Document Number: D2545R0
Date: 2022-02-11
Revises: None
Reply to: Paul E. McKenney
Meta
paulmckrcu@gmail.com

Why RCU Should be in C++26

Authors:
Paul McKenney, Michael Wong, Maged M. Michael, Andrew Hunter, Daisy Hollman, JF Bastien, Hans Boehm, David Goldblatt, Frank Birbacher, Erik Rigtorp, Tomasz Kamiński

email:
paulmckrcu@fb.com, michael@codeplay.com, maged.michael@acm.org, andrewhhunter@gmail.com, dhollman@google.com, cxx@jfbastien.com, hboehm@google.com, davidtgoldblatt@gmail.com, frank.birbacher@gmail.com, erik@rigtorp.se, tomaszkam@gmail.com

Contents

1 Introduction

We propose RCU for inclusion into C++26. This paper contains proposed rationale to support RCU into C++26 as well as the interface and wording for RCU, a technique for safe deferred reclamation.

1.1 Proposed Entry to C++26 IS

A near-superset of this proposal is implemented in the Folly RCU library. This library has used in production for several years, so we have good implementation experience for the proposed variant of RCU.

This proposal is identical to that in Concurrency TS 2. We expect that the proposal in Concurrency TS 2 will change over time, for example, adding some of the features that are present in the Folly RCU library or in the Linux kernel. Such features might include:

1. Multiple RCU domains. For example, SRCU provides these in the Linux kernel. However, RCU was in the Linux for four years before this was needed, so it is not yet in the TS nor is it in the proposal for C++26.
2. Special-purpose RCU implementations. For example, the Linux kernel has specialized implementations for preemptible environments, single-CPU systems, as well as three additional implementations required by the Linux kernel's tracing and extended Berkeley Packet Filter (eBPF) use cases. However, none of these seem applicable to userspace applications, so none of them are yet in the TS or yet in in the proposal for C++26.
3. Polling grace-period-wait APIs. These allow non-blocking algorithms to interface with RCU grace periods, for example, in the Linux kernel, they allow NMI handlers to do RCU updates. (NMI handlers could do RCU readers from the get-go.) However, RCU was in the Linux kernel for more than a decade before such APIs were needed, so they are not yet in the TS nor are they in the proposal for C++26.
4. Numerous efficiency-oriented APIs. For but one example, the Linux kernel has an alternative `rcu_access_pointer()` that can be used in place of `rcu_dereference()` (Linux-kernelese for "consume load") when the resulting pointer will not be dereferenced (for example, when it is only going to be compared to `NULL`). But it is not clear which (if any) of these would be accepted into the Linux kernel today, given the properties of modern computer hardware. Therefore, these are not yet in the TS nor are they in the proposal for C++26.

The snapshot library described in P0561R5 ("RAII Interface for Deferred Reclamation") provides an easy-to-use deferred-reclamation facility applying only to a single object which is intended to be based upon either RCU or Hazard Pointers. It cannot replace either RCU or Hazard Pointers.

The Hazard Pointers library described in D2530R0 ("Why Hazard Pointers Should Be in C++26"). As a very rough rule of thumb, Hazard Pointers can be considered to be a scalable replacement for reference counters and RCU can be considered to be a scalable replacement for reader-writer locking. A high-level comparison of reference counting, Hazard Pointers, and RCU is displayed in Table 1.

Note that we are making this working paper available before Concurrency TS2 been published, which some might feel is unconventional. On the other hand, Paul was asked to begin this effort in 2014, it is now 2022, and C++ implementations have been used in production for some time, perhaps most notably the Folly RCU library.

1.2 History

This paper is derived from N4895, which was in turn based on P1122R4.

Property	Reference Counting	Hazard Pointers	RCU
Readers	Slow and unscalable	*Fast and scalable*	*Fast and scalable*
Unreclaimed Objects	*Bounded*	*Bounded*	Unbounded
Traversal Retries?	If object deleted	If object deleted	*Never*
Reclamation latency?	*Fast*	Slow	Slow

Table 1: High-Level Comparison of Deferred-Reclamation Techniques

P1122R4 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].

