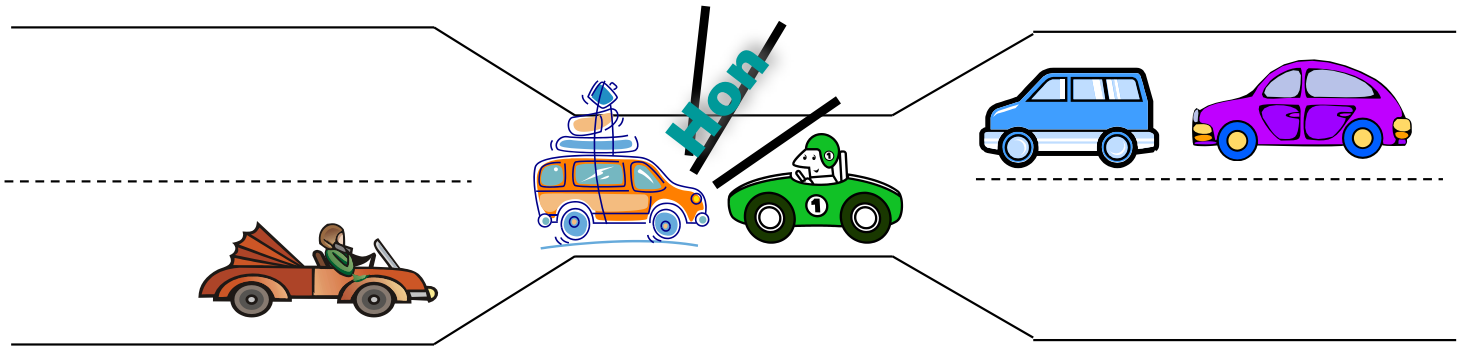
Deadlocks

- Example
  - System has 2 tape drives.
  - P_1 and P_2 each hold one tape drive and each needs another one.

- Example
  - semaphores *A* and *B*, initialized to 1

P_0	P_1
wait (A);	*wait(B)*
wait (B);	*wait(A)*

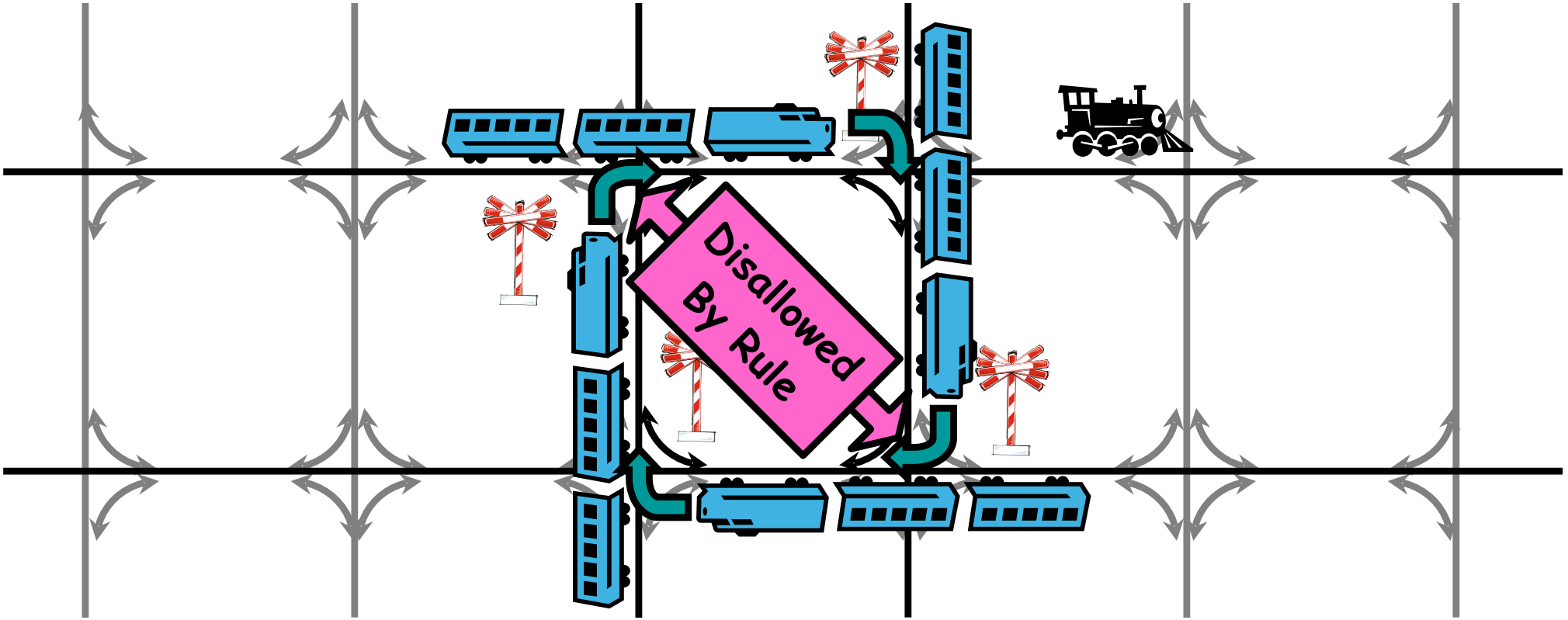
Bridge Crossing Example

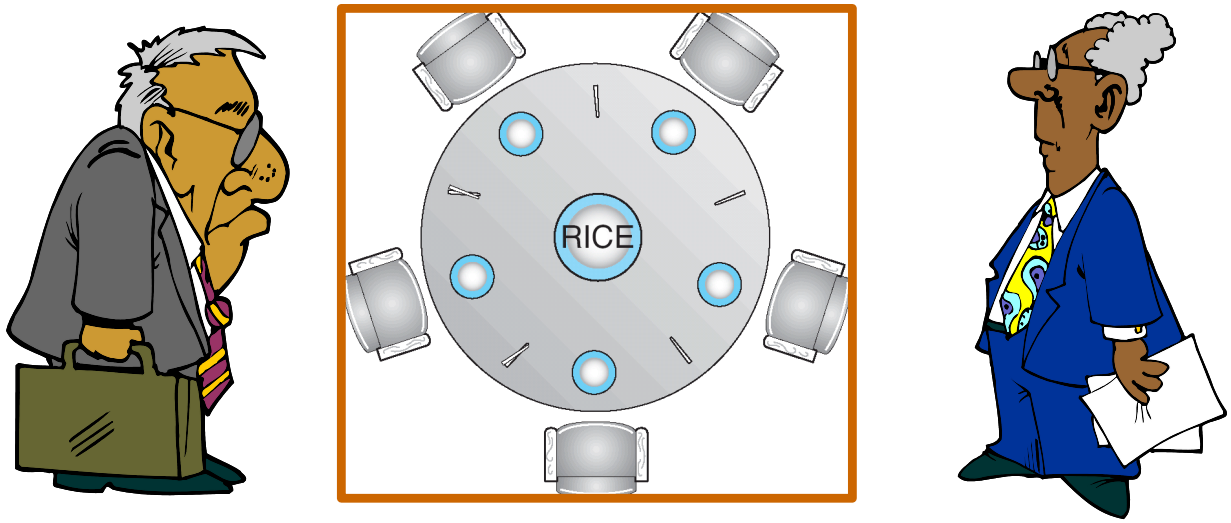


- Each segment of road can be viewed as a resource
  - Car must own the segment under them
  - Must acquire segment that they are moving into
- For bridge: must acquire both halves
  - Traffic only in one direction at a time
  - Problem occurs when two cars in opposite directions on bridge: each acquires one segment and needs next
- If a deadlock occurs, it can be resolved if one car backs up (preempt resources and rollback)
  - Several cars may have to be backed up
- Starvation is possible
  - East-going traffic really fast ⇒ no one goes west

