上节课重点：

1. 进程的三大状态：就绪态、执行态、阻塞态
   进程的五大状态：执行、活动就绪、静止就绪、活动阻塞、
   静止阻塞
   进程的七大状态：创建、运行态、活动就绪、静止就绪、
   活动阻塞、静止阻塞、终止
2. 进程同步的制约关系：间接相互制约、直接相互制约
3. 临界资源和临界区：访问临界资源的代码称为临界区。
4. 同步机制遵循的规则：空闲让进、忙则等待、有限等待和
   让权等待。
5. 临界区资源共享算法：单标志法 (违背空闲让进)
   双标志先检查法 (违反忙则等待)
   双标志后检查法 (违背空闲让进
   和有限等待)

- 为了实现对临界资源的互斥访问，同时保证系统整体性能，同步机制应遵循的准则：
  - **空闲则入**：其他进程均不处于临界区，应允许请求进入临界区的进程进入；
  - **忙则等待**：已有进程处于其临界区，请求进入临界区的进程应等待；
  - **有限等待**：等待进入临界区的进程不能“死等”；
  - **让权等待**：不能进入临界区的进程，应释放 CPU（如转换到阻塞状态）

```
do{
    entry section;   // 进入区
    critical section;   // 临界区
    exit  section;      // 退出区
    remainder section;     // 剩余区
}  while (true)
```

为解决临界区问题的最初尝试

- 只有 2 个进程： P_0 和 P_1
- 进程 P_i （或 P_j） 通常的结构：

```
do {
    进入区( entry section ) ;
   临界区( critical section ) ;
    退出区( exit section ) ;
    剩余区( reminder section ) ;
        } while (1);
```

- 进程可以通过共享某些公共变量来同步他们的活动。

算法 1

- **共享变量：**
  - int turn ；
    初值： turn = i;
  - **如果** turn = i $\Rightarrow$ P_i **能进入他的临界区**

- 进程 P_i　　进程 P_j

```
do {
    while (turn != i) ;
        临界区 ;
    turn = j;
        剩余区 ;
} while (1);
```

```
do {
    while (turn != j) ;
        临界区 ;
    turn = i ;
        剩余区 ;
} while (1);
```

- 可以**确保互斥**，但是不能做到**空闲让进**

算法 2

- **共享变量：**
  - boolean flag[2];
    初值： flag [i] = flag [j] = false.
  - **如果** flag [i] = true $\Rightarrow$ P_i **准备进入临界区**

进程 P_i

```
do {
    flag[i] = true;
    while (flag[ j]) ;
        临界区 ;
    flag [i] = false;
        剩余区 ;
    } while (1);
```

进程 P_j

```
do {
    flag[ j ] = true;
    while (flag[ i ]) ;
        临界区 ;
    flag [ j ] = false;
        剩余区 ;
    } while (1);
```

- 可以确保互斥，但是不能保证有限等待！

算法 3

- 综合算法 1 和算法 2
- flag[i]=true 表示进程 Pi 想进入临界区
- turn = i 表示轮到 Pi 进程进入临界区
- 初始 flag [i] = flag [j] = false

算法 3

Pi 进程：

```
do {
    flag [i] = true;
    turn = j;
    while(flag [j]
       && turn==j) ;
        临界区；
    flag [i] = false;
        剩余区；
  } while (1);
```

Pj 进程：

```
do {
    flag [j] = true;
    turn = i;
    while (flag [i]
       && turn == i) ;
        临界区；
    flag [j] = false;
        剩余区；
  } while (1);
```

- **满足上述解决两个进程的临界区问题的三个原则。**
- **只要采用 FCFS 调度算法，即可保证满足有空让进和有限等待**

算法 4: 单标志法

算法思想：两个进程在访问临界区后会把使用临界区的权限转交给另一个进程。也就是每个进程进入临界区的权限只能被另一个进程赋予。

```
int turn = 0; //turn 表示当前允许进入临界区的进程号

P0 进程:                                P1进程:
while (turn != 0);          ①           while (turn != 1);          ⑤ //进入区
critical section;           ②           critical section;           ⑥ //临界区
turn = 1;                   ③           turn = 0;                   ⑦ //退出区
remainder section;          ④           remainder section;          ⑧ //剩余区
```

turn 变量背后的逻辑：表达“谦让”

turn 的初值为 0 ，即刚开始只允许 0 号进程进入临界区。
若 P1 先上处理机运行，则会一直卡在⑤。直到 P1 的时间片用完，发生调度，切换 P0 上处理机运行。
代码①不会卡住 P0 ， P0 可以正常访问临界区，在 P0 访问临界区②期间即时切换回 P1 ， P1 仍然会卡在⑤。
只有 P0 在退出区③将 turn 改为 1 后， P1 才能进入临界区。

