Lock-Free Data Structures

Andrei Alexandrescu

December 17, 2007

After Generic⟨Programming⟩ has skipped one instance (it's quite naïve, I know, to think that grad school asks for anything less than 100% of one's time), there has been an embarrassment of riches as far as topic candidates for this article go. One topic candidate was a discussion of constructors, in particular forwarding constructors, handling exceptions, and two-stage object construction. One other topic candidate—and another glimpse into the Yaslander technology [2]—was creating containers (such as lists, vectors, or maps) of incomplete types, something that is possible with the help of an interesting set of tricks, but not guaranteed by the standard containers.

While both candidates were interesting, they couldn't stand a chance against *lock-free data structures*, which are all the rage in the multithreaded programming community. At this year's PLDI conference (`http://www.cs.umd.edu/~pugh/pldi04/`), Michael Maged presented the world's first lock-free memory allocator [7], which surpasses at many tests other more complex, carefully-designed lock-based allocators. This is the most recent of many lock-free data structures and algorithms that have appeared in the recent past... but let's start from the beginning.

1 What do you mean, "lock-*free*?"

That's exactly what I would have asked only a while ago. As the bona-fide mainstream multithreaded programmer that I was, lock-based multithreaded algorithms were quite familiar to me. In classic lock-based programming, whenever you need to share some data, you need to serialize access to it. The operations that change data must appear as atomic such that no other thread intervenes to spoil your data's invariant. Even a simple operation such as `++count_`, where `count_` is an integral type, must be locked as "`++`" is really a three steps (read, modify, write) operation that isn't necessarily atomic.

In short, with lock-based multithreaded programming, you need to make sure that any operation on shared data that is susceptible to race conditions is made atomic by locking and unlocking a mutex. On the bright side, as long as the mutex is locked, you can perform just about any operation, in confidence that no other thread will trump on your shared state.

It is exactly this "arbitrary"-ness of what you can do while a mutex is locked that's also problematic. You could, for example, read the keyboard or perform some slow I/O operation, which means that you delay any other threads waiting for the same mutex. Worse, you could decide you want access to some *other* piece of shared data and attempt to lock its mutex. If another thread has already locked that last mutex and wants access to the first mutex that your threads already holds, both processes hang faster than you can say "deadlock."

Enter lock-free programming. In lock-free programming, you can't do just about anything atomically. There is only a precious small set of things that you can do atomically, limitation that makes lock-free programming way harder. (In fact, there must be around half a dozen of lock-free programming experts around the world, and yours truly is not among them. With luck, however, this article will provide you with the basic tools, references, and enthusiasm to help you become one.) The reward of such a scarce framework is that you can provide much

better guarantees about thread progress and the interaction between threads. But what's that "small set of things" that you can do atomically in lock-free programming? In fact, what would be the minimal set of atomic primitives that would allow implementing *any* lock-free algorithm—if there's such a set?

If you believe that's a fundamental enough question to award a prize to the answerer, so did others. In 2003, Maurice Herlihy was awarded the Edsger W. Dijkstra Prize in Distributed Computing for his seminal 1991 paper "Wait-Free Synchronization" (see `http://www.podc.org/dijkstra/2003.html`, which includes a link to the paper, too). In his *tour-de-force* paper, Herlihy proves which primitives are good and which are bad for building lock-free data structures. That brought some seemingly hot hardware architectures to instant obsolescence, while clarifying what synchronization primitives should be implemented in future hardware.

For example, Herlihy's paper gave impossiblity results, showing that atomic operations such as test-and-set, swap, fetch-and-add, or even atomic queues (!) are insufficient for properly synchronizing more than two threads. (That's quite surprising because queues with atomic `push` and `pop` operations would seem to provide quite a powerful abstraction.) On the bright side, Herlihy also gave universality results, proving that some very simple constructs are enough for implementing any lock-free algorithm for any number of threads.

