Lockfree protection of data structures that are frequently read

by Max Neunhoeffer

Motivation

In multi-threaded applications running on multi-core systems, it occurs often that there are certain data structures, which are frequently read but relatively seldom changed. An example of this would be a database server that has a list of databases that changes rarely, but needs to be consulted for every single query hitting the database. In such sitations one needs to guarantee fast read access as well as protection against inconsistencies, use after free and memory leaks.

Therefore we seek a lock-free protection mechanism that scales to lots of threads on modern machines and uses only C++11 standard library methods. The mechanism should be easy to use and easy to understand and prove correct. This article presents a solution to this, which is probably not new, but which we still did not find anywhere else.

The concrete challenge at hand

Assume a global data structure on the heap and a single atomic pointer P to it. If (fast) readers access this completely unprotected, then a (slow) writer can create a completely new data structure and then change the pointer to the new structure with an atomic operation. Since writing is not time critical, one can easily use a mutex to ensure that there is only a single writer at any given time. The only problem is to decide, when it is safe to destruct the old value, because the writer cannot easily know that no reader is still accessing the old values. The challenge is aggravated by the fact that without thread synchronization it is unclear, when a reader actually sees the new pointer value, in particular on a multi-core machine with a complex system of caches.

If you want to see our solution directly, scroll down to “Source code links”. We first present a classical good approach and then try to improve on it.

Hazard pointers and their hazards

The “classical” lock-free solution to this problems are hazard pointers (see this paper and this article on Dr Dobbs). The basic idea is that each reading thread first registers the location of its own “hazard” pointer in some list, and whenever it wants to access the data structure, it sets its own hazard pointer to the value of P it uses to access the data, and restores it to `nullptr` when it is done with the read access.

A writer can then replace the old value of P with a pointer to a completely new value and then scan all registered hazard pointers to see whether any thread still accesses the old value. If all store operations to the hazard pointers and the one to P use `memory_order_seq_cst` (see this page for an explanation), then it is guaranteed that if a reader thread sees the old version of P, then it observes the change of its own hazard pointer earlier, therefore, because of the guaranteed sequential order of all stores with `memory_order_seq_cst`, the writer thread also observes the hazard pointer value before its own change of P.

This is a very powerful and neat argument, and it only uses the guaranteed memory model of the C++11 standard in connection with atomic operations in the STL. It has very good performance characteristics, because the readers just have to ensure `memory_order_seq_cst` by means of memory fence or equivalent instructions, and since one can assume that the actual hazard pointers reside in different cache lines there is no unnecessary cache invalidation.

However, this approach is not without its own hazards (pun intended). The practical problems in my opinion lie in the management of the hazard pointer allocations and deallocations and from the awkward registration procedure. A complex multi-threaded application can have various different types of threads, some dynamically created and joined. At the same time it can have multiple data structures that need the sort of protection discussed here. The position of the actual hazard pointer structure is thread-local information, and one needs a different one for each instance of a data structure that needs protection.

What makes matters worse is that at the time of thread creation the main thread function often does not have access to the protected data at all, due to data encapsulation and object-oriented design. One also does not want to do the allocation of hazard pointer structures lazily, since this hurts the fast path for read access.

If one were to design a “DataGuardian” class that does all the management of hazard pointers itself, then it would have to store the locations of the hazard pointers in thread-local data, but then it would have to be static and it would thus not be possible to use different hazard pointers for different instances of the DataGuardian. We have actually tried this and failed to deliver a simple and convenient implementation. This frustration lead us to our solution, which we describe next.

Lock-free reference counting

The fundamental idea is to use a special kind of reference counting in which a reading thread uses atomic compare-and-exchange operations to increase a reference counter before it reads P and the corresponding data and decreases the counter after it is done with the reading. However, the crucial difference to this standard approach is that every thread uses a different counter, all residing in pairwise different cache lines! This is important since it means that the compare-and-exchange operations are relatively quick since no contention with corresponding cache invalidations happens.

Before we do any more talking, here is the code for the simple version of the `DataProtector` class, first `DataProtector.h`:

```
#include <atomic>
#include <unistd.h>

template<int Nr>
class DataProtector {
    struct alignas(64) Entry {
      std::atomic<int> _count;
    };

    Entry* _list;

    std::atomic<int> _last;
    static thread_local int _mySlot;

  public:

    DataProtector () : _last(0) {
      _list = new Entry[Nr];
      // Just to be sure:
      for (size_t i = 0; i < Nr; i++) {
        _list[i]._count = 0;
      }
    }

    ~DataProtector () {
      delete[] _list;
    }

    void use () {
      int id = getMyId();
      _list[id]._count++;   // this is implicitly using memory_order_seq_cst
    }

    void unUse () {
      int id = getMyId();
      _list[id]._count--;   // this is implicitly using memory_order_seq_cst
    }

    void scan () {
      for (size_t i = 0; i < Nr; i++) {
        while (_list[i]._count > 0) {
          usleep(250);
        }
      }
    }

  private:

    int getMyId () {
      int id = _mySlot;
      if (id < 0) {
        id = _last++;
        if (_last > Nr) {
          _last = 0;
        }
        _mySlot = id;
      }
      return id;
    }

};
```

