计算机操作系统

进程同步作业

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

读者写者算法-写者优先

如何实现写者优先的算法?

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)

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)
```

信号量定义

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

最初信号量都是解锁态。
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

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)
```

信号量定义

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

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，而不需要signal noreader。当最后一个写者离开，noreader才解锁。写者才能进入。

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()

当然，这个算法下，读者可能被饿死

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

noWriters = Semaphore(1)

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

```
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)
    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()
```

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

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 生产者消费者

三个进程P1、P2、P3互斥使用一个包含N（N>0）个单元的缓冲区。P1每次用produce()生成一个正整数并用put()送入缓冲区某一个空单元中；P2每次用getodd()从该缓冲区中取出一个奇数并用countodd()统计奇数个数；P3每次用geteven()从该缓冲区中取出一个偶数并用counteven()统计偶数个数。请用信号量机制实现这三个进程的同步与互斥活动，并说明所定义的信号量的含义。要求用伪代码描述。

3 生产者消费者

(1) 缓冲区是一互斥资源，因此设互斥信号量mutex。

(2) 同步问题：P1、P2因为奇数的放置与取用而同步，设同步信号量odd；P1、P3因为偶数的放置于取用而同步，设同步信号量even；P1、P2、P3因为共享缓冲区，设同步信号量empty。

```
semaphore mutex = 1,odd = 0,even = 0,empty = N;
main()
cobegin{
Process P1
 while(true)
 {number = produce();
  P(empty);
  P(mutex);
  put();
  V(mutex);
  If number % 2 == 0
    V(even);
  else
    V(odd);
}
```

```
Process P2
  while(true)
  {P(odd);
   P(mutex);
    getodd();
   V(mutex);
   V(empty);
   countodd();}

Process P3
 while(true)
 {P(even);
  P(mutex);
  geteven();
  V(mutex);
  V(empty);
  counteven(); }

}coend
```

阅卷情况：

- 不会做，乱写一统
- 4种典型的错误
- 没有cobegin-coend,while等语句，扣分
- 回答完全正确，很少

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.wait(noSearcher)//对noSearcher上锁
# critical section //多个searcher互斥
searchSwitch.signal(noSearcher)

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

删除者：
noSearcher.wait()
noInserter.wait()
# critical section
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：

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

  # critical section //多个searcher无需互斥

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

删除者：

```
noSearcher.wait() //和searcher互斥
noInserter.wait()  //和inserter互斥
# critical section
noInserter.signal()
noSearcher.signal()
```

4.搜索插入删除问题

信号量实现

插入者：

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

 insertMutex.wait()   //多个inserter要互斥
 # critical section
 insertMutex.signal()

inserterMutex.wait()
  inserter--;
  if inserter == 0: //最后一个inserter解锁
     noInserter.signal()
inserterMutex.signal()
```