1.3 Source-Code Access

The Folly library is open source, and its RCU implementation may be accessed here:

— https://github.com/facebook/folly/blob/main/folly/synchronization/Rcu.h

— https://github.com/facebook/folly/blob/main/folly/synchronization/Rcu-inl.h

— https://github.com/facebook/folly/blob/main/folly/synchronization/Rcu.cpp

There is an additional reference implementation of this proposal. Unlike the Folly library's version, this reference implementation is not production quality. However, it is quite a bit simpler, having delegated the difficult parts to the C-language userspace RCU library:

— https://github.com/paulmckrcu/RCUCPPbindings/tree/master/Test/paulmck

— https://liburcu.org

1.4 Acknowledgments

We owe special thanks to Jens Maurer, Arthur O'Dwyer, and Geoffrey Romer for their many contributions to this effort.

2 Safe reclamation [saferecl]

2.1 General [saferecl.general]

This clause adds safe-reclamation techniques, which are most frequently used to straightforwardly resolve access-deletion races.

2.2 Read-copy update (RCU) [saferecl.rcu]

2.2.1 General [saferecl.rcu.general]

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 schedule specified actions such as deletion at some later time.

2 A class type `T` is *rcu-protectable* if it has exactly one public base class of type `rcu_obj_base<T,D>` for some `D` and no base classes of type `rcu_obj_base<X,Y>` for any other combination `X`, `Y`. An object is rcu-protectable if it is of rcu-protectable type.

3 An invocation of `unlock` U on an `rcu_domain dom` corresponds to an invocation of `lock` L on `dom` if L is sequenced before U and either

(3.1) — no other invocation of `lock` on `dom` is sequenced after L and before U or

(3.2) — every invocation of `unlock` U' on `dom` such that L is sequenced before U' and U' is sequenced before U corresponds to an invocation of `lock` L' on `dom` such that L is sequenced before L' and L' is sequenced before U'.

[*Note 1*: This pairs nested locks and unlocks on a given domain in each thread. — *end note*]

4 A *region of RCU protection* on a domain `dom` starts with a `lock` L on `dom` and ends with its corresponding `unlock` U.

5 Given a region of RCU protection R on a domain `dom` and given an evaluation E that scheduled another evaluation F in `dom`, if E does not strongly happen before the start of R, the end of R strongly happens before evaluating F.

6 The evaluation of a scheduled evaluation is potentially concurrent with any other such evaluation. Each scheduled evaluation is evaluated at most once.

2.2.2 Header <rcu> synopsis [saferecl.rcu.syn]

```
namespace std::experimental::inline concurrency_v2 {
  // 2.2.3, class template rcu_obj_base
  template<class T, class D = default_delete<T>>
    class rcu_obj_base;

  // 2.2.4, class rcu_domain
  class rcu_domain;

  // 2.2.5, rcu_default_domain
  rcu_domain& rcu_default_domain() noexcept;

  // 2.2.6, rcu_synchronize
  void rcu_synchronize(rcu_domain& dom = rcu_default_domain()) noexcept;

  // 2.2.7, rcu_barrier
  void rcu_barrier(rcu_domain& dom = rcu_default_domain()) noexcept;

  // 2.2.8, rcu_retire
  template<class T, class D = default_delete<T>>
    void rcu_retire(T* p, D d = D(), rcu_domain& dom = rcu_default_domain());
}
```

2.2.3 Class rcu_obj_base [saferecl.rcu.base]

Objects of type `T` to be protected by RCU inherit from a specialization of `rcu_obj_base<T,D>`.

```
template<class T, class D = default_delete<T>>
class rcu_obj_base {
public:
  void retire(D d = D(), rcu_domain& dom = rcu_default_domain()) noexcept;
protected:
  rcu_obj_base() = default;
private:
  D deleter;              // exposition only
};
```

1 A client-supplied template argument `D` shall be a function object type C++20 §20.14 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 The behavior of a program that adds specializations for `rcu_obj_base` is undefined.

3 `D` shall meet the requirements for *Cpp17DefaultConstructible* and *Cpp17MoveAssignable*.

4 `T` may be an incomplete type.

5 If `D` is trivially copyable, all specializations of `rcu_obj_base<T,D>` are trivially copyable.

```
void retire(D d = D(), rcu_domain& dom = rcu_default_domain()) noexcept;
```

6 *Mandates*: `T` is an rcu-protectable type.

7 *Preconditions*: `*this` is a base class subobject of an object `x` of type `T`. The member function `rcu_obj_base<T,D>::retire` was not invoked on `x` before. The assignment to ***`deleter`*** does not throw an exception. The expression ***`deleter`***`(addressof(x))` has well-defined behavior and does not throw an exception.

