

第5章 并发性：互斥和同步

5.1 互斥：软件解决方法

5.2 并发的原理

5.3 互斥：硬件的支持

5.4* 信号量

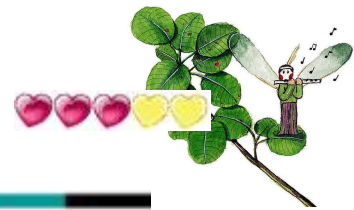
5.5* 管程

5.6 消息传递

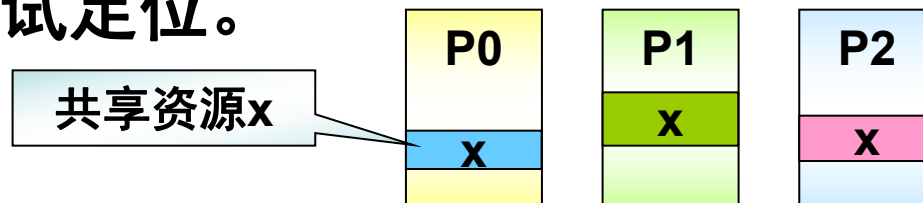
5.7 读者—写者问题



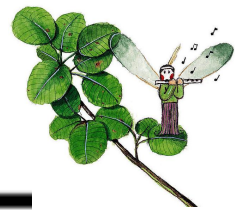
为什么要考虑“互斥与同步”？



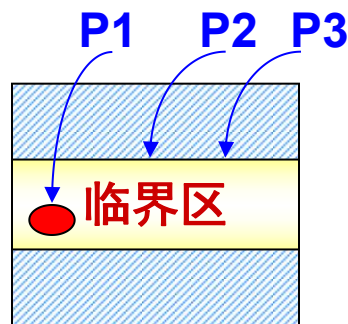
- ◆ 在单CPU、多CPU及分布式系统中，有多个进程并发甚至并行执行。对资源的共享和竞争，使得并发进程之间相互制约。
 - 由于共享某些资源（如变量、文件、设备等），一个进程的执行可能影响其它进程的执行结果。
 - 与同一共享资源有关的程序段分散在各进程中，且各进程的相对执行速度不可预知。
 - 由于每次并发执行顺序不同，并发进程的执行结果将不确定(不可再现)，甚至可能导致错误；但发生错误的精确条件和执行顺序很难再现，导致错误(bug)很难调试定位。



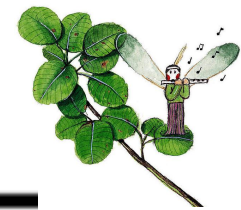
与并发相关的一些术语



- ◆ **原子操作**: 一个函数(原语)或动作的指令序列不可分割, 要么作为一个整体执行 (不可中断), 要么都不执行。
- ◆ **临界资源**: 一次仅允许一个进程独占使用的不可剥夺资源。
- ◆ **临界区**: 进程访问临界资源的那段程序代码。
一次仅允许一个进程在临界区中执行。
- ◆ **互斥**: 当一个进程正在临界区中访问临界资源时, 其它进程不能进入临界区。
- ◆ **同步**: 合作的并发进程需要按先后次序执行。
例如: 一个进程的执行依赖于合作进程的消息或信号。
当一个进程没有得到来自于合作进程的消息或信号时需阻塞等待, 直到消息或信号到达才被唤醒。
- ◆ **死锁**: 多个进程全部阻塞, 形成等待资源的循环链。
- ◆ **饥饿**: 一个就绪进程被调度程序长期忽视、不被调度执行; 一个进程长期得不到资源。



5.2 并发的原理



5.2.1 一个简单的例子



```
void echo()
```

```
{
```

```
① chin=getchar();
```

```
② chout=chin;
```

```
③ putchar(chout);
```

```
}
```

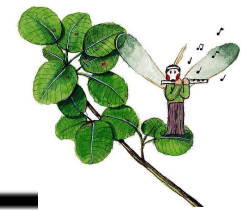
echo()是一个共享的键盘字符回显过程，所有进程都可以调用echo()。
chin和chout是全局共享变量。

◆ 在单CPU和多CPU系统中，多个进程对全局变量的访问不加以控制时，可能导致错误。

- 如：P1执行①后被中断，此时chin值为x；
之后P2执行①②③后结束，此时chin值为y；
最后P1被恢复调度，执行②③，输出y。

◆ 解决：一次只允许一个进程在echo中执行。当P2企图进入echo时将被阻塞，待P1退出echo后，P2才能进入。

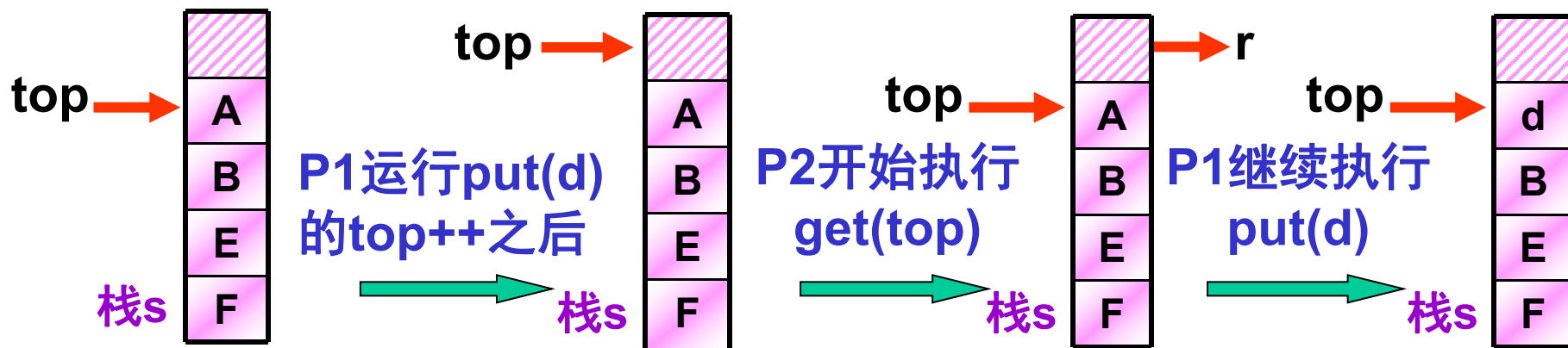
5.2 并发的原理



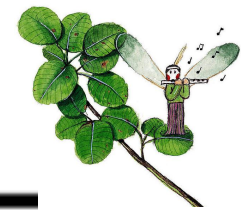
- 对程序的并发执行不加以控制时，运行结果具有不可再现性和错误的另一个例子：❤️❤️❤️❤️❤️
- 例：全局共享的堆栈s[]及栈顶指针top，将数据压栈的put(d)和取栈顶数据的get(top)可被P1、P2并发执行。

```
void put(int d)
{
    top++;
    s[top]=d;
}
```

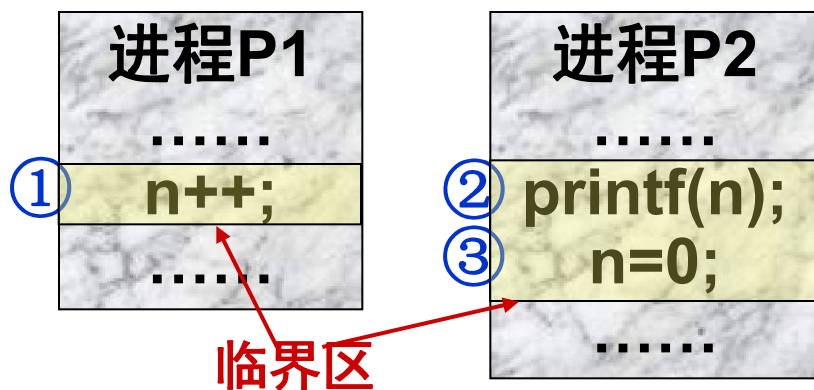
```
int get(int top)
{
    int r;
    r=s[top];
    top--;
    return r;
}
```



5.2.2 竞争条件



- ◆ **竞争条件**：多个线程或进程在读写共享数据时，最终结果依赖于它们的指令执行顺序。
- ◆ 例：n是共享变量。进程P1的n++、进程P2的printf(n)、n=0 交叉运行。设n初值=5。



不同执行顺序：

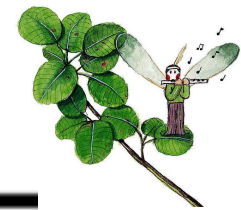
①②③：输出6, n=0;

②③①：输出5, n=1;

②①③：输出5, n=0;

- ◆ 上面的3条语句执行顺序：①②③和②③①都认为是正确的，但②①③被认为是错误的。

5.2.3 操作系统关注的问题

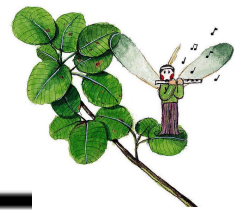


◆ 并发进程间应实现**互斥**与**同步**。

◆ OS应该：

- 跟踪不同进程，管理进程状态；
- 为每个活跃进程分配和释放各种资源；
- 保护每个进程的数据和物理资源，避免其它进程的有意无意干涉；
- 一个进程的功能和输出结果正确与否，应与其它并发进程的相对执行速度无关。

