

# 并行与分布式计算

## Project1: 线程安全队列的实现

院(系)名称: 数据科学与计算机学院

专业名称: 计算机科学与技术

学生姓名: 王永锋

学生学号: 16337237

指导教师: 陈鹏飞

二〇一八年四月三十日

# 目 录

<b>1</b>	<b>项目要求及背景</b>	<b>1</b>
1.1	项目要求	1
1.2	项目背景及分析	1
1.3	生产者消费者问题	1
<b>2</b>	<b>项目实施</b>	<b>2</b>
2.1	关键问题分析	2
2.2	代码实现	2
2.2.1	push 方法的实现	2
2.2.2	pop 方法的实现	2
<b>3</b>	<b>生产者消费者问题</b>	<b>4</b>
3.1	工作分配方式	4
<b>4</b>	<b>项目运行结果</b>	<b>5</b>
4.1	项目感想	5
	<b>附录 A 文件的组织</b>	<b>6</b>

# 1 项目要求及背景

## 1.1 项目要求

### 题目

Implement a multi-access threaded queue with multiple threads inserting and multiple threads extracting from the queue. Use mutex-locks to synchronize access to the queue. Document the time for 1000 insertions and 1000 extractions each by 64 insertion threads (producers) and 64 extraction threads (consumers).

实现一个多线程队列，能够多个线程读取，或写入队列中。使用互斥锁对队列的访问进行同步。通过 64 个插入线程（生产者）和 64 个提取线程（消费者）记录 1000 次插入和 1000 次提取的时间。

## 1.2 项目背景及分析

C 库中的很多代码，当初编写的时候就没有考虑到有关线程安全的问题，就连 c++ 中注明的 STL 库，在如今面对多线程的情况下，也可能会出现问题。这些在多线程下并发执行的代码，在不同的调度运行序列下，会产生不一样的结果，这样的代码我们称之为线程不安全的。反之，则是线程安全的。

## 1.3 生产者消费者问题

关于该问题，维基百科上有准确的描述。

生产者消费者问题（*Producer-consumer problem*），也称有限缓冲问题（*Bounded-buffer problem*），是一个多线程同步问题的经典案例。该问题描述了共享固定大小缓冲区的两个线程——即所谓的“生产者”和“消费者”——在实际运行时会发生的问题。生产者的主要作用是生成一定量的数据放到缓冲区中，然后重复此过程。与此同时，消费者也在缓冲区消耗这些数据。该问题的关键就是要保证生产者不会在缓冲区满时加入数据，消费者也不会缓冲区中空时消耗数据。

## 2 项目实施

### 2.1 关键问题分析

一个线程安全的队列，需要解决以下关键问题：

- 在对队列的数据结构进行修改的时候，要保证同一时刻只有一个线程在修改队列（push 或者 pop 操作）
- 在队列已空的时候，队列阻塞住直到队列中有新的内容。
- 在队列已满的时候，队列阻塞住直到有 consumer 从队列中取出数据。

### 2.2 代码实现

关键在 push 和 pop 的实现上，在队列中具有一个互斥锁，push 和 pop 方法都需要使用同一个锁来对队列进行操作，从而保证 push 方法和 pop 方法不会受到别的线程的影响。

#### 2.2.1 push 方法的实现

主要通过在对队列数据的操作前后加锁即可。注意在 push 前需要判断队列是否满，如果满了需要使用自旋锁，进行忙等待，直到有 Consumer 从队列中取得数据使得队列不满才能够向队列中放入数据。

```
1 void push(T contents){
2     while(this->full());
3     pthread_mutex_lock(&this->_mutex);
4     _queue.push(contents);
5     this->push_times++;
6     pthread_mutex_unlock(&this->_mutex);
7 }
```

#### 2.2.2 pop 方法的实现

与 push 的实现类似。同样需要注意如果队列空需要使用自旋锁锁住直到 Producer 往队列中放入数据。

```
1  while(_queue.size() == 0);
2  pthread_mutex_lock(&this->_mutex);
3  // 自旋锁，如果队列空，锁住直至有新元素加入
4  T r = _queue.front();
5  _queue.pop();
6  this->pop_times++;
7  pthread_mutex_unlock(&this->_mutex);
8  return r;
```

