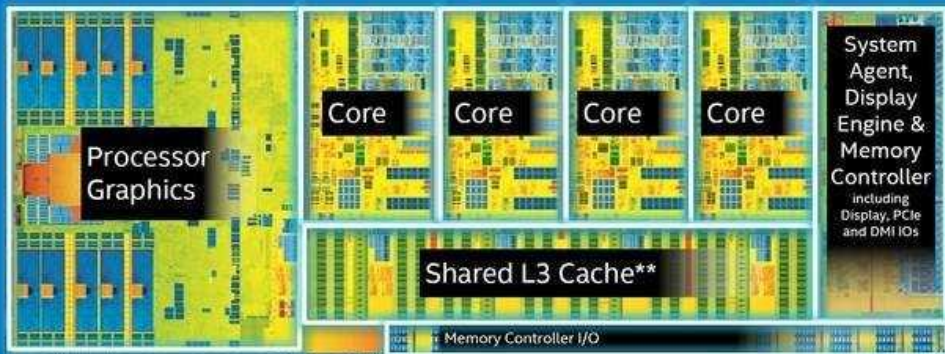


4th Gen Intel® Core™ Processor Die Map 22nm Tri-Gate 3-D Transistors



Quad core die shown above

Transistor count: 1.4 Billion

Die size: 177mm²

中关村在线
ZOL.com.cn

** Cache is shared across all 4 cores and processor graphics

2024

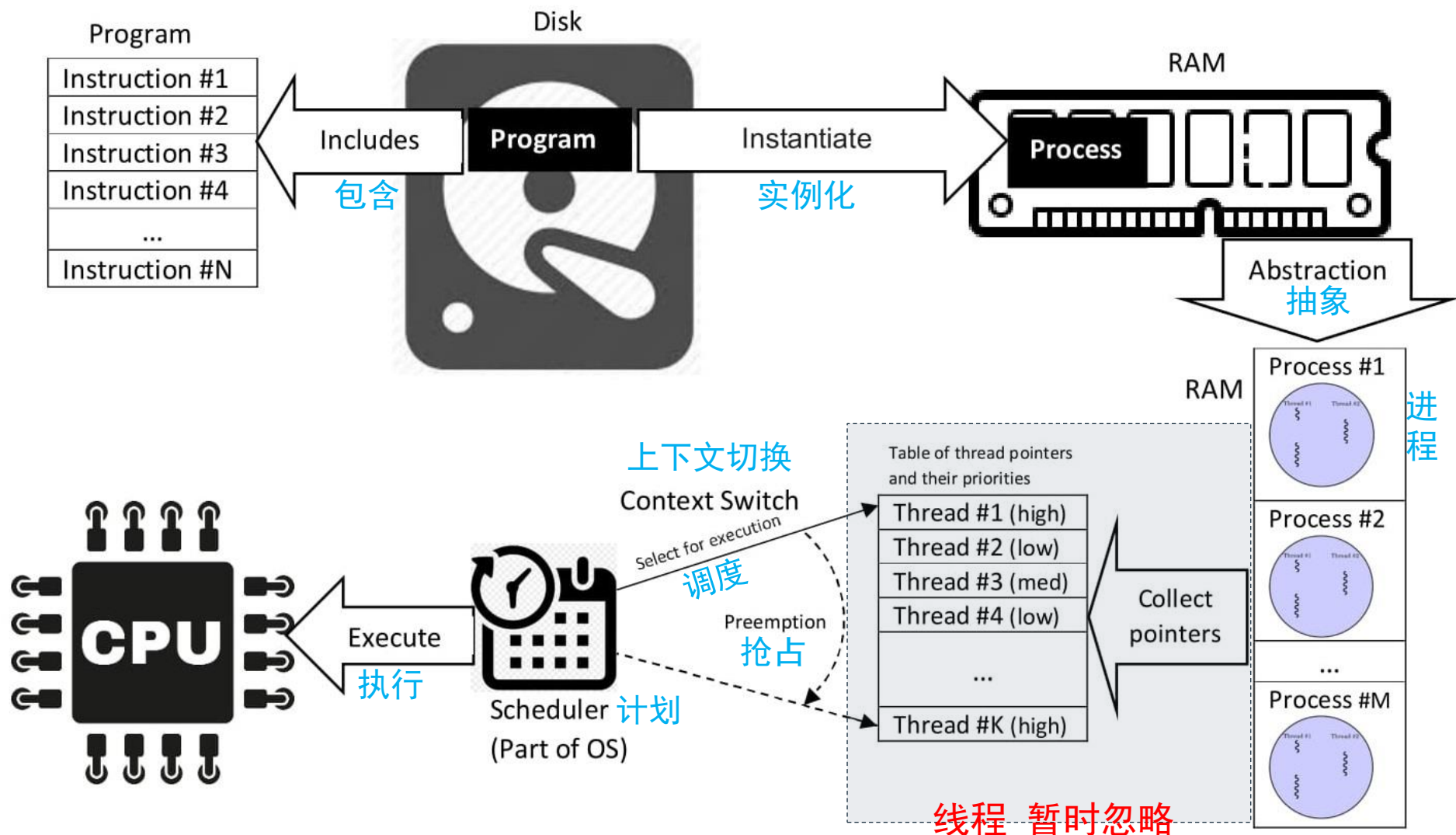
操作系统A

第三章 处理机调度与死锁

赵谷雨\ 信息科学与工程学院

Email: gaiazhao@ysu.edu.cn

程序的执行



这一系列的过程都是建立在【进程】上来实现的。

目录

CONTENTS

1

处理机调度的层次和调度算法的目标

2

作业与作业调度

3

进程调度

4

实时调度

5

死锁概述

6

预防死锁

7

避免死锁

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

问题：作业从提交到完成的处理机调度过程

目标：

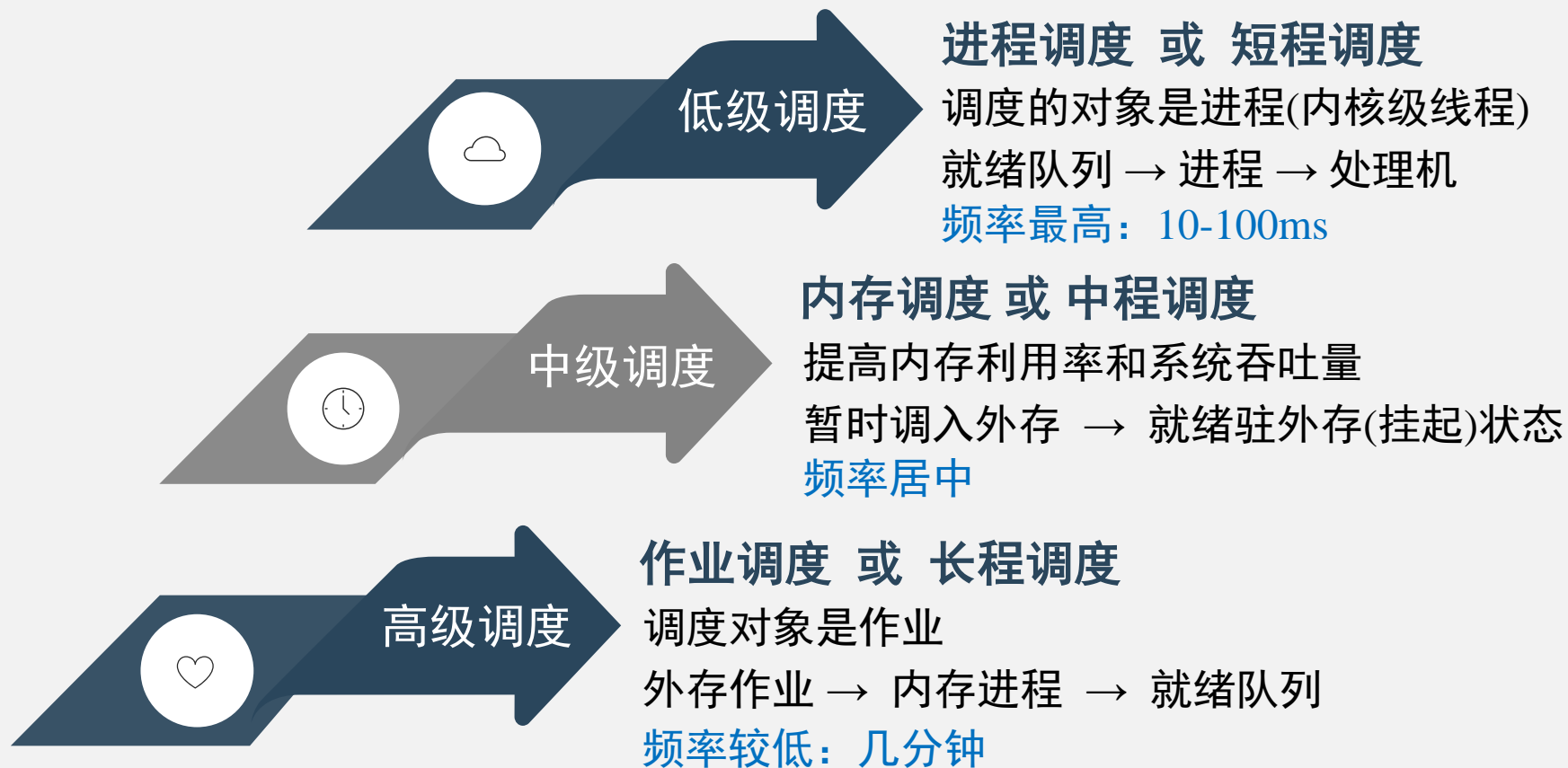
- 掌握处理机调度的层次
- 了解调度队列模型
- 掌握处理机调度算法

3.1 处理机调度的层次和调度算法的目标

在多道程序系统中，调度的实质是一种资源分配，处理机调度是对处理机资源进行分配。处理机调度算法是指根据处理机分配策略所规定的处理机分配算法。

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

3.1.1 处理机调度的层次



3.1.1 处理机调度的层次



进程调度 或 短程调度

调度的对象是进程(内核级线程)

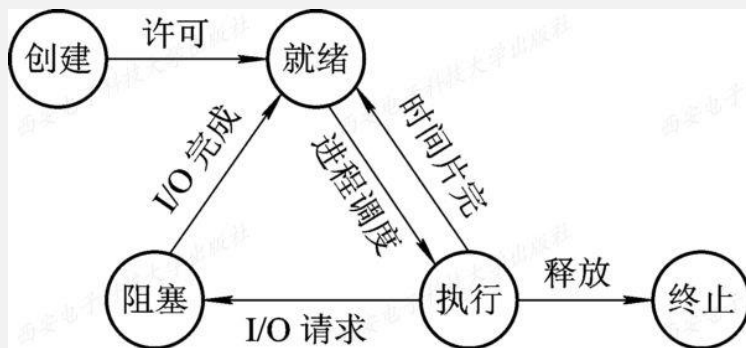
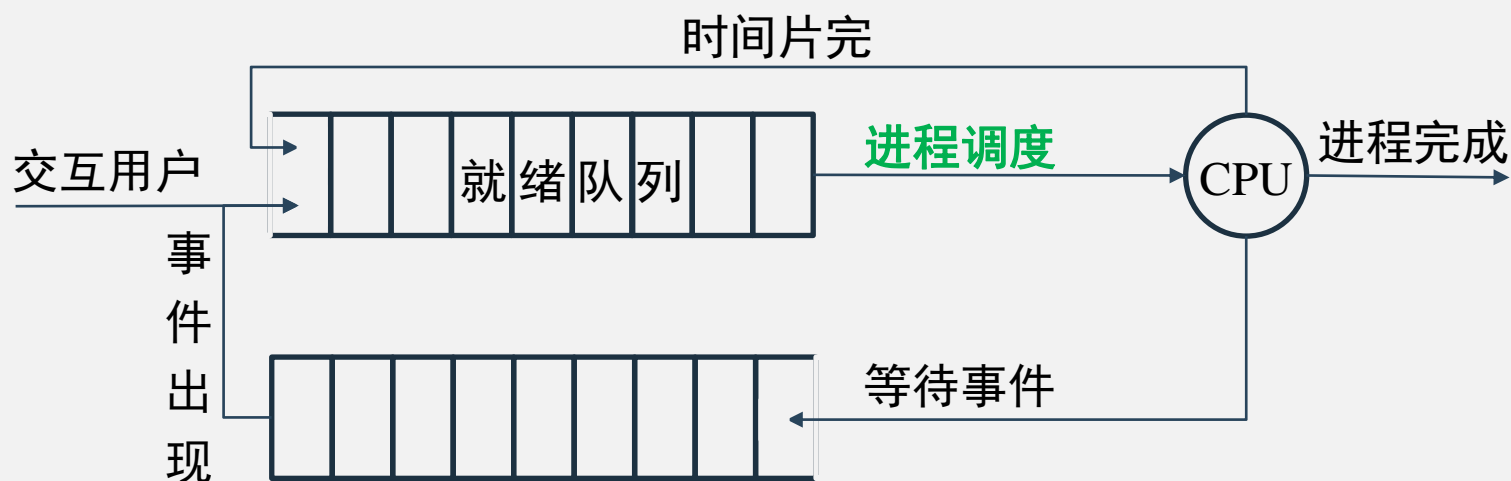
就绪队列 → 进程 → 处理机

频率最高: 10-100ms

主要功能是，根据某种算法，决定就绪队列中哪个进程应获得处理机，并由分派程序将处理机分派给被选中的进程。进程调度是最基本的调度，在多道批处理、分时和实时三种类型的OS中，都必须配置这级调度。

3.1.1 处理机调度的层次

仅有进程调度的调度队列模型



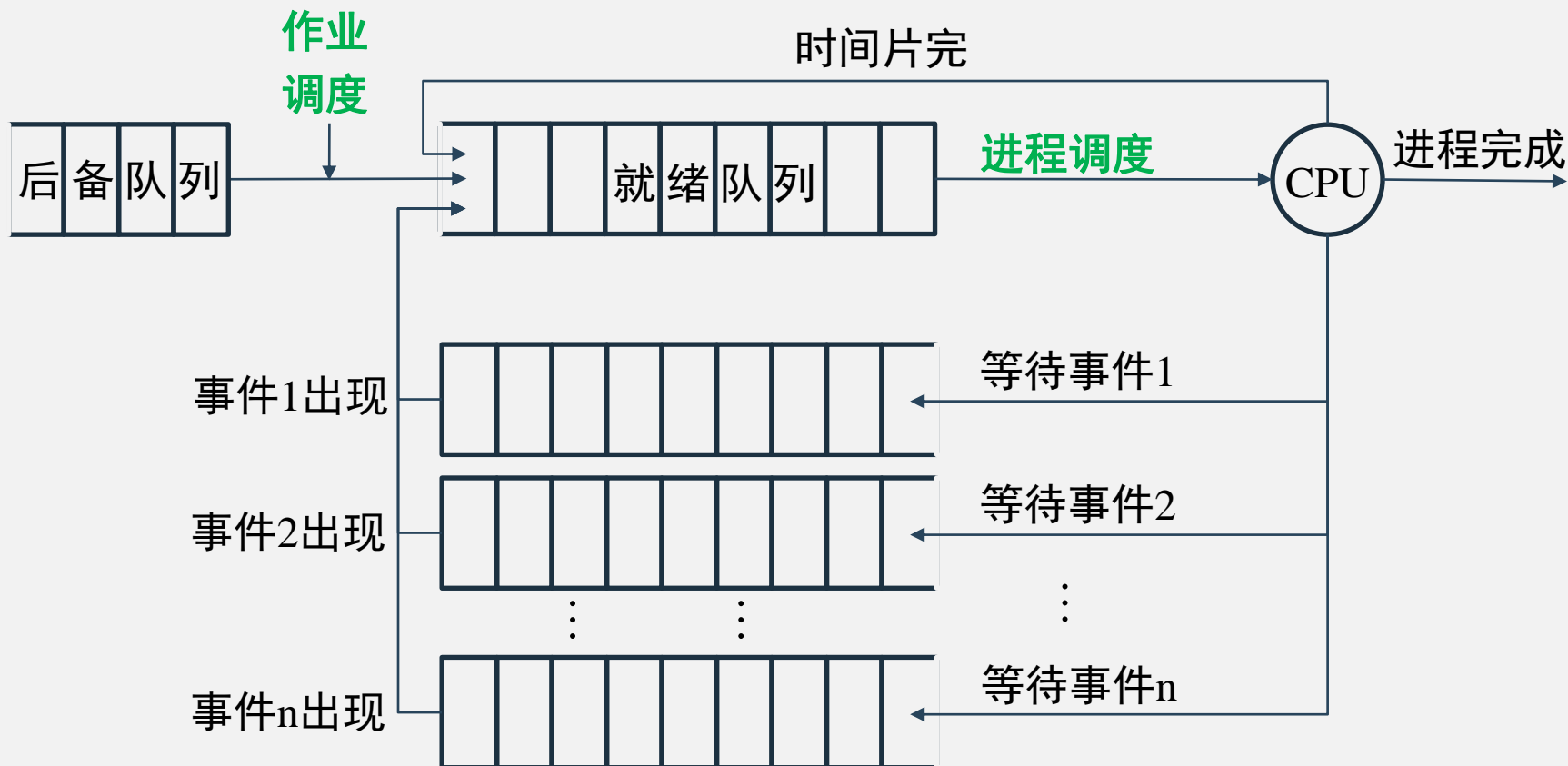
3.1.1 处理机调度的层次



主要功能是根据某种算法，决定将外存上处于后备队列中的哪几个作业调入内存，为它们创建进程、分配必要资源，并将它们放入就绪队列。高级调度主要用于多道批处理系统中，而在分时和实时系统中不设置高级调度。

