P0461R2: Proposed RCU C++ API

Doc. No.: WG21/P0461R2
Date: 2017-10-15
Reply to: Paul E. McKenney, Maged Michael, Michael Wong,
Isabella Muerte, Arthur O'Dwyer, David Hollman,
Andrew Hunter, Geoffrey Romer, and Lance Roy
Email: paulmck@linux.vnet.ibm.com, maged.michael@gmail.com,
fraggamuffin@gmail.com, isabella.muerte@mnmlstc.com,
arthur.j.odwyer@gmail.com, dshollm@sandia.gov,
ahh@google.com, gromer@google.com, and ldr709@gmail.com

October 15, 2017

This document is based on WG21/P0279R1 combined with feedback at the 2015 Kona, 2016 Jacksonville, 2016 Issaquah, 2017 Kona, and 2017 Toronto meetings, which most notably called for a C++-style method of handling different RCU implementations or domains within a single translation unit, and which also contains useful background material and references. Later feedback and evaluation of existing RCU implementations permitted significant simplification. These simplifications eliminated domains, and this elimination will likely be revisited, certainly before any merge into the International Standard. Unlike WG21/P0279R1, which simply introduced RCU's C-language practice, this document presents proposals for C++-style RCU APIs. At present, it appears that these are not conflicting proposals, but rather ways of handling different C++ use cases resulting from inheritance, templates, and different levels of memory pressure. This document also incorporates content from WG21/P0232R0[9].

Note that this proposal is related to the hazard-pointer proposal in that both proposals defer destructive actions such as reclamation until all readers have completed. See P0233R3, which updates "P0233R2: Hazard Pointers: Safe Resource Reclamation for Optimistic Concurrency" at `http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2016/p0233r2.pdf`.

This proposal is also related to "P0561R0 An RAII Interface for Deferred Reclamation" at `http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2017/p0561r0.html`. This RAII proposal has replaced the C++ wrapper APIs that appeared in earlier revisions of this document. There have been other proposals for C++ RCU APIs [4].

Note also that a redefinition of the infamous `memory_order_consume` is the subject of two separate papers:

1. P0190R3, which updates "P0190R2: Proposal for New `memory_order_consume` Definition", `http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2016/p0190r2.pdf`.

2. P0462R1, which updates "P0462R0: Marking `memory_order_consume` Dependency Chains", `http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2016/p0462r0.pdf`. Note however that P0462R1 is expected to be obsoleted by an alternative proposal by JF Bastien that has seen production use.

Draft wording for this proposal may be found in the new working paper "P0566R2: Proposed Wording for Concurrent Data Structures: Hazard Pointer and Read-Copy-Update (RCU)".

A detailed change log appears starting on page 17.

1 Introduction

This document proposes C++ APIs for read-copy update (RCU). For more information on RCU, including RCU semantics, see WG21/P0462R0 ("Marking `memory_order_consume` Dependency Chains"), WG21/P0279R1 ("Read-Copy Update (RCU) for C++"), WG21/P0190R2 ("Proposal for New `memory_order_consume` Definition"), WG21/P0098R1 ("Towards Implementation and Use of `memory_order_consume`"), and WG21/P0750R0 ("Consume").[1]

Specifically, this document proposes `rcu_reader` (Figure 2), `rcu_obj_base` (Figure 3), and several free functions (Figures 3 and 4).

Section 2 presents the base (C-style) RCU API, Section 3 presents a proposal for scoped RCU readers, Section 4 presents proposals for handling of RCU callbacks, Section 6 presents a table comparing reference counting, hazard pointers, and RCU, and finally Section 7 presents a summary. This is followed by an informational-only appendix that shows some alternatives that were considered and rejected.

A fully functional implementation is available on github: `https://github.com/paulmckrcu/RCUCPPbindings`. See the Test/paulmck directory: Other directories contain other alternatives that were considered and rejected, although the efforts of their respective authors are deeply appreciated. Their work was a critically important part of the learning process leading to the solution presented in this document.

2 Existing C-Language RCU API

Figure 1 shows the existing C-language RCU API as provided by implementations such as userspace RCU [2, 6]. This API is provided for compatibility with existing C-language practice as well as to provide the highest performance for

[1] WG21/P0750R0 expands on WG21/P0462R0 and WG21/P0190R2.

```
 1 void rcu_read_lock();
 2 void rcu_read_unlock();
 3 void synchronize_rcu();
 4 void call_rcu(struct rcu_head *rhp,
 5               void (*cbf)(struct rcu_head *rhp));
 6 void rcu_barrier();
 7 void rcu_register_thread();
 8 void rcu_unregister_thread();
 9 void rcu_quiescent_state();
10 void rcu_thread_offline();
11 void rcu_thread_online();
12 void defer_rcu(void (*fct)(void *p), void *p);
```

Figure 1: Existing C-Language RCU API

fast-path code. As we will see in Section 2.2, the C++ user will not need to be concerned with most of these API members.

