第三章 处理机调度与死锁

3.1 处理机调度的基本概念

3.1.1 高、中、低三级调度

高级调度（作业调度、长程调度、接纳调度）

- 概念：
  - 将外存作业调入内存，创建PCB等，插入就绪队列。
  - 一般用于批处理系统，分/实时系统一般直接入内存，无此环节。

- 调度特性
  - 接纳作业数（内存驻留数）

    太多———>　　周转时间T长

    太少———>　　系统效率低
  - 接纳策略：即采用何种调度算法：FCFS、短作业优先等

- **低级调度（进程调度，短程调度）**

  主要是由分派程序（Dispatcher）分派处理机。

  - **非抢占方式：简单，实时性差(如win31)**
  - **抢占方式**

    （1）时间片原则

    （2）优先权原则

    （3）短作业优先原则。

- **中级调度（中程）**

  为提高系统吞吐量和内存利用率而引入的内外存对换功能。

运行频率：低>中>高。

3.1.2 调度队列模型

1. 仅有进程调度的调度队列模型

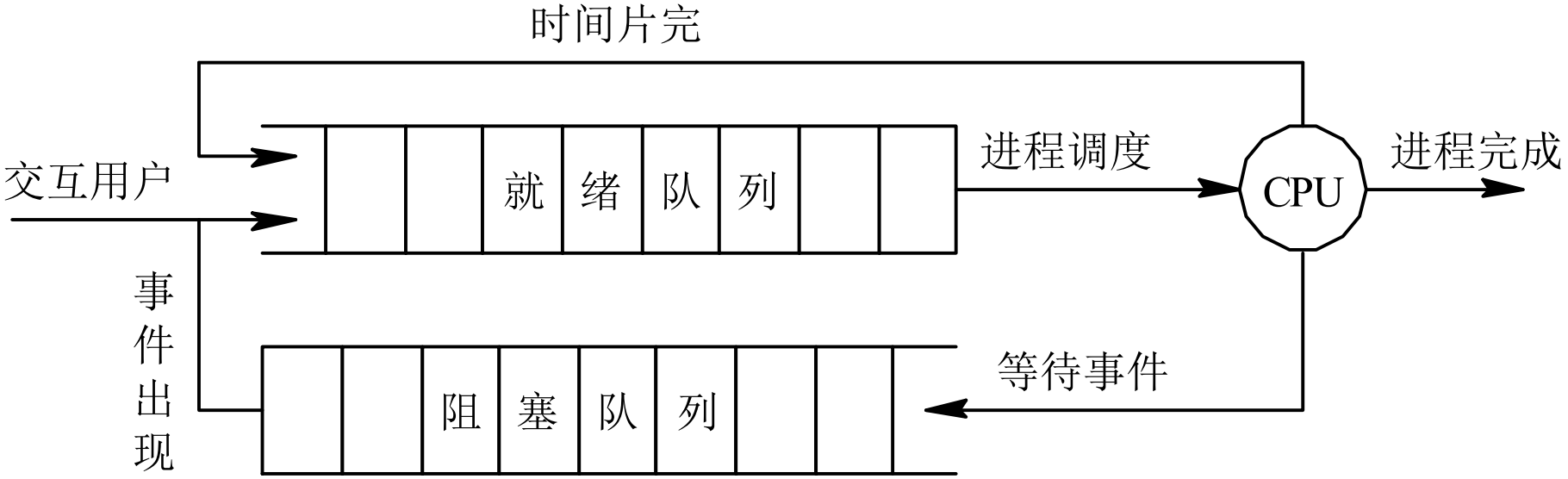
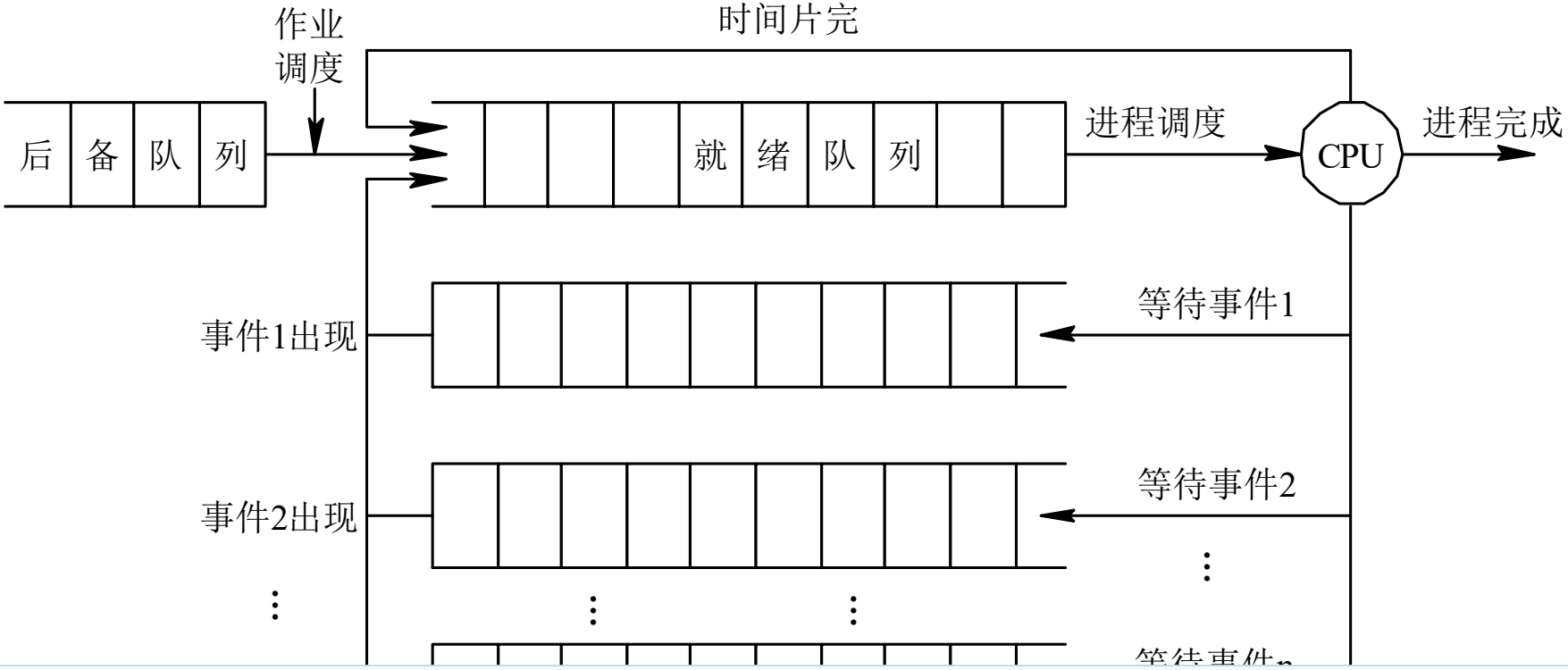


图 3 - 1 仅具有进程调度的调度队列模型

2. 具有高级和低级调度的调度队列模型



该模型与上一模型的主要区别在于如下两个方面。

(1) 就绪队列的形式。

(2) 设置多个阻塞队列。

3. 同时具有三级调度的调度队列模型

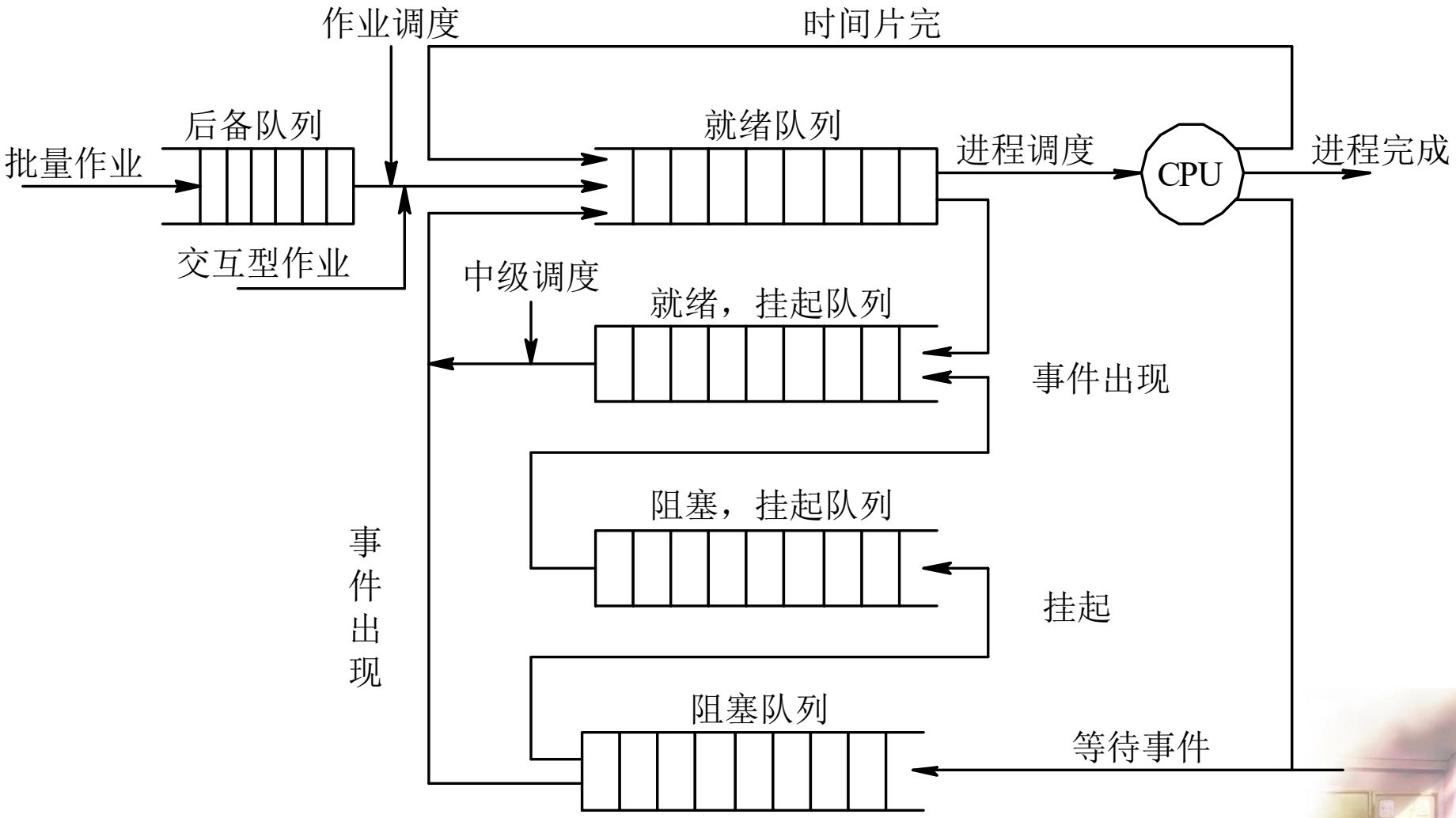


图 3-3 具有三级调度时的调度队列模型

3.1.3 选择调度方式和算法的若干准则

1. 面向用户的准则

- 周转时间短（常用于批处理系统）
  - 概念：作业从提交——>完成的时间。分为：
    （1）驻外等待调度时间
    （2）驻内等待调度时间
    （3）执行时间
    （4）阻塞时间