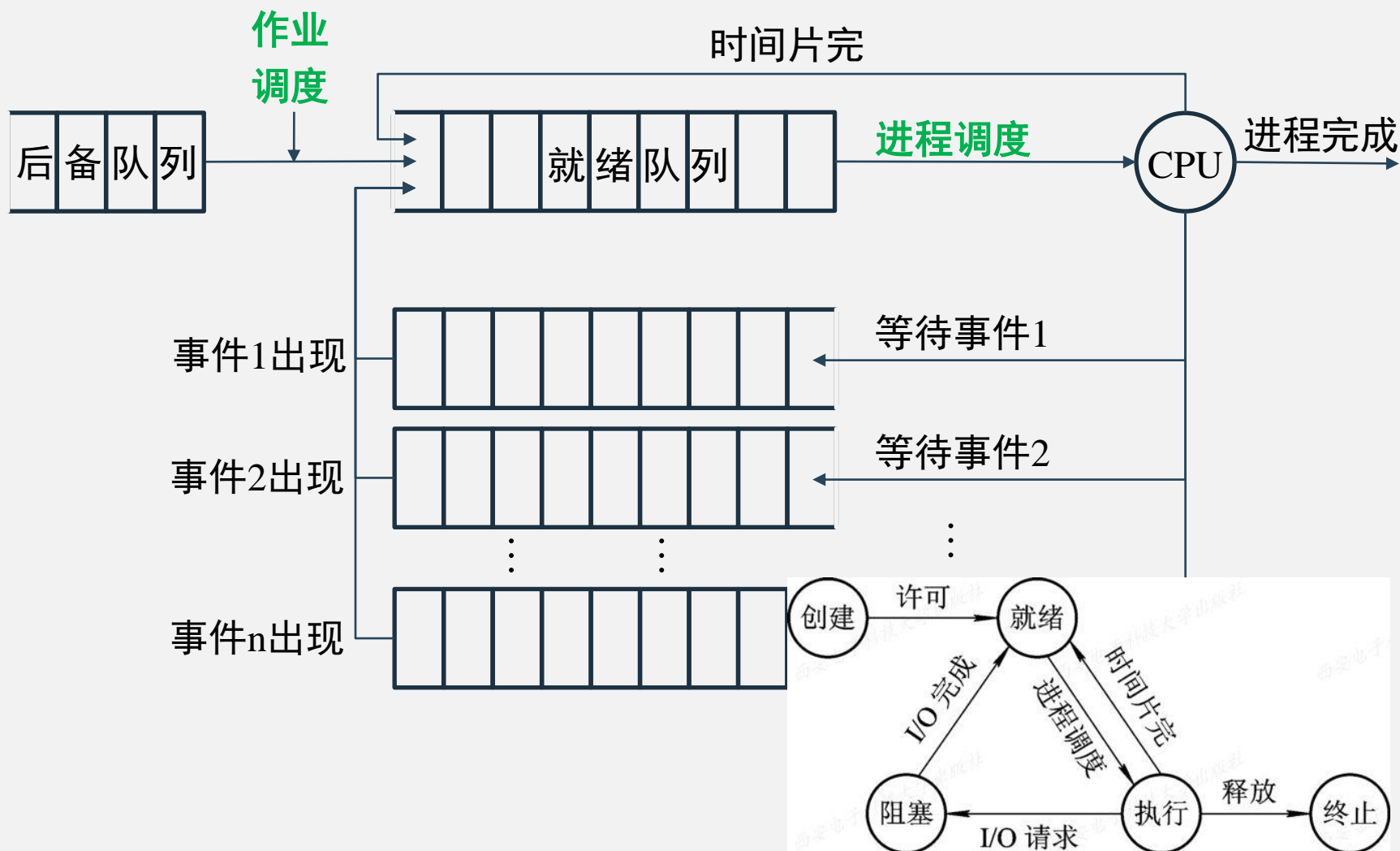
3.1.1 处理机调度的层次

仅有高级和低级调度的调度队列模型

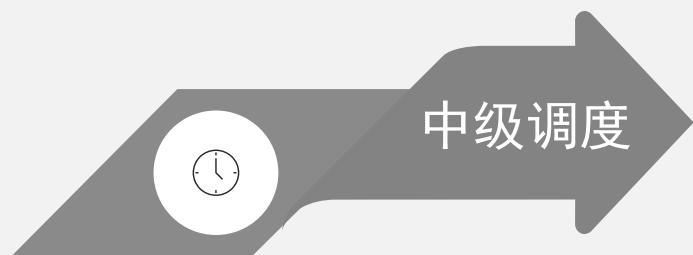


3.1.1 处理机调度的层次

仅有高级和低级调度的调度队列模型



3.1.1 处理机调度的层次



内存调度 或 中程调度

提高内存利用率和系统吞吐量

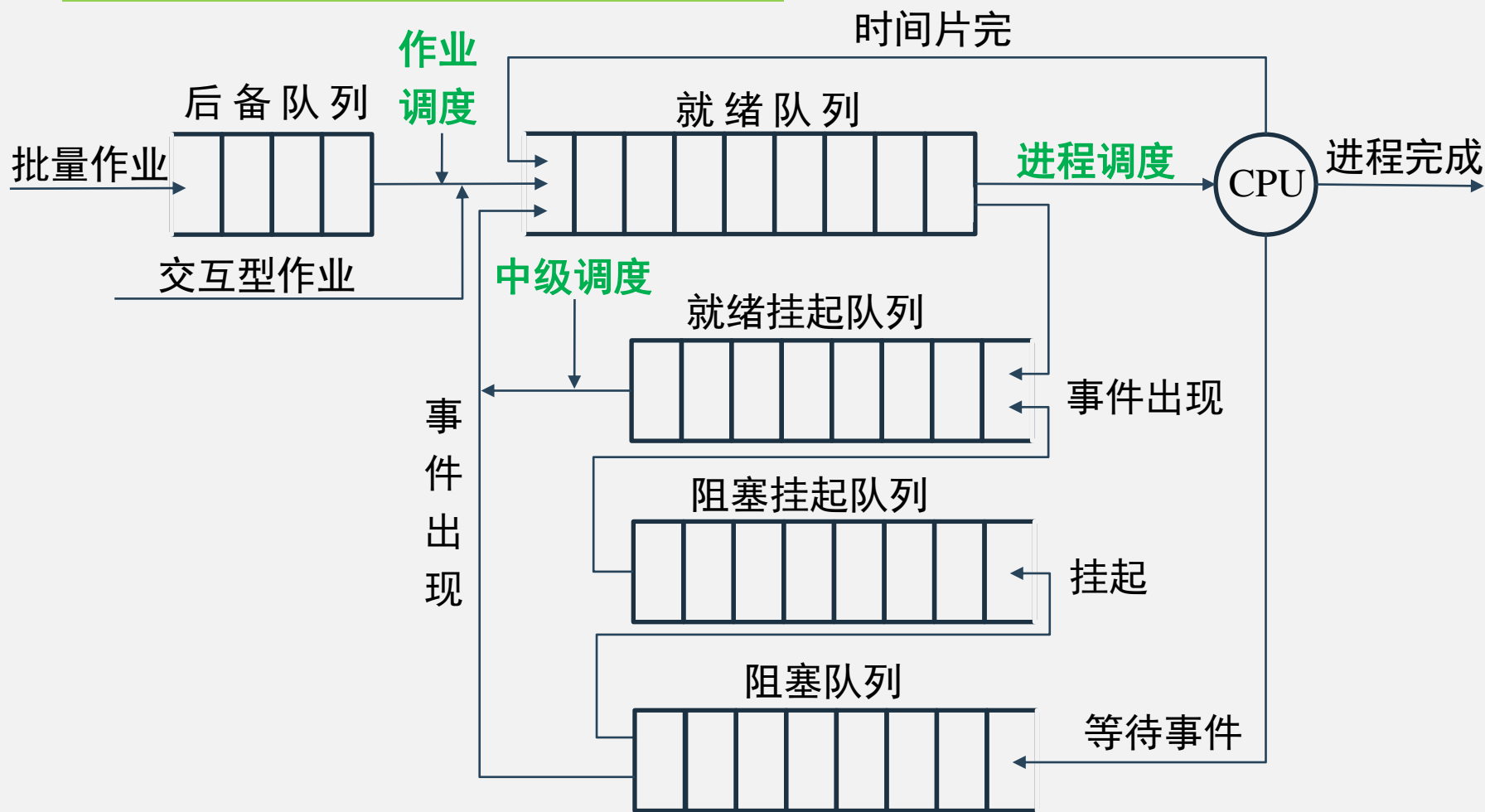
暂时调入外存 → 就绪驻外存(挂起)状态

频率居中

主要目的是，提高内存利用率和系统吞吐量。把暂时不能允许的进程，调至外存等待，进程的状态为就绪驻外存状态(或挂起状态)。当它们已具备运行条件且内存稍有空闲时，中级调度把外存上的已具备运行条件的就绪进程重新调入内存，并修改为就绪状态，挂在就绪队列等待。
中级调度实际上就是存储器管理中的对换功能(第四章)。

3.1.1 处理机调度的层次

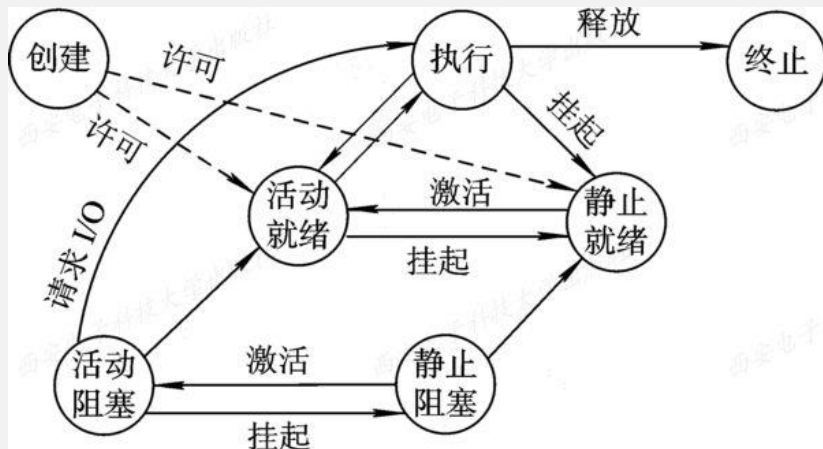
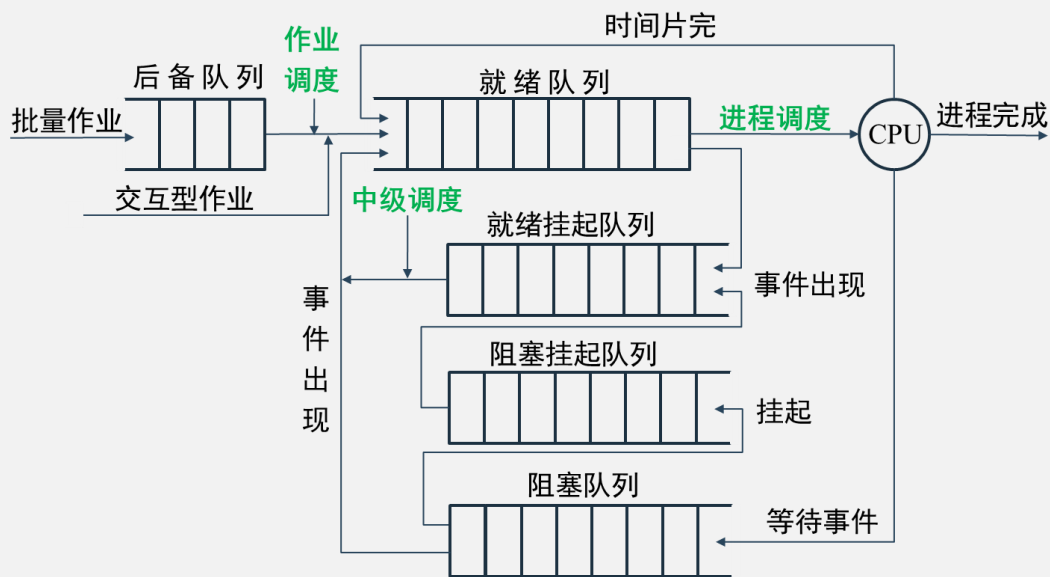
具有三级调度的调度队列模型



3.1.1 处理机调度的层次

3.1 处理机调度的层次和调度算法的目标

具有三级调度的调度队列模型



3.1.2 处理机调度算法的目标

3.1 处理机调度的层次和调度算法的目标

1. 处理机调度算法的共同目标

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

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

3.1.2 处理机调度算法的目标

3.1 处理机调度的层次和调度算法的目标

1. 处理机调度算法的共同目标

- (2) **公平性**。公平性是指应使**诸进程**都获得合理的CPU 时间，不会发生进程饥饿现象。公平性是相对的，对相同类型的进程应获得相同的服务；但对于不同类型的进程，由于其紧急程度或重要性的不同，则应提供不同的服务。
- (3) **平衡性**。由于在系统中可能具有多种类型的进程，有的属于计算型作业，有的属于I/O型。为使系统中的**CPU和各种外部设备**都能经常处于忙碌状态，调度算法应尽可能保持系统资源使用的平衡性。
- (4) **策略强制执行**。对所制订的策略其中包括安全策略，只要需要，就必须予以准确地执行，即使会造成某些工作的延迟也要执行。

3.1.2 处理机调度算法的目标

3.1 处理机调度的层次和调度算法的目标

2. 批处理系统的目标

(1) 平均周转时间短

周转时间，是指从作业提交给系统开始，到作业完成为止这段时间间隔。

1. 作业在外存后备队列上等待(作业)调度的时间
2. 进程在就绪队列上等待进程调度的时间
3. 进程在CPU上执行的时间
4. 进程等待I/O操作完成的时间

后三项在一个作业的整个处理过程中，可能发生多次。

3.1.2 处理机调度算法的目标

3.1 处理机调度的层次和调度算法的目标

2. 批处理系统的目标

(1) 平均周转时间短

每个用户，都希望自己的作业周转时间最短。计算机系统的管理者，希望能使平均周转时间最短，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。应使作业周转时间和作业的平均周转时间尽可能短。否则，会使许多用户的等待时间过长，这将会引起用户特别是短作业用户的不满。可把平均周转时间描述为：

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

3.1.2 处理机调度算法的目标

3.1 处理机调度的层次和调度算法的目标

2. 批处理系统的目标

(1) 平均周转时间短

为了进一步反映调度的性能，更清晰地描述各进程在其周转时间中，等待和执行时间的具体分配状况，往往使用带权周转时间，即作业的周转时间 T 与系统为它提供服务的时间 T_s 之比，即 $W = T/T_s$ 。而平均带权周转时间则可表示为：

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

3.1.2 处理机调度算法的目标

3.1 处理机调度的层次和调度算法的目标

2. 批处理系统的目标

(2) 系统吞吐量高

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有关。事实上，如果单纯是为了获得高的系统吞吐量，应尽量多地选择短作业运行。

(3) 处理机利用率高

对于大、中型计算机，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；而调度方式和算法又对处理机的利用率起着十分重要的作用。如果单纯是为使处理机利用率高，应尽量多地选择计算量大的作业运行。

由上所述可以看出，这些要求之间是存在着一定矛盾的。

3.1.2 处理机调度算法的目标

3.1 处理机调度的层次和调度算法的目标

3. 分时系统的目标

(1) 响应时间快

响应时间快是选择分时系统中进程调度算法的重要准则。所谓**响应时间**，是从用户通过键盘提交一个请求开始，直到屏幕上显示出处理结果为之的一段时间间隔，包括三部分：

- ① 请求信息从键盘输入开始，直至将其传送到处理机的时间。
- ② 处理机对请求进行处理的时间；
- ③ 将所形成的响应回送到终端显示器的时间。

(2) 均衡性

所谓均衡性，是指系统响应时间的快慢应与用户所请求服务的复杂性相适应。对复杂任务允许响应时间较长，而简单任务需要响应时间要短。

3.1.2 处理机调度算法的目标

3.1 处理机调度的层次和调度算法的目标

4. 实时系统的目标

(1) 截止时间的保证

截止时间，是指某个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。对实时系统而言，调度算法的一个主要目标是保证实时任务对截止时间的要求。

