

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

3.2 作业与作业调度

3.3 进程调度

3.4 实时调度

3.5 死锁概述

3.6 预防死锁

3.7 避免死锁

3.8 死锁的检测与解除



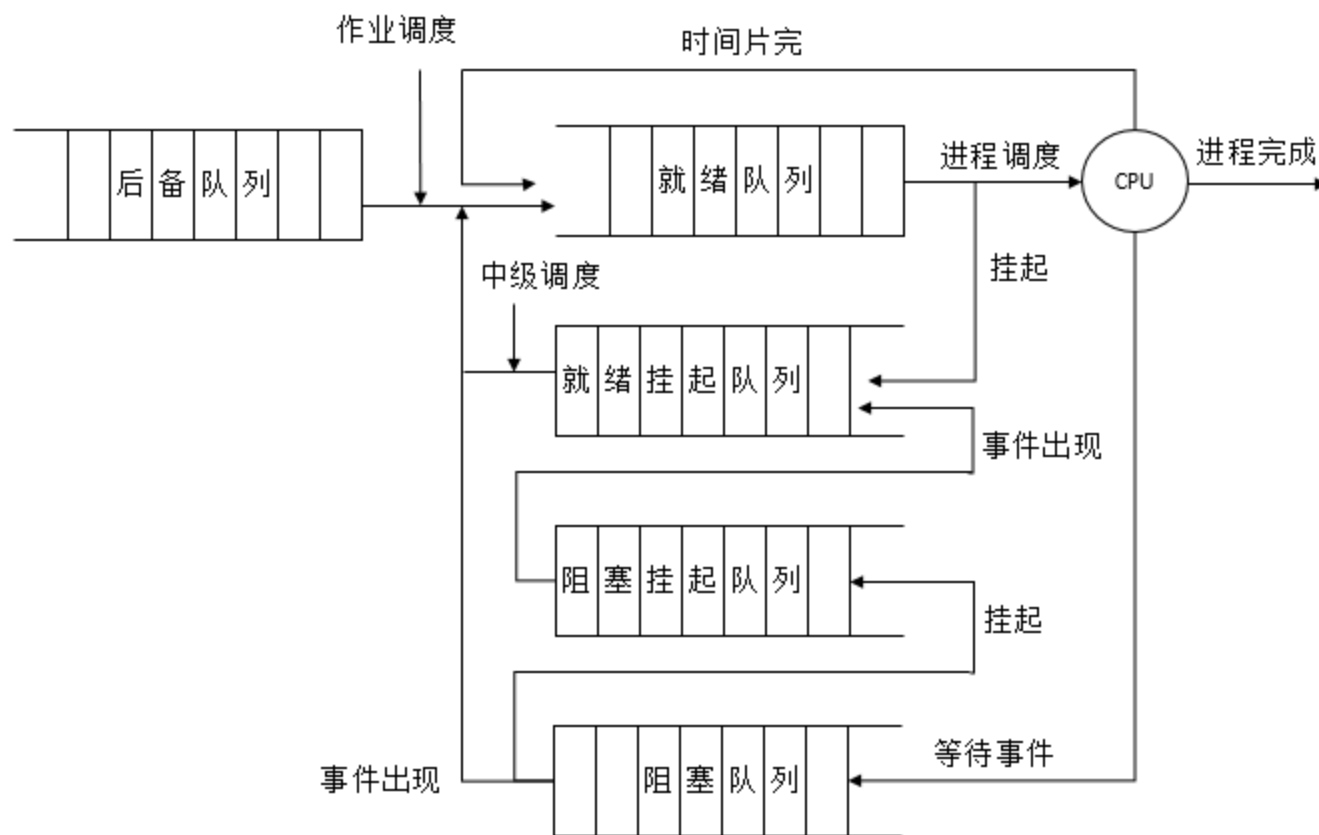
3.1 处理机调度的层次和调度算法的目标

在多道程序系统中，调度的实质是一种资源分配，处理机调度是对处理机资源进行分配。

处理机调度算法是指根据处理机分配策略所规定的处理机分配算法。

在多道批处理系统中，一个作业从提交到获得处理机执行，直至作业运行完毕，可能需要经历多级处理机调度。

3.1.1 处理机调度的层次



处理机的三级调度模型

1. 高级调度(High Level Scheduling)

高级调度又称长程调度或作业调度，它的调度对象是作业。

主要功能是根据某种算法，决定将外存上处于后备队列的哪几个作业中调入内存，为它们创建进程、分配资源，并将它们放入就绪队列。

高级调度主要用于多道批处理系统中，而在分时和实时系统中不设置高级调度。作业调度的执行频率较低且时间较长，通常为几分钟一次。



2. 低级调度(High Level Scheduling)

低级调度又称进程调度或短程调度，其所调度的对象是
进程（或内核级线程）。

根据某种算法，决定就绪队列中的哪个进程应获得处理机，并由分派程序将处理机分配给被选中的进程。

进程调度是最基本的一种调度，在多道批处理、分时和实时三种类型的OS中，都必须配置这级调度。

3. 中级调度 (Intermediate Scheduling)

中级调度又称内存调度。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**。

系统会把那些暂时不能运行的进程，调至外存等待，此时进程的状态称为就绪驻外状态（或者挂起状态）。当它们已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那些已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。**

中级调度实际上就是存储器管理中的**对换**功能。



	进程调度	作业调度	中级调度
运行频率	10-100ms一次	约几分钟一次	介于前两者之间
算法复杂程度	不宜太复杂	允许花费的时间较多	介于前两者之间
发生的时间	不一定在进程执行完后发生，由调度算法决定。	多在一个作业运行完毕后发生	内存有空闲，挂起的进程换入内存时发生



三层调度的联系、对比

	要做什么	调度发生在..	发生频率	对进程状态的影响
高级调度 (作业调度)	按照某种规则, 从后备队列中选择合适的作业将其调入内存, 并为其创建进程	外存→内存 (面向作业)	最低	无→创建态→就绪态
中级调度 (内存调度)	按照某种规则, 从挂起队列中选择合适的进程将其数据调回内存	外存→内存 (面向进程)	中等	挂起态→就绪态 (阻塞挂起→阻塞态)
低级调度 (进程调度)	按照某种规则, 从就绪队列中选择一个进程为其分配处理机	内存→CPU	最高	就绪态→运行态

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。为提高系统的资源利用率，应使系统中的处理机和其它所有资源都尽可能地保持忙碌状态，其中最重要的处理机利用率可用以下方法计算：

$$\text{CPU 的利用率} = \frac{\text{CPU 有效工作时间}}{\text{CPU 有效工作时间} + \text{CPU 空闲等待时间}}$$



(2) 公平性。公平性是指应使诸进程都获得合理的CPU时间，不会发生进程饥饿现象。

(3) 平衡性。调度算法应尽可能保持系统资源使用的平衡性。

(4) 策略强制执行。对所制订的策略其中包括安全策略，只要需要，就必须予以准确地执行，即使会造成某些工作的延迟也要执行。







2. 批处理系统的目标

(1) 平均周转时间短。

周转时间：从作业提交给系统开始，到作业完成为止的这段时间间隔（称为作业周转时间）。包括四部分时间：

- ①作业在外存后备队列上**等待作业调度**（高级调度）的时间；
- ②进程在就绪队列上**等待进程调度**（低级调度）的时间；
- ③进程在CPU上**执行的时间**；
- ④进程等待**I/O操作的时间**。



作为计算机系统的管理者，则总是希望能使平均周转时间最短。可把平均周转时间描述为：

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

	作业1	作业2	作业3	作业4	作业5
周转时间	6	5	8	9	12

$$T = (6 + 5 + 8 + 9 + 12) / 5 = 8$$





带权周转时间：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 $W = T/T_s$ （ $W \geq 1$ ）。

而平均带权周转时间则可表示为：

	作业1	作业2	作业3	作业4	作业5
周转时间	6	5	8	9	12
获得服务时间	4	3	6	3	6
W	1.5	1.6	1.33	3	2

$$W = (1.5 + 1.6 + 1.33 + 3 + 2) / 5 = 1.89$$







(2) 系统吞吐量高。吞吐量：指在单位时间内系统所完成的作业数，因而它与批处理作业的平均长度有关。

(3) 处理机利用率高。

处理机的利用率是衡量系统性能的十分重要的指标；而调度方式和算法对处理机的利用率起着十分重要的作用。

在实际系统中，CPU的利用率一般在40%(系统负荷较轻)到90%之间。



3. 分时系统的目标

(1) 响应时间快：响应时间指从用户通过键盘提交的一个请求开始，直到屏幕上显示出处理结果位置的一段时间间隔。

(2) 均衡性：指系统响应时间的快慢应与用户所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：截止时间指某任务必须开始执行的最迟时间，或必须完成的最迟时间。对于严格的实时系统，其调度方式和调度算法必须能保证这一点，否则可能造成难以预料的后果。

(2) 可预测性。

3.2 作业与作业调度

在多道批处理系统中，**作业**是用户提交给系统的一项**相对独立的工作**。操作员把用户提交的作业通过相应的输入设备输入到磁盘存储器，并保存在一个后备作业队列中。再由作业调度程序将其从外存调入内存。



3.2.1 批处理系统中的作业

1. 作业和作业步

(1) 作业(Job)。

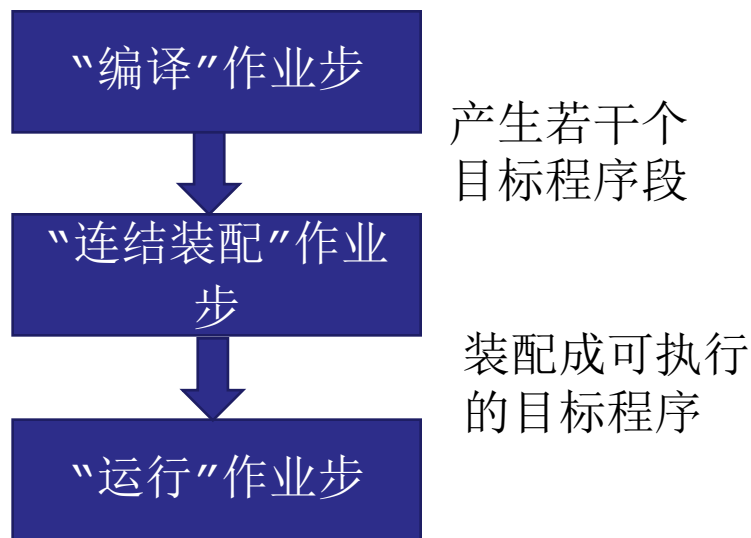
①包含了通常的**程序**和**数据**,

②配有一份**作业说明书**, 系统根据该说明书来对程序的运行进行控制。

只有在批处理系统中, 才有在作业这一概念, 批处理系统是以作业为基本单位从外存调入内存的。

(2) 作业步(Job Step)

通常，在作业运行期间，每个作业都必须经过若干个相对独立，又相互关联的顺序加工步骤才能得到结果，其中的每一个加工步骤称为一个作业步。各作业步之间存在着相互联系，往往是把上一个作业步的输出作为下一个作业步的输入。



2. 作业控制块 (Job

Control Block, JCB)



在多道批处理系统中，为每个作业设置了一个作业控制块JCB，它是作业在系统中存在的标志，其中保存了系统对作业进行管理和调度所需的全部信息。

JCB中所包含的内容因系统而异，通常应包含：

JCB:

- 作业标识
- 用户名称
- 用户帐户
- 作业类型(CPU 繁忙型、I/O 繁忙型、批量型、终端型)
- 作业状态
- 调度信息(优先级、作业已运行时间)
- 资源需求(预计运行时间、要求内存大小、要求I/O设备的类型和数量等)
- 进入系统时间
- 开始处理时间
- 作业完成时间
- 作业退出时间
- 资源使用情况

.....



(1) 每当作业进入系统时，系统便为每个作业建立一个JCB，根据作业类型将它插入相应的后备队列中。作业调度程序依据一定的调度算法来调度它们，被调度到的作业将会装入内存。

(2) 在作业运行期间，系统就按照JCB中的信息对作业进行控制。

(3) 当一个作业执行结束进入完成状态时，系统负责回收分配给它的资源，撤消它的作业控制块。



3. 作业运行的三个阶段和三种状态


作业从进入系统到运行结束，通常需要经历收容、运行和完成三个阶段。相应的作业也就有“后备状态”、“运行状态”和“完成状态”。

- (1) 收容阶段： 后备状态
- (2) 运行阶段： 运行状态
- (3) 完成阶段： 完成状态



3.2.2 作业调度的主要任务

作业调度的主要任务是，根据JCB中的信息，检查系统中的资源能否满足作业对资源的需求，以及按照一定的调度算法，从外存的后备队列中选取某些作业调入内存，并为它们创建进程、分配必要的资源。然后再将新创建的进程排在就绪队列上等待调度。因此，也把作业调度称为接纳调度(Admission Scheduling)。




在每次执行作业调度时，都须做出以下两个决定。

1) 决定接纳多少个作业

多道程序度(Degree of Multiprogramming): 允许多
少个作业同时在内存中运行?

