

# 第6章 进程同步

孙承杰

E-mail: [sunchengjie@hit.edu.cn](mailto:sunchengjie@hit.edu.cn)

哈工大计算学部人工智能教研室

2023年秋季学期

# 主要内容

6.1 背景

6.2 互斥与临界区问题

6.3 临界区问题解决方法

(1) 一般软件方法

(2) 关中断方法

(3) 硬件原子指令方法

(4) 信号量方法

6.4 进程同步

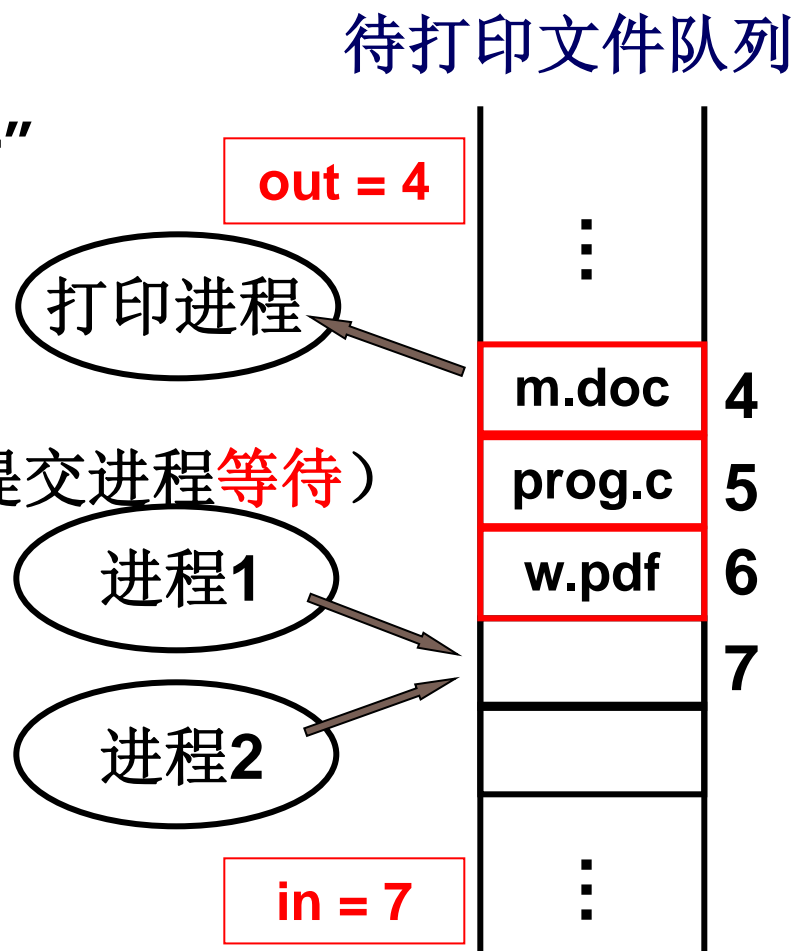
## 6.1 背景

### 进程工作为什么要同步？

- 多个进程共同完成一个任务!!!
- 在进程合作过程中，除了“并行”的工作之外，还经常出现相互等待的“协作”过程。

#### ■ 想一想打印工作过程

- 进程提交打印任务（其他欲提交进程等待）
- 进程去做其它工作
- 打印任务被放进打印队列
- 打印进程从队列中取出任务
- 打印进程控制打印机打印



这是一个典型的“生产者-消费者”问题！

# 生产者-消费者

## 共享数据

```
#define BUFFER_SIZE 10
typedef struct { . . . } item;
item buffer[BUFFER_SIZE];
int in = out = counter = 0;
```

## 生产者进程

```
while (true) {
    while(counter == BUFFER_SIZE)
        ; /*仓库已满，等待消费*/
    buffer[in] = item; /*生产入库*/
    in = (in + 1) % BUFFER_SIZE;
    counter++;
}
```

## 消费者进程

```
while (true) {
    while(counter == 0)
        ; /*仓库无货，等待生产*/
    item = buffer[out]; /*消费出库*/
    out = (out + 1) % BUFFER_SIZE;
    counter--;
}
```

# 生产者-消费者 – 简化代码

共享数据

```
#define BUFFER_SIZE 10  
int counter = 0;
```

生产者进程

```
while (true) {  
    while(counter == BUFFER_SIZE)  
        ; /*仓库已满, 等待消费*/  
    counter++;  
}
```

消费者进程

```
while (true) {  
    while(counter == 0)  
        ; /*仓库无货, 等待生产*/  
    counter --;  
}
```

# 生产者-消费者引出的一个问题

- 共享变量**counter**可能会出现读写错误
- 如生产者进程和消费者进程各执行一次

初始情况

```
counter = 5;
```

生产者P

```
register = counter;  
register = register + 1;  
counter = register;
```

消费者C

```
register = counter;  
register = register - 1;  
counter = register;
```

一个可能的执行序列6

```
P.register = counter;  
P.register = P.register + 1;  
P.re  
P.re  
C.register = counter;  
C.register = C.register - 1;  
C.re  
C.re  
counter = C.register; //4  
counter = P.register; //6  
cou  
cou
```

正常结果 = 5, 但实际结果可能为4/5/6 !!

## 6.2 互斥与临界区问题

**术语1: 竞争条件(Race Condition)**多个进程（任务）并发访问和操作同一（组）数据且执行结果与访问发生的特定顺序有关

第i次执行

```
P.register = counter;  
P.register = P.register + 1;  
C.register = counter;  
C.register = C.register - 1;  
counter = P.register;  
counter = C.register;
```

第j次执行

```
P.register = counter;  
P.register = P.register + 1;  
counter = P.register;  
C.register = counter;  
C.register = C.register - 1;  
counter = C.register;
```

- 含义1: 多个进程并发访问和操作共享数据
- 含义2: 和时间有关就是和调度顺序有关
- 经验: 这样的错误非常难于调试!!!

# 避免竞争条件的一个尝试

- 在写**counter**时阻断其他进程访问**counter**

一个可能的执行序列

```
P.register = counter;  
P.register = P.register + 1;  
C.register = counter;  
C.register = C.register - 1;  
counter = P.register;  
counter = C.register;
```

- 更衣室、洗手间如何操作？

生产者P

给**counter**上锁

```
P.register = counter;  
P.register = P.register + 1;
```

消费者C

检查**counter**锁

生产者P

```
counter = P.register;
```

给**counter**开锁

消费者C

给**counter**上锁

```
C.register = counter;  
C.register = C.register - 1;  
counter = C.register;
```

