Document number: P1121R0
Date: 2018-10-05 (pre-San Diego)
Project: Programming Language C++, WG21, LEWG, LWG
Authors: Maged M. Michael, Michael Wong, Paul McKenney, Geoffrey Romer, Andrew Hunter, Arthur O'Dwyer, David S. Hollman, JF Bastien, Hans Boehm, David Goldblatt, Frank Birbacher, Mathias Stearn
Email: maged.michael@gmail.com, michael@codeplay.com, paulmck@linux.ibm.com, gromer@google.com, andrewhhunter@gmail.com, arthur.j.odwyer@gmail.com, dshollm@sandia.gov, jfbastien@apple.com, hboehm@google.com, davidtgoldblatt@gmail.com, frank.birbacher@gmail.com, redbeard0531+isocpp@gmail.com
Reply to: maged.michael@gmail.com, michael@codeplay.com, paulmck@linux.vnet.ibm.com

Hazard Pointers: Proposed Interface and Wording for Concurrency TS 2

1 Introduction

This paper contains proposed interface and wording for hazard pointers [1], a technique for safe deferred reclamation. This wording is based on N4700 draft [2]. An implementation is in the Folly open source library [3]

This paper needs review by LEWG and LWG with the intention of this being initial content for Concurrency TS 2.

2 History/Changes from Previous Release

Until the June 2018 Rapperswil meeting the interface and wording for hazard pointers were presented together with those for RCU (Read-Copy-Update) [4] in P0566 [5] with associated Bugzilla Bug #382. For the history of P0566 see the last revision P0566R5 (pre-Rapperswil). Earlier interface proposals are in P0233 [6]. This paper is a revision of the hazard pointer related parts of P0566. The RCU related parts are now in P1122, and the chapter headings from P0566 are now in P0940.

2018-10 Changes in [P1121R0] (pre-San Diego) from [P0566R5] (pre-Rapperswil).

- Edited the wording of hazard_pointer_obj_base retire(): Added a clarifying note. Removed instances of the word "then". Added the word "reclaim" to clarify the meaning of "expression".
- Changed "hazptr" to "hazard_pointer" in class and function names.
- Changed the class name "hazptr_holder" to "hazard_pointer".
- Changed the member function name "reset_protected" to "reset_protection".
- Changed the free function name "hazptr_cleanup" to "hazard_pointer_clean_up".

2018-06 LEWG Review in Rapperswil

- See details in the next section.

2018-06 SG1 Review in Rapperswil

- SG1 voted to forward the proposal to LEWG, provided that the following changes to the wording of hazaptr_obj_base retire are made: Add a clarifying note. Remove instances of the word "then". Add the word "reclaim" to clarify the meaning of "expression".

3 Background for Requested Review by LEWG in San Diego

We request a review by LEWG in San Diego to follow up on the LEWG review in Rapperswil on 2018-06-06. In that review, LEWG voted to forward the proposal to LWG and took ten other votes to provide guidance. Shortly after that review, we realized that some member functions of hazard_pointer will no longer be noexcept. Our position is that disadvantages of that outcome outweigh the advantages of eliminating the empty state as discussed below in more detail.

We request that LEWG revisit two of the votes (mentioned below) and forward the proposal in this paper to LWG.

Empty State vs Lazy Construction

Rationale for empty state: In some use cases of hazard pointers it is wasteful to fully construct hazard_pointer objects that will never be fully used and are intended to be moved into. This motivated an empty state for hazptr_holder in P0566R5 (as hazard_pointer was called in that paper). The purpose of an empty hazptr_holder is that it is faster to construct than a fully functional one (approximately 1 ns. vs. 4 ns.).

Elimination of empty state: Subsequent discussion (in May 2018) on the LEWG reflector led to changing the proposal to eliminate the empty state and instead introduce lazy (and eager) construction, where lazy construction provides a way to construct hazard_pointer objects (as hazptr_holder was renamed based on discussions on the reflector at the time) that are not fully functional and faster to construct. This was the proposal that was presented to LEWG on 2018-06-06 and approved (see votes #7 and #8 below).

Shortly after that LEWG review, we realized that the possibility of lazy construction (as opposed to having an explicit empty state) will prevent the hazard_pointer member functions protect, try_protect, and reset_protection from being noexcept. Our view is that this disadvantage outweighs the benefits of eliminating the empty state.

Therefore, we propose restoring the empty state and revisiting votes #7 and #8, listed below.

LEWG Votes in Rapperswil on 2018-06-06

We request revisiting votes #7 and #8. We propose eliminating lazy construction because it prevents protect, try_protect, reset_protection from being noexcept. Instead of lazy construction, we propose restoring the empty state.

