Atomic Library and Lockfree Container

source codes in this doc are not verified

Part A – Memory model

Part B – Lockfree container

What is thread contention?

Mutex avoids race condition by protecting critical session using locks. However disadvantage is that many threads may be blocked on requesting a lock, resulting in contention. Contention disables concurrency.

What is thread preemption?

Preemption refers to the suspension of a thread by the scheduler for running another thread by the same physical core.

Reference

B2	Lockfree stack	*node based*	Anthony Williams, Concurrency in action
B3	Lockfree bounded MPMC queue	*array based*	Dmitry Vyukov, 1024 cores
B4	Lockfree hashmap	*array based*	Jeff Preshing, Preshing on programming
B5	Lockfree hashmap, WRRM	*STL based*	Andrei Alexandrescu, Dr Dobb blog

A1. Atomic variable 5x4, 3x3, 432

Operations		atomic flag	atomic<bool>	atomic<T*>	atomic<int>	atomic<T>
movable		no	no	no	no	no
copyable		no	no	no	no	no
clear	init	yes	no	no	no	no
test_and_set	init	yes	no	no	no	no
atomic<T>(const T&)	direct	no	yes	yes	yes	yes
T operator=(const T&)	copy	no	yes	yes	yes	yes
store	store	no	yes	yes	yes	yes
load	load	no	yes	yes	yes	yes
exchange	rmw	no	yes	yes	yes	yes
compare_ex...weak	rmw	no	yes	yes	yes	yes
compare_ex...strong	rmw	no	yes	yes	yes	yes
fetch_add, +=	rmw	no	no	yes	yes	no
fetch_sub, -=	rmw	no	no	yes	yes	no
fetch_and, &=	rmw	no	no	no	yes	no
fetch_or, |=	rmw	no	no	no	yes	no
fetch_xor, ^=	rmw	no	no	no	yes	no

```
void atomic<T>::store (T desired, memory_order tag);
T atomic<T>::load                (memory_order tag);
T atomic<T>::exchange (T desired, memory_order tag);
bool atomic<T>::compare_exchange_strong (T& expected, T desired, memory_order tag_success, memory_order tag_fail);
bool atomic<T>::compare_exchange_weak   (T& expected, T desired, memory_order tag_success, memory_order tag_fail);
```

Producer and writer use store release memory order, whereas consumer and reader use load acquire memory order. Memory order for successful CAS and failed CAS can be different, the latter must be looser than than the former.

What is atomic?

1.1 Atomic operation is an indivisible operation, it is either done or not-done, it cannot be half-done.

- Atomic variable is a type having all operations being atomic.
- Atomic variable may or may not be implemented by mutex internally.
- `std::atomic_flag` is the only type that guarantees lockfree implementation across all CPU architectures.
- Atomic variable is **non-copyable** and **non-movable**. Both copy and move involve 2 steps, hence cannot be atomic.

1.2 Atomic variable is either direct-initialized or copy-initialized :

```
T x;
std::atomic<T> atomic_x0(x);
std::atomic<T> atomic_x1 = x;
```

1.3 For `atomic<T>` type :

- `T` must be trivially copy-constructable and copy-assignable
- `T` must support `memcpy` and `memcmp`, as they are needed to build CAS.

About fetching

2.1 Difference between `fetch_add` and `operator+=`

- `atomic<T>::fetch_add` returns value prior to operation, it does support explicit memory order
- `atomic<T>::operator+=` returns value after operation, it doesn't support explicit memory order

2.2 Differennce between global `atomic_fetch_add` and member `atomic<T>::fetch_add`

- global functions take C-style ptr as input
- global functions come in pairs, with/without explicit memory order.

```
T atomic_fetch_add         (atomic<T>* ptr, T x)                   noexcept { return ptr->fetch_add(x);     }
T atomic_fetch_add_explicit(atomic<T>* ptr, T x, memory_order tag) noexcept { return ptr->fetch_add(x,tag); }
```

2.3 How to implement `std::fetch_apply` in general?

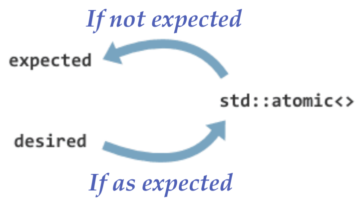
```
T atomic<T>::fetch_apply(std::function<T(const T&)> fct)
{
    T expected = this->load();
    while(!this->compare_exchange_strong(expected, fct(expected)));
    return expected; // return old value
}
```

About CAS

3.1 What is CAS?

Herlihy Maurice had proved that compare-and-swap *CAS* is the fundamental building block for lockfree containers and algorithms in paper *"Wait-free synchronization"* 1991. *CAS* is an atomic conditional swap defined as :

```
bool atomic<T>::compare_exchange_strong(T& expected, T desired)
{
    if (this->load() == expected)
    {
        this->store(desired); return true;
    }
    else
    {
        expected = this->load(); return false;
    }
}
```



3.2 How CAS help in protecting critical session without a lock?

Suppose we have a piece of critical session code, which ends with the modification of a flag, a counter or any variable that indicates work is done, instead of wrapping the whole thing with mutex which protects against racing condition at the expense of contention, we assume that current thread can finish critical session without preempted by other threads, then :

- loading an atomic-flag `x` (or atomic-counter `x`) at the beginning of critical session
- running the critical session
- invoke CAS on `x` with expected-old-value and desired-new-value
- if CAS is OK, we assume no preemption occurs, it is fine to move forward
- if CAS is not OK, preemption occurred and we need to retry again … thus put everything inside while-loop
- when critical session is the increment of `x` itself, then we can simplify the code as RHS, thus there are two patterns :

Pattern 1 – general pattern

```
while(true)
{
    T expected = x.load();
    critical_session();
    if (x.compare_exchange(expected, inc(expected)))
    {
        post_processing();
        break;
    }
}
```

Pattern 2 – if critical session is `inc` itself, for lockfree stack in B2.1

```
// step 0 ...
T expected = x.load(); //  step 1 fetch

while(!x.compare_exchange(expected, inc(expected))); // step 2 update
// step 3 ...
```

```
This pattern is useful in lockfree stack in B2.1.
For push fct : adding new node in step 0
For pop  fct : adding del node in step 3
```

3.3 Strong CAS and weak CAS

Weak verion *CAS* does fail spuriously even if `content==expected`, because for some CPU architectures, *CAS* weak is not implemented as single instruction, the thread in the middle of *CAS* may be preempted by another thread, *CPU* cannot guarantee atomicity, hence fails it, `expected` is not modified in this case. This is called ***spurious failure***. Strong *CAS* can be implemented with weak *CAS* as :

```
bool atomic<T>::compare_exchange_strong(T& expected, T desired)
{
    T expected_copy = expected;
    while(!this->compare_exchange_weak(expected, desired) && expected == expected_copy);
    return expected == expected_copy;
}
```

Strong *CAS* is usually used in a while loop like 3.2, there are actually two while loops :

- external while loop for claiming atomic counter `x`
- internal while loop for handling spurious failure
- when `inc(expected)` is simple, use weak *CAS* to avoid double while loop
- when `inc(expected)` is expensive, use strong *CAS* to avoid repeated `inc`

Summary 4 pieces of code / 3 with while loop / 2 with critical session

	2.3	3.1	3.2	3.3
code	y	y	y	y
with while loop	y	-	y	y
with critical sesion	pattern2	-	pattern1&2	-

A2. Happen-before and synchronized-with relationship

Experiment

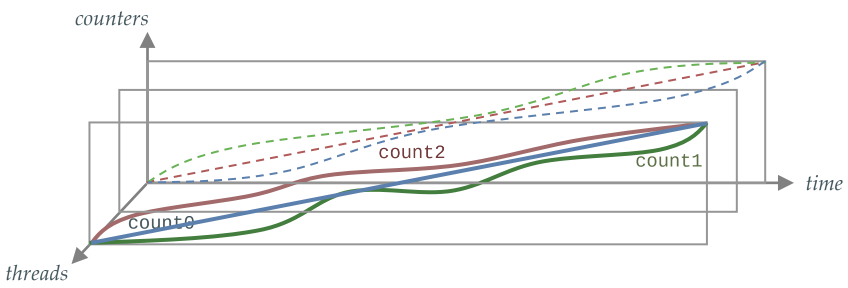
Consider 3 threads in different cores :

- each thread begins iterating until starting pistol `start` is fired
- each thread keeps incrementing an atomic counter `x/y/z` in a loop starting from zero
- each thread keeps recording snapshots of all counters
- memory order for store and load operations of `x/y/z` are all relaxed

```
std::atomic<bool> start = false;
std::atomic<int> x = -1;
std::atomic<int> y = -1;
std::atomic<int> z = -1;

struct snapshot { int x,y,z; };
snapshot ss_x[1000];
snapshot ss_y[1000];
snapshot ss_z[1000];

void  functor_x()
{
    while(!start);
    for(unsigned short n=0; n!=1000; ++n)
    {
        ss_x[n].x = x.load(std::memory_order_relaxed);
        ss_x[n].y = y.load(std::memory_order_relaxed);
        ss_x[n].z = z.load(std::memory_order_relaxed);
        x.store(n, std::memory_order_relaxed);
    }
}

std::thread thread_x(functor_x);
std::thread thread_y(functor_y);
std::thread thread_z(functor_z);
start = true;
thread_x.join();
thread_y.join();
thread_z.join();
```

We print the snapshots at the end. The red lines show 1-increment, while other snapshots are inconsistent.

	n	0	1	2	3	4	5	6	
thread_x	ss_x[n].x	0	1	2	3	4	5	6	
	ss_x[n].y	0	0	0	5	7	8	8	
	ss_x[n].z	0	3	3	4	4	5	6	...
thread_y	ss_y[n].x	0	0	0	1	1	2	3	
	ss_y[n].y	0	1	2	3	4	5	6	
	ss_y[n].z	0	3	3	4	4	5	6	...
thread_z	ss_z[n].x	0	1	1	3	3	3	7	
	ss_z[n].y	0	0	0	0	4	4	5	
	ss_z[n].z	0	1	2	3	4	5	6	...



