# 探索C++无锁队列：多线程编程的高效利器

原创 往事敬秋风 深度Linux 2024年11月06日 17:19 湖南

在 C++ 的多线程编程世界中，有一个神奇的存在 —— 无锁队列。它宛如一座坚固的桥梁，横跨在多线程协作的鸿沟之上，成为提升程序性能和稳定性的关键角色。

深度Linux
曾就职国内知名企业项目经理，部门负责人等职务。研究领域：Windows&Linux平台C/...
228篇原创内容

公众号

随着计算机硬件的不断发展，多核处理器已经成为主流。多线程编程由此变得愈发重要，然而，传统基于锁的同步机制却逐渐暴露出诸多问题。在这一背景下，无锁队列应运而生，它宛如黑暗中的灯塔，为多线程程序的高效运行照亮了前行的道路。

想象一下，在一个繁忙的交通枢纽，传统的锁机制就像是交通管制中的红绿灯，虽然能维持秩序，但频繁的等待和切换也会造成拥堵。而无锁队列则更像是智能交通系统，让数据在多线程之间流畅地穿梭，无需不必要的停顿。它为我们打开了一扇通往更高效多线程编程的大门，让我们一同走进 C++ 无锁队列的奇妙世界吧。

## 一、无锁队列简介

无锁队列（Lock-Free Queue）是一种并发数据结构，用于在多线程环境下实现高效的数据交换。与传统的基于锁的队列相比，无锁队列使用了一些特殊的算法和技术，避免了线程之间的互斥操作，从而提高了并发性能和响应性。

无锁队列通常基于原子操作（atomic operations）或其他底层同步原语来实现，并且它们采用一些巧妙的方法来确保操作的正确性。主要思想是通过使用原子读写操作或类似的机制，在没有显式锁定整个队列的情况下实现线程安全。

典型的无锁队列算法有循环缓冲区（Circular Buffer）和链表（Linked List）等。循环缓冲区通常使用两个指针（head 和 tail）来表示队列的开始和结束位置，利用自旋、CAS (Compare-and-Swap) 等原子操作来进行入队和出队操作。链表则通过利用 CAS 操作插入或删除节点来实现并发访问。

在当今多核心优化的大背景下，无锁队列在多线程编程中起着至关重要的作用。多核心优化是当前游戏开发以及许多领域的重点课题，无论是工程实践还是算法研究，将工作并行化交由多线程去处理是极为普遍的场景。

在这种情况下，线程池与命令队列的组合常常被采用，而其中的命令队列就可以选择使用互斥锁或者无锁队列。由于命令队列的读写通常是较轻量级的操作，采用无锁队列能够获得比有锁操作更高的性能。无锁队列通过使用原子操作来确保线程安全，避免了锁的开销，包括上下文切换、线程调度延迟以及潜在的死锁问题。

在多处理器系统中，无锁队列可以更好地扩展。随着处理器数量的增加，使用锁的队列可能会遇到瓶颈，因为多个线程竞争同一个锁。而无锁队列通过减少这种竞争，可以提供更好的并行性。在实时系统中，无锁队列可以提供更高效的响应时间，确保系统能够及时处理各种任务。

**优点：**

- 提供了更好的并发性能，避免了互斥操作带来的性能瓶颈。
- 对于高度竞争情况下可以提供更好的可伸缩性。

**缺点：**

- 实现相对复杂，需要考虑并发安全和正确性问题。
- 在高度竞争的情况下可能出现自旋等待导致的性能损失。

## 二、无锁队列工作原理

无锁队列的原理是通过使用原子操作（atomic operations）或其他底层同步原语来实现并发安全。它们避免了传统锁机制中的互斥操作，以提高并发性能和响应性。典型的无锁队列算法有循环缓冲区（Circular Buffer）和链表（Linked List）等。

在循环缓冲区的实现中，通常使用两个指针来表示队列的开始位置和结束位置，即头指针（head）和尾指针（tail）。入队时，通过自旋、CAS (Compare-and-Swap) 等原子操作更新尾指针，并将元素放入相应位置。出队时，同样利用原子操作更新头指针，并返回对应位置上的元素。

链表实现无锁队列时，在插入或删除节点时使用 CAS 操作来确保只有一个线程成功修改节点的指针值。这样可以避免对整个链表进行加锁操作。

无论是循环缓冲区还是链表实现，关键点在于如何利用原子操作确保不同线程之间的协调与一致性。需要仔细处理并发情况下可能出现的竞争条件，并设计合适的算法来保证正确性和性能。

### 2.1队列操作模型

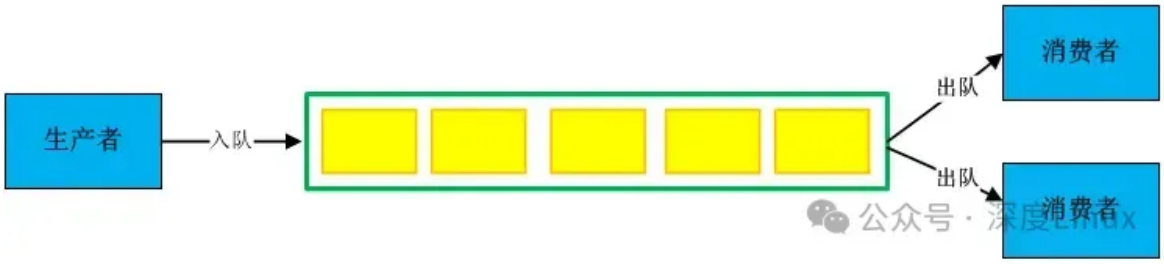
队列是一种非常重要的数据结构，其特性是先进先出（FIFO），符合流水线业务流程。在进程间通信、网络通信间经常采用队列做缓存，缓解数据处理压力。根据操作队列的场景分为：单生产者——单消费者、多生产者——单消费者、单生产者——多消费者、多生产者——多消费者四大模型。根据队列中数据分为：队列中的数据是定长的、队列中的数据是变长的。

（1）单生产者——单消费者

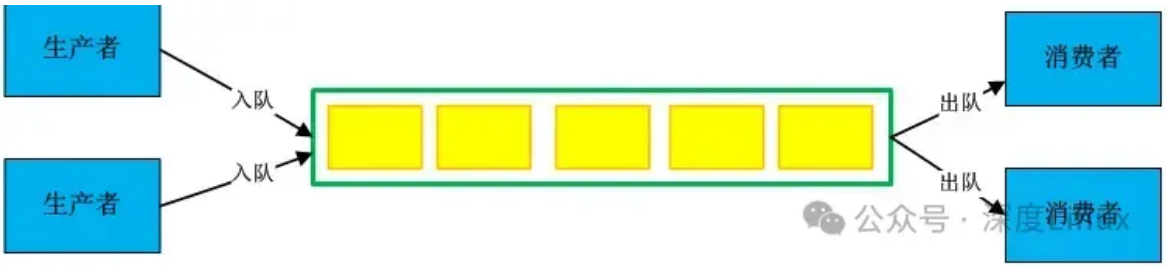


（2）多生产者——单消费者


（3）单生产者——多消费者




（4）多生产者——多消费者




## 2.2CAS操作

CAS即Compare and Swap，是所有CPU指令都支持CAS的原子操作（X86中CMPXCHG汇编指令），用于实现实现各种无锁（lock free）数据结构。

