C++ Lock-free Atomic Shared Pointer

2020.06.14 SF-Zhou

1. Shared Pointer

使用原子量引用计数实现一个简易的共享指针：

```cpp
#include <atomic>
#include <cassert>
#include <iostream>
#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_;
};

template <class T>
class SharedPtr {
 public:
  SharedPtr(std::unique_ptr<T> ptr = nullptr) noexcept
      : rc_(new ReferenceCount<T>(std::move(ptr))) {}

  ~SharedPtr() { rc_->Deref(); }

  T *Load() const { return rc_->Ptr(); }

  void Store(const SharedPtr &other) {
    auto old = rc_;
    rc_ = other.Copy();
    old->Deref();
  }

 private:
  ReferenceCount<T> *Copy() const { return rc_->Ref(); }

 private:
  ReferenceCount<T> *rc_;
```

```
};

class A {
 public:
  A(int value) : value_(value) {
    std::cout << "A(" << value_ << ")" << std::endl;
  }
  ~A() { std::cout << "~A(" << value_ << ")" << std::endl; }

  int Value() const { return value_; }

 private:
  int value_ = 0;
};

int main() {
  SharedPtr<A> a;
  assert(a.Load() == nullptr);

  SharedPtr<A> b(std::make_unique<A>(7));
  assert(b.Load() != nullptr);
  A &r = *b.Load();
  assert(r.Value() == 7);

  a.Store(b);
  assert(a.Load());
  assert(a.Load()->Value() == 7);

  b.Store(SharedPtr<A>(std::make_unique<A>(9)));
  assert(b.Load());
```

```
  assert(b.Load()->Value() == 9);

  a.Store(SharedPtr<A>());
  assert(a.Load() == nullptr);
}
```

标准库中的 std::shared_ptr 的实现与之类似。仔细观察可以发现 ReferenceCount 是线程安全的，无论是 Ref 还是 Unref，使用原子量都可以保证计数准确，并且有且仅有一次析构。但 SharedPtr 中对 rc_ 的操作并不是线程安全的，例如两个线程同时执行 Store，可能会对同一个 rc_ 对象重复执行两次 Deref。所以只能支持单线程写或多线程读。

2. Atomic Shared Pointer

如果希望线程安全，最简单的方案自然是加锁。可以在 SharedPtr 的 Load / Store / Copy 函数中加自旋锁或互斥锁，标准库也是这样实现的，但显然锁的开销有点大。

仔细分析这里的 Store 的过程，一来需要将原先的计数 -1，二来需要从新计数中复制指针并 +1 计数，如果可以原子化的实现这一步骤，就可以实现无锁的共享指针。直觉地写出如下的代码：

```
template <class T>
class SharedPtr {
 public:
  SharedPtr(std::unique_ptr<T> ptr = nullptr) noexcept {
    rc_.store(new ReferenceCount<T>(std::move(ptr)));
  }
```

```
  ~SharedPtr() { rc_.load()->Deref(); }

  T *Load() const { return rc_.load()->Ptr(); }

  void Store(const SharedPtr &other) {
    auto copy = other.rc_.load()->Ref();
    auto old = rc_.exchange(copy);
    old->Deref();
  }

 private:
  std::atomic<ReferenceCount<T> *> rc_;
};
```

然而随意地多线程跑下 Store，会发现这段代码是不靠谱的。仔细分析 Store 的过程，可以发现 other.rc_.load()->Ref() 并不是原子的。当一个线程完成 other.rc_.load() 后，可能另一个线程执行 old->Deref()，此时引用计数对象已经完成析构，也就没法再执行后面的 Ref 操作。这里的 Load 也是如此。换句话说，这里需要保证计数对象存活地情况下执行 Ref()。

