第三章 处理机调度与死锁

3.1处理机调度的层次和调度算法的目标

处理机是计算机系统中的重要资源。处理机调度算法对整个计算机系统的综合性能指标有重要影响。

处理机调度（CPU调度）要解决的问题：

WHAT：按什么原则分配CPU

—调度算法

WHEN：何时分配CPU

—调度的时机

HOW： 如何分配CPU

—CPU调度过程（进程的上下文切换）

3.1.1 处理机调度的层次

1. 高级调度(High Scheduling)

也称为作业调度或宏观调度，一般在批处理系统中有作业调度。

在每次执行作业调度时，都须做出以下两个决定。 ■

1) 接纳多少个作业

2) 接纳哪些作业

2. 低级调度(Low Level Scheduling)

也称微观调度、进程调度，从处理机资源分配的角度来看，处理机需要经常选择就绪进程或线程进入运行状态。由于低级调度算法的频繁使用，要求在实现时做到高效。

1) 非抢占方式(Non-preemptive Mode) ■

在采用非抢占调度方式时，可能引起进程调度的因素可归结为这样几个：① 正在执行的进程执行完毕， 或因发生某事件而不能再继续执行； ② 执行中的进程因提出I/O请求而暂停执行；③ 在进程通信或同步过程中执行了某种原语操作，如P操作(wait操作)、Block原语、Wakeup原语等。这种调度方式的优点是实现简单、系统开销小，适用于大多数的批处理系统环境。但它难以满足紧急任务的要求—立即执行，因而可能造成难以预料的后果。显然，在要求比较严格的实时系统中，不宜采用这种调度方式。

2) 抢占方式(Preemptive Mode)

抢占的原则有：■

(1) 优先权原则。

(2) 短作业(进程)优先原则。

(3) 时间片原则。

3. 中级调度(Intermediate-Level Scheduling)

中级调度又称中程调度(Medium-Term Scheduling)。 引入中级调度的主要目的，是为了提高内存利用率和系统吞吐量。 为此，应使那些暂时不能运行的进程不再占用宝贵的内存资源，而将它们调至外存上去等待，把此时的进程状态称为就绪驻外存状态或挂起状态。当这些进程重又具备运行条件、且内存又稍有空闲时，由中级调度来决定把外存上的哪些又具备运行条件的就绪进程，重新调入内存，并修改其状态为就绪状态，挂在就绪队列上等待进程调度。

3.1.2 处理机调度算法的目标

1. 处理机调度算法的共同目标

(1) 资源利用率：提高系统资源利用率。

(2) 公平性：诸进程都获得合理的CPU时间。

(3) 平衡性：保持系统资源均衡使用。

(4) 策略强制执行：制定的策略须得到准确执行。

2. 批处理系统目标

(1) 平均周转时间短。

可把平均周转时间描述为：$$T=\frac{1}{n}\left[\sum_{i=1}^{i}T_i\right]$$

作业的周转时间T与系统为它提供服务的时间T_S之比，即$W=T/T_S$，称为带权周转时间，而平均带权周转时间则可表示为:

$$W=\frac{1}{n}\left[\sum_{i=1}^{n}\frac{T_i}{T_{Si}}\right]$$

（2）系统吞吐量高

（3）处理机利用率高

3.分时系统的目标

（1）响应时间快：用户提交请求到获得处理结果的间隔时间短。

（2）均衡性：系统响应时间的快慢应与用户请求服务的复杂性相适应。

4.实时系统的目标

（1）截止时间的保证

（2）可预测性

3.2 作业与作业调度

3.2.1 批处理系统中的作业

1.作业和作业步

•作业：包含程序、数据、作业说明书。

•作业步：作业运行期间相对独立的步骤。

2.作业控制块（JCB）

在多道批处理系统中，为了管理和调度作业，为每一个作业设置了一个作业控制块。

3.作业运行的三个阶段和三种状态

•收容阶段----后备状态

•运行阶段----运行状态

•完成阶段----完成状态

3.2.2作业调度的主要任务

- **接纳多少个作业：**从后备作业队列中选取多少个作业进入内存。取决于内存容量和系统资源。
- **接纳哪些作业：**从后备作业队列中选取哪些作业进入内存。取决于调度算法。

3.2.3 先来先服务（FCFS）和短作业优先（SJF）调度算法

调度算法是指：根据系统的资源分配策略所规定的资源分配算法。对于不同的系统和系统目标，通常采用不同的调度算法。

1. 先来先服务调度算法

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

算法的优缺点：

优点：:实现简单。

缺点：:没考虑进程的优先级，平均周转时间长，不利于短作业。

2. 短作业(进程)优先调度算法

短作业(进程)优先调度算法SJ(P)F，是指对短作业或短进程优先调度的算法。它们可以分别用于作业调度和进程调度。短作业优先(SJF)的调度算法，是从后备队列中选择一个或若干个估计运行时间最短的作业，将它们调入内存运行。而短进程优先(SPF)调度算法，则是从就绪队列中选出一估计运行时间最短的进程，将处理机分配给它，使它立即执行并一直执行到完成，或发生某事件而被阻塞放弃处理机时，再重新调度。 ■

调度算法 \ 作业情况	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 (a)	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 (b)	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

FCFS和SJF调度算法的性能

SJ(P)F调度算法的优缺点：

优点：平均周转时间短，系统吞吐量大。

缺点：(1) 该算法对长作业不利，如作业C的周转时间由10增至16，其带权周转时间由2增至3.1。更严重的是，如果有一长作业(进程)进入系统的后备队列(就绪队列)，由于调度程序总是优先调度那些(即使是后进来的)短作业(进程)，将导致长作业(进程)长期不被调度。 ■

(2) 该算法完全未考虑作业的紧迫程度，因而不能保证紧迫性作业(进程)会被及时处理。 ■

(3) 由于作业(进程)的长短只是根据用户所提供的估计执行时间而定的，而用户又可能会有意或无意地缩短其作业的估计运行时间，致使该算法不一定能真正做到短作业优先调度。

3.2.4 优先级调度算法和高响应比优先调度算法

1. 优先级调度算法

优先级高的作业优先被调度运行。优先级的确定取决于调度算法。FCFS调度算法，等待时间长的优先级高。SJF调度算法，运行时间短的优先级高。紧迫的作业可赋予更高的优先级。

2. 高响应比优先调度算法

优先权的变化规律可描述为：

$$\text{优先权} = \frac{\text{等待时间} + \text{要求服务时间}}{\text{要求服务时间}}$$

由于等待时间与服务时间之和，就是系统对该作业的响应时间，故该优先权又相当于响应比R_P。据此，又可表示为：

$$\text{优先权} = \frac{\text{等待时间} + \text{要求服务时间}}{\text{要求服务时间}} = \frac{\text{响应时间}}{\text{要求服务时间}}$$

(1) 如果作业的等待时间相同，则要求服务的时间愈短，其优先权愈高，因而该算法有利于短作业。■

(2) 当要求服务的时间相同时，作业的优先权决定于其等待时间，等待时间愈长，其优先权愈高，因而它实现的是先来先服务。■

(3) 对于长作业，作业的优先级可以随等待时间的增加而提高，当其等待时间足够长时，其优先级便可升到很高，从而也可获得处理机。

3.3进程调度

3.3.1进程调度的任务、机制和方式

1.进程调度的任务

- 保存处理机的现场
- 按某种算法选取进程
- 把处理机分配给进程

2.进程调度机制

- 排队器
- 分派器
- 上下文切换

3. 进程调度方式

1) 非抢占式算法

在这种方式下，系统一旦把处理机分配给就绪队列中最高的进程后，该进程便一直执行下去，直至完成； 或因发生某事件使该进程放弃处理机时，系统方可再将处理机重新分配给另一优先权最高的进程。

2) 抢占式调度算法 ■

允许调度程序根据某种原则，去暂停某个正在执行的进程，将该进程的处理机分配给另一个进程。

抢占式调度遵循的原则：

- 优先权原则
- 短进程优先原则
- 时间片原则

3.3.2时间片轮转调度算法

- 轮转法的原理：所有就绪的进程按FCFS排成一个队列，每个进程轮流运行一个时间片。
- 进程切换时机：进程在时间片内结束、分配给进程的时间片用完。
- 时间片大小的确定：过短，导致切换频率高，增加系统开销；过长，则退化成FCFS算法。

时间片选择问题：

固定时间片。

可变时间片。

与时间片大小有关的因素：

系统响应时间。

就绪进程个数。

CPU能力。

3.3.3优先级调度算法

1.优先级调度算法的类型

1) 非抢占式优先权算法

在这种方式下，系统一旦把处理机分配给就绪队列中优先权最高的进程后，该进程便一直执行下去，直至完成；或因发生某事件使该进程放弃处理机时，系统方可再将处理机重新分配给另一优先权最高的进程。这种调度算法主要用于批处理系统中；也可用于某些对实时性要求不严的实时系统中。

2) 抢占式优先权调度算法 ■

在这种方式下，系统同样是把处理机分配给优先权最高的进程，使之执行。但在其执行期间，只要又出现了另一个其优先权更高的进程，进程调度程序就立即停止当前进程(原优先权最高的进程)的执行，重新将处理机分配给新到的优先权最高的进程。因此，在采用这种调度算法时，是每当系统中出现一个新的就绪进程i时，就将其优先权P_i与正在执行的进程j的优先权P_j进行比较。如果$P_i \leq P_j$，原进程P_j便继续执行；但如果是$P_i > P_j$，则立即停止P_j的执行，做进程切换，使i进程投入执行。显然，这种抢占式的优先权调度算法，能更好地满足紧迫作业的要求，故而常用于要求比较严格的实时系统中， 以及对性能要求较高的批处理和分时系统中。

2. 优先权的类型

1) 静态优先权 ■

静态优先权是在创建进程时确定的，且在进程的整个运行期间保持不变。一般地，优先权是利用某一范围内的一个整数来表示的，例如，0~7或0~255中的某一整数，又把该整数称为优先数。只是具体用法各异：有的系统用“0”表示最高优先权，当数值愈大时，其优先权愈低；而有的系统恰恰相反。

确定进程优先权的依据有如下三个方面：

(1) 进程类型。

(2) 进程对资源的需求。

(3) 用户要求。

2) 动态优先权 ■

动态优先权是指，在创建进程时所赋予的优先权，是可以随进程的推进或随其等待时间的增加而改变的，以便获得更好的调度性能。例如，我们可以规定，在就绪队列中的进程，随其等待时间的增长，其优先权以速率a提高。若所有的进程都具有相同的优先权初值，则显然是最先进入就绪队列的进程，将因其动态优先权变得最高而优先获得处理机，此即FCFS算法。若所有的就绪进程具有各不相同的优先权初值，那么，对于优先权初值低的进程，在等待了足够的时间后，其优先权便可能升为最高，从而可以获得处理机。当采用抢占式优先权调度算法时，如果再规定当前进程的优先权以速率b下降，则可防止一个长作业长期地垄断处理机。 ■

