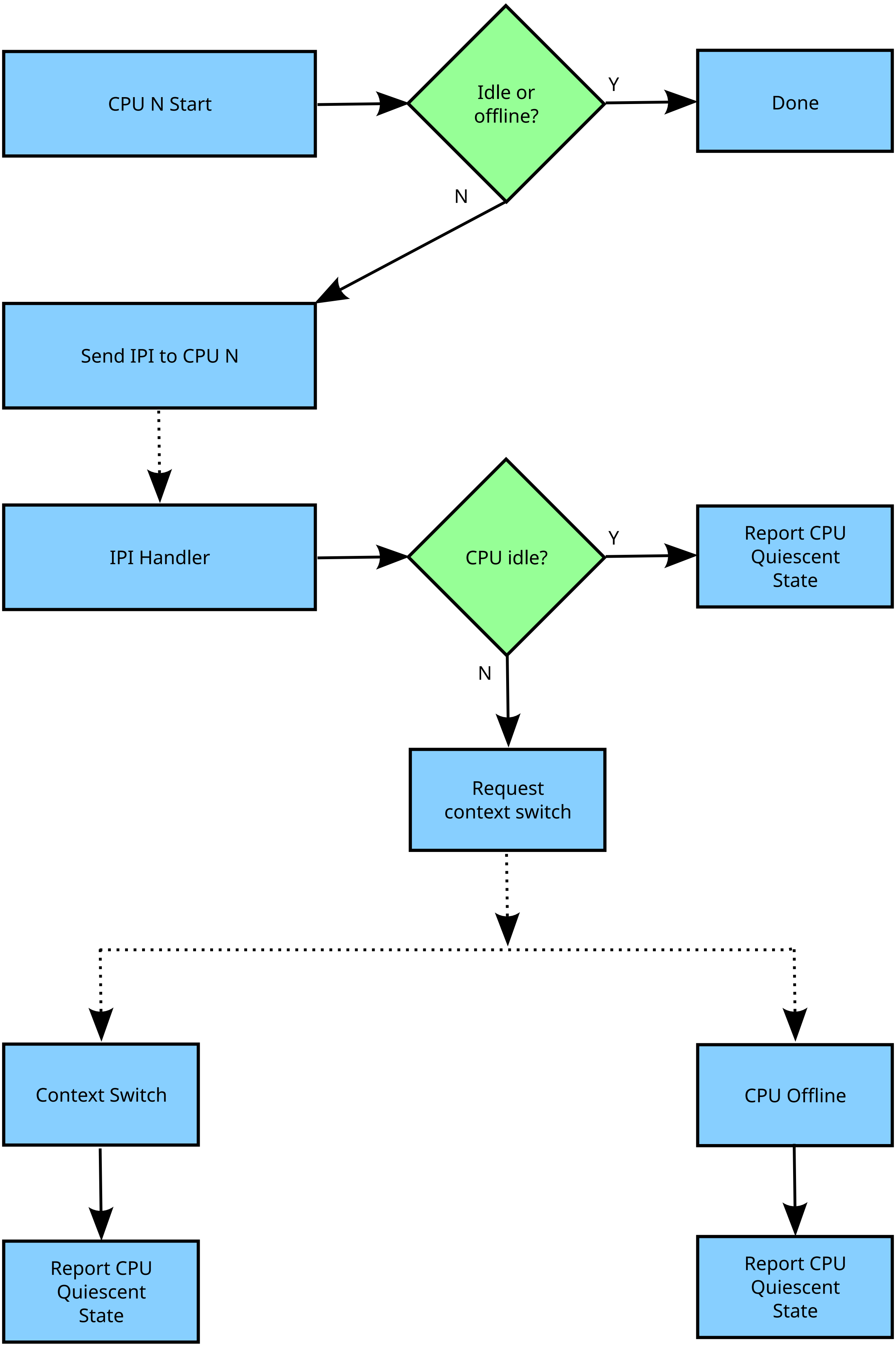
Please note that this is just the overall flow: Additional complications can arise due to races with CPUs going idle or offline, among other things.

16.3.1 RCU-sched Expedited Grace Periods

`CONFIG_PREEMPTION=n` kernels implement RCU-sched. The overall flow of the handling of a given CPU by an RCU-sched expedited grace period is shown in the following diagram:

As with RCU-preempt, RCU-sched's `synchronize_rcu_expedited()` ignores offline and idle CPUs, again because they are in remotely detectable quiescent states. However, because the `rcu_read_lock_sched()` and `rcu_read_unlock_sched()` leave no trace of their invocation, in general it is not possible to tell whether or not the current CPU is in an RCU read-side critical section. The best that RCU-sched's `rcu_exp_handler()` can do is to check for idle, on the off-chance that the CPU went idle while the IPI was in flight. If the CPU is idle, then `rcu_exp_handler()` reports the quiescent state.

Otherwise, the handler forces a future context switch by setting the NEED_RESCHED flag of the current task's thread flag and the CPU preempt counter. At the time of the context switch,


CPU N Start
Idle or offline?
Y
Done
N
Send IPI to CPU N
IPI Handler
CPU idle?
Y
Report CPU Quiescent State
N
Request context switch
Context Switch
CPU Offline
Report CPU Quiescent State
Report CPU Quiescent State

the CPU reports the quiescent state. Should the CPU go offline first, it will report the quiescent state at that time.

16.3.2 Expedited Grace Period and CPU Hotplug

The expedited nature of expedited grace periods require a much tighter interaction with CPU hotplug operations than is required for normal grace periods. In addition, attempting to IPI offline CPUs will result in splats, but failing to IPI online CPUs can result in too-short grace periods. Neither option is acceptable in production kernels.

The interaction between expedited grace periods and CPU hotplug operations is carried out at several levels:

1. The number of CPUs that have ever been online is tracked by the `rcu_state` structure's `->ncpus` field. The `rcu_state` structure's `->ncpus_snap` field tracks the number of CPUs that have ever been online at the beginning of an RCU expedited grace period. Note that this number never decreases, at least in the absence of a time machine.
2. The identities of the CPUs that have ever been online is tracked by the `rcu_node` structure's `->expmaskinitnext` field. The `rcu_node` structure's `->expmaskinit` field tracks the identities of the CPUs that were online at least once at the beginning of the most recent RCU expedited grace period. The `rcu_state` structure's `->ncpus` and `->ncpus_snap` fields are used to detect when new CPUs have come online for the first time, that is, when the `rcu_node` structure's `->expmaskinitnext` field has changed since the beginning of the last RCU expedited grace period, which triggers an update of each `rcu_node` structure's `->expmaskinit` field from its `->expmaskinitnext` field.
3. Each `rcu_node` structure's `->expmaskinit` field is used to initialize that structure's `->expmask` at the beginning of each RCU expedited grace period. This means that only those CPUs that have been online at least once will be considered for a given grace period.
4. Any CPU that goes offline will clear its bit in its leaf `rcu_node` structure's `->qsmaskinitnext` field, so any CPU with that bit clear can safely be ignored. However, it is possible for a CPU coming online or going offline to have this bit set for some time while `cpu_online` returns `false`.
5. For each non-idle CPU that RCU believes is currently online, the grace period invokes `smp_call_function_single()`. If this succeeds, the CPU was fully online. Failure indicates that the CPU is in the process of coming online or going offline, in which case it is necessary to wait for a short time period and try again. The purpose of this wait (or series of waits, as the case may be) is to permit a concurrent CPU-hotplug operation to complete.
6. In the case of RCU-sched, one of the last acts of an outgoing CPU is to invoke `rcu_report_dead()`, which reports a quiescent state for that CPU. However, this is likely paranoia-induced redundancy.

Quick Quiz:

Why all the dancing around with multiple counters and masks tracking CPUs that were once online? Why not just have a single set of masks tracking the currently online CPUs and be done with it?

Answer:

Maintaining single set of masks tracking the online CPUs *sounds* easier, at least until you try working out all the race conditions between grace-period initialization and CPU-hotplug operations. For example, suppose initialization is progressing down the tree while a CPU-offline operation is progressing up the tree. This situation can result in bits set at the top of the tree that have no counterparts at the bottom of the tree. Those bits will never be cleared, which will result in grace-period hangs. In short, that way lies madness, to say nothing of a great many bugs, hangs, and deadlocks. In contrast, the current multi-mask multi-counter scheme ensures that grace-period initialization will always see consistent masks up and down the tree, which brings significant simplifications over the single-mask method.

This is an instance of deferring work in order to avoid synchronization. Lazily recording CPU-hotplug events at the beginning of the next grace period greatly simplifies maintenance of the CPU-tracking bitmasks in the `rcu_node` tree.

16.3.3 Expedited Grace Period Refinements

Idle-CPU Checks

Each expedited grace period checks for idle CPUs when initially forming the mask of CPUs to be IPIed and again just before IPIing a CPU (both checks are carried out by `sync_rcu_exp_select_cpus()`). If the CPU is idle at any time between those two times, the CPU will not be IPIed. Instead, the task pushing the grace period forward will include the idle CPUs in the mask passed to `rcu_report_exp_cpu_mult()`.

For RCU-sched, there is an additional check: If the IPI has interrupted the idle loop, then `rcu_exp_handler()` invokes `rcu_report_exp_rdp()` to report the corresponding quiescent state.

For RCU-preempt, there is no specific check for idle in the IPI handler (`rcu_exp_handler()`), but because RCU read-side critical sections are not permitted within the idle loop, if `rcu_exp_handler()` sees that the CPU is within RCU read-side critical section, the CPU cannot possibly be idle. Otherwise, `rcu_exp_handler()` invokes `rcu_report_exp_rdp()` to report the corresponding quiescent state, regardless of whether or not that quiescent state was due to the CPU being idle.

In summary, RCU expedited grace periods check for idle when building the bitmask of CPUs that must be IPIed, just before sending each IPI, and (either explicitly or implicitly) within the IPI handler.

Batching via Sequence Counter

If each grace-period request was carried out separately, expedited grace periods would have abysmal scalability and problematic high-load characteristics. Because each grace-period operation can serve an unlimited number of updates, it is important to *batch* requests, so that a single expedited grace-period operation will cover all requests in the corresponding batch.

This batching is controlled by a sequence counter named `->expedited_sequence` in the `rcu_state` structure. This counter has an odd value when there is an expedited grace period in progress and an even value otherwise, so that dividing the counter value by two gives the number of completed grace periods. During any given update request, the counter must transition from even to odd and then back to even, thus indicating that a grace period has elapsed. Therefore, if the initial value of the counter is `s`, the updater must wait until the counter reaches at least the value `(s+3)&~0x1`. This counter is managed by the following access functions:

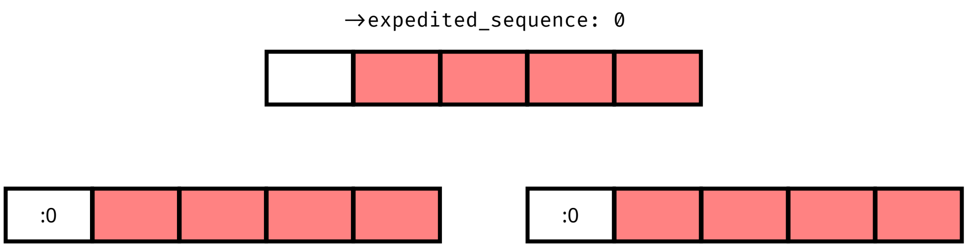
1. `rcu_exp_gp_seq_start()`, which marks the start of an expedited grace period.
2. `rcu_exp_gp_seq_end()`, which marks the end of an expedited grace period.
3. `rcu_exp_gp_seq_snap()`, which obtains a snapshot of the counter.
4. `rcu_exp_gp_seq_done()`, which returns `true` if a full expedited grace period has elapsed since the corresponding call to `rcu_exp_gp_seq_snap()`.

Again, only one request in a given batch need actually carry out a grace-period operation, which means there must be an efficient way to identify which of many concurrent requests will initiate the grace period, and that there be an efficient way for the remaining requests to wait for that grace period to complete. However, that is the topic of the next section.

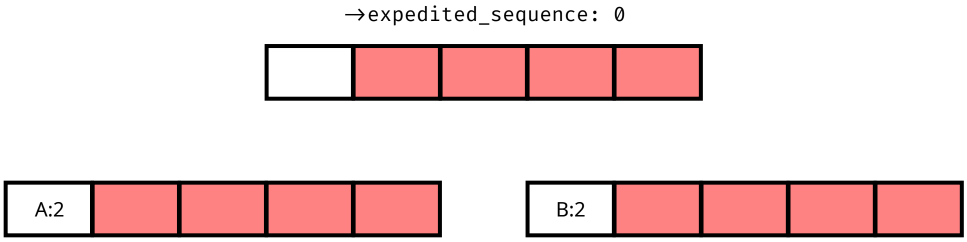
Funnel Locking and Wait/Wakeup

The natural way to sort out which of a batch of updaters will initiate the expedited grace period is to use the `rcu_node` combining tree, as implemented by the `exp_funnel_lock()` function. The first updater corresponding to a given grace period arriving at a given `rcu_node` structure records its desired grace-period sequence number in the `->exp_seq_rq` field and moves up to the next level in the tree. Otherwise, if the `->exp_seq_rq` field already contains the sequence number for the desired grace period or some later one, the updater blocks on one of four wait queues in the `->exp_wq[]` array, using the second-from-bottom and third-from bottom bits as an index. An `->exp_lock` field in the `rcu_node` structure synchronizes access to these fields.

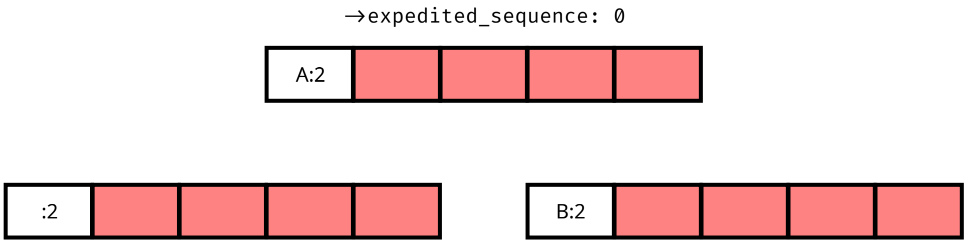
An empty `rcu_node` tree is shown in the following diagram, with the white cells representing the `->exp_seq_rq` field and the red cells representing the elements of the `->exp_wq[]` array.



The next diagram shows the situation after the arrival of Task A and Task B at the leftmost and rightmost leaf `rcu_node` structures, respectively. The current value of the `rcu_state` structure's `->expedited_sequence` field is zero, so adding three and clearing the bottom bit results in the value two, which both tasks record in the `->exp_seq_rq` field of their respective `rcu_node` structures:



Each of Tasks A and B will move up to the root `rcu_node` structure. Suppose that Task A wins, recording its desired grace-period sequence number and resulting in the state shown below:

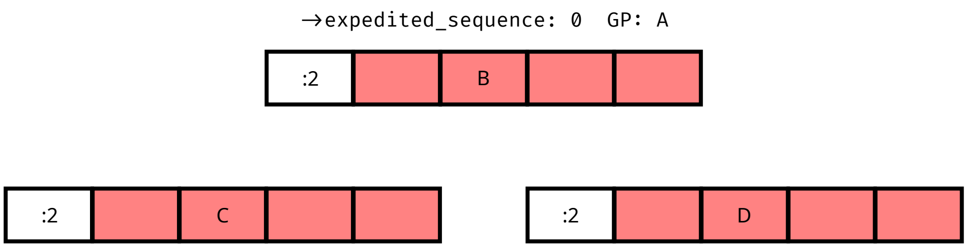


Task A now advances to initiate a new grace period, while Task B moves up to the root `rcu_node` structure, and, seeing that its desired sequence number is already recorded, blocks on `->exp_wq[1]`.

Quick Quiz:
Why `->exp_wq[1]`? Given that the value of these tasks' desired sequence number is two, so shouldn't they instead block on `->exp_wq[2]`?

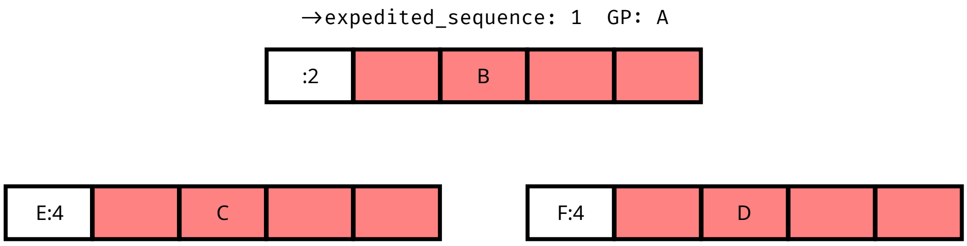
Answer:
No. Recall that the bottom bit of the desired sequence number indicates whether or not a grace period is currently in progress. It is therefore necessary to shift the sequence number right one bit position to obtain the number of the grace period. This results in `->exp_wq[1]`.

If Tasks C and D also arrive at this point, they will compute the same desired grace-period sequence number, and see that both leaf `rcu_node` structures already have that value recorded. They will therefore block on their respective `rcu_node` structures' `->exp_wq[1]` fields, as shown below:

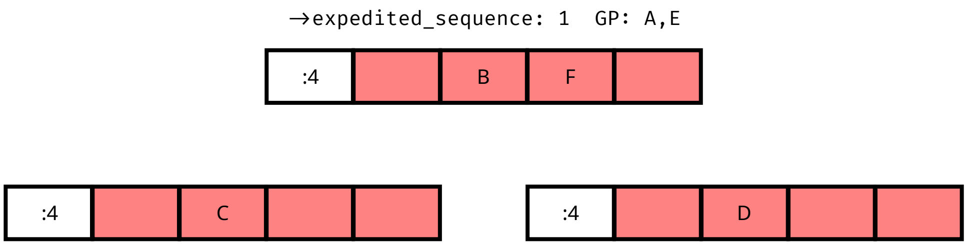


Task A now acquires the `rcu_state` structure's `->exp_mutex` and initiates the grace period, which increments `->expedited_sequence`. Therefore, if Tasks E and F arrive, they will compute a desired sequence number of 4 and will record this value as shown below:

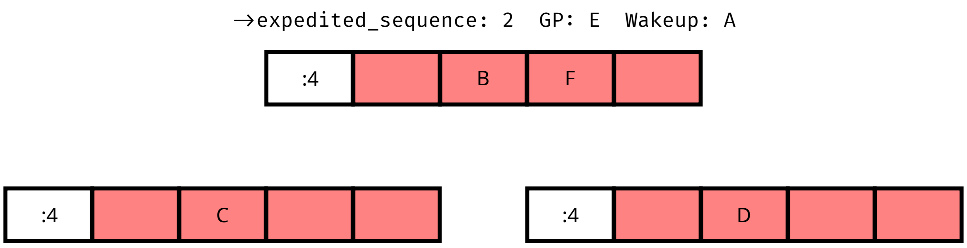
Tasks E and F will propagate up the `rcu_node` combining tree, with Task F blocking on the root



rcu_node structure and Task E wait for Task A to finish so that it can start the next grace period. The resulting state is as shown below:



Once the grace period completes, Task A starts waking up the tasks waiting for this grace period to complete, increments the ->expedited_sequence, acquires the ->exp_wake_mutex and then releases the ->exp_mutex. This results in the following state:

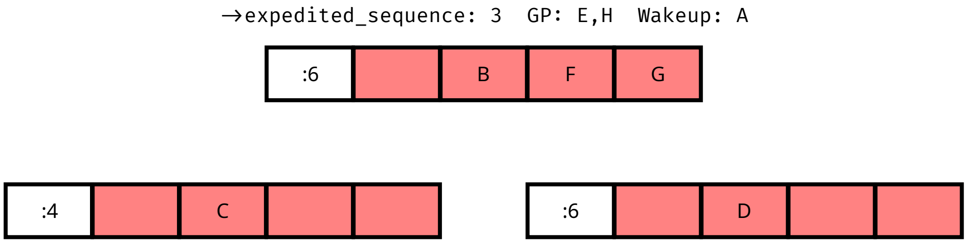


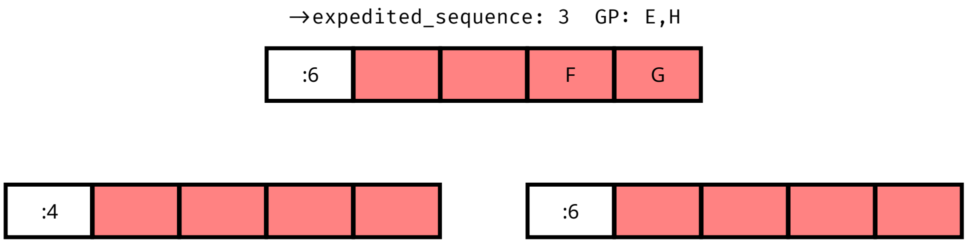
Task E can then acquire ->exp_mutex and increment ->expedited_sequence to the value three. If new tasks G and H arrive and moves up the combining tree at the same time, the state will be as follows:

Note that three of the root rcu_node structure's waitqueues are now occupied. However, at some point, Task A will wake up the tasks blocked on the ->exp_wq waitqueues, resulting in the following state:

Execution will continue with Tasks E and H completing their grace periods and carrying out their wakeups.

Quick Quiz:
What happens if Task A takes so long to do its wakeups that Task E's grace period completes?
Answer:
Then Task E will block on the ->exp_wake_mutex, which will also prevent it from releasing ->exp_mutex, which in turn will prevent the next grace period from starting. This last is important in preventing overflow of the ->exp_wq[] array.





Use of Workqueues

In earlier implementations, the task requesting the expedited grace period also drove it to completion. This straightforward approach had the disadvantage of needing to account for POSIX signals sent to user tasks, so more recent implementations use the Linux kernel's workqueues (see Documentation/core-api/workqueue.rst).

The requesting task still does counter snapshotting and funnel-lock processing, but the task reaching the top of the funnel lock does a `schedule_work()` (from `_synchronize_rcu_expedited()` so that a workqueue kthread does the actual grace-period processing. Because workqueue kthreads do not accept POSIX signals, grace-period-wait processing need not allow for POSIX signals. In addition, this approach allows wakeups for the previous expedited grace period to be overlapped with processing for the next expedited grace period. Because there are only four sets of waitqueues, it is necessary to ensure that the previous grace period's wakeups complete before the next grace period's wakeups start. This is handled by having the `->exp_mutex` guard expedited grace-period processing and the `->exp_wake_mutex` guard wakeups. The key point is that the `->exp_mutex` is not released until the first wakeup is complete, which means that the `->exp_wake_mutex` has already been acquired at that point. This approach ensures that the previous grace period's wakeups can be carried out while the current grace period is in process, but that these wakeups will complete before the next grace period starts. This means that only three waitqueues are required, guaranteeing that the four that are provided are sufficient.

Stall Warnings

Expediting grace periods does nothing to speed things up when RCU readers take too long, and therefore expedited grace periods check for stalls just as normal grace periods do.

Quick Quiz:
But why not just let the normal grace-period machinery detect the stalls, given that a given reader must block both normal and expedited grace periods?
Answer:
Because it is quite possible that at a given time there is no normal grace period in progress, in which case the normal grace period cannot emit a stall warning.

The `synchronize_sched_expedited_wait()` function loops waiting for the expedited grace period to end, but with a timeout set to the current RCU CPU stall-warning time. If this time is exceeded, any CPUs or `rcu_node` structures blocking the current grace period are printed. Each stall warning results in another pass through the loop, but the second and subsequent passes use longer stall times.

Mid-boot operation

The use of workqueues has the advantage that the expedited grace-period code need not worry about POSIX signals. Unfortunately, it has the corresponding disadvantage that workqueues cannot be used until they are initialized, which does not happen until some time after the scheduler spawns the first task. Given that there are parts of the kernel that really do want to execute grace periods during this mid-boot "dead zone", expedited grace periods must do something else during this time.

What they do is to fall back to the old practice of requiring that the requesting task drive the expedited grace period, as was the case before the use of workqueues. However, the requesting task is only required to drive the grace period during the mid-boot dead zone. Before mid-boot, a synchronous grace period is a no-op. Some time after mid-boot, workqueues are used.

Non-expedited non-SRCU synchronous grace periods must also operate normally during mid-boot. This is handled by causing non-expedited grace periods to take the expedited code path during mid-boot.

The current code assumes that there are no POSIX signals during the mid-boot dead zone. However, if an overwhelming need for POSIX signals somehow arises, appropriate adjustments can be made to the expedited stall-warning code. One such adjustment would reinstate the pre-workqueue stall-warning checks, but only during the mid-boot dead zone.

With this refinement, synchronous grace periods can now be used from task context pretty much any time during the life of the kernel. That is, aside from some points in the suspend, hibernate, or shutdown code path.

