

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 死锁的概述**
- 3.6 预防死锁**
- 3.7 避免死锁**
- 3.8 死锁的检测与解除**



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 死锁的概述**
- 3.6 预防死锁**
- 3.7 避免死锁**
- 3.8 死锁的检测与解除**



3.1 处理机调度的基本概念

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

分配处理机的任务是由处理机调度程序完成的。由于处理机是最重要的计算机资源，提高处理机的利用率及改善系统性能，在很大程度上取决于处理机调度的性能。

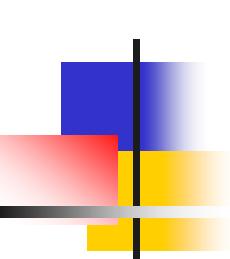
因此，处理机调度便成为OS设计的中心问题之一。



3.1 处理机调度的基本概念

3.1.1 处理机调度的层次

3.1.2 处理机调度的算法的目标



3.1 处理机调度的基本概念

3.1.1 处理机调度的层次

3.1.2 处理机调度的算法的目标

3.1.1 处理机调度的层次

一个批处理型作业，从进入系统并驻留在外存的后备队列上开始，直至作业运行完毕，可能要经历下述三级调度。

1. 高级调度（High Scheduling）

又称为**作业调度或长程调度（Long-Term Scheduling）**，用于决定把外存上处于后备队列中的哪些作业调入内存，并为它们创建进程、分配必要的资源，然后将新创建的进程排在就绪队列上，准备执行。

因此有时也称作业调度为**接纳调度（Admission Scheduling）**。

3.1.1 处理机调度的层次

1. 高级调度 (High Scheduling)

在批处理系统中，因作业进入系统后先驻留在外存，故需要有作业调度。在分时系统中为做到及时响应，作业被直接送入内存，故不需作业调度。在实时系统中，通常也不需作业调度。

在每次执行作业调度时，都须作出两个决定：

- **接纳多少作业**——每次接纳多少作业进入内存，即允许多少个作业同时在内存中运行---多道程序度。其确定应根据系统的规模、运行速度等情况综合考虑。
- **接纳哪些作业**——应接纳哪些作业从外存调入内存，取决于所采用的调度算法。如先来先服务，短作业优先等，在 § 3.2 中详细介绍。

3.1.1 处理机调度的层次

1. 高级调度（**High Scheduling**）
2. 低级调度（**Low Level Scheduling**）

通常也称为进程调度或短程调度（**Short-Term Scheduling**），用来决定就绪队列中的哪个进程应获得处理机，然后再由分派程序把处理机分配给该进程。为最基本的一种调度，三种**OS**中都有进程调度可采用下述两种调度方式：

- **非抢占方式（Non-preemptive Mode）**

一旦把处理机分配给某进程后，便让该进程一直执行，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他进程，决不允许进程抢占已分配出去的处理机。

评价：实现简单、系统开销小；适用于大多数的批处理**OS**，但在要求比较严格的实时系统中，不宜采用这种调度方式

3.1.1 处理机调度的层次

1. 高级调度（High Scheduling）
2. 低级调度（Low Level Scheduling）

进程调度可采用下述两种调度方式：

- 非抢占方式（Non-preemptive Mode）
- 抢占方式（Preemptive Mode）

允许调度程序根据某种原则，去暂停某个正在执行的进程，将处理机重新分配给另一进程。

抢占的原则：

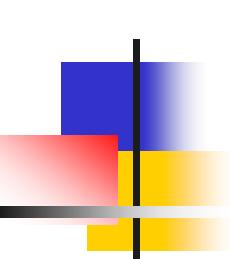
- 优先权原则：优先权高的可以抢占优先级低的进程的处理机。
- 短作业（进程）优先原则：短作业（进程）可以抢占长作业（进程）的处理机。
- 时间片原则：各进程按时间片运行，一个时间片用完时，停止该进程执行重新进行调度。

3.1.1 处理机调度的层次

1. 高级调度（High Scheduling）
2. 低级调度（Low Level Scheduling）
3. 中级调度（Intermediate-Level Scheduling）

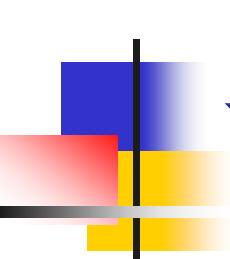
又称中程调度（Medium-Term Scheduling）。引入目的是为了提高内存利用率和系统吞吐量。为此，应使那些暂时不能运行的进程不再占用宝贵的内存资源，而将它们调之外存去等待，把此时的进程状态称为就绪驻外存状态或挂起状态。当这些进程重又具备运行条件、且内存又稍有空闲时，由中级调度来决定把外存上的哪些又具备运行条件的就绪进程，重新调入内存，并修改其状态为就绪状态，挂在就绪队列上等待进程调度。

中级调度实际上就是存储器管理中的对换功能，将在 § 4.4 中详述。



3.1.1 处理机调度的层次

- **进程调度**的运行频率最高，在分时系统中通常是**10~100ms**便进行一次进程调度，因而进程调度算法不能太复杂，以免占用太多的**CPU**时间。
- **作业调度**是发生在一个作业运行完毕，退出系统，而需要重新调度一个作业进入内存时，故作业调度的周期较长，大约几分钟一次。因而也允许作业调度算法花费较多的时间。
- **中级调度**的运行频率，基本上介于进程调度和作业调度之间。



调度队列模型

不论高级、中级或者低级调度，都涉及到进程队列，由此形成了三种类型的调度队列模型：

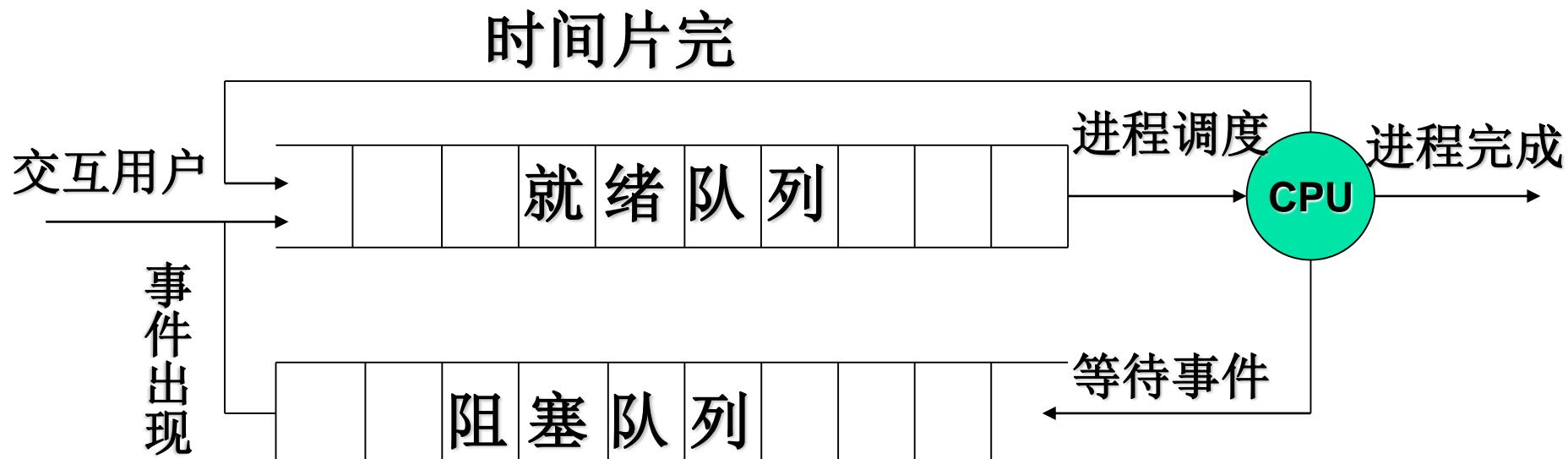
- 1.** 仅有进程调度的调度队列模型
- 2.** 具有高级和低级调度的调度队列模型
- 3.** 同时具有三级调度的调度队列模型

调度队列模型

1. 仅有进程调度的调度队列模型

在分时系统中，通常仅设置进程调度。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，形式取决于OS类型和所采用的调度算法。

在分时系统中就绪进程组织成**FIFO**队列形式，按时间片轮转方式运行。



调度队列模型

1. 仅有进程调度的调度队列模型

2. 具有高级和低级调度的调度队列模型

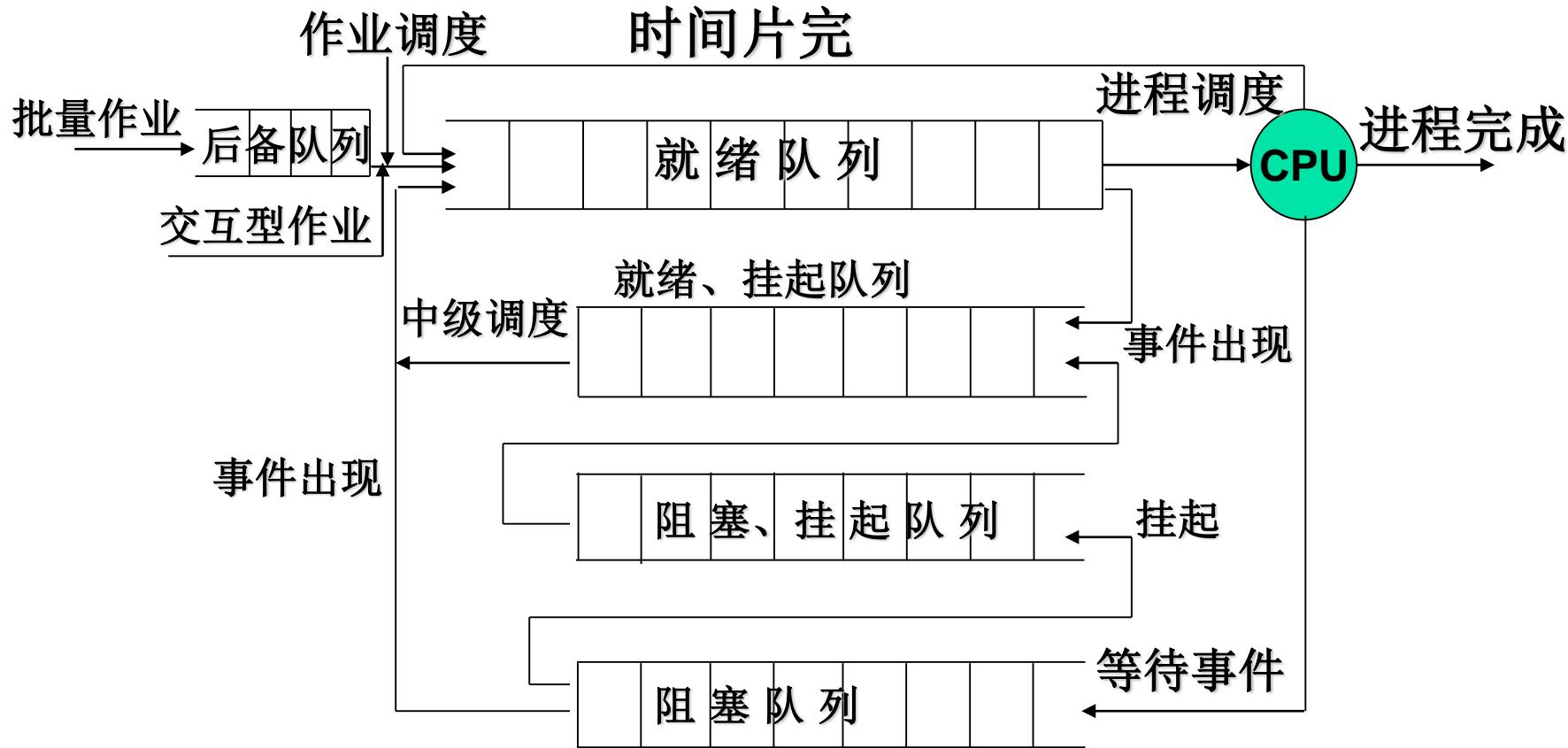
在批处理系统中，不仅需要进程调度，而且还需要作业调度，由作业调度按一定的调度算法，从外存的后备队列中选择一批作业调入内存，并为它们建立进程，送入就绪队列，然后才由进程调度算法按照一定的进程调度算法，选择一个进程，把处理机分配给该进程。

