第三章　处理机调度与死锁

3.1 处理机调度的层次

- 调度层次：

高级调度（作业调度，长程调度，Long-term）
低级调度（进程调度，短程调度，Short-term）
中级调度（交换调度，中程调度，Medium-term）

- 三种调度程序的比较

调度程序	调度对象	分配资源	调度频率	适宜的调度算法
长程调度（作业）	后备作业	内存、设备	低	资源搭配、先进先出、优先数、短作业优先、最高相应比优先
中程调度（交换）	就绪进程 等待进程	内存、外存	中	优先数法，FIFO
短程调度（进程）	就绪进程、 就绪线程	CPU	高	轮转法，多级反馈

3.1.1 高级调度

1．作业和作业步

(1) 作业(Job)。作业是一个比程序更为广泛的概念，它不仅包含了通常的程序和数据，而且还应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。在批处理系统中，是以作业为基本单位从外存调入内存的。

作业 = JCB+作业说明书 + 程序 + 数据

3.1.1 高级调度

(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，我们把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。例如，一个典型的作业可分成三个作业步：

① “编译”作业步，通过执行编译程序对源程序进行编译，产生若干个目标程序段；

② “连结装配”作业步，将“编译”作业步所产生的若干个目标程序段装配成可执行的目标程序；

③ “运行”作业步，将可执行的目标程序读入内存并控制其运行。

(3) 作业流。若干个作业进入系统后，被依次存放在外存上，这便形成了输入的作业流；在操作系统的控制下，逐个作业进行处理，于是便形成了处理作业流。

3.1.1 高级调度

2．作业控制块JCB(Job Control Block)

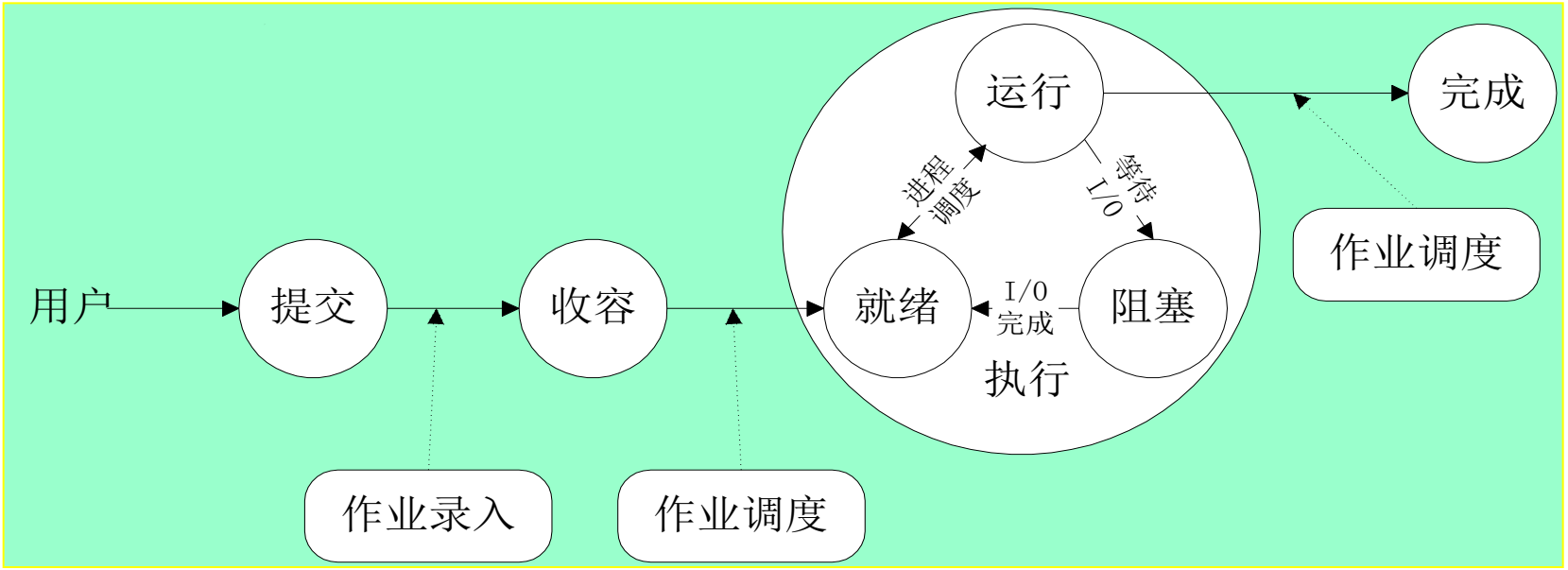
为了管理和调度作业，在多道批处理系统中为每个作业设置了一个作业控制块，它是作业在系统中存在的标志，其中保存了系统对作业进行管理和调度所需的全部信息。

在JCB中所包含的内容因系统而异，通常应包含的内容有：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、资源使用情况等。

3.1.1 高级调度

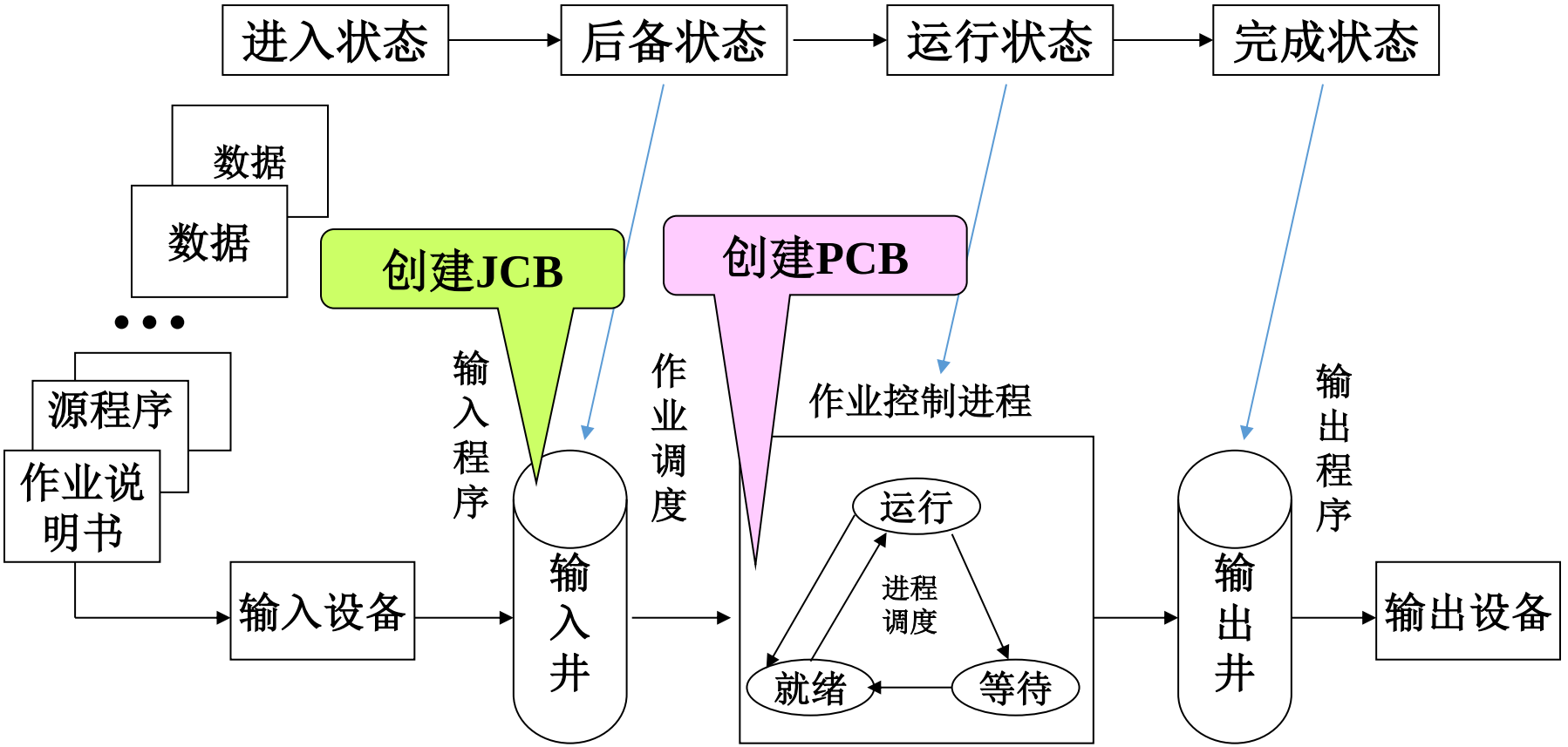
每当作业进入系统时，系统便为每个作业建立一个JCB，根据作业类型将它插入相应的后备队列中。作业调度程序依据一定的调度算法来调度它们，被调度到的作业将会装入内存。在作业运行期间，系统就按照JCB中的信息对作业进行控制。当一个作业执行结束进入完成状态时，系统负责回收分配给它的资源，撤消它的作业控制块。