On a different issue, allocation of hazard pointers, note that votes #2, #3, and #4 were inconclusive, we prefer using a PMR allocator (vs not allowing custom allocators or using conventional template allocators), because this is a proposal for TS and supporting custom allocators makes it more likely to gain usage feedback regarding custom allocation of hazard pointers. Traditional allocator templates are unsuitable as they would introduce a viral template parameter to the whole hazard pointer API.

From Bugzilla [Bug 382] P0566 Comment #10 on 2018-06-07.

(1) Replace “hazptr” with “hazard_pointer”
SF F N A SA
6 7 2 3 0

(2) Keep the PMR allocator.
SF F N A SA
1 5 6 1 2

(3) Use traditional allocators.
SF F N A SA
1 3 6 2 3

(4) Use no allocators.
SF F N A SA
0 4 5 3 1

(5) Keep the default domain function, returning a reference.
Unanimous consent

(6) Rename “cleanup” to “clean_up”
Unanimous consent

(7) Eliminate the empty state of hazard pointers (adding eager/lazy tags to determine whether we eagerly refresh from the domain).
SF F N A SA
2 8 2 1 0

(8) Provide an (optional) construction hint for eager/lazy.
Unanimous consent

(9) Eliminate the distinction between hazard pointer holders and hazard pointers.
Unanimous consent

(10) Rename “reset_protected” to “reset_protection”
Unanimous consent

(11) Forward the hazard pointer parts to LWG for Concurrency TS2.
Unanimous consent

4 Proposed wording

? Hazard Pointers [hazard_pointer]

1. The lifetime of each hazard pointer is split into a series of nonoverlapping epochs, with each epoch associated with a particular pointer to a hazard_pointer_obj_base instance (or `NULL`). Consecutive epochs associated with the same instance are treated as distinct epochs. The initial epoch of each hazard pointer is associated with NULL.
2. Certain ways of starting a hazard pointer epoch associated with a pointer to an object will defer reclamation of that object until the end of the epoch.
3. The hazard pointer library allows for multiple hazard pointer domains, where the reclamation of objects in one domain is not affected by the hazard pointers in different domains. It is possible for the same thread to participate in multiple domains concurrently.
4. Operations on hazard pointers are exposed through the `hazard_pointer` class. Each instance of `hazard_pointer` owns and operates on at most one hazard pointer. Each hazard_pointer call to `protect`, `try_protect`, or `reset_protection` begins a new epoch and ends the previous one for the owned hazard pointer. Non-empty construction begins an epoch associated with NULL, and destruction of a non-empty hazard_pointer ends its epoch.
5. A hazard pointer domain contains a set of hazard pointers. A domain is responsible for reclaiming objects retired to it (by calling `hazard_pointer_obj_base retire`), when such objects are not protected by hazard pointers that belong to this domain (including when this domain is destroyed).
6. A `hazard_pointer_obj_base` O is *definitely reclaimable* in domain D at program point P if:
    a. there is a call to O.`retire`(reclaim, D), and it happens before P, and
    b. For each epoch E in D associated with O, the end of E happens before P.
7. A `hazard_pointer_obj_base` O is *possibly reclaimable* in domain D at program point P if:
    a. There is a call to O.`retire`(reclaim, D) and P does not happen before the call, and
    b. For each epoch E in D associated with O, P does not happen before the end of E.

[Note— The following example shows how hazard pointers allow updates to be carried out in the presence of concurrent readers. Each `hazard_pointer` instance in `print_name` is used through the call to `protect` to start an epoch associated with `ptr` to protect the object *ptr from being reclaimed by `ptr->retire` until the end of the epoch.

```
struct Name : public hazard_pointer_obj_base<Name> { /* details */ };
atomic<Name*> name;
```

```
// called often and in parallel!
void print_name() {
  hazard_pointer h = make_hazard_pointer();
  Name* ptr = h.protect(name);
  /* ... safe to access *ptr ... */
}

// called rarely
void update_name(Name* new_name) {
  Name* ptr = name.exchange(new_name);
  ptr->retire();
}
```

—end note]

Header <hazard_pointer> synopsis

```
namespace std {
namespace experimental {

// ?.?, Class hazard_pointer_domain:
class hazard_pointer_domain;

// ?.?, Default hazard_pointer_domain:
hazard_pointer_domain& hazard_pointer_default_domain() noexcept;

// ?.?, Clean up
void hazard_pointer_clean_up(
    hazard_pointer_domain& domain = hazard_pointer_default_domaind());

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

// ?.?, Class hazard_pointer
class hazard_pointe

// ?.?, Construct non-empty hazard_pointer
hazard_pointer make_hazard_pointer(
    hazard_pointer_domain& domain = hazard_pointer_default_domain());

// ?.?, Hazard pointer swap
```

```
void swap(hazard_pointer&, hazard_pointer&) noexcept;

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

?.? Class hazard_pointer_domain [hazard_pointer.domain]

1. The number of unreclaimed possibly reclaimable objects retired to a domain is bounded. The bound is implementation-defined. The bound is independent of other domains and may be a function of the number of hazard pointers in the domain, the number of threads that retire objects to the domain, and the number of threads that use hazard pointers that belong to the domain.

