上节课重点：

1. 作业调度算法（先来先服务FCFS，短作业优先SJF，高响应比优先HRRN）

2. 进程调度算法（时间片轮转RR、优先级PS、多级反馈队列MFQ）

实现实时调度的基本条件

- 提供必要的信息
- 系统处理能力强
- 采用抢占式调度机制
- 具有快速切换机制

3.4 实时调度

由于在实时系统中都存在着若干个实时进程或任务，它们用来反应或控制某些外部事件，往往带有某种程度的紧迫性，因而对实时系统中的调度提出了某些特殊要求，前面所介绍的多种调度算法，并不能很好的满足实时系统对调度的要求，为此，需要引入一种新的调度，即实时调度。

3.4 实时调度

实时系统中包含两种任务：

硬实时任务指必须满足最后期限的限制，否则会给系统带来不可接受的破坏或者致命错误。

软实时任务也有一个与之关联的最后期限，并希望满足这个期限的要求，但这并不是强制的，即使超过了最后期限，调度和完成这个任务仍然是有意义的。

3.4.1　实现实时调度的基本条件

实现实时调度的基本条件

在实时系统中，硬实时任务和软实时任务都联系着一个截止时间。为保证系统能正常工作，实时调度必须满足任务对截止时间的要求，为此，实现实时调度应具备下列4个条件：

1. 提供必要的信息
2. 系统处理能力强
3. 采用抢占式调度机制
4. 具有快速切换机制

1. 提供必要的信息

- 为了实现实时调度，系统应向调度程序提供有关任务的下述信息：
  - 就绪时间：该任务成为就绪状态的起始时间。
  - 开始截止时间和完成截止时间
  - 处理时间
  - 资源要求
  - 优先级

- 2.　系统处理能力强

在实时系统中，通常都有多个任务，若处理机的能力不够强，则可能会出现某些实时任务不能得到及时处理导致发生难以预料的后果。假如系统中有M个周期性的硬实时任务，处理时间为Ci，周期时间表示为Pi ,则单机系统中必须满足条件：

$$\sum(Ci / Pi) \leq 1$$

3.4.1　实现实时调度的基本条件

如：系统中有6个硬实时任务，他们的周期时间都是50ms，每次的处理时间是10ms。则可以算出系统是不可调度的。

解决方法有二个：

增强单机系统的处理能力

采用多处理机系统（则限制条件为∑(Ci / Pi) ≤N，

N为处理机数）。

- 3. 采用抢占式调度机制

在含有硬实时任务的实时系统中，广泛采用抢占机制。

当一个优先权更高的任务到达时，允许将当前任务暂时挂起，令高优先权任务立即投入运行，这样可满足该硬实时任务对截止时间的要求。但此种机制较复杂。

- 4. 具有快速切换机制
  - 对外部中断的快速响应能力
    - 具有快速硬件中断机构，且禁止中断的时间间隔尽量短。
  - 快速任务分派能力
    - 为提高任务切换的速度，系统中每个运行单位要适当地小。

3.4.2 实时调度算法的分类

可按照不同方式对实时调度算法加以分类：

- 根据实时任务性质的不同，分为硬实时调度算法和软实时调度算法；
- 按调度方式的不同，分为非抢占调度算法和抢占调度算法；
- 根据调度程序调度时间的不同，分为静态调度算法和动态调度算法。
- 多处理机环境下，可分为集中式调度和分布式调度两种算法。

3.4.2 实时调度算法的分类

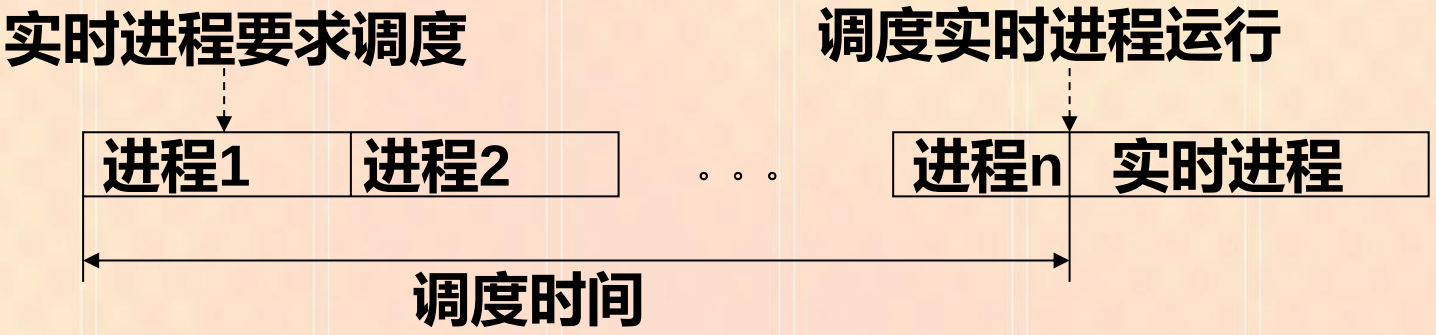
1、非抢占调度算法

该算法较简单，用于一些小型实时系统或要求不太严格的实时系统中。又可分为两种：

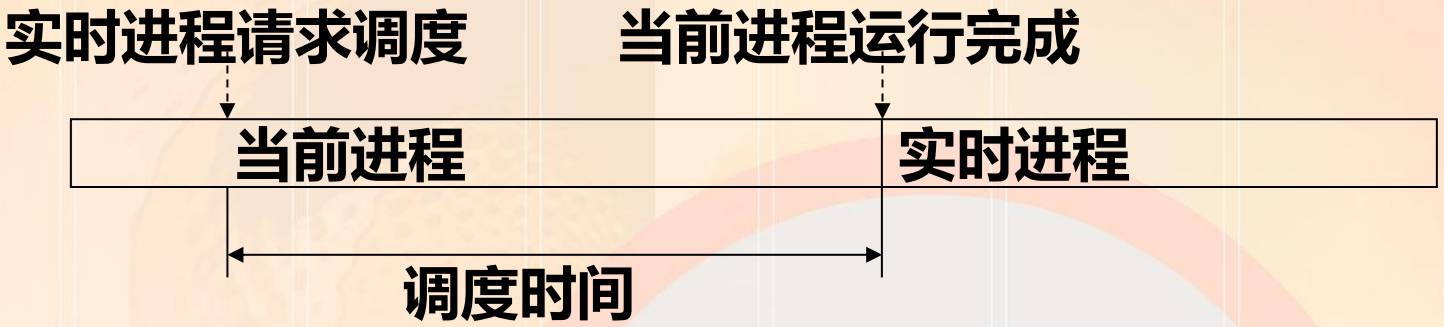
非抢占式轮转调度算法。常用于工业生产的群控系统中，要求不太严格（响应时间约为数秒）