(2) 可预测性

在实时系统中，可预测性显得非常重要。通过预测下一任务的执行要求，可对某些步骤或操作采取并行处理方式，从而提高实时性。

3.1.2 处理机调度算法的目标

1. 面向用户的准则

- 周转时间短(评价批处理的系统的指标)
 周转时间； 平均周转时间
 带权周转时间； 平均带权周转时间
- 响应时间快：响应时间(评价分时系统的指标)
- 截止时间的保证：截止时间(评价实时系统的指标)
- 优先权准则

3.1.2 处理机调度算法的目标

2. 面向系统的准则

- 系统吞吐量高(评价批处理系统的指标)
- 处理机利用率好
- 各类资源的平衡利用

第三章 处理机调度与死锁

3.2 作业与作业调度

问题： 如何从后备队列中调度作业

目标：

- 掌握批处理系统中作业管理的方法
- 理解作业调度的主要任务
- 掌握作业调度的4个方法

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

3.2.1 批处理系统中的作业

1. 作业与作业步

作业

作业(Job)是一个比程序更为广泛的概念，不仅包含通常的程序和数据，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。

作业步

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而又相互关联的顺序加工步骤才能得到结果。我们把其中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，往往是上一个作业步的输出作为下一个作业步的输入。

一个典型作业可分为：“编译”作业步，“链接”作业步和“运行”作业步。

3.2.1 批处理系统中的作业

2. 作业控制块(Job Control Block, JCB)

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JCB**，它是作业在系统中存在的标志，其中保存了系统对作业进行管理和调度所需的全部信息。

通常在JCB中包含的内容有：**作业标识、用户名称、用户账号、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、作业运行时间)、资源需求(预计运行时间、要求内存大小等)、资源使用情况等。**

3.2.1 批处理系统中的作业

2. 作业控制块(Job Control Block, JCB)

每当一个作业**进入系统时**，便由“作业注册”程序为该作业建立一个作业控制块JCB。

根据作业类型，将它放到相应的作业后备队列中等待调度。调度程序依据一定的调度算法来调度它们，被调度到的作业将被装入内存。在作业运行期间，系统按照JCB中信息和作业说明书对作业进行控制。

当一个作业执行结束进入完成状态时，系统负责回收已分配给它的资源，撤销该作业控制块。

3.2.1 批处理系统中的作业

3. 作业运行的三个阶段和三种状态

作业从进入系统到运行结束，通常需要三个阶段。

(1) **收容阶段**：用户输入，建立JCB，放入作业后备队列——**后备状态**

(2) **运行阶段**：从第一次进入就绪状态(作业调度后，为其分配资源，建立进程，放入就绪队列)开始，直到它运行结束前。——**运行状态**

(3) **完成阶段**：作业运行完成、异常结束，“终止程序”回收已分配给该作业的JCB和所有资源，运行结果形成输出文件输出。——**完成状态**

3.2.2 作业调度的主要任务

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

因此，作业调度称为接纳调度(Admission Scheduling)。在每次执行作业调度时，都需做出以下两个决定。

1. 接纳多少个作业——多道程序度
2. 接纳哪些作业——调度算法

3.2.3 先来先服务和短作业优先调度算法

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

FCFS是最简单的调度算法，该算法既可用于作业调度，也可用于进程调度。

作业调度，系统将按照作业到达的先后次序来进行调度，优先考虑在系统中等待时间最长的作业，不管该作业所需执行时间的长短。从后备作业队列中选择几个最先进入该队列的作业，调入内存，为它们分配资源和创建进程，放入就绪队列。进程调度，每次调度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，使其投入运行，直到运行完成或发生某事件阻塞后，调度程序将处理机重新分配。

3.2.3 先来先服务和短作业优先调度算法

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

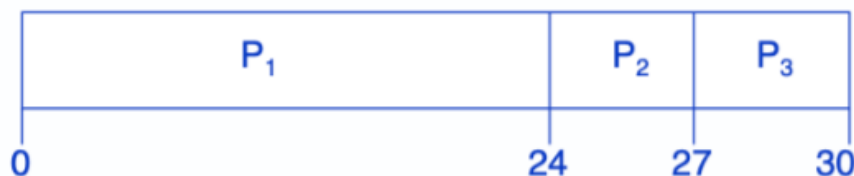
❖ 先来先服务 (First-Come, First-Served - FCFS)

按**进程请求CPU**的先后顺序使用CPU

❖ 举例:

进程	区间时间
P1	24
P2	3
P3	3

❖ 假定进程到达顺序如下: **P1** , **P2** , **P3** 该调度的Gantt (甘特) 图为:



❖ 周转时间: P1: 24; P2: 27; P3: 30; 平均周转时间: $(24 + 27 + 30)/3 = 27$

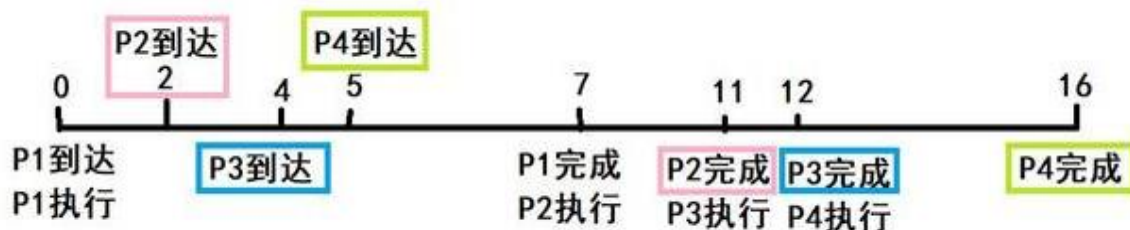
❖ 等待时间: P1: 0; P2: 24; P3: 27; 平均等待时间: $(0 + 24 + 27)/3 = 17$

3.2.3 先来先服务和短作业优先调度算法

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

先来先服务调度算法: FCFS

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



P1周转时间=完成时刻-到达时刻=7-0=7

P2周转时间=11-2=9

P3周转时间=12-4=8

P4周转时间=16-5=11

平均周转时间= (7+9+8+11) / 4 = 6.25

P1带权周转时间=周转时间/CPU提供服务的时间=7/7=1

P2带权周转时间=9/4=2.25

P3带权周转时间=8/1=8

P4带权周转时间=11/4=2.75

2. 短作业优先(short job first, SJF)的调度算法

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

1) 短作业优先算法

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

2. 短作业优先(short job first, SJF)的调度算法

2) 短作业优先算法的缺点

SJF调度算法较之FCFS算法有了明显的改进，但仍然存在不容忽视的缺点：

(1) 必须预知作业的运行时间。即使是程序员也很难准确估计作业的运行时间，如果估计过低，系统就可能按估计的时间终止作业的运行，但此时作业并未完成，故一般都会偏长估计。

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

3.2.3 先来先服务和短作业优先调度算法

2. 短作业优先(short job first, SJF)的调度算法

(3) 在采用SJF算法时，人—机无法实现交互。

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

3.2.4 优先级调度算法和高响应比优先调度算法

1. 优先级调度算法(priority-scheduling algorithm, PSA)

我们可以这样来看作业的优先级，对于先来先服务调度算法，**作业的等待时间就是作业的优先级**，等待时间越长，其优先级越高。对于短作业优先调度算法，作业的长短就是作业的优先级，作业所需运行的时间越短，其优先级越高。**在优先级调度算法中，则是基于作业的紧迫程度，由外部赋予作业相应的优先级，调度算法依据该优先级进行调度。**保证紧迫性作业优先运行。同样，即可作为作业调度算法，也可作为进程调度算法。

3.2.4 优先级调度算法和高响应比优先调度算法

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

在批处理系统中，FCFS算法所考虑的只是作业的等待时间，而忽视了作业的运行时间。而SJF算法正好与之相反，只考虑作业的运行时间，而忽视了作业的等待时间。**高响应比优先调度算法则是既考虑了作业的等待时间，又考虑作业运行时间的调度算法**，因此既照顾了短作业，又不致使长作业的等待时间过长，从而改善了处理机调度的性能。

3.2.4 优先级调度算法和高响应比优先调度算法

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

高响应比优先算法是如何实现的呢？

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

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

3.2.4 优先级调度算法和高响应比优先调度算法

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

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

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

第三章 处理机调度与死锁

3.3 进程调度

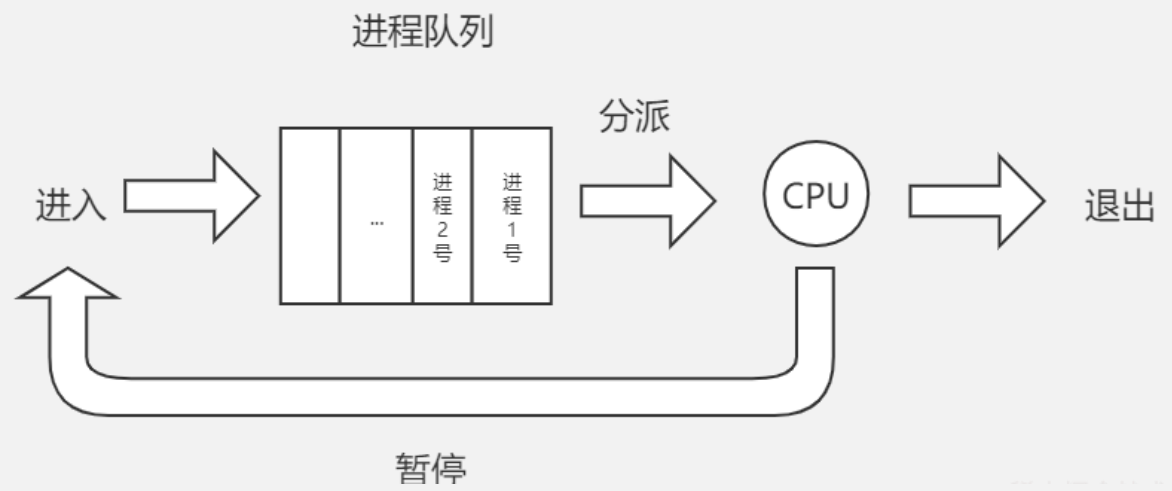
进程调度是OS中必不可少的一种调度。因此在三种类型的OS中，都无一例外地配置了**进程调度**。此外它也是对系统性能影响最大的一种处理机调度，相应的，有关进程调度的算法也较多。

3.3.1 进程调度的任务、机制和方式

1. 进程调度的任务

进程调度的任务主要有三：

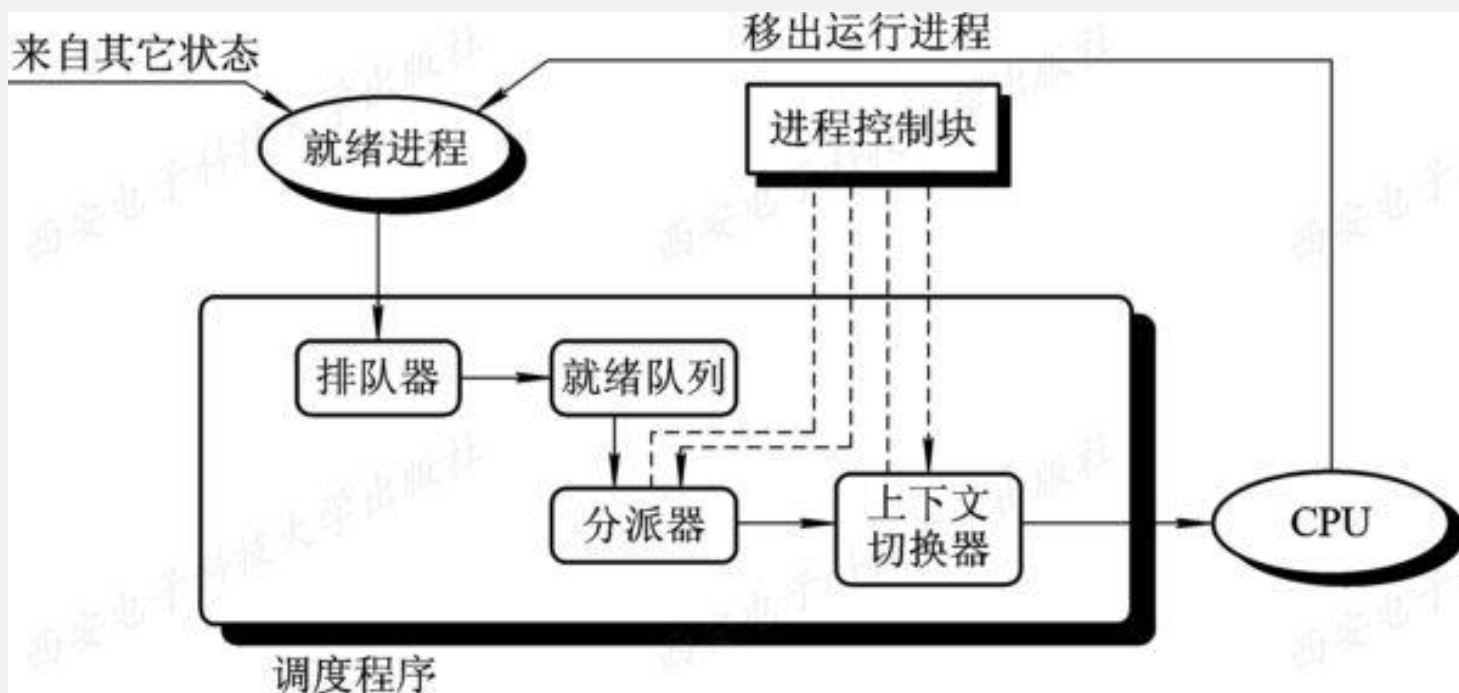
- (1) 保存处理机的现场信息。
- (2) 按某种算法选取进程。
- (3) 把处理器分配给进程。



2. 进程调度机制

为了实现进程调度，在进程调度机制中，应具有如下三个基本部分，如图3-1所示。

(1) 排队器。(2) 分派器。(3) 上下文切换器。



3.3.1 进程调度的任务、机制和方式

3. 进程调度方式

1) 非抢占方式(Nonpreemptive Mode)

在采用这种调度方式时，一旦把处理机分配给某进程后，就一直让它运行下去，决不会因为时钟中断或任何其它原因去抢占当前正在运行进程的处理机，直至该进程完成，或发生某事件而被阻塞时，才把处理机分配给其它进程。

不能用于分时系统和大多数实时系统。

3.3.1 进程调度的任务、机制和方式

3. 进程调度方式

2) 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，将已分配给该进程的处理机重新分配给另一进程。

在现代OS中广泛采用抢占方式：**批处理机系统**，可以防止一个长进程长时间地占用处理机，确保处理机能为所有进程提供更公平的服务。**分时系统**，只有采用抢占方式才有可能实现人—机交互。**实时系统**，抢占方式能满足实时任务的需求。但抢占方式比较复杂，所需付出的系统开销也较大。

3.3.1 进程调度的任务、机制和方式

3. 进程调度方式

2) 抢占方式(Preemptive Mode)

“抢占”不是一种任意行为，必须遵循一定的原则。

- **优先权原则：**

允许优先级高的新到进程抢占当前进程的处理机。

- **短进程优先原则**

允许新到的短进程可以抢占当前长进程的处理机。

- **时间片原则**

即各进程按照时间片轮转运行，正在执行进程的一个时间片用完后，便停止该进程执行而从新调度。

3.3.1 进程调度的任务、机制和方式

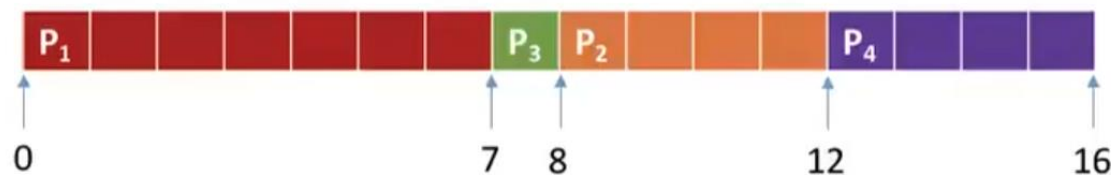
3. 进程调度方式

例题：各进程到达就绪队列的时间、需要的运行时间如下表所示。使用非抢占式的短作业优先调度算法，计算各进程的等待时间、平均等待时间、周转时间、平均周转时间、带权周转时间、平均带权周转时间。

