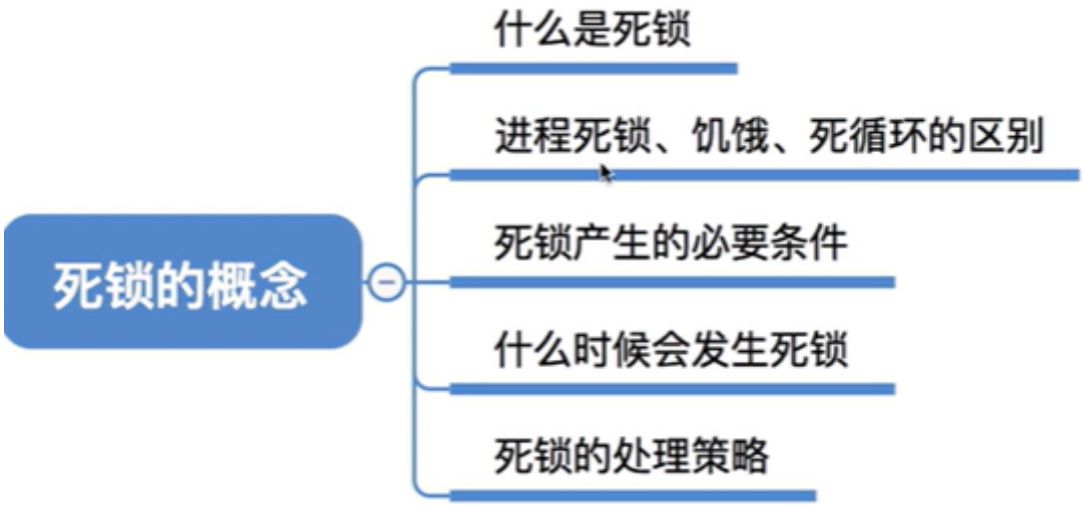
2.4_1_死锁的概念

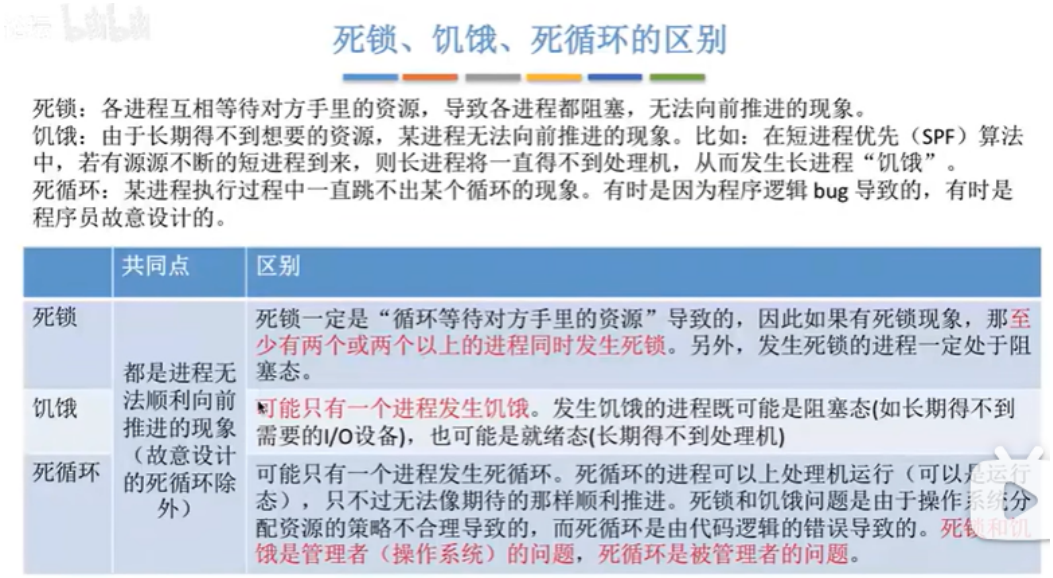
- gxy：
  掌握死锁概念。
  理解、区分死锁、饥饿、死循环。
  掌握死锁发生的必要条件：争夺互斥资源、进程不可剥夺、申请且保持、循环等待链。



