A Fast Lock-Free Queue for C++

Posted February 07, 2013

Update Nov 6, 2014: I've also just written a multi-producer multi-consumer lock-free queue.

Sharing data between threads in annoying. Really annoying. If you do it wrong, the data is likely to become corrupted. Even more error-prone is ensuring the shared data is perceived in the right order (relative to other data reads/writes).

There are, as always, two ways to deal with this: The easy way, and the hard way.

The easy way

Use the locks (e.g. mutexes, critical sections, or equivalent) that come with your platform. They're not too hard to understand conceptually, and even easier to use. You don't have to worry about ordering problems, because the libraries/OS take care of that for you. The only general problem with locks is that they tend to be slow ("slow" being relative here; for general use, they're plenty speedy enough).

The catch

I was looking into doing some audio programming, where the audio callback (which gets called when the device needs more samples to fill its buffer) was on another thread. The goal with audio programming is to never let the audio "glitch" -- which means your callback code must be as close to real-time programming as you can get on a conventional OS. So, the faster things compute, the better, but what really matters with real-time programming is how *deterministic* the computation time will be -- will your code reliably take 5ms or could there be an occasional 300ms spike?

What does this mean? Well, for starters you can't allocate heap memory -- because that might involve an OS call, which might take a non-deterministic amount of time to complete (e.g. if you have to wait for a page to be swapped to disk). In fact, it's best to avoid calling into the kernel at all. But, this actually isn't the main problem.

The main problem is synchronizing the audio thread with the main thread. We can't use locks, because aside from non-negligible overhead (considering the frequency of callbacks), they can cause priority inversion -- you don't want a high-priority audio thread waiting to enter a critical section that some background thread is obliviously taking its time releasing, causing the audio to glitch.

The hard way

Enter lock-free programming.

Lock-free programming is a way of writing thread-safe code such that in the case of contention, the system is guaranteed to advance as a whole. "Wait-free" programming takes this a step further: the code is set up such that each thread can always advance regardless of what the other is doing. This also has the benefit of avoiding expensive locks.

Why doesn't everyone use lock-free programming? Well, it's not easy to get right. You have to be very careful never to corrupt data under any circumstances, regardless of what each of the threads are doing. But even harder to get right are guarantees related to ordering. Consider this example (a and b start at 0):

```
Thread A    Thread B
b = 42
a = 1
            print(a)
            print(b)
```

What are the possible sets of values that thread B could output? It could be any one of {0 0}, {1 42}, {0 42}, and {1 0}. {0 0}, {1 42}, and {0 42} make sense (which one actually results depends on timing), but how can {1 0} happen? This is because the compiler is allowed to re-order the loads and stores (or even remove some or invent others) in order to improve efficiency, *as long as it appears to do the same thing on that thread*. But when another thread starts interacting with the first, this re-ordering can become apparent.

It gets worse. You can force the compiler to not reorder specific reads or writes, but the *CPU* is allowed to (and does, all the time!) re-order instructions too, again as long it appears to do the same thing on the core that the instructions are running on.

Memory barriers

Fortunately, you can force certain ordering guarantees all the way down to the CPU level using *memory barriers*. Unfortunately, they are extremely tricky to use properly. In fact, a major problem with lock-free code is that it's prone to bugs that can only be reproduced by specific non-deterministic interleavings of thread operations. This means a lock-free algorithm with a bug may work 95% of the time and only fail the other 5%. Or it may work all of the time on a developer's CPU, but fail occasionally on another CPU.

This is why lock-free programming is generally considered hard: It's easy to get something that *looks* right, but it requires much more effort and thought to create something which is *guaranteed* to work under all circumstances. Luckily, there are a few tools to help verify implementations. One such tool is Relacy: It works by running unit tests (that you write) under every possible permutation of thread interleaving, which is really cool.