作业状态



- 作业提交：作业的输入（从输入设备到外存）；
- 作业收容（就绪）：作业输入（到外存）完成，系统为其建立JCB，等待调度运行；
- 作业执行：分配资源，送入内存，被调度运行；
- 作业完成：释放资源，完成作业输出；

批处理作业的状态及转换



作业和进程的状态转换图

3.1.1 高级调度

3．作业调度

作业调度的主要功能是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，以及按照一定的算法，从外存的后备队列中选取某些作业调入内存，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插入就绪队列，准备执行。

3.1.1 高级调度

对用户而言，总希望自己作业的周转时间尽可能的少，最好周转时间就等于作业的执行时间。然而对系统来说，则希望作业的平均周转时间尽可能少，有利于提高CPU 的利用率和系统的吞吐量。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。

周转时间 = 完成时间 - 提交时间
= 等待时间 + 运行时间

3.1.1 高级调度

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。

1. 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，比如，使周转时间太长。
2. 但如果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，因此，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。

3.1.1 高级调度

2) 决定接纳哪些作业

- 调度算法：应将哪些作业从外存调入内存。

① 先来先服务调度算法：最早进入外存的作业最先调入内存；

② 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；

③ “响应比高者优先”的调度算法：综合考虑作业的大小与等待时间。

④ 基于作业优先级的调度算法：将外存上优先级最高的作业优先调入内存；

3.1.2 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(Short Term 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。

1．低级调度的功能

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，然后再由分派程序执行把处理机分配给该进程的具体操作。

3.1.2 低级调度

- 低级调度功能：

(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。

3.1.2 低级调度

2．进程调度中的三个基本机制

(1) 排队器。为了提高进程调度的效率，应事先将系统中所有的就绪进程按照一定的方式排成一个或多个队列，以便调度程序能最快地找到它。

(2) 分派器(分派程序)。分派器把由进程调度程序所选定的进程，从就绪队列中取出该进程，然后进行上下文切换，将处理机分配给它

(3) 上下文切换机制。两对上下文切换操作，在第一对上下文切换时，操作系统将保存当前进程的上下文，而装入分派程序的上下文，以便分派程序运行；在第二对上下文切换时，将移出分派程序，而把新选进程的CPU现场信息装入到处理机的各个相应寄存器中。

3.1.2 低级调度

- 切换开销：上下文切换花去的处理机时间。

即使是现代计算机，每一次上下文切换大约需要花费几毫秒的时间，该时间大约可执行上千条指令。为此，现在已有通过硬件(采用两组或多组寄存器)的方法来减少上下文切换的时间。一组寄存器供处理机在系统态时使用，另一组寄存器供应用程序使用。在这种条件下的上下文切换只需改变指针，使其指向当前寄存器组即可。

3.1.2 低级调度

3．进程调度方式

进程调度可采用两种调度方式（抢占与非强占）

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。

3.1.2 低级调度

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如下几个：

(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。

3.1.2 低级调度

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。

抢占方式的优点是，可以防止一个长进程长时间占用处理机，能为大多数进程提供更公平的服务，特别是能满足对响应时间有着较严格要求的实时任务的需求。抢占调度方式是基于一定原则的，主要有如下几条：

3.1.2 低级调度

(1) 优先权原则。通常是对一些重要的和紧急的作业赋予较高的优先权。当这种作业到达时，如果其优先权比正在执行进程的优先权高，便停止正在执行(当前)的进程，将处理机分配给优先权高的新到的进程，使之执行；或者说，允许优先权高的新到进程抢占当前进程的处理机。

(2) 短作业(进程)优先原则。当新到达的作业(进程)比正在执行的作业(进程)明显的短时，将暂停当前长作业(进程)的执行，将处理机分配给新到的短作业(进程)，使之优先执行； 或者说，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。

3.1.3 中级调度

- 中级调度：选择在外存上的那些具备运行条件的就绪进程，将它们重新调入内存，并修改其状态为就绪状态，挂在就绪队列上等待进程调度 。
- 中级调度实际上就是存储器管理中的对换功能。
- 引入目的：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，再将其调入内存。

3.2 调度队列模型和调度准则

3.2.1 调度队列模型

- 仅有进程调度的调度队列模型
- 具有高级和低级调度的调度队列模型
- 同时具有三级调度的调度队列模型

3.2.1 调度队列模型

1. 仅有进程调度的调度队列模型

- 系统可以把处于就绪状态的进程组织成栈、树或一个无序链表。例如，在分时系统中，常把就绪进程组织成FIFO队列形式。每当OS创建一个新进程时，便将它挂在就绪队列的末尾，然后按时间片轮转方式运行。
- 每个进程在执行时都可能出现以下三种情况：

(1) 任务在给定的时间片内已经完成，该进程便在释放处理机后进入完成状态；

(2) 任务在本次分得的时间片内尚未完成，OS便将该任务再放入就绪队列的末尾；

(3) 在执行期间，进程因为某事件而被阻塞后，被OS放入阻塞队列。

3.2.1 调度队列模型

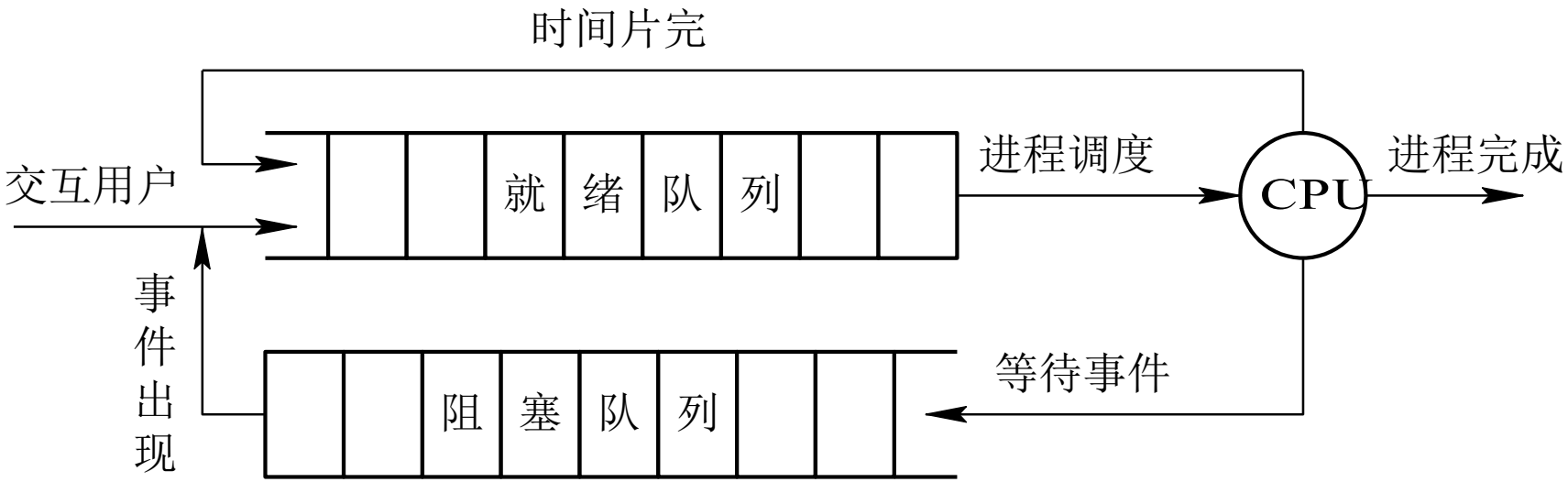


图 3-1 仅具有进程调度的调度队列模型

3.2.1 调度队列模型

2．具有高级和低级调度的调度队列模型

在批处理系统中，不仅需要进程调度，而且还需有作业调度，由后者按一定的作业调度算法，从外存的后备队列中选择一批作业调入内存，并为它们建立进程，送入就绪队列，然后才由进程调度按照一定的进程调度算法选择一个进程，把处理机分配给该进程。图3-2示出了具有高、低两级调度的调度队列模型。

