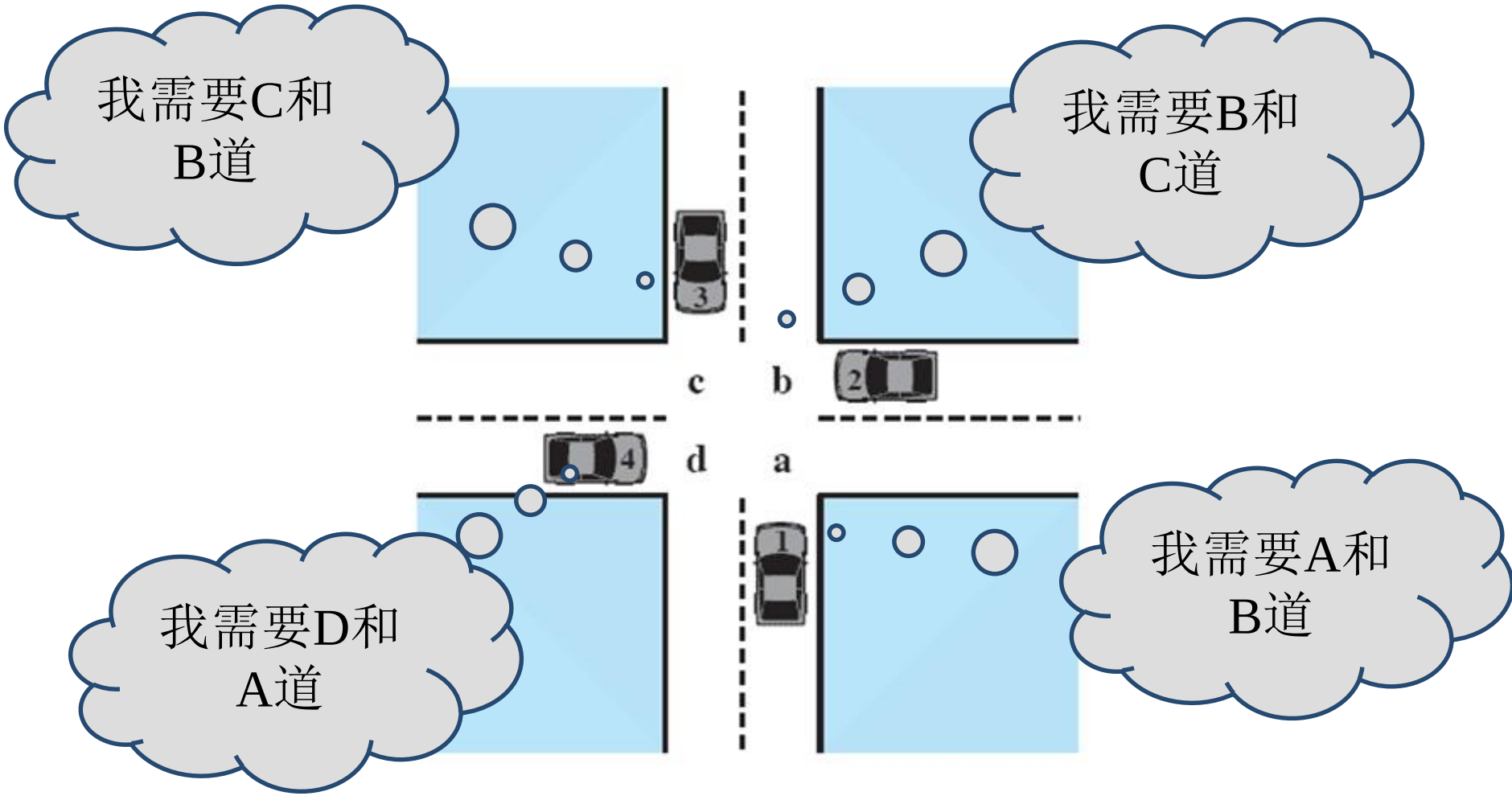
Chapter 3

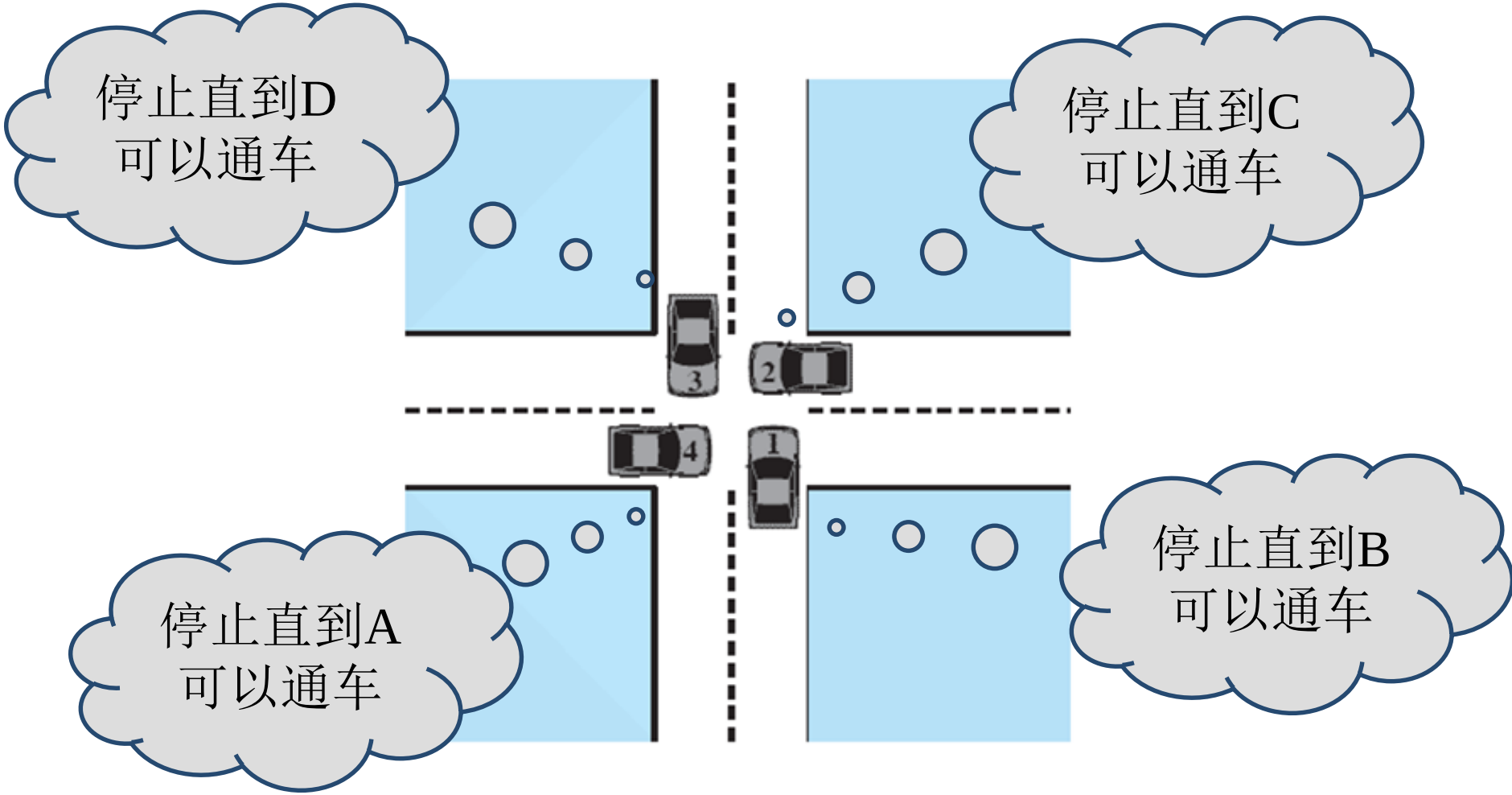
Deadlocks

3.1. 资源
3.2. 死锁概述
3.3. 驼鸟算法
3.4. 死锁检测和死锁恢复
3.5. 死锁避免
3.6. 死锁预防
3.7. 其他问题

死锁的概念：可能死锁



发生死锁



资源

- 一些独占性资源
  - 打印机
  - 磁带
  - 系统内部表中的表项
- 进程需要一个合理的顺序去访问资源
- 假设一个进程拥有资源 A 并请求资源 B
  - 同时另一个进程拥有B 并请求A
  - 两个进程都被阻塞,并且一直处于这样的状态

资源(1)

- 死锁有可能出现,当…
  - 进程对设备、文件等取得了排他性访问权时
  - 我们把这类需要排他性使用的对象称为资源 resources
- 可抢占资源
  - 可以从拥有它的进程中抢占而不会产生任何副作用
- 不可抢占资源
  - 指在不引相关的计算失败的情况下，无法把它从占有它的进程处抢占过来

资源(2)

- 使有一人资源所需要的事件顺序可以用抽象的形式表示如下：
  1. 请求资源
  2. 使用资源
  3. 释放资源

- 若请求资源不可用，则请求进程被迫等待
  - 请求进程可能被阻塞
  - 资源请求返回一个错误代码

死锁的概述

- 形式化定义:
  如果一个进程集合中的每个进程都在等待只能由该进程集合中的其他进程才能引发的事件，那么该进程集合就是死锁
- 在大多数情况下，每个进程所等待的事件是释放该进程集合中其他进程所占有的资源
- 没有一个进程可以…
  - 运行
  - 释放资源
  - 被唤醒
- **产生死锁的原因**
  - **竞争资源。**
  - **进程间推进顺序非法。**

死锁定义：others

- 死锁定义——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状态时，若无外力作用，它们都将无法再向前推进。
