进程同步-3.1

教师：姜博
E-Mail: gongbell@gmail.com

内容提要

- 同步与互斥问题
- 基于忙等待的互斥方法
- 基于信号量的方法
- 基于管程的同步与互斥
- 进程通信的主要方法
- 经典的进程同步与互斥问题

经典进程同步问题

- 生产者 - 消费者问题(the producer-consumer problem)
- 读者 - 写者问题(the readers-writers problem)
- 哲学家进餐问题(the dining philosophers problem)
- 理发师问题

OS和应用开发中，常常遇到类似的问题

生产者消费者问题

- 典型的类似应用场景-事件驱动的程序
- 事件-需要应用响应的事情
  - 用户的按键、滑动、点击
  - 网络数据到达、异步的操作完成
- 当事件发生时：
  - 生产者线程创建一个事件对象，放入事件缓冲区
  - 消费者线程（event handlers）从缓冲区取出事件，进行响应处理

读者 - 写者问题(the readers-writers problem)

- 问题描述：对共享资源的读写操作，任一时刻“写者”最多只允许一个，而“读者”则允许多个；“读 - 写”互斥，“写 - 写”互斥，“读 - 读”允许
- 实际应用场景，对共享数据结构、数据库、文件的多线程并发访问。

读者-写者问题分析

- 生活中的实例：火车/飞机定票
  - 读者：？
  - 写者：？
- 多个线程/进程共享内存中的对象
  - 有些进程读，有些进程写
  - 同一时刻，只有一个激活的写进程
  - 同一时刻，可以有多个激活的读进程
- 分类互斥问题：
  - 当写线程在临界区中，其他任何线程不能进入
  - 当读线程在临界区中，写线程不能进入，读线程可以

采用信号量机制-信号量定义

int readers = 0 //记录临界区内读者的数目

mutex = Semaphore(1)//保护对readers的访问

roomEmpty = Semaphore(1)

//对屋子的互斥访问，初值为1表示一个空屋子

好的命名能增加可读性：

wait：等待条件为真；signal：通知条件为真了。

Writer

```
roomEmpty.wait();
    write
  //critical region
roomEmpty.signal();
```

Reader

```
mutex.wait();
   readers=readers+1;
   if readers == 1 : //第一个读者
      roomEmpty.wait() //对屋子加锁
mutex.signal();

read //critical region

mutex.wait();
   readers = readers-1;
   if readers == 0:
      roomEmpty.signal(); //对屋子解锁
mutex.signal();
```

采用信号量机制-PV操作的版本

int readers = 0

Semaphore mutex = 1

Semaphore roomEmpty = 1

Writer

```
P(roomEmpty);
    write
//critical region
V(roomEmpty);
```

Reader

```
P(mutex);
   readers=readers+1;
   if readers == 1 : //第一个读者
      P(roomEmpty)
V(mutex);

read //critical region

P(mutex);
   readers = readers-1;
   if readers == 0:
      V(roomEmpty);
V(mutex);
```

Writer

```
P(roomEmpty);
    write
//critical region
V(roomEmpty);
```

Reader

```
P(mutex);
   readers=readers+1;
   if readers == 1 : //第一个读者
      P(roomEmpty)
V(mutex);
read //critical region
P(mutex);
   readers = readers-1;
   if readers == 0:
      V(roomEmpty);
V(mutex);
```

考试和作业推荐PV操作的写法

读者-写者- 灯开关模式

- 读者（分类互斥）算法的模式：
  - 第一个读线程加锁，最后一个读线程解锁。
- 通常也称为灯开关模式(Lightswitch)
  - 第一个进屋的人开灯（对mutex加锁）
  - 最后一个离开屋的人关灯（对mutex解锁）

