Multithreading Using Lockless Lists and RCU

Ansel Sermersheim

CppNow - May 2017

Introduction

- Multithreading revisited
- A better way
- Containers + Destructors = Deadlocks
- Introducing a new solution: RCU
- Putting it all together

Multithreading Revisited

- Part I

Multithreading Revisited

```
// example 1 - any issues?

ComplicatedObject * createObject(int param1, double param2)  {
  ComplicatedObject * retval;

  retval = new ComplicatedObject();
  retval->doSomething(param1);
  retval->somethingElse(param2);

  return retval;
}
```

Multithreading Revisited

```
// example 2 - any issues?

class MyCache {
  public:
    void insert(std::string key, ComplicatedObject * element);
    ComplicatedObject * lookup(std::string key) const;

  private:
    std::map<std::string, ComplicatedObject *> m_cache;
    std::shared_timed_mutex m_cacheMutex;
};


ComplicatedObject * MyCache::lookup(std::string key)  {
   std::shared_lock<std::shared_timed_mutex> lock(m_cacheMutex);

   return m_cache[key];
}
```

Multithreading Revisited

- Problems with example 2
  - returns a raw ptr, who is responsible for deleting it
  - what if someone else deletes the object
  - what if I delete the object but I do not remove it from the std::map
  - if the key is not found in the map, a reference to the mapped value with a nullptr is inserted in the map
    - undefined behavior since the lock is a “read” lock

A Better Way . . .

- Part II

A Better Way . . .

- class guarded<T>

class guarded<T> (1 of 4)

```
template <typename T, typename M = std::mutex>
class guarded {
  public:
    using handle = std::unique_ptr<T, deleter>;

    template <typename... Us>
    guarded(Us &&... data);

    handle lock();
    handle try_lock();

    template <class Duration>
    handle try_lock_for(const Duration & duration);
    template <class TimePoint>
    handle try_lock_until(const TimePoint & timepoint);

  private:
    T m_obj;
    M m_mutex;
};
```

class guarded<T> (2 of 4)

```
template <typename T, typename M>
template <typename... Us>
guarded<T, M>::guarded(Us &&... data) : m_obj(std::forward<Us>(data)...)
{
}

template <typename T, typename M>
auto guarded<T, M>::lock() -> handle
{
    std::unique_lock<M> lock(m_mutex);
    return handle(&m_obj, deleter(std::move(lock)));
}
```

class guarded<T> (3 of 4)

```
template <typename T, typename M>
auto guarded<T, M>::try_lock() -> handle
{
    std::unique_lock<M> lock(m_mutex, std::try_to_lock);

    if (lock.owns_lock()) {
        return handle(&m_obj,  deleter(std::move(lock)));
    } else {
        return handle(nullptr, deleter(std::move(lock)));
    }
}
```

class guarded<T> (4 of 4)

```
class deleter
{
    public:
        using pointer = T *;

        deleter(std::unique_lock<M> lock) : m_lock(std::move(lock))
        {
        }

        void operator()(T * ptr) {
            if (m_lock.owns_lock()) {
                m_lock.unlock();
            }
        }

    private:
        std::unique_lock<M> m_lock;
};
```

A Better Way . . . Recap

- class guarded<T>
  - exclusive locks
  - C++11

- class shared_guarded<T>
  - exclusive locks
  - shared locks
  - C++14 or boost::thread

- class ordered_guarded<T>
  - shared locks
  - blocking modifications to shared data (via lambda)
  - C++14 or boost::thread

A Better Way . . . Recap

- class deferred_guarded<T>
  - shared locks
  - nonblocking modifications to data (via lambda)
  - deadlock free eventual consistency
  - C++14 or boost::thread

- class lr_guarded<T>
  - shared access without locks
  - blocking modifications to data (via lambda)
  - readers block writers
  - readers never see data older than the previous write
  - C++11

A Better Way . . . Recap

- class cow_guarded<T>
  - shared access without locks
  - blocking modifications to data (via lambda)
  - only other writers can block writers
  - readers see a snapshot of data
  - unwanted modifications can be discarded
  - C++11

Example 2 Revisited -- Using shared_guarded<T>

```
class MyCache {
  public:
    void insert(std::string key, std::shared_ptr<ComplicatedObject> element);
    std::shared_ptr<ComplicatedObject> lookup(std::string key) const;

  private:
    shared_guarded<std::map<std::string,
                            std::shared_ptr<ComplicatedObject>>> m_cache;
};


std::shared_ptr<ComplicatedObject> MyCache::lookup(std::string key)  {
  auto handle = m_cache.lock_shared();
  auto iter   = handle->find(key);

  if (iter != handle->end()) {
     return iter->second;
  }
  return nullptr;
}
```

