Document number: P0566R0
Date: 20170206
Project: Programming Language C++, WG21, SG1,SG14, LEWG, LWG
Authors: Michael Wong, Maged M. Michael, Paul McKenney
Email: michael@codeplay.com, maged.michael@acm.org, paulmck@linux.vnet.ibm.com
Reply to: michael@codeplay.com

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

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]

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. It seems to make sense to start a new clause that covers the specific topic of concurrent data structures. This is because we anticipate there will be additional contributions in the form of concurrent queues as data structures.

This wording assumes undecided-as-yet clause number 31 as a placeholder and a new clause designation named Concurrent Data Structures Library. In particular, we believe that the Concurrent Data Structure clause should contain 2 sub-clauses:

31.1 Concurrent Data Structures Utilities
31.2 Concurrent Data Structures

where Concurrent Data Structures Utilities should contain the lock-based or lockless programming tools that enable end-to-end lock-free programming including lock-free deferred reclamation.

Proposed wording

Add new clause 31 as follows:
31 Concurrent Data Structures Library [concur]

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 either it must be completely immutable so the data never changes and no synchronization is necessary, or the program must be designed to ensure that changes are correctly synchronized between threads.

3. While one option is to use a separate mutex and external locking to protect the data as in clause 30 [thread], another is to design the data structure itself for concurrent access as is described in this clause.
4. If you can write data structures that are safe for concurrent access without locks, then there's the potential to avoid these limitation. Such a data structure is called a lockless data structure. (Please note that "lockless" differs from "lock-free" in that lock freedom implies forward-progress guarantees and linearizability which are not necessarily present in lockless techniques.)
5. Alternatively, fine-grained locking techniques can be used to achieve good performance and scalability, with the canonical example being hash tables.

Add new clause 31.1 as follows:

31.1 Concurrent Data Structures Utilities [concur.util]

1. This component provides utilities for lock-free operations that can resolve the problems of unsafe memory access, the ABA problem, (https://en.wikipedia.org/wiki/ABA_problem) and unsafe memory reclamation. These are all issues that can be a problem even with shared pointer or atomic shared pointer in a concurrent environment.

Add new clause 31.1.1 as follows

31.1.1 Concurrent Deferred Reclamation Utilities [concur.util.reclaim]

1. The following subclauses describe utilities to manage deferred reclamation and retire operations, as summarized in Table 142. These differ from shared_ptr in that they do not reclaim or retire their objects automatically, rather it is under user control.

Table 142 - Concurrent Data Structure Deferred Reclamation Utilities Summary

	Subclause	Header(s)
31.1.1.2	Requirements	
31.1.1.3	Hazard Pointers	<hazptr>
31.1.1.4	Read-Copy-Update	<rcu>

Add new clause 31.1.1.1 as follows:

31.1.1.1 Concurrent Deferred Reclamation Utilities General [concur.util.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., delete) 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.

Add new clause 31.1.1.2 as follows:

31.1.1.2 Requirements [concur.util.reclaim.req]

1. Deferred reclamation utilities differ from shared_ptr [util.smartptr.shared] in that they do not offer automatic reclamation. As such both hazard_ptr [concur.util.reclaim.hazptr] and RCU [concur.util.reclaim.rcu] offer the retire operation in common for retiring an object and pass the responsibility for reclaiming it to the library.
2. In the case of hazard pointers, any remaining objects that were retired to this domain are guaranteed not to be protected by hazard pointers that belong to this domain, and are consequently reclaimed.
3. In the case of RCU, 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.

Add new clause 31.1.1.3 as follows:

31.1.1.3 Hazard Pointers [concur.util.reclaim.hazptr]

1. To use the Hazard pointer system, objects or data structures can be protected by user threads that write to hazard pointers to protect these objects, or access them to provide ABA-safe (https://en.wikipedia.org/wiki/ABA_problem) comparison while removers shall read the hazard pointers to decide when it is safe to reclaim the removed objects. Objects that are not referenced by hazard pointers are free for reuse and reallocation.
2. 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) to protect such objects from being reclaimed.
3. 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.
4. The hazard pointers method 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.