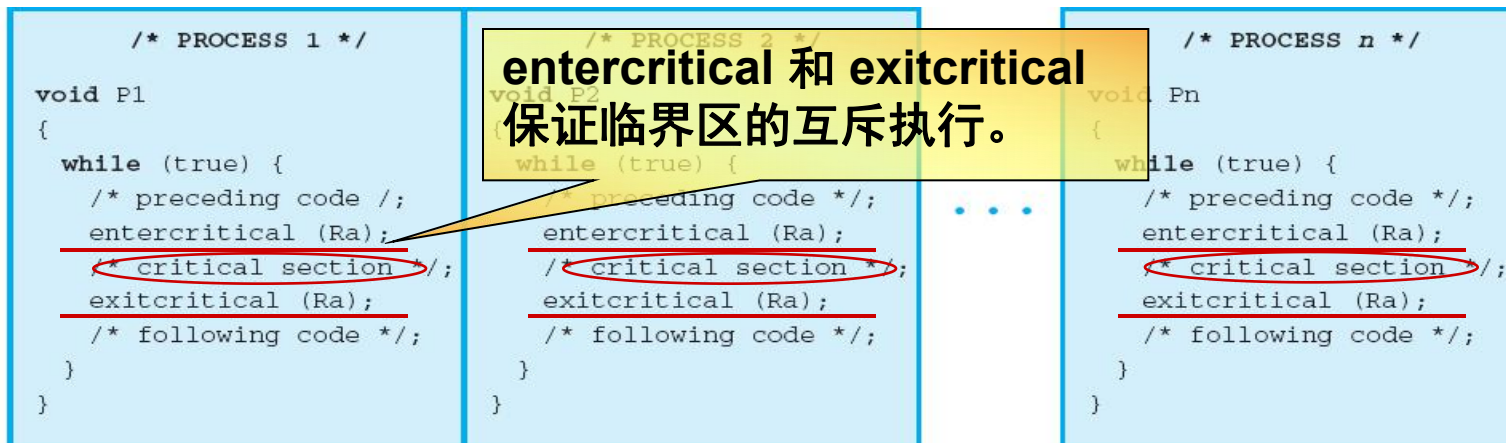
5.2.4 进程的交互



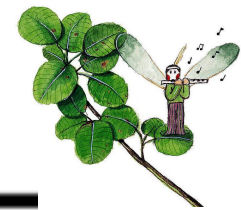
并发进程间的三种交互关系：（一）

◆ **进程间的资源竞争**：进程间相互不知道对方的存在

- 竞争使用CPU、存储器、独占式设备(如打印机)等。
- 竞争进程间没有任何信息交换，但执行过程可能受到其它竞争者的影响。如试图使用已分配设备时将阻塞。
- 竞争进程**互斥访问临界资源**，一次只允许一个进程在临界区中执行，其它试图进入临界区的进程将阻塞。但可能导致饥饿和死锁。



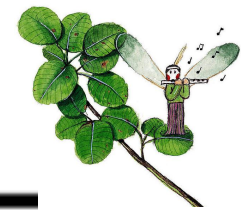
5.2.4 进程的交互



并发进程间的三种交互关系：（二）

- ◆ **进程间通过共享的合作**：进程间接知道对方的存在
 - 进程之间共享某些资源（如共享变量、文件、数据库等），但不能确切知道对方。
 - 多个进程**可以同时读**一个数据项，但须**互斥“写”**。
 - 多个进程读写共享数据时，需保证**数据一致性**。
 - 顺序执行每个进程会使共享数据保持一致性，但不同进程对共享数据交替读写会导致错误。如“竞争条件”节的例子。

5.2.4 进程的交互

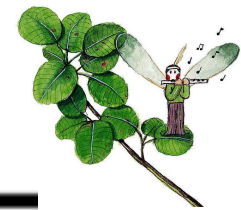


并发进程间的三种交互关系：（三）

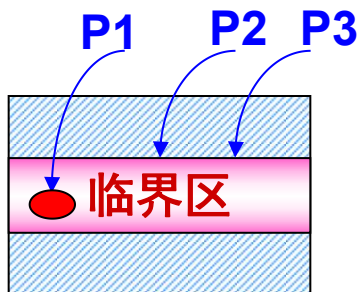
- ◆ **进程间通过通信的合作**：进程直接知道对方的存在
 - 多个进程**通过进程ID互相通信**（发送消息和接收消息），实现同步和协调各种活动。
 - 传递消息时，进程间未共享资源，则不需要互斥。但可能出现饥饿和死锁。



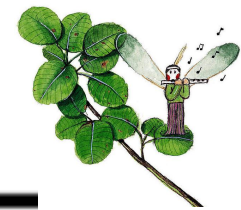
5.2.5 互斥的要求



- ◆ 对相关进程执行速度和处理器数目没有限制。
- ◆ **强制互斥（忙则等待）**：一次只允许一个进程进入临界区。有进程正在临界区执行时，其它请求进程等待；待该进程退出后，从多个请求进程中选择一个进入临界区。
- ◆ **有限等待**：请求进程应在有限的等待时间内进入临界区，不能造成进程死锁或饥饿。
 - 一个进程驻留在临界区中的时间是有限的。
- ◆ **有空让进**：当临界区空闲时，请求进程可立即进入。
- ◆ **让权等待**：进程不能进入临界区时，应释放处理器，避免忙等。



Question:



1、下列属于临界资源的是_____。

A. 磁盘存储介质

☒ B. 全局的公共队列

C. 局部变量

D. 可重入的代码

可同时被多个进程执行的纯代码

2、一个正在访问临界资源的进程由于申请等待I/O操作而被中断时，它_____。

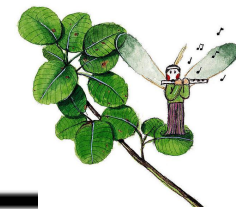
A. 可以允许其它进程进入该进程的相关临界区

B. 不允许其它进程进入任何临界区

C. 不允许其它进程抢占处理器

☒ D. 可以允许其它进程抢占处理器，但不能进入相关临界区

5.1 互斥：软件解决方法



5.1.1 Dekker算法 1965年

共享变量初值：turn = 0;
turn=i \Rightarrow P_i正在临界区中运行

共享变量初值：flag [0] = flag [1] = false;
flag[i]=true \Rightarrow P_i 可以进入临界区或正在临界区中

```
Process Pi
while(true)
{ while ( turn != i ) ;
  /* 临界区 */
  turn = j;
}
```

两进程P₀和P₁互斥
交替运行，但不满足
有空让进条件。

```
Process Pi
while(true)
{ while (flag[ j ]) ;
  flag[ i ] = true;
  /* 临界区 */
  flag [ i ] = false;
}
```

BUG: P₀和P₁可
能同时进入临界区

```
Process Pi
while(true)
{ flag[ i ] = true;
  while (flag[ j ]) ;
  /* 临界区 */
  flag [ i ] = false;
}
```

BUG: P₀和P₁可
能互相等待(死锁)

flag[i] =true && turn= i
 \Rightarrow P_i 正在临界区中运行

```
Process Pi
while(true)
{ flag[ i ] = true;
  while (flag[ j ])
  { if (turn==j)
    { flag[ i ]=false;
      while (turn==j) ;
      flag[ i ]=true; }
  }
  /* 临界区 */
  turn=j;
  flag [ i ] = false;
}
```

正确，只适于两
个进程的互斥。

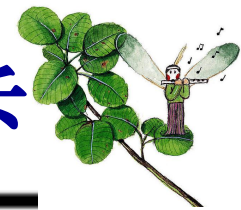
5.1.2 Peterson算法 1981年

共享变量初值：
flag [0] = flag [1] = false, turn = 0;
flag[i] =true && turn= i \Rightarrow P_i 正在临界区中运行

满足“强制互斥，有限等待，有空让进”
三个要求，但只适于两个进程的互斥。

```
Process Pi
while (true)
{ flag [ i ] = true;  turn = j;
  while (flag [ j ] && turn == j) ;
  /* 临界区 */
  flag [ i ] = false;
} // 只适于P0, P1的互斥
```


5.3 互斥：硬件的支持—用硬件实现互斥



5.3.1 中断禁用

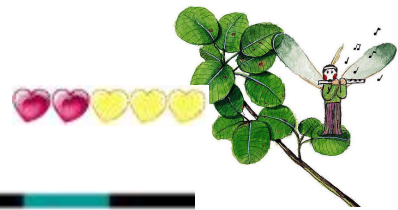


◆ 单CPU系统中，“**关中断**”可以实现临界区互斥。

```
while (true)
{
    /* 关中断 */;
    /* 临界区 */;
    /* 开中断 */;
    /* 其余部分 */;
}
```

- ◆ 不应该让用户关中断；
- ◆ 长时间关中断导致串行执行进程、系统的执行效率低；
- ◆ 不适合多处理器系统。每个CPU有各自的中断开关，禁止中断仅对执行关中断指令的那个CPU有效，其它CPU上的进程仍可继续运行并访问共享资源。

5.3.2 专用机器指令—实现互斥



◆ 两种机器指令(原子地在一个指令周期内执行):

● 比较和交换指令: **compare_and_swap**

```
int compare_and_swap (int *word, int testval, int newval)
{
    int oldval;
    oldval = *word;
    if (oldval == testval) *word = newval;
    return oldval;
}
```

比较word和testval, 相等时word赋值为newval。返回原word值。

一个临界资源一个bolt
变量, 初值为0(开锁)。
bolt为0时资源可用;
bolt为1时进程等待。
进入临界区时bolt=1;
退出临界区时bolt=0。

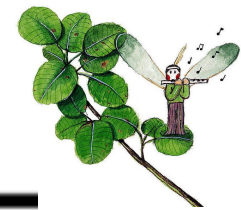


用**compare_and_swap**实现互斥

```
void P(int i) /* 进程Pi 代码。共n个进程。*/
{
    while (true) {
        while (compare_and_swap(bolt, 0, 1) == 1)
            /* do nothing, 返回值为1时循环比较 */;
        /* 临界区 */; /* 退出while循环时bolt=1加锁 */
        bolt = 0; /* 退出临界区后, bolt=0开锁 */
        /* 其余部分 */;
    }
}
```

忙等导致CPU效率低

比较和交换指令C&S机器指令—实现互斥



```
int compare_and_swap (int *word, int testval, int newval)
```

```
{  int oldval;
    oldval = *word;
    if (oldval == testval) *word = newval;
    return oldval;
```