- 平均周转时间

$$T = \frac{1}{n}[\sum_{i=1}^{n} T_i]$$

- 平均带权

$$W = \frac{1}{n}[\sum_{i=1}^{n} \frac{T_i}{T_s}]$$

可见带权w越小越好，Ts为实际服务时间。

- 响应时间快：（对交互性作业）

  概念：键盘提交请求到首次响应时间

  （1）输入传送时间

  （2）处理时间

  （3）响应传送时间

- 截止时间的保证（特别于实时系统）
- 优先权准则（即需要抢占调度）

2. 面向系统的准则

- 吞吐量高（特别于批处理）：单位时间完成作业数
- 处理机利用率好：（因CPU贵，特别于大中型多用户系统）
- 各类资源的平衡利用。

3.2 调度算法

3.2.1先来先服务和短作业（进程）优先调度算法

进程名	到达时间	服务时间	开始执行时间	完成时间	周转时间	带权周转时间
A	0	1	0	1	1	1
B	1	100	1	101	100	1
C	2	1	101	102	100	100
D	3	100	102	202	199	1.99

先来先服务例子

1. FCFS

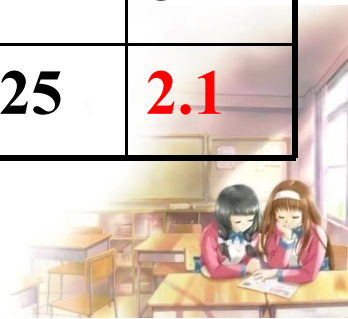
- 特点：简单，有利于长作业。

2. 短作业进程优先调度算法：SJ(P)F

- 降低了平均周转时间和平均带权周转时间（从而提高了系统吞吐量）
- 对长作业不利，有可能得不到服务（饥饿）
- 估计时间不易确定

FCFS和SJF比较

	进程名	A	B	C	D	E	平均
	到达时间	0	1	2	3	4	
	服务时间	4	3	5	2	4	
FCFS	完成时间	4	7	12	14	18	
	周转时间	4	6	10	11	14	9
	带权周转时间	1	2	2	5.5	3.5	2.8
SJF	完成时间	4	9	18	6	13	
	周转时间	4	8	16	3	9	8
	带权周转时间	1	2.67	3.1	1.5	2.25	2.1



3.2.2高优先权优先调度算法

1. 优先权调度算法类型
   - 非抢占式优先权算法
   - 抢占式优先权算法，实时性更好。

2. 优先权类型：
   - 静态优先权：
     - 进程优先权在整个运行期不变。
     - 确定优先权依据
       （1）进程类型
       （2）进程对资源的需求；
       （3）根据用户需求。
     - 特点：简单，但低优先权作业可能长期不被调度。

* 动态优先权：
  - 如：优先权随执行时间而下降，随等待时间而升高。
  - 响应比 Rp=（等待时间＋服务时间）/服务时间　作为优先权
  - 优点：长短兼顾　　缺点：需计算 Rp

3. 高响应比优先算法：

响应比Rp=（tw+ts）/ts

（1）短作业RP大。

（2）ts（要求服务时间）相同的进程间相当于FCFS。

（3）长作业等待一段时间仍能得到服务。

3.2.3 基于时间片的轮转调度算法

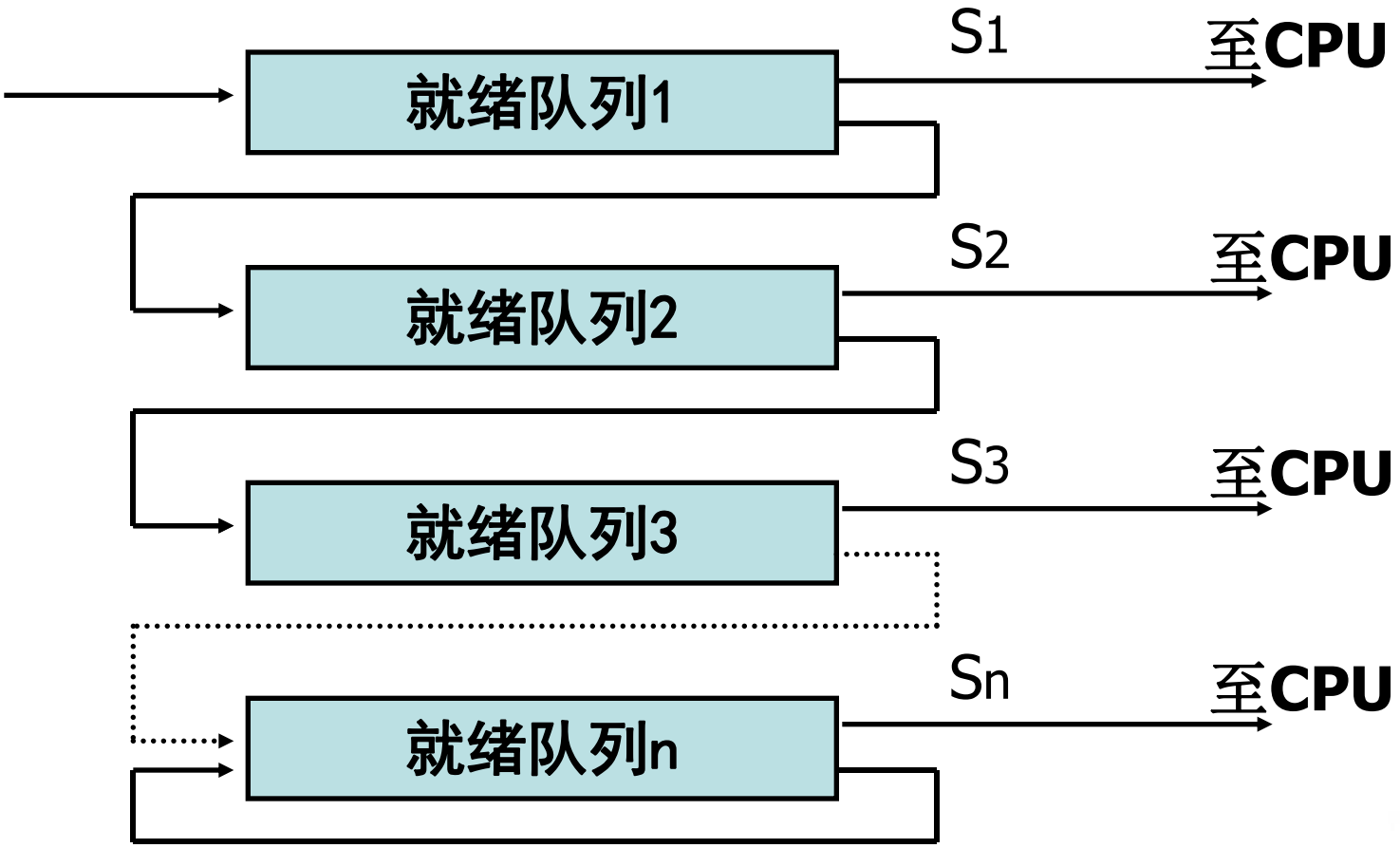
1. 时间片轮转算法

- 时间片大小q的确定
  - 太大：退化为FCFS；
  - 太小：系统开销过大
- 系统对响应时间的要求；T= nq
- n是就绪队列中进程的数目；
- 系统的处理能力：（应保证一个时间片处理完常用命令）

勘误：课本图3-3的周转时间和平均带权周转时间有误。

q=1时分别为3.25和3.43，q=4时分别为3.25和2.7。

2. 多级反馈队列调度



时间片：**S1<S2<S3**

* 特点：长、短作业兼顾，有较好的响应时间
  - 短作业一次完成；
  - 中型作业周转时间不长；
  - 大型作业不会长期不处理。

典型问题分析

3.3 实时调度

3.3.1 实时调度的基本条件

3.3.2 实时调度的类型

3.3.3 实时调度算法

3.3.1 实现实时调度的基本条件

1．提供必要的调度信息

（1）就绪时间；

（2）开始/完成截止时间；

（3）处理时间；

（4）资源要求；

（5）优先级；

2．系统处理能力强

$$\sum_{i=1}^{m}\frac{C_i}{P_i}\le 1 \qquad \sum_{i=1}^{m}\frac{C_i}{P_i}\le N$$

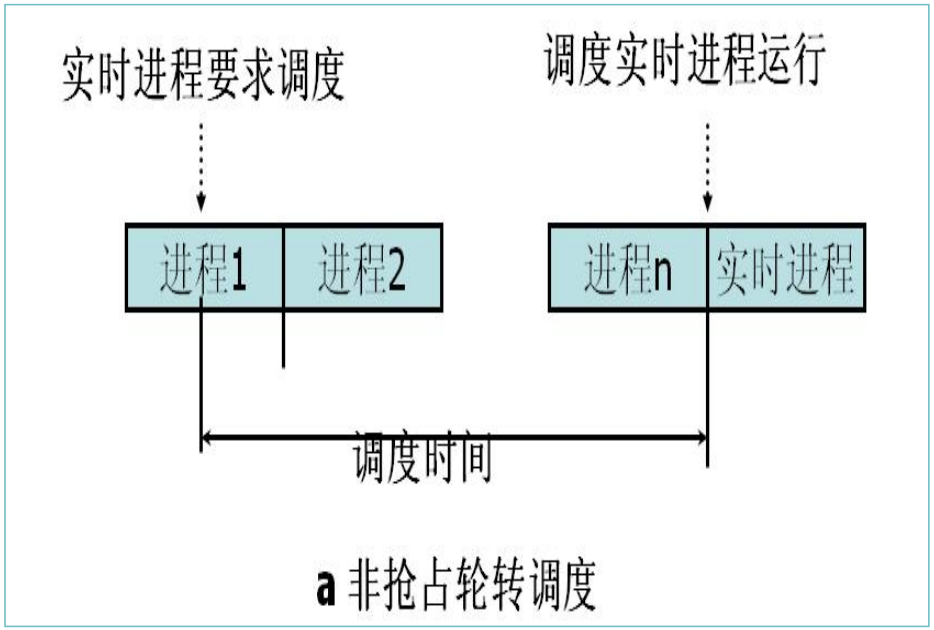
Ci为处理时间，Pi为周期时间（基于周期性实时任务）

3. 采用抢占调度方式
   - 剥夺方式：一般都采用此
   - 非剥夺方式（实现简单）：一般应使实时任务较小，以及时放弃CPU。
4. 具有快速切换机制
   - 具有快速响应外部中断能力。
   - 快速任务分派

