

Lockfree MPMC(*) Queue

Why do we need another Queue?

Erez Strauss ©

erez@erezstrauss.com

CppCon 2023

* Multi Producer, Multi Consumer

https://github.com/erez-schiff/lockfree_mpmc_queue

Lockfree, MPMC Queue - Legal

All Statements and representations are my own and do not reflect those of my employer.

The work presented here is my own and not that of my employer and I take sole responsibility for the work presented during this event.

```
// THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR  
// IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY,  
// FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE  
// AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER  
// LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM,  
// OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE  
// SOFTWARE.
```

— Legal —

Lockfree, MPMC Queue - Background

- Trading platform - every message is tracked, at microseconds resolution
- Queues are essential data structure used to transfer messages from one component to another
- Business decisions depends on the message content and arrival time

Lockfree, MPMC Queue - Connecting Prog. Components

Queues transfer messages and synchronize thread

- A message source to thread pools
- A buffer to handle bursts of incoming messages
- A pool of objects to be used by multiple threads
- A message queue between processes - IPC

Lockfree, MPMC Queue - Queues classification

- Number of producers / consumers - SPSC, SPMC, MPSC, **MPMC**
- **Bounded** / dynamic & memory allocation
- Serialization, **Strict global order** or per producer order
- API, **single item** / multiple items, **atomicity-ready** or not,
- message size: **fixed** / dynamic
- **Queue** - push(back / tail), pop(front / head)
- Deque - push_back(), push_front(), pop_front(), pop_back()
- Blocking / **non blocking** / wait-free
- Data ownership or just **value propagation**

Lockfree, MPMC Queue - Scheduler Interaction

```
template<typename T>
class QueueSchedulerIssue {
    std::atomic<unsigned> writeIndex;
    std::atomic<unsigned> readIndex;
    std::unique_ptr<std::pair<unsigned, T>> elements;

    bool try_push(T&& value)
    {
        auto my_entry = writeIndex.fetch_add(1);
        elements[my_entry].first = std::move(value);
        // No code, does not mean no-time.
        // scheduler will hit here, and
        // block the queue progress.
        elements[my_entry].second = my_entry;
        return true;
    }
};
```

```
template<typename T>
class QueueSchedulerIssue {
// ...
    bool try_pop(T& value)
    {
        auto my_entry = readIndex.fetch_add(1);
        while (elements[my_entry].second
               != my_entry)
            ; // spin and wait, maybe with pause();
        value =
            std::move(elements[my_entry].first);
        return true;
    }
};
```

Lockfree, MPMC Queue - The requirements

- Minimal latency for all message, not just good average
- Worst case scenario, as close as possible to average
- Multi producers threads - on multiple cpu-cores
- Multi consumers threads - on multiple cpu-cores
- C++17 - no need to support C++14
- No system calls during critical path
- No locking - no interaction with the scheduler
- Non blocking - is full, returns with a full-status, on empty the same
- No need of C++ object transfers, simple data types (*)

Lockfree, MPMC Queue - Unique Requirements

- Strict ordering - prevents the use queues with relaxed ordering - multiple send queues (spmc)
- Guaranteed progress - no blocking due to scheduler interruption between two operations, like placing data and increment index.

Lockfree, MPMC Queue - Design

- Fixed size array with atomic entries [1 ... 2^N], atomic entries are 8 or 16 bytes, aligned to cacheline.
- Each Entry contains: 1. Sequence/Index, 2. data-flag, 3. data-value
- Entries in the array are modified only using CAS operations (Compare and Swap)
- A successful CAS operation on an entry completes the push/pop operation.
- Indexes are also modified using CAS operations
- Atomic indexes for read and write

Lockfree, MPMC Queue - Queue Entry

```
class alignas(sizeof(helper_entry)) entry
{
    union entry_union
    {
        mutable entry_as_value _value;
        struct entry_data
        {
            value_type _data;
            index_type _seq;
        } _x;
        entry_union() { _value = 0; }
    } _u;
    index_type get_seq() noexcept { return _u._x._seq; }
    value_type get_data() noexcept { return _u._x._data; }
    bool is_empty() const { return !(_u._x._seq & 1U); }
    bool is_full() const { return !is_empty(); }
};
```

