操作系统 Operation System

进程同步

王雷 82316284

wanglei@buaa.edu.cn

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

程序的并发执行

并发是OS的设计基础，也是所有（如，同步互斥）问题产生的原因。

- 进程的三个特征：
  - **并发：**体现在进程的执行是间断性的；进程的相对执行速度是不可测的。（间断性）
  - **共享：**体现在进程/线程之间的制约性（如共享打印机）（非封闭性）。
  - **不确定性：**进程执行的结果与其执行的相对速度有关，是不确定的（不可再现性）。

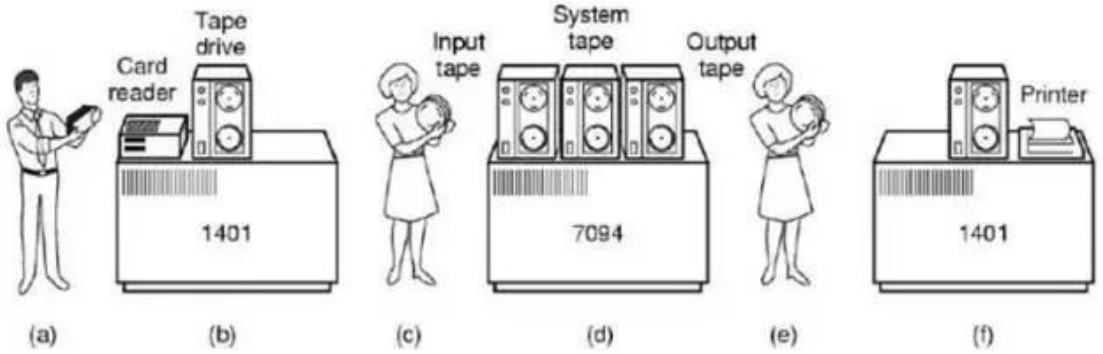
程序的并发执行

并发执行，不可避免地产生了多个进程对同一个共享资源访问，造成了资源的争夺。

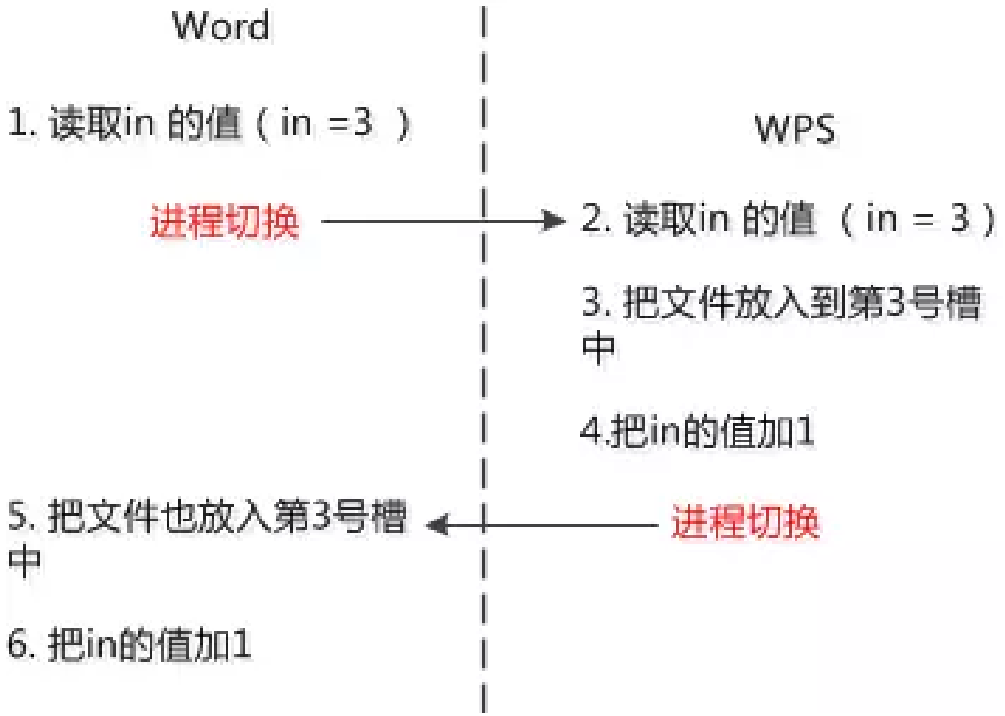
- **竞争：** 两个或多个进程对同一共享数据同时进行访问，而最后的结果是不可预测的，它取决于各个进程对共享数据访问的相对次序。这种情形叫做竞争。
- **竞争条件：** 多个进程并发访问和操作同一数据且执行结果与访问的特定顺序有关。
- **临界资源：** 我们将一次仅允许一个进程访问的资源称为临界资源。
- **临界区：** 每个进程中访问临界资源的那段代码称为临界区。

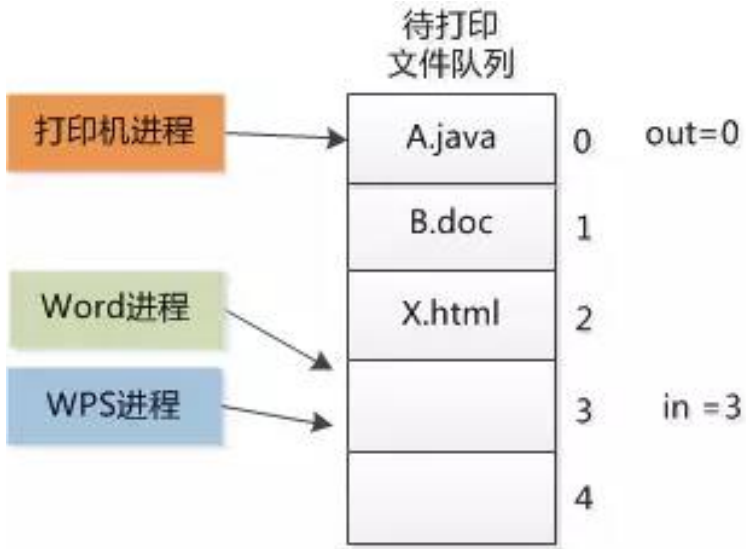
临界资源

- 一次只允许一个进程使用的资源，如打印机。



Spooling系统





临界区

- 每个进程中访问临界资源的那段代码称为临界区。

P：

……

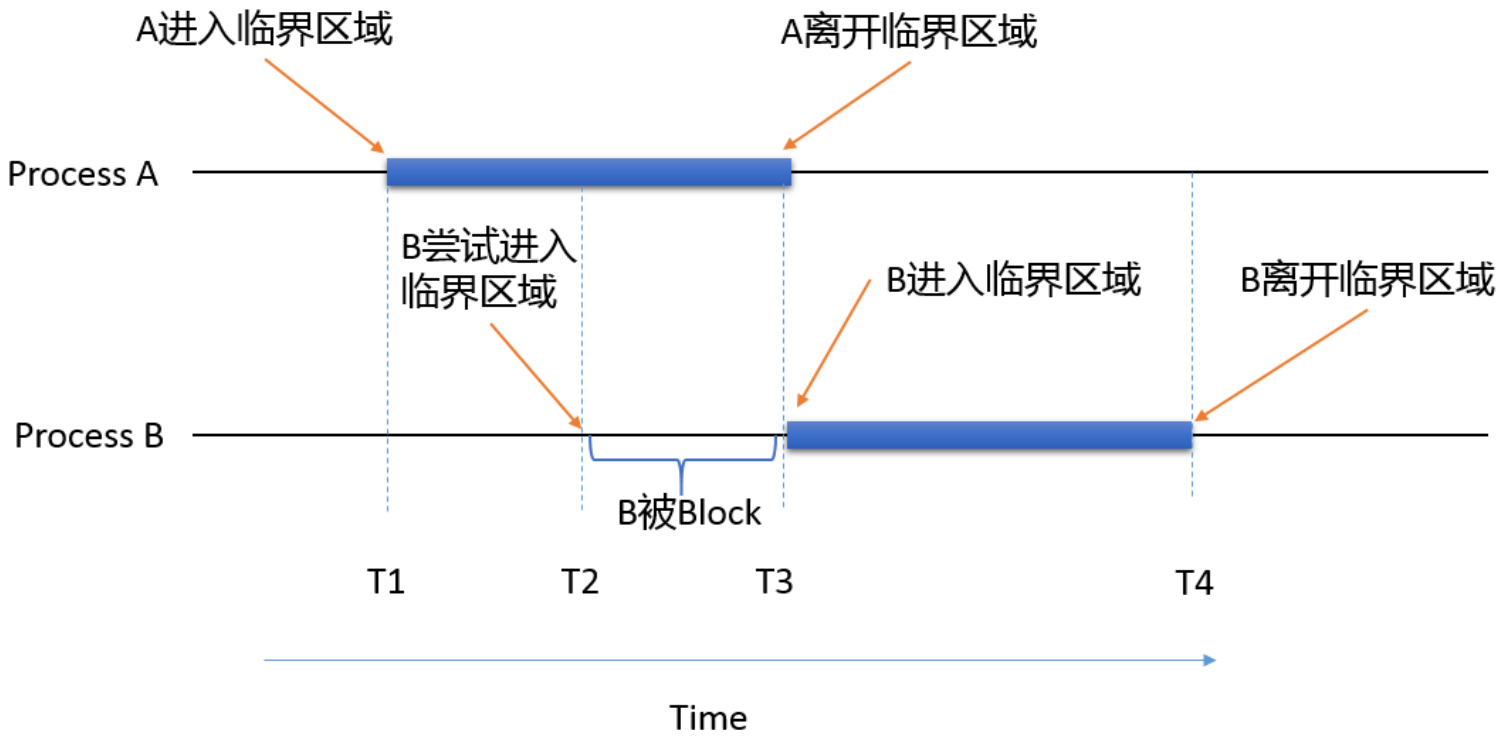
Entry section

critical section

exit section

remainder section

……



进程的同步与互斥

- 进程互斥（间接制约关系）：
  - 两个或两个以上的进程，不能同时进入关于同一组共享变量的临界区域，否则可能发生与时间有关的错误，这种现象被称作进程互斥。
  - 进程互斥是进程间发生的一种间接性作用，一般是程序不希望的。
- 进程同步（直接制约关系）：
  - 系统中各进程之间能有效地共享资源和相互合作，从而使程序的执行具有可再现性的过程称为进程同步。
  - 进程同步是进程间的一种刻意安排的直接制约关系。即为完成同一个任务的各进程之间，因需要协调它们的工作而相互等待、相互交换信息所产生的制约关系。

同步与互斥的区别与联系

- 互斥：某一资源同时只允许一个访问者对其进行访问，具有唯一性和排它性。互斥无法限制访问者对资源的访问顺序，即访问是无序访问。
- 同步：是指在互斥的基础上（大多数情况），通过其它机制实现访问者对资源的有序访问。在大多数情况下，同步已经实现了互斥，特别是所有对资源的写入的情况必定是互斥的。少数情况是指可以允许多个访问者同时访问资源。

临界区管理应满足的条件

1. 没有进程在临界区时，想进入临界区的进程可进入。
2. 任何两个进程都不能同时进入临界区（Mutual Exclusion）；
3. 当一个进程运行在它的临界区外面时，不能妨碍其他的进程进入临界区（Progress）；
4. 任何一个进程进入临界区的要求应该在有限时间内得到满足（Bounded Waiting）。

机制设计上应遵循的准则

- **空闲让进：**临界资源处于空闲状态，允许进程进入临界区。如，临界区内仅有一个进程执行
- **忙则等待：**临界区有正在执行的进程，所有其他进程则不可以进入临界区
- **有限等待：**对要求访问临界区的进程，应在保证在有限时间内进入自己的临界区，避免死等。
- **让权等待：**当进程（长时间）不能进入自己的临界区时，应立即释放处理机，尽量避免忙等。

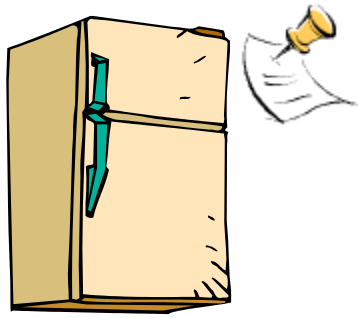
内容提要

- 同步与互斥问题
- 基于忙等待的互斥方法
  - 软件方法
  - 硬件方法
- 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

例子：Too much milk

Time	Person A	Person B
3:00	Look in Fridge. Out of milk	
3:05	Leave for store	
3:10	Arrive at store	Look in Fridge. Out of milk
3:15	Buy milk	Leave for store
3:20	Arrive home, put milk away	Arrive at store
3:25		Buy milk
3:30		Arrive home, put milk away