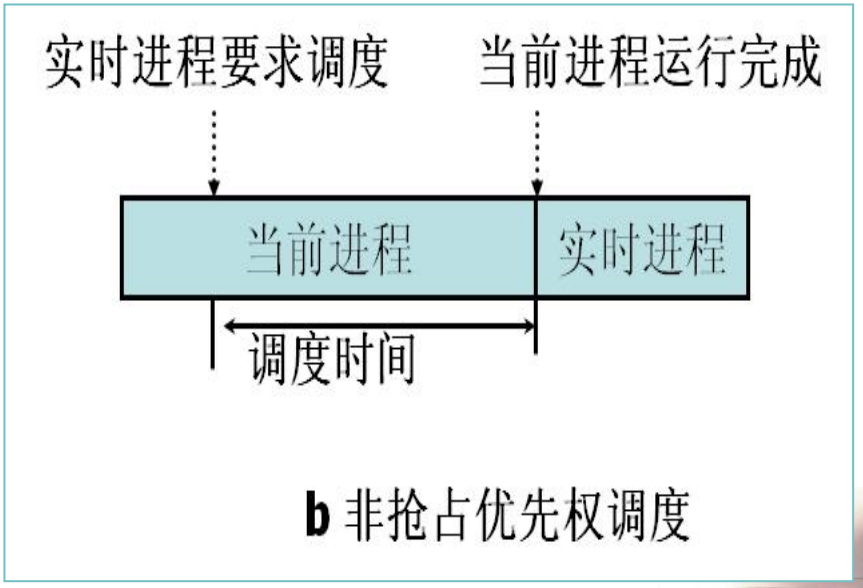
3.3.2 实时调度算法的分类

1非抢占式调度算法

* 时间片轮转　　秒级
* 非抢占优先权（协同）　　秒-毫秒级



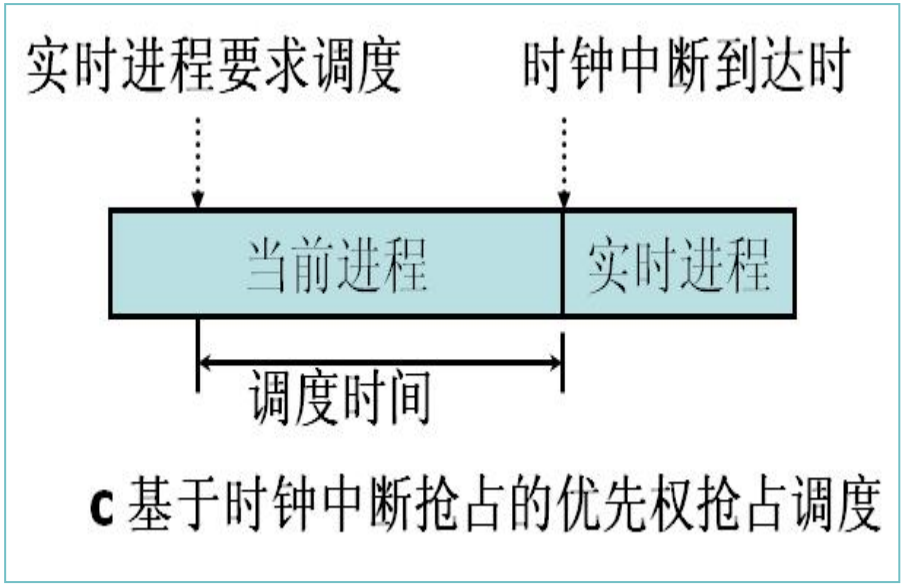
a 非抢占轮转调度



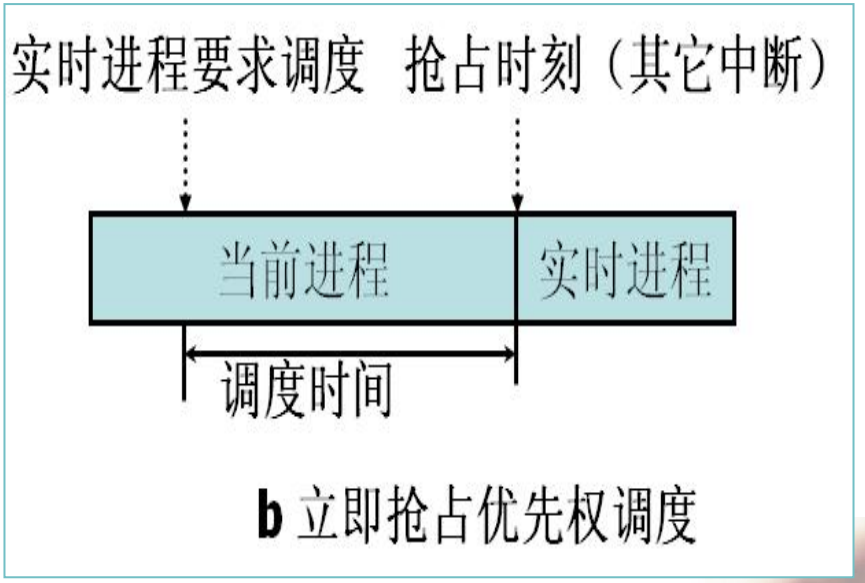
b 非抢占优先权调度

2抢占式调度算法

* 时钟中断抢占优先权　　毫秒级
  - 基于抢占点抢占
* 立即抢占immediate preemption　　毫秒-微秒级
  - 只要不在临界区即抢占（中断引发）



c 基于时钟中断抢占的优先权抢占调度



b 立即抢占优先权调度

3.3.3 常用的几种实时调度算法

1. 最早截止时间优先EDF（earliest deadline first)算法

- 根据任务的开始截止时间来确定任务的优先级
- 截止时间越早，优先级越高
- 可以是抢占式或非抢占式

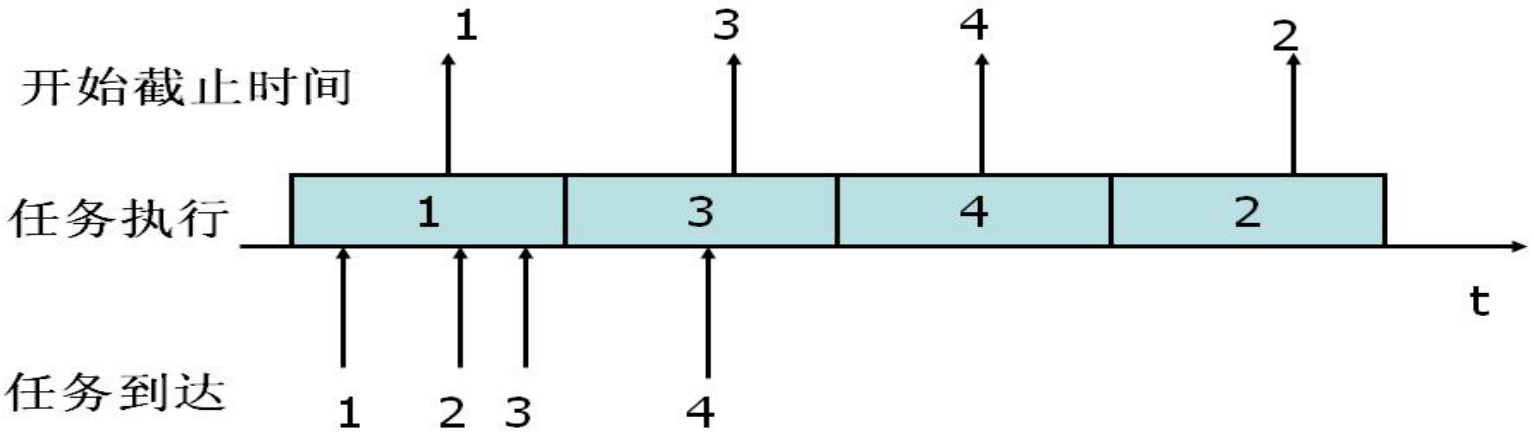


图3－7 EDF算法用于非抢占调度方式

2. 最低松弛度优先LLF算法

- 松弛度：
  - 根据任务紧急(或松弛)的程度，来确定任务的优先级。

    紧急程度(松弛程度)=完成截止时间-处理时间-当前时间

    例如：若A进程需在200ms时完成，其本身运行需要100ms，当前时刻是10ms，则A的松弛度为：200－100－10＝90

  - 主要用于可抢占的调度方式中

最低松弛度优先LLF算法举例

在一个实时系统中，有两个周期性实时任务A和B，任务A要求每 20 ms执行一次，执行时间为 10 ms；任务B只要求每50 ms执行一次，执行时间为 25 ms。

