Safety: off
How not to shoot yourself in the foot with C++ atomics

Anthony Williams

Just Software Solutions Ltd
`http://www.justsoftwaresolutions.co.uk`

9th June 2016

Safety: off
How not to shoot yourself in the foot with C++ atomics

Safety: off
How not to shoot yourself in the foot with C++ atomics

- C++ Atomic types and operations
- Worked examples
- Guidelines

Aside: Profiling

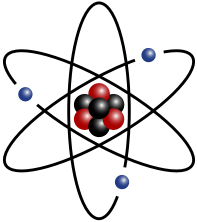
We use atomic operations rather than locks to *improve performance*.

We therefore need to specify the aspect we care about:

- Throughput
- Latency
- Something else

It is vital to profile *before and after* changing to atomic operations

Atomic types



Atomic types

- `std::atomic<T>` provides an atomic type that can store objects of type `T`.
    - `T` can be a built in type, or a class type of any size
    - `T` must be *trivially copyable*
    - `compare_exchange_`*`xxx`* operations require that you can compare `T` objects with `memcmp`
    - `std::atomic<T>` may not be lock free — especially for large types
- `std::atomic_flag` provides a guaranteed-lock-free flag type.
- The Concurrency TS provides `atomic_shared_ptr` and `atomic_weak_ptr`.

atomic

Adjective

Meaning

Of or forming a single irreducible unit or component in a larger system.

Origin

Late 15th century: from Old French **atome**, via Latin from Greek **atomos** ‘indivisible’, based on **a**- ‘not’ + **temnein** ‘to cut’

Atomic Operations

General ops

load(), store(), exchange(),
compare_exchange_weak(),
compare_exchange_strong()
=

Arithmetic ops for atomic<*Integral*> and atomic<T*>

fetch_add(), fetch_sub()
++, --, +=, -=

Bitwise ops for atomic<*Integral*>

fetch_and(), fetch_or(), fetch_xor()
&=, |=, ^=

Flag ops for atomic_flag

test_and_set(), clear()

Atomic Operations

General ops

`load()`, `store()`, `exchange()`,
`compare_exchange_weak()`,
`compare_exchange_strong()`
`=`

Arithmetic ops for `atomic<Integral>` and `atomic<T*>`

`fetch_add()`, `fetch_sub()`
`++`, `--`, `+=`, `-=`

Bitwise ops for `atomic<Integral>`

`fetch_and()`, `fetch_or()`, `fetch_xor()`
`&=`, `|=`, `^=`

MAY NOT BE LOCK FREE

Flag ops for `atomic_flag`

`test_and_set()`, `clear()`

GUARANTEED LOCK FREE

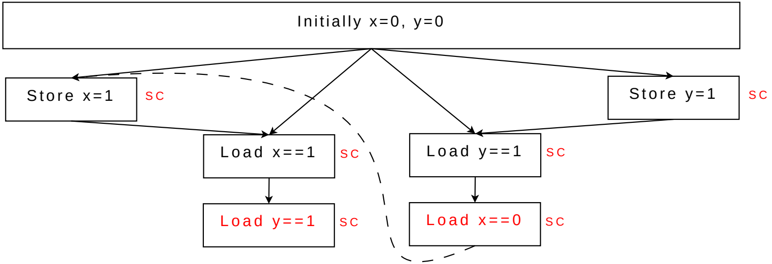
Memory Ordering Constraints

6 values for the ordering on an operation:

- `memory_order_seq_cst` (the default)
- `memory_order_acquire`
- `memory_order_release`
- `memory_order_acq_rel` (RMW ops only)
- `memory_order_relaxed` (Experts only)
- `memory_order_consume` (Optimized form of `memory_order_acquire`, for special circumstances, for experts only)

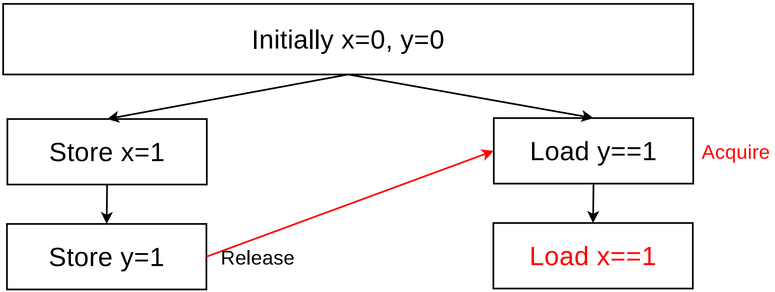
memory_order_seq_cst ordering

All memory_order_seq_cst operations to all variables form a **single total order.**



