Doc. No.: N3712
Date: 2013-08-30
Author: Arch D. Robison ⟨arch.robison@intel.com⟩

Policy-Based Design for Safe Destruction in Concurrent Containers

Contents

1 Introduction

Writing containers that support concurrently access/modification by multiple threads incurs a special problem: when is it safe to delete an object? The problem is surprisingly tricky when lock-free operations are involved. Numerous general means have been proposed, such as Hazard Pointers [1], Pass the

Buck [2], Epoch Based Reclamation [3], Read Copy Update (RCU) [4], Differential Reference Counting (DRC) [5], Reference Counting [6], Garbage Collection (GC), and Transactional Memory.[1] Each means has its own strengths and weaknesses, and no one approach is ideal in all contexts.

This paper sketches a way of abstracting these means so that concurrent container implementations could be parameterized over these means, thus allowing users to select the means best for their context. Though few users will want to change a container at whim, say from Hazard Pointers to RCU, different applications will have different requirements. For example, though RCU is popular for kernel writing, Hazard Pointers may be preferable for user-space code. Hence abstracting over the particular means could enable wider use of common concurrent container code.

What would be standardized is a conceptual *reclaimer* interface along with a concrete implementation. Concurrent containers (see N3732) would be parameterized over a *reclaimer* argument. This arrangement would be analogous to current practice with allocators: they have a conceptual *allocator* interface, a concrete implementation `std::allocator`, and are used as parameters to container templates.

This paper presents a sketch, not a complete proposal. It is intended to elicit discussion from experts as to whether the abstraction is viable. As such, this paper presents a fairly minimal interface that omits knobs for weak memory consistency and and extensive overloading of operators, so that experts can focus on a core question: Is the abstraction sufficient to support various means for deferred object deletion?

1.1 Weak Guarantee

The basic idea is to define some abstractions that support the weakest common guarantee that the various means can support. There are two fundamental abstractions:

- A *concurrent_ptr* acts like a pointer that supports atomic operations.
- A *guard_ptr* is an object that can atomically take a snapshot of the value of a *concurrent_ptr* and **if** the target has not yet been deleted, guarantee that the target will not be deleted while the *guard_ptr* holds a pointer to it.

[1] The recent "Drop the Anchor" [7] approach is omitted from the list since it seems difficult to apply generically.

It is important to note that this guarantee is much weaker than classic garbage collection: It is only *guard_ptr* references that protect against deletion.

In effect, a *concurrent_ptr* is a "weak" pointer and a *guard_ptr* is a "shared ownership" pointer. The key semantic differences from `std::weak_ptr` and `std::shared_ptr` are that

- *concurrent_ptr* and *guard_ptr* are abstract interfaces (a.k.a. "concepts"), not concrete interfaces.
- They support a snapshot operation that is an *atomic* analog of method `std::weak_ptr::lock()`.
- A `std::weak_ptr` can indicate whether it has "expired" — that is its target was deleted. A *concurrent_ptr* gives no such indication even if, as it can in some implementations, point to freed memory.

1.2 Marked Pointers

Many lock-free algorithms rely on marking pointers with one or more low-order bits borrowed from the pointer value itself. See references [8, 1, 9] for examples. Because of the way the bit borrowing interacts with some of the deferred deletion mechanisms, the notion of a marked pointer is part of the interface.

The abstractions allow more than one mark bit per pointer, as opposed to Java's `AtomicMarkableReference` that supports only a single bit. If the number of mark bits exceeds practicality on the target machine, the implementation can fall back on using locks, much the way existing `std::atomic<T>` can. Like the latter, a marked pointer has a method `is_lock_free()` that indicates whether operations on it are lock free.

2 Interface

Each concurrent container would take a *reclaimer* argument in addition to an allocator argument. Similar to the allocator argument, there would be a default reclaimer, for example `std::default_reclaimer`.

A reclaimer type `R` would define two abstractions necessary for safe destruction and deletion.

- `R::concurrent_ptr<T,D>` : Acts like an atomic markable pointer to objects of type `T` with deleter of type `D`. It supports atomic operations such as `load`, `store`, and `compare_exchange_weak`. Class `T` must be derived from `enable_concurrent_ptr<T,N>`.