2.1 Existing C-Language RCU API Detailed Description

Lines 1 and 2 show `rcu_read_lock()` and `rcu_read_unlock()`, which mark the beginning and the end, respectively, of an *RCU read-side critical section*. These primitives may be nested, and matching `rcu_read_lock()` and `rcu_read_unlock()` calls need not be in the same scope. (That said, it is good practice to place them in the same scope in cases where the entire critical section fits comfortably into one scope.)

Line 3 shows `synchronize_rcu()`, which waits for any pre-existing RCU read-side critical sections to complete. The period of time that `synchronize_rcu()` is required to wait is called a *grace period*. Note that a given call to `synchronize_rcu()` is *not* required to wait for critical sections that start later.

Lines 4 and 5 show `call_rcu()`, which, after a subsequent grace period elapses, causes the `cbf(rhp)` *RCU callback function* to be invoked. Thus, `call_rcu()` is the asynchronous counterpart to `synchronize_rcu()`. In most cases, `synchronize_rcu()` is easier to use, however, `call_rcu()` has the benefit of moving the grace-period delay off of the updater's critical path. Use of `call_rcu()` is thus critically important for good performance of update-heavy workloads, as has been demonstrated by implementation experience across a variety of environments [7].

Note that although `call_rcu()`'s callbacks are guaranteed not to be invoked too early, there is no guarantee that their execution won't be deferred for a considerable time. This can be a problem if a given program requires that all outstanding RCU callbacks be invoked before that program terminates. The `rcu_barrier()` function shown on line 6 is intended for this situation. This function blocks until all callbacks corresponding to previous `call_rcu()` invocations have been invoked and also until after those invocations have returned. Therefore, taking the following steps just before terminating a program will guarantee that all callbacks have completed:

1. Take whatever steps are required to ensure that there are no further in-

vocations of `call_rcu()`.

2. Invoke `rcu_barrier()`.

Carrying out this procedure just prior to program termination can be very helpful for avoiding false positives when using tools such as valgrind.

Many RCU implementations require that every thread announce itself to RCU prior to entering the first RCU read-side critical section, and to announce its departure after exiting the last RCU read-side critical section. These tasks are carried out via the `rcu_register_thread()` and `rcu_unregister_thread()`, respectively.

The implementations of RCU that feature the most aggressive implementations of `rcu_read_lock()` and `rcu_read_unlock()` require that each thread periodically pass through a *quiescent state*, which is announced to RCU using `rcu_quiescent_state()`. A thread in a quiescent state is guaranteed not to be in an RCU read-side critical section. Threads can also announce entry into and exit from *extended quiescent states*, for example, before and after blocking system calls, using `rcu_thread_offline()` and `rcu_thread_online()`.

Finally, `defer_rcu()` can be thought of as a non-intrusive variant of `call_rcu()` that maintains an array of references to memory awaiting callback invocation, as opposed to the traditional implementations of `call_rcu()`, which maintain references in a linked list.

2.2 Existing C-Language RCU API and C++

Because of recent advances in both userspace RCU and the Linux kernel's support for userspace RCU, C++ users will not need to be concerned with most of the C-language API members shown in Figure 1.

Both `rcu_read_lock()` and `rcu_read_unlock()` are encapsulated in a C++ RAII class named `rcu_reader`. The `synchronize_rcu()` function is exposed via the `std::synchronize_rcu()` free function, and the `rcu_barrier()` free function is exposed via the `std::rcu_barrier()` free function. The `call_rcu()` function is exposed via the `std::rcu_obj_base<T,D>::retire()` member function, and a non-intrusive variant of `call_rcu()` is exposed via the `std::rcu_retire()` templated free function.

We expect that `rcu_register_thread()` and `rcu_unregister_thread()` will be buried into the thread-creation and thread-exit portions of the standard library.

Use of the `sys_membarrier()` RCU implementation from the userspace RCU library allows `rcu_quiescent_state()`, `rcu_thread_offline()`, and `rcu_thread_online()` to be dispensed with.

