Multithreading is the answer. What is the question?

Ansel Sermersheim & Barbara Geller

ACCU / C++

January 2018

Introduction

- What is Multithreading
- Terminology
- When is Multithreading the answer
- Multithreading Analogies
- Real World Example
- A better way - libGuarded
- libGuarded RCU

What is Multithreading

- Bob is a seasoned C++ programmer

- Bob has a complex issue to solve

- Bob has always wanted to use multithreading

- Unfortunately, Bob now has N issues to solve
    - at least one race condition
    - maybe a few memory leaks
    - random runtime crash discovered by a high profile customer

What is Multithreading

- If you want to learn multithreading, find a problem which actually requires a multithreaded solution

- Do not take your current problem and force multithreading to be the solution

What is Multithreading

- Multithreading is the ability of a program to execute multiple instructions at the same time

    - a mechanism by which a single set of code can be used by several threads at different stages of execution

    - the ability to execute different parts of a program simultaneously

    - multithreading may be considered as concurrency if the threads interact or parallelism if they do not

Terminology

- Thread
  - work which can be scheduled to execute on one core
  - a thread is contained inside a process
  - each thread has its own call stack
- Process
  - used to start a separate program
  - threads in the same process share most resources
  - if only one thread in a process the program is not multithreaded
  - example: start make which in turn will launch clang in a separate process

Terminology

- Resource
  - computer memory location
  - file handle
  - non thread-safe C++ objects

- A resource must not be accessed by multiple threads simultaneously

Terminology

- Race condition
    - occurs when a resource is accessed by multiple threads simultaneously, and at least one access is a write
    - undefined behavior

Terminology

- A person enters a phone booth to use the phone
  - the person must hold the door handle of the booth to prevent another person from using the phone
  - when finished with the call, the person lets go of the door handle
  - now another person is allowed to use the phone

thread	a person
mutex	the door handle
lock	the person’s hand
resource	the phone

Why is Multithreading Complicated

- Abstraction, Coding, Communication
  - pair programming
  - multiple developers working on the same code
  - you read your own code a few months later

- Mutexes and Locks
  - multi-user programming relies on record locks
  - multithreading locks are not the same, however it uses the same word

When to use Multithreading

- Problems for which multithreading is the answer
  - tasks which can intuitively be split into independent processing steps
  - a problem where each step has a clear input and output
  - intensive computations
  - continuous access to a large read-only data set
  - processing a stream of large data files

When to use Multithreading

- Problems for which multithreading is the only answer
  - tasks whose performance would be unacceptable as a single thread
  - processes where the workload cannot be anticipated
  - manage concurrent access to multiple resources, such as an operating system
  - external clients sending requests to a process in a random and unpredictable fashion, such as PostgreSQL

Multithreading Analogy (A)

- Stage: A kitchen
  - two chefs
    - each chef will represent a thread
  - two knives
    - each knife is a local resource

- Requirement: make 50 fruit salads
- Solution: each chef will make 25 fruit salads

Threading Code (A)

```
std::thread chef1(
  []() {
    for(int i = 0; i < 25; ++i) {
      makeFruitSalad();
    }
  }
);

// same code as for chef one
std::thread chef2(...);

chef1.join();
chef2.join();
```

Multithreading Analogy (B)

- Stage: A kitchen
  - two chefs
    - each chef will represent a thread
  - two knives
    - each knife is a local resource
  - one oven
    - shared resource

- Requirement: make 50 apple pies
- Solution: each chef will independently make 25 apple pies

Threading Code (B)

```
Oven vikingOven;
std::mutex oven_mutex;

std::thread chef1( [&oven_mutex, &vikingOven]()
  {
    for(int i = 0; i < 25; ++i) {
      Pie anotherPie;
      anotherPie.makeCrust();
      anotherPie.putApplesInPie();
      std::lock_guard<std::mutex> oven_lock(oven_mutex);
      vikingOven.bakePie(anotherPie, 375, 35);
    }
  }
);

std::thread chef2(...);

chef1.join();
chef2.join();
```

Multithreading Analogy (C)

- Stage: A kitchen
  - two chefs
    - each chef will represent a thread
  - two knives
    - each knife is a local resource
  - one oven
    - shared resource

