







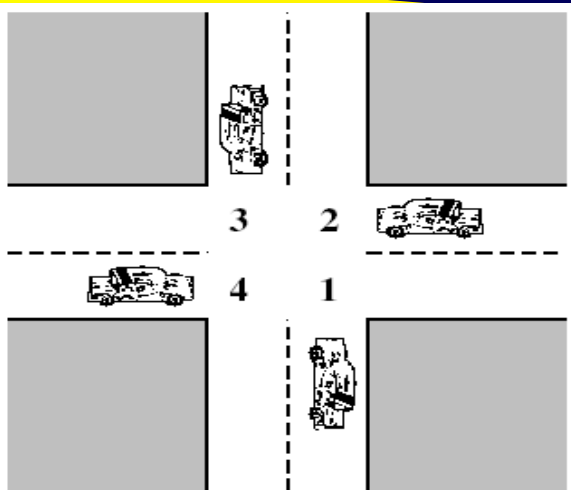




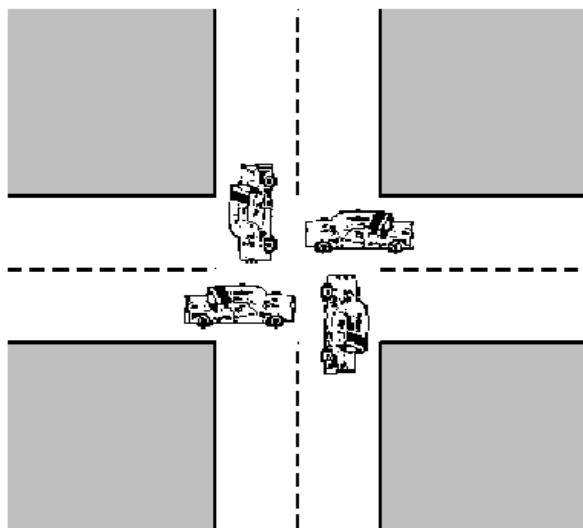
内容导航:

-  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



(a) Deadlock possible

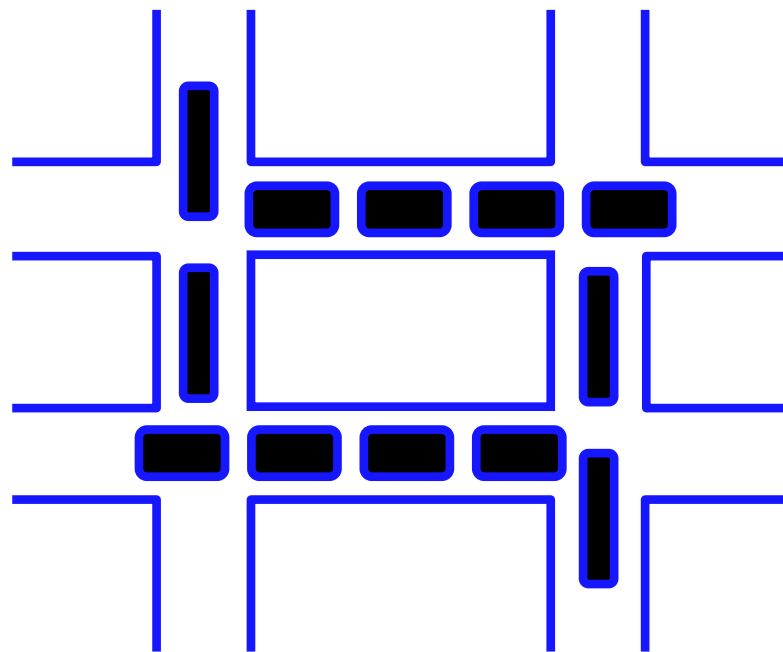


(b) Deadlock

显然各路车队等待的事件都不会发生(假设它们都不改变行车方向)。这样若不采用特殊方法,它们将永远停留在这“井”字形的路上,而处于**死锁**状态。

在计算机系统中,进程发生死锁与上述事例实质上是一样的。进程是因相互**竞争资源**而导致死锁的,与四路车队(可视为进程)竞争路口(可视为资源)类似。计算机系统中有各种资源,如**外设、数据、文件、程序**等。

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。



死锁问题(deadlock) —— OS中重点之一

1. Newton,G.:“Deadlock Prevention,Detection,and Resolution: An annotated Bibliography,” Operating Systems Review,vol.13,pp.33-44,April 1979.
2. Zobel,D.:“The Deadlock Problem: A Classifying Bibliography,”Operating Systems Review,vol.17,pp6-16,Oct.1983.
3. Karacali ,B., Tai,K>C., and Vouk,M.A.:“Deadlock Detection of EFSMs Using Simultaneous Reachability Analysis,” Proc. Int’l Conference on Dependable systems and networks(DSN 2000),IEEE,pp.315-324,2000.

- 早期的操作系统对申请某种资源的进程，若该资源尚未分配时，立即将该资源分配给这个进程。
- 对资源不加限制地分配可能导致进程间由于竞争资源而相互制约以至于无法继续运行的局面，人们把这种局面称为死锁 (deadlock)。
- 死锁问题在1965年由Dijkstra 发现，并随着计算机技术的发展，逐渐成为人们关心的问题。
- 什么是死锁？其确切的定义是什么？死锁是怎么产生的？用什么方法来解决死锁？这些正是今天我们要讨论的问题。



可重用性资源和可消耗性资源

- **可重用性资源**：可供用户重复使用多次
 - 只能分配给一个进程使用，不允许多个进程共享
 - 遵循：1) 请求资源；2) 使用资源；3) 释放资源
 - 系统中资源数目相对固定
 - 计算机中大部分资源都是可重用资源，如内存等。
- **可消耗性资源**：又称临时性资源，进程运行过程中，动态创建和消耗
 - 资源数目运行期间不断变化
 - 进程运行过程中，可不断创造资源
 - 进程运行过程中，可请求消耗资源
 - 典型的例子用于进程间通信的消息



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。这类资源不会引起死锁
 - 例如，CPU（优先级高的抢占）和内存（中级调度）
- **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
 - 例如，打印机、磁带机、刻录机

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。

```
P1  
.....  
open (F1,w);  
open (F2, w);
```

```
P2  
.....  
open (F2,w);  
open (F1,w);
```

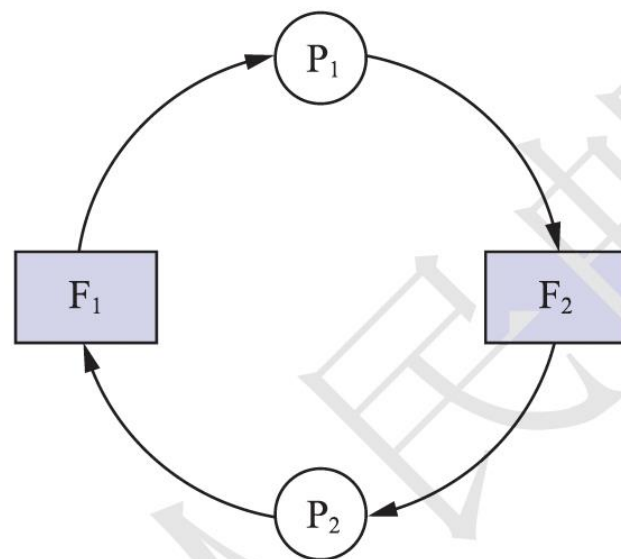


图 3-13 共享文件时的死锁

- 并行执行，P1先打开F1和F2，P2再执行时会阻塞，P1执行完，P2再继续；
- 如果P1打开F1的，同时P2打开了F2,...

