操作系统

第八讲

张涛

Review

线程的引入

进程和线程的比较

操作系统对线程的实现

线程举例

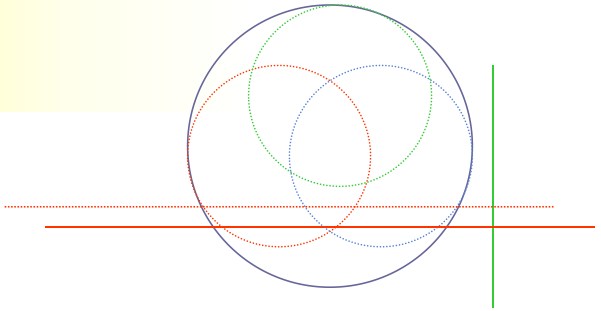
3.6 进程间通讯

进程同步和互斥

信号量和PV原语操作

经典进程同步问题

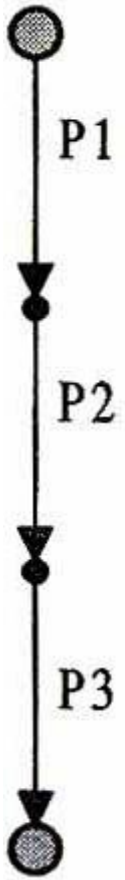
进程通信

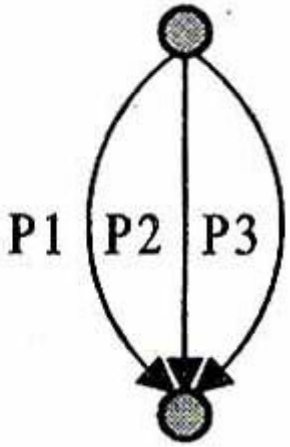


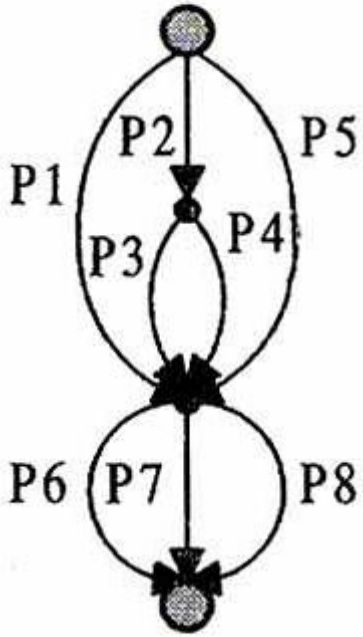
3.6.1 进程同步和互斥

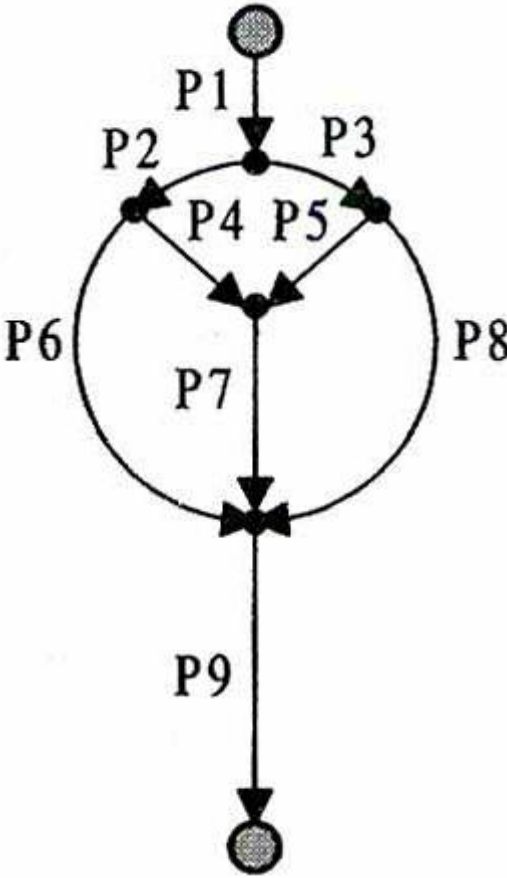
- 进程间的同步
- 进程间的互斥
- 临界资源及其访问过程
- 同步机制应遵循的准则
- 进程互斥的软件方法
- 进程互斥的硬件方法

