Lock-Free Single-Producer - Single Consumer Circular Queue

KjellKod.cc

31 Dec 2014 Public Domain

Rate me: ★★★★★ 4.84/5 (58 votes)

How to make a wait-free, lock-free CircularFifo using C++11.

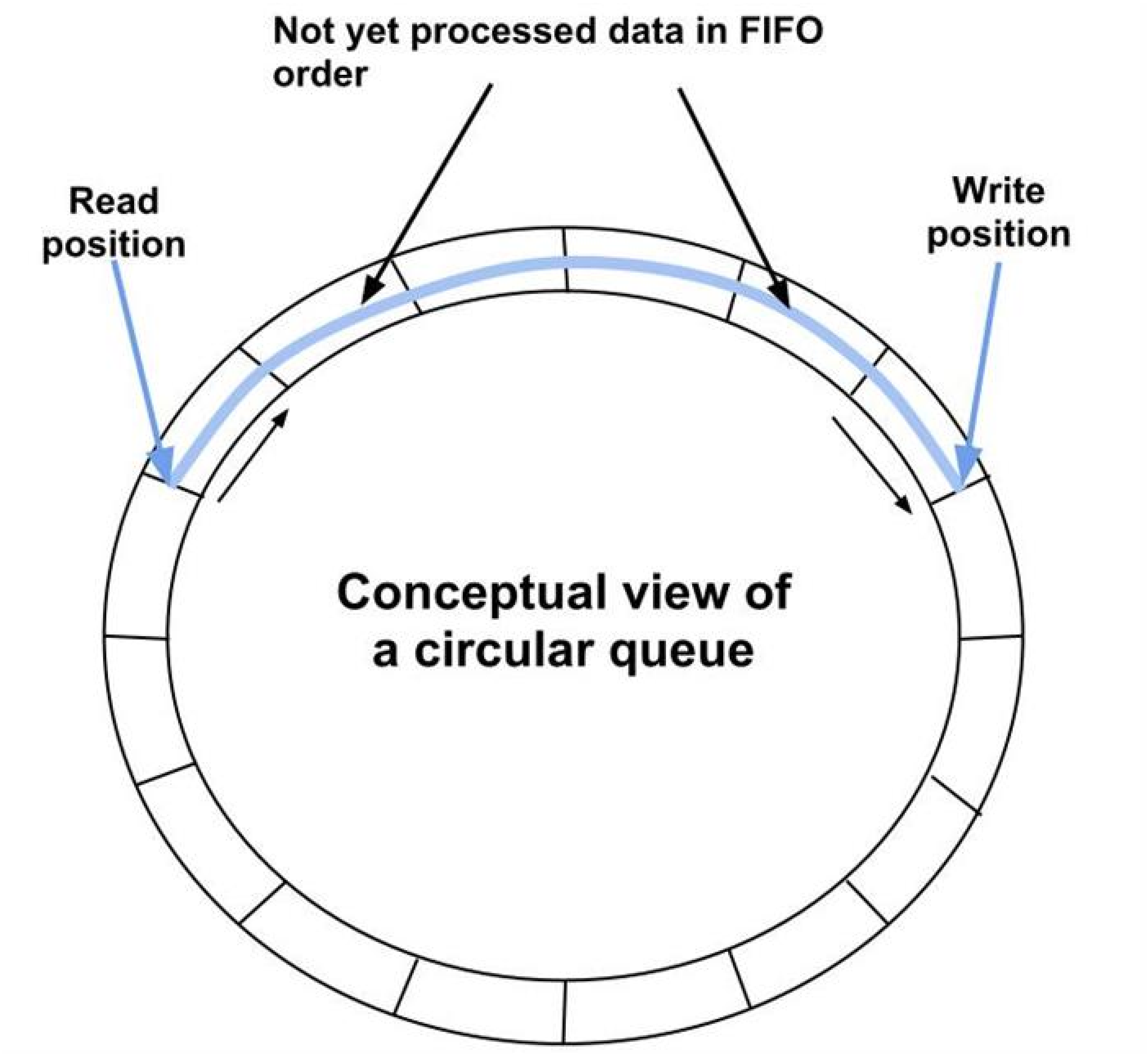
Download source code

Latest lock-free-wait-free-circularfifo from BitBucket

Introduction

The wait-free and lock-free circular queue is a useful technique for time and memory sensitive systems. The wait-free nature of the queue gives a fixed number of steps for each operation. The lock-free nature of the queue enables two thread communication from a single source thread (the Producer) to a single destination thread (the Consumer) without using any locks.