非抢占式优先调度算法。要求较为严格，根据任务的优先级安排等待位置。可用于有一定要求的实时控制系统中。（响应时间约为数百ms）

3.4.2 实时调度算法的分类



1）非抢占式轮转调度图示



2）非抢占式优先权调度图示

2、抢占调度算法

用于要求较严格的实时系统中，（t 约为数十ms），采用抢占式优先权调度算法。根据抢占发生时间的不同可分为两种：

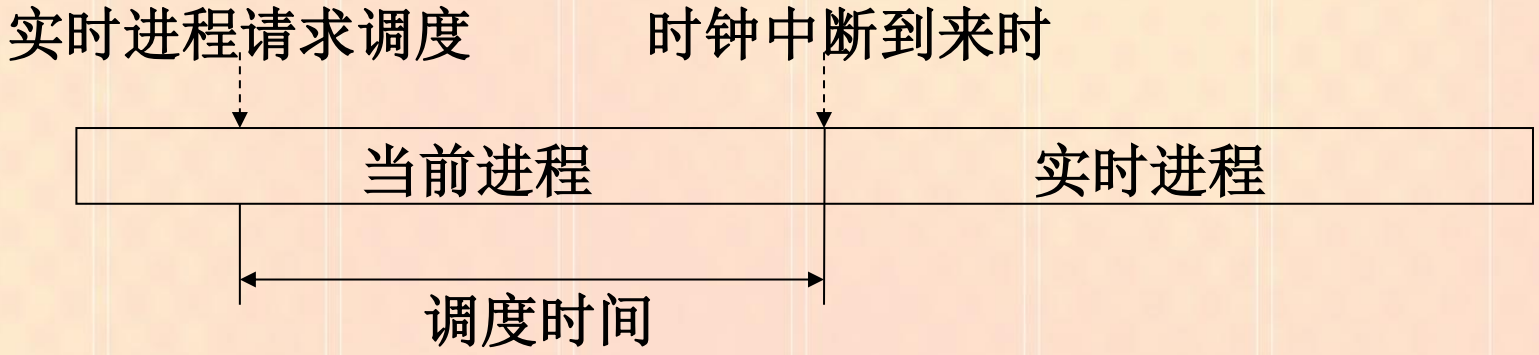
1、基于时钟中断的抢占式优先权调度算法

某高优先级任务到达后并不立即抢占，而等下一个时钟中断时抢占。（延时几十毫秒到几毫秒）

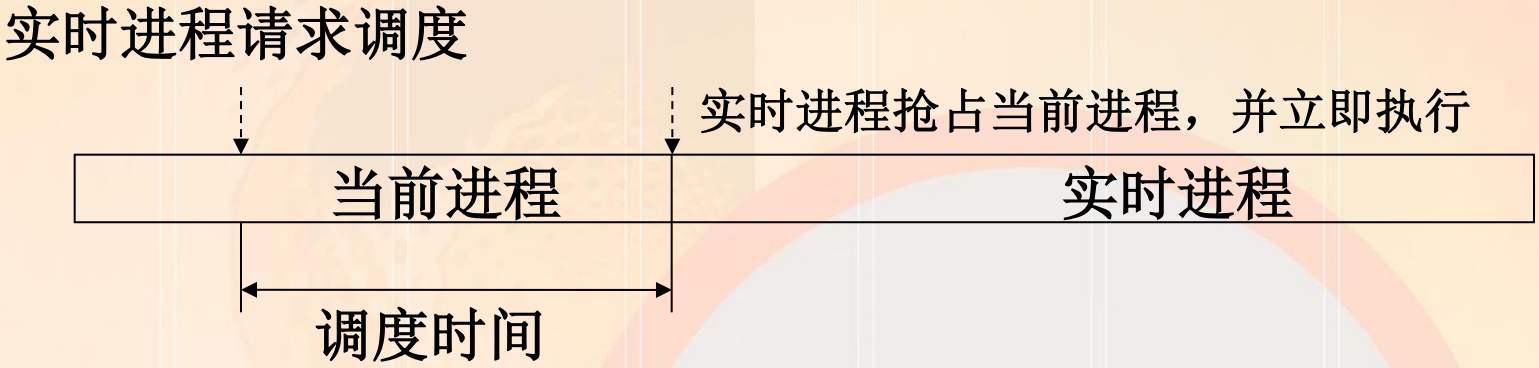
2、立即抢占的优先权调度算法

一旦出现外部中断，只要当前任务未处于临界区，就立即抢占处理机。（延时几毫秒到100微秒）

3.4.2 实时调度算法的分类



1） 基于时钟中断抢占的优先权调度图示



2） 立即抢占的优先权调度图示

3.4.3 常用的几种实时调度算法

目前有许多实时调度算法：

- 最早截止时间优先EDF（Earliest Deadline First）算法
- 最低松弛度优先LLF（Least Laxity First）算法

3.4.3 常用的几种实时调度算法

1. 最早截止时间优先算法EDF（Earliest Deadline First）

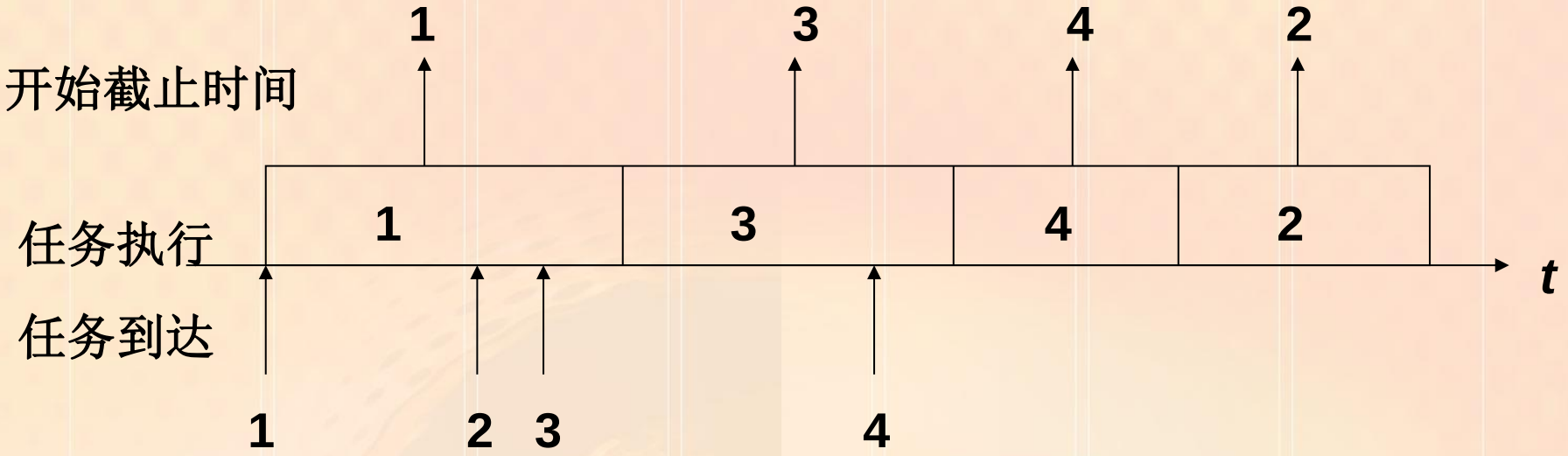
根据任务的截止时间来确定任务的优先级。截止时间越早，其优先级越高。

该算法要求在系统中保持一个实时任务就绪队列，该队列按各任务截止时间的早晚排序，调度程序在选择任务时总是选择就绪队列中的第一个任务，为之分配处理机，使之投入运行。

EDF算法既可以用于抢占式调度，也可用于非抢占式调度。

