Document number: P3427R0
Date: 2024-10-11
Project: Programming Language C++, SG1
Authors: Maged M. Michael, Michael Wong, Paul McKenney
Email: maged.michael@gmail.com, fraggamuffin@gmail.com, paulmck@kernel.org

Hazard Pointer Synchronous Reclamation

Table of Contents

Introduction

The Varna 2023 plenary voted in favor of including hazard pointers in the C++26 standard library ([2023-06 LWG Motion 7] P2530R3 Hazard Pointers for C++26).

P3135R1 was presented to SG1 in Tokyo 2024, reviewing potential extensions of the P2530R3 C++26 interface, and proposing two of those for inclusion in the standard library. The proposal for extending the P2530R3 C++26 interface to support synchronous reclamation was voted on by the Concurrency Study Group (SG1) in Tokyo 2024 as follows:

```
we want to continue work on hazard pointer cohorts (synchronous reclamation) for C++26,
with association to the cohort registered at the time of retire()
SF F N A SA
 6 6 0 0 0
Unanimous consent
```

This paper is a follow up on P3135R1, focusing on extending the P2530R3 C++26 hazard pointer interface to support synchronous reclamation, revised to take into account the feedback from SG1.

Background: P2530R3 C++26 Hazard Pointers

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;
```

Motivation

This paper proposes supporting object cohorts (in C++26 or 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.

Implementation and Use Experience

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).

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.

Possible 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 retire_to_cohort(hazard_pointer_cohort&, D d = D()) noexcept;
};

void hazard_pointer_asynchronous_reclamation() noexcept;
```

Notes:

- An object may be a member of at most one cohort.
- An object's cohort membership, if any, is specified at the object's retirement.
- An object that is not a member of a cohort is subject to asynchronous reclamation.
- An object retired to a cohort may be reclaimed asynchronously before the destruction of the cohort.

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.

P2530R3 C++26 (Asynchronous Reclamation Only)

```
template <class T> class Container {
  class Obj : hazard_pointer_obj_base<Obj>
  { T data; /* etc */ };

  void insert(T data) {
    Obj* obj = new Obj(data);
    /* Insert obj in container */
  }
  void erase(Args args) {
    Obj* obj = find(args);
    /* Remove obj from container */
    executor_.add([] {
      obj->retire();
    });

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


{
  Container<A> container;
  container.insert(a);
  container.erase(a);
}
// Obj containing 'a' may be not deleted
// yet.
```

Cohort-Based Synchronous Reclamation

```
template <class T> class Container {
  class Obj : hazard_pointer_obj_base<Obj>
  { T data; /* etc */ };
  hazard_pointer_cohort cohort_;
  void insert(T data) {
    Obj* obj = new Obj(data);
    /* Insert obj in container */
  }
  void erase(Args args) {
    Obj* obj = find(args);
    /* Remove obj from container */
    obj->retire_to_cohort(cohort_);
    exucutor_.add([] {
hazard_pointer_asynchronous_reclamation();
    });
  }
};
class B {
  // Deleter may depend on resources
  // with independent lifetime.
  ~B() { use_resource_XYZ(); }
};

make_resource_XYZ();
{
  Container<B> container;
  container.insert(b);
  container.erase(b);
}
// Obj containing 'b' was deleted.
destroy_resource_XYZ();
```

In the above example:

- In the code snippet on the left side, the removed object is retired asynchronously by submitting the retirement code to a dedicated thread pool (details not shown) to be executed asynchronously. Doing so avoids burdening the current thread with potentially performing amortized reclamation of tens of thousands of possibly unrelated retired objects, because calling **`retire`** may trigger amortized asynchronous reclamation.
- In contrast, the code snippet on the right side, retirement to the cohort must be executed synchronously in order for the object to be included as intended in synchronous reclamation of the associated cohort. Therefore, the call to **`retire_to_cohort`** must be synchronous. But in order not to burden worker threads with amortized asynchronous reclamation, **`retire_to_cohort`** does not try to perform asynchronous reclamation. Instead, if desired, the current thread may submit an asynchronous task to a dedicated thread pool to try to perform asynchronous reclamation (which attempts to reclaim unprotected retired objects, both cohort-associated and not).

Separating Cohort Object Retirement from Asynchronous Reclamation

Asynchronous Reclamation of Cohort Objects

If a cohort is long-lived, large numbers (e.g., billions) of objects may be retired to it. If such objects are not included in asynchronous reclamation, they would remain not reclaimed. Therefore, cohort objects need to be included in asynchronous reclamation.

Cohort Object Retirement Must to Be Synchronous

In order for an object to be included in synchronous reclamation in association with a cohort, the object must be retired to the cohort. Therefore, cohort object retirement needs to be synchronous.

Why Doesn't `retire_to_cohort` Try to Invoke Asynchronous Reclamation Implicitly?

As indicated above cohort-object retirement must be synchronous. However, it is often desirable to invoke asynchronous reclamation asynchronously in order to avoid burdening the thread retiring the object (typically a worker thread) with reclaiming a large number (e.g., tens of thousands) of (possibly unrelated) retired objects.

In contrast, the retirement of non-cohort objects (using `retire`) by convention may implicitly invoke asynchronous reclamation. If desired, a user may retire a non-cohort object `obj` asynchronously to avoid inline asynchronous reclamation as follows (assuming an executor associated with a separate execution resource):

```
    executor_.add([obj] { obj->retire(); });
```

However, the retirement of a cohort object needs to be synchronous, as mentioned above. Therefore a free function hazard_pointer_asynchronous_reclamation() is added, so that users can write the cohort equivalent to the above non-cohort code snippet as follows:

```
    obj->retire_to_cohort(cohort_);
    executor_.add([] { hazard_pointer_asynchronous_reclamation(); });
```

References

- P2530R3: Hazard Pointers for C++26 (2023-03-02).
- P3135R1: Hazard Pointer Extensions (2024-04-12).
- Folly: Facebook Open-source Library.
- CppCon 2021: *Hazard Pointer Synchronous reclamation beyond Concurrency TS2*