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

短作业/进程优先调度算法：每次调度时选择当前已到达且运行时间最短的作业/进程。

因此，调度顺序为：P1 → P3 → P2 → P4



周转时间 = 完成时间 - 到达时间

$$P1=7-0=7; P3=8-4=4; P2=12-2=10; P4=16-5=11$$

带权周转时间 = 周转时间/运行时间

$$P1=7/7=1; P3=4/1=4; P2=10/4=2.5; P4=11/4=2.75$$

等待时间 = 周转时间 - 运行时间

$$P1=7-7=0; P3=4-1=3; P2=10-4=6; P4=11-4=7$$

平均周转时间 = $(7+4+10+11)/4 = 8$

平均带权周转时间 = $(1+4+2.5+2.75)/4 = 2.56$

平均等待时间 = $(0+3+6+7)/4 = 4$

8.75

3.5

4.75

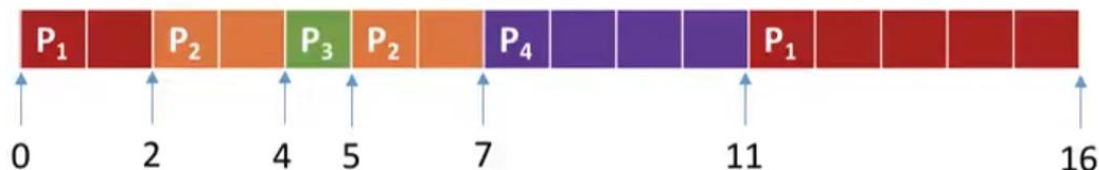
3.3.1 进程调度的任务、机制和方式

3. 进程调度方式

例题：各进程到达就绪队列的时间、需要的运行时间如下表所示。使用**抢占式的短作业优先**调度算法，计算各进程的等待时间、平均等待时间、周转时间、平均周转时间、带权周转时间、平均带权周转时间。

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

最短剩余时间优先算法：每当有进程加入**就绪队列**改变时就需要调度，如果新到达的进程**剩余时间**比当前运行的进程剩余时间**更短**，则由新进程**抢占**处理机，当前运行进程重新回到就绪队列。另外，当一个**进程完成**时也需要调度。



需要注意的是，当有新进程到达时就绪队列就会改变，就要按照上述规则进行检查。以下 $P_n(m)$ 表示当前 P_n 进程剩余时间为 m 。各个时刻的情况如下：

0时刻（P1到达）： $P_1(7)$

2时刻（P2到达）： $P_1(5)$ 、 $P_2(4)$

4时刻（P3到达）： $P_1(5)$ 、 $P_2(2)$ 、 $P_3(1)$

5时刻（P3完成且P4刚好到达）： $P_1(5)$ 、 $P_2(2)$ 、 $P_4(4)$

7时刻（P2完成）： $P_1(5)$ 、 $P_4(4)$

11时刻（P4完成）： $P_1(5)$

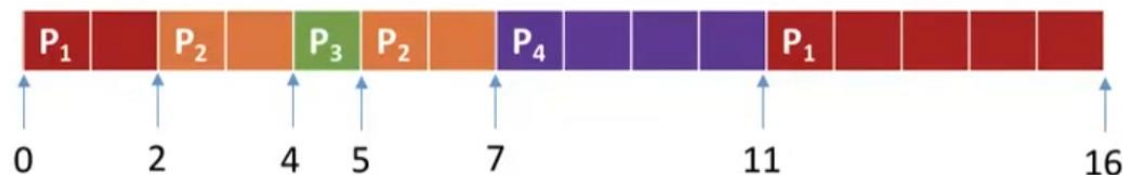
3.3.1 进程调度的任务、机制和方式

3. 进程调度方式

例题：各进程到达就绪队列的时间、需要的运行时间如下表所示。使用**抢占式的短作业优先**调度算法，计算各进程的等待时间、平均等待时间、周转时间、平均周转时间、带权周转时间、平均带权周转时间。

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

最短剩余时间优先算法：每当有进程加入就绪队列改变时就需要调度，如果新到达的进程**剩余时间**比当前运行的进程剩余时间**更短**，则由新进程**抢占**处理机，当前运行进程重新回到就绪队列。另外，当一个**进程完成时也需要调度**。



周转时间 = 完成时间 - 到达时间

$$P1=16-0=16; P2=7-2=5; P3=5-4=1; P4=11-5=6$$

带权周转时间 = 周转时间/运行时间

$$P1=16/7=2.28; P2=5/4=1.25; P3=1/1=1; P4=6/4=1.5$$

等待时间 = 周转时间 - 运行时间

$$P1=16-7=9; P2=5-4=1; P3=1-1=0; P4=6-4=2$$

平均周转时间 = $(16+5+1+6)/4 = 7$

平均带权周转时间 = $(2.28+1.25+1+1.5)/4 = 1.50$

平均等待时间 = $(9+1+0+2)/4 = 3$

8
2.56
4

3.3.2 轮转调度算法

1. 轮转法的基本原理

在轮转(RR)法中，系统将所有的就绪进程按FCFS策略排成一个就绪队列。

系统可设置每隔一定时间(如30 ms)便产生一次中断，去激活进程调度程序进行调度，把CPU分配给队首进程，并令其执行一个时间片。当它运行完毕后，又把处理机分配给就绪队列中新的队首进程，也让它执行一个时间片。这样，就可以保证就绪队列中的所有进程在确定的时间段内，都能获得一个时间片的处理机时间。

2. 进程切换时机

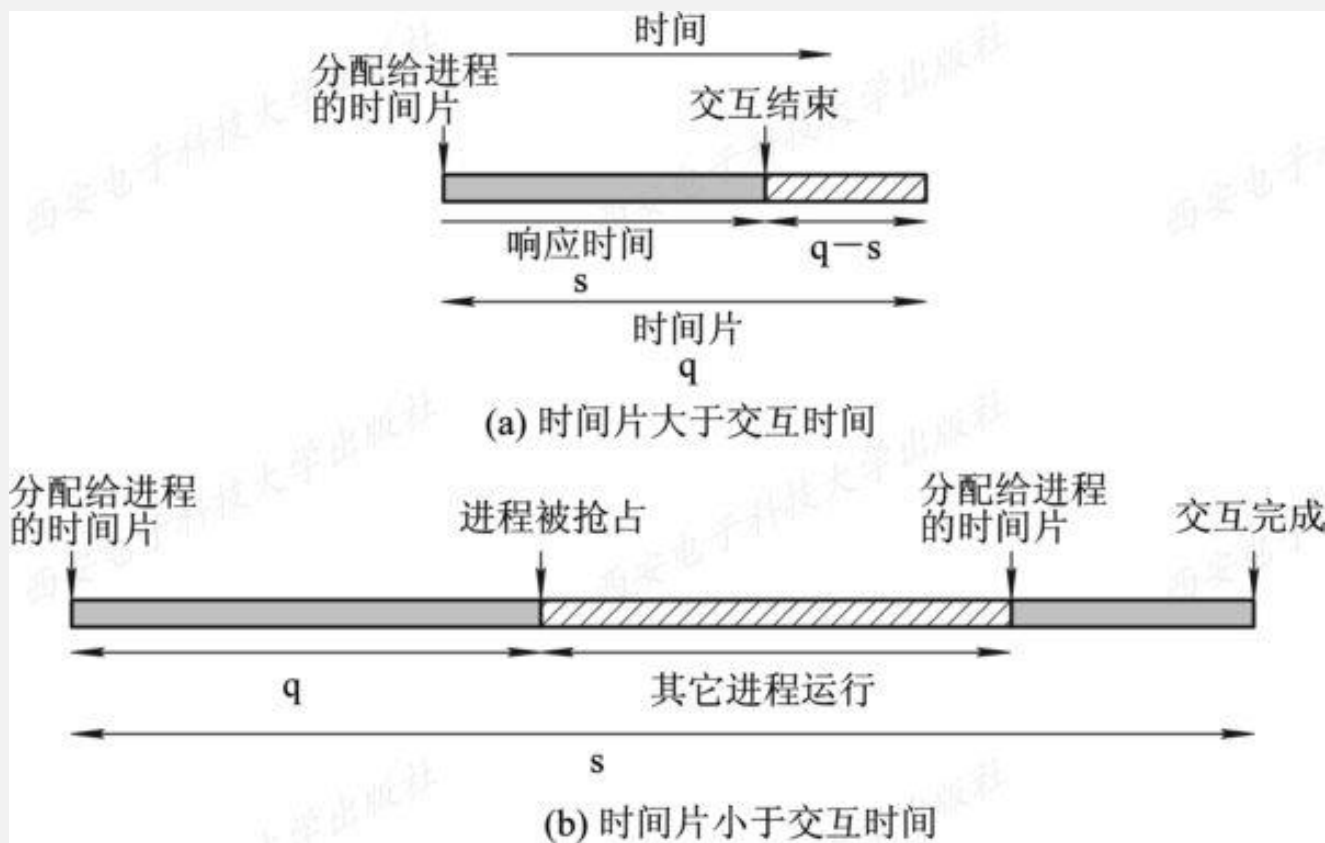
在RR调度算法中，应在何时进行**进程的切换**，可分为**两种情况**：

① 若一个时间片尚未用完，正在运行的进程便已经完成，就立即激活调度程序，将它从就绪队列中删除，再调度就绪队列中队首的进程运行，并启动一个新的时间片。② 在一个时间片用完时，计时器中断处理程序被激活。如果进程尚未运行完毕，调度程序将把它送往就绪队列的末尾。

3. 时间片大小的确定

轮转算法中，时间片大小对系统性能有很大影响。

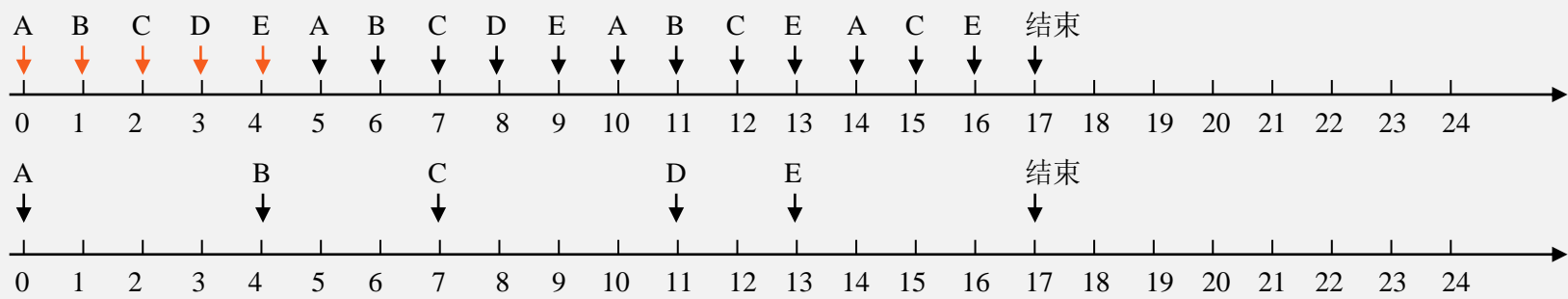
图3-2示出了时间片大小对响应时间的影响。



3. 时间片大小的确定

图3-3 q = 1和q = 4时进程的周转时间

作业 情况 时 间 片	进程名	A	B	C	D	E	平均
	到达时间	0	1	2	3	4	
	服务时间	4	3	4	2	4	
RR q=1	完成时间	15	12	16	9	17	
	周转时间	15	11	14	6	13	11.8
	带权周转时间	3.75	3.67	3.5	3	3.33	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



3.3.3 优先级调度算法

1. 优先级调度算法的类型

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

- (1) 非抢占式优先级调度算法。
- (2) 抢占式优先级调度算法。

2. 优先级的类型

1) 静态优先级

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

- (1) 进程类型。
- (2) 进程对资源的需求。
- (3) 用户要求。

2. 优先级的类型

2) 动态优先级

动态优先级是指在创建进程之初，先赋予其一个优先级，然后其值随进程的推进或等待时间的增加而改变，以便获得更好的调度性能。

例子：P102

将系统中的进程就绪队列从一个拆分为若干个，将不同类型或性质的进程固定分配在不同的就绪队列，不同的就绪队列采用不同的调度算法。一个就绪队列的进程可以设置不同的优先级，不同的就绪队列本身也可以设置不同的优先级。

多队列优先调度算法由于设置多个就绪队列，因此对每个就绪队列可实施不同的调度算法。

多处理机系统，方便为每个处理机设置一个单独的就绪队列。

3.3.5 多级反馈队列调度算法

1. 调度机制

多级反馈队列调度算法的调度机制可描述如下：

(1) **设置多个就绪队列**。多个就绪队列优先级不同，每个就绪队列时间片大小不同。

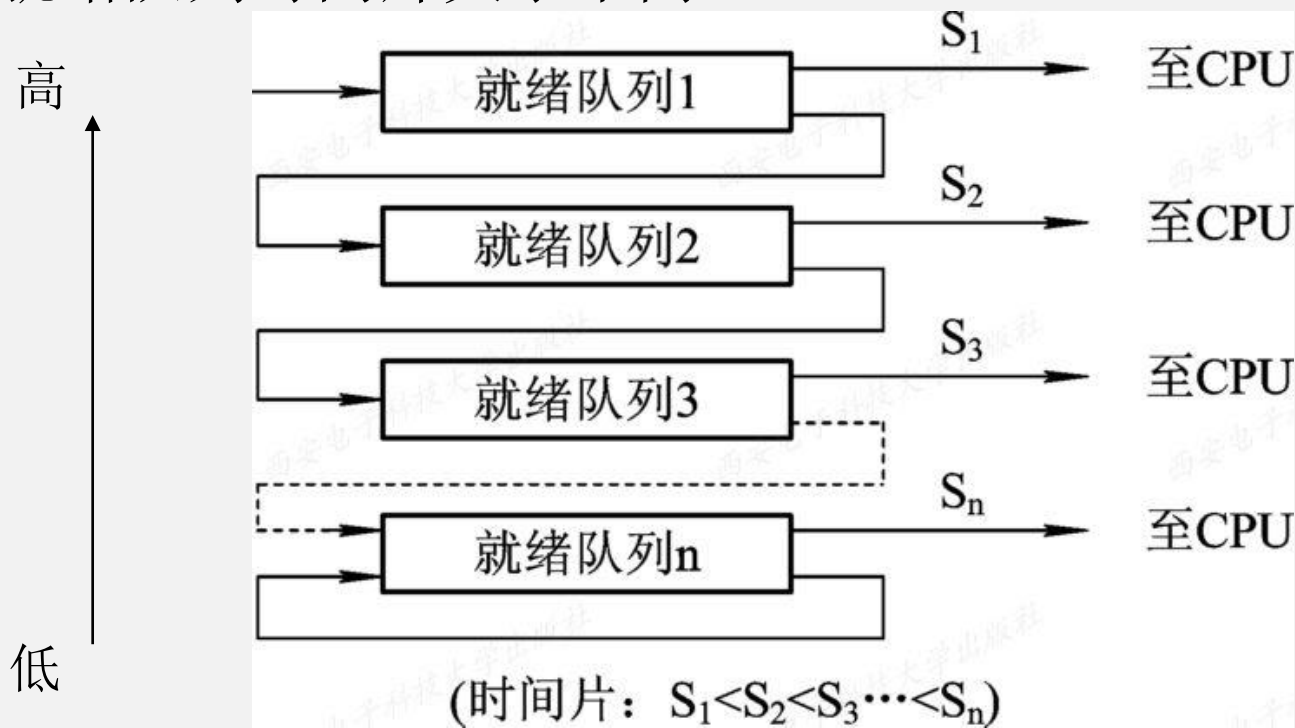


图3-4 多级反馈队列调度算法

3.3.5 多级反馈队列调度算法

1. 调度机制

(2) 每个队列都采用FCFS算法。

当新进程进入内存后，首先将它放入第一队列的末尾，按FCFS原则等待调度。当轮到该进程执行时，如它能在该时间片内完成，便可撤离系统。否则，即它在一个时间片结束时尚未完成，调度程序将其转入第二队列的末尾等待调度；如果它在第二队列中运行一个时间片后仍未完成，再依次将它放入第三队列，……，依此类推。当进程最后被降到第 n 队列后，在第 n 队列中便采取按RR方式运行。

1. 调度机制

(3) 按队列优先级调度。

调度程序首先调度最高优先级队列中的诸进程运行，仅当第一队列空闲时才调度第二队列中的进程运行；换言之，仅当第 $1 \sim (i-1)$ 所有队列均空时，才会调度第 i 队列中的进程运行。如果处理机正在第 i 队列中为某进程服务时又有新进程进入任一优先级较高的队列，此时须立即把正在运行的进程放回到第 i 队列的末尾，而把处理机分配给新到的高优先级进程。

2. 调度算法的性能

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

- (1) 终端型用户。
- (2) 短批处理作业用户。
- (3) 长批处理作业用户。

1. 保证调度算法

保证调度算法是另外一种类型的调度算法，它向用户所做出的保证并不是优先运行，而是明确的性能保证，该算法可以做到调度的公平性。一种比较容易实现的性能保证是处理机分配的公平性。**如果在系统中有 n 个相同类型的进程同时运行，为公平起见，须保证每个进程都获得相同的处理机时间 $1/n$ 。**

3.3.6 基于公平原则的调度算法

1. 保证调度算法

在实施公平调度算法时系统中必须具有这样一些功能：

- (1) 跟踪计算每个进程自创建以来已经执行的处理时间。
- (2) 计算每个进程应获得的处理机时间，即自创建以来的时间除以 n 。
- (3) 计算进程获得处理机时间的比率，即进程实际执行的处理时间和应获得的处理机时间之比。
- (4) 比较各进程获得处理机时间的比率。如进程A的比率最低，为0.5，而进程B的比率为0.8，进程C的比率为1.2等。
- (5) 调度程序应选择比率最小的进程将处理机分配给它，并让该进程一直运行，直到超过最接近它的进程比率为止。

2. 公平分享调度算法

分配给每个进程相同的处理机时间，显然，这对诸进程而言，是体现了一定程度的公平，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，使所有用户能获得相同的处理机时间，或所要求的时间比例。然而调度是以进程为进步单位，为此，必须考虑每个用户所拥有的进程数。例子P105，用户1有4个进程，用户2有1个进程E

用户1与用户2 处理机时间相同：A E B E C E D E A E B E C E D E ...