With the above results, we will study 4 concepts and 1 design pattern :

- sequence-before and happen-before
- sequence (in)consistency
- synchronized-with
- transitivity
- publication pattern

1. Sequence-before and happen-before relationship

We define two relationships between instructions A and B :

- 1.1 `sequenced-before` relationship means A is sequenced before B in source code
- 1.2 `happens-before` relationship means effects of A in memory is visible to the thread going to execute B before it is executed
- 1.3 if an instruction `sequenced-before` another one `happens-before` the latter, it is called sequence consistency
- 1.4 sequence consistency is guaranteed in single thread scenario

  sequence consistency is *NOT* guaranteed in multithread scenario

2. Two reasons for sequence inconsistency

Why is that? The reason is :

- 2.1 high level abstraction
- compiler is allowed to reorder ***instruction*** while *CPU* is allowed to reorder ***data*** in order to improve performance
- as long as sequence consistency is guaranteed in single thread scenario
- while it is legal to break sequence consistency in multi thread scenario

- 2.2 how does compiler reorder instruction under the hood?
- compiling program is like building *DAG*, with input / output / intermediate variables as nodes, function as edges
- perform a topological sorting from input to reach output
- find shortest path and remove redundant nodes and edges
- optimal solution may result in instruction reordering (as long as reordering is indifferent to topological sorting)

```
OUTPUT fct(const A& a, const B& b, const C& c)
{
    auto x = f0(a,b);
    auto y = f1(b,c);
    auto z = f2(c,a);
    auto w = f3(a,b,c);
    return x+y+z;
}
// f0, f1, f2 can be reordered, as long as they are executed before x+y+z, f3 is redundant
// Whats the problem? If another thread ables to access x,y,z who assumes updating sequence x>y>z
// then there will be problem if reordering does occur
```

- 2.3 how does *CPU* reorder data under the hood?
- as memory access is performance bottleneck, cache is added to *CPU*
- if some cache lines in different cores map to the same main memory address, multi-cores are reading and writing to them ...
- inter-core cache synchronization is needed by means of *MESI*, which blocks *CPUs* to wait for *MESI* acknowledgements
- to avoid blocking *CPUs*, store-buffer and invalidate-queue are added, yet it leads to sequence inconsistency (data reordered)
- this is fine for some applications, however for others, we need to flush store-buffer and invalidate-queue by memory barrier

3. Synchronized-with relationhip

What is synchronization (or `synchronizes-with` relationship)? Synchronization is :

- 3.1 alignment in time among threads at some point in the program
- 3.2 which usually involves one thread waiting for another
- 3.3 it can be done using mutex, spinlock, condition variable, future, promise and *atomic variables checking inside while loop*
- 3.4 with which we hope to create an *interthread* `happens-before` between :
- instructions before synchronization in producer-side and
- instructions after synchronization in consumer-side

4. Transtivity

What is transitive property between `happens-before` and `synchronizes-with` relationship? Consider publication-pattern :

- 4.1 if a store (write) to variable A `happens-before` a store to variable B in ***single thread*** (defined in source code) and
- 4.2 if a load (read) from variable B `happens-before` a load from variable A in ***another single thread*** (defined in source code) and
- 4.3 if the store to B is synchronized with the load from B by :
- making B atomic
- store to B with `memory_order_release`
- load from B with `memory_order_acquire` inside a while loop
- 4.4 then the store to A is ***inter-thread*** `happens-before` the load from A

5. Publication pattern

5.1 Given A is publication `p`, B is flag `ready`.

```
publication p = old_value;
std::atomic<bool> ready = false;

void writer()
{
    p = new_value;
    ready.store(true, std::memory_order_release);
}
void reader()
{
    while(!ready.load(std::memory_order_acquire));
    std::cout << p;
}
```

5.2 Publication pattern can be generalized into a transtivity chain.

```
publication p = old_value;
std::atomic<bool> writer_ready = false;
std::atomic<bool> agency_ready = false;

void writer()
{
    p = new_value;
    writer_ready.store(true, std::memory_order_release);
}
void agency()
{
    while(!writer_ready.load(std::memory_order_acquire));
    agency_ready.store(true, std::memory_order_release);
}
void reader()
{
    while(!agency_ready.load(std::memory_order_acquire));
    std::cout << p;
}

std::thread t0(writer);
std::thread t1(agency);
std::thread t2(reader);
```

5.3 Transitivity works for chain, thus the store to `p` ***inter-thread*** `happens-before` the load from `p`. We can merge the atomic booleans.

```
publication p = old_value;
std::atomic<int> stage = 0;

void writer()
{
    p = new_value;
    stage.store(1, std::memory_order_release);
}
void agency()
{
    int expected = 1;
    while(!stage.load(expected, 2, std::memory_order_acquire)) expected = 1;
}
void reader()
{
    while(agency_ready.load(std::memory_order_acquire)!=2);
    std::cout << p;
}
```

6. Referenece count increment and decrement

For shared pointer :

- increment can be implemented with `memory_order_relaxed`
- decrement can be implemented with `memory_order_acq_rel`

Why? This is because (my humble opinion only, please verify) :

- increment-increment race is safe
- as reference count should start from `1` when we do assignment (as `rhs` object counts `1`)
- when two increments race, it is about incrementing reference count `1->2` or `2->3`
- no deletion happens for both cases, we do not care which thread invokes `1->2` or `2->3`
- increment-decrement race and decrement-decrement race are not safe
- when the above races happen, we need to know which thread will end up with count `0`, as this is the thread which deletes
- thus decrement is protected with `memory_order_acq_rel`

A3. Memory order and Memory fence (or memory barrier)

Without explicit memory order, instructions in C++ program are defaulted to be *sequence consistent*. However `memory_order_seq_cst` is slow, we can relax memory order for the sake of lower latency. C++ offers fine-grained control over memory order in 2 ways :

- operations with ***memory order options***
- operations called ***memory fence***

1. Memory order

Memory order is an abstraction of store-buffer flush and invalidate-queue flush in cache coherency mechanism. I do not know how store-buffer and invalidate-queue are flushed, perhaps involve complicated algorithms, there is no need to drill too deep into cache coherency protocol. The good thing is, memory order does the cache coherency for us with *6* options :

more relaxed

Memory order	meaning from Bartosz Milewski.com
`std::memory_order_seq_cst`	`sequence consistency, all threads observe the same sequence of events`
`std::memory_order_release`	`preceding stores cannot be moved after current store or subsequent stores`
`std::memory_order_acquire`	`subsequent loads cannot be moved before current load or preceding loads`
`std::memory_order_consume`	`relaxed memory_order_acquire, restriction is valid only if there is dependency`
`std::memory_order_acq_rel`	`combination of memory_order_release and memory_order_acquire`
`std::memory_order_relax`	`all sequence of events are fine`

Besides, there is limitation when using those memory orders (note **memory orders ≠ memory fence = memory barrier**) :

Memory order	store	load	read-modify-write
`std::memory_order_seq_cst`	yes	yes	yes
`std::memory_order_release`	yes	no	yes
`std::memory_order_acquire`	no	yes	yes
`std::memory_order_consume`	no	yes	yes
`std::memory_order_acq_rel`	no	no	yes
`std::memory_order_relax`	yes	yes	yes

2. Memory fence (also known as memory barrier)

Memory fence is global function that provides the same set of memory order constraints without doing any modification on atomic variable. The previous publication pattern can be written as :

```
publication p = old_value;
std::atomic<bool> ready = false;

void writer()
{
    p = new_value;

    std::atomic_thread_fence(std::memory_order_release);
    ready.store(true, std::memory_order_relaxed);
}
void reader()
{
    while(!ready.load(std::memory_order_relaxed)) continue;
    std::atomic_thread_fence(std::memory_order_acquire);

    std::cout << p;
}
```

