课程主要内容

- 操作系统引论（第1章）
- 进程管理（第2-3章）
- 存储器管理（第4-5章）
- 设备管理（第6章）
- 文件管理（第7-8章）
- 操作系统接口（第9章）
- Unix操作系统

Process Management 进程管理

- 进程的基本概念与控制
  - 进程的基本概念
  - 进程控制
  - 线程的基本概念
  - UNIX中进程的描述与控制
- 进程同步与通信
  - 进程同步
  - 经典进程的同步问题
  - 管程机制
  - 进程通信
  - UNIX中进程的同步与通信
- 处理机调度与死锁（第3章）

第3章　处理机调度与死锁

在多道程序环境下，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、中级调度等。而系统的运行性能在很大程度上取决于调度，因此调度便成为多道程序的关键。

在多道程序环境下，由于多个进程的并发执行，改善了系统资源的利用率并提高了系统的处理能力，然而，多个进程的并发执行也带来了新的问题----死锁。

第3章　处理机调度与死锁

- 处理机调度的层次
- 调度队列模型和调度准则
- 调度算法
- 实时调度
- UNIX系统中进程的调度
- 产生死锁的原因和必要条件
- 预防死锁的方法
- 死锁的避免
- 死锁的检测与解除
- 本章作业

3.1 处理机调度的层次

在多道程序环境下，一个作业从提交直到完成，往往要经历多级调度。

在不同操作系统中所采用的调度层次不完全相同。

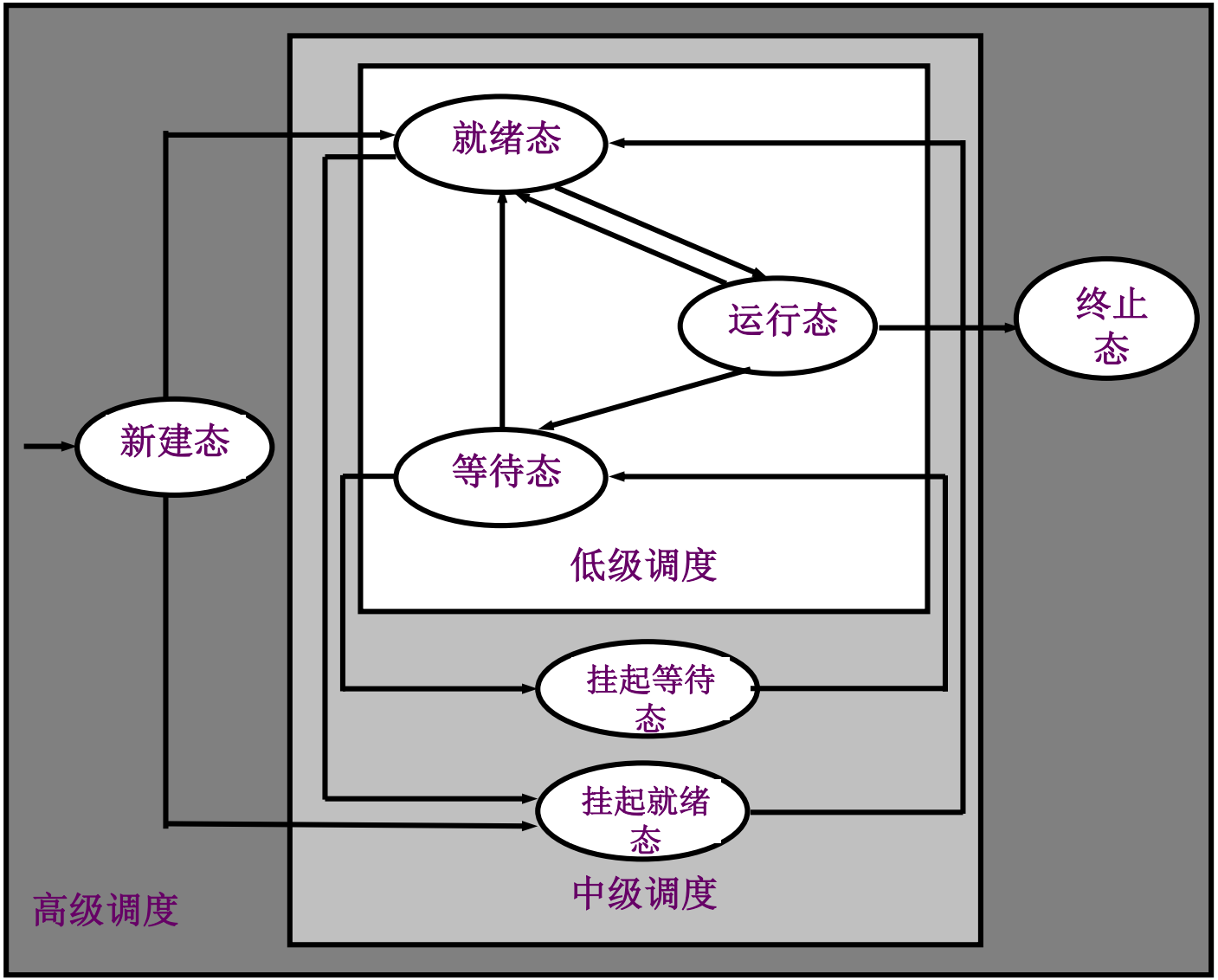
有些系统中：仅采用一级调度；

另一些系统：可能采用两级或三级调度。

在执行调度时所采用的调度算法也可能不同。

一、调度的层次

如图所示。



二、高级调度（1）

一个作业从提交开始，往往要经历三级调度：高级调度、低级调度、中级调度。

有关作业的几个基本概念：

（1）作业：

每个用户要求计算机完成一件事或任务称为一个作业。作业由程序、数据、作业说明书或作业控制块组成。

（2）作业步：

要求计算机系统做的一项相对独立又相互关联的顺序加工步骤叫做一个作业步。作业由若干个作业步组成。

二、高级调度（2）

（3）作业流：

一批作业在系统控制下，依次输入到外存中等待运行，就形成了一个作业流。

（4）作业控制块（JCB）：

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。

（5）作业的状态：

提交状态、后备状态、运行状态、完成状态

二、高级调度（3）

作业和程序：

两个概念既相互联系又有区别：如果一次业务处理可以由某一个程序完成，就是说这个业务处理只要提交这一个程序就够了，则这个程序就是一个作业；通常，完成一次业务需要由多个程序协同完成，则这多个程序、这些程序需要的数据以及必要的作业说明一起构成一个作业。系统通过作业说明书或JCB控制程序和相应的数据执行，完成整个业务处理。

二、高级调度（4）

高级调度（长程/作业/宏观调度）

（1）用于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。

（2）在批处理系统中，大多配有作业调度，但在分时系统及实时系统中，一般不配置.

（3）作业调度执行频率很低，通常为几分钟一次，甚至更久。

二、高级调度（5）

◆高级调度需解决的问题

（1）主要任务是从外存后备队列中选择多少作业进入就绪队列，即允许多少作业同时在内存中运行（多道程序的“道或度”）。若作业太多，则可能会影响系统的服务质量（如周转时间太长），若太少，又将导致系统资源利用率和吞吐量的下降。因此，应根据系统的规模和运行速度来确定，同时要求I/O型进程与CPU型进程中和调度。

（2）应将哪些作业从外存调入内存，将取决于调度算法（先来先服务、短作业优先等）。

三、低级调度(短程/CPU/进程/微观调度)

（1）主要任务是从就绪队列中选择一个进程来执行并分配处理机。

（2）是OS中最基本的调度。

（3）调度频率非常高，一般几十毫秒一次。

（4）常采用非抢占（非剥夺）方式和抢占（剥夺）方式两种。

三、低级调度(短程/CPU/进程/微观调度)

（5）引起进程调度的因素：

- 进程正常终止或异常终止
- 正在执行的进程因某种原因而阻塞
- 在引入时间片的系统中，时间片用完。
- 在抢占调度方式中，就绪队列中某进程的优先权变得比当前正执行的进程高。
- 在进程通信或同步过程中，执行了某种原语操作，如唤醒原语

调度方式--非抢占式进程调度、抢占式进程调度

- **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**，直到该进程完成或因某事件而被阻塞，才再把处理机分配给其它进程，决不允许某进程抢占已分配出去的处理机。

  实现简单，系统开销小，常用于批处理系统；但不利于处理紧急任务，故实时、分时系统不宜采用。

- **抢占方式**：允许调度程序根据某种原则（时间片、优先权、短进程优先），**停止正在执行**的进程，而将处理机重新分配给另一进程。

  有利于处理紧急任务，故实时与分时系统中常采用。

四、中级调度（中程/交换调度）

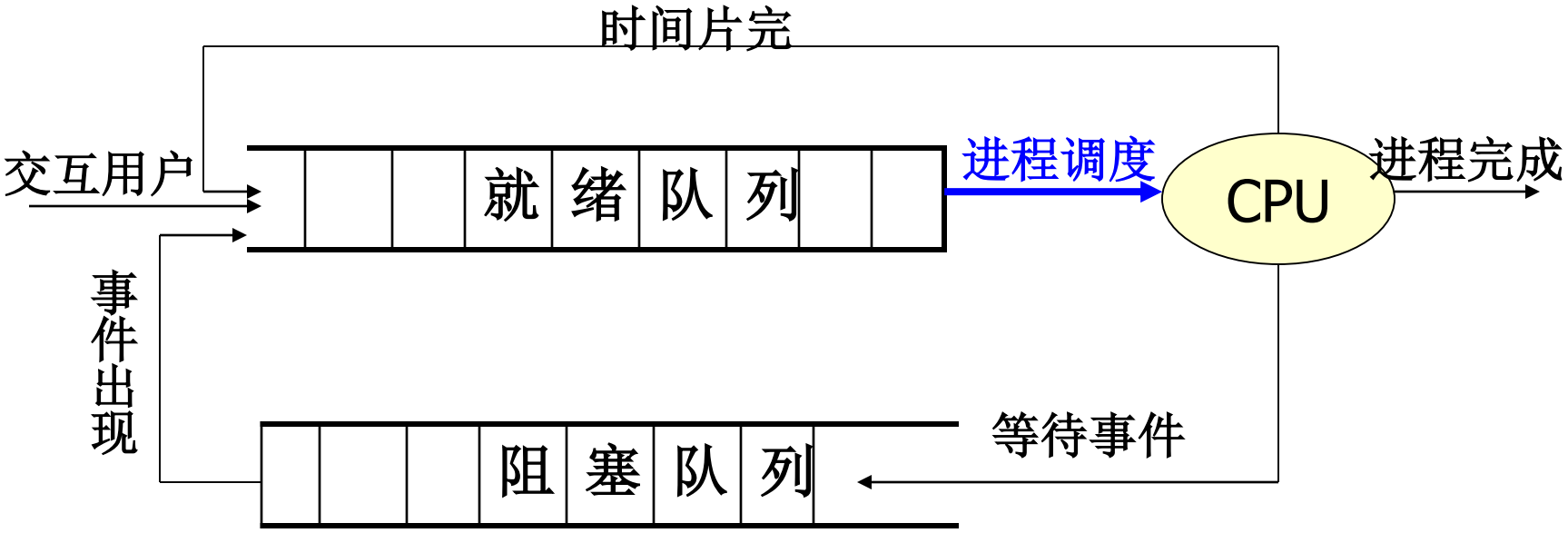
在内存和外存对换区之间按照给定的原则和策略选择进程对换，以解决内存紧张问题，从而提高内存的利用率和系统吞吐量，常用于分时系统或具有虚拟存储器的系统中。

