# 第三章　处理机调度与死锁

# 第三章 处理机调度与死锁

# 3.1 处理机调度的层次和处理机调度算法目标

## 3.1.1 处理机调度的层次

- 在多道程序系统中，一个作业被提交后必须经过处理机调度后，方能获得处理机执行。
- 处理机调度可分为三个层次：高级调度，中级调度和初级调度。

# 高级调度

高级调度又称为作业调度或长程调度，主要功能是根据某种算法，把外存上处于后备对列中的那些作业调入内存，调度的对象是作业。

高级调度主要用于多道批处理系统，分时系统和实时系统中不设置高级调度。

## 低级调度(Low Level Scheduling)

- 低级调度称为进程调度或短程调度，它所调度的对象是进程。
- 低级调度用于决定就绪队列中的哪个进程应获得处理机，然后再由分派程序执行把处理机分配给该进程的具体操作。

低级调度是最基本的一种调度，不能缺少。

## 中级调度(Intermediate-Level Scheduling)

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

### 3.1.2 处理机调度算法的目标

1 共同目标

资源利用率、公平性、平衡性、策略强制执行

2 批处理系统的目标

平均周转时间短、系统吞吐量高、处理机利用率高

3 分时系统目标

响应时间快、均衡性

4 实时系统

截止时间的保证、可预测性

# 周转时间

周转时间短

**周转时间**=完成时间—到达时间（提交时间）

=服务时间+等待时间

相对等待时间短

| 进程名 | 服务时间 | 等待时间 |
| --- | --- | --- |
| P1 | 1 | 10 |
| P2 | 100 | 100 |

带权周转时间

**带权周转时间**=周转时间/服务时间

=（服务时间+ 等待时间）/服务时间

=1+等待时间/服务时间

# 3.2 作业与作业调度

## 3.2.1 作业和作业布

什么是作业：是用户在一次解题或一个事务处理过程中要求计算机系统所做工作的集合。

它包括用户程序、所需要的数据及控制命令等。作业是由一系列有序的作业步组成的。一个作业由3部分组成，即程序、数据及作业说明书。其中，作业说明书体现了用户对作业的控制意图。

# 作业步：

每个作业必须经过的若干个相对独立又相互关联的顺序加工步骤，每一个步骤称为一个作业步。

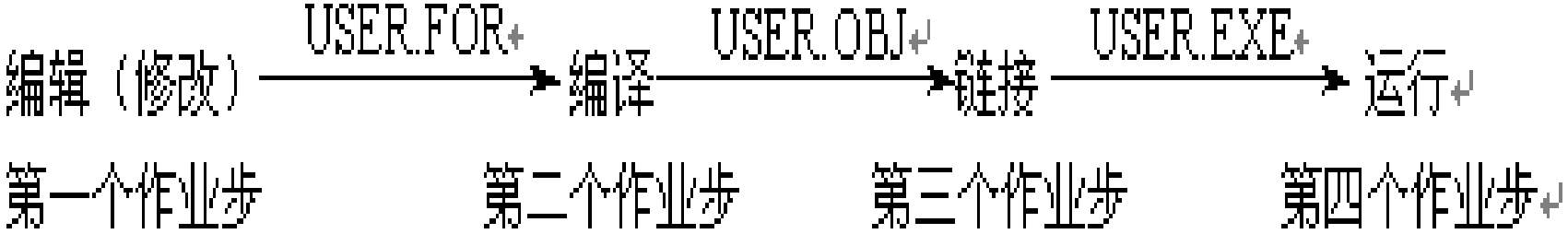



图 2.1　作业步之间的关系

# 作业控制块（JCB，Job Control Block）

- 每个作业进入系统时由系统为其建立一个作业控制块JCB（Job Control Block），它是存放作业控制和管理信息的数据结构，主要信息见右图。

| 作业名 | |
|---|---|
| 资源要求 | 估计运行时间 |
| | 最迟完成时间 |
| | 要求的内存量 |
| | 要求外设的类型及台数 |
| | 要求文件量和输出量 |
| 资源使用情况 | 进入系统的时间 |
| | 开始运行的时间 |
| | 已运行的时间 |
| | 内存地址 |
| | 外设台号 |
| 类型 | 控制方式 |
| | 作业类型 |
| 优先级 | |
| 状态 | |

# 3.2.2 作业调度的主要任务

**作业调度的功能：**

1.记录进入系统的各作业情况

1) 建立相应JCB（作业控制块）
2) 组成后备作业队列
3) 作业完成时，撤消JCB

- JCB的作用：作业调度和资源分配的依据。

# 作业调度的功能

2. 策略：按一定的策略，从后备作业队列中挑选一个或几个作业投入运行 。
3. 为选中的作业分配资源（如：内存、外设）
4. 作业运行结束作善后处理
5. 模块功能实现：创建一个进程

# 作业调度的任务

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

2) 接纳哪些作业

# 3.2.2 作业调度算法

- 先来先服务（FCFS）
- 短作业优先（SJF）
- 优先级调度算法（PSA）
- 高响应比优先（HRRN）

# 3.2 进程调度

## 3.2.1 进程调度的任务、机制和方式

任务：保存处理机现场信息→按照某种算法选取进程→把处理机分配给进程。

机制：三个做成部分：排队器、分派器和上下文切换器。

## 进程调度方式：

1) 非抢占方式(Non-preemptive Mode)

在采用非抢占调度方式时，可能引起进程调度的因素可归结为这样几个：

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；

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操作(wait操作)、Block原语、Wakeup原语等。

2）抢占方式：

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。

抢占方式是基于一定原则的：

- 优先权原则；
- 短进程优先原则；
- 时间片原则。

# 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

| 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 |

## 2. 短作业(进程)优先调度算法

短作业(进程)优先调度算法SJ(P)F，是指对短作业或短进程优先调度的算法。它们可以分别用于作业调度和进程调度。短作业优先(SJF)的调度算法，是从后备队列中选择一个或若干个估计运行时间最短的作业，将它们调入内存运行。而短进程优先(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。

SJ(P)F调度算法也存在不容忽视的缺点：

(1) 该算法对长作业不利，如作业C的周转时间由10增至16，其带权周转时间由2增至3.1。更严重的是，如果有一长作业(进程)进入系统的后备队列(就绪队列)，由于调度程序总是优先调度那些(即使是后进来的)短作业(进程)，将导致长作业(进程)长期不被调度。

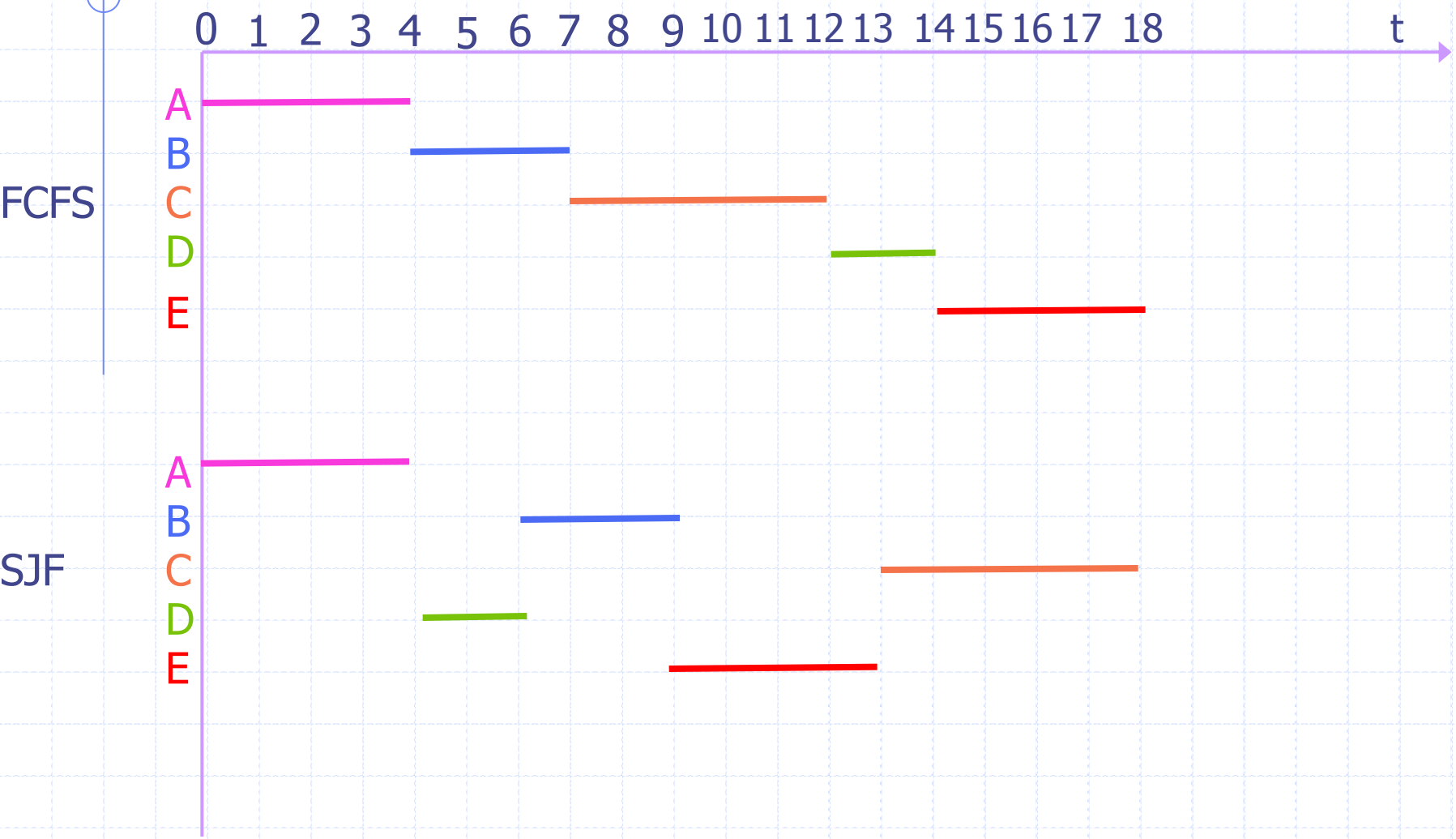
(2) 该算法完全未考虑作业的紧迫程度，因而不能保证紧迫性作业(进程)会被及时处理。