Train Example (Wormhole-Routed Network)

- Circular dependency (Deadlock!)
  - Each train wants to turn right
  - Blocked by other trains
  - Similar problem to multiprocessor networks
- Fix? Imagine grid extends in all four directions
  - Force ordering of channels (tracks)
    - Protocol: Always go east-west first, then north-south
  - Called "dimension ordering" (X then Y)





- Five chopsticks/Five lawyers (really cheap restaurant)
  - Free-for all: Lawyer will grab any one they can
  - Need two chopsticks to eat
- What if all grab at same time?
  - Deadlock!
- How to fix deadlock?
  - Make one of them give up a chopstick (Hah!)
  - Eventually everyone will get chance to eat
- How to prevent deadlock?
  - Never let lawyer take last chopstick if no hungry lawyer has two chopsticks afterwards

死锁不仅与系统拥有的资源数量有关，而且与资源分配策略，进程对资源的使用要求以及并发进程的推进顺序有关。

- 死锁产生的四个必要条件：

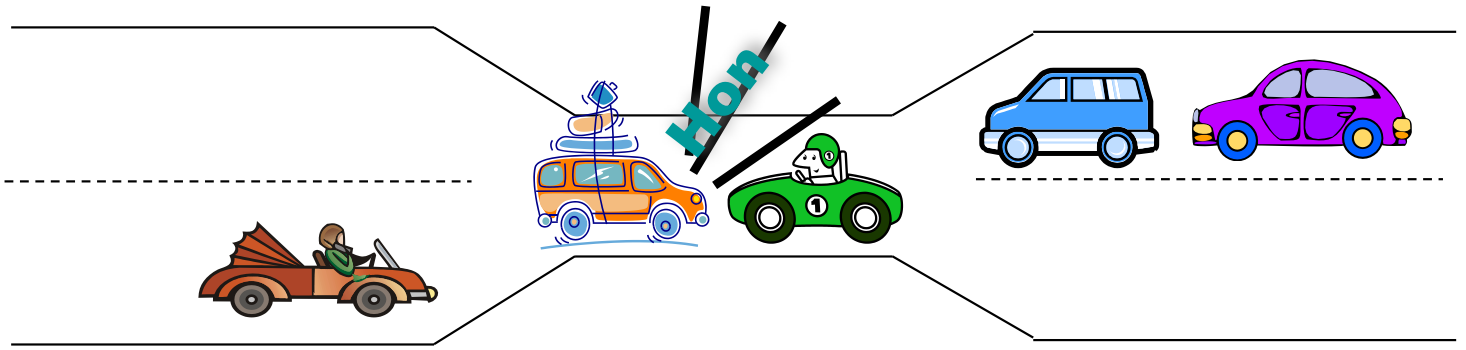
（1）互斥条件(**Mutual exclusion**)：进程互斥使用资源。

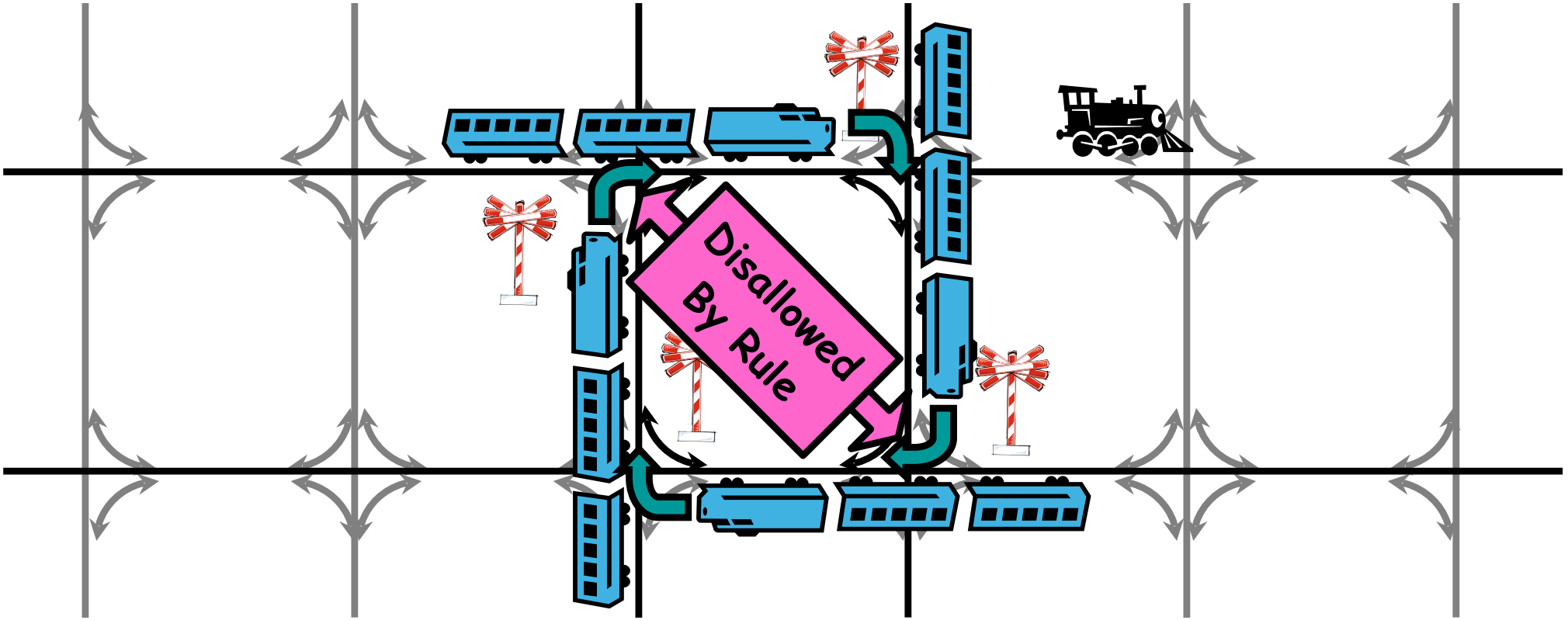
（2）部分分配条件(**Hold and wait**)：申请新资源时不释放已占有资源。

（3）不剥夺条件(**No preemption**)：一个进程不能抢夺其他进程占有的资源。

（4）环路条件(**Circular wait**)：存在一组进程循环等待资源。

示例





如何避免死锁

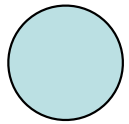
- 死锁的预防 (Deadlock Prevention)
- 死锁的避免 (Deadlock Avoidance)
- 死锁的检测与恢复 (Deadlock Detection)
  - 允许进入死锁状态
  - 检测死锁
  - 恢复策略

系统资源分配模型

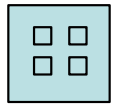
- V，节点集合，分为两类:
  - $P = \{P_1, P_2, \ldots, P_n\}$, the set consisting of all the processes in the system.
  - $R = \{R_1, R_2, \ldots, R_m\}$, the set consisting of all resource types in the system. Each resource type R_i has W_i instances
- 资源申请边： request edge – directed edge $P_1 \rightarrow R_j$
- 资源分配边： assignment edge – directed edge $R_j \rightarrow P_i$