```
class hazard_pointer_domain {
 public:

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

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

  // ?.?.? destructor:
  ~hazard_pointer_domain();
 private:
  std::pmr::polymorphic_allocator<byte> alloc_;  // exposition only
};
```

?.?.? `hazard_pointer_domain` constructors [hazard_pointer.domain.constructor]

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

1. Effects: Sets `alloc_` to `poly_alloc`.
2. Throws: Nothing.
3. Remarks: All allocation and deallocation of hazard pointers in this domain will use `alloc_`.

?.?.? `hazard_pointer_domain` destructor [hazard_pointer.domain.destructor]

```
~hazard_pointer_domain();
```

1. Requires: The destruction of all hazard pointers in this domain (including hazard pointers with epochs associated with NULL) 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 plus the number of hazard pointers contained in this domain.

?.? Default `hazard_pointer_domain` [hazard_pointer.default_domain]

```
hazard_pointer_domain& hazard_pointer_default_domain() noexcept;
```

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

?.? Cleanup [hazard_pointer.cleanup]

```
void hazard_pointer_clean_up(hazard_pointer_domain& domain =
hazard_pointer_default_domain());
```

1. Effects: For a set of hazard_pointer_obj_base objects O in domain for which O.retire(reclaim, domain) has been called, ensures that O has been reclaimed. The set contains all definitely reclaimable objects at the point of cleanup, and may contain some possibly reclaimable objects.
2. Synchronization: The end of evaluation of each reclaim function of objects in the set synchronizes with the return from this call.

[*Note:* To avoid deadlock, this function must not be called while holding resources that may be required by such expressions. — *end note*]

?.? Class template `hazard_pointer_obj_base` [hazard_pointer.base]

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

```
template <typename T, typename D = default_delete<T>>
class hazard_pointer_obj_base {
 public:
  // retire
  void retire(
      D reclaim = {},
      hazard_pointer_domain& domain = hazard_pointer_default_domain());
  void retire(hazard_pointer_domain& domain);
 protected:
  hazard_pointer_obj_base() = default;
```

```
};
```

1. If this template is instantiated with a `T` argument that is not publicly derived from `hazard_pointer_obj_base<T,D>` for some `D`, the program is ill-formed.
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. A client-supplied template argument `D` shall be a function object type ([function.object]) 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.
4. A program may not add specializations of this template.

```
void retire(
    D reclaim = {},
    hazard_pointer_domain& domain = hazard_pointer_default_domain());
```

1. Requires: `D` shall satisfy the requirements of MoveConstructible and such construction shall not exit via an exception. The *reclaim expression* `reclaim(static_cast<T*>(this))` shall be well-formed, shall have well-defined behavior, and shall not throw exceptions.
2. Effects: Registers the expression `reclaim(static_cast<T*>(this))` to be evaluated asynchronously. For every hazard pointer in the domain, for epoch E associated with the value `static_cast<T*>(this)`:
   a. If the beginning of E happens before this call, the end of E strongly happens before the evaluation of the reclaim expression.
   b. If E began as part of an evaluation of try_protect(ptr, src) returning true (for some src and ptr == `static_cast<T*>`(this)), let its associated atomic load operation be labelled A. If there exists an atomic modification B on src such that A observes a modification that is modification-ordered before B, and B happens before this call, the end of E strongly happens before the evaluation of the reclaim expression. [Note: in typical use, a store to src sequenced before this call will be such atomic operation B.]
   [Note: Both of these preconditions convey the informal notion that the hazard pointer epoch begins before the retire() call occurs, as implied either by the happens-before relation or the coherence order of some source.]

   The reclaim expression will be evaluated only once, and it will be evaluated by the evaluation of a `retire()` or `hazard_pointer_clean_up()` operation on `domain`.

   This function may also evaluate any number of reclaim expressions for hazard_pointer_obj_base objects possibly reclaimable in domain.

   [*Note:* To avoid deadlock, this function must not be called while holding resources required by such reclaim expressions. — *end note*]

```
  void retire(hazard_pointer_domain& domain);
```

1. Effects: Equivalent to
   `retire({}, domain);`

?.? Class hazard_pointer [hazard_pointer.holder]

A `hazard_pointer` object acts as a local handle on a hidden hazard pointer, which can be used to protect at most a single `hazard_pointer_obj_base` object at a time. Every object of type `hazard_pointer` is either empty or *owns* exactly one hazard pointer, and has no protection against data races other than what is specified for the library generally ([res.on.data.races]). Every hazard pointer is owned by exactly one `hazard_pointer` object.