- `R::enable_concurrent_ptr<T,N>` : Mandatory base class for targets of `concurent_ptr`. `T` is the derived class. `N` is the number of mark bits supported, which defaults to zero.

The intent of `enable_concurrent_ptr<T,N>` is to provide implementers of reclaimers with two things:

- A way to force the alignment of targets, which is a common way to provide mark bits in the pointers.

- A place to embed reclaimer state, such as reference counts, in the user's objects.

Class `concurrent_ptr<T,D>` provides two auxiliary types for working with `concurrent_ptr` objects.

- `concurrent_ptr<T,D>::marked_ptr` : Acts like a pointer, but has `N` mark bits, where `N` is specified by the base class `enable_concurrent_ptr<T,N>` of `T`.

- `concurrent_ptr<T,D>::guard_ptr` : Similar to a `marked_ptr`, but has shared ownership of its target *if* the target has not been deleted.

The key operation is `acquire`, which takes a snapshot with the weak guarantee. Here is an example of the syntax:

```
extern concurrent_ptr p;
extern guard_ptr g;
g.acquire(p);
```

In wait-free algorithms, `acquire` may be problematic when implemented with hazard pointers or pass the buck, because it may have to loop indefinitely to get a good snapshot. Thus a fancier form is also provided that enables quitting early if the value in `p` does not match a provided value `m`.

```
extern concurrent_ptr p;
extern guard_ptr g;
extern marked_ptr m;
bool b = g.acquire_if_equal(p,m);
```

If the pointer in `p` does not match `m`, `b==false`. Otherwise `b==true` and the net effect is the same as for a plain `acquire`, and afterwards `g.get()==m`. Listing 9 in Section 5.2 will show it in use.

Releasing a guard follows the standard smart pointer interface. The operation `g.reset` releases ownership and sets `g` to `nullptr`. The destructor of `guard_ptr` implicitly calls `reset`.

2.1 Possible Concrete Implementations

Some possible concrete implementations:

- Hazard Pointers: `guard_ptr` is a hazard pointer. The operation `acquire(p)` repeatedly copies `p` into a hazard pointer until the hazard pointer and `p` match. The operation `g.reset()` sets the hazard pointer to null.
- Pass the Buck: `guard_ptr` is a “guard”. Operation of `acquire(p)` is similar to hazard pointers.
- RCU: The operation `acquire` indicates start of a read critical section followed by a *rcu_dereference* operation. The operation `g.reset()` indicates the end of a read critical section.
- DRC: A `concurrent_ptr` is a “strong pointer” and a `guard_ptr` is a “basic pointer”.
- GC: `guard_ptr` is just a traceable pointer.

Note that RCU, DRC, and GC provide much stronger guarantees than our “weak guarantee”. Programmers should take care to not accidentally rely on the strong guarantees.

3 Alternatives

In theory, `shared_ptr` and `weak_ptr` could be used, but their higher degree of safety adds significant overhead. In particular, atomic operations on these are quite expensive, requiring expensive locking protocols and solve the problem poorly. (Michael and Scott [6] discuss why reference counting is problematic.) In contrast, this proposal enables lightweight mechanisms such as Hazard Pointers or RCU to be employed.

The sketched proposal separates *allocator* and *reclaimer* policy arguments. An alternative is for concurrent containers to take a single policy argument that combines both into a "concurrent allocator". The combination would eliminate the need for the type `enable_concurrent_ptr` since the concurrent allocator could do the equivalent alignment adjustments when allocating an object.

4 Concrete Interface

This section shows a basic concrete interface, sufficient for the examples in Section 5. Class templates `enable_concurrent_ptr` and `concurrent_ptr` are assumed to be members of a concrete reclaimer class, for example `rcu_reclaimer` or `hazard_pointer_reclaimer`.