} 比较word和testval，相等时word赋值为newval。返回原word值。

```
void P0
```

```
{  while (C&S(bolt, 0, 1) == 1)
    /* do nothing */;
    /* 临界区 */;
    bolt = 0;
    /* 其余部分 */;
}
```

```
void P1
```

```
{  while (C&S(bolt, 0, 1) == 1)
    /* do nothing */;
    /* 临界区 */;
    bolt = 0;
    /* 其余部分 */;
}
```

bolt初值=0。

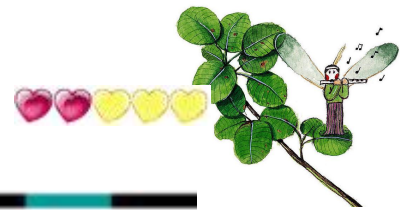
设P0先运行，**C&S(0,0,1)**将**bolt赋值=1**；返回bolt旧值0；while条件 0==1 不成立，结束while循环，进入临界区。

P1开始运行，**C&S(1,0,1)**返回bolt值1；while条件 1==1 成立，重复执行while循环（忙等待导致CPU效率低）；

P0继续运行，退出临界区后将**bolt赋值=0**。

P1再次运行，**C&S(0,0,1)**将**bolt赋值=1**；返回bolt旧值0；while条件 0==1 不成立，结束while循环，进入临界区。退出临界区后将**bolt赋值=0**。

5.3.2 专用机器指令—实现互斥



◆ 两种机器指令(原子地在一个指令周期内执行):

● 交换指令: **exchange**

```
void exchange (int *register, int *memory)
{
    int temp;
    temp = *memory;
    *memory = *register;
    *register = temp;
} 交换register和memory的值。
```



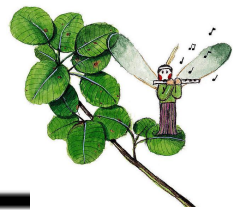
用**exchange**实现互斥

一个临界资源一个bolt
变量, 初值为0(开锁)。
bolt为0时资源可用;
bolt为1时进程等待。
进入临界区时bolt=1;
退出临界区时bolt=0。

```
void P(int i) /* 进程Pi 代码。共n个进程。*/
{
    int keyi = 1; /* 每个进程有自己的keyi */
    while (true) {
        while (keyi == 1) exchange (&keyi, &bolt);
        /* keyi为1时, 循环交换 */
        /* 临界区 */;
        bolt = 0; /* 退出临界区时开锁 */
        /* 其余部分 */;
    }
}
```

忙等(自旋等待)

交换指令EX机器指令—实现互斥



```
void exchange (int *register, int *memory)
{   int temp;
    temp = *memory;
    *memory = *register;
    *register = temp;
} 交换register和memory的值。
```

bolt初值=0。

```
void P0
{ int key0 = 1;
  while (key0 == 1)
    EX(&key0, &bolt);
  /* 临界区 */;
  bolt = 0;
}
```

设P0先运行，key0赋值=1；key0==1条件成立，交换key0和bolt值，使key0=0，**bolt=1**；再次判断循环条件key0==1不成立，结束循环进入临界区。

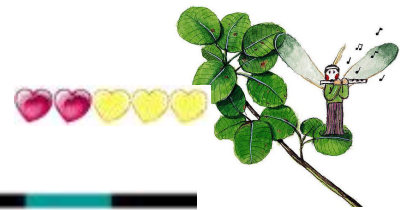
P1开始运行，key1赋值=1；key1==1条件成立，交换key1和bolt值，使key1=1，**bolt=1**；再次判断循环条件key1==1仍然成立，重复执行while循环（忙等导致CPU效率低）；

P0继续运行，退出临界区后将**bolt赋值=0**。

```
void P1
{ int key1 = 1;
  while (key1 == 1)
    EX(&key1, &bolt);
  /* 临界区 */;
  bolt = 0;
}
```

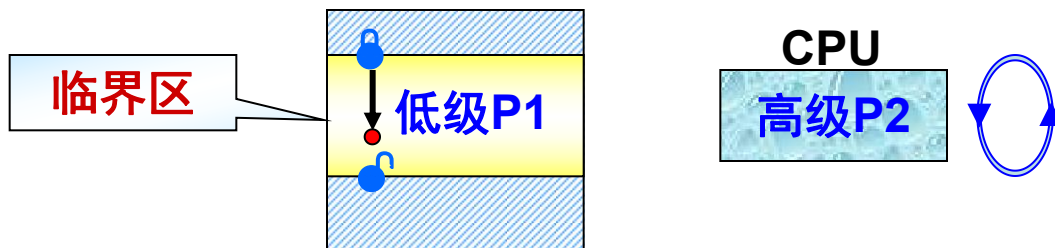
P1再次运行，key1==1条件成立，交换key1和bolt值，使key1=0，**bolt=1**；再次判断循环条件key1==1不成立，结束循环进入临界区。退出临界区后将**bolt赋值=0**。

5.3.2 专用机器指令—实现互斥

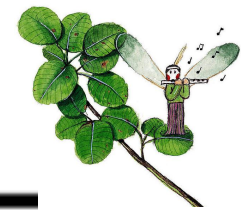


◆ 这两种机器指令可以实现互斥。

- 进程数量任意，适合于共享内存（要访问全局变量bolt）的单CPU和多CPU系统。
- 可支持多个临界区互斥。各临界资源有自己的“bolt”。
- 忙等（也称自旋等待）消耗CPU时间，效率低。
- 可能饥饿：多个进程等待进入临界区时，选择哪个等待进程是随机的。
- 可能死锁：如按优先级调度CPU时，设一低优先级的P1进入临界区后被中断，高优先级的P2抢占了CPU；P2试图使用同一临界资源时被拒绝且开始忙等循环；P1优先级低，无法被调度执行及退出临界区。此时P1和P2进入死锁状态。



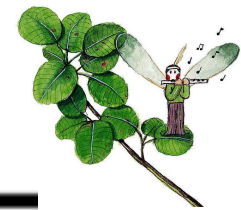
5.4 信号量 semaphore



- ◆ 信号量：不要求忙等的同步互斥工具。
 - 只考虑计数信号量(V8图5.3, V9图5.6)。二元信号量及互斥量略。
- ◆ 一个信号量表示一种资源。
- ◆ 当资源不可用时，进程将阻塞（避免忙等），直到另一进程释放资源时才被唤醒。
- ◆ 信号量 s 只能被下面的两个原语访问：
 - $\text{semWait}(s)$ 也称 $P(s)$ 操作, $\text{wait}(s)$
 - $\text{semSignal}(s)$ 也称 $V(s)$ 操作, $\text{signal}(s)$

1965年，荷兰，Dijkstra提出

5.4 信号量—信号量含义




- ◆ 为每种资源设置一个信号量 s ，可用于 n 个进程的同步互斥；信号量的值表示可用资源个数。只能由 $\text{semWait}(s)$ 、 $\text{semSignal}(s)$ 访问和修改；
- ◆ 用于互斥时， s 初值为1，取值为 $1 \sim -(n-1)$ 。
 - $s=1$ 时：有1个临界资源可用，一个进程可进入临界区。
 - $s=0$ 时：临界资源已分配，一个进程已进入临界区。
 - $s<0$ 时：临界区已被占用， $|s|$ 个阻塞进程正等待进入。
- ◆ 用于同步时， s 初值将 ≥ 0 。
 - $s \geq 0$ ：可用资源个数(或可进入临界区的进程个数)。
 - $s < 0$ ：该资源的等待队列长度(阻塞进程个数)。

资源1 

$s_1=1$

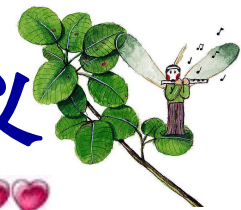
资源2  → P5

$s_2=0$

资源3  → P0

$s_3=-3$ → P2 → P1 → P4

5.4 信号量 — semWait、semSignal含义



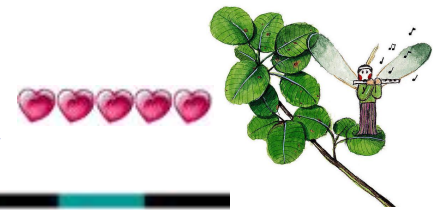
- ◆ **semWait(s):** 本进程请求分配一个资源。
- ◆ **semSignal(s):** 本进程释放一个资源。
- ◆ **semWait(s)、semSignal(s)操作必须成对出现。**
 - 用于互斥时，位于同一进程内、临界区前/后；
 - 用于同步时，交错出现于两个合作进程内。
- ◆ 多个semWait()的次序不能颠倒，否则可能导致死锁。用于同步的semWait(s1)应出现在用于互斥的semWait(s2)之前。
- ◆ 多个semSignal()操作的次序可任意。


s=1

进程P: semWait(s); 临界区; semSignal(s);	进程Q: semWait(s); 临界区; semSignal(s);
--	--

进程P: 代码A; semSignal(s);	进程Q: semWait(s); 代码B;
--------------------------------------	------------------------------------

5.4 信号量—信号量原语的定义



```
struct semaphore
{ int count;          /* 信号量的值，实质是可用资源个数 */
  queueType queue;    /* 因该信号量而阻塞的进程队列 */
};                    /* 信号量定义（V8图5.3，V9图5.6） */
```

```
semWait(semaphore s)
{
  s.count--;
  if (s.count < 0)
  { 将当前进程放入s.queue;
    阻塞当前进程;
  }
}
```

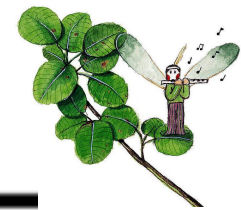
申请分配资源s

```
semSignal(semaphore s)
{
  s.count++;
  if (s.count <= 0)
  { 从s.queue中移除进程P;
    将进程P插入到就绪队列;
  }
}
```

释放资源s

V8图5.3, V9图5.6

5.4.3 信号量的实现

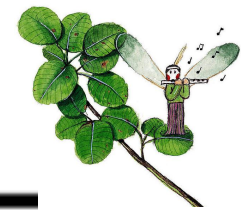


- ❖ 怎样保证 semWait和semSignal 被**原子地**执行?
 - Compare&Swap指令：适用于多CPU系统 (略)
 - 关中断方法：适用于单CPU系统

```
semWait(s)
{ 关中断;
  s.count - -;
  if (s.count < 0) {
    本进程放到s.queue阻塞;
  }
  开中断;
}
```

```
semSignal(s)
{ 关中断;
  s.count++;
  if (s.count <= 0) {
    唤醒s.queue的队首进程;
  }
  开中断;
}
```

5.4 信号量一强、弱信号量



- ◆ 当进程Q执行semSignal(s)释放资源时，将唤醒一个阻塞进程并移出阻塞队列。
- ◆ 进程按照什么顺序从阻塞队列中移出？
 - **FIFO**：被阻塞时间最久的进程最先被移出。
- ◆ **强信号量**：采用FIFO移出策略的信号量。
 - 不会导致饥饿。典型的信号量形式。
- ◆ **弱信号量**：未规定移出顺序的信号量(随机选择)。
(略)

5.4.1 互斥—用信号量实现互斥



- ❖ 对每一临界资源设一个信号量s，初值=1。
- ❖ **实现互斥**：每个进程在进入临界区前，执行semWait(s)；退出临界区后，执行semSignal(s)。
- ❖ 临界区内不应有可能引起阻塞或死锁的因素。

进程P1:

.....

semWait(s);
临界区1;
semSignal(s);

.....

