RCU part 3: the RCU API

January 7, 2008
This article was contributed by Paul McKenney

[*Editor's note: this is the third and final installment in Paul McKenney's "What is RCU?" series. The first and secondparts remain available for those who might have missed them. Many thanks to Paul for letting LWN run these articles.*]

Introduction

Read-copy update (RCU) is a synchronization mechanism that was added to the Linux kernel in October of 2002. RCU is most frequently described as a replacement for reader-writer locking, but has also been used in a number of other ways. RCU is notable in that RCU readers do not directly synchronize with RCU updaters, which makes RCU read paths extremely fast, and also permits RCU readers to accomplish useful work even when running concurrently with RCU updaters.

This leads to the question "what exactly is RCU?", a question that this document addresses from the viewpoint of the Linux kernel's RCU API.

1. RCU has a Family of Wait-to-Finish APIs
2. RCU has Publish-Subscribe and Version-Maintenance APIs
3. So, What is RCU Really?

These sections are followed by a references section and the answers to the Quick Quizzes.

RCU has a Family of Wait-to-Finish APIs

The most straightforward answer to "what is RCU" is that RCU is an API used in the Linux kernel, as summarized by the pair of tables in this section (the first table shows the wait-for-RCU-readers portions of the API, while the second table shows the publish/subscribe portions of the API). Or, more precisely, RCU is a family of APIs as shown in the first table, with each column corresponding to a member of the RCU API family.

If you are new to RCU, you might consider focusing on just one of the columns in the following table. For example, if you are primarily interested in understanding how RCU is used in the Linux kernel, "RCU Classic" would be the place to start, as it is used most frequently. On the other hand, if you want to understand RCU for its own sake, "SRCU" has the simplest API. You can always come back for the other columns later.

If you are already familiar with RCU, the following pair of tables can serve as a useful reference.

Attribute	RCU Classic	RCU BH	RCU Sched	Realtime RCU	SRCU	QRCU
Purpose	Original	Prevent DDoS attacks	Wait for hardirqs and NMIs	Realtime response	Sleeping readers	Sleeping readers and fast grace periods
Availability	2.5.43	2.6.9	2.6.12	Aug 2005 -rt	2.6.19	
Read-side primitives	`rcu_read_lock()` `rcu_read_unlock()`	`rcu_read_lock_bh()` `rcu_read_unlock_bh()`	`preempt_disable()` `preempt_enable()` (and friends)	`rcu_read_lock()` `rcu_read_unlock()`	`srcu_read_lock()` `srcu_read_unlock()`	`qrcu_read_lock()` `qrcu_read_unlock()`
Update-side primitives (synchronous)	`synchronize_rcu()` `synchronize_net()`		`synchronize_sched()`	`synchronize_rcu()` `synchronize_net()`	`synchronize_srcu()`	`synchronize_qrcu()`
Update-side primitives (asynchronous/callback)	`call_rcu()`	`call_rcu_bh()`		`call_rcu()`	N/A	N/A
Update-side primitives (wait for callbacks)	`rcu_barrier()`			`rcu_barrier()`	N/A	N/A
Read side constraints	No blocking	No irq enabling	No blocking	No blocking except preemption and lock acquisition	No `synchronize_srcu()`	No `synchronize_qrcu(`
Read side overhead	Preempt disable/enable (free on non-PREEMPT)	BH disable/enable	Preempt disable/enable (free on non-PREEMPT)	Simple instructions, irq disable/enable	Simple instructions, preempt disable/enable	Atomic increment and decrement of shared variable
Asynchronous update-side overhead (for example, `call_rcu()`)	sub-microsecond	sub-microsecond		sub-microsecond	N/A	N/A
Grace-period latency	10s of milliseconds	10s of milliseconds	10s of milliseconds	10s of milliseconds	10s of milliseconds	10s of *nanoseconds* ir absence of readers
Non-PREEMPT_RT implementation	RCU Classic	RCU BH	RCU Classic	N/A	SRCU	N/A
PREEMPT_RT implementation	N/A	Realtime RCU	Forced Schedule on all CPUs	Realtime RCU	SRCU	N/A

Quick Quiz 1: Why are some of the cells in the above table colored green?

