Read-Copy Update (RCU) for C++

ISO/IEC JTC1 SC22 WG21 P0279R1 - 2016-08-25

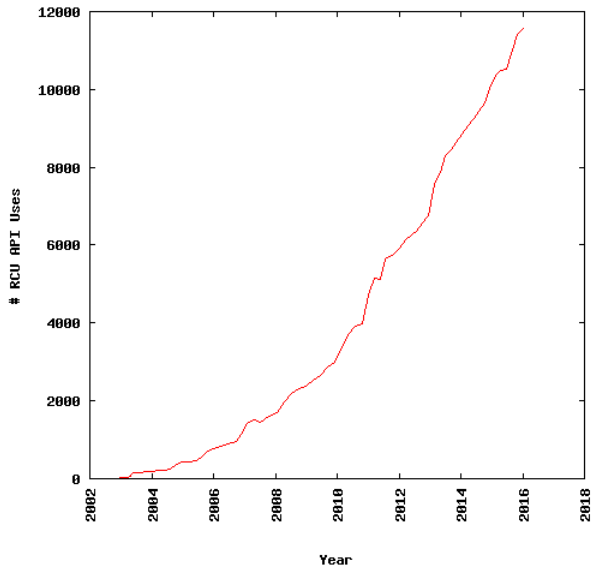
Paul E. McKenney, paulmck@linux.vnet.ibm.com
TBD

History

This document is a revision of P0279R0, revised based on discussions at the 2016 Jacksonville meeting. P0279R0 is in turn a follow-on to N4483, based on presentation and discussions at the 2015 Kona meeting.

Introduction

RCU has seen increasingly heavy use within the Linux kernel, as can be seen in the following graph, where it is most frequently used as a high-performance and highly scalable replacement for reader-writer locking. It has also seen significant uptake in some userspace applications via the userspace RCU library, which is available on many recent Linux distributions, and has also been tested on a number of versions of FreeBSD as well as on Cygwin. One example userspace use is the solution to the Issaquah Challenge, see slides 63-66 for performance and scalability information.



This document gives a brief introduction to RCU and describes one way that it might be incorporated into the C and C++ standards.

What Is RCU?

RCU provides guarantees and desiderata. A given implementation must provide the guarantees to qualify as an RCU implementation, and should also provide the desiderata if it is to be taken seriously.

RCU Guarantees

RCU provides a *grace-period guarantee*, a *publish-subscribe guarantee*, and *memory-ordering guarantees*. The following paragraphs provide a quick overview of these guarantees, with more detail available in the Additional References.

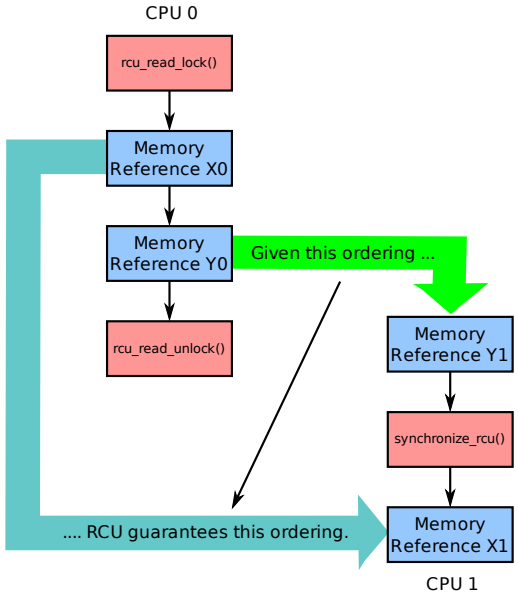
The grace-period guarantee allows updaters to wait for pre-existing RCU readers. The “pre-existing” phrase is important: RCU is not obligated to wait for readers that start after the beginning of the grace period. Grace periods are frequently used to privatize data elements that have been removed from a linked data structure. The key point is that once a given data element has been removed, only pre-existing readers can have access to it. Therefore, removing the element and then waiting for a grace period guarantees that no readers can possibly still have access to that element: Old readers have completed, and new readers never did have a path to the removed element. Thus, after the grace period completes, the removed element can safely be freed, even in the presence of concurrent

readers. In this way, grace periods greatly simplify maintenance of data structures subject to concurrent lookup and deletion.

