# 零声教育出品 Mark 老师 QQ：2548898954

# lock-free 、 wait-free、以及 blocking

在多线程编程中，"lock-free"（无锁）和"wait-free"（无等待）是两个相关但不完全相同的概念。

- lock-free：系统作为一个整体无论如何都向前移动，不能保证每个线程的前进进度（可能出现线程饿死）。
  通常使用 compare_exchange 原语实现。可以有循环，但类似 compare_exchange 实现的
  自旋锁不行。
- wait-free：考虑到其他线程争用，阻塞等情况下，每个线程向前移动，每个操作在有限步骤中执行
  通常使用 exchange、fetch_add 等原语实现，并且不包含可能被其他线程影响的循环。
- blocking：整个系统可能不会取得任何进展，阻塞、中断或终止的线程可能无限地阻止系统范围内的向前。

> 使用无锁算法，保证当前向前推进的线程总是当前运行的线程之一。基于互斥锁，通常也有一个线程可以向前推进，但它可能是一个当前未运行的线程。
>
> 在一些不能使用基于锁的算法的情况下，可以使用无等待、无锁算法。例如：信号处理程序使用锁不安全，锁当前可以由被抢占的线程获取，它会立刻导致死锁。硬实时系统，执行时间有严格的上限，使用无等待算法更可取。

> 总之：如果你的系统可以使用blocking(不需要任何特定的向前进度保证)，那么可以从blocking/ Lockfree /wait-free中选择最快的算法。
>
> 如果您的系统需要无锁向前进度保证，那么只能从lockfree/waitfree 中进行选择。
>
> 如果您的系统需要无等待保证，那么您只能选择无等待算法。因此，阻塞至少和无锁一样快，而且可能更快(当它发生时，已知最快的算法至少是部分阻塞的)。

# 生产者消费者队列

> 生产者-消费者队列是并发系统中最基本的组件之一，没有“放之四海而皆准”的并发队列。
>
> 根据允许的生产者和消费者线程的数量，可以划分为：
>
> `MPMC`：多生产者/多消费者队列。
>
> `SPMC`：单生产者/多消费者队列。
>
> `MPSC`：多生产者/单消费者队列。
>
> `SPSC`：单生产者/单消费者队列。
>
> 如果只有1个生产者和1个消费者线程，可以使用SPSC队列而不是更一般的MPMC队列，它将明显更快。
>
> 根据底层数据结构，可以划分为：
>
> `Array-based`：基于数组；
>
> `Linked-list-based`：基于链表；
>
> `Hybrid`：混合的；

> 基于数组的队列通常更快，但是它们通常不是严格无锁的。缺点是它们需要为最坏的情况预先分配内存。链表队列是动态增长的，因此不需要预先分配任何内存。混合队列(固定大小的小数组的链表)试图结合两者的优点。
>
> 根据链表队列是否是侵入式的，可以划分：
>
> `Intrusive`：侵入式；
>
> `Non-intrusive`：非侵入式；
>
> 如果需要转换已经动态分配的数据，则侵入式队列通常具有更好的性能，因为不需要额外的节点内存管理。
>
> 根据链表队列长度的最大大小，可以划分：
>
> `Bounded`：有界
>
> `Unbounded`：无界
>
> 无界队列很危险，通常需要一个有界队列，因为它将强制执行你认为应该发生的事情，从而避免无法控制的事情。
>
> 还有其他，如有界队列中溢出部分如何处理，是否需要 GC，是否支持任务优先级等划分方式。

# locked_queue

> 应用场景：队列为空时不阻塞消费者线程，适合任务耗时的情况。

```
#ifndef LOCKEDQUEUE_H
#define LOCKEDQUEUE_H

#include <deque>
#include <mutex>

```

```
7  template <class T, typename StorageType =
   std::deque<T> >
8  class LockedQueue
9  {
10     //! Lock access to the queue.
11     std::mutex _lock;
12
13     //! Storage backing the queue.
14     StorageType _queue;
15
16     //! Cancellation flag.
17     volatile bool _canceled;
18
19 public:
20
21     //! Create a LockedQueue.
22     LockedQueue()
23         : _canceled(false)
24     {
25     }
26
27     //! Destroy a LockedQueue.
28     virtual ~LockedQueue()
29     {
30     }
31
32     //! Adds an item to the queue.
33     void add(const T& item)
34     {
35         lock();
36
37         _queue.push_back(item);
38
39         unlock();
40     }
41
42     //! Adds items back to front of the queue
```

```
43      template<class Iterator>
44      void readd(Iterator begin, Iterator end)
45      {
46          std::lock_guard<std::mutex> lock(_lock);
47          _queue.insert(_queue.begin(), begin,
end);
48      }
49
50      //! Gets the next result in the queue, if
any.
51      bool next(T& result)
52      {
53          std::lock_guard<std::mutex> lock(_lock);
54
55          if (_queue.empty())
56              return false;
57
58          result = _queue.front();
59          _queue.pop_front();
60
61          return true;
62      }
63
64      template<class Checker>
65      bool next(T& result, Checker& check)
66      {
67          std::lock_guard<std::mutex> lock(_lock);
68
69          if (_queue.empty())
70              return false;
71
72          result = _queue.front();
73          if (!check.Process(result))
74              return false;
75
76          _queue.pop_front();
77          return true;
```

```
78      }
79
80      //! Peeks at the top of the queue. Check if
    the queue is empty before calling! Remember to
    unlock after use if autoUnlock == false.
81      T& peek(bool autoUnlock = false)
82      {
83          lock();
84
85          T& result = _queue.front();
86
87          if (autoUnlock)
88              unlock();
89
90          return result;
91      }
92
93      //! Cancels the queue.
94      void cancel()
95      {
96          std::lock_guard<std::mutex> lock(_lock);
97
98          _canceled = true;
99      }
100
101     //! Checks if the queue is cancelled.
102     bool cancelled()
103     {
104         std::lock_guard<std::mutex> lock(_lock);
105         return _canceled;
106     }
107
108     //! Locks the queue for access.
109     void lock()
110     {
111         this->_lock.lock();
112     }
```

```
113
114     //! Unlocks the queue.
115     void unlock()
116     {
117         this->_lock.unlock();
118     }
119
120     ///! Calls pop_front of the queue
121     void pop_front()
122     {
123         std::lock_guard<std::mutex> lock(_lock);
124         _queue.pop_front();
125     }
126
127     ///! Checks if we're empty or not with locks
    held
128     bool empty()
129     {
130         std::lock_guard<std::mutex> lock(_lock);
131         return _queue.empty();
132     }
133 };
134 #endif
135
```