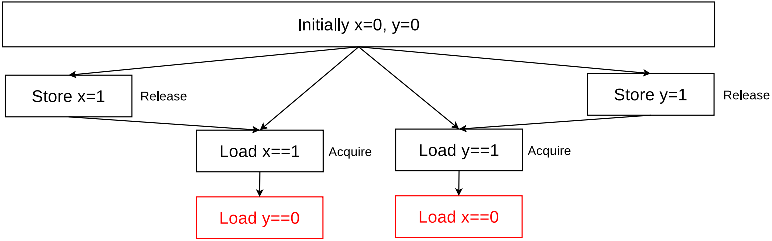
Release/acquire synchronization

A `memory_order_release` operation *synchronizes with* a `memory_order_acquire` operation that reads the value written.



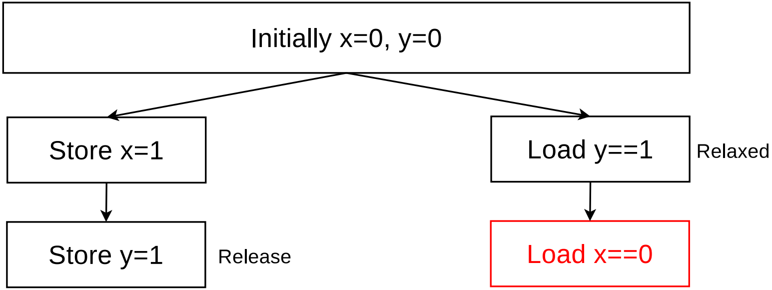
Release/acquire non-synchronization

Unrelated reads do not synchronize.



Relaxed atomics: anything can happen

Relaxed atomics can read out of order.



Fences

Fences

C++ has two kinds of fences:

- `std::atomic_thread_fence`
  $\Rightarrow$ Used for synchronizing between threads
- `std::atomic_signal_fence`
  $\Rightarrow$ Used for synchronizing between a thread and a signal handler in that thread

Fences

Fences in C++ effectively modify the ordering constraints on neighbouring atomic operations rather than providing any direct ordering constraints themselves.

```
x.load(memory_order_relaxed);
atomic_thread_fence(memory_order_acquire);
```

$\Rightarrow$ `x.load(memory_order_acquire);`

```
atomic_thread_fence(memory_order_release);
x.store(memory_order_relaxed);
```

$\Rightarrow$ `x.store(memory_order_release);`

Fences

`memory_order_acq_rel` fences behave as both `memory_order_acquire` and `memory_order_release` fences.

`memory_order_seq_cst` fences are special: they form part of the total order of `memory_order_seq_cst` operations, and can therefore enforce orderings beyond the direct pairwise acquire-release orderings. **If you're relying on this, you've probably done something wrong.**

Lock-free examples

Lock-free terminology

Obstruction free (*Weakest guarantee*)

If all other threads are paused then any given thread will complete its operation in a bounded number of steps.

Lock free (*Most common guarantee*)

If multiple threads are operating on a data structure then after a bounded number of steps **one** of them will complete its operation.

Wait free (*Strongest guarantee*)

Every thread operating on a data structure will complete its operation in a bounded number of steps, even if other threads are also operating on the data structure.

Queues

Why Queues?