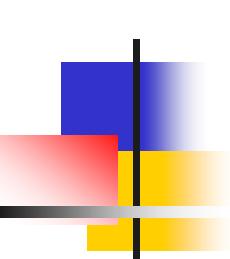


调度队列模型

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，可以把进程的就绪状态分为内存就绪和外存就绪。也可以把阻塞状态分为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变为外存就绪，由内存阻塞转变为外存阻塞；在中级调度的作用下，又可使外存就绪转变为内存就绪。





3.1 处理机调度的基本概念

3.1.1 处理机调度的层次

3.1.2 处理机调度的算法的目标

3.1.2 处理机调度的算法的目标

在一个**OS**的设计中，应如何选择调度方式和算法，很大程度上取决于**OS**的类型和目标。如在批处理系统、分时系统和实时系统中，通常都采用不同的调度方式和算法。选择的准则，有的是面向用户的，有的是面向系统的。

1. 处理机调度算法的共同目标

- 资源利用率
- 公平性
- 平衡性
- 策略强制执行

2. 批处理系统的目标

3. 分时系统的目标

4. 实时系统的目标

3.1.2 处理机调度的算法的目标

1. 处理机调度算法的共同目标

2. 批处理系统的目标

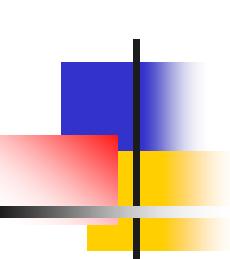
- 周转时间短
 - 周转时间； 平均周转时间
 - 带权周转时间； 平均带权周转时间
- 系统吞吐量高
- 处理机利用率好

3. 分时系统的目标

- 响应时间快： 响应时间
- 均衡性

4. 实时系统的目标

- 截止时间的保证： 截止时间
- 可预测性



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

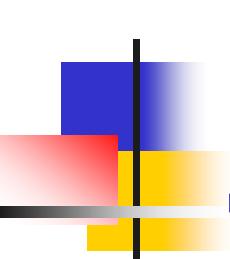
3.4 实时调度

3.5 死锁的概述

3.6 预防死锁

3.7 避免死锁

3.8 死锁的检测与解除



3.2 作业与作业调度

在**批处理系统**中，因作业进入系统后先驻留在外存，故需要有作业调度。在分时系统中为做到及时响应，作业被直接送入内存，故不需作业调度。在实时系统中，通常也不需作业调度。

在每次执行作业调度时，都须作出两个决定：

- 接纳多少作业
- 接纳哪些作业

作业调度算法

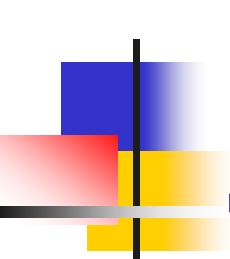
- 先来先服务和短作业（进程）优先调度算法
- 高优先权调度算法和高响应比优先调度算法

3.2 作业与作业调度

在OS中调度的实质是一种资源分配，因而调度算法是指：根据系统的资源分配策略所规定的资源分配算法。

对于不同的系统和系统目标，通常采用不同的调度算法：例如，在批处理系统中为照顾为数众多的短作业，应采用短作业优先的调度算法；又如在分时系统中，为了保证系统具有合理的响应时间，应采用轮转法进行调度。

目前存在的多种调度算法中，有的算法适用于作业调度，有的算法适用于进程调度；但有些算法作业调度和进程调度都可以采用。



3.2 作业与作业调度

- 作业调度算法
 - 先来先服务和短作业（进程）优先调度算法
 - 高优先权调度算法和高响应比优先调度算法

3.2.3 先来先服务和短作业（进程）优先调度算法

■ 先来先服务调度算法（FCFS）

是一种最简单的调度算法，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**FCFS**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，使之运行。

FCFS算法比较有利于长作业（进程），而不利于短作业（进程）。

3.2.3 先来先服务和短作业（进程）优先调度算法

■ 先来先服务调度算法（FCFS）

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
A	0	1
B	1	100
C	2	1
D	3	100

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**。**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：如通常的科学计算。
- **I/O**繁忙型作业：指**CPU**进行处理时，需频繁的请求**I/O**。

3.2.3 先来先服务和短作业（进程）优先调度算法

- 先来先服务调度算法（FCFS）
- 短作业（进程）优先调度算法 SJ(P)F

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。

短进程优先（SPF）调度算法，是从就绪队列中选出一估计运行时间最短的进程，将处理机分配给它，使它立即执行并一直执行到完成，或发生某事件而被阻塞放弃处理机时，再重新调度。

3.2.3 先来先服务和短作业（进程）优先调度算法

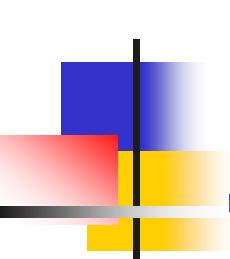
调度算法 作业情 况	进程名	A	B	C	D	E	平均
	到达时间	0	1	2	3	4	
	服务时间	4	3	5	2	4	
FCFS	完成时间						
	周转时间						
	带权周转时间						
SJF	完成时间						
	周转时间						
	带权周转时间						

采用**SJF算法**后，不论是平均周转时间还是平均带权周转时间，都较**FCFS**调度算法有较明显的改善，尤其是对短作业**D**。而平均带权周转时间从**2.8**降到了**2.1**。这说明**SJF**调度算法能有效的降低作业的平均等待事件，提高系统吞吐量。

3.2.3 先来先服务和短作业（进程）优先调度算法

SJF调度算法的优缺点：

- **优点：**有效降低作业的平均等待时间，提高系统吞吐量。
- **缺点：**
 - 对长作业不利。
 - 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。
 - 由于作业（进程）的长短含主观因素，不一定能真正做到短作业优先。



3.2 作业与作业调度

- 作业调度算法
 - 先来先服务和短作业（进程）优先调度算法
 - 高优先权调度算法和高响应比优先调度算法

3.2.4 高优先权优先调度算法

- 优先权调度算法的类型

为照顾紧迫性作业，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。它分为两种：

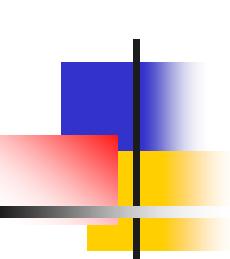
- 非抢占式优先权算法
- 抢占式优先权调度算法

3.2.4 高优先权优先调度算法

■ 优先权的类型

对于最高优先权优先调度算法，关键在于：使用静态优先权、动态优先权；如何确定进程的优先权。

- **静态优先权：**在创建进程时确定的，在进程的整个运行期间保持不变。一般利用某一范围的一个整数来表示，又称为优先数。
- **动态优先权：**在创建进程时所赋予的优先权可以随进程的推进或随其等待时间的增加而改变。
- ◆ 确定进程优先权的依据有如下三个方面：
 - **进程类型：**一般来说系统进程高于用户进程。
 - **进程对资源的需求：**如进程的估计时间及内存需要量的多少，对要求少的进程赋予较高优先权。
 - **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。



3.2.4 高响应比优先调度算法

- 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，其主要不足是长作业的运行得不到保证。我们为每个作业引入动态优先权，并使作业的优先级随着等待时间的增加而以速率**a**提高，则可解决问题。见下式：

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

由于等待时间与服务时间之和就是系统的响应时间，故上式又表示为： $R_p = \text{响应时间} / \text{要求服务时间}$

3.2.4 高响应比优先调度算法

■ 高响应比优先调度算法

优先权=(等待时间+要求服务时间)/要求服务时间
或 $R_p=响应时间/要求服务时间$

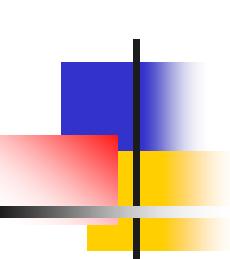
由上式可以看出：

- 如作业等待时间相同，则要求服务的时间愈短优先权愈高，所以该算法利于短作业。
- 当要求服务的时间相同，作业优先权的高低决定于其等待时间的长短，所以是先来先服务。
- 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也可获得处理机。

3.2 作业

- 假如有四道作业，它们的进入时间和运行时间由下表给出，在单道程序环境下，分别填写先来先服务和短作业优先情况的完成时间和平均周转时间。

作业号	进入时间(时)	运行时间(小时)	FCFS		SJF	
			完成时间(时)	周转时间(小时)	完成时间(时)	周转时间(小时)
1	10:00	0.4				
2	10:10	1				
3	10:20	0.6				
4	10:30	0.2				



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 死锁的概述**
- 3.6 预防死锁**
- 3.7 避免死锁**
- 3.8 死锁的检测与解除**

3.3 进程调度

在3.1.1节“处理机调度的层次”介绍过进程调度。

低级调度通常也称为进程调度或短程调度（**Short-Term Scheduling**），用来决定就绪队列中的哪个进程应获得处理机，然后再由分派程序把处理机分配给该进程。为最基本的一种调度，三种**OS**中都有。

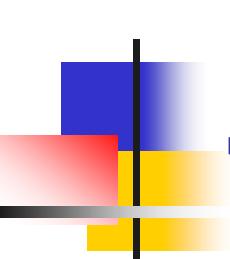
3.3 进程调度

进程调度可采用下述两种调度方式：

- 非抢占方式（**Non-preemptive Mode**）
- 抢占方式（**Preemptive Mode**）

抢占的原则：

- 优先权原则：优先权高的可以抢占优先级低的进程的处理机。
- 短作业（进程）优先原则：短作业（进程）可以抢占长作业（进程）的处理机。
- 时间片原则：各进程按时间片运行，一个时间片用完时，停止该进程执行重新进行调度。



3. 3 进程调度算法

在分时系统中，为保证能及时响应用户的请求，必须采用基于时间片的轮转式进程调度算法。在早期，分时系统中采用的是简单的时间片轮转法，进入**90**年代后，广泛采用多级反馈队列调度算法。下面分开介绍这两种方法并比较性能。

