Document number: P0566R3
Date: 20171015 (pre-Albuquerque)
Project: Programming Language C++, WG21, SG1,SG14, LEWG, LWG
Authors: Michael Wong, Maged M. Michael, Paul McKenney, Geoffrey Romer, Andrew Hunter, Arthur O'Dwyer
Email: michael@codeplay.com, maged.michael@acm.org, paulmck@linux.vnet.ibm.com, gromer@google.com, ahh@google.com, arthur.j.odwyer@gmail.com
Reply to: michael@codeplay.com

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

1 **1**

/

1

1 Introduction

This is proposed wording for Hazard Pointers [P0233] and Read-Copy-Update[P0461]. Both are techniques for safe deferred resource reclamation for optimistic concurrency, useful for lock-free data structures. Both have been progressing steadily through SG1 based on years of implementation by the authors, and are in wide use in MongoDB (for Hazard Pointers) and Linux OS (RCU).

We 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. They can be split on request or on subsequent presentation.

This wording is based on n4618 draft [N4618]

2 History/Changes from Previous Release

2017-10-15 [P0566R3]

- 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]

- 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]

- 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

Hazard Pointer and RCU are proposed additions to the C++ standard library, for the concurrency TS. It has been approved for addition through multiple SG1/SG14 sessions.
As hazard pointer and rcu are related, both being utility structures for deferred reclamation of concurrent data structures, we chose to do the wording together so that the similarity in structure and wording can be more apparent. They could be separated on request.
As both techniques are related to a concurrent shared pointer, it could be appropriate to be in Clause 20 with smart pointer, or Clause 30 with thread support, or even entirely in a new clause 31 labelled concurrent Data Structures Library. However, we also believe Clause 20 does not seem appropriate as it does not cover the kind of concurrent data structures that we anticipate, while clause 30 is just about Threads, mutex, condition variables, and futures but does not cover data structures. 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.

4 Proposed wording

.

1. The following subclauses describe components to create and manage concurrent data structures, perform lock-free or lock-based concurrent execution, and synchronize concurrent operations.
2. If a data structure is to be accessed from multiple threads, then the program must be designed to ensure that any changes are correctly synchronized between threads.This clause describes data structures that have such synchronization built in, and do not require external locking.

.1 .

1. This component provides utilities for lock-free operations that can provide safe memory access, safe memory reclamation, and ABA safety.

.1.1 .

1. The following subclauses describe low-level utilities that enable the user to schedule objects for destruction, while ensuring that they will not be destroyed until after all concurrent accesses to them have completed. These utilities are summarized in Table 1. These differ from `shared_ptr` in that they do not reclaim or retire their objects automatically, rather it is under user control, and they do not rely on reference counting.

Table 1 - Concurrent Data Structure Deferred Reclamation Utilities Summary

	Subclause	Header(s)
?.1.1.2	Hazard Pointers	<hazptr>
?.1.1.3	Read-Copy-Update	<rcu>

.1.1.1 . .

Highly scalable algorithms often weaken mutual exclusion so as to allow readers to traverse linked data structures concurrently with updates. Because updaters reclaim (e.g., destroy) objects removed from a given structure, it is necessary to prevent objects from being reclaimed while readers are accessing them: Failure to prevent such accesses constitute use-after-free bugs. Hazard pointers and RCU are two techniques to prevent this class of bugs. Reference counting (e.g., `atomic_shared_pointer`) and garbage collection are two additional techniques.

1. A hazard pointer is a single-writer multi-reader pointer that can be owned by at most one thread at any time. Only the owner of the hazard pointer can set its value, while any number of threads may read its value. A thread that is about to access dynamic objects optimistically acquires ownership of a set of hazard pointers (typically one or two for linked data structures) that it will use to protect such objects from being reclaimed.
2. The owner thread sets the value of a hazard pointer to point to an object in order to indicate to concurrent threads — that might remove such object — that the object is not yet safe to reclaim.
3. The hazard pointers library allows the presence of multiple hazard pointer domains, where the safe reclamation of objects in one domain does not require checking the hazard pointers in different domains. It is possible for the same thread to participate in multiple domains concurrently. A domain can be specific to one or more objects, or encompass all shared objects.