- 当内存中同时运行的作业数目太多时，可能会影响到系统的服务质量
- 在内存中同时运行作业的数量太少时，又会导致系统的资源利用率和系统吞吐量太低

多道程序度的确定应根据系统的规模和运行速度等情况做适当的折衷。



2) 决定接纳哪些作业

应将哪些作业从外存调入内存，这将取决于所采用的调度算法。

3.2.3 先来先服务(FCFS)和短作业优先(SJF)调度算法

1. 先来先服务(first-come first-served, FCFS)调度算法

FCFS是最简单的调度算法，**该算法既可用于作业调度，也可用于进程调度**。当在作业调度中采用该算法时，系统将按照作业到达的先后次序来进行调度，或者说它是优先考虑在系统中等待时间最长的作业，而不管该作业所需执行时间的长短，从后备作业队列中**选择几个**最先进入该队列的作业，将它们调入内存，为它们分配资源和创建进程。然后把它放入就绪队列。

FCFS算法较有利于长作业(进程)，而不利于短作业(进程)
 （会使短作业（进程）的周转时间长，主要是可能会等待前面的
 的长作业（进程））。

例：

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				
B	1	100				
C	2	1				
D	3	100				

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

2. 短作业优先(short job first, SJF)的调度算法

由于在实际情况中,短作业(进程)占有很大比例,为了能使它们能比长作业优先执行,而产生了短作业优先调度算法。

1) 短作业优先算法

SJF算法是以作业的长短来计算优先级,作业越短,其优先级越高。作业的长短是以作业所要求的运行时间来衡量的。**SJF算法可以分别用于作业调度和进程调度。**在把短作业优先调度算法用于作业调度时,它将从外存的作业后备队列中选择若干个估计运行时间最短的作业,优先将它们调入内存运行。



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						
	周转时间						
	带权周转时间						
SJF (b)	完成时间						
	周转时间						
	带权周转时间						



<div> <div>作业 情况</div> <div>调度 算法</div> </div>	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	3.2	1.5	2.25	2.1



2) 短作业优先算法的缺点

SJF调度算法较之FCFS算法有了明显的改进，但仍然存在不容忽视的缺点：

(1) **必须预知作业的运行时间**。在采用这种算法时，要先知道每个作业的运行时间。

(2) **对长作业非常不利**。长作业的周转时间会明显地增长。更严重的是，该算法完全忽视作业的等待时间，可能使作业等待时间过长，**出现饥饿现象**。

(3) 在采用SJF算法时，**人一机无法实现交互**。



(4) 该调度算法完全未考虑作业的紧迫程度，故不能保证紧迫性作业能得到及时处理。

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(priority-scheduling algorithm, PSA)

基于作业的紧迫程度，由外部赋予作业相应的优先级，调度算法是根据该优先级调度的。这样，就可以保证紧迫性作业优先运行。

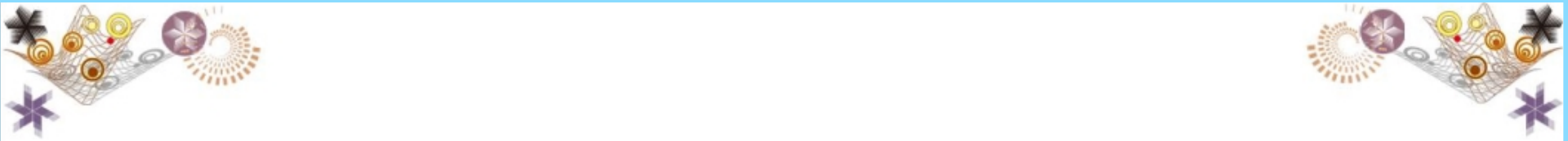
优先级调度算法可作为作业调度算法，也可作为进程调度算法。当把该算法用于作业调度时，系统是从后备队列中选择若干个优先级最高的作业装入内存。



2. 高响应比优先调度算法 (Highest Response Ratio Next, HRRN)

Next, HRRN)

为每个作业引入一个动态优先级，即优先级是可以改变的，令它随等待时间延长而增加，这将使长作业的优先级在等待期间不断地增加，等到足够的时间后，必然有机会获得处理机。该优先级的变化规律可描述为：



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**，故该优先级又相当于响应比 R_p 。据此，优先又可表示为：

可以看出，该算法实现了较好的折中：

- 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，因而类似于SJF算法，有利于短作业；
- ②当要求服务的时间相同时，作业的优先权又决定于其等待时间，因而该算法又类似于FCFS算法；
- ③对于长作业的优先级，可以随等待时间的增加而提高，当其等待时间足够长时，也可获得处理机。

高响应比优先调度算法（HRRN）举例

进程	到达时间	运行时间
P1	0	7
P2	2	4
P3	4	1
P4	5	4



0时刻：只有 P_1 到达就绪队列， P_1 上处理机

7时刻（ P_1 主动放弃CPU）：就绪队列中有 P_2 (响应比 $= (5+4)/4=2.25$)、 P_3 ($((3+1)/1=4)$)、 P_4 ($((2+4)/4=1.5)$)，

8时刻（ P_3 完成）： P_2 (2.5)、 P_4 (1.75)

12时刻（ P_2 完成）：就绪队列中只剩下 P_4



3.3 进 程 调 度

3.3.1 进程调度的任务、机制和方式

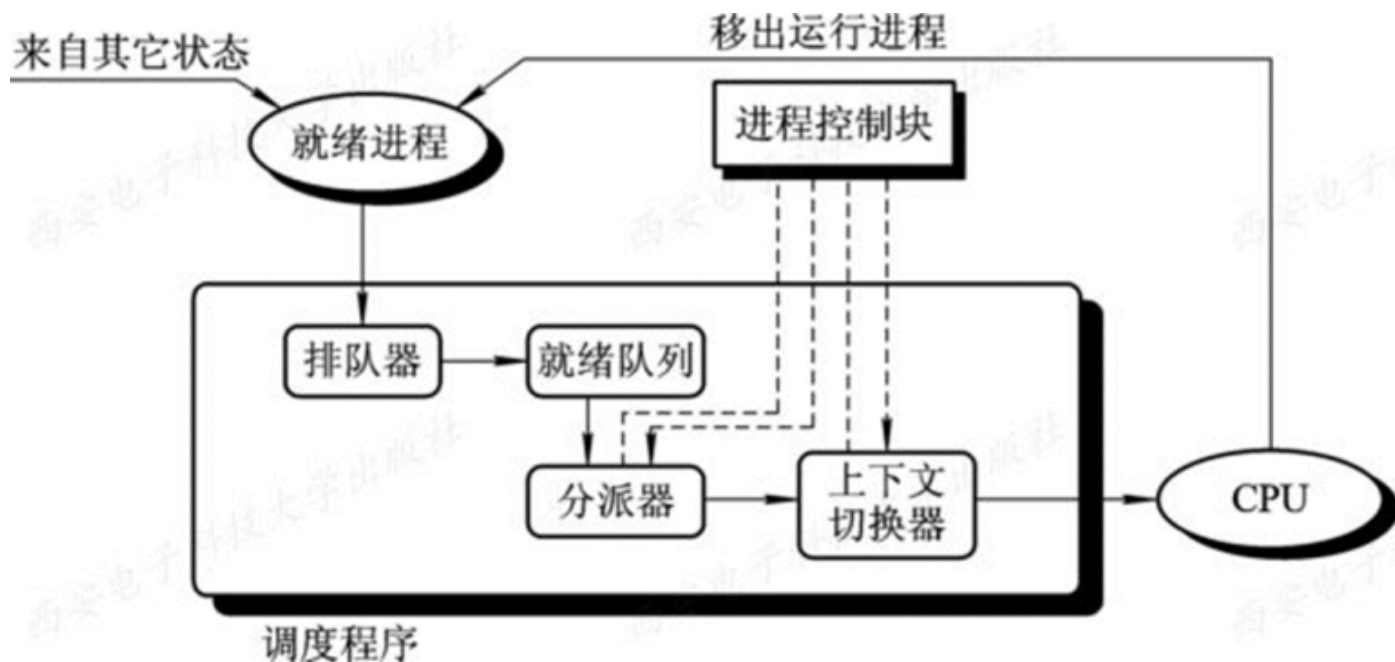
1. 进程调度的任务

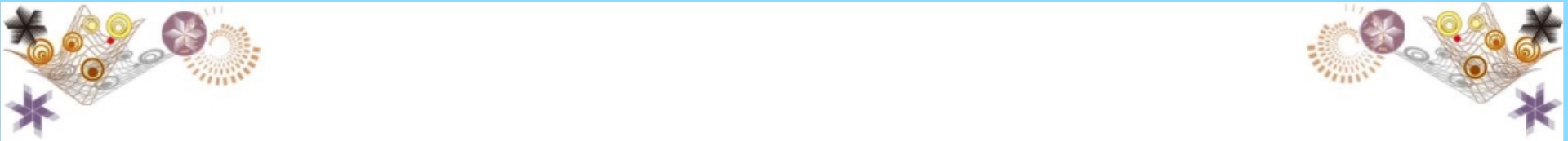
进程调度的主要有三项任务：

- (1) 保存处理机的现场信息
- (2) 按某种算法选取进程
- (3) 把处理器分配给进程

2. 进程调度机制

为了实现进程调度，在进程调度机制中，应具有如下三个基本部分：(1) 排队器；(2) 分派器；(3) 上下文切换器。



- 
- (1) 排队器。将系统中的所有就绪进程按照一定的策略排成一个或多个队列，以便调度程序选择。每当有一个进程转变为就绪态时，排队器就将它插入到相应的就绪队列。
- (2) 分派器。根据进程调度程序所选定的进程，将其从就绪队列中取出，然后进行从分派器到新选出进程间的上下文切换，将处理机分配给新选出的进程。
- (3) 上下文切换器。两次上下文的切换：①保存当前进程的上下文到该进程的PCB中，装入分派程序的上下文，以便分派程序运行；②移出分派程序的上下文，把新选进程的CPU现场信息恢复到处理机寄存器中，以便新选进程运行。

当前进程 $\xrightarrow{\text{上下文切换}}$ 分派程序 $\xrightarrow{\text{上下文切换}}$ 新选进程



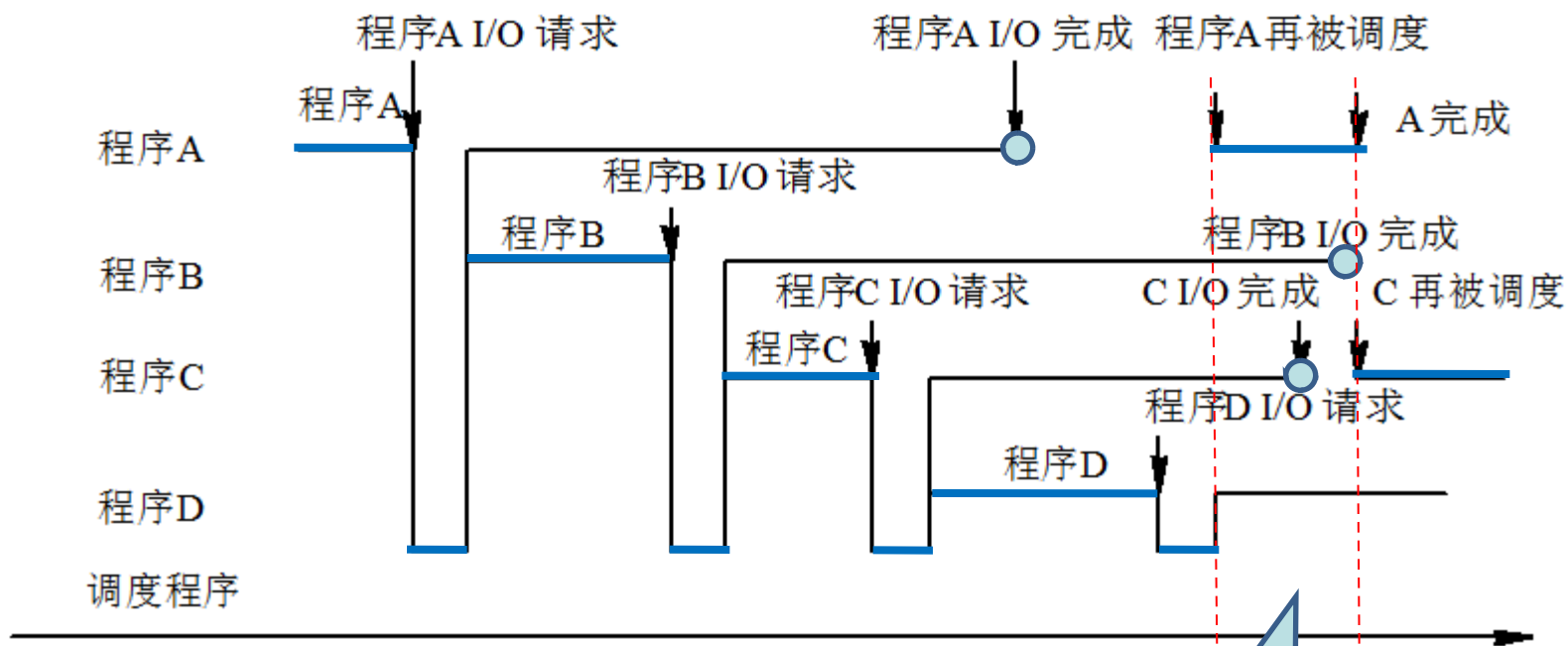
3. 进程调度方式

1) 非抢占方式(Nonpreemptive Mode)