Example 2 Revisited -- Using shared_guarded<T>

```
// any issues?

void MyCache::insert(std::string key, std::shared_ptr<ComplicatedObject> element)
{
    auto handle = m_cache->lock();
    handle->emplace(key, element);
}
```

Example 2 Revisited -- Using ordered_guarded<T>

```
// any issues?

void MyCache::insert(std::string key, std::shared_ptr<ComplicatedObject> element)
{
    m_cache.modify(
          [&key, &element]
          (std::map<std::string, std::shared_ptr<ComplicatedObject>> & map)
          {
             map.emplace(key, element);
          });
}
```

Example 2 Revisited -- Using deferred_guarded<T>

```
// any issues?

void MyCache::insert(std::string key, std::shared_ptr<ComplicatedObject> element)
{
    m_cache.modify_detach(
            [k = std::move(key), e = std::move(element)]
            (std::map<std::string, std::shared_ptr<ComplicatedObject>> & map)
            {
               map.emplace(k, e);
            });
}
```

Example 2 Revisited -- Using lr_guarded<T>

```
// any issues?

void MyCache::insert(std::string key, std::shared_ptr<ComplicatedObject> element)
{
    m_cache.modify(
            [&key, &element]
            (std::map<std::string, std::shared_ptr<ComplicatedObject>> & map)
            {
               map.emplace(key, element);
            });
}
```

Example 2 Revisited -- Using cow_guarded<T>

```
// any issues?

void MyCache::insert(std::string key, std::shared_ptr<ComplicatedObject> element)
{
    auto handle = m_cache->lock();
    handle->emplace(key, element);
}
```

Example 3

```
class MyCache {
  public:
    std::shared_ptr<ComplicatedObject> lookup(std::string key) const;

    void insert(std::string key, std::shared_ptr<ComplicatedObject> element);
    void insert_batch(std::map<std::string, std::shared_ptr<ComplicatedObject>>);

  private:
    // must be called with m_cacheMutex held
    void internal_insert(std::string key, std::shared_ptr<ComplicatedObject> e);

    std::map<std::string, ComplicatedObject *> m_cache;
    std::shared_timed_mutex m_cacheMutex;
};
```

Example 3 Revisited -- using deferred_guarded<T>

```
class MyCache {
  public:
    std::shared_ptr<ComplicatedObject> lookup(std::string key) const;

    void insert(std::string key, std::shared_ptr<ComplicatedObject> element);
    void insert_batch(std::map<std::string, std::shared_ptr<ComplicatedObject>>);

  private:
    using shared_handle = deferred_guarded<std::map<std::string,
                                std::shared_ptr<ComplicatedObject>>>::shared_handle;

    void internal_insert(std::string key, std::shared_ptr<ComplicatedObject> e,
                                shared_handle & cache);

    deferred_guarded<std::map<std::string,
                                std::shared_ptr<ComplicatedObject>>> m_cache;
};
```

Example 3 Revisited -- using deferred_guarded<T>

- Instead of writing code
  - class MyCache does not need to be implemented as a class

```
using MyCache = deferred_guarded<std::map<
        std::string, std::shared_ptr<ComplicatedObject>>>;
```

Containers + Destructor = Deadlocks

- Part III

Containers + Destructor = Deadlocks

- May 2016
  - libGuarded library release 1.0.0

- June 2016
  - medical leave   *(aftermarket knee installation)*

- Jan 2017
  - integrated libGuarded with CsSignal library
  - ran thread sanitizer and it reported a deadlock
  - libGuarded 1.0.0 was supposed to prevent threading issues
  - now what?

Containers + Destructor = Deadlocks

- Real world issue from CsSignal Library
  - each connection involves one sender object and one receiver object
  - example: a pushButton is connected to a window
  - signal: PushButton::clicked() slot: Window::close()

  - each sender of a signal has a connection list
    - pushButton destructor must update each receiver
  - each receiver of a signal has a sender list
    - window destructor must update each sender

Containers + Destructor = Deadlocks

- Real world issue from CsSignal Library
  - what order should these containers be locked
    - lock the sender’s connection list
    - lock the receiver’s sender list

  - pushButton destructor must:
    - read its own connection list to find receivers
    - write to each receiver’s sender list

  - window destructor must:
    - read its own sender list to find senders
    - write to each sender’s connection list

Containers + Destructor = Deadlocks

- Possible solutions, not really
  - ignore this problem (ostrich algorithm)
  - wait until the destructors work it out
  - try_lock()
  - alternating lock / unlock until someone wins
  - check for this deadlock and assert()
  - mark unit test flaky so your CI does not fail
  - never run thread sanitizer