- 死锁定义——一组进程处于死锁状态是指：如果在一个进程集合中的每一个进程都在等待只能由该集合中的其它一个进程才能引发的事件，则称一组进程或系统发生了死锁。——孙钟秀主编《操作系统教程》
- 死锁定义：一组竞争系统资源或相互通信的进程间相互的“永久”阻塞。——[美]William Stallings著《操作系统: 内核与设计原理(第四版）》

死锁定义(con.)

- 假设1：任意一个进程要求资源的最大数量不超过系统能提供的最大量
- 假设2：如果一个进程在执行中所提出的资源要求能够得到满足，那么，它一定能在有限的时间内结束
- 假设3：一个资源在任何时间最多只有一个进程所占有
- 假定4：一个进程一次申请一个资源，且只在申请资源得不到的满足时才处于等待状态。（即不考虑人工干预，等 待外设的情况）
- 假定5：一个进程结束时释放它占有的全部资源
- 假定6：系统具有有限个进程和有限个资源

- 定义：如果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。

死锁的四个条件

1. 互斥条件
   - 每个资源要么已经分配给了一个进程，要么就是可用的
2. 占有和等待条件
   - 已经得到了某个资源的进程可以再请求新的资源
3. 不可抢占条件
   - 已经分配给一个进程的资源不能强制性地被抢占，它只能被占有它的进程显式地释放
4. 环路等待条件
   - 一定有由两个或两个以上的进程组成的一条环路
   - 该环路中的每个进程都在等待着下一个进程所占有的资源

Deadlock Modeling

Strategies for dealing with Deadlocks

1. just ignore the problem altogether
2. detection and recovery
3. dynamic avoidance
   - careful resource allocation
4. prevention
   - negating one of the four necessary conditions