读者-写者- 灯开关模式

```
class Lightswitch:
    def __init__(self):
        self.counter = 0  //封装计数器和互斥锁
        self.mutex = Semaphore(1)
    def lock(self, semaphore): //信号量是参数
        self.mutex.wait()
            self.counter += 1
            if self.counter == 1:
                semaphore.wait()
        self.mutex.signal()
    def unlock(self, semaphore):
        self.mutex.wait()
            self.counter -= 1
            if self.counter == 0:
                semaphore.signal()
        self.mutex.signal()
```

读者-写者- 使用lightswitch

信号量初始化

```
readLightswitch = Lightswitch()
roomEmpty = Semaphore(1)
```

Reader

```
readLightswitch.lock(roomEmpty)
  read # critical section
readLightswitch.unlock(roomEmpty)
```

Writer 代码不变

```
roomEmpty.wait();
    write  //critical region
roomEmpty.signal();
```

采用一般“信号量集”机制

- 增加一个限制条件：同时读的“读者”最多RN个
- mx表示“允许写”，初值是1
- L表示“允许读者数目”，初值为RN

Writer

```
SP(mx, 1, 1; L, RN, 0);
    write
SV(mx, 1);
```

Reader

```
SP(L, 1, 1; mx, 1, 0);
    read
SV(L, 1);
```

L>=RN 允许写

SP(S, 1, 0):可作为一个可控开关：(当S≥1时，允许多个进程进入临界区；当S=0时禁止任何进程进入临界区)

“读者-写者“算法的问题

```
Writer

P(roomEmpty);
    write
//critical region
V(roomEmpty);
```

```
Reader

P(mutex);
   readers=readers+1;
   if readers == 1 : //第一个读者
      P(roomEmpty)
V(mutex);

read //critical region

P(mutex);
   readers = readers-1;
   if readers == 0:
      V(roomEmpty);
V(mutex);
```

该算法是对读者有利，还是对写者有利？

“读者-写者“算法的问题

Reader

```
P(mutex);
   readers=readers+1;
   if readers == 1 : //第一个读者
```

Writer

```
P(
//c
V(
```

写者可能被饿死（Starvation）！
当系统负载很低，可以工作，
当系统负载很高，写者会几乎没机会。

```
P(mutex);
   readers = readers-1;
   if readers == 0:
      V(roomEmpty);
V(mutex);
```

该算法是对读者有利，还是对写者有利？

读者写者算法的特性

- 给定读写序列：r1,w1,w2,r2,r3,w3…
  - 读者优先：r1,r2,r3,w1,w2,w3…
  - 写者优先：r1,w1,w2,w3,r2,r3…
  - 读写公平：r1,w1,w2,r2,r3,w3…
- 如何设计写者优先？
- 如何设计公平读写？

“读者-写者“算法的问题

信号量初始化

```
readLightswitch = Lightswitch()
roomEmpty = Semaphore(1)
```

Reader

```
readLightswitch.lock(roomEmpty)
  read # critical section
readLightswitch.unlock(roomEmpty)
```

Writer 代码不变

问题：如何扩展当前算法，实现公平读写?

一种低级通信原语：屏障Barriers

- 思考：如何使用PV操作实现Barrier？
  - n = the number of threads
  - count = 0 //到达汇合点的进程个数
  - semaphore mutex = 1 //保护count
  - semaphore queue = 0//进程到达之前都是0，用于进程排队。

  - P(mutex)
  - count = count + 1
  - V(mutex)

  - if count == n: V(queue) # 第n个进程到来，唤醒一个进程，触发。
  - P(queue) # 前n-1个进程在此排队
  - V(queue) # 一旦进程被唤醒，有责任唤醒下一个进程

Turnstile-闸机

- queue = Semaphore(0)
- P(queue)
- V(queue)

