C++ Lock-free Hazard Pointer

2020.07.19 SF-Zhou

1. Safe Reclamation Methods

Folly 的 Hazard Pointer 实现中有一段注释，详细描述了 C++ 里几种主流的安全内存回收方法，列表如下：

	优点	缺点	场景
Locking	易用	读高开销 / 抢占式 / 死锁	非性能敏感
Reference Couting	自动回收 / 线程无关 / 免于死锁	读高开销 / 抢占式	需要自动回收
Read-copy-update (RCU)	简单 / 高速 / 可拓展	对阻塞敏感	性能敏感
Hazard Pointer	高速 / 可拓展 / 阻塞场景可用	性能依赖 TLS	性能敏感 / 读多写少

C++ 标准库中提供了锁和引用计数方案。锁的缺点很明显，无论是哪种锁，在读的时候都会产生较大的开销。引用计数则相对好一些，但每次读取都需要修改引用计数，高并发场景下这样的原子操作也会成为性能瓶颈，毕竟原子加对应的 CPU 指令 `lock add` 也可以看成是微型锁。

Linux 内核中提供了 RCU 方法，笔者目前对此还没有太多的了解。本文主要介绍 Hazard Pointer，一种无锁编程中广泛使用的安全内存回收方法，适用于需要高性能读、读多写少的场景。其论文可参考文献 1，标准草案可参考文献 2，代码实现可参考 Folly 中的 HazPtr 。

2. Hazard Pointer

首先回忆下引用计数的做法：

```
#include <atomic>
#include <memory>

template <class T>
class ReferenceCount {
 public:
  ReferenceCount(std::unique_ptr<T> ptr) : ptr_(std::move(ptr)), cnt_(1) {}

  T *Ptr() const { return ptr_.get(); }

  ReferenceCount *Ref() {
    ++cnt_;
    return this;
  }

  void Deref() {
    if (--cnt  == 0) {
```

```
      delete this;
    }
  }

 private:

  std::unique_ptr<T> ptr_;
  std::atomic_uint32_t cnt_;
};
```

