
# 课 程 设 计

设计题目 线程安全型队列

学生姓名 毛冬阳

学　　号 201910311127

专业班级 网络 191

指导教师 毕　坤

**2022 年 3 月 9 日**

## 摘要

本设计希望实现一个线程安全型队列，比较的两种实现方式，其一是操作系统课程所学的通过信号量实现读者写者模型。这是保证线程安全的常用方法，但是对于简单操作其成本是高昂的，可能加锁成本远高于其计算成本。在多核心的情况下，对于读写的轻量级操作，采用无锁队列的性能要高于有锁的操作。因为无锁队列的实现，是基于 CPU 指令集的 CAS 原子操作，因此实现无锁队列等无锁数据结构更能充分利用计算机的硬件支持。并且了解相关的无锁理论。

源代码参见:https://github.com/MAO-Dongyang/OperatingSystem/tree/main/Course_Project



关键字：　　线程安全，无锁队列，原子操作

# 目录

# 1 引言

线程安全型队列在计算机软件开发的多个领域都有重要应用，例如在网上商城领域，线程安全型队列可以用了实现有序购买功能，在电子游戏领域，线程安全型队列可以用了实现任务队列功能。因此本次课程设计的目标是设计一个线程安全的队列，并尽可能提高其性能。所谓线程安全，就是该队列能够实现多个线程同时正确的增删改队列结点，也就是能够实现对队列这个临界资源的保护。正确性肯定是我们第一要追求的，同时我们也追求对于有大量读写操作的高性能实现。具体表现为使用 CAS 操作实现无锁队列。

## 1.1 动机

无论是在计算机内部还是应用场景中，当应用多线程去解决同一个问题的时候，或者对同一个问题给出不同的指令的时候。这时任务就是一个临界资源，具体表现为多线程会同时读取或者修改一个数据结构。而我们需要给出这样一个规则，保证多线程访问的正确性。并且当这种读写是密集的时候，我们要在保证正确性的前提下，尽可能的去追求高效率。

所以本设计，旨在展现不同的解决方案以及它们的优点，供未来的应用者进行参考。

## 1.2 要解决的问题

为实现线程安全的队列，本系统要以下几个功能：

1. 通过 Random 函数自动产生数据插入队列，并且最终通过脚本程序实现对结果的校验；
2. 以数据可视化的方式，比较阻塞法和 CAS 操作方法的优越性；
3. 可以支持任意的（操作系统或者硬件允许的）线程进行读写操作，保证软件不成为短板；
4. 保证尽可能的高性能，对于密集读写操作表现良好；
5. 源程序简单可读，易于理解。

## 1.3 具体设计

要创建一个线程安全型队列，首先我们可以用信号量机制*实现一个读者写者模型。这也是通常情况下保证线程安全的最重要的方式，通过信号量机制，我们可以建立一种对于互斥资源访问的规则。而在这个规则下，线程是安全的 [1]。

但在大约十五年后提出的“无锁队列”这个概念，这是另外一种解决线程安全问题的思路。就正如经常演示的作为例子的那个线程不安全的程序一样，两个线程对于同一个数进行加一操作和减一操作，最终却无法达到正确的结果。信号量就像制定了一个规则，让各个线程按规则去操作，不出现争用的情况。而造成这一切的本质原因，就是因为“+1”操作，不是原子操作（atomic operation），所以才造成了这么多问题。但如果我们对于临界资源的访问是原子操作，那么这个问题就不会存在，而这就是“无锁”的中心思想。无锁并不是凭空产生的，它要求 CPU 指令集支持，需要指令层面的实现，让我们的操作从事实上不会被打断。对于无锁队列而言，我们使用 CPU 提供的原子操作，一次性完成对 Head 或 Tail 指针的读写，实现无锁同步 [2]。

为了同时展现并且比较这两种解决问题的思路。解决方案部分被分成了两个部分，其中一部分采用阻塞模型，是传统的信号量的方法。另一部分是 lockfree 也就是采用无锁的方法。课程源代码也会包含这两种方法的实现。