Lockfree, MPMC Queue - Queue States

Queue is empty: read index refers to a cell with false data_flag and the sequence number in the entry is equal to the read index.

if entry.seqnum == read_index + size, then, just increase the read index

if (entry.seqnum == read_index && entry.data_flag)

{ replace the entry with an empty one with seqnum + size }

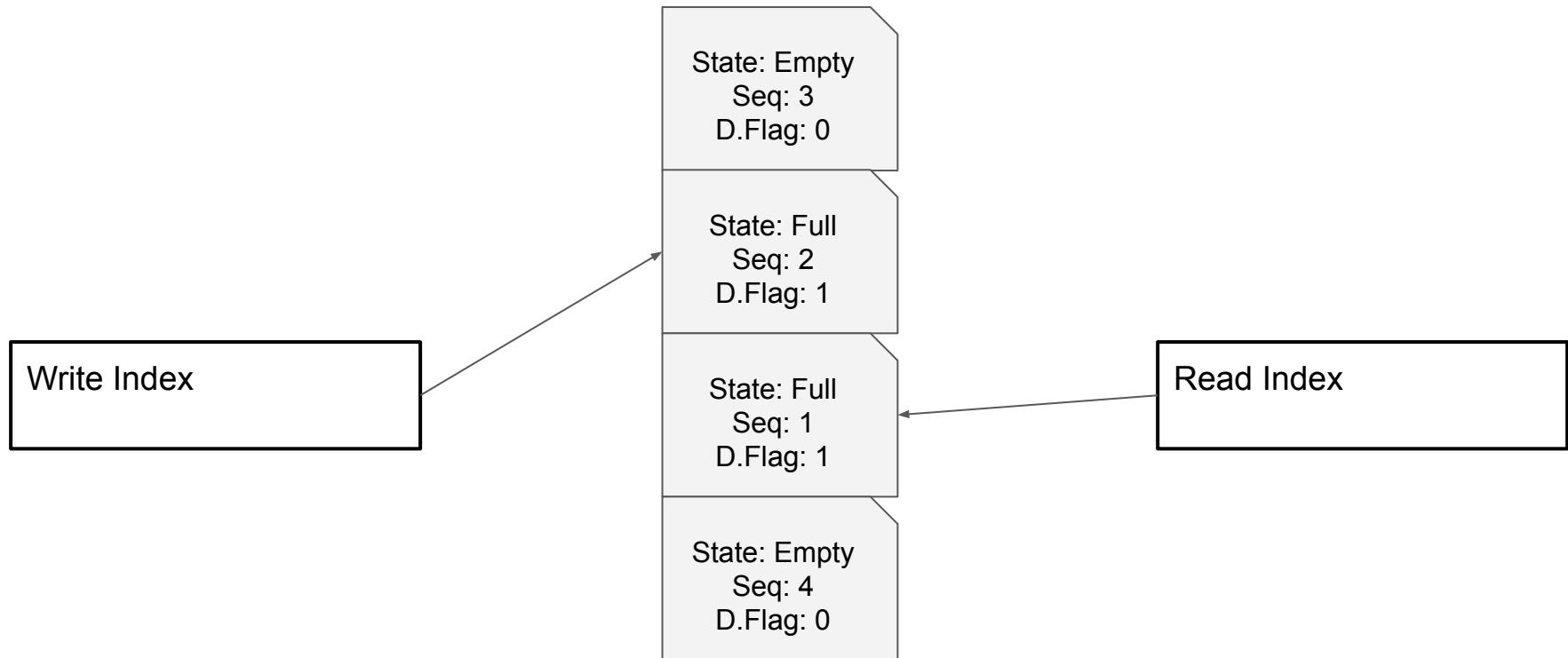
Lockfree, MPMC Queue - Hardware interaction

- C++11 atomic provides `atomic<T>::is_lock_free`
- C++17 provides `atomic<>::is_always_lockfree`
- padding and alignment - avoid false sharing
- CAS (Compare and Swap) on consecutive 16 bytes, using 16 bytes integers
- Alignment requirements, for Entries of 16 bytes, and for other data types
- gcc -mcx16 ...
- Numa - entries on reader side