4.1 enable_concurrent_ptr

```
template<class T, size_t MarkSize=0>
class enable_concurrent_ptr {
protected:
   enable_concurrent_ptr() noexcept;
   enable_concurrent_ptr(const enable_concurrent_ptr&) noexcept;
   enable_concurrent_ptr& operator=(const enable_concurrent_ptr&)
      noexcept;
   ~enable_concurrent_ptr() noexcept;
};
```

Listing 1: Concrete interface for enable_concurrent_ptr

4.2 concurrent_ptr

```
//! T must be derived from enable_concurrent_ptr<T>. D is a deleter.
template<class T, class D=std::default_delete<T> >
class concurrent_ptr {
public:

   // See Section 4.3
   typedef implementation-defined marked_ptr;

   // See Section 4.4
```

```
    typedef implementation-defined guard_ptr;

    // Constuct concurrent_ptr
    concurrent_ptr(const marked_ptr& p=marked_ptr());

    // Atomic load that does not guard target from being reclaimed.
    marked_ptr load() const;

    // Atomic store.
    void store(const marked_ptr& src);

    // Shorthand for store(src.get())
    void store(const guard_ptr& src);

    // Atomic exchange.
    void swap(guard_ptr& q);

    // Compare-and-swap.
    bool compare_exchange_weak(marked_ptr& expected, marked_ptr
        desired);

    // Compare-and-swap.
    bool compare_exchange_weak(guard_ptr& expected, marked_ptr
        desired);

    // Shorthand for compare_exchange_weak(src.get())
    bool compare_exchange_weak(marked_ptr& expected, guard_ptr&
        desired);

    // Shorthand for compare_exchange_weak(src.get())
    bool compare_exchange_weak(guard_ptr& expected, guard_ptr&
        desired);
};
```

Listing 2: Concrete interface for concurrent_ptr

The interface for atomic exchange departs somewhat from the rest of the standard library to avoid introducing the need for a third `guard_ptr` object, whose construction might be incur unnecessary overhead.

The several flavors of `compare_exchange_weak` exist to support various cases where one or both arguments do not need guard protection.

4.3 marked_ptr

The implementation-defined type `concurrent_ptr<T,D>::mark_ptr` should behave as if it were defined as follows.

```
template<class T, class D>
class concurrent_ptr<T,D>::marked_ptr {
public:
    // Construct a marked_ptr
    marked_ptr(T* p=nullptr, uintptr_t mark=0) noexcept;

    // Copy constructor
    marked_ptr(const marked_ptr&) noexcept;

    // Copy underlying marked pointer of a guard_ptr.
    marked_ptr(const guard_ptr& g) noexcept;

    // Destructor
    ~marked_ptr() noexcept;

    // Assignment
    marked_ptr& operator=(const marked_ptr&) noexcept;

    // Set to nullptr
    void reset() noexcept;

    // Get mark bits
    uintptr_t mark() const noexcept;

    // Get underlying pointer (with mark bits stripped off).
    T* get() const noexcept;

    // True iff get()==nullptr && mark()==0
    explicit operator bool() const noexcept;

    // Get pointer with mark bits stripped off.
    T* operator->() const noexcept;

    // Get reference to target of pointer.
    T& operator*() const noexcept;

    // True iff operations are lock-free
    static bool is_lock_free() noexcept;
};
```

Listing 3: Concrete interface for marked_ptr

Additionally, there is a requirement that `operator==` and `operator!=` operate on marked pointers, and include mark bits in their comparisons.

4.4 guard_ptr

The implementation-defined type `concurrent_ptr<T,D>::guard_ptr` should behave as if it were defined as follows.

```
template<class T, class D>
class concurrent_ptr<T,D>::guard_ptr {
public:
    // Guard a marked_ptr.
    guard_ptr(const marked_ptr& p=marked_ptr());

    // Copy constructor
    explicit guard_ptr(const guard_ptr& p);

    // Copy assignment
    guard_ptr& operator=(const guard_ptr& p);

    // Destructor
    ~guard_ptr();

    // Atomically take snapshot of p, and *if* it points to unreclaimed
        object, acquire shared ownership of it.
    void acquire(concurrent_ptr& p);

    // Like acquire, but quit early if a snapshot != expected.
    bool acquire_if_equal(concurrent_ptr& p, const marked_ptr&
        expected);

    // Release ownership. Postcondition: get()==nullptr.
    void reset() noexcept;

    // Get underlying pointer
    T* get() const noexcept;

    // Get mark bits
    uintptr_t mark() const noexcept;

    // True if snapshot is nullptr with mark bits set to zero.
    explicit operator bool() const noexcept;

    // Get pointer with mark bits stripped off. Undefined if target has
        been reclaimed.
    T* operator->() const noexcept;

    // Get reference to target of pointer. Undefined if target has been
        reclaimed.
    T& operator*() const noexcept;

    // Swap two guards
    void swap(guard_ptr& g) noexcept;

    // Reset. Deleter d will be applied some time after all owners release
        their ownership.
    void reclaim(D d=D()) noexcept;

    // True iff operations are lock-free
    static bool is_lock_free() noexcept;
};
```