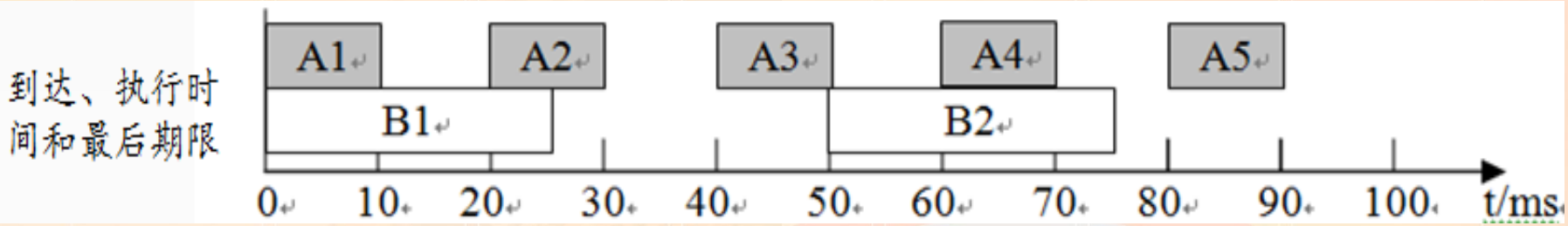
3.4.3 常用的几种实时调度算法

1）非抢占式调度方式用于非周期实时任务



2)抢占式调度方式用于周期实时任务

下图中有两个周期性任务，任务A的周期时间为20ms，每个周期的处理时间为10ms；任务B的周期时间为50ms，每个周期的处理时间为25ms。



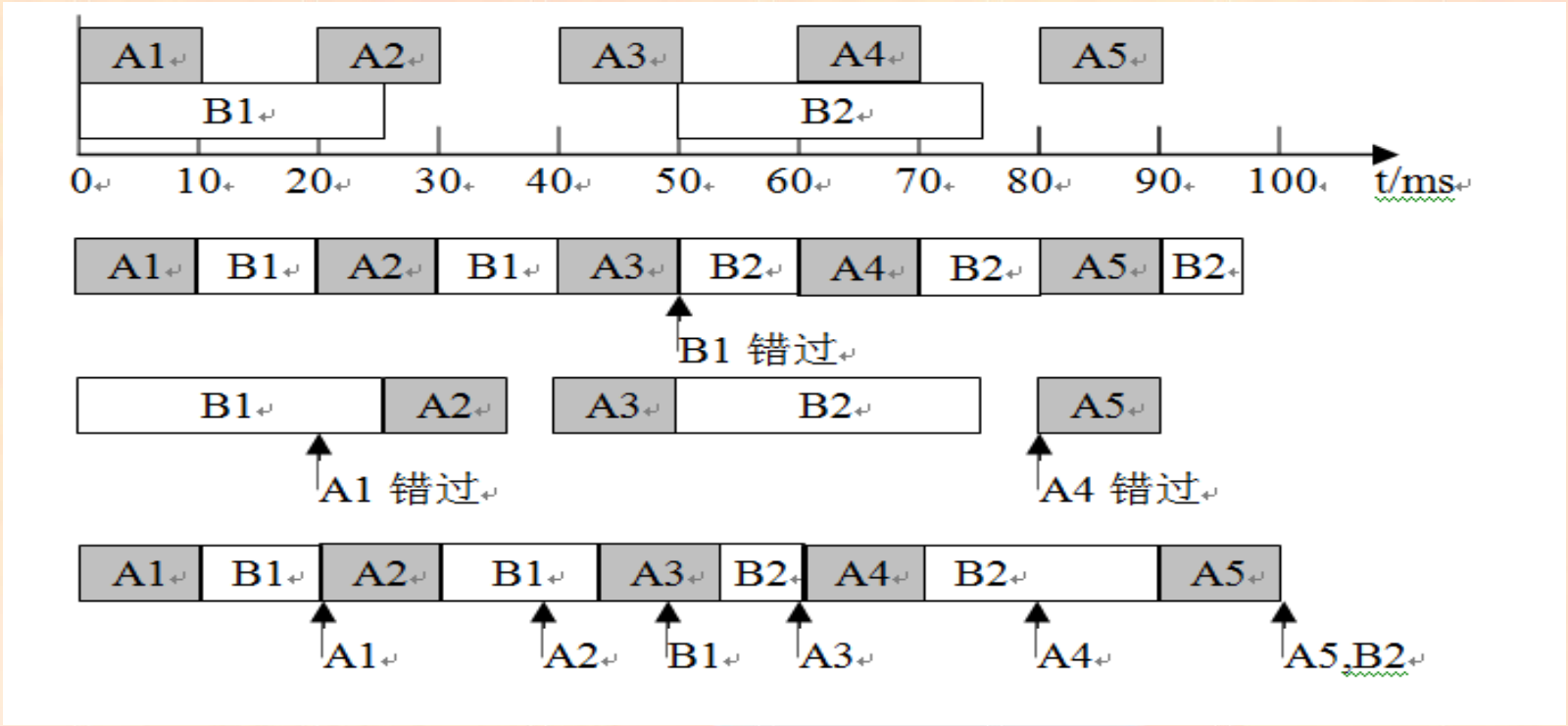
3.4.3 常用的几种实时调度算法

到达、执行时间和最后期限

固定优先级（A优先级高）

固定优先级（B优先级高）

使用完成最后期限最早和最后期限调度



最早截止时间优先算法用于抢占调度方式

3.4.3 常用的几种实时调度算法

在t=0时，A1和B1同时到达，由于A1的截止时间比B1早，所以调度A1执行；

在t=10时，A1完成，又调度B1执行；

在t=20时A2到达，由于A2的截止时间比B1早，故B1被中断而调度A2执行；

在t=30时，A2执行完成，又重新调度B1执行；

在t=40时A3又到达了，但B1的截止时间要比A3早，仍应让B1继续执行，直到完成（t=45）,然后再调度A3执行；

在t=50时，A3完成，又调度B2执行。

2. 最低松弛度优先LLF（Least Laxity First）算法

该算法是根据任务紧急（或松弛）的程度，来确定任务的优先级。任务的紧急程度越高，为之赋予的优先级就越高。

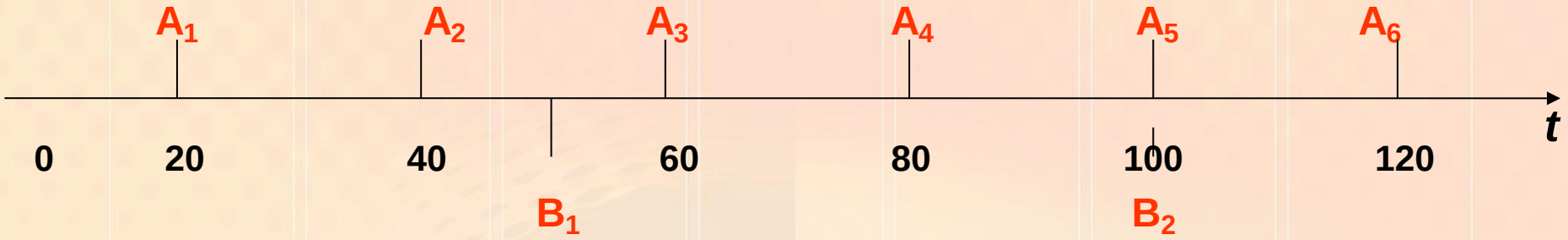
例如，任务A在200ms时必须完成，本身运行时间100ms，则必须在100ms之前调度执行，A任务的紧急（松弛）程度为100ms，又如任务B在400ms是必须完成，需运行150ms，其松弛程度为250ms.

