Hazard Pointer Synchronous Reclamation Beyond Concurrency TS2

MAGED MICHAEL

Cppcon
The C++ Conference

2021
October 24-29

Basic Hazard Pointer Algorithm

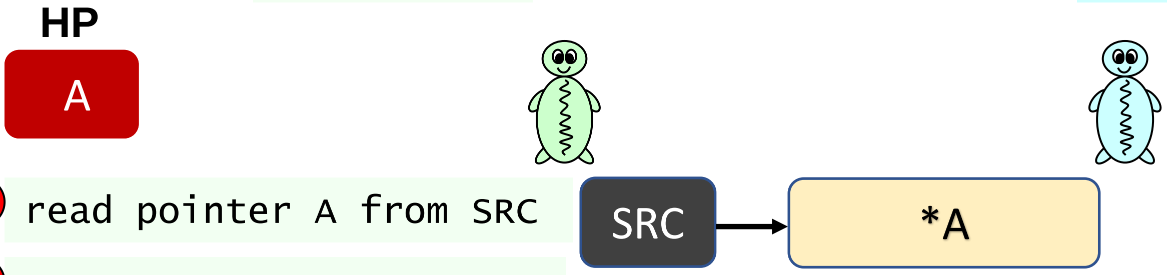
Hazard pointers **protect** access to **objects** that may be **removed** concurrently**.**

A **hazard pointer** is a **single-writer** multi-reader pointer.

If a hazard pointer **points to an object**
    before its removal,
        then the object will **not be reclaimed**
            as long as the hazard pointer **remains unchanged**.

Protector

Remover / Reclaimer



Protector:

```
1 read pointer A from SRC
2 set HP to A
3 if SRC == A
5   Safe to use pointer A
6 clear HP
```

Remover / Reclaimer:

```
4 remove A from SRC
7 if HP != A
8   Safe to delete A
```

SAFE ACCESS

SAFE RECLAMATION

Concurrency TS2 Essential Hazard Pointer Interface

Hazard pointers

```
class hazard_pointer {
  template <typename T> T* protect(const std::atomic<T*>& src) noexcept;
};

hazard_pointer make_hazard_pointer();
```

Base class for protectable objects

```
template <typename T> class hazard_pointer_obj_base<T> {
  void retire() noexcept;
};
```

See N4895 (wg21.link/n4895) for working draft of Concurrency TS2

Example Using Hazard Pointers

Shared data, frequently-used, infrequently-updated

```
class Foo : public hazard_pointer_obj_base<Foo> { /* Foo members */ };
```

```
Result read_and_use(std::atomic<Foo*>& src, Func fn) { // Called frequently
 folly::hazard_pointer h = folly::make_hazard_pointer();
 Foo* ptr = h.protect(src);
 return fn(ptr); // ptr is protected
}
```

```
Void update(std::atomic<Foo*>& src, Foo* newptr) { // Called infrequently
  Foo* oldptr = src.exchange(newptr);
  oldptr->retire();
}
```

Why use hazard pointers?

- Fast (low nanoseconds) and scalable protection.
- Supports arbitrary protection duration (e.g., the user-defined `fn` is allowed to block or take long time)

Asynchronous Reclamation

- Asynchronous reclamation is invoked when the number of retired objects reaches some threshold:
  - In the Folly library:
    - The threshold is the max of 1000 and twice the number of hazard pointers in the process.
    - Otherwise, invoked every 2 seconds.

Folly open-source library: `github.com/facebook/folly` under `synchronization/Hazptr.h`

- Steps:
  - Extract retired objects from lists in the (global) domain structure.
  - Read hazard pointer values
  - Match addresses of retired objects with values read from hazard pointers.
  - Push matched objects back into the domain lists.
  - Reclaim unmatched objects.

- Bounds the number of not-yet-reclaimed objects to approx. the number of hazard pointers

- No guarantee for the timing of reclamation of individual objects.

Is Asynchronous Reclamation Always Enough?