3.3.2 轮转调度算法

1. 时间片轮转法

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。

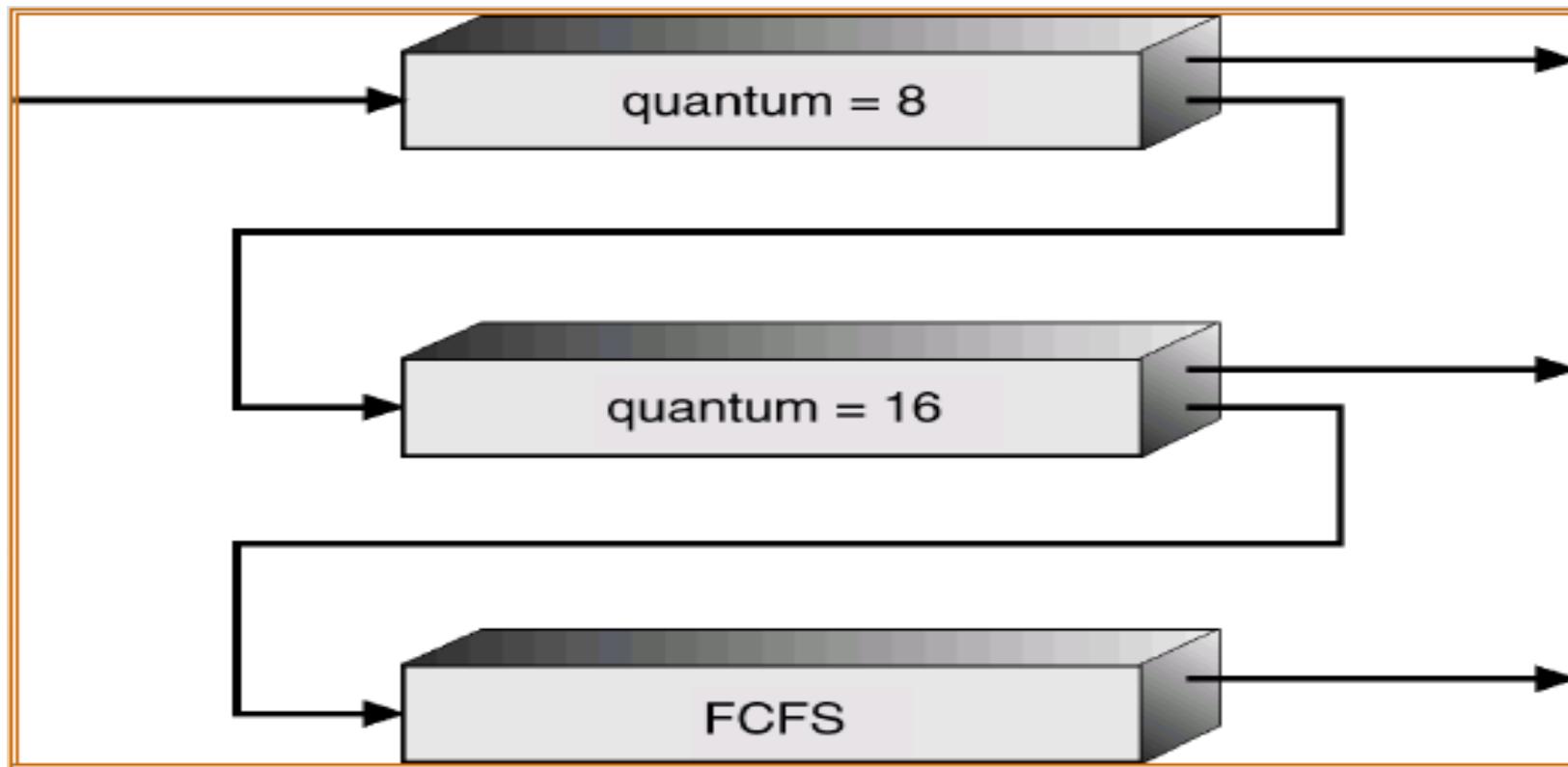
时间片的大小从几**ms**到几百**ms**。当执行的时间片用完时，停止该进程的执行并将其送往就绪队列的末尾。这样就可以保证就绪队列中所有进程在一定时间内均能获得一个时间片的处理机执行时间，也就是说系统能在给定的时间内响应所有用户的请求。

3.3.2 基于时间片的轮转调度算法

1. 时间片轮转法

2. 多级反馈队列调度算法

前面介绍的各种进程调度的算法都有一定的局限性。这里介绍一种目前被认为较好的进程调度算法——多级反馈队列调度算法。实施过程如下：



3.3.2 基于时间片的轮转调度算法

2. 多级反馈队列调度算法

- (1) 设置多个队列并为各个队列赋予不同的优先级。第一个最高，依次降低。各个队列中进程执行时间片的大小设置为：优先权越高，时间片越短。如第一个队列时间片为**1nms**, 第二个队列的时间片为**2nms**,
- (2) 当一个新进程进入内存后，首先将它放入第一个队列的末尾，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，在同样地按**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）从第一队列依次将到第**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

3.3.2 基于时间片的轮转调度算法

1. 时间片轮转法

2. 多级反馈队列调度算法

多级反馈队列调度算法具有较好的性能，能较好的满足各种类型用户的需要。

- 终端型作业用户。大多属于较小的交互性作业，只要能使作业在第一队列的时间片内完成，便可令用户满意。
- 短批处理作业用户。周转时间仍然较短，至多在第二到三队列即可完成。
- 长批处理作业用户。将依次在 $1 \sim n$ 级队列中轮转执行，不必担心作业长期得不到处理。

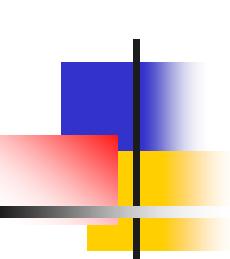
3.4 实时调度

由于在实时系统中都存在着若干个实时进程或任务，它们用来反应或控制某个（些）外部事件，往往带有某种程度的紧迫性，因而对实时系统中的调度提出了某些特殊要求，前面所介绍的多种调度算法，并不能很好的满足实时系统对调度的要求，为此，需要引入一种新的调度，即实时调度。

实时系统中包含两种任务：

硬实时任务指必须满足最后期限的限制，否则会给系统带来不可接受的破坏或者致命错误。

软实时任务也有一个与之关联的最后期限，并希望能满足这个期限的要求，但这并不是强制的，即使超过了最后期限，调度和完成这个任务仍然是有意义的。



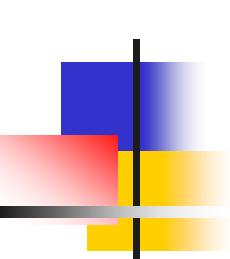
3.4 实时调度

3.4.1 实现实时调度的基本条件

3.4.2 实时调度算法的分类

3.4.3 最早截止时间优先**EDF**算法

3.4.4 最低松弛度优先**LLF**算法



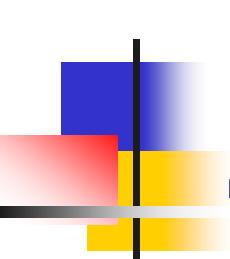
3.4 实时调度

3.4.1 实现实时调度的基本条件

3.4.2 实时调度算法的分类

3.4.3 最早截止时间优先**EDF**算法

3.4.4 最低松弛度优先**LLF**算法



3.4.1 实现实时调度的基本条件

在实时系统中，硬实时任务和软实时任务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下列**4**个条件：

- 提供必要的信息
- 系统处理能力强
- 采用抢占式调度机制
- 具有快速切换机制

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：

- **就绪时间**：该任务成为就绪状态的时间。
- **开始截止时间、完成截止时间**：只需知道一个。
- **处理时间**：从开始执行到完成所需时间。
- **资源要求**：任务执行时所需的一组资源。
- **优先级**：根据任务性质赋予不同优先级。

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，通常都有多个任务，若处理机的能力不够强，则可能会出现某些实时任务不能得到及时处理导致发生难以预料的后果。

假如系统中有**M**个周期性的硬实时任务，处理时间为**C_i**，周期时间表示为**P_i**，则单机系统中必须满足条件 $\sum (C_i / P_i) \leq 1$ 。

解决方法有二：增强单机系统的处理能力或采用多处理机系统（则限制条件为 $\sum (C_i / P_i) \leq N$ ，**N**为处理机数）。

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，这样可满足该硬实时任务对截止时间的要求。但此种机制较复杂。

对于一些小的实时系统，如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。

3.4.1 实现实时调度的基本条件

4. 具有快速切换机制

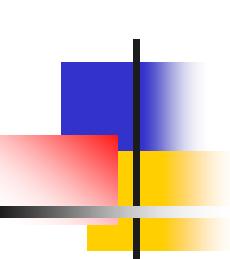
为保证要求较高的硬实时任务能及时运行，在实时系统中还应具有快速切换机制，以保证任务的快速切换。需要以下两种能力：

- **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，使可在紧迫的外部事件请求中及时响应。
- **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，为提高速度，应使系统中的运行功能单位适当的小。

3.4.1 实现实时调度的基本条件

在实时系统中，硬实时任务和软实时任务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下列条件：

- 提供必要的信息
- 系统处理能力强
- 采用抢占式调度机制
- 具有快速切换机制



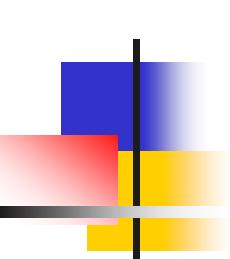
3.4 实时调度

3.4.1 实现实时调度的基本条件

3.4.2 实时调度算法的分类

3.4.3 最早截止时间优先**EDF**算法

3.4.4 最低松弛度优先**LLF**算法



3.4.2 实时调度算法的分类

可按照不同方式对实时调度算法加以分类：

- 根据实时任务性质的不同，分为硬实时调度算法和软实时调度算法；
- 按调度方式的不同，分为非抢占调度算法和抢占调度算法；
- 根据调度程序调度时间的不同，分为静态调度算法和动态调度算法。
- 多处理机环境下，可分为集中式调度和分布式调度两种算法。