Summary

Expedited grace periods use a sequence-number approach to promote batching, so that a single grace-period operation can serve numerous requests. A funnel lock is used to efficiently identify the one task out of a concurrent group that will request the grace period. All members of the group will block on waitqueues provided in the `rcu_node` structure. The actual grace-period processing is carried out by a workqueue.

CPU-hotplug operations are noted lazily in order to prevent the need for tight synchronization between expedited grace periods and CPU-hotplug operations. The dyntick-idle counters are used to avoid sending IPIs to idle CPUs, at least in the common case. RCU-preempt and RCU-sched use different IPI handlers and different code to respond to the state changes carried out by those handlers, but otherwise use common code.

Quiescent states are tracked using the `rcu_node` tree, and once all necessary quiescent states have been reported, all tasks waiting on this expedited grace period are awakened. A pair of mutexes are used to allow one grace period's wakeups to proceed concurrently with the next grace period's processing.

This combination of mechanisms allows expedited grace periods to run reasonably efficiently. However, for non-time-critical tasks, normal grace periods should be used instead because their longer duration permits much higher degrees of batching, and thus much lower per-request overheads.

CHAPTER SEVENTEEN

A TOUR THROUGH RCU'S REQUIREMENTS

Author: Paul E. McKenney

The initial version of this document appeared in the LWN on those articles: part 1, part 2, and part 3.

17.1 Introduction

Read-copy update (RCU) is a synchronization mechanism that is often used as a replacement for reader-writer locking. RCU is unusual in that updaters do not block readers, which means that RCU's read-side primitives can be exceedingly fast and scalable. In addition, updaters can make useful forward progress concurrently with readers. However, all this concurrency between RCU readers and updaters does raise the question of exactly what RCU readers are doing, which in turn raises the question of exactly what RCU's requirements are.

This document therefore summarizes RCU's requirements, and can be thought of as an informal, high-level specification for RCU. It is important to understand that RCU's specification is primarily empirical in nature; in fact, I learned about many of these requirements the hard way. This situation might cause some consternation, however, not only has this learning process been a lot of fun, but it has also been a great privilege to work with so many people willing to apply technologies in interesting new ways.

All that aside, here are the categories of currently known RCU requirements:

1. *Fundamental Requirements*
2. *Fundamental Non-Requirements*
3. *Parallelism Facts of Life*
4. *Quality-of-Implementation Requirements*
5. *Linux Kernel Complications*
6. *Software-Engineering Requirements*
7. *Other RCU Flavors*
8. *Possible Future Changes*

This is followed by a *summary*, however, the answers to each quick quiz immediately follows the quiz. Select the big white space with your mouse to see the answer.

17.2 Fundamental Requirements

RCU's fundamental requirements are the closest thing RCU has to hard mathematical requirements. These are:

1. *Grace-Period Guarantee*
2. *Publish/Subscribe Guarantee*
3. *Memory-Barrier Guarantees*
4. *RCU Primitives Guaranteed to Execute Unconditionally*
5. *Guaranteed Read-to-Write Upgrade*

17.2.1 Grace-Period Guarantee

RCU's grace-period guarantee is unusual in being premeditated: Jack Slingwine and I had this guarantee firmly in mind when we started work on RCU (then called "rclock") in the early 1990s. That said, the past two decades of experience with RCU have produced a much more detailed understanding of this guarantee.

RCU's grace-period guarantee allows updaters to wait for the completion of all pre-existing RCU read-side critical sections. An RCU read-side critical section begins with the marker rcu_read_lock() and ends with the marker rcu_read_unlock(). These markers may be nested, and RCU treats a nested set as one big RCU read-side critical section. Production-quality implementations of rcu_read_lock() and rcu_read_unlock() are extremely lightweight, and in fact have exactly zero overhead in Linux kernels built for production use with `CONFIG_PREEMPTION=n`.

This guarantee allows ordering to be enforced with extremely low overhead to readers, for example:

```
 1 int x, y;
 2
 3 void thread0(void)
 4 {
 5   rcu_read_lock();
 6   r1 = READ_ONCE(x);
 7   r2 = READ_ONCE(y);
 8   rcu_read_unlock();
 9 }
10
11 void thread1(void)
12 {
13   WRITE_ONCE(x, 1);
14   synchronize_rcu();
15   WRITE_ONCE(y, 1);
16 }
```

Because the synchronize_rcu() on line 14 waits for all pre-existing readers, any instance of thread0() that loads a value of zero from `x` must complete before thread1() stores to `y`, so that instance must also load a value of zero from `y`. Similarly, any instance of thread0() that loads a value of one from `y` must have started after the synchronize_rcu() started, and must therefore also load a value of one from `x`. Therefore, the outcome:

```
(r1 == 0 && r2 == 1)
```

cannot happen.

Quick Quiz:
Wait a minute! You said that updaters can make useful forward progress concurrently with readers, but pre-existing readers will block synchronize_rcu()!!! Just who are you trying to fool???

Answer:
First, if updaters do not wish to be blocked by readers, they can use call_rcu() or kfree_rcu(), which will be discussed later. Second, even when using synchronize_rcu(), the other update-side code does run concurrently with readers, whether pre-existing or not.

This scenario resembles one of the first uses of RCU in DYNIX/ptx, which managed a distributed lock manager's transition into a state suitable for handling recovery from node failure, more or less as follows:

```
#define STATE_NORMAL        0
#define STATE_WANT_RECOVERY 1
#define STATE_RECOVERING    2
#define STATE_WANT_NORMAL   3

int state = STATE_NORMAL;

void do_something_dlm(void)
{
  int state_snap;

  rcu_read_lock();
  state_snap = READ_ONCE(state);
  if (state_snap == STATE_NORMAL)
    do_something();
  else
    do_something_carefully();
  rcu_read_unlock();
}

void start_recovery(void)
{
  WRITE_ONCE(state, STATE_WANT_RECOVERY);
  synchronize_rcu();
  WRITE_ONCE(state, STATE_RECOVERING);
  recovery();
  WRITE_ONCE(state, STATE_WANT_NORMAL);
  synchronize_rcu();
  WRITE_ONCE(state, STATE_NORMAL);
}
```

The RCU read-side critical section in do_something_dlm() works with the synchronize_rcu() in start_recovery() to guarantee that do_something() never runs concurrently with recovery(), but with little or no synchronization overhead in do_something_dlm().

Quick Quiz:
Why is the synchronize_rcu() on line 28 needed?
Answer:
Without that extra grace period, memory reordering could result in do_something_dlm() executing do_something() concurrently with the last bits of recovery().

In order to avoid fatal problems such as deadlocks, an RCU read-side critical section must not contain calls to synchronize_rcu(). Similarly, an RCU read-side critical section must not contain anything that waits, directly or indirectly, on completion of an invocation of synchronize_rcu().

Although RCU's grace-period guarantee is useful in and of itself, with quite a few use cases, it would be good to be able to use RCU to coordinate read-side access to linked data structures. For this, the grace-period guarantee is not sufficient, as can be seen in function add_gp_buggy() below. We will look at the reader's code later, but in the meantime, just think of the reader as locklessly picking up the `gp` pointer, and, if the value loaded is non-`NULL`, locklessly accessing the `->a` and `->b` fields.

```
bool add_gp_buggy(int a, int b)
{
  p = kmalloc(sizeof(*p), GFP_KERNEL);
  if (!p)
    return -ENOMEM;
  spin_lock(&gp_lock);
  if (rcu_access_pointer(gp)) {
    spin_unlock(&gp_lock);
    return false;
  }
  p->a = a;
  p->b = a;
  gp = p; /* ORDERING BUG */
  spin_unlock(&gp_lock);
  return true;
}
```

The problem is that both the compiler and weakly ordered CPUs are within their rights to reorder this code as follows:

```
bool add_gp_buggy_optimized(int a, int b)
{
  p = kmalloc(sizeof(*p), GFP_KERNEL);
  if (!p)
    return -ENOMEM;
  spin_lock(&gp_lock);
  if (rcu_access_pointer(gp)) {
    spin_unlock(&gp_lock);
    return false;
  }
  gp = p; /* ORDERING BUG */
  p->a = a;
```

```
13   p->b = a;
14   spin_unlock(&gp_lock);
15   return true;
16 }
```

If an RCU reader fetches `gp` just after `add_gp_buggy_optimized` executes line 11, it will see garbage in the `->a` and `->b` fields. And this is but one of many ways in which compiler and hardware optimizations could cause trouble. Therefore, we clearly need some way to prevent the compiler and the CPU from reordering in this manner, which brings us to the publish-subscribe guarantee discussed in the next section.

17.2.2 Publish/Subscribe Guarantee

RCU's publish-subscribe guarantee allows data to be inserted into a linked data structure without disrupting RCU readers. The updater uses rcu_assign_pointer() to insert the new data, and readers use rcu_dereference() to access data, whether new or old. The following shows an example of insertion:

```
 1 bool add_gp(int a, int b)
 2 {
 3   p = kmalloc(sizeof(*p), GFP_KERNEL);
 4   if (!p)
 5     return -ENOMEM;
 6   spin_lock(&gp_lock);
 7   if (rcu_access_pointer(gp)) {
 8     spin_unlock(&gp_lock);
 9     return false;
10   }
11   p->a = a;
12   p->b = a;
13   rcu_assign_pointer(gp, p);
14   spin_unlock(&gp_lock);
15   return true;
16 }
```

The rcu_assign_pointer() on line 13 is conceptually equivalent to a simple assignment statement, but also guarantees that its assignment will happen after the two assignments in lines 11 and 12, similar to the C11 `memory_order_release` store operation. It also prevents any number of "interesting" compiler optimizations, for example, the use of `gp` as a scratch location immediately preceding the assignment.

Quick Quiz:

But rcu_assign_pointer() does nothing to prevent the two assignments to `p->a` and `p->b` from being reordered. Can't that also cause problems?

Answer:

No, it cannot. The readers cannot see either of these two fields until the assignment to `gp`, by which time both fields are fully initialized. So reordering the assignments to `p->a` and `p->b` cannot possibly cause any problems.

It is tempting to assume that the reader need not do anything special to control its accesses to the RCU-protected data, as shown in do_something_gp_buggy() below:

```
bool do_something_gp_buggy(void)
{
  rcu_read_lock();
  p = gp;  /* OPTIMIZATIONS GALORE!!! */
  if (p) {
    do_something(p->a, p->b);
    rcu_read_unlock();
    return true;
  }
  rcu_read_unlock();
  return false;
}
```

However, this temptation must be resisted because there are a surprisingly large number of ways that the compiler (or weak ordering CPUs like the DEC Alpha) can trip this code up. For but one example, if the compiler were short of registers, it might choose to refetch from `gp` rather than keeping a separate copy in `p` as follows:

```
bool do_something_gp_buggy_optimized(void)
{
  rcu_read_lock();
  if (gp) { /* OPTIMIZATIONS GALORE!!! */
    do_something(gp->a, gp->b);
    rcu_read_unlock();
    return true;
  }
  rcu_read_unlock();
  return false;
}
```

If this function ran concurrently with a series of updates that replaced the current structure with a new one, the fetches of `gp->a` and `gp->b` might well come from two different structures, which could cause serious confusion. To prevent this (and much else besides), do_something_gp() uses rcu_dereference() to fetch from `gp`:

```
bool do_something_gp(void)
{
  rcu_read_lock();
  p = rcu_dereference(gp);
  if (p) {
    do_something(p->a, p->b);
    rcu_read_unlock();
    return true;
  }
  rcu_read_unlock();
  return false;
}
```

The rcu_dereference() uses volatile casts and (for DEC Alpha) memory barriers in the Linux kernel. Should a high-quality implementation of C11 `memory_order_consume` [PDF] ever appear, then rcu_dereference() could be implemented as a `memory_order_consume` load. Regardless of the exact implementation, a pointer fetched by rcu_dereference() may not be used outside of the outermost RCU read-side critical section containing that rcu_dereference(), unless protection of the corresponding data element has been passed from RCU to some other synchronization mechanism, most commonly locking or reference counting (see ../../rcuref.rst).

In short, updaters use rcu_assign_pointer() and readers use rcu_dereference(), and these two RCU API elements work together to ensure that readers have a consistent view of newly added data elements.

Of course, it is also necessary to remove elements from RCU-protected data structures, for example, using the following process:

1. Remove the data element from the enclosing structure.
2. Wait for all pre-existing RCU read-side critical sections to complete (because only pre-existing readers can possibly have a reference to the newly removed data element).
3. At this point, only the updater has a reference to the newly removed data element, so it can safely reclaim the data element, for example, by passing it to kfree().

This process is implemented by remove_gp_synchronous():

```
 1 bool remove_gp_synchronous(void)
 2 {
 3   struct foo *p;
 4
 5   spin_lock(&gp_lock);
 6   p = rcu_access_pointer(gp);
 7   if (!p) {
 8     spin_unlock(&gp_lock);
 9     return false;
10   }
11   rcu_assign_pointer(gp, NULL);
12   spin_unlock(&gp_lock);
13   synchronize_rcu();
14   kfree(p);
15   return true;
16 }
```

This function is straightforward, with line 13 waiting for a grace period before line 14 frees the old data element. This waiting ensures that readers will reach line 7 of do_something_gp() before the data element referenced by `p` is freed. The rcu_access_pointer() on line 6 is similar to rcu_dereference(), except that:

1. The value returned by rcu_access_pointer() cannot be dereferenced. If you want to access the value pointed to as well as the pointer itself, use rcu_dereference() instead of rcu_access_pointer().
2. The call to rcu_access_pointer() need not be protected. In contrast, rcu_dereference() must either be within an RCU read-side critical section or in a code segment where the pointer cannot change, for example, in code protected by the corresponding update-side lock.

Quick Quiz:
Without the rcu_dereference() or the rcu_access_pointer(), what destructive optimizations might the compiler make use of?
Answer:
Let's start with what happens to do_something_gp() if it fails to use rcu_dereference(). It could reuse a value formerly fetched from this same pointer. It could also fetch the pointer from `gp` in a byte-at-a-time manner, resulting in *load tearing*, in turn resulting a bytewise mash-up of two distinct pointer values. It might even use value-speculation optimizations, where it makes a wrong guess, but by the time it gets around to checking the value, an update has changed the pointer to match the wrong guess. Too bad about any dereferences that returned pre-initialization garbage in the meantime! For remove_gp_synchronous(), as long as all modifications to `gp` are carried out while holding `gp_lock`, the above optimizations are harmless. However, `sparse` will complain if you define `gp` with `__rcu` and then access it without using either rcu_access_pointer() or rcu_dereference().

In short, RCU's publish-subscribe guarantee is provided by the combination of rcu_assign_pointer() and rcu_dereference(). This guarantee allows data elements to be safely added to RCU-protected linked data structures without disrupting RCU readers. This guarantee can be used in combination with the grace-period guarantee to also allow data elements to be removed from RCU-protected linked data structures, again without disrupting RCU readers.

This guarantee was only partially premeditated. DYNIX/ptx used an explicit memory barrier for publication, but had nothing resembling rcu_dereference() for subscription, nor did it have anything resembling the dependency-ordering barrier that was later subsumed into rcu_dereference() and later still into READ_ONCE(). The need for these operations made itself known quite suddenly at a late-1990s meeting with the DEC Alpha architects, back in the days when DEC was still a free-standing company. It took the Alpha architects a good hour to convince me that any sort of barrier would ever be needed, and it then took me a good *two* hours to convince them that their documentation did not make this point clear. More recent work with the C and C++ standards committees have provided much education on tricks and traps from the compiler. In short, compilers were much less tricky in the early 1990s, but in 2015, don't even think about omitting rcu_dereference()!

17.2.3 Memory-Barrier Guarantees

The previous section's simple linked-data-structure scenario clearly demonstrates the need for RCU's stringent memory-ordering guarantees on systems with more than one CPU:

1. Each CPU that has an RCU read-side critical section that begins before synchronize_rcu() starts is guaranteed to execute a full memory barrier between the time that the RCU read-side critical section ends and the time that synchronize_rcu() returns. Without this guarantee, a pre-existing RCU read-side critical section might hold a reference to the newly removed `struct foo` after the kfree() on line 14 of remove_gp_synchronous().

2. Each CPU that has an RCU read-side critical section that ends after synchronize_rcu() returns is guaranteed to execute a full memory barrier between the time that synchronize_rcu() begins and the time that the RCU read-side critical section begins. Without this guarantee, a later RCU read-side critical section running after the kfree() on line 14 of remove_gp_synchronous() might later run do_something_gp() and find the newly deleted `struct foo`.

3. If the task invoking synchronize_rcu() remains on a given CPU, then that CPU is guaranteed to execute a full memory barrier sometime during the execution of synchronize_rcu(). This guarantee ensures that the kfree() on line 14 of remove_gp_synchronous() really does execute after the removal on line 11.

4. If the task invoking synchronize_rcu() migrates among a group of CPUs during that invocation, then each of the CPUs in that group is guaranteed to execute a full memory barrier sometime during the execution of synchronize_rcu(). This guarantee also ensures that the kfree() on line 14 of remove_gp_synchronous() really does execute after the removal on line 11, but also in the case where the thread executing the synchronize_rcu() migrates in the meantime.

Quick Quiz:

Given that multiple CPUs can start RCU read-side critical sections at any time without any ordering whatsoever, how can RCU possibly tell whether or not a given RCU read-side critical section starts before a given instance of synchronize_rcu()?

Answer:

If RCU cannot tell whether or not a given RCU read-side critical section starts before a given instance of synchronize_rcu(), then it must assume that the RCU read-side critical section started first. In other words, a given instance of synchronize_rcu() can avoid waiting on a given RCU read-side critical section only if it can prove that synchronize_rcu() started first. A related question is "When rcu_read_lock() doesn't generate any code, why does it matter how it relates to a grace period?" The answer is that it is not the relationship of rcu_read_lock() itself that is important, but rather the relationship of the code within the enclosed RCU read-side critical section to the code preceding and following the grace period. If we take this viewpoint, then a given RCU read-side critical section begins before a given grace period when some access preceding the grace period observes the effect of some access within the critical section, in which case none of the accesses within the critical section may observe the effects of any access following the grace period.

As of late 2016, mathematical models of RCU take this viewpoint, for example, see slides 62 and 63 of the 2016 LinuxCon EU presentation.

Quick Quiz:
The first and second guarantees require unbelievably strict ordering! Are all these memory barriers *really* required?

Answer:
Yes, they really are required. To see why the first guarantee is required, consider the following sequence of events:

1. CPU 1: rcu_read_lock()
2. CPU 1: `q = rcu_dereference(gp); /* Very likely to return p. */`
3. CPU 0: `list_del_rcu(p);`
4. CPU 0: synchronize_rcu() starts.
5. CPU 1: `do_something_with(q->a); /* No smp_mb(), so might happen after kfree(). */`
6. CPU 1: rcu_read_unlock()
7. CPU 0: synchronize_rcu() returns.
8. CPU 0: `kfree(p);`

Therefore, there absolutely must be a full memory barrier between the end of the RCU read-side critical section and the end of the grace period.

The sequence of events demonstrating the necessity of the second rule is roughly similar:

1. CPU 0: `list_del_rcu(p);`
2. CPU 0: synchronize_rcu() starts.
3. CPU 1: rcu_read_lock()
4. CPU 1: `q = rcu_dereference(gp); /* Might return p if no memory barrier. */`
5. CPU 0: synchronize_rcu() returns.
6. CPU 0: `kfree(p);`
7. CPU 1: `do_something_with(q->a); /* Boom!!! */`
8. CPU 1: rcu_read_unlock()

And similarly, without a memory barrier between the beginning of the grace period and the beginning of the RCU read-side critical section, CPU 1 might end up accessing the freelist.

The "as if" rule of course applies, so that any implementation that acts as if the appropriate memory barriers were in place is a correct implementation. That said, it is much easier to fool yourself into believing that you have adhered to the as-if rule than it is to actually adhere to it!

Quick Quiz:
You claim that rcu_read_lock() and rcu_read_unlock() generate absolutely no code in some kernel builds. This means that the compiler might arbitrarily rearrange consecutive RCU read-side critical sections. Given such rearrangement, if a given RCU read-side critical section is done, how can you be sure that all prior RCU read-side critical sections are done? Won't the compiler rearrangements make that impossible to determine?

Answer:
In cases where rcu_read_lock() and rcu_read_unlock() generate absolutely no code, RCU infers quiescent states only at special locations, for example, within the scheduler. Because calls to schedule() had better prevent calling-code accesses to shared variables from being rearranged across the call to schedule(), if RCU detects the end of a given RCU read-side critical section, it will necessarily detect the end of all prior RCU read-side critical sections, no matter how aggressively the compiler scrambles the code. Again, this all assumes that the compiler cannot scramble code across calls to the scheduler, out of interrupt handlers, into the idle loop, into user-mode code, and so on. But if your kernel build allows that sort of scrambling, you have broken far more than just RCU!

Note that these memory-barrier requirements do not replace the fundamental RCU requirement that a grace period wait for all pre-existing readers. On the contrary, the memory barriers called out in this section must operate in such a way as to *enforce* this fundamental requirement. Of course, different implementations enforce this requirement in different ways, but enforce it they must.

17.2.4 RCU Primitives Guaranteed to Execute Unconditionally

The common-case RCU primitives are unconditional. They are invoked, they do their job, and they return, with no possibility of error, and no need to retry. This is a key RCU design philosophy.

However, this philosophy is pragmatic rather than pigheaded. If someone comes up with a good justification for a particular conditional RCU primitive, it might well be implemented and added. After all, this guarantee was reverse-engineered, not premeditated. The unconditional nature of the RCU primitives was initially an accident of implementation, and later experience with synchronization primitives with conditional primitives caused me to elevate this accident to a guarantee. Therefore, the justification for adding a conditional primitive to RCU would need to be based on detailed and compelling use cases.

17.2.5 Guaranteed Read-to-Write Upgrade

As far as RCU is concerned, it is always possible to carry out an update within an RCU read-side critical section. For example, that RCU read-side critical section might search for a given data element, and then might acquire the update-side spinlock in order to update that element, all while remaining in that RCU read-side critical section. Of course, it is necessary to exit the RCU read-side critical section before invoking synchronize_rcu(), however, this inconvenience can be avoided through use of the call_rcu() and kfree_rcu() API members described later in this document.

Quick Quiz:
But how does the upgrade-to-write operation exclude other readers?
Answer:
It doesn't, just like normal RCU updates, which also do not exclude RCU readers.

This guarantee allows lookup code to be shared between read-side and update-side code, and was premeditated, appearing in the earliest DYNIX/ptx RCU documentation.

17.3 Fundamental Non-Requirements

RCU provides extremely lightweight readers, and its read-side guarantees, though quite useful, are correspondingly lightweight. It is therefore all too easy to assume that RCU is guaranteeing more than it really is. Of course, the list of things that RCU does not guarantee is infinitely long, however, the following sections list a few non-guarantees that have caused confusion. Except where otherwise noted, these non-guarantees were premeditated.

1. *Readers Impose Minimal Ordering*
2. *Readers Do Not Exclude Updaters*

3. *Updaters Only Wait For Old Readers*
4. *Grace Periods Don’t Partition Read-Side Critical Sections*
5. *Read-Side Critical Sections Don’t Partition Grace Periods*

17.3.1 Readers Impose Minimal Ordering

Reader-side markers such as rcu_read_lock() and rcu_read_unlock() provide absolutely no ordering guarantees except through their interaction with the grace-period APIs such as synchronize_rcu(). To see this, consider the following pair of threads:

```
void thread0(void)
{
  rcu_read_lock();
  WRITE_ONCE(x, 1);
  rcu_read_unlock();
  rcu_read_lock();
  WRITE_ONCE(y, 1);
  rcu_read_unlock();
}

void thread1(void)
{
  rcu_read_lock();
  r1 = READ_ONCE(y);
  rcu_read_unlock();
  rcu_read_lock();
  r2 = READ_ONCE(x);
  rcu_read_unlock();
}
```

After thread0() and thread1() execute concurrently, it is quite possible to have

```
(r1 == 1 && r2 == 0)
```

(that is, y appears to have been assigned before x), which would not be possible if rcu_read_lock() and rcu_read_unlock() had much in the way of ordering properties. But they do not, so the CPU is within its rights to do significant reordering. This is by design: Any significant ordering constraints would slow down these fast-path APIs.

Quick Quiz:
Can’t the compiler also reorder this code?
Answer:
No, the volatile casts in READ_ONCE() and WRITE_ONCE() prevent the compiler from reordering in this particular case.