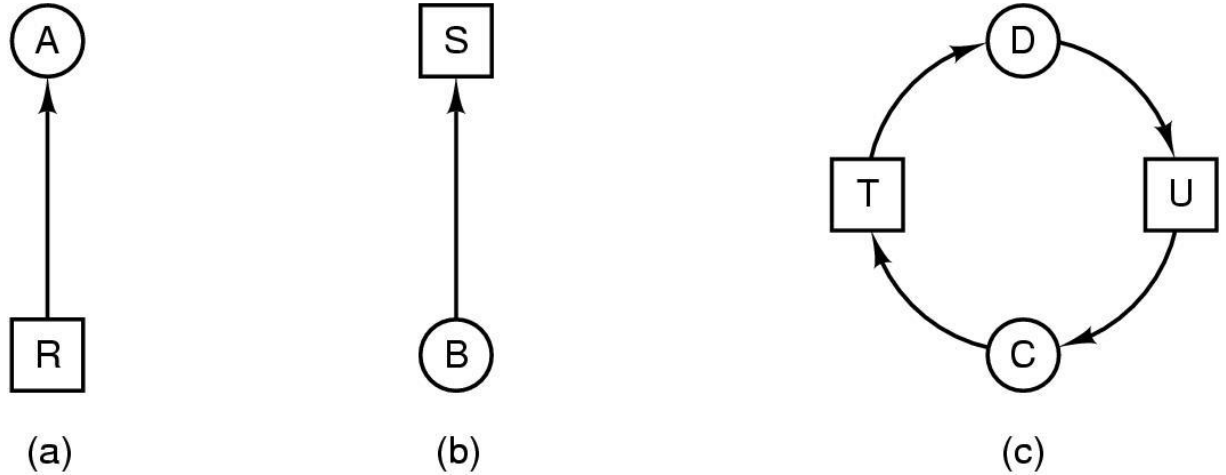
处理死锁的基本方法

方法	资源分配策略	各种可能模式	主要优点	主要缺点
预防 Prevention	保守的；宁可资源闲置（从机制上使死锁条件不成立）	一次请求所有资源<条件 1>	适用于作突发式处理的进程；不必剥夺	效率低；进程初始化时间延长
		资源剥夺<条件 3>	适用于状态可以保存和恢复的资源	剥夺次数过多；多次对资源重新起动
		资源按序申请<条件 4>	可以在编译时（而不必在运行时）就进行检查	不便灵活申请新资源
避免 Avoidance	是“预防”和“检测”的折衷（在运行时判断是否可能死锁）	寻找可能的安全的运行顺序	不必进行剥夺	使用条件：必须知道将来的资源需求；进程可能会长时间阻塞
检测 Detection	宽松的；只要允许，就分配资源	定期检查死锁是否已经发生	不延长进程初始化时间；允许对死锁进行现场处理	通过剥夺解除死锁，造成损失

Deadlock Modeling (con.)

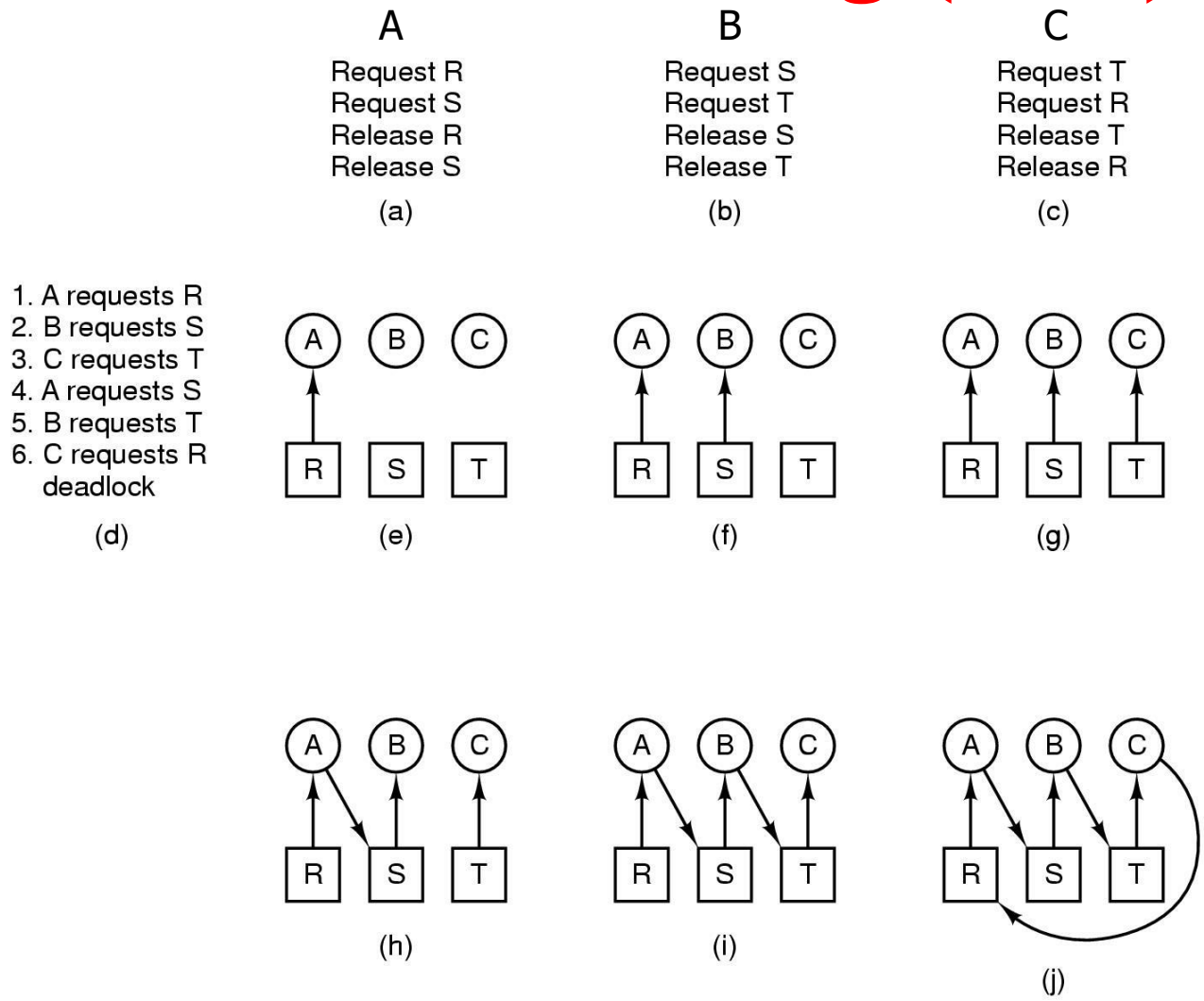
- Modeled with directed graphs

  资源分配图(resource allocation graph)



  - resource R assigned to process A
  - process B is requesting/waiting for resource S
  - process C and D are in deadlock over resources T and U

Deadlock Modeling (con.)



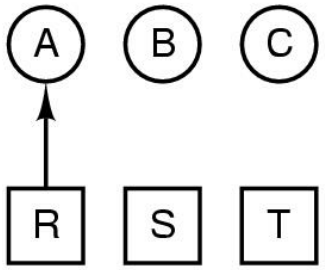
死锁是如何发生的

Deadlock Modeling (con.)