- 死锁的概念：
  各个进程因为争夺资源而造成的一种互相等待对方手里的资源，导致各进程都阻塞，无法向前推进的现象。

  发生死锁之后，如果没有外力的干涉，这些进程都将无法向前推进。

- 死锁、饥饿、死循环区别：



死锁、饥饿、死循环的区别

死锁：各进程互相等待对方手里的资源，导致各进程都阻塞，无法向前推进的现象。
饥饿：由于长期得不到想要的资源，某进程无法向前推进的现象。比如：在短进程优先（SPF）算法中，若有源源不断的短进程到来，则长进程将一直得不到处理机，从而发生长进程“饥饿”。
死循环：某进程执行过程中一直跳不出某个循环的现象。有时是因为程序逻辑 bug 导致的，有时是程序员故意设计的。

	共同点	区别
死锁	都是进程无法顺利向前推进的现象（故意设计的死循环除外）	死锁一定是“循环等待对方手里的资源”导致的，因此如果有死锁现象，那至少有两个或两个以上的进程同时发生死锁。另外，发生死锁的进程一定处于阻塞态。
饥饿		可能只有一个进程发生饥饿。发生饥饿的进程既可能是阻塞态(如长期得不到需要的I/O设备)，也可能是就绪态(长期得不到处理机)
死循环		可能只有一个进程发生死循环。死循环的进程可以上处理机运行（可以是运行态），只不过无法像期待的那样顺利推进。死锁和饥饿问题是由于操作系统分配资源的策略不合理导致的，而死循环是由代码逻辑的错误导致的。死锁和饥饿是管理者（操作系统）的问题，死循环是被管理者的问题。

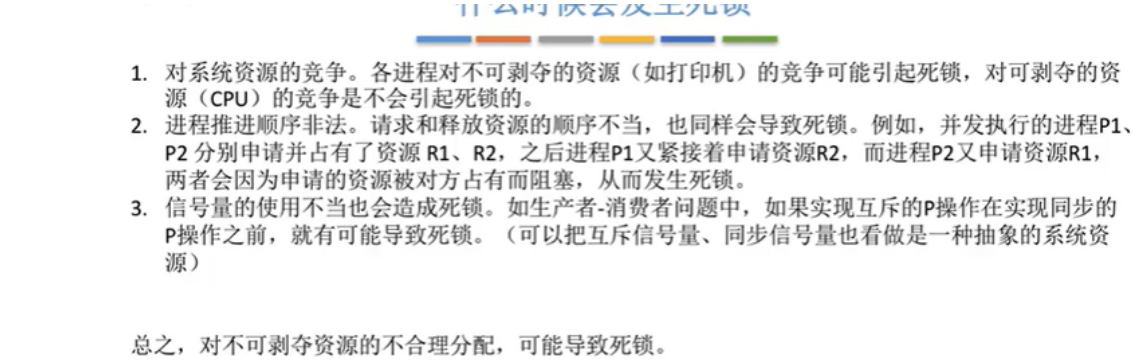
  饥饿：长期得不到自己想要的资源，导致一直不能往前推进的状态。
  比如之前的short process first中，长进程很有可能会一直没有办法得到处理机资源，就会一直等待，就是饥饿。

饥饿往往是 阻塞态或者就绪态
而死循环可以是运行态，只不过是不会往下推进，其实一直在执行。

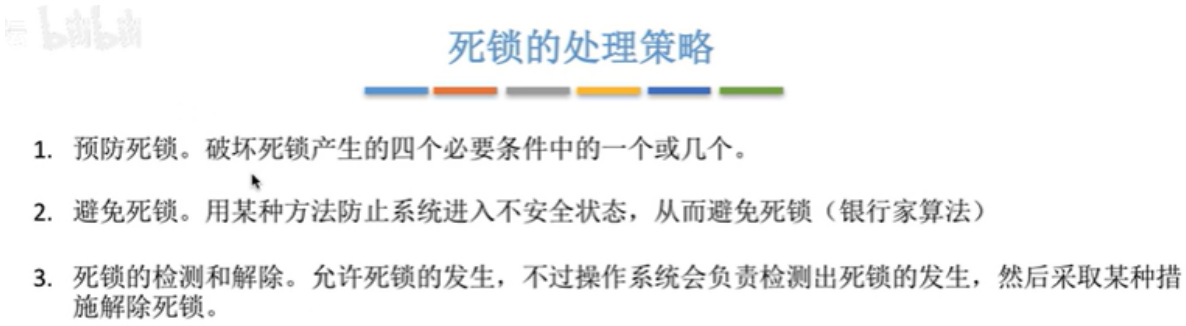
死循环：往往是逻辑错误，或者是程序员故意如此设计。

- 死锁产生的必要条件：
  1.互斥条件。必须是进程对于互斥资源的争抢。
  2.不剥夺条件。进程获得的资源只能主动释放，不能被其他进程强行夺走。
  3.请求和保持条件。每一个进程至少保持一个资源，至少申请一个新的资源，又对自己已经有的资源保持不放。
  4.循环等待条件。一定有循环等待链。循环等待链是死锁的必要不充分条件。