Lockfree, MPMC Queue - Internal algorithm

- Entries hold data value and sequence & data-present flag bit (lsb)
- The seq is 32 or 64 bits
- Data is 4 to 12 bytes, and can contain pointer / `unique_ptr<>` (simple one)
- `Write_Index` - where should be the next write/push operation
- `Read_Index` - where would be the next read/pop operation
- The relation between the content of the cell and the index defines the state of the index
- Cell & Entry content and Indices content are updated using CAS only
- There is no need that the push-data and write-index-increment will be done by the same thread
- There is no need that the pop-data and read-index-increment will be done by same thread - that way we achieve collaboration

Lockfree, MPMC Queue - depth / Capacity 4 example



Lockfree, MPMC Queue - push / try_push

bool push(T value) - executes two CAS operations, after verifying the write_index points to an empty entry

1. CAS operation on Entry in the array, sets:
sequence - same as before
data_present_flag - 1 - data available, and
data - the new value
2. CAS operation on the Write Index,
progress to next entry

If write index refers to a full entry, from previous round, queue is full, return false.

Lockfree, MPMC Queue - push() code

```
bool push(value_type d) noexcept {
    index_type wr_index = _write_index.load();

    while (true) {
        index_type seq = array[wr_index].get_seq();
        if (seq == static_cast<index_type>(wr_index << 1)) {
            entry e{static_cast<index_type>(wr_index << 1)};
            entry data_entry{static_cast<index_type>((wr_index << 1) | 1U), d};

            if (array[wr_index].compare_exchange(e, data_entry)) {
                if constexpr (!lazy_push)
                {
                    _write_index.compare_exchange_strong(wr_index, wr_index + 1);
                }
                return true;
            }
        } else if ((seq == static_cast<index_type>((wr_index << 1) | 1U)) ||
                   (static_cast<index_type>(seq) == static_cast<index_type>((wr_index + array.size()) << 1))) {
            write_index.compare_exchange_strong(wr_index, wr_index + 1);
        } else if (static_cast<index_type>(seq + (array.size() << 1)) ==
                   static_cast<index_type>((wr_index << 1) | 1U))
        { return false; }

        wr_index = _write_index.load();
    }
}
```

Lockfree, MPMC Queue - pop / try_pop

try_pop(T& value) - executes two CAS operations:

1. CAS operation on Entry in the array, sets:

sequence = sequence + size - make it ready for next write operation

data_present_flag - 0 - clear that entry, and

data - zero value

The Swap operation gets the old value into the value-reference

2. CAS operation on the Read Index,

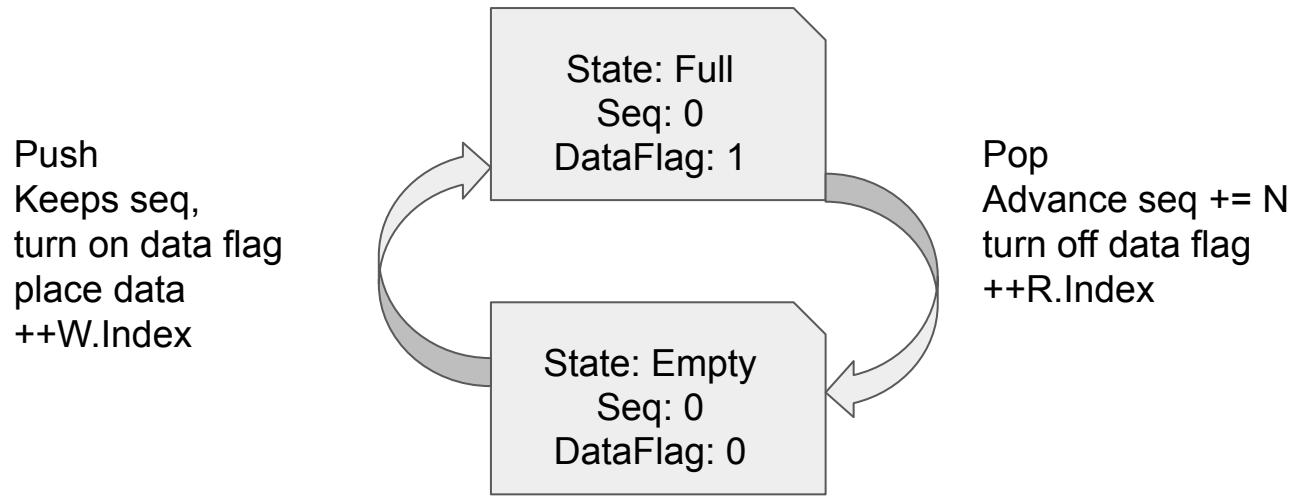
progress to next entry

Lockfree, MPMC Queue - pop() code

```
bool pop(value_type& d) noexcept {
    index_type rd_index = _read_index.load();
    while (true) {
        entry e{array[rd_index].load()};
        if (e.get_seq() == static_cast<index_type>((rd_index << 1) | 1U)) {
            entry empty_entry{static_cast<index_type>((rd_index + _array.size()) << 1U)};

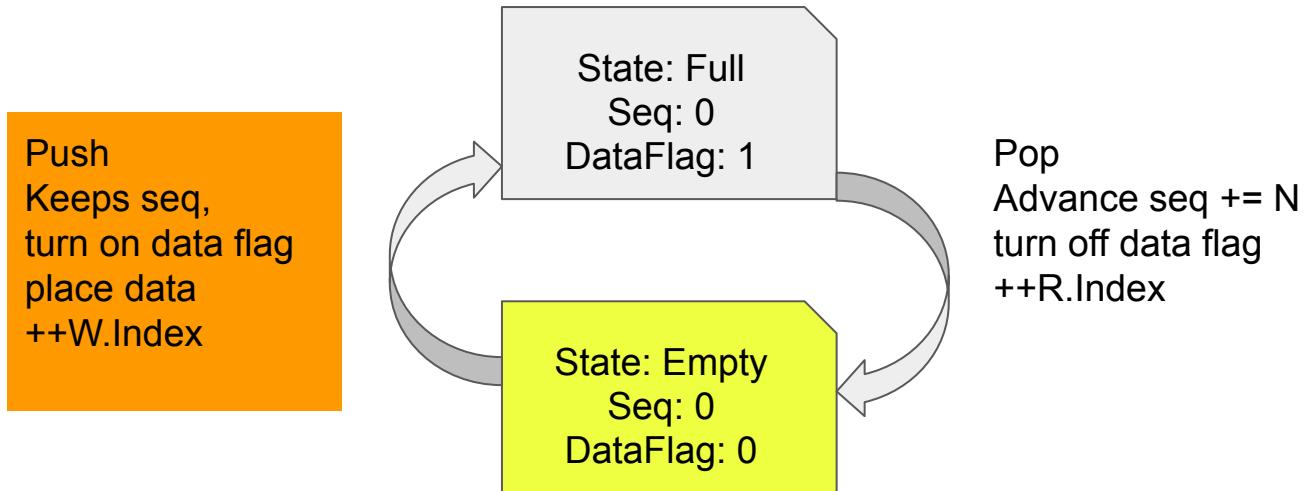
            if (array[rd_index].compare_exchange(e, empty_entry)) {
                d = e.get_data();
                if constexpr (!lazy_pop) {
                    index_type tmp_index = rd_index;
                    ++rd_index;
                    _read_index.compare_exchange_strong(tmp_index, rd_index);
                }
                return true;
            }
        } else if (static_cast<index_type>(e.get_seq() | 1U) ==
                   static_cast<index_type>((rd_index + _array.size()) << 1) | 1U)) {
            read_index.compare_exchange_strong(rd_index, rd_index + 1);
        } else if (e.get_seq() == static_cast<index_type>(rd_index << 1)) {
            { return false; }
            rd_index = _read_index.load();
        }
    }
}
```

