Project 1_5
银行柜员服务问题

班级：无 53

姓名：陈相宁

学号：2015011033

Email：c.xiangning1997@gmail.com

⑤ 银行柜员服务问题

问题描述

银行有 n 个柜员负责为顾客服务，顾客进入银行先取一个号码，然后等着叫号。当某个柜员空闲下来，就叫下一个号。

编程实现该问题，用 P、V 操作实现柜员和顾客的同步。

实现要求

1. 某个号码只能由一名顾客取得；
2. 不能有多于一个柜员叫同一个号；
3. 有顾客的时候，柜员才叫号；
4. 无柜员空闲的时候，顾客需要等待；
5. 无顾客的时候，柜员需要等待。

实现提示

1. 互斥对象：顾客拿号，柜员叫号；
2. 同步对象：顾客和柜员；
3. 等待同步对象的队列：等待的顾客，等待的柜员；
4. 所有数据结构在访问时也需要互斥。

测试文本格式

测试文件由若干记录组成，记录的字段用空格分开。记录第一个字段是顾客序号，第二字段为顾客进入银行的时间，第三字段是顾客需要服务的时间。

下面是一个测试数据文件的例子：

1 1 10

2 5 2

3 6 3

输出要求

对于每个顾客需输出进入银行的时间、开始服务的时间、离开银行的时间和服务柜员号。

思考题

1. 柜员人数和顾客人数对结果分别有什么影响？
2. 实现互斥的方法有哪些?各自有什么特点?效率如何?

一、 实验思路：

将每个用户（Customer）以及银行柜员（Clerk）当作独立的线程。因此实验关键在于实现线程之间的同步与互斥。

1. 同步：

   利用 Ticket 自建类实现。有 customer 取号时自动生成一个 Ticket 对象。其中设置一个初值为 0 的 Semaphore 用于判定是否有 clerk 叫到了该号：如果叫到，则执行 V 操作，用户停止等待；否则，用户线程阻塞，继续等待。

2. 互斥：

   互斥主要防止同时有 customer 取号和 clerk 叫号的情况。因此只需设置初值为 1 的 Semaphore mutex 即可。

实现了线程的同步及互斥后，还需要一个仓库来生成、存储 ticket，为 customer 发号并分配给 clerk 任务。

二、 代码实现：

编程语言为 python。利用 threading 库中的 Semaphore 类实现信号量。Semaphorea.acquire()为 P 操作，Semaphore.release()为 V 操作。继承 Thread 类实现 Customer 和 Clerk。利用 time 库中的 time 方法获取当前时间，sleep 方法在需要的时候将线程休眠。

1. Bank 类：

自建 Bank 类存储生成的 ticket。初始化时建立初值为 1 的 mutex 信号量，初值为 0 的 ticket_num 信号量，名为 tickets 的 list 来动态存储号。

Bank 类包含用户 fetch_ticket 以及银行柜员 call_ticket 方法。fetch_ticket 方法中，首先对 mutex 执行 P 操作保证同时只有一个人访问 Bank，然后生成一个 Ticket 对象并加入 tickets list 中，同时对 ticket_num 执行 V 操作，最后执行 mutex V 操作完成一次取票。

call_ticket 方法中，首先对 ticket_num 执行 P 操作保证仍有 customer 未被服务，之后执行 mutex P 操作。之后一步很关键：有两种机制，一种是保证平均服务时间最短，即取 tickets list 中服务时间最短的号；另一种是保证先来后到的情况下尽量缩短总服务时间，即取 tickets list 中最先来的一批 customer 中服务时间最短的。由于后一种较符合常理，因此之后采取第二种叫号方式。最后对 mutex 执行 V 操作完成一次叫号。

```
class Bank(object):
    def __init__(self):
        self.mutex = Semaphore(1)
        self.ticket_num = Semaphore(0)
        self.tickets = []

    def fetch_ticket(self, arrive_time, service_time):
        self.mutex.acquire()
        ticket = Ticket(arrive_time, service_time)
        self.tickets.append(ticket)
        self.ticket_num.release()
        self.mutex.release()
        return ticket

    def call_ticket(self):
        self.ticket_num.acquire()
        self.mutex.acquire()
        self.tickets.sort(key = attrgetter('arrive_time',
            'service_time'), reverse = True)
        ticket = self.tickets.pop()
        self.mutex.release()
        return ticket
```

2. Ticket 类:

为了实现 Bank 类中 call_ticket 方法，Ticket 对象初始化时需输入取票人编号，取票人到来时间以及服务时间。同时设置初值为 0 的 clerk 信号量判断该号是否被 clerk 叫到。

