操作系统实验一实验报告

银行柜员服务问题

张子扬 无06 2020010790

1. 实验目的

1. 通过对进程间通信同步/互斥问题的编程实现，加深理解信号量和P、V操作的原理；

2. 对Windows或Linux涉及的几种互斥、同步机制有更进一步的了解；

3. 熟悉Windows或Linux中定义的与互斥、同步有关的函数。

二、 实验平台

本实验在Python3.10.0上运行。

三、 实验原理

实验采用信号量（semaphore）的P、V操作，实现不同线程间的同步和资源访问的互斥。其中，用于实现互斥的二元信号量采用互斥量（mutex）。

Python提供的线程库threading中包含了对信号量和互斥量的封装，实验中直接使用它们进行同步与互斥操作。

四、 算法设计思路

实验采用同步算法的顾客（customer）线程和柜员（teller）线程的伪代码如下：

queue waiting\_customers;

mutex waiting\_customers\_mtx;

semaphore waiting\_customers\_sem = 0;

**class** customer

{

    idx;

    semaphore sem = 0;

};

**def** customer\_action(customer)

{

    enter\_bank();

    P(waiting\_customers\_mtx);

    waiting\_customers.put(customer);

    V(waiting\_customers\_sem);

    V(waiting\_customers\_mtx);

    P(customer.sem);

    leave\_bank();

}

**def** teller\_action(teller)

{

**while** **True** {

        P(waiting\_customers\_sem);

        P(waiting\_customers\_mtx);

        cust = waiting\_customers.get();

        V(waiting\_customers\_mtx);

        serve(cust);

        V(cust.sem);

    }

}

五、 算法实现

class Customer:

def \_\_init\_\_(self, id, arrival\_time, service\_time):

self.id = id

self.arrival\_time = arrival\_time

self.service\_time = service\_time

self.start\_service\_time = 0

self.leave\_time = 0

self.served\_by = None

self.sem = threading.Semaphore(0)

def customer\_action(customer, waiting\_customers, waiting\_customers\_mtx, waiting\_customers\_sem):

time.sleep(customer.arrival\_time) # 进入银行

waiting\_customers\_mtx.acquire() # P操作

waiting\_customers.put(customer)

waiting\_customers\_mtx.release() # V操作

waiting\_customers\_sem.release()

customer.sem.acquire() # 等待服务完成

# 离开银行，不需要操作

def teller\_action(teller\_id, init\_time, waiting\_customers, waiting\_customers\_mtx, waiting\_customers\_sem):

while True:

waiting\_customers\_sem.acquire() # 等待顾客到达

waiting\_customers\_mtx.acquire()

customer = waiting\_customers.get()

waiting\_customers\_mtx.release()

customer.start\_service\_time = round(time.time() - init\_time)

customer.served\_by = teller\_id

time.sleep(customer.service\_time) # 服务顾客

customer.leave\_time = customer.start\_service\_time + customer.service\_time

customer.sem.release() # 完成服务

print("顾客{}进入银行时间为{}，开始服务时间为{}，离开银行时间为{}，服务柜员号为{}".format(customer.id, customer.arrival\_time, customer.start\_service\_time, customer.leave\_time, customer.served\_by))

六、 运行测例

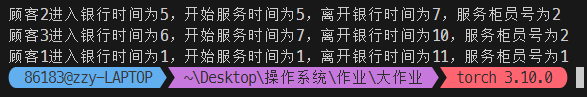
运行的测例默认采用大作业的要求测试数据：

1 1 10

2 5 2

3 6 3

设置银行柜员有2个，输出结果为：



设置银行柜员有3个，输出结果为：



七、 思考题

1. 柜员人数和顾客人数对结果分别有什么影响？

可以首先想象一个理想情况，每分钟有n个顾客到达银行，每个顾客需要被服务m分钟，那么此时需要n\*m个银行柜员则恰好不会有顾客等待，且恰好每个柜员都始终在工作。在这个理想情况的基础上，我们设想以下两种情况：

如果顾客平均到达速率n和平均需要服务时间m不变，柜员人数增加，那么顾客的平均等待时间t会减少，因为有更多的柜员可以为顾客服务。但是，如果柜员人数过多，顾客的平均等待时间t降至0之后就不会再下降，此时会有某些柜员空闲，从而浪费资源。

如果顾客平均到达速率n和平均需要服务时间m增多，而柜员人数不变，那么顾客的平均等待时间t会增加，因为有更多的顾客在等待服务。随着顾客平均到达速率n和平均需要服务时间m增多，顾客的平均等待时间t会一直增长，与n\*m近似成线性关系。

因此，在设计银行柜员服务系统时，需要根据实际情况合理安排柜员人数和顾客人数，以保证银行的运营效率和顾客的满意度。

2. 实现互斥的方法有哪些?各自有什么特点?效率如何?

（1） 禁用中断：适用于单核处理器，在进入临界区之前关闭硬件中断，防止调度。禁用中断效率高，但不安全且不适用于多核处理器。

（2） 忙等待：一些算法例如锁变量、严格轮换法、Petersen算法、硬件指令法，都使用了忙等待的算法。通过不断检查变量的值来实现同步，避免了用户态和内核态切换的开销。然而，忙等待占用CPU资源且可能存在优先级反转问题。

（3） 信号量：使用P、V原语实现同步和互斥，阻塞时不占用CPU。效率较高，但部分实现会引起用户态和内核态转换的开销。

（4） 互斥量：信号量的简化形式，只有二元信号量和同步的P、V操作。相对于信号量，互斥量使用简单且性能更好。

（5） 管程：编程语言提供的特性，通过互斥保证一个线程在管程函数内执行。便利但有些语言不支持和部分实现可能无法在中途阻塞。