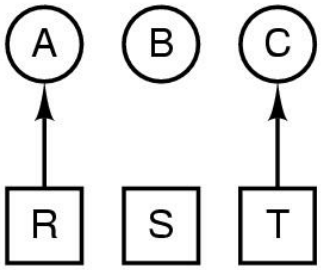
1. A requests R
2. C requests T
3. A requests S
4. C requests R
5. A releases R
6. A releases S

no deadlock

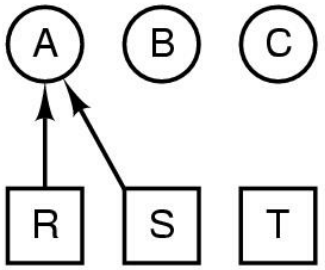
(k)



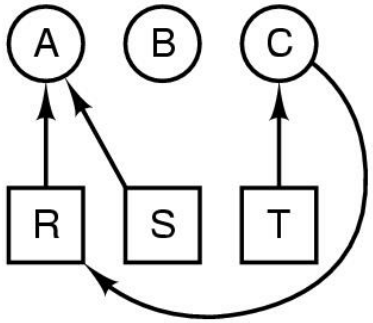
(l)



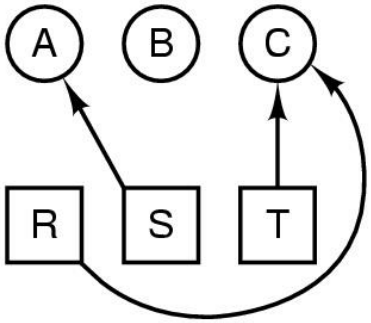
(m)



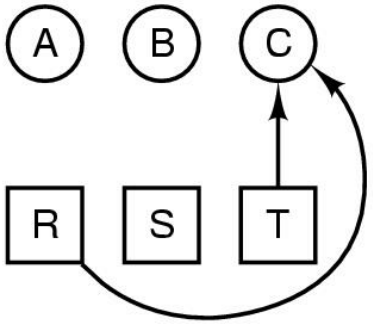
(n)



(o)



(p)



(q)

死锁是如何避免的

驼鸟算法

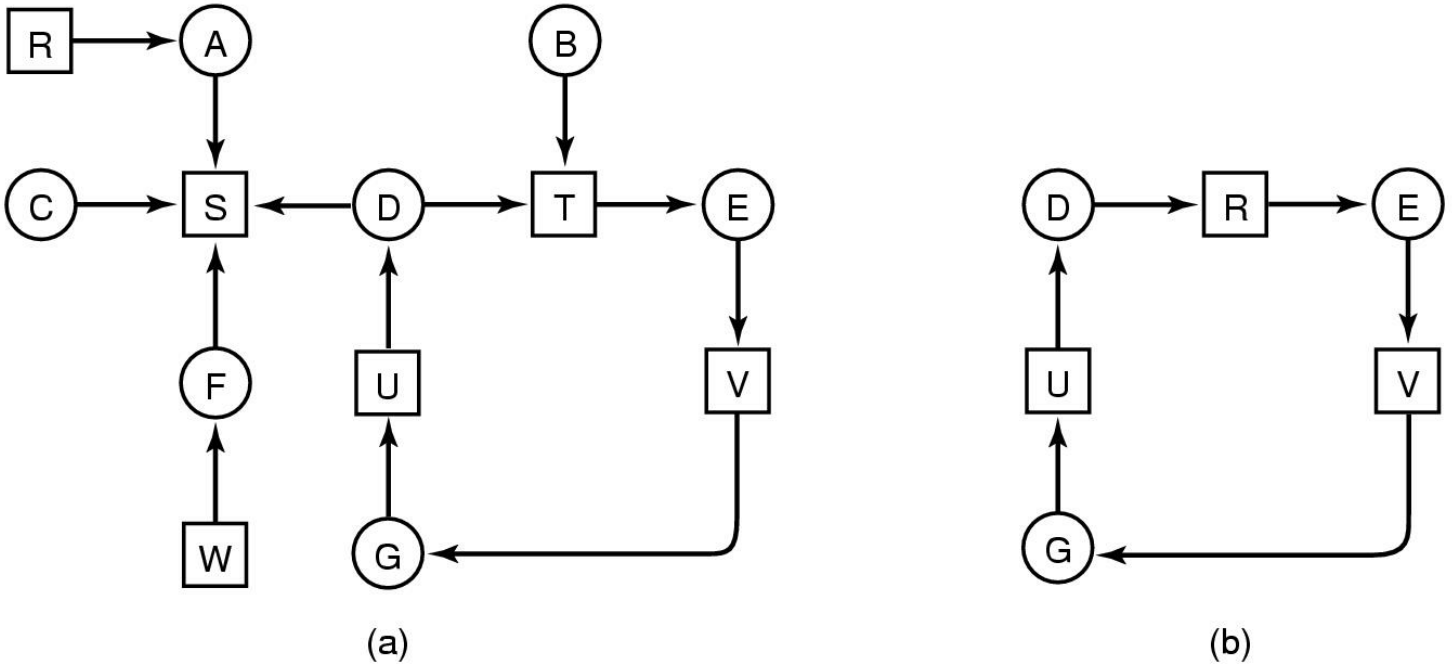
- 假装根本没有问题发生
- 可以接受，如果
  - 如果死锁发生频率低
  - 防止死锁的成本太高了
- UNIX 和Windows 采用此算法
- 它是下面两者的折衷
  - 方便性
  - 正确性

死锁发生的概率很小；
处理死锁开销太大；
而且目前没有较完善的处理死锁的方法。

死锁的检测和解除

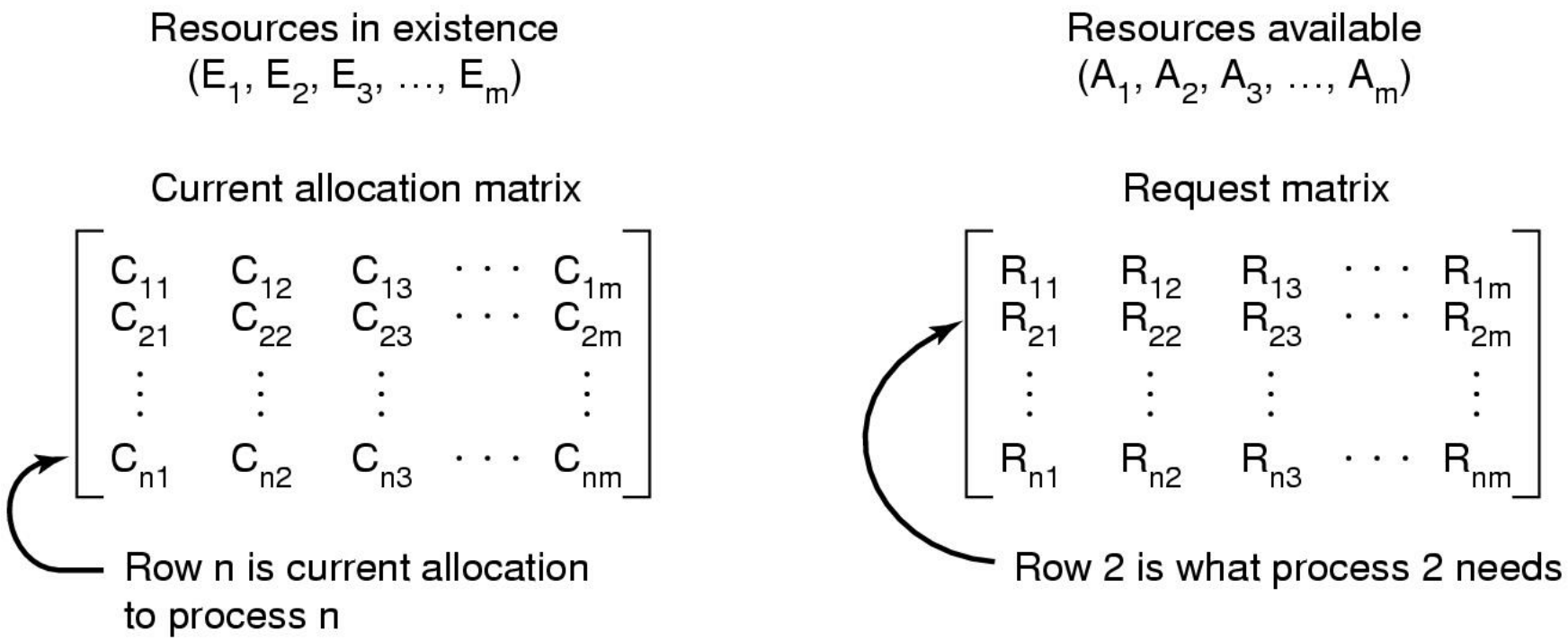
- 系统为进程分配资源时，若未采取避免和预防死锁的措施，则采用检测和解除死锁的手段。即：
  - 保存资源的请求和分配信息；
  - 利用某种算法对这些信息加以检查，以判断是否存在死锁。

