Implementing a Lock-Free Queue With C++ Atomics

Jeff Cohen — ACCU 10/17/2018

Don’t

The End

Seriously, Think Twice

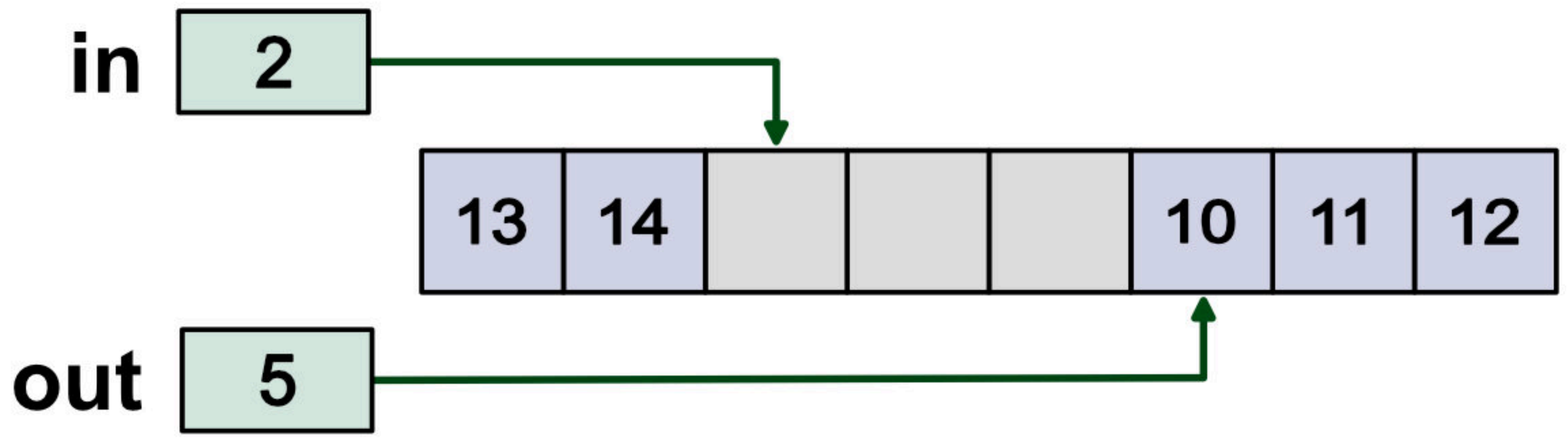
- It's really, *really*, ***really*** hard to avoid race conditions.
- Writing *portable* lock-free code is even more challenging.
    - The key to lock-free data structures are atomic operations, and different CPU architectures differ in their approaches to atomicity.
    - std::atomic hides these differences, but it doesn't make them go away.
    - Though it may work correctly, performance may not be what you expect.
- Only simple lock-free data structures are possible.

So why consider it?

- One word: **Performance**.
    - A sleeping thread that doesn’t hold a lock cannot block other threads.
    - Threads that do not block incur no context-switching overhead.
    - No priority inversion.
    - The higher the contention for a shared data structure, the bigger the speedup.

