



第4章 进程同步



第4章 进程同步

- 并发程序在进行协作时能够实现：
 - 共享逻辑地址空间：包括代码段+数据段
 - 线程的实现
 - 共享资源
 - 通过文件、消息等实现
 - 但是并发访问可能导致资源状态的不一致
- Q：如何确保共享同一逻辑地址空间的协作进程有序执行，不产生资源状态的不一致？
 - 即：在并发程序推进速度不一致、不可预测的情形下，确保多进程推进的相对次序是按照预期推进的，从而确保并发程序的正确性？



第4章 进程同步

4.1 同步与互斥的概念



4.1 同步与互斥的概念

- 在多道程序系统中，进程之间的相互制约关系体现在如下两个方面：
 - **直接制约关系**：合作进程之间产生的制约关系
 - 或称协作关系，需要同步（synchronization）
 - **间接制约关系**：共享资源产生的制约关系。
 - 或称竞争关系，需要互斥（mutual exclusion）

与时间有关的错误例子1：结果不唯一

- 现有订票系统，x为存储某班次飞机剩余票数的存储区域

■ A:

```
1. R1=x;  
2. if (R1>=1) {  
3.     R1--;  
4.     x=R1;  
5.     {出票}  
6. }
```

■ B:

```
1. R2=x;  
2. if (R2>=1){  
3.     R2--;  
4.     x=R2;  
5.     {出票}  
6. }
```

与时间有关的错误例子1：结果不唯一

进程A:

1. R1=x;
2. if (R1>=1) {
3. R1--;
4. x=R1;
5. {出票}
6. }

如果先执行A再执行B，则x值最终减少2，A、B输出的x不同。

进程B:

1. R2=x;
2. if (R2>=1){
3. R2--;
4. x=R2;
5. {出票}
6. }

与时间有关的错误例子1：结果不唯一

进程A:

1. $R1 = x;$
2. $\text{if } (R1 \geq 1) \{$

3. $R1--;$
4. $x = R1;$
5. $\{\text{出票}\}$
6. $\}$

进程B:

1. $R2 = x;$
2. $\text{if } (R2 \geq 1) \{$
3. $R2--;$
4. $x = R2;$
5. $\{\text{出票}\}$
6. $\}$

若按顺序 $R1 = x; R2 = x; R1--; x = R1; R2 = R2--;$
 $x = R2;$ 则 x 值最终减少1, A、B输出的 x 是相同的。

与时间有关的错误例子1：结果不唯一

■ A:

1. $R1 = x;$
2. $\text{if } (R1 \geq 1) \{$
3. $R1--;$
4. $x = R1;$
5. {出票}
6. }

■ B:

1. $R2 = x;$
2. $\text{if } (R2 \geq 1) \{$
3. $R2--;$
4. $x = R2;$
5. {出票}
6. }

- 这种错误称为 “**与时间有关的错误**”，有些地方称为 (race condition)
- 产生原因
 - 没有互斥使用共享变量，或者说没有在某进程进入时禁止另一个进程的进入。
 - 我们称以上例子中的错误为**结果不唯一**的错误

与时间有关的错误例子2：永远等待

- alloc和free为申请和归还资源的函数，进程在申请和释放时候调用它们。
- x为当前可用的总主存量，B为申请数量：

```
void alloc(int B)
{
    while(B>x)
        {将申请进程设置进入等待队列，等待主存资源}
    x=x-B;
    {修改主存分配表}
    {申请进程获得主存资源}
}
```

```
void free(int B)
{
    x=x+B;
    {修改主存分配表}
    {唤醒等待主存资源的进程}
}
```

与时间有关的错误例子2：永远等待

进程A:

```
void alloc(int B)
{
    while(B>x)
    {将申请进程设置进入等待
      队列，等待主存资源}
    x=x-B;
    {修改主存分配表}
    {申请进程获得主存资源}
}
```

进程B:

```
void free(int B)
{
    x=x+B;
    {修改主存分配表}
    {唤醒等待主存资源的进程}
}
```

与时间有关的错误例子2：永远等待

进程A:

```
void alloc(int B)
{
    while(B>x)
```

```
    {将申请进程设置进入等待队列，等待主存资源}
    x=x-B;
    {修改主存分配表}
    {申请进程获得主存资源}
}
```

进程B:

```
void free(int B)
{
    x=x+B;
    {修改主存分配表}
    {唤醒等待主存资源的进程}
}
```

如果某个时刻 $B > x$ ，进程A调用alloc，在执行while之后，{申请...等待...}之前，进程B进入处理器，调用了free，则由于A还未来得及进入排队，则不会被唤醒，此后A进程进入等待队列，但不再有人唤醒A。



4.1.1 临界资源与临界区

- **临界资源：**
 - 一段时间内仅允许一个进程使用的资源称为临界资源。(Critical Resource)
 - 如：打印机、共享变量。
- **临界区：**
 - 进程中访问临界资源的那段代码称为临界区，又称临界段。
- **同类临界区：**
 - 所有与同一临界资源相关联的临界区。



临界资源访问过程

一般包含四个部分：

1.进入区

- 检查临界资源访问状态
- 若可访问，设置临界资源被访问状态

2.临界区

- 访问临界资源代码

3.退出区

- 清除临界资源被访问状态

4.剩余区

- 其他部分



访问临界资源应遵循的原则

1. 空闲让进

- 若无进程处于临界区时，应允许一个进程进入临界区，且一次至多允许一个进程进入。

2. 忙则等待

- 当已有进程进入临界区，其他试图进入的进程应该等待。

3. 有限等待

- 应保证要求进入临界区的进程在有限时间内进入临界区。也蕴含着有限使用的意思！

4. 让权等待

- 当进程不能进入自己的临界区时，应释放处理机，不至于造成饥饿甚至死锁。



访问临界资源应遵循的原则cont.

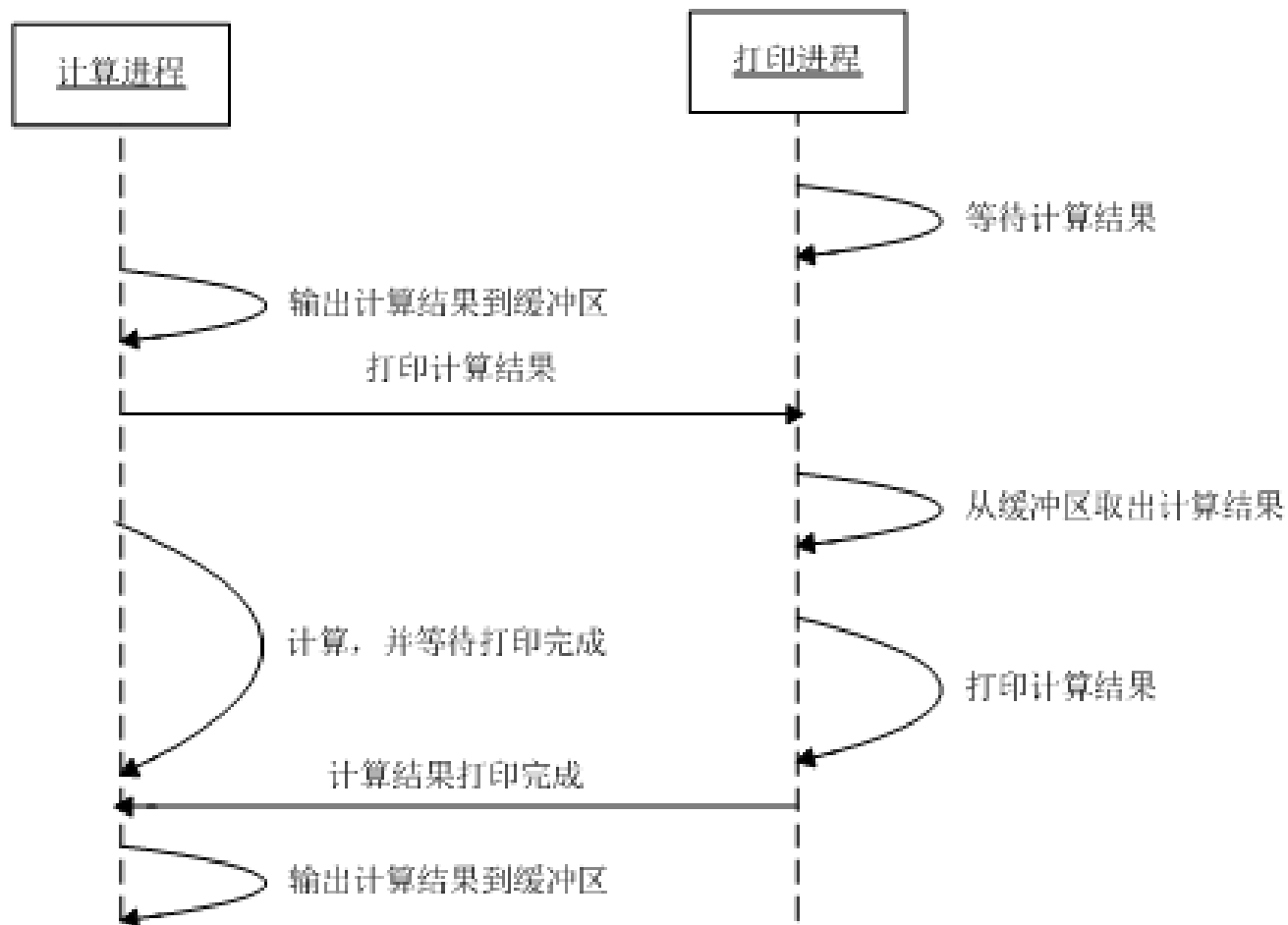
- 互斥原则, mutual exclusion
 - 如果进程 P_i 在其临界区内执行, 那么其他进程都不能在其临界区执行
- 推进原则, Progress
 - 如果没有进程在其临界区内执行, 并且有进程需要进入临界区, 那么只有那些不在剩余区执行的进程可以参与选择, 以便确定谁能下次进入临界区, 而且这种选择不能无限推迟。
- 有限等待, bounded waiting
 - 从一个进程作出进入临界区的请求, 直到这个请求允许为止, 其他进程允许进入临界区的次数具有上限。

4.1.2 同步与互斥

- **同步** (synchronization) :
 - 多个相互合作的进程在一些关键点上可能需要互相等待或互相交换信息，这种相互制约关系称为进程同步。
 - **不确定中蕴含了确定性！**
- 同步例子：计算进程与打印进程共享一个单缓冲区。



■ 计算进程与打印进程的同步





互斥

- **互斥**(mutual exclusion): 相互制约关系
 - 当一个进程正在使用某资源时, 其他希望使用该资源的进程必须等待
 - 当该进程用完资源并释放后, 才允许其他进程去访问此资源



第4章 进程同步

4.2 互斥的实现方法



4.2 互斥的实现方法

- 互斥的实现
 - 软件方法
 - 硬件方法
 - 更高级的抽象方法
- 这里我们讨论的并发进程所处环境是
 - 单处理器计算机系统（主）
 - 共享主存的多处理计算机系统（辅）



4.2.1 互斥的软件实现方法

- 假设：有两个进程P0和P1
 - P0、P1为并发进程
 - P0、P1互斥地共享某个临界资源
 - P0、P1不断工作，每次使用该资源一个有限的时间间隔。



算法1的思想

- 朴素想法：设置一个开关变量turn
 - turn为公用整型变量，用来指示允许进入临界区的进程标识。
 - 对于P0，若turn为0，则允许进程P0进入临界区;否则循环检查该变量，直到turn变为本进程标识0;
 - 在退出区，修改允许进入进程的标识为1。进程P1的算法与此类似。

算法1的描述

```
int turn=0;
```

```
P0(){
```

```
do{
```

```
    while (turn!=0);
```

```
    进程P0的临界区代码CS0;
```

```
    turn = 1;
```

```
    进程P0的其他代码;
```

```
    }while(true)
```

```
}
```

```
P1(){
```

```
do{
```

```
    while (turn!=1);
```

```
    进程P1的临界区代码CS1;
```

```
    turn = 0;
```

```
    进程P1的其他代码;
```

```
    }while(true)
```

```
}
```

■ 算法1存在的问题

- 此算法可以保证互斥访问临界资源，但两个进程在调度中必须以交替次序进入临界区。
- P0执行完一次循环之后，P1若不被调度执行，P0也无法继续运行；
- 此算法不能保证实现空闲让进准则。



算法2的思想

- 算法1的程序问题在于
 - 使用标识后设置了标识，导致标识与初始不一致，从而造成两程序的依赖关系
 - 注意：这种依赖关系在后续某些问题中是有价值的！。
- 改进思想：消除依赖关系
 - 设置彼此独立的标志数组flag[i]表示进程i是否在临界区中执行，初值均为假。
 - 在每个进程访问临界资源之前，先检查另一个进程是否在临界区中，若不在，则修改本进程的临界区标志为真，并进入临界区，
 - 在退出临界区时，修改本进程临界区标志为假。

算法2的描述

```
enum bool {false, true};  
bool flag[2]={false, false};
```

```
P0: {  
    do {  
        while (flag[1]);  
        flag[0] = true;  
        进程P0的临界区代码CS0;  
        flag[0] = false ;  
        进程P0的其他代码;  
    }  
    while(true)  
}
```

```
P1: {  
    do {  
        while (flag[0]);  
        flag[1] = true;  
        进程P1的临界区代码CS1;  
        flag[1] = false ;  
        进程P1的其他代码;  
    }  
    while(true)  
}
```

■ 算法2的问题：

- 此算法解决了空闲让进的问题，但有可能两个进程同时进入临界区。
- 当两个进程都未进入临界区时，它们各自的访问标志值都为false，若此时刚好两个进程同时都想进入临界区，并且都发现对方标志值为false，两个进程可以同时进入了各自的临界区，这就违背了临界区的访问原则**忙则等待**。



算法3的思想

- 算法2的程序问题在于
 - 检测点过早，标志flag[0]与flag[1]之间没有形成互反，导致进入条件均可满足
- 算法3的改进：
 - 延迟检测点，本算法仍然设置标志数组flag[i]，标志用来表示**进程i是否计划进入临界区，即一种意愿**。
 - 在每个进程访问临界资源之前，先将自己的标志设置为true，表示进程希望进入临界区
 - 然后再检查另一个进程的标志，若另一个进程的标志为true，则进程等待，否则进入临界区。

算法3的描述

```
enum bool {false, true};  
bool flag [2] = {false, false};
```

```
P0: {  
    do {  
        flag[0] = true;  
        while (flag[1]) ;  
        进程P0的临界区代码CS0;  
        flag[0] = false;  
        进程P0的其他代码;  
    }  
    while(true)  
}
```

```
P1: {  
    do {  
        flag[1] = true;  
        while (flag[0]) ;  
        进程P1的临界区代码CS1;  
        flag[1] = false;  
        进程P1的其他代码;  
    }  
    while(true)  
}
```

■ 算法3的问题:

- 该算法可以有效地防止两个进程同时进入临界区，但存在两个进程都进不了临界区的问题。
- 当两个进程同时想进入临界区时，它们分别将自己的标志位设置为true，并且同时去检查对方的状态，发现对方也要进入临界区，于是双方互相谦让，结果谁也进不了临界区。**违背了让权等待原则**



算法4的思想： Dekker算法

- 算法3的问题在于
 - 预期进入临界区标识Flag，无法识别当前已经进入的其他进程情况
- 算法4的改进：
 - 荷兰数学家T. Dekker提出了一种Dekker算法，解决此问题。
 - 本算法的基本思想是算法3和算法1的结合。是一个正确的算法。
 - 标志数组flag[]表示进程是否希望进入临界区或是否正在临界区中执行。
 - 还设置了一个turn变量，用于指示允许进入临界区的进程标识。

算法4的描述

```
enum bool {false, true};  
bool flag [2] = {false, false};  
int turn = 0 或者 1;
```

P0:

```
1.  do{  
2.    flag[0] = true;  
3.    while (flag[1]){  
4.      if (turn!=0){  
5.        flag[0] = false;  
6.        while (turn!=0);  
7.        flag[0] = true;  
8.      }  
9.    }  
10.   进程P0的临界区代码CS0;  
11.   turn = 1;  
12.   flag[0] = false;  
13.   进程P0的其他代码;  
14. } while (true)
```

