*零声学院 [C/C++Linux服务器开发/后台架构师 **](https://ke.qq.com/course/417774?taid=3659514000072686&tuin=137bb271)

qq 326873713 Darren
重点内容提示

- CAS的含义
- yqueue_t/ypipe_t无锁队列的设计原理
- 每次加入元素都动态分配节点对性能的影响（可以将ypipe_t<int, 10000> 、ypipe_t<int, 10>、ypipe_t<int, 1> 测试性能对比分析 ）
- ArrayLockFreeQueue无锁队列如何实现多线程安全
- 多写多读，多写单读，单写多读、单写单读不同队列的性能测试和分析

1 引言

锁是解决并发问题的万能钥匙，可是并发问题只有锁能解决吗?

2 什么是CAS?

比较并交换(compare and swap, CAS)，是原子操作的一种，可用于在多线程编程中实现不被打断的数据交换操作，从而避免多线程同时改写某一数据时由于执行顺序不确定性以及中断的不可预知性产生的数据不一致问题。 该操作通过将内存中的值与指定数据进行比较，当数值一样时将内存中的数据替换为新的值。

```
bool CAS( int * pAddr, int nExpected, int nNew )
atomically {
    if ( *pAddr == nExpected ) {
         *pAddr = nNew ;
         return true ;
    }
    else
        return false ;
}
```

上面的CAS返回bool告知原子性交换是否成功。

2 操作的原子性

test_i++.c

```
#include <stdio.h>

int i = 0;
int main(int argc, char **argv)
{
    i++;
    return 0;
}
```

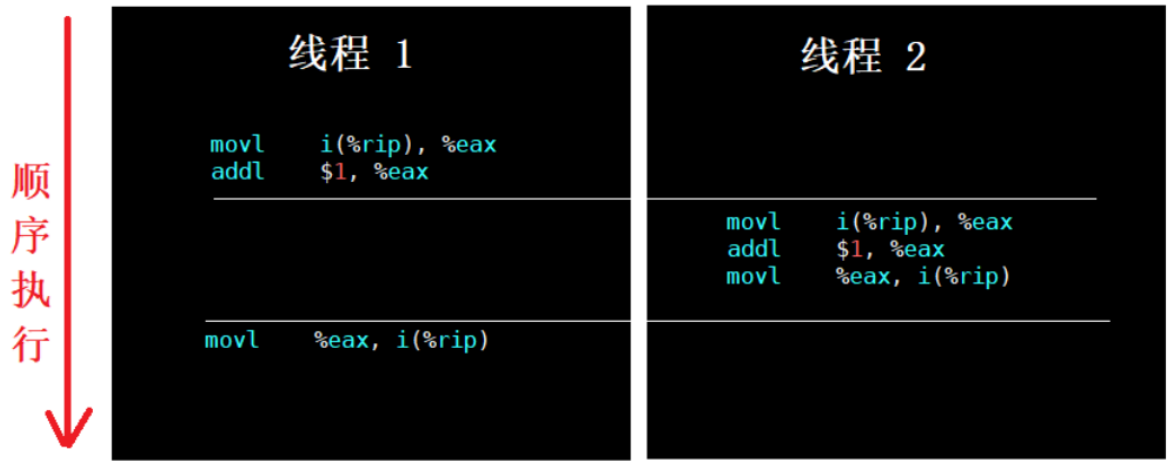
gcc -S test_i++.c 产生汇编指令
实际上当我们操作i++的时候，其汇编指令为：

```
23    movl  i(%rip), %eax
24    addl  $1, %eax
25    movl  %eax, i(%rip)
```

完整的汇编程序如下所示。

```
    .file   "test_i++.c"
    .text
    .globl  i
    .bss
    .align 4
    .type   i, @object
    .size   i, 4
i:
    .zero   4
    .text
    .globl  main
    .type   main, @function
main:
.LFB0:
    .cfi_startproc
    pushq   %rbp
    .cfi_def_cfa_offset 16
    .cfi_offset 6, -16
    movq    %rsp, %rbp
    .cfi_def_cfa_register 6
    movl    %edi, -4(%rbp)
    movq    %rsi, -16(%rbp)
    movl    i(%rip), %eax
    addl    $1, %eax
    movl    %eax, i(%rip)
    movl    $0, %eax
    popq    %rbp
    .cfi_def_cfa 7, 8
    ret
    .cfi_endproc
.LFE0:
    .size   main, .-main
    .ident  "GCC: (Ubuntu 8.3.0-6ubuntu1~18.10.1) 8.3.0"
    .section    .note.GNU-stack,"",@progbits
```

从上图能看到一个i++对应的操作是：
（1）把变量i从内存（RAM）加载到寄存器；
（2）把寄存器的值加1；
（3）把寄存器的值写回内存（RAM）。
那如果有多个线程去做i++操作的时候，也就可能导致这样一种情况：



上图这种执行情况就导致i变量的结果仅仅自增了一次，而不是两次，导致实际结果与预期结果不对。
可以使用：test_i++_threads.c 测试

总结下上面的问题，也就是说我们整个存储的结构如下图：

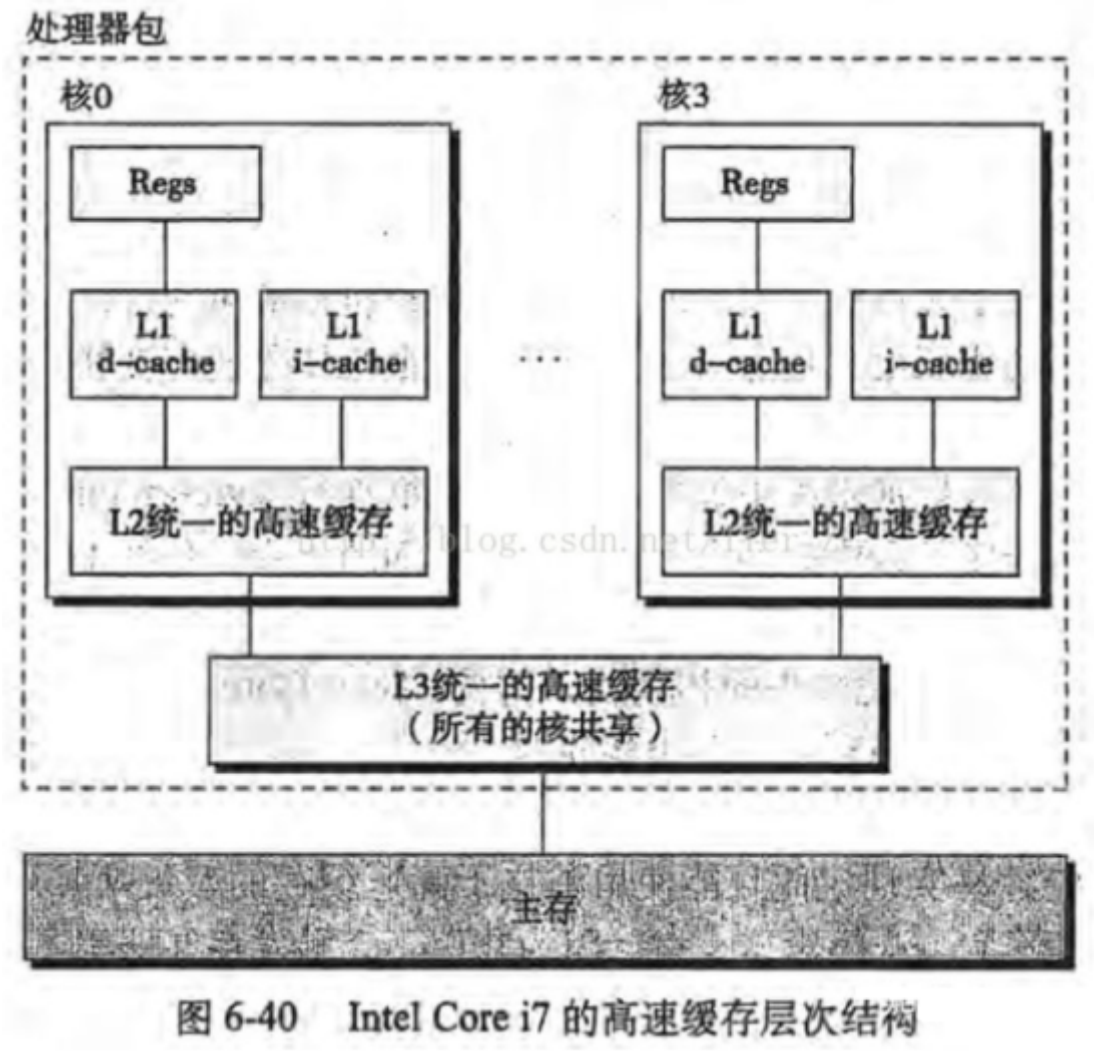


图 6-40 Intel Core i7 的高速缓存层次结构

L1 32K
L2 256K
L3 8M

我们所有的变量首先是存储在主存（RAM）上，CPU要去操作的时候首先会加载到寄存器，然后才操作，操作好了才写会主存。
关于CPU和cache的更详细介绍可以参考：https://www.cnblogs.com/jokerjason/p/10711022.html

3 原子操作

对于gcc、g++编译器来讲，它们提供了一组API来做原子操作：

```
type __sync_fetch_and_add (type *ptr, type value, ...)
type __sync_fetch_and_sub (type *ptr, type value, ...)
type __sync_fetch_and_or (type *ptr, type value, ...)
type __sync_fetch_and_and (type *ptr, type value, ...)
type __sync_fetch_and_xor (type *ptr, type value, ...)
type __sync_fetch_and_nand (type *ptr, type value, ...)

bool __sync_bool_compare_and_swap (type *ptr, type oldval type newval, ...)
type __sync_val_compare_and_swap (type *ptr, type oldval type newval, ...)

type __sync_lock_test_and_set (type *ptr, type value, ...)
void __sync_lock_release (type *ptr, ...)
```

详细文档见：https://gcc.gnu.org/onlinedocs/gcc-4.1.1/gcc/Atomic-Builtins.html#Atomic-Builtins

对于c++11来讲，也有一组atomic的接口：详细文档见： https://en.cppreference.com/w/cpp/atomic

但是这些原子操作都是怎么实现的呢?
X86的架构
Intel X86指令集提供了指令前缀lock用于锁定前端串行总线FSB，保证了指令执行时不会收到其他处理器的干扰。比如：

```
static int lxx_atomic_add(int *ptr, int increment)
{
    int old_value = *ptr;
    __asm__ volatile("lock; xadd %0, %1 \n\t"
                     : "=r"(old_value), "=m"(*ptr)
                     : "0"(increment), "m"(*ptr)
                     : "cc", "memory");
    return *ptr;
}
```

