Document number: P1122R1
Date: 20180704 (pre-San Diego)
Project: Programming Language C++, WG21, LWG, Core
Authors: Paul McKenney, Michael Wong, Maged M. Michael, Geoffrey Romer, Andrew Hunter, Arthur O'Dwyer, David S. Hollman, JF Bastien, Hans Boehm, David Goldblatt, Frank Birbacher
Email: paulmck@linux.vnet.ibm.com, michael@codeplay.com, maged.michael@acm.org, gromer@google.com, andrewhhunter@gmail.com, arthur.j.odwyer@gmail.com, dshollm@sandia.gov, jfbastien@apple.com, hboehm@google.com, davidtgoldblatt@gmail.com, frank.birbacher@gmail.com
Reply to: paulmck@linux.vnet.ibm.com

Proposed Wording for Concurrent Data Structures: Read-Copy-Update (RCU)

1. Introduction

This paper is a successor to the RCU portion of P0566R5, in response to LEWG's Rapperswil 2018 request that the two techniques be split into separate papers.

This is proposed wording for Read-Copy-Update [P0461], which is a technique for safe deferred resource reclamation for optimistic concurrency, useful for lock-free data structures. Both RCU and hazard pointers have been progressing steadily through SG1 based on years of implementation by the authors, and are in wide use in MongoDB (for Hazard Pointers), Facebook, and Linux OS (RCU).

We originally decided to do both papers' wording together to illustrate their close relationship, and similar design structure, while hopefully making it easier for the reader to review together for this first presentation. As noted above, they have been split on request.

This wording is based P0566r5, which in turn was based on that of on n4618 draft [N4618]

2. History/Changes from Previous Release

2018-07-06 [D1122R1] pre-San Diego meeting

- Added list of open issues to be addressed by TS implementation experience.

2018-06-07 [P1122R0] post-Rapperswil meeting

- Extracted the RCU portion of P0566R5 per request by LEWG.
- Add the `explicit` keyword to the `defer_lock_t` constructor of `rcu_reader` per request by LEWG.
- Add the destructor description per request by LEWG.
- Andrej Krzemienski of LEWG: Have `reset` to pre-destruct an `rcu_reader`? Defer to TS experience because we don't know this use case will actually appear.
- Anthony Williams via email: Allow the deleter to be specified at construction time in addition to at retire time. Defer to TS experience.
- Remove swap declaration, thus relying on default definition per request by LEWG.
- Moved Preamble to D0940R2.
- Archival version: https://docs.google.com/document/d/1wls6q2mE60l5uZFug5U5i21N75j--s0wFOJA3e7MuKY/edit

2018-03-12 [P0566R5] pre-Rapperswil meeting

- Updated RCU ordering guarantees for readers and deleters.
- Remove `noexcept` from `rcu_reader` destructor.
- Drop the detailed description of the `rcu_reader` destructor.
- Word the `retire` member function based on the `rcu_retire` free function.
- Word the `synchronize_rcu` free function based on `rcu_retire`.
- Rename `synchronize_rcu` to `rcu_synchronize`, as requested by LEWG in JAX.
- Confirmed that `rcu_synchronize` has SC fence semantics, and added a section to the RCU litmus-tests paper (P0868R1)
- Added feature test macros `__cpp_hazard_pointers` and `__cpp_read_copy_update`.
- Added wording constraining deleters.
- Hazard pointer changes:
    - Changed the introduction and the wording for hazptr_cleanup(), hazptr_obj_base retire(), hazptr_holder try_protect() to consider the lifetime of each hazard pointer as a series of epochs to facilitate specifying memory ordering (based on JAX evening session).
    - Changed the name of hazptr_holder get_protected() to protect(), as instructed by LEWG.
    - Changed the default constructor for hazptr_holder to return an empty hazptr_holder, as instructed by LEWG.
    - Removed the hazptr_holder member function make_empty(), by implication of changing the default constructor.
    - Added the free function make_hazptr(), which constructs a non-empty hazptr_holder, to replace the functionality of the hazptr_holder constructor.
    - Changed the name of hazptr_holder reset() to reset_protected() for clarity.