- 箭头从进程指向文件：进程请求该资源
- 箭头从文件指向进程：资源分配给进程
- 形成环路，进入死锁

2 竞争可消耗性资源引起死锁

➤ **临时性资源**，是指由一个进程产生，被另一个进程使用一短暂时间后便无用的资源，故也称之为**消耗性资源**，它也**可能**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 ); ...
```

改变顺序

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

OK

死锁

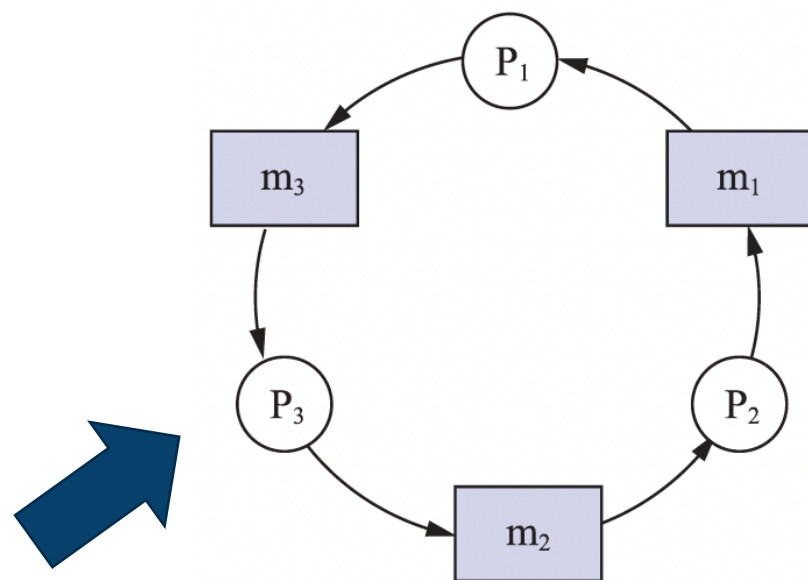


图 3-14 进程之间通信时的死锁

3 进程推进顺序不当引起死锁

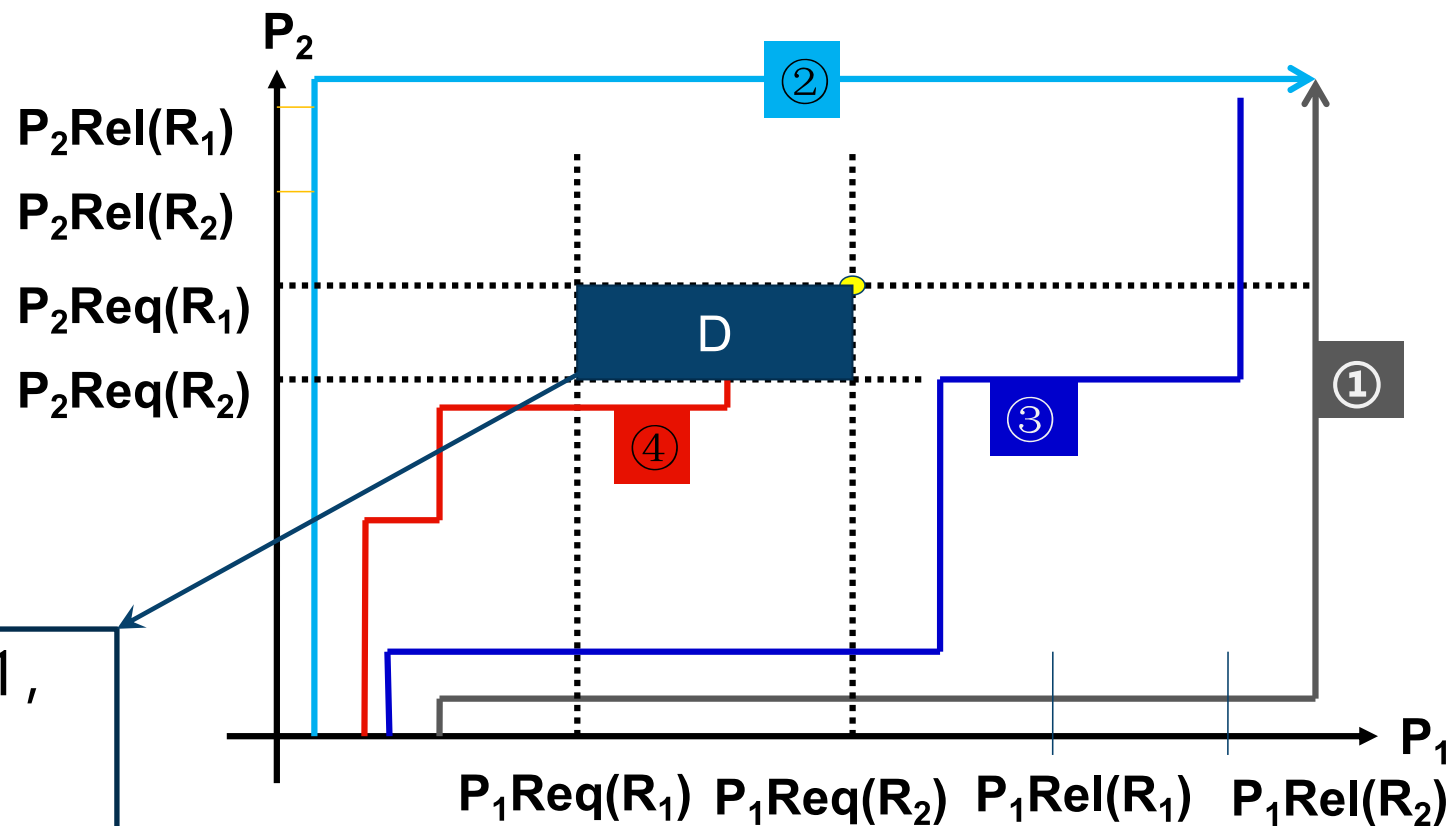
➤ 进程推进顺序合法
(折线1、2、3)

➤ 进程推进顺序非法
(折线4)

D: 不安全态, 此时P1占用了R1,
而P2占用了R2。

如果系统再继续往前推荐, 死锁便
会发生

系统中只有一台打印机R1和一台磁
带机R2供进程P1和P2使用



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。

2

竞争可消耗性资源引起死锁

- 临时性资源，是指由一个进程产生，被另一个进程使用一短暂时间后便无用的资源，故也称之为消耗性资源，它也可能引起死锁

3

进程推进顺序不当引起死锁

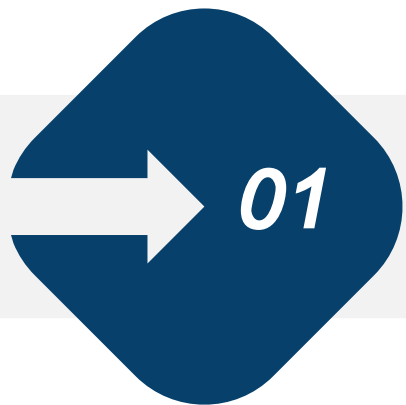
- 进程推进顺序合法
- 进程推进顺序非法

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。

如果一个进程集合中的每个进程都在等待只能由该组进程中的其他进程才能引发的事件，那么，该组进程是死锁的。

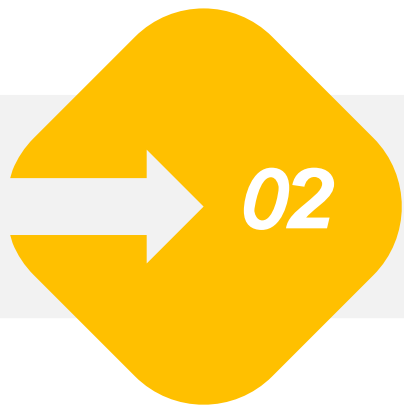
— 《现代操作系统》，Andrew S. Tanenbaum

互斥



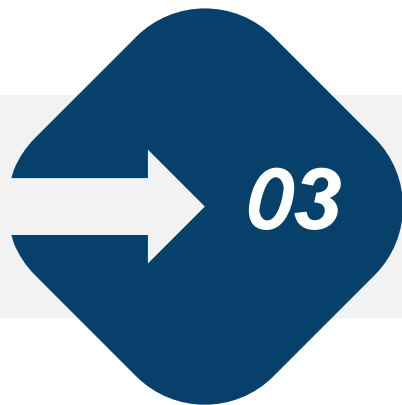
- 一段时间内某资源只能被一个进程占用。

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

上述死锁的四个必要条件不是彼此独立的，比如条件 4 包含了前三个条件。将它们分别列出是为了方便我们研究各种防止死锁的方法。

Coffman,E.G., Elphick,M.J.,and Shoshani,A.: "System Deadlocks," Computing Surveys,vol.3,pp.67-78,June 1971.