进程间运行关系





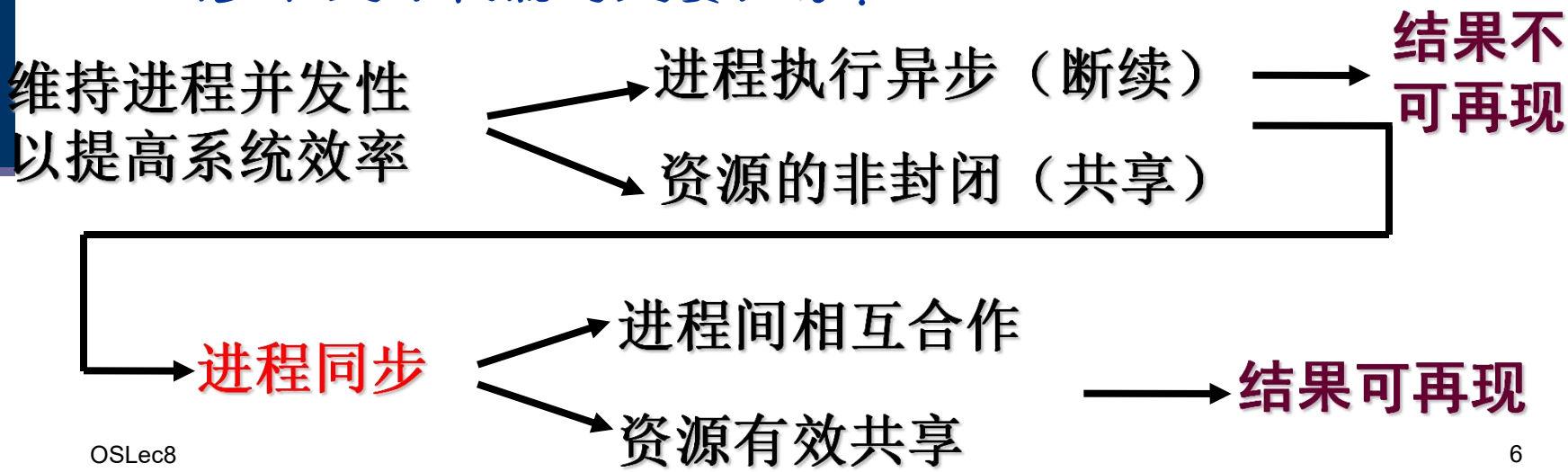




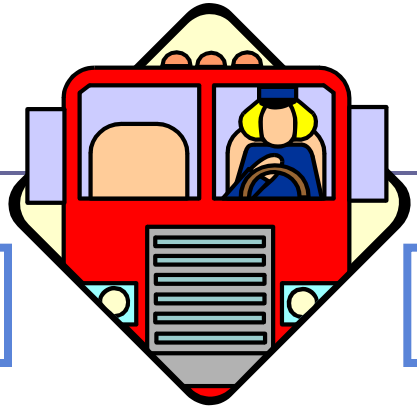
(a) (b) (c) (d)