```
class hazard_pointer {
 public:
  hazard_pointer() noexcept;

  hazard_pointer(hazard_pointer&&) noexcept;
  hazard_pointer& operator=(hazard_pointer&&) noexcept;

  hazard_pointer(const hazard_pointer&) = delete;
  hazard_pointer& operator=(const hazard_pointer&) = delete;

  ~hazard_pointer();

  bool empty() const noexcept;

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

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

  template <typename T>
  void reset_protection(const T* ptr) noexcept;
  void reset_protection(nullptr_t = nullptr) noexcept;

  void swap(hazard_pointer&) noexcept;
};
```

?.?.? `hazard_pointer` constructors [hazard_pointer.holder.constructors]

`hazard_pointer() noexcept;`

1. Effects: Constructs an empty hazard_pointer.

```
hazard_pointer(hazard_pointer&& other) noexcept;
```

1. Effects: If `other` is empty, constructs an empty `hazard_pointer`. Otherwise, constructs a hazard_pointer that owns the hazard pointer originally owned by `other`. `other` becomes empty.

?.?.? `hazard_pointer` destructor [hazard_pointer.holder.destructor]

```
~hazard_pointer();
```

1. Effects: If `*this` is not empty, destroys the owned hazard pointer which ends its current epoch.

?.?.? hazard_pointer assignment [hazard_pointer.holder.assignment]

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

1. Effects: If `this == &other`, no effect. Otherwise, if `other` is not empty, `*this` takes ownership of the hazard pointer originally owned by `other`, and `other` becomes empty. If `*this` was not empty before the call, destroys the owned hazard pointer which ends its current epoch
2. Returns: `*this`.

?.?.? hazard_pointer empty [hazard_pointer.holder.empty]

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

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

?.?.? hazard_pointer protect [hazard_pointer.holder.protect]

```
template <typename T>
  T* protect(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;
```

?.?.? hazard_pointer try_protect [hazard_pointer.holder.try_protect]

```
template <typename T>
```

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

1. Requires: `*this` is not empty.
2. Effects:
    a. Ends the owned hazard pointer's current epoch, and starts a new one.
    b. Performs an atomic acquire load on `src`. If `src == ptr`, the hazard pointer's new epoch is associated with the value `ptr`, and `try_protect()` returns true. Otherwise, the new epoch is associated with `NULL`, and `try_protect()` returns false.
    c. Sets ptr to the value read from src.
3. Returns: The result of the comparison. [*Note:* It is possible for `try_protect` to return `true` when `ptr` is a null pointer. — *end note*]
4. Complexity: Constant.

?.?.? hazard_pointer reset_protection [hazard_pointer.holder.reset]

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

1. Requires: `*this` is not empty.
2. Effects: Ends the owned hazard pointer's current epoch, and begins a new one associated with `ptr`.

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

1. Requires: `*this` is not empty.
2. Effects: Ends the owned hazard pointer's current epoch, and begins a new one associated with NULL..

?.?.? hazard_pointer swap[hazard_pointer.holder.swap]

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

1. Effects: Swaps the hazard pointer ownership and the associated domain of this object with those of `other`. [*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.

?.? make_hazard_pointer [hazard_pointer.make]

```
hazard_pointer make_hazard_pointer(
    hazard_pointer_domain& domain = hazard_pointer_default_domain());
```

1. Effects: Constructs a hazard pointer from `domain`, and returns a `hazard_pointer` that owns it.
2. Throws: Any exception thrown by `domain.alloc_.allocate()`.

?.? hazard_pointer specialized algorithms [hazard_pointer.holder.special]

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

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

5. Acknowledgements

The authors thank Keith Bostic, Olivier Giroux, Pablo Halpern, Davis Herring, Lee Howes, Bronek Kozicki, Nathan Myers, Xiao Shi, Viktor Vafeiadis, Tony Van Eerd, Dave Watson, Anthony Williams and other members of SG1 and LEWG for useful discussions and suggestions that helped improve this paper and its earlier versions.

6. References

[1] Maged M Michael. "Hazard pointers: Safe memory reclamation for lock-free objects." *Parallel and Distributed Systems, IEEE Transactions on* 15.6 (2004): 491-504.

[2] N4700 http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2017/n4700.pdf

[3] Hazard Pointer Implementation: https://github.com/facebook/folly/blob/master/folly/synchronization/Hazptr*

[4] P0461 Proposed RCU C++ API http://wg21.link/P0461

[5] P0566 Proposed Wording for Concurrent Data Structures: Hazard Pointer and ReadCopyUpdate (RCU). http://wg21.link/P0566

[6] P0233 Hazard Pointers: Safe Resource Reclamation for Optimistic Concurrency. http://wg21.link/P0233