- 当冰箱是空的时候，购买牛奶
- 但不能买太多（只有1个人买牛奶）
- 两个人通过纸条进行沟通



软件方法尝试1

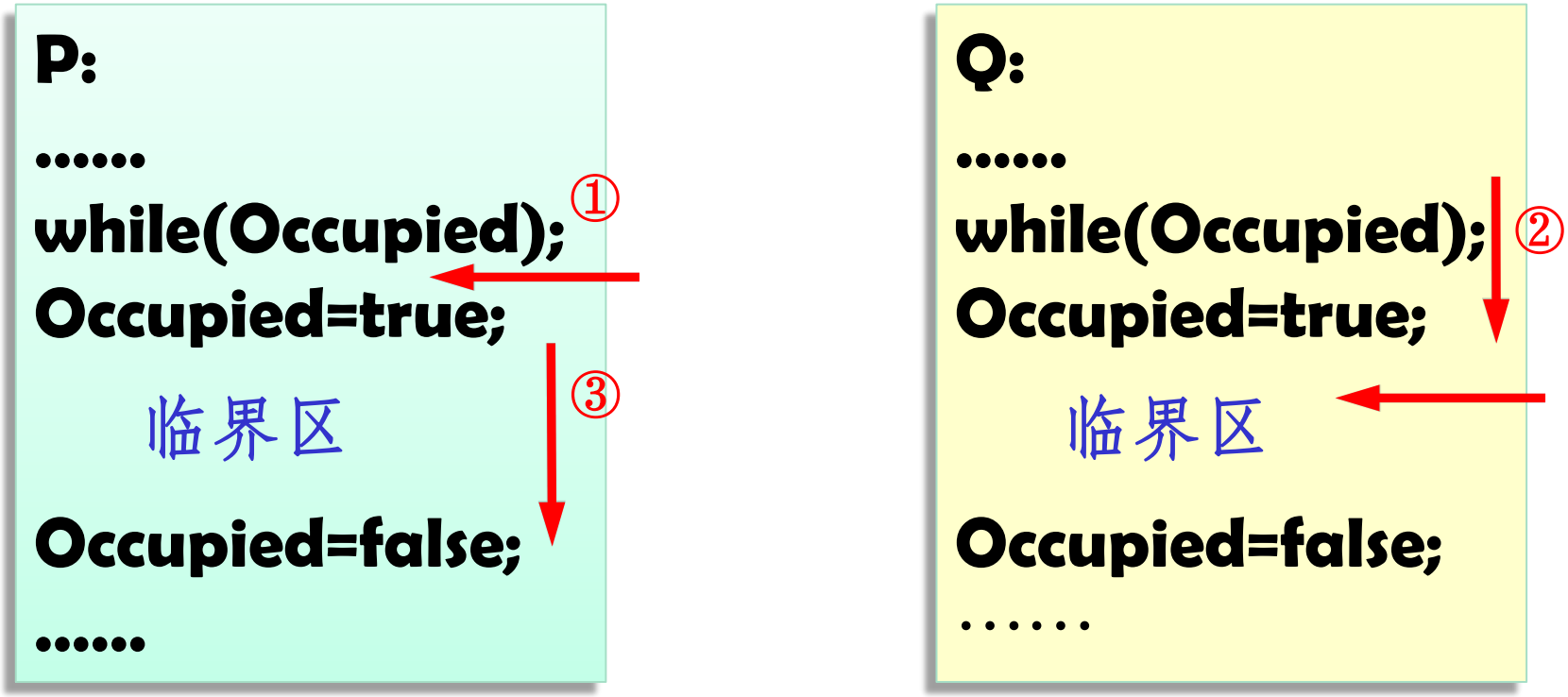
```
P:
……
while(turn==Q);
turn=P;
    临界区
turn=Q;
……
```

```
Q:
……
while(turn==P);
turn=Q;
    临界区
turn =P;
……
```

turn：进程进入临界区的优先标志。
turn的初值为P或Q（对两个进程而言就是0或1）

问题：固然实现了互斥，但要求两进程严格交替进入临界区。否则，一个临界区外的进程会阻止另一个进程进入临界区，违反了progress原则。

软件方法尝试2



Occupied：临界区空满标志
true：临界区内有进程
false：临界区内无进程
Occupied的初值为false

问题：先检查有无标志，后留标志，造成一个空挡，不能实现互斥。

软件方法尝试3


```
P:
……
pturn=true;
while(qturn);
    临界区
pturn=false;
……
```

```
Q:
……
qturn=true;
while(pturn);
    临界区
qturn=false;
……
```

pturn, qturn：进程需要进入临界区的标志，初值为false
P进入临界区的条件：pturn∧not qturn。
Q进入临界区的条件：qturn∧not pturn

问题：竞争时都可能无法进入临界区，违反了progress原则。

软件方法尝试4

```
P:
……
pturn=true;
while(qturn)
    pturn=false;
pturn=true;
    临界区
pturn=false;
……
```

```
Q:
……
qturn=true;
while(pturn)
    qturn=false;
qturn=true;
    临界区
qturn=false;
……
```

问题：增加了让权等待，导致互斥规则被破坏。

Dekker算法

引入变量turn，以便在竞争时选出进入临界区的进程

```
P:
……
pturn=true;
while(qturn)  {
   if(turn ==1) {
        pturn=false;
        while(turn==1);
        pturn=true;
   } }
   临界区
turn = 1;
pturn=false;
……
```

让权

```
Q:
……
qturn=true;
while(pturn)  {
   if(turn ==0) {
        qturn=false;
        while(turn==0);
        qturn=true;
   } }
   临界区
turn = 0;
qturn=false;
……
```

缺点：忙等
浪费CPU时间

1965年第一个用软件方法解决了临界区问题

Peterson算法

```
#define FALSE 0
#define TRUE  1
#define N     2                    // 进程的个数
int turn;                          // 轮到谁?
    int interested[N];
  // 兴趣数组，初始值均为FALSE

void enter_region ( int process)
                     // process = 0 或 1
   {
       int other;
            // 另外一个进程的进程号
       other = 1 - process;
       interested[process] = TRUE;
            // 表明本进程感兴趣
       turn = process;
            // 设置标志位
       while( turn == process &&
    interested[other] == TRUE);
   }
```

循环

```
void leave_region ( int process)
   {
       interested[process] = FALSE;
            // 本进程已离开临界区
   }
```

进程i：

```
... ...
enter_region ( i );
   临界区
leave_region ( i );
... ...
```

Peterson算法解决了互斥访问的问题，而且克服了强制轮流法的缺点，可以完全正常地工作（1981）

N个进程互斥的软件算法

- 实现进程互斥的软件的结构框架是：

```
Repeat
        entry section
        critical section
        exit section
        remainder section
Until false
```

- 进程互斥的软件实现算法有：Lamport面包店算法和Eisenberg算法。这两种算法均假定系统中进程的个数有限，如n个。

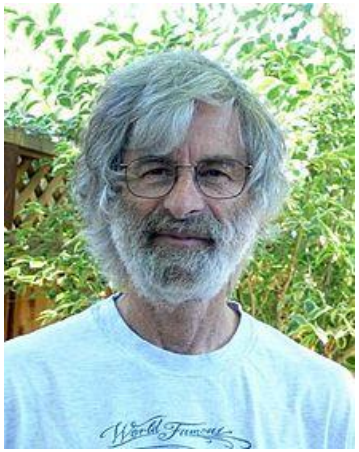
Lamport Bakery Algorithm

面包店算法（Bakery Algorithm）的基本思想来源于顾客在面包店购买面包时的排队原理。顾客进入面包店前，首先抓取一个号码，然后按号码从小到大的次序依次进入面包店购买面包，这里假定：

- (1)—面包店按由小到大的次序发放号码，且两个或两个以上的顾客有可能得到相同号码（要使顾客的号码不同，需互斥机制）；
- (2)—若多个顾客抓到相同号码，则按顾客名字的字典次序排序 **i.e., 1,2,3,3,3,3,4,5...** （假定顾客没有重名）。

Lamport Bakery Algorithm

- 计算机系统中，顾客相当于进程，每个进程有一个唯一的标识，用Pi表示，对于Pi和Pj，若有i<j，即Pi先进入临界区，则先为Pi服务。
- 基本思想：设置一个发号器，按由小到大的次序发放号码。进程进入临界区前先抓取一个号码，然后按号码从小到大的次序依次进入临界区。若多个进程抓到相同的号码则按进程编号依次进入。



Leslie Lamport
2013 Turing Award

- 面包店算法是1974年由莱斯利·兰波特给出的。著名的排版软件LaTeX也是他的贡献。

数据结构

int choosing[n];	//表示进程是否正在抓号，初值为0。若进程i正在抓号，则choosing[i]=1.
int number[n];	//记录进程抓到的号码，初值为0。若number[i]=0，则进程i没有抓号

排序方法：字典序

(a, b)＜(c, d) → (a＜c) or ((a == c) and (b＜d))

初始值：

Choosing，Number：array [1..N] of integer = {0};

Bakery Algorithm (for Pi)

```
Entry Section (i) {    // i → process i
    while (true) {
        Choosing[i] = 1;
        Number[i] = 1 + max(Number[1],...,Number[N]);
        Choosing [i] = 0;
        for (j=1; j<=N; ++j) {
              while (Choosing[j] != 0)  { }
              // wait until process j receives its number
              while ((Number[j]!=0) && ((Number[j],j) <(Number[i],i))) { }
              // wait until processes with smaller numbers, or with the
              // same number, but with higher priority, finish their work
        }
            // critical section...
        Number[i] = 0;
          // non-critical section...
    }
}
```

Bakery算法的说明

- 当进程Pi计算完max(…)+1但尚未将值赋给number[i]时，进程Pj中途插入，计算max(…)+1，得到相同的值。在这种情况下，i和j可保证编号较小的进程先进入临界区，并不会出现两个进程同时进入临界区的情况。
- 忙式等待：上述Lamport面包店算法中，若while循环的循环条件成立，则进程将重复测试，直到条件为假。实际上，当while循环条件成立时，进程Pi不能向前推进，而在原地踏步，这种原地踏步被称为忙式等待。忙式等待空耗CPU资源，其它进程无法使用，因而是低效的。

内容提要

- 同步与互斥问题
- 基于忙等待的互斥方法
  - 软件方法
  - 硬件方法
- 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

硬件方案1：中断屏蔽

中断屏蔽方法：使用“开关中断”指令。

- 执行“关中断”指令，进入临界区操作；
- 退出临界区之前，执行“开中断”指令。

优缺点：

- 简单。
- 不适用于多CPU系统：往往会带来很大的性能损失；
- 单处理器使用：很多日常任务，都是靠中断的机制来触发的，比如时钟，如果使用屏蔽中断，会影响时钟和系统效率，而且用户进程的使用可能很危险！
- 使用范围：内核进程（少量使用）。

硬件方案2：使用test and set指令

- TS（test-and-set）是一种不可中断的基本原语（指令）。它会写值到某个内存位置并传回其旧值。在多进程可同时存取内存的情况下，如果一个进程正在执行检查并设置，在它执行完成前，其它的进程不可以执行检查并设置。
- Test and Set指令
  - IBM370系列机器中称为TS；
  - 在INTEL8086中称为TSL；
  - MIPS中ll和sc指令。
- 语义：

```
TestAndSet(boolean_ref lock) {
    boolean  initial = lock;
    lock = true;
    return initial; }
```

自旋锁Spinlocks

- 利用test_and_set硬件原语提供互斥支持
- 通过对总线的锁定实现对某个内存位置的原子读与更新

```
acquire(lock) {
   while(test_and_set(lock) == 1)
}

        critical section

release(lock) {
    lock = 0;
}
```

Spinlocks

0

```
acquire(int *lock) {
    while(test_and_set(lock) == 1)
            /* do nothing */;
}
release(int *lock) { *lock = 0;  }
```

Let me in!!!