进程间的制约关系

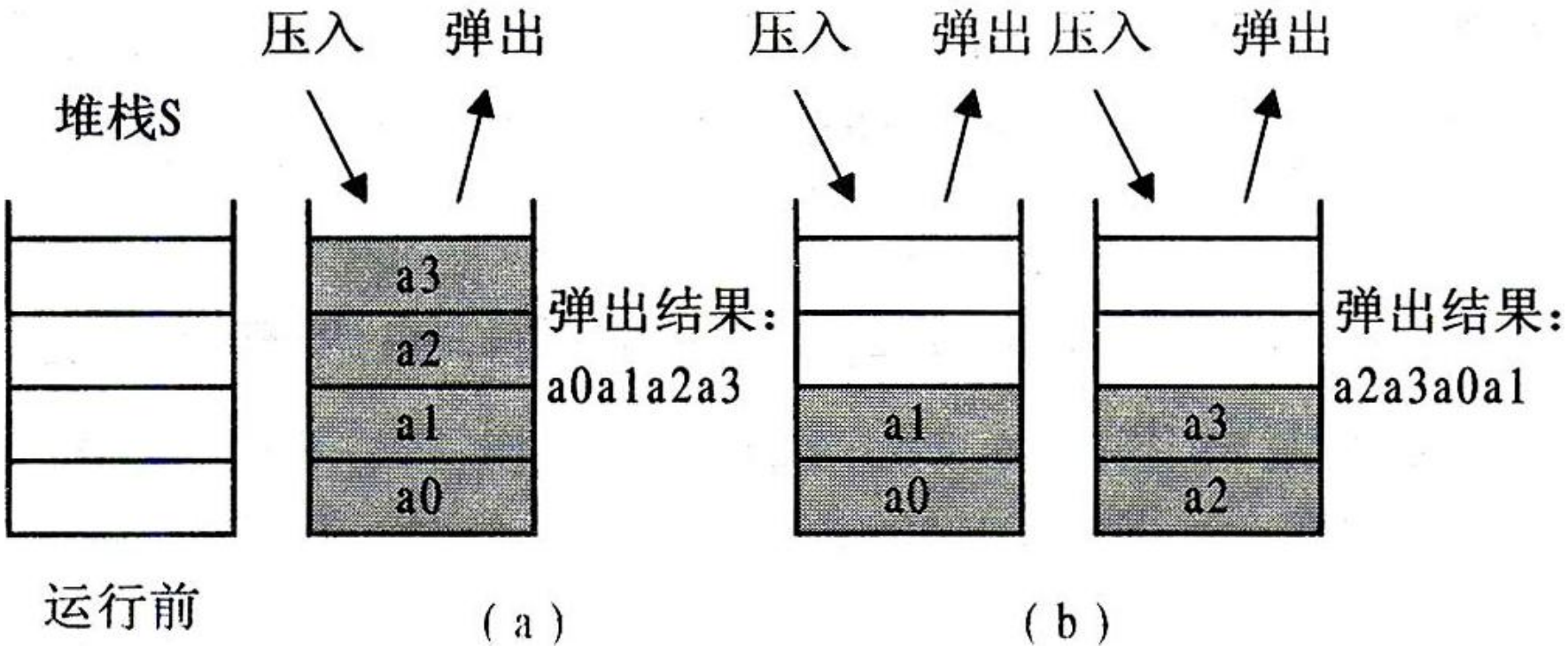
- 直接制约关系：进行协作——等待来自其他进程的信息，“同步”
- 间接制约关系：进行竞争——独占分配到的部分或全部共享资源，“互斥”
- 进程间这种相互依赖又相互制约，相互合作又相互竞争的关系表现为同步和互斥两个方面。
- 同步和互斥机制的主要任务：



同步



进程间的同步



进程A和进程B共用堆栈

- **进程同步**：指进程之间的一种协调配合关系，它表现在进程的**执行顺序**的规定上。
- **概念**：相互协调的几个进程在某些确定点上协调它们的工作，一个进程到达了这些点后，除非另一进程已完成了某些操作，否则就需要停下来等待这些操作的完成。

进程同步的传送消息实现

- ✓ 如果对一个事件或消息赋以唯一的消息名，则过程wait (消息名）表示进程等待合作进程发来消息，功能是等待到消息名为true的进程继续执行；
- ✓ 过程signal（消息名）表示向合作进程发送消息，功能则是向合作进程发送所需要的消息名，并将其值置为true。

例：计算进程和打印进程的同步关系.

- 设消息名bufempty表示buf空，设消息名buffull表示buf满.

初始化 bufempty =true,　　Buffull=false.