使用lock指令前缀之后，处理期间对count内存的并发访问（Read/Write）被禁止，从而保证了指令的原子性。

4 自旋锁

3_test_mutex_spin_lock.cpp
posix提供一组自旋锁（spin lock）的API：

```
int pthread_spin_destroy(pthread_spinlock_t *);
int pthread_spin_init(pthread_spinlock_t *, int pshared);
int pthread_spin_lock(pthread_spinlock_t *);
int pthread_spin_trylock(pthread_spinlock_t *);
int pthread_spin_unlock(pthread_spinlock_t *);
```

pshared的取值：

- PTHREAD_PROCESS_SHARED：该自旋锁可以在多个进程中的线程之间共享。
- PTHREAD_PROCESS_PRIVATE：仅初始化本自旋锁的线程所在的进程内的线程才能够使用该自旋锁。

我们先来看一个示例：

```c
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>
#include <time.h>
#define MAX_THREAD_NUM 2
#define FOR_LOOP_COUNT 20000000
int counter = 0;
pthread_spinlock_t spinlock;
pthread_mutex_t mutex;
typedef void *(*thread_func_t)(void *argv);
static int lxx_atomic_add(int *ptr, int increment)
{
    int old_value = *ptr;
    __asm__ volatile("lock; xadd %0, %1 \n\t"
                     : "=r"(old_value), "=m"(*ptr)
                     : "0"(increment), "m"(*ptr)
                     : "cc", "memory");
    return *ptr;
}

void *mutex_thread_main(void *argv)
{
    for (int i = 0; i < FOR_LOOP_COUNT; i++)
    {
        pthread_mutex_lock(&mutex);
        counter++;
        pthread_mutex_unlock(&mutex);
    }
    return NULL;
}


void *atomic_thread_main(void *argv)
{
    for (int i = 0; i < FOR_LOOP_COUNT; i++)
    {
        lxx_atomic_add(&counter, 1);
        // counter++;
    }
    return NULL;
}
void *spin_thread_main(void *argv)
{
    for (int i = 0; i < FOR_LOOP_COUNT; i++)
    {
        pthread_spin_lock(&spinlock);
        counter++;
        pthread_spin_unlock(&spinlock);
    }
    return NULL;
}


int test_lock(thread_func_t func, char **argv)
{
    clock_t start = clock();
    pthread_t tid[MAX_THREAD_NUM] = {0};
```

```
    for (int i = 0; i < MAX_THREAD_NUM; i++)
    {
        int ret = pthread_create(&tid[i], NULL, func, argv);
        if (0 != ret)
        {
            printf("create thread failed\n");
        }
    }
    for (int i = 0; i < MAX_THREAD_NUM; i++)
    {
        pthread_join(tid[i], NULL);
    }
    clock_t end = clock();
    printf("spend clock : %ld, ", (end - start)/CLOCKS_PER_SEC);
    return 0;
}

// 多尝试几次
int main(int argc, char **argv)
{
    printf("THREAD_NUM:%d\n\n", MAX_THREAD_NUM);
    counter = 0;
    printf("use mutex ----------->\n");
    test_lock(mutex_thread_main, NULL);
    printf("counter = %d\n", counter);

    counter = 0;
    printf("\nuse atomic ----------->\n");
    test_lock(atomic_thread_main, NULL);
    printf("counter = %d\n", counter);

    counter = 0;
    printf("\nuse spin ----------->\n");
    pthread_spin_init(&spinlock, PTHREAD_PROCESS_PRIVATE);
    test_lock(spin_thread_main, NULL);
    printf("counter = %d\n", counter);

    return 0;
}
```

自旋锁则不是一种休眠等待的方式，而是一种忙等待的过程，就是自旋锁的pthread_spin_lock里有一个死循环，这个死循环一直检查锁的状态，如果是lock状态，则继续执行死循环，否则上锁，结束死循环。

5 为什么需要无锁队列

锁引起的问题：

- Cache损坏(Cache trashing)
- 在同步机制上的争抢队列
- 动态内存分配

5.1 Cache损坏(Cache trashing)

在保存和恢复上下文的过程中还隐藏了额外的开销：Cache中的数据会失效,因为它缓存的是将被换出任务的数据,这些数据对于新换进的任务是没用的。处理器的运行速度比主存快N倍,所以大量的处理器时间被浪费在处理器与主存的数据传输上。这就是在处理器和主存之间引入Cache的原因。Cache是一种速度更快但容量更小的内存(也更加昂贵),当处理器要访问主存中的数据时,这些数据首先被拷贝到Cache中，因为这些数据在不久的将来可能又会被处理器访问。Cache misses对性能有非常大的影响,因为处理器访问Cache中的数据将比直接访问主存快得多。
线程被频繁抢占产生的Cache损坏将导致应用程序性能下降。

5.2 在同步机制上的争抢队列

阻塞不是微不足道的操作。它导致操作系统暂停当前的任务或使其进入睡眠状态(等待，不占用任何的处理器)。直到资源(例如互斥锁)可用，被阻塞的任务才可以解除阻塞状态(唤醒)。在一个负载较重的应用程序中使用这样的阻塞队列来在线程之间传递消息会导致严重的争用问题。也就是说，**任务将大量的时间(睡眠，等待，唤醒)浪费在获得保护队列数据的互斥锁，而不是处理队列中的数据上。**

非阻塞机制大展伸手的机会到了。任务之间不争抢任何资源，在队列中预定一个位置，然后在这个位置上插入或提取数据。这中机制使用了一种被称之为CAS(比较和交换)的特殊操作，这个特殊操作是一种特殊的指令，它可以原子的完成以下操作:它需要3个操作数m，A，B，其中m是一个内存地址，操作将m指向的内存中的内容与A比较，如果相等则将B写入到m指向的内存中并返回true，如果不相等则什么也不做返回false。

```
volatile int a;
    a = 1;

    // this will loop while 'a' is not equal to 1.
    // If it is equal to 1 the operation will atomically set a to 2 and return
true
    while (!CAS(&a, 1, 2))
    {
        ;
    }
```

5.3 动态内存分配

在多线程系统中,需要仔细的考虑动态内存分配。当一个任务从堆中分配内存时，标准的内存分配机制会阻塞所有与这个任务共享地址空间的其它任务(进程中的所有线程)。这样做的原因是让处理更简单，且它工作得很好。两个线程不会被分配到一块相同的地址的内存，因为它们没办法同时执行分配请求。显然**线程频繁分配内存会导致应用程序性能下降**(必须注意,向标准队列或map插入数据的时候都会导致堆上的动态内存分配)

5.4 小结

6 无锁队列的实现

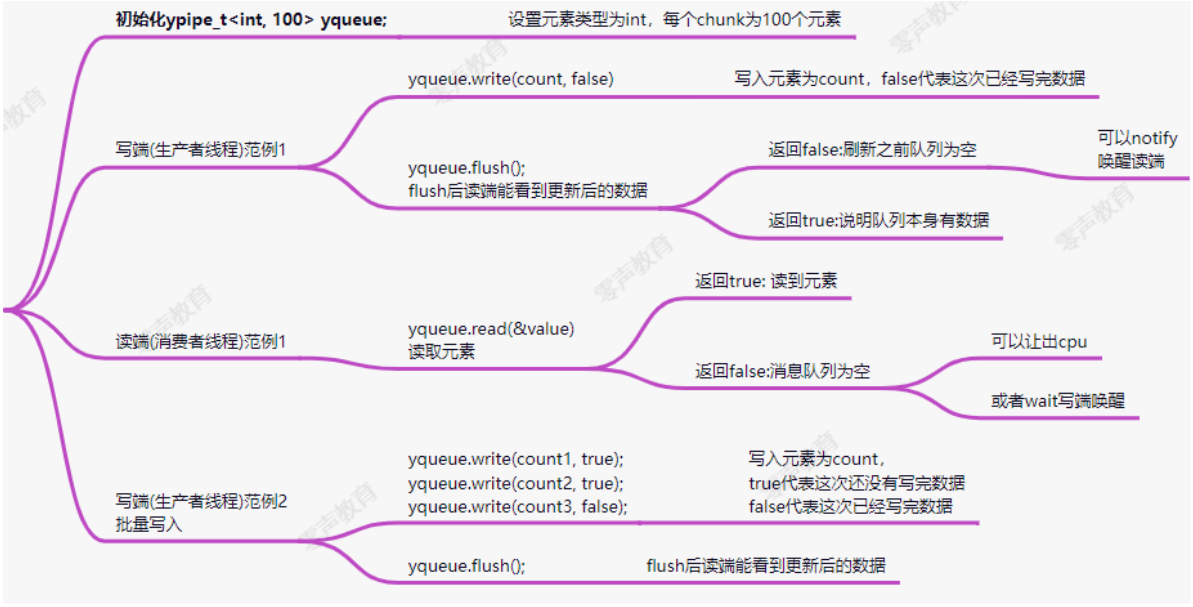
见源码的ypipe.hpp、yqueue.hpp（参考zmq），这些源码可以在工程项目使用，但要注意，**这里只支持单写单读取的场景。**

重点：

- yqueue_t和ypipe_t的数据结构
- **chunk设计的优点，N值大小对性能的影响、spare_chunk的作用(局部性原理)，这些可以在面试的时候和面试官讲述。**

难点：
ypipe_t r指针的预读机制不好理解，r可以理解为read_end，r并不是读数据的位置索引，而是我们可以最多读到哪个位置的索引。读数据的索引位置还是begin_pos。

ypipe_t无锁队列的使用



源码分析-原子操作函数


```
//  This class encapsulates several atomic operations on pointers.
    template <typename T> class atomic_ptr_t
    {
    public:
        inline void set (T *ptr_);      //非原子操作
        inline T *xchg (T *val_);       //原子操作，设置一个新的值，然后返回旧的值
        inline T *cas (T *cmp_, T *val_);//原子操作
    private:
        volatile T *ptr;
    }
```

- set函数，把私有成员ptr指针设置成参数ptr_的值，不是一个原子操作，需要使用者确保执行set过程没有其他线程使用ptr的值。
- xchg函数，把私有成员ptr指针设置成参数val_的值，并返回ptr设置之前的值。**原子操作**，线程安全。
- cas函数，原子操作，**线程安全**，把私有成员ptr指针与参数cmp_指针比较：
  - 如果相等，就把ptr设置为参数val_的值，返回ptr设置之前的值；
  - 如果不相等直接返回ptr值。

