# 计算机操作系统

Operating Systems

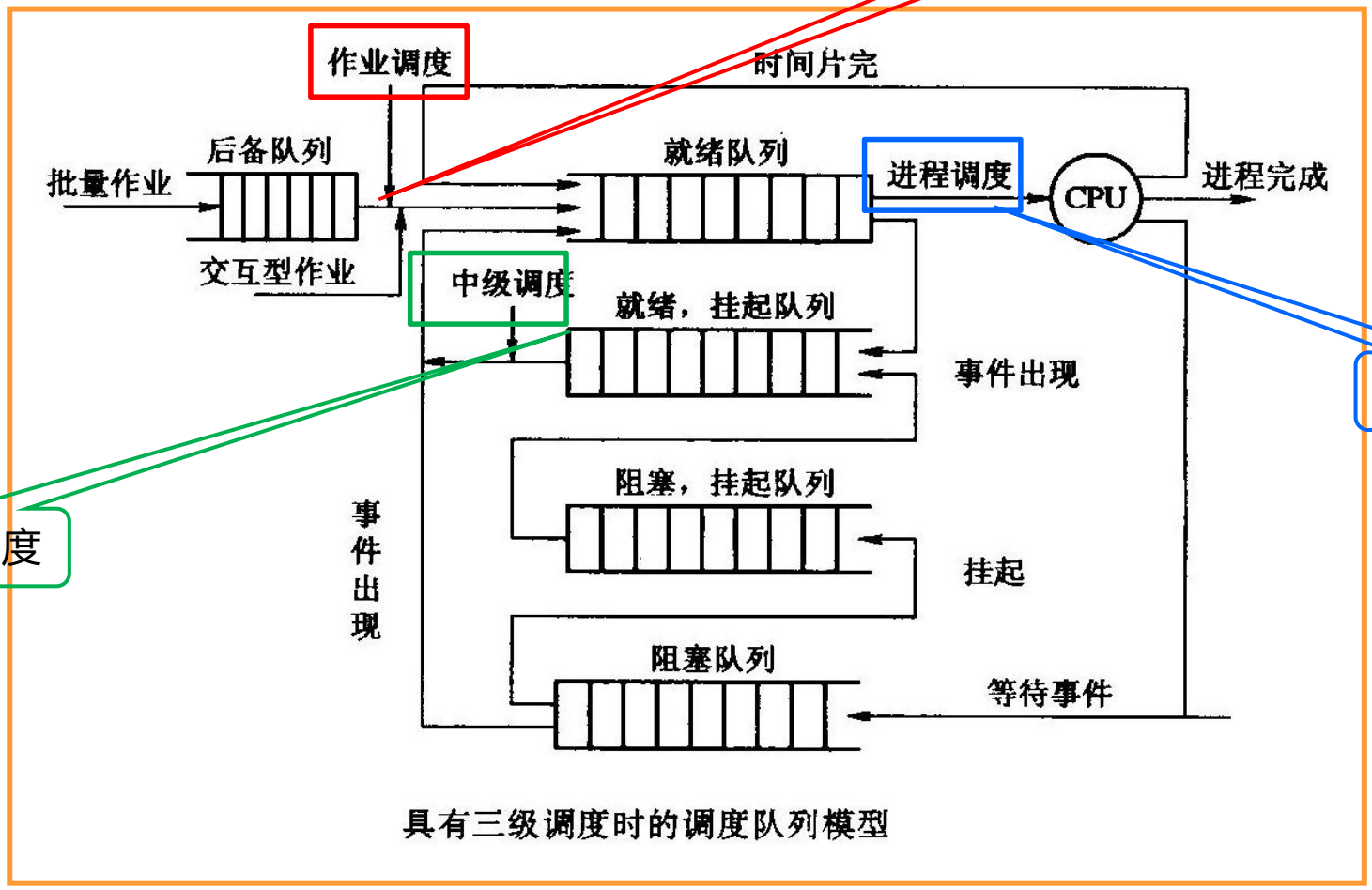
李　琳

# 第三章 处理机调度与死锁

# 3.1 处理机调度的基本概念

## 3.1.1 OS分配处理机的流程

高级调度




具有三级调度时的调度队列模型

低级调度

内外存调度

# 3.1 处理机调度的基本概念

## 3.1.2 高、中、低三级调度

- 高级调度（Long-Term Scheduling）
  - ✓决定把外存上处于后备队列中的哪些作业调入内存，并为它们创建进程、分配必要的资源，排在就绪队列上，准备执行。
  - ✓长程调度、作业调度、接纳调度
  - ✓调度频率低：几分钟或几十分钟
  - ✓调度算法可以复杂

考虑内存与I/O资源

- 低级调度（Low-Level Scheduling）
  - ✓决定就配给该进程的具体操作
  - ✓进程调度、短程调度
  - ✓调度频率高：几毫秒绪队列中的哪个进程应获得处理机，再由分派程序执行把处理机分或几十毫秒
  - ✓调度算法通常简单，保证算法执行时间短

考虑CPU与I/O资源

# 3.1 处理机调度的基本概念

## 3.1.2 高、中、低三级调度

- 中级调度（Intermediate-Level Scheduling）
  - ✓目的：为了提高内存利用率和系统吞吐量
  - ✓通过进程挂起状态实现
  - ✓调度对象：就绪进程、阻塞进程
  - ✓调度频率介于高级调度和低级调度之间
  - ✓实际就是内存管理的“对换”功能

考虑内存资源

- 抢占式与非抢占式
  - ✓非抢占式：即使有更重要的进程进入就绪队列，当前运行进程也不立即放弃CPU，又称非剥夺方式
  - ✓抢占式：如果有更重要的进程进入就绪队列，当前运行进程立刻释放CPU，又称剥夺方式
  - ✓非抢占方式系统开销小，实时性差；抢占霸占CPU，容易造成“进程饥饿”

# 3.1 处理机调度的基本概念

## 3.1.3 调度队列模型

- 仅有进程调度

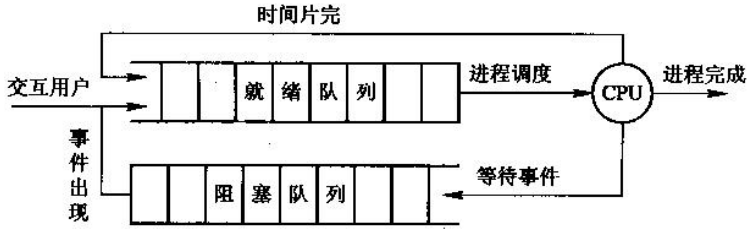



图 3－1　仅具有进程调度的调度队列模型

- 具有高级调度和进程调度

- 高、中、低三级调度模型

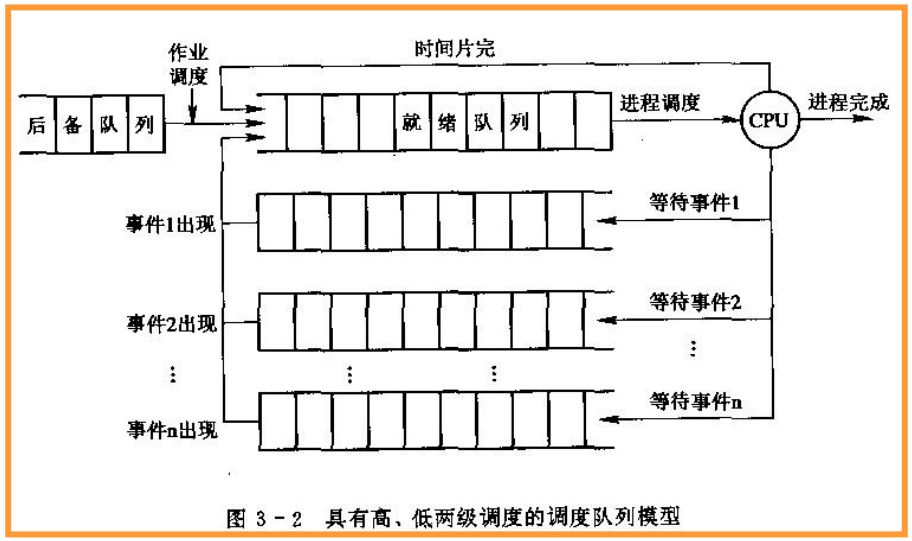
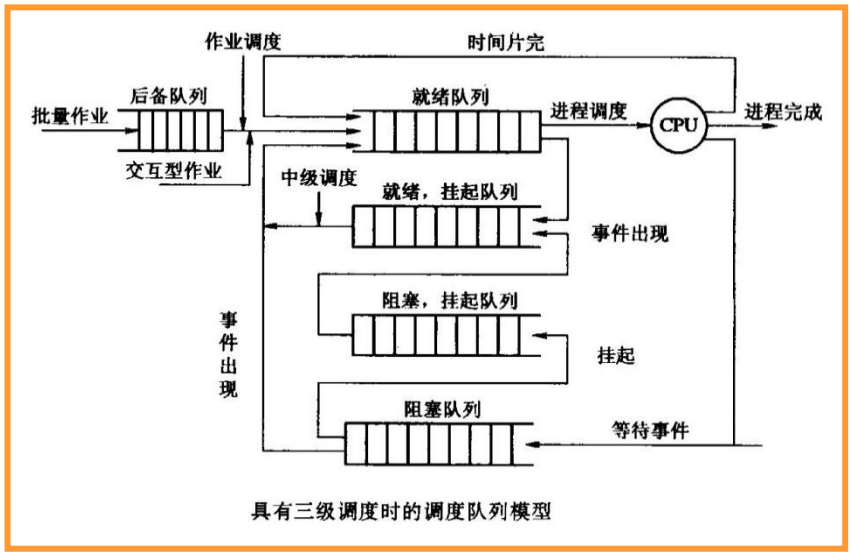



图 3－2　具有高、低两级调度的调度队列模型




具有三级调度时的调度队列模型

# 3.1 处理机调度的基本概念

## 3.1.4 调度算法目标

- 调度算法的共同目标
  - ✓资源利用率
  - ✓公平性、安全性
  - ✓开销低：调度器简单

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

- 批处理系统的目标
  - ✓周转时间：从作业提交给系统开始，到作业完成为止的时间间隔
  - ✓系统吞吐量：单位时间内完成的作业数

$$\text{平均周转时间：}T=\frac{1}{n}\sum_{i=1}^{n}T_i$$

- 分时系统的目标
  - ✓响应时间快
  - ✓均衡性

$$\text{平均带权周转时间：}W=\frac{1}{n}\sum_{i=1}^{n}\frac{T_i}{T_{si}}$$

- 实时系统的目标
  - ✓截止时间的保证
  - ✓可预测性

思考：满足用户需求？满足系统需求？

# 3.1 处理机调度的基本概念

## 3.1.5 调度器的权衡

- 调度开销与调度效果
  - ✓一个合理的调度决策可能需要足够多的计算
- 优先级与公平性
  - ✓高优先级与低优先级任务的公平
- 性能与能耗
  - ✓维持CPU高速运转会导致能耗加大