And a minuscule part in `DataProtector.cpp` for the definition of a static thread-local variable:

```
#include "DataProtector.h"
template<int Nr> thread_local int DataProtector<Nr>::_mySlot = -1;
template class DataProtector<64>;
```

In a multi-threaded application one would declare the following, either global or in some object intance:

```
std::atomic<Data*> P;
DataProtector Prot;
```

A reader uses this as follows:

```
Prot.use();
Data* Pcopy = P;   // uses memory_order_seq_cst by default
// continue accessing data via Pcopy
Prot.unUse();
```

A writer simply does (protected by some mutex):

```
Data* newP = new Data();
Data* oldP = P;
P = newP;
Prot.scan();
delete oldP;
```

The code speaks mostly for itself, because this is actually a very simple approach: We administrate multiple slots with reference counters, making sure that each resides in a different cache line by using alignment. Each thread chooses once and for all a slot (we store the number in static thread-local storage), valid for all instances of the DataProtector class. This leads to a very fast path for reading data.

The writer, which is always only one at a time using mutexes, first builds up a completely new copy of the protected data structure and then switches the atomic pointer P to the new value. From this point on all readers only see the new version. To ensure that no more readers access the old version, the writer simply scans all reference counters in the DataProtector class and waits until it has seen a 0 in each of them. It is not necessary to see zeros in all of them at the same time, it is enough to have seen a zero in each slot once. After that it is safe to destroy the old value of the protected data structure.

The proof that this works is equally simple as in the hazard pointer case above: The changes to the reference counters as well as the change to the global pointer P by the writer are all done with `memory_order_seq_cst`. That is, the C++ memory model guarantees that all these changes are observed by all threads in the same sequential order. A reader that observes the old value of P (and then subsequently reads the old version of the data structure), has incremented its reference counter before reading P. Therefore it observes the change to the counter before it observes the change to P by the writer. Thus the writer must observe the change to the counter also as happening before the change to P. Therefore it will always see some positive counter as long as a reader is still accessing the old value of P and the corresponding data structure.

We assumed that it is not a problem that the writer is somewhat slow, because writes are infrequent. Therefore, locking a mutex, reading all reference counters, whose number is of the order of magnitude of the number of reader threads, and waiting for each of them to drop to 0 once is not a performance problem.

The benefits of this approach are as follows: All management happens encapsulated in the DataProtector class, which is extremely simple to use. We have discussed the performance characteristics above and show a benchmark and comparison with other methods below.

There is a single convenience improvement, which we describe in the following section.

Convenience with scope guards

To make the usage for the readers even more convenient and reduce possiblities for bugs, we create a facility to use scope guards. We do this in the form of a small UnUser class, which is encapsulated in the DataProtector class. Modern C++11 features like type inference (auto) further help. After this modification, the reader uses the DataProtector as follows:

```
{
  auto unuser(Prot.use());
  Data* Pcopy = P;    // uses memory_order_seq_cst by default
  // continue accessing data via Pcopy
}
```

The unuser instance will have type DataProtector::UnUser and the use method of the DataProtector returns the right instance such that the destructor of the UnUser class automagically calls the unUse method of the DataProtector class, when the object goes out of scope. This method can then in turn be private. Without further talking, here is the code of the UnUser class:

```
// A class to automatically unuse the DataProtector:
class UnUser {
    DataProtector* _prot;
    int _id;

  public:
    UnUser (DataProtector* p, int i) : _prot(p), _id(i) {
    }

    ~UnUser () {
      if (_prot != nullptr) {
        _prot->unUse(_id);
      }
    }

    // A move constructor
    UnUser (UnUser&& that) : _prot(that._prot), _id(that._id) {
      // Note that return value optimization will usually avoid
      // this move constructor completely. However, it has to be
      // present for the program to compile.
      that._prot = nullptr;
    }

    // Explicitly delete the others:
    UnUser (UnUser const& that) = delete;
    UnUser& operator= (UnUser const& that) = delete;
    UnUser& operator= (UnUser&& that) = delete;
    UnUser () = delete;
};
```

There is nothing special to it, note that the rule of five is observed and that the implemented move constructor allows return value optimization to kick in, such that the value now returned by the use class of the DataProtector is directly constructed in the stack frame of the reader:

```
UnUser use () {
  int id = getMyId();
  _list[id]._count++;    // this is implicitly using memory_order_seq_cst
  return UnUser(this, id);  // return value optimization!
}
```

As already mentioned, the `unUse` method is now private, other than that, the code of the `DataProtector` is unchanged.

Source code links

All code is available online in this github repository:

https://github.com/neunhoef/DataProtector

There, we also publish the test code, which is used in the following section to measure performance.

Additionally, this is actually being used in published software in the source code of ArangoDB, see here and here for details.

Performance comparison with other methods

To assess the performance of our `DataProtector` class, we have done a comparison with the following methods:

1. `DataGuardian` with hazard pointers

   This is our own implementation of hazard pointers.