3.2 调度队列模型和调度准则

在OS中的任何一种调度中，都将涉及到进程队列，由此形成了三种类型的调度队列模型。

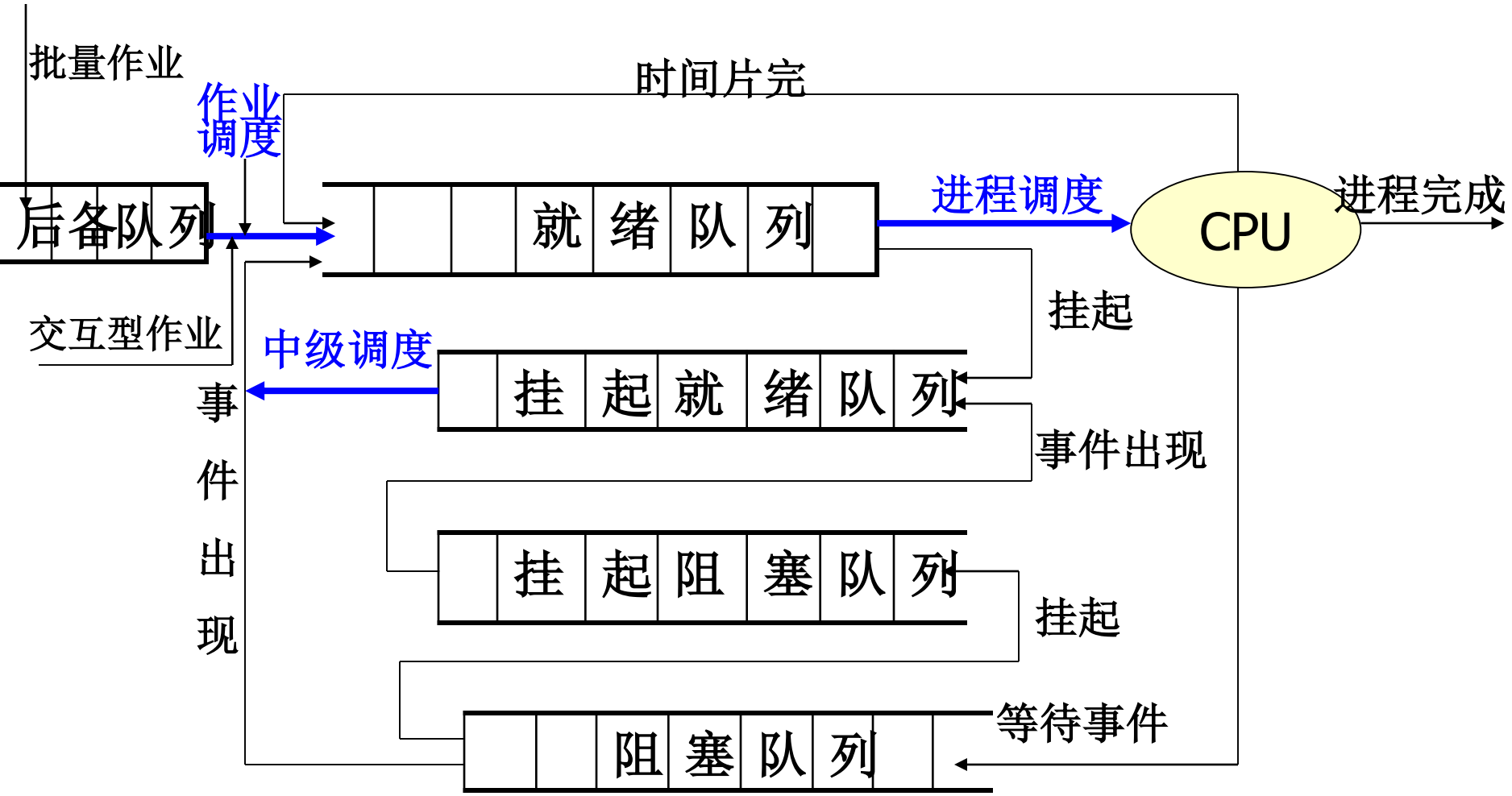
- 仅有进程调度的调度队列模型
- 具有高级和低级调度的调度队列模型
- 同时具有三级调度的调度队列模型

一、仅有进程调度的调度队列模型



二、具有高级和低级调度的调度队列模型

三、同时具有三级调度的调度队列模型



四、选择调度方式和算法的若干准则（1）

在一个操作系统的设计中，应如何选择调度方式和算法，在很大程度上取决于操作系统的类型及其目标，选择调度方式和算法的准则有：

- 面向用户的准则
  - 周转时间短
  - 响应时间快
  - 截止时间的保证
  - 优先权准则
- 面向系统的准则
  - 系统吞吐量
  - 处理机利用率好
  - 各类资源平衡利用
- 最优准则
  - 最大的CPU利用率
  - 最大的吞吐量
  - 最短的周转时间
  - 最短的等待时间
  - 最短的响应时间

四、选择调度方式和算法的若干准则（2）

CPU利用率=CPU有效工作时间/CPU总的运行时间，

CPU总的运行时间=CPU有效工作时间+CPU空闲等待时间。

响应时间--交互式进程从提交一个请求(命令)到接收到响应之间的时间间隔称响应时间。

周转时间--批处理用户从作业提交给系统开始，到作业完成为止的时间间隔称作业周转时间，实际上它是作业在系统里的等待时间与运行时间之和。应使作业周转时间或平均作业周转时间尽可能短。

四、选择调度方式和算法的若干准则（3）

带权周转时间—周转时间T/进程要求运行时间Ts

吞吐量--单位时间内处理的作业数。

公平性--确保每个用户每个进程获得合理的CPU份额或其他资源份额，不会出现饿死情况。

平均周转时间

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

平均带权周转时间

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

3.3 调度算法

进程调度的核心问题就是采用什么样的算法将处理机分配给进程，常用的进程调度算法有：

- 先来先服务调度算法
- 短作业/进程优先调度算法
- 时间片轮转调度算法
- 优先权调度算法
- 高响应比优先调度算法
- 多级反馈队列调度算法

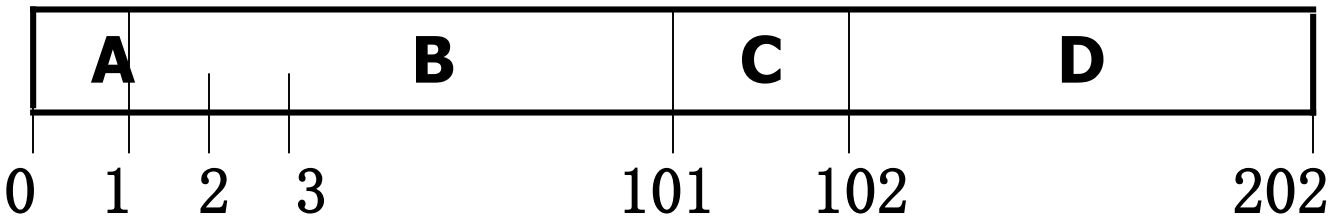
一、先来先服务调度算法FCFS

- **基本思想：** 按照进程进入就绪队列的先后次序来分配处理机。一般采用非剥夺的调度方式。
- Example:

进程名	到达时间	服务时间
A	0	1
B	1	100
C	2	1
D	3	100

甘特图(Gantt)是作业排序中最常用的一种工具，最早由Henry L. Gantt于1917年提出。这种方法基于作业排序的目的，是将任务与时间联系起来的表现形式之一。

- 该调度的Gantt（甘特）图为：



- 平均周转时间=((1-0)+(101-1)+(102-2)+(202-3)) /4=100
- 平均等待时间=((0-0)+(1-1)+(101-2)+(102-3))/4 = 49.5
- 平均带权周转时间=(1/1+100/100+100/1+199/100)/4≈26

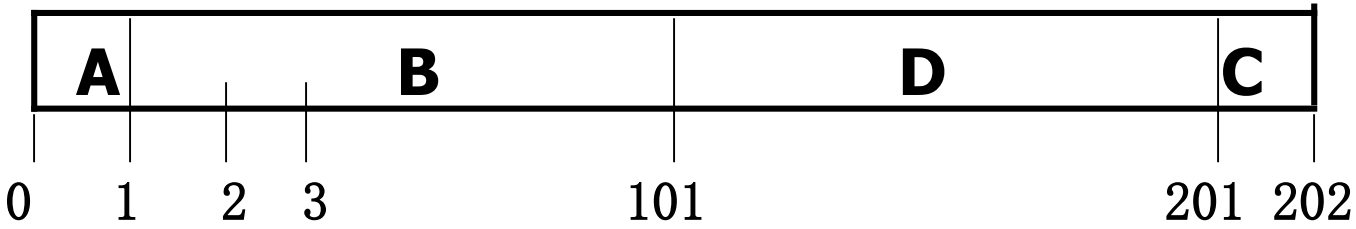
一、FCFS调度算法（续）

周转T	100	124.25
等待T	49.5	74.25

- 改变到达顺序：

进程名	到达时间	服务时间
A	0	1
B	1	100
D	2	100
C	3	1

- 该调度的Gantt图为：



- 平均周转时间=((1-0)+(101-1)+(202-3)+(201-2))/4=124.25
- 平均等待时间=((0-0)+(1-1)+(201-3)+(101-2))/4 = 74.25
- 平均带权周转时间=(1/1+100/100+199/1+199/100)/4≈50.75

FCFS调度算法存在的问题

从表面上，先来先服务对于所有作业是公平的，即按照它们到来的先后次序进行服务。但如果一个长作业先到达系统，就会使许多短作业等待很长的时间，从而引起许多短作业用户的不满。

所以，现在操作系统中，已很少用该算法作为主要调度策略，尤其是在分时系统和实时系统中。但它常被结合在其它调度策略中使用。

二、短作业/进程优先调度算法SJF/SPF

- 短作业优先调度算法（SJF）
  - 用于作业调度
  - 主要任务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。
- 短进程优先调度算法（SPF）
  - 用于进程调度
  - 主要任务是从就绪队列中选出一个估计运行时间最短的进程，将处理机分配给它。
  - 可采用抢占（剥夺）（有时称为最短剩余时间优先（shortest-remaining-time-first）调度）或者非抢占（非剥夺）调度方式。

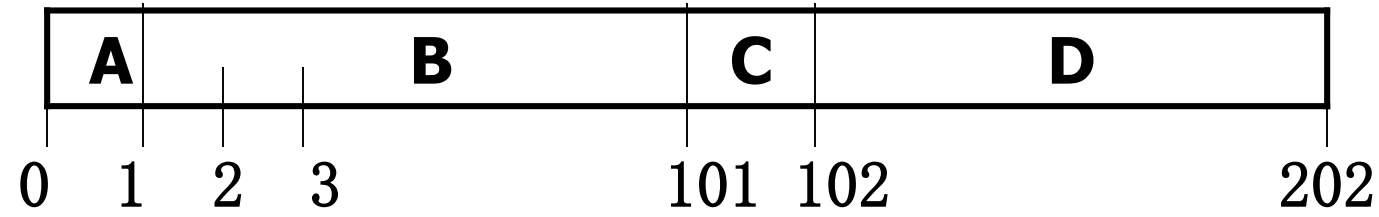
二、SJ(P)F –非抢占式

	FCFS	SPF
周转T	124.25	100
等待T	74.25	49.5

- 到达顺序：

进程名	到达时间	服务时间
A	0	1
B	1	100
D	2	100
C	3	1

- 该调度的Gantt图为：



- 平均周转时间=((1-0)+(101-1)+(102-3)+(202-2))/4=100
- 平均等待时间=((0-0)+(1-1)+(101-3)+(102-2))/4 = 49.5
- 平均带权周转时间= (1/1+100/100+99/1+200/100)/4≈26

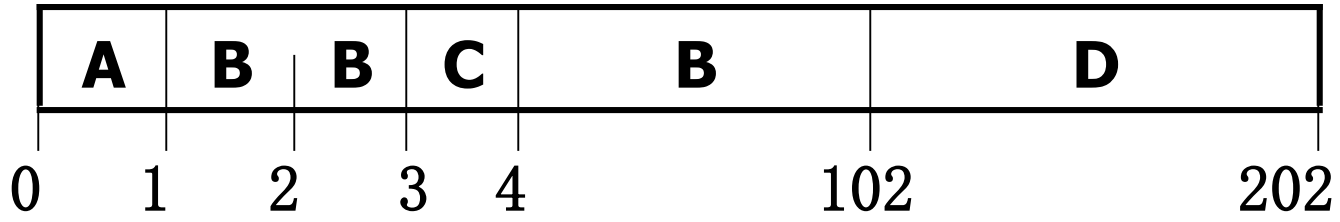
二、短作业/进程优先调度算法-抢占式

- 到达顺序：进程名 到达时间 服务时间

A	0	1
B	1	100
D	2	100
C	3	1

	FCFS	SPF-非	SPF-抢
周转T	124.25	100	75.75
等待T	74.25	49.5	25.25

- 该调度的Gantt图为：



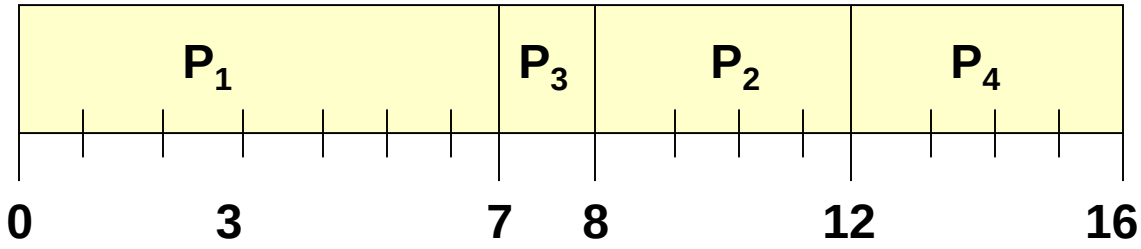
- 平均周转时间=((1-0)+(102-1)+(4-3)+(202-2)) /4=75.75
- 平均等待时间=((0-0)+(4-3)+(3-3)+(102-2))/4 = 25.25
- 平均带权周转时间=(1/1+101/100+1/1+200/100)/4=1.2525