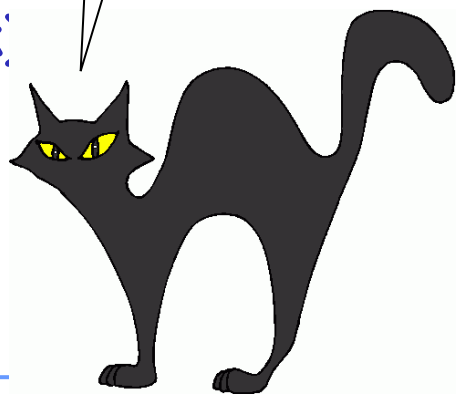
No, Let *me* in!!!

```
acquire(houselock);
Jump_on_the_couch();
Be_goofy();
release(houselock);
```

```
acquire(houselock);
Nap_on_couch();
Release(houselock)
```

1

1

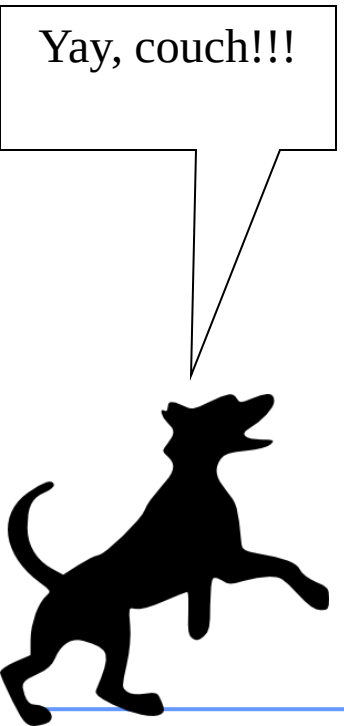


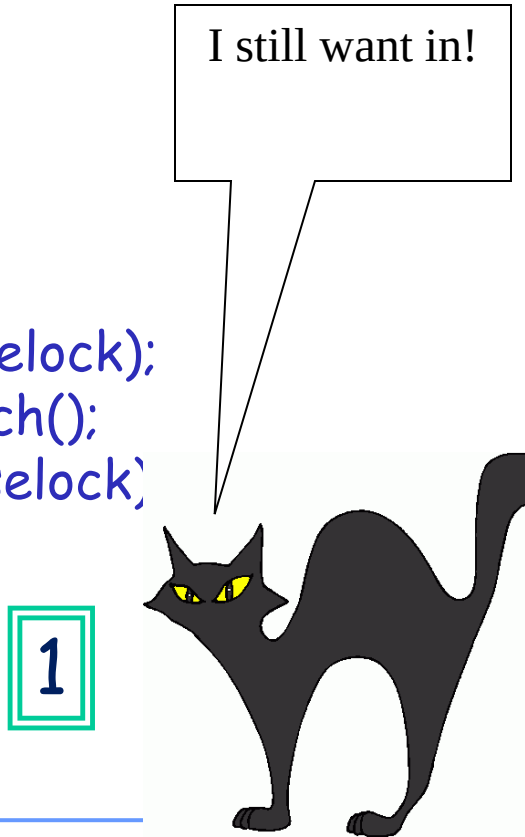
Spinlocks


```
acquire(int *lock) {
   while(test_and_set(lock) == 1)
            /* do nothing */;
}
release(int *lock) { *lock = 0;  }
```

```
acquire(houselock);
Jump_on_the_couch();
Be_goofy();
release(houselock);
```

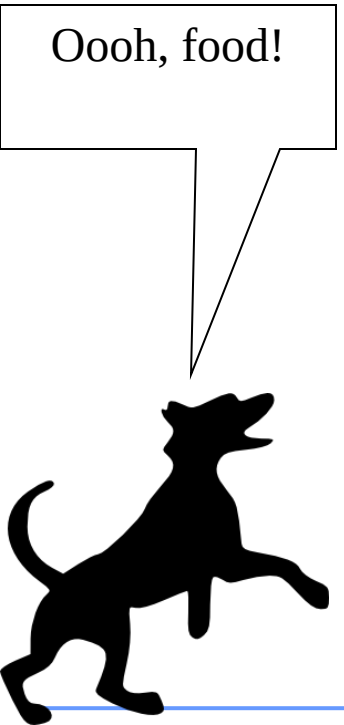
```
acquire(houselock);
Nap_on_couch();
Release(houselock);
```





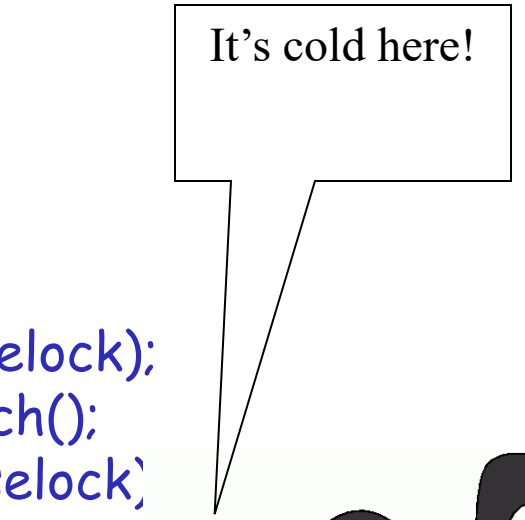
Spinlocks


```
acquire(int *lock) {
   while(test_and_set(lock) == 1)
            /* do nothing */;
}
release(int *lock) { *lock = 0;  }
```




```
acquire(houselock);
Jump_on_the_couch();
Be_goofy();
release(houselock);
```

```
acquire(houselock);
Nap_on_couch();
Release(houselock);
```



X86基于TSL的自旋锁汇编代码

```
enter_region:
        TSL REGISTER,LOCK                | copy lock to register and set lock to 1
        CMP REGISTER,#0                  | was lock zero?
        JNE enter_region                 | if it was non zero, lock was set, so loop
        RET | return to caller; critical region entered


leave_region:
        MOVE LOCK,#0                     | store a 0 in lock
        RET | return to caller
```

硬件方案3：使用swap指令

- Swap（对换）指令与TSL指令一样，是不会被中断的原子指令，其功能是交换两个字的内容，其语义如下：

```
Swap(boolean *a, Boolean *b)
{
    Boolean temp;
    Temp = *a;
    *a = *b;
    *b = temp;
}
```

- 在INTEL8086中为XCHG。

硬件方案3：使用swap指令

使用Swap指令实现进程互斥的描述如下：

```
Boolean k = true;                    // 初始化为1
Boolean use = false;                 // 初始资源空闲
while（k != 0）
        Swap（&use, &k）；           //进入区
Critial_region（）；                 //临界区
Use = 0;                             //退出区
Other_region（）；                   //剩余区
```

- 采用Swap指令与采用TSL指令类似，也会由于循环对换两个变量，造成忙等的情况。

MIPS中的spinlock

指令说明	指令定义	Alpha	MIPS64
寄存器和内存之间原子交换（用在锁和信号量上）	Temp<---Rd; Rd<---Mem[x]; Mem[x]<---Temp	LDL/Q_L; STL/Q_C	LL; SC ———

- MIPS 提供了 LL（Load Linked Word） 和 SC（Store Conditional Word） 这两个汇编指令来实现Spinlock(对共享资源的保护)。
- Read-modify-write 操作:当我们对某个Mem进行操作时, 我们首先从该 Mem中读回(read) data, 然后对该 data进行修改(modify)后, 再写回(write)该Mem。

MIPS中的spinlock

- LL 指令的功能是从内存中读取一个字,以实现接下来的 RMW（Read-Modify-Write） 操作;
- SC 指令的功能是向内存中写入一个字,以完成前面的 RMW 操作。
- LL/SC 指令的独特之处:
  - 当使用 LL 指令从内存中读取一个字后，如 *LL d, off(b)*，处理器会记住 LL 指令的这次操作（会在 CPU 的寄存器中设置一个不可见的 bit 位），同时 LL 指令读取的地址 *off(b)* 也会保存在处理器的寄存器中。
  - SC 指令，如 *SC t, off(b)*，会检查上次 LL 指令执行后的 RMW 操作是否是原子操作（即不存在其它对这个地址的操作），如果是原子操作，则 *t* 的值将会被更新至内存中，同时 *t* 的值也会变为1，表示操作成功；反之，如果 RMW 的操作不是原子操作（即存在其它对这个地址的访问冲突），则 *t* 的值不会被更新至内存中，且 *t* 的值也会变为0，表示操作失败。

MIPS中的spinlock

```
Spin_Unlock(lockkey)
        sync
        sw zero, lockkey   //给 lockKey 赋值为 0，
                           表示这个锁被释放
```

```
Spin_Lock(lockkey)      // lockKey 是共享资源锁
1:
        ll t0, lockkey      //将 lockKey 读入t0寄存器中
        bnez t0, 1b         //比较 lockKey 是否空闲，如果该锁不可
                              用的话则跳转到1：
        li t0, 1            //给 t0 寄存器赋值为1
        sc t0, lockkey      //将 t0 寄存器的值保存入 lockKey 中，
                              并返回操作结果于 t0 寄存器中。
        beqz t0, 1b         //判断 t0 寄存器的值，为0表示 li的操作失
                              败，则返回 1 :重新开始；为1表示 li的操
                              作成功。
        sync                // Sync 是内存操作同步指令，用来保证 sync
                              之前对于内存的操作能够在 sync 之后的指
                              令开始之前完成。
```

几个算法的共性问题

- 无论是软件解法（如Peterson）还是硬件（如TSL或XCHG）解法都是正确的，它们都有一个共同特点：当一个进程想进入临界区时，先检查是否允许进入，若不允许，则该进程将原地等待，直到允许为止。

1. 忙等待：浪费CPU时间

2. 优先级反转：低优先级进程先进入临界区，高优先级进程一直忙等

优先级反转

- 初始： greenbusy=redbusy=false

```
greenbusy = true
while redbusy:
    do_nothing()
if fridge_empty():
        buy_milk()
greenbusy = false
```

```
redbusy = true
if not greenbusy and
    fridge_empty():
        buy_milk()
redbusy = false
```

内容提要

- 同步与互斥问题
- 基于忙等待的互斥方法
- 基于信号量的同步方法
- 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

同步实现的基本思路

- 同步中，进程经常需要等待某个条件的发生，如果使用忙等待的解决方案，势必浪费大量CPU时间。
- 解决方法：将忙等变为阻塞，可使用两条进程间的通信原语：Sleep和Wakeup
  - Sleep原语将引起调用进程的阻塞，直到另外一个进程用Wakeup原语将其唤醒。很明显，wakeup原语的调用需要一个参数——被唤醒的进程ID。

E.g.银行排队

- 忙等：看到队很长，坚持排队
- 阻塞：看到队很长，先回家歇会儿（**sleep**），有空柜台了，大堂经理（**scheduler**）电话通知再过来（**wakeup**）

信号量

- 信号量是Dijkstra1965年提出的一种方法，它使用一个整型变量来累计唤醒次数，供以后使用。在他的建议中引入了一个新的变量类型，称作信号量（semaphore）。
- Dijkstra建议设立两种操作，P(S)和V(S)操作。P、V分别是荷兰语的test(proberen)和increment(verhogen)。P操作也称为semWait或Down操作，V操作也称为semSignal或Up操作。PV操作属于进程的低级通信。
- 信号量是一类特殊的变量，程序对其访问都是原子操作，且只允许对它进行P(信号变量)和V(信号变量)操作。

信号量的定义

- 信号量：一个确定的二元组(s, q)，其中s是一个具有非负初值的整型变量，q是一个初始状态为空的队列.
  - s为正，则该值等于发P操作后可立即执行的进程的数量；s为0，那么发出P操作后进程被继续执行完；s为负，那么发出P操作后的进程被阻塞，｜s｜是被阻塞的进程数。
  - q是一个初始状态为空的队列，当有进程被阻塞时就会进入此队列。

信号量的分类

二元信号量和一般信号量

- 二元信号量：取值仅为“0”或“1”，主要用作实现互斥；
- 一般信号量：初值为可用物理资源的总数，用于进程间的协作同步问题。

强信号量和弱信号量

- 强信号量：进程从被阻塞队列释放时采取FIFO
  - Guarantee freedom from starvation
- 弱信号量：没有规定进程从阻塞队列中移除顺序
  - Will not guarantee freedom from starvation

二元信号量机制

- 通常使用二元信号量的PV操作实现两个进程的互斥。
- 应用时应注意：
  - 每个进程中用于实现互斥的P、V操作必须成对出现，先做P操作，进临界区，后做V操作，出临界区。
  - P、V操作应分别紧靠临界区的头尾部，临界区的代码应尽可能短，不能有死循环。
  - 互斥信号量的初值一般为1。

```
P(S) :while S<=0 do skip
          S:=S-1;

V(S) :S:=S+1;
```

一般信号量的操作

- 一个信号量可能被初始化为一个非负整数.
- semWait 操作（P操作）使信号量减1。若值为负，则执行semWait的进程被阻塞。否则进程继续执行。
- semSignal操作（V操作）使信号量加1。若值小于或等于零，则被semWait操作阻塞的进程被解除阻塞。
- 除此之外，没有任何办法可以检查或操作信号量!