仔细观察可以发现：

1. 每一次的读取操作对应引用计数中增加的数值 1；
2. 当所有的读取操作都完成时引用计数归 0，此时内存可以安全回收。

总结起来，当对象正在使用时，就不能回收内存。每一个“正在使用”都需要对应一个标记，引用计数使用的标记是计数数值一，对应的原子操作性能问题就成为它无法摆脱的原罪。而 Hazard Pointer 使用的标记更为轻巧，一般通过在链表中标记该对象的指针实现，回收时如果发现链表中有对应的指针就不进行内存回收，将标记的复杂度转移到回收部分，也就更适合读多写少的场景。Hazard Pointer 的简单实现（在线执行）：

```
#include <atomic>
#include <memory>
#include <unordered_set>

template <class T>
struct HazardPointer {
 public:
```

```
 class Holder {
  public:
   explicit Holder(HazardPointer<T> *pointer) : pointer_(pointer) {}
   Holder(const HazardPointer &) = delete;
   ~Holder() { pointer_->Release(); }

   T *get() const { return pointer_->target_.load(std::memory_order_acquire); }
   operator bool() const { return get(); }
   T *operator->() const { return get(); }
   T &operator*() const { return *get(); }

  private:
   HazardPointer<T> *pointer_;
 };

public:
 ~HazardPointer() {
   if (next_) {
     delete next_;
   }
 }

 void Release() {
   target_.store(nullptr, std::memory_order_release);
   active_.clear(std::memory_order_release);
 }

 static Holder Acquire(const std::atomic<T *> &target) {
   auto pointer = HazardPointer<T>::Alloc();

   do {
```

```
    pointer->target_ = target.load(std::memory_order_acquire);
  } while (pointer->target_.load(std::memory_order_acquire) !=
           target.load(std::memory_order_acquire));
  return Holder(pointer);
}


static void Update(std::atomic<T *> &target, T *new_target) {
  auto old = target.exchange(new_target);
  Retire(old);
}

static void Reclaim() {
  // collect in-use pointers
  std::unordered_set<T *> in_use;
  for (auto p = head_list_.load(std::memory_order_acquire); p; p = p->next_) {
    in_use.insert(p->target_);
  }

  // delete useless pointers
  List *retire_head = nullptr;
  List *retire_tail = nullptr;

  auto p = retire_list_.exchange(nullptr);
  while (p != nullptr) {
    auto next = p->next;

    if (in_use.count(p->target.get()) == 0) {
      delete p;
    } else {
      p->next = retire_head;
```

```
        retire_head = p;
        if (retire_tail == nullptr) {
          retire_tail = p;
        }
      }


      p = next;
    }

    if (retire_head) {
      // push to retired list again
      auto &tail = retire_tail->next;
      do {
        tail = retire_list_.load(std::memory_order_acquire);
      } while (!retire_list_.compare_exchange_weak(tail, retire_head));
    }
  }

private:
  static HazardPointer<T> *Alloc() {
    for (auto p = head_list_.load(std::memory_order_acquire); p; p = p->next_) {
      if (!p->active_.test_and_set()) {
        return p;
      }
    }

    auto p = new HazardPointer<T>();
    p->active_.test_and_set();
    do {
      p->next_ = head_list_.load(std::memory_order_acquire);
```

```
    } while (!head_list_.compare_exchange_weak(p->next_, p));
    return p;
  }

  static void Retire(T *ptr) {

    auto p = new List;
    p->target.reset(ptr);
    do {
      p->next = retire_list_.load(std::memory_order_acquire);
    } while (!retire_list_.compare_exchange_weak(p->next, p));

    if (++retire_count_ == 1000) {
      retire_count_ = 0;
      Reclaim();
    }
  }

private:
  struct List {
    std::unique_ptr<T> target{nullptr};
    List *next = nullptr;
  };

private:
  std::atomic<T *> target_{nullptr};
  HazardPointer<T> *next_;
  std::atomic_flag active_ = ATOMIC_FLAG_INIT;

  static std::atomic<HazardPointer<T> *> head_list_;
  static std::atomic<uint32_t> retire_count_;
```

```
  static std::atomic<List *> retire_list_;
};

template <class T>
std::atomic<HazardPointer<T> *> HazardPointer<T>::head_list_{nullptr};

template <class T>
std::atomic<uint32_t> HazardPointer<T>::retire_count_{0};
template <class T>
std::atomic<typename HazardPointer<T>::List *> HazardPointer<T>::retire_list_{
    nullptr};

// example
#include <cstdint>
#include <iostream>
#include <thread>
#include <vector>

std::atomic<int> g_count{0};
class A {
 public:
  explicit A(int value) : value_(value) { ++g_count; }
  ~A() { --g_count; }
  int Value() { return value_; }

 private:
  int value_;
};

int main() {
  std::atomic<A *> target{new A(0)};
```

```
  constexpr int N = 8;
  constexpr int M = 1000000;
  constexpr int W = 1000;
  std::vector<std::thread> threads;

  std::atomic<uint64_t> sum{0};
  for (int t = 0; t < N; ++t) {
    threads.emplace_back([&, t] {
      for (int i = 0; i < M; ++i) {
        if (i % W == 0) {
          // write less
          HazardPointer<A>::Update(target, new A(t * M + i));
        } else {
          // read more
          auto holder = HazardPointer<A>::Acquire(target);
          sum.fetch_add(holder->Value());
        }
      }
    });
  }

  // wait finish
  for (auto &thread : threads) {
    thread.join();
  }

  HazardPointer<A>::Reclaim();
  printf("Remain: %d\n", g_count.load());
}
```

简单解释下原理。Hazard Pointer 申请读取时，会在对象链表中申请一个空位置，将对象的指针写入该位置中，读取结束时将该位置重新置空即可；而发生更新时，将更新替换下来的旧指针加入退休列表里，退休列表积攒到一定程度时则检查哪些对象已经不在对象链表中，不再使用的则可以执行删除。

如果使用 std::shared_ptr 实现上述逻辑，你会发现它的执行速度还要高于上述代码。原因在于这里实现的 Hazard Pointer 没有使用非对称内存屏障和线程本地存储优化。如果仔细观察，可以发现 Acquire 函数中使用顺序一致性内部屏障 pointer->target_ = ...，x86 平台上会翻译为 mfence 指令，与 lock add 指令相比也不遑多让。在读多写少的前提下，可以将读写两边的屏障替换为非对称内存屏障，将读部分的开销转移到写部分中。可参考先前内存屏障博文中的介绍。

另一方面，Hazard Pointer 的高性能依赖于平台上线程本地存储（TLS）的性能。单纯使用 CAS 更新全局的对象链表和退休列表的性能太低，可以使用 TLS 做为缓冲层，这样大部分时间都是更新本线程的数据。这部分可以参考 Folly 中的 ThreadLocalPtr，其中实现了遍历所有线程 TLS 的黑魔法。

References

1. "Hazard Pointers: Safe Memory Reclamation forLock-Free Objects", *Maged M. Michael*
2. "Proposed Wording for Concurrent Data Structures: Hazard Pointer and Read-CopyUpdate (RCU)", *Open Standards*