The power of wait-free and lock-free together makes this type of circular queue attractive in a range of areas, from interrupt and signal handlers to real-time systems or other time sensitive software.


Not yet processed data in FIFO order
Read position
Write position
Conceptual view of a circular queue

Background: Circular FIFO Queue

This circular FIFO queue can be used for communicating between **maximum two threads**. This scenario is often referred to as the single producer → single consumer problem.

The problem description: A high priority task that must not be delayed in lengthy operations and also requires the data processing to finish in the same order as they are started. Examples could be : Interrupt/Signal handler, GUI events, data recorder, etc.

The solution: Delegate time consuming jobs to another thread that receives (i.e. *consumes*) the produced message. Using a `FIFO` queue gives predictable behaviour for the **asynchronous** delegation and processing of tasks as they pass from the *Producer* to the *Consumer*.

The Producer can add new tasks to the queue as long as the queue is not full. The Consumer can remove tasks from the queue as long as the queue is not empty. At the queue there is **no waiting involved** for adding or retrieving data. If there are no jobs to process the *Consumer* thread can continue with other tasks. If there is no room left in the queue for the *Producer* it is not required to wait until there is space.

Obviously a full queue for a *Producer* is a non-ideal situation. Its maximum size and how that relates to the negative impact of dealing with a full queue must be carefully considered. Issues that deals with this, such as overwrite of old items, multiple priority queues, or other ways of handling this scenario are outside the scope of this article.

Background: Article Rationale

Back in 2009 I published my first CodeProject article about a lock-free circular FIFO queue. In the military avionic industry I had seen what was in my view a *hazardous* technique for creating a lock-free circular FIFO. The queue was used for an interrupt handler as it pushed incoming messages from the interrupt handling to the real-time system.

If reading the original version of this article (also published here!) there will be a description and reasoning of the *how, why and when* a hazardous technique **could possibly work** and when it **would not work**. The old article was written to try to understand this clearly discouraged technique but without recommending it.

The old article described a type of thread communication that goes against good `C++` practice using a circular FIFO queue that is highly platform dependent. It breaks easily as it is basically an ugly platform hack that looks appealing through an illusion of simplicity. It could break for something as simple as a compiler update or using it on another hardware architecture.

In the old article I gave a promise to publish a C++11 empowered queue that is lock-free through the use of `std::atomic`. **That time has come now**. `C++11` with its memory model, is available both on Linux and Windows. Using the `std::atomic` from the `‹atomic›` library effectively removes the need to ever again being tempted to use discouraged platform dependent hacks.

The aim with the article is not only to show two wait-free, lock-free CircularFifo queues. The article is also explaining the rules governing the `C++11` memory models. After reading this article you should have an understanding of some of the fundamentals behind the `‹atomic›` library.

Contents

For those of you who want to read up on the inner workings of the Circular Queue, I have provided a short description of it and my implementation in Part I. For those of you who are already familiar with it can shorten this read by going to Part II. Part II describes briefly the `C++11` different memory models and how they with `std::atomic` are used to create the lock-free aspect of these circular queues.

Part I

Part II

Disclaimer

Lock-free, really? But ...
The C++11 standard guarantees that the `std::atomic_flag` is lock-free. In fact the `std::atomic_flag` is the **only** type that is guaranteed to be lock-free. It is possible that some platform/library

implementations will use locks internally for the native atomic types. However, although this is system dependent it is **likely** that many native atomic types are lock-free.