17.3.2 Readers Do Not Exclude Updaters

Neither rcu_read_lock() nor rcu_read_unlock() exclude updates. All they do is to prevent grace periods from ending. The following example illustrates this:

```
void thread0(void)
{
  rcu_read_lock();
  r1 = READ_ONCE(y);
  if (r1) {
    do_something_with_nonzero_x();
    r2 = READ_ONCE(x);
    WARN_ON(!r2); /* BUG!!! */
  }
  rcu_read_unlock();
}

void thread1(void)
{
  spin_lock(&my_lock);
  WRITE_ONCE(x, 1);
  WRITE_ONCE(y, 1);
  spin_unlock(&my_lock);
}
```

If the thread0() function's rcu_read_lock() excluded the thread1() function's update, the WARN_ON() could never fire. But the fact is that rcu_read_lock() does not exclude much of anything aside from subsequent grace periods, of which thread1() has none, so the WARN_ON() can and does fire.

17.3.3 Updaters Only Wait For Old Readers

It might be tempting to assume that after synchronize_rcu() completes, there are no readers executing. This temptation must be avoided because new readers can start immediately after synchronize_rcu() starts, and synchronize_rcu() is under no obligation to wait for these new readers.

Quick Quiz:

Suppose that synchronize_rcu() did wait until *all* readers had completed instead of waiting only on pre-existing readers. For how long would the updater be able to rely on there being no readers?

Answer:

For no time at all. Even if synchronize_rcu() were to wait until all readers had completed, a new reader might start immediately after synchronize_rcu() completed. Therefore, the code following synchronize_rcu() can *never* rely on there being no readers.

17.3.4 Grace Periods Don't Partition Read-Side Critical Sections

It is tempting to assume that if any part of one RCU read-side critical section precedes a given grace period, and if any part of another RCU read-side critical section follows that same grace period, then all of the first RCU read-side critical section must precede all of the second. However, this just isn't the case: A single grace period does not partition the set of RCU read-side critical sections. An example of this situation can be illustrated as follows, where x, y, and z are initially all zero:

```
void thread0(void)
{
  rcu_read_lock();
  WRITE_ONCE(a, 1);
  WRITE_ONCE(b, 1);
  rcu_read_unlock();
}

void thread1(void)
{
  r1 = READ_ONCE(a);
  synchronize_rcu();
  WRITE_ONCE(c, 1);
}

void thread2(void)
{
  rcu_read_lock();
  r2 = READ_ONCE(b);
  r3 = READ_ONCE(c);
  rcu_read_unlock();
}
```

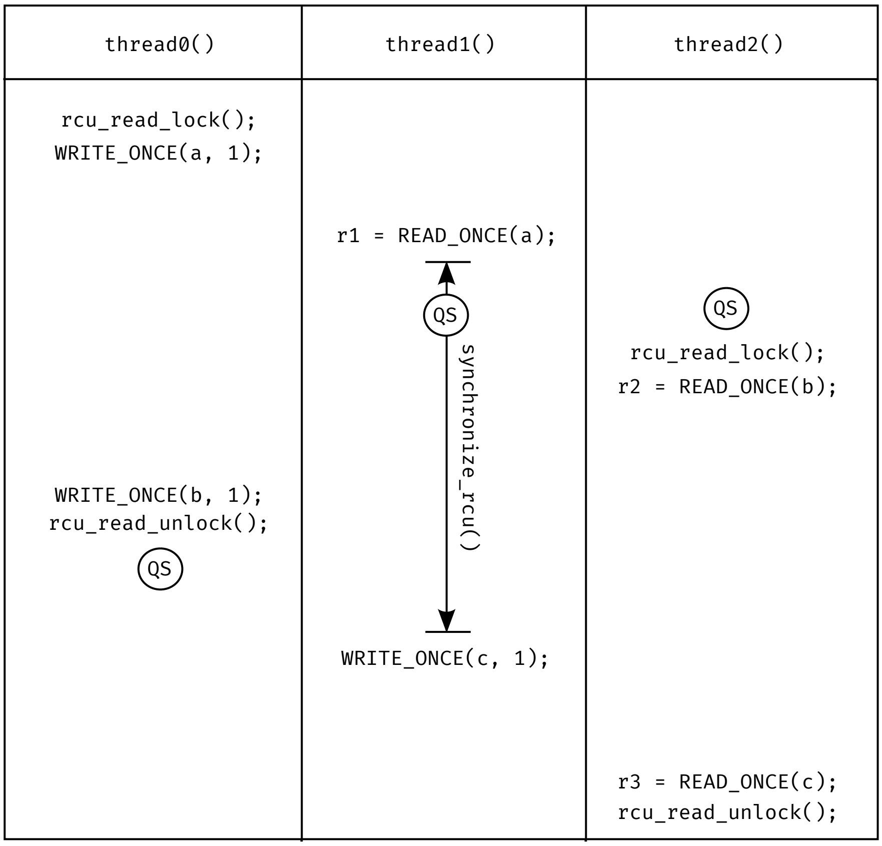
It turns out that the outcome:

```
(r1 == 1 && r2 == 0 && r3 == 1)
```

is entirely possible. The following figure show how this can happen, with each circled `QS` indicating the point at which RCU recorded a *quiescent state* for each thread, that is, a state in which RCU knows that the thread cannot be in the midst of an RCU read-side critical section that started before the current grace period:

If it is necessary to partition RCU read-side critical sections in this manner, it is necessary to use two grace periods, where the first grace period is known to end before the second grace period starts:

```
void thread0(void)
{
  rcu_read_lock();
  WRITE_ONCE(a, 1);
  WRITE_ONCE(b, 1);
  rcu_read_unlock();
}
```




```

void thread1(void)
{
  r1 = READ_ONCE(a);
  synchronize_rcu();
  WRITE_ONCE(c, 1);
}

void thread2(void)
{
  r2 = READ_ONCE(c);
  synchronize_rcu();
  WRITE_ONCE(d, 1);
}

void thread3(void)
{
  rcu_read_lock();
  r3 = READ_ONCE(b);
  r4 = READ_ONCE(d);
  rcu_read_unlock();
}
```

Here, if `(r1 == 1)`, then thread0()'s write to `b` must happen before the end of thread1()'s grace period. If in addition `(r4 == 1)`, then thread3()'s read from `b` must happen after the beginning of thread2()'s grace period. If it is also the case that (r2 == 1), then the end of thread1()'s grace period must precede the beginning of thread2()'s grace period. This mean that the two RCU read-side critical sections cannot overlap, guaranteeing that (r3 == 1). As a result, the outcome:

```
(r1 == 1 && r2 == 1 && r3 == 0 && r4 == 1)
```

cannot happen.

This non-requirement was also non-premeditated, but became apparent when studying RCU's interaction with memory ordering.

17.3.5 Read-Side Critical Sections Don't Partition Grace Periods

It is also tempting to assume that if an RCU read-side critical section happens between a pair of grace periods, then those grace periods cannot overlap. However, this temptation leads nowhere good, as can be illustrated by the following, with all variables initially zero:

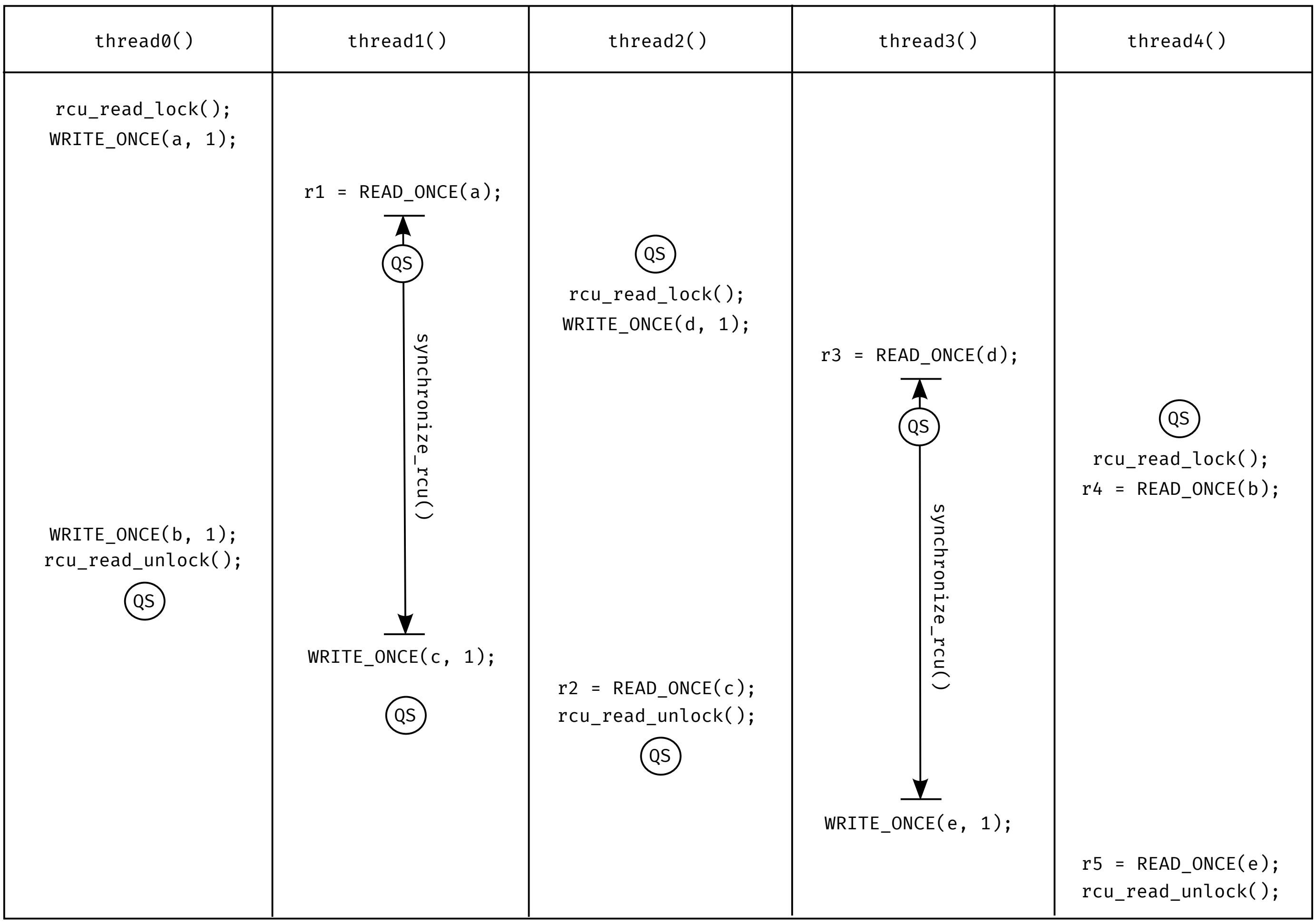
```
void thread0(void)
{
  rcu_read_lock();
  WRITE_ONCE(a, 1);
  WRITE_ONCE(b, 1);
  rcu_read_unlock();
}

void thread1(void)
{
  r1 = READ_ONCE(a);
  synchronize_rcu();
  WRITE_ONCE(c, 1);
}

void thread2(void)
{
  rcu_read_lock();
  WRITE_ONCE(d, 1);
  r2 = READ_ONCE(c);
  rcu_read_unlock();
}

void thread3(void)
{
  r3 = READ_ONCE(d);
  synchronize_rcu();
  WRITE_ONCE(e, 1);
}

void thread4(void)
{
  rcu_read_lock();
```

```
  r4 = READ_ONCE(b);
  r5 = READ_ONCE(e);
  rcu_read_unlock();
}
```

In this case, the outcome:

```
(r1 == 1 && r2 == 1 && r3 == 1 && r4 == 0 && r5 == 1)
```

is entirely possible, as illustrated below:



Again, an RCU read-side critical section can overlap almost all of a given grace period, just so long as it does not overlap the entire grace period. As a result, an RCU read-side critical section cannot partition a pair of RCU grace periods.

Quick Quiz:
How long a sequence of grace periods, each separated by an RCU read-side critical section, would be required to partition the RCU read-side critical sections at the beginning and end of the chain?

Answer:
In theory, an infinite number. In practice, an unknown number that is sensitive to both implementation details and timing considerations. Therefore, even in practice, RCU users must abide by the theoretical rather than the practical answer.

17.4 Parallelism Facts of Life

These parallelism facts of life are by no means specific to RCU, but the RCU implementation must abide by them. They therefore bear repeating:

1. Any CPU or task may be delayed at any time, and any attempts to avoid these delays by disabling preemption, interrupts, or whatever are completely futile. This is most obvious in preemptible user-level environments and in virtualized environments (where a given guest OS's VCPUs can be preempted at any time by the underlying hypervisor), but can also happen in bare-metal environments due to ECC errors, NMIs, and other hardware events. Although a delay of more than about 20 seconds can result in splats, the RCU implementation is obligated to use algorithms that can tolerate extremely long delays, but where "extremely long" is not long enough to allow wrap-around when incrementing a 64-bit counter.
2. Both the compiler and the CPU can reorder memory accesses. Where it matters, RCU must use compiler directives and memory-barrier instructions to preserve ordering.
3. Conflicting writes to memory locations in any given cache line will result in expensive cache misses. Greater numbers of concurrent writes and more-frequent concurrent writes will result in more dramatic slowdowns. RCU is therefore obligated to use algorithms that have sufficient locality to avoid significant performance and scalability problems.
4. As a rough rule of thumb, only one CPU's worth of processing may be carried out under the protection of any given exclusive lock. RCU must therefore use scalable locking designs.
5. Counters are finite, especially on 32-bit systems. RCU's use of counters must therefore tolerate counter wrap, or be designed such that counter wrap would take way more time than a single system is likely to run. An uptime of ten years is quite possible, a runtime of a century much less so. As an example of the latter, RCU's dyntick-idle nesting counter allows 54 bits for interrupt nesting level (this counter is 64 bits even on a 32-bit system). Overflowing this counter requires 2^{54} half-interrupts on a given CPU without that CPU ever going idle. If a half-interrupt happened every microsecond, it would take 570 years of runtime to overflow this counter, which is currently believed to be an acceptably long time.
6. Linux systems can have thousands of CPUs running a single Linux kernel in a single shared-memory environment. RCU must therefore pay close attention to high-end scalability.

This last parallelism fact of life means that RCU must pay special attention to the preceding facts of life. The idea that Linux might scale to systems with thousands of CPUs would have been met with some skepticism in the 1990s, but these requirements would have otherwise have been unsurprising, even in the early 1990s.

17.5 Quality-of-Implementation Requirements

These sections list quality-of-implementation requirements. Although an RCU implementation that ignores these requirements could still be used, it would likely be subject to limitations that would make it inappropriate for industrial-strength production use. Classes of quality-of-implementation requirements are as follows:

1. *Specialization*
2. *Performance and Scalability*

3. *Forward Progress*
4. *Composability*
5. *Corner Cases*

These classes is covered in the following sections.

17.5.1 Specialization

RCU is and always has been intended primarily for read-mostly situations, which means that RCU's read-side primitives are optimized, often at the expense of its update-side primitives. Experience thus far is captured by the following list of situations:

1. Read-mostly data, where stale and inconsistent data is not a problem: RCU works great!
2. Read-mostly data, where data must be consistent: RCU works well.
3. Read-write data, where data must be consistent: RCU *might* work OK. Or not.
4. Write-mostly data, where data must be consistent: RCU is very unlikely to be the right tool for the job, with the following exceptions, where RCU can provide:
   a. Existence guarantees for update-friendly mechanisms.
   b. Wait-free read-side primitives for real-time use.

This focus on read-mostly situations means that RCU must interoperate with other synchronization primitives. For example, the add_gp() and remove_gp_synchronous() examples discussed earlier use RCU to protect readers and locking to coordinate updaters. However, the need extends much farther, requiring that a variety of synchronization primitives be legal within RCU read-side critical sections, including spinlocks, sequence locks, atomic operations, reference counters, and memory barriers.

Quick Quiz:

What about sleeping locks?

Answer:

These are forbidden within Linux-kernel RCU read-side critical sections because it is not legal to place a quiescent state (in this case, voluntary context switch) within an RCU read-side critical section. However, sleeping locks may be used within userspace RCU read-side critical sections, and also within Linux-kernel sleepable RCU *(SRCU)* read-side critical sections. In addition, the -rt patchset turns spinlocks into a sleeping locks so that the corresponding critical sections can be preempted, which also means that these sleeplockified spinlocks (but not other sleeping locks!) may be acquire within -rt-Linux-kernel RCU read-side critical sections. Note that it *is* legal for a normal RCU read-side critical section to conditionally acquire a sleeping locks (as in mutex_trylock()), but only as long as it does not loop indefinitely attempting to conditionally acquire that sleeping locks. The key point is that things like mutex_trylock() either return with the mutex held, or return an error indication if the mutex was not immediately available. Either way, mutex_trylock() returns immediately without sleeping.

It often comes as a surprise that many algorithms do not require a consistent view of data, but many can function in that mode, with network routing being the poster child. Internet routing algorithms take significant time to propagate updates, so that by the time an update arrives at a given system, that system has been sending network traffic the wrong way for a considerable length of time. Having a few threads continue to send traffic the wrong way for a few more milliseconds is clearly not a problem: In the worst case, TCP retransmissions will eventually get the data where it needs to go. In general, when tracking the state of the universe outside of the computer, some level of inconsistency must be tolerated due to speed-of-light delays if nothing else.

Furthermore, uncertainty about external state is inherent in many cases. For example, a pair of veterinarians might use heartbeat to determine whether or not a given cat was alive. But how long should they wait after the last heartbeat to decide that the cat is in fact dead? Waiting less than 400 milliseconds makes no sense because this would mean that a relaxed cat would be considered to cycle between death and life more than 100 times per minute. Moreover, just as with human beings, a cat's heart might stop for some period of time, so the exact wait period is a judgment call. One of our pair of veterinarians might wait 30 seconds before pronouncing the cat dead, while the other might insist on waiting a full minute. The two veterinarians would then disagree on the state of the cat during the final 30 seconds of the minute following the last heartbeat.

Interestingly enough, this same situation applies to hardware. When push comes to shove, how do we tell whether or not some external server has failed? We send messages to it periodically, and declare it failed if we don't receive a response within a given period of time. Policy decisions can usually tolerate short periods of inconsistency. The policy was decided some time ago, and is only now being put into effect, so a few milliseconds of delay is normally inconsequential.

However, there are algorithms that absolutely must see consistent data. For example, the translation between a user-level SystemV semaphore ID to the corresponding in-kernel data structure is protected by RCU, but it is absolutely forbidden to update a semaphore that has just been removed. In the Linux kernel, this need for consistency is accommodated by acquiring spinlocks located in the in-kernel data structure from within the RCU read-side critical section, and this is indicated by the green box in the figure above. Many other techniques may be used, and are in fact used within the Linux kernel.

In short, RCU is not required to maintain consistency, and other mechanisms may be used in concert with RCU when consistency is required. RCU's specialization allows it to do its job extremely well, and its ability to interoperate with other synchronization mechanisms allows the right mix of synchronization tools to be used for a given job.

17.5.2 Performance and Scalability

Energy efficiency is a critical component of performance today, and Linux-kernel RCU implementations must therefore avoid unnecessarily awakening idle CPUs. I cannot claim that this requirement was premeditated. In fact, I learned of it during a telephone conversation in which I was given "frank and open" feedback on the importance of energy efficiency in battery-powered systems and on specific energy-efficiency shortcomings of the Linux-kernel RCU implementation. In my experience, the battery-powered embedded community will consider any unnecessary wakeups to be extremely unfriendly acts. So much so that mere Linux-kernel-mailing-list posts are insufficient to vent their ire.

Memory consumption is not particularly important for in most situations, and has become decreasingly so as memory sizes have expanded and memory costs have plummeted. However, as I learned from Matt Mackall's bloatwatch efforts, memory footprint is critically important on single-CPU systems with non-preemptible (`CONFIG_PREEMPTION=n`) kernels, and thus tiny RCU was born. Josh Triplett has since taken over the small-memory banner with his Linux kernel tinification project, which resulted in *SRCU* becoming optional for those kernels not needing it.

The remaining performance requirements are, for the most part, unsurprising. For example, in keeping with RCU's read-side specialization, rcu_dereference() should have negligible overhead (for example, suppression of a few minor compiler optimizations). Similarly, in non-preemptible environments, rcu_read_lock() and rcu_read_unlock() should have exactly zero overhead.

In preemptible environments, in the case where the RCU read-side critical section was not preempted (as will be the case for the highest-priority real-time process), rcu_read_lock() and rcu_read_unlock() should have minimal overhead. In particular, they should not contain atomic read-modify-write operations, memory-barrier instructions, preemption disabling, interrupt disabling, or backwards branches. However, in the case where the RCU read-side critical section was preempted, rcu_read_unlock() may acquire spinlocks and disable interrupts. This is why it is better to nest an RCU read-side critical section within a preempt-disable region than vice versa, at least in cases where that critical section is short enough to avoid unduly degrading real-time latencies.

The synchronize_rcu() grace-period-wait primitive is optimized for throughput. It may therefore incur several milliseconds of latency in addition to the duration of the longest RCU read-side critical section. On the other hand, multiple concurrent invocations of synchronize_rcu() are required to use batching optimizations so that they can be satisfied by a single underlying grace-period-wait operation. For example, in the Linux kernel, it is not unusual for a single grace-period-wait operation to serve more than 1,000 separate invocations of synchronize_rcu(), thus amortizing the per-invocation overhead down to nearly zero. However, the grace-period optimization is also required to avoid measurable degradation of real-time scheduling and interrupt latencies.

In some cases, the multi-millisecond synchronize_rcu() latencies are unacceptable. In these cases, synchronize_rcu_expedited() may be used instead, reducing the grace-period latency down to a few tens of microseconds on small systems, at least in cases where the RCU read-side critical sections are short. There are currently no special latency requirements for synchronize_rcu_expedited() on large systems, but, consistent with the empirical nature of the RCU specification, that is subject to change. However, there most definitely are scalability requirements: A storm of synchronize_rcu_expedited() invocations on 4096 CPUs should at least make reasonable forward progress. In return for its shorter latencies, synchronize_rcu_expedited() is permitted to impose modest degradation of real-time latency on non-idle online CPUs. Here, "modest" means roughly the same latency degradation as a scheduling-clock interrupt.

There are a number of situations where even synchronize_rcu_expedited()'s reduced grace-period latency is unacceptable. In these situations, the asynchronous call_rcu() can be used in place of synchronize_rcu() as follows:

```
struct foo {
  int a;
  int b;
  struct rcu_head rh;
};

static void remove_gp_cb(struct rcu_head *rhp)
{
  struct foo *p = container_of(rhp, struct foo, rh);

  kfree(p);
}

```

```
14 bool remove_gp_asynchronous(void)
15 {
16   struct foo *p;
17
18   spin_lock(&gp_lock);
19   p = rcu_access_pointer(gp);
20   if (!p) {
21     spin_unlock(&gp_lock);
22     return false;
23   }
24   rcu_assign_pointer(gp, NULL);
25   call_rcu(&p->rh, remove_gp_cb);
26   spin_unlock(&gp_lock);
27   return true;
28 }
```

A definition of `struct foo` is finally needed, and appears on lines 1-5. The function remove_gp_cb() is passed to call_rcu() on line 25, and will be invoked after the end of a subsequent grace period. This gets the same effect as remove_gp_synchronous(), but without forcing the updater to wait for a grace period to elapse. The call_rcu() function may be used in a number of situations where neither synchronize_rcu() nor synchronize_rcu_expedited() would be legal, including within preempt-disable code, local_bh_disable() code, interrupt-disable code, and interrupt handlers. However, even call_rcu() is illegal within NMI handlers and from idle and offline CPUs. The callback function (remove_gp_cb() in this case) will be executed within softirq (software interrupt) environment within the Linux kernel, either within a real softirq handler or under the protection of local_bh_disable(). In both the Linux kernel and in userspace, it is bad practice to write an RCU callback function that takes too long. Long-running operations should be relegated to separate threads or (in the Linux kernel) workqueues.

Quick Quiz:

Why does line 19 use rcu_access_pointer()? After all, call_rcu() on line 25 stores into the structure, which would interact badly with concurrent insertions. Doesn't this mean that rcu_dereference() is required?

Answer:

Presumably the `->gp_lock` acquired on line 18 excludes any changes, including any insertions that rcu_dereference() would protect against. Therefore, any insertions will be delayed until after `->gp_lock` is released on line 25, which in turn means that rcu_access_pointer() suffices.

