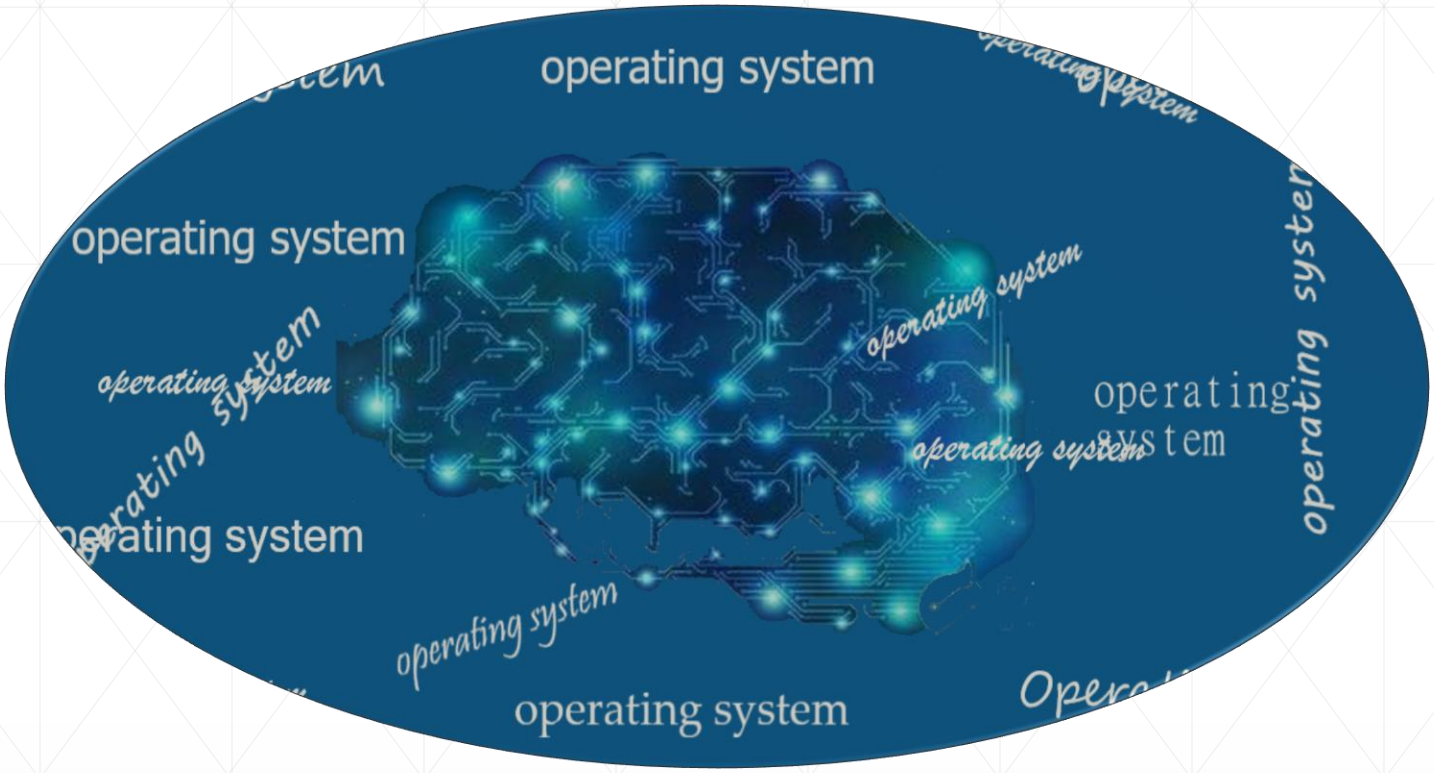


第三章 处理机调度与死锁

第三章　处理机调度与死锁

重点

- 掌握进程调度算法，各适用于何种情况
- 理解常用的几种实时调度算法
- 理解产生死锁的原因
- 掌握银行家算法避免死锁

难点

- 多道程序设计中的各种调度算法
- 响应比高者优先调度算法的计算过程
- 银行家算法

- 处理机是计算机系统中的重要资源
- 在多道程序环境下，进程数目通常多于处理机的数目
- 系统必须按一定方法动态地把处理机分配给就绪队列中的一个进程
- 处理机利用率和系统性能（吞吐量、响应时间）在很大程度上取决于处理机调度

WHAT：按什么原则分配CPU—调度算法

WHEN：何时分配CPU —调度的时机

HOW：如何分配CPU —调度过程及进程的上下文切换

作业（JOB）

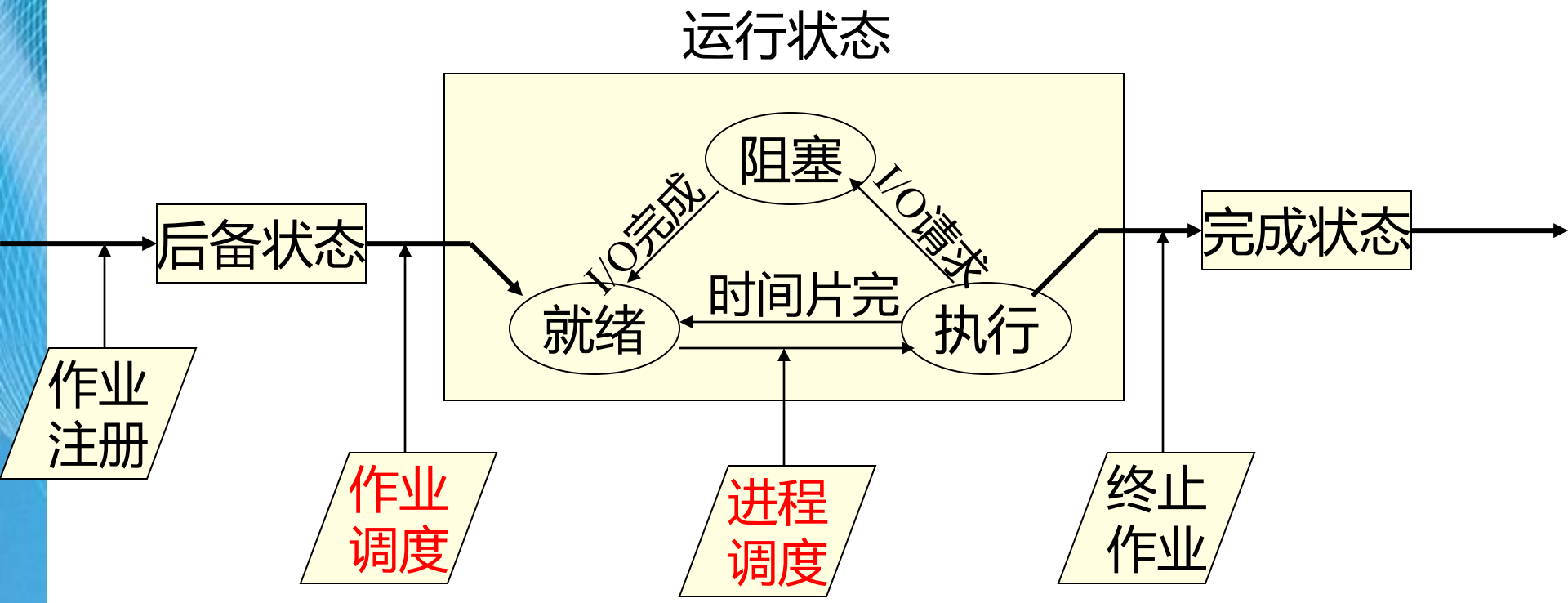
- 作业是用户在一次算题过程中或一次事务处理中，要求计算机系统所做的工作的集合
- 作业是比进程更广泛的概念，不仅包含了通常的程序和数据，而且还配有一份作业说明书，系统根据作业说明书对程序运行进行控制。在批处理系统中，以作业为单位从外存调入内存
- 用户为了让计算机完成某个特定任务，首先编写成源程序，然后提交给计算机通过编译或汇编、连接、装配、运行等步骤，最终由计算机输出用户所需要的运行结果。从计算机管理的角度看，上述**一系列的由计算机执行的任务的集合就是作业**。

作业步

- 计算机完成作业是通过执行一系列有序的工作步骤进行的，每个步骤完成作业的一部分特定工作

- 把计算机系统完成一个作业所需的一系列有序的相对独立的工作步骤称为作业步
- 作业的各个作业步虽然功能相对独立，但它们之间相互关联，往往是一个作业步的执行需要使用上一个作业步的执行结果。

作业状态间转换



作业控制块

◆ 作业提交给系统进入后备状态后，系统将为每个作业建立一个作业控制块JCB。

◆ JCB在作业的整个运行过程中始终存在，并且其内容与作业的状态同步地动态变化。只有当作业完成并退出系统时，JCB才被撤消。可以说，JCB是一个作业在系统中存在的唯一标志，系统根据JCB才感知到作业的存在

◆ 作业控制块JCB中包含了对作业进行管理的必要信息，JCB中的信息一部分是从用户提供的作业控制卡或作业说明书中得到，另一部分是记录作业运行过程中的动态信息

◆ JCB的具体内容因系统不同而异

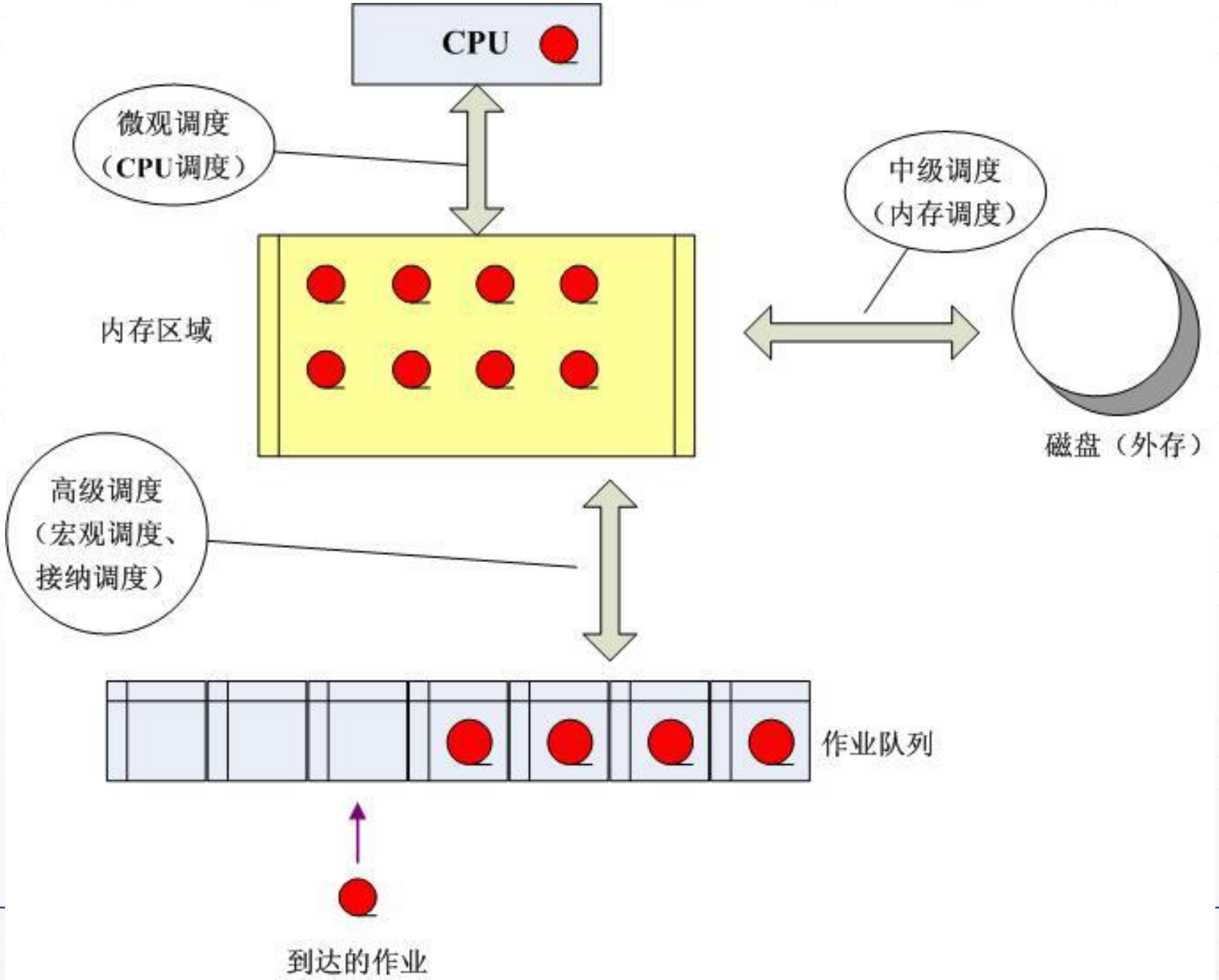
作业控制块（JCB）的内容

作业名	
资源要求	预估的运行时间 最迟完成时间 要求的内存量 要求外设类型、台数 要求的文件量和输出量
资源使用情况	进入系统时间 开始运行时间 已运行时间 内存地址 外设台号
类型级别	控制方式 作业类型 优先级
状态	
用户账户……	

3.1 处理机调度的层次

- 在多道程序系统中，一个作业从提交到执行，通常都要经历**多级调度**
  - 如高级调度、低级调度、中级调度以及I / O调度等
- 系统的**运行性能**在很大程度上取决于调度
  - 如吞吐量的大小、周转时间的长短、响应的及时性等
- 调度是多道系统的关键

- **CPU资源管理——多道程序设计面临的挑战**
  - 批处理系统：如何安排内存中多个作业的运行顺序?
  - 交互式系统：如何更好应对不同的交互式请求?
  - 实时系统：如何保证实时服务的高质量?
- **进程调度——有效的管理CPU资源**
  - When：何时进行进程调度?
  - How：遵循何种规则完成调度?
  - What：调度过程中需要完成哪些工作?
- **进程调度的级别**
  - 高级调度：决定哪些程序可以进入系统
  - 中级调度：决定内存中程序的位置和状态
  - 低级调度：决定CPU资源在就绪进程间的分配


CPU
微观调度
（CPU调度）
中级调度
（内存调度）
内存区域
磁盘（外存）
高级调度
（宏观调度、
接纳调度）
作业队列
到达的作业

3.1 处理机调度的层次

3.1.1 高级调度

高级调度(High Scheduling)：称作业调度、长程调度或接纳调度，其主要功能是根据某种算法，把外存上处于后备队列中的那些作业调入内存。批处理系统需要有作业调度，分时和实时系统无需此调度。

在每次执行作业调度时，都须做出以下两个决定：

1) 接纳多少个作业（取决于多道程序度，适当折衷）

- 作业太多 服务质量下降
- 作业太少 资源利用率低

2) 接纳哪些作业 （取决于采用的调度算法）

3.1 处理机调度的层次

3.1.1 高级调度

多道程序度：即允许多少个作业同时在内存中运行。

周转时间：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。

吞吐量：是指在单位时间内系统所完成的作业数。

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

3.1.1 高级调度

- **主要用于批处理系统。其设计目标是最大限度地发挥各种资源的利用率和保持系统内各种活动的充分并行**

一个例子：对资源需求不同的作业进行合理搭配

- 科学计算往往需要占用大量的CPU时间，属于**CPU繁忙型作业**，对于I/O设备的使用少；
- 数据处理要求占用较少的CPU时间，但要求大量I/O时间，属于**I/O繁忙型作业**；
- 有些递归计算，产生大量中间结果，需要很多内存单元存放它们，这属于**内存繁忙型作业**。
- 如果能把它们搭配在一起，程序A在使用处理机，程序B在利用通道I，而程序C恰好利用通道2等，这样一来，A、B和C从来不在同一时间使用同一资源，每个程序就好像单独在一个机器上运行

3.1.2 低级调度

- **低级调度又称为进程调度或短程调度，它所调度的对象是进程。三种类型OS都必须配置这级调度。（最基本调度）**

- **低级调度用于决定就绪队列中的哪个进程应获得处理机，然后再由分派进程执行把处理机分配给该进程的具体操作。**

- **进程调度方式：**
  - （1）**非抢占方式**：进程占用处理机直至自愿放弃或发生某事件被阻塞时，在把处理机分配给其他进程。
  - （2）**抢占方式**：允许暂停某个正在执行的进程，将处理机重新分配给另一个进程。

3.1.2 低级调度

- 1．低级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。
- (2) 分派器(调度程序)
  - 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，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
  - 当对处理机进行切换时，会发生两对上下文切换操作。

3.1.2 低级调度

➢2．低级调度的功能

(1)按某种算法选取进程（调度）。

(2)保存处理机的现场信息（上下文切换第一步骤）

(3)把处理器分配给进程（上下文切换第二步骤）。

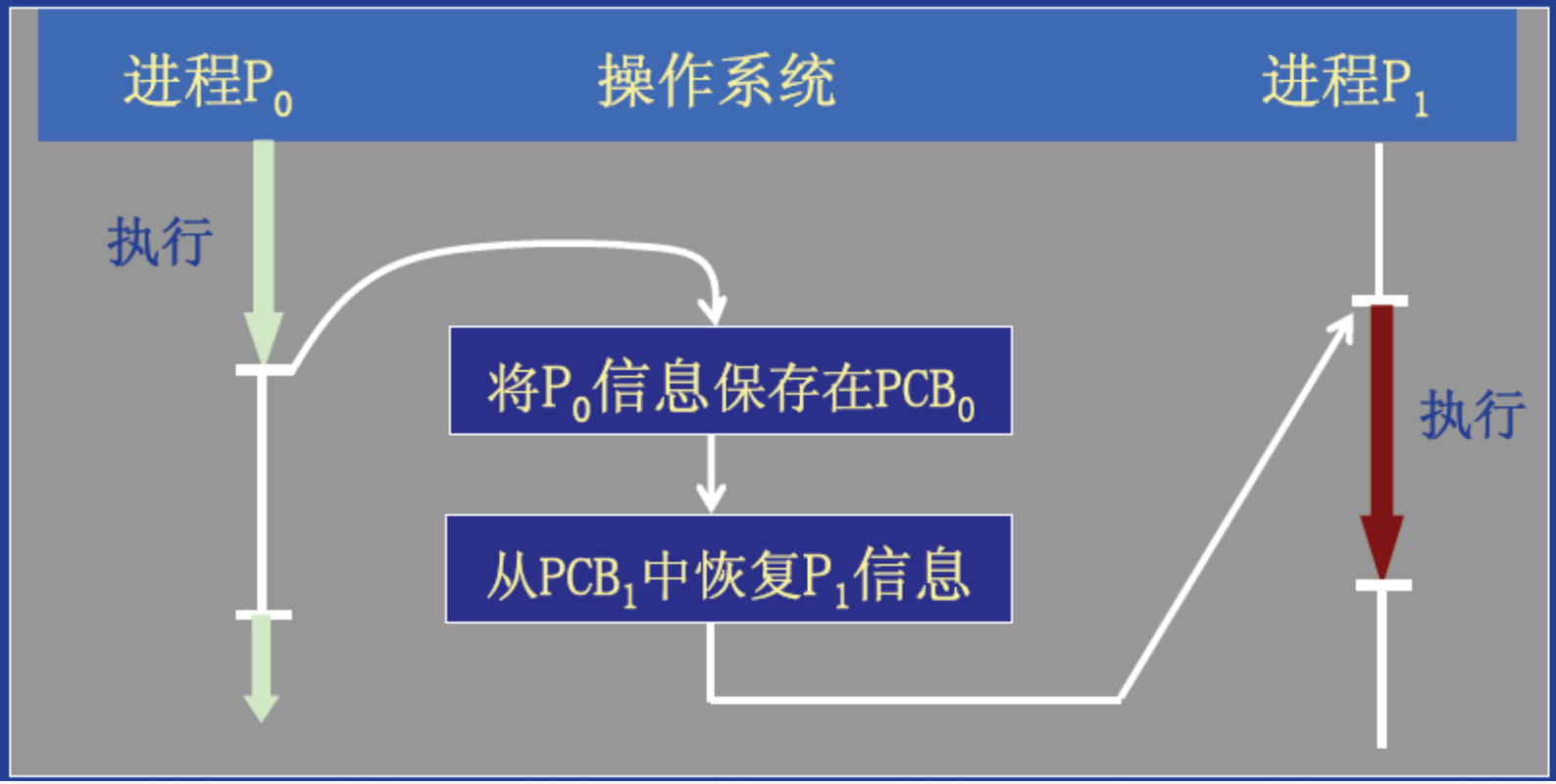
➢3. 时间尺度：通常是**毫秒级**的。

- 由于低级调度算法的**频繁使用**，要求在实现时做到**高效**。

3.1.2 低级调度

- 分派程序的主要功能：
  1. 进行进程切换；
  2. 转到用户态；
  3. 开始执行被选中的进程。

进程切换示意图



3.1.2 低级调度

1. 非抢占方式(Non-preemptive Mode)

- 进程正在处理机上执行时，新就绪的进程进入就绪队列，该进程仍继续执行，直到其完成或发生某种事件而进入完成或阻塞状态时，才转让处理机。
- 引起进程调度的因素
  - 正在执行的进程执行完毕，
  - 执行中的进程因提出I/O
  - 在进程通信或同步过程中
    wait、Block、Wakeup原

优点：算法简单，系统开销小

缺点：紧急任务不能及时响应；短进程到达要等待长进程运行结束

3.1.2 低级调度

2. 抢占方式(Preemptive Mode)

•进程正在处理机上执行时，若有某个更为重要或紧迫的进程进入就绪队列，则立即暂停正在执行的进程，将处理机分配给这个更为重要或紧迫的进程

•**优点**：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。

•**缺点**：抢占方式比非抢占方式调度所需付出的开销较大，且调度算法复杂。

2. 抢占方式(Preemptive Mode)

抢占的原则有：

(1)时间片原则。 适用于分时、大多数实时以及要求较高的批处理系统

(2)优先权原则。 重要紧急作业优先权高

(3) 短作业(进程)优先原则。

3.1.3 中级调度

中级调度(Intermediate-Level Scheduling)：又称中程调度(Medium-Term Scheduling)。

- **主要目的：为了提高内存利用率和系统吞吐量。**
- **具体实现：**
  - ✓ **使那些暂时不能运行的进程不再占用宝贵的内存资源，而将其调至外存去等待，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那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。。。。**

存储器管理中对换

课堂练习

- 分时系统中是否存在高级调度?
- 在哪一个层次的调度中涉及到内外存的交换?