P1:

```
1.  do{  
2.    flag[1] = true;  
3.    while (flag[0]){  
4.      if (turn!=1){  
5.        flag[1] = false;  
6.        while (turn!=1);  
7.        flag[1] = true;  
8.      }  
9.    }  
10.   进程P1的临界区代码CS1;  
11.   turn = 0;  
12.   flag[1] = false;  
13.   进程P1的其他代码;  
14. } while (true)
```



讨论

1. 会不会出现无法推进呢，例如，如果等待在p0第一层while循环里如何？如果等待在p0第二层循环里如何？
2. 为什么需要if(turn! =0)呢？如果没有会怎样？

讨论问题2

```
enum bool {false, true};  
bool flag [2] = {false, false};  
int turn = 0 ;
```

P0:

```
1.  do{  
2.    flag[0] = true;  
3.    while (flag[1]){  
4.      {  
5.        flag[0] = false;  
6.        while (turn!=0);  
7.        flag[0] = true;  
8.      }  
9.    }  
10.   进程P0的临界区代码CS0;  
11.   turn = 1;  
12.   flag[0] = false;  
13.   进程P0的其他代码;  
14. } while (true)
```

P1:

```
1.  do{  
2.    flag[1] = true;  
3.    while (flag[0]){  
4.      {  
5.        flag[1] = false;  
6.        while (turn!=1);  
7.        flag[1] = true;  
8.      }  
9.    }  
10.   进程P1的临界区代码CS1;  
11.   turn = 0;  
12.   flag[1] = false;  
13.   进程P1的其他代码;  
14. } while (true)
```

这里有问题吗? --1

```
enum bool {false, true};  
bool flag [2] = {false, false};  
int turn = 0;
```

P0:

```
1.  do{  
2.    flag[0] = true;  
3.    while (flag[1]){  
4.      if (turn!=0){  
5.        flag[0] = false;  
6.        while (flag[1]);  
7.        flag[0] = true;  
8.      }  
9.    }  
10.  进程P0的临界区代码CS0;  
11.  turn = 1;  
12.  flag[0] = false;  
13.  进程P0的其他代码;  
14. } while (true)
```

P1:

```
1.  do{  
2.    flag[1] = true;  
3.    while (flag[0]){  
4.      if (turn!=1){  
5.        flag[1] = false;  
6.        while (flag[0]);  
7.        flag[1] = true;  
8.      }  
9.    }  
10.  进程P1的临界区代码CS1;  
11.  turn = 0;  
12.  flag[1] = false;  
13.  进程P1的其他代码;  
14. } while (true)
```


这里有问题吗? --2(小作业)

```
enum bool {false, true};  
bool flag [2] = {false, false};  
int turn = 0;
```

P0:

```
1.  do{  
2.    flag[0] = true;  
3.    while (turn != 0){  
4.      {  
5.        flag[0] = false;  
6.        while (flag[1]);  
7.        flag[0] = true;  
8.      }  
9.    }  
10.  进程P0的临界区代码CS0;  
11.  turn = 1;  
12.  flag[0] = false;  
13.  进程P0的其他代码;  
14. } while (true)
```

P1:

```
1.  do{  
2.    flag[1] = true;  
3.    while (turn != 1){  
4.      {  
5.        flag[1] = false;  
6.        while (flag[0]);  
7.        flag[1] = true;  
8.      }  
9.    }  
10.  进程P1的临界区代码CS1;  
11.  turn = 0;  
12.  flag[1] = false;  
13.  进程P1的其他代码;  
14. } while (true)
```

算法5的描述:Peterson算法

- 1981年, G. L. Peterson 给出了一种更为简单的实现算法

```
enum boolean {false, true};  
boolean flag[2] = {false, false};  
int turn;
```

```
P0:  
{  
    do  
    {  
        flag[0] = true;  
        turn = 1;  
        while (flag[1] &&turn == 1)  
            ;  
        进程P0的临界区代码CS0;  
        flag[0] = false;  
        进程P0的其他代码;  
    } while (true)  
}
```

```
P1:  
{  
    do  
    {  
        flag[1] = true;  
        turn = 0;  
        while (flag[0] &&turn == 0)  
            ;  
        进程P1的临界区代码CS1;  
        flag[1] = false;  
        进程P1的其他代码;  
    } while (true)  
}
```



4.2.2 硬件方法

- 软件算法实现互斥是较为困难的
 - 最大的问题是对临界区的**检测与设置**动作很难作为一个**整体**来实现
 - 软件一条语句可能会被拆分
- 用硬件方法实现互斥的主要思想是
 - 在单处理器情况下，并发进程是交替执行的，因此只需要保证检查操作与修改操作不被中断即可，因此可以对关键部分进行硬件实现
 - 关中断方法
 - 原子指令方法



1、关中断方法

- 当进程执行临界区代码时，要防止其他进程进入其临界区访问，最简单的方法是关中断。Why?
- 关中断
 - 能保证当前运行进程将临界区代码顺利执行完，从而保证了互斥的正确实现，然后再允许中断。
 - 现代计算机系统都提供了关中断指令。
 - 中断响应将延迟到中断启用之后



用关中断方法实现互斥

┆
关中断;
临界区;
开中断;
┆



关中断方法的不足

- 效率问题

- 如果临界区执行工作很长，则无法预测中断响应的时间
- 系统将处于暂停状态，无法响应事件
- 限制了处理机交替执行程序的能力，执行的效率将会明显降低；

- 适用范围问题

- 关中断不一定适用于多处理器计算机系统
- 一个处理器关掉中断，并不意味着其他处理器也关闭中断，不能防止进程进入其他处理器执行临界代码。

- 安全性问题：

- 将关中断的权力交给用户进程则很不明智，若一个进程关中断之后不再开中断，则系统可能会因此终止甚至崩溃。



2、原子指令方法

- 许多计算机中提供了专门的硬件指令，实现对字节内容的检查和修改或交换两个字节内容的功能。
- 使用这样的硬件指令就可以解决临界区互斥的问题。
 - 测试设置方法
 - 对换指令方法



TS (Test-and-Set) 指令

- TS指令的功能可描述如下：

```
boolean TS(boolean * lock)
{
    if (false == *lock){
        *lock = true;
        return false;
    }
    else
        return true;
}
```

```
boolean TS(boolean *lock)
{
    boolean old;
    old=*lock;
    *lock=true;
    return old;
}
```




用TS指令实现进程互斥

- 为每个临界资源设置一个共享布尔变量
lock表示资源的两种状态： true表示正
被占用， false表示空闲。算法如下：

```
    |  
    while TS(&lock);  
    进程的临界区代码CS;  
    lock=false ;  
    进程的其他代码;  
    |
```



Swap指令（或Exchange指令）

- Swap指令的功能可描述如下：

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

用Swap指令实现进程互斥

- 为每个临界资源设置一个共享布尔变量lock表示临界资源状态;再设置一个局部布尔变量key用于与lock交换信息。算法如下:

```
┆  
key=true;  
while(false !=key) Swap(&lock, &key);  
进程的临界区代码CS;  
lock=false ;// or Swap(&lock, &key);  
进程的其他代码;  
┆
```



4.2.3 互斥锁机制

- **互斥锁**是一个代表资源状态的变量，通常用0表示资源可用（开锁），用1表示资源已被占用（关锁）。
- 在使用临界资源前需先考察锁变量的值
 - 如果值为0则将锁设置为1（关锁）
 - 如果值为1则回到第一步重新考察锁变量的值
 - 当进程使用完资源后，应将锁设置为0（开锁）。



锁原语

- 上锁

```
lock (w)
```

```
{
```

```
    while (w==1) ;//等待可锁状态
```

```
    w = 1;//加锁
```

```
}
```

- 开锁

```
unlock (w)
```

```
{
```

```
    w = 0;//解锁
```

```
}
```



用互斥锁机制实现互斥

进程 P_1

⋮

lock(w);

临界区;

unlock(w);

⋮

进程 P_2

⋮

lock(w);

临界区;

unlock(w);

⋮



自旋锁

■ 互斥锁

- 往往采用前面所介绍的硬件机制来实现
- 对于单处理器，为了提高效率，忙等会改为休眠，释放时要唤醒
- 锁的申请和释放，带来上下文切换，但是，在多处理器情况下，会造成非常大的浪费
- 自旋锁是专为防止多处理器并发而引入的一种锁。

■ 自旋锁工作机理

- 自旋锁最多只能被一个内核任务持有，如果一个内核任务试图请求一个已被争用(已经被持有)的自旋锁，那么这个任务就会一直进行忙循环——旋转——等待锁重新可用。要是锁未被争用，请求它的内核任务便能立刻得到它并且继续进行
- 多核处理器中的busy waiting：一个线程在一个处理器中自旋，另一个线程可以在另一个处理器中运行，表现形式则是用户进程可以继续工作
- 对于多核线程，预计线程等待锁的时间很短，短到持有自旋锁的时间小于两次上下文切换的时间，则使用自旋锁是划算的。



课堂讨论

1. 有了可以证明的软件方法解决同步问题，为什么还要硬件方法？
2. 为什么自旋锁对于单处理器系统不适合，而经常在多处理器系统中使用？



小结

- 关中断：
 - 对于单核处理器、非抢占式内核可行
 - 对于多核处理器、抢占式内核不佳
- 原子指令方法
 - 复杂，需要硬件支持
- 互斥锁
 - 浪费CPU：两次上下文切换
- 优先级翻转
 - 若进程间优先级： $L < M < H$ ，H依赖于L，L处于临界区，M抢占L，则M影响了H

小作业-1:

小作业

1. V9, 6.3

2. 以及dekker算法讨论

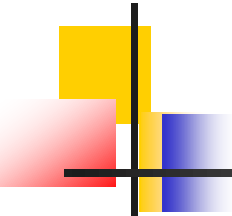
```
enum bool {false, true};  
bool flag [2] = {false, false};  
int turn = 0;
```

P0:

```
1.  do{  
2.    flag[0] = true;  
3.    while (turn != 0){  
4.      {  
5.        flag[0] = false;  
6.        while (flag[1]);  
7.        flag[0] = true;  
8.      }  
9.    }  
10.   进程P0的临界区代码CS0;  
11.   turn = 1;  
12.   flag[0] = false;  
13.   进程P0的其他代码;  
14. } while (true)
```

P1:

```
1.  do{  
2.    flag[1] = true;  
3.    while (turn != 1){  
4.      {  
5.        flag[1] = false;  
6.        while (flag[0]);  
7.        flag[1] = true;  
8.      }  
9.    }  
10.   进程P1的临界区代码CS1;  
11.   turn = 0;  
12.   flag[1] = false;  
13.   进程P1的其他代码;  
14. } while (true)
```



第4章 进程同步

4.3 信号量



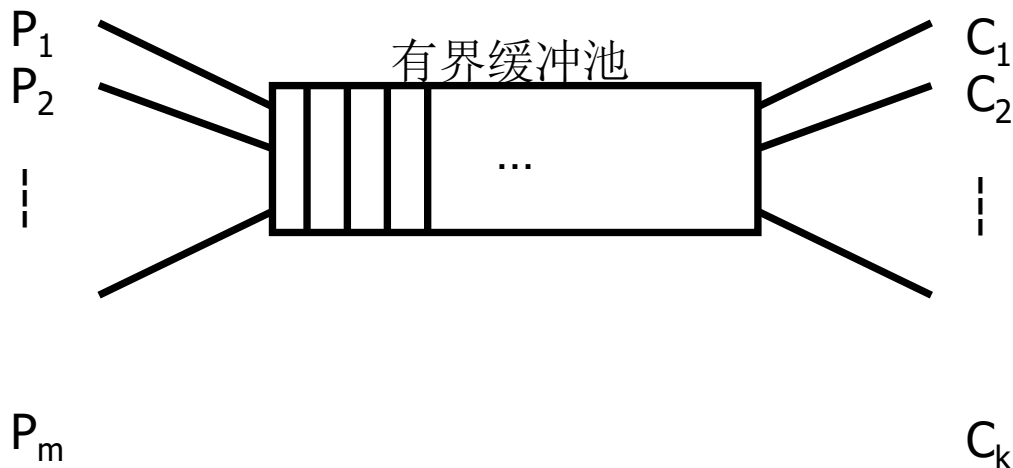
4.3 信号量：同步问题的解决

- 一个经典的同步问题：生产者与消费者
 - 著名的生产者--消费者问题是计算机操作系统中并发进程内在关系的一种抽象，是典型的进程同步问题。
 - 在操作系统中，生产者进程可以是计算进程、发送进程;而消费者进程可以是打印进程、接收进程等等。
 - 解决好生产者--消费者问题就解决好了一类并发进程的同步问题。

生产者—消费者问题

问题描述（有界缓冲区问题）

- 有 n 个生产者和 m 个消费者，连接在一个有 k 个单位缓冲区的有界缓冲上。
- 其中，生产者进程 P_i 和消费者进程 C_j 都是并发进程。
- 只要缓冲区未满，生产者 P_i 生产的产品就可投入缓冲区；
- 只要缓冲区不空，消费者进程 C_j 就可从缓冲区取走并消耗产品。



生产者-消费者问题一种算法描述

常量k为缓冲区大小;

```
struct item nextp, nextc, buffer[k];
```

```
int in=0, out=0, counter = 0;
```

```
Producer_i:
```

```
while (1){
```

```
    produce an item in nextp;
```

```
    /* 生产一个产品*/
```

```
    if (counter==k) sleep( );
```

```
    /* 缓冲满时, 生产者睡眠*/
```

```
    buffer[in]=nextp;
```

```
    /* 将一个产品放入缓冲区*/
```

```
    in=(in+1) % k;
```

```
    /* 指针推进*/
```

```
    counter++;
```

```
    /* 缓冲内产品数加1*/
```

```
    if (counter==1)
```

```
        wakeup(consumer);
```

```
    /* 之前缓冲是空的, 加进一件产品后唤醒消费者*/
```

```
}
```



```
Consumer_j:
```

```
while (1){
```

```
    if (counter==0)
```

```
        sleep( ); /* 缓冲区空消费者睡眠*/
```

```
    nextc=buffer[out];
```

```
    /*取一个产品到nextc*/
```

```
    out=(out+1)%k;
```

```
    /* 指针推进*/
```

```
    counter--;
```

```
    /* 取走一个产品, 计数减1*/
```

```
    if (counter==k-1)
```

```
        wakeup(producer);
```

```
    /* 此时缓冲取走了一件产品, 唤醒生产者*/
```

```
    consume the item in nextc;
```

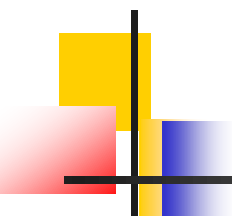
```
    /* 消耗产品*/
```

```
}
```



生产者-消费者问题一种算法描述

- 生产者和消费者进程对counter的交替执行会使其结果不唯一
- 生产者和消费者进程的交替执行会导致进程永远等待

- 
- 前节种种软件方法解决临界区调度问题的缺点:
 - 对不能进入临界区的进程，采用忙式等待测试法，浪费CPU时间。
 - 将测试能否进入临界区的责任推给各个竞争的进程会削弱系统的可靠性，加重了用户编程负担。
 - 1965年，E. W. Dijkstra提出了新的同步工具:信号量和P、V操作



4.3.1 信号量的定义

- 信号量(semaphore)由两个成员 (s, q) 组成
 - 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。
- 除信号量的初值外，信号量的值仅能由P操作（又称为wait操作）和V操作（又称为signal操作）改变。



P操作

- 设 $S = (s, q)$ 为一个信号量, $P(S)$ 执行时主要完成下述动作:
 - $s = s - 1;$
 - If $(s < 0)$ {
 - 设置进程状态为等待;
 - 将进程放入信号量等待队列 q ;
 - 转调度程序;}



V操作

- V(S)执行时主要完成下述动作：
 - $s = s + 1;$
 - $\text{If}(s \leq 0)\{$
 - 将信号量等待队列q中的第一个进程移出;
 - 设置其状态为就绪状态并插入就绪队列;
 - 然后再返回原进程继续执行;
 - }



几个重要含义

- 信号量中的整型变量 s 表示系统中某类资源的数目。
- 当 $s > 0$ 时,
 - 该值等于在封锁进程之前对信号量 S 可施行的 P 操作数,
 - 等于 S 所代表的实际还可以使用的资源数
- 当 $s < 0$ 时,
 - 其绝对值等于登记排列在该信号量 S 队列之中等待的进程个数,
 - 亦即恰好等于对信号量 s 实施 P 操作而被封锁起来并进入信号量 s 队列的进程数



注意

- 通常，P操作意味着请求一个资源，V操作意味着释放一个资源。
 - 在一定条件下，P操作代表使进程阻塞的操作，而V操作代表唤醒阻塞进程的操作。
- P、V操作的原子性要求
 - 即，一个进程在信号量上操作时，不会有别的进程同时修改该信号量。
 - 对于单处理器，可简单地在封锁中断的情况下执行
 - 对于多处理器环境，需要封锁所有处理器的中断，困难且影响性能，往往用swap()或自旋锁等方式加锁



信号量的另一种描述

```
typedef struct{
    int value;
    struct process *list;
} semaphore;

wait(semaphore *S) {
    S->value--;
    if (S->value < 0) {
        add this process to S->list;
        block();
    }
}

signal(semaphore *S) {
    S->value++;
    if (S->value <= 0) {
        remove a process P from S->list;
        wakeup(P);
    }
}
```



4.3.2 利用信号量实现互斥

- 设 $S=(s,q)$ 为两个进程P1、P2实现互斥的信号量
 - s 的初值应为1，即可用资源数目为1。
- 只需把临界区置于P (S) 和V (S) 之间，即可实现两进程的互斥。



互斥访问临界区的描述

进程P1:

|

P(S);

进程P1的临界区;

V(S);

|

进程P2:

|

P(S);

进程P2的临界区;

V(S);

|



一个老问题.....