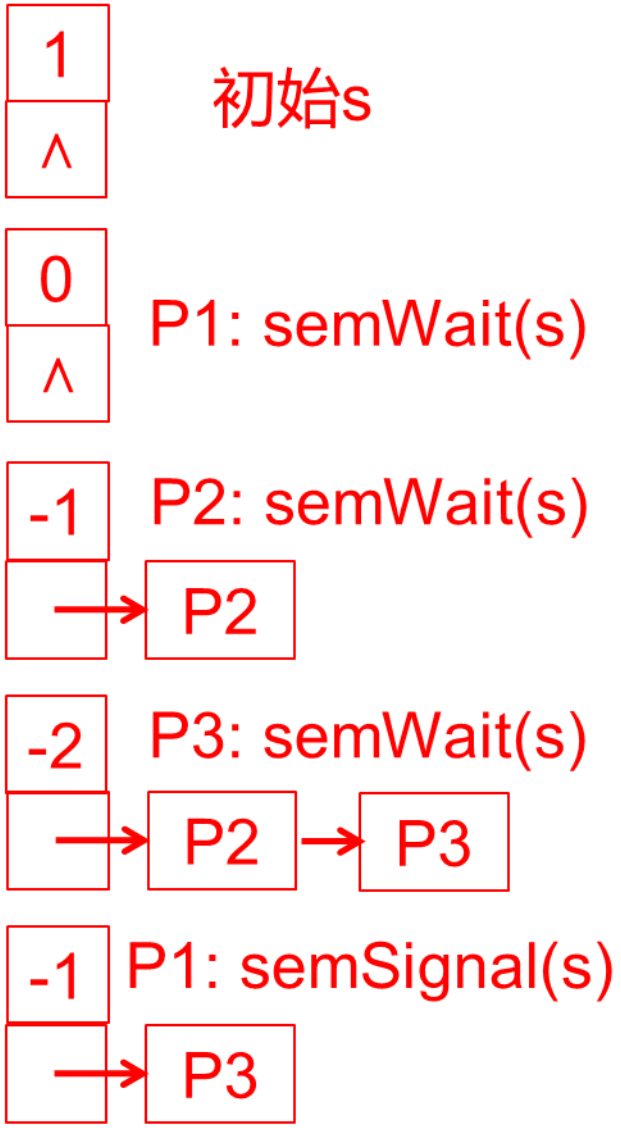
```
void semSignal(semaphore s)
{
        s.count++;
        if (s.count<= 0) {
          /* remove a process P from s.queue */;
          /* place process P on ready list */;
        }
}
```

```
void semWait(semaphore s)
{
        s.count--;
        if (s.count < 0) {
          /* place this process in s.queue */;
          /* block this process */;
        }
}
```

一般信号量的结构

- 信号量的数据结构，包含:
  - 一个初始值为正的整数: s.count
  - 一个初始为空的队列: s.queue
- 其上定义了三个原子操作
  - 初始化s
  - 发送信号s: semSignal(s)，(V, Up)
  - 接收信号s: semWait(s)，(P, Down)

```
struct semaphore {
        int count;
        queueType queue;
};
```



物理意义

- S.value为正时表示资源的个数
- S.value为负时表示等待进程的个数
- P操作分配资源
- V操作释放资源。

信号量机制的实现

- 原子性问题；
  - 关中断、TS指令等
- PCB链表形式。

信号量的实现

```
classSemaphore
{  int count;
   WaitQueue queue;
}
```

```
void semWait(semaphore s)
{
        s.count--;
        if (s.count < 0) {
          /* place this process in s.queue */;
          /* block this process */;
        }
}
```

```
void semSignal(semaphore s)
{
        s.count++;
        if (s.count<= 0) {
          /* remove a process P from s.queue */;
          /* place process P on ready list */;
        }
}
```

信号量机制的实现

- 指令实现方式：

```
semWait(s)
{
   while (compare_and_swap(s.flag, 0 , 1) == 1)
      /* do nothing */;
   s.count--;
   if (s.count < 0) {
      /* place this process in s.queue*/;
      /* block this process (must also set
s.flag to 0) */;
   }
   s.flag = 0;
}

semSignal(s)
{
   while (compare_and_swap(s.flag, 0 , 1) == 1)
      /* do nothing */;
   s.count++;
   if (s.count<= 0) {
      /* remove a process P from s.queue */;
      /* place process P on ready list */;
   }
   s.flag = 0;
}
```

有忙等待！但时间很短！

(a) Compare and Swap Instruction

- 中断实现方式：

```
semWait(s)
{
   inhibit interrupts;
   s.count--;
   if (s.count < 0) {
      /* place this process in s.queue */;
      /* block this process and allow inter-
rupts*/;
   }
   else
      allow interrupts;
}

semSignal(s)
{
   inhibit interrupts;
   s.count++;
   if (s.count<= 0) {
      /* remove a process P from s.queue */;
      /* place process P on ready list */;
   }
   allow interrupts;
}
```

禁止中断！但时间很短！

(b) Interrupts

信号量机制的实现

- 指令实现方式：

```
semWait(s)
{
   while (compare_and_swap(s.flag, 0 , 1) == 1)
      /* do nothing */;
   s.count--;
   if (s.count < 0) {
      /* place this process in s.queue*/;
      /* block this process (must also set
s.flag to 0) */;
   }
   s.fl
}

semSign
{
   while
      /* do nothing */;
   s.count++;
   if (s.count<= 0) {
      /* remove a process P from s.queue */;
      /* place process P on ready list */;
   }
   s.flag = 0;
}
```

有忙等待！但时间很短！

- 中断实现方式：

```
semWait(s)
{
   inhibit interrupts;
   s.count--;
   if (s.count < 0) {
      /* place this process in s.queue */;
      /* block this process and allow inter-
rupts*/;
   }
   inhibit interrupts;
   s.count++;
   if (s.count<= 0) {
      /* remove a process P from s.queue */;
      /* place process P on ready list */;
   }
   allow interrupts;
}
```

禁止中断！但时间很短！

CAS的语义是“我认为V的值应该为A，如果是，那么将V的值更新为B，否则不修改并告诉V的值实际为多少”

(a) Compare and Swap Instruction　　(b) Interrupts

补充练习

```
        movl sem, %ecx
        decl 0(%ecx)
        js  down_failed
        ……
down_failed: block
```

信号量在并发中的典型应用

应用	描述
互斥 (Mutual exclusion)	可以用初始值为1的信号量来实现进程间的互斥。一个进程在进入临界区之前执行semWait操作，退出临界区后再执行一个semSignal操作。这是实现临界区资源互斥使用的一个二元信号量。
有限并发 (Bounded concurrency)	是指有n个进程并发的执行一个函数或者一个资源。一个初始值为c的信号量可以实现这种并发。
进程同步 (Synchronization)	是指当一个进程P_i想要执行一个a_i操作时，它只在进程P_j执行完a_j后，才会执行a_i操作。可以用信号量如下实现：将信号量初始为0，P_i执行a_i操作前执行一个semWait操作；而P_j执行a_j操作后，执行一个semSignal操作。

信号量在互斥中的使用

```
var sem_CS : semaphore := 1;
Parbegin
    repeat
        wait (sem_CS);
        { Critical Section }
        signal (sem_CS);
        { Remainder of the cycle }
    forever;
Parend;
end.
```

Process P_i

```
repeat
    wait (sem_CS);
    { Critical Section }
    signal (sem_CS);
    { Remainder of the cycle }
forever;
```

Process P_j

信号量在有限并发中的应用

```
var sem_CS : semaphore := c
Parbegin
    repeat
        wait (sem_CS);
        { Critical Section }
        signal (sem_CS);
        { Remainder of the cycle }
    forever;
Parend;
end.
```

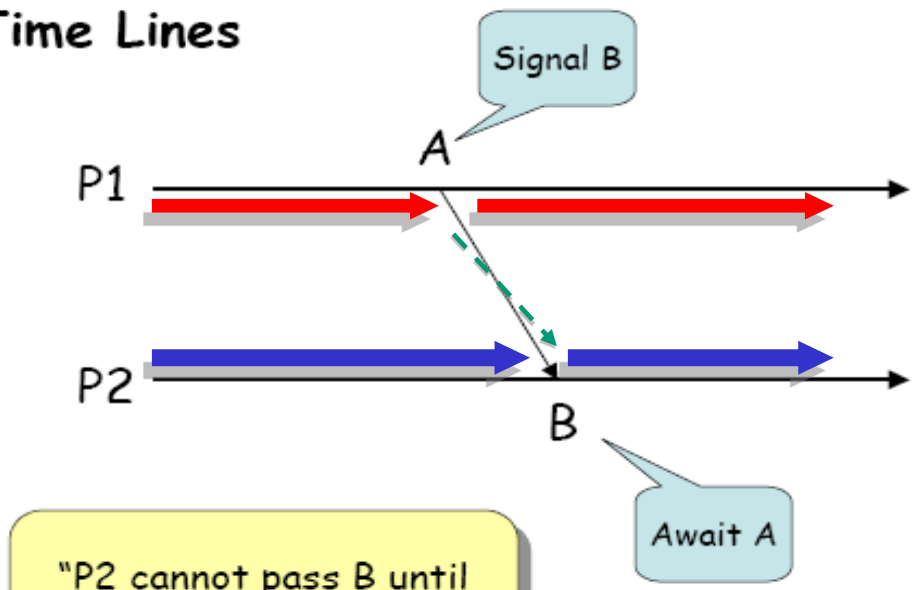
Process P_i

```
repeat
    wait (sem_CS);
    { Critical Section }
    signal (sem_CS);
    { Remainder of the cycle }
forever;
```

Process P_j

将信号量想成token是很有帮助的：
wait对应取token，signal对应释放token

信号量在进程同步中的应用




```
var sync : semaphore := 0;
Parbegin
        . . .                                   . . .
        wait (sync);                            { Performaction a_j }
        { Performaction a_i }                   signal (sync);
Parend;
end.
```

Process P_i *Process P_j*

使用信号量实现汇合 (Rendezvous)

- 使用信号量实现进程A和进程B的汇合(Rendezvous)。使得a1永远在b2之前，而b1永远在a2之前。
  - a1和b1的次序不加限制
- 基本的同步模式：使得两个进程在执行过程中一点汇合，直到两者都到后才能继续执行。

Thread A

```
1 statement a1
2 statement a2
```

Thread B

```
1 statement b1
2 statement b2
```

使用信号量实现汇合(Rendezvous)

- 使用信号量实现进程A和进程B的汇合(Rendezvous)。使得a1永远在b2之前，而b1永远在a2之前。
- 提示：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 statement a2
```

Thread B

```
1 statement b1
2 statement b2
```

使用信号量实现汇合(Rendezvous)

- 使用信号量实现进程A和进程B的汇合(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 aArrived.signal()
3 bArrived.wait()
4 statement a2
```

Thread B

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

使用信号量实现汇和 (Rendezvous)

- 使用信号量实现进程A和进程B的汇合(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

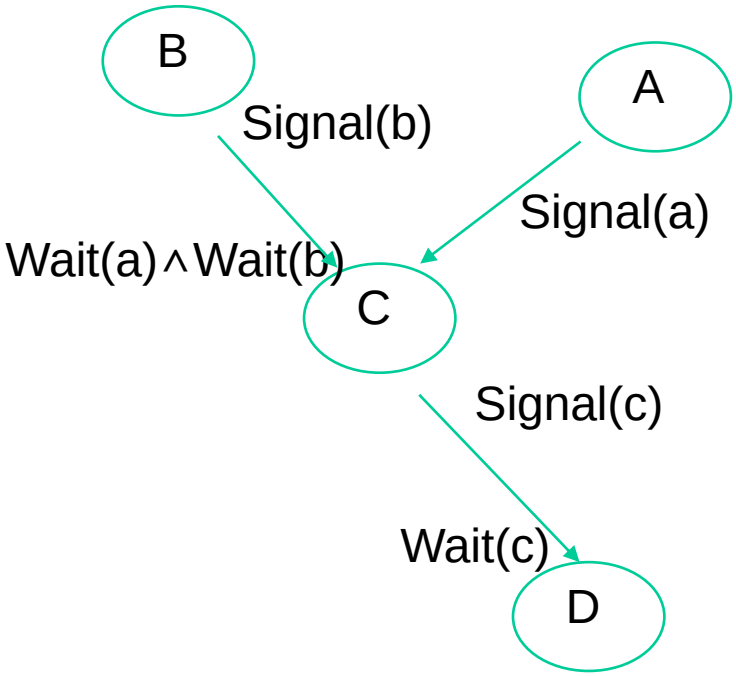
```
1 statement a1
2 bArrived.wait()
3 aArrived.signal()
4 statement a2
```