该算法主要用于抢占调度方式中。

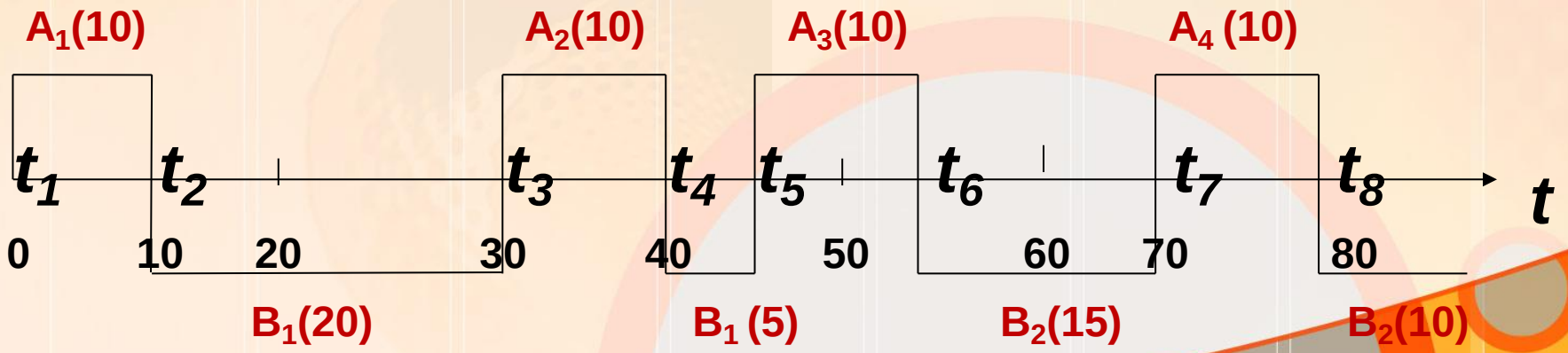
3.4.3 常用的几种实时调度算法

假如在一个实时系统中，有两个周期型实时任务A,B，任务A要求每20ms执行一次，执行时间为10ms；任务B要求每50ms执行一次，执行时间为25ms；由此可得知A，B任务每次必须完成的时间分别为A_1、A_2、A_3…和B_1、B_2、B_3…

LLF算法例

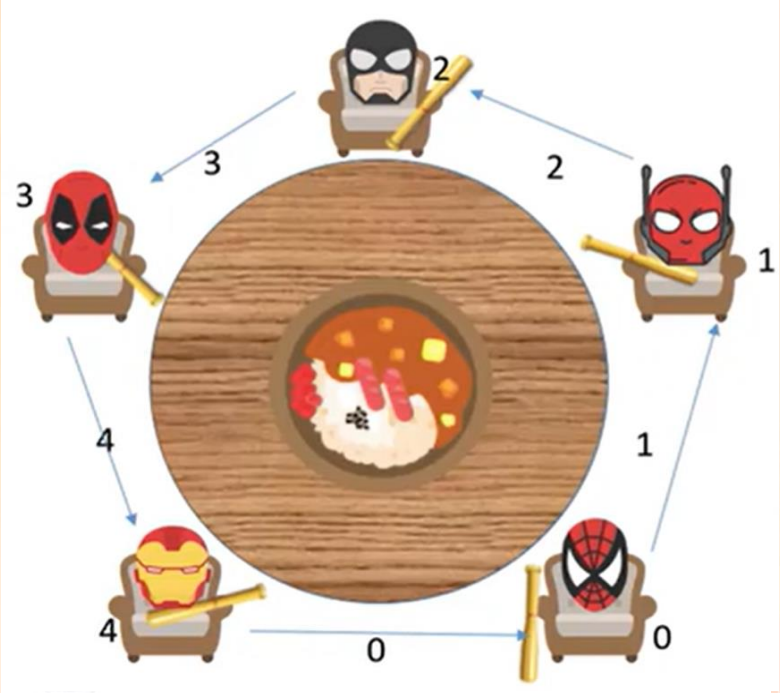


松弛度=必须完成时间-本身的运行时间-当前时间



什么是死锁？

哲学家进餐问题中，如果5位哲学家进程并发执行，都拿起了左手边的筷子：




```
semaphore chopstick[5]={1,1,1,1,1};
Pi (){
    while(1){
        P(chopstick[i]);                  //拿左
        P(chopstick[(i+1)%5]);    //拿右吃饭
        V(chopstick[i]);                  //放左
        V(chopstick[(i+1)%5]);    //放右思考
    }
}
```