2017-11-08 [P0566R4] pre-JAX meeting

- Full RCU wording review was done at this meeting. A repeat HP wording done at this meeting for any small design deltas, although HP was approved to move to LEWG in Toronto
- Three related bugzillas tracking this:
    - 382 C++ Concurre parallel@lists.isocpp.org CONF --- Proposed Wording for Concurrent Data Structures: Hazard Pointer and Read-Copy-Update (RCU) Tue 23:01
    - 291 C++ Library fraggamuffin@gmail.com SG_R --- Hazard Pointers 2017-07-06
    - 376 C++ Concurre maged.michael@acm.org SG_R --- Hazard Pointers Mon 22:58
- Rewrote the RCU preamble to give a better introduction to RCU's concepts and use cases, including adding example code.

- Updated the ordering guarantees to be more like the C++ memory model and less like the Linux kernel memory model. (There is still some refining that needs to be done, and this is waiting on an RCU litmus-test paper by Paul E. McKenney, now available as P0868R1.)
- Removed the `lock` and `unlock` member functions from the `rcu_reader` class. These member functions are not needed because `rcu_reader` directly provides the needed RAII functionality.
- Numerous additional wording changes were made, none of which represent a change to the design, implementation, or API.
- Added some authors.
- Hazard pointer wording changes:
    - Added `hazptr_cleanup()` free function, a stronger replacement for `hazptr_barrier()`. There was no consensus in Albuquerque on the requirements for a such a function. The decision on whether to provide one and its semantics was left to the authors.
    - Significant rewrite of the wording for `hazptr_obj_base::retire()` to address the issues with memory ordering raised in Toronto.
    - Rewrite of the wording for `hazptr_holder::try_protect()` for clarity.
    - Other minor editorial changes and corrections.

2017-10-15 [P0566R3] pre-ABQ Meeting

- Changed the syntax for the polymorphic allocator passed to the constructor of `hazptr_domain`. The constructor is no longer constexpr.
- Added the free function `hazptr_barrier()` that guarantees the completion of reclamation of all objects retired to a domain.
- Changed the syntax of constructing empty `hazptr_holder`-s.
- Changed the syntax of the `hazptr_holder` member function that indicated whether a `hazptr_holder` is empty or not.
- Added a note that an empty `hazptr_holder` is different from a `hazptr_holder` that owns a hazard pointer with null value.
- Added a note to clarify that it acceptable for `hazptr_holder` `try_protect` to return true when its first argument has null value.
- Update RCU presentation to reduce member-function repetition.
- Fix RCU s/Void/void/ typo
- Remove RCU's std::nullptr_t in favor of the new-age std::defer_lock_t.
- Remove RCU's barrier() member function in favor of free function based on BSI comment

2017-07-30 [P0566R2] Post-Toronto

- Allow hazptr_holder to be empty. Add a move constructor, empty constructor, move assignment operator, and a bool operator to check for empty state.

- A call by an empty hazptr_holder to any of the following is undefined behavior: reset(), try_protect() and get_protected().
- Destruction of an hazptr_holder object may be invoked by a thread other than the one that constructed it.
- Add overload of `hazptr_obj_base` `retire()`.

2017-06-18 [P0566R1] Pre-Toronto

- Addressed comments from Kona meeting
- Removed Clause numbering 31 to leave it to the committee to decide where to inject this wording
- Renamed `hazptr_owner` `hazptr_holder`.
- Combined `hazptr_holder` member functions `set()` and `clear()` into `reset()`.
- Replaced the member function template parameter `A` for `hazptr_holder` `try_protect()` and `get_protected` with `atomic<T*>`.
- Moved the template parameter `T` from the class `hazptr_holder` to its member functions `try_protect()`, `get_protected()`, and `reset()`.
- Added a non-template overload of `hazptr_holder::reset()` with an optional `nullptr_t` parameter.
- Removed the template parameter `T` from the free function `swap()`, as `hazptr_holder` is no longer a template.
- Almost complete rewrite of the hazard pointer wording.

---

3. Guidance to Editor

RCU is a proposed addition to the C++ standard library, for the concurrency TS. It has been approved for addition through multiple SG1/SG14 sessions.

As RCU (and Hazard Pointers) are related to a concurrent shared pointer, we are looking at building a new clause 33 for Concurrency Utilities Library through P0940 [P0940].In P0940, we plan to introduce subclause on Safe Reclamation which will support RCU, Hazard Pointer, as well as a other similar features.