With memory barrier, it is easy to visualize …

`preceding stores cannot be moved after current store`

`subsequent loads cannot be moved before current load`

3. Unify the two perspectives

High level abstraction of memory order :

- preceding stores cannot be moved after `std::memory_order_release` tagged store or subsequent stores
- subsequent loads cannot be moved before `std::memory_order_acquire` tagged load or preceding loads

What happens under the hood :

- `std::memory_order_release` is the flushing of store-buffer
- `std::memory_order_acquire` is the flushing of invalidate-queue

B1.1 Blocking vs lockfree vs waitfree

In order to illustrate the differences among these 3 mechanisms, lets consider a single core system running two threads, which race to execute the instructions inside a critical session. Protection against racing condition can be done by :

1. blocking with lock

- the faster thread enters critical session while the slower thread is blocked
- the faster thread may be suspended by scheduler some point in time to run the slower thread
- the slower thread preempts the faster thread, however *preempting thread waits to be notified by preempted thread*
- as a result, for a short period of time, no thread can make progress

2. lockfree : non-blocking, retry with undone

- both threads can enter critical session without being blocked
- the faster thread may be suspended by scheduler some point in time to run the slower thread
- the slower thread preempts the faster thread, now preempting thread finishes its job without waiting for preempted thread
- however when preempted thread resumes, on finding that it has been preempted, *undoes its half-done job* and retries
- as a result, for a short period of time, only one thread can make progress, called ***systemwise*** progress

3. waitfree : non-blocking, pick up where it left off

- both threads can enter critical session without being blocked
- the faster thread may be suspended by scheduler some point in time to run the slower thread
- the slower thread preempts the faster thread, again preempting thread finishes its job without waiting for preempted thread
- this time when preempted thread resumes, it picks up where it left off, without undoing anything
- as a result, for a short period of time, both threads can make progress, called ***threadwise*** progress

Examples

- blocking mutex lock, spinlock
- lockfree *CAS* pattern in *section A1. part 3.2*
- waitfree lockfree ring buffer without *CAS*

B1.2 Node and cell definition 7 x 5

- Node is for node-based container (such as list).
- Cell is for array-based container (such as buffer and hash table).

[B2.1] Stack - no reclamation	2 mem fct	push	pop	return value
	push, pop	INNER-while-loop 0. new node 1. fetch head 2. update head	OUTER-while-loop 1. fetch head A. check null 2. update head B. return value 3. reclaim node	shared_ptr<T>

```
template<typename T> struct node
{
    std::shared_ptr<T> ptr;
    node<T>* next;
};

template<typename T> struct lockfree_stack
{
    std::atomic<node<T>*> head;
};
```

[B2.3] Stack - shared ptr	2 mem fct	push	pop	return value
	push, pop	INNER-while-loop same as above	OUTER-while-loop same as above	shared_ptr<T>

```
template<typename T> struct node
{
    std::shared_ptr<T> ptr;
    std::shared_ptr<node<T>> next;
};

template<typename T> struct lockfree_stack
{
    std::shared_ptr<node<T>> head;
};
```

[B2.4] Stack - double reference counter	4 mem fct	push	pop	return value
	push, pop inc, dec	INNER-while-loop same as above	OUTER-while-loop same as above	shared_ptr<T>

```
template<typename T> struct node
{
    std::shared_ptr<T> ptr;
    counted_ptr<T> next;
    std::atomic<int> internal_count;
};

template<typename T> struct counted_ptr
{
    node<T>* impl;
    int external_count;
};

template<typename T> struct lockfree_stack
{
    std::atomic<counted_ptr<T>> head;
};
```

[B3] Bounded MPMC queue	2 mem fct	push	pop	return value
	push, pop	OUTER-while-loop 1. fetch next_write 2. resolve PC races 3. update next_write	OUTER-while-loop 1. fetch next_read 2. resolve PC races 3. update next_read	const T&, T

```
template<typename T> struct cell
{
    T value;
    std::atomic<bool> flag;
};

template<typename T> struct
{
    std::atomic<int> next_write;
    std::atomic<int> next_read;
}
```

[B4.1] Single thread hash	2 mem fct	set	get	return value
	set, get	for-loop-in-probe 1. hashing 2. probing 3. compare key with 4 cases Uninit / ==key !=key / full	for-loop-in-probe 1. hashing 2. probing 3. compare key	optional<T>

```
template<typename K, typename V> struct cell
{
    unsigned short hashed_key;
    K key;
    V value;
};
```

[B4.2] Lockfree array	2 mem fct	set	get	return value
	set, get	for-loop-in-probe 1. hashing 2. probing 3. compare key by atomic_CAS	for-loop-in-probe 1. hashing 2. probing 3. compare key by atomic_load	optional<T>

```
struct cell
{
    std::atomic<int> key;
    std::atomic<int> value;
};
```

[B4.3] Lockfree hash	2+3 mem fct	set	get	return value
	set, get inc, dec cpy_and_add	for-loop-in-probe 1. hashing 2. probing 3. compare key by atomic_CAS **cas optimization**	for-loop-in-probe 1. hashing 2. Probing 3. compare key by atomic_load	optional<T>

```
struct cell
{
    std::atomic<int> hashed_key;
    std::atomic<int> value;
};
```

B2.1 Lockfree stack – naïve version

This is a lockfree stack built on top of a list, which is a specialization of the singly-list in *algorithm.doc*, as we insert front only.

1. Push function

For single thread, `push` is done in 3 steps :

```
node<T>* new_node = new node<T>(x);   // step 0. new node
new_node->next = head;                // step 1. fetch head
head = new_node;                      // step 2. update head
```

Now two threads try to `push` at the same time. Suppose this is the time-interleaving :

```
thread0       thread1
push D        push E                 C → B → A → nullptr
                                     head

step1                                D → C → B → A → nullptr
step2  ---------------------------->     head

              step1                  head            (head is node E)
              step2                  E →
              step3 --------------> D → C → B → A → nullptr

step3  ---------------------------->  E →
                                     D → C → B → A → nullptr
                                     head            (head is node D)
```

Branches are created and only one of them is picked as the `head`. However our desired output is either :

```
E → D → C → B → A → nullptr
head
```

or

```
D → E → C → B → A → nullptr
head
```

Solution for push

Root cause is that `step1&2` together is not one atomic step. For lockfree container, instead of ensuring that certain operation is atomic, we speculate that a running thread is not preempted by other threads in the whole operation `step1&2` and verify if the speculation is valid at the end of operation using *CAS*, if it is preempted by other threads, *CAS* does tell you by returning `false`. The beauty of the *CAS* implementation is that, even if *CAS* fails because of preemption, it reloads `new_node->next` with the latest `head`, so that preempted thread can retry `step2` without undoing anything when it resumes execution.

```
template<typename T> struct node
{
    node(const T& x) : value(x) {}

    T value;
    node<T>* next = nullptr;
};

template<typename T> struct lockfree_stack
{
    void push(const T& x)
    {
        node<T>* new_node = new node(x);                                // step 0. new node
        new_node->next = head.load();                                   // step 1. fetch head
        while(!head.compare_exchange_weak(new_node->next, new_node));   // step 2. update head
    }

    std::atomic<node<T>*> head = nullptr;                               // Don't forget to make head atomic
};
```

2. Pop function

For single thread, `pop` is also done in 3 steps, correspond to `push` counterparts.

```
implementation for pop                      implementation for push is shown for comparison
                                            node<T>* new_node = new node<T>(x);  // step 0. new node
node<T>* del_node = head;          ⇔        new_node->next = head;               // step 1. fetch head
head = del_node->next;             ⇔        head = new_node;                     // step 2. update head
delete del_node;                                                                 // step 3. delete node
```

Now two threads try to `pop` at the same time. Suppose this is the time-interleaving :

```
thread0      thread1
step1       ─────────────→   D    →  C    →  B  →  A  →  nullptr
                             head
                             del0

             step1           D    →  C    →  B  →  A  →  nullptr
             step2  ─────→   del0    head
                             del1

step2                        XX      C    →  B  →  A  →  nullptr
step3        step3  ─────→   del0    head
                             del1 (XX stands for double deletion)
```

At a result, node `D` is deleted twice.

