Document number: P1022R0 (Informational)
Date: 20180420 (p)
Project: Programming Language C++, WG21, SG1, SG14, LEWG, LWG
Authors: Michael Wong, Maged M. Michael, Paul McKenney, Geoffrey Romer, Andrew Hunter, Arthur O'Dwyer, David S. Hollman, JF Bastien, Hans Boehm, David Goldblatt
Email: michael@codeplay.com, maged.michael@acm.org, paulmck@linux.vnet.ibm.com, gromer@google.com, ahh@google.com, arthur.j.odwyer@gmail.com, dshollm@sandia.gov, jfbastien@apple.com, hboehm@google.com, davidtgoldblatt@gmail.com
Reply to: michael@codeplay.com

Material for 2018 JAX Discussions of Hazard Pointer and Read-Copy-Update (RCU)

This informational document records some material for discussions of Hazard Pointers and RCU at the 2018 Jacksonville C++ meeting. It also includes notes from the ensuing discussion. These materials were initially kept at the end of D0566R5, but needed to be removed from that document for the next mailing. This document therefore preserves these materials for posterity.

Discussion in JAX: Hazard Pointer Patterns

R	Reader
W	Remover (Writer)
D	Reclaimer (Deleter)

ptr	pointer to protected object
src	points to reachable (not removed) object
P	hazard pointer owned by Reader
h	Reader's hazptr_holder that owns P

Library calls in bold
Library steps in italics

Note: Writer (Remover) removes ptr by storing a value != ptr to src.

Pattern 1: Object protected using hazptr_holder::try_protect()

R (Reader)	Writer/Deleter (Remover/Reclaimer)
R1: h.try_protect(ptr, src) *// R1a: P.store(ptr, rlx)* *// R1b: seq_cst fence* *// R1c: src.load(rlx) == ptr*	W1: src.store(null, rel); // app
R2: <use *ptr> // app	/* App guarantees that src != ptr between ptr->retire() and reclamation of *ptr */ **W2: ptr->retire()** */* retire() hb reclamation attempt on the same or another thread */*
	// D1a: <seq_cst fence> *// D1b: P.load(acq) == ptr* *// D1c: <don't reclaim *ptr>*
R3: h.reset() *// R3: P.store(null,rel)*	*// D2a: <seq_cst fence>* *// D2b: P.load(acq) != ptr* *// D2c: <reclaim *ptr>*

Pattern 2: Object protected using hazptr_holder::reset()

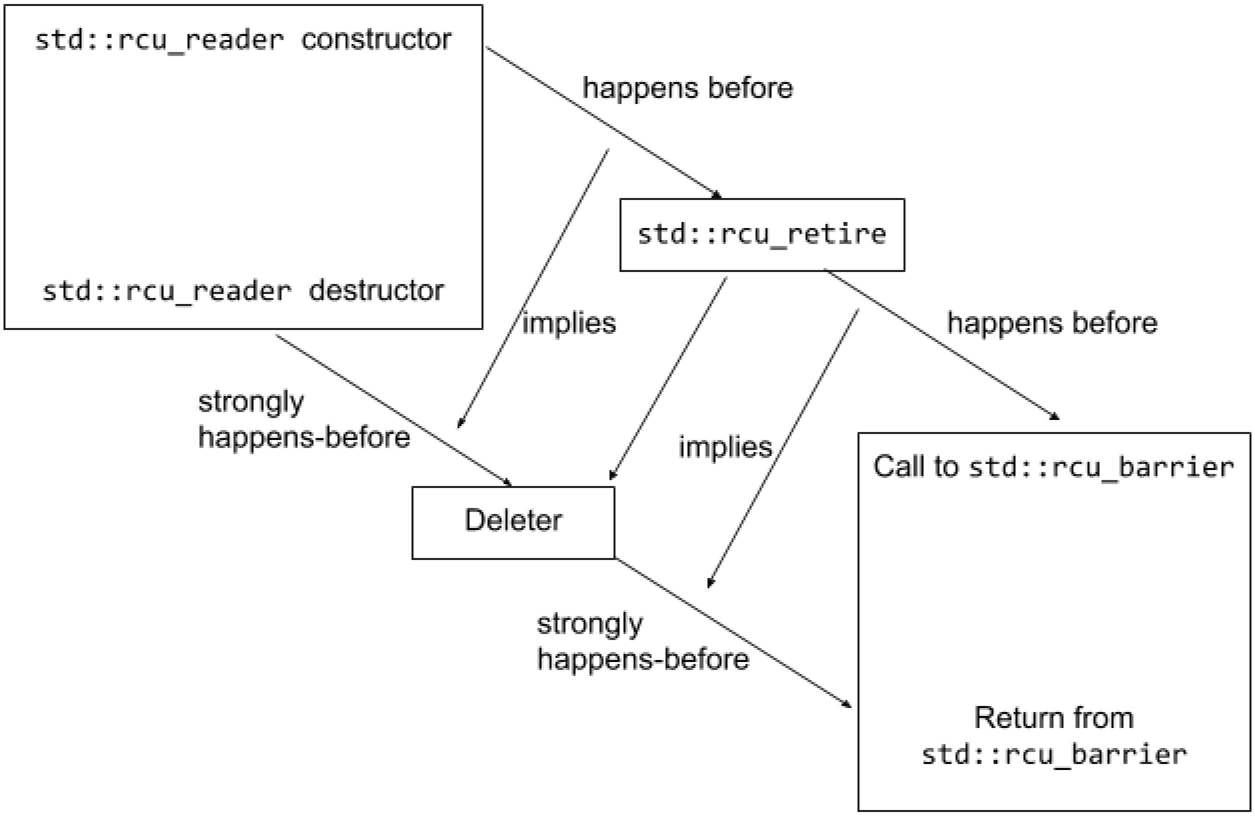
R (Reader)	W/D (Remover/Reclaimer)
R1: h.reset(ptr)	
// R1: P.store(ptr,rlx)	
<hb W2> // app	W1: src.store(null,rel); // app
	W2: ptr->retire()
R2: <use *ptr> // app	*/* retire() hb reclamation attempt on the same or another thread */*
	// D1a: <seq_cst fence>
	// D1b: P.load(acq) == ptr
	*// D1c: <don't reclaim *ptr>*
R3: h.reset()	
// R3: P.store(null,rel)	*// D2a: <seq_cst fence>*
	// D2b: P.load(acq) != ptr
	*// D2c: <reclaim *ptr>*