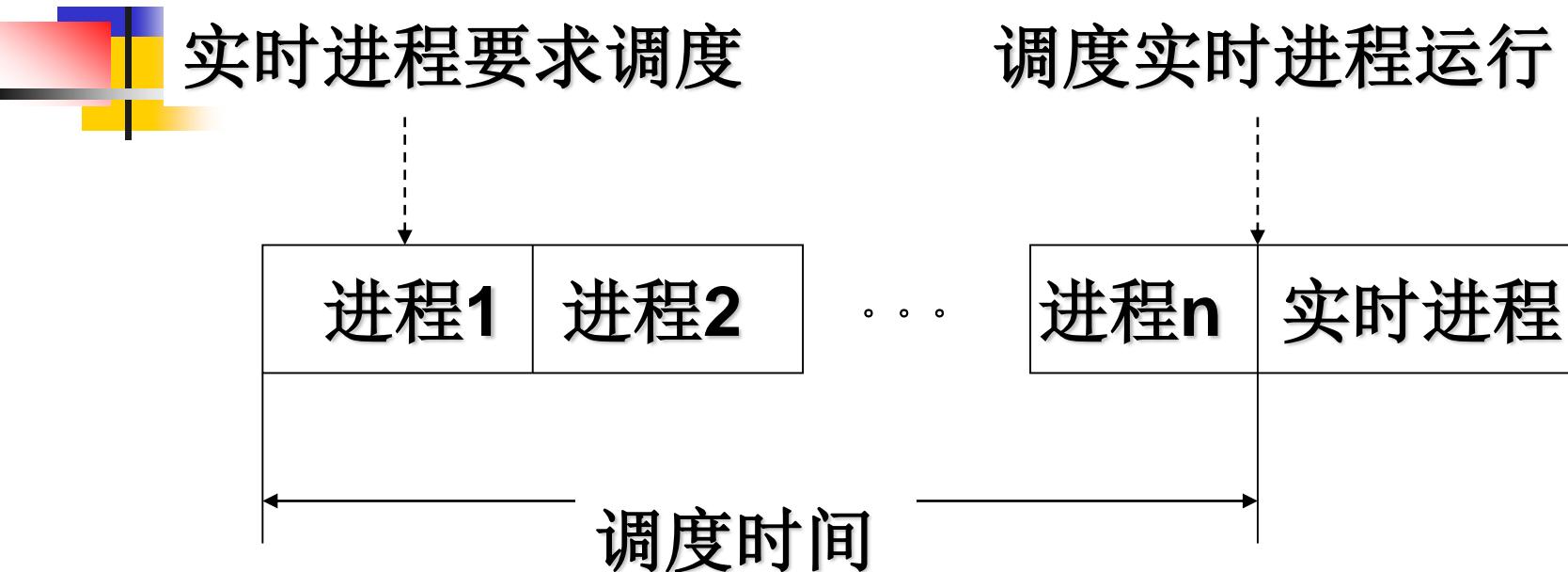
3.4.2 实时调度算法的分类

1. 非抢占调度算法

该算法较简单，用于一些小型实时系统或要求不太严格的实时系统中。又可分为两种：

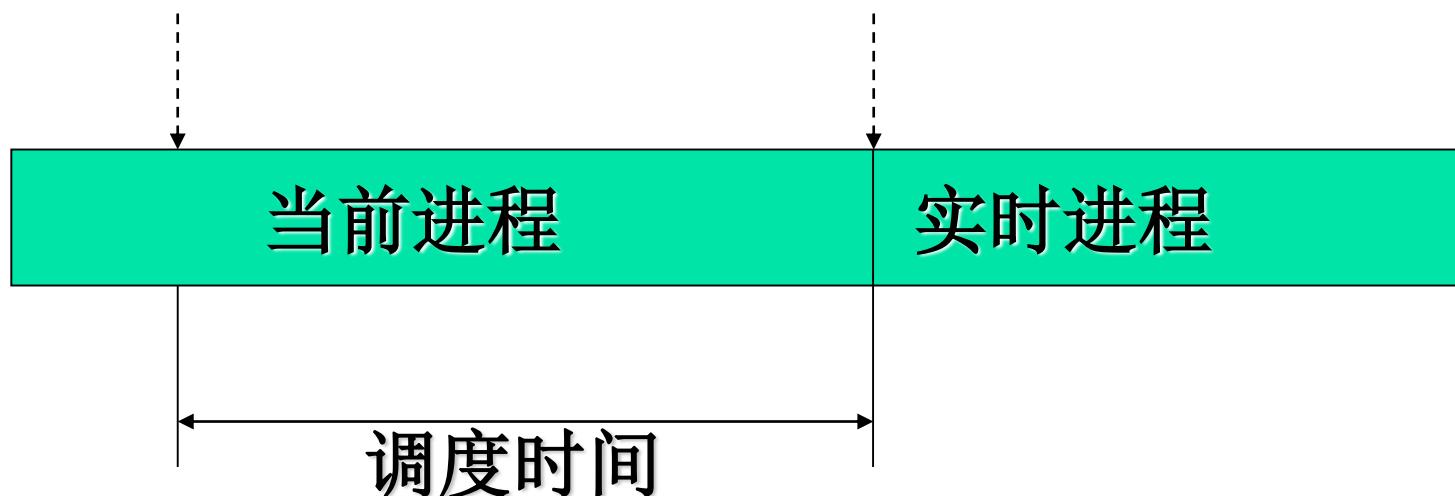
- **非抢占式轮转调度算法。** 常用于工业生产的群控系统中，要求不太严格。
- **非抢占式优先调度算法。** 要求较为严格，根据任务的优先级安排等待位置。可用于有一定要求的实时控制系统中。（响应时间 t 约为数百ms）

1) 非抢占式轮转调度图示



2) 非抢占式优先权调度图示

实时进程请求调度 当前进程运行完成



3.4.2 实时调度算法的分类

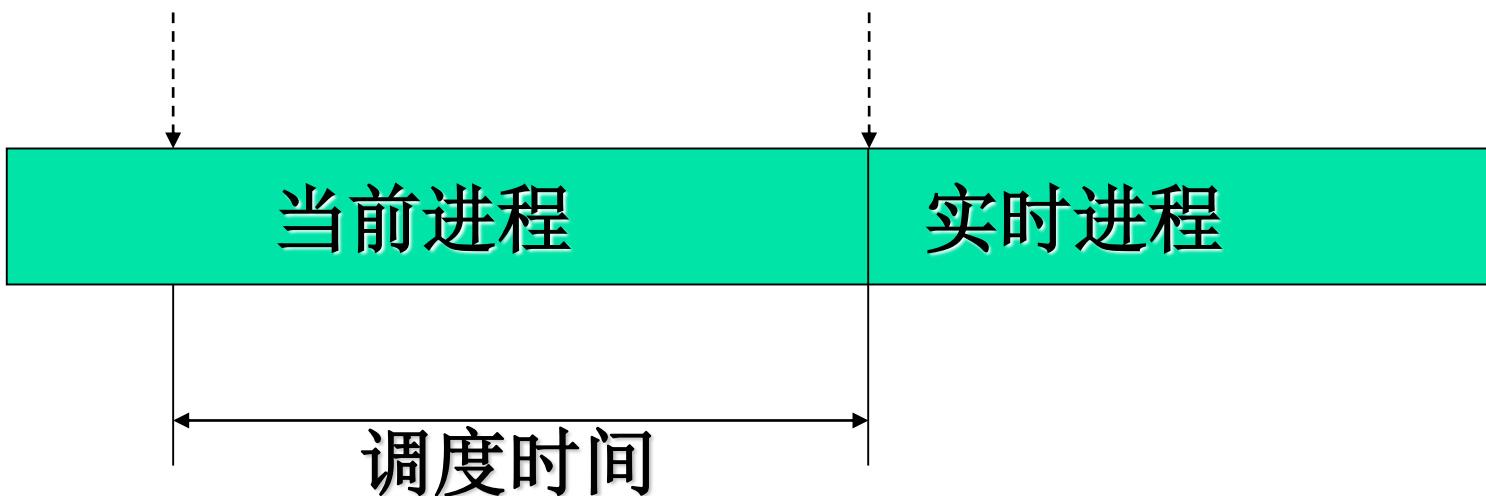
2. 抢占调度算法

用于要求较严格的实时系统中，(t 约为数十ms)，采用抢占式优先权调度算法。根据抢占发生时间的不同可分为两种：

- 基于时钟的抢占式优先权调度算法：某高优先级任务到达后并不立即抢占，而等下一个时钟中断时抢占。
- 立即抢占的优先权调度算法：一旦出现外部中断，只要当前任务未处于临界区，就立即抢占处理机。

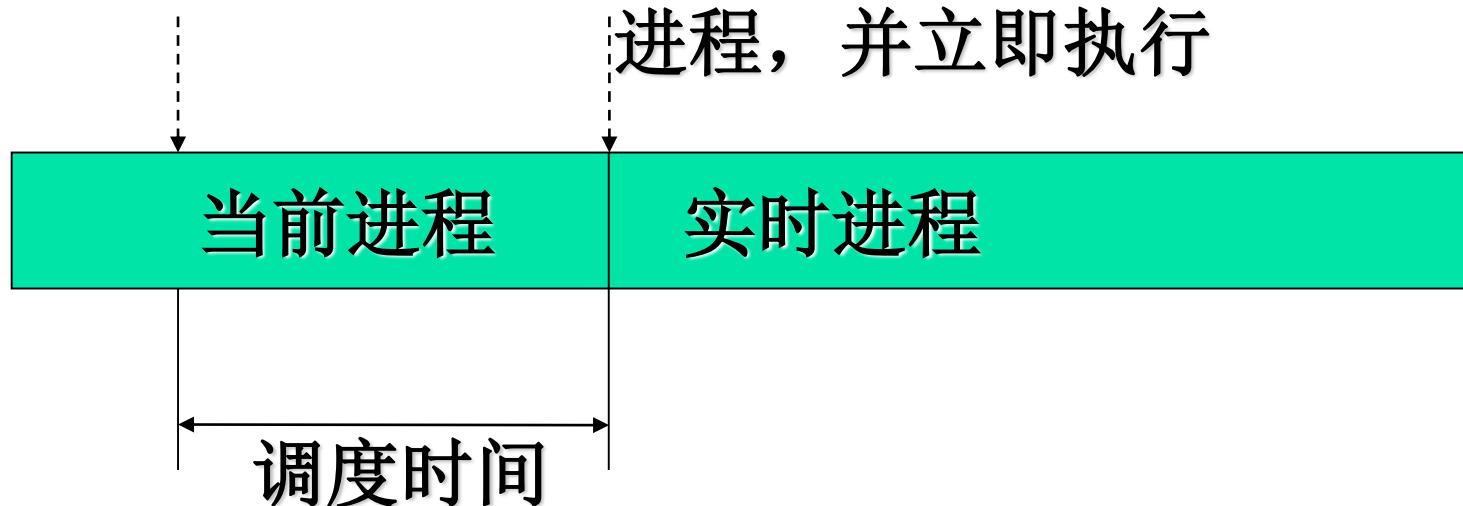
1) 基于时钟中断抢占的优先权调度图示

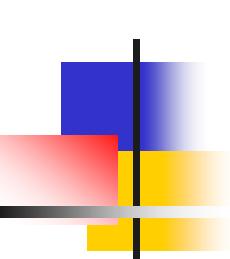
实时进程请求调度 时钟中断到来时



2) 立即抢占的优先权调度图示

实时进程请求调度 实时进程抢占当前进程，并立即执行





3.4 实时调度

3.4.1 实现实时调度的基本条件

3.4.2 实时调度算法的分类

3.4.3 最早截止时间优先EDF算法

3.4.4 最低松弛度优先LLF算法

3.4.3 常用的几种实时调度算法

目前有许多实时调度算法，在常用的算法中简单介绍两种实时调度算法：

- 最早截止时间优先**EDF**（**Earliest Deadline First**）算法
- 最低松弛度优先**LLF**（**Least Laxity First**）算法