用户1是用户2 处理机时间2倍：A B E C D E A B E C D E A B E C D E...

第三章 处理机调度与死锁

3.4 实时调度

3.4 实时调度

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

3.4.1 实现实时调度的基本条件

1. 提供必要的信息

为了实现实时调度，系统应向调度程序提供有关任务的信息：

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

3.4.1 实现实时调度的基本条件

- (3) **处理时间**，一个任务从开始执行，直至完成时所需的时间。
- (4) **资源要求**，任务执行时所需的一组资源。
- (5) **优先级**，如果某任务的开始截止时间错过，势必引起故障，则应为该任务赋予“绝对”优先级；如果其开始截止时间的错过，对任务的继续运行无重大影响，则可为其赋予“相对”优先级，供调度程序参考。

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，若处理机的处理能力不够强，则有可能因处理机忙不过，而致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HRT，它们的处理时间可表示为 C_i ，周期时间表示为 P_i ，则在单处理机情况下，必须满足下面的限制条件系统才是可调度的：

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

3.4.1 实现实时调度的基本条件

提高系统处理能力的途径有二：一是采用单处理机系统，但须增强其处理能力，以显著地减少对每一个任务的处理时间；二是采用多处理机系统。假定系统中的处理机数为N，则应将上述的限制条件改为：

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

3.4.1 实现实时调度的基本条件

3. 采用抢占式调度机制

在含有HRT任务的实时系统中，广泛采用抢占机制。这样便可满足HRT任务对截止时间的要求。但这种调度机制比较复杂。

3.4.1 实现实时调度的基本条件

4. 具有快速切换机制

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

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

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

可以按不同方式对实时调度算法加以分类：① 根据实时任务性质，可将实时调度的算法分为硬实时调度算法和软实时调度算法；② 按调度方式，则可分为非抢占调度算法和抢占调度算法。

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。

为若干个相同或类似的对象，分别建立一个实时任务，排成轮转队列。每次选择该队列第一个任务运行，任务完成后，挂在队列末尾等待。调度程序调度下一个队首任务。

(2) 非抢占式**优先**调度算法。

为特殊任务赋予较高优先级，将它们排在就绪队列队首，等待当前任务自我终止或运行完成后，便可去调度执行队首的高优先级进程。

3.4.2 实时调度算法的分类

2. 抢占式调度算法

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**的抢占式优先级调度算法。
- (2) **立即抢占**(Immediate Preemption)的优先级调度算法。

3.4.2 实时调度算法的分类

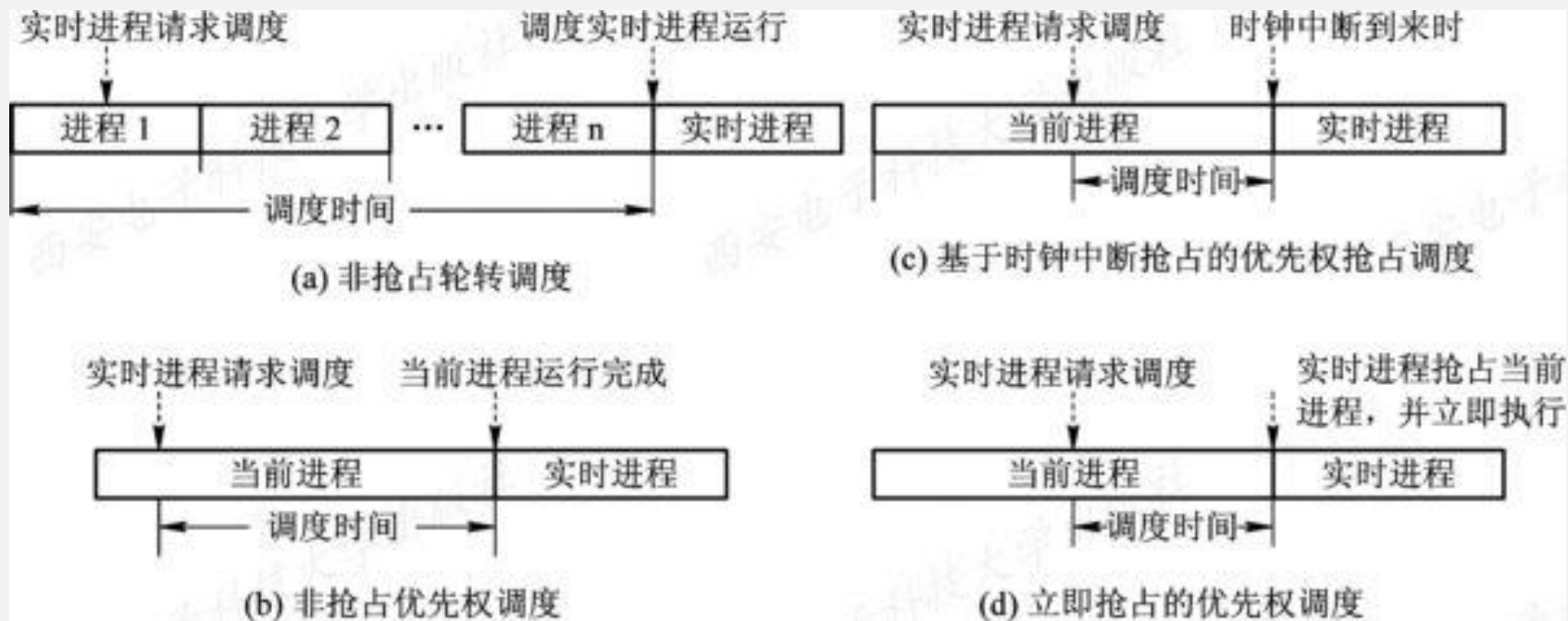


图3-5 实时进程调度

3.4.3 最早截止时间优先EDF算法

该算法是根据任务的**截止时间**确定任务的优先级，任务的截止时间愈早，其优先级愈高，具有最早截止时间的任务拍照队列的队首。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。最早截止时间优先算法既可以用于抢占式调度方式中，也可用于非抢占式调度方式中。

3.4.3 最早截止时间优先EDF算法

1. 非抢占式调度方式用于非周期实时任务

图3-6示出了将该算法用于非抢占调度方式之例。

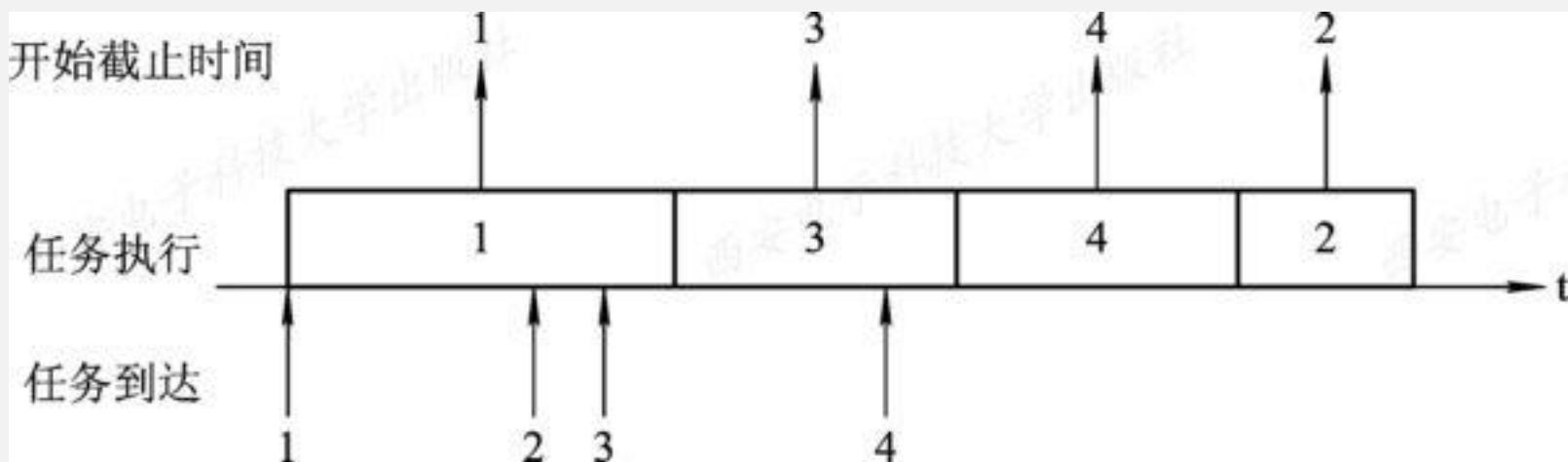


图3-6 EDF算法用于非抢占调度方式

3.4.3 最早截止时间优先EDF算法

2. 抢占式调度方式用于周期实时任务

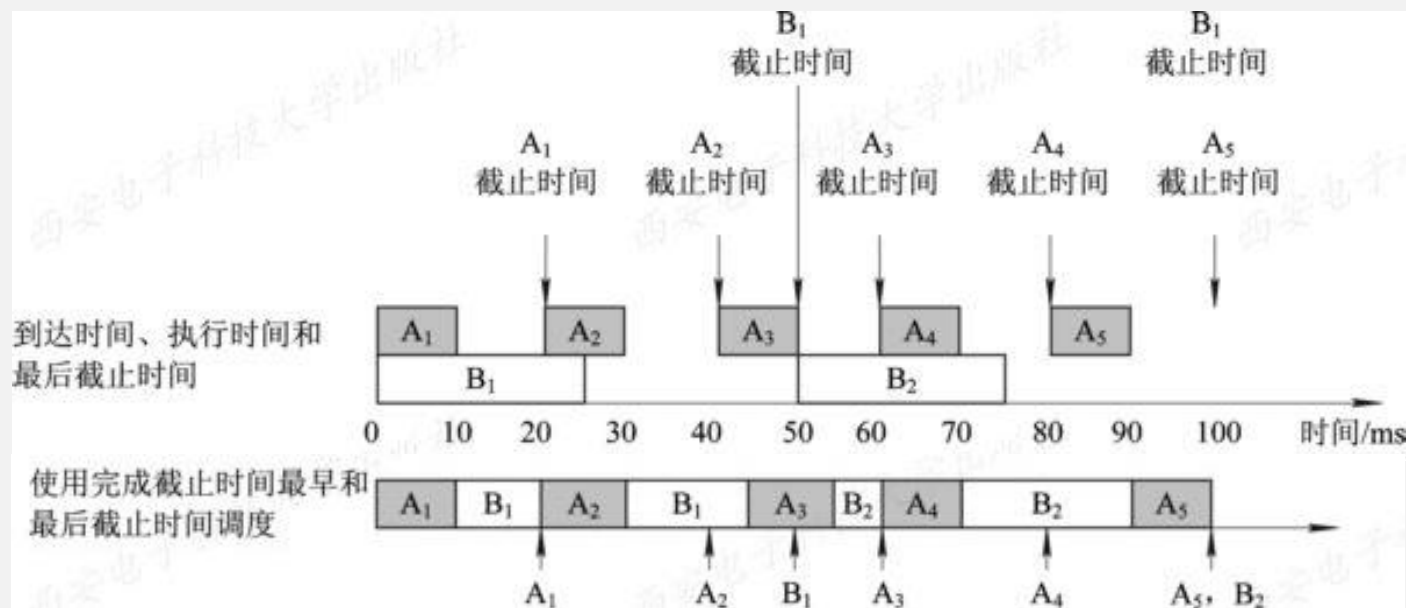
在该例中有两个周期任务，任务A和任务B的周期时间分别为20 ms和50 ms，每个周期的处理时间分别为10 ms和25 ms。



3.4.3 最早截止时间优先EDF算法

2. 抢占式调度方式用于周期实时任务

在该例中有两个周期任务，任务A和任务B的周期时间分别为20 ms和50 ms，每个周期的处理时间分别为10 ms和25 ms。



3.4.4 最低松弛度优先LLF算法

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

该算法主要用于可抢占调度方式中。

松弛度=必须完成时间-任务本身还需运行时间-当前时间

3.4.4 最低松弛度优先LLF算法

假如在一个实时系统中有两个周期性实时任务A和B，任务A要求每20 ms执行一次，执行时间为10 ms，任务B要求每50 ms执行一次，执行时间为25 ms。由此可知，任务A和B每次必须完成的时间分别为： A_1 、 A_2 、 A_3 、...和 B_1 、 B_2 、 B_3 、...，见图3-8。