The simplest and most popular universal primitive, and the one that we'll use throughout, is the compare-and-swap (`CAS`) operation:

```
template <class T>
bool CAS(T* addr, T exp, T val) {
  if (*addr == exp) {
    *addr = val;
    return true;
  }
  return false;
}
```

`CAS` compares the content of a memory address with an expected value, and if the comparison succeeds, replaces the content with a new value. The entire procedure is atomic.

Many modern processors implement `CAS` or equivalent primitives for different bit lengths (reason for which we've made it a template, assuming an implementation uses metaprogramming to restrict possible `T`s). As a rule of thumb, the more bits a `CAS` can compare-and-swap atomically, the easier it is to implement lock-free data structures with it. Most of today's 32-bit processors implement 64-bit `CAS`; for example, Intel's assembler calls it `CMPXCHG8` (you gotta love those assembler mnemonics).

2 A Word of Caution

Usually a C++ article is accompanied by C++ code snippets and examples. Ideally, that code is standard C++, and Generic⟨Programming⟩ strives to live up to that ideal.

When writing about multithreaded code, giving standard C++ code samples is simply impossible. Threads don't exist in standard C++, and you can't code something that doesn't exist. Therefore, this article's code is really "pseudocode" and not meant as standard C++ code meant for portable compilation.

Take memory barriers for example. Real code would need to be either assembly language translations of the algorithms described herein, or at least sprinkle C++ code with some so-called "memory-barriers"—processor-dependent magic that forces proper ordering of memory reads and writes. This article doesn't want to spread itself too thin by explaining memory barriers in addition to lock-free data structures. If you are interested, you may want to refer to Butenhof's excellent book [3] or to a short introduction [6]. For the purposes of this article, we can just assume that the compiler doesn't do funky optimizations (such as eliminating some "redundant" variable reads, a valid optimization under a single-thread assumption). Technically, that's called a "sequentially consistent" model in which reads and writes are performed in the exact order in which the source code does them.

3 Wait-Free and Lock-Free versus Locked

To clarify terms, let's provide a few definitions. A "wait-free" procedure is one that can complete in a finite number of steps, regardless of the relative speeds of other threads.

A "lock-free" procedure guarantees progress of at least one of the threads executing the procedure. That means, some threads can be delayed arbitrarily, but it is guaranteed that at least one thread of all makes progress at each step. Statistically, in a lock-free procedure, all threads will make progress.

Lock-based programs can't provide any of the above guarantees. If any thread is delayed while holding a lock to a mutex, progress cannot be made by threads that wait for the same mutex; and in the general case, locked algorithms are prey to deadlock—each waits for a mutex locked by the other—and livelock—each tries to dodge the other's locking behavior, just like two dudes in the hallway trying to go past one another but end up doing that social dance of swinging left and right in synchronicity. We humans are pretty good at ending that with a laugh; processors, however, often enjoy doing it til rebooting sets them apart.

Wait-free and lock-free algorithms enjoy more advantages derived from their definitions:

- *Thread-killing immunity:* any thread forcefully killed in the system won't delay other threads.
- *Signal immunity:* Traditionally, routines such as `malloc` can't be called during signals or asynchronous interrupts. This is because the interrupt might occur right while the routine holds some lock. With lock-free routines, there's no such problem anymore: threads can freely interleave execution.
- *Priority inversion immunity:* Priority inversion occurs when a low-priority thread holds a lock to a mutex needed by a high-priority thread, case in which CPU resources must be traded for locking privileges. This is tricky and must be provided by the OS kernel. Wait-free and lock-free algorithms are immune to such problems.

Now that introductions have been made, let's analyze a lock-free implementation of a small design.

4 A Lock-Free WRRM Map

Column writing offers the perk of defining acronyms, so let's define WRRM ("Write Rarely Read Many") maps as maps that are read a lot more times than they are mutated. Examples include object factories [1], many instances of the Observer design pattern [5], mappings of currency names to exchange rates that are looked up many, many times but are updated only by a comparatively slow stream, and various other look-up tables.