源码分析-yqueue_t

yqueue_t是比如ypipe_t更底层的类。

yqueue_t类比ypipe_t容易理解。

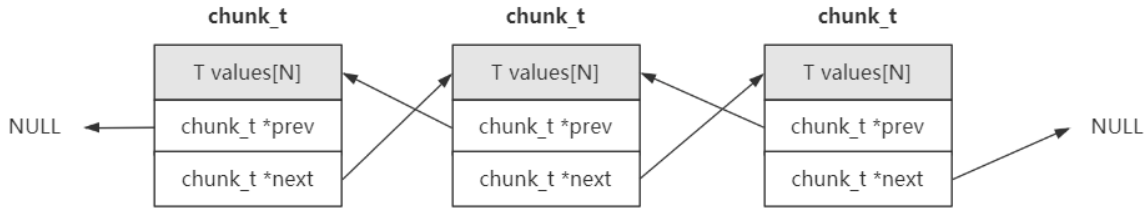
类接口和变量

```
//  T is the type of the object in the queue.队列中元素的类型
    //  N is granularity(粒度) of the queue，简单来说就是yqueue_t一个结点可以装载N个T类型的元素
    template <typename T, int N> class yqueue_t
    {
    public:
        inline yqueue_t ();//  Create the queue.
        inline ~yqueue_t ();//  Destroy the queue.
        inline T &front ();//  Returns reference to the front element of the queue. If the queue is empty, behaviour is undefined.
        inline T &back ();//  Returns reference to the back element of the queue.If the queue is empty, behaviour is undefined.
        inline void push ();//  Adds an element to the back end of the queue.
        inline void pop ();//  Removes an element from the front of the queue.
        inline void unpush ()//  Removes element from the back end of the queue。回滚时使用
    private:
        //  Individual memory chunk to hold N elements.
        struct chunk_t
        {
             T values [N];
             chunk_t *prev;
             chunk_t *next;
        };

        chunk_t *begin_chunk;
        int begin_pos;
        chunk_t *back_chunk;
        int back_pos;
        chunk_t *end_chunk;
        int end_pos;

        atomic_ptr_t<chunk_t> spare_chunk;  //空闲块（我把所有元素都已经出队的块称为空闲块），读写线程的共享变量
    };
```

数据结构逻辑

yqueue_t的实现，每次批量分配一批元素，减少内存的分配和释放（解决不断动态内存分配的问题）。yqueue_t内部由一个一个chunk组成，每个chunk保存N个元素。

```
    struct chunk_t
    {
        T values[N];  //每个chunk_t可以容纳N个T类型的元素，以后就以一个chunk_t为单位申请内存
        chunk_t *prev;
        chunk_t *next;
    };
```



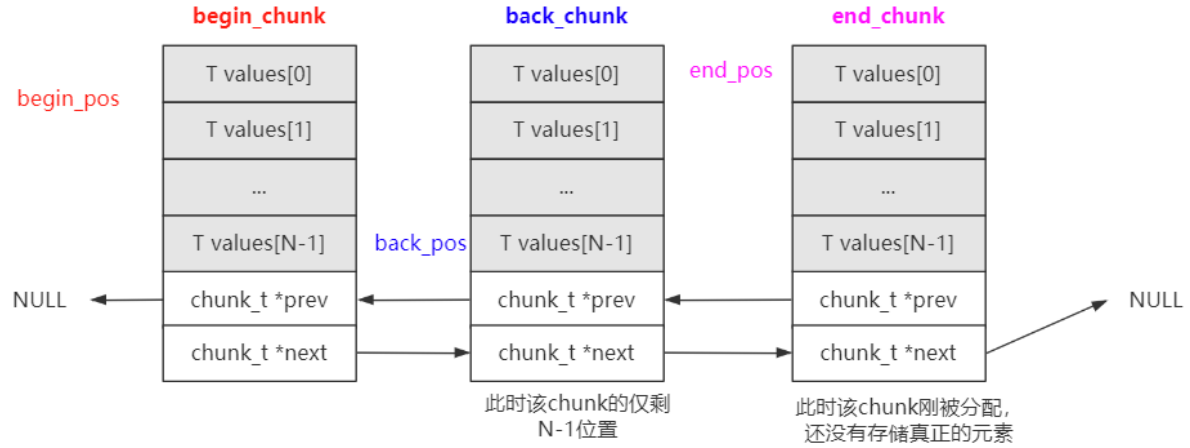
当队列空间不足时每次分配一个chunk_t，每个chunk_t能存储N个元素。
在数据出队列后，队列有多余空间的时候，**回收的chunk也不是马上释放，而是根据局部性原理先回收到spare_chunk里面**，当再次需要分配chunk_t的时候从spare_chunk中获取。

yqueue_t内部有三个chunk_t类型指针以及对应的索引位置：

- begin_chunk/begin_pos：begin_chunk用于指向队列头的chunk，begin_pos用于指向队列第一个元素在当前chunk中的位置。
- back_chunk/back_pos：**back_chunk**用于指向队列尾的chunk，back_pos用于指向队列最后一个元素在当前chunk的位置。
- end_chunk/end_pos：由于chunk是批量分配的，所以end_chunk用于指向分配的最后一个chunk位置。

这里特别需要注意区分back_chunk/back_pos和end_chunk/end_pos的作用：

- back_chunk/back_pos：对应的是元素存储位置；
- end_chunk/end_pos：决定是否要分配chunk或者回收chunk。



上图中：

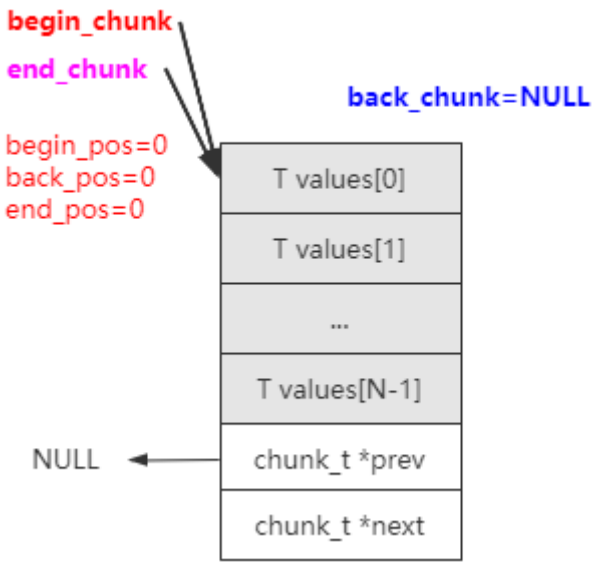
- 有三块chunk，分别由begin_chunk、back_chunk、end_chunk组成。
- **begin_pos指向begin_chunk中的第n个元素。**
- back_pos指向back_chunk的最后一个元素。

- 由于back_pos已经指向了back_chunk的最后一个元素，**所以end_pos就指向了end_chunk的第一个元素。**

另外还有一个spare_chunk指针，用于保存释放的chunk指针，当需要再次分配chunk的时候，会首先查看这里，从这里分配chunk。这里使用了原子的cas操作来完成，利用了操作系统的局部性原理。

yqueue_t构造函数

```
 // Create the queue.
inline yqueue_t()
{
    begin_chunk = (chunk_t *)malloc(sizeof(chunk_t)); // 预先分配chunk
    alloc_assert(begin_chunk);
    begin_pos = 0;
    back_chunk = NULL;  //back_chunk总是指向队列中最后一个元素所在的chunk，现在还没有元素，所以初始为空
    back_pos = 0;
    end_chunk = begin_chunk;    //end_chunk总是指向链表的最后一个chunk
    end_pos = 0;
}
```



end_chunk总是指向最后分配的chunk，刚分配出来的chunk，end_pos也总是为0。
back_chunk需要chunk有元素插入的时候才指向对应的chunk。

front、back函数

```
//  Returns reference to the front element of the queue.
//  If the queue is empty, behaviour is undefined.
inline T &front()   // 返回的是引用，是个左值，调用者可以通过其修改容器的值
{
    return begin_chunk->values[begin_pos]; // 队列首个chunk对应的的begin_pos
}

//  Returns reference to the back element of the queue.
//  If the queue is empty, behaviour is undefined.
inline T &back()     // 返回的是引用，是个左值，调用者可以通过其修改容器的值
{
    return back_chunk->values[back_pos];
}
```

这里的front()或者back()函数，需要注意的返回的是左值引用，我们可以修改其值。
对于先进后出队列而言：

- begin_chunk->values[begin_pos]代表队列头可读元素， 读取队列头元素即是读取begin_pos位置的元素；
- back_chunk->values[back_pos]代表队列尾可写元素，写入元素时则是更新back_pos位置的元素，要确保元素真正生效，还需要调用push函数更新back_pos的位置，避免下次更新的时候又是更新当前back_pos位置对应的元素。

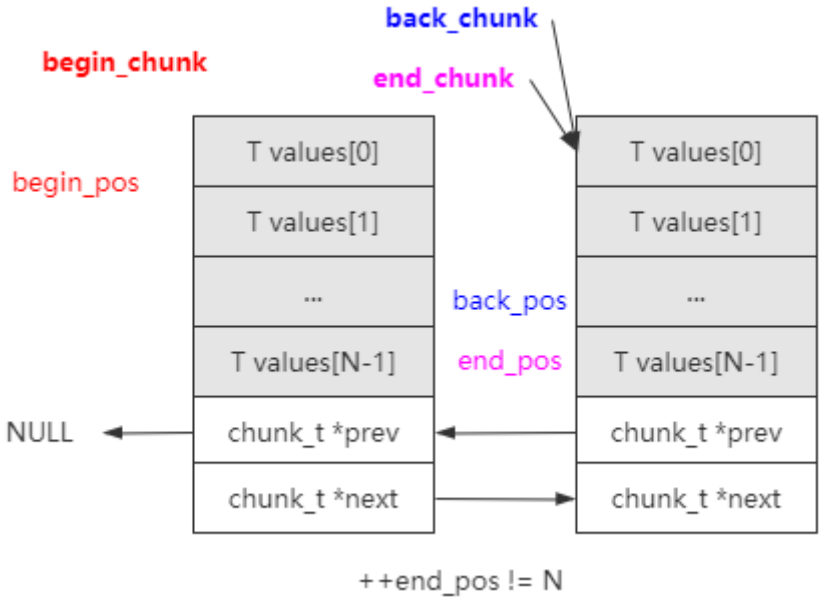
push函数

更新下一个元素写入位置，**如果end_pos超过chunk的索引位置(==N)则申请一个chunk**（先尝试从spare_chunk获取，如果为空再申请分配全新的chunk）