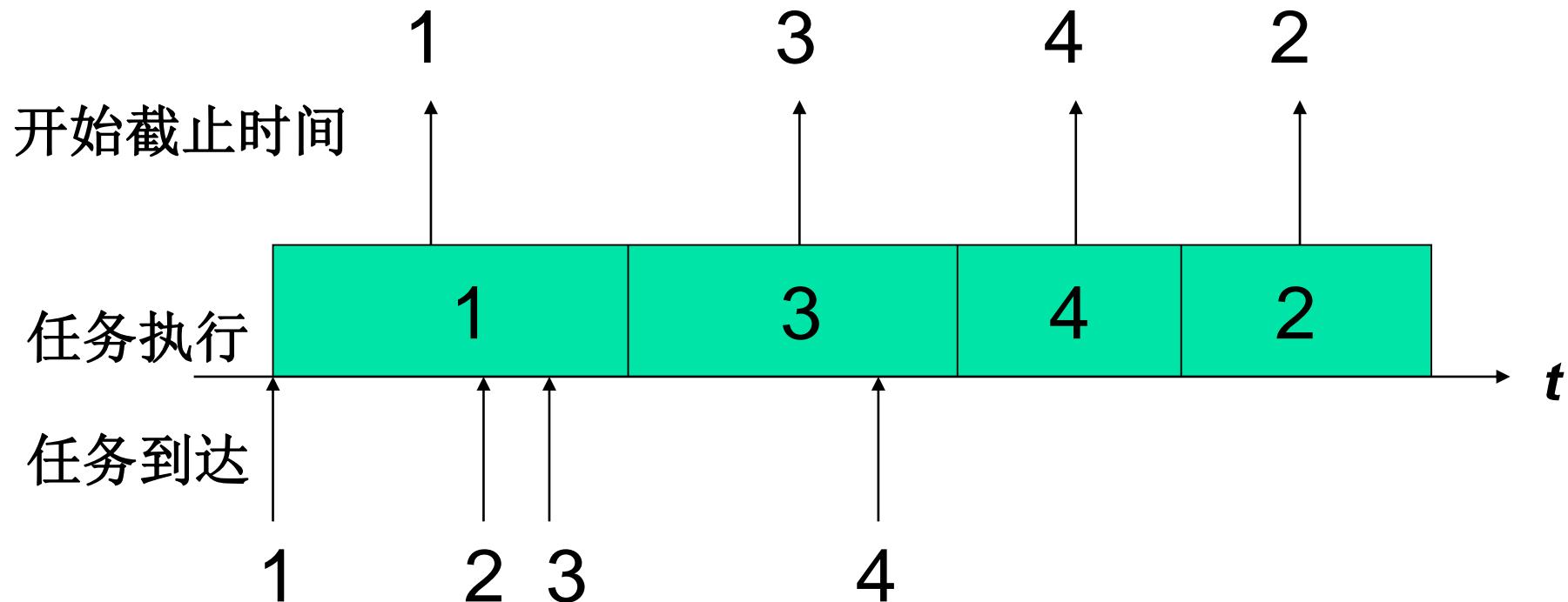
3.4.3最早截止时间优先算法

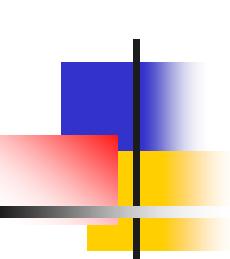
最早截止时间优先EDF (Earliest Deadline First) 算法

根据任务的截止时间来确定任务的优先级。截止时间越早，其优先级越高。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，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。

EDF算法既可以用于抢占式调度，也可用于非抢占式调度。

EDF算法用于非抢占调度图示

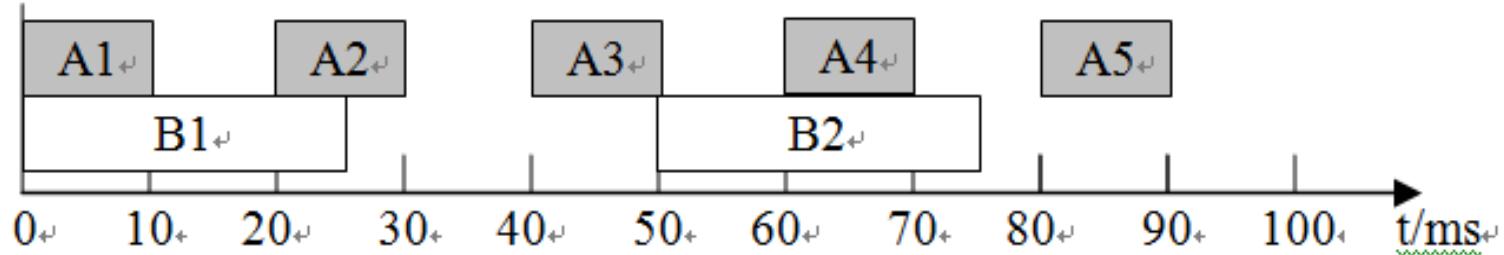




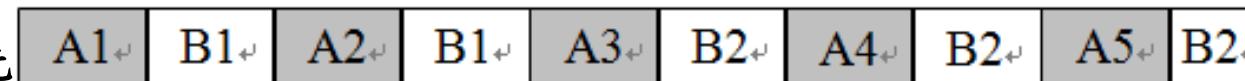
抢占式调度方式用于周期实时任务

下图中有两个周期性任务，任务**A**的周期时间为
20ms，每个周期的处理时间为**10ms**；任务**B**
的周期时间为**50ms**，每个周期的处理时间为
25ms

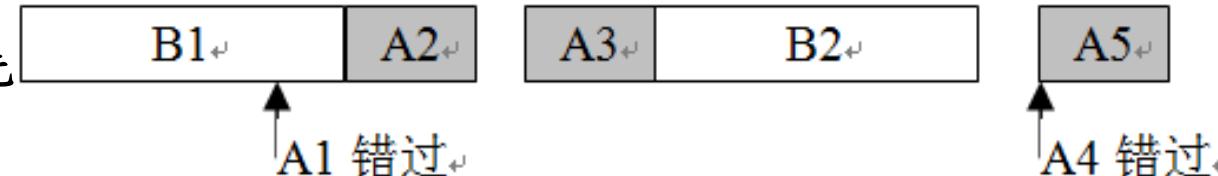
到达、执行时间和最后期限



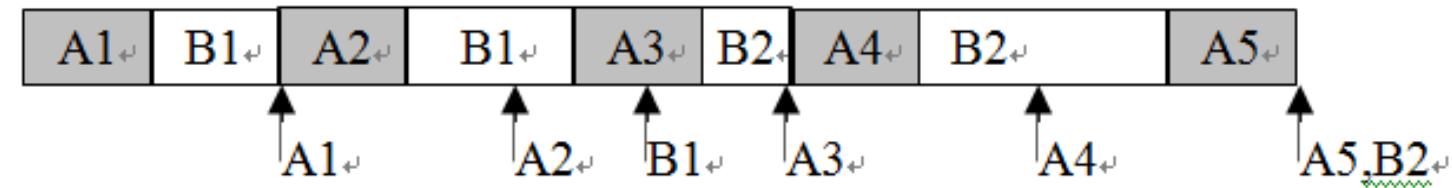
固定优先级 (A优先级高)



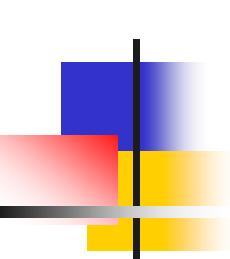
固定优先级 (B优先级高)



使用完成最后期限最早和最后期限调度



最早截止时间优先算法用于抢占调度方式



3.4 实时调度

3.4.1 实现实时调度的基本条件

3.4.2 实时调度算法的分类

3.4.3 最早截止时间优先**EDF**算法

3.4.4 最低松弛度优先**LLF**算法

3.4.4 最低松弛度优先算法

最低松弛度优先LLF（Least Laxity First）算法

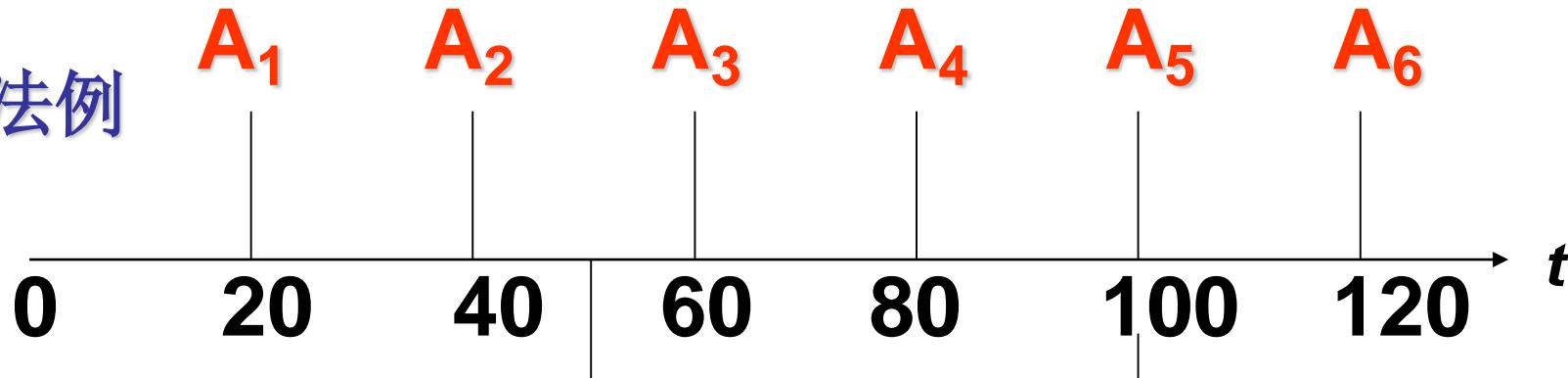
该算法是根据任务紧急（或松弛）的程度，来确定任务的先级。任务的紧急程度越高，为之赋予的优先级就越高。

例如，任务**A**在**200ms**时必须完成，本身运行时间**100ms**，则必须在**100ms**之前调度执行，**A**任务的紧急（松弛）程度为**100ms**，又如任务**B**在**400ms**是必须完成，需运行**150ms**，其松弛程度为**250ms**.

该算法主要用于抢占调度方式中。

假如在一个实时系统中，有两个周期型实时任务A,B，任务A要求每**20ms**执行一次，执行时间为**10ms**；任务B要求每**50ms**执行一次，执行时间为**25ms**；由此可得知A, B任务每次必须完成的时间分别为**A₁**、**A₂**、**A₃**...和**B₁**、**B₂**、**B₃**...