补充： 调度队列模型

1. 仅有进程调度的调度队列模型

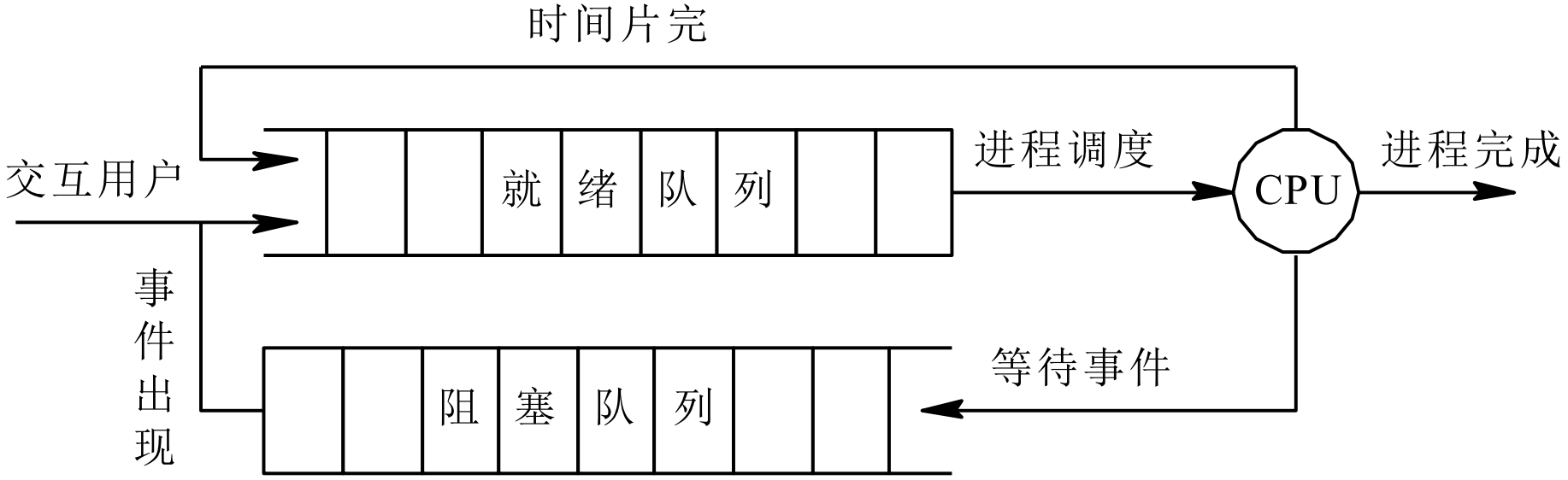


图 3 - 1 仅具有进程调度的调度队列模型

2. 具有高级和低级调度的调度队列模型

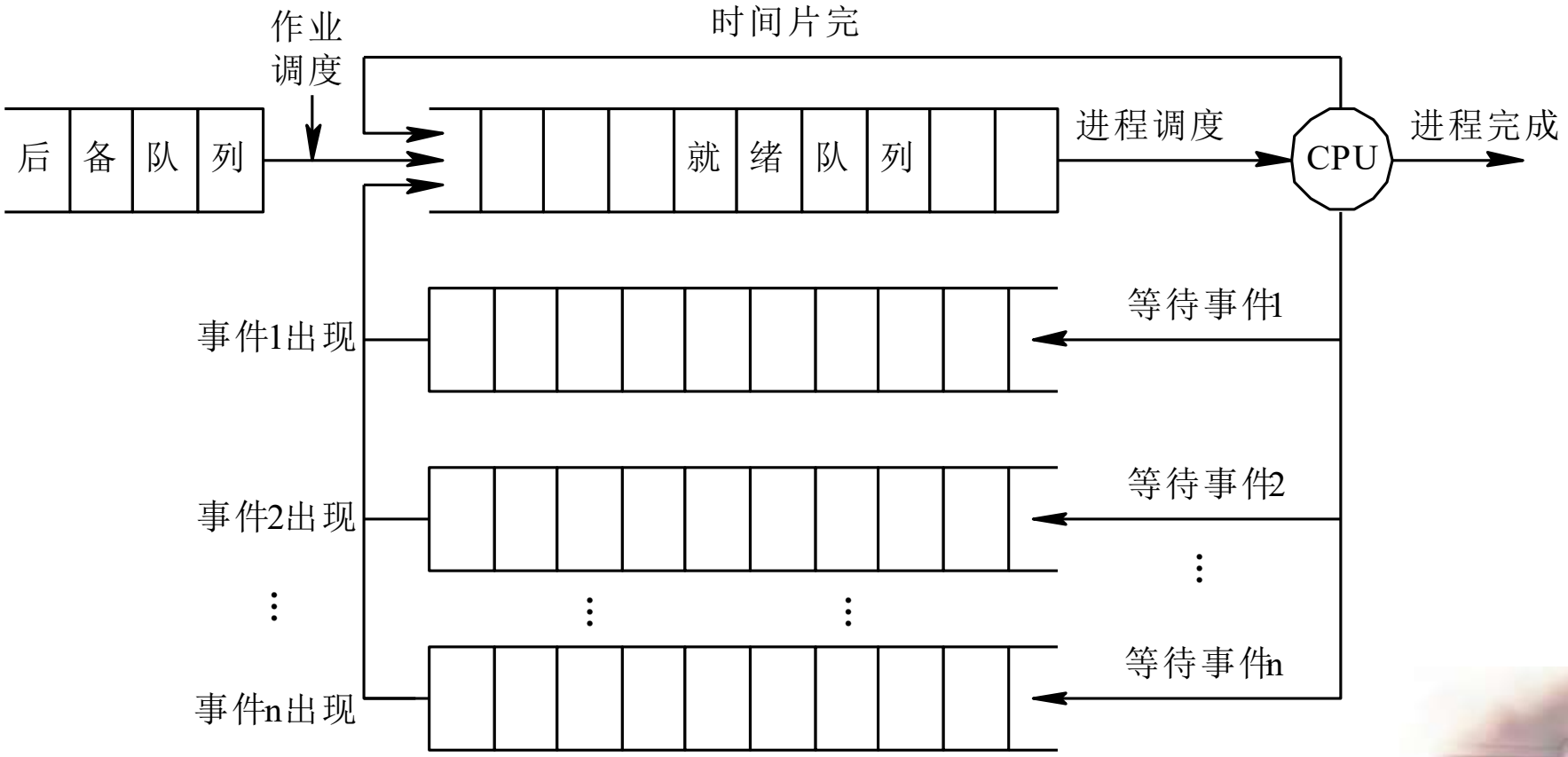


图 3-2 具有高、低两级调度的调度队列模型

图 3-2 示出了具有高、低两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

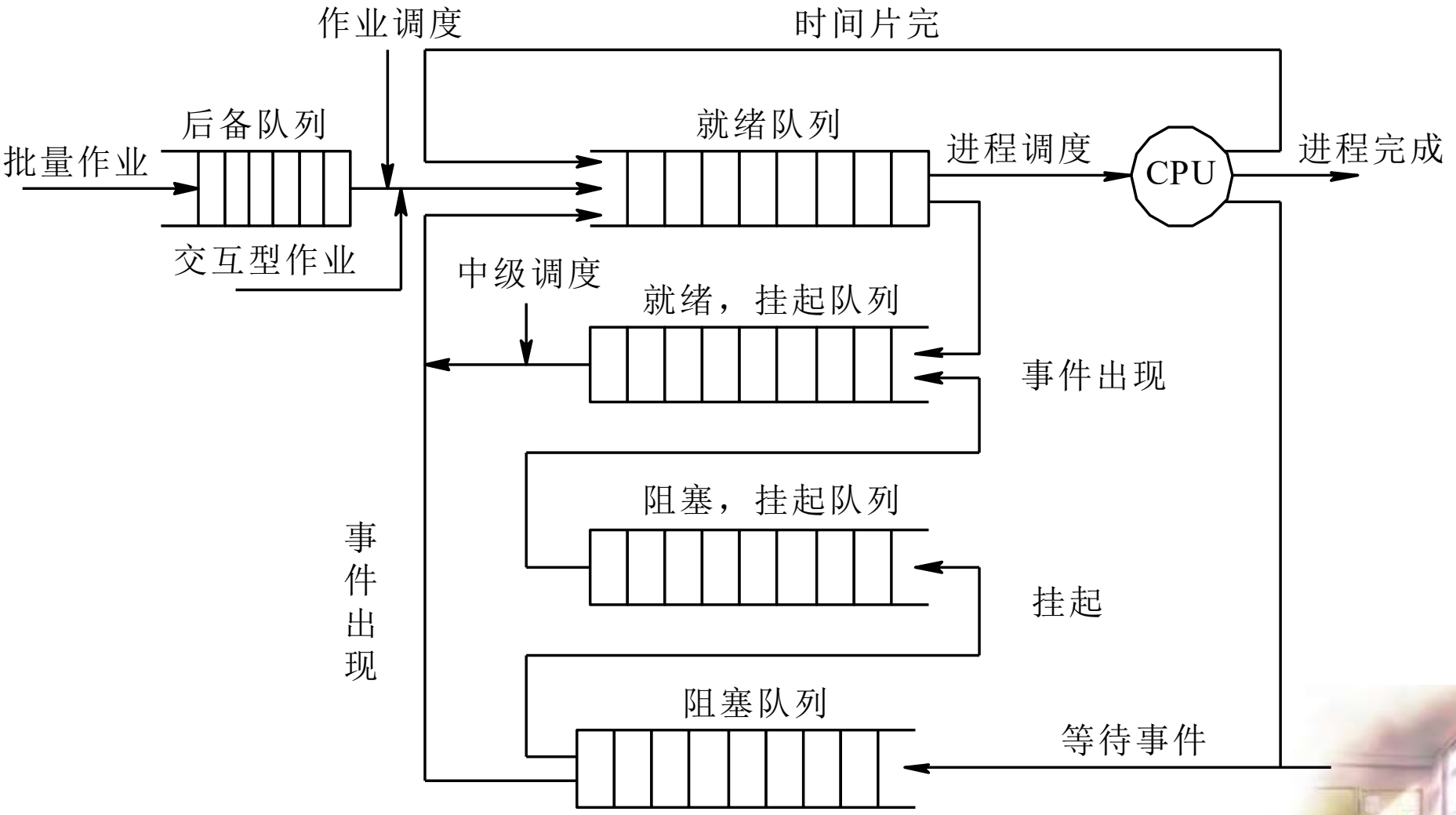


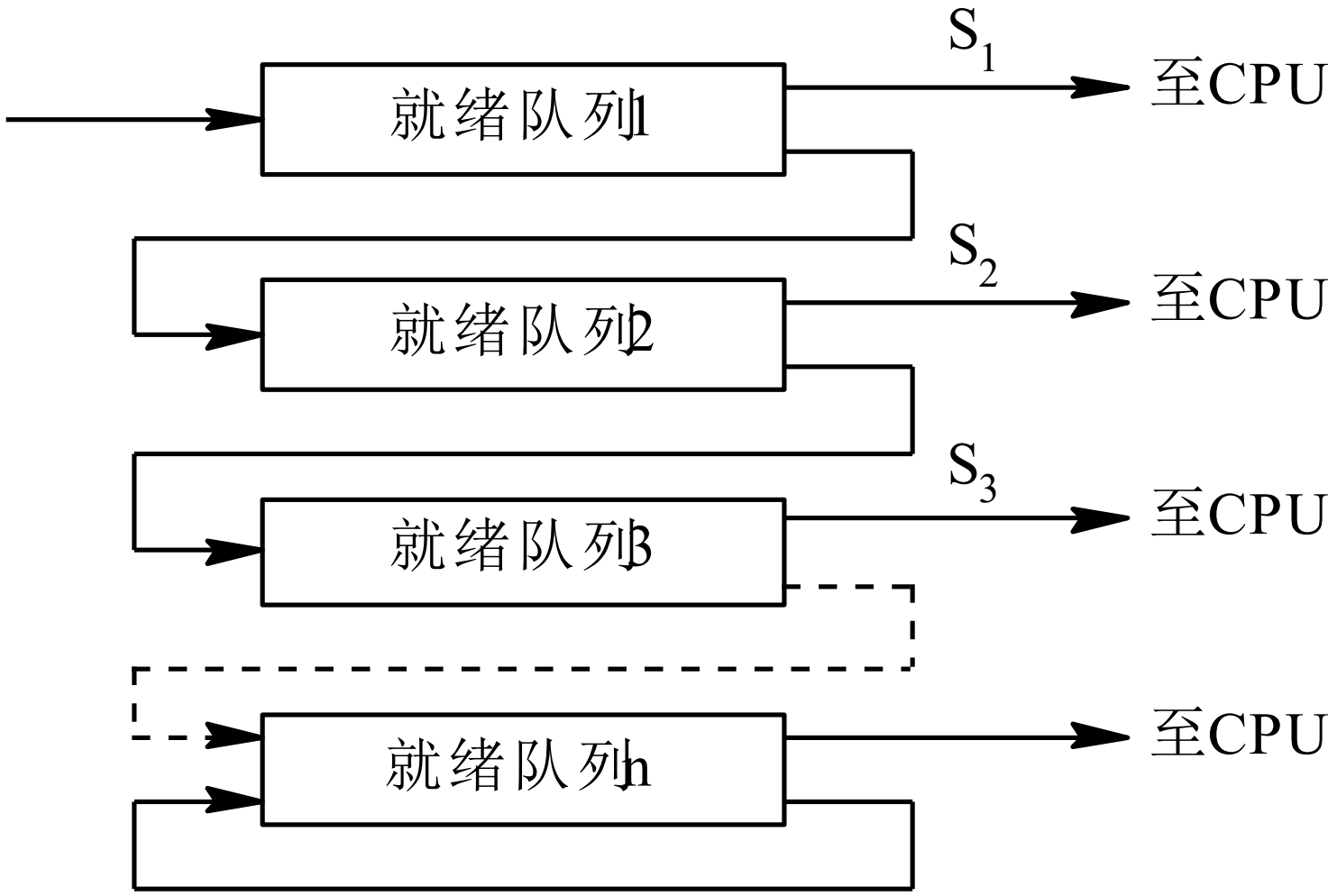
图 3-3 具有三级调度时的调度队列模型

3.3.4 多队列调度算法

将不同类型或性质的进程固定分配在不同的就绪队列，不同的就绪队列采用不同的调度算法，一个就绪队列中的进程可以设置不同的优先级，不同的就绪队列本身也可以设置不同的优先级。