每种类型一个资源的死锁检测



- 注意资源的所属和请求关系
- 如果图中包含一个或一个以上的环，那么死锁就存在

每种类型多个资源的死锁检测



死锁检测算法的数据结构

每种类型多个资源的死锁检测(con.)

Tape drives Plotters Scanners CD Roms

E = (4 2 3 1)

Tape drives Plotters Scanners CD Roms

A = (2 1 0 0)

Current allocation matrix

$$C = \begin{bmatrix} 0 & 0 & 1 & 0 \\ 2 & 0 & 0 & 1 \\ 0 & 1 & 2 & 0 \end{bmatrix}$$

Request matrix

$$R = \begin{bmatrix} 2 & 0 & 0 & 1 \\ 1 & 0 & 1 & 0 \\ 2 & 1 & 0 & 0 \end{bmatrix}$$

死锁检测算法的例子

资源分配图的化简

- 一个进程的所有资源要求均能被满足，则该进程得到其所需全部资源从而不断取得进展，直至完成全部任务并释放出全部资源。
- 在有向图中找到既非阻塞又非孤立的结点，分配它所需的全部资源后，当它继续执行并完成然后释放其全部资源而使之成为孤立结点。
- 把孤立结点释放的资源分配给阻塞进程，使之能继续运行，并且在有限时间后完成，再释放其全部资源而成为新的鼓励结点。
- 经过一系列上述步骤，若能消去图中所有的边，使所有进程都成为孤立结点，则图可完全化简。否则为不可完全化简。

死锁状态 vs安全状态

- 死锁定理：当且仅当系统某状态S所对应的资源分配图是不可化简的，则S是死锁状态。而不可被化简的进程即是被死锁的进程。反之，若状态S所对应的资源分配图是可化简的，则S是安全状态。
- 资源分配图的化简结果与化简顺序无关，最终结果是相同的