资源分配图

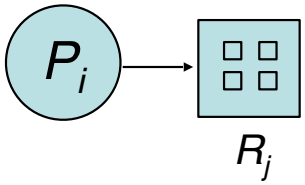
- Process



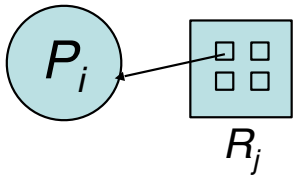
- Resource Type with 4 instances



- *Pi* requests instance of *Rj*

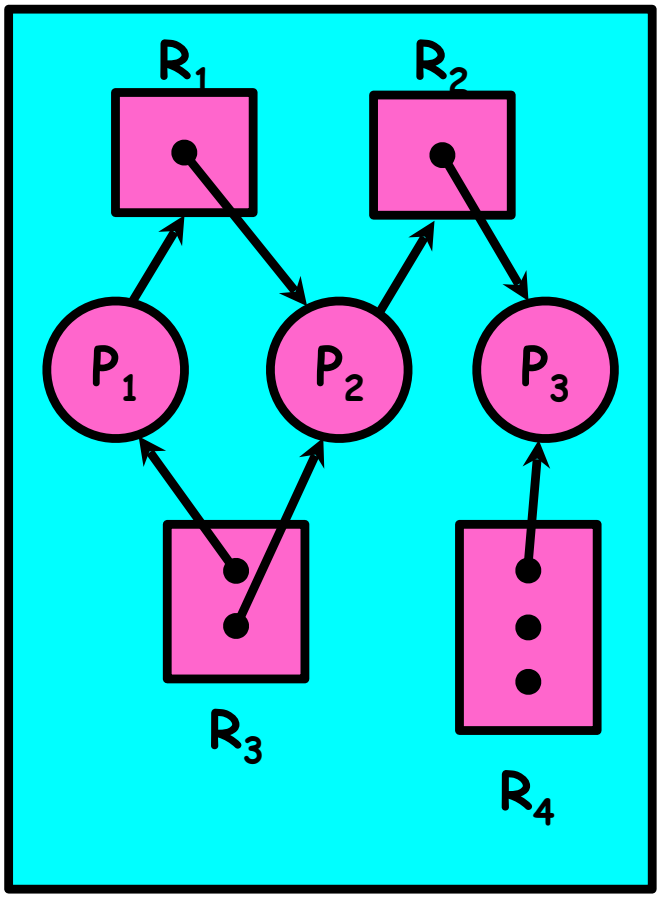


- *Pi* is holding an instance of *Rj*

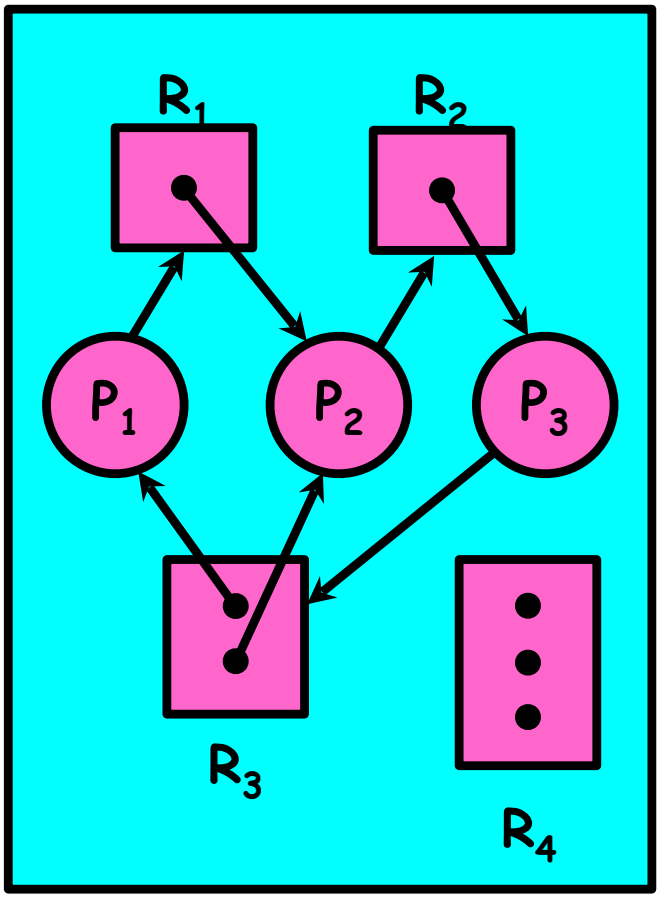


Resource Allocation Graph Examples

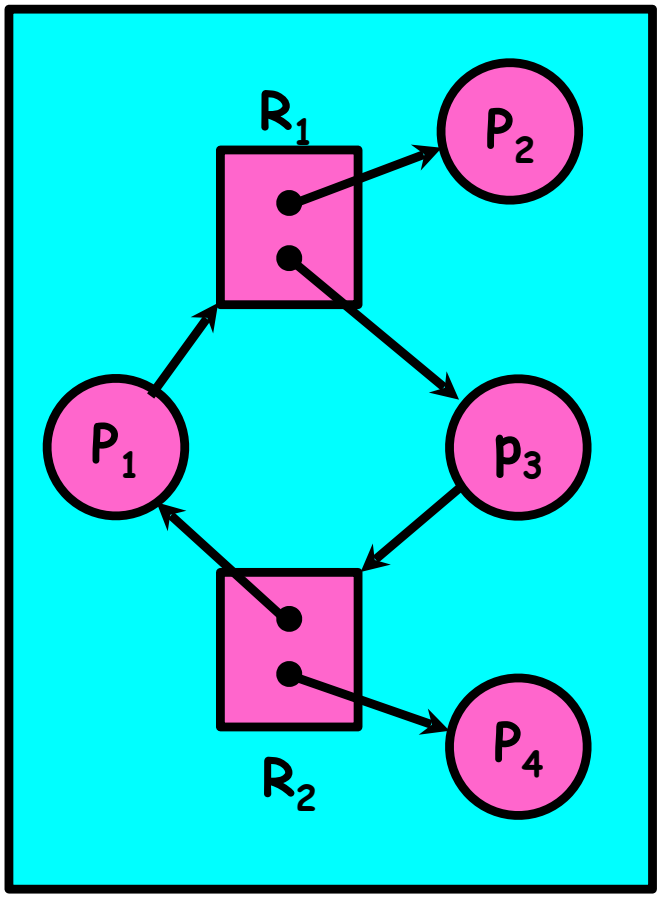
- Recall:
  - request edge – directed edge $T_1 \rightarrow R_j$
  - assignment edge – directed edge $R_j \rightarrow T_i$