Back to memory barriers: At the lowest level of the CPU, different cores each have their own cache, and these caches need to be kept in sync with each other (a sort of eventual consistency). This is accomplished with a sort of internal messaging system between the various CPU cores (sounds slow? It is. So it was highly optimized with, you guessed it, more caching!). Since both cores are running code at the same time, this can lead to some interesting orderings of memory loads/stores (like in the example above). Some stronger ordering guarantees can be built on top of this messaging system; that's what memory barriers do. The types of memory barriers that I found most powerful were the acquire and release memory barriers. A release memory barrier tells the CPU that any writes that came before the barrier should be made visible in other cores if any of the writes after the barrier become visible, provided that the other cores perform a read barrier after reading the data that was written to after the write barrier. In other words, if a thread B can see a new value that was written after a write barrier on a separate thread A, then after doing a read-barrier (on thread B), all writes that occurred before the write barrier on thread A are guaranteed to be visible on thread B. Once nice feature of acquire and release semantics is that they boil down to no-ops on x86 -- every write implicitly has release semantics, and every read implicitly has acquire semantics! You still need to add the barriers, though, because they prevent compiler re-ordering (and they'll generate the right assembly code on other processor architectures which don't have as strong a memory-ordering model).

If this sounds confusing, it is! But don't give up hope, because there's many well-explained resources; a good starting point is Jeff Preshing's very helpful articles on the subject. There's also a short list of links in this answer on Stack Overflow.

A wait-free single-producer, single consumer queue

Since I apparently find tangents irresistible, I, of course, set out to build my own lock-free data structure. I went for a wait-free queue for a single producer, single consumer architecture (which means there's only ever two threads involved). If you need a data structure which is safe to use from several threads at once, you'll need to find another implementation (MSVC++ comes with one). I chose this restriction because it's much easier to work with than one where there's a free-for-all of concurrent threads, and because it was all I needed.

I had looked at using some existing libraries (particularly liblfds), but I wasn't satisfied with the memory management (I wanted a queue that wouldn't allocate any memory at all on the critical thread, making it suitable for real-time programming). So, I ignored the copious advice about only doing lock-free programming if you're already an expert in it (how does anyone *become* an expert like that?), and was able to successfully implement a queue!

The design is as follows:

A contiguous circular buffer is used to store elements in the queue. This allows memory to be allocated up front, and may provide better cache usage. I call this buffer a "block".

To allow the queue to grow dynamically without needing to copy all the existing elements into a new block when the block becomes too small (which isn't lock-free friendly anyway), multiple blocks (of independent size) are chained together in a circular linked list. This makes for a queue of queues, essentially.

The block at which elements are currently being inserted is called the "tail block". The block at which elements are currently being consumed is called the "front block". Similarly, within each block there is a "front" index and a "tail" index. The front index indicates the next full slot to read from; the tail index indicates the next empty slot to insert into. If these two indices are equal, the block is empty (exactly one slot is left empty when the queue is full in order to avoid ambiguity between a full block and an empty block both having equal head and tail indices).

In order to keep the queue consistent while two threads are operating on the same data structure, we can take advantage of the fact that the producing thread always goes in one direction in the queue, and the same can be said for the consuming thread. This means that even if a variable's value is out of date on a given thread, we know what range it can possibly be in. For example, when we dequeue from a block, we can check the value of the tail index (owned by the other thread) in order to compare it against the front index (which the dequeue thread owns, and is hence always up-to-date). We may get a stale value for the tail index from the CPU cache; the enqueue thread could have added more elements since then. But, we know that *the tail will never go backwards* -- more elements could have been added, but as long as the tail was not equal to the front *at the time we checked it*, there's guaranteed to be at least one element to dequeue.

Using these sort of guarantees, and some memory barriers to prevent things like the tail from being incremented before (or perceived as being incremented before) the element is actually added to the queue, it's possible to design a simple algorithm to safely enqueue and dequeue elements under all possible thread interleavings. Here's the pseudocode:

```
# Enqueue
If room in tail block, add to tail
Else check next block
    If next block is not the head block, enqueue on next block
    Else create a new block and enqueue there
    Advance tail to the block we just enqueued to

# Dequeue
Remember where the tail block is
If the front block has an element in it, dequeue it
Else
    If front block was the tail block when we entered the function, return false
    Else advance to next block and dequeue the item there
```

The algorithms for enqueueing and dequeueing within a single block are even simpler since the block is of fixed size:

```
# Enqueue within a block (assuming we've already checked that there's room in the block)
Copy/move the element into the block's contiguous memory
Increment tail index (wrapping as needed)

# Dequeue from a block (assuming we've already checked that it's not empty)
Copy/move the element from the block's contiguous memory into the output parameter
Increment the front index (wrapping as needed)
```

Obviously, this pseudocode glosses over some stuff (like where memory barriers are placed), but the actual code is not much more complex if you want to check out the lower-level details.