SPF（非抢占式）调度

◆Eg：

进程	到达时间	服务时间
P_1	0.0	7
P_2	2.0	4
P_3	4.0	1
P_4	5.0	4

◆SPF （非抢占式）



◆平均周转时间=((7-0)+(12-2)+(8-4)+(16-5))/4=8

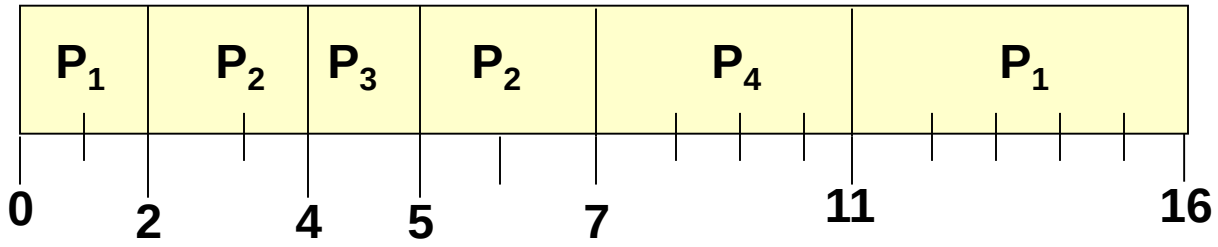
◆平均等待时间 =((0-0)+(8-2)+(7-4)+(12-5))/4 = 4

◆平均带权周转时间=(7/7+10/4+4/1+11/4)/4=2.5625

SPF抢占式调度

进程	到达时间	服务时间
P_1	0.0	7
P_2	2.0	4
P_3	4.0	1
P_4	5.0	4

- SPF （抢占式）

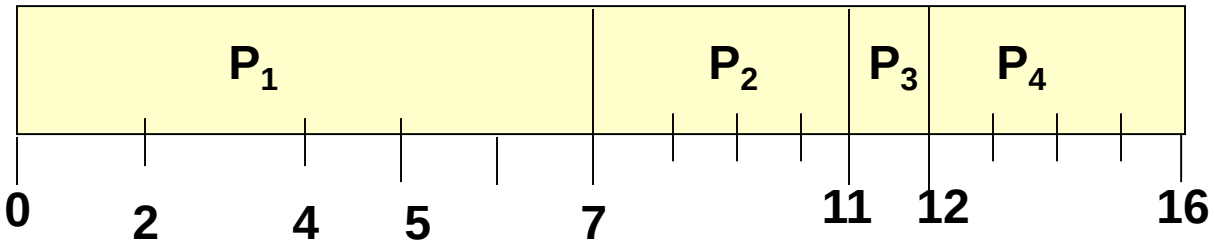


- 平均周转时间=((16−0)+(7−2)+(5−4)+(11−5))/4=7
- 平均等待时间=((11−2)+(5−4)+(4−4)+(7−5))/4=3
- 平均带权周转时间=(16/7+5/4+1/1+6/4)/4≈1.51

FCFS先来先服务调度

进程	到达时间	服务时间
P_1	0.0	7
P_2	2.0	4
P_3	4.0	1
P_4	5.0	4

- FCFS



- 平均周转时间=((7−0)+(11−2)+(12−4)+(16−5))/4=8.75
- 平均等待时间 =(0+(7−2)+(11−4)+(12−5))/4 =4.75
- 平均带权周转时间=(7/7+9/4+8/1+11/4)/4=3.5

SPF与FCFS的比较

	FCFS	非抢占SPF	抢占SPF
吞吐量0-7ms	1	1	2
平均周转时间	8.75	8	7
平均等待时间	4.75	4	3

SJ(P)F短作业/进程优先调度的优缺点

- 优点：
  1）能有效降低作业的平均等待时间；
  2）提高吞吐量；
  3）能有效缩短进程的周转时间；
- 缺点：
  1）对长作业不利；
  2）不考虑作业的紧迫程度；
  3）作业执行时间、剩余时间仅为估计时间；

故SJ(P)F算法虽然是优化的，但在CPU调度中很难实现。

三、时间片轮转调度算法RR（1）

应用于分时OS中，能保证及时响应用户的请求，是早期采用的一种调度算法；进入90年代后，广泛采用多级反馈队列调度算法。

- **时间片轮转法：**系统将所有原就绪进程按FCFS的原则，排成一个队列，依次调度，把CPU分配给队首进程，并令其执行一个时间片/CPU时间，通常为10-100ms。时间片用完后，该进程将被抢占并插入就绪队列末尾。

三、时间片轮转调度算法RR（2）

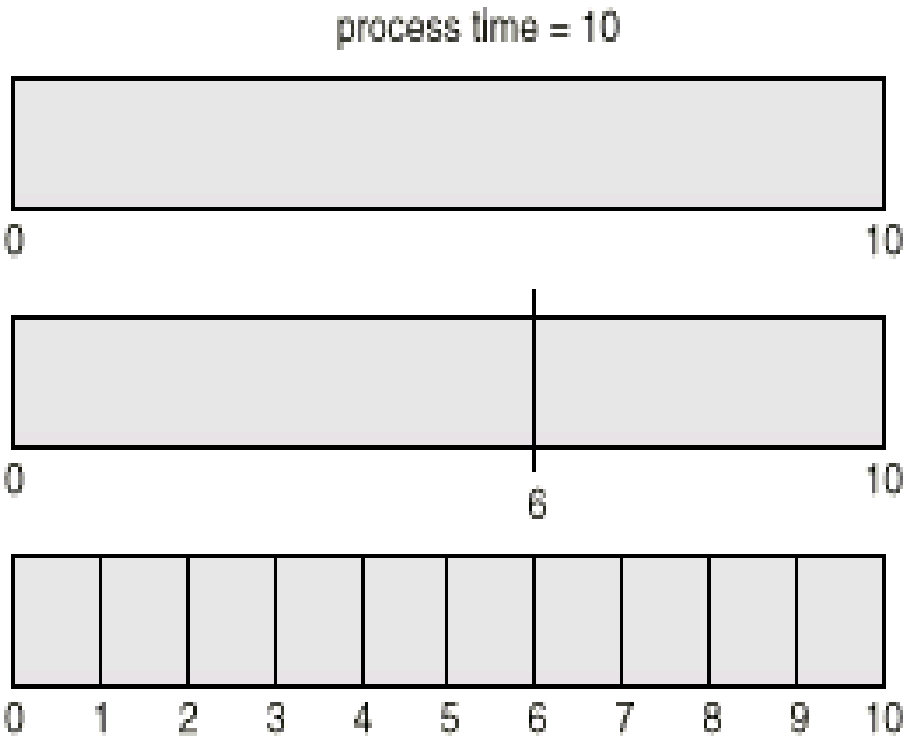
（1）保证了就绪队列中的所有进程在给定的时间内，均能获得一时间片来执行，即系统在给定的时间内，响应所有用户的请求。

（2）若进程的执行时间少于时间片，则自愿释放CPU。

（3）时间片将影响：

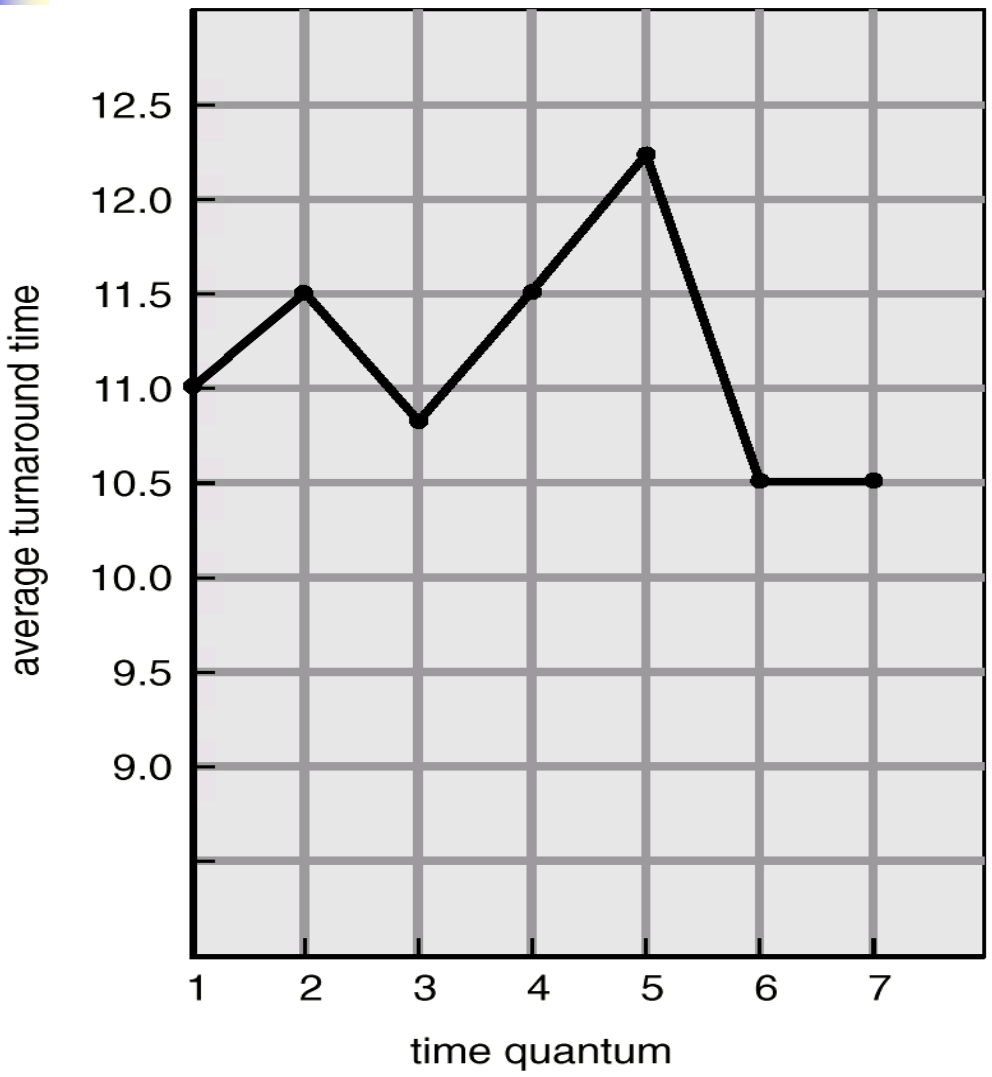
- 调度算法（太长--FCFS）；
- 上下文切换（太短--上下文切换频繁，如下页）；
- 平均周转时间。