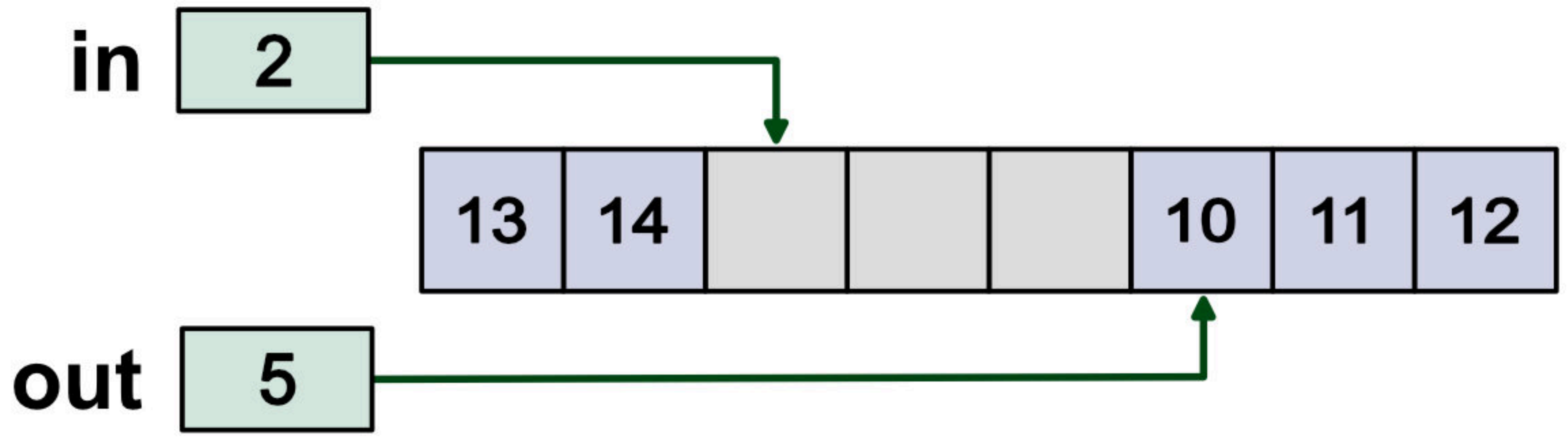
Circular Buffer

- Fixed size, because lock-free memory management is **hard**.
- in and out pointers wrap around the end.
- in points to the first empty slot.
- out points to the first non-empty slot.
- in == out means empty buffer.
  - A buffer with eight slots can only hold seven values.



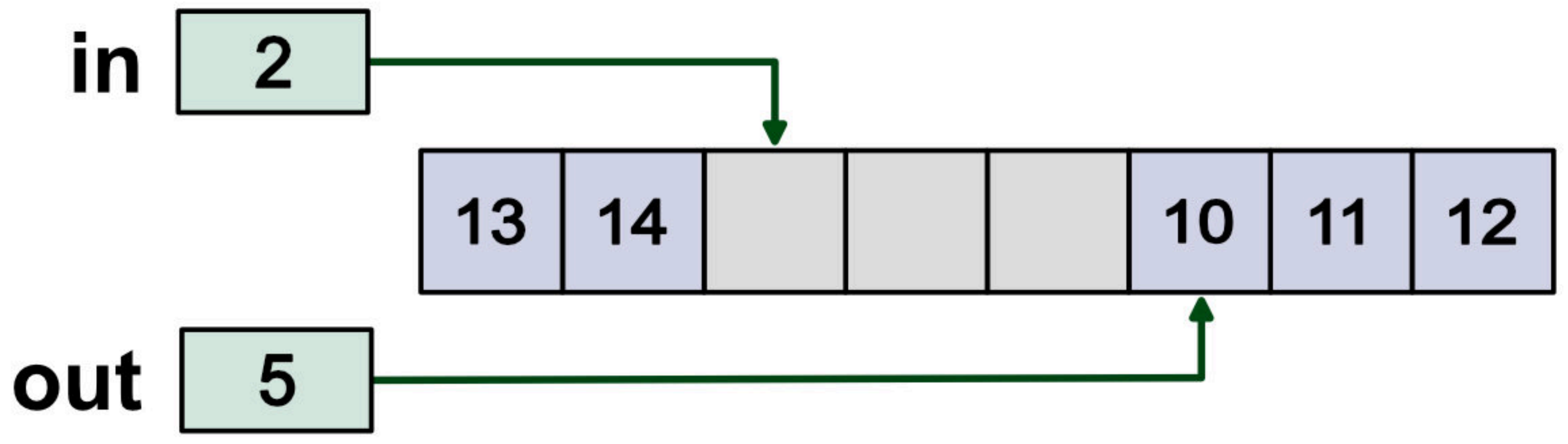
Unsafe Queue

```
class Queue {
    static constexpr int size = 8;
    static_assert(!(size & (size - 1)));

    int   in = 0, out = 0;
    int   queue[size];

    int next(int idx) { return (idx + 1) & (size - 1 ); }

public:
    bool put(int value) {
        int n = next(in);
        if (n == out) return false;

        queue[in] = value;
        in = n;
        return true;
    }

    std::optional<int> get() {
        if (in == out) return { };

        int v = queue[out];
        out = next(out);
        return { v };
    }
};
```



Mutex

```
class Queue {
    static constexpr int size = 16;
    static_assert(!(size & (size - 1)));

    std::mutex  mutex;

    int   in = 0, out = 0;
    int   queue[size];

    int next(int idx) { return (idx + 1) & (size - 1 ); }

public:
    bool put(int value) {
        std::lock_guard<std::mutex> guard(mutex);

        int n = next(in);
        if (n == out) return false;

        queue[in] = value;
        in = n;
        return true;
    }

    std::experimental::optional<int> get() {
        std::lock_guard<std::mutex> guard(mutex);

        if (in == out) return { };

        int v = queue[out];
        out = next(out);
        return { v };
    }
};
```

