Document number: P3135R0
Date: 2024-02-11
Project: Programming Language C++, SG1
Authors: Maged M. Michael, Michael Wong, Paul McKenney
Email: maged.michael@acm.org, michael@codeplay.com, paulmck@kernel.org

Hazard Pointer Extensions

Table of Contents

1 Introduction

Support for hazard pointers (P2530R3) was voted for inclusion in C++26 (Varna 2023 plenary). In this paper, we describe several hazard pointer extensions that can be beneficial to users. We recommend and prioritize some of these extensions based on production experience. We also discuss potential extensions that we do not yet recommend for standardization. We seek SG1's feedback on the prioritization of these extensions.

Background: Hazard Pointers for C++26

Hazard pointer interface from P2530R3:

```
template <class T, class D = default_delete<T>>
class hazard_pointer_obj_base {
public:
  void retire(D d = D()) noexcept;
protected:
  hazard_pointer_obj_base() = default;
  hazard_pointer_obj_base(const hazard_pointer_obj_base&) = default;
  hazard_pointer_obj_base(hazard_pointer_obj_base&&) = default;
  hazard_pointer_obj_base& operator=(const hazard_pointer_obj_base&) = default;
  hazard_pointer_obj_base& operator=(hazard_pointer_obj_base&&) = default;
  ~hazard_pointer_obj_base() = default;
private:
  D deleter ; // exposition only
};

class hazard_pointer {
public:
  hazard_pointer() noexcept;
  hazard_pointer(hazard_pointer&&) noexcept;
  hazard_pointer& operator=(hazard_pointer&&) noexcept;
  ~hazard_pointer();
  [[nodiscard]] bool empty() const noexcept;
  template <class T> T* protect(const atomic<T*>& src) noexcept;
  template <class T> bool try_protect(T*& ptr, const atomic<T*>& src) noexcept;
  template <class T> void reset_protection(const T* ptr) noexcept;
  void reset_protection(nullptr_t = nullptr) noexcept;
  void swap(hazard_pointer&) noexcept;
};

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

Brief notes (See P2530R3 for details):

- A `hazard_pointer_obj` object (i.e. *protectable*) can be protected from reclamation using a hazard pointer.
- A `hazard_pointer` object is either *empty* or *nonempty*. It is nonempty if and only if it owns a hazard pointer. Only nonempty `hazard_pointer` objects can be used to protect protectable objects.
- The default constructor of `hazard_pointer` constructs an empty object, whereas the free function `make_hazard_pointer` constructs a nonempty object.

Potential Extensions

The following is an overview of the extensions discussed in this paper:

Batches of Hazard pointers:

- Objective: Efficiently construct and destroy multiple hazard pointers as a batch, rather than individually.
- Rationale: The latency of the construction and the destruction of multiple hazard pointers individually can be amortized if they are contructed and destroyed as a batch.
- Advantage: Improved latency.

Synchronous reclamation:

- Objective: Support for protecting objects with deleters dependent on resources that may independently become unavailable.
- Rationale: Imagine a situation where an object's deleter depends on external resources (e.g., files, network connections) that may become unavailable.
- Advantage: Extend hazard pointers to handle such scenarios, ensuring safe reclamation. Increase the usability of hazard pointers.

Integrated protection counting:

- Objective: Combine commutative and noncommutative counted protection with hazard pointer protection.
- Rationale: Hazard pointer protection can be combined with counting-based protection.
- Advantages: User convenience of using integrated counting mechanisms without the need to implement them.

Dedicated reclamation execution:

- Objective: Specify policies for reclamation execution, allowing customization of the reclamation process (e.g., inline or on separate execution resources).
- Rationale: The appropriate reclamation execution policies vary among applications.
- Advantage: Enable users to specify reclamation execution policy.

Custom domains:

- Customization of hazard pointer domain structures and policies and separation from the default domain and other custom domains.
- Objective: Support custom hazard pointer domains (separate from the default global domain) with custom structures and policies.
- Rationale: Facilitating tailored hazard pointer configurations to better align with specific application architectures and usage scenarios.
- Advantage: Fine-tune hazard pointer behavior for specific use cases.

2. Batches of Hazard Pointers

Supporting batches of `hazard_pointer` objects has the effect of lowering the latency of the construction and destruction of multiple hazard pointers compared to their construction and destruction individually.

Suggested Interface

The following is a suggested interface:

```
template <uint8_t N>
class hazard_pointer_batch {
  hazard_pointer_batch() noexcept;
  hazard_pointer_batch(hazard_pointer_batch&&) noexcept;
  hazard_pointer_batch& operator=(hazard_pointer_batch&&) noexcept;
  ~hazard_pointer_batch();
  [[nodiscard]] bool empty() const noexcept;
  hazard_pointer& operator[](uint8_t) noexcept;
  void swap(hazard_pointer_batch&) noexcept;
};

template <uint8_t N> hazard_pointer_batch<N> make_hazard_pointer_batch();
template <uint8_t N>
void swap(hazard_pointer_batch<N>&, hazard_pointer_batch<N>&) noexcept;
```

Note that:

- Individual elements of a **`hazard_pointer_batch`** object can be used to invoke the following **`hazard_pointer`** member functions: **`empty`**, **`protect`**, **`try_protect`**, and **`reset_protection`**.
- Moving **`hazard_pointer`** objects to or from individual elements of a **`hazard_pointer_batch`** is prohibited for ensuring consistency within the batch.
- That is, the elements of a **`hazard_pointer_batch`** can be either all empty or all nonempty, at the same time.
- The free function **`make_hazard_pointer_batch`** constructs a nonempty **`hazard_pointer_batch`** object (i.e., a batch object with each of its elements nonempty).

Usage Example

The following table shows two functionally-equivalent code snippets using the P2530R3 C++26 hazard pointer interface and using hazard pointer batches.

C++26	Extension

`hazard_pointer hp[3];`
`hp[0] = make_hazard_pointer();`
`hp[1] = make_hazard_pointer();`
`hp[2] = make_hazard_pointer();`
`// src is atomic<T*>`
`T* ptr = hp[0].protect(src);`
	`hazard_pointer_batch<3> hp;`
	`hp = make_hazard_pointer_batch<3>();`
	`// src is atomic<T*>`
	`T* ptr = hp[0].protect(src);`

Recommendation

We recommend this extension for C++29 due its usefulness and simplicity.

Hazard pointer batches have been part of the Folly open source library (under the name **`hazard_pointer_array`**) and in heavy use in production since 2017.

3. Synchronous Reclamation

The P2530R3 C++26 hazard pointer interface supports only asynchronous reclamation which does not guarantee the timing of the reclamation of protectable objects that are no longer protected. As a result, hazard pointer users must guarantee separately that the deleters of such objects do not depends on resources that may become subsequently unavailable.

Support for synchronous reclamation allows users to synchronously induce and wait for the reclamation of unprotected objects.

Global Cleanup

A straightforward albeit inefficient solution to this problem is global cleanup, which guarantees the completion of deleters of all unprotected retired objects. This involves synchronously checking all retired objects against all hazard pointers.

The main drawback of the global cleanup approach is its high overhead that makes it impractical to use.For example, it may be useful to include global cleanup in the destructor of a generic container library. However, the prohibitive overhead of global cleanup makes that impractical for many use cases of such a container library.

Another drawback of the global cleanup approach is that it adds overhead to the hazard pointer implementation even when users never use this feature (e.g., would require synchronization on thread local private buffers of retired objects that would otherwise unnecessary).

We do not recommend the standardization of the global cleanup approach due to these drawbacks and the lack of production experience of the necessity of such approach in contrast to the more efficient approach discussed in the following subsection.

Object Cohorts

An alternative approach is the use of object cohorts, which are sets of protectable objects. Object cohorts support synchronous reclamation by guaranteeing that all the deleters of the object cohort members are completed before the completion of the cohorts destructor.

The main advantage of the object cohort approach is its performance. While its guarantees are weaker and less flexible than global cleanup, its efficiency enables users to use it in performance sensitive cases where the cost of global cleanup may be impractical. It strikes a better balance between practicality and performance. Therefore, we recommend object cohorts for standardization.

Suggested Interface

The following is a suggested interface:

```
class hazard_pointer_cohort {
  hazard_pointer_cohort() noexcept;
  hazard_pointer_cohort(const hazard_pointer_cohort&) = delete;
  hazard_pointer_cohort(hazard_pointer_cohort&&) = delete;
  hazard_pointer_cohort& operator=(const hazard_pointer_cohort&) = delete;
  hazard_pointer_cohort& operator=(hazard_pointer_cohort&&) = delete;
  ~hazard_pointer_cohort();
};

template <class T, class D = default_delete<T>>
class hazard_pointer_obj_base {
public:
  void include_in_cohort(hazard_pointer_cohort&) noexcept;
};
```

Notes:

- An object may be a member of at most one cohort.
- An object's cohort membership must be specified before the object's retirement.
- An object that is not a member of a cohort is subject to asynchronous reclamation.

Usage Example

The following table shows two code snippets one using the P2530R3 C++26 hazard pointer interface and one using object cohorts, respectively. The latter supports synchronous reclamation.

<table>
<tr><th>C++26</th><th>Cohort-Based Synchronous Reclamation</th></tr>
<tr><td><pre>
template <class T>
class Container {
  class Obj :
      hazard_pointer_obj_base<Obj>
  { T data; /* etc */ };

  void insert(T);

  void erase(Args args) {
    /* ... */

    obj->retire();
  }
};

class A {
  // Deleter cannot depend on resources
  // with independent lifetime.
   ~A();
};

{
  Container<A> cont;
  cont.insert(a);
  cont.erase(a);
}
// Obj containing a may be not deleted
// yet.
</pre></td><td><pre>
template <class T>
class Container {
  class Obj :
      hazard_pointer_obj_base<Obj>
  { T data; /* etc */ };

  hazard_pointer_cohort cohort_;

  void insert(T);

  void erase(Args args) {
    /* ... */
    obj->include_in_cohort(cohort_);
    obj->retire();
  }
};

class B {
  // Deleter may depend on resources
  // with independent lifetime.
  ~B() { use_resource_XYZ(); }
};

make_resource_XYZ();
{
  Container<B> cont;
  cont.insert(b);
  cont.erase(b);
}
// Obj containing b was deleted.

destroy_resource_XYZ();
</pre></td></tr>
</table>

Recommendation

We recommend supporting object cohorts for C++29 due the importance of synchronous reclamation for general purpose usability. For example, a concurrent hash map that uses hazard pointers is more generally usable if it allows arbitrary key and value types rather than only types without dependence on resources with independent lifetimes.

Object cohorts have been part of the Folly open source library (under the name **`hazptr_obj_cohort`**) and in heavy use in production since 2018. (See CppCon 2021 *Hazard Pointer Synchronous reclamation beyond Concurrency TS2* for details about the evolution of support for synchronous reclamation in Folly).

4. Integrated Protection Counting

Users can apply protection via counting on top of C++26 hazard pointers. Protection counting falls into two categories depending on whether the protection counting is commutative or noncommutative with hazard pointer protection.

A potential extension is to add integrated support for one or both of these protection counting techniques.

Commutative Protection Counting

This category represents cases where the release of counted protection is commutative with the retirement of the protectable object. Examples of such cases include linked counting in linked structures with immutable links (e.g., LIFO and FIFO linked lists)..

The commutativity of the release of protection counting with object retirement (and subsequently the determination of lack of hazard pointer protection) allows the reclamation of large numbers of protectable objects in deep linked structures in one hazard pointer reclamation pass.

Noncommutative Protection Counting

This category represents cases where the release of counted protection is noncommutative with the retirement of the protectable object. Examples of such cases include linked counting in linked structures with mutable links (e.g., linked lists that support insertion and removal of internal nodes).

For this category, the reclamation of objects in a linked structure require as many hazard pointer reclamation passes as the depth of the structure.

Examples

<table>
<tr><th>C++26 Noncommutative Counting</th><th>C++26 Commutative Counting</th></tr>
<tr><td><pre>

struct Node : hazard_pointer_obj_base<Node>

{
  Node* next_{nullptr};
  atomic<int> count_{0};
  Node();
  ~Node() {
    if (next_) next_->release();
  }
</pre></td><td><pre>
struct NodeDeleter {
  template <class T>
  void operator()(T* node)
  { node->release(); }
};

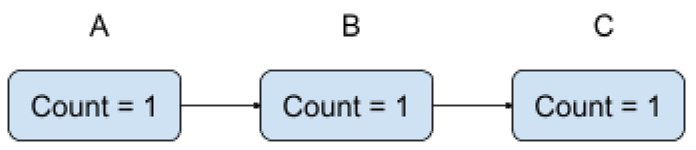
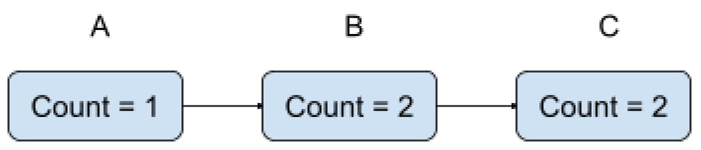
struct Node : hazard_pointer_obj_base<
                         Node, NodeDeleter>
{
  Node* next_{nullptr};
  atomic<int> count_{1};
  Node();
  ~Node() {
    if (next_) next_->release();
  }
</pre></td></tr>
</table>