- Turnstile：连续两个wait和signal组成
- 它可以关闭以阻止所有进程，也可以让进程轮流通过
  - 初值为0，闸机关闭，任何进程不能进入
  - 当值为1，多个进程可以轮流排队通过

```
int readers = 0
Semaphore mutex = 1
Semaphore roomEmpty =1
Semaphore turnstile = 1
```

```
Writer

P(turnstile);
    P(roomEmpty);
    write  //critical region
V(turnstile);
V(roomEmpty);
```

```
Reader

P(turnstile)
V(turnstile)
P(mutex);
   readers=readers+1;
   if readers == 1 : //第一个读者
      P(roomEmpty)
V(mutex);

read //critical region

P(mutex);
   readers = readers-1;
   if readers == 0:
      V(roomEmpty);
V(mutex);
```

```
int readers = 0
Semaphore mutex = 1
Semaphore roomEmpty =1
Semaphore turnstile = 1
```

```
Writer

P(turnstile);
    P(roomEmpty);
    write  //critical region
V(turnstile);
V(roomEmpty);
```

Writer先在闸机排队后，然后等待roomEmpty,

```
Reader

P(turnstile)
V(turnstile)
P(mutex);
   readers=readers+1;
   if readers == 1 : //第一个读者
      P(roomEmpty)
V(mutex);

read //critical region

P(mutex);
   readers = readers-1;
   if readers == 0:
      V(roomEmpty);
V(mutex);
```

readers 在闸机排队

非饥饿版本的读者写者算法-公平读写

```
readSwitch = Lightswitch()
roomEmpty = Semaphore(1)

turnstile = Semaphore(1)//对写者互斥锁，对读者闸机
Writer
turnstile.wait()
        roomEmpty.wait()
        # critical section for writers
turnstile.signal()
roomEmpty.signal()

Reader
turnstile.wait()
turnstile.signal()
readSwitch.lock(roomEmpty)
        # critical section for readers
readSwitch.unlock(roomEmpty)
```

非饥饿版本的读者写者算法-公平读写

```
readSwitch = Lightswitch()
roomEmpty = Semaphore(1)

turnstile = Semaphore(1)
```

Writer

```
turnstile.wait()
        roomEmpty.wait()
        # critical section for writers
```

调度器会决定闸机外排队的那个进程先被调度
完全的公平取决于调度器支持

Reader

```
turnstile.wait()
turnstile.signal()
readSwitch.lock(roomEmpty)
        # critical section for readers
readSwitch.unlock(roomEmpty)
```

理发师问题

- Dijkstra首先提出
- 理发店里有一位理发师、一把理发椅和n把供等候理发的顾客坐的椅子；
- 如果没有顾客，理发师便在理发椅上睡觉，当一个顾客到来时，叫醒理发师；
- 如果理发师正在理发时，又有顾客来到，则如果有空椅子可坐，就坐下来等待，否则就离开。

互斥访问资源：排队的顾客数（计数器 waiting）
同步：顾客唤醒理发师、理发师唤醒下一个位等待顾客

理发师问题

```
semaphore customers = 0; //等待理发的顾客
semaphore barbers = 0;  //等待顾客的理发师
semaphore mutex = 1;  //互斥访问waiting
int waiting = 0;  //等待的顾客数 (不包含正在理发的顾客)
```

算法描述

```
信号量：customers=0;barbers=0;mutex=1
整型变量：waiting=0;
假设：CHIRS=10
理发师进程：
begin
   While(true)then
   begin
      P(customers);
      P(mutex);
        waiting=waiting-1;
      V(mutex);
      V(barbers);
      Cut hair();
   end
end
```

```
顾客进程：
begin
   P(mutex);
      If （waiting<CHIRS)
then
          begin
            waiting=waiting+1;
            V(mutex);
           V(customers);
            P(barbers);
              Get_haircut();
          end
          else
          begin
            V(mutex);
          end
end
```

有座位等么？

若顾客为0，睡觉

唤醒barber

没有barber，等待

准备好剪发

没座位离开

理发师问题

```
#define CHAIRS 5   //chairs for waiting customers
typedef int semaphore;
semaphore customers = 0; //# of customers waiting service
semaphore barbers = 0;  //# of barbers waiting customers
semaphore mutex = 1;  //for mutual exclusion of waiting
int waiting = 0;  //customer are waiting (not being cut)
```