Listing 4: Concrete interface for guard_ptr

5 Examples

5.1 Classic Lock-Free Stack

The classic lock-free stack is a linked list. Listing 5 declares the data structures.

```
struct node;
```

```
typedef typename Reclaimer::template concurrent_ptr<node>
    concurrent_ptr;

// A node in a linked list
struct node: Reclaimer::template enable_concurrent_ptr<node> {
    const Value value;
    node* next;
    node(Value k) : value(k) {}
};

// Root of the linked list
concurrent_ptr root;

typedef typename concurrent_ptr::marked_ptr marked_ptr;
typedef typename concurrent_ptr::guard_ptr guard_ptr;
```

Listing 5: Data declarations for a lock-free stack

A subtle point is that `next` is declared as a plain pointer, not a `concurrent_ptr` or a `atomic<node*>`, though it could be. The reason is that its value is written only before it is published to other threads or after reclamation has determined it is accessible to only one thread.

Listing 6 shows how an item is pushed onto the list. Each `mark_ptr` here has zero mark bits. No `guard_ptr` is required since no other thread can possibly delete the newly created node until it is published in the list.

```
void push(Value value) {
    marked_ptr n = new node(value);
    marked_ptr m;
    do {
        m = root.load();
        n->next = m.get();
    } while(root.compare_exchange_weak(m,n));
}
```

Listing 6: Pushing onto a lock-free stack

Listing 7 shows how to pop an item from the stack. The `guard_ptr` is essential for preventing an ABA problem that could corrupt the list representation.

```
bool try_pop(Value& v) {
    guard_ptr g;
    do {
        // Get guarded snapshot of root.
        g.acquire(root);
        if(!g)
            // Stack is empty.
            return false;
    } while(!root.compare_exchange_weak(g, g->next));
    // Successfully popped item from stack.
    v = g->value;
    g.reclaim();
    return true;
}
```

Listing 7: Popping from a lock-free stack

5.2 Lock-Free List-Based Set

This example is adapted from Figure 9 of reference[1]. Listing 8 declares the data structures.

```
    struct node;
    typedef typename Reclaimer::template concurrent_ptr<node>
        concurrent_ptr;
    typedef Key key;
    struct node: Reclaimer::template enable_concurrent_ptr<node,1> {
        const Key key;
        concurrent_ptr next;
        node(Key k) : key(k) {}
    };
    typedef typename concurrent_ptr::guard_ptr guard_ptr;
    typedef typename concurrent_ptr::marked_ptr marked_ptr;
    concurrent_ptr head;
```

Listing 8: Data declarations for a lock-Free list-based set

Listing 9 shows routine `find`, which is the workhorse for the lock-free list. It searches the list for a given key or suitable insertion point for the key.

```
bool find(Key key, concurrent_ptr*& prev, guard_ptr& cur,
    marked_ptr& next, guard_ptr& save) {
retry:
    prev = &head;
    next = prev->load();
    for(;;) {
        if(!cur.acquire_if_equal(*prev, next))
            goto retry;
        if(!cur)
            return false;
        next = cur->next.load();
        if(next.mark()!=0) {
            // Node *cur is marked for deletion.
            next = marked_ptr(next.get(),0);
            // Try to splice out node
            if(!prev->compare_exchange_weak(cur, next))
                goto retry;
            cur.reclaim();
        } else {
            Key ckey = cur->key;
            if(prev->load()!=marked_ptr(cur))
                // *cur might be cut from list.
                goto retry;
            if(ckey>=key)
                return ckey==key;
            prev = &cur->next;
            cur.swap(save);
        }
    }
}
```

Listing 9: Find operation on a lock-free List

If it returns true, then the following properties hold:

- `prev` points to a live (not in reclaimed memory) `concurrent_ptr`.
- `cur` points to a live node with a key greater or equal to the search key.
- `save` is guarding the node containing `*prev`, unless `prev==&head`, in which case no guard is necessary.
- While `cur->next` and `next` match, the node pointed to by `cur` is still part of the list.
- While `prev->load()` and `cur` match, it is safe to insert a new node there. See Listing 10 for details.

