



经典教材《计算机操作系统》**最新版**

# 第3章 处理机调度与死锁

主讲教师：陆丽萍

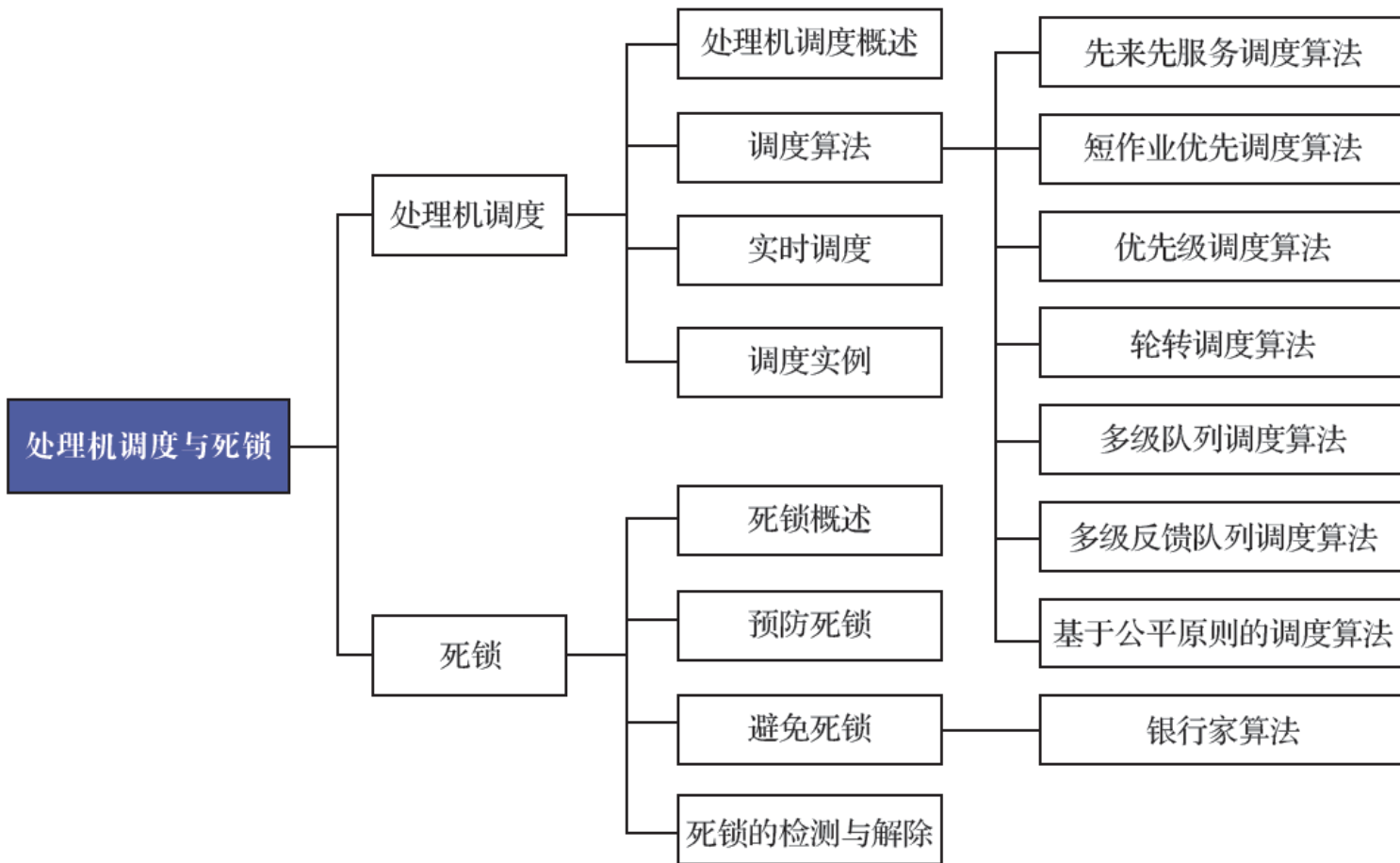
学院：计算机与人工智能学院













# 第3章知识导图

第1章	操作系统引论
第2章	进程的描述与控制
第3章	处理机调度与死锁
第4章	进程同步
第5章	存储器管理
第6章	虚拟存储器
第7章	输入/输出系统
第8章	文件管理





## 内容导航:

-  **3.1 处理机调度概述**
-  3.2 调度算法
-  3.3 实时调度
-  3.4 Linux进程调度
-  3.5 死锁概述
-  3.6 预防死锁
-  3.7 避免死锁
-  3.8 死锁的检测与解除

# 第3章 处理机调度与死锁

---



高级调度（长程调度/作业调度）



低级调度（短程调度/进程调度）



中级调度（中程调度/内存调度）



调度对象：作业



根据某种算法，决定将外存上处于后备队列中的作业调入内存，并为它们创建进程和分配必要的资源。然后，将新创建的进程排在就绪队列上等待调度。



主要用于多道批处理系统中



内存调度，将暂不运行的进程，调至外存等待；



将处于外存上的急需运行的进程，调入内存运行。



即 “对换” 功能

将在第5章 存储器管理中介绍



调度对象：进程



根据某种调度算法，决定就绪队列中的哪个进程应获得处理机



应用在于多道批处理、分时和实时OS



## 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。





## 作业调度的主要任务

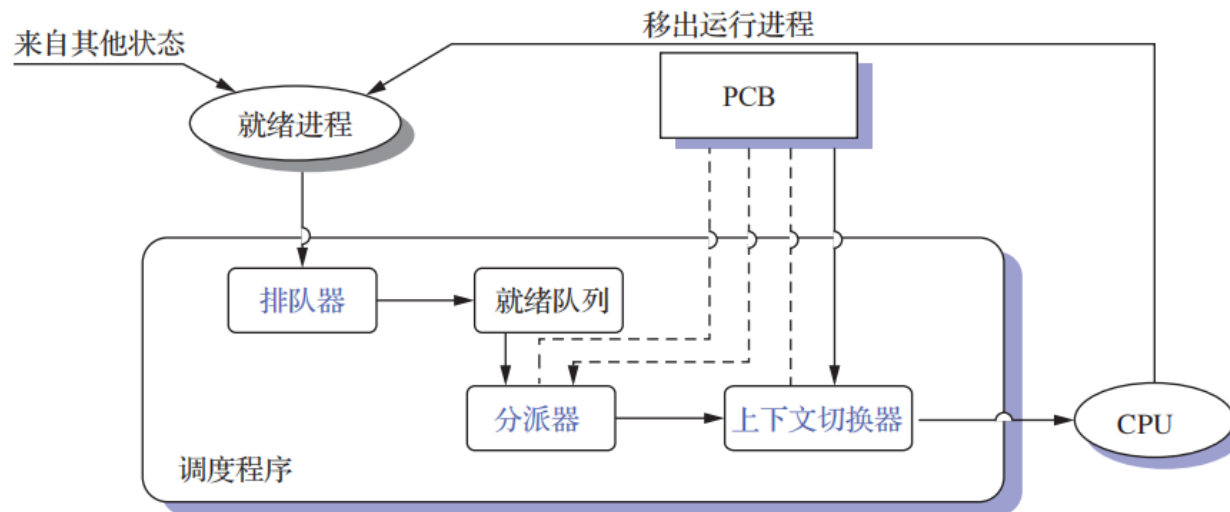
作业调度的主要任务是，根据JCB中的信息，检查系统中的资源能否满足作业对资源的需求，以及按照一定的调度算法，从外存的后备队列中选取某些作业调入内存，并为它们创建进程、分配必要的资源。然后再将新创建的进程排在就绪队列上等待调度。因此，也把作业调度称为接纳调度(Admission Scheduling)。在每次执行作业调度时，都需做出以下两个决定。

1. 接纳多少个作业
2. 接纳哪些作业



## 进程调度的任务

- 保存处理机的现场信息
- 按某种算法选取进程
- 把处理器分配给进程



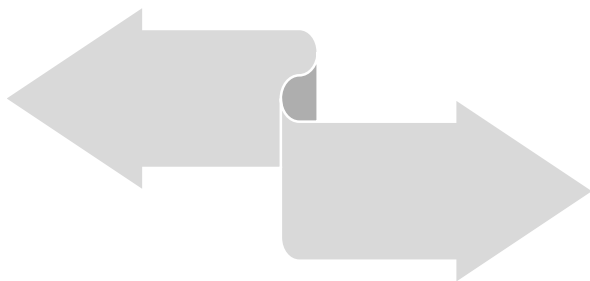
## 进程调度机制（调度程序分为三部分）

- 排队器：用于将就绪进程插入相应的就绪队列
- 分派器：用于将选定的进程移出就绪队列
- 上下文切换器：进行新旧进程之间的上下文切换



### 非抢占方式:

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, 才再把处理机分配给其他进程, 决不允许某进程抢占已经分配出去的处理机。



**抢占方式:** 允许调度程序根据某种原则, 去暂停某个正在执行的进程, 将已分配给该进程的处理机重新分配给另一进程。(现代OS广泛采用)

- 优先权原则: 允许优先权高的新到进程抢占当前进程的处理机
- 短作业优先原则: 短作业可以抢占当前较长作业的处理机
- 时间片原则: 各进程按时间片运行, 当一个时间片用完后, 便停止该进程的执行而重新进行调度



## 共同目标:

### ➤ 资源利用率

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

### ➤ 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。

### ➤ 平衡性

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。

### ➤ 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。



## 批处理系统的目标:

- 平均周转时间短、系统吞吐量高、处理机利用率高



## 周转时间:

- 从作业提交给系统开始，到作业完成为止的这段时间间隔。

作业*i*的周转时间 $T_i$ 为

$$T_i = T_{ei} - T_{si}$$

其中 $T_{ei}$ 为作业*i*的完成时间， $T_{si}$ 为作业的提交时间。

一个作业的周转时间说明了该作业在系统内停留的时间，包含两部分：等待时间；

执行时间，即： $T_i = T_{wi} + T_{ri}$

$$T = \frac{1}{n} \left( \sum_{i=1}^n T_i \right)$$

- 平均周转时间
- 带权周转时间：权值为作业周转时间 $T$ 与系统为之服务时间 $T_s(T_{ri})$ 之比。
- 平均带权周转时间

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



响应时间：

- 从用户通过键盘提交请求开始，直到系统首次显示出处理结果为止的一段时间。



等待时间（进程调度）：

- 进程在就绪队列中等待调度的所有时间之和。



# 处理机调度算法的目标



分时系统的目标：

- 响应时间快、均衡性





实时系统的目标：

- 截止时间的保证、可预测性



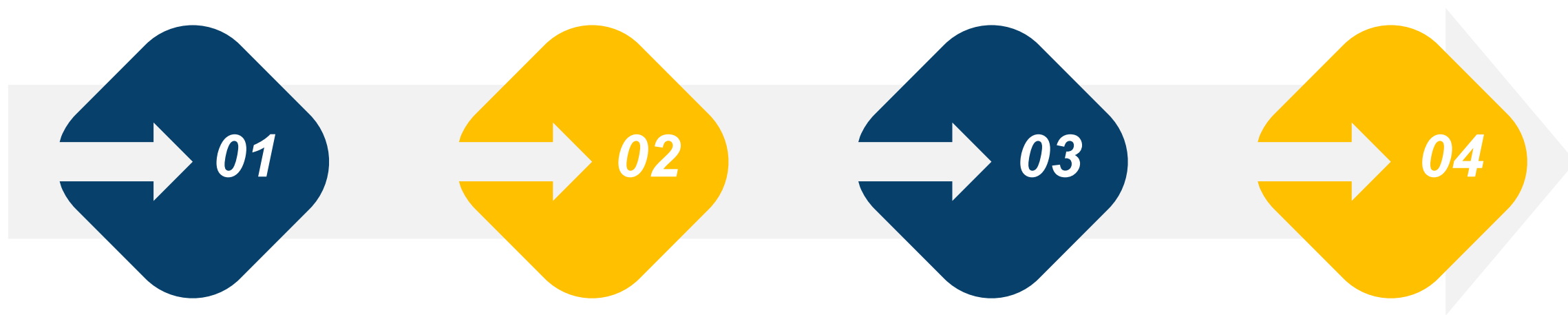
## 内容导航:

-  3.1 处理机调度概述
-  **3.2 调度算法**
-  3.3 实时调度
-  3.4 Linux进程调度
-  3.5 死锁概述
-  3.6 预防死锁
-  3.7 避免死锁
-  3.8 死锁的检测与解除

## 第3章 处理机调度与死锁

---





➤ 先来先服务调度算法(FCFS)

➤ 短作业优先调度算法(SJF)

➤ 优先级调度算法(PR)

➤ 高响应比优先调度算法(HRRN)

---

FCFS、SJF、PR既可用于作业调度，也可用于进程调度

---



先来先服务调度算法(FCFS)



多级队列调度算法



短作业优先调度算法(SJF)



多级反馈队列调度算法



优先权调度算法(PR)



基于公平原则的调度算法



时间片轮转调度算法(RR)

00

按照作业到达的先后次序来进行调度

作业	运行时间
J1	24
J2	3
J3	3

00

假定作业到达顺序如下: J1, J2, J3  
该调度的甘特图(Gantt)为:



➤ 平均等待时间 =  $(0 + 24 + 27)/3 = 17$

➤ 平均周转时间 =  $(24 + 27 + 30)/3 = 27$

01

假定进程到达顺序如下 J<sub>2</sub> , J<sub>3</sub> , J<sub>1</sub> .

02

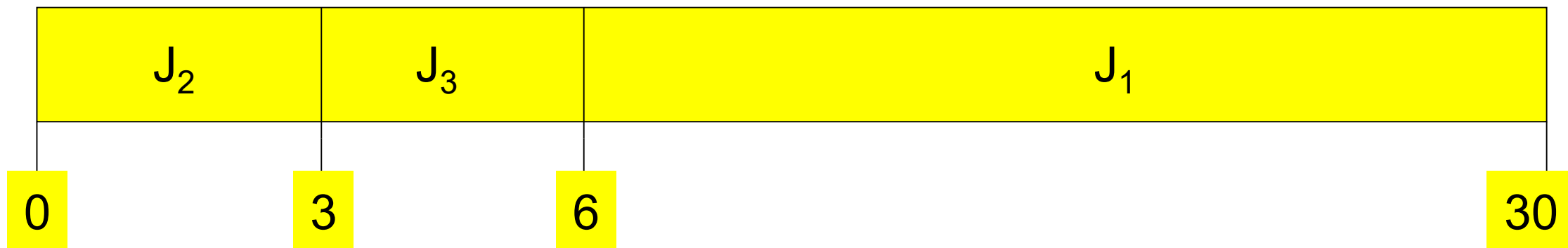
该调度的Gantt图为：

03

比前例好得多

04

此结果产生是由于短进程先于长进程到达



➤ 平均等待时间 =  $(6 + 0 + 3)/3 = 3$

➤ 平均周转时间 =  $(30 + 3 + 6)/3 = 13$



SJF算法：既可用于作业，也可用于进程

- 对作业：从后备队列中选择若干个估计运行时间最短的作业。
- 对进程：关联到每个进程下次运行的CPU区间长度，调度最短的进程。



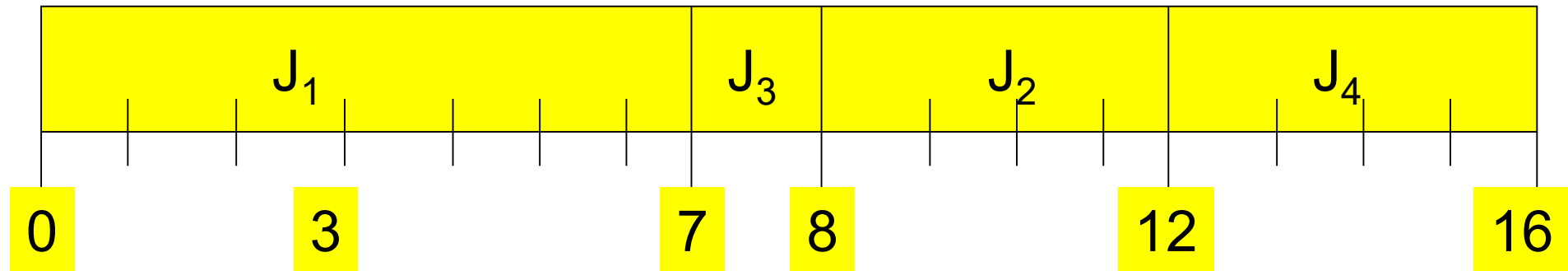
对进程调度，SJF有两种模式：

- 非抢占式SJF
- 抢占式SJF—抢占发生在有比当前进程剩余时间片更短的进程到达时，也称为最短剩余时间优先调度



SJF是最优的（对一组指定的进程而言），它给出了最短的平均等待时间。

进程	到达时间	运行时间
$J_1$	0.0	7
$J_2$	2.0	4
$J_3$	4.0	1
$J_4$	5.0	4

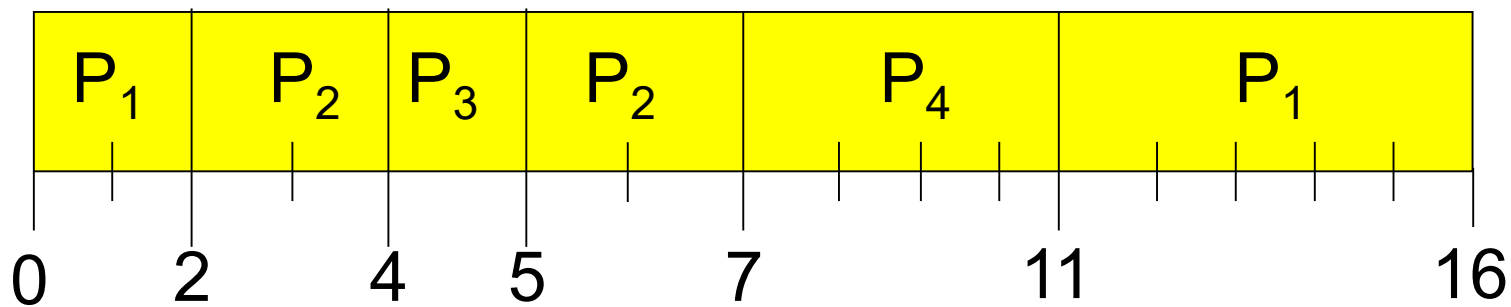


➤ 平均等待时间 =  $(0 + 6 + 3 + 7)/4 = 4$

➤ 平均周转时间 =  $(7 + 10 + 4 + 11)/4 = 8$

前例:

<u>进程</u>	<u>到达时间</u>	<u>区间时间</u>
P1	0	7
P2	2	4
P3	4	1
P4	5	4



➤ 平均等待时间 =  $(9 + 1 + 0 + 2)/4 = 3$

➤ 平均周转时间 =  $(16 + 5 + 1 + 6)/4 = 7$



SJF比FCFS算法有明显改进

### 缺点:



只能估算进程的运行时间 (估值不准确), 所以通常用于作业调度



对长作业不利



采用SJF算法时, 人-机无法实现交互



完全未考虑作业的紧迫程度





既可用于作业调度，也可用于进程调度。



基于作业/进程的紧迫程度，由外部赋予作业相应的优先级，调度算法根据优先级进行调度。

- 每个进程都有一个优先数，优先数为整数。
- 默认：小的优先数具有高优先级。
- 目前主流的操作系统调度算法。



高响应比优先调度算法是一种优先级调度算法，用于作业调度。



## 优先级调度算法的类型

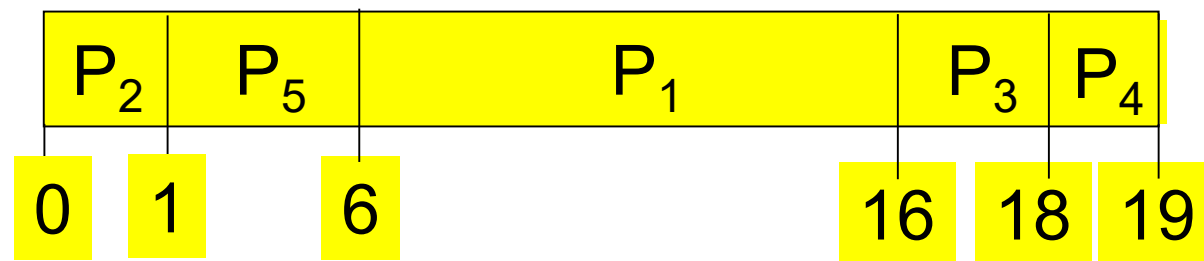
- 非抢占式
- 抢占式



## 优先级类型

- 静态优先级
  - ❑ 创建进程时确定优先数(整数), 在进程的整个运行期间保持不变
  - ❑ 简单易行, 系统开销小
  - ❑ 不够精确, 可能会出现优先级低的进程长期没有被调度的情况
- 动态优先级
  - ❑ 创建进程时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

进程	优先级	运行时间
P1	3	10
P2	1	1
P3	3	2
P4	4	1
P5	2	5



- 平均等待时间 =  $(6+0+16+18+1)/5 = 8.2$
- 平均周转时间 =  $(16 + 1 + 18+19+6)/5 = 12$



## 优点

- 实现简单，考虑了进程的紧迫程度
- 灵活，可模拟其它算法



## 存在问题

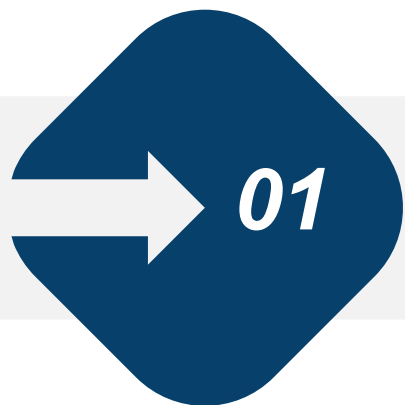
- 饥饿 —— 低优先级的进程可能永远得不到运行



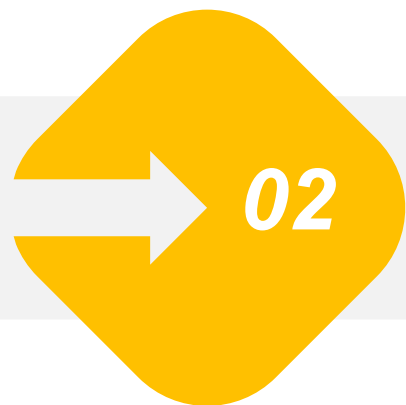
## 解决方法

- 老化 —— 视进程等待时间的延长提高其优先数

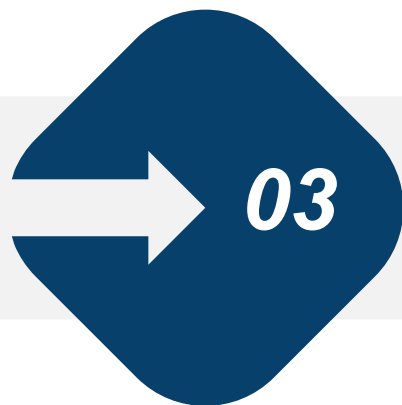
- 既考虑作业的等到时间，又考虑作业的运行时间
- 优先级：  $\text{优先级} = \frac{\text{等待时间} + \text{要求服务时间}}{\text{要求服务时间}}$
- 响应比：  $R_p = \frac{\text{等待时间} + \text{要求服务时间}}{\text{要求服务时间}} = \frac{\text{响应时间}}{\text{要求服务时间}}$
- 如等待时间相同，运行时间越短，类似于SJF
- 如运行时间相同，取决于等待时间，类似于FCFS
- 长作业可随其等待时间的增加而提高，也可得到服务
- 缺点：每次调度之前，都需要计算响应比，增加系统开销



专为分时系统设计，类似于FCFS，但增加了抢占



时间片  
➤ 小单位的CPU时间，通常为10~100毫秒



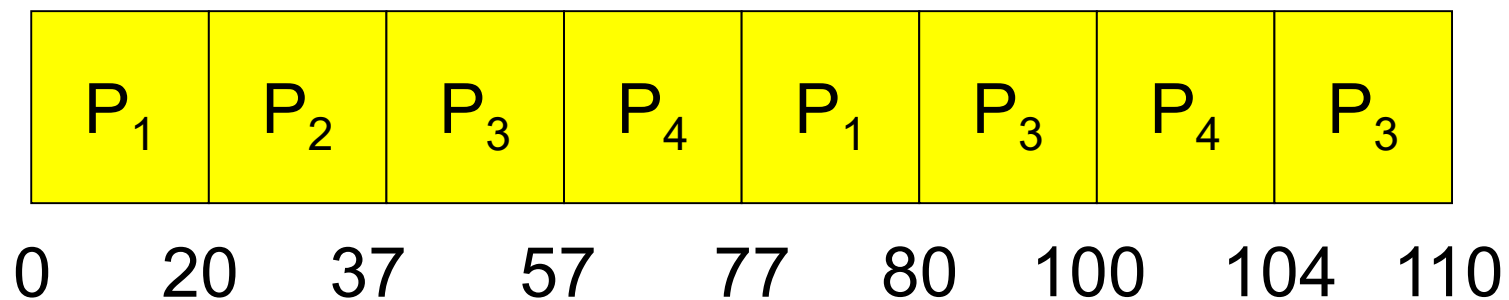
为每个进程分配不超过一个时间片的CPU。时间片用完后，该进程将被抢占并插入就绪队列末尾，循环执行



假定就绪队列中有 $n$ 个进程、时间片为 $q$ ，则每个进程每次得到 $1/n$ 的、不超过 $q$ 单位的成块CPU时间，没有任何一个进程的等待时间会超过 $(n-1)q$ 单位

进程	运行时间
P1	23
P2	17
P3	46
P4	24

➤ Gantt图如下:



- 平均等待时间:  $(57+20+64+80)/4 = 55.25$
- 平均响应时间:  $(80+37+110+104)/4=253$
- 通常, RR的平均周转时间比SJF长, 但响应时间要短一些.



## 特性

- q 大 □ FCFS
- q 小 □ 增加上下文切换的时间

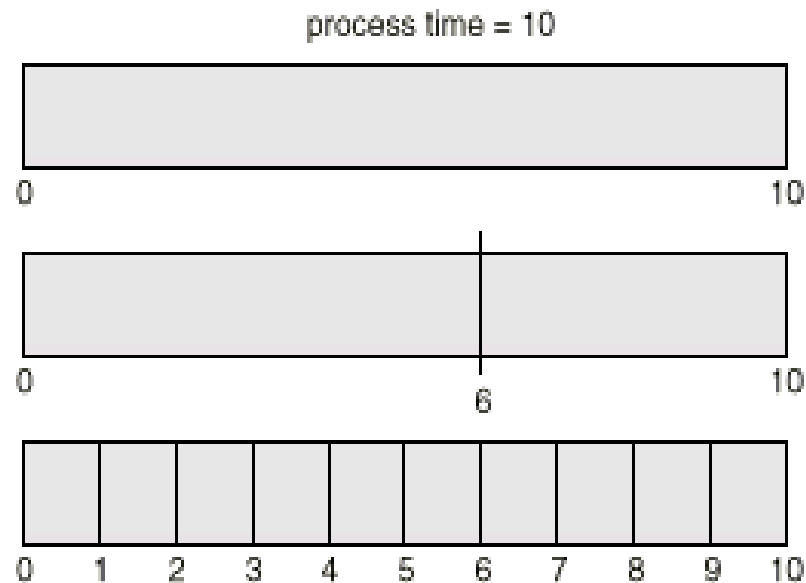


## 时间片设置应考虑

- 系统对响应时间的要求
- 就绪队列中进程的数目
- 系统的处理能力



**一般准则：** 时间片/10 > 进程上下文切换时间



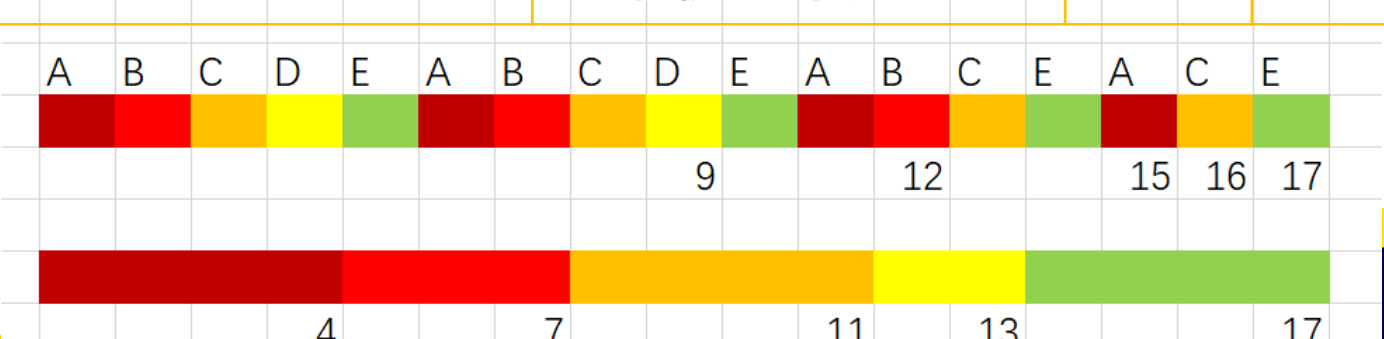
quantum	context switches
12	0
6	1
1	9





# RR例子

作业情况		进程名	A	B	C	D	E	平均
时间片	到达时间		0	1	2	3	4	
	服务时间		4	3	4	2	4	
	完成时间		15	12	16	9	17	
RR q=1		周转时间	15	11	14	6	13	11.8
		带权周转时间	3.75	3.67	3.5	3	3.25	3.43
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	3.25	2.7





就绪队列从一个分为多个，如：

- 前台[交互式]
- 后台[批处理]



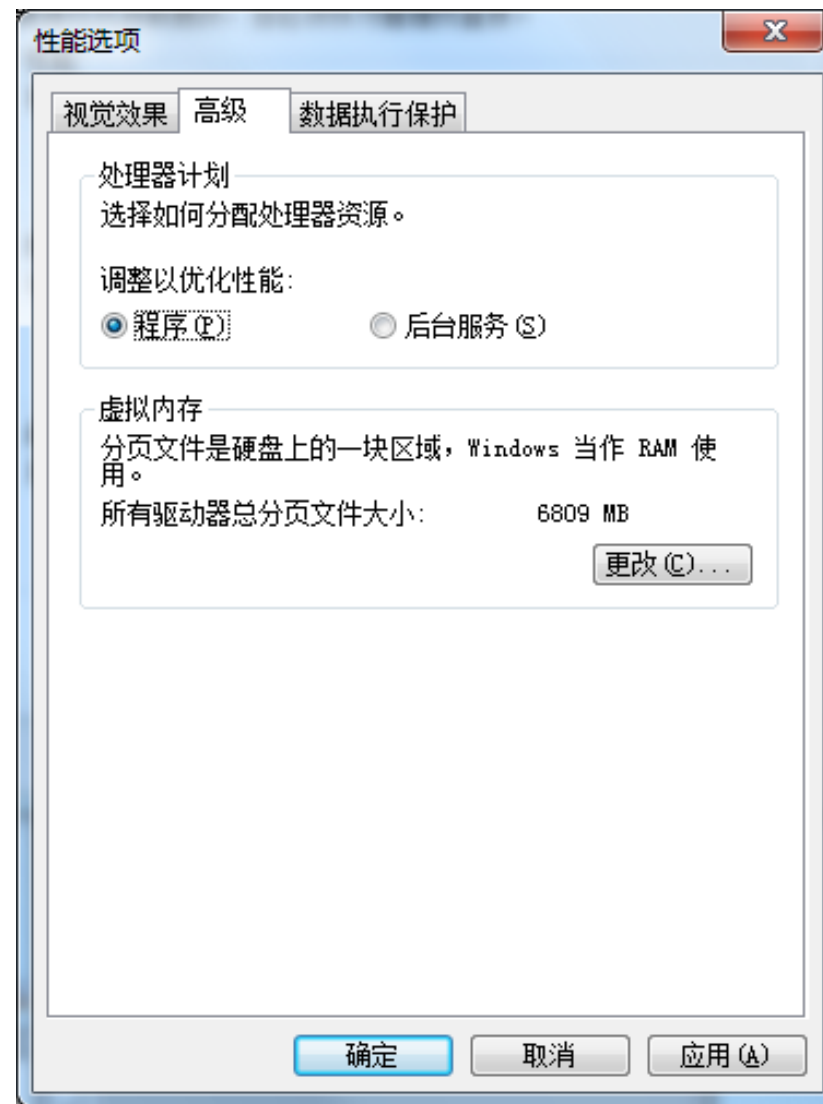
每个队列有自己的调度算法

- 前台 – RR
- 后台 – FCFS



调度须在队列间进行

- 固定优先级调度，即前台运行完后再运行后台，有可能产生饥饿。
- 给定时间片调度，即每个队列得到一定的CPU时间，进程在给定时间内执行；如80%的时间执行前台的RR调度，20%的时间执行后台的FCFS调度





进程能在不同的队列间移动



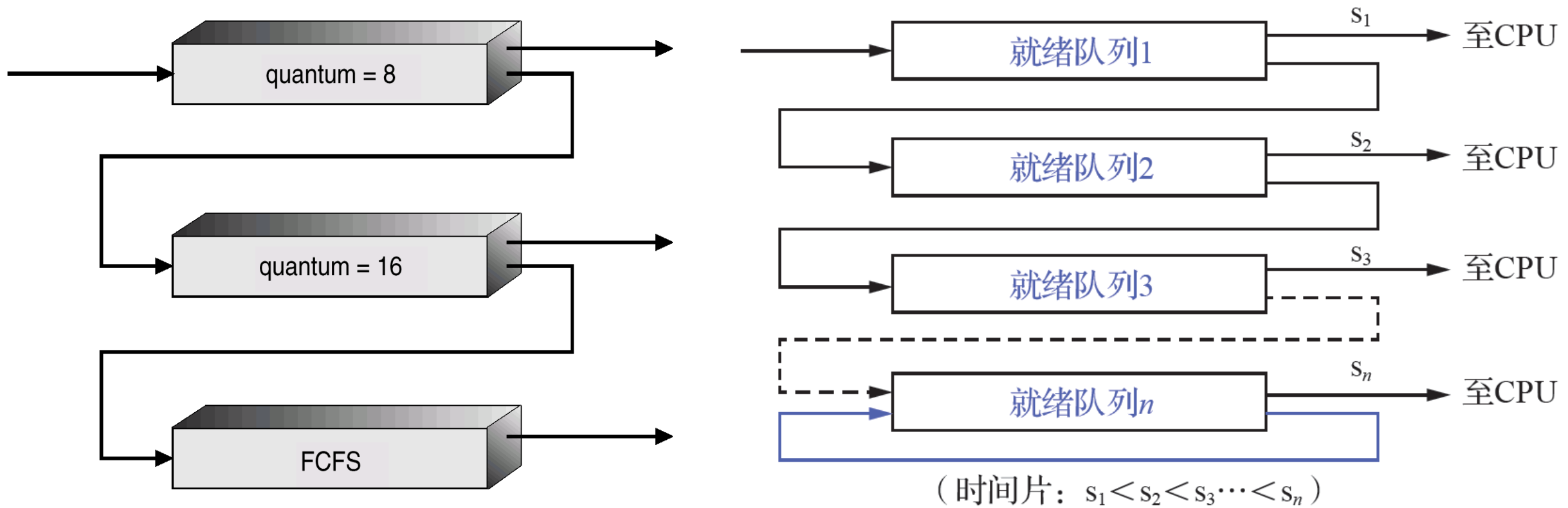
其他调度算法的局限性

- 短进程优先的调度算法，仅照顾了短进程而忽略了长进程
- 如果并未指明进程的长度，则短进程优先和基于进程长度的抢占式调度算法都将无法使用。



优点：

- 不必事先知道各种进程所需的执行时间；
- 可以满足各种类型进程的需要。





主要考虑调度的公平性。



保证调度算法：

- 性能保证，而非优先运行；
- 如保证处理机分配的公平性（处理机时间为 $1/n$ ）。











公平分享调度算法：

- 调度的公平性主要针对用户而言；
- 使所有用户能获得相同的处理机时间或时间比例。



## 内容导航:

-  3.1 处理机调度概述
-  3.2 调度算法
-  **3.3 实时调度**
-  3.4 Linux进程调度
-  3.5 死锁概述
-  3.6 预防死锁
-  3.7 避免死锁
-  3.8 死锁的检测与解除