When thinking about this data structure, it's helpful to make the distinction between variables that are "owned" by (i.e. written to exclusively by) the consumer or producer threads. A variable that is owned by a given thread is never out of date on that thread. A variable that is owned by a thread may have a stale value when it is read on a different thread, but by using memory barriers carefully we can guarantee that the rest of the data is at least as up to date as the variable's value indicates when reading it from the non-owning thread.

To allow the queue to potentially be created and/or destroyed by any thread (independent from the two doing the producing and consuming), a full memory barrier (memory_order_seq_cst) is used at the end of the constructor and at the beginning of the destructor; this effectively forces all the CPU cores to synchronize outstanding changes. Obviously, the producer and consumer must have stopped using the queue before the destructor can be safely called.

Give me teh codes

What's the use of a design without a solid (tested) implementation? :-)

I've released my implementation on GitHub. Feel free to fork it! It consists of two headers, one for the queue and a header it depends on which contains some helpers.

It has several nice characteristics:

- Compatible with C++11 (supports moving objects instead of making copies)
- Fully generic (templated container of any type) -- just like `std::queue`, you never need to allocate memory for elements yourself (which saves you the hassle of writing a lock-free memory manager to hold the elements you're queueing)
- Allocates memory up front, in contiguous blocks
- Provides a `try_enqueue` method which is guaranteed never to allocate memory (the queue starts with an initial capacity)
- Also provides an `enqueue` method which can dynamically grow the size of the queue as needed

- Doesn't use a compare-and-swap loop; this means enqueue and dequeue are O(1) (not counting memory allocation)
- On x86, the memory barriers compile down to no-ops, meaning enqueue and dequeue are just a simple series of loads and stores (and branches)
- Compiles under MSVC2010+ and GCC 4.7+ (and should work on any C++11 compliant compiler)

It should be noted that this code will only work on CPUs that treat aligned integer and native-pointer-size loads/stores atomically; fortunately, this includes every modern processor (including ARM, x86/x86-64, and PowerPC). It is *not* designed to work on the DEC Alpha (which seems to have the weakest memory-ordering guarantees of all time).

I'm releasing the code and algorithm under the terms of the simplified BSD license. Use it at your own risk; in particular, lock-free programming is a patent minefield, and this code may very well violate a pending patent (I haven't looked). It's worth noting that I came up with the algorithm and implementation from scratch, independent of any existing lock-free queues.

Performance and correctness

In addition to agonizing over the design for quite some time, I tested the algorithm using several billion randomized operations in a simple stability test (on x86). This, of course, helps inspire confidence, but proves nothing about the correctness. In order to ensure it was correct, I also tested using Relacy, which ran all the possible interleavings for a simple test which turned up no errors; it turns out this simple test wasn't comprehensive, however, since I eventually did find a bug later using a different set of randomized runs (which I then fixed).

I've only tested this queue on x86-64, which is rather forgiving as memory ordering goes. If somebody is willing to test this code on another architecture, let me know! The quick stability test I whipped up is available here.

In terms of performance, it's *fast*. Really fast. In my tests, I was able to get up to about 12+ million *concurrent* enqueue/dequeue pairs per second! (The dequeue thread had to wait for the enqueue thread to catch up if there was nothing in the queue.) After I had implemented my queue, though, I found another single-consumer, single-producer templated queue (written by the author of Relacy) published on Intel's website; his queue is roughly twice as fast, though it doesn't have all the features that mine does, and his only works on x86 (and, at this scale, "twice as fast" means the difference in enqueue/dequeue time is in the nanosecond range). [Update: Mine is now competitive speed-wise with his.]

Update Updated May 06, 2013

I spent some time properly benchmarking, profiling, and optimizing the code, using Dmitry's single-producer, single-consumer lock-free queue (published on Intel's website) as a baseline for comparison. Mine's now faster in general, particularly when it comes to enqueueing many elements (mine uses a contiguous block instead of separate linked elements). Note that different compilers give different results, and even the same compiler on different hardware yields significant speed variations. The 64-bit version is generally faster than the 32-bit one, and for some reason my queue is much faster under GCC on a Linode. Here are the benchmark results in full:

```
32-bit, MSVC2010, on AMD C-50 @ 1GHz
-------------------------------------
```