每位哲学家都在等待自己右边的人放下筷子，这些哲学家进程都因等待筷子资源而被阻塞。即发生“死锁”。

死锁的定义：

在多道程序系统中，虽借助于多个进程的并发执行，改善了系统的资源利用率，提高了系统的吞吐量，但可能发生一种危险--死锁。

死锁（Deadlock）：是指多个进程在运行过程中因争夺资源而造成的一种僵局，当进程处于这种状态时，若无外力作用，它们都将无法再向前推进。

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。

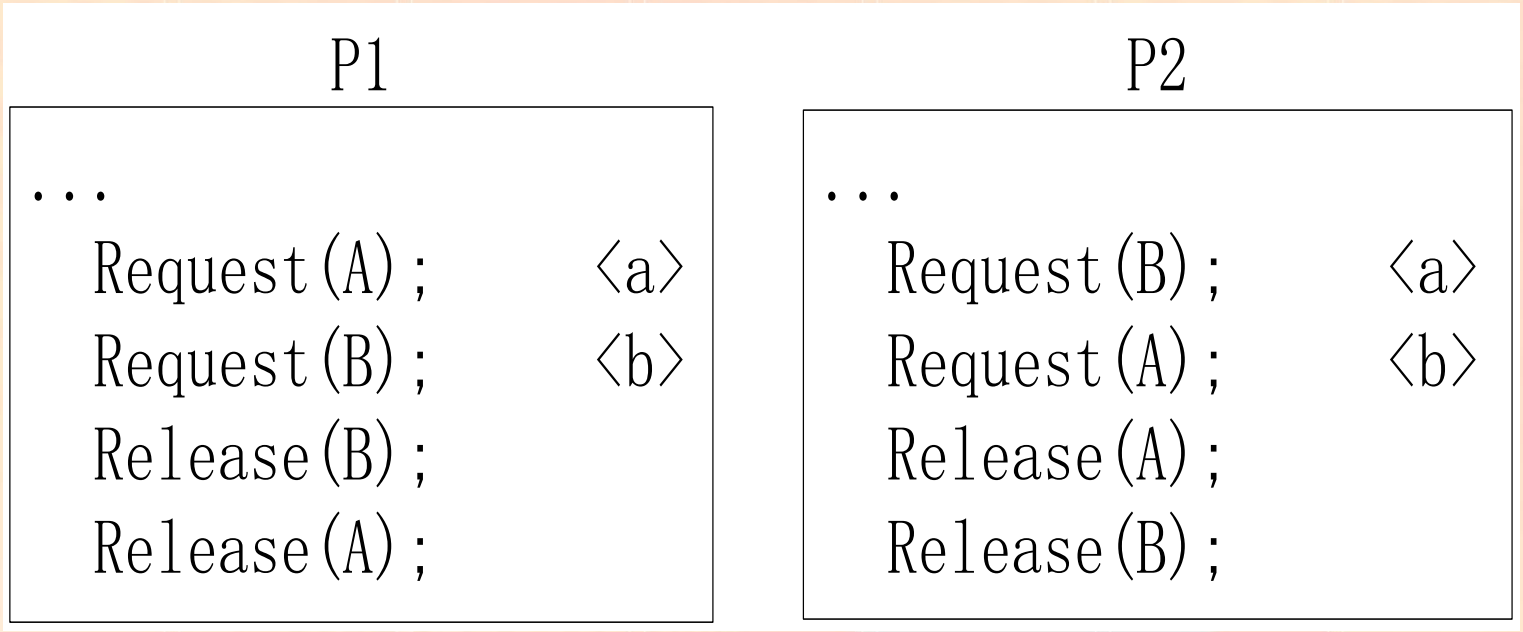
它们都是进程无法顺利向前推进的现象。

产生死锁的原因和必要条件

- 产生死锁的原因：
  - 竞争资源
    - 资源数目不能满足进程需要
  - 顺序不当
    - 进程运行过程中，请求和释放资源顺序不当
    - 多个进程并发执行，相互的推进顺序不确定，可能会导致两种结果：不出现死锁和出现死锁。
  - 可剥夺和不可剥夺资源
    - 可剥夺的如：处理机、内存（优先权高的剥夺优先权低的）
    - 不可剥夺资源如：磁带、打印机

竞争不可剥夺资源引起死锁

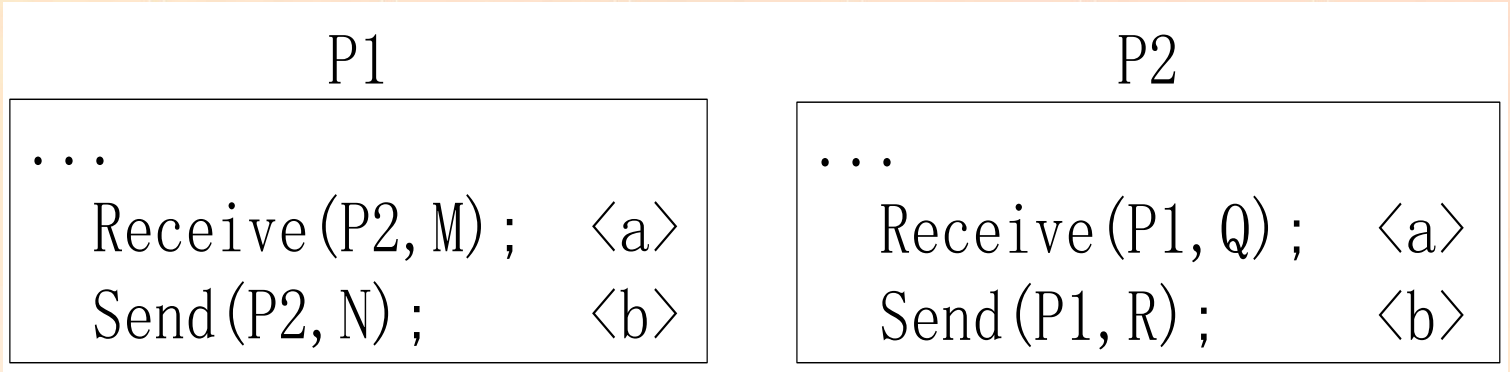
- 竞争不可剥夺资源可能会引起死锁：进程所获得的资源在未使用完之前，不能由其他进程强行夺走，只能主动释放。



- 死锁发生：双方都拥有部分资源，同时在请求对方已占有的资源。

竞争临时性资源引起死锁

- 竞争临时性资源可能会引起死锁
  - 可以动态生成和消耗，一般不限制数量。如硬件中断、信号、消息、缓冲区内的数据。



信号量使用不当造成死锁

- 信号量的使用不当造成死锁
  - 生产者-消费者问题中，实现互斥的P操作在实现同步的P操作之前，就有可能导致死锁。（可以把互斥信号量、同步信号量也看作是一种抽象的系统资源）

```
producer (){
    while(1){
        生产一个产品;
        P(mutex);    ①
        P(empty);    ②
        把产品放入缓冲区;
        V(mutex);
        V(full);
    }
}
```

mutex的P操作在前

```
consumer (){
    while(1){
        P(full);        ③
        P(mutex);       ④
        从缓冲区取出一个产品;
        V(mutex);
        V(empty);
        使用产品;
    }
}
```

产生死锁的必要条件

- 只有4个条件都满足时，才会出现死锁。
  - ① 互斥：只有对互斥资源的争抢才会发生死锁。
  - ② 请求和保持：进程保持了至少一个资源，但又提出了新的资源请求，该资源又被其他进程占用。
  - ③ 不剥夺：进程已经占用的资源，未使用完，不能被剥夺。
  - ④ 环路等待：存在进程－资源环形链，即有进程集合{P0, P1, P2, ... Pn}, P0等待P1占用的资源，P1等待P2占用的资源.....Pn等待P0占用的资源。

注意！发生死锁时一定有循环等待，但是发生循环等待时未必死锁（循环等待是死锁的必要不充分条件）

死锁的处理方法

1. 预防死锁
   - 采用某种策略，限制并发进程对资源的请求，使系统在任何时刻都不同时满足死锁的四个必要条件
2. 避免死锁
   - 在资源的动态分配过程中，防止系统进入不安全状态。
3. 检测死锁
   - 允许系统进入死锁，但系统及时检测，并采取措施。
4. 解除死锁
   - 当检测到系统进入了死锁，采取措施解除。