Example: Removed objects are reclaimed immediately

```
~Foo() { use_resource_X (); }
```

```
Foo* ptr = new Foo;
```

Remove ptr

```
delete ptr
```

```
shutdown_resource_X()
```

No problem with immediate reclamation

Is Asynchronous Reclamation Always Enough? No

Same example but using hazard pointer deferred reclamation

```
~Foo() { use_resource_X (); }
```

```
Foo* ptr = new Foo;
```

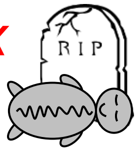
Remove ptr

```
ptr-retire()
```

```
shutdown_resource_X()
```

Asynchronous reclamation:
```
delete ptr
```

Using unavailable resource *X*



Some use cases need synchronous reclamation

Synchronous Reclamation in Concurrency TS2

Global cleanup

```
void hazard_pointer_clean_up(/* domain */);
```

- A call to **`hazard_pointer_clean_up`** guarantees that all objects that were
    - retired to the domain
    - and not protected by hazard pointers
  - before this call
  - will be reclaimed before this call returns

Same Example Using Global Cleanup

```
~Foo() { use_resource_X (); }
```

```
Foo* ptr = new Foo;
```

Remove ptr

```
ptr-retire()
```

Synchronous reclamation:

```
hazard_pointer_clean_up()
delete ptr
```

```
shutdown_resource_X()
```

No problem - No use of unavailable resource

Concurrency TS2 Global Cleanup

Simple and strong semantics

Performance?

Global Cleanup Implementation

- Must collect all retired objects (including in private caches)
- Must check all hazard pointers against collected objects
- Must complete reclamation of all unprotected objects
- Must wait for concurrent asynchronous reclamation.

Slow

Not Scalable

Effect on thread-local caches of retired objects
Even when not using global cleanup

Without global cleanup

```
SequentialObjList list_;
```

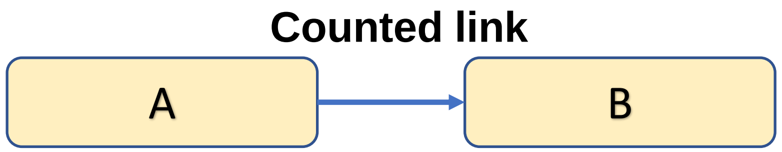
With global cleanup

```
SharedObjList list_;
```

Extra synchronization

Removed private cache from Folly
Using sharded domain lists instead

Is One Call to Global Cleanup Always Enough? No



- B is retired only when A is reclaimed.
- User removes A (implicitly removing B)
- A has no more inbound links. A is automatically retired.
- User removes all objects with dependence on resource X.
- User calls hazard_pointer_clean_up.
- A is reclaimed. B is automatically retired (as a result of calling hazard_pointer_clean_up).
- Resource X is shutdown.
- B is not yet reclaimed.

Solutions for Dependent Retirement

In User Code

- Maintain an indicator of automatic dependent retirement.
- Clear indicator before invoking global cleanup.
- Repeat as needed if the indicator is set.
- Problem: Where is user code?

In Library

- Integrated link counting.
- Automatically repeat global cleanup if dependent retirement happens.
- Implication: Asynchronous reclamation must be transitive and keep repeating whenever dependent retirement happens.
- Folly supports transitive global cleanup.
- Stronger semantics than TS2 global cleanup.

How about Custom Domains?

```
template <typename Key> class Container {
  struct Obj : hazard_pointer_obj<Obj> { Key k; /* etc */ };

  hazard_pointer_domain dom_;
  // other members

  void erase(Key k) {
    Obj* obj = foo_remove_obj(k);
    obj->retire(dom_);
  }
};
```

- Works. The domain and all objects retired to it are destroyed before the the completion of destruction of Container.
- Problem: High setup overhead of constructing/destroying per custom domain hazard pointers.
- Even worse if many instances of Container are used by thousands of threads.

Is there a good solution using the default domain?
Is there any good solution?

Fast Scalable Robust Synchronous Reclamation

Cohorts

- A cohort is a set of retired objects.
- A retired object can belong to at most one cohort.
- The completion of a cohort's destructor guarantees the completion of all deleters of objects that belong/belonged to the cohort.