# 试一试

（1）进程调度的准则不能是（　）

A 最大的内存利用率　　B 最短的周转时间

C 最大的CPU利用率　　D 最短的等待时间

(2) 以下有关短程调度和长程调度的论述，正确的是（　）

A 短程调度比长程调度开销大

B 短程调度比长程调度开销小

C 短程调度比长程调度切换频率低

D 短程调度比长程调度切换频率高

# 3.2 调度算法

- 实质是一种资源分配方法
- 主要使用三种策略，不同的策略可单独使用，也可叠加使用形成一种算法。
- 对于不同的系统和系统目标，通常采用不同的调度算法；
- 有的算法适用于作业调度，有的算法适用于进程调度；但也有些调度算法既可用于作业调度，也可用于进程调度

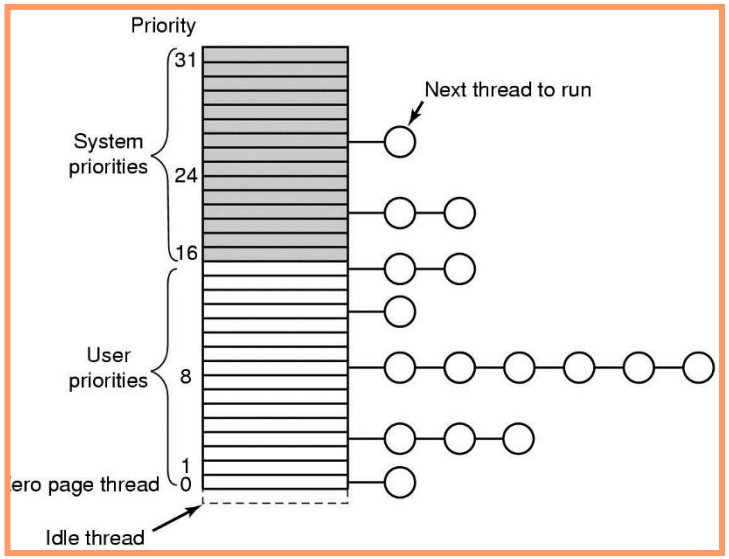
| • 优先级策略 | • 轮流策略 | • 分类调度策略 |
| --- | --- | --- |
| ✓先来先服务算法 | ✓分时调度 | ✓前后台队列调度 |
| ✓短进程优先算法 | ✓彩票调度 | ✓多级反馈队列调度 |
| ✓高响应比优先算法 | ✓步幅调度 | |

# 3.2 调度算法

## 3.2.1 优先级策略

- 优先权算法
  - ✓反映作业/进程执行时的迫切程度，通常用1个整型数来表示
  - ✓静态优先权：进程创建时确定（根据进程类型、资源需求和用户要求等），直到进程执行结束，保持不变
  - ✓动态优先权：进程创建时确定（根据进程类型、资源需求和用户要求等）初始优先权，在进程执行过程中，可以发生变化。

**所谓调度算法就是在一个调度时机从一堆候选者中挑选一个对象，是一个优化问题。**



# 3.2 调度算法

## 3.1.2 优先级策略

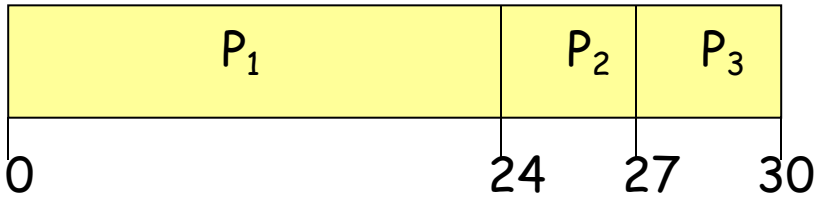
- 先来先服务算法（FCFS）

$$\text{带权周转时间}=\frac{\text{完成时间}-\text{到达时间}}{\text{服务时间}}$$

| Process | 运行时间 |
|---|---|
| $P_1$ | 24 |
| $P_2$ | 3 |
| $P_3$ | 3 |

- 性能与进程长度和运行顺序相关
- 对I/O密集性任务不友好

假设进程到达顺序为:P1(0) , P2(1) , P3(2)
用Gantt图表示的调度顺序为:



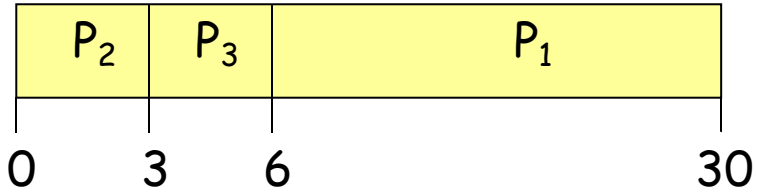

等待时间： $P_1$ = 0; $P_2$ = 23; $P_3$ = 25

平均等待时间：(0 + 23 + 25)/3 = 16

周转时间：$P_1$ = 24; $P_2$ = 26; $P_3$ = 28

平均周转时间：(24+26+28)/3=26

假设进程到达顺序为: P2(0) , P3(1) ,P1(2)
用Gantt图表示的调度顺序为:




等待时间： $P_1$ = 4； $P_2$ = 0； $P_3$ = 2

平均等待时间：(4 + 0 + 2)/3 = 2

周转时间：$P_2$ = 3； $P_3$ = 5； $P_1$ = 28

平均周转时间：(3+5+28)/3=12

# 3.2 调度算法

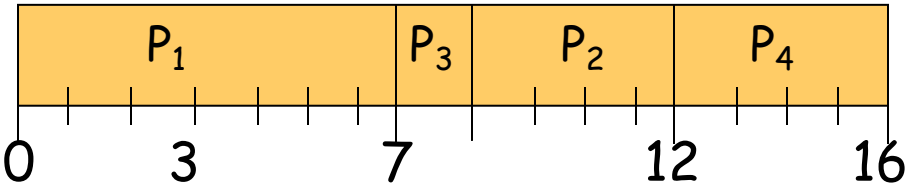
## 3.2.1 优先级策略

- 短进程/作业优先（SPF/SJF）

| Process | Arrival Time | 运行时间 |
|---|---|---|
| $P_1$ | 0.0 | 7 |
| $P_2$ | 2.0 | 4 |
| $P_3$ | 4.0 | 1 |
| $P_4$ | 5.0 | 4 |

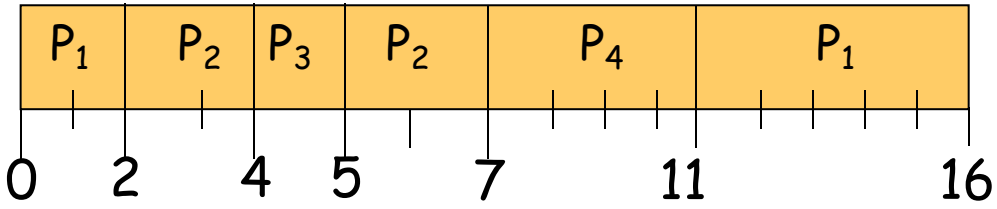
算法对长进程不利！

SPF (非抢占式)




平均等待时间 (0 + 6 + 3 + 7)/4 = 4

SPF (抢占式)




平均等待时间 (9 + 1 + 0 +2)/4 = 3

最短完成时间任务优先（STCF）

# 3.2 调度算法

## 3.2.1 优先级策略

- 高响应比优先算法

体现先来先服务

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

体现短作业优先

**该算法既照顾了短作业，又考虑了作业到达的先后次序，优先权动态变化。**

设有三个作业J1、J2、J3，它们的到达时间分别为8：00、8：45、9：30，计算时间分别为2小时、1小时、0.25小时，它们在一台处理机上按单道运行，9点处理机开始运转,若采用响应比高者优先的调度算法，这三个作业的执行次序是什么?

9：00 J1运行
11：00 J1完成，计算 J2：（135+60）/60=2.25 J3：（90+15）/15=7 J3运行
11：15 J2运行

# 3.2 调度算法

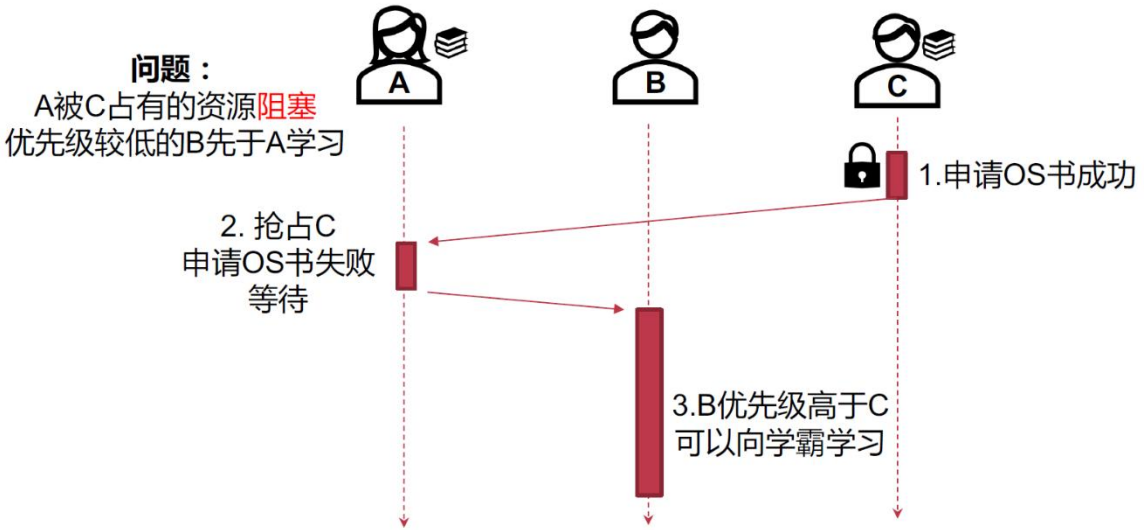
## 3.2.1 优先级策略

- 其它可以作为优先权值的考虑因素
  - ✓系统进程的优先权通常比用户进程高
  - ✓占用内存大小
  - ✓I/O时间长短
  - ✓。。。
- 优先权因素的层次、加权和自定义公式
  - ✓每次只考虑一个因素F，相同情况下看下一个因素
  - ✓优先权=t1*F1+ t2*F2+ 。。。+tn*Fn
  - ✓自定义公式：如高响应比
  - ✓自定义时间间隔定量变化

