

**并 行 计 算**

课程实验报告

实验名称：Python并发编程

任课教师：徐悦甡

课程班级：15级 云计算方向

学号姓名：15130130273 石明皓

提交日期：2018年6月27日

**软件学院本科生《并行计算》**

**课程实验报告**

1. **实验名称**

第4次实验：Python并发编程

1. **实验日期**

2018年6月27日 软件学院实验室G346

1. **实验学生**

15130130273 石明皓

1. **实验目的**

本次实验通过展示2个简单的多线程应用场景，编写了实现其业务逻辑的Python多线程程序。学习了通过继承threading.Thread类，然后重载run方法来实现多线程，以及学会了Python多线程锁的简单应用，理解了基本Python多线程编程思想，掌握了Python基础并发程序设计的方法。

1. **实验内容**

题目一：

四个售票窗口同时出售30张电影票。

题目二：

两个人张三与李四，通过一个同一个账户，张三在柜台取钱，李四在 ATM机取钱。

1. **程序思路、结构**

题目一：

电影票的票数使用同一个静态值，不同售票窗口对象操作的均为该静态变量；

为了不出现不同柜台卖出同一张票的情况，要用到Python多线程同步锁，需要使用threading.Lock()创建一个lock对象；

创建售票窗口类BoxOffice继承threading.Thread类，该类构造时使用threading.Thread.\_\_init\_\_(self)实现线程的初始化，同时使用self.setName()赋予售票窗口号，在BoxOffice类中重写run()方法，在run()方法中进行售票操作；

进行售票操作时使用同步锁，即：任意一个窗口正在出售某张票时，其他窗口必须先等待该窗口卖出这张票，完成其完整售票流程。

每个窗口对象的run()方法循环执行，对某张电影票操作前lock.acquire()加锁，操作完成后lock.release()解锁；

创建4个线程模拟4个售票窗口，使用start()方法启动线程。

题目二：

创建一个Bank类（用于存放账户金额，并提供柜台取钱和ATM机取钱两种方法）、一个张三类（代表在柜台取钱）、一个李四类（代表在ATM机取钱）；

Bank类中提供的两种取钱方法，方法内部对账户金额进行操作前有加锁操作self.lock.acquire()，操作后有解锁操作self.lock.release()；

Bank类构造时初始化了账户金额，并使用self.lock = threading.Lock()创建了线程同步锁；

两种取钱方法对同一账户进行取钱操作，故创建同一个静态值作为账户金额；

张三类的构造函数和李四类的构造函数传入的是同一个Bank类的对象，说明它们操作的是同一账户；

张三类和李四类均继承threading.Thread类，重写了其run()方法，循环执行取钱操作；

创建2个线程模拟张三和李四的同时取款，start()启动线程。

1. **程序代码**

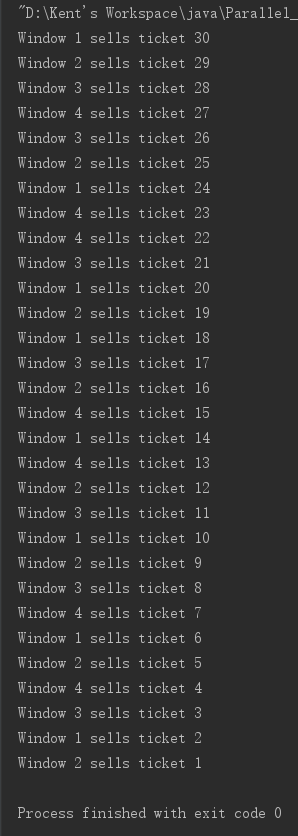
题目一：

import threading  
import time  
  
  
class BoxOffice(threading.Thread):  
 def \_\_init\_\_(self, num):  
 threading.Thread.\_\_init\_\_(self)  
 self.setName('Window ' + str(num)) # 售票窗口号  
  
 def run(self):  
 global ticket  
 global lock  
  
 while ticket > 0:  
 lock.acquire() # 加锁  
 print(self.getName() + ' sells ticket ' + str(ticket))  
 ticket = ticket - 1 # 票数减1  
 lock.release() # 解锁  
 time.sleep(0.1) # 每次售票完毕后休眠100ms  
  
  
if \_\_name\_\_ == '\_\_main\_\_':  
 ticket = 30 # 共30张电影票  
  
 lock = threading.Lock() # 创建锁  
  
 threads = []  
 for i in range(4): # 创建4个线程  
 threads.append(BoxOffice(i + 1))  
  
 for t in threads:  
 t.start()

题目二：

import threading  
import time  
  
class Bank:  
 def \_\_init\_\_(self):  
 self.money = 5000 # 初始5000元  
 self.lock = threading.Lock() # 创建锁  
  
 def counter\_withdraw(self, money1):  
 self.lock.acquire() # 加锁  
 self.money = self.money - money1  
 print('ZhangSan gets $' + str(money1) + ' from counter, and remains $' + str(self.money) + ' in the account')  
 self.lock.release() # 解锁  
  
 def atm\_withdraw(self, money2):  
 self.lock.acquire() # 加锁  
 self.money = self.money - money2  
 print('LiSi gets $' + str(money2) + ' from ATM, and remains $' + str(self.money) + ' in the account')  
 self.lock.release() # 解锁  
  
class ZhangSan(threading.Thread):  
 def \_\_init\_\_(self, same\_account): # 初始化为同一账户  
 threading.Thread.\_\_init\_\_(self)  
 self.bank = same\_account  
  
 def run(self):  
 while self.bank.money >= 100:  
 self.bank.counter\_withdraw(100) # 柜台取钱100  
 time.sleep(0.05)  
  
class LiSi(threading.Thread):  
 def \_\_init\_\_(self, same\_account): # 初始化为同一账户  
 threading.Thread.\_\_init\_\_(self)  
 self.bank = same\_account  
  
 def run(self):  
 while self.bank.money >= 300:  
 self.bank.atm\_withdraw(300) # ATM取钱300  
 time.sleep(0.1)  
  
  
if \_\_name\_\_ == '\_\_main\_\_':  
 account = Bank()  
  
 p1 = ZhangSan(account)  
 p2 = LiSi(account)  
 p1.start()  
 p2.start()

1. **实验结果**

题目一：

从程序运行结果可知，4个售票窗口线程同时卖票，不同窗口卖出不同的票，没有出现卖出同一张票的情况。

题目二：



从程序运行结果可知，张三和李四同时调用Bank类的不同方法对同一账户进行取款，由于取款过程有同步锁，无法多个线程同时对账户金额进行更改操作，因此没有出现账户金额的一致性问题。

1. **总结建议**

经过本次实验，我对并发程序设计的原理加深了认识；对并发程序设计时需要注意的同步问题、一致性问题（如threading.Lock()创建锁）有了进一步了解；同时更加熟悉Python语言在多线程编程方面的编程模式（如继承threading.Thread并重写run()方法），进而提高了个人的Python编程能力。