短时间片增加上下文切换频率



quantum	context switches
12	0
6	1
1	9

周转时间随时间片变化



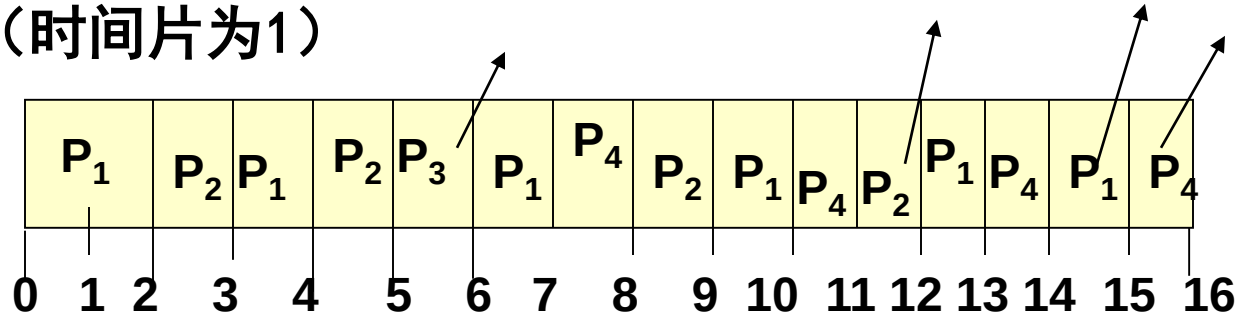
process	time
P_1	6
P_2	3
P_3	1
P_4	7

三、时间片轮转调度算法—例(1)

Eg：

进程	到达时间	服务时间
P_1	0.0	7
P_2	2.0	4
P_3	4.0	1
P_4	5.0	4

◆RR（时间片为1）



◆平均周转时间=((15−0)+(12−2)+(6−4)+(16−5))/4=9.5

◆平均等待时间=(8+6+1+7)/4 = 5.5

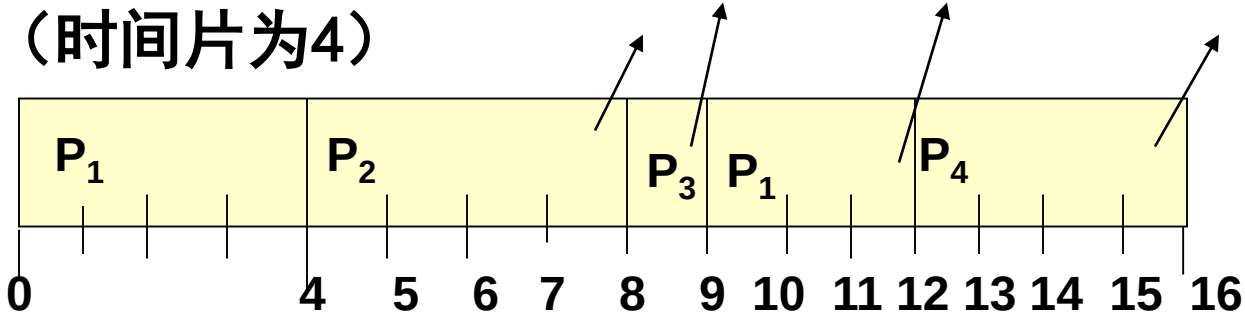
◆平均带权周转时间=(15/7+10/4+2/1+11/4)/4≈2.35

三、时间片轮转调度算法—例(2)

Eg：

进程	到达时间	服务时间
P_1	0.0	7
P_2	2.0	4
P_3	4.0	1
P_4	5.0	4

◆RR（时间片为4）



◆平均周转时间=((12−0)+(8−2)+(9−4)+(16−5))/4=8.5

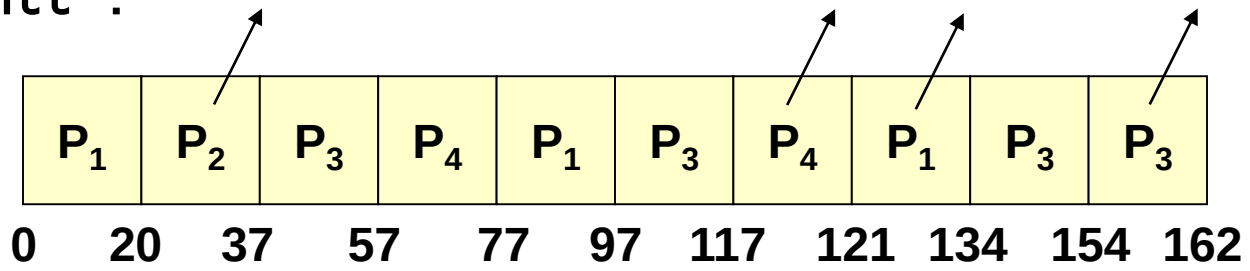
◆平均等待时间=(5+2+4+7)/4 = 4.5

	FCFS	非抢占SPF	抢占SPF	RR-1	RR-4
平均周转时间	8.75	8	7	9.5	8.5
平均等待时间	4.75	4	3	5.5	4.5

Eg: RR 时间片=20

Process	Burst Time
P_1	53
P_2	17
P_3	68
P_4	24

◆Gantt ：



◆RR的平均周转时间比SPF长，但响应时间要短一些.

四、优先权调度算法

- **非抢占式优先权算法**：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，直到完成/因发生某事件而放弃处理机时，系统才重新分配处理机。
- **抢占式优先权算法**：系统把处理机分配给就绪队列中优先权最高的进程，使之执行。但在其执行期间，只要出现了另一个优先权更高的进程进入就绪队列，进程调度程序就立即停止当前进程的执行，将其放入就绪队列，重新将处理机分配给新到的优先权最高的进程。

优先权的类型（1）

静态优先权

优先权在创建进程时确定，且在进程的整个运行期间保持不变。

一般用一个整数表示，越小表示优先级越高。

优先权的类型（2）

动态优先权

优先权在创建进程时确定，但在进程的运行期间会发生变化。

①根据进程占有CPU时间多少来决定，当进程占有CPU时间愈长，那么，在它被阻塞之后再次获得调度的优先级就越低，反之，进程获得调度的可能性越大

②根据进程等待CPU时间多少来决定，当进程在就绪队列中等待时间愈长，那么，在它被阻塞之后再次获得调度的优先级就越高，反之，进程获得调度的可能性越小

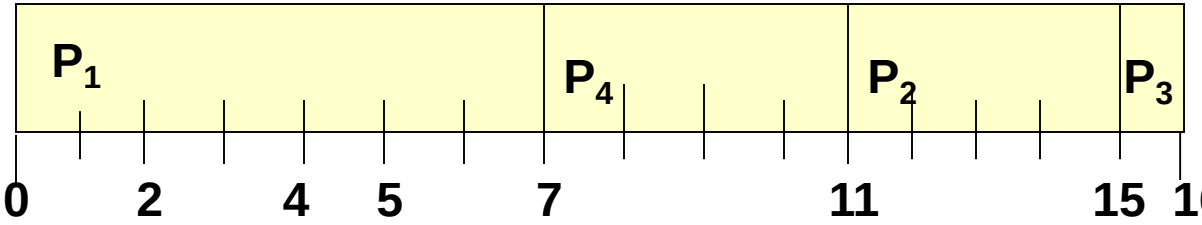
	FCFS	SPF-非	SPF-抢	RR-4	优-非
平均周转时间	8.75	8	4	8.5	9.5
平均等待时间	4.75	4	3	4.5	5.5

非抢占式优先权算法—例

EG：

进程	到达时间	服务时间	优先权
P_1	0.0	7	3
P_2	2.0	4	2
P_3	4.0	1	4
P_4	5.0	4	1

◆Gantt图



◆平均周转时间=((7-0)+(15-2)+(16-4)+(11-5))/4=9.5

◆平均等待时间=(0+9+11+2)/4 = 5.5

◆平均带权周转时间=(7/7+13/4+12/1+6/4)/4=4.4375

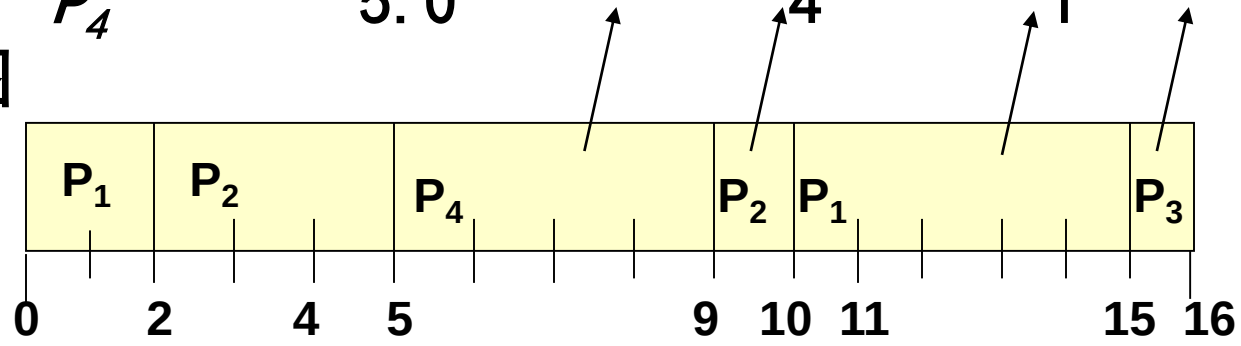
	FCFS	SPF-非	SPF-抢	RR-4	优-非	优-抢
平均周转时间	8.75	8	4	8.5	9.5	9.75
平均等待时间	4.75	4	3	4.5	5.5	5.75

抢占式优先权算法—例

Eg:

进程	到达时间	服务时间	优先权
P_1	0.0	7	3
P_2	2.0	4	2
P_3	4.0	1	4
P_4	5.0	4	1

◆Gantt图



◆平均周转时间=((15-0)+(10-2)+(16-4)+(9-5))/4=9.75

◆平均等待时间=(8+4+11+0)/4 = 5.75

◆平均带权周转时间=(15/7+8/4+12/1+4/4)/4≈4.29

五、高响应比优先调度算法(HRRN)(1)

FCFS与SJF是片面的调度算法。

- FCFS只考虑作业等候时间而忽视了作业的计算时间；
- SJF只考虑用户估计的作业计算时间而忽视了作业等待时间。

HRRN是介于这两者之间的折衷算法，既考虑作业等待时间，又考虑作业的运行时间，既照顾短作业又不使长作业的等待时间过长，改进了调度性能。

五、高响应比优先调度算法（2）

- 优先权的变化为

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

R_P 为响应比。

注：

1）如等待时间相同，则要求服务时间越短，其优先权越高--SPF.

2）如要求服务时间相同，优先权决定于等待时间----FCFS。

3）对长作业，若等待时间足够长，优先权也高，也能获得CPU。

五、高响应比优先调度算法（3）

高响应比优先算法举例：

例如，以下四个作业先后到达系统进入调度：

作业名	到达时间	所需CPU时间（服务时间）
P1	0	20
P2	5	15
P3	10	5
P4	15	10

五、高响应比优先调度算法（4）

- 高响应比优先调度算法
  - 开始只有P1，被选中，执行时间20；P1的周转时间为20；
  - P1执行完毕，P2、P3、P4的响应比依次为1+15/15、1+10/5、1+5/10，P3被选中，执行时间5；P3的周转时间为25−10=15；
  - P3执行完毕，P2、P4的响应比依次为1+20/15、1+10/10，P2被选中，执行时间15；P2的周转时间为40−5=35
  - P2执行完毕，P4被选中，执行时间10；P4的周转时间为50−15=35

平均周转时间 = (20+15+35+35)/4 = 26.25

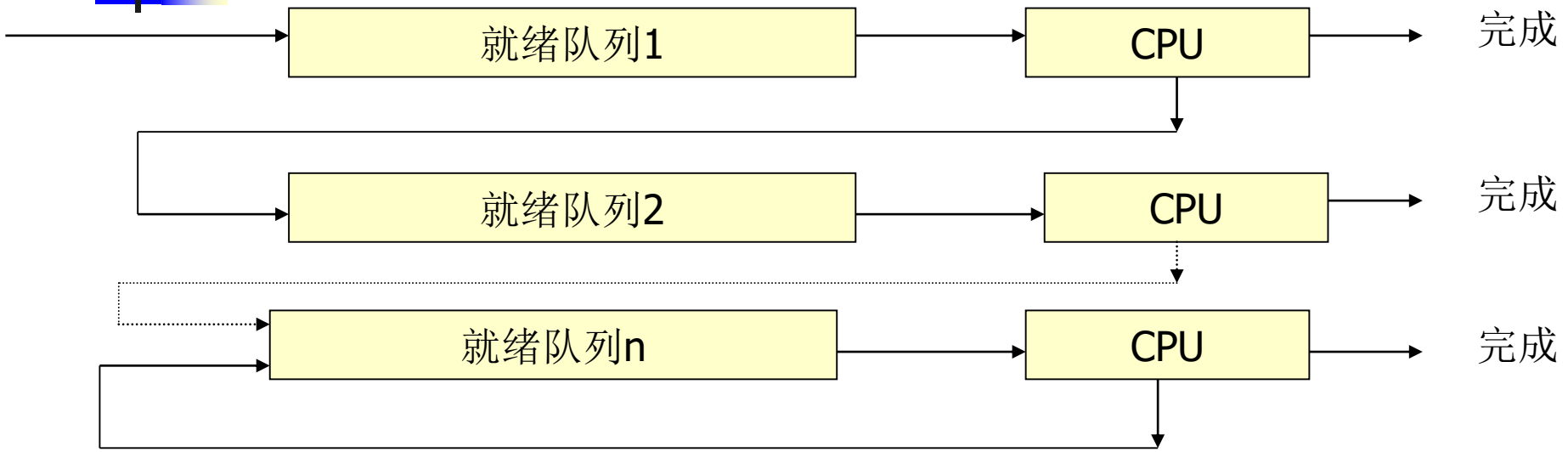
平均带权周转时间 = (20/20+15/5+35/15+35/10)/4 = 2.42