3.3.5 多级反馈队列调度算法

(1) 应设置多个就绪队列，并为各个队列赋予不同的优先级。 第一个队列的优先级最高，第二个队列次之，其余各队列的优先权逐个降低。该算法赋予各个队列中进程执行时间片的大小也各不相同，在优先权愈高的队列中，为每个进程所规定的执行时间片就愈小。例如，第二个队列的时间片要比第一个队列的时间片长一倍，……，第*i*+1个队列的时间片要比第i个队列的时间片长一倍。 图 3-5 是多级反馈队列算法的示意。



（时间片：$S_1 < S_2 < S_3$）

图 3-5 多级反馈队列调度算法

(2) 当一个新进程进入内存后，首先将它放入第一队列的末尾，按FCFS原则排队等待调度。当轮到该进程执行时，如它能在该时间片内完成，便可准备撤离系统；如果它在一个时间片结束时尚未完成，调度程序便将该进程转入第二队列的末尾，再同样地按FCFS原则等待调度执行；如果它在第二队列中运行一个时间片后仍未完成，再依次将它放入第三队列，.....，如此下去，当一个长作业(进程)从第一队列依次降到第n队列后，在第n队列中便采取按时间片轮转的方式运行。 ■

(3) 仅当第一队列空闲时，调度程序才调度第二队列中的进程运行； 仅当第1~(i-1) 队列均空时，才会调度第i队列中的进程运行。如果处理机正在第i队列中为某进程服务时，又有新进程进入优先权较高的队列(第1~(i-1)中的任何一个队列)，则此时新进程将抢占正在运行进程的处理机，即由调度程序把正在运行的进程放回到第i队列的末尾，把处理机分配给新到的高优先权进程。

3. 多级反馈队列调度算法的性能

(1) 终端型作业用户。

(2) 短批处理作业用户。

(3) 长批处理作业用户。

3.3.6基于公平原则的调度算法

1. **保证调度算法：**保证各进程公平获得处理机。
2. **公平分享调度算法：**分给每个进程的时间相同。

3.4 实 时 调 度

3.4.1 实现实时调度的基本条件

1. 提供必要的信息

(1) 就绪时间。

(2) 开始截止时间和完成截止时间。

(3) 处理时间。

(4) 资源要求。

(5) 优先级。

2. 系统处理能力强

在实时系统中，通常都有着多个实时任务。若处理机的处理能力不够强，则有可能因处理机忙不过来而使某些实时任务不能得到及时处理， 从而导致发生难以预料的后果。假定系统中有m个周期性的硬实时任务，它们的处理时间可表示为C_i，周期时间表示为P_i，则在单处理机情况下，必须满足下面的限制条件：

$$\sum_{i=1}^{m}\frac{C_i}{P_i}\leq 1$$

系统才是可调度的。假如系统中有6个硬实时任务，它们的周期时间都是 50 ms，而每次的处理时间为 10 ms，则不难算出，此时是不能满足上式的，因而系统是不可调度的。

解决的方法是提高系统的处理能力，其途径有二：其一仍是采用单处理机系统， 但须增强其处理能力， 以显著地减少对每一个任务的处理时间；其二是采用多处理机系统。假定系统中的处理机数为N，则应将上述的限制条件改为：

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

3. 采用抢占式调度机制

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，而令高优先权任务立即投入运行，这样便可满足该硬实时任务对截止时间的要求。但这种调度机制比较复杂。■

对于一些小的实时系统，如果能预知任务的开始截止时间，则对实时任务的调度可采用非抢占调度机制，以简化调度程序和对任务调度时所花费的系统开销。但在设计这种调度机制时，应使所有的实时任务都比较小，并在执行完关键性程序和临界区后，能及时地将自己阻塞起来，以便释放出处理机， 供调度程序去调度那种开始截止时间即将到达的任务。

4. 具有快速切换机制

该机制应具有如下两方面的能力：■

(1) 对外部中断的快速响应能力。为使在紧迫的外部事件请求中断时系统能及时响应，要求系统具有快速硬件中断机构，还应使禁止中断的时间间隔尽量短， 以免耽误时机(其它紧迫任务)。■

(2) 快速的任务分派能力。在完成任务调度后，便应进行任务切换。为了提高分派程序进行任务切换时的速度，应使系统中的每个运行功能单位适当的小，以减少任务切换的时间开销。

3.4.2 实时调度算法的分类

1. 非抢占式调度算法

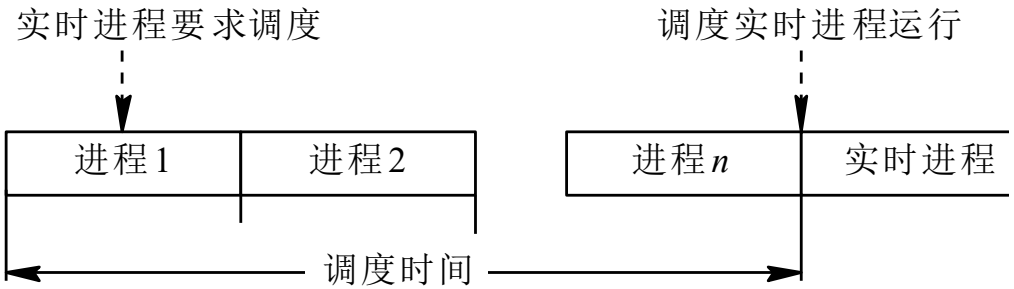
(1) 非抢占式轮转调度算法。

(2) 非抢占式优先调度算法。

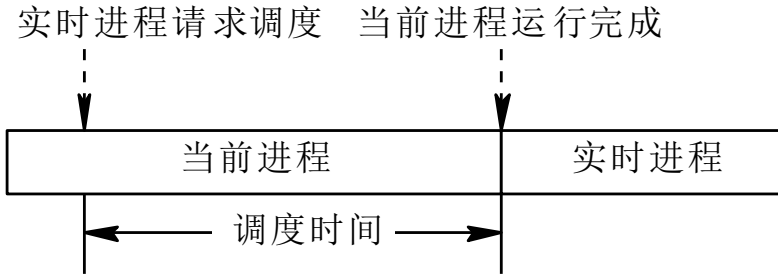
2. 抢占式调度算法

(1) 基于时钟中断的抢占式优先权调度算法。

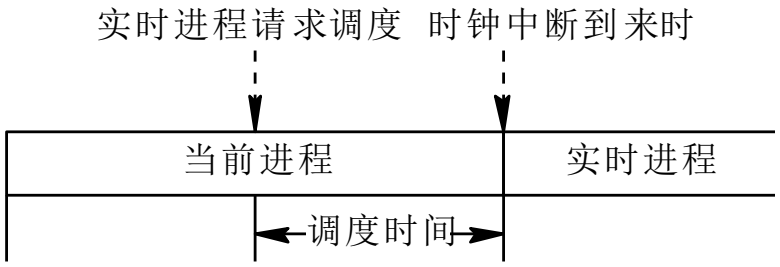
(2) 立即抢占(Immediate Preemption)的优先权调度算法。



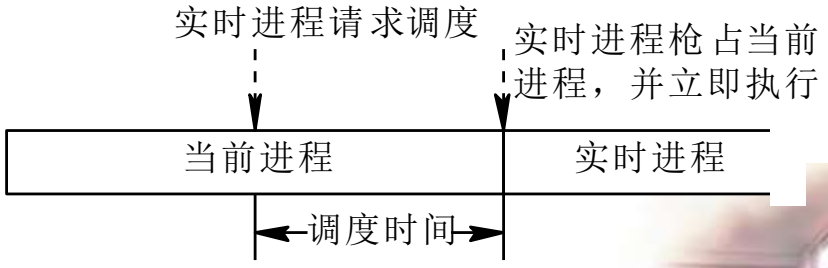
(*a*) 非抢占轮转调度



(*b*) 非抢占优先权调度



(*c*) 基于时钟中断抢占的优先权抢占调度



(*d*) 立即抢占的优先权调度

图 3-6 实时进程调度

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

采用：最早截止时间优先即EDF(Earliest Deadline First)算法

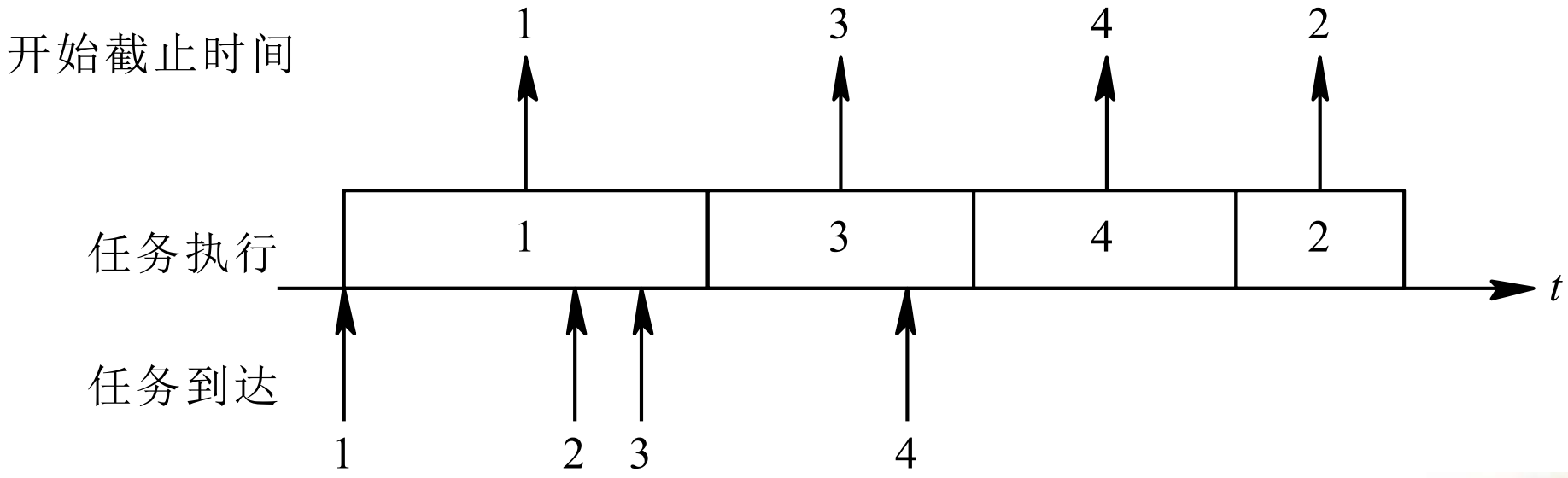


图 3-7 EDF算法用于非抢占调度方式

2.抢占式调度方式用于周期实时任务

例：假如在一个实时系统中，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。

用优先级调度不能适用于实时系统。

采用：最早截止时间优先即EDF(Earliest Deadline First)算法 。

图见P100。

3.4.4 最低松弛度优先即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。任务的紧急程度愈高，为该任务所赋予的优先级就愈高， 以使之优先执行。例如，一个任务在200ms时必须完成，而它本身所需的运行时间就有100ms，因此，调度程序必须在100 ms之前调度执行，该任务的紧急程度(松弛程度)为100 ms。又如，另一任务在400 ms时必须完成，它本身需要运行 150 ms，则其松弛程度为 250 ms。在实现该算法时要求系统中有一个按松弛度排序的实时任务就绪队列，松弛度最低的任务排在队列最前面，调度程序总是选择就绪队列中的队首任务执行。该算法主要用于可抢占调度方式中。假如在一个实时系统中，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。