进程P2:

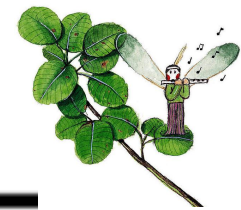
.....

semWait(s);
临界区2;
semSignal(s);

.....

```
const int n = /* 进程数 */;  
semaphore s = 1; /*s初值=1*/  
void P(int i)      /*进程Pi */  
{ while (true) {  
    semWait(s);  
    /* 临界区 */;  
    semSignal(s);  
    /* 其它部分 */;  
}  
}  
void main()      /*n个进程并发*/  
{ parbegin (P(1), P(2), . . . , P(n)); }
```

两个进程实现互斥例子




semWait(semaphore s)

```
{ s.count--;  
  if (s.count < 0)  
  { 将当前进程放入s.queue;  
    阻塞当前进程;  
  }  
}
```

semSignal(semaphore s)

```
{ s.count++;  
  if (s.count <= 0)  
  { 从s.queue中移除进程P;  
    将进程P插入到就绪队列;  
  }  
}
```

资源 
s=1

进程P1:

```
.....  
semWait(s);  
临界区1;  
semSignal(s);  
.....
```

s初值=1;

P1先运行, 执行 **Wait(s)**, 使得 **s→0**; 进入临界区1执行;
时间片超时后, 让出CPU;

P2被调度执行, 执行 **Wait(s)**, 使得 **s→-1**; P2被阻塞;

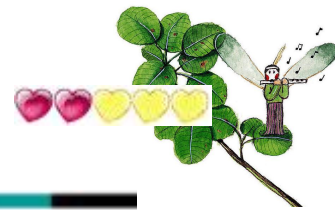
进程P2:

```
.....  
semWait(s);  
临界区2;  
semSignal(s);  
.....
```

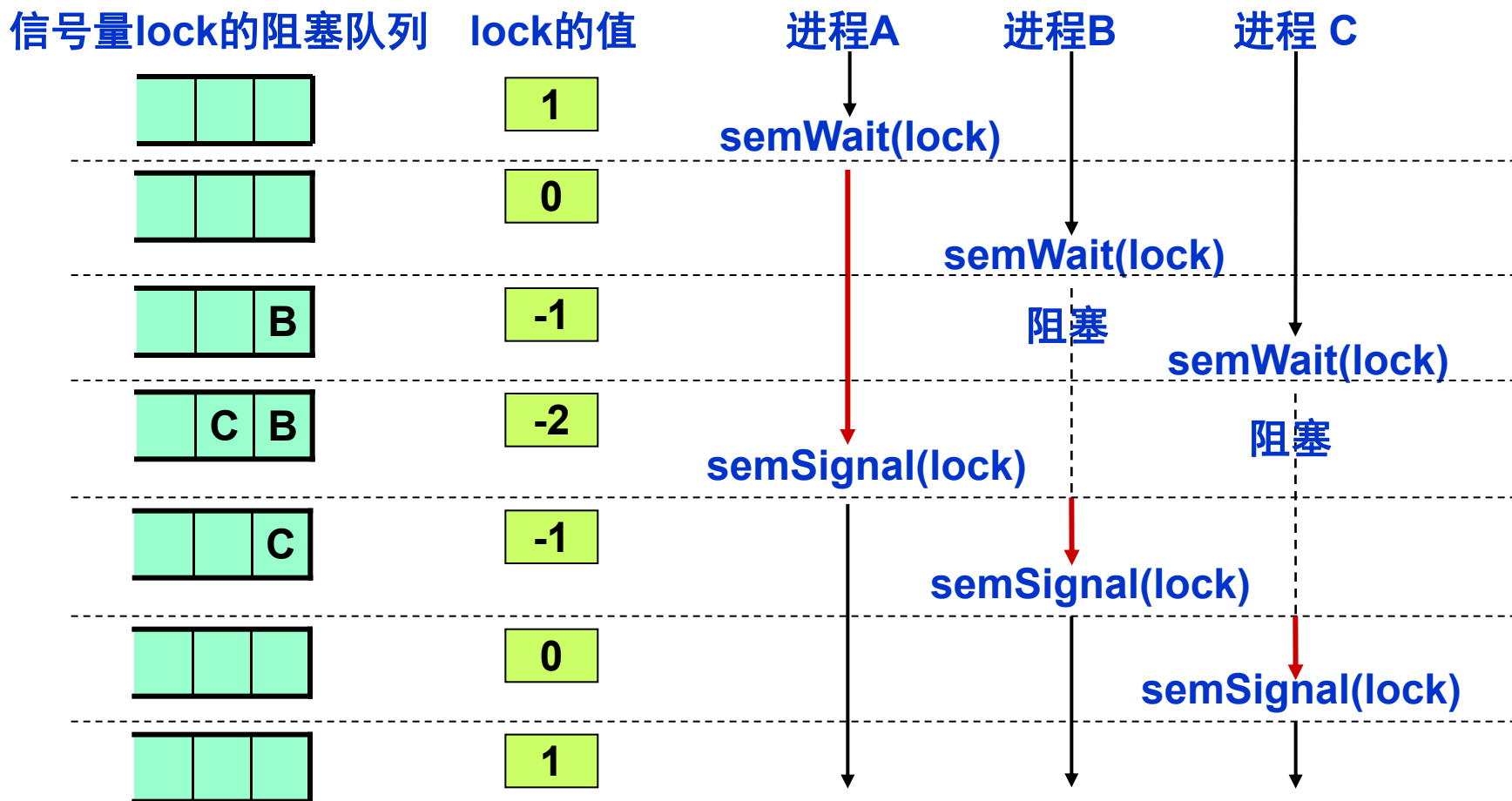
P1再次运行, 执行 **Signal(s)**, 使得 **s→0**; 唤醒阻塞进程P2;
运行完毕。

P2再次运行, 直接进入临界区2执行; 之后再执行 **Signal(s)**,
使得 **s→1**; 没有阻塞进程需要被唤醒; 运行完毕。

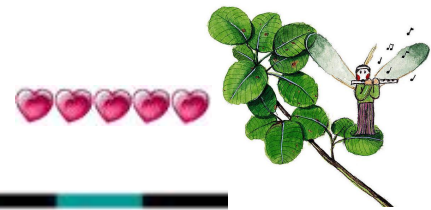
5.4.1 互斥—用信号量实现互斥例



◆ 例：三个进程访问一个受信号量lock保护的临界资源。
其它代码可并行执行，但临界区代码只能“串行”执行。



5.4.1 互斥—用信号量实现同步



- ◆ 同步时，对每“**一类资源**”设一信号量 s ，初值 ≥ 0 ，表示可用资源个数。
 - 资源：可以是同一物理资源的不同状态，如缓冲的空和满。对空缓冲和满缓冲分别设置信号量。
- ◆ 要求实现：P1已执行过A后，P2才能开始执行B。设一信号量 s ，初值为0。

■ Process P1:

do

{

代码段 A;

semSignal(s);

}

■ Process P2:

do

{ **semWait(s);**

代码段 B;

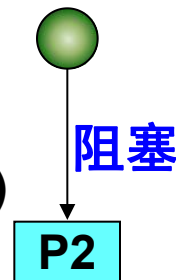
.....

}

例： **$s=0$**

P2先运行
semWait(s)

$s=-1$

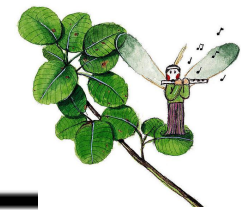


P1再运行
semSignal(s)

$s=0$

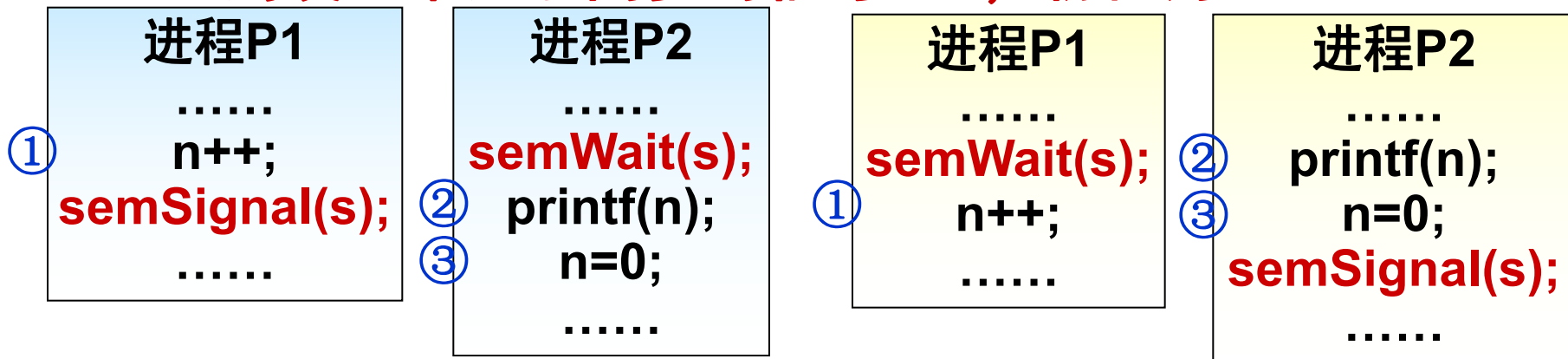
唤醒P2

Question:

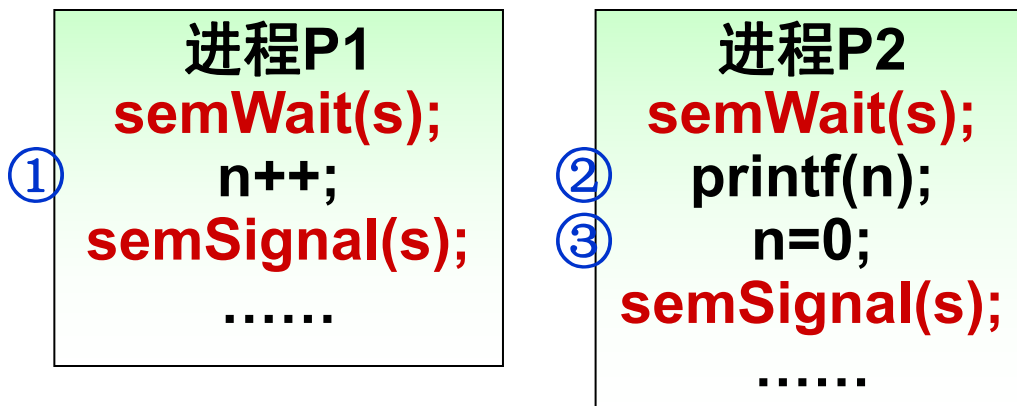


◆考虑5.1.2 竞争条件小节例，如何实现语句执行顺序为①②③或②③①？

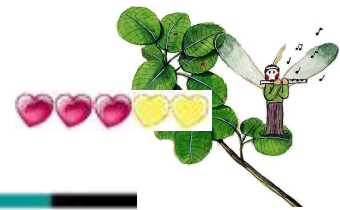
若必须以①②③顺序执行： 若必须以②③①顺序执行：
则设一个用于同步的信号量s，初值为0。



若不强制要求P1或P2的执行顺序，则设一互斥信号量s，初值为1。



5.4.1 互斥 — semWait顺序不能颠倒



- ◆ 多个 semWait 操作的顺序不能颠倒，否则可能导致死锁。semSignal 的顺序可以颠倒。
- ◆ 例：设 S 和 Q 是两个初值为1的信号量。

P1	P2
{	{
semWait(S);	semWait(S);
semWait(Q);	semWait(Q);
.....
semSignal(S);	semSignal(Q);
semSignal(Q);	semSignal(S);
}	}

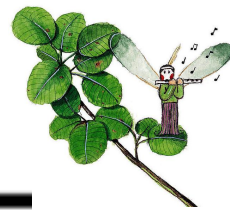
正确顺序

P1	P2
{	{
semWait(S);	semWait(Q);
semWait(Q);	semWait(S);
.....
semSignal(S);	semSignal(S);
semSignal(Q);	semSignal(Q);
}	}

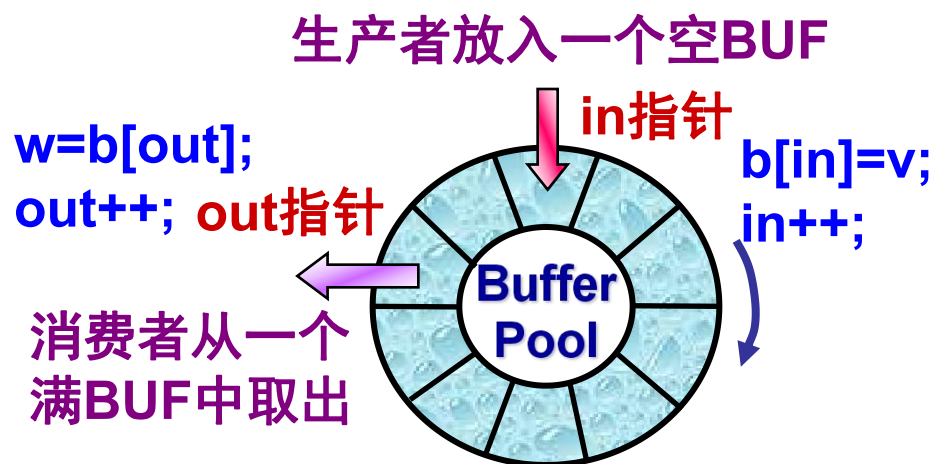
错误顺序

Question: 如果一个糟糕的程序员漏写或多写了 semWait 或 semSignal，将会怎样？

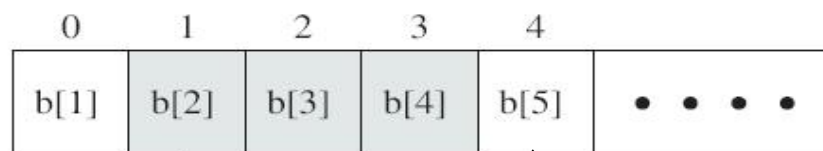
5.4.2 生产者/消费者问题



- ◆ 一组生产者进程产生数据，放到缓冲区。
- ◆ 一组消费者进程从缓冲区中取出数据使用。
 - 有限缓冲：假设缓冲池大小固定，包含k个缓冲区。生产者和消费者共用一个循环缓冲池。
 - 无限缓冲：缓冲区数量无限制。

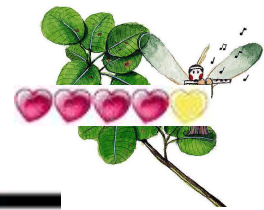


in和out指针可被多个进程同时访问修改，故缓冲池需互斥访问。



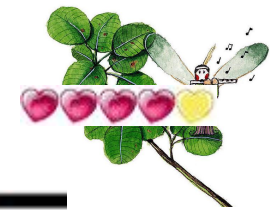
缓冲区数量无限。
白色：空缓冲。
灰色：有效的满缓冲。

5.4.2 生产者/消费者问题—有限缓冲



- ◆ 一组生产者进程和一组消费者进程共用一个有sizeofbuffer个缓冲区的缓冲池来交换数据。
- ◆ 资源、约束条件及信号量设置：
 - **缓冲池**一次只能让一个进程访问。(互斥)
设一信号量 **s**，初值为 **1**。
 - 生产者需要**空缓冲**来发送数据。(同步)
设一信号量 **empty**，初值为 **sizeofbuffer**。
 - 消费者需要**满缓冲**来获取数据。(同步)
设一信号量 **full**，初值为 **0**。

5.4.2 生产者/消费者问题—有限缓冲



```
void producer()
{ while (true) {
    生产一个数据;
    semWait(empty);
    semWait(s);
    把数据送到缓冲区;
    semSignal(s);
    semSignal(full);
  }
}
```

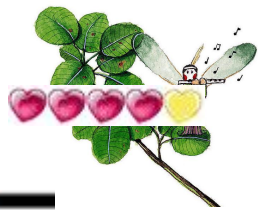
```
void consumer()
{ while (true) {
    semWait(full);
    semWait(s);
    从缓冲区中取出数据;
    semSignal(s);
    semSignal(empty);
    消费数据;
  }
}
```

V8图
5.13,
V9图
5.16

semSignal的顺序任意，但semWait的顺序不能颠倒。
用于同步的semWait()应出现在用于互斥的semWait()之前

Question: 这里，哪个是同步semWait，哪个是互斥semWait?

5.4.2 生产者/消费者问题—wait顺序正确例



◆ 设2个生产者、2个消费者共用一个8缓冲的缓冲池。

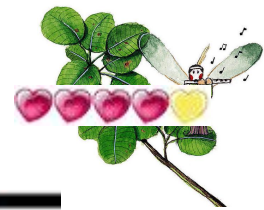
$P_{生1}$	$P_{生2}$	$P_{消1}$	$P_{消2}$
.....
生产数据1;	生产数据2;	Wait(full);	Wait(full);
Wait(empty);	Wait(empty);	Wait(s);	Wait(s);
Wait(s);	Wait(s);	从缓冲取数x;	从缓冲取数y;
放入缓冲;	放入缓冲;	Signal(s);	Signal(s);
Signal(s);	Signal(s);	Signal(empty);	Signal(empty);
Signal(full);	Signal(full);	消费数据x;	消费数据y;
.....

各信号量初值如下：

$s=1$
 $empty=8$
 $full=0$

无论生产者、消费者以何顺序交替执行，都可保证正确地同步互斥。

5.4.2 生产者/消费者问题—wait顺序颠倒例



◆ 设生产者wait颠倒:

$P_{生}$

$P_{消}$

.....

.....

生产数据;

Wait(full);

Wait(s);

Wait(s);

Wait(empty);

从缓冲取数x;

放入缓冲;

Signal(s);

Signal(s);

Signal(empty);

Signal(full);

消费数据x;

.....

.....

信号量当前值如下。生产者先运行，消费者后运行时将死锁。

$s=1$

$empty=0$

$full=8$

$P_{生}$ 执行 **Wait(s)**，使得 $s \rightarrow 0$;

$P_{生}$ 时间片超时，调度 $P_{消}$ 执行;

$P_{消}$ 执行 **Wait(full)**，使得 $full \rightarrow 7$;

$P_{消}$ 执行 **Wait(s)**，使得 $s \rightarrow -1$ ， **$P_{消}$ 阻塞**;

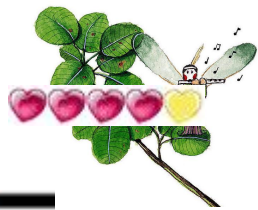
调度 $P_{生}$ 执行;

$P_{生}$ 执行 **Wait(empty)**，使得 $empty \rightarrow -1$ ，

$P_{生}$ 阻塞;

两个进程发生死锁。

5.4.2 生产者/消费者问题—wait顺序颠倒例



⑩ 设消费者wait颠倒:

$P_{生}$

$P_{消}$

.....