Of course, implementation experience and use may result in changes to the C++ API as well as to the userspace RCU library. For example, the possibility of RCU domains is discussed in Appendix A, and a few historical methods of handling retire are shown in Appendix B.

```
1   class std::rcu_reader {
2   public:
3     rcu_reader() noexcept;
4     rcu_reader(std::defer_lock_t) noexcept;
5     rcu_reader(const rcu_reader &) = delete;
6     rcu_reader(rcu_reader &&other) noexcept;
7     rcu_reader& operator=(const rcu_reader&) = delete;
8     rcu_reader& operator=(rcu_reader&& other) noexcept;
9     ~rcu_reader() noexcept;
10    void swap(rcu_reader& other) noexcept;
11    void lock() noexcept;
12    void unlock() noexcept;
13  };
```

Figure 2: RAII RCU Readers

3 RAII RCU Readers

The `rcu_reader` class shown in Figure 2 may be used for RAII RCU read-side critical sections, and satisfies the requirements of `BasicLockable`. An argumentless constructor enters an RCU read-side critical section, a constructor with an argument of type `defer_lock_t` defers entering a critical section until a later call to a `std::rcu_reader::lock()` member function, and a constructor taking another `rcu_reader` instance transfers the RCU read-side critical section from that instance to the newly constructed instance. The assignment operator may be used to transfer an RCU read-side critical section from another `rcu_reader` instance to an already-constructed instance.

This class is intended to be used in a manner similar to `std::lock_guard`, so the destructor exits the RCU read-side critical section.

This implementation is movable but not copyable, which enables an RCU read-side critical section's scope to be arbitrarily extended across function boundaries via `std::forward` and `std::move`. The `std::rcu_reader::swap()` member function swaps the roles of a pair of `rcu_reader` instances in the expected manner. Finally, the `std::rcu_reader::lock()` member function enters an RCU read-side critical section and the `std::rcu_reader::unlock()` member function exits its critical section. Invoking `std::rcu_reader::lock()` on an instance already corresponding to an RCU read-side critical section and invoking `std::rcu_reader::unlock()` on an instance not corresponding to an RCU read-side critical section results in undefined behavior.

4 Retiring RCU-Protected Objects

The traditional C-language RCU callback uses address arithmetic to map from the `rcu_head` structure to the enclosing struct, for example, via the `container_of()` macro. Of course, this approach also works for C++, but this section describes a more palatable approach that is quite similar to that proposed for hazard pointers. Other historical approaches may be found in Sections B.1 and B.2.

4.1 Retiring: Implementation Experience

Known implementation experience either (1) links newly retired objects together or (2) references them from a set of vectors. Userspace RCU provides API members that do both: `call_rcu()` links retired objects and `defer_rcu()` references them from a vector.

The main advantage of the linking approach is that the retirement process can easily and efficiently ensure that no allocations occur on the retirement path, which is often part of the free path, thus avoiding out-of-memory deadlocks. Such deadlocks could otherwise occur if there was no memory available, and that lack of memory prevented retirement, and in turn preventing freeing.

However, many applications avoid out-of-memory deadlocks by overprovisioning memory. For these applications, there is little or no reason to avoid allocating memory during the process of retiring objects. Such applications might choose to occasionally allocate vectors to keep track of newly retired objects. And this choice brings additional benefits:

1. The greater cache locality of vectors has been shown to improve the efficiency of the retirement process [3].

2. Removing the need to associate retirement-time objects with the RCU-protected object also removes any restrictions in type and inheritance. For but one example, the vector approach allows retiring an RCU-protected `string` instance.

3. If retirement is infrequent, but there is a very large number of RCU-protected objects in existence, the vector approach can offer a smaller memory footprint.

There are a number of ways to manage the vectors:

1. Fixed per-thread allocation, as is done for the `defer_rcu()` interface in userspace RCU, with a `synchronize_rcu()` invoked internally to `defer_rcu()` when the vector fills. This performs quite well [3], but is not appropriate in environments where it is necessary to retire objects from within RCU read-side critical sections.

2. Provide an initial set of per-thread vectors, allocating more as these vectors fill, and enqueuing them for reuse after their element's deleters have been invoked.