因此，该算法可以实现“同一时刻最多只允许一个进程访问临界区”。

算法 4: 单标志法

```
int turn = 0; //turn 表示当前允许进入临界区的进程号
```

turn 变量背后的逻辑：表达“谦让”

```
P0 进程:                          P1进程:
while (turn != 0);        ①       while (turn != 1);        ⑤ //进入区
critical section;         ②       critical section;         ⑥ //临界区
turn = 1;                 ③       turn = 0;                 ⑦ //退出区
remainder section;        ④       remainder section;        ⑧ //剩余区
```

只能按照 P0-->P1-->P0-->P1... 这样轮流访问。这种必须“轮流访问”带来的问题是：如果 turn 的初值为 0 ，刚开始 P1 先上处理机运行，则会一直卡在⑤，此时允许进入临界区的进程是 P0 ，而 P0 一直不访问临界区，那么虽然此时临界区空闲，但是并不允许 P1 访问。

因此，**单标志法存在的主要问题是：违背“空闲让进”原则。**

算法 5 双标志先检查法

算法思想：设置一个布尔型数组 flag[] ，数组中各个元素用来标记各进程想进入临界区的意愿，比如“ flag[0]=true” 意味着 0 号进程 P0 现在想要进入临界区。每个进程在进入临界区之前，先检查当前有没有别的进程想进入临界区，如果没有，则把自身对应的标志 flag[i] 设置为 true ，之后开始访问临界区。

```
bool flag[2];          //表示进入临界区意愿的数组
flag[0] = false;
flag[1] = false;       //刚开始设置为两个进程都不想进入临界区

P0 进程:                        P1 进程:
while (flag[1]);    ①           while (flag[0]);    ⑤   //如果此时 P0 想进入临界区，P1 就一直循环等待
flag[0] = true;     ②           flag[1] = true;     ⑥   //标记为 P1 进程想要进入临界区
critical section;   ③           critical section;   ⑦   //访问临界区
flag[0] = false;    ④           flag[1] = false;    ⑧   //访问完临界区，修改标记为 P1 不想使用临界区
remainder section;              remainder section;
```

理解背后的含义：“表达意愿”

进入区

若按照①⑤②⑥③⑦ ... 的顺序执行， P0 和 P1 将会同时访问临界区。
因此，双标志先检查法的主要问题是：违反”忙则等待”原则。
原因在于，进入区的“检查”和“上锁”两个处理不是一气呵成的。
“ 检查”后，“上锁”前可能发生进程切换。

算法 6 双标志后检查法

算法思想：双标志先检查法的改版。前一个算法的问题是先“检查”后“上锁”，但是这两个操作又无法一气呵成，因此导致了两个进程同时进入临界区的问题。因此，人们又想到先“上锁”后“检查”的方法，来避免上述问题。

```
bool flag[2];          //表示进入临界区意愿的数组
flag[0] = false;
flag[1] = false;       //刚开始设置为两个进程都不想进入临界区

P0 进程:                          P1 进程:
flag[0] = true;       ①          flag[1] = true;       ⑤   //标记为 P1 进程想要进入临界区
while (flag[1]);      ②          while (flag[0]);      ⑥   //如果 P0 也想进入临界区，则 P1 循环等待
critical section;     ③          critical section;     ⑦   //访问临界区
flag[0] = false;      ④          flag[1] = false;      ⑧   //访问完临界区，修改标记为 P1 不想使用临界区
remainder section;                remainder section;
```

理解背后的含义：“表达意愿”

进入区

若按照①⑤②⑥ ... 的顺序执行， P0 和 P1 将都无法进入临界区。
因此，双标志后监察法虽然解决了“忙则等待”的问题，但是又违背了“空闲让进”和“有限等待”原则，会因各进程都长期无法访问临界资源而产生“饥饿”现象。
两个进程都想争着进入临界区，但是谁都不让谁，最后谁都无法进入临界区。

算法 7 Peterson 算法

算法思想：结合双标志法、单标志法的思想。如果双方都争着想进入临界区，那可以让进程尝试“孔融让梨”(谦让)。做一个有礼貌的进程。

```
bool flag[2];          //表示进入临界区意愿的数组，初始值都是false
int turn = 0;          //turn 表示优先让哪个进程进入临界区
P0 进程:
flag[0] = true;                    ①
turn = 1;                          ②
while (flag[1] && turn==1);        ③
critical section;                  ④
flag[0] = false;                   ⑤
remainder section;
P1 进程:
flag[1] = true;                    ⑥ //表示自己想进入临界区
turn = 0;                          ⑦ //可以优先让对方进入临界区
while (flag[0] && turn==0);        ⑧ //对方想进，且最后一次是自己“让梨”，那自己就循环等待
critical section;                  ⑨
flag[1] = false;                   ⑩ //访问完临界区，表示自己已经不想访问临界区了
remainder section;
```

背后的含义：“表达意愿”

表达“谦让”

进入区：1. 主动争取；2. 主动谦让；3. 检查对方是否也想使用，且最后一次是不是自己说了“客气话”

谁最后说了“客气话”，谁就失去了行动的优先权。

Eg: 过年了，某阿姨给你发压岁钱。

场景一

阿姨：乖，收下阿姨的心意~

你：不用了阿姨，您的心意我领了

阿姨：对阿姨来说你还是个孩子，你就收下吧

结局...