(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。

| 调度算法 \ 作业情况 | 进程名 | A | B | C | D | E | 平　均 |
|---|---|---|---|---|---|---|---|
| | 到达时间 | 0 | 1 | 2 | 3 | 4 | |
| | 服务时间 | 4 | 3 | 5 | 2 | 4 | |
| FCFS (a) | 完成时间 | 4 | 7 | 12 | 14 | 18 | |
| | 周转时间 | 4 | 6 | 10 | 11 | 14 | 9 |
| | 带权周转时间 | 1 | 2 | 2 | 5.5 | 3.5 | 2.8 |
| SJF (b) | 完成时间 | 4 | 9 | 18 | 6 | 13 | |
| | 周转时间 | 4 | 8 | 16 | 3 | 9 | 8 |
| | 带权周转时间 | 1 | 2.67 | 3.1 | 1.5 | 2.25 | 2.1 |

图 3-4 FCFS和SJF调度算法的性能

# 时间图



| 作业 | 进入时间 | 估计运行时间（分钟） | SJF 完成时刻 | FCFS 完成时刻 |
|---|---|---|---|---|
| JOB1 | 8:00 | 120 | 10:00 | 10:00 |
| JOB2 | 8:50 | 50 | 11:20 | 10:50 |
| JOB3 | 9:00 | 10 | 10:10 | 11:00 |
| JOB4 | 9:50 | 20 | 10:30 | 11:20 |

- FCFS : T= ( 120+120+120+90 ) / 4 = 112.5
- SJF: T=（120+150+70+40）/4 = 95
- FCFS：

  W=（120/120 + 120/50 +120/10 +90/20）/4=4.975
- SJF:

  W=（120/120 + 150/50 + 70/10 +40/20） /4=3.25

# 常用作业调度算法

- 先来先服务和短作业优先算法都有其片面性：
- 先来先服务调度算法只考虑作业的等待时间，而忽视了作业的运行时间
- 短作业优先算法则相反，只考虑了作业的运行时间，而忽视了作业的等待时间。
- 高响应比优先调度算法是介于这两种算法之间的一种折衷的算法。

# 高优先权优先调度算法

## 1 优先级调度算法

- 优先级调度算法（priority-scheduling algorithm）是指每个进程都有一个优先级与其相关联，具有最高优先级的就绪进程会被分派到CPU。具有相同优先级的进程按FCFS顺序调度。
- 优先级通常为固定区间的数字，如0到7，或者0到4095。不过，对于0是最高还是最低的优先级，并没有定论。有的系统用低数字表示低优先级，而有的系统使用低数字表示高优先级。

例：有5个进程P1、P2、P3、P4、P5，它们同时依次进入就绪队列，它们的优先数和需要的处理机时间如下：

| 进程 | 处理机时间 | 优先数 |
|---|---|---|
| P1 | 10 | 3 |
| P2 | 1 | 1 |
| P3 | 2 | 3 |
| P4 | 1 | 4 |
| P5 | 5 | 2 |

| FCFS等待时间 | 静态优先级法等待时间 |
|---|---|
| 0 | 1 |
| 10 | 18 |
| 11 | 11 |
| 13 | 0 |
| 14 | 13 |

忽略进程调度所花的时间，要求：（1）分别写出采用先来先服务调度算法和静态优先级调度算法中进程的执行次序；（2）分别计算各进程在就绪队列中的等待时间和平均等待时间。

解:(1)采用FCFS调度算法时各进程的执行次序为：P1→P2→P3→P4→P5。

采用静态优先级调度算法时各进程的执行次序为：

P4→P1→P3→P5→P2（假设优先数与优先权成正比）。

(2)FCFS中，平均等待时间＝（0+10+11+13+14）/5＝9.6；

静态优先级法中，平均等待时间＝（1+18+11+0+13）/5＝8.6

# 高优先权优先调度算法

## 2. 优先权调度算法的类型

### 1) 非抢占式优先权算法

在这种方式下，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。这种调度算法主要用于批处理系统中；也可用于某些对实时性要求不严的实时系统中。

## 高优先权优先调度算法

2) 抢占式优先权调度算法

在这种方式下，系统同样是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其优先权更高的进程，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。

显然，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，故而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。

## 3. 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。一般地，优先权是利用某一范围内的一个整数来表示的，例如，0~7或0~255中的某一整数，又把该整数称为优先数。只是具体用法各异：有的系统用“0”表示最高优先权，当数值愈大时，其优先权愈低；而有的系统恰恰相反。

## 2. 优先权的类型

2) 动态优先权

动态优先权是指，在创建进程时所赋予的优先权，是可以随进程的推进或随其等待时间的增加而改变的，以便获得更好的调度性能。

例如，我们可以规定，在就绪队列中的进程，随其等待时间的增长，其优先权以速率a提高或下降。

# 一道考研题（南京大学1999）

三、（共 10 分）某多道程序设计系统配有一台处理器和两台外设 IO1、IO2，现有三个优先级由高到低的作业 J1、J2 和 J3 都以装入了主存，它们使用资源的先后顺序和占用时间分别是：

J1: IO2(30ms),CPU(10ms),IO1(30ms),CPU(10ms)

J2: IO1(20ms),CPU(20ms),IO2(40ms)

J3: CPU(30ms),IO1(20ms)

处理器调度采用可抢占的优先数算法，忽略其他辅助操作时间，回答下列问题：

（1）分别计算作业 J1、J2、J3 从开始到完成所用的时间。

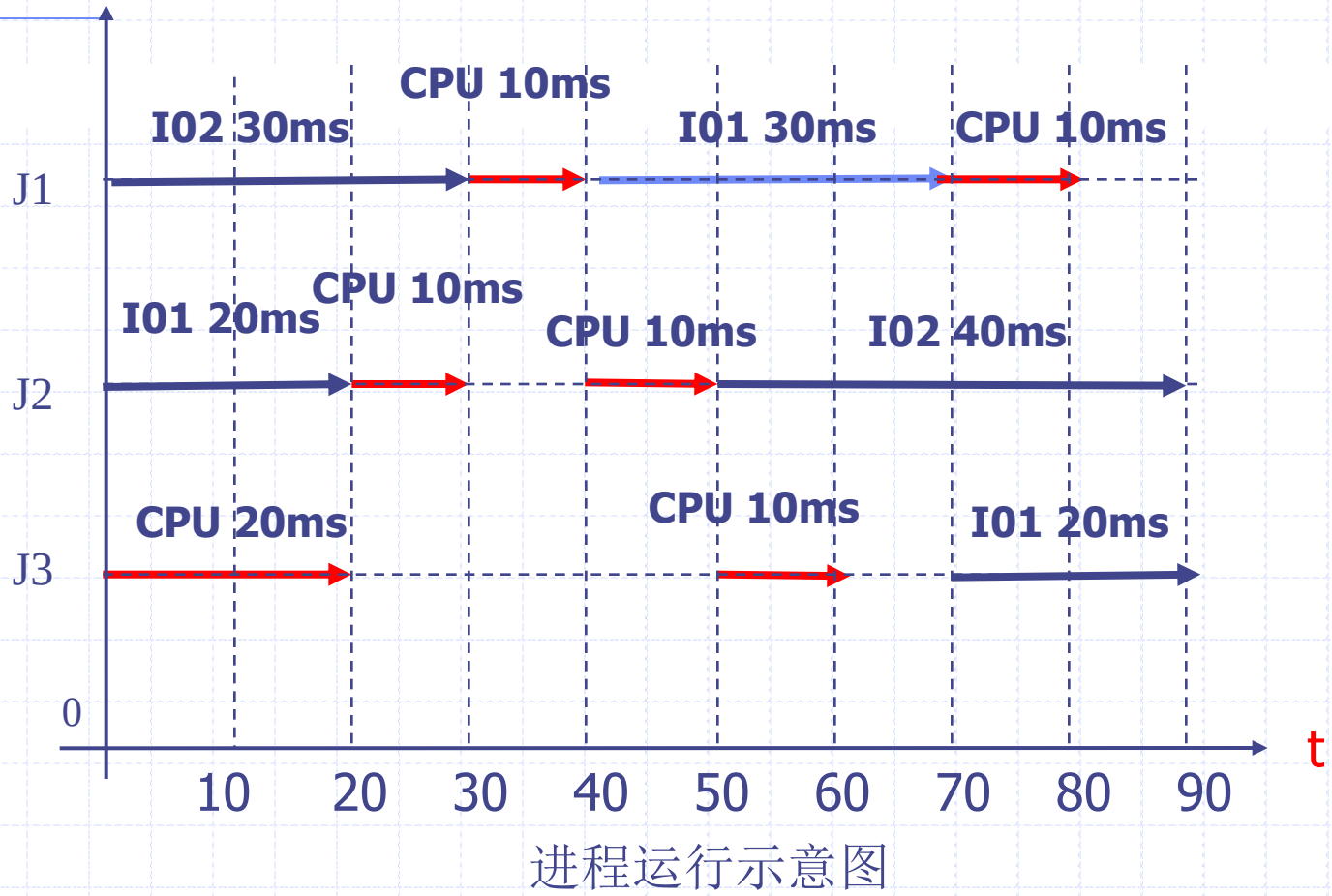
（2）三个作业全部完成时 CPU 的利用率。

（3）三个作业全部完成时 外设 IO1 的利用率。

J1: IO2(30ms),CPU(10ms),IO1(30ms),CPU(10ms)