Thread B

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

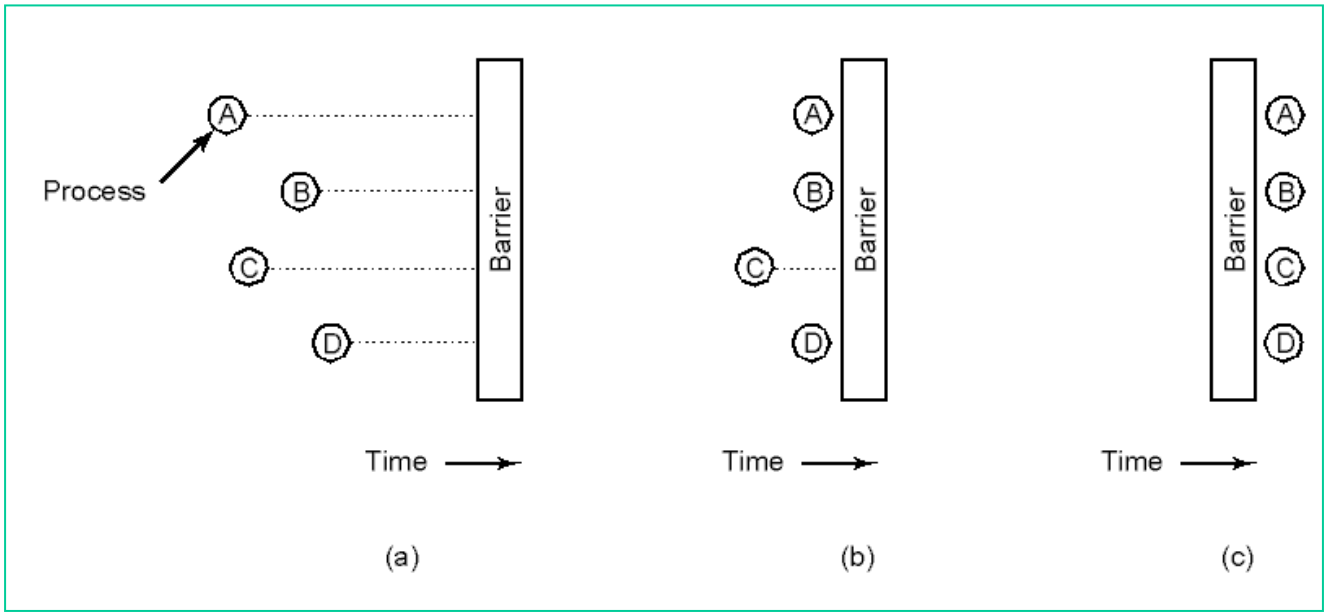
需要两次程序切换

前趋关系



多进程同步原语：屏障Barriers

- 用于进程组的同步



- 思考：如何使用信号量实现Barrier?

多进程同步原语：屏障Barriers

- 对汇合（rendezvous）进行泛化，使其能够用于多个进程，用于进程组的同步
  - 科学计算中的迭代
  - 深度学习的卷进神经网络迭代
  - GPU编程中的渲染算法迭代

- 思考：如何使用信号量实现Barrier?

多进程同步原语：屏障Barriers

- 对rendezvous进行泛化，使其能够用于多个进程，用于进程组的同步
- 提示：

```
1 n = the number of threads
2 count = 0
3 mutex = Semaphore(1)
4 barrier = Semaphore(0)
```

- Count记录到达汇合点的进程数。mutex保护count，barrier在当所有进程到达之前都是0或者负值。到达后取正值。
- 思考：如何使用信号量实现Barrier?

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 //到达汇合点的进程数
  - mutex = Semaphore(1) //保护count
  - barrier = Semaphore(0)//进程到达之前都是0或者负值。到达后取正值

  - mutex.wait()
  - count = count + 1
  - mutex.signal()

  - if count == n: barrier.signal() # 唤醒一个进程
  - barrier.wait()
  - barrier.signal() # 一旦进程被唤醒，有责任唤醒下一个进程

补充练习

```
1  rendezvous
2
3  mutex.wait()
4      count += 1
5  mutex.signal()
6
7  if count == n: turnstile.signal()
8
9  turnstile.wait()
10 turnstile.signal()
11
12 critical point
13
14 mutex.wait()
15     count -= 1
16 mutex.signal()
17
18 if count == 0: turnstile.wait()
```

进程同步/互斥类问题的求解

- 解题步骤：
  - 分析问题，确定哪些操作是并发的。在并发的操作中，哪些是互斥的，哪些是同步的。
    - 多个进程操作同一个临界资源就是互斥
    - 多个进程要按一定的顺序执行就是同步
  - 根据同步和互斥规则设置信号量，说明其含义和初值；
  - 用P、V操作写出程序描述。

“信号量集”机制

```
Process A:
   P(Dmutex);
   P(Emutex);
```

```
Process B:
   P(Emutex);
   P(Dmutex);
```

Dmutex, Emutex = 1;

Process A: P(Dmutex);
Process B: P(Emutex);
Process A: P(Emutex);
Process B: P(Dmutex);

“信号量集”机制

当利用信号量机制解决了单个资源的互斥访问后，我们讨论如何控制同时需要多个资源的互斥访问。信号量集是指同时需要多个资源时的信号量操作。

- “AND型”信号量集
- 一般信号量集

AND型信号量集机制

- 基本思想：将进程需要的所有共享资源一次全部分配给它；待该进程使用完后再一起释放。

```
SP(S1, S2, … ,Sn)
  if S1=>1 and … and Sn=>1 then
     for I :=1 to n do
        Si := Si - 1;
     endfor
  else
   wait in Si;
  endif
```

```
SV(S1, S2, … ,Sn)
  for I :=1 to n do
        Si := Si + 1;
        wake waited process
  endfor
```

一般“信号量集”机制

- 基本思想：在AND型信号量集的基础上进行扩充：进程对信号量Si的测试值为ti（用于信号量的判断，即Si >= ti，表示资源数量低于ti时，便不予分配），占用值为di（用于信号量的增减，即Si = Si - di和Si = Si +di）

```
SP(S1, t1, d1, … , Sn, tn, dn)
  if S1=>t1 and … and Sn=>tn then
     for I :=1 to n do
        Si := Si - di;
     endfor
  else
   wait in Si;
  endif
```

```
SV(S1, d1, … ,Sn, dn)
  for I :=1 to n do
        Si := Si + di;
        wake waited process
  endfor
```

一般“信号量集”机制

几个例子：

- SP(S, d, d)：表示每次申请d个资源，当资源数量少于d个时，便不予分配。
- SP(S, 1, 1)：表示互斥信号量。
- 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)。

P.V操作的优缺点

- 优点：
  - 简单，而且表达能力强（用P.V操作可解决任何同步互斥问题）
- 缺点：
  - 不够安全；P.V操作使用不当会出现死锁；遇到复杂同步互斥问题时实现复杂

管程 (Monitor)

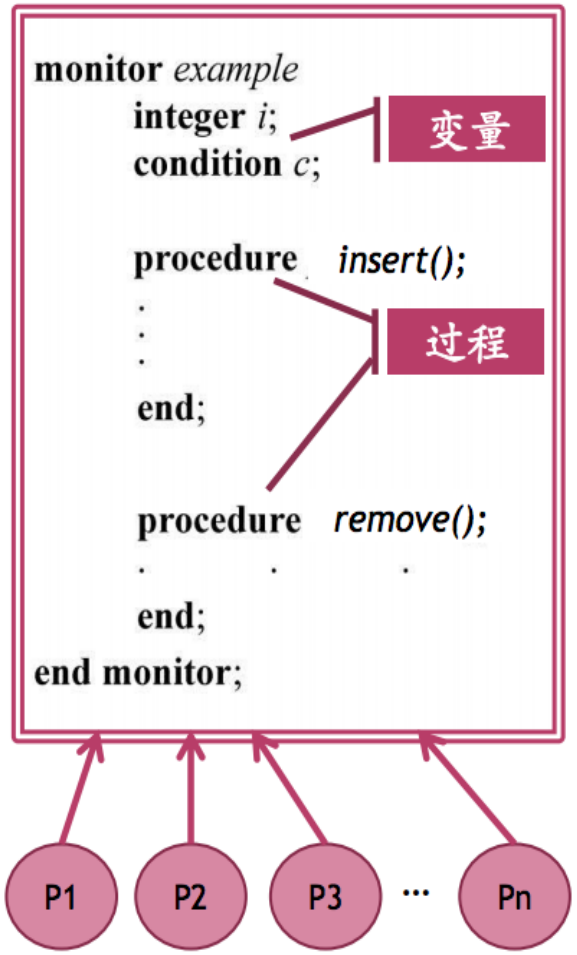
- 在引入了信号量及 PV 操作之后，为什么又出现了一种新的同步机制管程呢？ 主要的原因是：
  - 信号量 这种机制具有一些缺点，如用信号量及 PV 操作解决问题时，程序编写需要很高的技巧。
  - 如果没有合理地安排 PV 操作的位置，就会导致一些出错的结果，如出现死锁等问题。
- 是在程序设计语言当中引入的一个成分，是一种高级同步机制

管程的定义和组成

1973年，Hoare和Hanson所提出，“一个管程定义了一个数据结构和能为并发进程所执行（在该数据结构上）的一组操作，这组操作能同步进程和改变管程中的数据”。

管程由四部分组成：

1. 管程的名称；
2. 局部于管程内部的共享数据结构（变量）说明；
3. 对该数据结构进行操作的一组互斥执行的过程；
4. 对局部于管程内部的共享数据设置初始值的语句。



管程要解决的问题

- 作为一种同步机制，需要解决三个问题：
  - 互斥：只能有一个进程可对其内部数据结构进行相应的操作，即管程进入是互斥的。由编译器来保证（管程是一个语言机制 ）。
  - 同步：通过设置条件变量（CV） 以及在条件变量上实施的 wait 和 signal 操作，它可以使一个进程或线程，当条件不满足/满足的时候在条件变量上等待/唤醒。
  - 条件变量：为了区别等待的不同原因，管程引入了条件变量。不同的条件变量，对应不同原因的进程阻塞等待队列，初始时为空。条件变量上能作wait和signal原语操作，若条件变量名为X，则调用同步原语的形式为wait(X)和signal(X)。