# ProducerConsumerQueue

```
1 #ifndef _PCQ_H
2 #define _PCQ_H
3
4 #include <condition_variable>
5 #include <mutex>
6 #include <queue>
7 #include <atomic>
8 #include <type_traits>
9
```

```cpp
template <typename T>
class ProducerConsumerQueue
{
private:
    std::mutex _queueLock;
    std::queue<T> _queue;
    std::condition_variable _condition;
    std::atomic<bool> _shutdown;

public:

    ProducerConsumerQueue<T>() : _shutdown(false) { }

    void Push(const T& value)
    {
        std::lock_guard<std::mutex> lock(_queueLock);
        _queue.push(std::move(value));

        _condition.notify_one();
    }

    bool Empty()
    {
        std::lock_guard<std::mutex> lock(_queueLock);

        return _queue.empty();
    }

    size_t Size() const
    {
        return _queue.size();
    }

    bool Pop(T& value)
```

```
44      {
45          std::lock_guard<std::mutex>
    lock(_queueLock);
46
47          if (_queue.empty() || _shutdown)
48              return false;
49
50          value = _queue.front();
51
52          _queue.pop();
53
54          return true;
55      }
56
57      void WaitAndPop(T& value)
58      {
59          std::unique_lock<std::mutex>
    lock(_queueLock);
60
61          // we could be using .wait(lock,
    predicate) overload here but it is broken
62          //
    https://connect.microsoft.com/VisualStudio/feedb
    ack/details/1098841
63          while (_queue.empty() && !_shutdown)
64              _condition.wait(lock);
65
66          if (_queue.empty() || _shutdown)
67              return;
68
69          value = _queue.front();
70
71          _queue.pop();
72      }
73
74      void Cancel()
75      {
```

```
        std::unique_lock<std::mutex> lock(_queueLock);

        while (!_queue.empty())
        {
            T& value = _queue.front();

            DeleteQueuedObject(value);

            _queue.pop();
        }

        _shutdown = true;

        _condition.notify_all();
    }

private:
    template<typename E = T>
    typename std::enable_if<std::is_pointer<E>::value>::type DeleteQueuedObject(E& obj) { delete obj; }

    template<typename E = T>
    typename std::enable_if<!std::is_pointer<E>::value>::type DeleteQueuedObject(E const& /*packet*/) { }
};

#endif