仔细思考下，这里无法基于可能被析构的 rc_ 做引用计数的原子加。一个可行的方案是增加本地引用计数。除了 rc_ 指向的全局引用计数外，再增加一个本地引用计数变量。在 Load 时首先原子地增加本地引用计数，并在 Release 时删去。那么如何使得 rc_ 也能感知到本地引用计数的存在、不至于提前“自杀”呢？一个简单粗暴的方法是预支。首先在 rc_ 指向的全局引用计数上增加一个大计数，用来表示共享指针提前预支的引用计数量，

保证它不会因为外界的原因先析构掉。后面每次 Load 的时候，从预支的计数中取出，CAS 更新本地计数剩余量，最后 Release 时再减去剩下的预支计数量。

本地引用计数如果使用独立的变量存储，就需要使用 128 位的 CAS 操作了，但这个操作是很低效的。好在 x64 平台上，指针的高 16 位是全 0 的，刚好可以用来放本地引用计数，也就可以直接使用 64 位的 CAS 操作了。这也就是 folly 中的无锁共享指针的实现方法，简化实现如下：

```
#include <atomic>
#include <iostream>
#include <memory>
#include <thread>
#include <vector>

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(uint32_t cnt = 1) {
    cnt_.fetch_add(cnt);
    return this;
  }

  void Deref(uint32_t cnt = 1) {
    if (cnt_.fetch_sub(cnt) == cnt) {
```

```
      delete this;
    }
  }

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

template <class T>
class AtomicSharedPtr;

template <class T>
class SharedPtr {
 public:
  SharedPtr(std::unique_ptr<T> ptr = nullptr) noexcept
      : rc_(new ReferenceCount<T>(std::move(ptr))) {}

  ~SharedPtr() { rc_->Deref(); }

  T *Load() const { return rc_->Ptr(); }

  void Store(const SharedPtr &other) {
    auto old = rc_;
    rc_ = other.Copy();
    old->Deref();
  }

 private:
  SharedPtr(ReferenceCount<T> *rc) : rc_(rc) {}
```

```
  friend class AtomicSharedPtr<T>;

  ReferenceCount<T> *Copy() const { return rc_->Ref(); }

 private:
  ReferenceCount<T> *rc_;
};

template <class T>
class AtomicSharedPtr {
 public:
  ~AtomicSharedPtr() { Release(rc_.load()); }

  SharedPtr<T> Load() { return SharedPtr<T>(Acquire()); }

  void Store(SharedPtr<T> &ptr) {
    ptr.rc_->Ref(kCnt);
    auto old = rc_.exchange((uint64_t)ptr.rc_ | (kCnt << 48));
    Release(old);
  }

 private:
  ReferenceCount<T> *Acquire() {
    uint64_t local = 0;
    do {
      local = rc_.load();
    } while (!rc_.compare_exchange_weak(local, local - (1ull << 48)));
    return reinterpret_cast<ReferenceCount<T> *>(local & (-1ull >> 16));
  }
```

```
  static void Release(uint64_t local) {
    if (local == 0) {
      return;
    }
    uint32_t local_cnt = (local >> 48);
    reinterpret_cast<ReferenceCount<T> *>(local & (-1ull >> 16))
        ->Deref(local_cnt);
  }

 private:
  static constexpr uint64_t kCnt = 0x2000;
  std::atomic<uint64_t> rc_{0};
};

std::atomic<int32_t> cnt{0};
class A {
 public:
  A(int value) : value_(value) { ++cnt; }

  ~A() { --cnt; }

  int Value() const { return value_; }

 private:
  int value_ = 0;
};

int main() {
  constexpr uint32_t N = 1000000;
  constexpr uint32_t T = 4;
```

```
  AtomicSharedPtr<A> x;
  AtomicSharedPtr<A> y;
  std::vector<std::thread> threads;
  for (uint32_t t = 0; t < T; ++t) {
    threads.emplace_back([&] {
      for (uint32_t i = 0; i < N; ++i) {
        SharedPtr<A> a(std::make_unique<A>(t * N + i));
        x.Store(a);
        SharedPtr<A> b = x.Load();
        y.Store(b);
      }
    });
  }

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

  return cnt.load() == 1 ? 0 : -1;
}
```