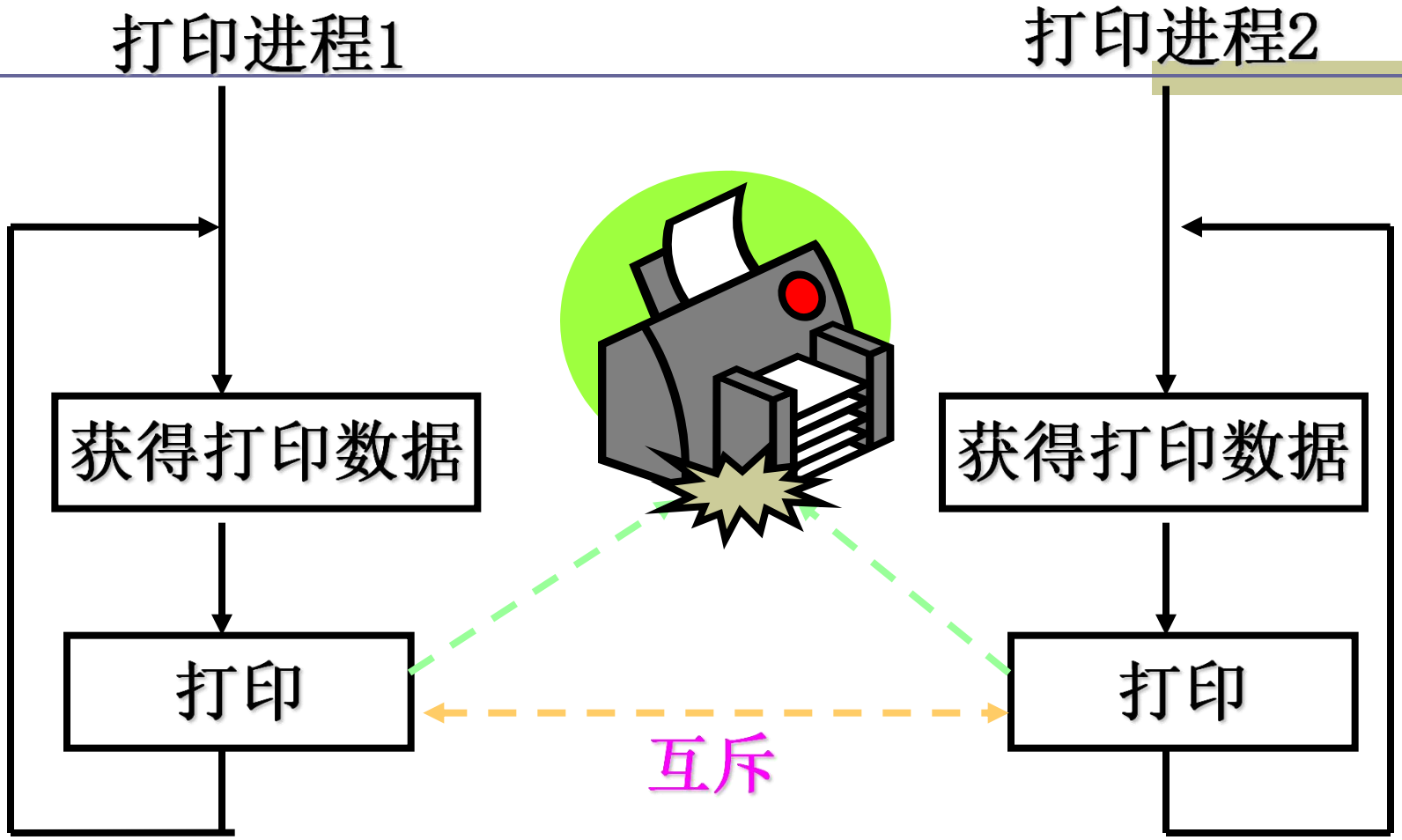
Pc:

```
while(true){
        计算
        wait(bufempty)
        buf←计算结果
        bufempty ← false
        signal(buffull)}
```

Pp:

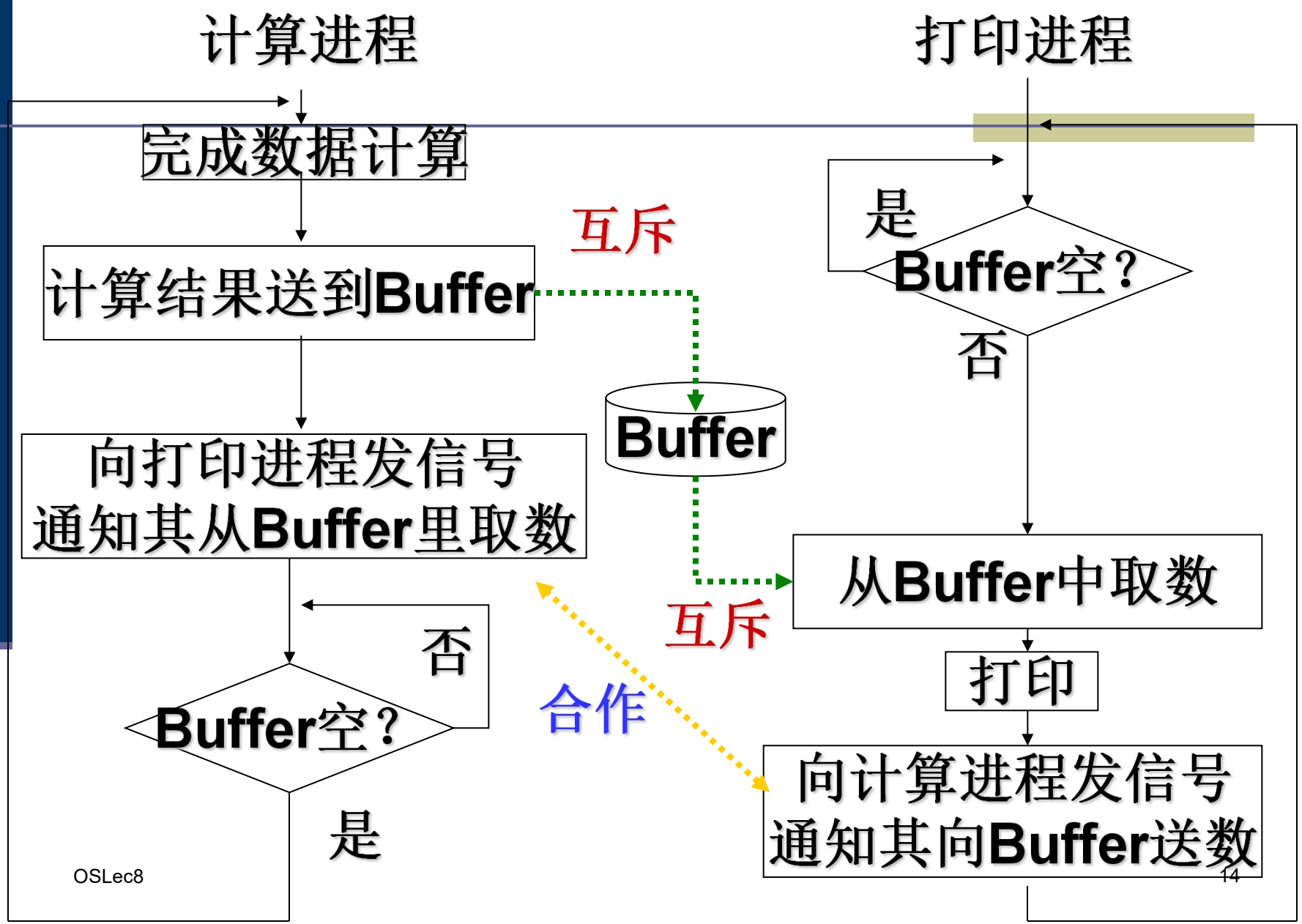
```
while(true){
        wait(buffull)
        打印Buf中的数据
        清除Buf中的数据
        buffull ←false
        signal(bufempty)}
```

互斥



进程间的互斥

- 进程互斥：两个或两个以上的进程由于不能同时使用同一资源，只能一个进程使用完了另一个进程才能使用的现象。
- 互斥关系也是一种协调关系，从广义上讲它也属于同步关系的范畴。


计算进程
打印进程
完成数据计算
计算结果送到Buffer
向打印进程发信号
通知其从Buffer里取数
Buffer空？
否
是
互斥
Buffer
互斥
合作
是
Buffer空？
否
从Buffer中取数
打印
向计算进程发信号
通知其向Buffer送数

进程同步时面临的两种主要关系

相互合作 ← 司机与售票员

竞争资源 ← 计算者与打印者

竞争资源 ← 多个打印者

事件、设备等抽象为资源

对进程间关系的处理变为对资源的访问方式

临界资源与临界区

- 临界资源(critical resource)：一次仅允许一个进程访问的资源。
  - 一次只允许一个进程访问的资源
  - 资源状态为临界：0 或 1
- 临界区(critical section)：临界段，在每个程序中，访问临界资源的那段程序。
- 对临界段的设计有如下原则：
  - 每次至多只允许一个进程处于临界段中。
  - 对于请求进入临界段的多个进程，在有限时间内只让一个进入。
  - 进程只应在临界段中停留有限时间。