在采用这种调度方式时，一旦把处理机分配给某进程后，就**一直让它运行下去**，决不会因为时钟中断或任何其它原因去抢占当前正在运行进程的处理机，直至该进程完成，或发生某事件而被阻塞时，才把处理机分配给其它进程。

引起进程调度的因素有：①正在执行的进程运行完毕，或因发生某事件而使其无法再继续运行；②正在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Block原语。

优点：实现简单，系统开销小，适用于大多数的批处理系统。但是不能用于分时系统和大多数实时系统。



(b) 四道程序运行情况

不会发生抢占





2) 抢占方式(Preemptive Mode)

这种调度方式允许调度程序根据某种原则，去暂停某个正在执行的进程，将已分配给该进程的处理机重新分配给另一进程。

在现代OS中广泛采用抢占方式，这是因为：对于批处理机系统，可以防止一个长进程长时间地占用处理机，以确保处理机能为所有进程提供更为公平的服务。

在分时系统中，只有采用抢占方式才有可能实现人—机交互。

在实时系统中，抢占方式能满足实时任务的需求。但抢占方式比较复杂，所需付出的系统开销也较大。



“抢占”的主要原则：

① 优先权原则：对一些重要的和紧急的作业赋予较高的优先权。允许优先权高的新到进程抢占当前进程的处理机。

② 短进程优先原则：短作业(进程)可以抢占当前较长作业(进程)的处理机。

③ 时间片原则：各进程按时间片轮流运行，当一个时间片用完后，便停止该进程的执行而重新进行调度。这种原则适用于分时系统、大多数的实时系统，以及要求较高的批处理系统。



3.3.2 轮转调度算法

1. 轮转法的基本原理

在轮转(RR)法中，系统将所有的就绪进程按FCFS策略排成一个就绪队列。

系统可设置每隔一定时间(如30 ms)便产生一次中断，去激活进程调度程序进行调度，把CPU分配给队首进程，并令其执行一个时间片。

当它运行完毕后，又把处理机分配给就绪队列中新的队首进程，也让它执行一个时间片。这样，就可以保证就绪队列中的所有进程在确定的时间段内，都能获得一个时间片的处理机时间。

2. 进程切换时机

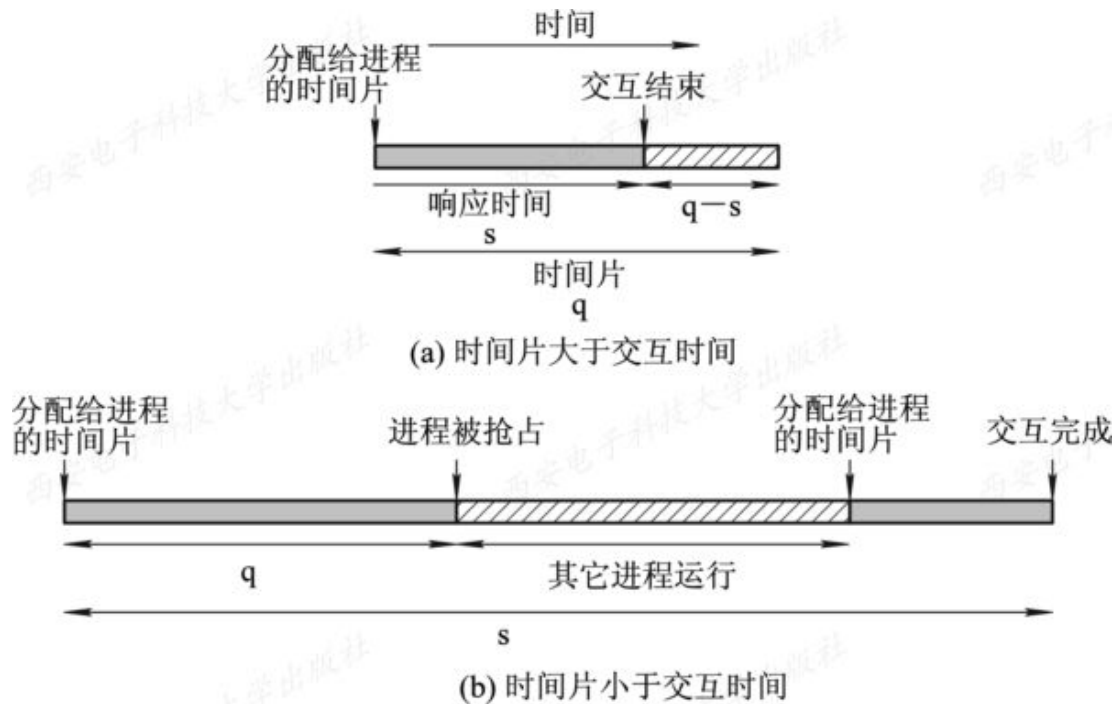
在RR调度算法中，应在何时进行进程的切换，可分为两种情况：

① 若一个时间片尚未用完，正在运行的进程便已经完成，就立即激活调度程序，将它从就绪队列中删除，再调度就绪队列中队首的进程运行，并启动一个新的时间片。

② 在一个时间片用完时，计时器中断处理程序被激活。如果进程尚未运行完毕，调度程序将把它送往就绪队列的末尾。

3. 时间片大小的确定

在轮转算法中，时间片的大小对系统性能有很大的影响。一个较为可取的时间片大小是略大于一次典型的交互所需要的时间。





时间片大小对响应时间的影响

<div> <div>作业情况</div> <div>时间片</div> </div>	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	4	2	4	
RR q=1	完成时间						
	周转时间						
	带权周转时间						
RR q=4	完成时间						
	周转时间						
	带权周转时间						

图3-3 $q = 1$ 和 $q = 4$ 时进程的周转时间

作业 情况 时 间 片	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	4	2	4	
RR $q=1$	完成时间	15	12	16	9	17	
	周转时间	15	11	14	6	13	11.8
	带权周转时间	3.75	3.67	3.5	3	3.25	3.43
RR $q=4$	完成时间	4	7	11	13	17	
	周转时间	4	6	9	10	13	8.4
	带权周转时间	1	2	2.25	5	3.25	2.7

图3-3 $q = 1$ 和 $q = 4$ 时进程的周转时间



3.3.3 优先级调度算法

1. 优先级调度算法的类型

优先级进程调度算法，是把处理机分配给就绪队列中优先级最高的进程。可进一步把该算法分成如下两种。

(1) **非抢占式优先级调度算法**：一旦把处理机分配给就绪队列中优先级最高的进程后，该进程便一直执行下去直至完成，或者因该进程发生某事件而放弃处理机时，系统方可将处理机重新分配给另一优先级更高的进程。

进程	到达时间	运行时间	优先数
P1	0	7	1
P2	2	4	2
P3	4	1	3
P4	5	4	2



注：以下括号内表示当前处于就绪队列的进程

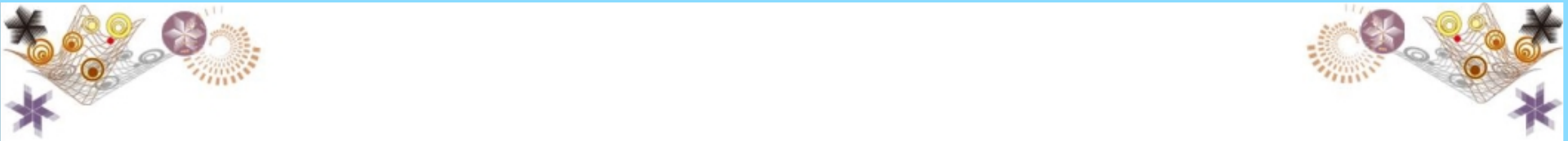
0时刻（P1）：只有P1到达，P1上处理机。

7时刻（P2、P3、P4）：P1运行完成主动放弃处理机，其余进程都已到达，P3优先级最高，P3上处理机。

8时刻（P2、P4）：P3完成，P2、P4优先级相同，由于P2先到达，因此P2优先上处理机

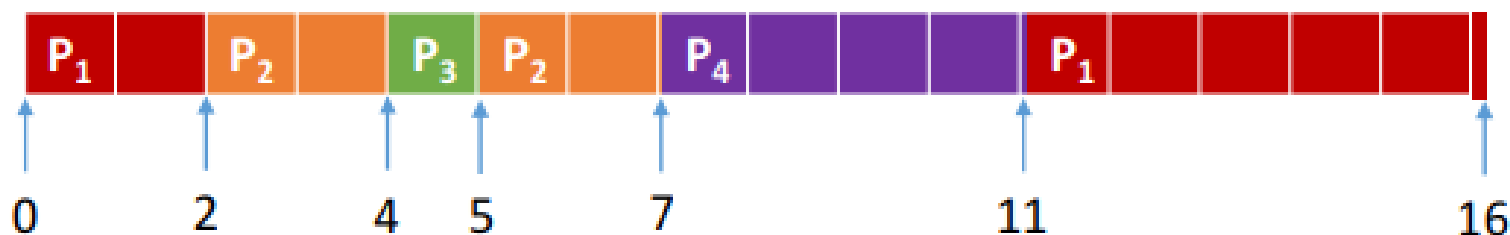
12时刻（P4）：P2完成，就绪队列只剩P4，P4上处理机。

16时刻（）：P4完成，所有进程都结束



(2) **抢占式优先级调度算法**：把处理机分配给优先级最高的进程，使之执行。但在其执行期间，只要出现了另一个其优先级更高的进程，调度程序就将处理机分配给新的优先级更高的进程。

进程	到达时间	运行时间	优先数
P1	0	7	1
P2	2	4	2
P3	4	1	3
P4	5	4	2



注：以下括号内表示当前处于就绪队列的进程

0时刻（P1）：只有P1到达，P1上处理机。

2时刻（P2）：P2到达就绪队列，优先级比P1更高，发生抢占。P1回到就绪队列，P2上处理机。

4时刻（P1、P3）：P3到达，优先级比P2更高，P2回到就绪队列，P3抢占处理机。

5时刻（P1、P2、P4）：P3完成，主动释放处理机，同时，P4也到达，由于P2比P4更先进入就绪队列，因此选择P2上处理机

7时刻（P1、P4）：P2完成，就绪队列只剩P1、P4，P4上处理机。

11时刻（P1）：P4完成，P1上处理机

16时刻（）：P1完成，所有进程均完成

2. 优先级的类型

1) 静态优先级

静态优先级是在创建进程时确定的，在进程的整个运行期间保持不变。优先级是利用某一范围内的一个整数来表示的，例如0~255中的某一整数，又把该整数称为优先数。确定进程优先级大小的依据有如下三个：

(1) 进程类型：通常，系统进程(如接收进程、对换进程、磁盘I/O进程)的优先权高于一般用户进程的优先权；前台进程优先级高于后台进程；I/O型进程优于计算型进程。

(2) 进程对资源的需求：如进程的估计执行时间及内存需要量的多少，对这些要求少的进程应赋予较高的优先权。

(3) 用户要求：由用户进程的紧迫程度及用户所付费用的多少来确定优先权的。

2) 动态优先级

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，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。