Containers + Destructor = Deadlocks

- Possible solutions
  - CsSignal library was designed to delegate responsibility for thread management to libGuarded
  - valid for the pushButton and the window to both be in their respective destructors concurrently
  - the solution to this deadlock needs to be a change in libGuarded and not in CsSignal

Containers + Destructor = Deadlocks

- What can be added to libGuarded
  - what we really want is a thread aware container
  - writers must not block readers
  - readers do not block at all
  - iterators are not invalidated by writers

Containers + Destructor = Deadlocks

- Feb 2017
  - add a new class to libGuarded to support the CsSignal threading requirements

- March 2017
  - completed libGuarded 1.1.0 integration with CsSignal
  - thread sanitizer run on CsSignal, happy

Containers + Destructor = Deadlocks

- CsSignal library, before libGuarded

```
CsSignal::SignalBase::~SignalBase()
{
  std::lock_guard<std::mutex> lock(m_mutex_connectList);

  if (m_activateBusy > 0)  {
    std::lock_guard<std::mutex> lock(get_mutex_beingDestroyed());
    get_beingDestroyed().insert(this);
  }

  for (auto & item : m_connectList) {
    const SlotBase * receiver = item.receiver;

    std::lock_guard<std::mutex> lock{receiver->m_mutex_possibleSenders};

    auto &senderList = receiver->m_possibleSenders;
    senderList.erase(std::remove_if(senderList.begin(), senderList.end(),
                      [this](const SignalBase * x){ return x == this; }),
                  senderList.end());
  }
}
```

Containers + Destructor = Deadlocks

- CsSignal Library, after libGuarded

```
CsSignal::SignalBase::~SignalBase()
{
  auto senderListHandle = m_connectList.lock_read();

  for (auto & item : * senderListHandle) {
    auto receiverListHandle = item.receiver->m_possibleSenders.lock_write();
    auto iter = receiverListHandle->begin();

    while (iter != receiverListHandle->end())   {

      if (*iter == this) {
        iter = receiverListHandle->erase(iter);

      } else {
         ++iter;

      }
    }
  }
}
```

Introducing a new solution: RCU

- Part IV

Introducing a new solution: RCU

- What is RCU?
  - RCU stands for “Read, Copy, Update”
  - a published algorithm for a multithreaded linked list
  - only one writer at a time
  - multiple concurrent readers
  - readers are lockless
  - readers do not block writers

Introducing a new solution: RCU

- How does RCU work?
  - defined procedure for modifying a list node
    - *read* current node
    - make a *copy* of the node
    - *update* pointers so all subsequent readers see only the new node *(nodes are not deleted at this step)*
    - wait until "later"
    - delete the old node

Introducing a new solution: RCU

- Example of RCU - In the linux kernel
  - the concept of “later” as defined in the kernel
    - each CPU goes through a step called a “grace period”
    - references to an RCU list can not be held during a grace period
    - while reading, a thread must never sleep or block
    - since there is a fixed number of CPUs, there is a limit on how many readers can exist
    - writer waits for all CPUs to execute a grace period, then it is safe to delete the old node

Introducing a new solution: RCU

- Example of RCU - In ~~the linux kernel~~ libGuarded
    - the concept of “later” ~~as defined in the kernel~~
        - ~~each CPU goes through a step called a “grace period”~~
        - ~~references to an RCU list can not be held during a grace period~~
        - ~~while reading, a thread must never sleep or block~~
        - ~~since there is a fixed number of CPUs, there is a limit on how many readers can exist~~

Introducing a new solution: RCU

- So what is RCU in a C++ library?
  - why defining “later” is complicated
    - there is no concept of a grace period
    - references may be held for a long time
    - references may be held while sleeping or blocking
    - number of threads currently running is dynamic
    - making writers block until readers finish is undesirable

- Is there a way to implement the RCU technique in a C++ library?