Header <hazptr> synopsis

```
namespace std {
namespace experimental {
namespace hazptr {

//  31.1.1.3.1, Class hazptr_domain: Class of hazard pointer domains.
```

```
// Each domain manages a set of hazard pointers and a set of retired
objects.
class hazptr_domain {
 public:

  // 31.1.1.3.1.1, constructors:
  constexpr explicit hazptr_domain(
      std::memory_resource* = std::get_default_resource()) noexcept;

  // 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;

  // 31.1.1.3.1.2, destructor:
  ~hazptr_domain();
};

// 31.1.1.3.2, hazptr_domain get default:
hazptr_domain& default_hazptr_domain() noexcept;

// 31.1.1.3.3, Class template hazptr_obj_base:
template <typename T, typename D = std::default_delete<T>>
class hazptr_obj_base {
 public:
  // 31.1.1.3.3.1, hazptr_obj_base: Retire a removed object and pass the
  // responsibility for reclaiming it to the hazptr library:
  void retire(
      hazptr_domain& domain = default_hazptr_domain(),
      D reclaim = {});
};

// 31.1.1.3.4, class template  hazptr_owner: automatic acquisition and
// release of hazard pointers, and interface for hazard pointer operations:
template <typename T> class hazptr_owner {
 public:
  // 31.1.1.3.4.1, Constructor automatically acquires a hazard pointer:
  explicit hazptr_owner(hazptr_domain& domain = default_hazptr_domain());

  // disallow copy and move constructors and assignment operators because:
  // - each hazptr_owner owns exactly one hazard pointer at any time.
  // - each hazard pointer may have up to one owner at any time. */
```

```
  hazptr_owner(const hazptr_owner&) = delete;
  hazptr_owner(hazptr_owner&&) = delete;
  hazptr_owner& operator=(const hazptr_owner&) = delete;
  hazptr_owner& operator=(hazptr_owner&&) = delete;

  // 31.1.1.3.4.2, Destructor automatically clears and releases the owned
  // hazard pointer:
  ~hazptr_owner();

  // 31.1.1.3.4.3, hazptr_owner: Returns a protected pointer from the
source:
  template <typename A = std::atomic<T*>>
  T* get_protected(const A& src) noexcept;

  // 31.1.1.3.4.4, hazptr_owner: setting the hazard pointer.  Otherwise
sets
  // ptr to src.
  template <typename A = std::atomic<T*>>:
  bool try_protect(T*& ptr, const A& src) noexcept;

  // 31.1.1.3.4.5, hazptr_owner:  Set the hazard pointer to ptr:
  void set(const T* ptr) noexcept;

  // 31.1.1.3.4.6, hazptr_owner: Clear the hazard pointer:
  void clear() noexcept;

  // 31.1.1.3.4.7, hazptr_owner: Swap ownership of hazard pointers between
  // hazptr_owners:
  void swap(hazptr_owner&) noexcept;
};

// 31.1.1.3.5, hazptr_owner: Swap two hazptr_owner<T> objects:
template <typename T>
void swap(hazptr_owner<T>&, hazptr_owner<T>&) noexcept;

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

31.1.1.3.1 **Class hazptr_domain [concur.util.reclaim.hazptr.hazptr_domain]**

1. A hazard pointer domain owns a set of hazard pointers. A domain is responsible for reclaiming objects retired to it (i.e., objects retired to this domain), when such objects are

not protected by hazard pointers that belong to this domain (including when this domain is destroyed).

```
class hazptr_domain {
 public:

  // 31.1.1.3.1.1, constructor:
  constexpr explicit hazptr_domain(
      std::memory_resource* = std::get_default_resource()) noexcept;

  // 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;

  // 31.1.13.1.2, destructor:
  ~hazptr_domain();
};
```

31.1.1.3.1.1 hazptr_domain constructors [concur.util.reclaim.hazptr.hazptr_domain.constructor]

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

1. Requires: The memory resource must be valid.
2. Effects: Sets the memory resource for this domain to the specified memory resource.
3. Postconditions: All allocation and deallocation of hazard pointers throughout the lifetime of this domain will use the specified memory resource.The memory resource must not be destroyed before the destruction of this domain.
4. Complexity: Constant.

31.1.1.3.1.2 hazptr_domain destructor [concur.util.reclaim.hazptr.hazptr_domain.destructor]

```
~hazptr_domain();
```

1. Requires: All uses of hazard pointers that belong to this domain and all retirements of objects to this domain have ended. The memory resource associated with this domain must not be destroyed before the destruction of this domain.
2. Effects: Deallocate all hazard pointer storage used by this domain. Consequently any remaining objects that were retired to this domain are guaranteed not to be protected by hazard pointers that belong to this domain, and are consequently reclaimed.

31.1.1.3.2 hazptr_domain get default **[concur.util.reclaim.hazptr.default_hazptr_domain]**

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

1. Effects:If called for the first time, constructs the default hazptr_domain.
2. Returns: A reference to the default hazptr_domain.
3. Complexity: Constant.

31.1.1.3.3 Class template hazptr_obj [concur.util.reclaim.hazptr.hazptr_obj_base]
Type T of objects to be protected by hazard pointers inherits from hazptr_obj_base<T>.