3.3.4 多队列调度算法

将系统中的进程就绪队列从一个拆分为若干个，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。

优点：对于不同用户进程的需求，很容易提供多种调度策略；便于应用于多处理机系统。

3.3.5 多级反馈队列 (multi leveled feedback queue) 调度算法

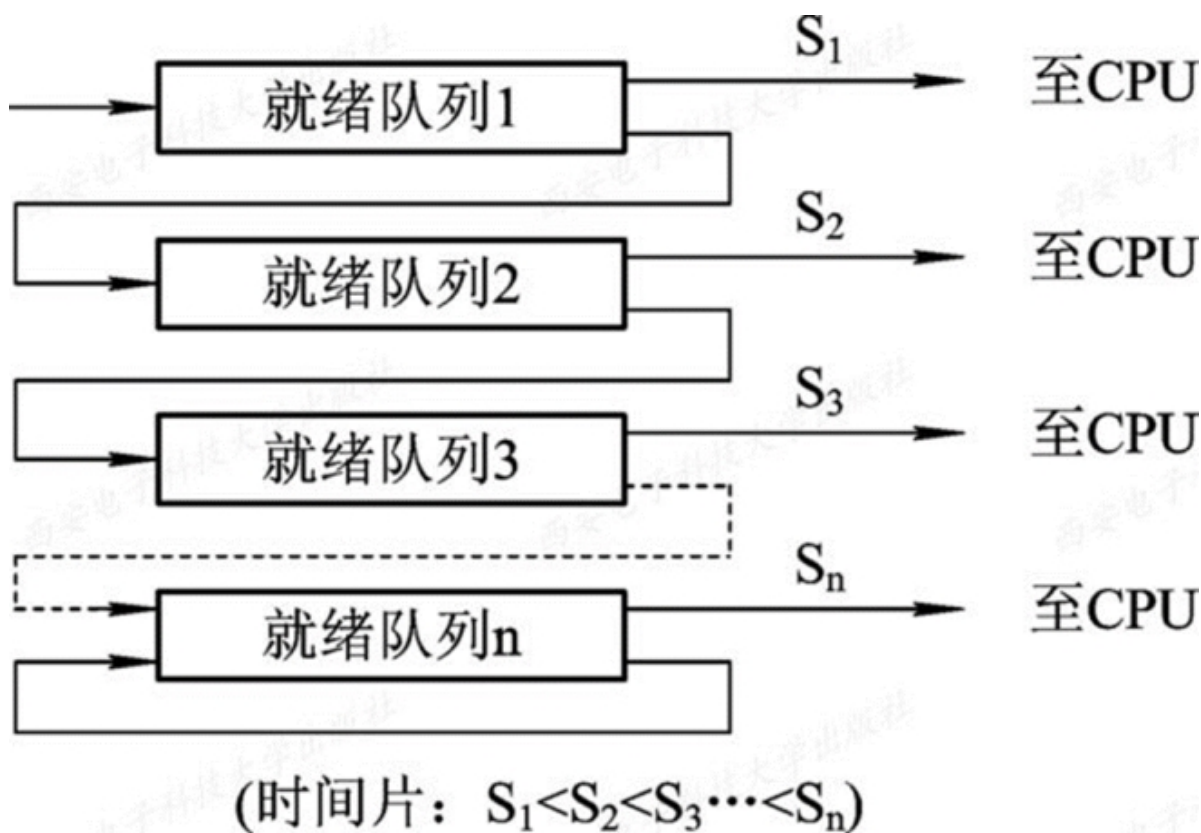
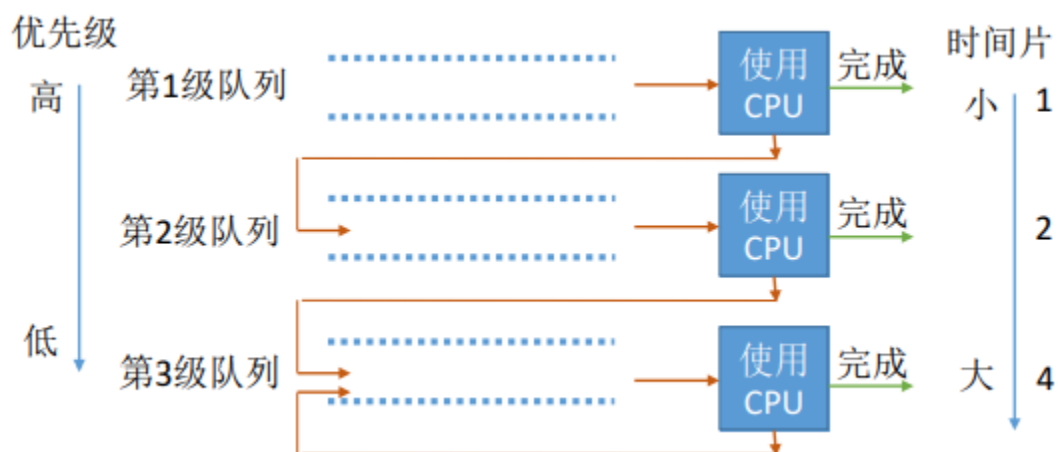


图3-4 多级反馈队列调度算法

进程	到达时间	运行时间
P1	0	8
P2	1	4
P3	5	1





P1(1) → P2(1) → P1(2)
 → P2(1) → P3(1) → P2(2)
 → P1(4) → P1(1)

1. 调度机制



多级反馈队列调度算法的调度机制可描述如下：

(1) 设置多个就绪队列。在系统中设置多个就绪队列，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。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。



(2) 每个队列都采用**FCFS**算法。当新进程进入内存后，**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，便可撤离系统。否则，即它在一个时间片结束时尚未完成，调度程序将其**转入第二队列的末尾等待调度**；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**，……，依此类推。当进程最后被降到第 n 队列后，在第 n 队列中便采取按RR方式运行。



(3) 按队列优先级调度。调度程序首先调度最高优先级队列中的诸进程运行，仅当第一队列空闲时才调度第二队列中的进程运行；换言之，仅当第 $1 \sim (i-1)$ 所有队列均空时，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任一优先级较高的队列，此时须立即把正在运行的进程放回到第 i 队列的末尾（抢占处理机），而把处理机分配给新到的高优先级进程。



2. 调度算法的性能

在多级反馈队列调度算法中，如果规定第一个队列的时间片略大于多数人机交互所需之处理时间时，便能较好地满足各种类型用户的需要。

(1) 终端型用户。

由于终端型作业用户所提交的作业大多属于交互型作业，作业通常较小，系统只要能使这些作业(进程)在第1队列所规定的时间片内完成，便可使终端型作业用户都感到满意。




(2) 短批处理作业用户。

开始时像终端型作业一样，如果仅在第一队列中执行一个时间片即可完成，便可获得与终端型作业一样的响应时间。对于稍长的作业，通常也只需在第二队列和第三队列各执行一个时间片即可完成，其周转时间仍然较短。

(3) 长批处理作业用户。

对于长作业，它将依次在第1, 2, ..., n个队列中运行，然后再按轮转方式运行，用户不必担心其作业长期得不到处理。



3.3.6 基于公平原则的调度算法

1. 保证调度算法

保证调度算法是另外一种类型的调度算法，它向用户所做出的保证并不是优先运行，而是明确的性能保证，该算法可以做到调度的公平性。

一种比较容易实现的性能保证是处理机分配的公平性。如果在系统中有 n 个相同类型的进程同时运行，为公平起见，须保证每个进程都获得相同的处理机时间 $1/n$ 。



在实施公平调度算法时系统中必须具有这样一些功能：

(1) **跟踪**计算每个进程自创建以来已经执行的**处理时间**。

(2) **计算**每个进程**应获得的处理机时间**，即自创建以来的时间除以 n 。

(3) **计算**进程获得处理机时间的**比率**，即进程实际执行的**处理时间**和**应获得的处理机时间**之比。

(4) **比较**各进程获得处理机时间的**比率**。如进程A的比率最低，为0.5，而进程B的比率为0.8，进程C的比率为1.2等。

(5) 调度程序**应选择比率最小的进程**将处理机分配给它，并让该进程一直运行，直到超过最接近它的进程比率为止。



2. 公平分享调度算法

不同于给每个进程相同的处理机时间，本调度算法中，调度的公平性主要是针对用户而言，使**所有用户**能获得相同的处理机时间。

	先来先服务	短作业优先	高响应比优先	时间片轮转	优先级调度算法	多级反馈队列
是否是抢占方式	非抢占方式	非抢占方式	非抢占方式	抢占方式	有抢占方式的 也有非抢占方式的	抢占方式
优点	公平，实现简单	平均等待时间最少，效率最高	兼顾长短作业	兼顾长短作业，公平	区分优先级	兼顾长短作业，有较好的响应时间，可行性强
缺点	不利于短作业	对长作业不利，可能会导致饥饿现象	计算响应比的开销比较大	平均等待时间较长，上下文切换浪费时间	可能导致饥饿	一般没有缺点
适用系统	多道批处理系统	多道批处理系统	多道批处理系统	分时系统	实时系统	通用
是否会导致饥饿	不会	会	不会	不会	会	会



3.4 实时调度

在实时系统中，可能存在着两类不同性质的实时任务，即硬实时任务（HRT）和软实时任务（SRT），它们都联系着一个截止时间。为保证系统能正常工作，实时调度必须能满足实时任务对截止时间的要求。为此，实现实时调度应具备一定的条件。



3.4.1 实现实时调度的基本条件

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信息：

- (1) **就绪时间**，是指某任务成为就绪状态的起始时间，在周期任务的情况下，它是事先预知的一串时间序列。
- (2) **开始截止时间和完成截止时间**，对于典型的实时应用，只须知道开始截止时间，或者完成截止时间。



(3) **处理时间**，一个任务从开始执行，直至完成时所需的时间。

(4) **资源要求**，任务执行时所需的一组资源。



(5) **优先级**，如果某任务的开始截止时间错过，势必引起故障，则应为该任务赋予“**绝对**”**优先级**；如果其开始截止时间的错过，对任务的继续运行无重大影响，则可为其赋予“**相对**”**优先级**，供调度程序参考。



2. 系统处理能力强

假定系统中有 m 个周期性的硬实时任务HRT，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系统才是可调度的：

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



提高系统处理能力的途径有二：一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；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

在含有HRT任务的实时系统中，广泛采用抢占机制。这样便可满足HRT任务对截止时间的要求。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，以简化调度程序和任务调度时所花费的系统开销。

4. 具有快速切换机制

为保证硬实时任务能及时运行，在系统中还应具有快速切换机制，使之能进行任务的快速切换。该机制应具有如下两方面的能力：

(1) **对中断的快速响应能力**。对紧迫的外部事件请求中断能及时响应，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**。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

可以按不同方式对实时调度算法加以分类：

- ① **根据**实时任务性质，可将实时调度的算法分为硬实时调度算法和软实时调度算法；
- ② **按**调度方式，则可分为非抢占调度算法和抢占调度算法。

1. 非抢占式调度算法

(1) 非抢占式轮转调度算法

常用于工业生产的群控系统中，由一台计算机控制若干个相同的(或类似的)对象，为每一个被控对象建立一个实时任务，并将它们排成一个轮转队列。



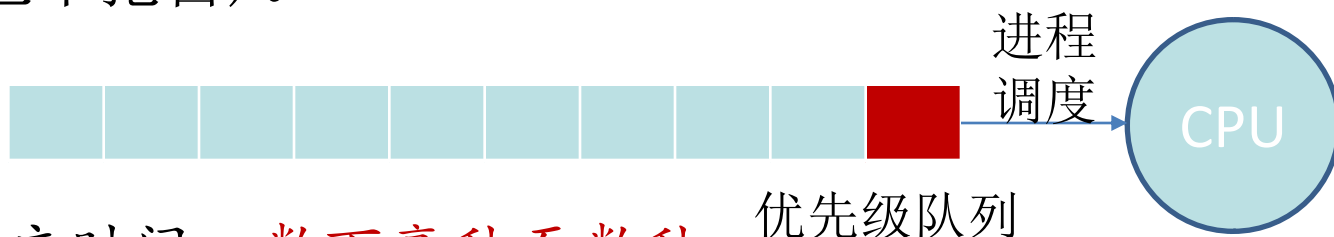
轮转队列

响应时间：数秒至数十秒

可用于要求不太严格的实时控制系统中

2) 非抢占式优先调度算法

为要求较为严格(响应时间为数百毫秒)的任务赋予较高的**优先级**。当这些实时任务到达时，把它们**安排**在就绪队列的**队首**，等待当前任务自我终止或运行完成后才能被调度执行(即使该任务的优先级高于当前任务的优先级也不抢占)。



响应时间：数百毫秒至数秒

可用于有一定要求的实时控制系统

要求较严格的实时系统

响应时间为数十毫秒以下

2. 抢占式调度算法

根据抢占发生时间的不同而进一步分成：

1) 基于时钟中断的抢占式优先权调度算法

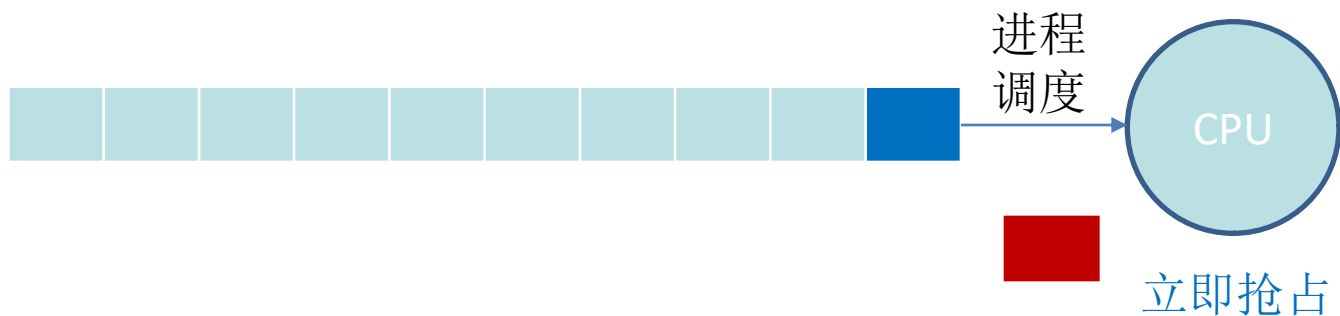
在某实时任务到达后，如果该任务的优先级高于当前任务的优先级，这时并不立即抢占当前任务的处理机，而是等到时钟中断到来时，调度程序才剥夺当前任务的执行，将处理机分配给新到的高优先权任务（发生抢占）。



调度延迟可降为几毫秒至几十毫秒。
可用于大多数的实时系统中。

2) 立即抢占(Immediate Preemption)的优先权调度算法

在这种调度策略中，要求操作系统具有快速响应外部事件中断的能力。一旦出现外部中断，只要当前任务未处于临界区，便立即剥夺当前任务的执行，把处理机分配给请求中断的紧迫任务。