3. As above, but also using some mechanism to free vectors that are not being used.

4. Allocate the retirement vectors as needed during memory allocation, ensuring that there are enough slots to handle sudden retirement of all instances of all dynamically allocated objects.[2] The amount of retirement-path allocation can of course be reduced by enqueuing them for reuse after

[2] Kudos to Lance Roy for pointing out this possibility.

```
1 template<typename T, typename D = std::default_delete<T>>
2 class std::rcu_obj_base {
3 public:
4         void retire(D d = {}) noexcept;
5 };
6
7 template<typename T, typename D = std::default_delete<T>>
8 void std::rcu_retire(T *p, D d = {});
```

Figure 3: Retiring RCU-Protected Objects

their element's deleters have been invoked. However, please note that this approach assumes that retirement leads to freeing, and that there are some rare but important algorithms for which this is not the case,[3] with the Linux kernel's `rcu_sync` mechanism perhaps being the most prominent. The key point is that RCU's function is waiting for readers, not necessarily reclaiming memory. Furthermore, C++ allows special-purpose memory allocators to be created easily, and it is not likely to be practical to require all of them to interact with RCU.

We expect further implementation experience to uncover additional strategies for tracking newly retired objects. In particular, we expect to see hybrid schemes that make use of per-object space via the `retire()` member function and that use vectors to track object passed to the `rcu_retire()` free function.

4.2 Retiring: Proposed C++ APIs

Bowing to both types of implementation experience, we propose an API that lends itself to linking retired objects and an API that lends itself to referencing retired objects via vectors. Note that a vector-based retirement implementation can simply ignore any per-object storage, and a linked-list retirement implementation can simply allocate the required storage on each retire. Higher-quality implementations can of course provide further optimizations.

The `rcu_obj_base` class provides a `retire()` method that takes a deleter, as shown in Figure 3. This class contains any storage required by that deleter, so that the implementation of `std::rcu_obj_base<T,D>::retire()` never needs to allocate memory on the retire/free path. The deleter's `operator()` is invoked after a grace period. The deleter type defaults to `std::default_delete<T>`, but one could also use a custom functor class with an `operator()` that carries out teardown actions before freeing the object, or a raw function pointer type such as `void(*)(T*)`. Given sufficient type erasure and reconstitution, the `call_rcu()` C-language free function from the userspace RCU library can be used to implement `retire()`.

We recommend avoiding deleter types such as `std::function<void(T*)>` (and also any other type requiring memory allocation) because allocating memory on the free path can result in out-of-memory deadlocks, but we nevertheless

[3] Especially when using `synchronize_rcu()` instead of `call_rcu()` [12, 1, 5, 10, 8].

```
1 void synchronize_rcu() noexcept;
2 void rcu_barrier() noexcept;
```

Figure 4: RCU Updaters

recognize that C++ applications that assume ample memory might use such deleters for convenience. However, such users are better served by the `std::rcu_retire()` templated free function shown on lines 7-8 of Figure 3. Although this function can allocate on the retire/free path, high-quality implementations will take steps to ensure that such allocation is very rare, for example, by pre-allocating sufficient storage to avoid such allocation in the common case. Simple implementations can instead provide a trivial `std::rcu_retire()` function that is a thin wrapper around `std::rcu_obj_base::retire()`.

Implementation experience has shown that high-quality implementations of `std::rcu_retire()` can achieve better cache locality than can high-quality implementations of `std::rcu_obj_base::retire()`, which results in better performance and scalability.[4] Users wishing the best performance and scalability will therefore tend to prefer `std::rcu_retire()` over `std::rcu_obj_base::retire()`.

Finally, `std::rcu_retire()` is non-intrusive. This means that (for example) an object of type `std::string` can be passed to `std::rcu_retire()`. In contrast, `std::rcu_obj_base::retire()` can only be passed types related to `std::rcu_obj_base`.

5 RCU Updaters

RCU updaters can use the free functions shown in Figure 4. The `<rcu>` header provides a `synchronize_rcu()` free function, which maps to the C-language `synchronize_rcu()` function, which waits for all pre-existing RCU readers to complete. This header also provides the `rcu_barrier()` free function, which maps to the C-language `rcu_barrier()` function, which waits for all pending `retire()` and `rcu_retire()` deleters to be invoked.

Implementation experience thus far indicates that both of these functions must scale well, but that it is OK for them to have significant latency. In fact, the significant latency helps reduce per-request overhead by allowing concurrent callers' requests to be satisfied by the same underlying operation. Use cases that might otherwise require `synchronize_rcu()` to have lower latency should instead use one of the retire APIs, as these retire APIs impose extremely low latencies on their callers.