动手推导：

按不同的顺序穿插执行会发生什么？

①②③⑥⑦⑧...

①⑥②③...

①③⑥⑦⑧...

①⑥②⑦⑧...

面包店算法

解决 n 个进程的临界区问题

- **在进入临界区之前，进程试图去获取一个排队号码，持有最小号码的进程进入他的临界区。**
- **如果进程 P_i 和 P_j 得到的是同样的号码，如果 $i < j$，那么进程 P_i 先进入临界区，否则进程 P_j 进入临界区。**
- **系统生成的后一个号码总是大于等于前一个。例如：1,2,3,3,3,3,4,5...**

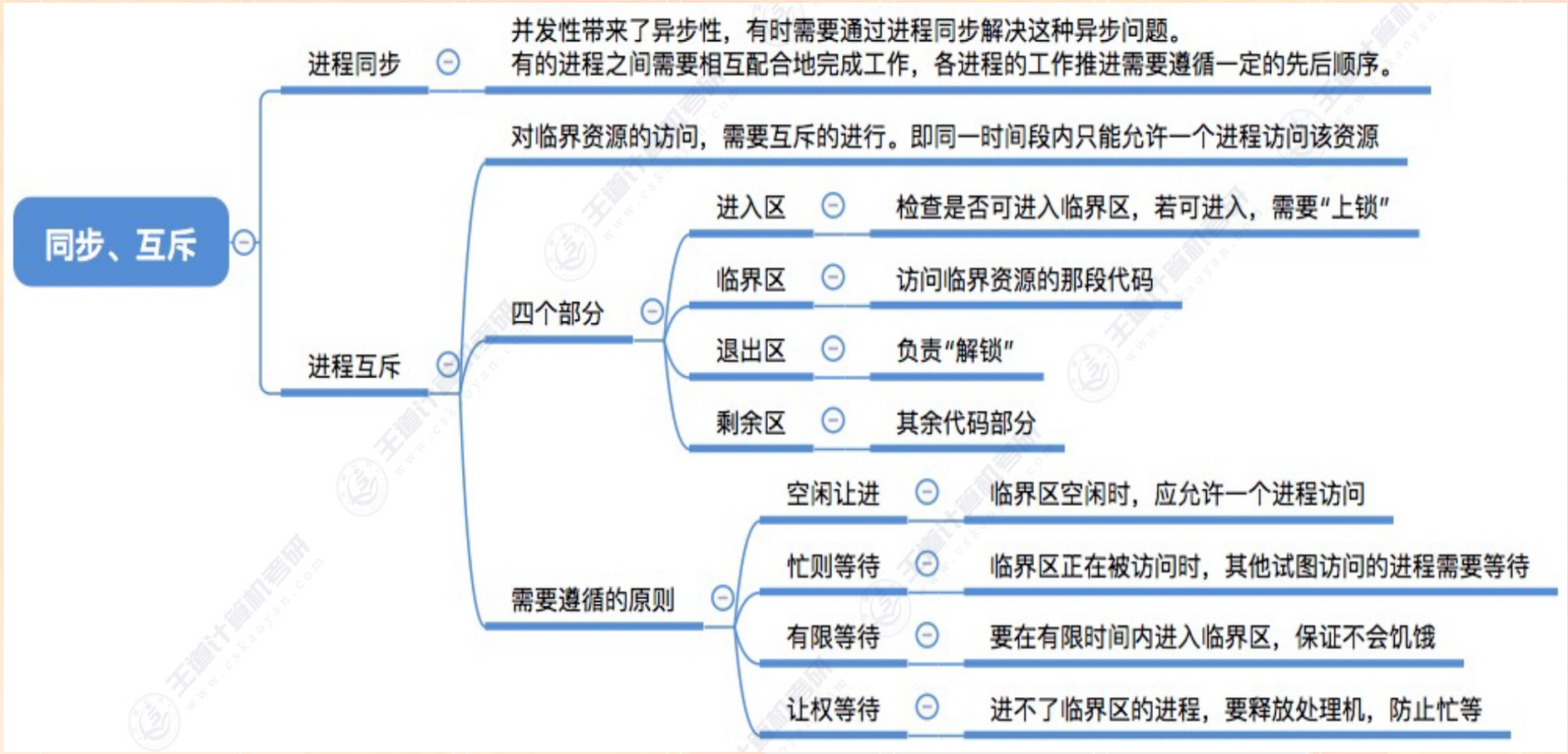
- 符号 < 表示按字典顺序排序关系（序号#, 进程 id #）
  - $(a,b) < (c,d)$ 如果 $a < c$ 或者 $a = c$ 且 $b < d$
  - $\max(a_0, \ldots, a_{n-1})$ 得到整数 k, 从而 $k \geq a_i$, $i = 0, \ldots, n-1$

