第四章 处理机调度与死锁

操作系统的性能在很大程度上取决于处理机调度性能的好坏，因而，处理机调度便成为操作系统设计的中心问题之一。

提高处理机的利用率及改善系统性能（吞吐量、响应时间）是处理机调度的主要目标。

在多道程序环境下，进程数目往往多于处理机数目。这就要求系统能按照某种算法，动态地把处理机分配给就绪队列中的一个进程，使之执行。

本章主要讲述各种常用调度算法及其优缺点；介绍死锁及其解决的办法。

本章主要内容

4.1 调度的基本概念

4.1.1 作业概念及状态

1. 作业：在一次应用业务处理过程中，从输入开始到输出结束，用户要求计算机所做的有关该次业务处理的全部工作为一个作业。（一系列交互命令可看成是一个作业）

如：用语言编制一个程序，系统完成如下工作：①编辑 ②编译 ③链接 ④执行

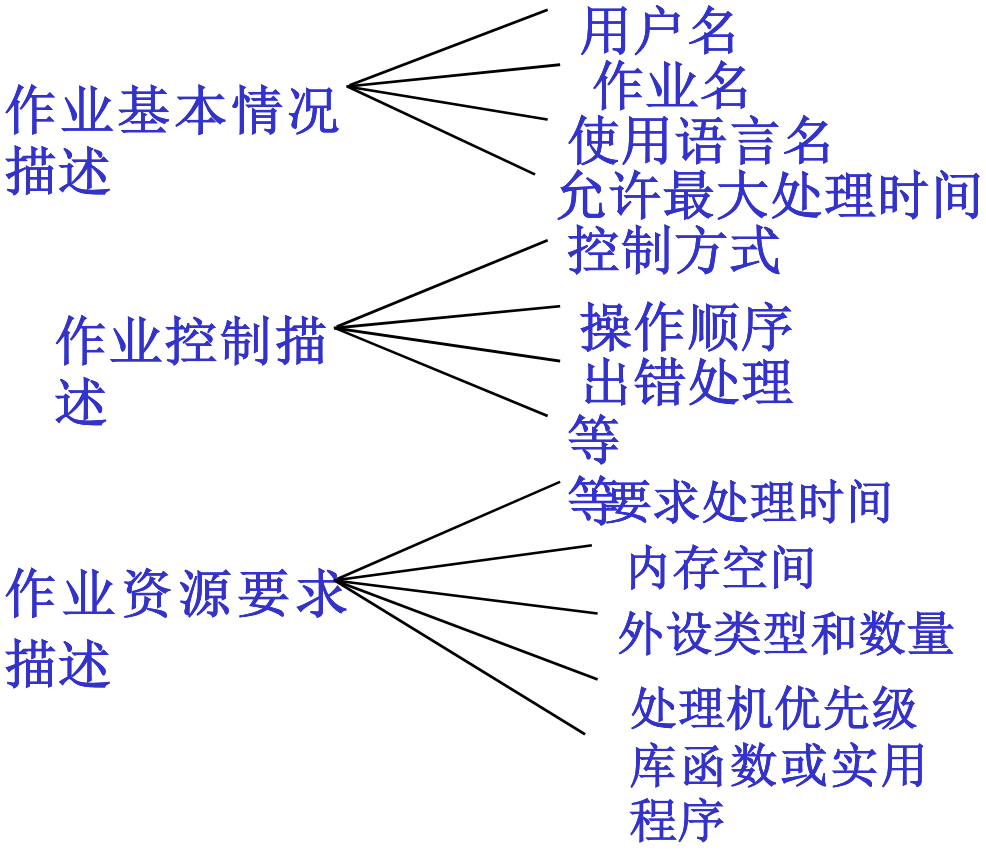
以上几个步骤总和就是一个作业。（作业的概念大于进程）

作业步：作业步是在一个作业的处理过程中，计算机相对独立的工作。

作业的组成：由程序、数据和作业说明书组成。

微机中：批处理文件或SHELL程序方式编写作业说明书。

作业说明书的主要内容

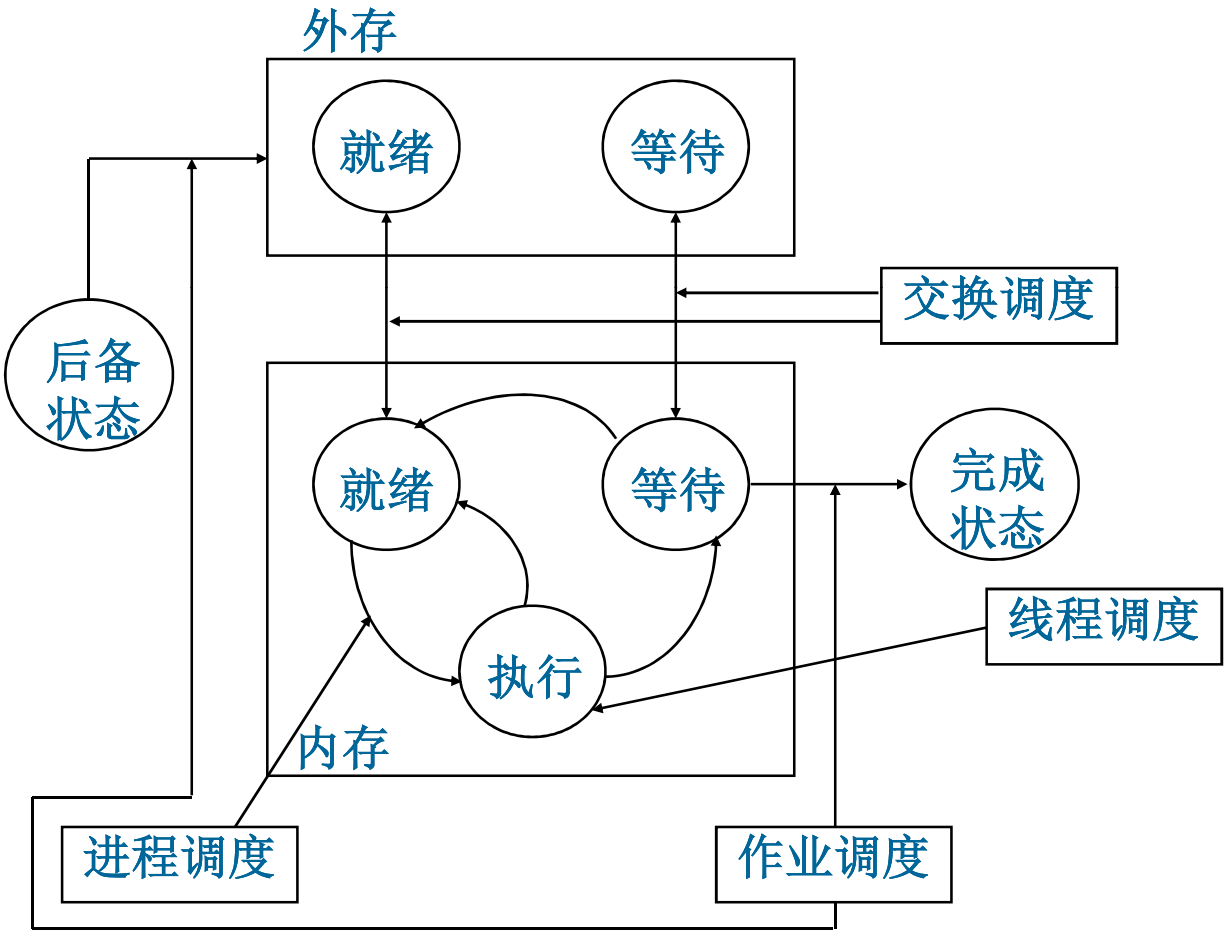


每个作业进入系统时由系统为根据作业本说明书的内容为其建立一个作业控制块JCB（Job Control Block)，它是存放作业控制和管理信息的数据结构,作业存在于系统的标志，内容大致同作业说明书。

2. 作业的状态及其转换

不同的阶段对应不同的状态：

后备状态　　执行状态　　完成状态



4.1.2 分级调度

作业调度

中级调度

进程调度

线程调度

4.1.3 调度的功能和时机

1．作业调度

作业调度的主要功能：

（1）根据作业控制块中的信息，审查系统能否满足用户作业的资源需求，按照一定的算法，从外存的后备队列中选取某些作业调入内存；

（2）为这些作业创建进程、分配必要的资源。

（3）将新创建的进程插入就绪队列，准备执行。

因此，有时也把作业调度称为接纳调度(Admission Scheduling)。

用户和系统对周转时间的要求不同。

为此，每个系统在选择作业调度算法时，既应考虑用户的要求，又能确保系统具有较高的效率。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

作业调度每次要接纳多少个作业进入内存，取决于多道程序度(Degree of Multiprogramming)，即允许多少个作业同时在内存中运行。作业数目太多时，可能会使周转时间太长。但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。最简单的是先来先服务调度算法，这是指将最早进入外存的作业最先调入内存；较常用的一种算法是短作业优先调度算法，是将外存上最短的作业最先调入内存；另一种较常用的是基于作业优先级的调度算法，该算法是将外存上优先级最高的作业优先调入内存；比较好的一种算法是“响应比高者优先”的调度算法。

在批处理系统中，作业先驻留在外存的后备队列上，因此需要有作业调度的过程，以便将它们分批地装入内存。

在分时系统中，为了做到及时响应，用户通过键盘输入的命令或数据等都是被直接送入内存的，也需要有某些限制性措施来限制进入系统的用户数。即，如果系统尚未饱和，将接纳所有授权用户，否则，将拒绝接纳。在实时系统中通常也不需要作业调度。

作业调度的时机：

（1）主机上作业数小于支持的系统最大作业数时，同时后备队列中有作业时，启动作业调度。

（2）当一个作业终止而被撤销后，如果存在后备作业，则启动作业调度。

2. 低级调度

通常也把低级调度(Low Level Scheduling)称为进程调度或短程调度(ShortTerm Scheduling)，它所调度的对象是进程(或内核级线程)。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。

低级调度用于决定就绪队列中的哪个进程(或内核级线程，为叙述方便，以后只写进程)应获得处理机。

低级调度的主要功能如下：

（1）保存处理机的现场信息。在进程调度进行调度时，首先需要保存当前进程的处理机的现场信息，如程序计数器、多个通用寄存器中的内容等，将它们送入该进程的进程控制块(PCB)中的相应单元。

(2) 按某种算法选取进程。低级调度程序按某种算法如优先数算法、轮转法等，从就绪队列中选取一个进程，把它的状态改为运行状态，并准备把处理机分配给它。

(3) 把处理器分配给进程。由分派程序(Dispatcher)把处理器分配给进程。此时需为选中的进程恢复处理机现场，即把选中进程的进程控制块内有关处理机现场的信息装入处理器相应的各个寄存器中，把处理器的控制权交给该进程，让它从取出的断点处开始继续运行。

