作业4-进程同步

写者问题（写者优先）

- 1. 读者写者问题要实现以下要求：
  - 读写互斥
  - 读共享，写互斥
  - 优先写

有写者来时，无论当时是在读，还是在写，都要能对这个写者优先做出响应。

读者写者算法-写者优先

信号量定义

readSwitch = Lightswitch()
writeSwitch = Lightswitch()
noReaders = Semaphore(1)
noWriters = Semaphore(1)

Lightswitch 是一种并发控制结构，用来确保：
第一个读者获取对共享资源的访问权（加锁），最后一个读者释放资源（解锁）
典型的实现中，它内部会统计当前读者数量 count，并结合一个共享的锁进行控制

最初信号量都是解锁态。

readSwitch 是reader在进入临界区时，用来给noWriter上锁
最后一个读者离开时，会唤醒noWriter上排除的写者

writeSwitch 是writer到来时，用来给noreader上锁，让后续读者在noreader排队等待

读者写者算法-写者优先

Reader：

```
noReaders.wait()
        readSwitch.lock(noWriters)
noReaders.signal()
        # critical section for readers

readSwitch.unlock(noWriters)
```

Writer：

```
writeSwitch.lock(noReaders)
        noWriters.wait()
            # critical section for writers
        noWriters.signal()
writeSwitch.unlock(noReaders)
```

如果reader在临界区，会给noWriter上锁。但是不会给noReader上锁。

如果这时候writer到来，会给noReader加锁，会让后续读者排队在noReader。

当最后一个读者离开，他会signal noWriter，这时写者可以进入。

读者写者算法-写者优先

信号量定义

readSwitch = Lightswitch()
writeSwitch = Lightswitch()
noReaders = Semaphore(1)
noWriters = Semaphore(1)

Reader：

```
noReaders.wait()
        readSwitch.lock(noWriters)
noReaders.signal()
        # critical section for readers
readSwitch.unlock(noWriters)
```

Writer：

```
writeSwitch.lock(noReaders)
        noWriters.wait()
            # critical section for writers
        noWriters.signal()
writeSwitch.unlock(noReaders)
```

当写者进入临界区，他同时拿着noreader和nowriter两个锁。一方面，其他读者和写者不能同时访问临界区。
另一方面，writeSwitch 允许其他写者通过，并在noWriter等待。但是读者只能在noReader等待。

这样，所有排队的写者能够通过临界区，当最后一个写者离开，noreader才解锁。读者才能进入。

读者写者算法-写者优先

Reader:

```
noReaders.wait()
        readSwitch.lock(noWriters)
noReaders.signal()
        # critical section for readers

readSwitch.unlock(noWriters)
```

信号量定义

```
readSwitch = Lightswitch()
writeSwitch = Lightswitch()
noReaders = Semaphore(1)

noWriters = Semaphore(1)
```

当然，这个算法下，读者可能被饿死

Writer:

```
writeSwitch.lock(noReaders)
        noWriters.wait()
            # critical section for writers
        noWriters.signal()
writeSwitch.unlock(noReaders)
```

2. 寿司店问题

假设一个寿司店有5个座位，如果你到达的时候有一个空座位，你可以立刻就坐。但是如果你到达的时候5个座位都是满的有人已经就坐，这就意味着这些人都是一起来吃饭的，那么你需要等待所有的人一起离开才能就坐。编写同步原语，实现这个场景的约束。

寿司店问题

- eating 和 waiting 记录在寿司店就餐和等待的线程。mutex对他们进行保护。
- must_wait 表示寿司店现在是满的，新来的顾客必须等待

```
eating = waiting = 0        //就餐和等待的顾客数
mutex = Semaphore (1)
block = Semaphore (0)     //等待队列
must_wait = False
```

寿司店问题

eating = waiting = 0
mutex = Semaphore (1)
block = Semaphore (0)
must_wait = False

```
mutex.wait()
if must_wait: //需要等待所有人离开
    waiting += 1
     mutex.signal()
     block.wait() //在block上睡眠
else: //可以直接吃，如果就坐后满了，后续需要等待
    eating += 1
    must_wait = (eating == 5) //更新must_wait
    mutex.signal()
# eat sushi
mutex.wait()
eating -= 1 //吃完了
if eating == 0:  //如果最后一个顾客，可唤醒block上等待的顾客
     n = min(5, waiting) //最多5个顾客可以吃
     waiting -= n
     eating += n
     must_wait = (eating == 5) //reset must_wait
     block.signal(n) //唤醒n个顾客
mutex.signal()
```