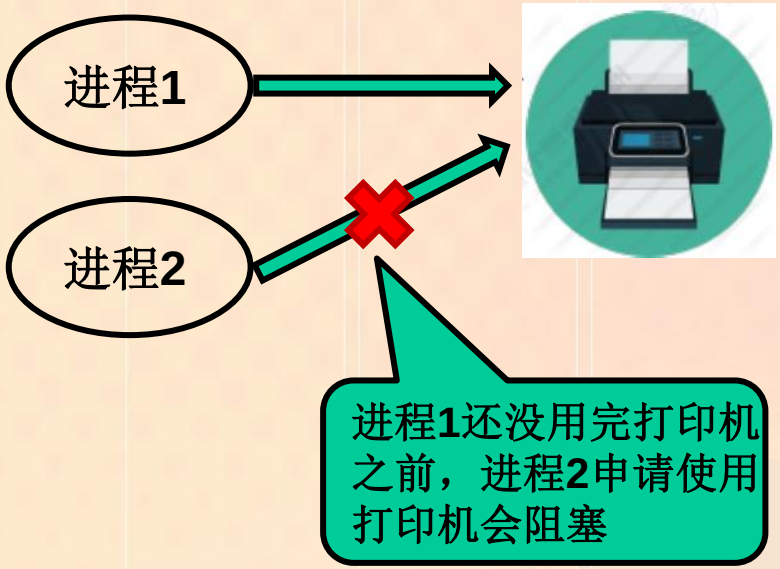
- 破坏互斥条件

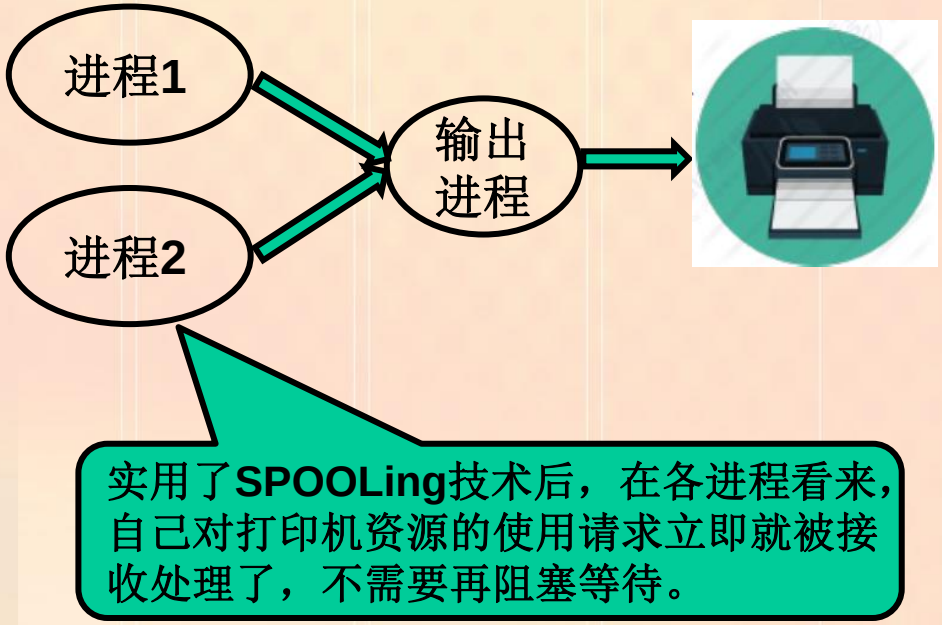
互斥条件：只有对必须互斥使用的资源的争抢才会导致死锁。

如果把只能互斥使用的资源改造为允许共享使用，则系统不会进入死锁状态。比如：SPOLLing技术。

操作系统可以采用SPOOLing技术把独占设备在逻辑上改造成共享设备。比如，用SPOOLing技术将打印机改造为共享设备...

3.6 预防死锁





该策略的缺点：并不是所有的资源都可以改造成共享使用的资源。并且为了系统安全，很多地方还必须保护这种互斥性。因此，很多时候都无法破坏互斥条件。

- 破坏“请求保持”条件

请求和保持条件：进程已经保持了至少一个资源，但又提出了新的资源请求，而该资源又被其他进程占有，此时请求进程被阻塞，但又对自己已有的资源保持不放。

可以采用静态分配方法，即进程在运行前一次申请完它所需要的全部资源，在它的资源未满足前，不让它投入运行。一旦投入运行后，这些资源就一直归它所有，该进程就不会再请求别的任何资源了。

优点：简单、易于实现、安全。

缺点：资源被严重浪费，降低了对资源的利用率，降低进程的并发程度；有可能无法预先知道所需资源。

- 破坏“不可剥夺”条件

不可剥夺条件：进程所获得的资源在未使用完之前，不能由其他进程强行夺走，只能主动释放。

1. 当一个已经保持了某些资源的进程，再提出新的资源请求而不能立即得到满足时，必须释放它已持有的资源。
2. 当某个进程需要的资源被其他进程所占有的时候，可以由操作系统协助，将想要的资源强行剥夺。需要考虑各进程的优先级。

缺点：

- 实现较复杂，需付出代价。
- 反复申请释放资源，降低系统吞吐率。

- 破坏“环路等待”条件

环路条件：存在一种进程资源的循环等待链，链中的每一个进程已获得的资源同时被下一个进程所请求。

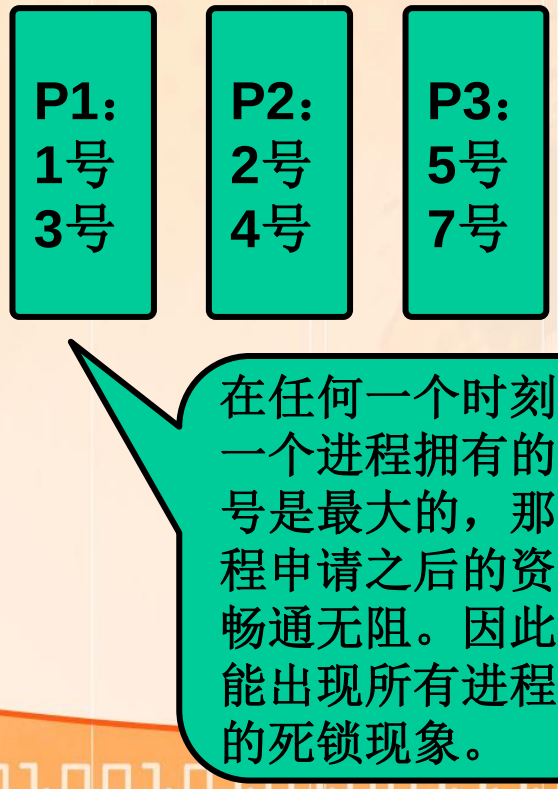
有序资源使用法：把资源分类按顺序排列，所有进程 对资源的请求 必须按照资源序号递增次序提出，同类资源(即编号相同的资源)一次申请完。保证不形成环路。

缺点：资源序号固定，限制新设备的增加；降低资源利用率；限制了用户简单、自主地编程。

3.6 预防死锁

原理分析：一个进程只有已占用小编号的资源时，才有资格申请更大编号的资源。按此规则，已持有大编号资源的进程不可能逆向地回来申请小编号的资源，从而就不会产生循环等待的现象。