Pattern 3: Cleanup

R (Reader)	W (Remover)	D (Reclaimer)
R1: h.reset(ptr)		
// R1: P.store(ptr,rlx)		
<hb W2> // app	W1: src.store(null,rel); // app	
	W2: ptr->retire()	
R2: <use *ptr> // app	**<hb D1>**	
		D1: hazptr_cleanup()
R3: h.reset()		*// D1a: <seq_cst fence>*
// R3: P.store(null,rel)		*// D1b: P.load(acq) != ptr*
		*// D1c: <reclaim *ptr>*

Discussion in JAX: RCU memory-ordering diagrams

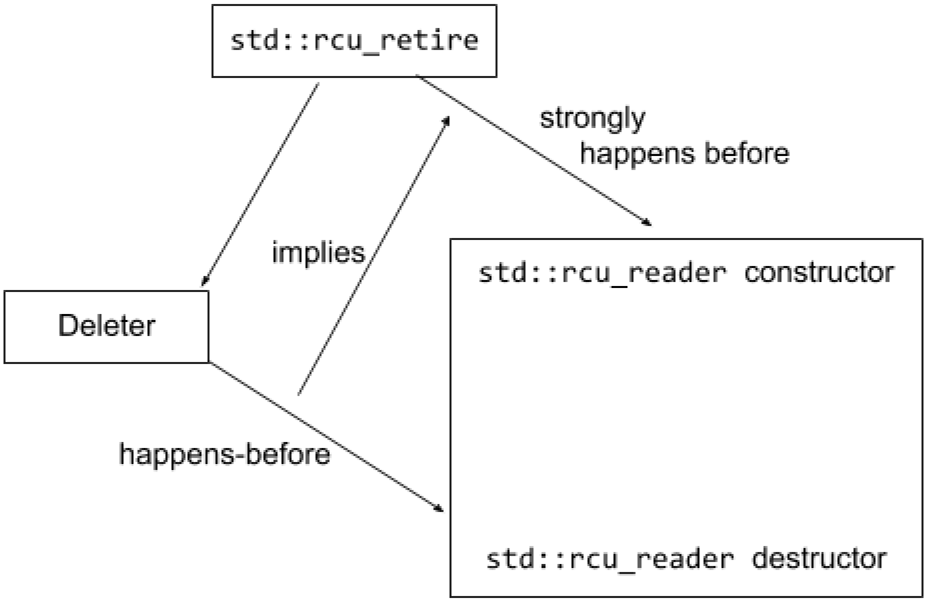
RCU provides strong ordering guarantees between `std::rcu_reader`, `std::rcu_retire`, and `std::rcu_barrier`. One set of guarantees applies to earlier instances of `std::rcu_reader` and later instances of `std::rcu_retire` along with `std::rcu_barrier`, as shown below:



In other words, if an `std::rcu_reader` constructor *happens before* an `std::rcu_retire`, then the RCU implementation guarantees that the `std::rcu_reader` destructor *strongly happens before* the corresponding deleter. Separately, if an `std::rcu_retire` *happens before* the call to an `std::rcu_barrier`, then the RCU implementation guarantees that the deleter *strongly happens before* the return from that `std::rcu_barrier`. Because *strongly happens before* is transitive, if the `std::rcu_barrier` is sequenced before some memory reference A, the entirety of the code protected by the `std::rcu_reader` *strongly happens before* A.