进程调度方式

进程调度可采用下述两种调度方式。

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，不管它要运行多长时间，都一直让它运行下去，决不会因为时钟中断等原因而抢占正在运行进程的处理机，也不允许其它进程抢占已经分配给它的处理机。直至该进程完成，自愿释放处理机，或发生某事件而被阻塞时，才再把处理机分配给其他进程。

在采用非抢占调度方式时，进程调度的时机：

(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)

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。

抢占方式比非抢占方式调度所需付出的开销较大。抢占调度方式是基于一定原则的，主要有如下几条：

(1) 优先权原则。

(2) 短作业(进程)优先原则。

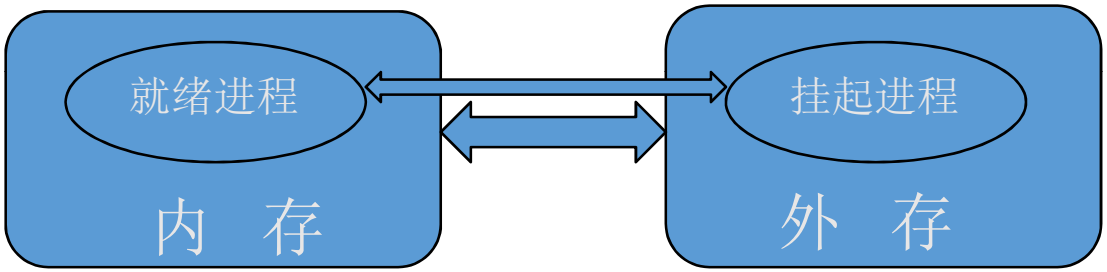
(3) 时间片原则。

◆CPU连续周期(服务时间）：一个进程在CPU的一次连续执行过程称为该进程的一个CPU周期。当进程需要等待某个事件而进入等待态时，便终止了它的当前CPU周期。

例：一个进程需要在CPU上执行1S，在100ms、450ms、600ms的执行点处它分别要等待三个事件而暂停执行，即该进程有四个分别为100ms、350ms、150ms、400ms的CPU连续周期。

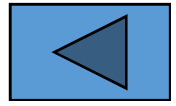
3．中级调度

中级调度(Intermediate Level Scheduling)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。



运行频率

进程调度（短程调度）	10-100ms之间	算法不宜太复杂
作业调度（长程调度）	几分钟	算法可以复杂
中级调度（中程调度）	上述二者之间	



4.1.4 调度原则与性能衡量

原则：公平、有效、高吞吐量、及时响应、支持优先

衡量指标：

（1）周转时间：

$$T_i = T_{ie} - T_{is} = T_{iw} + T_{ir}$$

平均周转时间：

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

（2）平均带权周转时间：

$$W = \frac{1}{n}\sum_{i=1}^{n} W_i \left(W_i = T_i / T_{ir}\right)$$

（3）响应时间

（4）截止时间

4.2 调度算法

4.2.1 先来先服务和短作业(进程)优先调度算法

1. 先来先服务调度算法

先来先服务(FCFS)调度算法是一种最简单的调度算法，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，每次调度都是从后备作业队列中选择一个或多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、创建进程，然后放入就绪队列。

在进程调度中采用FCFS算法时，则每次调度是从就绪队列中选择一个最先进入该队列的进程，为之分配处理机，使之投入运行。该进程一直运行到完成或发生某事件而阻塞后才放弃处理机。

FCFS算法比较有利于长作业(进程)，而不利于短作业(进程)。下表列出了A、B、C、D四个作业分别到达系统的时间、要求服务的时间、开始执行的时间及各自的完成时间，并计算出各自的周转时间和带权周转时间。

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

从表上可以看出，其中短作业C的带权周转时间竟高达100，这是不能容忍的；而长作业D的带权周转时间仅为1.99。据此可知，FCFS调度算法有利于CPU繁忙型的作业，而不利于I/O繁忙型的作业(进程)。CPU繁忙型作业是指该类作业需要大量的CPU时间进行计算，而很少请求I/O。通常的科学计算便属于CPU繁忙型作业。I/O繁忙型作业是指CPU进行处理时需频繁地请求I/O。目前的大多数事务处理都属于I/O繁忙型作业。

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个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。

在此，我们通过一个例子来比较采用FCFS调度算法和使用短作业优先调度算法时的调度性能。图3-4(a)示出有五个进程A、B、C、D、E，它们到达的时间分别是0、1、2、3和4，所要求的服务时间分别是4、3、5、2和4，其完成时间分别是4、7、12、14和18。从每个进程的完成时间中减去其到达时间，即得到其周转时间，进而可以算出每个进程的带权周转时间。

作业 情况 / 调度 算法	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

SJ(P)F调度算法也存在不容忽视的缺点：

(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。

4.2.2　高优先权优先调度算法

1．优先权调度算法的类型

为了照顾紧迫型作业，使之在进入系统后便获得优先处理，引入了最高优先权优先(FPF)调度算法。

此算法常被用于批处理系统中，作为作业调度算法，也作为多种操作系统中的进程调度算法，还可用于实时系统中。

当把该算法用于作业调度时，系统将从后备队列中选择若干个优先权最高的作业装入内存。当用于进程调度时，该算法是把处理机分配给就绪队列中优先权最高的进程，这时，又可进一步把该算法分成如下两种。

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$P_i \leqslant P_j$，原进程Pj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。

抢占式的优先权调度算法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

对于最高优先权优先调度算法，其关键在于：它是使用静态优先权，还是用动态优先权，以及如何确定进程的优先权。

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，不同的系统用法不同。

确定进程优先权的依据有如下三个方面：

（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。

（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。这是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。

静态优先权法简单易行，系统开销小，但不够精确，很可能出现优先权低的作业(进程)长期没有被调度的情况。因此，仅在要求不高的系统中才使用静态优先权。

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。当采用抢占式优先权调度算法时，如果再规定当前执行的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。

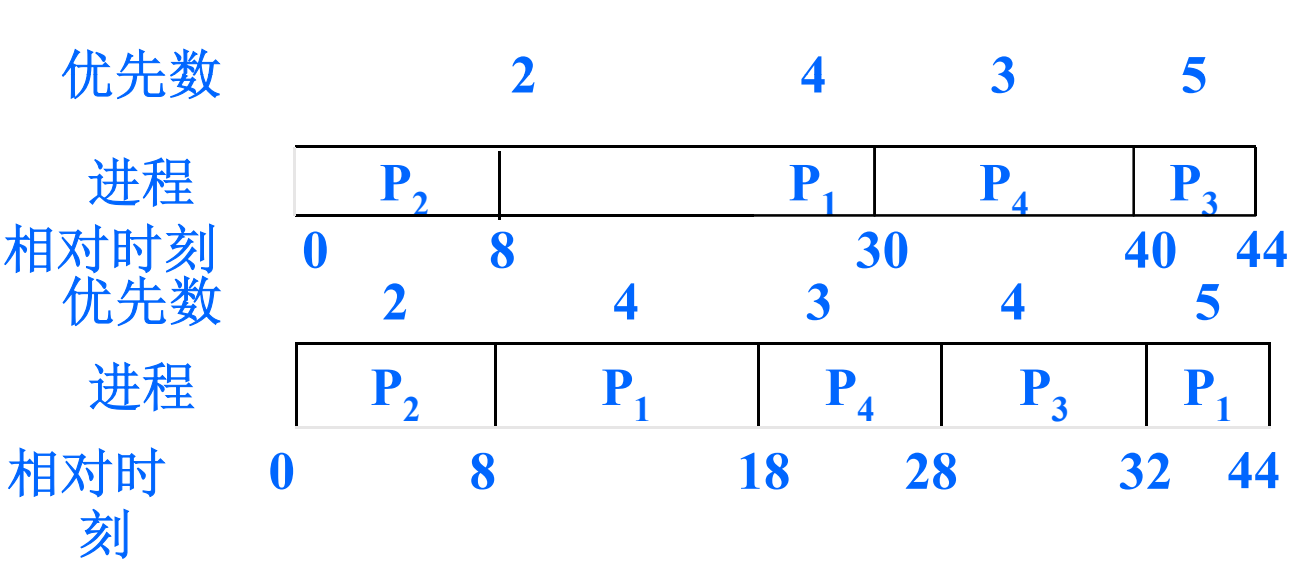
假定有五个就绪进程，它们进入就绪队列的相对时刻、各自的本次CPU周期长度及初始优先数如下表。在此，规定小的优先数表示高的优先级，调度时间忽略不计。

（1）给出采用HPF调度算法，非剥夺静态设置方式时，五个进程的平均周转时间及平均带权周转时间。

（2）给出采用HPF调度算法，在剥夺动态设置方式下，五个进程的平均周转时间及平均带权周转时间。假设现行进程每连续执行10ms以上其优先数加1（即降低优先级），而就绪进程每等待20ms后其优先数减1（即提高优先级）。

HPF(Highest-Priority-First)

进程	到达时刻	CPU周期(ms)	优先数
p_1	0	22	4
P_2	0	8	2
P_3	0	4	5
P_4	18	10	3



T＝26(ms)
W＝3.89(ms)

T＝23.5(ms)
W＝3(ms)

优点：可以使紧迫的任务得到优先执行。

缺点：静态优先级易于实现，系统开销小，但作业或进程的优先级不够精确；动态优先级须计算优先级，增加系统的开销。

3．高响应比优先调度算法

在批处理系统中，短作业优先算法是一种比较好的算法，其主要的不足之处是长作业的运行得不到保证。如果我们能为每个作业引入前面所述的动态优先权，并使作业的优先级随着等待时间的增加而以速率a提高，则长作业在等待一定的时间后，必然有机会分配到处理机。该优先权的变化规律可描述为：

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

R＝1＋已等待的时间/要求服务时间

作业号	到达时刻	运行时间	开始时间	完成时间	周转时间	带权周转时间
1	0	10	0	10	10	1
2	1	5	11	16	15	3
3	5	1	10	11	6	6
4	10	2	16	18	8	4
平均周转时间T＝9.75(ms)；平均带权周转时间W＝3.5（ms）。						

R12=1+(10-1)/5=2.8 R13=1+(10-5)/1=6 R14=1+(10-10)/2=1

R22=1+(11-1)/5=3 R24=1+(11-10)/2=1.5

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：

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

由上式可以看出：

（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。

简言之，该算法既照顾了短作业，又考虑了作业到达的先后次序，不会使长作业长期得不到服务。因此，该算法实现了一种较好的折衷。当然，在利用该算法时，每要进行调度之前，都须先做响应比的计算，这会增加系统开销。