However, all that remove_gp_cb() is doing is invoking kfree() on the data element. This is a common idiom, and is supported by kfree_rcu(), which allows "fire and forget" operation as shown below:

```
struct foo {
  int a;
  int b;
  struct rcu_head rh;
};

bool remove_gp_faf(void)
{
```

```
  struct foo *p;

  spin_lock(&gp_lock);
  p = rcu_dereference(gp);
  if (!p) {
    spin_unlock(&gp_lock);
    return false;
  }
  rcu_assign_pointer(gp, NULL);
  kfree_rcu(p, rh);
  spin_unlock(&gp_lock);
  return true;
}
```

Note that remove_gp_faf() simply invokes kfree_rcu() and proceeds, without any need to pay any further attention to the subsequent grace period and kfree(). It is permissible to invoke kfree_rcu() from the same environments as for call_rcu(). Interestingly enough, DYNIX/ptx had the equivalents of call_rcu() and kfree_rcu(), but not synchronize_rcu(). This was due to the fact that RCU was not heavily used within DYNIX/ptx, so the very few places that needed something like synchronize_rcu() simply open-coded it.

Quick Quiz:

Earlier it was claimed that call_rcu() and kfree_rcu() allowed updaters to avoid being blocked by readers. But how can that be correct, given that the invocation of the callback and the freeing of the memory (respectively) must still wait for a grace period to elapse?

Answer:

We could define things this way, but keep in mind that this sort of definition would say that updates in garbage-collected languages cannot complete until the next time the garbage collector runs, which does not seem at all reasonable. The key point is that in most cases, an updater using either call_rcu() or kfree_rcu() can proceed to the next update as soon as it has invoked call_rcu() or kfree_rcu(), without having to wait for a subsequent grace period.

But what if the updater must wait for the completion of code to be executed after the end of the grace period, but has other tasks that can be carried out in the meantime? The polling-style get_state_synchronize_rcu() and cond_synchronize_rcu() functions may be used for this purpose, as shown below:

```
bool remove_gp_poll(void)
{
  struct foo *p;
  unsigned long s;

  spin_lock(&gp_lock);
  p = rcu_access_pointer(gp);
  if (!p) {
    spin_unlock(&gp_lock);
    return false;
  }
  rcu_assign_pointer(gp, NULL);
  spin_unlock(&gp_lock);
```

```
14   s = get_state_synchronize_rcu();
15   do_something_while_waiting();
16   cond_synchronize_rcu(s);
17   kfree(p);
18   return true;
19 }
```

On line 14, get_state_synchronize_rcu() obtains a "cookie" from RCU, then line 15 carries out other tasks, and finally, line 16 returns immediately if a grace period has elapsed in the meantime, but otherwise waits as required. The need for `get_state_synchronize_rcu` and cond_synchronize_rcu() has appeared quite recently, so it is too early to tell whether they will stand the test of time.

RCU thus provides a range of tools to allow updaters to strike the required tradeoff between latency, flexibility and CPU overhead.

17.5.3 Forward Progress

In theory, delaying grace-period completion and callback invocation is harmless. In practice, not only are memory sizes finite but also callbacks sometimes do wakeups, and sufficiently deferred wakeups can be difficult to distinguish from system hangs. Therefore, RCU must provide a number of mechanisms to promote forward progress.

These mechanisms are not foolproof, nor can they be. For one simple example, an infinite loop in an RCU read-side critical section must by definition prevent later grace periods from ever completing. For a more involved example, consider a 64-CPU system built with `CONFIG_RCU_NOCB_CPU=y` and booted with `rcu_nocbs=1-63`, where CPUs 1 through 63 spin in tight loops that invoke call_rcu(). Even if these tight loops also contain calls to cond_resched() (thus allowing grace periods to complete), CPU 0 simply will not be able to invoke callbacks as fast as the other 63 CPUs can register them, at least not until the system runs out of memory. In both of these examples, the Spiderman principle applies: With great power comes great responsibility. However, short of this level of abuse, RCU is required to ensure timely completion of grace periods and timely invocation of callbacks.

RCU takes the following steps to encourage timely completion of grace periods:

1. If a grace period fails to complete within 100 milliseconds, RCU causes future invocations of cond_resched() on the holdout CPUs to provide an RCU quiescent state. RCU also causes those CPUs' need_resched() invocations to return `true`, but only after the corresponding CPU's next scheduling-clock.
2. CPUs mentioned in the `nohz_full` kernel boot parameter can run indefinitely in the kernel without scheduling-clock interrupts, which defeats the above need_resched() strategem. RCU will therefore invoke resched_cpu() on any `nohz_full` CPUs still holding out after 109 milliseconds.
3. In kernels built with `CONFIG_RCU_BOOST=y`, if a given task that has been preempted within an RCU read-side critical section is holding out for more than 500 milliseconds, RCU will resort to priority boosting.
4. If a CPU is still holding out 10 seconds into the grace period, RCU will invoke resched_cpu() on it regardless of its `nohz_full` state.

The above values are defaults for systems running with HZ=1000. They will vary as the value of HZ varies, and can also be changed using the relevant Kconfig options and kernel boot parameters. RCU currently does not do much sanity checking of these parameters, so please use caution when changing them. Note that these forward-progress measures are provided only for RCU, not for *SRCU* or *Tasks RCU*.

RCU takes the following steps in call_rcu() to encourage timely invocation of callbacks when any given non-rcu_nocbs CPU has 10,000 callbacks, or has 10,000 more callbacks than it had the last time encouragement was provided:

1. Starts a grace period, if one is not already in progress.
2. Forces immediate checking for quiescent states, rather than waiting for three milliseconds to have elapsed since the beginning of the grace period.
3. Immediately tags the CPU's callbacks with their grace period completion numbers, rather than waiting for the RCU_SOFTIRQ handler to get around to it.
4. Lifts callback-execution batch limits, which speeds up callback invocation at the expense of degrading realtime response.

Again, these are default values when running at HZ=1000, and can be overridden. Again, these forward-progress measures are provided only for RCU, not for *SRCU* or *Tasks RCU*. Even for RCU, callback-invocation forward progress for rcu_nocbs CPUs is much less well-developed, in part because workloads benefiting from rcu_nocbs CPUs tend to invoke call_rcu() relatively infrequently. If workloads emerge that need both rcu_nocbs CPUs and high call_rcu() invocation rates, then additional forward-progress work will be required.

17.5.4 Composability

Composability has received much attention in recent years, perhaps in part due to the collision of multicore hardware with object-oriented techniques designed in single-threaded environments for single-threaded use. And in theory, RCU read-side critical sections may be composed, and in fact may be nested arbitrarily deeply. In practice, as with all real-world implementations of composable constructs, there are limitations.

Implementations of RCU for which rcu_read_lock() and rcu_read_unlock() generate no code, such as Linux-kernel RCU when CONFIG_PREEMPTION=n, can be nested arbitrarily deeply. After all, there is no overhead. Except that if all these instances of rcu_read_lock() and rcu_read_unlock() are visible to the compiler, compilation will eventually fail due to exhausting memory, mass storage, or user patience, whichever comes first. If the nesting is not visible to the compiler, as is the case with mutually recursive functions each in its own translation unit, stack overflow will result. If the nesting takes the form of loops, perhaps in the guise of tail recursion, either the control variable will overflow or (in the Linux kernel) you will get an RCU CPU stall warning. Nevertheless, this class of RCU implementations is one of the most composable constructs in existence.

RCU implementations that explicitly track nesting depth are limited by the nesting-depth counter. For example, the Linux kernel's preemptible RCU limits nesting to INT_MAX. This should suffice for almost all practical purposes. That said, a consecutive pair of RCU read-side critical sections between which there is an operation that waits for a grace period cannot be enclosed in another RCU read-side critical section. This is because it is not legal to wait for a grace period within an RCU read-side critical section: To do so would result either in deadlock or in RCU implicitly splitting the enclosing RCU read-side critical section, neither of which is conducive to a long-lived and prosperous kernel.

It is worth noting that RCU is not alone in limiting composability. For example, many transactional-memory implementations prohibit composing a pair of transactions separated by an irrevocable operation (for example, a network receive operation). For another example, lock-based critical sections can be composed surprisingly freely, but only if deadlock is avoided.

In short, although RCU read-side critical sections are highly composable, care is required in some situations, just as is the case for any other composable synchronization mechanism.

17.5.5 Corner Cases

A given RCU workload might have an endless and intense stream of RCU read-side critical sections, perhaps even so intense that there was never a point in time during which there was not at least one RCU read-side critical section in flight. RCU cannot allow this situation to block grace periods: As long as all the RCU read-side critical sections are finite, grace periods must also be finite.

That said, preemptible RCU implementations could potentially result in RCU read-side critical sections being preempted for long durations, which has the effect of creating a long-duration RCU read-side critical section. This situation can arise only in heavily loaded systems, but systems using real-time priorities are of course more vulnerable. Therefore, RCU priority boosting is provided to help deal with this case. That said, the exact requirements on RCU priority boosting will likely evolve as more experience accumulates.

Other workloads might have very high update rates. Although one can argue that such workloads should instead use something other than RCU, the fact remains that RCU must handle such workloads gracefully. This requirement is another factor driving batching of grace periods, but it is also the driving force behind the checks for large numbers of queued RCU callbacks in the call_rcu() code path. Finally, high update rates should not delay RCU read-side critical sections, although some small read-side delays can occur when using synchronize_rcu_expedited(), courtesy of this function's use of smp_call_function_single().

Although all three of these corner cases were understood in the early 1990s, a simple user-level test consisting of `close(open(path))` in a tight loop in the early 2000s suddenly provided a much deeper appreciation of the high-update-rate corner case. This test also motivated addition of some RCU code to react to high update rates, for example, if a given CPU finds itself with more than 10,000 RCU callbacks queued, it will cause RCU to take evasive action by more aggressively starting grace periods and more aggressively forcing completion of grace-period processing. This evasive action causes the grace period to complete more quickly, but at the cost of restricting RCU's batching optimizations, thus increasing the CPU overhead incurred by that grace period.

17.6 Software-Engineering Requirements

Between Murphy's Law and "To err is human", it is necessary to guard against mishaps and misuse:

1. It is all too easy to forget to use rcu_read_lock() everywhere that it is needed, so kernels built with `CONFIG_PROVE_RCU=y` will splat if rcu_dereference() is used outside of an RCU read-side critical section. Update-side code can use rcu_dereference_protected(), which takes a lockdep expression to indicate what is providing the protection. If the indicated protection is not provided, a lockdep splat is emitted. Code shared between readers and updaters can use rcu_dereference_check(), which also takes a lockdep expression, and

emits a lockdep splat if neither rcu_read_lock() nor the indicated protection is in place. In addition, rcu_dereference_raw() is used in those (hopefully rare) cases where the required protection cannot be easily described. Finally, rcu_read_lock_held() is provided to allow a function to verify that it has been invoked within an RCU read-side critical section. I was made aware of this set of requirements shortly after Thomas Gleixner audited a number of RCU uses.

2. A given function might wish to check for RCU-related preconditions upon entry, before using any other RCU API. The rcu_lockdep_assert() does this job, asserting the expression in kernels having lockdep enabled and doing nothing otherwise.

3. It is also easy to forget to use rcu_assign_pointer() and rcu_dereference(), perhaps (incorrectly) substituting a simple assignment. To catch this sort of error, a given RCU-protected pointer may be tagged with `__rcu`, after which sparse will complain about simple-assignment accesses to that pointer. Arnd Bergmann made me aware of this requirement, and also supplied the needed patch series.

4. Kernels built with `CONFIG_DEBUG_OBJECTS_RCU_HEAD=y` will splat if a data element is passed to call_rcu() twice in a row, without a grace period in between. (This error is similar to a double free.) The corresponding `rcu_head` structures that are dynamically allocated are automatically tracked, but `rcu_head` structures allocated on the stack must be initialized with init_rcu_head_on_stack() and cleaned up with destroy_rcu_head_on_stack(). Similarly, statically allocated non-stack `rcu_head` structures must be initialized with init_rcu_head() and cleaned up with destroy_rcu_head(). Mathieu Desnoyers made me aware of this requirement, and also supplied the needed patch.

5. An infinite loop in an RCU read-side critical section will eventually trigger an RCU CPU stall warning splat, with the duration of "eventually" being controlled by the `RCU_CPU_STALL_TIMEOUT Kconfig` option, or, alternatively, by the `rcupdate.rcu_cpu_stall_timeout` boot/sysfs parameter. However, RCU is not obligated to produce this splat unless there is a grace period waiting on that particular RCU read-side critical section.

   Some extreme workloads might intentionally delay RCU grace periods, and systems running those workloads can be booted with `rcupdate.rcu_cpu_stall_suppress` to suppress the splats. This kernel parameter may also be set via `sysfs`. Furthermore, RCU CPU stall warnings are counter-productive during sysrq dumps and during panics. RCU therefore supplies the rcu_sysrq_start() and rcu_sysrq_end() API members to be called before and after long sysrq dumps. RCU also supplies the rcu_panic() notifier that is automatically invoked at the beginning of a panic to suppress further RCU CPU stall warnings.

   This requirement made itself known in the early 1990s, pretty much the first time that it was necessary to debug a CPU stall. That said, the initial implementation in DYNIX/ptx was quite generic in comparison with that of Linux.

6. Although it would be very good to detect pointers leaking out of RCU read-side critical sections, there is currently no good way of doing this. One complication is the need to distinguish between pointers leaking and pointers that have been handed off from RCU to some other synchronization mechanism, for example, reference counting.

7. In kernels built with `CONFIG_RCU_TRACE=y`, RCU-related information is provided via event tracing.

8. Open-coded use of rcu_assign_pointer() and rcu_dereference() to create typical linked data structures can be surprisingly error-prone. Therefore, RCU-protected linked lists and,

more recently, RCU-protected hash tables are available. Many other special-purpose RCU-protected data structures are available in the Linux kernel and the userspace RCU library.

9. Some linked structures are created at compile time, but still require `__rcu` checking. The RCU_POINTER_INITIALIZER() macro serves this purpose.

10. It is not necessary to use rcu_assign_pointer() when creating linked structures that are to be published via a single external pointer. The RCU_INIT_POINTER() macro is provided for this task.

This not a hard-and-fast list: RCU's diagnostic capabilities will continue to be guided by the number and type of usage bugs found in real-world RCU usage.

17.7 Linux Kernel Complications

The Linux kernel provides an interesting environment for all kinds of software, including RCU. Some of the relevant points of interest are as follows:

1. *Configuration*
2. *Firmware Interface*
3. *Early Boot*
4. *Interrupts and NMIs*
5. *Loadable Modules*
6. *Hotplug CPU*
7. *Scheduler and RCU*
8. *Tracing and RCU*
9. *Accesses to User Memory and RCU*
10. *Energy Efficiency*
11. *Scheduling-Clock Interrupts and RCU*
12. *Memory Efficiency*
13. *Performance, Scalability, Response Time, and Reliability*

This list is probably incomplete, but it does give a feel for the most notable Linux-kernel complications. Each of the following sections covers one of the above topics.

17.7.1 Configuration

RCU's goal is automatic configuration, so that almost nobody needs to worry about RCU's `Kconfig` options. And for almost all users, RCU does in fact work well "out of the box."

However, there are specialized use cases that are handled by kernel boot parameters and `Kconfig` options. Unfortunately, the `Kconfig` system will explicitly ask users about new `Kconfig` options, which requires almost all of them be hidden behind a `CONFIG_RCU_EXPERT` `Kconfig` option.

This all should be quite obvious, but the fact remains that Linus Torvalds recently had to remind me of this requirement.

17.7.2 Firmware Interface

In many cases, kernel obtains information about the system from the firmware, and sometimes things are lost in translation. Or the translation is accurate, but the original message is bogus.

For example, some systems' firmware overreports the number of CPUs, sometimes by a large factor. If RCU naively believed the firmware, as it used to do, it would create too many per-CPU kthreads. Although the resulting system will still run correctly, the extra kthreads needlessly consume memory and can cause confusion when they show up in `ps` listings.

RCU must therefore wait for a given CPU to actually come online before it can allow itself to believe that the CPU actually exists. The resulting "ghost CPUs" (which are never going to come online) cause a number of interesting complications.

17.7.3 Early Boot

The Linux kernel's boot sequence is an interesting process, and RCU is used early, even before rcu_init() is invoked. In fact, a number of RCU's primitives can be used as soon as the initial task's `task_struct` is available and the boot CPU's per-CPU variables are set up. The read-side primitives (rcu_read_lock(), rcu_read_unlock(), rcu_dereference(), and rcu_access_pointer()) will operate normally very early on, as will rcu_assign_pointer().

Although call_rcu() may be invoked at any time during boot, callbacks are not guaranteed to be invoked until after all of RCU's kthreads have been spawned, which occurs at early_initcall() time. This delay in callback invocation is due to the fact that RCU does not invoke callbacks until it is fully initialized, and this full initialization cannot occur until after the scheduler has initialized itself to the point where RCU can spawn and run its kthreads. In theory, it would be possible to invoke callbacks earlier, however, this is not a panacea because there would be severe restrictions on what operations those callbacks could invoke.

Perhaps surprisingly, synchronize_rcu() and synchronize_rcu_expedited(), will operate normally during very early boot, the reason being that there is only one CPU and preemption is disabled. This means that the call synchronize_rcu() (or friends) itself is a quiescent state and thus a grace period, so the early-boot implementation can be a no-op.

However, once the scheduler has spawned its first kthread, this early boot trick fails for synchronize_rcu() (as well as for synchronize_rcu_expedited()) in `CONFIG_PREEMPTION=y` kernels. The reason is that an RCU read-side critical section might be preempted, which means that a subsequent synchronize_rcu() really does have to wait for something, as opposed to simply returning immediately. Unfortunately, synchronize_rcu() can't do this until all of its kthreads are spawned, which doesn't happen until some time during early_initcalls() time. But this is no excuse: RCU is nevertheless required to correctly handle synchronous grace periods during this time period. Once all of its kthreads are up and running, RCU starts running normally.

Quick Quiz:

How can RCU possibly handle grace periods before all of its kthreads have been spawned???

Answer:

Very carefully! During the "dead zone" between the time that the scheduler spawns the first task and the time that all of RCU's kthreads have been spawned, all synchronous grace periods are handled by the expedited grace-period mechanism. At runtime, this expedited mechanism relies on workqueues, but during the dead zone the requesting task itself drives the desired expedited grace period. Because dead-zone execution takes place within task context, everything works. Once the dead zone ends, expedited grace periods go back to using workqueues, as is required to avoid problems that would otherwise occur when a user task received a POSIX signal while driving an expedited grace period.

And yes, this does mean that it is unhelpful to send POSIX signals to random tasks between the time that the scheduler spawns its first kthread and the time that RCU's kthreads have all been spawned. If there ever turns out to be a good reason for sending POSIX signals during that time, appropriate adjustments will be made. (If it turns out that POSIX signals are sent during this time for no good reason, other adjustments will be made, appropriate or otherwise.)

I learned of these boot-time requirements as a result of a series of system hangs.

17.7.4 Interrupts and NMIs

The Linux kernel has interrupts, and RCU read-side critical sections are legal within interrupt handlers and within interrupt-disabled regions of code, as are invocations of call_rcu().

Some Linux-kernel architectures can enter an interrupt handler from non-idle process context, and then just never leave it, instead stealthily transitioning back to process context. This trick is sometimes used to invoke system calls from inside the kernel. These "half-interrupts" mean that RCU has to be very careful about how it counts interrupt nesting levels. I learned of this requirement the hard way during a rewrite of RCU's dyntick-idle code.

The Linux kernel has non-maskable interrupts (NMIs), and RCU read-side critical sections are legal within NMI handlers. Thankfully, RCU update-side primitives, including call_rcu(), are prohibited within NMI handlers.

The name notwithstanding, some Linux-kernel architectures can have nested NMIs, which RCU must handle correctly. Andy Lutomirski surprised me with this requirement; he also kindly surprised me with an algorithm that meets this requirement.

Furthermore, NMI handlers can be interrupted by what appear to RCU to be normal interrupts. One way that this can happen is for code that directly invokes ct_irq_enter() and ct_irq_exit() to be called from an NMI handler. This astonishing fact of life prompted the current code structure, which has ct_irq_enter() invoking ct_nmi_enter() and ct_irq_exit() invoking ct_nmi_exit(). And yes, I also learned of this requirement the hard way.

17.7.5 Loadable Modules

The Linux kernel has loadable modules, and these modules can also be unloaded. After a given module has been unloaded, any attempt to call one of its functions results in a segmentation fault. The module-unload functions must therefore cancel any delayed calls to loadable-module functions, for example, any outstanding mod_timer() must be dealt with via timer_shutdown_sync() or similar.

Unfortunately, there is no way to cancel an RCU callback; once you invoke call_rcu(), the callback function is eventually going to be invoked, unless the system goes down first. Because it is normally considered socially irresponsible to crash the system in response to a module unload request, we need some other way to deal with in-flight RCU callbacks.

RCU therefore provides rcu_barrier(), which waits until all in-flight RCU callbacks have been invoked. If a module uses call_rcu(), its exit function should therefore prevent any future invocation of call_rcu(), then invoke rcu_barrier(). In theory, the underlying module-unload code could invoke rcu_barrier() unconditionally, but in practice this would incur unacceptable latencies.

Nikita Danilov noted this requirement for an analogous filesystem-unmount situation, and Dipankar Sarma incorporated rcu_barrier() into RCU. The need for rcu_barrier() for module unloading became apparent later.

Important: The rcu_barrier() function is not, repeat, *not*, obligated to wait for a grace period. It is instead only required to wait for RCU callbacks that have already been posted. Therefore, if there are no RCU callbacks posted anywhere in the system, rcu_barrier() is within its rights to return immediately. Even if there are callbacks posted, rcu_barrier() does not necessarily need to wait for a grace period.

Quick Quiz:

Wait a minute! Each RCU callbacks must wait for a grace period to complete, and rcu_barrier() must wait for each pre-existing callback to be invoked. Doesn't rcu_barrier() therefore need to wait for a full grace period if there is even one callback posted anywhere in the system?

Answer:

Absolutely not!!! Yes, each RCU callbacks must wait for a grace period to complete, but it might well be partly (or even completely) finished waiting by the time rcu_barrier() is invoked. In that case, rcu_barrier() need only wait for the remaining portion of the grace period to elapse. So even if there are quite a few callbacks posted, rcu_barrier() might well return quite quickly.

So if you need to wait for a grace period as well as for all pre-existing callbacks, you will need to invoke both synchronize_rcu() and rcu_barrier(). If latency is a concern, you can always use workqueues to invoke them concurrently.

17.7.6 Hotplug CPU

The Linux kernel supports CPU hotplug, which means that CPUs can come and go. It is of course illegal to use any RCU API member from an offline CPU, with the exception of *SRCU* read-side critical sections. This requirement was present from day one in DYNIX/ptx, but on the other hand, the Linux kernel's CPU-hotplug implementation is "interesting."

The Linux-kernel CPU-hotplug implementation has notifiers that are used to allow the various kernel subsystems (including RCU) to respond appropriately to a given CPU-hotplug operation. Most RCU operations may be invoked from CPU-hotplug notifiers, including even synchronous grace-period operations such as (synchronize_rcu() and synchronize_rcu_expedited()). However, these synchronous operations do block and therefore cannot be invoked from notifiers that execute via stop_machine(), specifically those between the `CPUHP_AP_OFFLINE` and `CPUHP_AP_ONLINE` states.

In addition, all-callback-wait operations such as rcu_barrier() may not be invoked from any CPU-hotplug notifier. This restriction is due to the fact that there are phases of CPU-hotplug operations where the outgoing CPU's callbacks will not be invoked until after the CPU-hotplug operation ends, which could also result in deadlock. Furthermore, rcu_barrier() blocks CPU-hotplug operations during its execution, which results in another type of deadlock when invoked from a CPU-hotplug notifier.

Finally, RCU must avoid deadlocks due to interaction between hotplug, timers and grace period processing. It does so by maintaining its own set of books that duplicate the centrally maintained `cpu_online_mask`, and also by reporting quiescent states explicitly when a CPU goes offline. This explicit reporting of quiescent states avoids any need for the force-quiescent-state loop (FQS) to report quiescent states for offline CPUs. However, as a debugging measure, the FQS loop does splat if offline CPUs block an RCU grace period for too long.

An offline CPU's quiescent state will be reported either:

1. As the CPU goes offline using RCU's hotplug notifier (rcu_report_dead()).
2. When grace period initialization (rcu_gp_init()) detects a race either with CPU offlining or with a task unblocking on a leaf `rcu_node` structure whose CPUs are all offline.

The CPU-online path (rcu_cpu_starting()) should never need to report a quiescent state for an offline CPU. However, as a debugging measure, it does emit a warning if a quiescent state was not already reported for that CPU.

During the checking/modification of RCU's hotplug bookkeeping, the corresponding CPU's leaf node lock is held. This avoids race conditions between RCU's hotplug notifier hooks, the grace period initialization code, and the FQS loop, all of which refer to or modify this bookkeeping.