设计无死锁的OS：确保系统永远不会进入死锁状态

- 死锁预防 (deadlock prevention)
- 死锁避免 (deadlock avoidance)



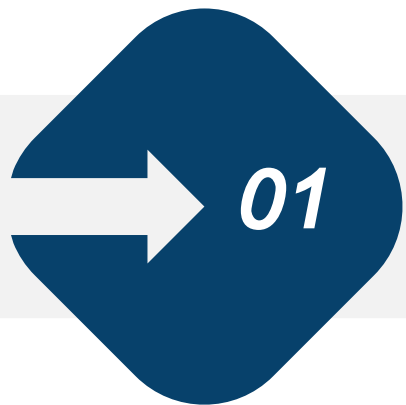
先礼后兵：允许系统进入死锁状态，然后恢复系统

- 死锁检测 (deadlock detection)
- 死锁恢复 (deadlock recovery)



鸵鸟策略：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

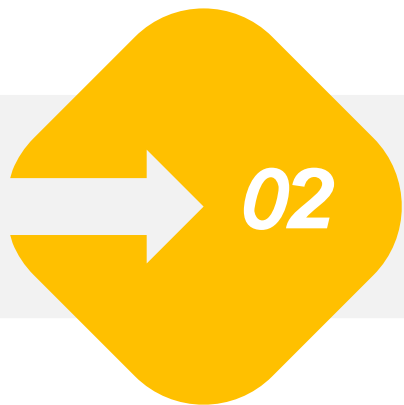
预防死锁



- 破坏死锁的四个必要条件中一个或几个。

(deadlock prevention)

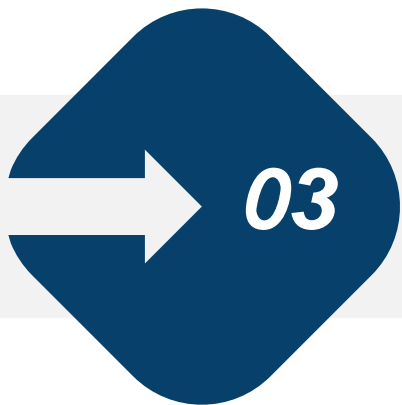
避免死锁



- 在资源动态分配时，防止系统进入不安全状态。

(deadlock avoidance)

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。

(deadlock detection)

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。

(deadlock recovery)



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

1. 思想：不采取任何措施，对死锁视而不见
2. 实例：UNIX系统、Windows（还有其它许多系统）
3. 由于不预防、避免死锁，故系统中可能发生死锁。而发生时又无法知道（因不检测），造成系统崩溃，需要重启
4. 之所以被某些系统采用，是因为死锁不常发生

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
- 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。








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。

- 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

预防和避免的差别是什么？

1. 死锁 \rightarrow 四个必要条件

2. $\neg(\text{四个必要条件}) \rightarrow \neg(\text{死锁})$

命题与逆否命题是等价

3. 四个必要条件 \rightarrow 死锁 (不一定)