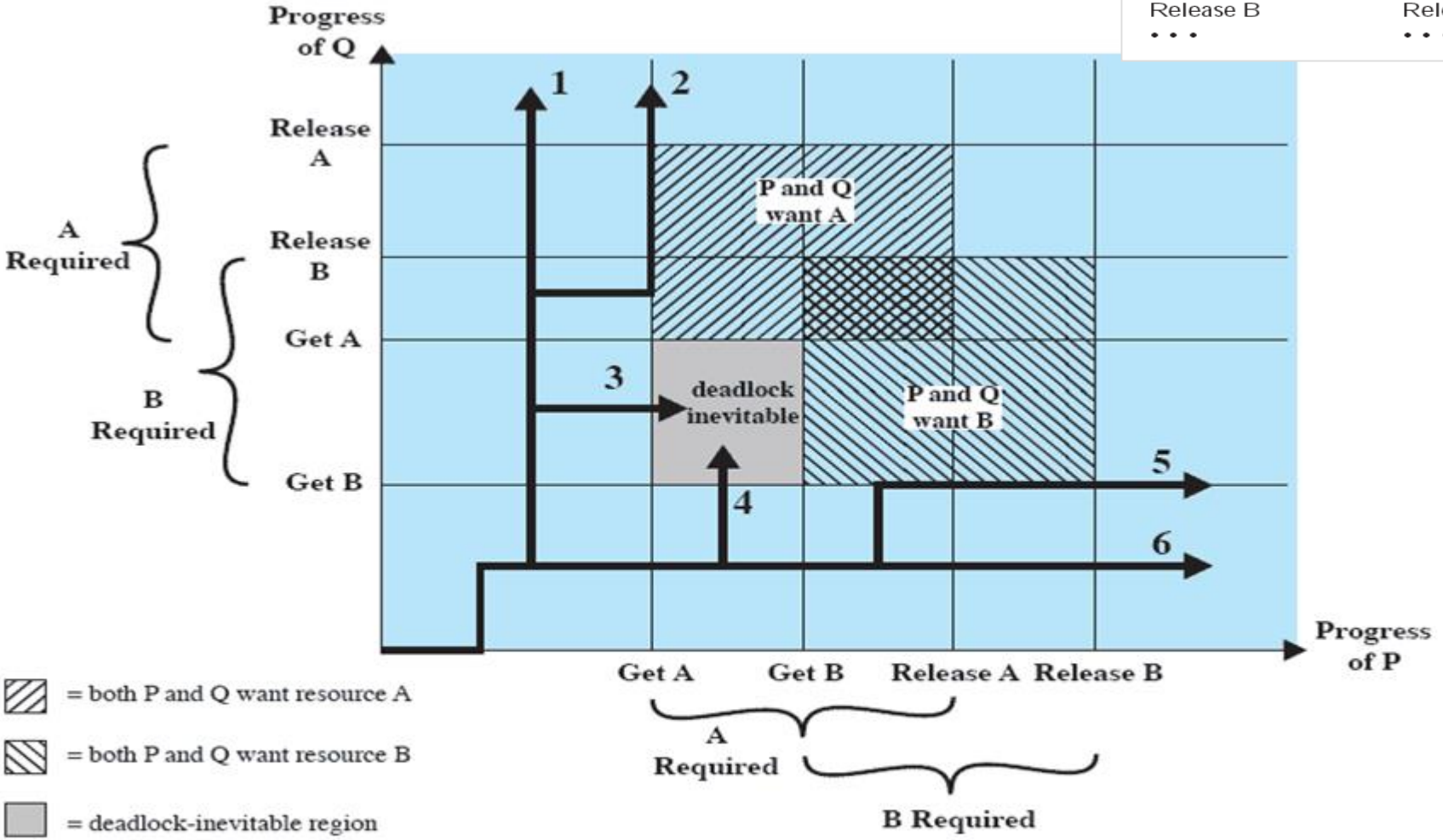
从死锁中恢复

- 利用抢占恢复
  - 将某一资源从一个进程强行取走给另一个进程使用
  - 取决于该资源本身的特性。
- 利用回滚恢复
  - 周期性对进程进程进行检查点检查
  - 使用检查点保存资源状态
  - 如果发现死锁回滚进程
- 通过杀死进程恢复
  - 最直接也是最简单的解决死锁的方法
  - 杀掉环中的一个进程
  - 环外的进程释放其资源
  - 杀死可以从头开始重新运行而且不会带来副作用的进程

Deadlock Avoidance

资源轨迹图

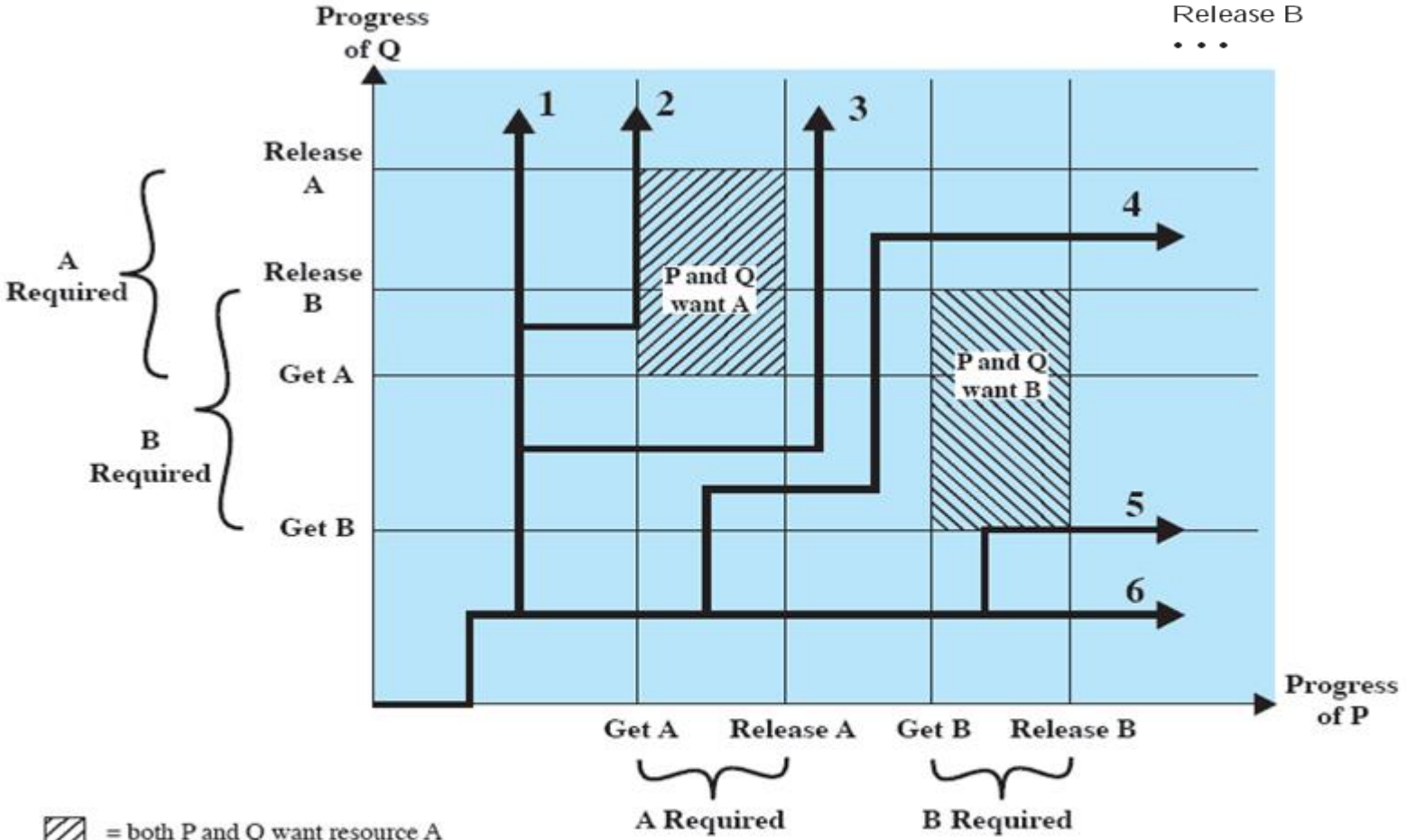
Process P	Process Q
• • •	• • •
Get A	Get B
• • •	• • •
Get B	Get A
• • •	• • •
Release A	Release B
• • •	• • •
Release B	Release A
• • •	• • •



Example of Deadlock

替代方案

Process P	Process Q
• • •	• • •
Get A	Get B
• • •	• • •
Release A	Get A
• • •	• • •
Get B	Release B
• • •	• • •
Release B	Release A
• • •	• • •