给**counter**开锁

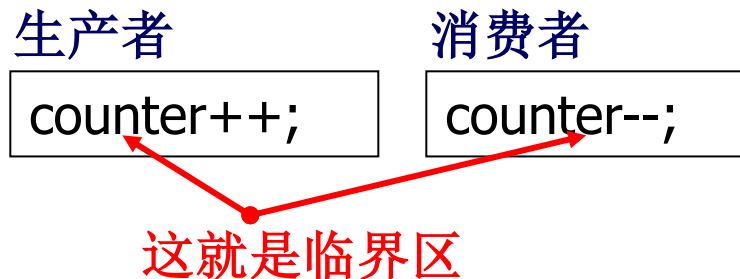
共享数据一次只允许一个进程操作



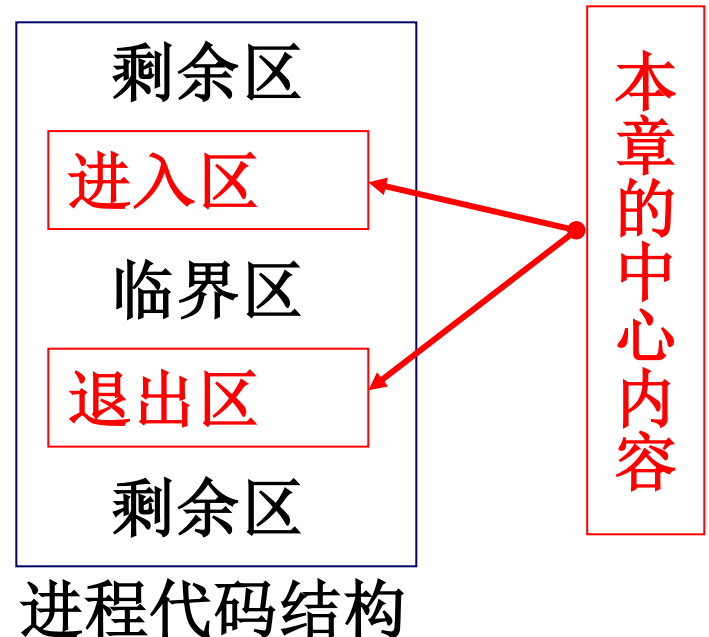
## 6.2 互斥与临界区问题

### 术语2: 临界区(Critical Section)

- (1) 临界资源: 把一次仅允许一个进程使用的资源称为临界资源。  
如只能独享的物理设备(打印机)、共享变量等等。
- (2) 临界区: 在每个进程中, 访问临界资源的那段程序称为临界区。



- 临界区定义了进程间的协作协议, 因此一个非常重要的工作: 找出进程中的临界区代码并限定



## 6.2 互斥与临界区问题

### 如何进入临界区? 有多种方法!

临界区问题的解决方案，必须满足下面3个条件：

■ **1. 互斥进入：** 如果一个进程在临界区中执行，则其他进程不允许进入

- 这种进程间的约束关系称为**互斥(mutual exclusion)**
- 这是临界区进入的基本原则(正确性保证)

一个好的临界区进入方法还需考虑…

■ **2. 有空让进：** 若干进程要求进入空闲临界区（临界资源空闲）时，应一定能使某个进程进入临界区

■ **3. 有限等待：** 从进程发出进入请求到允许进入，不能无限等待

## 6.3 临界区问题解决方法

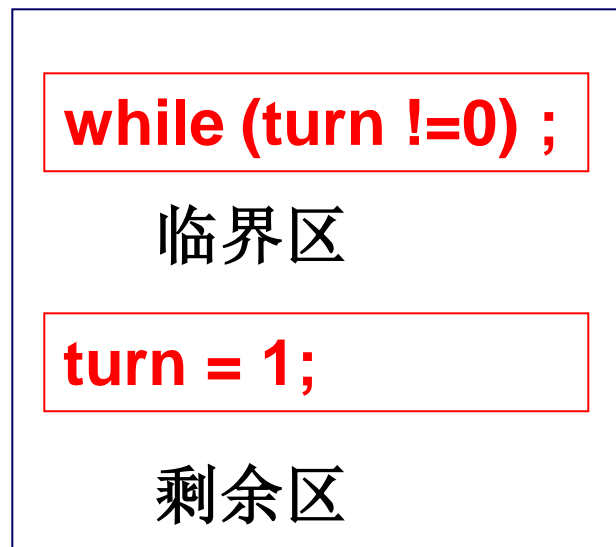
**解决临界区问题 = 如何进入和退出临界区，  
以达到进程互斥及同步的目的！**

有多种方法：

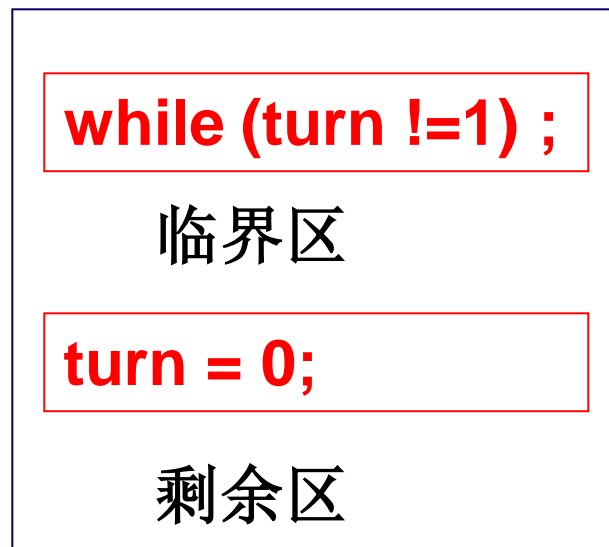
- (1) 一般软件方法
- (2) 关中断方法
- (3) 硬件原子指令方法
- (4) 信号量方法

## 6.3 临界区问题解决方法

### 6.3.1 一般软件方法 (1) 轮换法（值日法）



进程 $P_0$



进程 $P_1$

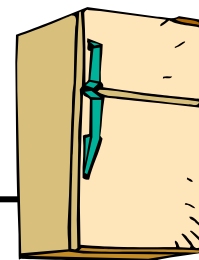
- 上面的轮换法类似于值日
- 满足互斥进入要求
- 问题:  $P_0$ 完成后不能接着再次进入, 尽管进程 $P_1$ 不在临界区...(不满足“有空让进”)(只能交替进入)

## 6.3 临界区问题解决方法

### 6.3.1 一般软件方法 (2) 标记法

■ 想别的办法...

可借鉴生活中的道理



时间	丈夫	妻子
3:00	打开冰箱，没有牛奶了	
3:05	离开家去商店	
3:10	到达商店	打开冰箱，没有牛奶了
3:15	买牛奶	离开家去商店
3:20	回到家里，牛奶放进冰箱	到达商店
3:25		买牛奶
3:30		回到家里，牛奶放进冰箱

■ 更好的方法应该是立即去买，留一个便条（发一条消息）

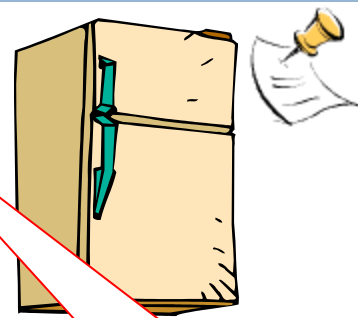
# 6.3 临界区问题解决方法

## 6.3.1 一般软件方法 (2) 标记法

```
if(noNote) {  
    leave Note;  
    buy milk;  
    remove Note;  
}
```



```
leave Note;  
if(noOtherNote) {  
    buy milk;  
}  
remove Note;
```