The `CircularFifo` provides a function `bool isLockFree() const {...}` that can be called to see if it is truly lock-free, even behind the scenes. ´

But what about ...
Just like in the old CodeProject article this text is written to be simple and easy to understand. All the nitty-gritty details that an optimal solution might have are not covered. These details are usually in the area of:

- Consumer/Producer handling for empty/full queues.
- Handling of variable sized items

~~Multiple~~ Producers, ~~Multiple~~ Consumers
Never use multiples of either Consumer or Producer for this queue. Using more Producer or Consumer threads breaks this design and corrupts the queue. This queue must only be used in a **single Producer** and **single Consumer** scenario. If more threads are needed, then a different solution should be used.

Nothing *New*!
This article **does not claim** to introduce a new concept, or a new way of writing this type of lock-free circular queue. On the contrary, this article presents as far as I understand a well known, widely accepted way of writing such a queue. The resulting code is in fact remarkably similar to the old code, almost identical. Comparing the code might give an illusion of only trivial code changes but the impact of the changes is enormous. The real difference is the atomic declaration and handling. This is of course a huge difference since there are lots of things happening behind the scenes

The old code should be considered *broken* and where the old code is not guaranteed to work by any standard the code presented here should work according to the new `C++11` standard [2].

Public Domain
Use the code on your own risk. I am using it myself of course but because of the *Public Domain* status I will not give any guarantees. Nor will I give any promises in case of bugs that might have slipped me by. Using the code or not is your own responsibility. Using the code can be done freely with no obligations to me or towards a license. Modify and use the code to your hearts content.

Tail first or Head first?
Should items be added at the tail and removed at the head, or is the opposite more logical? The opinions differ. This article uses the methodology of "*items are added at the tail, removed from the head*" to be consistent with the old article and to make it similar with the common and well-known `pop_front` and `push_back` queue operations.

Memory model: sequential or relaxed/acquire/release?

Two versions of the wait and lock-free circular FIFO are presented. The first, most intuitive, use C++11 default memory-ordering `memory_order_seq_cst`. The other, somewhat more complicated, use `memory_order_relaxed`, `memory_order_acquire` and `memory_order_release`. The use of the memory orders will be explained and I will also try to show that the default `memory_order_seq_cst` is not only the easiest to reason about but also has very good performance on x86-x64 architectures.

Part I

Using the code

Please remember to use only **one** thread as the Producer and only **one** thread as the Consumer. If more threads are involved on either side it will break the thread-safe design and corrupt the Producer → Consumer communication.

It could not be any simpler to use the code.

C++

```cpp
CircularFifo‹Message, 128› queue;     // 128 'Messages' will fit

/* Producer thread */
Message m = ...
if ( false == queue.push(m)) { /*false equals full queue */ }


/* Consumer thread */
Message m;
if ( false == queue.pop(m)) { /* false equals empty queue */ }
```

You can find more examples in the unit tests that come with the code.

Code Explained: Circularfifo‹Type, Size›

Let us go straight to the sequential consistent code without further ado. The code is slightly simplified to increase article readability. If you prefer the full syntax you could open up the code and view it side-by-side.

C++

```cpp
template‹typename Element, size_t Size›
class CircularFifo{
public:
  enum { Capacity = Size+1 };

  CircularFifo() : _tail(0), _head(0){}
  virtual ~CircularFifo() {}

  bool push(const Element& item);
  bool pop(Element& item);

  bool wasEmpty() const;
  bool wasFull() const;
  bool isLockFree() const;

private:
  size_t increment(size_t idx) const;

  std::atomic<size_t>  _tail;
  Element              _array[Capacity];
  std::atomic<size_t>  _head;
};
```