Simple Resource Allocation Graph

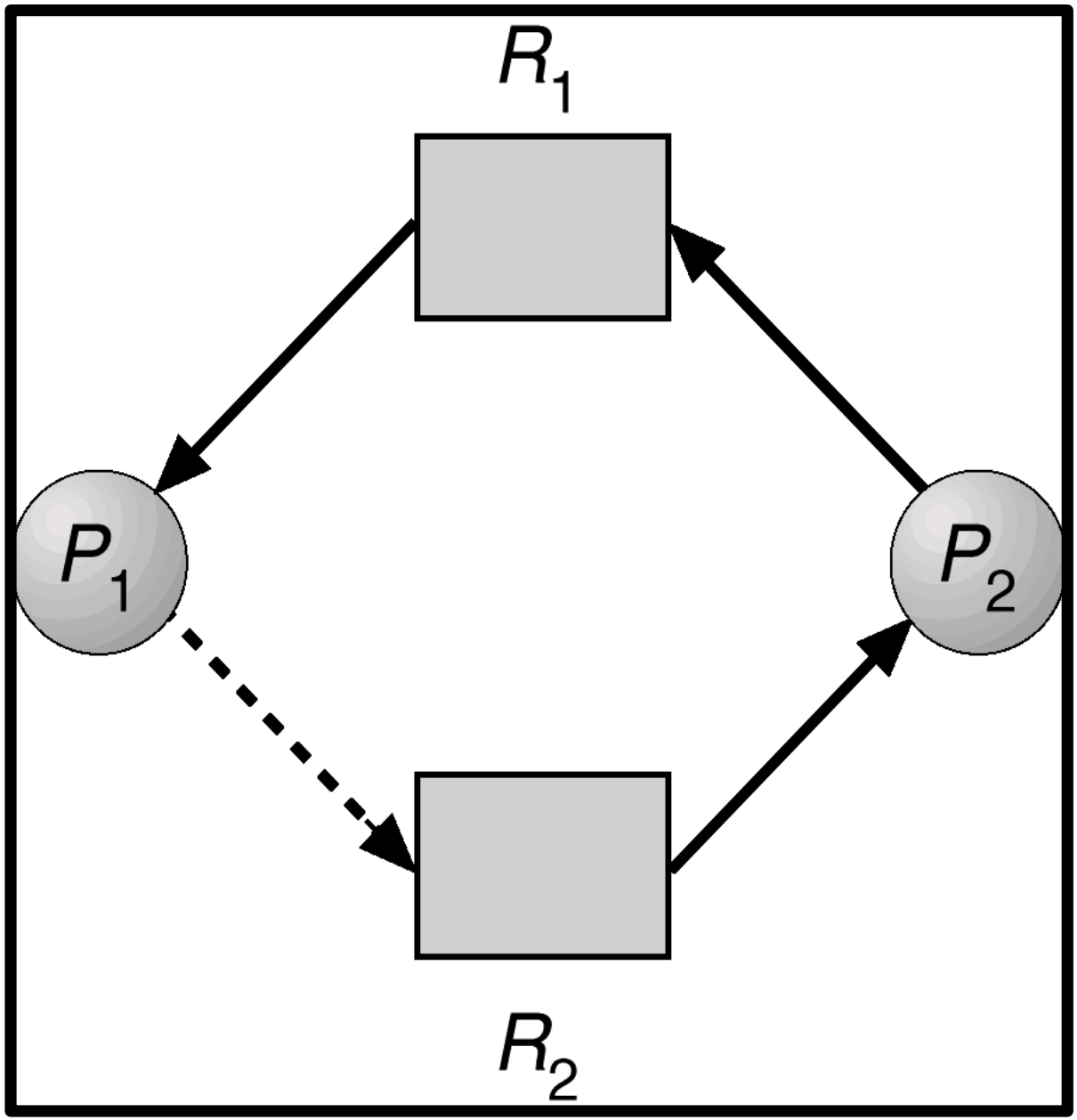


Allocation Graph



Allocation Graph

不安全的状态图



死锁判断:

- 如果图没有环，那么不会有死锁
- 如果图有环
  - 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，可能死锁

系统为死锁状态的充分条件:

当且仅当该状态的进程-资源分配图是不可完全简化的。该充分条件称为死锁定理。

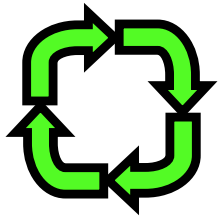
资源分配图化简:

1）找一个非孤立点进程结点且只有分配边，去掉分配边，将其变为孤立结点。

2）再把相应的资源分配给一个等待该资源的进程，即将某进程的申请边变为分配边。

如果进程-资源分配图中有环路，且涉及的资源类中有多个资源，则环路的存在只是产生死锁的必要条件而不是充分条件。

如果能在进程-资源分配图中消去此进程的所有请求边和分配边，成为孤立结点。经一系列简化，使所有进程成为孤立结点，则该图是可完全简化的；否则则称该图是不可完全简化的。



操作系统如何对待死锁

- Allow system to enter deadlock and then recover
  - Requires deadlock detection algorithm
  - Some technique for forcibly preempting resources and/or terminating tasks

- Ensure that system will *never* enter a deadlock
  - Need to monitor all lock acquisitions
  - Selectively deny those that *might* lead to deadlock

- Ignore the problem and pretend that deadlocks never occur in the system
  - Used by most operating systems, including UNIX

死锁发生后的处理:

1）立即结束所有进程的执行，并重新启动操作系统。方法简单,但以前工作全部作废,损失可能很大。

2）撤销陷于死锁的所有进程,解除死锁继续运行。

3）逐个撤销陷于死锁的进程,回收其资源,直至死锁解除。

4）剥夺陷于死锁的进程占用的资源，但并不撤销它,直至死锁解除。

5）根据系统保存的checkpoint,让所有进程回退,直到足以解除死锁。

6）当检测到死锁时，如果存在某些未卷入死锁的进程，而这些进程随着建立一些新的抑制进程能执行到结束，则它们可能释放足够的资源来解除死锁。

死锁的预防（Prevention）

- 破坏第一个条件：使资源可同时访问而不是互斥使用，这是个简单的办法，磁盘可用这种办法管理,但有许多资源往往是不能同时访问，所以这种做法许多场合行不通。

- 破坏第三个条件：采用剥夺式调度方法可破坏第三个条件，但只适用于对主存资源和处理器资源的分配，当进程在申请资源未获准许的情况下,如果主动释放资源(一种剥夺式),然后才去等待，以后再一起向系统提出申请,也能防止死锁。

- 破坏第二个条件或第四个条件：种种死锁防止办法施加于资源的限制条件太严格,会造成资源利用率和吞吐率低。两种比较实用的死锁防止方法，它们能破坏第二个条件或第四个条件：
  - 执行前一次申请全部资源
  - 没有占有资源时才能分配资源