计算机考虑  
问题的方式

互斥访问  
有空让进  
有限等待

```
flag[0] = true;  
while (flag[1]) ;
```

临界区

```
flag[0] = false;
```

剩余区

进程 $P_0$

```
flag[1] = true;  
while (flag[0]) ;
```

临界区

```
flag[1] = false;
```

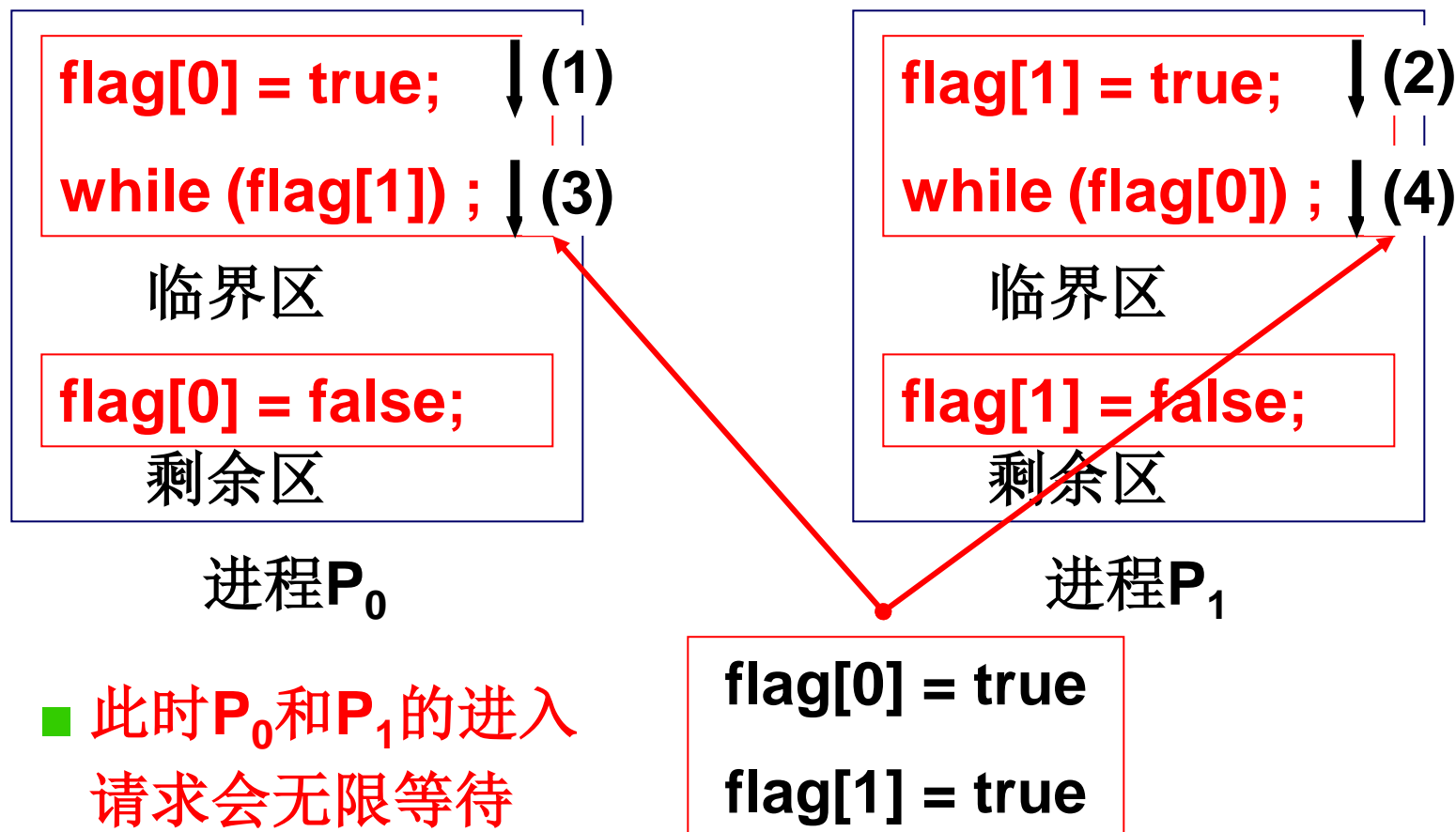
剩余区

进程 $P_1$

## 6.3 临界区问题解决方法

### 6.3.1 一般软件方法 (2) 标记法

#### ■ 考虑下面的执行顺序



## 6.3 临界区问题解决方法

### 6.3.1 一般软件方法 (3) Peterson算法

- 结合了标记和轮转两种思想

```
flag[0] = true;  
turn = 1;  
while (flag[1] && turn == 1) ;
```

临界区

```
flag[0] = false;
```

剩余区

进程 $P_0$

```
flag[1] = true;  
turn = 0;  
while (flag[0] && turn == 0) ;
```

临界区

```
flag[1] = false;
```

剩余区

进程 $P_1$



## 6.3 临界区问题解决方法

### 6.3.1 一般软件方法 (3) Peterson算法

#### Peterson算法的正确性

- 满足互斥进入:

如果两个进程都进入, 则  
 $\text{flag}[0]=\text{flag}[1]=\text{true}$ ,  
 $\text{turn}==0==1$ , 矛盾!

- 满足有空让进:

如果进程 $P_1$ 不在临界区, 则  
 $\text{flag}[1]=\text{false}$ , 或者 $\text{turn}=0$ ,  
则 $P_0$ 能进入!

- 满足有限等待:

$P_0$ 进入临界区,  $\text{flag}[0]=\text{true}$ ;  $P_1$ 欲进临界区有 $\text{turn}=0$ ,  
 $P_1$ 不再改变 $\text{turn}$ 值, 所以后面的 $P_1$ 会循环等待,  $P_0$ 退出

```
flag[0] = true;  
turn = 1;  
while (flag[1] && turn == 1) ;
```

临界区

```
flag[0] = false;
```

剩余区

进程 $P_0$

```
flag[1] = true;  
turn = 0;  
while (flag[0] && turn == 0) ;
```

临界区

```
flag[1] = false;
```

剩余区

进程 $P_1$

# 多个进程怎么办? – 面包店算法

## ■ 仍然是标记和轮转的结合

■ **如何轮转:** 每个进程都获得一个序号, 序号最小的进入

■ **如何标记:** 进程离开时序号为0, 不为0的序号即标记

■ **面包店:** 每个进入商店的客户都获得一个当前号码, 号码最小的先得到服务; 号码相同时, 名字靠前的先服务。

进程  
 $P_i$

```
choosing[i] = true; num[i] = max(num[0], ..., num[n-1])+1;  
choosing[i] = false; for(j=0; j<n; j++) { while(choosing[j]);  
while ((num[j] != 0) && (num[j], j)<(num[i], i)); }
```

临界区

```
num[i] = 0;
```

剩余区

若 $a < c$ ; 或 $a = c$ 和 $b < d$ 同时成立  $\rightarrow (a, b) < (c, d)$

# 面包店算法的正确性

进程  
 $P_i$

```
choosing[i] = true; num[i] = max(num[0], ..., num[n-1])+1;  
choosing[i] = false; for(j=0; j<n; j++) { while(choosing[j]);  
while ((num[j] != 0) && (num[j], j)<(num[i], i)); }
```

临界区

```
num[i] = 0;
```

- **满足互斥进入:**  $P_i$ 在临界区内,  $P_k$ 试图进入, 一定有  $(num[i], i) < (num[k], k)$ ,  $P_k$ 循环等待。
- **满足有空让进:** 如果没有进程在临界区中, 最小序号的进程一定能够进入。
- **满足有限等待:** 离开临界区的进程再次进入一定排在最后(**FIFO**), 所以任一个想进入的进程至多等待 $n-1$ 个进程

# 6.3 临界区问题解决方法

## 6.3.1 一般软件方法

### 临界区问题的一般软件方法讨论：

- 轮换法、标记法：
  - 不满足解决临界区问题的所有3个要求（部分满足）
- Peterson算法：
  - 只适合2个进程情况；
- 面包店算法：
  - 可以适用n个进程的情况，但太过复杂
- 一般软件方法，从机器语言（指令）角度看：
  - “进入区”和“退出区”代码本身也是“临界区”
- 操作系统用系统调用进行“封装”后可以实用：
  - 但“忙等待”现象需要消除（浪费CPU资源）

# 6.3 临界区问题解决方法

## 6.3.2 关中断方法

让用户考虑这样复杂的事显然不合适

- 怎么办？ 给出简单的互斥办法
- 回顾一下临界区的概念： 只允许一个进程进入
- 再想一想： 另一个进程怎么样能执行？
  - 被调度： 另一个进程只有被调度才能执行，才可能进入临界区
  - 自然就能想到一个互斥办法： 阻止调度
  - 怎么阻止调度？ 大胆设想： 将中断暂时关闭！！！！

## 6.3 临界区问题解决方法

### 6.3.2 关中断方法



■ 优点: 简单

■ 缺点: 风险高

```
while (true) {  
    cli();  
    while(counter == BUFFER_SIZE);  
    buffer[in] = item;  
    in = (in + 1) % BUFFER_SIZE;  
    counter++;  
    sti();  
}
```

- 让用户开关中断是非常危险且不方便;
- 基于时钟的时间片轮转调度有效! 优先级抢占调度不适用。

## 6.3 临界区问题解决方法

### 6.3.3 硬件原子指令方法

- 如果右边的三条机器指令原子执行就解决问题了!