How it works

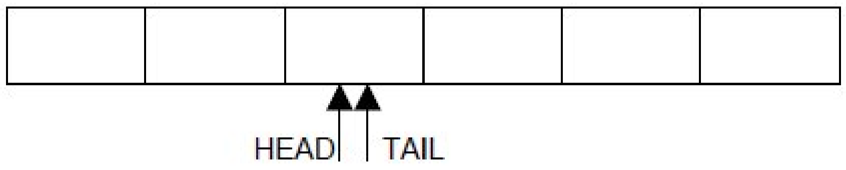
The first described queue uses the **default, sequential** atomic operations. The sequential model makes it easy to reason about the atomics and how they relate in respect to atomic operations in other threads. On the strong hardware memory (e.g. processor) architectures that `x86` and `x64` have it gives some performance overhead when using the *sequential atomic* `CircularFifo` compared to the later discussed `CircularFifo` that uses fine grained `std::memory_order_`{`relaxed/acquire/release`}.

On a weakly ordered memory architecture the performance difference between the queues would be more significant.

Push and Pop

`bool push(Item&)` will be done by a Producer thread. The `bool pop(Item&)` will be used by the Consumer thread.

Empty



When the buffer is **empty**, both indexes will be the same. Any reads by the Consumer will fail.

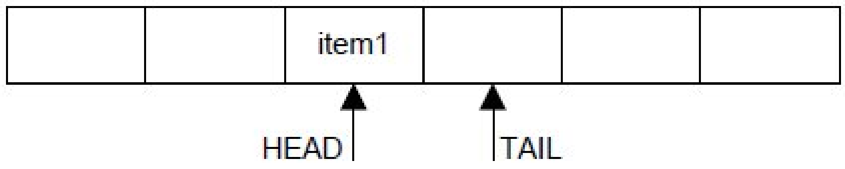
C++

```
bool wasEmpty() const
{
  return (_head.load() == _tail.load());
}
```

Any attempts by the Consumer to retrieve, `bool pop(Item&)`, when the queue is empty will **fail**. Checking if the queue is empty with `wasEmpty()` will give a snapshot of the queue status. Of course the *empty* status might change before the reader has even acted on it. This is OK and fine, it should be treated as a snapshot and nothing else.

Producer adds items

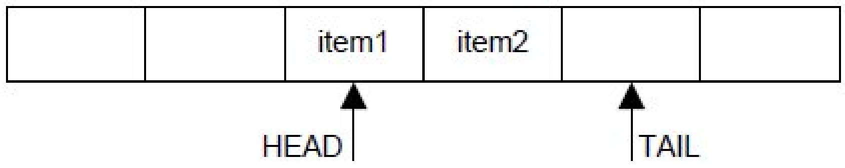
The Producer adds, `push()`, a new item at the position indexed by the tail. After writing, the tail is incremented one step, or wrapped to the beginning if at end of the queue. The queue grows with the tail.



C++

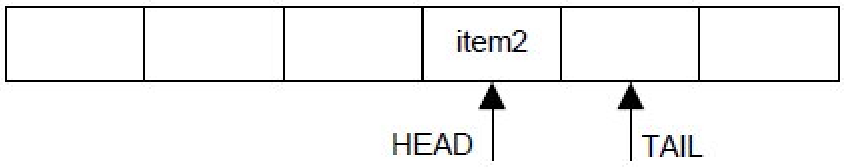
```cpp
/* Producer only: updates tail index after setting the element in place */
bool push(Element& item_)
{
  auto current_tail = _tail.load();
  auto next_tail = increment(current_tail);
  if(next_tail != _head.load())
  {
    _array[current_tail] = item;
    _tail.store(next_tail);
    return true;
  }

  return false;  // full queue
}
```

The Producer adds another item and the tail is incremented again.



Consumer retrieves item

The Consumer retrieves, `pop()`, the item indexed by the head. The head is moved toward the tail as it is incremented one step. The queue shrinks with the head.