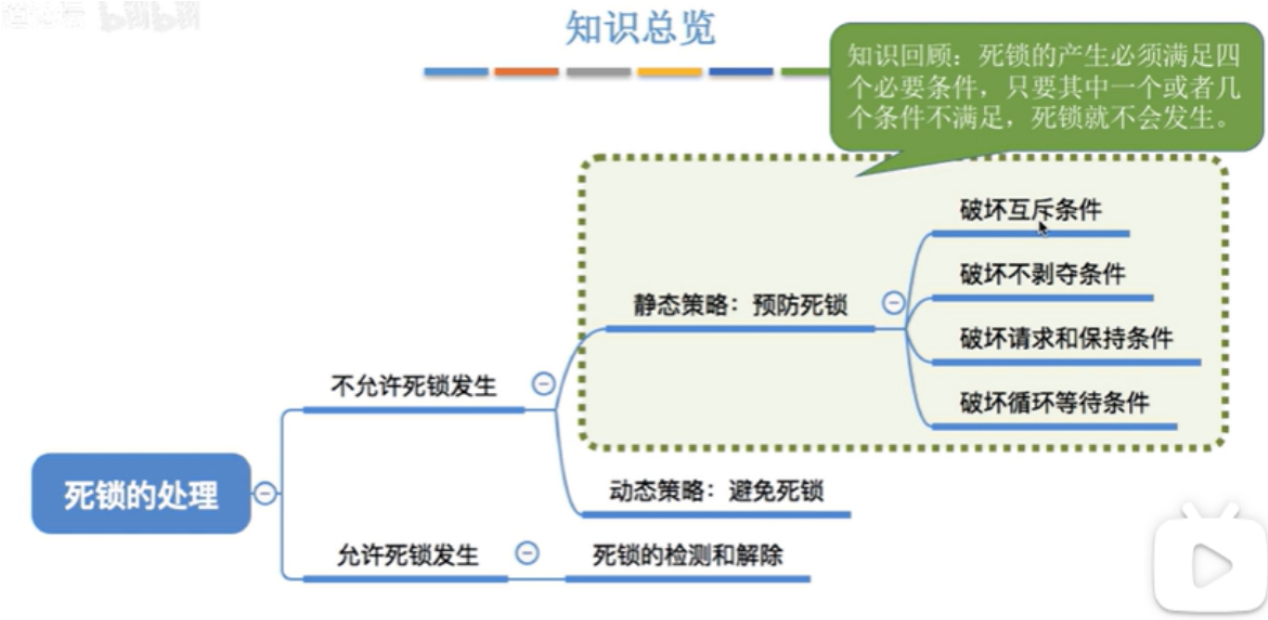
- 什么时候会发生死锁：
  下面会提供一个例子。
  一句话总结就是 对不可剥夺资源的不合理分配就可能会导致死锁。