**CAS操作的C语言实现如下：**

```
bool compare_and_swap ( int *memory_location, int expected_value, int new_
{
    if (*memory_location == expected_value)
    {
        *memory_location = new_value;
        return true;
    }
    return false;
}
```

CAS用于检查一个内存位置是否包含预期值，如果包含，则把新值复赋值到内存位置。成功返回true，失败返回false。

（1）GGC对CAS支持，GCC4.1+版本中支持CAS原子操作。

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

（2）Windows对CAS支持，Windows中使用Windows API支持CAS。

```
LONG InterlockedCompareExchange(
  LONG volatile *Destination,
  LONG          ExChange,
  LONG          Comperand
);
```

（3）C11对CAS支持，C11 STL中atomic函数支持CAS并可以跨平台。

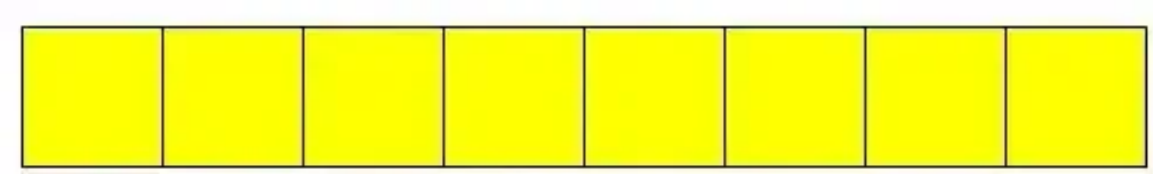
```
template< class T >
bool atomic_compare_exchange_weak( std::atomic* obj,T* expected, T desired
template< class T >
bool atomic_compare_exchange_weak( volatile std::atomic* obj,T* expected,
```

其它原子操作如下：

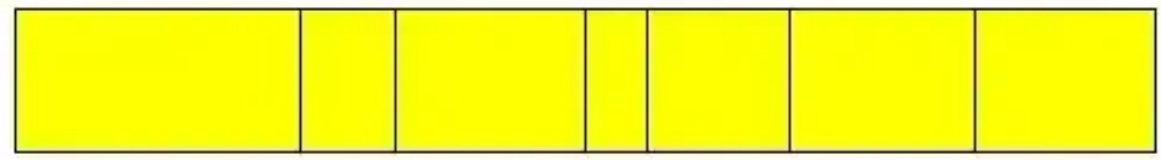
Fetch-And-Add：一般用来对变量做+1的原子操作
Test-and-set：写值到某个内存位置并传回其旧值

## 2.3队列数据定长与变长

（1）队列数据定长



（2）队列数据变长



**(1)问题描述**

在多线程环境下，原始队列会出现各种不可预料的问题。以两个线程同时写入为例，假设线程 A 和线程 B 同时对原始队列进行写入操作。首先看原始队列的入队伪代码：void Enqueue(Node *node){m_Tail->next = node;m_Tail = node;}，这个操作分为两步。当两个线程同时执行时，

可能出现这样的情况：线程 A 执行完第一步m_Tail->next = nodeC后，线程 B 开始执行并完成了整个入队操作，接着线程 A 继续执行第二步m_Tail = nodeB，这就导致了 Tail 指针失去与队列的链接，后加的节点从 Head 开始就访问不到了。这种情况会使得队列的状态变得混乱，无法保证数据的正确存储和读取。

**⑵解决方法**

为了解决上述问题，可以使用原子操作实现无锁同步。原子操作是不可分割的操作，CPU 的一个线程在执行原子操作时，不会被其他线程中断或抢占。其中，典型的原子操作有 Load / Store（读取与保存）、Test and Set（针对 bool 变量，如果为 true 则返回 true，如果为 false，则将变量置为 true 并返回 false）、Clear（将 bool 变量设为 false）、Exchange（将指定位置的值设置为传入值，并返回其旧值）等。

而 CAS（Compare And Swap）在实现无锁同步中起着关键作用。CAS 操作包含三个参数：一个内存地址 V、一个期望值 A 和一个新值 B。当执行 CAS 操作时，如果当前内存地址 V 中存储的值等于期望值 A，则将新值 B 写入该内存地址，并返回 true；否则，不做任何修改，并返回 false。在无锁队列中，可以利用 CAS 操作来确保对 Head 或 Tail 指针的读写操作是原子性的，从而避免多线程同时写入或读取时出现的指针混乱问题。例如，在入队操作中，可以使用 CAS 来确保在更新 Tail 指针时，不会被其他线程干扰。如果当前 Tail 指针指向的节点的_next指针与期望值不一致，说明有其他线程进行了写入操作，此时可以重新尝试 CAS 操作，直到成功为止。这样就可以实现无锁队列的安全写入和读取操作。

## 三、无锁队列方案

### 3.1boost方案

boost提供了三种无锁方案，分别适用不同使用场景。

- boost::lockfree::queue是支持多个生产者和多个消费者线程的无锁队列。
- boost::lockfree::stack是支持多个生产者和多个消费者线程的无锁栈。
- boost::lockfree::spsc_queue是仅支持单个生产者和单个消费者线程的无锁队列，比boost::lockfree::queue性能更好。

Boost无锁数据结构的API通过轻量级原子锁实现lock-free，不是真正意义的无锁。

Boost提供的queue可以设置初始容量，添加新元素时如果容量不够，则总容量自动增长；但对于无锁数据结构，添加新元素时如果容量不够，总容量不会自动增长。

### 3.2并发队列

ConcurrentQueue采用了无锁算法来实现并发操作。它基于CAS（Compare-and-Swap）原子操作和其他底层同步原语来保证线程安全性。具体来说，它使用自旋锁和原子指令来确保对队列的修改是原子的，并且在多个线程之间共享数据时提供正确性保证。

```
ConcurrentQueue是基于C++实现的工业级无锁队列方案。
GitHub：https://github.com/cameron314/concurrentqueue
ReaderWriterQueue是基于C++实现的单生产者单消费者场景的无锁队列方案。
GitHub：https://github.com/cameron314/readerwriterqueue
```

ConcurrentQueue具有以下特点：

- 线程安全：多个线程可以同时对队列进行操作而无需额外加锁。
- 无阻塞：入队和出队操作通常是非阻塞的，并且具有较低的开销。
- 先进先出（FIFO）顺序：元素按照插入顺序排列，在出队时会返回最早入队的元素。

使用ConcurrentQueue可以方便地处理多个线程之间共享数据，并减少由于加锁引起的性能开销。但需要注意，虽然ConcurrentQueue提供了高效、线程安全的并发操作，但在某些特定情况下可能不适合所有应用场景，因此在选择数据结构时需要根据具体需求进行评估。

### 3.3Disruptor

Disruptor是一种高性能的并发编程框架，用于实现无锁（lock-free）的并发数据结构。它最初由LMAX Exchange开发，并成为了其核心交易引擎的关键组件。

Disruptor旨在解决在高度多线程环境下的数据共享和通信问题。它基于环形缓冲区（Ring Buffer）和事件驱动模型，通过优化内存访问和线程调度，提供了非常高效的消息传递机制。

```
Disruptor是英国外汇交易公司LMAX基于JAVA开发的一个高性能队列。

GitHub：https://github.com/LMAX-Exchange/disruptor
```

