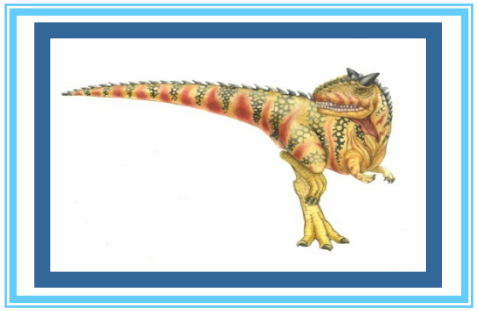
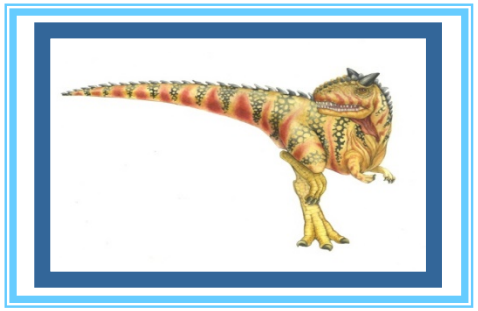
Chapter 7：Deadlocks死锁



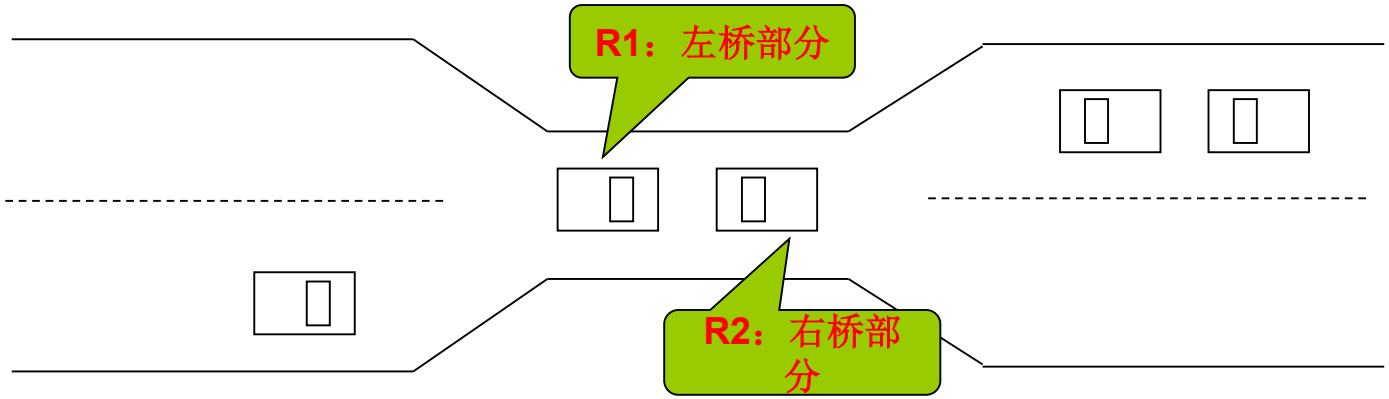
Deadlocks 死锁

- 7.1 System Model
- 7.2 Deadlock in Multithreaded Applications
- 7.3 Deadlock Characterization
- 7.4 Methods for Handling Deadlocks
- 7.5 Deadlock Prevention
- 7.6 Deadlock Avoidance
- 7.7 Deadlock Detection
- 7.8 Recovery from Deadlock

7.1 System Model



The Deadlock Problem



- Traffic only in one direction.
- Each section of a bridge can be viewed as a resource.
- 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 if a deadlock occurs.
- Starvation is possible.

The Deadlock Problem

- Deadlock： A set of blocked processes each holding a resource and waiting to acquire a resource held by another process in the set.
- 死锁：指多个进程因竞争共享资源而造成相互等待的一种僵局，若无外力作用，这些进程都将永远不能再向前推进
- Example 1
  - System has 2 tape drives.
  - P_1 and P_2 each hold one tape drive and each needs another one.
- Example 2
  - semaphores *A* and *B*, initialized to 1

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

How Deadlock ?

Deadlock program demo deadlock例

System Model

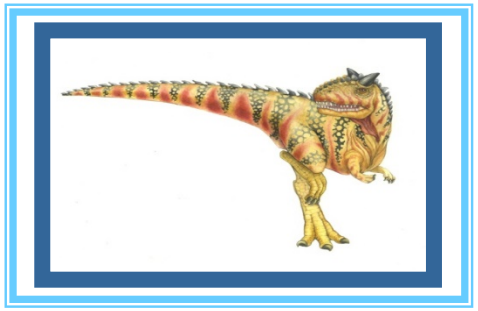
- Resource types $R_1, R_2, \ldots, R_m$:

  CPU cycles, memory space, I/O devices

- Each resource type R_i has W_i instances.