3.2.1 调度队列模型

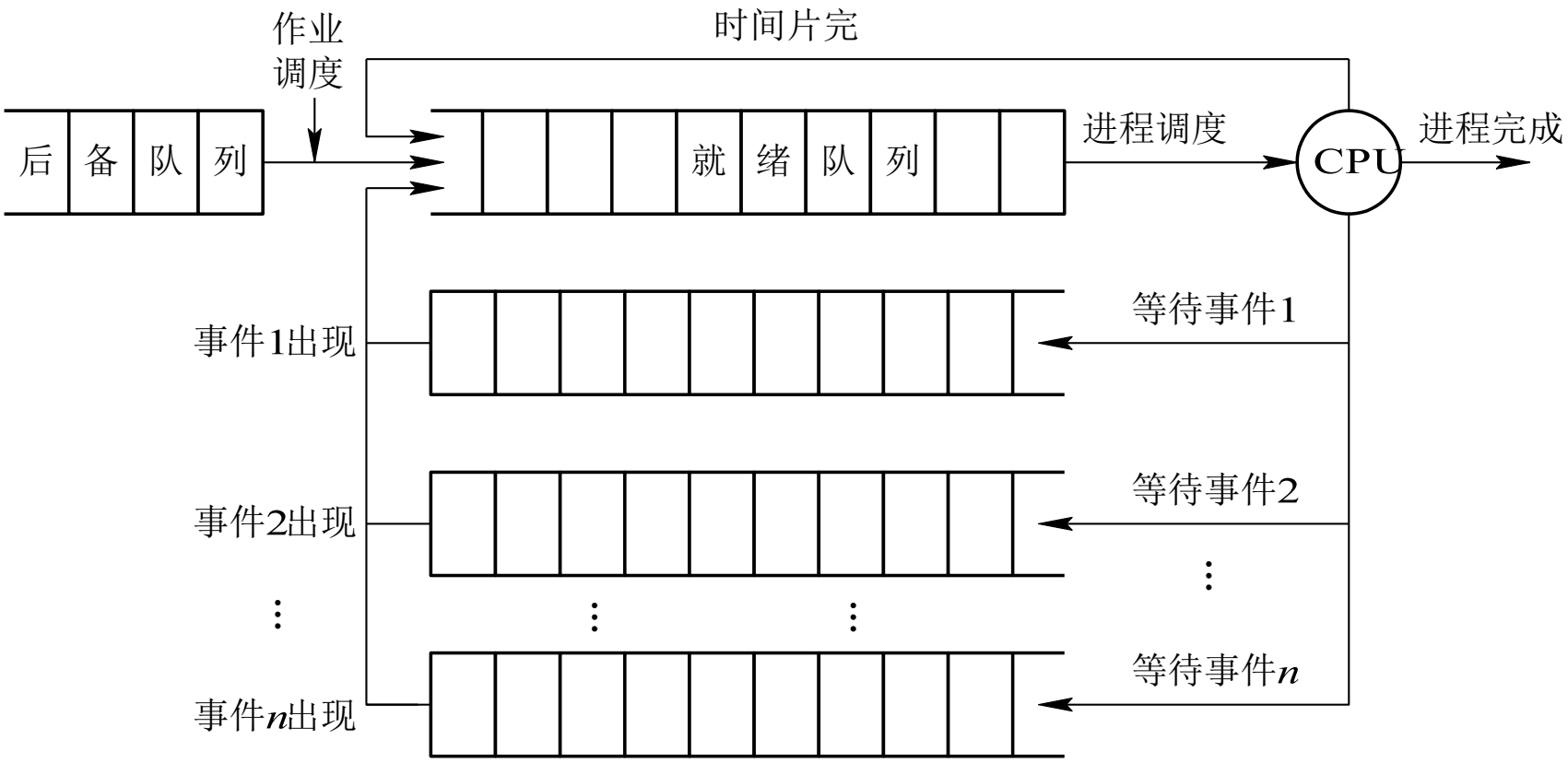


图 3-2 具有高、低两级调度的调度队列模型

3.2.1 调度队列模型

(1) 就绪队列的形式。在批处理系统中，最常用的是最高优先权优先调度算法，相应地，最常用的就绪队列形式是优先权队列。

- 进程在进入优先级队列时，根据其优先权的高低，被插入具有相应优先权的位置上，这样，调度程序总是把处理机分配给就绪队列中的队首进程。
- 无序链表方式，即每次把新到的进程挂在链尾，而调度程序每次调度时，是依次比较该链中各进程的优先权，从中找出优先权最高的进程，将之从链中摘下，并把处理机分配给它。显然，无序链表方式与优先权队列相比，这种方式的调度效率较低。

3.2.1 调度队列模型

(2) 设置多个阻塞队列。对于小型系统，可以只设置一个阻塞队列；但当系统较大时，若仍只有一个阻塞队列，其长度必然会很长，队列中的进程数可以达到数百个，这将严重影响对阻塞队列操作的效率。故在大、中型系统中通常都设置了若干个阻塞队列，每个队列对应于某一种进程阻塞事件。

3.2.1 调度队列模型

3．同时具有三级调度的调度队列模型

当在OS中引入中级调度后，人们可把进程的就绪状态分为内存就绪(表示进程在内存中就绪)和外存就绪(进程在外存中就绪)。类似地，也可把阻塞状态进一步分成内存阻塞和外存阻塞两种状态。

在调出操作的作用下，可使进程状态由内存就绪转为外存就绪，由内存阻塞转为外存阻塞；在中级调度的作用下，又可使外存就绪转为内存就绪。图3-3示出了具有三级调度的调度队列模型。

3.2.1 调度队列模型

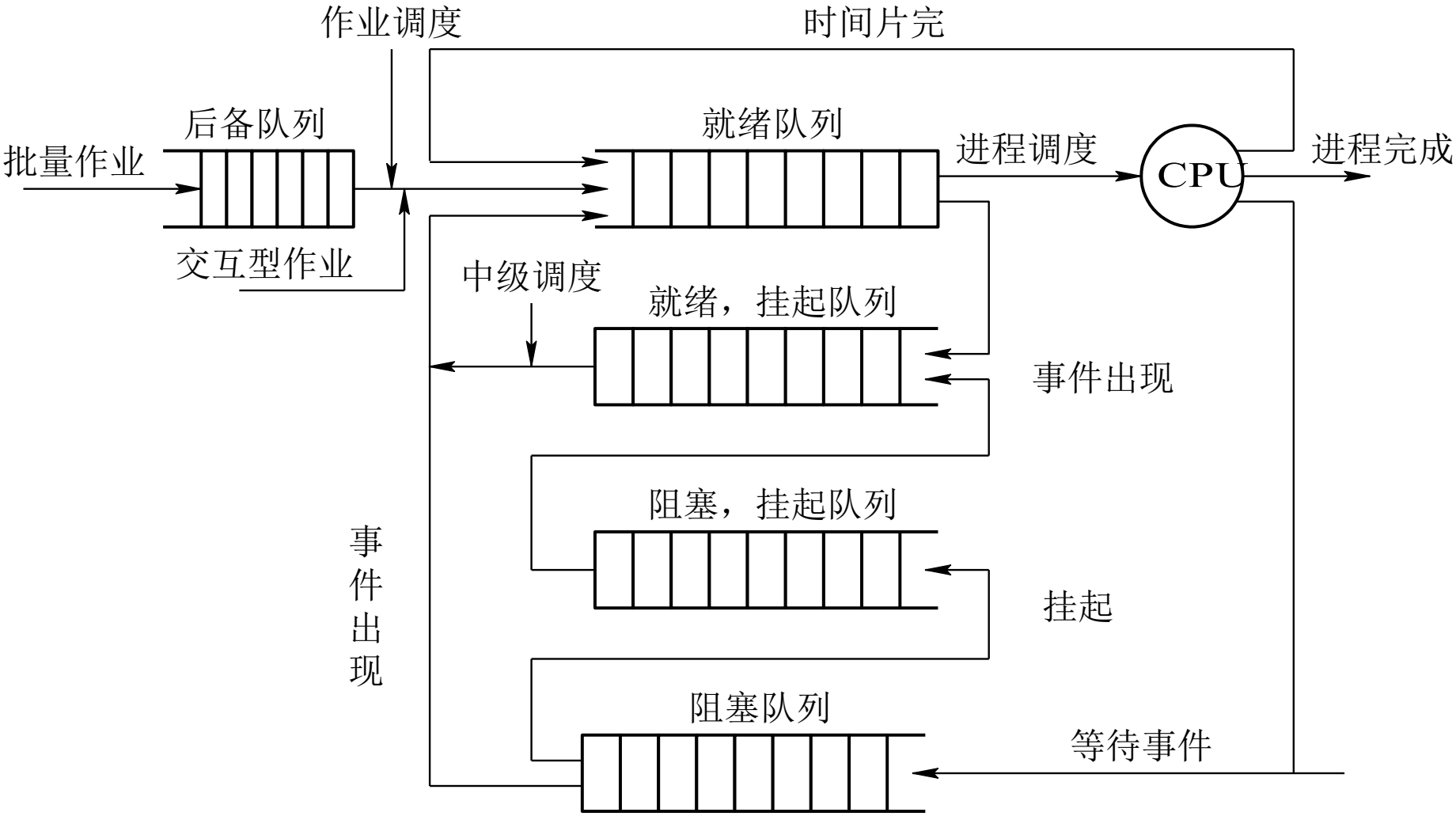


图 3-3 具有三级调度时的调度队列模型

3.2.2 选择调度方式和调度算法的若干准则

1．面向用户的准则

- 周转时间短
- 响应时间快
- 截止时间的保证
- 优先权准则

2．面向系统的准则

- 系统吞吐量高
- 处理机利用率好
- 各类资源的平衡利用

3.2.2 选择调度方式和调度算法的若干准则

1．面向用户的准则

(1) 周转时间短。通常把周转时间的长短作为评价批处理系统的性能、选择作业调度方式与算法的重要准则之一。所谓周转时间，是指从作业被提交给系统开始，到作业完成为止的这段时间间隔(称为作业周转时间)。它包括四部分时间：作业在外存后备队列上等待(作业)调度的时间，进程在就绪队列上等待进程调度的时间，进程在CPU上执行的时间，以及进程等待I/O操作完成的时间。其中的后三项在一个作业的整个处理过程中可能会发生多次。