主要特点如下：

- 无锁设计：Disruptor使用CAS（Compare-and-Swap）等无锁算法来避免使用传统锁带来的竞争和阻塞。
- 高吞吐量：Disruptor利用环形缓冲区和预分配内存等技术，在保证正确性前提下追求尽可能高的处理速度。
- 低延迟：由于无锁设计和紧凑的内存布局，Disruptor能够实现非常低的消息处理延迟。
- 线程间协调：Disruptor提供了灵活而强大的事件发布、消费者等待及触发机制，可用于实现复杂的线程间通信模式。

使用Disruptor可以有效地解决生产者-消费者模型中数据传递过程中的性能瓶颈，特别适用于高并发、低延迟的应用场景，例如金融交易系统、消息队列等。然而，由于Disruptor对编程模型和理解要求较高，使用时需要仔细考虑，并根据具体需求评估是否适合。

## 四、无锁队列的实现方式

## 4.1环形缓冲区

RingBuffer是生产者和消费者模型中常用的数据结构，生产者将数据追加到数组尾端，当达到数组的尾部时，生产者绕回到数组的头部；消费者从数组头端取走数据，当到达数组的尾部时，消费者绕回到数组头部。

如果只有一个生产者和一个消费者，环形缓冲区可以无锁访问，环形缓冲区的写入index只允许生产者访问并修改，只要生产者在更新index前将新的值保存到缓冲区中，则消费者将始终看到一致的数据结构；读取index也只允许消费者访问并修改，消费者只要在取走数据后更新读index，则生产者将始终看到一致的数据结构。

- 空队列时，front与rear相等；当有元素进队，则rear后移；有元素出队，则front后移。
- 空队列时，rear等于front；满队列时，队列尾部空一个位置，因此判断循环队列满时使用(rear-front+maxn)%maxn。

入队操作：

```
data[rear] = x;
rear = (rear+1)%maxn;
```

出队操作：

```
x = data[front];
front = (front+1)%maxn;
```

**单生产者单消费者**

对于单生产者和单消费者场景，由于read_index和write_index都只会有一个线程写，因此不需要加锁也不需要原子操作，直接修改即可，但读写数据时需要考虑遇到数组尾部的情况。

**线程对write_index和read_index的读写操作如下：**

- （1）写操作。先判断队列时否为满，如果队列未满，则先写数据，写完数据后再修改write_index。
- （2）读操作。先判断队列是否为空，如果队列不为空，则先读数据，读完再修改read_index。

多生产者单消费者：多生产者和单消费者场景中，由于多个生产者都会修改write_index，所以在不加锁的情况下必须使用原子操作。

## 4.2RingBuffer实现

RingBuffer.hpp文件：

```
#pragma once

template <class T>
class RingBuffer
{
public:
    RingBuffer(unsigned size): m_size(size), m_front(0), m_rear(0)
    {
        m_data = new T[size];
    }

    ~RingBuffer()
    {
        delete [] m_data;
        m_data = NULL;
    }

    inline bool isEmpty() const
    {
        return m_front == m_rear;
    }

    inline bool isFull() const
    {
        return m_front == (m_rear + 1) % m_size;
    }

    bool push(const T& value)
    {
        if(isFull())
        {
            return false;
        }
        m_data[m_rear] = value;
        m_rear = (m_rear + 1) % m_size;
        return true;
    }

    bool push(const T* value)
    {
        if(isFull())
        {
            return false;
        }
        m_data[m_rear] = *value;
        m_rear = (m_rear + 1) % m_size;
        return true;
    }
```

```
    inline bool pop(T& value)
    {
        if(isEmpty())
        {
            return false;
        }
        value = m_data[m_front];
        m_front = (m_front + 1) % m_size;
        return true;
    }

    inline unsigned int front()const
    {
        return m_front;
    }

    inline unsigned int rear()const
    {
        return m_rear;
    }

    inline unsigned int size()const
    {
        return m_size;
    }
private:
    unsigned int m_size;// 队列长度
    int m_front;// 队列头部索引
    int m_rear;// 队列尾部索引
    T* m_data;// 数据缓冲区
};
```

RingBufferTest.cpp测试代码：

```
#include <stdio.h>
#include <thread>
#include <unistd.h>
#include <sys/time.h>
#include "RingBuffer.hpp"


class Test
{
public:
   Test(int id = 0, int value = 0)
   {
        this->id = id;
```

```
        this->value = value;
        sprintf(data, "id = %d, value = %d\n", this->id, this->value);
    }

    void display()
    {
       printf("%s", data);
    }
private:
    int id;
    int value;
    char data[128];
};

double getdetlatimeofday(struct timeval *begin, struct timeval *end)
{
    return (end->tv_sec + end->tv_usec * 1.0 / 1000000) -
           (begin->tv_sec + begin->tv_usec * 1.0 / 1000000);
}

RingBuffer<Test> queue(1 << 12);2u000

#define N (10 * (1 << 20))

void produce()
{
    struct timeval begin, end;
    gettimeofday(&begin, NULL);
    unsigned int i = 0;
    while(i < N)
    {
        if(queue.push(Test(i % 1024, i)))
        {
           i++;
        }
    }

    gettimeofday(&end, NULL);
    double tm = getdetlatimeofday(&begin, &end);
    printf("producer tid=%lu %f MB/s %f msg/s elapsed= %f size= %u\n", pth
}

void consume()
{
    sleep(1);
    Test test;
    struct timeval begin, end;
    gettimeofday(&begin, NULL);
    unsigned int i = 0;
```

```
    while(i < N)
    {
        if(queue.pop(test))
        {
           // test.display();
           i++;
        }
    }
    gettimeofday(&end, NULL);
    double tm = getdetlatimeofday(&begin, &end);
    printf("consumer tid=%lu %f MB/s %f msg/s elapsed= %f, size=%u \n", pt
}

int main(int argc, char const *argv[])
{
    std::thread producer1(produce);
    std::thread consumer(consume);
    producer1.join();
    consumer.join();
    return 0;
}
```

编译:

```
g++ --std=c++11  RingBufferTest.cpp -o test -pthread
```

单生产者单消费者场景下，消息吞吐量为350万条/秒左右。

### 4.3LockFreeQueue实现

LockFreeQueue.hpp:

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include <fcntl.h>
#include <stdbool.h>
#include <sys/stat.h>
#include <sys/types.h>
#include <sys/time.h>
#include <sys/mman.h>

#define SHM_NAME_LEN 128
#define MIN(a, b) ((a) > (b) ? (b) : (a))
#define IS_POT(x) ((x) && !((x) & ((x)-1)))
```

```
#define MEMORY_BARRIER __sync_synchronize()

template <class T>
class LockFreeQueue
{
protected:
    typedef struct
    {
        int m_lock;
        inline void spinlock_init()
        {
            m_lock = 0;
        }

        inline void spinlock_lock()
        {
            while(!__sync_bool_compare_and_swap(&m_lock, 0, 1)) {}
        }

        inline void spinlock_unlock()
        {
            __sync_lock_release(&m_lock);
        }
    } spinlock_t;

public:
    // size:队列大小
    // name:共享内存key的路径名称，默认为NULL，使用数组作为底层缓冲区。
    LockFreeQueue(unsigned int size, const char* name = NULL)
    {
        memset(shm_name, 0, sizeof(shm_name));
        createQueue(name, size);
    }