- 现有订票系统, x 为存储某班次飞机剩余票数的存储区域
- A:
 $R1 = x;$
 if ($R1 \geq 1$) {
 $R1--;$
 $x = R1;$
 {出票}
 }
- B:
 $R2 = x;$
 if ($R2 \geq 1$) {
 $R2--;$
 $x = R2;$
 {出票}
 }

一个信号量实现互斥的例子

改进订票系统:

A:

P(S);

R1=x;

if (R1 >= 1) {

 R1--;

 x=R1;

V(S);

 {出票}

}

else{

V(S)

 {提示无票}

}

B:

P(S);

R2=x;

if (R2 >= 1){

 R2--;

 x=R2;

V(S)

 {出票}

}

else{

V(S)

 {提示无票}

}

互斥信号量的取值范围

- 若2个进程共享一个临界资源，信号量的取值范围是：

若没有进程使用临界资源 1

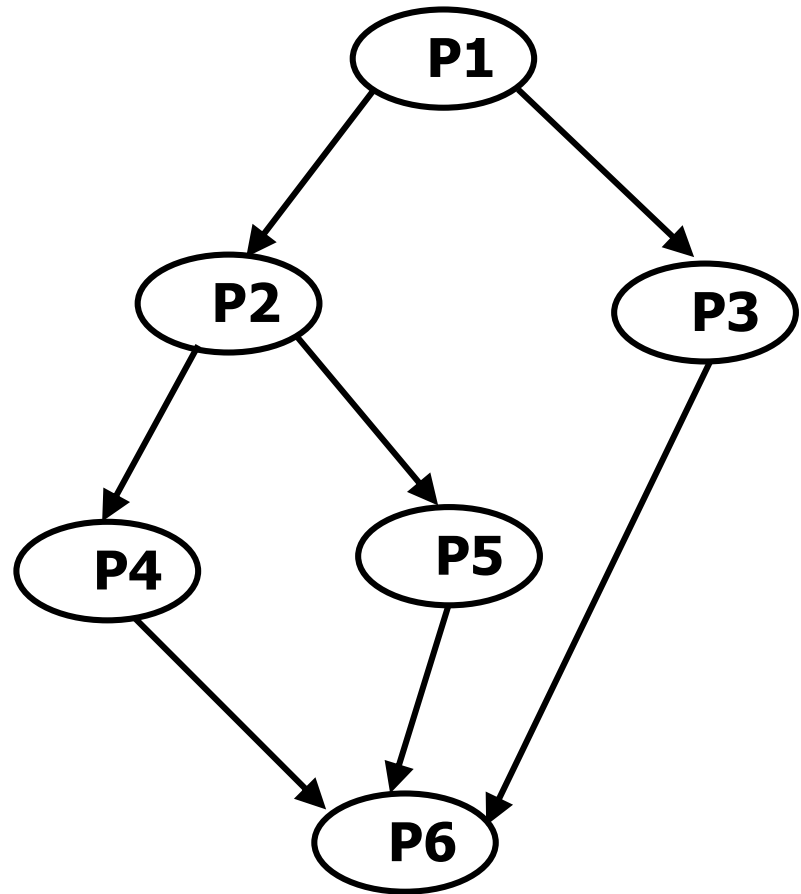
若只有1个进程使用临界资源 0

若1个进程使用临界资源，另1个进程等待使用临界资源 -1

Q：若N个进程共享一个临界资源，其取值范围如何？

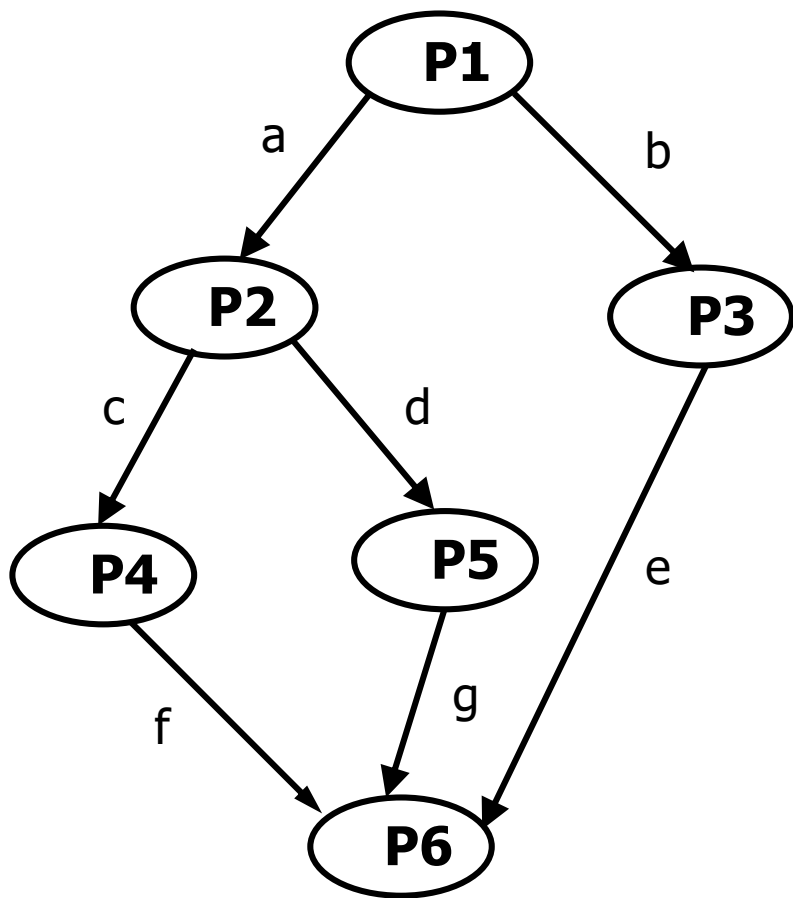
4.3.3 利用信号量实现前趋关系

- 例如：P1、P2、P3、P4、P5、P6为一组合作进程，其前驱图如下所示，试用P、V操作完成这六个进程的同步。



解法1

- 设七个同步信号量a、b、c、d、e、f、g分别表示进程之间的前驱关系，如图所示，其初值均为0。这六个进程的同步描述如下：



解法1 (1)

P1()

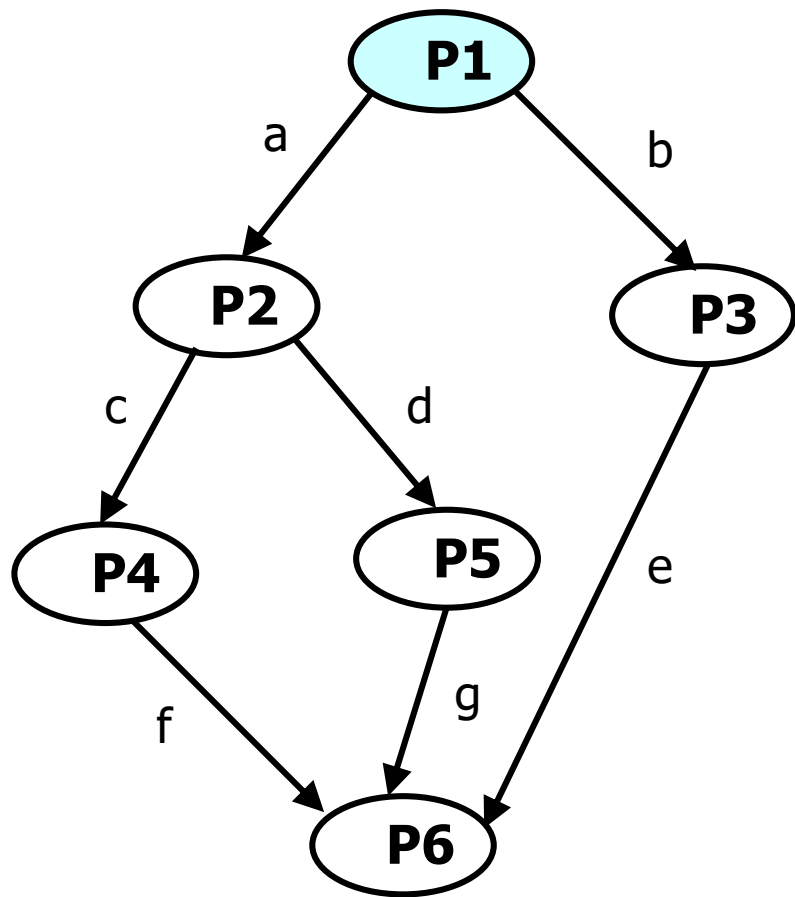
{

执行P1的代码;

v(a);

v(b);

}



解法1 (2)

P2 ()

{

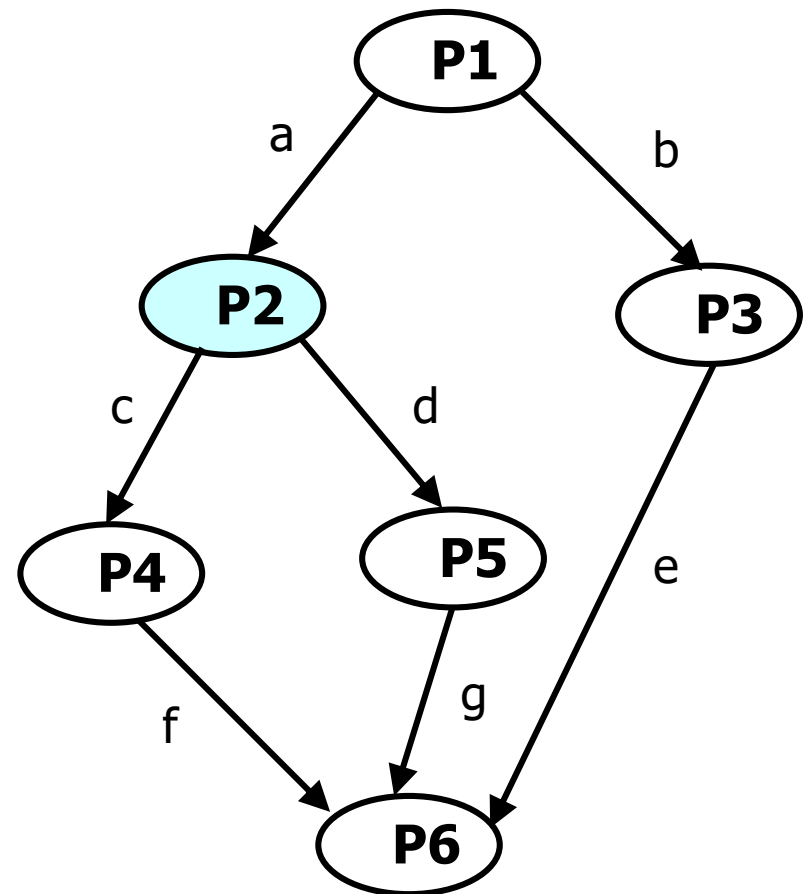
p(a);

执行P2的代码;

v(c);

v(d);

}



解法1 (3)

P3 ()

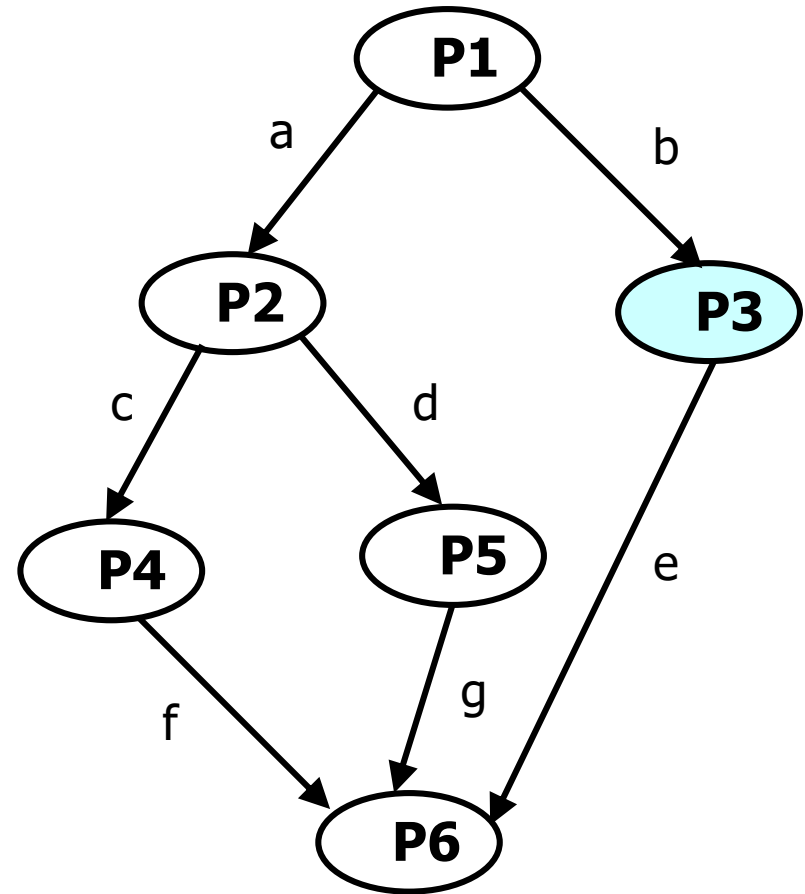
{

p(b);

执行P3的代码;

v(e);

}



解法1 (4)

P4 ()