4. Hazard pointers are not directly exposed by this interface. Operations on hazard pointers are exposed through the `hazptr_holder` class template. Each instance of `hazptr_holder` owns and operates on exactly one hazard pointer.

```
namespace std {
namespace experimental {

// ?.1, Class hazptr_domain:
class hazptr_domain;

// ?.2, Default hazptr_domain:
hazptr_domain& default_hazptr_domain() noexcept;

// ?.?, Barrier
void hazptr_barrier(hazptr_domain& domain = default_hazptr_domain());

// ?.3, Class template hazptr_obj_base:
template <typename T, typename D = std::default_delete<T>>
  class hazptr_obj_base;

// ?.4, class hazptr_holder: automatic acquisition and release of
// hazard pointers, and interface for hazard pointer operations:
class hazptr_holder;

// ?.5, hazptr_holder: Swap two hazptr_holder objects:
void swap(hazptr_holder&, hazptr_holder&) noexcept;

} // namespace experimental
} // namespace std
```

.1 **.**

1. A hazard pointer domain contains a set of hazard pointers. A domain is responsible for reclaiming objects retired to it (i.e., objects retired to this domain by calls to `hazptr_obj_base::retire()`), when such objects are not protected by hazard pointers that belong to this domain (including when this domain is destroyed).
2. The number of unreclaimed objects retired to a domain D is bounded by O(A * R * H), where A is the maximum number of simultaneously-live threads that have constructed a hazptr_holder with D as the first constructor argument, R is the maximum number of simultaneously-live threads that have invoked hazptr_obj_base::retire() with D as the first

argument, and H is the maximum number of simultaneously-live hazptr_holder objects that were constructed by a single thread with D as the first argument..

```
class hazptr_domain {
 public:

  // ?.1.1 constructor:
  explicit hazptr_domain(
      std::pmr::polymorphic_allocator<byte>* poly_alloc = {});

  // disable copy and move constructors and assignment operators
  hazptr_domain(const hazptr_domain&) = delete;
  hazptr_domain(hazptr_domain&&) = delete;
  hazptr_domain& operator=(const hazptr_domain&) = delete;
  hazptr_domain& operator=(hazptr_domain&&) = delete;

  // ?.1.2 destructor:
  ~hazptr_domain();
 private:
  std::pmr::polymorphic_allocator<byte>* alloc;  // exposition only
};
```

?**.1.1 hazptr_domain** . .

```
explicit hazptr_domain(
      pmr::polymorphic_allocator<byte>* poly_alloc = {});
```

1. Requires: `alloc` shall be the address of a valid polymorphic allocator.
2. Effects: Sets `alloc` to `poly_alloc`.
3. Throws: Nothing.
4. Remarks: All allocation and deallocation of hazard pointers in this domain will use `*alloc`. `*alloc` must not be destroyed before the destruction of this domain.

.1. hazptr_domain . .

```
~hazptr_domain();
```

1. Requires: The destruction of all `hazptr_holder` objects constructed with this domain and all `retire()` calls that take this domain as argument must happen before the destruction of the domain.
2. Effects: Deallocates all hazard pointer storage used by this domain. Reclaims any remaining objects that were retired to this domain.
3. Complexity: Linear in the number of objects retired to this domain that have not been reclaimed yet and the number of hazard pointers contained in this domain.

. **hazptr_domain**

.

```
hazptr_domain& default_hazptr_domain() noexcept;
```

1. Returns: A reference to the default `hazptr_domain`.

.

.

```
void hazptr_barrier(hazptr_domain& domain = default_hazptr_domain());
```

1. Requires: This call must happen after all calls to `retire()` with `domain` as the second argument, and happen after the threads that executed those calls (other than the current thread) have been joined.
2. Postconditions: If a `retire()` call with `domain` as its second argument happens before this call, then the return from the corresponding invocation of `reclaim` will happen before this call returns.