J2: IO1(20ms),CPU(20ms),IO2(40ms)

J3: CPU(30ms),IO1(20ms)




进程运行示意图

# 解答

- （1）由上图可以看出，J1从开始到完成的时间为0~80ms，J2从开始到完成的时间为0~90ms， J3从开始到完成的时间为0~90ms
- （2）3个作业共需时间为90s，CPU总共时间为：20+10+10+10+10+10=70s

  所以CPU利用率为：70/90=77.8%
- （3）3个作业全部完成是IO1的利用率是
- （20+30+20）/90=70/90=77.8%

# 4. 高响应比优先调度算法

- 该算法，就是每调度一个作业投入运行时，计算后备作业表中每个作业的响应比，然后挑选响应比最高的投入运行。

## 3. 高响应比优先调度算法

(1) 如果作业的等待时间相同，则要求服务的时间愈短，其优先权愈高，因而该算法有利于短作业。

(2) 当要求服务的时间相同时，作业的优先权决定于其等待时间，等待时间愈长，其优先权愈高，因而它实现的是先来先服务。

(3) 对于长作业，作业的优先级可以随等待时间的增加而提高，当其等待时间足够长时，其优先级便可升到很高， 从而也可获得处理机。

# 例题：

- 有一个内存中只能装入两道作业的批处理系统，作业调度采用短作业优先的调度算法，进程调度采用以优先数为基础的抢占式调度算法。有如下表所示的作业序列，表中所列的优先数是指进程调度的优先数，且优先数越小优先级越高。

（1）列出所有作业进入内存的时刻以及结束的时刻

（2）计算作业的平均周转时间

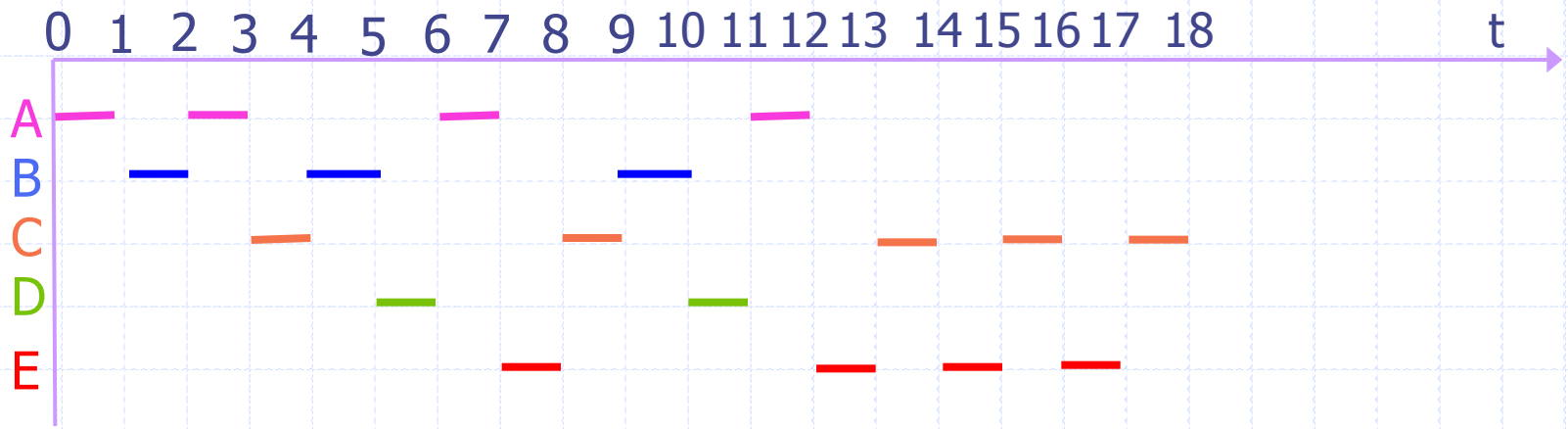
| 作业名 | 到达时间 | 估计运行时间 | 优先数 |
| --- | --- | --- | --- |
| A | 10：00 | 40分 | 5 |
| B | 10：20 | 30分 | 3 |
| C | 10：30 | 50分 | 4 |
| D | 10：50 | 20分 | 6 |

# 基于时间片的轮转调度算法

## 1. 时间片轮转法

在早期的时间片轮转法中，系统将所有的就绪进程按先来先服务的原则，排成一个队列，每次调度时，把CPU分配给队首进程，并令其执行一个时间片。时间片的大小从几ms到几百ms。当执行的时间片用完时，由一个计时器发出时钟中断请求，调度程序便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。这样就可以保证就绪队列中的所有进程，在一给定的时间内，均能获得一时间片的处理机执行时间。

| 调度算法 \ 作业情况 | 进程名 | A | B | C | D | E | 平 均 |
|---|---|---|---|---|---|---|---|
| | 到达时间 | 0 | 1 | 2 | 3 | 4 | |
| | 服务时间 | 4 | 3 | 5 | 2 | 4 | |




就绪队列：

0-A
1-BA
2- ACB
3- CBDA
4- BDAEC
5- DAECB
6- AECBD
7- ECBDA
8- CBDAE
9- BDAEC
10- DAEC
11- AEC
12- EC
13- CE
14- EC
15- CE
16- EC
17- C

# 基于时间片的轮转调度算法

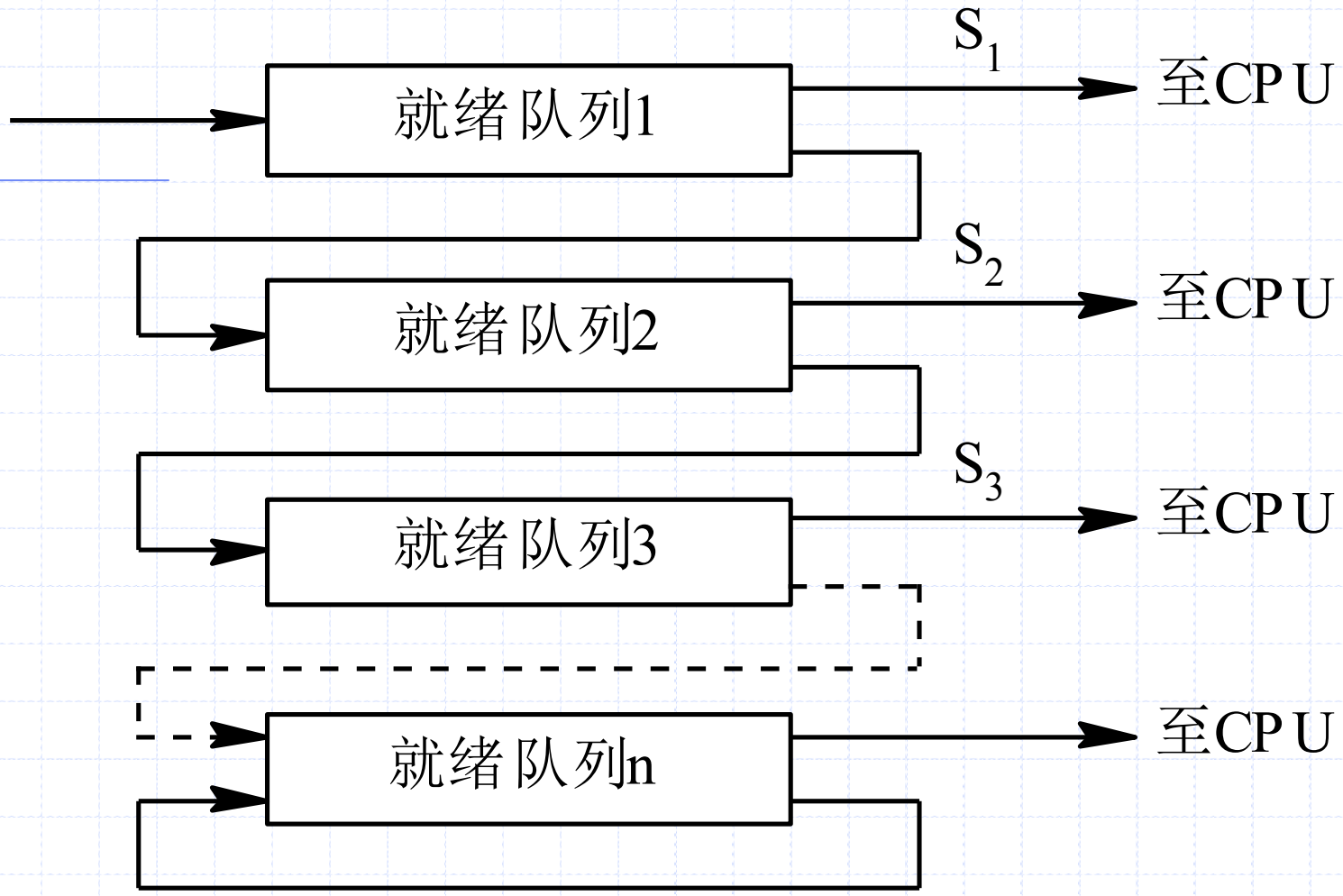
## 1. 时间片轮转法

在时间片轮转算法中，时间片的大小对系统性能有很大的影响，如选择很小的时间片，将很有利于短作业，但会频繁发生中断和进程上下文的切换；反之，如选择太长的时间片，使得每一个进程都能在一个时间片内完成，时间片轮转算法便退化为FCFS算法，无法满足交互式用户的需求。