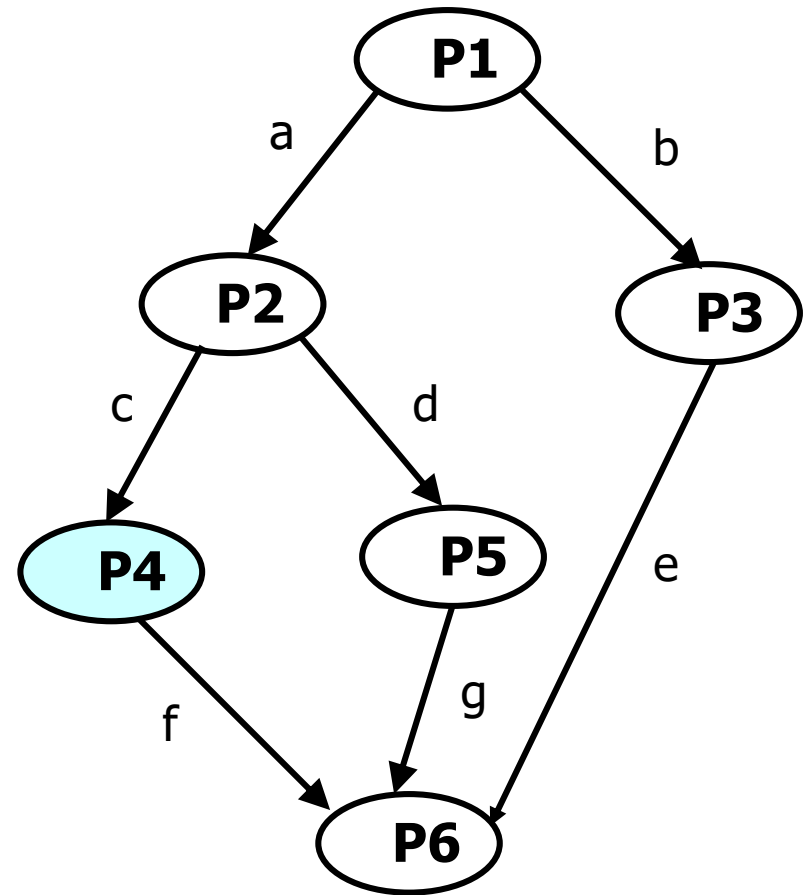
{

p(c);

执行P4的代码;

v(f);

}



解法1 (5)

P5 ()

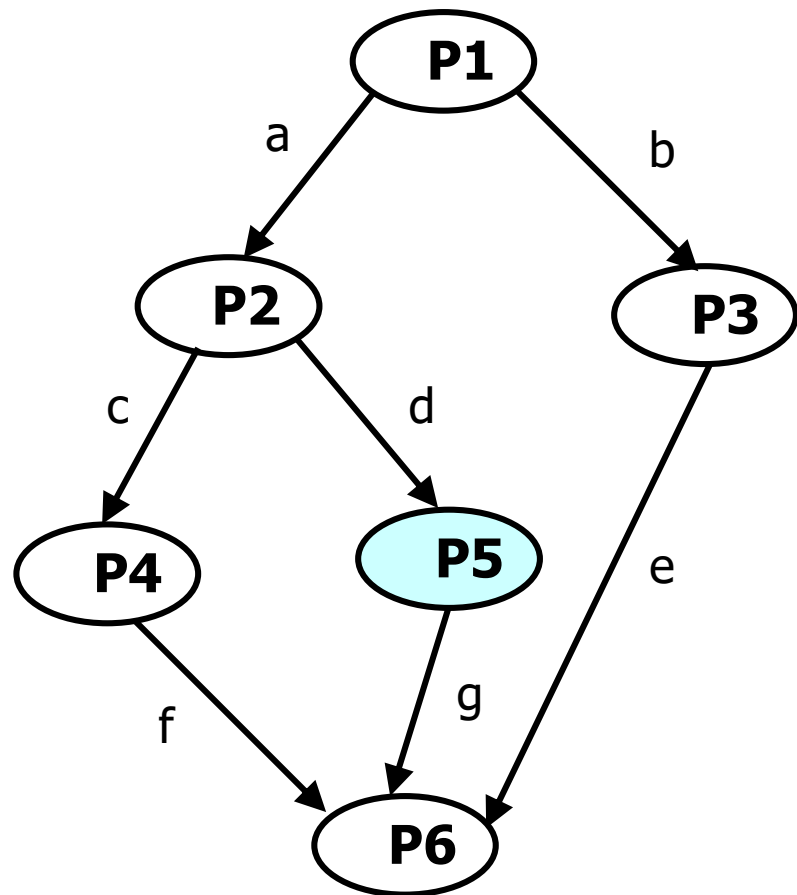
{

p(d);

执行P5的代码;

v(g);

}



解法1 (6)

P6 ()

{

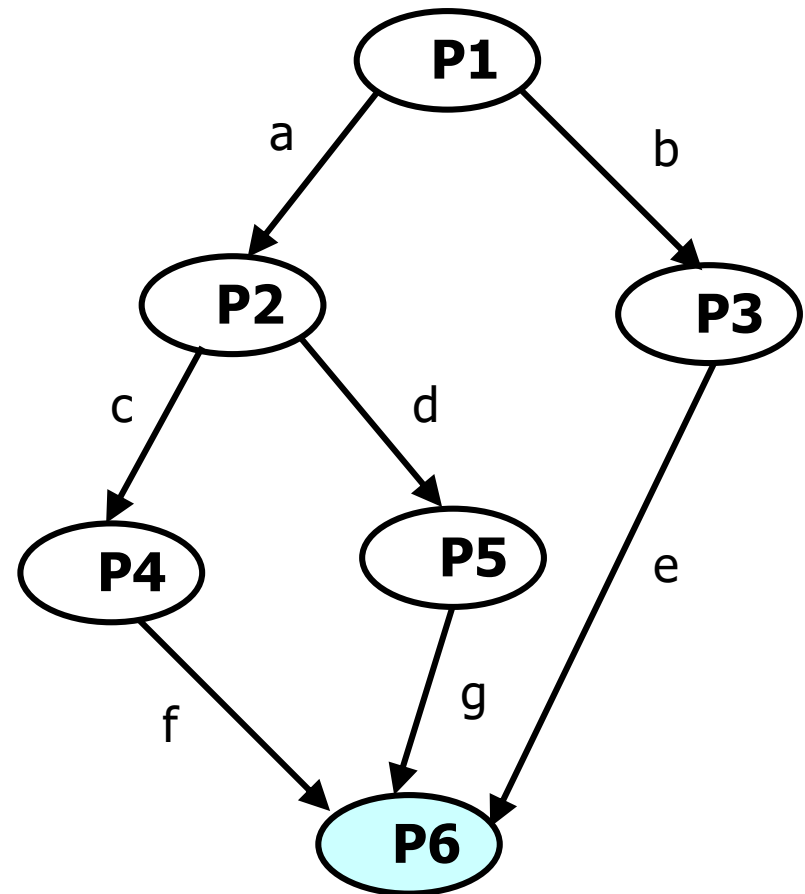
 p(e);

 p(f);

 p(g);

 执行P6的代码;

}



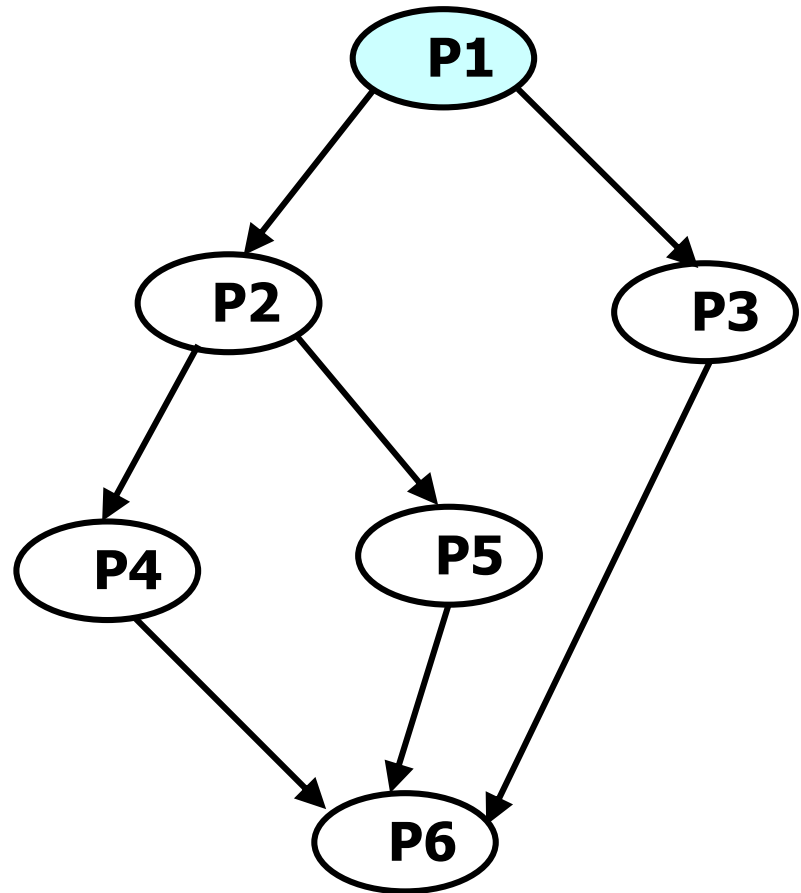


解法2

- 提示：设五个同步信号量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。这六个进程的同步描述如何？

解法2 (1)

```
P1 ()  
{  
    执行P1的代码;  
    v(f1);  
    v(f1);  
}
```



解法2 (2)

P2 ()

{

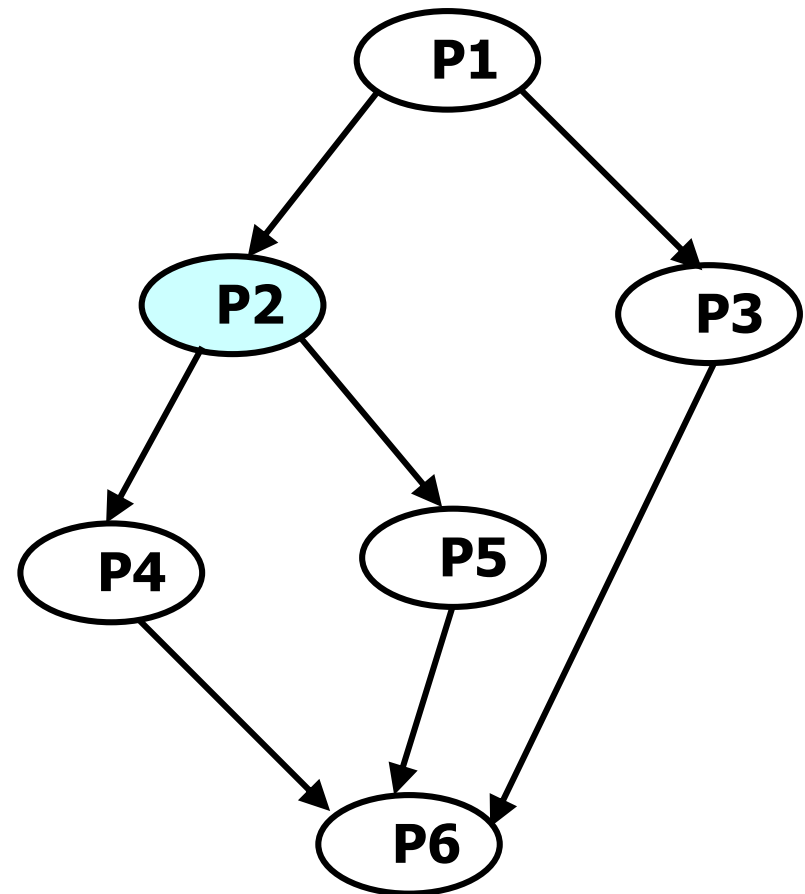
 p(f1);

 执行P2的代码;

 v(f2);

 v(f2);

}



解法2 (3)

P3 ()

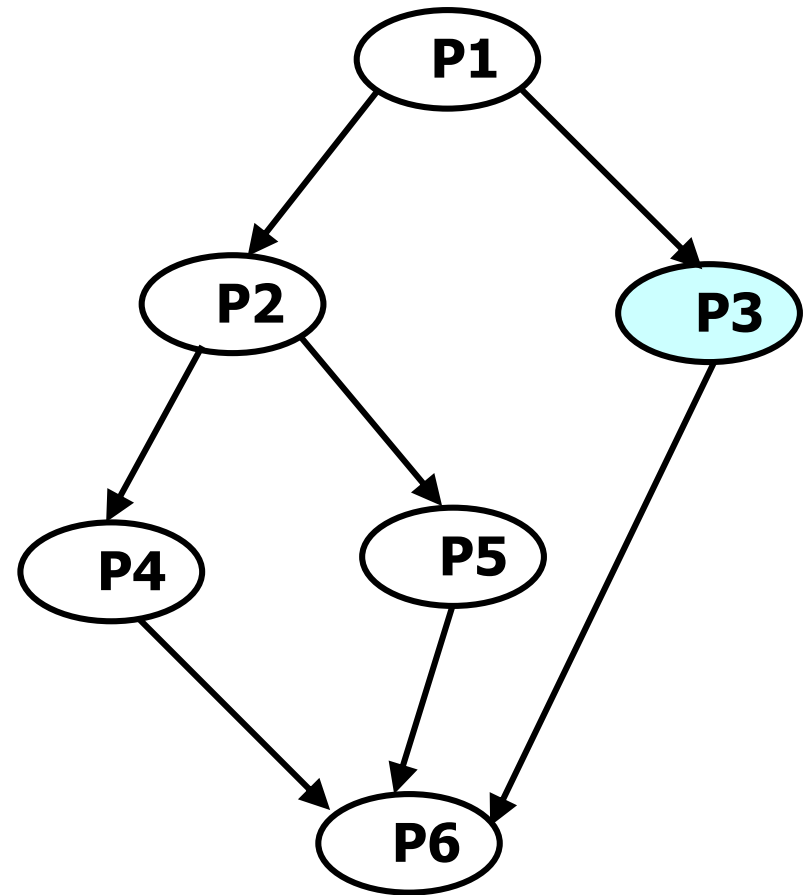
{

 p(f1);

 执行P3的代码;

 v(f3);

}



解法2 (4)

P4 ()

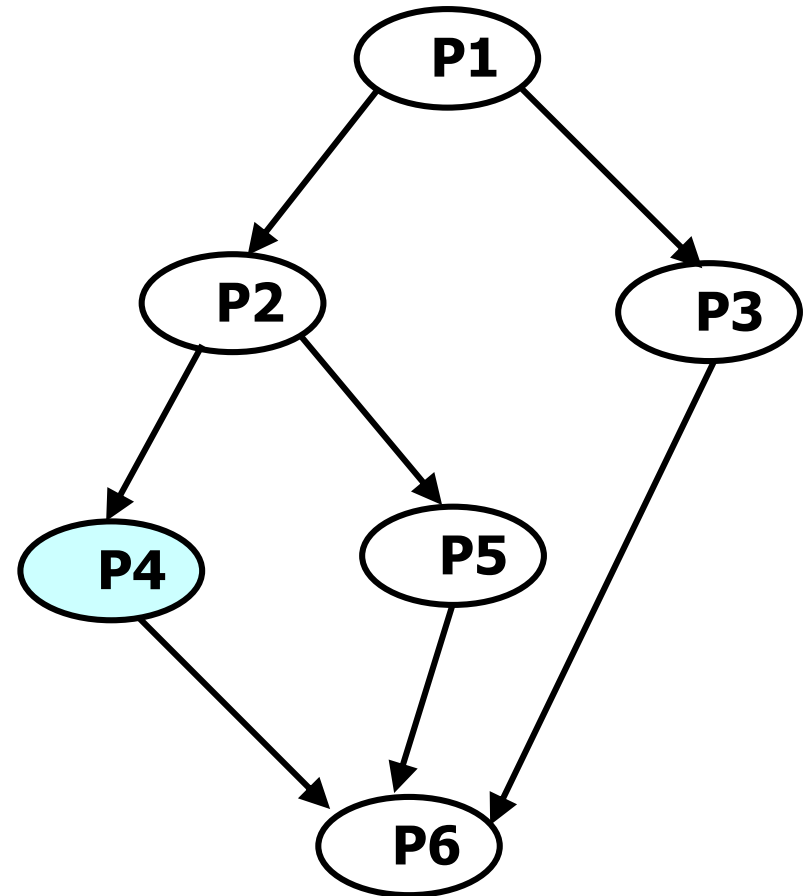
{

p(f2);

执行P4的代码;

v(f4);

}



解法2 (5)

P5 ()