```
  void set_next(Node*);
  void acquire() { ++count_;  }
  void release() {
    if (--count_ == 0) this->retire();
  }
};

// Linked list of 1000 Nodes takes at
// least 1000 hazard pointer reclamation
// passes.
```

```
  void set_next(Node*);
  void acquire() { ++count_; }
  void release() {
    if (--count_ == 0) this->~Node();
  }
};

// Linked list of 1000 Nodes may be
// reclaimed in one hazard pointer
// reclamation pass.
```

Noncommutative Counting	Commutative Counting
	
User calls A**.release()** A's count goes from 1 to 0 A is automatically retired **Hazard pointer reclamation pass <<<<<<< #1** A is reclaimed A's dtor calls B**.release()** B's count goes from 1 to 0 B is retired **Hazard pointer reclamation pass <<<<<<< #2** B is reclaimed B's dtor calls C**.release()** C's count goes from 1 to 0 C is retired **Hazard pointer reclamation pass <<<<<<< #3** C is reclaimed	User retires A User retires B User retires C **Hazard pointer reclamation pass <<<<<<<<<** B is not protected by hazard pointers B**.release()** is called B's count goes from 2 to 1 C is not protected by hazard pointers C**.release()** is called C's count goes from 2 to 1 A is not protected by hazard pointers A**.release()** is called A's count goes from 1 to 0 A is reclaimed A's dtor calls B**.release()** B's count goes from 1 to 0 B is reclaimed B's dtor calls C**.release()** C's count goes from 1 to 0 C is reclaimed

Recommendation

As protection counting can be implemented efficiently using the C++26 hazard pointer interface, we do not see a compelling reason to recommend standardizing the integration of protection counting with hazard pointers.

5. Dedicated Reclamation Execution

By default the reclamation of retired hazard pointer protectable objects happens asynchronously. A library implementation may perform reclamation inline upon the retirement of an object. A hazard pointer reclamation pass to identify unprotected retired objects is an expensive heavily amortized operation. Therefore, it may be undesirable for a worker thread that retires objects to experience long delays (determining the protection status of tens of thousands of objects). Alternatively, a library implementation may use a dedicated background thread pool for asynchronous reclamation. However, the possibility of spawning background threads is unsuitable for some application.

A possible extension to the standard is to specify a default policy for the execution of hazard pointer reclamation and providing an interface for opting in or opting out of the use of a dedicated execution resource for reclamation.

The hazard pointer implementation in the Folly open source library uses a dedicated reclamation thread pool by default and provides a mechanism for opting out.

User-Defined Alternative

Instead of retiring an object directly, a worker thread can submit the work to retire the object to the dedicated reclamation thread pool.

Possible inline reclamation execution	Using a dedicated reclamation executor

void worker() {   /* ... */
  obj->retire();
 	
  // Possible inline reclamation
  // execution.
}
	void worker() { /* ... */
	  dedicate_pool.submit([obj]() {
	    obj->retire();
	  });
	
	  // No inline reclamation execution.
	}

Recommendation

We do not have a compelling reason to recommend extending the standard to specify the policy for reclamation execution, due to the existence of an adequate user-defined alternative.

6. Custom Domains

The hazard pointer implementation in the Folly open source library has supported custom domains since 2017. However, no need arose for custom domains, as the functionality of the default (global) domain expanded (with object cohorts and integrated counting).

Recommendation: We do not have sufficient information to recommend extending the standard to support for custom domains.

7. Proposed Extensions

We recommend the following for inclusion in C++29:

1. Hazard pointer batches
2. Cohort-based synchronous reclamation

As mentioned above, both extensions have been part of the Folly open source library and in heavy use in production since 2017 and 2018, respectively.

We seek SG1's feedback on the substance and the prioritization of the proposed extensions.