  A 高级调度

  B 中级调度

  C 低级调度

否，B

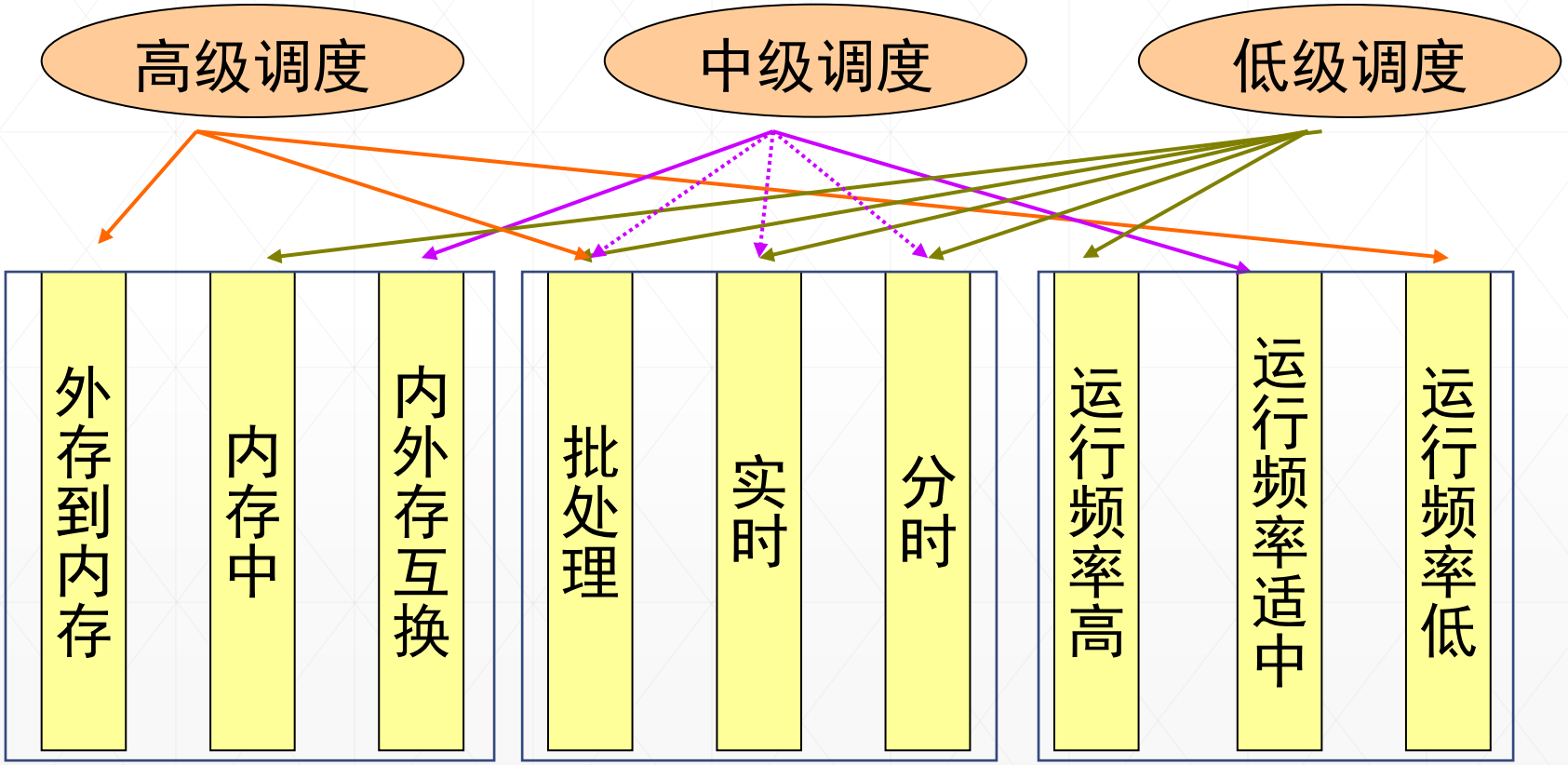
小结：处理机调度的层次

- 终端型作业：低级。
- 批量性作业：高级---低级。
- 现代较完善的os具有三级调度。
- **低级调度**运行频率最高，不宜复杂。
- **高级调度**发生在一批作业完成，重新调入一批作业到内存的时候，执行频率低
- **中级调度**介于上述两者之间。

三级调度的比较

调度类型	运行频率	运行时间	算法复杂性
进程调度	高	短	低
中程调度	中等	较短	中等
作业调度	低	长	高

三级调度的相互比较



3.2 调度队列模型和调度准则

- 引述：
  - 三级调度都涉及进程的队列。
  - 可以形成以下三种调度队列模型
    - 1、仅有进程调度
    - 2、具有高级和低级调度
    - 3、具有三级调度

3.2 调度队列模型和调度准则

3.2.1 调度队列模型

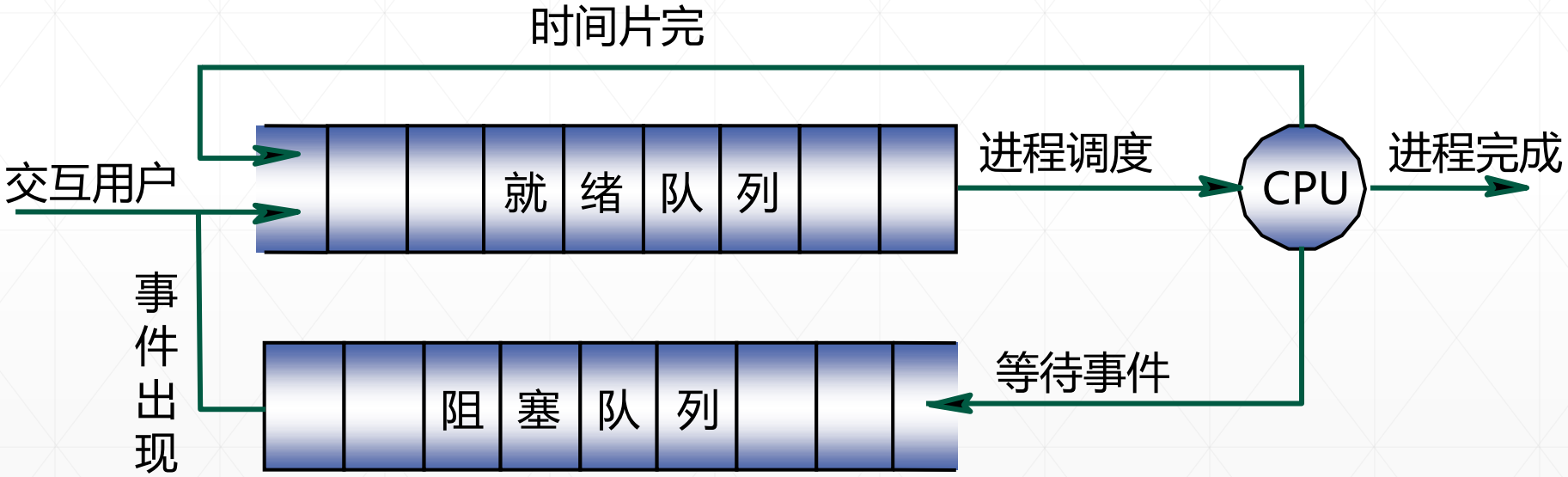
1. 仅有进程调度的调度队列模型 （分时系统）

- 在分时系统中，通常仅设有进程调度
- 系统把这些进程组织成一个就绪队列
- 每个进程在执行时，可能有以下几种情况
  - 进程获得CPU正在执行；
  - 任务在给定时间片内已完成，释放处理机后为完成状态；
  - 任务在时间片内未完成，进入就绪队列末尾；
  - 在执行期间因某事件而阻塞。

3.2 调度队列模型和调度准则

3.2.1 调度队列模型

1. 仅有进程调度的调度队列模型 （分时系统）



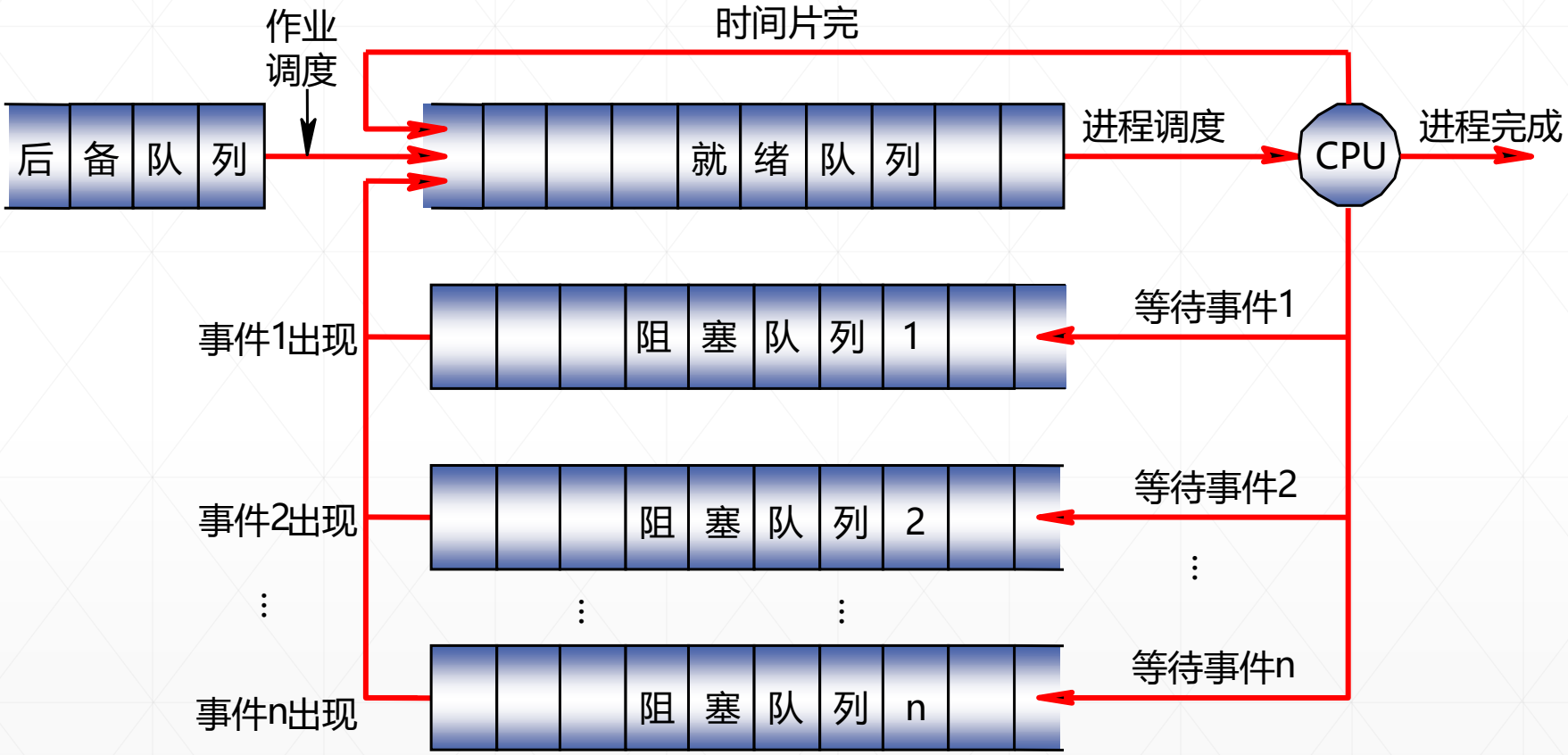
- 就绪队列时间片轮转，常采用FCFS(FIFO)算法， FCFS(FIFO)队列。
- 进程执行时三种情况：完成、时间片到、阻塞

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**，而且还要有**作业调度**
- **就绪队列的形式**
  - 在批处理系统中，常用高优先权队列。进程进入就绪队列时，按优先权高低插入相应位置，调度程序总是把处理机分配给就绪队列首进程
- **设置多个阻塞队列**
  - 根据事件的不同设置多个队列提高效率

2. 具有高级和低级调度的调度队列模型



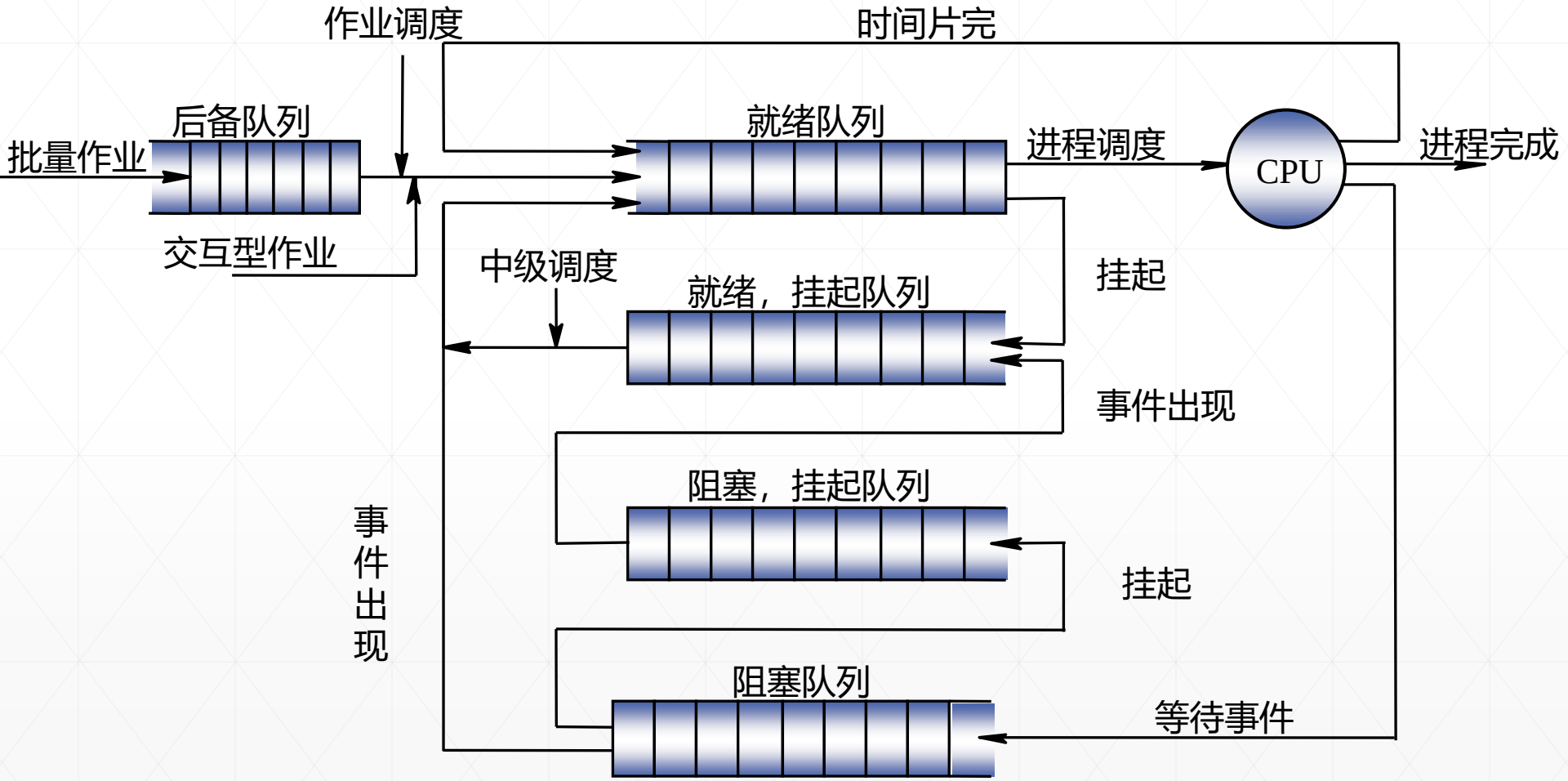
- 常采用高优先权优先调度算法
  - 可采用优先队列，进程来时按优先权插队，从队首调度 （效率高）
  - 可采用无序列表，每次调度时，先比较优先权
- 多个阻塞队列

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

- 在OS中引入中级调度后，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)。
- 同样，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。

3. 同时具有三级调度的调度队列模型



3.2.2选择调度方式和调度算法的若干准则

- 在不同的系统中通常采用不同的调度方式和算法。
- 调度的目标：
  1、提高处理机的利用率
  2、提高系统吞吐量
  3、尽量减少进程的响应时间
  4、防止进程长期得不到运行
- 系统选择调度方式和算法的准则分为两种
  - ✓面向用户的准则
  - ✓面向系统的准则

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。（评价批处理）

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）。

用来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

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

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 $W = T/T_S$，称为带权周转时间，而平均带权周转时间可表示为:

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

1. 面向用户的准则

(2) 响应时间快。
① 从键盘输入的请求信息传送到处理机的时间。
② 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。

用来评价分时系统的性能、选择进程调度算法的重要准则之一

(3) 截止时间的保证。（评价实时）
① 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。（也叫做时限，即deadline）

用来评价实时系统的性能、选择实时调度算法的重要准则之一

(4) 优先权准则。
- 让某些紧急的作业能得到及时处理。
- 往往还需选择抢占式调度方式，才能保证紧急作业得到及时处理。

适合批处理、分时和实时系统

2. 面向系统的准则

(1) 系统吞吐量高。（评价批处理系统）

- 吞吐量是指在单位时间内，系统所完成的作业数
- 与批处理作业的平均长度有关

(2) 处理机利用率高。主要对大、中型多用户系统，对单用户或实时系统不重要。

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

(3) 各类资源的平衡利用。（内存、外存、I/O设备等）主要对大、中型系统，对微型机或实时系统不重要。

引起进程调度的因素

- 引起进程调度的因素可归结为：

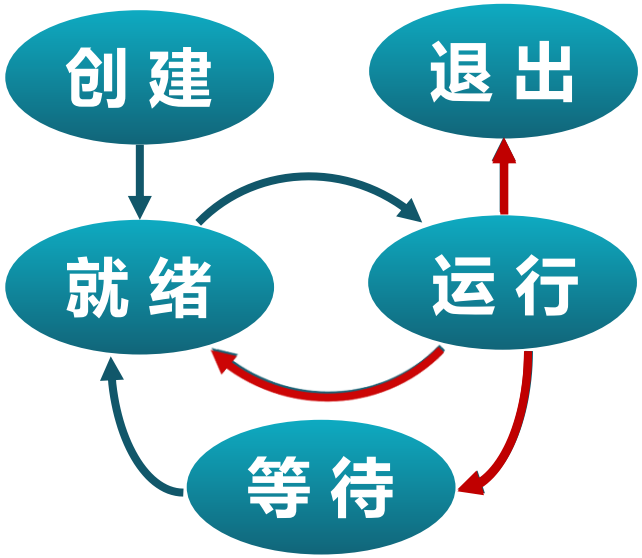
① 正在执行的进程执行完毕，或因发生某事件而不能再继续执行（包括：当前执行进程被中断、时间片用完了、挂起自己、退出等）；

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、V操作原语，Block原语， Wakeup原语等。

调度时机

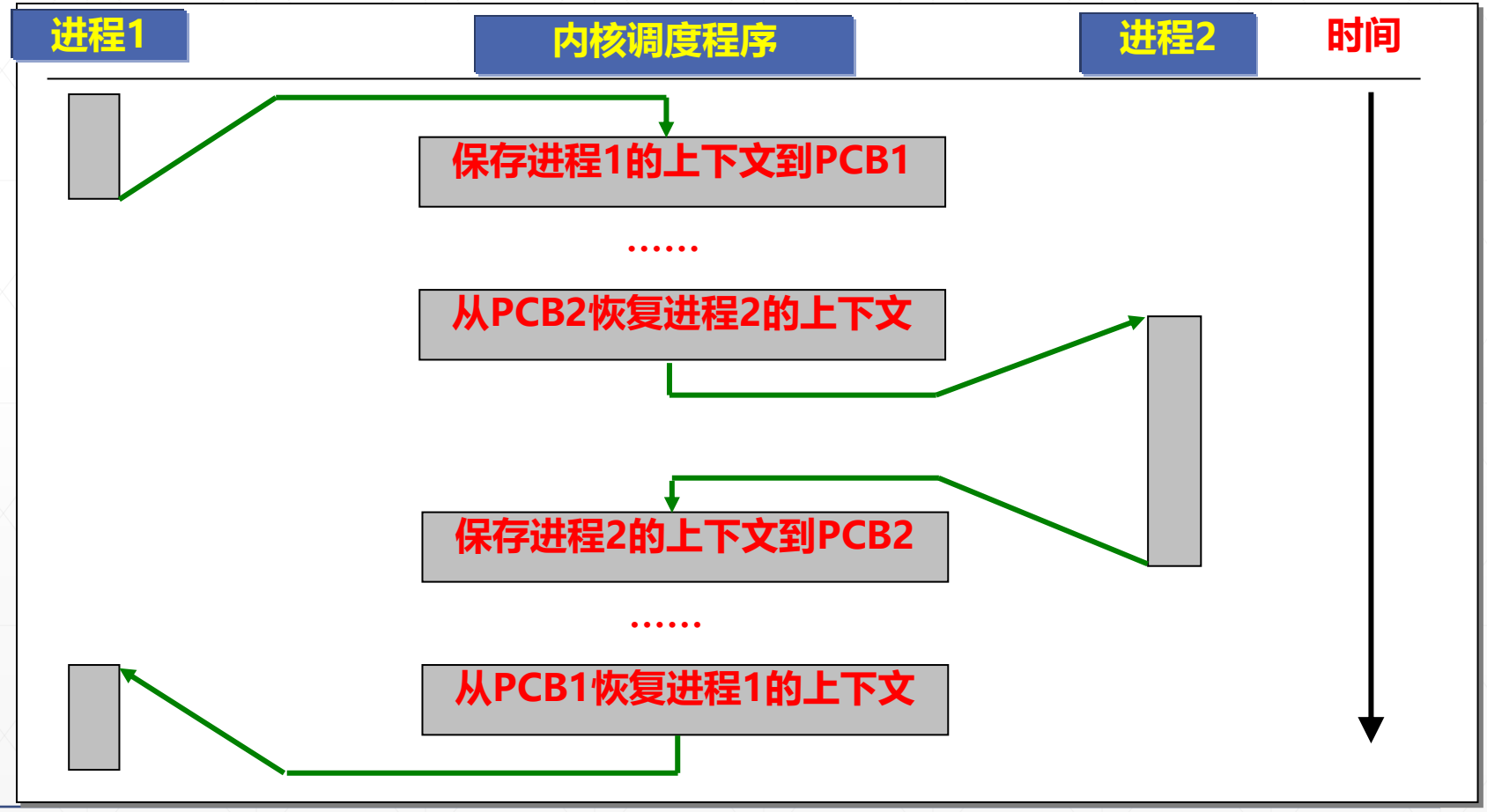
- 在进程的生命周期中的什么时候进行调度?
- 内核运行调度程序的条件
  - 进程从运行状态切换到等待状态
  - 进程被终结了
- 非抢占系统
  - 当前进程主动放弃CPU时
- 可抢占系统
  - 中断请求被服务例程响应完成时
  - 当前进程被抢占
    - 进程时间片用完
    - 进程从等待切换到就绪



进程切换

- 当一个进程占用处理机执行完（或不能继续执行），则换另一个进程占用处理机执行，称为**进程切换**。
- 把处理机分配给不同的进程占用执行，称为**进程调度**。
- 实现分配处理机的程序称为**调度程序**。
- 在进程切换时，要保护执行现场。
- 执行现场称为**进程的上下文**。

进程切换过程


进程1
内核调度程序
进程2
时间
保存进程1的上下文到PCB1
……
从PCB2恢复进程2的上下文
保存进程2的上下文到PCB2
……
从PCB1恢复进程1的上下文

进程切换基本步骤

1 保存进程上下文环境

2 更新当前运行进程的控制块内容，将其状态改为就绪或阻塞状态

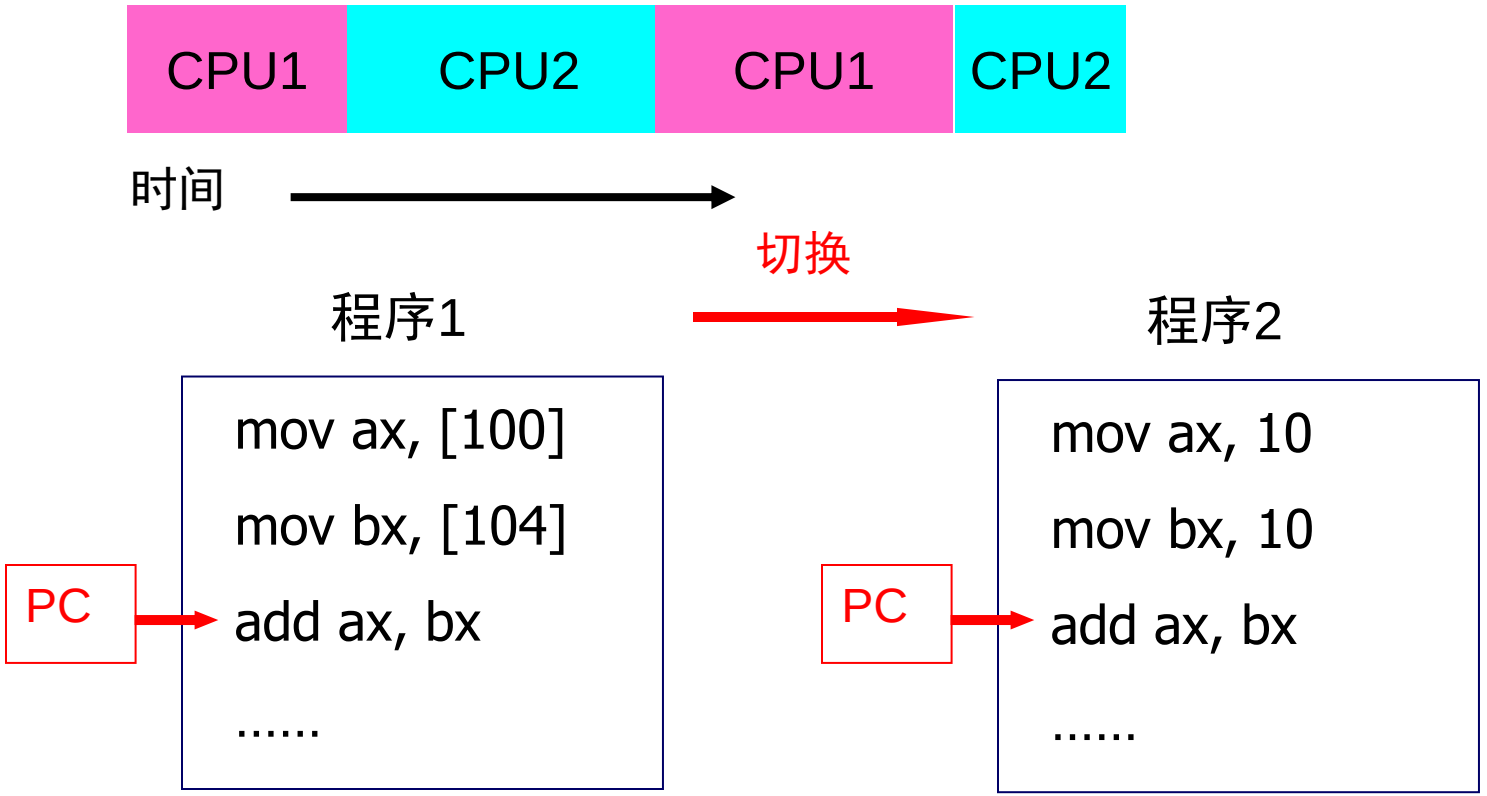
3 将进程控制块移到相应队列（就绪队列或阻塞队列）

4 改变需投入运行进程的控制块内容，将其状态变为运行状态

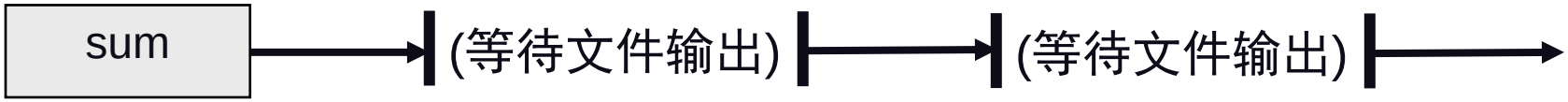
5 恢复需投入运行进程的上下文环境

并发回顾

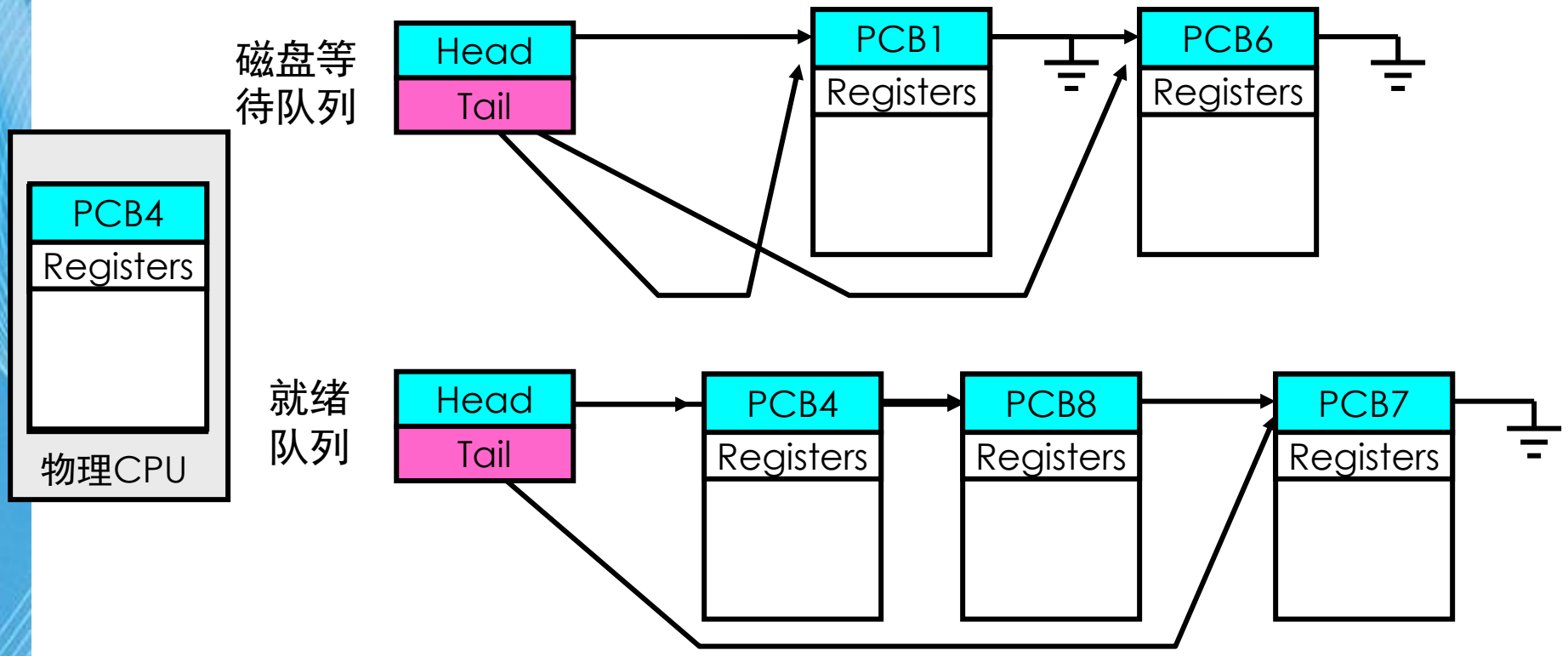
- 并发：一个CPU交替执行多个程序；可以极大地提高资源使用效率



如何让出CPU?