- Each process utilizes a resource as follows:
  - Request 申请
  - Use 使用
  - Release 释放

7.2 Deadlock Characterization



Deadlock Characterization

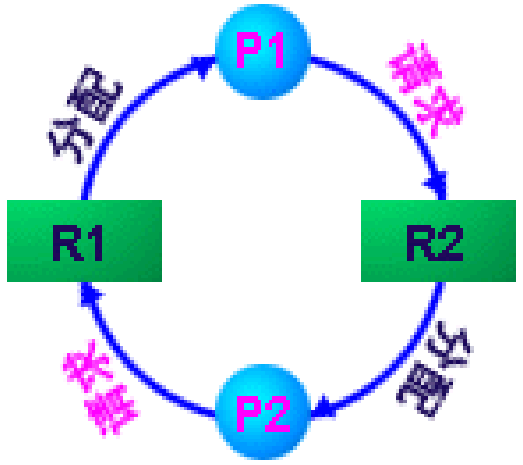
Necessary Conditions

- Deadlock can arise if four conditions hold simultaneously. 产生死锁的四个必要条件：
- Mutual exclusion（互斥）：only one process at a time can use a resource.
- Hold and wait（占有并等待、请求和保持）：a process holding at least one resource is waiting to acquire additional resources held by other processes.

  请求和保持(Hold and wait)条件：进程已经保持了至少一个资源，但又提出了新的资源要求，而该资源又已被其它进程占有，此时请求进程阻塞，但又对已经获得的其它资源保持不放

Deadlock Characterization (Cont.)

- No preemption（不可抢占、不剥夺）： a resource can be released only voluntarily by the process holding it, after that process has completed its task.
- Circular wait（循环等待）： there exists a set $\{P_0, P_1, \ldots, P_n\}$ of waiting processes such that P_0 is waiting for a resource that is held by P_1, P_1 is waiting for a resource that is held by P_2, …, P_{n-1} is waiting for a resource that is held by P_n, and P_n is waiting for a resource that is held by P_0.



Resource-Allocation Graph资源分配图

A set of vertices *V* and a set of edges *E*.

- V is partitioned into two types:
  - $P = \{P_1, P_2, \ldots, P_n\}$, 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.
- request edge – directed edge $P_i \rightarrow R_j$

  请求边：$P_i \rightarrow R_j$
- assignment edge – directed edge $R_j \rightarrow P_i$

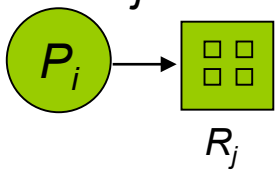
  分配边：$R_j \rightarrow P_i$

Resource-Allocation Graph (Cont.)

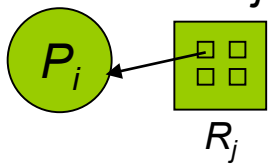
- Process

- Resource Type with 4 instances (实例、个数)