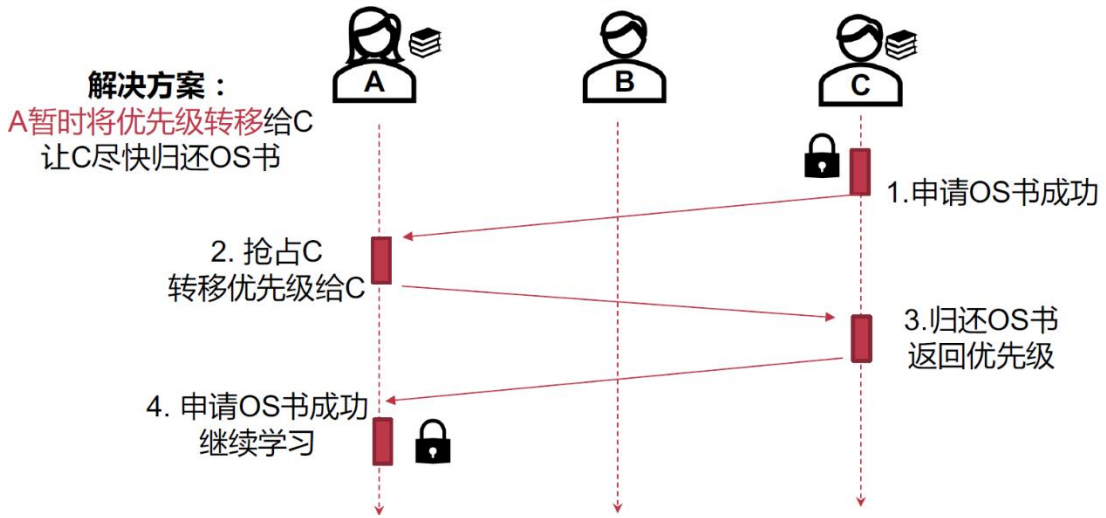
# 3.2 调度算法

## 3.2.1 优先级策略

- 优先级反转
  - ✓优先级 A>B>C




- 优先级继承



# 试一试

（3）下列选项中，降低进程优先级的合理时机是（　　）

A. 进程的时间片用完　B. 进程刚完成I/O，进入就绪列队

C. 进程长期处于就绪列队中　D. 进程从就绪态转为运行态

（4）某系统正在执行三个进程P1、P2和P3，各进程的计算(CPU)时间和I/O时间比例如下表所示。

| 进程 | 计算时间 | I/O 时间 |
|---|---|---|
| P1 | 90% | 10% |
| P2 | 50% | 50% |
| P3 | 15% | 85% |

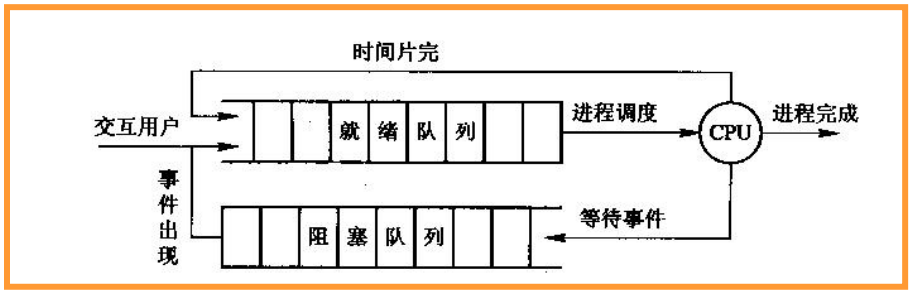
为提高系统资源利用率，合理的进程优先级设置为：

A. P1 >P2 >P3　B. P3>P2 >P1　C. P2>P1 =P3　D. P1>P2=P3

# 3.2 调度算法

## 3.2.2 轮流策略

- 分时调度




时间片选择

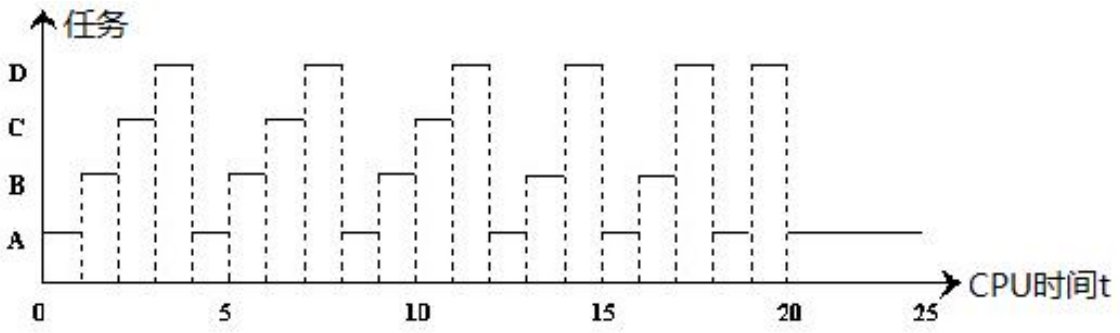
- ✓固定时间片
- ✓可变时间片

时间片大小

- ✓不可太大：影响最大响应时间（$T=nq$）
- ✓不可太小：调度开销，增加周转时间

与其它模型叠加?

- ✓时间片+先来先服务
- ✓时间片+短进程优先
- ✓时间片+高响应比优先

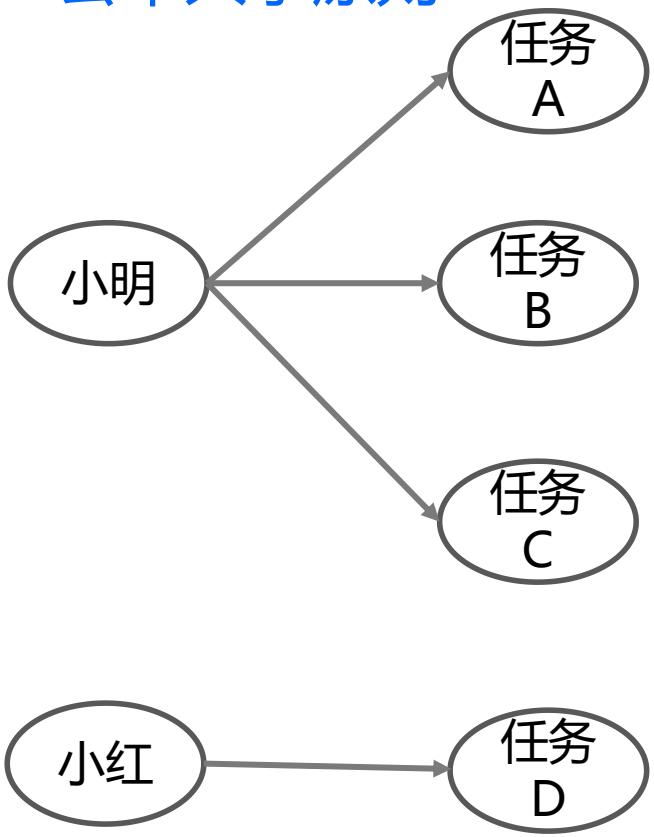



思考：时间片大小和周转时间的关系?

# 3.2 调度算法

## 3.2.2 轮流策略

- 公平共享原则




时间片均分：小明 75%  小红  25%

| 任务A | 任务B | 任务C | 任务D |
|---|---|---|---|

资源公平：小明 50%  小红  50%

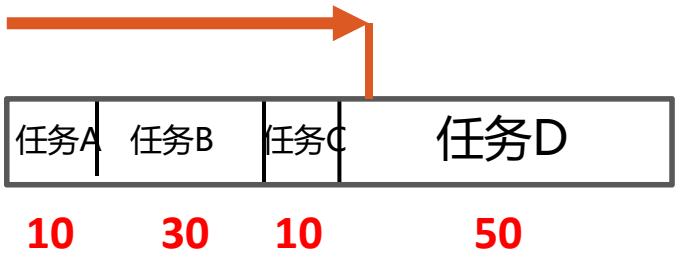
| 任务A | 任务B | 任务C | 任务D |
|---|---|---|---|

# 3.2 调度算法

## 3.2.2 轮流策略

随机数会带来什么问题?

- 彩票调度
  - ✓每次调度时，生成随机数$R$ ,找到对应的任务
  - ✓R=55，调度任务D




```
R = random(0, T)
sum = 0
foreach(task in task_list) {
    sum += task.ticket
    if (R < sum) {
        break
    }
}
schedule()
```

- 彩票转让
  - ✓允许用户间转让彩票避免优先级翻转
- 彩票通胀
  - ✓允许用户动态改变自己的彩票

# 3.2 调度算法

## 3.2.2 轮流策略

- 步幅调度
  - ✓Stride——步幅，任务一次执行增加的虚拟时间

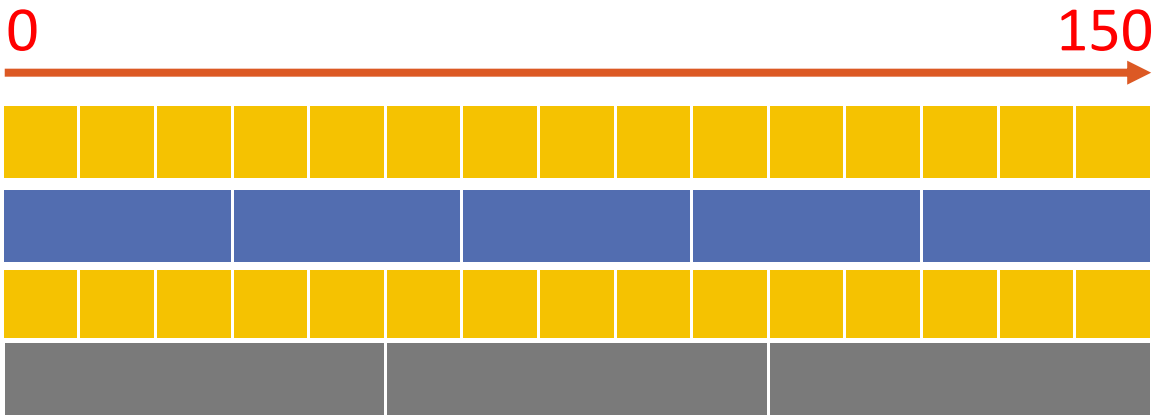
$$stride = \frac{MaxStride}{ticket}$$

  - ✓Pass——累计执行的虚拟时间

确定性版本的彩票调度

```
/* select client with minimum pass value */
task = remove_queue_min(q);
/* use resource for quantum */
schedule(task);
/* compute next pass using stride */
task->pass += task->stride;
insert_queue(q, current);
```

| | Ticket | Stride |
|---|---|---|
| 任务A | 10 | 15 |
| 任务B | 30 | 5 |
| 任务C | 10 | 15 |
| 任务D | 50 | 3 |



# 试一试

（5）在分时系统中，假设就绪队列中有10个进程，系统将时间片设为200ms，CPU进行进程切换要花费10ms。则系统开销所占的比率约为（　　）

A．1%　　B．5%　　C．10%　　D．20%

（6）下列选项中，满足短任务优先且不会发生饥饿现象的调度算法是（　　）

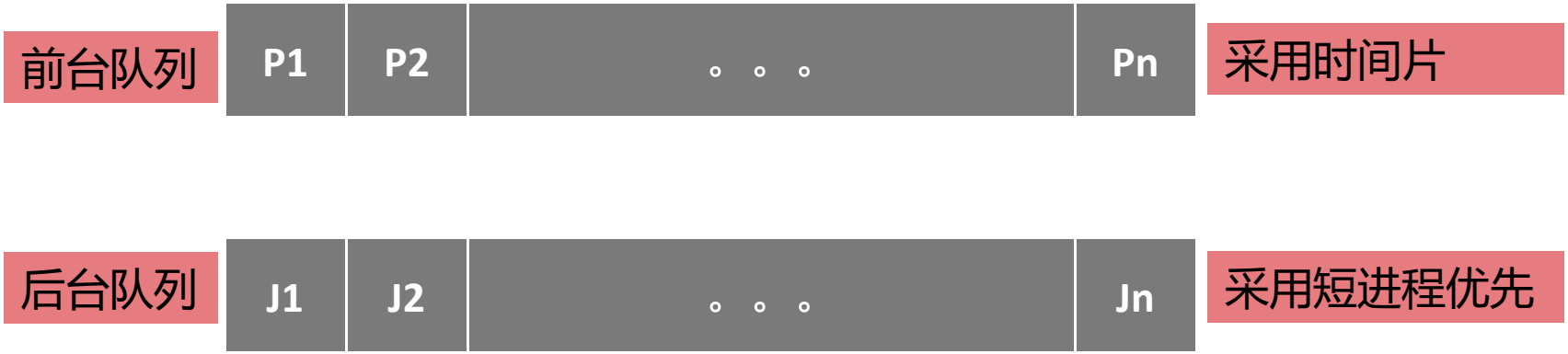
A．先来先服务　　B. 高响应比优先

C. 时间片轮转　　D.抢占式短任务优先

# 3.2 调度算法

## 3.2.3 分类调度策略

- 前后台队列调度
  - ✓既满足交互性很强的终端用户需求，又能及时满足远程用户提交的作业请求
  - ✓前后台采取的调度算法不同
  - ✓前台无进程的情况下，才调度后台队列的进程（如何实现？）



