C++ 中的生产者/消费者队列

米尔恰·内克苏

2020 年 10 月 5 日 麻省理工学院

评价我: 5.00/5 (3 票)

生产者/消费者机制的简单实现

本文描述了生产者/消费者队列的“普通”C++ 实现，并展示了这种同步机制的内部工作原理。

下载源代码 - 17.4 KB

背景

早在 70 年代多线程变得重要之前，生产者-消费者进程的问题就已经被研究过。这是因为这些过程在计算机与外部世界之间的任何交互中都起着不可或缺的作用。即使从打孔卡中读取数据，程序也需要“消耗”一张卡片，然后读取过程才能“生成”另一张卡片。同样，在生产下一条之前，打印机需要“消耗”一条打印线。

一段时间以来，这些问题一直存在于操作系统领域，一旦用户进程访问多线程设施，它们就会成为日常用户的关注点。有关不同实现的简短背景，可以查看维基百科页面。

如果您正在寻找 C# 实现，您可以查看Mark Clifton关于此主题的文章。在这里，我们将继续使用旧的 C++。

生产者/消费者示例

我们这里的示例将直接取自上述 Mark 的文章。我们将编写一个程序，找出所有小于某个极限的素数。然而，我们希望利用 CPU 的所有内核来“加速”进程。我将加速放在引号中，因为这里我们对实际速度不感兴趣：按照 Mark 的例子，我们将采用同样非常低效的函数来确定一个数是否为质数：

C++ 复制代码

```
bool IsPrime (int n)
{
  bool ret = true;
  for (int i = 2; i <= n / 2 && ret; ret = n % i++ != 0)
    ;
  return ret;
}
```

一般策略是让一个生产者线程将要检查“素数”的数字添加到队列中，而许多消费者线程将获取这些数字并将所有正结果放入另一个队列。为了增加幻想，输出队列将同时保留素数和计算它的消费者线程的 ID。

以下是用于此设置的数据结构：

C++ 复制代码

```
struct result
{
  int prime;
  int worker;
};
...
sync_queue<int> nums;
sync_queue<result> primes;
```

该nums队列保留所有号码进行测试和primes队列保持积极的成果。

sync_queue是一个生产者/消费者队列结构，为所有线程提供有序访问；我们马上就会看到它的内部运作。

生产者线程

这是最简单的。它只是将nums队列中的所有数字填充到某个限制。最后，它会放置许多零作为消费者完成工作的信号。消费者在0从队列中提取 a 时将终止：

C++ 复制代码

```
thread producer ([&nums]()->int {
  for (int i = 2; i < 500000; i++)
    nums.produce (i);

  for (int i = 0; i < NTHREADS; i++)
    nums.produce (0);
  return 0;
});
```

消费者线程

消费者线程的代码并不复杂。它从nums队列中提取一个数字，使用IsPrime函数检查它是否是素数，如果是素数，则在primes队列中发布一个新结果，添加自己的消费者编号以了解是谁计算的：

C++ 复制代码

```
auto checker = [&nums, &primes, thnum]()->int {
  int n = 1;
  while (n = nums.consume ())
  {
    if (IsPrime (n))
      primes.produce ({ n,thnum });
  }
  return 0;
};
```

当检索到的数字为 时0，函数返回并且线程终止。

主应用程序线程创建多个消费者线程并在启动生产者线程之前启动它们：

C++ 复制代码

```
thread* consumers[NTHREADS];
for (int thnum = 0; thnum < NTHREADS; thnum++)
{
  auto checker = [&nums, &primes, thnum]()->int {
    int n = 1;
    while (n = nums.consume ())
    {
      if (IsPrime (n))
        primes.produce ({ n,thnum });
    }
    return 0;
```

```
  };

  consumers[thnum] = new thread (checker);
  consumers[thnum]->start ();
}
```

跑圈地

现在一切都设置好了，我们可以开始表演了：

C++ 缩小▲ 复制代码

```
stopwatch t_prod, t_cons;
t_prod.start ();
t_cons.start ();

producer.start ();    //Start producer
producer.wait ();     //Wait to finish producing
t_prod.stop ();

//Show producer statistics
cout << "sync_queue finished producing" << " in " << fixed
  << setprecision (2) << t_prod.msecEnd ()/1000. << "sec" << endl;

//Wait for consumers to finish
for (int i = 0; i < NTHREADS; i++)
  consumers[i]->wait ();
t_cons.stop ();
cout << "finished consuming" << " in " << fixed
  << setprecision (2) << t_cons.msecEnd () / 1000. << "sec" << endl;

//Did we find all the primes?
cout << "Expecting 41538 primes, found "
  << primes.size () <<endl;

//Check who did what
vector<int> found_by(NTHREADS);
while (!primes.empty ())
{
  result r = primes.consume ();
  found_by[r.worker]++;
}

//Show consumers statistics
for (int i = 0; i < NTHREADS; i++)
  cout << "Consumer " << i << " found " << found_by[i]
    << " primes." << end;
```

在我的机器上（这个没有速度怪物），我得到类似的信息：

复制代码

```
sync_queue finished producing in 1.67sec
finished consuming in 4.34sec
Expecting 41538 primes, found 41538
Consumer 0 found 4869 primes.
Consumer 1 found 5530 primes.
Consumer 2 found 5467 primes.
Consumer 3 found 4844 primes.
Consumer 4 found 4863 primes.
Consumer 5 found 5529 primes.
Consumer 6 found 5596 primes.
Consumer 7 found 4840 primes.
```