LLF算法例



任务A, B每次必须
完成的时间

A₁(10)

t_1	t_2
0	10

A₂(10) A₃(10)

t_3	t_4	t_5
30	40	50

A₄(10)

t_6	t_7	t_8
60	70	80

t

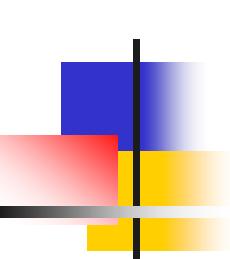
任务A,B的调度

B₁(20)

B₁ (5)

B₂(15)

B₂(10)

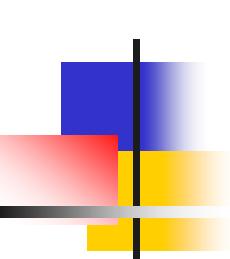


3.4 多处理机系统中的调度

3.4.1 多处理器系统的类型

3.4.2 进程分配方式

3.4.3 进程（线程）调度方式



3.4 多处理机系统中的调度

3.4.1 多处理器系统的类型

3.4.2 进程分配方式

3.4.3 进程（线程）调度方式

3.4.1 多处理器系统的类型

多处理器系统MPS (MultiProcessor System)

- 从多处理器之间耦合的紧密程度可以把**MPS**分为两类：**紧密耦合MPS**；**松弛耦合MPS**
- 根据系统中所用处理器的相同与否可分为两类：**对称MPS**；**非对称MPS**

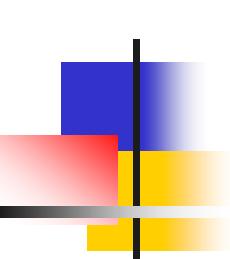
3.4.1 多处理器系统的类型

1. 紧密耦合MPS和松弛耦合MPS

- **紧密耦合 (Tightly Coupled)** 通常通过高速总线或高速交叉开关来实现多个处理器之间的互连。共享主存储器系统和I/O设备。系统中所有进程和资源由OS统一控制管理
- **松散耦合 (Loosely Coupled)** 通常通过通道或通信线路来实现多台计算机之间互连。每台计算机都有自己的存储器和I/O设备，可以独立工作。

2. 对称MPS和非对称MPS

- **对称多处理系统SMPS (Symmetric MultiProcessor System)** 在系统中所包含的各处理器单元在功能上和结构上都相同。当前的绝大多数MPS属于此类。
- **非对称多处理器系统。** 系统中有多种类型的处理单元，它们的功能和结构各不相同，其中只有一个主处理器，其余为从处理器。



3.4 多处理机系统中的调度

3.4.1 多处理器系统的类型

3.4.2 进程分配方式

3.4.3 进程（线程）调度方式

3.4.2 进程分配方式

在多处理器系统中，进程的调度与系统结构有关。

例如，在同构型系统中，由于所有的处理器都是相同的，因而可将进程分配到任一处理器上运行；

但对于非对称**MPS**，对任一进程而言，都只能将其分配到某一适合于其运行的处理机上去执行。

下面分开介绍对称**MPS**和非对称**MPS**中的进程分配方式。

3.4.2 进程分配方式

1. 对称MPS中的进程分配方式

因各处理器相同，故在进程分配时可任意。一般有以下两种具体分配方式：

- **静态分配 (Static Assignment) 方式：**一个进程从开始执行直至其完成都被固定的分配到一个处理器上去执行。优点是进程调度开销小；缺点是各处理器可能出现忙闲不均。
- **动态分配 (Dynamic Assignment) 方式：**在系统中仅设置一个公共的就绪队列，分配进程总是给空闲处理器，且对某一进程的执行来讲也可能曾在不同的处理器上。优点是消除忙闲不均现象；对松散耦合系统增大开销。

3.4.2 进程分配方式

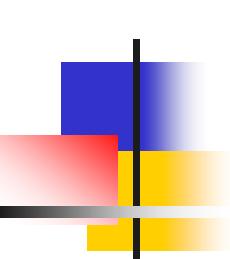
2. 非对称MPS中的进程分配方式

对于非对称**MPS**, 其**OS**大多是采用主-从式**OS**, 即**OS**的核心部分驻留在一台主机上, 而从机上只是用户程序, 进程调度只由主机执行。每当从机空闲时向主机发一索求进程信号, 然后等待主机分配进程。主机中保持有一个就绪队列。

此种方式优点是: 系统处理比较简单;

缺点是: 不可靠。

(克服缺点的方法是利用多台而非一台管理系统)

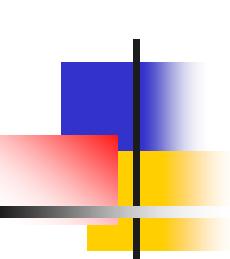


3.4 多处理机系统中的调度

3.4.1 多处理器系统的类型

3.4.2 进程分配方式

3.4.3 进程（线程）调度方式



3.4.3 进程（线程）调度方式

MPS已广为流行多年，存在多种调度方式，有许多都是以线程作为基本调度单位的，比较有代表的是以下几个：

- **自调度（Self-Scheduling）方式**
- **成组调度（Gang Scheduling）方式**
- **专用处理器（Dedicated Processor Assignment）分配方式**

3.4.3 进程（线程）调度方式

1. 自调度（Self-Scheduling）方式

■ 自调度机制

是多处理系统中最简单的一种调度方式。在系统中设置有一个公共的进程或线程就绪队列，所有的处理器空闲时，都可自己到该队列中取得一进程（线程）来运行。调度算法可以采用**FCFS**、**FIFO**和抢占式最高优先权优先调度算法等。经实验证明**FCFS**算法在多处理器环境下简单开销小，目前成为较好的调度算法。

优点：

- 易将单机环境下的调度机制移植到**MPS**中；
- 不会发生处理器忙闲不均的现象，有利于提高处理器的利用率

缺点：

- 瓶颈问题。多处理器互斥访问唯一就绪队列。
- 低效性。高速缓存的使用效率很低。
- 线程切换频繁。相关线程未必会同时获得处理器进而切换频繁

3.4.3 进程（线程）调度方式

2. 成组调度（Gang Scheduling）方式

为了解决自调度方式中线程被频繁切换的问题提出的成组调度方式，是指将一个进程中的一组线程分配到一组处理器上去执行。

如何分配处理器事件有下列两种方式：

- 面向所有应用程序平均分配处理器时间
- 面向所有线程平均分配处理器时间

按线程平均分配处理器时间的方法更有效。

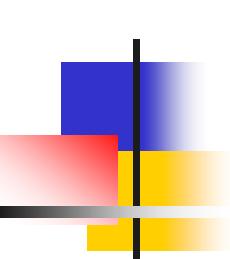
成组调度方式的主要优点：如果一组相互合作的进程或线程能并行执行，则可有效的减少线（进）程阻塞情况的发生。从而可以减少线程的切换，使系统性能得到改善；此外因每次调度都可以解决一组线程的处理器分配问题，故可显著减少调度频率，减少了调度开销。

3.4.3 进程（线程）调度方式

3. 专用处理器（Dedicated Processor Assignment）方式

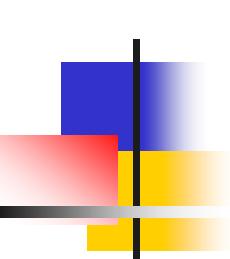
该方式是指在一个应用程序的执行期间，专门为该应用程序分配一组处理器，每一个线程一个处理器。这组处理器仅供该应用程序专用，直至该应用程序完成。很明显，这种方式很浪费。但理由如下：

- 对系统的性能和效率来讲，单个处理器的利用率已不太重要。
- 由于每个进（线）程专用一台处理器，可避免进（线）程的切换，从而大大加速了程序运行。



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 死锁的概述**
- 3.6 预防死锁**
- 3.7 避免死锁**
- 3.8 死锁的检测与解除**

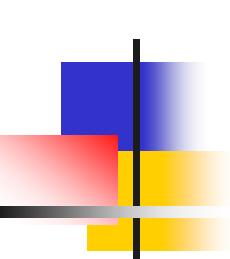


3.5 死锁概述

关于死锁

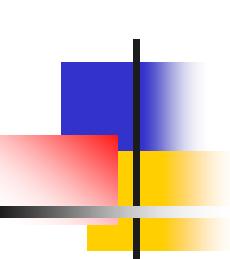
在多道程序系统中，虽可借助于多个进程的并发执行，来改善系统的资源利用率，提高系统的吞吐量，但可能发生一种危险——死锁。

死锁（Deadlock）：是指多个进程在运行过程中因争夺资源而造成的一种僵局，当进程处于这种状态时，若无外力作用，它们都将无法再向前推进。



3.5 死锁概述

- 1 产生死锁的原因**
- 2 产生死锁的必要条件**
- 3 处理死锁的基本方法**

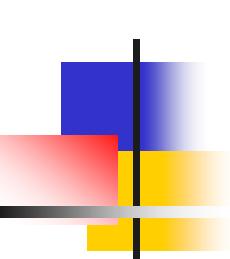


3.5 死锁概述

1 产生死锁的原因

2 产生死锁的必要条件

3 处理死锁的基本方法



1 产生死锁的原因

- 产生死锁的原因可归结为如下两点：
- **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。
- **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，也同样会导致产生死锁。

1 产生死锁的原因

1. 竞争资源引起进程死锁

- 可剥夺和非剥夺性资源（可把系统中的资源分为两类）

可剥夺性资源：资源分配给进程后可以被高优先级的进程剥夺。如**CPU**、主存。

不可剥夺性资源：分配给进程后只能在进程用完后才释放的资源。如磁带机、打印机等。

- 竞争非剥夺性资源引起死锁

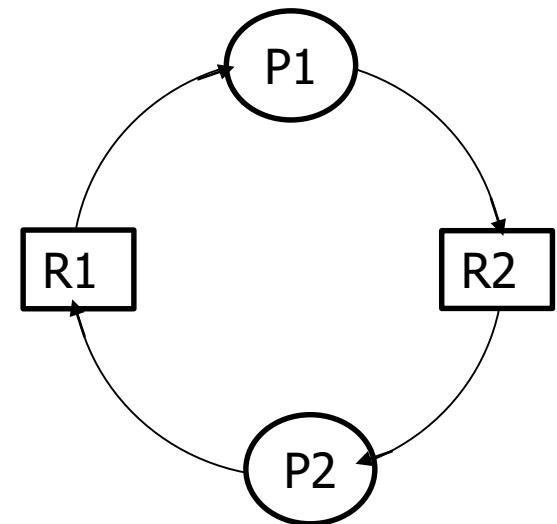
在系统中配置的非剥夺性资源，由于数量不足，会使进程在运行过程中，因争夺这些资源而陷入僵局。

- 竞争临时性资源引起死锁

打印机之类的资源属于可顺序重复使用型资源，称为永久性资源。与之相对的是临时性资源，是指由一个进程产生，被另一进程使用一短暂时后便无用的资源，也称为消耗性资源，它也可能引起死锁。

竞争不可剥夺性资源引起死锁

在系统中配置的非剥夺性资源，由于数量不足，会使进程在运行过程中，因争夺这些资源而陷入僵局。如，打印机R1，磁带机R2，可供进程P1和P2共享，假定P1已占用R1，P2已占用R2，此时P2又要求R1，因得不到进入阻塞状态，P1又要求R2，也得不到而进入阻塞状态，产生死锁。



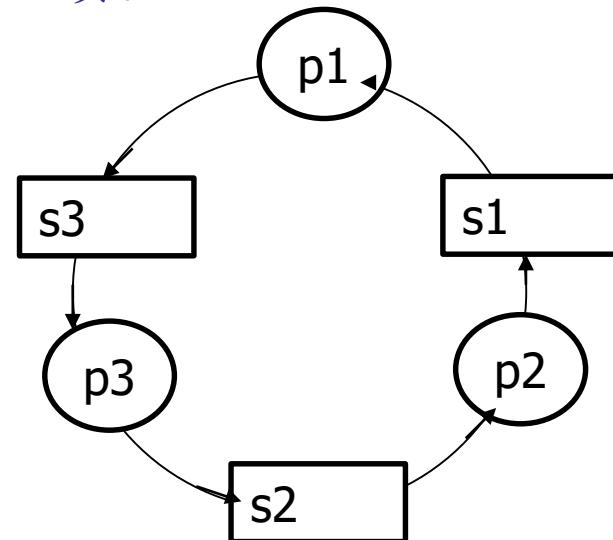
竞争临时性资源引起死锁

打印机之类的资源属于可顺序重复使用型资源，称为永久性资源。与之相对的是临时性资源，是指由一个进程产生，被另一进程使用一短暂时后便无用的资源，也称为消耗性资源，它也可能引起死锁。

P1: ...Request(s3);release(s1)...