RCU readers are delimited by `rcu_read_lock()` and `rcu_read_unlock()`, which may be nested. RCU updaters wait for all pre-existing readers by invoking `synchronize_rcu()`, which blocks for at least one grace period, that is, until all such readers have completed. The asynchronous counterpart to `synchronize_rcu()` is `call_rcu()`, which invokes the specified function after a grace period has elapsed.

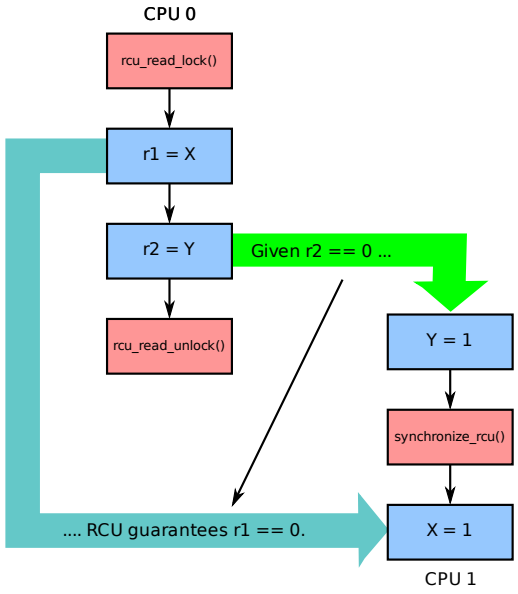
The publish-subscribe guarantee is used to simplify insertion into a linked data structure that is subject to concurrent lookup. For this purpose, RCU provides API members that incorporate any compiler directives and memory-barrier instructions that might be required to ensure that when a reader encounters a newly inserted data element, that reader will be guaranteed to see any initialization that might have been applied to that data element prior to its insertion. Publication is carried out via `rcu_assign_pointer()`, which could be implemented using a `memory_order_release` store, and subscription is carried out via `rcu_dereference()` which could be implemented using a `memory_order_consume` load, or could be if a high-quality implementation of `memory_order_consume` existed.

The memory-ordering guarantee is a direct consequence of the grace-period guarantee, but is well worth discussing separately. The general principle is illustrated in the following figure:



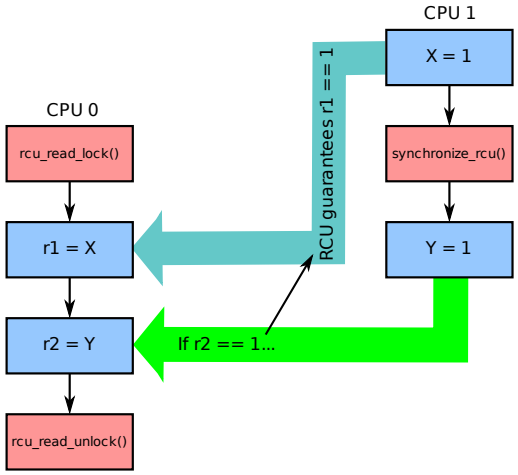
In you can see from the figure, if any reference in a given RCU read-side critical section precedes a given grace period, then all references in that RCU read-side critical section are guaranteed to precede any reference following that grace period.

The next figure illustrates the same principle, but uses relaxed assignments, and guarantees `r2 != 0 || r1 == 0`.



These ordering guarantees also operate in reverse, so that if any reference in a given RCU read-side critical section follows any

reference following a given grace period, then all references in that RCU read-side critical section are guaranteed to follow any reference preceding that grace period. In particular, given the relaxed accesses in the following figure, it is guaranteed that `r1 == 1 || r2 != 1`.