4. 死锁防止是严格破坏4个必要条件之一，一定不出现死锁；而**死锁的避免是不那么严格地限制死锁必要条件的存在**，其目的是提高系统的资源利用率。万一当死锁有**可能出现**时，就**小心避免**这种情况的发生。

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。

- 允许进程动态申请资源
- 将系统分为安全态和不安全态
- 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 → 确保系统永远不会进入不安全状态



做法：每次进行资源分配时，首先检测下资源分配后系统处于何种状态，若有风险，则将申请资源的进程阻塞

- 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。

系统中有12台磁带机。由进程P1、 P2和P3共享，运行一段时间后（假设此时为T0），情况如下表所示：

进 程	最 大 需 求	已 分 配	可 用
P ₁	10	5	3
P ₂	4	2	
P ₃	9	2	

1. 系统在T0时刻是否安全？
2. 如果接下来系统把剩余3台其中的1台分配给P3，系统是否安全？

系统在T0时刻是否安全?

结果: T0时刻系统处于安全状态。(安全序列 $\langle P_2, P_1, P_3 \rangle$)

P_1	5(5)	5(5)	5(5)	10(0)	☹	☹	☹
P_2	2(2)	4(0)	☹	☹	☹	☹	☹
P_3	2(7)	2(7)	2(7)	2(7)	2(7)	9(0)	☹
可用	3	1	5	0	10	3	12

安全!!!

如果接下来系统把剩余3台其中的1台分配给P3，系统是否安全？

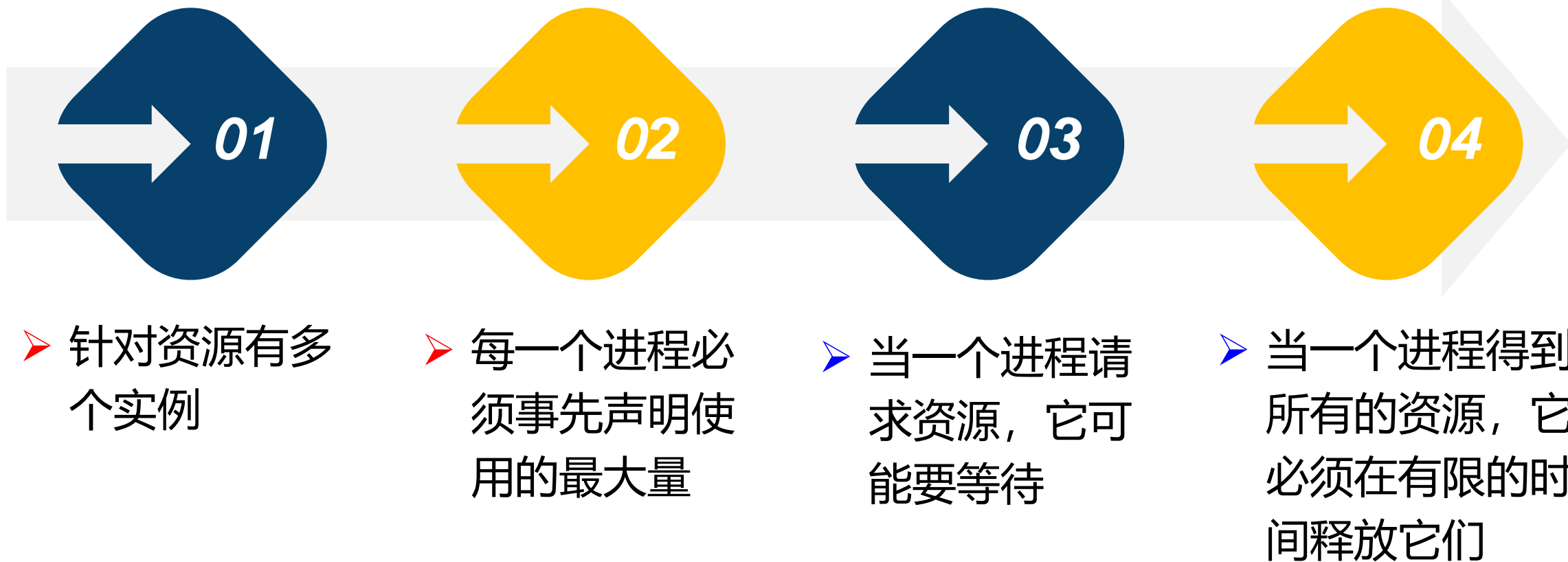
结果：T1时刻系统处于不安全状态。

P ₁	5(5)	⇒	5(5)	⇒	5(5)
P ₂	2(2)		4(0)		☞
P ₃	3(6)		3(6)		3(6)
可用	2		0		4

不安全!!!

- 仿照银行**发放贷款**采取的控制方式而设计的死锁避免算法
- 为控制风险，银行在给客户发放贷款之前，先**审核客户的信用额度**。信用额度由客户提出请求，银行审核后决定是否批准。批准后，客户对资金的**使用是按阶段的**。
- 假定所有客户信用良好，能够按期还款。银行审核考虑的因素就是银行是否有钱满足客户。即：只要客户的信用额度不超过银行的全部流动资产，即予以批准，否则拒绝。
- 面对客户请求，银行有两种可选策略：1) 只要银行有钱就发放贷款；2) **需考虑发放贷款后银行面临的风险**

银行家算法的实质就是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。