# 2 采用阻塞模型的线程安全队列

## 2.1 模块功能

此程序包含 6 个模块，分别是初始化兼创建模块、插入模块、删除模块、清空模块、查找模块和打印模块。先是有进程自动初始化并随机产生一

*1965 年，荷兰学者 Dijkstra 提出的信号量（Semaphores）机制是一种卓有成效的进程同步工具。在长期且广泛的应用中，信号量机制又得到了很大的发展，它从整型信号量经记录型信号量，进而发展为“信号量集”机制。现在，信号量机制已经被广泛地应用于单处理机和多处理机系统以及计算机网络中。

个队列，然后创建多个线程，线程同时进行插入结点，删除指定位置结点，查找指定元素及打印队列操作，为清楚区分读和写的操作这里分出了两个线程一个为读者线程一个为写者线程，前者具有查找指定元素功能，后者具有插入和删除兼打印队列功能。对于所有的查找，插入和删除数都是随机产生的，更具有代表性。工作原理如下图2.1所示

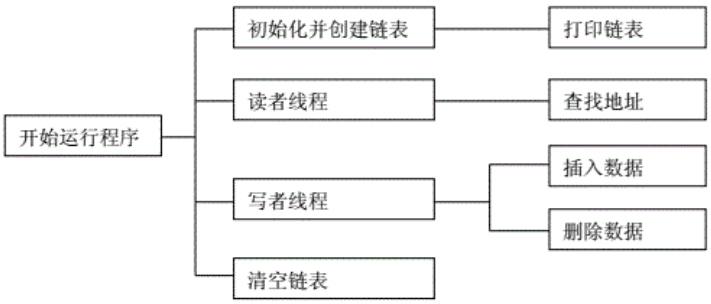


图 2.1: 程序工作原理图

## 2.2 程序详细流程

该队列以链表的数据结构进行实现，链表不需要连续的储存空间进行存储，同时具有较高的插入删除效率。

第一步，构建链表的基本属性。包括结点结构体，链表结构体，初始化及创建链表函数，插入函数，删除函数，清空函数，查找函数，打印函数。

第二步，创建三个线程分别进行插入、删除和查找操作，主函数内创建完链表后通过传参结构体向线程传递链表参数，在线程内使用互斥量对链表的修改操作进行保护。运行过程中所有变量给定固定初始值，测试多线程同步的正确性。

第三步，构建两个线程分别为读者线程（执行查找操作），写者线程（执行插入和删除操作），所有变量使用随机数定义，利用互斥量对读者数的修改操作进行保护，利用信号量对链表的修改操作进行保护，从而实现读读共享，读写互斥，写写互斥的读者写者问题。并测试读者优先状态下链表多线程操作的正确性。具体流程如下图2.2所示