It is important to note that these ordering guarantees apply to *all* accesses in the RCU read-side critical section, regardless of ordering. As expected, the following litmus test guarantees `r1 != 1 || r2 == 1`:

```
CPU 0                   CPU 1

rcu_read_lock();        Y = 1;
r1 = X;                 synchronize_rcu();
r2 = Y;                 X = 1;
rcu_read_unlock();
```

However, the following litmus test also provides this exact same guarantee, despite the loads in the RCU read-side critical section having been interchanged:

```
CPU 0                   CPU 1

rcu_read_lock();        Y = 1;
r2 = Y;                 synchronize_rcu();
r1 = X;                 X = 1;
rcu_read_unlock();
```

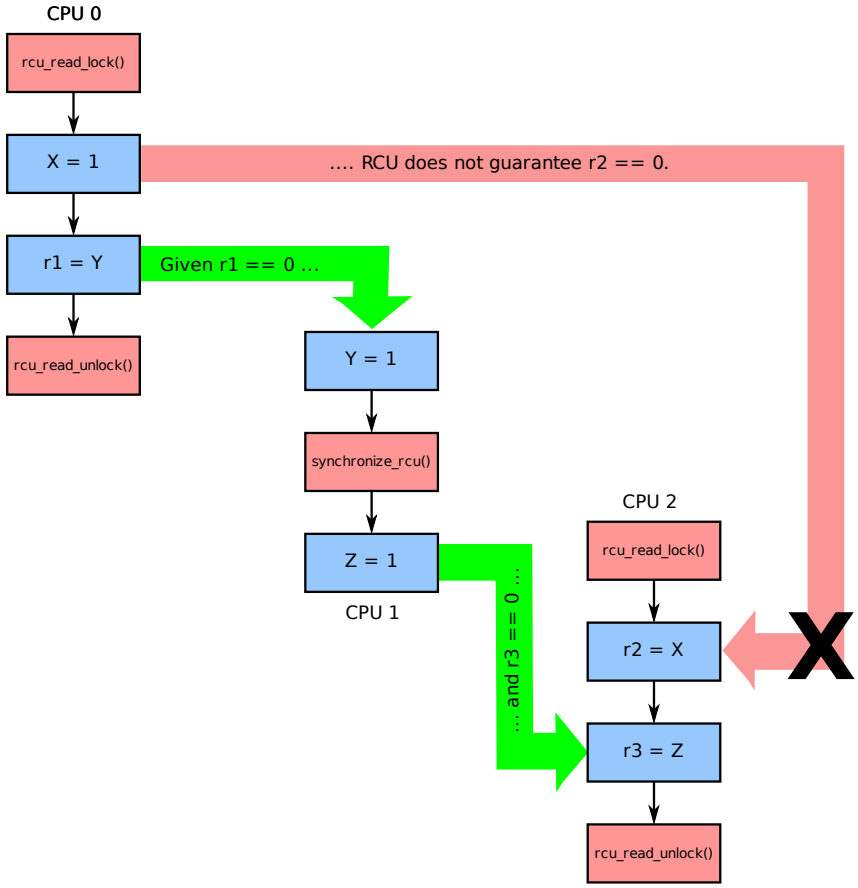
It is important to note that these guarantees are based on an interaction between the readers' `rcu_read_lock()` and `rcu_read_unlock()` on the one hand and the updater's `synchronize_rcu()` on the other. In particular, in the absence of `synchronize_rcu()`, `rcu_read_lock()` and `rcu_read_unlock()` offer no ordering guarantees whatsoever. For example, consider the following litmus test, again with `X` and `Y` both initially equal to zero:

```
CPU 0                   CPU 1

rcu_read_lock();        rcu_read_lock();
X = 1;                  r1 = Y;
rcu_read_unlock();      rcu_read_unlock();
rcu_read_lock();        rcu_read_lock();
Y = 1;                  r2 = X;
rcu_read_unlock();      rcu_read_unlock();
```

In this case, because `rcu_read_lock()` and `rcu_read_unlock()` offer no ordering guarantees, all four outcomes are possible for the values of `r1` and `r2`.

Finally, if any part of a given RCU read-side critical section is ordered before a given grace period, and if any part of some other RCU read-side critical section is ordered after that same grace period, then there are no guarantees. The accesses in CPU 0's RCU read-side critical section can be reordered down to the end of the grace period, and the accesses in CPU 2's RCU read-side critical section can be reordered up to the beginning of the grace period, so the accesses to `X` can overlap. This overlap could be prevented, if desired, by having CPU 1 execute a pair of back-to-back grace periods in place of the single grace period shown in the figure.



More details on RCU's memory-ordering guarantees may be found in the an LWN article entitled “The RCU-barrier menagerie”.