The "RCU Classic" column corresponds to the original RCU implementation, in which RCU read-side critical sections are delimited by `rcu_read_lock()`and `rcu_read_unlock()`, which may be nested. The corresponding synchronous update-side primitives, `synchronize_rcu()`, along with its synonym `synchronize_net()`, wait for any currently executing RCU read-side critical sections to complete. The length of this wait is known as a "grace period". The asynchronous update-side primitive, `call_rcu()`, invokes a specified function with a specified argument after a subsequent grace period. For example, `call_rcu(p,f);` will result in the "RCU callback" `f(p)`being invoked after a subsequent grace period. There are situations, such as when unloading a module that uses `call_rcu()`, when it is necessary to wait for all outstanding RCU callbacks to complete. The `rcu_barrier()`primitive does this job.

In the "RCU BH" column, `rcu_read_lock_bh()` and `rcu_read_unlock_bh()`delimit RCU read-side critical sections, and `call_rcu_bh()` invokes the specified function and argument after a subsequent grace period. Note that RCU BH does not have a synchronous `synchronize_rcu_bh()` interface, though one could easily be added if required.

Quick Quiz 2: What happens if you mix and match? For example, suppose you use `rcu_read_lock()` and `rcu_read_unlock()` to delimit RCU read-side critical sections, but then use `call_rcu_bh()` to post an RCU callback?

In the "RCU Sched" column, anything that disables preemption acts as an RCU read-side critical section, and `synchronize_sched()` waits for the corresponding RCU grace period. This RCU API family was added in the 2.6.12 kernel, which split the old `synchronize_kernel()` API into the current`synchronize_rcu()` (for RCU Classic) and `synchronize_sched()` (for RCU Sched). Note that RCU Sched does not have an asynchronous`call_rcu_sched()` interface, though one could be added if required.

Quick Quiz 3: What happens if you mix and match RCU Classic and RCU Sched?

The "Realtime RCU" column has the same API as does RCU Classic, the only difference being that RCU read-side critical sections may be preempted and may block while acquiring spinlocks. The design of Realtime RCU is described in the LWN article The design of preemptible read-copy-update.

Quick Quiz 4: What happens if you mix and match Realtime RCU and RCU Classic?

The "SRCU" column displays a specialized RCU API that permits general sleeping in RCU read-side critical sections, as was described in the LWN article Sleepable RCU. Of course, use of `synchronize_srcu()` in an SRCU read-side critical section can result in self-deadlock, so should be avoided. SRCU differs from earlier RCU implementations in that the caller allocates an `srcu_struct` for each distinct SRCU usage. This approach prevents SRCU read-side critical sections from blocking unrelated `synchronize_srcu()` invocations. In addition, in this variant of RCU, `srcu_read_lock()` returns a value that must be passed into the corresponding `srcu_read_unlock()`.

The "QRCU" column presents an RCU implementation with the same API structure as SRCU, but optimized for extremely low-latency grace periods in absence of readers, as described in the LWN article Using Promela and Spin to verify parallel algorithms. As with SRCU, use of `synchronize_qrcu()`can result in self-deadlock, so should be avoided. Although QRCU has not yet been accepted into the Linux kernel, it is worth mentioning given that it is the only RCU implementation that can boast deep sub-microsecond grace-period latencies.

Quick Quiz 5: Why do both SRCU and QRCU lack asynchronous `call_srcu()` or `call_qrcu()` interfaces?

Quick Quiz 6: Under what conditions can `synchronize_srcu()` be safely used within an SRCU read-side critical section?

The Linux kernel currently has a surprising number of RCU APIs and implementations. There is some hope of reducing this number, evidenced by the fact that a given build of the Linux kernel currently has at most three implementations behind four APIs (given that RCU Classic and Realtime RCU share the same API). However, careful inspection and analysis will be required, just as would be required for one of the many locking APIs.

RCU has Publish-Subscribe and Version-Maintenance APIs

Fortunately, the RCU publish-subscribe and version-maintenance primitives shown in the following table apply to all of the variants of RCU discussed above. This commonality can in some cases allow more code to be shared, which certainly reduces the API proliferation that would otherwise occur.