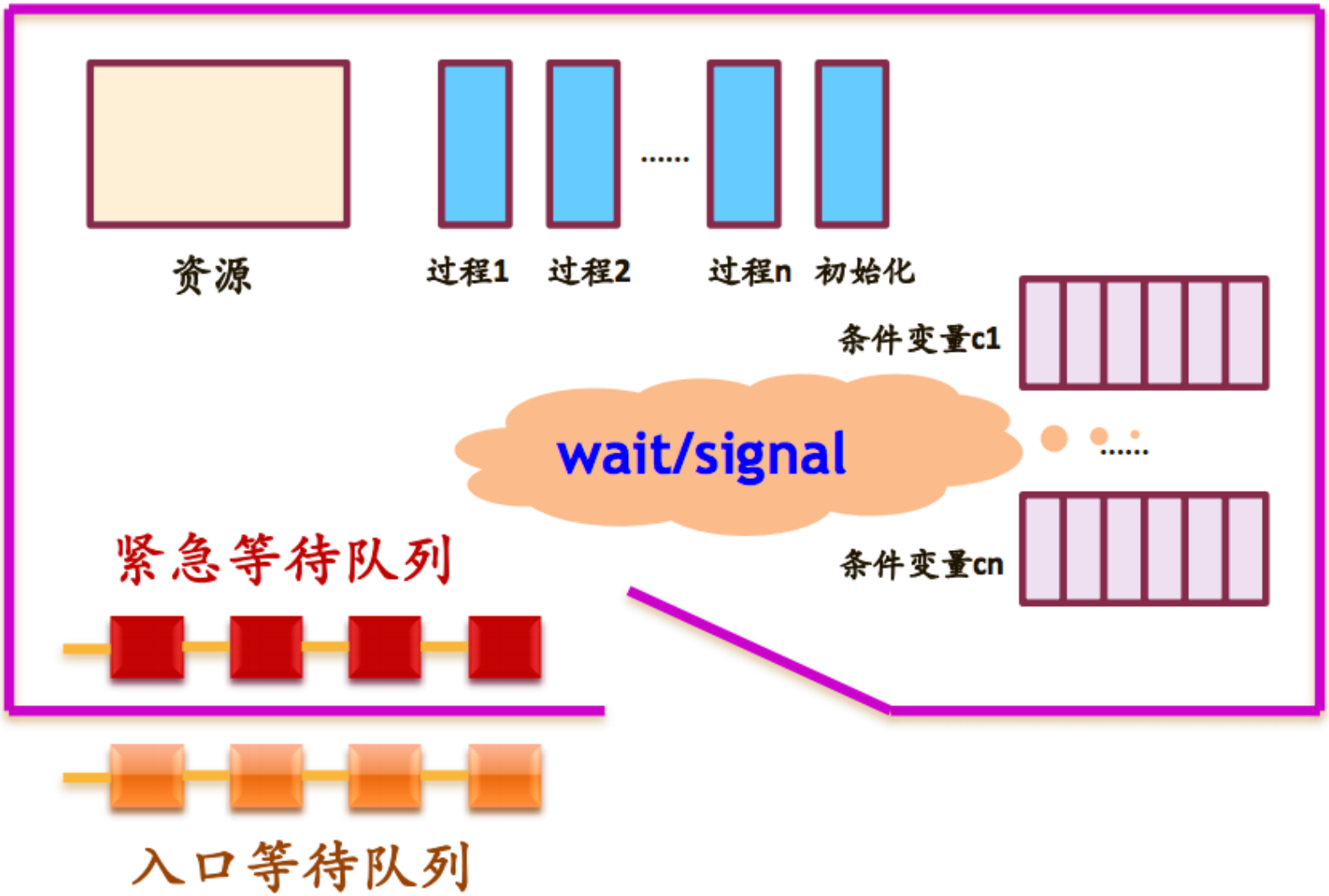
条件变量与信号量的区别

- 条件变量的值不可增减，P-V操作的信号量值可增减
  - wait操作一定会阻塞当前进程；但P操作只有当信号量的值小于0时才会阻塞。
  - 如果没有等待的进程，signal将丢失；而V操作增加了信号量的值，不会丢失。
- 访问条件变量必须拥有管程的锁

多个进程同时在管程中出现

- 场景：
  - 当一个进入管程的进程执行等待操作时，它应当释放管程的互斥权。
  - 当后面进入管程的进程执行唤醒操作时（例如P唤醒Q），管程中便存在两个同时处于活动状态的进程。
- 三种处理方法：
  - P等待Q执行（Hoare管程）；
  - Q等待P继续执行（MESA管程）；
  - 规定唤醒操作为管程中最后一个可执行的操作（Hansen管程，并发pascal）。

Hoare管程



Hoare管程

- **入口等待队列**(entry queue)：因为管程是互斥进入的，所以当一个进程试图进入一个已被占用的管程时它应当在管程的入口处等待，因而在管程的入口处应当有一个进程等待队列，称作入口等待队列。
- **紧急等待队列**：如果进程P唤醒进程Q，则P等待Q继续，如果进程Q在执行又唤醒进程R，则Q等待R继续，…，如此，在管程内部，由于执行唤醒操作，可能会出现多个等待进程（已被唤醒，但由于管程的互斥进入而等待），因而还需要有一个进程等待队列，这个等待队列被称为紧急等待队列。它的优先级应当高于入口等待队列的优先级。

Hoare管程的条件变量

- 由于管程通常是用于管理资源的，因而在管程内部，应当存在某种等待机制。当进入管程的进程因资源被占用等原因不能继续运行时使其等待。为此在管程内部可以说明和使用一种特殊类型的变量----条件变量。
- 每个条件变量表示一种等待原因，并不取具体数值——相当于每个原因对应一个队列。

Hoare管程的同步原语

- 同步操作原语wait和signal：针对条件变量x，x.wait()将自己阻塞在x队列中，x.signal()将x队列中的一个进程唤醒。
  - x.wait()：如果紧急等待队列非空，则唤醒第一个等待者；否则释放管程的互斥权，执行此操作的进程排入x队列尾部（紧急等待队列与x队列的关系：紧急等待队列是由于管程的互斥进入而等待的队列，而x队列是因资源被占用而等待的队列）。
  - x.signal()：如果x队列为空，则相当于空操作，执行此操作的进程继续；否则唤醒第一个等待者，执行x.signal()操作的进程排入紧急等待队列的尾部。

Hoare管程：信号量定义(1)

- 每个管程必须提供一个用于互斥的信号量mutex（初值为1）。
- 进程调用管程中的任何过程时，应执行P(mutex)；进程退出管程时应执行V(mutex)开放管程，以便让其他调用者进入。
- 为了使进程在等待资源期间，其他进程能进入管程，故在wait操作中也必须执行V(mutex)，否则会妨碍其他进程进入管程，导致无法释放资源。

Hoare管程：信号量定义(2)

- 对每个管程，引入信号量next（初值为0），凡发出signal操作的进程应该用P(next)挂起自己，直到被释放进程退出管程或产生其他等待条件。
- 进程在退出管程的过程前，须检查是否有别的进程在信号量next上等待，若有，则用V(next)唤醒它。next-count（初值为0），用来记录在next上等待的进程个数。

Hoare管程：信号量定义(3)

- 引入信号量x-sem（初值为0），申请资源得不到满足时，执行P(x-sem)挂起。由于释放资源时，需要知道是否有别的进程在等待资源，用计数器x-count（初值为0）记录等待资源的进程数。
- 执行signal操作时，应让等待资源的诸进程中的某个进程立即恢复运行，而不让其他进程抢先进入管程，这可以用V(x-sem)来实现

Hoare管程的实现

1. 编译器对管程外部调用的入/出口的处理

```
P(mutex);          管程入口
   ...
Body of F          管程调用
   ...
If(next_count > 0)
   V(next);        释放使用signal的进程
Else
   V(mutex);       管程出口
```

2. X.Wait

```
x_count++;          使用wait的进程数
If(next_count>0)
   V(next);         唤醒之前使用signal的进程
else
   V(mutex);        退出管程
P(x_sem);           等待配对的signal
x_count--;
```

3. X.Signal

```
if(x_count>0){      使用signal的进程数
   next_count ++;
   V(x_sem);        唤醒使用wait的进程
   P(next);         等待配对的wait进程
   next_count--;
}
```

这些机制将由编译器自动完成，对编程人员是透明的！

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

进程间通信(Inter-Process-Comm)

- 低级通信：只能传递状态和整数值（控制信息），包括进程互斥和同步所采用的信号量和管程机制。
  缺点：
  - 传送信息量小：效率低，每次通信传递的信息量固定，若传递较多信息则需要进行多次通信。
  - 编程复杂：用户直接实现通信的细节，编程复杂，容易出错。
- 高级通信：适用于分布式系统，基于共享内存的多处理机系统，单处理机系统，能够传送任意数量的数据，可以解决进程的同步问题和通信问题，主要包括三类：管道、共享内存、消息系统。

IPC概述

- IPC历史
  - AT&T的贝尔实验室： System V IPC，通信进程局限在单个计算机内。
  - BSD的加州大学伯克利分校的伯克利软件发布中心：基于套接字（Socket）的进程间通信机制。
  - 电子电气工程协会（IEEE）： POSIX
- SolarisIPC
  - Solaris则把两者（SYSTEM V和BSD）都继承了下来，并用于不同场合。POSIX放在了函数库中。
  - 都有轻微变动和增加。
- WindowsIPC

IPC概述

- 管道（Pipe）及命名管道（Named pipe或FIFO）
- 消息队列（Message）
- 共享内存（Shared memory）
- 信号量（Semaphore）
- 套接字（Socket）
- 信号（Signal）

无名管道 (Pipe)

- 管道是半双工的，数据只能向一个方向流动；需要双方通信时，需要建立起两个管道；
- 只能用于父子进程或者兄弟进程之间（具有亲缘关系的进程）；
- 单独构成一种独立的文件系统：管道对于管道两端的进程而言，就是一个文件，但它不是普通的文件，它不属于某种文件系统，而是自立门户，单独构成一种文件系统，并且只存在在内存中。
- 数据的读出和写入：一个进程向管道中写的内容被管道另一端的进程读出。写入的内容每次都添加在管道缓冲区的末尾，并且每次都是从缓冲区的头部读出数据。

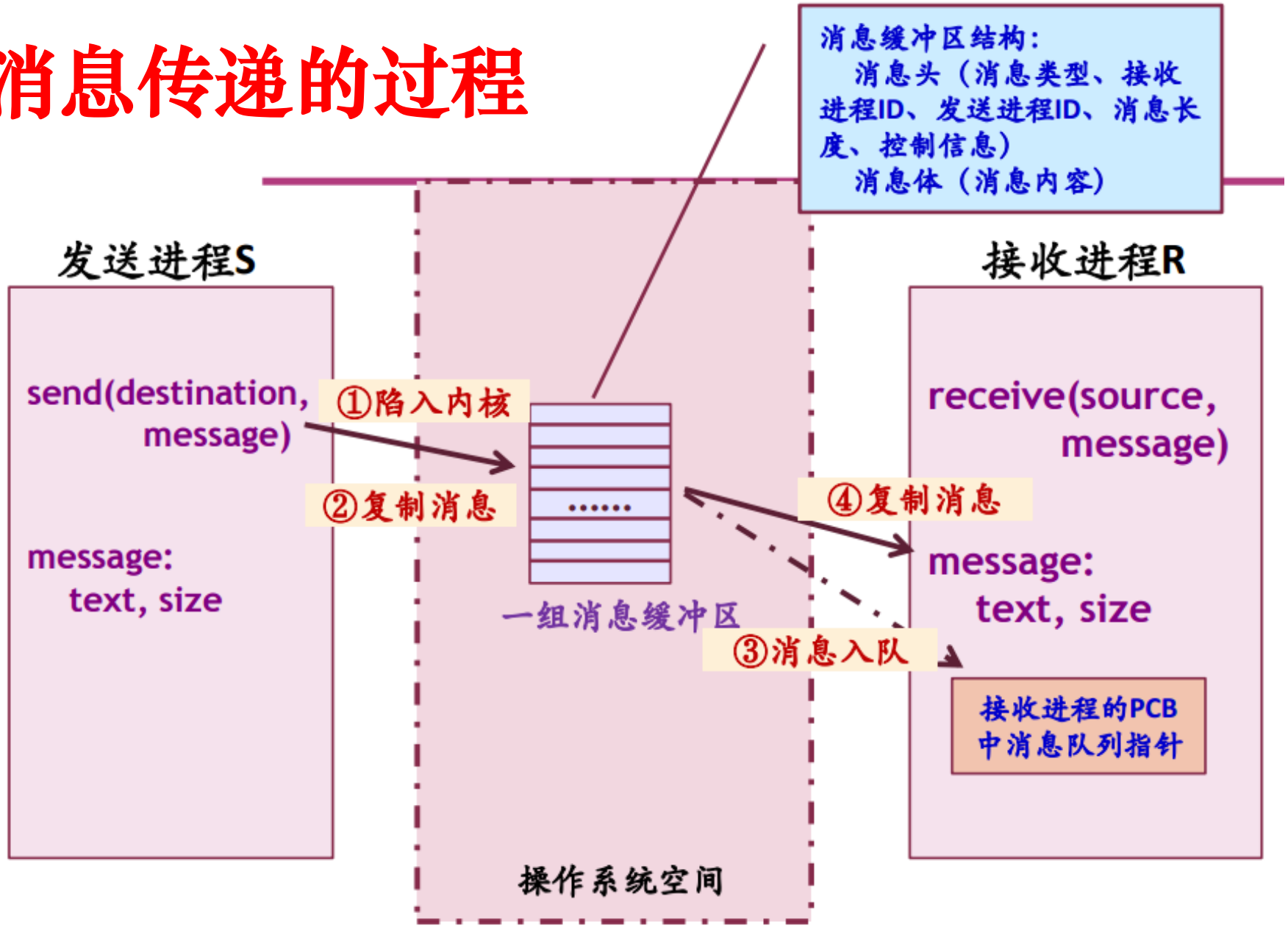
有名管道（Named Pipe或FIFO）

- 无名管道应用的一个重大限制是它没有名字，因此，只能用于具有亲缘关系的进程间通信，在有名管道提出后，该限制得到了克服。
- FIFO不同于管道之处在于它提供一个路径名与之关联，以FIFO的文件形式存在于文件系统中。这样，即使与FIFO的创建进程不存在亲缘关系的进程，只要可以访问该路径，就能够彼此通过FIFO相互通信（能够访问该路径的进程以及FIFO的创建进程之间），因此，通过FIFO不相关的进程也能交换数据。
- FIFO严格遵循先进先出（first in first out），对管道及FIFO的读总是从开始处返回数据，对它们的写则把数据添加到末尾。