    ~LockFreeQueue()
    {
        if(shm_name[0] == 0)
        {
            delete [] m_buffer;
            m_buffer = NULL;
        }
        else
        {
            if (munmap(m_buffer, m_size * sizeof(T)) == -1) {
                perror("munmap");
            }
            if (shm_unlink(shm_name) == -1) {
                perror("shm_unlink");
            }
```

```
        }
    }

    bool isFull()const
    {
#ifdef USE_POT
        return m_head == (m_tail + 1) & (m_size - 1);
#else
        return m_head == (m_tail + 1) % m_size;
#endif
    }

    bool isEmpty()const
    {
        return m_head == m_tail;
    }

    unsigned int front()const
    {
        return m_head;
    }

    unsigned int tail()const
    {
        return m_tail;
    }

    bool push(const T& value)
    {
#ifdef USE_LOCK
        m_spinLock.spinlock_lock();
#endif
        if(isFull())
        {
#ifdef USE_LOCK
            m_spinLock.spinlock_unlock();
#endif
            return false;
        }
        memcpy(m_buffer + m_tail, &value, sizeof(T));
#ifdef USE_MB
        MEMORY_BARRIER;
#endif

#ifdef USE_POT
        m_tail = (m_tail + 1) & (m_size - 1);
#else
        m_tail = (m_tail + 1) % m_size;
#endif
```

```
#ifdef USE_LOCK
        m_spinLock.spinlock_unlock();
#endif
        return true;
    }

    bool pop(T& value)
    {
#ifdef USE_LOCK
        m_spinLock.spinlock_lock();
#endif
        if (isEmpty())
        {
#ifdef USE_LOCK
            m_spinLock.spinlock_unlock();
#endif
            return false;
        }
        memcpy(&value, m_buffer + m_head, sizeof(T));
#ifdef USE_MB
        MEMORY_BARRIER;
#endif

#ifdef USE_POT
        m_head = (m_head + 1) & (m_size - 1);
#else
        m_head = (m_head + 1) % m_size;
#endif

#ifdef USE_LOCK
        m_spinLock.spinlock_unlock();
#endif
        return true;
    }

protected:
    virtual void createQueue(const char* name, unsigned int size)
    {
#ifdef USE_POT
        if (!IS_POT(size))
        {
            size = roundup_pow_of_two(size);
        }
#endif
        m_size = size;
        m_head = m_tail = 0;
        if(name == NULL)
        {
```

```
            m_buffer = new T[m_size];
        }
        else
        {
            int shm_fd = shm_open(name, O_CREAT | O_RDWR, 0666);
            if (shm_fd < 0)
            {
                perror("shm_open");
            }

            if (ftruncate(shm_fd, m_size * sizeof(T)) < 0)
            {
                perror("ftruncate");
                close(shm_fd);
            }

            void *addr = mmap(0, m_size * sizeof(T), PROT_READ | PROT_WRIT
            if (addr == MAP_FAILED)
            {
                perror("mmap");
                close(shm_fd);
            }
            if (close(shm_fd) == -1)
            {
                perror("close");
                exit(1);
            }

            m_buffer = static_cast<T*>(addr);
            memcpy(shm_name, name, SHM_NAME_LEN - 1);
        }
#ifdef USE_LOCK
    spinlock_init(m_lock);
#endif
    }
    inline unsigned int roundup_pow_of_two(size_t size)
    {
        size |= size >> 1;
        size |= size >> 2;
        size |= size >> 4;
        size |= size >> 8;
        size |= size >> 16;
        size |= size >> 32;
        return size + 1;
    }
protected:
    char shm_name[SHM_NAME_LEN];
    volatile unsigned int m_head;
    volatile unsigned int m_tail;
```

```
    unsigned int m_size;
#ifdef USE_LOCK
    spinlock_t m_spinLock;
#endif
    T* m_buffer;
};


#define USE_LOCK
```

开启spinlock锁，多生产者多消费者场景

```
#define USE_MB
```

开启Memory Barrier

```
#define USE_POT
```

开启队列大小的2的幂对齐

LockFreeQueueTest.cpp测试文件：

```
#include "LockFreeQueue.hpp"
#include <thread>

//#define USE_LOCK

class Test
{
public:
   Test(int id = 0, int value = 0)
   {
        this->id = id;
        this->value = value;
        sprintf(data, "id = %d, value = %d\n", this->id, this->value);
   }

   void display()
   {
      printf("%s", data);
   }
private:
   int id;
   int value;
   char data[128];
};

double getdetlatimeofday(struct timeval *begin, struct timeval *end)
```

```
{
    return (end->tv_sec + end->tv_usec * 1.0 / 1000000) -
           (begin->tv_sec + begin->tv_usec * 1.0 / 1000000);
}

LockFreeQueue<Test> queue(1 << 10, "/shm");

#define N ((1 << 20))

void produce()
{
    struct timeval begin, end;
    gettimeofday(&begin, NULL);
    unsigned int i = 0;
    while(i < N)
    {
        if(queue.push(Test(i >> 10, i)))
            i++;
    }
    gettimeofday(&end, NULL);
    double tm = getdetlatimeofday(&begin, &end);
    printf("producer tid=%lu %f MB/s %f msg/s elapsed= %f size= %u\n", pth
}

void consume()
{
    Test test;
    struct timeval begin, end;
    gettimeofday(&begin, NULL);
    unsigned int i = 0;
    while(i < N)
    {
        if(queue.pop(test))
        {
           //test.display();
           i++;
        }
    }
    gettimeofday(&end, NULL);
    double tm = getdetlatimeofday(&begin, &end);
    printf("consumer tid=%lu %f MB/s %f msg/s elapsed= %f size= %u\n", pth
}

int main(int argc, char const *argv[])
{
    std::thread producer1(produce);
    //std::thread producer2(produce);
    std::thread consumer(consume);
    producer1.join();
```

```
    //producer2.join();
    consumer.join();