- Core facility for communication between threads
- Many types of queue:
  - SPSC / MPSC / MPMC / SPMC
  - bounded / unbounded
  - FIFO / priority / unordered
  - intrusive / non-intrusive
- Good for demonstrating issues

Lock-based queue

Lock-based, unbounded, MPMC, FIFO queue

```
template<typename T>
class queue1{
private:
  std::mutex m;
  std::condition_variable c;
  std::queue<T> q;
};
```

Push

```
void push_back(T t){
  {
    std::lock_guard<std::mutex> guard(m);
    q.push(t);
  }
  c.notify_one();
}
```

Pop

```
T pop_front(){
  std::unique_lock<std::mutex> guard(m);
  c.wait(guard,[=]{return !q.empty();});
  auto ret=q.front();
  q.pop();
  return ret;
}
```

Lock-free SPSC FIFO queue

Let's start simple with our lock-free queue:

- One producer thread
- One consumer thread
- Bounded, so no memory allocation
- Assume `T` has a `noexcept` copy constructor

Lock-free SPSC FIFO queue: bounded buffer

```
template<typename T,unsigned buffer_size=42>
class queue2{
  typedef typename std::aligned_storage<
    sizeof(T),alignof(T)>::type storage_type;
  struct entry{
    std::atomic<bool> initialized{false};
    storage_type storage;
  };
  entry buffer[buffer_size];
};
```

Lock-free SPSC FIFO queue: pushing

```
template<typename T,unsigned buffer_size=42>
class queue2{
  unsigned push_pos{0};
public:
  void push_back(T t){
    unsigned my_pos=push_pos;
    auto& my_entry=buffer[my_pos];
    if(my_entry.initialized.load())
      throw std::runtime_error("Full");
    push_pos=(my_pos+1)%buffer_size;
    new(&my_entry.storage) T(t);
    my_entry.initialized.store(true);
  }
};
```

Aside: avoid busy waits

Busy waits are to be avoided: they consume processor power for no purpose.

It is acceptable for a `compare_exchange_weak` loop to have no body: we're hoping to avoid spinning more than a couple of times.

If you need to wait, use a proper wait mechanism such as `std::condition_variable`.

Lock-free SPSC FIFO queue: popping

```
    unsigned pop_pos{0};
  public:
    T pop_front(){
      if(!buffer[pop_pos].initialized.load())
        throw std::runtime_error("empty");
      auto ptr=static_cast<T*>(
        static_cast<void*>(&buffer[pop_pos].storage))
      auto ret=*ptr;
      ptr->~T();
      buffer[pop_pos].initialized.store(false);
      pop_pos=(pop_pos+1)%buffer_size;
      return ret;
    }
```

Broken Lock-free MPSC FIFO queue

Now let's try and make an MPSC FIFO based on `queue2`.
A naive attempt would be to make `push_pos` atomic:

```
std::atomic<unsigned> push_pos{0};

void push_back(T t){
  unsigned my_pos=push_pos.load();
  while(!push_pos.compare_exchange_weak(
          my_pos,(my_pos+1)%buffer_size)){}
```

Broken Lock-free MPSC FIFO queue

Now let's try and make an MPSC FIFO based on `queue2`.
A naive attempt would be to make `push_pos` atomic:

```
std::atomic<unsigned> push_pos{0};

void push_back(T t){
  unsigned my_pos=push_pos.load();
  while(!push_pos.compare_exchange_weak(
          my_pos,(my_pos+1)%buffer_size)){}
```

This is still broken.

Broken Lock-free MPSC FIFO queue

1. Queue is empty, `push_pos` is 0.
2. Thread 1 calls `push_back`, gets `my_pos` is 0, and increments `push_pos` to 1.
3. Thread 1 checks the cell is empty.
4. Thread 1 gets suspended by scheduler
5. Thread 2 calls `push_back` `buffer_size`-1 times, so `push_pos` loops round to 0.
6. Thread 2 calls `push_back` again. Thread 2 gets `my_pos` of 0, and sets `push_pos` to 1.
7. Thread 2 checks that the cell is empty.
8. Thread 2 populates the cell.
9. Thread 1 is woken by the scheduler.
10. Thread 1 populates the cell. **DATA RACE**.

Not-lock-free MPSC queue

The problem on the previous slide only occurs if the buffer is full. Can we prevent this by checking for a full buffer?

```
std::atomic<unsigned> size{0};

void push_back(T t){
  unsigned old_size=size.load();
  for(;;){
    if(old_size==buffer_size)
      old_size=size.load();
    else if(size.compare_exchange_weak(
              old_size,old_size+1))
      break;
  }
```

Not-lock-free MPSC queue

Our queue is now not even obstruction free.

