Document number: D0566R1 `(actually P0566R1)`
Date: 20170619 (pre-Toronto)
Project: Programming Language C++, WG21, SG1,SG14, LEWG, LWG
Authors: Michael Wong, Maged M. Michael, Paul McKenney, Geoffrey Romer, Andrew Hunter
Email: michael@codeplay.com, maged.michael@acm.org, paulmck@linux.vnet.ibm.com, gromer@google.com, ahh@google.com
Reply to: michael@codeplay.com

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

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

2.1 2017-02-06 [P0566R0]

- 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

? Concurrent Data Structures Library [concur.data]

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 Concurrent Data Structures Utilities [concur.util]

1. This component provides utilities for lock-free operations that can provide safe memory access, safe memory reclamation, and ABA safety. Current shared pointer or atomic shared pointer in a concurrent environment does not guarantee any of these without special effort.

?.1.1 Concurrent Deferred Reclamation Utilities [concur.reclaim]

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 Concurrent Deferred Reclamation Utilities General [concur.reclaim.general]

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.

? Hazard Pointers [hazptr]

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.

Header <hazptr> synopsis

```
namespace std {
namespace experimental {

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

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

// ?.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 Class hazptr_domain [hazptr.domain]

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:
  constexpr explicit hazptr_domain(std::pmr::memory_resource* resource
      = std::pmr::get_default_resource());

  // 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::memory_resource* mr;  // exposition only
};
```

?.1.1 `hazptr_domain` constructors [hazptr.domain.constructor]

```
constexpr explicit hazptr_domain(
      std::memory_resource* resource = std::pmr::get_default_resource());
```

1. Requires: `resource` shall be the address of a valid memory resource.
2. Effects: Sets `mr` to `resource`.

3. Throws: Nothing.
4. Remarks: All allocation and deallocation of hazard pointers in this domain will use `*mr`. `*mr` must not be destroyed before the destruction of this domain.

?.1.2 `hazptr_domain` destructor [hazptr.domain.destructor]

```
~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.

?.2 Default `hazptr_domain` [hazptr.default_domain]

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

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

?.3 Class template `hazptr_obj_base` [hazptr.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(
      hazptr_domain& domain = default_hazptr_domain(),
      D reclaim = {});
};
```

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

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.

?.4 class hazptr_holder [hazptr.holder]

Every object of type `hazptr_holder`, throughout its lifetime, is guaranteed to *own* exactly one hazard pointer.

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

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

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

  // ?.4.3, get_protected
  template <typename T>
    T* get_protected(const atomic<T*>& src) noexcept;

  // ?.4.4, try_protect
  template <typename T>
    bool try_protect(T*& ptr, const atomic<T*>& src) noexcept;

  // ?.4.5, reset
  template <typename T>
  void reset(const T* ptr) noexcept;
  void reset(nullptr_t = nullptr) noexcept;

  // ?.4.6, swap
  void swap(hazptr_holder&) noexcept;
```

```
};
```

?.4.1 `hazptr_holder` constructor [hazptr.holder.constructor]

```
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()`.

?.4.2 `hazptr_holder` destructor [hazptr.holder.destructor]

```
~hazptr_holder();
```

1. Effects: Sets the owned hazard pointer to null and then releases ownership of it.
2. Requires: Must be invoked in the same thread that constructed `*this`.

?.4.3, hazptr_holder get_protected [hazptr.holder.get_protected]

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

1. Effects: Equivalent to

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

?.4.4 hazptr_holder try_protect [hazptr.holder.try_protect]

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

1. 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`.
2. Returns: The result of the comparison.
3. Complexity: Constant.