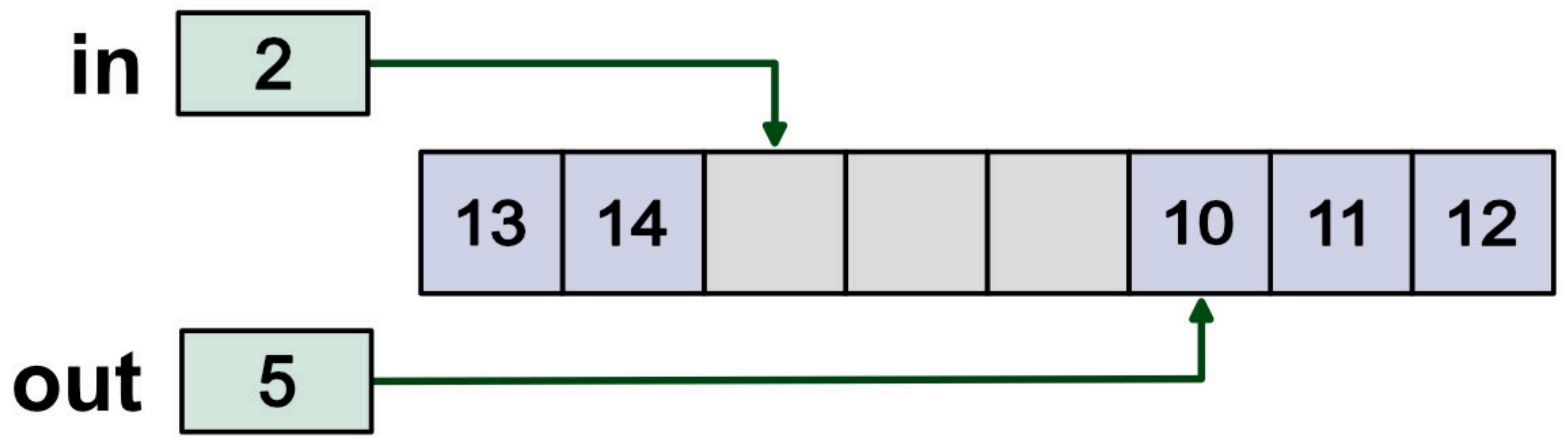


How to make this safe with a mutex?

- Note that two separate memory locations are involved with each get and put.
  - In must be updated after the new value is added.
  - Out must be updated after a value is removed.
  - Otherwise there will be a race condition.
- **But**: the CPU may reorder loads and stores. This reordering is visible to other cores.

Memory Fences

```
class Queue {
    static constexpr int size = 8;
    static_assert(!(size & (size - 1)));

    int   in = 0, out = 0;
    int   queue[size];

    int next(int idx) { return (idx + 1) & (size - 1 ); }

public:
    bool put(int value) {
        int n = next(in);
        if (n == out) return false;

        queue[in] = value;
        std::atomic_thread_fence(std::memory_order_release);
        in = n;
        return true;
    }

    std::optional<int> get() {
        if (in == out) return { };

        int o = out;
        int v = queue[o];
        std::atomic_thread_fence(std::memory_order_acquire);
        out = next(o);
        return { v };
    }
};
```

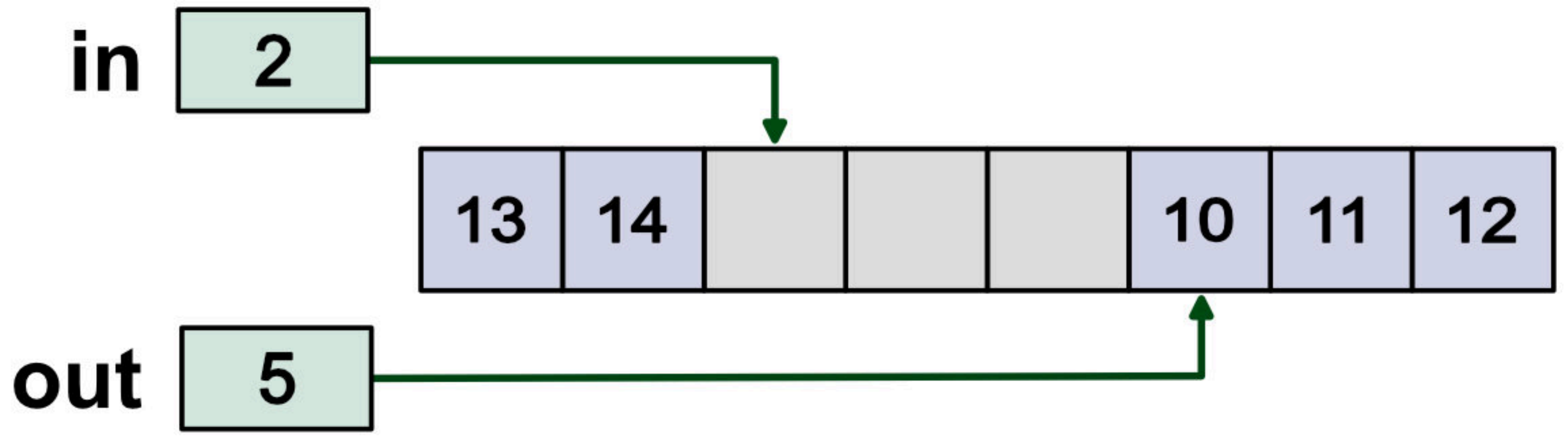


But does it work?