RCU Desiderata

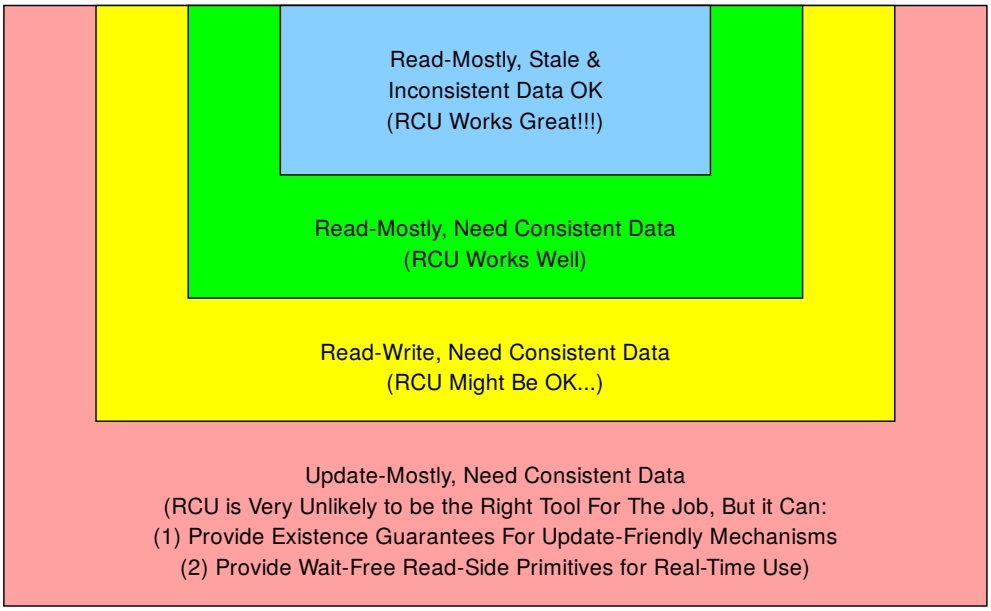
A high-quality RCU implementation will in addition satisfy the following desiderata:

1. RCU's read-side primitives should deterministic and extremely small overheads. Here, “extremely small” will typically rule out use of locks, atomic instructions, explicit memory-barrier instructions, and, in many cases, conditional branches. At a minimum, deterministic overhead should be provided to the highest-priority thread in a strict-priority environment. This desideratum helps avoid most deadlocks.
2. RCU's primitives, both read-side and update-side, should be unconditional. They should therefore avoid failure returns and retry operations. This desideratum helps avoid most livelocks and greatly simplifies use of these primitives.
3. Although RCU's update-side primitives are not required to have deterministic overheads, given a fair scheduler, it should not be possible to starve them, even given a massive influx of RCU read-side critical sections. Starvation should only occur in the presence of an unfair scheduler (for example, a fixed-priority scheduler) or in the presence of at least one RCU read-side critical section containing an infinite loop.
4. RCU read-side critical sections should be permitted to contain any operation, with the exception of those operations that wait, either directly or indirectly, for an RCU grace period to complete.
5. RCU read-side critical sections should be permitted to modify RCU-protected data structures, for example, by acquiring the update-side lock from within an RCU read-side critical section.
6. RCU's semantics and implementation should not be closely intertwined with those of the memory allocators.
7. Concurrent requests for an RCU grace period should be satisfied by a single RCU grace period. (Within the Linux kernel, it is not difficult to arrange for more than 1,000 requests to be satisfied by a single grace period.)

Meeting these desiderata greatly simplifies use of RCU, however, this list is not necessarily complete.

Where Is RCU Best Used?

When used properly, RCU can be a very powerful tool, however, it gains its power through specialization. The following figure gives a rough guide to where RCU may be profitably applied:



More information on how RCU is used in the Linux kernel may be found here and here.