- Weaker and stronger semantics than (TS2) hazard_pointer_clean_up.
- Weaker: Doesn't cover all retired objects.
- Stronger: Seamlessly handles dependent retirement.

Cohort Interface

Cohort class

```
class hazard_pointer_cohort {
  hazard_pointer_cohort();
  // Not copyable or movable
  ~hazard_pointer_cohort();
};
```

Base class for protectable cohort objects

```
template <typename T> class hazard_pointer_cohort_obj_base<T> {
  void set_cohort(hazard_pointer_cohort* cohort);
  void retire(); // Same as hazard_pointer_obj_base
};
```

Simple Cohort Use Example

```
template <typename Key> class Container {
  struct Obj : hazard_pointer_cohort_obj<Obj> { Key k; /* etc */ };

  hazard_pointer_cohort cohort_;
  // other members

  void erase(Key k) {
    Obj* obj = foo_remove_obj(k);
    obj->set_cohort(&cohort_);
    obj->retire();
  }
};
```

Cohort Implementation Requirements

- Fast and scalable operations including destruction
- Seamless support for dependent retirement (objects in the same cohort)
- Seamless support for hierarchies:
    - E.g., Maps that contain maps that contain maps etc. Destruction of a cohort leads to the destruction of other cohorts etc.
- Interoperation with asynchronous reclamation.

Basic Cohort Implementation

```
class hazard_pointer_cohort {
  bool pushed_objects_to_domain_{false}; // opt
  SharedList retiredObjects_; // opt

  void pushRetiredObject(Obj* obj); // opt
  // etc
};
```

```
class hazard_pointer_domain {
  LockedHeadOnlyList cohortObjects_;
  // etc
};
```

LockedHeadOnlyList:

- Pop all operation locks list for other pop all operations.
- Push operations are lock-free.
- ~hazard_pointer_cohort()
    - Pops all objects and locks list
    - Extracts objects with matching cohort
    - Pushes back unmatching objects and unlocks list
    - Reclaim matching objects
- Why lock? Concurrent cohort destructors.

Asynchronous Reclamation Revisited

- Pop all objects from domain cohort object list and lock list
    - Why lock? Concurrent cohort destructors.
- Read all hazard pointer values.
- Match addresses of extracted objects against hazard pointers.
- **Reclaim unmatched objects.(under lock).**
- Push matched objects back into domain cohort object list and unlock list.

LockedHeadOnlyList (revisited):

- Reentrant lock:
    - Why? Asynchronous reclamation of objects that lead to destruction of a cohorts.

Can the Domain Cohort Object List Be Sharded?

- Asynchronous reclamation of (maybe non-cohort) object A.
- Destruction of object A entails the destruction of cohort C0.
- Destruction of cohort C0 requires locking shard 0.
- Cohort C0 contains object B0.
- Destruction of object B0 entails the destruction of cohort C1 (e.g., map of maps).
- Destruction of cohort C1 requires locking shard 1.

- Now also consider the asynchronous reclamation of object A' that leads to locking shards 1 and 0 in the reverse order.

- **DEADLOCK**

- **We cannot shard the domain cohort object list** ***(or can we?)***

Wait. You Said Fast and Scalable

Cohort Structures Revisited

```
class hazard_pointer_cohort {
  bool pushed_objects_to_domain_{false}; // opt
  SharedList retiredObjects_; // opt
  SharedList reclaimableObjects_;

  void pushRetiredObject(Obj* obj);
  void pushReclaimableObject(Obj* obj);
  // etc
};
```

```
class hazard_pointer_domain {
  LockedHeadOnlyList cohortObjects_[kNumShards]; // Sharded!!!
  // etc
};
```

Asynchronous Reclamation Re-Revisited

- Pop all objects from domain cohort object list shards and lock shards
- Read all hazard pointer values.
- Match addresses of extracted objects against hazard pointers.
- **Don’t reclaim unmatched objects (under lock).**
- **Instead: Push each object into its cohort’s reclaimableObjects_ list.**
- Push matched objects back into their domain cohort object list shards and unlock shards.