- Yes!
  - If there is only one producing thread and one consuming thread.
  - If it’s compiled with -O0.
- It does not work when optimized, because the optimizer moves or eliminates loads and stores.

std::atomic

```
class Queue {
    static constexpr int size = 8;
    static_assert(!(size & (size - 1)));

    std::atomic<int>   in = 0, out = 0;
    std::atomic<int>   queue[size];

    int next(int idx) { return (idx + 1) & (size - 1 ); }

public:
    bool put(int value) {
        int in_ = in.load(std::memory_order_relaxed);
        int n = next(in_);
        if (n == out) return false;

        queue[in_].store(value, std::memory_order_release);
        in.store(n, std::memory_order_relaxed);
        return true;
    }

    std::optional<int> get() {
        int out_ = out.load(std::memory_order_relaxed);
        if (in == out_) return { };

        int v = queue[out_].load(std::memory_order_acquire);
        out.store(next(out_), std::memory_order_relaxed);
        return { v };
    }
};
```



But does this work?

- Yes!
    - If there is only one producing thread and one consuming thread.
    - If it’s compiled with -O0, -O1, -O2, or -O3.
- The optimizer is prohibited from moving or eliminating atomic loads and stores.

How does this work?

```
C++:

   in.load(std::memory_order_acquire)

LLVM IR:

   %21 = load atomic i32, i32* %20 acquire, align 4

x86_64 Assembler:

   movl     (%rdi), %ecx
```

```
C++:

   out.store(value, std::memory_order_release)

LLVM IR:

   store atomic i32 %12, i32* %23 release, align 4

x86_64 Assembler:

   movl     %ecx, 4(%rdi)
```

But aren't these ordinary move instructions?

What’s going on?

- In the x86 architecture, loads and stores automatically have atomic acquire/release semantics.
- There is a lock prefix byte that can be applied to instructions to make them atomic, but modern x86 cache coherency makes it unnecessary (with an important exception).
- std::atomic works by preventing the optimizer from messing with memory accesses.
- Atomic loads and stores are cheap. There’s no cost unless there’s contention from multiple cores. Over-simplifying, this cost shows up as cache misses.
- The above does not apply to all CPU architectures!

Memory Ordering

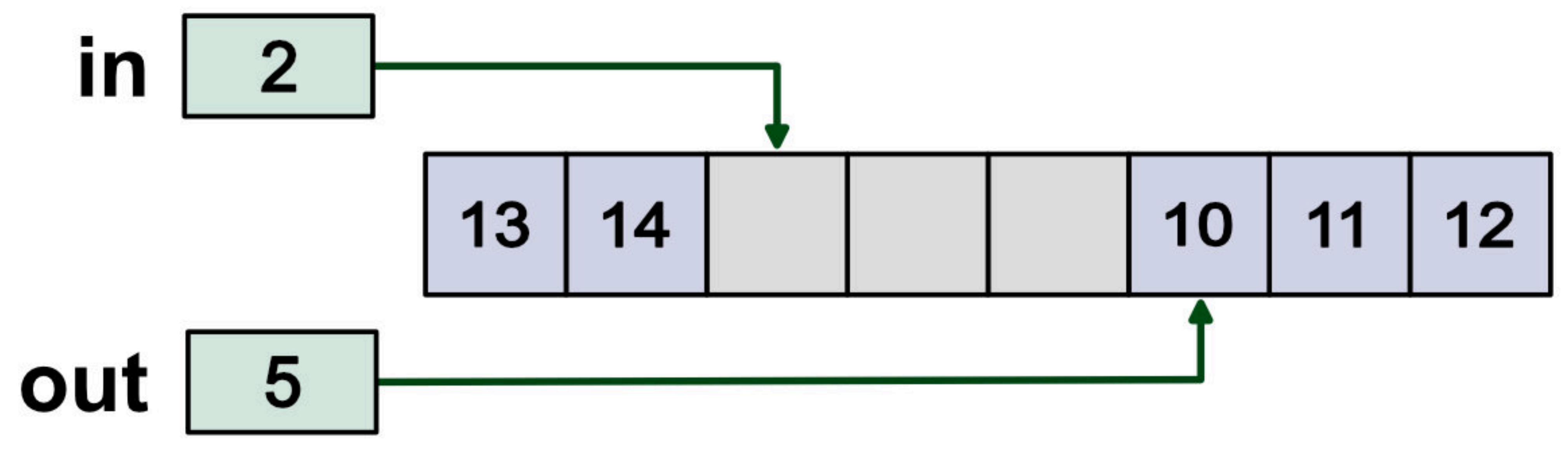
relaxed	No synchronization or ordering constraints. Only atomicity is guaranteed.
acquire	No read or writes in the current thread can be reordered before this load. All writes in other threads that release the same atomic variable are visible in the current thread.
release	No read or writes in the current thread can be reordered after this store. All writes in the current thread are visible in other threads that acquire the same atomic variable.
acquire release	A read-modify-write operation is both an acquire and a release. No read or writes in the current thread can be reordered before or after this store. All writes in other threads that release the same atomic variable are visible before the modification, and the modification is visible in other threads that acquire the same atomic variable.
sequential consistency	In addition to acquire and release semantics, a single total order exists in which all threads observe all sequentially consistent modifications in the same order.

