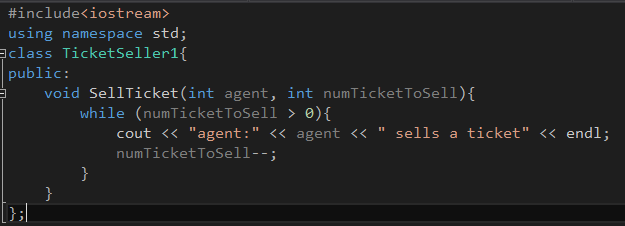
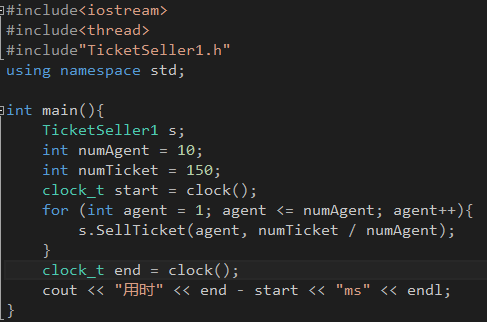
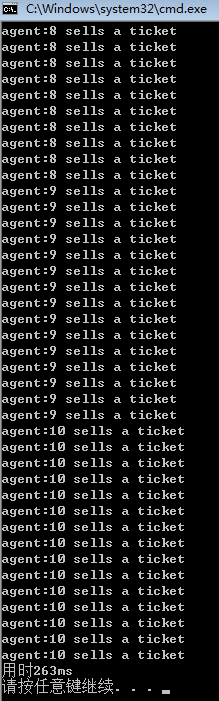
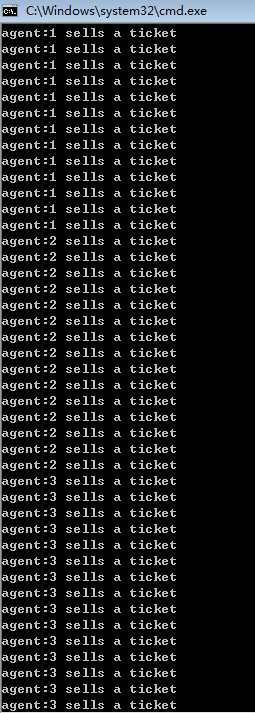
**1.2.1 TicketSeller1**

代码及运行结果如下





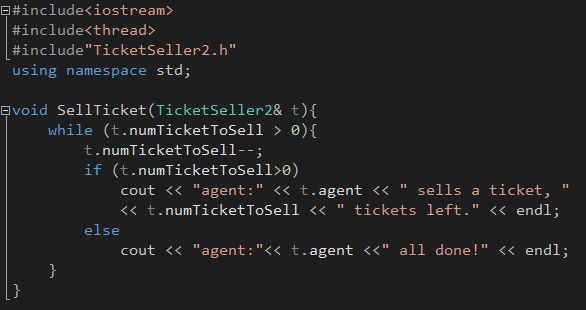
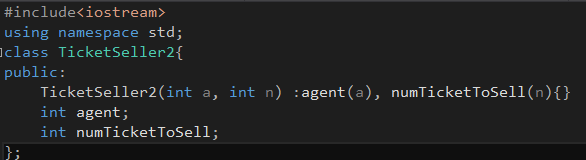