# 第3章 处理机调度与死锁

---



实时调度是针对实时任务的调度



实时任务，都联系着一个截止时间

- 硬实时HRT任务
- 软实时SRT任务



实时调度应具备一定的条件

## 1. 提供必要的信息

- 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。
- 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。
- 处理时间，一个任务从开始执行，直至完成时所需的时间。
- 资源要求，任务执行时所需的一组资源。
- 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。



## 2. 系统处理能力强

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 $P_i$ ，

■ 单处理机系统，满足 
$$\sum_{i=1}^m \frac{C_i}{P_i} \leq 1$$

■ 多处理机系统，满足

假定系统中的处理机数为 $N$ ，则应将上述的限制条件改为：

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



3. 采用抢占式调度机制

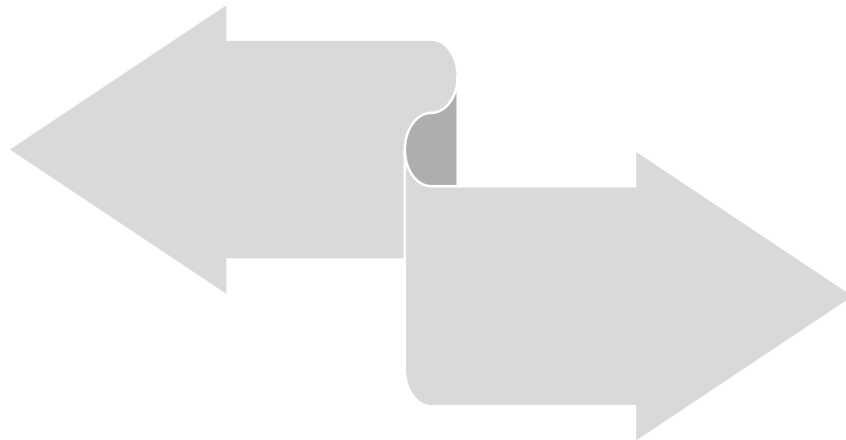
4. 采用快速切换机制

- 对中断具有快速响应能力
- 快速的任務分派能力



根据实时任务性质

- HRT调度算法
- SRT调度算法



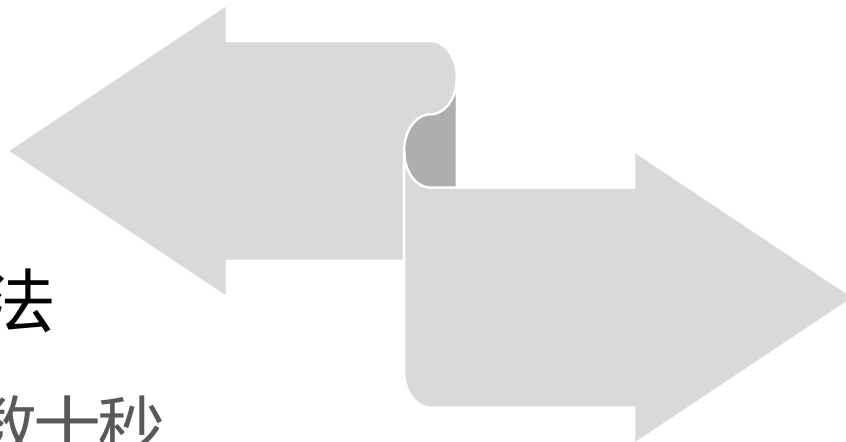
根据调度方式

- 非抢占式调度算法
- 抢占式调度算法



### 非抢占式轮转调度算法

- 响应时间：数秒至数十秒
- 可用于要求不太严格的实时控制系统



### 非抢占式优先调度算法

- 响应时间：数秒至数百毫秒
- 可用于有一定要求的实时控制系统



## 基于时钟中断的抢占式优先级调度

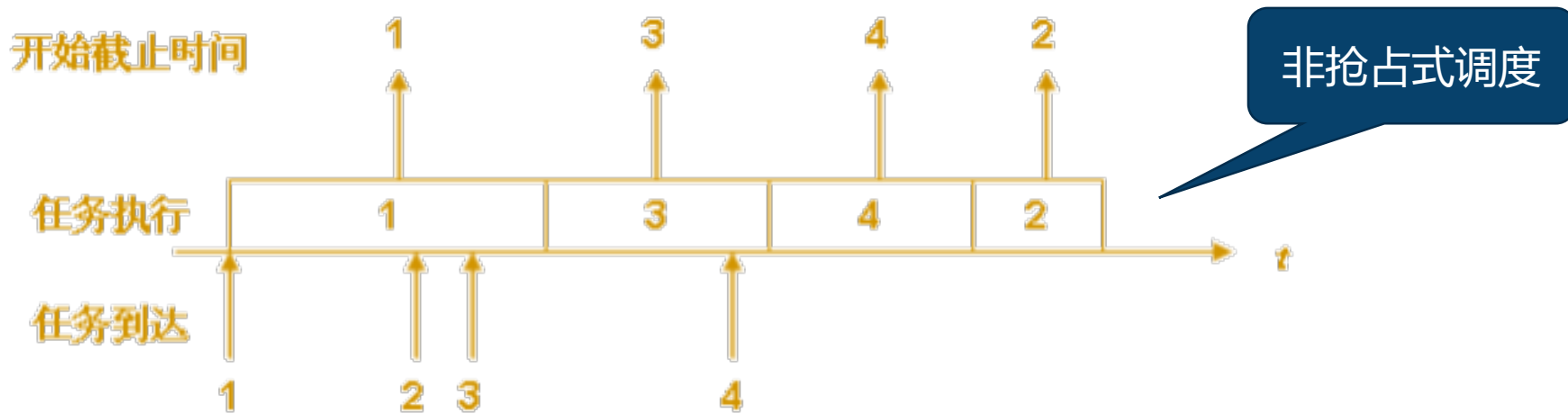
- 响应时间：几十毫秒至几毫秒
  - 可用于大多数实时系统
- 



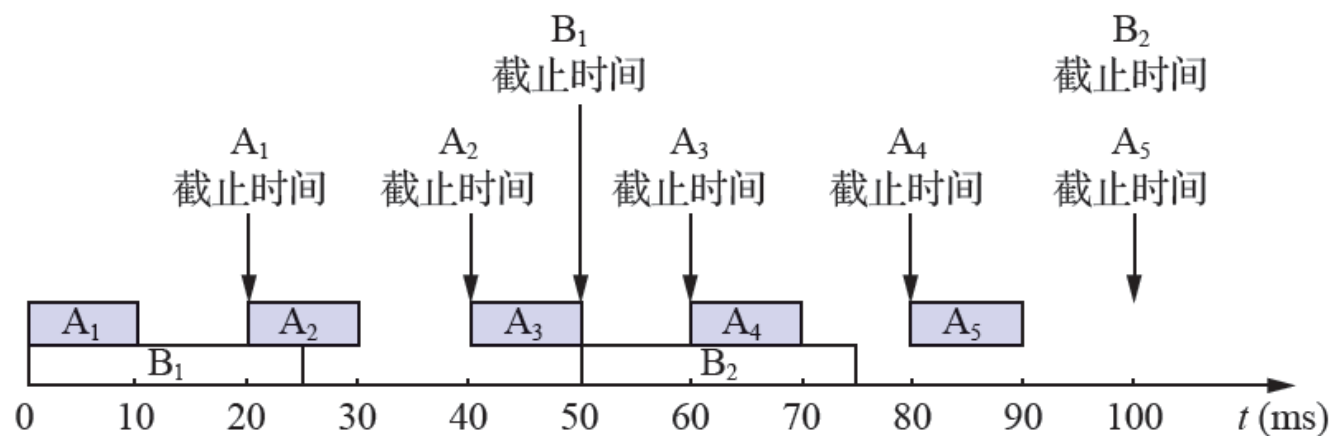
## 立即抢占的优先级调度

- 响应时间：几毫秒至几百微秒
- 可用于有严格时间要求的实时系统

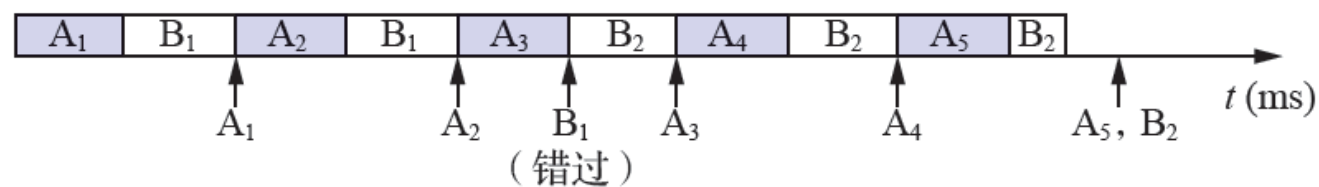
- OS EDF根据任务的截止时间确定优先级，截止时间越早，优先级越高
- OS 既可用于抢占式调度，也可用于非抢占式调度
- OS 非抢占式调度用于非周期实时任务
- OS 抢占式调度用于周期实时任务



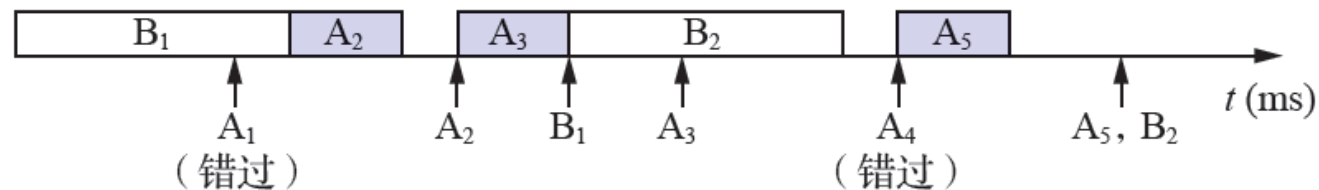
到达时间、执行时间和最后截止时间



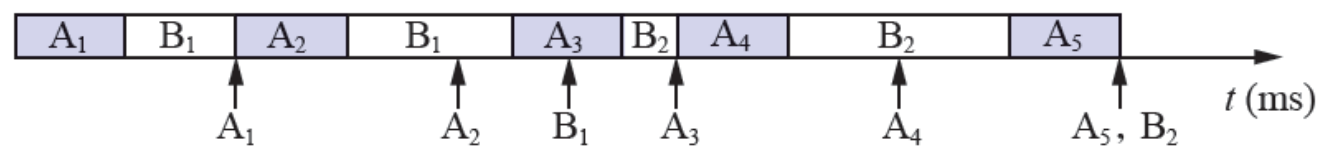
固定优先级调度



固定优先级调度



使用完成截止时间最早和最后截止时间调度





根据任务的紧急程度（**松弛度**）确定任务优先级

- 紧急程度越高（松弛度越低），优先级越高
- $\text{松弛度} = \text{必须完成时间} - \text{其本身的运行时间} - \text{当前时间}$



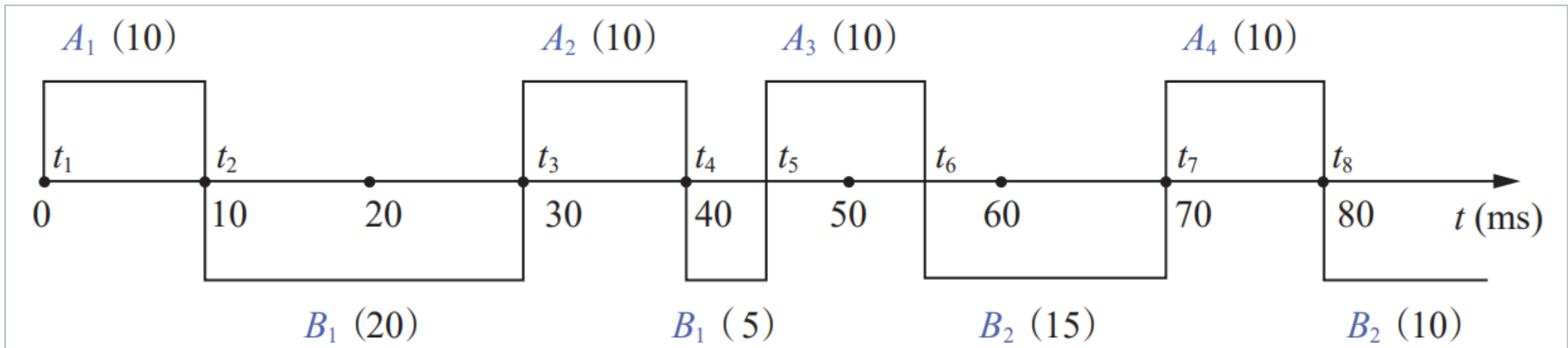
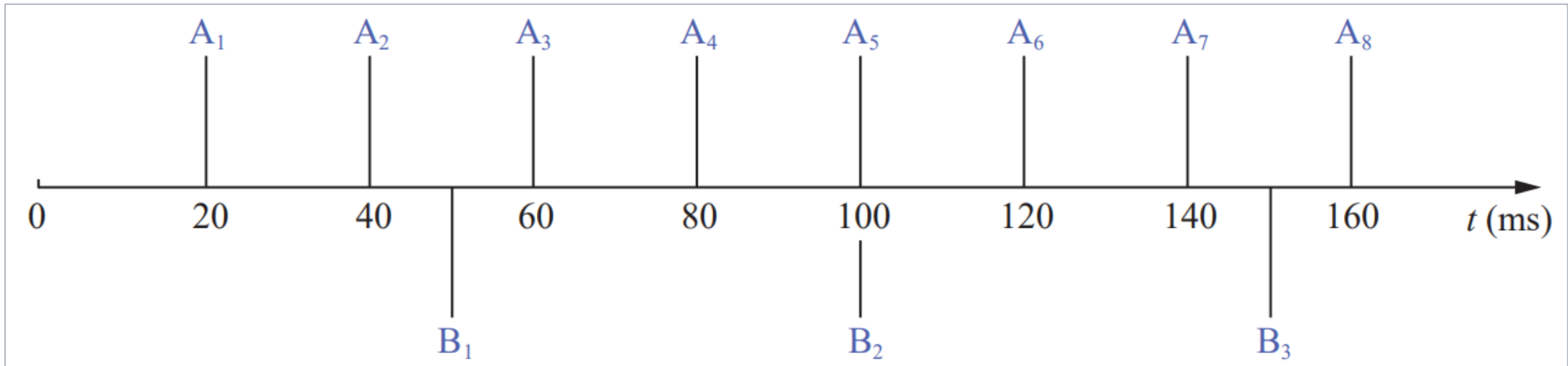
主要用在抢占式调度方式中，松弛度为0是发生抢占