# 3.2 调度算法

## 3.2.3 分类调度策略

- 多级反馈队列调度算法
  - ✓不同队列时间片不同
  - ✓不同队列优先权不同
  - ✓一个队列执行一个时间片，执行不完进入下一级队列
  - ✓交互进程可以在第一个队列结束
  - ✓短进程可以在前2、3个队列中结束
  - ✓越到最后时间片越长，长进程终可以完成

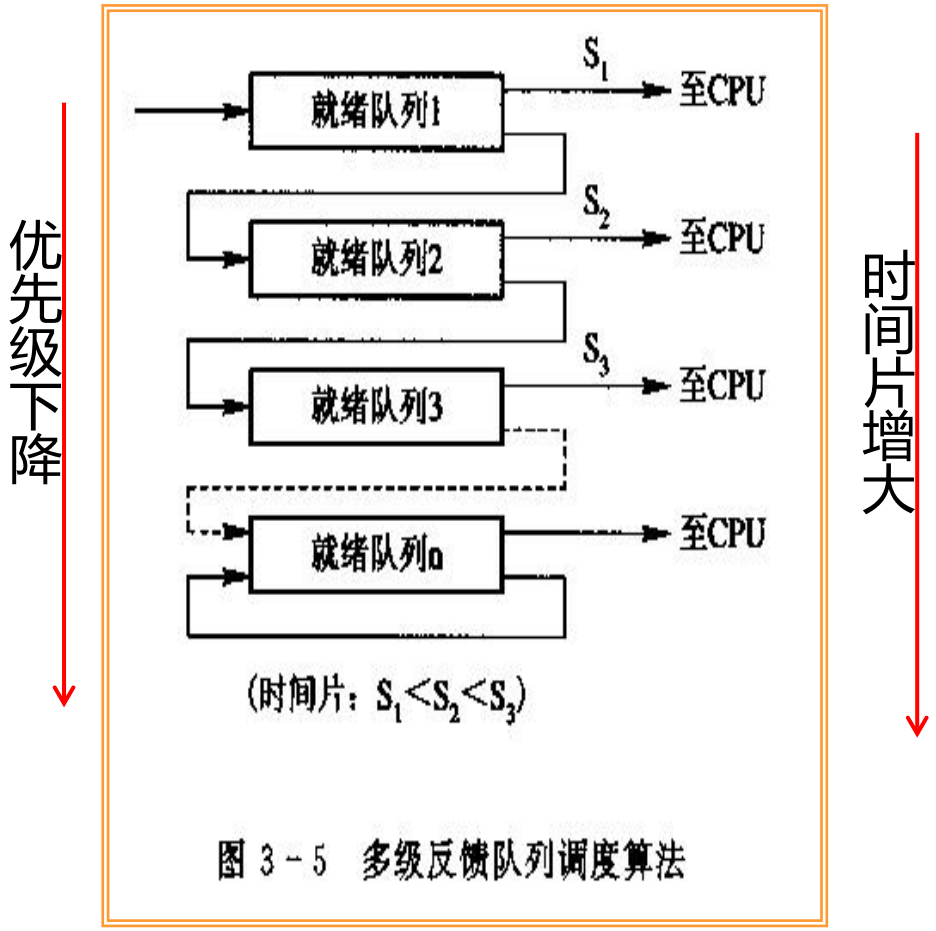
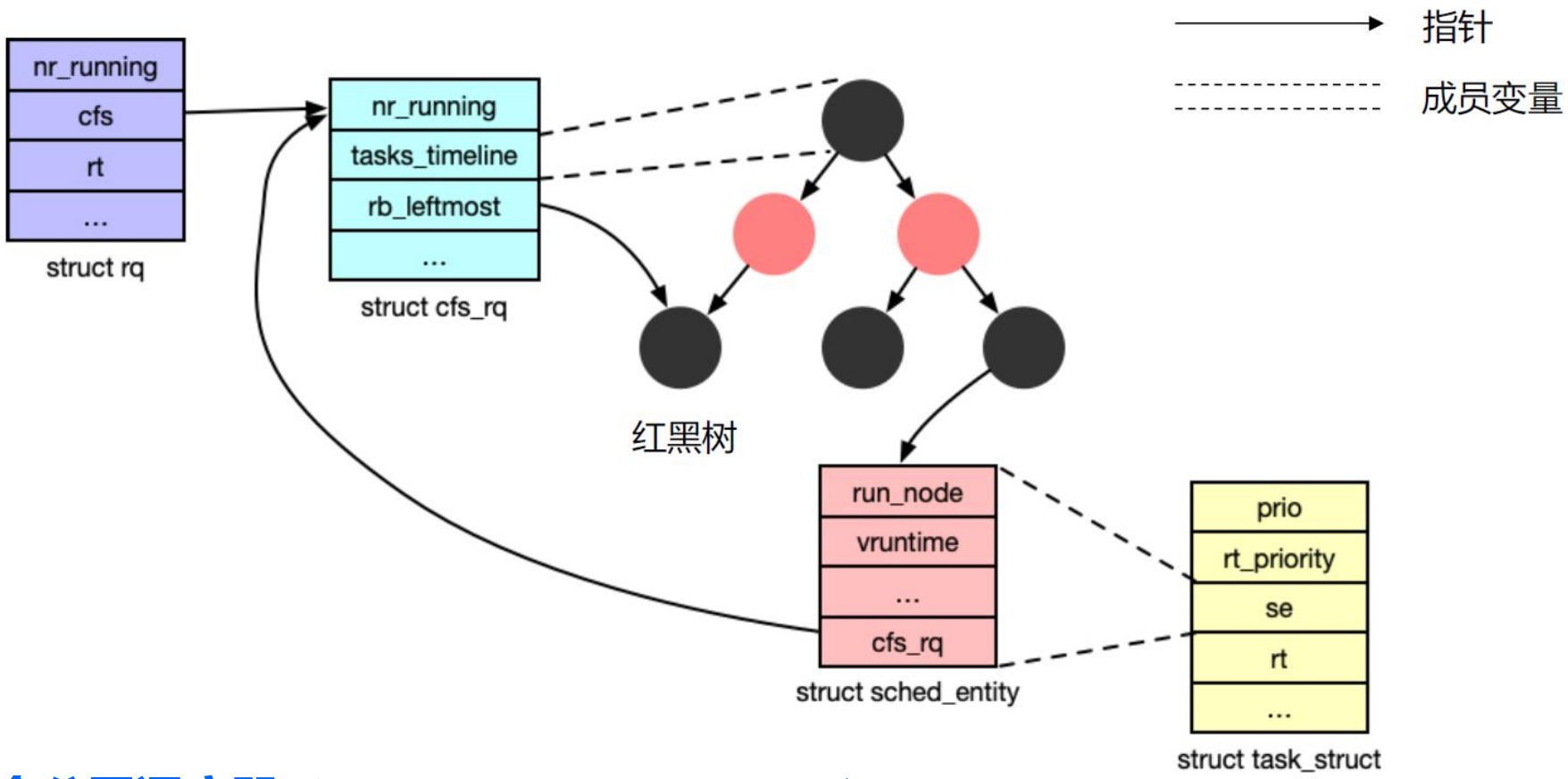



图 3-5 多级反馈队列调度算法

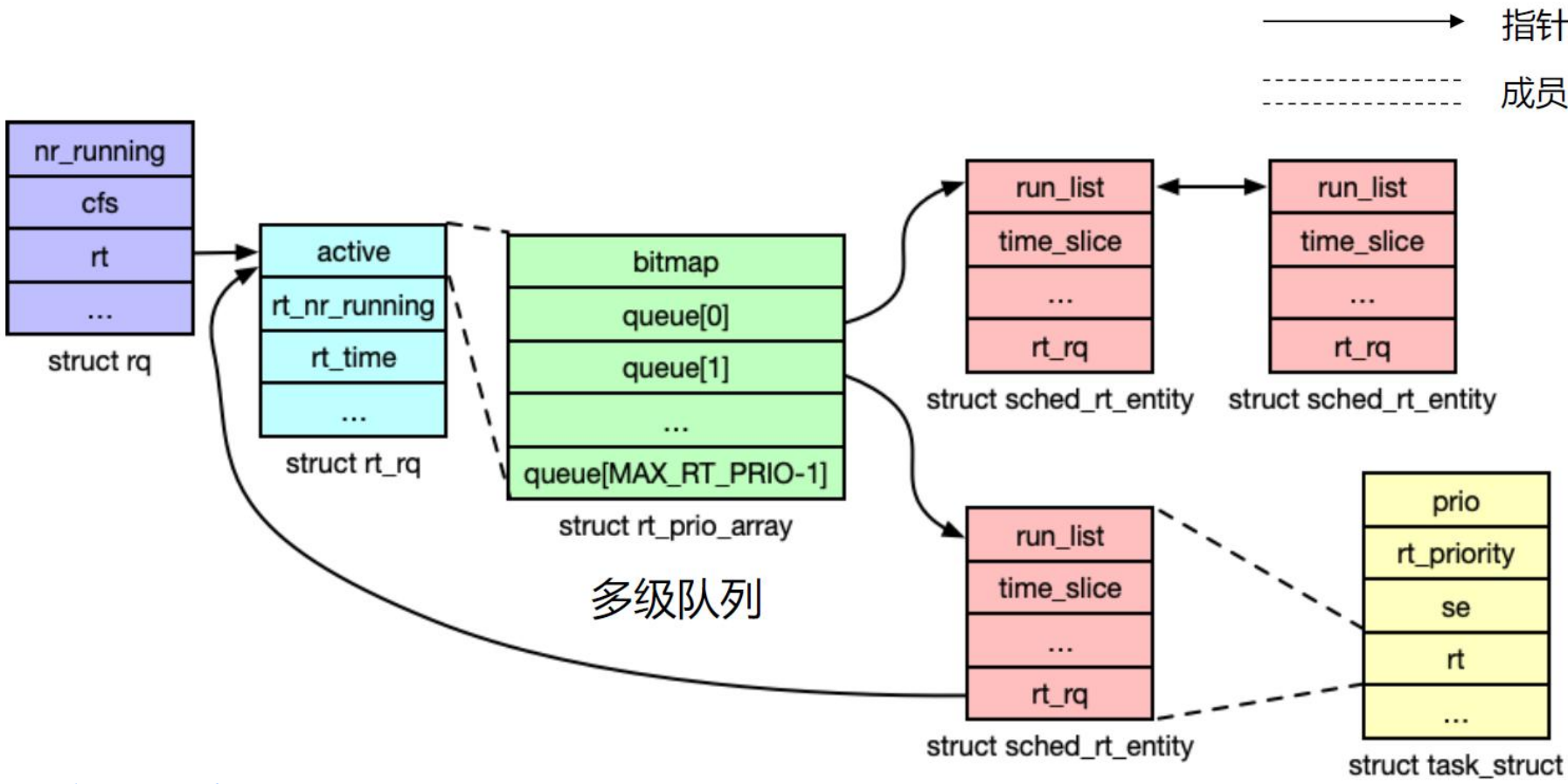
在一个使用多级反馈队列进行调度的系统中，一个进行需要执行50秒，如果第一个队列时间片为5，较低一级的时间片是上一级时间片的2倍，那么这个作业会被中断多少次，当它终止时处于哪个队列？

