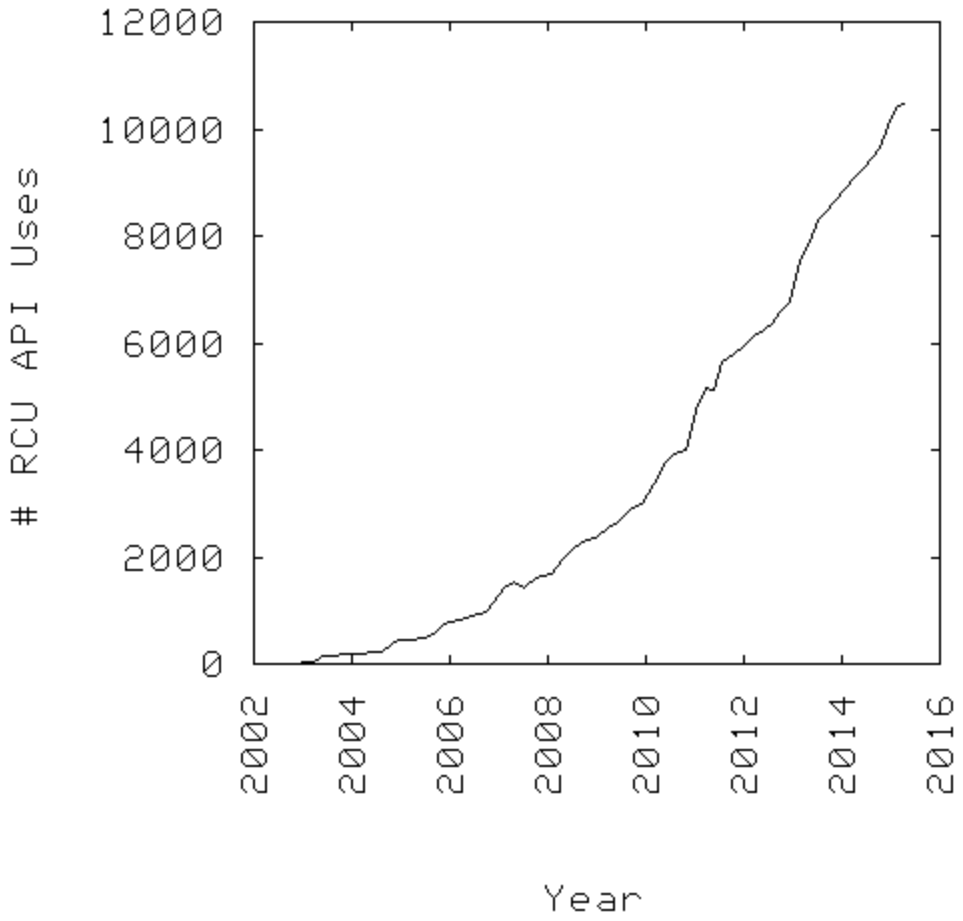
# Read-Copy Update (RCU) for C++

ISO/IEC JTC1 SC22 WG21 N4483 - 2015-04-14

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

## 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

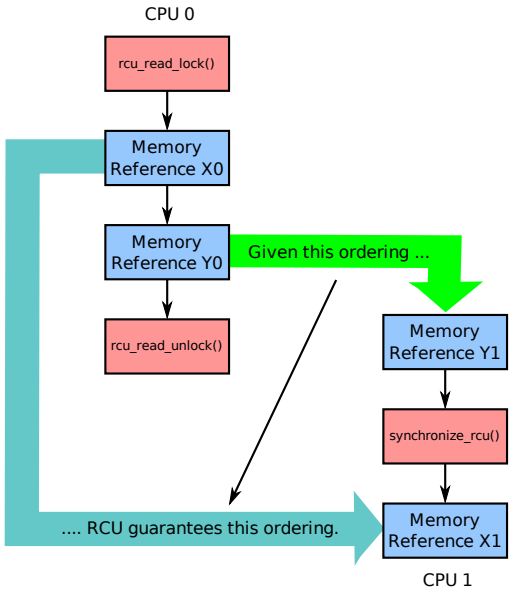
*memory-ordering guarantees*. The following paragraphs provide a quick overview of these guarantees, with more detail available in the Additional References.