理发师问题

```
void barber(void) {
    while (TRUE) {
            P(&customers); /* go to sleep if # of customers is 0 */
            P(&mutex); /* acquire access to "waiting' */
            waiting = waiting - 1; /* decrement count of waiting customers */
            V(&mutex); /* release 'waiting' */
            V(&barbers); /* one barber is now ready to cut hair */
            cut_hair(); /* cut hair (outside critical region */
            }
 }
```

理发师问题

```
void customer(void) {
      down(&mutex); /* enter critical region */
      if (waiting < CHAIRS) {
        /* if there are no free chairs, leave */
            waiting = waiting + 1; /* increment count of waiting customers */
            V(&mutex); /* release access to 'waiting' */
            V(&customers); /* wake up barber if necessary */
            P(&barbers); /* go to sleep if # of free barbers is 0 */
            get_haircut(); /* be seated and be served */ }
      else {
        V(&mutex); /* shop is full; do not wait */
      }
}
```

“生产者-消费者”扩展问题1

- 某银行有n个服务柜台。每个顾客进店后先取一个号，并且等待叫号。当一个柜台人员空闲下来时，就叫下一个号。试设计一个使柜台人员和顾客同步的算法。
- 谁是生产者？
- 谁是消费者？

“生产者-消费者”扩展问题1

- 某银行有n个服务柜台。每个顾客进店后先取一个号，并且等待叫号。当一个柜台人员空闲下来时，就叫下一个号。试设计一个使柜台人员和顾客同步的算法。
- 谁是生产者？
- 谁是消费者？

算法描述

```
int next_cstmr = 0; //下一个要服务的客户
Semaphore s_mutex=1;//服务器进程互斥访问next_cstmr
Semaphore cstmr_cnt = 0; //客户服务器进程同步
```

```
process customer i
Begin

   v(cstmr_cnt);

  }
end
```

```
process servers i(i=1,...,n)
begin
    while(true){
        p(s_mutex);
        p(cstmr_cnt);
        next_cstmr ++;
        v(s_mutex);
        为持有next_cstmr的客户服务；
    }
end
```

进程同步-3.2

教师：姜博
E-Mail: gongbell@gmail.com

“生产者-消费者”扩展问题1

- 某银行有n个服务柜台。每个顾客进店后先取一个号，并且等待叫号。当一个柜台人员空闲下来时，就叫下一个号。试设计一个使柜台人员和顾客同步的算法。
- 谁是生产者？
- 谁是消费者？

算法描述

```
int next_cstmr = 0; //下一个要服务的客户
Semaphore s_mutex=1;//服务器进程互斥访问next_cstmr
Semaphore cstmr_cnt = 0; //客户服务器进程同步
```

```
process customer i
Begin

   v(cstmr_cnt);

  }
end
```

```
process servers i(i=1,...,n)
begin
    while(true){
        p(s_mutex);
        p(cstmr_cnt);
        next_cstmr ++;
        v(s_mutex);
        为持有next_cstmr的客户服务；
    }
end
```

另外一种算法描述

```
int cstmr_id = 0;   //当前客户编号
semaphore mutex=1; //对cstmr_id互斥访问
int next_cstmr = 0; //下一个要服务客户编号
semaphore s_mutex=1; //服务器进程互斥访next_cstmr
```

```
process customer i
Begin
  p(mutex);
     cstmr_id ++;
  v(mutex);

  }
end
```

```
process servers i(i=1,...,n)
begin
    while(true){
        p(s_mutex);
        p(mutex);
           if(next_cstmr < cstmr_id)
             next_cstmr ++;
        v(mutex)
        v(s_mutex);
        为next_cstmr号码持有者服务；
    }
end
```

“生产者-消费者”扩展问题2

- 设有一个可以装A、B两种物品的仓库,其容量无限大,但要求仓库中A、B两种物品的数量满足下述不等式:
  - -M≤A物品数量-B物品数量≤N
  - 其中M和N为正整数.
- 试用信号量和PV操作描述A、B两种物品的入库过程.