We will not make any assumption for now as to the placement of this wording and leave it to SG1/LEWG/LWG to decide and have used ? as a Clause placeholder. We believe that Hazard Pointers and RCU should appear in the same section.

4. Proposed wording

?.1 Read-Copy Update (RCU) [rcu]

1. RCU is a synchronization mechanism that can be used for linked data structures that are frequently read, but seldom updated. RCU does not provide mutual exclusion, but instead allows the user to defer specified actions to a later time at which there are no longer any RCU *read-side critical sections* that were executing at the time the deferral started. Threads executing within an RCU read-side critical section are called *readers*.
2. RCU read-side critical sections are designated using an RAII class `std::rcu_reader`.
3. In one common use case (example shown below), RCU linked-structure updates are divided into two segments.
   [Note— The following example shows how RCU allows updates to be carried out in the presence of concurrent readers. The reader function executes in one thread and the update function in another. The `rcu_reader` instance in `print_name` protects the referenced object `name` from being deleted by `rcu_retire` until the reader has completed.

```
std::atomic<std::string *> name;

// called often and in parallel!
void print_name() {
  std::rcu_reader rr;
  std::string *s = name.load(std::memory_order_acquire);
 /* ...use *s... */
}

// called rarely
void update_name(std::string *new_name) {
  std::string *s = name.exchange(new_name, std::memory_order_acq_rel);
  std::rcu_retire(s);
}
```

   —end note]
   The first segment can be safely executed while RCU readers are concurrently traversing the same part of the linked structure, for example, removing some objects from a linked list. The second segment cannot be safely executed while RCU readers are accessing the removed objects; for example, the second segment typically deletes the objects removed by the first segment. RCU can also be used to prevent RCU readers from observing transient atomic values, also known as the A-B-A problem.
4. A class `T` can inherit from `std::rcu_obj_base<T>` to inherit the `retire` member function and the intrusive machinery required to make it work. Alternatively, any class `T` can be passed to the `std::rcu_retire` free function template, whether it inherits from

`std::rcu_obj_base<T>` or not. The free function is expected to have performance and memory-footprint advantages, but unlike the member function can potentially allocate. Both types of retire functions arrange to invoke the deleter at a later time, when it can guarantee that no *read-side critical section* is still accessing (or can later access) the deleted data.

5. A `std::rcu_synchronize` free function blocks until all preexisting or concurrent *read-side critical sections* have ended. This function may be used as an alternative to the retire functions, in which case the `rcu_synchronize` follows the first (removal) segment of the update and precedes the second (deletion) segment of the update.
6. A `std::rcu_barrier` free function blocks until all previous (*happens before* [intro.multithreading]) calls to `std::rcu_retire` have invoked and completed their deleters. This is helpful, for instance, in cases where deleters have observable effects, or when it is desirable to bound undeleted resources, or when clean shutdown is desired.

Header <rcu> synopsis

```
namespace std {
namespace experimental {

// ?.2, class template rcu_obj_base
template<typename T, typename D = default_delete<T>>
    class rcu_obj_base;

// ?.2.2, class rcu_reader: RCU reader as RAII
class rcu_reader;

// ?.2.3 function rcu_synchronize
void rcu_synchronize() noexcept;

// ?.2.4 function rcu_barrier
void rcu_barrier() noexcept;

// ?.2.5 function template rcu_retire
template<typename T, typename D = default_delete<T>>
void rcu_retire(T* p, D d = {});
} // namespace experimental
} // namespace std
```

?.2.1, class template rcu_obj_base [rcu.base]

Objects of type T to be protected by RCU inherit from rcu_obj_base<T>. Note that rcu_obj_base<T> has no non-default constructors or destructors.

```
template<typename T, typename D = default_delete<T>>
    class rcu_obj_base {
public:
  // ?.2.1.1, rcu.base.retire: Retire a removed object and pass the
  // responsibility for reclaiming it to the RCU library.
  void retire(
      D d = {});
};
```

1. A client-supplied template argument D shall be a function object type ([function.objects]) for which, given a value d of type D and a value ptr of type T*, the expression d(ptr) is valid and has the effect of disposing of the pointer as appropriate for that deleter.