- 死锁处理策略：



死锁处理的大纲：

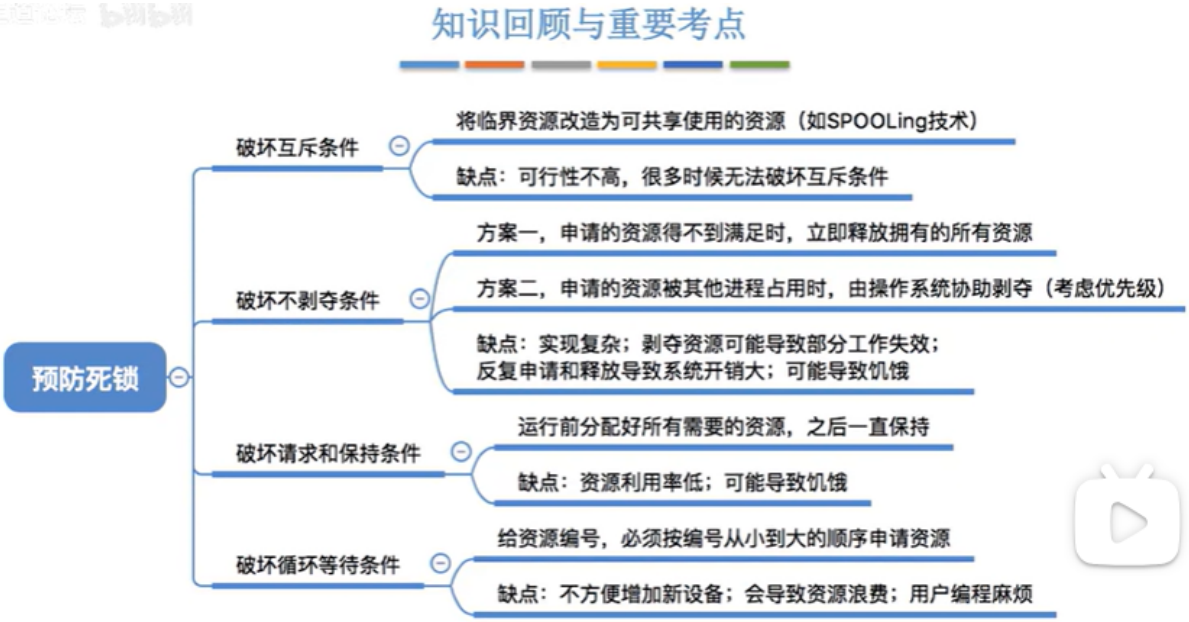


预防和避免都是从不允许死锁发生的角度。
死锁的检测和解除是允许死锁的发生的角度。

2.4_2_死锁处理策略--预防死锁

预防死锁的思路，就是看能不能破坏死锁产生的四个必要条件，只要破坏了至少一个，死锁就不会发生。

- gxy:
  这一节课的所有内容，理解即可，不需要会背。

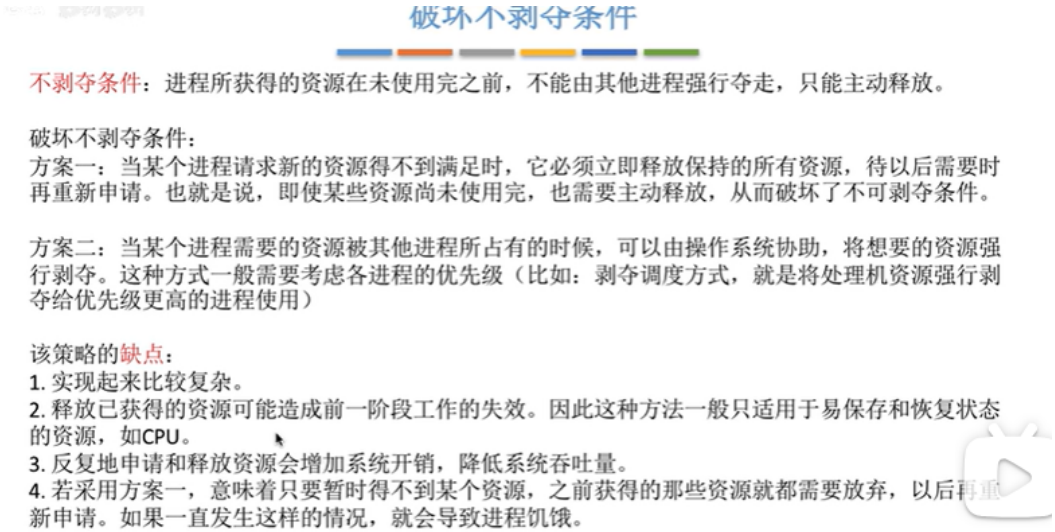


- 破坏互斥条件：只有对互斥资源进行争夺才会产生死锁。

比如在多个进程同时争夺打印机的使用权时，可以采用SPOOLing技术，可以把打印机这个独占设备改为共享设备。
具体实现的细节时SPOOLing技术里面的输出进程接受了多个进程的打印机请求之后，自己内部进行处理并在最后输出。然后从进程的角度看，打印机的资源变为了共享设备，但只是从进程的角度看。