```

# MPSCQueue

```
#ifndef MPSCQueue_h__
#define MPSCQueue_h__

```

```
3
4  #include <atomic>
5  #include <utility>
6
7  template<typename T>
8  class MPSCQueueNonIntrusive
9  {
10 public:
11     MPSCQueueNonIntrusive() : _head(new Node()),
   _tail(_head.load(std::memory_order_relaxed))
12     {
13         Node* front =
   _head.load(std::memory_order_relaxed);
14         front->Next.store(nullptr,
   std::memory_order_relaxed);
15     }
16
17     ~MPSCQueueNonIntrusive()
18     {
19         T* output;
20         while (Dequeue(output))
21             delete output;
22
23         Node* front =
   _head.load(std::memory_order_relaxed);
24         delete front;
25     }
26
27     void Enqueue(T* input)
28     {
29         Node* node = new Node(input);
30         Node* prevHead = _head.exchange(node,
   std::memory_order_acq_rel);
31         prevHead->Next.store(node,
   std::memory_order_release);
32     }
33
```

```
34      bool Dequeue(T*& result)
35      {
36          Node* tail =
_tail.load(std::memory_order_relaxed);
37          Node* next = tail-
>Next.load(std::memory_order_acquire);
38          if (!next)
39              return false;
40
41          result = next->Data;
42          _tail.store(next,
std::memory_order_release);
43          delete tail;
44          return true;
45      }
46
47  private:
48      struct Node
49      {
50          Node() = default;
51          explicit Node(T* data) : Data(data)
52          {
53              Next.store(nullptr,
std::memory_order_relaxed);
54          }
55
56          T* Data;
57          std::atomic<Node*> Next;
58      };
59
60      std::atomic<Node*> _head;
61      std::atomic<Node*> _tail;
62
63      MPSCQueueNonIntrusive(MPSCQueueNonIntrusive
const&) = delete;
64      MPSCQueueNonIntrusive& operator=
(MPSCQueueNonIntrusive const&) = delete;
```

```
65 };
66
67 template<typename T, std::atomic<T*> T::*
   IntrusiveLink>
68 class MPSCQueueIntrusive
69 {
70 public:
71     MPSCQueueIntrusive() :
   _dummyPtr(reinterpret_cast<T*>
   (std::addressof(_dummy))), _head(_dummyPtr),
   _tail(_dummyPtr)
72     {
73         // _dummy is constructed from
   aligned_storage and is intentionally left
   uninitialized (it might not be default
   constructible)
74         // so we init only its IntrusiveLink
   here
75         std::atomic<T*>* dummyNext = new (&
   (_dummyPtr->*IntrusiveLink)) std::atomic<T*>();
76         dummyNext->store(nullptr,
   std::memory_order_relaxed);
77     }
78
79     ~MPSCQueueIntrusive()
80     {
81         T* output;
82         while (Dequeue(output))
83             delete output;
84     }
85
86     void Enqueue(T* input)
87     {
88         (input->*IntrusiveLink).store(nullptr,
   std::memory_order_release);
89         T* prevHead = _head.exchange(input,
   std::memory_order_acq_rel);
```

```
90          (prevHead->*IntrusiveLink).store(input,
    std::memory_order_release);
91      }
92
93      bool Dequeue(T*& result)
94      {
95          T* tail =
    _tail.load(std::memory_order_relaxed);
96          T* next = (tail-
    >*IntrusiveLink).load(std::memory_order_acquire)
    ;
97          if (tail == _dummyPtr)
98          {
99              if (!next)
100                 return false;
101
102             _tail.store(next,
    std::memory_order_release);
103             tail = next;
104             next = (next-
    >*IntrusiveLink).load(std::memory_order_acquire)
    ;
105         }
106
107         if (next)
108         {
109             _tail.store(next,
    std::memory_order_release);
110             result = tail;
111             return true;
112         }
113
114         T* head =
    _head.load(std::memory_order_acquire);
115         if (tail != head)
116             return false;
117
```

```
118         Enqueue(_dummyPtr);
119         next = (tail->*IntrusiveLink).load(std::memory_order_acquire);
120         if (next)
121         {
122             _tail.store(next, std::memory_order_release);
123             result = tail;
124             return true;
125         }
126         return false;
127     }
128 
129 private:
130     // std::aligned_storage_t模板来创建一个与类型T大小和对齐方式相匹配的临时缓冲区。
131     std::aligned_storage_t<sizeof(T), alignof(T)> _dummy;
132     T* _dummyPtr;
133     std::atomic<T*> _head;
134     std::atomic<T*> _tail;
135 
136     MPSCQueueIntrusive(MPSCQueueIntrusive const&) = delete;
137     MPSCQueueIntrusive& operator=(MPSCQueueIntrusive const&) = delete;
138 };
139 
140 template<typename T, std::atomic<T*> T::* IntrusiveLink = nullptr>
141 using MPSCQueue = std::conditional_t<IntrusiveLink != nullptr, MPSCQueueIntrusive<T, IntrusiveLink>, MPSCQueueNonIntrusive<T>>;
142 
143 #endif // MPSCQueue_h__
```

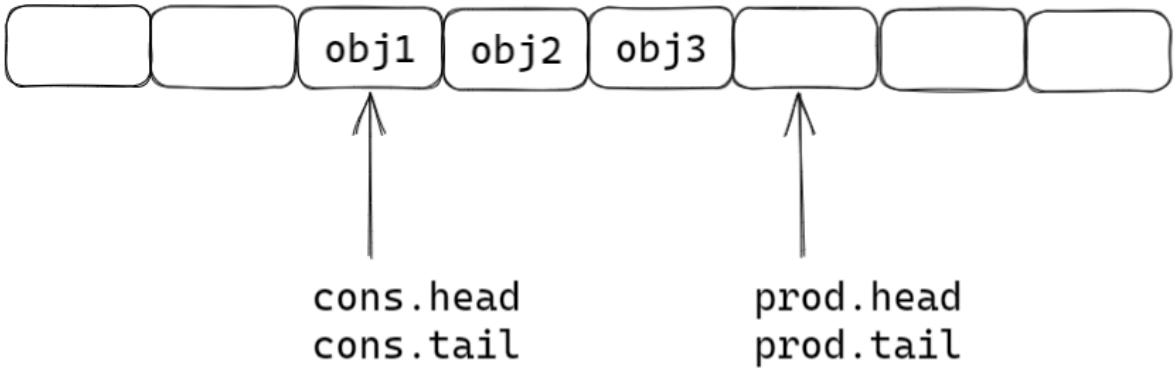
144

# rte_ring

> 这里提供结构说明，具体看 rte_ring.h/ rte_ring.c 的实现。
>
> 特性：
>
> - 无锁
> - 支持不同数量的生产者和消费者的出队和入队操作。
> - 支持批量入队和出队。

```
1
2  struct rte_ring {
3    char name[RTE_RING_NAMESIZE];    /**< Name of
   the ring. */
4    int flags;                       /**< Flags
   supplied at creation. */
5
6    /** Ring producer status. */
7    struct prod {
8      uint32_t watermark;      /**< Maximum items
   before EDQUOT. */
9      uint32_t sp_enqueue;     /**< True, if single
   producer. */
10     uint32_t size;           /**< Size of ring.
   */
11     uint32_t mask;           /**< Mask (size-1)
   of ring. */
12     volatile uint32_t head;  /**< Producer head.
   */
13     volatile uint32_t tail;  /**< Producer tail.
   */
14   } prod __rte_cache_aligned;
15
16   /** Ring consumer status. */
17   struct cons {
```