# Linux调度机制：CFS Run Queue




**完全公平调度器（Completely Fair Schedular）**

# Linux调度机制：RT Run Queue




**实时调度器（Real-Time Schedular）**

# 3.3 实时调度

## 3.3.1 实现实时调度的基本条件

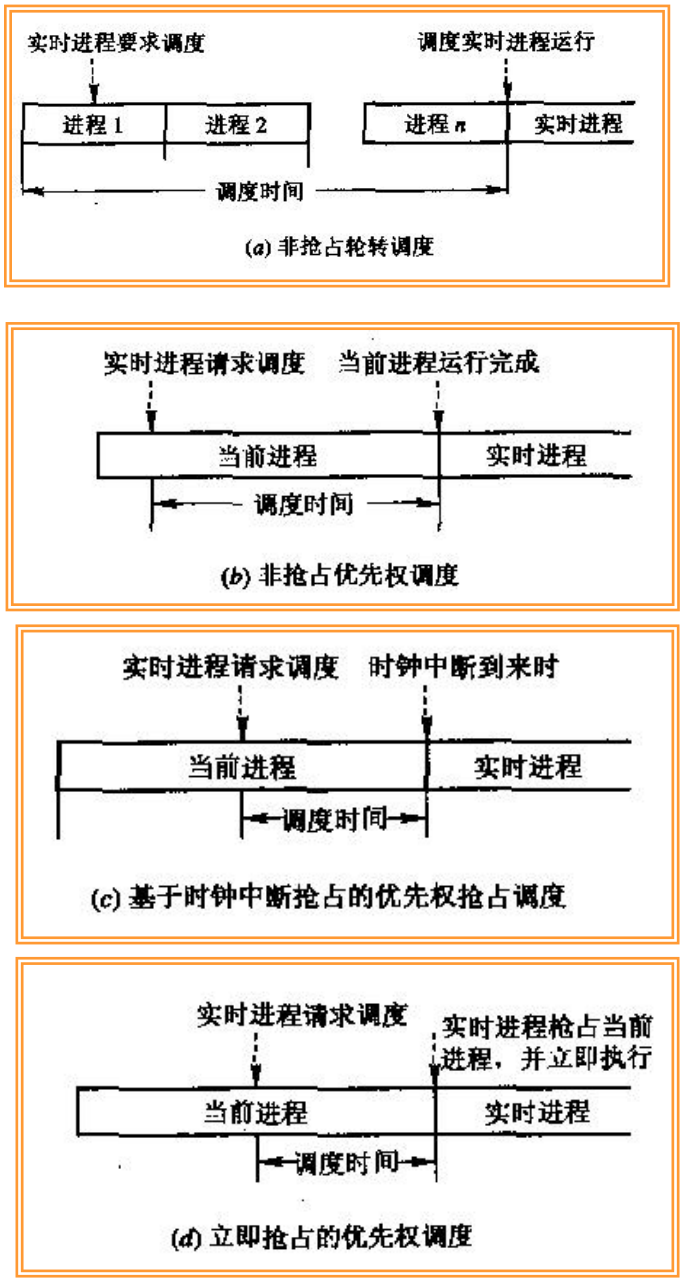
- 对于m个实时任务，处理时间为$C_i$，周期时间为$P_i$，则系统是可调度的，如果满足下列条件：

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

# 3.3 实时调度

## 3.3.2 实时调度的分类

- 根据实时任务性质
  - ✓ 硬实时调度算法：严格实时
  - ✓ 软实时调度算法：非严格实时
- 根据调度方式
  - ✓ 非抢占调度算法
  - ✓ 抢占调度算法
- 根据调度时间
  - ✓ 静态调度：在进程执行前，调度程序便已经决定了各进程间的执行顺序
  - ✓ 动态调度：在进程的执行过程中，由调度程序届时根据情况临时决定将哪一进程投入运行



# 3.3 实时调度

## 3.3.3 两个典型算法

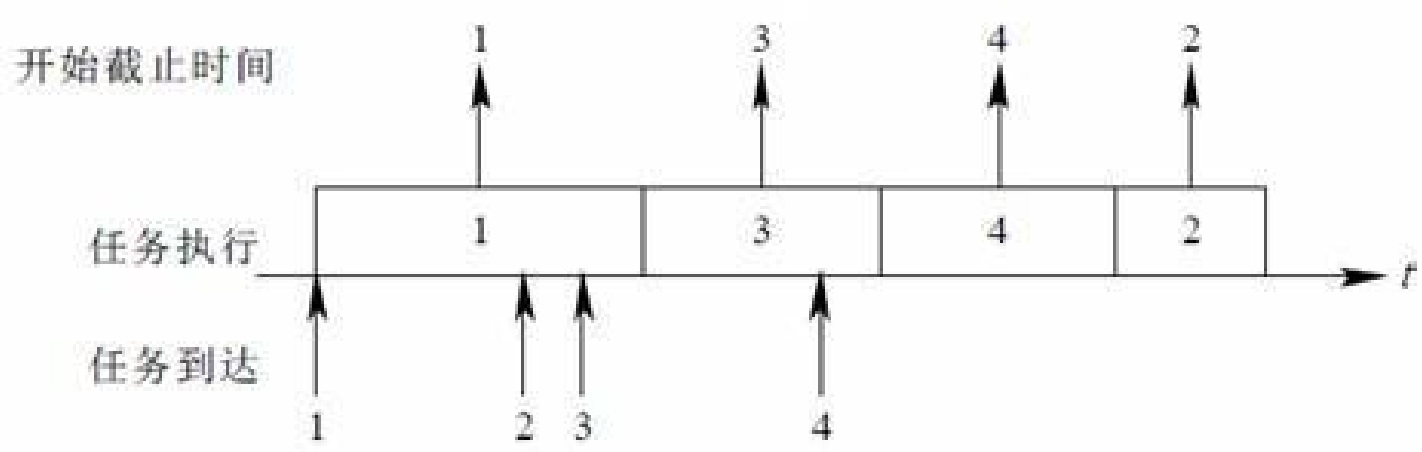
- 最早截止时间优先调度算法(EDF)

✓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

✓设置实时任务就绪队列，按各任务截止时间的早晚排序

✓调度程序选择就绪队列中的第一个任务运行，即最需要开始的任务



# 3.3 实时调度

## 3.3.3 两个典型算法

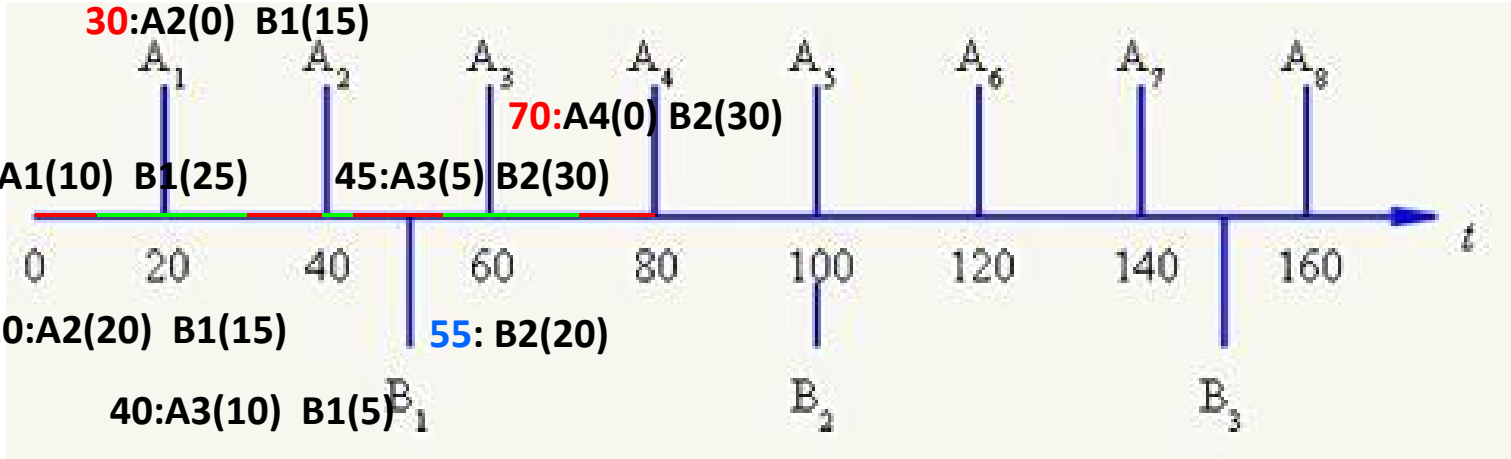
计算松弛度的时机？？

- 最低松弛度优先调度算法(LLF)