这个方法的缺点：不是所有的互斥资源都可以更改为共享资源，而且出于系统安全的考虑，很多互斥条件时必须保持的，不能被破坏。

- 破坏不剥夺条件：进程中获得的资源在使用完毕之前，不能被其他进程强行夺走，只能自己主动释放。

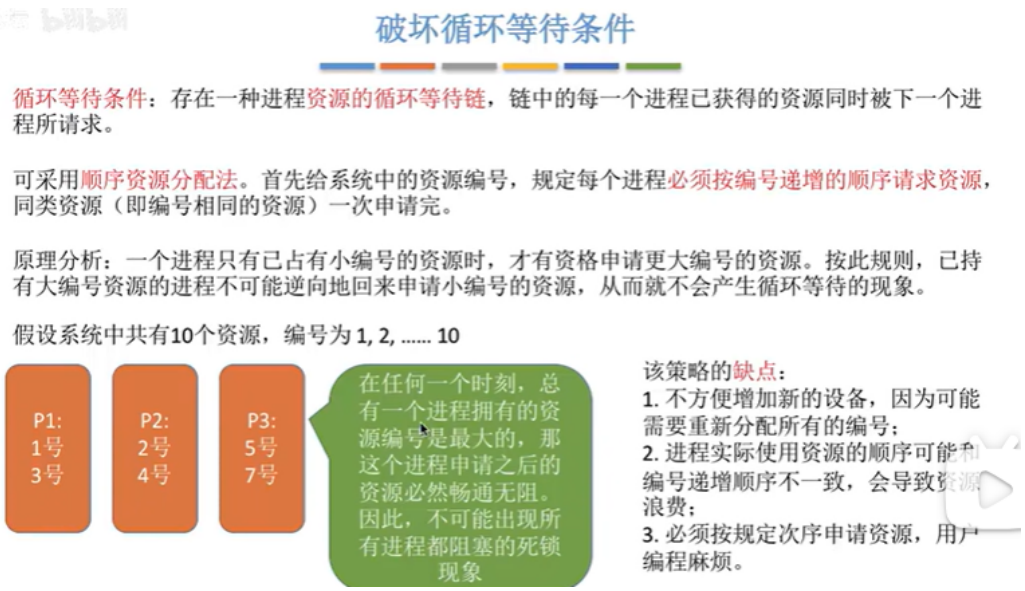


- 破坏请求和保持条件：进程已经至少保持了一个资源，但是又提出了新的资源的请求，但是新的资源被其他进程等占用，所以自己进入阻塞状态，同时保持自己原来的资源不变。

  可以采用 静态分配方法 ：进程在运行前一次申请完所有它需要的全部资源，只有得到全部资源之后才会进入运行，并且一旦运行之后，这些资源就一直属于这个进程。

  缺点：如果一个资源利用时间很短，进程运行时间长，这个资源就会一直闲置，导致这个资源的利用率低。有可能产生饥饿：有一种进程1需要资源1，进程2需要资源2，进程3需要1\2两个资源，在有很多进程1的情况下，进程3就一直不能得到自己需要的所有的资源，就会饥饿。