点击此处查看线上运行结果。

如果本地引用计数不足了怎么办？继续预支一笔就好。设定一个阈值，小于阈值时就预支一笔，并 CAS 更新本地引用计数，使其始终保持足够的余额，可参考文献 3 中 folly 的实现。

以下代码截取自 folly，删除了 folly 本身的一些调用，可直接在 C++ 11 中使用：

```cpp
#include <atomic>
#include <cassert>
#include <climits>
#include <memory>
#include <thread>
#include <type_traits>
#include <vector>

// copy from
// https://github.com/facebook/folly/blob/master/folly/concurrency/detail/AtomicSharedPtr
// https://github.com/facebook/folly/blob/master/folly/PackedSyncPtr.h
// https://github.com/facebook/folly/blob/master/folly/concurrency/AtomicSharedPtr.h

#if !__x86_64__
#error "PackedSyncPtr is x64 specific code."
#endif

namespace std {
namespace detail {

// This implementation is specific to libstdc++, now accepting
// diffs for other libraries.

// Specifically, this adds support for two things:
// 1) incrementing/decrementing the shared count by more than 1 at a time
// 2) Getting the thing the shared_ptr points to, which may be different from
//    the aliased pointer.

class shared_ptr_internals {
 public:
```

```
  template <typename T, typename... Args>
  static std::shared_ptr<T> make_ptr(Args &&... args) {
    return std::make_shared<T>(std::forward<Args...>(args...));
  }
  typedef std::__shared_count<std::_S_atomic> shared_count;
  typedef std::_Sp_counted_base<std::_S_atomic> counted_base;
  template <typename T>
  using CountedPtr = std::shared_ptr<T>;

  template <typename T>
  static counted_base *get_counted_base(const std::shared_ptr<T> &bar);

  static void inc_shared_count(counted_base *base, long count);

  template <typename T>
  static void release_shared(counted_base *base, long count);

  template <typename T>
  static T *get_shared_ptr(counted_base *base);

  template <typename T>
  static T *release_ptr(std::shared_ptr<T> &p);

  template <typename T>
  static std::shared_ptr<T> get_shared_ptr_from_counted_base(counted_base *base,
                                                             bool inc = true);

 private:
  /* Accessors for private members using explicit template instantiation */
  struct access_shared_ptr {
```

```
  typedef shared_count std::__shared_ptr<const void, std::_S_atomic>::*type;
  friend type fieldPtr(access_shared_ptr);
};

struct access_base {
  typedef counted_base *shared_count::*type;
  friend type fieldPtr(access_base);
};

struct access_use_count {
  typedef _Atomic_word counted_base::*type;
  friend type fieldPtr(access_use_count);
};

struct access_weak_count {
  typedef _Atomic_word counted_base::*type;
  friend type fieldPtr(access_weak_count);
};

struct access_counted_ptr_ptr {
  typedef const void
      *std::_Sp_counted_ptr<const void *, std::_S_atomic>::*type;
  friend type fieldPtr(access_counted_ptr_ptr);
};

struct access_shared_ptr_ptr {
  typedef const void *std::__shared_ptr<const void, std::_S_atomic>::*type;
  friend type fieldPtr(access_shared_ptr_ptr);
};
```

```
  struct access_refcount {
    typedef shared_count std::__shared_ptr<const void, std::_S_atomic>::*type;
    friend type fieldPtr(access_refcount);
  };

  template <typename Tag, typename Tag::type M>
  struct Rob {
    friend typename Tag::type fieldPtr(Tag) { return M; }
  };
};

template struct shared_ptr_internals::Rob<
    shared_ptr_internals::access_shared_ptr,
    &std::__shared_ptr<const void, std::_S_atomic>::_M_refcount>;
template struct shared_ptr_internals::Rob<
    shared_ptr_internals::access_base,
    &shared_ptr_internals::shared_count::_M_pi>;
template struct shared_ptr_internals::Rob<
    shared_ptr_internals::access_use_count,
    &shared_ptr_internals::counted_base::_M_use_count>;
template struct shared_ptr_internals::Rob<
    shared_ptr_internals::access_weak_count,
    &shared_ptr_internals::counted_base::_M_weak_count>;
template struct shared_ptr_internals::Rob<
    shared_ptr_internals::access_counted_ptr_ptr,
    &std::_Sp_counted_ptr<const void *, std::_S_atomic>::_M_ptr>;
template struct shared_ptr_internals::Rob<
    shared_ptr_internals::access_shared_ptr_ptr,