算法描述

```
Semaphore mutex=1，sa=N，sb=M;
cobegin
    procedure A:                    procedure B:
      while(TURE)                       while(TURE)
       begin                             begin
         p(sa);                          p(sb);
         p(mutex);                       p(mutex);
         A产品入库；                          B产品入库；
         V(mutex);                       V(mutex);
         V(sb);                          V(sa);
       end                               end
coend
```

思考

- 如果仓库容量有限，如何处理？

- 如果增加一个消费者，同时消费A和B两个物品，如何设计算法？

构建水分子(H2O)问题

- Berkeley OS 课程习题.
- Gregory R. Andrews. Concurrent Programming: Principles and Practice. Addison-Wesley, 1991.
- 存在两种线程，一个线程提供氧原子O，一个线程提供氢原子H。为了构建水分子，我们需要使用barrier让线程同步从而构建水分子(H_2O)。
- 当线程通过barrier，需要调用bond（形成化学键），需要保证构建同一个分子的线程调用bond。
  - 当氧原子线程到达barrier，而氢原子线程还没到达，需要等待氢原子。
  - 当1个氢原子到达而没有其他线程到达，需要等待1个氢原子1个氧原子
- 只需要保证成组通过barrier

构建水分子(H2O)问题

- Berkeley OS 课程习题.

如何构建同步原语？

- 存在两种线程，一个线程提供氧原子O，一个线程提供氢原子H。为了构建水分子，我们需要使用barrier让线程同步从而构建水分子(H_2O)。
- 当线程通过barrier，需要调用bond（形成化学键），需要保证构建同一个分子的线程调用bond。
  - 当氧原子线程到达barrier，而氢原子线程还没到达，需要等待氢原子。
  - 当1个氢原子到达而没有其他线程到达，需要等待1个氢原子1个氧原子
- 只需要保证成组通过barrier，不需要

构建水分子(H2O)问题

- **信号量定义**

oxygen = 0 //氧原子的计数器
hydrogen = 0 //氢原子的计数器
Semaphore mutex = 1 //保护计数器的mutex
Barrier barrier(3) //3表示需要调用3次wait后barrier才开放
//3个线程调用bond后的同步点，之后允许下一个线程继续
Semaphore oxyQueue = 0 //氧气线程等待的信号量
Semaphore hydroQueue = 0 //氢气线程等待的信号量
//用在信号量上睡眠来模拟队列
P(oxyQueue) 表示加入队列
V(oxyQueue) 表示离开队列

构建水分子(H2O)问题

- **氧气线程**

```
P(mutex)
oxygen += 1
if hydrogen >= 2:
    V(hydroQueue)
    V(hydroQueue)
    hydrogen -= 2
    V(oxyQueue)
    oxygen -= 1
else:
    V(mutex)
P(oxyQueue)
bond()
barrier.wait()
V(mutex)
```

构建H_2O成功

- **氢气线程**

```
P(mutex)
hydrogen += 1
if hydrogen >= 2 and oxygen >= 1:
    V(hydroQueue)
    V(hydroQueue)
    hydrogen -= 2
    V(oxyQueue)
    oxygen -= 1
else:
    mutex.signal()
P(hydroQueue)
bond()
barrier.wait()
```

构建H_2O成功

不成功释放mutext

同步以生成水分子

构建水分子(H2O)问题

- **氧气线程**

```
P(mutex)
oxygen += 1
if hydrogen >= 2:
    V(hydroQueue)
    V(hydroQueue)
    hydrogen -= 2
    V(oxyQueue)
    oxygen -= 1
else:
    V(mutex)
P(oxyQueue)
bond()
barrier.wait()
V(mutex)
```

- **氢气线程**

```
P(mutex)
hydrogen += 1
if hydrogen >= 2 and oxygen >= 1:
    V(hydroQueue)
    V(hydroQueue)
    hydrogen -= 2
    V(oxyQueue)
    oxygen -= 1
else:
    V(mutex)
P(hydroQueue)
bond()
barrier.wait()
```