时间片应略大于一次典型的交互所需要的时间。

## 2. 多级反馈队列调度算法

(1) 应设置多个就绪队列，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，第二个队列次之，其余各队列的优先权逐个降低。该算法赋予各个队列中进程执行时间片的大小也各不相同，在优先权愈高的队列中，为每个进程所规定的执行时间片就愈小。例如，第二个队列的时间片要比第一个队列的时间片长一倍，……，第*i*+1个队列的时间片要比第i个队列的时间片长一倍。 图 3-5 是多级反馈队列算法的示意。




（时间片：$S_1 < S_2 < S_3$）

图 3-5 多级反馈队列调度算法

(2) 当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，便可准备撤离系统；如果它在一个时间片结束时尚未完成，调度程序便将该进程转入第二队列的末尾，再同样地按FCFS原则等待调度执行；如果它在第二队列中运行一个时间片后仍未完成，再依次将它放入第三队列，……，如此下去，当一个长作业(进程)从第一队列依次降到第n队列后，在第n队列中便采取按时间片轮转的方式运行。

(3) 仅当第一队列空闲时，调度程序才调度第二队列中的进程运行； 仅当第1~(i-1) 队列均空时，才会调度第i队列中的进程运行。如果处理机正在第i队列中为某进程服务时，又有新进程进入优先权较高的队列(第1~(i-1)中的任何一个队列)，则此时新进程将抢占正在运行进程的处理机，即由调度程序把正在运行的进程放回到第i队列的末尾，把处理机分配给新到的高优先权进程。

## 3. 多级反馈队列调度算法的性能

(1) 终端型作业用户。

(2) 短批处理作业用户。

(3) 长批处理作业用户。

•有如下进程，

•（1）画出下列调度算法下的调度时间图：FCFS、抢占式\非抢占式SPF、抢占式\非抢占式HPF、HRRN和RR（q=1,q=2）

（2）对于上述每种算法，各个作业的周转时间是多少？平均周转时间是多少？

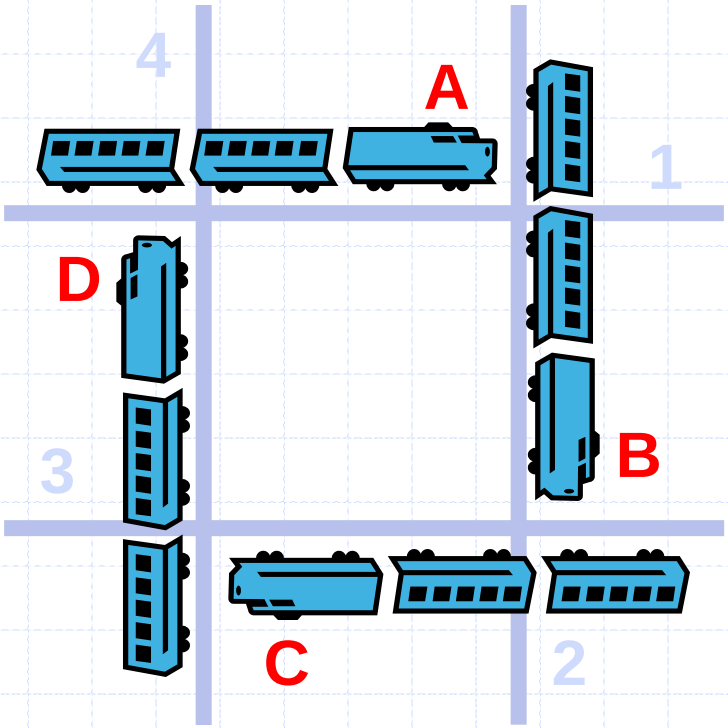
（3）对于上述每种算法，各个作业的带权周转时间和平均带权周转时间各是多少？

| 进程 | 到达时间 | 运行时间 | 优先级 |
|---|---|---|---|
| A | 0 | 3 | 3 |
| B | 1 | 1 | 1 |
| C | 3 | 3 | 3 |
| D | 4 | 1 | 4 |
| E | 5 | 5 | 2 |

# §3.5 死锁概述

## 3.5.1 死锁的概念

1、死锁的例子：



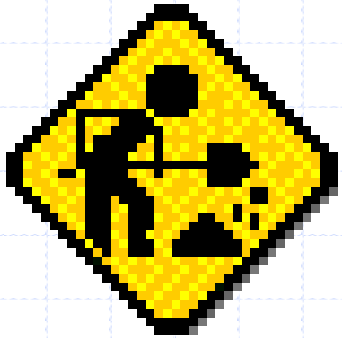

分析：

- 争用了资源：道路
- A占有道路1，又要请求道路2，B占有…
- 形成了无限等待

# 例子：死锁的生活中的影子

A

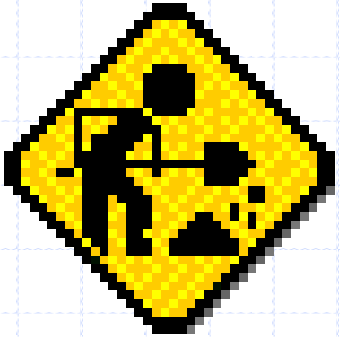
B



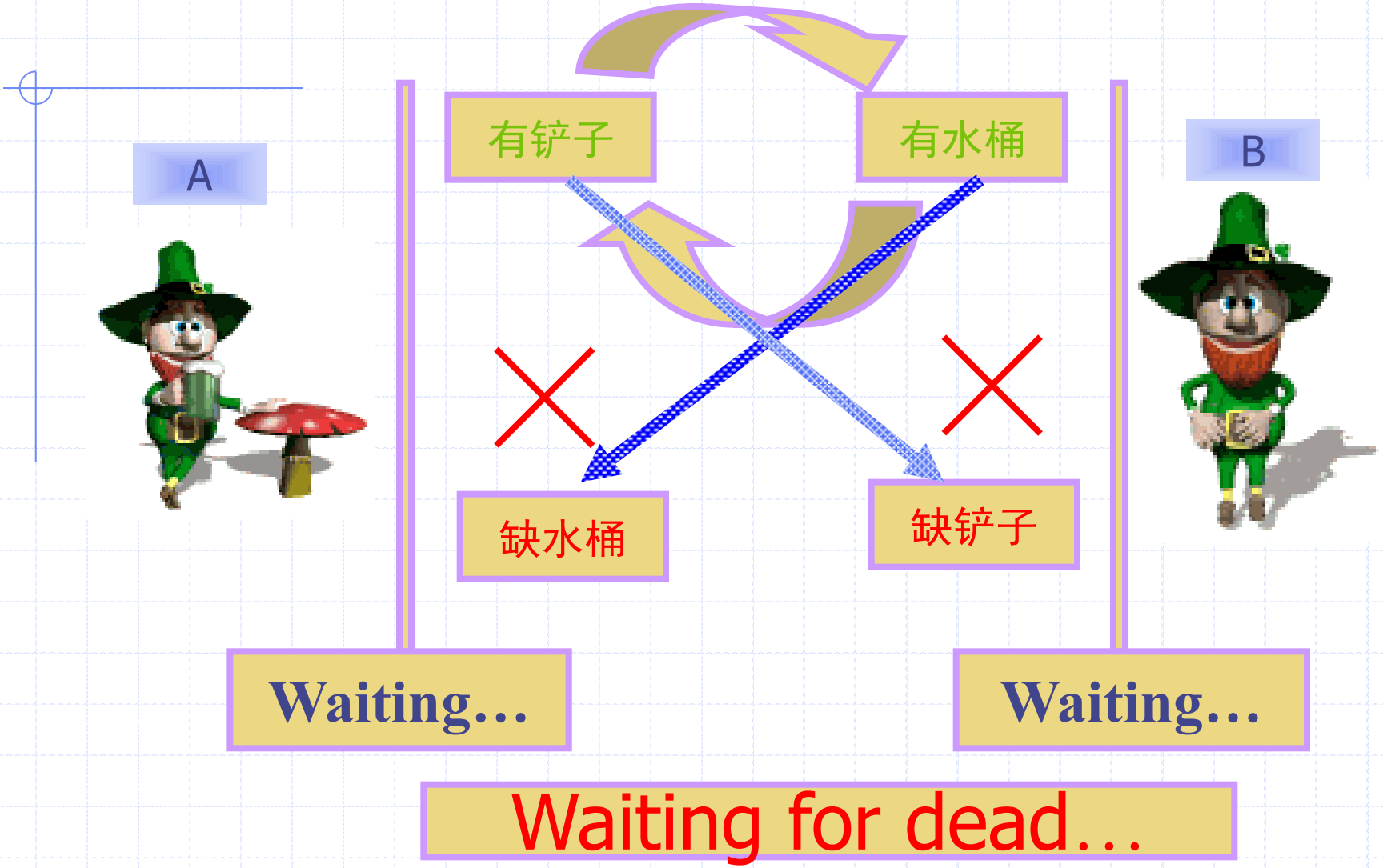
- 假设有这么两个人A，B：地位平等且自私。
- 任务：每个人都独立去植树
- 器具：目前只有1把铲子和1个水桶

