A Tour Through RCU's Requirements

Author: Paul E. McKenney

The initial version of this document appeared in the LWN articles here, here, and here.

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:

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.

Fundamental Requirements

RCU's fundamental requirements are the closest thing RCU has to hard mathematical requirements. These are:

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_PREEMPT=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:

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:

```
 1 #define STATE_NORMAL        0
 2 #define STATE_WANT_RECOVERY 1
 3 #define STATE_RECOVERING    2
 4 #define STATE_WANT_NORMAL   3
 5
 6 int state = STATE_NORMAL;
 7
 8 void do_something_dlm(void)
 9 {
10   int state_snap;
11
12   rcu_read_lock();
13   state_snap = READ_ONCE(state);
14   if (state_snap == STATE_NORMAL)
```

```
15     do_something();
16   else
17     do_something_carefully();
18   rcu_read_unlock();
19 }
20
21 void start_recovery(void)
22 {
23   WRITE_ONCE(state, STATE_WANT_RECOVERY);
24   synchronize_rcu();
25   WRITE_ONCE(state, STATE_RECOVERING);
26   recovery();
27   WRITE_ONCE(state, STATE_WANT_NORMAL);
28   synchronize_rcu();
29   WRITE_ONCE(state, STATE_NORMAL);
30 }
```

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:

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.

```
 1 bool add_gp_buggy(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   gp = p; /* ORDERING BUG */
14   spin_unlock(&gp_lock);
15   return true;
16 }
```

The problem is that both the compiler and weakly ordered CPUs are within their rights to reorder this code as follows:

```
 1 bool add_gp_buggy_optimized(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   gp = p; /* ORDERING BUG */
12   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.

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:

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:

```
1 bool do_something_gp_buggy(void)
2 {
3   rcu_read_lock();
```

```
 4   p = gp;  /* OPTIMIZATIONS GALORE!!! */
 5   if (p) {
 6     do_something(p->a, p->b);
 7     rcu_read_unlock();
 8     return true;
 9   }
10   rcu_read_unlock();
11   return false;
12 }
```

However, this temptation must be resisted because there are a surprisingly large number of ways that the compiler (to say nothing of DEC Alpha CPUs) 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:

```
 1 bool do_something_gp_buggy_optimized(void)
 2 {
 3   rcu_read_lock();
 4   if (gp) { /* OPTIMIZATIONS GALORE!!! */
 5     do_something(gp->a, gp->b);
 6     rcu_read_unlock();
 7     return true;
 8   }
 9   rcu_read_unlock();
10   return false;
11 }
```

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`:

```
 1 bool do_something_gp(void)
 2 {
 3   rcu_read_lock();
 4   p = rcu_dereference(gp);
 5   if (p) {
 6     do_something(p->a, p->b);
 7     rcu_read_unlock();
 8     return true;
 9   }
10   rcu_read_unlock();
11   return false;
12 }
```

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.

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:

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 `smp_read_barrier_depends()` 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()`!

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:

2016 LinuxCon EU

Quick Quiz:
The first and second guarantees require unbelievably strict ordering! Are all these memory barriers *really* required?
Answer:

1.
2.
3.
4.
5.
6.
7.
8.

1.
2.
3.
4.
5.
6.
7.
8.

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:

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.

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.

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:

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.

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.

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:

```
 1 void thread0(void)
 2 {
 3   rcu_read_lock();
 4   WRITE_ONCE(x, 1);
 5   rcu_read_unlock();
 6   rcu_read_lock();
 7   WRITE_ONCE(y, 1);
 8   rcu_read_unlock();
 9 }
10
11 void thread1(void)
12 {
13   rcu_read_lock();
14   r1 = READ_ONCE(y);
15   rcu_read_unlock();
16   rcu_read_lock();
17   r2 = READ_ONCE(x);
18   rcu_read_unlock();
19 }
```

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:

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:

```
 1 void thread0(void)
 2 {
 3   rcu_read_lock();
 4   r1 = READ_ONCE(y);
 5   if (r1) {
 6     do_something_with_nonzero_x();
 7     r2 = READ_ONCE(x);
 8     WARN_ON(!r2); /* BUG!!! */
 9   }
10   rcu_read_unlock();
11 }
12
13 void thread1(void)
14 {
15   spin_lock(&my_lock);
16   WRITE_ONCE(x, 1);
17   WRITE_ONCE(y, 1);
18   spin_unlock(&my_lock);
19 }
```

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.

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:

Grace Periods Don't Partition Read-Side Critical Sections

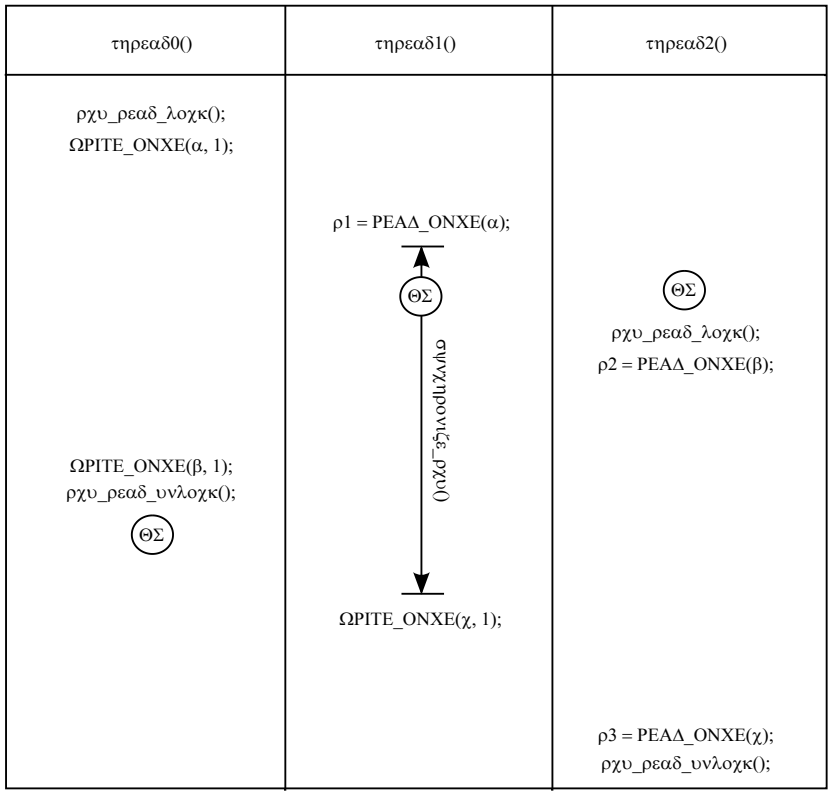
It is tempting to assume that if any part of one RCU read-side critical section precedes a given grace period, and if any part of another RCU read-side critical section follows that same grace period, then all of the first RCU read-side critical section must precede all of the second. However, this just 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:

```
 1 void thread0(void)
 2 {
 3   rcu_read_lock();
 4   WRITE_ONCE(a, 1);
 5   WRITE_ONCE(b, 1);
 6   rcu_read_unlock();
 7 }
 8
 9 void thread1(void)
10 {
11   r1 = READ_ONCE(a);
12   synchronize_rcu();
13   WRITE_ONCE(c, 1);
14 }
15
16 void thread2(void)
17 {
18   rcu_read_lock();
19   r2 = READ_ONCE(b);
20   r3 = READ_ONCE(c);
21   rcu_read_unlock();
22 }
```

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:

```
 1 void thread0(void)
 2 {
 3   rcu_read_lock();
 4   WRITE_ONCE(a, 1);
 5   WRITE_ONCE(b, 1);
 6   rcu_read_unlock();
 7 }
 8
 9 void thread1(void)
10 {
11   r1 = READ_ONCE(a);
12   synchronize_rcu();
13   WRITE_ONCE(c, 1);
14 }
15
16 void thread2(void)
17 {
18   r2 = READ_ONCE(c);
19   synchronize_rcu();
20   WRITE_ONCE(d, 1);
21 }
22
23 void thread3(void)
24 {
25   rcu_read_lock();
26   r3 = READ_ONCE(b);
27   r4 = READ_ONCE(d);
28   rcu_read_unlock();
29 }
```

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.

Read-Side Critical Sections Don't Partition Grace Periods

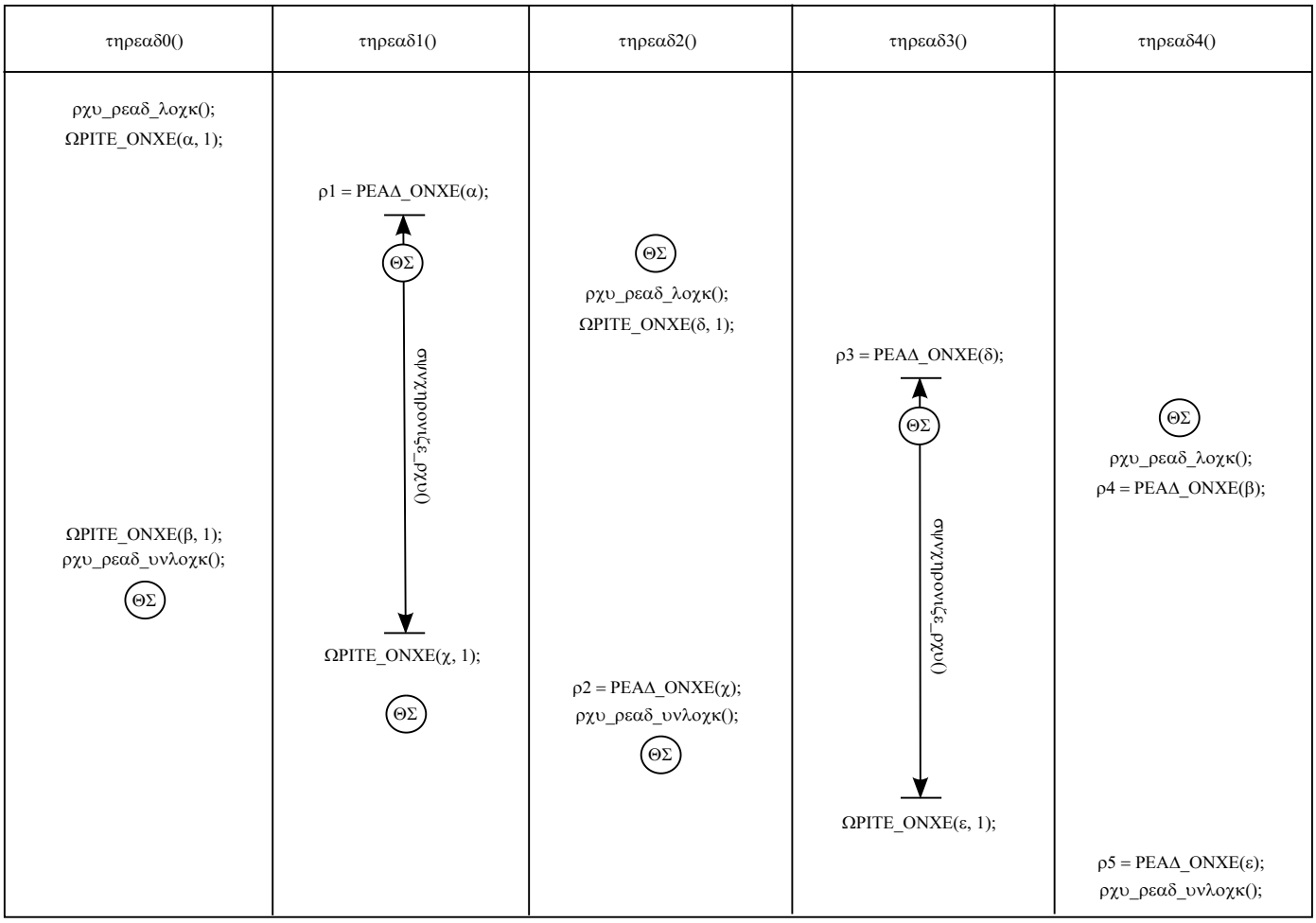
It is also tempting to assume that if an RCU read-side critical section happens between a pair of grace periods, then those grace periods cannot overlap. However, this temptation leads nowhere good, as can be illustrated by the following, with all variables initially zero:

```
 1 void thread0(void)
 2 {
 3   rcu_read_lock();
 4   WRITE_ONCE(a, 1);
 5   WRITE_ONCE(b, 1);
 6   rcu_read_unlock();
 7 }
 8
 9 void thread1(void)
10 {
11   r1 = READ_ONCE(a);
12   synchronize_rcu();
13   WRITE_ONCE(c, 1);
14 }
15
16 void thread2(void)
17 {
18   rcu_read_lock();
19   WRITE_ONCE(d, 1);
20   r2 = READ_ONCE(c);
21   rcu_read_unlock();
22 }
23
24 void thread3(void)
25 {
26   r3 = READ_ONCE(d);
27   synchronize_rcu();
28   WRITE_ONCE(e, 1);
29 }
30
31 void thread4(void)
32 {
33   rcu_read_lock();
34   r4 = READ_ONCE(b);
35   r5 = READ_ONCE(e);
36   rcu_read_unlock();
37 }
```

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:

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.

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.

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:

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.

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_PREEMPT=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:

```
1 struct foo {
2   int a;
3   int b;
4   struct rcu_head rh;
5 };
6
7 static void remove_gp_cb(struct rcu_head *rhp)
8 {
```

```
 9   struct foo *p = container_of(rhp, struct foo, rh);
10
11   kfree(p);
12 }
13
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:

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:

```
 1 struct foo {
 2   int a;
 3   int b;
 4   struct rcu_head rh;
 5 };
 6
 7 bool remove_gp_faf(void)
 8 {
 9   struct foo *p;
10
11   spin_lock(&gp_lock);
12   p = rcu_dereference(gp);
13   if (!p) {
14     spin_unlock(&gp_lock);
15     return false;
16   }
```

```
17   rcu_assign_pointer(gp, NULL);
18   kfree_rcu(p, rh);
19   spin_unlock(&gp_lock);
20   return true;
21 }
```

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:

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:

```
 1 bool remove_gp_poll(void)
 2 {
 3   struct foo *p;
 4   unsigned long s;
 5
 6   spin_lock(&gp_lock);
 7   p = rcu_access_pointer(gp);
 8   if (!p) {
 9     spin_unlock(&gp_lock);
10     return false;
11   }
12   rcu_assign_pointer(gp, NULL);
13   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.

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.

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_PREEMPT=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.

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.

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 and also for assigning `NULL` pointers at runtime.

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.

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 non-maskable interrupts (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.

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.

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.

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_PREEMPT=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:

I learned of these boot-time requirements as a result of a series of system hangs.

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 `rcu_irq_enter()` and `rcu_irq_exit()` to be called from an NMI handler. This astonishing fact of life prompted the current code structure, which has `rcu_irq_enter()` invoking `rcu_nmi_enter()` and `rcu_irq_exit()` invoking `rcu_nmi_exit()`. And yes, I also learned of this requirement the hard way.

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 `del_timer_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 note: 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:

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, all-callback-wait operations such as `rcu_barrier()` are also not supported, 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.

Scheduler and RCU

RCU depends on the scheduler, and the scheduler uses RCU to protect some of its data structures. The preemptible-RCU `rcu_read_unlock()` implementation must therefore be written carefully to avoid deadlocks involving the scheduler's runqueue and priority-inheritance locks. In particular, `rcu_read_unlock()` must tolerate an interrupt where the interrupt handler invokes both `rcu_read_lock()` and `rcu_read_unlock()`. This possibility requires `rcu_read_unlock()` to use negative nesting levels to avoid destructive recursion via interrupt handler's use of RCU.

This scheduler-RCU requirement came as a complete surprise.

As noted above, 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.

It is forbidden to hold any of scheduler's runqueue or priority-inheritance spinlocks across an `rcu_read_unlock()` unless interrupts have been disabled across the entire RCU read-side critical section, that is, up to and including the matching `rcu_read_lock()`. Violating this restriction can result in deadlocks involving these scheduler spinlocks. There was hope that this restriction might be lifted when interrupt-disabled calls to `rcu_read_unlock()` started deferring the reporting of the resulting RCU-preempt quiescent state until the end of the corresponding interrupts-disabled region. Unfortunately, timely reporting of the corresponding quiescent state to expedited grace periods requires a call to `raise_softirq()`, which can acquire these scheduler spinlocks. In addition, real-time systems using RCU priority boosting need this restriction to remain in effect because deferred quiescent-state reporting would also defer deboosting, which in turn would degrade real-time latencies.

In theory, if a given RCU read-side critical section could be guaranteed to be less than one second in duration, holding a scheduler spinlock across that critical section's `rcu_read_unlock()` would require only that preemption be disabled across the entire RCU read-side critical section, not interrupts. Unfortunately, given the possibility of vCPU preemption, long-running interrupts, and so on, it is not possible in practice to guarantee that a given RCU read-side critical section will complete in less than one second. Therefore, as

noted above, if scheduler spinlocks are held across a given call to `rcu_read_unlock()`, interrupts must be disabled across the entire RCU read-side critical section.

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.

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:

```
1 rcu_read_lock();
2 p = rcu_dereference(gp);
3 v = p->value;
4 rcu_read_unlock();
5 get_user(user_v, user_p);
6 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_PREEMPT=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.

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.) The `RCU_NONIDLE()` macro and

_rcuidle event tracing is provided to work around this restriction. In addition, rcu_is_watching() may be used to test whether or not it is currently legal to run RCU read-side critical sections on this CPU. I learned of the need for diagnostics on the one hand and RCU_NONIDLE() on the other while inspecting idle-loop code. Steven Rostedt supplied _rcuidle event tracing, which is used quite heavily in the idle loop. However, there are some restrictions on the code placed within RCU_NONIDLE():

1. Blocking is prohibited. In practice, this is not a serious restriction given that idle tasks are prohibited from blocking to begin with.
2. Although nesting RCU_NONIDLE() is permitted, they cannot nest indefinitely deeply. However, given that they can be nested on the order of a million deep, even on 32-bit systems, this should not be a serious restriction. This nesting limit would probably be reached long after the compiler OOMed or the stack overflowed.
3. Any code path that enters RCU_NONIDLE() must sequence out of that same RCU_NONIDLE(). For example, the following is grossly illegal:

```
1     RCU_NONIDLE({
2       do_something();
3       goto bad_idea;  /* BUG!!! */
4       do_something_else();});
5   bad_idea:
```

   It is just as illegal to transfer control into the middle of RCU_NONIDLE()'s argument. Yes, in theory, you could transfer in as long as you also transferred out, but in practice you could also expect to get sharply worded review comments.

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!

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:

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 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 rcu_irq_enter() and rcu_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:

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!

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.

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_PREEMPT=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.

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

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()`, `synchronize_rcu_bh()`, `synchronize_rcu_bh_expedited()`, `call_rcu_bh()`, `rcu_barrier_bh()`, and `rcu_read_lock_bh_held()`. However, the update-side APIs are now simple wrappers for other RCU flavors, namely RCU-sched in CONFIG_PREEMPT=n kernels and RCU-preempt otherwise.

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 for pre-existing interrupt and NMI handlers. In kernels built with `CONFIG_PREEMPT=n`, the RCU and RCU-sched APIs have identical implementations, while kernels built with `CONFIG_PREEMPT=y` provide a separate implementation for each.

Note well that in `CONFIG_PREEMPT=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()`, `synchronize_sched()`, `synchronize_rcu_sched_expedited()`, `call_rcu_sched()`, `rcu_barrier_sched()`, and `rcu_read_lock_sched_held()`. However, 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.

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.

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.

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_PREEMPT=n` kernels, trampolines cannot be preempted, so these APIs map to `call_rcu()`, `synchronize_rcu()`, and `rcu_barrier()`, respectively. In `CONFIG_PREEMPT=y` kernels, trampolines can be preempted, and these three APIs are therefore implemented by separate functions that check for voluntary context switches.

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.

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.

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.