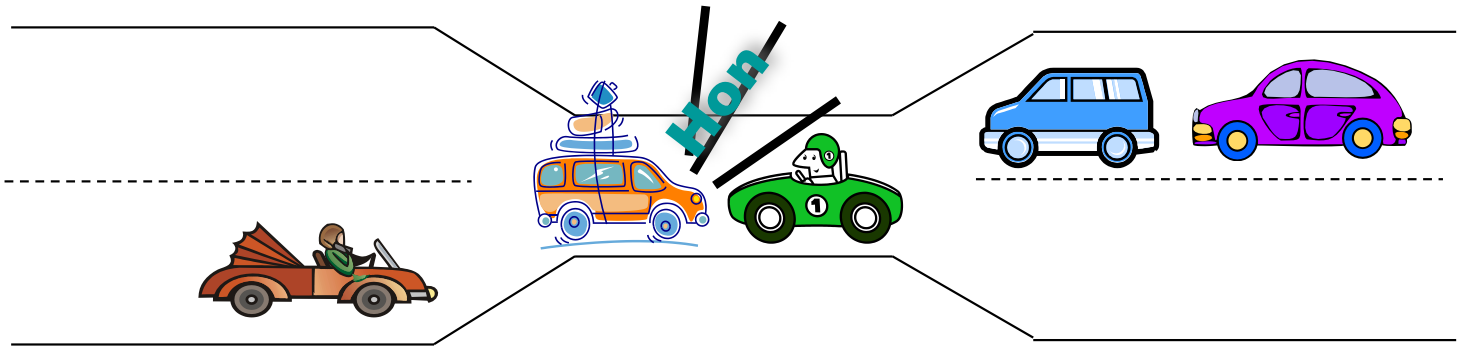
预防措施

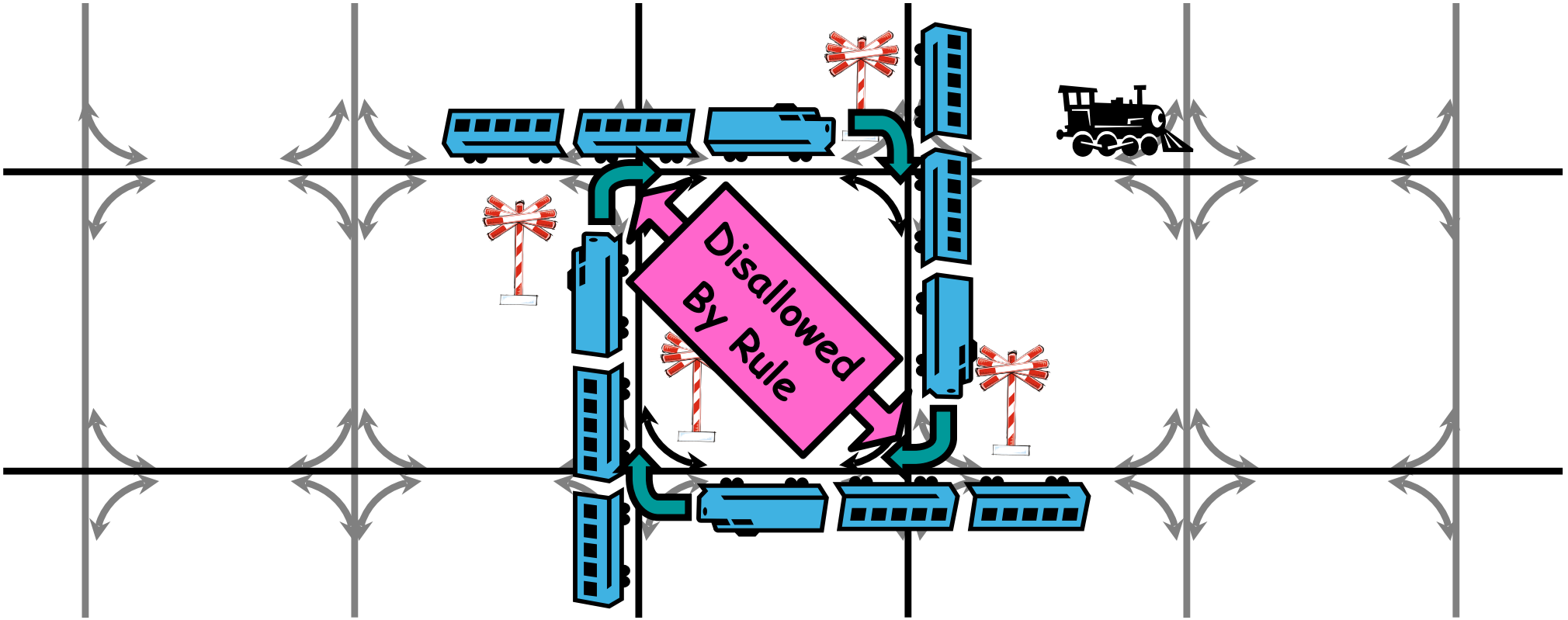
（1）静态分配：一个进程必须在执行前就申请它所要的全部资源，并且直到它所要的资源都得到满足后才开始执行。

（2）层次分配策略：资源被分成多个层次，当进程得到某一层的一个资源后，它只能再申请较高层次的资源。当进程要释放某层的一个资源时，必须先释放占有的较高层次的资源。当进程得到某一层的一个资源后，它想申请该层的另一个资源时，必须先释放该层中的已占资源。

（3）层次分配策略的变种按序分配策略：把系统的所有资源排一个顺序，例如，系统若共有n个进程,共有m个资源，用ri表示第i个资源，于是这m个资源是：r1,r2……,rm。规定如果进程不得在占用资源ri($1 \le i \le m$)后再申请rj($j<i$)。不难证明，按这种策略分配资源时系统不会发生死锁。

示例





死锁的避免 (Avoidance)

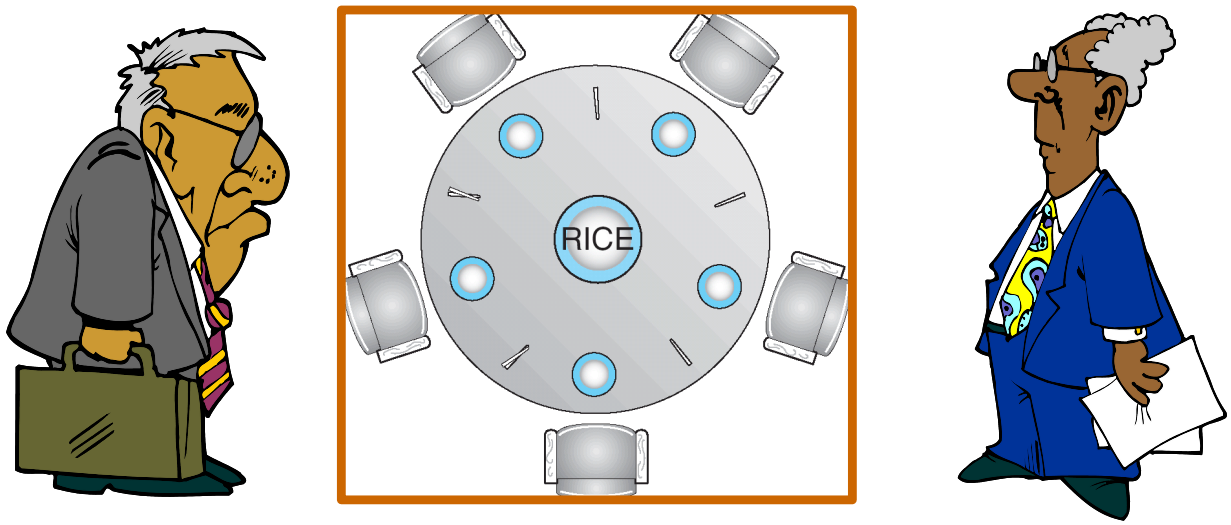
每次进行资源分配时，通过判断系统状态来决定这次分配后，是否仍存在不安全状态，否则不予分配。

- 每个进程声明其所需每种资源的最大数目
- 动态检查当前资源分配状态
- 资源分配状态取决于当前可用资源，已分配资源，以及进程声明所需最大资源数

Safe State安全状态

- 如果系统能按某个顺序为每个进程分配资源（不超过其最大值）并能避免死锁，那么系统状态就是安全的。
- 如果存在一个安全序列，那么系统处于安全态，

- 如果一个系统处于安全状态 ⇒ 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
- 避免：确保系统永远不会进入死锁状态