进程A和进程B在各自的执行过程中，都需要使用变量M作为其中间变量，它们的程序如下：

进程A：

```
X: =1;
Y: =2;
M: =X;
X: =Y;
Y: =M;
PRINT ("A:", X,Y);
```

进程B：

```
K: =3;
L: =4;
M: =K;
K: =L;
L: =M;
PRINT ("B:", K,L);
```

```
进程A:    X: =1;
进程B:    K: =3;
进程A:    Y: =2;
进程B:    L: =4;
进程A:    M: =X;
进程B:    M: =K;
进程A:    X: =Y;
进程B:    K: =L;
进程A:    Y: =M;
进程B:    L: =M;
进程A:    PRINT ("A: ", X,Y) ;
进程B:    PRINT ("B: ", K,L) ;
```

临界区的访问过程

entry section

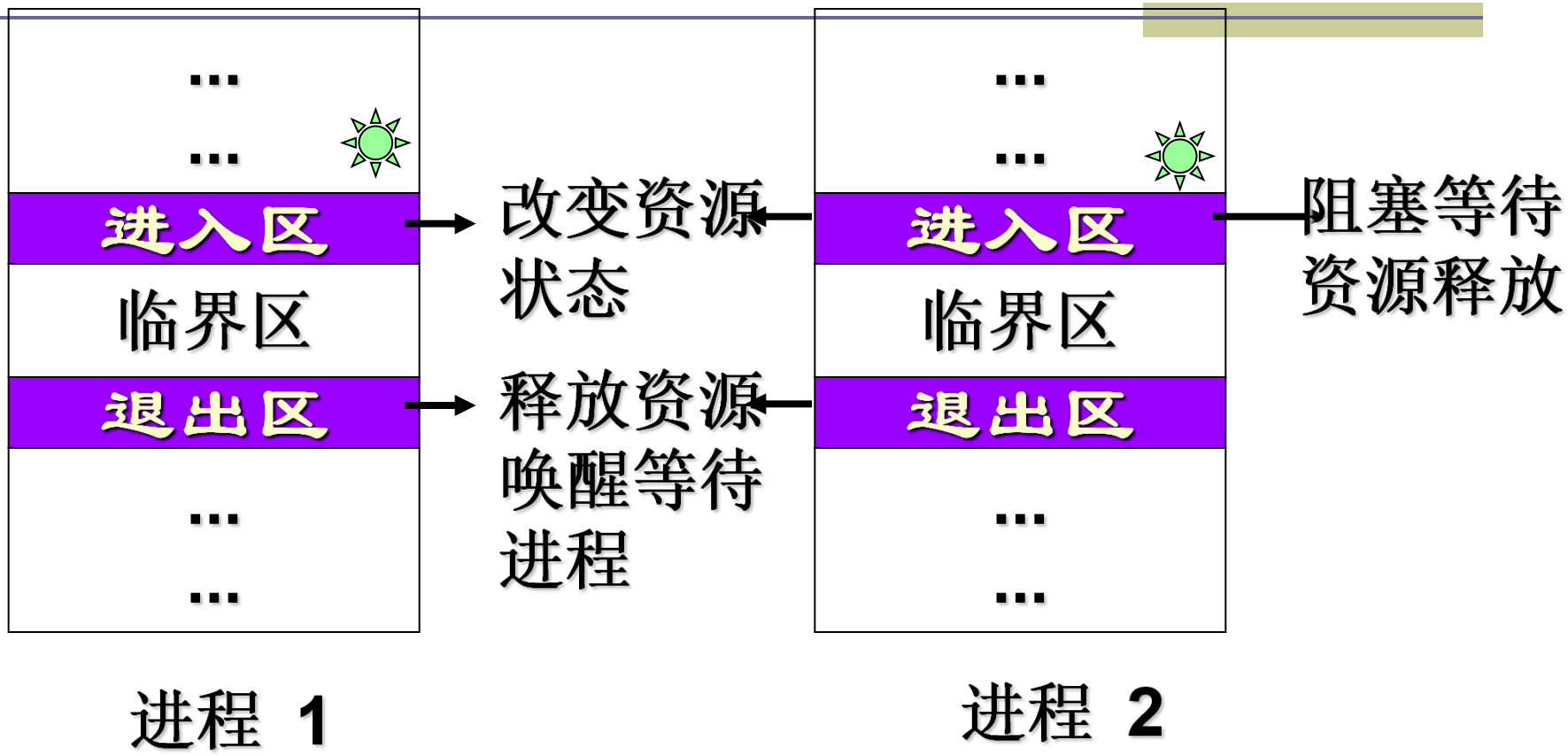
critical section

exit section

remainder section

临界区

- 临界区(critical section)：进程中访问临界资源的一段代码。
- 进入区(entry section)：在进入临界区之前，检查可否进入临界区的一段代码。如果可以进入临界区，通常设置相应"正在访问临界区"标志
- 退出区(exit section)：用于将"正在访问临界区"标志清除。
- 剩余区(remainder section)：代码中的其余部分。