需修改一下硬件!

```
P.register = counter;  
P.register = P.register + 1;  
counter = P.register;
```

- 显然针对右边的代码没法设计这样的原子指令! 怎么办?

```
while (true) {  
    while(counter == BUFFER_SIZE);  
    buffer[in] = item;  
    in = (in + 1) % BUFFER_SIZE;  
    counter++;  
}
```

## 6.3 临界区问题解决方法

### 6.3.3 硬件原子指令方法

#### ■ 提供硬件“加锁”原子指令 **TestAndSet**

```
boolean TestAndSet(boolean  
    &target)  
{  
    boolean rv = target;  
    target = true;  
    return rv;  
}
```

一次  
执行  
完毕

非常巧妙简洁的代码  
Lock加锁true，返回值也为true

```
while(TestAndSet  
    (&lock)) ;
```

临界区

```
lock = false;
```

剩余区

该条语句的机器码是  
一条，当然是原子的

- 如果lock一开始为false，则返回值为false，lock变为true
- 如果Lock为true，加锁true，返回值也为true

#### ■ 内存加锁和汇编指令保证一次执行完



## 6.3 临界区问题解决方法

### 6.3.4 信号量方法

一般软件方法、关中断方法、硬件原子指令方法小结：

- 解决了临界区**进出互斥**的问题
- 一般软件法未解决“**忙等待**”问题
- 关中断方法、硬件原子指令方法不方便、麻烦，操作系统停止调度
- 用来解决**多进程同步**问题很复杂、不方便

“**信号量**”方法是前面方法的更“**一般化**”体现

# 6.3 临界区问题解决方法

## 6.3.4 信号量方法

### 信号量 – 由伟大人物提出的伟大概念！！

- 信号量: 1965年, 由荷兰学者Dijkstra提出的一种特殊整型变量。
- 信号量定义: 1个数据结构+2个基本操作

```
struct semaphore
{
    int value;    /*记录资源个数或等待资源进程个数*/
    PCB *queue;  /*等待在该信号量上的进程队列*/
}

P(semaphore s); /*分配资源或组织进程排队等待并记录
                排队进程数*/

V(semaphore s); /*资源归还或唤醒等待的进程*/
```

# 6.3 临界区问题解决方法

## 6.3.4 信号量方法

### 信号量的概念：

- 信号量是一个确定的二元组  $(s, q)$
- 其中  $s$  是一个具有非负初值的整形变量， $q$  是一个初始状态为空的队列
- 整形变量  $s$  表示系统中某类资源的数目：
  - 当其值  $\geq 0$  时，表示系统中当前可用资源的数目
  - 当其值  $< 0$  时，其绝对值表示系统中因请求该类资源而被阻塞的进程数目
- 除信号量的初值外，信号量的值仅能由  $P$  操作和  $V$  操作更改，操作系统利用它的状态对进程和资源进行管理

# 6.3 临界区问题解决方法

## 6.3.4 信号量方法

### 信号量的概念：

#### P操作：

- P操作记为P(s)，其中s为一信号量，它执行时主要完成以下动作：
  - $s.value = s.value - 1$ ； /\*可理解为占用1个资源，若原来就没有则记帐“欠”1个\*/
  - 若 $s.value \geq 0$ ，则进程继续执行
  - 否则（即 $s.value < 0$ ），则进程被阻塞，并将该进程插入到信号量s的等待队列s.queue中
- 说明：实际上，P操作可以理解为分配资源的计数器；或是使进程处于等待状态的控制指令

# 6.3 临界区问题解决方法

## 6.3.4 信号量方法

### 信号量的概念：

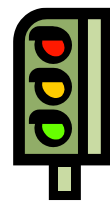
#### V操作：

- V操作记为V(s)，其中s为一信号量，它执行时，主要完成以下动作：
  - $s.value = s.value + 1$ ； /\*可理解为归还1个资源，若原来就没有则意义是用此资源还1个欠帐\*/
  - 若 $s.value > 0$ ，则进程继续执行；
  - 否则（即 $s.value \leq 0$ ），则从信号量s的等待队s.queue中移出第一个进程，使其变为就绪状态。
  - 说明：实际上，V操作可以理解为归还资源的计数器；或是唤醒进程使其处于就绪状态的控制指令

# 信号量 vs. 锁

## ■ 信号量是一个一般的锁...

- 锁:  $\{0,1\}$  信号量:  $\{-\infty, \dots, -2, -1, 0, 1, 2, \dots, \infty\}$
- 信号量即信号的数量，唤醒实际上就是发一个信号
- 开锁动作也是发一个信号



## ■ 信号量的更多用途...

- 可以用来实现互斥（0-1，又叫互斥信号量）
- 可以记录资源个数( $s.value \geq 0$ )
- 可以记录等待进程个数( $|s.value|$  ,  $s.value < 0$ )
- 可以用来实现复杂的进程间同步关系

## 6.3 临界区问题解决方法

### 6.3.4 信号量方法

#### 信号量的应用1:

**例1：**利用信号量及其**P、V**操作能方便地实现进程互斥。

设**S**为一**互斥信号量（0、1）**，其初值=1，表示某临界资源未被占用。利用信号量实现并发进程**P1、P2**互斥访问临界区的描述如下：

进程P1

.

.

**P(S)**

临界区

**V(S)**

.

.

进程P2

.

.

**P(S)**

临界区

**V(S)**

.

.