1. Queue is empty. `push_pos` is 0. `pop_pos` is 0.
2. Thread 1 calls `push_back` and increases size.
3. Thread 1 gets `my_pos` as 0, increments `push_pos`
4. Thread 1 is suspended by scheduler.
5. Thread 2 pushes `buffer_size`-1 entries.
6. Thread 2 tries to push another entry, but `size==buffer_size`
7. Thread 3 calls `pop_front`, but `pop_pos` is 0 and the entry at 0 hasn't been filled in.
8. **All threads now stalled waiting for thread 1.**

Fixing queue4

Can we fix this? First we need to identify the problem.

Pushing a value consists of 3 steps:

1. Find a free slot in the buffer
2. Construct the pushed value in the slot
3. Mark the value as available to the consuming thread

Fixing queue4

Can we fix this? First we need to identify the problem.

Pushing a value consists of 3 steps:

1. Find a free slot in the buffer
2. Construct the pushed value in the slot
3. Mark the value as available to the consuming thread

We need to publish in step 3, rather than step 1.

Fixing queue4

We need to separate the buffer ordering from the queue ordering, so we need to redo steps 1 and 3.

1. **Hunt the buffer for a free slot**
2. Construct the pushed value in the slot
3. **Link that entry into the queue**

Fixing queue4: Linking entries into the queue

Let's use a linked list — that's easy, isn't it? Just push entries on the tail, and pop them off the head.

We still have two locations to update: the `next` pointer in the previous node, and the `tail` pointer.

Having the push thread do them in either order can lead to a race.

Fixing queue4: Linking entries into the queue

Answer: update the `next` pointers from the (one and only) pop thread.

In `push_back` we record the **previous** tail entry:

```
void push_back(T t){
  auto my_entry=allocate_entry();
  new(&my_entry->storage) T(t);
  my_entry->next=nullptr;
  my_entry->prev=tail.load();
  while(!tail.compare_exchange_weak(
          my_entry->prev,my_entry)){}
}
```

Fixing queue4: Linking entries into the queue

In `pop_front`, if there is no `next` value for the current entry we can start at the `tail` and fill them all in:

```
T pop_front(){
  entry* old_head=head;
  while(!old_head)
    old_head=chase_tail();
  head=old_head->next;
  auto ptr=static_cast<T*>(
    static_cast<void*>(&old_head->storage));
  auto ret=*ptr;
  ptr->~T();
  recycle_node(old_head);
  return ret;
}
~queue5(){
```

Fixing queue4: Linking entries into the queue

```
entry* chase_tail(){
  entry* next=tail.exchange(nullptr);
  if(!next)
    return nullptr;

  while(next->prev){
    next->prev->next=next;
    next=next->prev;
  }
  return next;
}
```

A lock-free? MPSC FIFO queue

Our queue is now **obstruction free**, but is it **lock-free** or **wait-free**?

- If the queue is full then we have to wait.
  $\Rightarrow$ Use a lock-free allocator instead of a fixed buffer.
- If the queue is empty then we have to wait.
- Otherwise, only waiting is in compare-exchange loops
  $\Rightarrow$ No upper limit on loops, so cannot be wait-free.
- `compare_exchange_weak` can fail spuriously
  $\Rightarrow$ If it does then there is no bound to the number of steps.

Lock-free vs obstruction-free strictly depends on the `compare_exchange_weak` implementation.

Performance: Cache Ping-Pong

Performance: Cache Ping-Pong

Cache Ping-Pong is where a cacheline is continuously shuttled back and forth between two processors. This occurs when two threads are accessing either:

- **the same** atomic variable
- **different** variables on **the same cache line**

This can have a **big** performance impact, because transferring cache lines is **slow**.

Cache Ping-Pong in `queue5`

`queue5` can be accessed by many threads in `push_back`, and one more thread in `pop_front` simultaneously.

```
std::atomic<unsigned> push_hint{0};
entry* head{nullptr};
std::atomic<entry*> tail{nullptr};
entry buffer[buffer_size];
```

`head` and `tail` are adjacent, but accessed by different threads
⇒ unnecessary cache ping-pong.
There are many examples in this data structure.