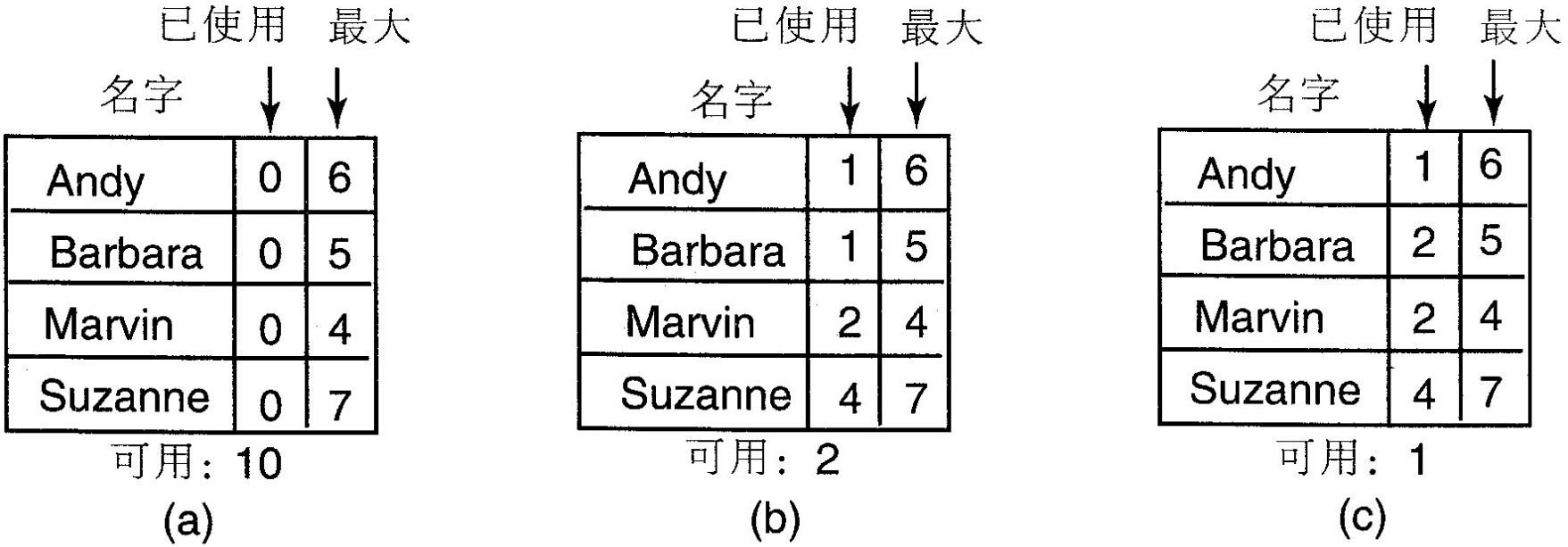


- 当前状态是安全状态 if:
  - 桌上剩余筷子多于一支
  - 桌上剩余一支筷子但随后会有人释放筷子

- 如果每个哲学家需要 k支筷子? :
  - 最后一支, 算上后没有任何人手上能有 k支
  - 倒数第二支, 算上后没有任何人手上能有 k-1支
  - 倒数第三支, 算上后没有任何人手上能有 k-2支
  - ...

- **银行家算法：** 银行家拥有一笔周转资金，客户要求分期贷款，如果客户能够得到各期贷款，就一定能够归还贷款，否则就一定不能归还贷款，银行家应谨慎的贷款，防止出现坏帐。

- **采用单种资源银行家算法避免死锁：** 操作系统（银行家）、操作系统管理的资源(周转资金)、进程（要求贷款的客户）。



（1）对每个请求进行检查，是否会导致不安全状态。若是，则不满足该请求；否则便满足。

（2）检查状态是否安全的方法是看他是否有足够的资源满足一个距最大需求最近的客户，如此反复下去。如果所有投资最终都被收回，则该状态是安全的，最初的请求可以批准。

（3）4个客户每个都有一个贷款额度：

（4）一个状态被称为是安全的：条件是存在一个状态序列能够使所有的客户均得到其所有的贷款。

（5）图示状态是安全的，以使Marvin运行结束，释放所有的4个单位资金。这样下去便可满足Suzanne或Barbara的请求。

（6）如果所有客户忽然都申请，希望得到最大贷款额，而银行家无法满足其中任何一个要求，则发生死锁。

多种资源的银行家算法：

总的资源E、已分配资源P、剩余资源A。

进程	磁带机	绘图仪	打印机	CD-ROM
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

已分配的资源

进程	磁带机	绘图仪	打印机	CD-ROM
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

仍需要的资源

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

1. 查找右边矩阵是否有一行，其未被满足的设备数均小于或等于向量A。如果找不到，系统将死锁，任何进程都无法运行结束。
2. 2. 若找到这样一行，可以假设它获得所需的资源并运行结束，将该进程标记为结束，并将资源加到向量A上。
3. 3. 重复以上两步，直到所有进程都标记为结束，则状态是安全的，否则将发生死锁。

银行家算法的基本思想:

1. 系统中的所有进程进入进程集合,
2. 在安全状态下系统收到进程的资源请求后,先把资源试探性分配给它。
3. 系统用剩下的可用资源和进程集合中其他进程还要的资源数作比较,在进程集合中找到剩余资源能满足最大需求量的进程,从而,保证这个进程运行完毕并归还全部资源。
4. 把这个进程从集合中去掉, 系统的剩余资源更多了,反复执行上述步骤。
5. 最后,检查进程集合,若为空表明本次申请可行,系统处于安全状态,可实施本次分配;否则,有进程执行不完， 系统处于不安全状态,本次资源分配暂不实施,让申请进程等待。

银行家算法中的数据结构

- 资源向量 Resource
  - 是一个含有m个元素，其中的每一个元素代表一类资源全部数目。
- 可利用资源向量Available
  - 是一个含有m个元素，其中的每一个元素代表一类可利用的资源数目，其初值是系统中所配置的该类全部可用资源数目。如果Available[j]=k，　表示系统中现有R_j类资源k个。
- 最大需求矩阵Claim
  - 是一个含有n×m的矩阵，它定义了系统中n个进程中的每一个进程对m类资源的最大需求。如果Claim(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Allocated

  是一个含有n×m的矩阵，它定义了系统中每一类资源当前已分配给每一进程的资源数。如果Allocation(i,j)=k， 表示进程i当前已分得R_j类资源k个。

- 需求矩阵Request

  是一个含有n×m的矩阵，用以表示每一个进程尚需的各类资源数。如果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。

  Request(i,j)= Claim(i,j)-Allocated(i,j)

银行家算法的程序：

```
type state= record          /*全局数据结构*/
    resource,available:array[0…m-1]of  integer;
    claim,allocated:array[0…n-1,0…m-1]of
    integer;
  end                       /*资源分配算法*/
```

```
if alloc[i,*]+request[*]>claim[i,*]  then  <error> /*申请量超过最大需求量*/
    else
      if request[*]>available[*] then <suspend process>
      else /*模拟分配*/
       <define newstate by:
        allocated[i,*]:=allocated[i,*]+request[*]
        available[*]:=available[*]-request[*]>
      end;
      if safe(newstate)  then <carry out allocation>
      else
       <restore original state> <suspend process>
      end
    end
```

```
function safe(state:s):boolean;   /*banker's algorithm*/
  var currentavail:array{0…m-1} of integer;
      rest:set of process;
begin
   currentavail:=available;
   rest:={all process};
   possible:=true;
   while possible do
     find a P_k in rest such that
      claim[k,*]-alloc[k,*]≤currentavail;
     if found  then
        currentavail:=currentavail+allocation[k,*];
        rest:=rest-[P_k];
         else
         possible:=false;
     end
    end;
    safe:=(rest=null)
end.
```

银行家算法的简短说明:

1. 申请量超过最大需求量时出错,否则转2;
2. 申请量超过目前系统拥有的可分配量时,挂起进程等待，否则转3;
3. 系统对P_i进程请求资源作试探性分配、执行:

   allocated[i,*]:=allocated[i,*]+request[*]

   available[*]:=available[*]-request[*]
4. 执行安全性测试算法,如果安全状态则承认试分配,否则抛弃试分配，进程P_i等待。

示例

假定系统中有五个进程{P0、P1、P2、P3、P4}和三种类型的资源{A，B，C}，每一种资源的数量分别为10、5、7，在T0时刻的资源分配情况如图

请找出该表中T0时刻以后存在的安全序列（至少2种）

资源情况 / 进程	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
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	

假定系统有进程集合（Po，Pl，P2，P3，P4），资源集合为（A，B，C），资源数量分别为（10，8，7）。假定某时刻系统的状态如表所示。

	Allocation			MAX			Available		
	A	B	C	A	B	C	A	B	C
PO	0	2	0	7	7	3	3	3	1
P1	2	1	0	3	3	2			
P2	3	0	2	9	1	2			
P3	2	1	2	2	3	3			
P4	0	1	2	4	3	4			