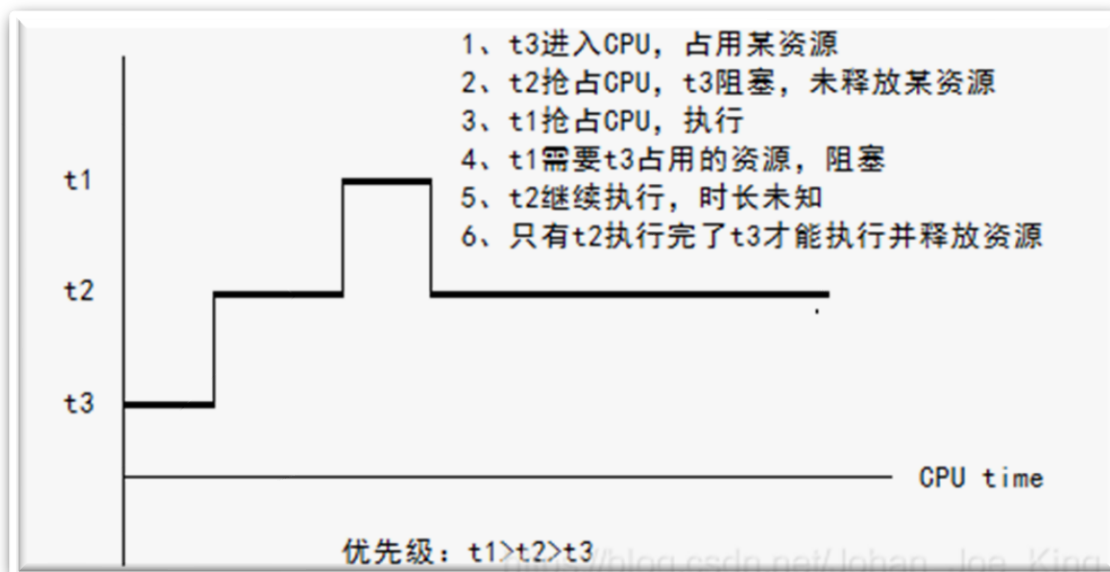
还有没有忙等待的问题？如果有另外一个中等优先级进程存在？

## 6.3 临界区问题解决方法

### 6.3.4 信号量方法

- 优先级翻转问题(priority inversion): 基于优先级抢占方式, 即当一个高优先级任务访问共享资源时, 资源已被一低优先级任务占有, 而这个低优先级任务在访问共享资源时可能又被其它一些中等优先级任务抢占, 因此造成高优先级任务被许多具有较低优先级任务阻塞。
- 解决办法:
  - ① 优先级天花板(priority ceiling): 当任务申请某资源时, 把该任务的优先级提升到可访问这个资源的所有任务中的最高优先级。
  - ② 优先级继承(priority inheritance): 占有资源的低优先级任务阻塞了高优先级任务时, 提升低优先级到高优先级。

优先级继承, 只有当占有资源的低优先级的任务被阻塞时, 才会提高占有资源任务的优先级; 而优先级天花板, 不论是否发生阻塞, 都提升。





## 6.3 临界区问题解决方法

### 6.3.4 信号量方法

#### 信号量的应用2:

**例2:** 假如系统中有2台打印机可用;  
现有4个进程P1, P2, P3, P4都在不同时间里  
以不同数量申请该设备。

**S初值=2**, 表示共有2台打印机可用。

**P1、P2、P3、P4**为并发进程, 本例假设第1  
个被调度的为**P2**。

列出使用**P、V**操作使这4个进程互斥工作过程。

# 6.3 临界区问题解决方法

## 6.3.4 信号量方法

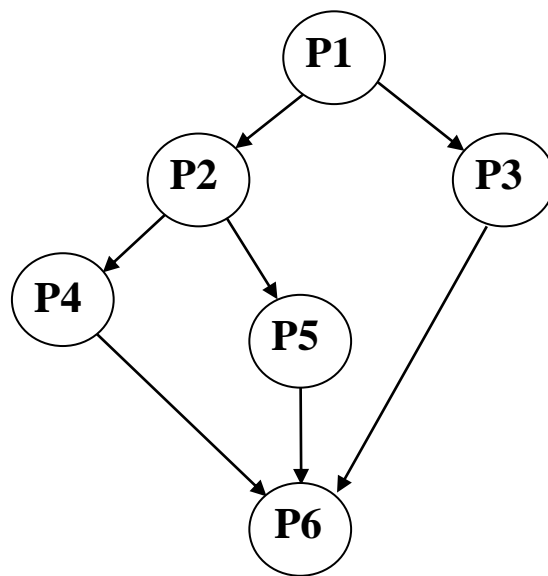
并发进程Pi工作序列 ( )数字表示运行顺序				并发执行 顺序	当前处于 运行态 进程	所执行的 操作 (P/V)	信号量S 的值 (初值=2)	被唤醒的 进程	信号量S的 等待队列
P1	P2	P3	P4						
·	·	·	·	(1)					
·	·	·	·	(2)					
(2) P(S)	(1) P(S)	(3) P(S)	(5) P(S)	(3)					
·	·	·	·	(4)					
打印	打印	打印	打印	(5)					
·	·	·	·	(6)					
(6) V(S)	(4) P(S)	(7) V(S)	(10) V(S)	(6)					
·	·	·	·	(7)					
·	打印	·	·	(8)					
·	(9) V(S)	(8) P(S)	·	(9)					
	·	·	·	(10)					
	打印	打印	·	(11)					
	(11) V(S)	(12) V(S)	·	(12)					
	·	·	·						
	·	·	·						
	·	·	·						

## 6.3 临界区问题解决方法

### 6.3.4 信号量方法

#### 信号量的应用3:

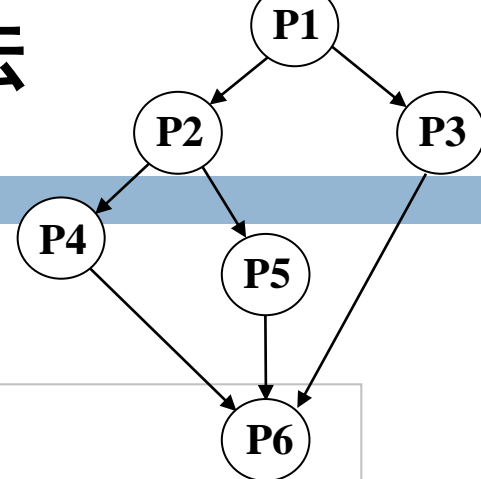
**例3:** 若干进程协作完成一个共同任务而并发执行。  
下图描述进程执行先后次序的前趋图。  
使用P、V操作写出使这6个进程同步的过程。



描述进程执行先后次序的前趋图

## 6.3 临界区问题解决方法

### 6.3.4 信号量方法



#### 信号量的应用3:

**分析：**

- (1) 图中说明任务启动后**P1**先执行；
- (2) 当**P1**结束后，**P2**和**P3**可以开始；
- (3) 当**P2**完成后，**P4**，**P5**可以开始执行；
- (4) 仅当**P3**，**P4**，**P5**都执行完后，**P6**才能开始执行；
- (5) 后面的进程需要等待“上游”进程完成的“通知”。

描述进程执行先后次序的前趋图

**解答：** 需要设置5个同步信号量f1、f2、f3、f4、f5，分别表示进程P1、P2、P3、P4、P5是否执行完成，其初值均为0（未完成，或未发“完成通知”）。

```
int f1=0 /* 表示进程P1是否执行完成 */
int f2=0 /* 表示进程P2是否执行完成 */
int f3=0 /* 表示进程P3是否执行完成 */
int f4=0 /* 表示进程P4是否执行完成 */
int f5=0 /* 表示进程P5是否执行完成 */
```

# 6.3 临界区问题解决方法

## 6.3.4 信号量方法

### 信号量的应用3:

```
main( )
{  cobegin
    P1( ); P2( ); P3( ); P4( ); P5( ); P6( );
  coend
}
```

```
P1( )
{  ...
  v(f1);
  v(f1);
}
```

```
P2( )
{  p(f1);
  ...
  v(f2);
  v(f2);
}
```