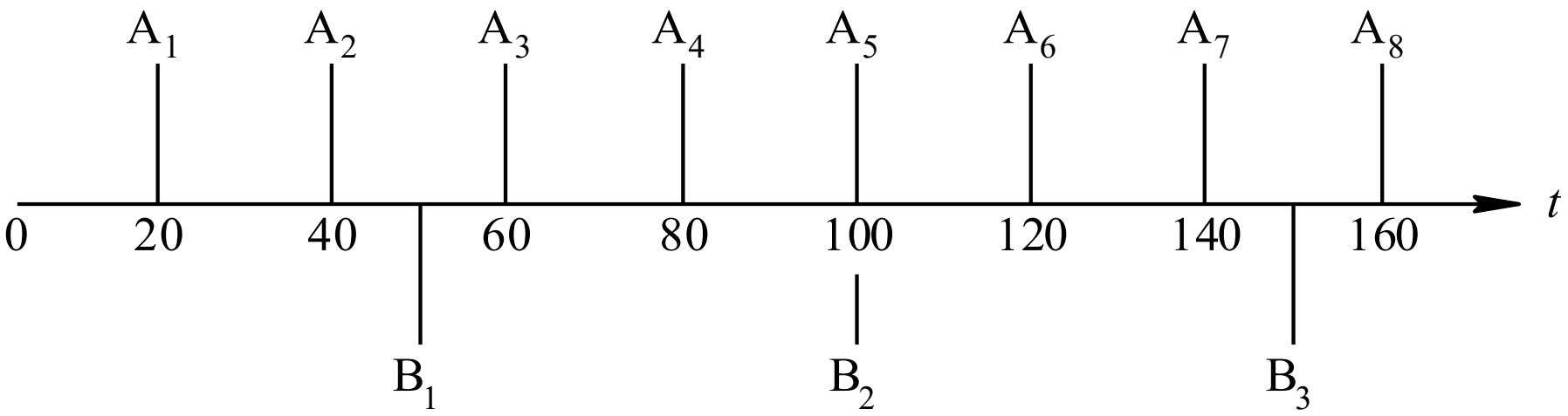


图 3-8 A和B任务每次必须完成的时间

在刚开始时(t_1=0)，A_1必须在20ms时完成，而它本身运行又需 10 ms，可算出A_1的松弛度为10ms；B_1必须在50ms时完成， 而它本身运行就需25 ms，可算出B_1的松弛度为25 ms，故调度程序应先调度A_1执行。在t_2=10 ms时，A_2的松弛度可按下式算出：

♡ A_2的松弛度=必须完成时间-其本身的运行时间-当前时间

=40 ms-10 ms-10 ms=20 ms

类似地，可算出B_1的松弛度为15ms，故调度程序应选择B_2运行。在t_3=30 ms时，A_2的松弛度已减为0(即40-10-30)，而B_1的松弛度为15 ms(即50-5-30)，于是调度程序应抢占B_1的处理机而调度A_2运行。在t_4=40 ms时，A_3的松弛度为10 ms(即60-10-40)，而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。在t_5=45 ms时，B_1执行完成，而此时A_3的松弛度已减为5 ms(即60-10-45)，而B_2的松弛度为30 ms(即100-25-45)，于是又应调度A_3执行。在t_6=55ms时，任务A尚未进入第4周期，而任务B已进入第2周期，故再调度B_2执行。在t_7=70 ms时，A_4的松弛度已减至0 ms(即80-10-70)，而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。

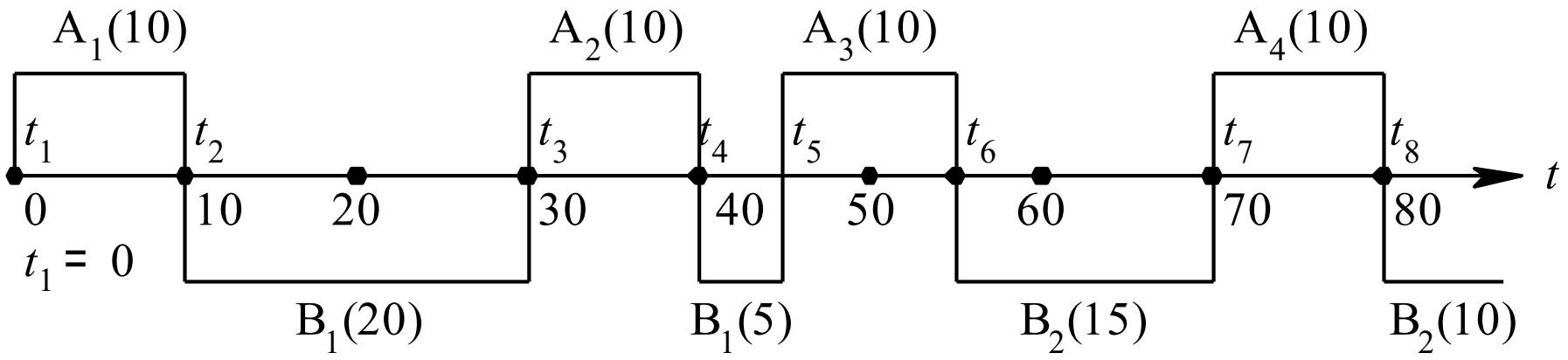


图 3-9 利用ELLF算法进行调度的情况

3.4.5优先级倒置

- 优先级倒置的形成
- 优先级倒置的解决方法

补充：多处理机系统中的调度

一、多处理器系统的类型

(1) 紧密耦合(Tightly Coupted)MPS。

这通常是通过高速总线或高速交叉开关，来实现多个处理器之间的互连的。它们共享主存储器系统和I/O设备，并要求将主存储器划分为若干个能独立访问的存储器模块，以便多个处理机能同时对主存进行访问。系统中的所有资源和进程，都由操作系统实施统一的控制和管理。

(2) 松散耦合(Loosely Coupled)MPS。

在松散耦合MPS中，通常是通过通道或通信线路，来实现多台计算机之间的互连。每台计算机都有自己的存储器和I/O设备，并配置了OS来管理本地资源和在本地运行的进程。因此，每一台计算机都能独立地工作， 必要时可通过通信线路与其它计算机交换信息，以及协调它们之间的工作。

2. 对称多处理器系统和非对称多处理器系统

(1) 对称多处理器系统SMPS(Symmetric MultiProcessor System)。 在系统中所包含的各处理器单元，在功能和结构上都是相同的， 当前的绝大多数MPS都属于SMP系统。例如，IBM公司的SR/6000 Model F50, 便是利用4片Power PC处理器构成的。 ■

(2) 非对称多处理器系统。在系统中有多种类型的处理单元， 它们的功能和结构各不相同，其中只有一个主处理器，有多个从处理器。

二、 进程分配方式

1. 对称多处理器系统中的进程分配方式 ■

在SMP系统中，所有的处理器都是相同的，因而可把所有的处理器作为一个处理器池(Processor pool)，由调度程序或基于处理器的请求， 将任何一个进程分配给池中的任何一个处理器去处理。在进行进程分配时，可采用以下两种方式之一。

1) 静态分配(Static Assigenment)方式

2) 动态分配(Dynamic Assgement)方式 ■

2. 非对称MPS中的进程分配方式 ■

对于非对称MPS， 其OS大多采用主–从(Master-Slave)式OS， 即OS的核心部分驻留在一台主机上(Master)， 而从机(Slave)上只是用户程序， 进程调度只由主机执行。 每当从机空闲时， 便向主机发送一索求进程的信号， 然后， 便等待主机为它分配进程。 在主机中保持有一个就绪队列，只要就绪队列不空， 主机便从其队首摘下一进程分配给请求的从机。从机接收到分配的进程后便运行该进程， 该进程结束后从机又向主机发出请求。

三、 进程(线程)调度方式

1. 自调度(Self-Scheduling)方式

1) 自调度机制 ■

在多处理器系统中，自调度方式是最简单的一种调度方式。它是直接由单处理机环境下的调度方式演变而来的。在系统中设置有一个公共的进程或线程就绪队列， 所有的处理器在空闲时，都可自己到该队列中取得一进程(或线程)来运行。在自调度方式中，可采用在单处理机环境下所用的调度算法，如先来先服务(FCFS)调度算法、最高优先权优先(FPF)调度算法和抢占式最高优先权优先调度算法等。

2) 自调度方式的优点 ■

自调度方式的主要优点表现为：首先，系统中的公共就绪队列可按照单处理机系统中所采用的各种方式加以组织； 其调度算法也可沿用单处理机系统所用的算法，亦即，很容易将单处理机环境下的调度机制移植到多处理机系统中， 故它仍然是当前多处理机系统中较常用的调度方式。其次， 只要系统中有任务，或者说只要公共就绪队列不空，就不会出现处理机空闲的情况，也不会发生处理器忙闲不均的现象，因而有利于提高处理器的利用率。