17.7.7 Scheduler and RCU

RCU makes use of kthreads, and it is necessary to avoid excessive CPU-time accumulation by these kthreads. This requirement was no surprise, but RCU's violation of it when running context-switch-heavy workloads when built with `CONFIG_NO_HZ_FULL=y` did come as a surprise [PDF]. RCU has made good progress towards meeting this requirement, even for context-switch-heavy `CONFIG_NO_HZ_FULL=y` workloads, but there is room for further improvement.

There is no longer any prohibition against holding any of scheduler's runqueue or priority-inheritance spinlocks across an rcu_read_unlock(), even if interrupts and preemption were enabled somewhere within the corresponding RCU read-side critical section. Therefore, it is now perfectly legal to execute rcu_read_lock() with preemption enabled, acquire one of the scheduler locks, and hold that lock across the matching rcu_read_unlock().

Similarly, the RCU flavor consolidation has removed the need for negative nesting. The fact that interrupt-disabled regions of code act as RCU read-side critical sections implicitly avoids earlier issues that used to result in destructive recursion via interrupt handler's use of RCU.

17.7.8 Tracing and RCU

It is possible to use tracing on RCU code, but tracing itself uses RCU. For this reason, rcu_dereference_raw_check() is provided for use by tracing, which avoids the destructive recursion that could otherwise ensue. This API is also used by virtualization in some architectures, where RCU readers execute in environments in which tracing cannot be used. The tracing folks both located the requirement and provided the needed fix, so this surprise requirement was relatively painless.

17.7.9 Accesses to User Memory and RCU

The kernel needs to access user-space memory, for example, to access data referenced by system-call parameters. The get_user() macro does this job.

However, user-space memory might well be paged out, which means that get_user() might well page-fault and thus block while waiting for the resulting I/O to complete. It would be a very bad thing for the compiler to reorder a get_user() invocation into an RCU read-side critical section.

For example, suppose that the source code looked like this:

```
rcu_read_lock();
p = rcu_dereference(gp);
v = p->value;
rcu_read_unlock();
get_user(user_v, user_p);
do_something_with(v, user_v);
```

The compiler must not be permitted to transform this source code into the following:

```
1 rcu_read_lock();
2 p = rcu_dereference(gp);
3 get_user(user_v, user_p); // BUG: POSSIBLE PAGE FAULT!!!
4 v = p->value;
5 rcu_read_unlock();
6 do_something_with(v, user_v);
```

If the compiler did make this transformation in a `CONFIG_PREEMPTION=n` kernel build, and if get_user() did page fault, the result would be a quiescent state in the middle of an RCU read-side critical section. This misplaced quiescent state could result in line 4 being a use-after-free access, which could be bad for your kernel's actuarial statistics. Similar examples can be constructed with the call to get_user() preceding the rcu_read_lock().

Unfortunately, get_user() doesn't have any particular ordering properties, and in some architectures the underlying `asm` isn't even marked `volatile`. And even if it was marked `volatile`, the above access to `p->value` is not volatile, so the compiler would not have any reason to keep those two accesses in order.

Therefore, the Linux-kernel definitions of rcu_read_lock() and rcu_read_unlock() must act as compiler barriers, at least for outermost instances of rcu_read_lock() and rcu_read_unlock() within a nested set of RCU read-side critical sections.

17.7.10 Energy Efficiency

Interrupting idle CPUs is considered socially unacceptable, especially by people with battery-powered embedded systems. RCU therefore conserves energy by detecting which CPUs are idle, including tracking CPUs that have been interrupted from idle. This is a large part of the energy-efficiency requirement, so I learned of this via an irate phone call.

Because RCU avoids interrupting idle CPUs, it is illegal to execute an RCU read-side critical section on an idle CPU. (Kernels built with `CONFIG_PROVE_RCU=y` will splat if you try it.)

It is similarly socially unacceptable to interrupt an `nohz_full` CPU running in userspace. RCU must therefore track `nohz_full` userspace execution. RCU must therefore be able to sample state at two points in time, and be able to determine whether or not some other CPU spent any time idle and/or executing in userspace.

These energy-efficiency requirements have proven quite difficult to understand and to meet, for example, there have been more than five clean-sheet rewrites of RCU's energy-efficiency code, the last of which was finally able to demonstrate real energy savings running on real hardware [PDF]. As noted earlier, I learned of many of these requirements via angry phone calls: Flaming me on the Linux-kernel mailing list was apparently not sufficient to fully vent their ire at RCU's energy-efficiency bugs!

17.7.11 Scheduling-Clock Interrupts and RCU

The kernel transitions between in-kernel non-idle execution, userspace execution, and the idle loop. Depending on kernel configuration, RCU handles these states differently:

HZ Kconfig	In-Kernel	Usermode	Idle
`HZ_PERIODIC`	Can rely on scheduling-clock interrupt.	Can rely on scheduling-clock interrupt and its detection of interrupt from usermode.	Can rely on RCU's dyntick-idle detection.
`NO_HZ_IDLE`	Can rely on scheduling-clock interrupt.	Can rely on scheduling-clock interrupt and its detection of interrupt from usermode.	Can rely on RCU's dyntick-idle detection.
`NO_HZ_FULL`	Can only sometimes rely on scheduling-clock interrupt. In other cases, it is necessary to bound kernel execution times and/or use IPIs.	Can rely on RCU's dyntick-idle detection.	Can rely on RCU's dyntick-idle detection.

Quick Quiz:

Why can't `NO_HZ_FULL` in-kernel execution rely on the scheduling-clock interrupt, just like `HZ_PERIODIC` and `NO_HZ_IDLE` do?

Answer:

Because, as a performance optimization, `NO_HZ_FULL` does not necessarily re-enable the scheduling-clock interrupt on entry to each and every system call.

However, RCU must be reliably informed as to whether any given CPU is currently in the idle loop, and, for `NO_HZ_FULL`, also whether that CPU is executing in usermode, as discussed *earlier*. It also requires that the scheduling-clock interrupt be enabled when RCU needs it to be:

1. If a CPU is either idle or executing in usermode, and RCU believes it is non-idle, the scheduling-clock tick had better be running. Otherwise, you will get RCU CPU stall warnings. Or at best, very long (11-second) grace periods, with a pointless IPI waking the CPU from time to time.
2. If a CPU is in a portion of the kernel that executes RCU read-side critical sections, and RCU believes this CPU to be idle, you will get random memory corruption. **DON'T DO THIS!!!** This is one reason to test with lockdep, which will complain about this sort of thing.
3. If a CPU is in a portion of the kernel that is absolutely positively no-joking guaranteed to never execute any RCU read-side critical sections, and RCU believes this CPU to be idle, no problem. This sort of thing is used by some architectures for light-weight exception handlers, which can then avoid the overhead of ct_irq_enter() and ct_irq_exit() at exception entry and exit, respectively. Some go further and avoid the entireties of irq_enter() and irq_exit(). Just make very sure you are running some of your tests with `CONFIG_PROVE_RCU=y`, just in case one of your code paths was in fact joking about not doing RCU read-side critical sections.
4. If a CPU is executing in the kernel with the scheduling-clock interrupt disabled and RCU believes this CPU to be non-idle, and if the CPU goes idle (from an RCU perspective) every few jiffies, no problem. It is usually OK for there to be the occasional gap between idle periods of up to a second or so. If the gap grows too long, you get RCU CPU stall warnings.
5. If a CPU is either idle or executing in usermode, and RCU believes it to be idle, of course no problem.
6. If a CPU is executing in the kernel, the kernel code path is passing through quiescent states at a reasonable frequency (preferably about once per few jiffies, but the occasional excursion to a second or so is usually OK) and the scheduling-clock interrupt is enabled, of course no problem. If the gap between a successive pair of quiescent states grows too long, you get RCU CPU stall warnings.

Quick Quiz:

But what if my driver has a hardware interrupt handler that can run for many seconds? I cannot invoke schedule() from an hardware interrupt handler, after all!

Answer:

One approach is to do `ct_irq_exit();ct_irq_enter();` every so often. But given that long-running interrupt handlers can cause other problems, not least for response time, shouldn't you work to keep your interrupt handler's runtime within reasonable bounds?

But as long as RCU is properly informed of kernel state transitions between in-kernel execution, usermode execution, and idle, and as long as the scheduling-clock interrupt is enabled when RCU needs it to be, you can rest assured that the bugs you encounter will be in some other part of RCU or some other part of the kernel!

17.7.12 Memory Efficiency

Although small-memory non-realtime systems can simply use Tiny RCU, code size is only one aspect of memory efficiency. Another aspect is the size of the `rcu_head` structure used by call_rcu() and kfree_rcu(). Although this structure contains nothing more than a pair of pointers, it does appear in many RCU-protected data structures, including some that are size critical. The `page` structure is a case in point, as evidenced by the many occurrences of the `union` keyword within that structure.

This need for memory efficiency is one reason that RCU uses hand-crafted singly linked lists to track the `rcu_head` structures that are waiting for a grace period to elapse. It is also the reason why `rcu_head` structures do not contain debug information, such as fields tracking the file and line of the call_rcu() or kfree_rcu() that posted them. Although this information might appear in debug-only kernel builds at some point, in the meantime, the `->func` field will often provide the needed debug information.

However, in some cases, the need for memory efficiency leads to even more extreme measures. Returning to the `page` structure, the `rcu_head` field shares storage with a great many other structures that are used at various points in the corresponding page's lifetime. In order to correctly resolve certain race conditions, the Linux kernel's memory-management subsystem needs a particular bit to remain zero during all phases of grace-period processing, and that bit happens to map to the bottom bit of the `rcu_head` structure's `->next` field. RCU makes this guarantee as long as call_rcu() is used to post the callback, as opposed to kfree_rcu() or some future "lazy" variant of call_rcu() that might one day be created for energy-efficiency purposes.

That said, there are limits. RCU requires that the `rcu_head` structure be aligned to a two-byte boundary, and passing a misaligned `rcu_head` structure to one of the call_rcu() family of functions will result in a splat. It is therefore necessary to exercise caution when packing structures containing fields of type `rcu_head`. Why not a four-byte or even eight-byte alignment requirement? Because the m68k architecture provides only two-byte alignment, and thus acts as alignment's least common denominator.

The reason for reserving the bottom bit of pointers to `rcu_head` structures is to leave the door open to "lazy" callbacks whose invocations can safely be deferred. Deferring invocation could potentially have energy-efficiency benefits, but only if the rate of non-lazy callbacks decreases significantly for some important workload. In the meantime, reserving the bottom bit keeps this option open in case it one day becomes useful.

17.7.13 Performance, Scalability, Response Time, and Reliability

Expanding on the *earlier discussion*, RCU is used heavily by hot code paths in performance-critical portions of the Linux kernel's networking, security, virtualization, and scheduling code paths. RCU must therefore use efficient implementations, especially in its read-side primitives. To that end, it would be good if preemptible RCU's implementation of rcu_read_lock() could be inlined, however, doing this requires resolving `#include` issues with the `task_struct` structure.

The Linux kernel supports hardware configurations with up to 4096 CPUs, which means that RCU must be extremely scalable. Algorithms that involve frequent acquisitions of global locks or frequent atomic operations on global variables simply cannot be tolerated within the RCU implementation. RCU therefore makes heavy use of a combining tree based on the `rcu_node` structure. RCU is required to tolerate all CPUs continuously invoking any combination of RCU's runtime primitives with minimal per-operation overhead. In fact, in many cases, increasing load must *decrease* the per-operation overhead, witness the batching optimizations for synchronize_rcu(), call_rcu(), synchronize_rcu_expedited(), and rcu_barrier(). As a general rule, RCU must cheerfully accept whatever the rest of the Linux kernel decides to throw at it.

The Linux kernel is used for real-time workloads, especially in conjunction with the -rt patchset. The real-time-latency response requirements are such that the traditional approach of disabling preemption across RCU read-side critical sections is inappropriate. Kernels built with `CONFIG_PREEMPTION=y` therefore use an RCU implementation that allows RCU read-side critical sections to be preempted. This requirement made its presence known after users made it clear that an earlier real-time patch did not meet their needs, in conjunction with some RCU issues encountered by a very early version of the -rt patchset.

In addition, RCU must make do with a sub-100-microsecond real-time latency budget. In fact, on smaller systems with the -rt patchset, the Linux kernel provides sub-20-microsecond real-time latencies for the whole kernel, including RCU. RCU's scalability and latency must therefore be sufficient for these sorts of configurations. To my surprise, the sub-100-microsecond real-time latency budget applies to even the largest systems [PDF], up to and including systems with 4096 CPUs. This real-time requirement motivated the grace-period kthread, which also simplified handling of a number of race conditions.

RCU must avoid degrading real-time response for CPU-bound threads, whether executing in usermode (which is one use case for `CONFIG_NO_HZ_FULL=y`) or in the kernel. That said, CPU-bound loops in the kernel must execute cond_resched() at least once per few tens of milliseconds in order to avoid receiving an IPI from RCU.

Finally, RCU's status as a synchronization primitive means that any RCU failure can result in arbitrary memory corruption that can be extremely difficult to debug. This means that RCU must be extremely reliable, which in practice also means that RCU must have an aggressive stress-test suite. This stress-test suite is called `rcutorture`.

Although the need for `rcutorture` was no surprise, the current immense popularity of the Linux kernel is posing interesting—and perhaps unprecedented—validation challenges. To see this, keep in mind that there are well over one billion instances of the Linux kernel running today, given Android smartphones, Linux-powered televisions, and servers. This number can be expected to increase sharply with the advent of the celebrated Internet of Things.

Suppose that RCU contains a race condition that manifests on average once per million years of runtime. This bug will be occurring about three times per *day* across the installed base. RCU could simply hide behind hardware error rates, given that no one should really expect their smartphone to last for a million years. However, anyone taking too much comfort from this thought should consider the fact that in most jurisdictions, a successful multi-year test of a given mechanism, which might include a Linux kernel, suffices for a number of types of safety-critical certifications. In fact, rumor has it that the Linux kernel is already being used in production for safety-critical applications. I don't know about you, but I would feel quite bad if a bug in RCU killed someone. Which might explain my recent focus on validation and verification.

17.8 Other RCU Flavors

One of the more surprising things about RCU is that there are now no fewer than five *flavors*, or API families. In addition, the primary flavor that has been the sole focus up to this point has two different implementations, non-preemptible and preemptible. The other four flavors are listed below, with requirements for each described in a separate section.

1. *Bottom-Half Flavor (Historical)*
2. *Sched Flavor (Historical)*
3. *Sleepable RCU*
4. *Tasks RCU*

17.8.1 Bottom-Half Flavor (Historical)

The RCU-bh flavor of RCU has since been expressed in terms of the other RCU flavors as part of a consolidation of the three flavors into a single flavor. The read-side API remains, and continues to disable softirq and to be accounted for by lockdep. Much of the material in this section is therefore strictly historical in nature.

The softirq-disable (AKA "bottom-half", hence the "_bh" abbreviations) flavor of RCU, or *RCU-bh*, was developed by Dipankar Sarma to provide a flavor of RCU that could withstand the network-based denial-of-service attacks researched by Robert Olsson. These attacks placed so much networking load on the system that some of the CPUs never exited softirq execution, which in turn prevented those CPUs from ever executing a context switch, which, in the RCU implementation of that time, prevented grace periods from ever ending. The result was an out-of-memory condition and a system hang.

The solution was the creation of RCU-bh, which does local_bh_disable() across its read-side critical sections, and which uses the transition from one type of softirq processing to another as a quiescent state in addition to context switch, idle, user mode, and offline. This means that RCU-bh grace periods can complete even when some of the CPUs execute in softirq indefinitely, thus allowing algorithms based on RCU-bh to withstand network-based denial-of-service attacks.

Because rcu_read_lock_bh() and rcu_read_unlock_bh() disable and re-enable softirq handlers, any attempt to start a softirq handlers during the RCU-bh read-side critical section will be deferred. In this case, rcu_read_unlock_bh() will invoke softirq processing, which can take considerable time. One can of course argue that this softirq overhead should be associated with the code following the RCU-bh read-side critical section rather than rcu_read_unlock_bh(), but the fact is that most profiling tools cannot be expected to make this sort of fine distinction. For example, suppose that a three-millisecond-long RCU-bh read-side critical section executes during a time of heavy networking load. There will very likely be an attempt to invoke at least one softirq handler during that three milliseconds, but any such invocation will be delayed until the time of the rcu_read_unlock_bh(). This can of course make it appear at first glance as if rcu_read_unlock_bh() was executing very slowly.

The RCU-bh API includes rcu_read_lock_bh(), rcu_read_unlock_bh(), rcu_dereference_bh(), rcu_dereference_bh_check(), and rcu_read_lock_bh_held(). However, the old RCU-bh update-side APIs are now gone, replaced by synchronize_rcu(), synchronize_rcu_expedited(), call_rcu(), and rcu_barrier(). In addition, anything that disables bottom halves also marks an RCU-bh read-side critical section, including local_bh_disable() and local_bh_enable(), local_irq_save() and local_irq_restore(), and so on.

17.8.2 Sched Flavor (Historical)

The RCU-sched flavor of RCU has since been expressed in terms of the other RCU flavors as part of a consolidation of the three flavors into a single flavor. The read-side API remains, and continues to disable preemption and to be accounted for by lockdep. Much of the material in this section is therefore strictly historical in nature.

Before preemptible RCU, waiting for an RCU grace period had the side effect of also waiting for all pre-existing interrupt and NMI handlers. However, there are legitimate preemptible-RCU implementations that do not have this property, given that any point in the code outside of an RCU read-side critical section can be a quiescent state. Therefore, *RCU-sched* was created, which follows "classic" RCU in that an RCU-sched grace period waits for pre-existing interrupt and NMI handlers. In kernels built with `CONFIG_PREEMPTION=n`, the RCU and RCU-sched APIs have identical implementations, while kernels built with `CONFIG_PREEMPTION=y` provide a separate implementation for each.

Note well that in `CONFIG_PREEMPTION=y` kernels, rcu_read_lock_sched() and rcu_read_unlock_sched() disable and re-enable preemption, respectively. This means that if there was a preemption attempt during the RCU-sched read-side critical section, rcu_read_unlock_sched() will enter the scheduler, with all the latency and overhead entailed. Just as with rcu_read_unlock_bh(), this can make it look as if rcu_read_unlock_sched() was executing very slowly. However, the highest-priority task won't be preempted, so that task will enjoy low-overhead rcu_read_unlock_sched() invocations.

The RCU-sched API includes rcu_read_lock_sched(), rcu_read_unlock_sched(), rcu_read_lock_sched_notrace(), rcu_read_unlock_sched_notrace(), rcu_dereference_sched(), rcu_dereference_sched_check(), and rcu_read_lock_sched_held(). However, the old RCU-sched update-side APIs are now gone, replaced by synchronize_rcu(), synchronize_rcu_expedited(), call_rcu(), and rcu_barrier(). In addition, anything that disables preemption also marks an RCU-sched read-side critical section, including preempt_disable() and preempt_enable(), local_irq_save() and local_irq_restore(), and so on.

17.8.3 Sleepable RCU

For well over a decade, someone saying "I need to block within an RCU read-side critical section" was a reliable indication that this someone did not understand RCU. After all, if you are always blocking in an RCU read-side critical section, you can probably afford to use a higher-overhead synchronization mechanism. However, that changed with the advent of the Linux kernel's notifiers, whose RCU read-side critical sections almost never sleep, but sometimes need to. This resulted in the introduction of sleepable RCU, or *SRCU*.

SRCU allows different domains to be defined, with each such domain defined by an instance of an `srcu_struct` structure. A pointer to this structure must be passed in to each SRCU function, for example, `synchronize_srcu(&ss)`, where `ss` is the `srcu_struct` structure. The key benefit of these domains is that a slow SRCU reader in one domain does not delay an SRCU grace period in some other domain. That said, one consequence of these domains is that read-side code must pass a "cookie" from srcu_read_lock() to srcu_read_unlock(), for example, as follows:

```
1 int idx;
2
3 idx = srcu_read_lock(&ss);
4 do_something();
5 srcu_read_unlock(&ss, idx);
```

As noted above, it is legal to block within SRCU read-side critical sections, however, with great power comes great responsibility. If you block forever in one of a given domain's SRCU read-side critical sections, then that domain's grace periods will also be blocked forever. Of course, one good way to block forever is to deadlock, which can happen if any operation in a given domain's SRCU read-side critical section can wait, either directly or indirectly, for that domain's grace period to elapse. For example, this results in a self-deadlock:

```
1 int idx;
2
3 idx = srcu_read_lock(&ss);
4 do_something();
5 synchronize_srcu(&ss);
6 srcu_read_unlock(&ss, idx);
```

However, if line 5 acquired a mutex that was held across a synchronize_srcu() for domain `ss`, deadlock would still be possible. Furthermore, if line 5 acquired a mutex that was held across a synchronize_srcu() for some other domain `ss1`, and if an `ss1`-domain SRCU read-side critical section acquired another mutex that was held across as `ss`-domain synchronize_srcu(), deadlock would again be possible. Such a deadlock cycle could extend across an arbitrarily large number of different SRCU domains. Again, with great power comes great responsibility.

Unlike the other RCU flavors, SRCU read-side critical sections can run on idle and even offline CPUs. This ability requires that srcu_read_lock() and srcu_read_unlock() contain memory barriers, which means that SRCU readers will run a bit slower than would RCU readers. It also motivates the smp_mb__after_srcu_read_unlock() API, which, in combination with srcu_read_unlock(), guarantees a full memory barrier.

Also unlike other RCU flavors, synchronize_srcu() may **not** be invoked from CPU-hotplug notifiers, due to the fact that SRCU grace periods make use of timers and the possibility of timers being temporarily "stranded" on the outgoing CPU. This stranding of timers means that timers posted to the outgoing CPU will not fire until late in the CPU-hotplug process. The problem is that if a notifier is waiting on an SRCU grace period, that grace period is waiting on a timer, and that timer is stranded on the outgoing CPU, then the notifier will never be awakened, in other words, deadlock has occurred. This same situation of course also prohibits srcu_barrier() from being invoked from CPU-hotplug notifiers.

SRCU also differs from other RCU flavors in that SRCU's expedited and non-expedited grace periods are implemented by the same mechanism. This means that in the current SRCU implementation, expediting a future grace period has the side effect of expediting all prior grace periods that have not yet completed. (But please note that this is a property of the current implementation, not necessarily of future implementations.) In addition, if SRCU has been idle for longer than the interval specified by the `srcutree.exp_holdoff` kernel boot parameter (25 microseconds by default), and if a synchronize_srcu() invocation ends this idle period, that invocation will be automatically expedited.

As of v4.12, SRCU's callbacks are maintained per-CPU, eliminating a locking bottleneck present in prior kernel versions. Although this will allow users to put much heavier stress on call_srcu(), it is important to note that SRCU does not yet take any special steps to deal with callback flooding. So if you are posting (say) 10,000 SRCU callbacks per second per CPU, you are probably totally OK, but if you intend to post (say) 1,000,000 SRCU callbacks per second per CPU, please run some tests first. SRCU just might need a few adjustment to deal with that sort of load. Of course, your mileage may vary based on the speed of your CPUs and the size of your memory.

The SRCU API includes srcu_read_lock(), srcu_read_unlock(), srcu_dereference(), srcu_dereference_check(), synchronize_srcu(), synchronize_srcu_expedited(), call_srcu(), srcu_barrier(), and srcu_read_lock_held(). It also includes DEFINE_SRCU(), DEFINE_STATIC_SRCU(), and init_srcu_struct() APIs for defining and initializing `srcu_struct` structures.

More recently, the SRCU API has added polling interfaces:

1. start_poll_synchronize_srcu() returns a cookie identifying the completion of a future SRCU grace period and ensures that this grace period will be started.
2. poll_state_synchronize_srcu() returns `true` iff the specified cookie corresponds to an already-completed SRCU grace period.
3. get_state_synchronize_srcu() returns a cookie just like start_poll_synchronize_srcu() does, but differs in that it does nothing to ensure that any future SRCU grace period will be started.

These functions are used to avoid unnecessary SRCU grace periods in certain types of buffer-cache algorithms having multi-stage age-out mechanisms. The idea is that by the time the block has aged completely from the cache, an SRCU grace period will be very likely to have elapsed.

17.8.4 Tasks RCU

Some forms of tracing use "trampolines" to handle the binary rewriting required to install different types of probes. It would be good to be able to free old trampolines, which sounds like a job for some form of RCU. However, because it is necessary to be able to install a trace anywhere in the code, it is not possible to use read-side markers such as rcu_read_lock() and rcu_read_unlock(). In addition, it does not work to have these markers in the trampoline itself, because there would need to be instructions following rcu_read_unlock(). Although synchronize_rcu() would guarantee that execution reached the rcu_read_unlock(), it would not be able to guarantee that execution had completely left the trampoline. Worse yet, in some situations the trampoline's protection must extend a few instructions *prior* to execution reaching the trampoline. For example, these few instructions might calculate the address of the trampoline, so that entering the trampoline would be pre-ordained a surprisingly long time before execution actually reached the trampoline itself.