n为进程的数目， m为资源类型的数目

- 可用资源向量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
- 最大需求矩阵Max: n x 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
- 已分配矩阵Allocation: n x m 矩阵。 如果Allocation[i,j]=k,那么进程P_j当前分配了k个资源R_j的实例
- 需求矩阵Need: n x m 矩阵。 如果Need[i,j]=k,那么进程P_j还需要k个资源R_j的实例

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

最大矩阵

—

已分配矩阵

=

需求矩阵

可用资源向量

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			

Request_i = 进程 P_i 的资源请求向量, 如果 $\text{Request}_i[j] = k$ 则进程 P_i 想要资源类型为 R_j 的 k 个实例, 银行家算法会按照下列步骤进行检查

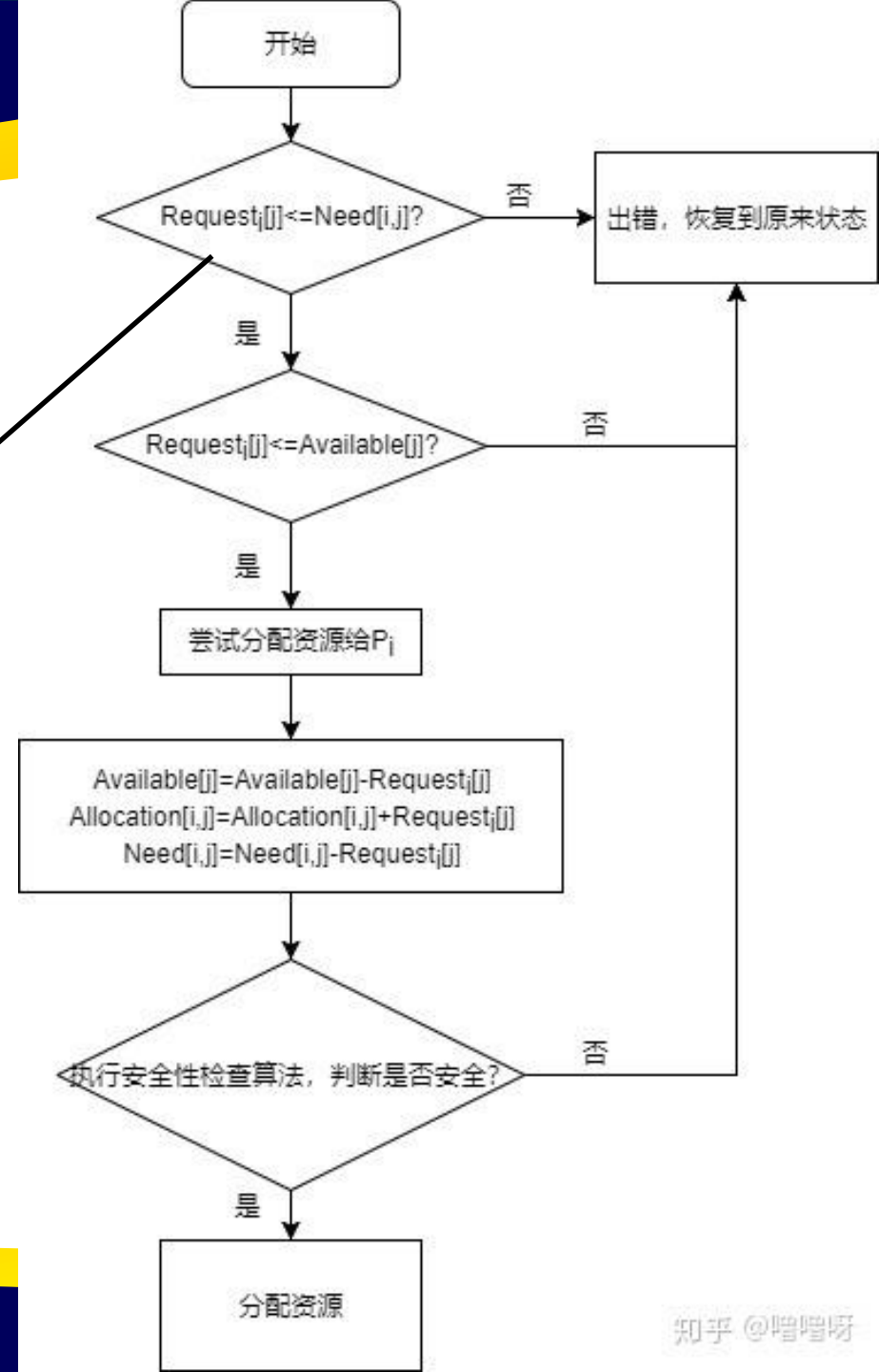
- ① 如果 $\text{Request}_i[j] \leq \text{Need}[i,j]$ 转 step 2. 否则报错, 因为进程请求超出了其声明的最大值
- ② 如果 $\text{Request}_i[j] \leq \text{Available}[j]$, 转 step 3. 否则 P_i 必须等待, 因为资源不可用
- ③ 假设通过修改下列状态来试探着分配请求的资源给进程 P_i :
 - $\text{Available}[j] := \text{Available}[j] - \text{Request}_i[j];$
 - $\text{Allocation}[i,j] := \text{Allocation}[i,j] + \text{Request}_i[j];$
 - $\text{Need}[i,j] := \text{Need}[i,j] - \text{Request}_i[j];$
- 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



银行家算法——算法流程

$Request_i$ = 进程 P_i 的资源请求向量, 如果 $Request_i[j] = k$ 则进程 P_i 想要资源类型为 R_j 的 k 个实例

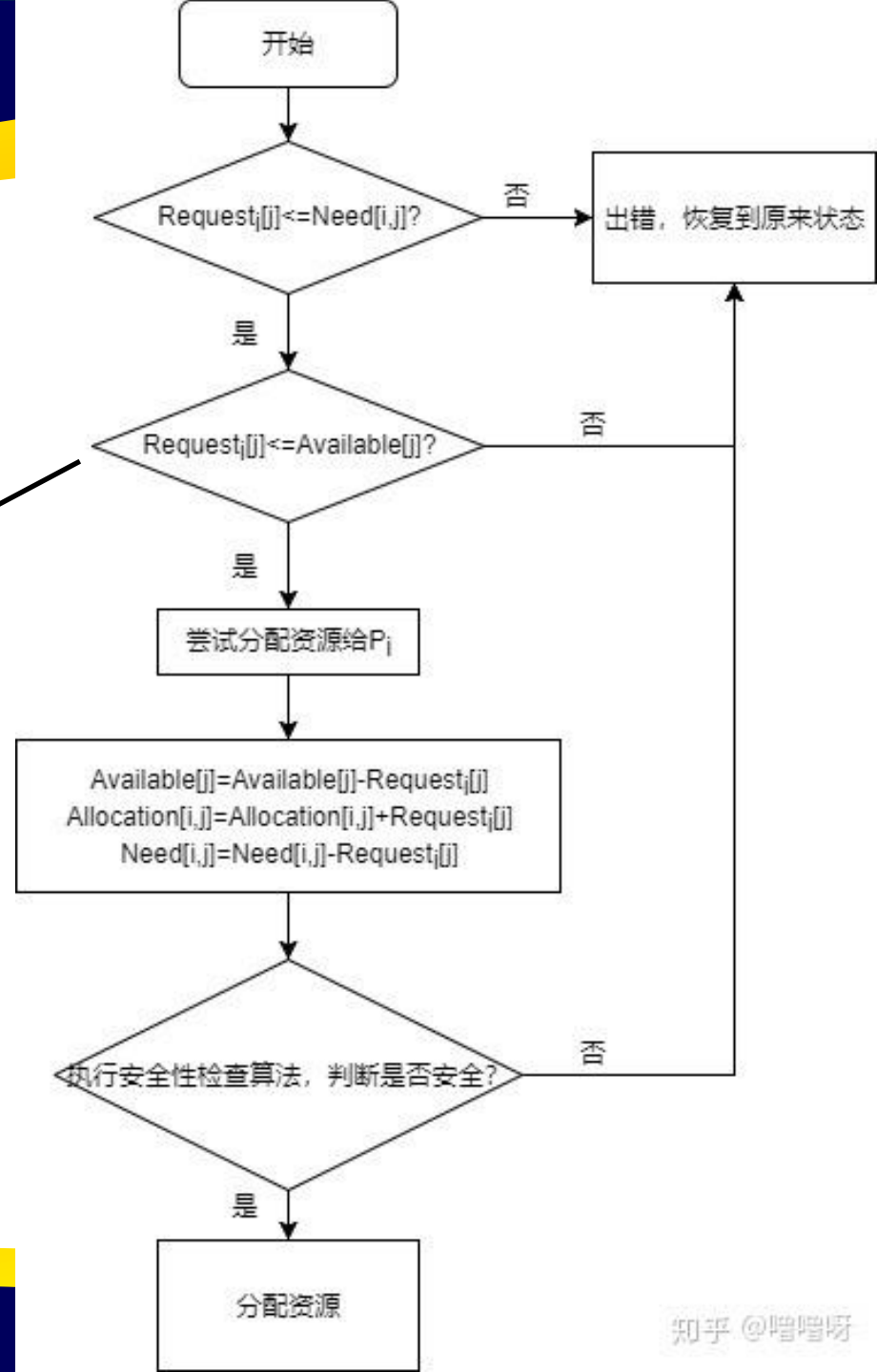