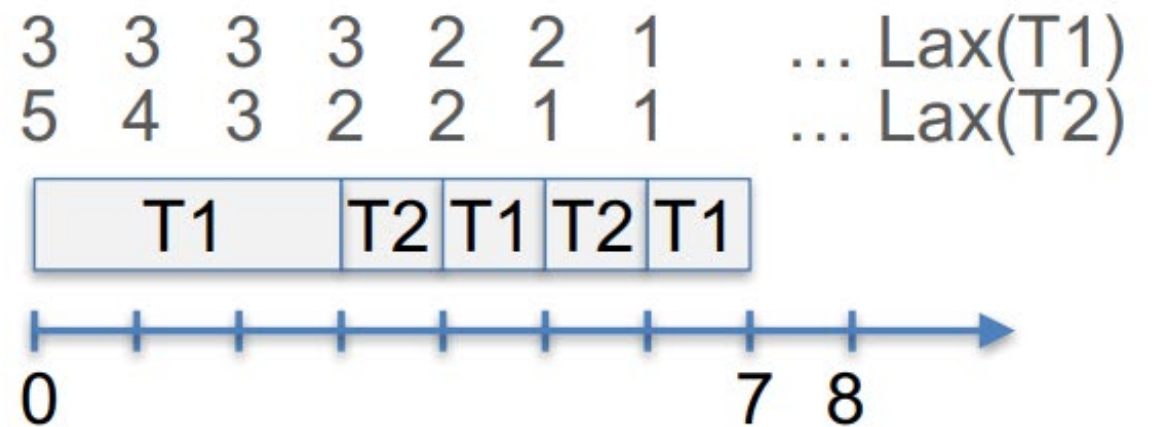
例子：

- 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



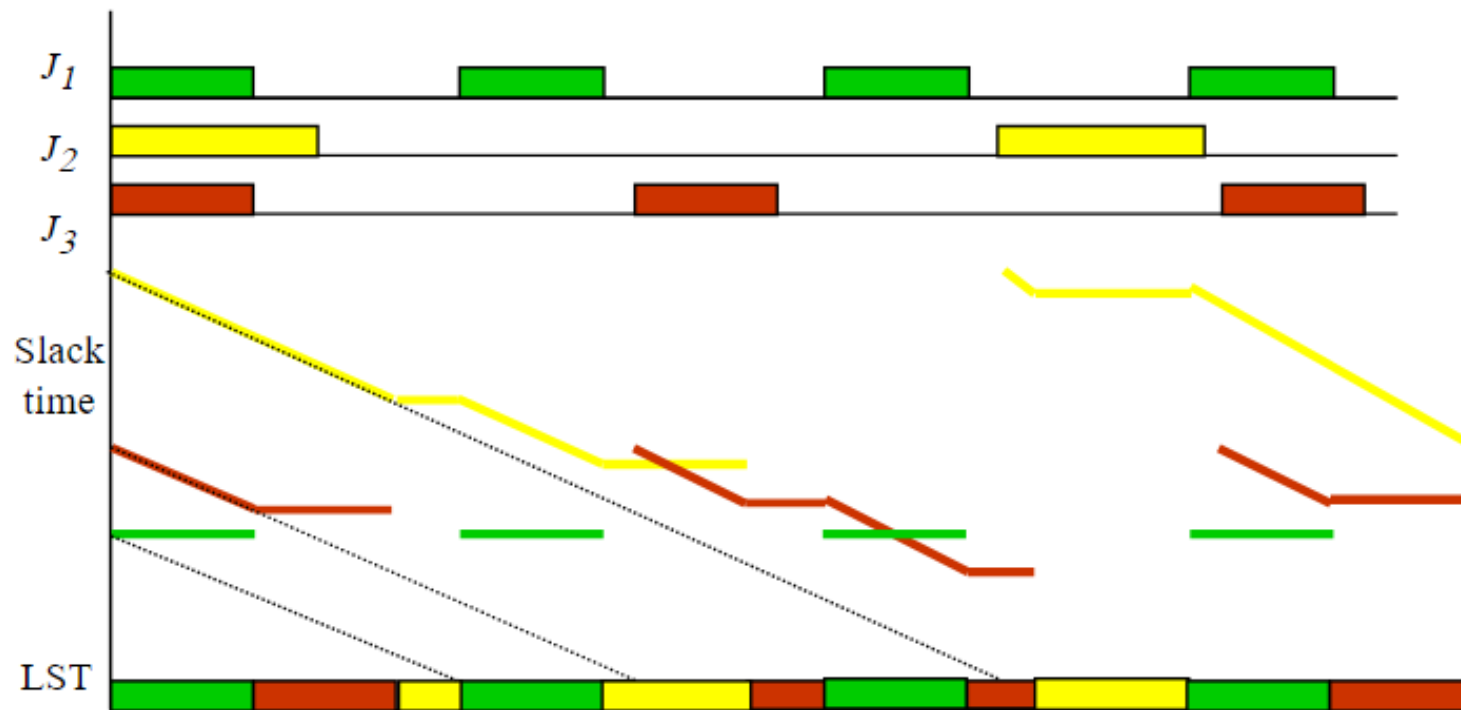


Task	Deadline	WCET
T1	8	5
T2	7	2



## □ Priority preemptive scheduling based on slack (laxity) time ( $d_i - e_i^*$ )

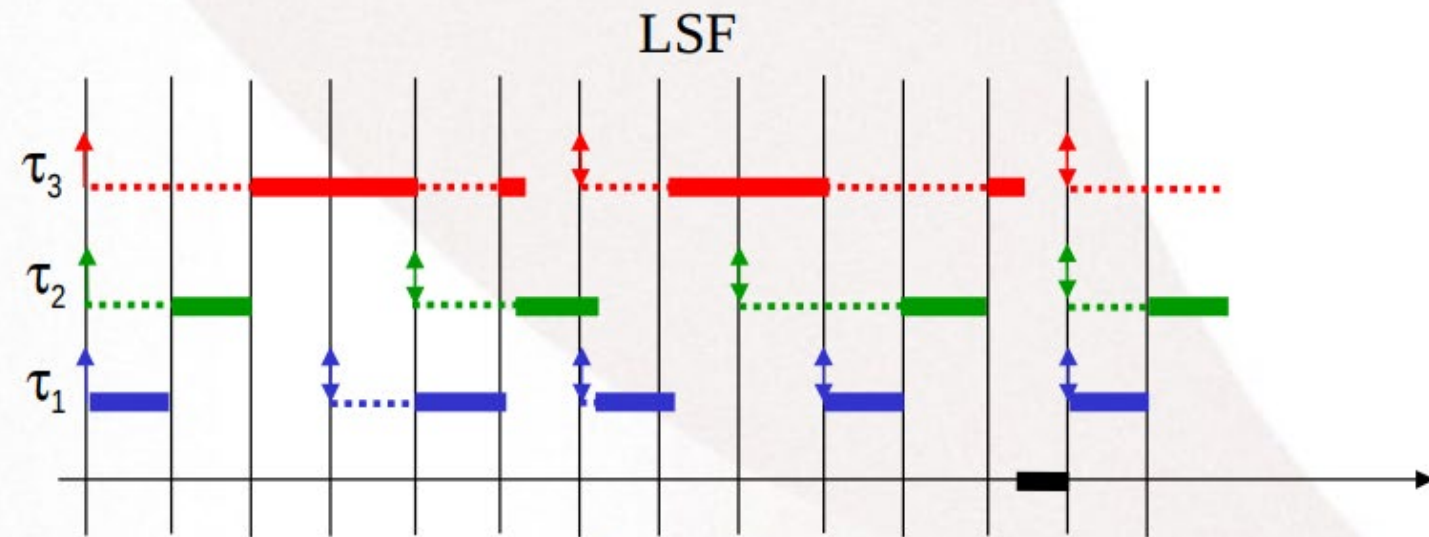
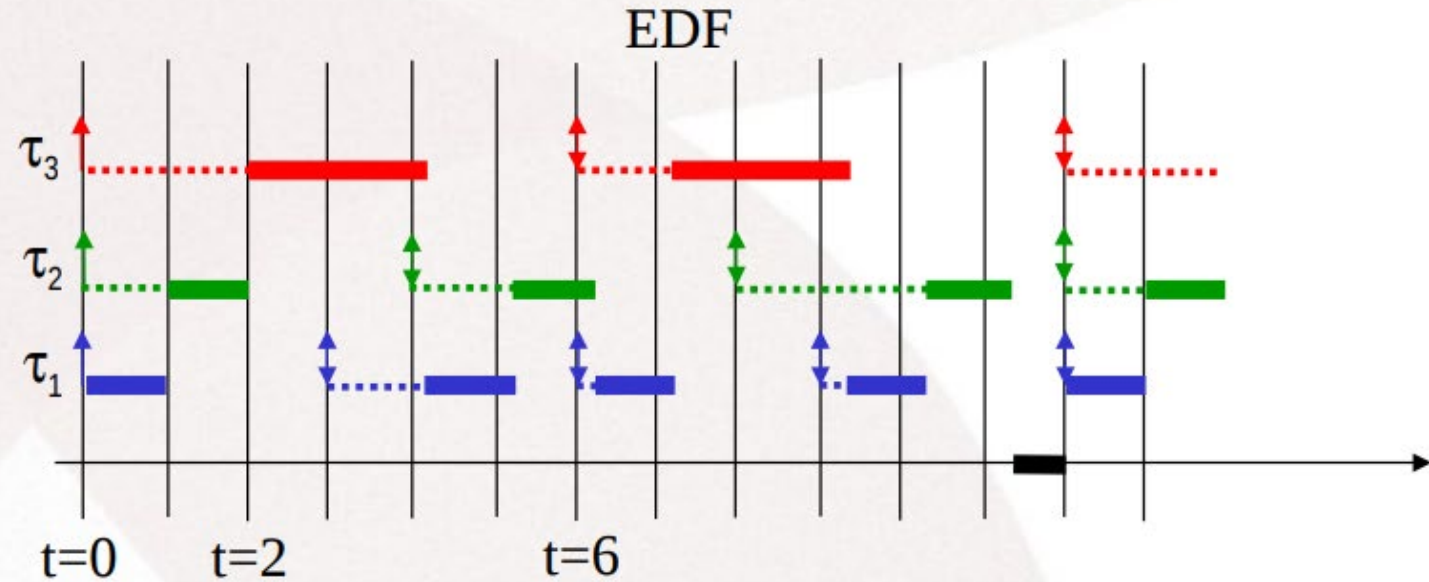
- ❖ schedule instants: when jobs are released or completed.
- ❖ optimal for preemptive single processor schedule




# LSF scheduling– same example

Task set

$\tau_i$	$T_i$	$C_i$
1	3	1
2	4	1
3	6	2.1



 采用优先级调度和抢占方式，可能产生**优先级倒置**。现象：高优先级进程被低优先级进程延迟或阻塞。

---

 **解决方法：**

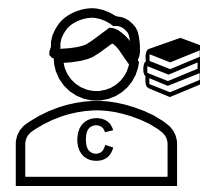
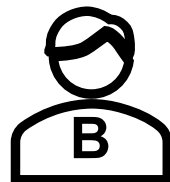
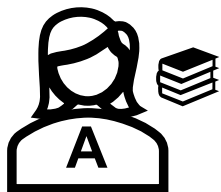
- 制定一些规定，如规定低优先级进程执行后，其所占用的处理机不允许被抢占；
- 建立动态优先级继承。

## 问题2：优先级反转



优先级：A>B>C

**问题：**  
A被C占有的资源**阻塞**  
优先级较低的B先于A学习



1. 申请秘籍成功

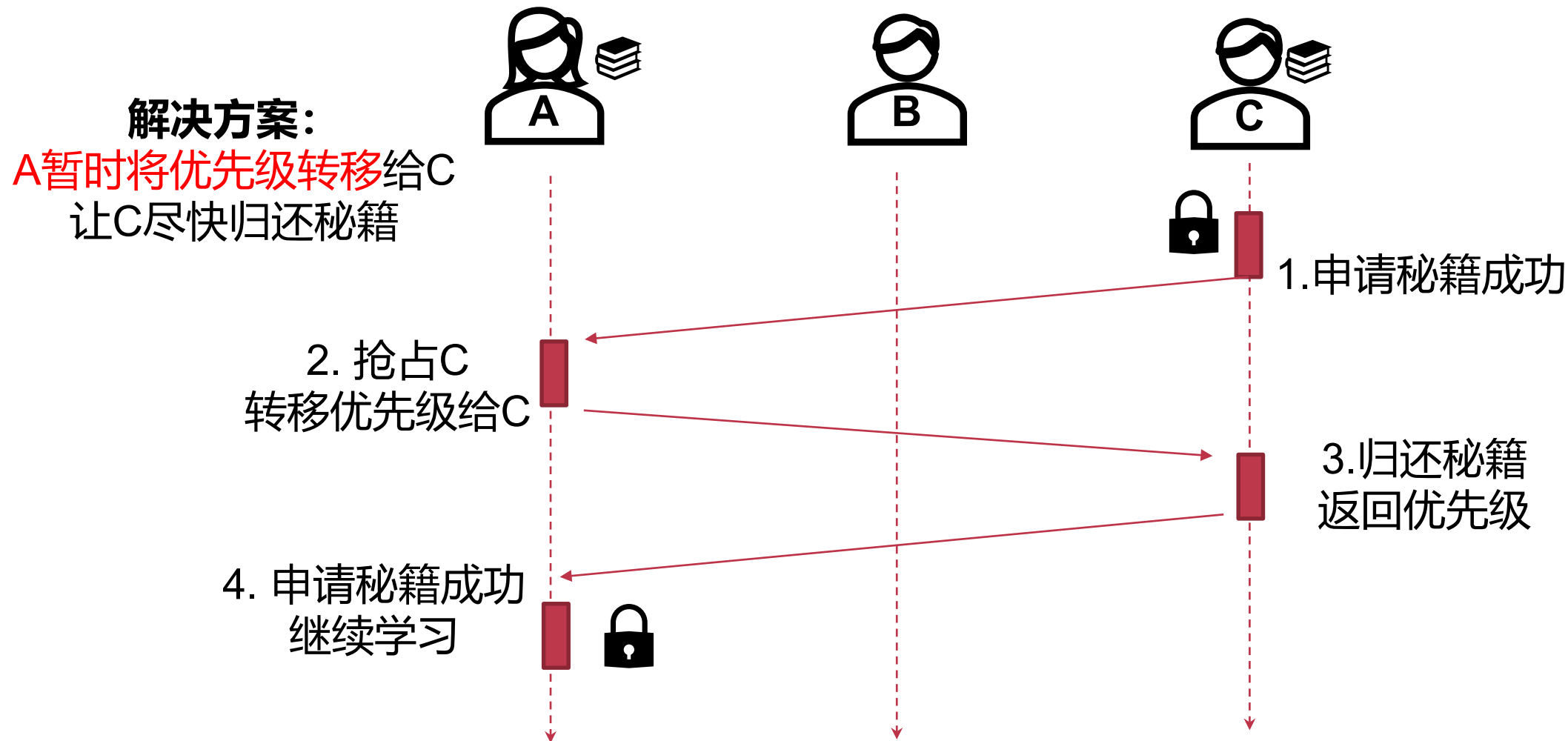
2. 抢占C  
申请秘籍失败  
等待

3. B优先级高于C  
可以向学霸学习

# 解决方法：优先级继承









优先级：A>B>C





## 内容导航:

-  3.1 处理机调度概述
-  3.2 调度算法
-  3.3 实时调度
-  **3.4 Linux进程调度**
-  3.5 死锁概述
-  3.6 预防死锁
-  3.7 避免死锁
-  3.8 死锁的检测与解除

## 第3章 处理机调度与死锁

---





# Linux进程调度 (1)

默认调度算法：完全公平调度CFS算法。

基于调度器类：允许不同的可动态添加的调度算法并存，每个类都有一个特定的优先级。



总调度器：根据调度器类的优先顺序，依次对调度器类中的进程进行调度。



调度器类：使用所选的调度器类算法（调度策略）进行内部的调度。

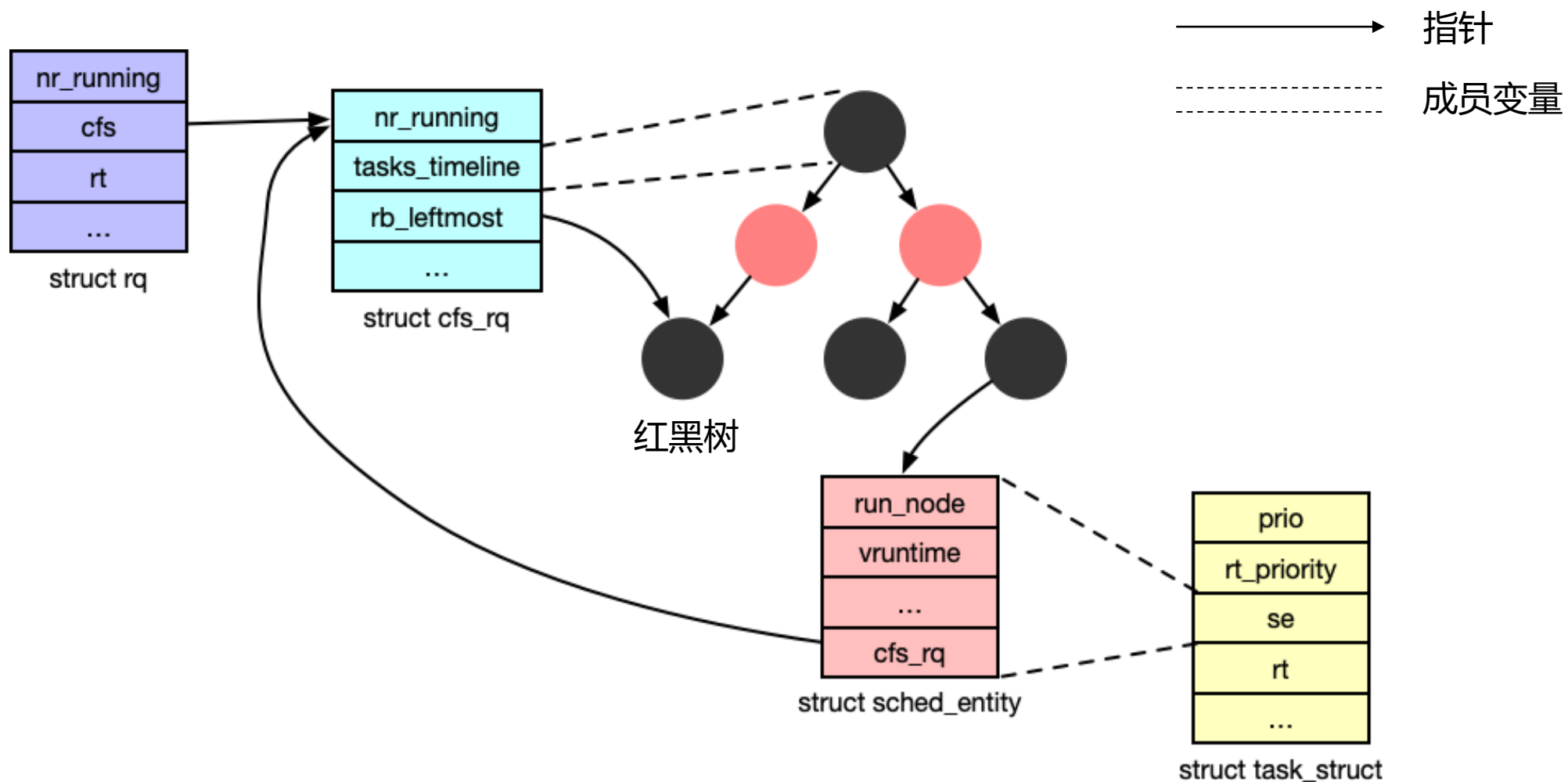


调度器类的默认优先级顺序为：Stop\_Task > Real\_Time > Fair > Idle\_Task

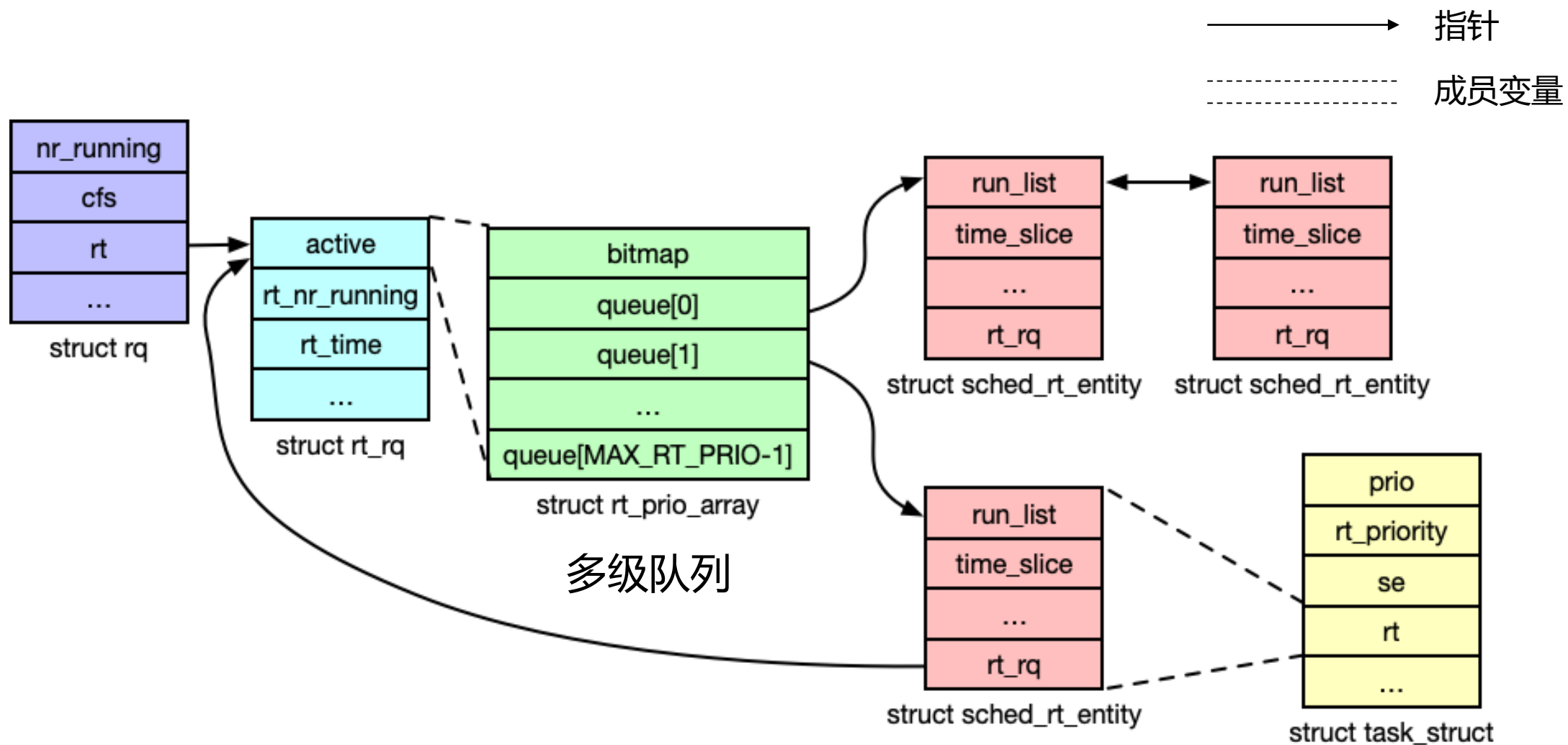
# Linux中的调度策略

- 为了满足不同需求提供多种调度策略
- 以Linux两种调度器为例，每种对应多个调度策略
  - Complete Fair Scheduler(CFS)
    - SCHED\_OTHER
    - SCHED\_BATCH
    - SCHED\_IDLE
  - Real-Time Scheduler(RT)
    - SCHED\_FIFO
    - SCHED\_RR