. **hazptr_obj_base** .

The base class template of objects to be protected by hazard pointers.

```
template <typename T, typename D = std::default_delete<T>>
class hazptr_obj_base {
 public:
  // retire
  void retire(
      D reclaim = {}, hazptr_domain& domain = default_hazptr_domain());
  void retire(
      hazptr_domain& domain);
};
```

1. `hazptr_obj_base<T, D>*` must be convertible to `T*`. [*Note:* Typically, `T` is derived from `hazptr_obj_base<T, D>`. — *end note*]
2. A client-supplied template argument `D` shall be a function object type 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.
3. `D` shall satisfy the requirements of `Destructible`.

```
  void retire(
      D reclaim = {}, hazptr_domain& domain = default_hazptr_domain());
  void retire(
      hazptr_domain& domain);
```

1. Effects: Registers the expression `reclaim(static_cast<T*>(this))` to be evaluated asynchronously. For every hazard pointer P in `domain`, if P is set to `this` by the last modification of P (in its modification order) that happens before the `retire` call, then the evaluation of the expression will happen after a later modification of P that sets it to a different value. The expression will only be evaluated once.

   This function may also evaluate any number of expressions that were previously registered by `retire()` calls with the same `domain` argument, subject to the restrictions above.

. .

Every object of type hazptr_holder is either empty or *owns* exactly one hazard pointer.

```
class hazptr_holder {
 public:
  // ?.4.1, Constructors
  explicit hazptr_holder(hazptr_domain& domain = default_hazptr_domain());
  hazptr_holder(hazptr_holder&&) noexcept;
  static hazptr_holder make_empty() noexcept;

  // disallow copy operations
  hazptr_holder(const hazptr_holder&) = delete;
  hazptr_holder& operator=(const hazptr_holder&) = delete;

  // ?.4.2, destructor
  ~hazptr_holder();

  // ?.4.3 assignment
  hazptr_holder& operator=(hazptr_holder&&) noexcept;

  // ?.4.4 empty
  bool empty() const noexcept;

  // ?.4.5- get_protected
  template <typename T>
    T* get_protected(const atomic<T*>& src) noexcept;

  // ?.4.6 try_protect
  template <typename T>
    bool try_protect(T*& ptr, const atomic<T*>& src) noexcept;

  // ?.4.7 reset
  template <typename T>
```

```
  void reset(const T* ptr) noexcept;
  void reset(nullptr_t = nullptr) noexcept;

  // ?.4.8 swap
  void swap(hazptr_holder&) noexcept;
};
```

. .1 hazptr_holder . .

```
explicit hazptr_holder(hazptr_domain& domain = default_hazptr_domain());
```

1. Effects: Acquires ownership of a hazard pointer from `domain`.
2. Throws: Any exception thrown by `domain.mr->allocate()`.

```
hazptr_holder& hazptr_holder(hazptr_holder&& other) noexcept;
```

1. Effects: Constructs a hazptr_holder that owns the pointer originally owned by `other`. `other` becomes empty.

```
static make_empty() noexcept;
```

1. Effects: Constructs an empty hazptr_holder.
2. Return: Returns an empty hazptr_holder.

. . hazptr_holder . .

```
~hazptr_holder();
```

1. Effects: If the hazptr_holder is not empty, sets the owned hazard pointer to null and then releases ownership of it.

. . . .

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

1. Effects: If *this != other, then *this takes ownership of the pointer originally owned by other, and other becomes empty. Otherwise no effect.
2. Returns: `*this`.