...
...
进入区
临界区
退出区
...
...
改变资源状态
释放资源
唤醒等待进程
...
...
进入区
临界区
退出区
...
...
阻塞等待
资源释放
进程 1
进程 2

同步机制应遵循的准则

空闲让进

忙则等待

有限等待

让权等待

软件方法解决互斥

- 特点
  - 无需硬件、OS和程序设计语言的支持
  - 处理开销大，容易出错
- 学习的意义
  - 更好地理解并发处理的复杂性
- 适用范围
  - 单处理器系统
  - 共享主存的多处理系统
  - 前提假设：存储器级的访问是互斥的

Dekker算法

- 尝试1：单标志
- 设整型变量turn，用于指示被允许进入临界区的进程的编号，即若turn=0，表示进程Pi可进入。turn =1表示进程Pj可进入。

进程Pi：

```
Repeat
  While turn<>i do no_op;

  Critical section
  turn:=j;

  Other code
Until false;
```

算法1的问题

该算法可确保每次只允许一个进程进入临界区。但它强制两个进程轮流进入。如当Pi退出时将turn置为1，以便 Pj能进入，但Pj暂不需要进入，而这时Pi又需要进入时，它无法进入。这不能保证准则1。

- 尝试2：双标志、先检查

设var flag:array[0..1] of boolean，若flag[i]=true，表示进程Pi正在临界区内。flag[i]=false表示进程Pi不在临界区内。若flag[j]=true，表示进程Pj正在临界区内。flag[j]=false表示进程Pj不在临界区内。

```
Pi进程：
Repeat
   While flag[ j ] do no_op;

   flag[i]:=true;
   Critical section
   flag[i]:=false;

   Other code
Until false;
```

算法2的问题

该算法可确保准则1。但又出现新问题。当pi和pj都未进入时，它们各自的访问标志都为false。如果pi和pj几乎同时要求进入，它们都发现对方的标志为false，于是都进入了。这不能保证准则2。

```
while (flag[j]);                <a>
flag[i] = TRUE;                 <b>
  critical section
flag[i] = FALSE;
  remainder section
```

- 尝试3：双标志、后检查
- 在算法3 中，设var Flag:array[0..1] of boolean，若flag[i]=true，表示进程Pi希望进入临界区内。若flag[j]=true，表示进程Pj希望进入临界区。

```
Pi进程：
Repeat
    flag[i]:=true;
    While flag[j] do no_op;

    Critical section
    flag[I]:=false;

    Other code
Until false;
```

算法3的问题

该算法可确保准则2。但又出现新问题。它可能造成谁也不能进入。如，当pi和pj几乎同时都要进入，分别把自己的标志置为true，都立即检查对方的标志，发现对方为true. 最终谁也不能进入。这不能保证准则1。

```
flag[i] = TRUE;              <b>
while (flag[j]);             <a>
  critical section
flag[i] = FALSE;
  remainder section
```

算法4（Peterson算法）

- 组合算法1、3，为每一进程设标志位flag[i]，当flag[i]=true时，表示进程pi要求进入，或正在临界区中执行。此外再设一个变量turn，用于指示允许进入的进程编号。
- 先修改、后检查、后修改者等待：
- 进程为了进入先置flag[i]=true，并置turn为j，表示应轮到pj进入。接着再判断flag[j] and turn=j的条件是否满足。若不满足则可进入。或者说，当flag[j]=false或者turn=i时，进程可以进入。前者表示pj未要求进入，后者表示仅允许pi进入。

算法4

Repeat

flag[i]:=true;

turn:=j;

While (flag[j] and turn=j) do
no_op;

Critical section

flag[i]:=false;