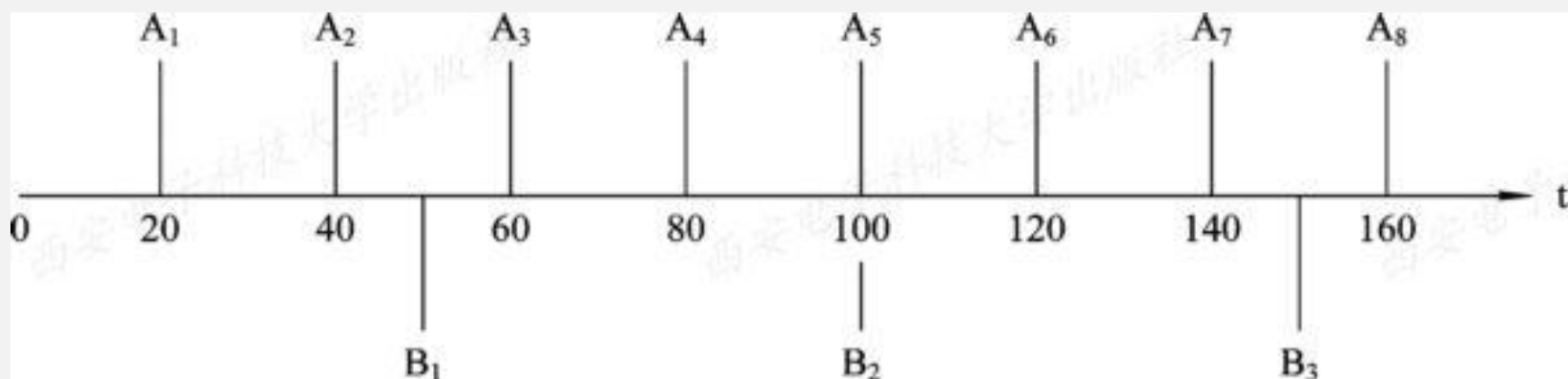


图3-8 A和B任务每次必须完成的时间

3.4.4 最低松弛度优先LLF算法

图3-9示出了具有两个周期性实时任务的调度情况。

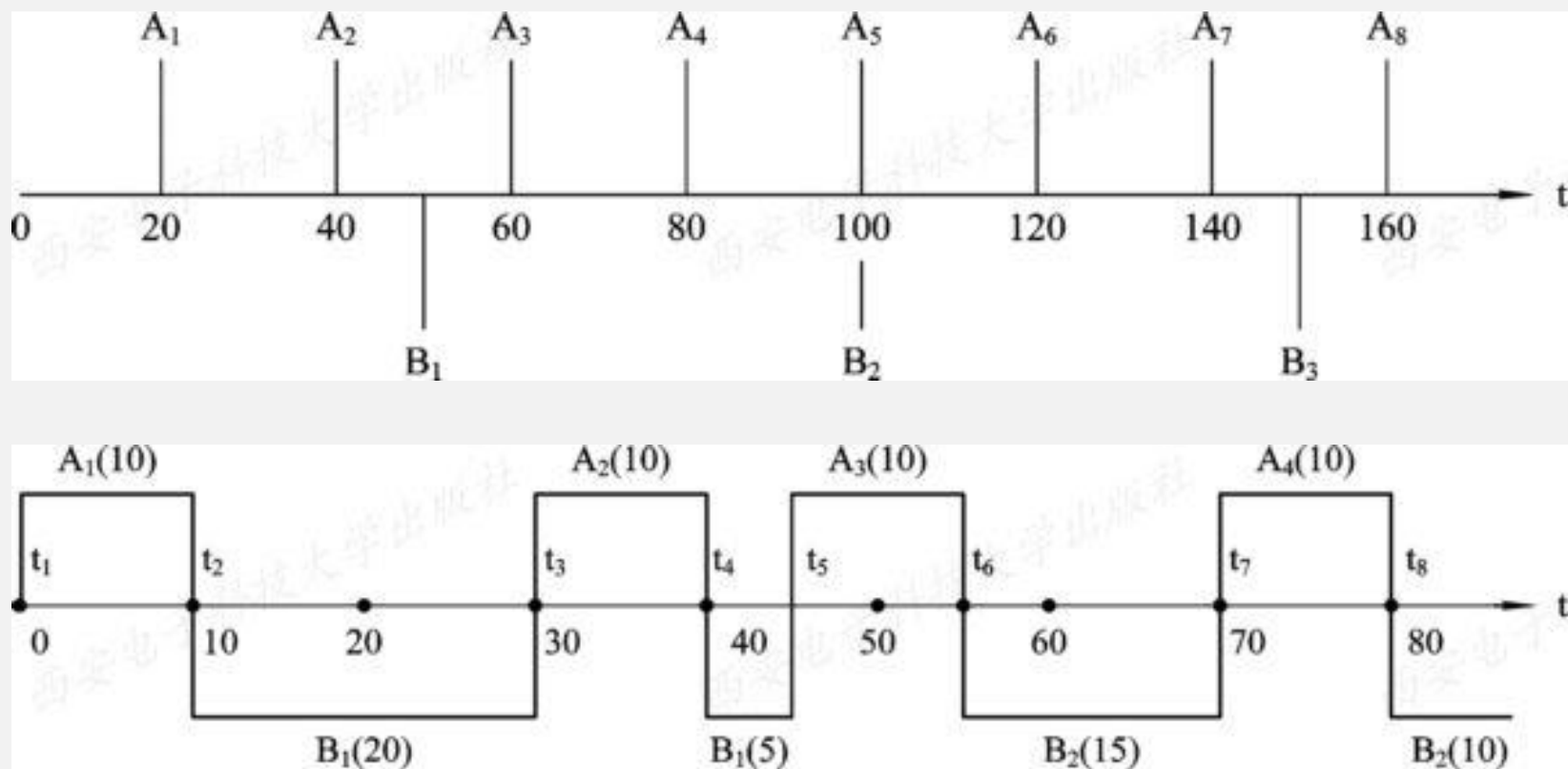


图3-9 利用ELLF算法进行调度的情况

3.4.4 最低松弛度优先LLF算法

图3-9示出了具有两个周期性实时任务的调度情况。

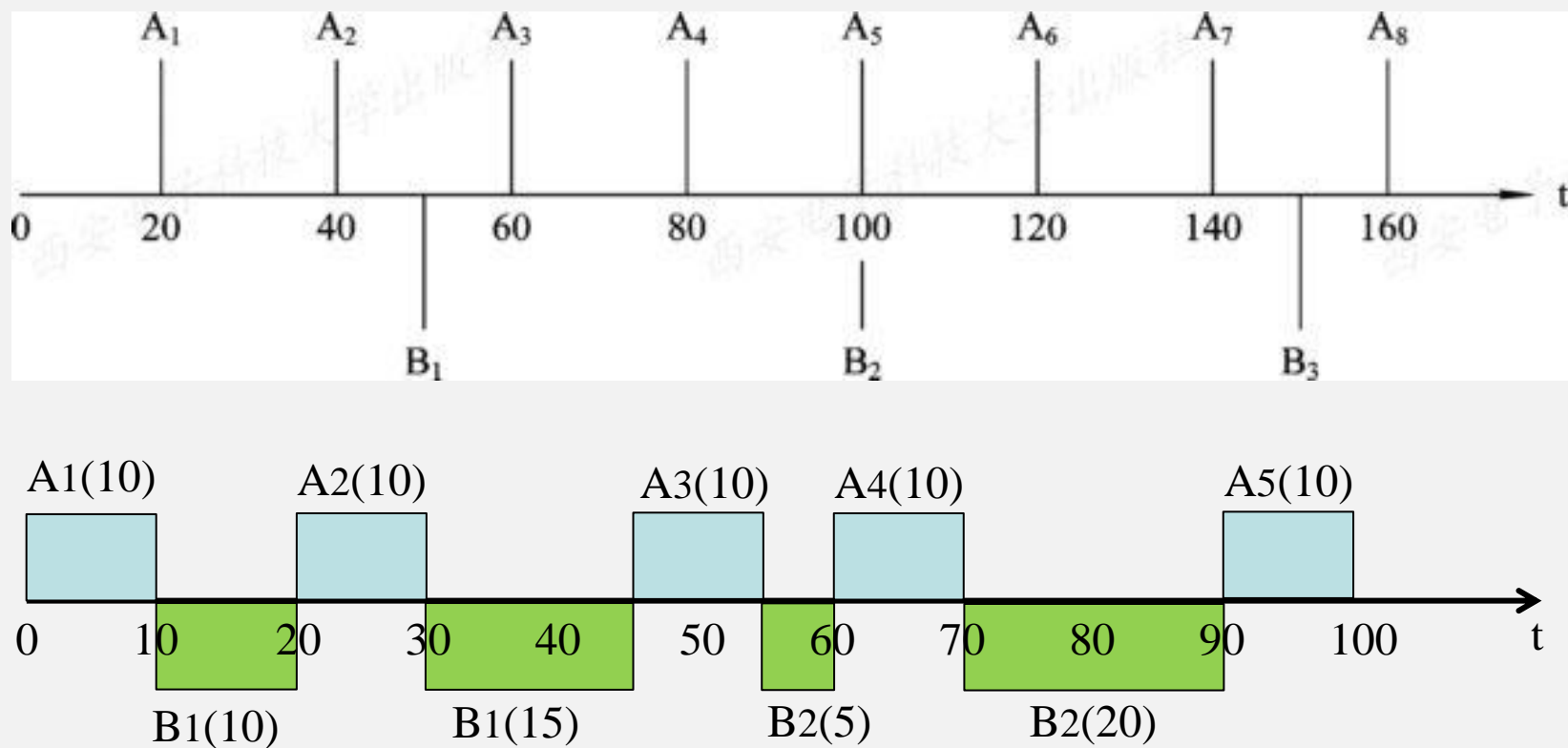


图3-9 利用ELLF算法进行调度的情况

1. 优先级倒置的形成

当前OS广泛采用优先级调度算法和抢占方式，然而在系统中存在着影响进程运行的资源而可能产生“优先级倒置”的现象，即高优先级进程(或线程)被低优先级进程(或线程)延迟或阻塞。我们通过一个例子来说明该问题。

假如有三个完全独立的进程P1、P2和P3，P1优先级最高，P2次之，P3最低。P1和P3共享一个临界资源进行交互。

P1: ...P(mutex); CS-1; V(mutex);...

P2: ...program2...

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

优先级： $P_1 > P_2 > P_3$ ， P_3 最先执行，执行P(mutex)操作后，进入到临界区CS-3。在时刻a， P_2 就绪，因为它比 P_3 的优先级高， P_2 抢占了 P_3 的处理机而运行，如图3-10所示。

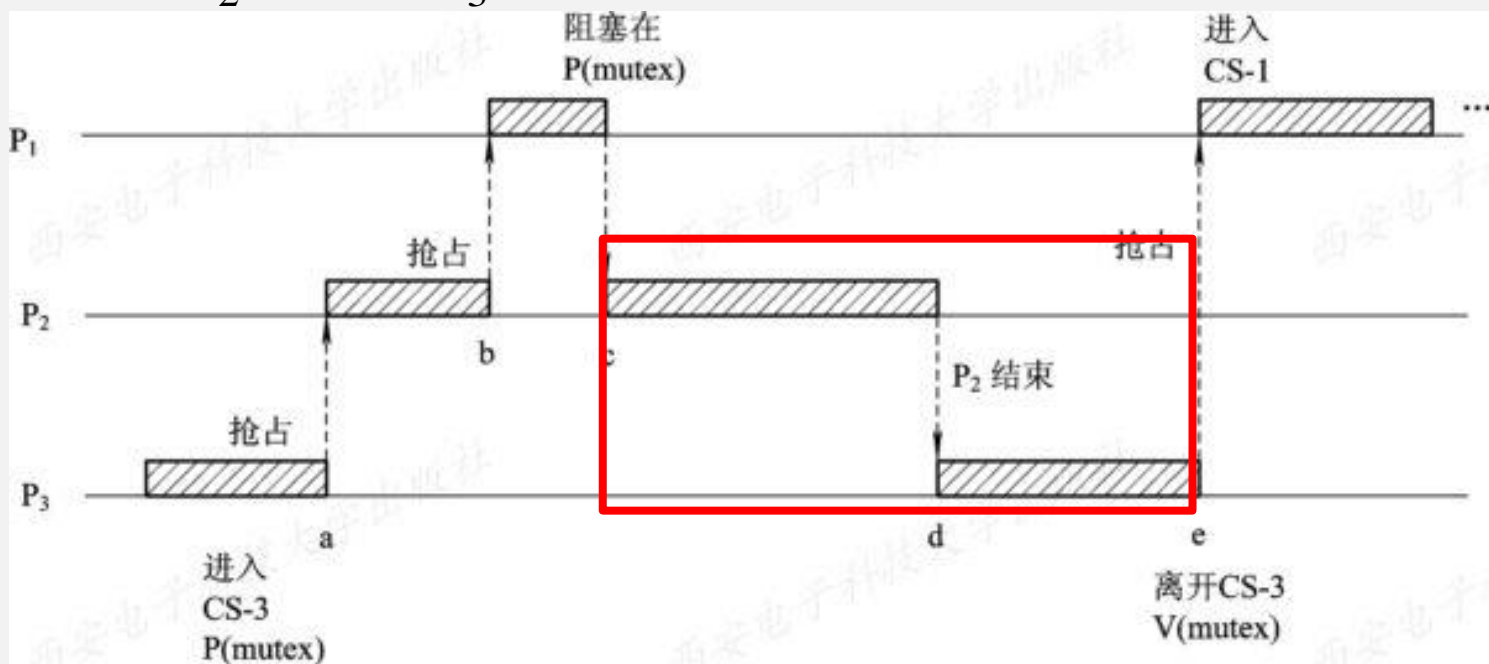


图3-10 优先级倒置示意图

3.4.5 优先级倒置

2. 优先级倒置的解决方法

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

2) 图3-11示出了采用动态优先级继承方法后， P_1 、 P_2 、 P_3 三个进程的运行情况。

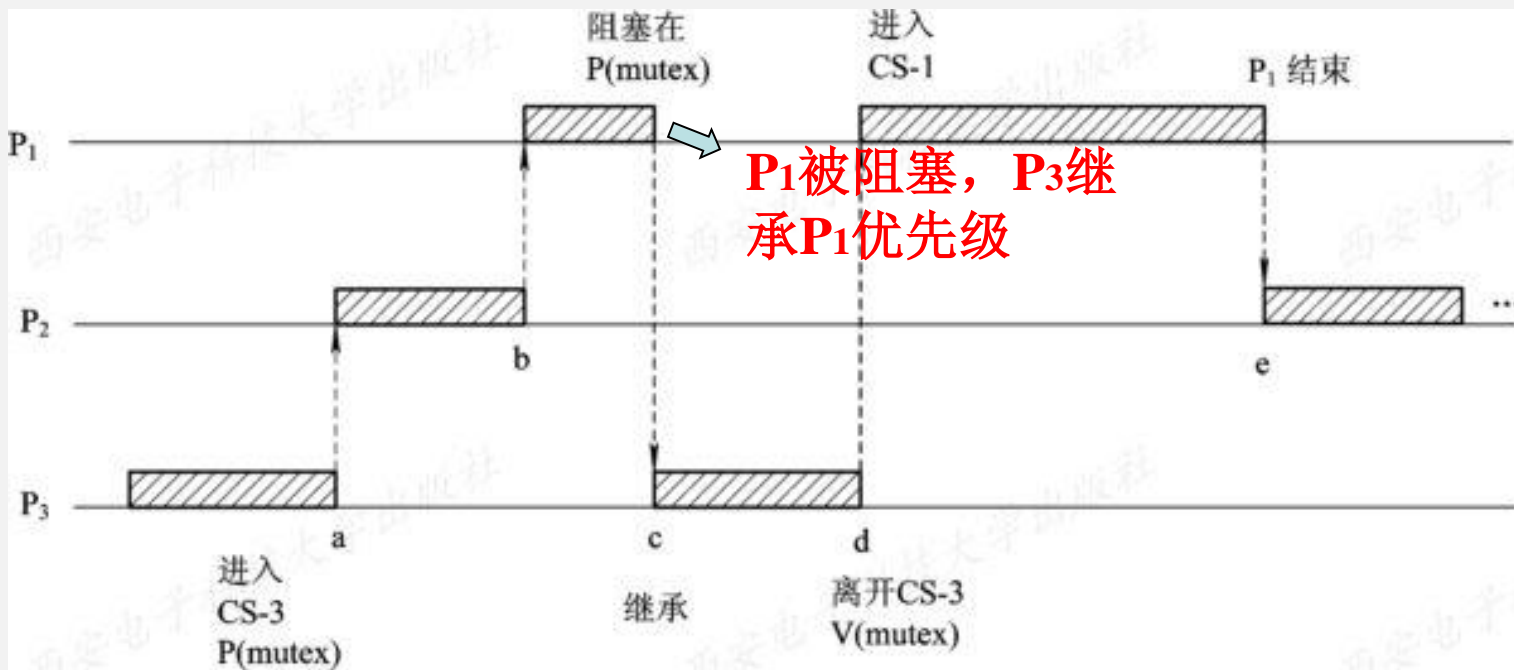


图3-11 采用了动态优先级继承方法的运行情况

目录

CONTENTS

1

处理机调度的层次和调度算法的目标

2

作业与作业调度

3

进程调度

4

实时调度

5

死锁概述

6

预防死锁

7

避免死锁

8

死锁的检测与解除

第三章 处理机调度与死锁

3.5 死锁概述

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

1. 可重用性资源和消耗性资源

1) 可重用性资源

可重用性资源是一种可供用户重复使用多次的资源，它具有如下**性质**：

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

(2) 进程在使用可重用性资源时，须按照这样的顺序：

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

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

2) 可消耗性资源

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

- ① 每一类可消耗性资源的单元数目在进程运行期间是可以不断变化的，有时它可以有许多，有时可能为0；
- ② 进程在运行过程中，可以不断地创造可消耗性资源的单元，将它们放入该资源类的缓冲区中，以增加该资源类的单元数目。
- ③ 进程在运行过程中，可以请求若干个可消耗性资源单元，用于进程自己的消耗，不再将它们返回给该资源类中。

可消耗性资源通常是由生产者进程创建，由消费者进程消耗。

2. 可抢占性资源和不可抢占性资源

1) 可抢占性资源

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

2) 不可抢占性资源

另一类资源是不可抢占性资源，即一旦系统把某资源分配给该进程后，就不能将它强行收回，只能在进程用完后自行释放。例如，某进程刻录光盘时，不可分配其他进程。

3.5.2 计算机系统中的死锁

1. 竞争不可抢占性资源引起死锁

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

3.5.2 计算机系统中的死锁

系统中有两个进程P1和P2，都准备写两个文件F1和F2，在P1打开F1同时P2打开F2，利用**资源分配图**进行描述，用方块代表可重用的资源(文件)，用圆圈代表进程：