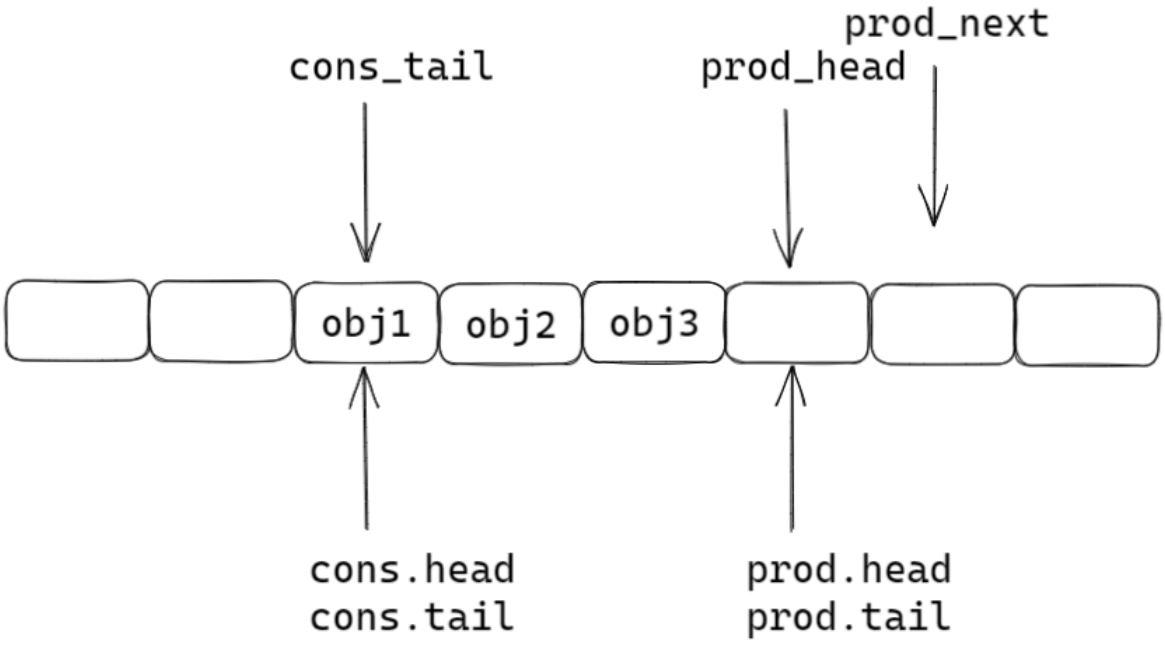
```
18      uint32_t sc_dequeue;     /**< True, if single
consumer. */
19      uint32_t size;           /**< Size of the
ring. */
20      uint32_t mask;           /**< Mask (size-1)
of ring. */
21      volatile uint32_t head;  /**< Consumer head.
*/
22      volatile uint32_t tail;  /**< Consumer tail.
*/
23  #ifdef RTE_RING_SPLIT_PROD_CONS
24    } cons __rte_cache_aligned;
25  #else
26    } cons;
27  #endif
28
29  #ifdef RTE_LIBRTE_RING_DEBUG
30    struct rte_ring_debug_stats
stats[RTE_MAX_LCORE];
31  #endif
32
33    void * ring[0] __rte_cache_aligned; /**< Memory
space of ring starts here.
34                        * not volatile so need to be
careful
35                        * about compiler re-ordering
*/
36  };
37
38
```



