Document Number: P2545R1
Date: 2022-10-04
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, Olivier Giroux, David Vernet

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, ogiroux@apple.com, dvernet@fb.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. We further propose that the wording in Section 2.2 be adopted as a new "Safe reclamation" chapter of the IS, and we anticipate that hazard pointers would be covered by another section of this same chapter.

The purpose of adding RCU to the IS is to provide a small number of known-good implementations of RCU in standard libraries. RCU is easy to get wrong, and one purpose standard libraries is to provide good implementations of things that are easy to get wrong.

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 in this 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 in this 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 in this proposal for C++26.
4. Async-friendly APIs for RCU's blocking APIs. These might leverage the aforementioned polling APIs. However, more work is required to determine exactly what support is required, so they are not in this proposal for C++26.
5. A free function similar to `rcu_retire` that uses an `rcu_obj_base` if available, but which invokes `rcu_retire` if not. (Suggested by Tomasz Kamiński.) However, this facility has not yet been spotted in the wild, so it is not in this proposal for C++26.
6. A memory allocator might be supplied for the use of `rcu_retire`. Please note that if different allocators can be supplied to different calls to `rcu_retire`, then there must be a way to tag the allocated memory with the corresponding deleter. However, this facility has not yet been spotted in the wild, so it is not in this proposal for C++26.
7. 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 in this 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,

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

With Reader-Writer Locking	With RCU in the intrusive style
`struct Data  /* members */ ;`	`struct Data :  std::rcu_obj_base<Data>  /* members */ ;`
`Data* data_;` `std::shared_mutex m_;`	`std::atomic<Data*> data_;`

`template <typename Func>`
`Result reader_op(Func fn) {`
` std::shared_lock<std::shared_mutex> l(m_);`
` Data* p = data_;`
` // fn should not block too long or call update()`
 	
` return fn(p);`
`}`
	`template <typename Func>`
	`Result reader_op(Func fn) {`
	` std::scoped_lock l(std::rcu_default_domain());`
	` Data* p = data_;`
	` // fn should not block too long or call`
	` // rcu_synchronize(), rcu_barrier(), or`
	` // rcu_retire(), directly or indirectly`
	` return fn(p);`
	`}`

`// May be called concurrently with reader_op`
`void update(Data* newdata) {`
` Data* olddata;`
` {`
`  std::unique_lock<std::shared_mutex> wlock(m_);`
`  olddata = std::exchange(data_, newdata);`
` }`
` delete olddata; // reclaim *olddata immediately`
`}`
	`// May be called concurrently with reader_op`
	`void update(Data* newdata) {`
	` Data* olddata = data_.exchange(newdata);`
	
	` olddata->retire(); // reclaim *olddata when safe`
	`}`

Table 2: Tony Table for Reader-Writer Locking and Intrusive RCU

and C++ implementations have been used in production for some time, perhaps most notably the Folly RCU library.

1.2 Feature-Test Macro

We propose a new feature-test macro `__cpp_lib_rcu` be added to Section 17.3.2 of the IS.

1.3 Tony Tables

Although RCU can be applied to a great many use cases, its most common use case is as a replacement for reader-writer locking. The reader-writer usage patterns most susceptible to conversion to RCU are those where a value is computed while read-holding that lock, then used after releasing that same lock.

Table 2 compares reader-writer locking and intrusive RCU, that is, when the RCU-protected data items inherit from `std::rcu_obj_base<T>` and use the `->retire()` member function.

Table 3 compares reader-writer locking and non-intrusive RCU, that is, when the RCU-protected data items do *not* inherit from `std::rcu_obj_base<T>` and instead use the `std::rcu_retire()` free function.

Table 4 compares reader-writer locking and synchronous RCU, that is, when the RCU updater does an explicit wait for readers. When using this style, RCU-protected data items need not inherit from `std::rcu_obj_base<T>`.

1.4 History

This paper updates P2545R0 based on discusssions in SG1 and LEWG.

P2545R0 was derived from N4895, which was in turn based on P1122R4.

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.