The `rcu_barrier()` API is used when removing code or data structures that are used by the deleters passed to `call_rcu()`. In such cases, it is necessary to wait for any outstanding deleters to complete before freeing any code or data that those deleters might make accesses to.

[4] Note that this implementation experience is not limited to Google [3].

6 Hazard Pointers and RCU: Which to Use?

Table 1 provides a rough summary of the relative advantages of reference counting, RCU, and hazard pointers. Advantages are marked in bold with green background, or with a blue background for strong advantages.

Although reference counting has normally had quite limited capabilities and been quite tricky to apply for general linked data-structure traversal, given a double-pointer-width compare-and-swap instruction, it can work quite well, as shown in the "Reference Counting with DCAS" column.

As a rough rule of thumb, for best performance and scalability, you should use RCU for read-intensive workloads and hazard pointers for workloads that have significant update rates. As another rough rule of thumb, a significant update rate has updates as part of more than 10% of its operations. Reference counting with DCAS is well-suited for small systems and/or low read-side contention, and particularly on systems that have limited thread-local-storage capabilities. Both RCU and reference counting with DCAS allow unconditional reference acquisition.

Specialized workloads will have other considerations. For example, small-memory multiprocessor systems might be best-served by hazard pointers, while the read-mostly data structures in real-time systems might be best-served by RCU.

The relationship between the Hazard Pointers proposal and this RCU proposal is as follows:

1. The `hazptr_obj_base` class is analogous to `rcu_obj_base`.
2. There is no RCU counterpart to `hazptr_domain`, in part because RCU does not explicitly track read-side references to specific objects.
3. The private `hazptr_obj` class is analogous to the pre-existing `rcu_head` struct used in many RCU implementations. Because this class is an implementation detail, there is no need to have compatible names.
4. There is no RCU class analogous to `hazptr_rec` because RCU does not track (or need to track) references to individual RCU-protected objects.
5. There is no hazard pointers counterpart to the `rcu_reader` class. This is because hazard pointers does not have (or need) a counterpart to `rcu_read_lock()` and `rcu_read_unlock()`.
6. There is no hazard pointers counterpart to the `rcu_retire()` templated free function because no hazard-pointers user has expressed a need for it.

7 Summary

This paper demonstrates a way of creating C++ bindings for a C-language RCU implementation, which has been tested against the userspace RCU library.

	Reference Counting	Reference Counting with DCAS	RCU	Hazard Pointers
Unreclaimed objects	**Bounded**	**Bounded**	Unbounded	**Bounded**
Contention among readers	Can be very high	Can be very high	**No contention**	**No contention**
Traversal forward progress	Either blocking or lock-free with limited reclamation	**Lock free**	**Bounded population oblivious wait-free**	**Lock-free**
Reclamation forward progress *	Either blocking or lock-free with limited reclamation	**Lock free**	Blocking	**Bounded wait-free**
Traversal speed	Atomic read-modify-write updates	Atomic read-modify-write updates	**No or low overhead**	Store-load fence
Reference acquisition	**Unconditional**	**Unconditional**	**Unconditional**	Conditional
Automatic reclamation	**Yes**	**Yes**	No	No
Purpose of domains	N/A	N/A	Isolate long-latency readers	Limit contention, reduce space bounds, etc.

Table 1: Comparison of Deferred-Reclamation Mechanisms

* Does not include memory allocator, just the reclamation itself.

Specifically, this document proposes `rcu_reader` (Figure 2), `rcu_obj_base` (Figure 3), and a few free functions (Figures 3 and 4). We believe that these bindings are also appropriate for the type-oblivious C++ RCU implementations that information-hiding considerations are likely to favor.

Acknowledgments

We owe thanks to Pedro Ramalhete for his review and comments. We are grateful to Jim Wasko for his support of this effort.

This appendix contains historical proposals and directions. As such, it is strictly informational.

A RCU Domains

All the RCU implementations we are aware of started with a single domain, and many still support only one domain. Where domains were added, they were added for two reasons:

1. To allow alternative implementations with different design tradeoffs, for example, the "bottom half" variant of Linux-kernel RCU was added to allow Linux-kernel networking to better defend against network-based denial-of-service attacks.

2. In the case of Linux-kernel sleepable RCU (SRCU), to isolate different SRCU users from each other, so that one user having unusually long SRCU read-side critical sections would not delay the grace periods of other users.