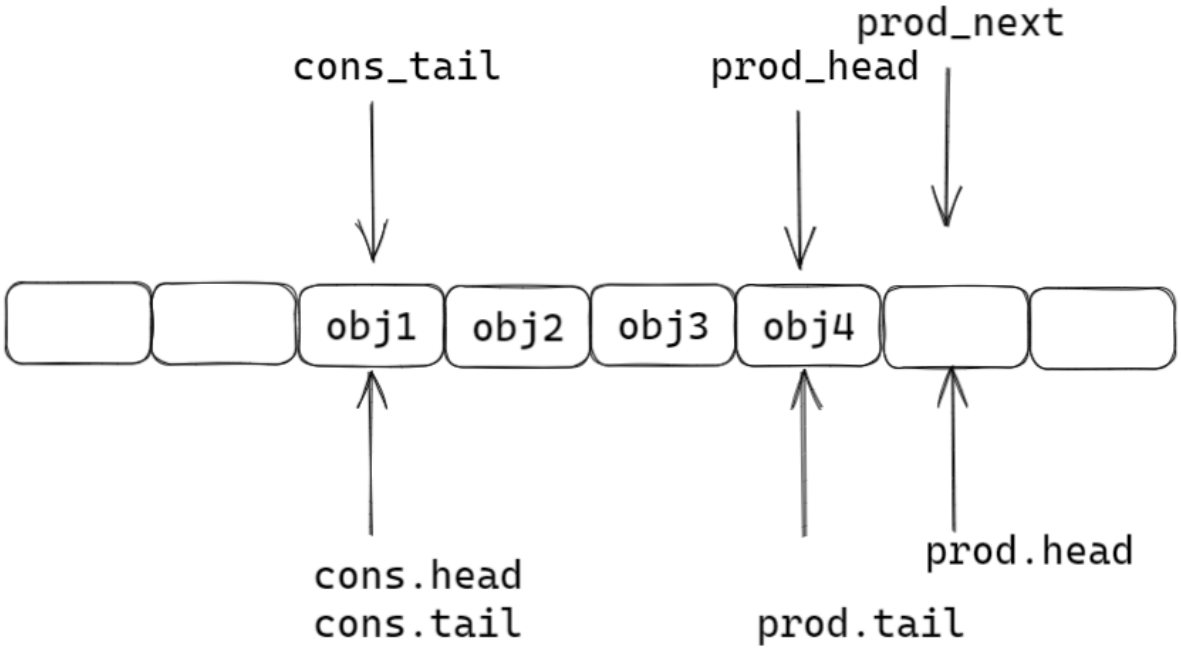

# 入队操作

## 单生产者入队

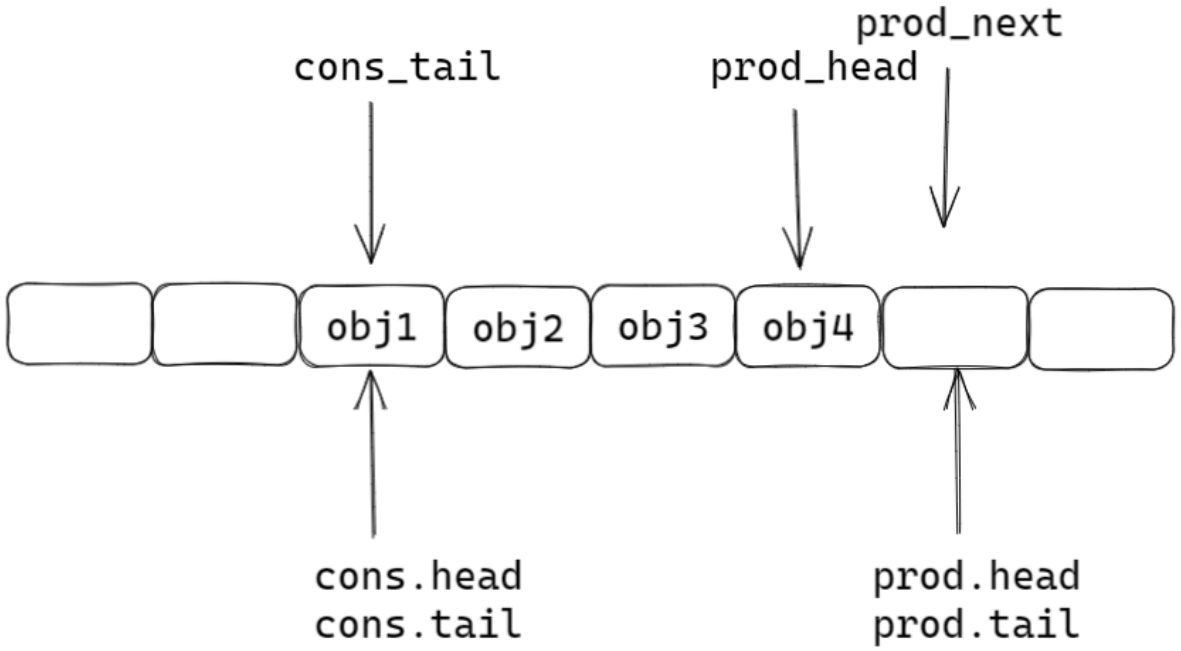
- step 1




- step 2