- 让出CPU的动作细节



让出CPU的具体实现

```
extern Queue ReadyQueue;
extern Queue DiskWaitQueue;
...
DiskWaitQueue.EnQueue(pCur);
Dispatch();
...
Dispatch()
{
    pNew = PickNext(ReadyQueue);
    Switch(pCur,pNew);
}
```

该函数就是CPU调度，调度就是下一步该选择哪一个进程或线程来执行(分配CPU资源)!

进程和线程都可能是调度单位，统称为任务

3.3 调度算法

3.3.1 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

适合于作业调度和进程调度

按照作业/进程进入系统的先后次序选择系统当前最早进入系统的作业/进程，或者先调度；即启动等待时间最长的作业/进程

优点：

- **有利于长作业（进程）**
- **有利于CPU繁忙型作业（进程）**

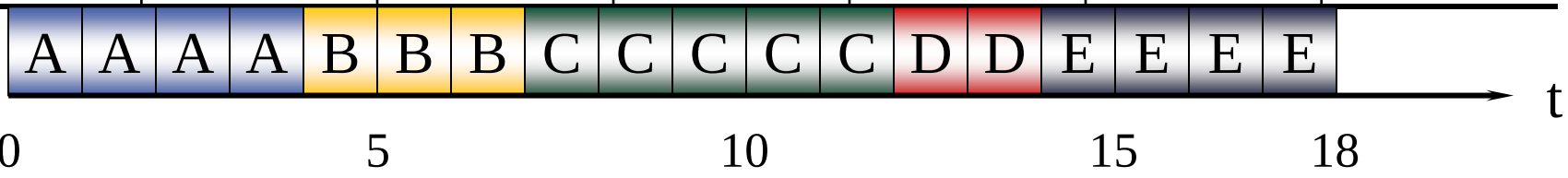
缺点：

- **不利用短作业（进程），特别是来的较晚的短作业（进程）。**
- **不利于I/O繁忙型作业（进程）**

用于批处理系统，不适于分时系统

先来先服务（先进先出）

进程名	到达时间	服务时间	开始时间	完成时间	周转时间	带权周转时间
A	0	4	0	4	4	1
B	1	3	4	7	6	2
C	2	5	7	12	10	2
D	3	2	12	14	11	5.5
E	4	4	14	18	14	3.5
平均					9	2.8



First-Come-First-Served（FCFS)

A、B、C、D四个作业分别到达系统的时间、要求服务的时间，写出开始执行时间及完成时间并计算出各自的周转时间和带权周转时间。

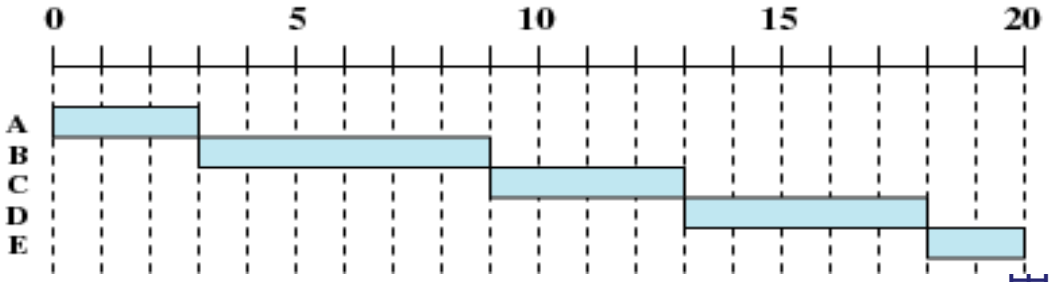
进程名	到达时间	服务时间	开始执行时间	完成时间	周转时间	带权周转时间
A	0	1	*0*	*1*	*1*	*1*
B	1	100	*1*	*101*	*100*	*1*
C	2	1	*101*	*102*	*100*	*100*
D	3	100	*102*	*202*	*199*	*1.99*

缺陷：对待短作业（进程）不公平，如果他们排在队列后面，则其等待时间远大于其执行时间。

First-Come-First-Served（FCFS）

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	3	0	3	3	1
B	2	6	3	9	7	1.17
C	4	4	9	13	9	2.25
D	6	5	13	18	12	2.4
E	8	2	18	20	12	6
					8.60	2.56

First-Come-First Served (FCFS)



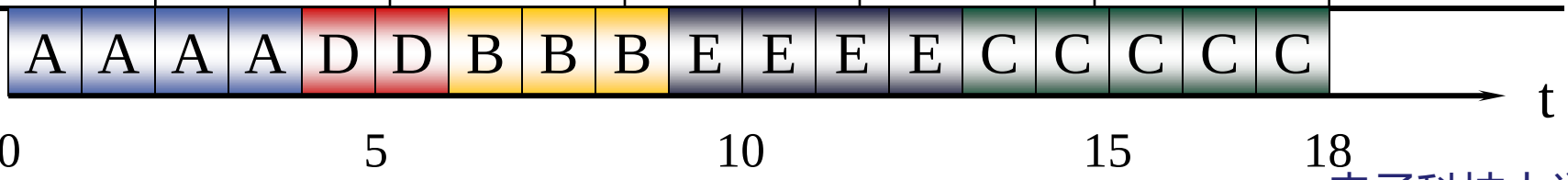
2. 短作业(进程)优先调度算法 SJ(P)F

- 以要求**运行时间长短**进行调度，即启动要求运行时间最短的作业
- 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，则是从**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。

短作业/短进程优先（SJF/SPF）：

适合于作业调度和进程调度

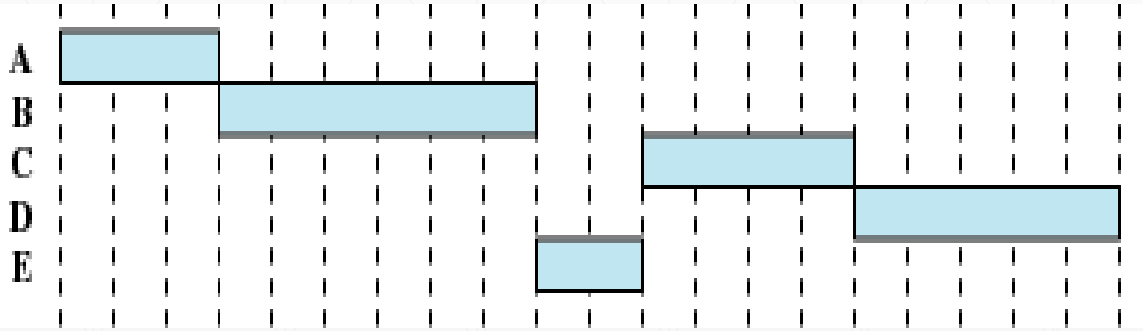
进程名	到达时间	服务时间	开始时间	完成时间	周转时间	带权周转时间
A	0	4	0	4	4	1
B	1	3	6	9	8	2.67
C	2	5	13	18	16	3.2
D	3	2	4	6	3	1.5
E	4	4	9	13	9	2.25
平均					8	2.1



短作业(进程)优先调度算法：

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	3	0	3	3	1
B	2	6	3	9	7	1.17
C	4	4	11	15	11	2.75
D	6	5	15	20	14	2.8
E	8	2	9	11	3	1.5
					7.60	1.84

Shortest Process Next (SPN)



例 FCFS和SPF调度算法的性能比较

FCFS

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	3	0	3	3	1
B	2	6	3	9	7	1.17
C	4	4	9	13	9	2.25
D	6	5	13	18	12	2.4
E	8	2	18	20	12	6
					8. 60	2.56

SPF

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	3	0	3	3	1
D	6	5	15	20	14	2.8
E	8	2	9	11	3	1.5
					7.60	1.84

SPF优点： (1)能有效降低作业/进程的平均等待时间。

(2)提高系统的吞吐量。

SJ(P)F调度算法的缺点

(1) 该算法对**长作业**不利，更严重的是可能将导致长作业(进程)长期不被调度。

(2) 该算法完全未考虑作业的**紧迫程度**，因而不能保证紧迫性作业(进程)会被及时处理。

(3) 由于作业(进程)的长短只是根据用户所提供的**估计执行时间**而定的，而用户又可能会有意或无意地缩短其作业的估计运行时间，致使该算法不一定能真正做到短作业优先调度。

(4) 无法实现人—机交互

补充内容：最短剩余时间优先调度算法 Shortest Remaining Time

- 简称SRT。
- 调度时选择**预期剩余时间最短**的进程。
- 当一个新进程加入到就绪队列时，它可能比当前运行的进程具有更短的剩余时间。因此，只要新进程就绪，调度器可能**抢占**当前正在运行的进程。
- 也可能存在长进程被饿死的危险。

抢占式SRT的实例

进程	到达时间	要求服务时间	开始执行时间	完成时间	周转时间	带权周转时间
P1	0	7 ① ⑦	0 /15	21	21	3
P2	1	5 ② ⑤	1 /7	11	10	2
P3	2	3 ③	2	5	3	1
P4	3	2 ④	5	7	4	2
P5	4	4 ⑥	11	15	11	2.75

等待时间：P1 = 14；P2 = 5；P3 = 0；P4=2；P5=7

平均等待时间：(14 + 5 + 0+2+7) / 5 = 5.6

平均周转时间：(21 + 10 + 3 + 4 + 11）/5=9.8

平均带权周转时间：(3+2+1+2+2.75）/5=2.15

等待时间=周转时间-服务时间

3.3.2 优先权调度算法 PSA

适合于作业调度和进程调度

1. 优先权调度算法的类型

1)非抢占式优先权算法（用于批处理、要求不严的实时）

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。

2) 抢占式优先权调度算法（用于要求严格的实时、性能要求较高的批处理和分时）

系统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。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。

- 只要系统中出现一个新的就绪进程，就进行优先权比较
- 该调度算法常用于要求比较严格的实时系统中，以及对性能要求较高的批处理和分时系统中

2. 优先权的类型

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**。

确定进程优先权的依据有如下三个方面：

(1) 进程类型

系统进程高，一般用户进程低。

(2) 进程对资源的需求

进程的估计执行时间、内存需求量等。要求少的进程赋予较高的优先权。

(3) 用户要求

紧迫程度、所付费用。

静态优先权法的优缺点：

优点：简单易行、系统开销小。

缺点：不够精确，可能出现优先权低的作业或进程长期得不到调度的情况。

静态优先权，非抢占式

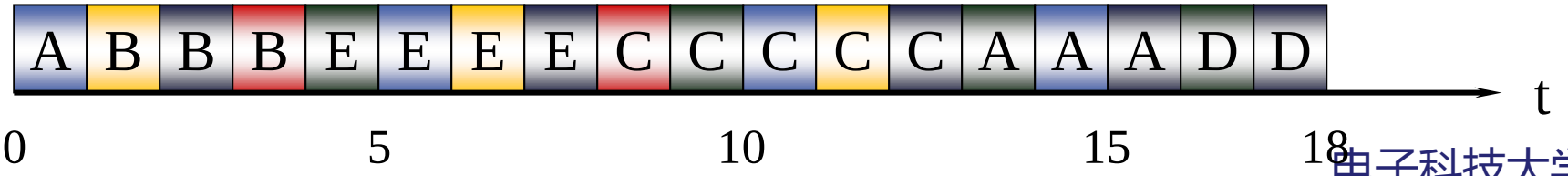
进程名	到达时间	服务时间	静态优先权	开始时间	完成时间	周转时间	带权周转时间
A	0	4	2	0	4	4	1
B	1	3	4	8	11	10	3.33
C	2	5	3	11	16	14	2.8
D	3	2	1	16	18	15	7.5
E	4	4	5	4	8	4	1
平均						9.4	2.93

考虑一下抢占式，情况如何?

就绪队列

静态优先权，抢占式

进程名	到达时间	服务时间	静态优先权	开始时间	完成时间	周转时间	带权周转时间
A	0	4	2	0	16	16	4
B	1	3	4	1	4	3	1
C	2	5	3	8	13	11	2.2
D	3	2	1	16	18	15	7.5
E	4	4	5	4	8	4	1
平均						9.8	3.14



2) 动态优先权

- 随进程的推进或随其等待时间的增加而改变，以获得更好的调度性能
- 可规定，在就绪队列中的进程，随其等待时间的增长，其优先权以某一速率提高
  - 具有相同优先权初值的进程进入就绪队列，FCFS算法
  - 具有各不相同的优先权初值的就绪进程，则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某一速率下降，则可防止一个长作业长期地垄断处理机

问题

- 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。

有没有什么办法既考虑到短作业，又能够估计长作业呢?

- 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。

3. 高响应比优先调度算法（动态优先权机制）

HRRN(Highest Response Ratio Next)

优先权的变化规律可描述为：

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

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：

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

非抢占

HRRN

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	3	0	3	3	1
B	2	6	3	9	7	1.17
C	4	4	9	13	9	2.25
D	6	5	15	20	14	2.8
E	8	2	13	15	7	3.5
					8.0	2.14

- Rc=(9-4+4)/4=2.25　　Rd=(9-6+5)/5=1.6

  Re=(9-8+2)/2=1.5

- **调度C执行。**

- Rd=(13-6+5)/5=2.4　　Re=(13-8+2)/2=3.5

- **调度E执行。**

例 FCFS和HRRN调度算法的性能比较

FCFS

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	3	0	3	3	1
B	2	6	3	9	7	1.17
C	4	4	9	13	9	2.25
D	6	5	13	18	12	2.4
E	8	2	18	20	12	6
					8. 60	2.56

HRRN

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	3	0	3	3	1
B	2	6	3	9	7	1.17
C	4	4	9	13	9	2.25
D	6	5	15	20	14	2.8
E	8	2	13	15	7	3.5
					8.0	2.14

例 HRN和SPF调度算法的性能比较

HRRN

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	3	0	3	3	1
B	2	6	3	9	7	1.17
C	4	4	9	13	9	2.25
D	6	5	15	20	14	2.8
E	8	2	13	15	7	3.5

8.0 2.14

SPF

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	3	0	3	3	1
B	2	6	3	9	7	1.17
C	4	4	9	13	9	2.25
D	6	5	15	20	14	2.8
E	8	2	9	11	3	1.5

7.60 1.84

HRRN是介于FCFS和SJ(P)F之间的一种折中算法。由于长作业也有机会投入运行，在同一时间内处理的作业数显然要少于SJ(P)F法，从而采用HRRN方式时其吞吐量将小于采用SJF 法时的吞吐量。另外，由于每次调度前要计算响应比，系统开销也要相应增加。

非抢占

HRRN

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	3	0	3	3	1
B	2	6	3	9	7	1.17
C	4	4	9	13	9	2.25
D	6	5	15	20	14	2.8
E	8	2	13	15	7	3.5
					8.0	2.14

抢占

HRRN

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	3	0	5	5	1.67
B	2	6	2	15	13	2.17
C	4	4	5	11	7	1.75
D	6	5	15	20	14	2.8
E	8	2	11	13	5	2.5
					8.8	2.18

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

对HRRN的小结

- **等待时间相同**的作业，则**要求服务的时间愈短**，其**优先权愈高**，
  ——对短作业有利
- **要求服务的时间相同**的作业，则**等待时间愈长**，其**优先权愈高**，
  ——是先来先服务
- 长作业，优先权**随等待时间的增加**而提高，其等待时间足够长时，其优先权便可升到很高，从而也可获得处理机
  ——对长作业有利
- 是一种折衷，既照顾了短作业，又考虑了作业到达的先后次序，又不会使长作业长期得不到服务。

缺点：要进行响应比计算，增加了系统开销

高响应比优先，非抢占式

进程名	到达时间	服务时间	响应比	开始时间	完成时间	周转时间	带权周转时间
A	0	4		0	4	4	1
B	1	3	2	4	7	6	2
C	2	5	12.4	9	14	12	2.4
D	3	2	1.35	7	9	6	3
E	4	4	1.75	14	18	14	3.5
平均						8.4	2.38

$$\text{优先权} = \frac{\text{等待时间} + \text{要求服务时间}}{\text{要求服务时间}} = \frac{\text{当前时间-到达时间} + \text{服务时间}}{\text{服务时间}}$$

高响应比优先，抢占式

进程	到达时间	要求服务时间	开始执行时间	完成时间	周转时间	带权周转时间
P1	**8.0**	**2.0** ①③⑦	8.0/8.8/10.1	11.3	3.3	1.65
P2	**8.6**	**0.6** ②⑤	8.6/9.2	9.6	1.0	1.67
P3	**8.8**	**0.2** ④	9.0	9.2	0.4	2.0
P4	**9.0**	**0.5** ⑥	9.6	10.1	1.1	2.2
平均					1.45	1.88

3.3.3 基于时间片的轮转调度算法

适合于进程调度

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**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。当时间片用完时，由一个计时器发出时钟中断请求，调度程序便根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。

保证就绪队列中的所有进程，在一给定的时间内，均能获得一个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。