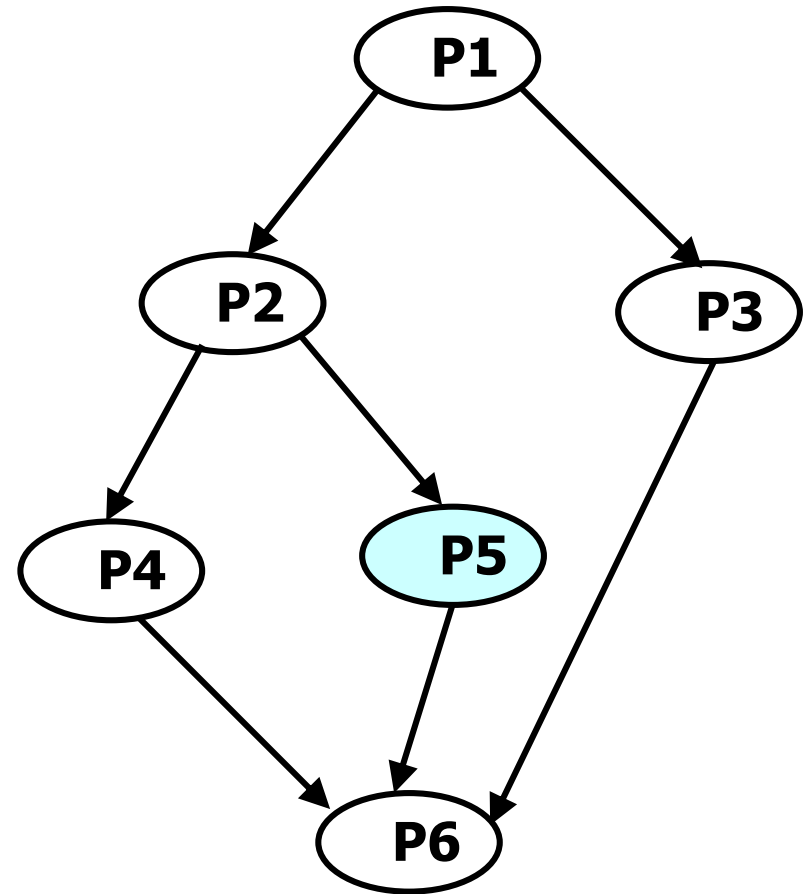
{

 p(f2);

 执行P5的代码;

 v(f5);

}



解法2 (6)

P6 ()

{

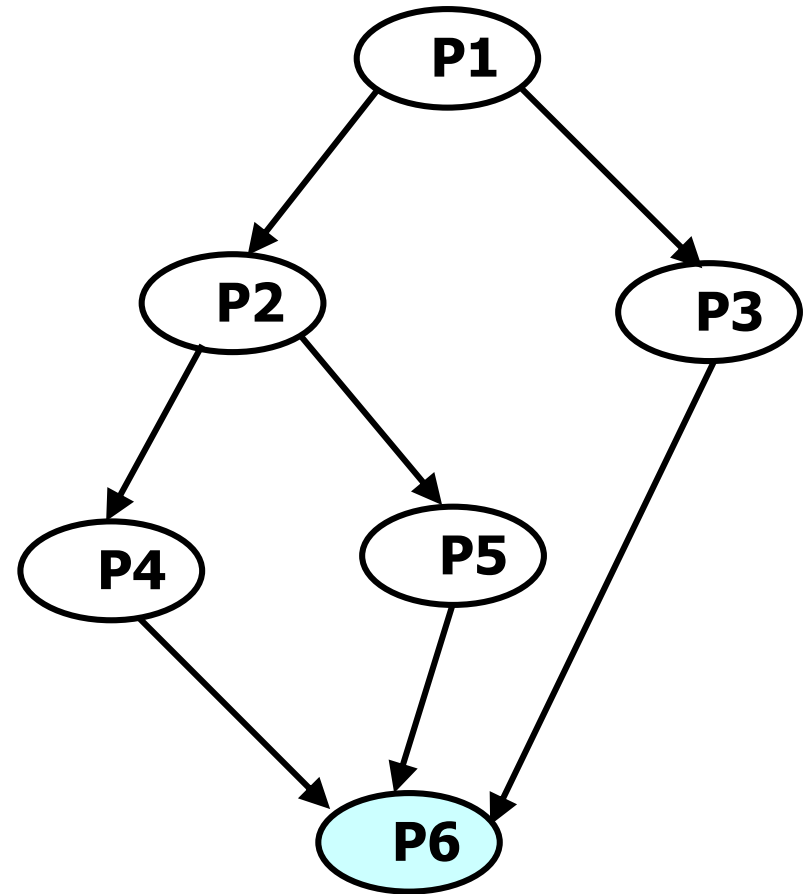
 p(f3);

 p(f4);

 p(f5);

 执行P6的代码;

}



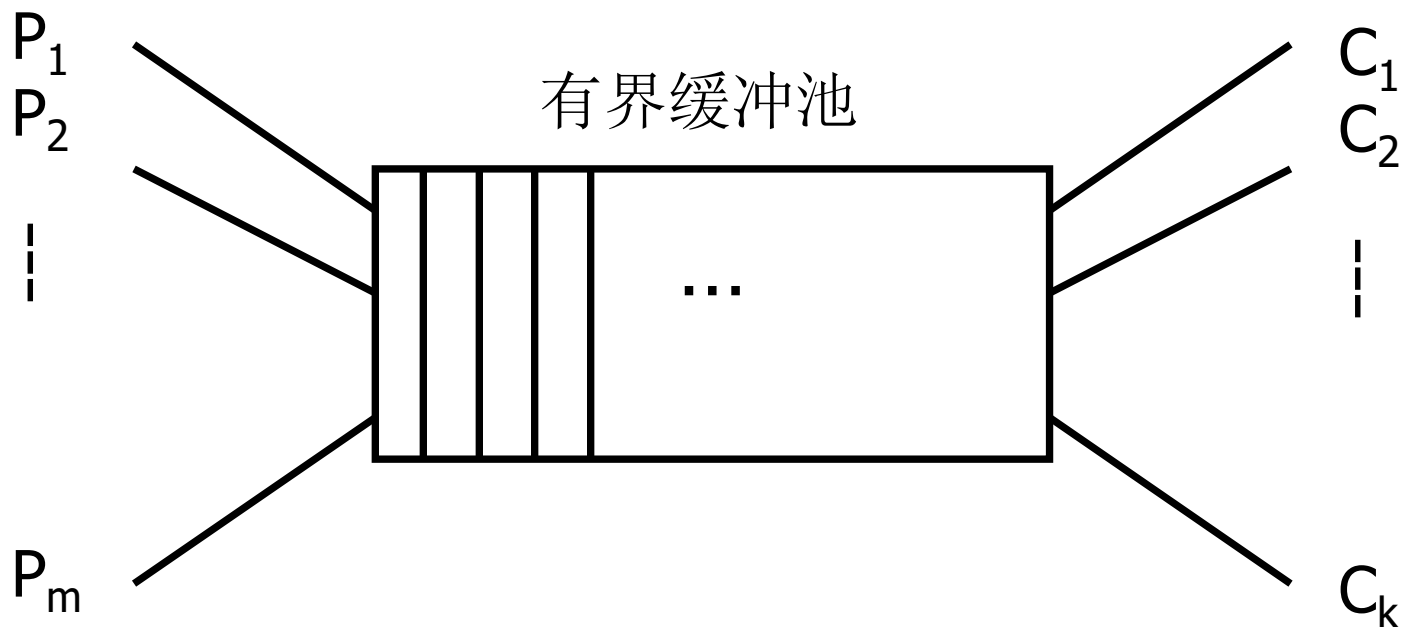


4.3.4 经典进程同步问题

- 多道程序环境中的进程同步是一个非常有趣的问题，吸引了很多学者研究，从而产生了一系列经典进程同步问题。

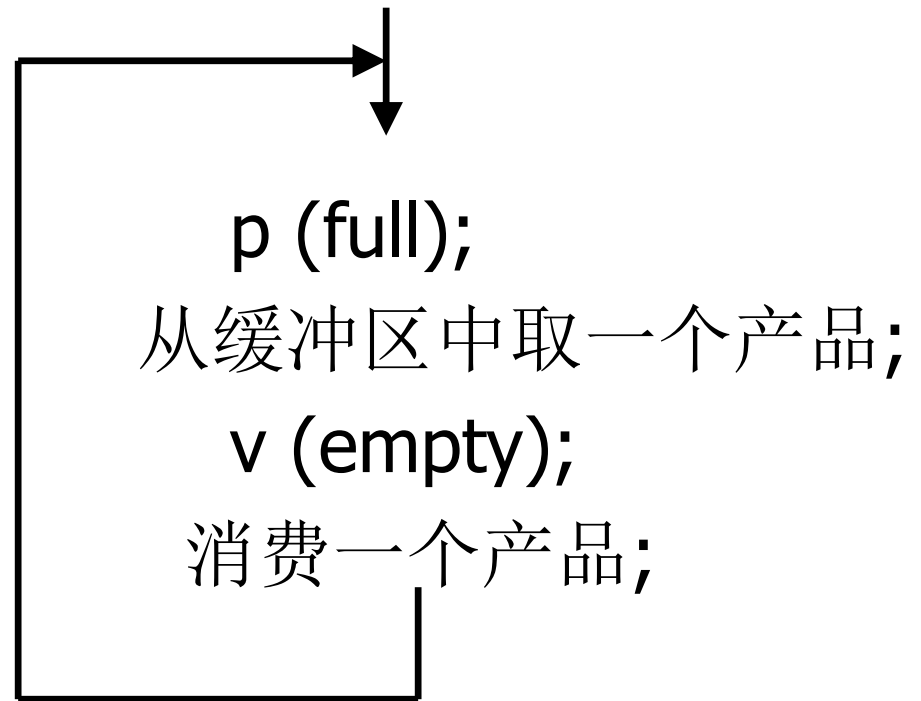
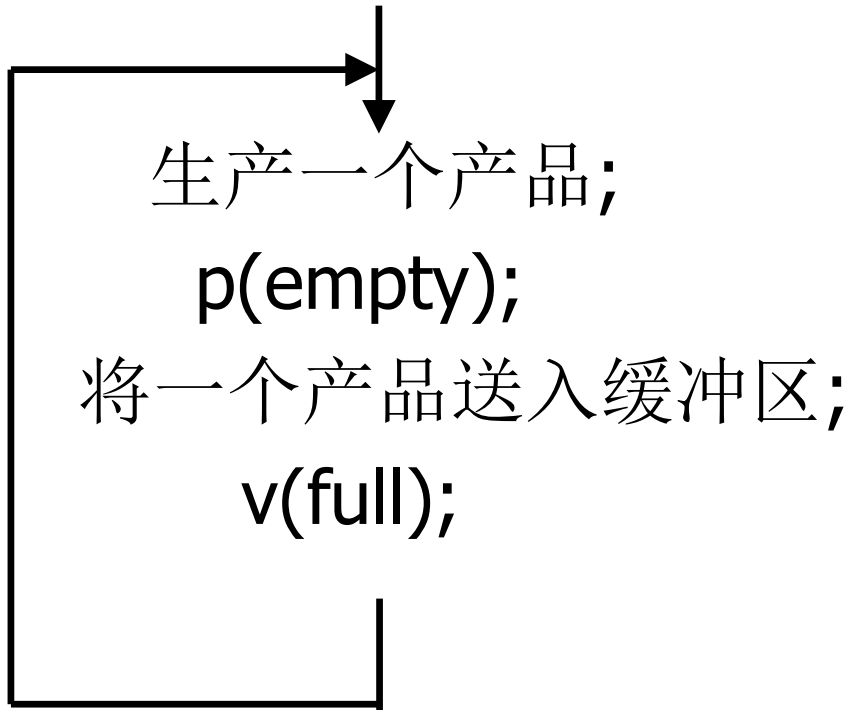
1、生产者—消费者问题

- 同步关系有：当缓冲池满时生产者进程需等待，当缓冲池空时消费者进程需等待。诸进程应互斥使用缓冲池。



一个简化：只有一个缓冲区

- 设置两个同步信号量empty、full，其初值分别为1、0



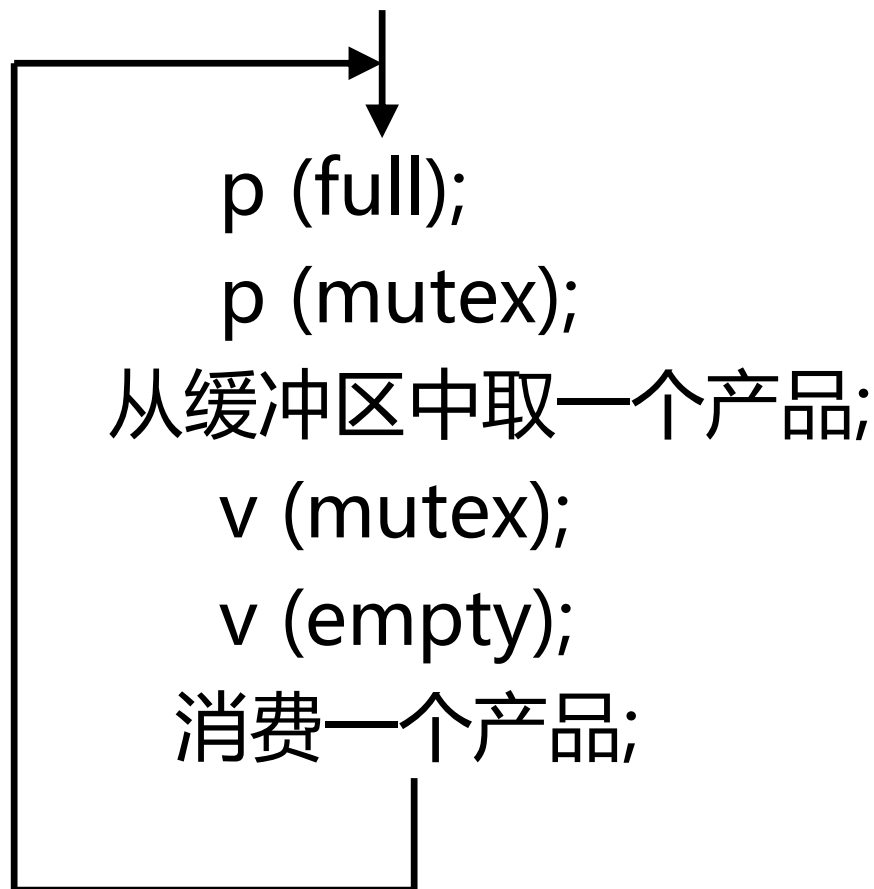
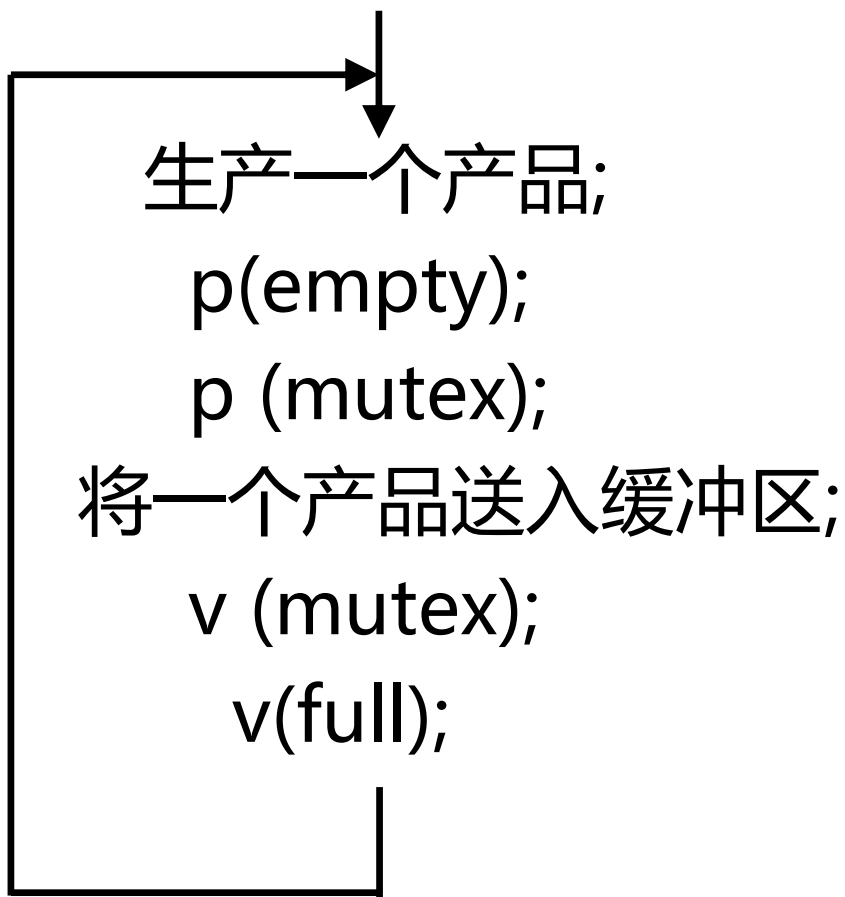