非常快的响应，调度延迟降低到100微秒至几毫秒，甚至更低。



(a) 非抢占轮转调度



(c) 基于时钟中断抢占的优先级抢占调度



(b) 非抢占优先级调度



(d) 立即抢占的优先级调度

图3-5 实时进程调度

3.4.3 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。截止时间愈早，其优先级愈高。



排队器：
按各任务截止时
间的早晚排队

实时任务就绪队列

既可用于抢占式调度，也可用于非抢占式调度方式中。

1. 非抢占式调度方式用于非周期实时任务

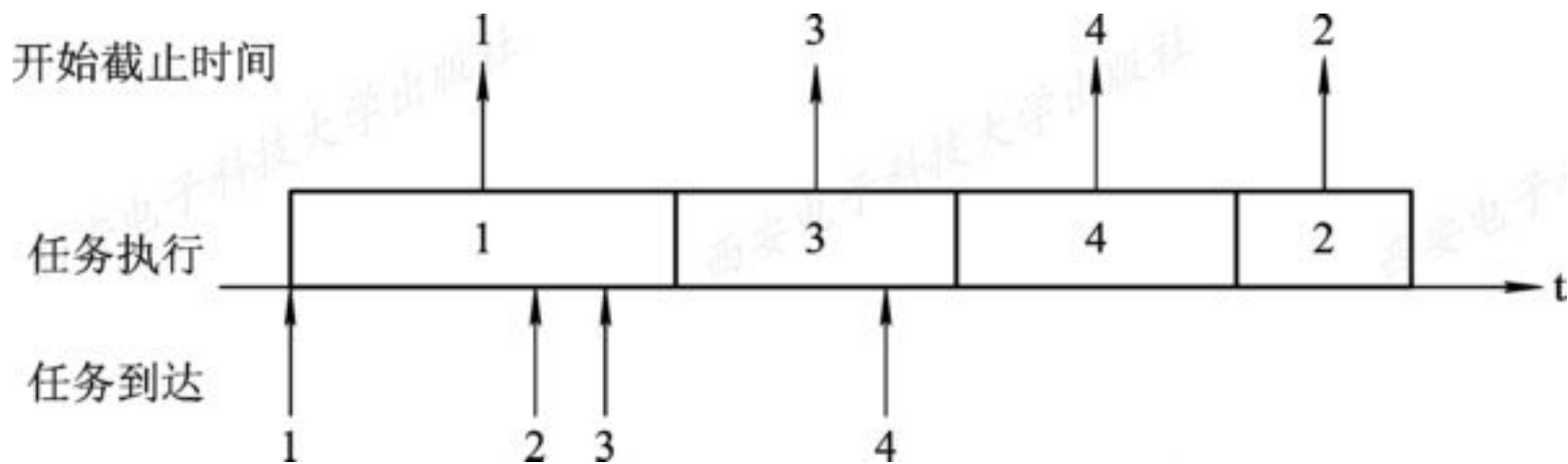


图3-6 EDF算法用于非抢占调度方式

2. 抢占式调度方式用于周期实时任务

图3-7示出了将该算法用于抢占调度方式之例。在该例中有两个周期任务，任务A和任务B的周期时间分别为20 ms和50 ms，每个周期的处理时间分别为10 ms和25 ms。

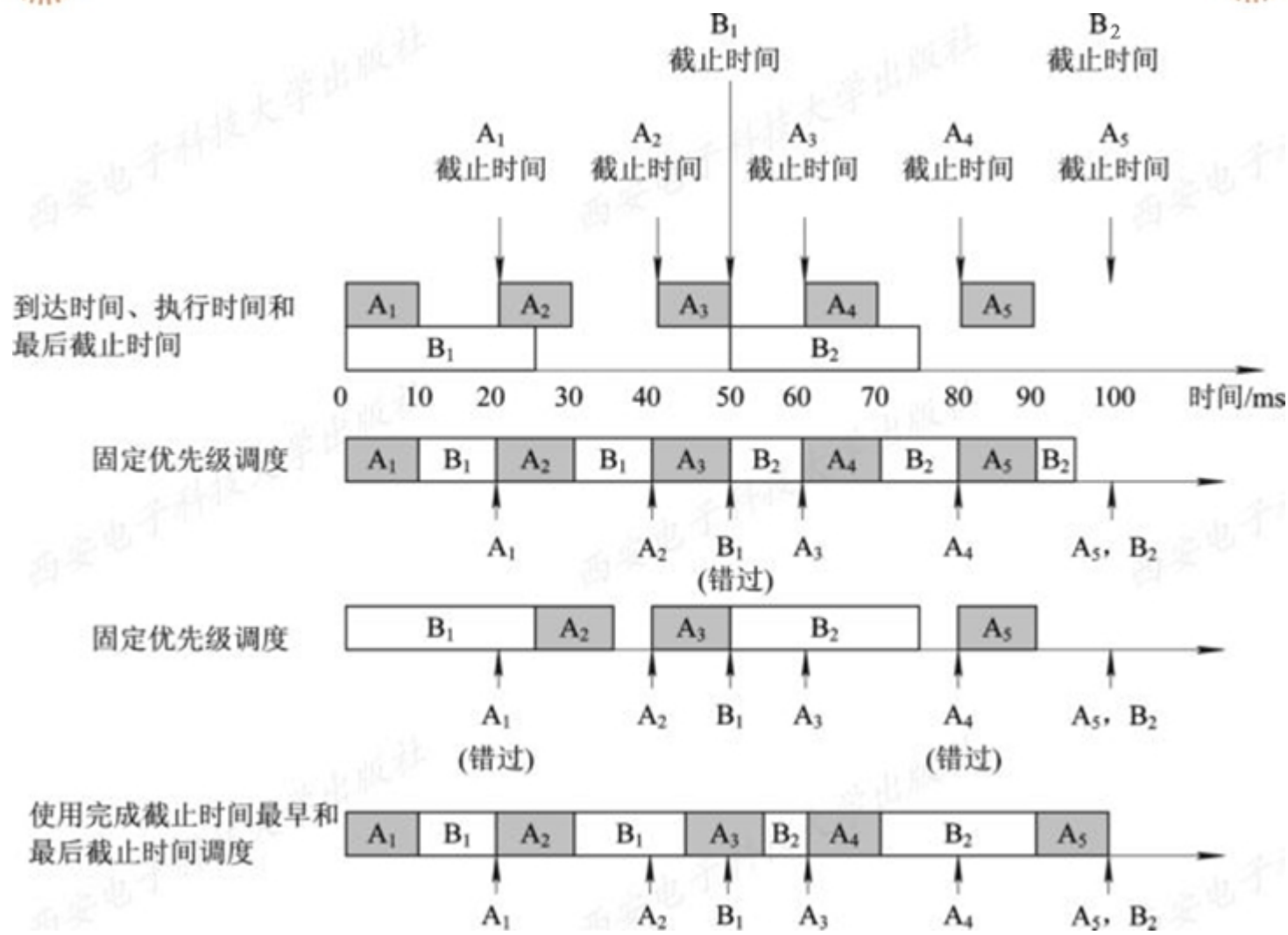


图3-7 最早截止时间优先算法用于抢占调度方式之例

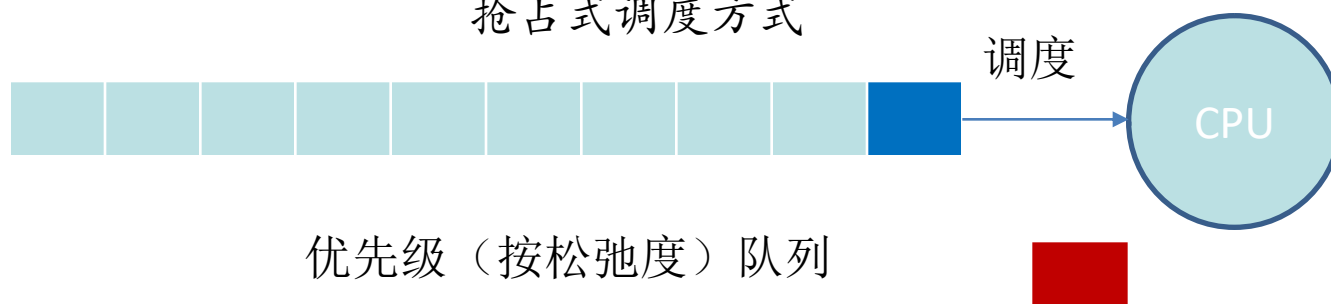
3.4.4 最低松弛度优先LLF (Least Laxity First) 算法

该算法根据的是任务的紧急(或松弛)程度。任务紧急程度愈高，赋予该任务的优先级就愈高，以使之优先执行。

松弛度的计算 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100ms之前调度执行，该任务的紧急程度(松弛程度)为100ms。

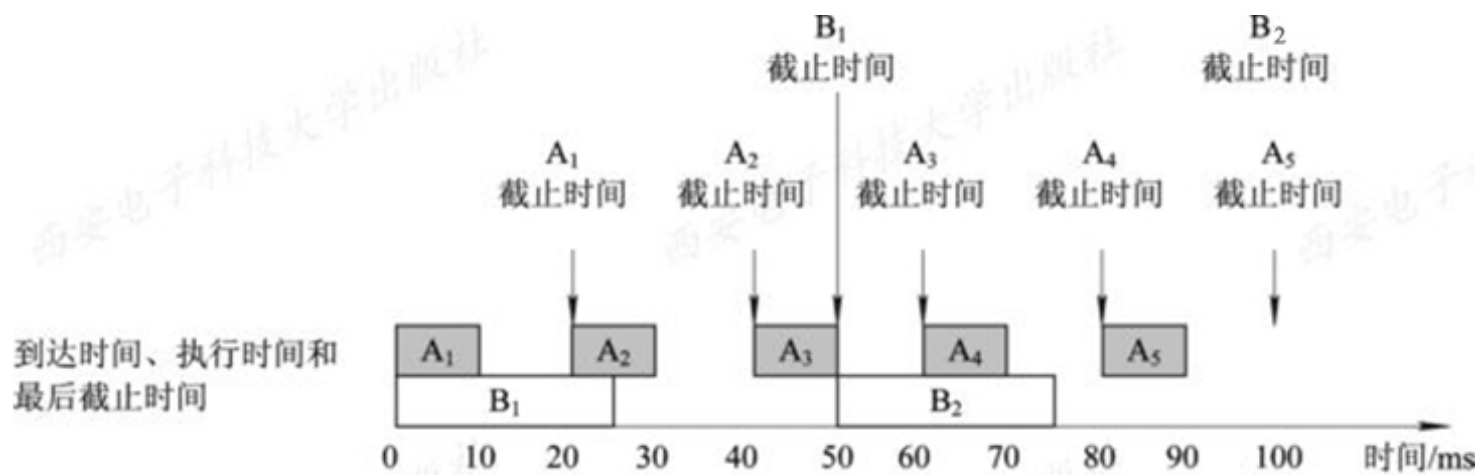
松弛度=必须完成的时间-其本身的运行时间-当前时间

抢占式调度方式



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要求每50 ms执行一次，执行时间为25 ms。由此可知，任务A和B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 、...。



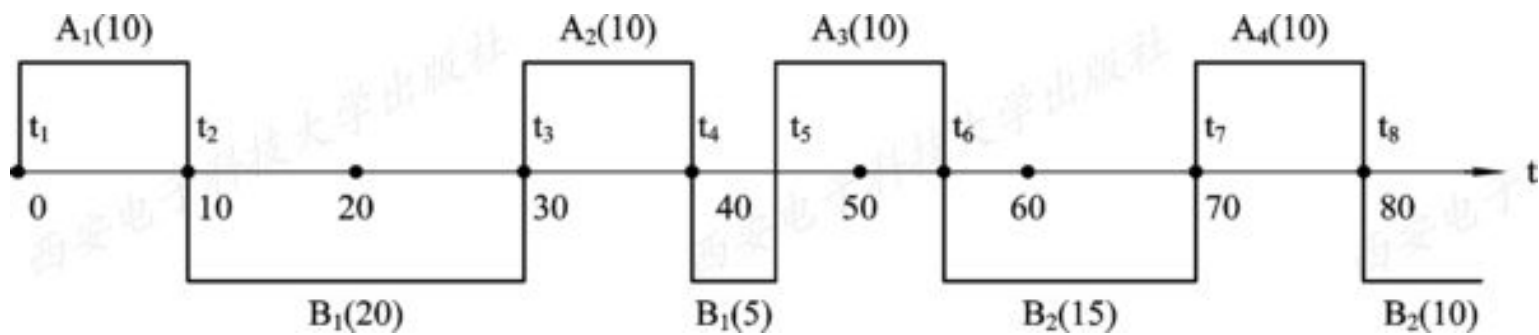


图3-9 利用ELLF算法进行调度的情况



3.4.5 优先级倒置(priority inversion problem)

1. 优先级倒置的形成

当前OS广泛采用优先级调度算法和抢占方式，然而在系统中存在着影响进程运行的资源而可能产生“优先级倒置”的现象，即高优先级进程(或线程)被低优先级进程(或线程)延迟或阻塞。

优先级倒置举例

假如有三个完全独立的进程 P_1 、 P_2 和 P_3 。 P_1 的优先级 $> P_2$ 的优先级 $> P_3$ 的优先级。 P_1 和 P_3 通过共享的一个临界资源进行交互。下面是一段代码：

P_1 : ...P(mutex); CS-1; V(mutex); ...

P_2 : ...Program2...;

P_3 : ...P(mutex); CS-3; V(mutex); ...