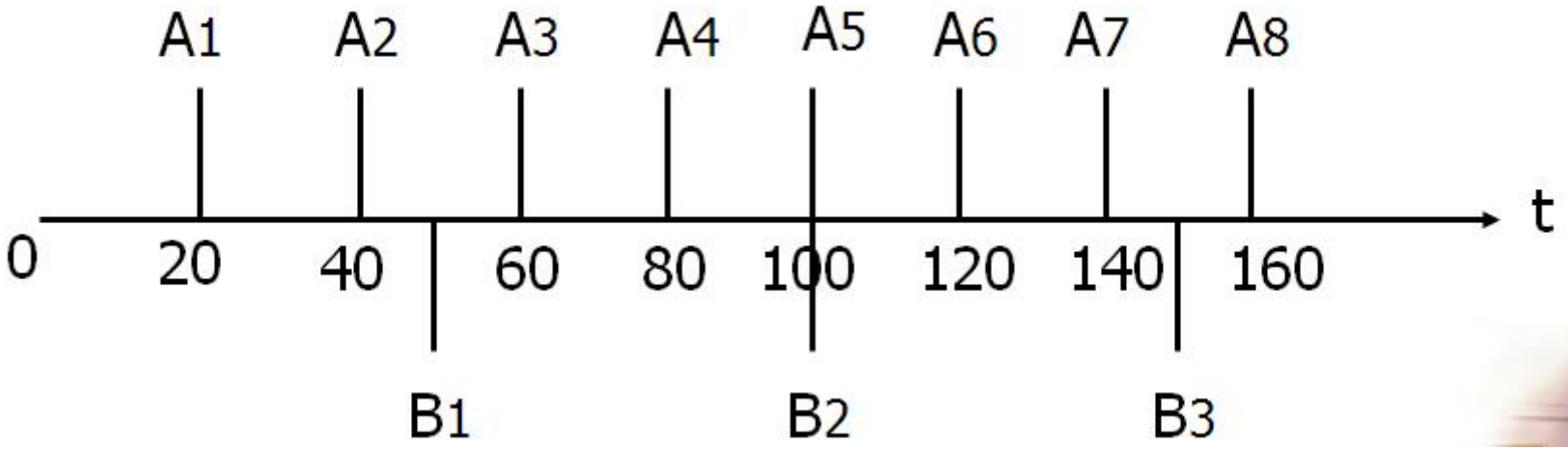


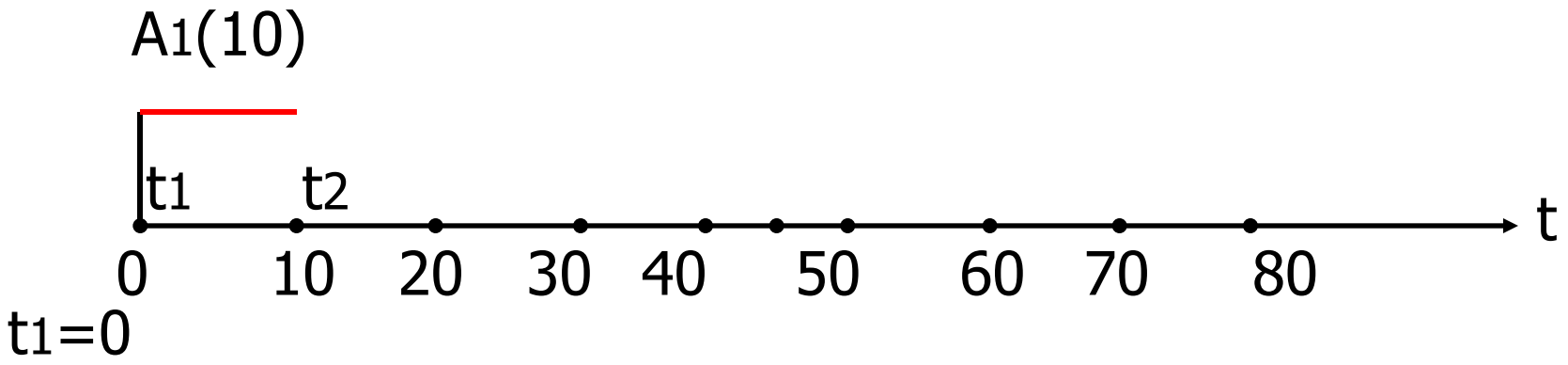
图3－8 A/B任务每次必须完成的时间

t_1=0时，

A_1的松弛度=20-10=10ms；

B_1的松弛度=50-25=25 ms，

故调度程序应先调度A_1执行10ms.



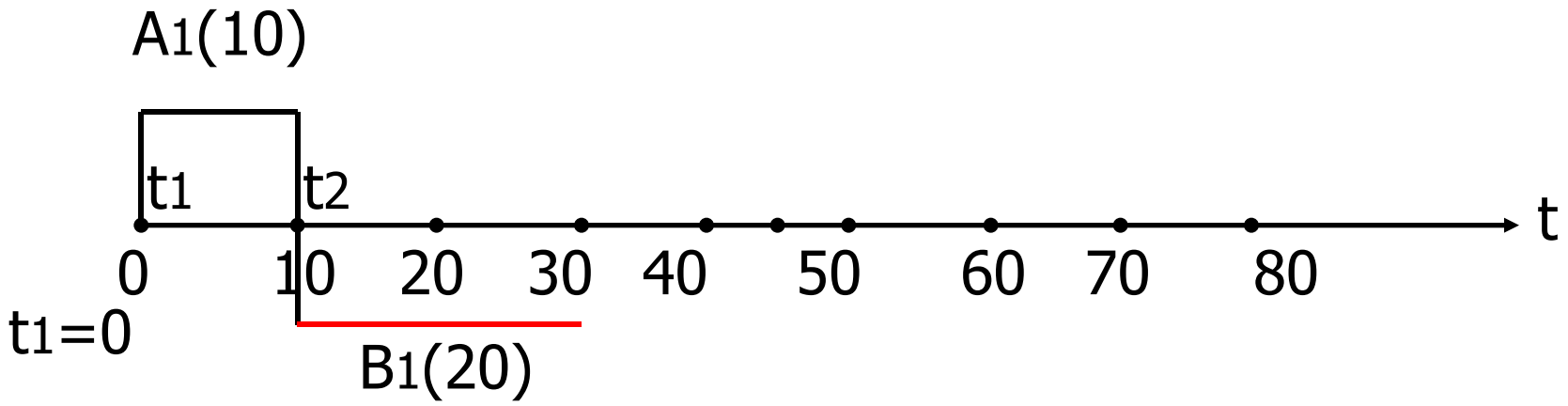
任务A要求每 20 ms执行一次，执行时间为 10 ms；任务B只要求每50 ms执行一次，执行时间为 25 ms。

t2=10 ms时，

A2的松弛度=40-10-10=20 ms

B1的松弛度=50-25-10=15ms

任务A2未到达，故调度程序应先调度B1执行20ms



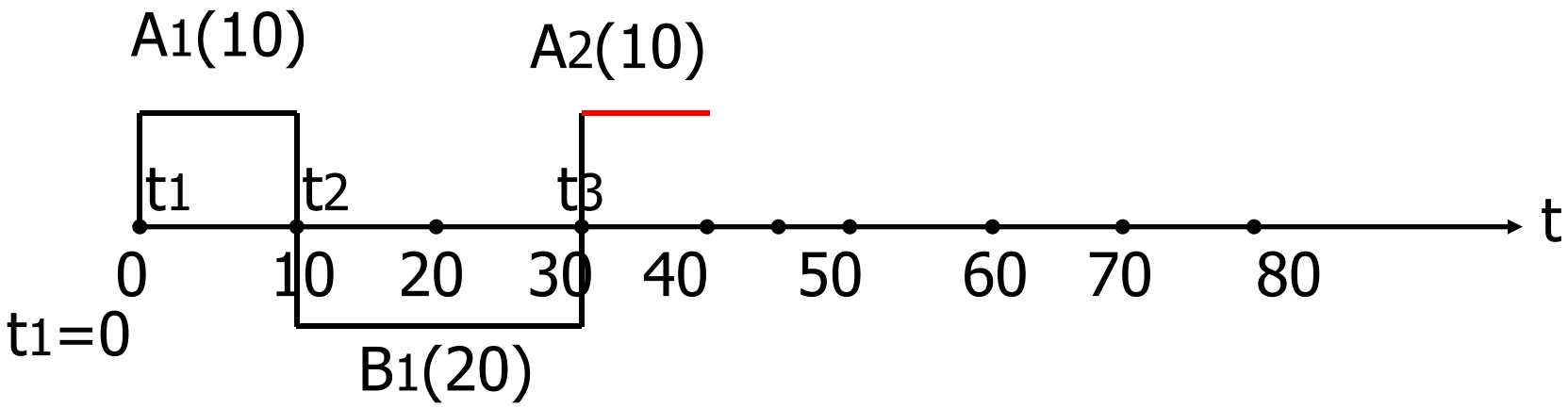
任务A要求每 20 ms执行一次，执行时间为 10 ms；任务B只要求每50 ms执行一次，执行时间为 25 ms。