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.

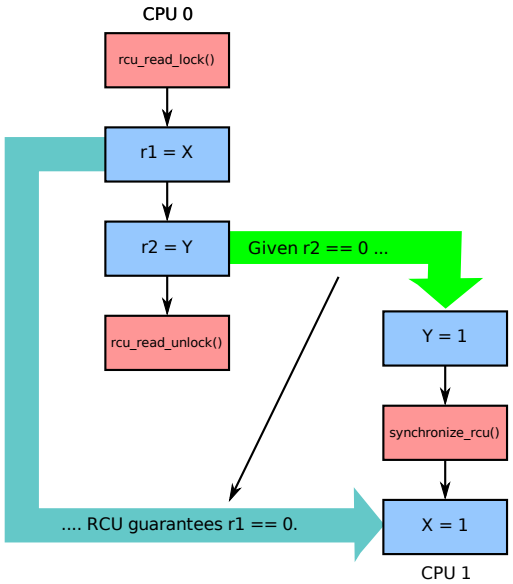
The publish-subscribe guarantee is used to simplify insertion into a linked data structure that is subject to concurrent lookup. For this purpose, RCU provides API members that incorporate any compiler directives and memory-barrier instructions that might be required to ensure that when a reader encounters 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:



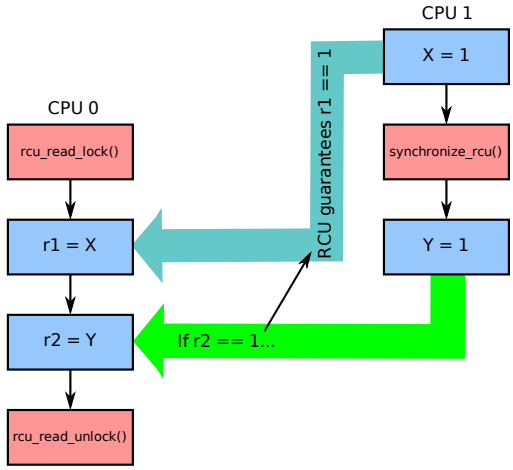

In you can see from the figure, if any reference in a given RCU read-side critical section preceds a given grace period, then all references in that RCU read-side critical section are guaranteed to precede any reference following that grace period.

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:

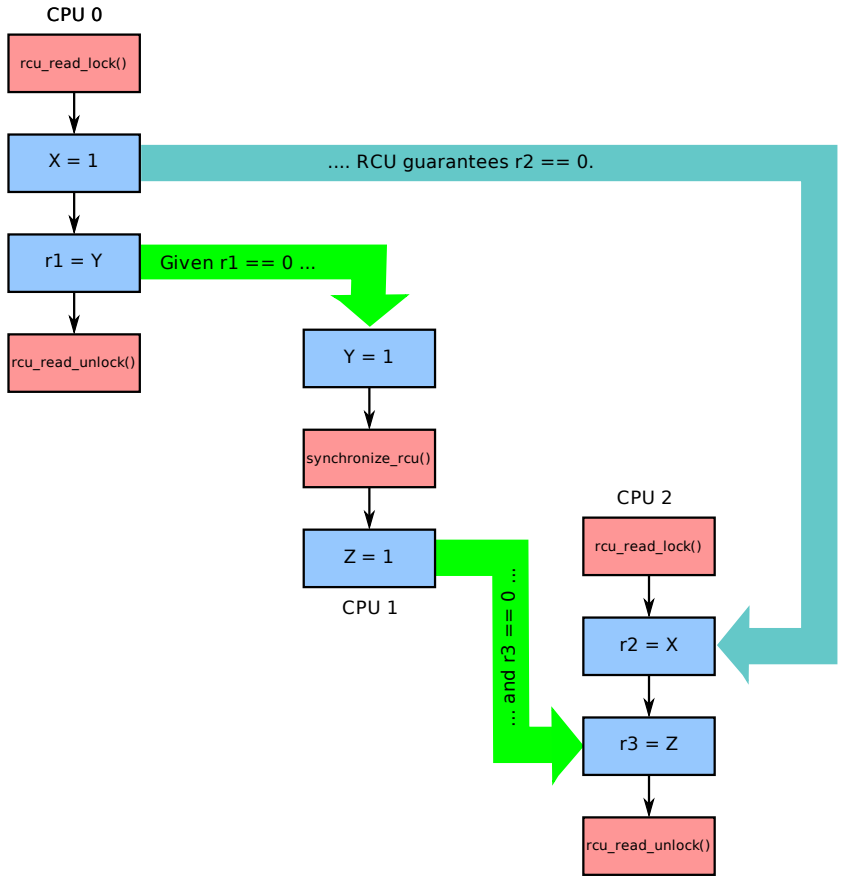
```
CPU 0                   CPU 1

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 the entirety of the first RCU read-side critical section is ordered before the entirety of the second RCU read-side critical section, as shown below:




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

## RCU Desiderata

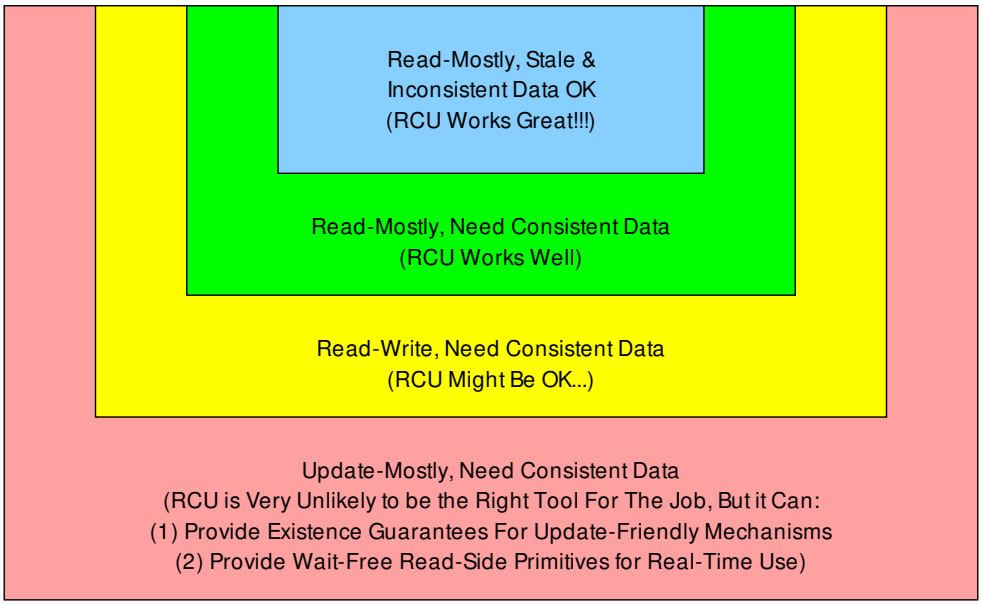
A high-quality RCU implementation will in addition satisfy the following desiderata:

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)`. 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.

# 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?". However, there are two basic styles, the global RCU approach described below (and implemented in the userspace RCU library) and the domain-based *sleepable RCU (SRCU)*, which includes an `srcu_struct` structure that defines a given SRCU domain.

The SRCU approach is probably more in keeping with the object-oriented approach, however, it is worth noting that a high-quality SRCU implementation requires a separate set of thread-local variables for each and every instance of the `srcu_struct` structure—and it is easy to imagine that these structures might be dynamically allocated. We should start with the global RCU API described above.

# 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. The classic introduction to RCU, still used in some university coursework, is here.
11. Lots more RCU-related material here.