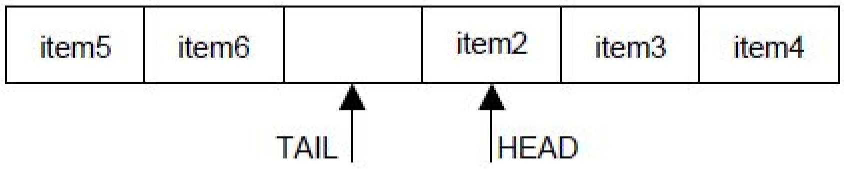


C++

```cpp
/* Consumer only: updates head index after retrieving the element */
bool pop(Element& item)
{
  const auto current_head = _head.load();
  if(current_head == _tail.load())
    return false;   // empty queue

  item = _array[current_head];
  _head.store(increment(current_head));
  return true;
}
```

Full

If the Producer pushes more items onto the queue than the Consumer can keep up with, the queue will eventually become full.



When the queue is full, there will be a one slot difference between head and tail. At this point, any writes by the Producer will fail. Yes it **must** even fail since otherwise the **empty queue** criterion i.e. `head == tail` would come true.

C++

```cpp
bool wasFull() const
{
  const auto next_tail = increment(_tail.load());
  return (next_tail == _head.load());
}


size_t increment(size_t idx) const
{
  return (idx + 1) % Capacity;
}
```

% modulus operator

The % modulus operator can be confusing if not encountered before. The `% operator` gives the remainder from integer division. It can be used in a range of ways and here it is used to calculate when the index should be wrapped back to the beginning in a *circular* way.

C++

```
 // Example: A CircularFifo with Size of 99
 // Capacity is Size+1. The padding of +1 is used to simplify the logic
 increment(98) ---> will yield 99
 increment(99) ---> will yield 0
}
```

When the `next_tail` showed in `push()` is one larger then the queue's size it wraps back to the beginning, i.e., `index 0`.

Thread Safe Use and Implementation

The thread safety is ensured through the class design and its intended use. Only two functions can modify the state of the queue, `push()` and `pop()`. Only one thread should be allowed to touch `push()`. Only one thread should be allowed to touch `pop()`. Thread safety is guaranteed in this way when `push()` and `pop()` are used by a at most **one** corresponding **thread**. The Producer thread should only access `push()` and the Consumer thread only `pop()`.

The design can be further emphasized with access through interfaces but for this article I think it is enough to clarify it here and in the comments. For a real-world project it *might* make sense to use something like `SingleConsumerInputQueue` and `SingleProducerOutputQueue` interfaces to decrease the risk for *error by confusion*.

Atomic access when used the intended way
Producer: `push()` writes only to the `tail`, but reads a `head` snapshot to verify queue is not full.
Consumer: `pop()` writes only to the `head`, but reads a `tail` snapshot to verify that the queue is not empty. Only the Producer thread will update `tail` and only the Consumer thread will update head.

However reading and writing of both tail and head must be thread safe in the sense that we do not want to read old, cached values and both reads and writes must be atomic. As explained below the update of `tail` and `head` indexes are atomic. The atomic read/write together with this class design will ensure thread safety for at most one Producer thread and one Consumer thread.

Important: The text further down explains how atomic, non-reordered reads and writes are achieved for this Circular Queue. It is probably the most important part of this article. It explains both the `sequential` and the `acquire-release/relaxed models`.

Part II

Making It Work

It is imperative that the head and tail indexes must be updated only **after** the element (`pop()` or `push()`) is read or written. Thus, any updates to the `head` or `tail` will ensure proper access to the elements. For this to work, the read and writes of the `head` and `tail` must be atomic and must not be re-ordered.

Atomic operations and the different Memory Models

Below is described a subset of the functionality that you can get from the different C++11 memory models and atomic operations. It is in no way comprehensive but contains the necessary pieces to explain both types of the C++11 empowered wait-free, lock-free circular queue.

There are **three different memory-ordering models** in C++11 and with them six ordering options.

1. Sequentially consistent model
    - `memory_order_seq_cst`
2. acquire-release model
    - `memory_order_consume`
    - `memory_order_acquire`
    - `memory_order_release`
    - `memory_order_acq_rel`
3. relaxed model
    - `memory_order_relaxed`

Sequential Consistent

The default, sequential memory ordering is used in the code example above. This atomic, sequentially consistent memory model is probably what most of us are comfortable with when reasoning about possibly sequences of threaded atomic operations.