2. unprotected access

   This is just unprotected access, which is of course not an option at all, but interesting nevertheless. One sees, that the readers essentially just consult their caches which are updated eventually. There is no guarantee against use-after-free at all.

3. a mutex implementation

   This is a very simple-minded application where all readers and the writer share a global mutex.

4. a spin-lock implementation using boost atomics

   Again a very simple-minded implementation of spin-locks.

5. `DataProtector`

   This is our new class described in this article.

The test program simply starts a number of reader threads which constantly read a dummy data structure, thereby detecing use-after-delete and seeing a `nullptr`. We count reads per second and reads per second and thread.

Here are the results on an n1-standard-16 instance on Google Compute Engine (GCE) for various numbers of threads. The code has been compiled with `g++ -std=c++11 -O3 -Wall`. Results are in million reads per second (M/s), and million reads per second and thread (M/s/thread):

```
|Nr |DataGuardian|unprotected|Mutex       |Spinlock    |DataProtector
|Thr|  M/s   M/s/T|   M/S M/s/T|   M/S M/s/T|   M/S M/s/T|   M/S M/s/T
----------------------------------------------------------------------
| 1 | 34.0  34.06| 2259   2259|60.42 60.42|102.1 102.1| 84.3 84.35
| 2 | 64.6  32.31| 4339   2169| 9.33  4.66| 97.5  48.7|160.1 80.05
| 3 | 92.6  30.86| 6162   2054| 7.89  2.63| 93.5  31.1|234.3 78.12
| 4 |120.5  30.13| 8021   2005|13.96  3.49| 89.9  22.4|300.4 75.12
| 5 |146.3  29.26| 9814   1962|11.43  2.28| 88.3  17.6|367.4 73.49
| 6 |168.6  28.10|11302   1883| 9.23  1.53| 83.9  13.9|431.2 71.87
| 7 |208.3  29.76|12790   1827| 8.43  1.20| 81.0  11.5|498.5 71.21
| 8 |220.1  27.51|14075   1759| 8.39  1.04| 81.2  10.1|554.7 69.34
| 9 |258.1  28.68|14040   1560| 8.24  0.91| 81.0   9.0|542.1 60.23
|10 |282.4  28.24|14078   1407| 8.36  0.83| 81.9   8.1|540.9 54.09
|11 |301.0  27.37|14029   1275| 8.10  0.73| 81.0   7.3|524.4 47.67
```

```
|12 |321.2  26.76|14060  1171| 7.99  0.66| 82.2   6.8|518.4 43.20
|13 |345.0  26.54|14031  1079| 7.79  0.59| 82.3   6.3|510.9 39.30
|14 |366.7  26.19|14086  1006| 7.63  0.54| 82.8   5.9|502.5 35.89
|15 |385.9  25.73|14092   939| 7.54  0.50| 83.4   5.5|498.5 33.23
|16 |408.6  25.54|14276   892| 7.53  0.47| 82.9   5.1|491.9 30.74
|20 |408.0  20.40|14317   715| 7.98  0.39| 84.5   4.2|488.9 24.44
|24 |402.6  16.77|14196   591| 8.29  0.34| 84.0   3.5|525.3 21.88
|28 |398.6  14.23|14235   508| 8.43  0.30| 83.4   2.9|501.8 17.92
|32 |389.9  12.18|14123   441| 8.53  0.26| 83.4   2.6|528.9 16.52
|48 |385.2   8.02|14202   295| 8.62  0.17| 81.0   1.6|488.4 10.17
|64 |375.1   5.86|14196   221| 8.91  0.13| 79.2   1.2|508.3  7.94
```

The first column is the number of reader threads, in each of the five following columns there is first the total number of reads in all threads in millions per second and then the same number divided by the number of threads, which is the total number of reads per second and thread. The results have some random variation and are very similar when using the `clang` compiler.

One can see that both the hazard pointers in the `DataGuardian` class and the `DataProtector` class scale well, until the number of actual CPUs (16 vCPUs are 8 cores with hyperthreading) is reached. On such a machine 554 million reads per second with 8 threads is a good result, this means that every thread achieves 70 M reads per second and thus only spends around 14 nanoseconds for each. This shows that in this uncontented situation the atomic compare-and-exchange operations are quite fast.

References

- Hazard pointer article

  `http://www.research.ibm.com/people/m/michael/ieeetpds-2004.pdf`
- Dr Dobbs

  `http://www.drdobbs.com/lock-free-data-structures-with-hazard-po/184401890`
- Memory orders in C++

  `http://www.cplusplus.com/reference/atomic/memory_order/`
- DataProtector implementation and test code

  `https://github.com/neunhoef/DataProtector`
- The multi-model NoSQL database ArangoDB

  `https://www.arangodb.com`
- Source code of ArangoDB on github

  `https://github.com/ArangoDB/ArangoDB`
- DataProtector in the ArangoDB source code

  `https://github.com/ArangoDB/ArangoDB/blob/devel/lib/Basics/DataProtector.h`