Example of No Deadlock

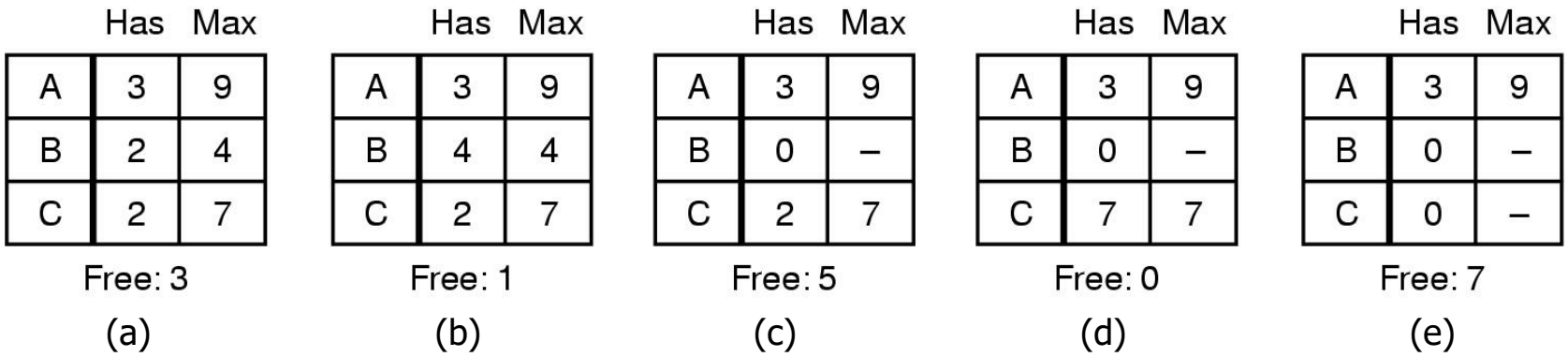
系统安全状态

在避免死锁的方法中，允许进程动态地申请资源，但系统在进行资源分配之前，应先计算此次资源分配的安全性。若此次分配不会导致系统进入不安全状态，则将资源分配给进程； 否则，令进程等待。

所谓安全状态，是指系统能按某种进程顺序(P_1，P_2，…，P_n)(称〈P_1, P_2, ..., P_n〉序列为安全序列)，来为每个进程P_i分配其所需资源，直至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。如果系统无法找到这样一个安全序列，则称系统处于不安全状态。

- **并非所有不安全状态都是死锁状态，但它迟早会进入死锁状态。**
- **只要系统处于安全状态，便可避免进入死锁状态。**

Safe and Unsafe States (1)



Demonstration that the state in (a) is safe

Safe and Unsafe States (2)

	Has	Max
A	3	9
B	2	4
C	2	7

Free: 3

(a)

	Has	Max
A	4	9
B	2	4
C	2	7

Free: 2

(b)

	Has	Max
A	4	9
B	4	4
C	2	7

Free: 0

(c)

	Has	Max
A	4	9
B	—	—
C	2	7

Free: 4

(d)

Demonstration that the sate in b is not safe

银行家算法

- 最具代表性的避免死锁的算法，是Dijkstra的银行家算法。

银行家算法：

在资源动态分配过程中，若分配后系统状态仍是安全的，则同意分配，否则将拒绝分配，这样可防止系统进入不安全状态，从而避免死锁。

The Banker's Algorithm for a Single Resource

	Has	Max
A	0	6
B	0	5
C	0	4
D	0	7

Free: 10

(a)

	Has	Max
A	1	6
B	1	5
C	2	4
D	4	7

Free: 2

(b)

	Has	Max
A	1	6
B	2	5
C	2	4
D	4	7

Free: 1

(c)

- Three resource allocation states
  - safe
  - safe
  - unsafe

银行家算法的特点

- 允许互斥、部分分配和不可抢占，可提高资源利用率；
- 要求事先说明最大资源要求，在现实中很困难；

利用银行家算法避免死锁

- 银行家算法中的数据结构
  - (1) 可利用资源向量Available: Available［j］=K 表示系统中现有R_j类资源K个。
  - 最大需求矩阵Max: Max[i, j]=K 表示进程i需要R_j类资源的最大数目为K。
  - 分配矩阵Allocation: Allocation[i, j]=K 表示进程i当前已分得R_j类资源的数目为K。
  - 需求矩阵Need: Need[i, j]=K 表示进程i还需要R_j类资源K个，方能完成其任务。

Need[i, j] = Max[i, j] – Allocation[i, j]

银行家算法步骤

- 设$Request_i$是进程P_i的请求向量，如果$Request_i[j]=K$，表示进程P_i需要K个R_j类型的资源。当P_i发出资源请求后，系统按下述步骤进行检查：
  - (1) 如果$Request_i[j] \le Need[i,j]$，便转向步骤(2)；否则认为出错，因为它所需要的资源数已超过它所宣布的最大值。
  - (2) 如果$Request_i[j] \le Available[j]$，便转向步骤(3)；否则， 表示尚无足够资源，$P_i$须等待。
  - (3) 系统试探着把资源分配给进程Pi，并修改下面数据结构中的数值：

    Available[j] = Available[j] - Requesti[j];

    Allocation[i, j] = Allocation[i, j] + Requesti[j];

    Need[i, j] = Need[i, j] – Requesti[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，才正式将资源分配给进程Pi，以完成本次分配；否则， 将本次的试探分配作废，恢复原来的资源分配状态，让进程Pi等待(阻塞)。

安全性算法

- 判断安全性的算法可描述如下：
  - (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，它含有*m*个元素，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; 当有足够资源分配给进程时， 再令Finish［i］：=true。
  - (2) 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］≤Work［j］； 若找到， 执行步骤(3)， 否则，执行步骤(4)。
  - (3) 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[i] + Allocation[i, j];

    Finish[i] = true;

    go to step 2;
  - (4) 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。

Banker's Algorithm for Multiple Resources

Process	Tape drives	Plotters	Scanners	CD ROMs
A	3	0	1	1
B	0	1	0	0
C	1	1	1	0
D	1	1	0	1
E	0	0	0	0

Resources assigned

Process	Tape drives	Plotters	Scanners	CD ROMs
A	1	1	0	0
B	0	1	1	2
C	3	1	0	0
D	0	0	1	0
E	2	1	1	0

Resources still needed

E = (6342)
P = (5322)
A = (1020)

Example of banker's algorithm with multiple resources

银行家算法实例：设系统中有5个进程{P_0，P_1，P_2，P_3，P_4}和3类资源{A，B，C}，各类资源总数分别为10、5、7，在T_0时刻的资源分配情况如下表所示：

进程 \ 资源情况	Max A	Max B	Max C	Allocation A	Allocation B	Allocation C	Need A	Need B	Need C	Available A	Available B	Available 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	(2	3	0)
				(3	0	2)	(0	2	0)			
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			

问：

（1）T_0时刻系统是否安全，为什么？

（2）P_1发出请求向量$Request_1$（1，0，2），分析系统是否可同意请求。

（3）P_4发出请求向量$Request_4$（3，3，0），分析系统是否可同意请求。

（4）P_0发出请求向量$Request_4$（0，2，0），分析系统是否可同意请求。

解答：（1）T_0时刻系统是否安全，为什么？

利用安全性算法对T_0时刻的资源分配情况进行分析(见下表)可知，在T_0时刻存在着一个安全序列{P_1,P_3,P_4,P_2,P_0}，故系统是安全的。

进程 \ 资源情况	Work A	Work B	Work C	Need A	Need B	Need C	Allocation A	Allocation B	Allocation C	Work+Available A	Work+Available B	Work+Available 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	2	0	1	0	10	5	7	true