```
template <typename T, typename D = std::default_delete<T>>
class hazptr_obj_base {
 public:
  // 31.1.1.3.3.1, hazptr_obj_base:  Retire a removed object and pass the
  // responsibility for reclaiming it to the hazptr library
  void retire(
      hazptr_domain& domain = default_hazptr_domain(),
      D reclaim = {});
};
```

31.1.1.3.3.1 hazptr_obj_base retire[concur.util.reclaim.hazptr.hazptr_obj_base.retire]
```
void retire(hazptr_domain& domain = default_hazptr_domain(), D reclaim =
{});
```

1. Requires: The specified domain and reclaimer must be valid.
2. Effects: The specified domain becomes responsible for using the specified reclaimer to reclaim the object, when none of the hazard pointers that belong to the domain has been continuously pointing to the object since before this call.

31.1.1.3.4 class template hazptr_owner **[concur.util.reclaim.hazptr.hazptr_owner]**
Every object of type hazptr_owner<T>, throughout its lifetime, is guaranteed to own exactly one hazard pointer capable of protecting objects of type T, derived from hazptr_obj_base<T>.

```
template <typename T> class hazptr_owner {
 public:
  // 31.1.1.3.4.1, Constructor automatically acquires a hazard pointer:
  explicit hazptr_owner(hazptr_domain& domain = default_hazptr_domain());

  // disallow copy and move constructors and assignment operators because:
  // - each hazptr_owner owns exactly one hazard pointer at any time.
  // - each hazard pointer may have up to one owner at any time. */
  hazptr_owner(const hazptr_owner&) = delete;
  hazptr_owner(hazptr_owner&&) = delete;
  hazptr_owner& operator=(const hazptr_owner&) = delete;
```

```
  hazptr_owner& operator=(hazptr_owner&&) = delete;

  // 31.1.1.3.4.2, Destructor automatically clears and releases the owned
  // hazard pointer:
  ~hazptr_owner();

  // 31.1.1.3.4.3, hazptr_owner: Returns a protected pointer from the
source:
  template <typename A = std::atomic<T*>>
  T* get_protected(const A& src) noexcept;

  // 31.1.1.3.4.4, hazptr_owner: setting the hazard pointer.  Otherwise
sets
  // ptr to src.
  template <typename A = std::atomic<T*>>:
  bool try_protect(T*& ptr, const A& src) noexcept;

  // 31.1.1.3.4.5, hazptr_owner:  Set the hazard pointer to ptr:
  void set(const T* ptr) noexcept;

  // 31.1.1.3.4.6, hazptr_owner: Clear the hazard pointer:
  void clear() noexcept;

  // 31.1.1.3.4.7, hazptr_owner: Swap ownership of hazard pointers between
  // hazptr_owners:
  void swap(hazptr_owner&) noexcept;
};
```

31.1.1.3.4.1 hazptr_owner constructor [concur.util.reclaim.hazptr.hazptr_owner.constructor]

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

1. Requires: The specified domain must be valid.
2. Effects: Acquire a hazard pointer that belongs to the specified domain. The acquired hazard pointer may be newly allocated or previously released. The specified memory_resource for the domain is used to allocate a new hazard pointer if needed.
3. Throws: May throw only whatever the memory_resource of the specified domain may throw. That is, if the specified domain's memory_resource does not throw, then this constructor shall not throw.

31.1.1.3.4.2 hazptr_owner destructor [concur.util.reclaim.hazptr.hazptr_owner.destructor]

```
~hazptr_owner();
```

1. Effects: Equivalent to calling clear() and then releasing ownership of the owned hazard pointer.

31.1.1.3.4.3, hazptr_owner get_protect [concur.util.reclaim.hazptr.hazptr_owner.get_protected]

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

1. Effects: It retrieves a value from src, sets the value of the owned hazard pointer to that value, and then validates that src holds that value (i.e., compares the contents of src for equality). If the validation fails, it starts over. It keeps repeating these steps until the validation succeeds.
2. Postconditions: If the returned value is not std::nullptr, it guarantees that as long as the hazard pointer remains unchanged, the object pointed to by the return value will not be reclaimed, provided that the reclamation of the object will be requested only by calling the object's `retire()` member function.
3. Returns: The successfully validated value retrieved from src.
4. Complexity: Constant if src is not concurrently changed.

31.1.1.3.4.4 hazptr_owner try_protect [concur.util.reclaim.hazptr.hazptr_owner.try_protect]

```
template <typename A = std::atomic<T*>>
bool try_protect(T*& ptr, const A& 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 then, the contents of ptr are replaced by the value read from src during the comparison.
2. Postconditions: If returns true and ptr is not std::nullptr, then it it guarantees that as long as the hazard pointer remains unchanged, the object pointed to by ptr will not be reclaimed, provided that the reclamation of the object will be requested only by calling the object's `retire()` member function.
3. Returns: The result of the comparison.
4. Complexity: Constant.