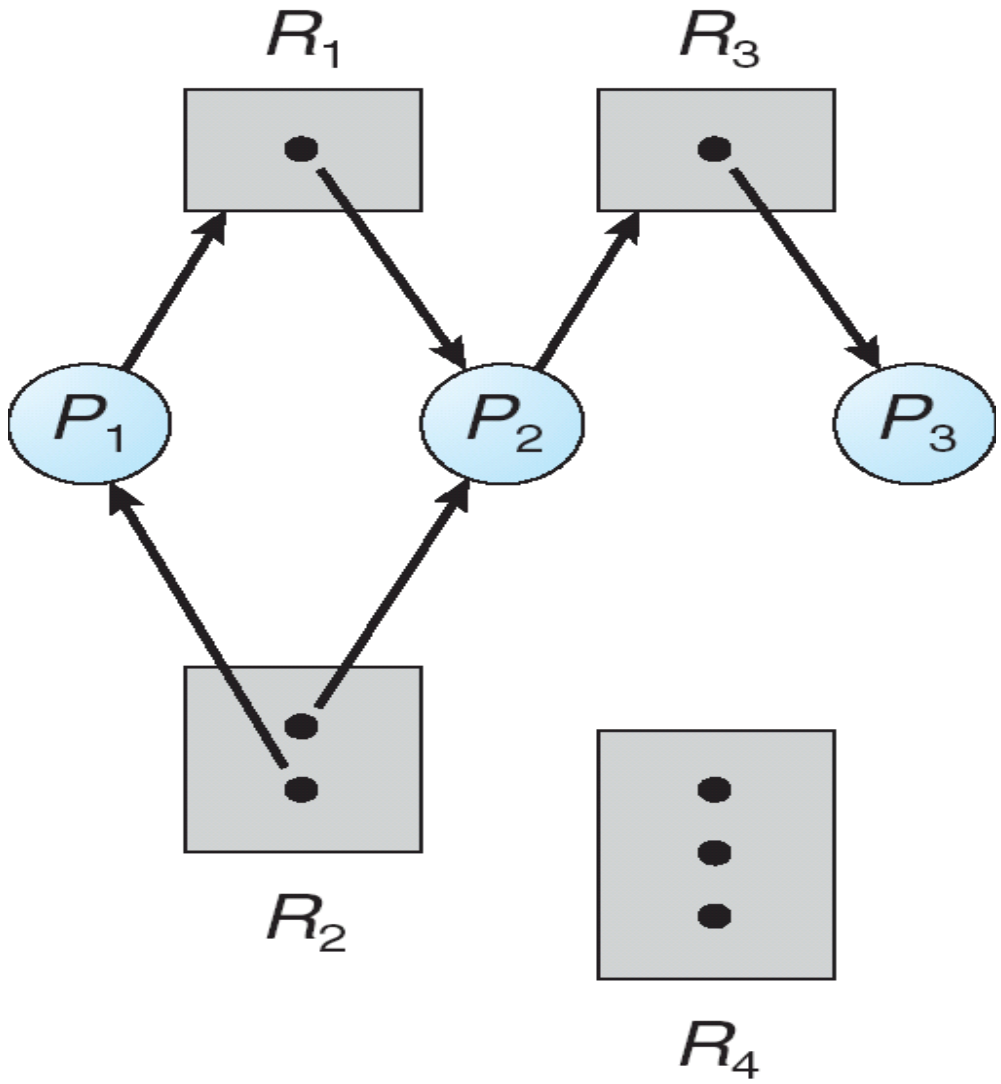
- P_i requests instance of R_j



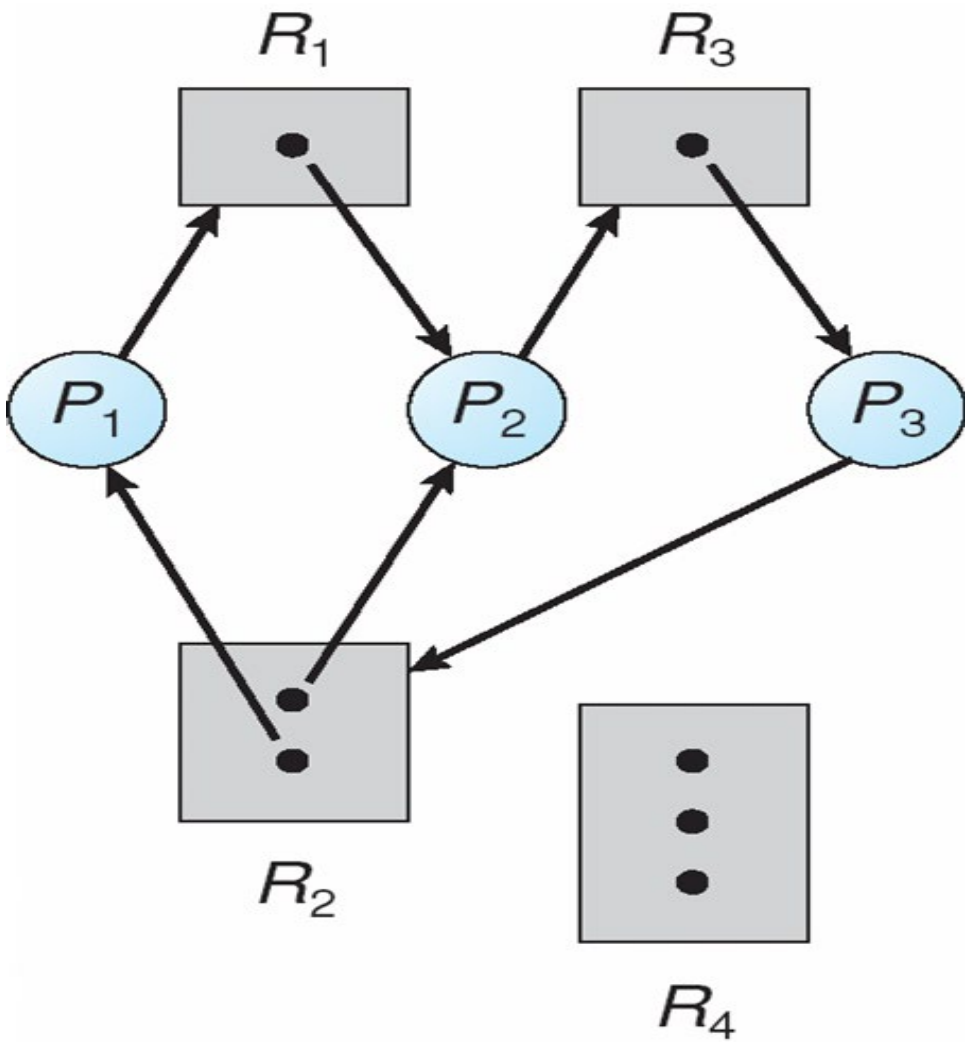
- P_i is holding an instance of R_j



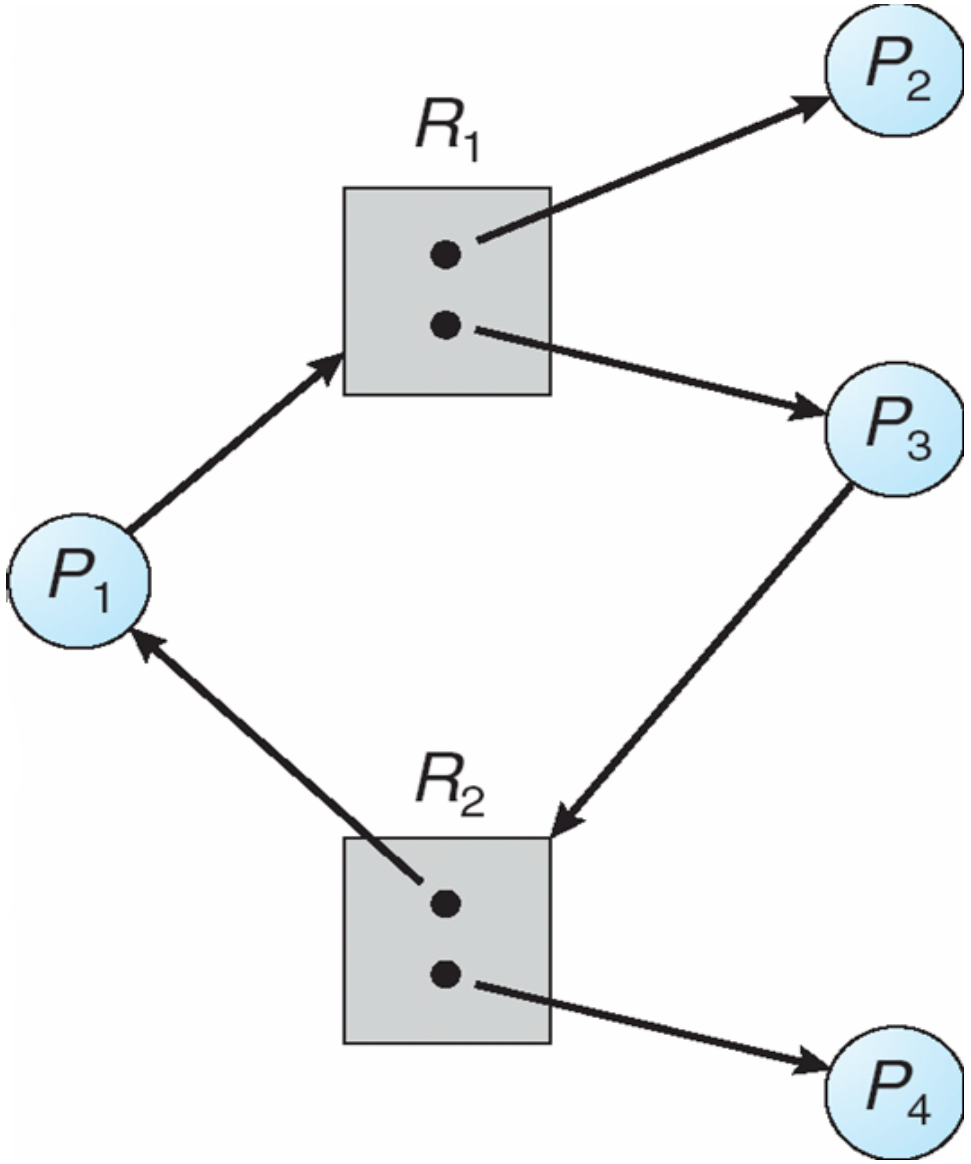