Solution for pop

In order to solve the problem, we try *CAS*, which gives a symmetrical implementation to `push`.

```
template<typename T> struct lockfree_stack
{
    void pop()
    {
        node<T>* del_node = head.load();                                // step 1. fetch head
        while(!head.compare_exchange_weak(del_node, del_node->next));   // step 2. update head
        delete del_node;                                                // step 3. delete node
    }

    std::atomic<node<T>*> head = nullptr;                               // Don't forget to make head atomic
};
```

Now `head` movement and deletion are consistent.

```
thread0      thread1
step1       ─────────────→   D    →  C    →  B  →  A  →  nullptr
                             head
                             del0

             step1           D    →  C    →  B  →  A  →  nullptr
             step2  ─────→           head
                             del1

step2                        D    →  X       B  →  A  →  nullptr
step3       ─────────────→                   head
                             del1    del0 (CAS fails and reloads del0)

                             X       X       B  →  A  →  nullptr
             step3  ─────→                   head
                             del1    del0
```

3. Remark for pop – Reclamation instead of deletion

Consider `thread0` completing `step1` is preempted by `thread1`, which runs all the way to `step3`, there is a potential hazard when `thread0` resumes execution of `step2` and calls *CAS* with `del_node->next`, a dangling pointer `del_node = D` is dereferenced, resulting in crash. The solution is to replace `delete` with `reclaim`. **Reclamation** postpones deletion until it is safe to do so, i.e. when the last popping thread finishes dereferencing `del_node`. Various reclamation schemes are introduced to solve this problems :

- reclamation with num-of-threads-inside-critical-session counter — not covered here
- reclamation with hazard pointer — not covered here
- reclamation with `std::shared_ptr<T>` — please see part B2.3
- reclamation with double reference counter — please see part B2.4

4. Remark for pop – Return value of popped node

There is no `top` function for lockfree stack (otherwise we need to handle race among `push`, `pop` and `top` making it more complicated), we had better return `head` value from `pop`. Thus we have :

```
template<typename T> void lockfree_stack<T>::pop(T& output)                CAS pattern2
{
    node<T>* pop_node = head.load();                                       // step1 : fetch head
    while(!head.compare_exchange_weak(pop_node, pop_node->next));          // step2 : update head
    output = pop_node->value;
    reclaim (pop_node);                                                    // step3 : reclaim node
}
```

If deep copy to `output` throws, then the stack is exception-unsafe as node is popped in `step2` with nothing returned. We should avoid copying `T` by replacing `T node<T>::value` with shared pointer `std::shared_ptr<T> node<T>::ptr`. Besides, we change the *CAS* pattern.

```
template<typename T> std::shared_ptr<T> lockfree_stack<T>::pop()          CAS pattern1
{
    while(true)
    {
        node<T>* pop_node = head.load();                                   // step1 : fetch head
        if (head.compare_exchange_weak(pop_node, pop_node->next))          // step2 : update head
        {
            std::shared_ptr<T> result = pop_node->ptr;
            reclaim(pop_node);                                             // step3 : reclaim node
            return result;
        }
    }
}
```

5. Remark for pop – Handle empty stack

Finally, we need to check empty stack. We also add a destructor for the stack.

```
template<typename T> struct node
{
    node(const T& x) : ptr(std::make_shared<T>(x)) {}
    std::shared_ptr<T> ptr;
    node<T>* next = nullptr;
};

template<typename T> struct lockfree_stack
{
    lockfree_stack<T>::~lockfree_stack() { while(pop()); }

    void push(const T& x)                                                  CAS pattern2
    {
        node<T>* new_node = new node(x);                                   // step0
        new_node->next = head.load();                                      // step1
        while(!head.compare_exchange_weak(new_node->next, new_node));     // step2
    }

    std::shared_ptr<T> pop()                                               CAS pattern1
    {
        while(true)
        {
            node<T>* pop_node = head.load();                               // step1 : fetch head
            if (!pop_node) return std::shared_ptr<T>();                    // stepA : null check
            if (head.compare_exchange_weak(pop_node, pop_node->next))      // step2 : next head
            {
                std::shared_ptr<T> result = pop_node->ptr;                 // stepB : return output
                reclaim(pop_node);                                         // step3 : reclaim node
                return result;
            }
        }
    }

    std::atomic<node<T>*>    head = nullptr;
};
```

B2.2 Lockfree stack – ABA problem

The last naïve version of lockfree stack in the previous page is ***NOT*** exposed to ABA problem as it performs memory allocation and deallocation (we shall do it later) inside the stack itself. In order to illustrate ABA problem, which is common in lockfree concurrent programming, we modify the lockfree stack as the following, the main difference is delegations of node allocation and deallocation to the caller. After demonstrating ABA problem, we will ***NOT*** use the following implementation again for later sections.

```
// This implementation very similar to that in wikipedia - ABA problem.
template<typename T> struct node
{
    node(const T& x) : ptr(std::make_shared<T>(x)) {}

    std::shared_ptr<T> ptr;
    node<T>* next = nullptr;
};

template<typename T> struct lockfree_stack
{
    void push(node<T>* new_node)
    {
    //  node<T>* new_node = new node(x); // This line is removed, user is responsible for allocating new_node.
        new_node->next = head.load();
        while(!head.compare_exchange_weak(new_node->next, new_node));
    }

    node<T>* pop()
    {
        node<T>* pop_node = head.load();
        while(pop_node && !head.compare_exchange_weak(pop_node, pop_node->next));
        return pop_node;
    }

    std::atomic<node<T>*> head = nullptr;
};
```

Most lockfree containers are based on *CAS* which permits a write to an atomic variable only if the working thread is not preempted by others during its read-modify-write operation, and *CAS* assumes no preemption occur if the value of atomic variable is the same as expected. However this checking is not good enough and *CAS* can be fooled. Consider the following scenario, given stack :

```
nodeC    →    nodeB    →    nodeA    →    nullptr
Head
```

time	thread 1	thread 2	variables
1	`node<T>* n0 = stack.pop()`		`pop_node = nodeC` `pop_node->next = nodeB`
2	`preempted by thread2 right before CAS`	`node<T>* n1 = stack.pop()`	`n1 = nodeC`
3		`node<T>* n2 = stack.pop()`	`n2 = nodeB`
4		`delete n2`	`nodeB is deleted`
5		`stack.push(n1)`	`stack = nodeC → nodeA → nullptr`
6	`resumes and calls ...` `head.CAS(nodeC, nodeB)`		 `thread1 reads dangling pointer and crash`

The root cause of *ABA* problem is that, identical value does not mean no preemption nor nothing has changed. *CAS* can be fooled if a preempting thread performs a series of operations, that change the lockfree container, while keeping the surface value unchanged. The hidden changes may result in a stack with a deleted head. Dereferencing of dangling pointer is resulted, when the invalid head is accessed. *ABA* problem can be avoided by Anthony Williams stack implementation, which execute allocation and deallocation of nodes inside the container.

B2.3 Lockfree stack – reclamation using `std::shared_ptr` 1/4/3x3

In B2.3 and B2.4, reclamation is solved by counting number of threads accessing popped `del_node`.

- B2.3 is done by `std::shared_ptr`, it is may not be lockfree for current *CPU* architecture
- B2.4 is done by double-reference-counting `counted_ptr`

1. Implementation

Here is the implementation using `std::shared_ptr`, main differences are highlighted in red.

```
template<typename T> struct node
{
    node(const T& x) : ptr(std::make_shared<T>(x)) {}

    std::shared_ptr<T> ptr;
    std::shared_ptr<node<T>> next;
};

template<typename T> struct lockfree_stack
{
    std::shared_ptr<node<T>> head;

    void push(const T& x)
    {
        std::shared_ptr<node<T>> new_node = std::make_shared<node<T>>(x);              // step1
        new_node->next = std::atomic_load(&head);                                      // step2
        while(!std::atomic_compare_exchange_weak(&head, &new_node->next, new_node));   // step3
    }

    std::shared_ptr<T> pop()
    {
        while(true)
        {
            std::shared_ptr<node<T>> pop_node = std::atomic_load(&head);               // step1
            if (!pop_node) return std::shared_ptr<T>();                                // stepA
            if (std::atomic_compare_exchange_weak(&head, &pop_node, pop_node->next))   // step2
            {
                std::shared_ptr<T> result = pop_node->ptr;                             // stepB
                return result;                                                         // step3 : reclamation
            }
        }
    }
};
```

2. Design concept

Main ideas are that :

- `shared_ptr<node<T>>` is atomic by itself, but it may not be lockfree
- `shared_ptr<node<T>>` is reclaimed automatically, when reference count falls to zero
- `shared_ptr<node<T>>` is stored or loaded via global function instead of member functions
- passing `shared_ptr<T>` as argument using C-style pointer
- returning `shared_ptr<T>` as output instead of `T*`
- the C++11 syntax is confusing, as the prototype of different for `std::atomic<T>` and `shared_ptr<T>`

```
T                  std::atomic_load(const std::atomic<T>*);
std::shared_ptr<T> std::atomic_load(const std::shared_ptr<T>*);
```

- the C++20 syntax is made consistent (the above syntax will depreciate in C++20) :

```
// std::atomic<std::shared_ptr<T>> is a specialization of std::atomic<U>
template<typename T> struct std::atomic<std::shared_ptr<T>>
{
    void store(std::shared_ptr<T> desired, std::memory_order);
    std::shared_ptr<T> load(std::memory_order) const;
    bool compare_exchange_strong(std::shared_ptr<T>& expected, std::shared_ptr<T> desired, std::memory_order) const;
    bool compare_exchange_weak  (std::shared_ptr<T>& expected, std::shared_ptr<T> desired, std::memory_order) const;
};
```

3. Types of head and next

	B2.1 naïve implementation	B2.3 shared pointer	B2.4 double-reference-count
head node pointer	`std::atomic<node<T>*>`	`std::shared_ptr<node<T>>`	`std::atomic<counted_ptr<T>>`
next node pointer	`node<T>*`	`std::shared_ptr<node<T>>`	`counted_ptr<T>`
data pointer	`std::shared_ptr<T>`	`std::shared_ptr<T>`	`std::shared_ptr<T>`

B2.4 Lockfree stack – reclamation using double reference counting 14321

We have to implement our own lockfree `counted_ptr<T>`, which manages the deallocation of node using double reference counting. Following implementation is lockfree for CPU supporting double CAS, red denotes changes on top of previous implementation.