不是让某个中央控制在消费者之间分配工作，而是sync_queue允许每个消费者选择其工作单元并生成结果。有些线程多一点，有些少一点，但总而言之，工作是公平分配的。

生产者/消费者队列的内部工作原理

sync_queue是从std::queue. 它提供了两种主要方法：produce和consume。为了同步访问，它使用信号量和临界区对象来保持一切一致。该produce方法很简单：

C++ 复制代码

```
template <class M, class C=std::deque<M>>
class sync_queue : protected std::queue<M, C>
{
public:
...
  /// Append an element to queue
  virtual void produce (const M& obj)
  {
    lock l (update);        //take control of the queue
    this->push (obj);       //put a copy of the object at the end
    con_sema.signal ();     //and signal the semaphore
  }
...
protected:
  semaphore con_sema;       ///< consumers' semaphore counts down until queue is empty
  criticalsection update;   ///< critical section protects queue's integrity
```

一个lock对象获得的关键部分，以防止同时访问。要生成的对象被推送到队列中，并发出信号量。当函数锁对象超出范围并且临界区被释放时。

消费稍微复杂一点：

C++ 复制代码

```
/// Extract and return first element in queue
virtual M consume ()
{
  M result;
  update.enter ();
  while (std::queue<M, C>::empty ())
  {
    update.leave ();
    con_sema.wait ();        //wait for a producer
    update.enter ();
  }
  result = this->front ();  //get the message
  this->pop ();
  update.leave ();
  return result;
}
```

再次，我们进入临界区并检查队列是否为空。如果是这样，我们离开临界区并开始等待消费者的信号量由生产者发出信号。当被信号唤醒时，我们再次进入临界区并再次循环。

此时，可能发生了两件事：

- 没有其他人得到该对象，我们发现队列不为空。在这种情况下，我们退出while循环，拿起对象并离开临界区。
- 另一个饥饿的消费者得到了对象，我们发现队列是空的。在这种情况下，我们离开临界区并等待消费者信号量处的另一个信号。

除了这些主要方法之外，还有另一种方法来检查队列是否为空，另一种方法是返回队列的大小。请注意，它们都只是指示性的，因为结果可能会在调用者有机会检查之前发生变化。

有界生产者/消费者队列

眼尖的读者可能已经注意到该sync_queue::produce方法没有错误检查。它愉快地调用std::queue::push并假设新对象有足够的内存。从上面例子中生产者和消费者线程的运行时间也可以看出这一点：生产者只用了 1.7 秒来填充要检查的数字队列，而

消费者用了 4.4 秒来清空它。

该bounded_queue级可以限制可排队的对象的数量。如果生产者发现有界队列已满，则必须等到消费者移除某些对象。为此，我们还需要一个信号量，pro_sema它用队列的最大大小进行初始化。该produce方法变为：

C++ 缩小▲ 复制代码

```
template< class M, class C = std::deque<M> >
class bounded_queue : public sync_queue<M, C>
{
public:
  bounded_queue (size_t limit_) : limit (limit_)
  {
    pro_sema.signal ((int)limit);
  }

  /// Append an element to queue. If queue is full, waits until space
  /// becomes available.
  void produce (const M& obj)
  {
    this->update.enter ();
    while (std::queue<M, C>::size () > limit)
    {
      this->update.leave ();
      pro_sema.wait ();
      this->update.enter ();
    }
    this->push (obj);
    this->con_sema.signal ();
    this->update.leave ();
  }
...
protected:
  size_t limit;
  semaphore pro_sema;   ///< producers' semaphore counts down until queue is full
```

您可以看到它与consume之前显示的方法更加相似。它进入临界区，如果队列已满，则通过等待反复尝试为新对象寻找空间pro_sema。

如果我们将质数示例更改为使用bounded_queue具有 20 个条目的结构，结果如下所示：

C++ 复制代码

```
bounded_queue finished producing in 4.32sec
finished consuming in 4.32sec
Expecting 41538 primes, found 41538
Consumer 0 found 5103 primes.
Consumer 1 found 5156 primes.
Consumer 2 found 5192 primes.
Consumer 3 found 5240 primes.
Consumer 4 found 5227 primes.
Consumer 5 found 5091 primes.
Consumer 6 found 5267 primes.
Consumer 7 found 5262 primes.
```

生产者线程所需时间与消费者所需时间相同。那是因为生产者被有限的队列大小所阻碍。

结论

本文中介绍的生产者/消费者队列为线程间通信提供了一种易于使用的机制。此处显示的线程原语（thread、critical_section、semaphore等）是 MLIB 项目的一部分。您可以从GitHub 项目页面下载完整的项目。

历史

- 2020 年 10 月4日：初始版本

执照

分享

关于作者

米尔恰·内克苏 没有提供传记

加拿大 手表该会员

评论和讨论

添加评论或问题 电子邮件提醒 Search Comments

第一 页上一页 下一页

虚拟方法
MBuehrer 5-Oct-20 19:58

回复：虚拟方法
Mircea Neacsu 5-Oct-20 20:19

刷新 1

一般 新闻 建议 问题 错误 答案 笑话 赞美 咆哮 管理员

使用Ctrl+Left/Right 切换消息，Ctrl+Up/Down 切换主题，Ctrl+Shift+Left/Right 切换页面。