Example of a Resource Allocation Graph



Resource Allocation Graph With A Deadlock



Resource Allocation Graph With A Cycle But No Deadlock



Basic Facts

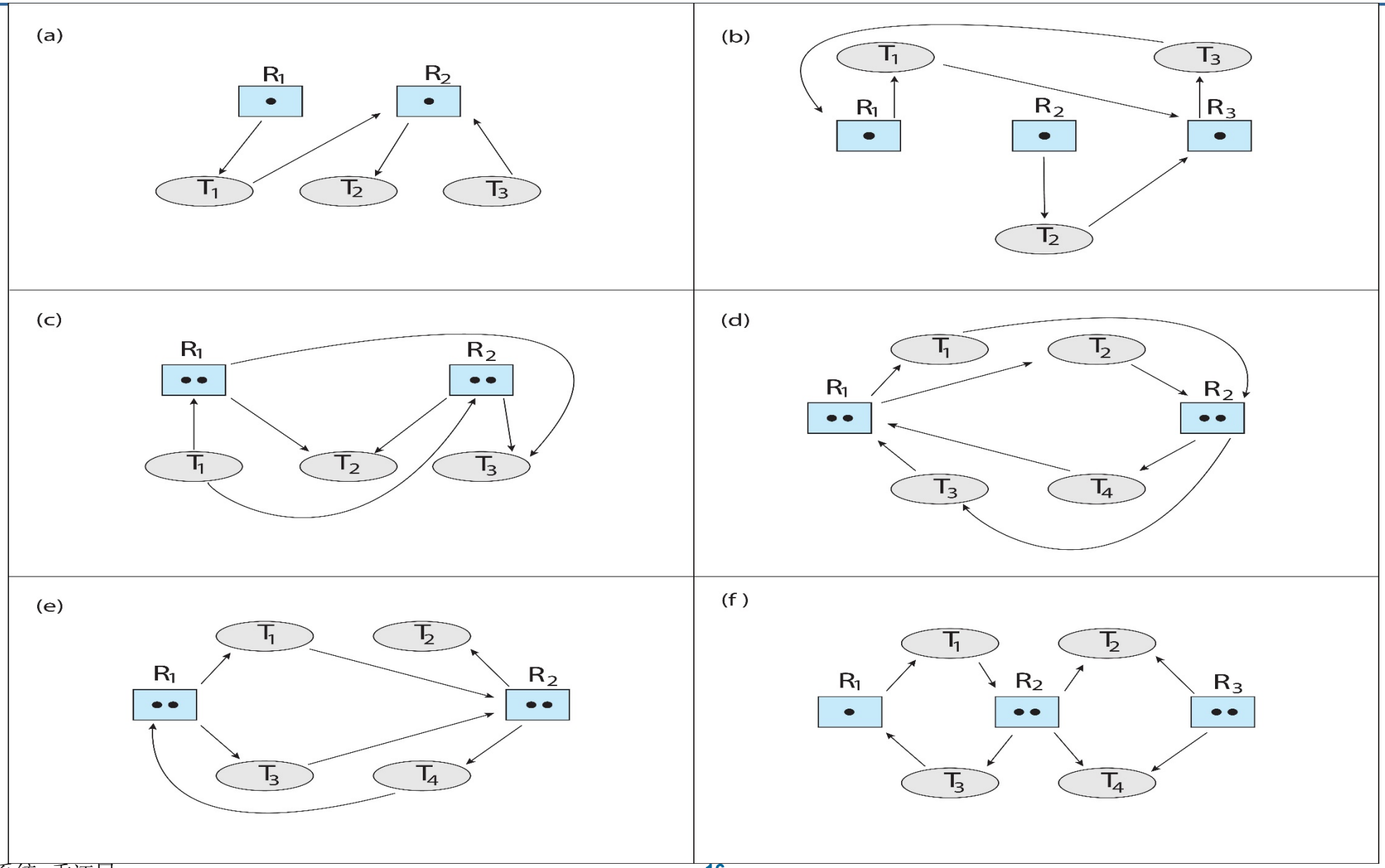
- If graph contains no cycles ⇒ no deadlock.

Why ?

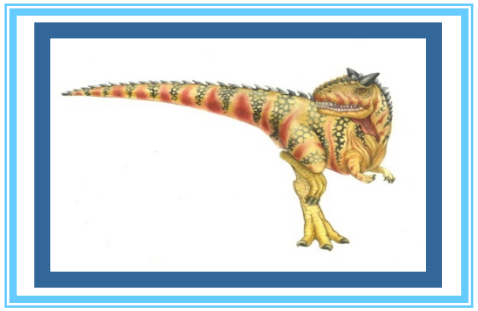
- If graph contains a cycle ⇒
  - if only one instance per resource type, then deadlock.
  - if several instances per resource type, possibility of deadlock.