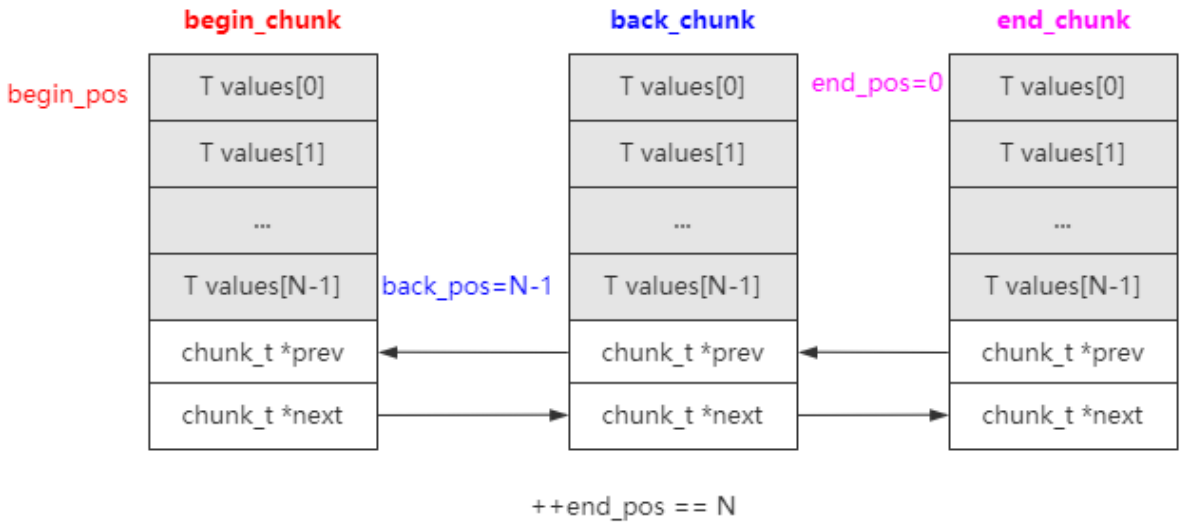
```
//  Adds an element to the back end of the queue.
inline void push()
{
    back_chunk = end_chunk;
    back_pos = end_pos;              // 更新可写的位置，根据end_pos取更新

    if (++end_pos != N) //end_pos!=N表明这个chunk还没有满
        return;

    chunk_t *sc = spare_chunk.xchg(NULL);        // 为什么设置为NULL？ 因为如果把之前值取出来了则没有spare chunk了，所以设置为NULL
    if (sc)      // 如果有spare chunk则继续复用它
    {
        end_chunk->next = sc;
        sc->prev = end_chunk;
    }
    else         // 没有则重新分配
    {
        end_chunk->next = (chunk_t *)malloc(sizeof(chunk_t));    // 分配一个chunk
        alloc_assert(end_chunk->next);
        end_chunk->next->prev = end_chunk;
    }
    end_chunk = end_chunk->next;
    end_pos = 0;
}
```

这里分为两种情况：

- 第一种情况：++end_pos != N，说明当前chunk还有空余的位置可以继续插入新元素；
- 第二种情况：++end_pos == N，说明该chunk只有[N-1]的索引位置可以写入元素了，需要再分配一个chunk空间。
    - 需要新分配chunk时，先尝试从spare_chunk获取，如果获取到则直接使用，如果spare_chunk为NULL则需要重新分配chunk。
    - 最终都是要更新end_chunk和end_pos。



第一种情况 ++end_pos != N，此时 back_pos和end_pos相差一个位置，即是 (back_pos +1)%N == end_pos。



第二种情况：++end_pos == N，说明该chunk只有N-1的位置可以写入元素了，需要再分配一个chunk空间。

pop函数

这里主要更新下一次读取的位置，并检测是否需要释放chunk（先保存到spare_chunk，然后检测spare_chunk返回值是否为空，如果返回值不为空说明之前有保存chunk，但我们只能保存一个chunk，所以把之前的chunk释放掉）

```
//  Removes an element from the front end of the queue.
inline void pop()
{
    if (++begin_pos == N)       // 删除满一个chunk才回收chunk
    {
        chunk_t *o = begin_chunk;
        begin_chunk = begin_chunk->next;    // 更新begin_chunk位置
        begin_chunk->prev = NULL;
        begin_pos = 0;
```

```
        //  'o' has been more recently used than spare_chunk,
        //  so for cache reasons we'll get rid of the spare and
        //  use 'o' as the spare.
        chunk_t *cs = spare_chunk.xchg(o);  //由于局部性原理，总是保存最新的空闲块而释放先前的空闲快
        free(cs);
    }
}
```

整个chunk的元素都被取出队列才去回收chunk，而且是把最后回收的chunk保存到spare_chunk，然后释放之前保存的chunk。
这里有两个点需要注意：

1. pop掉的元素，其销毁工作交给调用者完成，即是pop前调用者需要通过front()接口读取并进行销毁（比如动态分配的对象）。
2. 空闲块的保存，要求是原子操作。因为闲块是读写线程的共享变量，因为在push中也使用了spare_chunk。

源码分析-ypipe_t

ypipe_t在yqueue_t的基础上构建一个单写单读的无锁队列

最核心的点：

- **w: 用来控制是否需要唤醒读端，当读端没有数据可以读取的时候，将c变量设置为NULL**
- **f: 用来控制写入位置，当该f被更新到c的时候读端才能看到写入的数据**
- **r: 用来控制可读位置，特别重点注意，这个r不是读位置的索引，而是读位置==r的时候说明已经队列为空了。**

类接口和变量

```
template <typename T, int N>
class ypipe_t
{
public:
    //  Initialises the pipe.
    inline ypipe_t();

    //  The destructor doesn't have to be virtual. It is mad virtual
    //  just to keep ICC and code checking tools from complaining.
    inline virtual ~ypipe_t();


    //  Write an item to the pipe.  Don't flush it yet. If incomplete is
    //  set to true the item is assumed to be continued by items
    //  subsequently written to the pipe. Incomplete items are neverflushed down the stream.
    // 写入数据，incomplete参数表示写入是否还没完成，在没完成的时候不会修改flush指针，即这部分数据不会让读线程看到。
    inline void write(const T &value_, bool incomplete_);
```

```
    //  Pop an incomplete item from the pipe. Returns true is such
    //  item exists, false otherwise.
    inline bool unwrite(T *value_);

    //  Flush all the completed items into the pipe. Returns false if
    //  the reader thread is sleeping. In that case, caller is obliged to
    //  wake the reader up before using the pipe again.
    // 刷新所有已经完成的数据到管道，返回false意味着读线程在休眠，在这种情况下调用者需要唤醒读线程。
    inline bool flush();

    //  Check whether item is available for reading.
    // 这里面有两个点，一个是检查是否有数据可读，一个是预取
    inline bool check_read();

    //  Reads an item from the pipe. Returns false if there is no value.
    //  available.
    inline bool read(T *value_);

    //  Applies the function fn to the first elemenent in the pipe
    //  and returns the value returned by the fn.
    //  The pipe mustn't be empty or the function crashes.
    inline bool probe(bool (*fn)(T &));

protected:
    //  Allocation-efficient queue to store pipe items.
    //  Front of the queue points to the first prefetched item, back of
    //  the pipe points to last un-flushed item. Front is used only by
    //  reader thread, while back is used only by writer thread.
    yqueue_t<T, N> queue;

    //  Points to the first un-flushed item. This variable is used
    //  exclusively by writer thread.
    T *w;//指向第一个未刷新的元素,只被写线程使用

    //  Points to the first un-prefetched item. This variable is used
    //  exclusively by reader thread.
    T *r;//指向第一个还没预提取的元素，只被读线程使用

    //  Points to the first item to be flushed in the future.
    T *f;//指向下一轮要被刷新的一批元素中的第一个

    //  The single point of contention between writer and reader thread.
    //  Points past the last flushed item. If it is NULL,
    //  reader is asleep. This pointer should be always accessed using
    //  atomic operations.
    atomic_ptr_t<T> c;//读写线程共享的指针，指向每一轮刷新的起点（看代码的时候会详细说）。当c为空时，表示读线程睡眠（只会在读线程中被设置为空）

    //  Disable copying of ypipe object.
    ypipe_t(const ypipe_t &);
    const ypipe_t &operator=(const ypipe_t &);
};
```

主要变量：

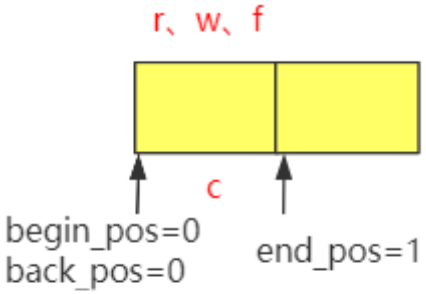
// Points to the first un-flushed item. This variable is used exclusively by writer thread.
T *w;//指向第一个未刷新的元素,只被写线程使用
// Points to the first un-prefetched item. This variable is used exclusively by reader thread.
T *r;//指向第一个还没预提取的元素，只被读线程使用
// Points to the first item to be flushed in the future.
T *f;//指向下一轮要被刷新的一批元素中的第一个
// The single point of contention between writer and reader thread.
// Points past the last flushed item. If it is NULL,reader is asleep.
// This pointer should be always accessed using atomic operations.
atomic_ptr_t c;//读写线程共享的指针，指向每一轮刷新的起点（看代码的时候会详细说）。当c为空时，表示读线程睡眠（只会在读线程中被设置为空）

主要接口：

- void write (const T &value, bool incomplete)：写入数据，incomplete参数表示写入是否还没完成，在没完成的时候不会修改flush指针，即这部分数据不会让读线程看到。
- bool flush ()：刷新所有已经完成的数据到管道，返回false意味着读线程在休眠，在这种情况下调用者需要唤醒读线程。
- bool read (T *value_):读数据，将读出的数据写入value指针中，返回false意味着没有数据可读。

ypipe_t ()初始化

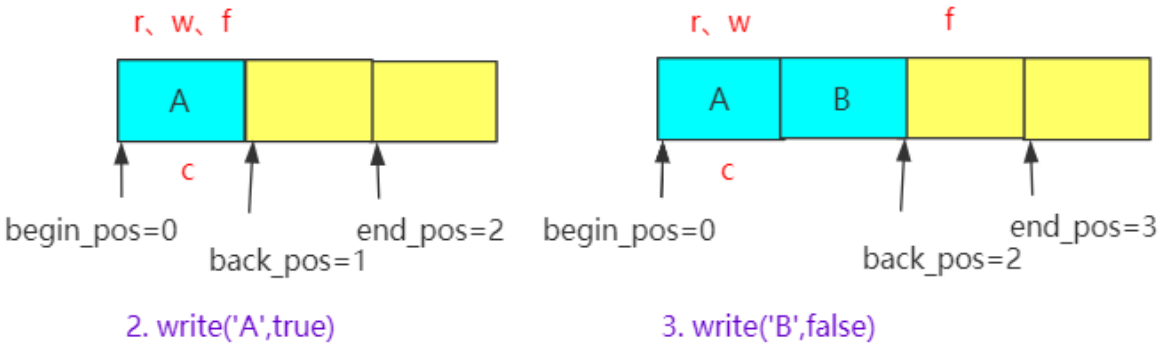
```
inline ypipe_t()
{
    //  Insert terminator element into the queue.
    queue.push();//yqueue_t的尾指针加1，开始back_chunk为空，现在back_chunk指向第一个chunk_t块的第一个位置

    //  Let all the pointers to point to the terminator.
    //  (unless pipe is dead, in which case c is set to NULL).
    r = w = f = &queue.back();//就是让r、w、f、c四个指针都指向这个end迭代器
    c.set(&queue.back()); // 保存[0]索引的位置
}
```