死锁的检测和解除

- 允许死锁发生，操作系统不断监视系统进展情况，判断死锁是否发生。
- 一旦死锁发生则采取专门的措施，解除死锁并以最小的代价恢复操作系统运行。

检测时机：当进程等待时检测死锁（系统开销大）；定时检测；系统资源利用率下降时检测死锁。

死锁检测算法：

（1）每个进程和资源指定唯一编号；

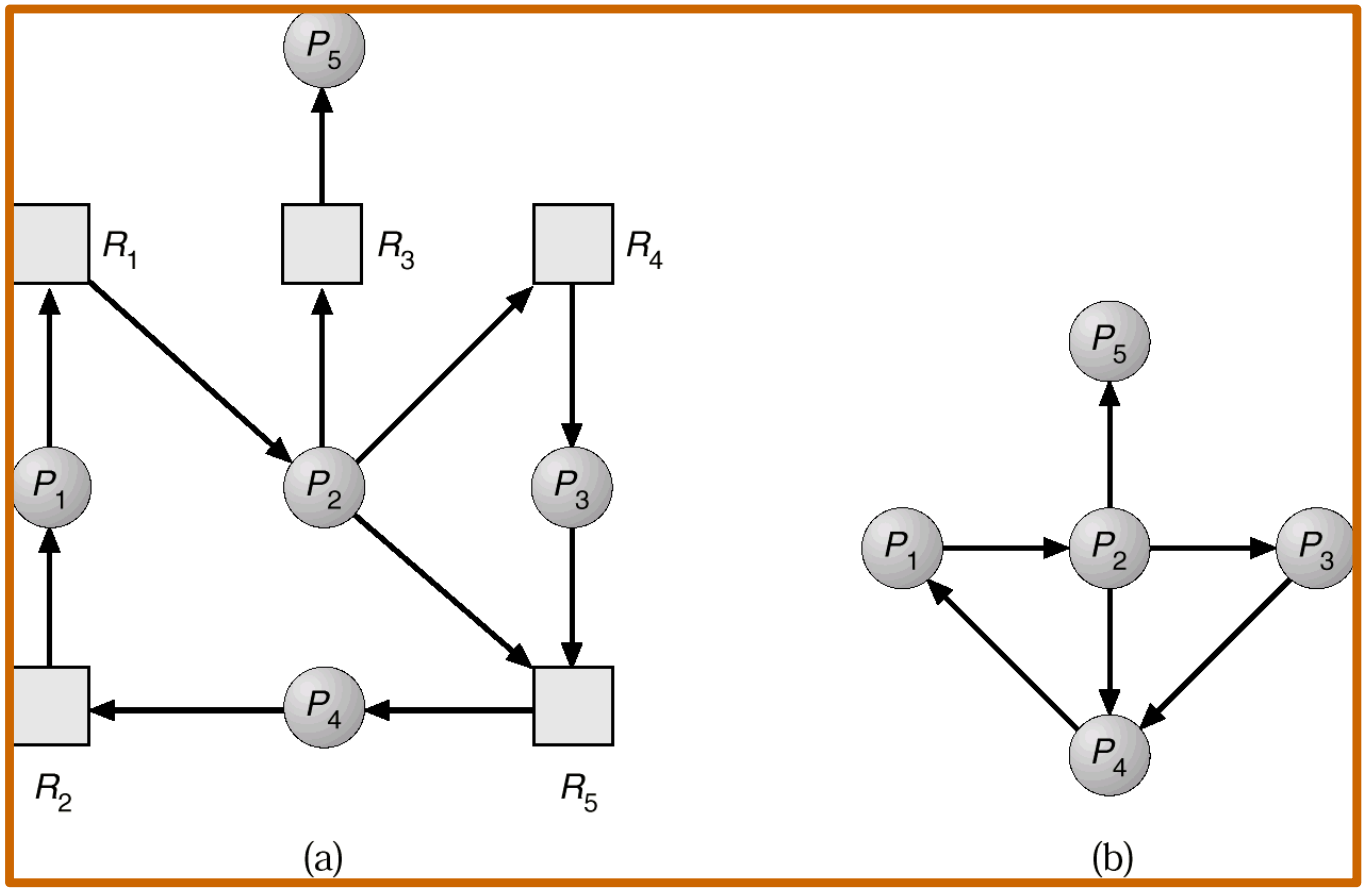
（2）设置一张资源分配表；

（3）记录各进程与其占用资源之间的关系；

（4）设置一张进程等待表；

（5）记录各进程与要申请资源之间的关系。

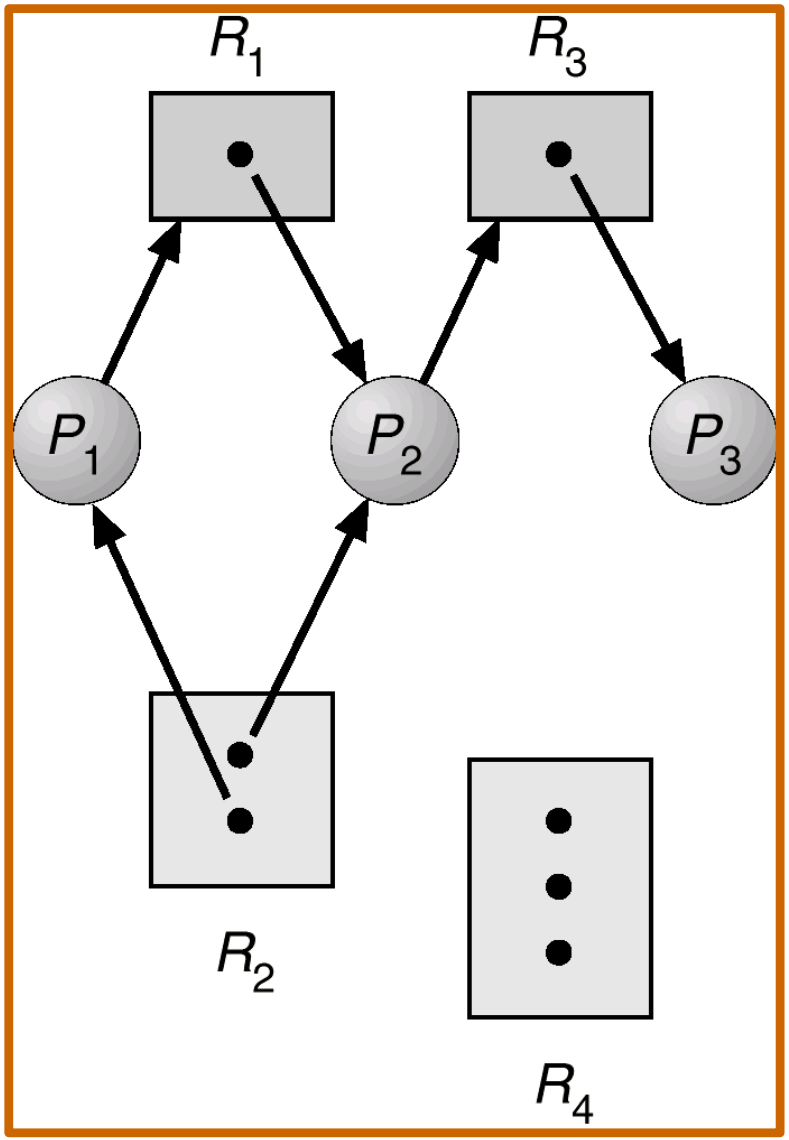
资源分配图 和 等待图示例 1



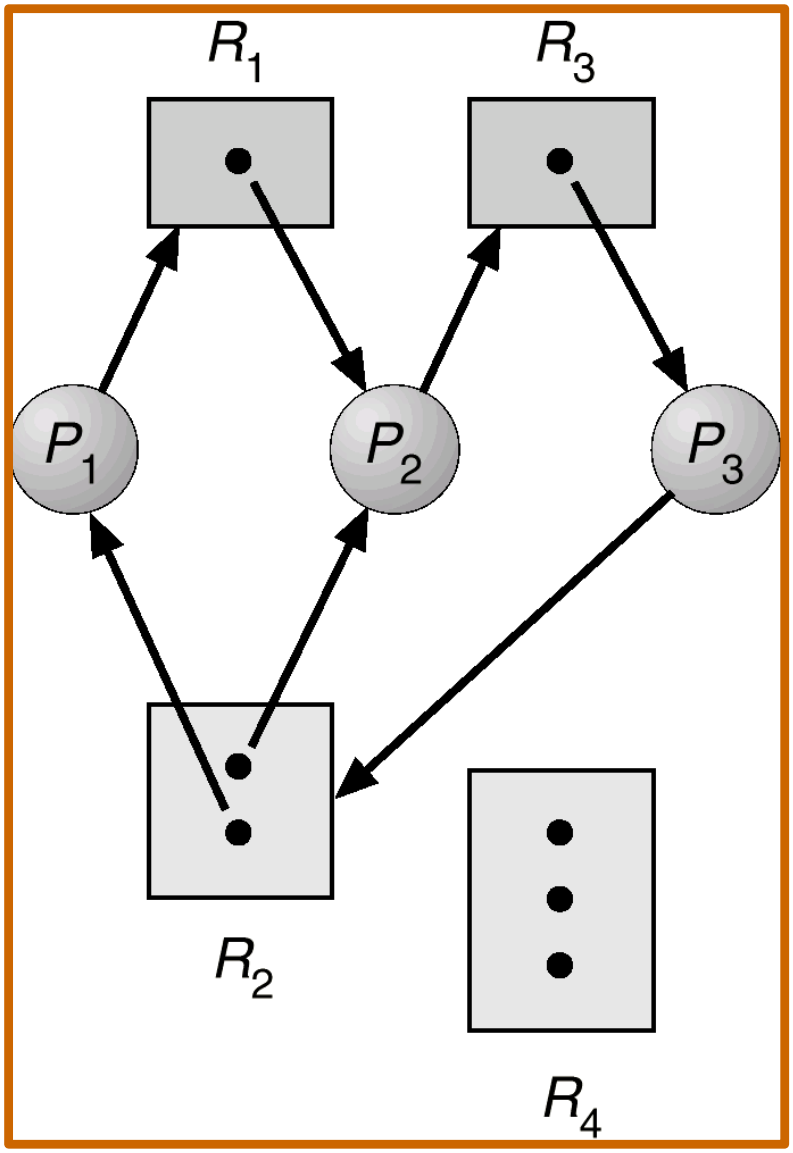
Resource-Allocation Graph

Corresponding wait-for graph