    &std::__shared_ptr<const void, std::_S_atomic>::_M_ptr>;
template struct shared_ptr_internals::Rob<
```

```
    shared_ptr_internals::access_refcount,
    &std::__shared_ptr<const void, std::_S_atomic>::_M_refcount>;

template <typename T>
inline shared_ptr_internals::counted_base *
shared_ptr_internals::get_counted_base(const std::shared_ptr<T> &bar) {
  // reinterpret_pointer_cast<const void>
  // Not quite C++ legal, but explicit template instantiation access to
  // private members requires full type name (i.e. shared_ptr<const void>, not
  // shared_ptr<T>)
  const std::shared_ptr<const void> &ptr(
      reinterpret_cast<const std::shared_ptr<const void> &>(bar));
  return (ptr.*fieldPtr(access_shared_ptr{})).*fieldPtr(access_base{});
}

inline void shared_ptr_internals::inc_shared_count(counted_base *base,
                                                   long count) {
  // Check that we don't exceed the maximum number of atomic_shared_ptrs.
  // Consider setting EXTERNAL_COUNT lower if this CHECK is hit.
  assert(base->_M_get_use_count() + count < INT_MAX);
  __gnu_cxx::__atomic_add_dispatch(&(base->*fieldPtr(access_use_count{})),
                                   count);
}

template <typename T>
inline void shared_ptr_internals::release_shared(counted_base *base,
                                                 long count) {
  // If count == 1, this is equivalent to base->_M_release()
  auto &a = base->*fieldPtr(access_use_count{});
  if (__gnu_cxx::__exchange_and_add_dispatch(&a, -count) == count) {
```

```
    base->_M_dispose();

    auto &b = base->*fieldPtr(access_weak_count{});
    if (__gnu_cxx::__exchange_and_add_dispatch(&b, -1) == 1) {
      base->_M_destroy();
    }
  }
}

template <typename T>
inline T *shared_ptr_internals::get_shared_ptr(counted_base *base) {
  // See if this was a make_shared allocation
  auto inplace = base->_M_get_deleter(typeid(std::_Sp_make_shared_tag));
  if (inplace) {
    return (T *)inplace;
  }
  // Could also be a _Sp_counted_deleter, but the layout is the same
  using derived_type = std::_Sp_counted_ptr<const void *, std::_S_atomic>;
  auto ptr = reinterpret_cast<derived_type *>(base);
  return (T *)(ptr->*fieldPtr(access_counted_ptr_ptr{}));
}

template <typename T>
inline T *shared_ptr_internals::release_ptr(std::shared_ptr<T> &p) {
  auto res = p.get();
  std::shared_ptr<const void> &ptr(
      reinterpret_cast<std::shared_ptr<const void> &>(p));
  ptr.*fieldPtr(access_shared_ptr_ptr{}) = nullptr;
  (ptr.*fieldPtr(access_refcount{})).*fieldPtr(access_base{}) = nullptr;
  return res;
```

```
}

template <typename T>
inline std::shared_ptr<T>
shared_ptr_internals::get_shared_ptr_from_counted_base(counted_base *base,
                                                       bool inc) {
  if (!base) {
    return nullptr;
  }
  std::shared_ptr<const void> newp;
  if (inc) {
    inc_shared_count(base, 1);
  }
  newp.*fieldPtr(access_shared_ptr_ptr{}) =
      get_shared_ptr<const void>(base);  // _M_ptr
  (newp.*fieldPtr(access_refcount{})).*fieldPtr(access_base{}) = base;
  // reinterpret_pointer_cast<T>
  auto res = reinterpret_cast<std::shared_ptr<T> *>(&newp);
  return std::move(*res);
}

template <class T>
class PackedSyncPtr {
  // This just allows using this class even with T=void.  Attempting
  // to use the operator* or operator[] on a PackedSyncPtr<void> will
  // still properly result in a compile error.
  typedef typename std::add_lvalue_reference<T>::type reference;

 public:
  /*
```

```
 * If you default construct one of these, you must call this init()
 * function before using it.
 *
 * (We are avoiding a constructor to ensure gcc allows us to put
 * this class in packed structures.)
 */
void init(T *initialPtr = nullptr, uint16_t initialExtra = 0) {
  auto intPtr = reinterpret_cast<uintptr_t>(initialPtr);
  assert(!(intPtr >> 48));
  data_ = intPtr;
  setExtra(initialExtra);
}

/*
 * Sets a new pointer.  You must hold the lock when calling this
 * function, or else be able to guarantee no other threads could be