. . . .

```
bool empty() const noexcept;
```

1. Returns: `true` if and only if hazptr_holder is empty. [*Note:* An empty `hazptr_holder` is different from a nonempty `hazptr_holder` that owns a hazard pointer with null value. An empty `hazptr_holder` does not own any hazard pointers. — *end note*]

. . . .

```
template <typename T>
  T* get_protected(const atomic<T*>& src) noexcept;
```

1. Requires: `*this` is not empty.
2. Effects: Equivalent to

```
T* ptr = src.load(memory_order_relaxed);
while (!try_protect(ptr, src)) {}
return ptr;
```

```
template <typename T>
  bool try_protect(T*& ptr, const atomic<T*>& src) noexcept;
```

1. Requires: `*this` is not empty.
2. Effects: Retrieves the value in `ptr`. It sets the owned hazard pointer to that value. It compares the contents of `src` for equality with the value retrieved from `ptr`. If and only if the comparison is false, the contents of `ptr` are replaced by the value read from `src` during the comparison and the owned hazard pointer is set to null. If and only if the comparison is true, performs an acquire operation on `src`.
3. Returns: The result of the comparison. [*Note:* It is possible for `try_protect` to return `true` when `ptr` has value `null`. — *end note*]
4. Complexity: Constant.

```
template <typename T>
void reset(const T* ptr) noexcept;
```

1. Requires: `*this` is not empty.
2. Effects: Sets the value of the owned hazard pointer to `ptr`.

```
void reset(nullptr_t = nullptr) noexcept;
```

1. Requires: `*this` is not empty.
2. Effects: Sets the value of the owned hazard pointer to `nullptr`.

```
void swap(hazptr_holder& other) noexcept;
```

1. Effects: Swaps the owned hazard pointer and the domain of this object with those of the other object. [*Note:* The owned hazard pointers, if any, remain unchanged during the swap and continue to protect the respective objects that they were protecting before the swap, if any. — *end note*]
2. Complexity: Constant.

. .

```
void swap(hazptr_holder& a, hazptr_holder& b) noexcept;
```

1. Effects: Equivalent to `a.swap(b)`.

.1.1. - ( )

1. RCU read-side critical sections are designated using an RAII class `std::rcu_reader`.
2. RCU-protected data structures use an intrusive scheme via the `std::rcu_obj_base` class.
3. The `std::rcu_obj_base::retire` member function invokes the specified deleter after all pre-existing `std::rcu_reader` instances have been destructed.
4. In the typical use case where a call to `std::rcu_obj_base::retire` is placed after an object is made inaccessible to readers, any object accessed within an RCU read-side critical section is guaranteed not be be reclaimed until that critical section completes. This in turn ensures that code within a critical section is ABA-safe . Objects that were removed prior to the beginning of the oldest RCU read-side critical section may be reclaimed and reused.
5. RCU protects all data that might be accessed within an RCU read-side critical section instead of protecting specific individual objects.
6. We anticipate that the constructor of `std::thread` will register each new thread with the RCU runtime, and that the destructor of `std::thread` will unregister the thread.

```
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;
void swap(rcu_reader& a, rcu_reader& b) noexcept;

// ?.2.3 function synchronize_rcu
void synchronize_rcu() 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
```

. .1, .

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, rcu_obj_base: Retire a removed object and pass the responsibility
  // for reclaiming it to the RCU library:
  void retire(
      D d = {});
};
```

. .1.1, . .

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

1. Requires: This object is no longer reachable by new RCU readers, that is, it has been removed from whatever reader-accessible linked data structure previously contained it, and that removal happens-before the `std::rcu_obj_base::retire` invocation.
2. Effects: Posts the deleter for invocation; the deleter will be invoked once all currently active RCU read-side critical sections have completed. The deleter will be invoked in the context of some implementation-specified thread of execution. [Note: This is not required to be the thread of execution that invoked the corresponding `retire`.] A pair of deleters might be executed concurrently, even if one of the corresponding `retire` functions happens-before the other.
3. Complexity: Constant.
4. Postconditions: Upon return, the callback function for the specified deleter has been posted for later invocation. At the time that deleter is invoked, pre-existing RCU read-side critical sections have completed.
5. Return: None.
6. Synchronization: Implementations may use heavy-weight synchronization mechanisms.

. . , .

This class template provides RAII RCU readers.