    return 0;
}
```

多线程场景下，需要定义USE_LOCK宏，开启锁保护。

编译：

```
g++ --std=c++11 -O3 LockFreeQueueTest.cpp -o test -lrt -pthread
```

## 4.4场景分析（不同场景的实现）

单生产者单消费者队列（SPSC 队列）：在这种队列中，只有一个生产者线程和一个消费者线程操作该队列。由于只有一个线程操作队列，因此不需要考虑线程同步和数据竞争的问题，可以实现非常高效的数据访问。

例如，在某些特定的任务处理场景中，一个线程负责生成任务，另一个线程负责处理任务，此时使用 SPSC 队列可以避免复杂的同步机制，提高处理效率。

多生产者多消费者队列（MPMC 队列）：在这种队列中，有多个生产者线程和多个消费者线程操作该队列。由于存在多个线程同时操作队列，因此必须考虑线程同步和数据竞争的问题，需要使用一些同步机制来保证数据的正确性。

常见的实现方式是使用原子操作和 CAS 等技术。例如，在一个高并发的服务器程序中，多个客户端请求可以看作是生产者，服务器的多个处理线程可以看作是消费者，使用 MPMC 队列可以有效地管理这些请求和处理任务。

单生产者多消费者队列（SPMC 队列）：在这种队列中，生产者线程向队列中写入数据，多个消费者线程从队列中读取数据。这种队列的实现可以使用原子操作或者互斥锁来实现线程同步。

比如在一个视频处理系统中，一个视频采集线程作为生产者，多个视频编码线程作为消费者，使用 SPMC 队列可以实现高效的数据传输和处理。

多生产者单消费者队列（MPSC 队列）：在这种队列中，多个生产者线程向队列中写入数据，一个消费者线程从队列中读取数据。这种队列的实现也可以使用原子操作或者互斥锁来实现线程同步。

例如在一个日志收集系统中，多个日志生成线程作为生产者，一个日志分析线程作为消费者，使用 MPSC 队列可以方便地管理日志数据。

## 4.5常见队列形式

### ⑴链式队列（Lock-free Linked Queue）

- 链式队列是一种基于链表实现的队列，每个节点包含一个数据元素和一个指向下一个节点的指针。
- 特点：可以动态地分配和释放内存，适用于数据量不确定或者数据大小不固定的情况。在多线程环境下，需要使用无锁算法来避免锁的性能损失。
- 例如，在一个实时数据采集系统中，数据量可能随时变化，使用链式队列可以灵活地适应这种变化。

**⑵数组队列（Lock-free Array Queue）**

- 数组队列是一种基于数组实现的队列，它可以提高数据的读写效率，适用于数据量比较大且大小固定的情况。
- 实现比较简单，可以使用一个指针来记录队尾位置，一个指针来记录队头位置。在多线程环境下，需要使用无锁算法来避免锁的性能损失。
- 比如在一个图像处理系统中，图像数据的大小相对固定，使用数组队列可以提高数据处理效率。

**⑶环形队列**

- 实现环形队列的方式可以基于数组或者基于链表。
- 优点：可以有效地利用内存空间，避免了数据的移动和浪费。在多线程环境下，也可以使用无锁算法来实现高效的数据访问。
- 例如，在一个网络数据包处理系统中，环形队列可以快速地存储和读取数据包，提高系统的性能。

## 五、无锁队列的性能优势

### 5.1高效性

无锁队列之所以能够避免锁竞争和开销，从而提高性能，主要有以下几个原因。首先，锁的使用会带来上下文切换和线程调度延迟。当一个线程获取锁时，如果其他线程也在竞争这个锁，那么这些线程可能会被阻塞，等待锁的释放。而上下文切换和线程调度需要消耗一定的时间和系统资源，这会降低程序的执行效率。无锁队列通过使用原子操作和 CAS 等技术，避免了锁的使用，从而减少了上下文切换和线程调度的次数，提高了程序的性能。

其次，无锁队列可以更好地利用处理器的缓存。在多线程环境下，锁的使用可能会导致缓存一致性问题，因为多个线程可能会同时访问和修改共享数据。为了保证数据的一致性，处理器需要进行缓存同步操作，这会降低缓存的命中率，增加内存访问的延迟。无锁队列通过使用原子操作和 CAS 等技术，可以避免缓存一致性问题，提高缓存的命中率，从而减少内存访问的延迟，提高程序的性能。

据统计，在某些高并发的场景下，无锁队列的性能可以比有锁队列提高数倍甚至数十倍。

### 5.2线程安全

无锁队列在多线程环境下具有很高的安全性。这是因为无锁队列通过使用原子操作和 CAS 等技术，确保了对队列的操作是原子性的。原子操作是不可分割的操作，它要么全部执行成功，要么全部执行失败，不会出现部分执行成功的情况。CAS 操作可以确保在对队列进行操作时，不会被其他线程干扰。如果当前内存地址中存储的值与期望值不一致，说明有其他线程进行了写入操作，此时可以重新尝试 CAS 操作，直到成功为止。

此外，无锁队列还可以避免死锁问题。在有锁队列中，如果多个线程同时获取锁，并且在获取锁的顺序上出现问题，就可能会导致死锁。而无锁队列不需要使用锁，自然也就避免了死锁问题。

### 5.3可扩展性和低延迟

无锁队列具有很好的可扩展性，可以在多个处理器上并行运行。在多处理器系统中，无锁队列可以更好地利用多个处理器的资源，提高队列的吞吐量。这是因为无锁队列通过使用原子操作和 CAS 等技术，避免了锁的竞争，从而可以让多个线程在不同的处理器上同时对队列进行操作。

无锁队列还具有低延迟的特点。在实时系统中，低延迟是非常重要的。无锁队列可以实现非阻塞式的数据访问，从而降低队列的延迟。这是因为无锁队列通过使用原子操作和 CAS 等技术，可以在不阻塞其他线程的情况下，对队列进行操作。如果当前操作无法成功，可以立即返回，而不会等待其他线程释放锁。这样可以减少线程的等待时间，提高系统的响应速度。

例如，在一个高并发的网络服务器中，无锁队列可以快速地处理大量的网络请求，提高服务器的响应速度和吞吐量。

## 六、Kfifo内核队列

计算机科学家已经证明，当只有一个读线程和一个写线程并发操作时，不需要任何额外的锁，就可以确保是线程安全的，也即kfifo使用了无锁编程技术，以提高kernel的并发。

Linux kernel里面从来就不缺少简洁，优雅和高效的代码，只是我们缺少发现和品味的眼光。在Linux kernel里面，简洁并不表示代码使用神出鬼没的超然技巧，相反，它使用的不过是大家非常熟悉的基础数据结构，但是kernel开发者能从基础的数据结构中，提炼出优美的特性。

kfifo就是这样的一类优美代码，它十分简洁，绝无多余的一行代码，却非常高效。

**关于kfifo信息如下**：

```
本文分析的原代码版本： 2.6.24.4
kfifo的定义文件： kernel/kfifo.c
kfifo的头文件： include/linux/kfifo.h
```

kfifo是Linux内核的一个FIFO数据结构，采用环形循环队列的数据结构来实现，提供一个无边界的字节流服务，并且使用并行无锁编程技术，即单生产者单消费者场景下两个线程可以并发操作，不需要任何加锁行为就可以保证kfifo线程安全。

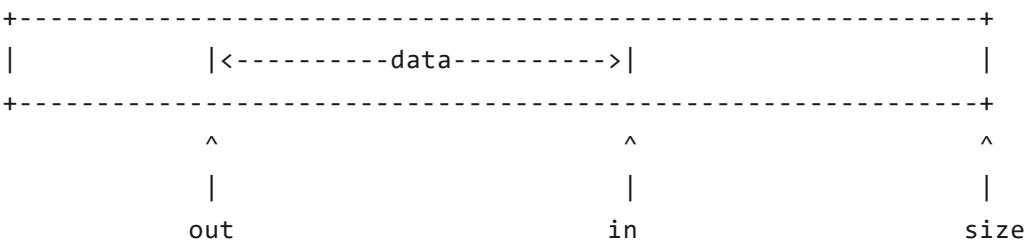
kfifo代码既然肩负着这么多特性，那我们先一敞它的代码：