✓根据任务的紧急程度（松弛度）来确定任务的优先级。

松弛度 = 必须完成时间/完成截至时间 - 本身运行时间 - 当前时间

✓调度程序在选择任务时，总是选择就绪队列中(在任务执行周期)紧急程度（松弛度最低）最大任务，为之分配处理机，使之投入运行。

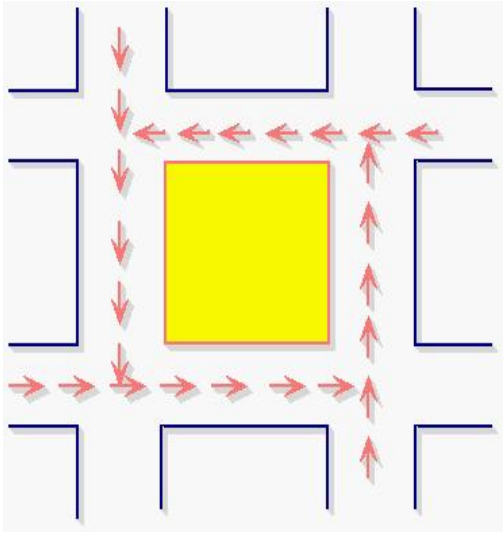



周期性任务A和B，执行时间分别为10ms和25ms，周期分别为20ms和50ms。

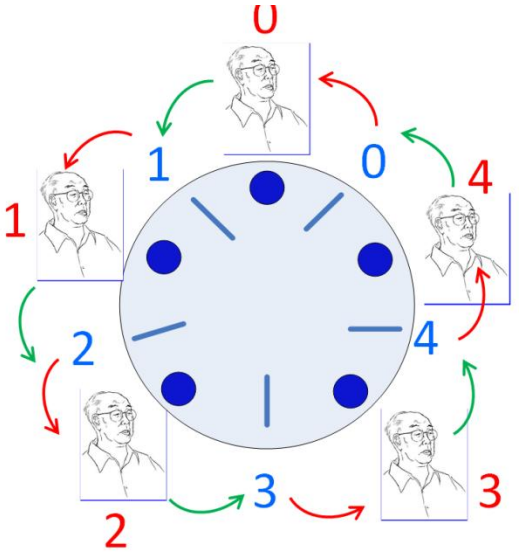
# 3.4 死锁概述

## 3.4.1 死锁与死锁状态

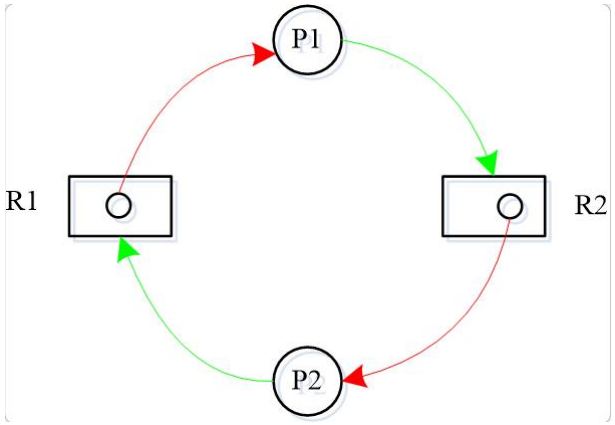
例1：交通死锁



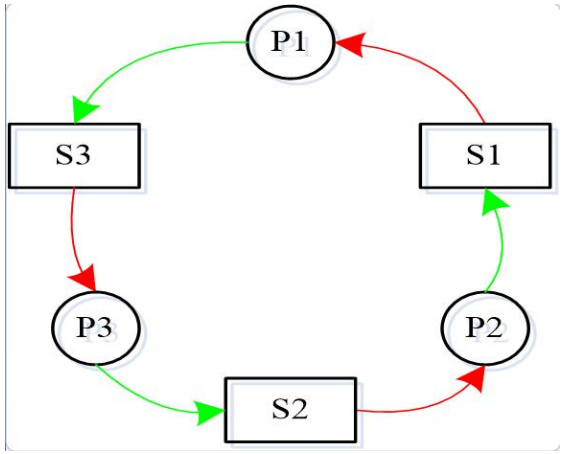
例2：哲学家进餐




例3：两进程共享资源




例4：三进程接发消息



# 3.4 死锁概述

## 3.4.1 死锁与死锁状态

- 死锁（Deadlock）的定义
  - ✓指多个进程在运行过程中因争夺资源而造成的一种僵局。当进程处于这种僵局时，若无外力作用，他们都将无法再向前推进。
- 死锁与阻塞的区别
  - ✓死锁的进程处于阻塞状态，但仅依靠自己，无法继续运行。
- 死锁与死机的区别

# 3.4 死锁概述

## 3.4.2 产生死锁的原因

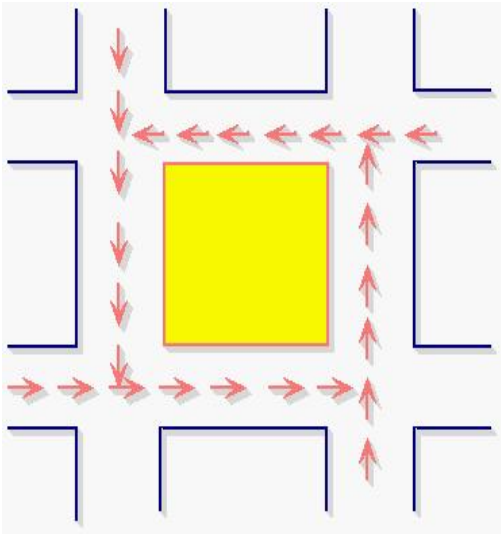
- 竞争非剥夺性资源
  - ✓可剥夺性资源（未用完可以被剥夺）： 内存等
  - ✓非剥夺性资源（必须在用完后才能被剥夺）： 打印机等
- 进程推进顺序非法
  - ✓错误的进程推进顺序 导致 死锁
  - ✓正确的进程推进顺序 不导致 死锁

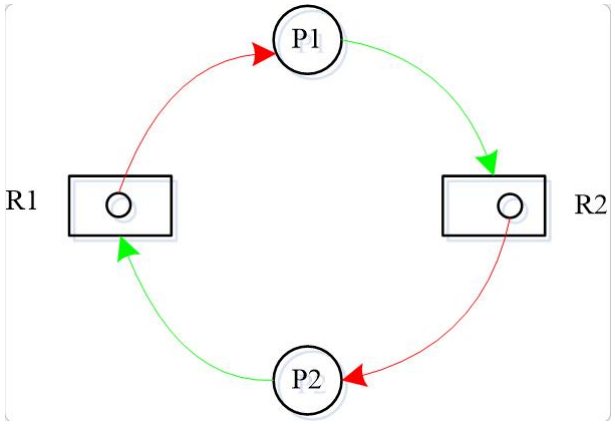
# 3.4 死锁概述

## 3.4.3 死锁的必要条件

- 互斥条件
  - ✓互斥使用资源：资源自身的特点
- 请求和保持条件
  - ✓进程占有一个资源的同时，请求另外的资源。
- 不剥夺条件
  - ✓进程的资源使用完之前，不能被剥夺。
- 环路等待条件
  - ✓死锁的n个进程形成环形的进程-资源链。

<P1→P2 → … → Pi → … → Pn>





# 3.4 死锁概述

## 3.4.4 处理死锁的基本方法

- 预防死锁 ——→ 程序员行为，OS提供方法
  - ✓作法：破坏死锁产生的四个必要条件
  - ✓优点：简单
  - ✓ 缺点：资源利用率低，效率低
- 避免死锁 ——→ OS自发行为
  - ✓作法：进程申请资源时，由系统审查申请的合理性， 只有不导致死锁的申请，才被认可
  - ✓优点：效率略高
  - ✓缺点：实现困难：进程对资源的需求不好事先确定
- 检测和解除死锁 ——→ 可以是用户行为，也可以OS自发
  - ✓作法：进程申请资源时不作任何限制，仅在系统中定时或资源不足时，检查是否有死锁；如果有，解决之。
  - ✓优点：不影响系统执行性能，效率高

# 3.5 预防死锁

## 3.5.1 破坏请求和保持条件

- 做法
  - ✓一次申请所有资源。成功则运行，否则阻塞。
- 优点
  - ✓简单、易于实现，安全
- 缺点
  - ✓资源严重浪费
  - ✓进程执行进度大大延迟
- OS提供方法
  - ✓信号量集操作

# 3.5 预防死锁

## 3.5.2 破坏不剥夺条件

- 做法
  - ✓进程申请资源时，成功则运行，否则释放所有资源后阻塞。
- 优点
  - ✓无
- 缺点
  - ✓资源严重浪费；
  - ✓代价太大；
  - ✓进程执行进度严重延迟
- OS提供方法
  - ✓新的wait()函数

# 3.5 预防死锁

## 3.5.3 破坏环路等待条件

- 做法
  - ✓为所有资源编号，进程申请资源时必须按序申请资源。

  例如：

  R1：打印机, R2: 磁带机；R3: 磁盘；R4:输入机

  进程P1使用资源顺序：输入机R4 ,磁带机R2, 打印机R1

  但是P1资源申请顺序必须为：R1,R2,R4.
- 优点
  - ✓资源利用率、系统吞吐量显著提高
- 缺点
  - ✓资源还是会浪费；资源编号困难；新资源加入困难
- 程序员自行设计

# 3.6 避免死锁

- 允许系统具备四个必要条件
- 进程在执行过程中动态地申请和释放资源
- 操作系统在发生死锁前进行检查推断

进程申请资源的时候

每次资源分配时，由系统仔细检查此次资源分配的安全性，只有在安全时才分配，否则不分配，并令进程等待

Wait操作如何改?

# 3.6 避免死锁

## 3.6.1 安全状态与不安全状态

- 假定某系统有n个进程并发执行，对于某个时刻T0，划分系统安全和不安全的标准如下：

系统能够按照某种进程顺序<P1,P2,...,Pn>执行，当轮到每个进程Pi执行时，都能满足该进程对资源的最大需求，则该系统处于安全状态，否则为不安全状态。
<P1,P2,...,Pn>称为安全序列。

示例：系统3个进程P1,P2,P3，共享12台磁带机资源，某时刻T0的资源分配状况如下表所示

| 进程 | 最大需求量 | 已分配 | | 可用 |
|---|---|---|---|---|
| P1 | 10 | 5 | | 3 |
| P2 | 4 | 2 | | |
| P3 | 9 | 2 | | |

（1）T0时刻安全性：安全序列：<P2,P1,P3>

# 3.6 避免死锁

## 3.6.1 安全状态与不安全状态

(2) P3请求1台磁带机后

| 进程 | 最大需求量 | 已分配 | 还需要 | 可用 |
|---|---|---|---|---|
| P1 | 10 | 5 | 5 | 3 |
| P2 | 4 | 2 | 2 | |
| P3 | 9 | 2 | 7 | |

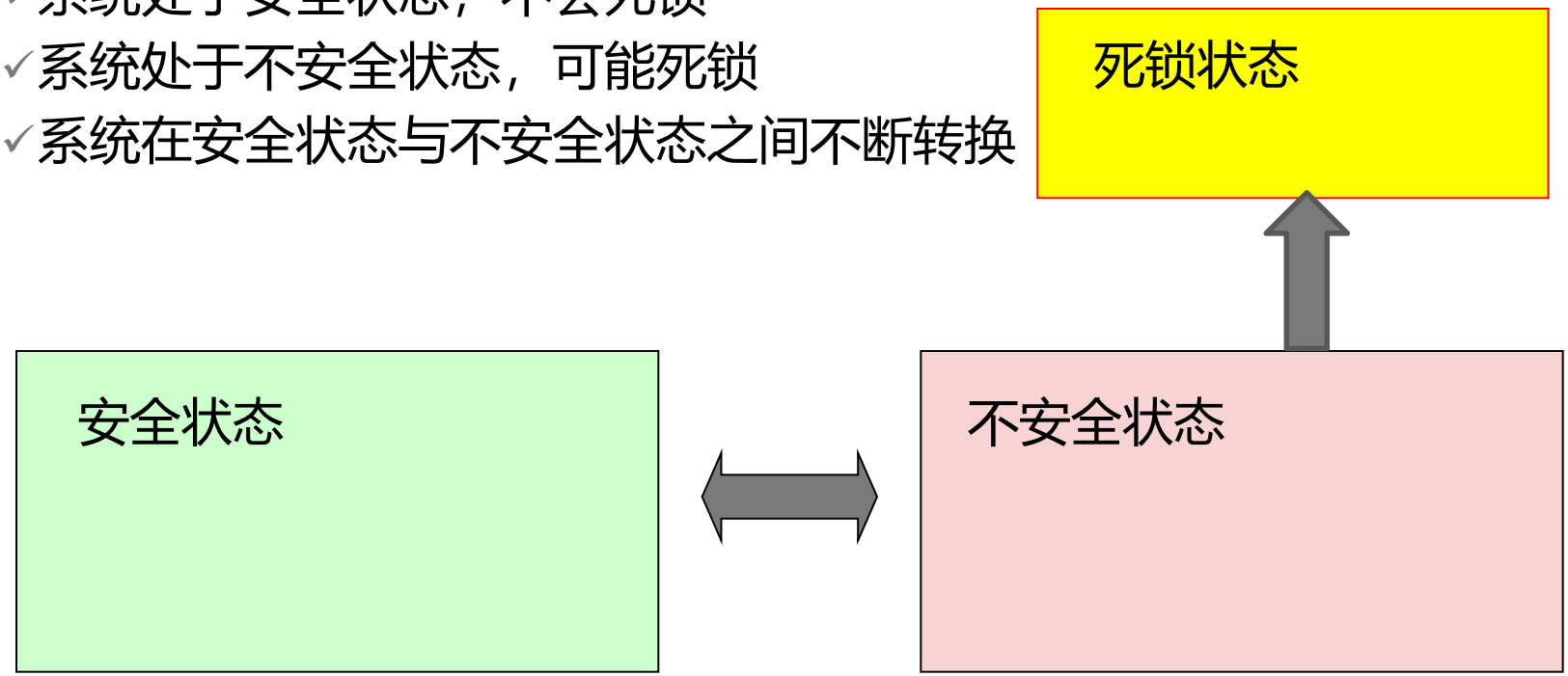
| 进程 | 最大需求量 | 已分配 | 还需要 | 可用 |
|---|---|---|---|---|
| P1 | 10 | 5 | 5 | 2 |
| P2 | 4 | 2 | 2 | |
| P3 | 9 | 3 | 6 | |