Rules for Sequential Consistent Memory Model and Operations

1. A sequential consistent store will be synchronized with a sequential consistent load of the same data. It is guaranteed thanks to a global synchronization between involved threads.

2. All threads participating in these sequential atomic operations will see exactly the same order of the operations.

3. Atomic operations within the same thread cannot be reordered.

4. A seqence of atomic operations in one thread will have the same sequence of order seen by other threads

Sequential Consistent Atmic Code

These rules give that the sequential atomic operations used in `push()` and `pop()` are safe. Let us recap with `pop()` to demonstrate this:

C++

```
/* Consumer only: updates head index after retrieving the element */
bool pop(Element& item)
{
  const auto current_head = _head.load();  // 1. loads sequentially the head
  if(current_head == _tail.load())         // 2. compares head to snapshot of tail
    return false;    // empty queue

  item = _array[current_head];             // 3. retrievs the item pointed to by head
  _head.store(increment(current_head));    // 4. update the head index to new
position
  return true;
}
```

The *atomic sequential* rules guarantee that step 3-4 will not be reordered (*reordering would however not be harmful*). The *atomic sequential rules* guarantee also that the *Consumer* thread will not get out-of-order updates when reading the tail snapshot with `head.load()`

It is with this sequential memory model you can get this warm fuzzy feeling in your stomach. It seems easy to reason about, right?

Caveats with the Sequential Memory Model

Caveat 1:
Non-sequential, relaxed memory atomic operations might not see the same ordering of operations as the sequential, atomic, operations. To get the power of *easy-to-reason-about* ordering of operations the sequentially consistent memory ordering should be used for all atomic operations that are interacting.

Caveat 2:
On the x86-x64 processor architecture which use a strong hardware memory model [6] the sequential consistent model is relatively cheap. On a weakly-ordered hardware memory architecture [7] the sequential consistent model is more expensive as it requires more internal synchronization operations. On such weakly-ordered hardware memory architecture the sequential consistent model will have a significant performance penalty due to much higher synchronization overhead then the *acquire-release* memory model.

Simplified Atomic Sequential operations

One last thing on the sequential atomic operations: Not only are they easy to reason about they are also easier to write. Since they are the default choice it is not necessary to write explicitly the memory operation type.

C++

```
bool CircularFifo::wasFull() const
{
  auto index = _tail.load();
  auto next_tail = (index + 1) % Capacity;
  return (next_tail == _head.load());
}
```

Using the default means that you can write `_tail.load()` and not specify the memory order. You **can** of course explicitly write that it is using the sequential consistent memory order `std::memory_order_seq_cst`.

C++

```
bool CircularFifo::wasFull() const
{
  auto index = _tail.load(std::memory_order_seq_cst);
  auto next_tail = (index + 1) % Capacity;
  return (next_tail == _head.load(std::memory_order_seq_cst));
}
```

Refined Memory Order

Using the fine grained memory models *acquire-release* and *relaxed*, together can squeeze out better performance thanks to less synchronization overhead. The nice to reason about *global synchronization guarantee* from the sequential consistent model is no longer there.

Threads may now see different views of the interacting atomic operations. It is time to thread (pun intended) very carefully. Quoting Anthony Williams: [1, chapter 5.3.3 Memory ordering for atomic operations]

> "*In the absence of other ordering constraints, the only requirements is that all threads agree on the modification order of each individual variable*"

Below are written short, simplified examples for the two fine-grained memory models, relaxed and aquire-release. The area is more advanced then explained here but it is enough for explaining this lock-free code and can work as an **initial** introduction to the fine grained atomic operations.

Relaxed-Memory Model

The relaxed atomic operations can be powerful if used together with acquire-release operations. By itself the relaxed operations can cause quite a bit of confusion for the unwary.

