实验一 进程间同步/互斥问题

无 16 李根 2021012787

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

实现方法

首先我先定义了两个类，用于记录顾客与柜员的相关信息：

```
class Customer:
    def __init__(self, customer_id, arrival_time, service_time):
        self.customer_id = customer_id
        self.arrival_time = arrival_time
        self.service_time = service_time
        self.begin_service_time = None
        self.end_service_time = None
        self.server_id = None
2 usages
class Server:
    def __init__(self, server_id):
        self.server_id = server_id
        self.working = None
```

其中顾客类需要记录顾客的 id，到达时间，离开时间，服务时间，以及实际的开始服务，结束服务时间以及服务员的 id；柜员则只需要记录 id 与是否在工作即可。

然后定义了顾客信号量与柜员信号量，并以此设计了互斥锁与时间调用锁。通过 global 变量来进行时间流逝的模拟：

```
#信号量
customer_semaphore = threading.Semaphore(0)
server_semaphore = threading.Semaphore(0)
#锁
mutex = threading.Lock()
time_lock = threading.Lock()
#全局时间
global_time = 0
```

本实验中的两个主要事件便是顾客到达与柜员服务。因此我们对这两个主要时间设计函数。

对于柜员服务，我们的函数定义如下：

```
def server_work(server,customer_quene):
    with mutex:
        if not customer_quene.empty():
            customer = customer_quene.get()
    server.working = True
    with time_lock:
        customer.begin_service_time = global_time
    customer.server_id = server.server_id
    service_end_time = global_time + customer.service_time

    while global_time < service_end_time:
        pass

    with time_lock:
        customer.end_service_time = global_time

    server.working = False
    print(f"顾客 {customer.customer_id} 到达时间： {customer.arrival_time}, "
          f"开始服务时间： {customer.begin_service_time}, 离开时间： {customer.end_service_time}, "
          f"服务人员编号： {server.server_id}")
    server_semaphore.release()
```

我们采用了 with mutex 与 with time_lock 这两个语法用于控管理进程的互斥与时间的同步，然后记录当前柜员的结束服务时间。在服务结束时将 server.working 置为 False，同时 release 信号量 server_semaphore

对于用户到达，我们的函数定义如下：

```
def customer_arrival(customer,customer_quene):
    global global_time
    while global_time < customer.arrival_time:
        pass
    with mutex:
        customer_quene.put(customer)
    customer_semaphore.release()
    server_semaphore.acquire()
```

相比较上面的柜员服务要简单一些，因为此处的用户到达，即使有 mutex 我们也只需要将其放入 customer 的等待队列当中即可。

定义好了上面的辅助函数，我们的银行的实际工作函数 bank_work 定义如

下：

```
customer_quene = Queue()

server_threads = []
customer_threads = []

for server in servers:
    server_thread = threading.Thread(target=server_work, args=(server, customer_quene))
    server_thread.start()
    server_threads.append(server_thread)

for customer in customers:
    customer_thread = threading.Thread(target=customer_arrival, args=(customer, customer_quene))
    customer_thread.start()
    customer_threads.append(customer_thread)

global global_time

total_time = max([customer.arrival_time + customer.service_time for customer in customers])
```

先声明好用户与柜员的线程数组，然后对每一个柜员与顾客采用 threading.Thread()与 start 方法来启动线程。

在最大时间范围内，我们采用 for 循环和 threading.Event().wait()来模拟时间变化与进程同步：

```
for time in range(total_time+1):
    with time_lock:
        global_time = time
    threading.Event().wait(0.1)
```

最后只需要将上面定义好的线程数组 join()即可等待并完成运行：

```
for customer_thread in customer_threads:
    customer_thread.join()
for server_thread in server_threads:
    server_thread.join()
```

至于读取文件的函数此处就不多赘述了。

运行结果与说明

我们先直接使用文档中提到的实例，并初始化两个 server：

```
D:\anaconda3\python.exe C:\Users\LiGen\Desktop\os\bank.py
顾客 2 到达时间： 5, 开始服务时间： 5, 离开时间： 7, 服务人员编号： 2
顾客 3 到达时间： 6, 开始服务时间： 7, 离开时间： 10, 服务人员编号： 2
顾客 1 到达时间： 1, 开始服务时间： 1, 离开时间： 11, 服务人员编号： 1
```

这与我们的预期是完全相符合的。

思考题

1.柜员人数和顾客人数对结果分别有什么影响?

柜员人数越多,总的结束时间越短,相应的发生互斥 mutex 的几率就会越小;顾客人数越多，则越容易触发互斥锁与时间锁，总的用时也相对会更长

2.实现互斥的方法有哪些?各自有什么特点?效率如何?