- Requirement: make 50 apple pies
- Solution: one chef prepares pies, the second chef bakes the pies in the oven

Threading Code (C-1)

```
Oven vikingOven;
threadsafe_queue<Pie> conveyorBelt;

std::thread chef1( [&conveyorBelt]()
  {
    for(int i = 0; i < 50; ++i) {
      Pie anotherPie;
      anotherPie.makeCrust();
      anotherPie.putApplesInPie();

      // give the pie away
      conveyor_belt.queue(std::move(anotherPie));
    }
  }
);
```

Threading Code (C-2)

```
std::thread chef2( [&conveyorBelt, &vikingOven]()
  {
    for(int i = 0; i < 50; ++i) {
      Pie anotherPie = conveyorBelt.dequeue();

      // bakePie method is blocking
      vikingOven.bakePie(anotherPie, 375, 35);
    }
  }
);

chef1.join();
chef2.join();
```

Threading Code (C)

- Can this design be optimized?
- Can these threads cause a deadlock?
- Are there any race conditions?

Multithreading Analogy (D)

- Stage: A kitchen

- Requirement: 25 fruit salads and 25 chicken sandwiches
- Solutions:
  - each chef independently makes a fruit salad, cleans up, and then makes a chicken sandwich, 25 times
  - one chef makes only the 25 fruit salads while the other chef makes only the 25 chicken sandwiches
  - both chefs each make the 25 fruit salads tracking how many were made in a shared data location
    - as soon as the fruit salads are finished they both switch to making chicken sandwiches

Multithreading Analogy (E)

- Stage: A kitchen
  - one oven, one brick pizza oven, one ice cream maker
    - shared resources
- Requirement:
  - anyone can randomly order pizza, garlic knots, apple pie, or ice cream
- Solution: pandemonium

Multithreading Analogy (E)

```
Oven vikingOven;
std::mutex vikingOven_mutex;

Oven brickOven;
std::mutex brickOven_mutex;

IceCreamMaker iceCreamMaker;
std::mutex iceCream_maker_mutex;

class Food  { ... };

class Pizza { ... };
class GarlicKnots { ... };
class ApplePie { ... };
class IceCream { ... };
```

Multithreading Analogy (E)

```
void eat(Food && food) {
  std::cout << "Patron was served: " << food.name();
};

using PatronTicket = std::future<std::unique_ptr<Food>>;
using ChefTicket   = std::promise<std::unique_ptr<Food>>;
```

Multithreading Analogy (E)

```
std::thread patron1( []() {
    PatronTicket knots    = orderGarlicKnots();
    PatronTicket pizza    = orderPizza();
    PatronTicket iceCream = orderIceCream();

    eat(knots.get());
    eat(pizza.get());
    eat(icecream.get());
});

std::thread patron2( []() {
    PatronTicket iceCream = orderIceCream();
    PatronTicket applePie = orderApplePie();

    eat(iceCream.get());
    eat(applePie.get());
});
```

Multithreading Analogy (E)

```
class Order { ... };
std::atomic<bool> restaurantOpen;
threadsafe_queue<Order> orderQueue;

std::thread chef1( [&]() {
  while(restaurantOpen) {
    Order nextOrder = orderQueue.dequeue();
    nextOrder.process();
  }
});

std::thread chef2( [&]() {
  while(restaurantOpen) {
    Order nextOrder = orderQueue.dequeue();
    nextOrder.process();
  }
});
```

Multithreading Analogy (E)

```
PatronTicket orderPizza() {
  ChefTicket chefTicket;
  PatronTicket patronTicket = chefTicket.get_future();

  Order order{ [ticket = std::move(chefTicket)]() {
    std::unique_ptr<Pizza> pizza = std::make_unique<Pizza>();
    pizza->addSauce();
    pizza->addCheese();
    std::lock_guard<std::mutex> lock(brickOven_mutex);
    pizza = brickOven.bake(std::move(pizza));
    ticket->set_value(std::move(pizza));
  }};

  orderQueue.queue(std::move(order));
  return patronTicket;
}
```

Multithreading Analogy (E)