The solution, in the form of Tasks RCU, is to have implicit read-side critical sections that are delimited by voluntary context switches, that is, calls to schedule(), cond_resched(), and synchronize_rcu_tasks(). In addition, transitions to and from userspace execution also delimit tasks-RCU read-side critical sections.

The tasks-RCU API is quite compact, consisting only of call_rcu_tasks(), synchronize_rcu_tasks(), and rcu_barrier_tasks(). In `CONFIG_PREEMPTION=n` kernels, trampolines cannot be preempted, so these APIs map to call_rcu(), synchronize_rcu(), and rcu_barrier(), respectively. In `CONFIG_PREEMPTION=y` kernels, trampolines can be preempted, and these three APIs are therefore implemented by separate functions that check for voluntary context switches.

17.8.5 Tasks Rude RCU

Some forms of tracing need to wait for all preemption-disabled regions of code running on any online CPU, including those executed when RCU is not watching. This means that synchronize_rcu() is insufficient, and Tasks Rude RCU must be used instead. This flavor of RCU does its work by forcing a workqueue to be scheduled on each online CPU, hence the "Rude" moniker. And this operation is considered to be quite rude by real-time workloads that don't want their `nohz_full` CPUs receiving IPIs and by battery-powered systems that don't want their idle CPUs to be awakened.

The tasks-rude-RCU API is also reader-marking-free and thus quite compact, consisting of call_rcu_tasks_rude(), synchronize_rcu_tasks_rude(), and rcu_barrier_tasks_rude().

17.8.6 Tasks Trace RCU

Some forms of tracing need to sleep in readers, but cannot tolerate SRCU's read-side overhead, which includes a full memory barrier in both srcu_read_lock() and srcu_read_unlock(). This need is handled by a Tasks Trace RCU that uses scheduler locking and IPIs to synchronize with readers. Real-time systems that cannot tolerate IPIs may build their kernels with `CONFIG_TASKS_TRACE_RCU_READ_MB=y`, which avoids the IPIs at the expense of adding full memory barriers to the read-side primitives.

The tasks-trace-RCU API is also reasonably compact, consisting of rcu_read_lock_trace(), rcu_read_unlock_trace(), rcu_read_lock_trace_held(), call_rcu_tasks_trace(), synchronize_rcu_tasks_trace(), and rcu_barrier_tasks_trace().

17.9 Possible Future Changes

One of the tricks that RCU uses to attain update-side scalability is to increase grace-period latency with increasing numbers of CPUs. If this becomes a serious problem, it will be necessary to rework the grace-period state machine so as to avoid the need for the additional latency.

RCU disables CPU hotplug in a few places, perhaps most notably in the rcu_barrier() operations. If there is a strong reason to use rcu_barrier() in CPU-hotplug notifiers, it will be necessary to avoid disabling CPU hotplug. This would introduce some complexity, so there had better be a *very* good reason.

The tradeoff between grace-period latency on the one hand and interruptions of other CPUs on the other hand may need to be re-examined. The desire is of course for zero grace-period latency as well as zero interprocessor interrupts undertaken during an expedited grace period operation. While this ideal is unlikely to be achievable, it is quite possible that further improvements can be made.

The multiprocessor implementations of RCU use a combining tree that groups CPUs so as to reduce lock contention and increase cache locality. However, this combining tree does not spread its memory across NUMA nodes nor does it align the CPU groups with hardware features such as sockets or cores. Such spreading and alignment is currently believed to be unnecessary because the hotpath read-side primitives do not access the combining tree, nor does call_rcu() in the common case. If you believe that your architecture needs such spreading and alignment, then your architecture should also benefit from the `rcutree.rcu_fanout_leaf` boot parameter, which can be set to the number of CPUs in a socket, NUMA node, or whatever. If the number of CPUs is too large, use a fraction of the number of CPUs. If the number of CPUs is a large prime number, well, that certainly is an "interesting" architectural choice! More flexible arrangements might be considered, but only if `rcutree.rcu_fanout_leaf` has proven inadequate, and only if the inadequacy has been demonstrated by a carefully run and realistic system-level workload.

Please note that arrangements that require RCU to remap CPU numbers will require extremely good demonstration of need and full exploration of alternatives.

RCU's various kthreads are reasonably recent additions. It is quite likely that adjustments will be required to more gracefully handle extreme loads. It might also be necessary to be able to relate CPU utilization by RCU's kthreads and softirq handlers to the code that instigated this CPU utilization. For example, RCU callback overhead might be charged back to the originating call_rcu() instance, though probably not in production kernels.

Additional work may be required to provide reasonable forward-progress guarantees under heavy load for grace periods and for callback invocation.

17.10 Summary

This document has presented more than two decade's worth of RCU requirements. Given that the requirements keep changing, this will not be the last word on this subject, but at least it serves to get an important subset of the requirements set forth.

17.11 Acknowledgments

I am grateful to Steven Rostedt, Lai Jiangshan, Ingo Molnar, Oleg Nesterov, Borislav Petkov, Peter Zijlstra, Boqun Feng, and Andy Lutomirski for their help in rendering this article human readable, and to Michelle Rankin for her support of this effort. Other contributions are acknowledged in the Linux kernel's git archive.

CHAPTER

EIGHTEEN

A TOUR THROUGH TREE_RCU'S DATA STRUCTURES [LWN.NET]

December 18, 2016

This article was contributed by Paul E. McKenney

18.1 Introduction

This document describes RCU's major data structures and their relationship to each other.

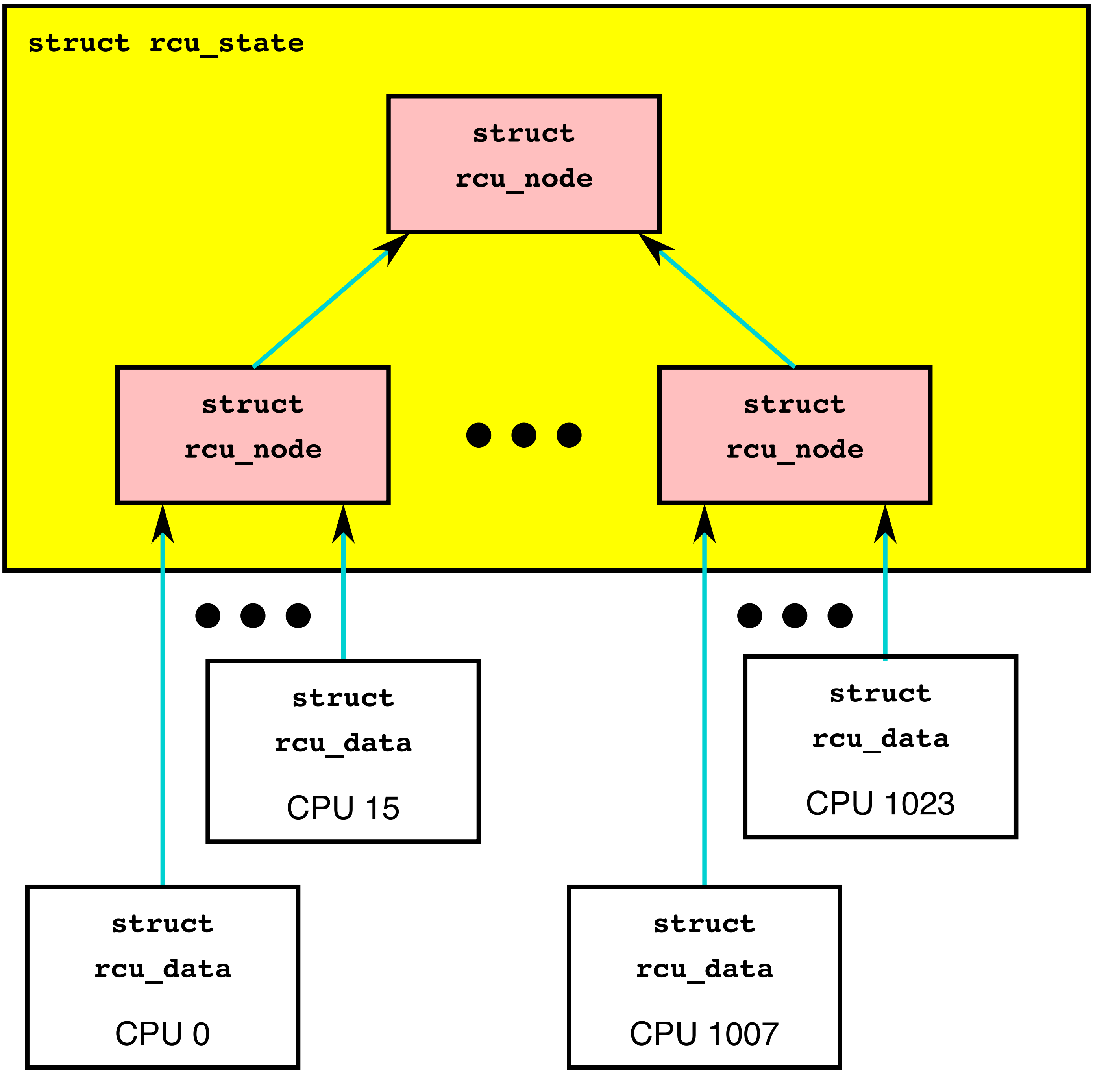
18.2 Data-Structure Relationships

RCU is for all intents and purposes a large state machine, and its data structures maintain the state in such a way as to allow RCU readers to execute extremely quickly, while also processing the RCU grace periods requested by updaters in an efficient and extremely scalable fashion. The efficiency and scalability of RCU updaters is provided primarily by a combining tree, as shown below:

This diagram shows an enclosing `rcu_state` structure containing a tree of `rcu_node` structures. Each leaf node of the `rcu_node` tree has up to 16 `rcu_data` structures associated with it, so that there are `NR_CPUS` number of `rcu_data` structures, one for each possible CPU. This structure is adjusted at boot time, if needed, to handle the common case where `nr_cpu_ids` is much less than `NR_CPUs`. For example, a number of Linux distributions set `NR_CPUs=4096`, which results in a three-level `rcu_node` tree. If the actual hardware has only 16 CPUs, RCU will adjust itself at boot time, resulting in an `rcu_node` tree with only a single node.

The purpose of this combining tree is to allow per-CPU events such as quiescent states, dyntick-idle transitions, and CPU hotplug operations to be processed efficiently and scalably. Quiescent states are recorded by the per-CPU `rcu_data` structures, and other events are recorded by the leaf-level `rcu_node` structures. All of these events are combined at each level of the tree until finally grace periods are completed at the tree's root `rcu_node` structure. A grace period can be completed at the root once every CPU (or, in the case of `CONFIG_PREEMPT_RCU`, task) has passed through a quiescent state. Once a grace period has completed, record of that fact is propagated back down the tree.

As can be seen from the diagram, on a 64-bit system a two-level tree with 64 leaves can accommodate 1,024 CPUs, with a fanout of 64 at the root and a fanout of 16 at the leaves.


struct rcu_state
struct
rcu_node
struct
rcu_node
struct
rcu_node
struct
rcu_data
CPU 15
struct
rcu_data
CPU 1023
struct
rcu_data
CPU 0
struct
rcu_data
CPU 1007

Quick Quiz:

Why isn't the fanout at the leaves also 64?

Answer:

Because there are more types of events that affect the leaf-level `rcu_node` structures than further up the tree. Therefore, if the leaf `rcu_node` structures have fanout of 64, the contention on these structures' `->structures` becomes excessive. Experimentation on a wide variety of systems has shown that a fanout of 16 works well for the leaves of the `rcu_node` tree.

Of course, further experience with systems having hundreds or thousands of CPUs may demonstrate that the fanout for the non-leaf `rcu_node` structures must also be reduced. Such reduction can be easily carried out when and if it proves necessary. In the meantime, if you are using such a system and running into contention problems on the non-leaf `rcu_node` structures, you may use the `CONFIG_RCU_FANOUT` kernel configuration parameter to reduce the non-leaf fanout as needed.

Kernels built for systems with strong NUMA characteristics might also need to adjust `CONFIG_RCU_FANOUT` so that the domains of the `rcu_node` structures align with hardware boundaries. However, there has thus far been no need for this.

If your system has more than 1,024 CPUs (or more than 512 CPUs on a 32-bit system), then RCU will automatically add more levels to the tree. For example, if you are crazy enough to build a 64-bit system with 65,536 CPUs, RCU would configure the `rcu_node` tree as follows:

RCU currently permits up to a four-level tree, which on a 64-bit system accommodates up to 4,194,304 CPUs, though only a mere 524,288 CPUs for 32-bit systems. On the other hand, you can set both `CONFIG_RCU_FANOUT` and `CONFIG_RCU_FANOUT_LEAF` to be as small as 2, which would result in a 16-CPU test using a 4-level tree. This can be useful for testing large-system capabilities on small test machines.

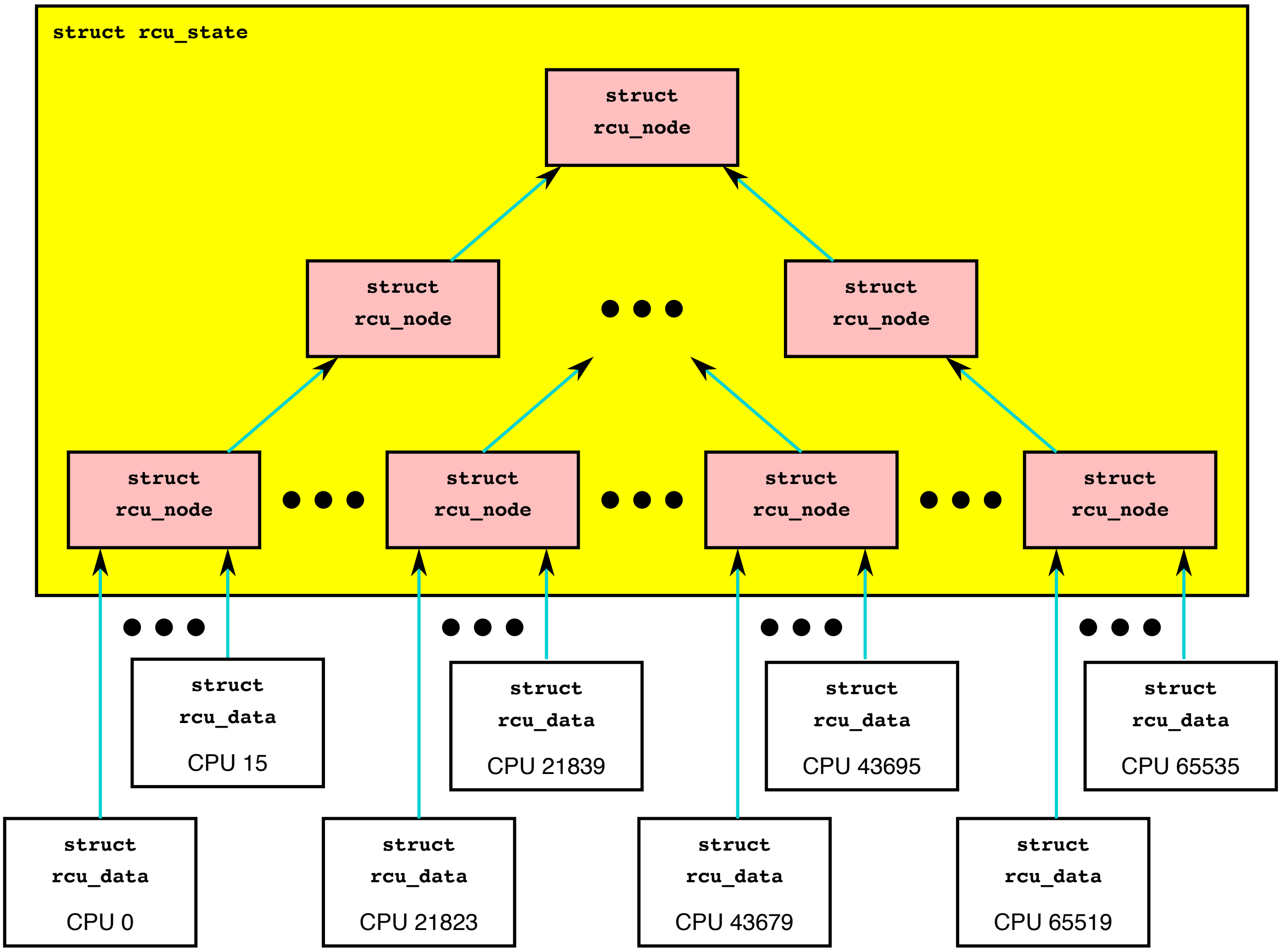
This multi-level combining tree allows us to get most of the performance and scalability benefits of partitioning, even though RCU grace-period detection is inherently a global operation. The trick here is that only the last CPU to report a quiescent state into a given `rcu_node` structure need advance to the `rcu_node` structure at the next level up the tree. This means that at the leaf-level `rcu_node` structure, only one access out of sixteen will progress up the tree. For the internal `rcu_node` structures, the situation is even more extreme: Only one access out of sixty-four will progress up the tree. Because the vast majority of the CPUs do not progress up the tree, the lock contention remains roughly constant up the tree. No matter how many CPUs there are in the system, at most 64 quiescent-state reports per grace period will progress all the way to the root `rcu_node` structure, thus ensuring that the lock contention on that root `rcu_node` structure remains acceptably low.

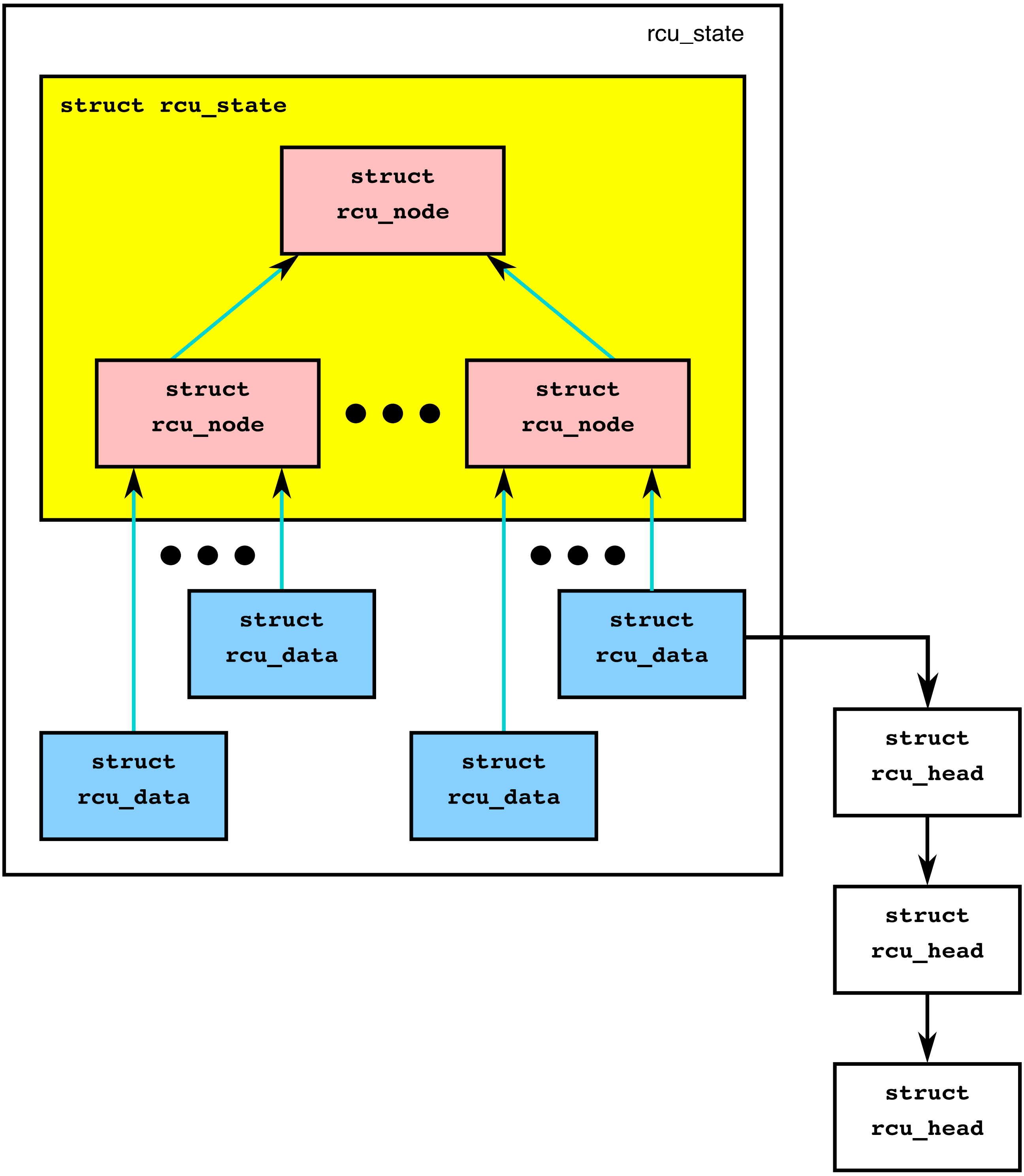
In effect, the combining tree acts like a big shock absorber, keeping lock contention under control at all tree levels regardless of the level of loading on the system.

RCU updaters wait for normal grace periods by registering RCU callbacks, either directly via `call_rcu()` or indirectly via `synchronize_rcu()` and friends. RCU callbacks are represented by `rcu_head` structures, which are queued on `rcu_data` structures while they are waiting for a grace period to elapse, as shown in the following figure:

This figure shows how `TREE_RCU`'s and `PREEMPT_RCU`'s major data structures are related. Lesser data structures will be introduced with the algorithms that make use of them.

Note that each of the data structures in the above figure has its own synchronization:


struct rcu_state
struct rcu_node
struct rcu_node
struct rcu_node
struct rcu_node
struct rcu_node
struct rcu_node
struct rcu_node
struct rcu_data CPU 15
struct rcu_data CPU 21839
struct rcu_data CPU 43695
struct rcu_data CPU 65535
struct rcu_data CPU 0
struct rcu_data CPU 21823
struct rcu_data CPU 43679
struct rcu_data CPU 65519


rcu_state
struct rcu_state
struct rcu_node
struct rcu_node
struct rcu_node
struct rcu_data
struct rcu_data
struct rcu_data
struct rcu_data
struct rcu_head
struct rcu_head
struct rcu_head

1. Each rcu_state structures has a lock and a mutex, and some fields are protected by the corresponding root rcu_node structure's lock.
2. Each rcu_node structure has a spinlock.
3. The fields in rcu_data are private to the corresponding CPU, although a few can be read and written by other CPUs.

It is important to note that different data structures can have very different ideas about the state of RCU at any given time. For but one example, awareness of the start or end of a given RCU grace period propagates slowly through the data structures. This slow propagation is absolutely necessary for RCU to have good read-side performance. If this balkanized implementation seems foreign to you, one useful trick is to consider each instance of these data structures to be a different person, each having the usual slightly different view of reality.

The general role of each of these data structures is as follows:

1. rcu_state: This structure forms the interconnection between the rcu_node and rcu_data structures, tracks grace periods, serves as short-term repository for callbacks orphaned by CPU-hotplug events, maintains rcu_barrier() state, tracks expedited grace-period state, and maintains state used to force quiescent states when grace periods extend too long,
2. rcu_node: This structure forms the combining tree that propagates quiescent-state information from the leaves to the root, and also propagates grace-period information from the root to the leaves. It provides local copies of the grace-period state in order to allow this information to be accessed in a synchronized manner without suffering the scalability limitations that would otherwise be imposed by global locking. In CONFIG_PREEMPT_RCU kernels, it manages the lists of tasks that have blocked while in their current RCU read-side critical section. In CONFIG_PREEMPT_RCU with CONFIG_RCU_BOOST, it manages the per-rcu_node priority-boosting kernel threads (kthreads) and state. Finally, it records CPU-hotplug state in order to determine which CPUs should be ignored during a given grace period.
3. rcu_data: This per-CPU structure is the focus of quiescent-state detection and RCU callback queuing. It also tracks its relationship to the corresponding leaf rcu_node structure to allow more-efficient propagation of quiescent states up the rcu_node combining tree. Like the rcu_node structure, it provides a local copy of the grace-period information to allow for-free synchronized access to this information from the corresponding CPU. Finally, this structure records past dyntick-idle state for the corresponding CPU and also tracks statistics.
4. rcu_head: This structure represents RCU callbacks, and is the only structure allocated and managed by RCU users. The rcu_head structure is normally embedded within the RCU-protected data structure.