1. 初始化ypipe_t()

write函数

```
//  Write an item to the pipe.  Don't flush it yet. If incomplete is
//  set to true the item is assumed to be continued by items
//  subsequently written to the pipe. Incomplete items are neverflushed down the
stream.
// 写入数据，incomplete参数表示写入是否还没完成，在没完成的时候不会修改flush指针，即这部分数
据不会让读线程看到。
inline void write(const T &value_, bool incomplete_)
{
    //  Place the value to the queue, add new terminator element.
    queue.back() = value_;
    queue.push();   // 更新下一次写的位置

    //  Move the "flush up to here" poiter.
    if (!incomplete_)           // 如果f不更新，flush的时候 read也是没有数据
        f = &queue.back();      // 记录要刷新的位置
}
```



write(val, false); 触发更新f的位置。f实际是back_pos的位置，**即是下一次可以写入的位置。**

flush函数

主要是将**w更新到f**的位置，说明已经写到的位置。

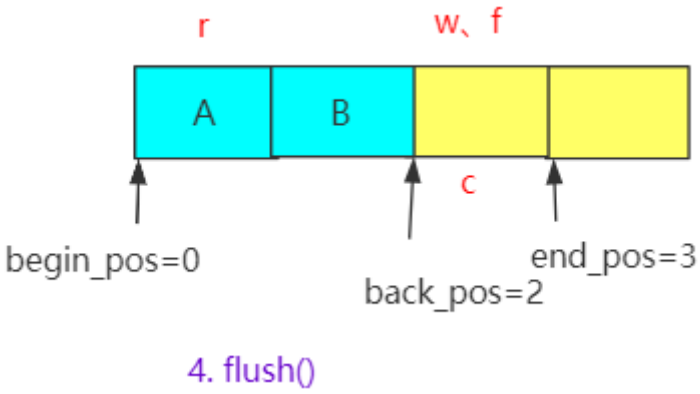
cas函数，原子操作，**线程安全**，把私有成员ptr指针与参数cmp_指针比较：

- 如果相等，就把ptr设置为参数val_的值，返回ptr设置之前的值；
- 如果不相等直接返回ptr值。

```
//  Flush all the completed items into the pipe. Returns false if
//  the reader thread is sleeping. In that case, caller is obliged to
//  wake the reader up before using the pipe again.
// 刷新所有已经完成的数据到管道，返回false意味着读线程在休眠，在这种情况下调用者需要唤醒读线
程。
inline bool flush()
{
    //  If there are no un-flushed items, do nothing.
    if (w == f)     // 不需要刷新，即是还没有新元素加入，在 3.步骤后 w=0,f=2
        return true;
```

```
    //  Try to set 'c' to 'f'.
    // read导致数据没有可读取后，c会被置为NULL，这点非常重要
    // 如果c为NULL，则此时w和null不相等，返回null，null和w不相等则 走return false逻辑， c被设置为f
    if (c.cas(w, f) != w)   //   尝试将c设置为f，即是准备更新w的位置
    {  //  - 如果c==w，则更新f->c，并返回原来的c；
        // - 如果c!=w，  则返回原来的c；
        //  Compare-and-swap was unseccessful because 'c' is NULL.
        //  This means that the reader is asleep. Therefore we don't
        //  care about thread-safeness and update c in non-atomic
        //  manner. We'll return false to let the caller know
        //  that reader is sleeping.
        c.set(f);
        w = f;
        return false;       // 线程看到flush返回false之后会发送一个消息给读线程，这个是需要写业务去做处理
    }
    //  Reader is alive. Nothing special to do now. Just move
    //  the 'first un-flushed item' pointer to 'f'.
    w = f;
    return true;
}
```

刷新之后，w、f、c、r的关系：



4. flush()

flush后w一定为f，w的作用主要是用来控制return false/true。

read函数

怎么检测是否有数据可以读取?

r实际上是用来控制可以读取到的位置（注意不是读到r，而是r的前一位置可以读取，r位置是不可以读取的），当front和r重叠的时候说明没有数据可以读取。

```
//  Check whether item is available for reading.
    // 这里面有两个点，一个是检查是否有数据可读，一个是预取
    inline bool check_read()
    {
        //  Was the value prefetched already? If so, return.
        // 第一次进来 r == &queue.front() 这里不成立
```

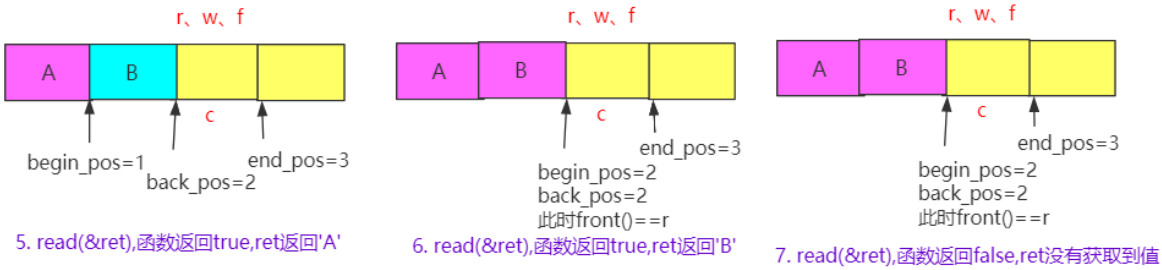
```
        if (&queue.front() != r && r) //判断是否在前几次调用read函数时已经预取数据了
return true;
            return true; // 有数据可以读取

        //  There's no prefetched value, so let us prefetch more values.
        //  Prefetching is to simply retrieve the
        //  pointer from c in atomic fashion. If there are no
        //  items to prefetch, set c to NULL (using compare-and-swap).
        // 如果此时还没有写入数据，则c和&queue.front()相等，则c被设置为NULL
        // 如果此时已经写入数据，则c和&queue.front()不相等，返回w的位置
        r = c.cas(&queue.front(), NULL);//尝试预取数据， r其实是指向可以读取到的位置，并
不是用来表示读索引位置

        //  If there are no elements prefetched, exit.
        //  During pipe's lifetime r should never be NULL, however,
        //  it can happen during pipe shutdown when items
        //  are being deallocated.
        if (&queue.front() == r || !r)//判断是否成功预取数据，如果重叠或者为空说明没有数
据可以读取
            return false;

        //  There was at least one value prefetched.
        return true;
    }

//  Reads an item from the pipe. Returns false if there is no value.
    //  available.
    inline bool read(T *value_)
    {
        //  Try to prefetch a value.
        if (!check_read())  // 检测有没有数据可以读取
            return false;

        //  There was at least one value prefetched.
        //  Return it to the caller.
        *value_ = queue.front();
        queue.pop();
        return true;
    }
```

如果：

- 指针r指向的是队头元素（r==&queue.front()） （**最核心的一点r不是我们read的位置索引，而是用来识别我们可以读取到哪个位置就不能再读**）
- 或者r没有指向任何元素（NULL）

则说明队列中并没有可读的数据，这个时候check_read尝试去预取数据。所谓的预取就是令 r=c (cas函数就是返回c本身的值，看上面关于cas的实现)， 而c在write中被指向f（见上图），这时从queue.front()到f这个位置的数据都被预取出来了，然后每次调用read都能取出一段。值得注意的是，当c==&queue.front()时，代表数据被取完了，这时把c指向NULL，接着读线程会睡眠，这也是给写线程检查读线程是否睡眠的标志。