进一步：多个缓冲区n

- 设置两个同步信号量empty、full，其初值分别为n、0。
- 有界缓冲池是一个临界资源，还需要设置一个互斥信号量mutex，其初值为1。
- 生产者—消费者问题的同步描述如下：

生产者

消费者



- **注意：**无论在生产者进程还是在消费者进程中，**P**操作的次序都不能颠倒，否则将可能造成死锁。

颠倒生产者进程中的P操作

当 $\text{mutex}=1$, $\text{full}=n$, $\text{empty}=0$, 且生产者先执行时

生产一个产品;

$\text{p}(\text{mutex});$

$\text{p}(\text{empty});$

将一个产品送入缓冲区;

$\text{v}(\text{mutex});$

$\text{v}(\text{full});$

mutex	full	empty
0	n	-1

$\text{p}(\text{full});$

$\text{p}(\text{mutex});$

从缓冲区中取一个产品;

$\text{v}(\text{mutex});$

$\text{v}(\text{empty});$

消费一个产品;



2.读者—写者问题

- 一个数据对象（如文件或记录）可以被多个并发进程所共享
- 其中有些进程只要求读数据对象的内容——读者
- 而另一些进程则要求修改或写数据对象的内容——写者
- 允许多个读进程同时读此数据对象
- 但是一个写进程不能与其他进程（不管是写进程还是读进程）同时访问此数据对象



读者—写者问题分类

- **读者优先：**

- 当存在一个读者时候，读者优先，写者会长时间等待。

- **写者优先：**

- 当写者提出存取共享对象的要求后，已经开始读的进程让其读完，但不允许新读者进入。
- 写者结束时，判断是否有写者等，有则优先写者

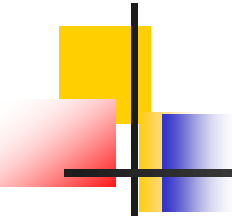
- **读写者公平：**

- 按提出请求的顺序先来先服务
- 读者前面无写者等待时，可进入
- 读者前面有写者等待时，等待



用信号量解决读者优先问题

- 为解决读者优先问题，应设置两个信号量和一个共享变量：
 - 互斥信号量rmutex，用于使读进程互斥地访问共享变量readcount，其初值为1；
 - 共享变量readcount，用于记录当前正在读数据集的读进程数目，初值为0。
 - 写互斥信号量wmutex，用于实现写进程与读进程的互斥以及写进程与写进程的互斥，其初值为1；



```
p(rmutex);  
if (readcount==0) p(wmutex);  


---

readcount++;  
v(rmutex);  
读数据集;  
p(rmutex);  
readcount--;  
if (readcount==0) v(wmutex);  
v(rmutex);
```

```
p(wmutex);  
写数据集;  
v(wmutex);
```



对读者优先问题的理解

- 请注意对信号量rmutex意义的理解。
- rmutex是一个互斥信号量，用于使读进程互斥地访问共享变量readcount。该信号量并不表示读进程的数目，表示读进程数目的是共享变量readcount。
- 问题首次讨论于：
 - P.J. Courtois, F. Heymans. Concurrent Control with "Readers" and "Writers" . Communications of the ACM. 1971.



写者优先

- 应该满足的要求
 - 多个读者可以同时进行读；
 - 写者必须互斥
 - 只允许一个写者写，不能读者写者同时进行
 - 写者优先于读者
 - 一旦有写者，则后续读者必须等待，在唤醒时优先考虑写者，直到最后一个写者完成
 - 对于已经开始的读者让其读完，对于未开始的读者让其等待
 - 如果一堆读者排在写者前面，应该按照排队顺序工作，or写者能插队吗？



这个对吗?

Reader()

```
{  
    P(mutexReadCount);  
    P(r);  
    readcount ++;  
    V(r);  
    V(mutexReadCount);  
    reading is performed....  
    P(mutexReadCount);  
    readcount --;  
    V(mutexReadCount);  
}
```

Writer()

```
{  
    P(mutexWriteCount);  
    //写者入队列  
    writecount ++;  
    //若为第一个写者，阻止后续的读者  
    if (writecount==1) P(r);  
    V(mutexWriteCount);  
    //互斥其他的写者  
    P(w);  
    writing is performed...  
    V(w);  
    P(mutexWriteCount);  
    writecount --;  
    if(writecount == 0) V(r);  
    V(mutexWriteCount);  
}
```



这个对吗?

Reader()

```
{
    P(mutexReadCount);
    P(r);
    readcount ++;
    //若为第一个读者，互斥写者
    if (readcount==1) P(w);
    V(r);
    V(mutexReadCount);
    reading is performed....
    P(mutexReadCount);
    readcount --;
    //若当前读者为最后一个，则唤醒写者
    if (readcount==0) V(w);
    V(mutexReadCount);
}
```

Writer()

```
{
    P(mutexWriteCount);
    //写者入队列
    writecount ++;
    //若为第一个写者，阻止后续的读者
    if (writecount==1) P(r);
    V(mutexWriteCount);
    //互斥其他的写者
    P(w);
    writing is performed...
    V(w);
    P(mutexWriteCount);
    writecount --;
    if(writecount == 0) V(r);
    V(mutexWriteCount);
}
```




这个还有问题吗？

Reader()

```
{
    P(r);
    P(mutexReadCount);
    readcount ++;
    //若为第一个读者，互斥写者
    if (readcount==1) P(w);
    V(mutexReadCount);
    V(r);
    reading is performed....
    P(mutexReadCount);
    readcount --;
    //若当前读者为最后一个，则唤醒写者
    if (readcount==0) V(w);
    V(mutexReadCount);
}
```

Writer()

```
{
    P(mutexWriteCount);
    //写者入队列
    writecount ++;
    //若为第一个写者，阻止后续的读者
    if (writecount==1) P(r);
    V(mutexWriteCount);
    //互斥其他的写者
    P(w);
    writing is performed...
    V(w);
    P(mutexWriteCount);
    writecount --;
    if(writecount == 0) V(r);
    V(mutexWriteCount);
}
```



写者优先解法

变量int readcount=0, writecount = 0。

设置5个信号量: mutexReadCount, mutexWriteCount, mutexPriority, r, w.

Reader()

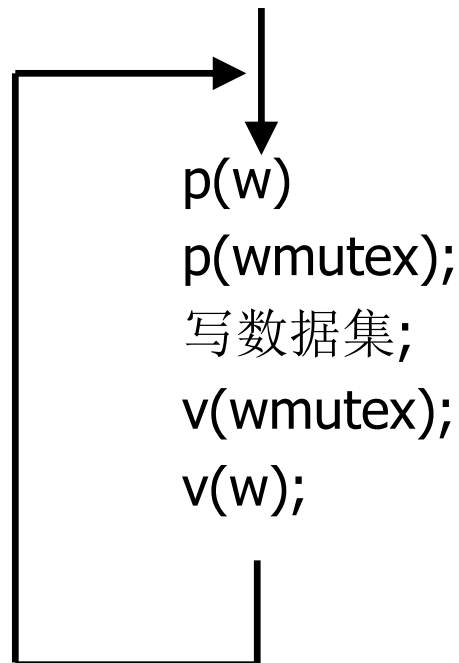
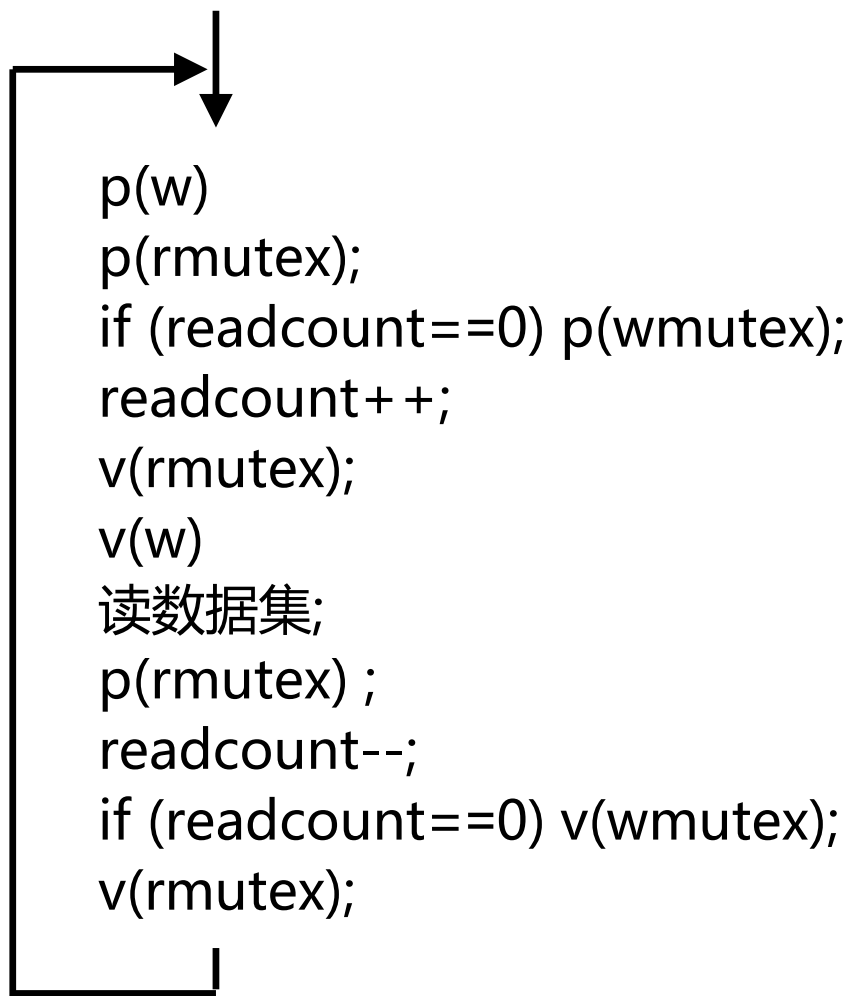
```
{
    P(mutexPriority) ;
    P(r);
    P(mutexReadCount);
    readcount ++;
    //若为第一个读者，互斥写者
    if (readcount==1) P(w);
    V(mutexReadCount);
    V(r);
    V(mutexPriority);
    reading is performed....
    P(mutexReadCount);
    readcount --;
    //若当前读者为最后一个，则唤醒写者
    if (readcount==0) V(w);
    V(mutexReadCount);
}
```

Writer()

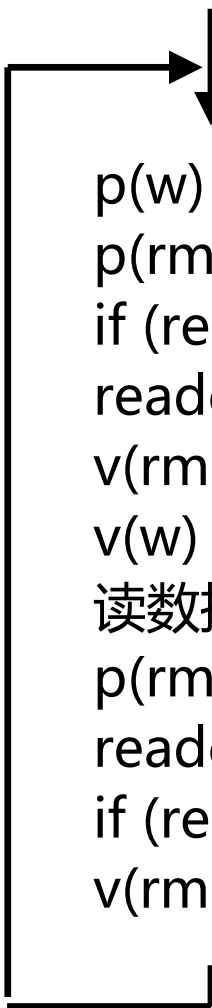
```
{
    P(mutexWriteCount);
    //写者入队列
    writecount ++;
    //若为第一个写者，阻止后续的读者
    if (writecount==1) P(r);
    V(mutexWriteCount);
    //互斥其他的写者
    P(w);
    writing is performed...
    V(w);
    P(mutexWriteCount);
    writecount --;
    if(writecount == 0) V(r);
    V(mutexWriteCount);
}
```



思考一下，如下算法是怎样效果呢？

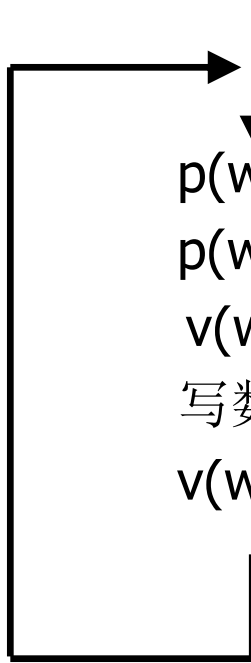


思考一下，如下算法又是怎样效果呢？



```
graph TD; Entry(( )) --> P1[p(w)]; P1 --> P2[p(rmutex)]; P2 --> P3["if (readcount==0) p(wmutex);"]; P3 --> P4["readcount++;"]; P4 --> P5[v(rmutex);]; P5 --> P6[v(w)]; P6 --> P7["读数据集;"]; P7 --> P8[p(rmutex);]; P8 --> P9["readcount--;"]; P9 --> P10["if (readcount==0) v(wmutex);"]; P10 --> P11[v(rmutex);]; P11 --> Entry;
```

p(w)
p(rmutex);
if (readcount==0) p(wmutex);
readcount++;
v(rmutex);
v(w)
读数据集;
p(rmutex);
readcount--;
if (readcount==0) v(wmutex);
v(rmutex);



```
graph TD; Entry(( )) --> P1[p(w)]; P1 --> P2[p(wmutex);]; P2 --> P3[v(w);]; P3 --> P4["写数据集;"]; P4 --> P5[v(wmutex);]; P5 --> Entry;
```

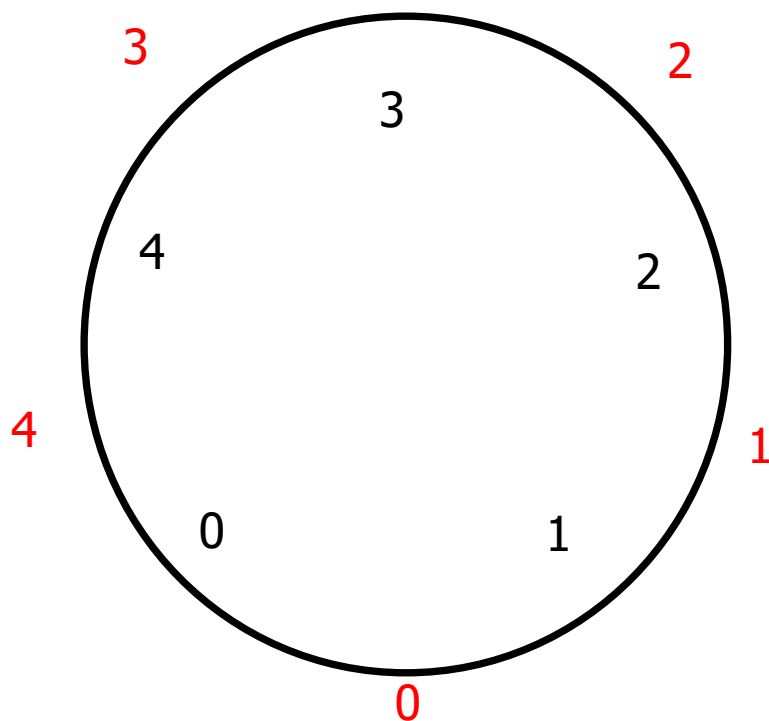
p(w)
p(wmutex);
v(w);
写数据集;
v(wmutex);



3. 哲学家进餐问题

- 哲学家进餐问题描述：
 - 有五个哲学家，他们的生活方式是交替地进行思考和进餐，
 - 哲学家们共用一张圆桌，分别坐在周围的五张椅子上，在圆桌上有一盆米饭和五支筷子，
 - 平时哲学家进行思考，饥饿时便试图取其左、右最靠近他的筷子，只有在他拿到两支筷子时才能进餐，
 - 进餐完毕，放下筷子又继续思考。