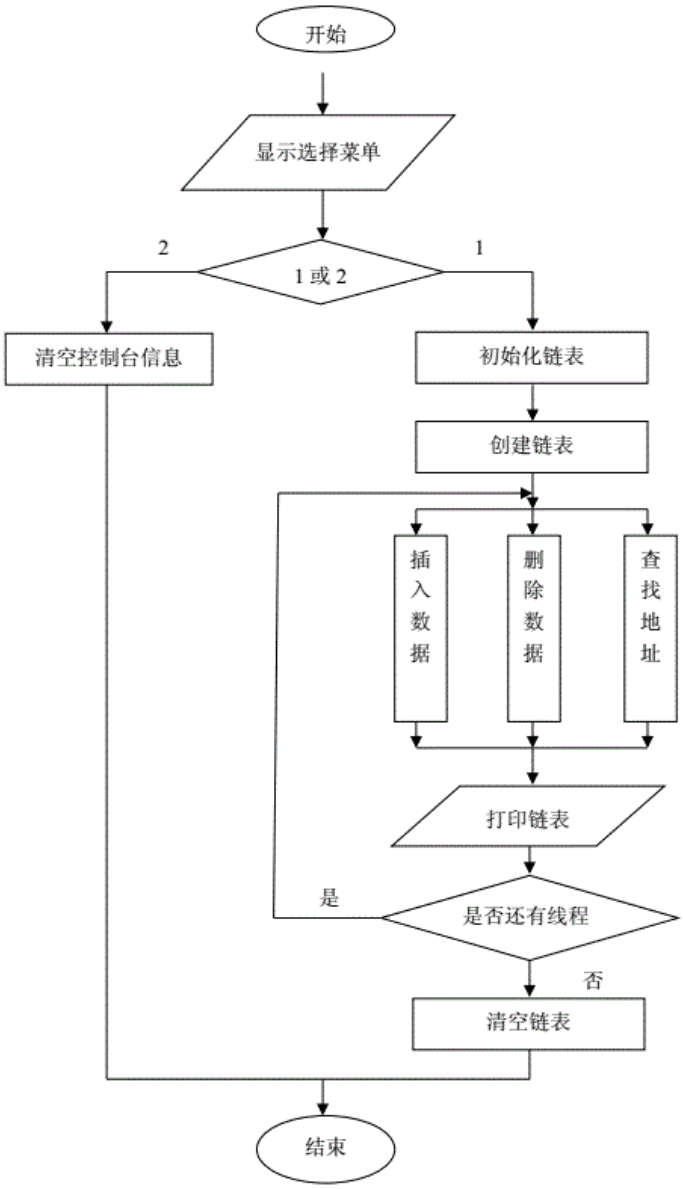


图 2.2: 程序流程图

## 2.3 程序实现

首先要实现列表的结构，在这里以链表的数据结构实现队列，然后要定义实现各种功能的函数。

### 2.3.1 节点结构

节点构成简单，仅由一个整型数构成内容，以及一个指向下一节点的指针。在实际情况中内容代表着不同资源，用列表的形式进行调度。但这里为了说明简单与测试方便，仅选取一个数字。

```
class QueueNode {
  public:
  int        val;
  QueueNode* next;
  QueueNode(int val) : val(val) {
    next = NULL;
  }
};
```

### 2.3.2 队列类

这就是数据结构中普通的队列，我们可以充分使用面向对象程序语言的特性，将我们需要的功能函数写在类里面，并且把锁以及条件变量当做成员变量。这样管理起来更方便，调动起来更直观。我们设定队列的最大长度为 10000，虽然列表可以充分利用储存空间，但是在计算机中也不能无限长。队列满了之后，就需要先出队才能入队。同时我们把 readcount 也定义在这里面。函数包括 enqueue、dequeue、search、show 功能非常直观。

```
class MutexQueue {
  public:
  MutexQueue();
  bool enqueue(int val);
  int  dequeue();
  ~MutexQueue();
  QueueNode* search(int val);
  int show();

  private:
  int        max_size = 100000;
  int        queue_size;
```

```
    QueueNode* tail;
    QueueNode* head;
    std::mutex wrt;
    std::mutex mtx;
    int readcount;

    std::condition_variable not_full;
    std::condition_variable not_empty;
};
```

### 2.3.3 函数实现

这个是入队函数，std::unique_lock 是写锁的管理类，简而言之就是在函数作用域内自动上锁，出了函数作用域，就自动释放。同时 wait 和 signal 功能和 mutex 类没有区别。首先我们判断队列是不是满了，如果满了，我们要 wait 一个条件变量，相当于释放 wrt(写锁)，同时进行等待，等待 not_full 条件，如果条件满足，队列又不满了，就可以继续执行，添加节点，把队列长度加 1，并且释放一个 not_empty 的条件变量。最后释放掉写锁。

```
bool MutexQueue::enqueue(int val) {
    QueueNode* add_node = new QueueNode(val);
    std::unique_lock<std::mutex> lck(wrt);

    while (queue_size == max_size) {
        printf("Slot Full!\n");
        not_full.wait(lck);
    }
    tail->next          = add_node;
    tail                = add_node;
    queue_size++;
    not_empty.notify_all();
    lck.unlock();
    return 1;
}
```