继续上面写入AB数据的场景，第一次调用read时，会先check_read，把指针r指向指针c的位置（所谓的预取），这时r,c,w,f的关系如下：

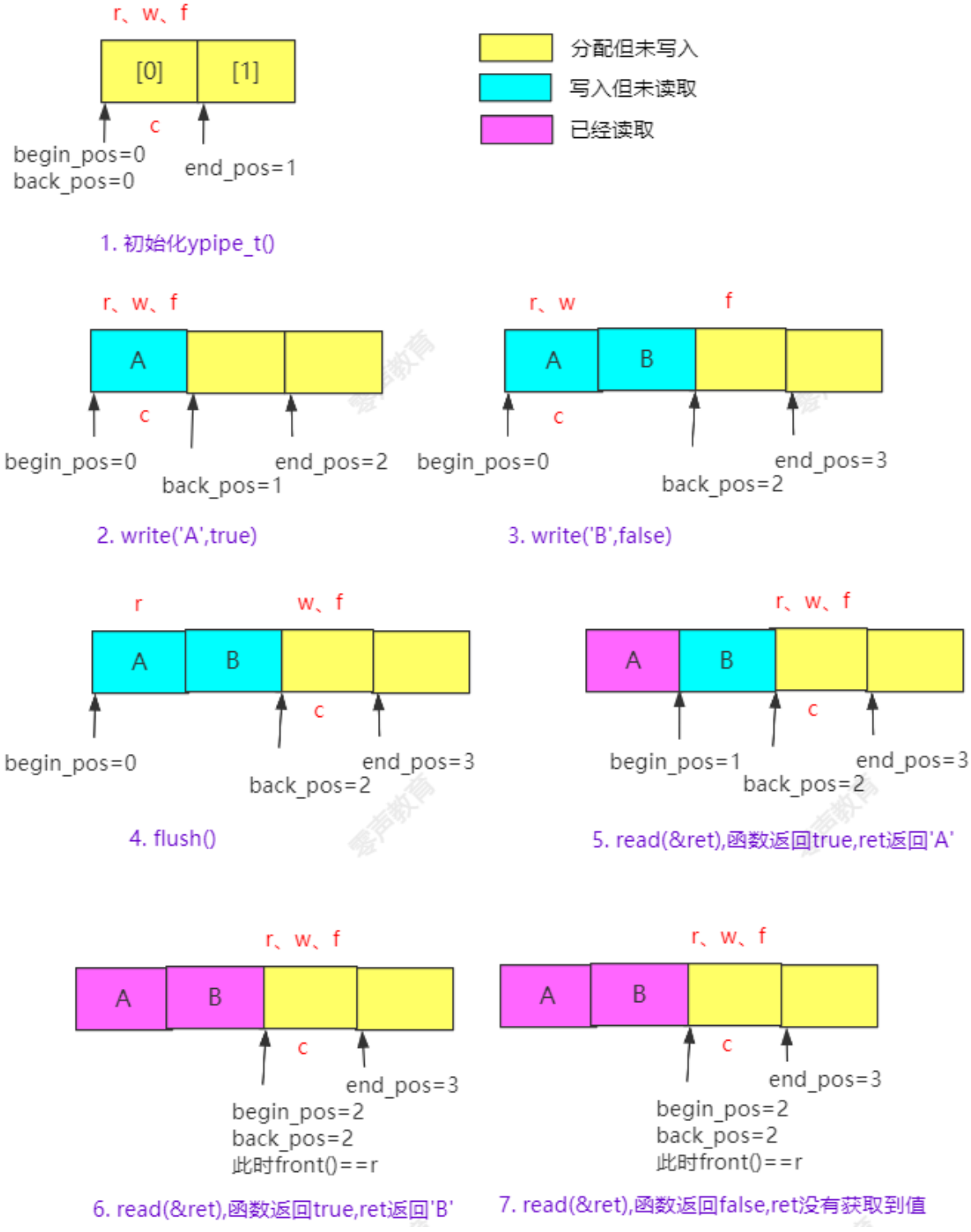


在“7.read(&ret),函数返回false,ret没有获取到值”的时候，front()和r相等。

- 如果此时在r = c.cas(&queue.front(), NULL); 执行时没有flush的操作。则说明没有数据可以读取，最终返回false；
- 如果在r = c.cas(&queue.front(), NULL); 之前写入方write新数据后并调用了flush，则r被更新，最终返回true。

而_c指针，则是读写线程都可以操作，因此需要使用原子的CAS操作来修改，它的可能值有以下几种：

- NULL：**读线程设置，此时意味着已经没有数据可读，读线程在休眠。**
- 非零：写线程设置，这里又区分两种情况：
    - 旧值为_w的情况下，`cas(_w,_f)`操作修改为`_f`，意味着如果原先的值为`_w`，则原子性的修改为`_f`，表示有更多已被刷新的数据可读。
    - 在旧值为`NULL`的情况下，此时读线程休眠，因此可以安全的设置为当前`_f`指针的位置。



7 基于循环循环数组的无锁队列

重点

- ArrayLockFreeQueue数据结构，可以理解为一个环形数组；
- 多线程写入时候，m_maximumReadIndex、m_writeIndex**索引如何更新**
- 在更新m_maximumReadIndex**的时候为什么要让出cpu**；
- **多线程读取的时候**，m_readIndex如何更新。
- **可读位置是由m_maximumReadIndex控制**，而不是m_writeIndex去控制的。
    - **m_maximumReadIndex的更新由m_writeIndex。**

类接口和变量

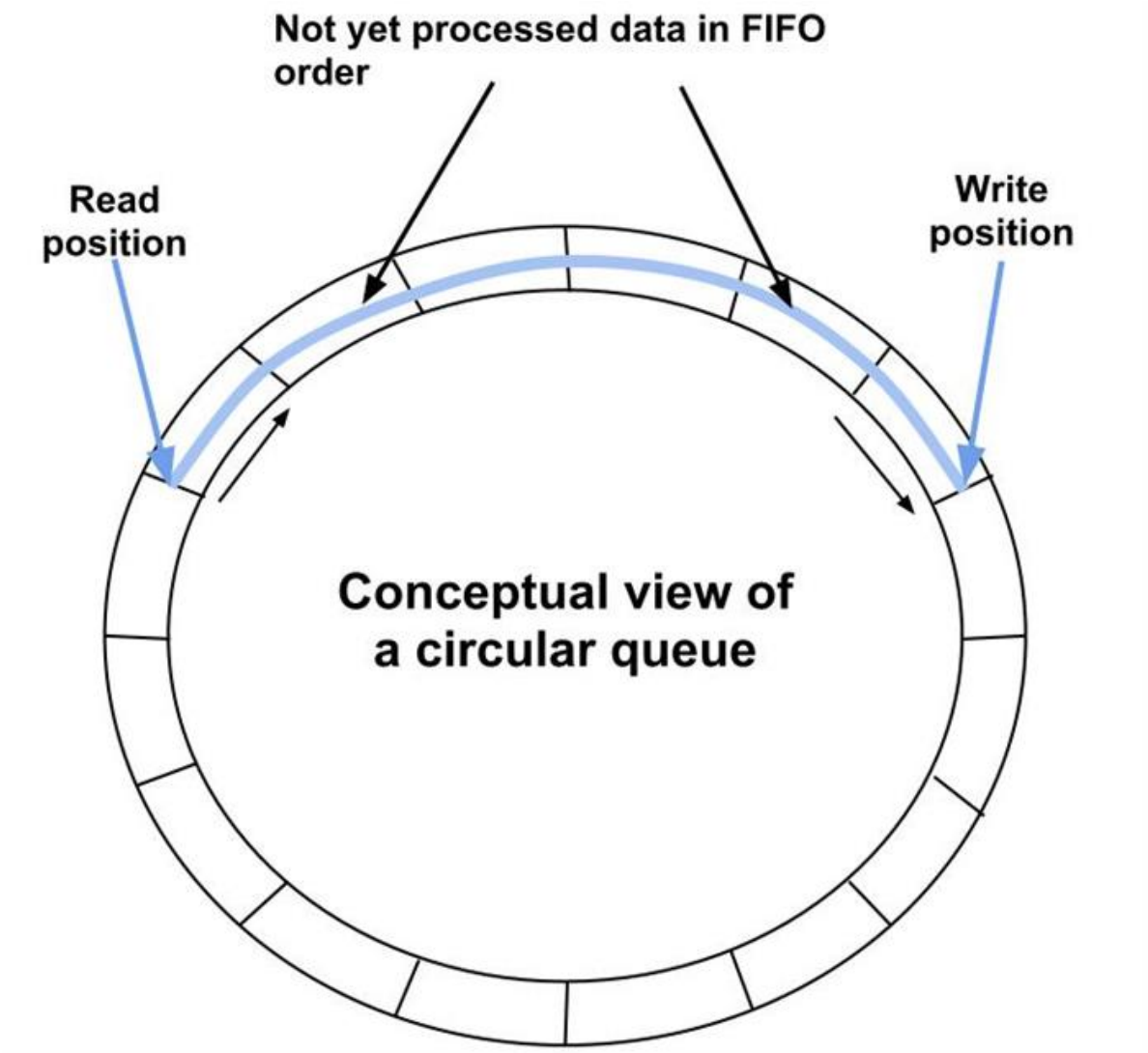
```
template <typename ELEM_T, QUEUE_INT Q_SIZE = ARRAY_LOCK_FREE_Q_DEFAULT_SIZE>
class ArrayLockFreeQueue
{
public:

    ArrayLockFreeQueue();
    virtual ~ArrayLockFreeQueue();

    QUEUE_INT size();

    bool enqueue(const ELEM_T &a_data);  // 入队列

    bool dequeue(ELEM_T &a_data);        // 出队列

    bool try_dequeue(ELEM_T &a_data);    // 尝试入队列

private:

    ELEM_T m_thequeue[Q_SIZE];

    volatile QUEUE_INT m_count;  // 队列的元素格式
    volatile QUEUE_INT m_writeIndex;//新元素入列时存放位置在数组中的下标

    volatile QUEUE_INT m_readIndex;// 下一个出列元素在数组中的下标

    volatile QUEUE_INT m_maximumReadIndex; //最后一个已经完成入列操作的元素在数组中的下
标

    inline QUEUE_INT countToIndex(QUEUE_INT a_count);
};
```

三种不同下标：

- m_count; // 队列的元素个数
- **m_writeIndex**;//新元素入列时存放位置在数组中的下标
- **m_readIndex**;// 下一个出列的元素在数组中的下标
- **m_maximumReadIndex**; //最后一个已经完成入列操作的元素在数组中的下标。如果它的值跟m_writeIndex不一致，表明有写请求尚未完成。这意味着，**有写请求成功申请了空间但数据还没完全写进队列。**所以如果有线程要读取，必须要等到写线程将数据完全写入到队列之后。

必须指明的是使用3种不同的下标都是必须的，因为队列允许任意数量的生产者和消费者围绕着它工作。已经存在一种基于循环数组的无锁队列，使得唯一的生产者和唯一的消费者可以良好的工作[11]。它的实现相当简洁非常值得阅读。

该程序使用gcc内置的**sync_bool_compare_and_swap，但重新做了宏定义封装。**
#define CAS(a_ptr, a_oldVal, a_newVal) sync_bool_compare_and_swap(a_ptr, a_oldVal, a_newVal
)