Cache Ping-Pong avoidance in `queue5`

The solution to cache ping-pong is to put data on different cache lines by adding padding. **This trades memory space for performance.**

```
std::atomic<unsigned> push_hint{0};
char padding1[padding_size];
entry* head{nullptr};
char padding2[padding_size];
std::atomic<entry*> tail{nullptr};
char padding3[padding_size];
entry buffer[buffer_size];
```

Cache Ping-Pong avoidance in `queue5`

Times for 10,000,000 pushes of an integer on each of 3 threads, with another thread popping all 30,000,000 entries.

Run	No padding	With padding	With Lock
1	26.4s	11.4s	27.1s
2	22.4s	9.8s	17.8s
3	22.1s	15.4s	25.4s
4	14.3s	9.0s	24.3s
Mean	**21.3s**	**11.4s**	**23.7s**

Performance: Memory Ordering Constraints

All the examples so far have used the default ordering constraint: `memory_order_seq_cst`.

You should use `memory_order_seq_cst` unless you have a strong reason not to.

Performance: Memory Ordering Constraints

For x86, only `store` is affected by the memory order, but for architectures like POWER and ARM with weaker default synchronization, all operations can be affected.

You **must** test on a weakly-ordered system like POWER or ARM if you're using anything other than `memory_order_seq_cst`.

Stacks

Stacks

A stack is a simpler data structure than a queue. It's great for examples, but bad for real use, as all threads are contending to access the top-of-stack.

I'm going to use it to demonstrate a specific problem: the **A-B-A** problem.

A simple MPSC stack: pushing

```
template<typename T>
class stack1{
  struct node{
    T  val;
    node* next;
  };
  std::atomic<node*> head{nullptr};
public:
  void push(T newval){
    auto newnode=new node{newval,head.load()};
    while(!head.compare_exchange_weak(
            newnode->next,newnode)){}
  }
```

A simple MPSC stack: popping

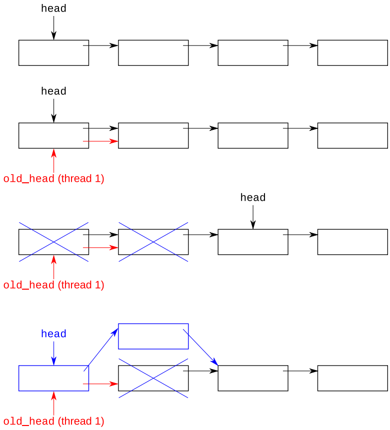
```
T pop(){
  auto old_head=head.load();
  for(;;){
    if(!old_head)
      old_head=head.load();
    else if(head.compare_exchange_strong(
            old_head,old_head->next)){
      auto res=old_head->val;
      delete old_head;
      return res;
    }
  }
}
```

A simple stack: A-B-A issues

Why is this a single-consumer stack?
Answer: the A-B-A problem.

A simple stack: A-B-A issues

1. Thread 1 calls `pop()`
2. Thread 1 reads `head` into `old_head` **(A)**
3. Thread 1 reads `old_head->next`
4. Thread 1 is suspended
5. Thread 2 pops two items, `head` has new value (**B**)
6. Thread 2 pushes two items
7. **Second new item is given address of old item**, `head` has original value (**A**)
8. Thread 1 resumes and calls `compare_exchange_strong`, which **succeeds because the address is the same**
9. **Stack is now corrupt**



A-B-A

The setup:

1. A value changes from A to B and back to A,
2. Other aspects of the data structure have changed, and
3. A thread makes a change based on the first time the value was A that is inconsistent with the new state of the data structure.

This most commonly happens where the value is a pointer.

A-B-A: Solutions

Do not allow a variable to return to its previous value while a thread can do something based on the old value.

- Use a change count as part of the variable:

```
struct Value{ T* ptr; unsigned count;};
std::atomic<Value> v;
```

- Ensure that objects are not recycled when still accessible, so A-B-A never happens.
  $\Rightarrow$ Reference count the objects, e.g. with `std::shared_ptr` and `atomic_shared_ptr` or use hazard pointers, or something similar.

Guidelines

Guidelines

- Don’t use atomics unless you have to
- Profile before and after
- Test on a weakly-ordered architecture such as POWER or ARM
- Don’t use atomics unless you **really** have to

Guidelines

Think in transactions
: Do work off to the side and commit with a single atomic operation.