Cell / Entry State Diagram - Empty & Full



Cell / Entry State Diagram - Empty & Full

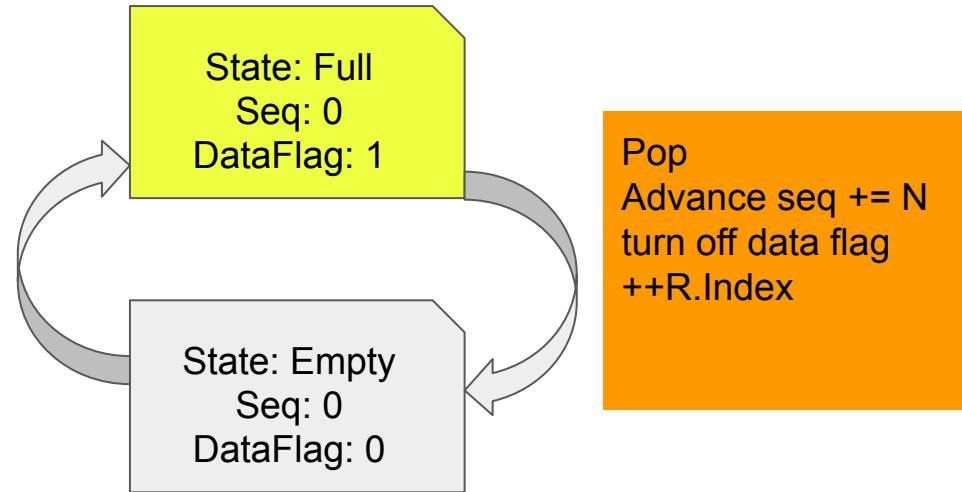
Cell index 0, Start, empty - Round



Cell / Entry State Diagram - Empty & Full

Cell index 0, push1, full, round 0

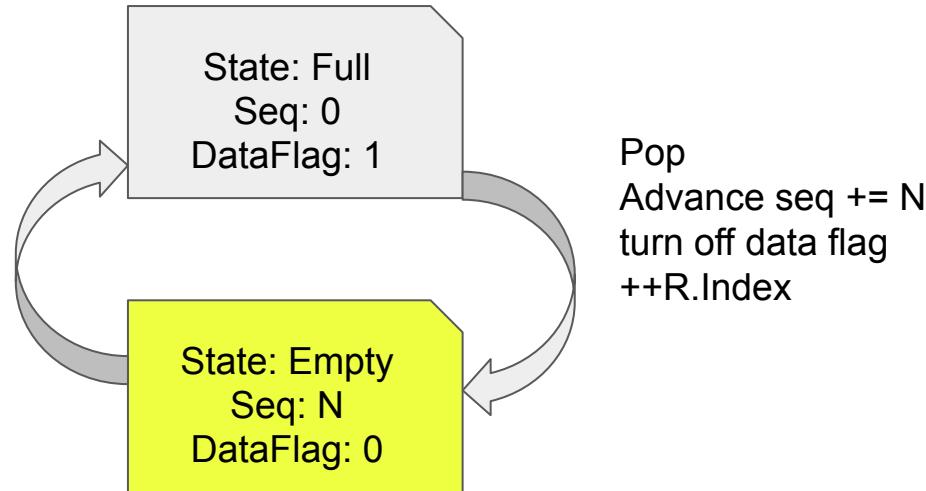
Push
Keeps seq,
turn on data flag
place data
++W.Index