一个等待顾客需要请求mutex的唯一原因就是更新eating和waiting的状态。所以解决这个问题的办法是让离开的顾客进行更新，因为他们拥有mutex。当最后离开的顾客释放了mutex，eating已经更新过，所以新到来的顾客能看到正确的状态，并根据需要阻塞。这个模式也叫"我帮你做"，因为离开线程做了逻辑上属于等待线程的工作。

```
if must_wait: //需要等待所有人离开
    waiting += 1
     mutex.signal()
     block.wait() //等待block
else: //可以直接吃，如果就坐后满了，后续需要等待
    eating += 1
    must_wait = (eating == 5)
    mutex.signal()
# eat sushi
mutex.wait()
eating -= 1 //吃完了
if eating == 0:  //如果最后一个顾客，可唤醒block上等待的顾客
     n = min(5, waiting) //最多5个顾客可以吃
     waiting -= n
     eating += n
     must_wait = (eating == 5) //reset must_wait
     block.signal(n) //唤醒n个顾客
mutex.signal()
```

3 进门问题

1）请给出P、V操作和信号量的物理意义。
2）一个软件公司有5名员工，每人刷卡上班。员工刷卡后需要等待，直到所有员工都刷卡后才能进入公司。为了避免拥挤，公司要求员工一个一个通过大门。所有员工都进入后，最后进入的员工负责关门。请用P、V操作实现员工之间的同步关系。

3 进门问题

答:

1. P操作申请资源，V操作释放资源；信号量大于0时，表示资源的个数；信号量小于0时，绝对值表示等待进程的个数。

2.

信号量和变量定义:

n = 5 //the number of 员工

count = 0 //到达汇合点的线程数

mutex = Semaphore(1) //保护count

barrier = Semaphore(0)//线程到达之前都是0或者负值。到达后取正值

3 进门问题

```
 第i个员工（）{
P(mutex);
count = count + 1; //相当于刷卡
if (count == n)
    V(barrier) ;# 唤醒一个线程，
V(mutex);

P(barrier)；//排队等待&轮流通过
V(barrier)；# 一旦线程被唤醒，有责任唤醒下一个线程

P(mutex);
count = count -1;
if (count == 0)
    P(barrier) ; #置为0，关门
V(mutex);
}
```

4. 搜索插入删除问题

三个线程对一个单链表进行并发的访问，分别进行搜索、插入和删除。

- 搜索线程仅仅读取链表，因此多个搜索线程可以并发。
- 插入线程把数据项插入到链表最后的位置；多个插入线程必须互斥防止同时执行插入操作. 一个插入线程可以和多个搜索线程并发执行。
- 最后，删除线程可以从链表中任何一个位置删除数据。一次只能有一个删除线程执行；删除线程之间，删除线程和搜索线程，删除线程和插入线程都不能同时执行。

请编写三类线程的同步互斥代码，描述这种三路的分类互斥问题。

4. 搜索插入删除问题

Lightswitch 实现

信号量定义：
insertMutex = Semaphore(1)
noSearcher = Semaphore(1)
noInserter = Semaphore(1)
searchSwitch = Lightswitch()
insertSwitch = Lightswitch()

4.搜索插入删除问题

搜索者：
searchSwitch.lock(noSearcher)//对noSearcher上锁
# critical section //多个searcher可以进入
searchSwitch.unlock(noSearcher)

插入者：
insertSwitch.lock(noInserter)
insertMutex.wait() //多个inserter互斥访问
# critical section
insertMutex.signal()
insertSwitch.unlock(noInserter)

Lightswitch实现

删除者：
noSearcher.wait() //与searcher互斥
noInserter.wait() //与inserter互斥
# critical section //一次只能有一个删除线程执行
noInserter.signal()
noSearcher.signal()

4. 搜索插入删除问题

信号量定义：

insertMutex = Semaphore(1) //inserter之间的互斥访问
noSearcher = Semaphore(1) //=1表示没有searcher访问
noInserter = Semaphore(1) //=1表示没有inserter访问
//deleter拿到上面两个信号量才能访问临界区

searcher = 0; //searcher个数
//对searcher计数器互斥访问
searcherMutex = Semaphore(1)
inserter = 0; //inserter个数
//对inserter计数器互斥访问
inserterMutex=Semaphore(1)

信号量实现

4.搜索插入删除问题

信号量实现

搜索者：

```
P(searcherMutex)
  searcher++;
  if searcher == 1: //第一个搜索者加锁
     P(noSearcher) //和deleter互斥
V(searcherMutex)

   # critical section //多个searcher无需互斥

P(searcherMutex)
  searcher--;
  if searcher == 0: //最后一个搜索者解锁
     V(noSearcher)
V(searcherMutex)
```

删除者：

```
P(noSearcher) //和searcher互斥
P(noInserter)  //和inserter互斥
# critical section
V(noInserter)
V(noSearcher)
```

4.搜索插入删除问题

信号量实现

插入者：

```
P(inserterMutex)
  inserter++;
  if inserter == 1: // 第一个inserter加锁
     P(noInserter.wait) //和deleter互斥
V(inserterMutex)

P(insertMutex)   //多个inserter要互斥
 # critical section
V(insertMutex)

P(inserterMutex)
  inserter--;
  if inserter == 0: //最后一个inserter解锁
     V(noInserter)
V(inserterMutex)
```