P2: ...Request(s1);release(s2)...

P3: ...Request(s2);release(s3)...

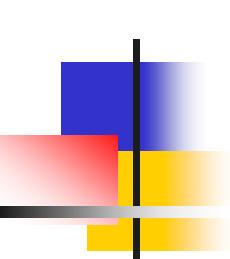


1 产生死锁的原因

1. 竞争资源引起进程死锁

2. 进程推进顺序不当引起死锁

- 由于进程在运行中具有异步性特征，这就可能使上述**P1**和**P2**两个进程按下述两种顺序向前推进。
 - 进程推进顺序合法
 - 进程推进顺序非法



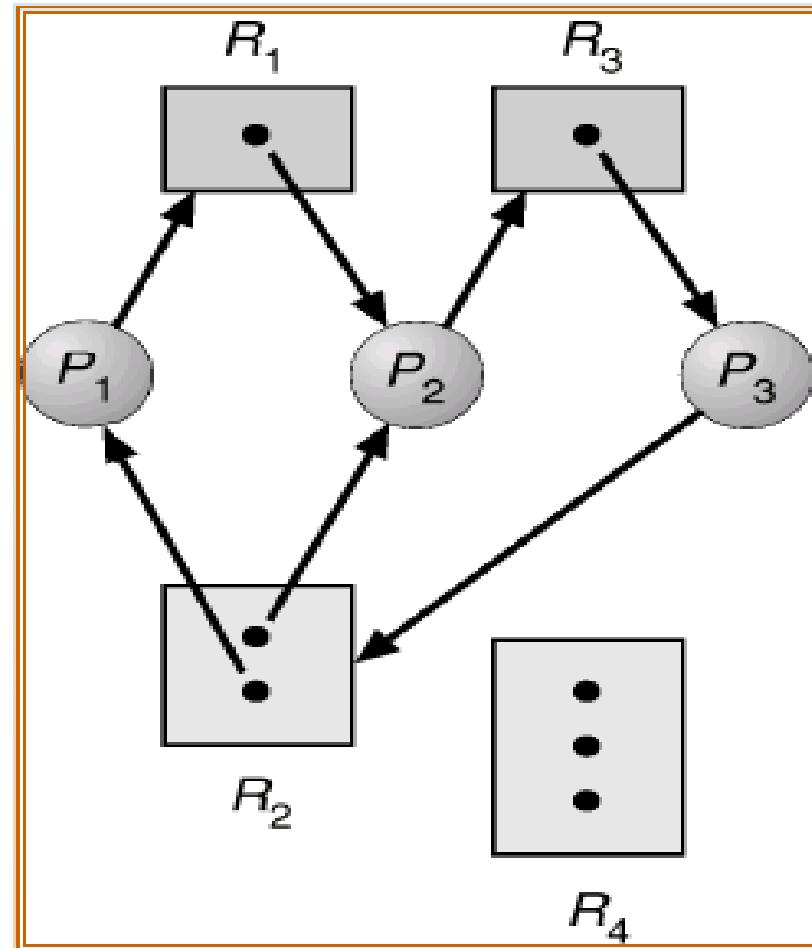
3.5 死锁概述

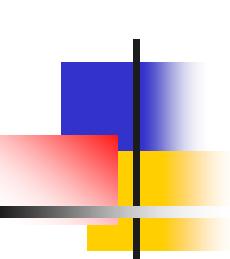
- 1 产生死锁的原因**
- 2 产生死锁的必要条件**
- 3 处理死锁的基本方法**

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。可以看出，必须具备以下四个必要条件：
 - 1.互斥条件：**指进程对所分配到的资源进行排他性使用，即在一段时间内某资源只由一个进程占用。如果此时还有其他进程请求该资源，则请求着只能等待，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（该资源又被其他进程占有，此时请求进程阻塞，但又对自己已获得的其他资源保持不放）。
 - 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 $\{P_0, P_1, P_2, \dots, P_n\}$ 中的 P_0 正在等待一个 P_1 占用的资源； P_1 正在等待一个 P_2 占用的资源，……， P_n 正在等待一个已被 P_0 占用的资源。

有死锁情况的资源分配图





3.5 死锁概述

- 1 产生死锁的原因**
- 2 产生死锁的必要条件**
- 3 处理死锁的基本方法**

3 处理死锁的基本方法

- 为保证系统中诸进程的正常运行，应事先采取必要的措施，来预防发生死锁。在系统中已经出现死锁后，则应及时检测到死锁的发生，并采取适当措施来解除死锁。目前处理死锁的方法可归结为以下四种：

1. 预防死锁

这是一种较简单和直观的事先预防方法。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的一个或几个，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。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。这种方法只须事先加以较弱的限制条件，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。目前在较完善的系统中，常用此方法来避免发生死锁。

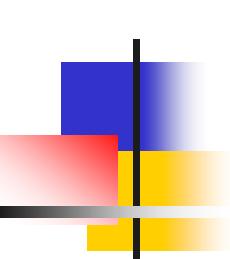
3 处理死锁的基本方法

- 1) 预防死锁
- 2) 避免死锁
- 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。此方法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，及时的检测出死锁的发生，并精确的确定与死锁有关的进程和资源；然后采取适当的措施，从系统中将已发生的死锁清除掉。

4) 解除死锁

这是与死锁检测相配套的一种措施。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销或挂起一些进程，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。死锁的检测与解除措施，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

- 预防死锁和避免死锁这两种方法，实质上都是通过施加某些限制条件的方法，来预防发生死锁。

两者的主要区别在于：为预防死锁所施加的限制条件较严格，这往往会影响进程的并发执行，而为避免死锁所施加的限制条件则较宽松，这给进程的运行提供了较为宽松的环境，有利于进程的并发执行。

3.6 预防死锁

- 预防死锁的方法是使四个必要条件中的第**2、3、4**条件之一不能成立，来避免发生死锁。至于必要条件**1**，因为它是由设备的固有条件所决定的，不仅不能改变，还应加以保证。下面分别对三种情况加以说明：

1. 破坏“请求和保持”条件

- 采用这种方法时，系统规定所有进程在开始运行之前，都必须一次性的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就分配给该进程，该进程在运行期间不会提出资源要求，从而摒弃了“请求”条件。若系统没有足够资源分配给它，就让该进程等待。因而也摒弃了“保持”条件，从而可避免发生死锁。
- 优点是算法简单、易于实现且很安全。但缺点是资源浪费严重和进程延迟运行。

3.6 预防死锁

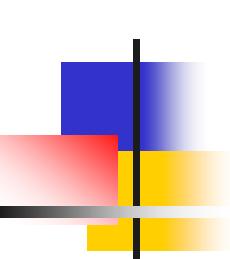
1. 破坏“请求和保持”条件

2. 破坏“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，提出新的要求不能被满足时必须释放它已经保持的所有资源，待以后需要时再重新申请。从而摒弃了“不剥夺”条件。
- 该方法实现起来比较复杂且付出很大代价。可能会造成前功尽弃，反复申请和释放等情况。

3. 破坏“环路等待”条件

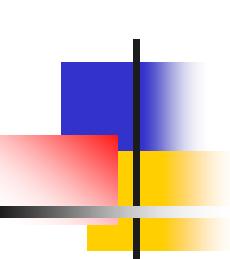
- 此方法规定，系统将所有资源按类型进行线性排队，并赋予不同的序号。所有进程对资源的请求必须严格按照资源序号递增的次序提出，这样在所形成的资源分配图中，不可能会出现环路，因而摒弃了“环路等待”条件。
- 与前两种策略比较，资源利用率和系统吞吐量都有较明显的改善。但也存在严重问题：为资源编号限制新设备的增加；进程使用设备顺序与申请顺序相反；限制用户编程自由。



3.7 避免死锁

3.7.1 系统安全状态

3.7.2 利用银行家算法避免死锁



3.7.1 系统安全状态

- 在预防死锁的几种方法中，都施加了较强的限制条件；
- 为避免死锁的方法中，所施加的限制条件较弱，有可能获得令人满意的系统性能。在该方法中把系统的状态分为安全状态和不安全状态，只要能使系统始终都处于安全状态，便可避免发生死锁。

3.7.1 系统安全状态

安全状态

- 安全状态，是指系统能按某种进程顺序（ $P_0, P_1, P_2, \dots, P_n$ ）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。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。
- 虽然并不是所有的不安全状态都是死锁状态，但当系统进入不安全状态后，便可能进而进入死锁状态；反之只要处于安全状态就不会死锁。
- 避免死锁的实质是：系统在进行资源分配时，设法使系统不进入不安全状态。

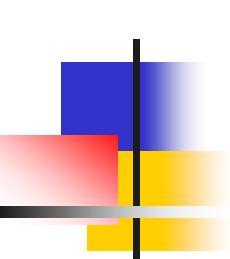
3.7.1 系统安全状态

安全状态之例

假定系统中有三个进程**A**、**B**和**C**，共有**12**台磁带机。进程**A**总共要求**10**台，**B**和**C**分别要求**4**台和**9**台。假设在**T₀**时刻，进程**A**、**B**和**C**已分别获得**5**台、**2**台和**2**台，尚有**3**台未分配，如下表所示：

进程	最大需求	已分配	可用
A	10	5	3
B	4	2	
C	9	2	

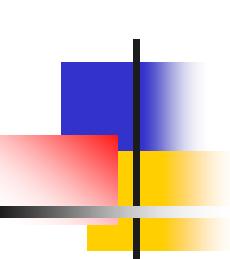
- 经分析发现，在**T₀**时刻系统是安全的，因此时存在一个安全序列**<B,A,C>**，即只要系统按此进程序列分配资源，就能使每个进程都顺利完成。可验证。
- 如果不按照安全序列分配资源，则系统可能会由安全状态进入不安全状态。例如，在**T₀**时刻后，**C**又请求到一台磁带机，若此时系统把剩余**3**台中的**1**台分配给**C**，则系统便进入不安全状态。因为，此时再也无法找到一个安全序列，结果导致死锁。



3.7 避免死锁

3.7.1 系统安全状态

3.7.2 利用银行家算法避免死锁



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。这是由于该算法能用于银行系统现金贷款的发放而得名的。

3.7.2 利用银行家算法避免死锁

1. 银行家算法中的数据结构 设系统中有m类资源，n个进程

(1) 可利用资源向量**Available**。含有m个元素的一维数组，每个元素代表一类可利用的资源数目，初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与回收而动态的改变。**Available[j]=K**, 表示系统中现有R_j类资源K个。

(2) 最大需求矩阵**Max**。为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类资源的最大需求数。**Max[i,j]=K**, 表示进程i需要R_j类资源的最大数目为K。

(3) 分配矩阵**Allocation**。为n*m的矩阵，它定义了系统中每类资源当前已分配给每一进程的资源数。**Allocation[i,j]=K**, 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**。为n*m的矩阵，表示每个进程尚需的各类资源数。**Need[i,j]=K**, 表示进程i还需要R_j类资源K个，方能完成其任务。