Listing 10 uses `find` to insert an item.

```
bool insert(Key key) {
    node* n = new node(key);
    concurrent_ptr* prev;
    guard_ptr cur, save;
    marked_ptr next;
    do {
        if(find(key,prev,cur,next,save)) {
            delete n;
            return false;
        }
        // Try to install new node
        n->next.store(cur.get());
    } while(!prev->compare_exchange_weak(cur,n));
    return true;
}
```

Listing 10: Insert operation on a lock-Free list

Listing 11 uses `find` to erase an item.

```
bool erase(Key key) {
    concurrent_ptr* prev;
    guard_ptr cur, save;
    marked_ptr next;
    // Find node in list with matching key and mark it for erasure.
    do {
        if(!find(key,prev,cur,next,save))
            // No such node in the list
            return false;
    } while(!cur->next.compare_exchange_weak(next,marked_ptr(next.
        get(),1)));
    // Try to splice out node
    if(prev->compare_exchange_weak(cur,next))
        cur.reclaim();
    else
        // Another thread interfered. Rewalk list to ensure reclamation of
             marked node before returning.
        find(key,prev,cur,next,save);
    return true;
18 }
```

Listing 11: Erase operation on a lock-Free list

The `find` on line 16 assures progress guarantees [8].

6 Acknowledgments

Paul McKenney raised the issue of flexibly supporting different mechanisms for deferred deletion, and provided feedback on an earlier draft. Maged Michael and Paul raised issues of lock-free versus wait-free, which motivated extending `acquire` to `acquire_if_equal`. Discussions with them also reminded me of the ubiquity of mark bits in lock-free algorithms. Dmitry Vyukov's "strong pointer" and "basic pointer" interface inspired the basic interface. Artur Laksberg critiqued an earlier draft, and Anton Potapov corrected a later draft.

References

[1] Maged M. Michael. Hazard pointers: Safe memory reclamation for lock-free objects. *IEEE Trans. Parallel Distrib. Syst.*, 15(6):491–504, June 2004.

[2] Maurice Herlihy, Victor Luchangco, and Mark Moir. The repeat offender problem: A mechanism for supporting dynamic-sized, lock-free data structures. In *Proceedings of the 16th International Conference on Distributed Computing*, DISC '02, pages 339–353, London, UK, UK, 2002. Springer-Verlag.

[3] Keir Fraser. Practical lock-freedom. Technical Report UCAM-CL-TR-579, University of Cambridge, Computer Laboratory, February 2004.

[4] Mathieu Desnoyers, Paul E. McKenney, Alan S. Stern, Michel R. Dagenais, and Jonathan Walpole. User-level implementations of read-copy update. *IEEE Trans. Parallel Distrib. Syst.*, 23(2):375–382, February 2012.

[5] Differential reference counting. `http://www.1024cores.net/home/lock-free-algorithms/object-life-time-management/differential-reference-counting`. Accessed 2013-08-07.

[6] Maged M. Michael and Michael L. Scott. Correction of a memory management method for lock-free data structures. Technical report, Rochester, NY, USA, 1995.

[7] Anastasia Braginsky, Alex Kogan, and Erez Petrank. Drop the anchor: lightweight memory management for non-blocking data structures. In *Proceedings of the 25th ACM symposium on Parallelism in algorithms and architectures*, SPAA '13, pages 33–42, New York, NY, USA, 2013. ACM.

[8] Timothy L. Harris. A pragmatic implementation of non-blocking linked-lists. In *Proceedings of the 15th International Conference on Distributed Computing*, DISC '01, pages 300–314, London, UK, UK, 2001. Springer-Verlag.

[9] Hans-J. Boehm. An almost non-blocking stack. In *Proceedings of the twenty-third annual ACM symposium on Principles of distributed computing*, PODC '04, pages 40–49, New York, NY, USA, 2004. ACM.