?.4.5 hazptr_holder reset [hazptr.holder.reset]

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

1. Effects: Sets the value of the owned hazard pointer to `ptr`.

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

1. Effects: Sets the value of the owned hazard pointer to `nullptr`.

?.4.6 hazptr_holder swap[hazptr.holder.swap]

```
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 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.

?.5 hazptr_holder Swap hazptr_holder [hazptr.swap_owners]

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

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

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

1. RCU read-side critical sections each begin with an `rcu_domain::read_lock()` and end with the matching `rcu_domain::read_unlock()`. A call to `rcu_domain::synchronize()` will wait until all pre-existing RCU read-side critical sections have completed.
2. In the typical use case where a call to `rcu_domain::synchronize()` is placed after an object is made inaccessible to readers, but before it is reclaimed, 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.
3. RCU protects all data that might be accessed within an RCU read-side critical section instead of protecting specific objects.

Header <rcu> synopsis

```
namespace std {
namespace experimental {

//  ?.1, class rcu_domain: Each domain manages an
// independent set of RCU read-side critical sections and grace periods:
class rcu_domain {
 public:
```

```
// ?.1.1, constructors:
  constexpr explicit rcu_domain() noexcept;

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

// ?.1.2, destructor:
  ~rcu_domain();

// ?.1.3, rcu_domain thread management:
  virtual void register_thread() = 0;
  virtual void unregister_thread() = 0;
  static bool constexpr register_thread_needed();
  virtual void quiescent_state() noexcept = 0;
  virtual void thread_offline() noexcept = 0;
  virtual void thread_online() noexcept = 0;
  static constexpr bool quiescent_state_needed();

// ?.1.4, rcu_domain read-side critical sections:
  virtual void read_lock() noexcept = 0;
  virtual void read_unlock() noexcept = 0;

// ?.1.5, rcu_domain grace periods:
  virtual void synchronize() noexcept = 0;
  virtual void retire(rcu_head *rhp, void (*cbf)(rcu_head *rhp)) = 0; // rcu_head for exposition only
  virtual void barrier() noexcept = 0;
};

// ?.2, class template rcu_obj_base
template<typename T, typename D = default_delete<T>, bool E = is_empty<D>::value>
    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(
      rcu_domain& rd,
      D d = {});
  void retire(
      D d = {});
};
```

```
// ?.3, class template rcu_guard
    class rcu_guard {
public:
  // ?.3.1, rcu_guard: RCU reader as guard
   rcu_guard() noexcept;
   explicit rcu_guard(rcu_domain *rd);
   rcu_guard(const rcu_guard &) = delete;
   rcu_guard&operator=(const rcu_guard &) = delete;
   ~rcu_guard() noexcept;
};
} // namespace experimental
} // namespace std
```

?.1 **Class rcu_domain [rcu.rcu_domain]**

An rcu_domain manages a set of interacting RCU read-side critical sections and grace periods. A domain is responsible for reclaiming objects retired to it (i.e., objects retired to this domain), once the last RCU read-side critical section that was in existence at retirement time has ended.

```
class rcu_domain {
 public:

// ?.1.1, constructor:
  constexpr explicit rcu_domain() noexcept;

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

// ?.1.2, destructor:
  ~rcu_domain();

// ?.1.3, rcu_domain thread management:
  virtual void register_thread() = 0;
  virtual void unregister_thread() = 0;
  static constexpr bool register_thread_needed();
  virtual void quiescent_state() noexcept = 0;
```

```
  virtual void thread_offline() noexcept = 0;
  virtual void thread_online() noexcept = 0;
  static constexpr bool quiescent_state_needed();

// ?.1.4, rcu_domain read-side critical sections:
  virtual void read_lock() noexcept = 0;
  virtual void read_unlock() noexcept = 0;

// ?.1.5, rcu_domain grace periods:
  virtual void synchronize() noexcept = 0;
  virtual void retire(rcu_head *rhp, void (*cbf)(rcu_head *rhp)) = 0;
  virtual void barrier() noexcept = 0;
};
```

?.1.1 rcu_domain constructor [rcu.rcu_domain.constructor]

