# What is RCU? -- "Read, Copy, Update"

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

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

## What is RCU?

RCU is a synchronization mechanism that was added to the Linux kernel during the 2.5 development effort that is optimized for read-mostly situations. Although RCU is actually quite simple once you understand it, getting there can sometimes be a challenge. Part of the problem is that most of the past descriptions of RCU have been written with the mistaken assumption that there is "one true way" to describe RCU. Instead, the experience has been that different people must take different paths to arrive at an understanding of RCU. This document provides several different paths, as follows:

1. RCU OVERVIEW
2. WHAT IS RCU'S CORE API?
3. WHAT ARE SOME EXAMPLE USES OF CORE RCU API?
4. WHAT IF MY UPDATING THREAD CANNOT BLOCK?
5. WHAT ARE SOME SIMPLE IMPLEMENTATIONS OF RCU?
6. ANALOGY WITH READER-WRITER LOCKING
7. FULL LIST OF RCU APIs
8. ANSWERS TO QUICK QUIZZES

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

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

## 1. RCU OVERVIEW

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

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

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

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

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

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

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

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

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

## 2. WHAT IS RCU'S CORE API?

The core RCU API is quite small:

a. rcu_read_lock()
b. rcu_read_unlock()
c. synchronize_rcu() / call_rcu()
d. rcu_assign_pointer()
e. rcu_dereference()

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

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

rcu_read_lock()

```
void rcu_read_lock(void);
```

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

rcu_read_unlock()

```
void rcu_read_unlock(void);
```

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

synchronize_rcu()

```
void synchronize_rcu(void);
```

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

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

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

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

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

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

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

rcu_assign_pointer()

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

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

The updater uses this function to assign a new value to an RCU-protected pointer, in order to safely communicate the change in value from the updater to the reader. This macro does not evaluate to an rvalue, but it does execute any memory-barrier instructions required for a given CPU architecture.

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

rcu_dereference()

```
typeof(p) rcu_dereference(p);
```

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

The reader uses rcu_dereference() to fetch an RCU-protected pointer, which returns a value that may then be safely dereferenced. Note that rcu_dereference() does not actually dereference the pointer, instead, it protects the pointer for later dereferencing. It also executes any needed memory-barrier instructions for a given CPU architecture. Currently, only Alpha needs memory barriers within rcu_dereference() -- on other CPUs, it compiles to nothing, not even a compiler directive.

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

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

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

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

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

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

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

```
		y = p->data;	/* BUG!!! */
		rcu_read_unlock();
```

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

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

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

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

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

```
	    rcu_assign_pointer()
	                            +--------+
	    +---------------------->| reader |---------+
	    |                       +--------+         |
	    |                           |              |
	    |                           |              | Protect:
	    |                           |              | rcu_read_lock()
	    |                           |              | rcu_read_unlock()
	    |        rcu_dereference()  |              |
	    +---------+                 |              |
	    | updater |<----------------+              |
	    +---------+                                V
	    |                                    +-----------+
	    +----------------------------------->| reclaimer |
	                                         +-----------+
	      Defer:
	      synchronize_rcu() & call_rcu()
```

The RCU infrastructure observes the time sequence of rcu_read_lock(), rcu_read_unlock(), synchronize_rcu(), and call_rcu() invocations in order to determine when (1) synchronize_rcu() invocations may return to their callers and (2) call_rcu() callbacks may be invoked. Efficient implementations of the RCU infrastructure make heavy use of batching in order to amortize their overhead over many uses of the corresponding APIs.

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

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

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

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

These three flavors are used as follows:

a. RCU applied to normal data structures.

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

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

Again, most uses will be of (a). The (b) and (c) cases are important for specialized uses, but are relatively uncommon.

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

This section shows a simple use of the core RCU API to protect a global pointer to a dynamically allocated structure. More-typical uses of RCU may be found in listRCU.txt, arrayRCU.txt, and NMI-RCU.txt.

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

	struct foo __rcu *gbl_foo;

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

		new_fp = kmalloc(sizeof(*new_fp), GFP_KERNEL);
		spin_lock(&foo_mutex);
		old_fp = rcu_dereference_protected(gbl_foo, lockdep_is_held(&foo_mutex));
```

```
                *new_fp = *old_fp;
                new_fp->a = new_a;
                rcu_assign_pointer(gbl_foo, new_fp);
                spin_unlock(&foo_mutex);
                synchronize_rcu();
                kfree(old_fp);
        }

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

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

So, to sum up:

- Use rcu_read_lock() and rcu_read_unlock() to guard RCU read-side critical sections.

- Within an RCU read-side critical section, use rcu_dereference() to dereference RCU-protected pointers.

- Use some solid scheme (such as locks or semaphores) to keep concurrent updates from interfering with each other.

- Use rcu_assign_pointer() to update an RCU-protected pointer. This primitive protects concurrent readers from the updater, -not- concurrent updates from each other!  You therefore still need to use locking (or something similar) to keep concurrent rcu_assign_pointer() primitives from interfering with each other.

- Use synchronize_rcu() -after- removing a data element from an RCU-protected data structure, but -before- reclaiming/freeing the data element, in order to wait for the completion of all RCU read-side critical sections that might be referencing that data item.

See checklist.txt for additional rules to follow when using RCU. And again, more-typical uses of RCU may be found in listRCU.txt, arrayRCU.txt, and NMI-RCU.txt.