```
struct kfifo {
    unsigned char *buffer;    /* the buffer holding the data */
    unsigned int size;    /* the size of the allocated buffer */
    unsigned int in;    /* data is added at offset (in % size) */
    unsigned int out;    /* data is extracted from off. (out % size) */
    spinlock_t *lock;    /* protects concurrent modifications */
};
```

这是kfifo的数据结构，kfifo主要提供了两个操作，__kfifo_put(入队操作)和__kfifo_get(出队操作)。 它的各个数据成员如下：

- buffer: 用于存放数据的缓存
- size: buffer空间的大小，在初化时，将它向上扩展成2的幂（如5，向上扩展 与它最接近的值且是2的n次方的值是2^3，即8)
- lock: 如果使用不能保证任何时间最多只有一个读线程和写线程，需要使用该lock实施同步。
- in, out: 和buffer一起构成一个循环队列。 in指向buffer中队头，而且out指向buffer中的队尾

它的结构如示图如下：




当然，内核开发者使用了一种更好的技术处理了in, out和buffer的关系，我们将在下面进行详细分析。

## 6.1kfifo功能描述

kfifo提供如下对外功能规格

- 只支持一个读者和一个读者并发操作
- 无阻塞的读写操作，如果空间不够，则返回实际访问空间

**(1)kfifo_alloc 分配kfifo内存和初始化工作**

```
struct kfifo *kfifo_alloc(unsigned int size, gfp_t gfp_mask, spinlock_t *l
{
    unsigned char *buffer;
    struct kfifo *ret;
```

```
    /*
     * round up to the next power of 2, since our 'let the indices
     * wrap' tachnique works only in this case.
     */
    if (size & (size - 1)) {
        BUG_ON(size > 0x80000000);
        size = roundup_pow_of_two(size);
    }

    buffer = kmalloc(size, gfp_mask);
    if (!buffer)
        return ERR_PTR(-ENOMEM);

    ret = kfifo_init(buffer, size, gfp_mask, lock);

    if (IS_ERR(ret))
        kfree(buffer);

    return ret;
}
```

这里值得一提的是，kfifo->size的值总是在调用者传进来的size参数的基础上向2的幂扩展（roundup_pow_of_two，我自己的实现在文章末尾），这是内核一贯的做法。这样的好处不言而喻——对kfifo->size取模运算可以转化为与运算，如下：

```
kfifo->in % kfifo->size 可以转化为 kfifo->in & (kfifo->size – 1)
```

在kfifo_alloc函数中，使用size & (size – 1)来判断size 是否为2幂，如果条件为真，则表示size不是2的幂，然后调用roundup_pow_of_two将之向上扩展为2的幂。

这都是常用的技巧，只不过大家没有将它们结合起来使用而已，下面要分析的__kfifo_put和__kfifo_get则是将kfifo->size的特点发挥到了极致。

**(2)__kfifo_put和__kfifo_get巧妙的入队和出队**

__kfifo_put是入队操作，它先将数据放入buffer里面，最后才修改in参数；__kfifo_get是出队操作，它先将数据从buffer中移走，最后才修改out。（确保即使in和out修改失败，也可以再来一遍）

你会发现in和out两者各司其职。下面是__kfifo_put和__kfifo_get的代码

```
unsigned int __kfifo_put(struct kfifo *fifo,
             unsigned char *buffer, unsigned int len)
{
    unsigned int l;
```

```
    len = min(len, fifo->size - fifo->in + fifo->out);
    /*
     * Ensure that we sample the fifo->out index -before- we
     * start putting bytes into the kfifo.
     */
    smp_mb();

    /* first put the data starting from fifo->in to buffer end */
    l = min(len, fifo->size - (fifo->in & (fifo->size - 1)));
    memcpy(fifo->buffer + (fifo->in & (fifo->size - 1)), buffer, l);

    /* then put the rest (if any) at the beginning of the buffer */
    memcpy(fifo->buffer, buffer + l, len - l);

    /*
     * Ensure that we add the bytes to the kfifo -before-
     * we update the fifo->in index.
     */

    smp_wmb();

    fifo->in += len;

    return len;
}
```

奇怪吗？代码完全是线性结构，没有任何if-else分支来判断是否有足够的空间存放数据。内核在这里的代码非常简洁，没有一行多余的代码。

```
l = min(len, fifo->size - (fifo->in & (fifo->size - 1)));
```

这个表达式计算当前写入的空间，换成人可理解的语言就是：

```
l = kfifo可写空间和预期写入空间的最小值
```

**(3)使用min宏来代if-else分支**

__kfifo_get也应用了同样技巧，代码如下：

```
unsigned int __kfifo_get(struct kfifo *fifo,
             unsigned char *buffer, unsigned int len)
{
    unsigned int l;

    len = min(len, fifo->in - fifo->out);
    /*
```

```
     * Ensure that we sample the fifo->in index -before- we
     * start removing bytes from the kfifo.
     */
    smp_rmb();

    /* first get the data from fifo->out until the end of the buffer */
    l = min(len, fifo->size - (fifo->out & (fifo->size - 1)));
    memcpy(buffer, fifo->buffer + (fifo->out & (fifo->size - 1)), l);

    /* then get the rest (if any) from the beginning of the buffer */
    memcpy(buffer + l, fifo->buffer, len - l);

    /*
     * Ensure that we remove the bytes from the kfifo -before-
     * we update the fifo->out index.
     */

    smp_mb();

    fifo->out += len;

    return len;
}
```

认真读两遍吧，我也读了多次，每次总是有新发现，因为in, out和size的关系太巧妙了，竟然能利用上unsigned int回绕的特性。

原来，kfifo每次入队或出队，kfifo->in或kfifo->out只是简单地kfifo->in/kfifo->out += len，并没有对kfifo->size 进行取模运算。因此kfifo->in和kfifo->out总是一直增大，直到unsigned in最大值时，又会绕回到0这一起始端。但始终满足：

```
kfifo->in - kfifo->out <= kfifo->size
```

即使kfifo->in回绕到了0的那一端，这个性质仍然是保持的。

对于给定的kfifo:

```
数据空间长度为：kfifo->in - kfifo->out
```

而剩余空间（可写入空间）长度为：kfifo->size - (kfifo->in - kfifo->out)

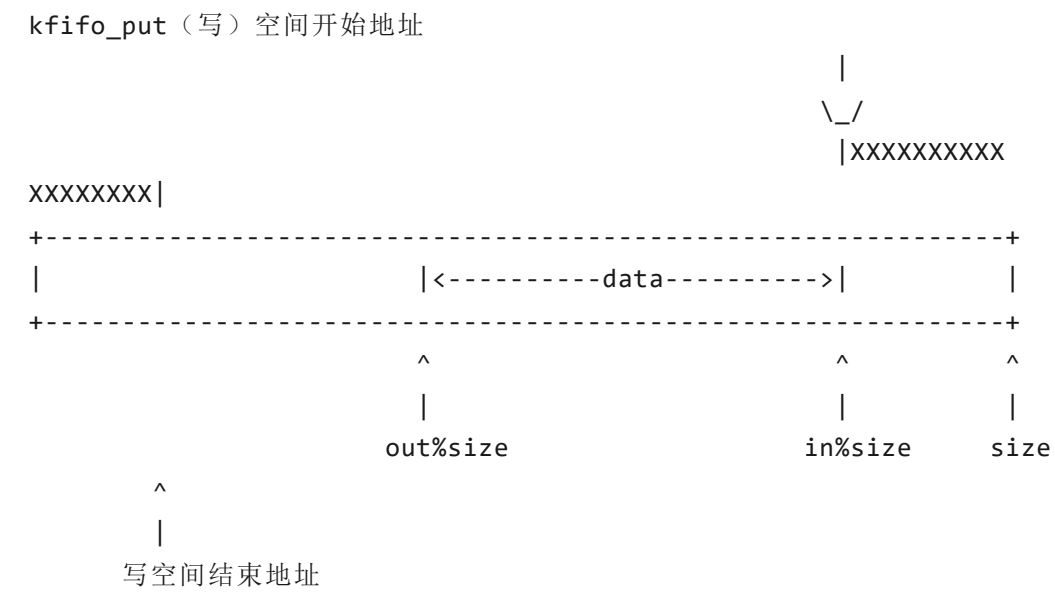
尽管kfifo->in和kfofo->out一直超过kfifo->size进行增长，但它对应在kfifo->buffer空间的下标却是如下：