It seems likely that the advent of a low-latency `sys_membarrier()` system call will permit almost all userspace RCU use cases to be addressed with a single implementation, so the first reason seems unlikely to apply to Linux systems in the short term. Note that Windows has had similar functionality for some time, and `sys_membarrier()` is quite simple, so can be easily added to other systems as needed. In the meantime, systems lacking `sys_membarrier()` can use one of the several userspace-RCU algorithms not relying on it.

As to the second reason, SRCU did not appear until some years after RCU was first accepted into the Linux kernel, and it was many years before SRCU was used at all heavily. Even today, there are more than an order of magnitude more uses of RCU than of SRCU. Furthermore, many of the recent SRCU uses were motivated by the accidental fact that SRCU grace periods are shorter than non-expedited RCU grace periods, and unlike expedited RCU grace periods, SRCU grace periods do not degrade real-time response.

In addition, initial uses of RCU in C++ appear to be targeted towards performance rather than abstraction (as expected), and performance use cases tend to focus on short RCU read-side critical sections and low update rates, each of which make domains less useful. Initial uses of RCU in C++ also seem to be focused solely on memory reclamation, and this use case can tolerate longer grace periods than can the more esoteric non-reclamation RCU use cases, again making RCU domains less useful.

Finally, use of domains can reduce RCU's update-side efficiency. To see this, note that production-quality RCU implementations can serve many thousands of grace-period requests with a single grace period [11], reducing the per-request grace-period overhead to nearly zero. Introducing domains increases the per-request overhead because each domain needs its own grace-period computation. Therefore, introduction of domains is an expensive step that should not be taken without a strong and urgent need.

In time, it is quite possible that a strong and urgent need for C++ RCU domains will appear. However, we should start with a very simple non-domain API for the initial TS, and drive any proposals for the addition of domains from actual experience with both uses and implementations.

Nevertheless, the following sections describe some historical proposals for RCU C++ domains. These were illustrated using the multiple userspace-RCU implementations, but were also intended to address the potential need for isolation of one user's readers from other users' grace periods.

A.1 Compile-Time Domain Selection

The quiescent-state based reclamation (QSBR) implementation is intended for standalone applications where the developers have full control over the entire application, and where extreme read-side performance and scalability is required. Applications use `#include "urcu-qsbr.hpp"` to select QSBR and `-lurcu -lurcu-qsbr` to link to it. These applications must use `rcu_register_thread()` and `rcu_unregister_thread()` to announce the coming and going of each thread that is to execute `rcu_read_lock()` and `rcu_read_unlock()`. They must also use `rcu_quiescent_state()`, `rcu_thread_offline()`, and `rcu_thread_online()` to announce quiescent states to RCU.

The memory-barrier implementation is intended for applications that can announce threads (again using `rcu_register_thread()` and `rcu_unregister_thread()`), but for which announcing quiescent states is impractical. Such applications use `#include "urcu-mb.hpp"` and `-lurcu-mb` to select the memory-barrier implementation. Such applications will incur the overhead of a full memory barrier in each call to `rcu_read_lock()` and `rcu_read_unlock()`.

The signal-based implementation represents a midpoint between the QSBR and memory-barrier implementations. Like the memory-barrier implementation, applications must announce threads, but need not announce quiescent states. On the one hand, readers are almost as fast as in the QSBR implementation, but on the other applications must give up a signal to RCU, by default `SIGUSR1`. Such applications use `#include "urcu-signal.hpp"` and `-lurcu-signal` to select signal-based RCU.

So-called "bullet-proof RCU" avoids the need to announce either threads or quiescent states, and is therefore the best choice for use by libraries that might well be linked with RCU-oblivious applications. The penalty is that `rcu_read_lock()` incurs both a memory barrier and a test and `rcu_read_unlock()` incurs a memory barrier. Such applications or libraries use `#include urcu-bp.hpp` and `-lurcu-bp`.

A.2 Run-Time Domain Selection

Figure 5 shows the abstract base class for runtime selection of RCU domains. Each domain creates a concrete subclass that implements its RCU APIs:

- Bullet-proof RCU: `class rcu_bp`

```
 1 class rcu_domain {
 2 public:
 3   constexpr explicit rcu_domain() noexcept { };
 4   rcu_domain(const rcu_domain&) = delete;
 5   rcu_domain(rcu_domain&&) = delete;
 6   rcu_domain& operator=(const rcu_domain&) = delete;
 7   rcu_domain& operator=(rcu_domain&&) = delete;
 8   virtual void register_thread() = 0;
 9   virtual void unregister_thread() = 0;
10   static constexpr bool register_thread_needed() { return true; }
11   virtual void quiescent_state() noexcept = 0;
12   virtual void thread_offline() noexcept = 0;
13   virtual void thread_online() noexcept = 0;
14   static constexpr bool quiescent_state_needed() { return false; }
15   virtual void read_lock() noexcept = 0;
16   virtual void read_unlock() noexcept = 0;
17   virtual void synchronize() noexcept = 0;
18   virtual void retire(rcu_head *rhp, void (*cbf)(rcu_head *rhp)) = 0;
19   virtual void barrier() noexcept = 0;
20 };
```