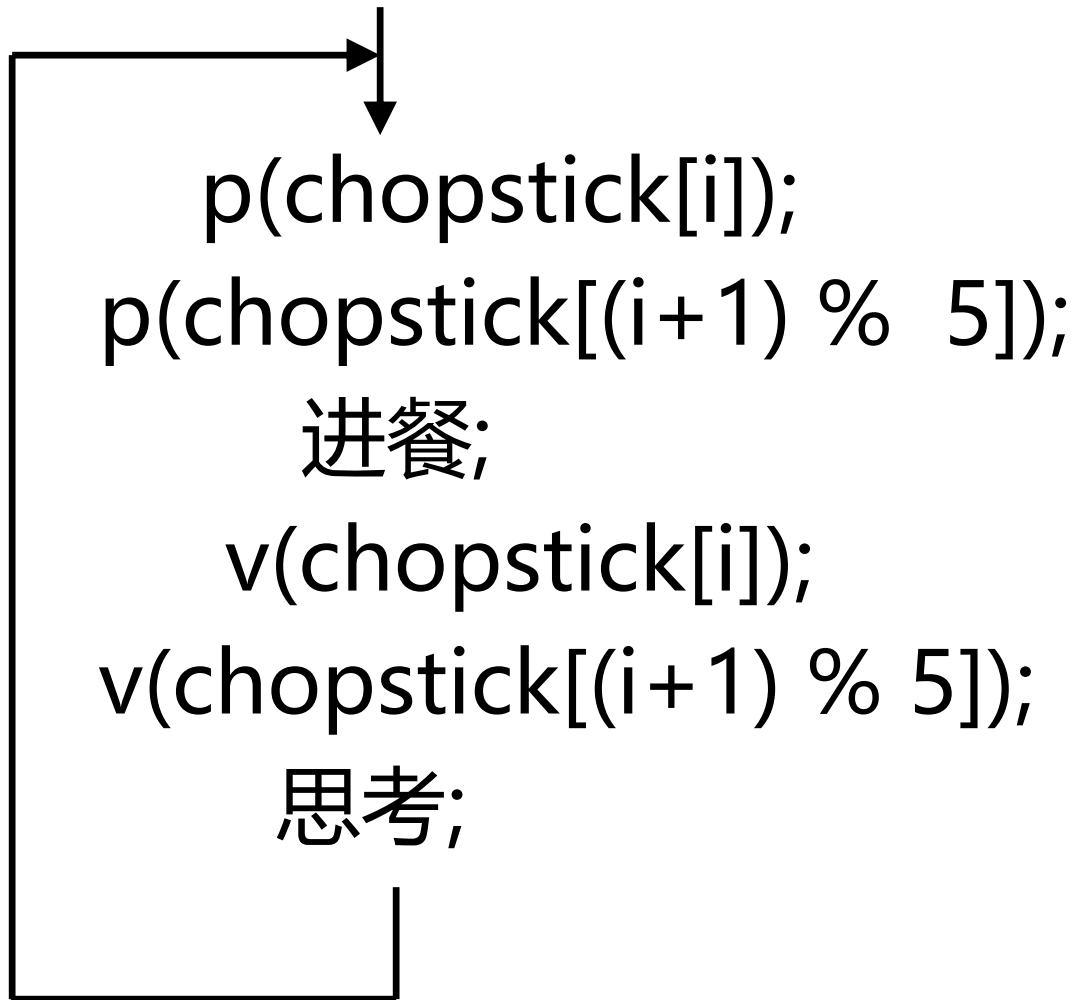
用信号量解决哲学家进餐问题



- 用五支筷子的信号量构成信号量数组：
 - semaphore chopstick[5];
 - 所有信号量初值为1,



某哲学家活动的简单描述



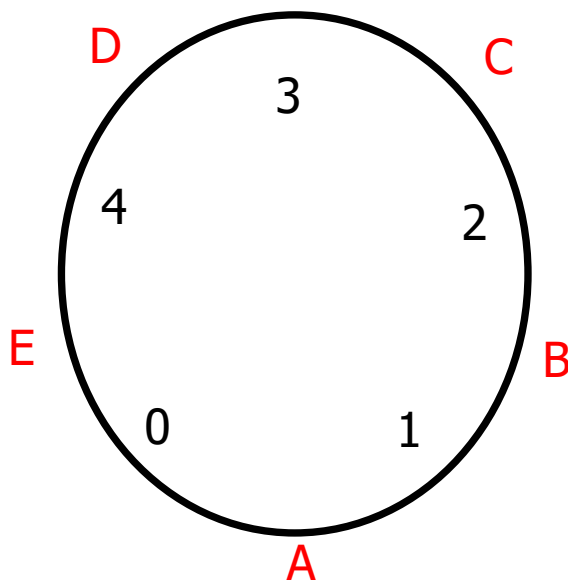


算法描述存在的问题

- 上述算法有可能引起死锁
 - 如所有人都并发的先拿起右边的筷子，然后再拿起左边的筷子。
- 对于这样的死锁问题有如下办法解决：
 - 破坏请求保持条件
 - 只有一个哲学家两个筷子都可用时候，才能拿起筷子，即在临界区里面拿起两根筷子
 - 引入一个服务生来监管
 - 破坏环路
 - 至多4个哲学家同时进餐
 - 采用非对称方案，奇数号哲学家先拿左筷子再拿右筷子，偶数号哲学家相反。
 - 按照一定顺序获取资源，按照相反顺序释放资源。破坏环路 {资源分级法}

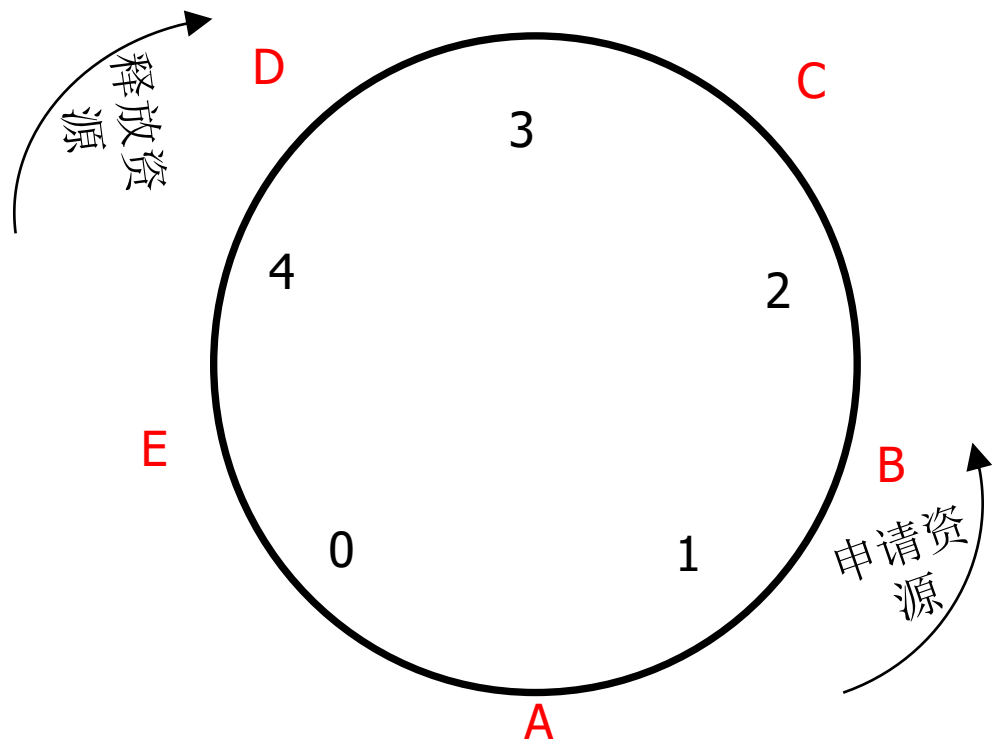
服务生解法

- 引入一个餐厅服务生，哲学家必须经过他的允许才能拿起筷子
- 服务生：了解当前筷子使用情况，能够正确判断，避免死锁



资源分级解法

- 为资源分配一个偏序关系
- 获取：按照这一顺序
- 释放：按照相反顺序
- 保证不会有两个无关资源同时被同一项工作所需要。





资源分级解法

- 筷子编号为#0、.....、#4
- 取筷子原则：每个哲学家总先拿起左右两边编号较低的筷子，再拿编号较高的。
- 放筷子规则：用完餐后，他总是先放下编号较高的筷子，再放下编号较低的。
- 为什么不会产生死锁？
 - 当A—E号哲学家同时拿起他们手边编号较低的筷子时（#0—3），则只有#4筷子留在桌上
 - 此时A—E之一能拿起最高号的筷子，吃完，并放下筷子，从而让其他哲学家继续。



资源分级解法

■ 不足：

- 尽管资源分级能避免死锁，但这种策略并不总是实用的，特别是当所需资源的列表并不是事先知道的时候，且效率存在问题。
- 例如，假设一个工作单元拿着资源3和5，并决定需要资源2，则必须先要释放5，之后释放3，才能得到2，之后必须重新按顺序获取3和5。



Chandy/Misra解法

- 筷子：每只筷子都是“干净的”或者“脏的”。
- 初始分配：
 - 对每一对竞争一个资源的哲学家，就新拿一个筷子，并分给标号较低的哲学家。最初每一只筷子都是脏。
- 请求：
 - 当一位哲学家要使用资源时，他必须从与他竞争的邻居那里得到。对每支他当前没有的筷子，他都发送一个请求
- 响应：
 - 当拥有筷子的哲学家收到请求时，如果筷子是干净的，那么他继续留着，否则就擦干净并交出筷子
- 吃饭：
 - 当某个哲学家拥有两个筷子，就能吃东西
 - 吃东西后，他的筷子就变脏了
 - 如果另一个哲学家之前请求过其中的筷子，那他就擦干净并交出筷子。



4.睡眠的理发师问题

- 理发店里有一位理发师，一把理发椅和N把供等候理发顾客坐的椅子。
- 如果没有顾客，理发师睡眠，当一个顾客到来时叫醒理发师；
- 若理发师正在理发时又有顾客来，那么有空椅子就坐下，否则离开理发店。



用信号量解决睡眠的理发师问题

- 为解决睡眠的理发师问题，应使用三个信号量：
 - Customers记录等候理发的顾客数（不包括正在理发的顾客）；初值为0
 - barbers记录正在等候理发师的顾客数；初值为0
 - mutex用于互斥访问count。初值为1
- 一个变量：
 - 变量count记录等候的顾客数，它是customers的一份拷贝。之所以使用count是因为无法读取信号量的当前值。

问题描述

```
Barber(){
    while(true){
        //是否有等候的顾客,若无顾客,理发师睡眠
        p(customers);
        //若有顾客,则进入临界区
        p(mutex);
        //等待中的顾客数减1
        count--;
        //理发师发出通知,已经准备好为顾客理发
        v(barbers);
        //退出临界区
        v(mutex);
        cut_hair();
    }
}
```

```
Customer-i (){
    p(mutex);
    //判断是否有空椅子
    if(count < N)
    {
        //等待顾客数加1
        count++;
        //唤醒理发师
        v(customers);
        //退出临界区
        v(mutex);
        //等待理发师准备好,如果理发师忙,
        //顾客等待
        p(barbers);
        get_haircut();
    }
    else
        //无空椅子人满了,顾客离开
        v(mutex);
}
```




讨论:

- 1.为什么在baber中, $P(\text{mutex})$ 不可以在 $p(\text{customer})$ 之前出现?
- 2.为什么在baber中, $v(\text{mutex})$ 不可以在 $v(\text{barbers})$ 之前?
- 3.为什么在customer中, $v(\text{customers})$ 与 $\text{count}++$ 要放到 mutex 保护中

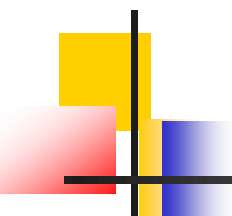


讨论1:

为什么在barber中, P(mutex)不可以在p(customer)之前出现?

```
Barber(){  
    while(true){  
        p(mutex);  
        p(customers);  
        count--;  
        v(barbers);  
        v(mutex);  
        cut_hair();  
    }  
}
```

```
Customer-i (){  
    p(mutex);  
    if(count < N)  
    {  
        count+ +;  
        v(customers);  
        v(mutex);  
        p(barbers);  
        get_haircut();  
    }  
    else  
        v(mutex);
```



讨论2：为什么在barber中，v(mutex)和v(babers)不能反过来？

```
Barber(){  
    while(true){  
        p(customers);  
        p(mutex);  
        count--;  
        v(mutex);  
        v(barbers);  
        cut_hair();  
    }  
}
```

```
Customer-i (){  
    p(mutex);  
    if(count < N)  
    {  
        count+ +;  
        v(customers);  
        v(mutex);  
        p(barbers);  
        get_haircut();  
    }  
    else  
        v(mutex);
```



讨论3:

为什么在customer中，v(customers)与count++要放到mutex保护中

```
Barber(){  
    while(true){  
        p(customers);  
        p(mutex);  
        count--;  
        v(barbers);  
        v(mutex);  
        cut_hair();  
    }  
}
```

```
Customer-i (){  
    p(mutex);  
    if(count < N)  
    {  
        count++;  
        v(mutex);  
        v(customers);  
        p(barbers);  
        get_haircut();  
    }  
    else  
        v(mutex);
```



讨论:

```
Barber(){  
    while(true){  
        p(customers);  
        p(mutex);  
        count--;  
        v(barbers);  
        v(mutex);  
        cut_hair();  
    }  
}
```

```
Customer-i (){  
    p(mutex);  
    if(count < N)  
    {  
        count+ +;  
        v(customers);  
        v(mutex);  
        p(barbers);  
        get_haircut();  
    }  
    else  
        v(mutex);
```



利用信号量解决同步问题的思路

- 理清同步与互斥关系
 - 哪些资源及对象需要互斥访问
 - 哪些资源的访问顺序对进程调度有制约关系
 - 同步信号量要表示出资源的等待条件及数目
 - P操作内包含等待；V操作内包含唤醒
 - 依多个访问顺序约束，同类资源可设置多个信号量
 - 生产者消费者问题中的empty和full
 - 一定要注意互斥量与同步信号量的顺序
 - 同步P优先于互斥P
 - 信号量的操作只能为PV，切记不要直接取指和赋值
 - 可设置副本，如理发师问题中的count 和customers



小作业2

- 1. 有P1、P2、P3三类进程（每类进程都有K个）共享一组表格F（F有N个），P1对F只读不写，P2对F只写不读，P3对F先读后写。进程可同时读同一个 F_i ($0 \leq i \leq N$)；但有进程写时，其他进程不能读和写。
 - 用信号量和P、V操作给出方案
 - 对方案的正确性进行分析说明
 - 对访问的公平性进行分析

小作业2

- 2.三组进程P1 (K个, $K > 1$)、P2 (1个)、P3 (1个)、互斥使用一个包含 $N(N > 1)$ 个单元的缓冲区buf1, P2-1、P3-1、P4 (各有1个) 执行定期统计
 - P1每次用produce(), 生成一个正整数并用put()送入缓冲区buf1某一空单元中, 循环访问缓冲区
 - P2每次用getodd()从buf1缓冲区取出一个奇数, 放到自己私有缓冲区buf2中(缓冲区长度为 $M > 1$), 然后由P2-1 (1个) 进程读取P2的私有缓冲区, 使用countodd()统计奇数个数
 - P3每次用geteven()从buf1缓冲区中取出一个偶数, 放到自己私有缓冲区buf3中(缓冲区长度为 $M > 1$), 然后由P3-1 (1个) 进程读取P3的私有缓冲区, 使用counteven()统计偶数个数
 - P4在P2-1、P3-1产生一个统计值后, 就输出一个包含统计时间的结果
 - 请用信号量机制实现这几个进程的同步与互斥活动, 并说明所定义的信号量含义。



小作业2

■ 3. 独木桥问题

- ① 独木桥只允许一台汽车过河，当车到达桥头时，如果桥上无车，则可上桥，否则车在桥头等待，直到桥上无车。使用PV操作解决该问题。
- ② 如果独木桥允许多台车同方向过河，当车到达桥头时，如果桥上只有同方向车且不超过N台，或者桥上无车，则可上桥通过，否则等待，直到满足上桥条件。使用PV操作解决该问题。