3.2.2 选择调度方式和调度算法的若干准则

- 平均周转时间：

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

对每个用户而言，都希望自己作业的周转时间最短。但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，这不仅会有效地提高系统资源的利用率，而且还可使大多数用户都感到满意。

3.2.2 选择调度方式和调度算法的若干准则

- 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Ts之比，即W = T/Ts，可表示为：

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

3.2.2 选择调度方式和调度算法的若干准则

(2) 响应时间快。常把响应时间的长短用来评价分时系统的性能，这是选择分时系统中进程调度算法的重要准则之一。所谓响应时间，是从用户通过键盘提交一个请求开始，直至系统首次产生响应为止的时间，或者说，直到屏幕上显示出结果为止的一段时间间隔。

(3) 截止时间的保证。这是评价实时系统性能的重要指标，因而是选择实时调度算法的重要准则。所谓截止时间，是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，否则将可能造成难以预料的后果。

(4) 优先权准则。在批处理、分时和实时系统中选择调度算法时，都可遵循优先权准则，以便让某些紧急的作业能得到及时处理。在要求较严格的场合，往往还须选择抢占式调度方式，才能保证紧急作业得到及时处理。

3.2.2 选择调度方式和调度算法的若干准则

2．面向系统的准则

这是为了满足系统要求而应遵循的一些准则。其中，较重要的有以下几点：

(1) 系统吞吐量高。这是用于评价批处理系统性能的另一个重要指标，因而是选择批处理作业调度的重要准则。由于吞吐量是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具有密切关系。对于大型作业，一般吞吐量约为每小时一道作业；对于中、小型作业，其吞吐量则可能达到数十道作业之多。作业调度的方式和算法对吞吐量的大小也将产生较大影响。事实上，对于同一批作业，若采用了较好的调度方式和算法，则可显著地提高系统的吞吐量。

3.2.2 选择调度方式和调度算法的若干准则

(2) 处理机利用率高。对于大、中型多用户系统，由于CPU价格十分昂贵，致使处理机的利用率成为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。在大、中型系统中，在选择调度方式和算法时，应考虑到这一准则。但对于单用户微机或某些实时系统，则此准则就不那么重要了。

(3) 各类资源的平衡利用。在大、中型系统中，不仅要使处理机的利用率高，而且还应能有效地利用其它各类资源，如内存、外存和I/O设备等。选择适当的调度方式和算法可以保持系统中各类资源都处于忙碌状态。但对于微型机和某些实时系统而言，该准则并不重要。

3.3 调度算法

3.2.1 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。

3.3.1 先来先服务和短作业(进程)优先调度算法

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

3.3.1 先来先服务和短作业(进程)优先调度算法

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。

3.3.1 先来先服务和短作业(进程)优先调度算法

在此，我们通过一个例子来说明采用FCFS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。

3.3.1 先来先服务和短作业(进程)优先调度算法

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

图3-4 FCFS和SJF调度算法的性能

3.3.1　先来先服务和短作业(进程)优先调度算法

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。

3.3.1 先来先服务和短作业(进程)优先调度算法

为了和FCFS调度算法进行比较，我们仍利用FCFS算法中所使用的实例，并改用SJ(P)F算法重新调度，再进行性能分析。

由图3-4 中的(a)和(b)可以看出，采用SJ(P)F算法后，不论是平均周转时间还是平均带权周转时间，都有较明显的改善，尤其是对短作业D，其周转时间由原来的(用FCFS算法时)11降为3；而平均带权周转时间是从5.5降到1.5。这说明SJF调度算法能有效地降低作业的平均等待时间，提高系统吞吐量。

3.3.1 先来先服务和短作业(进程)优先调度算法

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。

3.3.2 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。

3.3.2 高优先权优先调度算法

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。

3.3.2 高优先权优先调度算法

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。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。

3.3.2 高优先权优先调度算法

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，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。

3.3.2 高优先权优先调度算法

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。

3.3.2 高优先权优先调度算法

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。

1. 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。
2. 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。

3.3.2 高优先权优先调度算法

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：

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

3.3.2 高优先权优先调度算法

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：

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

3.3.2 高优先权优先调度算法

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。

3.3.2 高优先权优先调度算法

(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。

3.3.3 基于时间片的轮转调度算法

1．时间片轮转法

1) 基本原理

在早期的时间片轮转法中，系统将所有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。这样就可以保证就绪队列中的所有进程在一给定的时间内均能获得一时间片的处理机执行时间。换言之，系统能在给定的时间内响应所有用户的请求。

3.3.3 基于时间片的轮转调度算法

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，如选择很小的时间片将有利于短作业，因为它能较快地完成，但会频繁地发生中断、进程上下文的切换，从而增加系统的开销；反之，如选择太长的时间片，使得每个进程都能在一个时间片内完成，时间片轮转算法便退化为FCFS算法，无法满足交互式用户的需求。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。

图3-5示出了时间片分别为q=1和q=4时，A、B、C、D、E五个进程的运行情况，而图3-6为q=1和q=4时各进程的平均周转时间和带权平均周转时间。图中的RR(Round Robin)表示轮转调度算法。