(2) 时间片大小的确定

退化成

- 时间片太大 ⟶ FCFS算法
- 时间片过小切换开销大。

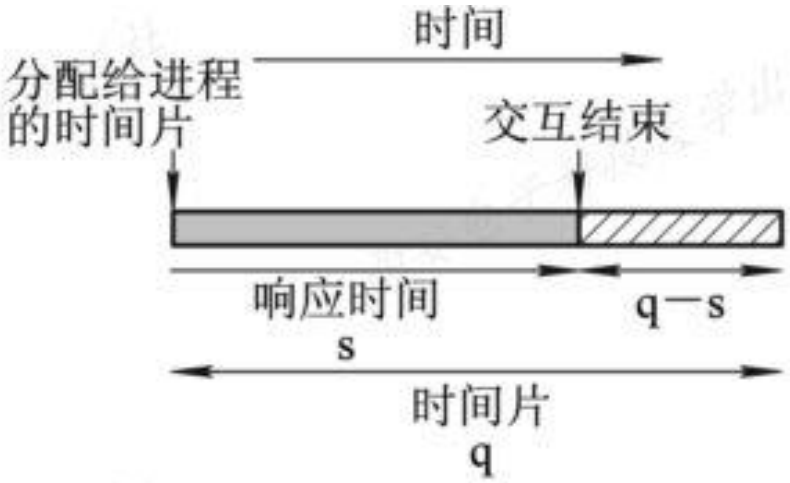
时间片大小确定要考虑的因素：

(1) 系统对响应时间的要求。（用户数一定时，成正比）

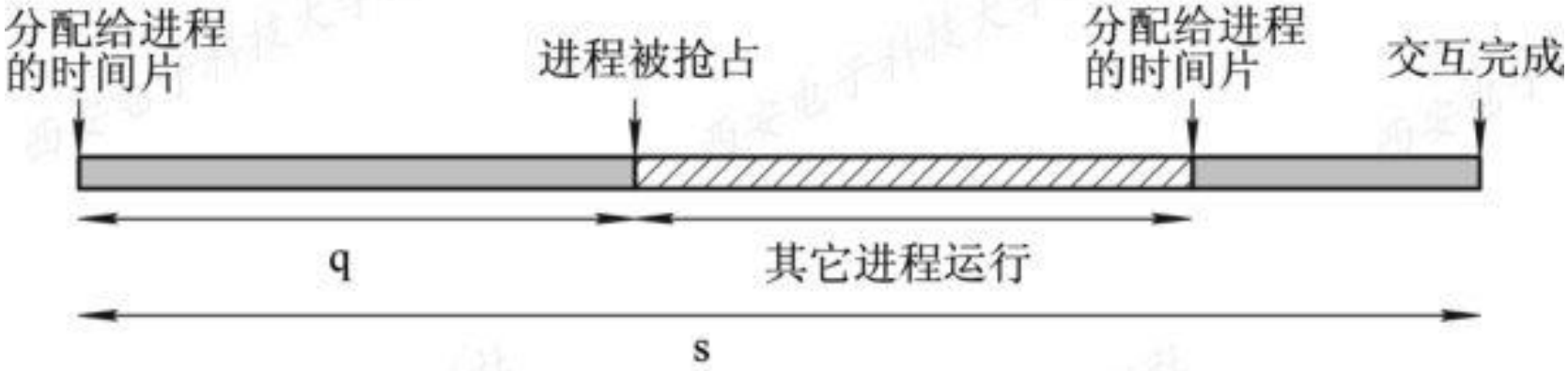
(2)就绪队列中的进程数目。（保证响应时间，成反比）

(3)系统的处理能力。（保证用户键入的命令能在一个时间片内处理完毕）

时间片大小对响应时间的影响



(a) 时间片大于交互时间

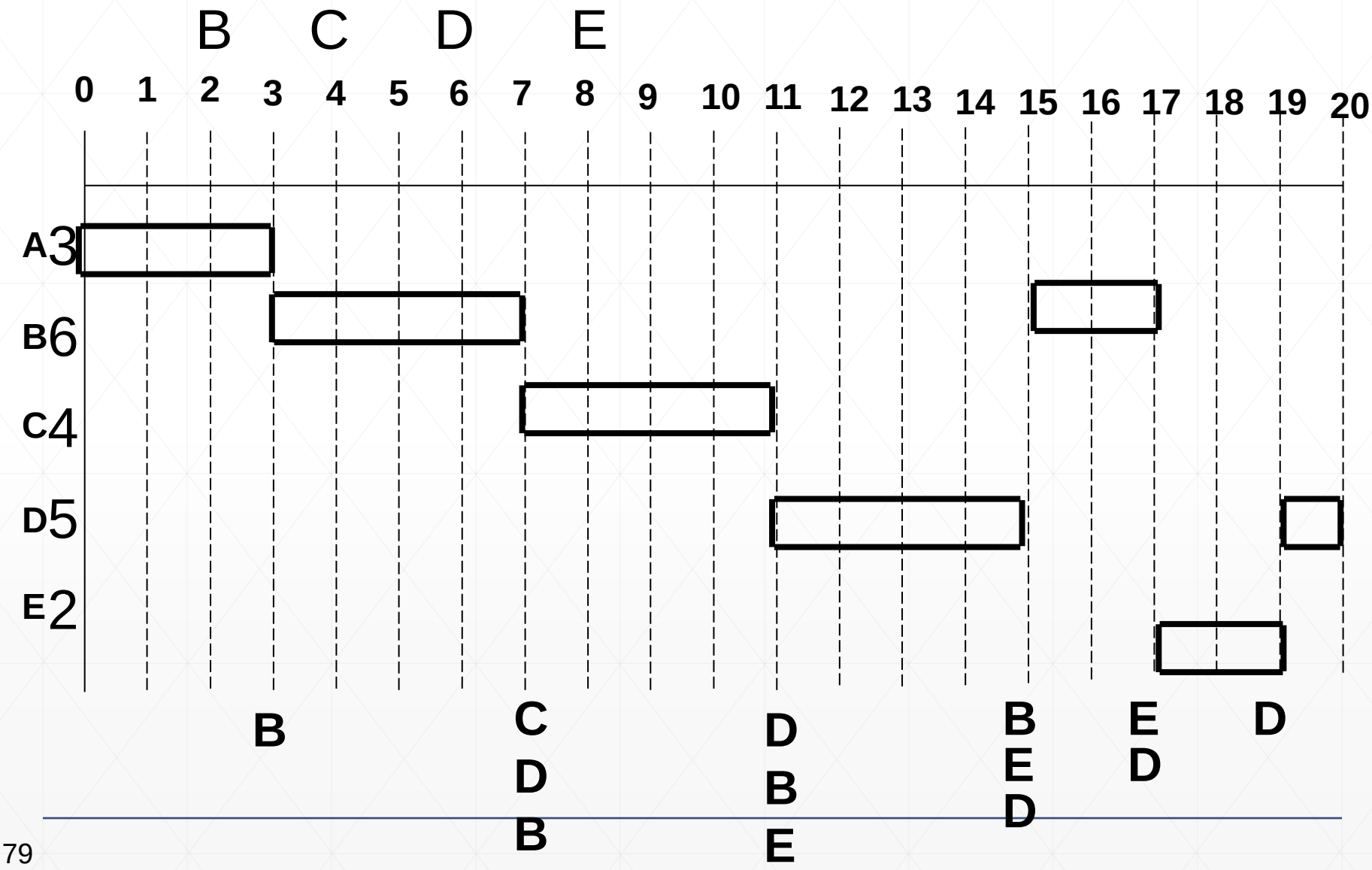


(b) 时间片小于交互时间

例 1

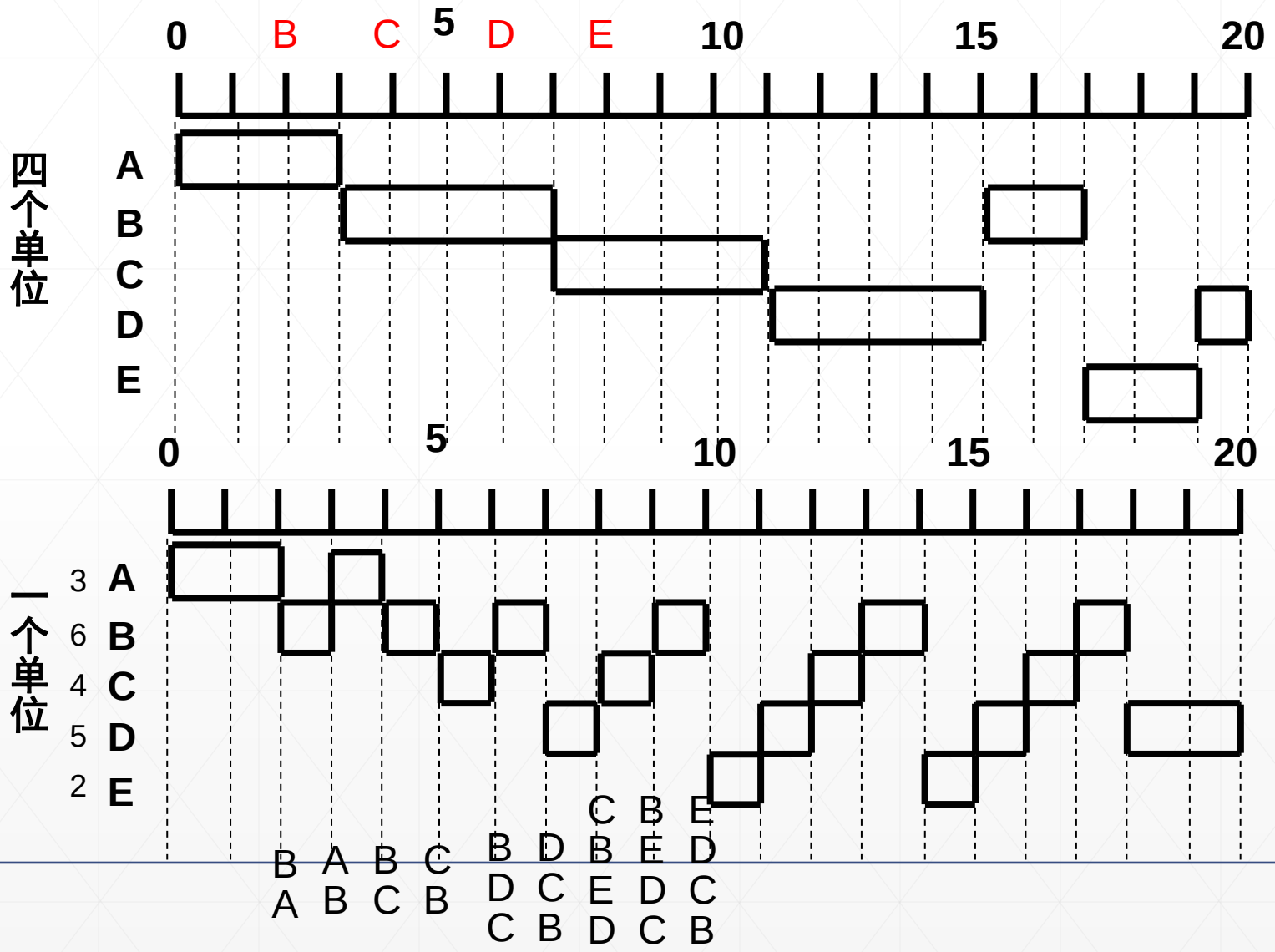
Process	Arrival Time	Service Time
A	0	3
B	2	6
C	4	4
D	6	5
E	8	2

RR(q=4)


B C D E
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20
A 3
B 6
C 4
D 5
E 2
B
C D B
D B E
B E D
E D
D

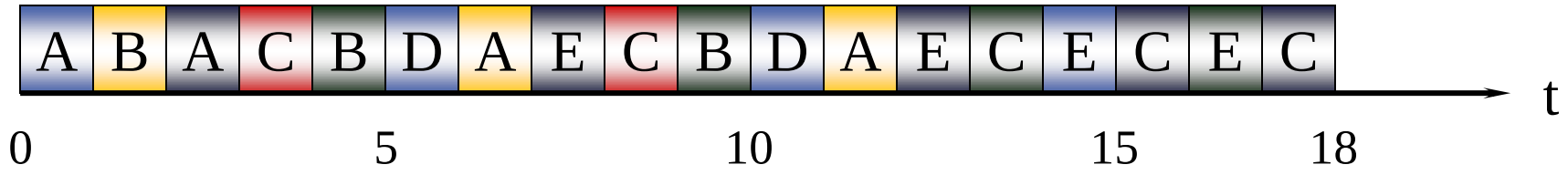
79

- **RR(q=1)**


四个单位
0
B
C
5
D
E
10
15
20
A
B
C
D
E
0
5
10
15
20
一个单位
3 A
6 B
4 C
5 D
2 E
B A
A B
B C
C B
B D C
D C B
C B E D
B E D C
E D C B

课堂练习 RR q=1

进程名	到达时间	服务时间	开始时间	完成时间	周转时间	带权周转时间
A	0	4	0	12	12	3
B	1	3	1	10	9	3
C	2	5	3	18	16	3.2
D	3	2	5	11	8	4
E	4	4	7	17	13	3.25
平均					11.6	3.29



时间片轮转调度法优缺点

1.时间片的大小对计算机性能的影响。

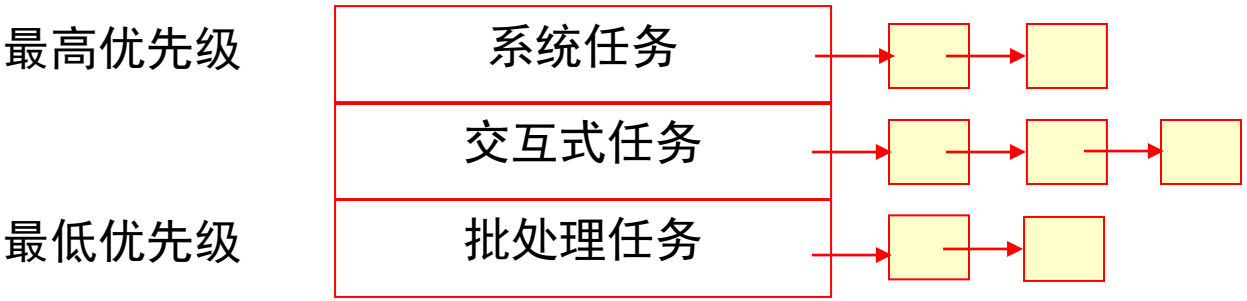
2.存在的问题：未有效利用系统资源。

对于短的、计算型进程比较有利，因为该进程充分利用时间片，而I/O型进程却不利，因为在两次I/O之间仅需很少的CPU时间，却需要等待一个时间片。

3.常用于分时系统及事务处理系统。

3.3.4 混合多种调度算法 – 多级队列调度

- **多个就绪队列，不同的队列采用不同的调度方法**




```
if(!IsEmpty(KernelQ)){ next=Pri(); return;}
if(!IsEmpty(ResponseQ)){ next=RR(); return;}
if(!IsEmpty(BatchQ)){ next=SJF(); return;}
...
```

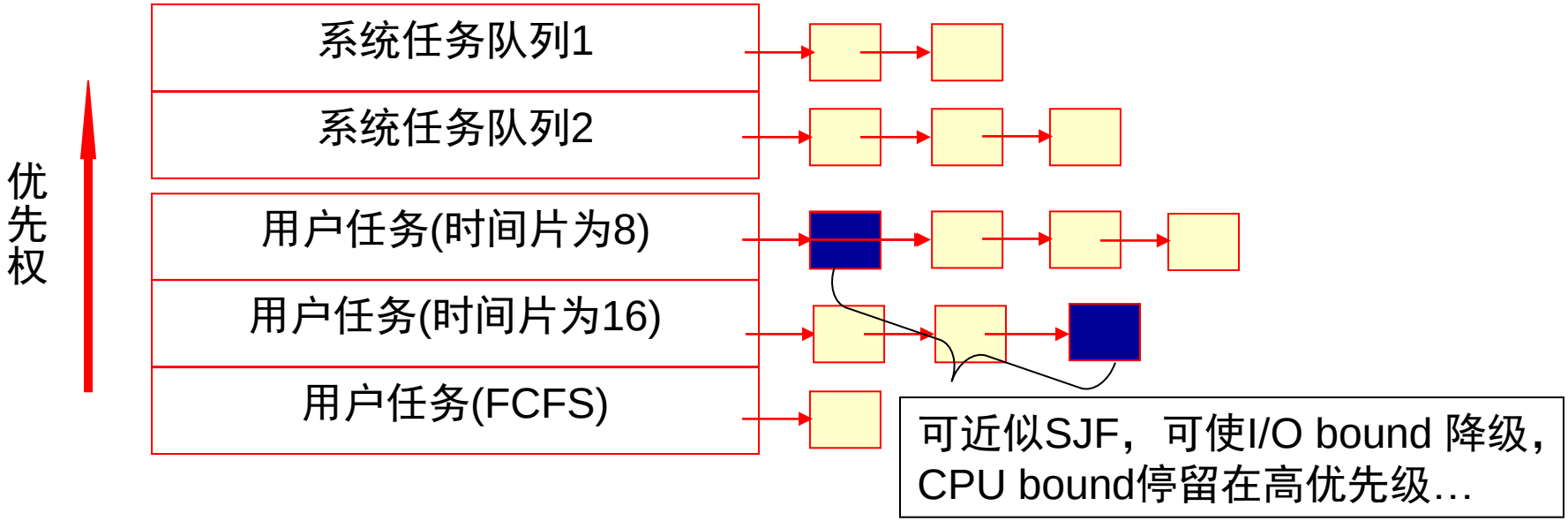
- 存在问题: 没法区分I/O bound和CPU bound。

3.3.4 多级队列调度算法

- 前台的就绪队列是交互性作业的进程，采用时间片轮转。
- 后台的就绪队列是批处理作业的进程，采用优先权或短作业优先算法。
- 调度方式有两种：
  - 优先调度前台，若前台无可运行进程，才调度后台。
  - 分配占用CPU的时间比例，如：前台80%，后台20%

3.3.5 更成熟的多级队列调度 – 多级反馈队列

- **任务可以在队列之间移动，更细致的区分任务**



- **最通用的调度算法，多数OS都使用该方法或其变形，如UNIX、Windows等。**

3.3.5 多级反馈队列调度算法

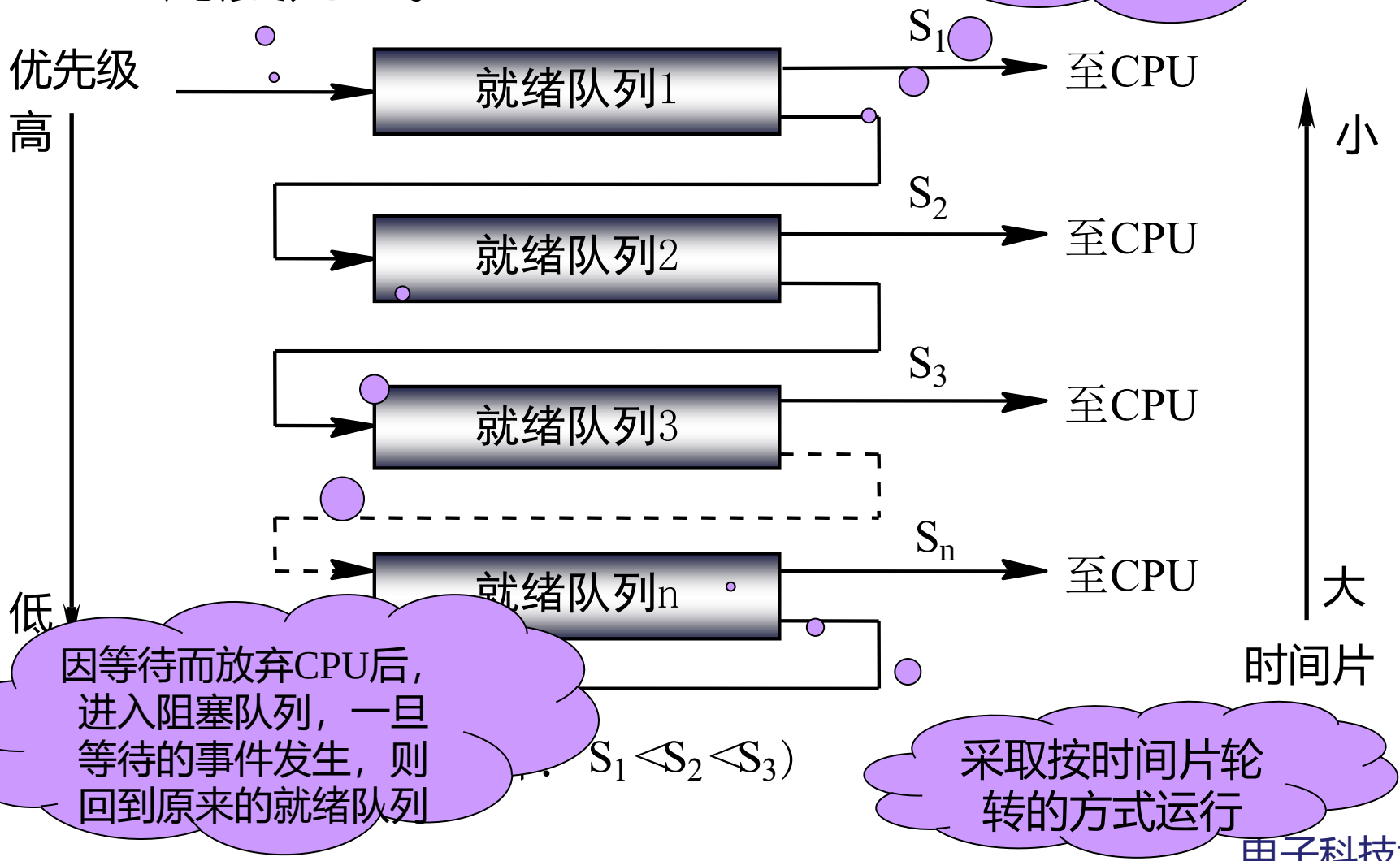
- a. 调度算法
  - (1) 设置多个就绪队列，并为各个队列赋予不同的优先级。第一个队列的优先级最高，第二个次之，其余各队列的优先级逐个降低。规定在优先权越高的队列，每个进程的时间片就越小。
  - (2) 一个新进程进入内存后，首先将它放入第一队列的末尾，按FCFS原则排队等待调度。当轮到该进程执行时，如它能在该时间片内完成，便可撤离系统；如果在一个时间片内尚未完成，调度程序便将它转入第二队列末尾，再按FCFS原则等待调度执行；如果在第二队列中运行一个时间片后仍未完成，再依次放入第三队列，……，如此下去，当一个长作业(进程)从第一队列依次降到第n队列后，在第n队列中便采取按时间片轮转方式运行。

(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。

按FIFO原则排队等待调度

尚未完成转入第二队列的末尾，按FIFO原则等待调度

调度方式



因等待而放弃CPU后，进入阻塞队列，一旦等待的事件发生，则回到原来的就绪队列

采取按时间片轮转的方式运行

多级反馈队列调度算法过程

(1)按优先级由高到低设置多个队列 RQ_0，RQ_1 … RQ_n，高优先级队列时间片小。

(2)刚进入系统的进程按FCFS放入最高的RQ_0中。

(3)进程一次时间片没执行完，就降至下一级队列，以此类推，降至最低优先级队列后，一直在此队列中不再下降。

(4)系统优先调度高优先级队列中的进程，仅当RQ_0空闲时才调度RQ_1队列进程，以此类推。

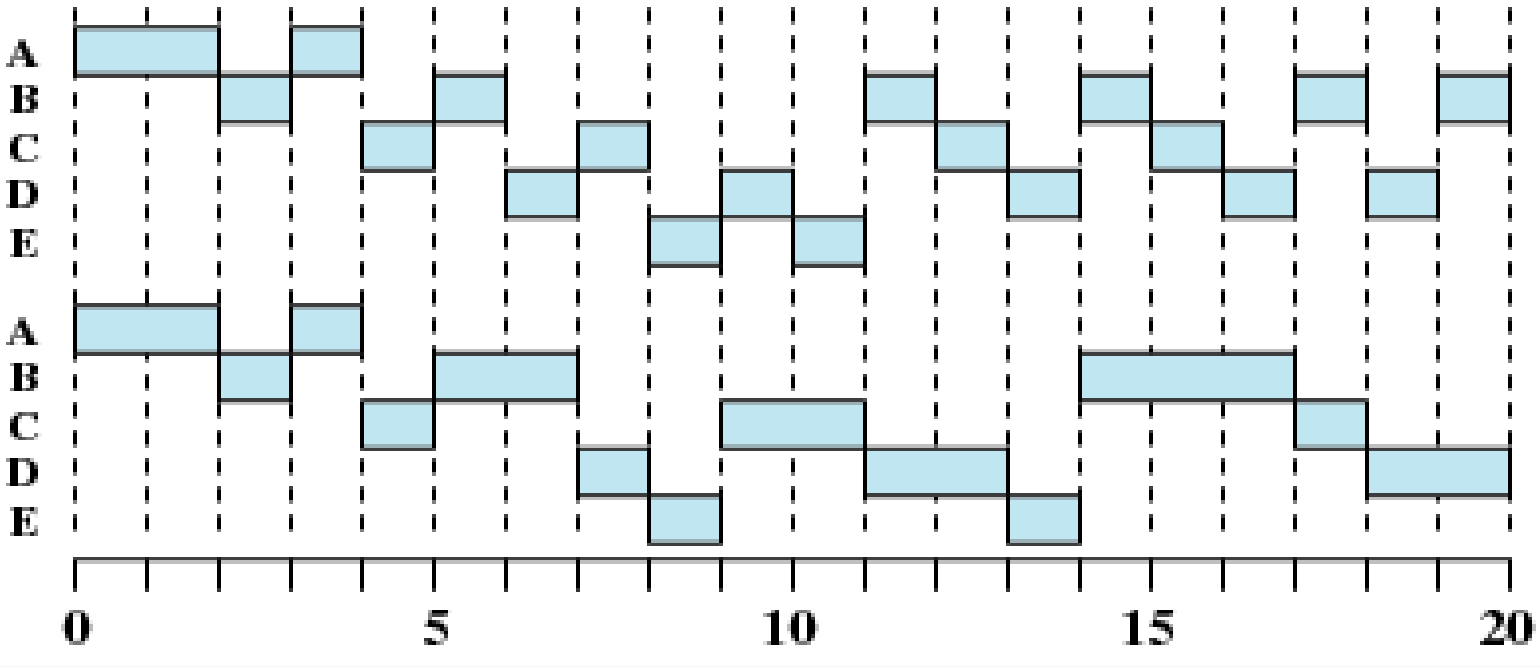
- **不需要知道进程执行所需时间**

多级反馈队列调度算法

Process	Arrival Time	Service Time
A	0	3
B	2	6
C	4	4
D	6	5
E	8	2

q=1

q=2^{i-1}



b. 多级反馈队列调度算法的性能

(1)终端型作业用户。

交互型作业，通常较小，第一队列一个时间片即可完成

(2) 短批处理作业用户。

第一队列一个时间片即可完成，或第一队列、第二队列各一个时间片

(3) 长批处理作业用户。

可能到第N个队列，按时间片轮转，不必担心得不到处理

调度算法

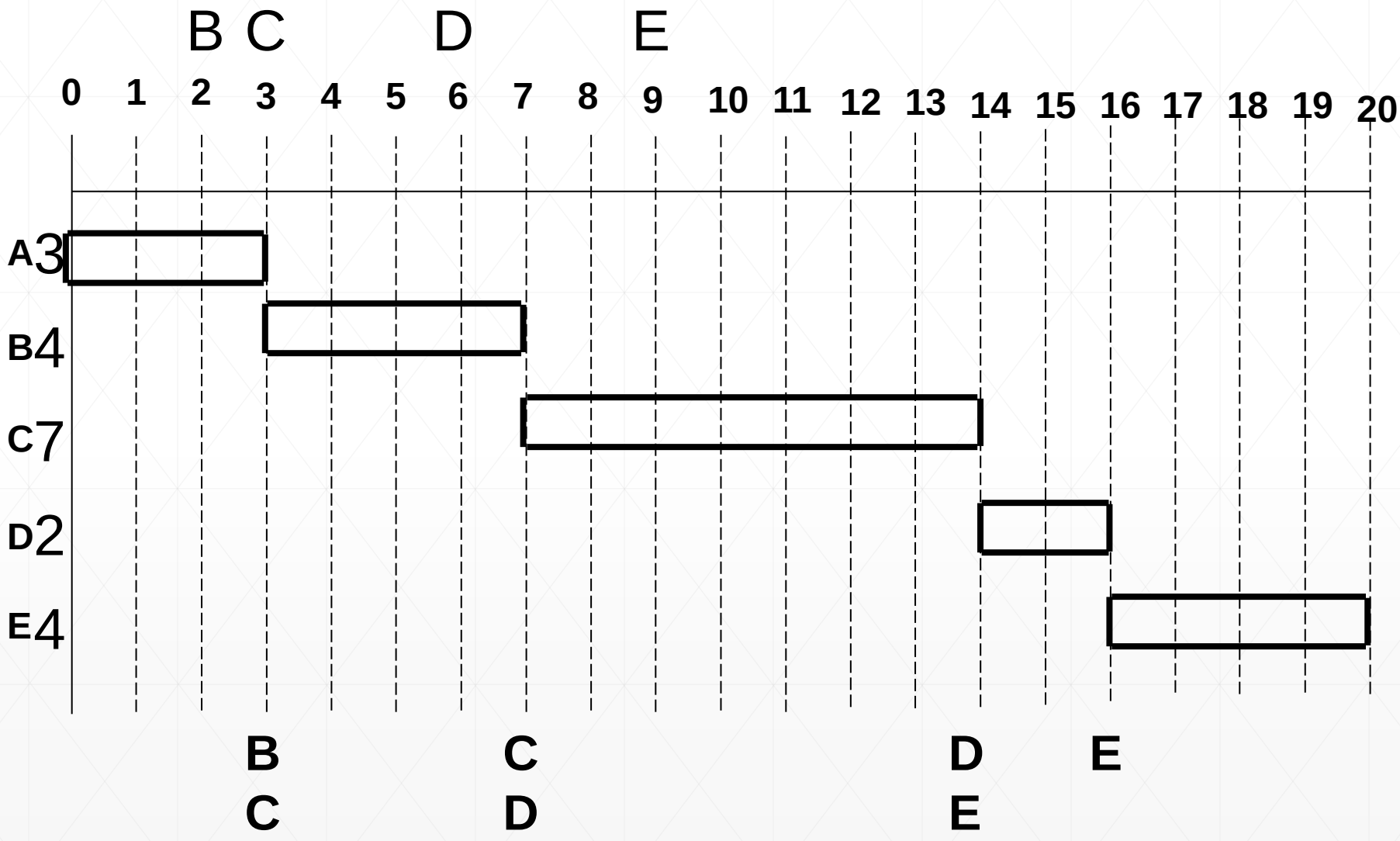
- 练习题1
  - 请按照FCFS、SPF、HRRN、RR(1)算法对上面的进程进行调度，要求画出调度过程。