4.2.3 基于时间片的轮转调度算法

1．时间片轮转法

1）基本原理

系统将所有的就绪进程按先来先服务的原则排成一个队列，每次调度时，把CPU分配给队首进程，并令其执行一个时间片。时间片的大小从几ms到几百ms。

当执行的时间片用完时，由一个计时器发出时钟中断请求，调度程序便据此信号来停止该进程的执行，并将它送往就绪队列的末尾；

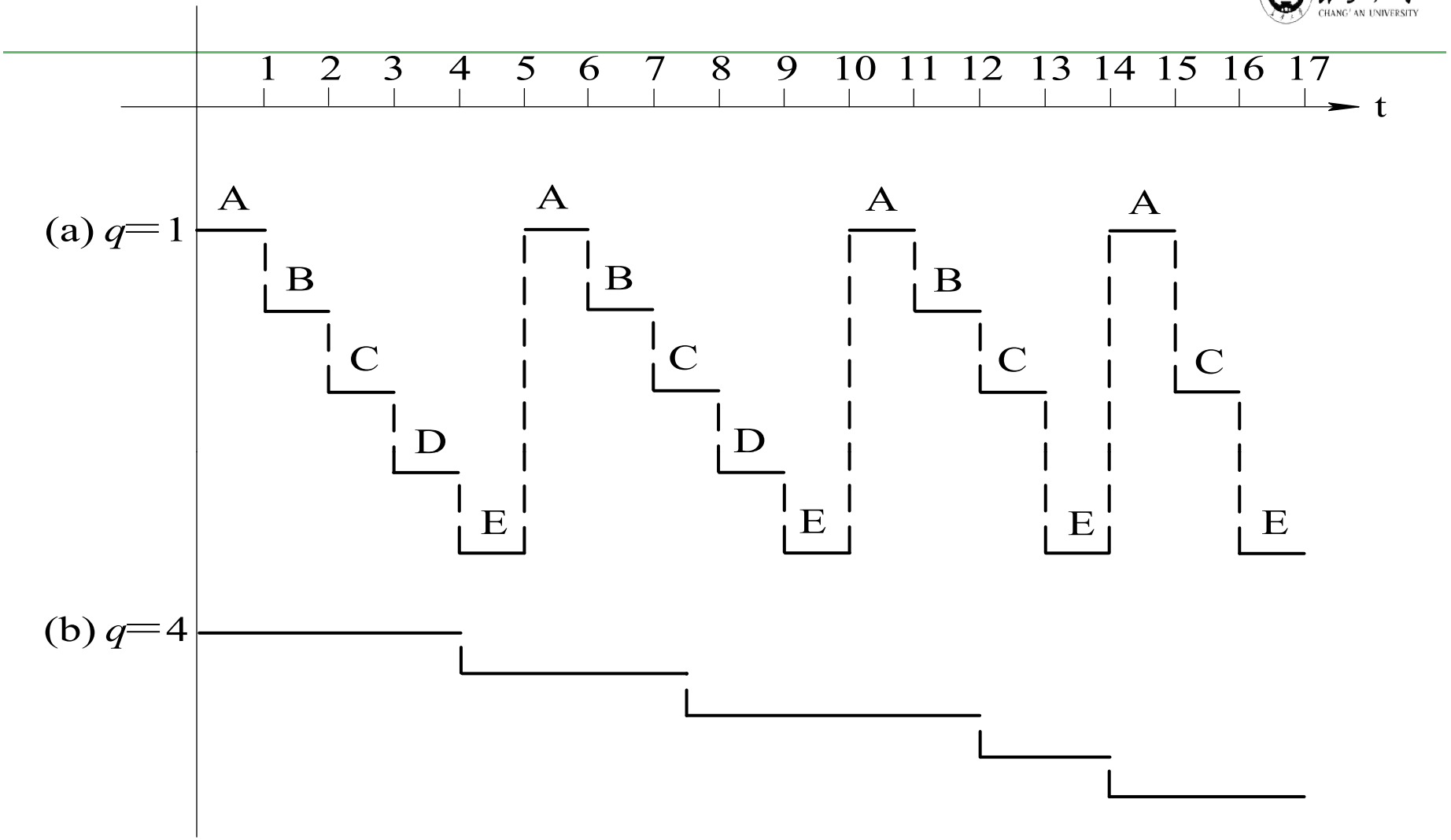
然后，再把处理机分配给就绪队列中新的队首进程，同时也让它执行一个时间片。这样就可以保证系统能在给定的时间内响应所有用户的请求。

2）时间片大小的确定

在时间片轮转算法中，时间片的大小对系统性能有很大的影响。考虑一下：

选择很小的时间片会怎么样？选择太长的时间片又会怎么样？

一个较为可取的大小是，时间片略大于一次典型的交互所需要的时间。这样可使大多数进程在一个时间片内完成。


1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17
t
(a) q=1
A
B
C
D
E
A
B
C
D
E
A
B
C
E
A
C
E
(b) q=4

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	3.33	3.46
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	3.33	2.5

$q=R/Nmax$

优点：可以使用户得到及时的响应和服务。

缺点：短进程用户和I/O繁忙型进程是不利的。特别是，当“紧迫型”的进程到来时，并不能及时的得到处理。

重要问题：时间片q值的设置

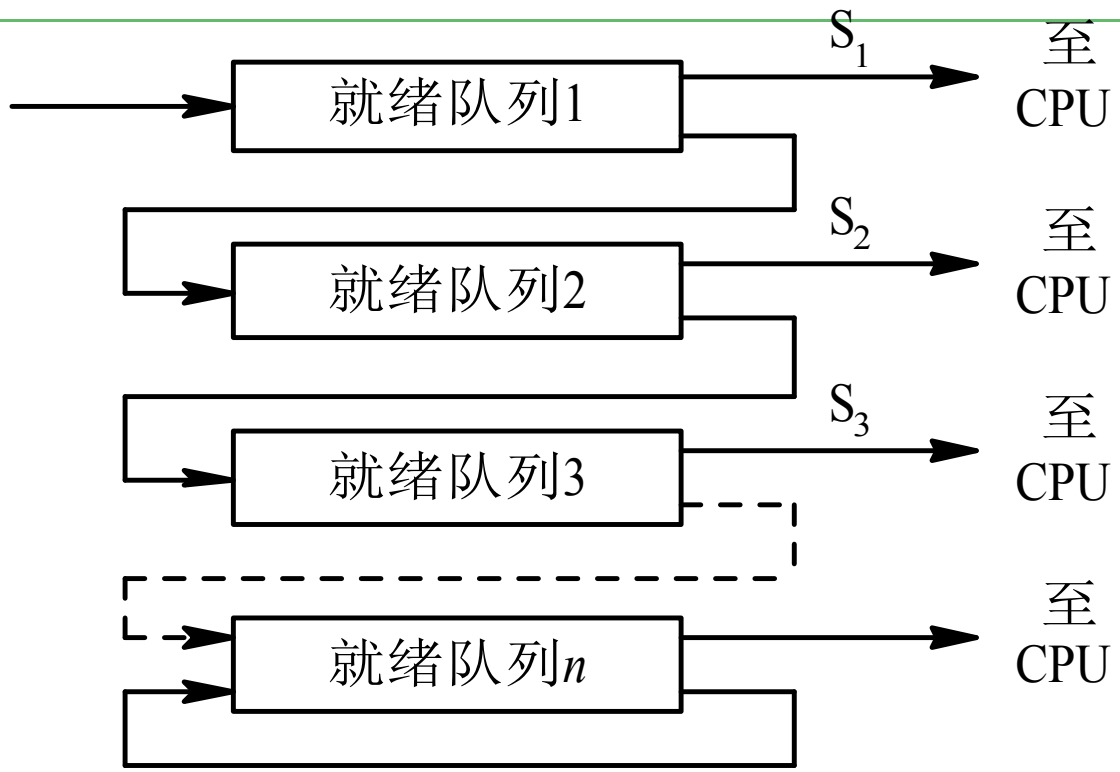
一般来说，时间片是动态设置的，每当一轮新的调度开始，系统便设置一次时间片(为什么？)

例如：假设时间片为0.1ms，当就绪进程n＝20时，用户的响应时间为r=2ms；当n=6时，r=0.6ms，对用户来说，2ms的响应时间和0.6ms的响应时间并没有太大的差别，而系统的进程切换开销并没有减小。

倘若保持响应时间不变为2ms，当n=6时，时间片变为0.35ms，显著的减少了系统开销。因此，为了进一步改善RR算法的调度性能，可采用可变时间片的RR调度算法。一种可行的办法是，每当新的一轮调度开始时，系统便根据就绪队列中已有的进程数目计算一次时间片q，作为新一轮调度的时间片。

2．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。下图是多级反馈队列算法的示意。



（时间片：$S_1 < S_2 < S_3$）

(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。

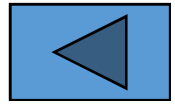
3．多级反馈队列调度算法的性能

多级反馈队列调度算法具有较好的性能，能很好地满足各种类型用户的需要。

(1) 终端型作业用户。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)在第一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。对于很短的批处理型作业，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。



例如，一个进程运行完成需要100个时间片，如果采用RR算法，则这个进程需要切换100次。如果采用MF算法，第一次运行分配给它一个时间片，第二次分配给它2个时间片，依次为4个、8个……64个，则这个进程运行需要切换7次就可完成，大大提高了CPU的使用效率，同时随着运行优先级的不断降低，它的运行频度放慢，为其它进程让出了CPU。

4.2.4　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。

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

分配给每个进程相同的处理机时间，对进程公平，但如果各个用户所拥有的进程数不同，就会发生对用户的不公平问题。

在该调度算法中，调度的公平性主要是针对用户而言，使所有的用户都能获得相同的处理机时间，或所要求的时间比例。

例如系统中有两个用户，用户1有四个进程A、B、C、D，用户2只有1个进程E。为保证两个用户能获得相同的处理机时间，则必须执行如下所示的强制调度序列：

A E B E C E D E A E B E C E D E

如果希望用户1所获得的处理机时间是用户2的两倍，则必须执行如下所示的强制调度序列：

A B E C D E A B E C D E A B E C D E

4.3 实时调度

4.3.1 实时任务的特点

1．每个实时任务都要联系着一个截止时间。实时任务的处理和控制的正确性不仅取决于计算结果的正确性，而且取决于计算结果产生的时间。

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一些信息：

(1) 就绪时间。这是该任务成为就绪状态的起始时间，在周期任务的情况下，它就是事先预知的一串时间序列；而在非周期任务的情况下，它也可能是预知的。

(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，或者知道完成截止时间。

(3) 处理时间。这是指一个任务从开始执行直至完成所需的时间。在某些情况下，该时间也是系统提供的。

(5) 优先级。 “绝对”优先级和“相对”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Ci，周期时间表示为Pi，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