# Linux调度机制：CFS Run Queue



# Linux调度机制：RT Run Queue





### 普通进程调度：

- 采用SCHED\_NORMAL调度策略。
- 分配优先级、挑选进程并允许、计算使其运行多久。
- CPU运行时间与友好值（-20~+19）有关，数值越低优先级越高。



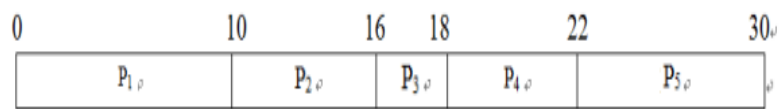
### 实时进程调度：

- 实时调度的进程比普通进程具有更高的优先级。
- SCHED\_FIFO：进程若处于可执行的状态，就会一直执行，直到它自己被阻塞或者主动放弃CPU。
- SCHED\_RR：与SCHED\_FIFO大致相同，只是进程在耗尽其时间片后，不能再执行，而是需要接受CPU的调度。

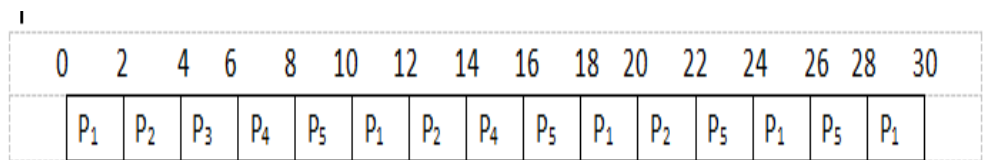
5个进程 $P_1$ 、 $P_2$ 、 $P_3$ 、 $P_4$ 、 $P_5$ 几乎同时到达，预期运行时间分别为10、6、2、4、8个时间单位。各进程的优先级分别为3、5、2、1、4（数值越大，优先级越高）。请按下列调度算法计算任务的平均周转时间（进程切换开销可忽略不计）。

- （1）先来先服务（按 $P_1$ 、 $P_2$ 、 $P_3$ 、 $P_4$ 、 $P_5$ 顺序）算法。
- （2）时间片轮转算法，假定时间片大小为2个时间单位。
- （3）优先权调度算法。

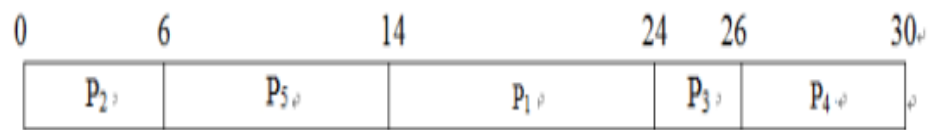
(1) FCFS调度顺序如图所示。



(2) 时间片轮转调度顺序如图所示。










(3) 优先权调度算法的调度顺序如图所示。



算法	时间类型	P <sub>1</sub>	P <sub>2</sub>	P <sub>3</sub>	P <sub>4</sub>	P <sub>5</sub>	平均
	运行时间	10	6	2	4	8	
FCFS	周转时间	10	16	18	22	30	19.2
	带权周转时间	1	2.67	9	5.5	3.75	4.384
RR	周转时间	30	22	6	16	28	20.4
	带权周转时间	3	3.67	3	4	3.5	3.434
优先权	周转时间	24	6	26	30	14	20
	带权周转时间	2.4	1	13	7.5	1.75	5.13



## 内容导航:

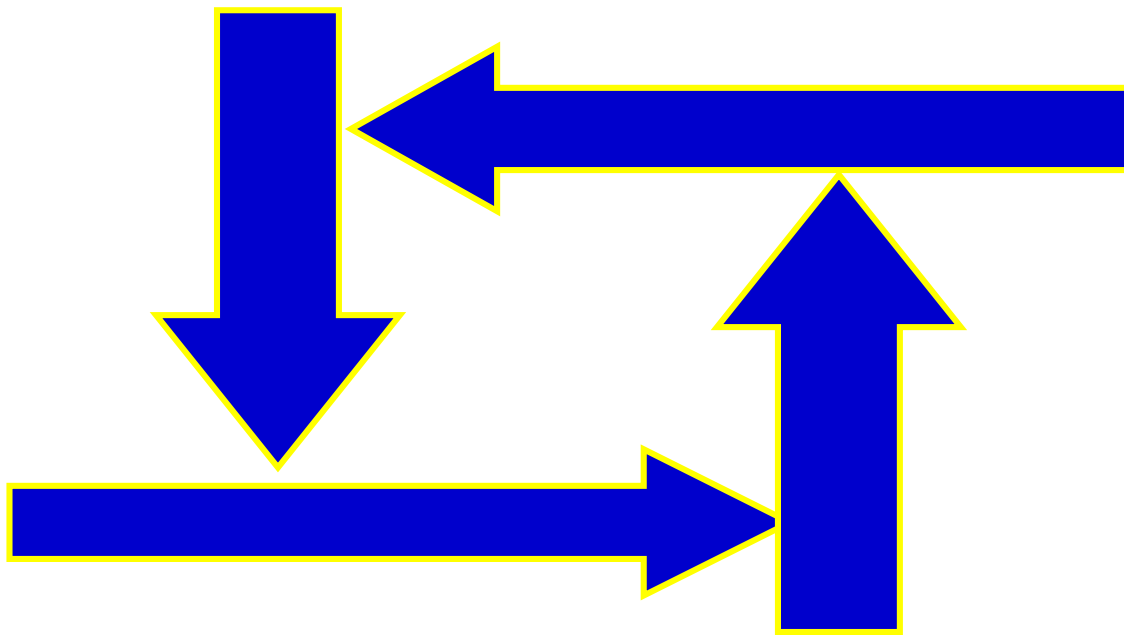
-  3.1 处理机调度概述
-  3.2 调度算法
-  3.3 实时调度
-  3.4 Linux进程调度
-  **3.5 死锁概述**
-  3.6 预防死锁
-  3.7 避免死锁
-  3.8 死锁的检测与解除

## 第3章 处理机调度与死锁

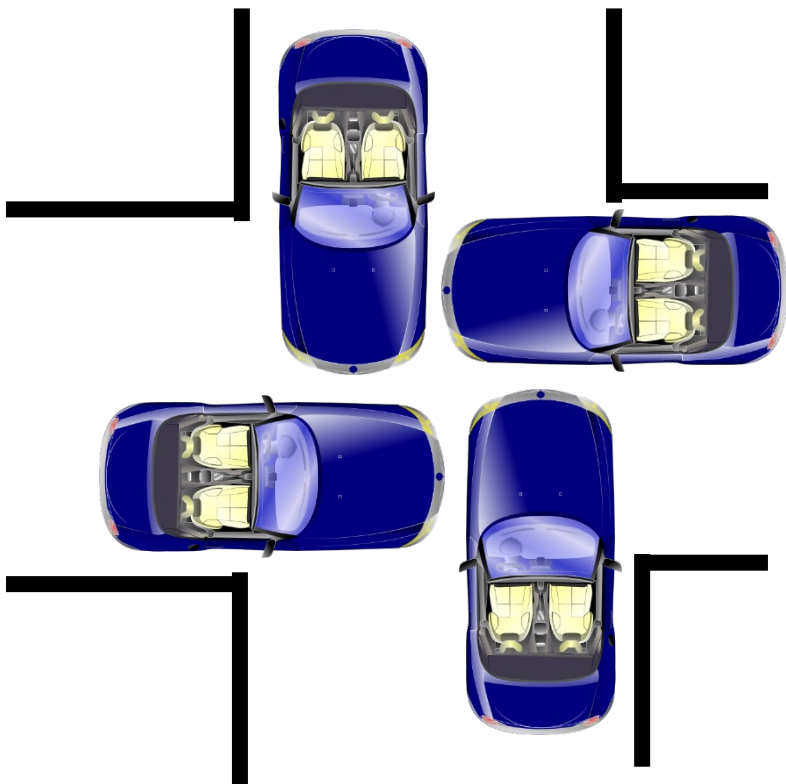
---



**死锁** (Deadlock)：指多个进程在运行过程中因争夺资源而造成的一种僵局，当进程处于这种僵持状态时，若无外力作用，这些进程都将永远不能再向前推进。



# 死锁



十字路口的“困境”

```
void proc_A(void) {  
    lock(A);  
    /* Time T1 */  
    lock(B);  
    /* Critical Section */  
    unlock(B);  
    unlock(A);  
}  
  
void proc_B(void) {  
    lock(B);  
    /* Time T1 */  
    lock(A);  
    /* Critical Section */  
    unlock(A);  
    unlock(B);  
}
```

T1时刻的死锁



## 可重用性资源和可消耗性资源

- 可重用性资源：一次只能分配给一个进程，不允许多个进程共享，遵循：请求资源□ 使用资源□ 释放资源（大部分资源）。
  - ✓ 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。
  - ✓ 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。
  - ✓ 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。
- 可消耗性资源：由进程动态创建和消耗（进程间通信的消息）。



## 可抢占性和不可抢占性资源

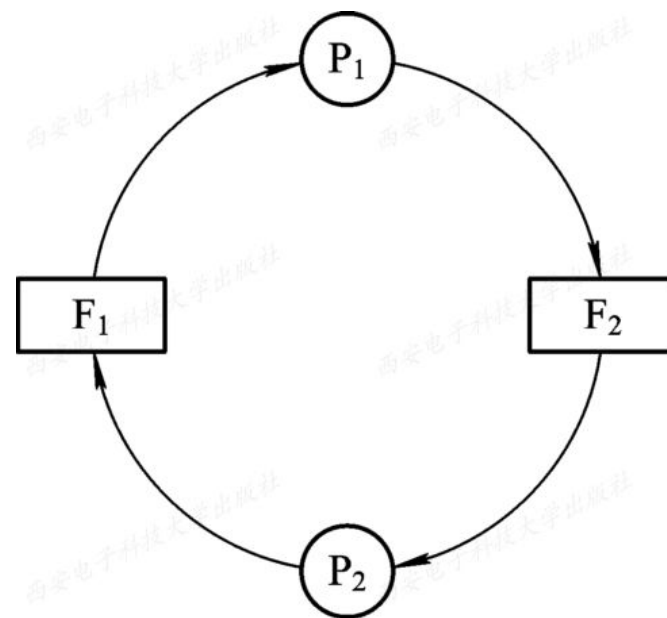
- 可抢占性资源：某进程在获得这类资源后，该资源可以再被其他进程或系统抢占，**CPU（处理机）和主存区**。
- 不可抢占资源：当系统把这类资源分配给某进程后，再不能强行收回，只能在进程用完后自行释放，**打印机、磁带机**。

1

## 竞争不可抢占性资源引起死锁

- 系统中的不可抢占性资源，由于它们的数量不能满足诸进程运行的需要，会使进程在运行过程中，因争夺这些资源而陷入僵局。

$P_1$	$P_2$
.....	.....
open ( $F_1, w$ );	open ( $F_2, w$ );
open ( $F_2, w$ );	open ( $F_1, w$ );

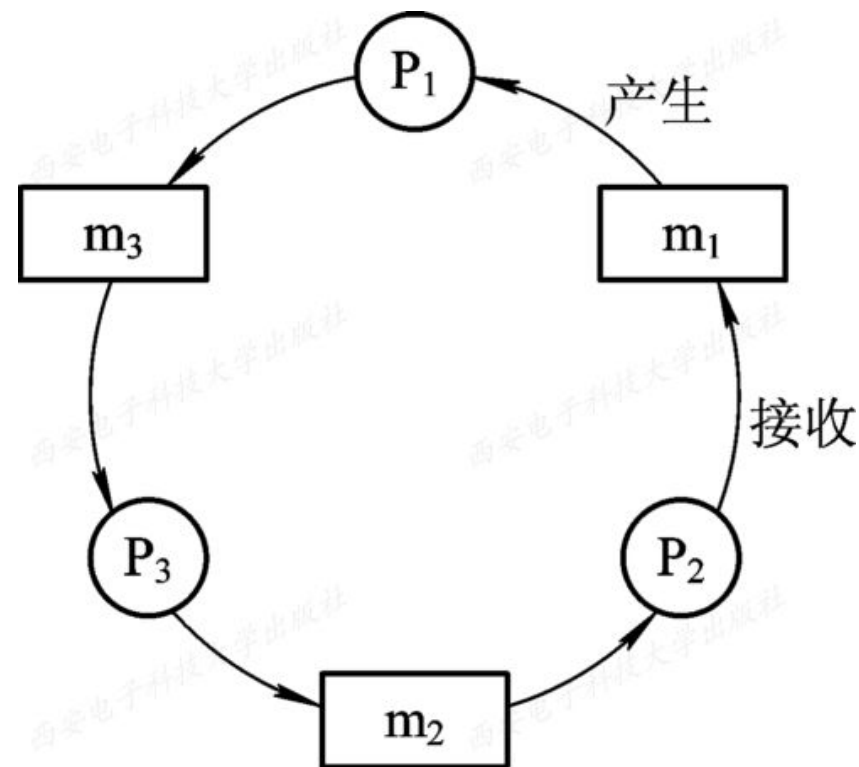


共享文件时的死锁情况

## 2 竞争可消耗性资源引起死锁

```
P1 : ...send( P2, m1 );   receive( P3, m3 ); ...  
P2 : ...send( P3, m2 );   receive( P1, m1 ); ...  
P3 : ...send( P1, m3 );   receive( P2, m2 ); ...
```

```
P1 : ...receive( P3, m3 );   send( P2, m1 ); ...  
P2 : ...receive( P1, m1 );   send( P3, m2 ); ...  
P3 : ...receive( P2, m2 );   send( P1, m3 ); ...
```



进程之间通信时的死锁

## 3

## 进程推进顺序不当引起死锁

## ➤ 进程推进顺序合法

在进程 $P_1$ 和 $P_2$ 并发执行时，如果按图曲线①所示的顺序推进：

$P_1$ : Request( $R_1$ ) $\rightarrow P_1$ :

Request( $R_2$ ) $\rightarrow P_1$ :

Release( $R_1$ ) $\rightarrow P_1$ :

Release( $R_2$ ) $\rightarrow P_2$ :

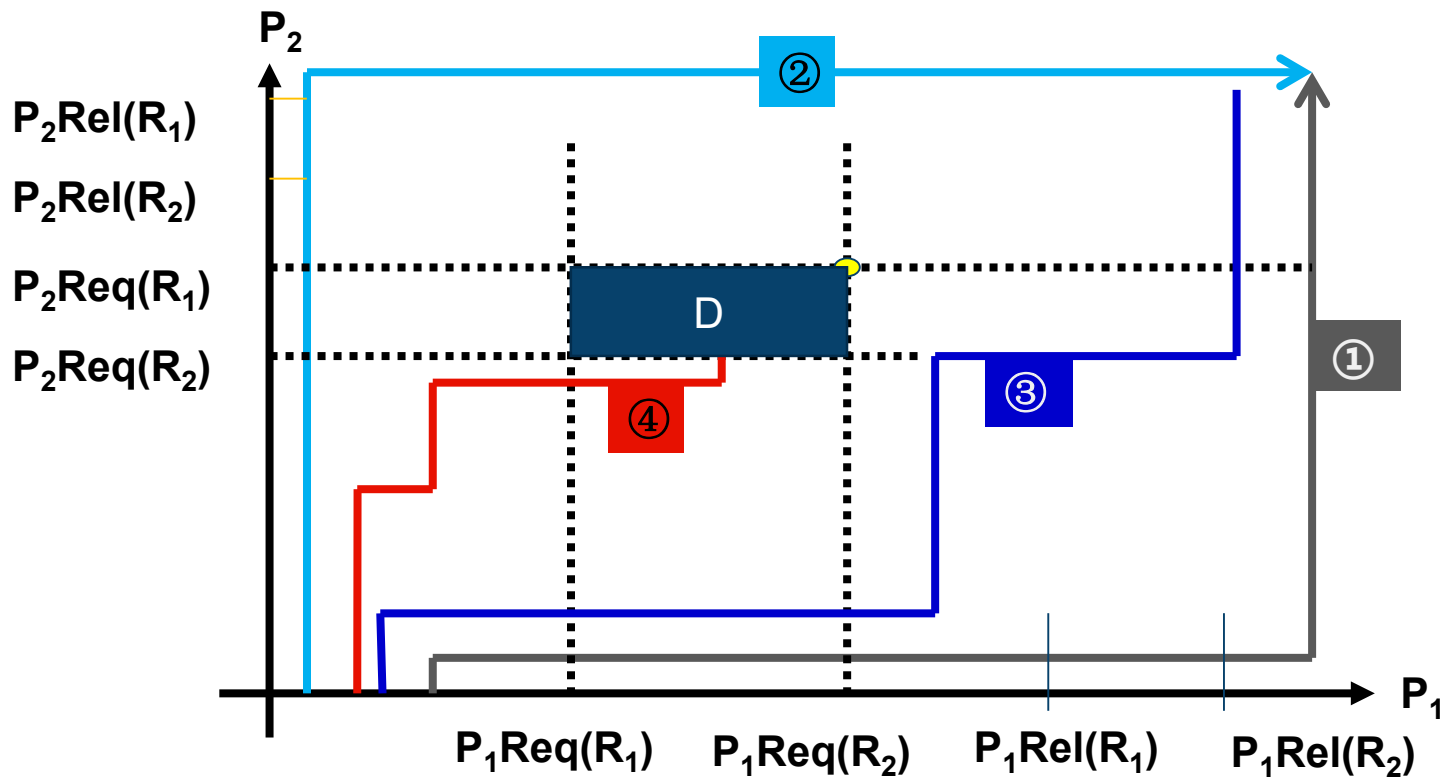
Request( $R_2$ ) $\rightarrow P_2$ :