t3=30 ms时，

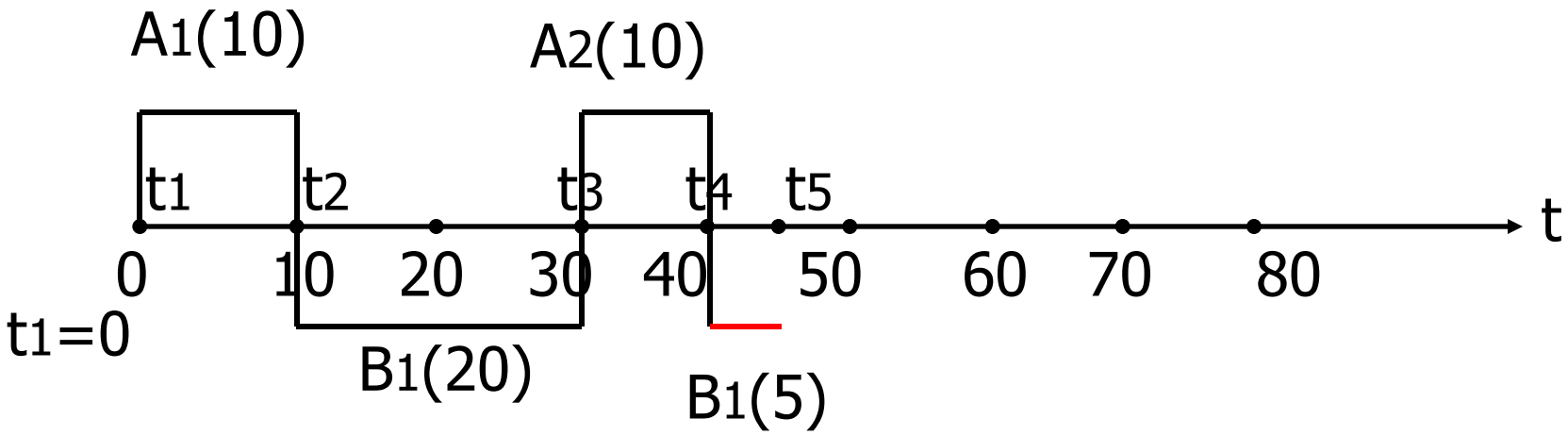
A2的松弛度=40-10-30=0 ms

B1的松弛度=50-5-30=15ms

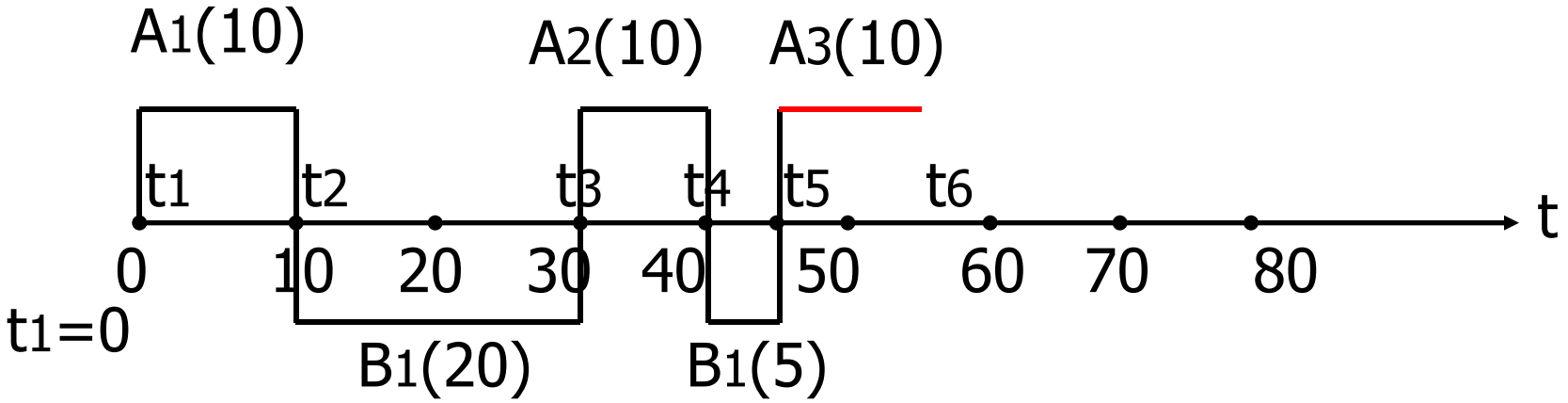
故调度程序应抢占B1调度A2执行10ms。



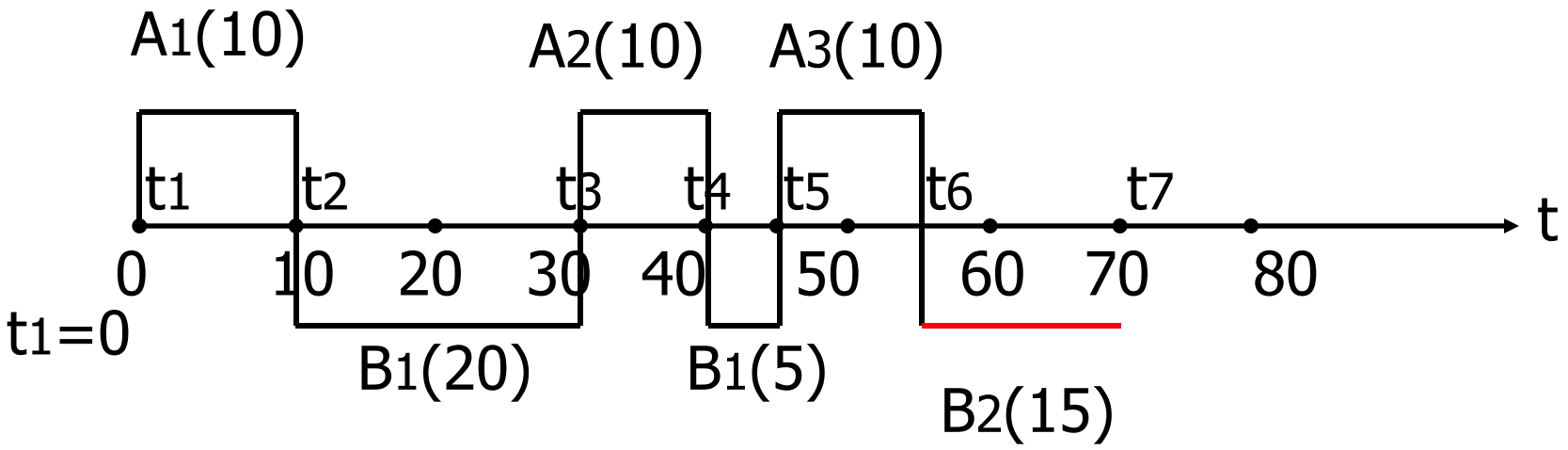
t_4=40 ms时，
A_3的松弛度为10 ms(即60-10-40)，
B_1的松弛度仅为5 ms(即50-5-40)，
故又应重新调度B_1执行5ms。



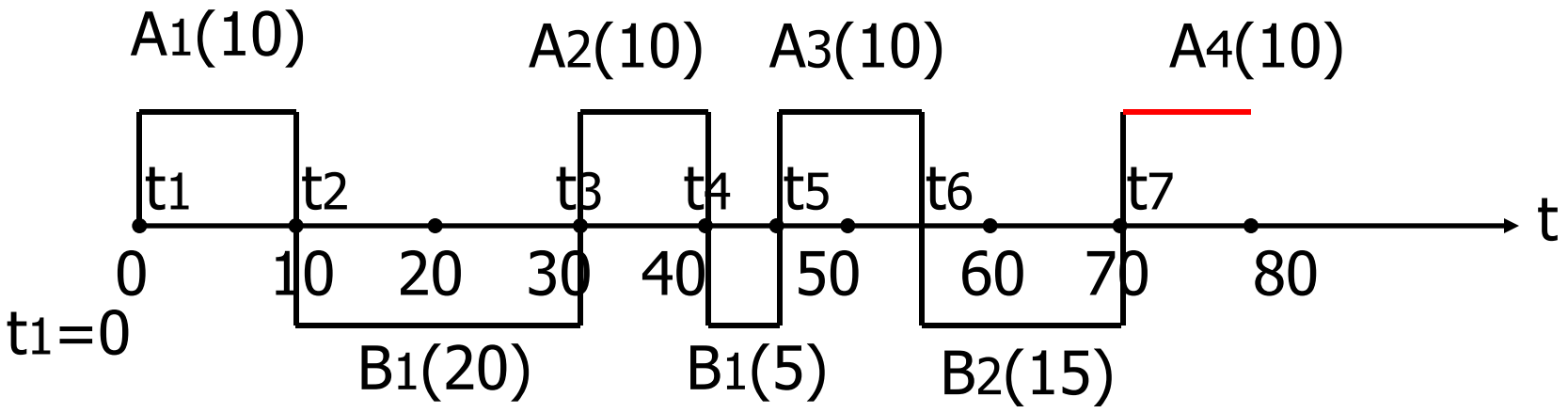
t_5=45 ms时，B_1执行完成，
A_3的松弛度已减为5 ms(即60-10-45)
B_2的松弛度为30 ms(即100-25-45)，
于是又应调度A_3执行10ms。



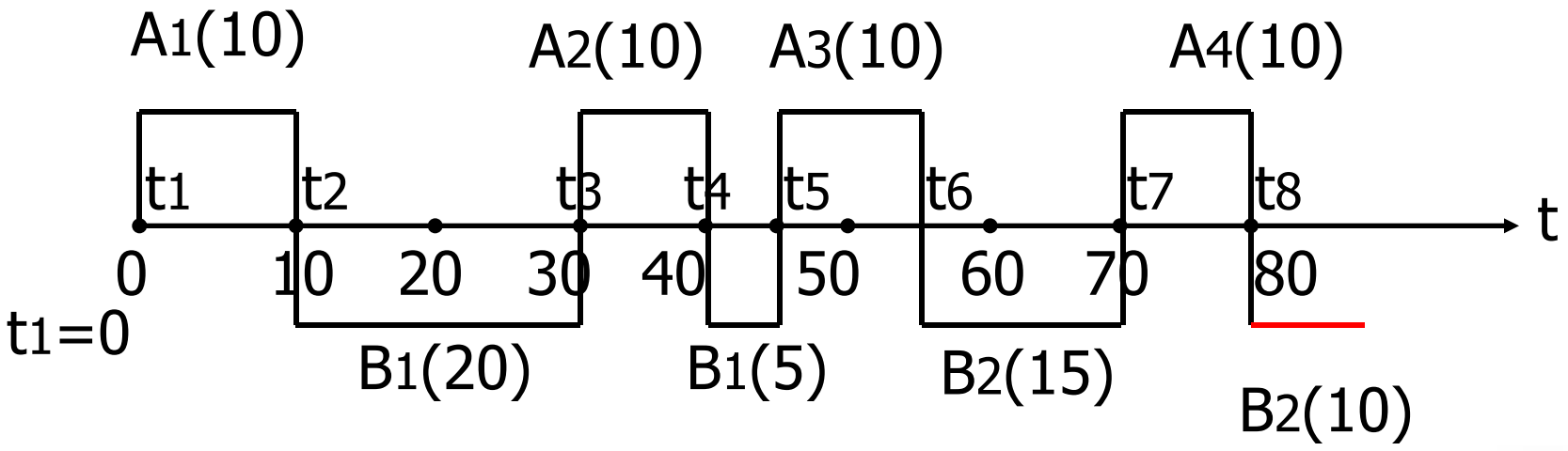
t6=55ms时，A3执行完毕，
A4的松弛度为15 ms(即80-10-55)
B2的松弛度为20 ms(即100-25-55)，
任务A4尚未到达，故调度B2执行15ms。



t_7=70 ms时，
A_4的松弛度已减至0 ms(即80-10-70)，
B_2的松弛度为20 ms(即100-10-70)，
故此时调度又应抢占B_2的处理机而调度A_4执行10ms。