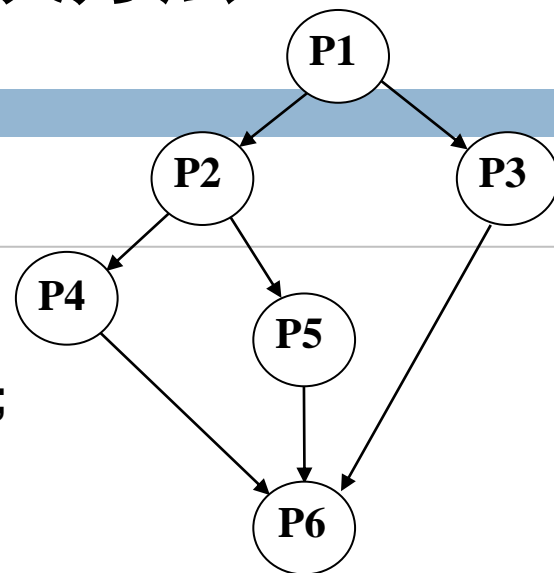
*/\*P1相关任务代码\*/*

描述进程执行先后次序的前趋图

*/\*P1完成后会发出2个通知，与P1作为前驱处理的2个进程可以工作了。尽管此处不能看出来到底是通知哪个进程的，但从后面P2()与P3()代码中可以看出\*/*

*/\*①检查P1是否完成；②若未完成，则P2处于阻塞态并等待P1唤醒；③若P1已完成，则继续执行下面内容\*/*

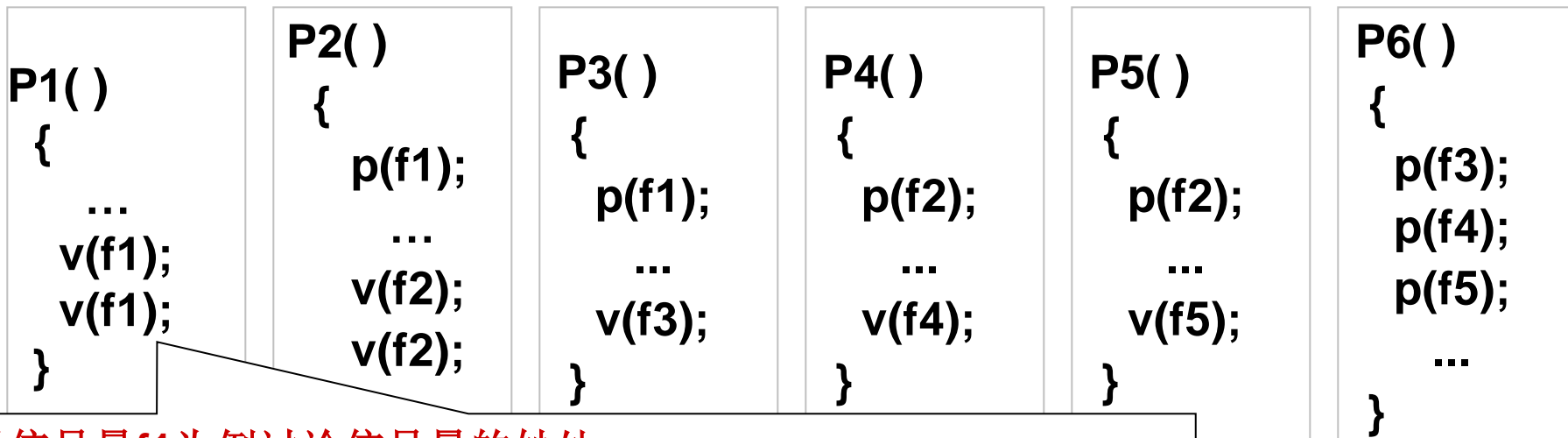
*/\*P2相关任务代码\*/*



# 6.3 临界区问题解决方法

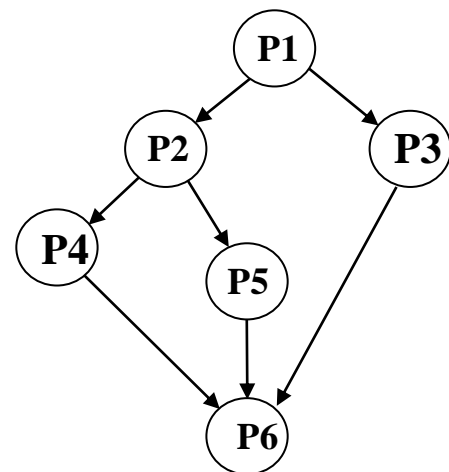
## 6.3.4 信号量方法

### 信号量的应用3:



以信号量f1为例讨论信号量的妙处:

- (1)如果P1被先调度并完成全部代码，则f1被增加2，即表示f1“资源”有2个可用；若之后P2、P3被调度可以全部通过p(f1)的“验证”并顺利完成全部代码；
- (2)如果P1被先调度，当执行到第1个v(f1)时，即表示f1“资源”有1个可用；若之后P1让出CPU，则P2或P3任意一个进程被调度，可以全部完成代码；
- (3)如果P2或P3被先调度，则它们只能执行到p(f1)位置，此时f1=-1或-2，P2或P3（或2个都）被阻塞到f1的队列中，直到P1运行完成，它们才可被唤醒并继续运行



## 6.3 临界区问题解决方法

### 6.3.5 锁Lock

**锁Lock：** 用来对多个线程之间共享的临界区进行保护

POSIX threads(简称Pthreads)是支持多核平台上进行多线程编程的一套API。线程同步最典型的方法就是用Pthreads提供的锁机制(Lock)。

Pthreads提供了多种锁机制：

- Mutex(互斥锁): pthread\_mutex\_t
- Spin lock(自旋锁): pthread\_spin\_t
- Read/Write lock(读写锁): pthread\_rwlock\_t
- Condition Variable(条件变量): pthread\_cond\_t

## 6.3 临界区问题解决方法

### 6.3.5 锁Lock

**锁Lock：**用来对多个线程之间共享的临界区进行保护

POSIX threads(简称Pthreads)是支持多核平台上进行多线程编程的一套API。线程同步最典型的方法就是用Pthreads提供的锁机制(Lock)。

Mutex（互斥锁）属于sleep-waiting类型的锁。

- ① 例如在一个双核的机器上有两个线程（线程A和线程B），它们分别运行在Core0和Core1上。假设线程A想要通过pthread\_mutex\_lock操作去得到一个临界区的锁，而此时这个锁正被线程B所持有，那么线程A就会被阻塞。
- ② Core0会在此时进行上下文切换(Context Switch)将线程A置于等待队列中，此时Core0就可以运行其它的任务而不必进行忙等待。

Spin lock（自旋锁），它属于busy-waiting类型的锁。

与互斥锁相比，如果线程A是使用pthread\_spin\_lock操作去请求锁，那么线程A就会一直在Core0上进行忙等待并不停的进行锁请求，直到得到这个锁为止。没有调度和线程切换的过程。



## 6.3 临界区问题解决方法

### 6.3.5 锁Lock

**锁Lock：**用来对多个线程之间共享的临界区进行保护

POSIX threads(简称Pthreads)是支持多核平台上进行多线程编程的一套API。线程同步最典型的方法就是用Pthreads提供的锁机制(Lock)。

**互斥锁与自选锁比较：**

互斥锁更适用于临界区持锁时间比较长的操作，比如：

- ① 临界区有IO操作
- ② 临界区代码复杂或者循环量大
- ③ 临界区竞争非常激烈
- ④ 单核处理器

自旋锁就主要用在临界区持锁时间非常短且CPU资源不紧张的情况下。所以一般用在多核的服务器时较多。短期自旋不引起调度和上下文切换，效率更高。

# 6.3 临界区问题解决方法

## 6.3.5 锁Lock

**锁Lock:** 用来对多个线程之间共享的临界资源进行互斥访问

POSIX threads(简称Pthreads)是支持多核平台上线程同步最典型的方法就是用Pthreads提供的锁机制

互斥锁与自旋锁的实现过程比较

采用TestAndSet被实现为一条机器指令

■ 提供硬件“加锁”原子指令TestAndSet

```
boolean TestAndSet(boolean &target)
{
    boolean rv = target;
    target = true;
    return rv;
}
```