Multiple Producers/Consumers

- When adding a value, both the in pointer and the cell it points at must be updated simultaneously from the standpoint of other producers.

- When removing a value, both the cell the out pointer points at must be read and the out pointer then simultaneously updated from the standpoint of other consumers.

Simple atomic load/stores cannot atomically access multiple, non-contiguous memory locations!



Compare-Exchange

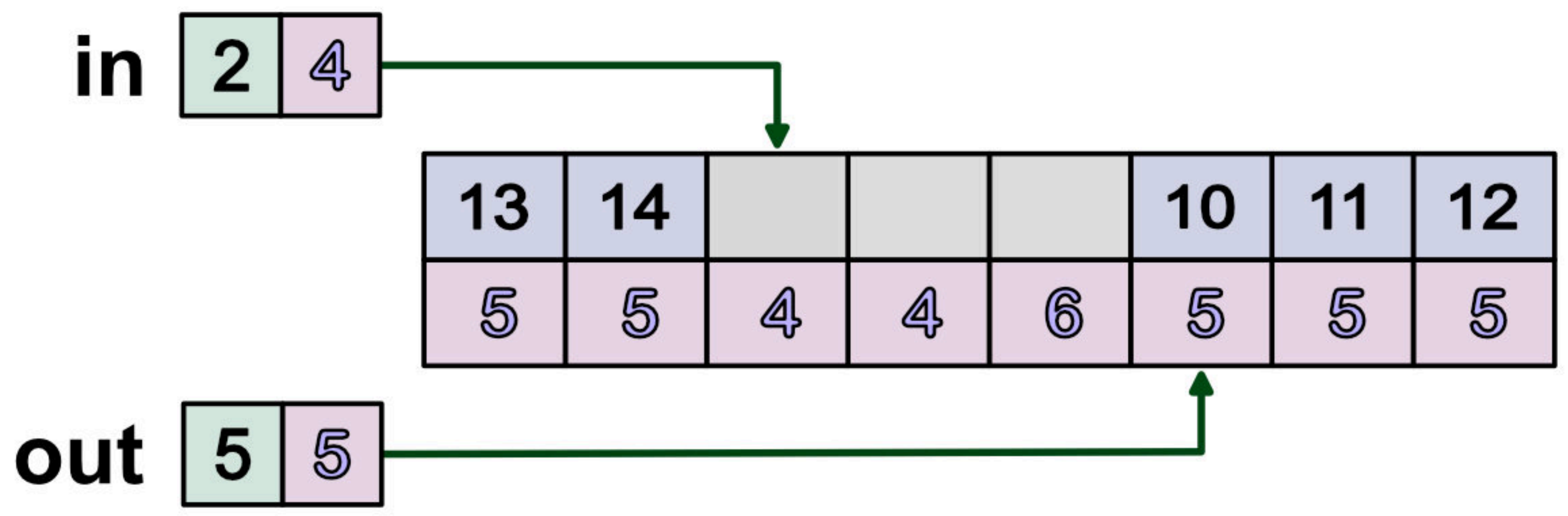
A compare-exchange is a CPU instruction that performs the following function as an atomic, indivisible operation:

```
template<typename T>
bool compare_exchange(T* data, T* expected, T new_data) {
    if (*data != *expected) {
        *expected = *data;
        return false;
    } else {
        *data = new_data;
        return true;
    }
}
```

std::atomic provides this atomic operation. The x86 supports compare-exchanges of up to 16 contiguous and aligned bytes. Some CPUs do not provide this instruction and it must be emulated—sometimes quite slowly.