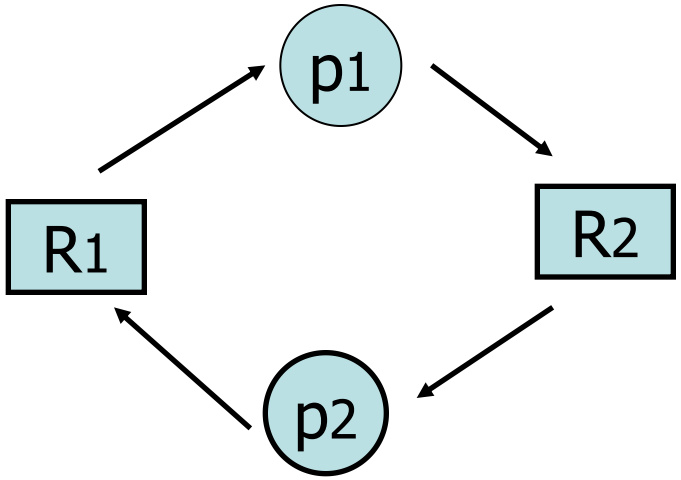
t8=80 ms时，A4执行完，
B2的松弛度=100-15-80=5ms
A5的松弛度=100-10-80=10ms
故调度程序调度B2执行10ms。



3.5 产生死锁的原因和必要条件

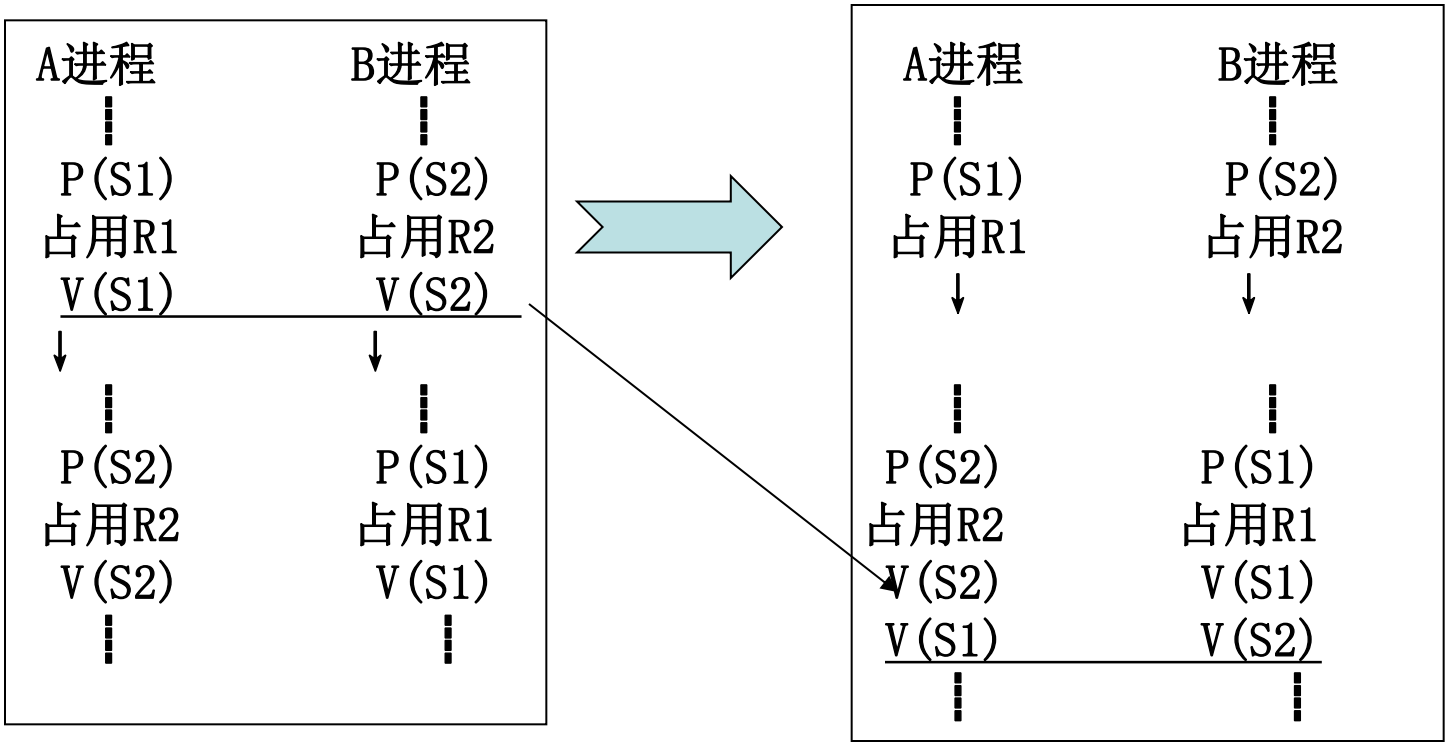
3.5.1产生死锁的原因

1. 竞争资源引起死锁。
   - 可剥夺（CPU、内存，）和非剥夺性（打印机，磁带机）资源
   - 竞争非剥夺性资源——可造成死锁
   - 竞争临时性资源 ——可造成死锁



2. 进程推进顺序不当引起死锁

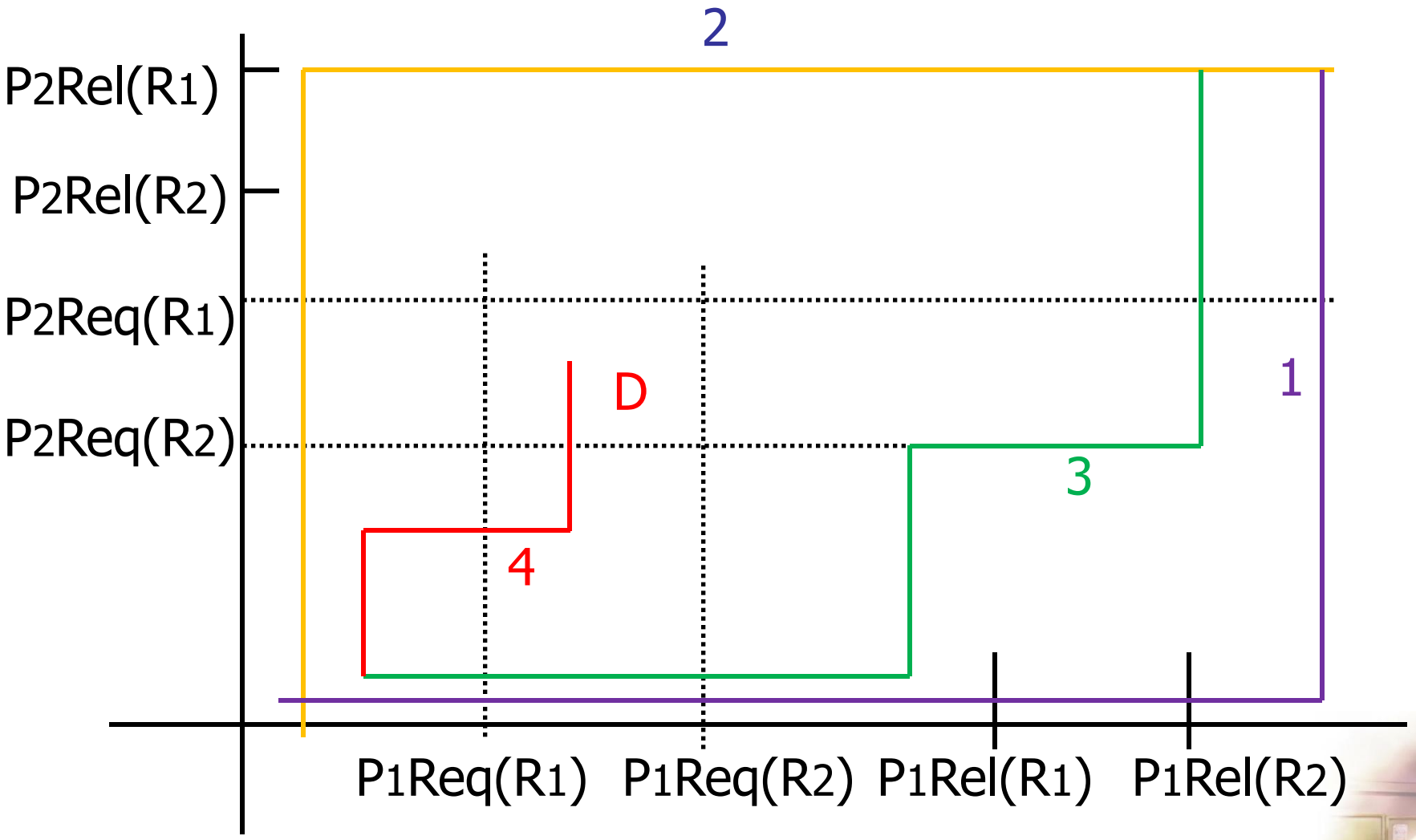
例：设系统有一台打印机（R1），一台输入设备(R2)，两进程共用这两台设备。用信号量S1,S2表示R1,R2资源，初值1。

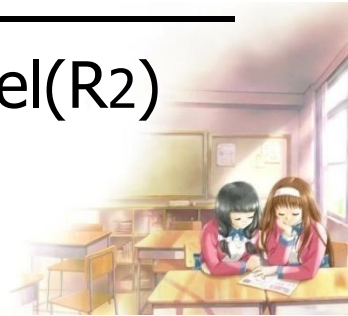


推进顺序合法　　顺序不合法

从例中可看出：A进程占用打印机，又想得到输入设备，而输入设备正被B进程占用，B进程在未释放输入设备时又要求占用正被A进程占用的打印机，则两进程无法运行，进入死锁状态。

进程推进顺序不当引起死锁





3.5.2 产生死锁的必要条件

1. 互斥条件（资源的临界性）
2. 请求和保持条件
3. 不剥夺条件
4. 环路等待

3.5.3 处理死锁的基本方法

1. 预防：破坏4个必要条件之一；

   有效，但使资源利用率低

2. 避免：防止进入不安全状态；

   允许动态申请资源，银行家算法

3. 检测：检测到死锁再清除；

4. 解除：与“检”配套。

3.6 死锁预防和避免

3.6.1 死锁预防

预防是采用某种策略，限制并发进程对资源的请求，使系统在任何时刻都不满足死锁的必要条件。

- 摒弃互斥条件：互斥是资源固有属性，不能避免。
- 摒弃请求和保持条件：资源的一次性分配。

全分配，全释放（AND）

缺点：（1）延迟进程运行

（2）资源严重浪费

- 摒弃“不剥夺”条件：新申请不能满足则释放已获得资源。
  缺点：增加系统开销，且进程前段工作可能失效。

- 摒弃“环路”条件：资源的有序分配法。

  资源有序分配法：为资源编号，申请时需按编号进行。

  缺点：

  （1）新增资源不便;（原序号已排定）

  （2）用户不自由;

  （3）资源与进程使用顺序不同造成浪费.