Ticket 类中包含用户 wait_for_call 以及 call 方法。wait_for_call 方法中，首先对 clerk 执行 P 操作，如果有 clerk 叫号则返回该 clerk 的编号。call 方法中，首先更新该 ticket 的 clerk 编号，之后执行 clerk V 操作唤醒等待中的 customer 线程，函数返回服务时间以对当前 clerk 线程休眠。

```
class Ticket(object):
    def __init__(self, arrive_time, service_time):
        self.clerk = Semaphore(0)
        self.arrive_time = arrive_time
        self.service_time = service_time
        self.clerk_number = None

    def wait_for_call(self):
        self.clerk.acquire()        # judge whether a clerk call this ticket
        return self.clerk_number

    def call(self, clerk_number):
        self.clerk_number = clerk_number
        self.clerk.release()
        return self.service_time
```

3. Customer 线程：

继承 Thread 类，初始化时输入 customer 编号、到达时间以及服务时间。

run 方法中，先 sleep 一段时间保证按时到达，到达后调用 fetch_ticket 方法从全局 Bank 对象中取 ticket。之后开始等待，某一 clerk 叫号该 ticket 唤醒当前 customer 线程后，再 sleep 一定时间表示正在服务中，最后线程退出并输出结果。

```
class Customer(Thread):
    def __init__(self, customer_number, arrive_time, service_time):
        super().__init__()
        self.customer_number = customer_number
        self.arrive_time = arrive_time
        self.service_time = service_time

    def run(self):
        global START, bank
        sleep(self.arrive_time)
        ticket = bank.fetch_ticket(self.arrive_time, self.service_time)
        clerk_number = ticket.wait_for_call()
        service_begin_time = int(time() - START)
        sleep(self.service_time)
        leave_time = int(time() - START)
        print_mutex.acquire()
        print('customer_number:', self.customer_number, 'arrive_time:',
            self.arrive_time, 'service_begin_time:', service_begin_time,
            'leave_time:', leave_time, 'clerk_number:', clerk_number)
        print_mutex.release()
        return
```

4. Clerk 线程：

继承 Thread 类，初始化时输入 clerk 编号，最大工作时间（最大工作时间保证该线程工作一定时间后能够退出）。

run 方法中，执行一无限循环，循环中先调用 call_ticket 方法从全局 Bank 中取 ticket，取到 ticket 后对该 ticket 执行 call 操作唤醒持有该 ticket 的 customer 并获得 service_time，然后 sleep 相应时间完成一次 service。

```
class Clerk(Thread):
    def __init__(self, clerk_number, work_time):
        super().__init__()
        self.clerk_number = clerk_number
        self.work_time = work_time

    def run(self):
        global START, bank
        while True:
            if time() - START > self.work_time:
                break
            ticket = bank.call_ticket()
            service_time = ticket.call(self.clerk_number)
            sleep(service_time)
```

全部代码：

Bank.py：

```python
from threading import Semaphore
from operator import attrgetter

class Bank(object):
    def __init__(self):
        self.mutex = Semaphore(1)
        self.ticket_num = Semaphore(0)
        self.tickets = []

    def fetch_ticket(self, arrive_time, service_time):
        self.mutex.acquire()
        ticket = Ticket(arrive_time, service_time)
        self.tickets.append(ticket)
        self.ticket_num.release()
        self.mutex.release()
        return ticket

    def call_ticket(self):
        self.ticket_num.acquire()
        self.mutex.acquire()
        self.tickets.sort(key = attrgetter('arrive_time',
            'service_time'), reverse = True)
        # self.tickets.sort(key = attrgetter('service_time'),
            # reverse = True)
        ticket = self.tickets.pop()
        self.mutex.release()
        return ticket

class Ticket(object):
    def __init__(self, arrive_time, service_time):
        self.clerk = Semaphore(0)
        self.arrive_time = arrive_time
        self.service_time = service_time
        self.clerk_number = None

    def wait_for_call(self):
        self.clerk.acquire()        # judge whether a clerk call this ticket
        return self.clerk_number

    def call(self, clerk_number):
        self.clerk_number = clerk_number
        self.clerk.release()
        return self.service_time


