C++ atomics, Boost.Lookfree, Hazard-Pointers und die Thread-Hölle

Aachen, den 14. Januar 2016

Max Neunhöffer

Unser Problem heute

Es war einmal . . .

eine multi-threaded Applikation, mit einer globalen Datenstruktur, die

- sehr häufig von verschiedenen Threads gelesen wird,
- sehr selten von verschiedenen Threads geändert wird.

Wie sorgt man für geschützten Verkehr?

(Mit guter Leseperformance!)

Hauptprogramm

```
int N = atoi(argv[1]);     // number of threads
std::vector<std::thread> readerThreads;
readerThreads.reserve(N);
std::thread* writerThread = new thread(writer);
usleep(500000);
for (int i = 0; i < N; i++) { readerThreads.emplace_back(reader, i); }

writerThread->join();
for (int i = 0; i < N; i++) { readerThreads[i].join(); }
delete writerThread;
```

Schützenswerte Daten

```
struct DataToBeProtected {
  DataToBeProtected(int i) : nr(i), isValid(true) {
  }
  ~DataToBeProtected() {
    isValid = false;
  }
  int nr;
  bool isValid;
};
```

Ein seltener Schreiber — ungeschützt

```
DataToBeProtected const* unprotected = nullptr;

void writer_unprotected () {
  DataToBeProtected* p;
  for (int i = 0; i < T+2; i++) {
    DataToBeProtected const* q = unprotected;
    p = new DataToBeProtected(i);
    unprotected = p;
    usleep(1000000);
    delete q;
  }
  delete unprotected;
  unprotected = nullptr;
}
```

Ein häufiger Leser — ungeschützt

```
std::atomic<uint64_t> total = 0;
void reader_unprotected (int) {
  uint64_t count = 0;
  time_t start = time(nullptr);
  while (time(nullptr) < start + T) {
    for (int i = 0; i < 1000; i++) {
      count++;
      DataToBeProtected const* p = unprotected;
      if (p == nullptr) { nullptrsSeen++; }
      else if (! p->isValid) { alarmsSeen++; }
    }
  }
  total += count;
}
```

Performance — ungeschützter Verkehr

# threads	M reads/s	M reads/s pro thread
1	1815.52	1815.52
2	3509.67	1754.83
3	3541.82	1180.61
4	3569.70	892.43
5	3361.92	672.38
6	3373.72	562.28
7	3570.97	510.13
8	3569.28	446.15

(auf meinem Laptop, Intel Core i7, 4 vCPUs (2 cores mit je 2 hyperthreads))

Ein häufiger Leser — mit Mutex

```
std::atomic<uint64_t> total = 0;
std::mutex mut;                                          // NEU
void reader_mutex (int) {
  uint64_t count = 0;
  time_t start = time(nullptr);
  while (time(nullptr) < start + T) {
    for (int i = 0; i < 1000; i++) {
      count++;
      std::lock_guard<std::mutex> locker(mut);   // NEU
      DataToBeProtected const* p = unprotected;
      if (p == nullptr) { nullptrsSeen++; }
      else if (! p->isValid) { alarmsSeen++; }
    }
  }
  total += count;
}
```

HÖLLE 1: Performance — mit Mutex

# threads	M reads/s	M reads/s pro thread
1	40.7785	40.7785
2	6.8525	3.4263
3	14.4974	4.8325
4	10.5641	2.6410
5	10.7236	2.1447
6	10.6119	1.7687
7	11.2502	1.6072
8	12.0467	1.5058

(auf meinem Laptop, Intel Core i7, 4 vCPUs (2 cores mit je 2 hyperthreads))

ALARM — ALARM — ALARM

HÖLLE 2: Klassische Lösung — Hazard Pointers

Idee Hazard-Pointer

Jeder **lesende thread** registriert die Adresse `&hazard` eines Pointers:

```
DataToBeProtected const* hazard = nullptr;
```

irgendwo.

Wenn er `DataToBeProtected const* d;` lesen möchte, dann

- setzt er `hazard = d;` davor,
- greift dann lesend zu,
- und setzt `hazard = nullptr;` danach.

HÖLLE 2: Klassische Lösung — Hazard Pointers

Idee Hazard-Pointer (Fortsetzung)

Jeder **lesende thread** registriert die Adresse `&hazard` eines Pointers:

```
DataToBeProtected const* hazard = nullptr;
```

irgendwo.

Ein **Schreiber** muss:

- Die Daten komplett neu in `DataToBeProtected* neu;` aufbauen,
- dann `DataToBeProtected const* alt = d;` setzen,
- dann `d = neu;` setzen,
- dann die Liste der registrierten Hazard-Pointer durchgehen bis er in jedem einen anderen Wert als `alt` gesehen hat,
- und dann `delete alt;` machen.

Performance — mit Hazard Pointers

# threads	M reads/s	M reads/s pro thread
1	30.728	30.728
2	59.510	29.754
3	82.316	27.439
4	105.378	26.342
5	104.152	20.830
6	102.441	17.074
7	97.923	13.989
8	99.136	12.392

(auf meinem Laptop, Intel Core i7, 4 vCPUs (2 cores mit je 2 hyperthreads))

HÖLLE 2: Funktioniert das???

NEIN! SO NICHT.

WARUM?

Es kann sein, dass der Schreiber den Update des Hazard-Pointers nicht sieht.

Unschönheiten von Hazard Pointers

- Allokation der Hazard-Pointers für jeden Thread.
- Die (zentrale) Registratur der Adressen.
- Man braucht für jede Datenstruktur eine getrennte Hazard-Pointer-Infrastruktur.