1. Implementation

```
template<typename T> struct node
{
    node(const T& x) : ptr(std::make_shared<T>(x)) {}

    std::shared_ptr<T> ptr;
    counted_ptr<T> next;
    std::atomic<int> internal_count = 0;
};

template<typename T> struct counted_ptr
{
    counted_ptr(const T& x) : impl(new node<T>(x)) {}

    void dec_count(bool is_CAS_thread)
    {
        if (is_CAS_thread)
        {
            if (impl->internal_count.fetch_sub(external_count-1) == external_count-1) delete impl;
        }
        else
        {
            if (impl->internal_count.fetch_add(1) == -1) delete impl;
        }
    }

    node<T>* impl = nullptr;   } This algo is lockfree only if double word CAS is supported.
    int external_count = 0;
};

counted_ptr<T> fetch_inc_count(std::atomic<counted_ptr<T>>* px) // This method is used again in WRRM hashmap later.
{
    counted_ptr<T> y = px->load();
    counted_ptr<T> z = y;
    ++z.external_count;
    while(!px->compare_exchange_strong(y,z))
    {
        z = y;
        ++z.external_count;
    }
    return z;
}

template<typename T> struct lockfree_stack
{
    ~lockfree_stack() { while(pop()); }

    void push(const T& x)
    {
        counted_ptr<T> new_node(x);                                              // step0 : new node
        new_node.impl->next = head.load();                                       // step1 : fetch head
        while(!head.compare_exchange_weak(new_node.impl->next, new_node));       // step2 : update head
    }

    std::shared_ptr<T> pop()
    {
        while(true)
        {
            counted_ptr<T> pop_node = fetch_inc_count(&head);                    // step1 : fetch head
            counted_ptr<T> tmp_node = pop_node;
            if (!pop_node.impl) return std::shared_ptr<T>();                     // stepA : nullity check
            if (head.compare_exchange_strong(pop_node, pop_node.impl->next))     // step2 : update head
            {
                std::shared_ptr<T> result; result.swap(pop_node.impl->ptr);      // stepB : return output
                pop_node.dec_count(true);                                        // step3 : delete node
                return result;
            }
            else tmp_node.dec_count(false);                                      // step3 : delete node
        }
    }

    std::atomic<counted_ptr<T>> head;
};
```

2. Four functions

- External counter is incremented by global function `fetch_inc_count` taking `std::atomic<counted_ptr<T>>*` as argument.
- Internal counter is incremented by member function `dec_count` taking boolean as argument.
- `lockfree_stack<T>::push` is the same naïve implementation.
- `lockfree_stack<T>::pop` makes a copy of `pop_node` as `tmp_node`.

3. Remark for pop - Three races

Race to fetch head in `step1`

- `external_count` inside atomic `head` counts number of threads calling `pop()` before `head` is *CAS-swapped* successfully
- each thread calling `pop()` has its own local copy `pop_node.external_count`, each has a different value
- latest thread calling `pop()` has its own local copy `pop_node.external_count == head.load().external_count`

Race to update head in `step2`

- only latest thread calling `pop()` can win this race, others have `pop_node.external_count != head.load().external_count`
- once `head.compare_exchange_strong()` is done, next node becomes `head`, `external_count` is freezed
- both *thread succeeded in CAS* and *threads failed in CAS* then race to `dec_count` and quit current `while` loop

Race to reclaim popped node in `step3`

- each *thread failed in CAS* increments `internal_count` by one
- *thread succeeded in CAS* increments `internal_count` by one and decrements `internal_count` by `pop_node.external_count` ...
  as *thread succeeded in CAS* is the only thread having correct `pop_node.external_count` ...
  so *thread succeeded in CAS* is responsible for passing it to `internal_count`
- the last thread quitting current `while` loop (making `internal_count` zero) is responsible for deletion of popped `head` ...
  Why zero? Since `internal_count` is a net count of threads entering plus quitting current `while` loop.

4. Remark for pop - Why two counters (external and internal)?

Firstly we need `counted_ptr<T>::external_count` (as a direct member of `counted_ptr<T>`) for counting the number of threads calling `pop()` to pop the same `head` node. The increment must be done atomically by invoking *CAS* on `std::atomic<counted_ptr<T>> head`, like :

```
auto x = head.load();
auto y = x;
++y.external_count;
head.compare_exchange_strong(x,y);
```

Therefore `external_count` must be declared :

- as a direct member of `counted_ptr<T>` in `line2` (instead of as an indirect member in `line1`)
- as type `int` instead of `std::atomic<int>`, as we don't have `atomic` of `atomic`

Secondly each thread should have a local copy `pop_node` after calling `fetch_inc_count()`, hence we cannot count the number of threads remaining in `pop()` by deducting `pop_node.external_count`, we need another counter, `node<T>::internal_count`, which is declared :

- as a direct member of `node<T>` in `line1` so that it can be accessed by all threads via `impl->internal_count`
- as type `std::atomic<int>` instead of `int` to avoid race condition among all threads

5. Remark for pop - Transfer external count to internal count

Now we have two counters one external and one internal, now we need a promising way to transfer `external_count` to `internal_count`. the thread succeeding *CAS* in `step2` is called *CAS thread*, it is the only thread with `pop_node.external_count==head.load().external_count`, thus *CAS thread* is responsible for transferring `external_count` to `internal_count` through atomic increment or decrement. Hence :

- *non CAS thread* increments `impl->internal_count` by one on quiting current `while` loop
- *CAS thread* increments `impl->internal_count` by one minus `external_count` on quiting current `while` loop

Whenever resulting value of `impl->internal_count` becomes zero, it implies that the thread is the last thread quitting `while` loop and it is responsible for deleting popped node. However for `fetch_add()` or `fetch_sub()`, pre-add or pre-sub (instead of post-add or post-sub) value is returned, thus the comparison is not done against zero :

- if `impl->internal_count.fetch_add(1)` returns `-1` for *non CAS thread*, the thread must `delete pop_node`
- if `impl->internal_count.fetch_add(1-external_count)` returns `external_count-1` for *CAS thread*, the thread `delete pop_node`

Illustration with a pop example

Consider 7 threads calling `pop` concurrently. The following shows how different variables evolve through time :

```
T1   = thread1::pop_node.external_count // different threads has different external_count
T2   = thread2::pop_node.external_count // only latest thread has correct external_count
...

C.ex = nodeC::external_count
C.in = nodeC::impl->internal_count.load()
B.ex = nodeB::external_count
B.in = nodeB::impl->internal_count.load()
```

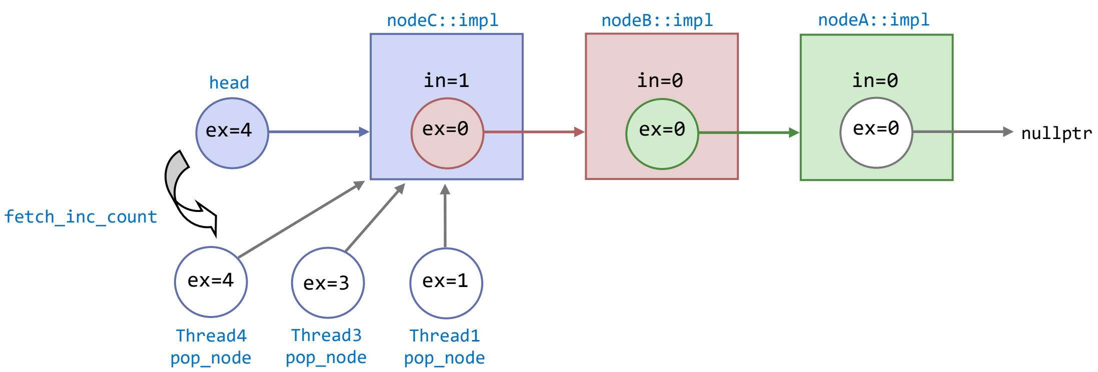
time axis	T1	T2	T3	T4	C.ex	C.in
	x	x	x	x	0	0
T1 fetch	1	x	x	x	1	0
T2 fetch	1	2	x	x	2	0
T3 fetch	1	2	3	x	3	0
T2 CAS-fail	1	2	3	x	3	0
T2 quit	1	x	3	x	3	1
T4 fetch	1	x	3	4	4	1
T4 CAS-OK	1	x	3	4*	4	1
T3 quit	1	x	x	4*	4	2
T1 quit	x	x	x	4*	4	3
T4 quit	x	x	x	del	4	0