Category	Primitives	Availability	Overhead
List traversal	`list_for_each_entry_rcu()`	2.5.59	Simple instructions (memory barrier on Alpha)
	`list_add_rcu()`	2.5.44	Memory barrier
	`list_add_tail_rcu()`	2.5.44	Memory barrier
List update	`list_del_rcu()`	2.5.44	Simple instructions
	`list_replace_rcu()`	2.6.9	Memory barrier
	`list_splice_init_rcu()`	2.6.21	Grace-period latency
Hlist traversal	`hlist_for_each_entry_rcu()`	2.6.8	Simple instructions (memory barrier on Alpha)
	`hlist_add_after_rcu()`	2.6.14	Memory barrier
	`hlist_add_before_rcu()`	2.6.14	Memory barrier
Hlist update	`hlist_add_head_rcu()`	2.5.64	Memory barrier
	`hlist_del_rcu()`	2.5.64	Simple instructions
	`hlist_replace_rcu()`	2.6.15	Memory barrier
Pointer traversal	`rcu_dereference()`	2.6.9	Simple instructions (memory barrier on Alpha)
Pointer update	`rcu_assign_pointer()`	2.6.10	Memory barrier

The first pair of categories operate on Linux `struct list_head` lists, which are circular, doubly-linked lists. The `list_for_each_entry_rcu()` primitive traverses an RCU-protected list in a type-safe manner, while also enforcing memory ordering for situations where a new list element is inserted into the list concurrently with traversal. On non-Alpha platforms, this primitive incurs little or no performance penalty compared to `list_for_each_entry()`. The `list_add_rcu()`, `list_add_tail_rcu()`, and `list_replace_rcu()` primitives are analogous to their non-RCU counterparts, but incur the overhead of an additional memory barrier on weakly-ordered machines. The `list_del_rcu()`primitive is also analogous to its non-RCU counterpart, but oddly enough is very slightly faster due to the fact that it poisons only the `prev` pointer rather than both the `prev` and `next` pointers as `list_del()` must do. Finally, the `list_splice_init_rcu()` primitive is similar to its non-RCU counterpart, but incurs a full grace-period latency. The purpose of this grace period is to allow RCU readers to finish their traversal of the source list before completely disconnecting it from the list header -- failure to do this could prevent such readers from ever terminating their traversal.

Quick Quiz 7: Why doesn't `list_del_rcu()` poison both the `next` and `prev`pointers?

The second pair of categories operate on Linux's `struct hlist_head`, which is a linear linked list. One advantage of `struct hlist_head` over`struct list_head` is that the former requires only a single-pointer list header, which can save significant memory in large hash tables. The `struct hlist_head` primitives in the table relate to their non-RCU counterparts in much the same way as do the `struct list_head` primitives.

The final pair of categories operate directly on pointers, and are useful for creating RCU-protected non-list data structures, such as RCU-protected arrays and trees. The `rcu_assign_pointer()` primitive ensures that any prior initialization remains ordered before the assignment to the pointer on weakly ordered machines. Similarly, the `rcu_dereference()` primitive ensures that subsequent code dereferencing the pointer will see the effects of initialization code prior to the corresponding `rcu_assign_pointer()` on Alpha CPUs. On non-Alpha CPUs, `rcu_dereference()` documents which pointer dereferences are protected by RCU.

Quick Quiz 8: Normally, any pointer subject to `rcu_dereference()` should always be updated using `rcu_assign_pointer()`. What is an exception to this rule?

Quick Quiz 9: Are there any downsides to the fact that these traversal and update primitives can be used with any of the RCU API family members?

So, What *is* RCU Really?

At its core, RCU is nothing more nor less than an API that supports publication and subscription for insertions, waiting for all RCU readers to complete, and maintenance of multiple versions. That said, it is possible to build higher-level constructs on top of RCU, including the reader-writer-locking, reference-counting, and existence-guarantee constructs listed in the companion article. Furthermore, I have no doubt that the Linux community will continue to find interesting new uses for RCU, just as they do for any of a number of synchronization primitives throughout the kernel.

Finally, a complete view of RCU would also include all of the things you can do with these APIs.

Acknowledgements

We are all indebted to Andy Whitcroft, Jon Walpole, and Gautham Shenoy, whose review of an early draft of this document greatly improved it. I owe thanks to the members of the Relativistic Programming project and to members of PNW TEC for many valuable discussions. I am grateful to Dan Frye for his support of this effort.

References

This section gives a short annotated bibliography describing using RCU, Linux-kernel RCU implementations, background, and historical perspectives. For more information, see Paul E. McKenney's RCU Page.