```
// ?.2.2, class template rcu_readers
class rcu_reader {
public:
  // ?.2.1, rcu_reader: RAII RCU readers
   rcu_reader() noexcept;
   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() noexcept;
};
```

. . .**1**, . .

`rcu_reader() noexcept;`

1. Requires: There may be implementation restrictions on nesting depth. If present, such restrictions must allow at least 100 levels of nesting.
2. Effects: Enters an RCU read-side critical section, which is exited when the current scope ends.
3. Complexity: QOI issue, but complexity should be constant in the common case.
4. Postconditions: Prevents any subsequent `retire` invocations from invoking their deleters until the current scope ends.
5. Return: None. [Note: Underlying RCU implementations in which some value must be transferred from critical-section entry to critical-section exit should stash this value in a private member of this class.]
6. Synchronization: QOI issue. High-quality implementations will make common-case use of neither locking, read-modify-write atomic operations, nor memory accesses incurring cache misses.

`rcu_reader(std::defer_lock_t) noexcept;`

1. Requires: None.
2. Effects: Creates an empty `rcu_reader`.
3. Complexity: Constant.
4. Postconditions: None.
5. Return: None.
6. Synchronization: None.

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

1. Requires: None.
2. Effects: Creates an 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 empty.
3. Complexity: QOI issue, but complexity should be constant in the common case.
4. Postconditions: The RCU read-side critical section associated with `this` no longer prevents deleter invocation.
5. Return: None.
6. Synchronization: None.

. . . , . .

The `rcu_reader` class template satisfies the requirements of `BasicLockable`.

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

1. Requires: None
2. Effects: If `this` is non-empty, the corresponding RCU read-side critical section ends as described in the destructor.
   Otherwise, if `other` is empty, no effect.
   Otherwise, `this` become active and holds the RCU read-side critical section corresponding to `other`, and `other` becomes empty.
3. Complexity: QOI issue, but complexity should be constant.
4. Postconditions: None.
5. Return: None.
6. Synchronization: None.

. . . , . .

```
~rcu_reader() noexcept;
```

1. Requires: None
2. Effects: If `this` is empty, none.
   Otherwise, `this` is non-empty, and exits the corresponding RCU read-side critical section. If an operation `A` happens before this destruction and if `A` is in turn coherence-ordered before any operation `B` that happens before a given `std::rcu_obj_base::retire` operation, and if the invocation of the corresponding `deleter` happens before an operation `C`, then any operation `D` that happens before this destruction also happens before `C`. [Note: This is considerably weaker than the actual

guarantee.]
Similarly, if an invocation of some `std::rcu_obj_base::retire`'s deleter happens before an operation `F` and if either `F` is coherence-ordered before an operation `G` such that the construction that initiated this RCU read-side critical section happens-before `G`, then any operation `H` that happens-before that `std::rcu_obj_base::retire` also happens-before any operation `I` such that the construction that initiated this RCU read-side critical section happens-before `I`. [Note: This is considerably weaker than the actual guarantee.]

3. Complexity: QOI issue, but complexity should be constant in the common case.
4. Postconditions: None.
5. Return: None.
6. Synchronization: QOI issue. High-quality implementations will make common-case use of neither locking, read-modify-write atomic operations, nor memory accesses incurring cache misses.

. . . , . .

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

1. Requires: None
2. Effects: Swaps `this` and `other` thus swapping their RCU read-side critical section states.
3. Complexity: Constant.
4. Postconditions: None.
5. Return: None.
6. Synchronization: None.