3.6.2 系统的安全状态

- 安全状态是在“避免死锁”方法中的判断条件。

- 安全状态
  - 按某种顺序并发进程都能达到获得最大资源而顺序完成的序列为安全序列。
  - 能找到安全序列的状态为安全状态。

- 银行家算法：一种能够避免死锁的调度算法。

银行家算法（Dijkstra，1965）问题

一个银行家把他的固定资金（capital）代给若干顾客。只要不出现一个顾客借走所有资金后还不够，银行家的资金应是安全的。银行家需一个算法保证借出去的资金在有限时间内可收回。

❖ 假定：顾客借款分成若干次进行；并在第一次借款时，能说明他的最大借款额。

具体算法

S1　某个客户提出贷款请求；
S2　假设批准该请求，将得到系统状态T；
S3　判断状态T是否安全，
　　如果安全，则批准该请求，转S1；
　　如果不安全，则不批准该请求，延期到以后处理，转S1；

判断一个状态T是否安全

S1　银行家检查一下，看他手里的资源能否满足某个客户的请求（剩余的最大限额）；
S2　如果可以，则该客户的贷款请求已经满足，因此他偿还了所有贷款。转S1；
S3　如果到最后，所有的贷款都能偿还，则状态T就是安全的，否则就是不安全的

银行家算法的特点

- 允许动态申请资源，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；

- 一个例子：

设可用资源数为12，t时刻状态如下：

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

若P3再申请一个资源：

不安全

预分配

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

取消分配

安全与不安全的区别是：从安全状态出发，系统能够保证所有进程都能完成；而从不安全状态出发，就没有这种保证。

3.6.3 利用银行家算法避免死锁

1. 数据结构

- available[j]=k　系统现有R_j类资源k个；
- max[i,j]=k　进程i需要R_j的最大数k个；
- alloc[i,j]=k　进程i已得到R_j类资源k个；
- need[i,j]=k　进程i还需要R_j类资源k个；

  有：need[i,j]= max[i,j]－alloc[i,j]

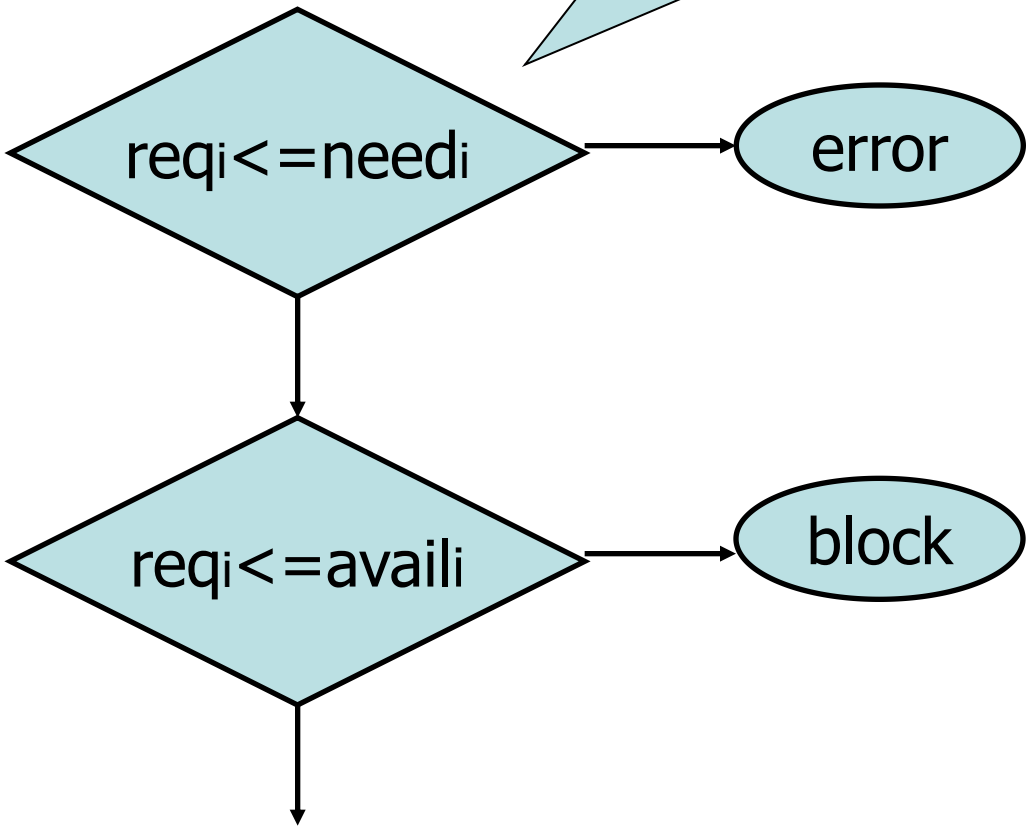
- $Request_i[j]=k$　表示进程i需要k个R_j类型的资源；
- Work　表示进程继续运行所需的各类资源数目

  在执行安全算法开始时，Work：=Available;

- finish[i]　布尔量，表进程i能否顺序完成。

2. 银行家算法

首先，检查请求是否合法和可满足？



试探分配，修改相关数据结构

avail=avail-reqi
alloci=alloci+reqi
needi=needi-reqi

finish[i]=.F.
needi<=work

work=work+alloci
finish[i]=.T.

寻找安全序列：有则完成分配，否则取消试探分配

3 实例

假定系统中有五个进程{P_0，P_1，P_2，P_3，P_4}和三类资源{A，B，C}，各种资源的数量分别为10、5、7，在T_0时刻的资源分配情况如下图所示。

	Max			Allocation			Need			Available		
	A	B	C	A	B	C	A	B	C	A	B	C
p0	7	5	3	0	1	0	7	4	3	3	3	2
p1	3	2	2	2	0	0	1	2	2			
p2	9	0	2	3	0	2	6	0	0			
p3	2	2	2	2	1	1	0	1	1			
p4	4	3	3	0	0	2	4	3	1			

T0时刻的安全序列

安全

	Work A B C	Need A B C	Alloc A B C	Work+alloc A B C	Finish
p1	3 3 2	1 2 2	2 0 0	5 3 2	true
p3	5 3 2	0 1 1	2 1 1	7 4 3	true
p4	7 4 3	4 3 1	0 0 2	7 4 5	true
p2	7 4 5	6 0 0	3 0 2	10 4 7	true
p0	10 4 7	7 4 3	0 1 0	10 5 7	true

P1申请资源(1,0,2)时安全性检查

	Max			Allocation			Need			Available		
	A	B	C	A	B	C	A	B	C	A	B	C
p0	7	5	3	0	1	0	7	4	3	3	3	2
p1	3	2	2	2 (3	0 0	0 2)	1 (0	2 2	2 0)	(2	3	0)
p2	9	0	2	3	0	2	6	0	0			
p3	2	2	2	2	1	1	0	1	1			
p4	4	3	3	0	0	2	4	3	1			

T1时刻的安全序列

安全

	Work A B C	Need A B C	Alloc A B C	Work+alloc A B C	Finish
p1	2 3 0	0 2 0	3 0 2	5 3 2	true
p3	5 3 2	0 1 1	2 1 1	7 4 3	true
p4	7 4 3	4 3 1	0 0 2	7 4 5	true
p0	7 4 5	7 4 3	0 1 0	7 5 5	true
p2	7 5 5	6 0 0	3 0 2	10 5 7	true

P0申请资源（0，2，0）时安全性检查

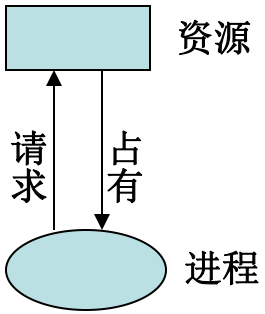
不安全

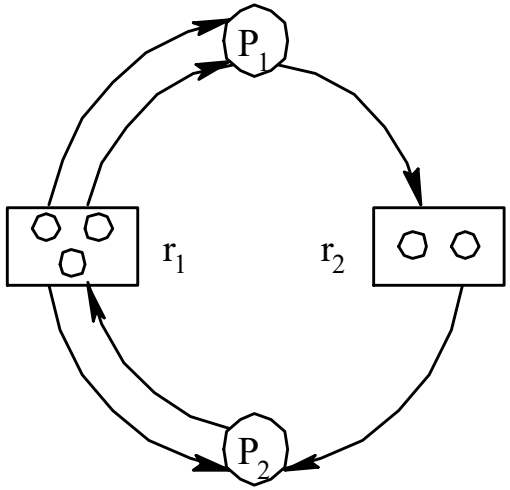
	Max			Allocation			Need			Available		
	A	B	C	A	B	C	A	B	C	A	B	C
p0	7	5	3	0 (0	1 3	0 0)	7 (7	4 2	3 3)	2 (2	3 1	0 0)
p1	3	2	2	3	0	2	0	2	0			
p2	9	0	2	3	0	2	6	0	0			
p3	2	2	2	2	1	1	0	1	1			
p4	4	3	3	0	0	2	4	3	1			