- 破坏循环等待链条件：



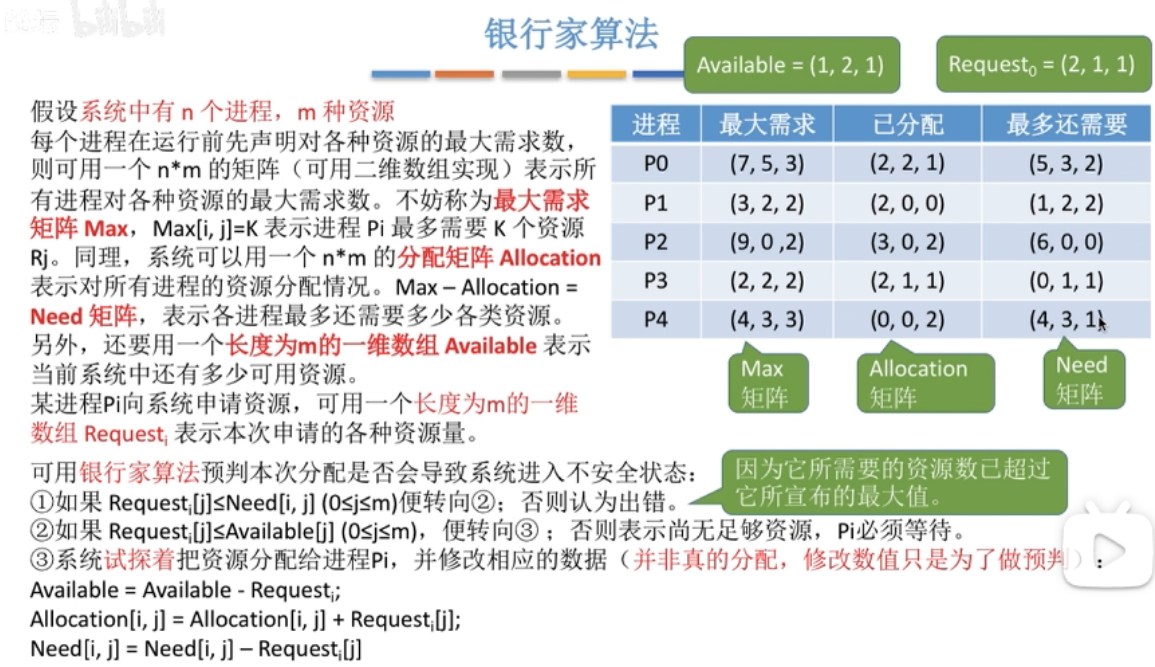
2.4_3_死锁处理策略避免死锁

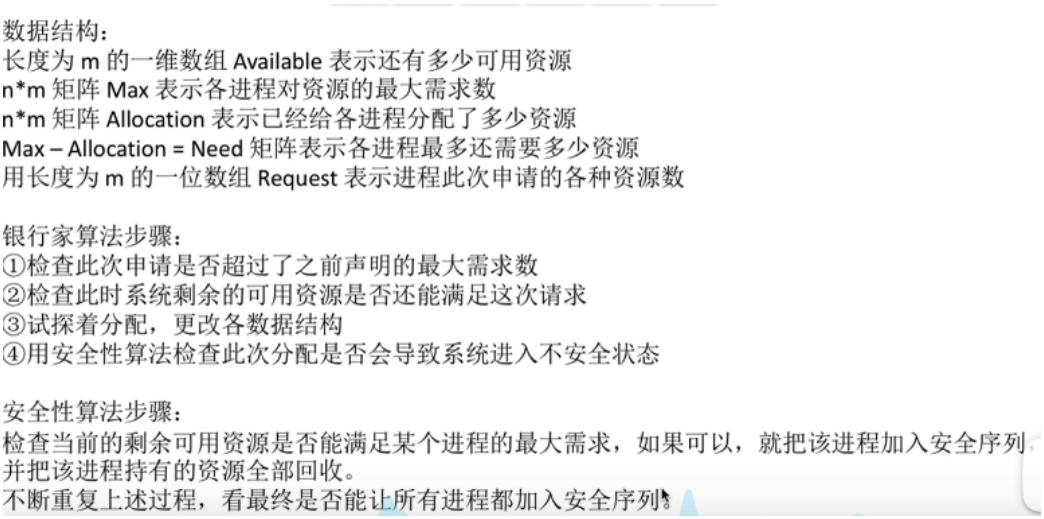
- gxy总结：
  理解什么是安全序列，掌握不安全状态和死锁的关系
  掌握银行家算法流程。
  需要掌握 need 是需求矩阵 max 是最大需求矩阵 allocation是已经分配的资源矩阵

- 安全序列：就是指如果系统按照这种序列分配资源，每一个进程都可以顺利完成。只要能找到一个安全序列，就是处于安全状态。
  如果系统分配资源之后，找不到一种安全序列，系统就进入了不安全状态，意味着之后有可能进程没有办法顺利执行下去。但是也有可能会顺利执行。（但是分配资源的时候要考虑最坏的情况）
  所以：
  不安全状态不一定会发生死锁，但是死锁一定是处于不安全状态。
  安全状态一定不会发生死锁。

- 银行家算法：
  思路：在资源分配之前预先判断这次分配是否会导致系统进入不安全状态，以此来决定是否要分配请求。如果会进入不安全状态，就不进行这一次资源的分配，就应该让这个进程首先阻塞等待。

  存储的数据结构：





- 银行家算法步骤:

  1.计算这一次的资源分配是不是超过了这个进程的最大需求。

  2.检查资源是不是可以满足这一次的分配。

  3.试探分配，之后更改数据结构。

  4.用安全性算法检查这一次分配之后会不会进入不安全状态。

- 安全性算法检查：

  假设当前是a,b,c 找一下所有进程的need矩阵，如果a b c可以满足要求，就可以完成这个进程，就可以让当前资源 加上 这个进程的所有资源，其实就是加上allocation，已经分配的资源。然后再从第一个进程开始判断，最后看能不能找到一种顺序把所有的进程都处理完毕。