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

任一个进程 Pi 的代码段如下：

```
do {
    choosing[i] = true;
    number[i] = max ( number[0], number[1], …, number
    [n – 1] )+1;
    choosing[i] = false;
    for (j = 0; j < n; j++) {     // 遍历数组
      while (choosing[j]);
      //  判断进程是否抽号，若是则等待该进程抽完号再比较
    while ((number[j]!=0)&&(number[j],j)<(number[i],i));
     //  判断自己的号是否最小，若是则继续直到遍历完成后进入
    临界区；否则等待比自己号小的进程处理完毕。
    }
    临界区
    number[i] = 0;     // 进程处理完后，将自己的号清零
    剩余区
} while (1);
```

进程同步与进程互斥总结



2.4.2 进程互斥的硬件实现方法

- **中断屏蔽方法**

利用“开 / 关中断指令”实现（与原语的实现思想相同，即在某进程开始访问临界区到结束访问临界区为止。都不允许中断，也就不可能发生进程切换。因此，也不可能发生两个进程，同时访问临界区的情况）

```
...
关中断；
临界区；
开中断；
...
```

关中断后即不允许当前进程被中断，也必然不会发生进程切换

直到当前进程访问完临界区，再执行开中断指令，才有可能有别的进程上处理机并访问临界区。

优点：简洁、高效
缺点：不适用于多处理机；只适用于操作系统内核进程，不适用于用户进程（因为开 / 关中断指令只能运行在内核态，这组指令如果能让用户随意使用会很危险）

2.4.2 进程互斥的硬件实现方法

- **TestAndSet 指令**

简称 TS 指令，有的地方也成为 TestAndSetLock 指令，或 TSL 指令。TSL 指令是用硬件实现的，执行的过程不允许被中断，只能一气呵成。以下是用 C 语言描述的逻辑。

```
//布尔型共享变量 lock 表示当前临界区是否被加锁
//true 表示已加锁，false 表示未加锁
bool TestAndSet (bool *lock){
  bool old;
  old = *lock;  //old用来存放lock 原来的值
  *lock = true; //无论之前是否已加锁，都将lock设为true
  return old;   //返回lock原来的值
}
```

```
//以下是使用 TSL 指令实现互斥的算法逻辑
while (TestAndSet (&lock)); //"上锁"并"检查"
临界区代码段...
lock = false;      //"解锁"
剩余区代码段...
```

若刚开始 lock 为 false ，则 TSL 返回的 old 值为 false ， while 循环条件不满足，直接跳过循环，进入临界区。若刚开始 lock 为 true ，则执行 TLS 后 old 返回的值是 true ， while 循环条件满足，会一直循环，直到当前访问临界区的进程在退出区进行“解锁”。

相比软件实现方法， TSL 指令把“上锁”和“检查”操作用硬件的方式变成了一气呵成的原子操作。优点：实现简单，无需像软件实现方法那样严格检查是否有逻辑漏洞；适用于多处理机环境。缺点：不满足“让权等待”原则，暂时无法进入临界区的进程会占用 CPU 并循环执行 TSL 指令，从而导致“忙等”。

2.4.2 进程互斥的硬件实现方法

- **Swap 指令**

有的地方也叫做 Exchange 指令，或简称 XCHG 指令。

Swap 指令是用硬件实现的，执行的过程不允许被中断，只能一气呵成。以下是用 C 语言描述的逻辑：

```
//Swap 指令的作用是交换两个变量的值
Swap (bool *a, bool *b) {
  bool temp;
  temp = *a;
  *a = *b;
  *b = temp;
}
```

```
//以下是用 Swap 指令实现互斥的算法逻辑
//lock 表示当前临界区是否被加锁
bool old = true;
while (old == true)
  Swap (&lock, &old);
临界区代码段...
lock = false;
剩余区代码段...
```

逻辑上来看 Swap 和 TSL 并无太大区别，都是先记录下此时临界区是否已经被上锁（通过 Swap 指令，记录在 old 变量上），再将上锁标记 lock 设置为 true ，最后检查 old 变量，如果 old 为 false 则说明之前没有别的进程对临界区上锁，则可跳出循环、进入临界区。

优点：实现简单，无需像软件实现方法那样严格检查是否会有逻辑漏洞；适用于多处理机环境。缺点：不满足“让权等待”原则，暂时无法进入临界区的进程会占用 CPU 并循环执行 Swap 指令，从而导致“忙等”。

2.4.2 进程互斥的硬件实现方法

2.4.3 信号量机制 (semaphore)

- **1965 年，荷兰学者 Dijkstra 提出了一种卓有成效的实现进程互斥、同步的方法 -- 信号量机制。**
- **信号量是 OS 提供的管理公有资源的有效手段。**
- **整型信号量或记录型信号量可以代表可用资源实体的数量。**

1. 整型信号量机制

- 1965 年，荷兰学者 Dijkstra 提出（所以 P 、 V 分别是荷兰语的 test(proberen) 和 increment(verhogen) ），是一种卓有成效的进程同步机制。
- 最初 Dijkstra 把信号量定义为整型量 s 和两个原子操作（除初始化操作）： P 和 V, 现又称为： wait 和 signal 。

- **Wait（ S ）：**
  while S<=0 do no-op;
  S:=S-1;

- **Signal（ S ）：**
  S:=S+1;

- **wait（ s ）和 signal（ S ）是原子操作**
- **只要信号量 S≤0 就不断测试，不满足让权等待**

信号量机制——整型信号量

用一个整数型的变量作为信号量，用来表示系统中某种资源的数量。

Eg：某计算机系统中有一台打印机...