WRRM maps can be implemented via `std::map` or the post-standard `hash_map`, but as Modern C++ Design argues, `assoc_vector` (a sorted vector or pairs) is a good candidate for WRRM maps because it trades update speed for lookup speed. Whatever structure is used, our lock-free aspect is orthogonal on it; we'll just call our back-end `Map<Key, Value>`. Also, we don't care about the iteration aspect that maps provide; we treat the maps as tables that provide means to lookup a key or update a key-value pair.

To recap how a "lockful" implementation would look like, we'd combine a `Map` object with a `Mutex` object like so:

```
// A lockful implementation of WRRMMap
template <class K, class V>
class WRRMMap {
  Mutex mtx_;
  Map<K, V> map_;
public:
  V Lookup(const K& k) {
    Lock lock(mtx_);
    return map_[k];
  }
  void Update(const K& k,
      const V& v) {
    Lock lock(mtx_);
    map_.insert(make_pair(k, v));
  }
};
```

Rock-solid—but at a cost. Every lookup locks and unlocks the `Mutex`, although (1) parallel lookups don't need to interlock, and (2) by the spec, `Update` is much less often called than `Lookup`. Ouch! Let's now try to provide a better `WRRMMap` implementation.

5 Garbage Collector, Where Are Thou?

Our first shot at implementing a lock-free `WRRMMap` rests on the following idea:

- Reads have no locking at all.
- Updates make a copy of the entire map, update the copy, then try to `CAS` it with the old map. While the `CAS` operation does not succeed, the copy/update/`CAS` process is tried again in a loop.
- Because `CAS` is limited in how many bytes it can swap, we store the `Map` as a pointer and not as a direct member of `WRRMMap`.

```
// 1st lock-free implementation of WRRMMap
// Works only if you have GC
template <class K, class V>
class WRRMMap {
  Map<K, V>* pMap_;
public:
  V Lookup(const K& k) {
    // Look, ma, no lock
    return (*pMap_)[k];
  }
  void Update(const K& k,
      const V& v) {
    Map<K, V>* pNew = 0;
    do {
      Map<K, V>* pOld = pMap_;
      delete pNew;
      pNew = new Map<K, V>(*pOld);
      pNew->insert(make_pair(k, v));
    } while (!CAS(&pMap_, pOld, pNew));
    // DON'T delete pMap_;
  }
};
```

It works! In a loop, the `Update` routine makes a full-blown copy of the map, adds the new entry to it, and then attempts to swap the pointers. It is important to do `CAS` and not a simple assignment; otherwise, the following sequence of events could corrupt our map:

- Thread A copies the map;
- Thread B copies the map as well and adds an entry;
- Thread A adds some other entry;
- Thread A replaces the map with its version of the map—a version that does not contain whatever B added.

With `CAS`, things work pretty neatly because each thread says something like, "assuming the map hasn't changed since I last looked at it, copy it. Otherwise, I'll start all over again."

Note that this makes `Update` lock-free but not wait-free by our definitions above. If many threads call `Update` concurrently, any particular thread might loop indefinitely, but at all times some thread will be guaranteed to update the structure successfully, thus global progress is being made at each step. Luckily, `Lookup` is wait-free.

In a garbage-collected environment, we'd be done, and this article would end in an upbeat note. Without garbage collection, however, there is much, much pain to come (for one thing, you have to read more of my writing). This is because we cannot simply dispose the old `pMap_` willy-nilly; what if, just as we are trying to **`delete`** it, some many other threads are frantically looking for things inside `pMap_` via the `Lookup` function? You see, a garbage collector would have access to all threads' data and private stacks; it would have a good perspective on when the unused `pMap_` pointers aren't perused anymore, and would nicely scavenge them. Without a garbage collector, things get harder. Much harder, actually, and it turns out that deterministic memory freeing is quite a fundamental problem in lock-free data structures.