实现互斥的方法主要有忙等待，锁，信号量，条件变量，监视器与原子操作等。我在实现本次实验的任务时，主要考虑了锁，信号量，为了实现时间流逝与时间锁，我也使用了条件变量。具体说明如下:

锁（Locks）：通过锁机制来保护共享资源，避免多个线程同时访问。效率在高竞争环境下一般，但对于我们的测试样例已经足够了

信号量（Semaphores）：信号量被用作一个计数器，表示可用资源的数量。其可以用于多个线程或进程之间的同步,比互斥锁更灵活。效率上也比锁更高效,因为它可以支持多个线程同时访问资源

条件变量（Condition Variables）：允许线程在某些条件不满足时挂起，并在条件满足时被唤醒。我们是与互斥锁一起使用（time_lock 等锁），用于线程间的同步。条件变量允许线程在等待条件满足时释放 CPU 资源，效率较高。

至于其他的操作，我们没有在本次实验中使用，也就不多做赘述了。

心得与体会

必须承认操作系统的代码对我来说的确陌生，因为我对 windows 或 linux 的具体实现代码，函数等都缺乏了解，因此我最终选择了 python 的方法，经过助教的允许，通过调用 threading 包的形式来实现了本次的大作业。在实现过程中，我发现 threading 库有时并不算好用，因为我想要实现让程序运行完成后自动停止时，总会发现函数会卡在处于等待状态的柜员进程中，如果不考虑使用计数，在别的地方添加代码总会导致程序错误，可见代码本身的 frigile。

完整代码

```
import threading
from queue import Queue

class Customer:
    def __init__(self, customer_id, arrival_time,
service_time):
        self.customer_id = customer_id
        self.arrival_time = arrival_time
        self.service_time = service_time
        self.begin_service_time = None
        self.end_service_time = None
        self.server_id = None
```

```
class Server:
    def __init__(self, server_id):
        self.server_id = server_id
        self.working = None

#信号量
customer_semaphore = threading.Semaphore(0)
server_semaphore = threading.Semaphore(0)
#锁
mutex = threading.Lock()
time_lock = threading.Lock()
#全局时间
global_time = 0
def server_work(server,customer_quene):
    global global_time
    while True:
        customer_semaphore.acquire()
        with mutex:
            if not customer_quene.empty():
                customer = customer_quene.get()
        server.working = True
        with time_lock:
            customer.begin_service_time = global_time
        customer.server_id = server.server_id
        service_end_time = global_time + customer.service_time

        while global_time < service_end_time:
            pass

        with time_lock:
            customer.end_service_time = global_time

        server.working = False
        print(f"顾客 {customer.customer_id} 到达时间:
{customer.arrival_time}, "
              f"开始服务时间: {customer.begin_service_time}, 离开
时间: {customer.end_service_time}, "
              f"服务人员编号: {server.server_id}")
        server_semaphore.release()

def customer_arrival(customer,customer_quene):
    global global_time
    while global_time < customer.arrival_time:
        pass
```

```
    with mutex:
        customer_quene.put(customer)
    customer_semaphore.release()
    server_semaphore.acquire()

def bank_work(servers,customers):
    customer_quene = Queue()

    server_threads = []
    customer_threads = []

    for server in servers:
        server_thread = threading.Thread(target=server_work,
args=(server, customer_quene))
        server_thread.start()
        server_threads.append(server_thread)

    for customer in customers:
        customer_thread =
threading.Thread(target=customer_arrival, args=(customer,
customer_quene))
        customer_thread.start()
        customer_threads.append(customer_thread)

    global global_time

    total_time = max([customer.arrival_time +
customer.service_time for customer in customers])

    for time in range(total_time+1):
        with time_lock:
            global_time = time
        threading.Event().wait(0.1)
        '''

print('time:',time,'total_time:',global_time,time==total_time)
            if time == total_time:
                return 0
        '''

    for customer_thread in customer_threads:
        customer_thread.join()
    for server_thread in server_threads:
        server_thread.join()
```

```
def read_customers_from_file(filename):
    customers = []
    with open(filename, 'r') as file:
        for line in file:
            parts = line.split()
            customer_id = int(parts[0])
            arrival_time = int(parts[1])
            service_time = int(parts[2])
            customers.append(Customer(customer_id, arrival_time,
service_time))
    return customers

customers = read_customers_from_file('test_data.txt')
servers = [Server(1), Server(2)]
bank_work(servers, customers)
```