与普通整数变量的区别：对信号量的操作只有三种，即 初始化、P操作、V操作

```
int S = 1;  //初始化整型信号量s，表示当前系统中可用的打印机资源数

void wait (int S) {    //wait 原语，相当于"进入区"
  while (S <= 0);      //如果资源数不够，就一直循环等待
  S=S-1;               //如果资源数够，则占用一个资源
}

void signal (int S) { //signal 原语，相当于"退出区"
  S=S+1;          //使用完资源后，在退出区释放资源
}
```

"检查"和"上锁"一气呵成，避免了并发、异步导致的问题

存在的问题：不满足"让权等待"原则，会发生"忙等"

```
进程P0:
...
wait(S);          //进入区，申请资源
使用打印机资源...  //临界区，访问资源
signal(S);        //退出区，释放资源
...
```

```
进程P1:
...
wait(S);
使用打印机资源...
signal(S);
...
```

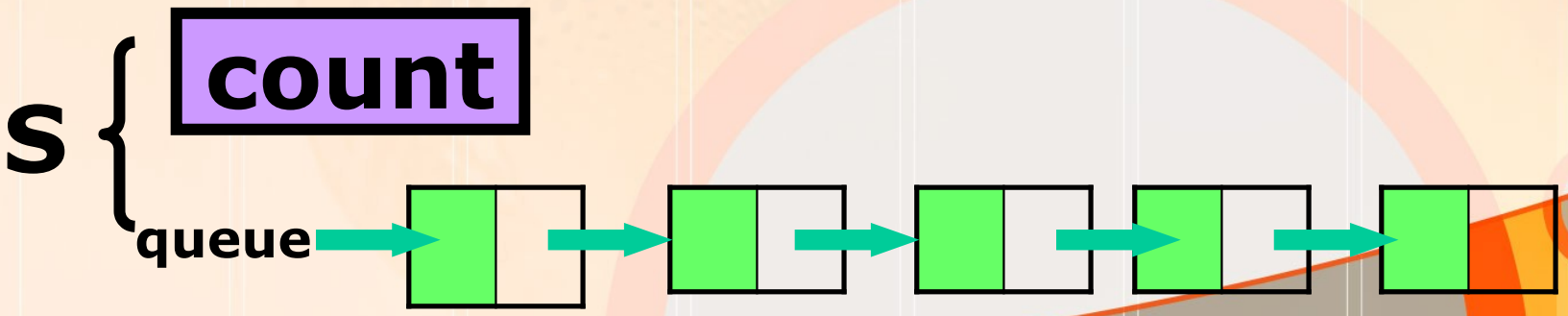
.....

```
进程Pn:
...
wait(S);
使用打印机资源...
signal(S);
...
```

- **2. 记录型信号量机制**

```
typedef struct {
        int count;
        struct  process *queue;
  } semaphore;
```

- 每个信号量 s
  - 一个整数值 s.count ：其初值表示某类资源的数目（又称为“资源信号量”）
  - 一个进程等待队列 s.queue ，是阻塞在该信号量的各个进程的 PCB 链成的队列。



- **信号量只能通过初始化和两个标准的原语（ wait 、 signal ）来访问 - - 作为 OS 核心代码执行，不受进程调度的打断。**

（ 1 ）P 原语

wait(s) ：**申请一个单位资源**

```
--s.count;
if (s.count <0)
    block(s.queue);
// 阻塞调用进程
// 调用进程 PCB 放入等待队列 s.queue
```

（ 2 ）V 原语

signal(s) ：**释放一个单位资源**

```
++s.count;
if (s.count <= 0)
    wakeup(s.queue);
// 从等待队列 s.queue 中取出一个进程，通常头一个进程
// 将其 PCB 放入就绪队列
```

- **S.count≥0 ：表示系统中可用的资源数量**
- **S.count<0 ：其绝对值表示已阻塞的进程数量**
- **S.count 初值为 1 时：只允许一个进程访问临界资源，是互斥信号量**

两个进程 A 和 B ，共享数据 D 和 E ，为其分别设置互斥信号量 Dmutex 和 Emutex ，初值均为 1 。

```
Process A ：
    wait(Dmutex);
    wait(Emutex);
        使用 D 、 E
    Signal(Dmutex)
    Signal(Emutex)
```

```
Process B ：
    wait(Emutex);
    wait(Dmutex);
        使用 D 、 E
    Signal(Dmutex)
    Signal(Emutex)
```

共享的资源越多，死锁的可能越大。

记录型信号量导致死锁的僵持状态，主要是由于资源竞争、循环等待、非原子操作、错误初始化等原因。解决方法包括保持获取顺序一致、设定超时机制、确保信号量释放、缩短持有时间等。合理设计信号量的使用策略，可以有效降低死锁的风险。