- 死锁定理：S为死锁状态的充分条件是：尚且仅当S状态的资源分配图是不可完全简化的。
- 资源分配图（有向图）的简化。--离散数学算法

资源分配图（线程）


(a)
R1
R2
T1
T2
T3
(b)
T1
T3
R1
R2
R3
T2
(c)
R1
R2
T1
T2
T3
(d)
T1
T2
R1
R2
T3
T4
(e)
T1
T2
R1
R2
T3
T4
(f)
T1
T2
R1
R2
R3
T3
T4

7.3 Methods for Handling Deadlocks



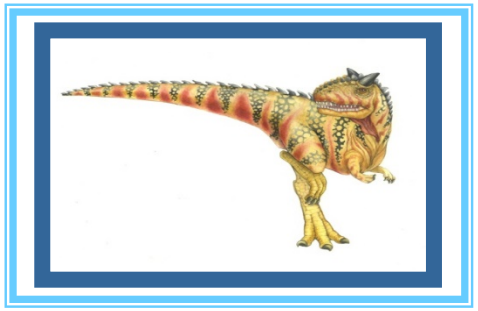
Methods for Handling Deadlocks

死锁处理的策略：

- Ensure that the system will *never* enter a deadlock state.
  - Prevention死锁预防、 Avoidance死锁避免
- Allow the system to enter a deadlock state and then recover.
  - Detection死锁检测、 Recovery死锁解除
- Ignore the problem and pretend that deadlocks never occur in the system; used by most operating systems, including *UNIX*、*Linux*、*Windows*.

  ---鸵鸟方法

7.4 Deadlock Prevention



Deadlock Prevention（死锁预防）

- Mutual Exclusion – not required for sharable resources; must hold for nonsharable resources.
  - 虚拟化
- **Restrain the ways request can be made.**
- Hold and Wait – must guarantee that whenever a process requests a resource, it does not hold any other resources.
  - Require process to request and be allocated all its resources before it begins execution, or allow process to request resources only when the process has none. 资源静态预分配方式
  - Low resource utilization; starvation possible.

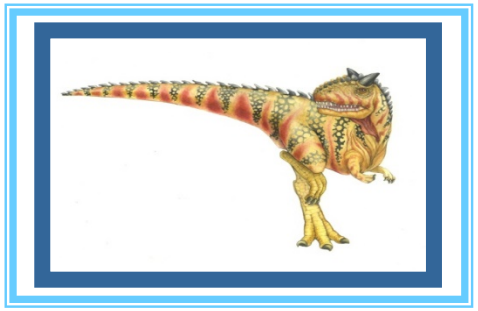
Deadlock Prevention (Cont.)

- No Preemption
  - If a process that is holding some resources requests another resource that cannot be immediately allocated to it, then all resources currently being held are released.
  - Preempted resources are added to the list of resources for which the process is waiting.
  - Process will be restarted only when it can regain its old resources, as well as the new ones that it is requesting.

Deadlock Prevention (Cont.)

- Circular Wait
  - impose a total ordering of all resource types, and require that each process requests resources in an increasing order of enumeration.
  - F(tape drive)=1，F(disk drive)=5，F(printer)=12
  - 资源的有序申请破坏了循环等待条件

7.5 Deadlock Avoidance



Deadlock Avoidance 死锁避免

Requires that the system has some additional *a prior* information available.

- Simplest and most useful model requires that each process declare the *maximum number* of resources of each type that it may need.
- The deadlock-avoidance algorithm dynamically examines the resource-allocation state to ensure that there can never be a circular-wait condition.
- Resource-allocation *state* is defined by the number of available and allocated resources, and the maximum demands of the processes.

Deadlock Avoidance (Cont.)

- 7.5.1 Safe State安全状态
- When a process requests an available resource, system must decide if immediate allocation leaves the system in a *safe state*.
- 安全状态是指系统的一种状态，在此状态开始系统能按某种顺序（如P1、P2……Pn）来为各个进程分配其所需资源，直至最大需求，使每个进程都可顺序地一个个地完成。这个序列（P1、P2……Pn）称为**安全序列**。若系统此状态不存在一个安全序列，则称系统处于不安全状态。
- System is in safe state if there exists a safe sequence of all processes.