```
void swap(rcu_reader& a, rcu_reader& b) noexcept; // free function
```

1. Requires: None
2. Effects: Swaps `a` and `b` thus swapping their RCU read-side critical section states.
3. Complexity: Constant.
4. Postconditions: None.
5. Return: None.
6. Synchronization: None.

. . . , / . .

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

1. Requires: Inactive `this`. There may be implementation restrictions on nesting depth. If present, such restrictions must allow at least 100 levels of nesting.

2. Effects: Enters an RCU read-side critical section, which is exited when the current scope ends.
3. Complexity: QOI issue, but complexity should be constant in the common case.
4. Postconditions: Prevents any subsequent `retire` invocations from invoking their deleters until the current scope ends.
5. Return: None. [Note: Underlying RCU implementations in which some value must be transferred from critical-section entry to critical-section exit should stash this value in a private member of this class.]
6. Synchronization: QOI issue. High-quality implementations will make common-case use of neither locking, read-modify-write atomic operations, nor memory accesses incurring cache misses.

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

1. Requires: Active `this`.
2. Effects: Exits the corresponding RCU read-side critical section. If an operation `A` happens before this destruction and if `A` is in turn coherence-ordered before any operation `B` that happens before a given `std::rcu_obj_base::retire` operation, and if the invocation of the corresponding `deleter` happens before an operation `C`, then any operation `D` that happens before this destruction also happens before `C`. [Note: This is considerably weaker than the actual guarantee.]
   Similarly, if an invocation of some `std::rcu_obj_base::retire`'s deleter happens before an operation `F` and if either `F` is coherence-ordered before an operation `G` such that the construction that initiated this RCU read-side critical section happens-before `G`, then any operation `H` that happens-before that `std::rcu_obj_base::retire` also happens-before any operation `I` such that the construction that initiated this RCU read-side critical section happens-before `I`. [Note: This is considerably weaker than the actual guarantee.]
3. Complexity: QOI issue, but complexity should be constant in the common case.
4. Postconditions: None.
5. Return: None.
6. Synchronization: QOI issue. High-quality implementations will make common-case use of neither locking, read-modify-write atomic operations, nor memory accesses incurring cache misses.

. . , .

```
void synchronize_rcu() noexcept;
```

1. Requires: None.
2. Effects: If the beginning of a given RCU read-side critical section happens-before operation `A` and if `A` is in turn coherence-ordered before any operation `B` that happens before this `synchronize` operation, and if this `synchronize` happens before an

operation `C`, then any operation `D` that happens before the end of this RCU read-side critical section also happens before `C`. [Note: This is considerably weaker than the actual guarantee.]
Similarly, if this `synchronize` happens before an operation `F` and if `F` is coherence-ordered before an operation `G` that happens-before the end of a second RCU read-side critical section, then any operation `H` that happens-before this `synchronize` also happens-before any operation `I` such that the beginning of this same second RCU read-side critical section happens before `I`. [Note: This is considerably weaker than the actual guarantee.]

3. Complexity: Blocking, can have significant latency, but should scale well.
4. Postconditions: All pre-existing RCU read-side critical sections have completed.
5. Return: None.
6. Synchronization: Implementations may use heavyweight blocking synchronization mechanisms.

. . , .

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

1. Requires: None.
2. Effects: For each invocation of `std::rcu_obj_base::retire` that happens-before barrier, any operation `A` that is part of that `std::rcu_obj_base::retire` invocation's `deleter` happens before any operation `B` such that `barrier` happens-before `B`.
3. Complexity: Blocking, can have significant latency, but should scale well.
4. Postconditions: All pre-existing `std::rcu_obj_base::retire` operation's `deleter` invocations have completed.
5. Return: None.
6. Synchronization: Implementations may use heavyweight blocking synchronization mechanisms.

. . , .

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

1. Requires: This object is no longer reachable by new RCU readers, that is, it has been removed from whatever reader-accessible linked data structure previously contained it, and that removal happens-before the `std::retire` invocation.
2. Effects: Posts the deleter for invocation; the deleter will be invoked once all currently active RCU read-side critical sections have completed. The deleter will be invoked in the context of some implementation-specified thread of execution. [Note: This is not required to be the thread of execution that invoked the corresponding `retire`.] A pair of

deleters might be executed concurrently, even if one of the corresponding `retire` functions happens-before the other.

3. Complexity: Constant.
4. Postconditions: Upon return, the callback function for the specified deleter has been posted for later invocation. At the time that the deleter is invoked, pre-existing RCU read-side critical sections have completed.
5. Return: None.
6. Synchronization: Implementations may use heavy-weight synchronization mechanisms.

6 References

Hazptr implementation:
https://github.com/facebook/folly/blob/master/folly/experimental/hazptr/hazptr.h

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