# 例子：死锁的生活中的影子

- 要求：每个人若想独立去把植树完成，植树时必须同时具备1把铲子和1个水桶
- 场景：现在，A手中有1把铲子，B手中有1个水桶
- 问题：A、B两人能否分别完成自己的任务呢？



# 分析...



# §3.5 死锁概述

2、死锁的定义——“你不让，我也不让”

如果在一个进程集合中的每个进程都在等待只能由该集合中的其他一个进程才能引发的事件，则称一组进程或系统此时发生了死锁。

3、关于死锁的一些重要结论

（1）参与死锁的进程数至少为2

（2）参与死锁的所有进程均等待资源

（3）参与死锁的进程至少有两个占有资源

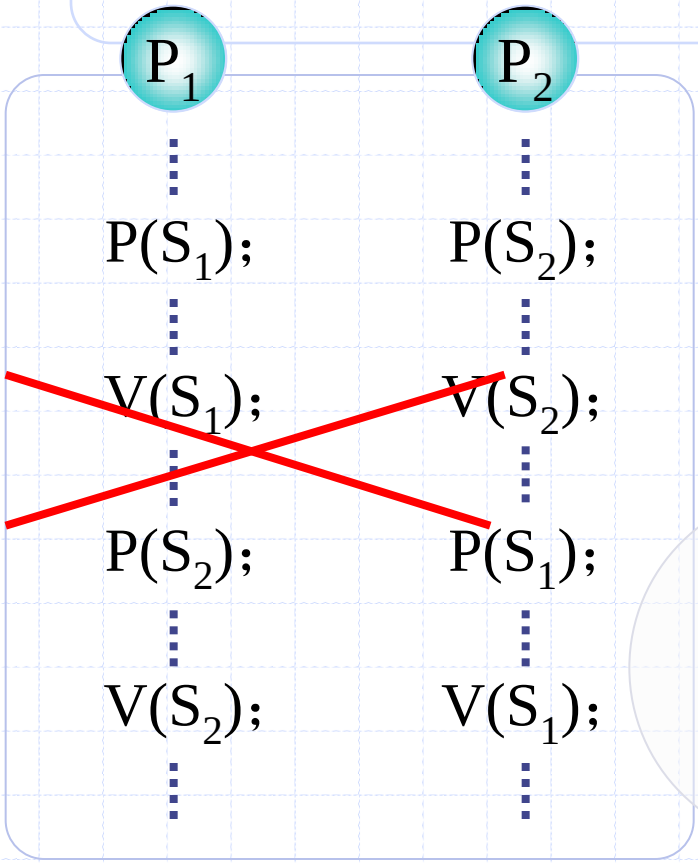
（4）参与死锁的进程是系统中当前正在运行进程的一部分。

操作系统中的例子

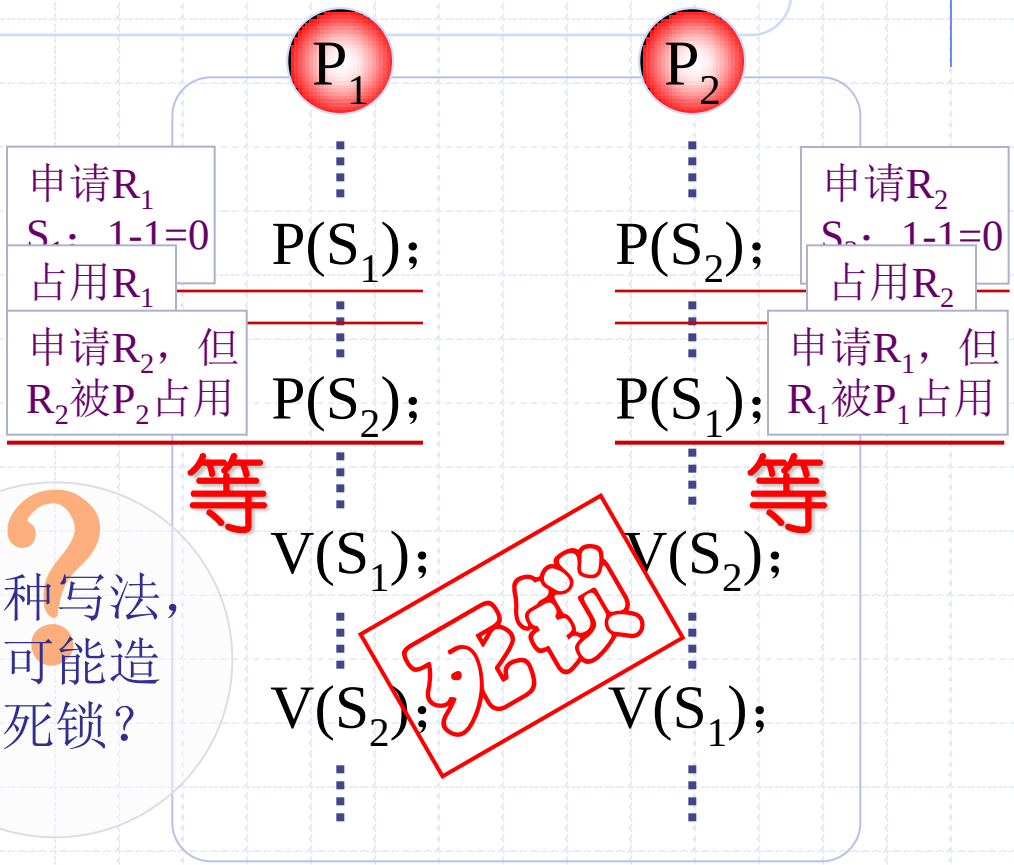
Hold and wait

$_1$、$P_2$，两个设备打印机$R_1$，读卡机$R_2$。

设$S_1$=1，打印机可用。$S_2$=1，读卡机可用。

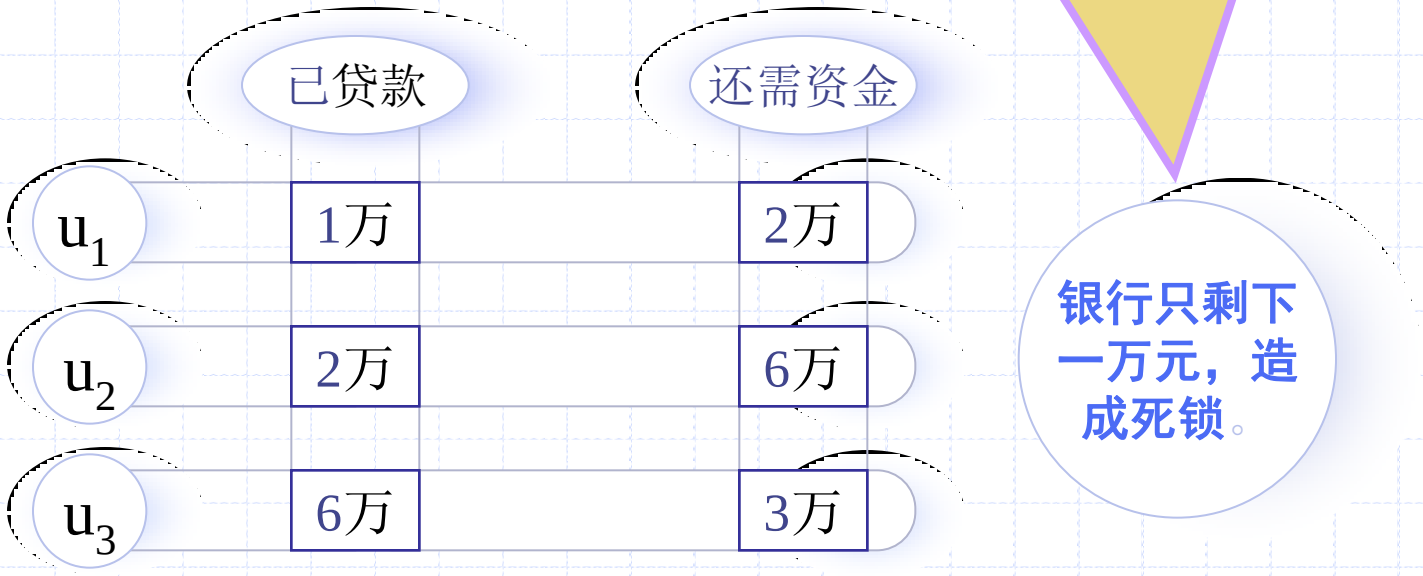



两种写法，谁可能造成死锁？



2. 银行家问题

- 银行共[illegible]客户$u_2$需贷款6万，客户$u_3$需[illegible]。某一时刻：




•对于客户来说，只有所需要的所有贷款全部得到满足，这样生意才能完成，之后才能把所贷款项归还。
•贷不到款，生意做不成，客户死
•贷出款得不到归还，破产，银行家死

# §3.5 死锁

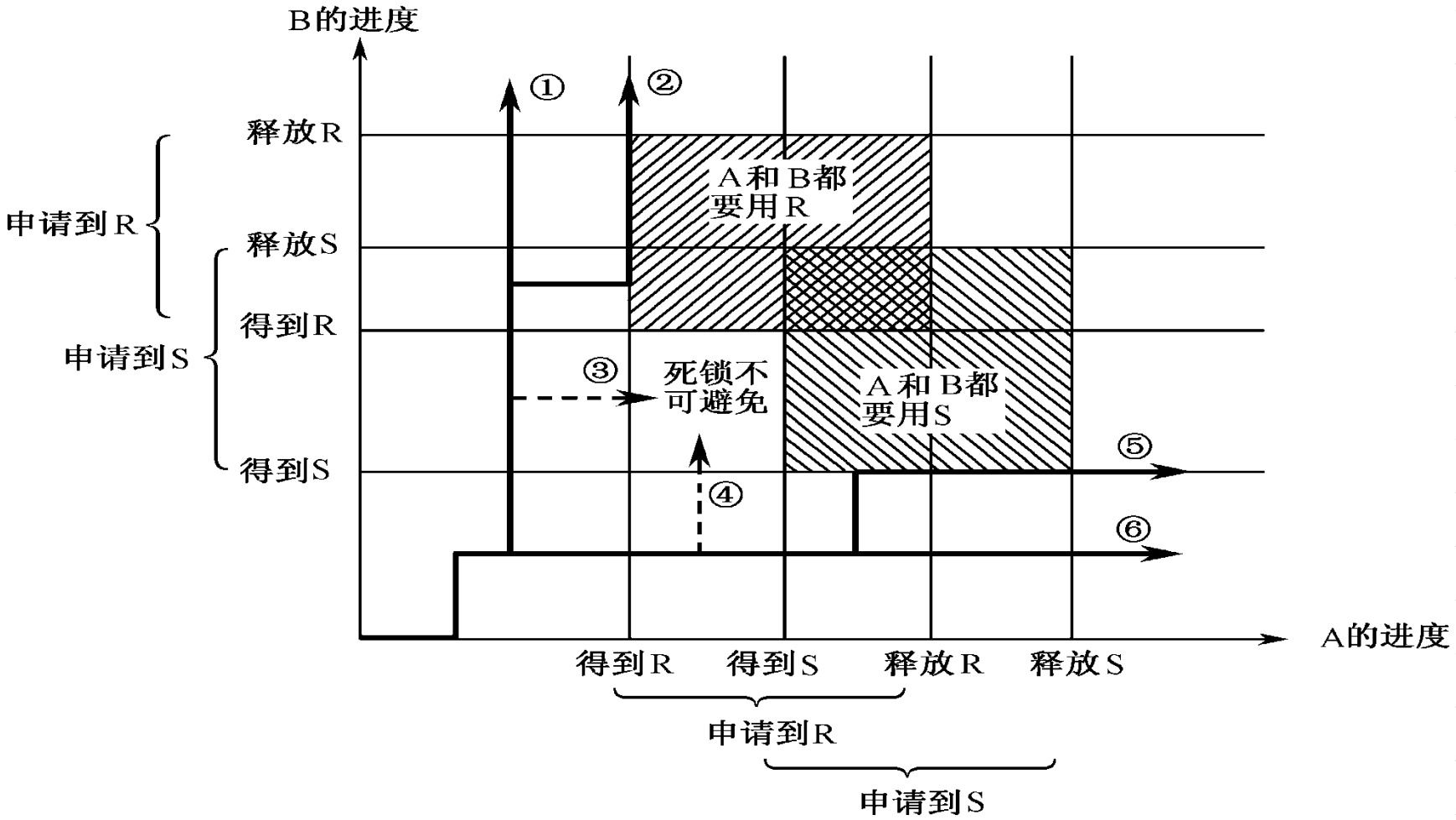
## 3.5.2 死锁产生的原因和必要条件

1. 死锁产生的原因:

（1）资源有限，竞争资源（不可抢占性资源和可消耗资源）。当系统中多个进程共享资源，如打印机、公用队列等，其数目不足以满足诸进程的需要，会引起进程对资源的竞争而产生死锁。

（2）并发进程间的推进顺序不当。进程在运行过程中，请求和释放资源的顺序不当，也会导致产生进程死锁。

# §3.5 死锁


B的进度
①
②
释放R
申请到R
释放S
A和B都
要用R
得到R
申请到S
③
死锁不
可避免
A和B都
要用S
⑤
得到S
④
⑥
A的进度
得到R
得到S
释放R
释放S
申请到R
申请到S

# §3.5 死锁

**注意：**

如果不考虑资源分配的合理性，若要不产生死锁，则资源的个数必须满足以下条件（**即系统不会产生死锁的最小资源数**）：

设系统所拥有的资源总数为M，共享该资源的进程数为P，每个进程所需使用该资源的最大需求为N，则

$M \geq P*(N-1)+1$ **时**

无论如何分配都不会产生死锁。

# §3.5 死锁

思考与练习：

1、一个操作系统有20个进程，竞争是用65个同类资源，申请方式是逐个进行的，一旦某进程获得它所需要的全部资源，立即归还所有资源。每个进程最多使用3个资源。若仅考虑这类资源，该系统有无可能发生死锁，为什么？（北京大学1995年试题）

2、一台计算机有8台磁带机，它们由N个进程竞争使用，每个进程可能需要3台磁带机，请问当N为多少时，系统没有死锁的危险，并说明原因。（上海交通大学1999年试题）