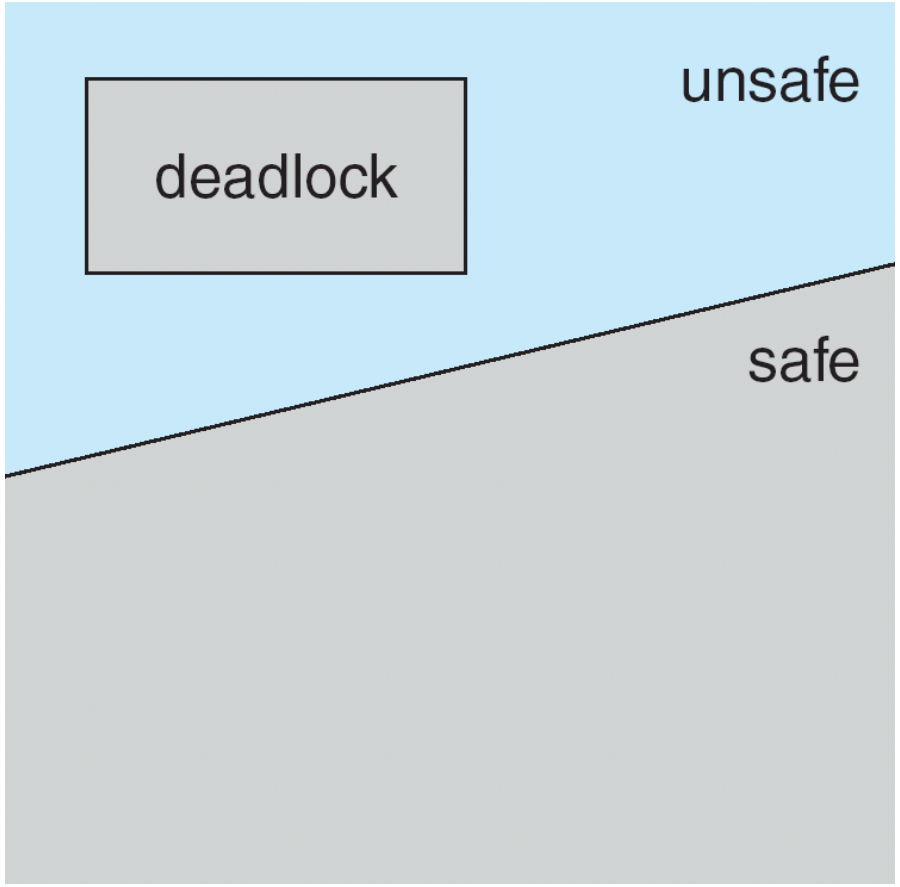
Safe State (Cont.)

- Sequence <P_1, P_2, …, P_n> is safe if for each P_i, the resources that *Pi* can still request can be satisfied by currently available resources + resources held by all the P_j, with $j<i$.
  - If P_i resource needs are not immediately available, then P_i can wait until all P_j have finished.
  - When P_j is finished, P_i can obtain needed resources, execute, return allocated resources, and terminate.
  - When P_i terminates, P_{i+1} can obtain its needed resources, and so on.

Basic Facts

- If a system is in safe state ⇒ no deadlocks.
- If a system is in unsafe state ⇒ possibility of deadlock.
- Avoidance ⇒ ensure that a system will never enter an unsafe state.

Safe, Unsafe , Deadlock State spaces



例

例：假如系统中有P1、P2和P3三个进程和12台磁带机。各进程最大需求和T0时刻分配状态如下：

进程	最大需求	已分配	还需请求	可用
P1	10	5	5	3
P2	4	2	2	
P3	9	2	7	

分析T0状态，可以找到一个安全序列（P2、P1、P3）,即系统按此进程序列分配资源，每个进程都可顺利完成。

由安全状态向不安全状态的转换

如果在T0 状态不按安全序列进行分配，可能会导致系统进入一个不安全状态，例如在T0状态下P3申请1台磁带机。如系统实施此次分配使系统状态由T0变为T1状态，分析T1状态安全情况。

T1时刻状态：

进程	最大需求	已分配	还需请求	可用
P1	10	5	5	2
P2	4	2	2	
P3	9	3	6	

Avoidance algorithms

- Single instance of a resource type
  - Use a resource-allocation graph
- Multiple instances of a resource type
  - Use the banker's algorithm

7.5.2 Resource-Allocation Graph Algorithm
资源分配图算法

- *Claim edge*（需求边）$P_i \rightarrow R_j$ indicated that process P_j may request resource R_j; represented by a dashed line.
- Claim edge converts to *request edge*（请求边）when a process requests a resource.
- Request edge converted to an assignment edge （分配边） when the resource is allocated to the process
- When a resource is released by a process, assignment edge reconverts to a claim edge.
- Resources must be claimed *a priori* in the system.