2.4_4_死锁的处理策略，检测和解除

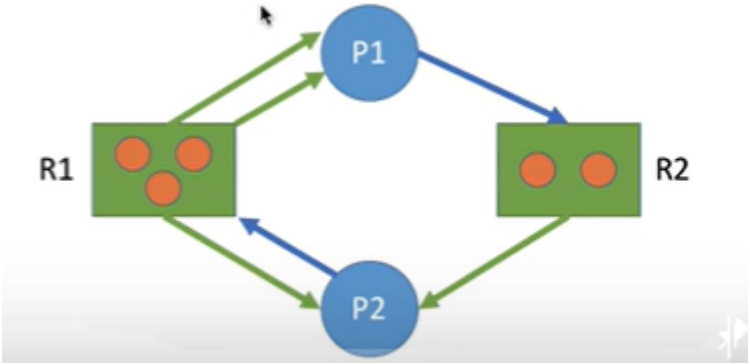
- gxy 总结：
  死锁的检测 比较重要，资源分配图需要懂，会简化过程
  死锁检测算法：依次消除与不阻塞进程想连的边，直到无边可消。
  死锁定理：如果某时刻系统的资源分配图是不可以简化的，就说明发生了死锁。

  解除死锁的方法：资源剥夺法，撤销进程法，进程回退法。

- 死锁的检测：
  使用 资源分配图 这种数据结构。

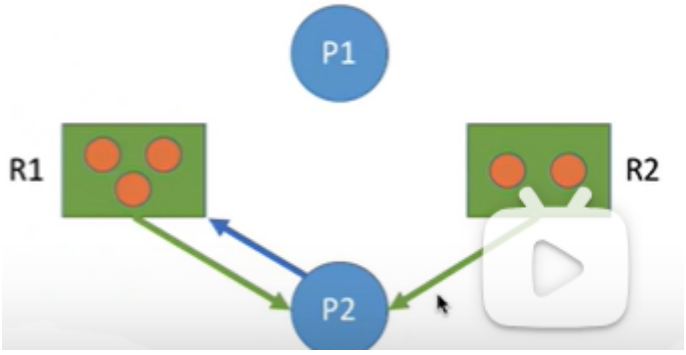
  资源分配图的组成：
  两种结点：进程结点对应一个进程；资源结点对应一类资源；
  两种边：
  进程结点→资源结点:表示进程想要申请几个资源(每条边代表一个)。
  资源结点→进程结点：表示已经为当前进程分配了几个这个资源。(也是每一条边代表一个)

  举例：

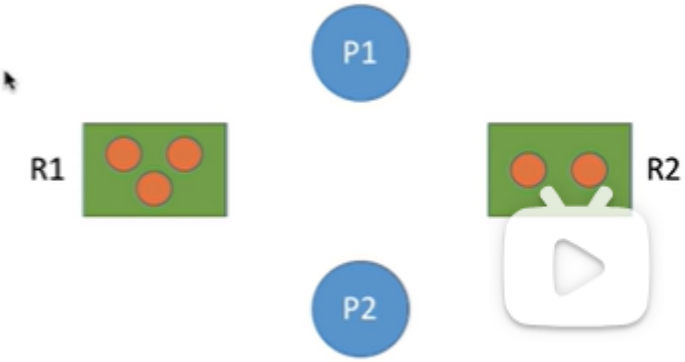


  P1请求1个R2资源，P2请求一个R1资源。
  R1分配给P1两个资源，R1\R2都分配给了P2一个资源。

  - 简化过程：对于P2，请求一个R1，但是R1已经分配出去了三个，并且自己是3个，所以没有办法给P2。但是对于P1,申请P2。P2已经分配出去一个，所以可以给P1再分配一个，所以P1可以简化为：