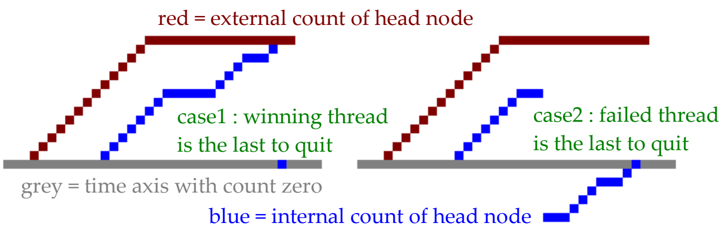
Take snapshot here

time axis	T5	T6	T7	T1	T2	T3	B.ex	B.in
	x	x	x	x	x	x	0	0
	x	x	x	x	x	x	0	0
	x	x	x	x	x	x	0	0
	x	x	x	x	x	x	0	0
	x	x	x	x	x	x	0	0
	x	x	x	x	x	x	0	0
	x	x	x	x	x	x	0	0
	x	x	x	x	x	x	0	0
T5 fetch	1	x	x	x	x	x	1	0
T6 fetch	1	2	x	x	x	x	2	0
T7 fetch	1	2	3	x	x	x	3	0
T1 fetch	1	2	3	4	x	x	4	0
T7 CAS-fail	1	2	3	4	x	x	4	0
T7 quit	1	2	x	4	x	x	4	1
T2 fetch	1	2	x	4	5	x	5	1
T3 fetch	1	2	x	4	5	6	6	1
T3 CAS-OK	1	2	x	4	5	6*	6	1
T1 quit	1	2	x	x	5	6*	6	2
T3 quit	1	2	x	x	5	x	6	-3
T5 quit	x	2	x	x	5	x	6	-2
T6 quit	x	x	x	x	5	x	6	-1
T2 quit	x	x	x	x	del	x	6	0

`external_count` is freezed

circle denotes `counted_ptr<T>`
arrow denotes `node<T>*`
square denotes `node<T>`





B3. Lockfree bounded MPMC queue

by Dmitry Vyukov in blog 1024 cores

1. Naive implementation

Implemented with bound-sized array

- contiguous memory which is cache friendly
- use of stack memory, no heap memory allocation / deallocation
- size must be 2^N, modulus is performed using bitwise *AND* with `mask=N-1` which is all-one in binary
- memory alignment and additional padding to avoid false sharing

Lockfree but not waitfree (undone for retry)

- atomic `next_write` resolves race between two producers
- atomic `next_read` resolves race between two consumers
- atomic `cell::flag` in ***publication pattern*** resolves race between producer and consumer
- ownership of dequeued element must be moved to `x` before resetting `array[m].flag`, otherwise producers will overwrite

Function `dequeue` involves two move semantics. We have to constraint using `std::enable_if` :

- `T` to be move constructible and
- `N` to be power of 2 to implement modulus using bitwise AND with `mask`

```
template<typename T> struct cell
{
    alignas(64) T value;
    std::atomic<bool> flag; // normally, value and flag are accessed by the same flag, no need to alignas(64) flag
};

template<typename T, size_t N=1024, typename = std::enable_if_t<std::is_move_constructible<T>::Value && (N & N-1)==0>>
struct mpmc_queue final
{
    mpmc_queue() : next_write(0), next_read(0)
    {
        for(int n=0; n!=size; ++n) array[n].flag.store(false);
    }

    // *** invoked by producer (using section A1-3.2 CAS pattern1) *** //
    template<typename... ARGGS> bool enqueue(ARGS&&... args) noexcept
    {
        while(true)
        {
            int n = next_write.load();
            int m = n & mask;

            if (array[m].flag.load()) return false;               // std::this_thread over produces
            if (next_write.compare_exchange_strong(n,n+1))         // std::this_thread wins producer-producer race
            {
                new (&array[m].value) T{std::forward(args)...};    // placement new invoking constructor
                array[m].flag.store(true);
                return true;
            }
            else continue;                                         // std::this_thread loses producer-producer race
        }
    }

    // *** invoked by consumer (using section A1-3.2 CAS pattern1) *** //
    std::optional<T> dequeue()
    {
        while(true)
        {
fetch head  int n = next_read.load();
            int m = n & mask;

            if (!array[m].flag.load()) return std::nullopt;        // std::this_thread over consumes
update head if (next_read.compare_exchange_strong(n,n+1))          // std::this_thread wins consumer-consumer race
            {
                std::optional<T> x = std::make_optional(std::move(array[m].value));
                array[m].flag.store(false);
                return x;
            }
            else continue;                                         // std::this_thread loses consumer-consumer race
        }
    }

    std::array<cell<T>, N> array;
    alignas(64) constexpr int size = N;
    alignas(64) constexpr int mask = N-1;
    alignas(64) std::atomic<int> next_write;
    alignas(64) std::atomic<int> next_read;
    alignas(64) char pad;
};
```

Apparently, this implementation seems to work. However there is a bug in case of producer-producer race. With further testing :
1. by taking snapshot of all `cell<T>::flag` in `mpmcq`, pattern like `TFFTTFTF` can be observed, is this expected instead of (`TTTFFFTT`) ?
2. three behaviours which thought to be originated from same bug can be observed :
- valgrind detected memory-access without initialization
- the executable may crash with segmentation fault
- the executable may have one out of M threads stuck in infinity loop (thus it cannot be joined)

1. Pattern like `TFFTTFTF` is expected.
- Consider the case when `N=8`, `next_write=9` and `next_read=6`, now the `mpmcq` looks like `TFFFFFTT`, with size `3` ...
- producer `A` claimed `m=1` (i.e. succeeded the *CAS*) but before it can set `array[1].flag`
- producer `B/C` claimed `m=2/3` respectively, if `C` preempts `A/B` and set `array[3].flag`
- producer `C` then runs all the way to claim next element, resulting in `TFFTTFTT`
- similar things happen in consumer-consumer race
- consumer `E` claimed `m=6`, while consumer `F` catches up to claim `m=7` and reset `array[7].flag`, resulting in `TFFTTFTF`

2. Uninitialized memory / crash / infinity loop are all related to the bug in producer-producer (or consumer-consumer) race.
- Consider the case when `N=8`, `next_write=9` and `next_read=9`, now the `mpmcq` looks like `FFFFFFFF`, with size `0` ...
- producer `A` claimed `m=1` (i.e. succeeded the *CAS*) but before it can set `array[1].flag`, it is preempted by producer `B`
- producer `B` claimed `m=2/3/.../7/0`, set all those `array[m].flag` true, at this moment `next_write=17` and `next_read=9`
- producer `B` gets `m=1`, observed that `array[1]` is empty (as producer `A` has not updated `array[1].flag` yet)
- producer `B` then writes to `array[1]` before producer `A` does, resulting in size `9` which is greater than `N=8`

2. Revised implementation

The root cause of this problem is that the binary state by `cell<T>::flag` is not good enough to safe guard producer-producer race nor consumer-consumer race. This is an example of ABA problem. To solve it, we add more states into the `cell<T>` like the following :

```
template<typename T> struct cell
{
    alignas(64) T value;
    std::atomic<std::uint32_t> state;
};
```

Now we extend the state of `cell[m]` where `m∈[0,N)`, so that it can indicate / differentiate the following states :
- `cell[m]` is empty now, waiting to fill the (`m)th` element — denoted as `cell[m].state = m`
- `cell[m]` is filled with the `(m)th` element — denoted as `cell[m].state = m+1`
- `cell[m]` is empty now, waiting to fill the (`m+N)th` element — denoted as `cell[m].state = m+N`
- `cell[m]` is filled with the `(m+N)th` element — denoted as `cell[m].state = m+N+1`
- `cell[m]` is empty now, waiting to fill the (`m+2N)th` element — denoted as `cell[m].state = m+2N`
- `cell[m]` is filled with the `(m+2N)th` element — denoted as `cell[m].state = m+2N+1`
- and so on ...
- thus there are infinite possible states, we need an integer to represent the state (as shown by red figures above)
- note there are `N-2` unused state under this design (thus it can be applied to disruptor with `N-1` processors)

Resolving the races :
- atomic `next_write` resolves race between two producers racing for next cell
- atomic `next_read` resolves race between two consumers racing for next cell
- atomic `cell::state` resolves race between :
  - producer vs consumer race, and also
  - producer vs *one-cycle-ahead-producer* race
  - consumer vs *one-cycle-ahead-consumer* race
- atomic `next_write` does not interact with atomic `next_read`, instead, both interact with atomic `cell::state`
- atomic `cell::state` is also known as a barrier