假如 P_3 最先执行，在执行了P(mutex)操作后，进入到临界区CS-3。在时刻a， P_2 就绪，因为它比 P_3 的优先级高， P_2 抢占了 P_3 的处理机而运行。

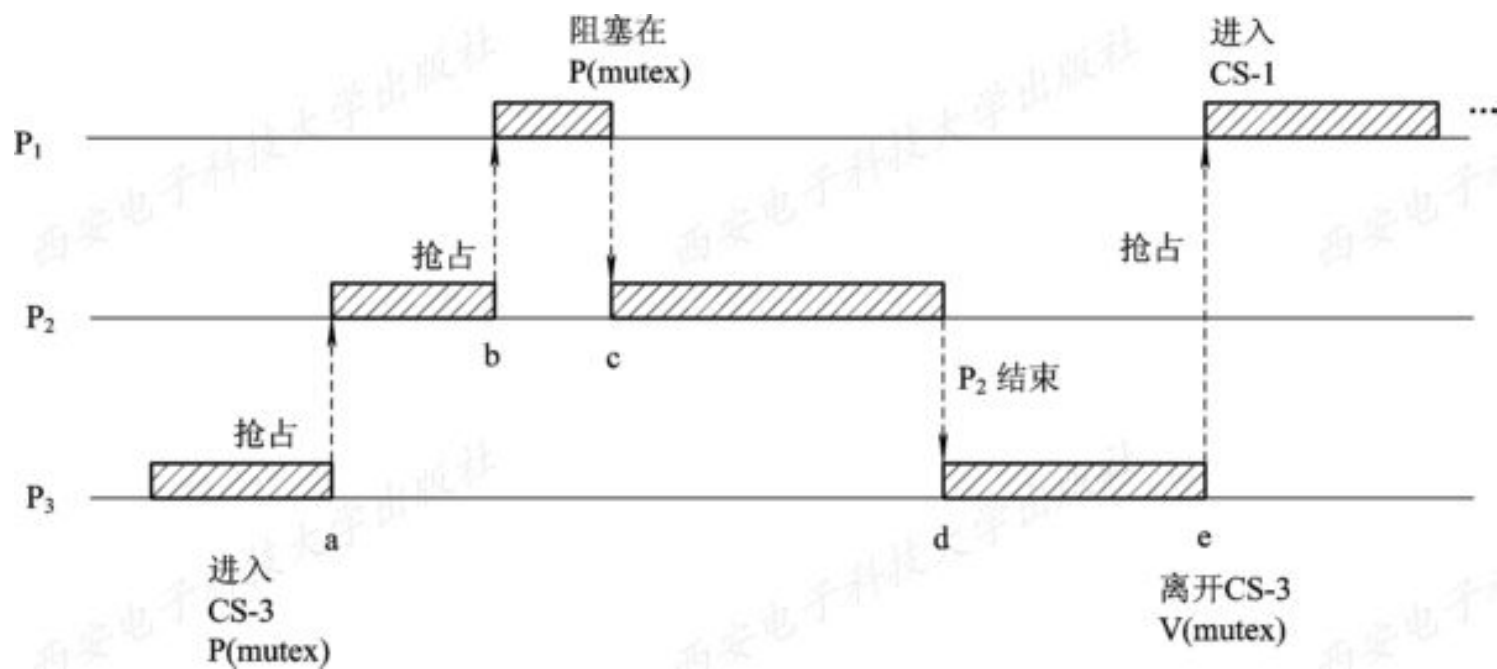


图3-10 优先级倒置示意图

2. 优先级倒置的解决方法

一种简单的解决方法是规定：假如进程 P_3 在进入临界区后 P_3 所占用的处理机就不允许被抢占。

一种比较使用的方法是建立在动态优先级继承的基础上。该方法规定，当高优先级进程 P_1 要进入临界区，去使用临界资源 R ，如果已有一个低优先级进程 P_3 正在使用该资源，此时一方面 P_1 被阻塞，另一方面 P_3 继承 P_1 的优先级，并一直保持到 P_3 退出临界区。

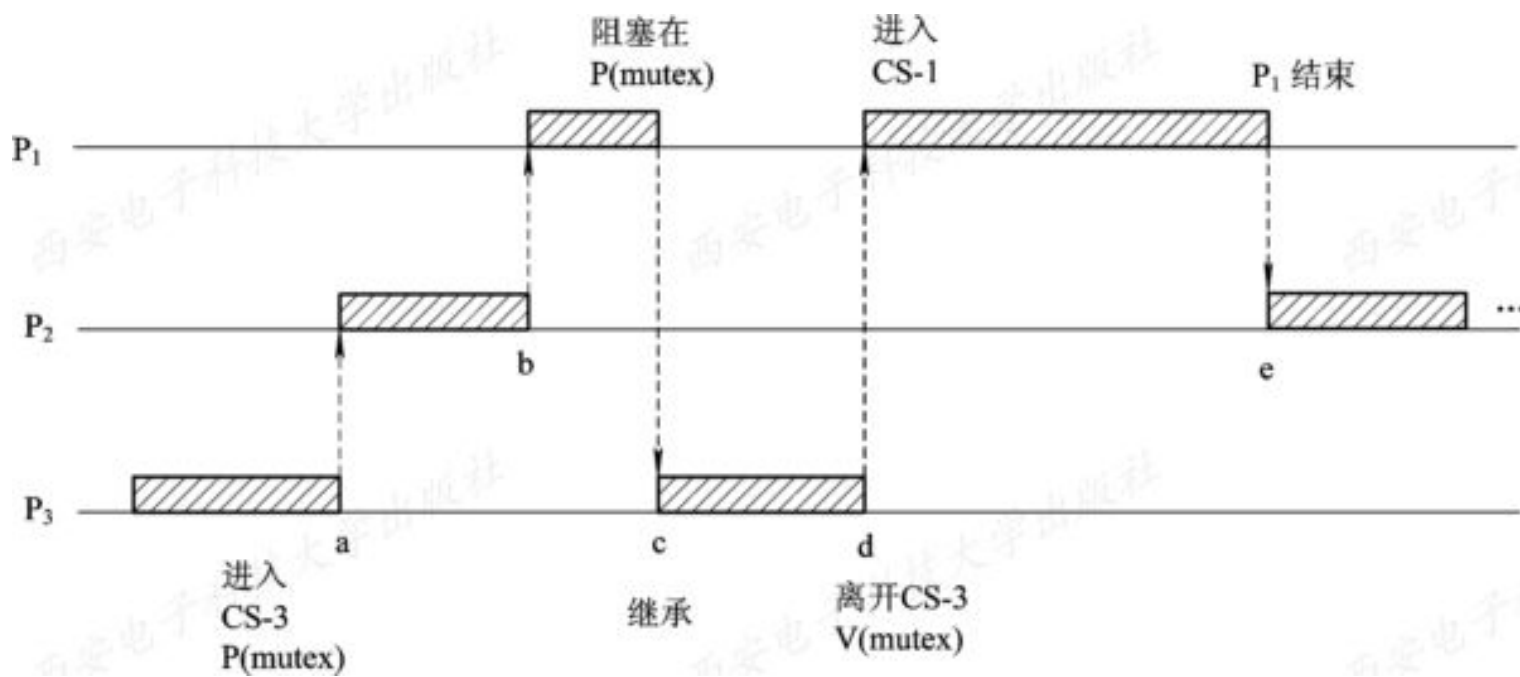


图3-11 采用了动态优先级继承方法的运行情况



3.5 死 锁 概 述

3.5.1 资源问题

在系统中有许多不同类型的资源，其中可以引起死锁的主要是：**需要采用互斥访问方法的、不可以被抢占的资源**，即在前面介绍的**临界资源**。系统中这类资源有很多，如打印机、数据文件、队列、信号量等。

1. 可重用性资源和消耗性资源

1) 可重用性资源

可重用性资源是一种可供用户重复使用多次的资源，它具有如下性质：

(1) 每一个可重用性资源中的单元只能分配给一个进程使用，不允许多个进程共享。

(2) 进程在使用可重用性资源时，须按照这样的顺序：

① 请求资源。如果请求资源失败，请求进程将会被阻塞或循环等待。② 使用资源。进程对资源进行操作，如用打印机进行打印；③ 释放资源。当进程使用完后自己释放资源。

(3) 系统中每一类可重用性资源中的单元数目是相对固定的，进程在运行期间既不能创建也不能删除它。

计算机中的大部分资源属于可重用性资源



2) 可消耗性资源

可消耗性资源又称为临时性资源，它是在进程运行期间，由进程动态地创建和消耗的，它具有如下性质：

① **每一类**可消耗性资源的单元数目在进程运行期间是可以不断变化的，有时它可以有许多，有时可能为0；

② 进程在运行过程中，可以不断地创造可消耗性资源的单元，将它们放入该资源类的缓冲区中，以增加该资源类的单元数目。

③ 进程在运行过程中，可以请求若干个可消耗性资源单元，用于进程自己的消耗，不再将它们返回给该资源类中。

最典型的可消耗性资源就是用于进程间通信的消息等。



2. 可抢占性资源和不可抢占性资源

1) 可抢占性资源

可把系统中的资源分成两类，一类是可抢占性资源，是指某进程在获得这类资源后，该资源可以再被其它进程或系统抢占。**CPU和主存均属于可抢占性资源，对于这类资源是不会引起死锁的。**

2) 不可抢占性资源

另一类资源是不可抢占性资源，即一旦系统把某资源分配给该进程后，就不能将它强行收回，只能在进程用完后自行释放。**如磁带机、打印机等。**

3.5.2 计算机系统中的死锁

1. 竞争不可抢占性资源引起死锁

通常系统中所拥有的不可抢占性资源其数量不足以满足多个进程运行的需要，使得进程在运行过程中，会因争夺资源而陷入僵局。

用方块代表可重用的资源(文件), 用圆圈代表进程。

P_1 和 P_2 都准备写两个文件 F_1 和 F_2 。

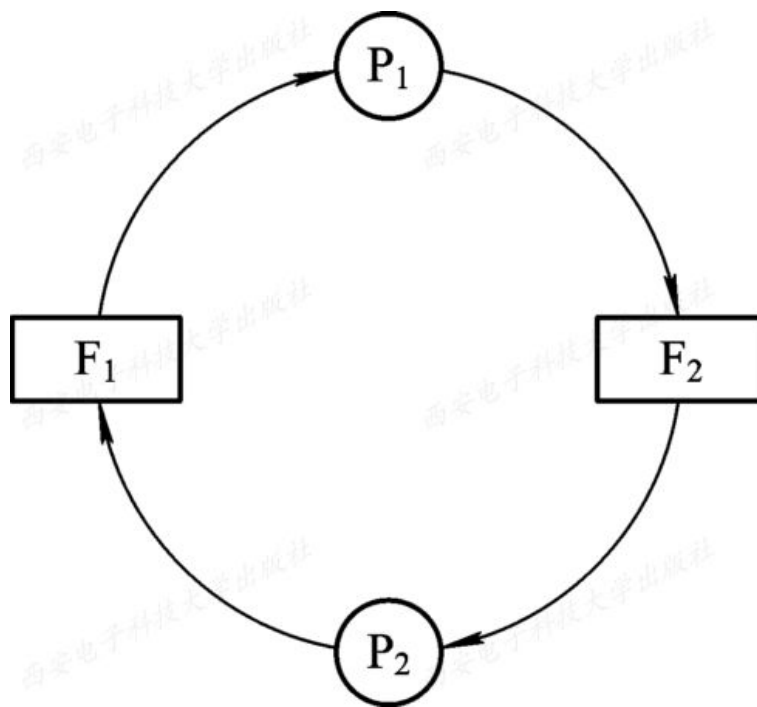


图3-12 共享文件时的死锁情况

2. 竞争可消耗资源引起死锁

例如： m_1 、 m_2 和 m_3 是临时性资源。如果消息通信按下述顺序进行：

P_1 : ...send(p_2, m_1); receive(p_3, m_3); ...

P_2 : ...send(p_3, m_2); receive(p_1, m_1); ...

P_3 : ...send(p_1, m_3); receive(p_2, m_2); ...

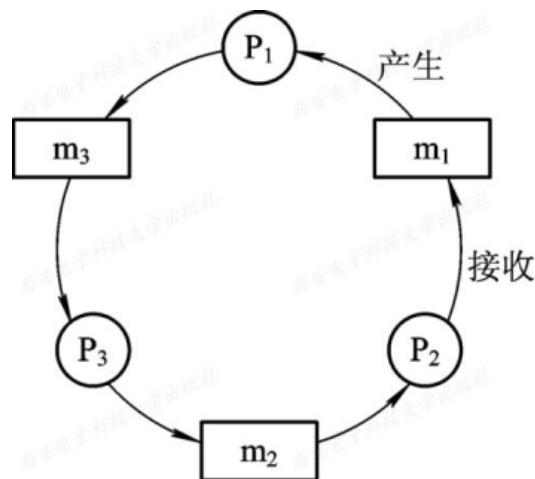
不会产生死锁

如果消息通信按下述顺序进行：

P_1 : ... receive(p_3 , m_3); send(p_2 , m_1); ...

P_2 : ... receive(p_1 , m_1); send(p_3 , m_2); ...

P_3 : ... receive(p_2 , m_2); send(p_1 , m_3); ...