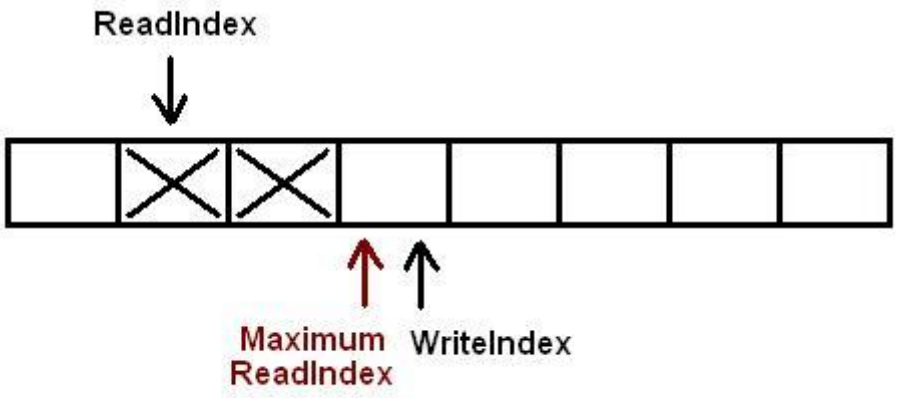


enqueue入队列

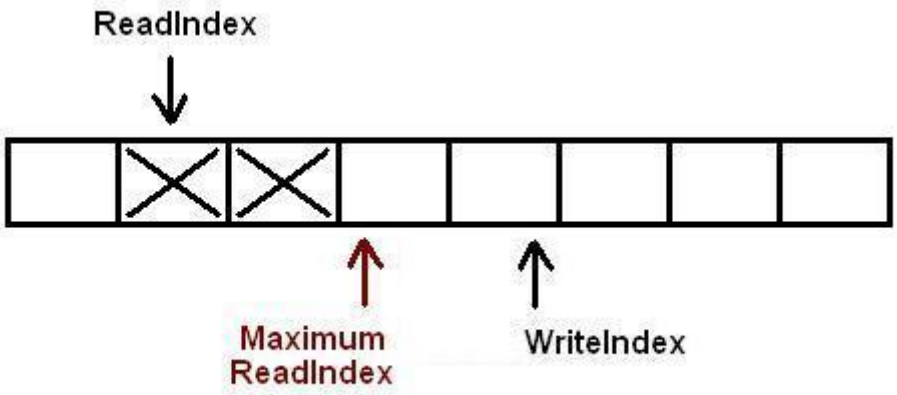
```
template <typename ELEM_T, QUEUE_INT Q_SIZE>
bool ArrayLockFreeQueue<ELEM_T, Q_SIZE>::enqueue(const ELEM_T &a_data)
{
    QUEUE_INT currentWriteIndex;                // 获取写指针的位置
    QUEUE_INT currentReadIndex;
    do
    {
        currentWriteIndex = m_writeIndex;
        currentReadIndex = m_readIndex;
        if(countToIndex(currentWriteIndex + 1) ==
            countToIndex(currentReadIndex))
        {
            return false;   //队列未空
        }
    } while(!CAS(&m_writeIndex, currentWriteIndex, (currentWriteIndex+1)));
    // We know now that this index is reserved for us. Use it to save the data
    m_thequeue[countToIndex(currentWriteIndex)] = a_data;

    // update the maximum read index after saving the data. It wouldn't fail if
there is only one thread
    // inserting in the queue. It might fail if there are more than 1 producer
threads because this
    // operation has to be done in the same order as the previous CAS
    while(!CAS(&m_maximumReadIndex, currentWriteIndex, (currentWriteIndex + 1)))
```

```
    {
         // this is a good place to yield the thread in case there are more
        // software threads than hardware processors and you have more
        // than 1 producer thread
        // have a look at sched_yield (POSIX.1b)
        sched_yield();
    }

    AtomicAdd(&m_count, 1);

    return true;

}
```

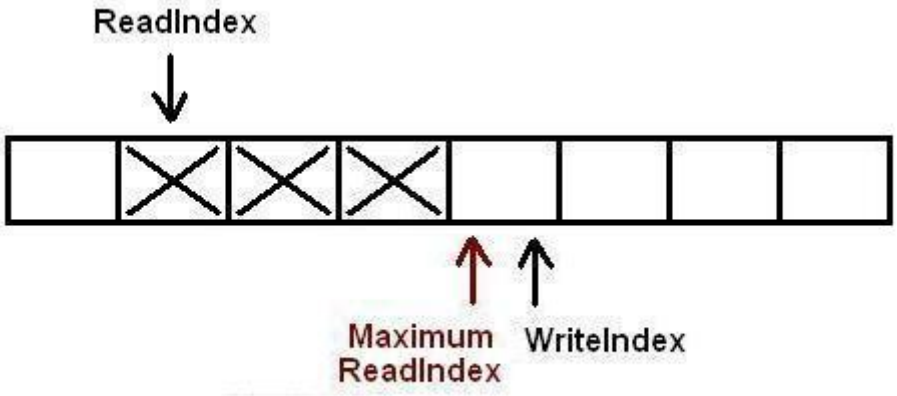
下面的图示特别重要，需要耐心分析。
以下插图展示了对队列执行操作时各下标是如何变化的。如果一个位置被标记为X，标识这个位置里存放了数据。空白表示位置是空的。对于下图的情况,队列中存放了两个元素。WriteIndex指示的位置是新元素将会被插入的位置。ReadIndex指向的位置中的元素将会在下一次pop操作中被弹出。



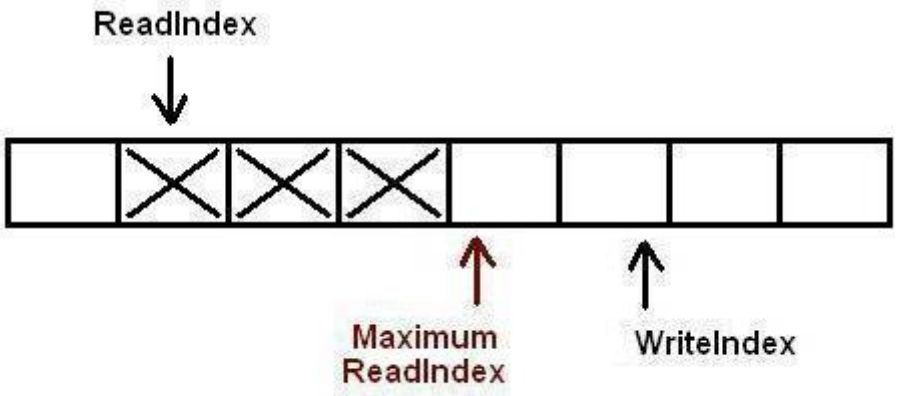
当生产者准备将数据插入到队列中,它首先通过增加WriteIndex的值来申请空间。MaximumReadIndex指向最后一个存放有效数据的位置(也就是实际的队列尾)。



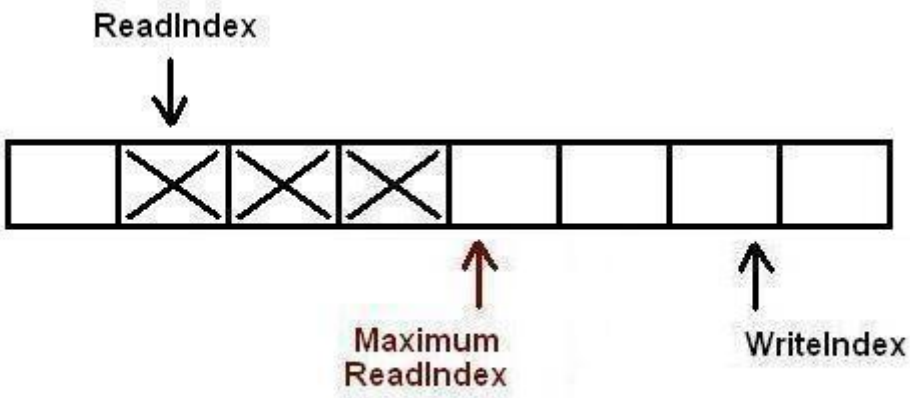
一旦空间的申请完成,生产者就可以将数据拷贝到刚刚申请到的位置中。完成之后增加MaximumReadIndex使得它与WriteIndex的一致。



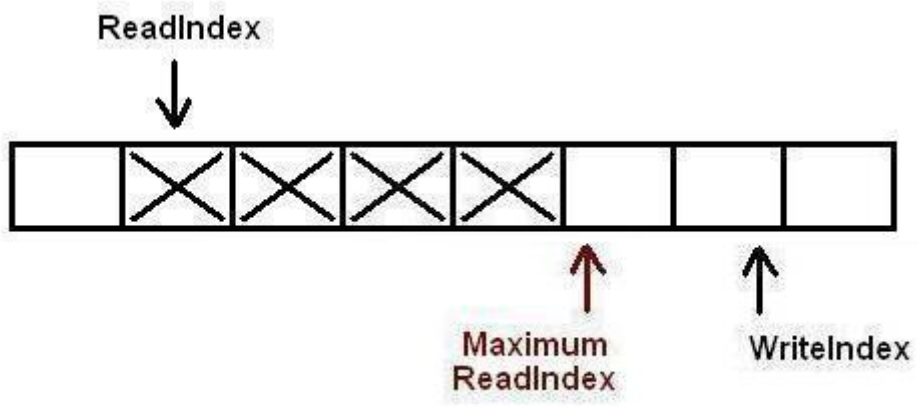
现在队列中有3个元素，接着又有一个生产者尝试向队列中插入元素。



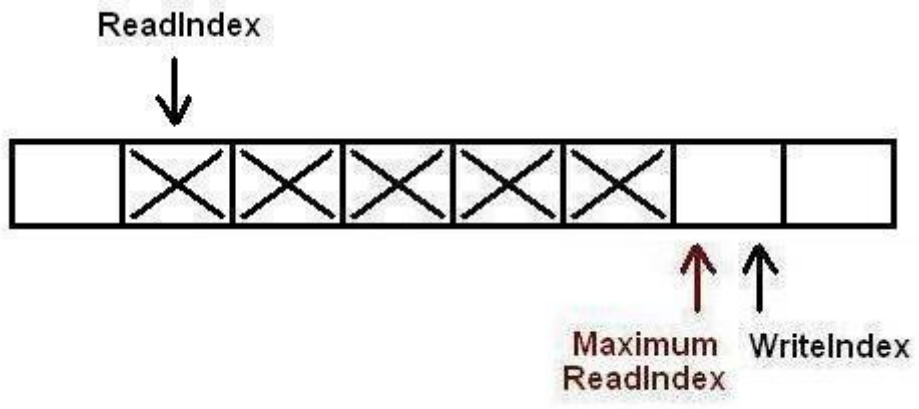
在第一个生产者完成数据拷贝之前，又有另外一个生产者申请了一个新的空间准备拷贝数据。现在有两个生产者同时向队列插入数据。



现在生产者开始拷贝数据，在完成拷贝之后，对MaximumReadIndex的递增操作必须严格遵循一个顺序：第一个生产者线程首先递增MaximumReadIndex，接着才轮到第二个生产者。这个顺序必须被严格遵守的原因是，我们必须保证数据被完全拷贝到队列之后才允许消费者线程将其出列。
（while(!CAS(&m_maximumReadIndex, currentWriteIndex, (currentWriteIndex + 1)))
{sched_yield(); } 让出cpu的目的也是为了让排在最前面的生产者完成m_maximumReadIndex的更新）