	Min		Max		Avg		
Benchmark	RWQ	SPSC	RWQ	SPSC	RWQ	SPSC	Mult
Raw add	0.0039s	0.0268s	0.0040s	0.0271s	0.0040s	0.0270s	6.8x
Raw remove	0.0015s	0.0017s	0.0015s	0.0018s	0.0015s	0.0017s	1.2x
Raw empty remove	0.0048s	0.0027s	0.0049s	0.0027s	0.0048s	0.0027s	0.6x
Single-threaded	0.0181s	0.0172s	0.0183s	0.0173s	0.0182s	0.0173s	0.9x
Mostly add	0.0243s	0.0326s	0.0245s	0.0329s	0.0244s	0.0327s	1.3x
Mostly remove	0.0240s	0.0274s	0.0242s	0.0277s	0.0241s	0.0276s	1.1x
Heavy concurrent	0.0164s	0.0309s	0.0349s	0.0352s	0.0236s	0.0334s	1.4x
Random concurrent	0.1488s	0.1509s	0.1500s	0.1522s	0.1496s	0.1517s	1.0x

```
Average ops/s:
    ReaderWriterQueue: 23.45 million
    SPSC queue:        28.10 million


64-bit, MSVC2010, on AMD C-50 @ 1GHz
-------------------------------------
```

	Min		Max		Avg		
Benchmark	RWQ	SPSC	RWQ	SPSC	RWQ	SPSC	Mult
Raw add	0.0022s	0.0210s	0.0022s	0.0211s	0.0022s	0.0211s	9.6x
Raw remove	0.0011s	0.0022s	0.0011s	0.0023s	0.0011s	0.0022s	2.0x
Raw empty remove	0.0039s	0.0024s	0.0039s	0.0024s	0.0039s	0.0024s	0.6x
Single-threaded	0.0060s	0.0054s	0.0061s	0.0054s	0.0061s	0.0054s	0.9x
Mostly add	0.0080s	0.0259s	0.0081s	0.0263s	0.0080s	0.0261s	3.3x
Mostly remove	0.0092s	0.0109s	0.0093s	0.0110s	0.0093s	0.0109s	1.2x
Heavy concurrent	0.0150s	0.0175s	0.0181s	0.0200s	0.0165s	0.0190s	1.2x
Random concurrent	0.0367s	0.0349s	0.0369s	0.0352s	0.0368s	0.0350s	1.0x

```
Average ops/s:
    ReaderWriterQueue: 34.90 million
    SPSC queue:        32.50 million


32-bit, MSVC2010, on Intel Core 2 Duo T6500 @ 2.1GHz
----------------------------------------------------
```

	Min		Max		Avg		
Benchmark	RWQ	SPSC	RWQ	SPSC	RWQ	SPSC	Mult
Raw add	0.0011s	0.0097s	0.0011s	0.0099s	0.0011s	0.0098s	9.2x
Raw remove	0.0005s	0.0006s	0.0005s	0.0006s	0.0005s	0.0006s	1.1x
Raw empty remove	0.0018s	0.0011s	0.0019s	0.0011s	0.0018s	0.0011s	0.6x
Single-threaded	0.0047s	0.0040s	0.0047s	0.0040s	0.0047s	0.0040s	0.9x
Mostly add	0.0052s	0.0114s	0.0053s	0.0116s	0.0053s	0.0115s	2.2x
Mostly remove	0.0055s	0.0067s	0.0056s	0.0068s	0.0055s	0.0068s	1.2x
Heavy concurrent	0.0044s	0.0089s	0.0075s	0.0128s	0.0066s	0.0107s	1.6x
Random concurrent	0.0294s	0.0306s	0.0295s	0.0312s	0.0294s	0.0310s	1.1x

```
Average ops/s:
    ReaderWriterQueue: 71.18 million
    SPSC queue:        61.02 million


64-bit, MSVC2010, on Intel Core 2 Duo T6500 @ 2.1GHz
----------------------------------------------------
```

	Min	Max	Avg

Benchmark	RWQ	SPSC	RWQ	SPSC	RWQ	SPSC	Mult
Raw add	0.0007s	0.0097s	0.0007s	0.0100s	0.0007s	0.0099s	13.6x
Raw remove	0.0004s	0.0015s	0.0004s	0.0020s	0.0004s	0.0018s	4.6x
Raw empty remove	0.0014s	0.0010s	0.0014s	0.0010s	0.0014s	0.0010s	0.7x
Single-threaded	0.0024s	0.0022s	0.0024s	0.0022s	0.0024s	0.0022s	0.9x
Mostly add	0.0031s	0.0112s	0.0031s	0.0115s	0.0031s	0.0114s	3.7x
Mostly remove	0.0033s	0.0041s	0.0033s	0.0041s	0.0033s	0.0041s	1.2x
Heavy concurrent	0.0042s	0.0035s	0.0067s	0.0039s	0.0054s	0.0038s	0.7x
Random concurrent	0.0142s	0.0141s	0.0145s	0.0144s	0.0143s	0.0142s	1.0x