# §3.5 死锁

2. 死锁产生的必要条件：

- **互斥条件**：涉及的资源是非共享的。
- **不剥夺条件**：不能强行剥夺进程拥有的资源。
- **请求和保持条件（部分分配条件）**：进程在等待一新资源时继续占有已分配的资源。
- **环路条件**：存在一种进程的循环链，链中的每一个进程已获得的资源同时被链中的下一个进程所请求。

# 死锁产生的条件

- 以上前三个条件
  - 互斥条件（mutual exclusion）
  - 占有和等待条件（hold and wait）
  - 不剥夺条件（no preemption）

是死锁存在的必要条件（necessary condition），但不是充分条件。

- 也就是说发生了死锁，必然具备互斥、占有和等待、不剥夺等条件。

而具备互斥、占有和等待、不剥夺等条件却不一定一定产生死锁。

# 死锁产生的条件

- 第四个条件：
  循环等待条件（circular wait）
  是前三个条件同时存在时产生的结果，所以，这些条件并不完全独立。
- 但单独考虑每个条件是有用的，**只要能破坏这四个必要条件之一，死锁就可防止**。

# §3.5 死锁

## 3.5.3 处理死锁的方法

- **鸵鸟算法**：指像鸵鸟一样对死锁视而不见，即不理睬死锁。
- **预防死锁**：指通过设置某些限制条件，去破坏产生死锁的四个必要条件中的一个或几个条件，来防止死锁的发生。
- **避免死锁**：指在资源的动态分配过程中，用某种方法去防止系统进入不安全状态，从而避免死锁的发生。
- **检测死锁**：允许系统在运行过程中发生死锁，但可设置检测机构及时检测死锁的发生，并采取适当措施加以清除。
- **解除死锁**：当检测出死锁后，便采取适当措施将进程从死锁状态中解脱出来。

# §3.6 预防死锁

死锁的预防—— 破坏死锁的四个必要条件

（1）破坏互斥条件：即允许多个进程同时访问资源。但由于资源本身固有特性限制，有的资源根本不能同时访问，只能互斥访问，所以破坏互斥条件来预防死锁，这不可行。

（2）破坏请求和保持条件：A、可采用预先静态分配方法，即要求进程在运行之前一次申请它所需要的全部资源，在它的资源未满足前，不把它投入运行。一旦运行后，这些资源全归其占有，同时它也不再提出其它资源要求，这样可以保证系统不会发生死锁。B、释放已用完的资源再申请新的资源。

# §3.6 预防死锁

（3）**破坏不可剥夺条件**：即一个已经获得某些资源的进程，若又请求新的资源时不能得到满足，则它必须释放出已获得的所有资源，以后需要资源时再请求。也即一个进程已获得的资源在运行过程中可被剥夺。从而破坏了该条件。

（4）**破坏环路条件**：可采用有序资源分配方法，即将系统中的所有资源都按类型赋予一个编号，要求每一个进程均严格按照编号递增的次序来请求资源，同类资源一次申请完。也就是，只要进程提出请求资源Ri，则在以后的请求中，只能请求Ri后的资源，这样不会出现几个进程请求资源而形成环路。

# §3.7 避免死锁

1、**死锁避免定义**：在系统运行过程中，对进程发出的每一个系统能够满足的资源申请进行动态检查，并根据检查结果决定是否分配资源，若分配后系统可能发生死锁，则不予分配，否则予以分配