Request( $R_1$ ) $\rightarrow P_2$ :

Release( $R_2$ ) $\rightarrow P_2$ :

Release( $R_1$ ), 两个进程可顺利完成。

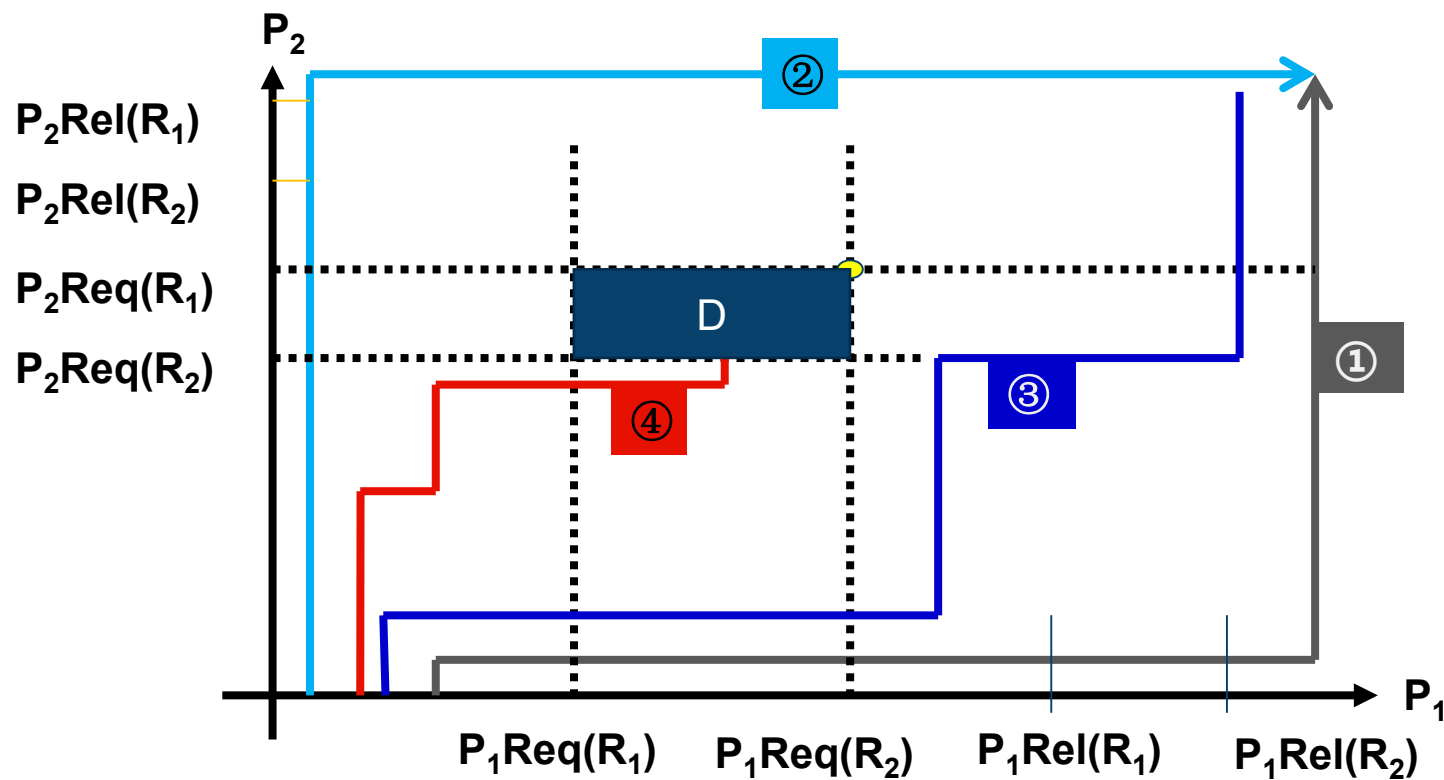
类似地，若按图中曲线②和③所示的顺序推进，两进程也可以顺利完成。



### 3 进程推进顺序不当引起死锁

#### ➤ 进程推进顺序非法

若并发进程 $P_1$ 和 $P_2$ 按图中曲线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，

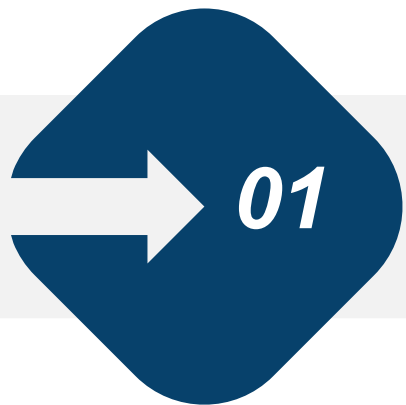






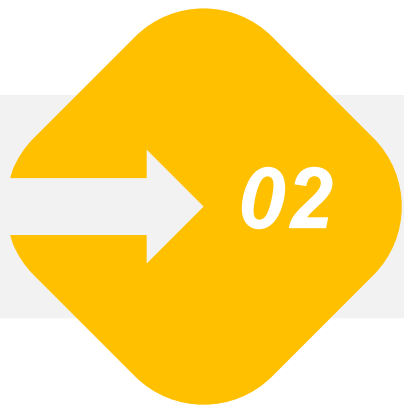
**死锁：**一组等待的进程，其中每一个进程都持有资源，并且等待着由这个组中其他进程所持有的资源。

## 互斥



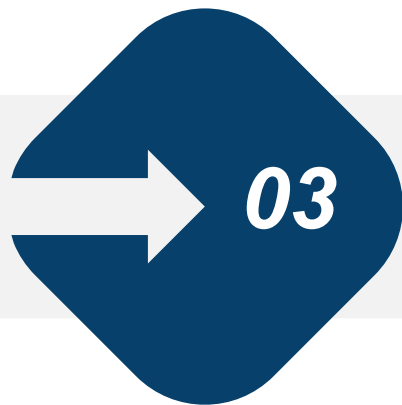
- 一段时间内某资源只能被一个进程占用。

## 请求和保持



- 一个至少持有一个资源的进程等待获得额外的由其他进程所持有的资源。

## 不可抢占



- 一个资源只有当持有它的进程完成任务后,自由的释放。

## 循环等待



- 等待资源的进程之间存在环  $\{P_0, P_1, \dots, P_n\}$ 。
- $P_0$  等待  $P_1$  占有的资源,  $P_1$  等待  $P_2$  占有的资源, ...,  $P_{n-1}$  等待  $P_n$  占有的资源,  $P_0$  等待  $P_n$  占有的资源



确保系统永远不会进入死锁状态

➤ 死锁预防

➤ 死锁避免



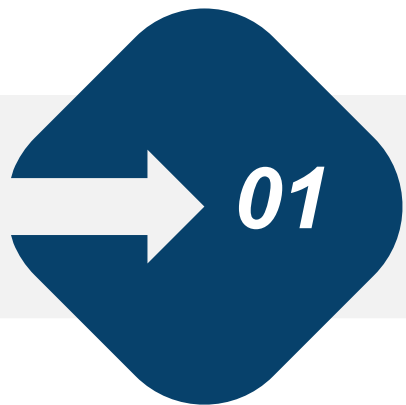
允许系统进入死锁状态，然后恢复系统

➤ 死锁检测    ➤ 死锁恢复



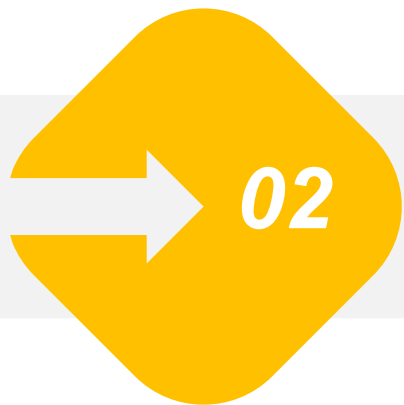
忽略这个问题，假装系统中从未出现过死锁。这个方法被大部分的操作系统采用，包括UNIX

## 预防死锁



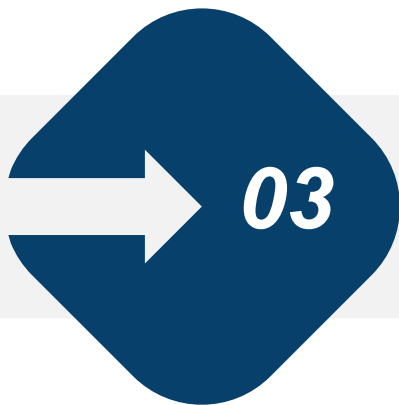
- 破坏死锁的四个必要条件中一个或几个。

## 避免死锁



- 在资源动态分配时，防止系统进入不安全状态。

## 检测死锁



- 事先不采取任何措施，允许死锁发生，但及时检测死锁发生。









## 解除死锁



- 检测到死锁发生时，采取相应措施，将进程从死锁状态中解脱出来。



## 内容导航:

-  3.1 处理机调度概述
-  3.2 调度算法
-  3.3 实时调度
-  3.4 Linux进程调度
-  3.5 死锁概述
-  **3.6 预防死锁**
-  3.7 避免死锁
-  3.8 死锁的检测与解除

## 第3章 处理机调度与死锁

---

破坏死锁的四个必要条件中的一个或几个



**互斥：**互斥条件是共享资源必须的，不仅不能改变，还应加以保证



**请求和保持：**必须保证进程申请资源的时候没有占有其他资源

- 要求进程在执行前一次性申请全部的资源，只有没有占有资源时才可以分配资源，缺点：资源利用率低，可能出现饥饿
- 改进：进程只获得运行初期所需的资源后，便开始运行；其后在运行过程中逐步释放已分配的且用毕的全部资源，然后再请求新资源



## 非抢占:

- 如果一个进程的申请没有实现，它要释放所有占有的资源；
- 先占的资源放入进程等待资源列表中；
- 进程在重新得到旧的资源的时候可以重新开始。（实现复杂，代价大）










## 循环等待：对所有的资源类型排序进行线性排序，并赋予不同的序号，要求进程按照递增顺序申请资源。（可防止出现环路）

缺点：

- 如何规定每种资源的序号是十分重要的；
- 限制新类型设备的增加；
- 作业使用资源的顺序与系统规定的顺序不同；
- 限制用户简单、自主的编程。



## 内容导航:

-  3.1 处理机调度概述
-  3.2 调度算法
-  3.3 实时调度
-  3.4 Linux进程调度
-  3.5 死锁概述
-  3.6 预防死锁
-  **3.7 避免死锁**
-  3.8 死锁的检测与解除

## 第3章 处理机调度与死锁

---





设一个简单而有效的模型，要求每一个进程声明它所需要的资源的最大数。



**死锁避免算法**动态检查资源分配状态以确保不会出现循环等待的情况。



资源分配状态定义为可用的与已分配的资源数，和进程所需的最大资源量。

- 当进程申请一个有效的资源的时候，系统必须确定分配后是安全的。
- 如果存在一个安全序列，则系统处于安全态。
- 进程序列 $\langle P_1, P_2, \dots, P_n \rangle$ 是安全的，如果每一个进程 $P_i$ 所申请的可以被满足的资源数加上其他进程所持有的该资源数小于系统总数。
  - 如果  $P_i$  需要的资源不能马上获得，那么 $P_i$  等待直到所有的 $P_{i-1}$ 进程结束。
  - 当 $P_{i-1}$  结束后，  $P_i$ 获得所需的资源，执行、返回资源、结束。
  - 当 $P_i$ 结束后，  $P_{i+1}$ 获得所需的资源执行，依此类推。

## 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	

存在安全序列 ( $P_2$  ,  $P_1$  ,  $P_3$ )

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。



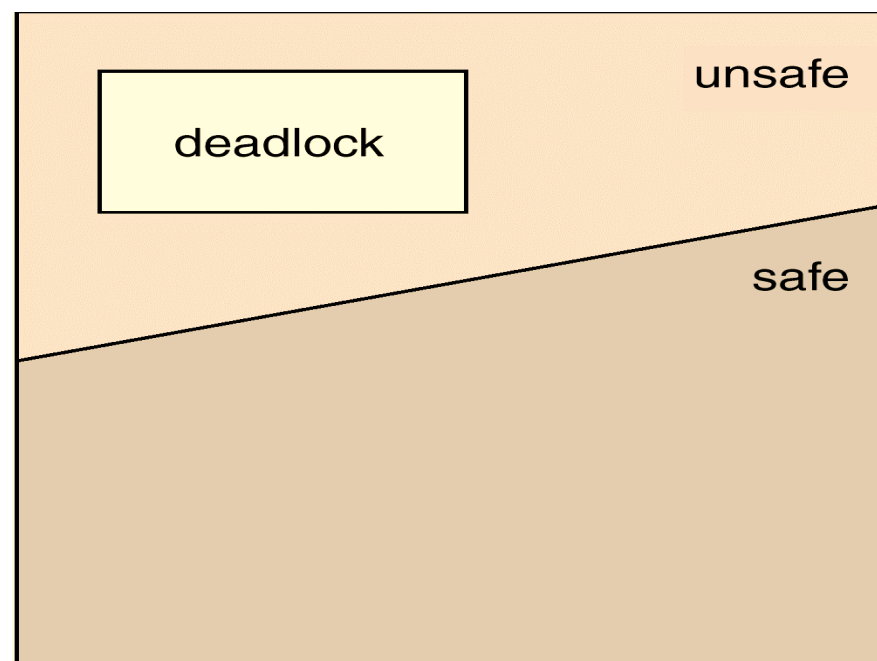
如果一个系统在安全状态，就没有死锁

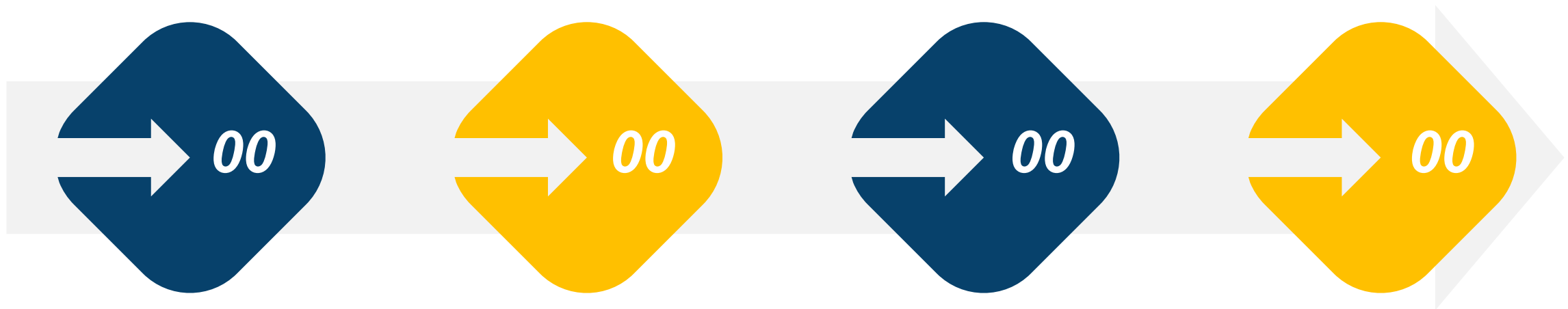


如果一个系统不是处于安全状态，就有可能死锁



死锁避免  $\Rightarrow$  确保系统永远不会进入不安全状态





➤ 针对资源有多个实例

➤ 每一个进程必须事先声明使用的最大量

➤ 当一个进程请求资源，它可能要等待

➤ 当一个进程得到所有的资源，它必须在有限的时间释放它们

由  $m \times n$  个数  $a_{ij}$  排成的  $m$  行  $n$  列的数表称为  $m$  行  $n$  列的矩阵，简称  $m \times n$  矩阵。记作：

$$\mathbf{A} = \begin{bmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ a_{21} & a_{22} & \cdots & a_{2n} \\ a_{31} & a_{32} & \cdots & a_{3n} \\ \cdots & \cdots & & \cdots \\ a_{m1} & a_{m2} & \cdots & a_{mn} \end{bmatrix}$$

这  $m \times n$  个数称为矩阵  $\mathbf{A}$  的元素，简称为元，数  $a_{ij}$  位于矩阵  $\mathbf{A}$  的第  $i$  行第  $j$  列，称为矩阵  $\mathbf{A}$  的  $(i,j)$  元，以数  $a_{ij}$  为  $(i,j)$  元的矩阵可记为  $(a_{ij})$  或  $(a_{ij})_{m \times n}$ ， $m \times n$  矩阵  $\mathbf{A}$  也记作  $\mathbf{A}_{mn}$ 。

**n为进程的数目， m为资源类型的数目**

- OS Available: 长度为 m 的向量。如果  $available[j]=k$ , 那么资源  $R_j$  有 k 个实例有效
  - OS Max:  $n \times m$  矩阵。 如果  $Max[i, j]=k$ , 那么进程  $P_i$  可以最多请求资源  $R_j$  的 k 个实例
  - OS Allocation:  $n \times m$  矩阵。 如果  $Allocation[i, j]=k$ , 那么进程  $P_i$  当前分配了 k 个资源  $R_j$  的实例
  - OS Need:  $n \times m$  矩阵。 如果  $Need[i, j]=k$ , 那么进程  $P_i$  还需要 k 个资源  $R_j$  的实例
- $$Need[i, j] = Max[i, j] - Allocation[i, j].$$