Average ops/s:
    ReaderWriterQueue: 101.21 million
    SPSC queue: 71.42 million

32-bit, Intel ICC 13, on Intel Core 2 Duo T6500 @ 2.1GHz

	Min		Max		Avg		
Benchmark	RWQ	SPSC	RWQ	SPSC	RWQ	SPSC	Mult
Raw add	0.0014s	0.0095s	0.0014s	0.0097s	0.0014s	0.0096s	6.8x
Raw remove	0.0007s	0.0006s	0.0007s	0.0007s	0.0007s	0.0006s	0.9x
Raw empty remove	0.0028s	0.0013s	0.0028s	0.0018s	0.0028s	0.0015s	0.5x
Single-threaded	0.0039s	0.0033s	0.0039s	0.0033s	0.0039s	0.0033s	0.8x
Mostly add	0.0049s	0.0113s	0.0050s	0.0116s	0.0050s	0.0115s	2.3x
Mostly remove	0.0051s	0.0061s	0.0051s	0.0062s	0.0051s	0.0061s	1.2x
Heavy concurrent	0.0066s	0.0036s	0.0084s	0.0039s	0.0076s	0.0038s	0.5x
Random concurrent	0.0291s	0.0282s	0.0294s	0.0287s	0.0292s	0.0286s	1.0x

Average ops/s:
    ReaderWriterQueue: 55.65 million
    SPSC queue: 63.72 million

64-bit, Intel ICC 13, on Intel Core 2 Duo T6500 @ 2.1GHz

	Min		Max		Avg		
Benchmark	RWQ	SPSC	RWQ	SPSC	RWQ	SPSC	Mult
Raw add	0.0010s	0.0099s	0.0010s	0.0100s	0.0010s	0.0099s	9.8x
Raw remove	0.0006s	0.0015s	0.0006s	0.0018s	0.0006s	0.0017s	2.7x
Raw empty remove	0.0024s	0.0016s	0.0024s	0.0016s	0.0024s	0.0016s	0.7x
Single-threaded	0.0026s	0.0023s	0.0026s	0.0023s	0.0026s	0.0023s	0.9x
Mostly add	0.0032s	0.0114s	0.0032s	0.0118s	0.0032s	0.0116s	3.6x
Mostly remove	0.0037s	0.0042s	0.0037s	0.0044s	0.0037s	0.0044s	1.2x
Heavy concurrent	0.0060s	0.0092s	0.0088s	0.0096s	0.0077s	0.0095s	1.2x
Random concurrent	0.0168s	0.0166s	0.0168s	0.0168s	0.0168s	0.0167s	1.0x

Average ops/s:
    ReaderWriterQueue: 68.45 million
    SPSC queue: 50.75 million

64-bit, GCC 4.7.2, on Linode 1GB virtual machine (Intel Xeon L5520 @ 2.27GHz)

	Min		Max		Avg		
Benchmark	RWQ	SPSC	RWQ	SPSC	RWQ	SPSC	Mult
Raw add	0.0004s	0.0055s	0.0005s	0.0055s	0.0005s	0.0055s	12.1x
Raw remove	0.0004s	0.0030s	0.0004s	0.0030s	0.0004s	0.0030s	8.4x
Raw empty remove	0.0009s	0.0060s	0.0010s	0.0061s	0.0009s	0.0060s	6.4x
Single-threaded	0.0034s	0.0052s	0.0034s	0.0052s	0.0034s	0.0052s	1.5x
Mostly add	0.0042s	0.0096s	0.0042s	0.0106s	0.0042s	0.0103s	2.5x
Mostly remove	0.0042s	0.0057s	0.0042s	0.0058s	0.0042s	0.0058s	1.4x
Heavy concurrent	0.0030s	0.0164s	0.0036s	0.0216s	0.0032s	0.0188s	5.8x
Random concurrent	0.0256s	0.0282s	0.0257s	0.0290s	0.0257s	0.0287s	1.1x

Average ops/s:
    ReaderWriterQueue: 137.88 million
    SPSC queue: 24.34 million

In short, my queue is blazingly fast, and actually *doing* anything with it will eclipse the overhead of the data structure itself.