Figure 5: RCU Domain Base Class

- Memory-barrier RCU: `class rcu_mb`
- QSBR RCU: `class rcu_qsbr`
- Signal-based RCU: `class rcu_signal`

Of course, additional implementations of RCU may be constructed by deriving from `rcu_domain` and/or by implementing the API shown in Figure 1.

B Historical RCU-Protected Retirement Plans

This section lists alternative methods of retiring RCU-protected objects. These might prove helpful for some use cases, and can be implemented in terms of the intrusive approach described in Section 4.

B.1 Pointer To Enclosing Class (Informational Only)

If complex inheritance networks make inheriting from an `rcu_head` derived type impractical, one alternative is to maintain a pointer to the enclosing class as shown in Figure 6. This `rcu_head_ptr` class is included as a member of the RCU-protected class. The `rcu_head_ptr` class's pointer must be initialized, for example, in the RCU-protected class's constructor.

If the RCU-protected class is `foo` and the name of the `rcu_head_ptr` member function is `rh`, then `foo1.rh.retire(my_cb)` would cause the function `my_cb()` to be invoked after the end of a subsequent grace period. As with the previous classes, omitting the deleter results in the object being passed to `delete` and an `rcu_domain` object may be specified.

Please note that this section is informational only: This approach is *not* being proposed for standardization.

```
1   template<typename T>
2   class rcu_head_ptr: public rcu_head {
3   public:
4     rcu_head_ptr()
5     {
6       this->container_ptr = nullptr;
7     }
8
9     rcu_head_ptr(T *containing_class)
10    {
11      this->container_ptr = containing_class;
12    }
13
14    static void trampoline(rcu_head *rhp)
15    {
16      T *obj;
17      rcu_head_ptr<T> *rhdp;
18
19      rhdp = static_cast<rcu_head_ptr<T> *>(rhp);
20      obj = rhdp->container_ptr;
21      if (rhdp->callback_func)
22        rhdp->callback_func(obj);
23      else
24        delete obj;
25    }
26
27    void retire(void callback_func(T *obj) = nullptr)
28    {
29      this->callback_func = callback_func;
30      call_rcu(static_cast<rcu_head *>(this), trampoline);
31    }
32
33    void retire(class rcu_domain &rd,
34        void callback_func(T *obj) = nullptr)
35    {
36      this->callback_func = callback_func;
37      rd.retire(static_cast<rcu_head *>(this), trampoline);
38    }
39
40  private:
41    void (*callback_func)(T *obj);
42    T *container_ptr;
43  };
```

Figure 6: RCU Callbacks: Pointer (Informational Only)

```
1   template<typename T>
2   class rcu_head_container_of {
3   public:
4     static void set_field(const struct rcu_head T::*rh_field)
5     {
6       T t;
7       T *p = &t;
8
9       rh_offset = ((char *)&(p->*rh_field)) - (char *)p;
10    }
11
12    static T *enclosing_class(struct rcu_head *rhp)
13    {
14      return (T *)((char *)rhp - rh_offset);
15    }
16
17  private:
18    static inline size_t rh_offset;
19  };
20
21  template<typename T>
22  size_t rcu_head_container_of<T>::rh_offset;
```

Figure 7: RCU Callbacks: Address Arithmetic (Informational Only)

```
1 void my_cb(struct std::rcu_head *rhp)
2 {
3   struct foo *fp;
4
5   fp = std::rcu_head_container_of<struct foo>::enclosing_class(rhp);
6   std::cout << "Callback fp->a: " << fp->a << "\n";
7 }
```

Figure 8: RCU Callbacks: Address Arithmetic in Callback (Informational Only)

B.2 Address Arithmetic (Informational Only)

Figure 7 shows an approach that can be used if memory is at a premium and the inheritance techniques cannot be used. The `set_field()` method sets the offset of the `rcu_head_container_of` member within the enclosing RCU-protected structure, and the `enclosing_class()` member function applies that offset to translate a pointer to the `rcu_head_container_of` member to the enclosing RCU-protected structure.