3. AND 型信号量

- **基本思想：将进程在整个运行中需要的所有资源，一次性全部分配给进程，待进程使用完后一起释放。**

- **只要尚有一个资源未能分配给进程，其他所有可能为之分配的资源，也不分配给它。即对临界资源的分配采取原子操作。称为同时 wait 操作即 Swait() 。**

- 在 wait 中加入 AND 条件，又称 AND 同步或同时 wait 操作： Swait

```
Swait(S1,S2,…Sn)
  if  S1≥1 and Sn≥1 then
      for i:=1 to n do
          Si:= Si-1;
      endfor
  else
      当发现第一个 Si<1 就
      把该进程放入等待队列
      并将其程序计数器置于
      Swait 操作的开始位置
  endif
```

```
Ssignal(S1,S2,…Sn)
    for i:=1 to n  do
      Si:= Si+1;
      将所有等待 Si
      的 进程由等待
      队列取出放入
      到就绪队列
    endfor;
```

4. 二进制（ Binary ）信号量

- 其值只能是 0 和 1 ；易于实现。
- 利用二进制信号量可以实现整型信号量。
- 数据结构：

  二进制信号量 S1, S2;

  int C:

- 初值：

  S1 = 1

  S2 = 0

  C = 代表共享资源的初始值

***wait* 操作：**

```
wait(S1);
C - -;
if (C < 0) {
        signal(S1);
        wait(S2);
        }
signal(S1);
```

***Signal* 操作：**

```
wait(S1);
C ++;
if (C <= 0)
    signal(S2);
else
        signal(S1);
```

5. 信号量集

记录型信号量机制：

- 每次只能获得或释放一个单位的资源，低效
- 每次分配前必须测试资源数量，看其是否大于其下界值

对 AND 信号量机制加以扩充

S 为信号量；t 为下限值；d 为需求值

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

```
Ssignal(S1, d1, …, Sn, dn)
    for i:=1 to n do
        Si:= Si +di ;
        _____
    endfor
```

一般信号量集的几种特殊情况：

- Swait(S, d, d) ，只有一个信号量 S ，允许每次申请 d 个资源，若现有资源数少于 d ，不予分配。
- Swait(S, 1, 1) ，蜕化为一般的记录型信号量 (S>1 时) 或互斥信号量 (S=1 时) 。
- Swait(S, 1, 0) ，当 S>=1 时，允许多个进程进入某特定区，当 S 变为 0 后，阻止任何进程进入特定区，相当于可控开关。

1. 利用信号量实现互斥：

- 为临界资源设置一个互斥信号量 *mutex* ，其初值为 1 ；在每个进程中将临界区代码置于 wait(mutex) 和 signal(mutex) 原语之间

```
semaphore mutex = 1;
            …
do{
  …
  wait ( mutex );
  critical section
  signal( mutex );
  remaider section
}while (true);
```

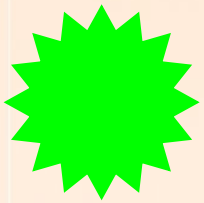
- 注意：
  - wait(mutex) 和 signal(mutex) 必须成对地出现。
  - 缺 wait(mutex) 将会引起系统混乱，不能保证对临界资源的互斥访问
  - 缺 signal(mutex) 将会使该临界资源永久不被释放