Occasionally objects in a cohort’s reclaimableObjects_ list are reclaimed when retiring objects to the cohort (by a related thread).

Side-effect: Cohort objects are reclaimed only by related threads (threads that retired objects to the specific cohort).

No reclamation while holding any domain shard locks.

The locks on the domain shards don’t even need to be reentrant.

Experience with Synchronous Reclamation in Folly

- Open-source: `github.com/facebook/folly` under `synchronization/Hazptr.h`
- 2017: Need for synchronous reclamation.
- 2017: Global cleanup as in TS2.
- 2018: Stronger global cleanup for linked objects (ConcurrentHashMap).
- Global cleanup is too expensive for every ConcurrentHashMap destruction.
- Global cleanup as needed AFTER ConcurrentHashMap destruction doesn't always work. Sometimes it is too late.
- 2018: Sharded cohorts (1st try). Almost all users happy. One user: deadlock (maps of maps).
- 2018: Unsharded domain cohort object list. Almost all users happy.
- Global cleanup does not apply to cohort object.
- 2018: Global cleanup deprecated. Still supported for testing and microbenchmarking.
- 2020: One user: High frequency of retiring objects. Unsharded list grew out of control.
- 2020: Sharded cohorts without reclamation under lock. Fast. Scalable. Robust.

Quasi-Synchronous Reclamation

- Some users need reclamation of some objects to happen “soon” but not necessarily synchronously.
- Global cleanup can be used, but it is an overkill.
- Instead: On-demand asynchronous reclamation?
    - Domain-level?
    - Cohort-level?
    - Object-level? Watch only protecting hazard pointers?

Synchronous Reclamation Roadmap

- Concurrency TS2 (draft in wg21.link/n4895):
    - hazard_pointer_clean_up
    - Feedback welcome
- Proposal for C++26: Minimal useful subset of TS2.
    - Why minimal? High confidence in proposal. Require less committee time.
- Cohorts:
    - High confidence in usefulness based on Folly experience.
    - Why not C++26? Maybe. But unlikely. Complex wording takes committee time.
- Global cleanup?
    - As in TS2? Depends on TS2 feedback.
    - Transitive cleanup after integration of link counting (as in Folly)?
    - Maybe deprecate global cleanup in Folly?
- Other extensions:
    - Quasi-synchronous on-demand asynchronous reclamation?

Minimal Proposal for C++26

```
class hazard_pointer_obj_base {
 public:
  void retire(D d = D()) noexcept;
};

class hazard_pointer {
 public:
  hazard_pointer() noexcept; // Empty
  hazard_pointer(hazard_pointer&&) noexcept;
  hazard_pointer& operator=(hazard_pointer&&) noexcept;
  ~hazard_pointer();
  [[nodiscard]] 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 make_hazard_pointer(); // Nonempty

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

Cohort Interface (preliminary)

```
class hazard_pointer_cohort {
 public:
  hazard_pointer_cohort();
  ~hazard_pointer_cohort();
};

class hazard_pointer_cohort_obj_base {
 public:
  void set_cohort(hazard_pointer_cohort* cohort) noexcept;
  void retire(D d = D()) noexcept;
};
```

```
// Extend hazard_pointer_obj_base (subject to ABI-compatibility)
class hazard_pointer_obj_base {
 public:
  void set_cohort(hazard_pointer_cohort* cohort) noexcept;
  void retire(D d = D()) noexcept;
};
```

References

- Folly `github.com/facebook/folly` under `folly/synchronization/Hazptr.h`
- Working Draft, Extensions to C++ for Concurrency Version 2 (wg21.link/n4895).
- Hazard Pointers: Safe Memory Reclamation for Lock-Free Objects, *IEEE Transactions on Parallel and Distributed Systems.* 15 (8): 491–504, June 2004.

Watch CPPCON 2021 Talk on Concurrency TS2

The Upcoming Concurrency TS Version 2 for Low-Latency and Lockless Synchronization (with Paul McKenney and Michael Wong)