- ① 如果 $Request_i[j] \leq Need[i,j]$ 转 step 2. 否则报错, 因为进程请求超出了其声明的最大值





银行家算法——算法流程

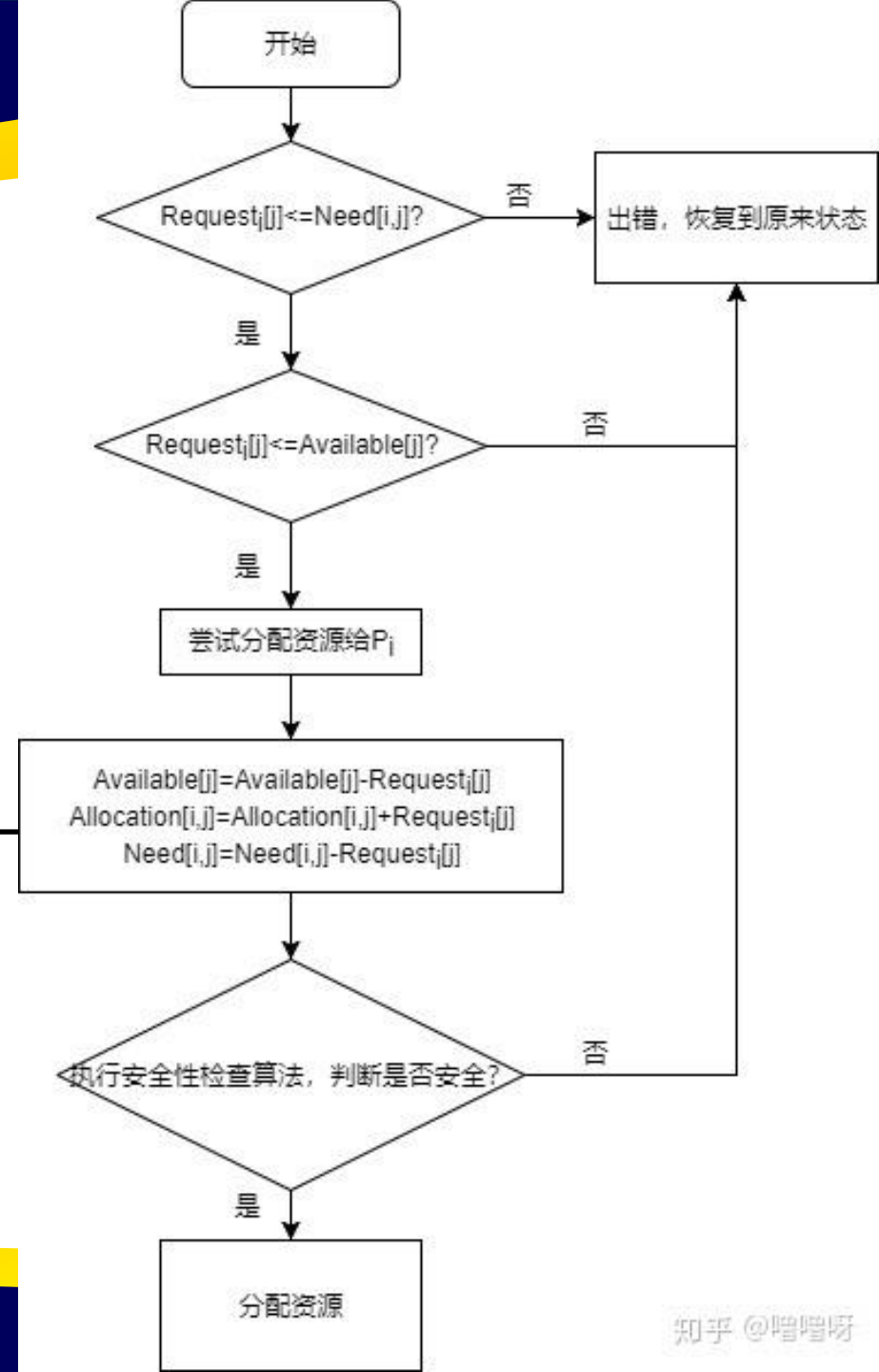
- ② 如果 $\text{Request}_i[j] \leq \text{Available}[j]$, 转 step 3.
3. 否则 P_i 必须等待, 因为资源不可用





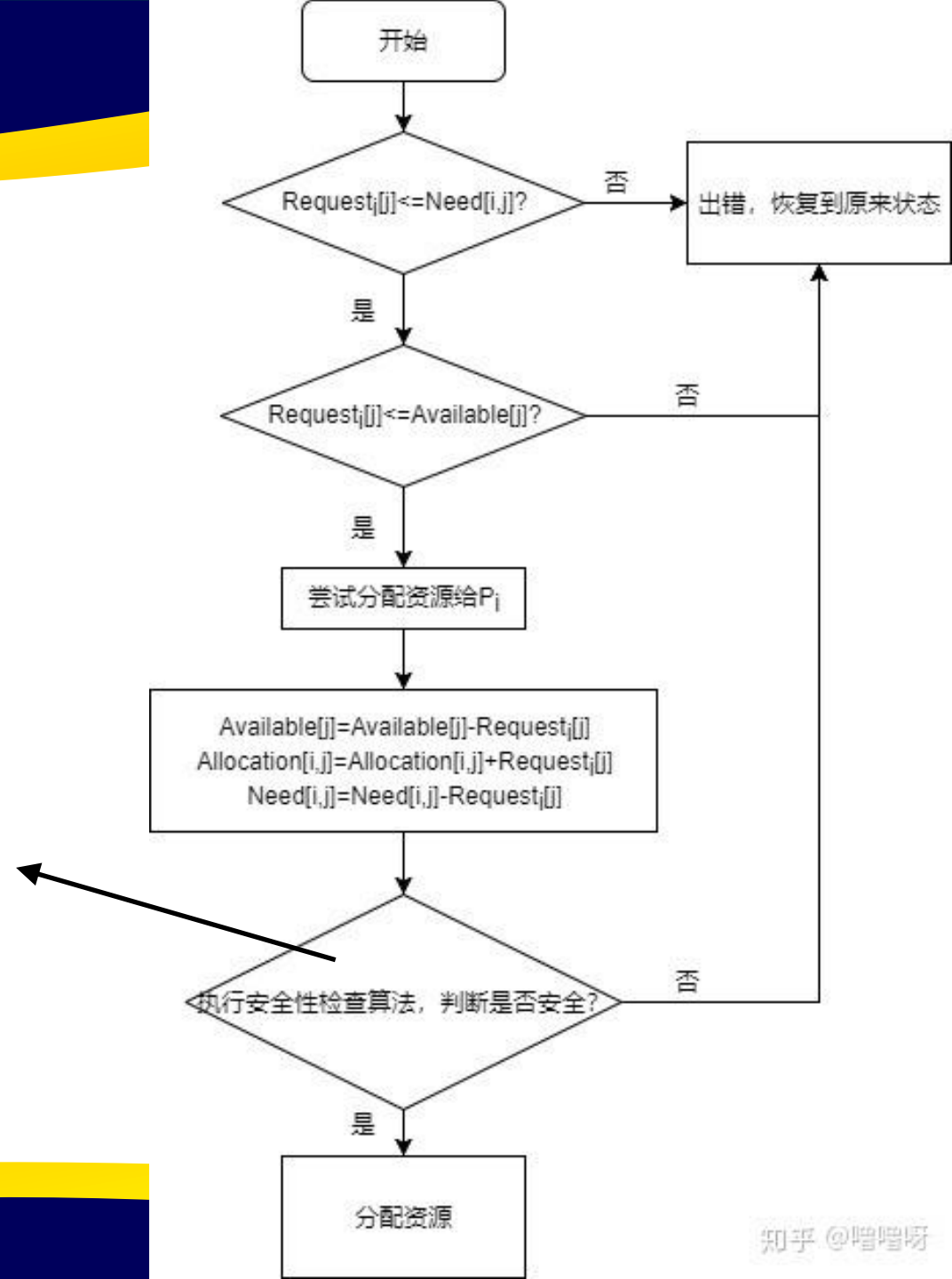
③ 假设通过修改下列状态来**试着**分配请求的资源给进程 P_i ：

- $Available[j] := Available[j] - Request_i[j];$
- $Allocation[i,j] := Allocation[i,j] + Request_i[j];$
- $Need[i,j] := Need[i,j] - Request_i[j];$



④ 系统执行安全性算法

- 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



01

初始化长度为m和n的工作向量Work和完成向量Finish

$Work := Available$

$Finish[i] = false$ for $i = 1, 3, \dots, n$.

02

查找i

(a) $Finish[i] = false$

(b) $Need[i,j] \leq Work[j]$

如果i不存在, 转向步骤4.

03

$Work := Work + Allocation_i$

$Finish[i] := true$

转向步骤4.

04

如果对所有i的 $Finish[i] = true$, 则系统处在安全状态

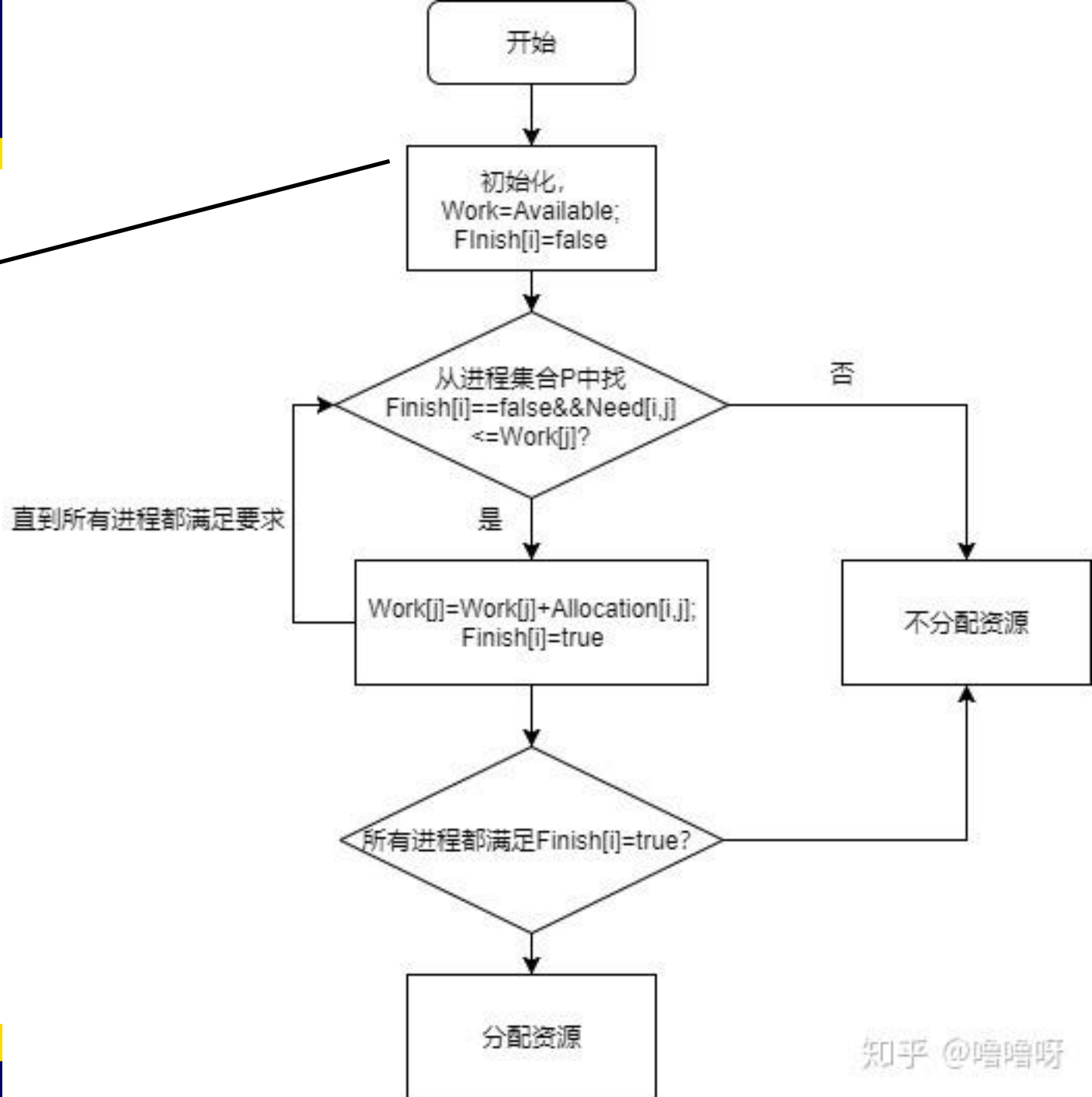
安全性算法主要检测当前的系统状态会不会发生死锁。

01

初始化长度为m和n的工作向量
Work和完成向量Finish(标志)

$Work := Available$

$Finish[i] = false$ for $i = 1, 3, \dots, n$.



02

查找i

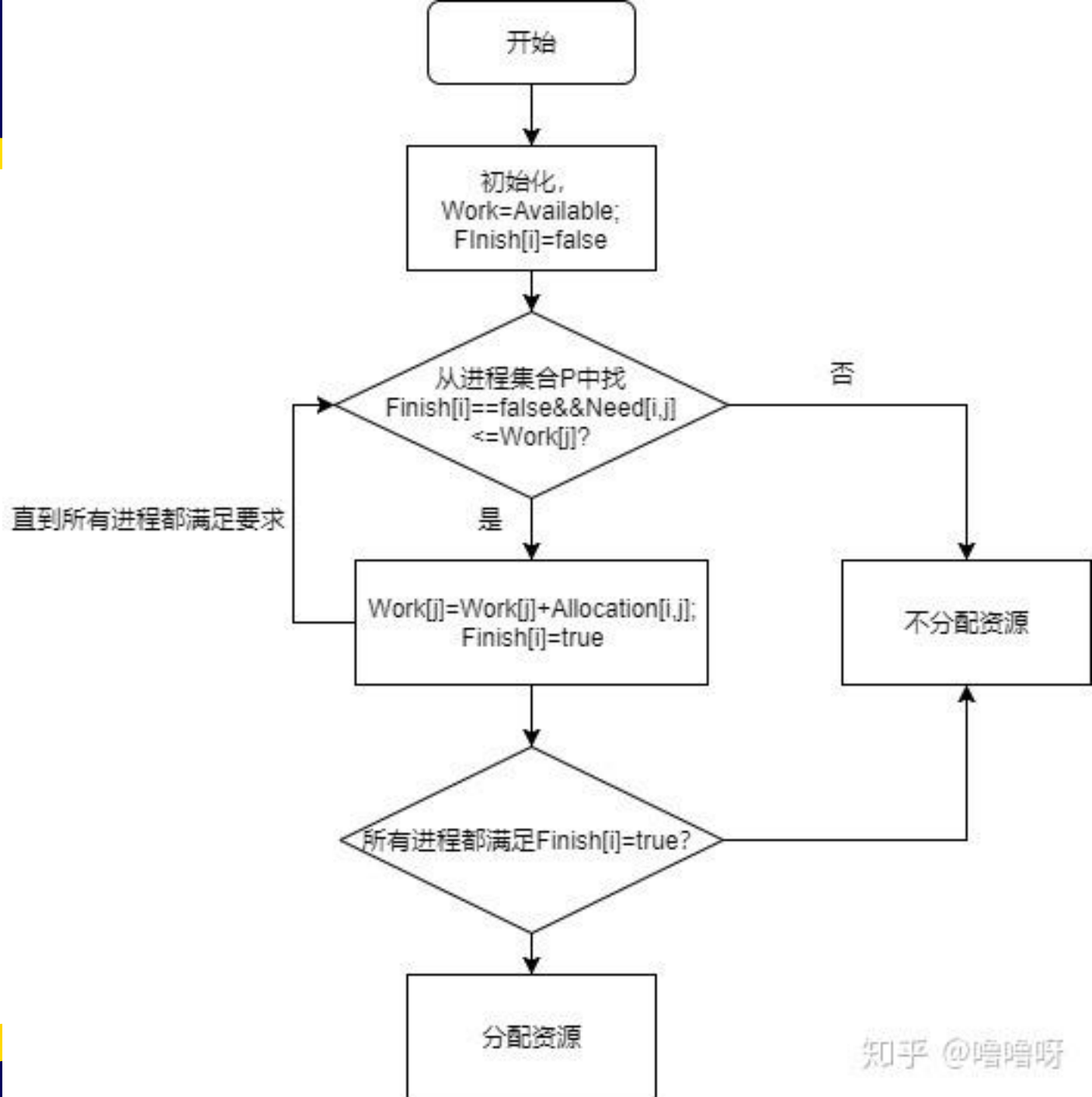
从进程集合中查找一个能同时满足下述条件的进程i

(a) $Finish[i] = false$

(b) $Need[i,j] \leq Work[j]$

如果i不存在，转向步骤4.

还没有进行 $\&\& Need \leq Available$