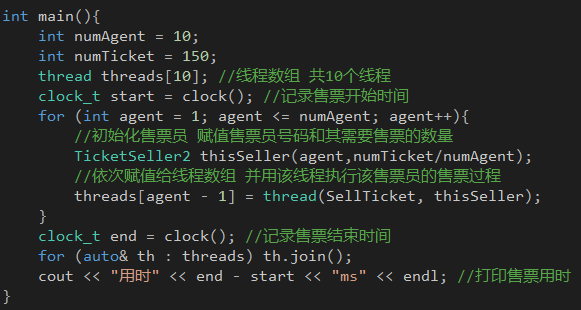
**代码分析：**

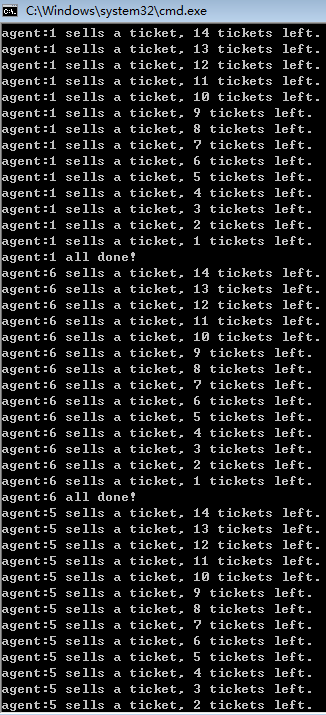
此样例中，共有10个售票员售票，有150张票需售出，主要思想是将所有的票平均分配给每个售票员进行销售。样例首先定义了一个TicketSeller1类，并在循环中逐次调用SellTicket方法，实际上并未采用多线程编程，相当于现实世界中10个售票员轮流进行售票，且每个人只能销售固定数量的票，极大地浪费了效率，由结果可看出耗时263ms。

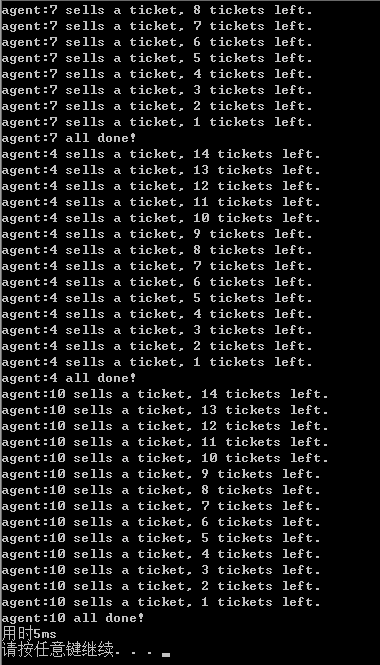
**1.2.2 TicketSeller2**

代码及运行结果如下







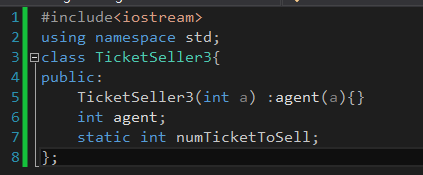


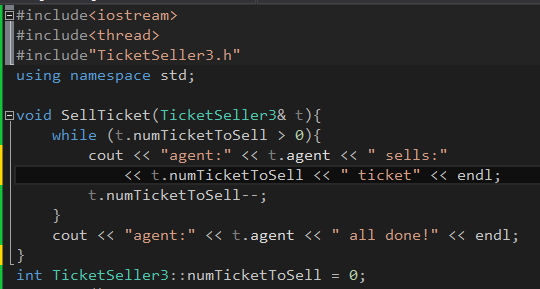
**代码分析：**

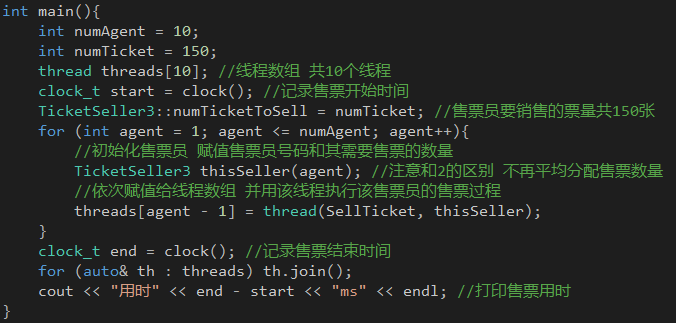
在此样例中，仍然采用的是10个售票员平均分配销售总计150张票的思路，但是由于采用了多线程编程，每个线程可以代表每个售票员的售票过程，加快了速度，故而整个售票过程只耗时5ms。但是此代码仍然存在着问题，每个售票员都只能销售固定数量的票，这样售票速度较快的售票员也无法多售票，浪费了资源，降低了整体效率。

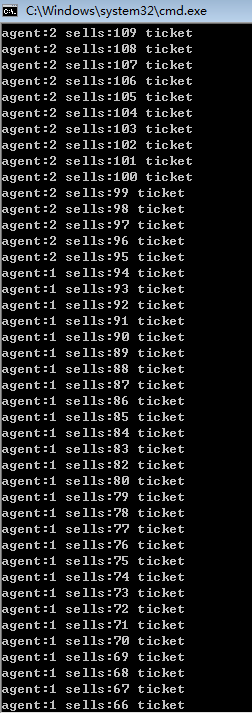
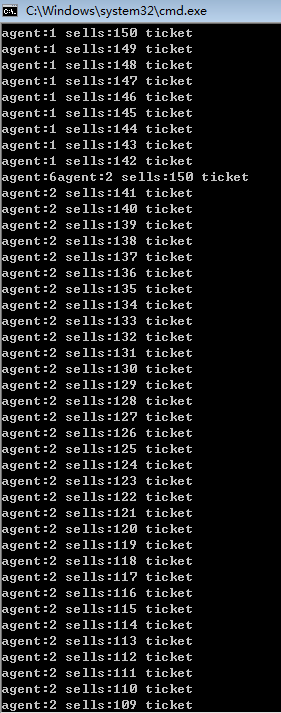
**1.2.3 TicketSeller3**