构建H_2O成功，有一个mutex没释放，释放mutex

当三个线程离开barrier时候，最后那个线程拿着mutex，虽然我们不知道那个线程hold mutext，但是我们一定要释放一次。因为氧气只有一个线程，就放在氧气线程中做了。

```
P(mutex)
oxygen += 1
if hydrogen >= 2:
    V(hydroQueue)
    V(hydroQueue)
    hydrogen -= 2
    V(oxyQueue)
    oxygen -= 1
else:
    V(mutex)
P(oxyQueue)
bond()
barrier.wait()
V(mutex)
```

```
P(mutex)
hydrogen += 1
if hydrogen >= 2 and oxygen >= 1:
    V(hydroQueue)
    V(hydroQueue)
    hydrogen -= 2
    V(oxyQueue)
    oxygen -= 1
else:
    V(mutex)
P(hydroQueue)
bond()
barrier.wait()
```

死锁（deadlock）

教师：姜博
E-Mail: gongbell@gmail.com

内容提要

- 死锁的概念
- 处理死锁的基本方法
- 小结

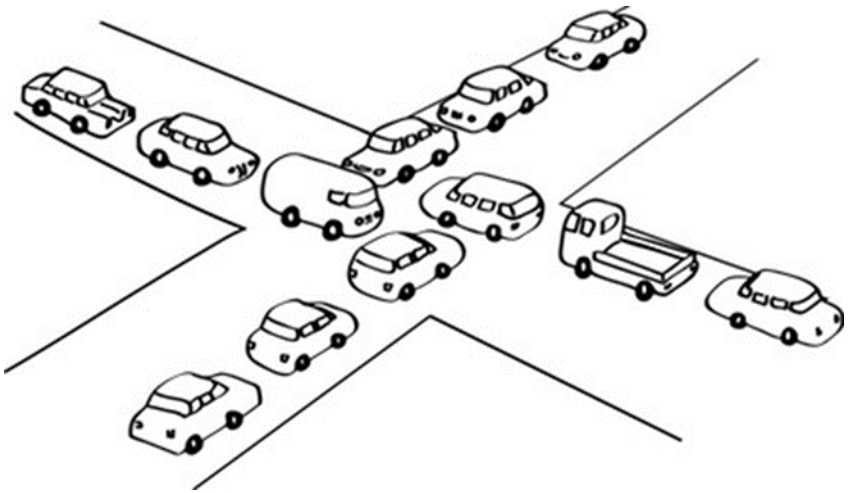
死锁问题(Deadlock)

死锁定义：

- 如果一个进程集合中的每个进程都在等待只能由该进程集合中其他进程才能引发的事件，那么该进程集合就是死锁的。

死锁发生原因

- 竞争资源
- 并发执行的顺序不当



死锁问题(Deadlock)