```
Work := Work + Allocation[i,j]
Finish[i] := true
转向步骤4.
```

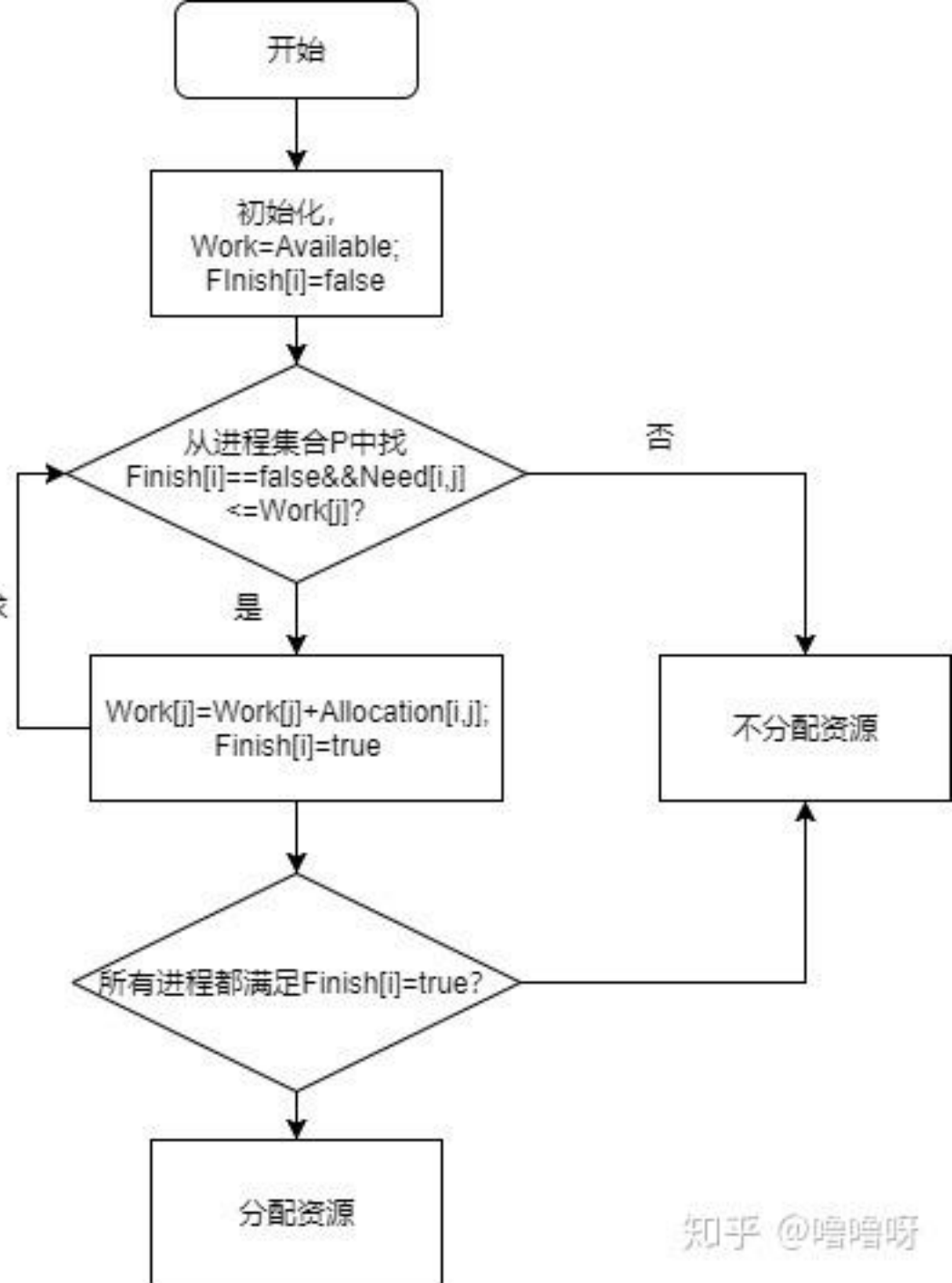


04

所有 i 的 $\text{Finish}[i] = \text{true}$?

如果是, 则表示系统处于安全状态,
否则系统处于不安全状态

直到所有进程都满足要求



- ◆ 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			

➤ 系统处在安全状态?安全序列是?

➤ 矩阵的内容。Need被定义为Max – Allocation

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 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

➤ 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

检查Request \leq 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	(2 0 0)	0	2	0	(1 2 2)			
P ₂	3	0	2		6	0	0				
P ₃	2	1	1		0	1	1				
P ₄	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 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 ₀	7	4	5	7	4	3	0	1	0	7	5	5	true
P ₂	7	5	5	6	0	0	3	0	2	10	5	7	true

执行安全算法表明序列<P₁, P₃, P₄, P₀, P₂> 满足要求









补充思考:

- P₄的请求(3,3,0)是否可以通过?
- P₀的请求(0,2,0)是否可以通过?

1. 假设资源P1申请资源，银行家算法先**试探着**分配给它，若申请的资源数量小于等于Available，然后接着判断分配给P1后剩余的资源，能不能使进程队列的某个进程执行完毕，若没有进程可执行完毕，则系统处于不安全状态（即此时没有一个进程能够完成并释放资源，随时间推移，系统终将处于死锁状态）
2. 若有进程可执行完毕，则假设回收已分配给它的资源（剩余资源数量增加），把这个进程标记为可完成，并继续判断队列中的其它进程，**若所有进程都可执行完毕，则系统处于安全状态，并根据可完成进程的分配顺序生成安全序列**



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。

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 $R_j \rightarrow P_i$

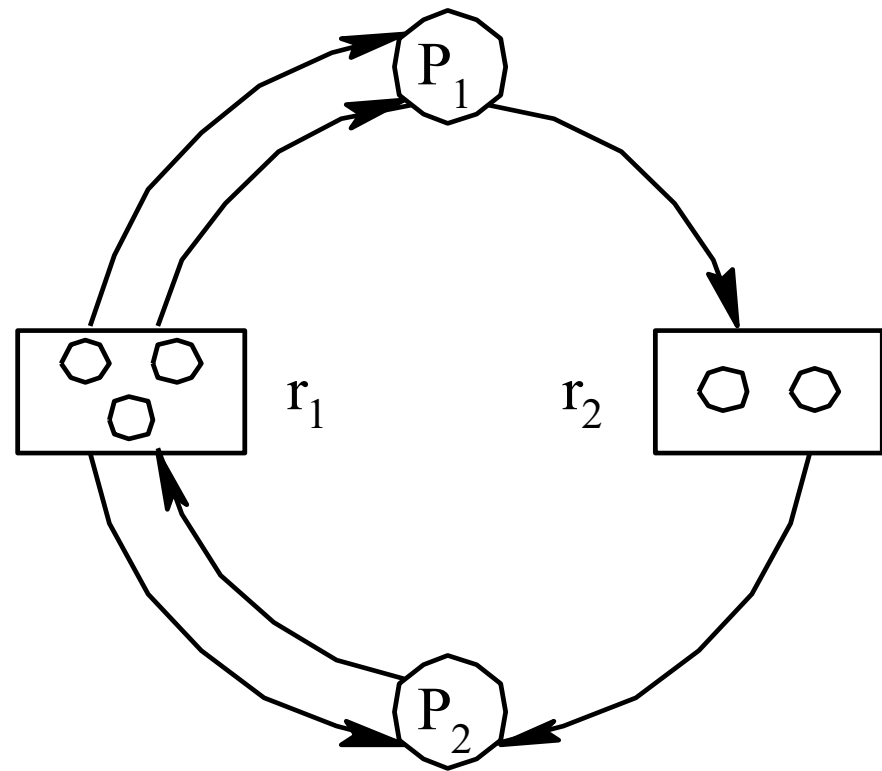


图 每类资源有多个时的情况

1. 资源分配图在系统中是随时间变化的。
2. 当进程 P_i 请求 r_j 时, 将一条请求边(P_i, r_j)加在图中
3. 若此请求被满足(分给 P_i 一个 r_j 类的资源), 则将这个请求边改成分配边(r_j, P_i)
4. 当进程 P_i 释放一个 r_j 时, 便去掉分配边(r_j, P_i)。

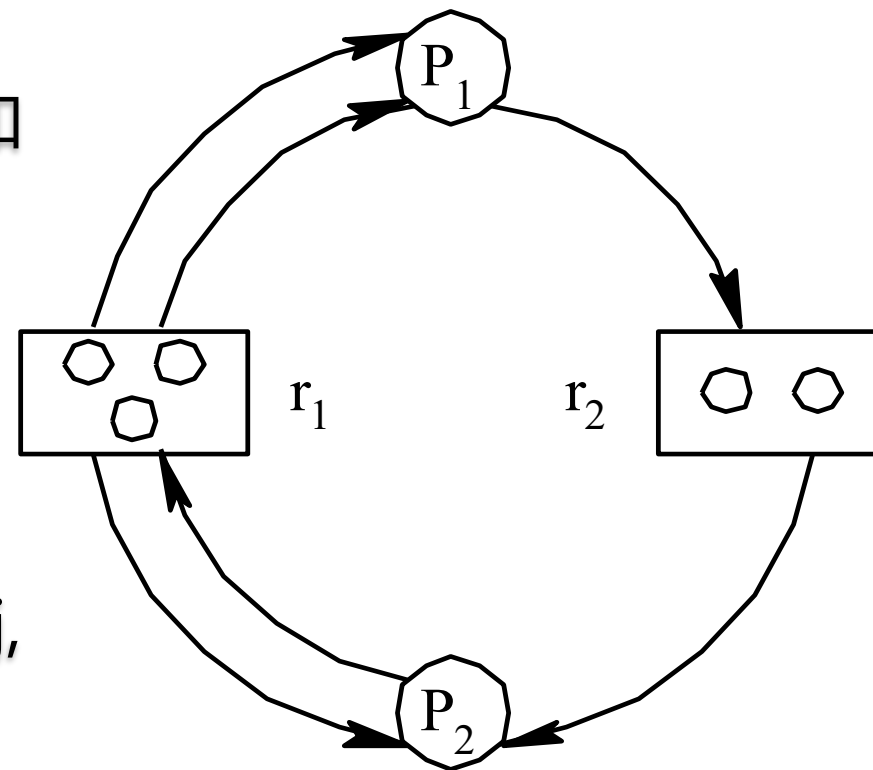