.....

生产数据2;

Wait(s);

Wait(empty);

Wait(full);

Wait(s);

从缓冲取数y;

放入缓冲;

Signal(s);

Signal(s);

Signal(empty);

Signal(full);

消费数据y;

.....

.....

信号量当前值如下。消费者先运行，生产者后运行时将死锁。

$s=1$

$empty=8$

$full=0$

$P_{消}$ 执行 **Wait(s)**，使得 $s \rightarrow 0$;

$P_{消}$ 时间片超时，调度 $P_{生}$ 执行;

$P_{生}$ 执行 **Wait(empty)**，使得 $empty \rightarrow 7$;

$P_{生}$ 执行 **Wait(s)**，使得 $s \rightarrow -1$ ， **$P_{生}$ 阻塞**;

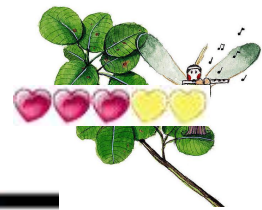
调度 $P_{消}$ 执行;

$P_{消}$ 执行 **Wait(full)**，使得 $full \rightarrow -1$ ，

$P_{消}$ 阻塞;

两个进程发生死锁。

5.4.2 生产者/消费者问题—无限缓冲



- 无限缓冲时，仍然需要对缓冲区数组互斥访问。
设一信号量 s ，初值为 1。
- 无限缓冲时，认为空缓冲总是存在的，因此只考虑“满缓冲数量”即可。设一信号量 $full$ ，初值为 0。

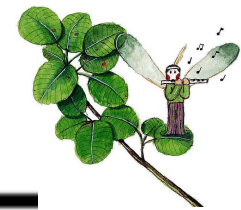
semaphore full=0, s=1;

```
void producer()
{ while (true) {
    生产数据;
    semWait(s);
    放入缓冲;
    semSignal(s);
    semSignal(full);
  }
}
```

```
void consumer()
{ while (true) {
    semWait(full);
    semWait(s);
    从缓冲取数;
    semSignal(s);
    消费数据;
  }
}
```

V8图
5.11,
V9图
5.14

Question:



- ◆ 设有三个进程：进程get读数并送到buf1，进程copy复制buf1到buf2，进程put打印buf2中的数据。用信号量实现get、copy、put的同步。



设4个信号量代表4种资源：S1—空buf1、S2—满buf1、S3—空buf2、S4—满buf2，初值 1、0、1、0。

进程get

```
while TRUE
{ semWait(S1);
  读数到buf1;
  semSignal(S2);
}
```

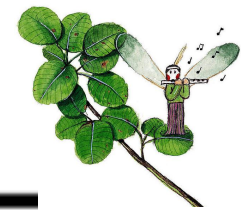
进程copy

```
while TRUE
{ semWait(S2);
  semWait(S3);
  复制buf1到buf2;
  semSignal(S1);
  semSignal(S4);
}
```

进程put

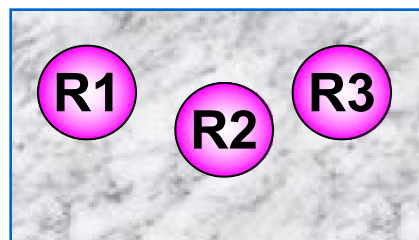
```
while TRUE
{ semWait(S4);
  打印buf2;
  semSignal(S3);
}
```

5.7 读者/写者问题



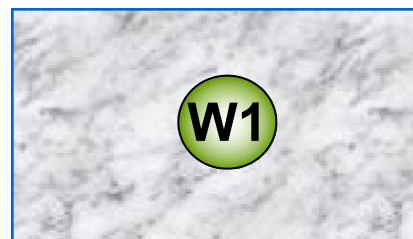
◆ 有一个共享的数据对象：

- 允许多个读者进程同时读；
- 一次只允许一个写者进程写；当一个写者正在写时，不允许其它任何读者或写者同时访问该共享对象。



W1

W2

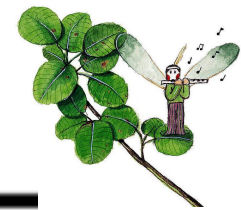


R1

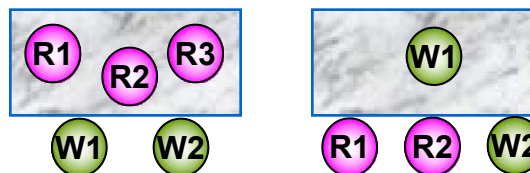
R2

W2

5.7.1 读者优先



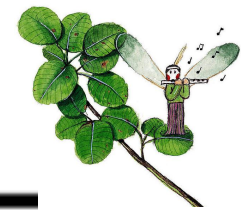
- ◆ 当至少已有一个读者正在读时，随后的读者直接进入，开始读数据对象。但写者将等待。
- ◆ 当一个写者正在写时，随后到来的读者和写者都等待。



- ◆ 信号量设置：

- 一次只能让一个写者或一群读者访问数据。
设一互斥信号量 **wsem**，初值为1。
- 正在读数据的读者数目由**全局变量readcount**表示(初值=0)，它被多个读者互斥访问。
(第1个读者需对数据加锁，最后1个读者对数据解锁)
为readcount设一互斥信号量 **x**，初值为1。

5.7.1 读者优先



```
■ void reader()
{ while true {
    semWait(x);
    readcount++;
    if (readcount == 1)
        semWait(wsem);

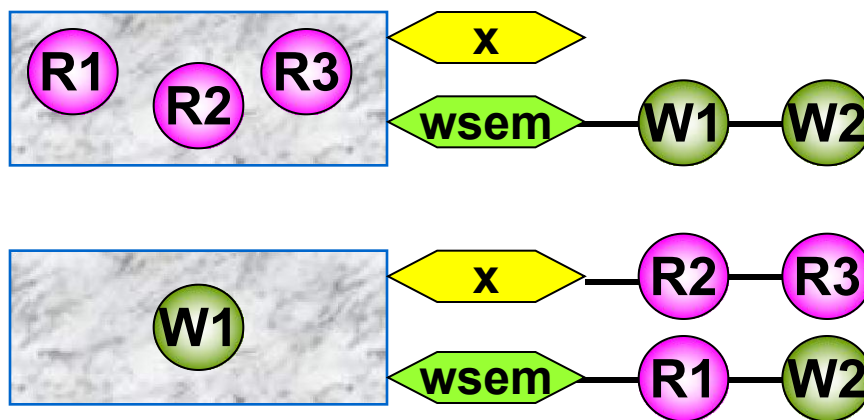
    semSignal(x);

    读数据对象;

    semWait(x);
    readcount--;
    if (readcount == 0)
        semSignal(wsem);

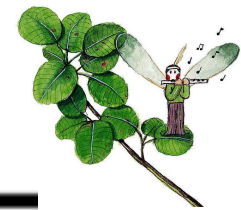
    semSignal(x);
}}
```

```
■ void writer()
{ while true {
    semWait(wsem);
    写数据对象;
    semSignal(wsem);
}}
```

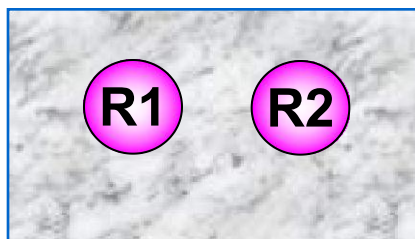


信号量的等待队列

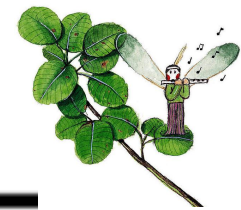
5.7.2 写者优先



- ◆ 当一个写者声明想写时，不允许新的读者进入数据对象，只需等待已有的读者读完即可开始写。可以避免写者饥饿。
- ◆ 具体实现（ V8图5.23， V9图5.26 略）。



Question:



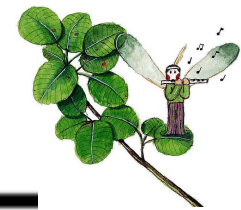
- ◆ 设有两个机器人拣黑、白棋子。
P1拣白子，P2拣黑子，交替拣，互斥拣。
用信号量实现P1、P2的同步。

设两个信号量：w表示可拣白子、b表示可拣黑子。
设先拣黑子，w初值0，b初值1。

```
P1:  
while TRUE  
{ semWait(w);  
  拣白子;  
  semSignal(b);  
}
```

```
P2:  
while TRUE  
{ semWait(b);  
  拣黑子;  
  semSignal(w);  
}
```

Question:



- ◆ 桌上有一空盘，允许放一只水果。爸爸可向盘中放桔子，也可放苹果。儿子专等吃盘中的桔子，女儿专等吃盘中的苹果。规定当盘空时一次只能放一只水果供吃者取用。试用信号量实现爸爸、儿子、女儿三个并发进程的同步。

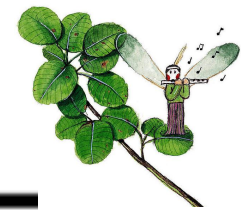
设3个信号量，S表示盘子是否为空(空盘子数)，初值为1。
So表示盘中桔子数，初值为0。Sa表示盘中苹果数，初值为0。

```
爸爸：
while TRUE
{ semWait(S);
  将水果放入盘中；
  if (放入的是桔子)
    semSignal(So);
  else
    semSignal(Sa);
}
```

```
儿子：
while TRUE
{ semWait(So);
  从盘中取桔子；
  semSignal(S);
  吃桔子；
}
```

```
女儿：
while TRUE
{ semWait(Sa);
  从盘中取苹果；
  semSignal(S);
  吃苹果；
}
```