进程名	达到时间	服务时间
A	0	3
B	1	6
C	3	2
D	5	5
E	7	4

- 练习题2
  - 请按照FCFS、SPF、HRRN、RR(2)算法对上面的进程进行调度，要求画出调度过程。

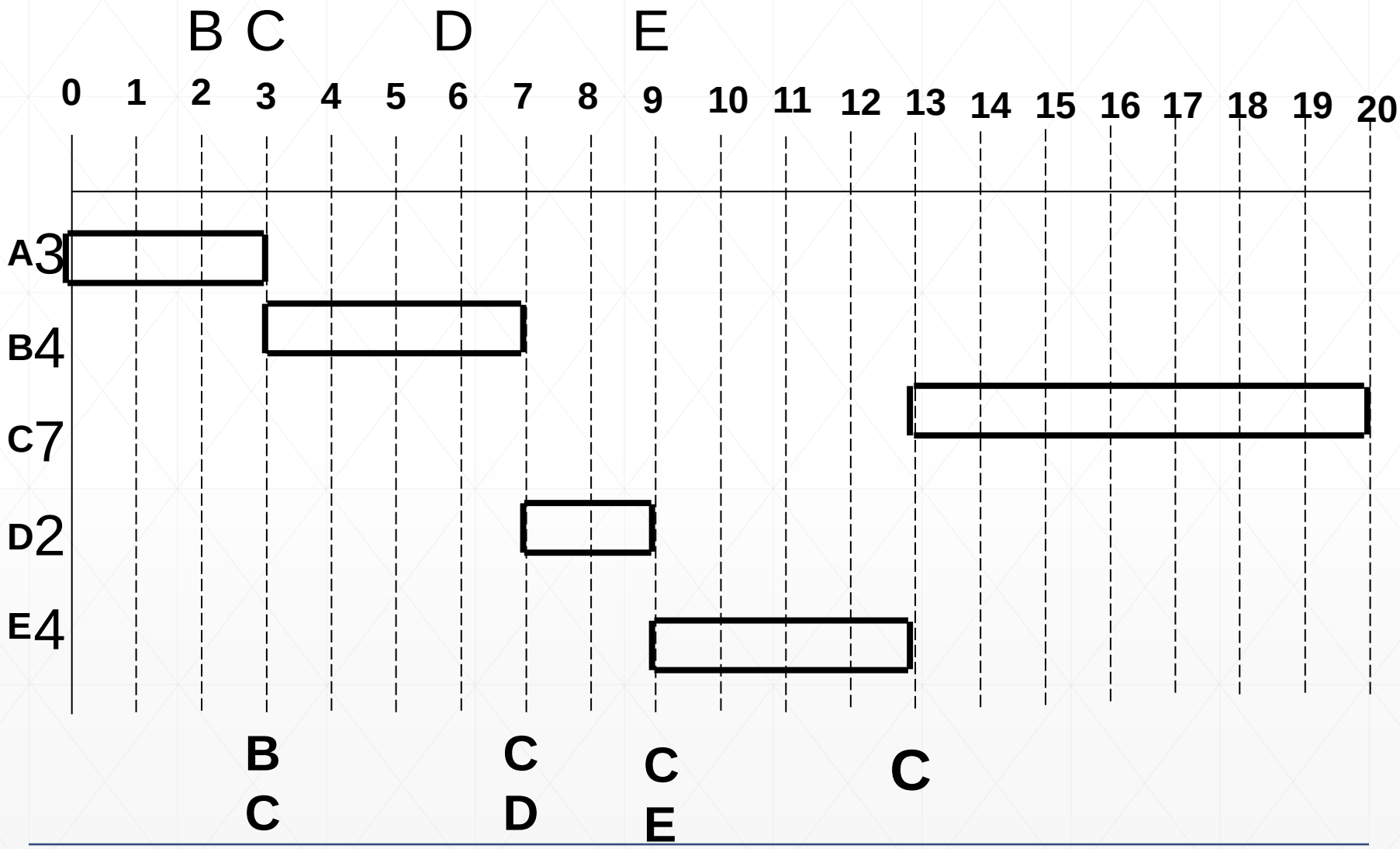
进程名	达到时间	服务时间
A	0	3
B	2	4
C	3	7
D	6	2
E	9	4

FCFS

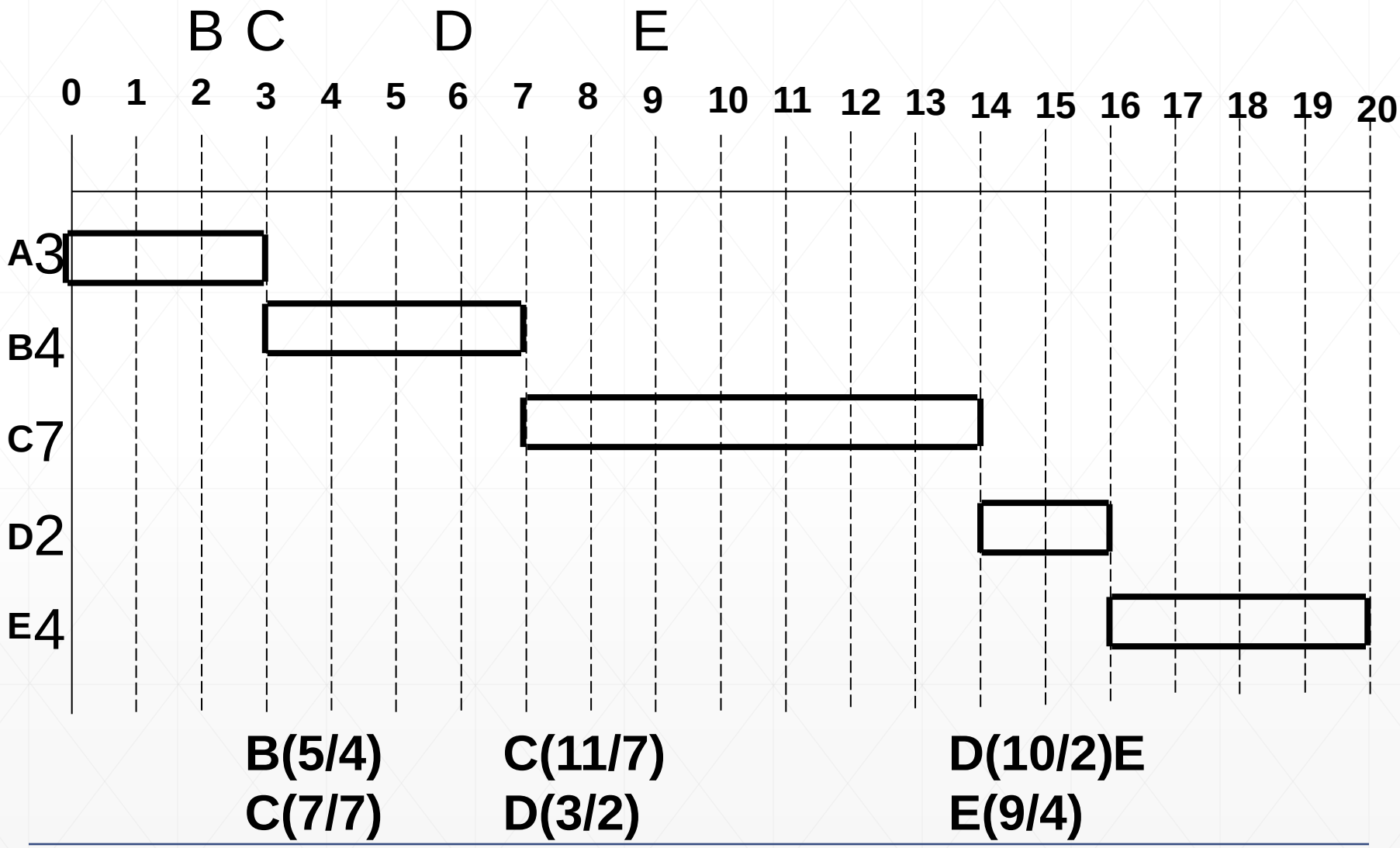

B C D E
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20
A 3
B 4
C 7
D 2
E 4
B C
C D
D E
E

94

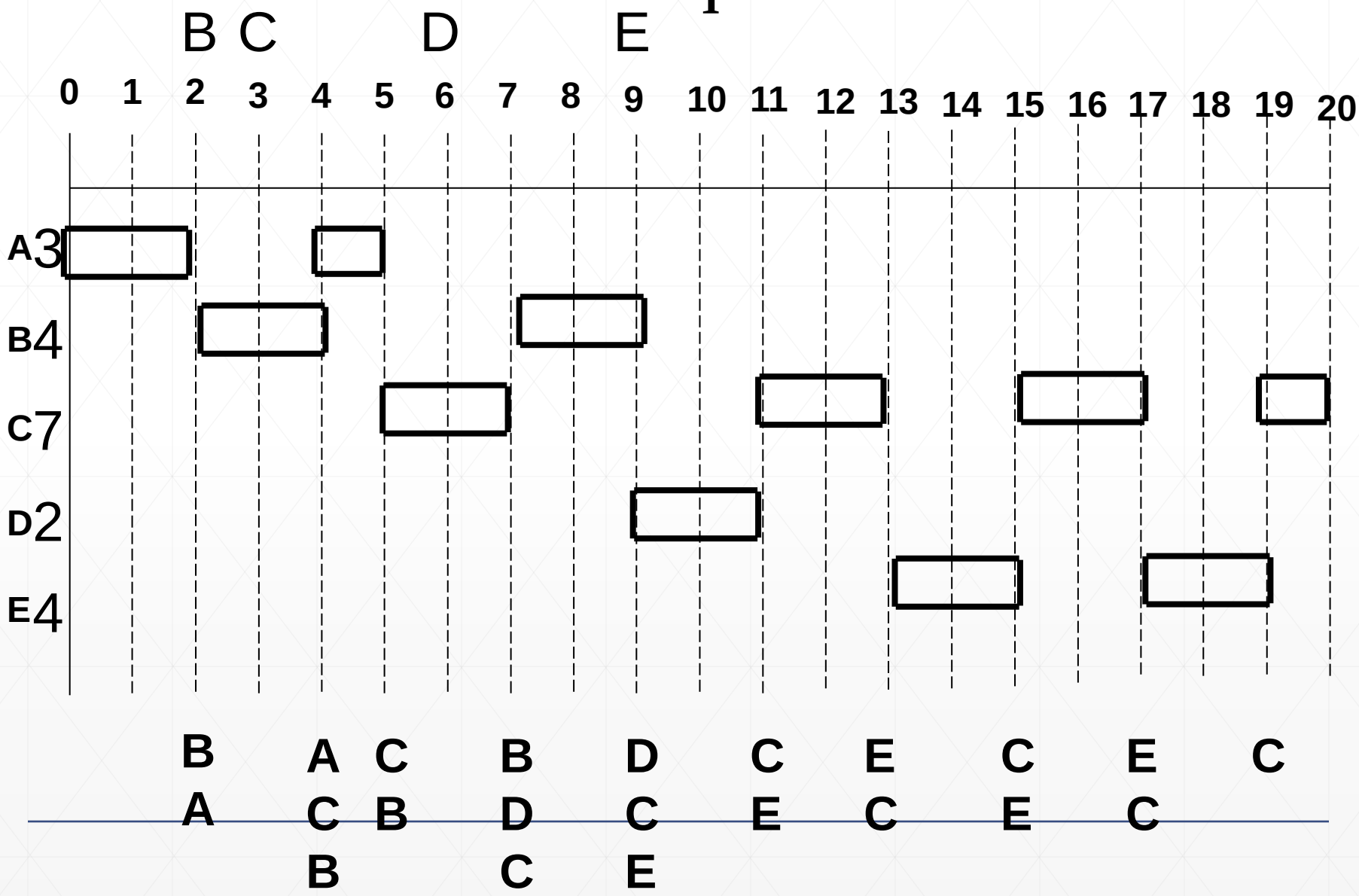
SPF


B C D E
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20
A 3
B 4
C 7
D 2
E 4
B C
C D
C E
C

HRRN


B C D E
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20
A 3
B 4
C 7
D 2
E 4
B(5/4)
C(7/7)
C(11/7)
D(3/2)
D(10/2)E
E(9/4)

RR（q=2）


B C D E
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20
A 3
B 4
C 7
D 2
E 4
B A A C B C B C B D C B D C E C E C E C

- 练习题3

- 表中列出了五个进程的执行时间
- 在某一时刻这五个进程按照P0，P1，P2，P3，P4的顺序同时到达，
- 现分别采用FCFS（先来先服务）、SPF（短进程优先）算法对进程进行调度

进程名	执行时间（ms）
P0	20
P1	15
P2	35
P3	25
P4	40

请在表格中填写各进程的周转时间和带权周转时间，并计算进程的平均周转时间和平均带权周转时间。

- FCFS（先来先服务）

进程执行顺序	执行时间（ms）	周转时间（ms）	带权周转时间
P_0	20	20	1
P_1	15	35	2.3
P_2	35	70	2
P_3	25	95	3.8
P_4	40	135	3.375

- 进程的平均周转时间： 71
- 进程的平均带权周转时间：2.495

- SPF（短进程优先）

进程执行顺序	执行时间（ms）	周转时间（ms）	带权周转时间
P_1	15	15	1
P_0	20	35	1.75
P_3	25	60	2.4
P_2	35	95	2.71
P_4	40	135	3.375

- 进程的平均周转时间： 68
- 进程的平均带权周转时间：2.247

3.3.6 基于公平原则的调度算法

1 保证调度算法

- 保证的是绝对运行时间，即启动后在某个时间段内必须获得多少运行时间。
- 例如N个进程平均分配时间。

2 公平分享调度算法

- 按照用户数量平均分配时间，而不是进程间平均分配。

3.3.6 基于公平原则的调度算法

例：公平分享调度算法

假设：用户1有4个进程 ABCD

用户2有1个进程 E

对用户2不公平

(1) 时间片轮转法

ABCDEABCDEABCDEABCDE……

(2) 所有用户获得相同的处理机时间

AEBECEDEAEBECEDEAEBECEDE……

(3) 用户1获得的处理机时间是用户2的两倍

ABECDEABECDEABECDEABECDE……

3.4 实时调度

- **实时任务：**
  - **任务的结束时间有严格约束(Deadline)，即任务执行必须在Deadline之前完成。**
  - **具有紧迫性。**
  - **前述算法不能很好地满足实时系统对调度的特殊要求，所以引入实时调度。**

3.4 实时调度

- **实时操作系统RTOS**
  - Real-Time Operating System
  - 对外部输入的信息，实时操作系统能够在规定的时间内处理完毕并做出反应
  - **正确性**：不仅依靠计算逻辑的正确，而且要求在规定的时间内得到该结果
  - 通常给定一个开始时间或者结束时间的最后期限
  - 多用于工业、军事等控制领域或实时信息处理方面

实时操作系统RTOS

- **硬实时系统**有一个刚性的、不可改变的时间限制，它不允许任何超出时限的错误。超时错误会带来损害甚至导致系统失败、或者导致系统不能实现它的预期目标。
- **软实时系统**的时限是柔性灵活的，它可以容忍偶然的超时错误。失败后造成的后果并不严重，例如在网络中仅仅轻微地降低了系统的吞吐量。
- **硬实时HRT与软实时SRT之间最关键的差别在于：**软实时只能提供统计意义上的实时。例如，有的应用要求系统在95%的情况下都会确保在规定的时间内完成某个动作，而不一定要求100%

实时操作系统RTOS

- 嵌入式操作系统的实时性都比较强，可归为RTOS
  - VxWorks操作系统
    - 美国WindRiver公司于1983年设计开发，实时性强，内核可极微（据说最小可8K），可靠性较高等，主要在通信设备等实时性要求较高的系统中。
  - uCOS
    - uCOS是一种免费公开源代码、结构小巧、具有可剥夺实时内核的实时操作系统。
  - 嵌入式Linux操作系统
    - Linux本身嵌入式操作系统，嵌入式领域使用的是专为嵌入式设计的已被裁减过的Linux系统，如uClinux
  - Windows CE
    - 主要用于PDA、手机、显示仪表等界面要求较高或者要求快速开发的场合
  - 其他RTOS：Symbian、Palm OS等

3.4.1 实现实时调度的基本条件

1. 提供必要的调度信息

(1) 就绪时间。（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。

(3) 处理时间。（任务开始执行到完成所需时间）

(4) 资源要求。

(5) 优先级。 若错过开始截止时间则赋予“绝对”优先级。

3.4 实时调度

2. 系统处理能力强

- 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：

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

系统才是可调度的。

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50ms，而每次的处理时间为10ms，不难算出，此时系统是不可调度的。

解决方法是提高系统的处理能力。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，显著地减少每个任务的处理时间；其二是采用多处理机系统。假定系统中处理机数为N，则应将限制条件改为：

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

3.4.1 实现实时调度的基本条件

在含有硬实时任务的实时系统中，广泛采用抢占机制。

3. 采用抢占式调度机制

调度程序先调度开始截止时间即将到达的任务。

4. 具有快速切换机制

4.1 具有快速响应外部中断的能力

及时响应紧迫的外部事件的中断请求，要求快速的硬件中断机构，尽量短的禁止中断的时间间隔。

4.2 快速的任务分派能力

应使系统中的每个运行功能单位适当的小，以减少任务切换的时间开销。

3.4.2 实时调度算法的分类

1. 非抢占式调度算法

(1)非抢占式轮转调度算法（如工业生产群控系统）

- 调度程序每次选择队列中的第一个任务投入运行。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。
- 常用于要求不太严格的实时控制系统。

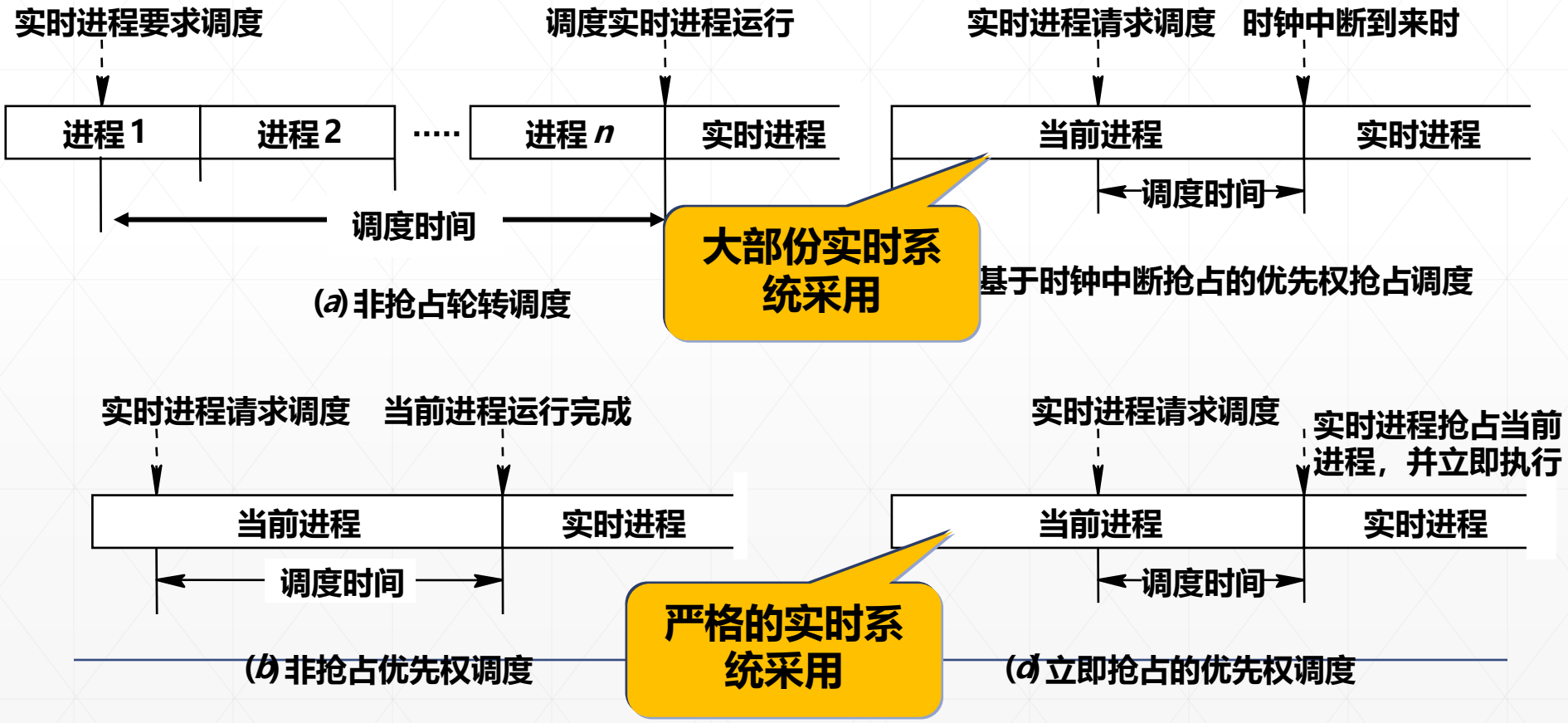
(2)非抢占优先权调度算法

- 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。
- 常用于有一定要求的实时控制系统。

2. 抢占式调度算法

(1) 基于时钟中断的抢占式优先权调度算法

(2) 立即抢占(Immediate Preemption)的优先权调度算法



3.4.3 常用的几种实时调度算法

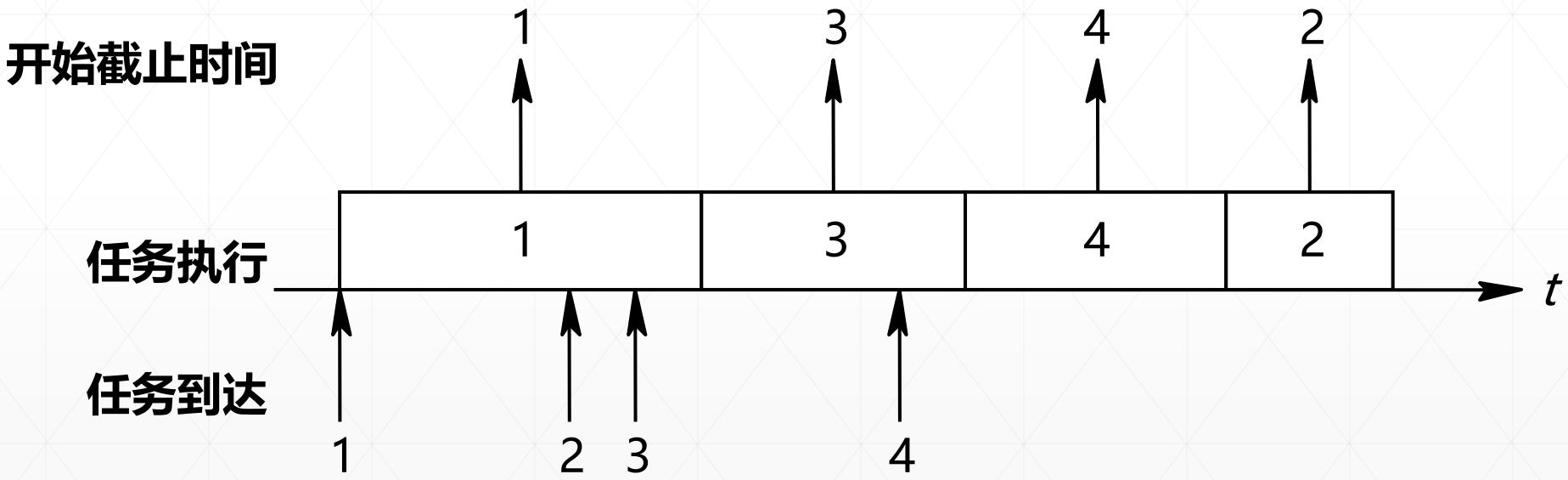
1. 最早截止时间优先即EDF(Earliest Deadline First) 算法

- **优先级确定**：根据任务的开始截止时间来确定任务的优先级。截止时间愈早，其优先级愈高。
- **实时任务就绪队列**：按各任务截止时间的早晚排序；具有最早截止时间的任务排在队列的最前面。
- **调度顺序**：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。
- **适用范围**：既可用于抢占式调度，也可用于非抢占式调度方式中。

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

(1) 非抢占式调度方式用于非周期实时任务

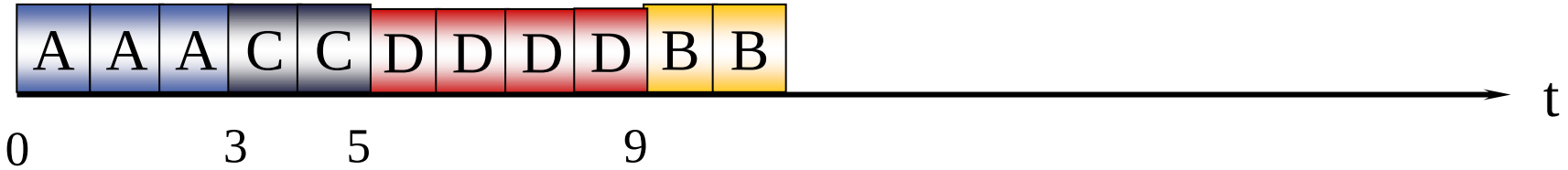
例：四个非周期任务，它们先后到达



EDF算法用于非抢占调度方式

非抢占式的EDF

任务名	到达时间	服务时间	开始截止时间	开始时间	完成时间	截止与完成差	是否能保证
A	0	3	1	0	3	C>A	是
B	2	1	10	9	10		
C	2	2	4	3	5	D>C	是
D	3	4	6	5	9	B>D	是



