Lock-free programming: MPSC queue

Donnerstag, 6. Februar 2025 09:16

Author: Damir Ljubic
email: damirlj@yahoo.com

Introduction

I'm keep saying - beside the software architecture and design, as most fundamental, entry point of every software development, the alphabet of the programming that everyone should master: is **asynchronous programming.**
I would even dare to say - the most of the time in the implementation phase I'm spending dealing with all kind of "*asynchronicity*", and how to solve these challenges within the given constraints: available hardware, toolchain, framework, even language.

From the language perspective, the C++ influenced by the other languages and libraries made a significant effort to provide the mechanisms for concurrency as part of the standard library itself, starting with a thread abstraction, synchronization primitives and a new memory model - all introduced with C++11.
These enabled us to address concurrency topics from different, non-conventional point of view: writing **lock-free** (even wait-free) code, for potentially overcoming the lock-based approach main issues:
→ The performance and dead-locks.
Potentially because the operations on atomic types (atomic operations) are notably more expensive then the same operations on the non-atomic types, and may even involve the locking mechanism.
There is also *Cashe coherence* involved - synchronization between the cores hosting the threads synchronized around the same atomics, etc. - but these are usually the minor issues on contemporary platforms.

The main pre-warning for lock-free programming is that it's difficult to write it correctly.
→ *What does it mean?*

With lock-based programming - we have **a strong thread-safety guaranties**, that our Jailkeeper will not left any between inmates shared rooms without monitoring, as long as there is possibility for the prison brake: as long as there is possibility for the race condition and undefined behavior.
With lock-free programming, we are joggling with the trade-safety guaranties, being much more "*relaxed*".
We are talking about the ***memory ordering*** and the ***visibility*** of the same code in multithread context - non-atomic operations around the atomic variables used for synchronizing the multiple threads.
→ More on that: https://github.com/damirlj/modern_cpp_tutorials/blob/main/docs/memory/Lock-free%20programming%2C%20part%201.pdf

This allows us more flexibility - especially designing the lock-free structures so that they reflect certain use-cases, certain scenarios where we conditionally can lowering these expectations on a thread-safety: not being so rigid Jailkeeper, but rather the one who closes one eye on race conditions - and still maintains the order, having all prisoners on count.

Enough with the prisons analogies.
Let's examine some concreate implementation.

MPSC - Multiple Producers Single Consumer lock-free queue

Relaxing these thread-safety guaranties we simplify the implementation of the lock-free structures, that will be tailored to work correctly - in line with expectations only under certain circumstances.
The most simple one, in terms of the lock-free queue would be Single Producer Single Consumer implementation.

In the *lock-based* implementation, for bounded queue, we would have a std::array as memory storage and two semaphores (*std::counting_semaphore*), where the internal counter for the writing semaphore is initially set to the number of empty slots in memory storage, while the read semaphore will be initially set to 0 - as indication of the actual number of elements.
The producer acquires the write semaphore - that will block if there is no empty slots, otherwise the producer will write in the first empty slot and notify the consumer - by releasing the read semaphore.
The consumer will acquire the read semaphore - and wait if the internal counter is 0.
Otherwise, it will proceed - pop the queue and release the write semaphore.

→ Possible implementation: https://github.com/damirlj/modern_cpp_tutorials/tree/main/src/ring%20buffer

For *lock-free* version of the queue, we would use *std::atomic<std::size_t>* - to represent the write/read index

```
alignas(64) std::array<T, N> data_;
alignas(64) std::atomic<std::size_t> head_ {0};
alignas(64) std::atomic<std::size_t> tail_ {0};
```

Assuming that producer will ever call *push* method, for SPSC implementation we could
write something like

```
template <typename U>
requires std::convertible_to<U, value_type>
bool try_push(U&& u) noexcept(std::is_nothrow_constructible_v<U>)
{
    const auto tail = tail_.load(std::memory_order_relaxed); // maintain by the single producer
    const auto head = head_.load(std::memory_order_acquire); // maintain by the single consumer
    if ( ((tail + 1) & (N - 1)) == head) return false; // queue is full

    data_[tail] = std::forward<U>(u);
    tail_.store((tail + 1) & (N - 1), std::memory_order_release);// signal write event to consumer

    return true;
}
```

For MPSC scenario, we need to take into consideration that multiple producers can advance the tail index simultaneously,
so we need to check the "*full buffer*" condition repeatedly

```
template <typename U>
requires std::convertible_to<U, value_type>
bool push(U&& u) noexcept(std::is_nothrow_constructible_v<U>)
{
    // expected value - otherwise, another producer modifies it
    auto tail = tail_.load(std::memory_order_relaxed);
    while (is_full() || not tail_.compare_exchange_weak(tail, (tail + 1) & (N - 1), std::memory_order_release));

    data_[tail] =std::forward<U>(u);

    return true;
}
```

The consumer part remains the same, in terms of the *dequeuing*, since we have a single consumer
We have a private utility method

```
template<typename Func, typename...Args>
requires std::invocable<Func, std::size_t, Args...> && std::is_same_v<bool, std::invoke_result_t<Func,
std::size_t, Args...>>
inline std::optional<value_type> pop(Func&& func, Args&&...args)
noexcept(std::is_nothrow_move_constructible_v<value_type>)
{
    const auto head = head_.load(std::memory_order_relaxed);

    if(not std::invoke(std::forward<Func>(func), head, std::forward<Args>(args)...)) return {};

    auto data = std::optional<value_type>(std::move(data_[head]));

    head_.store((head + 1) & (N - 1), std::memory_order_release);

    return data;
}
```

We obtain the head index maintained within the single, consumer-thread (relaxed), and in case that the given condition
is fulfilled - the data in buffer on the given slog will be moved, and afterwards the head index will be advanced - with
release memory barrier: to synchronize with producers happens-before relationship, as they are checking "*full buffer*"
condition being fulfilled.
We have, similar as with enqueuing mechanism, different variants of *pop* method

try_pop() : try to dequeue the element from the queue, if the queue is not empty

```
auto try_pop() -> std::optional<value_type>
{
    return pop([this](std::size_t head) {return not is_empty(head); });
}
```

For the waiting dequeue operations, until the event is moved, or the stop token is signaled

```
auto pop_wait(const std::atomic_flag& stop) ->std::optional<value_type>
{
    return pop([&stop, this](std::size_t head)
                {
                    while (is_empty(head)) // wait until is non-empty, or stop is signaled
```

```
                    {
                        if (stop.test(std::memory_order_relaxed)) return false;
                        std::this_thread::yield();
                    }
                    return true;
                });
}
```

The same as previous, but with additional exit criteria: timeout is expired

```
auto pop_wait_for(const std::atomic_flag& stop, std::chrono::milliseconds timeout) ->std::optional<value_type>
{
     return pop([&stop, timeout, this](std::size_t head)
                {
                    using namespace std::chrono;
                    auto start = steady_clock::now();

                    while (is_empty(head)) // wait until is non-empty, stop is signaled, or timeout expired
                    {
                        if (stop.test(std::memory_order_relaxed)) return false;
                        if (duration_cast<milliseconds>(steady_clock::now() - start) > timeout) return false;
                        std::this_thread::yield();
                    }
                    return true;
                });
}
```

For more details, check the entire implementation

<**Compiler Explorer**>: https://godbolt.org/z/8T4vqrWoE