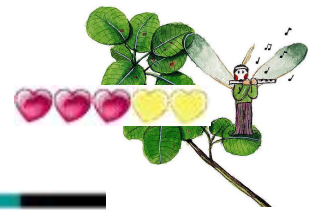
5.5 管程Monitor



◆ 引入管程：1974年

- 各个进程分别对每个可共享的临界资源进行semWait和semSignal操作，容易发生错误。
- 在多种程序设计语言中实现了**管程(Monitor)**。允许用管程锁定任何对象，如变量、数组、链表、甚至表中每个元素。
- 管程将共享对象以及能对它进行的所有操作(原本分散在各进程的临界区)集中(封装)在一个模块中。管程本身结构保证各操作的互斥执行。
- 各进程调用相关操作，可防止进程有意/无意的互斥/同步错误。

5.5.1 使用信号的管程—组成部分



◆ 管程的组成:

```
monitor 管程名  
{ 声明局部数据和条件变量
```

```
    procedure P1 (...)
```

```
    { ... }
```

```
    procedure P2 (...)
```

```
    { ... }
```

```
    .....
```

```
    procedure Pn (...)
```

```
    { ... }
```

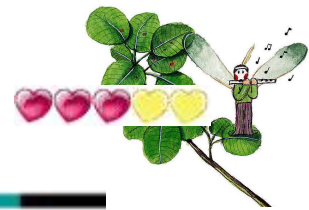
```
{ 初始化代码部分 }
```

只有管程内部的过程能访问它们

一组对共享的局部数据和条件变量进行操作的过程。一个进程通过调用某个过程，进入管程。管程本身保证互斥，故管程中只能有一个活跃进程。

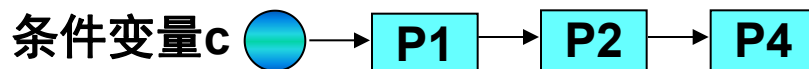
初始化局部数据

5.5.1 使用信号的管程—条件变量

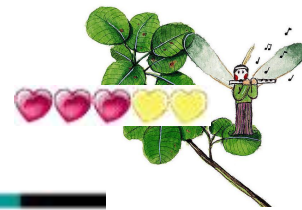


◆ 条件变量 condition variable:

- 表示进程正在等待的资源或原因。只用于维护等待队列，但没有相关联的值。
- ◆ 有两个原语可以操作条件变量：
 - **cwait(c)**: 条件不满足时，调用 **cwait(c)** 的进程被放到条件变量c的等待队列中阻塞。
 - **csignal(c)**: 唤醒一个条件变量c的阻塞进程。
- ◆ 如果没有可用资源，调用 **cwait(c)** 的进程将被阻塞；直到另一进程释放资源时，调用**csignal(c)** 将其唤醒。

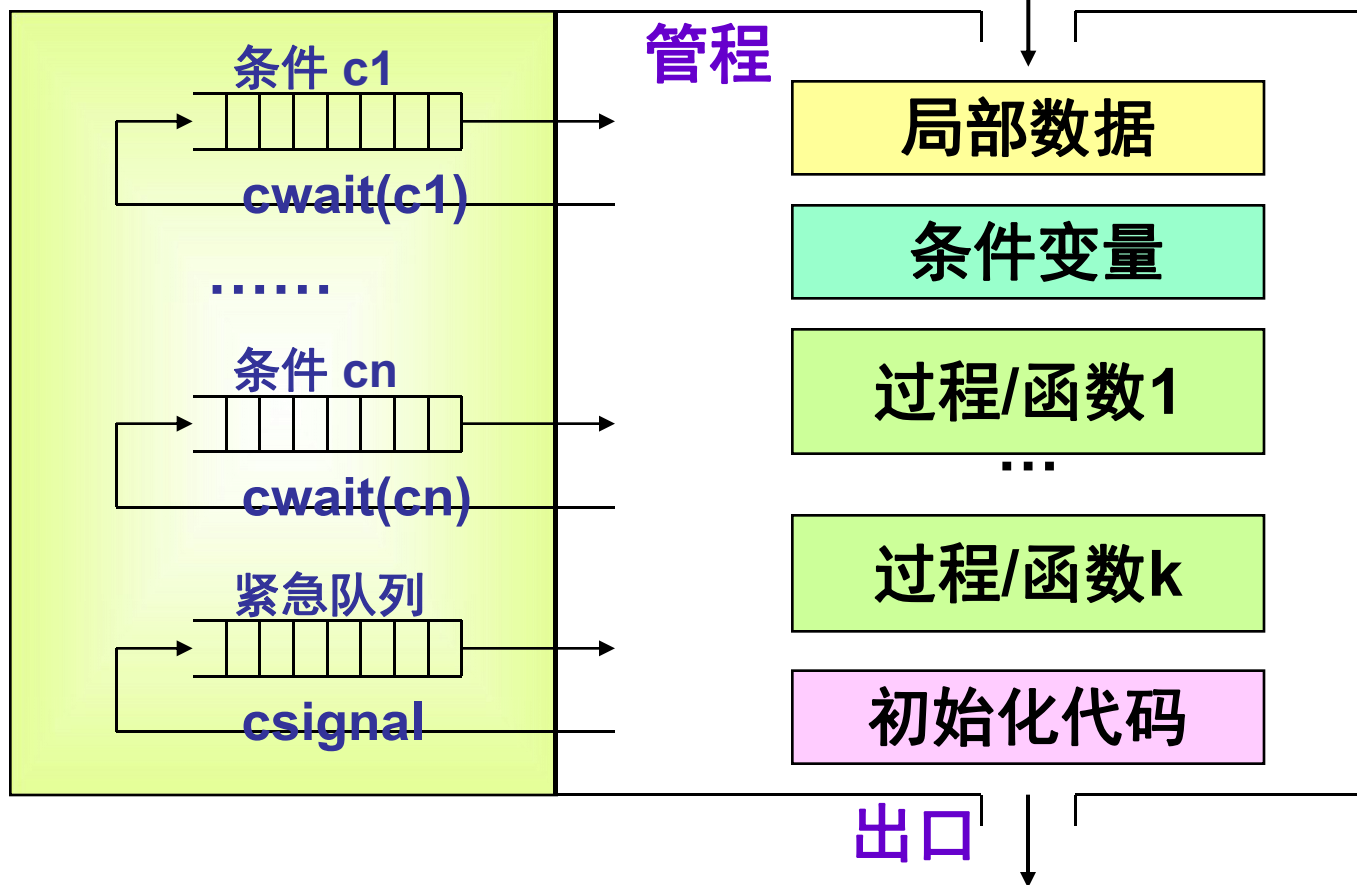


5.5.1 使用信号的管程—内部队列



◆ 管程的结构及内部队列:

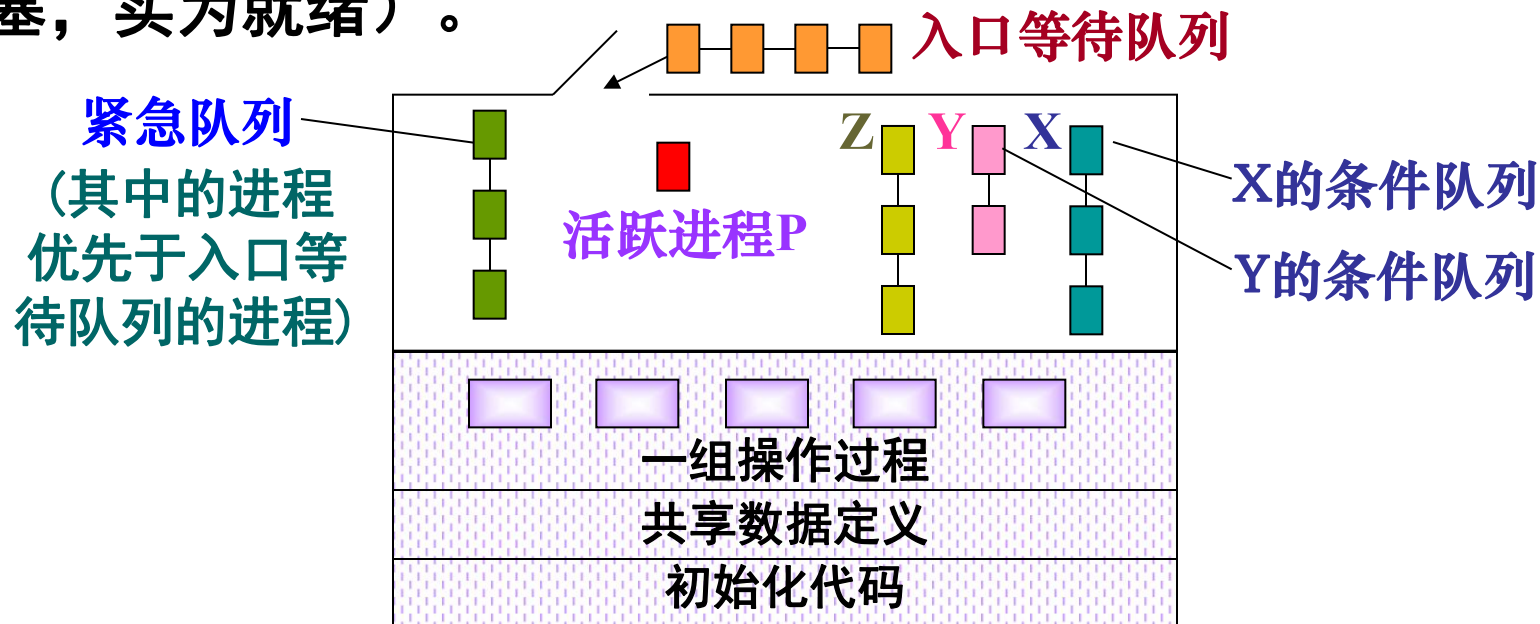
管程内的等待队列



5.5.1 使用信号的管程 — 实现互斥/同步



- ❖ **互斥**：管程内总是只有一个活跃进程，实现对共享数据/资源的互斥使用。不能进入管程的进程排在**入口等待队列**中。其它已进入管程的进程处于阻塞状态。
- ❖ **同步**：不能满足进程Q请求时，`cwait(x)`使Q进入x的**条件队列**（阻塞）。另一进程P释放资源时，调用 `csignal(x)` 将Q唤醒。然后Q继续执行，而P进入**紧急队列**（名为阻塞，实为就绪）。