 * using this PackedSyncPtr<>.
 */
void set(T *t) {
  auto intPtr = reinterpret_cast<uintptr_t>(t);
  auto shiftedExtra = uintptr_t(extra()) << 48;
  assert(!(intPtr >> 48));
  data_ = (intPtr | shiftedExtra);
}

/*
 * Get the pointer.
 *
 * You can call any of these without holding the lock, with the
 * normal types of behavior you'll get on x64 from reading a pointer
```

```
   * without locking.
   */
  T *get() const { return reinterpret_cast<T *>(data_ & (-1ull >> 16)); }
  T *operator->() const { return get(); }
  reference operator*() const { return *get(); }
  reference operator[](std::ptrdiff_t i) const { return get()[i]; }

  /*
   * Access extra data stored in unused bytes of the pointer.
   *
   * It is ok to call this without holding the lock.
   */
  uint16_t extra() const { return data_ >> 48; }

  /*
   * Don't try to put anything into this that has the high bit set:
   * that's what we're using for the mutex.
   *
   * Don't call this without holding the lock.
   */
  void setExtra(uint16_t extra) {
    assert(!(extra & 0x8000));
    auto ptr = data_ & (-1ull >> 16);
    data_ = ((uintptr_t(extra) << 48) | ptr);
  }

 private:
  uintptr_t data_;
};
```

```
static_assert(std::is_pod<PackedSyncPtr<void>>::value,
              "PackedSyncPtr must be kept a POD type.");
static_assert(sizeof(PackedSyncPtr<void>) == 8,
              "PackedSyncPtr should be only 8 bytes---something is "
              "messed up");

}  // namespace detail

template <typename T, typename CountedDetail = detail::shared_ptr_internals>
class atomic_shared_ptr {
  using SharedPtr = typename CountedDetail::template CountedPtr<T>;
  using BasePtr = typename CountedDetail::counted_base;
  using PackedPtr = detail::PackedSyncPtr<BasePtr>;

 public:
  atomic_shared_ptr() noexcept { init(); }
  explicit atomic_shared_ptr(SharedPtr foo) /* noexcept */
      : atomic_shared_ptr() {
    store(std::move(foo));
  }
  atomic_shared_ptr(const atomic_shared_ptr<T> &) = delete;

  ~atomic_shared_ptr() { store(SharedPtr(nullptr)); }
  void operator=(SharedPtr desired) /* noexcept */ {
    store(std::move(desired));
  }
  void operator=(const atomic_shared_ptr<T> &) = delete;

  bool is_lock_free() const noexcept {
    // lock free unless more than EXTERNAL_OFFSET threads are
```

```
    // contending and they all get unlucky and scheduled out during
    // load().
    //
    // TODO: Could use a lock-free external map to fix this
    // corner case.
    return true;
  }

  SharedPtr load(
      std::memory_order order = std::memory_order_seq_cst) const noexcept {
    auto local = takeOwnedBase(order);
    return get_shared_ptr(local, false);
  }

  /* implicit */ operator SharedPtr() const { return load(); }

  void store(SharedPtr n, std::memory_order order =
                              std::memory_order_seq_cst) /* noexcept */ {
    auto newptr = get_newptr(std::move(n));
    auto old = ptr_.exchange(newptr, order);
    release_external(old);
  }

  SharedPtr exchange(
      SharedPtr n,
      std::memory_order order = std::memory_order_seq_cst) /* noexcept */ {
    auto newptr = get_newptr(std::move(n));
    auto old = ptr_.exchange(newptr, order);

    SharedPtr old_ptr;
```

```
  if (old.get()) {
    old_ptr = get_shared_ptr(old);
    release_external(old);
  }

  return old_ptr;
}

bool compare_exchange_weak(
    SharedPtr &expected, const SharedPtr &n,
    std::memory_order mo = std::memory_order_seq_cst) noexcept {
  return compare_exchange_weak(expected, n, mo, mo);
}
bool compare_exchange_weak(SharedPtr &expected, const SharedPtr &n,
                           std::memory_order success,
                           std::memory_order failure) /* noexcept */ {
  auto newptr = get_newptr(n);
  PackedPtr oldptr, expectedptr;

  oldptr = takeOwnedBase(success);
  if (!owners_eq(oldptr, CountedDetail::get_counted_base(expected))) {
    expected = get_shared_ptr(oldptr, false);
    release_external(newptr);
    return false;
  }
  expectedptr = oldptr;  // Need oldptr to release if failed