```
kfifo->in % kfifo->size (i.e. kfifo->in & (kfifo->size - 1))
kfifo->out % kfifo->size (i.e. kfifo->out & (kfifo->size - 1))
```

往kfifo里面写一块数据时，数据空间、写入空间和kfifo->size的关系如果满足：

```
kfifo->in % size + len > size
```

那就要做写拆分了，见下图：




第一块当然是: [kfifo->in % kfifo->size, kfifo->size]
第二块当然是：[0, len - (kfifo->size - kfifo->in % kfifo->size)]

下面是代码，细细体味吧：

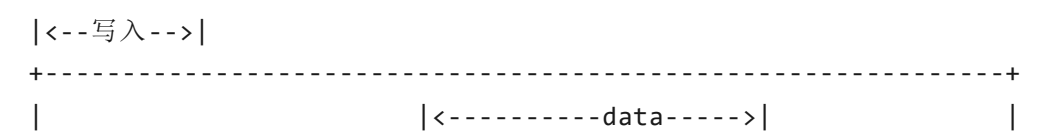
```
/* first put the data starting from fifo->in to buffer end */
l = min(len, fifo->size - (fifo->in & (fifo->size - 1)));
memcpy(fifo->buffer + (fifo->in & (fifo->size - 1)), buffer, l);

/* then put the rest (if any) at the beginning of the buffer */
memcpy(fifo->buffer, buffer + l, len - l);
```

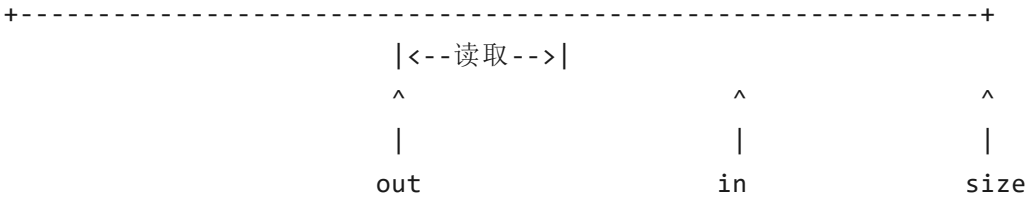
对于kfifo_get过程，也是类似的，请各位自行分析。

**(4)kfifo_get和kfifo_put无锁并发操作**

计算机科学家已经证明，当只有一个读经程和一个写线程并发操作时，不需要任何额外的锁，就可以确保是线程安全的，也即kfifo使用了无锁编程技术，以提高kernel的并发。

kfifo使用in和out两个指针来描述写入和读取游标，对于写入操作，只更新in指针，而读取操作，只更新out指针，可谓井水不犯河水，示意图如下：






为了避免读者看到写者预计写入，但实际没有写入数据的空间，写者必须保证以下的写入顺序：

```
往[kfifo->in, kfifo->in + len]空间写入数据
更新kfifo->in指针为 kfifo->in + len
```

在操作1完成时，读者是还没有看到写入的信息的，因为kfifo->in没有变化，认为读者还没有开始写操作，只有更新kfifo->in之后，读者才能看到。

那么如何保证1必须在2之前完成，秘密就是使用内存屏障：smp_mb()，smp_rmb(), smp_wmb()，来保证对方观察到的内存操作顺序。

## 6.2kfifo内核队列实现

kfifo数据结构定义如下：

```
struct kfifo
{
    unsigned char *buffer;
    unsigned int size;
    unsigned int in;
    unsigned int out;
    spinlock_t *lock;
};

// 创建队列
struct kfifo *kfifo_init(unsigned char *buffer, unsigned int size, gfp_t g
{
    struct kfifo *fifo;
    // 判断是否为2的幂
    BUG_ON(!is_power_of_2(size));
    fifo = kmalloc(sizeof(struct kfifo), gfp_mask);
    if (!fifo)
        return ERR_PTR(-ENOMEM);
    fifo->buffer = buffer;
    fifo->size = size;
    fifo->in = fifo->out = 0;
    fifo->lock = lock;

    return fifo;
}
```

```
// 分配空间
struct kfifo *kfifo_alloc(unsigned int size, gfp_t gfp_mask, spinlock_t *l
{
    unsigned char *buffer;
    struct kfifo *ret;
    // 判断是否为2的幂
    if (!is_power_of_2(size))
    {
        BUG_ON(size > 0x80000000);
        // 向上扩展成2的幂
        size = roundup_pow_of_two(size);
    }

    buffer = kmalloc(size, gfp_mask);
    if (!buffer)
        return ERR_PTR(-ENOMEM);
    ret = kfifo_init(buffer, size, gfp_mask, lock);

    if (IS_ERR(ret))
        kfree(buffer);
    return ret;
}

void kfifo_free(struct kfifo *fifo)
{
    kfree(fifo->buffer);
    kfree(fifo);
}

// 入队操作
static inline unsigned int kfifo_put(struct kfifo *fifo, const unsigned ch
{
    unsigned long flags;
    unsigned int ret;
    spin_lock_irqsave(fifo->lock, flags);
    ret = __kfifo_put(fifo, buffer, len);
    spin_unlock_irqrestore(fifo->lock, flags);
    return ret;
}

// 出队操作
static inline unsigned int kfifo_get(struct kfifo *fifo, unsigned char *bu
{
    unsigned long flags;
    unsigned int ret;
    spin_lock_irqsave(fifo->lock, flags);
    ret = __kfifo_get(fifo, buffer, len);
    //当fifo->in == fifo->out时，buufer为空
```