图 每类资源有多个时的情况

可以通过简化资源分配图来检测当前系统是否处于死锁状态

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。

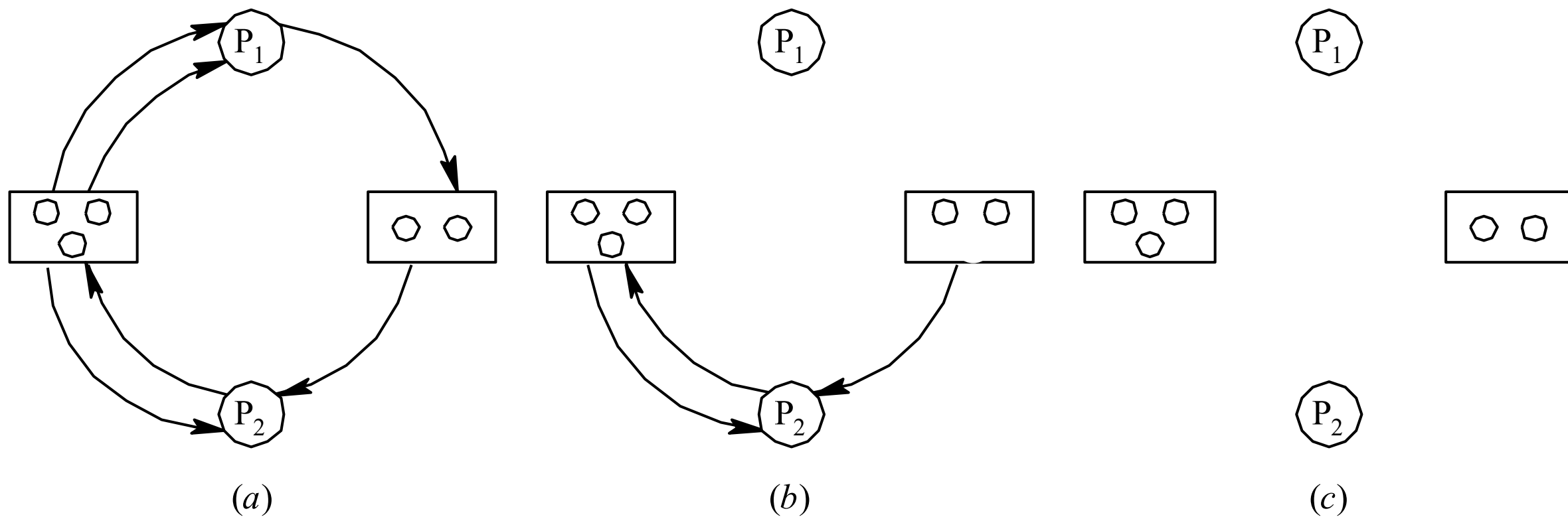


图 资源分配图的简化

若能消除所有的分配边，使得所有进程节点成为孤立节点，那么称该图为可**完全简化**的（相当于找到一个安全序列），否则成为不可完全简化的

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？

➤ 引理

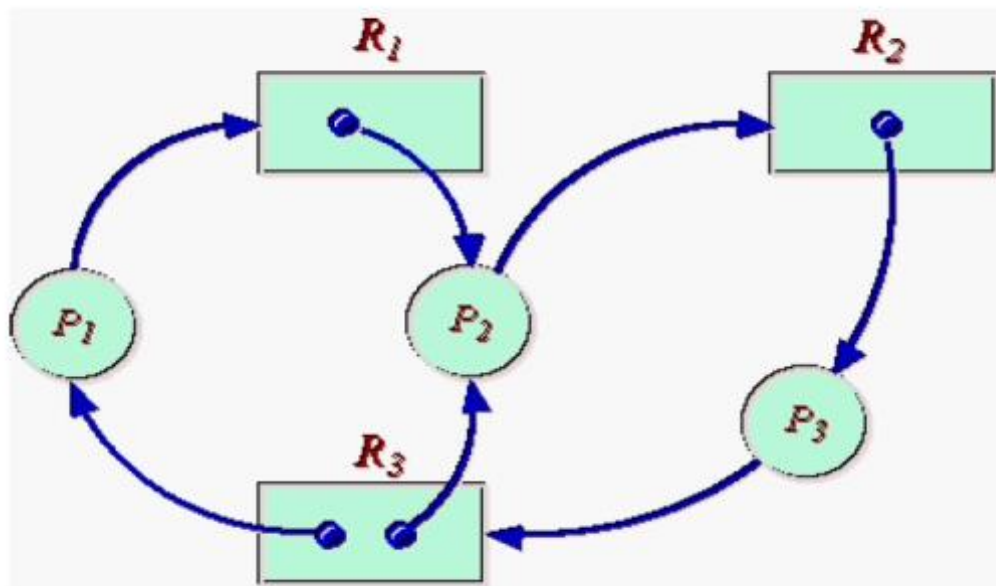
有关文献证明，所有的简化顺序，都将得到相同的不可简化图。

➤ 死锁定理

S状态为死锁状态的充分条件是：当且仅当S的资源分配图是不可完全简化的。

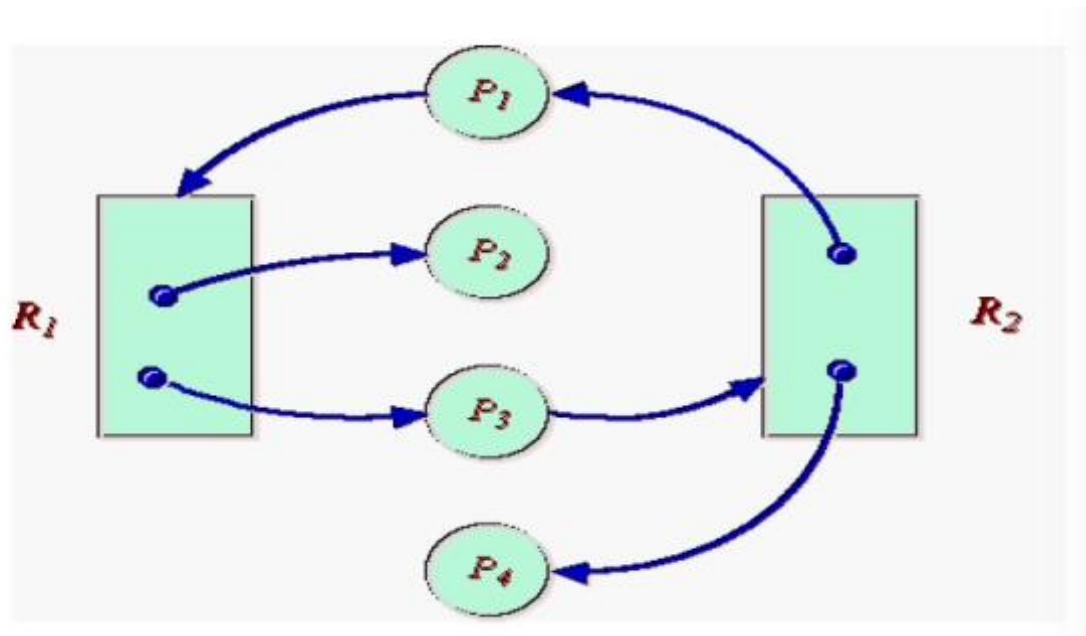


如果图没有环，那么不会有死锁！



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。



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 + Allocation_i ;

$L_i \cup L$;

end

end

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

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。



死锁检测算法——示例



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

Request

A B C

P₀ 0 0 0

P₁ 2 0 1

P₂ 0 0 1

P₃ 1 0 0

P₄ 0 0 2



系统的状态?

- 可以归还P₀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P1、P2、P3和P4。

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**”？

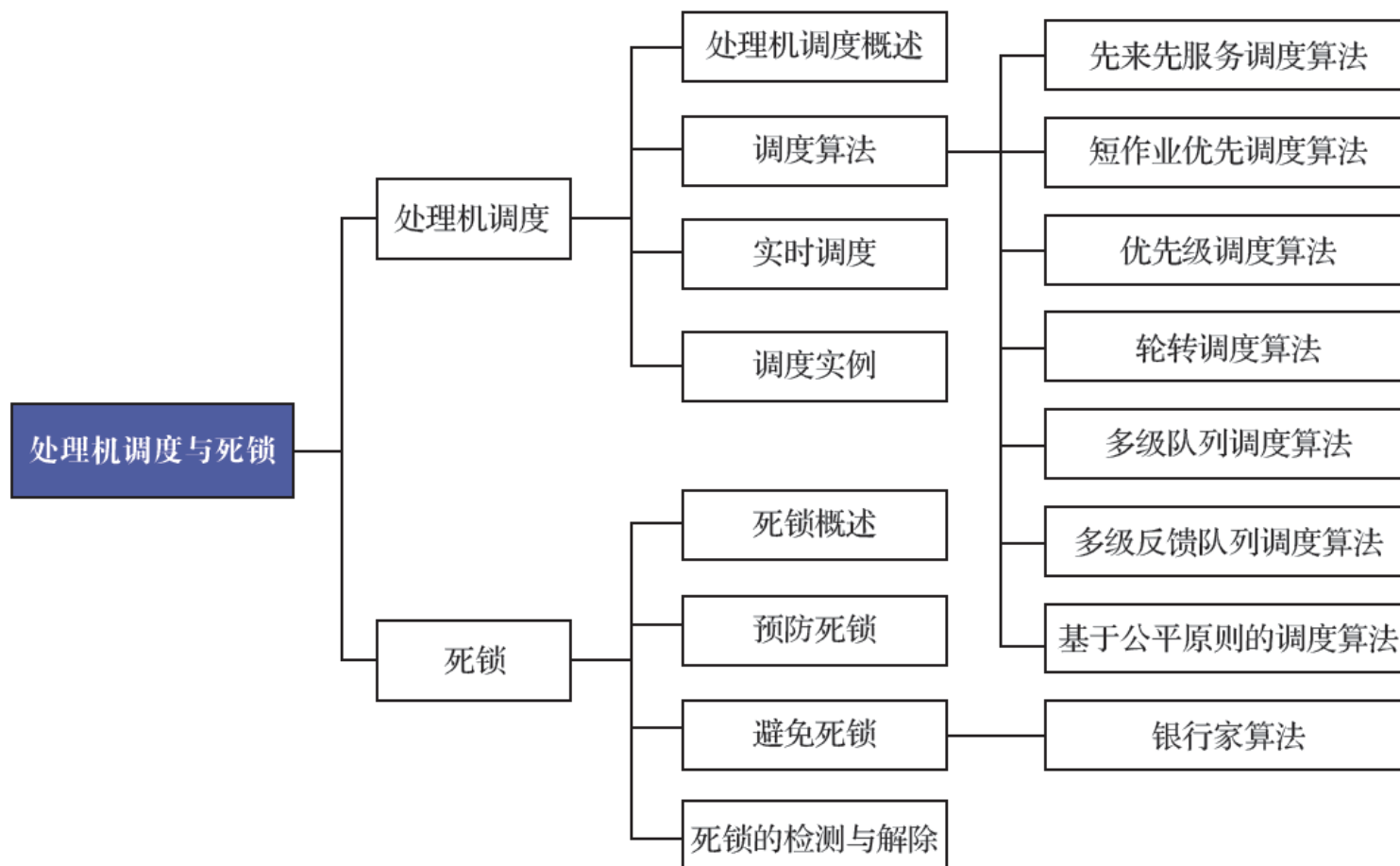
- 进程的优先级；
- 进程需要计算多长时间，以及需要多长时间结束；
- 进程使用的资源，进程完成还需要多少资源；
- 进程是交互的还是批处理的。

- **死锁预防**：避免形成死锁出现的条件
- **死锁避免**：在分配资源之前进行检验（**预分配 or 多重宇宙中的“以身试药”**），确定不会出现死锁再分配资源
- **死锁检测**：不再预防或者避免死锁，而是允许进入死锁状态，定时调用死锁检测算法，发现死锁则采用死锁恢复机制



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
第9章	磁盘存储器管理
第10章	多处理机操作系统
第11章	虚拟化和云计算
第12章	保护和安全



本章学习结束



第六次作业

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

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！