（2）P_1发出请求向量$Request_1$（1，0，2），按银行家算法，分析系统是否可同意请求。

进程 \ 资源情况	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	2	3	0
P_1	3	2	2	3	0	2	0	2	0			
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			

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

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

③系统先假定可为P1分配资源，并修改Available, Allocation1和Need1向量，由此形成资源变化情况如表所示。

④再利用安全性算法检查此时系统是否安全。如下表所示。

进程 \ 资源情况	Work			Need			Allocation			Work+Available			Finish
	A	B	C	A	B	C	A	B	C	A	B	C	
P_1	3	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	2	0	1	0	10	5	7	true

即存在安全序列{P_1,P_3,P_4,P_2,P_0}，故系统是安全的，可以立即将P_1所申请的资源分配给它。

实际上，(1)中的安全序列中的第一个进程就是P_1，当然对P_1的请求可以满足。

（3）P_4发出请求向量$Request_4(3,3,0)$，按银行家算法，分析系统是否可同意请求。

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

②$Request_4(3,3,0) \leq Available(2,3,0)$，让$P_4$等待。

（4）P_0发出请求向量$Request_4$（0，2，0），按银行家算法，分析系统是否可同意请求。

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

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

③系统暂时先假定可为P_0分配资源，并修改有关数据，如图所示。

资源情况 / 进程	Max A B C	Allocation A B C	Need A B C	Available A B C
P_0	7 5 3	0 3 0	7 2 3	2 1 0
P_1	3 2 2	3 0 2	0 2 0	
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	

④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，系统进入不安全状态，故系统不能同意P0的请求，让其阻塞。

Deadlock Prevention(死锁预防)

破坏互斥条件

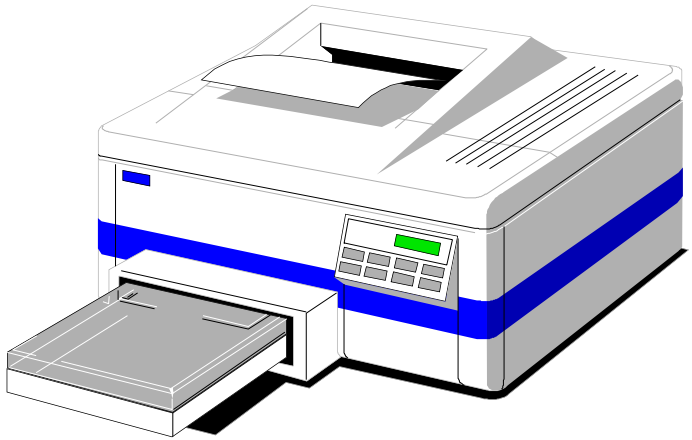
- 一些设备 (例如打印机) 假脱机
  - 惟一真正请求使用物理打印机的进程是打印机的守护进程
  - 这样消除因打印机产生死锁
- 并不是所有的设备都可以使用假脱机技术
- 原理:
  - 避免分配那些不是绝对必需的资源
  - 尽量做到尽可能少的进程可以真正请求资源

破坏占有和等待条件

- 所有进程在开始执行前请求所需的全部资源
  - 一个进程不会因其他资源而等待
- 问题
  - 很多进程直到运行时才知道它需要多少资源
  - 资源利用率不是最优的
- 变种:
  - 进程请求先暂时释放其当前占用的所有资源
  - 然后再尝试一次获得所需的全部资源

破坏不可抢占条件

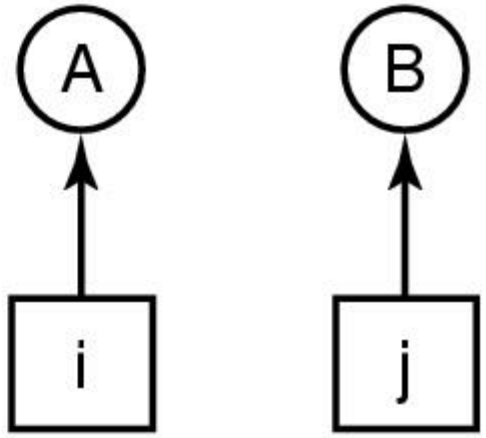
- 破坏第三个条件也是可能的
- 假若一个进程已分配到一台打印机
  - 正在打印输出
  - 现在强制地把它占有的打印机抢占
  - !!??



破坏环路等待条件

1. Imagesetter
2. Scanner
3. Plotter
4. Tape drive
5. CD Rom drive

(a)



(b)

- 对资源排序编号
- 资源分配图

死锁预防方法汇总

Condition	Approach
Mutual exclusion	Spool everything
Hold and wait	Request all resources initially
No preemption	Take resources away
Circular wait	Order resources numerically

Summary of approaches to deadlock prevention

其他问题
两阶段加锁

- 第一阶段
  - 进程试图对所有所需的记录进行加锁，一次锁一个记录
  - 如果第一阶段加锁成功, 就开始第二阶段
  - (在第一阶段并没有做实际的工作)
- 如果第一阶段成功，那以开始第二阶段,
  - 执行更新
  - 释放锁
- 注意这类似于同时请求所有资源
- 算法只有当程序员仔细安排了程序
  - 使得第一阶段程序在任意一点停下来，并重新开始而不会产生错误

Nonresource Deadlocks
通信死锁

- 两个进程通信可能会导致死锁
  - 每个进程因为等待另外一个进程引发的事件而产生阻塞
- 通过超时技术解决
- 通过信号量解决
  - 进程通过含有两个信号量的函数 (mutex 和另一个信号量)
  - 如果以错误的顺序工作，那么会导致死锁

活锁（livelock）

- 活锁指的是任务或者执行者没有被阻塞，由于某些条件没有满足，导致一直重复尝试，失败，尝试，失败。处于活锁的实体是在不断的改变状态，而处于死锁的实体表现为等待；活锁有一定几率解开。而死锁（deadlock）是无法解开的。
- 因为没有出现死锁现象，进程没有阻塞，但是进程也无法推进
- 解决方法：让程序等待一段随机长的时间，然后再尝试运行

饥饿

- 算法分配资源
  - 可能情况：短作业优先
- 在系统中多个短作业可以工作得很好
- 可能导致长作业无限期推后
  - 尽管它没有被阻塞
- 解决方案:
  - 先来先服务资源分配策略