3.3.3 基于时间片的轮转调度算法

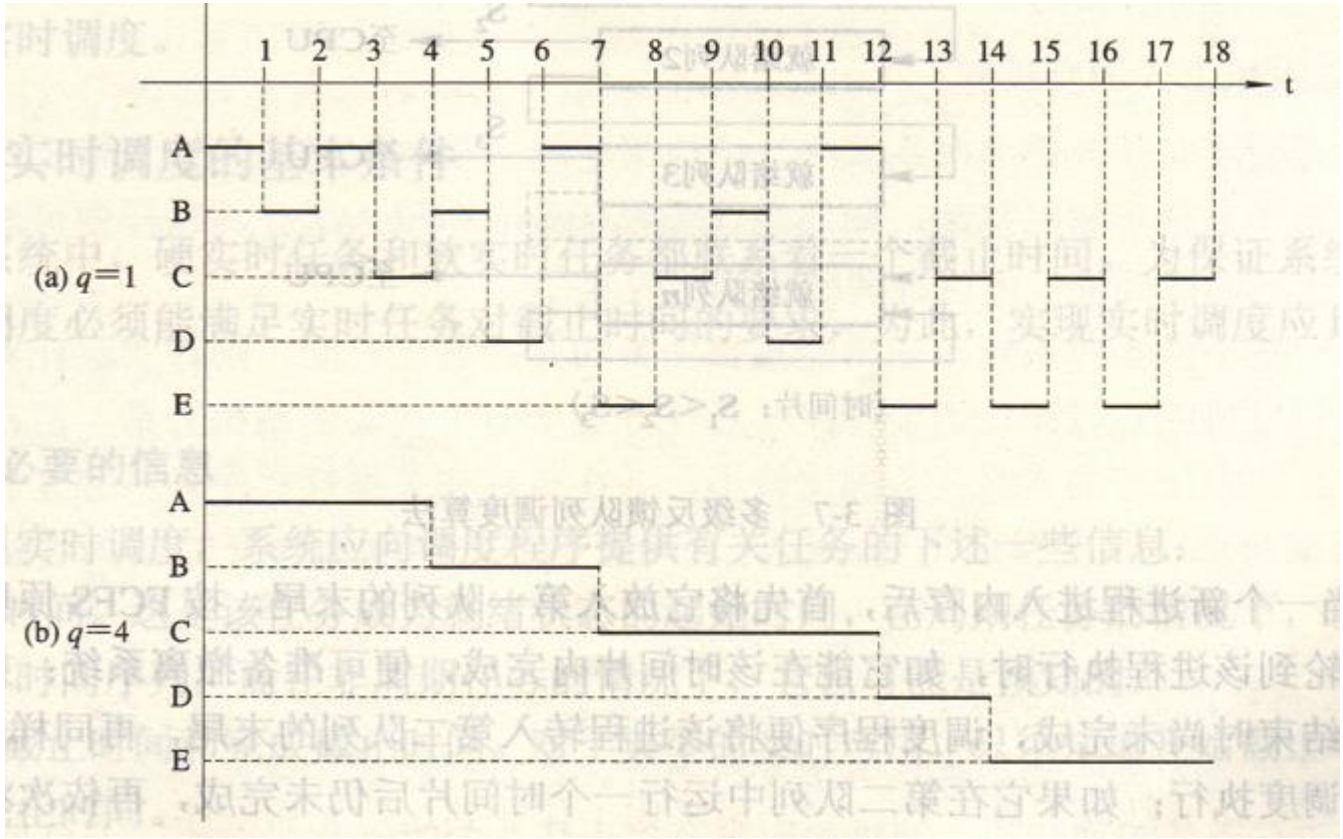


图3-5 q=1和q=4时的进程运行情况

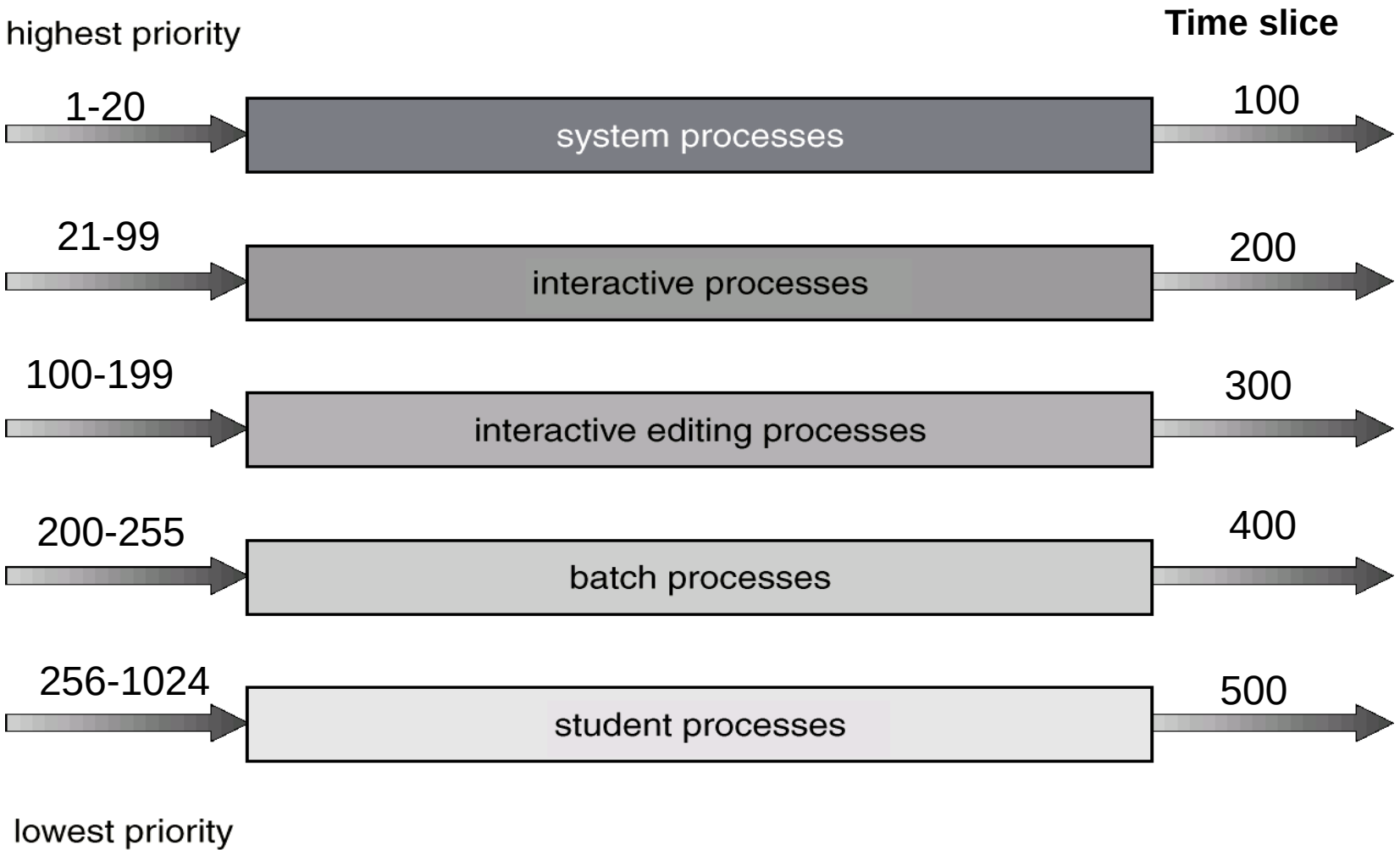
3.3.3 基于时间片的轮转调度算法

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	5	2	4	
RR q=1	完成时间	12	10	18	11	17	
	周转时间	12	9	16	8	13	11.6
	带权周转时间	3	3	3.2	4	3.25	3.29
RR q=4	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

图3-6 q=1和q=4时进程的周转时间

3.3.3 基于时间片的轮转调度算法

1.多级队列法例



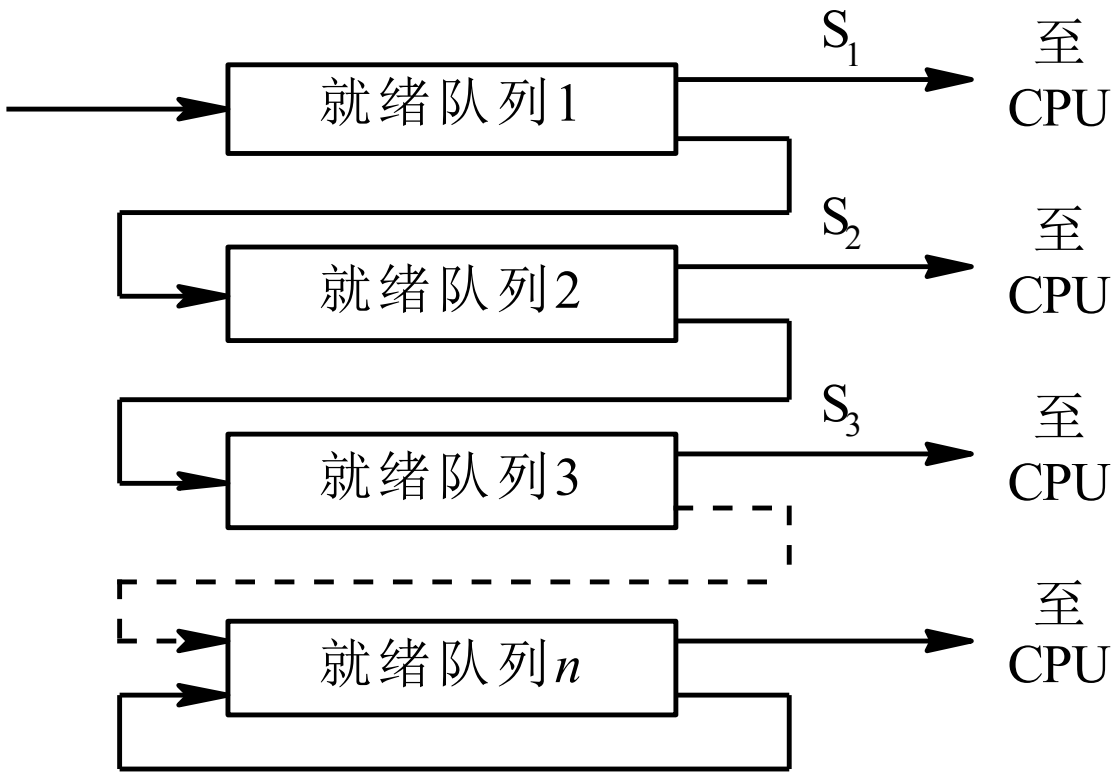
3.3.3 基于时间片的轮转调度算法

2．多级反馈队列调度算法

(1) 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3-7是多级反馈队列算法的示意。

3.3.3 基于时间片的轮转调度算法



（时间片：$S_1 < S_2 < S_3$）

图 3-7　多级反馈队列调度算法

3.3.3 基于时间片的轮转调度算法

(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。

3.3.3 基于时间片的轮转调度算法

(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。

3.3.3 基于时间片的轮转调度算法

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。

3.3.3 基于时间片的轮转调度算法

(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。

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

为了实现实时调度，系统应向调度程序提供有关任务的下述信息：

① 就绪时间。这是该任务成为就绪状态的起始时间。
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。
④ 资源要求。这是指任务执行时所需的一组资源。
⑤ 优先级。

3.4.1 实现实时调度的基本条件

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，系统才是可调度的。

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

3.4.1 实现实时调度的基本条件

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：

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

3.4.1　实现实时调度的基本条件

3．采用抢占式调度机制

在含有硬实时任务的实时系统中，广泛采用抢占机制。当一个优先权更高的任务到达时，允许将当前任务暂时挂起，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。

3.4.1 实现实时调度的基本条件

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，以保证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地小，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

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。

调度程序每次选择队列中的第一个任务投入运行。当该任务完成后，便把它挂在轮转队列的末尾，等待下次调度运行，而调度程序再选择下一个(队首)任务运行。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。