2、**安全状态**：如果系统能按某种顺序（如P4，P1，…，Pn，称为安全序列）为每个进程分配其所需的资源，直至所有进程都能运行完成，称系统处于安全状态。若不存在这样一个安全序列称系统处于不安全状态。

# §3.7 避免死锁

**安全状态举例：**

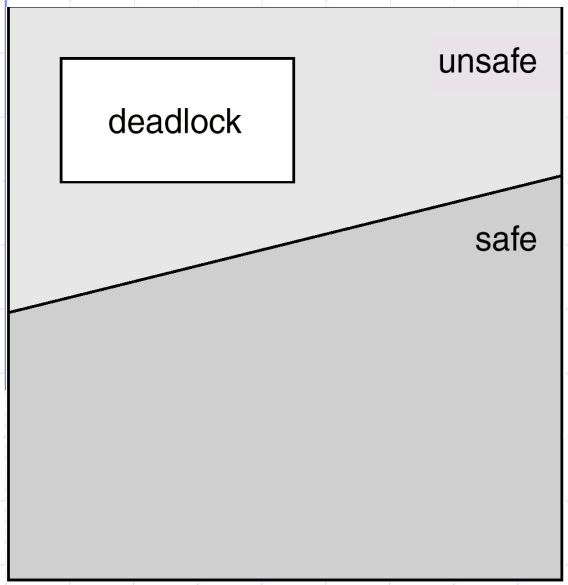
有三个进程p1, p2, p3，有12台磁带机。P1共要求10台，P2共要求4台，P3共要求9台。在T0时刻，p1, p2, p3分别获得5、2、2台，尚有3台空闲。

| 进程 | 最大需求 | 已分配 | 还需 | 可用 |
|---|---|---|---|---|
| p1 | 10 | 5 | 5 | 3 |
| p2 | 4 | 2 | 2 | |
| p3 | 9 | 2 | 7 | |

经分析，在T0时刻，系统是安全的。因为存在一个安全序列<p2、p1、p3>

# §3.7 避免死锁

## 安全、不安全、死锁状态的关系




（1）如果一个系统在安全状态，就没有死锁。

（2）如果一个系统处于不安全状态，就有可能死锁。

**注意：**在死锁避免的方法中，允许进程动态申请资源，系统在进行资源分配之前，先计算资源分配的安全性，若此次分配不会导致系统进入不安全状态，便将资源分配给进程，否则进程等待。

# §3.7 避免死锁

## 3、银行家算法

- **银行家算法的实质：**要设法保证系统动态分配资源后不进入不安全状态，以避免可能产生的死锁。即每当进程提出资源请求且系统的资源能够满足该请求时，系统将判断如果满足此次资源请求，系统状态是否安全。如果判断结果为安全，则给该进程分配资源，否则不分配资源，申请资源的进程将阻塞，直到其他进程释放出足够资源为止。
- **银行家算法执行的前提条件：**即要求进程必须预先提出自己的最大资源请求数量，这一数量不可能超过系统资源的总量，系统资源的总量是一定的。

# §3.7 避免死锁

银行家算法的数据结构：

- **可用资源向量Available**：长度为m的向量，表示系统中各类资源的当前可用实例数。
- **最大需求矩阵Max**：n×m矩阵，定义每个进程对各类资源的最大需求量。
- **已分配资源矩阵Allocation**：n×m矩阵，定义了每个进程现在所分配的各种资源类型的实例数。
- **需求矩阵Need**：n×m矩阵，表示每个进程还需要的各类资源的数目。

Need[i,j]=Max[i,j] - Allocation[i,j]，因而，这些数据结构的大小和值会随着时间而改变。

# §3.7 避免死锁

- 安全检查算法：

1）令Work和Finish分别表示长度为m和n的向量。按如下方式进行初始化：

Work= Available，Finish[i]=false（i=1，2，…，n）。

2）搜寻满足下列条件的*i*值：

Finish[i] = =false，且Need[i]≤Work。如果没有这样的i存在，则转向步骤④。

3）修改数据值，并返回步骤②

Work=Work + Allocation[i]（Pi释放所占的全部资源）

Finish[i]=true

4）若Finish[i] = =true对所有i都成立，则系统处于安全状态；否则，系统处于不安全状态。

# §3.7 避免死锁

- **资源请求算法**

设$Request_i$是进程Pi的请求向量，设$Request_i[j]=k$，表示进程Pi请求分配Rj类资源k个。当进程Pi 发出资源请求后，系统按如下步骤进行检查：

(1) 如$Request_i[j] \leqslant Need[i,j]$，转(2)；否则出错，因为进程申请资源量超过它申明的最大量。

(2) 如$Request_i[j] \leqslant Available[j]$，转(3)；否则表资源不够，需等待。

# § 3.6 死锁的预防和避免

（3）假设系统可以给进程Pi分配所请求的资源，则应对有关数据结构进行修改：

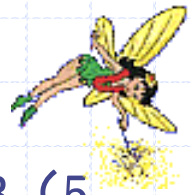
**Available = Available – Request[i];**

**Allocation[i] = Allocation[i] + Request[i];**

**Need[i] = Need[i] – Request[i];**

（4） 系统执行安全性算法，查看此时系统状态是否安全。如果是安全的，就实际分配资源，满足进程Pi 的此次申请；否则，若新状态是不安全的，则Pi等待，对所申请资源暂不予分配，并且把资源分配状态恢复成③之前的情况。

# 银行家算法的例子



假定系统中有5个进程P0到P4，3类资源及数量分别为A(10个)，B（5个），C（7个），T0时刻的资源分配情况。

表1 T0时刻的资源分配表

| | 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 | |

## (1) T0时刻的安全性

利用安全性算法对T0时刻的资源分配情况进行分析，可得下表。

表2 T0时刻的安全性检查表

| | Work<br>A B C | Need<br>A B C | Allocation<br>A B C | Work+ Allocation<br>A B C | Finish |
|---|---|---|---|---|---|
| $P_1$ | 3 3 2 | 1 2 2 | 2 0 0 | 5 3 2 | true |
| $P_3$ | 5 3 2 | 0 1 1 | 2 1 1 | 7 4 3 | true |
| $P_4$ | 7 4 3 | 4 3 1 | 0 0 2 | 7 4 5 | true |
| $P_2$ | 7 4 5 | 6 0 0 | 3 0 2 | 10 4 7 | true |
| $P_0$ | 10 4 7 | 7 4 3 | 0 1 0 | 10 5 7 | true |

分析得知： T0时刻存在着一个安全序列{P1 P3 P4 P2 P0}，故系统是安全的。

## (2) P1请求资源 $Request_1(1,0,2)$

P1发出请求向量$Request_1(1,0,2)$，系统按银行家算法进行检查:

1) $Request_1(1,0,2) \leqslant Need_1\ (1,2,2)$

2) $Request_1\ (1,0,2) \leqslant Available\ (3,3,2)$

3) 系统试为P1分配资源,并修改相应的向量(见下表3所示)

Avaliable,Need,Allocation

# 表3 P1请求资源时的资源分配表

| | 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<br>(2 3 0) |
| P1 | 3 2 2 | 2 0 0<br>(3 0 2) | 1 2 2<br>(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 | |

4) 利用安全性算法检查资源分配后此时系统是否安全. 如表4

# 表4　P1请求资源时的安全性检查表


| | Work<br>A B C | Need<br>A B C | Allocation<br>A B C | Work+ Allocation<br>A B C | Finish |
|---|---|---|---|---|---|
| $P_1$ | 2 3 0 | 0 2 0 | 3 0 2 | 5 3 2 | true |
| $P_3$ | 5 3 2 | 0 1 1 | 2 1 1 | 7 4 3 | true |
| $P_4$ | 7 4 3 | 4 3 1 | 0 0 2 | 7 4 5 | true |
| $P_2$ | 7 4 5 | 6 0 0 | 3 0 2 | 10 4 7 | true |
| $P_0$ | 10 4 7 | 7 4 3 | 0 1 0 | 10 5 7 | true |

由安全性检查分析得知：　此时刻存在着一个安全序列{P1 P3 P4 P2 P0}，故系统是安全的，可以立即将P1所申请的资源分配给它。

# §3.8 死锁的检测和解除

解决死锁问题的一条途径是死锁检测和解除，这种方法对资源的分配不加任何限制，也不采取死锁避免措施，但系统定时地运行一个“死锁检测”程序，判断系统内是否已出现死锁，如果检测到系统已发生了死锁，再采取措施解除它。