01

(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，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 $Finish[i] := false$ 。

02

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  $Need[i, j] \leq Work[j]$ ;

若找到，执行步骤(3)，否则，执行步骤(4)。

03

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;

04

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。



$\text{Request}_i$  = 进程  $P_i$  的资源请求向量, 如果  $\text{Request}_i[m] = k$  则进程  $P_i$  想要资源类型为  $R_m$  的  $k$  个实例

- ① 如果  $\text{Request}_i[j] \leq \text{Need}[i, j]$  转 step 2. 否则报错, 因为进程请求超出了其声明的最大值
- ② 如果  $\text{Request}_i[j] \leq \text{Available}[j]$ , 转 step 3. 否则  $P_i$  必须等待, 因为资源不可用
- ③ 假设通过修改下列状态来试着分配请求的资源给进程  $P_i$  :
  - $\text{Available}[j] = \text{Available}[j] - \text{Request}_i[j]$ ;    ■  $\text{Need}[i, j] = \text{Need}[i, j] - \text{Request}_i[j]$ ;
  - $\text{Allocation}[i, j] = \text{Allocation}[i, j] + \text{Request}_i[j]$ ;
- ④ 系统执行安全性算法
  - 如果系统安全  $\Rightarrow$  将资源分配给  $P_i$ .
  - 如果系统不安全  $\Rightarrow P_i$  必须等待, 恢复原有的资源分配状态

- ◆ 5个进程 $P_0$ 到 $P_4$ ; 3个资源类型A(10个实例), B (5个实例), C (7个实例)
- ◆ 时刻 $T_0$ 的快照:

	<u>Allocation</u>			<u>Max</u>			<u>Available</u>		
	A	B	C	A	B	C	A	B	C
$P_0$	0	1	0	7	5	3	3	3	2
$P_1$	2	0	0	3	2	2			
$P_2$	3	0	2	9	0	2			
$P_3$	2	1	1	2	2	2			
$P_4$	0	0	2	4	3	3			

- 矩阵的内容。Need被定义为Max – Allocation
- 系统处在安全状态，因为序列 $\langle P_1, P_3, P_4, P_2, P_0 \rangle$  满足了安全标准

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			

进程 \ 资源情况	Work			Need			Allocation			Work+Allocation			Finish
	A	B	C	A	B	C	A	B	C	A	B	C	
P <sub>1</sub>	3	3	2	1	2	2	2	0	0	5	3	2	true
P <sub>3</sub>	5	3	2	0	1	1	2	1	1	7	4	3	true
P <sub>4</sub>	7	4	3	4	3	1	0	0	2	7	4	5	true
P <sub>2</sub>	7	4	5	6	0	0	3	0	2	10	4	7	true
P <sub>0</sub>	10	4	7	7	4	3	0	1	0	10	5	7	true



检查  $\text{Request} \leq \text{Available}$ , 就是说,  $(1,0,2) \leq (3,3,2)$  为真

	Allocation			Need			Available		
	A	B	C	A	B	C	A	B	C
$P_0$	0	1	0	7	4	3	2	3	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			

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 <sub>1</sub>	2	3	0	0	2	0	3	0	2	5	3	2	true
P <sub>3</sub>	5	3	2	0	1	1	2	1	1	7	4	3	true
P <sub>4</sub>	7	4	3	4	3	1	0	0	2	7	4	5	true
P <sub>0</sub>	7	4	5	7	4	3	0	1	0	7	5	5	true
P <sub>2</sub>	7	5	5	6	0	0	3	0	2	10	5	7	true

执行安全算法表明序列  $\langle P_1, P_3, P_4, P_0, P_2 \rangle$  满足要求

补充思考:

➤ P<sub>4</sub>的请求(3,3,0)是否可以通过?

➤ P<sub>0</sub>的请求(0,2,0)是否可以通过?

(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。



资源情况 进 程		Allocation			Need			Available		
		A	B	C	A	B	C	A	B	C
P <sub>0</sub>		0	3	0	7	2	3	2	1	0
P <sub>1</sub>		3	0	2	0	2	0			
P <sub>2</sub>		3	0	2	6	0	0			
P <sub>3</sub>		2	1	1	0	1	1			
P <sub>4</sub>		0	0	2	4	3	1			

图3-20 为P<sub>0</sub>分配资源后的有关资源数据








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



## 系统模型

-  资源类型  $R_1, R_2, \dots, R_m$ 
  - CPU周期, 内存空间, I/O设备
-  每一种资源  $R_i$  有  $W_i$  种实例
-  每一个进程通过如下方法来使用资源
  - 申请      ➤ 使用      ➤ 释放

一个顶点的集合V和边的集合E



V被分为两个部分

- $P = \{P_1, P_2, \dots, P_n\}$ , 含有系统中全部的进程
- $R = \{R_1, R_2, \dots, R_m\}$ , 含有系统中全部的资源

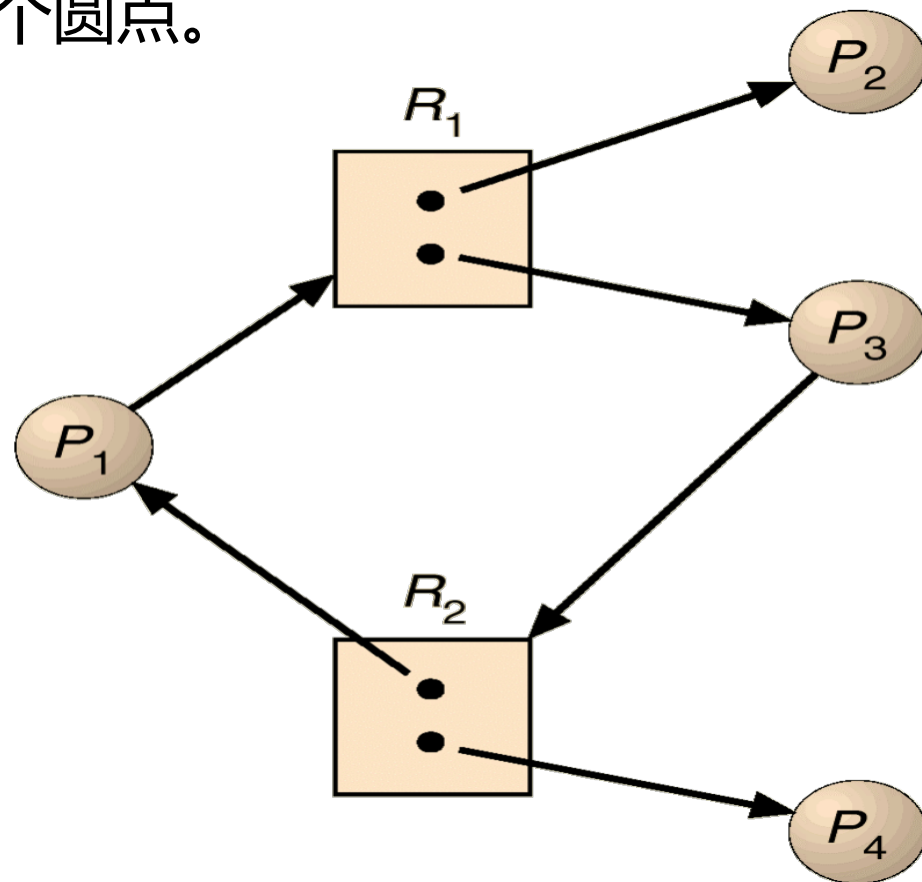
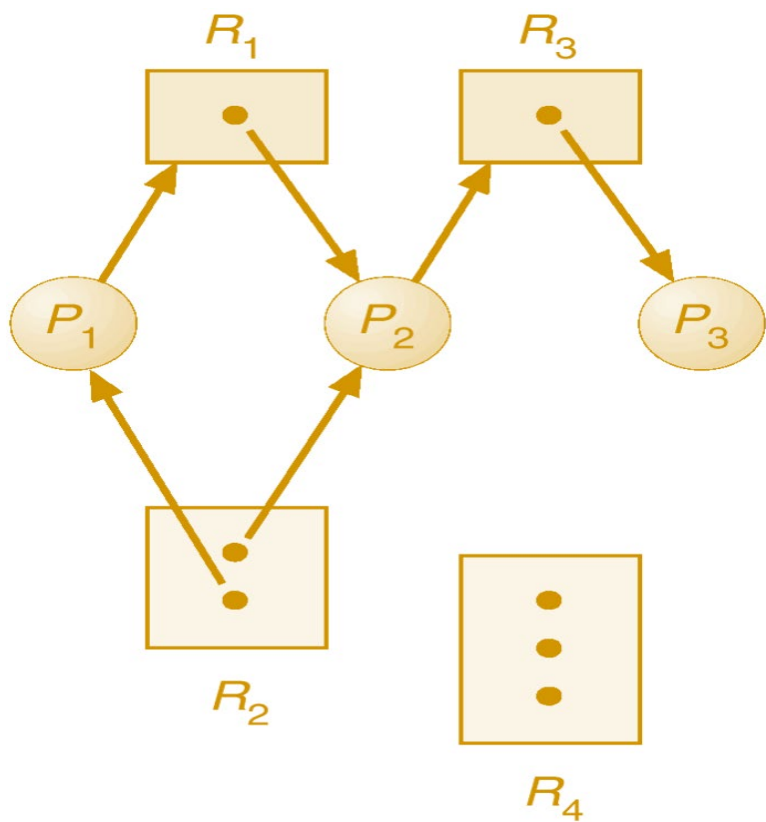


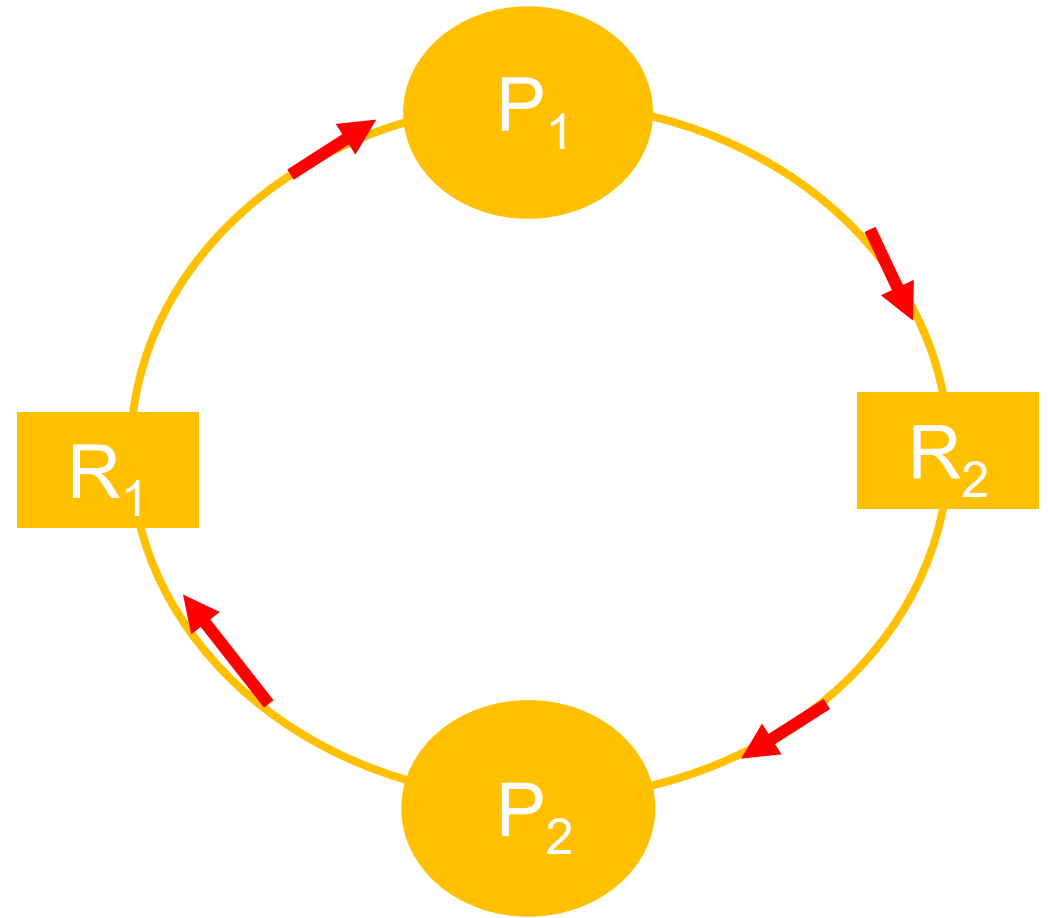
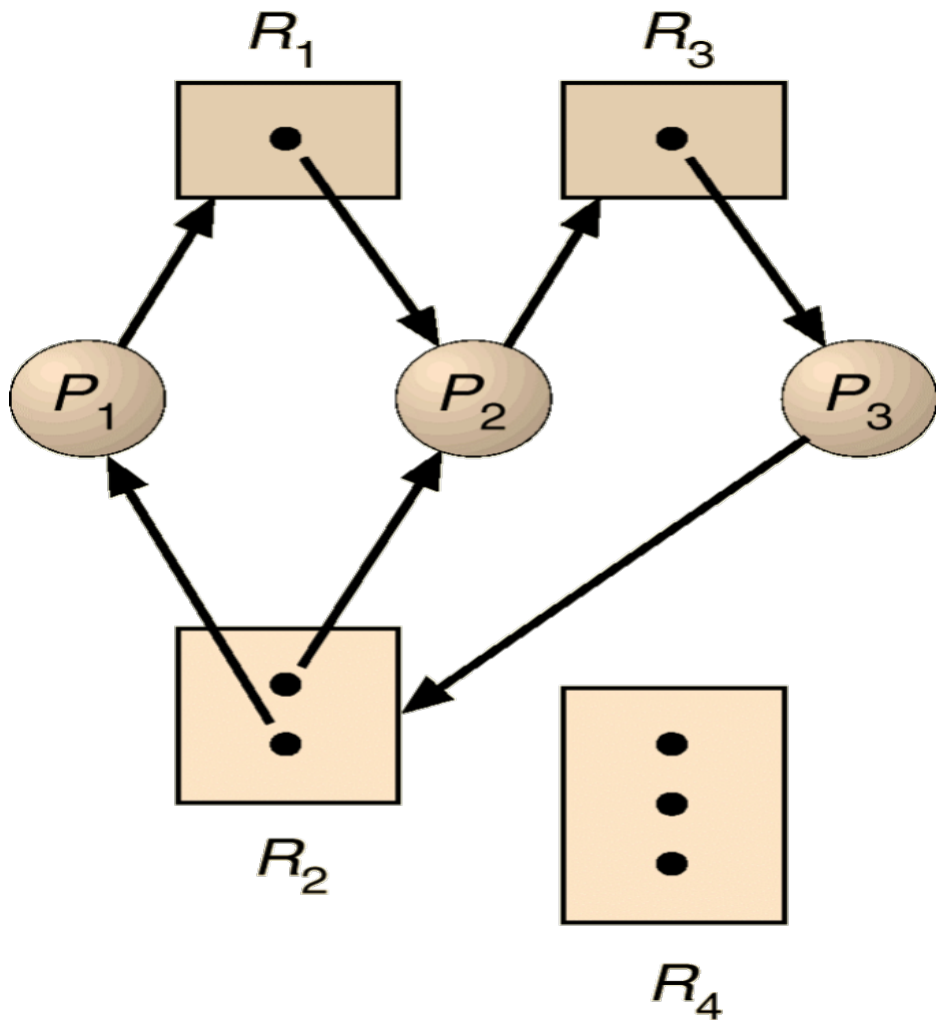
请求边: 有向边  $P_i \rightarrow R_j$



分配边: 有向边  $R_j \rightarrow P_i$

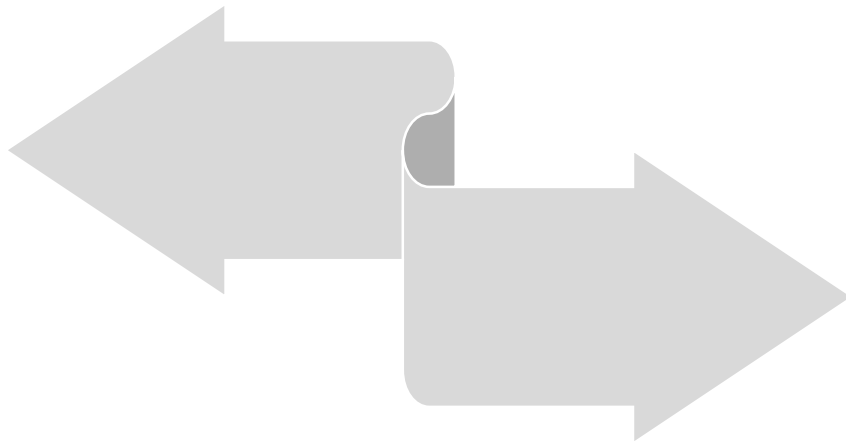
用圆圈代表一个进程，用方框代表一类资源。由于一类资源可以包含多个资源实例，我们用方框中的一个圆点来代表一类资源中的一个资源实例。此时，请求边由进程指向方框中的R，而分配边则始于方框中的一个圆点。







如果图没有环，那  
么不会有死锁！











如果图有环，那么：

- 如果每一种资源类型只有一个实例，那么死锁发生；
- 如果一种资源类型有多个实例，那么可能死锁。



## 内容导航:

-  3.1 处理机调度概述
-  3.2 调度算法
-  3.3 实时调度
-  3.4 Linux进程调度
-  3.5 死锁概述
-  3.6 预防死锁
-  3.7 避免死锁
-  **3.8 死锁的检测与解除**