In both cases, the precondition is *happens before* and the postcondition is *strongly happens before*.

A similar guarantee applies to earlier instances of `std::rcu_retire` and later instances of `Std::rcu_reader`, as shown on the following diagram:



In other words, if a deleter *happens before* an `std::rcu_reader` destructor, then the RCU implementation guarantees that the `std::rcu_retire` *strongly happens before* the `std::rcu_reader` constructor.

In both diagrams, the `std::rcu_retire` can be replaced by the `std::retire` member function. Alternatively, and again in both diagrams, the `std::rcu_retire` may be replaced by a call to `rcu_synchronize` and the deleter may be replaced with the corresponding return from `rcu_synchronize`. In other words, these other two ways of inducing RCU grace periods have memory-ordering semantics that are identical to those of `std::rcu_retire`.

Notes from discussion in JAX

Hazard pointers and RCU.

Olivier: Need:

- Wording device
- Actual wording

Hans Boehm: Usage rules.

Frank: Usage rules.

JF: Not trying to change design. Just trying to make sure that the wording will fit into the design of Library.

Maged: try_protect, retire, cleanup.

Frank: Ordering? Maged: Ordering implied by the rules, for example, the "happens before" in the last normative paragraph of the retire member function.

Hans: Want better wording for Hazard Pointer retire member function.

Andrew:

The implementations divides the lifetime of each hazard pointer into a series of epochs separated by updates that change the hazard pointer's value. A non-nullptr hazard pointer protects the corresponding object from being reclaimed, and that protection persists until the end of that epoch. The implementation must guarantee that end of that epoch strongly happens before evaluation of the retire expression.

Note: A failed try_protect may create a spurious epoch. end note.

Frank: Invalid pointers!

Laundring? Geoff: No need, no dereferencing.

Maged: Cleanup. Andrew: Try epoch wording.

User must arrange for all retires and epoch ends happen before the cleanup.

Frank: The retire registers the expression, and that regist... Frank to email the issue, which he did:

> The [hazptr.base] describes the retire() call. That call takes a parameter “reclaim” which then is “registered for evaluation.” This basically means that the value is copied or moved into some storage internal to the facility. The call expression is later invoked on that instance. The effects of invoking the user provided copy/move constructor on the type D need to become visible to the invocation of the call expression on that object.
>
> Probably the implementation needs to try hard to fail this requirement, but the spec should say it nonetheless, I think.

Michael Young: cleanup does minimum cleanup.

RCU

JF: Horror-show litmus test? Olivier: RCU Litmus tests paper.

Geoff: Put front matter after synopsis.

Hans: Move ~rcu_reader precondition to be a non-normative note.

Andrew: Remove ~rcu_reader effects.

Geoff: No, strike entire ~rcu_reader section.

Ben: Normative weak CAS text.

Andrew: To define synchronize_rcu in terms of rcu_retire. Hans has input.

Andrew: Make retire member function as-if rcu_retire.

Frank: Concerned about the lifetime of the "d" argument to rcu_retire. Must not throw. Geoff: Overspecified, refer to unique_ptr. Which Frank did:

> [unique.ptr.single]
>
> 1) The default type for the template parameter D is default_delete. A client-supplied template argument D shall be a function object type (23.14), lvalue reference to function, or lvalue reference to function object type for which, given a value d of type D and a value ptr of type unique_ptr<T, D>::pointer, the expression d(ptr) is valid and has the effect of disposing of the pointer as appropriate for that deleter.

> [unique.ptr.single.ctor]
>
> 12) Requires: For the first constructor, if D is not a reference type, D shall satisfy the requirements of CopyConstructible and such construction shall not exit via an exception. For the second constructor, if D is not a reference type, D shall satisfy the requirements of MoveConstructible and such construction shall not exit via an exception.
>
> 18) Requires: If D is not a reference type, D shall satisfy the requirements of MoveConstructible (Table 23). Construction of the deleter from an rvalue of type D shall not throw an exception
>
> [unique.ptr.single.dtor]
>
> 1) Requires: The expression get_deleter()(get()) shall be well-formed, shall have well-defined behavior, and shall not throw exceptions. [Note: The use of default_delete requires T to be a complete type. — end note]

Olivier: Why many uses of strongly happens before? A: Need transitivity. Olivier: Synchronizes with? Geoff: "Synchronizes with" is primitive edge. Olivier: First time we order on both sides of a box (deleter) whose contents we do not control.

Nathan Myers: Allocators has lots of diagrams. Geoff: Wording reflect diagrams, so not needed, but maybe OK as non-normative note.

General acclamation of diagrams.