出队和入队同理，不过如果长度为零时，出队要等待 not_empty 的条件变量。并且完成操作后释放掉一个 not_full 的条件变量。同样也使用管理类对写锁进行管理。在实践操作中，锁的管理类，可以避免释放遗忘，它自动保护一个函数的作用域。在阻塞法的多线程编程中经常用到。

```
int MutexQueue::dequeue() {
  int val;
  std::unique_lock<std::mutex> lck(wrt);

  while (queue_size == 0) {
    //printf("Slot empty!\n");
    not_empty.wait(lck);
  }
  queue_size--;
  val = head->next->val;
  head = head->next;
  not_full.notify_all();
  lck.unlock();
  return val;
}
```

这是一个读操作，使用最基础的读者写者模型，读锁 (mtx)，保证读操作可以并行，并且排斥写操作。readLock.unlock() 和 readLock.lock() 之间的部分，也就是整个程序的中间部分，是有效操作，最终返回指定值 val 节点的指针。

```
QueueNode* MutexQueue::search(int val) {
  std::unique_lock<std::mutex> readLock(mtx);
  ++readcount;
  if (readcount == 1) {
    wrt.lock();
  }
  readLock.unlock();
  QueueNode* p;
  p = head;
  for (int i = 0; i < queue_size; i++) {
    if (p->val == val) {
```

```
        break;
      } else {
        p = p->next;
      }
    }
    readLock.lock();
    --readcount;
    if (readcount == 0) {
      wrt.unlock();
    }
    readLock.unlock();
    return p;
  }
```

show 和 search 同理，只不过是将所有节点值打印出来。有效代码如下：

```
  int MutexQueue::show() {
    ......
    QueueNode* p;
    p = head;
    std::cout << "| |-->";
    for (int i = 0; i < queue_size; i++) {
      if (p->val > 0) {
        std::cout << p->val << "-->";
      }
      p = p->next;
    }
    std::cout << "NULL\n";
    ......
  }
```

### 2.3.4 测试程序

我们在测试程序中实现两个功能函数 produce 和 consume，每次调用函数保证生产或者消费一个 task_number 量的节点，并把节点放到队列中。

特别的 produce 的值是由线程号作为偏移量生成的保证不重不漏，这样在验证阶段才能保证我们对资源的管理是没有任何差错的。能处理差异资源的程序一定能处理非差异资源。

```
void produce(int offset)
{
  for (int i = task_number * offset; i < task_number * (offset
      + 1); i++) {
    //printf("produce %d\n", i);
    lfq->enqueue(i);
  }
}

void consume()
{
  for (int i = 0; i < task_number; i++) {
    int res = lfq->dequeue();
    if (res > 0) {
      //printf("consume %d\n", res);
    }
    else {
      //printf("Fail to consume!\n");
    }
  }
}
```

在主函数中，我们以线程向量的方式实现多线程，thread_vector1 是生产线程的向量集合，thread_vector2 是消费线程的向量集合。我们通过命令行参数，指定总共有多少对[†]线程 (thread_number)，以及每个线程处理多少个任务 (task_number)，默认是 10 对线程，每个线程处理 1000 个任务。在启动线程和等待线程上下文有 system_clock 用于计时，统计任务完成时间。

```
int main(int argc, char** argv) {
  lfq = new MutexQueue;
```

[†]每创建一个生产线程，就要创建一个消费线程，并且任务量相等

```
    std::vector<std::thread> thread_vector1;
    std::vector<std::thread> thread_vector2;
    if (argc < 3) {
      thread_number = 10;
      task_number = 1000;
    } else {
      thread_number = atoi(argv[1]);
      task_number = atoi(argv[2]);
    }
    auto start = std::chrono::system_clock::now();
    for (int i = 0; i < thread_number; i++) {
      thread_vector1.push_back(std::thread(produce, i));
      thread_vector2.push_back(std::thread(consume));
    }
    for (auto& thr1 : thread_vector1) {
      thr1.join();
    }
    for (auto& thr2 : thread_vector2) {
      thr2.join();
    }
    auto end = std::chrono::system_clock::now();
    auto elapsed = std::chrono::duration_cast<std::chrono::
       milliseconds>(end - start);
    std::cout << elapsed.count() << "ms" << "\n";
    return 0;
  }
```