3) 自调度方式的缺点

(1) 瓶颈问题。

(2) 低效性。

(3) 线程切换频繁。

2. 成组调度(Gang Scheduling)方式 ■

在成组调度时，如何为应用程序分配处理器时间，

1) 面向所有应用程序平均分配处理器时间

2) 面向所有线程平均分配处理器时间

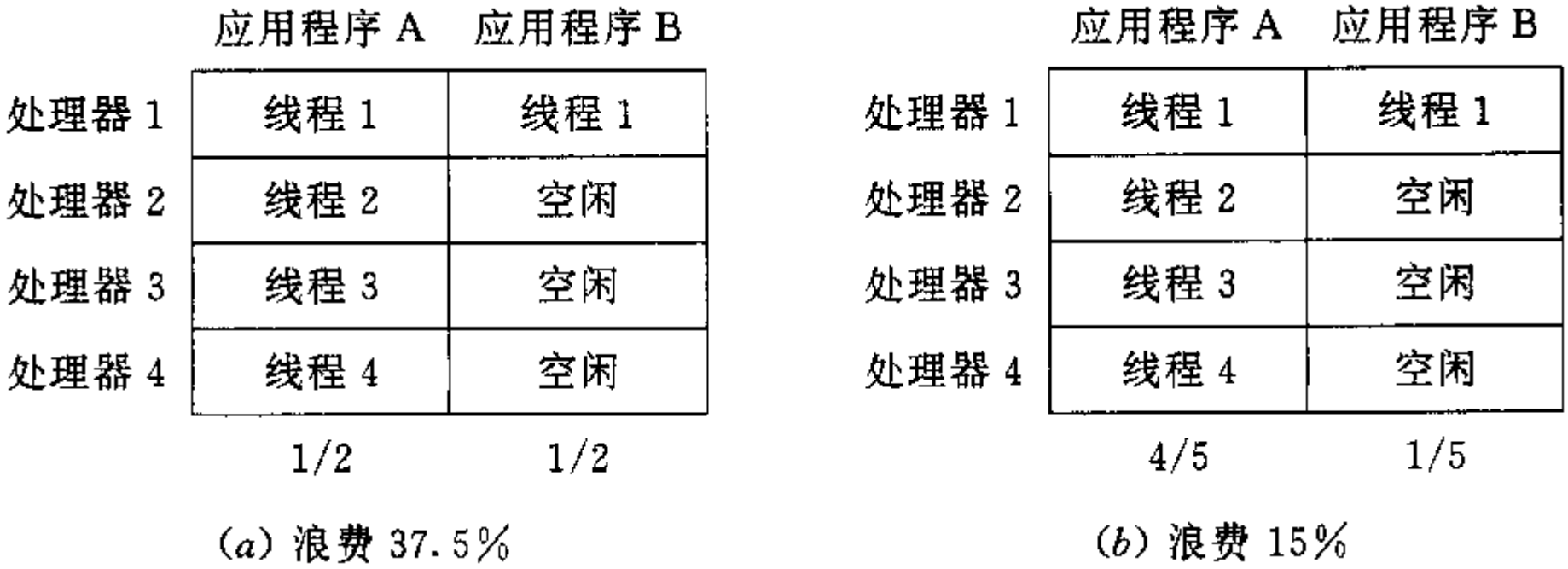


图 3 - 10 两种分配处理器时间的方法

3. 专用处理器分配(Dedicated Processor Assigement)方式

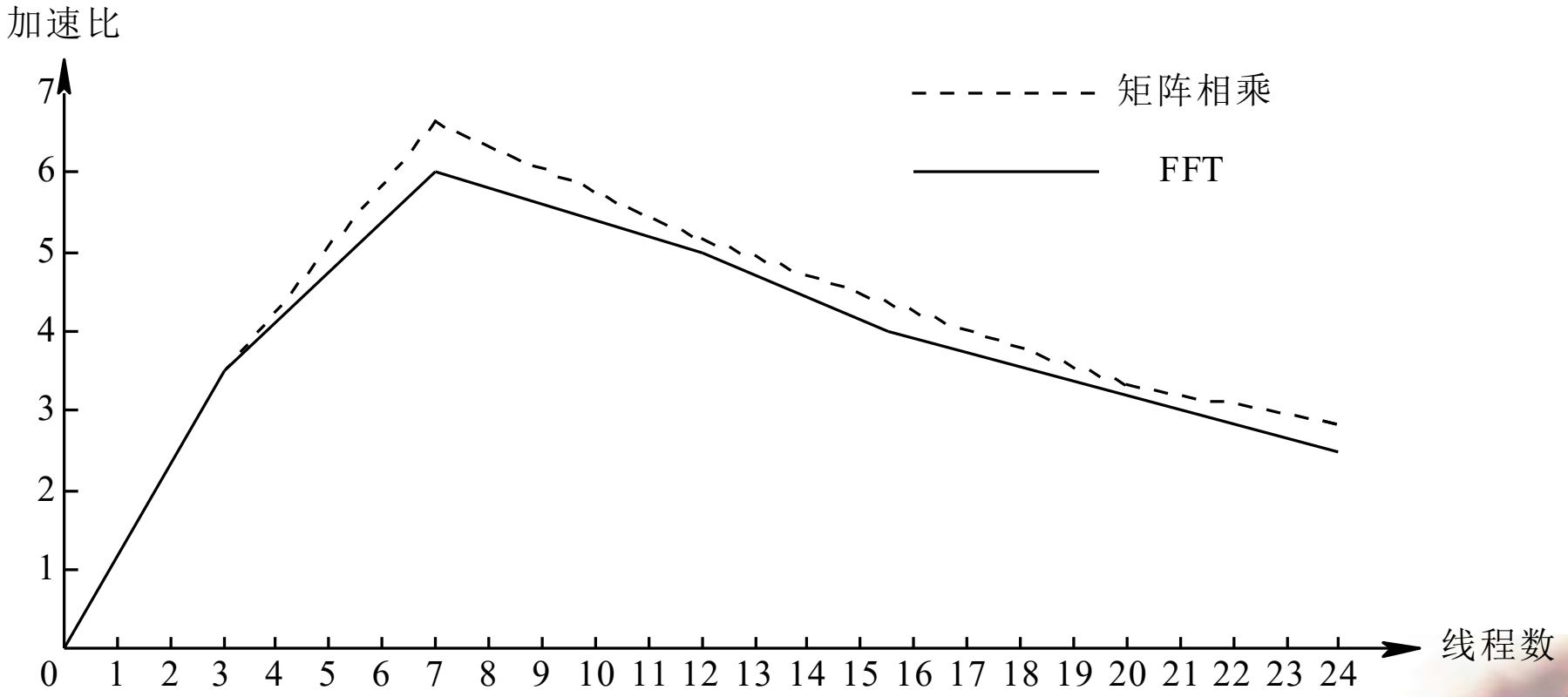


图 3-11 线程数对加速比的影响

3.5 死锁概述

死锁举例

- 例1：

  两个小孩在一起玩遥控汽车，一个拿着遥控器，另一个拿着汽车，如果这两个小孩都要对方手中的玩具，而又不肯先放掉自己拿着的玩具，这时就发生了僵持局面。

- 例2：

设系统有一台打印机和一台扫描仪，进程P1、P2并发执行，在某时刻T，进程P1和P2分别占用了打印机和扫描仪。在时刻T1（T1>T），P1又要申请扫描仪，但由于扫描仪被P2占用，P1只有等待。在时刻T2（T2>T），P2又申请打印机，但由于打印机被P1占用，P2只有等待。如此两进程均不能执行完成。称这种现象为**死锁**。

- 例3：

在生产者-消费者问题中将生产者进程的两个wait操作颠倒时会发生死锁。

将消费者进程的两个wait操作颠倒时也会发生死锁。

3.5.1资源问题

1. 可重用性资源和消耗性资源

可重用性资源：是一种可供用户重复使用多次的资源。

可消耗性资源：临界资源，在进程运行期间，由进程动态创建和消耗。

2. 可抢占性资源和不可抢占性资源

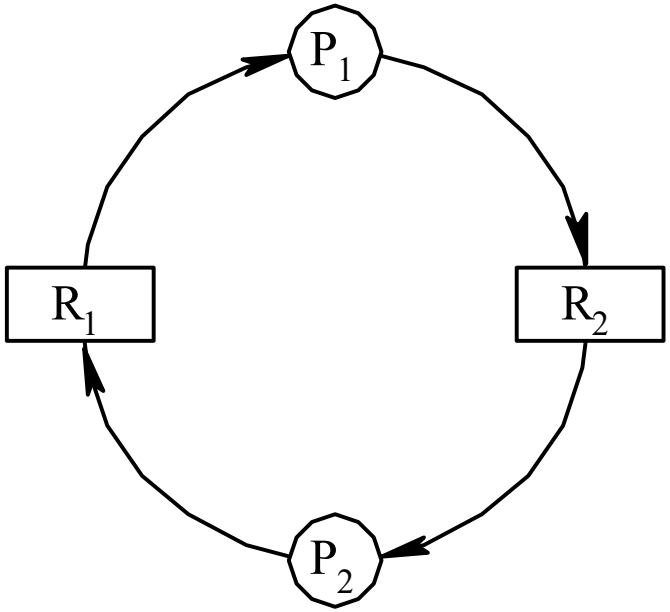
可抢占性资源：进程在获得这类资源后，可再被其他进程或系统抢占不会引起死锁。如：CPU。