初始状态

mutex:=1

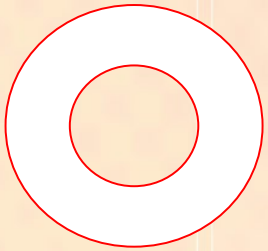
没有并发进程使用临界区

互斥的进程



一个进程申请临界区

mutex:=1



没有并发进程使用临界区

申请成功，进程使用临界区

mutex:=0

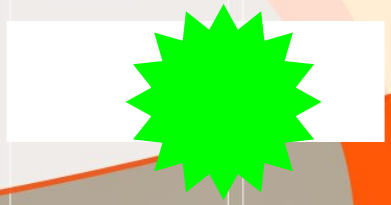
另一个进程也申请临界区

mutex:=0

申请失败

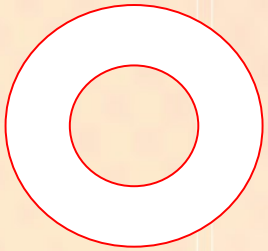
mutex:=-1

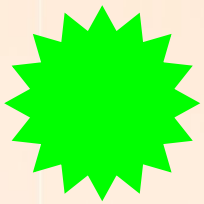
阻塞队列



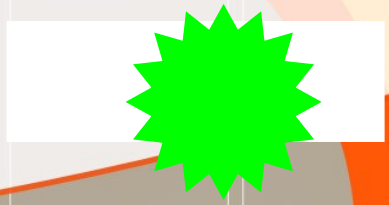
释放资源

mutex:=0



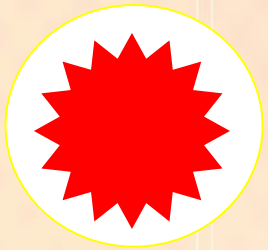


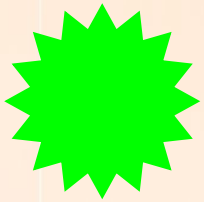
阻塞队列



释放资源

mutex:=0





阻塞队列

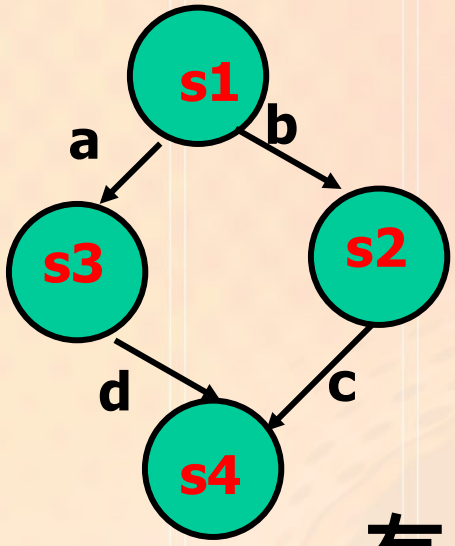
2. 利用信号量实现同步

- 前趋关系：并发执行的进程 P_1 和 P_2 中，分别有代码 C_1 和 C_2 ，要求 C_1 在 C_2 开始前完成；
- 为每个前趋关系设置一个互斥信号量 S12 ，其初值为 0
- P1: P2:

  C1; wait(s12);

  signal(s12) C2 ;

进程同步例题



有四个前趋关系，所以定义四个信号量：

Semaphore a, b, c, d=0;

进程 s1:

```
while (1) {
    .....
    signal (a);
    signal (b);
   }
```

进程 s2:

```
   while (1) {
      wait (b);
         ……
      signal (c);
      }
```

进程 s3:

```
  while (1) {
        wait (a);
           ……
        signal (d);
        }
```

进程 s4:

```
    while (1) {
        wait (d);
        wait (c);
          ……
      }
```

2.4.5 管程的基本概念

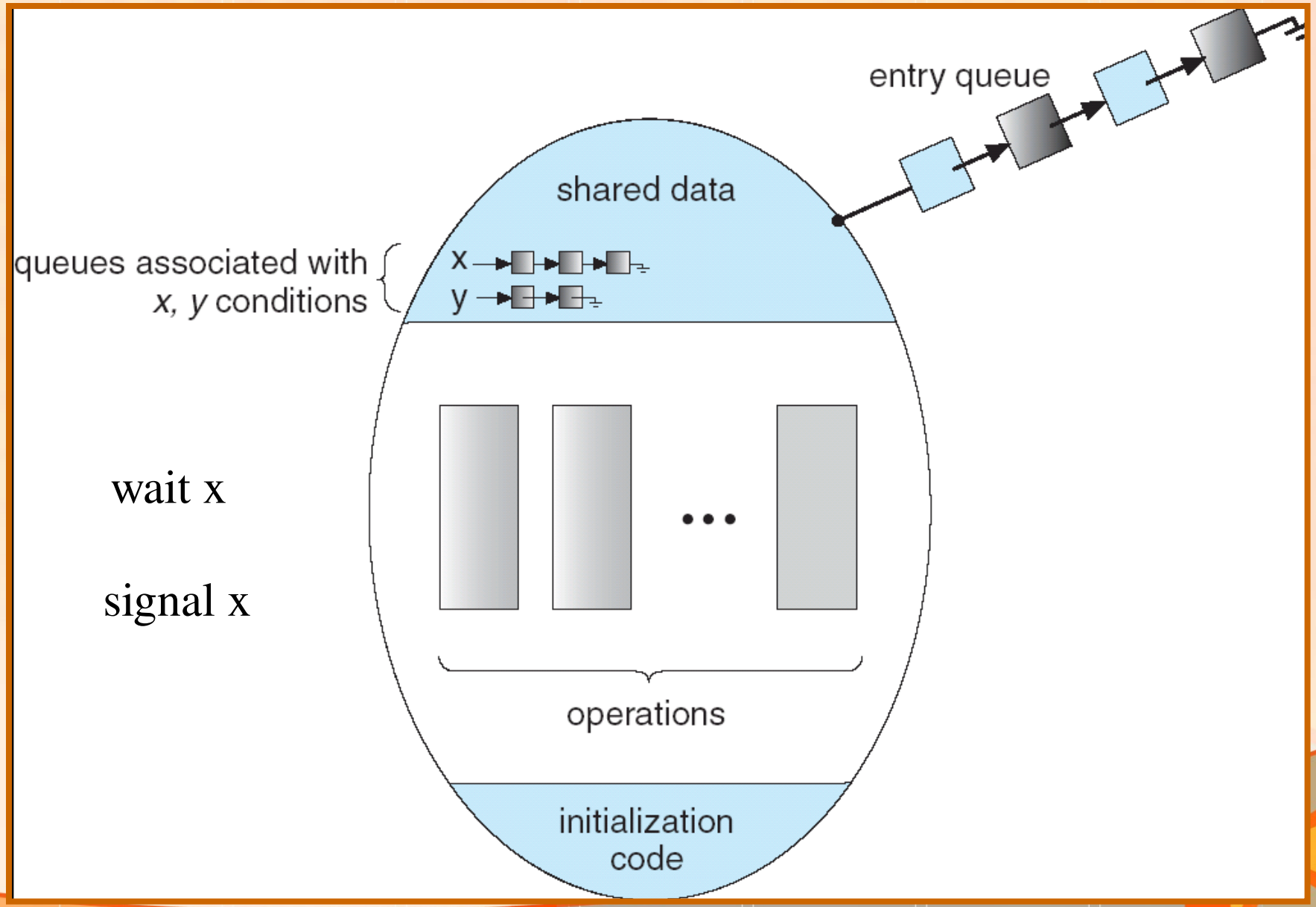
- 利用信号量实现进程同步，使大量的同步操作分散在各个进程中。使系统管理麻烦，同步操作使用不当会引起死锁。
- 引入新的进程同步工具 - - - 管程 (Monitors)
- 目的：分离互斥和条件同步的关注