第一个生产者完成了数据拷贝，并对MaximumReadIndex完成了递增，现在第二个生产者可以递增MaximumReadIndex了。



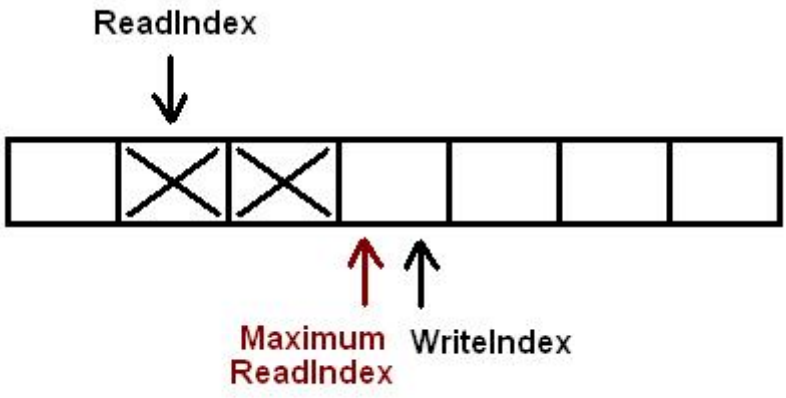
第二个生产者完成了对MaximumReadIndex的递增,现在队列中有5个元素。

dequeue出队列

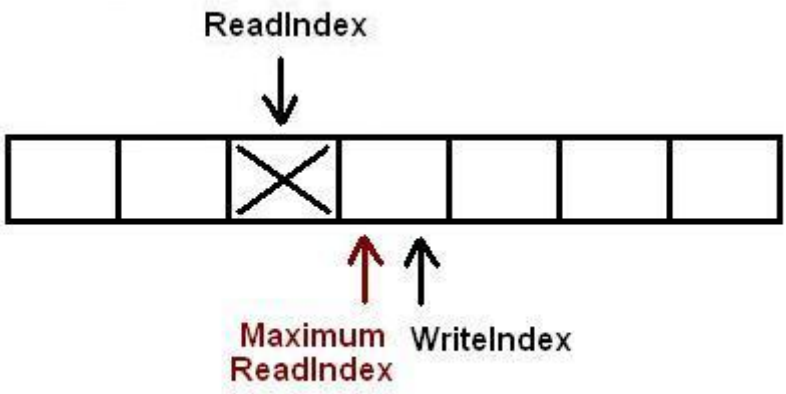
```
template <typename ELEM_T, QUEUE_INT Q_SIZE>
bool ArrayLockFreeQueue<ELEM_T, Q_SIZE>::dequeue(ELEM_T &a_data)
{
    QUEUE_INT currentMaximumReadIndex;
    QUEUE_INT currentReadIndex;

    do
    {
         // to ensure thread-safety when there is more than 1 producer thread
        // a second index is defined (m_maximumReadIndex)
        currentReadIndex = m_readIndex;
        currentMaximumReadIndex = m_maximumReadIndex;

        if(countToIndex(currentReadIndex) ==
            countToIndex(currentMaximumReadIndex))  // 判断是否有可读数据
        {
            // the queue is empty or
            // a producer thread has allocate space in the queue but is
            // waiting to commit the data into it
            return false;
        }
        // retrieve the data from the queue
        a_data = m_thequeue[countToIndex(currentReadIndex)];
```

```
        // try to perfrom now the CAS operation on the read index. If we succeed
        // a_data already contains what m_readIndex pointed to before we
        // increased it，因为其他线程也可能在读取数据。
        if(CAS(&m_readIndex, currentReadIndex, (currentReadIndex + 1)))
        {
            AtomicSub(&m_count, 1); // 真正读取到了数据
            return true;
        }
    } while(true);

    assert(0);
     // Add this return statement to avoid compiler warnings
    return false;

}
```

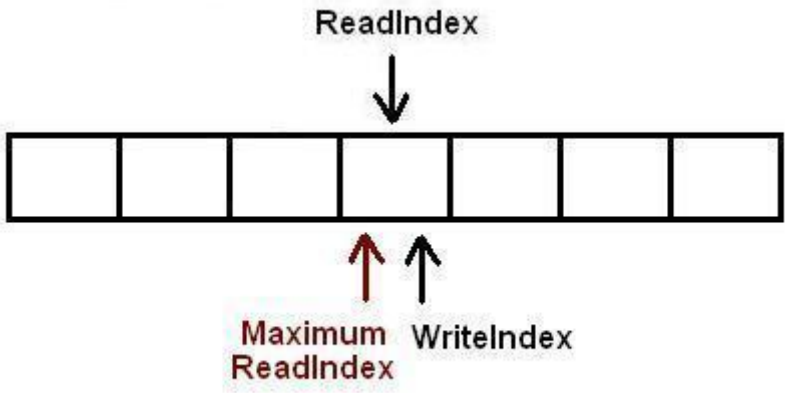
以下插图展示了元素出列的时候各种下标是如何变化的,队列中初始有2个元素。WriteIndex指示的位置是新元素将会被插入的位置。ReadIndex指向的位置中的元素将会在下一次pop操作中被弹出。



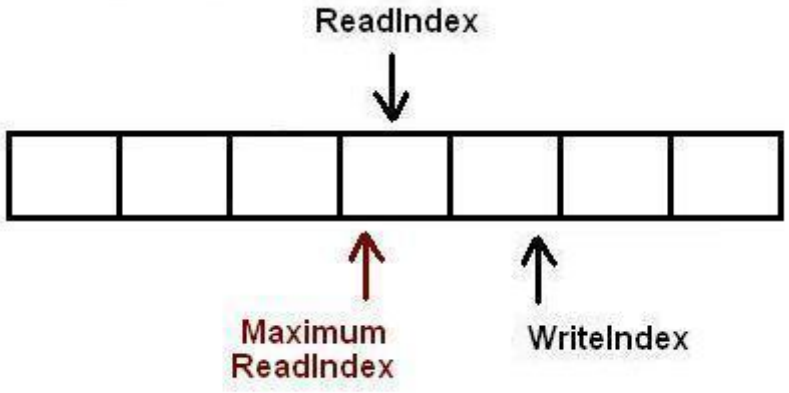
消费者线程拷贝数组ReadIndex位置的元素，然后尝试用CAS操作将ReadIndex加1。如果操作成功消费者成功的将数据出列。因为CAS操作是原子的，所以只有唯一的线程可以在同一时刻更新ReadIndex的值。如果操作失败，读取新的ReadIndex值，以重复以上操作(copy数据，CAS)。



现在又有一个消费者将元素出列，队列变成空。



现在有一个生产者正在向队列中添加元素。它已经成功的申请了空间，但尚未完成数据拷贝。任何其它企图从队列中移除元素的消费者都会发现队列非空(因为writeIndex不等于readIndex)。但它不能读取readIndex所指向位置中的数据，因为readIndex与MaximumReadIndex相等。消费者将会在do循环中不断的反复尝试，直到生产者完成数据拷贝增加MaximumReadIndex的值，或者队列变成空(这在多个消费者的场景下会发生)。



当生产者完成数据拷贝，队列的大小是1，消费者线程可以读取这个数据了.

在多于一个生产者线程的情况下yielding处理器的必要性

读者可能注意到了enqueue函数中使用了sched_yield()来主动的让出处理器，对于一个声称无锁的算法而言，这个调用看起来有点奇怪。正如文章开始的部分解释过的，多线程环境下影响性能的其中一个因素就是Cache损坏。而产生Cache损坏的一种情况就是一个线程被抢占，操作系统需要保存被抢占线程的上下文，然后将被选中作为下一个调度线程的上下文载入。此时Cache中缓存的数据都会失效，因为它是被抢占线程的数据而不是新线程的数据。
所以，当此算法调用sched_yield()意味着告诉操作系统："我要把处理器时间让给其它线程，因为我要等待某件事情的发生"。无锁算法和通过阻塞机制同步的算法的一个主要区别在于**无锁算法不会阻塞在线程同步上**，那么为什么在这里我们要主动请求操作系统抢占自己呢?这个问题的答案没那么简单。它与有多少个生产者线程在并发的往队列中存放数据有关：每个生产者线程所执行的CAS操作都必须严格遵循FIFO次序，一个用于申请空间，另一个用于通知消费者数据已经写入完成可以被读取了。

如果我们的应用程序只有唯一的生产者操作这个队列，sche_yield()将永远没有机会被调用，第2个CAS操作永远不会失败。因为在一个生产者的情况下没有人能破坏生产者执行这两个CAS操作的FIFO顺序。

而当多于一个生产者线程往队列中存放数据的时候，问题就出现了。概括来说，一个生产者通过第1个CAS操作申请空间，然后将数据写入到申请到的空间中，然后执行第2个CAS操作通知消费者数据准备完毕可供读取了。这第2个CAS操作必须遵循FIFO顺序，也就是说，如果A线程第首先执行完第一个CAS操作，那么它也要第1个执行完第2个CAS操作，如果A线程在执行完第一个CAS操作之后停止，然后B线程

执行完第1个CAS操作，那么B线程将无法完成第2个CAS操作，因为它要等待A先完成第2个CAS操作。而这就是问题产生的根源。让我们考虑如下场景，3个消费者线程和1个消费者线程：

- 线程1，2，3按顺序调用第1个CAS操作申请了空间。那么它们完成第2个CAS操作的顺序也应该与这个顺序一致，1，2，3。
- 线程2首先尝试执行第2个CAS，但它会失败，因为线程1还没完成它的第2此CAS操作呢。同样对于线程3也是一样的。
- 线程2和3将会不断的调用它们的第2个CAS操作，直到线程1完成它的第2个CAS操作为止。
- 线程1最终完成了它的第2个CAS，现在线程3必须等线程2先完成它的第2个CAS。
- 线程2也完成了，最终线程3也完成。

在上面的场景中，生产者可能会在第2个CAS操作上自旋一段时间，用于等待先于它执行第1个CAS操作的线程完成它的第2次CAS操作。在一个物理处理器数量大于操作队列线程数量的系统上，这不会有太严重的问题：因为每个线程都可以分配在自己的处理器上执行，它们最终都会很快完成各自的第2次CAS操作。虽然算法导致线程处理忙等状态，但这正是我们所期望的，因为这使得操作更快的完成。也就是说在这种情况下我们是不需要sche_yield()的，它完全可以从代码中删除。
但是，在一个物理处理器数量少于线程数量的系统上，sche_yield()就变得至关重要了。让我们再次考查上面3个线程的场景，当线程3准备向队列中插入数据：如果线程1在执行完第1个CAS操作，在执行第2个CAS操作之前被抢占，那么线程2，3就会一直在它们的第2个CAS操作上忙等(它们忙等，不让出处理器，线程1也就没机会执行，它们就只能继续忙等)，直到线程1重新被唤醒，完成它的第2个CAS操作。这就是需要sche_yield()的场合了，操作系统应该避免让线程2，3处于忙等状态。它们应该尽快的让出处理器让线程1执行，使得线程1可以把它的第2个CAS操作完成。这样线程2和3才能继续完成它们的操作。

8 源码测试

见上课源码包。

参考文档

【基于数组的无锁队列(译)】https://zhuanlan.zhihu.com/p/33985732
【A Fast Lock-Free Queue for C++】 https://moodycamel.com/blog/2013/a-fast-lock-free-queue-for-c++.htm#benchmarks
【A Fast General Purpose Lock-Free Queue for C++】https://moodycamel.com/blog/2014/a-fast-general-purpose-lock-free-queue-for-c++.htm#benchmarks
[10] sched_yield documentation
[11] lock-free single producer - single consumer circular queue
《说说无锁(Lock-Free)编程那些事》https://www.yuque.com/docs/share/31e0227e-1925-472c-93c5-3384fe43fd2f?#