```

People_Thread.py:

```python
from threading import Thread, Semaphore
from time import time, sleep
from Bank import *

START = time()
bank = Bank()
print_mutex = Semaphore(1)

class Customer(Thread):
    def __init__(self, customer_number, arrive_time, service_time):
        super().__init__()
        self.customer_number = customer_number
        self.arrive_time = arrive_time
        self.service_time = service_time

    def run(self):
        global START, bank
        sleep(self.arrive_time)
        ticket = bank.fetch_ticket(self.arrive_time, self.service_time)
        clerk_number = ticket.wait_for_call()
        service_begin_time = int(time() - START)
        sleep(self.service_time)
        leave_time = int(time() - START)
        print_mutex.acquire()
        print('customer_number:', self.customer_number, 'arrive_time:',
            self.arrive_time, 'service_begin_time:', service_begin_time,
            'leave_time:', leave_time, 'clerk_number:', clerk_number)
        print_mutex.release()
        return

class Clerk(Thread):
    def __init__(self, clerk_number, work_time):
        super().__init__()
        self.clerk_number = clerk_number
        self.work_time = work_time

    def run(self):
        global START, bank
        while True:
            if time() - START > self.work_time:
                break
            ticket = bank.call_ticket()
            service_time = ticket.call(self.clerk_number)
            sleep(service_time)
```

```
def load_data(file_path):
    customers = []
    with open(file_path, 'r') as f:
        for line in f:
            customer_number, arrive_time, service_time = line.split()
            customers.append(Customer(customer_number, int(arrive_time),
                int(service_time)))
    return customers

def generate_clerk(max_clerk, work_time = 1e2):
    clerks = []
    for i in range(max_clerk):
        clerks.append(Clerk(i + 1, work_time))
    return clerks

file_path = 'input2.txt'
customers = load_data(file_path)
clerks = generate_clerk(max_clerk = 4)

for customer in customers:
    customer.start()
for clerk in clerks:
    clerk.start()
```

测试结果：

1. 简单例子：

输入：

1 1 10

2 5 2

3 6 3

输出：

1 个 clerk：

```
customer_number: 1 arrive_time: 1 service_begin_time: 1 leave_time: 11 clerk_number: 1
customer_number: 2 arrive_time: 5 service_begin_time: 11 leave_time: 13 clerk_number: 1
customer_number: 3 arrive_time: 6 service_begin_time: 13 leave_time: 16 clerk_number: 1
```

2 个 clerk：

```
customer_number: 2 arrive_time: 5 service_begin_time: 5 leave_time: 7 clerk_number: 2
customer_number: 3 arrive_time: 6 service_begin_time: 7 leave_time: 10 clerk_number: 2
customer_number: 1 arrive_time: 1 service_begin_time: 1 leave_time: 11 clerk_number: 1
```

5 个 clerk：

```
customer_number: 2 arrive_time: 5 service_begin_time: 5 leave_time: 7 clerk_number: 2
customer_number: 3 arrive_time: 6 service_begin_time: 6 leave_time: 9 clerk_number: 3
customer_number: 1 arrive_time: 1 service_begin_time: 1 leave_time: 11 clerk_number: 1
```

如果只有 1 个 clerk，则总服务时间为 10 + 2 + 3 = 15。

如果有 2 个 clerk，则 1 号 clerk 服务 1 号 customer，11 时结束。2 号 clerk 先服务 2 号 clerk，7 时结束。之后服务 3 号 customer，10 时结束。

如果有 3 个以上 clerk，则每个 customer 都能立即被服务。

综上，程序通过小数据测试。

2. 复杂例子：

输入：

1 1 10

2 5 2

3 6 3

4 6 5

5 3 8

6 7 1

7 10 5

8 9 7

9 2 8

10 8 2

输出：

1 个 clerk：

```
customer_number: 1 arrive_time: 1 service_begin_time: 1 leave_time: 11 clerk_number: 1
customer_number: 9 arrive_time: 2 service_begin_time: 11 leave_time: 19 clerk_number: 1
customer_number: 5 arrive_time: 3 service_begin_time: 19 leave_time: 27 clerk_number: 1
customer_number: 2 arrive_time: 5 service_begin_time: 27 leave_time: 29 clerk_number: 1
customer_number: 3 arrive_time: 6 service_begin_time: 29 leave_time: 32 clerk_number: 1
customer_number: 4 arrive_time: 6 service_begin_time: 32 leave_time: 37 clerk_number: 1
customer_number: 6 arrive_time: 7 service_begin_time: 37 leave_time: 38 clerk_number: 1
customer_number: 10 arrive_time: 8 service_begin_time: 38 leave_time: 40 clerk_number: 1
customer_number: 8 arrive_time: 9 service_begin_time: 40 leave_time: 47 clerk_number: 1
customer_number: 7 arrive_time: 10 service_begin_time: 47 leave_time: 52 clerk_number: 1
```