## 3 生产者消费者问题

在实现生产者与消费者时，主要思路是：创建 64 个 Producer 线程和 64 个 Consumer 线程，并在创建的过程使用一些方法将工作尽可能平均分配到每一个线程中，分配之后每一个线程就可以各自完成它们自己的工作。

### 3.1 工作分配方式

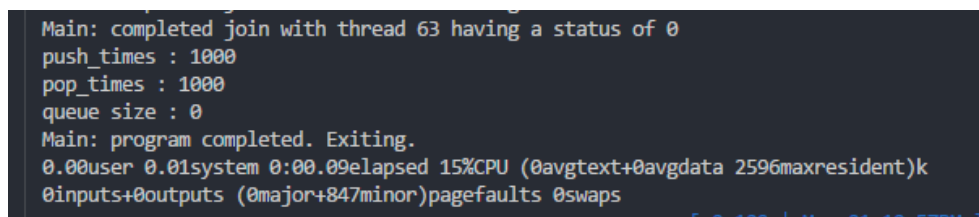
64 个 Producer 需要向队列中 push1000 个数据，为了将这个工作平均分配到 64 个线程中，我使用了以下的代码进行分配。

```
1 // NUM_THREADS = 64
2 // t为当前线程号，从0-63遍历
3 // 对64个线程分配任务，做到尽可能平均分配
4 int cur_push_times = left_push_times / (NUM_THREADS-t);
5 left_push_times = left_push_times - cur_push_times;
6 printf("cur_push_times:%d", cur_push_times );
```

每一个线程通过这一个过程，得到 cur\_push\_times，并将这个作为函数参数传递到生产者中，该生产者线程就只会执行 cur\_push\_times 次的 push 方法。通过这种方法，一些线程需要 push15 次，另一些线程需要 push16 次，做到了尽可能平均的分配。

## 4 项目运行结果

使用 `time ./a.out` 指令，得到程序的运行时间。同时，通过代码内部的实现，得到 `queue` 的 `push` 和 `pop` 运行的次数。截图如图 4.1 所示

A terminal window with a dark background and light-colored text. The text shows the output of a program, including completion status, push and pop counts, queue size, and system statistics like CPU usage and memory. The output is as follows:

```
Main: completed join with thread 63 having a status of 0
push_times : 1000
pop_times : 1000
queue size : 0
Main: program completed. Exiting.
0.00user 0.01system 0:00.09elapsed 15%CPU (0avgtext+0avgdata 2596maxresident)k
0inputs+0outputs (0major+847minor)pagefaults 0swaps
```

图 4.1 代码运行截图

### 4.1 项目感想

实现过程中，我在队列上花的时间不多，倒是在生产者，消费者上花了很长时间来实现。一开始我没有做任务的分配，直接由操作系统自己调度各个进程的运行，然后通过一个全局变量来控制 `push` 和 `pop` 次数，这部分的代码在 `main.cpp` 中。测试过程中发现，的确是可以运行，也能够做到控制 `push` 和 `pop` 的次数，但是运行效率极低，我开 64 个线程甚至会使我的电脑卡机，完全无法测试。可能是有代码的问题吧，后来我转变了思路，在主线程中分配各个线程的任务，然后各线程之间就不需要看看对方做什么了，只管完成自己的任务就行，这部分的代码在 `main2.cpp`。

在多现场的程序编写中，`posix` 虽然提供了接口，但是当线程创建，任务分配等工作都需要自己处理的时候，我的收获是对底层的细节了解得更加清楚，但与此同时，代码编写效率极低也是不可忽视的一个缺点。不过想到之后学习的 `OpenMP` 编程模型，代码编写的效率将会大大提高。

## 附录 A 文件的组织

```
1 16 计科+16337237+王永锋+Proj1
2 16 计科+16337237+王永锋+Proj1.pdf
3 code
4 a.out // 可执行文件
5 main2.cpp // 生产者与消费者的第二种实现
6 main.cpp // 生产者与消费者的第一种实现
7 makefile // 编译
8 queue.hpp // 线程安全队列的实现
```