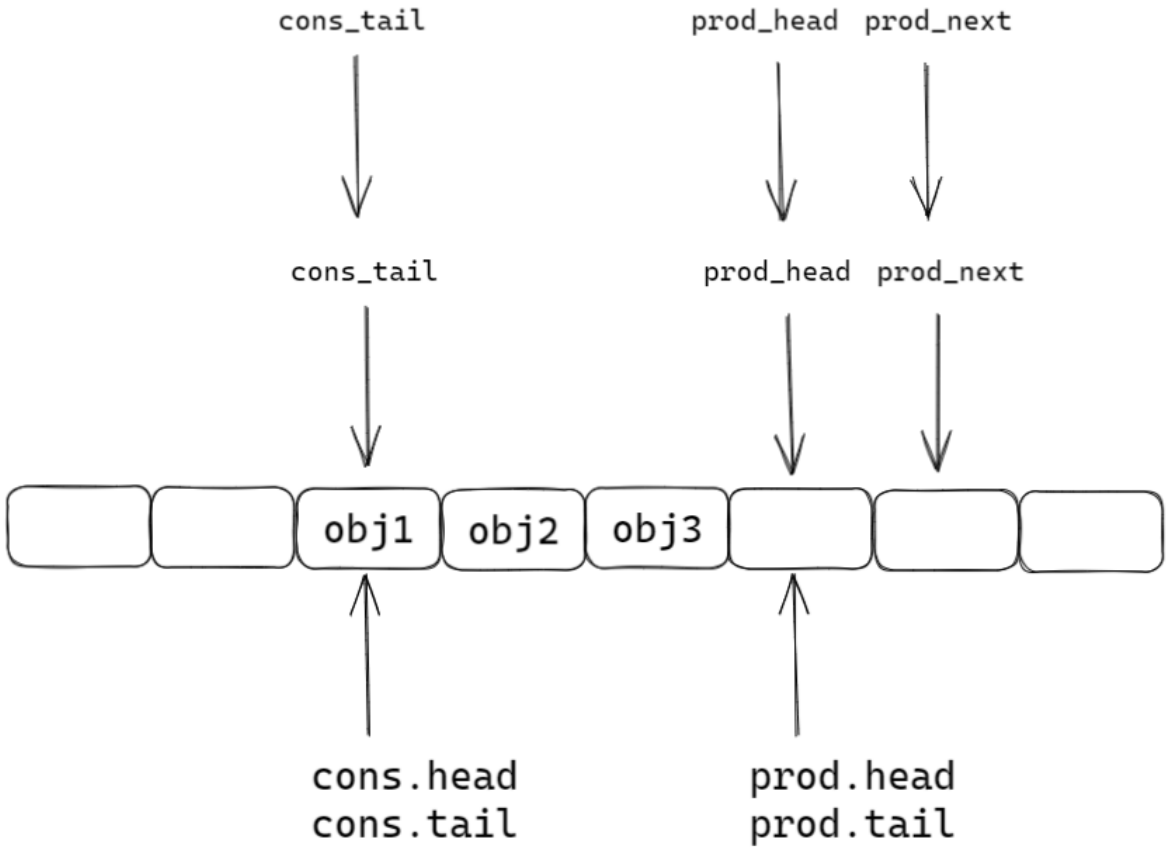



- step 3

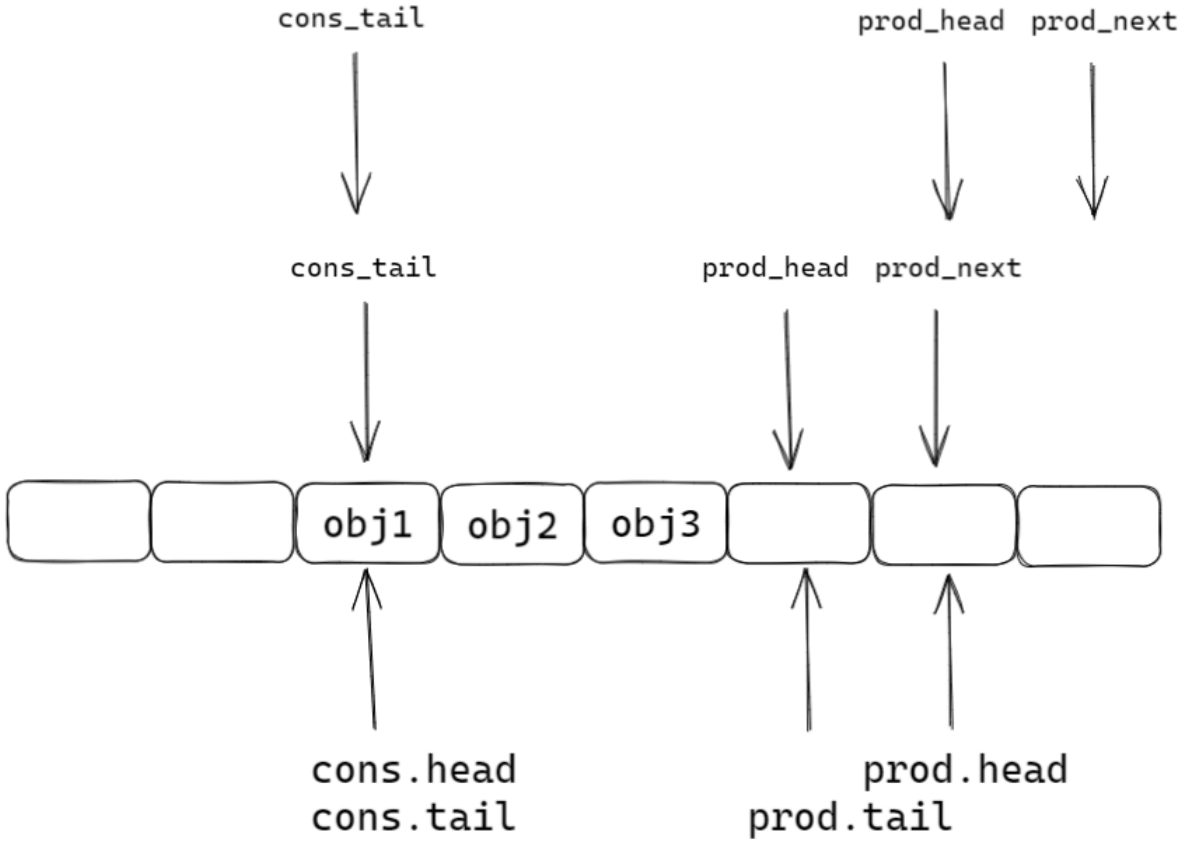



## 多生产者入队

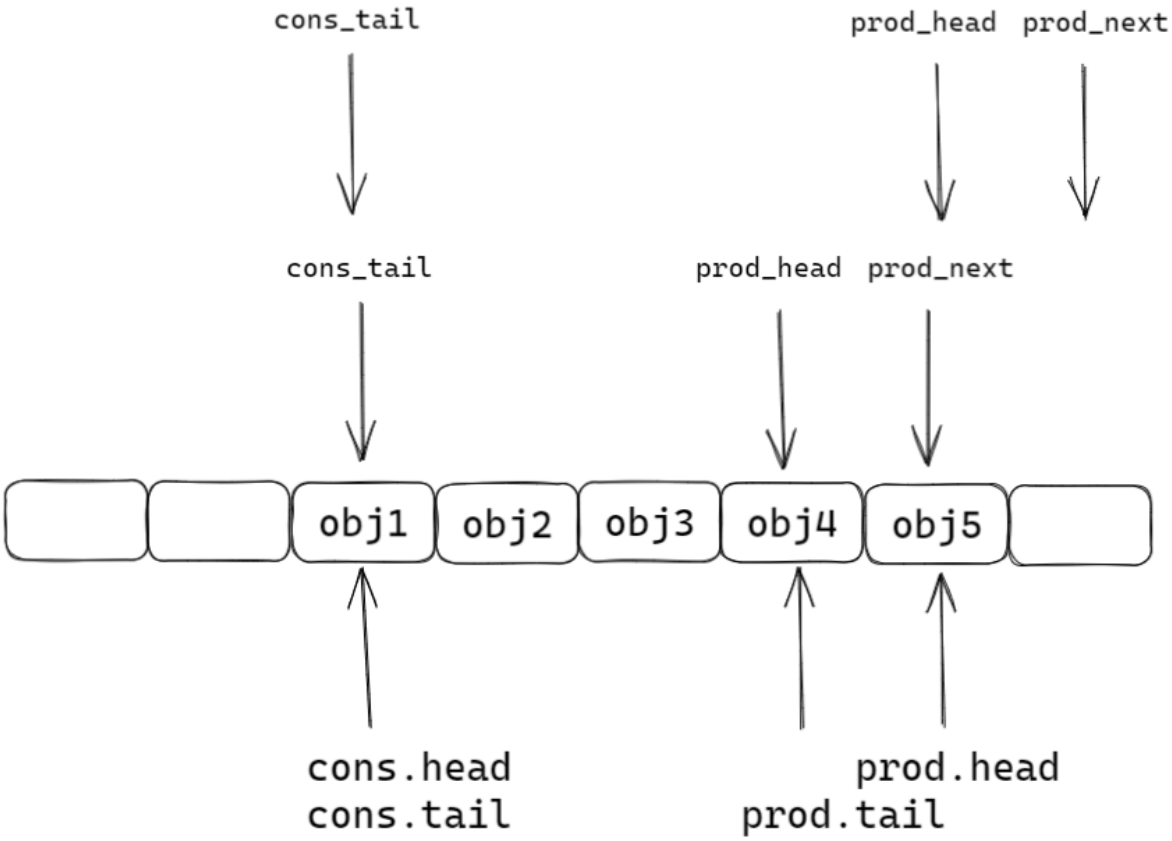
- step 1


cons_tail
prod_head
prod_next
cons_tail
prod_head
prod_next
obj1
obj2
obj3
cons.head
cons.tail
prod.head
prod.tail