If all you wanted from this article was a general notion of how RCU's data structures are related, you are done. Otherwise, each of the following sections give more details on the rcu_state, rcu_node and rcu_data data structures.

18.2.1 The `rcu_state` Structure

The `rcu_state` structure is the base structure that represents the state of RCU in the system. This structure forms the interconnection between the `rcu_node` and `rcu_data` structures, tracks grace periods, contains the lock used to synchronize with CPU-hotplug events, and maintains state used to force quiescent states when grace periods extend too long,

A few of the `rcu_state` structure's fields are discussed, singly and in groups, in the following sections. The more specialized fields are covered in the discussion of their use.

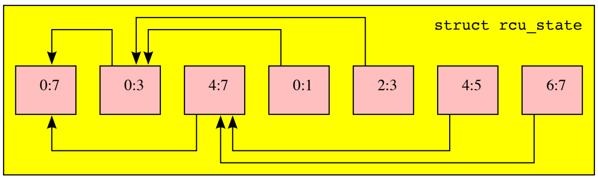
Relationship to rcu_node and rcu_data Structures

This portion of the `rcu_state` structure is declared as follows:

```
struct rcu_node node[NUM_RCU_NODES];
struct rcu_node *level[NUM_RCU_LVLS + 1];
struct rcu_data __percpu *rda;
```

Quick Quiz:
Wait a minute! You said that the `rcu_node` structures formed a tree, but they are declared as a flat array! What gives?
Answer:
The tree is laid out in the array. The first node In the array is the head, the next set of nodes in the array are children of the head node, and so on until the last set of nodes in the array are the leaves. See the following diagrams to see how this works.

The `rcu_node` tree is embedded into the `->node[]` array as shown in the following figure:



One interesting consequence of this mapping is that a breadth-first traversal of the tree is implemented as a simple linear scan of the array, which is in fact what the `rcu_for_each_node_breadth_first()` macro does. This macro is used at the beginning and ends of grace periods.

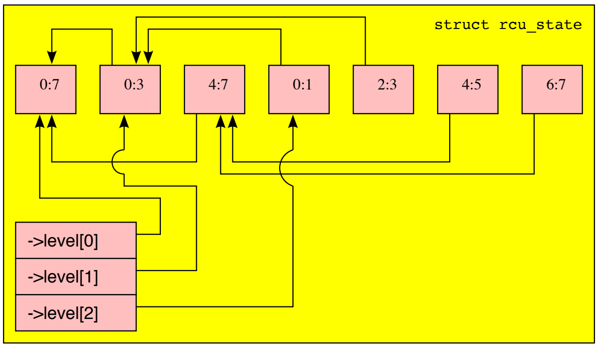
Each entry of the `->level` array references the first `rcu_node` structure on the corresponding level of the tree, for example, as shown below:

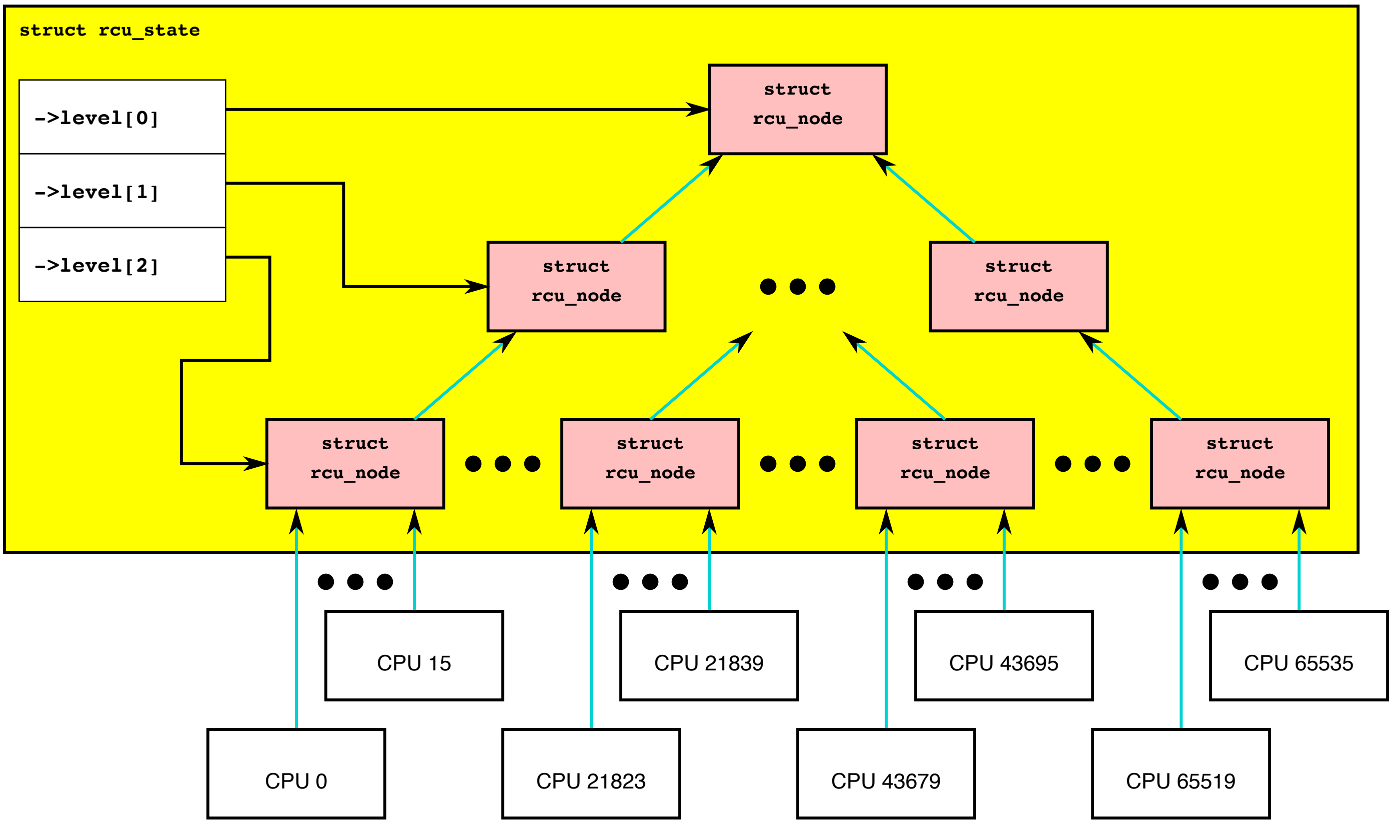
The zeroth element of the array references the root `rcu_node` structure, the first element references the first child of the root `rcu_node`, and finally the second element references the first leaf `rcu_node` structure.

For whatever it is worth, if you draw the tree to be tree-shaped rather than array-shaped, it is easy to draw a planar representation:

Finally, the `->rda` field references a per-CPU pointer to the corresponding CPU's `rcu_data` structure.

All of these fields are constant once initialization is complete, and therefore need no protection.


struct rcu_state
0:7
0:3
4:7
0:1
2:3
4:5
6:7
->level[0]
->level[1]
->level[2]


struct rcu_state
->level[0]
->level[1]
->level[2]
struct rcu_node
struct rcu_node
struct rcu_node
struct rcu_node
struct rcu_node
struct rcu_node
struct rcu_node
CPU 15
CPU 21839
CPU 43695
CPU 65535
CPU 0
CPU 21823
CPU 43679
CPU 65519

Grace-Period Tracking

This portion of the `rcu_state` structure is declared as follows:

```
unsigned long gp_seq;
```

RCU grace periods are numbered, and the `->gp_seq` field contains the current grace-period sequence number. The bottom two bits are the state of the current grace period, which can be zero for not yet started or one for in progress. In other words, if the bottom two bits of `->gp_seq` are zero, then RCU is idle. Any other value in the bottom two bits indicates that something is broken. This field is protected by the root `rcu_node` structure's `->lock` field.

There are `->gp_seq` fields in the `rcu_node` and `rcu_data` structures as well. The fields in the `rcu_state` structure represent the most current value, and those of the other structures are compared in order to detect the beginnings and ends of grace periods in a distributed fashion. The values flow from `rcu_state` to `rcu_node` (down the tree from the root to the leaves) to `rcu_data`.

Miscellaneous

This portion of the `rcu_state` structure is declared as follows:

```
unsigned long gp_max;
char abbr;
char *name;
```

The `->gp_max` field tracks the duration of the longest grace period in jiffies. It is protected by the root `rcu_node`'s `->lock`.

The `->name` and `->abbr` fields distinguish between preemptible RCU ("rcu_preempt" and "p") and non-preemptible RCU ("rcu_sched" and "s"). These fields are used for diagnostic and tracing purposes.

18.2.2 The `rcu_node` Structure

The `rcu_node` structures form the combining tree that propagates quiescent-state information from the leaves to the root and also that propagates grace-period information from the root down to the leaves. They provides local copies of the grace-period state in order to allow this information to be accessed in a synchronized manner without suffering the scalability limitations that would otherwise be imposed by global locking. In `CONFIG_PREEMPT_RCU` kernels, they manage the lists of tasks that have blocked while in their current RCU read-side critical section. In `CONFIG_PREEMPT_RCU` with `CONFIG_RCU_BOOST`, they manage the per-`rcu_node` priority-boosting kernel threads (kthreads) and state. Finally, they record CPU-hotplug state in order to determine which CPUs should be ignored during a given grace period.

The `rcu_node` structure's fields are discussed, singly and in groups, in the following sections.

Connection to Combining Tree

This portion of the rcu_node structure is declared as follows:

```
struct rcu_node *parent;
u8 level;
u8 grpnum;
unsigned long grpmask;
int grplo;
int grphi;
```

The ->parent pointer references the rcu_node one level up in the tree, and is NULL for the root rcu_node. The RCU implementation makes heavy use of this field to push quiescent states up the tree. The ->level field gives the level in the tree, with the root being at level zero, its children at level one, and so on. The ->grpnum field gives this node's position within the children of its parent, so this number can range between 0 and 31 on 32-bit systems and between 0 and 63 on 64-bit systems. The ->level and ->grpnum fields are used only during initialization and for tracing. The ->grpmask field is the bitmask counterpart of ->grpnum, and therefore always has exactly one bit set. This mask is used to clear the bit corresponding to this rcu_node structure in its parent's bitmasks, which are described later. Finally, the ->grplo and ->grphi fields contain the lowest and highest numbered CPU served by this rcu_node structure, respectively.

All of these fields are constant, and thus do not require any synchronization.

Synchronization

This field of the rcu_node structure is declared as follows:

```
raw_spinlock_t lock;
```

This field is used to protect the remaining fields in this structure, unless otherwise stated. That said, all of the fields in this structure can be accessed without locking for tracing purposes. Yes, this can result in confusing traces, but better some tracing confusion than to be heisenbugged out of existence.

Grace-Period Tracking

This portion of the rcu_node structure is declared as follows:

```
unsigned long gp_seq;
unsigned long gp_seq_needed;
```

The rcu_node structures' ->gp_seq fields are the counterparts of the field of the same name in the rcu_state structure. They each may lag up to one step behind their rcu_state counterpart. If the bottom two bits of a given rcu_node structure's ->gp_seq field is zero, then this rcu_node structure believes that RCU is idle.

The >gp_seq field of each rcu_node structure is updated at the beginning and the end of each grace period.

The ->gp_seq_needed fields record the furthest-in-the-future grace period request seen by the corresponding rcu_node structure. The request is considered fulfilled when the value of the

->gp_seq field equals or exceeds that of the ->gp_seq_needed field.

Quick Quiz:
Suppose that this rcu_node structure doesn't see a request for a very long time. Won't wrapping of the ->gp_seq field cause problems?

Answer:
No, because if the ->gp_seq_needed field lags behind the ->gp_seq field, the ->gp_seq_needed field will be updated at the end of the grace period. Modulo-arithmetic comparisons therefore will always get the correct answer, even with wrapping.

Quiescent-State Tracking

These fields manage the propagation of quiescent states up the combining tree.

This portion of the rcu_node structure has fields as follows:

```
unsigned long qsmask;
unsigned long expmask;
unsigned long qsmaskinit;
unsigned long expmaskinit;
```

The ->qsmask field tracks which of this rcu_node structure's children still need to report quiescent states for the current normal grace period. Such children will have a value of 1 in their corresponding bit. Note that the leaf rcu_node structures should be thought of as having rcu_data structures as their children. Similarly, the ->expmask field tracks which of this rcu_node structure's children still need to report quiescent states for the current expedited grace period. An expedited grace period has the same conceptual properties as a normal grace period, but the expedited implementation accepts extreme CPU overhead to obtain much lower grace-period latency, for example, consuming a few tens of microseconds worth of CPU time to reduce grace-period duration from milliseconds to tens of microseconds. The ->qsmaskinit field tracks which of this rcu_node structure's children cover for at least one online CPU. This mask is used to initialize ->qsmask, and ->expmaskinit is used to initialize ->expmask and the beginning of the normal and expedited grace periods, respectively.

Quick Quiz:

Why are these bitmasks protected by locking? Come on, haven't you heard of atomic instructions???

Answer:

Lockless grace-period computation! Such a tantalizing possibility! But consider the following sequence of events:

1. CPU 0 has been in dyntick-idle mode for quite some time. When it wakes up, it notices that the current RCU grace period needs it to report in, so it sets a flag where the scheduling clock interrupt will find it.
2. Meanwhile, CPU 1 is running `force_quiescent_state()`, and notices that CPU 0 has been in dyntick idle mode, which qualifies as an extended quiescent state.
3. CPU 0's scheduling clock interrupt fires in the middle of an RCU read-side critical section, and notices that the RCU core needs something, so commences RCU softirq processing.
4. CPU 0's softirq handler executes and is just about ready to report its quiescent state up the `rcu_node` tree.
5. But CPU 1 beats it to the punch, completing the current grace period and starting a new one.
6. CPU 0 now reports its quiescent state for the wrong grace period. That grace period might now end before the RCU read-side critical section. If that happens, disaster will ensue.

So the locking is absolutely required in order to coordinate clearing of the bits with updating of the grace-period sequence number in `->gp_seq`.

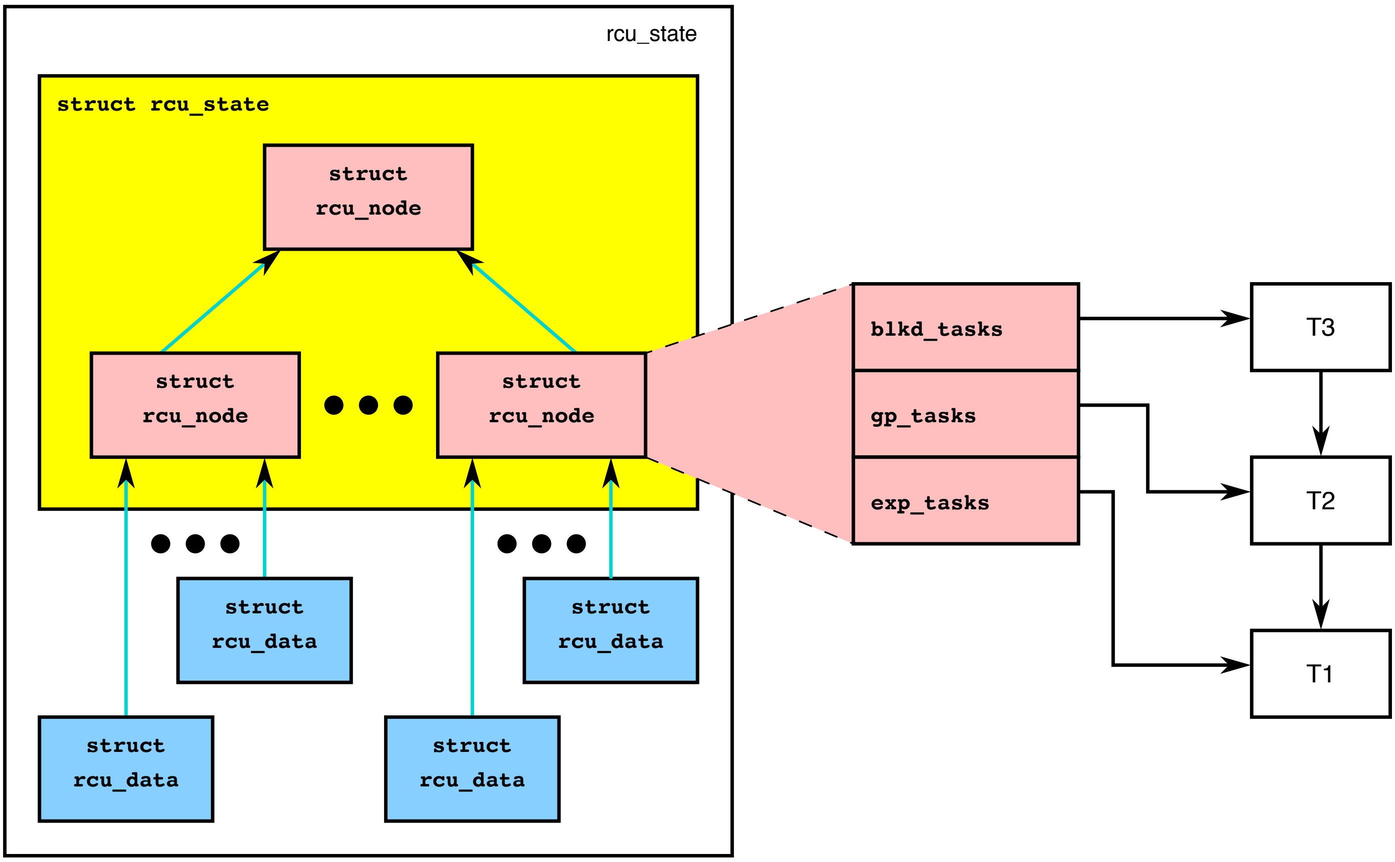
Blocked-Task Management

`PREEMPT_RCU` allows tasks to be preempted in the midst of their RCU read-side critical sections, and these tasks must be tracked explicitly. The details of exactly why and how they are tracked will be covered in a separate article on RCU read-side processing. For now, it is enough to know that the `rcu_node` structure tracks them.

```
struct list_head blkd_tasks;
struct list_head *gp_tasks;
struct list_head *exp_tasks;
bool wait_blkd_tasks;
```

The `->blkd_tasks` field is a list header for the list of blocked and preempted tasks. As tasks undergo context switches within RCU read-side critical sections, their `task_struct` structures are enqueued (via the `task_struct`'s `->rcu_node_entry` field) onto the head of the `->blkd_tasks` list for the leaf `rcu_node` structure corresponding to the CPU on which the outgoing context switch executed. As these tasks later exit their RCU read-side critical sections, they remove themselves from the list. This list is therefore in reverse time order, so that if one of the tasks is blocking the current grace period, all subsequent tasks must also be blocking that same grace period. Therefore, a single pointer into this list suffices to track all tasks blocking a given grace period. That pointer is stored in `->gp_tasks` for normal grace periods and in `->exp_tasks` for expedited grace periods. These last two fields are `NULL` if either there is no grace period in flight or if there are no blocked tasks preventing that grace period from completing. If either of these two pointers is referencing a task that removes itself from the `->blkd_tasks` list, then that task must advance the pointer to the next task on the list, or set the pointer to `NULL` if there are no subsequent tasks on the list.

For example, suppose that tasks T1, T2, and T3 are all hard-affinitied to the largest-numbered CPU in the system. Then if task T1 blocked in an RCU read-side critical section, then an expedited grace period started, then task T2 blocked in an RCU read-side critical section, then a normal grace period started, and finally task 3 blocked in an RCU read-side critical section, then the state of the last leaf `rcu_node` structure's blocked-task list would be as shown below:



Task T1 is blocking both grace periods, task T2 is blocking only the normal grace period, and task T3 is blocking neither grace period. Note that these tasks will not remove themselves from this list immediately upon resuming execution. They will instead remain on the list until they execute the outermost `rcu_read_unlock()` that ends their RCU read-side critical section.

The `->wait_blkd_tasks` field indicates whether or not the current grace period is waiting on a blocked task.

Sizing the `rcu_node` Array

The `rcu_node` array is sized via a series of C-preprocessor expressions as follows:

```
#ifdef CONFIG_RCU_FANOUT
#define RCU_FANOUT CONFIG_RCU_FANOUT
#else
# ifdef CONFIG_64BIT
# define RCU_FANOUT 64
# else
```

```
# define RCU_FANOUT 32
# endif
#endif

#ifdef CONFIG_RCU_FANOUT_LEAF
#define RCU_FANOUT_LEAF CONFIG_RCU_FANOUT_LEAF
#else
# ifdef CONFIG_64BIT
# define RCU_FANOUT_LEAF 64
# else
# define RCU_FANOUT_LEAF 32
# endif
#endif

#define RCU_FANOUT_1        (RCU_FANOUT_LEAF)
#define RCU_FANOUT_2        (RCU_FANOUT_1 * RCU_FANOUT)
#define RCU_FANOUT_3        (RCU_FANOUT_2 * RCU_FANOUT)
#define RCU_FANOUT_4        (RCU_FANOUT_3 * RCU_FANOUT)

#if NR_CPUS <= RCU_FANOUT_1
#  define RCU_NUM_LVLS        1
#  define NUM_RCU_LVL_0        1
#  define NUM_RCU_NODES        NUM_RCU_LVL_0
#  define NUM_RCU_LVL_INIT    { NUM_RCU_LVL_0 }
#  define RCU_NODE_NAME_INIT  { "rcu_node_0" }
#  define RCU_FQS_NAME_INIT   { "rcu_node_fqs_0" }
#  define RCU_EXP_NAME_INIT   { "rcu_node_exp_0" }
#elif NR_CPUS <= RCU_FANOUT_2
#  define RCU_NUM_LVLS        2
#  define NUM_RCU_LVL_0        1
#  define NUM_RCU_LVL_1        DIV_ROUND_UP(NR_CPUS, RCU_FANOUT_1)
#  define NUM_RCU_NODES        (NUM_RCU_LVL_0 + NUM_RCU_LVL_1)
#  define NUM_RCU_LVL_INIT    { NUM_RCU_LVL_0, NUM_RCU_LVL_1 }
#  define RCU_NODE_NAME_INIT  { "rcu_node_0", "rcu_node_1" }
#  define RCU_FQS_NAME_INIT   { "rcu_node_fqs_0", "rcu_node_fqs_1" }
#  define RCU_EXP_NAME_INIT   { "rcu_node_exp_0", "rcu_node_exp_1" }
#elif NR_CPUS <= RCU_FANOUT_3
#  define RCU_NUM_LVLS        3
#  define NUM_RCU_LVL_0        1
#  define NUM_RCU_LVL_1        DIV_ROUND_UP(NR_CPUS, RCU_FANOUT_2)
#  define NUM_RCU_LVL_2        DIV_ROUND_UP(NR_CPUS, RCU_FANOUT_1)
#  define NUM_RCU_NODES        (NUM_RCU_LVL_0 + NUM_RCU_LVL_1 + NUM_RCU_LVL_2)
#  define NUM_RCU_LVL_INIT    { NUM_RCU_LVL_0, NUM_RCU_LVL_1, NUM_RCU_LVL_2 }
#  define RCU_NODE_NAME_INIT  { "rcu_node_0", "rcu_node_1", "rcu_node_2" }
#  define RCU_FQS_NAME_INIT   { "rcu_node_fqs_0", "rcu_node_fqs_1", "rcu_node_fqs_2" }
#  define RCU_EXP_NAME_INIT   { "rcu_node_exp_0", "rcu_node_exp_1", "rcu_node_exp_2" }
```

```
53 #elif NR_CPUS <= RCU_FANOUT_4
54 #  define RCU_NUM_LVLS        4
55 #  define NUM_RCU_LVL_0        1
56 #  define NUM_RCU_LVL_1        DIV_ROUND_UP(NR_CPUS, RCU_FANOUT_3)
57 #  define NUM_RCU_LVL_2        DIV_ROUND_UP(NR_CPUS, RCU_FANOUT_2)
58 #  define NUM_RCU_LVL_3        DIV_ROUND_UP(NR_CPUS, RCU_FANOUT_1)
59 #  define NUM_RCU_NODES        (NUM_RCU_LVL_0 + NUM_RCU_LVL_1 + NUM_RCU_LVL_
↪2 + NUM_RCU_LVL_3)
60 #  define NUM_RCU_LVL_INIT    { NUM_RCU_LVL_0, NUM_RCU_LVL_1, NUM_RCU_LVL_2,
↪ NUM_RCU_LVL_3 }
61 #  define RCU_NODE_NAME_INIT  { "rcu_node_0", "rcu_node_1", "rcu_node_2",
↪"rcu_node_3" }
62 #  define RCU_FQS_NAME_INIT   { "rcu_node_fqs_0", "rcu_node_fqs_1", "rcu_
↪node_fqs_2", "rcu_node_fqs_3" }
63 #  define RCU_EXP_NAME_INIT   { "rcu_node_exp_0", "rcu_node_exp_1", "rcu_
↪node_exp_2", "rcu_node_exp_3" }
64 #else
65 # error "CONFIG_RCU_FANOUT insufficient for NR_CPUS"
66 #endif
```

The maximum number of levels in the `rcu_node` structure is currently limited to four, as specified by lines 21-24 and the structure of the subsequent "if" statement. For 32-bit systems, this allows 16*32*32*32=524,288 CPUs, which should be sufficient for the next few years at least. For 64-bit systems, 16*64*64*64=4,194,304 CPUs is allowed, which should see us through the next decade or so. This four-level tree also allows kernels built with `CONFIG_RCU_FANOUT=8` to support up to 4096 CPUs, which might be useful in very large systems having eight CPUs per socket (but please note that no one has yet shown any measurable performance degradation due to misaligned socket and `rcu_node` boundaries). In addition, building kernels with a full four levels of `rcu_node` tree permits better testing of RCU's combining-tree code.