    之后就可以再简化为：

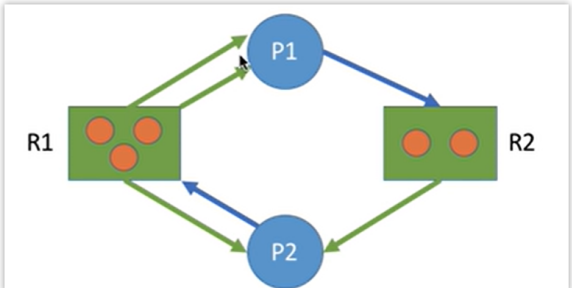


按照上面过程分析，最终能够消除所有边，就称这个图是可以完全简化的，可以完全简化的不会发生死锁。

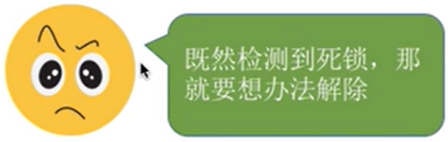
如果最终不能消除所有的边，那么此时就是发生了死锁。

死锁的检测

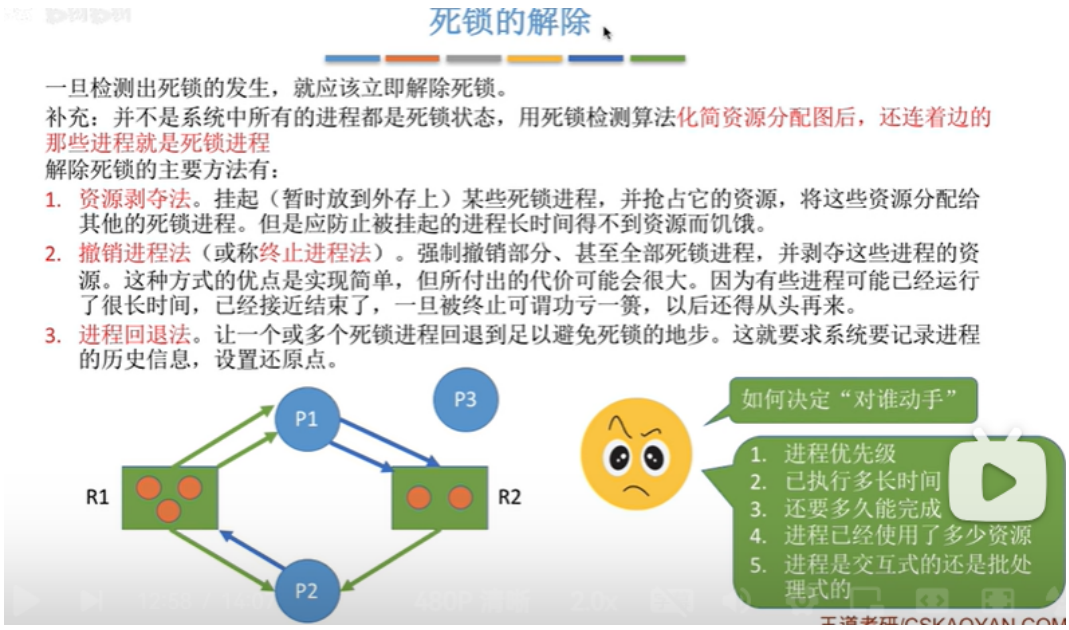
检测死锁的算法：
1）在资源分配图中，找出既不阻塞又不是孤点的进程 Pi（即找出一条有向边与它相连，且该有向边对应资源的申请数量小于等于系统中已有空闲资源数量。如下图中，R1没有空闲资源，R2有一个空闲资源。若所有的连接该进程的边均满足上述条件，则这个进程能继续运行直至完成，然后释放它所占有的所有资源）。消去它所有的请求边和分配变，使之称为孤立的结点。在下图中，P1 是满足这一条件的进程结点，于是将P1的所有边消去。
2）进程 Pi 所释放的资源，可以唤醒某些因等待这些资源而阻塞的进程，原来的阻塞进程可能变为非阻塞进程。在下图中，P2 就满足这样的条件。根据 1）中的方法进行一系列简化后，若能消去途中所有的边，则称该图是可完全简化的。



死锁定理：如果某时刻系统的资源分配图是不可完全简化的，那么此时系统死锁



- 死锁解除的策略和处理哪一个进程的一些优先级选择：


死锁的解除
一旦检测出死锁的发生，就应该立即解除死锁。
补充：并不是系统中所有的进程都是死锁状态，用死锁检测算法化简资源分配图后，还连着边的那些进程就是死锁进程
解除死锁的主要方法有：
1. 资源剥夺法。挂起（暂时放到外存上）某些死锁进程，并抢占它的资源，将这些资源分配给其他的死锁进程。但是应防止被挂起的进程长时间得不到资源而饥饿。
2. 撤销进程法（或称终止进程法）。强制撤销部分、甚至全部死锁进程，并剥夺这些进程的资源。这种方式的优点是实现简单，但所付出的代价可能会很大。因为有些进程可能已经运行了很长时间，已经接近结束了，一旦被终止可谓功亏一篑，以后还得从头再来。
3. 进程回退法。让一个或多个死锁进程回退到足以避免死锁的地步。这就要求系统要记录进程的历史信息，设置还原点。
P1
P2
P3
R1
R2
如何决定“对谁动手”
1. 进程优先级
2. 已执行多长时间
3. 还要多久能完成
4. 进程已经使用了多少资源
5. 进程是交互式的还是批处理式的

这一部分 理解即可。