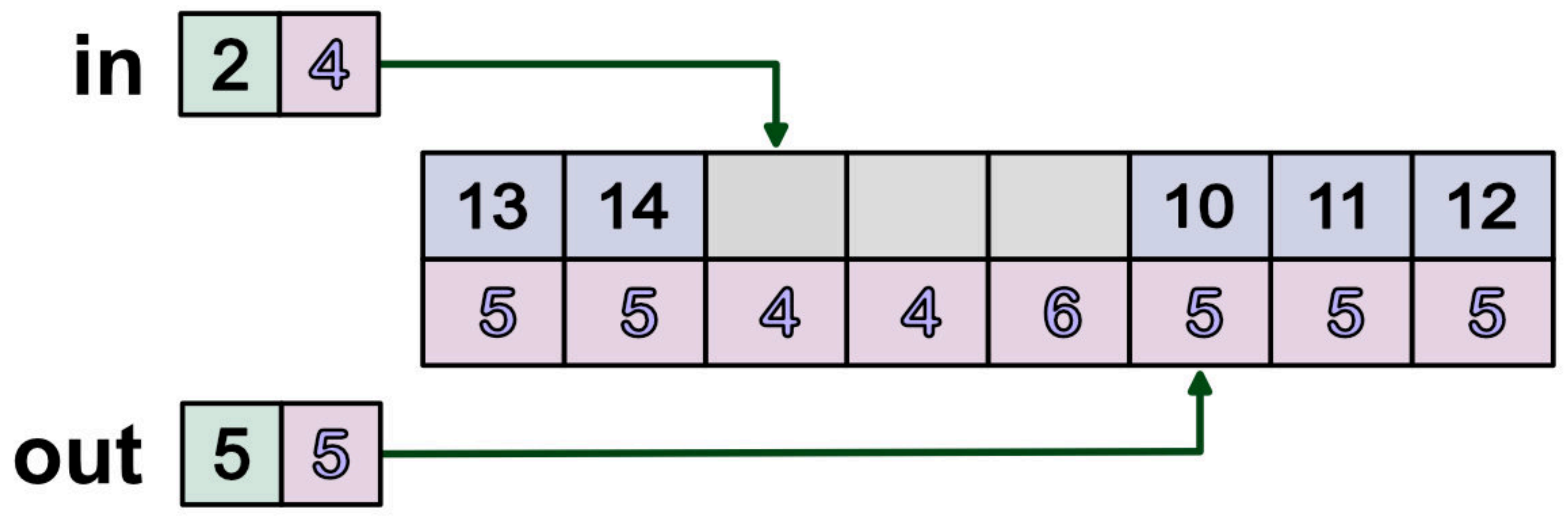
Tagged Circular Buffer

- Everything is tagged with a version number. Compare-exchange will be applied to **both** a value and its version tag.
- Whenever a cell in the buffer is set to a value or its value is retrieved, its version is incremented.
- The version of the in or out pointer is identical to the cell it points to.
- An empty cell always has an even version, an occupied cell an odd version. in == out no longer necessarily means an empty queue.



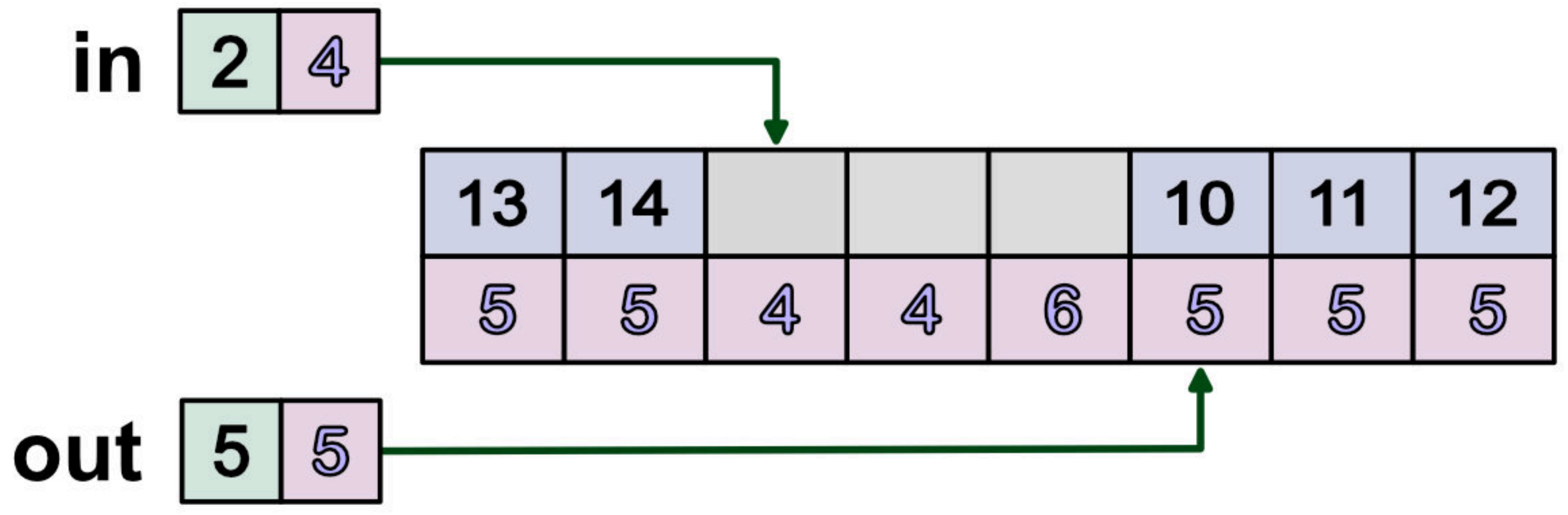
CEX Queue

```
class Queue {
    static constexpr int size = 8;
    static_assert(!(size & (size - 1)));

    struct Index {
        int   index;
        int   version;
    };

    struct Element {
        int   data;
        int   version;
    };

    std::atomic<Index>   in, out;
    std::atomic<Element> queue[size];

    int next(int idx) { return (idx + 1) & (size - 1 ); }

public:
    Queue() {
        in = { 0, 0 }; out = { 0, 0 };
        for (auto &e : queue)
            e = { 0, 0 };
    }

    bool put(int value);
    std::optional<int> get();
};
```