The benchmarking code is available here (compile and run with full optimizations).

Update update (May 20, '13): I've since changed the benchmark to preallocate the queues' memory, and added a comparison against Facebook's folly::ProducerConsumerQueue (which is a smidgeon faster but cannot grow as needed). Here are the results:

```
64-bit, GCC 4.7.2, on Linode 1GB virtual machine (Intel Xeon L5520 @ 2.27GHz)
--------------------------------------------------------------------------------
                   |----------  Min ------------|------------ Max ------------|--------
Benchmark          |   RWQ    |  SPSC    |  Folly   |   RWQ    |  SPSC    |  Folly   |   RWQ
-------------------+----------+----------+----------+----------+----------+----------+--------
Raw add            | 0.0003s  | 0.0018s  | 0.0003s  | 0.0003s  | 0.0018s  | 0.0003s  | 0.0003s
Raw remove         | 0.0003s  | 0.0030s  | 0.0004s  | 0.0003s  | 0.0030s  | 0.0004s  | 0.0003s
Raw empty remove   | 0.0009s  | 0.0061s  | 0.0006s  | 0.0010s  | 0.0061s  | 0.0006s  | 0.0010s
Single-threaded    | 0.0034s  | 0.0053s  | 0.0033s  | 0.0034s  | 0.0053s  | 0.0033s  | 0.0034s
Mostly add         | 0.0042s  | 0.0046s  | 0.0042s  | 0.0043s  | 0.0046s  | 0.0042s  | 0.0042s
Mostly remove      | 0.0042s  | 0.0049s  | 0.0043s  | 0.0042s  | 0.0051s  | 0.0043s  | 0.0042s
Heavy concurrent   | 0.0025s  | 0.0100s  | 0.0024s  | 0.0028s  | 0.0101s  | 0.0026s  | 0.0026s
Random concurrent  | 0.0265s  | 0.0268s  | 0.0262s  | 0.0273s  | 0.0287s  | 0.0284s  | 0.0269s

Average ops/s:
    ReaderWriterQueue: 142.60 million
    SPSC queue:        33.86 million
    Folly queue:       181.16 million
```