例：有两个周期性任务，任务A的周期时间为20 ms，每个周期的处理时间为10 ms；任务B 的周期时间为50 ms，每个周期的处理时间为25 ms。

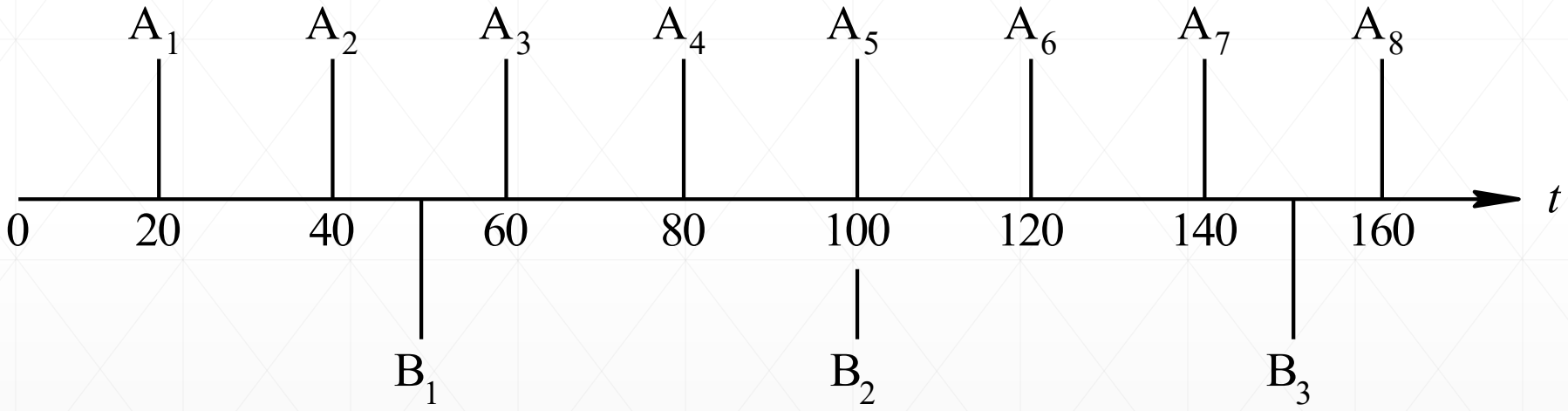
2. 最低松弛度优先即LLF(Least Laxity First)算法

- 松弛度=完成截止时间–剩余运行时间–当前时间

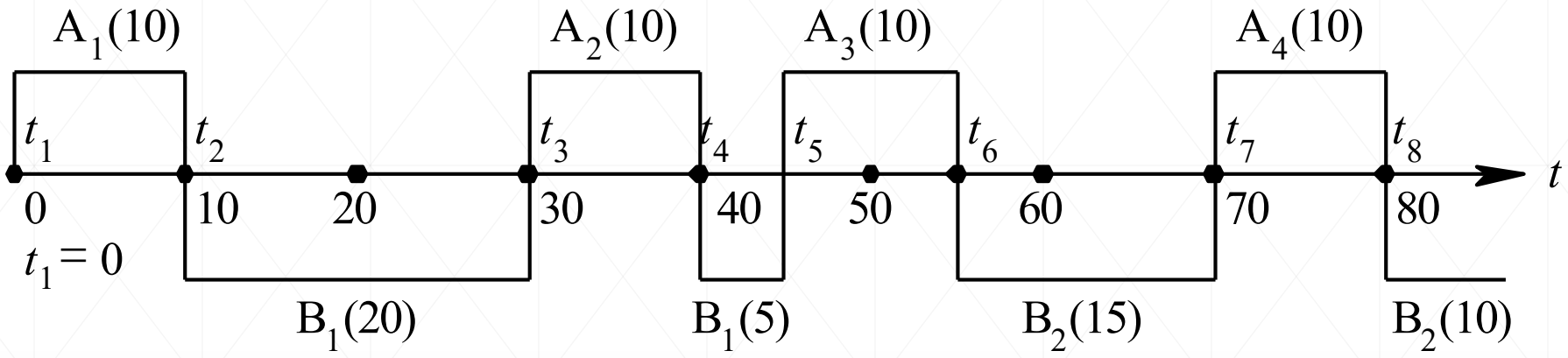
- 该算法按松弛度排序实时任务的就绪队列，松弛度值最小的任务排在队列最前面，调度程序总是选择就绪队列中的队首任务执行。

- 该算法主要用于可抢占调度方式中。

假如在一个实时系统中，有两个周期性实时任务A和B，任务A要求每20ms执行一次，执行时间为 10ms；任务B只要求每50ms执行一次，执行时间为 25ms。



A和B任务每次必须完成的时间



利用LLF算法进行调度的情况

t=0，A_1必须在20ms时完成，而它本身运行又需10ms，可算出A_1的松弛度为10ms；B_1必须在50ms时完成，而它本身运行就需25ms，可算出B_1的松弛度为25 ms，故调度程序应先调度A_1执行。

t=10ms，任务A尚未进入第2周期，故调度程序应选择B_1运行。

t=20ms，A_2的松弛度为10(即40-10-20)，而B_1的松弛度为15ms(即50-15-20)，虽然A_2的松弛度小于B_1但未到0，所以A_2不抢占B_1的处理机。

t=30ms，A_2的松弛度已减为0(即40-10-30)，而B_1的松弛度为15ms(即50-5-30)，于是调度程序应抢占B_1的处理机而调度A_2运行。

t=40ms，A_3的松弛度为10ms(即60-10-40)，而B_1的松弛度仅为5ms(即50-5-40)，故又应重新调度B_1执行。

t=45ms，B_1执行完成，而任务B未进入第2周期，于是应调度A_3执行。

t=50ms，A_3的松弛度为5ms(即60-5-50)，B_2的松弛度为25ms(即100-25-50)，于是A_3继续执行。

t=55ms，A3执行完成，任务A尚未进入第4周期，而任务B已进入第2周期，故调度B2执行。

t=60ms，A_4的松弛度为10ms(即80-10-60)，B_2的松弛度为20ms(即100-20-60)，虽然A_4的松弛度小于B_2但未到0，所以A_4不抢占B_2的处理机。于是B_2继续执行。

t7=70ms，A4的松弛度已减至0ms(即80-10-70)，而B2的松弛度为20ms(即100-10-70)，故此时调度应抢占B2的处理机而调度A4执行。

抢占方式和时机

- **当等待任务的松弛度值为0时才进行抢占**（如20ms时虽然A2的松弛度比B1的松弛度小，但A2并没有抢占B1）。
  - 当有任务执行时，只有等待任务的松弛度值为0才会发生任务的调度，其他情况不发生调度。
  - 任务执行结束后或无任务执行时，再比较等待任务的松弛度值，较小的先执行。

3.4.4 优先级倒置(priority inversion problem)

- **1. 优先级倒置的形成**

  “优先级倒置”的现象，即高优先级进程(或线程)被低优先级进程(或线程)延迟或阻塞。

- **例如：优先级：P1 》P2 》P3**
  - **P1：…; P(mutex); CS-1; V(mutex);…**
  - **P2: ……program2……**
  - **P3:……;P(mutex); CS-3; V(mutex);……**

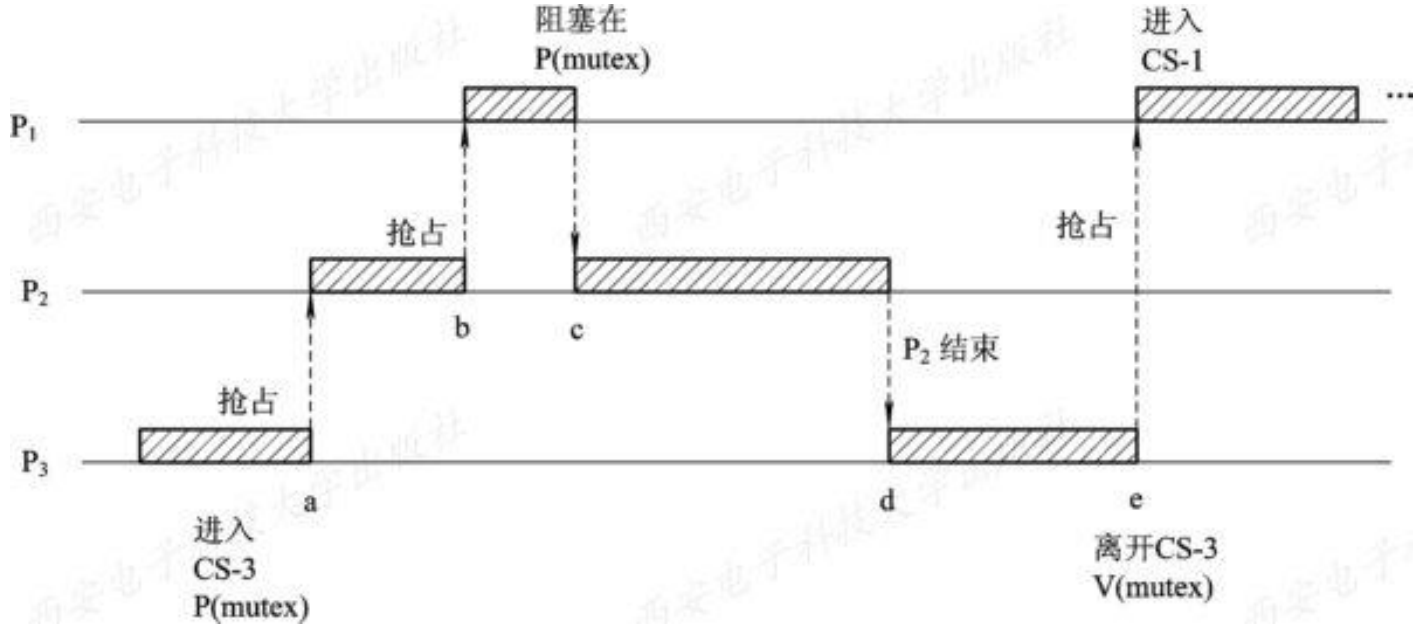


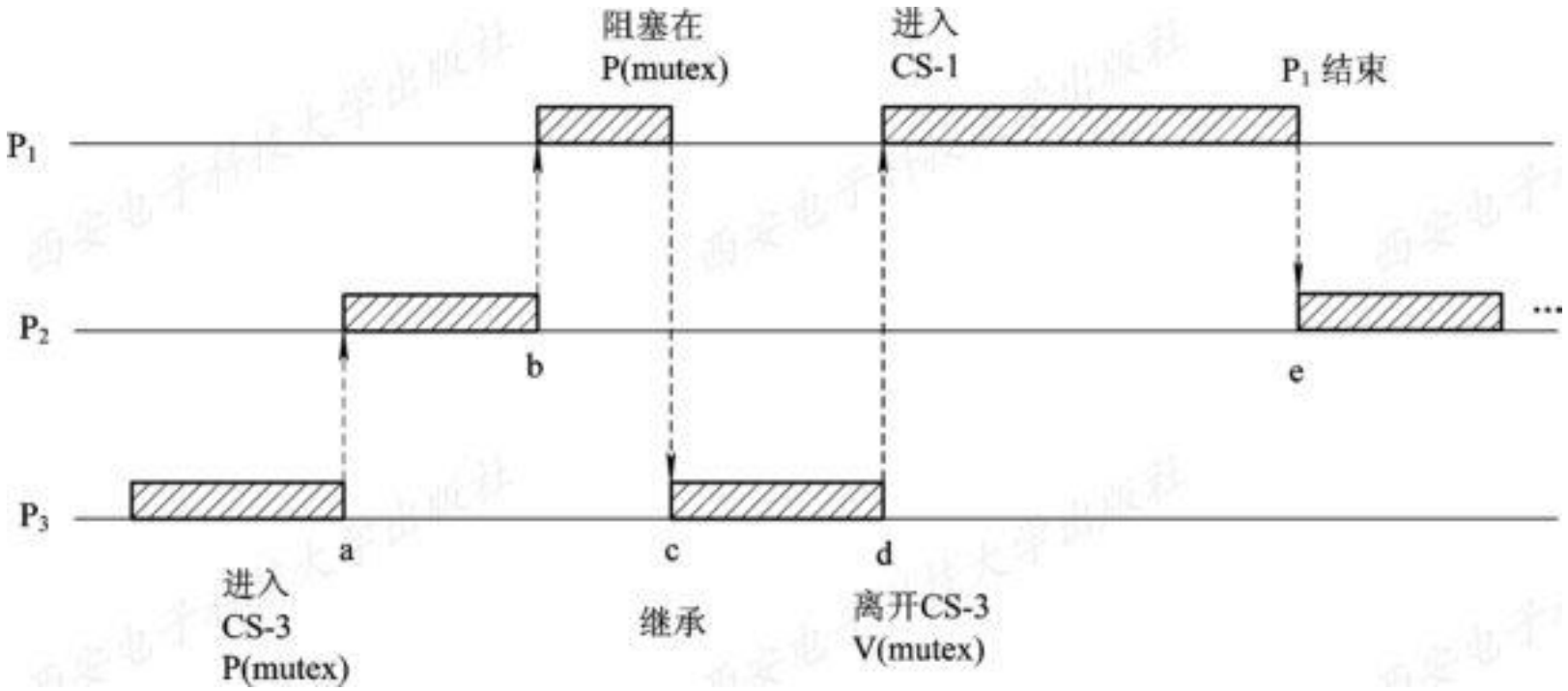
图3-10 优先级倒置示意图

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

2. 优先级倒置的解决方法

- 规定：假如进程P3在进入临界区后P3所占用的处理机就不允许被抢占
  - 仅适用于临界区较短情况
- 采用动态优先级继承方法
  - 防范中间优先级进程插入

动态优先级继承法解决优先级倒置



采用了动态优先级继承方法的运行情况

引例

- 设系统中只有一台打印机和一台读卡机，它们被进程P和进程Q共用。
- 进程P和Q各自对资源的申请使用情况如下：

P：申请读卡机	Q：申请打印机
申请打印机	申请读卡机
…	…
释放读卡机	释放打印机
释放打印机	释放读卡机

考虑下面的执行序列

P： 申请读卡机

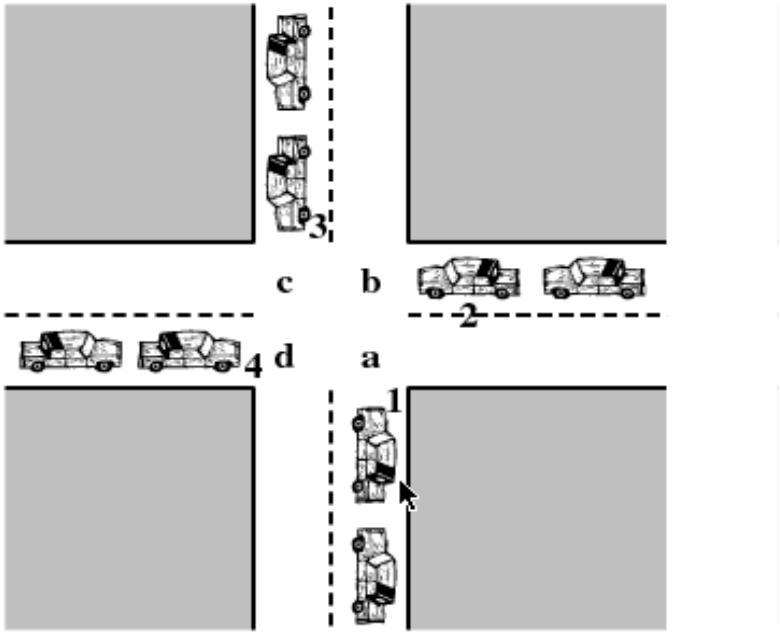
Q：申请打印机

P： 申请打印机

Q：申请读卡机

- 结果：进程P占有读卡机，..但未申请到打印机，需要等待；进程Q占有打印机，但等待读卡机。最终，P和Q都无法运行下去，彼此等待对方释放自己所需的资源。

交通实例



(a) Deadlock possible

独木桥实例

- 过一条独木桥，过桥人都只能向前进并不后退，那么当两个人从桥两端同时上桥，就会在桥中间相遇。这样两个人就只能在桥上相对而立，谁也过不去。

系统资源

主存、CPU

- 永久（可重用）性资源
  - 可抢占性资源: 是指某进程在获得这类资源后，该资源可以再被其他进程或系统剥夺。
  - 不可抢占性资源: 当系统把这类资源分配给某进程后，就不能强行收回，只能在进程用完后自行释放。
- 临时性（消耗性）资源
  - 只可使用一次的资源

磁带机、打印机

3.5 产生死锁的原因和必要条件

死锁：指多个进程因竞争资源或相互通信而造成的一种僵局，都在等待着对方释放出自己所需的资源，但同时又不释放出自己已经占有的资源，若无外力作用，这些进程都将永远无法向前推进。

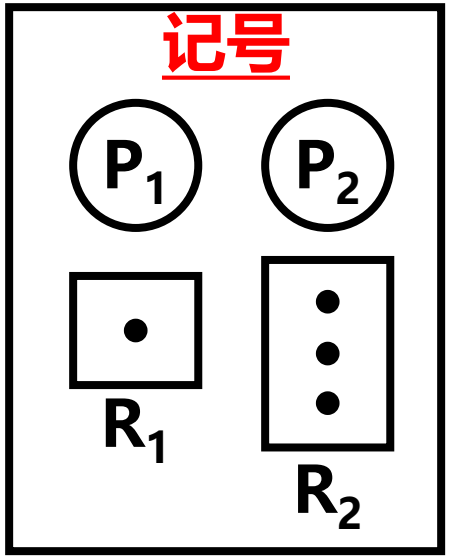
3.5.1 产生死锁的原因

1.竞争资源。

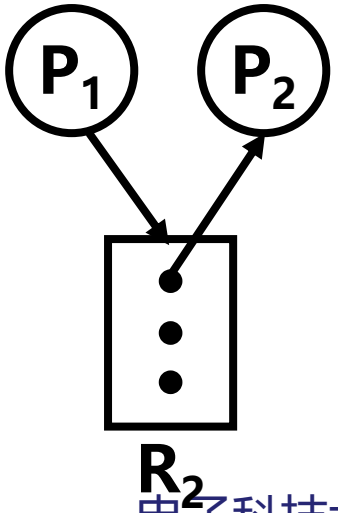
2. 进程间推进顺序不当。

资源分配图

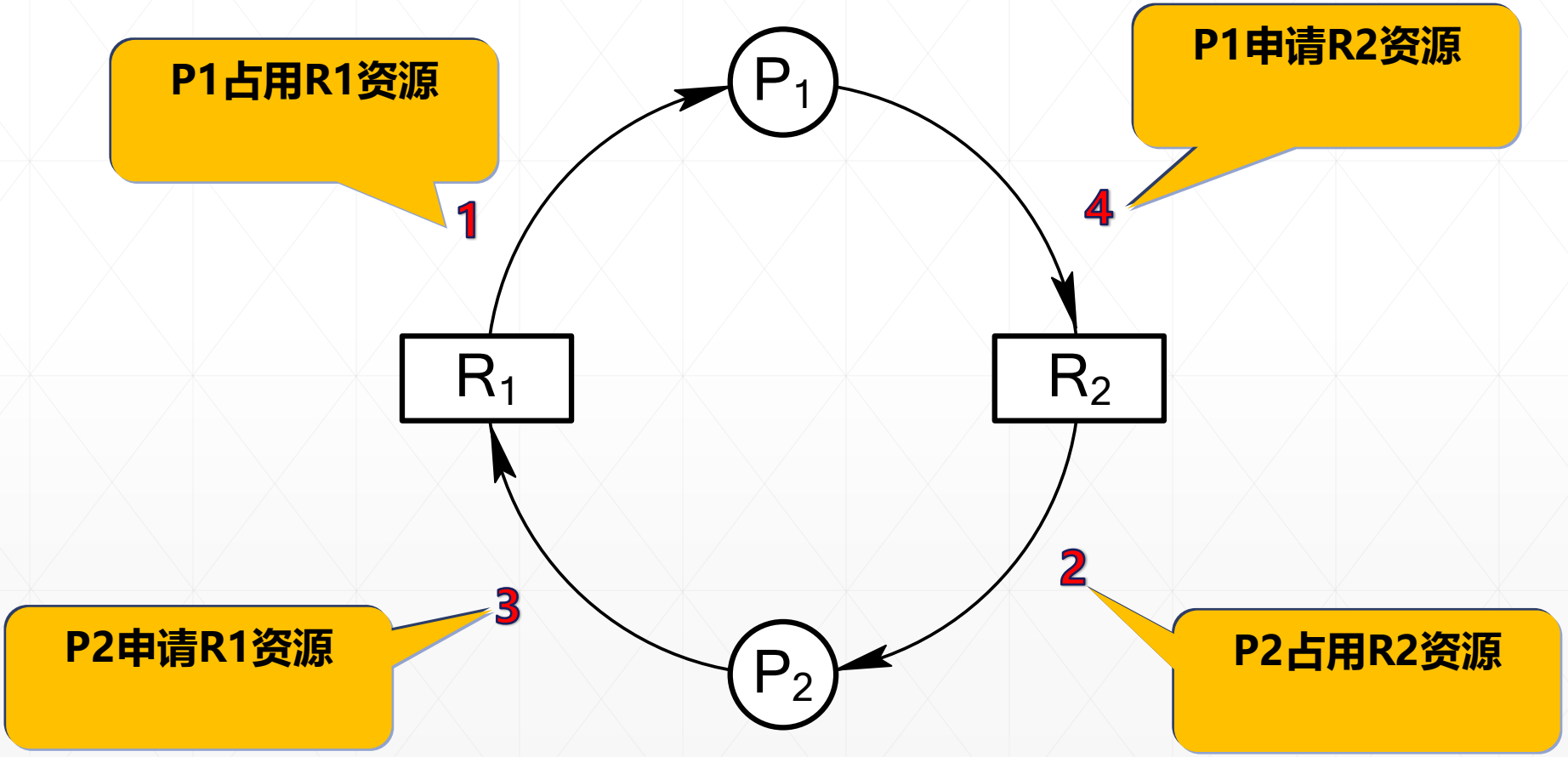
- **资源分配图模型**
  - 一个进程集合{P_1,P_2,…,P_n}
  - 一资源类型集合{R_1,R_2,…,R_m}
  - 资源类型R_i有W_i个实例