?.2.1.1, rcu_obj_base retire [rcu.base.retire]

```
   void retire(
      D d = {}) noexcept;
```

1. Effects: Equivalent to rcu_retire(this, d).
2. Throws: Nothing. [*Note:* This implies it may not allocate memory via operator new. *end note*]

?.2.2, class rcu_reader [rcu.reader]

This class provides RAII RCU readers.

```
// ?.2.2, class template rcu_readers
class rcu_reader {
public:
  // ?.2.2.1, rcu_reader: RAII RCU readers
   rcu_reader() noexcept;
   explicit rcu_reader(std::defer_lock_t) noexcept;
   rcu_reader(const rcu_reader&) = delete;
   rcu_reader(rcu_reader&& other) noexcept;
   rcu_reader& operator=(const rcu_reader&) = delete;
   rcu_reader& operator=(rcu_reader&& other) noexcept;
   ~rcu_reader();
};
```

?.2.2.1, class template rcu_reader constructors [rcu.reader.cons]

```
  rcu_reader() noexcept;
```

1. Effects: Creates an active `rcu_reader` that is associated with a new RCU read-side critical section.
2. Postconditions: For each retire-function (`std::rcu_obj_base::retire` or `std::rcu_retire`) invocation such that this constructor does not happen before (C++Std [intro.races]) that retire-function invocation, prevents the corresponding deleter from being invoked.

```
explicit rcu_reader(std::defer_lock_t) noexcept;
```

1. Effects: Creates an inactive `rcu_reader`.

```
rcu_reader(rcu_reader&& other) noexcept;
```

1. Effects: Creates an active `rcu_reader` that is associated with the RCU read-side critical section that was associated with `other`. If `this` was already associated with an RCU read-side critical section, that critical section ends as described in the destructor. The `rcu_reader other` becomes inactive.

```
~rcu_reader();
```

1. Effects: If the `rcu_reader` is active, ends the associated RCU read-side critical section.

?.2.2.2, class template rcu_reader assignment [rcu.reader.assign]

```
rcu_reader& operator=(rcu_reader&& other) noexcept;
```

1. Effects: If `this` is active, the corresponding RCU read-side critical section ends as described in the destructor. In either case, `this` become active and holds the RCU read-side critical section corresponding to `other`, and `other` becomes inactive.

?.2.3, function rcu_synchronize [rcu.synchronize]

```
void rcu_synchronize() noexcept;
```

1. Effects: Equivalent to:

```
std::atomic<bool> b;
rcu_retire(&b, [](std::atomic<bool> *b) {
  b->store(true);
});
while (!b->load());
```

2. Throws: nothing.

?.2.4, function rcu_barrier [rcu.barrier]

```
void rcu_barrier() noexcept;
```

1. Effects: For each invocation of a retire function (`std::rcu_obj_base::retire` or `std::rcu_retire`) that happens before this call, blocks until the corresponding deleter has completed. May additionally wait for the completion of the corresponding deleter of any retire function call that does not happen after this call.
2. Synchronization: The completion of each such deleter strongly happens before the return from `rcu_barrier`.

?.2.5, function template rcu_retire [rcu.retire]

```
template<typename T, typename D = default_delete<T>>
void rcu_retire(T* p, D d = {});
```

1. Requires: `D` shall satisfy the requirements of MoveConstructible and such construction shall not exit via an exception. The expression `d(p)` shall be well-formed, shall have well-defined behavior, and shall not throw exceptions. The object referenced by `p` must not be passed to any other invocation of a retire function.
2. Effects: Causes `d(p)` to be invoked later at an unspecified point on unspecified execution agents. Guarantees that for each instance R of `rcu_reader`, one of two things hold:
   - `rcu_retire` strongly happens before R's constructor
   - R's destructor strongly happens before the invocation of the deleter.
   [Note: If R's constructor happens before `rcu_retire`, then R's destructor strongly happens before the deleter. If the deleter happens before R's destructor, then `rcu_retire` strongly happens before R's constructor. --- end note]
3. Progress: If R_s is the set of `rcu_reader` instances where the invocation of `rcu_retire` does not happen before the construction of those readers, once the destructors of all instances in R_s have completed, the deleter is invoked on an unspecified execution agent within a finite amount of time.