RCU API

The minimal RCU API is straightforward:

1. `rcu_read_lock()`: Begin an RCU read-side critical section.
2. `rcu_read_unlock()`: End an RCU read-side critical section. RCU read-side critical sections may be nested.
3. `atomic_load_explicit(p, memory_order_consume)`: Fetch a pointer to an RCU-protected data element. This is `rcu_dereference()` in the Linux kernel and in the userspace RCU library.
4. `atomic_store_explicit(p, newp, memory_order_release)`: Store a pointer to an RCU-protected data element. This is `rcu_assign_pointer()` in the Linux kernel and in the userspace RCU library.
5. `synchronize_rcu()`: Wait for an RCU grace period to elapse. In other words, for every thread within an RCU read-side critical section at the beginning of `synchronize_rcu()`'s execution, wait for that thread to exit its RCU read-side critical section. Note that `synchronize_rcu()` need not wait for RCU read-side critical sections that were entered after the beginning of `synchronize_rcu()`'s execution. Note also that it is permissible for `synchronize_rcu()` to wait somewhat longer than needed.
6. `void call_rcu(struct rcu_head *head, void (*func)(struct rcu_head *head))`: After a grace period elapses, invoke `func(head)` (which in C++ could be a lambda). The `head` structure is normally embedded within the RCU-protected data element. Both the Linux kernel and usermode RCU implement `struct rcu_head` as a pair of pointers, one to hold `func` and the other to link a series of such structures together. The *RCU callback function* `func` is responsible for mapping from the address of `head` to the beginning of the enclosing structure. Both the Linux kernel and the userspace RCU library use an address-arithmetic macro named `container_of()` to do this mapping.

For a view of how elaborate an RCU API can become, see the 2014 edition of the Linux-kernel RCU API or the user-space RCU API. Much of the added complexity is due to the addition of some RCU-protected data structures.

Alternative API Choices

RCU has heretofore been used almost exclusively in C programs, so it is likely that some adjustment of API is desirable for C++ use. Here are some likely adjustments:

1. Scoped readers. This is similar to scoped locks, where an object invokes `rcu_read_lock()` in its constructor and `rcu_read_unlock()` in its destructor.
2. In the C-language userspace RCU library, there is an additional `rcu_register_thread()` function that must be called by each thread prior to its first invocation of `rcu_read_lock()`, as well as an `rcu_unregister_thread)()` that must be called by each thread following its last invocation of `rcu_read_unlock()`. In C++, these might be implicit in the constructors of an RCU-reading subclass of std::thread.
3. In the C-language userspace RCU library, there is an additional `rcu_thread_offline()` that is invoked before a given reader thread sleeps for a long time, and and additional `rcu_thread_online()` that is invoke after awakening. In C++, these might be used in scoped extended quiescent state, where an object invokes `rcu_thread_offline()` in its constructor and `rcu_thread_online()` in its destructor. These functions are required by some implementations to allow RCU reader threads to block for extended time periods.
4. The relationship of RCU to other execution agents needs to be worked out.
5. For C++, `call_rcu()` might also be a member function of the `rcu_head` structure, assuming that `rcu_head` is converted from a `struct` to a `class`. *(WG14 liason issue.)*

A C++ standardization of RCU should of course follow existing practice. Given that most C++ code appears to be in userspace applications and libraries, the best example to follow is the C-language userspace RCU library. This library has seen substantial use in production and has been subjected to proofs of correctness (here and here). A reasonable strategy is therefore to wrap this library's

C-language API in a C++ class library.

Given that object-oriented nature of C++, it is only natural to ask if a C++ specification of RCU should feature “domains” that are tied to specific objects. And within the Linux kernel, there is in fact a variant of RCU called “sleepable RCU” (SRCU). In this variant, `rcu_read_lock()` is replaced with `srcu_read_lock()`, which returns an integer. This integer is passed to `srcu_read_unlock()`, which replaces `rcu_read_unlock()`. However, within the Linux kernel, the default RCU implementation is used more than 20 times more heavily than is SRCU. Futhermore, SRCU tends to be used not because of the ability to provide separate domains, but rather because it can be used in contexts where the default RCU implementation cannot be, for example, from idle CPUs, offline CPUs, and from earlier points in exception handling. These contexts do not exist for userspace applications and libraries. The other (rare!) use of SRCU is for cases where readers must block for extended time periods, in which case the separate domains prevent those readers from delaying unrelated updates.