Using RCU

1. Overview of Linux-Kernel Reference Counting (McKenney, January 2007) [PDF]. Overview of Linux-kernel reference counting (including RCU) prepared for the Concurrency Working Group of the C/C++ standards committee.

2. RCU and Unloadable Modules (McKenney, January 2007). Describes how to unload modules that use `call_rcu()`, so as to avoid RCU callbacks trying to use the module after it has been unloaded.

3. Recent Developments in SELinux Kernel Performance. James Morris describes a performance problem in the SELinux Access Vector Cache (AVC), and its resolution via RCU in a patch by Kaigai Kohei.

4. Using Read-Copy-Update Techniques for System V IPC in the Linux 2.5 Kernel (Arcangeli et al., June 2003) [PDF]. Describes how RCU is used in the Linux kernel's System V IPC implementation.

Linux-Kernel RCU Implementations

1. The design of preemptible read-copy-update (McKenney, October 2007). Describes a high-performance RCU implementation for realtime use.

2. Sleepable RCU (McKenney, October 2006). Description of SRCU.

3. Using Promela and Spin to verify parallel algorithms (McKenney, August 2007). Description of the QRCU patch.

4. RCU dissertation (McKenney, July 2004) [PDF].
   - Section 2.2.20 (pages 62-64) gives a history of RCU-like mechanisms, a very brief summary of which can be found below.
   - Chapter 4 (pages 71-98) and Appendix C (pages 326-345) review a number of different types of RCU implementations, summarizing a number of earlier papers.
   - Chapter 5 (pages 137-178) gives an overview of a number of "design patterns" guiding use of RCU.
   - Chapter 6 (pages 179-234) describes some early uses of RCU.

5. Using RCU in the Linux 2.5 Kernel (October 2003). Brief summary of why RCU can be helpful, along with an analogy between RCU and reader-writer locking.

6. Anyone who is laboring under the misapprehension that the Linux community would never have independently invented RCU should read this netdev posting and this one as well. Both postings pre-date the earliest known introduction of RCU to the Linux community.

Background

1. Real-Time Linux Wiki. Provides much valuable information on the -rt patchset for both kernel and application developers.

2. Home of the -rt kernel patchsets.

3. Memory Ordering in Modern Microprocessors (McKenney, August 2005) [PDF]. Gives an overview of how Linux's memory-ordering primitives work on a number of computer architectures.

Historical Perspectives on RCU and Related Mechanisms

1. Tornado: Maximizing Locality and Concurrency in a Shared Memory Multiprocessor Operating System (Gamsa et al., February 1999) [PDF]. Independent invention of a mechanism very similar to RCU. Tornado is a research operating system developed at the University of Toronto. This operating system uses its analog to RCU pervasively. Some of the University of Toronto students brought this operating system with them to IBM Research, where it was developed as part of the K42 project.

2. Read-Copy Update: Using Execution History to Solve Concurrency Problems (McKenney and Slingwine, October 1998) [PDF]. First non-patent publication of DYNIX/ptx's RCU implementation.

3. Passive Serialization in a Multitasking Environment (Hennessey et al., February 1989). This patent describes an RCU-like mechanism that was apparently used in IBM's VM/XA mainframe hypervisor. This is the earliest known production use of an RCU-like mechanism.

4. Concurrent Manipulation of Binary Search Trees (Kung and Lehman, September 1980). The earliest known publication of an RCU-like mechanism, using a garbage collector to implicitly compute grace periods.

Answers to Quick Quizzes

Quick Quiz 1: Why are some of the cells in the above table colored green?

Answer: The green API members (`rcu_read_lock()`, `rcu_read_unlock()`, and `call_rcu()`) were the only members of the Linux RCU API that Paul E. McKenney was aware of back in the mid-90s. During this timeframe, he was under the mistaken impression that he knew all that there is to know about RCU.

Back to Quick Quiz 1.

Quick Quiz 2: What happens if you mix and match? For example, suppose you use `rcu_read_lock()` and `rcu_read_unlock()` to delimit RCU read-side critical sections, but then use `call_rcu_bh()` to post an RCU callback?

Answer: If there happened to be no RCU read-side critical sections delimited by `rcu_read_lock_bh()` and `rcu_read_unlock_bh()` at the time `call_rcu_bh()` was invoked, RCU would be within its rights to invoke the callback immediately, possibly freeing a data structure still being used by the RCU read-side critical section! This is not merely a theoretical possibility: a long-running RCU read-side critical section delimited by `rcu_read_lock()`and `rcu_read_unlock()` is vulnerable to this failure mode.