This address arithmetic must be carried out in the callback function, as shown in Figure 8.

Please note that this section is informational only: This approach is *not* being proposed for standardization.

References

[1] Bhat, S. S. percpu_rwlock: Implement the core design of per-CPU reader-writer locks. `https://patchwork.kernel.org/patch/2157401/`, February 2014.

[2] Desnoyers, M. [RFC git tree] userspace RCU (urcu) for Linux. `http://liburcu.org`, February 2009.

[3] Desnoyers, M., McKenney, P. E., Stern, A., Dagenais, M. R., and Walpole, J. User-level implementations of read-copy update. *IEEE Transactions on Parallel and Distributed Systems 23* (2012), 375–382.

[4] Khiszinsky, M. Lock-free data structures. the inside. rcu. `https://kukuruku.co/post/lock-free-data-structures-the-inside-rcu/`, February 2015.

[5] Liu, R., Zhang, H., and Chen, H. Scalable read-mostly synchronization using passive reader-writer locks. In *2014 USENIX Annual Technical Conference (USENIX ATC 14)* (Philadelphia, PA, June 2014), USENIX Association, pp. 219–230.

[6] McKenney, P. E., Desnoyers, M., and Jiangshan, L. User-space RCU. `https://lwn.net/Articles/573424/`, November 2013.

[7] McKenney, P. E., and Prasad, A. Recent read-mostly research in 2015. `http://lwn.net/Articles/667593/`, December 2015.

[8] McKenney, P. E., and Slingwine, J. D. Read-copy update: Using execution history to solve concurrency problems. In *Parallel and Distributed Computing and Systems* (Las Vegas, NV, October 1998), pp. 509–518.

[9] McKenney, P. E., Wong, M., and Michael, M. P0232r0: A concurrency toolkit for structured deferral or optimistic speculation. `http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2016/p0232r0.pdf`, February 2016.

[10] Ramalhete, P. Implementing a reader-writer lock using rcu. `http://concurrencyfreaks.blogspot.com/2015/10/implementing-reader-writer-lock-using.html`, October 2015.

[11] Sarma, D., and McKenney, P. E. Making RCU safe for deep submillisecond response realtime applications. In *Proceedings of the 2004 USENIX Annual Technical Conference (FREENIX Track)* (June 2004), USENIX Association, pp. 182–191.

[12] Shenoy, G. R. [patch 4/5] lock_cpu_hotplug: Redesign - lightweight implementation of lock_cpu_hotplug. Available: `http://lkml.org/lkml/2006/10/26/73` [Viewed January 26, 2009], October 2006.

Change Log

This paper first appeared as **P0461R0** in October of 2016. Revisions to this document are as follows:

- Convert to single-column mode. (November 16, 2016.)
- Change `call()` to `retire()` for hazard-pointer compatibility. (January 4, 2017.)
- Change `rcu_scoped_reader` to `rcu_guard` for compatibility with existing RAII mechanisms. (January 19, 2017.)
- Change `rcu_head_delete` to `rcu_obj_base` for compatibility with hazard pointers. (January 19, 2017.)
- Update to indicate preferred C++ RCU approach. (January 19, 2017.)
- Call out relationships between classes for RCU and for hazard pointers. (January 19, 2017.)
- Add constructors to `rcu_domain` to match those of `hazptr_domain`. (February 1, 2017.)
- Add `quiescent_state_needed()` member function to `rcu_domain` to allow code using RCU to complain if its requirements are not met, based on discussions with Geoffrey Romer and Andrew Hunter. (February 3, 2017.)
- Added references to related papers. (February 5, 2017.)

At this point, the paper was published as **P0461R1**.
Further revisions to this document are as follows:

- Indicate which sections are preferred vs. informational. (February 17, 2017.)
- Update document number and boilerplate. (March 2, 2017.)
- Drop RCU domains, focusing on the `sys_membarrier()` implementation from the userspace RCU library.
- Provide `rcu_reader` as RAII class, replacing the old `rcu_guard`.
- Trim down the `rcu_obj_base` class.
- Remove implementation details from the code shown in the figures.
- Add a pointer to the github repository containing a working implementation.
- Add free functions for grace-period wait, retire-invocation wait, and non-intrusive retire.
- Add Khiszinsky citation. (August 31, 2017.)
- Add explanation of the purpose of domains. (October 2, 2017.)
- Add reference to WG21/P0750R0. (October 15, 2017.)