3.4.2 实时调度算法的分类

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。

3.4.2 实时调度算法的分类

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。

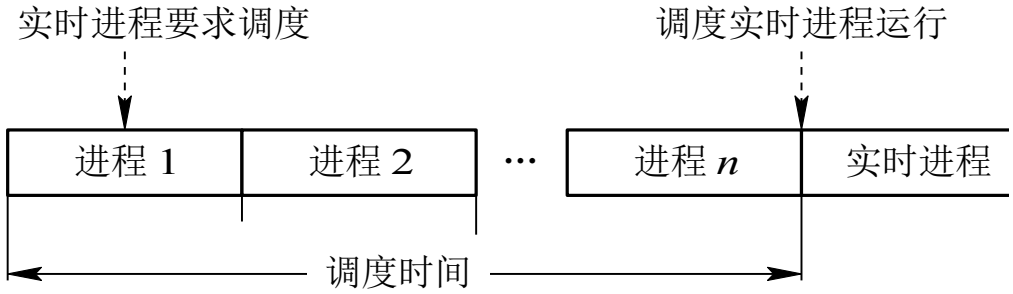
3.4.2 实时调度算法的分类

2) 立即抢占(Immediate Preemption)的优先权调度算法

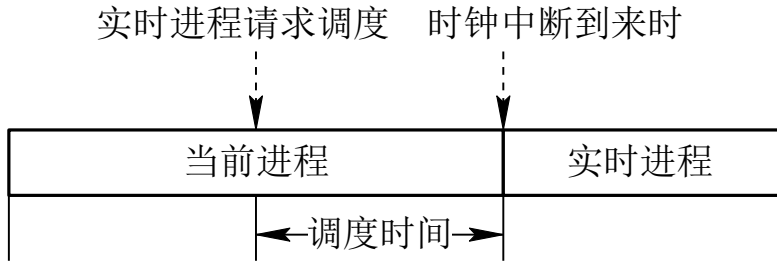
在这种调度策略中，要求操作系统具有快速响应外部事件中断的能力。一旦出现外部中断，只要当前任务未处于临界区，便立即剥夺当前任务的执行，把处理机分配给请求中断的紧迫任务。这种算法能获得非常快的响应，可把调度延迟降低到几毫秒至100微秒，甚至更低。

图3-8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。

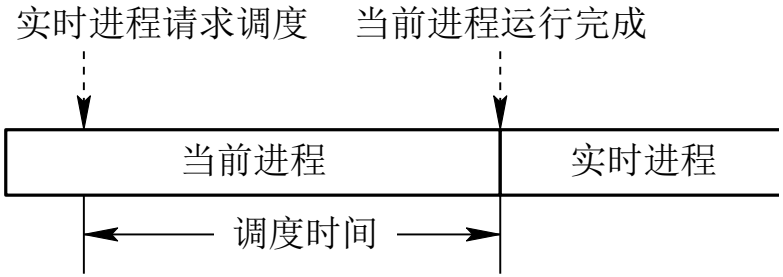
3.4.2 实时调度算法的分类



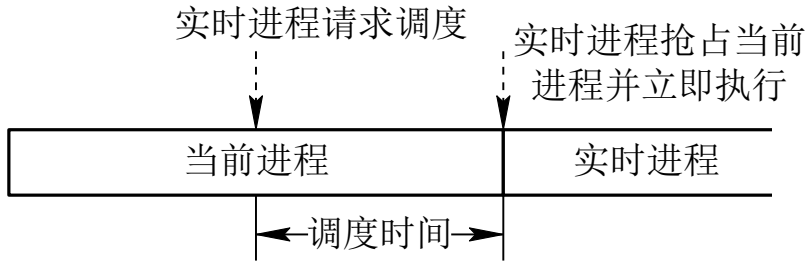
(a) 非抢占式轮转调度



(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

图3-8 实时进程调度

3.5 产生死锁的原因和必要条件

3.5.1 产生死锁的原因

产生死锁的原因可归结为如下两点：

(1) 竞争资源。当系统中供多个进程共享的资源如打印机、公用队列等，其数目不足以满足诸进程的需要时，会引起诸进程对资源的竞争而产生死锁。

(2) 进程间推进顺序非法。进程在运行过程中，请求和释放资源的顺序不当，也同样会导致产生进程死锁。

(3) P操作顺序不当

3.5.1 产生死锁的原因

1．竞争资源引起进程死锁

1) 可剥夺和非剥夺性资源

可把系统中的资源分成两类，一类是可剥夺性资源，是指某进程在获得这类资源后，该资源可以再被其他进程或系统剥夺。例如，优先权高的进程可以剥夺优先权低的进程的处理机。又如，内存区可由存储器管理程序把一个进程从一个存储区移到另一个存储区，此即剥夺了该进程原来占有的存储区。甚至可将一个进程从内存调出到外存上。

可见，CPU和主存均属于可剥夺性资源。另一类资源是不可剥夺性资源，当系统把这类资源分配给某进程后，再不能强行收回，只能在进程用完后自行释放，如磁带机、打印机等。

3.5.1 产生死锁的原因

2) 竞争非剥夺性资源

在系统中所配置的非剥夺性资源，由于它们的数量不能满足诸进程运行的需要，会使进程在运行过程中，因争夺这些资源而陷入僵局。

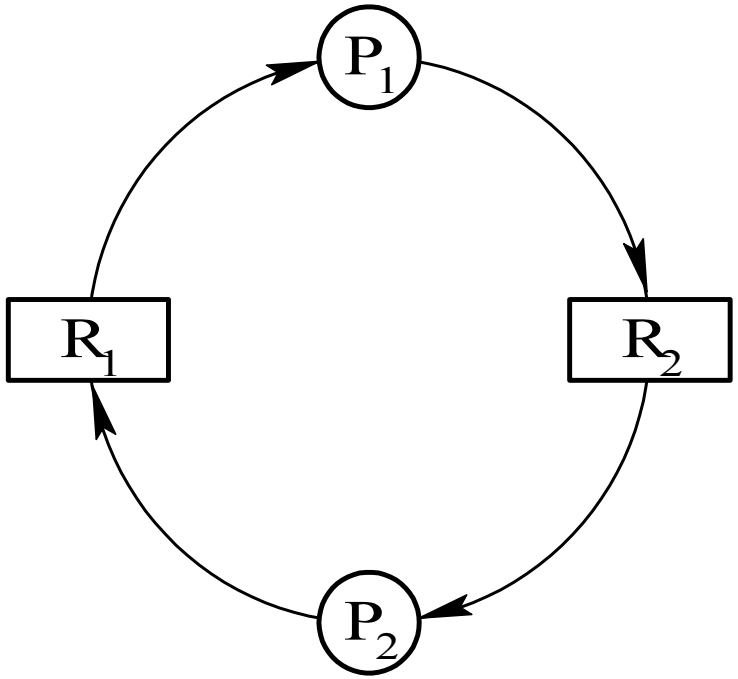


图3-13 I/O设备共享时的死锁情况

3.5.1 产生死锁的原因

2．进程推进顺序不当引起死锁

P_1：	P2：
Request(R_1)；	Request(R2)；
Request(R_2)；	Request(R1)；

3.5.1 产生死锁的原因

P操作顺序不当引起死锁

Producer process

```
repeat
    …
  produce an item in nextp;
  …
  P(empty);
  P(mutex);
   ….
  put nextp into buffer;
  ….
  V(mutex);
  V(full);
until false;
```

P(mutex);
P(empty);

Consumer process

```
repeat
  P(full)
  P(mutex);
  …
  remove an item from buffer to nextc
  …
  V(mutex);
  V(empty);
  …
  consume the item in nextc
  …
until false;
```

P(mutex);
P(full);

3.5.2 产生死锁的必要条件

虽然进程在运行过程中可能发生死锁，但死锁的发生也必须具备一定的条件,四个必要条件:

(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。

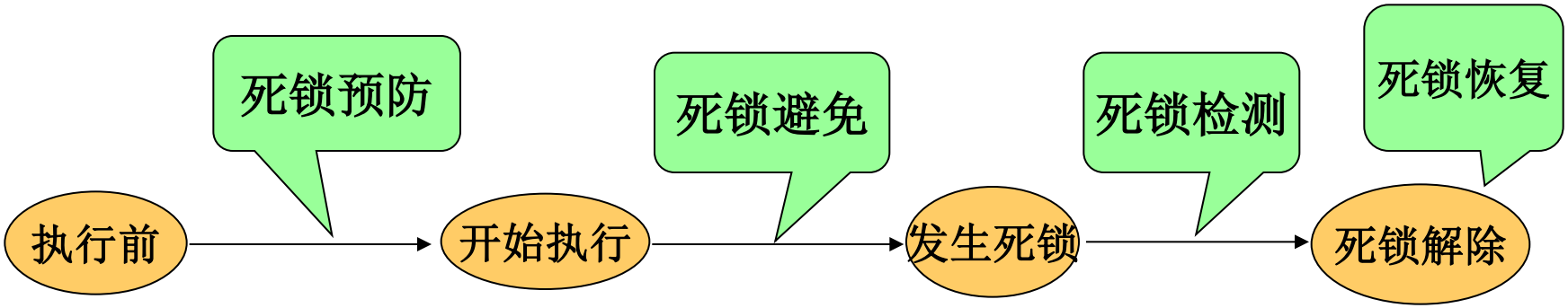
(2) 请求和保持条件：指进程已经保持了至少一个资源，但又提出了新的资源请求，而该资源又已被其它进程占有，此时请求进程阻塞，但又对自己已获得的其它资源保持不放。

3.5.2 产生死锁的必要条件

(3) 不剥夺条件：指进程已获得的资源，在未使用完之前，不能被剥夺，只能在使用完时由自己释放。

(4) 环路等待条件：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，…，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。