With Reader-Writer Locking	With RCU in the non-intrusive style
struct Data /* members */ ;	struct Data /* members */ ;
Data* data_; std::shared_mutex m_;	std::atomic<Data*> data_;

template <typename Func>
Result reader_op(Func fn) {
 std::shared_lock<std::shared_mutex> l(m_);
 Data* p = data_;
 // fn should not block too long or call update()
 return fn(p);
}
	template <typename Func>
	Result reader_op(Func fn) {
	 std::scoped_lock l(std::rcu_default_domain());
	 Data* p = data_;
	 // fn should not block too long or call
	 // rcu_synchronize(), rcu_barrier(), or
	 // rcu_retire(), directly or indirectly
	 return fn(p);
	}

// May be called concurrently with reader_op
void update(Data* newdata) {
 Data* olddata;
 {
  std::unique_lock<std::shared_mutex> wlock(m_);
  olddata = std::exchange(data_, newdata);
 }
 delete olddata; // reclaim *olddata immediately
}
	// May be called concurrently with reader_op
	void update(Data* newdata) {
	 Data* olddata = data_.exchange(newdata);
	 std::rcu_retire(olddata); // reclaim *olddata when safe
	}

Table 3: Tony Table for Reader-Writer Locking and Non-Intrusive RCU

With Reader-Writer Locking	With RCU in the synchronous style
struct Data /* members */ ;	struct Data /* members */ ;
Data* data_; std::shared_mutex m_;	std::atomic<Data*> data_;

template <typename Func>
Result reader_op(Func fn) {
 std::shared_lock<std::shared_mutex> l(m_);
 Data* p = data_;
 // fn should not block too long or call update()
 return fn(p);
}
	template <typename Func>
	Result reader_op(Func fn) {
	 std::scoped_lock l(std::rcu_default_domain());
	 Data* p = data_;
	 // fn should not block too long or call
	 // rcu_synchronize(), rcu_barrier(), or
	 // rcu_retire(), directly or indirectly
	 return fn(p);
	}

// May be called concurrently with reader_op
void update(Data* newdata) {
 Data* olddata;
 {
  std::unique_lock<std::shared_mutex> wlock(m_);
  olddata = std::exchange(data_, newdata);
 }
 delete olddata; // reclaim *olddata immediately
}
	// May be called concurrently with reader_op
	void update(Data* newdata) {
	 Data* olddata = data_.exchange(newdata);
	 std::rcu_synchronize(); // wait until it's safe
	 delete olddata; // then reclaim *olddata
	}

Table 4: Tony Table for Reader-Writer Locking and Synchronous RCU

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 the committee's request.

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

1.5 Source-Code Access

This section presents C++ reference implementations, other C++ implementations, additional implementations and use cases, and performance implications.

Counting the two reference implementation, this section points out eleven implementations of RCU-like mechanisms in C++.

1.5.1 Reference C++ Implementations

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

1.5.2 Other C++ Implementations

Maxim Khizhinsky added a C++ implementation of RCU to his libcds around 2017. URL: `https://github.com/khizmax/libcds/tree/master/cds/urcu`

Avi Kivity added a C++ implementation of RCU to the OSv kernel in 2010. URL: `https://github.com/cloudius-systems/osv/blob/master/include/osv/rcu.hh`

Google uses an internally developed C++ RCU implementation alluded to by Andrew Hunter's and Geoffrey Romer's P0561 C++ working paper. This implementation makes use of restartable sequences in addition to facilities defined in the standard. URL: `https://www.open-std.org/jtc1/sc22/wg21/docs/papers/2020/p0561r5.html`

Isaac Gelado and Michael Garland describe use of a CUDA/C++ RCU in GPU programming in their 2019 PPoPP paper entitled "Throughput-Oriented GPU Memory Allocation". URL: `https://dl.acm.org/doi/10.1145/3293883.3295727`

Márton et al. present a sample C++ implementation in their paper entitled "High-level C++ Implementation of the Read-Copy-Update Pattern", which appeared in the 2017 IEEE 14th International Scientific Conference on Informatics. URL: `https://martong.github.io/high-level-cpp-rcu_informatics_2017.pdf` The corresponding journal paper appeared in the September 2018 Acta Electrotechnica et Informatica.