进程之间通信产生死锁

3. 进程推进顺序不当引起死锁

除了系统中多个进程对资源的竞争会引发死锁外，进程在运行过程中，对资源进行申请和释放的顺序是否合法，也是在系统中是否会产生死锁的一个重要因素。

1) 进程推进顺序合法

在进程 P_1 和 P_2 并发执行时，如果按图3-14中的曲线①所示的顺序推进： P_1 : Request(R_1)→ P_1 : Request(R_2)→ P_1 : Release(R_1)→ P_1 : Release(R_2)→ P_2 : Request(R_2)→ P_2 : Request(R_1)→ P_2 : Release(R_2)→ P_2 : Release(R_1), 两个进程可顺利完成。类似地，若按图中曲线②和③所示的顺序推进，两进程也可以顺利完成。我们称这种不会引起进程死锁的推进顺序是合法的。

2) 进程推进顺序非法

若并发进程 P_1 和 P_2 按图3-14中曲线④所示的顺序推进，它们将进入不安全区D内。此时 P_1 保持了资源 R_1 ， P_2 保持了资源 R_2 ，系统处于不安全状态。此刻，如果两个进程继续向前推进，就可能发生死锁。例如，当 P_1 运行到 P_1 : Request(R_2)时，将因 R_2 已被 P_2 占用而阻塞；当 P_2 运行到 P_2 : Request(R_1)时，也将因 R_1 已被 P_1 占用而阻塞，于是发生了进程死锁，这样的进程推进顺序就是非法的。

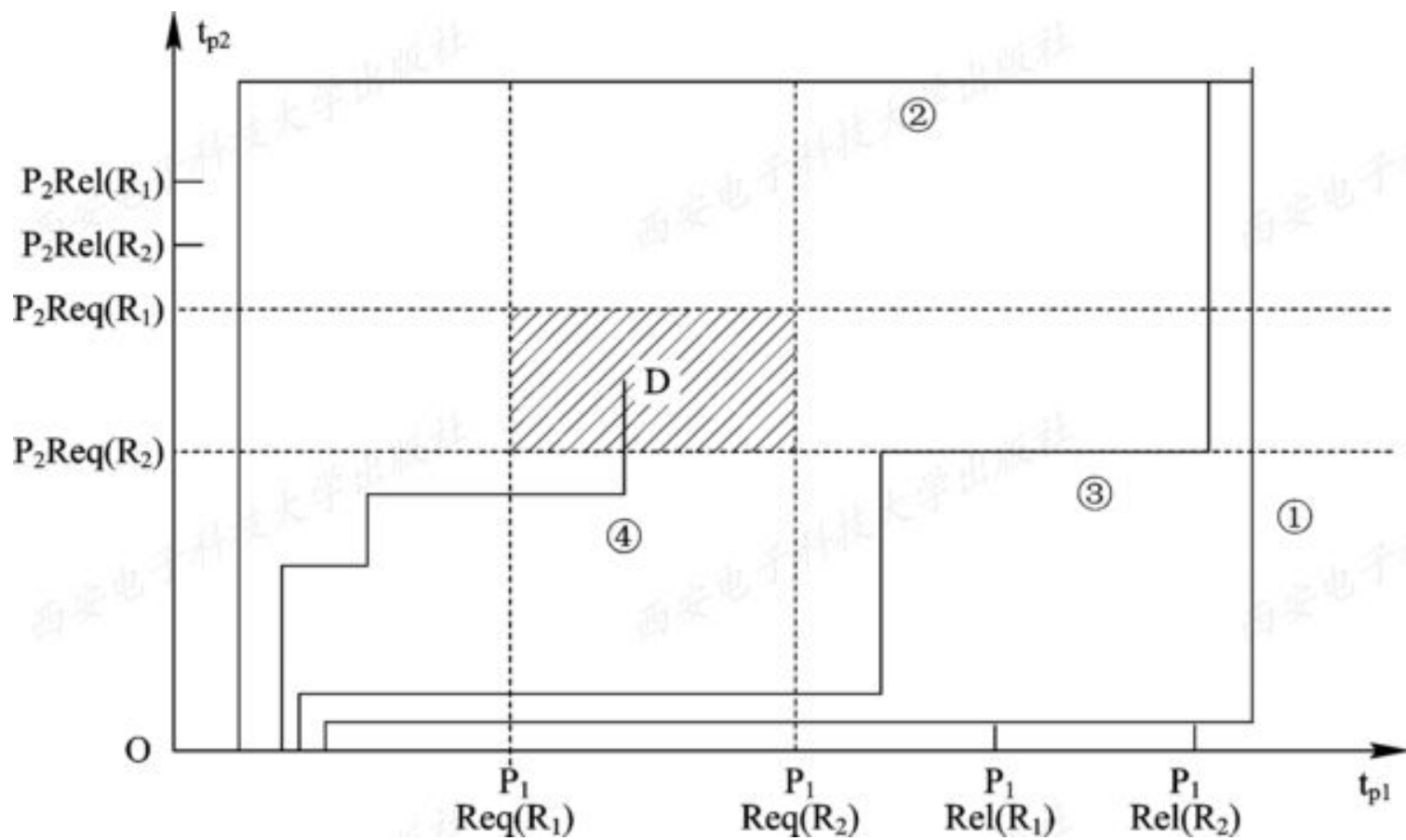




图3-14 进程推进顺序对死锁的影响



3.5.3 死锁的定义、必要条件和处理方法

1. 死锁的定义



在一组进程发生死锁的情况下，这组死锁进程中的每一个进程，都在等待另一个死锁进程所占有的资源。



2. 产生死锁的必要条件

(1) **互斥条件**：指**进程对所分配到的资源进行排它性使用**，即在一段时间内某资源只由一个进程占用。如果此时还有其它进程请求该资源，则请求者只能等待，直至占有该资源的进程用毕释放。

(2) **请求和保持条件（部分分配）**：指**进程已经保持了至少一个资源，但又提出了新的资源请求，而该资源又已被其它进程占有，此时请求进程阻塞，但又对自己已获得的其它资源保持不放。**



(3) **不可抢占条件**：指进程已获得的资源，在未使用完之前，**不能被其他进程强行剥夺**，只能该进程在使用完时由自己释放。

(4) **循环等待条件**：指在发生死锁时，**必然存在一个进程——资源的环形链**，即进程集合 $\{P_0, P_1, P_2, \dots, P_n\}$ 中的 P_0 正在等待一个 P_1 占用的资源； P_1 正在等待 P_2 占用的资源，……， P_n 正在等待已被 P_0 占用的资源。

3. 处理死锁的方法

目前处理死锁的方法可归结为四种：

(1) 预防死锁：通过设置某些限制条件，去破坏产生死锁的四个必要条件中的一个或几个条件（除第一个条件外），来预防发生死锁。（静态策略）

(2) 避免死锁：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，从而避免发生死锁。（动态策略）


(3) 检测死锁：允许系统在运行过程中发生死锁。但可通过检测机构及时地检测出死锁的发生，然后采取适当措施，从进程从死锁中解脱出来。

(4) 解除死锁：这是与检测死锁相配套的一种措施。当检测到系统中已发生死锁时，须将进程从死锁状态中解脱出来。



3.6 预 防 死 锁

预防死锁的方法是通过破坏产生死锁的四个必要条件中的一个或几个，以避免发生死锁。由于互斥条件是非共享设备所必须的，不仅不能改变，还应加以保证，因此主要是破坏产生死锁的后三个条件。



3.6.1 破坏“请求和保持”条件



为了能破坏“请求和保持”条件，系统必须保证做到：当一个进程在请求资源时，它**不能持有不可抢占资源**。该保证可通过两个不同的协议实现。

1. 第一种协议

该协议规定，所有进程在开始运行之前，必须一次性地申请其在整个运行过程中所需的全部资源。

这种预防死锁的方法的优点是简单、易于实现且很安全。
缺点是：

- (1) **资源被严重浪费**，严重地恶化了资源的利用率。
- (2) **使进程经常会发生饥饿现象**。



2. 第二种协议

该协议是对第一种协议的改进，它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给自己的、且已用毕的全部资源，然后再请求新的所需资源。

优点：能使进程更快地完成任务，提高设备的利用率，还可以减少进程发生饥饿的机率。

3.6.2 破坏“不可抢占”条件



规定：这意味着进程已占有的资源会被暂时地释放。当一个已经保持了某些不可被抢占资源的进程，提出新的资源请求而不能得到满足时，它必须释放已经保持的所有资源，待以后需要时再重新申请。或者说是被抢占了，从而破坏了“不可抢占”条件。

缺点：实现复杂；可能会造成前一阶段工作的失效；反复地申请和释放资源使进程的执行被无限地延迟。

3.6.3 破坏“循环等待”条件

一个能保证“循环等待”条件不成立的方法是，对系统所有资源类型进行线性排序，并赋予不同的序号。

例如，令输入机的序号为1，打印机的序号为2，磁带机为3，磁盘为4。所有进程对资源的请求必须严格按照资源序号递增的次序提出，这样，在所形成的资源分配图中，不可能再出现环路，因而破坏了“循环等待”条件。



资源利用率和系统吞吐量都有较明显的改善。但存在下述严重问题：

(1) 为系统中各类资源所分配(确定)的序号必须相对稳定，这就限制了新类型设备的增加。

(2) 尽管在为资源的类型分配序号时，已经考虑到大多数作业在实际使用这些资源时的顺序，但也经常会发生这种情况：即作业(进程)使用各类资源的顺序与系统规定的顺序不同，造成对资源的浪费。

(3) 为方便用户，系统对用户编程时所施加的限制条件应尽量少。然而这种按规定次序申请的方法，必然会限制用户简单、自主地编程。

3.7 避免死锁



避免死锁同样是属于事先预防的策略，但并不是事先采取某种限制措施，破坏产生死锁的必要条件，而是在资源动态分配过程中，防止系统进入不安全状态，以避免发生死锁。这种方法所施加的限制条件较弱，可能获得较好的系统性能，目前常用此方法来避免发生死锁。

3.7.1 系统安全状态

在死锁避免方法中，把系统的状态分为安全状态和不安全状态。当系统处于安全状态时，可避免发生死锁。反之，当系统处于不安全状态时，则可能进入到死锁状态。

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。



安全状态：是指系统能按某种进程顺序(P_1, P_2, \dots, P_n), 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。此时称(P_1, P_2, \dots, P_n)序列为**安全序列**。

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**。

虽然并非所有的不安全状态都必然会转为死锁状态, 但当系统进入不安全状态后, 便有可能进入死锁状态; 反之, 只要系统处于安全状态, 系统便可避免进入死锁状态。

2. 安全状态之例

假定系统中三个进程 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	

存在安全序列 $\langle P_2, P_1, P_3 \rangle$



避免死锁的实质：

系统在进行资源分配时，

如何使系统不进入不安全状态。

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。

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

不存在安全序列

3.7.2 利用银行家算法避免死锁

最有代表性的避免死锁的算法是Dijkstra的银行家算法。起这样的名字是由于该算法原本是为银行系统设计的，以确保银行在发放现金贷款时，不会发生不能满足所有客户需要的情况。在OS中也可用它来实现避免死锁。

核心思想：每一个新进程在进入系统时，它必须**申明**在运行过程中可能需要每种资源类型的**最大单元数目**。当新进程请求一组资源时，系统必须首先确定是否有足够的资源分配给该进程。若有，再进一步计算在将这些资源分配给进程后，**是否会使系统处于不安全状态**。如果不会，才将资源分配给它。否则让进程等待。

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。

资源 情况 进 程	Max			Allocation			Need			Available		
	A	B	C	A	B	C	A	B	C	A	B	C
P_0	7	5	3	0	1	0	7	4	3	3	3	2
P_1	3	2	2	2	0	0	1	2	2			
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：利用安全性算法对 T_0 时刻的资源分配情况进行分析(如图3-16所示)可知，在 T_0 时刻存在着一个安全序列 $\{P_1, P_3, P_4, P_2, P_0\}$ ，故系统是安全的。

资源 情况 进 程	Max			Need			Allocation			Work+Allocation			Finish
	A	B	C	A	B	C	A	B	C	A	B	C	
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:

资源 情况 进 程	Max			Allocation			Need			Available		
	A	B	C	A	B	C	A	B	C	A	B	C
P_0	7	5	3	0	1	0	7	4	3	2	3	0
P_1	3	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			



④ 再利用安全性算法检查此时系统是否安全，如图3-17所示。

图3-17 P_1 申请资源时的安全性检查

找到安全序列 $\{P_1, P_3, P_4, P_0, P_2\}$ 。因此，系统是安全的，可以立即将 P_1 所申请的资源分配给它。

(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 $Request_4(3, 3, 0) > Available(2, 3, 0)$, 让 P_4 等待。

资源 情况 进 程	Max			Allocation			Need			Available		
	A	B	C	A	B	C	A	B	C	A	B	C
P_0	7	5	3	0	1	0	7	4	3	2	3	0
P_1	3	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			

(4) P_0 请求资源: P_0 发出请求向量 $Request_0(0, 2, 0)$, 系统按银行家算法进行检查:

- ① $Request_0(0, 2, 0) \leq Need_0(7, 4, 3)$;
- 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:

资源 情况 进 程	Max			Allocation			Need			Available		
	A	B	C	A	B	C	A	B	C	A	B	C
P_0	7	5	3	0	3	0	7	2	3	2	1	0
P_1	3	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			



(5) 进行安全性检查：可用资源Available(2, 1, 0)已不能满足任何进程的需要，故系统进入不安全状态，此时系统不分配资源。

1. 银行家算法中的数据结构

为了实现银行家算法，在系统中必须设置这样四个数据结构，分别用来描述系统中可利用的资源、所有进程对资源的最大需求、系统中的资源分配，以及所有进程还需要多少资源的情况。（设有 n 个进程， m 类资源）