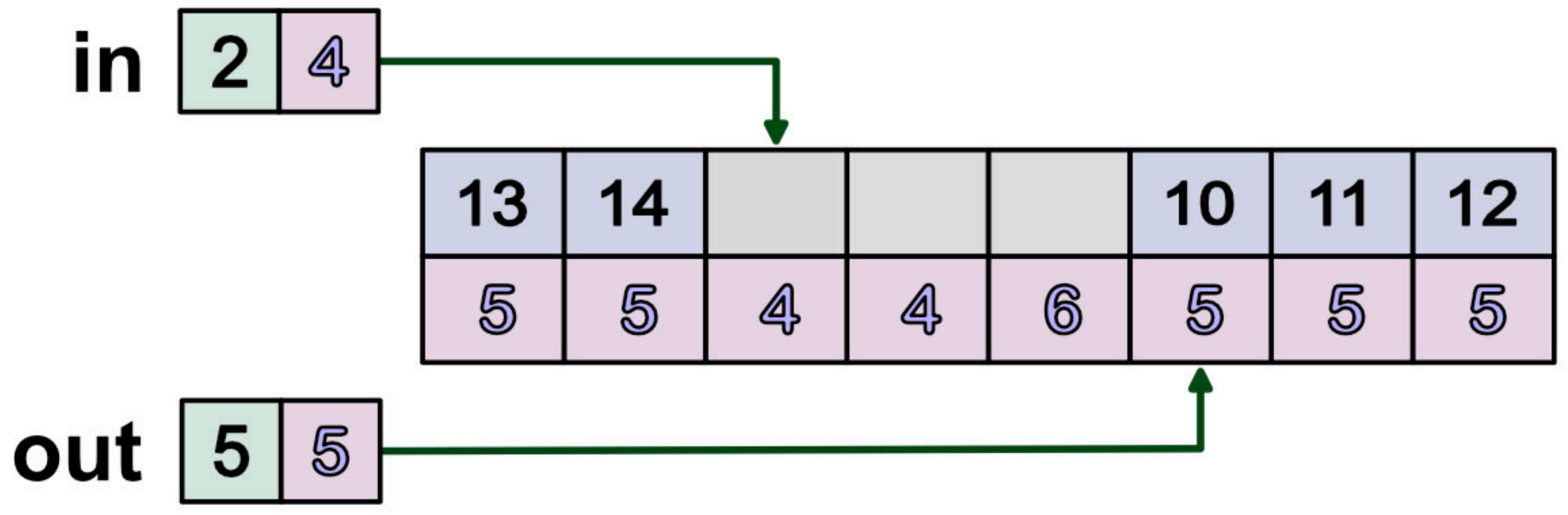
CEX Queue::Put

```
bool Queue::put(int value) {
    while (true)  {
        Index idx = in.load(std::memory_order_relaxed);
        int n = next(idx.index);
        int v = idx.version;
        int nv = queue[n].load(std::memory_order_relaxed).version;

        Element e = queue[idx.index].load(std::memory_order_relaxed);

        // First check if it appears the queue is full (in version is
        // odd).
        if (v & 1) {
            // The queue might not be full; removing data from a full
            // queue does not update in's version.
            if (e.version == v)
                return false;   // it is indeed full

            // Adjust in's version to account for added data.
            ++v;
        }

        // Check if the previous put has updated the in pointer yet.
        if (e.version != v) {
            if (e.version != v + 1)
                continue;       // race condition; try again

            // It hasn’t (see below).  Update it now.
            Index idx2 { n, nv };
            in.compare_exchange_weak(idx, idx2);
            continue;   // retry from the top
        }

        // Attempt to append the data.
        Element e2 { value, e.version + 1};
        if (!queue[idx.index].compare_exchange_weak(e, e2))
            continue;  // we lost the race; try again

        // We appended the data.  Increment in.  We don't care if
        // this fails; if it does, it's probably because we lost
        // the CPU and another thread did it for us.  In that
        // case, we *can't* update it, because who knows how many
        // values have been added and removed while we were
        // asleep?
        Index idx2 { n, nv };
        in.compare_exchange_weak(idx, idx2);
        return true;
    }
}
```