六、多级反馈队列调度算法（1）

- **基本思想**：

多级反馈队列调度算法是时间片轮转算法和优先级调度算法的综合和发展，通过动态调整进程优先级和时间片大小，不必事先估计进程的执行时间，多级反馈队列可兼顾多方面的系统目标，是目前公认的一种较好的进程调度算法。

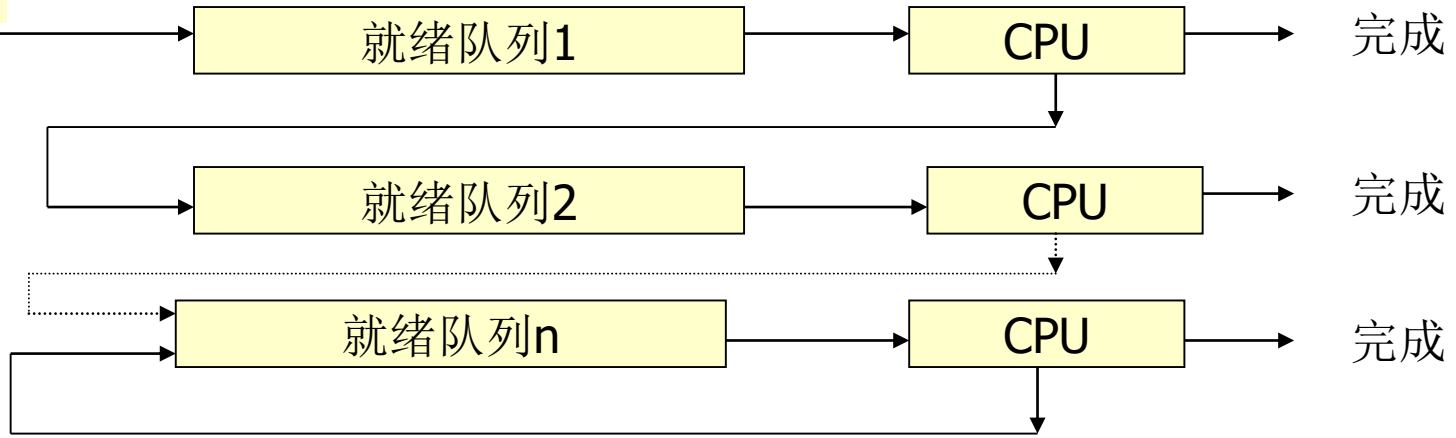
六、多级反馈队列调度算法（2）



（1）设置多个就绪队列，并为每个队列赋予不同的优先级。队列1的优先级最高，其余队列逐个降低。

（2）每个队列中进程执行时间片的大小也各不相同，进程所在队列的优先级越高，其相应的时间片就越短。

六、多级反馈队列调度算法（3）



（3）当一个新进程进入系统时，首先将它放入队列1的末尾，按FCFS等待调度。如能完成，便可准备撤离系统，反之由调度程序将其转入队列2的末尾，按FCFS再次等待调度，如此下去，进入队列n按RR算法调度执行。

（4）仅当队列1为空时，才调度队列2中的进程运行。若队列i中的进程正在执行，此时有新进程进入优先级较高的队列（第1~(i-1)队列）中，则新进程将抢占运行，原进程转移至本队列（第i队列）的末尾。

3.5 死锁的基本概念（1）

在多道程序系统中，由于多个进程的并发执行，改善了系统资源的利用率并提高了系统的处理能力，然而，多个进程的并发执行也带来了新的问题------死锁。

- 例：设有一台输入机和一台输出机，进程P1和P2需要使用这两个资源。设两个进程的活动分别为：

P1的活动：
……
申请输入机
……
申请输出机
……
释放输出机
……
释放输入机

P2的活动：
……
申请输出机
……
申请输入机
……
释放输入机
……
释放输出机

P1与P2均因得不到资源而无法继续向前推进，这种由于进程竞争资源而引起的僵持称为死锁。

3.5 死锁的基本概念（2）

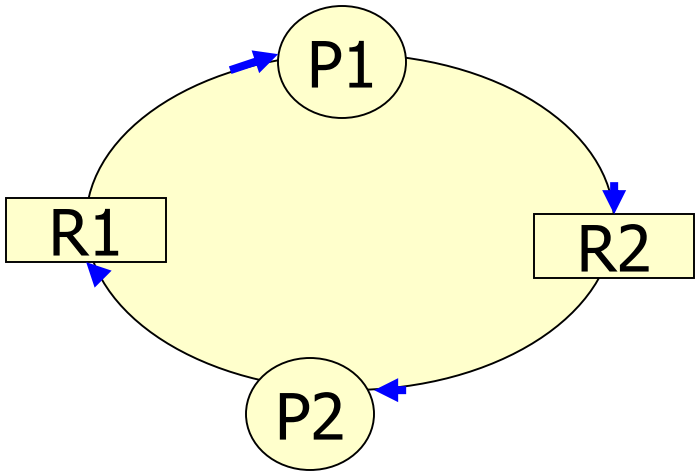
- 死锁
- 产生死锁的原因
- 产生死锁的必要条件
- 处理死锁的基本方法

3.5 死锁的基本概念（3）

一、死锁

指多个进程在运行过程中因争夺资源而造成的一种僵局（deadly-Embrace），若无外力作用，这些进程都将无法向前推进。

◆ 如图



注意：

（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。

二、产生死锁的原因（1）

◆资源分类

操作系统管理着系统内所有资源，它负责分配不同类型的资源给进程使用，系统中的资源从不同角度可分：

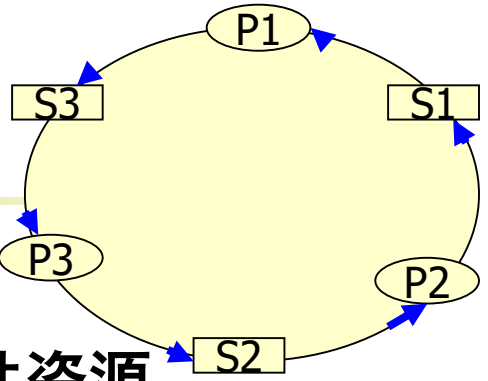
根据资源本身的性质

- ❖可剥夺资源：如主存、CPU
- ❖不可剥夺资源：如驱动器、打印机等

根据资源使用期限

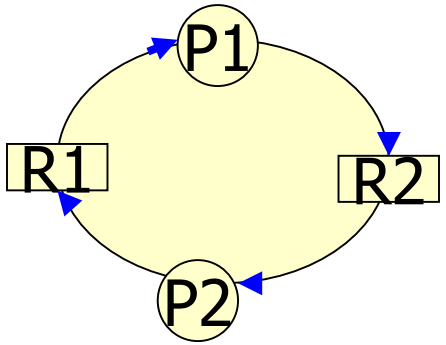
- ❖永久性资源：可再次使用，如所有硬件。
- ❖临时性资源：消耗性的资源，如消息、信号和数据

二、产生死锁的原因（2）



◆竞争资源

▸竞争非剥夺性资源



竞争非剥夺性资源

R1代表系统中仅有的一台打印机

R2代表系统中仅有的一台磁带机

P1、 P2代表可共享资源的进程

▸竞争临时性资源

竞争临时性资源

若按下列顺序进行无死锁产生：

P1：...Release(S1);Request(S3);...

P2：...Release(S2);Request(S1);...

P3：...Release(S3);Request(S2);...

但若按下列顺序进行可能产生死锁：

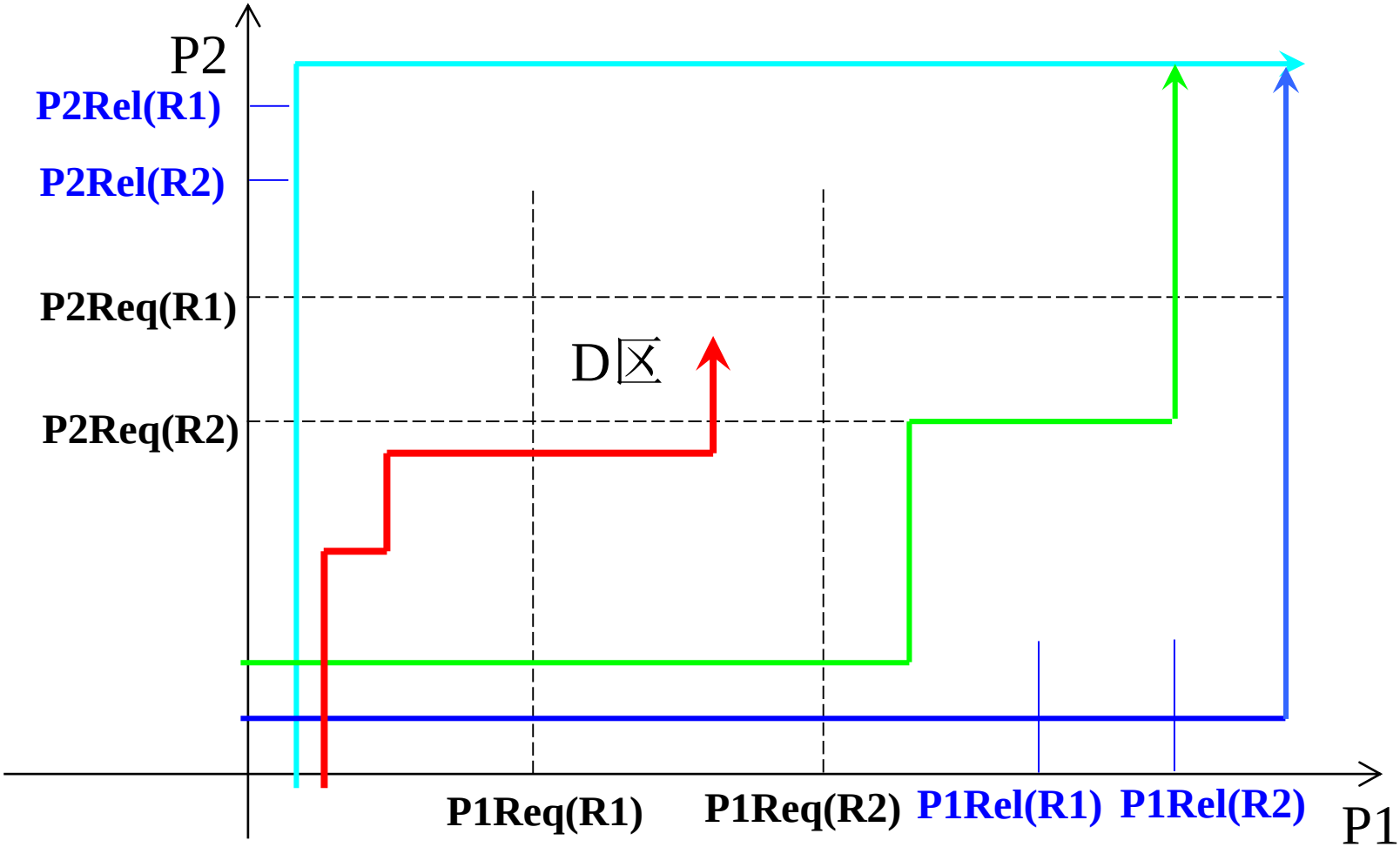
P1：...Request(S3); Release(S1); ...

