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 provides the same potential simplifications 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 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). 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. Disabling of preemption can serve as rcu_read_lock_sched(), but is less readable and prevents lockdep from detecting locking issues.

   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. 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 operations 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.

      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 rcu_dereference.txt in this directory 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 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 call_srcu() is used, the callback function will 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_srcu(), synchronize_rcu_expedited(), 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 (with the exception of synchronize_srcu_expedited()) 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. However, real-time workloads can use 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.

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 interrupts.

   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() usually results in even simpler code than does synchronize_rcu() without synchronize_rcu()'s multi-millisecond latency. So please take advantage of kfree_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() 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 synchronize_rcu(), permitting a limited number of updates per grace period.

   The same cautions apply to call_srcu() and kfree_rcu().

   Note that although these primitives do take action to avoid memory exhaustion when any given CPU has too many callbacks, a determined user could 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 lockdep.txt.

   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 read your code.

11. 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 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.

13. 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 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().

    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 register a callback using call_rcu() or call_srcu(), and pass in a function defined within a loadable module, then it in necessary to wait for all pending callbacks to be invoked after the last invocation and before unloading that module. Note that it is absolutely *not* sufficient to wait for a grace period! The current (say) synchronize_rcu() implementation is *not* guaranteed to wait for callbacks registered on other CPUs. 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()

    However, these barrier functions are absolutely *not* guaranteed to wait for a grace period. In fact, if there are no call_rcu() callbacks waiting anywhere in the system, rcu_barrier() is within its rights to return immediately.

So if you need to wait for both an RCU grace period and for all pre-existing call_rcu() callbacks, you will need to execute both rcu_barrier() and synchronize_rcu(), if necessary, using something like workqueues to to execute them concurrently.

See rcubarrier.txt for more information.