In 2016, Pedro Ramalhete and Andreia Correia produced a C++ prototype implementation of RCU in the ConcurrencyFreaks GitHub repository. URL: `https://github.com/pramalhe/ConcurrencyFreaks/tree/master/CPP/papers/gracesharingurcu` This appeared in the August 2017 issue of ACM SIGPLAN Notices. URL: `https://dl.acm.org/doi/abs/10.1145/3155284.3019021`

Peter Goodman produced a prototype C++ implementation of RCU in his GitHub repository in 2012. URL: `https://github.com/pgoodman/rcu`

StackExchange user Jamal posted a C++ RCU-like linked-list algorithm in 2017. URL: `https://codereview.stackexchange.com/questions/151936/rcu-in-c11-using-stdshared-ptr-and-a-little-more`.

Gamsa et al. describe an RCU-like implementation within the Tornado and K42 research operating systems, both of which were coded in C++. Sections 5.2 and 5.3 of their 1999 OSDI paper entitled "Tornado:

Maximizing Locality and Concurrency in a Shared Memory Multiprocessor Operating System" gives an overview of their RCU-like mechanism for providing what they call "existence guarantees". URL: https://www.usenix.org/legacy/events/osdi99/full_papers/gamsa/gamsa.pdf

There are implementations of RCU-like mechanisms in proprietary applications, but these cannot be divulged to the committee without the permission of their respective copyright holders. However, in the words of Fedor Pikus:

> In fact, you may already be using the RCU approach in your program without realizing it! Wouldn't that be cool? But careful now: you may be already using the RCU approach in your program in a subtly wrong way. I'm talking about the kind of way that makes your program pass every test you can throw at it and then crash in front of your most important customer (but only when they run their most critical job, not when you try to reproduce the problem).

URL: https://cppcon2017.sched.com/event/BgtF/read-copy-update-then-what-rcu-for-non-kernel-programmers

With these words, Fedor has pinpointed a major motivation for adding RCU to the C++ standard: To provide a smaller number of known-good RCU implementations to C++ users.

1.5.3 Other Use Cases

The C-language userspace RCU library appeared around 2009. The QEMU project created its own version of this library in 2015. URL: https://liburcu.org

A list of additional RCU implementations in a variety of languages may be found in Sections 9.5.5, 9.5.5.2, and 9.6.3.3 of "Is Parallel Programming Hard, And, If So, What Can You Do About It?". URL: https://kernel.org/pub/linux/kernel/people/paulmck/perfbook/perfbook-e2.pdf

RCU is used heavily in the Linux kernel:

1. http://www.rdrop.com/~paulmck/RCU/linuxusage.html
2. http://www.rdrop.com/~paulmck/techreports/survey.2012.09.17a.pdf
3. http://www.rdrop.com/~paulmck/techreports/RCUUsage.2013.02.24a.pdf
4. https://dl.acm.org/doi/10.1145/3421473.3421481

1.5.4 Performance Implications

RCU provides the best results in read-mostly situations involving linked data structures, and is most often used as a replacement for reader-writer locking. Experience in the Linux kernel indicates that well over half of the situations to which reader-writer locking is applied can be handled by RCU. RCU has provided orders-of-magnitude performance and scalability improvements in many situations, a few of which are listed below:

1. https://lwn.net/Kernel/Index/#Read-copy-update
2. http://www2.rdrop.com/~paulmck/RCU/hart_ipdps06.pdf
3. https://lkml.org/lkml/2004/8/20/137
4. https://www.linuxjournal.com/article/7124
5. https://www.linuxjournal.com/article/6993
6. http://www2.rdrop.com/~paulmck/RCU/rcu.FREENIX.2003.06.14.pdf
7. http://www2.rdrop.com/~paulmck/RCU/rcu.2002.07.08.pdf
8. http://www2.rdrop.com/~paulmck/RCU/rclock_OLS.2001.05.01c.pdf
9. https://docs.google.com/document/d/1X0lThx8OK0ZgLMqVoXiR4ZrGURHrXK6NyLRbeXe3Xac/edit?usp=sharing

Additional information may be found in Section 9.5.4 of the aforementioned "Is Parallel Programming Hard, And, If So, What Can You Do About It?".

1.6 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 {
  // 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*]