  - 资源请求边：有向边$P_i \rightarrow R_j$
  - 资源分配边：有向边$R_i \rightarrow P_k$

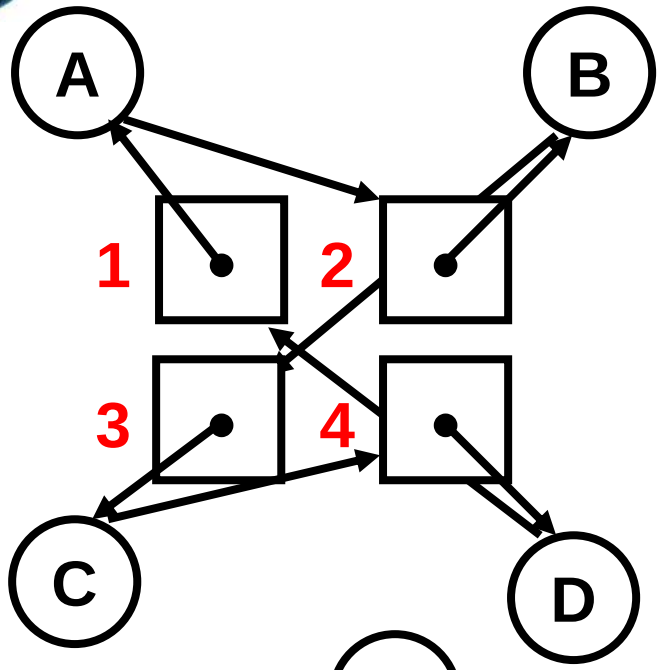


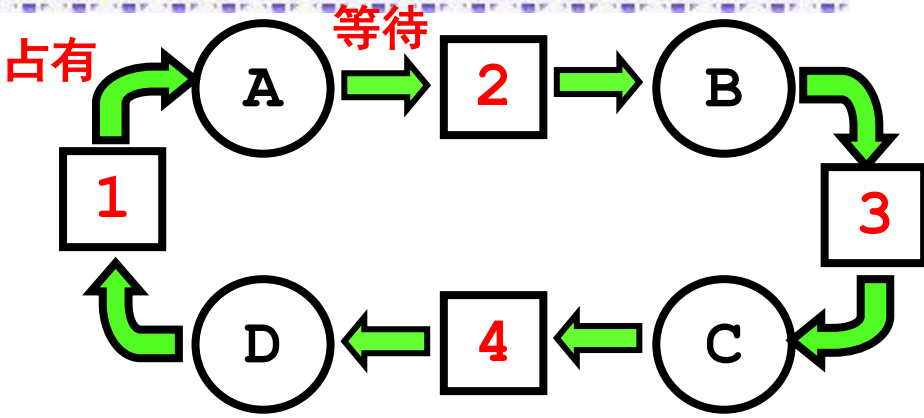
1. 竞争不可抢占性资源引起死锁



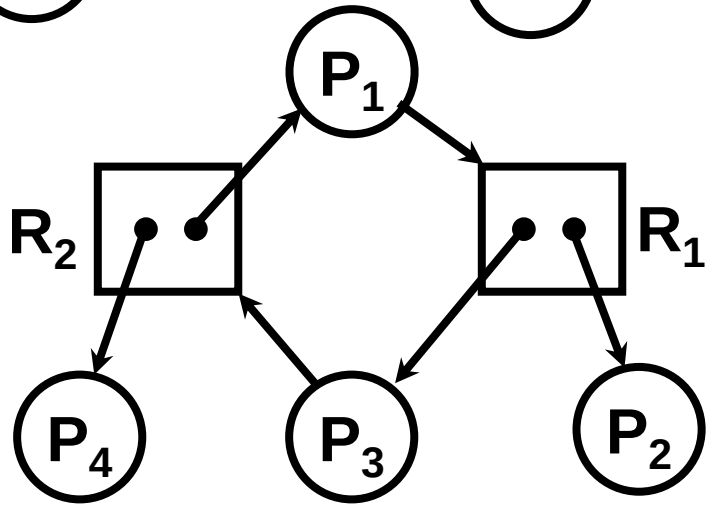
I/O设备共享时的死锁情况

资源分配图实例





- 存在环路: 1-A-2-B-3-C-4-D-1

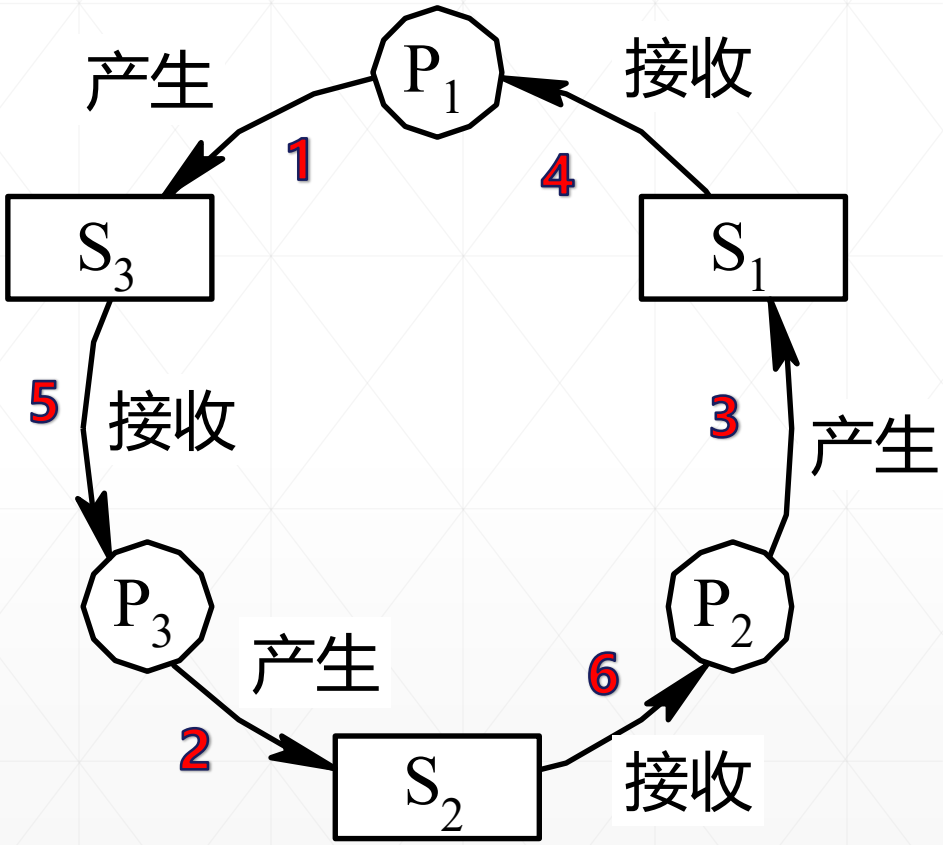


- 存在环路: P_1-R_1-P_3-R_2-P_1
- 但并不死锁，仍可继续执行

2. 竞争临时性(消耗性)资源引起进行死锁

•临时性资源，可以创造（生产）和撤消（消耗）的资源，也称之为消耗性资源
•如信号量、消息、buffer中的数据等资源。

•例如：S1、S2和S3是临时性资源，是由进程P1、P2和P3产生的消息。如果消息通信处理顺序不当也会发生死锁。



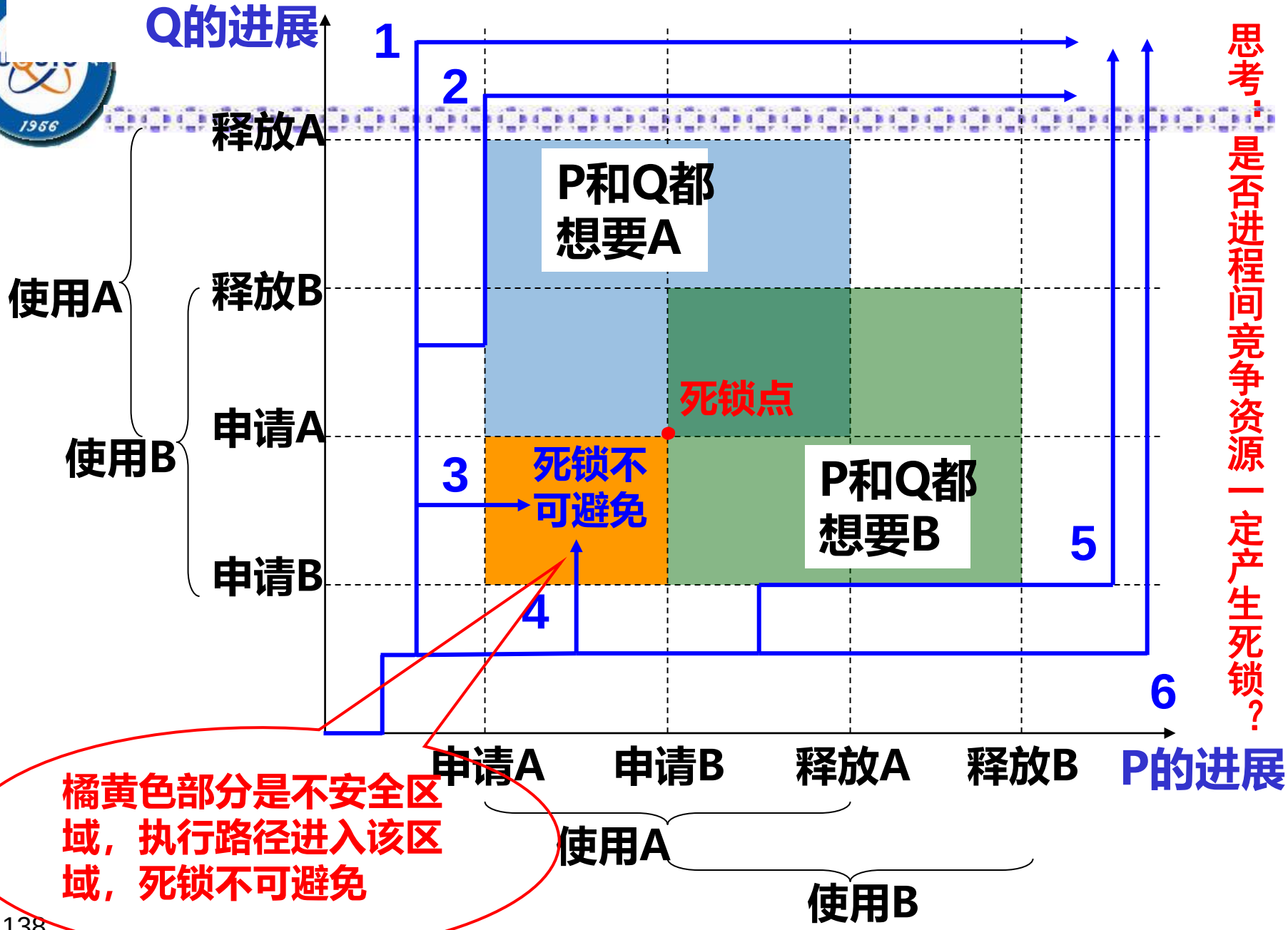
进程之间通信时的死锁

3. 进程推进顺序不当引起死锁

- 联合进程图（Joint Progress Diagram）记录共享资源的多个进程的执行进展。

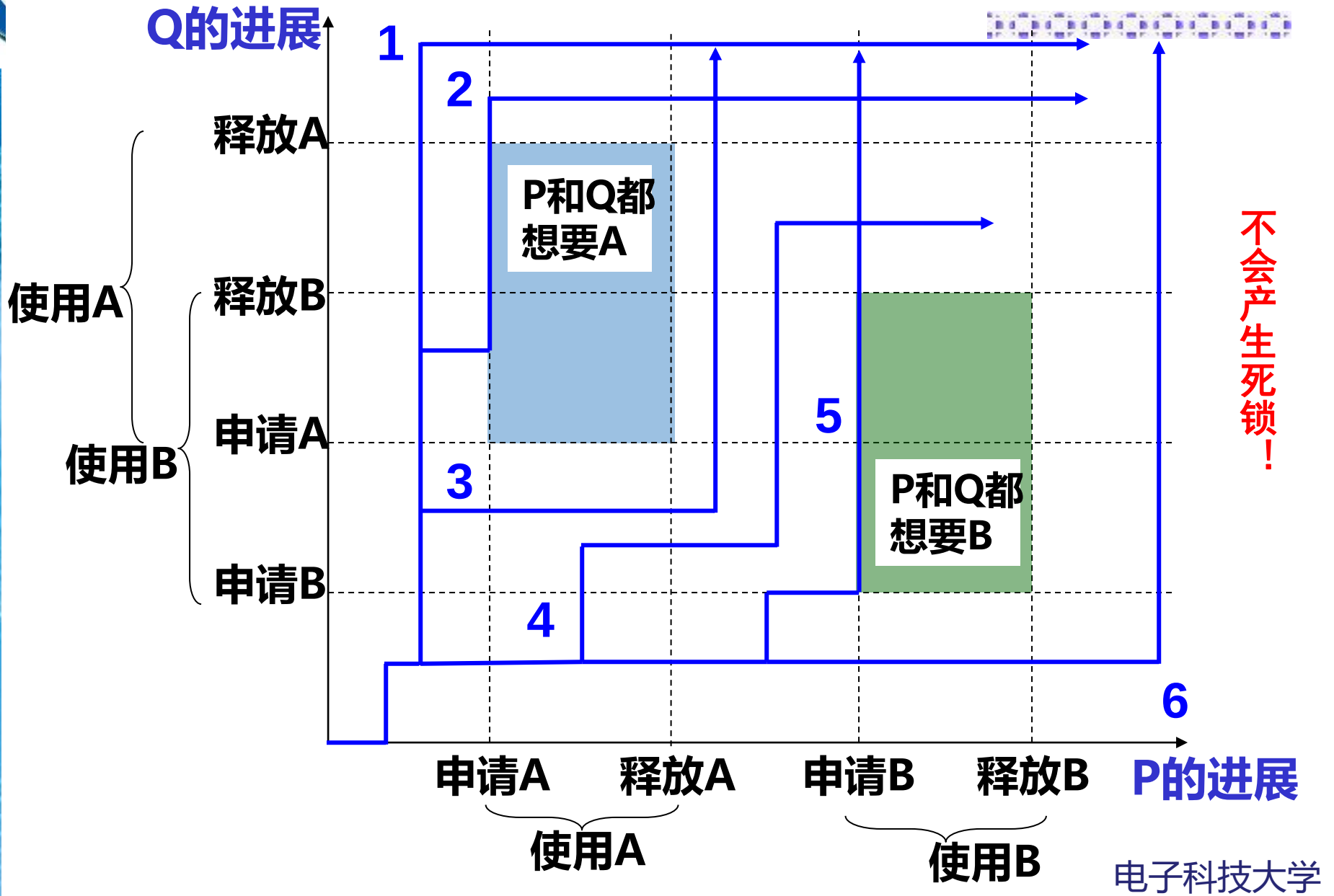
进程P	进程Q
…	…
申请A（读卡机）	申请B（打印机）
…	…
申请B	申请A
…	…
释放A	释放B
…	…
释放B	释放A
…	…

特点：一个进程互斥使用的资源不只一个。每个进程都需要独占两个资源一段时间。


Q的进展
1
2
释放A
P和Q都
想要A
使用A
释放B
死锁点
申请A
使用B
3
死锁不
可避免
P和Q都
想要B
5
申请B
4
6
思考：是否进程间竞争资源一定产生死锁？
申请A
申请B
释放A
释放B
P的进展
使用A
使用B
橘黄色部分是不安全区域，执行路径进入该区域，死锁不可避免

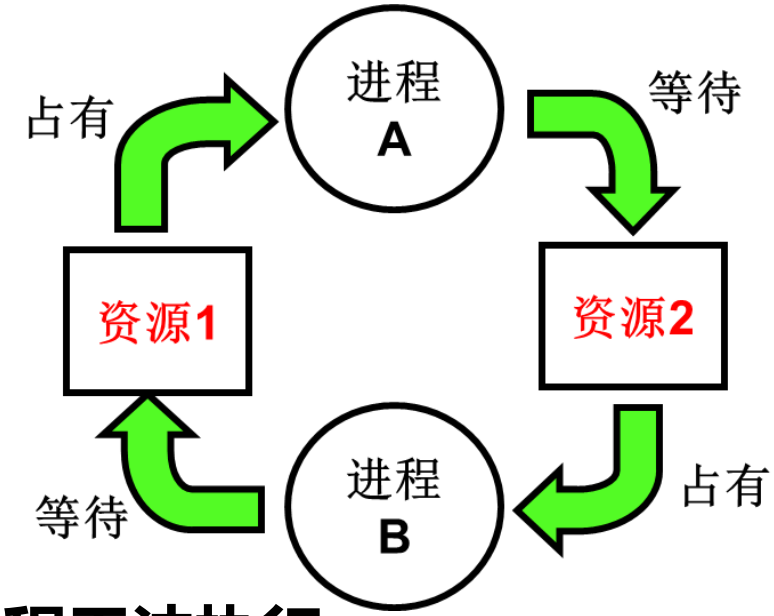
进程P	进程Q
…	…
申请A	申请B
…	…
释放A	申请A
…	…
申请B	释放B
…	…
释放B	释放A
…	…

- 竞争资源，未必产生死锁。
- 是否产生死锁，还取决于动态执行和应用程序细节。


Q的进展
1
2
释放A
P和Q都
想要A
使用A
释放B
使用B
申请A
3
5
P和Q都
想要B
申请B
4
不会产生死锁！
6
申请A
释放A
申请B
释放B
P的进展
使用A
使用B

死锁（Deadlock）的定义

- 如果一组进程中的每一个进程都在等待仅由该组进程中的其它进程才能引发的事件，那么该组进程是死锁的。



- 死锁会造成进程无法执行
- 死锁会造成系统资源的极大浪费(资源没法释放)

3.5.2 产生死锁的必要条件

(1)互斥条件： 进程对分配到的资源进行排它性使用。

(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要求，而该资源又被其他进程占有，请求进程阻塞，但对已经获得的资源不释放。

(3) 不剥夺条件： 进程已获得的资源，使用完之前不能被剥夺，只能用完自己释放。

(4) 环路等待条件： 发生死锁时，必然存在进程—资源的环形链。

有环路不一定死锁！！

3.5.3 处理死锁的基本方法

1.预防死锁： 设置某些限制条件，破坏四个必要条件中的一个或几个。 “no smoking”，预防火灾

优点：容易实现。缺点:系统资源利用率和吞吐量降低。

2.避免死锁： 在资源的动态分配过程用某种方法防止系统进入不安全状态。 检测到煤气超标时，自动切断电源

优点：较弱限制条件可获得较高系统资源利用率和吞吐量。

缺点：有一定实现难度。

3.检测死锁： 预先不采取任何限制，也不检查系统是否已进入不安全区，通过设置检测机构，检测出死锁后解除。

4.解除死锁： 常用撤消或挂起一些进程，回收一些资源。

检测+解除：发现火灾时，立刻拿起灭火器

课堂思考

1. 两个进程争夺同一个资源（　　）。

（A）一定死锁　　（B）不一定死锁

（C）不死锁　　（D）以上说法都不对

2.如果发现系统有（　　）的进程队列就说明系统有可能发生死锁了。

（A）互斥　　（B）可剥夺

（C）循环等待　　（D）同步

3.预先静态分配法是通过破坏（　　）条件，来达到预防死锁目的的。

（A）互斥使用资源/循环等待资源

（B）非抢占式分配/互斥使用资源

（C）请求且保持资源/循环等待资源

（D）循环等待资源/互斥使用资源

B,C,C

课堂思考

- **学生想到了下面这个消除死锁的方法。当某一进程请求一个资源时，规定一个时间限。如果进程由于得不到需要的资源而阻塞，定时器开始运行。当超过时间限时，进程会被释放掉，并且允许该进程重新运行。如果你是老师，你会给这样的学生多少分？为什么？**

- 我会给它一个F（失败）的成绩。进程会做什么？很显然由于它需要资源，所以它再次询问并再次阻塞，这不如保持阻塞。事实上，可能会更糟糕的是，系统可能会跟踪竞争进程等待了多长时间，并将新释放的资源分配给等待时间最长的进程。通过定期释放和重新运行，一个进程丢失了它已经等待的时间的记录。

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属性所决定的，不仅不能改变，还应加以保证。

3.6 预防死锁的方法

1. 摒弃“请求和保持”条件

第一种协议：系统要求所有进程一次性申请所需的全部资源，只要有一种资源要求不能满足，即使是已有的其它各资源，也全部不分配给该进程，而让其等待。

优点：简单、易于实现且很安全。

缺点：资源严重浪费；进程延迟运行。

1．摒弃“请求和保持”条件

第二种协议：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、且已使用完毕的全部资源，然后再请求新的所需资源 。

优点：使进程更快地完成任务，提高设备的利用率，减少进程发生饥饿的概率。

2. 摒弃“不剥夺”条件

进程在需要资源时才提出请求，一个已经保持了某些资源的进程，再提出新的资源要求而不能立即得到满足时，必须释放已经保持的所有资源，待以后需要时再重新申请。

优点： 摒弃了“不剥夺”条件。

缺点： 实现复杂，代价大；延长了进程的周转时间，增加系统开销，降低系统吞吐量。

3. 摒弃“环路等待”条件

系统将所有资源按类型进行线性排队（常用→不常用），并赋予不同的序号。所有进程对资源的请求必须严格按资源序号递增的次序提出，按序号递减的次序释放。

假设有三类资源A(1), B(4), C(5)，进程的请求如下，分析是否可以完成。

（1） 请求A，请求C，请求B

（2） 请求A，请求C，释放C，请求B

不可以

可以

3.7 避免死锁

3.7.1 安全状态

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**。

- 并非所有不安全状态都是死锁状态，但当系统进入不安全状态后，便可能有进入死锁状态。
- 系统处于安全状态时，不会进入死锁状态。

安全状态：

假定系统中有三个进程P_1、P_2和P_3，共有12台打印机。进程P_1总共要求10台打印机，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台打印机，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	

T_0时刻系统是安全的，

存在一个安全序列<P2,P1,P3>

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。

由安全状态向不安全状态的转换：

如果不按照安全序列分配资源，则系统可能会由安全状态进入不安全状态。例如，在T_0时刻以后，P_3又请求1台磁带机，若此时系统把剩余3台中的1台分配给P_3，则系统便进入不安全状态。

进程	最大需求	已分配	可用
P_1	10	5	2
P_2	4	2	
P_3	9	3	

3.7.2 利用银行家算法避免死锁

- 避免死锁的关键在于如何准确的预测是否会出现死锁，从而避免死锁。
- 最有代表性的避免死锁的算法是Dijkstra的银行家算法。

银行家算法

问题描述：

- 该算法可用于银行发放一笔贷款前，预测该笔贷款是否会引起银行资金周转问题。
- 银行的资金就类似于计算机系统的资源，贷款业务类似于计算机的资源分配。银行家算法能预测一笔贷款业务对银行是否是安全的，该算法也能预测一次资源分配对计算机系统是否是安全的。
- 为实现银行家算法，系统中必须设置若干数据结构。

银行家算法思想——死锁避免策略

1.当前状态下，某进程申请资源；

2.系统**假设**将资源分给该进程，满足它的需求；

3.检查**分配后**的系统状态是否是安全的，如果是安全，就确认本次分配；如果系统是不安全的，就取消本次分配并阻塞该进程。（第三步又称安全算法）

避免死锁实质：在进程资源分配时，如何使系统不进入不安全状态。

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$n\times m$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) **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。

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$Request_i[j] \le Need[i,j]$，便转向步骤2；否则认为出错，因为它所申请的资源数已超过它所宣布的需要的资源数。

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

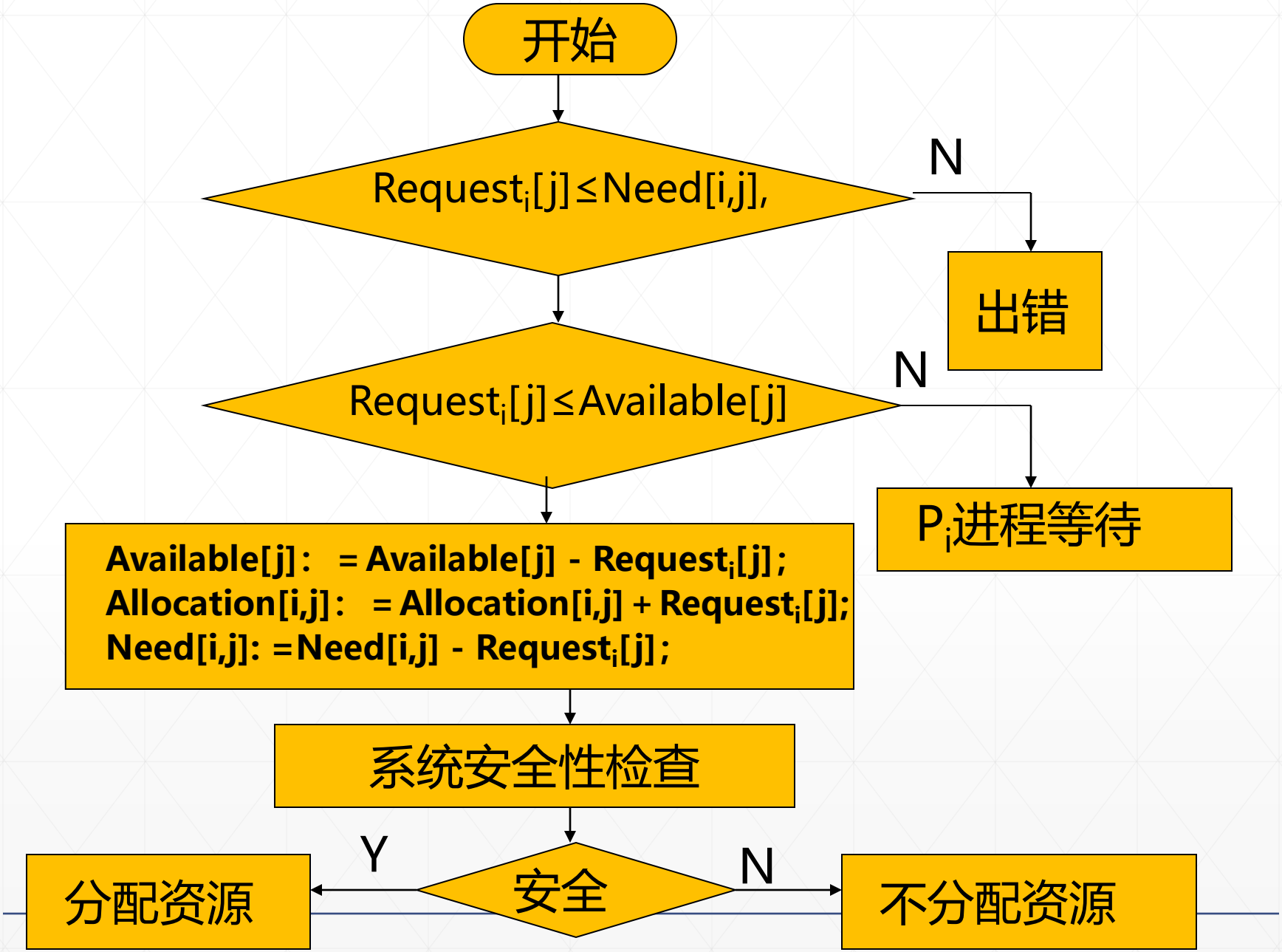
(3) 系统试探着把资源分配给进程P_i，并修改下面数据结构中的数值：

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。


开始
Request$_i$[j]≤Need[i,j],
N
出错
Request$_i$[j]≤Available[j]
N
P$_i$进程等待
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];
系统安全性检查
Y
安全
N
分配资源
不分配资源

3. 安全性算法

(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 Finish [i] =false;

② Need [i,j] ≤Work [j]； 若找到， 执行步骤(3)，否则，执行步骤(4)。

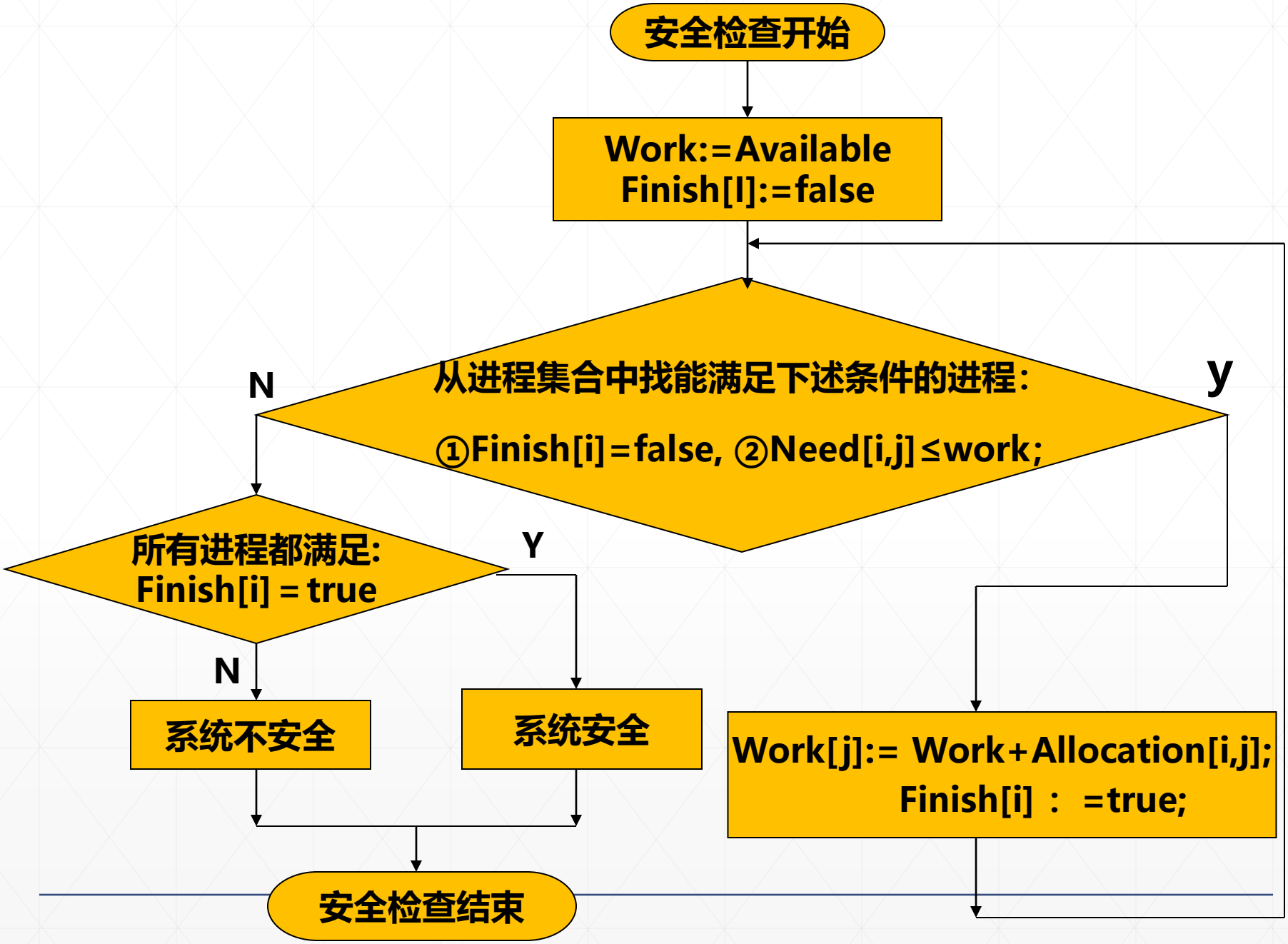
(3) 当进程P_i获得资源后，可顺利执行，直至完成，并释放出分配给它的资源，故应执行：

Work [j] =Work [j] +Allocation [i,j]；

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。


安全检查开始
Work:=Available
Finish[I]:=false
从进程集合中找能满足下述条件的进程:
①Finish[i]=false, ②Need[i,j]≤work;
N
y
所有进程都满足:
Finish[i] = true
Y
N
系统不安全
系统安全
Work[j]:= Work+Allocation[i,j];
Finish[i] : =true;
安全检查结束

4. 银行家算法举例

- 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下表所示。

	Max			Allocation			Need			Available		
	A	B	C	A	B	C	A	B	C	A	B	C
P0	7	5	3	0	1	0	7	4	3	3	3	2
P1	3	2	2	2	0	0	1	2	2			
P2	9	0	2	3	0	2	6	0	0			
P3	2	2	2	2	1	1	0	1	1			
P4	4	3	3	0	0	2	4	3	1			

- 问：

(1)P1提出资源请求Request（1，0，2）

(2)P4提出资源请求Request（3，3，0）

(3)P0提出资源请求Request（0，2，0）

按照避免死锁的要求，系统能否满足上述资源请求?