不可抢占性资源：系统把某资源分配给该进程，就不能被强行收回，只能在进程用完后自行释放。如：打印机等。

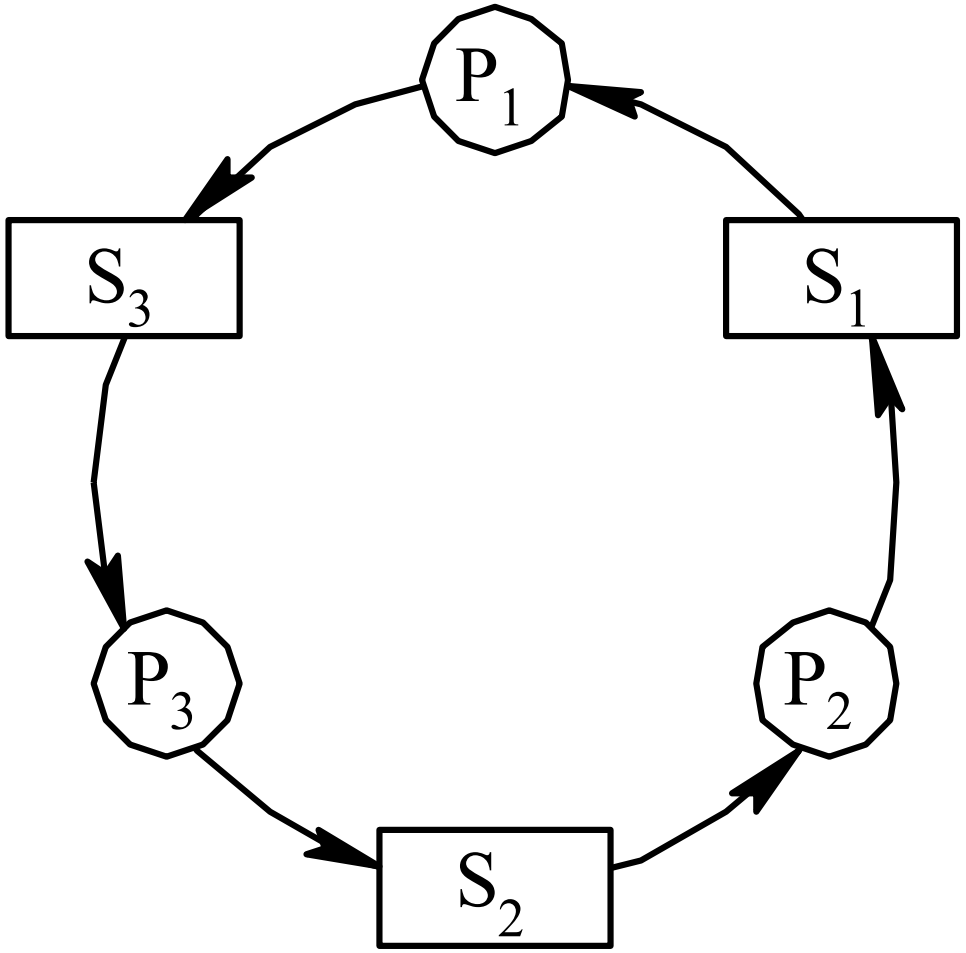
3.5.2 计算机系统中的死锁

死锁产生的原因：

- 竞争不可抢占性资源引起死锁



- 竞争可消耗性资源引起死锁



- 进程推进顺序不当引起死锁

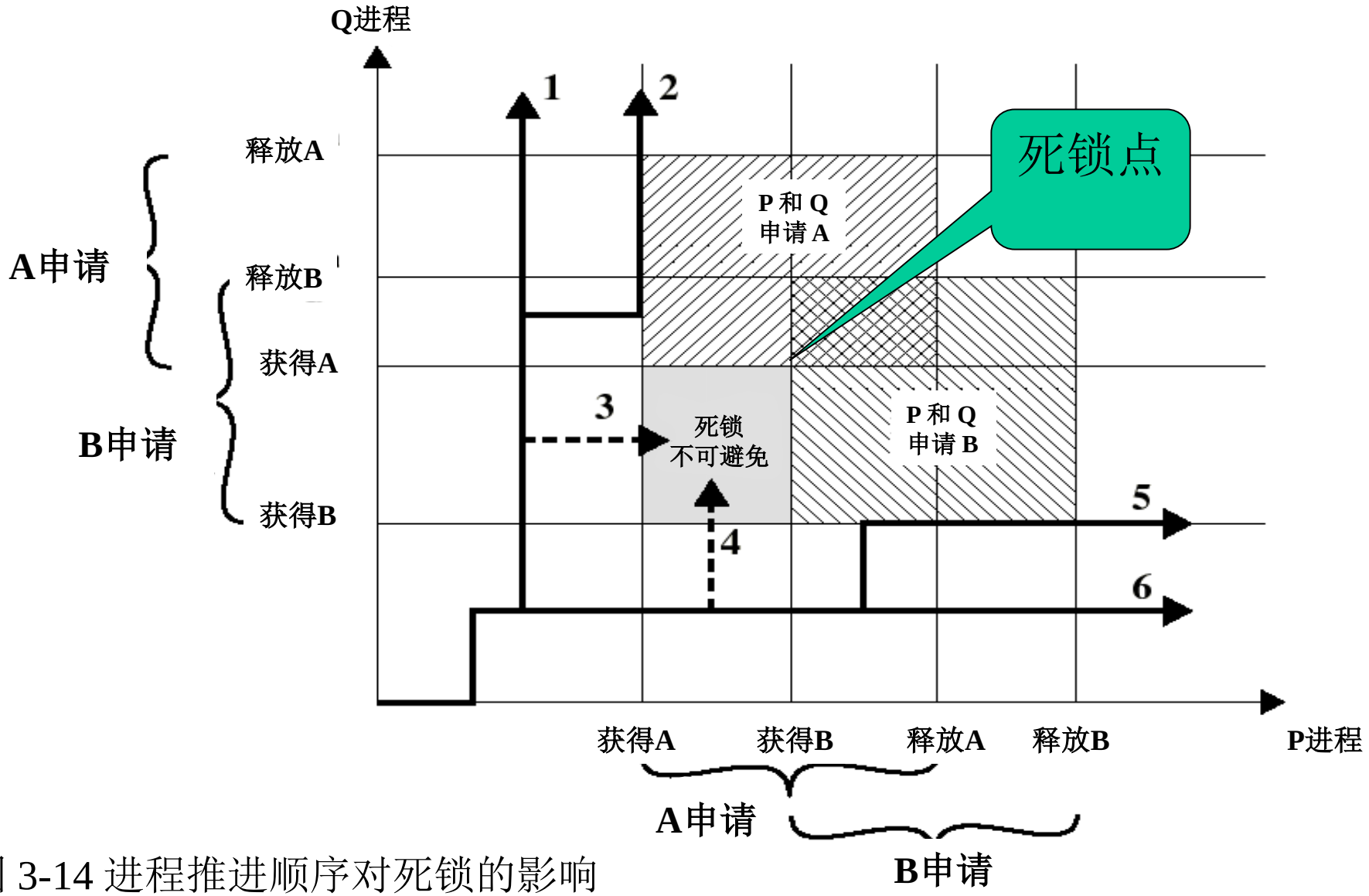
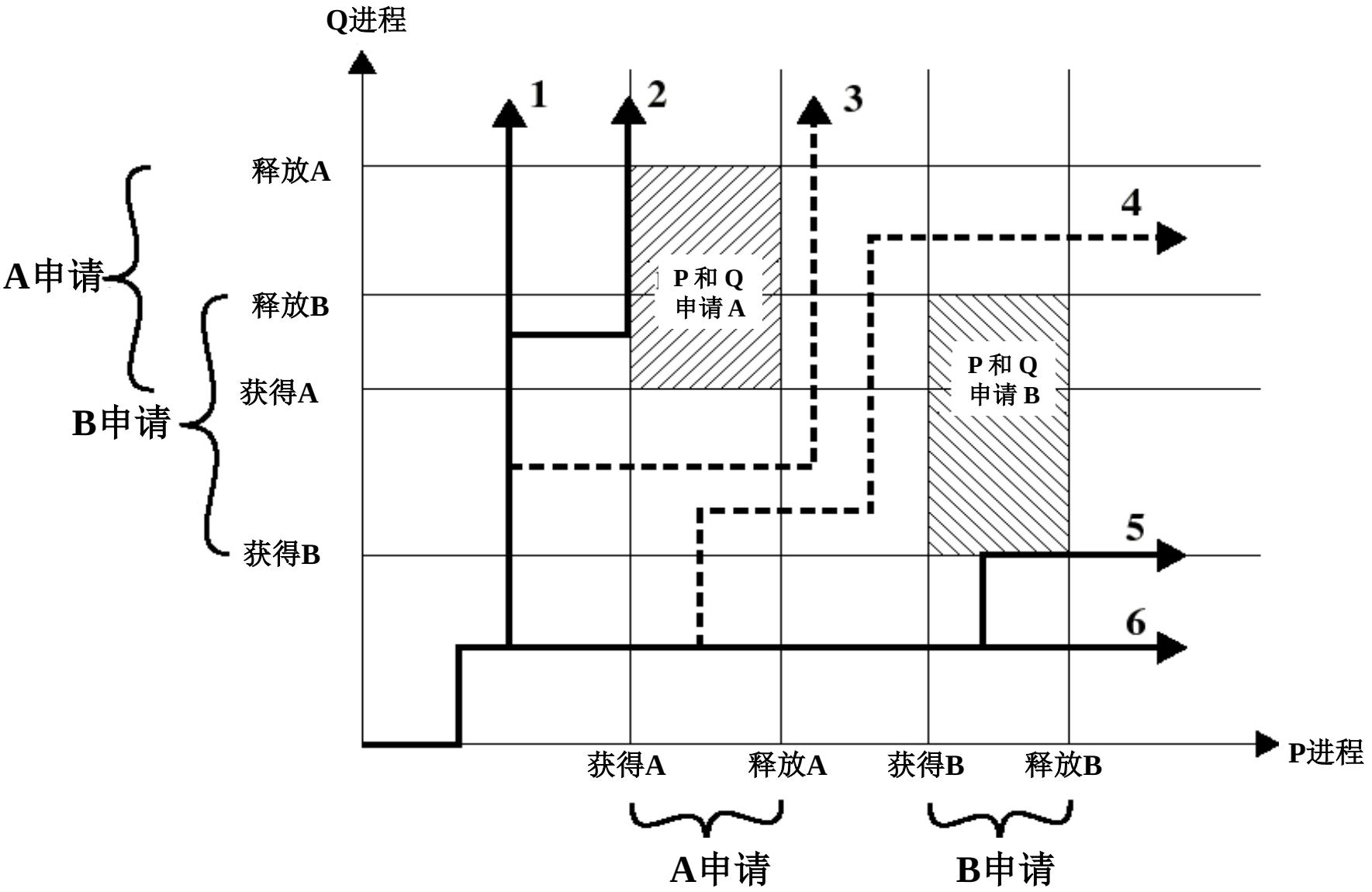


图 3-14 进程推进顺序对死锁的影响

若并发进程P和Q按曲线④所示的顺序推进，它们将进入不安全区D内。此时P保持了资源A, Q保持了资源B, 系统处于不安全状态。因为，这时两进程再向前推进，便可能发生死锁。例如，当P运行到Request(B)时，将因B已被Q占用而阻塞；当Q运行到Request(A)时，也将因A已被P占用而阻塞，于是发生了进程死锁。


Q进程
释放A
A申请
释放B
获得A
B申请
获得B
1
2
3
4
5
6
P 和 Q
申请 A
P 和 Q
申请 B
P进程
获得A
释放A
获得B
释放B
A申请
B申请

3.5.3 死锁的定义、必要条件和处理方法

1.死锁的定义:所谓死锁，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，当进程处于这种僵持状态时，若无外力作用，它们将无法再向前推进。

关于死锁的一些结论：

参与死锁的进程最少是两个。（两个以上进程才会出现死锁）

参与死锁的所有进程都在等待资源。

参与死锁的进程是当前系统中所有进程的子集。

思考题：“死锁”与“进程饥饿”现象有何区别？

2.死锁产生的必要条件

(1) 互斥条件 ：涉及的资源是非共享的。

(2) 请求和保持条件：进程在等待一新资源时继续占有已分配的资源。

(3) 不剥夺条件：不能强行剥夺进程拥有的资源。

(4) 环路等待条件：存在一种进程的循环链，链中的每一个进程已获得的资源同时被链中的下一个进程所请求。

3 .处理死锁的基本方法

(1) 忽略死锁(即鸵鸟算法)

(2)预防死锁：通过设置某些限制条件，去破坏死锁四个必要条件中的一个或多个，来防止死锁。较易实现，广泛使用，但由于所施加的限制往往太严格，可能导致系统资源利用率和系统吞吐量的降低。

(3) 避免死锁：不事先采取限制去破坏产生死锁的条件，而是在资源的动态分配过程中，用某种方法去防止系统进入不安全状态，从而避免死锁的发生。实现较难，只需要较弱的限制条件，可获得较高的资源利用率和系统吞吐量。

(4) 检测死锁:事先并不采取任何限制，也不检查系统是否进入不安全区，允许死锁发生，但可通过检测机构及时检测出死锁的发生，并精确确定与死锁有关的进程和资源，然后采取适当措施，将系统中已发生的死锁清除掉。

(5) 解除死锁：与检测死锁相配套，用于将进程从死锁状态解脱出来。

常用的方法是撤消或挂起一些进程。以回收一些资源，再将它们分配给处于阻塞状态的进程，使之转为就绪状态。实现难度大，但可获得较好的资源利用率和系统吞吐量。

3.6 预防死锁

在系统设计时确定资源分配算法，保证不发生死锁。具体的做法是破坏产生死锁的四个必要条件中的第2、3、4条件之一。“互斥”条件不能破坏。

3.6.1 破坏“请求和保持条件”

1. 第一种协议： 系统要求所有进程要一次性地申请在整个运行过程中所需的全部资源。若系统有足够资源则完全分配。

优点：简单、易于实现且安全。

缺点：

- 一个用户在作业运行之前可能提不出他的作业将要使用的全部设备。
- 可能出现“进程饥饿现象”：用户作业必须等待，直到所有资源满足才能运行。实际上某些资源可能要到运行后期才会用到。
- 资源被严重浪费：一个作业运行期间，对某些设备的使用时间很短，甚至不会用到。如：当用户作业出错时才需要打印机输出错误信息，但采用静态分配法必须把打印机分配给该作业，并长期占用。采用该方法对系统来说是非常浪费的。

2.第二种协议：允许一个进程只获得运行初期所需的资源后，便开始运行，进程运行过程中再逐步释放已分配已分配给自己的、且已用完的全部资源，然后再申请新的所需资源。

相较于第一种协议，提高了资源的利用率、减少了“进程饥饿现象”。

3.6.2 破坏“不可抢占”条件

一个已拥有资源的进程，若它再提出新资源要求而不能立即得到满足时，它必须释放已经拥有的所有资源。以后需要时再重新申请。实现复杂、要付出很大的代价。

缺点：

实现较复杂；抢占资源可能导致进程的前期工作作废。

3.6.3 破坏“循环等待”条件

系统中的所有资源都有一个确定的唯一号码，所有分配请求必须以序号上升的次序进行。

优点：同前两法相比，其资源利用率和系统吞吐量有较明显的改善。

缺点：进程实际需要资源的顺序不一定与资源的编号一致，因此仍会造成资源浪费。

3.7 避免死锁

3.7.1 系统安全状态

1. 安全状态

在避免死锁的方法中，允许进程动态地申请资源，但系统在进行资源分配之前，应先计算此次资源分配的安全性。若此次分配不会导致系统进入不安全状态，则将资源分配给进程；否则，令进程等待。■

所谓安全状态，是指系统能按某种进程顺序(P_1，P_2，…，P_n)(称〈P_1, P_2, ..., P_n〉序列为安全序列)，来为每个进程P_i分配其所需资源，直至满足每个进程对资源的最大需求，使每个进程都可顺利地完成。如果系统无法找到这样一个安全序列，则称系统处于不安全状态。

2. 安全状态之例 ■

我们通过一个例子来说明安全性。假定系统中有三个进程P_1、 P_2和P_3，共有12台磁带机。进程P_1总共要求10台磁带机，P_2和P_3分别要求4台和9台。假设在T_0时刻，进程P_1、P_2和P_3已分别获得5台、2台和2台磁带机，尚有3台空闲未分配，如下表所示：

进 程	最 大 需 求	已 分 配	可 用
P_1	10	5	3
P_2	4	2	
P_3	9	2	

3. 由安全状态向不安全状态的转换 ■

如果不按照安全序列分配资源，则系统可能会由安全状态进入不安全状态。例如，在T_0时刻以后，P_3又请求1台磁带机，若此时系统把剩余3台中的1台分配给P_3，则系统便进入不安全状态。 因为，此时也无法再找到一个安全序列， 例如，把其余的2台分配给P_2，这样，在P_2完成后只能释放出4台，既不能满足P_1尚需5台的要求，也不能满足P_3尚需6台的要求，致使它们都无法推进到完成，彼此都在等待对方释放资源，即陷入僵局，结果导致死锁。