31.1.1.3.4.5 hazptr_owner set [concur.util.reclaim.hazptr.hazptr_owner.set]

```
void set(const T* ptr) noexcept;
```

1. Effects: Sets the value of the owned hazard pointer to the value ptr.
2. Postconditions If ptr points to an object and the object is not yet retired (i.e., a call to retire() has not yet been invoked for the object), then the object will not be reclaimed as long as the owned hazard pointer continues to hold the value ptr.
3. Complexity: Constant.

31.1.1.3.4.6 hazptr_owner clear [concur.util.reclaim.hazptr.hazptr_owner.clear]

```
void clear() noexcept;
```

1. Effects: Sets the owned hazard pointer to std::nullptr.
2. Complexity: Constant.

31.1.1.3.4.7 hazptr_owner swap[concur.util.reclaim.hazptr.hazptr_owner.swap]

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

1. Effects: Swaps the owned hazard pointer of this object with those of the other object. Note that 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.
2. Complexity: Constant.

31.1.1.3.5 hazptr_owner Swap hazptr_owner **[concur.util.reclaim.hazptr.swap_owners]**

```
template <typename T>
void swap(hazptr_owner<T>& a, hazptr_owner<T>& b) noexcept;
```

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

Add new clause 31.1.1.4 as follows:

31.1.1.4 Read-Copy Update (RCU) [concur.util.reclaim.rcu]

1. To use the RCU, critical code is enclosed within RCU read-side critical sections that 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 between removal of an object and its reclamation, 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 (https://en.wikipedia.org/wiki/ABA_problem). Objects that were removed prior to the beginning of the oldest RCU read-side critical section may be reclaimed and reused. (Note: There are a great many other use cases, but this one is the most common.)
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 {
namespace rcu {
```

```
//  31.1.1.4.1, class rcu_domain: Each domain manages an
// independent set of RCU read-side critical sections and grace periods:
class rcu_domain {
 public:

// 31.1.1.4.1.1, 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;

// 31.1.1.4.1.2, destructor:
  ~rcu_domain();

// 31.1.1.4.1.3, 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();

// 31.1.1.4.1.4, rcu_domain read-side critical sections:
  virtual void read_lock() noexcept = 0;
  virtual void read_unlock() noexcept = 0;

// 31.1.1.4.1.5, 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;
};

// 31.1.1.4.2, class template rcu_obj_base
template<typename T, typename D = default_delete<T>, bool E = is_empty<D>::value>
    class rcu_obj_base {
public:
  // 31.1.1.4.2.1, 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 = {});
};

// 31.1.1.4.3, class template rcu_guard
    class rcu_guard {
public:
  // 31.1.1.4.3.1, 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 rcu
} // namespace experimental
} // namespace std
```

31.1.1.4.1 **Class rcu_domain [concur.util.reclaim.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:

// 31.1.1.4.1.1, 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;

// 31.1.1.4.1.2, destructor:
```

```
  ~rcu_domain();

// 31.1.1.4.1.3, 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();

// 31.1.1.4.1.4, rcu_domain read-side critical sections:
  virtual void read_lock() noexcept = 0;
  virtual void read_unlock() noexcept = 0;

// 31.1.1.4.1.5, 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;
};
```

31.1.1.4.1.1 rcu_domain constructor [concur.util.reclaim.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. Complexity: Constant.
4. Postconditions: The rcu_domain instance is ready for use.
5. Return: None.
6. Synchronization: Implementations may use locking.

31.1.1.4.1.2 destructor [concur.util.reclaim.rcu.rcu_domain.destructor]

```
  ~rcu_domain();
```

1. Requires: This instance of rcu_domain must have been constructed.
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: There must no longer be any threads using this rcu_domain instance.
5. Return: None.

6. Synchronization: Implementations may use locking.

31.1.1.4.1.3 rcu_domain thread management [concur.util.reclaim.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.

31.1.1.4.1.4 rcu_domain read-side critical sections [concur.util.reclaim.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.

31.1.1.4.1.5, rcu_domain grace periods [concur.util.reclaim.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.

31.1.1.4.2, class template rcu_obj_base **[concur.util.reclaim.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:
  // 31.1.1.4.2.1, 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 = {});
};
```

31.1.1.4.2.1, rcu_obj_base retire [concur.util.reclaim.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.

31.1.1.4.3, class template rcu_guard **[concur.util.reclaim.rcu.rcu_guard]**

This class template provides scoped RCU-reader guard capability.

```
// 31.1.1.4.3, class template rcu_guard
    class rcu_guard {
public:
  // 31.1.1.4.3.1, 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.

Acknowledgement

The author wishes to thank Geoffrey Romer and Andrew Hunter.

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