## 4.  WHAT IF MY UPDATING THREAD CANNOT BLOCK?

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

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

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

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

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

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

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

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

The foo_reclaim() function might appear as follows:

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

        foo_cleanup(fp->a);

        kfree(fp);
}
```

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

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

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

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

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

        kfree_rcu(old_fp, rcu);

Again, see checklist.txt for additional rules governing the use of RCU.


5.  WHAT ARE SOME SIMPLE IMPLEMENTATIONS OF RCU?

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

        http://www.rdrop.com/users/paulmck/RCU

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


5A.  "TOY" IMPLEMENTATION #1: LOCKING

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

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

It is extremely simple:

        static DEFINE_RWLOCK(rcu_gp_mutex);

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

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

        void synchronize_rcu(void)
        {
                write_lock(&rcu_gp_mutex);
                smp_mb__after_spinlock();
                write_unlock(&rcu_gp_mutex);
        }

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

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

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

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

	Documentation/RCU/Design/Requirements/Requirements.rst

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

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

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

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

```
	void rcu_read_lock(void) { }

	void rcu_read_unlock(void) { }

	void synchronize_rcu(void)
	{
		int cpu;

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

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

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

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

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

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

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

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

## 6. ANALOGY WITH READER-WRITER LOCKING

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

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

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

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

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

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

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

```
struct el {                            struct el {
  struct list_head list;                 struct list_head list;
  long key;                              long key;
  spinlock_t mutex;                      spinlock_t mutex;
  int data;                              int data;
  /* Other data fields */                /* Other data fields */
};                                     };
rwlock_t listmutex;                    spinlock_t listmutex;
struct el head;                        struct el head;

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

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

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

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

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

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

Also, the presence of synchronize_rcu() means that the RCU version of delete() can now block. If this is a problem, there is a callback-based mechanism that never blocks, namely call_rcu() or kfree_rcu(), that can be used in place of synchronize_rcu().

## 7. FULL LIST OF RCU APIs

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

RCU list traversal:

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

RCU pointer/list update:

```
        rcu_assign_pointer
        list_add_rcu
        list_add_tail_rcu
        list_del_rcu
        list_replace_rcu
        hlist_add_behind_rcu
        hlist_add_before_rcu
        hlist_add_head_rcu
        hlist_del_rcu
        hlist_del_init_rcu
        hlist_replace_rcu
        list_splice_init_rcu()
        hlist_nulls_del_init_rcu
        hlist_nulls_del_rcu
        hlist_nulls_add_head_rcu
        hlist_bl_add_head_rcu
        hlist_bl_del_init_rcu
        hlist_bl_del_rcu
        hlist_bl_set_first_rcu
```

| RCU: | Critical sections | Grace period | Barrier |
|---|---|---|---|
| | rcu_read_lock | synchronize_net | rcu_barrier |
| | rcu_read_unlock | synchronize_rcu | |
| | rcu_dereference | synchronize_rcu_expedited | |
| | rcu_read_lock_held | call_rcu | |
| | rcu_dereference_check | kfree_rcu | |
| | rcu_dereference_protected | | |

| bh: | Critical sections | Grace period | Barrier |
|---|---|---|---|
| | rcu_read_lock_bh | call_rcu | rcu_barrier |
| | rcu_read_unlock_bh | synchronize_rcu | |
| | [local_bh_disable] | synchronize_rcu_expedited | |
| | [and friends] | | |
| | rcu_dereference_bh | | |
| | rcu_dereference_bh_check | | |
| | rcu_dereference_bh_protected | | |
| | rcu_read_lock_bh_held | | |

```
sched:  Critical sections       Grace period            Barrier

        rcu_read_lock_sched     call_rcu                rcu_barrier
        rcu_read_unlock_sched   synchronize_rcu
        [preempt_disable]       synchronize_rcu_expedited
        [and friends]
        rcu_read_lock_sched_notrace
        rcu_read_unlock_sched_notrace
        rcu_dereference_sched
        rcu_dereference_sched_check
        rcu_dereference_sched_protected
        rcu_read_lock_sched_held


SRCU:   Critical sections       Grace period            Barrier

        srcu_read_lock          call_srcu               srcu_barrier
        srcu_read_unlock        synchronize_srcu
        srcu_dereference        synchronize_srcu_expedited
        srcu_dereference_check
        srcu_read_lock_held

SRCU:   Initialization/cleanup
        DEFINE_SRCU
        DEFINE_STATIC_SRCU
        init_srcu_struct
        cleanup_srcu_struct

All:  lockdep-checked RCU-protected pointer access

        rcu_access_pointer
        rcu_dereference_raw
        RCU_LOCKDEP_WARN
        rcu_sleep_check
        RCU_NONIDLE
```

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

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

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

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

c. Do you need to treat NMI handlers, hardirq handlers, and code segments with preemption disabled (whether via preempt_disable(), local_irq_save(), local_bh_disable(), or some other mechanism) as if they were explicit RCU readers? If so, RCU-sched is the only choice that will work for you.

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

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

f. Do you need read-side critical sections that are respected even though they are in the middle of the idle loop, during user-mode execution, or on an offlined CPU? If so, SRCU is the only choice that will work for you.

g. Otherwise, use RCU.

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

## 8. ANSWERS TO QUICK QUIZZES

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

Answer: Consider the following sequence of events:

1. CPU 0 acquires some unrelated lock, call it "problematic_lock", disabling irq via spin_lock_irqsave().

2. CPU 1 enters synchronize_rcu(), write-acquiring rcu_gp_mutex.

3. CPU 0 enters rcu_read_lock(), but must wait because CPU 1 holds rcu_gp_mutex.

4. CPU 1 is interrupted, and the irq handler attempts to acquire problematic_lock.

The system is now deadlocked.

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

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

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

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

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

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

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

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

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

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

## ACKNOWLEDGEMENTS

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

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