死锁定义：

- 如果一个进程集合中的每个进程都在等待只能由该进程集合中其他进程才能引发的事件，那么该进程集合就是死锁的。

资源死锁

- 如果每个进程等待的事件是释放该进程集合中其他进程所占有的资源。

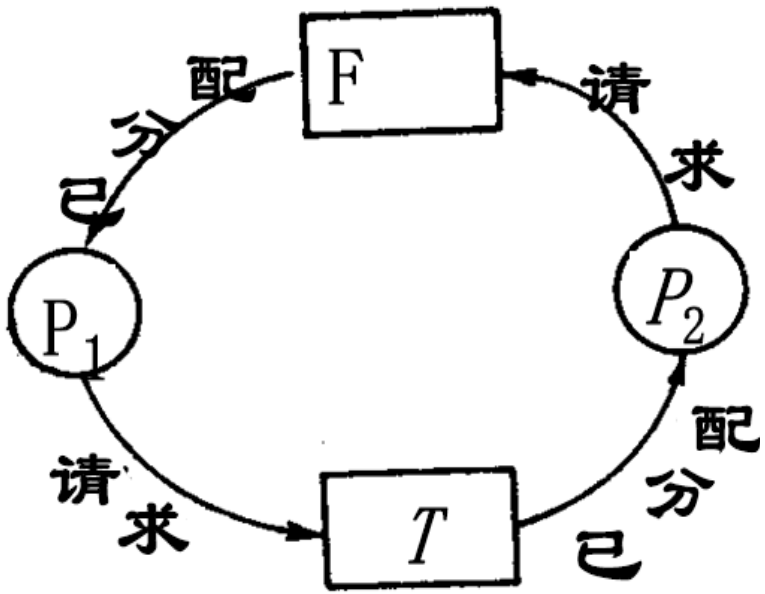
饥饿！=死锁：

最短作业优先调度，大作业饥饿而非死锁。

死锁的例子

进程P1
…
申请文件F
申请打印机T
…
释放打印机T
释放文件F
…

进程P2
…
申请打印机T
申请文件F
…
释放文件F
释放打印机T
…



简单的死锁例子

竞争资源引起死锁

- 可剥夺资源：是指某进程在获得这类资源后，该资源可以再被其他进程或系统剥夺。 如CPU，内存；
- 非可剥夺资源：当系统把这类资源分配给某进程后，再不能强行收回，只能在进程用完后自行释放。如CD刻录机、打印机；
  - 临时性资源：这是指由一个进程产生，被另一个进程使用，短时间后便无用的资源，故也称为消耗性资源。如消息、中断；(不可剥夺)

临时性资源竞争示例

例如，SI，S2，S3是临时性资源，进程P1产生消息S1，又要求从P3接收消息S3；进程P3产生消息S3，又要求从进程P2处接收消息S2；进程P2产生消息S2，又要求从P1处接收产生的消息S1。如果消息通信按如下顺序进行：

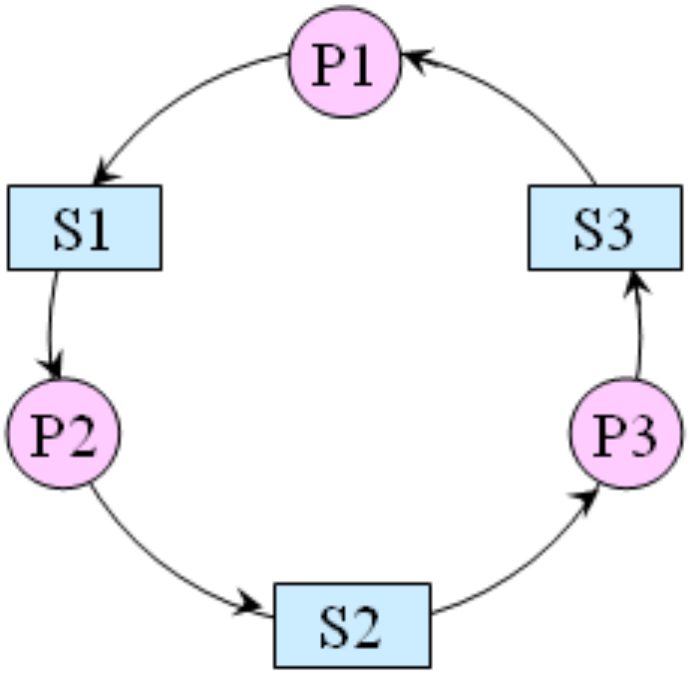
P1: Relese (S1) ; Request (S3) ;
P2: Relese (S2) ; Request (S1) ;
P3: Relese (S3) ; Request (S2) ;

并不会发生死锁。

若改成下述的运行顺序：

P1: Request (S3) ; Relese (S1) ;
P2: Request (S1) ; Relese (S2) ;
P3: Request (S2) ; Relese (S3) ;

则可能发生死锁。



使用信号量实现会和 (Rendezvous)