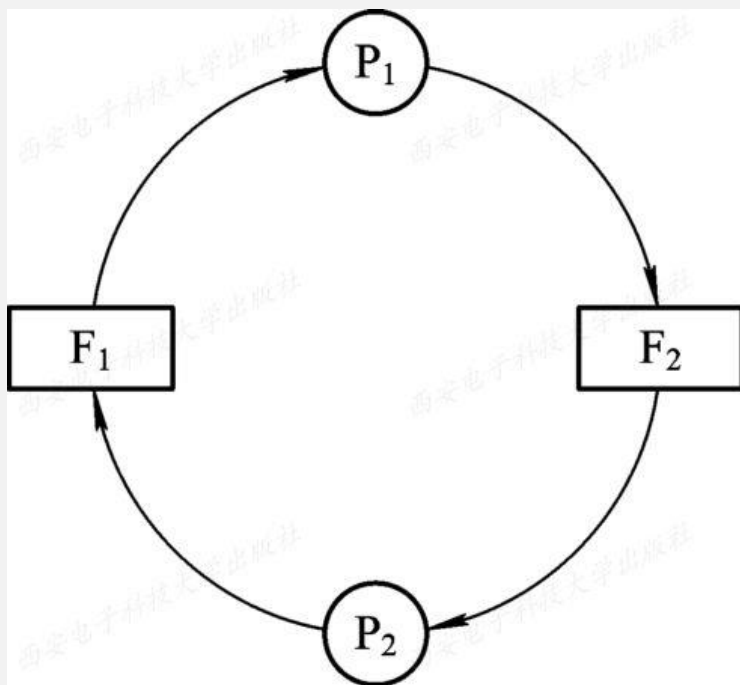


图3-12 共享文件时的死锁情况

2. 竞争可消耗资源引起死锁

现在进一步介绍竞争可消耗资源所引起的死锁。图3-13示出了在三个进程之间，在利用消息通信机制进行通信时所形成的死锁情况。

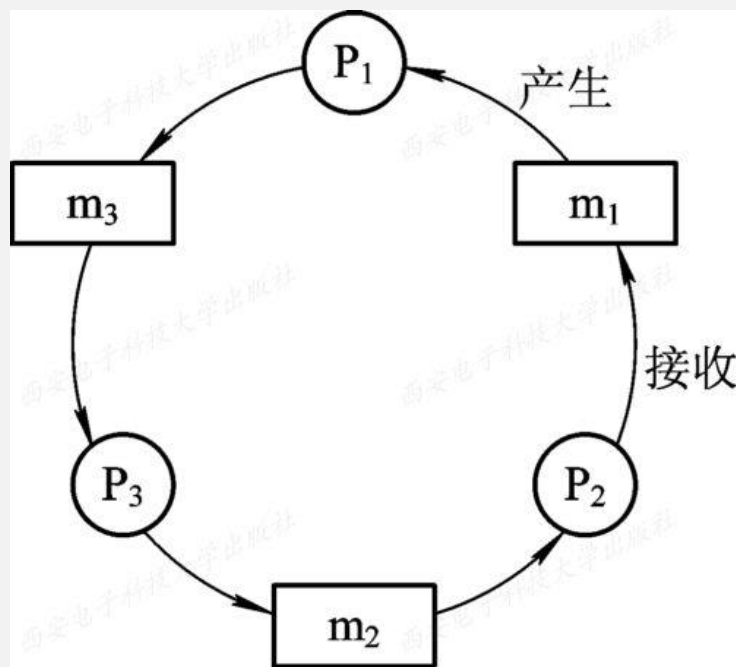


图3-13 进程之间通信时的死锁

3. 进程推进顺序不当引起死锁

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

1) 进程推进顺序合法

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

3.5.2 计算机系统中的死锁

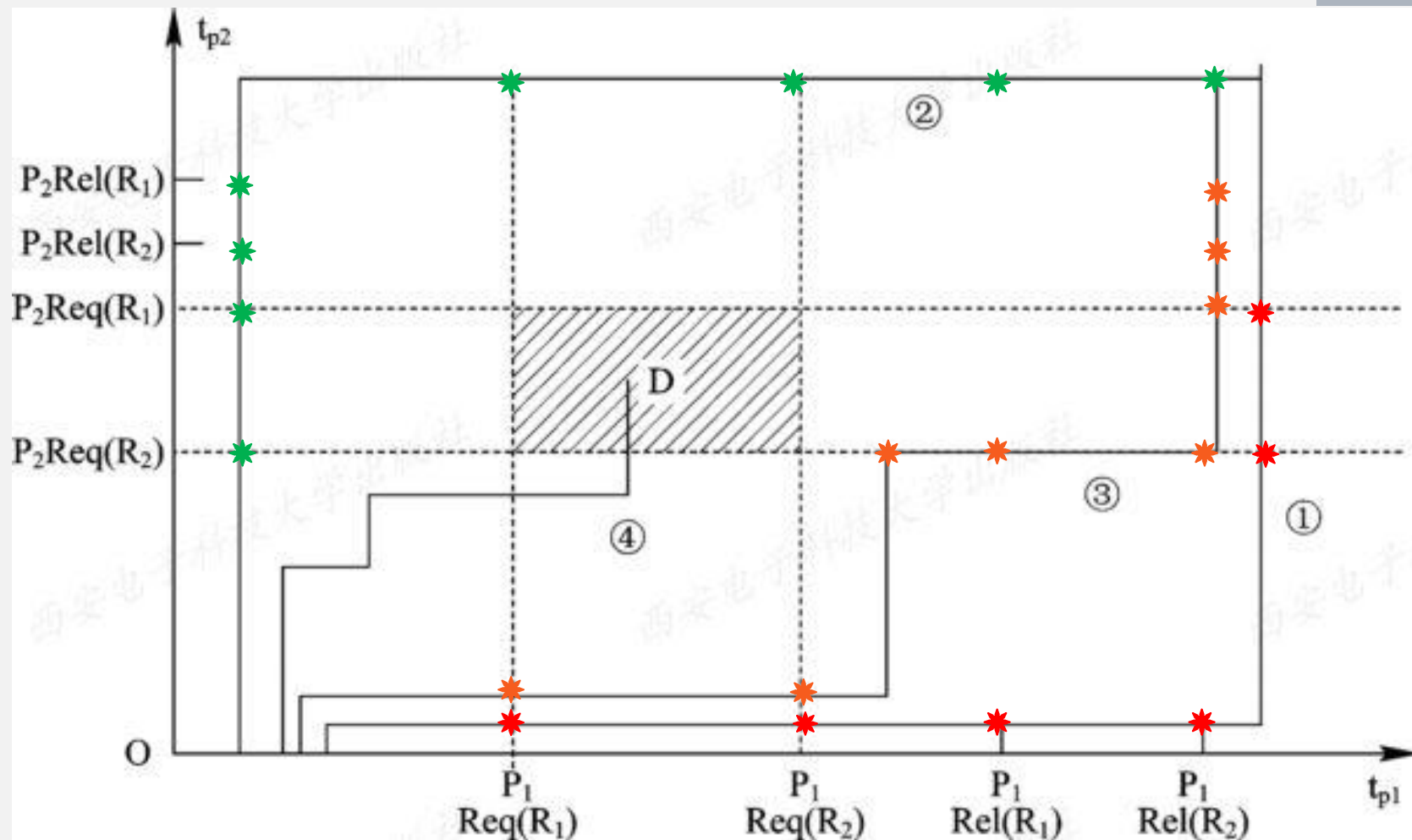


图3-14 进程推进顺序对死锁的影响

2) 进程推进顺序非法

若并发进程 P_1 和 P_2 按图3-14中曲线④所示的顺序推进，它们将进入**不安全区D**内。此时 P_1 保持了资源 R_1 ， P_2 保持了资源 R_2 ，系统处于不安全状态。此刻，**如果两个进程继续向前推进，就可能发生死锁。**

例如，当 P_1 运行到 $P_1: \text{Request}(R_2)$ 时，将因 R_2 已被 P_2 占用而阻塞；当 P_2 运行到 $P_2: \text{Request}(R_1)$ 时，也将因 R_1 已被 P_1 占用而阻塞，于是发生了进程死锁，这样的进程推进顺序就是非法的。

3.5.2 计算机系统中的死锁

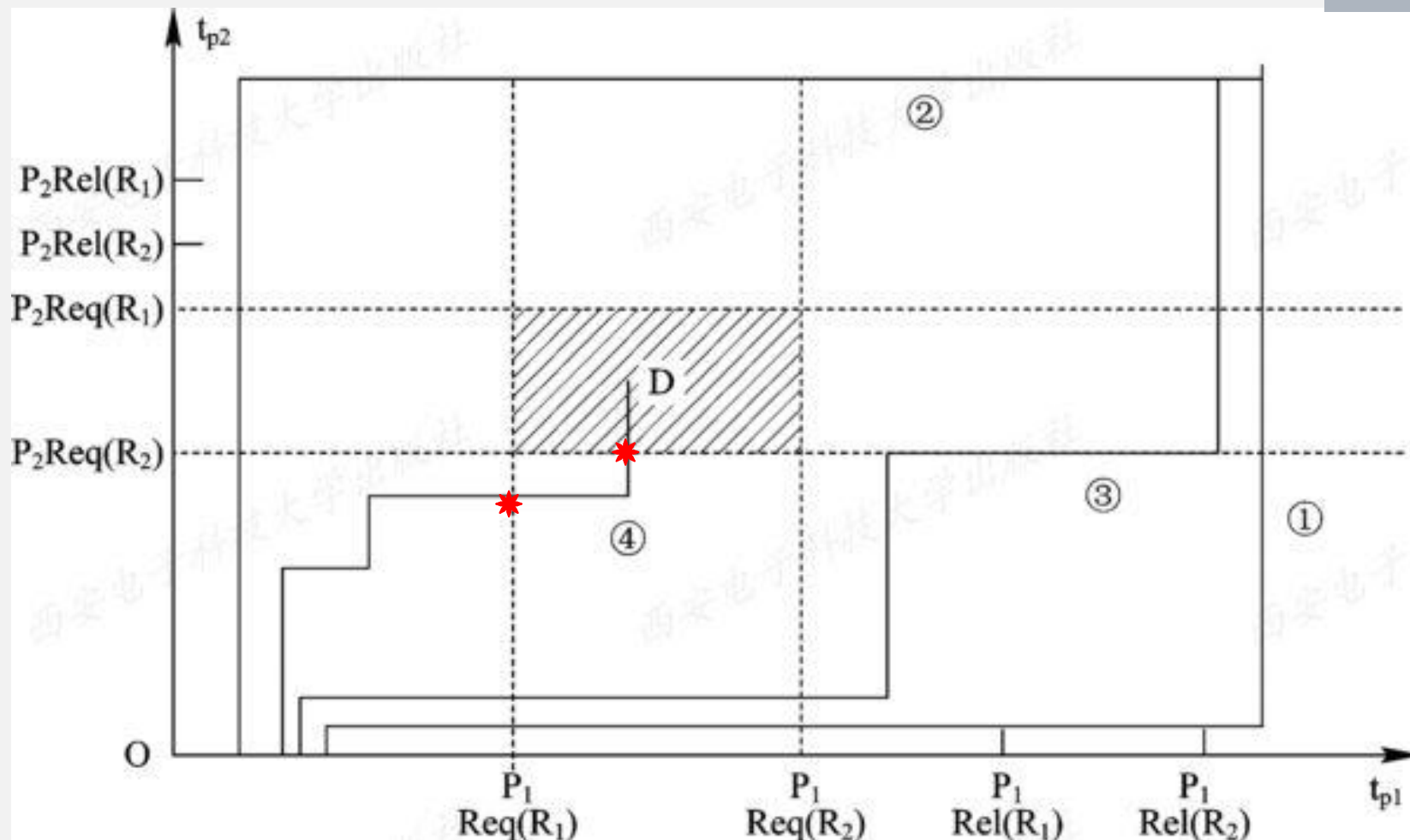


图3-14 进程推进顺序对死锁的影响

1. 死锁的定义

在一组进程发生死锁的情况下，这组死锁进程中每个进程，都在等待另一个进程所占有的资源。或者说每个进程所等待的事件是该组中其他进程释放所占有的资源。但由于所有进程无法继续运行，都不能释放资源，致使该组没有任何进程被唤醒，只能无限期等待。

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

2. 产生死锁的必要条件

虽然进程在运行过程中可能会发生死锁，但产生进程死锁是必须具备一定条件的。综上所述不难看出，产生死锁必须同时具备下面四个必要条件，只要其中任一个条件不成立，死锁就不会发生：

- (1) 互斥条件。
- (2) 请求和保持条件。
- (3) 不可抢占条件。
- (4) 循环等待条件。

2. 产生死锁的必要条件

(1) 互斥条件。

进程对所分配到的资源进行排他性的使用，即在一段时间内，某资源只能被一个进程占用。如果此时还有其他进程请求该资源，则请求进程只能等待，直至占有该资源的进程用毕释放。

3.5.3 死锁的定义、必要条件和处理方法

2. 产生死锁的必要条件

(2) 请求和保持条件。

进程已经保持了至少一个资源，但又提出了新的资源请求，而该资源已被其它进程占有，此时请求进程被阻塞，但对自己已获得的资源保持不变。

2. 产生死锁的必要条件

(3) 不可抢占条件。

进程已获得的资源在未使用完之前不能被抢占，只能在进程使用完时由自己释放。

2. 产生死锁的必要条件

(4) 循环等待条件。

在发生死锁时，必然存在一个进程——资源的循环链，即进程集合 $\{P_0, P_1, P_2, \dots, P_n\}$ 中的 P_0 正在等待一个 P_1 占用的资源， P_1 正在等待 P_2 占用的资源，……， P_n 正在等待一个 P_0 占用的资源。

3.5.3 死锁的定义、必要条件和处理方法

3. 处理死锁的方法

目前处理死锁的方法可归结为四种：

- (1) 预防死锁。
- (2) 避免死锁。
- (3) 检测死锁。
- (4) 解除死锁。

3. 处理死锁的方法

(1) 预防死锁。

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，去破坏产生死锁四个必要条件中的一个或几个来预防产生死锁。预防死锁是一种较易实现的方法，已被广泛使用。

3. 处理死锁的方法

(2) 避免死锁。

同样属于事先预防策略，但并不是事先采取各种限制措施，去破坏产生死锁的四个必要条件，而是在资源动态分配的过程中，用某种方法防止系统进入不安全状态，从而可以避免死锁。**银行家算法。**

3. 处理死锁的方法

(3) 检测死锁。

这种方法无需事先采取任何限制性措施，允许进程在运行过程中发生死锁。但可通过检测机构及时地检测出死锁的发生，然后采取适当的措施，把进程从死锁中解脱出来。

3. 处理死锁的方法

(4) 解除死锁。

当检测到系统已发生死锁时，就采取相应措施，将进程从死锁状态中解脱出来。常用的方法是撤消一些进程，回收它们的资源，将它们分配给已处于阻塞状态的进程，使其能继续运行。

第三章 处理机调度与死锁

3.6 预防死锁

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

3.6.1 破坏“请求和保持”条件

为了能破坏“**请求和保持**”条件，系统必须保证做到：
当一个进程在请求资源时，它不能持有不可抢占资源。该保证可通过如下两个不同的协议实现：

1. 第一种协议

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

优点：简单、易行且安全。

缺点：(1) 资源被严重浪费，严重地恶化了资源利用率。
(2) 会使得进程经常发生饥饿现象。

3.6.1 破坏“请求和保持”条件

2. 第二种协议

该协议是对第一种协议的改进，它允许一个进程只获得运行初期所需的资源后，便开始运行。进程运行过程中再逐步释放已分配给自己的、且已用毕的全部资源，然后再请求新的资源。

例：进程需要完成任务为，先将数据从磁带上复制到磁盘文件上，然后对磁盘文件进行排序，最后将其打印出来。

3.6.2 破坏“不可抢占”条件

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

该方法实现起来比较复杂，且需付出很大的代价。

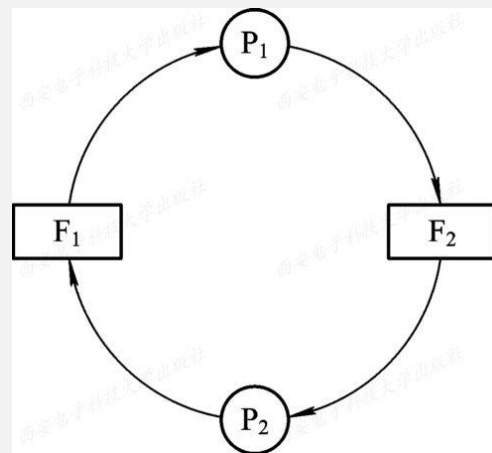
3.6.3 破坏“循环等待”条件

一个能保证“循环等待”条件不成立的方法是，对系统所有资源类型进行线性排序，并赋予不同的序号。在对系统所有资源类型排序后，可采用预防协议：规定每个进程必须按序号递增顺序请求资源。

设 $R = (R_1, R_2, R_3, \dots, R_m)$ 为资源类型的集合，为每个资源类型赋予唯一的序号。如果系统中有磁带驱动器、硬盘驱动器、打印机，则函数 F 可定义：

$$F(\text{tape drive}) = 1; \quad F(\text{disk drive}) = 5;$$

$$F(\text{printer}) = 12;$$



第三章 处理机调度与死锁

3.7 避免死锁

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

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

1. 安全状态

在该方法中，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。**安全状态**是指系统能按某种进程推进顺序(P_1, P_2, \dots, P_n)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完成。此时，称(P_1, P_2, \dots, P_n)为安全序列。

2. 安全状态之例

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

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

3. 由安全状态向不安全状态的转换

如果不按照安全序列分配资源，则系统可能会由安全状态进入不安全状态。假如在 T_0 时刻后， P_3 又请求1台磁带机，若此时系统把剩余3台中的1台分配给它，则系统进入**不安全状态**。