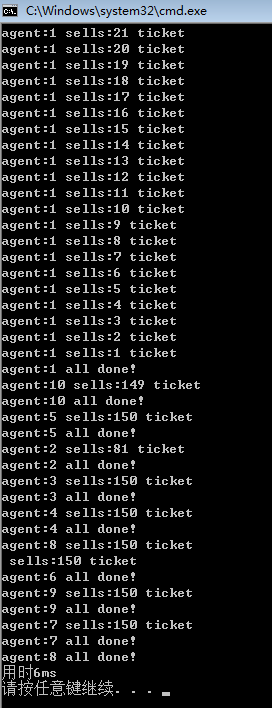
代码及运行结果如下







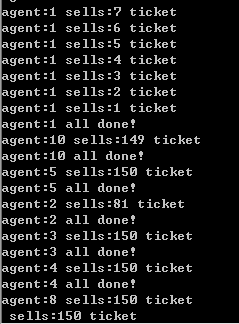




代码分析：

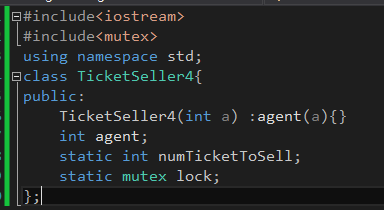
此样例的思路是解决TicketSeller2中的效率问题，即每个售票员不再平均分配售票量，所有售票员的共同任务都是将所有的150张票销售出去，销售快的售票员可以多售票，提高了整体的售票效率。但是此代码存在着一个问题，即没有对临界资源numTicketToSell这个变量进行互斥访问，或者说实现原子操作。导致了一个结果，就是多个售票员可能会同时销售同一张票，正如上图中的：

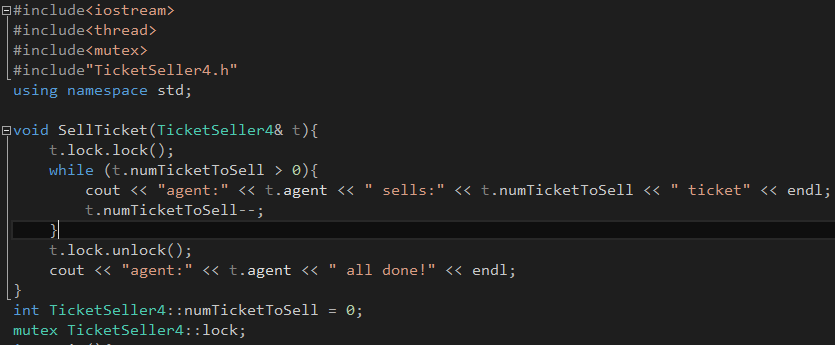
所示，6号售票员和2号售票员同时售出150号票；此外还存在一个问题，由于没有进行互斥访问该变量，导致对该变量更新的时候不能及时通知到其他线程，正如上图所示：

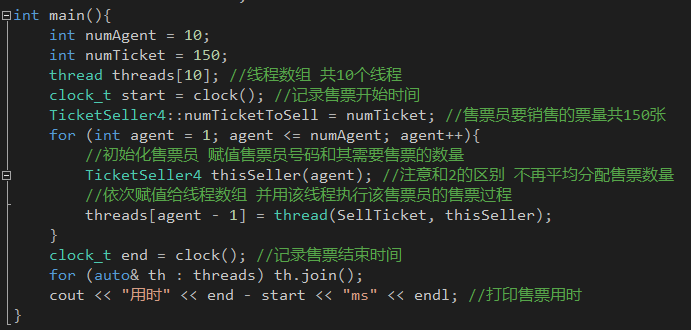
1号售票员其实已经将所有的票售完了，但是由于没有互斥访问numTicketToSell，在其他线程看来，此时的numTicketToSell并不为0，仍有余票可售，于是才会出现以上状况，当某销售员售完票后，还有其他售原员售票的行为。