6 Write-Locked WRRM Maps

To understand the viciousness of our adversary, it is instructive to first try a classic reference-counting implementation and see where it fails. So, let's think of associating a reference count with the pointer to map, and have `WRRMMap` store a pointer to the thusly-formed structure:

```
template <class K, class V>
class WRRMMap {
  typedef std::pair<Map<K, V>*,
    unsigned> Data;
  Data* pData_;
  ...
};
```

Sweet. Now, `Lookup` increments `pData_->second`, searches through the map all it wants, then decrements `pData_->second`. When the reference count hits zero, `pData_->first` can be **`delete`**d, and then so can `pData_` itself. Sounds foolproof, except...

Except it's "foolful" (or whatever the antonym to "foolproof" is). Imagine that right at the time some thread notices the refcount is zero and proceeds on deleting `pData_`, another thread... no, better: a *bazillion* threads have just loaded the moribund `pData_` and are about to read through it! No matter how smart a scheme is, it will hit this fundamental catch-22: to read the pointer to the data, one needs to increment a reference count; but the counter must be part of the data itself, so it can't be read *without accessing the pointer first*. It's like an electric fence that has the turn-off button up on top of it: to safely climb the fence you need to disable it first, but to disable it you need to climb it.

So let's think of other ways to **`delete`** the old map properly. One solution would be to wait, then **`delete`**. You see, the old `pMap_` objects will be looked up by less and less threads as processor eons (milliseconds) go by; this is because new lookups will use the new maps; as soon that the lookups that were active between the `CAS` finish, the `pMap_` is ready to go to Hades. Therefore, a solution would be to queue up old `pMap_` values to some "boa serpent" thread that, in a loop, sleeps for, say, 200 milliseconds, then wakes up and **`delete`**s the least recent map, to go back to sleep for digestion.

This is not a theoretically safe solution (although it practically could well be within bounds). One nasty thing is that if, for whatever reason, a lookup thread is delayed a lot, the boa serpent thread can **`delete`** the map under that thread's feet. This could be solved by always assignining the boa serpent thread a lower priority than any other's, but as a whole the solution has a stench with it that is hard to remove. If you agree with me that it's hard to defend this technique with a straight face, let's move on.

Other solutions [4] rely on an extended `DCAS` atomic instruction, which is able to compare-and-swap two non-contiguous words in memory:

```
template <class T1, class T2>
bool DCAS(T1* p1, T2* p2,
    T1 e1, T2 e2,
    T1 v1, T2 v2) {
  if (*p1 == e1 && *p2 == e2) {
    *p1 = v1; *p2 = v2;
    return true;
  }
  return false;
}
```

Naturally, the two locations would be the pointer and the reference count itself. `DCAS` has been implemented (very inefficiently) by the Motorola 68040 processors, but not by other processors. Because of that, `DCAS`-based solutions are considered of primarily theoretical value.

The first shot we'll take at a solution with deterministic destruction is to rely on the less-demanding `CAS2`. As mentioned before, many 32-bit machines implement a 64-bit `CAS`, often dubbed as `CAS2`. (Because it only operates on contiguous words, `CAS2` is obviously less powerful than `DCAS`.) For starters, we'll store the reference count next to the pointer that it guards:

```
template <class K, class V>
class WRRMMap {
  typedef std::pair<Map<K, V>*,
    unsigned> Data;
  Data data_;
  ...
```

```
};
```

(Notice that this time we store the count next to the pointer that it protects, and this rids us of the catch-22 problem mentioned earlier. We'll see the cost of this setup in a minute.)