一次执行完毕

while(TestAndSet(&lock));

临界区

lock = false;

剩余区

进程P<sub>i</sub>

- 如果lock一开始为false, 则返回值为false, lock变为true
- 如果lock为true, 加锁true, 返回值也为true

无忙等待

使用TS指令实现自旋锁(spinlock)

```
class Lock {
    int value = 0;
}
```

```
Lock::Acquire() {
    while (test-and-set(value))
        ; //spin
}
```

```
Lock::Release() {
    value = 0;
}
```

如果锁被释放, 那么TS指令读取0并将值设置为1

➤ 锁被设置为忙并且需要等待完成

如果锁处于忙状态, 那么TS指令读取1并将值设置为1

➤ 不改变锁的状态并且需要循环

■ 线程在等待的时候消耗CPU时间

```
class Lock {
    int value = 0;
    WaitQueue q;
}

Lock::Acquire() {
    while (test-and-set(value)) {
        add this TCB to wait queue q;
        schedule();
    }
}

Lock::Release() {
    value = 0;
    remove one thread t from q;
    wakeup(t);
}
```

<https://blog.csdn.net/hjw199666>

# 6.3 临界区问题解决方法

## 6.3.5 锁Lock

**锁Lock：**用来对多个线程之间共享的临界区进行保护

POSIX threads(简称Pthreads)是支持多核平台上进行多线程编程的一套API。线程同步最典型的方法就是用Pthreads提供的锁机制(Lock)。

读写锁Read/Write lock实际是一种读写不同的特殊互斥锁。

- ① 它把对共享资源的访问者划分成读者和写者，读者只对共享资源进行读访问，写者则需要对共享资源进行写操作。
- ② 这种锁相对于自旋锁而言，能提高并发性。在多处理器系统中，它允许同时有多个读者来访问共享资源，多个逻辑CPU 访问同一共享资源。但写者是排他性的。

总结：一个读写锁同时只能有一个写者或多个读者(与CPU数相关)，但不能同时既有读者又有写者。

## 6.4 进程同步

### 术语3: 同步(Synchronization)

- **同步**: 多个进程按确定的协作顺序执行（包括中断服务和进程）
- 同步的例子随处可见...

#### 实例1:

```
司机
while (true){
    启动车辆;
    正常运行;
    到站停车; }
```

```
售票员
while (true){
    关门;
    开门;
}
```

#### 实例2:

读者-写者问题  
两组并发进程：读者和写者，共享一个数据区。

- (1)允许多个读者同时执行读操作;
- (2)不允许读者、写者同时操作;
- (3)不允许多个写者同时操作。

## 6.4 进程同步

### 信号量方法实现：生产者 – 消费者互斥与同步控制

```
semaphore fullBuffers = 0; /*仓库中已填满的货架个数*/  
semaphore emptyBuffers = BUFFER_SIZE; /*仓库货架空闲个数*/  
semaphore mutex = 1; /*生产-消费互斥信号*/
```

Producer()

```
{  
    while(True)  
    { /*生产产品item*/  
        emptyBuffers.P();  
        mutex.P();  
        /*item存入仓库buffer*/  
        mutex.V();  
        fullBuffers.V();  
    }  
}
```

互斥控制

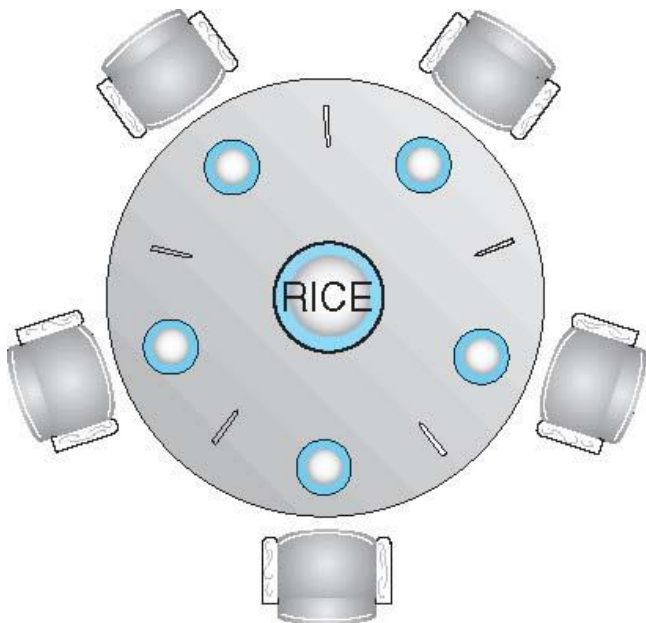
Consumer()

```
{  
    while(True)  
    { fullBuffers.P();  
        mutex.P();  
        /*从仓库buffer中取产品item*/  
        mutex.V();  
        emptyBuffers.V();  
        /*消费产品item*/  
    }  
}
```

同步控制

## 6.4 进程同步

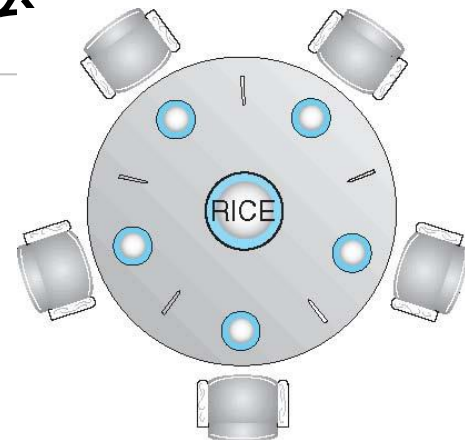
### 经典同步实例：哲学家进餐问题



- 有五位哲学家围坐在一圆桌旁，桌中央有一盘面条，每人面前有一只空盘子，每两人之间放一根筷子
- 每位哲学家相互不进行任何交流，只独自思考，若感到饥饿就拿起二根筷子吃面条
- 为吃到面条，每位哲学家必须获得二根筷子，且每人只能从自己左边或右边拿筷子。

## 6.4 进程同步

### 经典同步实例：哲学家进餐问题 – 直观解法



```
semaphore chopstick[5];  
for(i=0;i<5;i++) chopstick[i]=1;  
  
Philosopher-i( ) /* i=0, 1, 2, 3, 4 共5个函数 */  
{ while (True)  
  {  
    chopstick[i].p(); /* 取左边筷子 */  
    chopstick[(i+1)%5].p(); /* 取右边筷子 */  
    eating(); /* 用餐 */  
    chopstick[i].v(); /* 放下左边筷子 */  
    chopstick[(i+1)%5].v(); /* 放下右边筷子 */  
    thinking(); /* 思考 */  
  }  
}
```

## 6.4 进程同步

### 经典同步实例：哲学家进餐问题 – 直观解法

**存在问题：**所有哲学家都拿到了左手边的筷子，都在等待右手边的筷子。

每个哲学家都出现永远等待，即“死锁”问题。

有若干种办法可避免这种“死锁”。


**例如：**


- 至多允许四个哲学家同时吃；
- 奇数号哲学家先取左手边的筷子，偶数号哲学家先取右手边的筷子；
- 每位哲学家只有能取到手边的两只筷子时才取，否则一只筷子也不取。