3.7 死锁的检测和解除

3.7.1 检测

1. 资源分配图





每类资源有多个时的情况

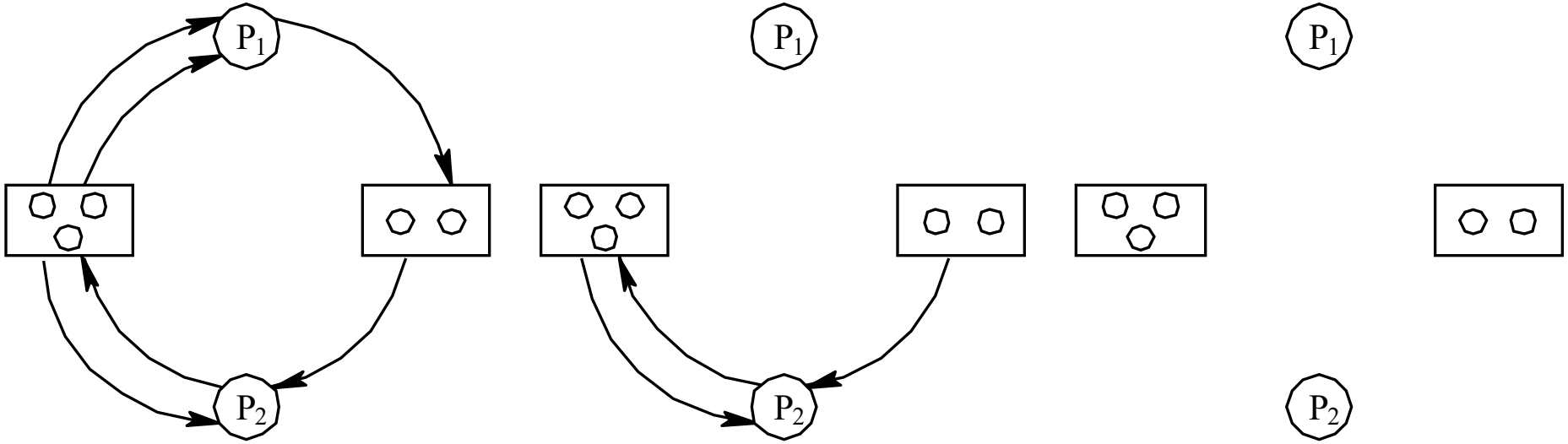
基本方法 ：圆圈表示进程，方框表示资源，方框内点表示资源数。箭头表示“占有”或者“请求”。

2. 死锁定理

S为死锁状态的充分条件是当且仅当S状态的资源分配图是不可完全简化。

- 找出既不阻塞又非独立结点，去掉所有分配边和请求边；
- 继续化简，若能完全简化，则不会发生死锁，否则会死锁。

既不阻塞又非独立的进程结点指从进程集合中找到一个有边与其连接，并且资源申请数量小于系统中已有空闲资源数。



(*a*) (*b*) (*c*)

3. 检测死锁的算法：

Work= available
L:={Li| $alloc_i$=0 req_i=0} (孤立进程点)
For all $L_i \notin L$ do
Begin
    For all req_i <=work do(是不阻塞的结点)
    Begin
        Work=work+$alloc_i$(回收所有边)
        $L=L_i \cup L$
    end
End
Deadlock= ~(L={p_1 … p_n}) (若L中并入了所有的进程，则不死锁)

讨论

我们知道了如何去检测死锁，但问题是在于我们应该何时去检测死锁？

- *一种方法是每当有资源请求时去检测，毫无疑问越早发现越好，但这种方法会占用昂贵的CPU时间；*
- *另一种方法是每隔k分钟去检测一次，或者当CPU的使用率降到某一限值时去检测；*
- *重视CPU使用率的原因在于如果死锁的进程数达到一定数量，就没有多少进程可以被执行，CPU会经常被限制。*

3.7.2 死锁的解除

系统中若出现死锁现象，将有大量的系统资源被占用、被浪费，甚至可能导致系统的崩溃。因此，一旦检测出死锁的存在，就应该尽快地采取措施，解除死锁，恢复系统的运行。

主要有三种办法：

- ☑ 剥夺资源
- ☑ 进程回退
- ☑ 撤消进程

A. 剥夺资源

- 把一个资源从一个进程手中抢过来，交给另一个进程使用，等它用完之后，再还给原来的进程。
- 在资源剥夺过程中所造成的不良影响的大小，完全取决于该资源自身的性质。实际上，死锁问题正是由不可抢占资源所引起的，所以强行剥夺资源必然会影响进程的正常运行。
- 这种方法实现起来有点困难，甚至是不可能的。因此只能选择那些相对而言容易剥夺的资源。
- 野蛮程度：★★★

B. 进程回退

- 定期地把每个进程的状态信息保存在文件当中，这样就得到一个文件序列，每个文件分别记载了该进程在不同时刻的状态。
- 进程的状态信息包括它的内存映象和它所占用的资源的状态。
- 当系统检测到死锁时，先查明有哪些资源涉及，然后把其中一个资源的拥有者（进程）回退到以前的某个时刻（尚未拥有该资源），从而打破死锁。但该进程从那时刻开始的所有工作都丢失了
- 野蛮程度：★。但代价高昂。

C. 撤消进程

- 撤消一个或多个处于死锁状态的进程。
- 先撤消一个死锁进程（或未死锁但占用了资源的进程），若其他死锁进程能够运行起来，则说明有效；否则继续撤消进程，直到死锁解除。
- 为了减少伤害程度，应尽可能地选择那些能安全地重新运行的进程，如编译进程；而不要去选择那些无法安全地重新运行的进程，如对数据库的更新。
- 野蛮程度：★★★★★

典型问题分析

问题一：假设一个系统包括A-G七个进程，R-W六种资源（各1个）。资源的所有权关系如下所示：

- A进程持有R资源，申请S资源。
- B进程不持有资源，申请T资源。
- C进程不持有资源，申请S资源。
- D进程持有U资源，申请S、T资源。
- E进程持有T资源，申请V资源。
- F进程持有W资源，申请S资源。
- G进程持有V资源，申请U资源。

问：系统是否存在死锁？如果存在的话，死锁包含哪些进程？

分析：由于每种资源都只有1个，因此，根据资源所有权关系画出资源分配图，看是否存在环？有则存在死锁，否则不存在死锁。

问题二：假定某计算机系统有R1和R2两类可使用资源（其中R1有两个单位，R2有一个单位），他们被进程P1和P2共享，且已知两进程均以下列顺序使用资源：

申请R1⇨ 申请R2⇨ 申请R1⇨ 释放R1⇨ 释放R2 ⇨ 释放R1

试求出系统运行过程中可能达到的死锁点，并画出死锁资源分配图。

问题三：

（1）3个进程共享4个同类型资源，每个进程最大需要2个资源，请问该系统是否会因为竞争该资源而死锁？

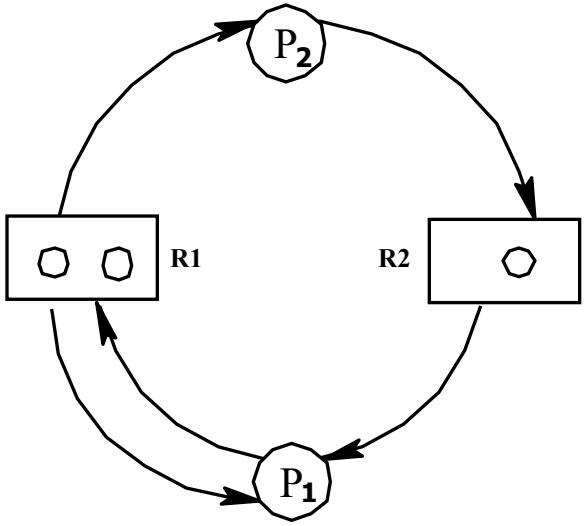
（2）n个进程共享m个同类资源，若每个进程都需要用该类资源，且各进程最大需求量之和小于m+n，试证明这个系统不会因为竞争该资源而发生死锁。

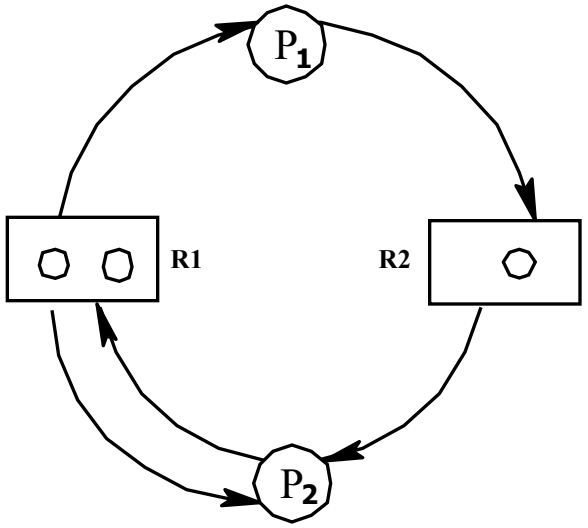
提示：max(1)+…+max(n)=need(1)+…+need(n)+alloc(1)+…alloc(n)<m+n

（3）在（2）中，如果没有“每个进程都需要用该类资源”的限制，情况又会如何？

问题二分析：

当两个进程都执行完第1步后，即进程p1和p2都申请到一个r1资源时，系统进入不安全状态。随着进程的推进，无论哪个进程执行完第2步，系统都将进入死锁状态。可能达到的死锁点是：p1、p2各占有一个r1，p1占有一个r2时会出现死锁；或者， p1、p2各占有一个r1，p2占有一个r2时会出现死锁。





（1）该系统不会因为竞争该资源而死锁。因为必有一个进程获得2个资源，故能顺利完成，并释放给其他进程使用，使它们也顺利完成。

（2）根据题意，有 $1<=max(i)<=m$； $\Sigma max<m+n$

反设系统处于死锁状态，则有

$\Sigma alloc=m$

因此， $\Sigma need= \Sigma max- \Sigma alloc<m+n-m=n$

这样，至少存在一个进程，其need(i) =0，这显然与题意不符，假设不成立，所以该系统不可能因竞争该类资源而进入死锁状态。

（3）此时系统可能发生死锁，如n=4,m=3时，若p1的max为0，而其余三个进程的都为2，则仍然满足最大需求量之和（即6）小于m+n（即7）的要求，但当除p1以外的其余三个进程各得到一个资源时，这三个进程将进入死锁状态。

小结：处理死锁的3种基本方法

方法	资源分配策略	各种可能模式	主要优点	主要缺点
预防 Prevention	保守的；宁可资源闲置（从机制上使死锁条件不成立）	一次请求所有资源	适用于作突发式处理的进程；不必剥夺	效率低；进程初始化时间延长 剥夺次数过多；多次对资源重新起动 不便灵活申请新资源
		资源剥夺	适用于状态可以保存和恢复的资源	
		资源按序申请	可以在编译时（而不必在运行时）就进行检查	
避免 Avoidance	是“预防”和“检测”的折衷（在运行时判断是否可能死锁）	寻找可能的安全的运行顺序	不必进行剥夺	使用条件：必须知道将来的资源需求；进程可能会长时间阻塞
检测 Detection	宽松的；只要允许，就分配资源	定期检查死锁是否已经发生	不延长进程初始化时间；允许对死锁进行现场处理	通过剥夺解除死锁，造成损失