Then, we modify `Lookup` to increment the reference count before accessing the map, and decrement it after. In the following code snippets, we will ignore exception safety issues (which can be taken care of with standard techniques) for the sake of brevity.

```
V Lookup(const K& k) {
  Data old;
  Data fresh;
  do {
    old = data_;
    fresh = old;
    ++fresh.second;
  } while (!CAS(&data_, old, fresh));
  V temp = (*fresh.first)[k];
  do {
    old = data_;
    fresh = old;
    --fresh.second;
  } while (!CAS(&data_, old, fresh));
  return temp;
}
```

Finally, `Update` replaces the map with a new one—but only in the window of opportunity when the reference count is 1.

```
void Update(const K& k,
    const V& v) {
  Data old;
  Data fresh;
  old.second = 1;
  fresh.first = 0;
  fresh.second = 1;
  Map<K, V>* last = 0;
  do {
    old.first = data_.first;
    if (last != old.first) {
      delete fresh.first;
      fresh.first =
        new Map<K, V>(old.first);
      fresh.first->insert(
        make_pair(k, v));
      last = old.first;
    }
  } while (!CAS(&data_, old, fresh));
  delete old.first; // whew
}
```

Here's how `Update` works. We have the by-now-familiar `old` and `fresh` variables. But this time `old.second` (the count) is never assigned from `data_.second`; it is always 1. That means, `Update` will loop until it has a window of opportunity of replacing a pointer with a counter of 1, with another pointer having a counter of 1. In plain English, the loop says "I'll replace the old map with a new, updated one, and I'll be on lookout for any other updates of the map, but I'll only do the replacement when the reference count of the existing map is one." The variable `last` and its associated code are only one optimization: avoid rebuilding the map over and over again if the old map hasn't been replaced (only the count).

Neat, huh? Not that much. `Update` is now locked: it will need to wait for all `Lookup`s to finish before it has a chance to update the map. Gone with the wind are all the nice properties of lock-free data structures. In particular, it is very easy to starve `Update` to death: just look up the map at a high-enough rate—and the reference count will never go down to one. So what we really have so far is not a WRRM (Write-Rarely-Read-Many) map, but a WRRMBNTM (Write-Rarely-Read-Many-But-Not-Too-Many) one instead.

7 Conclusions

Lock-free data structures are very promising. They exhibit good properties with regards to thread killing, priority inversion, and signal safety. They never deadlock or livelock. In tests, recent lock-free data structures surpass their locked counterparts by a large margin.

However, lock-free programming is tricky especially with regards to memory deallocation. A garbage collected environment is a plus because it has the means to stop and inspect all threads, but if

you want deterministic destruction, you need special support from the hardware or the memory allocator.

The next installment of Generic⟨Programming⟩ will look into ways to optimize `WRRMMap` such that it stays lock-free while performing deterministic destruction. And if this installment's garbage-collected and WRRMBNTM map dissatisfied you, here's a money saver: don't go watch the movie Alien vs. Predator, unless you like "so bad it's funny" movies.

8 Acknowledgments

Many thanks to Krzysztof Machelski who reviewed the code and prompted two bugs in the implementation.

References

[1] Andrei Alexandrescu. *Modern C++ Design*. Addison-Wesley Longman, 2001.

[2] Andrei Alexandrescu. Generic⟨Programming⟩: `yasli::vector` is on the move. *C++ Users Journal*, June 2004.

[3] D.R. Butenhof. *Programming with POSIX Threads*. Addison-Wesley, Reading, Massachusetts, USA, 1997.

[4] David L. Detlefs, Paul A. Martin, Mark Moir, and Guy L. Steele, Jr. Lock-free reference counting. In *Proceedings of the twentieth annual ACM symposium on Principles of distributed computing*, pages 190–199. ACM Press, 2001. ISBN 1-58113-383-9.

[5] Erich Gamma, Richard Helm, Ralph E. Johnson, and John Vlissides. *Design Patterns*. Addison-Wesley, Reading, MA, 1995.

[6] Scott Meyers and Andrei Alexandrescu. The Perils of Double-Checked Locking. *Dr. Dobb's Journal*, July 2004.

[7] Maged M. Michael. Scalable lock-free dynamic memory allocation. In *Proceedings of the ACM SIGPLAN 2004 conference on Programming language design and implementation*, pages 35–46. ACM Press, 2004. ISBN 1-58113-807-5.