三个矩阵间的关系：**Need[i,j]=Max[i,j]-Allocation[i,j]**

2. 银行家算法

Request_i为含有m个元素的一维数组

设**Request_i**是进程**P_i**的请求向量。**Request_i[j]=K**, 表示**P_i**需**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**需等待。
- (3) 系统试探着把资源分配给进程**P_i**, 并修改下面数据结构中的
数值: **Available[j]:= Available[j]- Request_i[j];**
Allocation[i,j]:=Allocation[i,j]+Request_i[j];
Need[i,j]:=Need[i,j]-Request_i[j];
- (4) 系统执行**安全性算法**, 检查此次资源分配后系统是否出于安
全状态以决定是否完成本次分配。
- (5) 若**安全**, 则分配资源;

否则, 不分配资源**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];

P_i需等待

3. 安全性算法

(1) 设置两个向量：

工作向量**work**: 表示系统可提供给进程继续运行所需的各类资源数目，**含有m个元素的一维数组**，初始时，**work:=Available**；

Finish: **含n个元素的一维数组**，表示系统是否有足够的资源分配给n个进程，使之运行完成。开始时先令**Finish[i]:=false** ($i=1..n$)；当有足够的资源分配给进程*i*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：

Finish[i]=false; Need[i,j]<=work[j]; 若找到，执行步骤(3)，否则执行步骤(4)。

(3) 当进程P_j获得资源后，可顺利执行，直至完成，并释放出分配给它的资源，故应执行：

work[j]:= work[i]+ Allocation[i,j] ;

Finish[i]:=true;

goto 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**。

3.7.2 利用银行家算法避免死锁

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。

系统中 T_0 时刻的资源分配情况:

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	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			

考察系统在 T_0 时刻的安全性?

T_0 时刻的安全性：用安全性算法对 T_0 时刻的资源分配情况进行分析，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

进程 资源情况	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			

P_1 发出资源请求向量**Request₁(1,0,2)**，系统按银行家算法检查：

- (1) **Request₁(1,0,2) \leq Need₁(1,2,2)**
- (2) **Request₁(1,0,2) \leq Available₁(3,3,2)**
- (3) 系统先假定可为 P_1 分配资源，并修改向量**Available**, **Allocation₁**, **Need₁**
- (4) 再利用安全性算法检查此时系统是否安全。如下表：

由安全性检查得知，能找到一个安全序列 $\{P_1, P_3, P_4, P_0, P_2\}$ ，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。

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

P1发出的资源请求向量Request1(1,0,2)可以得到满足。
给P1分配Request1(1,0,2)资源请求后，系统的资源分配请求关系变为：

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			

- P4请求资源： P4发出请求向量Request4(3,3,0), 系统按银行家算法进行检查：
 - (1) Request4(3,3,0) \leq Need4(4,3,1);
 - (2) Request4(3,3,0) \leq Available(2,3,0), 让P4等待。

P1发出的资源请求向量Request₁(1,0,2)可以得到满足。
给P1分配Request₁(1,0,2)资源请求后，系统的资源分配请求关系变为：

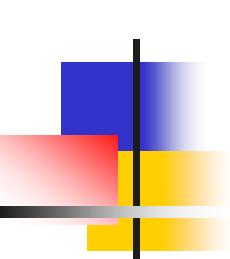
资源情况 进程	Max			Allocation			Need			Available		
	A	B	C	A	B	C	A	B	C	A	B	C
P ₀	7	5	3	0	1	0	7	4	3	2	3	0
P ₁	3	2	2	3	0	2	1	2	2			
P ₂	9	0	2	3	0	2	6	0	0			
P ₃	2	2	2	2	1	1	0	1	1			
P ₄	4	3	3	0	0	2	4	3	1			

- P₀请求资源：P₀发出请求向量Request₀(0,2,0),系统按银行家算法进行检查：
 - (1) Request₀(0,2,0)≤Need₀(7,4,3);
 - (2) Request₀(0,2,0)≤Available(2,3,0);
 - (3) 系统暂时先假定可为P₀分配资源，并修改有关数据，如下表所示：

P0发出请求向量**Request₀(0,2,0)**, 系统暂时先假定可为P0分配资源，修改数据结构内的相应值，变为：

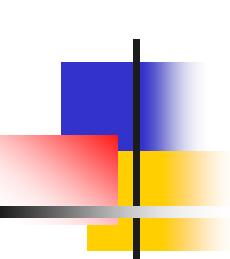
- 进行安全性检查：可用资源**Available(2,1,0)**已不能满足任何进程的需要，故系统进入不安全状态，此时系统不分配资源。
- 如果在银行家算法中，把**P₀**发出的请求向量改为**Request₀(0,1,0)**, 系统是否能将资源分配给它，请考虑。

资源情况 进程	Alloc ation	Need	Avail able
	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	



第三章 处理机调度与死锁

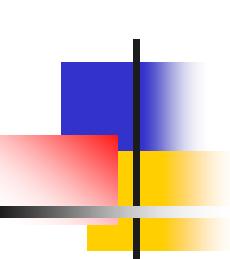
- 3.1 处理机调度的层次和调度算法的目标**
- 3.2 作业与作业调度**
- 3.3 进程调度**
- 3.4 实时调度**
- 3.5 死锁的概述**
- 3.6 预防死锁**
- 3.7 避免死锁**
- 3.8 死锁的检测与解除**



3.8 死锁的检测与解除

3.8.1 死锁的检测

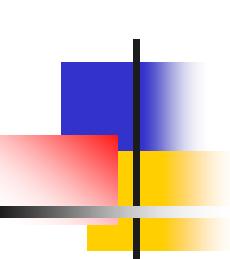
3.8.2 死锁的解除



3.8 死锁的检测与解除

3.8.1 死锁的检测

3.8.2 死锁的解除



3.8.1 死锁的检测

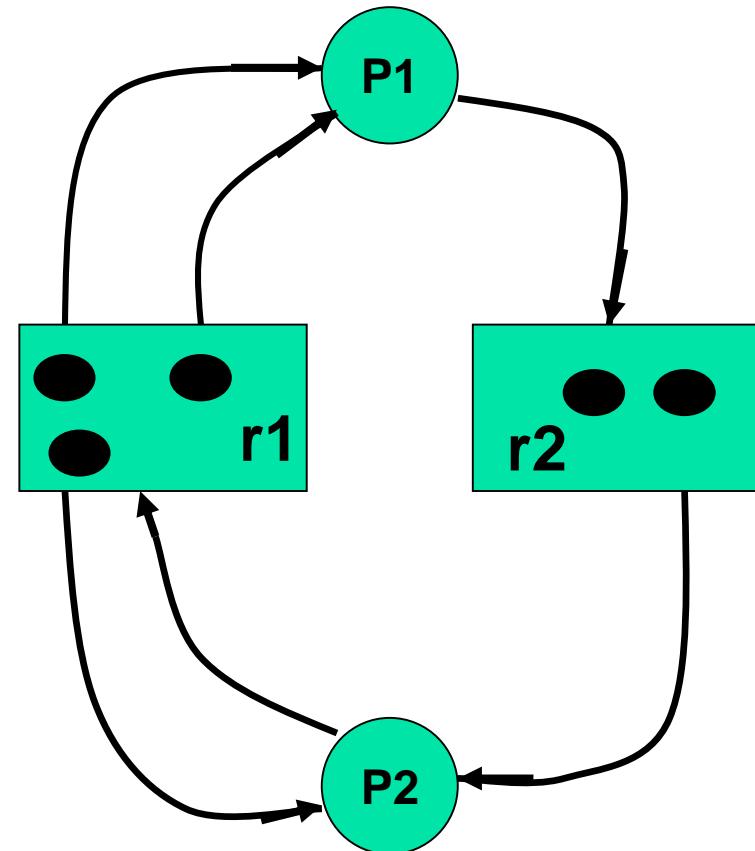
- 当系统为进程分配资源时，若未采取任何限制性措施，则系统**必须**提供检测和解除死锁的手段，为此系统必须：
 - 保存有关资源的请求和分配信息；
 - 提供一种算法，以利用这些信息来检测系统是否已进入死锁状态。

3.8.1 死锁的检测

资源分配图

- 系统死锁可利用资源分配图来描述。

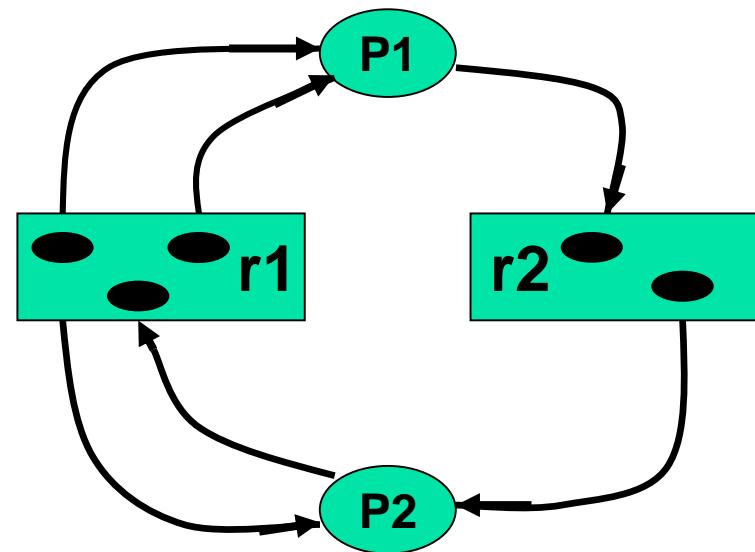
该图由表示进程的圆圈和表示一类资源的方框组成，其中方框中的一个点代表一个该类资源，请求边是由进程指向方框中的 r_j ，而分配边则应始于方框中的一个点。如图所示。

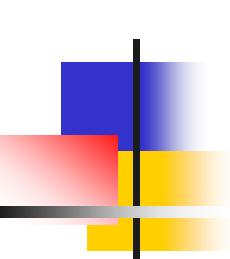


3.8.1 死锁的检测

死锁定理

- 我们可以利用资源分配图加以简化的方法，来检测系统处于某状态时是否为死锁状态。简化方法如下：
 - 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 ，在顺利的情况下运行完毕，释放其占有的全部资源。
 - 由于释放了资源，这样能使其它被阻塞的进程获得资源继续运行。
 - 在经过一系列简化后若能消去图中的所有的边，使所有进程结点都孤立，则称该图是可完全简化的，反之是不可完全简化的。
- 可以证明： **S** 状态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。 <**死锁定理**>

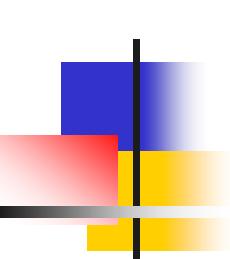




3.8 死锁的检测与解除

3.8.1 死锁的检测

3.8.2 死锁的解除



3.8.2 死锁的解除

- 当发现进程死锁时，便应立即把它们从死锁状态中解脱出来。常采用的方法是：
 1. **剥夺资源**：从其他进程剥夺足够数量的资源给死锁进程以解除死锁状态。
 2. **撤销进程**：最简单的是让全部进程都死掉；温和一点的是按照某种顺序逐个撤销进程，直至有足够的资源可用，使死锁状态消除为止。