Cell / Entry State Diagram - Empty & Full

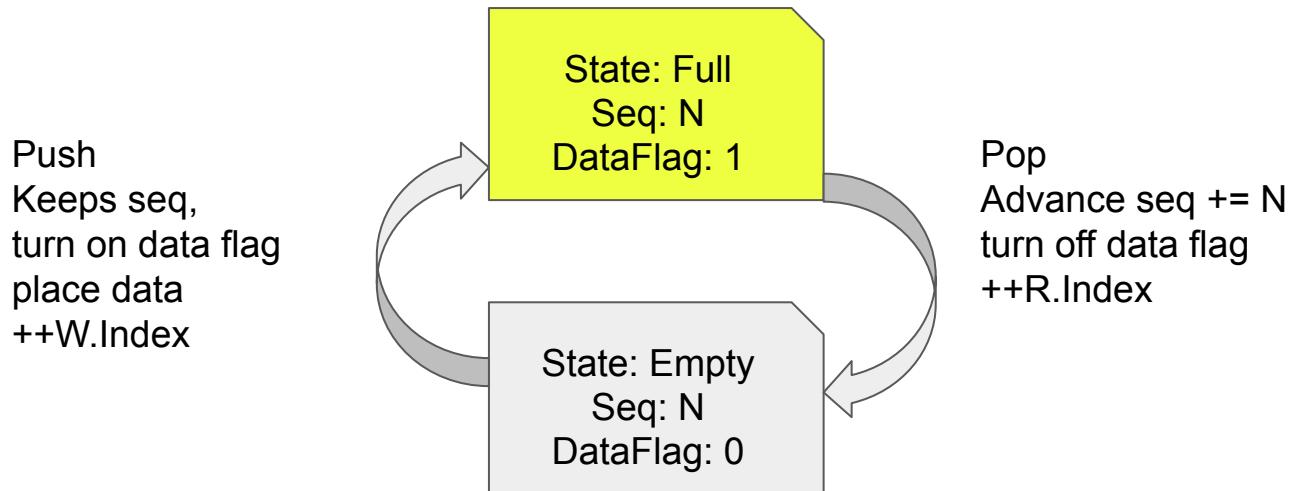
Cell index 0, push1, pop1, empty - round 1

Push
Keeps seq,
turn on data flag
place data
++W.Index



Cell / Entry State Diagram - Empty & Full

Cell index 0, push1, pop1, push2, full - Round 1

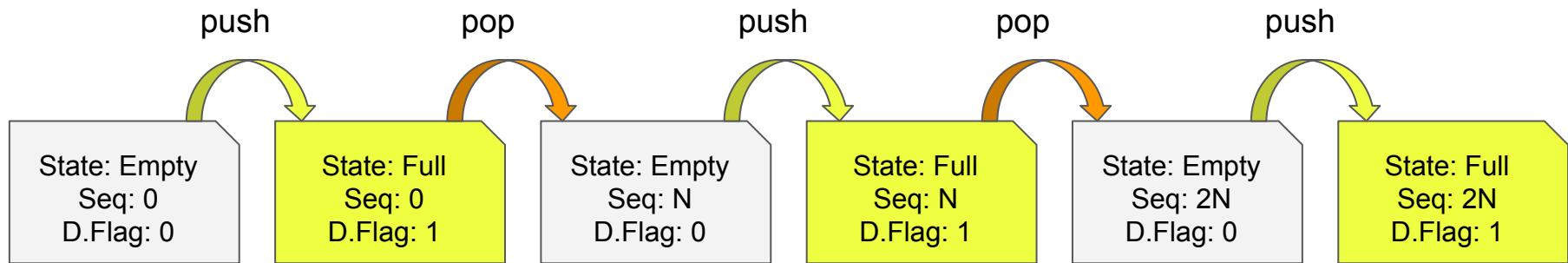


Cell 0, seq#: 0, N, 2N, 3N, 4N 0, N, 2N, ...

Cell 1, seq#: 1, N+1, 2N+1, 3N+1, 4N+1 1, N+1, 2N+1, ...

Cell 2, seq#: 2, N+2, 2N+2, 3N+2, 4N+2 2, N+2, 2N+2, ...

Lockfree, MPMC Queue - Push / pop - one entry



Lockfree, MPMC Queue - Instantiation of a queue

```
template<typename DataT,           // Data Type
         size_t N = 0,           // 0 - set at constructor
         typename IndexT = uint32_t, // index type
         bool lazy_push = false,   // delay write index progress
         bool lazy_pop = false>    // delay read index progress
class mpmc_queue {

    explicit mpmc_queue(uint64_t n = N);
    bool push(value_type d);
    bool pop(value_type& d);
    bool push(value_type d, index_type& i);
    bool pop(value_type& d, index_type& i);
    bool push_keep_n(value_type d);
    bool push_keep_n(value_type d, index_type& i);

    // ...
};

}
```