P2：...Request(S1); Release(S2);...

P3：...Request(S2); Release(S3);...

二、产生死锁的原因（3）

- 进程推进顺序不当引起死锁
  - 进程推进顺序合法
  - 进程推进顺序非法（红色折线）



三、产生死锁的必要条件

产生死锁必须具备以下四个必要条件，这四个条件是Coffman首先提出的，所以称为Coffman 条件：

- 互斥条件（资源独占条件）
- 请求和保持条件（部分分配条件）
- 不剥夺条件
- 循环等待条件（环路条件）

四、处理死锁的基本方法

目前处理死锁的基本方法有四种：

- **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 死锁的预防和避免方法（1）

一、死锁的预防

—— 破坏死锁的四个必要条件（常针对条件2, 3, 4）

（1）破坏互斥条件（不可行）

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，所以不可能用破坏互斥条件来预防死锁。

3.6 死锁的预防和避免方法（2）

（2）破坏请求和保持条件

第一种协议：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。

此方法虽简单安全，但降低了资源利用率，同时必须预知进程所需要的全部资源；使进程会发生“饥饿”现象。

第二种协议：第一种协议的改进。允许一个进程只获得运行初期所需的资源后，就开始运行，运行过程中逐步释放已分配且用完的资源，再请求新的所需要的资源，举例。

3.6 死锁的预防和避免方法（3）

（3）破坏不可剥夺条件

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。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“不剥夺”条件。

这种方法实现较复杂，会增加系统开销，降低系统吞吐量。

3.6 死锁的预防和避免方法（4）

（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后面的资源，这样不会出现几个进程请求资源而形成环路。

该方法虽提高了资源的利用率，但编号难，加重进程负担及因使用资源顺序与申请顺序不同而造成资源浪费。

3.6 死锁的预防和避免方法（5）

◆死锁的避免

在死锁预防的几种方法中，都施加了较强的限制条件，严重降低了系统性能。

在死锁避免的方法中，所施加的限制条件较弱，对于进程发出的每一个资源申请命令实施动态检查，并根据检查结果决定是否实施资源分配。

在该方法中把系统的状态分为安全状态和不安全状态，只要能使系统始终处于安全状态，便可以避免死锁的发生。

二、系统的安全状态（1）

指在某一时刻，系统能按某种进程顺序(P1, P2, …， Pn)来为每个进程Pi分配其资源，直到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，称此时的系统状态为安全状态，称序列(P1, P2, …， Pn)为安全序列。

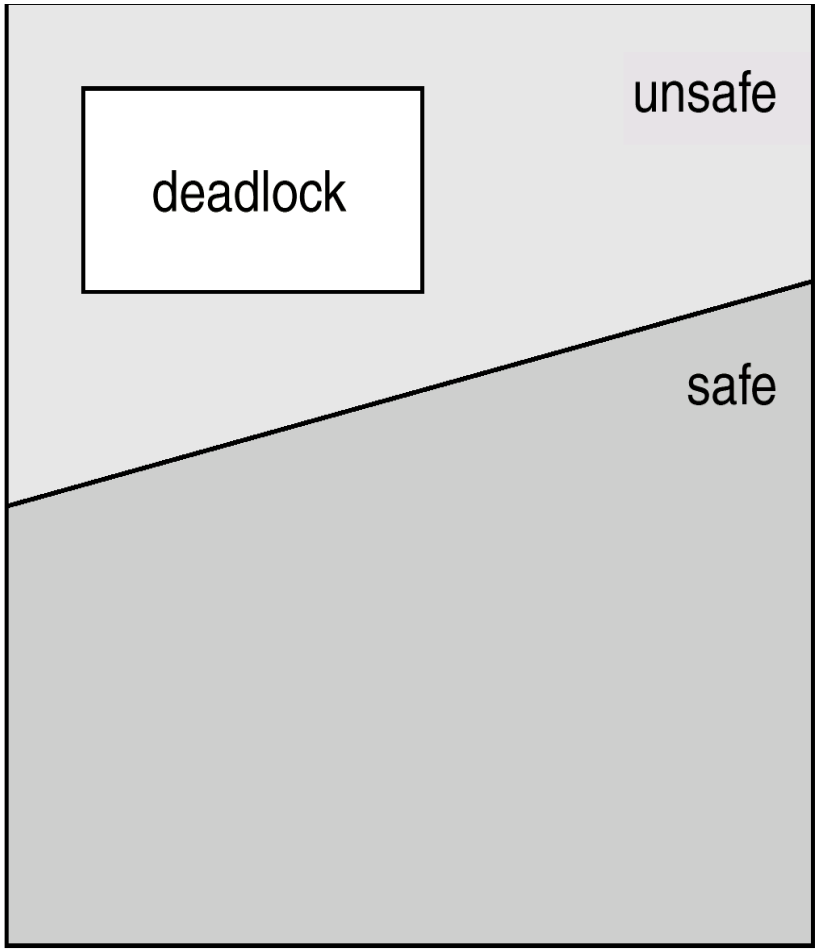
若某一时刻系统中不存在这样一个安全序列，则称此时的系统状态为不安全状态。

二、系统的安全状态（2）

注：

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。

安全、不安全、死锁状态空间



- 基本事实
  - 如果一个系统在安全状态，就没有死锁
  - 如果一个系统处于不安全状态，就有可能死锁
  - 避免死锁的实质：确保系统不进入不安全状态

安全状态实例

- 假定系统中有三个进程P1，P2和P3，共有12台磁带机，T0时刻，三个进程对磁带机的需求和占有情况如下表所示：

进 程	最大需求	已分配	可用
P1	10	5	3
P2	4	2	
P3	9	2	

T0时刻，存在一个安全序列（P2，P1，P3），所以系统是安全的。

在T0时刻后，P3又请求一台磁带机，若满足P3的请求，系统会怎样？

三、避免死锁的算法-银行家算法

具有代表性的避免死锁算法，是Dijkstra给出的银行家算法，为实现银行家算法，系统中必须设置若干数据结构。假定系统中有n个进程（P1，P2，...，Pn），m类资源（R1，R2，...，Rm），银行家算法中使用的数据结构如下：

◆可利用资源向量：available[j]=k，资源Rj类有k个可用

◆最大需求矩阵：Max[i,j]=k,进程Pi最大请求k个Rj类资源

◆分配矩阵：Allocation[i,j]=k,进程Pi分配到k个Rj类资源

◆需求矩阵：Need[i,j]=k,进程Pi还需要k个Rj类资源

三个矩阵的关系：

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

银行家算法描述—资源分配算法（1）

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

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

银行家算法描述—资源分配算法（2）

(3) 系统试分配资源给进程Pi，并作如下修改：

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，则正式进行分配，否则恢复原状态，让进程Pi等待。

银行家算法描述—安全性检查算法

为了进行安全性检查，需要定义如下数据结构：

- int Work[m]　工作变量，记录可用资源。开始时， Work:= Available
- boolean Finish[n]　工作变量，记录进程是否执行完。开始时， Finish[i]=false;当有足够资源分配给进程Pi时，令Finish[i]=true。

银行家算法描述—安全性检查算法

安全性检查算法：

(1) Work := Available

Finish[i] := false

(2)寻找满足如下条件的进程Pi

Finish [i] = false 并且 Need[i,j] ≤ Work

如果找到，转(3)，否则转(4)

(3)当进程Pi获得资源后，可顺利执行完，并释放分配给它的资源，故执行：

Work := Work + Allocation

Finish[i] := true

转 (2).

(4)若所有进程的Finish [i] = true，则表示系统处于安全状态，否则处于不安全状态。

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 A B C	Need A B C	Allocation A B C	Work+ Allocation 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( )所示）

Available、Need、Allocation

表3　P1请求资源时的资源分配表

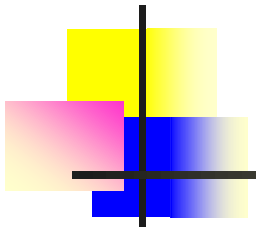
	Max A B C	Allocation A B C	Need A B C	Available A B C
P0	7 5 3	0 1 0	7 4 3	3 3 2 (2 3 0)
P1	3 2 2	2 0 0 (3 0 2)	1 2 2 (0 2 0)	
P2	9 0 2	3 0 2	6 0 0	
P3	2 2 2	2 1 1	0 1 1	
P4	4 3 3	0 0 2	4 3 1	

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 A B C	Need A B C	Allocation A B C	Work+ Allocation 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）P4请求资源 $Request_4(3,3,0)$

P4发出请求向量$Request_4(3,3,0)$，系统按银行家算法进行检查：

1）$Request_4(3,3,0) \leqslant Need_4\ (4,3,1)$

2）$Request_4\ (3,3,0) > Available\ (2,3,0)$，表示资源不够，则让P4等待

（4）P0请求资源 $Request_0(0,2,0)$

P0发出请求向量$Request_0(0,2,0)$，系统按银行家算法进行检查：

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

2） $Request_0(0,2,0) \leqslant Available(2,3,0)$

3） 系统试为P0分配资源，并修改相应的向量（见下表[ ]所示）

表5　P0请求资源时的资源分配表

	Max A B C	Allocation A B C	Need A B C	Available A B C
P0	7 5 3	0 1 0 [0 3 0]	7 4 3 [7 2 3]	3 3 2 (2 3 0) [2 1 0]
P1	3 2 2	2 0 0 (3 0 2)	1 2 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	

4)进行安全性检查：资源分配后此时系统是否安全，因Avaliable(2,1,0)已不能满足任何进程需要，故系统进入不安全状态，此时系统不分配资源。

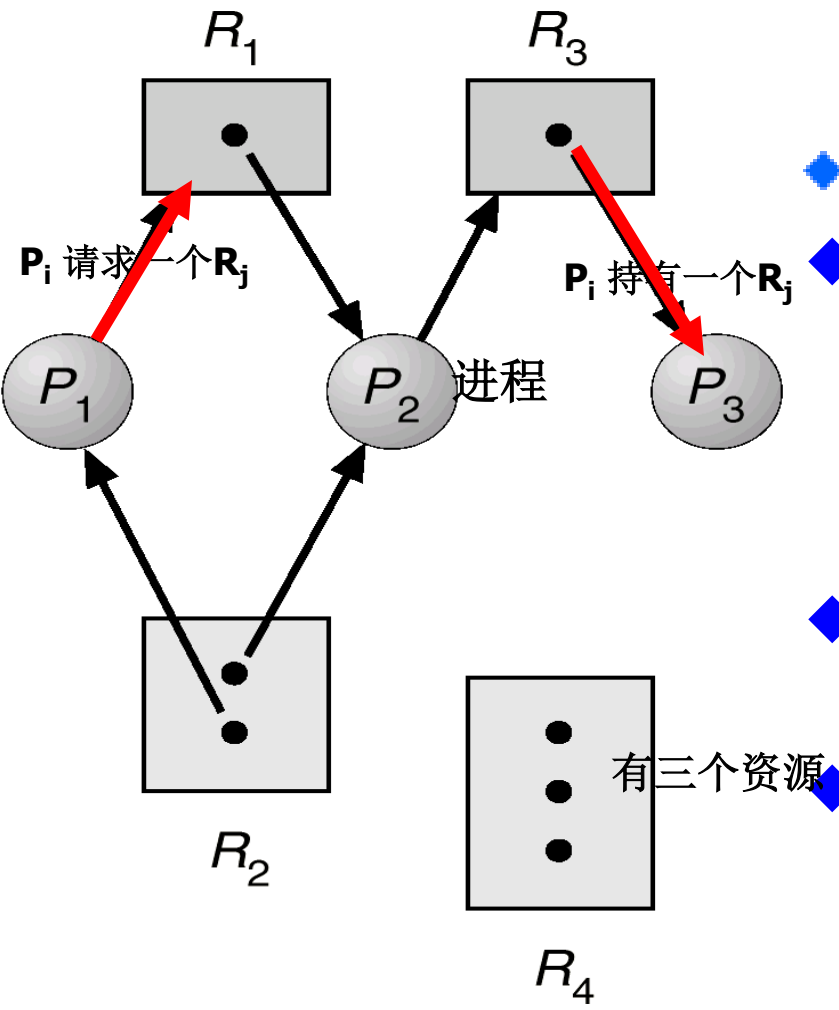
3.7 死锁的检测和解除

如果在一个系统中，既未采用死锁预防方法，也未采用死锁避免方法，而是直接为进程分配资源，则系统中便有可能发生死锁。一旦死锁发生，系统应能将其找到并加以消除，为此需提供死锁检测和解除的手段。