  if (ptr_.compare_exchange_weak(expectedptr, newptr, success, failure)) {
    if (oldptr.get()) {
      release_external(oldptr, -1);
```

```
      }
      return true;
    } else {
      if (oldptr.get()) {
        expected = get_shared_ptr(oldptr, false);
      } else {
        expected = SharedPtr(nullptr);
      }
      release_external(newptr);
      return false;
    }
  }
  bool compare_exchange_weak(
      SharedPtr &expected, SharedPtr &&desired,
      std::memory_order mo = std::memory_order_seq_cst) noexcept {
    return compare_exchange_weak(expected, desired, mo, mo);
  }
  bool compare_exchange_weak(SharedPtr &expected, SharedPtr &&desired,
                             std::memory_order success,
                             std::memory_order failure) /* noexcept */ {
    return compare_exchange_weak(expected, desired, success, failure);
  }
  bool compare_exchange_strong(
      SharedPtr &expected, const SharedPtr &n,
      std::memory_order mo = std::memory_order_seq_cst) noexcept {
    return compare_exchange_strong(expected, n, mo, mo);
  }
  bool compare_exchange_strong(SharedPtr &expected, const SharedPtr &n,
                               std::memory_order success,
                               std::memory_order failure) /* noexcept */ {
```

```
    auto local_expected = expected;
    do {
      if (compare_exchange_weak(expected, n, success, failure)) {
        return true;
      }
    } while (local_expected == expected);

    return false;
  }
  bool compare_exchange_strong(
      SharedPtr &expected, SharedPtr &&desired,
      std::memory_order mo = std::memory_order_seq_cst) noexcept {
    return compare_exchange_strong(expected, desired, mo, mo);
  }
  bool compare_exchange_strong(SharedPtr &expected, SharedPtr &&desired,
                               std::memory_order success,
                               std::memory_order failure) /* noexcept */ {
    return compare_exchange_strong(expected, desired, success, failure);
  }

 private:
  // Matches packed_sync_pointer.  Must be > max number of local
  // counts.  This is the max number of threads that can access this
  // atomic_shared_ptr at once before we start blocking.
  static constexpr unsigned EXTERNAL_OFFSET{0x2000};
  // Bit signifying aliased constructor
  static constexpr unsigned ALIASED_PTR{0x4000};

  mutable std::atomic<PackedPtr> ptr_;
```

```
void add_external(BasePtr *res, int64_t c = 0) const {
  assert(res);
  CountedDetail::inc_shared_count(res, EXTERNAL_OFFSET + c);
}
void release_external(PackedPtr &res, int64_t c = 0) const {
  if (!res.get()) {
    return;
  }
  int64_t count = get_local_count(res) + c;
  int64_t diff = EXTERNAL_OFFSET - count;
  assert(diff >= 0);
  CountedDetail::template release_shared<T>(res.get(), diff);
}
PackedPtr get_newptr(const SharedPtr &n) const {
  BasePtr *newval;
  unsigned count = 0;
  if (!n) {
    newval = nullptr;
  } else {
    newval = CountedDetail::get_counted_base(n);
    if (n.get() != CountedDetail::template get_shared_ptr<T>(newval)) {
      // This is an aliased sharedptr.  Make an un-aliased one
      // by wrapping in *another* shared_ptr.
      auto data = CountedDetail::template make_ptr<SharedPtr>(n);
      newval = CountedDetail::get_counted_base(data);
      count = ALIASED_PTR;
      // (add external must happen before data goes out of scope)
      add_external(newval);
    } else {
      add_external(newval);
```

```
    }
  }

  PackedPtr newptr;
  newptr.init(newval, count);

  return newptr;
}
PackedPtr get_newptr(SharedPtr &&n) const {
  BasePtr *newval;
  unsigned count = 0;
  if (!n) {
    newval = nullptr;
  } else {
    newval = CountedDetail::get_counted_base(n);
    if (n.get() != CountedDetail::template get_shared_ptr<T>(newval)) {
      // This is an aliased sharedptr.  Make an un-aliased one
      // by wrapping in *another* shared_ptr.
      auto data = CountedDetail::template make_ptr<SharedPtr>(std::move(n));
      newval = CountedDetail::get_counted_base(data);