```
    if (fifo->in == fifo->out)
        fifo->in = fifo->out = 0;

    spin_unlock_irqrestore(fifo->lock, flags);
    return ret;
}

// 入队操作
unsigned int __kfifo_put(struct kfifo *fifo, const unsigned char *buffer,
{
    unsigned int l;
    //buffer中空的长度
    len = min(len, fifo->size - fifo->in + fifo->out);
    // 内存屏障：smp_mb()，smp_rmb(), smp_wmb()来保证对方观察到的内存操作顺序
    smp_mb();
    // 将数据追加到队列尾部
    l = min(len, fifo->size - (fifo->in & (fifo->size - 1)));
    memcpy(fifo->buffer + (fifo->in & (fifo->size - 1)), buffer, l);
    memcpy(fifo->buffer, buffer + l, len - l);

    smp_wmb();
    //每次累加，到达最大值后溢出，自动转为0
    fifo->in += len;
    return len;
}

// 出队操作
unsigned int __kfifo_get(struct kfifo *fifo, unsigned char *buffer, unsign
{
    unsigned int l;
    //有数据的缓冲区的长度
    len = min(len, fifo->in - fifo->out);
    smp_rmb();
    l = min(len, fifo->size - (fifo->out & (fifo->size - 1)));
    memcpy(buffer, fifo->buffer + (fifo->out & (fifo->size - 1)), l);
    memcpy(buffer + l, fifo->buffer, len - l);
    smp_mb();
    fifo->out += len; //每次累加，到达最大值后溢出，自动转为0

    return len;
}

static inline void __kfifo_reset(struct kfifo *fifo)
{
    fifo->in = fifo->out = 0;
}

static inline void kfifo_reset(struct kfifo *fifo)
{
```

```
    unsigned long flags;
    spin_lock_irqsave(fifo->lock, flags);
    __kfifo_reset(fifo);
    spin_unlock_irqrestore(fifo->lock, flags);
}

static inline unsigned int __kfifo_len(struct kfifo *fifo)
{
    return fifo->in - fifo->out;
}

static inline unsigned int kfifo_len(struct kfifo *fifo)
{
    unsigned long flags;
    unsigned int ret;
    spin_lock_irqsave(fifo->lock, flags);
    ret = __kfifo_len(fifo);
    spin_unlock_irqrestore(fifo->lock, flags);
    return ret;
}
```

### 6.3kfifo设计要点

（1）保证buffer size为2的幂

kfifo->size值在调用者传递参数size的基础上向2的幂扩展，目的是使kfifo->size取模运算可以转化为位与运算（提高运行效率）。kfifo->in % kfifo->size转化为 kfifo->in & (kfifo->size – 1)

保证size是2的幂可以通过位运算的方式求余，在频繁操作队列的情况下可以大大提高效率。

（2）使用spin_lock_irqsave与spin_unlock_irqrestore 实现同步。

Linux内核中有spin_lock、spin_lock_irq和spin_lock_irqsave保证同步。

```
static inline void __raw_spin_lock(raw_spinlock_t *lock)
{
    preempt_disable();
    spin_acquire(&lock->dep_map, 0, 0, _RET_IP_);
    LOCK_CONTENDED(lock, do_raw_spin_trylock, do_raw_spin_lock);
}

static inline void __raw_spin_lock_irq(raw_spinlock_t *lock)
{
    local_irq_disable();
    preempt_disable();
    spin_acquire(&lock->dep_map, 0, 0, _RET_IP_);
```

```
    LOCK_CONTENDED(lock, do_raw_spin_trylock, do_raw_spin_lock);
}
```

spin_lock比spin_lock_irq速度快，但并不是线程安全的。spin_lock_irq增加调用local_irq_disable函数，即禁止本地中断，是线程安全的，既禁止本地中断，又禁止内核抢占。

spin_lock_irqsave是基于spin_lock_irq实现的一个辅助接口，在进入和离开临界区后，不会改变中断的开启、关闭状态。

如果自旋锁在中断处理函数中被用到，在获取自旋锁前需要关闭本地中断，spin_lock_irqsave实现如下：

- A、保存本地中断状态；
- B、关闭本地中断；
- C、获取自旋锁。

解锁时通过 spin_unlock_irqrestore完成释放锁、恢复本地中断到原来状态等工作。

（3）线性代码结构

代码中没有任何if-else分支来判断是否有足够的空间存放数据，kfifo每次入队或出队只是简单的+len 判断剩余空间，并没有对kfifo->size 进行取模运算，所以kfifo->in和kfifo->out总是一直增大，直到unsigned in超过最大值时绕回到0这一起始端，但始终满足：kfifo->in - kfifo->out <= kfifo->size。

（4）使用Memory Barrier

- mb()：适用于多处理器和单处理器的内存屏障。
- rmb()：适用于多处理器和单处理器的读内存屏障。
- wmb()：适用于多处理器和单处理器的写内存屏障。
- smp_mb()：适用于多处理器的内存屏障。
- smp_rmb()：适用于多处理器的读内存屏障。
- smp_wmb()：适用于多处理器的写内存屏障。

Memory Barrier使用场景如下：

- A、实现同步原语（synchronization primitives）
- B、实现无锁数据结构（lock-free data structures）
- C、驱动程序

程序在运行时内存实际访问顺序和程序代码编写的访问顺序不一定一致，即内存乱序访问。内存乱序访问行为出现是为了提升程序运行时的性能。内存乱序访问主要发生在两个阶段：

- A、编译时，编译器优化导致内存乱序访问（指令重排）。
- B、运行时，多CPU间交互引起内存乱序访问。

Memory Barrier能够让CPU或编译器在内存访问上有序。Memory barrier前的内存访问操作必定先于其后的完成。Memory Barrier包括两类：

- A、编译器Memory Barrier。
- B、CPU Memory Barrier。

通常，编译器和CPU引起内存乱序访问不会带来问题，但如果程序逻辑的正确性依赖于内存访问顺序，内存乱序访问会带来逻辑上的错误。

在编译时，编译器对代码做出优化时可能改变实际执行指令的顺序（如GCC的O2或O3都会改变实际执行指令的顺序）。

在运行时，CPU虽然会乱序执行指令，但在单个CPU上，硬件能够保证程序执行时所有的内存访问操作都是按程序代码编写的顺序执行的，Memory Barrier没有必要使用（不考虑编译器优化）。为了更快执行指令，CPU采取流水线的执行方式，编译器在编译代码时为了使指令更适合CPU的流水线执行方式以及多CPU执行，原本指令就会出现乱序的情况。在乱序执行时，CPU真正执行指令的顺序由可用的输入数据决定，而非程序员编写的顺序。

## 七、总结

C++ 无锁队列在多线程编程中占据着举足轻重的地位。它不仅是解决多核心优化问题的关键技术之一，也是迈向高效多线程编程的重要基石。

在多线程编程的实际应用中，无锁队列的性能优势使其在各种场景下都能发挥出色的作用。无论是游戏开发、高性能计算、并发编程还是高并发网络编程，无锁队列都能为程序提供高效的数据处理和同步机制。

从单生产者单消费者队列到多生产者多消费者队列，不同的场景需求都能找到合适的无锁队列实现方式。链式队列、数组队列和环形队列等多种形式的无锁队列，为开发者提供了丰富的选择，以适应不同的数据量和应用场景。

无锁队列的高效性、线程安全性、可扩展性和低延迟等特点，使其成为处理高并发任务的理想选择。它避免了锁竞争和死锁问题，减少了上下文切换和线程调度的开销，提高了缓存的命中率，从而极大地提升了程序的性能。

随着多核心处理器的普及和高并发应用的不断增加，C++ 无锁队列的应用前景将更加广阔。开发者可以继续深入研究和优化无锁队列的实现，以满足不断增长的性能需求和复杂的应用场景。相信在未来的多线程编程领域，C++ 无锁队列将继续发挥重要作用，为构建高效、可靠的多线程应用程序提供有力支持。

多线程 4　C/C++开发 133　linux内核 129

上一篇 · 深入了解Pthread：强大的多线程编程利器