Lockfree, MPMC Queue - How To Use

```
#include <mpmc_queue.h>
int main()
{
    es::lockfree::mpmc_queue<unsigned> q{32};

    constexpr unsigned N{1000000};
    constexpr unsigned P{2};
    std::atomic<uint64_t> prod_sum{0};
    std::atomic<uint64_t> cons_sum{0};

    auto producer = [&]() {
        for (unsigned x = 0; x < N; ++x) {
            while (!q.push(x))
                ;
            prod_sum += x;
        }
    };
    std::vector<std::thread> producers;
    producers.resize(P);
    for (auto& p : producers) p = std::thread{producer};

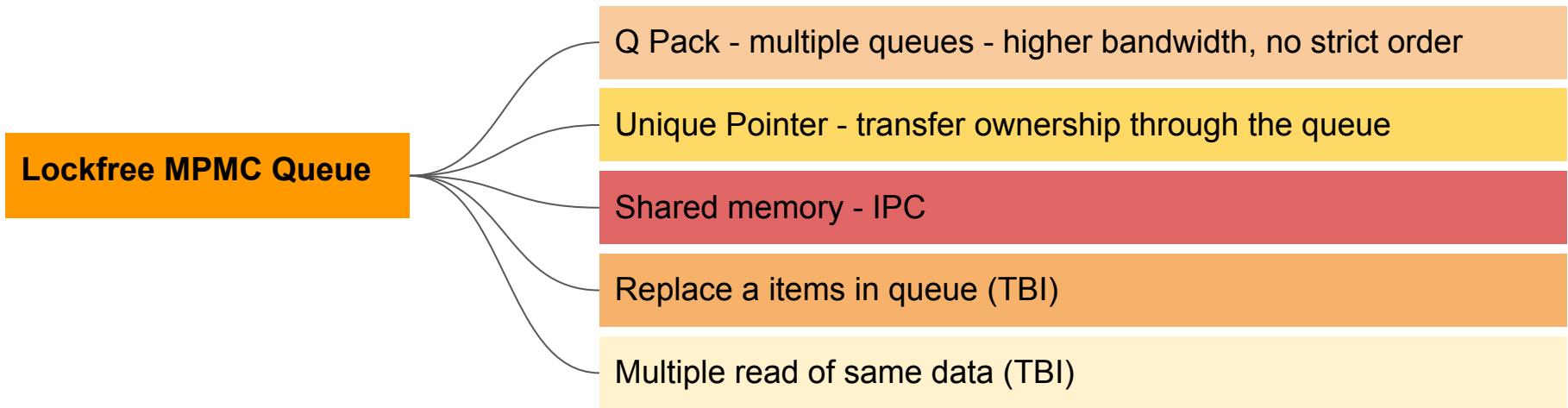
    auto consumer = [&]() {
        unsigned v{0};
        for (unsigned x = 0; x < N; ++x) {
            while (!q.pop(v))
                ;
            cons_sum += v;
        }
    };
    std::vector<std::thread> consumers;
    consumers.resize(P);
    for (auto& c : consumers) c = std::thread{consumer};

    for (auto& p : producers) p.join();
    for (auto& c : consumers) c.join();
    std::cout << (cons_sum && cons_sum == prod_sum ? "OK"
    : "ERROR") << " " << cons_sum << '\n';
    return 0;
}
```

Lockfree, MPMC Queue - Special features

1. Works between threads and between processes - two or more address space
2. Supports atomic conflation - writer can replace atomically a value if not read by any reader

Lockfree, MPMC Queue - Derived work



Lockfree, MPMC Queue - Benchmark

What to measure:

- number of messages per second:
 - different Arch (Intel , AMD)
 - data size - 4, 8, 12 bytes
 - number of producers, consumers 1-1,2-2,3-3,... 1-2,1-3, 2-3, 3-3, 2-1, 3-2 3-3
 - Lazy increment operation in the push / pop - default not lazy
- Time inside the queue per message - need to assume reader is faster than writer
- Half RTT - using echo server - using two queues

Lockfree, MPMC Queue - Performance

CPU overhead in read, write

RTT with echo server

bandwidth - how many messages per second.

benchmark Inputs: data-size, producers-count, consumer-count, lazy read, lazy write

worst case, maximum time in the queue when sending from producer to consumer.

Lockfree, MPMC Queue - Testing

No message drop

No message duplication

No messages reordering

no starvation of reader or writers?