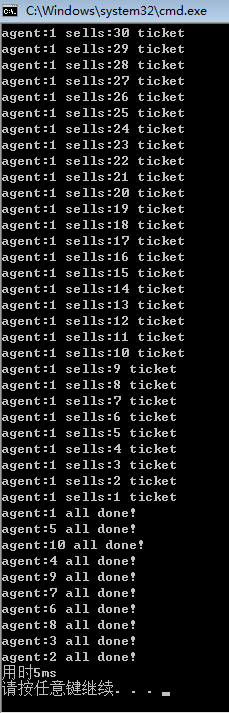
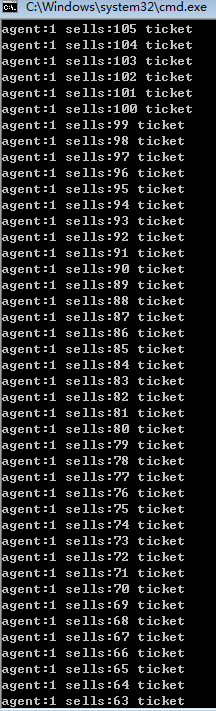
**1.2.4 TicketSeller4**

代码及运行结果如下



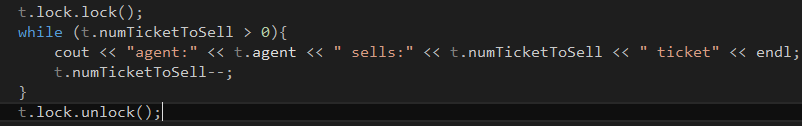






代码分析：

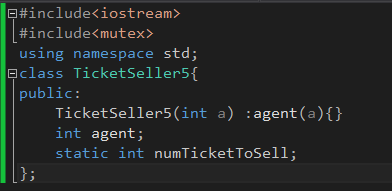
此样例在类中加入了一个公有静态变量mutex lock，使用互斥锁mutex类来保证对numTicketToSell的互斥访问，所以解决了TicketSeller3中的多个线程同时访问该变量，同时售出同一张票的问题，且保证了每时每刻对于任何线程来说numTicketToSell的值都是一致的。但此代码仍然存在问题，1是互斥锁的位置存在问题，在图中：

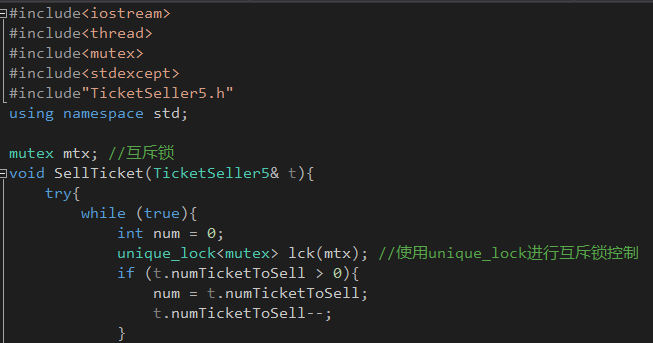


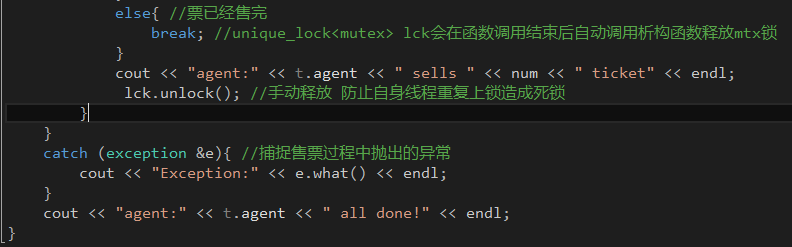
此位置进行上锁，会使得一个线程在票尚未售完之前持续地占有mutex锁，导致其他线程在锁释放之前无法进行售票。最终结果是，只有一个线程进行了售票过程，其他线程都处于等待状态，任何一张票都没有售出，正如上图结果显示，1号售票员进行售票，其他售票员只能等待直至售票结束，造成了资源的极大浪费；2是使用mutex类手动的上锁解锁，一旦在某个线程执行售票的过程中发生了异常，会导致mutex锁再也无法释放，最终导致其他售票员也无法进行售票的后果；