3.5.3 处理死锁的基本方法



3.5.3 处理死锁的基本方法

(1) 预防死锁。这是一种较简单和直观的事先预防的方法。该方法是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，来预防发生死锁。预防死锁是一种较易实现的方法，已被广泛使用。但由于所施加的限制条件往往太严格，因而可能会导致系统资源利用率和系统吞吐量降低。

(2) 避免死锁。该方法同样是属于事先预防的策略，但它并不须事先采取各种限制措施去破坏产生死锁的四个必要条件，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，从而避免发生死锁。这种方法只需事先施加较弱的限制条件，便可获得较高的资源利用率及系统吞吐量，但在实现上有一定的难度。

3.5.3 处理死锁的基本方法

(3) 检测死锁。这种方法并不须事先采取任何限制性措施，也不必检查系统是否已经进入不安全区，而是允许系统在运行过程中发生死锁。但可通过系统所设置的检测机构，及时地检测出死锁的发生，并精确地确定与死锁有关的进程和资源； 然后，采取适当措施，从系统中将已发生的死锁清除掉。

(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。常用的实施方法是撤消或挂起一些进程，以便回收一些资源，再将这些资源分配给已处于阻塞状态的进程，使之转为就绪状态，以继续运行。死锁的检测和解除措施有可能使系统获得较好的资源利用率和吞吐量，但在实现上难度也最大。

3.6 预防死锁的方法

3.6.1 预防死锁

死锁的预防一般是从破坏导致发生死锁的必要条件着手，只要能使四个必要条件其中的任何一个不成立，就可防止死锁。

1．摒弃“请求和保持”条件。动态分配＝》静态分配

系统规定所有进程在开始运行之前，都必须一次性地申请其在整个运行过程所需的全部资源。此时，若系统有足够的资源分配给某进程，便可把其需要的所有资源分配给该进程，这样，该进程在整个运行期间便不会再提出资源要求，从而摒弃了请求条件。

但在分配资源时，只要有一种资源不能满足某进程的要求，即使其它所需的各资源都空闲，也不分配给该进程，而让该进程等待。由于在该进程的等待期间，它并未占有任何资源，因而也摒弃了保持条件，从而可以避免发生死锁。

3.6.1 预防死锁

这种预防死锁的方法其优点是简单、易于实现且很安全。但其缺点却也极其明显：

1. 首先表现为资源被严重浪费，因为一个进程是一次性地获得其整个运行过程所需的全部资源的，且独占资源，其中可能有些资源很少使用，甚至在整个运行期间都未使用，这就严重地恶化了系统资源的利用率；
2. 其次是使进程延迟运行，仅当进程在获得了其所需的全部资源后，才能开始运行，但可能因有些资源已长期被其它进程占用而致使等待该资源的进程迟迟不能运行。

3.6.1 预防死锁

2．摒弃“不剥夺”条件

在采用这种方法时系统规定，进程是逐个地提出对资源的要求的。当一个已经保持了某些资源的进程，再提出新的资源请求而不能立即得到满足时，必须释放它已经保持了的所有资源，待以后需要时再重新申请。这意味着某一进程已经占有的资源，在运行过程中会被暂时地释放掉，也可认为是被剥夺了，从而摒弃了“不剥夺”条件。

这种预防死锁的方法实现起来比较复杂且要付出很大的代价。此外，这种策略还可能因为反复地申请和释放资源，致使进程的执行被无限地推迟，这不仅延长了进程的周转时间，而且也增加了系统开销，降低了系统吞吐量。

3.6.1 预防死锁

3．摒弃“环路等待”条件：资源有序分配

这种方法中规定，系统将所有资源按类型进行线性排队，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，因而摒弃了“环路等待”条件。

事实上，在采用这种策略时，总有一个进程占据了较高序号的资源，此后它继续申请的资源必然是空闲的，因而进程可以一直向前推进。

3.6.1 预防死锁

这种预防死锁的策略与前两种策略比较，其资源利用率和系统吞吐量都有较明显的改善。但也存在下述严重问题：

首先是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。

其次，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。例如，某进程先用磁带机，后用打印机，但按系统规定，该进程应先申请打印机而后申请磁带机，致使先获得的打印机被长时间闲置。

第三，为方便用户，系统对用户在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。

3.6.2 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。

3.6.2 系统安全状态

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。

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，便有可能进而进入死锁状态；反之，只要系统处于安全状态，系统便可避免进入死锁状态。因此，避免死锁的实质在于：系统在进行资源分配时，如何使系统不进入不安全状态。

3.6.2 系统安全状态

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：

3.6.2 系统安全状态

安全状态 安全序列：<P2，P1，P3>

进程号	总共需求	已分配	还需	剩余
P1	10	5	5	3
P2	4	2	2	
P3	9	2	7	

不安全状态 无安全序列

进程号	总共需求	已分配	还需	剩余
P1	10	5	5	2
P2	4	2	2	
P3	9	3	6	

3.6.3 利用银行家算法避免死锁

- 银行家算法

一个银行家把他的固定资金（capital）借给若干客户，使这些客户在满足对资金的要求后能完成其交易，把所借资金归还给银行家。假定客户分成若干次进行借款，每个客户预先说明自己所要求的最大资金量，并每次提出部分资金量申请和获得分配。如果银行家满足了客户对资金的最大需求量，则该客户在资金运作后，应在有限时间内将资金全部归还银行家。具体过程如下：

①顾客的借款操作依次顺序进行，直到全部操作完成；

②银行家对顾客的借款操作进行判断，确定其安全性（能否支持顾客借款，直到全部归还）；

③如果是安全的，则借款给顾客，否则暂不借款。

3.6.3 利用银行家算法避免死锁

银行家算法例

假定银行可以分期贷款10法郎给顾客，P,Q,R

顾客	目前已贷法郎	所需最大法郎
P	5	9
Q	2	5
R	2	3

当前还剩几个？当P申请一个法郎时，银行能满足他的要求吗？

R申请一个呢？

3.6.3 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×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。

3.6.3 利用银行家算法避免死锁

(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。

上述三个矩阵间存在下述关系：

Need[i, j]=Max[i, j]-Allocation[i, j]

3.6.3 利用银行家算法避免死锁

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。

3.6.3 利用银行家算法避免死锁

(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。

3.6.3 利用银行家算法避免死锁

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 进程完成向量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Finish[i]:=false；当有足够资源分配给进程时，再令Finish[i]:=true。

3.6.3 利用银行家算法避免死锁

(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]≤Work；若找到，执行步骤(3)，否则，执行步骤(4)。

(3) 当进程Pi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:= Work+Allocation[i]；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。

3.6.3 利用银行家算法避免死锁

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3-16所示。

3.6.3 利用银行家算法避免死锁

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			

图3-16 T_0时刻的资源分配表

3.6.3 利用银行家算法避免死锁

(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

3.6.3 利用银行家算法避免死锁

(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3-16中的圆括号所示。

3.6.3 利用银行家算法避免死锁

④ 再利用安全性算法检查此时系统是否安全。如图3-18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

图3-18 P_1申请资源时的安全性检查

3.6.3 利用银行家算法避免死锁

(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 $Request_4(3，3，0) \leq Available(2，3，0)$，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3-19所示。

3.6.3 利用银行家算法避免死锁

资源 / 情况 / 进程	Allocation			Need			Available		
	A	B	C	A	B	C	A	B	C
P_0	0	3	0	7	3	2	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3-19 为P_0分配资源后的有关资源数据

3.6.3 利用银行家算法避免死锁

(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，请读者考虑。

3.7 死锁的检测与解除

3.7.1 死锁的检测

- 允许死锁发生
- 保存有关资源的请求和分配信息；提供检测算法
- 检测时机
  - 太频繁：系统效率低。
  - 选择时机：当CPU利用率低于某一值时。

3.7.1 死锁的检测

1．资源分配图(Resource Allocation Graph)

系统死锁可利用资源分配图来描述。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，…，p_n}和一组资源结点R={r_1，r_2，…，r_n}，N=P∪R。在图3-20所示的例子中，P={p_1，p_2}，R={r_1，r_2}，N={r_1，r_2}∪{p_1，p_2}。

3.7.1 死锁的检测

(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。图3-13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)}。

我们用圆圈代表一个进程，用方框代表一类资源。由于一种类型的资源可能有多个，我们用方框中的一个点代表一类资源中的一个资源。此时，请求边是由进程指向方框中的r_j，而分配边则应始于方框中的一个点。图3-20示出了一个资源分配图。图中，p_1进程已经分得了两个r_1资源，并又请求一个r_2资源；p_2进程分得了一个r_1和一个r_2资源，并又请求r_1资源。