```
  constexpr explicit rcu_domain() noexcept;
```

1. Requires: This instance of rcu_domain must not yet have been constructed.
2. Effects: Allocates any memory required by this rcu_domain instance.
3. Postconditions: The rcu_domain instance is ready for use.
4. Return: None.
5. Synchronization: Implementations may use locking.

?.1.2 destructor [rcu.rcu_domain.destructor]

```
  ~rcu_domain();
```

1. Requires: This instance of rcu_domain must have been constructed and there must no longer be any threads using this rcu_domain_instance.
2. Effects: Deallocates any memory used by this rcu_domain instance.
3. Complexity: O(n) where n is the number of threads still registered with this rcu_domain instance.
4. Postconditions: This rcu_domain instance has been destroyed and is no longer usable.
5. Return: None.
6. Synchronization: Implementations may use locking.

?.1.3 rcu_domain thread management [rcu.rcu_domain.threads]

```
  static constexpr bool register_thread_needed();
```

1. Requires: Nothing

2. Effects: None.
3. Complexity: Constant.
4. Postconditions: None.
5. Return: std::true if each thread using RCU read-side critical sections must use rcu_domain::register_thread() before their first invocation of rcu_read_lock().
6. Synchronization: N/A.

virtual void register_thread() = 0;

1. Requires: The current thread has invoked rcu_domain::unregister_thread() since its last call to rcu_domain::register_thread().
2. Effects: When rcu_domain::register_thread_needed() returns std::true, this method will allocate any needed per-thread storage.. When rcu_domain::quiescent_state_needed() returns std::false, no effect.
3. Complexity: Constant.
4. Postconditions: The thread is permitted to invoke rcu_read_lock() and rcu_read_unlock().
5. Return: None.
6. Synchronization: Implementations for which rcu_domain::register_thread_needed() returns true may use locking.

virtual void unregister_thread() = 0;

1. Requires: The current thread has invoked rcu_domain::register_thread() since its last call to rcu_domain::unregister_thread().
2. Effects: When rcu_domain::register_thread_needed() returns std::true, this method will deallocate any needed per-thread storage.. When rcu_domain::quiescent_state_needed() returns std::false, no effect.
3. Complexity: Constant.
4. Postconditions: The thread is forbidden from invoking rcu_read_lock() and rcu_read_unlock().
5. Return: None.
6. Synchronization: Implementations for which rcu_domain::register_thread_needed() returns true may use locking.

virtual void quiescent_state() noexcept = 0;

1. Requires: All of the following:
    a. The current thread has invoked rcu_domain::thread_online() since its last call to rcu_domain::thread_offline().
    b. The current thread has invoked rcu_domain::register_thread() since its last call to rcu_domain::unregister_thread().

    c. When rcu_domain::register_thread_needed() returns true, the current thread has invoked rcu_domain::register_thread() since its creation.
2. Effects: When rcu_domain::quiescent_state_needed() returns std::true, this method will report a quiescent state to RCU for the current grace period. When rcu_domain::quiescent_state_needed() returns std::false, no effect.
3. Complexity: Constant.
4. Postconditions: None.
5. Return: None.
6. Synchronization: Implementations for which rcu_domain::quiescent_state_needed() returns true may use locking.

virtual void thread_offline() noexcept = 0;

1. Requires: All of the following:
    a. The current thread has invoked rcu_domain::rcu_read_unlock() as many times as it has invoked rcu_domain::rcu_read_lock(), that is, the thread must not be in an RCU read-side critical section.
    b. The current thread has invoked rcu_domain::thread_online() since its last call to rcu_domain::thread_offline().
    c. The current thread has invoked rcu_domain::register_thread() since its last call to rcu_domain::unregister_thread().
    d. When rcu_domain::register_thread_needed() returns true, the current thread has invoked rcu_domain::register_thread() since its creation.
2. Effects: When rcu_domain::quiescent_state_needed() returns std::true, RCU will no longer consider this thread when computing grace periods, so that this thread need not invoke rcu_domain::quiescent_state(). When rcu_domain::quiescent_state_needed() returns std::false, no effect.
3. Complexity: Constant.
4. Postconditions: The thread is forbidden from invoking rcu_read_lock() and rcu_read_unlock().
5. Return: None.
6. Synchronization: Implementations for which rcu_domain::quiescent_state_needed() returns true may use locking.

virtual void thread_online() noexcept = 0;

1. Requires: The current thread has previously invoked rcu_domain::thread_offline().