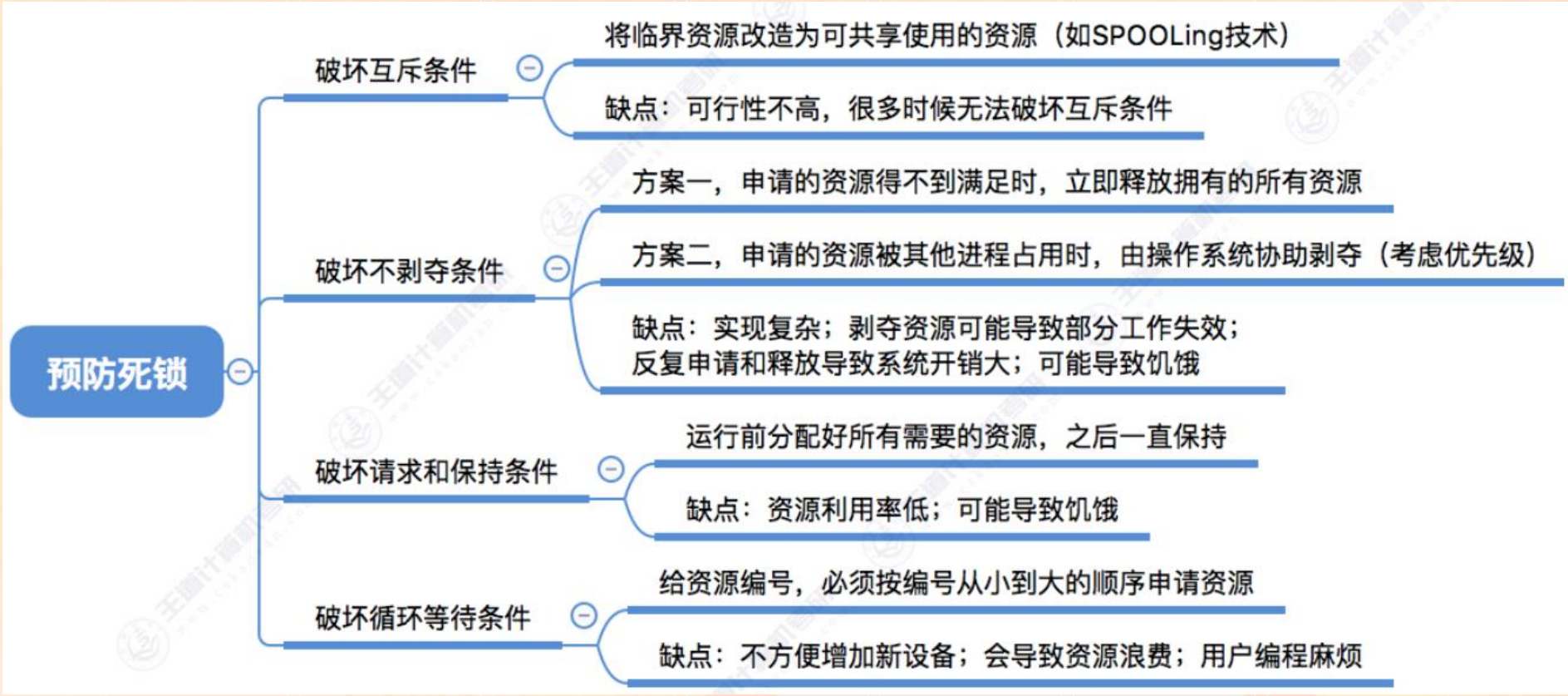
假设系统中共有10个资源，编号为1，2，.....10



该策略的缺点：

1. 不方便增加新的设备，因为可能需要重新分配所有的编号；
2. 进程实际使用资源的顺序可能和编号增递顺序不一致，会导致资源浪费；
3. 必须按照规定次序申请资源，用户编程麻烦。

3.6 预防死锁


预防死锁
破坏互斥条件
将临界资源改造为可共享使用的资源（如SPOOLing技术）
缺点：可行性不高，很多时候无法破坏互斥条件
破坏不剥夺条件
方案一，申请的资源得不到满足时，立即释放拥有的所有资源
方案二，申请的资源被其他进程占用时，由操作系统协助剥夺（考虑优先级）
缺点：实现复杂；剥夺资源可能导致部分工作失效；
反复申请和释放导致系统开销大；可能导致饥饿
破坏请求和保持条件
运行前分配好所有需要的资源，之后一直保持
缺点：资源利用率低；可能导致饥饿
破坏循环等待条件
给资源编号，必须按编号从小到大的顺序申请资源
缺点：不方便增加新设备；会导致资源浪费；用户编程麻烦

- 安全状态
  - 是指系统能按某种进程顺序（P1，P2，....Pn），为进程P_i分配其所需资源，直至进程的最大需求，使每个进程都可顺利完成。 则（P1，P2，....Pn）称为安全序列。
  - 若系统无法找到安全序列，则称系统处于不安全状态。

- 安全状态之例
  - 设系统共有12台磁带机，T_0时刻系统状态如下：

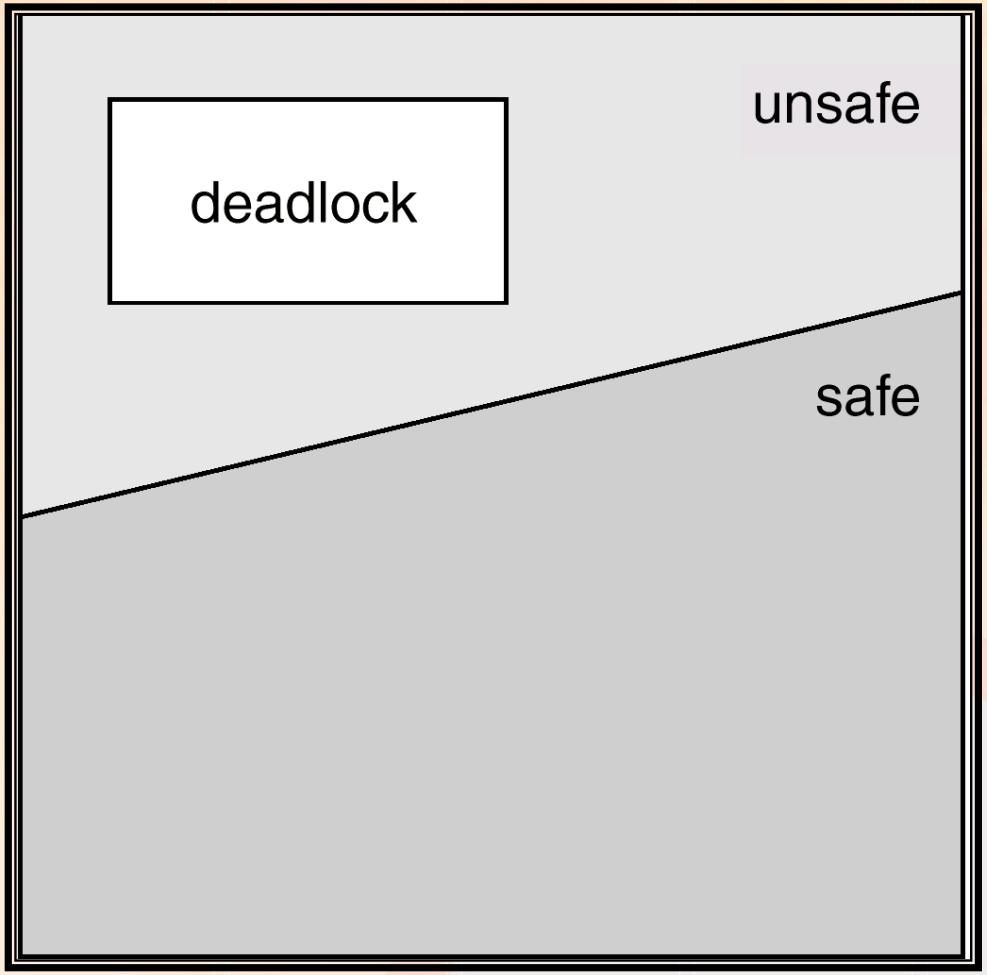
进程	最大需求	已分配	可用
P1	10	5	3
P2	4	2	
P3	9	2	

安全序列：〈P2, P1, P3〉，当前系统为安全状态

- 由安全状态向不安全状态的转换
  - 若不按照安全序列分配资源，系统可能会由安全状态进入不安全状态。
  - 例：T0时刻后，P3请求1台磁带机，系统分配给它，则无法再找到一个安全序列，进入不安全状态。