安全序列：<P2, ？ , ？ >

# 3.6 避免死锁

## 3.6.1 安全状态与不安全状态

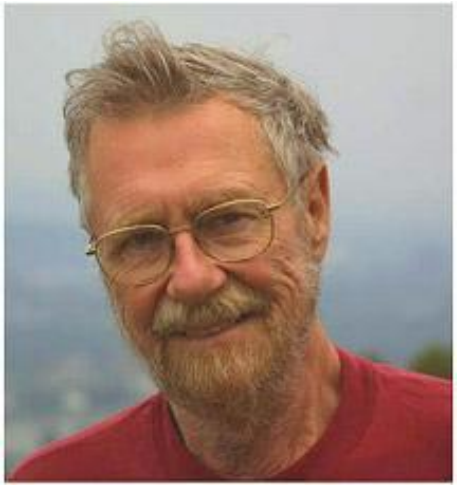
- 安全状态、不安全状态、死锁状态之间的关系
  - ✓系统处于安全状态，不会死锁
  - ✓系统处于不安全状态，可能死锁
  - ✓系统在安全状态与不安全状态之间不断转换



# 3.6 避免死锁

## 3.6.2 银行家算法

- Dijkstra：荷兰计算机科学家
- 贡献
  - ✓提出“goto有害论”；
  - ✓提出信号量和PV原语;
  - ✓解决了有趣的“哲学家聚餐”问题;
  - ✓最短路径算法(SPF)的创造者;
  - ✓第一个Algol 60编译器的设计者和实现者;
  - ✓THE操作系统的设计者和开发者;



问题：n个进程{P1,P2,..,Pn}共享m类资源{R1,R2,.., Rm}，进程并发执行时，如何避免死锁。

# 3.6 避免死锁

## 3.6.2 银行家算法

**m类资源，n个进程**

- 数据对象
  - ✓可用资源向量(m)：Available[j]
  - ✓最大需求矩阵(n*m)：Max[i][j]
  - ✓分配矩阵(n*m)：Allocation[i][j]
  - ✓需求矩阵(n*m)：Need[i][j]
  - ✓资源请求向量(m)：$Request_i[j]$

**Max = Allocation + Need**

当Pi进程提出资源请求$Request_i$，执行银行家算法Bank(i, Request)，如下描述：

step1：IF $Request_i$ not ≤ $Need_i$
THEN 出错；

step2：IF $Request_i$ not ≤ Available
THEN Pi等待；

step3：试分配，修改数据结构；
Available := Available - $Request_i$
$Allocation_i$ := $Allocation_i$ + $Request_i$
$Need_i$ := $Need_i$ - $Request_i$

step4：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；

step5：IF 安全
THEN 正式分配
ELSE 取消试分配，Pi等待；

# 3.6 避免死锁

## 3.6.2 银行家算法

- 安全性算法 safe()

step1：设置工作向量Work,长度m,Work := Available ;

step2：设置状态向量Finish,长度n, Finish := false ;

step3：从进程集合查找满足下列条件之进程$P_k$：

Finish[k] = false ;

$Need_k \leq Work$ ;

IF 未找到这样的进程 THEN GOTO (5)

step4：执行如下操作：

Work := Work + $Allocation_k$

Finish[k] := true ;

GOTO (3)

step5： IF Finish = true THEN 安全

ELSE 不安全;

# 3.6 避免死锁

## 3.6.2 银行家算法

- 进程{P0,P1,P2,P3,P4}共享资源{A,B,C}。T0时刻资源状况如下：

| | Max<br>A B C | Allocation<br>A B C | Need<br>A B C | Available<br>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 | |

问题：

(1) T0时刻安全性；

(2) P1请求$Request_1(1,0,2)$；

(3) P4请求$Request_4(3,3,0)$；

(4) P0请求$request_0(0,2,0)$；

# 试一试

（1）3个进程共享资源A，每个进程最大需求为2个，则A至少有多少个资源时，才永远不会产生死锁?

（2）下面关于系统的安全状态的描述中正确的是（）

A、系统处于不安全状态可能会发生死锁

B、系统处于不安全状态一定会发生死锁

C、系统处于安全状态时也可能会发生死锁

D、不安全状态是死锁的一个特例

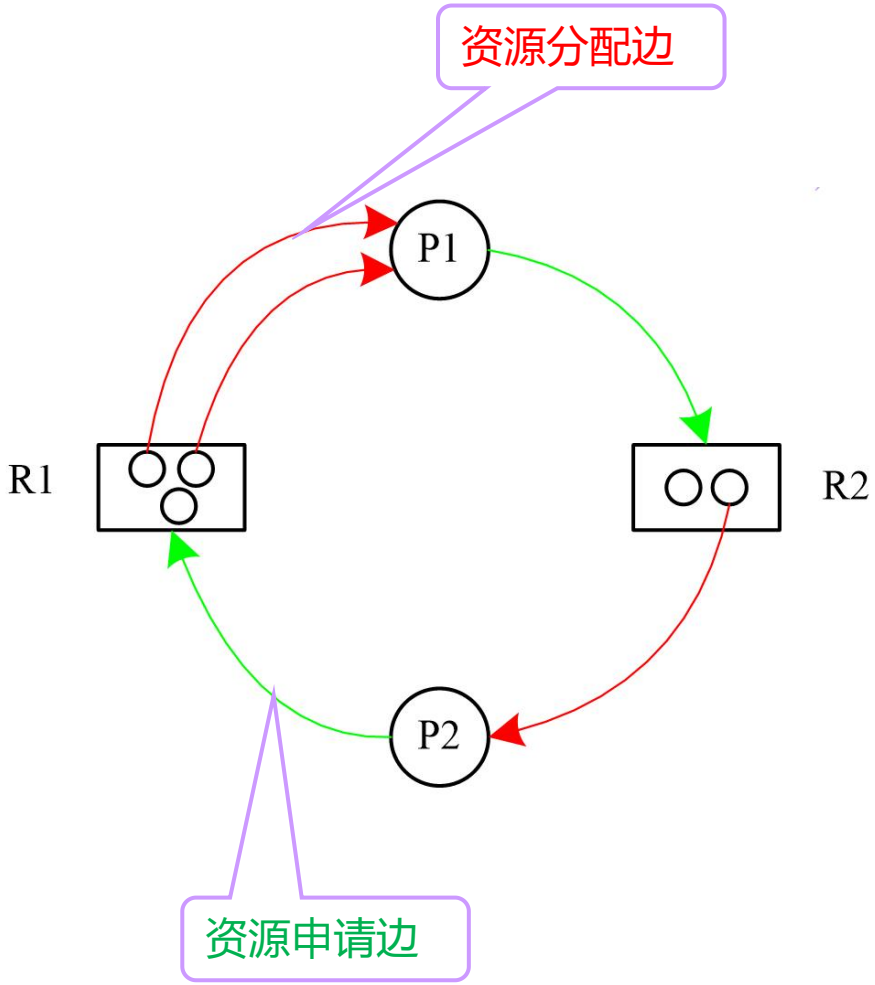
（3）某时刻进程的资源使用情况如下表所示：此时的安全序列是（ ）

| 进程 | 已分配资源 | | | 尚需分配 | | | 可用资源 | | |
|---|---|---|---|---|---|---|---|---|---|
| | R1 | R2 | R3 | R1 | R2 | R3 | R1 | R2 | R3 |
| P1 | 2 | 0 | 0 | 0 | 0 | 1 | 0 | 2 | 1 |
| P2 | 1 | 2 | 0 | 1 | 3 | 2 | | | |
| P3 | 0 | 1 | 1 | 1 | 3 | 1 | | | |
| P4 | 0 | 0 | 1 | 2 | 0 | 0 | | | |

# 3.7 检测死锁

- 基本思想
  - ✓进程动态的申请和释放资源
  - ✓允许系统产生死锁
  - ✓定时或者按需检测

- 如何检测
  - ✓使用资源分配图来描述系统现状
  - ✓圆圈代表进程
  - ✓方框代表资源
  - ✓小圆圈代表资源数目



# 3.7 检测死锁

- 检测算法

step1：对T0时刻资源分配图，寻找一个非阻塞非孤立结点，如找不到，转(4)；

step2：删除与该结点相连的所有边；

step3： GOTO (1)