## 2.4 检测与运行

### 2.4.1 检测脚本

程序检测的思路是，我们首先将运行结果进行输出，输出为如下格式。表示生产的资源与消耗的资源，后面的整数表明生产或消耗哪个资源。也就是节点中存储的整数。

```
produce 363
```

```
consume 21
```

然后我们把运行结果输出到文件中，用 Python 写成的脚本检测文件，检测思路是依据 output 文件的内容创建两个字典，把运行结果添加到每对应的字典中。如果生产消费一致就说明程序运行不出错，任何生产的都被消费了。生产与消费相对应，最后输出程序运行正确。即该脚本程序运行程序的运行结果是正确的。如此保证了，所有程序是在运行结果是正确的。并且所有程序都通过该脚本的检验，以保证正确。

```
import os

f = open("./output")
line = f.readline()

dict_produce = {}
dict_consume = {}

while line:
a = line.split()
key = a[0]
if key == "produce":
val = int(a[1])
if dict_produce.get(val, 0) == 1 or dict_consume.get(val, 0)
   == 1:
print("Mutex error: ", key, val)
exit()
dict_produce[val] = 1
elif key == "consume":
val = int(a[1])
if dict_produce.get(val, 1) == 0 or dict_consume.get(val, 0)
   == 1:
print("Mutex error: ", key, val)
exit()
dict_consume[val] = 1
line = f.readline()
f.close()
```

```
26

  print("============================")
28  print("Mutex correctness test pass!")
  print("============================")
```

由于我选了 Makefile 可以配合 shell 脚本可以一键运行检测，make gcov 和 make gporf 可以显示，程序运行状态，比如每行运行了多少次，以及程序的时间消耗在哪个函数。是进行调试观察的工具。与主程序无关，仅仅只是便于分析和调试。

```
1  #!/bin/sh
  #make run analyse and check the program
3  make
  ./a.out > output
5  make gcov
  make gprof
7  python3 check.py
```

# 3 采用无锁模型的线程安全队列

无锁算法 (lock-free algorithm) 和无等待算法 (wait-free algorithm) 的近代起源可以追溯到 Herlihy 的论文 Wait-free synchronization,1991。在此之前，计算机硬件所提供的同步原语 (synchronization primitives) 只有原子寄存器 (atomic registers) 和少数经典原语比如 test_and_set 和 fetch_and_add。算法的研究也因此局限在利用这些有限的同步原语来设计简单的并发对象 (concurrent object)。Herlihy 的这篇论文从理论上证明了现有同步原语不够强大，不足以用来实现所有的并行对象。论文的核心在于用解决共识协议 (consensus protocol) 的能力来衡量同步原语的同步能力。原子寄存器只能解决一个线程的共识问题，所以共识数 (consensus number) 是 1；而 fetch_and_add 能解决最多两个线程的共识问题，其共识数是 2。Herlihy 更进一步提出了存在共识数为无限的同步原语，也就是现在被主流硬件所

广泛采用的 compare_and_swap，能用来实现所有的并发对象。

从此依赖原子寄存器的算法成了历史，compare_and_swap 可以轻易实现各种简单的并发对象比如互斥锁。随后 Herlihy 在 A methodology for implementing highly concurrent data objects, 1993 中提出了一种通用构建 (universal construction) 能把所有的顺序数据结构 (sequential data structure) 转化为无锁数据结构。基本思路是线程间共享一个指向数据结构的指针。每当一个线程企图修改数据结构的时候，它在线程局部创建一个当前数据结构的拷贝然后做出相应的修改。完成修改后使用 compare_and_swap 来尝试将共享的数据结构指针更新成指向本地拷贝的指针。如果 compare_and_swap 失败则说明有其他线程抢先完成了修改，这个线程将重新读取共享指针并重复拷贝和修改的操作直到 compare_and_swap 成功。这个算法保证了无锁进展 (lock-free progress)：就算任何一个线程在操作数据结构时崩溃，都不会影响到其他线程，系统作为一个整体总是能保证进展。但是不难想到这种通用构建的效率并不高，因为它实质上阻止了任何对于数据结构的并发操作——任何时刻都只可能有一个线程成功修改共享的数据结构指针，其他线程的操作必然会失败和重试 [2]。