进程	最大需求	已分配	可用
P1	10	5	3（2）
P2	4	2	
P3	9	2（3）	

安全、不安全、死锁状态的关系



3.7.2 避免死锁--银行家算法

- 银行家算法（Dijkstra, 1965）问题
  - 在银行中，客户申请贷款的数量是有限的。银行家应尽量满足所有客户的贷款需求。
  - 银行家就好比操作系统，资金就是资源，客户就相当于要申请资源的进程。

- 核心思想：在进程提出资源申请时，先预判此次分配是否会导致系统进入不安全状态。如果会进入不安全状态，就暂时不答应这次请求，让该进程先阻塞等待。

3.7.2 避免死锁--银行家算法

1、银行家算法中的数据结构

设系统中有m类资源，n个进程

（1）可利用资源向量Available。含有m个元素的一维数组，每个元素代表一类可利用的资源数目。

如果Available[j]=k,表示系统中现有Rj类资源k个。

（2）最大需求矩阵Max，是一个n*m的矩阵，定义了系统中n个进程中的每一个进程对m类资源的最大需求。

如果Max[i,j]=k，则表示进程i，需要Rj类资源的最大数目为k。

（3）分配矩阵Allocation。为n*m的矩阵，它定义了系统中每类资源当前已分配给每个进程的资源数。Allocation[i,j]=K,表示进程i当前已分得Rj类资源的数目为K。

（4）需求矩阵Need。为n*m的矩阵，表示每个进程尚需的各类资源数。Need[i,j]=K, 表示进程i还需要Rj类资源K个，方能完成其任务。

三个矩阵间的关系：Need[i,j]=Max[i,j]-Allocation[i,j]

2、银行家算法的处理步骤

设$Request_i[j]=k$,表示进程Pi需要k个Rj类型的资源

（1）如果$Request_i[j] <= Need[i,j]$,便转向步骤2；否则认为出错，因为它所请求的资源数已超过它所需要的最大值。

（2）如果$Request_i[j] <= Available[j]$,便转向步骤3；

否则，表示尚无足够资源，Pi需等待。

（3）系统试探着把资源分配给进程Pi，并修改下面数据结构中数值

$Available[j]:= Available[j]- Request_i[j];$

$Allocation[i,j]:=Allocation[i,j]+Request_i[j];$

$Need[i,j]:=Need[i,j]-Request_i[j];$

3.7.2　避免死锁--银行家算法

（4）系统执行安全性算法，检查此次资源分配后系统是否处于安全状态以决定是否完成本次分配。若安全，则正式分配资源给进程Pi；否则，不分配资源，让进程Pi等待。

3、安全性算法

（1）设置两个向量：

工作向量**work**：表示系统可提供给进程继续运行所需的各类资源数目，含有m个元素的一维数组，初始时，work:=Available;

Finish: 含n个元素的一维数组，表示系统是否有足够的资源分配给n个进程，使之运行完成。开始时先令Finish[i]:=false (i=1..n);当有足够资源分配给进程i时，再令Finish[i]:=true。

（2）从进程集合中找到一个能满足下述条件的进程：

Finish[i]=false;Need[i,j]<=work[j]; 若找到，执行步骤(3)，否则执行步骤(4)。

（3）当进程P_i获得资源后，可顺利执行，直至完成，并释放出分配给它的资源，故应执行：

work[j]:= work[j]+ Allocation[i,j] ;

Finish[i]:=true;

goto step (2);

（4）如果所有进程的**Finish[i]=true**都满足，则表示系统处于安全状态；否则，系统处于不安全状态。

银行家算法之例

- 5个进程：P_0~P_4；3类资源：A (10个实例)，B (5个实例)，C (7个实例).
- 假设在 T_0时刻,系统状态如下：

	Allocation			Max			Available		
	A	B	C	A	B	C	A	B	C
P_0	0	1	0	7	5	3	3	3	2
P_1	2	0	0	3	2	2			
P_2	3	0	2	9	0	2			
P_3	2	1	1	2	2	2			
P_4	0	0	2	4	3	3			

银行家算法之例

- 需要导出Need矩阵。 Need矩阵定义为：Need = Max – Allocation。则得到：

	Allocation			Max			Need			Available		
	A	B	C	A	B	C	A	B	C	A	B	C
P_0	0	1	0	7	5	3	7	4	3	3	3	2
P_1	2	0	0	3	2	2	1	2	2			
P_2	3	0	2	9	0	2	6	0	0			
P_3	2	1	1	2	2	2	0	1	1			
P_4	0	0	2	4	3	3	4	3	1			

- 用安全算法可证明系统目前是安全的，因为存在安全序列：< P_1, P_3, P_4, P_2, P_0>.

银行家算法之例

问：

(1)考察系统在T_0时刻的安全性?

(2)P_1请求资源$Request_1(1, 0, 2)$?

(3)P_4请求资源$Request_4(3, 3, 0)$?

(4)P_0请求资源$Request_0(0, 2, 0)$?

银行家算法之例

- 假设这时进程P_1发出资源请求：

  Request1 = (1,0,2)

- 首先检查：

  Request1 ≤ Need1　(1,0,2) ≤ (1,2,2)

- 再检查 Request1 ≤ Available (1,0,2) ≤　(3,3,2)
  ⇒ 即满足申请资源的条件。

- 假设将资源分配给它，则得到如下的新状态：

银行家算法之例

	Allocation			Max			Need			Available		
	A	B	C	A	B	C	A	B	C	A	B	C
P_0	0	1	0	7	5	3	7	4	3	2	3	0
P_1	3	0	2	3	2	2	0	2	0			
P_2	3	0	2	9	0	2	6	0	0			
P_3	2	1	1	2	2	2	0	1	1			
P_4	0	0	2	4	3	3	4	3	1			

- **执行安全算法可以找到安全序列：**
  $<P_1, P_3, P_4, P_0, P_2>$
- **接下来进程P_4请求资源 (3,3,0) 能够满足它吗?**
- **若是进程P_0请求资源 (0,2,0)能满足它吗?**

银行家算法的特点

- 允许互斥、部分分配和不可抢占，可提高资源利用率；
- 要求事先说明最大资源要求，在现实中很困难。

作业

- P128　　31（第四版）