- Items to consider about this example
    - single queue is not efficient
        - one queue per thread will improve performance
        - an idle thread can steal work from other queues, this is called "work stealing" and is a common feature
    - a chef should not be waiting for a pizza to bake
    - locking should not be arbitrary
        - `std::lock_guard<std::mutex> lock(brickOven_mutex);`
    - if we add a new menu item and forget to lock a resource, we may introduce a race condition

How to review Multithreading Code

```
// example 1 - any issues?

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

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

  return retval;
}
```

How to review Multithreading Code

```
// example 2 - any issues?

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

  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];
}
```

How to review Multithreading Code

- Problems with example 2
  - returns a raw ptr, who is responsible for deleting it
  - what if someone else deletes the object
  - what if we delete the object but do not remove it from the map
  - if the key is not found in the map, an entry is implicitly inserted with a value of nullptr
    - this insert is undefined behavior since the lock was for a read and now a write has been performed

Real World Example

- CsSignal Library - dilemma
  - 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

Real World Example

- 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

Real World Example

- 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

A Better Way . . .

- class guarded<T>

class guarded<T>

```
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();

    ( continued . . . )
```

class guarded<T>

```
    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>

```
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>

```
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>

```
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;
};
```

Multithreading Analogy -- using guarded<T>

```
PatronTicket orderPizza() {
  ChefTicket chefTicket;
  PatronTicket patronTicket = chefTicket.get_future();

  Order order ( [ticket=std::move(chefTicket)] ()
    {
        std::unique_ptr<Pizza> pizza = std::make_unique<Pizza>();
        pizza.addSauce();
        pizza.addCheese();
        ovenHandle = brickOven.lock();
        pizza = ovenHandle->bake(std::move(pizza));
        ticket->set_value(std::move(pizza));
    }
  );

  orderQueue.queue(std::move(order));
  return patronTicket;
}
```

A Better Way . . .

- class guarded<T>
- class shared_guarded<T>

class shared_guarded<T>

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

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

    handle lock();
    handle try_lock();

    ( continued . . . )
```

class shared_guarded<T>

```
  template <class Duration>
  handle try_lock_for(const Duration & duration);

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

public:
  using shared_handle = std::unique_ptr<const T, shared_deleter>;

  shared_handle lock_shared() const;
  shared_handle try_lock_shared() const;

  ( continued . . . )
```

class shared_guarded<T>

```
    template <class Duration>
    shared_handle try_lock_shared_for(const Duration & duration) const;

    template <class TimePoint>
    shared_handle try_lock_shared_until(const TimePoint & timepoint) const;

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

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;
};
```

Example 2 Revisited -- Using shared_guarded<T>

```
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);
}
```

A Better Way . . .

- class guarded<T>
- class shared_guarded<T>
- class ordered_guarded<T>

class ordered_guarded<T>

```
template <typename T, typename M = std::shared_timed_mutex>
class ordered_guarded
{
  public:
    using shared_handle = std::unique_ptr<const T, shared_deleter>;

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

    template <typename Func>
    void modify(Func && func);

    ( continued . . . )
```

class ordered_guarded<T>

```
    shared_handle lock_shared() const;
    shared_handle try_lock_shared() const;

    template <class Duration>
    shared_handle try_lock_shared_for(const Duration & duration) const;

    template <class TimePoint>
    shared_handle try_lock_shared_until(const TimePoint & timepoint) const;

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

class ordered_guarded<T>

```
template <typename T, typename M>
template <typename Func>
void ordered_guarded<T, M>::modify(Func && func)
{
    std::lock_guard<M> lock(m_mutex);

    func(m_obj);
}
```

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);
          });
}
```

A Better Way . . .

- class guarded<T>
- class shared_guarded<T>
- class ordered_guarded<T>
- class deferred_guarded<T>

class deferred_guarded<T>

```
template <class T>
typename std::add_lvalue_reference<T>::type declref();

template <typename T, typename M = std::shared_timed_mutex>
class deferred_guarded
{
  public:
    using shared_handle = std::unique_ptr<const T, shared_deleter>;

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

    template <typename Func>
    void modify_detach(Func && func);

    ( continued . . . )
```

class deferred_guarded<T>

```
template <typename Func>
auto modify_async(Func && func) ->
    typename std::future<decltype(std::declval<Func>()(declref<T>()))>;

shared_handle lock_shared() const;
shared_handle try_lock_shared() const;

template <class Duration>
shared_handle try_lock_shared_for(const Duration & duration) const;

template <class TimePoint>
shared_handle try_lock_shared_until(const TimePoint & timepoint) const;

( continued . . . )
```

class deferred_guarded<T>

```
  private:
    T          m_obj;
    mutable M m_mutex;
    mutable std::atomic<bool>                                  m_pendingWrites;
    mutable guarded<std::vector<std::packaged_task<void(T &)>>> m_pendingList;
};
```