- 管程由四部分组成：
  - 管程的名称
  - 局部于管程的共享变量说明
  - 对该数据结构进行操作的一组过程
  - 对管程中数据设置初值的语句
- 任何管程外的过程都不能访问管程内的数据结构。管程相当于围墙，将共享变量和对它进行操作的若干过程围了起来，进程只要访问临界资源就必须通过管程。
- 管程每次只允许一个进程进入管程，实现了互斥。
- 使用信号量的效率比管程高。
- 管程结构在一些程序设计语言中得到实现。如并发 Pascal 和 Java ， C# 等，它还被作为一个程序库实现。

- **1. 管程的定义**
  - 系统中各种硬件和软件资源可用抽象数据结构加以描述。
  - 如，一台传真机，可用与该资源有关的状态信息（ busy/ free ）和对它执行的请求和释放操作，以及等待该资源的进程队列来描述。
  - 如，一个 FIFO 队列可用队长、队首、队尾以及在该队列上执行的一组操作来描述。
  - 当共享资源用共享数据结构表示时，资源管理程序可用对该数据结构进行操作的一组过程来表示。
    如， request 、 release
  - 管程定义了一个数据结构和能为并发进程所执行（在该数据结构上）的一组操作，这组操作能同步进程，改变管程中的数据。

- **条件变量**
  - 由于管程通常是用于管理资源的，因而在管程内部，应当存在某种等待机制。当进入管程的进程因资源被占用等原因不能继续运行时，要使其等待。
  - 为了区别不同的等待原因，设置了条件变量和在条件变量上进行操作的两个同步原语 wait, signal 。
  - 条件变量说明形式为： condition ： x ， y;
  - 同步原语 wait 使调用进程等待，并将它排在相应的等待队列上； signal 唤醒等待队列的队首进程。使用方式为： x.wait ， x.signal 。

管程与条件变量


entry queue
shared data
queues associated with
x, y conditions
x
y
wait x
signal x
operations
initialization
code

锁和条件变量

- **Lock**
  - **Lock::Acquire() – 等待直到锁可用，然后抢占锁**
  - **Lock::Release() – 释放锁，唤醒等待者如果有**

- **Condition Variable**
  - **允许等待状态进入临界区**
    - **允许处于等待（睡眠）的线程进入临界区**
    - **某个时刻原子释放锁进入睡眠**
  - **Wait() operation**
    - **释放锁，睡眠，重新获得锁返回后**
  - **Signal() operation (or broadcast() operation)**
    - **唤醒等待者（或者所有等待者），如果有**

条件变量实现

- **实现**
  - **需要维持每个条件队列**
  - **线程等待的条件等待 signal()**

```
Class Condition {
    int numWaiting = 0;
    WaitQueue q;
}
```

```
Condition::Wait(lock){
    numWaiting++;
    Add this thread t  to q;
    release(lock);
    schedule(); //need mutex
    require(lock);
}
```

```
Condition::Signal(){
    if (numWaiting > 0) {
        Remove a thread t from q;
        wakeup(t); //need mutex
        numWaiting--;
    }
}
```

管程例子：生产者 - 消费者问题

```
classBoundedBuffer {
    …
    Lock lock;
    int count = 0;
    Condition notFull, notEmpty;
}
```

```
BoundedBuffer::Deposit(c) {
    lock->Acquire();
    while (count == n)
        notFull.Wait(&lock);
    Add c to the buffer;
    count++;
    notEmpty.Signal();
    lock->Release();
}
```

```
BoundedBuffer::Remove(c) {
    lock->Acquire();
    while (count == 0)
        notEmpty.Wait(&lock);
    Remove c from buffer;
    count--;
    notFull.Signal();
    lock->Release();
}
```