Another update (Jan 28, '15): I optimized my queue a bit further, vastly improving its cache-friendliness under heavy contention (using a trick inspired by the MCRingBuffer paper). Note that my queue is now generally roughly comparable or significantly better than the SPSC and Folly ones, except when it comes to trying to dequeue from an empty queue, which the others can do faster. This skews the final average ops/s summary numbers a little, alas. (We're talking about single-digit nanoseconds here -- a "slow" failed dequeue operation with my queue still only takes 7ns, and that's on my 1GHz C-50 netbook processor.) The results in full:

```
64-bit, GCC 4.8.1, on Linode 1GB virtual machine (Intel Xeon L5520 @ 2.27GHz)
--------------------------------------------------------------------------------
                   |----------  Min ------------|------------ Max ------------|--------
Benchmark          |   RWQ    |  SPSC    |  Folly   |   RWQ    |  SPSC    |  Folly   |   RWQ
-------------------+----------+----------+----------+----------+----------+----------+--------
Raw add            | 0.0004s  | 0.0005s  | 0.0004s  | 0.0004s  | 0.0005s  | 0.0004s  | 0.0004s
Raw remove         | 0.0004s  | 0.0004s  | 0.0005s  | 0.0005s  | 0.0004s  | 0.0005s  | 0.0004s
Raw empty remove   | 0.0061s  | 0.0024s  | 0.0023s  | 0.0061s  | 0.0025s  | 0.0023s  | 0.0061s
Single-threaded    | 0.0121s  | 0.0113s  | 0.0112s  | 0.0122s  | 0.0113s  | 0.0113s  | 0.0122s
Mostly add         | 0.0059s  | 0.0103s  | 0.0096s  | 0.0059s  | 0.0112s  | 0.0109s  | 0.0059s
Mostly remove      | 0.0072s  | 0.0060s  | 0.0061s  | 0.0073s  | 0.0061s  | 0.0077s  | 0.0072s
Heavy concurrent   | 0.0053s  | 0.0082s  | 0.0151s  | 0.0087s  | 0.0106s  | 0.0158s  | 0.0071s
Random concurrent  | 0.0310s  | 0.0328s  | 0.0278s  | 0.0310s  | 0.0329s  | 0.0279s  | 0.0310s

Average ops/s:
    ReaderWriterQueue: 164.08 million
    SPSC queue:        204.05 million
    Folly queue:       206.35 million


64-bit, MinGW GCC 4.9.2, on AMD C-50 @ 1GHz
--------------------------------------------
                   |----------  Min ------------|------------ Max ------------|--------
Benchmark          |   RWQ    |  SPSC    |  Folly   |   RWQ    |  SPSC    |  Folly   |   RWQ
-------------------+----------+----------+----------+----------+----------+----------+--------
Raw add            | 0.0010s  | 0.0023s  | 0.0012s  | 0.0010s  | 0.0023s  | 0.0016s  | 0.0010s
Raw remove         | 0.0010s  | 0.0020s  | 0.0012s  | 0.0011s  | 0.0021s  | 0.0012s  | 0.0011s
Raw empty remove   | 0.0205s  | 0.0121s  | 0.0085s  | 0.0222s  | 0.0122s  | 0.0086s  | 0.0213s
Single-threaded    | 0.0292s  | 0.0285s  | 0.0283s  | 0.0293s  | 0.0286s  | 0.0284s  | 0.0293s
Mostly add         | 0.0172s  | 0.0309s  | 0.0233s  | 0.0174s  | 0.0312s  | 0.0234s  | 0.0173s
```

```
Mostly remove      | 0.0236s | 0.0182s | 0.0172s | 0.0237s | 0.0182s | 0.0174s | 0.0236s
Heavy concurrent   | 0.0147s | 0.0290s | 0.0199s | 0.0159s | 0.0291s | 0.0258s | 0.0155s
Random concurrent  | 0.0982s | 0.0989s | 0.1015s | 0.0986s | 0.0998s | 0.1021s | 0.0984s

Average ops/s:
    ReaderWriterQueue: 64.38 million
    SPSC queue:        53.05 million
    Folly queue:       67.30 million
```

The future of lock-free programming

I think that as multi-threaded code becomes more prevalent, more and more libraries will become available that leverage lock-free programming for speed (while abstracting the gory, error-prone guts from the application developer). I wonder, though, how far this multi-core-with-shared-memory architecture will go -- the cache coherence protocols and memory barrier instructions don't seem to scale very well (performance-wise) to highly parallel CPUs. Immutable shared memory with thread-local writes may be the future. Sounds like a job for functional programming!

MattosChens • a year ago
Hi,
can you add https://github.com/rigtorp/... in benchmarks project?
Thanks.
• Reply • Share ›

MIke Cheng • 2 years ago • edited
Hi,

I remember there is a quite good wiki page describing readerwriterqueue and concurrentqueue. Now I couldn't find them. If there is, please share it. thanks.

readerwriterqueue will copy user data into its internal memory. If my data is quite big, say hundreds/thousands bytes, should I copy it to a buffer and pass the pointer to the queue or directly pass the data to the queue from total performance viewpoint?

https://github.com/cameron3... issue is for issue. how to send some queries? Or stackoverflow?

Ming.
Thanks.

• Reply • Share ›

Cameron Mod ➚ MIke Cheng • 2 years ago

Hmm, don't know of any wiki pages about my queues.

Yes, if your data structures are large, consider passing around unique_ptrs to them instead for reduced copying and increased throughput.

Asking questions in GitHub issues is actually preferred. Email has terrible formatting and I don't see the questions on Stack Overflow.

• Reply • Share ›

MIke Cheng ➚ Cameron • 2 years ago • edited

very glad to know readerwriterqueue can support std::unique_ptr. Would you please elaborate a little bit more on unique_ptr than just native pointer from the queue point of view. Of course unique_ptr will help me to manage the allocated buffer.

Say unique_ptr points to a structure buffer which has several members. Can some of these members are unique_ptr also?

• Reply • Share ›

Cameron Mod ➚ MIke Cheng • 2 years ago

You can use a plain pointer too if you like. unique_ptr was just a suggestion.

And yes, you can have a smart pointer to a structure whose members are other smart pointers.

• Reply • Share ›

MIke Cheng ➚ Cameron • 2 years ago

from now on, I'll move to github Issues. all the best.

• Reply • Share ›