

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.

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.

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 contexts, 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?

ref: [Answers to Quick Quiz <answer_quick_quiz_up>](#)

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\linux-master\Documentation\RCU\linux-master) (Documentation) (RCU) UP.rst, line 54); [backlink](#)

Unknown interpreted text role "ref".

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.

In some cases, it would be possible to restructure the 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?

ref: [Answers to Quick Quiz <answer_quick_quiz_up>](#)

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\linux-master\Documentation\RCU\linux-master) (Documentation) (RCU) UP.rst, line 86); [backlink](#)

Unknown interpreted text role "ref".

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 preemptable 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 preemptable 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.