- 使用信号量实现线程A和线程B的会和(Rendezvous)。使得a1永远在b2之前，而b1永远在a2之前。
- 定义两个信号量，aArrived, bArrived,并且初始化为0，表示a和b是否执行到汇合点。

Thread A

```
1 statement a1
2 bArrived.wait()
3 aArrived.signal()
4 statement a2
```

Thread B

```
1 statement b1
2 aArrived.wait()
3 bArrived.signal()
4 statement b2
```

死锁版本，signal在wait后

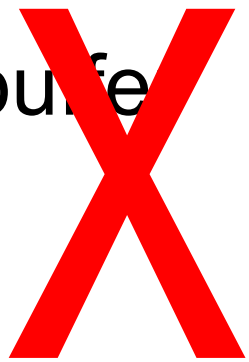
生产者和消费者问题 - sleep&wakeup

```
#define N 100                                   /* number of slots in the buffer */
int count = 0;                                  /* number of items in the buffer */

void producer(void)
{
     int item;

     while (TRUE) {                             /* repeat forever */
          item = produce_item( );               /* generate next item */
          if (count == N) sleep( );             /* if buffer is full, go to sleep */
          insert_item(item);                    /* put item in buffer */
          count = count + 1;                    /* increment count of items in buffer */
          if (count == 1) wakeup(consumer);     /* was buffer empty? */
     }
}


void consumer(void)
{
     int item;

     while (TRUE) {                             /* repeat forever */
          if (count == 0) sleep( );             /* if buffer is empty, got to sleep */
          item = remove_item( );                /* take item out of buffer */
          count = count – 1;                    /* decrement count of items in buffer */
          if (count == N – 1) wakeup(producer); /* was buffer full? */
          consume_item(item);                   /* print item */
     }
}
```

丢失一个sleep后，互相等待唤醒

- full是“满”数目，初值为0，empty是“空”数目，初值为N。实际上，full和empty是同一个含义：full + empty == N
- mutex用于访问缓冲区时的互斥，初值是1

生产者	消费者
P(mutex);	P(mutex);
P(empty);	P(full);
one >> buffer	one << buffer
V(full)	V(empty)
V(mutex)	V(mutex)



（资源）死锁发生的四个必要条件

1. 互斥条件：指进程对所分配到的资源进行排它性使用，即在一段时间内某资源只能有一个进程占用。
2. 保持和请求条件：已经获得资源的线程可以请求新的资源。
3. 不剥夺条件：指进程已获得的资源，在未使用完之前不能被强制剥夺，只能在使用完时由自己释放。
4. 环路等待条件：指在发生死锁时，必然存在两个或多个进程组成的环形链，每个进程都在等待环形链中下一个节点占用的资源。例如，进程集合{P0，P1，P2，…，Pn}中的P0正在等待P1占用的资源；P1正在等待P2占用的资源，……，Pn正在等待已被P0占用的资源。

内容提要

- 死锁的概念
- 处理死锁的基本方法
  - 鸵鸟算法
  - 死锁检测
  - 死锁预防
  - 死所避免
- 小结

处理死锁方法

- 鸵鸟算法：无所作为，无视死锁。
  - 如果死锁发生概率小、影响低时，可以考虑
- 死锁检测（deadlock detection）与死锁恢复(deadlock recovery)
  - 允许死锁发生，当检测死锁发生后，采取措施恢复。
  - 死锁检测算法
    - 基于资源分配图-每类一个资源
    - 基于资源向量计算-每类多个资源

处理死锁方法

- 死锁预防（静态 deadlock avoidance）：破坏死锁的产生的四个条件之一。
  - 互斥、占有等待、不可抢占、环路等待
- 死锁避免（动态 deadlock prevention）：在资源分配之前判断是否安全，仅当安全才进行分配。
  - 需要依赖执行前获取额外的信息
    - 银行家算法：运行前需要知道进程所需资源最大值

内容提要

- 死锁的概念
- 处理死锁的基本方法
  - 鸵鸟算法
  - 死锁检测与恢复
  - 死锁预防
  - 死琐避免
- 小结