
LEVEL 3

```cpp
template<typename T>
class drow_mpmc_v6
{
public:
    drow_mpmc_v6 (size_t capacity_)
        : capacity (std::bit_ceil (capacity_)),
          slots (capacity)
    {
        // Initialise each slot's sequence number to its index
        for (size_t i = 0; i < capacity; ++i)
            slots[i].sequence.store (i, std::memory_order_relaxed);
    }

    bool try_push (const T& v)
    {
        for (;;)
        {
            size_t current_tail = tail.load (std::memory_order_relaxed);
            size_t index = current_tail & (capacity - 1);
            auto& slot = slots[index];

            size_t seq = slot.sequence.load (std::memory_order_acquire);

            if (ssize_t diff = seq - current_tail;
                diff == 0) // Slot is ready for writing
            {
                // Try to claim this slot
                if (tail.compare_exchange_weak (current_tail, current_tail + 1,
                                                std::memory_order_relaxed,
                                                std::memory_order_relaxed))
                {
                    // Successfully claimed, write data
                    slot.data = v;

                    // Mark slot as ready for reading
                    slot.sequence.store (current_tail + 1, std::memory_order_release);

                    return true;
                }

                // CAS failed, another producer claimed it, retry
            }
            else if (diff < 0) // Queue is full (slot hasn't been consumed yet)
            {
                return false;
            }

            // If diff > 0, slot isn't ready yet (another producer is writing), retry
        }
    }

private:
    struct slot
    {
        std::atomic<size_t> sequence;
        T data;
    };

    size_t capacity = 0;
    alignas(hardware_destructive_interference_size) std::atomic<size_t> head { 0 };
    alignas(hardware_destructive_interference_size) std::atomic<size_t> tail { 0 };
    std::vector<slot> slots;
};
```

```cpp
    bool try_pop (T& v)
    {
        for (;;)
        {
            size_t current_head = head.load (std::memory_order_relaxed);
            size_t index = current_head & (capacity - 1);
            auto& slot = slots[index];

            size_t seq = slot.sequence.load (std::memory_order_acquire);

            if (ssize_t diff = seq - (current_head + 1);
                diff == 0) // Slot is ready for reading
            {
                // Try to claim this slot
                if (head.compare_exchange_weak (current_head, current_head + 1,
                                                std::memory_order_relaxed,
                                                std::memory_order_relaxed))
                {
                    // Successfully claimed, read data
                    v = slot.data;

                    // Mark slot as consumed and ready for writing
                    slot.sequence.store (current_head + capacity, std::memory_order_release);

                    return true;
                }

                // CAS failed, another consumer claimed it, retry
            }
            else if (diff < 0) // Queue is empty
            {
                return false;
            }

            // If diff > 0, slot isn't ready yet (another consumer is reading), retry
        }
    }
```