Rules for the Relaxed Memory Model and Operations

1. Relaxed atomic opertions on the same variable, on the same thread guarantee happens-before ordering within that same thread.
2. Operations on the same thread, on the same variable will not be reordered
3. For a synchronized ordering between threads there must be a pair of *acquire* - *release*.
4. Only the modification order of the atomic operations on a variable is communicated to other threads, but as the relaxed memory operation does not use *synchronize-with* the result may surprise you...

Relaxed-Memory operations example

Example inspired from C++11 standard [2, §29.3 Atomics Opertions Library: "Order and Consistency"].

C++

```cpp
// A possible thread interleaving snapshot, i.e. below is an
// possible 'chronological' order of events between two threads
// as they actually "happen"

// x, y are initially zero

// thread_1
auto r1 = y.load(std::memory_order_relaxed);    // t1: 1
x.store(r1, std::memory_order_relaxed);         // t1: 2

// thread_2
auto r2 = x.load(std:memory_order_relaxed);     // t2: 1
y.store(123, std::memory_order_relaxed);        // t2: 2
```

These are thread operations that are **memory unordered**. Even if the thread interleaving snapshot happens in the above *chronological* order the events can *impact* in a different sequence.

I.e. the following **impact** sequence is possible

C++

```cpp
y.store(123, std::memory_order_relaxed);        // t2: 2
auto r1 = y.load(std::memory_order_relaxed);    // t1: 1
x.store(r1, std::memory_order_relaxed);         // t1: 2
auto r2 = x.load(std:memory_order_relaxed);     // t2: 1
```

Resulting in **r1 = r2 = 123;**.

Acquire-Release Memory model

The acquire-release memory model give *some guarantees* for thread synchronization

Rules for the Acquire-Release Memory Model and Operations

1. No guaranteed order for *all* atomic operations
2. Still, it has stronger synchronization orderings than relaxed
3. Guaranteed *thread-pair* ordering synchronization. I.e.between the thread that does *release* and the thread that does *acquire*
4. Reordering is restricted but still not sequential...

C++

```
// thread_1:
y.store(true, std::memory_order_release);

//thread_2:
while(!y.load(std::memory_order_acquire));
```

It is guaranteed that *thread_2* will, **at some point**, see that `y` is true and exit the *while-loop*. The `memory_order_release` synchronizes with the `memory_order_acquire`.

5. Relaxed operations obey a *happens-before* rule that can be used together with acquire-release operations...

C++

```
// thread_1
x.store(123, std::memory_order_relaxed); // relaxed X happens-before Y store
y.store(true, std::memory_order_release);

// thread_2
while(!y.load(std::memory_order_acquire));
assert(123 == x.load(std::memory_order_relaxed));
```

The *happens-before* relationship **guarantees** that there will be **no assert failure** in `assert(123 == x.load(relaxed))`.

Since the *thread_1*: `x.store(123,relaxed)` **happens-before** the `y.store(true, release)` it is guaranteed that when *thread_2*: reads `true == y.load(acquire)` the value of x must already be 123.

Code Explained: Acquire-Release/Relaxed Circularfifo

The fine grained memory models and operations explained above can be used to create even more effective `push()` and `pop()` operations.

Remember that the Producer could only update one variable (`tail`) and the Consumer could only update one member variable (`head`).

The relexed memory operation is only used for loading the atomic variable in that very thread that is responsible for updating the variable. Within the same thread the relaxed operation cannot be reordered for the variable.

bool push(const Element& item)

C++

```cpp
bool push(const Element& item)
{
  const auto current_tail = _tail.load(std::memory_order_relaxed);  // 1
  const auto next_tail = increment(current_tail);
  if(next_tail != _head.load(std::memory_order_acquire))           // 2
  {
    _array[current_tail] = item;                                   // 3
    _tail.store(next_tail, std::memory_order_release);             // 4
    return true;
  }
  return false; // full queue
}
```