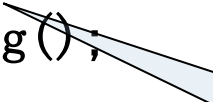


## 6.4 进程同步

### 经典同步实例：哲学家进餐问题 – 无“死锁”解法

```
semaphore chopstick[5], allow-eat=4; 
for(i=0;i<5;i++) chopstick[i]=1;

Philosopher-i( ) /* i=0, 1, 2, 3, 4 共5个函数 */
{ while (True)
    {
         allow-eat.p();
        chopstick[i].p(); /* 取左手边筷子 */
        chopstick[(i+1)%5].p(); /* 取右手边筷子 */
        eating(); /* 用餐 */
        chopstick[i].v(); /* 放下左手边筷子 */
        chopstick[(i+1)%5].v(); /* 放下右手边筷子 */
        thinking(); /* 思考 */
    }
}
```

 allow-eat.v();

## 6.4 进程同步

□ **进程同步实例**：有一个阅览室，共有100个座位。读者进入阅览室时必须在入口处进行登记；离开阅览室时必须进行注销。试用PV操作描述读者进入/离开阅览室的同步与互斥关系。

Reader进程

```
{  
    100个座位是否已满?  
    登记（同一时间办理一个人的登记）  
    座位数减一  
    进入阅览室  
    读书  
    离开阅览室  
    注销（同一时间办理一个人的登记）  
    座位数加一  
}
```

## 6.4 进程同步

### □分析：

在入口和出口处读者应该**互斥**进行登记和注销，  
100个座位，100个**互斥**资源

### □设置信号量

- ▣教室内空座位数量，seat，初值100
- ▣为入口处进行登记设置互斥信号量Sin，初值 1，表示当前可用
- ▣为出口处进行注销设置互斥信号量Sout，初值 1，表示当前可用

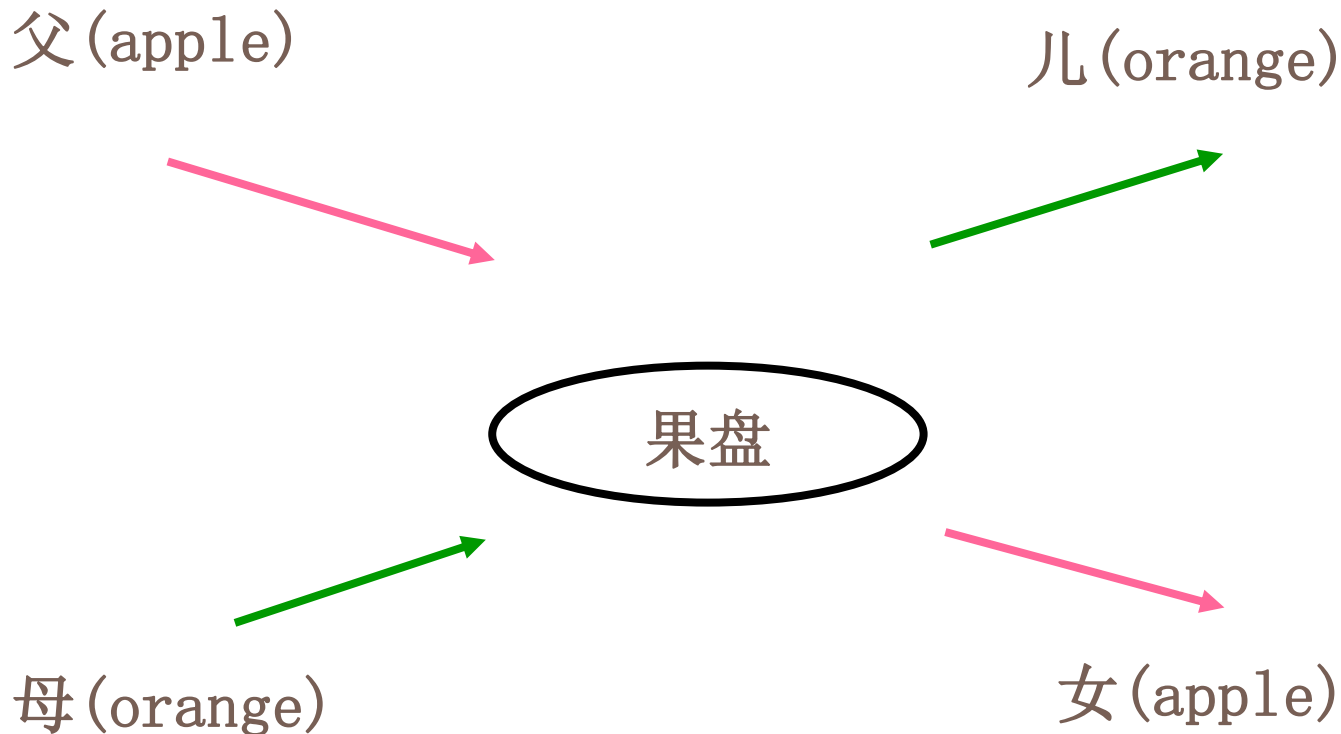
## 6.4 进程同步

```
begin
  Sin, Sout, seat: semaphore; //计数信号量
  seat := 100; Sin := 1; Sout := 1;
cobegin
  process Reader-i ( i = 1,2,...,n );
  begin
    P(seat); //座位是否有空位? 为空等待, 不空则减1
    P(Sin); //互斥操作
    登记;
    V(Sin);
    进入阅览室; 读书; 离开阅览室;
    P(Sout); //互斥操作
    注销;
    V(Sout);
    V (seat) ; //座位数加1
  end
coend;
end;
```

## 6.4 进程同步

### 进程同步实例：

- 1、吃水果, 一次只能有一个人使用果盘, 盘子只能放1个水果
- 2、吃水果, 一次只能有一个人使用果盘, 盘子只能放2个水果



## 6.4 进程同步

1、吃水果,一次只能有一个人使用果盘, 盘子只能放1个水果

Plate:=1, 表示盘子空否

apple:=0, orange:=0表示有否水果

父:

```
P(plate);  
Put apple;  
V(apple);
```

女:

```
P(apple);  
Take apple;  
V(plate);
```

母:

```
P(plate);  
Put orange;  
V(orange);
```

儿:

```
P(orange);  
Take orange;  
V(plate);
```

## 6.4 进程同步

**信号量实现同步与互斥，P/V操作出现的特点总结：**

- (1) 定义一个信号量sem;
- (2) 用sem来实现临界区互斥进入问题时，  
sem.p()和sem.v()应该出现在同一进程代码中的相关临界区的“进入区”和“退出区”；
- (3) 用sem来实现进程间同步问题时，  
sem.p()和sem.v()一般应出现在协作进程组中的不同进程。

## 6.4 进程同步

信号量的P/V操作常见的表示法或函数名称：

- (1) 一般逻辑上(伪代码)的表达:  $P(s) / V(s)$
- (2) 常见的函数名1: `wait()` / `signal()`
- (3) 常见的函数名2: `down()` / `up()`
- (4) 常见的函数名3: `TakeSem()` / `GiveSem()`



# 进程同步总结

- 并发  $\Rightarrow$  多个进程同时存在  $\Rightarrow$  相互影响
- 非原子操作共享变量  $\Rightarrow$  出现语义错误  $\Rightarrow$  竞争条件
- 竞争条件  $\Rightarrow$  临界区  $\Rightarrow$  互斥  $\Rightarrow$  临界区进入方法
- 复杂的面包店算法  $\Rightarrow$  强硬的关中断  $\Rightarrow$  硬件支持的 **TestAndSet**
- 都不适合用户实现  $\Rightarrow$  封装成锁
- 一般的锁会“忙等”  $\Rightarrow$  引入睡眠  $\Rightarrow$  将锁一般化为信号量
- 信号量也容易出错（死锁）  $\Rightarrow$  管程将复杂性交给编译器、死锁检测
- 所有一切都是为了使用户更容易、使系统更好用(不出错)