4 个 clerk（先到先服务）：

```
customer_number: 2 arrive_time: 5 service_begin_time: 5 leave_time: 7 clerk_number: 4
customer_number: 9 arrive_time: 2 service_begin_time: 2 leave_time: 10 clerk_number: 2
customer_number: 3 arrive_time: 6 service_begin_time: 7 leave_time: 10 clerk_number: 4
customer_number: 5 arrive_time: 3 service_begin_time: 3 leave_time: 11 clerk_number: 3
customer_number: 1 arrive_time: 1 service_begin_time: 1 leave_time: 11 clerk_number: 1
customer_number: 6 arrive_time: 7 service_begin_time: 10 leave_time: 11 clerk_number: 4
customer_number: 10 arrive_time: 8 service_begin_time: 11 leave_time: 13 clerk_number: 3
customer_number: 4 arrive_time: 6 service_begin_time: 10 leave_time: 15 clerk_number: 2
customer_number: 7 arrive_time: 10 service_begin_time: 11 leave_time: 16 clerk_number: 4
customer_number: 8 arrive_time: 9 service_begin_time: 11 leave_time: 18 clerk_number: 1
```

4 个 clerk（平均服务时间最短）：

```
customer_number: 2 arrive_time: 5 service_begin_time: 5 leave_time: 7 clerk_number: 4
customer_number: 6 arrive_time: 7 service_begin_time: 7 leave_time: 8 clerk_number: 4
customer_number: 9 arrive_time: 2 service_begin_time: 2 leave_time: 10 clerk_number: 2
customer_number: 10 arrive_time: 8 service_begin_time: 8 leave_time: 10 clerk_number: 4
customer_number: 5 arrive_time: 3 service_begin_time: 3 leave_time: 11 clerk_number: 3
customer_number: 1 arrive_time: 1 service_begin_time: 1 leave_time: 11 clerk_number: 1
customer_number: 3 arrive_time: 6 service_begin_time: 11 leave_time: 14 clerk_number: 1
customer_number: 7 arrive_time: 10 service_begin_time: 10 leave_time: 15 clerk_number: 2
customer_number: 4 arrive_time: 6 service_begin_time: 11 leave_time: 16 clerk_number: 3
customer_number: 8 arrive_time: 9 service_begin_time: 10 leave_time: 17 clerk_number: 4
```

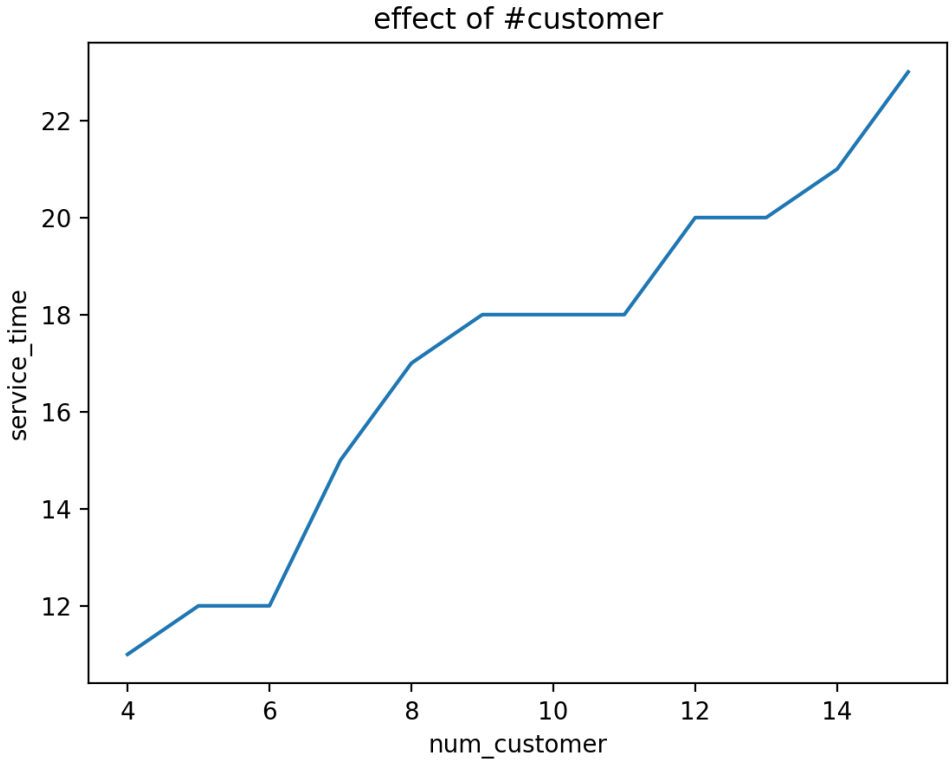
可以看出，该多线程程序运行结果符合预期。先到先服务以及平均服务时间最短方式各有其合理性，应视具体情况而定。(之后测试采取先到先服务模式)