Other code

Until false

```
flag[i] = TRUE; turn = j;
while (flag[j] && turn == j);
```

critical section

```
flag[i] = FALSE;
```

remainder section

软件解法的缺点

1. 忙等待
2. 实现过于复杂，需要较高的编程技巧

硬件方法解决进程互斥

- 1. 中断禁用 (关中断, Interrupt Disabling)
  - 一个进程一直运行，直到调用了一个OS服务或被中断如果进程访问临界资源时(执行临界区代码)不被中断，就可以利用它来保证互斥地访问
  - 途径：使用关中断原语、开中断原语
  - 过程：
    关中断原语；
    临界区
    开中断原语
    其余部分
  - 存在问题
    - 代价高：限制了处理器交替执行各进程的能力
    - 不能用于多处理器结构：关中断不能保证互斥

硬件方法解决进程互斥

- 2.专门的机器指令
  - 设计一些机器指令，用于保证两个动作的原子性，如在一个指令周期中实现测试和修改
    - TestandSet指令
    - Swap指令

Test-and-Set指令实现互斥

1、Test-and-Set指令

```
Function TS(var lock:boolean):boolean;
Begin
   TS:=lock;
   Lock:=true;
End;
```

其中，有lock有两种状态：当lock=false时，表示该资源空闲；当lock=true时，表示该资源正在被使用。

2、利用TS指令实现进程互斥

为每个临界资源设置一个全局布尔变量lock，并赋初值false，表示资源空闲。在进入区利用TS进行检查：有进程在临界区时，重复检查；直到其它进程退出时，检查通过；

```
repeat
   while TS(lock) do skip;
   critical section
   lock:=false;
   Other code
Until false;
```

```
while TS(&lock);
    critical section
lock = FALSE;
    remainder section
```

swap指令实现进程互斥

1、swap指令，又称交换指令，X86中称为XCHG指令。

```
Procedure swap(var a,b:boolean);
Var temp:boolean;
Begin
  Temp:=a;      A:=b;      B:=temp;
End;
```

2、利用swap实现进程互斥

为每一临界资源设置一个全局布尔变量lock，其初值为false，在每个进程中有局部布尔变量key。

```
Repeat
   key:=true;
   Repeat      Swap(lock,key);     Until key=false;
   Critical section
   lock:=false;
   Other code
Until false;
```

硬件方法的优点

- 适用于任意数目的进程，在单处理器或多处理器上
- 简单，容易验证其正确性
- 可以支持进程内存在多个临界区，只需为每个临界区设立一个布尔变量

硬件方法的缺点

- 等待要耗费CPU时间，不能实现"让权等待"
- 可能"饥饿"：从等待进程中随机选择一个进入临界区，有的进程可能一直选不上
- 可能死锁

用锁实现进程互斥

- 锁即操作系统中的一标志位，0 表示资源可用，1 表示资源已被占用。用户程序不能对锁直接操作，必须通过操作系统提供的上锁和开锁原语来操作。
- 通常锁用w表示，上锁开锁原语分别用lock(w)、unlock(w)来表示。

上锁和开锁原语

- 上锁原语lock(w)可描述为：

```
L：  if(w==1) goto L
     else w=1；
```

- 开锁原语unlock(w)可描述为：

```
w=0；
```

用原语实现进程互斥

```
pa ( )
{
    :
    lock (w);
    csa;
    unlock (w);
    :
}
```

```
pb ( )
{
    :
    lock (w);
    csb;
    unlock (w);
    :
}
```

进程的交互关系：可以按照相互感知的程度来分类

相互感知的程度	交互关系	一个进程对其他进程的影响	潜在的控制问题
相互不感知(完全不了解其它进程的存在)	竞争(competition)	一个进程的操作对其他进程的结果无影响	互斥，死锁（可释放的资源），饥饿
间接感知(双方都与第三方交互，如共享资源)	通过共享进行协作	一个进程的结果依赖于从其他进程获得的信息	互斥，死锁（可释放的资源），饥饿，数据一致性
直接感知(双方直接交互，如通信)	通过通信进行协作	一个进程的结果依赖于从其他进程获得的信息	死锁，饥饿

互斥，指多个进程不能同时使用同一个资源；
死锁，指多个进程互不相让，都得不到足够的资源；
饥饿，指一个进程一直得不到资源（其他进程可能轮流占用资源）

What you need to do?

- 复习、预习课本3.6节的内容
- 作业：P91页10,11题

See you next time!