1. *Relaxed loading* of the `tail` value. Since the Producer thread is the only thread calling `push()` it is guaranteed that the tail value will be the latest. No cross-thread synchronization is needed for the load.
2. The `next_tail` is compared against the `head` value snapshot. The `head` value is pair-wise *acquire-release* synchronized. It is guaranteed to never come *out-of-order* with following `head` updates.
3. The item is saved in the position pointed to by the *not-yet-synchronized* `next_tail`. This *happens-before* the `release` store in (4) and is guaranteed that the save is *seen* before (4).
4. The `tail` is updated with `release` order. Pair-synchronization with the Consumer thread is guaranteed with the Consumer's `acquire` reading of `tail`.

bool pop(const Element& item)

C++

```cpp
bool pop(Element& item)
{
  const auto current_head = _head.load(std::memory_order_relaxed);    // 1
  if(current_head == _tail.load(std::memory_order_acquire))           // 2
    return false; // empty queue
  item = _array[current_head];                                 // 3
  _head.store(increment(current_head), std::memory_order_release);   // 4
  return true;
}
```

1. *Relaxed loading* of the head value. Since the Consumer thread is the only thread calling pop() it is guaranteed that the head value will be the latest. No cross-thread synchronization is needed for the load.
2. The current_head is compared against the tail value snapshot. The tail value is pair-wise *acquire-release* synchronized. It is guaranteed to never come *out-of-order* with following tail updates.
3. The item is retrieved from the position pointed to by the current_head. This *happens-before* the release store in (4).
4. The head is updated with release order. Pair-wise synchronization with the Producer's thread is guaranteed with the Producer's acquire reading of tail.

Summary
There is *thread-pair* ordering synchronization between the Producer and the Consumer for tail and head. The Producer can never see out-of-order updates for the Consumer updated head and vice-versa for tail.

x86-x64 Comparison: Sequential vs. Refined Memory Order

So does it really matter if using the acquire-release-relaxed type of CircularFifo compared to the sequential?

A very quick and not very scientific test both on Windows (VS2012) and Linux (gcc 4.7.2) showed that

- Windows (x64): acquire-release was approximately 23% faster then the sequential.
- Windows (x86): acquire-release was approximately 22% faster then the sequential.
- Linux (x64): acquire-release was approximately 44% faster then the sequential.
- Linux (x68): acquire-release was approximately 36% faster then the sequential.

Conclusion

When I wrote the original article it was a platform hack. Such hacks are no longer necessary. With C++11 std::atomic can supply all the needed functionality. The big caveat is that apart from the sequential consistent model it is pretty hard to reason about how the operations interact.

The sequential consistent model is very easy to reason about and to implement in a simple structure such as the CircularFifo. Whether or not you use that one or the acquire-release type is up to you.

Finally I have reached the end of this long article. Thank you for reading it. Any comments, improvement suggestions or questions are more then welcome.

History

Initial Version

- 3rd November, 2009: Initial version
- 4th November, 2009: Corrected sentences and highlighted keywords
- 12th November, 2009: Corrected text after feedback from readers. Fixed pictures and code display problems
- 10th June, 2011: "full queue" image and corresponding text after feedback from a reader

Complete rewrite for safe use thanks to C++11 atomic

- 27th November, 2012: Complete **rewrite**. Using C++11 atomics, showing sequential consistent and acquire-release, relaxed orderded atomics to build the CircularFifo.
- 2nd December, 2012: Spell check correction, separation of article in Part I and Part II for an easier read.
- 29th December, 2014. Updated the atomic sequential text in regard to reordering and integrity of the queues. Corrected typos.

References

1. Anthony Williams book C++ Concurrency in Action: Practical Multithreading
2. C++11 ISO IEC_14882:2011 (costs $)".
   A free, slightly newer draft (n3337) version (but contains more than the standard).
   An older, free draft, version (n3242)
3. The **original** version of this article. `Warning`: highly platform dependent
4. Wikipedia on FIFO queues
5. std::atomic_flag class
6. Explained: Weak vs Strong Memory Models (Preshing on programming)
7. Example of weakly-ordered hardware memory model (Preshing on programming)

This article was originally posted at http://kjellkod.wordpress.com/2012/11/28/c-debt-paid-in-full-wait-free-lock-free-queue

License