Inside the main logic of `enqueue` and `dequeue`, we compare the state of the cell, with expected values (as shown by red figures above).

```
    template<typename T, size_t N=1024, typename = std::enable_if_t<std::is_move_constructible<T>::Value && (N & N-1)==0>>
    struct mpmc_queue final
    {
        mpmc_queue() : next_write(0), next_read(0)
        {
            for(int n=0; n!=size; ++n) array[n].state.store(n);
        }

        template<typename ARGS...> STATUS enqueue(ARGS&&... args)
        {
            while(true)
            {
                int n = next_write.load();
                int m = n & mask;
                int s = array[m].state.load();

1.              if (s < n) return OVER_PRODUCE; // Regarding to above example, this line prevents producer B from entering CAS
                else if (s == n)
                {
2.                  if (next_write.compare_exchange_strong(n,n+1)
                    {
                        new (&array[m].value) T{std::forward(args)...};
                        array[m].state.store(n+1);
                        return OK;
                    }
3.                  else continue; // std::this_thread retry on losing PP race, winner has completed CAS, but not state.store
                }
4.              else continue; // std::this_thread retry on losing PP race, winner has completed both CAS and state.store
            }
        }

        std::optional<T> dequeue()
        {
            while(true)
            {
                int n = next_read.load();
                int m = n & mask;
                int s = array[m].state.load();

1.              if (s < n+1) return std::nullopt; // Regarding to above example, this line prevents consumer B from entering CAS
                else if (s == n+1)
                {
2.                  if (next_read.compare_exchange_strong(n,n+1))
                    {
                        std::optional<T> x = std::make_optional(std::move(array[m].value));
                        array[m].state.store(n+N);
                        return x;
                    }
3.                  else continue; // std::this_thread retry on losing CC race, winner has completed CAS, but not state.store
                }
4.              else continue; // std::this_thread retry on losing CC race, winner has completed both CAS and state.store
            }
        }

        std::uint32_t peek_size() const
        {
            return next_write.load()-next_read.load();
        }
    };
```

For both `enqueue` and `dequeue`, there are 4 cases. Now, consider the former.

1. before producer tries to claim `n=9` and `m=1`, however `n=1` is not consumed yet, it should return to caller without retry
2. noting that `n=1` has been consumed, producer is claiming `n=9` and `m=1`, and succeeds
3. noting that `n=1` has been consumed, producer is claiming `n=9` and `m=1`, but fails, it should retry
4. before producer tries to claim `n=9` and `m=1`, someone preempted, filled and leads by one complete cycle

In other words :	i.e. for producer
1. implies that current thread is too fast	when `s=n-N, n-N+1` but not `n-2N, n-2N+1, ...`
2. implies that current thread is just fast enough to succeed	when `s=n`
3. implies that current thread is just a little but slow	when `s=n`
4. implies that current thread is too slow	when `s=n+1, n+N` but not `n+N+1, n+2N, n+2N+1, ...`

Speed test is done to measure the absolute latency. We do the following for reliable measurement :

- build test program with release mode `cmake -DCMAKE_BUILD_TYPE=Release ..`
- set cpu frequency to highest `cpufreq-set -d 4.5GHz`
- use `clock_gettime()` monotonic mode to get timetamp (timer resolution is 15ns in my machine)
- 10K messages are sent from producers to consumers, with 100 us interval, absolute latency is measured

Here is the latency in nano second (percentile) for different implementations :

	percentile	0%	1%	10%	25%	50%	75%	90%	99%	100%
1P1C	locked STL	143	168	191	208	255	3198	>delay	>delay	>delay
	locked cbuf	125	161	169	193	204	4516	>delay	>delay	>delay
	lfree mpmcq	92	116	121	137	154	170	191	311	4975
	lfree spscq	97	113	117	120	124	154	163	207	22394
2P2C	locked STL	199	259	2337	16563	58976	>delay	>delay	>delay	>delay
	locked cbuf	192	243	733	8126	33657	81286	>delay	>delay	>delay
	lfree mpmcq	105	131	142	155	170	190	205	246	1904
3P3C	locked STL	161	276	500	963	2941	12979	31989	>delay	>delay
	locked cbuf	111	255	413	624	1379	5870	>delay	>delay	>delay
	lfree mpmcq	104	126	139	149	167	185	209	270	425

I have also compared our lockfree `mpmcq` with `boost::lockfree::queue`, with slightly different setting from above. Still 3P3C. What are the differences in testing making such a difference between `mpmcq` above and `mpmcq` below? Please check code in `YExperiement` and code in `threadpool`.

	percentile	0.1%	1%	10%	25%	50%	75%	90%	99%	99.9%
3P3C	lfree mpmcq	72	80	86	88	93	98	104	113	135
	boost::lfree	167	174	184	197	212	231	250	283	307

The following items must be exist to achieve the sub-100-nano latency :

- set thread affinity for each consumer and producer thread using `pthread_setaffinity_np`
- set thread nice value for each consumer and producer thread using `setpriority`
- set thread policy and priority for consumer (but ***NOT*** producer) using `pthread_setschedparam`
  setting thread policy `FIFO` for producer will end up drawing all resource and program hanged
- no sleeping is needed in consumer loop, so that they can response with minimum latency
- sleeping is needed in producer loop, to avoid too much contention, suggestion `std::chrono::nanoseconds(100)`

3. Extension to disruptor

In *MPMCQ* there is only one `next_write` shared by multiple producers and only one `next_read` shared by multiple consumers. Unlike *MPMCQ* although disruptor has only one `next_produce` shared by multiple producers, it has multiple `next_process` shared by multiple processors. Each cell should be processed by each processor once and once only. There is no constraints on the visiting order by the processors. Therefore disruptor can be considered as a generic version *MPMCQ*, whereas disruptor can be extended by introducing order constraint on the processors. Now suppose there are *K* processors ...

```
template<typename T, size_t N=1024, typename = std::enable_if_t<std::is_move_constructible<T>::Value && (N & N-1)==0>>
struct disruptor final
{
    template<typename ARGS...> STATUS enqueue(ARGS&&... args)
    {
        while(true)
        {
            int n = next_write.load();
            int m = n & mask;
            int s = array[m].state.load();

            if (s < n) return OVER_PRODUCE;
            else if (s == n)
            {
                if (next_write.compare_exchange_strong(n,n+1)
                {
                    new (&array[m].value) T{std::forward(args)...};
                    array[m].state.store(n+1);
                    return OK;
                }
                else continue;
            }
            else continue;
        }
    }

    std::optional<T> visit(int processor_id) // processor_id = [0,K)
    {
        while(true)
        {
            int n = next_read[processor_id].load();
            int m = n & mask;
            int s = array[m].state.load();

            if (s < n+1) return std::nullopt;
            else if (s <= n+K)
            {
                if (next_read[processor_id].compare_exchange_strong(n,n+1))
                {
                    std::optional<T> x = std::make_optional(array[m].value); // copy, not move
                    auto s_old = array[m].state.fetch_add(1);

                    // When K==1, this if-block vanishes, disruptor then degenerates into a MPMCQ.
                    if (s_old == n+K)
                    {
                        array[m].state.fetch_add(N-K-1);
                    }
                    return x;
                }
                else continue;
            }
            else continue;
        }
    }
};
```

B4. Lockfree hashmap

	B5.1	B5.2	B5.3
generic key type	*yes*	no	*yes*
generic value type	*yes*	no	no
with hash function	*yes*	no	*yes*
with probing scheme	no	*yes*	*yes*
multithread and lockfree	no	*yes*	*yes*
reduce number of CAS	no	no	*yes*

B4.1 Single thread hashmap

by Jeff Preshing in blog Preshing on Programming

1. What is a hashmap?

- Hashmap is an unordered [key, value] map targetting to achieve *O(1)* search time
- implemented as array, with contiguous memory, cache friendly, no memory allocation
- hash function is a ***shuffle function*** which maps any key into a hashed-key, which is an integer
- if hashed-key goes outside bound, find bucket-index by taking modulus of bucket-number
- if no collision happens, *O(1)* search time is achieved
- if collision happens (i.e. various keys sharing same bucket-index), resolve by separate chaining or open addressing.

