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.

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 big multiprocessor machines (for example, if the task is updating information relating to itself that other tasks can read, there by definition can be no bottleneck).

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. Disabling of preemption can serve as rcu_read_lock_sched(), but is less readable.

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. Of course, fields that the readers refrain from accessing can be guarded by some other lock acquired only by updaters, if desired.

      This works quite well, also.

   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 perations performed under a lock will -not- appear to be atomic to RCU readers, nor will sequences of multiple atomic primitives.

      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 barriers (smp_wmb(), smp_rmb(), smp_mb()) through the code, making it difficult to understand and to test.

      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. If you don't believe me, see:

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.

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 lockdep.txt 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 call_rcu(), or a related primitive such as call_rcu_bh() or call_rcu_sched(), is used, the callback function must be written to be called from softirq context. In particular, it cannot block.

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

   The expedited forms of these primitives have the same semantics as the non-expedited forms, but expediting is both expensive and unfriendly to real-time workloads. 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.

7. If the updater uses call_rcu() or synchronize_rcu(), then the corresponding readers must use rcu_read_lock() and rcu_read_unlock(). If the updater uses call_rcu_bh() or synchronize_rcu_bh(), then the corresponding readers must use rcu_read_lock_bh() and rcu_read_unlock_bh(). If the updater uses call_rcu_sched() or synchronize_sched(), then the corresponding readers must disable preemption, possibly by calling rcu_read_lock_sched() and rcu_read_unlock_sched(). If the updater uses synchronize_srcu(), the the corresponding readers must use srcu_read_lock() and srcu_read_unlock(), and with the same srcu_struct. The rules for the expedited primitives are the same as for their non-expedited counterparts. Mixing things up will result in confusion and broken kernels.

   One exception to this rule: rcu_read_lock() and rcu_read_unlock() may be substituted for rcu_read_lock_bh() and rcu_read_unlock_bh() in cases where local bottom halves are already known to be disabled, for example, in irq or softirq context. Commenting such cases is a must, of course! And the jury is still out on whether the increased speed is worth it.

8. Although synchronize_rcu() is slower than is call_rcu(), it usually results in simpler code. So, unless update performance is critically important or the updaters cannot block, synchronize_rcu() should be used in preference to call_rcu().

   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() 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.

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. The dcache subsystem takes this approach -- updates are guarded by 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. Use call_rcu_bh() rather than call_rcu(), in order to take advantage of call_rcu_bh()'s faster grace periods.

e. Periodically invoke synchronize_rcu(), permitting a limited number of updates per grace period.

The same cautions apply to call_rcu_bh() and call_rcu_sched().

9. All RCU list-traversal primitives, which include rcu_dereference(), list_for_each_entry_rcu(), list_for_each_continue_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 lockdep.txt.

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 read your code.

11. Note that synchronize_rcu() -only- guarantees to wait until all currently executing rcu_read_lock()-protected RCU read-side critical sections complete. It does -not- necessarily guarantee that all currently running interrupts, NMIs, preempt_disable() code, or idle loops will complete. Therefore, if you do not have rcu_read_lock()-protected read-side critical sections, do -not- use synchronize_rcu().

Similarly, disabling preemption is not an acceptable substitute

for rcu_read_lock(). Code that attempts to use preemption disabling where it should be using rcu_read_lock() will break in real-time kernel builds.

If you want to wait for interrupt handlers, NMI handlers, and code under the influence of preempt_disable(), you instead need to use synchronize_irq() or synchronize_sched().

12. Any lock acquired by an RCU callback must be acquired elsewhere with softirq disabled, e.g., via spin_lock_irqsave(), spin_lock_bh(), etc. Failing to disable irq 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.

13. 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.

RCU callbacks are -usually- executed on the same CPU that executed the corresponding call_rcu(), call_rcu_bh(), or call_rcu_sched(), but are by -no- means guaranteed to be. 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.)

14. SRCU (srcu_read_lock(), srcu_read_unlock(), srcu_dereference(), synchronize_srcu(), and synchronize_srcu_expedited()) may only be invoked from process context. Unlike other forms 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 via init_srcu_struct() and cleanup_srcu_struct(). These 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(), and synchronize_srcu_expedited(). 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.

Note that, rcu_assign_pointer() relates to SRCU just as they do to other forms of RCU.

15. 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.

16. 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.