CEX Queue::Get

```
std::optional<int> Queue::get() {
    while (true) {
        Index odx = out.load(std::memory_order_relaxed);
        int n = next(odx.index);
        int v = odx.version;
        int nv = queue[n].load(std::memory_order_relaxed).version;

        Element e = queue[odx.index].load(std::memory_order_relaxed);

        // First check if it appears the queue is empty (out version
        // is even).
        if (!(v & 1)) {
            // The queue might not be empty; adding data to an empty
            // queue does not update out's version.
            if (e.version == v)
                return { };   // it is indeed empty

            // Adjust out's version to account for added data.
            ++v;
        }

        // Check if the previous get has updated the out pointer yet.
        if (e.version != v) {
            if (e.version != v + 1)
                continue;   // we lost the race; try again

            // It hasn't (see below).  Update it now.
            Index odx2 { n, nv };
            out.compare_exchange_weak(odx, odx2);
            continue;   // retry from the top
        }

        // Attempt to acquire the data.
        Element e2 { e.data, e.version + 1};
        if (!queue[odx.index].compare_exchange_weak(e, e2))
            continue;  // we lost the race; try again

        // We got the data.  Increment out.  We don't care if this
        // fails; if it does, it's probably because we lost the
        // CPU and another thread did it for us.  In that case, we
        // *can't* update it, because who knows how many values
        // have been added and removed while we were asleep?
        Index odx2 { n, nv };
        out.compare_exchange_weak(odx, odx2);
        return { e.data };
    }
}
```



Test Driver (one thread)

```
int main(int argc, char **argv) {
    int count = argc > 1 ? std::atoi(argv[1]) : 100000000;

    Queue queue;
    for (int i = 0; i < count; i++) {
        bool success = queue.put(i);
        assert(success);

        auto value = queue.get();
        assert(value && i == *value);
    }
}
```

Test Driver (two threads)

```
void get_thread(Queue *queue, int count) {
   for (int i = 0; i < count; i++) {
        while (true) {
            auto value = queue->get();
            if (value) {
                assert(i == *value);
                break;
            }
        }
    }
}

void put_thread(Queue *queue, int count) {
   for (int i = 0; i < count; i++) {
        while (!queue->put(i))
            ;
    }
}
```

```
int main(int argc, char **argv) {
    int count = argc > 1 ? std::atoi(argv[1]) : 100000000;

    Queue queue;
    std::thread t1(get_thread, &queue, count);
    std::thread t2(put_thread, &queue, count);

    t1.join();
    t2.join();
    return 0;
}
```

Test Driver (many threads)

```
void get_thread(Queue *queue, int start, int count, int ccnt,
                bool *got) {
   for (int i = start; i < count; i += ccnt) {
        while (true) {
            auto value = queue->get();
            if (value) {
                assert(*value >= 0 && *value < count);
                assert(!got[*value]);
                got[*value] = true;
                break;
            }
        }
    }
}

void put_thread(Queue *queue, int start, int count, int pcnt)
{
   for (int i = start; i < count; i += pcnt) {
        while (!queue->put(i))
            ;
    }
}
```

```
int main(int argc, char **argv) {
    int count = argc > 1 ? std::atoi(argv[1]) : 100000000;
    int pcnt = argc > 2 ? std::atoi(argv[2]) : 1;
    int ccnt = argc > 3 ? std::atoi(argv[3]) : pcnt;

    bool *got[ccnt];
    got[0] = new bool[count * ccnt];
    for (int i = 1; i < ccnt; i++)
        got[i] = got[i - 1] + count;

    std::vector<std::thread> ts;
    Queue queue;

    for (int i = 0; i < pcnt; i++)
        ts.emplace_back(put_thread, &queue, i, count, pcnt);

    for (int i = 0; i < ccnt; i++)
        ts.emplace_back(get_thread, &queue, i, count, ccnt, got[i]);

    for (auto& t : ts)
        t.join();

    for (int i = 1; i < ccnt; i++)
        for (int j = 0; j < count; j++) got[0][j] |= got[i][j];

    for (int i = 0; i < count; i++)
        assert(got[0][i]);

    return 0;
}
```

Performance

	One Thread	Two Threads	Six Producers / Six Consumers
Unsafe	0.86ns		
Atomic Load/Store	1.82ns	33.5ns 200%	
Compare-Exchange	1.92ns	148ns 200%	2.7µs 1160%
Mutex	35.1ns	8.4µs 132%	8.4µs 119%

Questions