      count = ALIASED_PTR;
      CountedDetail::release_ptr(data);
      add_external(newval, -1);
    } else {
      CountedDetail::release_ptr(n);
      add_external(newval, -1);
    }
  }

  PackedPtr newptr;
```

```
    newptr.init(newval, count);

    return newptr;
  }
  void init() {
    PackedPtr data;
    data.init();
    ptr_.store(data);
  }

  unsigned int get_local_count(const PackedPtr &p) const {
    return p.extra() & ~ALIASED_PTR;
  }

  // Check pointer equality considering wrapped aliased pointers.
  bool owners_eq(PackedPtr &p1, BasePtr *p2) {
    bool aliased1 = p1.extra() & ALIASED_PTR;
    if (aliased1) {
      auto p1a = CountedDetail::template get_shared_ptr_from_counted_base<T>(
          p1.get(), false);
      return CountedDetail::get_counted_base(p1a) == p2;
    }
    return p1.get() == p2;
  }

  SharedPtr get_shared_ptr(const PackedPtr &p, bool inc = true) const {
    bool aliased = p.extra() & ALIASED_PTR;

    auto res = CountedDetail::template get_shared_ptr_from_counted_base<T>(
        p.get(), inc);
```

```
  if (aliased) {
    auto aliasedp =
        CountedDetail::template get_shared_ptr_from_counted_base<SharedPtr>(
            p.get());
    res = *aliasedp;
  }
  return res;
}

/* Get a reference to the pointer, either from the local batch or
 * from the global count.
 *
 * return is the base ptr, and the previous local count, if it is
 * needed for compare_and_swap later.
 */
PackedPtr takeOwnedBase(std::memory_order order) const noexcept {
  PackedPtr local, newlocal;
  local = ptr_.load(std::memory_order_acquire);
  while (true) {
    if (!local.get()) {
      return local;
    }
    newlocal = local;
    if (get_local_count(newlocal) + 1 > EXTERNAL_OFFSET) {
      // spinlock in the rare case we have more than
      // EXTERNAL_OFFSET threads trying to access at once.
      std::this_thread::yield();
      // Force DeterministicSchedule to choose a different thread
      local = ptr_.load(std::memory_order_acquire);
    } else {
```

```
      newlocal.setExtra(newlocal.extra() + 1);
      assert(get_local_count(newlocal) > 0);
      if (ptr_.compare_exchange_weak(local, newlocal, order)) {
        break;
      }
    }
  }

  // Check if we need to push a batch from local -> global
  auto batchcount = EXTERNAL_OFFSET / 2;
  if (get_local_count(newlocal) > batchcount) {
    CountedDetail::inc_shared_count(newlocal.get(), batchcount);
    putOwnedBase(newlocal.get(), batchcount, order);
  }

  return newlocal;
}

void putOwnedBase(BasePtr *p, unsigned int count,
                  std::memory_order mo) const noexcept {
  PackedPtr local = ptr_.load(std::memory_order_acquire);
  while (true) {
    if (local.get() != p) {
      break;
    }
    auto newlocal = local;
    if (get_local_count(local) > count) {
      newlocal.setExtra(local.extra() - count);
    } else {
      // Otherwise it may be the same pointer, but someone else won
```

```
        // the compare_exchange below, local count was already made
        // global.  We decrement the global count directly instead of
        // the local one.
        break;
      }
      if (ptr_.compare_exchange_weak(local, newlocal, mo)) {
        return;
      }
    }

    CountedDetail::template release_shared<T>(p, count);
  }
};

}  // namespace std

// example
std::atomic<int32_t> cnt{0};
class A {
 public:
  A(int value) : value_(value) { ++cnt; }

  ~A() { --cnt; }

  int Value() const { return value_; }

 private:
  int value_ = 0;
};
```

```
int main() {
  constexpr uint32_t N = 1000000;
  constexpr uint32_t T = 4;

  std::atomic_shared_ptr<A> x;
  std::atomic_shared_ptr<A> y;
  std::vector<std::thread> threads;
  for (uint32_t t = 0; t < T; ++t) {
    threads.emplace_back([&] {
      for (uint32_t i = 0; i < N; ++i) {
        auto a = std::make_shared<A>(t * N + i);
        x.store(a);
        auto b = x.load();
        y.store(b);
      }
    });
  }

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

  return cnt.load() == 1 ? 0 : -1;
}
```