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 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>>>;
```

A Better Way . . .

- class guarded<T>
- class shared_guarded<T>
- class ordered_guarded<T>
- class deferred_guarded<T>
- class lr_guarded<T>

class lr_guarded<T>

```
template <typename T, typename Mutex = std::mutex>
class lr_guarded
{
  public:
    using shared_handle = std::unique_ptr<const T, shared_deleter>;

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

    template <typename Func>
    void modify(Func && f);

    ( continued . . . )
```

class lr_guarded<T>

```
// shared access
shared_handle lock_shared() const;
shared_handle try_lock_shared() const;

template <class Duration>
shared_handle try_lock_shared_for(const Duration & duration) const;

template <class TimePoint>
shared_handle try_lock_shared_until(const TimePoint & timepoint) const;
```

(continued . . .)

class lr_guarded<T>

```
  private:
    T                       m_left;
    T                       m_right;
    std::atomic<bool>       m_readingLeft;
    std::atomic<bool>       m_countingLeft;
    mutable std::atomic<int> m_leftReadCount;
    mutable std::atomic<int> m_rightReadCount;
    mutable Mutex           m_writeMutex;
};
```

class lr_guarded<T>

```
template <typename T, typename M>
template <typename Func>
void lr_guarded<T, M>::modify(Func && func)
{
    std::lock_guard<M> lock(m_writeMutex);
    T * firstWriteLocation;
    T * secondWriteLocation;

    bool local_readingLeft = m_readingLeft.load();

    ( continued . . . )
```

class lr_guarded<T>

```
if (local_readingLeft) {
    firstWriteLocation  = &m_right;
    secondWriteLocation = &m_left;
} else {
    firstWriteLocation  = &m_left;
    secondWriteLocation = &m_right;
}

try {
    func(*firstWriteLocation);
} catch (...) {
    *firstWriteLocation = *secondWriteLocation;
    throw;
}
```

class lr_guarded<T>

```
m_readingLeft.store(! local_readingLeft);
bool local_countingLeft = m_countingLeft.load();

if (local_countingLeft) {
    while (m_rightReadCount.load() != 0) {
        std::this_thread::yield(); }
} else {
    while (m_leftReadCount.load() != 0) {
        std::this_thread::yield(); }
}

m_countingLeft.store(! local_countingLeft);
```

class lr_guarded<T>

```
    if (local_countingLeft) {
        while (m_leftReadCount.load() != 0) {
            std::this_thread::yield();  }
    } else {
        while (m_rightReadCount.load() != 0)  {
            std::this_thread::yield();  }
    }
    try {
        func(*secondWriteLocation);
    } catch (...) {
        *secondWriteLocation = *firstWriteLocation;
        throw;
    }
}
```

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);
          });
}
```

A Better Way . . .

- class guarded<T>
- class shared_guarded<T>
- class ordered_guarded<T>
- class deferred_guarded<T>
- class lr_guarded<T>
- class cow_guarded<T>

class cow_guarded<T>

```
template <typename T, typename Mutex = std::mutex>
class cow_guarded
{
  public:
    template <typename... Us>
    cow_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);
```

class cow_guarded<T>

```
    shared_handle lock_shared() const;
    shared_handle try_lock_shared() const;

    template <class Duration>
    shared_handle try_lock_shared_for(const Duration & duration) const;

    template <class TimePoint>
    shared_handle try_lock_shared_until(const TimePoint & timepoint) const;

  private:
    lr_guarded<std::shared_ptr<const T>> m_data;
    Mutex                                m_writeMutex;
};
```

class cow_guarded<T>

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

    deleter(std::unique_lock<Mutex> && lock, cow_guarded & guarded)
        : m_lock(std::move(lock)), m_guarded(guarded), m_cancelled(false)
    {
    }

    void cancel()
    {
        m_cancelled = true;
    }
```

class cow_guarded<T>

```
void operator()(T * ptr)
{
    if (m_cancelled) {
        delete ptr;
    } else if (ptr) {
        std::shared_ptr<const T> newPtr(ptr);

        m_guarded.m_data.modify([newPtr]
              (std::shared_ptr<const T> & ptr)
              { ptr = newPtr; } );
    }

    if (m_lock.owns_lock()) {
        m_lock.unlock();
    }
}
```

class cow_guarded<T>