step4： IF 最终的资源分配图中都是孤立结点

THEN T0时刻资源分配图无死锁

ELSE 有死锁（非孤立结点部分）

从进程集合查找满足下列条件之进程$P_k$：

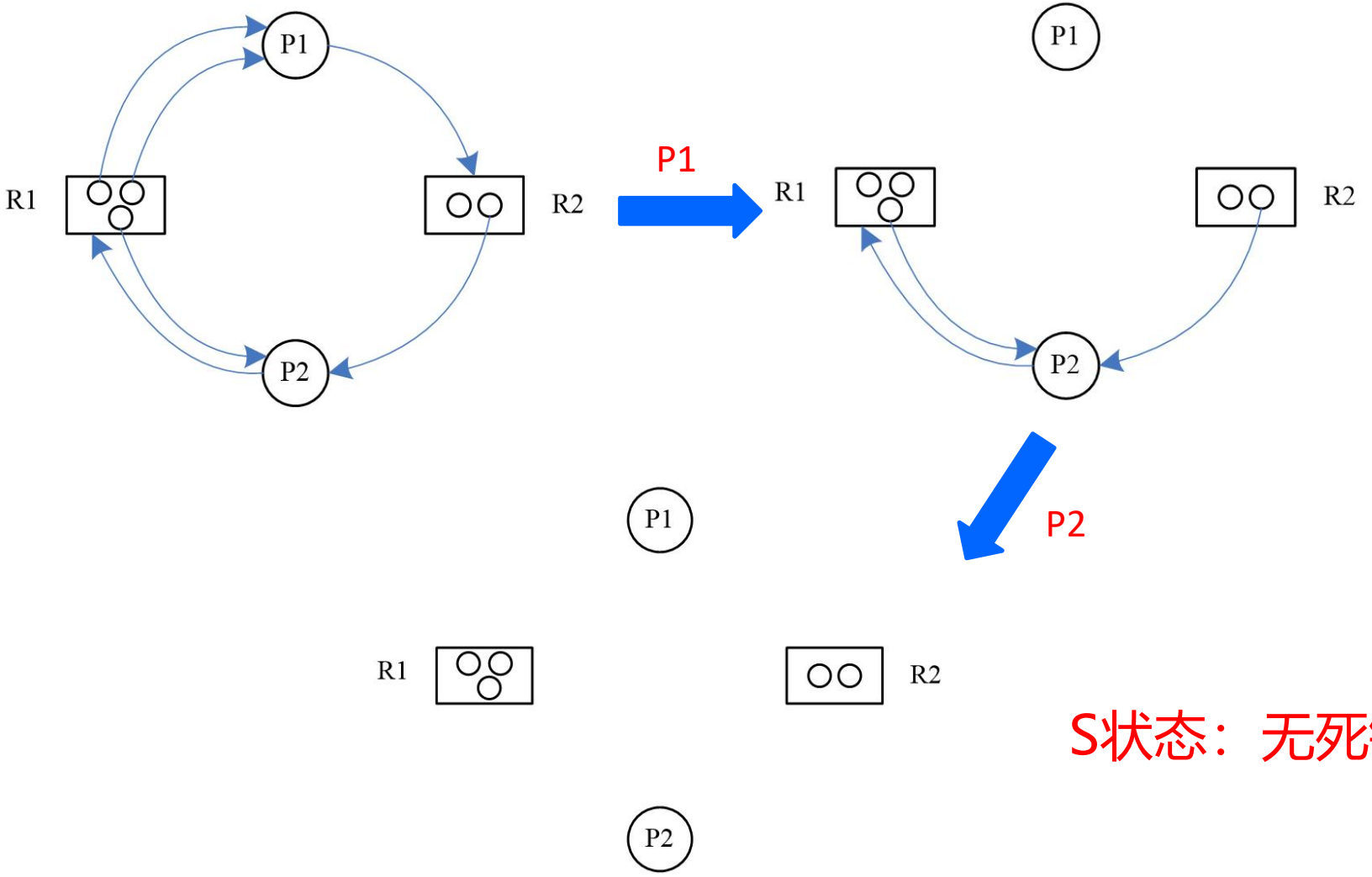
Finish[k] = false；

$Need_k \leq Work$；

**死锁定理**：

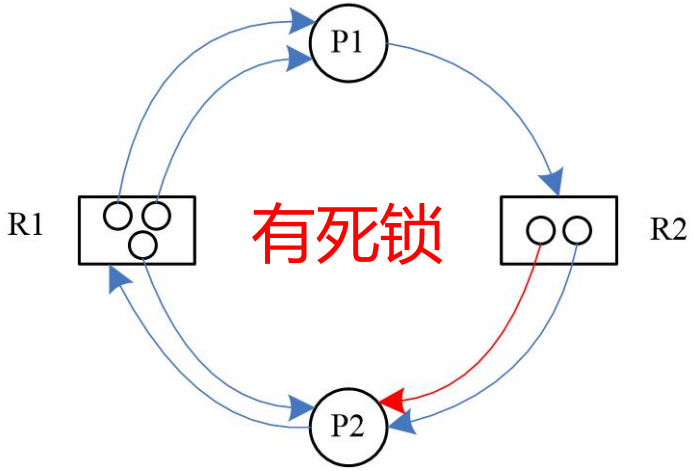
S状态为死锁状态的充分条件是：当且仅当S状态的资源分配图是不可完全简化的。

# 3.7 检测死锁

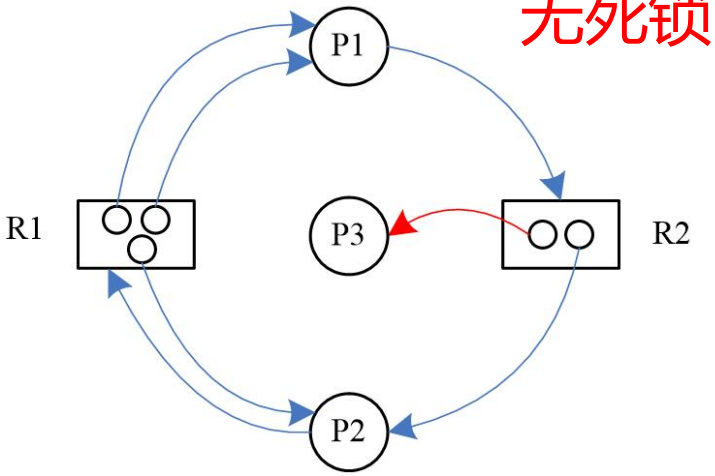



S状态：无死锁

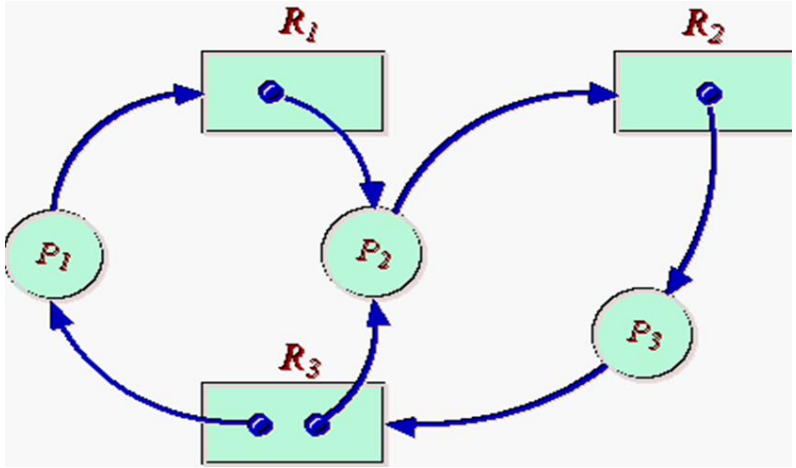
# 3.7 检测死锁




无死锁



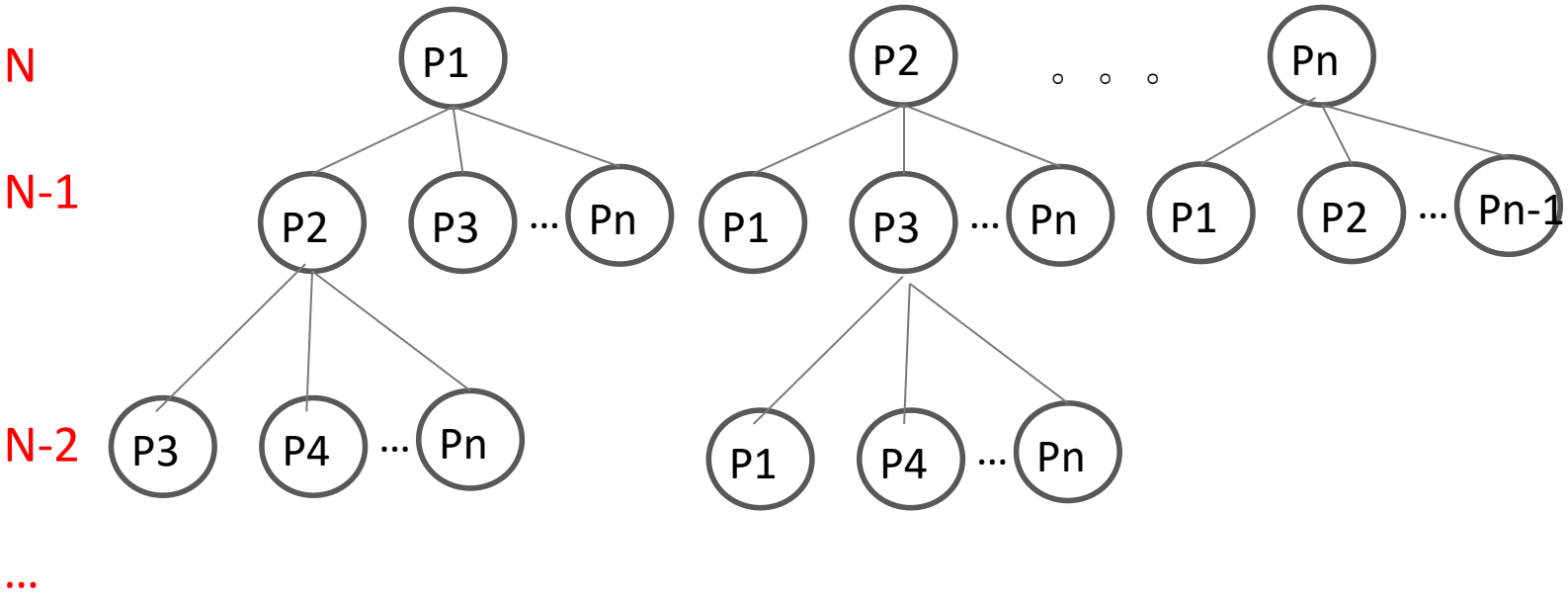

有死锁



# 3.8 解除死锁

是以最小的代价恢复系统的运行
撤消陷于死锁的全部进程

- 剥夺资源
- 撤消进程
  - ✓采用深度进程策略撤消：采用最短路径在森林中找一条代价最短的路径，局部最优
  - ✓采用宽度进程策略撤消：有N！种撤销序列的可能性，如全部遍历，则全局最优



# 第三次作业

1、为操作系统设计调度方案需要权衡哪些彼此矛盾的因素？为每对矛盾的因素权衡举一个实际的例子。

2、假设一个系统中有3个进程，到达时间依次为0，1，3。运行时间依次为3、5和2。按时间片轮转（时间片为2）调度算法或者先来先服务方法进行调度，给出不同方法下的平均周转时间。

3、在一个只允许四个进程在内存的批处理操作系统中，设在一段时间内先后到达6个作业，它们的提交时刻和运行时间由下表给出，作业被调度进入运行后不再退出（作业调度采用先来先服务，非抢占），但当每一作业成为进程运行时，可以调整运行的优先次序，采用抢占式短进程优先的调度算法。

（1）给出6个作业的执行时间序列

（2）计算在上述调度算法下作业的平均周转时间

| 作业号 | 提交时刻（时） | 运行时间（分钟） |
|---|---|---|
| J1 | 8：00 | 60 |
| J2 | 8：20 | 35 |
| J3 | 8：25 | 20 |
| J4 | 8：30 | 25 |
| J5 | 8：35 | 5 |
| J6 | 8：40 | 10 |

# 第三次作业

**4、采用银行家算法控制资源分配的系统，包含5进程(P0~P4)和4种资源(A~D)，假设在T0时刻资源的分配情况如下表所示，请问：**

**(1) T0时刻系统是否安全？**

**(2) P4进程提出资源申请Requets$_4$( 0,0,1,1)，系统是否能满足它？**

（写出完整过程）

| | Allocation<br>A B C D | Need<br>A B C D | Available<br>A B C D |
|---|---|---|---|
| P0 | 0 3 2 1 | 0 1 2 3 | 3 2 2 3 |
| P1 | 0 0 0 0 | 7 5 0 1 | |
| P2 | 3 0 4 2 | 3 5 4 2 | |
| P3 | 3 3 2 0 | 6 5 2 6 | |
| P4 | 0 1 4 1 | 6 6 8 6 | |