在含有硬实时任务的实时系统中，广泛采用抢占机制。但这种调度机制比较复杂。

对于一些小型实时系统，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

为保证要求较高的硬实时任务能及时运行，在实时系统中还应具有快速切换机制，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。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地小，以减少任务切换的时间开销。

4.3.2 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

该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依次调度运行。

这种调度算法可获得数秒至数十秒的响应时间，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

如果在实时系统中存在着要求较为严格(响应时间为数百毫秒)的任务，则可采用非抢占式优先调度算法为这些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。这种调度算法在做了精心的处理后，有可能获得仅为数秒至数百毫秒级的响应时间，因而可用于有一定要求的实时控制系统中。

2．抢占式调度算法

在要求较严格的(响应时间为数十毫秒以下)的实时系统中，应采用抢占式优先权调度算法。可根据抢占发生时间的不同而进一步分成以下两种调度算法。

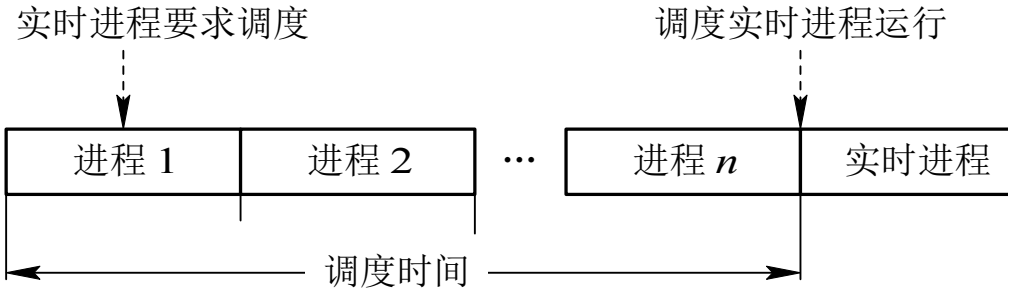
1）基于时钟中断的抢占式优先权调度算法

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。这种调度算法能获得较好的响应效果，其调度延迟可降为几十毫秒至几毫秒。因此，此算法可用于大多数的实时系统中。

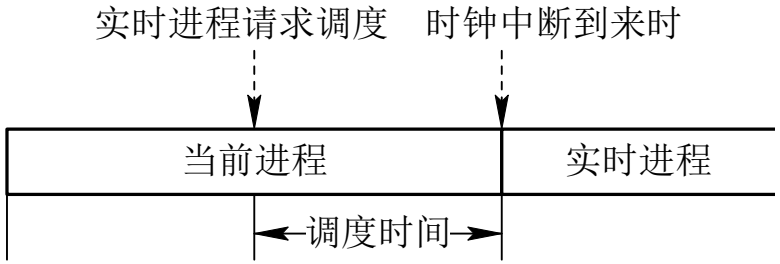
2）立即抢占(Immediate Preemption)的优先权调度算法

在这种调度策略中，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。这种算法能获得非常快的响应，可把调度延迟降低到几毫秒至100微秒，甚至更低。

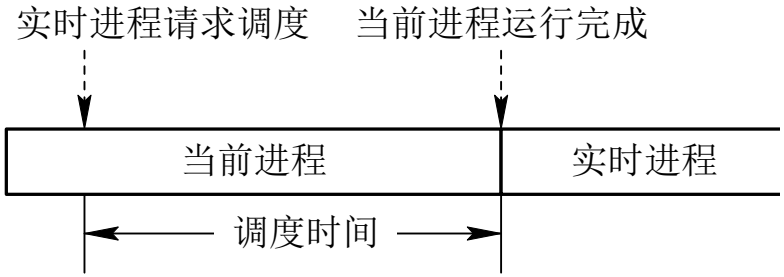
下图中的(a)、(b)、(c)、(d)分别示出了采用非抢占式轮转调度算法、非抢占式优先权调度算法、基于时钟中断抢占的优先权调度算法和立即抢占的优先权调度算法四种情况的调度时间。



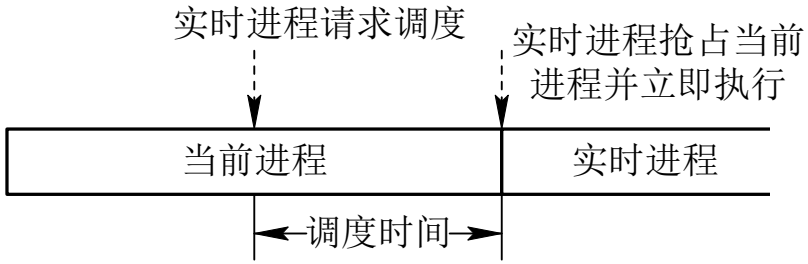
(a) 非抢占式轮转调度



(c) 基于时钟中断抢占的优先权抢占调度



(b) 非抢占式优先权调度



(d) 立即抢占的优先权调度

4.3.3 常用的几种实时调度算法

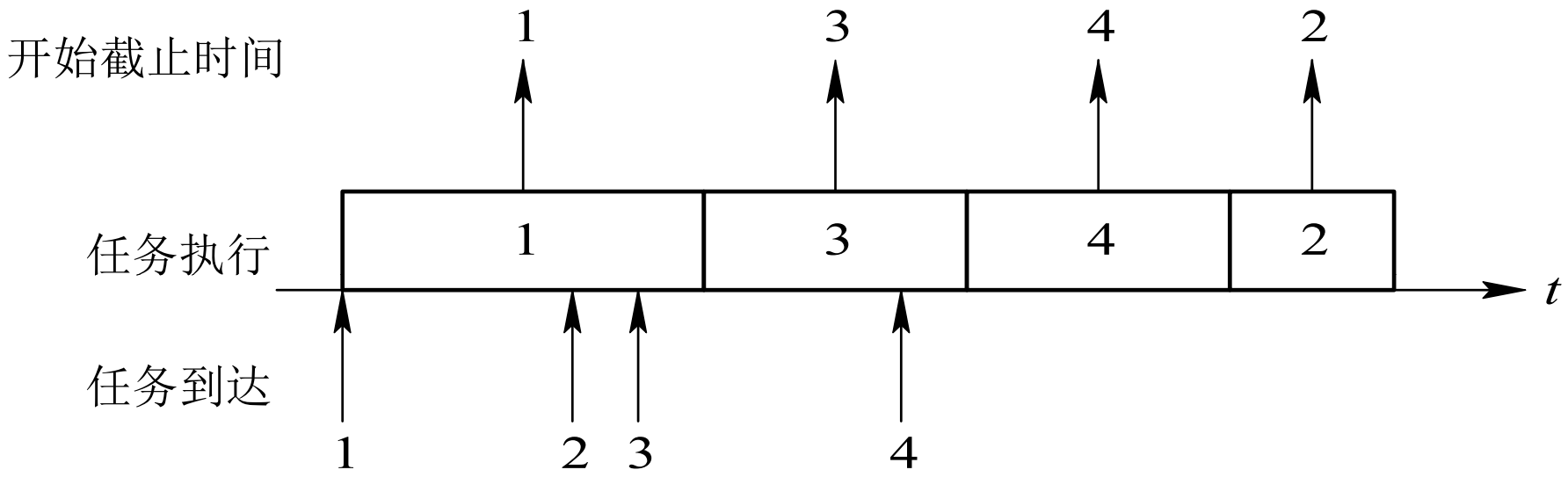
1．最早截止时间优先即EDF(Earliest Deadline First)算法

该算法是根据任务的开始截止时间来确定任务的优先级。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；当然，具有最早截止时间的任务排在队列的最前面。

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。最早截止时间优先算法既可用于抢占式调度，也可用于非抢占式调度方式。

1）非抢占式调度方式用于非周期实时任务

下图示出了将该算法用于非抢占调度方式之例。该例中具有四个非周期任务，它们先后到达。系统首先调度任务1执行，在任务1执行期间，任务2、3又先后到达。由于任务3的开始截止时间早于任务2，故系统在任务1后将调度任务3执行。在此期间又到达作业4，其开始截止时间仍是早于任务2的，故在任务3执行完后，系统又调度任务4执行，最后才调度任务2执行。