(1) 可利用资源向量Available: m 个元素的一维数组，代表每一类可利用的资源数目。

(2) 最大需求矩阵Max: $n \times m$ 的矩阵，定义了系统中每个进程对每类资源的最大需求。

(3) 分配矩阵Allocation: $n \times m$ 的矩阵，定义了系统中每一类资源已分配给每一进程的资源数。

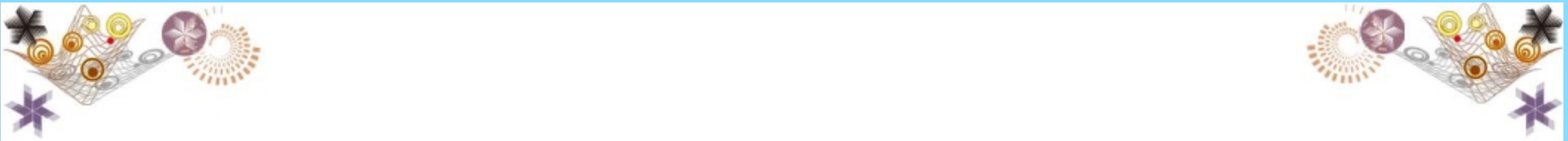
(4) 需求矩阵Need: $n \times m$ 的矩阵，表示每一个进程尚需的各类资源数， $Need = Max - Allocation$ 。

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 $Request_i[j] \leq Need[i, j]$ ，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 $Request_i[j] \leq Available[j]$ ，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：

$$\text{Available}[j] = \text{Available}[j] - \text{Request}_i[j];$$

$$\text{Allocation}[i, j] = \text{Allocation}[i, j] + \text{Request}_i[j];$$

$$\text{Need}[i, j] = \text{Need}[i, j] - \text{Request}_i[j];$$

(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

系统所执行的安全性算法可描述如下：

(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 $\text{Finish}[i] = \text{false}$;

② $\text{Need}[i, j] \leq \text{Work}[j]$;

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：

$\text{Work}[j] = \text{Work}[j] + \text{Allocation}[i, j]$;

$\text{Finish}[i] = \text{true}$;

go to step 2;

(4) 如果所有进程的 $\text{Finish}[i] = \text{true}$ 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。

3.8 死锁的检测与解除

如果在系统中，既不采取死锁预防措施，也未配有死锁避免算法，系统很可能会发生死锁。在这种情况下，系统应当提供两个算法：

① **死锁检测算法**。该方法用于检测系统状态，以确定系统中是否发生了死锁。

② **死锁解除算法**。当认定系统中已发生了死锁，利用该算法可将系统从死锁状态中解脱出来。

3.8.1 死锁的检测

为了能对系统中是否已发生了死锁进行检测，在系统中必须：① 保存有关资源的请求和分配信息；② 提供一种算法，它利用这些信息来检测系统是否已进入死锁状态。

1. 资源分配图(Resource Allocation Graph)

系统死锁，可利用资源分配图来描述。

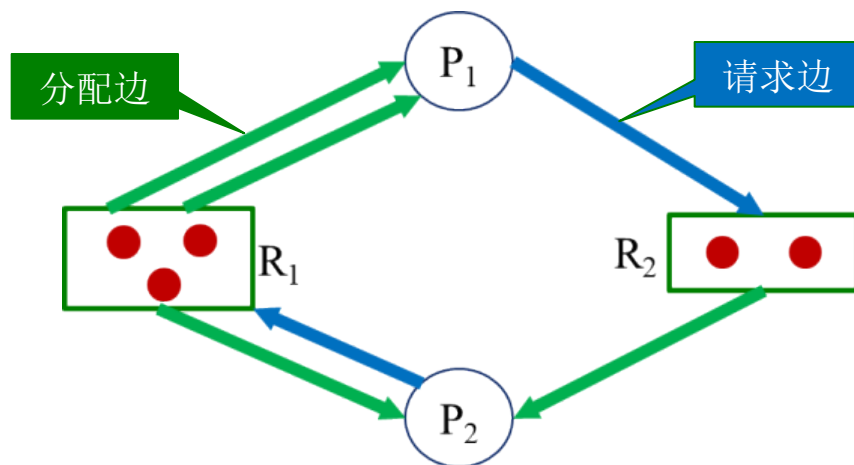


图3-19 每类资源有多个时的情况

该图是由一组结点N和一组边E所组成的一个对偶 $G = (N, E)$ ，它具有下述形式的定义和限制：

(1) 把N分为两个互斥的子集，即一组**进程结点** $P = \{P_1, P_2, \dots, P_n\}$ 和一组**资源结点** $R = \{R_1, R_2, \dots, R_n\}$ ， $N = P \cup R$ 。在图3-19所示的例子中， $P = \{P_1, P_2\}$ ， $R = \{R_1, R_2\}$ ， $N = \{R_1, R_2\} \cup \{P_1, P_2\}$ 。

(2) 凡属于E中的一个边 $e \in E$ ，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 。图3-19中示出了两个请求边和两个分配边，即 $E = \{(P_1, R_2), (R_2, P_2), (P_2, R_1), (R_1, P_1)\}$ 。

2. 死锁定理

可以利用把资源分配图加以简化的方法(图3-19), 来检测当系统处于S状态时, 是否为死锁状态。

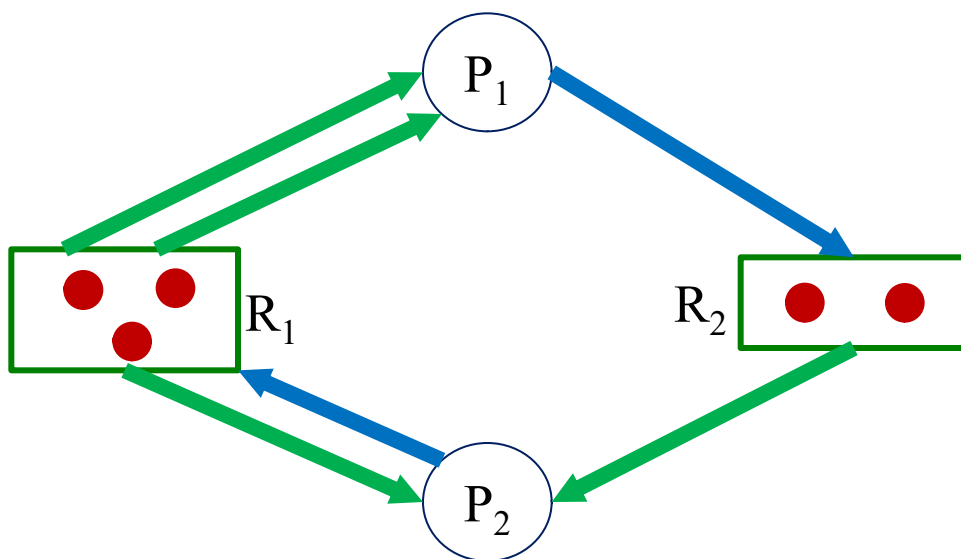


图3-20 资源分配图的简化



简化方法如下：

(1) 在资源分配图中，**找出一个既不阻塞又非独立的进程结点 P_i** ，消去 P_i 的请求边和分配边，使之成为孤立的结点。

(2) P_1 释放资源后，便可使 P_2 获得资源而继续运行，直至 P_2 完成后又释放出它所占有的全部资源，即将 P_2 的两条请求边和一条分配边消去。

(3) 在进行一系列的简化后，若能消去图中所有的边，使**所有的进程结点都成为孤立结点**，则称该图是**可完全简化的**；若不能通过任何过程使该图完全简化，则称该图是**不可完全简化**的。

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的。

3. 死锁检测中的数据结构

死锁检测中的数据结构类似于银行家算法中的数据结构：

(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。



3.8.2 死锁的解除

当发现有进程死锁时，便应立即把它们从死锁状态中解脱出来。常采用解除死锁的两种方法：

(1) **抢占资源**。从一个或多个死锁进程中抢占足够数量的资源，分配给其他死锁进程，以解除死锁状态。

(2) **终止（或撤销）进程**。终止（或撤销）系统中的一个或多个死锁进程，直至打破循环环路，使系统从死锁状态解脱出来。

在出现死锁时，可采用各种策略来撤消进程。例如，为解除死锁状态所需撤消的进程数目最小；或者，撤消进程所付出的代价最小等。



1. 终止进程的方法

1) 终止所有死锁进程

这是一种最简单的方法，即是**终止所有的死锁进程**，死锁自然也就解除了，但所付出的代价可能会很大。

因为其中有些进程可能已经运行了很长时间，已接近结束，一旦被终止真可谓“功亏一篑”，以后还得从头再来。还可能会有其它方面的代价。

2) 逐个终止进程

稍微温和的方法是：按照某种顺序，**逐个地终止进程**，直至有足够的资源，以打破循环等待，把系统从死锁状态解脱出来为止。

缺点：**该方法所付出的代价也可能很大**。因为每终止一个进程，都需要用死锁检测算法确定系统死锁是否已经被解除，若未解除还需再终止另一个进程。

选择策略：最主要的依据是为死锁解除所付出的“代价最小”。但怎么样才算是“代价最小”，很难有一个精确的度量。

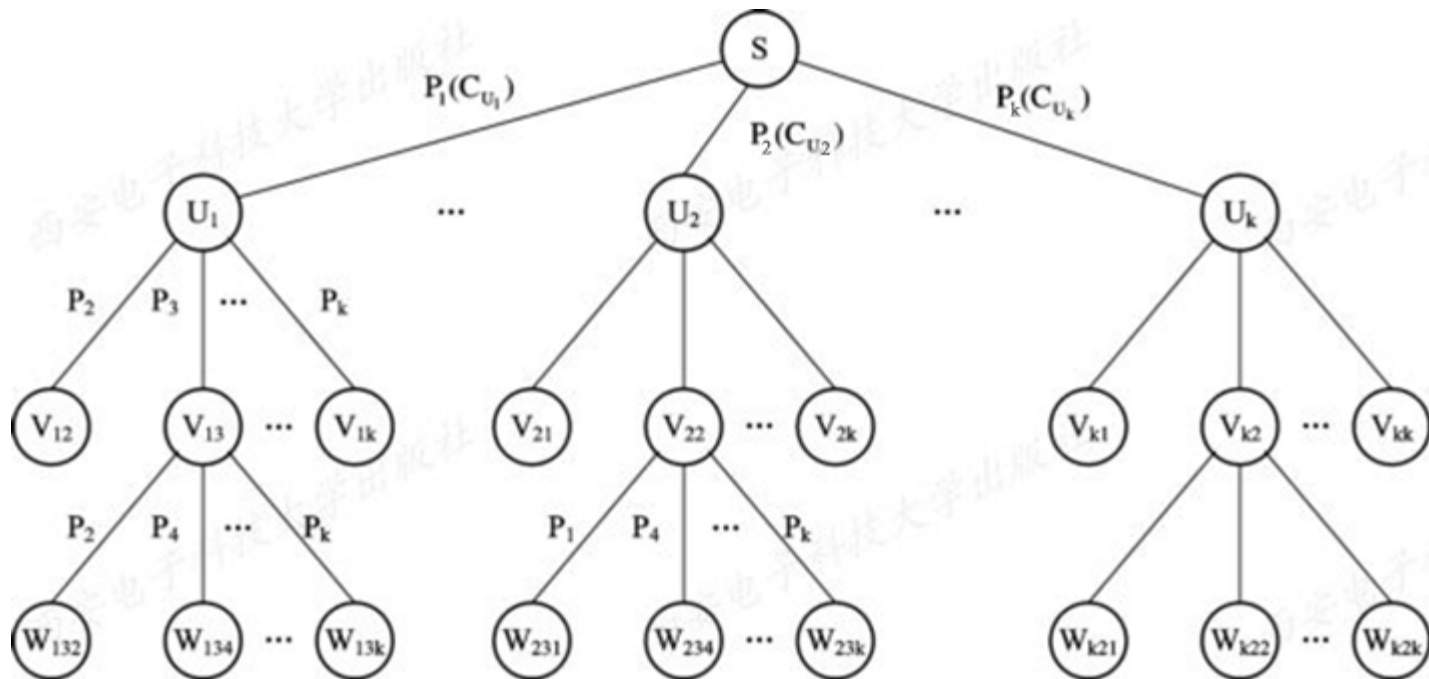


在选择被终止进程时应考虑的若干因素：



- ① 进程的优先级的大小；
- ② 进程已执行了多少时间，还需要多少时间才能完成？
- ③ 进程在运行中已经使用资源的多少，以后还需要多少资源？
- ④ 进程的性质是交互式的还是批处理式的？

2. 付出代价最小的死锁解除算法

一种付出代价最小的死锁解除算法如下图所示。



在所有的解除死锁的序列中找到代价最小的，然后解除死锁。**缺点：需要穷举所有解除死锁序列，代价太大。**



另外一种比较有效的方法（局部最优法）：

1. 从死锁状态 S 中，依次计算解除 P_1, P_2, \dots, P_k 进程的代价，从中选择代价最小的花费，终止该进程；
2. 若系统仍然处于死锁状态，则重复步骤1。
3. 直至死锁状态解除为止。

解除死锁的过程可能不是代价最小的，但是效率是最高的。