This vulnerability disappears in -rt kernels, where RCU Classic and RCU BH both map onto a common implementation.

Back to Quick Quiz 2.

Quick Quiz 3: What happens if you mix and match RCU Classic and RCU Sched?

Answer: In a non-PREEMPT or a PREEMPT kernel, mixing these two works "by accident" because in those kernel builds, RCU Classic and RCU Sched map to the same implementation. However, this mixture is fatal in PREEMPT_RT builds using the -rt patchset, due to the fact that Realtime RCU's read-side critical sections can be preempted, which would permit`synchronize_sched()` to return before the RCU read-side critical section reached its `rcu_read_unlock()` call. This could in turn result in a data structure being freed before the read-side critical section was finished with it, which could in turn greatly increase the actuarial risk experienced by your kernel.

In fact, the split between RCU Classic and RCU Sched was inspired by the need for preemptible RCU read-side critical sections.

Back to Quick Quiz 3.

Quick Quiz 4: What happens if you mix and match Realtime RCU and RCU Classic?

Answer: That would be up to you, because you would have to code up changes to the kernel to make such mixing possible. Currently, any kernel running with RCU Classic cannot access Realtime RCU and vice versa.

Back to Quick Quiz 4.

Quick Quiz 5: Why do both SRCU and QRCU lack asynchronous `call_srcu()` or `call_qrcu()` interfaces?

Answer: Given an asynchronous interface, a single task could register an arbitrarily large number of SRCU or QRCU callbacks, thereby consuming an arbitrarily large quantity of memory. In contrast, given the current synchronous `synchronize_srcu()` and `synchronize_qrcu()` interfaces, a given task must finish waiting for a given grace period before it can start waiting for the next one.

Back to Quick Quiz 5.

Quick Quiz 6: Under what conditions can `synchronize_srcu()` be safely used within an SRCU read-side critical section?

Answer: In principle, you can use `synchronize_srcu()` with a given `srcu_struct` within an SRCU read-side critical section that uses some other`srcu_struct`. In practice, however, doing this is almost certainly a bad idea. In particular, the following could still result in deadlock:

```
idx = srcu_read_lock(&ssa);
synchronize_srcu(&ssb);
srcu_read_unlock(&ssa, idx);

/* . . . */

idx = srcu_read_lock(&ssb);
synchronize_srcu(&ssa);
srcu_read_unlock(&ssb, idx);
```

Back to Quick Quiz 6.

Quick Quiz 7: Why doesn't `list_del_rcu()` poison both the `next` and `prev`pointers?

Answer: Poisoning the `next` pointer would interfere with concurrent RCU readers, who must use this pointer. However, RCU readers are forbidden from using the `prev` pointer, so it may safely be poisoned.

Back to Quick Quiz 7.

Quick Quiz 8: Normally, any pointer subject to `rcu_dereference()` *must*always be updated using `rcu_assign_pointer()`. What is an exception to this rule?

Answer: One such exception is when a multi-element linked data structure is initialized as a unit while inaccessible to other CPUs, and then a single `rcu_assign_pointer()` is used to plant a global pointer to this data structure. The initialization-time pointer assignments need not use`rcu_assign_pointer()`, though any such assignments that happen after the structure is globally visible *must* use `rcu_assign_pointer()`.

However, unless this initialization code is on an impressively hot code-path, it is probably wise to use `rcu_assign_pointer()` anyway, even though it is in theory unnecessary. It is all too easy for a "minor" change to invalidate your cherished assumptions about the initialization happening privately.

Back to Quick Quiz 8.

Quick Quiz 9: Are there any downsides to the fact that these traversal and update primitives can be used with any of the RCU API family members?

Answer: It can sometimes be difficult for automated code checkers such as "sparse" (or indeed for human beings) to work out which type of RCU read-side critical section a given RCU traversal primitive corresponds to. For example, consider the following:

```
rcu_read_lock();
preempt_disable();
p = rcu_dereference(global_pointer);

/* . . . */

preempt_enable();
rcu_read_unlock();
```

Is the `rcu_dereference()` primitive in an RCU Classic or an RCU Sched critical section? What would you have to do to figure this out?

Back to Quick Quiz 9.