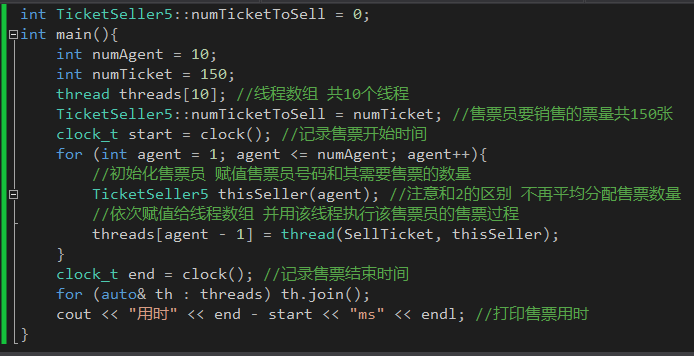
**1.2.5 TicketSeller5**

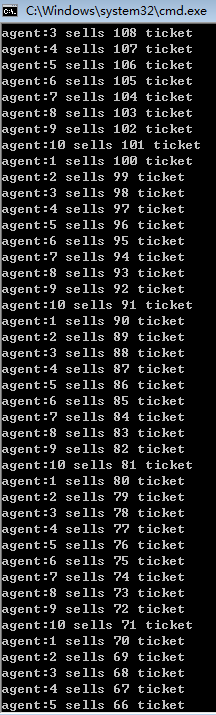
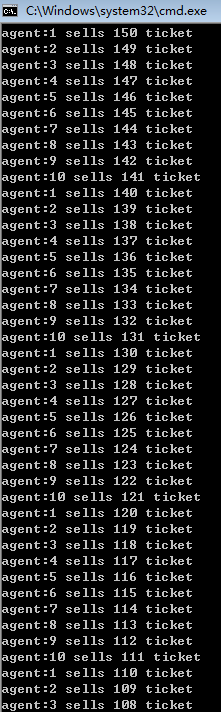
代码及运行结果如下

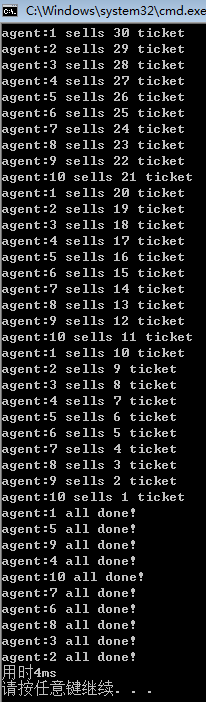
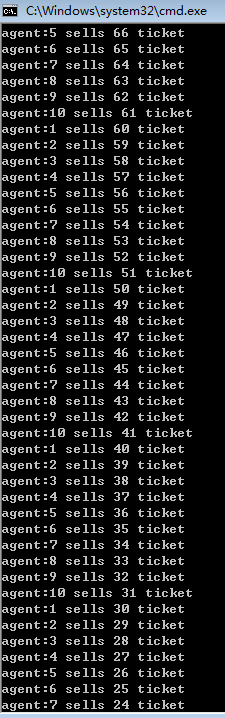












代码分析：

在本样例中，本人使用了C++多线程库中的unique\_lock类来管理互斥量mutex，解决了TicketSeller4中的问题：1、调整了互斥锁的上锁位置和循环逻辑，使用if而不是while判断是否尚有余票，有则售票，无则退出循环，故能实现多线程的并发执行；2、是利用了unique\_lock的特性，加入了try-catch语句进行异常捕捉，这样，即使再售票中发生异常也能检测出来，且unique\_lock可以在程序退出之后，自动的调用其析构函数对mutex进行解锁，防止了发生异常后mutex被持续锁住而无法被其他线程利用的问题。此外，阅读unique\_lock析构函数的源码：

~unique\_lock() \_NOEXCEPT { // 析构

if (\_Owns) // 若互斥量在析构之前已经解锁了，由于存在\_Owns标志，不再解锁，所以不会发生析构异常

\_Pmtx->unlock(); // 若没有，则正常析构

}

可知，unique\_lock提供了更大的灵活性，既可以手动上锁解锁，也可以自动调用析构函数解锁，做到了异常安全；正如上图中代码所见：



可以手动释放unique\_lock，防止自身线程重复上锁造成死锁。

**结果分析：**

在运行的结果中，可以看出，线程（售票员）1-10号是以并行的、速度几乎相当的速度进行售票，即实际结果是1号卖150、140、130…10号票；2号卖149、139、129…9号票；…；10号卖141、131、121…1号票，同步执行，极大的提高了效率，耗时仅需4ms。

至此，本人利用C++多线程库中的thread、mutex类，模拟了一个多人售票的过程，实现了多线程并发执行，且保证了线程安全和异常安全。