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

3.7.2 利用银行家算法避免死锁

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

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

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

(1) 可利用资源向量**Available**。

(2) 最大需求矩阵**Max**。

(3) 分配矩阵**Allocation**。

(4) 需求矩阵**Need**。

3.7.2 利用银行家算法避免死锁

1. 银行家算法中的数据结构

(1) 可利用资源向量**Available**。

含有 m 个元素的数组，其中每个元素代表一类可利用资源的数目，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。

$Available[j]=K$ ，表示系统中现有 R_j 类资源数目为 K 。

(2) 最大需求矩阵**Max**。

(3) 分配矩阵**Allocation**。

(4) 需求矩阵**Need**。

3.7.2 利用银行家算法避免死锁

1. 银行家算法中的数据结构

(1) 可利用资源向量**Available**。

(2) 最大需求矩阵**Max**。

是一个 $n*m$ 的矩阵，定义系统中 n 个进程中的每个进程对 m 类资源的最大需求量。 $Max[i, j]=K$ ，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**。

(4) 需求矩阵**Need**。

3.7.2 利用银行家算法避免死锁

1. 银行家算法中的数据结构

(1) 可利用资源向量**Available**。

(2) 最大需求矩阵**Max**。

(3) 分配矩阵**Allocation**。

是一个 $n*m$ 的矩阵，定义系统中每一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 类资源数目为 K 。。

(4) 需求矩阵**Need**。

3.7.2 利用银行家算法避免死锁

1. 银行家算法中的数据结构

(1) 可利用资源向量**Available**。

(2) 最大需求矩阵**Max**。

(3) 分配矩阵**Allocation**。

(4) 需求矩阵**Need**。

是一个 $n*m$ 的矩阵，用以表示每一进程尚需的各类资源数。 $Need[i, j]=K$ ，表示进程 i 还需要 R_j 类资源数目 K 个才能完成其任务。

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

2. 银行家算法

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

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

(2) 如果 $Request\ i[j] \leq Available[j]$ ，便转向**步骤(3)**；否则，表示尚无足够资源， P_i 须等待。

(3) 系统试探着把资源分配给进程 P_i ，并修改下面数据结构中的数值：

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

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

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

(4) 系统执行**安全性算法**，检查此次资源分配后系统是否处于安全状态。若安全，才正式将资源分配给进程 P_i ，以完成本次分配；否则，将本次的试探分配作废，恢复原来的资源分配状态，让进程 P_i 等待。

3.7.2 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 **Finish**：它表示系统是否有足够的资源分配给进程，使之运行完成。**开始时先做Finish[i] := false**；当有足够资源分配给进程时，再令Finish[i] := true。

(2) 从进程集合中找到一个能满足下述条件的进程：

① $\text{Finish}[i] = \text{false}$;

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

若找到，执行步骤(3)，否则，执行步骤(4)。

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

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

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

go to step 2;

(4) 如果所有进程的 $\text{Finish}[i] = \text{true}$ 都满足，则表示系统处于安全状态；否则，系统处于不安全状态。

4. 银行家算法之例

假定系统中有五个进程{P₀, P₁, P₂, P₃, P₄}和三类资源{A, B, C}，各种资源的数量分别为10、5、7，在T₀时刻的资源分配情况如图3-15所示。

资源 情况 进 程	Max			Allocation			Need			Available		
	A	B	C	A	B	C	A	B	C	A	B	C
P ₀	7	5	3	0	1	0	7	4	3	3 (2	3 3	2 (0)
P ₁	3	2	2	2	0	0	1	2	2			
				(3	0	2)	(0	2	0)			
P ₂	9	0	2	3	0	2	6	0	0			
P ₃	2	2	2	2	1	1	0	1	1			
P ₄	4	3	3	0	0	2	4	3	1			

3.7.2 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(如图3-16所示)可知，在 T_0 时刻存在着一个安全序列 $\{P_1, P_3, P_4, P_2, P_0\}$ ，故系统是安全的。

资源 情况 进 程	Max			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	0	0	1	2	2			
				(3	0	2)	(0	2	0)			
P_2	9	0	2	3	0	2	6	0	0			
P_3	2	2	2	2	1	1	0	1	1			
P_4	4	3	3	0	0	2	4	3	1			

3.7.2 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(如图3-16所示)可知，在 T_0 时刻存在着一个安全序列 $\{P_1, P_3, P_4, P_2, P_0\}$ ，故系统是安全的。

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

图3-16 T_0 时刻的安全序列

(2) P_1 请求资源: P_1 发出请求向量 $\text{Request}_1(1, 0, 2)$, 系统按银行家算法进行检查:

① $\text{Request}_1(1, 0, 2) \leq \text{Need}_1(1, 2, 2)$;

② $\text{Request}_1(1, 0, 2) \leq \text{Available}_1(3, 3, 2)$;

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

④ 再利用安全性算法检查此时系统是否安全, 如图3-17所示。

3.7.2 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 3	2 0)
P ₁	3	2	2	2	0	0	1	2	2			
				(3	0	2)	(0	2	0)			
P ₂	9	0	2	3	0	2	6	0	0			
P ₃	2	2	2	2	1	1	0	1	1			
P ₄	4	3	3	0	0	2	4	3	1			

图3-15 T₀时刻的资源分配表

3.7.2 利用银行家算法避免死锁

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3-17 P₁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 $Request_4(3, 3, 0) > Available(2, 3, 0)$, 让 P_4 等待。

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

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

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

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

3.7.2 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	2	3	2	1	0
P ₁	3	0	2	0	2	0			
P ₂	3	0	2	6	0	0			
P ₃	2	1	1	0	1	1			
P ₄	0	0	2	4	3	1			

图3-18 为P₀分配资源后的有关资源数据

3.7.2 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。

第三章 处理机调度与死锁

3.8 死锁的检测与解除

3.8 死锁的检测与解除

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

① 死锁检测算法。该方法用于检测系统状态，以确定系统中是否发生了死锁。

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

3.8.1 死锁的检测

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

1. 资源分配图(Resource Allocation Graph)

系统死锁，可利用资源分配图来描述。

3.8.1 死锁的检测

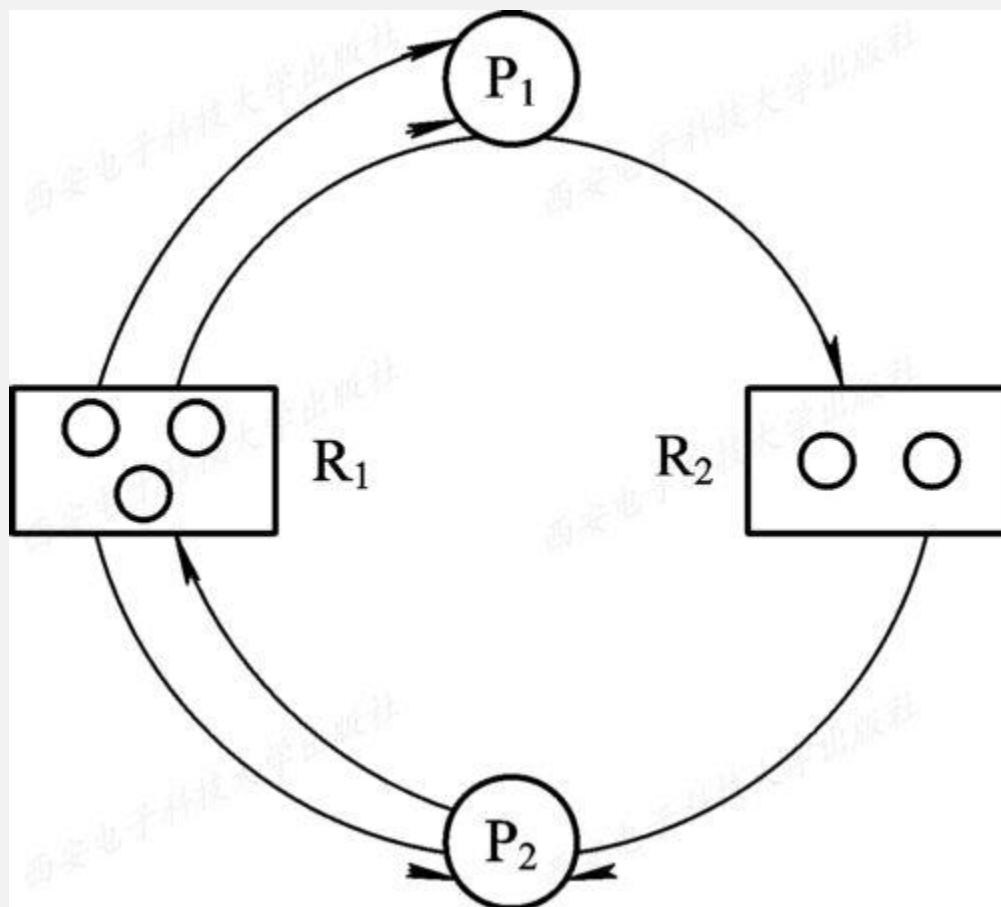


图3-19 每类资源有多个时的情况

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

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

(2) 凡属于 E 中的一个边 $e \in E$ ，都连接着 P 中的一个结点和 R 中的一个结点， $e = \{P_i, R_j\}$ 是资源请求边，由进程 P_i 指向资源 R_j ，它表示进程 P_i 请求一个单位的 R_j 资源。 $E = \{R_j, P_i\}$ 是资源分配边，由资源 R_j 指向进程 P_i ，它表示把一个单位的资源 R_j 分配给进程 P_i 。图3-19中示出了两个请求边和两个分配边，即 $E = \{(P_1, R_2), (R_2, P_2), (P_2, R_1), (R_1, P_1)\}$ 。

2. 死锁定理

我们可以利用把资源分配图加以简化的方法(图3-19), 来检测当系统处于S状态时, 是否为死锁状态。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 的请求边和分配边, 使之成为孤立的结点。在图3-20(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。

3.8.1 死锁的检测

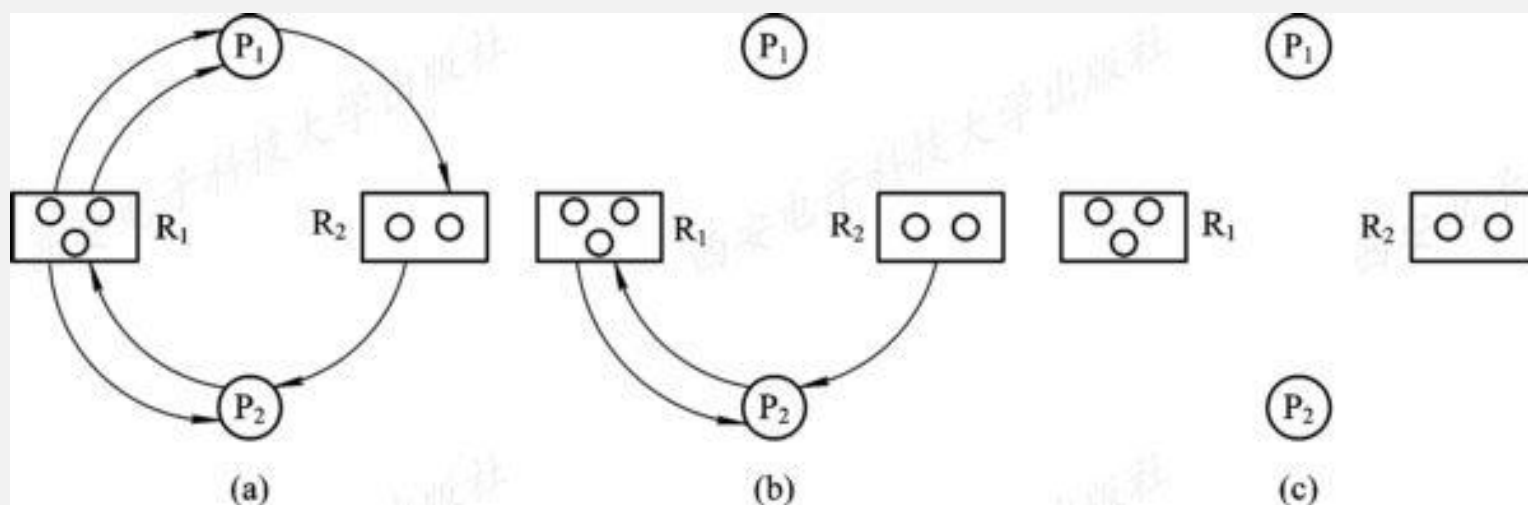


图3-20 资源分配图的简化

(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，即将 P_2 的两条请求边和一条分配边消去。

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

3.8.1 死锁的检测

3. 死锁检测中的数据结构

死锁检测中的数据结构类似于银行家算法中的数据结构：

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

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

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

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

1. 终止进程的方法

1) 终止所有死锁进程

这是一种最简单的方法，即是**终止所有的死锁进程**，死锁自然也就解除了，但所付出的代价可能会很大。因为其中有些进程可能已经运行了很长时间，已接近结束，一旦被终止真可谓“功亏一篑”，以后还得从头再来。还可能会有其它方面的代价，在此不再一一列举。

2) 逐个终止进程

稍微温和的方法是，**按照某种顺序，逐个地终止进程，直至有足够的资源，以打破循环等待，把系统从死锁状态解脱出来为止。**但该方法所付出的代价也可能很大。因为每终止一个进程，都需要用死锁检测算法确定系统死锁是否已经被解除，若未解除还需再终止另一个进程。另外，在采取逐个终止进程策略时，还涉及到应采用什么策略选择一个要终止的进程。选择策略最主要的依据是，为死锁解除所付出的“代价最小”。但怎么样才算是“代价最小”，很难有一个精确的度量。

2. 付出代价最小的死锁解除算法

一种付出代价最小的死锁解除算法如图3-21所示。

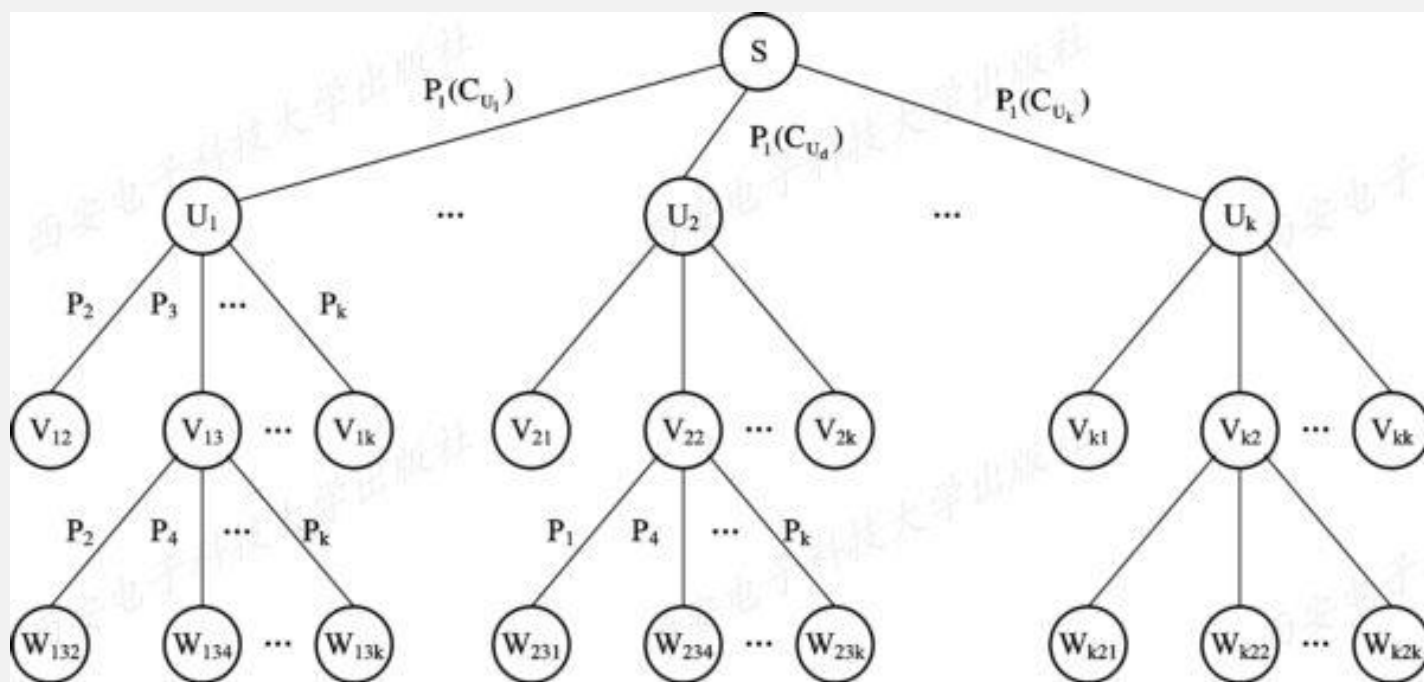



图3-21 付出代价最小的死锁解除算法



谢谢大家！ Q & A