References

1. std::shared_ptr, *C++ Reference*
2. std::atomic(std::shared_ptr), *C++ Reference*

3. folly::atomic_shared_ptr, *Facebook*

uestc-lfs Sep 11, 2020

```
ReferenceCount *Ref() {
    ++cnt_; // 1
    return this;
  }

  void Deref() {
    if (--cnt_ == 0) { // 2
      delete this; // 3
    }
  }
```

这个不是线程安全的吧, 可能的执行循序有 2 -> 1 -> 3

SF-Zhou Sep 11, 2020 Owner

> **@uestc-lfs**
>
> ```
> ReferenceCount *Ref() {
> ```

```
    ReferenceCount* Ref() {
      ++cnt_; // 1
      return this;
    }

    void Deref() {
      if (--cnt_ == 0) { // 2
        delete this; // 3
      }
    }
```

> 这个不是线程安全的吧, 可能的执行循序有 2 -> 1 -> 3

你说的这种情况确实会导致线程不安全，但在 SharedPtr 单线程写或多线程读场景下并不会出现。

↑ 1

0 replies

uestc-lfs Sep 11, 2020

哦哦，是 获取 Outlook for Android<https://aka.ms/ghei36>

...

↑ 1

0 replies

Write Preview Aa

Sign in to comment

Sign in to comment

Styling with Markdown is supported

Sign in with GitHub