三、 思考题:

1. 柜员人数和顾客人数对结果分别有什么影响:

①顾客人数影响:

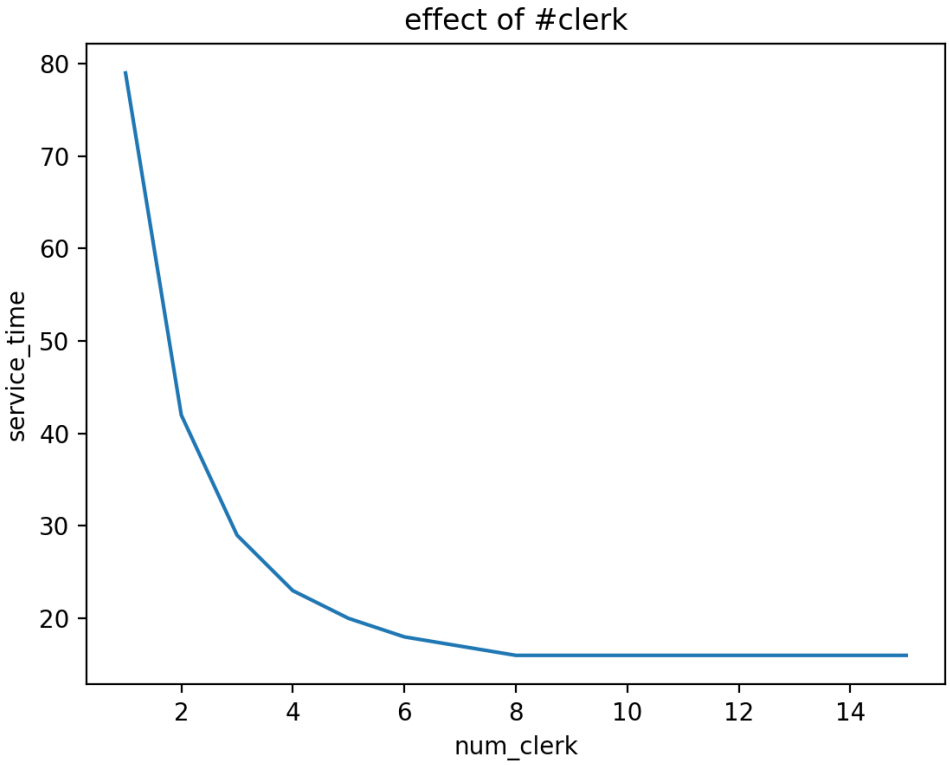
固定柜员人数为 4。



结合图像发现，总服务时间随 customer 数量上升近似线性增长。但具体时间跟每个用户到达时间跟服务时间有关，较为复杂。

②柜台人数影响：

固定顾客人数为 10。



结合图像发现，总服务时间一开始随柜台人数增加而减少，但减缓速度越来越慢，最终趋于稳定值。此时，再增加柜员人数对总服务时间无提升。曲线形状类似负指数函数。

2. 实现互斥的方法有哪些？各自有什么特点？效率如何？

方法	特点	种类	效率
禁止中断	简单。但是把中断权力交给用户导致可靠性较差。且不适用于多处理器	禁止中断	高
自旋锁	忙等待，浪费 CPU 时间，只有等待时间非常短时使用	忙等待	较低，且可能带来优先级反转问题
严格轮转法	要求两个进程严格轮流进入临界区。忙等待问题		
Peterson 算法	严格轮转法的优化，可以正常工作。但仍存在忙等待问题		
硬件指令方法	适用于任意数目进程。简单，容易验证正确性可支持进程中存在多个临界区。但仍有忙等待问题		
信号量	可实现进程同步，但信号量的控制分布在整个程序中，很难分析其正确性：同步分散操作、易读性差、不利于修改和维护、正确性难以保证	信号量	较高
管程	信号量的优化，提高代码可读性，便于修改和维护，正确性易于保证。模块化、抽象数据类型、信息封装。但 C 及多数语言不支持管程	管程	较高
消息传递	适用于不同机器之间的进程通信	消息传递	较高