On Intel and AMD platforms

Lockfree, MPMC Queue - Bandwidth Performance

1-1 data width 4:

```
@erez-schlesinger → /workspaces/lockfree_mpmc_queue/cbuild (master) $ ./q_bandwidth -W4 -p1 -c1
```

Q BW: data size: 4 index size: 4 capacity: 32 producers: 1 consumers: 1 for: 1000ms mpmc_queue<ff> push: 73842434 pop: 73842434 tsc: 2470341002 tsc/op: 33 push/pop per sec: 73120193

Q BW: data size: 4 index size: 4 capacity: 32 producers: 1 consumers: 1 for: 1000ms mpmc_queue<ft> push: 78907377 pop: 78907377 tsc: 2445654794 tsc/op: 30 push/pop per sec: 78924290

Q BW: data size: 4 index size: 4 capacity: 32 producers: 1 consumers: 1 for: 1000ms mpmc_queue<tf> push: 85094559 pop: 85094559 tsc: 2445654223 tsc/op: 28 push/pop per sec: 85112818

Q BW: data size: 4 index size: 4 capacity: 32 producers: 1 consumers: 1 for: 1000ms mpmc_queue<tt> push: 75885964 pop: 75885964 tsc: 2445682659 tsc/op: 32 push/pop per sec: 75901364

Lockfree, MPMC Queue - Bandwidth Performance

2-2 data width 4:

```
@erez-schiff → /workspaces/lockfree_mpmc_queue/cbuild (master) $ ./q_bandwidth -W4 -p2 -c2 -d 1024
```

```
Q BW: data size: 4 index size: 4 capacity: 1024 producers: 2 consumers: 2 for: 1000ms mpmc_queue<ff>  
push: 11958731 pop: 11958731 tsc: 2470260104 tsc/op: 206 push/pop per sec: 11841101
```

```
Q BW: data size: 4 index size: 4 capacity: 1024 producers: 2 consumers: 2 for: 1000ms mpmc_queue<ft>  
push: 8372482 pop: 8372482 tsc: 2445914533 tsc/op: 292 push/pop per sec: 8372644
```

```
Q BW: data size: 4 index size: 4 capacity: 1024 producers: 2 consumers: 2 for: 1000ms mpmc_queue<tf>  
push: 8245603 pop: 8245603 tsc: 2445644652 tsc/op: 296 push/pop per sec: 8246672
```

```
Q BW: data size: 4 index size: 4 capacity: 1024 producers: 2 consumers: 2 for: 1000ms mpmc_queue<tt>  
push: 9377623 pop: 9377623 tsc: 2445890844 tsc/op: 260 push/pop per sec: 9377895
```

```
$ ./cbuild/q_bandwidth -W4 -p4 -c4 -d 2048
```

```
Q BW: data size: 4 index size: 4 capacity: 2048 producers: 4 consumers: 4 for: 1000ms mpmc_queue<ff> push: 7372234  
pop: 7372234 tsc: 2230384516 tsc/op: 302 push/pop per sec: 7304356
```

Lockfree, MPMC Queue - Next steps:

1. Benchmarks - compare with similar queues / disruptor
2. Porting to other platforms Windows, RISC-V, Arm
3. Blocking using conditional variable - non-busy wait on empty or full
4. Unify API, according to the WG21 concurrent queue paper
5. Conflation implementation
6. Performance improvement
7. Multiple read of the same entry - reads count

Lockfree, MPMC Queue - Summary

Header only Queue template

- Atomic, lockfree - no interaction with scheduler, collaborative
- bounded - no memory allocation, other than init time
- Multi producer, multi consumer
- limited data size
- ownership transfer using `unique_ptr<T>` wrapper
- between threads or processes - for simple types

Lockfree, MPMC Queue - References

- https://github.com/erez-schiff/lockfree_mpmc_queue
- <https://www.open-std.org/jtc1/sc22/wg21/docs/papers/2020/p0260r4.html>
- moodycamel::ConcurrentQueue:
<https://github.com/cameron314/concurrentqueue>
- Intel:
https://github.com/oneapi-src/oneTBB/blob/master/include/oneapi/tbb/concurrent_queue.h
- Rigtorp's Queue <https://github.com/rigtorp/MPMCQueue>

Lockfree, MPMC Queue

Thank You!