而很重要的技术就是 CAS 操作——Compare And Set，或是 Compare And Swap，现在几乎所有的 CPU 指令都支持 CAS 的原子操作，X86 下对应的是 CMPXCHG 汇编指令。有了这个原子操作，我们就可以用其来实现各种无锁（lock free）的数据结构 [3]。这个操作用 C 语言等价描述如下，看一看内存 reg 里的值是不是 oldval，如果是的话，则对其赋值 newval。

```
int compare_and_swap (int* reg, int oldval, int newval)
{
  int old_reg_val = *reg;
  if (old_reg_val == oldval) {
    *reg = newval;
  }
  return old_reg_val;
}
```

不过不同于我们写的 C 语言函数，编译器会把这个函数编译成一行汇编语言，这样这就成了一个原子操作，不会被中断。相当于从底层层面进行

加锁 [4]。

## 3.1 设计思路

### 3.1.1 插入节点的实现

如图3.1所示，基本实现就是一个尝试-失败-重试的模型，其实现的语义与有锁版本一样：[5]

- 获取当前尾指针 Tail 指向的节点；
- 然后修改尾节点的后继指针①；
- 修改尾指针 Tail 指向新的尾节点②。

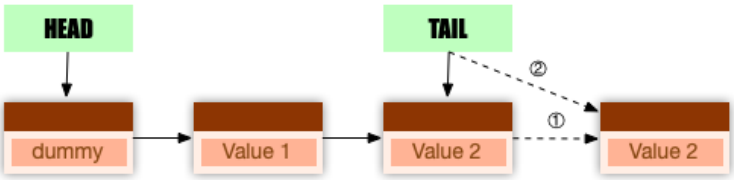


图 3.1: 利用 CAS 操作插入节点

### 3.1.2 删除节点的实现

如图3.2所示, 进行节点的比较，然后删除。当然，对于删除操作而言，这两个操作还是分开的，因为标记在当前节点上，但是 next 在前驱结点。只是我们在判断是否删除时，我们可以同时判断这两个条件。

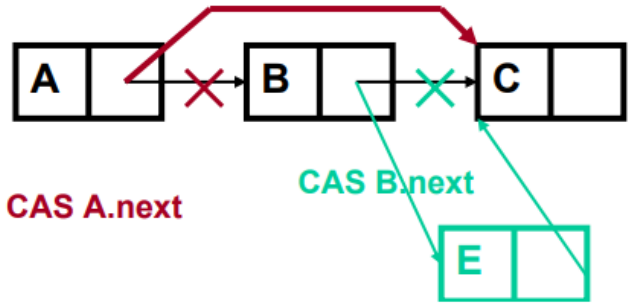


图 3.2: 利用 CAD 操作删除节点

## 3.2 程序实现

队列的实现，与使用阻塞方法的方式类似，所以不再重复。检测方式同理。本部分重点展现，如何在代码中实现原子操作以及其注意事项。

### 3.2.1 CAS 原子操作

入队具体代码实现，同3.1.1小结，如果比较后结果正确，就用 add_node 进行替换，并且 break 跳出循环，如果不正确就后移寻找下一个。

```
bool LockFreeQueue::enqueue(int val)
{
  QueueNode* cur_node;
  QueueNode* add_node = new QueueNode(val);
  while (1) {
    cur_node = tail;
    if (__sync_bool_compare_and_swap(&(cur_node->next), NULL,
        add_node)) {
      break;
    }
    else {
      __sync_bool_compare_and_swap(&tail, cur_node, cur_node->
          next);
    }
  }
  __sync_bool_compare_and_swap(&tail, cur_node, add_node);
  return 1;
```

```
16  }
```