消息传递（message passing）

- 消息传递——两个通信原语（OS系统调用）
  - **send (destination, &message)**
  - **receive(source, &message)**
- 调用方式
  - 阻塞调用
  - 非阻塞调用
- 主要问题：
  - 解决消息丢失、延迟问题（TCP协议）
  - 编址问题：mailbox

消息传递的过程


消息缓冲区结构：
消息头（消息类型、接收进程ID、发送进程ID、消息长度、控制信息）
消息体（消息内容）
发送进程S
接收进程R
send(destination, message)
①陷入内核
②复制消息
message:
text, size
一组消息缓冲区
④复制消息
receive(source, message)
message:
text, size
③消息入队
接收进程的PCB中消息队列指针
操作系统空间

用P-V操作实现Send原语

```
Send（destination, message）
{
 根据destination找接收进程;
 如果未找到，出错返回;

 申请空缓冲区P(buf-empty);
   P(mutex1);
     取空缓冲区;
   V(mutex1);

把消息从message处复制到空缓
冲区;
```

```
   P(mutex2);
    把消息缓冲区挂到接收进程
    的消息队列;
   V(mutex2);

   V(buf-full);
}
```

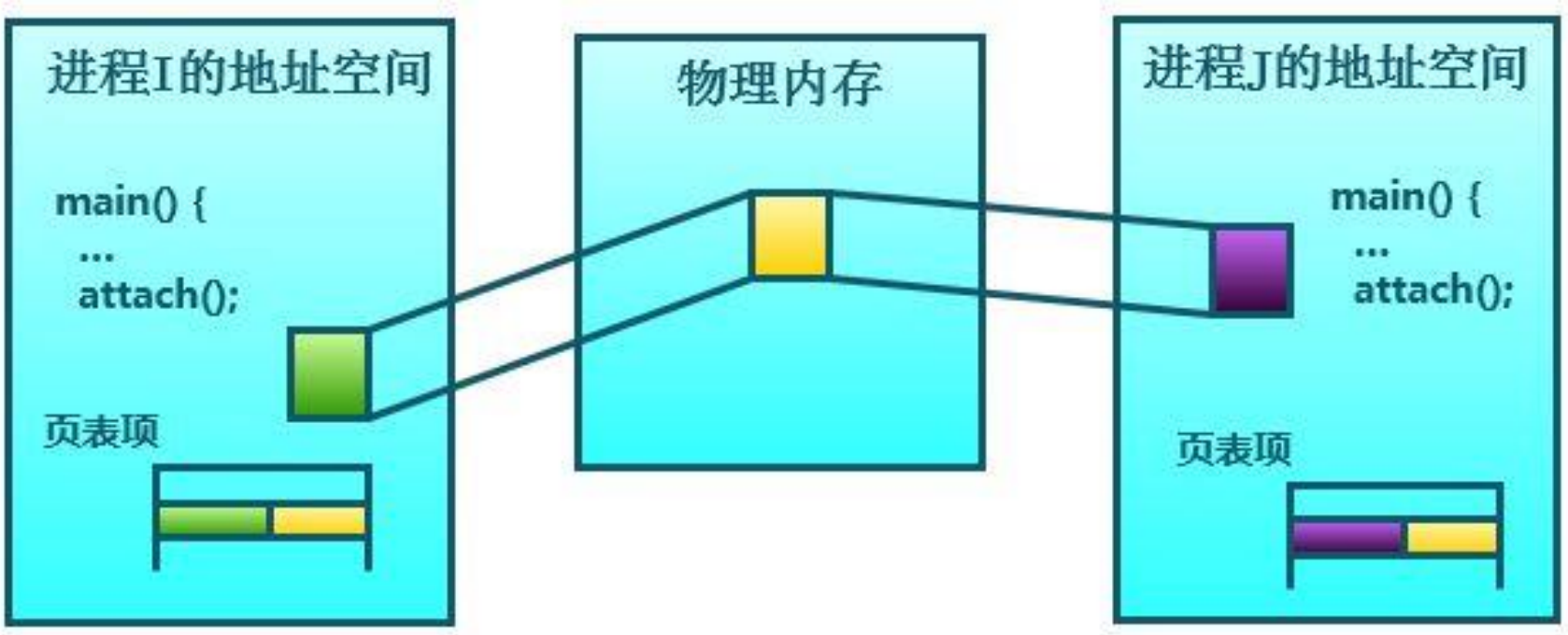
receive原语的实现?

信号量:
buf-empty初值为N
buf-full初值为0
mutex1初值为1
mutex2初值为1

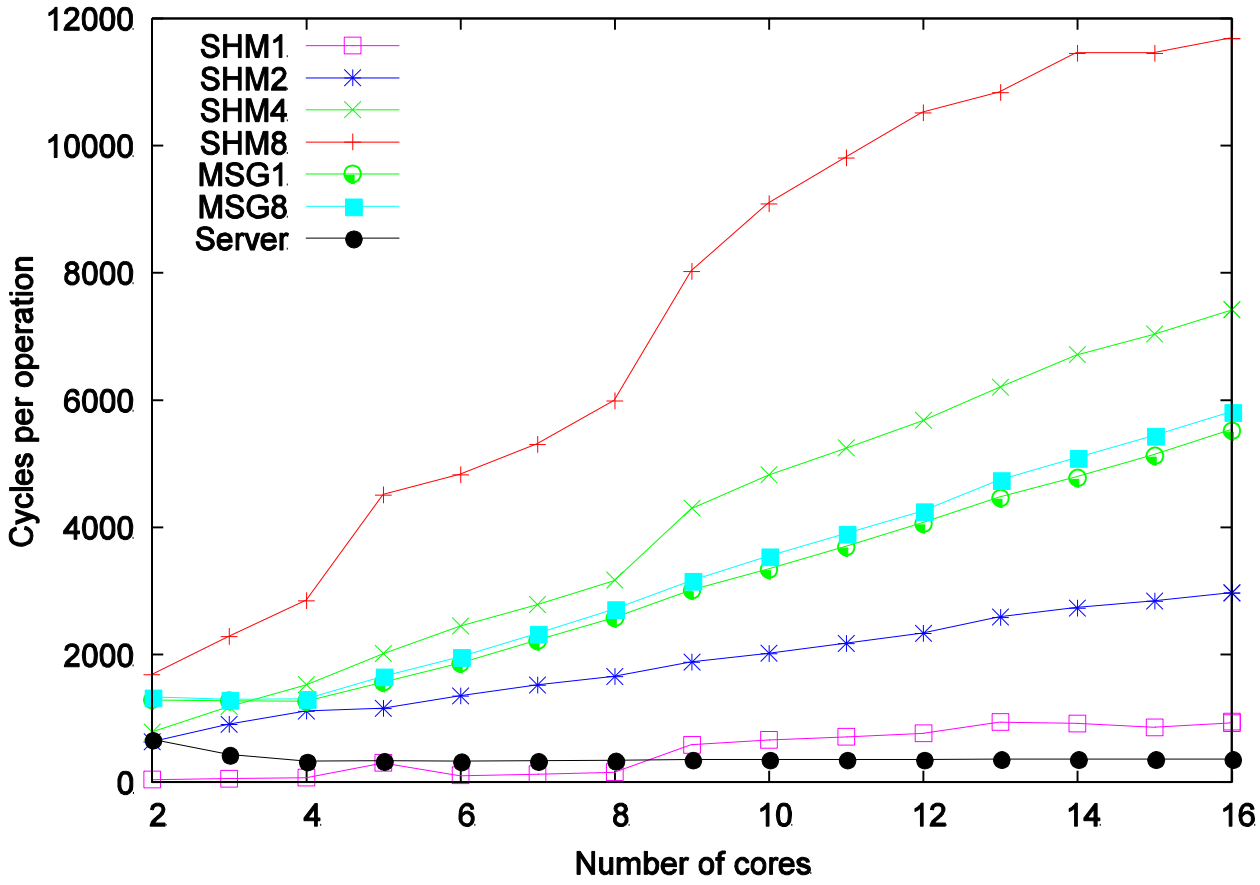
共享内存

- 共享内存是最有用的进程间通信方式，也是最快的IPC形式（因为它避免了其它形式的IPC必须执行的开销巨大的缓冲复制）。
- 两个不同进程A、B共享内存的意义是，同一块物理内存被映射到进程A、B各自的进程地址空间。
- 当多个进程共享同一块内存区域，由于共享内存可以同时读但不能同时写，则需要同步机制约束（互斥锁和信号量都可以）。
- 共享内存通信的效率高（因为进程可以直接读写内存）。
- 进程之间在共享内存时，保持共享区域直到通信完毕。

共享内存机制



Message Passing VS Memory Sharing



核数大于4时，消息传递性能优于共享存储

POSIX

Semaphores	Message Queues	Shared Memory
sem_open	mq_open	shm_open
sem_close	mq_close	shm_unlink
sem_unlink	mq_unlink	
sem_init	mq_getattr	
sem_destroy	mq_setattr	
sem_wait	mq_send	
sem_trywait	mq_receive	
sem_post	mq_notify	
sem_getvalue	mq_getvalue	

内容提要

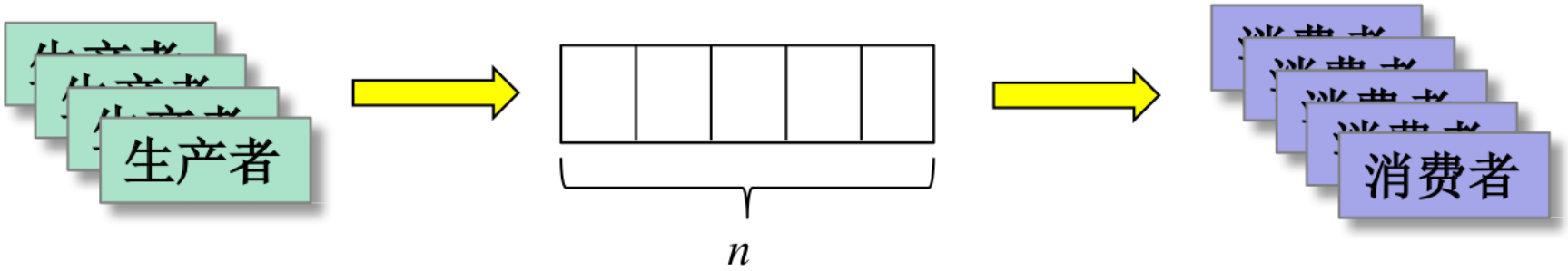
- 同步与互斥问题
- 基于忙等待的互斥方法
- 基于信号量的方法
- 基于管程的同步与互斥
- 进程通信的主要方法
- 经典的进程同步与互斥问题

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)

生产者－消费者问题

- 问题描述：若干进程通过有限的共享缓冲区交换数据。其中，“生产者”进程不断写入，而“消费者”进程不断读出；共享缓冲区共有N个；任何时刻只能有一个进程可对共享缓冲区进行操作。



生产者－消费者问题

- 两个隐含条件：

  1.消费者和生产者数量不固定。

  2.消费者和生产者不能同时使用缓存区。

- 行为分析：
  - 生产者：生产产品，放置产品(有空缓冲区)。
  - 消费者：取出产品(有产品)，消费产品。
- 行为关系：
  - 生产者之间:互斥(放置产品)
  - 消费者之间:互斥(取出产品)
  - 生产者与消费者之间：互斥(放/取产品) 同步(放置——取出)

用PV操作解决生产者/消费者问题

- 信号量设置：

  semaphore mutex =1;　　//互斥

  semaphore empty=N;　　//空闲数量

  semaphore full=0;　　//产品数量

- 实际上，full和empty是同一个含义：

  full + empty == N

用PV操作解决生产者/消费者问题

```
P(s)
{
   s.count --;
   if (s.count < 0)
   {
      该进程状态置为阻塞状态;
      将该进程插入相应的等待队列s.queue末尾;
      重新调度;
   }
}
```

down, semWait

```
V(s)
{
   s.count ++;
   if (s.count < = 0)
   {
      唤醒相应等待队列s.queue中等待的一个进程;
      改变其状态为就绪态，并将其插入就绪队列;
   }
}
```

up, semSignal

用PV操作解决生产者/消费者问题

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

用PV操作解决生产者/消费者问题

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)