8 *Effects*: Evaluates ***`deleter`*** `= std::move(d)` and schedules the evaluation of the expression `deleter(addressof(x))` in the domain `dom`.

9 *Remarks*: It is implementation-defined whether or not scheduled evaluations in `dom` can be invoked by the `retire` function.

[*Note 1*: If such evaluations acquire resources held across any invocation of retire on `dom`, deadlock can occur. — *end note*]

2.2.4 Class `rcu_domain` [saferecl.rcu.domain]

This class meets the requirements of *Cpp17BasicLockable* C++20 §32.2.5.2 and provides regions of RCU protection.

[*Example 1*:

```
std::scoped_lock<rcu_domain> rlock(rcu_default_domain());
```

— *end example*]

```
class rcu_domain {
public:
  rcu_domain(const rcu_domain&) = delete;
  rcu_domain& operator=(const rcu_domain&) = delete;

  void lock() noexcept;
  void unlock() noexcept;
};
```

The functions `lock` and `unlock` establish (possibly nested) regions of RCU protection.

2.2.4.1 `rcu_domain::lock` [saferecl.rcu.domain.lock]

```
void lock() noexcept;
```

1 *Effects*: Opens a region of RCU protection.

2 *Remarks*: Calls to the function lock do not introduce a data race (C++20 §6.9.2.1) involving `*this`.

2.2.4.2 `rcu_domain::unlock` [saferecl.rcu.domain.unlock]

```
void unlock() noexcept;
```

1 *Preconditions*: A call to the function `lock` that opened an unclosed region of RCU protection is sequenced before the call to `unlock`.

2 *Effects*: Closes the unclosed region of RCU protection that was most recently opened.

3 *Remarks*: It is implementation-defined whether or not scheduled evaluations in `*this` can be invoked by the `unlock` function.

[*Note 1*: If such evaluations acquire resources held across any invocation of `unlock` on `*this`, deadlock can occur. — *end note*]

Calls to the function `unlock` do not introduce a data race involving `*this`.

[*Note 2*: Evaluation of scheduled evaluations can still cause a data race. — *end note*]

2.2.5 `rcu_default_domain` [saferecl.rcu.default.domain]

```
rcu_domain& rcu_default_domain() noexcept;
```

1 *Returns*: A reference to the default object of type `rcu_domain`. A reference to the same object is returned every time this function is called.

2.2.6 `rcu_synchronize` [saferecl.rcu.synchronize]

```
void rcu_synchronize(rcu_domain& dom = rcu_default_domain()) noexcept;
```

1 *Effects*: If the call to `rcu_synchronize` does not strongly happen before the lock opening an RCU protection region `R` on `dom`, blocks until the `unlock` closing `R` happens.

2 *Synchronization*: The `unlock` closing `R` strongly happens before the return from `rcu_synchronize`.

2.2.7 `rcu_barrier` [saferecl.rcu.barrier]

```
void rcu_barrier(rcu_domain& dom = rcu_default_domain()) noexcept;
```

1 *Effects*: May evaluate any scheduled evaluations in `dom`. For any evaluation that happens before the call to `rcu_barrier` and that schedules an evaluation E in `dom`, blocks until E has been evaluated.

2 *Synchronization*: The evaluation of any such E strongly happens before the return from `rcu_barrier`.

2.2.8 Template `rcu_retire` [saferecl.rcu.retire]

```
template<class T, class D = default_delete<T>>
void rcu_retire(T* p, D d = D(), rcu_domain& dom = rcu_default_domain());
```

1 *Mandates*: `is_move_constructible_v<D>` is true.

2 *Preconditions*: `D` meets the *Cpp17MoveConstructible* and *Cpp17Destructible* requirements. The expression `d1(p)`, where `d1` is defined below, is well-formed and its evaluation does not exit via an exception.

3 *Effects*: May allocate memory. It is unspecified whether the memory allocation is performed by invoking `operator new`. Initializes an object `d1` of type `D` from `std::move(d)`. Schedules the evaluation of `d1(p)` in the domain `dom`.

[*Note 1*: If `rcu_retire` exits via an exception, no evaluation is scheduled. — *end note*]

4 *Throws*: Any exception that would be caught by a handler of type `bad_alloc`. Any exception thrown by the initialization of `d1`.

5 *Remarks*: It is implementation-defined whether or not scheduled evaluations in dom can be invoked by the `rcu_retire` function.

[*Note 2*: If such evaluations acquire resources held across any invocation of `rcu_retire` on `dom`, deadlock can occur. — *end note*]