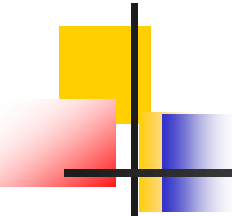
■ 4. 红黑客过河问题

- 有红客和黑客两组人员需要过河。河上有船，但是每次只能乘坐4人，且每次乘客满员时才能开船，到河对岸后空船返回。但船上乘客不能同时有3个红客、1个黑客 或者 1个红客、3个黑客的组合。（其他组合安全）。请编写程序，用PV操作解决红客、黑客过河的问题。



大作业

1. 参考V9版，课后编程项目，针对读者/写者、生产者/消费者、哲学家问题、理发师（or睡觉助教），选择一个实现，要求使用Pthread和 Windows 线程API实现



第4章 进程同步

4.4 管程



4.4 管程

- 信号量的同步操作分散在各进程中不便于管理，还可能导致系统死锁。如：生产者消费者问题中将P颠倒可能死锁。
- 为此Dijkstra于1971年提出：把所有进程对某一种临界资源的同步操作都集中起来，构成一个所谓的秘书进程。凡要访问该临界资源的进程，都需先报告秘书，由秘书来实现诸进程对同一临界资源的互斥使用。



有了管程的优势

- 把分散在各进程中的临界区集中起来进行管理；
- 防止进程有意或无意的违法同步操作，
- 便于用高级语言来书写程序，也便于程序正确性验证。



4.4.1 管程定义

- Hansen为管程(Monitor)所下的定义是：管程定义了一个数据结构和能为并发进程所执行的一组操作，这组操作能同步进程和改变管程中的数据。



管程的构成

- 局部于管程的共享数据结构
- 对共享数据结构进行操作的一组函数
- 对局部于管程的数据设置初始值的语句



管程的语法

```
type 管程名=monitor {  
    局部变量说明;  
    条件变量说明;  
    初始化语句;  
define 管程内定义的, 管程外可调用的过程或函数名列表;  
use 管程外定义的, 管程内将调用的过程或函数名列表;  
过程名/函数名(形式参数表) {  
    <过程/函数体>;  
}  
  
    ...  
过程名/函数名(形式参数表) {  
    <过程/函数体>;  
}  
}
```


管程的基本特性

1. 安全性

- 管程内的局部数据**只能**被该管程内的函数所访问。

2. 共享性

- 进程对管程的访问**都可以通过**共享相同的管程函数来实现。

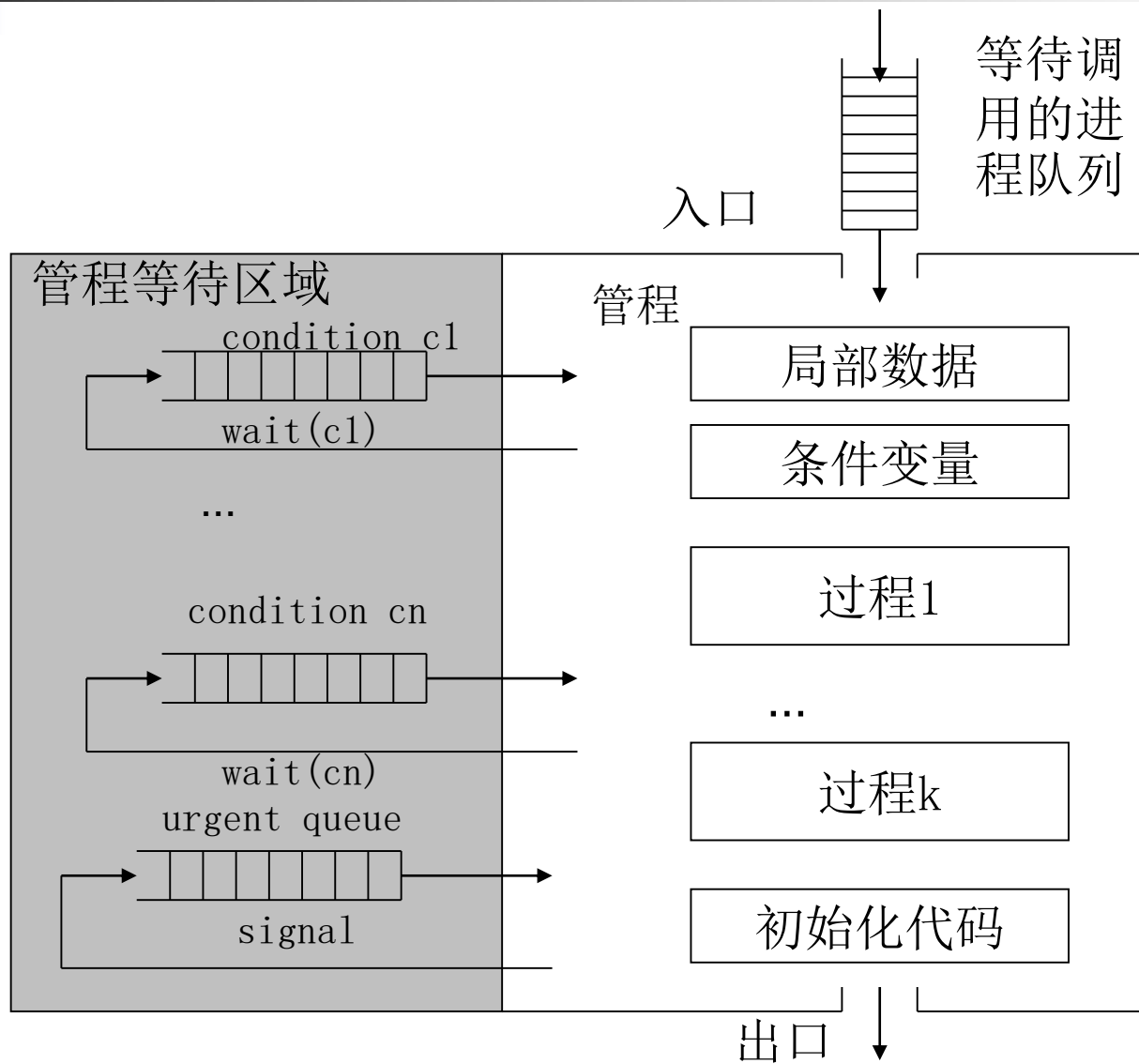
3. 互斥性:

- **每次仅允许一个**进程在管程内执行某个函数，共享资源的进程可以访问管程，但是**只有至多一个**调用者**真正**进入管程，其他调用者**必须**等待。

4. 透明性

- 管程是一个语言成分，所以管程的互斥访问完全由编译程序在编译时自动添加上，**无需**程序员关心，而且保证正确。

管程的结构



管程中涉及的同步等问题 (1)

- 条件变量：
 - 当调用管程过程的进程无法运行时，用于阻塞进程的一种特殊的信号量。相应地，还应该设置同步等待原语与同步唤醒原语。
- 同步原语C.wait与C.Signal
 - 当一个调用管程过程的进程发现无法继续时，它在某些条件变量condition上执行C.wait，这个动作引起调用进程阻塞
 - 另一个进程可以通过对同一个条件变量condition上执行同步原语C.signal操作来唤醒等待进程。



条件变量

- 条件变量的基本定义形式为：condition: x,y;
- 注意：条件变量与P、V操作中信号量是不同的
 - P、V操作内部会对信号量进行计数，通过对累计值的管理，实现对进程间的关系进行有效管理
 - 条件变量只是一种简单的信号量，只进行维护等待队列的操作，不进行计数，没有累计值。若条件变量上没有等待的进程，该信号量会被丢弃，即signal触发一个空操作。



管程中涉及的同步等问题（2）

- 使用signal释放等待进程时，可能造成调用signal的进程在管程中，被恢复的那个进程也在管程中，即出现两个进程同时停留在管程内
- 解决方法：
 1. 执行signal的进程等待，直到被释放进程退出管程或等待另一个条件
 2. 被释放进程等待，直到执行signal的进程退出管程或等待另一个条件
- 霍尔（Tony Hoare）的方法采用了第一种办法
- 汉森选择了两者的折衷，规定管程中的过程所执行的signal操作是过程体的最后一个操作



4.4.2 Hoare方法管程的实现

- 霍尔方法的原则：
 - 使用P和V操作原语来实现外部进程对管程中过程的互斥调用，并实现对共享资源互斥使用的管理。
 - 当signal唤醒另一个进程时候，调用了signal操作的进程进入等待，直到其他进程操作完成退出，或者由于其他条件而等待。
 - wait和signal操作可被设计成可以中断的过程。



Hoare管程的数据结构

1.mutex: 实现管程互斥调用所需信号量

- ①当进程调用管程时候，管程将使用内部的互斥变量mutex(初值为1)，实现仅一个进程能调用管程中的过程运行。
- ②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。
- ③为了使得调用进程在等待资源期间，其他进程能够访问管程，在管程条件变量所对应的wait操作过程中，需要调用V(mutex)，开放管程的使用，从而提高资源的使用，有利于资源的释放。



Hoare管程的数据结构(Cont.)

2. x_sem 和 x_count :实现管程条件变量wait和signal操作所需信号量

- ① 管程调用进程由于等待资源，会调用条件变量的wait操作，进入等待队列。
- ② x_sem 用于条件变量wait和signal所需的信号量（初值为0，从而当wait内部调用P操作即进入等待）
- ③ 由于 $x_sem < 0$ ，仍然可以将进程转等待，因此可能有多个进程等待资源，故需要一个计数器知道多少进程在等待，为 x_count （初值为0）。
- ④ 执行signal时候，需要让等待的某个进程立即恢复，而其他进程不允许进入，因此需要用到 $V(x_sem)$

Hoare管程的数据结构(Cont.)

3. next, next count:处理signal操作时, 唤醒其他进程, 而自身进入等待的信号量

- ①对于管程, 发出signal操作后, 立即通过V(x_sem)释放一个等待的进程, 并且使用P(next)将自己阻塞, 进入等待
- ②该进程直到被释放的进程退出管道时候, 被V(next)唤醒 (故Leave中有V(next)) ;
- ③或者该进程由于等待其他条件, 而进入等待时被V(next)唤醒 (故wait中有V(next)) 。
- ④进程在退出管道时候, 需要检查是否有其他进程由于signal而等待于next上, 若有则唤醒。因此, 需要一个计数器next_count来记录next上等待的进程数。



Hoare管程的定义

```
typedef struct InterfaceModule{
    semaphore mutex;           //调用管程互斥信号量
    semaphore next;            //signal操作等待信号量
    int next_count;            //next上等待进程数
};

mutex=1;next=0;next_count=0; //初始化语句
void enter(InterfaceModule &IM){
    P(IM.mutex);
}
void leave(InterfaceModule &IM){
    if (IM.next_count>0)
        V(IM.next);           //唤醒因为signal操作而等待的进程
    else
        V(IM.mutex);          //唤醒因访问管程而等待的进程
}
```



Hoare管程的定义 (Cont.)

```
void wait(semaphore &x_sem,int &x_count, Interface &IM){
    x_count++;
    if (IM.next_count>0)
        V(IM.next);          //自身等待，将因signal而休眠的进程释放一个
    else
        V(IM.mutex);          //否则开放管程
    P(x_sem);                  //自己阻塞自己
    x_count--;                 //将来被唤醒时候，执行等待资源数减少一个
}

void signal(semaphore &x_sem,int &x_count, Interface &IM){
    if (x_count>0)    //判断是否有等待资源的进程
    {
        IM.next_count++;          //因为signal休眠的进程数+1
        V(x_sem);                 //唤醒一个等待资源进程
        P(IM.next);               //因为发signal唤醒别的进程自己阻塞
        IM.next_count--;          //等将来被唤醒时候，因signal休眠数-1
    }
}
```



基于Hoare方法的哲学家问题解决

- 用三种不同状态表示哲学家的活动：进餐、饥饿、思考。（thinking, hungry, eating）state[5];
- 为每个哲学家设置一个条件变量self (i) , 当哲学家饥饿又不能获得筷子时，用self来阻塞自己：
- 管程设置三个函数：pickup取筷子，putdown放筷子，test测试是否具备进餐条件。



基于Hoare方法的哲学家问题解决

```
type dining_philosophers=monitor
    enum {thinking, hungry, eating} state[5]; //哲学家的状态
    semaphore self[5];                        //条件变量
    int self_count[5];                        //共享变量
    InterfaceModule IM;
    for (int i=0;i<5;i++) state[i] = thinking;
    define pickup,putdown;
    use enter, leave, wait, signal;

void pickup(int i){
    enter(IM);
    state[i]=hungry;
    test[i];
    if (state[i]!=eating);
        wait(self[i],self_count[i],IM);
    leave(IM);
}
```



基于Hoare方法的哲学家问题解决

```
void putdown(int i){  
    enter(IM);  
    state[i]=thinking;  
    test[(i-1)%5];  
    test[(i+1)%5];  
    leave(IM);  
}
```

```
void test (int k){  
    if (state[(k-1)%5]!=eating)&&(state[k]==hungry)  
        &&(state[(k+1)%5]!=eating)){  
        state[k] = eating;  
        signal(self[k],self_count[k],IM);  
    }  
}
```



基于Hoare方法的哲学家问题解决

```
main()
{
    cobegin
        philosopher(0);
        philosopher(1);
        philosopher(2);
        philosopher(3);
        philosopher(4);
    coend
}
```

```
void philosopher(int i)
{
    while(true)
    {
        Thinking;
        dining-philosophers.pickup(i);
        Eating;
        dining-philosophers.putdown(i);
    }
}
```



本章小节

- 临界资源访问的原则
- 同步与互斥的定义
- 互斥的实现方法（软件、硬件、锁）
- 信号量的定义
- 信号量解决经典问题
- 管程的基本概念



作业整理：小作业

1. V9, 6.3
2. 有P1、P2、P3三类进程（每类进程都有K个）共享一组表格F（F有N个），P1对F只读不写，P2对F只写不读，P3对F先读后写。进程可同时读同一个 F_i ($0 \leq i \leq N$)；但有进程写时，其他进程不能读和写。
 - 用信号量和P、V操作给出方案
 - 对方案的正确性进行分析说明
 - 对访问的公平性进行分析

小作业

- 3.三组进程P1 (K个, $K > 1$)、P2 (1个)、P3 (1个)、互斥使用一个包含N($N > 1$)个单元的缓冲区buf1, P2-1、P3-1、P4 (各有1个) 执行定期统计
 - P1每次用produce(), 生成一个正整数并用put()送入缓冲区buf1某一空单元中, 循环访问缓冲区
 - P2每次用getodd()从buf1缓冲区取出一个奇数, 放到自己私有缓冲区buf2中(缓冲区长度为 $M > 1$), 然后由P2-1 (1个) 进程读取P2的私有缓冲区, 使用countodd()统计奇数个数
 - P3每次用geteven()从buf1缓冲区中取出一个偶数, 放到自己私有缓冲区buf3中(缓冲区长度为 $M > 1$), 然后由P3-1 (1个) 进程读取P3的私有缓冲区, 使用counteven()统计偶数个数
 - P4在P2-1、P3-1各产生一个统计值后, 就输出一个包含统计时间的结果
 - 请用信号量机制实现这几个进程的同步与互斥活动, 并说明所定义的信号量含义。



小作业

■ 4. 独木桥问题

1. 独木桥只允许一台汽车过河，当车到达桥头时，如果桥上无车，则可上桥，否则车在桥头等待，直到桥上无车。使用PV操作解决该问题。
2. 如果独木桥允许多台车同方向过河，当车到达桥头时，如果桥上只有同方向车且不超过N台，或者桥上无车，则可上桥通过，否则等待，直到满足上桥条件。使用PV操作解决该问题。

■ 5. 红黑客过河问题

- 有红客和黑客两组人员需要过河。河上有船，但是每次只能乘坐4人，且每次乘客满员时才能开船，到河对岸后空船返回。但船上乘客不能同时有3个红客、1个黑客 或者 1个红客、3个黑客的组合。（其他组合安全）。请编写程序，用PV操作解决红客、黑客过河的问题。



大作业

1. 参考V9版，课后编程项目，针对读者/写者、生产者/消费者、哲学家问题、理发师（or睡觉助教），选择一个实现，要求使用Pthreads 和 Windows 线程API 实现