如果两个V交换顺序呢？

完整的伪代码写法

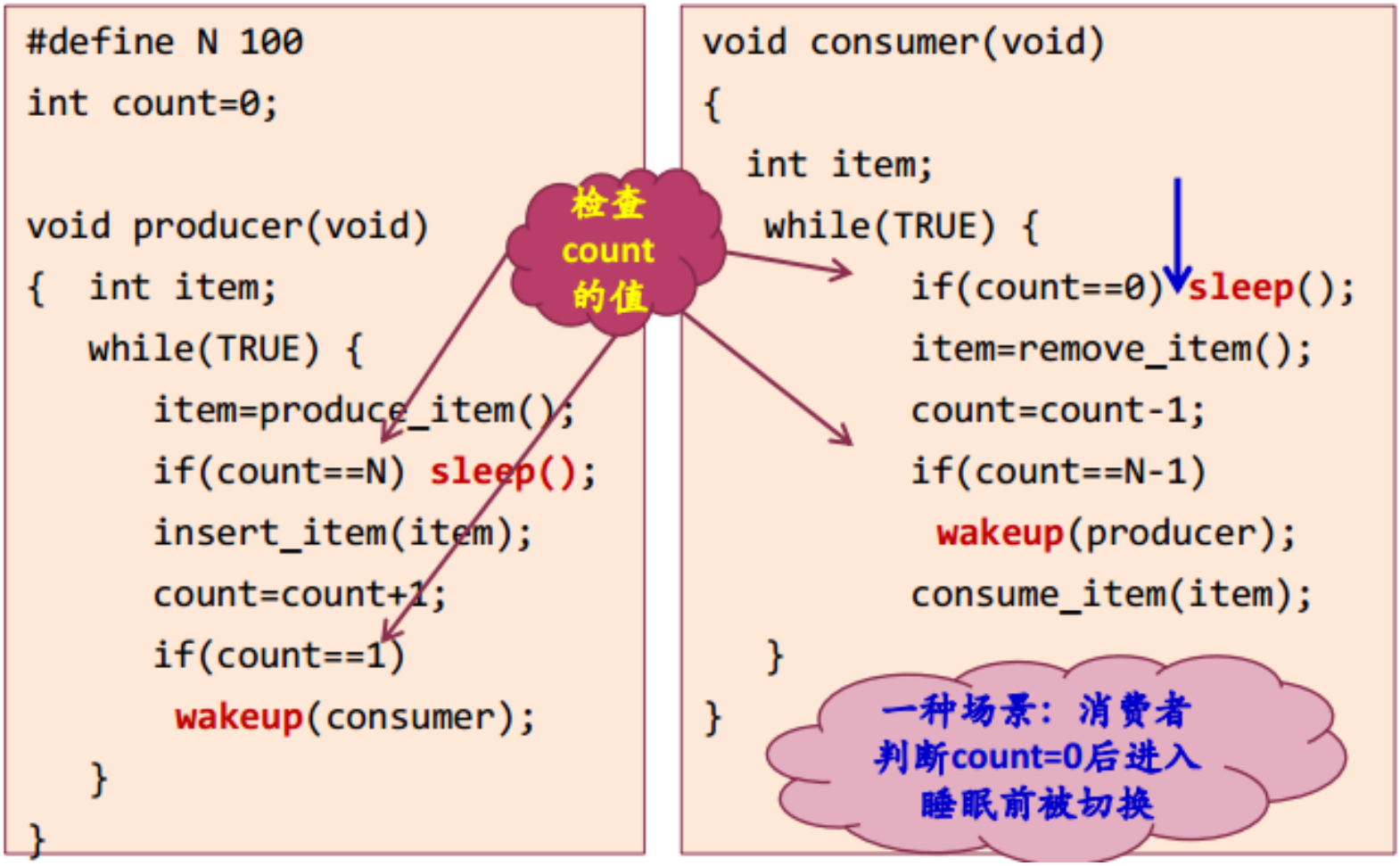
```
Semaphore full = 0;
Semaphore empty = n;
Semaphore mutex = 1;
ItemType buffer[0..n-1];
int in = 0, out =0;
```

```
main(){
    Cobegin
        producer();
        consumer();
    Coend
}
```

```
producer() {
    while(true){
        生产产品nextp;
        P(empty);
        P(mutex);
        buffer[in] = nextp;
        in = (in + 1) MOD n;
        V(mutex);
        V(full);
    }
}
```

```
consumer() {
    while(true){
        P(full);
        P(mutex);
        nextc = buffer[out];
        out = (out + 1) MOD n;
        V(mutex);
        V(empty);
        消费nextc中的产品
    }
}
```

采用Sleep和Wakeup原语的方法



用管程解决生产者消费者问题

```
monitor ProducerConsumer
    condition full, empty;
    integer count;

    procedure insert (item: integer);
    begin
        if count == N then wait(full);
        insert_item(item); count++;
        if count ==1 then signal(empty);
    end;

    function remove: integer;
    begin
        if count ==0 then wait(empty);
        remove = remove_item; count--;
        if count==N-1 then signal(full);
    end;

    count:=0;
end monitor;
```

```
procedure producer;
begin
    while true do
    begin

        item = produce_item;
        ProducerConsumer.insert(item);
    end
end;

procedure consumer;
begin
    while true do
    begin
        item=ProducerConsumer.remove;
        consume_item(item);

    end
end;
```

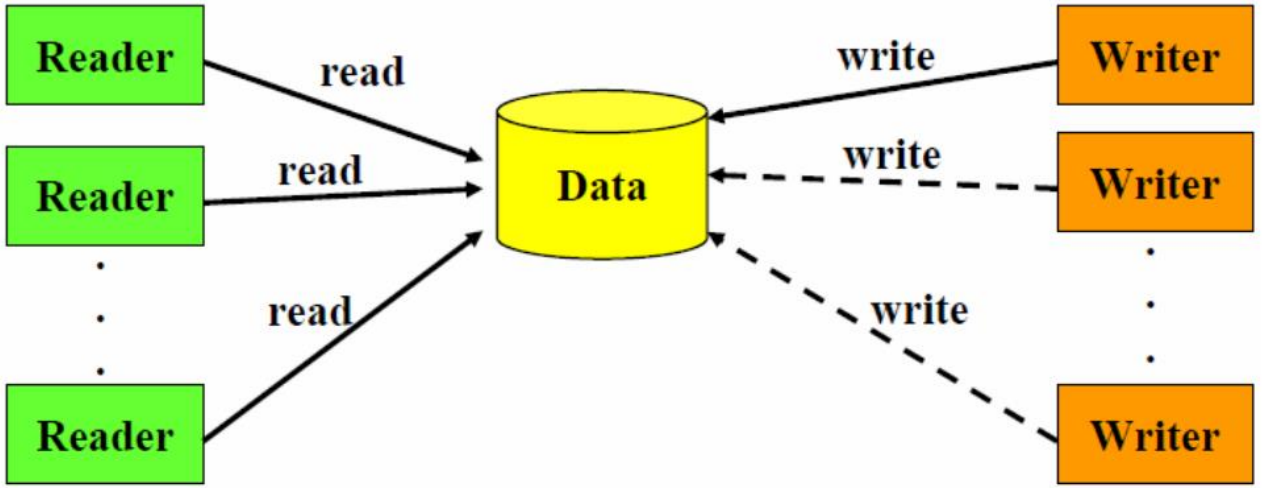
采用AND信号量集

- SP(empty, mutex),
- SP(full, mutex)
- 同学们可自行练习

读者－写者问题

- 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。即：

“读－写”互斥，“写－写”互斥，“读－读”允许。



读者-写者问题分析

- 生活中的实例：12306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
- 当读进程激活时，是否允许写进程进入?
- 当写进程激活时，？

读者-写者问题行为分析

- 读进程的行为：
  - 系统中会有多个读进程同时访问共享数据。
  - 可分为三类：第一个进入的读进程(占有资源),最后一个离开的读进程(释放资源)和其他读进程。
  - 需要设置一个计数器readcount来记录读进程的数目。
- 写进程的行为：排他性的使用资源。
- 确定同步与互斥关系：
  - 读者-读者:共享Data, 互斥访问readcount
  - 读者-写者:互斥访问Data
  - 写者-写者:互斥访问Data
- 确定临界资源：
  - **Data, readcount**

采用信号量机制

- 信号量设置：
  - int readcount=0;　　//“正在读”的进程数，初值是0。
  - semaphore rmutex=1;　//信号量，用于readcount的互斥。
  - semaphore fmutex=1;　//信号量，用于Data访问的互斥。
- 基本框架

Writer	Reader

P(fmutex);
写数据
V(fmutex);
	if first_reader? then P(fmutex);
	读数据
	if last_reader? then V(fmutex);

Writer

```
P(fmutex);
    writing;
V(fmutex);
```

Reader

```
P(rmutex);    //对readcount互斥
if readcount==0 then
    P(fmutex); //申请使用data资源
readcount := readcount + 1;
V(rmutex);    //释放readcount
   reading;
P(rmutex);    //对readcount互斥
readcount := readcount - 1;
if readcount==0 then
    V(fmutex); //释放data资源
V(rmutex);    //释放readcount
```

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);
```

深究“读者-写者”问题

Reader

```
P(rmutex);
if readcount==0 then  P(fmutex);
readcount := readcount +1;
V(rmutex);
read
P(rmutex)
readcount := readcount -1;
if readcount==0 then V(fmutex);
V(rmutex)
```

Writer

```
P(fmutex);
  write
V(fmutex);
```

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。

对读者有利的算法总结

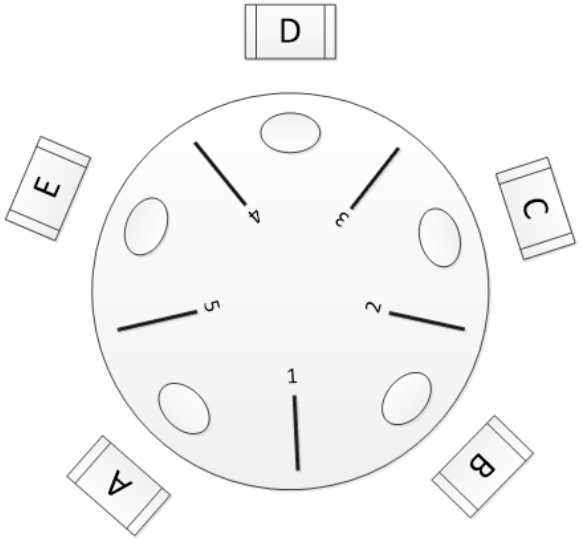
从读者、写者自身角度总结几条规则

- 写者：
  - 开始后，没有其他写者、读者可以进入
  - 必须确保没有正在读的读者，才能开始写
  - 写完之后允许其他读写者进入
- 读者开始读之前需要确保：
  - 第一个读者开始读，到最后一个读者结束读之间不允许有写进程进入写过程

深究“读者-写者”问题

- 给定读写序列：r1,w1,w2,r2,r3,w3…
  - 读者优先：r1,r2,r3,w1,w2,w3…
  - 写着优先：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?

哲学家进餐问题



问题描述：

（由Dijkstra首先提出并解决）5个哲学家围绕一张圆桌而坐，桌子上放着5支筷子，每两个哲学家之间放一支；哲学家的动作包括思考和进餐，进餐时需要同时拿起他左边和右边的两支筷子，思考时则同时将两支筷子放回原处。如何保证哲学家们的动作有序进行？如：不出现相邻者同时要求进餐；不出现有人永远拿不到筷子；

哲学家进餐问题

```
Var chopstick : array[0..4] of semaphore;

P(chopstick[i]);
P(chopstick[(i+1)mod 5]);
    eating
V(chopstick[i]);
V(chopstick [(i+1)mod 5]);
   thinking
```

如果**P(chopstick[i])**表示拿起左边的筷子，且5个哲学家同时拿？？

哲学家就餐问题的解题思路

- 至多只允许四个哲学家同时（尝试）进餐,以保证至少有一个哲学家能够进餐,最终总会释放出他所使用过的两支筷子,从而可使更多的哲学家进餐。设置信号量room=4。（破除资源互斥）
- 对筷子进行编号，奇数号先拿左，再拿右；偶数号相反。（破除循环等待）
- 同时拿起两根筷子，否则不拿起。（破除保持等待）

小结

- 同步与互斥
- 基于忙等待的方法
- 基于信号量的方法
- 管程方法
- IPC
- 经典同步与互斥问题
  - 生产者-消费者
  - 读者-写者
  - 哲学家就餐