2. Effects: When rcu_domain::quiescent_state_needed() returns std::true, RCU will now consider this thread when computing grace periods, so that this thread must now periodically invoke rcu_domain::quiescent_state() until the next call to rcu_domain::thread_offline(). When rcu_domain::quiescent_state_needed() returns std::false, no effect.
3. Complexity: Constant.

4. Postconditions: The thread is permitted to invoke rcu_read_lock() and rcu_read_unlock().
5. Return: None.
6. Synchronization: Implementations for which rcu_domain::quiescent_state_needed() returns true may use locking.

static constexpr bool quiescent_state_needed();

1. Requires: Nothing
2. Effects: None.
3. Complexity: Constant.
4. Postconditions: None.
5. Return: std::true if each thread using RCU read-side critical sections must periodically invoke rcu_domain::quiescent_state() on the one hand, or invoke rcu_domain::thread_offline() to indicate an extended quiescent state on the other.
6. Synchronization: N/A.

?.1.4 rcu_domain read-side critical sections [rcu.rcu_domain.readers]

virtual void read_lock() noexcept = 0;

1. Requires: All of the following:
    a. The current thread has invoked rcu_domain::thread_online() since its last call to rcu_domain::thread_offline().
    b. The current thread has invoked rcu_domain::register_thread() since its last call to rcu_domain::unregister_thread().
    c. When rcu_domain::register_thread_needed() returns true, the current thread has invoked rcu_domain::register_thread() since its creation.
2. Effects: Enters an RCU read-side critical section.
3. Complexity: Constant.
4. Postconditions: Prevents any subsequent RCU grace periods from completing.
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.

virtual void read_unlock() noexcept = 0;

1. Requires: All of the following:
    a. The current thread has invoked rcu_domain::thread_online() since its last call to rcu_domain::thread_offline().
    b. The current thread has invoked rcu_domain::register_thread() since its last call to rcu_domain::unregister_thread().

    c. When rcu_domain::register_thread_needed() returns true, the current thread has invoked rcu_domain::register_thread() since its creation.
2. Effects: If the thread has invoked rcu_domain::read_unlock() as many times as it has invoked rcu_domain::read_lock(), counting this invocation, exits an RCU read-side critical section.
3. Complexity: Constant.
4. Postconditions: If this invocation resulted in an exit from an RCU read-side critical section, stops preventing RCU grace periods from completing.
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.

?.1.5, rcu_domain grace periods [rcu.rcu_domain.grace_periods]

virtual void synchronize() noexcept = 0;

1. Requires: The current thread has invoked rcu_domain::rcu_read_unlock() as many times as it has invoked rcu_domain::rcu_read_lock(), that is, the thread must not be in an RCU read-side critical section.
2. Effects: Waits for an RCU grace period to elapse, that is, waits for all pre-existing RCU read-side critical sections to complete.
3. Complexity: Blocking. As a general rule, per-invocation overhead increases with increasing number of threads, and decreases with increasing numbers of concurrent calls to rcu_domain::synchronize(), rcu_domain::retire(), and rcu_obj_base::retire() in the case all these calls use the same instance of rcu_domain.
4. Postconditions: All pre-existing RCU read-side critical sections have completed.
5. Return: None.
6. Synchronization: Implementations may use heavy-weight synchronization mechanisms.

virtual void retire(rcu_head *rhp, void (*cbf)(rcu_head *rhp)) = 0;

1. Requires: All of the following:
    a. The current thread has invoked rcu_domain::thread_online() since its last call to rcu_domain::thread_offline().
    b. The current thread has invoked rcu_domain::register_thread() since its last call to rcu_domain::unregister_thread().
    c. When rcu_domain::register_thread_needed() returns true, the current thread has invoked rcu_domain::register_thread() since its creation.
2. Effects: After a subsequent RCU grace period elapses, invoke cbf(rhp).
3. Complexity: Constant.

4. Postconditions: Upon return, the callback function cbf has been posted for later invocation. At the time that cbf(rhp) is invoked, pre-existing RCU read-side critical sections have completed.
5. Return: None.
6. Synchronization: Implementations may use heavy-weight synchronization mechanisms.
7. Remark: We expect that most C++ developers will use rcu_obj_base::retire() in preference to rcu_domain::retire(). However, there are RCU use cases for which there is no rcu_obj_base, in which case rcu_domain::retire() is useful.