一、资源分配图

检测死锁的基本思想：在操作系统中保存资源的请求和分配信息，利用某种算法对这些信息加以检查，以判断是否存在死锁。为此，将进程和资源间的申请和分配关系描述成一个有向图---资源分配图。

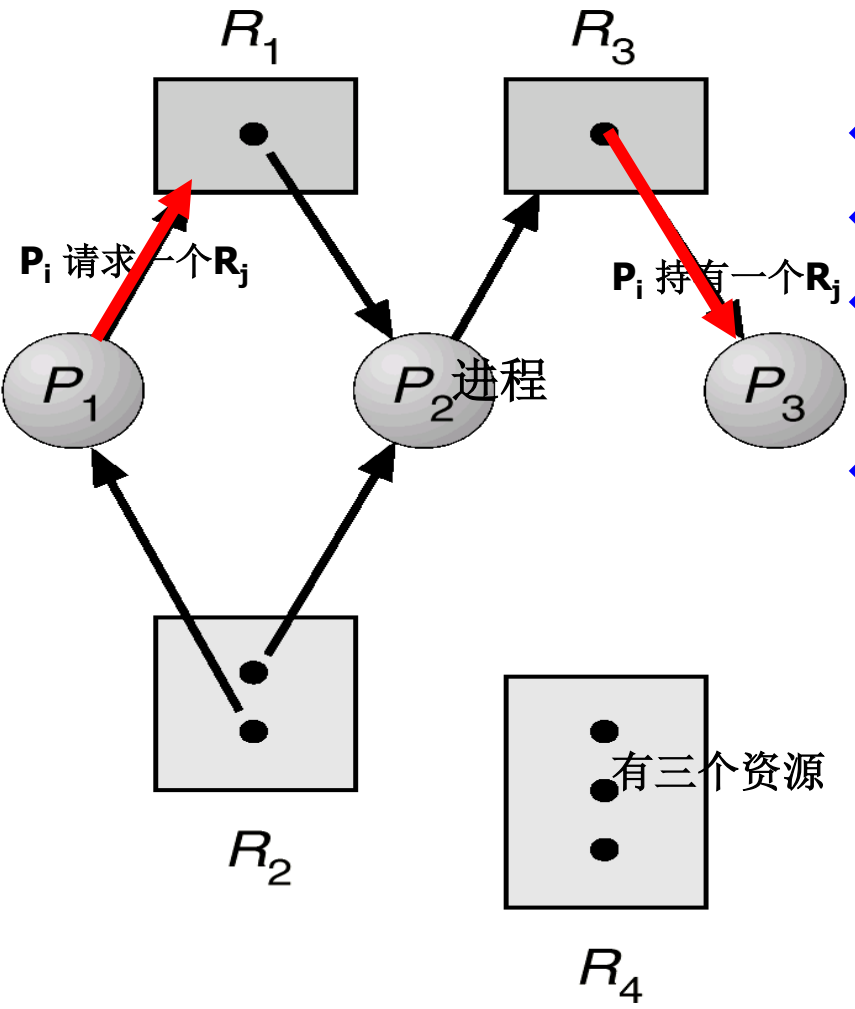
资源分配图



资源分配图又称进程-资源图，它描述了进程和资源间的申请和分配关系，该图是一个有向图，具有以下定义和限制：

- 一个结点集合N和边集合E
- 结点N被分为两个互斥子集
  - 进程结点子集 P = {P1, P2, ..., Pn}
  - 资源结点子集 R = {R1, R2, ..., Rm}
  - N=PUR={P1,P2,...,Pn}U{R1,R2,...,Rm}
- 圆圈表示一进程，方框表示一类资源，其数目由方框中的小圆圈数表示
- 边E
  - 请求边：直接$P_i \rightarrow R_j$，即e=(P_i , R_j)
  - 分配边：$P_i \leftarrow R_j$ 即e= (R_j , P_i)

资源分配图



- 进程结点子集 P = {P1,P2,P3}
- 资源结点子集 R = {R1,R2,R3,R4}
- N=P U R={P1,P2,P3} U {R1,R2,R3, R4}={P1,P2,P3,R1,R2,R3,R4}
- E={(R2, P1), (R2, P2), (P1, R1), (R 1, P2), (P2, R3), (R3, P3)}

资源分配图例

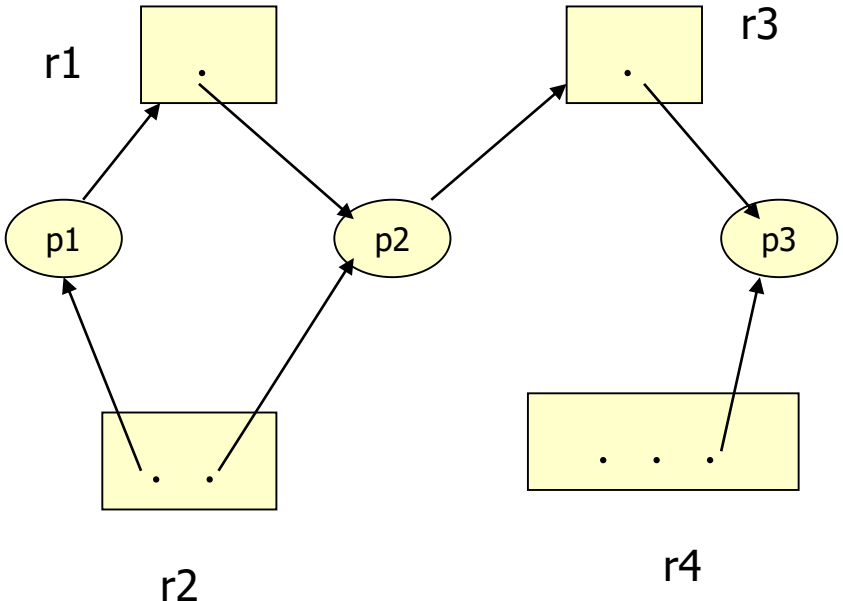
- 设进程集P、资源类集R及边集E如下：

  P={p1，p2，p3}

  R={r1(1)，r2(2)，r3(1)，r4(3)}

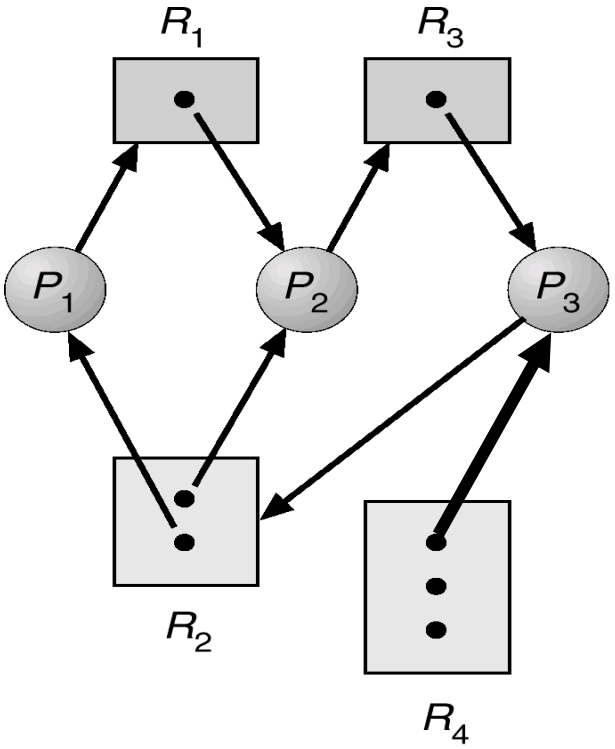
  E={(r1，p2)，(r2，p2)，(r2，p1)，(r3，p3)，(p1，r1)，(p2，r3)，(r4，p3)}