Neue Lösung — lock-free reference counting

Idee DataProtector

Jeder **lesende thread** bekommt dynamisch eine ID (`thread_local`).

Wenn er `DataToBeProtected const* d;` lesen möchte, dann

- erhöht er den Referenzzähler zu seiner ID
- greift dann lesend zu,
- und senkt dann den Referenzzähler zu seiner ID wieder.

Ein **Schreiber** muss:

- Die Daten komplett neu in `DataToBeProtected* neu;` aufbauen,
- dann `DataToBeProtected const* alt = d;` setzen,
- dann `d = neu;` setzen,
- dann die Referenzzähler scannen und in jedem eine 0 sehen,
- und dann `delete alt;` machen.

DataProtector 1

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

    static 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; }
}
```

DataProtector 2

```
private:
  int getMyId () {
    int id = _mySlot;
    if (id >= 0) { return id; }
    while (true) {
      int newId = _last + 1;
      if (newId >= Nr) {
        newId = 0;
      }
      if (_last.compare_exchange_strong(id, newId)) {
        _mySlot = newId;
        return newId;
      }
    }
  }
```

DataProtector 3

```
void use () {
  int id = getMyId();
  _list[id]._count++;
}
void unUse () {
  int id = getMyId();
  _list[id]._count--;
}
void scan () {
  for (size_t i = 0; i < Nr; i++) {
    while (_list[i]._count > 0) {
      usleep(250);
    }
  }
}
```

Ein häufiger Leser — mit DataProtector

```
std::atomic<DataToBeProtected*> protected;          // NEU
std::atomic<uint64_t> total = 0;
DataProtector protector;                            // NEU
void reader_dataprotector (int) {
  uint64_t count = 0; time_t start = time(nullptr);
  while (time(nullptr) < start + T) {
    for (int i = 0; i < 1000; i++) {
      count++;
      protector.use();                              // NEU
      DataToBeProtected const* p = protected;       // NEU
      if (p == nullptr) { nullptrsSeen++; }
      else if (! p->isValid) { alarmsSeen++; }
      protector.unUse();                            // NEU
    }
  }
  total += count;
}
```

Performance — mit DataProtector

# threads	M reads/s	M reads/s pro thread
1	66.356	66.356
2	128.639	64.319
3	126.808	42.270
4	124.165	31.041
5	120.777	24.156
6	122.100	20.350
7	121.239	17.320
8	119.507	14.938

(auf meinem Laptop, Intel Core i7, 4 vCPUs (2 cores mit je 2 hyperthreads))

PROBLEM 1

```
int a=0;
int b=0;
```

Thread A	Thread B
`a=1;`	`b=2;`
`cout << "A" << a << b << endl;`	`cout << "B" << a << b << endl;`

Kann `A10 B12` oder `A12 B02` oder `A12 B12` oder `A10 B02` erzeugen!

Caches haben alte Daten!

PROBLEM 2

```
int a=0;
int b=0;
```

Thread A	Thread B
`a=1;`	`cout << "B" << a << b << endl;`
`b=2;`	

Kann `B00` oder `B10` oder `B12` oder `B02` erzeugen!

Compiler oder CPU ändern die Reihenfolge von Operationen!

BEWEIS: Der `DataProtector` funktioniert!

```
std::atomic<uint32_t> refcount = 0;
std::atomic<char*>    data;
```

Reader thread	Writer thread
`refcount++;`	`char* old = data;`
`char* tmp = data;`	`data = newData;`
`// read *tmp`	`while (refcount > 0) { }`
`refcount--;`	`delete old;`

Entweder `tmp == newData` oder

`data = newData;` passiert nach `tmp = data;`

(in der totalen Ordnung ***S*** in §29.3 Absatz 3)

und deshalb sieht der Writer den erhöhten `refcount`.

BEWEIS: Hazard-Pointer funktionieren!

```
std::atomic<char*> hazard = nullptr;
std::atomic<char*> data;
```

Reader thread	Writer thread
`char* tmp;`	`char* old = data;`
`do { tmp = data;`	`data = newData;`
`hazard = tmp;`	`while (hazard == old) { }`
`} while (tmp != data);`	`delete old;`
`// read *tmp`	
`hazard = nullptr;`	

Entweder `tmp == newData` oder

`data = newData;` passiert nach `tmp != data;`

und deshalb sieht der Writer `hazard == old`.

Der C++-Standard

§29.3 Satz 3

There shall be a single total order *S* on all `memory_order_seq_cst` operations, **consistent** with the "happens before" order and modification orders for all affected locations, such that each `memory_order_seq_cst` operation *B* that loads a value from an atomic object *M* observes one of the following values:

- the **result of the last modification A of M that precedes B in** *S*, if it exists, or
- if *A* exists, the result of some modification of *M* that is not `memory_order_seq_cst` and that does not happen before *A*, or
- if *A* does not exist, the result of some modification of *M* that is not `memory_order_seq_cst`.

Boost lock-free

Boost hat:

- `boost::lockfree::queue`
  a lock-free multi-produced/multi-consumer queue
- `boost::lockfree::stack`
  a lock-free multi-produced/multi-consumer stack
- `boost::lockfree::spsc_queue`
  a wait-free single-producer/single-consumer queue
  (commonly known as ringbuffer)

Links

ArangoDB	`https://www.arangodb.com`
Blog-Artikel	`https://www.arangodb.com/2015/08/lockfree-protection-of-data-structures-that-are-frequently-read/`
Code	`https://github.com/neunhoef/DataProtector`
Hazard-Ptrs	`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`
C++ atomics	`http://www.cplusplus.com/reference/atomic/memory_order/`
Boost lockfree	`http://www.boost.org/doc/libs/1_59_0/doc/html/lockfree.html`