```
virtual void barrier() noexcept = 0;
```

1. Requires: All of the following:
    a. The current thread has invoked rcu_domain::rcu_read_unlock() as many times as it has invoked rcu_domain::rcu_read_lock(), that is, the thread must not be in an RCU read-side critical section.
    b. The current thread has invoked rcu_domain::thread_online() since its last call to rcu_domain::thread_offline().
    c. The current thread has invoked rcu_domain::register_thread() since its last call to rcu_domain::unregister_thread().
    d. When rcu_domain::register_thread_needed() returns true, the current thread has invoked rcu_domain::register_thread() since its creation.
2. Effects: Wait for the invocation of all pre-existing callbacks from rcu_domain::retire() and rcu_obj_base::retire() for this instance of rcu_domain.
3. Complexity: Blocking. As a general rule, the greater number of concurrent invocations of rcu_domain::barrier() for a given instance of rcu_domain, the lower the per-invocation overhead.
4. Postconditions: At the time that cbf(rhp) is invoked, pre-existing RCU read-side critical sections have completed.
5. Return: None.
6. Synchronization: Implementations may use heavy-weight synchronization mechanisms.

?.2, class template rcu_obj_base [rcu.rcu_obj_base]

Objects of type T to be protected by RCU inherit from rcu_obj_base<T>.

```
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(
     rcu_domain& rd,
     D d = {});
```

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

?.2.1, rcu_obj_base retire [rcu.rcu_obj_base.retire]

```
  void retire(
     rcu_domain& rd,
     D d = {});
  void retire(
     D d = {});
```

1. Requires: All of the following, given the instance of rcu_domain used by this instance of rcu_obj_base:
    a. The current thread has invoked rcu_domain::thread_online() since its last call to rcu_domain::thread_offline().
    b. The current thread has invoked rcu_domain::register_thread() since its last call to rcu_domain::unregister_thread().
    c. When rcu_domain::register_thread_needed() returns true, the current thread has invoked rcu_domain::register_thread() since its creation.
2. Effects: After a subsequent RCU grace period elapses, invoke the deleter on this object.
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.

?.3, class template rcu_guard [rcu.rcu_guard]

This class template provides scoped RCU-reader guard capability.

```
// ?.3, class template rcu_guard
    class rcu_guard {
public:
  // ?.3.1, rcu_guard: RCU reader as guard
   rcu_guard() noexcept;
   explicit rcu_guard(rcu_domain *rd);
   rcu_guard(const rcu_guard &) = delete;
   rcu_guard&operator=(const rcu_guard &) = delete;
   ~rcu_guard() noexcept;
```

1. Requires: All of the following:

    a. The current thread has invoked rcu_domain::thread_online() since its last call to rcu_domain::thread_offline().
    b. The current thread has invoked rcu_domain::register_thread() since its last call to rcu_domain::unregister_thread().
    c. When rcu_domain::register_thread_needed() returns true, the current thread has invoked rcu_domain::register_thread() since its creation.
2. Effects: Enters an RCU read-side critical section, which is exited when the current scope ends.
3. Complexity: Constant.
4. Postconditions: Prevents any subsequent RCU grace periods from completing until the current scope ends.
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.

6 References

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

[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