## 第3章 处理机调度与死锁

---

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：**

01

保存有关资源的请求和分配信息；

02

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。



01

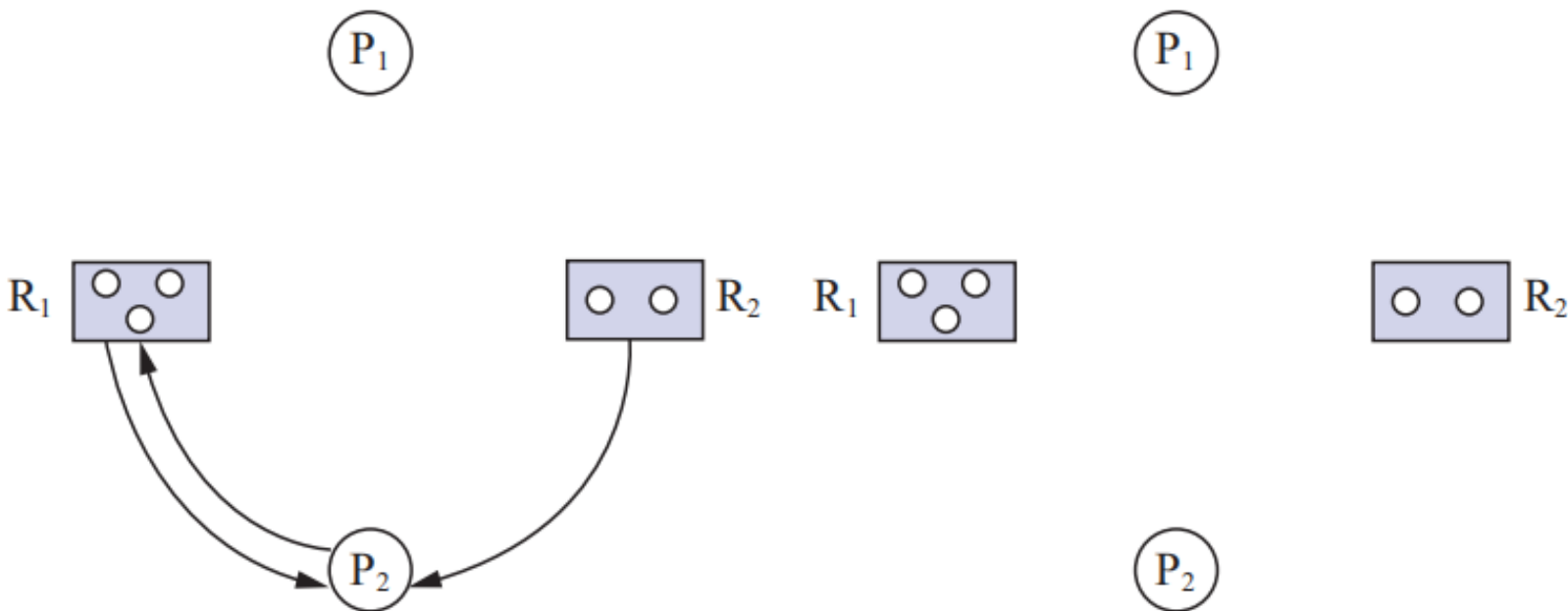
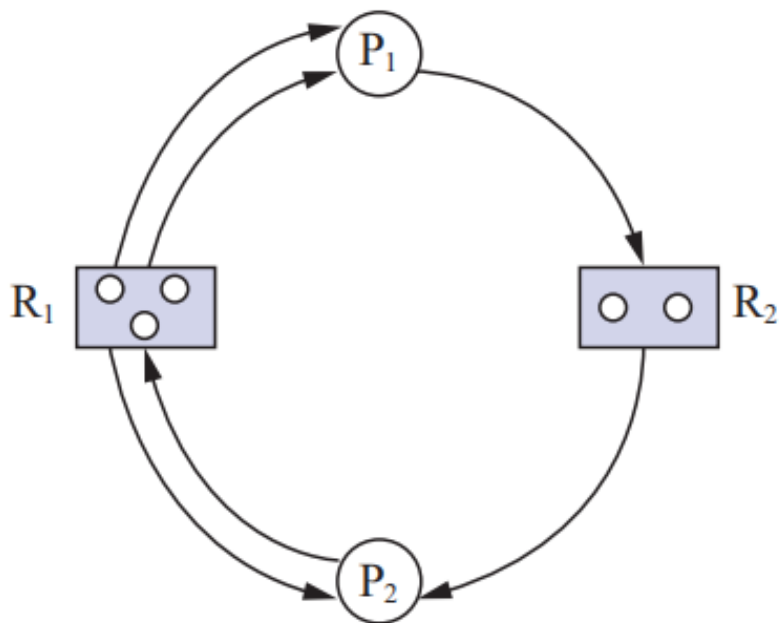
在资源分配图中，**找出一个既不阻塞又非独立的进程结点 $p_i$** 。在顺利的情况下， $p_i$ 可获得所需资源而继续运行，直至运行完毕，再释放其所占有得全部资源，这相当于消去 $p_i$ 所有的请求边和分配边，使之成为孤立的结点；

02

**$p_1$ 释放资源后，便可使 $p_2$ 获得资源而继续运行**，直到 $p_2$ 完成后又释放出它所占有的全部资源；

03

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





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



S为死锁状态的充分条件是：当且仅当S状态的资源分配图是不可完全简化的。该充分条件称为死锁定理。

数据结构类似于银行家算法 (基于资源分配图简化)

Work: = Available;

$L := \{L_i \mid \text{Allocation}_i = 0 \cap \text{Request}_i = 0\}$

for all  $L_i \notin L$  do

begin

for all  $\text{Request}_i \leq \text{Work}$  do

begin

Work: = Work +  $\text{Allocation}_i$ ;

$L_i \cup L$ ;

end

end

deadlock: =  $(L = \{P_1, P_2, \dots, P_n\})$  ;

- (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<sub>i</sub>≤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<sub>i</sub>。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。

01

让Work和Finish作为长度为m和n的向量初始化

(a)  $Work := Available$

(b) For  $i = 1, 2, \dots, n$ , if  $Allocation_i \neq 0$ , then  
     $Finish[i] := false$ ; otherwise,  $Finish[i] := true$ .

02

找到满足下列条件的下标i

(a)  $Finish[i] = false$

(b)  $Request_i \leq Work$

如果没有这样的i存在, 转4

03

$Work := Work + Allocation_i$

$Finish[i] := true$

转 2.

04

如果有一些i,  $1 \leq i \leq n$ ,  
 $Finish[i] = false$ , 则系统处在死  
锁状态。而且, 如果  $Finish[i]$   
 $= false$ , 则进程  $P_i$  是死锁的。



# 检测算法的例子



五个进程 $P_0$ 到 $P_4$ ,三个资源类型A (7个实例) , B (2个实例) , C (6个实例) 。  
时刻 $T_0$ 的状态:

	<u>Allocation</u>			<u>Request</u>			<u>Available</u>		
	A	B	C	A	B	C	A	B	C
$P_0$	0	1	0	0	0	0	0	0	0
$P_1$	2	0	0	2	0	2			
$P_2$	3	0	3	0	0	0			
$P_3$	2	1	1	1	0	0			
$P_4$	0	0	2	0	0	2			



对所有 $i$ , 序列  $\langle P_0, P_2, P_3, P_1, P_4 \rangle$  将导致 $Finish[i] = \text{true}$ , 因此死锁不存在。



P2请求一个额外的C实例

	<u>Request</u>		
	A	B	C
$P_0$	0	0	0
$P_1$	2	0	1
$P_2$	0	0	1
$P_3$	1	0	0
$P_4$	0	0	2



系统的状态?

- 可以归还 $P_0$ 所有的资源，但是资源不够完成其他进程的请求。
- 死锁存在，包括进程 $P_1$ 、 $P_2$ 、 $P_3$ 和 $P_4$ 。

常用解除死锁的两种方法：

01

**抢占资源。**从一个或多个进程中抢占足够数量的资源给死锁进程，以解除死锁状态

02

**终止或撤消进程。**终止系统中一个或多个死锁进程，直到打破循环环路，使死锁状态消除为止。

- 终止所有死锁进程（最简单方法）
- 逐个终止进程（稍温和方法）





中断所有的死锁进程。



一次中断一个进程，直到死锁环消失。



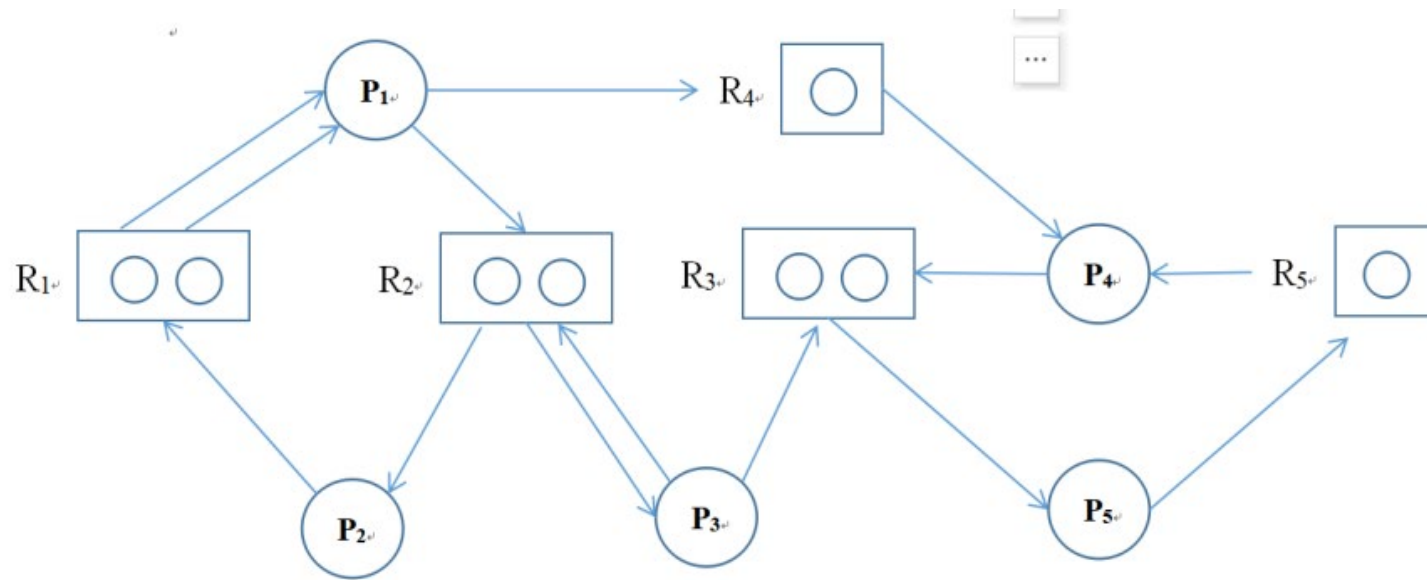
应该选择怎样的中断顺序，使“**代价最小**”？

- 进程的优先级；
- 进程需要计算多长时间，以及需要多长时间结束；
- 进程使用的资源，进程完成还需要多少资源；
- 进程是交互的还是批处理的。

假设系统有5类独占资源： $R_1$ 、 $R_2$ 、 $R_3$ 、 $R_4$ 、 $R_5$ 。各类资源分别有2、2、2、1、1个。系统有5个进程： $P_1$ 、 $P_2$ 、 $P_3$ 、 $P_4$ 、 $P_5$ 。其中 $P_1$ 已占有2个 $R_1$ ，且申请1个 $R_2$ 和1个 $R_4$ ； $P_2$ 已占有1个 $R_2$ ，且申请1个 $R_1$ ； $P_3$ 已占有1个 $R_2$ ，且申请1个 $R_2$ 和1个 $R_3$ ； $P_4$ 已占有1个 $R_4$ 和1个 $R_5$ ，且申请1个 $R_3$ ； $P_5$ 已占有1个 $R_3$ ，且申请1个 $R_5$ 。

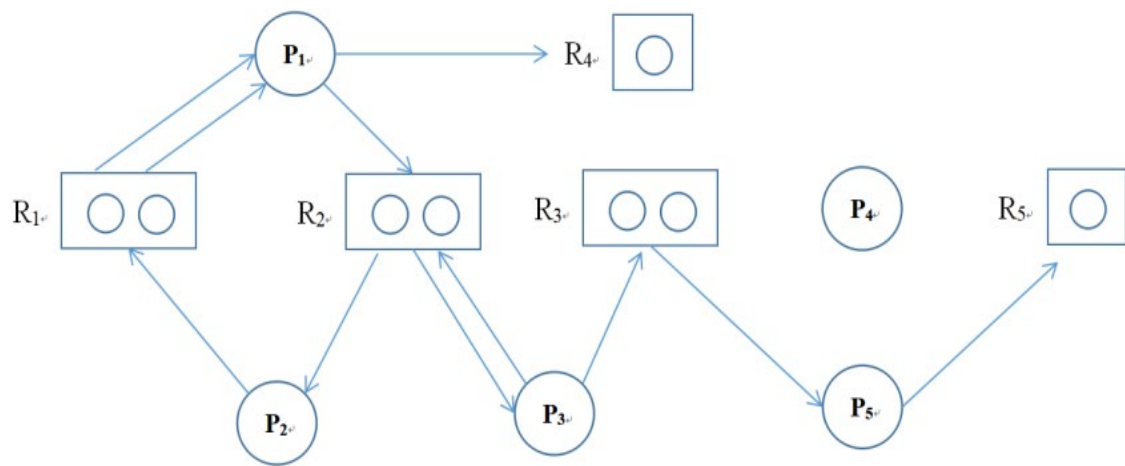
(1) 试画出该时刻的资源分配图。

(2) 什么是死锁定理？如何判断（1）中给出的资源分配图有无死锁？给出判断过程和结果。

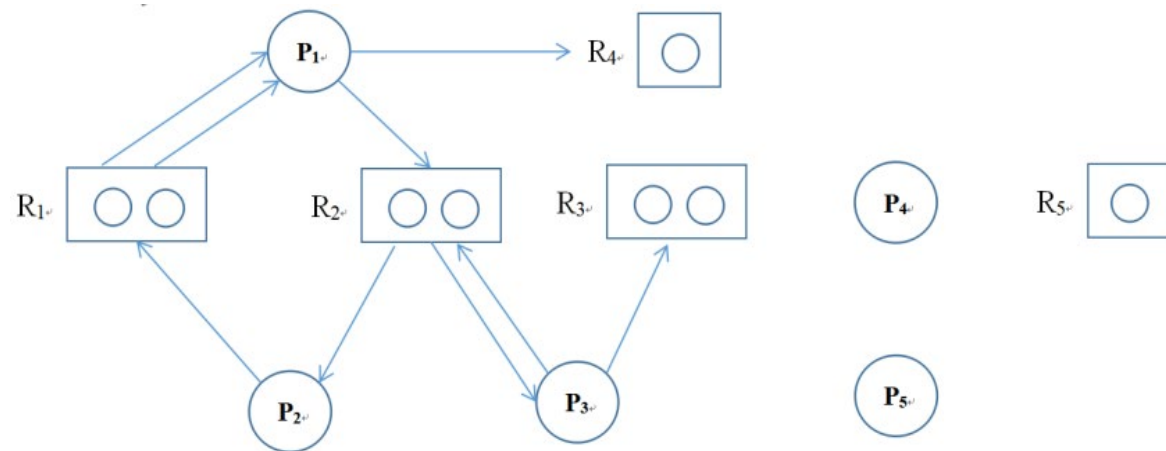


(2) 系统状态S为死锁状态的充分条件：当且仅当S状态的资源分配图是不可完全简化的。  
该充分条件被称为死锁定理。

当前状态下系统可用资源数为  $(0, 0, 1, 0, 0)$ ，可以满足  $P_4$  的申请需求，可将  $P_4$  申请的资源进行分配， $P_4$  执行完毕后，系统的状态如图所示。



$P_4$  释放资源后，系统可用资源数变为  $(0, 0, 1, 1, 1)$ ，可以满足  $P_5$  的申请需求，可将  $P_5$  申请的资源进行分配， $P_5$  执行完毕后，系统的状态如图所示。



$P_5$  释放资源后，系统可用资源数变为  $(0, 0, 2, 1, 1)$ ，已不能满足任何进程的申请需求，系统当前资源分配图已经不能再简化，故系统处于死锁状态。



# 第一次作业

简答题

1	2	3	4	5	6	7	8
9	10	11	12	13	14	15	16
17	18						

计算题  
综合应用题

19	20	21	22
----	----	----	----

23	24	25
----	----	----

标黄色为本次作业



简答题

1	2	3	4	5	6	7	8
9	10	11	12	13	14	15	16
17	18						

计算题  
综合应用题

19	20	21	22
----	----	----	----

23	24	25
----	----	----

标黄色为本次作业



# 经典教材《计算机操作系统》**最新版**



# 学习进步！

**作者：汤小丹、王红玲、姜华、汤子瀛**

## 中科红旗之后，谁将再次勇战操作系统世界霸主？

提及北京中科红旗软件技术有限公司（简称中科红旗），读者可能有所不知，从事计算机信息技术领域具备国产自主知识产权的“红旗Linux系统”研发的中科红旗，曾长期占据国产操作系统市场老大位置。因此对于操作系统领域的相关人员而言，中科红旗绝对是一个“难以忘怀”的名字。

20世纪90年代初期，在微软公司大举进入中国市场并展示傲慢的信息技术（information technology, IT）巨头形象时，中科红旗被它的过分张扬与霸道所激怒，随即燃起了“起来，抵抗微软公司”的民族情绪，并联合北京金山办公软件股份有限公司（简称金山）等民族软件厂商抵抗微软公司的市场垄断行为。之后，红旗Linux系统在国内外渐渐具备一定的影响力，广泛





## 中科红旗之后，谁将再次勇战操作系统世界霸主？

应用于国家信息平台正版化、中国科学院、国家外汇管理局等全国性关键应用领域的国产信息化建设之中。联想、戴尔、惠普等公司的计算机也都曾预装过红旗Linux系统。

不料在种种因素的影响下，2014年2月10日，中科红旗突然宣布解散，引发业界轩然大波，令人感到万分遗憾！实际上，中科红旗的对手——微软公司是一家称霸全球、长期占据操作系统第一位置的企业，其庞大的生态系统，就像一艘巨型游轮，与之较量，实属不易。此外，令中科红旗十分尴尬的事实是，国人似乎已形成一种消费心理定势，宁可将预装的正版Linux系统卸载，也要安装上盗版的Windows系统！



## 中科红旗之后，谁将再次勇战操作系统世界霸主？

但是，这些都不是理由！当下，我们必须重整旗鼓、自主创新，要以蚕食桑叶之精神，与垄断型的操作系统争完善、比实用，待势均力敌之时再与它们抢市场、争地位。金山的WPS办公软件不就如此一路走来，现已可以取代微软公司的Office办公软件了吗？！

因此，无须畏惧现在的微软公司、苹果、谷歌等公司，只要我们坚持创新，机会的大门就会永远为我们敞开。卧薪尝胆，刻苦自励，坚信下一个勇于挑战Windows系统霸主地位的国产操作系统企业，定会浩然而至！