(1)判断 T_0 时刻的安全性

- 判断T_0时刻是否是安全状态?

	Max			Allocation			Need			Available		
	A	B	C	A	B	C	A	B	C	A	B	C
P0	7	5	3	0	1	0	7	4	3	3	3	2
P1	3	2	2	2	0	0	1	2	2			
P2	9	0	2	3	0	2	6	0	0			
P3	2	2	2	2	1	1	0	1	1			
P4	4	3	3	0	0	2	4	3	1			

- 利用安全性算法检测。

	Max			Allocation			Need			Available		
	A	B	C	A	B	C	A	B	C	A	B	C
P0	7	5	3	0	1	0	7	4	3	3	3	2
P1	3	2	2	2	0	0	1	2	2			
P2	9	0	2	3	0	2	6	0	0			
P3	2	2	2	2	1	1	0	1	1			
P4	4	3	3	0	0	2	4	3	1			

T0时刻是安全的

	Work			Need			Allocation			Work+ Allocation			Finish
	A	B	C	A	B	C	A	B	C	A	B	C	
P1	3	3	2	1	2	2	2	0	0	5	3	2	true
P3	5	3	2	0	1	1	2	1	1	7	4	3	true
P4	7	4	3	4	3	1	0	0	2	7	4	5	true
P2	7	4	5	6	0	0	3	0	2	10	4	7	true
P0	10	4	7	7	4	3	0	1	0	10	5	7	true

(2) P1提出请求Request（1，0，2）？

	Max			Allocation			Need			Available		
	A	B	C	A	B	C	A	B	C	A	B	C
P0	7	5	3	0	1	0	7	4	3	2	3	0
P1	3	2	2	3	0	2	0	2	0			
P2	9	0	2	3	0	2	6	0	0			
P3	2	2	2	2	1	1	0	1	1			
P4	4	3	3	0	0	2	4	3	1			

- ① $Request_1(1, 0, 2) \le Need_1(1, 2, 2)$
- ② $Request_1(1, 0, 2) \le Available(3, 3, 2)$
- ③ $Available_1 = (3, 3, 2) - (1, 0, 2) = (2, 3, 0)$

  $Need_1 = (1, 2, 2) - (1, 0, 2) = (0, 2, 0)$

  $Allocation_1 = (2, 0, 0) + (1, 0, 2) = (3, 0, 2)$

- **利用安全性算法检测状态是否安全。**

	Max			Allocation			Need			Available		
	A	B	C	A	B	C	A	B	C	A	B	C
P0	7	5	3	0	1	0	7	4	3	2	3	0
P1	3	2	2	3	0	2	0	2	0			
P2	9	0	2	3	0	2	6	0	0			
P3	2	2	2	2	1	1	0	1	1			
P4	4	3	3	0	0	2	4	3	1			

状态是安全的

	Work			Need			Allocation			Work+ Allocation			Finish
	A	B	C	A	B	C	A	B	C	A	B	C	
P1	2	3	0	0	2	0	3	0	2	5	3	2	true
P3	5	3	2	0	1	1	2	1	1	7	4	3	true
P4	7	4	3	4	3	1	0	0	2	7	4	5	true
P2	7	4	5	6	0	0	3	0	2	10	4	7	true
P0	10	4	7	7	4	3	0	1	0	10	5	7	true

- 由所进行的安全性检查得知，可以找到一个安全序列{P_1、P_3、P_4、P_2、P_0}，因此系统是安全的，所以可以将P_1所申请的资源分配给它。
- (3)P4提出资源请求Request（3，3，0）能满足吗?

(3) P4提出请求Request（3，3，0）？

	Max			Allocation			Need			Available		
	A	B	C	A	B	C	A	B	C	A	B	C
P0	7	5	3	0	1	0	7	4	3	2	3	0
P1	3	2	2	3	0	2	0	2	0			
P2	9	0	2	3	0	2	6	0	0			
P3	2	2	2	2	1	1	0	1	1			
P4	4	3	3	0	0	2	4	3	1			

- ① $Request_4(3, 3, 0) \leq Need_4(4, 3, 1)$
- ② $Request_4(3, 3, 0) \not\leq Available(2, 3, 0)$
- P4提出的资源请求不能满足，让P4等待。

(4)P_0提出资源请求Request（0，2，0），能满足吗?

(4) P0提出请求Request（0，2，0）？

	Max			Allocation			Need			Available		
	A	B	C	A	B	C	A	B	C	A	B	C
P0	7	5	3	0	3	0	7	2	3	2	1	0
P1	3	2	2	3	0	2	0	2	0			
P2	9	0	2	3	0	2	6	0	0			
P3	2	2	2	2	1	1	0	1	1			
P4	4	3	3	0	0	2	4	3	1			

- ① $Request_0(0, 2, 0) \le Need_0(7, 4, 3)$
- ② $Request_0(0, 2, 0) \le Available(2, 3, 0)$
- ③ $Available = (2, 3, 0) - (0, 2, 0) = (2, 1, 0)$

$Need_0 = (7, 4, 3) - (0, 2, 0) = (7, 2, 3)$

$Allocation_0 = (0, 1, 0) + (0, 2, 0) = (0, 3, 0)$

- **利用安全性算法检测状态是否安全。**

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

总结

- 预防死锁
- 避免死锁

事先施加限制条件，以防死锁发生。

- **两者的区别：**
  - **预防死锁：** 破坏死锁的必要条件，施加的条件比较严格，可能会影响到进程的并发执行。
  - **避免死锁：** 资源动态分配，施加的限制条件较弱一些，有利于进程的并发执行。

总结

- 死锁避免策略并不能确切的预测死锁，仅仅是预料死锁的可能性并确保永远不会出现这种可能性。
- 死锁避免比死锁预防限制少，但使用中也有许多限制：
  - 必须事先声明每个进程请求的最大资源。
  - 考虑的进程必须是无关的，执行的顺序必须没有任何同步的要求。
  - 分配的资源数目必须是固定的。
  - 在占有资源时，进程不能退出。

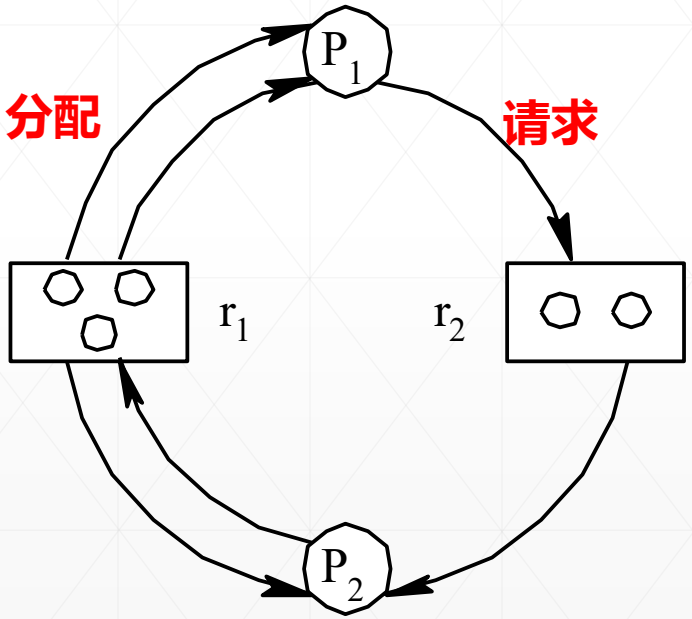
3.8 死锁的检测与解除

- 如果系统不愿意附加太多约束条件预防死锁，也不希望系统额外开销预测并避免死锁，那么，只能允许死锁出现，然后，再解除它。
- 因此，系统需要利用某种方法来检测死锁。

3.8 死锁的检测与解除

3.8.1 死锁的检测

1. 资源分配图(Resource Allocation Graph)



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），其中：

（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 = PUR。

（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}

它表示进程P_i请求一个单位的R_j资源。

e={R_j，P_i}

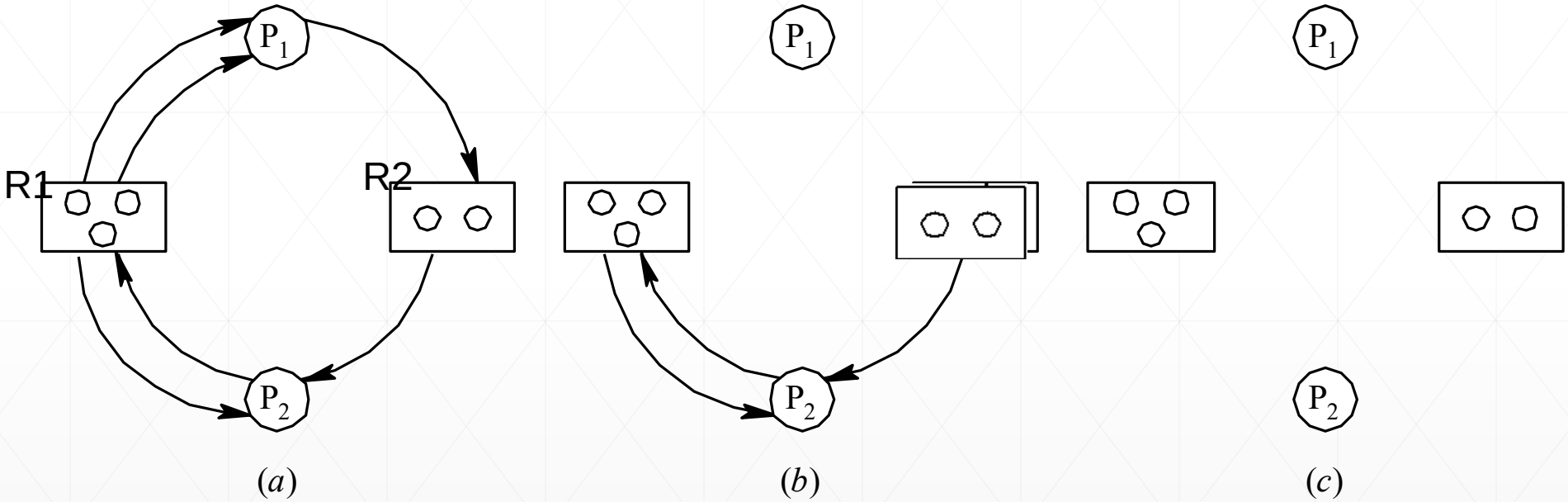
它表示把一个单位的资源R_j分配给进程P_i。

每类资源有多个的情况

化简资源分配图—检测死锁

- 第一步：先看系统还剩下多少资源没分配，再看有哪些进程是不阻塞（“不阻塞”即：系统有足够的空闲资源分配给它）的
- 第二步：把不阻塞的进程的所有边都去掉，形成一个孤立的点，再把系统分配给这个进程的资源回收回来
- 第三步：看剩下的进程有哪些是不阻塞的，然后又把它们逐个变成孤立的点。
- 第四步：最后，所有的资源和进程都变成孤立的点。这样的图就叫做“可完全简化”。
- 如果一个图可完全简化，则不会产生死锁；如果一个图不可完全简化（即：图中还有“边”存在），则会产生死锁。这就是“死锁定理”。

2. 死锁定理：S为死锁状态当且仅当S状态的资源分配图是不可完全简化的。



- 首先去掉图a中的p1结点，去掉p1的两条分配边和一条请求边，成为图b；
- 从而，进程p2可获得所需要的资源并顺利完成，简化p2结点的分配边和申请边，成为图c。

3. 死锁检测中的数据结构

(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。

```
Work =Available;
   L= {Li|Allocationi=0∩Requesti=0}
  for(i=1;  Pi ∉ L ;i++)
      {
            if  Requesti≤Work
            {
              Work=Work+Allocationi;
              Li=Pi;
              L=Li∪L;
            }
      }
   deadlock= ¬(L={p1, p2, …, pn});
```

3.8.2 死锁的解除

- 当发现有进程死锁时，常采用的两种方法是解除死锁：

（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；或者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。

解除死锁

按照解除死锁复杂度递增的顺序列出解除死锁的方法：

1. 撤消死锁进程。

   -该方法是目前操作系统中解除死锁的常用方法。

1. 把死锁进程恢复到前一个检查点，重新执行每个进程。
2. 按照某种原则逐个选择死锁进程进行撤消，直到解除系统死锁。
3. 按照某种原则逐个剥夺进程资源，直到解除死锁。

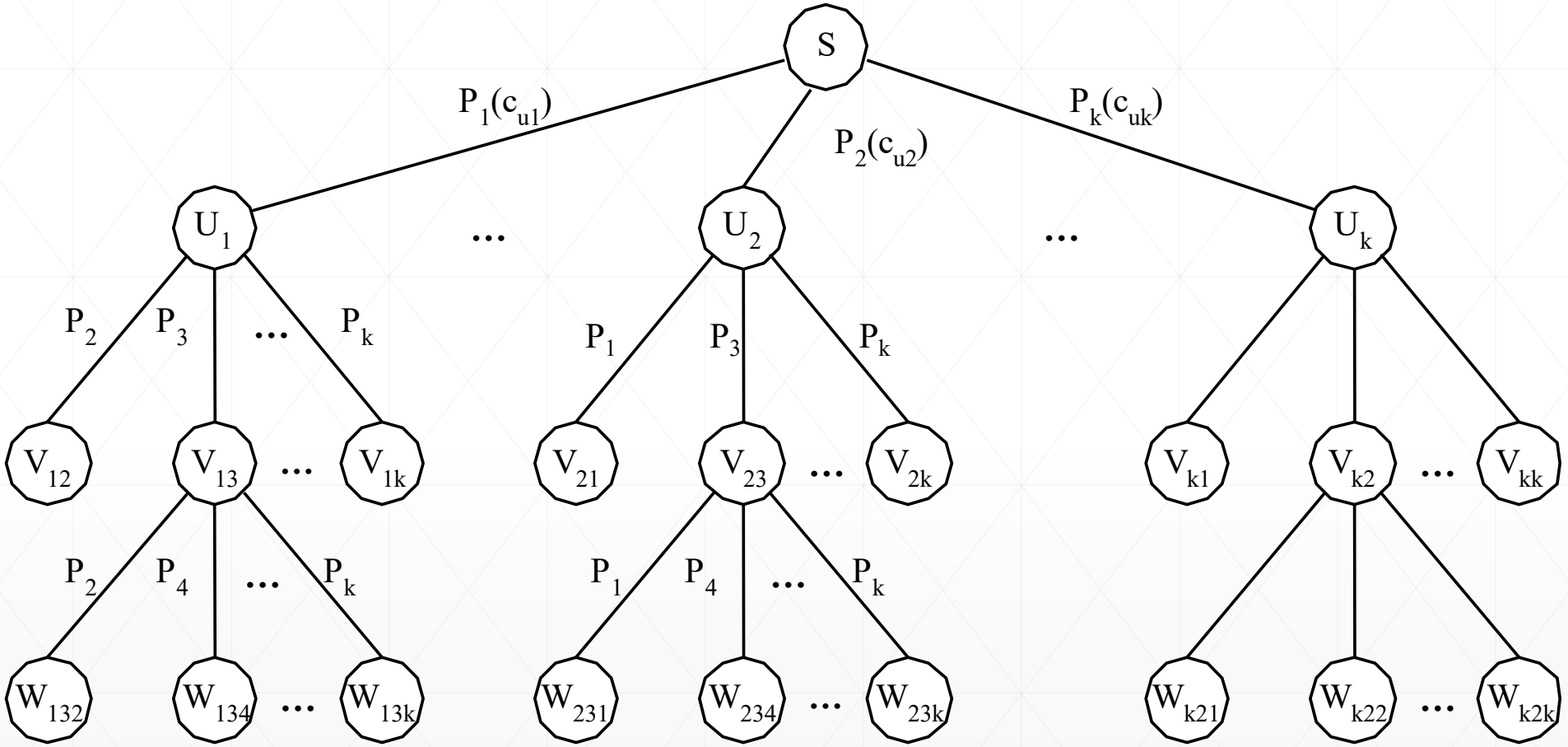
最小代价原则

- 第三种和第四种方法需要选择系统付出代价最小的进程，最小代价原则：
  - 到目前为止，花费处理机的时间最少的进程；
  - 到目前为止，产生输出最少的进程；
  - 估计未执行部分最多的进程；
  - 到目前为止，已获得资源量最少的进程；
  - 优先级最低的进程。

为把系统从死锁状态中解脱出来，所花费的代价(最小)可表示为：

$$R(S)_{min}=min\{C_{ui}\}+min\{C_{uj}\}+min\{C_{uk}\}+\ldots$$

付出代价最小的死锁解除方法



课堂思考

- 在解除死锁的方法中，代价最大的是_____。

A．终止一个死锁进程
B．终止所有的死锁进程
C．重启系统
D．剥夺一个死锁进程的资源

第三章　小结

- 调度类型、模型，重点理解各种算法，比较优缺点；
- 死锁原因，必要条件，处理方法、如何预防；
- 安全状态；
- 银行家算法（结合具体题目），理解方法；
- 死锁检测与解除。

第三章 典型问题分析和解答

1.假设一个系统中有5个进程，它们的到达时间和服务时间如表所示，忽略I/O以及其它开销时间，若分别按先来先服务、非抢占及抢占的短进程优先、高响应比优先、时间片轮转、多级反馈队列（第i级队列的时间片= 2^{i-1}）以及立即抢占的多级反馈队列调度算法进行CPU调度，请给出各进程的完成时间、周转时间、带权周转时间、平均周转时间和平均带权周转时间。

进程	到达时间	服务时间
A	0	3
B	2	6
C	4	4
D	6	5
E	8	2

	进　　程	A	B	C	D	E	平　均
FCFS	完成时间	3	9	13	18	20	-
	周转时间	3	7	9	12	12	8.6
	带权周转时间	1	1.17	2.25	2.4	6	2.56
SPF（非抢占）	完成时间	3	9	15	20	11	-
	周转时间	3	7	11	14	3	7.6
	带权周转时间	1.0	1.17	2.75	2.8	1.5	1.84
SPF（抢占）	完成时间	3	15	8	20	10	-
	周转时间	3	13	4	14	2	7.2
	带权周转时间	1.00	2.16	1	2.8	1.00	1.59
HRRN	完成时间	3	9	13	20	15	-
	周转时间	3	7	9	14	7	8
	带权周转时间	1.00	1.17	2.25	2.80	3.50	2.14
RR（q=1）	完成时间	4	18	17	20	15	-
	周转时间	4	16	13	14	7	10.8
	带权周转时间	1.33	2.67	3.25	2.80	3.50	2.71
FB	完成时间	3	17	18	20	14	-
	周转时间	3	15	14	14	6	10.4
	带权周转时间	1	2.50	3.50	2.80	3.00	2.56
FB（立即抢占）	完成时间	4	18	15	20	16	-
	周转时间	4	16	11	14	8	10.6
	带权周转时间	1.33	2.67	2.75	2.80	4.00	2.87

2.不安全状态是否必然导致系统进入死锁状态?

答：不一定。因为安全性检查中使用的向量Max是进程执行前提供的，而在实际运行过程中，一进程需要的最大资源量可能小于Max，如一进程对应的程序中有一段进行错误处理的代码，其中需要n个A种资源，若该进程在运行过程中没有碰到相应错误而不需调用该段错误处理代码，则它实际上将完全不会请求这n个A种资源 。

并非所有不安全状态都是死锁状态，但当系统进入不安全状态后，便可能进入死锁状态。

3.在哲学家就餐问题中，如果将先拿起左边筷子的哲学家称为左撇子，而将先拿起右边的筷子的哲学家称为右撇子，请说明在同时存在左撇子和右撇子的情况下，任何就座安排都不会产生死锁。

分析：这类题目的关键是必须证明产生死锁的4个必要条件的其中一个不可能成立。在本题中，互斥条件、请求与保持条件、不剥夺条件是肯定成立的，因此必须证明“循环等待”条件不成立。

4.在银行家算法中，若出现下面的资源分配情况，试问：

① **该状态是否安全?**

② **若进程P2提出请求Request（1，2，2，2）后，系统能否将资源分配给它?**

③ **如果系统立即满足P2的上述请求，请问，系统是否立即进入死锁状态?**

Process	Allocation	Need	Available
P0	0，0，3，2	0，0，1，2	1，6，2，2
P1	1，0，0，0	1，6，5，0	
P2	1，3，5，4	2，3，5，6	
P3	0，0，3，2	0，6，5，2	
P4	0，0，1，4	0，6，5，6	

解：①对上面的状态利用安全性算法进行分析，找到了一个安全序列{P0，P3，P4，P1，P2}，故系统是安全的。

资源 / 进程	Work A B C D	Need A B C D	Allocation A B C D	**Work+ Allocation** A B C D	Finish
P0	1 6 2 2	0 0 1 2	0 0 3 2	1 6 5 4	true
P3	1 6 5 4	0 6 5 2	0 0 3 2	1 6 8 6	true
P4	1 6 8 6	0 6 5 6	0 0 1 4	1 6 9 10	true
P1	1 6 9 10	1 6 5 0	1 0 0 0	2 6 9 10	true
P2	2 6 9 10	2 3 5 6	1 3 5 4	3 9 14 14	true

②P2发出请求向量Request（1，2，2，2）后，系统按银行家算法进行检查:

a. $Request_2$（1，2，2，2）≦$Need_2$（2，3，5，6）；

b. $Request_2$（1，2，2，2）≦$Available_2$（1，6，2，2）；

c. 系统先假定可为P2分配资源，并修改相关向量:

Available=（0，4，0，0），$Allocation_2$=（2，5，7，6），$Need_2$=（1，1，3，4）

d. 进行安全性检查，此时对所有的进程，条件$Need_i$≦Available（0，4，0，0）都不成立，即Available不能满足任何进程的请求，故系统进入不安全状态。

因此，进程P2提出资源请求后，系统不能将资源分配给它。

③ 系统立即满足进程P2的请求（1，2，2，2）后，并没有马上进入死锁状态。因为，此时上述进程并没有申请新的资源，也没有因得不到资源而进入阻塞状态。只有当上述进程提出新的请求，并导致所有没执行完的多个进程因得不到资源而阻塞时，系统才进入死锁状态。

5. p个进程共享m个同类资源，每一资源在任一时刻只能供一个进程使用，每一进程对任一资源都只能使用一有限时间，使用完便立即释放，并且每个进程对该类资源的最大需求量小于该类资源的数目。设所有进程对资源的最大需要之和小于p+m。试证：在系统中不会发生死锁。

证明：假设系统发生死锁。

设Max(i)为进程i的最大资源需求量，Need(i)为进程i尚需资源量，Allocation(i)为已分配资源量，则系统在任意时刻有：

$$\sum_{i=1..p}Max(i)=\sum_{i=1..p}Need(i)+\sum_{i=1..p}Allocation(i)<p+m \quad ①$$

系统发生死锁，则一方面说明所有m个资源都应该已经分配出去：

$$\sum_{i=1..p}Allocation(i)=m \quad ②$$

另一方面，进程将处于无限等待状态之中。

由① ②可以得到：$\sum_{i=1..p}Need(i)<p$ ③

即死锁后p个进程还需要的资源量之和少于p，这就意味着此刻至少有一个进程譬如j，已经获得了所需要的全部资源数，因此它的Need（i）=0。但是系统发生死锁时，每个进程至少还需要一个资源单位，则有$\sum_{i=1..p}Need(i)\geq p$才对。既然该进程已经获得了所需要的全部资源数，那么就能完成其任务并释放占有的资源，以保证系统能进一步前进，这与前面的假定死锁矛盾。

作业

- 书P118
- 1、7-9、15、20、27、30、31
- 补充：32，33，34

作业

- **32. 请按照FCFS、SPF、HRRN、RR(q=2)，多级反馈队列算法对下面的进程进行调度，要求画出调度过程。**

进程名	达到时间	服务时间
A	0	3
B	1	6
C	3	2
D	5	5
E	7	4

作业

33. 一个系统有4个进程和5个可分配资源，当前分配和最大需求如下：

进程	已分配资源	最大需求量	可用资源
A	10211	11213	00X12
B	20110	22210	
C	11010	21310	
D	11110	11221	

若保持该状态是安全状态，x的最小值是多少?

作业

34 .假设一个系统中有5个进程，它们的到达时间和服务时间如表所示，忽略I/O以及其它开销时间，若分别按先来先服务、非抢占及抢占的短进程优先、高响应比优先、时间片轮转、多级反馈队列（第i级队列的时间片= 2^{i-1}）以及立即抢占的多级反馈队列调度算法进行CPU调度，请给出各进程的完成时间、周转时间、带权周转时间、平均周转时间和平均带权周转时间。

进程	到达时间	服务时间
A	0	3
B	2	6
C	4	4
D	6	5
E	8	2