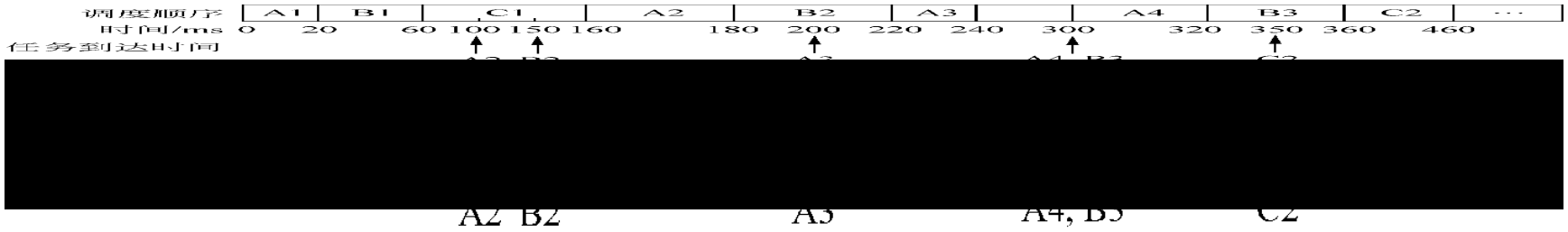

开始截止时间
1
3
4
2
任务执行
1
3
4
2
t
任务到达
1
2
3
4

频率单调调度RMS（Rate-Monotontic Scheduling）算法

RMS算法是面向周期性任务的非抢占式调度算法。该算法的基本原理是频率越低（周期越长）的任务优先级越低。已经证明，RMS算法可调度的充分条件是：

$$\sum_{i=1}^{n} \frac{C_i}{P_i} \leq n(2^{\frac{1}{n}} - 1)$$

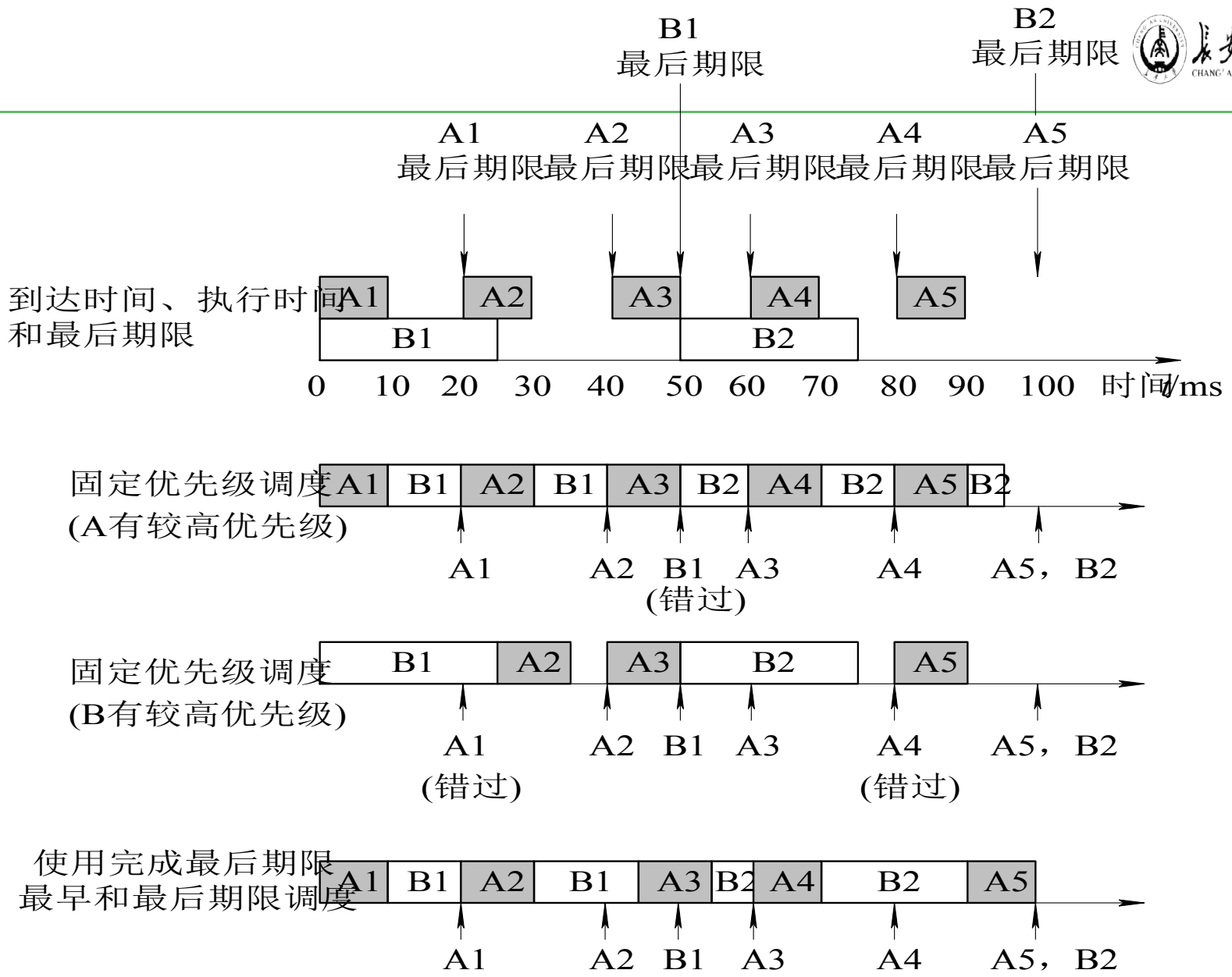
例4-5：　有A、B、C3个周期性任务,它们的发生周期T1、T2、T3 分别为100ms、150ms、350ms,每个周期任务的处理时间C1、C2、C3 分别为20ms、40ms、100ms,可否采用频率单调调度算法进行调度? 如果可以,则画出它们的进程调度顺序。


调度顺序
A1
B1
C1
A2
B2
A3
A4
B3
C2
...
时间/ms
0
20
60
100
150
160
180
200
220
240
300
320
350
360
460
任务到达时间
A2 B2
A3
A4, B3
C2

RMS算法的调度情况

3）抢占式调度方式用于周期实时任务

下图示出了将最早截止时间优先算法用于抢占调度方式之例。在该例中有两个周期性任务，任务A的周期时间为20 ms，每个周期的处理时间为10 ms；任务B的周期时间为50 ms，每个周期的处理时间为25 ms。图中的第一行示出了两个任务的到达时间、最后期限和执行时间图。其中任务A的到达时间为0、20、40、…；任务A的最后期限为20、40、60、…；任务B的到达时间为0、50、100、…；任务B的最后期限为50、100、150、…(注：单位皆为ms)。


B1
最后期限
B2
最后期限
A1
最后期限
A2
最后期限
A3
最后期限
A4
最后期限
A5
最后期限
到达时间、执行时间
和最后期限
A1
A2
A3
A4
A5
B1
B2
0 10 20 30 40 50 60 70 80 90 100 时间/ms
固定优先级调度
(A有较高优先级)
A1 B1 A2 B1 A3 B2 A4 B2 A5 B2
A1 A2 B1 A3 A4 A5，B2
(错过)
固定优先级调度
(B有较高优先级)
B1 A2 A3 B2 A5
A1 A2 B1 A3 A4 A5，B2
(错过) (错过)
使用完成最后期限
最早和最后期限调度
A1 B1 A2 B1 A3 B2 A4 B2 A5
A1 A2 B1 A3 A4 A5，B2

为了说明通常的优先级调度不能适用于实时系统，该图特增加了第二和第三行。在第二行中假定任务A具有较高的优先级，所以在 t=0 ms时，先调度A1执行，在A1完成后(t = 10 ms)才调度B1执行；在t = 20 ms时，调度A2执行；在t = 30 ms时，A2完成，又调度B1执行；在t = 40 ms时，调度A3执行；在t = 50 ms时，虽然A3已完成，但B1已错过了它的最后期限，这说明了利用通常的优先级调度已经失败。第三行与第二行类似，只是假定任务B具有较高的优先级。

第四行是采用最早截止时间优先算法的时间图。在t＝0时，A1和B1同时到达，由于A1的截止时间比B1早，故调度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，B1被中断而调度A2执行；在t＝30时，A2完成，又重新调度B1执行；在t＝40时，A3又到达，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＝55时，A3完成，又调度B2执行。在该例中利用最早截止时间优先算法可以满足系统的要求。

2．最低松弛度优先即LLF(Least Laxity First)算法

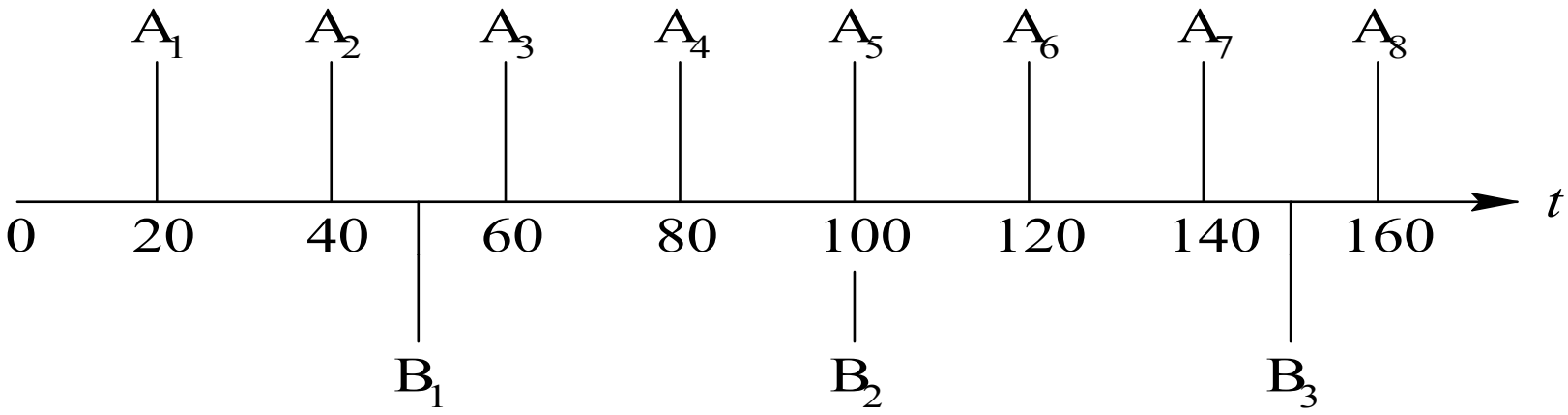
该算法是根据任务紧急(或松弛)的程度，来确定任务的优先级。任务的紧急程度愈高，为该任务所赋予的优先级就愈高，以使之优先执行。

例如，一个任务在200 ms时必须完成，而它本身所需的运行时间就有100 ms，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。由此可得知任务A和B每次必须完成的时间分别为：A1、A2、A3、…和B1、B2、B3、…，见图3-11。为保证不遗漏任何一次截止时间，应采用最低松弛度优先的抢占调度策略。

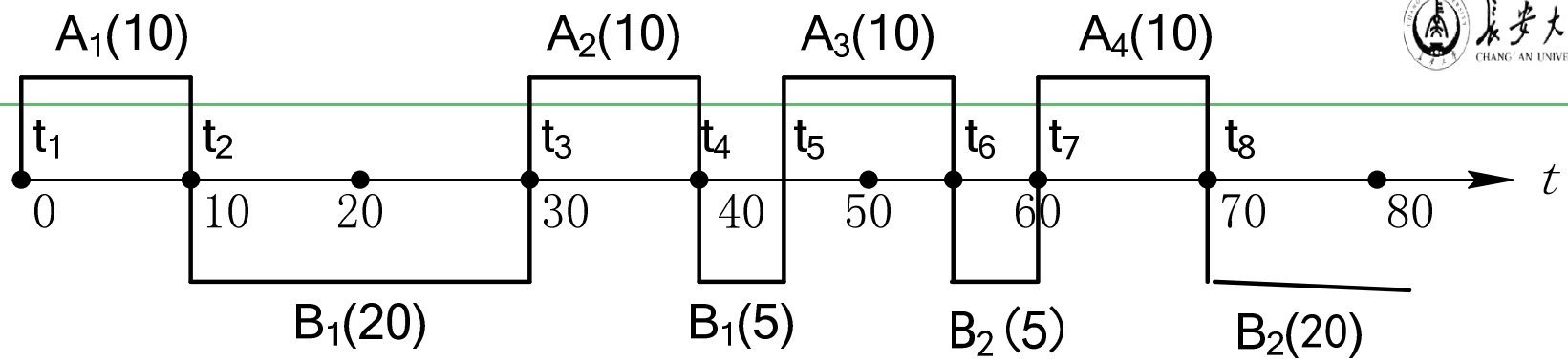

A1
A2
A3
A4
A5
A6
A7
A8
t
0
20
40
60
80
100
120
140
160
B1
B2
B3

在刚开始时($t_1 = 0$)，A_1必须在20 ms时完成，而它本身运行又需 10 ms，可算出A_1的松弛度为10 ms；B_1必须在50 ms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15 ms，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$t_6=55$ ms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A4执行。图3-12示出了具有两个周期性实时任务的调度情况。