Introducing a new solution: RCU

- rcu_guarded<rcu_list<T, A>>
  - wrapper which controls access to the RCU container
- rcu_list<T, A>
  - container which implements the RCU algorithm

Introducing a new solution: RCU

- rcu_guarded public API
  - const method, nonblocking, returns a const read_handle
    - lock_read()
  - non-const method, exclusive lock, returns a write_handle
    - lock_write()

Introducing a new solution: RCU

- rcu_list public API
  - const methods accessible to readers
    - begin(), end()

  - non-const methods accessible to writers
    - insert(), erase(), push_back(), etc

Introducing a new solution: RCU

- rcu_list<T>::insert()
  - allocate new node
  - update new node’s next and prev pointers
  - update prev node’s next pointer
  - update next node’s prev pointer

  - concurrent readers will either see the new node or not
  - corner cases, when inserting at head or tail
  - pointers must be updated atomically

Introducing a new solution: RCU

- rcu_list<T>::erase()
  - update prev->next and next->prev to skip over this node
  - mark this node deleted
  - add this node to the head of a special internal list

  - concurrent readers will either see the old node or not
  - corner cases, when erasing the head or tail
  - pointers must be updated atomically

Introducing a new solution: RCU

- The special internal list - zombie list
  - (single) linked list
  - used to track when a node in rcu_list has been erased
    - zombie_node
  - used to track when a read handle to rcu_list was requested
    - read_in_process

```
struct zombie_list_node {
    ...
    std::atomic<zombie_list_node *> next;

    node * zombie_node;
    std::atomic<rcu_guard *> read_in_process;
};
```

Introducing a new solution: RCU

- Zombie list maintenance
  - when a read handle is requested rcu_guard adds an entry to the zombie list   (for reference this is spot c)
  - when the reader completes rcu_guard begins walking from this entry (spot c) in an attempt to clean up the zombie list
  - if the end of the zombie list is reached before another reader type entry, then every zombie from (spot c) to the end of the list is safe to delete
  - if another reader type entry is found, the reader entry (spot c) is removed and no other action is taken

Introducing a new solution: RCU

- Additional aspects of rcu_list
  - read_lock() returns a read handle to the rcu_list
  - a read handle can be used to retrieve an iterator
  - this iterator will be valid as long as the read handle is in scope

  - normally erasing an element of a list would invalidate iterators to that element

Introducing a new solution: RCU

- Additional aspects of rcu_list
  - no synchronization between readers so modifying an element directly can result in a race condition
    - to prevent this race condition all iterators are const
  - data in a list which is mutable can be modified by a reader even though the iterator is const
    - readers typically should not modify data
    - mutable data should be atomic if possible
  - to modify data in an rcu_list use insert() and erase()

Introducing a new solution: RCU

- Difference between linux RCU and libGuarded RCU
  - linux RCU readers have very little cost
  - libGuarded requires memory allocation for each read handle
  - libGuarded requires cleanup each time a reader completes

  - linux RCU writers wait for readers to finish
  - libGuarded writers do not need to wait

  - linux RCU is optimized for read speed, write performance can be poor
  - libGuarded RCU is designed for nonblocking operations

Introducing a new solution: RCU

- libGuarded 1.2.0
  - rcu_list::replace()
  - rcu_list::update()
  - read_handle::unlock()
  - write_handle::unlock()

  - add associative containers

Putting it all Together

- Part V

Putting it all Together

- Piece by piece
  - developing CopperSpice proved we needed to design a standalone Signal / Slot library (CsSignal)
  - deadlocks in CsSignal demanded a threading library
  - unable to document CopperSpice we created DoxyPress and switched parsing from lex to clang for C++
  - mangled text required a Unicode aware string library

  - CsSignal uses libGuarded
  - CopperSpice uses CsSignal and CsString
  - DoxyPress uses CopperSpice

Future Plans

- CsString
    - add ISO-8859-1 encoding (maybe others)
    - implement small string optimization
    - add locale aware comparison using Unicode algorithms
    - add normalization functions

- libGuarded
    - add associative containers
    - add lock free containers

Future Plans

- CopperSpice
  - complete QString8 and QString16
  - redesign QMap and QHash leveraging the STL
  - optimize QVariant
  - lambda based indexOf and lastIndexOf, all container classes
  - MSVC using clang front end

- CsSignal
  - improve move semantics

Future Plans

- DoxyPress
  - add parsing support for clang 3.8 and clang 3.9
  - optimize clang integration used in parsing
  - refactor comment parser
  - improve unicode support

Libraries & Applications

- CopperSpice
  - libraries for developing GUI applications
- PepperMill
  - converts Qt headers to CS standard C++ header files
- CsSignal Library
  - thread aware signal / slot library
- CsString Library
  - unicode aware string support library
- LibGuarded
  - multithreading library for shared data

Libraries & Applications

- KitchenSink
  - one program which contains 30 demos
  - links with almost every CopperSpice library

- Diamond
  - programmers editor which uses the CS libraries

- DoxyPress & DoxyPressApp
  - application for generating documentation for a variety of computer languages in numerous output formats

Where to find our libraries

- www.copperspice.com
- download.copperspice.com
- forum.copperspice.com

- ansel@copperspice.com
- barbara@copperspice.com

- Questions? Comments?