2. Separate chaining

Separate chaining solves collision by appending a list of [key, value] or [hashed-key, value] pairs to each bucket.

```
template<typename K, typename V> struct cell { int hashed_key; K key; V value; };
template<typename K, typename V> struct hash_table
{
    void set(const K& key, const V& value)
    {
        int hashed_key = hash_fct(key);
        int index      = hashed_key & mask;
        for(auto& x : buckets[index])
        {
            if (x.hashed_key == hashed_key) { x.value = value; return; }
            if (x.key        == key)        { x.value = value; return; }
        }
        buckets[index].insert(cell<K,V>{hashed_key, key, value});
    }

    std::optional<V> get(const K& key) const
    {
        int hashed_key = hash_fct(key);
        int index      = hashed_key & mask;
        for(auto& x : buckets[index])
        {
            if (x.hashed_key == hashed_key) { return std::make_optional<V>{x.value}; }
            if (x.key        == key)        { return std::make_optional<V>{x.value}; }
        }
        return std::nullopt;
    }

    static const int size = 1024;
    static const int mask = size-1;
    std::list<cell<K,V>> buckets[size]; // Each bucket is a list.
};
```

3. Open addressing

On collision, we jump to next available bucket according to certain probing schemes. Open addressing is more cache-friendly.

Given	`hashed_key       =  hash_fct(key)`	
and also	`bucket_index     =  hashed_key % bucket_number`	
⇒ linear probing	`probe_chain_iter = (bucket_index + n) % bucket_number`	`for n=0,1,2,...`
⇒ quadratic probing	`probe_chain_iter = (bucket_index + n*n) % bucket_number`	`for n=0,1,2,...`
⇒ leapfrog probing	`probe_chain_iter = (bucket_index + leapfrog(n)) % bucket_number`	`for n=0,1,2,...`

B4.2 Lockfree array

This is a simple array without hash function, which is simply used to illustrate the lockfree technique in linear probing. In the array, both key and value are integers, being made atomic to be thread safe, take `EMPTY` value when uninitialized. Without a hash function, all keys are hashed to zero, from which linear probing starts. As a result, all filled cells locate consecutively at the front,whereas the first `EMPTY` cell denotes the `end()` iterator of the array.

1. Implementation

```
key      13      12      15      10      14      EMPTY   EMPTY   EMPTY   ...
value    666     42      1       314     1000    EMPTY   EMPTY   EMPTY   ...

struct cell
{
    std::atomic<int> key   = EMPTY;
    std::atomic<int> value = EMPTY;
};

struct lockfree_array
{
    STATUS set(int key, int value)
    {
        for(int n=0; n!=size; ++n)
        {
            int expected = EMPTY;
            buckets[n].key.compare_exchange_strong(expected, key, std::memory_order_relaxed);

            if      (expected == EMPTY) { buckets[n].value.store(value, std::memory_order_relaxed); return OK; }
            else if (expected == key)   { buckets[n].value.store(value, std::memory_order_relaxed); return OK; }
            else continue;
        }
        return TABLE_IS_FULL;
    }

    std::optional<int> get(int key)
    {
        for(int n=0; n!=size; ++n)
        {
            int tmp = buckets[n].key.load(std::memory_order_relaxed);
            if (tmp == EMPTY)   return std::nullopt;
            if (tmp == key)     return std::optional<int>(buckets[n].value.load(std::memory_order_relaxed));
        }
        return std::nullopt;
    }

    static const int size = 1024;
    static const int mask = size-1;
    cell buckets[size];
};
```

2. Four cases

There are 4 possible outcomes for `set` function :

- `CAS done && expected == EMPTY` → insert the new entry
- `CAS fail && expected == key` → update the new entry
- `CAS fail && expected != key` → search for new next entry
- container is full, no key is matched → return error

3. Four orderings

Option `std::memory_order_relaxed` is fine. When consumer loads a cell that is being stored by producer, there are 4 valid outcomes :

- `buckets[n] = (EMPTY,EMPTY)` → considered as end of hash table and consumer returns `std::nullopt`
- `buckets[n] = (key,EMPTY)` → cell half-stored, consumer returns `std::nullopt`
- `buckets[n] = (key,value)` → cell fully-stored, consumer returns `value`
- `buckets[n] = (EMPTY,value)` → reordering due to unflushed store-buffer, consumer returns `std::nullopt`

B4.3 Lockfree hashmap

1. What is it?

On top of the lockfree array in B4.2, we build a lockfree hashmap, such that :

- support generic key
- support `int` value
- support hash function (MurmurHash3)
- support probing (modulus using bitwise *AND*)
- lockfree like B4.2
- reduce the number of *CAS* call

There are problems with this implementation :

- *we must keep original* `key` *in* `cell` *as hash function does not guarantee 1 to 1 mapping*
- *we wish to support* `MPMC` *with generic key and generic value though key must be* ***immutable****, we hope value to be* ***mutable***
- *mutable value means it has to handle producer-producer race as well*

This ideal lockfree hashmap cannot be achieved, here are possible tradeoffs :

- `SPMC_hashmap<K,V>` *using publication pattern, however* `V` *is immutable*
- `SPMC_hashmap<K,int>` *in which* `V` *is mutable*
- `MPMC_hashmap<int,int>` *in which* `V` *is mutable*

2. Implementation

```
struct cell
{
    std::atomic<int> hashed_key = EMPTY; // Use hashed_key (NOT key) as we can't declare atomic<K>.
    std::atomic<int> value;
};

int probing(int n) { return n;     }
int probing(int n) { return n * n; }

template<typename K> struct lockfree_hash
{
    STATUS set(const K& key, int value)
    {
        // (1) hash-function
        int hashed_key = murmurhash3(key);

        // (2) probing
        for(int n=0; n!=size; ++n)
        {
            int index = (hashed_key + probing(n)) & mask;

            // (3) compare-key
            int tmp = buckets[index].hashed_key.load(); // A "DCLP-liked" trick to reduce chance of slow CAS.
            if (tmp == EMPTY)
            {
                int expected = EMPTY;

                buckets[index].hashed_key.compare_exchange_strong(expected, hashed_key);
                if (expected == EMPTY ||
                    expected == hashed_key)
                {
                    buckets[index].value.store(value);
                    return OK;
                }
            }
            else if (tmp == hashed_key)
            {
                buckets[index].value.store(value);
                return OK;
            }
        }
        return TABLE_IS_FULL;
    }

    std::optional<V> get(const K& key)
    {
        // (1) hash-function
        int hashed_key = murmurhash3(key);

        // (2) probing
        for(int n=0; n!=size; ++n)
        {
            int index = (hashed_key + probing(n)) & mask;

            // (3) compare-key
            int tmp = buckets[index].hashed_key.load();
            if (tmp == EMPTY)      return std::nullopt;
            if (tmp == hashed_key) return std::optional<V>(bucket[index].value.load());
        }
        return std::nullopt;
    }

    static const int size = 1024;
    static const int mask = size-1;
    cell<V> buckets[size];
};
```

This part is the same as lockfree array.

B5. Lockfree write-rarely-read-many hashmap (WRRM hashmap)
by Andrei Alexandrescu, in Dr Dobb blog

1. What is it?

This is a technique for wrapping any `STL` containers to become a lockfree thread-safe container for write-rare-read-many purpose.

- underlying `STL` container is wrapped with `counted_ptr`, which is in turn wrapped with `std::atomic<counted_ptr>`
- counting number of accesses (read plus write) to underlying `STL` container
- reading is implemented as atomic `load()` followed by underlying `STL` container `get()`
- writing is implemented as replacement of underlying `STL` container when reference count equals one, called the migration

2. Implementation

```
template<typename K, typename V> struct wrrm_hash
{
    struct counted_ptr
    {
        std::unordered_map<K,V>* impl;
        int count = 0;
    };

    std::atomic<counted_ptr> root;

    counted_ptr fetch_inc()
    {
        counted_ptr x = root.load();
        counted_ptr y = x;
        ++y.count;
        while(!root.compare_exchange_strong(x, y))
        {
            y = x;
            ++y.count;
        }
        return new_hash;
    }

    counted_ptr fetch_dec()
    {
        counted_ptr x = root.load();
        counted_ptr y = x;
        --y.count;
        while(!root.compare_exchange_strong(x, y))
        {
            y = x;
            --y.count;
        }
        return y;
    }

    counted_ptr deep_copy_and_insert(counted_ptr x, const K& key, const V& value)
    {
        counted_ptr y;
        y.impl  = new std::unordered_map<K,V>(*x.impl);
        y.count = x.count;
        (*y.impl)[key] = value;
        return y;
    }

    // *** get and set function *** //
    void set(const K& key, const V& value)
    {
        counted_ptr x = fetch_inc(); x.count = 1;
        counted_ptr y = deep_copy_and_insert(x, key, value);
        // CAS succeeds only when count = 1, i.e. std::this_thread is the only one calling set()
        while(!root.compare_exchange_strong(x, y))
        {
            x.count = 1;
            delete y.impl;
            y = deep_copy_and_insert(x, key, value);
        }
        fetch_dec();
        delete x.impl;
    }

    std::optional<V> get(const K& key)
    {
        std::optional<V> result;

        auto x = fetch_inc();
        auto iter = x.impl->find(key);
        if  (iter!= x.impl->end()) result = std::optional<V>(iter->second);
        fetch_dec();
        return result;
    }
};
```

3 different usages of `count` :
- increment by 1
- decrement by 1
- constrainted at 1