$t_1 = 0$， A_1=20-10=10 B_1=50-25=25

$t_2 = 10$, A_2=40-10-10=20, B_1 =50-25-10=15

$t_3 = 30$, A_2 =40-10-30=0,B_1 =50-5-30=15,

$t_4 = 40$,A3=60-10-40=10,B1=50-40-5=5

$t_5 = 45$, A3=60-10-45=5,B2=100-25-45=30

t_6=55,调度B2执行 t7=60,A4=80-60-10=10,B2=100-20-60=20,A4执行

t_8=70,A4＝80－10－70＝0， B2=100-10-70=20

4.3.4 优先级倒置（priority inversion problem）

1.优先级倒置的形成

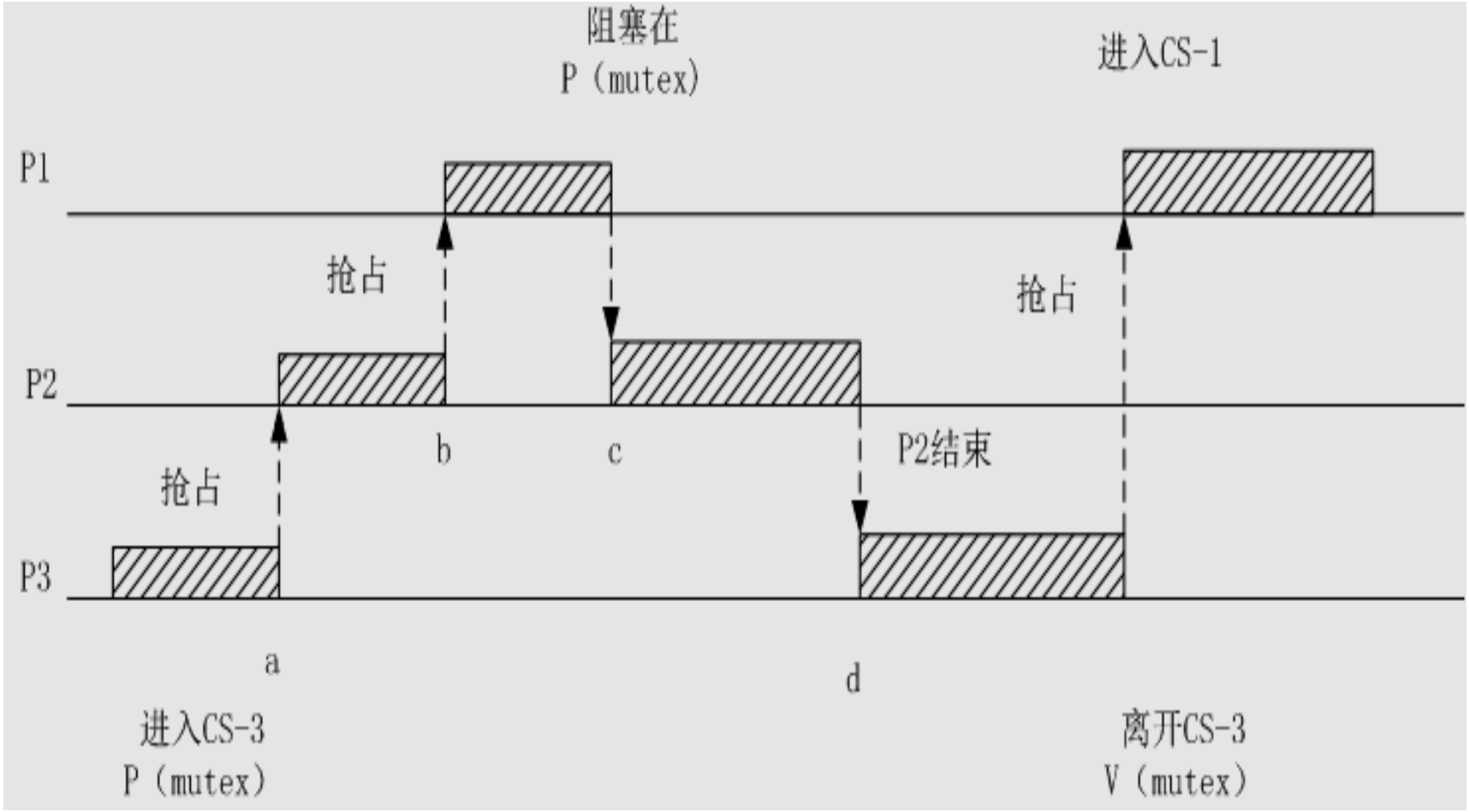
当前操作系统广泛采用优先级调度算法和抢占方式，然而在系统中存在着影响进程运行的资源而可能产生“优先级倒置”的现象，即高优先级进程（或线程）被低优先级进程或线程延迟或阻塞。

例如有三个完全独立的进程P1，P2，P3,P1的优先级最高，P2次之，P3最低。P1和P3通过共享的一个临界资源进行交互。下面是一段代码：

P1：...P(mutex);CS-1;V(mutex);...

P2: ...program2

P3: ...P(mutex);CS-3;V(mutex);...

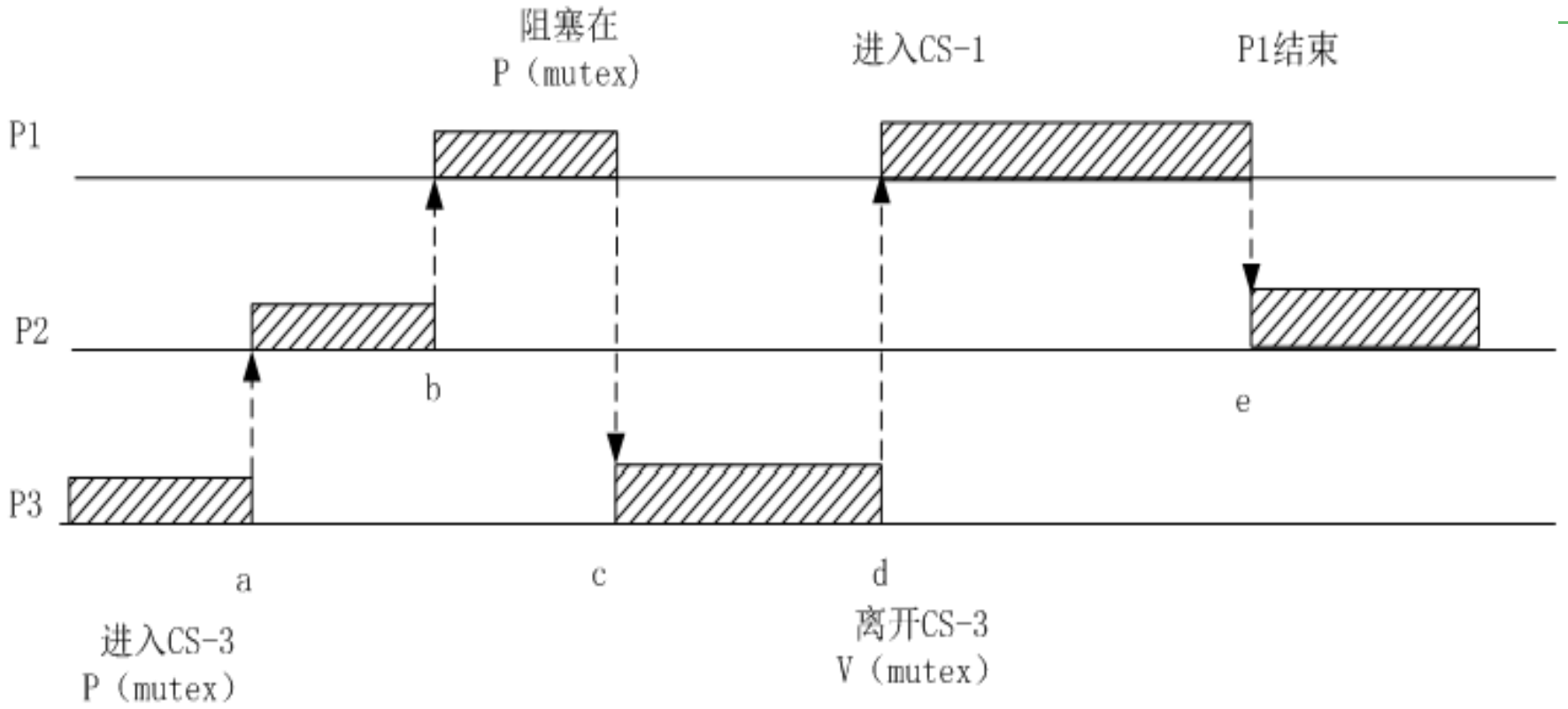

阻塞在
P（mutex）
进入CS-1
P1
抢占
抢占
P2
抢占
b
c
P2结束
P3
a
d
进入CS-3
P（mutex）
离开CS-3
V（mutex）

2.优先级倒置的解决办法

一种简单的解决方法是规定：假如进程P3在进入临界区后P3所占用的处理机就不允许被抢占。如果系统中的临界区都较短且不多，该方法可行。反之，如果P3临界区非常长，则高优先级进程P1仍会等待很长的时间，其效果是无法令人满意的。

个比较实用的方法是建立在动态优先级继承基础上的。该方法规定，当高优先级进程P1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P3正在使用该资源，此时一方面P1被阻塞，另一方面由P3继承P1的优先级，并一直保持到P3退出临界区。

这样做的目的在于不让比P3优先级稍高，但比P1优先级低的进程如P2进程插进来，导致延缓P3退出临界区。由图3-13可以看出，采用动态优先级继承方法后，P1,P2,P3三个进程的运行情况。可以看出，在时刻c，P1被阻塞，但由于P3已继承了P1的优先级，它比P2优先级高，这样就避免了P2的插入，使P1在时刻d进入临界区。该方法已在一些操作系统中得到应用。


阻塞在
P（mutex）
进入CS-1
P1结束
P1
P2
P3
b
e
a
c
d
进入CS-3
P（mutex）
离开CS-3
V（mutex）

3.最低松弛度优先LLF算法

最低松弛度优先(Least-laxity-First,LLF)算法是根据任务的松弛度来确定任务的优先级。任务的松弛度越高,为该任务所赋予的优先级就越高,以使之优先执行。

