P0461R0: Proposed RCU C++ API

Doc. No.: WG21/P0461R0
Date: 2016-10-16
Reply to: Paul E. McKenney, Maged Michael, Michael Wong,
Isabella Muerte, and Arthur O'Dwyer
Email: paulmck@linux.vnet.ibm.com, maged.michael@gmail.com, fraggamuffin@gmail.com,
isabella.muerte@mnmlstc.com, and arthur.j.odwyer@gmail.com

October 16, 2016

This document is based on WG21/P0279R1 combined with feedback at the 2015 Kona and 2016 Jacksonville 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. 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[4].

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.

Note also that a redefinition of the infamous `memory_order_consume` is the subject of a separate paper.

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"), and WG21/P0098R1 ("Towards Implementation and Use of `memory_order_consume`").

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 5 presents a table comparing reference counting, hazard pointers, and RCU, and finally Section 6 presents a summary.

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

Figure 1: Base RCU API

2 Base RCU API

Figure 1 shows the base RCU API as provided by implementations such as userspace RCU [1, 3]. This API is provided for compatibility with existing practice as well as to provide the highest performance for

fast-path code. (See Figure 2 for a proposed API that permits multiple RCU domains, as requested by several committee members.)

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 repeatedly discovered by any number of people new to RCU [2].

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 invocations 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()`.

2.1 RCU Domains

The userspace RCU library features several RCU implementations, each optimized for different use cases.

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

```
 1 class rcu_domain {
 2 public:
 3   virtual void register_thread() = 0;
 4   virtual void unregister_thread() = 0;
 5   static inline bool register_thread_needed()
 6           { return true; }
 7   virtual void read_lock() noexcept = 0;
 8   virtual void read_unlock() noexcept = 0;
 9   virtual void synchronize() noexcept = 0;
10   virtual void call(class rcu_head *rhp,
11         void cbf(class rcu_head *rhp)) = 0;
12   virtual void barrier() noexcept = 0;
13   virtual void quiescent_state() noexcept = 0;
14   virtual void thread_offline() noexcept = 0;
15   virtual void thread_online() noexcept = 0;
16 };
```

Figure 2: RCU Domain Base Class

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`.

2.2 Run-Time Domain Selection

Figure 2 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`
- Memory-barrier RCU: `class rcu_mb`
- QSBR RCU: `class rcu_qsbr`
- Signal-based RCU: `class rcu_signal`

```
 1  class rcu_scoped_reader {
 2  public:
 3    rcu_scoped_reader() noexcept
 4    {
 5      this->rd = nullptr;
 6      rcu_read_lock();
 7    }
 8
 9    explicit rcu_scoped_reader(rcu_domain *rd)
10    {
11      this->rd = rd;
12      rd->read_lock();
13    }
14
15    rcu_scoped_reader(const rcu_scoped_reader &) = delete;
16
17    rcu_scoped_reader&operator=(const rcu_scoped_reader &) = delete;
18
19    ~rcu_scoped_reader() noexcept
20    {
21      if (this->rd)
22        this->rd->read_unlock();
23      else
24        rcu_read_unlock();
25    }
26
27  private:
28    rcu_domain *rd;
29  };
```

Figure 3: RCU Scoped Readers

3 Scoped Readers

In some cases, it might be convenient to use a scoped style for RCU readers, especially if the read-side critical section might be exited via exception. The `rcu_scoped_reader` class shown in Figure 3 may be used for this purpose. An argumentless constructor uses the API, or an `rcu_domain` class may be passed to the constructor to use the specified RCU implementation.

This is intended to be used in a manner similar to `std::lock_guard`.

```
1   template<typename T, typename D = default_delete<T>,
2     bool E = is_empty<D>::value>
3   class rcu_head_delete: private rcu_head {
4     D deleter;
5   public:
6     static void trampoline(rcu_head *rhp)
7     {
8       auto rhdp = static_cast<rcu_head_delete *>(rhp);
9       auto obj = static_cast<T *>(rhdp);
10      rhdp->deleter(obj);
11    }
12
13    void call(D d = {})
14    {
15      deleter = d;
16      call_rcu(static_cast<rcu_head *>(this), trampoline);
17    }
18
19    void call(rcu_domain &rd, D d = {})
20    {
21      deleter = d;
22      rd.call(static_cast<rcu_head *>(this), trampoline);
23    }
24  };
25
```

Figure 4: RCU Callbacks: Derived Function Call

```
1   template<typename T, typename D>
2   class rcu_head_delete<T,D,true>: private rcu_head {
3   public:
4     static void trampoline(rcu_head *rhp)
5     {
6       auto rhdp = static_cast<rcu_head_delete *>(rhp);
7       auto obj = static_cast<T *>(rhdp);
8       D()(obj);
9     }
10
11    void call(D d = {})
12    {
13      call_rcu(static_cast<rcu_head *>(this), trampoline);
14    }
15
16    void call(rcu_domain &rd, D d = {})
17    {
18      rd.call(static_cast<rcu_head *>(this), trampoline);
19    }
20  };
```

Figure 5: RCU Callbacks: Derived Deletion

4 RCU Callback Handling

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 first looks at some approaches that leverage C++ overloading and inheritance, which has the benefit of avoiding macros and providing better type safety. This will not be an either-or situation: Several of these approaches are likely to be generally useful.

4.1 Derived Function Call

The `rcu_head_derived` class provides overloaded `call()` methods, as shown in Figure 4. These methods take a deleter and an optional `rcu_domain` class instance. 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*)`, or a lambda type. 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.

If an `rcu_domain` is supplied, its `call()` member function is used, otherwise the `call_rcu()` free function is used.

The next section provides a specialization that only permits `delete`, which allows omitting the deleter, thus saving a bit of memory.

4.2 Derived Deletion

By far the most common RCU callback simply frees the data structure. Figure 5 shows a specialization of the `rcu_head_delete` class, which supports this idiom in cases where the RCU-protected data structure may inherit from this class.

The `rcu_head_delete` class supplies a pair of overloaded `call()` member functions, the first of which has no non-defaulted argument. This argument-free member function arranges to `delete` the object after a grace period, using `call_rcu()` to do so.

The second `call()` member function takes an `rcu_domain` argument, and uses that domain's `call_rcu()` function to wait for a grace period.

Use of this approach is quite straightforward. For example, a class `foo` would inherit from `rcu_head_`

`delete<foo>`, and given a `foo` pointer `fp`, would execute `fp->call()` to cause the object referenced by `fp` to be passed to `delete` at the end of a subsequent grace period. No further action is required.

However, it is sometimes necessary to do more than simply free an object. In many cases, additional teardown actions are required, and it is often necessary to use a non-standard deallocator instead of the C++ `delete`. This possibility is covered by another specialization of the `rcu_head_delete` class, which was described in the previous section.

4.3 Pointer To Enclosing Class

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.call(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.

4.4 Address Arithmetic

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.

```
 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 call(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 call(class rcu_domain &rd,
34        void callback_func(T *obj) = nullptr)
35    {
36      this->callback_func = callback_func;
37      rd.call(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

```
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

5 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.

6 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. 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.

References

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

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

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

[4] 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.

```
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

	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.