5.5.1 使用信号的管程——解决生产者/消费者问题



◆ 先定义管程：

```
monitor boundedbuffer
char buffer[N];
int in, out, count;
cond notfull, notempty;
```

```
void append(char x)
{ if (count==N)
    cwait(notfull);
    /*缓冲池满，阻塞*/
    buffer[in]=x;
    in=(in+1)%N;
    count++;
    csignal(notempty);
} //生产者送数
```

count是缓冲池中的有效缓冲数目；
notfull, notempty是条件变量。

```
void take(char x)
{ if (count==0)
    cwait(notempty);
    /*缓冲池空，阻塞*/
    x=buffer[out];
    out=(out+1)%N;
    count--;
    csignal(notfull);
} //消费者取数
```

```
{ in=0; out=0; count=0; }
```

5.5.1 使用信号的管程 — 解决生产者/消费者问题



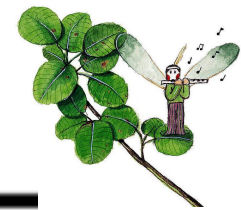
◆ 然后，在用户程序中调用管程提供的append()和take()过程：

```
■ void producer()
{ char x;
  while (true)
  { produce(x);
    append(x);
  }
}
```

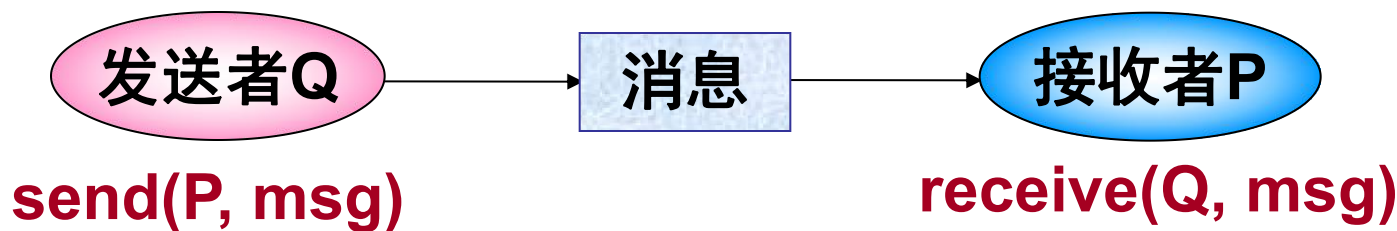
```
■ void consumer()
{ char x;
  while (true)
  { take(x);
    consume(x);
  }
}
```

5.5.2 使用通知和广播的管程（略）

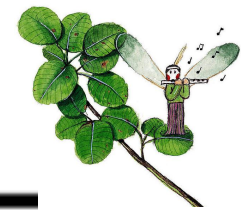
5.6 消息传递



- ◆ 进程间通过“消息传递”交换信息。
- ◆ 消息传递的两个原语：
 - **send(P, message)**—给进程P发消息message
 - **receive(Q, message)**—接收来自进程Q的消息

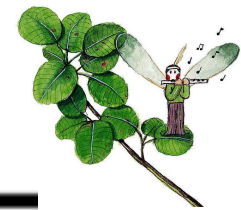


5.6.1 消息传递—同步

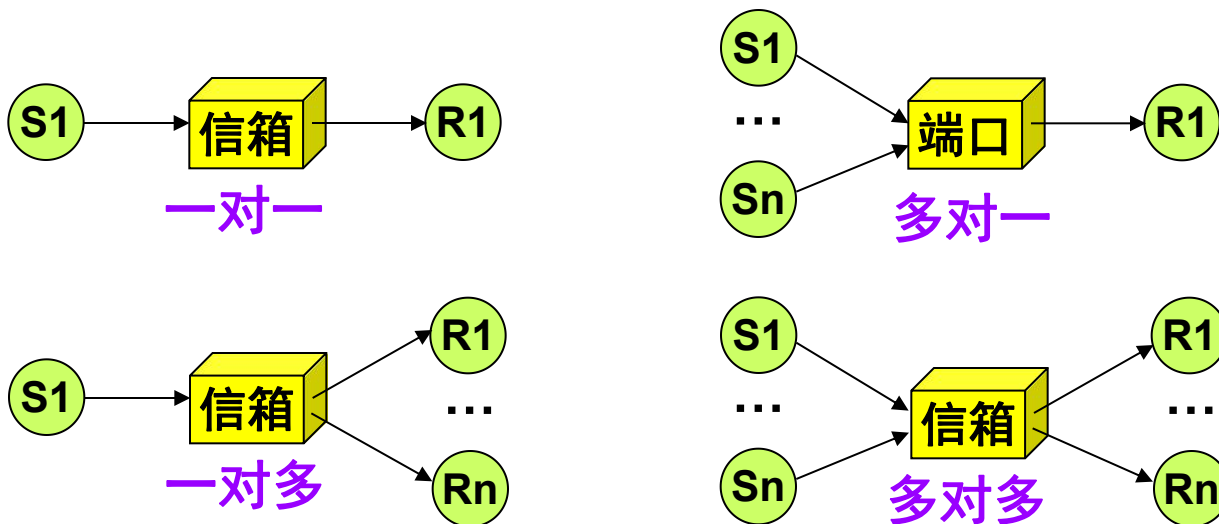


- ◆ **同步**：发送者发出消息后，接收者才能接收。
- ◆ 当一个进程执行send()时，该进程可以：
 - 不阻塞，继续执行；
 - 被阻塞，直到这个消息被接收者接收。
- ◆ 当一个进程执行receive()时，该进程可以：
 - 不阻塞，接收已发来的消息或放弃接收，继续执行；
 - 被阻塞，直到所等待的消息到达。
- ◆ **无阻塞式send** 和 **阻塞式receive** 最常用。

5.6.2 消息传递—寻址



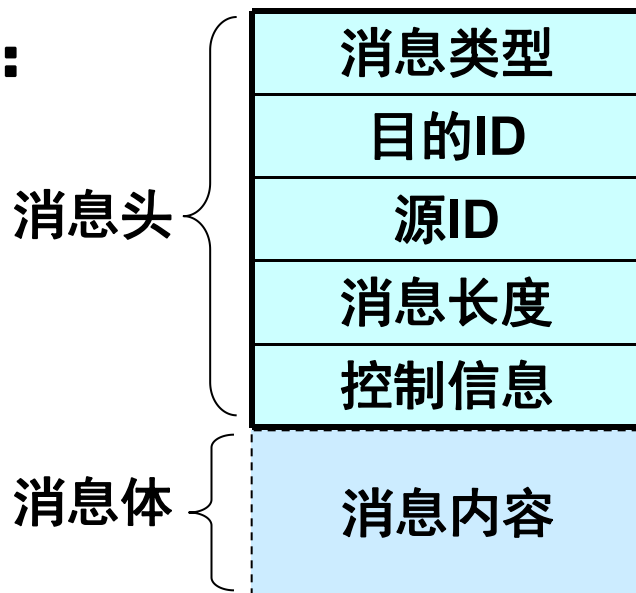
- ◆ send 和 receive 如何指明接收者和发送者？
- ◆ **直接寻址**：指明目标进程或源进程的标识ID。
 - 公共服务进程使用receive()时，不指明源进程(隐式)。
- ◆ **间接寻址**：通过 **信箱** 发送和接收消息。
 - 信箱A：send (A, msg), receive(A, msg)



5.6.3 消息传递—消息格式



◆ 一般消息格式：

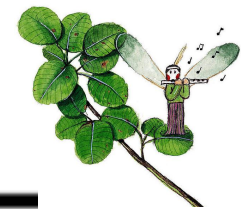


5.6.4 消息传递—排队原则



- ⑩ **消息队列**：先入先出（最常见）。
- ⑩ 可指定消息优先级，高级消息先被接收。
- ⑩ 接收者可检查消息队列，并选择接收哪个消息。

5.6.5 消息传递—互斥



◆ 用消息和两个信箱解决生产者/消费者问题：

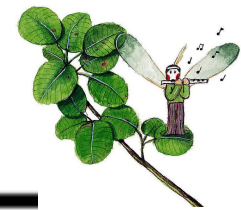
- mayconsume信箱：用作有界缓冲池，存放消息。
- mayproduce信箱：“空消息”邮箱。最初填满空消息。
空消息数=空缓冲数。只用于同步。

```
void producer()  
{ message pmsg; //存放生产者数据的临时缓冲  
  while (true) {  
    receive (mayproduce, pmsg); //接收空消息(空缓冲数减1)。无空消息时阻塞  
    pmsg = produce(); //生产的数据放到临时缓冲pmsg中  
    send (mayconsume, pmsg); //将pmsg放到有界缓冲池mayconsume  
  }  
}
```

阻塞式receive(), 即无消息时阻塞, 直至消息到来。

```
void consumer()  
{ message cmsg;  
  while (true) {  
    receive (mayconsume, cmsg); //接收一个缓冲。阻塞时表明缓冲池全空  
    consume (cmsg);  
    send (mayproduce, null); //发送空消息, 实际是使空缓冲数量增1  
  }  
}
```

作业：



- ◆ 1、写出信号量定义，semWait和semSignal原语，以及用信号量实现互斥的伪代码。（V8图5.3，V9图5.6）
- ◆ 2、假设一个阅览室有**100个座位**，没有座位时读者在阅览室外等待；每个读者进入阅览室时都必须在阅览室门口的一个**登记本**上登记座位号和姓名，然后阅览，离开阅览室时要去掉登记项。每次只允许一个人登记或去掉登记。用信号量操作描述读者的行为。
- ◆ 3、独木桥问题：东、西向汽车过独木桥。桥上无车时允许一方汽车过桥，待全部过完后才允许另一方汽车过桥。用信号量操作写出同步算法。（提示：参考读者优先中“读者”的解法，第1辆车加锁，最后1辆车解锁）