# §3.8 死锁的检测和解除

## 3.8.1 死锁的检测

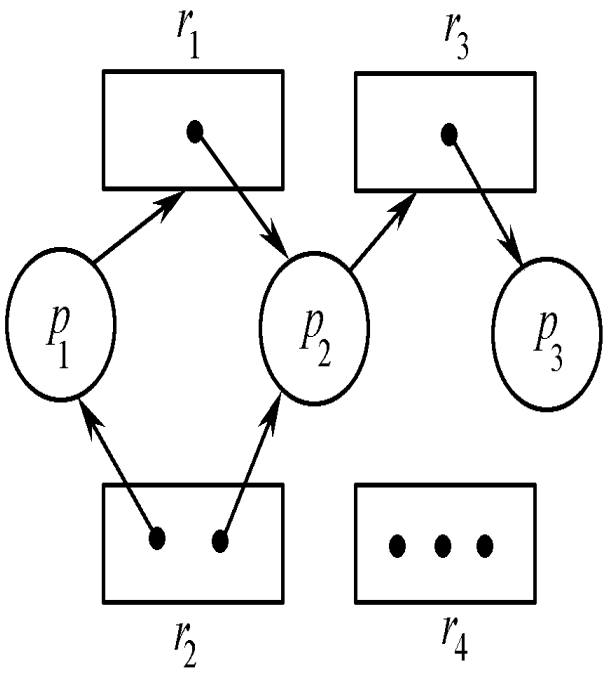
**1、死锁的描述——资源分配图**

- 系统资源分配图是一个有向图，该图有一个节点的集合V和一个边的集合E组成。节点集合V分成两种类型的节点$P=\{p1, p2, \cdots, pn\}$，它由系统中所有活动进程组成；$R=\{r1, r2, \cdots, rm\}$，它由系统中全部资源类型组成。
- 从进程$pi$到资源类型$rj$的有向边记为$pi \rightarrow rj$，它表示进程$pi$申请了资源类型$rj$的一个实例，并正在等待资源。从资源类型$rj$到进程$pi$的有向边记为$rj \rightarrow pi$，它表示资源类型的一个实例$rj$已经分配给进程$pi$。

  **有向边$pi \rightarrow rj$称为申请边，而有向边$rj \rightarrow pi$称为分配边。**

# §3.8 死锁的检测和解除

- 在资源分配图中，通常用圆圈表示每个进程，用方框表示每种资源类型。由于同一资源类型可能有多个实例，所以在矩形中用圆点数表示实例数。




（1）集合P、R和E：

P={P1,P2,P3}

R={r1,r2,r3,r4}

E={p1→r1, p2→r3, r1→p2, r2→p2, r2→p1, r3→p3}

（2）资源实例

资源类型r1有1个实例，资源类型r2有2个实例，资源类型r3有1个实例，资源类型r4有3个实例

# §3.8 死锁的检测和解除

## 2、死锁定理

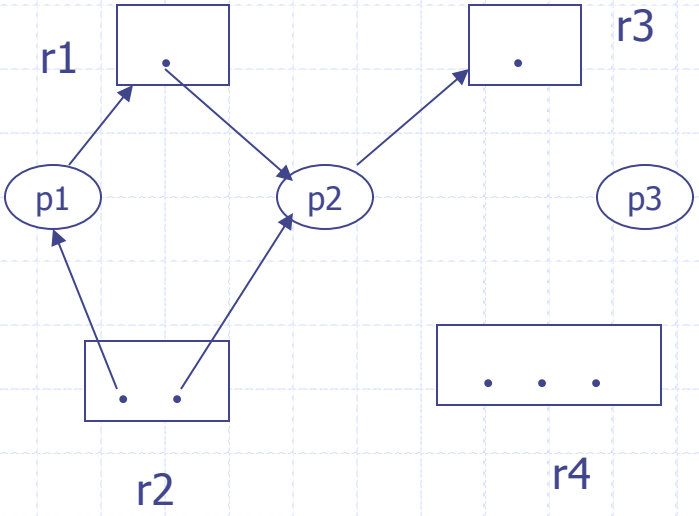
资源分配图的化简方法如下：

（1）寻找一个即不阻塞又非孤立的进程结点Pi，若无则算法结束；

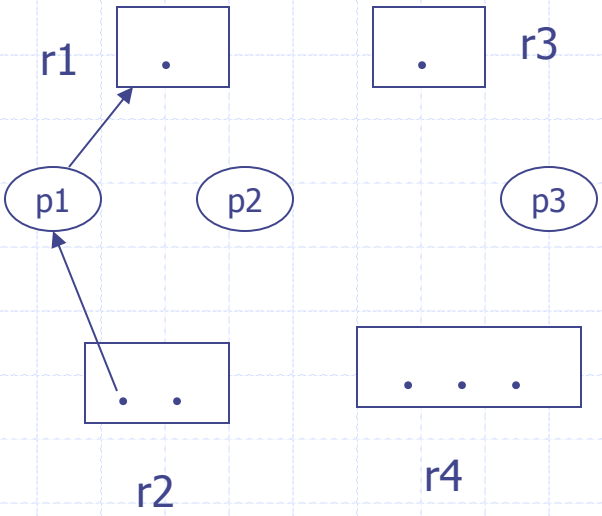
（2）去除Pi的所有分配边和请求边，使Pi成为一个孤立节点；

（3） 转步骤（1）。

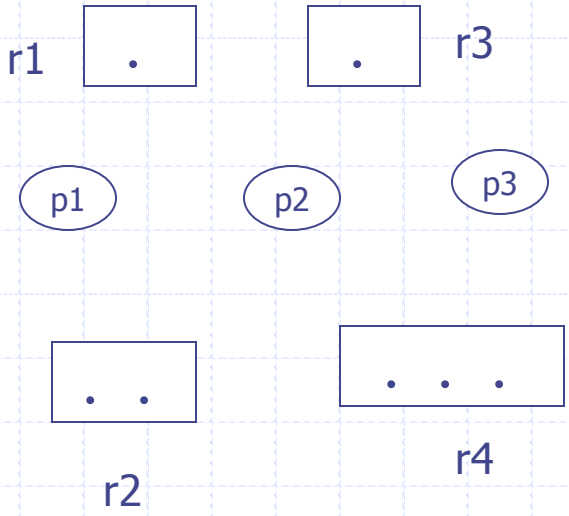
在进行一系列化简后，若能消去图中所有的边，使所有进程都成为孤立结点，则称该图是**可完全简化的**；反之，称该图是**不可完全简化的**。




(a)



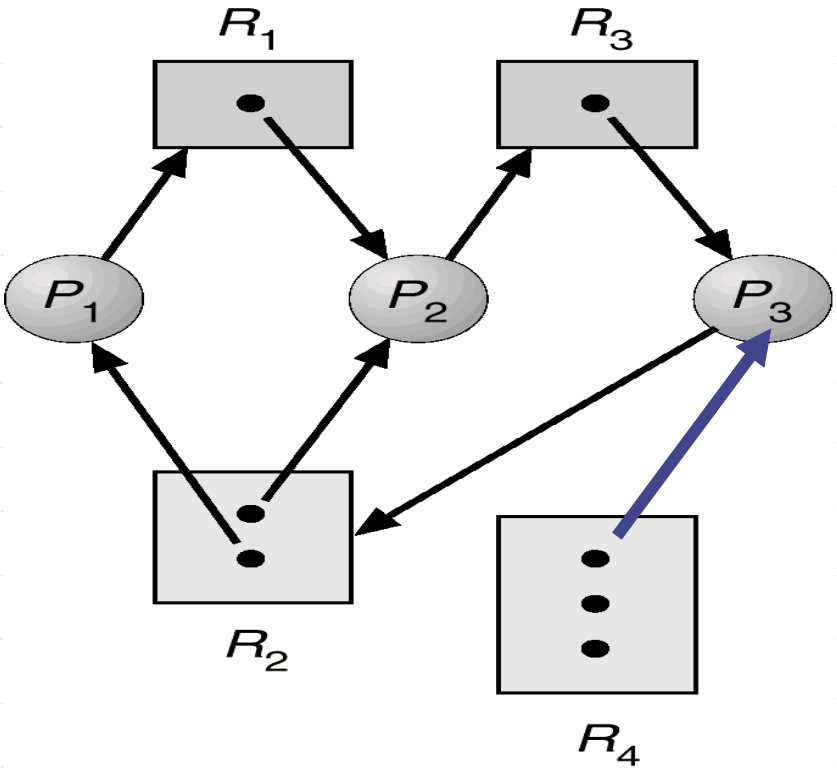

(b)




(c)

# §3.8 死锁的检测和解除

又如下图：该图是不可完全简化的




死锁定理：

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。

# §3.8 死锁的检测和解除

## 3.8.2 死锁的解除

通过抢占资源实现恢复和通过杀掉进程解除死锁。

1. 通过抢占资源实现恢复

即临时性地把资源从当前占有它的进程那里拿过来，分给另外某些进程，直至死锁环路被打破。

2. 通过杀掉进程实现恢复

- 终止所有的死锁进程。
- 一次终止一个进程，直至消除死锁环路。

| 策略 | 死锁预防 | | | 死锁避免 | 死锁检测和恢复 |
|---|---|---|---|---|---|
| | 很保守；对资源不做调配使用 | | | 介于预防和检测方法之间，安全状态下才分配 | 非常开放；申请资源就分配，但定期检测死锁 |
| 采用的不同方式 | 一次性分配所有资源 | 抢占式分配资源 | 资源编号，按序分配 | 至少应找出一个安全序列 | 定期调用检测算法，查看是否出现死锁 |
| 主要优点 | ·适用于执行单一突发活动的进程<br>·不需要抢占 | ·适用于资源状态便于保存和恢复的情况 | ·由于系统设计时已解决问题，不需要运行时计算 | 不需要抢占 | ·从来不延误进程的开始执行<br>·便于联机处理 |
| 主要缺点 | ·效率低<br>·延误进程的开始执行 | ·抢占动作比实际需要的次数更多<br>·易出现环路重启 | ·不允许增加对资源的申请 | ·必须知道以后对资源的申请情况<br>·进程可能被阻塞很长时期 | 丧失固有的抢占性 |

# 本讲完毕