3.7.2 利用银行家算法避免死锁

1. 银行家算法中的数据结构

(1) 可利用资源向量Available。这是一个含有*m*个元素的数组，其中的每一个元素代表一类可利用的资源数目，其初始值是系统中所配置的该类全部可用资源的数目，其数值随该类资源的分配和回收而动态地改变。如果Available［j］=K，则表示系统中现有R_j类资源*K*个。

(2) 最大需求矩阵Max。这是一个$n\times m$的矩阵，它定义了系统中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R_j类资源的最大数目为K 。

(3) 分配矩阵Allocation。这也是一个$n\times m$的矩阵，它定义了系统中每一类资源当前已分配给每一进程的资源数。如果Allocation［i,j］=K，则表示进程i当前已分得R_j类资源的数目为K。 ■

(4) 需求矩阵Need。这也是一个$n\times m$的矩阵，用以表示每一个进程尚需的各类资源数。如果Need［i,j］=K，则表示进程i还需要R_j类资源K个，方能完成其任务。

Need［i,j］=Max［i,j］-Allocation［i,j］

2. 银行家算法

设$Request_i$是进程P_i的请求向量，如果$Request_i$［j］=*K*，表示进程P_i需要*K*个R_j类型的资源。当P_i发出资源请求后，系统按下述步骤进行检查：■

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

(2) 如果$Request_i$［j］≤Available［j］，便转向步骤(3)；否则， 表示尚无足够资源，P_i须等待。

(3) 系统试探着把资源分配给进程P_i，并修改下面数据结构中的数值：■

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］;■

(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。若安全，才正式将资源分配给进程P_i，以完成本次分配；否则， 将本次的试探分配作废，恢复原来的资源分配状态，让进程P_i等待。

3. 安全性算法

(1) 设置两个向量：① 工作向量Work: 它表示系统可提供给进程继续运行所需的各类资源数目，它含有*m*个元素，在执行安全算法开始时，Work：=Available; ② Finish: 它表示系统是否有足够的资源分配给进程，使之运行完成。开始时先做Finish［i］：=false; 当有足够资源分配给进程时，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： ■

① Finish［i］=false; ② 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［i］+Allocation［i,j］; ■

Finish［i］：= true; ■

go to 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。

4. 银行家算法之例

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图 3-15 所示。

进程 \ 资源情况	Max A B C	Allocation A B C	Need A B C	Available A B C
P_0	7 5 3	0 1 0	7 4 3	3 3 2
				(2 3 0)
P_1	3 2 2	2 0 0	1 2 2	
		(3 0 2)	(0 2 0)	
P_2	9 0 2	3 0 2	6 0 0	
P_3	2 2 2	2 1 1	0 1 1	
P_4	4 3 3	0 0 2	4 3 1	

图 3-15 T_0时刻的资源分配表

(1) T_0时刻的安全性：

进程 \ 资源情况	Work A	Work B	Work C	Need A	Need B	Need C	Allocation A	Allocation B	Allocation C	Work+Allocation A	Work+Allocation B	Work+Allocation C	Finish
P_1	3	3	2	1	2	2	2	0	0	5	3	2	true
P_3	5	3	2	0	1	1	2	1	1	7	4	3	true
P_4	7	4	3	4	3	1	0	0	2	7	4	5	true
P_2	7	4	5	6	0	0	3	0	2	10	4	7	true
P_0	10	4	7	7	4	3	0	1	0	10	5	7	true

图 3-16 T_0时刻的安全序列

(2) P_1请求资源：P_1发出请求向量$Request_1(1，0，2)$，系统按银行家算法进行检查：

① $Request_1(1, 0, 2) \leq Need_1(1, 2, 2)$ ■

② $Request_1(1, 0, 2) \leq Available_1(3, 3, 2)$ ■

③ 系统先假定可为P_1分配资源，并修改Available, $Allocation_1$和$Need_1$向量，由此形成的资源变化情况如图 3-15 中的圆括号所示。 ■

④ 再利用安全性算法检查此时系统是否安全。

进程 \ 资源情况	Work A B C	Need A B C	Allocation A B C	Work+Allocation A B C	Finish
P_1	2 3 0	0 2 0	3 0 2	5 3 2	true
P_3	5 3 2	0 1 1	2 1 1	7 4 3	true
P_4	7 4 3	4 3 1	0 0 2	7 4 5	true
P_0	7 4 5	7 4 3	0 1 0	7 5 5	true
P_2	7 5 5	6 0 0	3 0 2	10 5 7	true

图 3-17 P_1申请资源时的安全性检查

(3) P_4请求资源：P_4发出请求向量$Request_4$(3，3，0)，系统按银行家算法进行检查：■

① $Request_4(3, 3, 0) \leq Need_4(4, 3, 1)$; ■

② $Request_4(3, 3, 0) \prec\neq Available(2, 3, 0)$，让$P_4$等待。■ (4) P_0请求资源：P_0发出请求向量$Requst_0$(0，2，0)，系统按银行家算法进行检查：■

① $Request_0(0, 2, 0) \leq Need \quad 0(7, 4, 3)$; ■

② $Request_0(0, 2, 0) \leq Available(2, 3, 0)$; ■

③ 系统暂时先假定可为P_0分配资源，并修改有关数据，如图 3-18 所示。

进程 \ 资源情况	Allocation A B C	Need A B C	Available A B C
P_0	0 3 0	7 2 3	2 1 0
P_1	3 0 2	0 2 0	
P_2	3 0 2	6 0 0	
P_3	2 1 1	0 1 1	
P_4	0 0 2	4 3 1	

图 3-18 为P_0分配资源后的有关资源数据

3.8 死锁的检测与解除

3.8.1 死锁的检测

允许死锁发生，操作系统不断监视系统进展情况，判断死锁是否发生。

一旦死锁发生则采取专门的措施，解除死锁并以最小的代价恢复操作系统运行。

1. 资源分配图(Resource Allocation Graph)

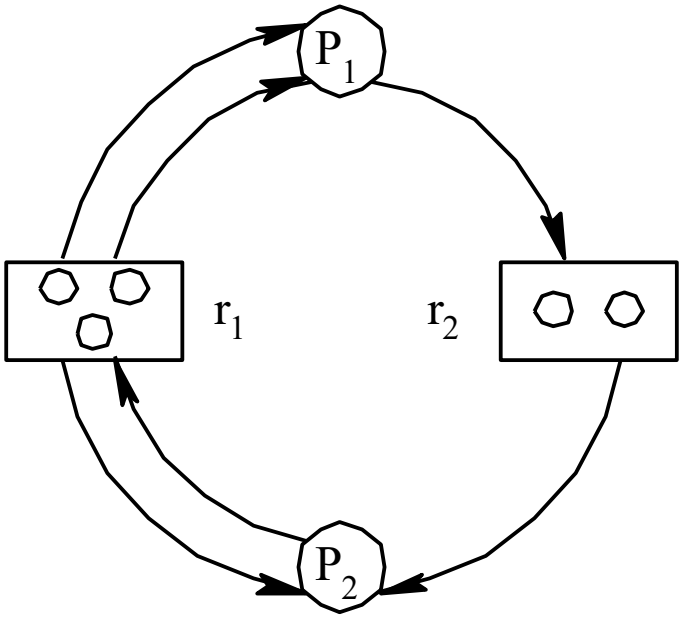


图 3-19 每类资源有多个时的情况

资源分配图G=(N,E)：N表示一组结点；E表示一组边。

(1) 把N分成两个互斥子集，进程结点P ={p_1, P_2，…，p_n}和资源结点R ={r_1, r_2，…，r_n}

(2) 凡属于E中的一个边e∈E，都连接着P中的一个结点和R中的一个结点，e={p_i, r_j}是资源请求边，由进程p_i指向资源r_j， 它表示进程p_i请求一个单位的r_j资源。e={r_j, p_i}是资源分配边，由资源r_j指向进程p_i，它表示把一个单位的资源r_j分配给进程p_i。