The `RCU_FANOUT` symbol controls how many children are permitted at each non-leaf level of the `rcu_node` tree. If the `CONFIG_RCU_FANOUT` Kconfig option is not specified, it is set based on the word size of the system, which is also the Kconfig default.

The `RCU_FANOUT_LEAF` symbol controls how many CPUs are handled by each leaf `rcu_node` structure. Experience has shown that allowing a given leaf `rcu_node` structure to handle 64 CPUs, as permitted by the number of bits in the `->qsmask` field on a 64-bit system, results in excessive contention for the leaf `rcu_node` structures' `->lock` fields. The number of CPUs per leaf `rcu_node` structure is therefore limited to 16 given the default value of `CONFIG_RCU_FANOUT_LEAF`. If `CONFIG_RCU_FANOUT_LEAF` is unspecified, the value selected is based on the word size of the system, just as for `CONFIG_RCU_FANOUT`. Lines 11-19 perform this computation.

Lines 21-24 compute the maximum number of CPUs supported by a single-level (which contains a single `rcu_node` structure), two-level, three-level, and four-level `rcu_node` tree, respectively, given the fanout specified by `RCU_FANOUT` and `RCU_FANOUT_LEAF`. These numbers of CPUs are retained in the `RCU_FANOUT_1`, `RCU_FANOUT_2`, `RCU_FANOUT_3`, and `RCU_FANOUT_4` C-preprocessor variables, respectively.

These variables are used to control the C-preprocessor `#if` statement spanning lines 26-66 that computes the number of `rcu_node` structures required for each level of the tree, as well as the number of levels required. The number of levels is placed in the `NUM_RCU_LVLS` C-preprocessor variable by lines 27, 35, 44, and 54. The number of `rcu_node` structures for the topmost level of the tree is always exactly one, and this value is unconditionally placed into NUM_RCU_LVL_0 by lines 28, 36, 45, and 55. The rest of the levels (if any) of the rcu_node tree are computed by dividing the maximum number of CPUs by the fanout supported by the number of levels from the current level down, rounding up. This computation is performed by lines 37, 46-47, and 56-58. Lines 31-33, 40-42, 50-52, and 62-63 create initializers for lockdep lock-class names. Finally, lines 64-66 produce an error if the maximum number of CPUs is too large for the specified fanout.

18.2.3 The rcu_segcblist Structure

The rcu_segcblist structure maintains a segmented list of callbacks as follows:

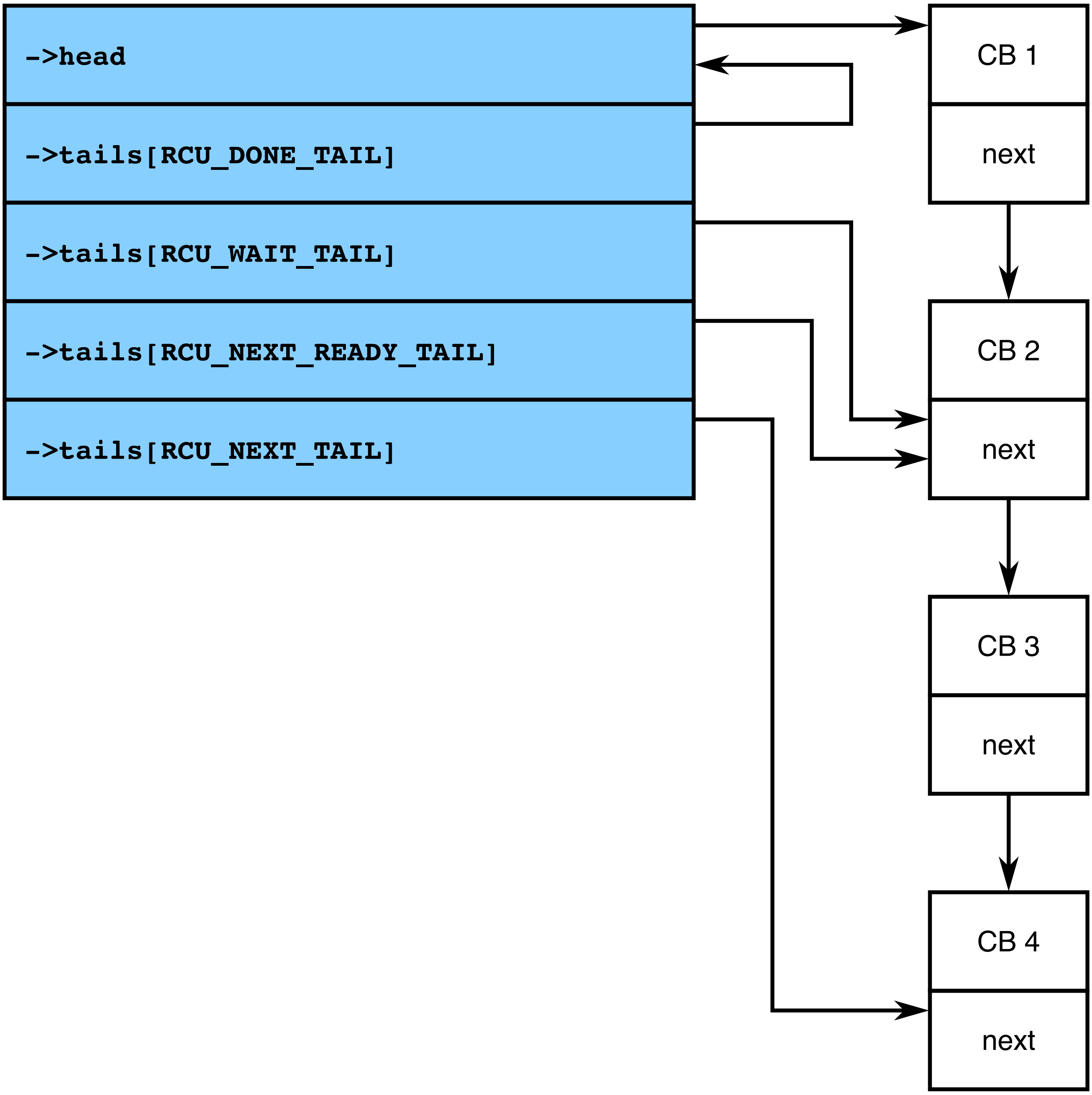
```
#define RCU_DONE_TAIL        0
#define RCU_WAIT_TAIL        1
#define RCU_NEXT_READY_TAIL  2
#define RCU_NEXT_TAIL        3
#define RCU_CBLIST_NSEGS     4

struct rcu_segcblist {
  struct rcu_head *head;
  struct rcu_head **tails[RCU_CBLIST_NSEGS];
  unsigned long gp_seq[RCU_CBLIST_NSEGS];
  long len;
  long len_lazy;
};
```

The segments are as follows:

1. RCU_DONE_TAIL: Callbacks whose grace periods have elapsed. These callbacks are ready to be invoked.
2. RCU_WAIT_TAIL: Callbacks that are waiting for the current grace period. Note that different CPUs can have different ideas about which grace period is current, hence the ->gp_seq field.
3. RCU_NEXT_READY_TAIL: Callbacks waiting for the next grace period to start.
4. RCU_NEXT_TAIL: Callbacks that have not yet been associated with a grace period.

The ->head pointer references the first callback or is NULL if the list contains no callbacks (which is *not* the same as being empty). Each element of the ->tails[] array references the ->next pointer of the last callback in the corresponding segment of the list, or the list's ->head pointer if that segment and all previous segments are empty. If the corresponding segment is empty but some previous segment is not empty, then the array element is identical to its predecessor. Older callbacks are closer to the head of the list, and new callbacks are added at the tail. This relationship between the ->head pointer, the ->tails[] array, and the callbacks is shown in this diagram:

In this figure, the ->head pointer references the first RCU callback in the list. The ->tails[RCU_DONE_TAIL] array element references the ->head pointer itself, indicating that none of the callbacks is ready to invoke. The ->tails[RCU_WAIT_TAIL] array element references callback CB 2's ->next pointer, which indicates that CB 1 and CB 2 are both waiting on the current grace period, give or take possible disagreements about exactly which grace period is the current one. The ->tails[RCU_NEXT_READY_TAIL] array element references the


->head
->tails[RCU_DONE_TAIL]
->tails[RCU_WAIT_TAIL]
->tails[RCU_NEXT_READY_TAIL]
->tails[RCU_NEXT_TAIL]
CB 1
next
CB 2
next
CB 3
next
CB 4
next

same RCU callback that ->tails[RCU_WAIT_TAIL] does, which indicates that there are no callbacks waiting on the next RCU grace period. The ->tails[RCU_NEXT_TAIL] array element references CB 4's ->next pointer, indicating that all the remaining RCU callbacks have not yet been assigned to an RCU grace period. Note that the ->tails[RCU_NEXT_TAIL] array element always references the last RCU callback's ->next pointer unless the callback list is empty, in which case it references the ->head pointer.

There is one additional important special case for the ->tails[RCU_NEXT_TAIL] array element: It can be NULL when this list is *disabled*. Lists are disabled when the corresponding CPU is offline or when the corresponding CPU's callbacks are offloaded to a kthread, both of which are described elsewhere.

CPUs advance their callbacks from the RCU_NEXT_TAIL to the RCU_NEXT_READY_TAIL to the RCU_WAIT_TAIL to the RCU_DONE_TAIL list segments as grace periods advance.

The ->gp_seq[] array records grace-period numbers corresponding to the list segments. This is what allows different CPUs to have different ideas as to which is the current grace period while still avoiding premature invocation of their callbacks. In particular, this allows CPUs that go idle for extended periods to determine which of their callbacks are ready to be invoked after reawakening.

The ->len counter contains the number of callbacks in ->head, and the ->len_lazy contains the number of those callbacks that are known to only free memory, and whose invocation can therefore be safely deferred.

Important: It is the ->len field that determines whether or not there are callbacks associated with this rcu_segcblist structure, *not* the ->head pointer. The reason for this is that all the ready-to-invoke callbacks (that is, those in the RCU_DONE_TAIL segment) are extracted all at once at callback-invocation time (rcu_do_batch), due to which ->head may be set to NULL if there are no not-done callbacks remaining in the rcu_segcblist. If callback invocation must be postponed, for example, because a high-priority process just woke up on this CPU, then the remaining callbacks are placed back on the RCU_DONE_TAIL segment and ->head once again points to the start of the segment. In short, the head field can briefly be NULL even though the CPU has callbacks present the entire time. Therefore, it is not appropriate to test the ->head pointer for NULL.

In contrast, the ->len and ->len_lazy counts are adjusted only after the corresponding callbacks have been invoked. This means that the ->len count is zero only if the rcu_segcblist structure really is devoid of callbacks. Of course, off-CPU sampling of the ->len count requires careful use of appropriate synchronization, for example, memory barriers. This synchronization can be a bit subtle, particularly in the case of rcu_barrier().

18.2.4 The rcu_data Structure

The rcu_data maintains the per-CPU state for the RCU subsystem. The fields in this structure may be accessed only from the corresponding CPU (and from tracing) unless otherwise stated. This structure is the focus of quiescent-state detection and RCU callback queuing. It also tracks its relationship to the corresponding leaf rcu_node structure to allow more-efficient propagation of quiescent states up the rcu_node combining tree. Like the rcu_node structure, it provides a local copy of the grace-period information to allow for-free synchronized access to this information from the corresponding CPU. Finally, this structure records past dyntick-idle state for the corresponding CPU and also tracks statistics.

The `rcu_data` structure's fields are discussed, singly and in groups, in the following sections.

Connection to Other Data Structures

This portion of the `rcu_data` structure is declared as follows:

```
int cpu;
struct rcu_node *mynode;
unsigned long grpmask;
bool beenonline;
```

The `->cpu` field contains the number of the corresponding CPU and the `->mynode` field references the corresponding `rcu_node` structure. The `->mynode` is used to propagate quiescent states up the combining tree. These two fields are constant and therefore do not require synchronization.

The `->grpmask` field indicates the bit in the `->mynode->qsmask` corresponding to this `rcu_data` structure, and is also used when propagating quiescent states. The `->beenonline` flag is set whenever the corresponding CPU comes online, which means that the debugfs tracing need not dump out any `rcu_data` structure for which this flag is not set.

Quiescent-State and Grace-Period Tracking

This portion of the `rcu_data` structure is declared as follows:

```
unsigned long gp_seq;
unsigned long gp_seq_needed;
bool cpu_no_qs;
bool core_needs_qs;
bool gpwrap;
```

The `->gp_seq` field is the counterpart of the field of the same name in the `rcu_state` and `rcu_node` structures. The `->gp_seq_needed` field is the counterpart of the field of the same name in the rcu_node structure. They may each lag up to one behind their `rcu_node` counterparts, but in `CONFIG_NO_HZ_IDLE` and `CONFIG_NO_HZ_FULL` kernels can lag arbitrarily far behind for CPUs in dyntick-idle mode (but these counters will catch up upon exit from dyntick-idle mode). If the lower two bits of a given `rcu_data` structure's `->gp_seq` are zero, then this `rcu_data` structure believes that RCU is idle.

Quick Quiz:
All this replication of the grace period numbers can only cause massive confusion. Why not just keep a global sequence number and be done with it???
Answer:
Because if there was only a single global sequence numbers, there would need to be a single global lock to allow safely accessing and updating it. And if we are not going to have a single global lock, we need to carefully manage the numbers on a per-node basis. Recall from the answer to a previous Quick Quiz that the consequences of applying a previously sampled quiescent state to the wrong grace period are quite severe.

The `->cpu_no_qs` flag indicates that the CPU has not yet passed through a quiescent state, while the ->core_needs_qs flag indicates that the RCU core needs a quiescent state from the corresponding CPU. The ->gpwrap field indicates that the corresponding CPU has remained idle for so long that the gp_seq counter is in danger of overflow, which will cause the CPU to disregard the values of its counters on its next exit from idle.

RCU Callback Handling

In the absence of CPU-hotplug events, RCU callbacks are invoked by the same CPU that registered them. This is strictly a cache-locality optimization: callbacks can and do get invoked on CPUs other than the one that registered them. After all, if the CPU that registered a given callback has gone offline before the callback can be invoked, there really is no other choice.

This portion of the rcu_data structure is declared as follows:

```
struct rcu_segcblist cblist;
long qlen_last_fqs_check;
unsigned long n_cbs_invoked;
unsigned long n_nocbs_invoked;
unsigned long n_cbs_orphaned;
unsigned long n_cbs_adopted;
unsigned long n_force_qs_snap;
long blimit;
```

The ->cblist structure is the segmented callback list described earlier. The CPU advances the callbacks in its rcu_data structure whenever it notices that another RCU grace period has completed. The CPU detects the completion of an RCU grace period by noticing that the value of its rcu_data structure's ->gp_seq field differs from that of its leaf rcu_node structure. Recall that each rcu_node structure's ->gp_seq field is updated at the beginnings and ends of each grace period.

The ->qlen_last_fqs_check and ->n_force_qs_snap coordinate the forcing of quiescent states from call_rcu() and friends when callback lists grow excessively long.

The ->n_cbs_invoked, ->n_cbs_orphaned, and ->n_cbs_adopted fields count the number of callbacks invoked, sent to other CPUs when this CPU goes offline, and received from other CPUs when those other CPUs go offline. The ->n_nocbs_invoked is used when the CPU's callbacks are offloaded to a kthread.

Finally, the ->blimit counter is the maximum number of RCU callbacks that may be invoked at a given time.

Dyntick-Idle Handling

This portion of the rcu_data structure is declared as follows:

```
int dynticks_snap;
unsigned long dynticks_fqs;
```

The ->dynticks_snap field is used to take a snapshot of the corresponding CPU's dyntick-idle state when forcing quiescent states, and is therefore accessed from other CPUs. Finally, the ->dynticks_fqs field is used to count the number of times this CPU is determined to be in dyntick-idle state, and is used for tracing and debugging purposes.

This portion of the rcu_data structure is declared as follows:

```
long dynticks_nesting;
long dynticks_nmi_nesting;
atomic_t dynticks;
bool rcu_need_heavy_qs;
bool rcu_urgent_qs;
```

These fields in the rcu_data structure maintain the per-CPU dyntick-idle state for the corresponding CPU. The fields may be accessed only from the corresponding CPU (and from tracing) unless otherwise stated.

The `->dynticks_nesting` field counts the nesting depth of process execution, so that in normal circumstances this counter has value zero or one. NMIs, irqs, and tracers are counted by the `->dynticks_nmi_nesting` field. Because NMIs cannot be masked, changes to this variable have to be undertaken carefully using an algorithm provided by Andy Lutomirski. The initial transition from idle adds one, and nested transitions add two, so that a nesting level of five is represented by a `->dynticks_nmi_nesting` value of nine. This counter can therefore be thought of as counting the number of reasons why this CPU cannot be permitted to enter dyntick-idle mode, aside from process-level transitions.

However, it turns out that when running in non-idle kernel context, the Linux kernel is fully capable of entering interrupt handlers that never exit and perhaps also vice versa. Therefore, whenever the `->dynticks_nesting` field is incremented up from zero, the `->dynticks_nmi_nesting` field is set to a large positive number, and whenever the `->dynticks_nesting` field is decremented down to zero, the `->dynticks_nmi_nesting` field is set to zero. Assuming that the number of misnested interrupts is not sufficient to overflow the counter, this approach corrects the `->dynticks_nmi_nesting` field every time the corresponding CPU enters the idle loop from process context.

The `->dynticks` field counts the corresponding CPU's transitions to and from either dyntick-idle or user mode, so that this counter has an even value when the CPU is in dyntick-idle mode or user mode and an odd value otherwise. The transitions to/from user mode need to be counted for user mode adaptive-ticks support (see Documentation/timers/no_hz.rst).

The `->rcu_need_heavy_qs` field is used to record the fact that the RCU core code would really like to see a quiescent state from the corresponding CPU, so much so that it is willing to call for heavy-weight dyntick-counter operations. This flag is checked by RCU's context-switch and `cond_resched()` code, which provide a momentary idle sojourn in response.

Finally, the `->rcu_urgent_qs` field is used to record the fact that the RCU core code would really like to see a quiescent state from the corresponding CPU, with the various other fields indicating just how badly RCU wants this quiescent state. This flag is checked by RCU's context-switch path (`rcu_note_context_switch`) and the cond_resched code.

Quick Quiz:

Why not simply combine the `->dynticks_nesting` and `->dynticks_nmi_nesting` counters into a single counter that just counts the number of reasons that the corresponding CPU is non-idle?

Answer:

Because this would fail in the presence of interrupts whose handlers never return and of handlers that manage to return from a made-up interrupt.

Additional fields are present for some special-purpose builds, and are discussed separately.

18.2.5 The `rcu_head` Structure

Each `rcu_head` structure represents an RCU callback. These structures are normally embedded within RCU-protected data structures whose algorithms use asynchronous grace periods. In contrast, when using algorithms that block waiting for RCU grace periods, RCU users need not provide `rcu_head` structures.

The `rcu_head` structure has fields as follows:

```
struct rcu_head *next;
void (*func)(struct rcu_head *head);
```

The `->next` field is used to link the `rcu_head` structures together in the lists within the `rcu_data` structures. The `->func` field is a pointer to the function to be called when the callback is ready to be invoked, and this function is passed a pointer to the `rcu_head` structure. However, `kfree_rcu()` uses the `->func` field to record the offset of the `rcu_head` structure within the enclosing RCU-protected data structure.

Both of these fields are used internally by RCU. From the viewpoint of RCU users, this structure is an opaque "cookie".

Quick Quiz:
Given that the callback function `->func` is passed a pointer to the `rcu_head` structure, how is that function supposed to find the beginning of the enclosing RCU-protected data structure?

Answer:
In actual practice, there is a separate callback function per type of RCU-protected data structure. The callback function can therefore use the `container_of()` macro in the Linux kernel (or other pointer-manipulation facilities in other software environments) to find the beginning of the enclosing structure.

18.2.6 RCU-Specific Fields in the `task_struct` Structure

The `CONFIG_PREEMPT_RCU` implementation uses some additional fields in the `task_struct` structure:

```
#ifdef CONFIG_PREEMPT_RCU
  int rcu_read_lock_nesting;
  union rcu_special rcu_read_unlock_special;
  struct list_head rcu_node_entry;
  struct rcu_node *rcu_blocked_node;
#endif /* #ifdef CONFIG_PREEMPT_RCU */
#ifdef CONFIG_TASKS_RCU
  unsigned long rcu_tasks_nvcsw;
  bool rcu_tasks_holdout;
  struct list_head rcu_tasks_holdout_list;
  int rcu_tasks_idle_cpu;
#endif /* #ifdef CONFIG_TASKS_RCU */
```

The ->rcu_read_lock_nesting field records the nesting level for RCU read-side critical sections, and the ->rcu_read_unlock_special field is a bitmask that records special conditions that require rcu_read_unlock() to do additional work. The ->rcu_node_entry field is used to form lists of tasks that have blocked within preemptible-RCU read-side critical sections and the ->rcu_blocked_node field references the rcu_node structure whose list this task is a member of, or NULL if it is not blocked within a preemptible-RCU read-side critical section.

The ->rcu_tasks_nvcsw field tracks the number of voluntary context switches that this task had undergone at the beginning of the current tasks-RCU grace period, ->rcu_tasks_holdout is set if the current tasks-RCU grace period is waiting on this task, ->rcu_tasks_holdout_list is a list element enqueuing this task on the holdout list, and ->rcu_tasks_idle_cpu tracks which CPU this idle task is running, but only if the task is currently running, that is, if the CPU is currently idle.

18.2.7 Accessor Functions

The following listing shows the rcu_get_root(), rcu_for_each_node_breadth_first and rcu_for_each_leaf_node() function and macros:

```
static struct rcu_node *rcu_get_root(struct rcu_state *rsp)
{
  return &rsp->node[0];
}

#define rcu_for_each_node_breadth_first(rsp, rnp) \
  for ((rnp) = &(rsp)->node[0]; \
       (rnp) < &(rsp)->node[NUM_RCU_NODES]; (rnp)++)

#define rcu_for_each_leaf_node(rsp, rnp) \
  for ((rnp) = (rsp)->level[NUM_RCU_LVLS - 1]; \
       (rnp) < &(rsp)->node[NUM_RCU_NODES]; (rnp)++)
```

The rcu_get_root() simply returns a pointer to the first element of the specified rcu_state structure's ->node[] array, which is the root rcu_node structure.

As noted earlier, the rcu_for_each_node_breadth_first() macro takes advantage of the layout of the rcu_node structures in the rcu_state structure's ->node[] array, performing a breadth-first traversal by simply traversing the array in order. Similarly, the rcu_for_each_leaf_node() macro traverses only the last part of the array, thus traversing only the leaf rcu_node structures.

Quick Quiz:
What does rcu_for_each_leaf_node() do if the rcu_node tree contains only a single node?
Answer:
In the single-node case, rcu_for_each_leaf_node() traverses the single node.

18.2.8 Summary

So the state of RCU is represented by an `rcu_state` structure, which contains a combining tree of `rcu_node` and `rcu_data` structures. Finally, in `CONFIG_NO_HZ_IDLE` kernels, each CPU's dyntick-idle state is tracked by dynticks-related fields in the `rcu_data` structure. If you made it this far, you are well prepared to read the code walkthroughs in the other articles in this series.

18.2.9 Acknowledgments

I owe thanks to Cyrill Gorcunov, Mathieu Desnoyers, Dhaval Giani, Paul Turner, Abhishek Srivastava, Matt Kowalczyk, and Serge Hallyn for helping me get this document into a more human-readable state.

18.2.10 Legal Statement