出队代码类似,具体原理同3.1.2,将头节点 head 复制给现有节点 cur_node，然后头节点替换成下一个节点，返回现有节点的值，完成出队。

```
  int LockFreeQueue::dequeue()
2 {
    QueueNode* cur_node;
4   int        val;
    while (1) {
6     cur_node = head;
      if (cur_node->next == NULL) {
8       return -1;
      }
10
      if (__sync_bool_compare_and_swap(&head, cur_node, cur_node
         ->next)) {
12      break;
      }
14  }
    val = cur_node->next->val;
16  delete cur_node;
    return val;
18 }
```

### 3.2.2 注意事项

由于 CAS 操作是直接对内存进行操作，我们不希望编译器对此进行优化将规则进行打破。所以一般我们会使用 volite 选项，在这里我直接使用 gcc 编译器的-O0 选项，不进行优化。将 Makefile 中的 CFLAGS 调整为：

```
CFLAGS=-std=c++20 -O0
```

### 3.2.3 程序测试

虽然方式发生改变，但程序结构相对于采用阻塞法的模型并未发生变化，测试主程序仍然继续沿用。并且该程序通过2.4.1小结的 shell 脚本一键运行，通过检测。

# 4 性能比较

## 4.1 阻塞模型与无锁模型的比较

通过2.3.4小结的方法读取函数的运行时间。最终统计运行结果如下图4.1所示，横轴表示两种方式在不同问题情形下的分布，prod 表示生产者，cons 表示消费者 (横轴标号 2prod-2cons，即意为两个生产者两个消费者的情形)，纵轴每百万次入队出队耗时的时间消耗表，我们可以很容易的看出，在只有一个生产者的情况下，无锁队列明显要比阻塞算法表现的好，但当有几个生产者时，无锁队列的性能很快就衰减，耗时大量增加。可能是因为尝试的出错概率增大，导致最终性能劣化明显。