2. 死锁定理

如果资源分配图中没有环路，则系统中没有死锁，如果图中存在环路则系统中可能存在死锁。

如果每个资源类中只包含一个资源实例，则环路是死锁存在的充分必要条件。

资源分配图化简

1）找一个非孤立点进程结点且只有分配边，去掉分配边，将其变为孤立结点

2）再把相应的资源分配给一个等待该资源的进程，即将某进程的申请边变为分配边

3）重复以上步骤，若所有进程成为孤立结点，称该图是可完全简化的，否则称该图是不可完全简化的。

死锁状态的充分条件是：当且仅当资源分配图是不可完全简化的。

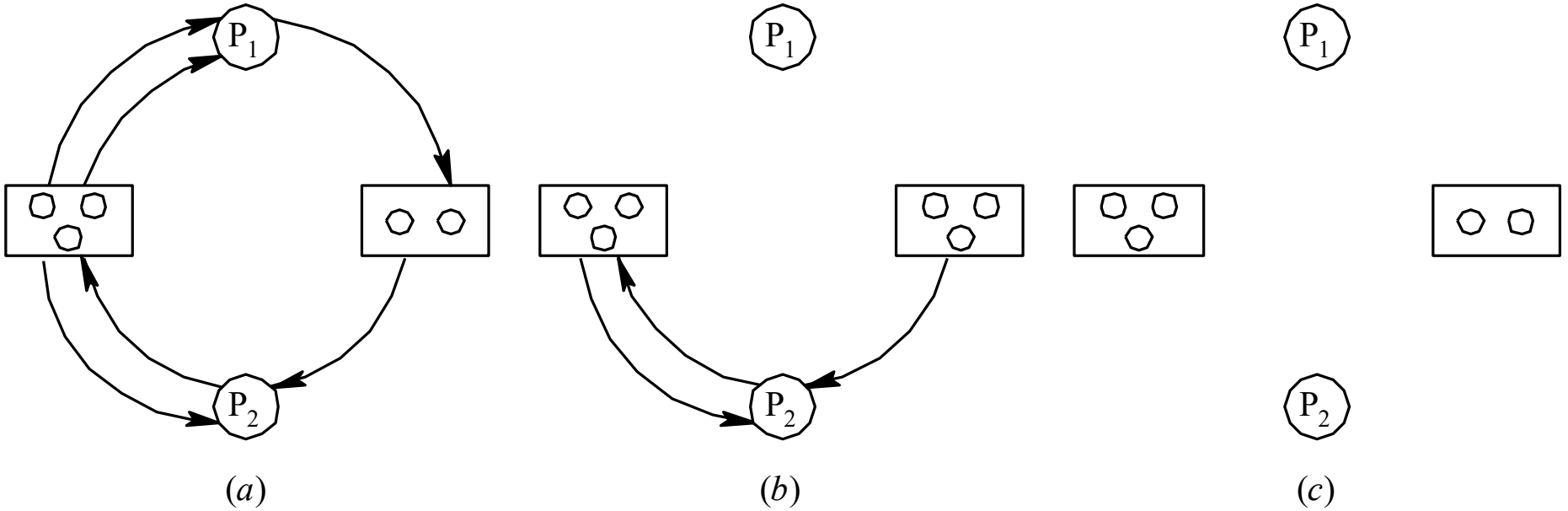
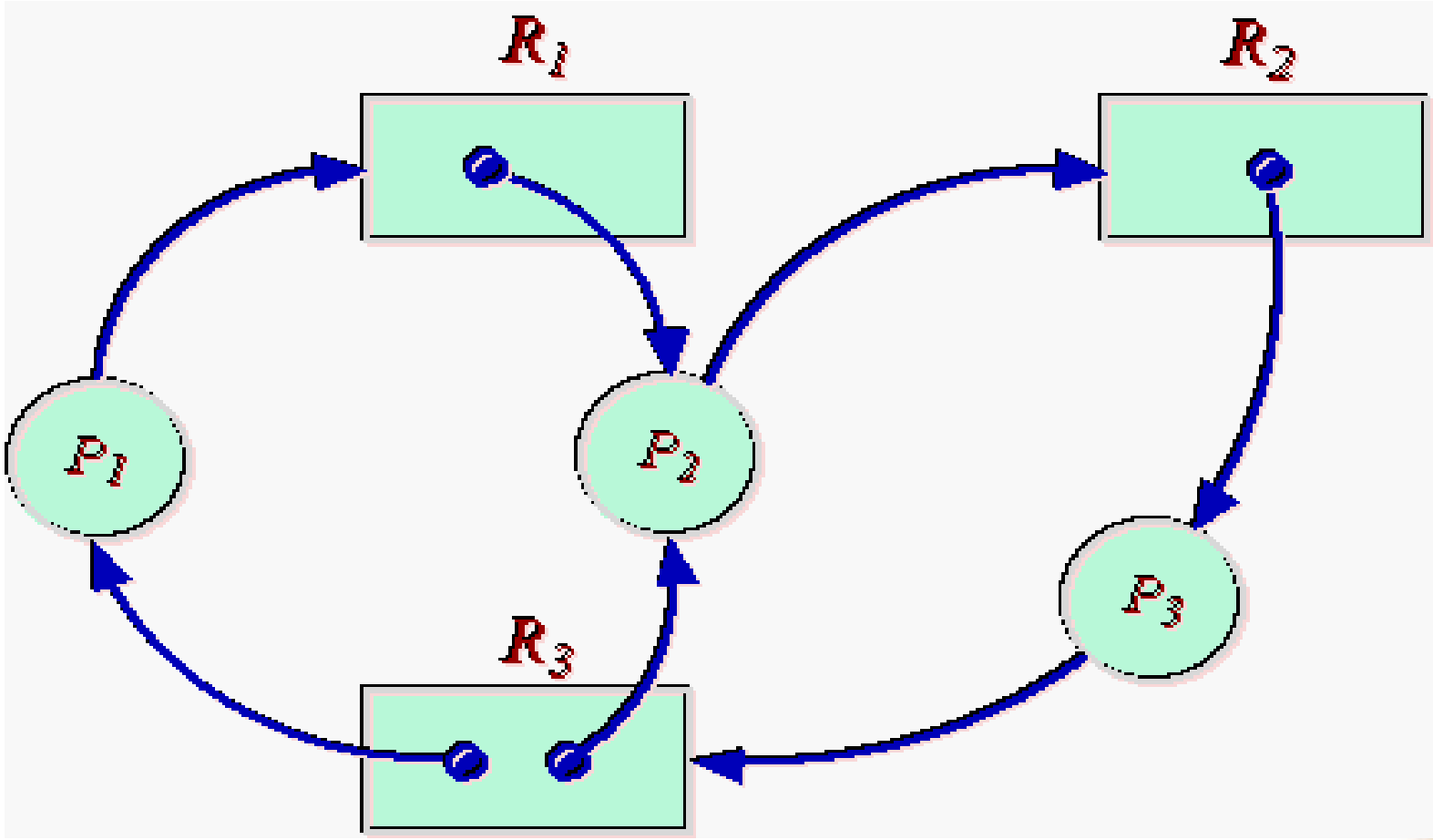
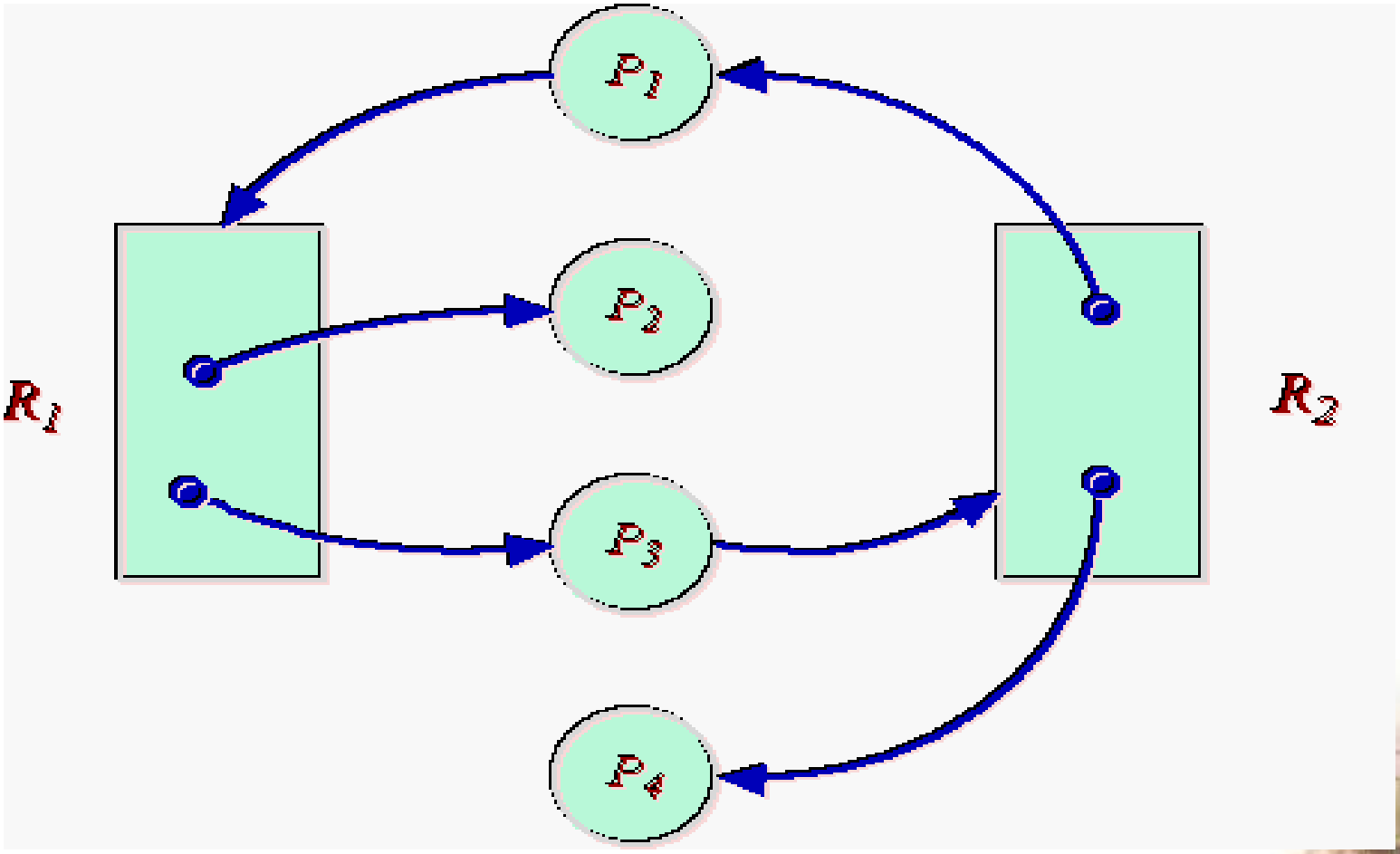


图 3-20 资源分配图的简化

例1：有环有死锁



例2：有环无死锁



3. 死锁检测中的数据结构

(1) 可利用资源向量Available，它表示了*m*类资源中每一类资源的可用数目。 ■

(2) 把不占用资源的进程(向量Allocation：=0)记入L表中，即$L_i \cup L$。 ■

(3) 从进程集合中找到一个$Request_i \leq Work$的进程，做如下处理：① 将其资源分配图简化，释放出资源，增加工作向量Work：=Work+$Allocation_i$。 ② 将它记入L表中。

(4) 若不能把所有进程都记入L表中， 便表明系统状态S的资源分配图是不可完全简化的。 因此，该系统状态将发生死锁。Work ：=Available; ■

L ：= $\{L_i|Allocation_i=0 \cap Request_i=0\}$ ■

for all $L_i \notin L$ do ■

begin ■

for all $Request_i \leq Work$ do ■

begin ■

Work ：=Work+$Allocation_i$; ■

$L_i \cup L$; ■

end ■

end ■

deadlock： = ($L=\{p_1, p_2, \ldots, p_n\}$);

检测时机：

- 当进程等待时检测死锁。
  （其缺点是系统的开销大。）
- 定时检测。
- 系统资源利用率下降时检测死锁。

3.8.2 死锁的解除

(1) 剥夺资源。

(2) 撤消进程。

1.终止进程的方法

1）终止所有死锁进程

2) 逐个终止进程，直到死锁解除。

2. 付出代价最小的死锁解除算法

为把系统从死锁状态中解脱出来，所花费的代价可表示为：

$$R(S)_{min}=min\{C_{ui}\}+min\{C_{uj}\}+min\{C_{uk}\}+\cdots$$

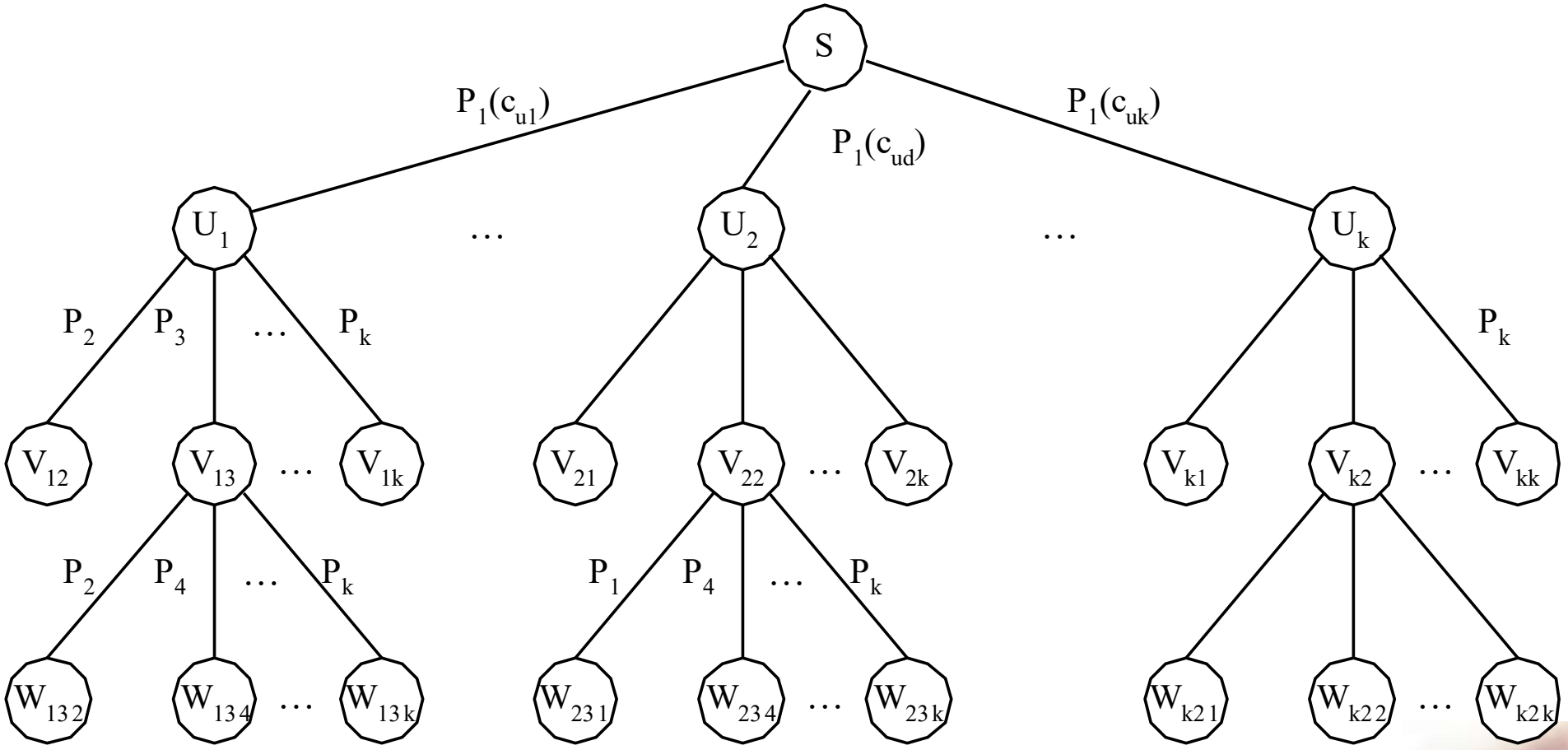


图 3-21 付出代价最小的死锁解除方法