Split big operations
: If the operation is too big to do in one step, split it into smaller steps **that retain the data structure invariants**.

Limit use cases
: Restrict the permitted concurrency levels where possible to reduce implementation complexity.

Watch out for ABA problems
: These require the circumstances to align just so, but will destroy your data structure when they happen. They can be easily missed in testing.

Guidelines

Avoid cache ping pong
: Add padding between variables that are accessed from different threads. Try and avoid too many threads accessing the same variable.

Stick to `memory_order_seq_cst`
: Unless you **really** know what you're doing, and **really** need the performance gain, stick to the default `memory_order_seq_cst`. Anything else can be a nightmare to prove correct.

Package things up
: Wrap atomic operations with types that only expose the desired functionality, to clarify the user code and hide the complexity.

Guidelines

Aim for lock-free

Aim for your code to be at least *obstruction-free*, and preferably *lock-free*. Leave *wait-free* for those rare circumstances where you need it.

Avoid busy waits

If you're actually waiting (as opposed to spinning on a `compare_exchange_weak` operation), use a proper wait mechanism.

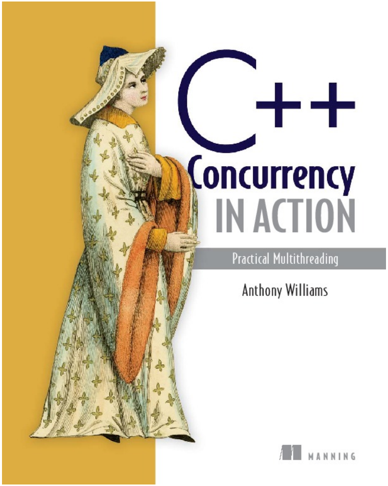
Questions?

Just::Thread

`just::thread` provides a complete implementation of the C++14 thread library for MSVC and g++ on Windows, and g++ for Linux and MacOSX.

`Just::Thread` Pro gives you actors, concurrent hash maps, concurrent queues and synchronized values.

My Book



C++ Concurrency in Action:
Practical Multithreading

`http://stdthread.com/book`

Picture credits

The images listed below are from the specified source, with the specified license. All other images are copyright Just Software Solutions Ltd, licensed under Creative Commons Attribution ShareAlike 4 `https://creativecommons.org/licenses/by-sa/4.0/`.

- Safety Switch: `https://www.flickr.com/photos/jamescridland/6163838972/` by James Cridland, Creative Commons Attribution
- Stop watch: `https://www.flickr.com/photos/o5com/5488964999/` by o5com, Creative Commons Attribution-NoDerivs
- Lithium Atom: `https://commons.wikimedia.org/wiki/File:Stylised_Lithium_Atom.svg` by Indolences and Rainer Klute, Creative Commons Attribution-ShareAlike
- Fence: `http://www.public-domain-image.com/full-image/nature-landscapes-public-domain-images-pictures/sunshine-public-domain-images-pictures/sunlight-over-picket-fence.jpg-royalty-free-stock-photograph.html` by Leon Brooks, Public Domain
- Lock image: `http://pixabay.com/en/lock-closed-shut-keyhole-306311/` by Nemo, Public Domain
- "Unauthorised" overlay: `http://pixabay.com/en/unauthorised-denied-ban-prohibition-156169/` by OpenClips, Public Domain
- Queue for Apple Store: `http://www.geograph.org.uk/photo/3143246` by Robin Stott, Creative Commons
- Ping Pong Set: `https://commons.wikimedia.org/wiki/File:Ping-Pong_2.jpg` by Daniel Schwen, Creative Commons Attribution-ShareAlike
- Stack of presents: `http://christmasstockimages.com/free/ideas_concepts/slides/christmas_gift_stack.htm` by christmasstockimages.com, Creative Commons Attribution
- Abacus: `https://commons.wikimedia.org/wiki/File:Abacus_6.jpg` by Loadmaster (David R. Tribble), Creative Commons Attribution ShareAlike
- Success sign: `https://commons.wikimedia.org/wiki/File:Success_sign.jpg` by Keith Ramsey (RambergMediaImages), Creative Commons Attribution ShareAlike