例4-7: 对于例4-6的两个周期性任务,画出采用LLF算法时它们的调度顺序。

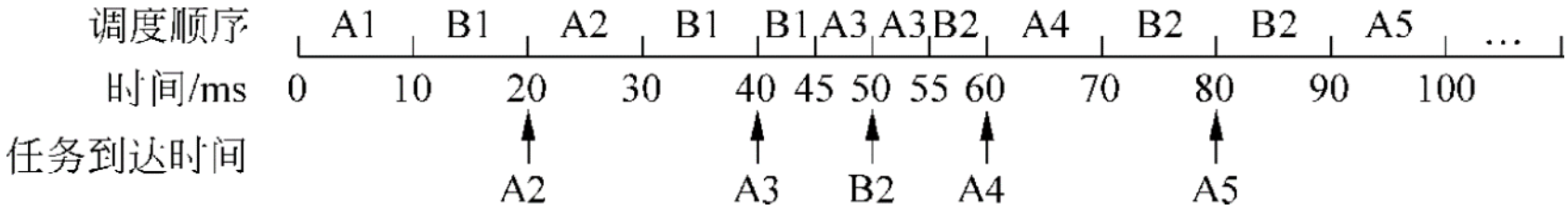
在20ms时,进程A2的松弛度可按下式算出:

A2的松弛度= 截止时间- 其本身的运行时间- 当前时间

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

B1的松弛度= 截止时间- 其本身的运行时间- 当前时间

=50ms-15ms-20ms=15ms



4.4 多处理机调度

4.4.1 多处理机系统的类型

1.紧密耦合MPS:总线连接多个处理机；共享存储器和外设；统一的OS。

松散耦合MPS：通信线路；有各自的存储器和外设；本地OS。

2．对称多处理器系统：所有处理器一样

非对称多处理器系统不同处理器，有一主多分处理器。

4.4.2 多处理机系统调度方式

1．非对称多处理机系统调度方式：主机分配各个进程，空闲机向主机发出请求。简单，主机故障和瓶颈

2．对称多处理机系统调度方式

自调度：系统种只有一个就绪队列，某一处理机空闲就取一个进程运行。任务分配均衡，同一进程中的多个线程不能保证同时调度，两次调度可能在不同的处理器上，局部数据失效。

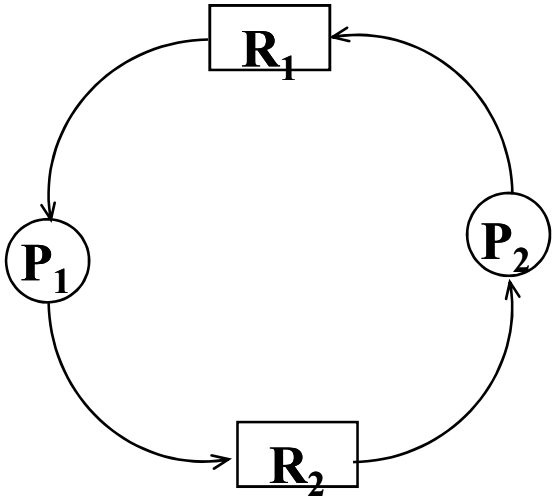
组调度：一组相关的线程同时分配到多个处理器上运行。

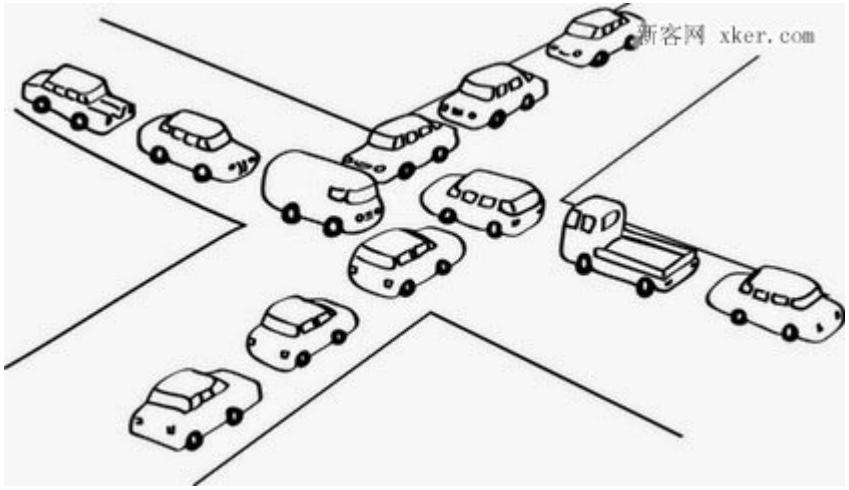
4.5 死锁

4.5.1 死锁的产生

1. 什么是死锁

死锁是指一组并发进程彼此互相等待对方所拥有的资源，且这些并发进程在得到对方的资源之前不会释放自己所拥有的资源，从而使并发进程不能继续向前推进的状态。陷入死锁状态的进程称之为死锁进程。





2. 产生死锁的原因：系统资源不足，进程推进顺序不恰当

4.5.2 死锁的必要条件

（1）互斥条件。指进程竞争的资源具有互斥性，即在一段时间内某资源被一个进程占用，如果此时还有其它进程请求使用该资源，则只能等待，直到占用该资源的进程用完后主动释放。

（2）不可剥夺条件（不可抢占条件）。指已分配给某一进程的资源，在它未使用完之前，不能强行剥夺，只能在使用完后，由进程自己释放。

（3）部分分配条件（请求与保持条件）。指进程已经占用了一部分资源，但又提出新的资源请求，而该资源又被其它的进程所占用，此时请求进程只能阻塞，但又对自己占用的资源保持不放。

（4）环路条件（循环等待条件）。指进程发生死锁时，必然存在一个进程—资源的环形链。即一组进程P1，P2，…，Pn，其中，P1正在等待P2占用的资源；P2正在等待P3占用的资源，……，Pn正在等待P1占用的资源。图4—7所示为两个进程竞争两个资源的环形链图。

4.6 解决死锁的方法

4.6.1 死锁的预防

1．防止“部分分配”条件的出现

一次性分配

2．防止“环路等待”条件的出现

资源顺序使用法

4.6 解决死锁的方法

4.6.2 死锁的避免

1. 安全状态

存在安全序列（p2，p1，p3），系统处于安全状态。

进程	最大需求	已分配	可用资源
P_1	9	4	3
P_2	4	2	
P_3	12	3	

此时，系统进入不安全状态。

进程	最大需求	已分配	可用资源
P_1	**9**	**4**	**2**
P_2	**4**	**2**	
P_3	**12**	**4**	

2．银行家算法

（1）数据结构

银行家算法中用到下列数据结构，令n是系统中的进程数，m是资源类数。

①可用资源向量A（Available）。向量A的长度为m，向量元素A[j](j=1，2，…，m)为系统中资源类r_j的当前可用数。

②最大需求矩阵M（Max）。M是一个n×m的矩阵，矩阵元素M[i,j]为进程p_i关于资源类r_j的最大需求数，每个进程必须预先申报。

③资源占用矩阵U（Use）(Allocation)。U是一个n×m的矩阵，矩阵元素U[i,j]为进程p_i关于资源类r_j的当前占用数。

④剩余需求矩阵N（Need）。N是一个n×m的矩阵，矩阵元素N[Ii,j]是进程p_i还需要的资源类r_j的单位数。显然有，N[Ii,j]＝M[i,j]－U[i,j]。

（2）简记法

为了简化对算法的描述，对上述数据结构采用如下的简记法

①令X和Y为长度是m的向量，若X≤Y，当且仅当对任意的i（i＝1，2，…，m）有X[i]≤Y[i]。

②对于n×m的矩阵$Z_{n\times m}$，Z_i（i＝1，2，…，n）表示矩阵$Z_{n\times m}$的第i个行向量。

（3）算法描述

令RR_i是长度为m的进程p_i的资源请求向量，元素RR[i,j]是进程p_i希望请求分配的资源类r_j的单位数。当进程p_i向系统提交一个资源请求向量RR_i时，系统调用银行家算法执行下述工作：

①若$RR_i > N_i$，则有（$RRi + U_i$）$> M_i$，即进程p_i请求的资源单位数大于它申请的最大需求数，故请求无效，作出错处理；否则进行下一步。

②若$RR_i > A$，则进程p_i必须等待，即系统当前没有足够的资源满足进程p_i当前的请求；否则进行下一步；

③系统进行假分配，即假设系统给进程p_i分配所请求的资源，对资源分配状态作如下修改：

$$A = A - RR_i;$$
$$U_i = U_i + RR_i;$$
$$N_i = Ni - RR_i;$$

④调用安全算法检查此次资源分配后的现行状态是否安全状态。若安全，则正式将资源分配给进程p_i，完成进程p_i的资源请求分配工作。否则，拒绝分配让进程等待，并恢复此次的假设分配，即撤消步骤③对分配状态所作的修改。

（4）安全算法描述

①设向量W（Work），向量元素W[j](j=1，2，…，m)表示系统可供给各个进程继续运行的j类资源数；向量F（Finish），向量元素F[i](i =1，2，…，n)表示系统是否有足够的资源可使进程pi完成。初始化W＝A，F[i]=false。

②从进程集合中找到一个进程p_i，有F[i]=false且Ni≤W，则执行步骤③；如果这样的进程不存在，则转去执行步骤④；

③进程p_i可得到所需的全部资源，顺利执行完成，并释放它所占用的资源，所以执行W＝W＋U_i及F[i]=true，转去执行②；

④若对所有的进程，都有F[i]=true，则存在一个安全序列，现行状态是安全的，否则是不安全的。

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 (2	3 3	2 0)
P_1	3	2	2	2 (3	0 0	0 2)	1 (0	2 2	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			

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(见图 3-17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	Work			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3-17　T0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下图中的圆括号所示。

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 (2	3	2 0)
P_1	3	2	2	2 (3	0 0	0 2)	1 (0	2 2	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。如图所示。

资源 / 情况 / 进程	Work			Need			Allocation			Work+Allocation			Finish
	A	B	C	A	B	C	A	B	C	A	B	C	
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

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 $Request_4(3，3，0) \geq Available(2，3，0)$，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 $Request_0(0，2，0) \leq Need_0(7，4，3)$;

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，如图所示。

资源 / 情况 / 进　程	Allocation			Need			Available		
	A	B	C	A	B	C	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			

为P_0分配资源后的有关资源数据

（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。

如果在银行家算法中，把P0发出的请求向量改为$Request_0$(0，1，0)，系统是否能将资源分配给它？

例4－10：

$$A=(1,5,2,0)$$

$$U_{5\times4}=\begin{bmatrix}0012\\1000\\1354\\0632\\0014\end{bmatrix}$$

$$N_{5\times4}=\begin{bmatrix}0000\\0750\\1002\\0020\\1642\end{bmatrix}$$

解：利用银行家算法进行检查：

(1) RR2≦N2，即（0,4,2,0）≦（0,7,5,0），继续下一步。

(2) RR2≦A，即（0,4,2,0）≦（1,5,2,0），继续下一步。

(3) 进行假分配

$$A = (1,1,0,0) \qquad U_{5\times4} = \begin{bmatrix} 0012 \\ 1420 \\ 1354 \\ 0632 \\ 0014 \end{bmatrix} \qquad N_{5\times4} = \begin{bmatrix} 0000 \\ 0330 \\ 1002 \\ 0020 \\ 1642 \end{bmatrix}$$

(4) 执行安全算法

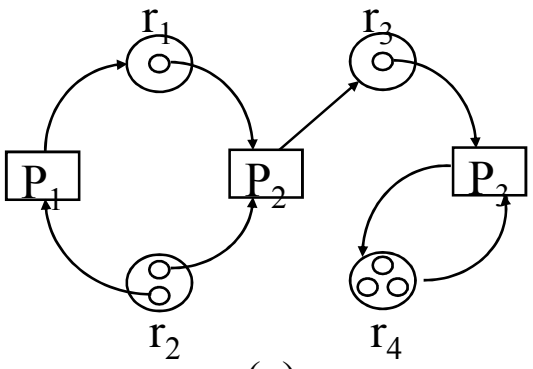
对所有的进程，都有F[i]=true，得到一个安全序列（P1、 P3、 P2、P4 、P5），故状态是安全的。（注意：安全序列不是唯一的）

RR5≦（1,0,0,0）?