3.7.1 死锁的检测

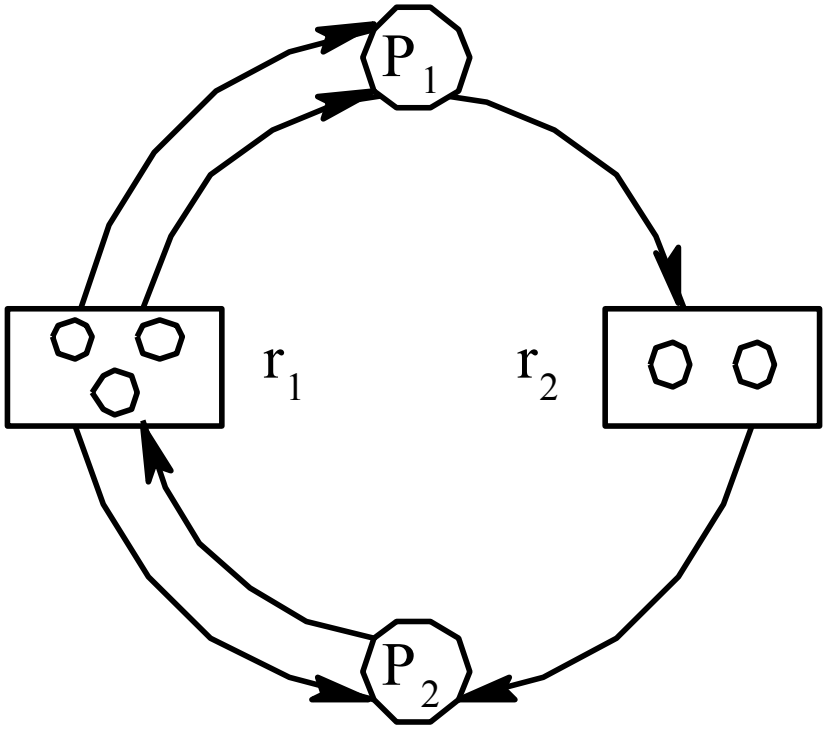


图3-20 每类资源有多个时的情况

3.7.1 死锁的检测

2．死锁定理

我们可以利用把资源分配图加以简化的方法(图3-21)，来检测当系统处于S状态时是否为死锁状态。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。在顺利的情况下，P_i可获得所需资源而继续运行，直至运行完毕，再释放其所占有的全部资源，这相当于消去P_i所求的请求边和分配边，使之成为孤立的结点。在图3-21(a)中，将p_1的两个分配边和一个请求边消去，便形成图(b)所示的情况。

3.7.1 死锁的检测

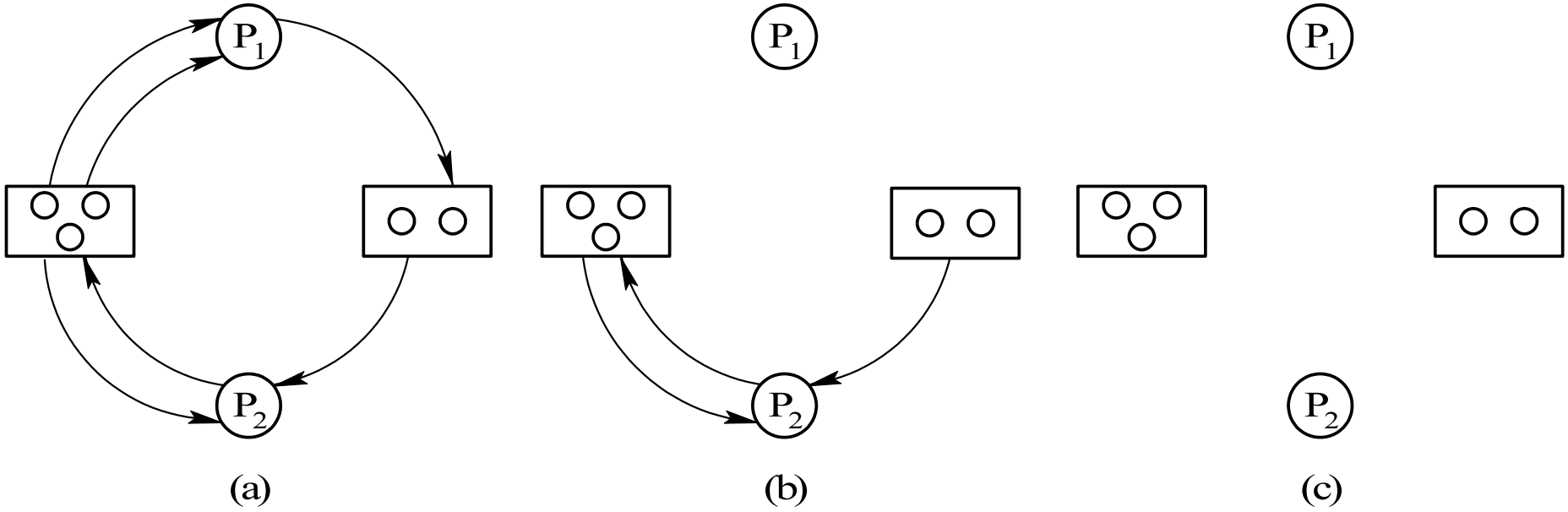


图3-21 资源分配图的简化

3.7.1 死锁的检测

(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，形成图(c)所示的情况。

(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。

对于较复杂的资源分配图，可能有多个既未阻塞，又非孤立的进程结点，不同的简化顺序是否会得到不同的简化图？有关文献已经证明，所有的简化顺序，都将得到相同的不可简化图。同样可以证明：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。

3.7.2 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。

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。

小结

- 概念：周转时间、带权周转时间、死锁、死锁预防、死锁避免、死锁检测、死锁恢复
- 三种调度类型的比较
- 调度时机、切换与过程
- 调度方式、调度的基本准则
- 调度算法（先来先服务调度算法；短作业（短任务、短进程、短线程）优先调度算法；时间片轮转调度算法；优先级调度算法；高响应比优先调度算法；多级反馈队列调度算法。）
- 产生死锁的原因、四个必要条件。
- 进程资源图、银行家算法。
- 死锁定理

调度算法小结

- FCFS
  - 适应：作业，进程（线程）
  - 特点：实现简单，开销小，有利于长作业，适合批处理系，平均周转时间长
- SJF
  - 适应：作业，进程（线程）
  - 特点：实现复杂，开销大，有利于短作业，适合批处理系统，平均周转时间短，有饥饿现象。
- HRN(Highest Response-ratio Next,最高响应比优先）

  响应比RR=（等待时间+估计运行时间）/估计运行时间
  - 适应：作业，进程（线程）
  - 特点：实现复杂，开销大，公平，适合批处理系统平均周转时间较短

调度算法小结

- Priority
  - 适应：作业，进程（线程）
  - 特点：实现复杂，开销大，有利于紧迫作业，
    适合实 时系统
- RR
  - 适应：进程（线程)
  - 特点：实现简单，开销大，响应时间短，不区分进程的性质
    适合分时系统
- MLQ&MLFQ
  - 适应：进程（线程)
  - 特点：实现复杂，开销大，响应时间较短，区分进程性质，
    适合分时系统和通用系统。

作业

1.假定某多道程序系统有供用户使用的主存空间100k,磁带机2台,打印机1台,系统采用可变式分区管理主存,优先分配内存的低地址部分,且作业在内存中不能移动.设备采用静态分配技术.主存中的作业平分cpu的时间.作业采用先来先服务调度算法.I/O操作的时间忽略不计.

现有如下一组作业序列:

作用号	提交时间	运行时间(分)	主存量	磁带机	打印机
1	8:00	25	15k	1	1
2	8:20	10	30k	0	1
3	8:20	20	60k	1	0
4	8:30	20	20k	1	0
5	8:35	15	10k	1	1

试求：

1. 调度程序选中作业的次序是____. (用作业号描述)
2. 最大的作业周转时间是____分.
3. 最小的作业周转时间是____分
4. 作业全部完成的时间是____(时:分描述)
5. 作业3的带权周转时间是____.

作业

2．一个计算机有6台磁带机，由n个进程竞争使用，每个进程可能需要两台磁带机，那么n是多少时，系统才没有死锁的危险？

3. P1、P2两个进程同步算法如下：
设semaphore:s1=1,s2=0,s3=1;

```
P1: begin                P2: begin
    ......                   ......
    P(s1);                   P(s3);
    P(s2);                   P(s1);
    ......                   ......
    V(s2);                   V(s3);
    V(s1);                   V(s1);
    end                      end
```

试分析P1,P2是否会产生死锁。

作业

4．设系统中有A类资源10个，B类资源6个，C 类资源 7个。系统采用银行家算法避免死锁发生，现有5个进程，当前状态如下图所示。

	Allocation			Max			Available			Need		
	A	B	C	A	B	C	A	B	C	A	B	C
P0	0	1	0	7	5	3	3	4	2	7	4	3
P1	2	0	0	3	2	3				1	2	3
P2	3	0	2	6	1	6				3	1	4
P3	2	1	1	5	2	2				3	1	1
P4	0	0	2	4	3	3				4	3	1

- 问 当进程p2提出一个请求（1，1， 1），系统是否可以满足？进程p3提出请求（1，1，1）呢？以上两个请求若系统可以满足，请给出系统处理过程中，判断系统是安全状态时的进程完成顺序。