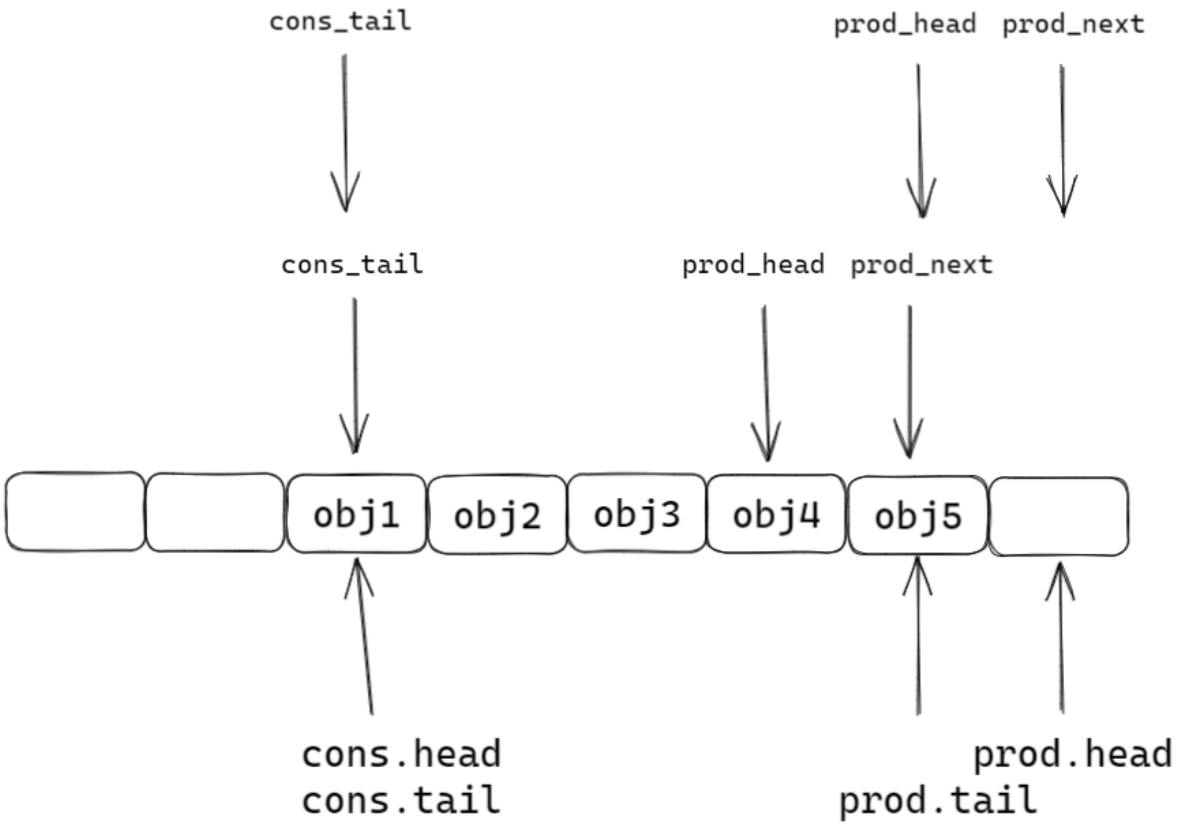
- step 2


cons_tail
prod_head
prod_next
cons_tail
prod_head
prod_next
obj1
obj2
obj3
cons.head
cons.tail
prod.head
prod.tail

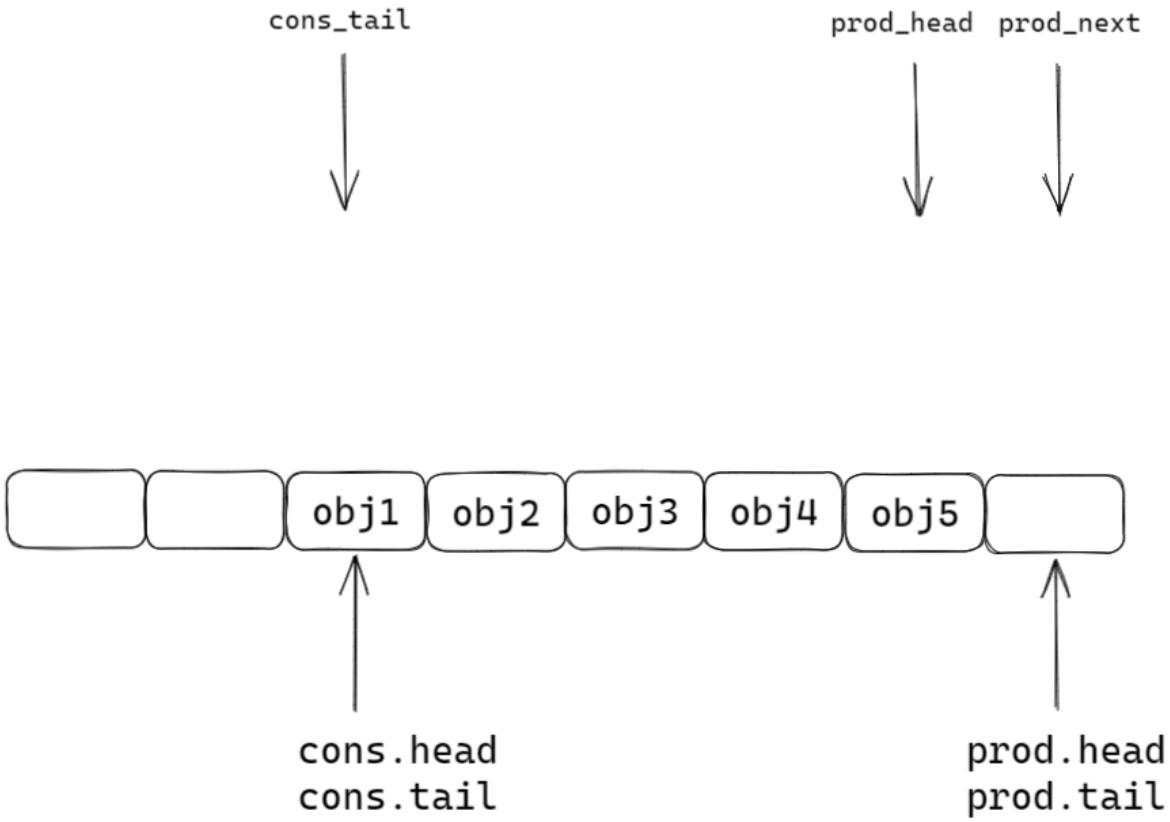
- step 3


cons_tail
prod_head
prod_next
cons_tail
prod_head
prod_next
obj1
obj2
obj3
obj4
obj5
cons.head
cons.tail
prod.head
prod.tail

- step 4


cons_tail
prod_head
prod_next
cons_tail
prod_head
prod_next
obj1
obj2
obj3
obj4
obj5
cons.head
cons.tail
prod.head
prod.tail

- step 5


cons_tail
prod_head
prod_next
obj1
obj2
obj3
obj4
obj5
cons.head
cons.tail
prod.head
prod.tail