```
  private:
    std::unique_lock<Mutex> m_lock;
    cow_guarded &           m_guarded;
    bool                    m_cancelled;
};
```

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);
}
```

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

Real World Example - Revisited

- Possible solutions, really
    - CsSignal library was changed to delegate responsibility for thread management to libGuarded
    - valid for the pushButton and the window to both be in their respective destructors concurrently
    - we have potential deadlock

    - solution should be handled in libGuarded and not CsSignal
    - unfortunately the preceding classes did not solve anything

Real World Example - Revisited

- What was still needed in libGuarded
  - a thread aware container
  - writers must not block readers
  - readers do not block at all
  - iterators are not invalidated by writers

libGuarded RCU

- class rcu_guarded<T>
- class rcu_list<T>

RCU

- What is RCU?
    - RCU stands for Read, Copy, Update
    - established algorithm for multithreaded linked lists
    - used internally in Linux
    - writers
        - only one writer at a time (blocking)
    - readers
        - multiple concurrent readers
        - readers are lockless
        - readers do not block writers

RCU

- How does RCU work?
  - uses a linked list where each element is called a node
  - 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

RCU

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

RCU

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

- public API
    - const method, nonblocking, returns a const read_handle
        - lock_read()

    - non-const method, exclusive lock, returns a write_handle
        - lock_write()

RCU

- rcu_list<T, A>
  - container which implements the RCU algorithm

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

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

RCU

- rcu_list<T>::insert()
  - allocate new node
  - initialize the pointers in the new node
    - node->next, node->prev

  - update the next pointer in the node which is before the new node
  - update the prev pointer in the node which is after the new node

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

RCU

- rcu_list<T>::erase()
  - update node_p->next and node_n->prev to skip over current node
  - mark current node deleted
  - add current 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

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;
};
```

RCU

- Zombie list maintenance
  - when rcu_guard::lock_read() is called, an entry is added to the zombie list   (consider this spot z)
  - when the reader completes, rcu_guard begins walking from this saved location (spot z) in an attempt to clean up the zombie list

  - if the end of the zombie list is reached, before another reader type entry is found, then every zombie from (spot z) to the end of the list is safe to delete
  - if another reader type entry is found before reaching the end of the list, the reader entry (spot z) is removed and no other action is taken

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

RCU

- Additional aspects of rcu_list
    - begin() and end() do not return the same data type

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()

Real World Example - Before libGuarded RCU

```
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());
  }
}
```

Real World Example - After libGuarded RCU

```
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;

      }
    }
  }
}
```

Continued . . .

- Really? Who knew there was so much material about multithreading code and libGuarded. Since we probably did not cover all the slides, and there seems to be more questions, we agree to come back another time and continue this discussion.

- In the meantime there are some great videos on our YouTube channel about libGuarded

- Come join us at the Tied House for a beer and we can keep talking about C++

Please subscribe to our Youtube Channel

- Presentations
  - Why DoxyPress
  - Why CopperSpice
  - Compile Time Counter
  - Modern C++ Data Types
  - CsString library
  - Multithreading in C++
  - Build Systems
  - Templates
  - Next video available on Jan 11

https://www.youtube.com/copperspice

Libraries & Applications

- CopperSpice
  - libraries for developing GUI applications
- CsSignal Library
  - standalone thread aware signal / slot library
- CsString Library
  - standalone unicode aware string library
- libGuarded
  - standalone 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

Where to find libGuarded

- www.copperspice.com

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

- source, binaries, documentation files
  - download.copperspice.com

- source code repository
  - github.com/copperspice/libguarded

- discussion
  - forum.copperspice.com