5. Issues Requiring TS Implementation Experience

We expect that implementation experience will help shed light on the following open issues:

1. Should the standard provide RCU domains, such that readers in one domain have no effect on updaters in any other domain? Although these have proven useful in other environments (e.g., Linux-kernel SRCU), they are rarely used and it has usually taken

some years in any given environment for the need for them to arise. In addition, RCU can take advantage of extremely effective batching optimizations that are partially or even wholly defeated by excessive use of domains. These considerations have resulted in the initial proposal to exclude RCU domains. (The discussions covering this topic have been primarily face to face.)

2. Should the `rcu_obj_base` class provide an additional constructor that takes a deleter argument, thus allowing the deleter to be omitted from the call to `retire()` or `rcu_retire()`? (Note that this might result in an extra store at constructor time when the deleter is specified at retire time.) Should this class delete copy constructors? (See the isocpp-parallel email thread in May 2018 with subject line "Feedback on Proposed wording for RCU (p0566R5)".)
3. Should the type specifier for the deleter more closely resemble that of `unique_ptr`? (See the isocpp-parallel email thread in May 2018 with subject line "Feedback on Proposed wording for RCU (p0566R5)".) The following options have been suggested:
    a. "D must be DefaultConstructible (for the constructors without an explicit deleter) (23.11.1.2.1p1 and p5)", or
    b. "D must be constructible from std::forward<decltype(D)>(d), where d is the supplied deleter (which probably means MoveConstructible, but also accounts for reference types) (23.11.1.2.1p9-12)"
4. Should the `[[no_unique_address]]` attribute be applied to the deleter to take advantage of the C++20 equivalent of empty base optimization (EBO)? (See the isocpp-parallel email thread in May 2018 with subject line "Feedback on Proposed wording for RCU (p0566R5)".)
5. What constraints need to be placed on the context in which deleters are invoked? If there are background threads, what control do users need over the time at which they are spawned and terminated? Should it be possible to construct an implementation with no background threads, and, if so, in what context do the destructors run? What additional constraints (e.g., deadlock avoidance) need to be placed on users of implementations with no background threads? (See the isocpp-parallel email thread in July 2018 with subject line "Background threads and P0561".)
6. Is special handling required for Microsoft dynamic-link libraries (DLLs), particularly surrounding special handling of background threads and pending deleters when DLLs are unloaded? Similarly, is special handling required to support clean shutdown of applications? (See the isocpp-parallel email thread in July 2018 with subject line "Background threads and P0561".)
7. Can all major platforms support `latest<T>` instances with static storage duration? (See the isocpp-parallel email thread in July 2018 with subject line "Background threads and P0561".)
8. Clean shutdown when deleters invoke rcu_retire(). Note that we can have cross-retires where hazard-pointer deleters invoke rcu_retire() and the corresponding RCU deleters retire a hazard pointer. Note that users can prevent clean shutdown by cascading retires forever. Should high-quality implementations do a best-effort attempt to shut down (even in the absence of rcu_barrier()), given that malicious users can prevent this. Note

that this not unprecedented: Users can also prevent clean shutdown via infinite loops and various other forms of infinite recursion. However, even without infinite recursion, it is possible for a shutdown-time issues involving shutdown-time destructors using RCU after RCU itself has destructed (the Linux kernel avoids this by never destructing RCU). We hope that TS implementation experience will help us identify good strategies for best-effort shutdown. (Off-list discussion between Paul, Maged, and Michael.)

We therefore are not working to resolve these before the concurrency TS2 is created, but rather expecting that implementation experience based on the TS will shed light on the various possible resolutions.

6. Acknowledgements

The authors thank Olivier Giroux, Pablo Halpern, Lee Howes, Xiao Shi, Viktor Vafeiadis, Dave Watson and other members of SG1 and LEWG for useful discussions and suggestions that helped improve this paper and its earlier versions.

7. References

RCU implementation: https://github.com/paulmckrcu/RCUCPPbindings (See Test/paulmck)

[N4618] http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2016/n4618.pdf

[P0233] Hazard Pointers: Safe Resource Reclamation for Optimistic Concurrency http://wg21.link/P0233

[P0461] Proposed RCU C++ API http://wg21.link/P0461

[P0566] Proposed Wording for Concurrent Data Structures: Hazard Pointer and Read-Copy-Update (RCU) http://wg21.link/P0566

[P0940] Concurrency TS is growing: Utilities and Data Structures http://wg21.link/P0940