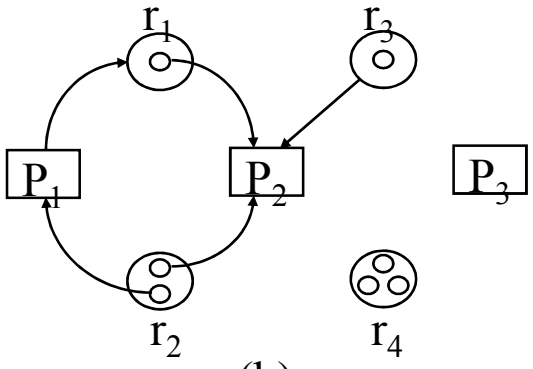
银行家算法虽然能有效的避免死锁的发生，但是也存在一些缺点。如它要求系统中被分配的每类资源固定；用户进程数目保持固定不变；要求用户事先说明他们的最大资源需求；同时每次进行资源分配前都要进行安全性检查，花费处理机时间等。

4.6.3 死锁的检测与解除

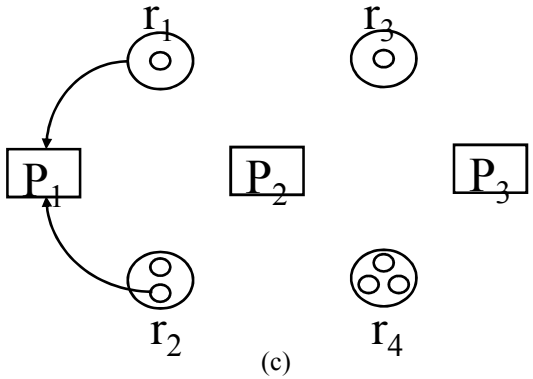
1．资源分配图RAG



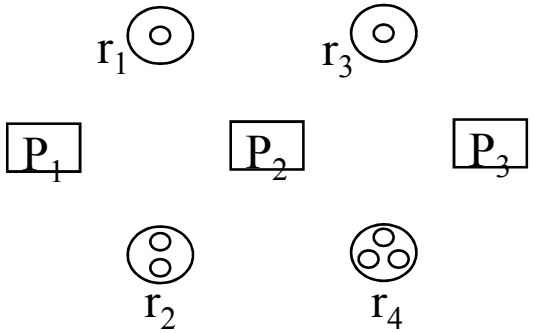
(a)



(b)



(c)



(d)

2．死锁的检测

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。该充分条件被称为死锁定理。

3．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是：

（1）剥夺资源。从其它进程剥夺足够数量的资源给死锁进程，以解除死锁状态。

（2）撤消进程。最简单的撤消进程的方法是使全部死锁进程都夭折掉；稍微温和一点的方法是按照某种顺序逐个地撤消进程，直至有足够的资源可用，使死锁状态消除为止。

4.7 Linux进程调度

4.7.1 调度的时机

(1) 当前正在CPU 执行的进程结束,或因某种原因阻塞睡眠。

(2) 当就绪队列中增加一个新进程时,need_resched置为1。当内核校验到调度标志为1时便重新调度。

(3) 当正在执行的进程分到的时间片用完时,调度标志need_resched置为1,要重新调度。

(4) 当进程从执行系统调用返回到用户态时,要检测调度need_resched状态,若是1,则启动调度程序。

(5) 当内核结束中断处理返回用户态时,要重新调度。

(6) 直接执行调度程序。

4.7.2 调度策略

(1) SCHED_FIFO(先到先服务)。对于所有相同优先级的进程,最先进入运行队列的进程总能优先获得调度;进程一旦占用CPU 则一直运行,直到有更高优先级的任务到达或自己放弃。

(2) SCHED_RR(时间片轮转)。该策略采用更加公平的轮转策略,当进程的时间片用完时,系统将重新分配时间片,并置于就绪队列尾。放在队列尾保证了所有具有相同优先级的RR任务的调度公平。

(3) SCHED_NORMAL(普通进程调度策略,在Linux2.6 内核以前为SCHED_OTHER)。

4.7.3 调度算法

1.分时动态优先级的调度算法

每个进程在创建时都被赋予一个时间片counter。调度程序调用goodness()函数遍历就绪队列中的进程,计算每个进程的动态优先级(counter+20-nice),选择计算结果最大的一个去运行。时钟中断递减当前运行进程的时间片,当这个时间片用完后(counter减至0)或者主动放弃CPU 时,该进程将被放在就绪队列TASK_RUNNING 末尾。

调度程序选择进程时需要遍历整个TASK_RUNNING 队列,从中选出优先执行的进程,因此该算法的执行时间与进程数成正比。另外,每次重新计算counter所花费的时间也
会随着系统中进程数的增加而线性增长,当进程数很大时,更新counter操作的代价会非常高,导致系统整体的性能下降。

2. O(1)的调度算法

O(1)算法中将可运行态(TASK_RUNNING)进程分为两类:一类是活动进程，即那些还没有用完时间片的进程；另一类是过期进程,即那些已经用完时间片的进程,在其他进程没有用完自己的时间片之前，过期进程不能再被运行。系统一共有140个不同的优先级，因此两类进程各有140个不同优先级的进程链表。各个队列还采用优先级位示图来标记每个链表中是否存在进程。调度执行的进程都会被按照先进先出的顺序添加到各自的链表末尾。每个进程都有一个时间片,这取决于系统允许执行这个进程多长时间。

O(1)调度程序区分交互式进程和批处理进程的算法与以前相比虽大有改进,但仍然在很多情况下会失效。有一些著名的程序总能让该调度程序性能下降,导致交互式进程反应缓慢,对于NUMA(NonUniform MemoryAccessArchitecture,非统一内存的多处理器架构)支持也不完善。该算法的复杂性主要来自动态优先级的计算,调度程序根据平均睡眠时间和一些很难理解的经验公式来修正进程的优先级以及区分交互式进程,这样的代码很难阅读和维护

3. 完全公平调度算法

分配给进程的运行时间= 调度周期× 进程权重/所有进程权重之和

可以看到,进程的权重越大,分到的运行时间越多。

为确保每个进程只在公平分配给它的处理机时间内运行，CFS中引入了虚拟运行时间(vruntime)的概念。vruntime记录了一个可执行进程到当前时刻为止执行的总时间。

vruntime+= 当前进程的运行时间×NICE_0_LOAD/进程权重

其中NICE_0_LOAD是一个定值，为系统默认的进程的权值。

调度算法每次选择vruntime值最小的进程进行调度，内核中使用红黑树可以方便地得到vruntime值最小的进程。

系统定时器周期性地计算当前进程的执行时间。时钟周期中断函数主要是更新当前进程的vruntime值和实际运行时间值，并判断当前进程在本次调度中的实际运行时间是否超过了调度周期分配的实际运行时间，如果是则设置重新调度标志。

练习题

1. 某车站售票厅，任何时刻最多可容纳 **20** 名购票者进入，当售票厅中少于 **20** 名购票者时，则厅外的购票者可立即进入，否则需在外面等待。若把一个购票者看作一个进程，请回答下列问题：

（**1**）用 **PV** 操作管理这些并发进程时，应怎样定义信号量，写出信号量的初值以及信号量各种取值的含义。

（**2**）根据所定义的信号量，写出应执行的 **PV** 操作。

2. 某寺庙，有小，老和尚若干，由小和尚提水入缸供老和尚饮用。水缸可容**10**桶水，水取自同一井中。水井窄，每次只能容一个桶取水。水桶总数为**3**个。每次取，入缸水仅为**1**桶，且不可同时进行。试给出有关取井水入缸水，取缸水的算法。

3.考虑一个理发店，只有一个理发师，只有**n**张可供顾客等待理发的椅子，如果没有顾客，则理发师睡觉；如果有一顾客进入理发店发现理发师在睡觉，则把他叫醒，写一个程序协调理发师和顾客之间的关系。

4.有一个阅览室，共有**100**个座位，读者进入时必须先在一张登记表上登记，该表为每一座位列一表目，包括座号和读者姓名等，读者离开时要消掉登记的信息，试问：

（**1**）为描述读者的动作，应编写几个程序，设置几个进程？

（**2**）试用**PV**操作描述读者进程之间的同步关系。

5.设某计算机系统有一台输入机、一台打印机。现有两道程序同时投入运行，且程序A先开始运行，程序B后运行。程序A的运行轨迹为：计算50MS，打印信息100MS，再计算50MS，打印信息100MS，结束。程序B的运行轨迹为计算50MS，输入数据80MS，再计算100MS。试说明：

（1）两道程序运行时，CPU有无空闲等待？若有，在哪段时间内等待？为什么会空闲等待？

（2）程序A、B运行时有无等待现象？若有，在什么时候会发生等待现象？

6.设有一组作业，它们的提交时间及运行时间如下所示：

作业号	提交时间	运行时间（分钟）
1	8:00	70
2	8:40	30
3	8:50	10
4	9:10	5

试问在单道方式下，采用响应比高者优先调度算法，作业的执行顺序是什么，周转时间，平均周转时间？

7.有一个具有两道作业的批处理系统，作业调度采用短作业优先的调度算法，进程调度采用以优先数为基础的抢占式调度算法。在下表所示的作业序列，作业优先数即为进程优先数，优先数越小优先级越高。

作业名	到达时间	估计运行时间	优先数
A	10:00	40分	5
B	10:20	30分	3
C	10:30	50分	4
D	10:50	20分	6

(1)列出所有作业进入内存时间以及结束时间。

(2)计算平均周转时间。

8.设公交车上，为保证安全，司机停车后售票员才能开门，售票员关车门后司机才能行车

司机的活动：启动车辆，正常行车，到站停车

售票员活动：关车门，售票，开车门，

用信号量和**P-V**操作实现它的同步。