  对应的资源分配图：

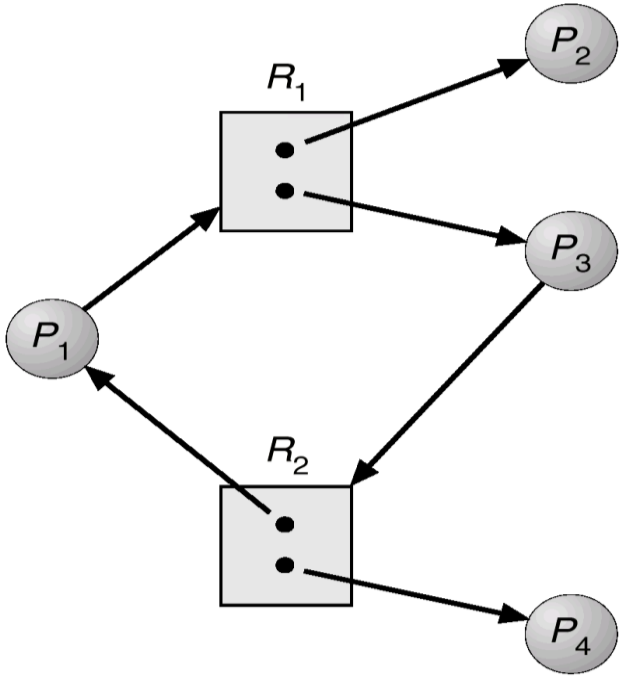


无环路，故不存在死锁

有环路且有死锁的资源分配图



有环但没有死锁的资源分配图



- **重要结论：**如果资源分配图中不存在环路，则系统中不存在死锁；反之，如果资源分配图中存在环路，则系统中可能存在死锁，也可能不存在死锁。

资源分配图的化简（1）

可以通过对资源分配图进行化简，来判断系统是否处于死锁状态。资源分配图中的约简方法如下：

（1）寻找一个既不阻塞又非孤立的进程结点Pi，若无，则算法结束；

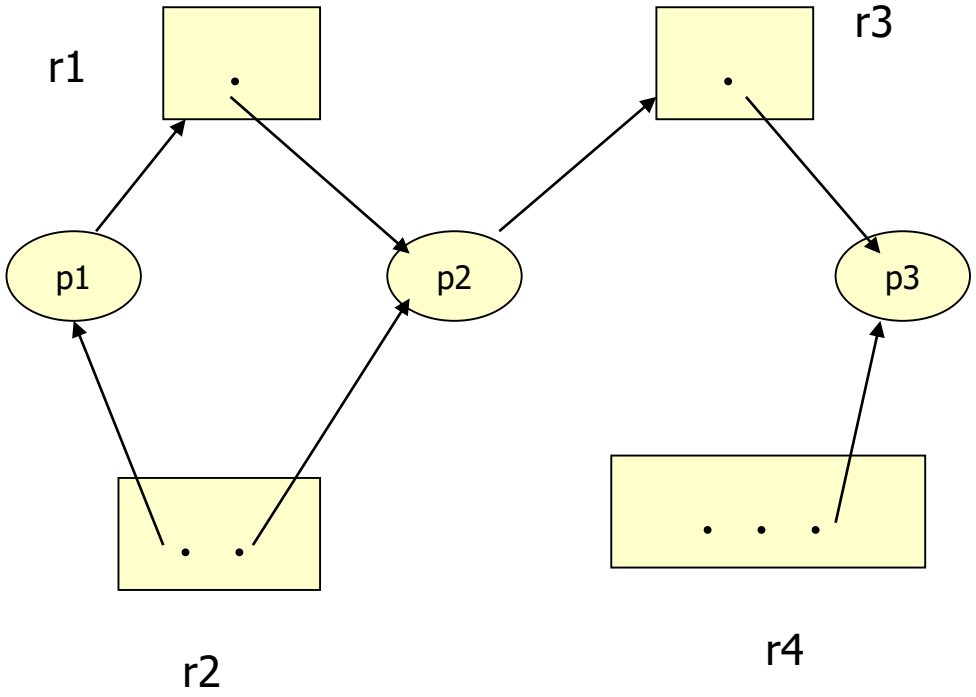
（2）去除Pi的所有分配边和请求边，使Pi成为一个孤立节点；

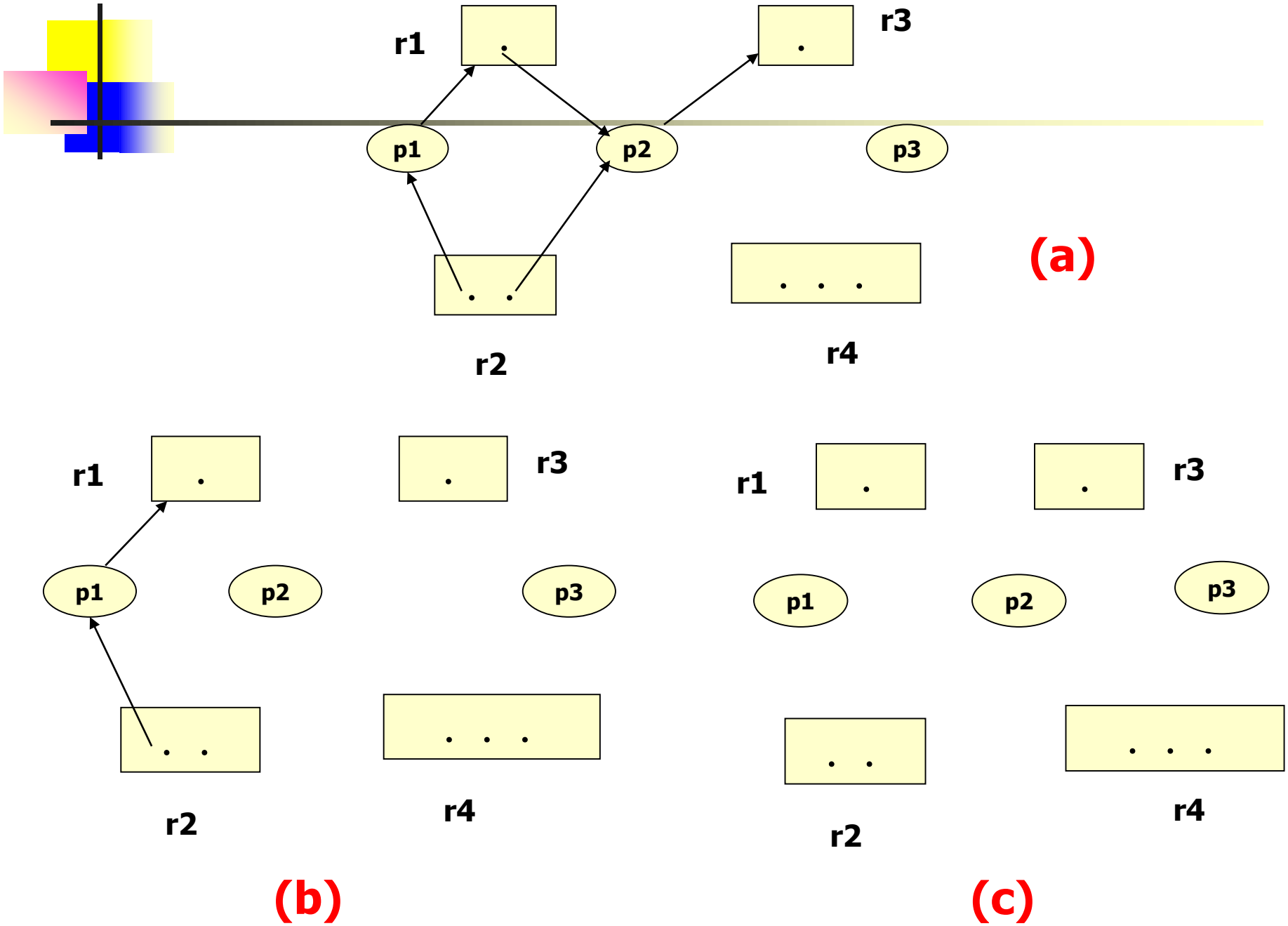
（3）转步骤（1）。

资源分配图的化简（2）

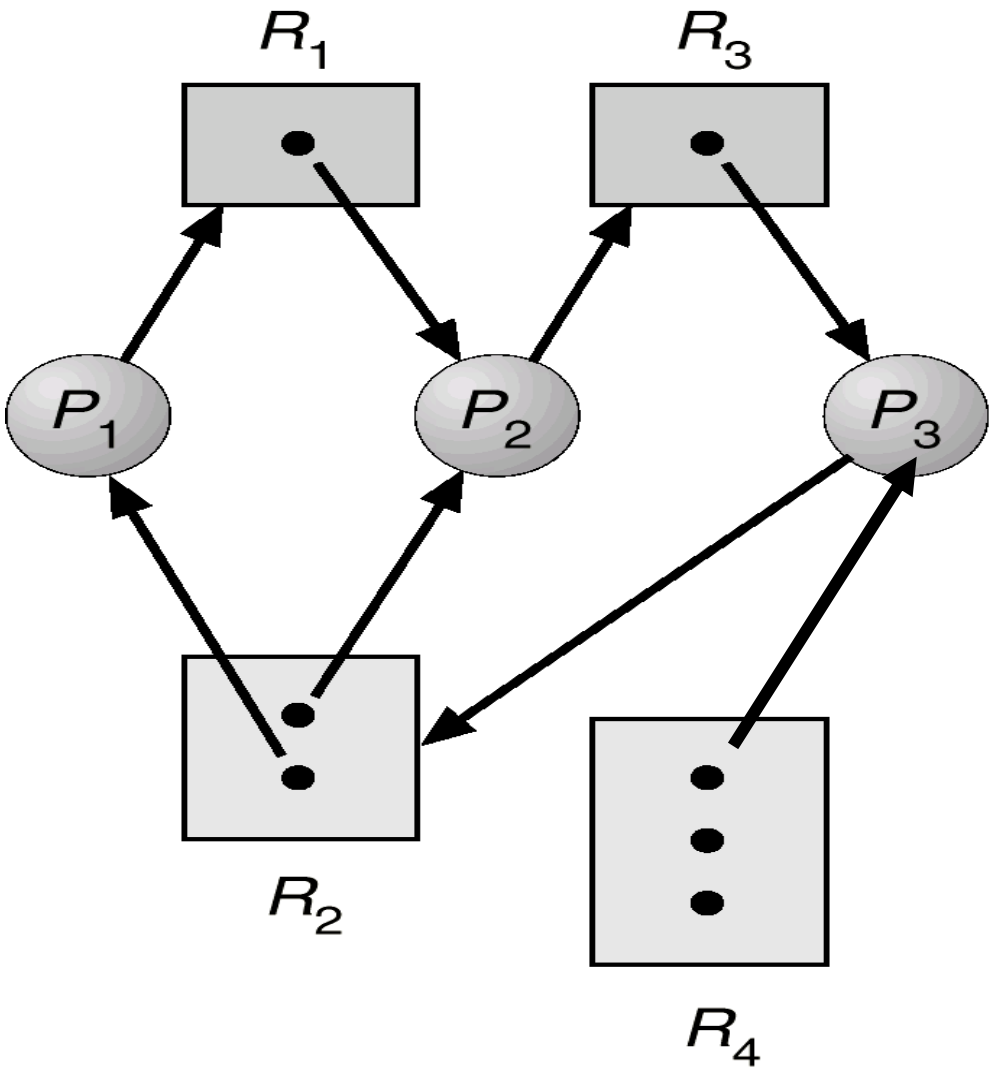
在进行一系列化简后，若能消去图中所有的边，使所有进程都成为孤立结点，则称该图是可完全简化的；反之，称该图是不可完全简化的。

如下图：



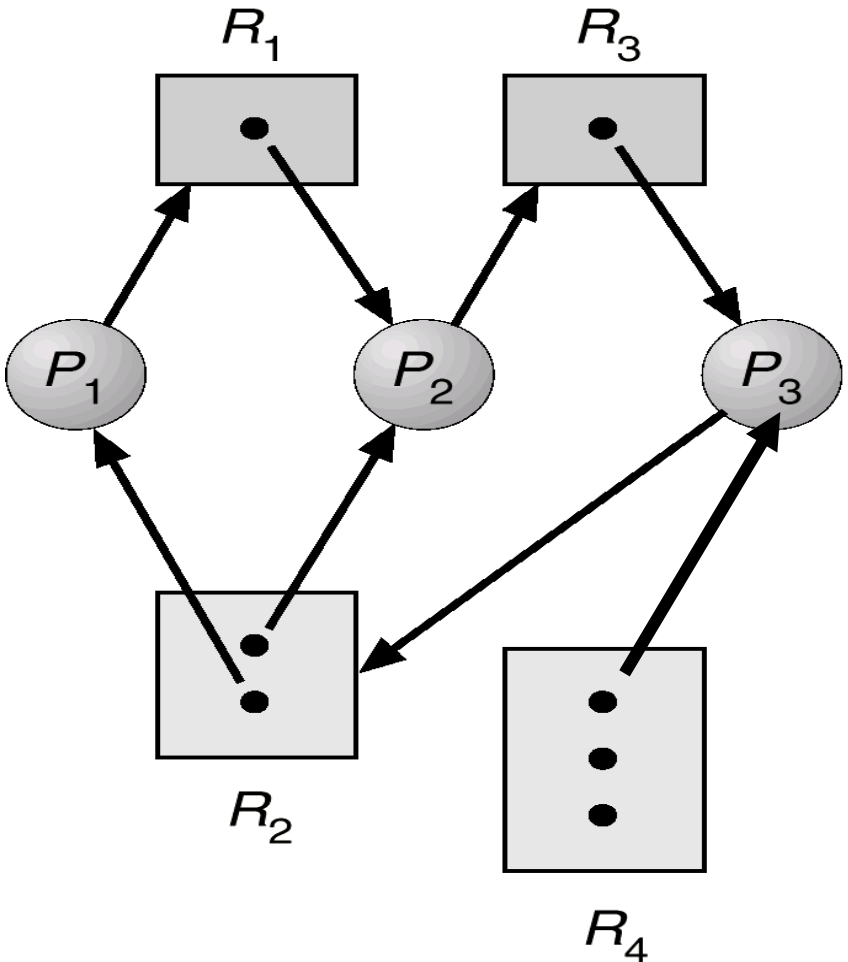

r1
r3
p1
p2
p3
r2
r4
(a)
r1
r3
p1
p2
p3
r2
r4
(b)
r1
r3
p1
p2
p3
r2
r4
(c)

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。



二、死锁检测算法

- 基本思想

获得某时刻t系统中各类可利用资源的数目向量w(t)，对于系统中的一组进程{ P1, P2, ..., Pn}，找出那些对各类资源请求数目均小于系统现有的各类可利用资源数目的进程。这样的进程可以获得它们所需要的全部资源并运行结束，当它们运行结束后释放所占有的全部资源，从而可用资源数目增加，将这样的进程加入到可运行结束的进程序列L中，然后对剩下的进程再作上述考查。如果一组进程{ P1, P2, ..., Pn}中有几个进程不属于序列L中，那么它们会发生死锁。

死锁检测中的数据结构

类似于银行家算法中的数据结构：

◆可利用资源向量:Available[j]=k， 资源Rj有k个可用

◆最大需求矩阵:Max[i，j]=k,进程Pi最大请求k个Rj资源

◆分配矩阵：Allocation[i，j]=k， 进程Pj分配到k个Rj资源

◆需求矩阵：Need[i，j]=k,进程Pj还需要k个Rj资源

三个矩阵的关系：

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

检测算法

算法需要$m \times n^2$的操作

1. 让Work和Finish作为长度为m和n的向量
   (a) Work := Available
   (b) For i = 1, 2, ..., n, if $Allocation_i \neq 0$, then
   Finish[i] := false; otherwise, Finish[i] := true.
2. 找到下标i
   (a) Finish[i] = false
   (b) $Request_i \leq Work$
   如果没有这样的i存在， go to step 4
3. Work := Work + $Allocation_i$
   Finish[i] := true
   go to step 2.
4. If Finish[i] = false, for some i, $1 \leq i \leq n$, then the system is in deadlock state. Moreover, if Finish[i] = false, then P_i is deadlocked.

三、死锁的解除

一旦检测出系统中出现了死锁，就应将陷入死锁的进程从死锁状态中解脱出来，常用的解除死锁方法有两种：

- **资源剥夺法**：当发现死锁后，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.
- **撤消进程法**：
  - 最简单撤消进程的方法是使全部死锁的进程夭折掉；但以前工作全部作废，损失可能很大
  - 另一方法是按照某种顺序逐个地撤消进程，直至有足够的资源可用，死锁状态消除为止