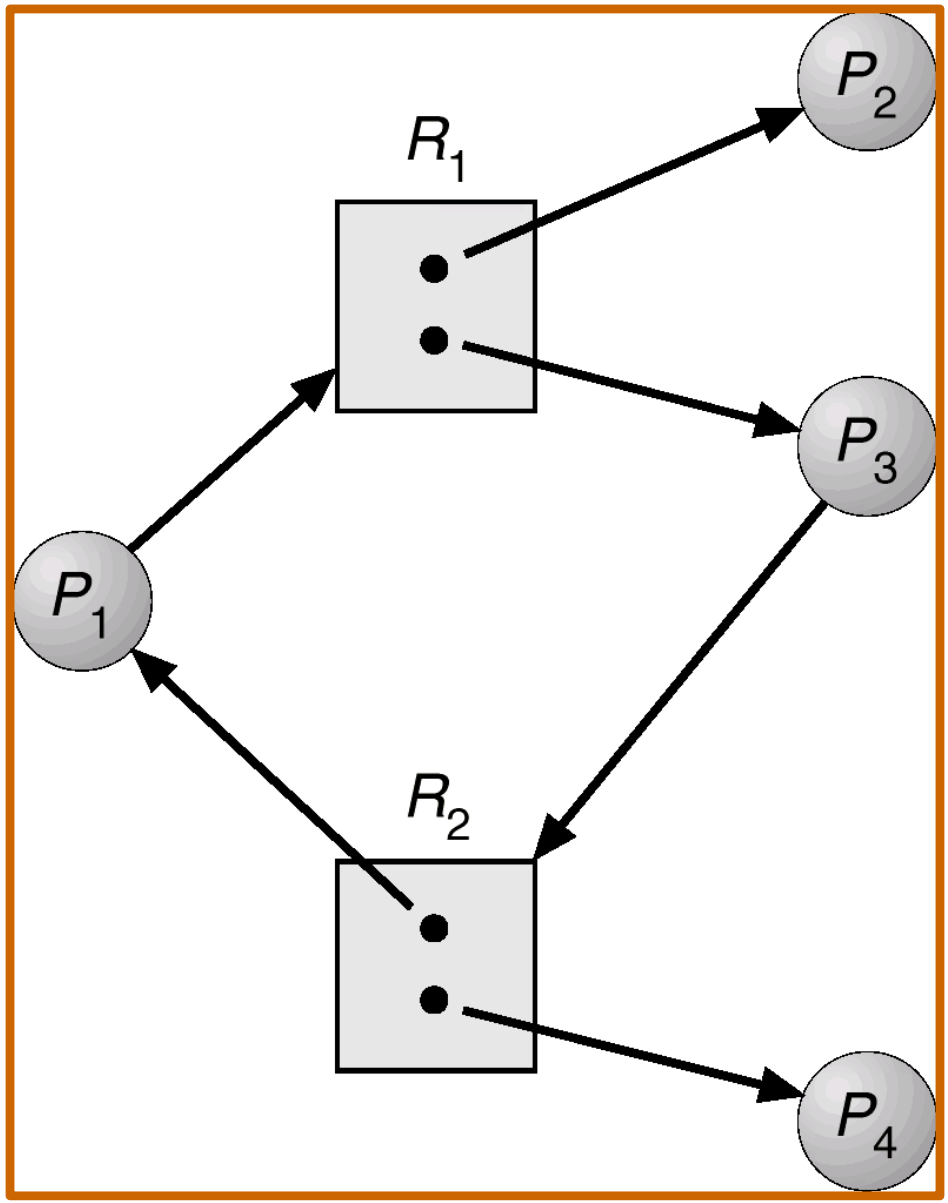
资源分配图 示例 2



资源分配图 示例 3



资源分配图 示例 4



思考题

- 一个OS有20个进程，竞争使用65个同类资源，申请方式是逐个进行的，一旦某个进程获得它所需要的全部资源，则立即归还所有资源。每个进程最多使用三个资源。若仅考虑这类资源，该系统有无可能产生死锁，为什么?

- 在某系统中，三个进程共享四台同类型的设备资源，这些资源一次只能一台地为进程服务和释放，每个进程最多需要二台设备资源，试问在系统中是否会产生死锁?

- 某系统中有n个进程和m台打印机，系统约定：打印机只能一台一台地申请、一台一台地释放，每个进程需要同时使用的打印机台数不超过m。如果n个进程同时使用打印机的总数小于m+n，试讨论，该系统可能发生死锁吗?并简述理由。

- 仅涉及一个进程的死锁有可能存在吗?为什么?