In addition, there are a number of reasons why separate RCU domains is a particularly bad idea:

1. Production-quality RCU implementations promote performance and scalability by batching updates, so that multiple concurrent `synchronize_rcu()` and `call_rcu()` invocations are serviced by a single grace-period computation. In fact, within the Linux kernel, an undemanding workload can easily have a single grace-period computation serve thousands of `synchronize_rcu()` and `call_rcu()` requests. This batching optimization thus reduces the per-updated overhead of the grace-period computation down to nearly zero. Splitting RCU up into domains defeats this important batching optimization.
2. RCU uses a small fixed quantity of TLS. A production-quality per-domain RCU implementation would instead use a small fixed quantity of TLS *per domain*, which for a large application could easily translate to a very large quantity of TLS. This TLS issue is especially pressing given that domains could be dynamically allocated, as is in fact the case with Linux-kernel SRCU domains. The global RCU implementation is therefore much more amenable to use by lightweight execution agents.
3. In stark contrast to many other synchronization mechanisms, the read-side scalability of RCU is not improved by domains. The common-case overhead of RCU readers is deterministic in that a fixed sequence of instructions is executed, and this fixed sequence is completely independent of the number of CPUs and of the read-side offered load.

In short, experience indicates that the per-domain style does not help and in fact causes significant problems.

Nevertheless, here is the C-language API for SRCU, which is the Linux kernel's per-domain RCU implementation:

1. `int srcu_read_lock(struct srcu_struct *sp)`: Begin an SRCU read-side critical section with respect to the specified domain.
2. `void srcu_read_unlock(struct srcu_struct *sp, int idx)`: End an SRCU read-side critical section. The `idx` parameter takes the return value from `srcu_read_lock()`.
3. `synchronize_srcu(struct srcu_struct *sp)`: Wait for completion of all pre-existing SRCU readers belonging to the specified domain.
4. `void call_srcu(struct srcu_struct *sp, struct rcu_head *head, void (*func)(struct rcu_head *head))`: Some time after the completion of all pre-existing SRCU readers belonging to the specified domain.

These could of course be member functions of `srcu_struct`, assuming this is implemented as a class. *(WG14 liason issue.)*

The `atomic_load_explicit()` and `atomic_store_explicit()` are used by SRCU in exactly the same was as they are by RCU.

However, the initial proposal contains only a global RCU, as implemented by the userspace RCU library.

Implementation Alternatives

There are a surprisingly large number of plausible RCU implementations, and more are being created all the time. The best guide for userspace RCU implementations are in the userspace RCU library, and the best tutorial for these implementations is Appendix D of the supplementary materials to “User-Level Implementations of Read-Copy Update”. Simpler (but less useful) implementations may be found in Section 9.3.5 of “Is Parallel Programming Hard, And, If So, What Can You Do About It?”.

In addition, there is a large number of alternative implementations in the Linux kernel.

As noted earlier, C++ should start with the userspace RCU library.

Additional References

There is a large body of background information on RCU, including the following:

1. The 2013 CACM paper entitled “Structured deferral: synchronization via procrastination” provides a good overview of the motivation and use of RCU. The CACM paper may be found here, and its ACM Queue predecessor here.
2. The 2013 IEEE TPDS paper entitled “User-Level Implementations of Read-Copy Update” provides some conceptual background for RCU, performance comparisons, and implementations. The main paper is here (with pre-publication draft here), and the supplementary materials are here.
3. Section 9.3.5 of “Is Parallel Programming Hard, And, If So, What Can You Do About It?” provides a list of low-quality but simple RCU implementations. Section 9.3 covers RCU in general, and Chapters 10 and 13 cover some example uses of RCU.
4. The userspace RCU library is available here, and also as part of many Linux distributions.
5. The full user-space RCU API is here.
6. The full Linux-kernel RCU API is here.
7. Descriptions of how RCU is typically used in the Linux kernel may be found here and here.
8. The Issaquah Challenge shows how RCU may be used to help orchestrate complex updates, and the most recent presentation is here.
9. RCU's memory-ordering guarantees are described here.
10. A list of requirements for the Linux-kernel RCU implementation may be found in the LWN series here, here, and here. It is quite fortunate that most of these requirements are specific to the complexities of the Linux kernel environment, however, the full list

can be quite illuminating.
11. The classic introduction to RCU, still used in some university coursework, is here.
12. Quite a bit more RCU-related material is available here.