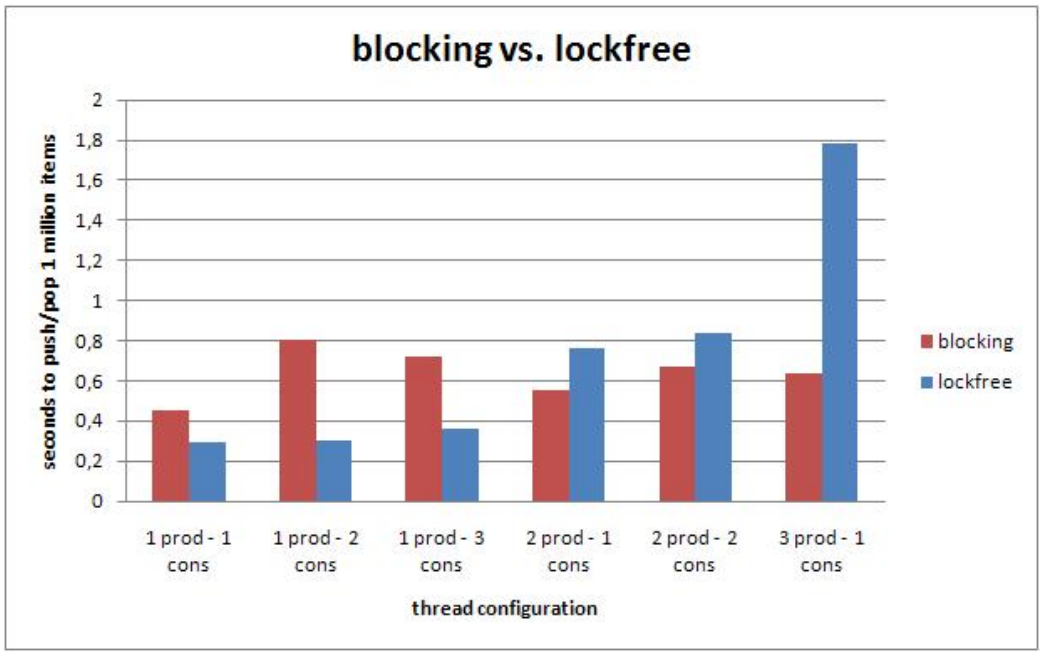


图 4.1: 若干情况下的性能比较图

虽然比较并不完全严谨，仅仅只是粗略的测算。但这也解释了为什么经典的信号量机制，仍然具有它的优越性。很明显信号量机制适用于更广泛的使用场景，尤其是多入队多出队操作。在此情况下错误的成本还是远高于预防的成本，只有当错误概率很小的情况下，尝试-失败-重试的模型才有它的优越性。而产生这种错误，是由于队列操作只发生在队头和队尾。导致在无锁情况下，并发的插入操作发生争抢，并发的删除操作也发生争抢，导致纠错成本极速增高。如果我们可以对队列进行随机读写操作，随机选择插入或者删除的位置，那么争抢发生概率就会大大减少，从而最大程度发挥无锁操作的效率。

# 5 总结

最终事实说明，无锁队列拥有它的优越性，但并不是所有情况下都拥有优越性。在特定的数据结构下，比如队列或者栈，由于频繁的操作，仅发生在固定的点 (比头或者尾)，导致错误发生率增高，纠错成本变大。在这种情况下，阻塞法可以起到很好的确定规则的作用，虽然不够灵活，但避免了纠错成本。但如果进行随机读写，无锁模型可以避免锁住整个数据结构，从而实现真正的并发操作，如果发生错误再及时纠错，增加了操作的灵活性也提高了效率。两种方法各有优劣，需要在需求场景中进行选择。

同时该无锁队列也没有实现所有优化，就比如 RCU 操作[‡]并且也没有防止 ABA 问题[§]的出现，但是也没出现过 ABA 的问题，ABA 问题的出现概率在内存足够大的情况下会很小 [6]。虽不完美，但毕竟实现了无锁队列。

本实验深刻的说明了，充分利用计算机的属性可以提升计算机的性能。所有计算机使用的方式与策略，都是为了让计算机执行正确的操作，所谓最好的技术就是充分利用计算机工具属性的技术。任何对于计算机底层了解都可以转化成软件上的奇效，如果我们用计算机工作，那么就需要了解它，才能更充分的应用它。更充分的说明了，没有最好的方法，只有在问题场景中最合适的办法。

---

[‡]读-复制-更新（RCU）是一种同步机制，它避免使用锁基元，而多个线程同时读取和更新通过指针链接并属于共享数据结构的元素。（例如，链表、树、哈希表）

[§]实现无锁数据结构时会遇到 ABA 问题的常见情况。如果从列表中删除某个项目，然后将其删除，然后分配新项目并将其添加到列表中，则由于 MRU 内存分配，分配的对象通常与已删除的对象位于同一位置。因此，指向新项目的指针通常等于指向旧项目的指针，从而导致 ABA 问题。

# 插图

# 参考文献

[1] H. Vantilborgh and A. V. Lamsweerde. On an extension of dijkstra's semaphore primitives. *Information Processing Letters*, 1(5):181–186, 1972.

[2] Herlihy and Maurice. A methodology for implementing highly concurrent data structures. *ACM SIGPLAN Notices*, 25(3):197–206, 1990.

[3] M. Herlihy. Transactional memory: architectural support for lock-free data structures. *Acm Sigarch Computer Architecture News*, 21(2):289–300, 1993.

[4] Maged M. Michael and Michael L. Scott. Simple, fast, and practical non-blocking and blocking concurrent queue algorithms. In *Proc Acm Symp on Principles of Distributed Computing*, 1996.

[5] Marçais, Guillaume, Kingsford, and Carl. A fast, lock-free approach for efficient parallel counting of occurrences of k-mers. *Bioinformatics*, 2011.

[6] R. Rajwar. Computer architecture providing transactional, lock-free execution of lock-based programs, 2008.