Resource-Allocation Graph Algorithm

- 算法：假设进程Pi申请资源Rj。只有在**需求边**$P_i \rightarrow R_j$ 变成**分配边** $R_j \rightarrow P_i$ 而不会导致资源分配图形成环时，才允许申请。
- 用算法循环检测，如果没有环存在，那么资源分配会使系统处于安全状态。如果存在环，资源分配会使系统不安全。进程Pi必须等待。

Resource-Allocation Graph For Deadlock Avoidance

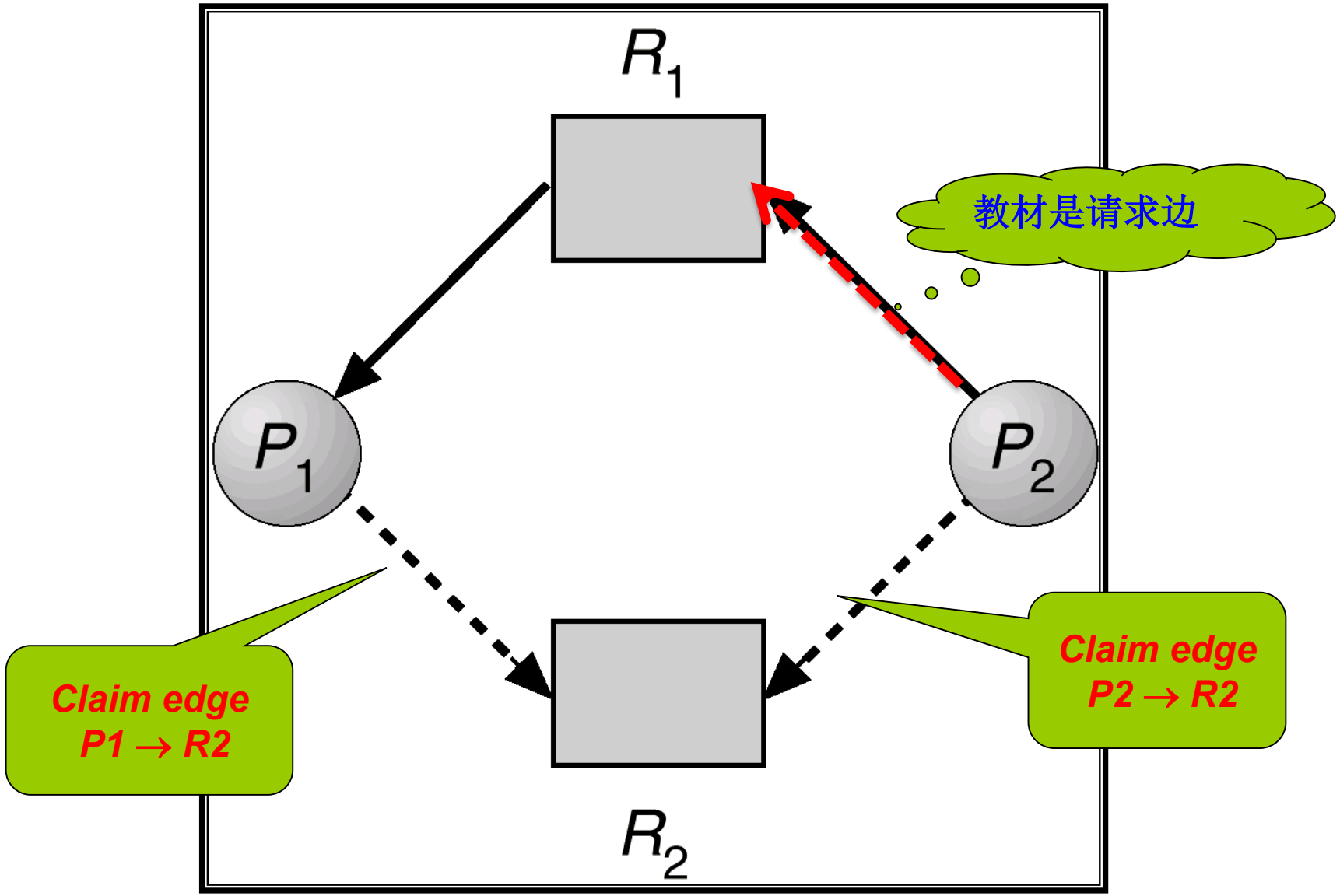
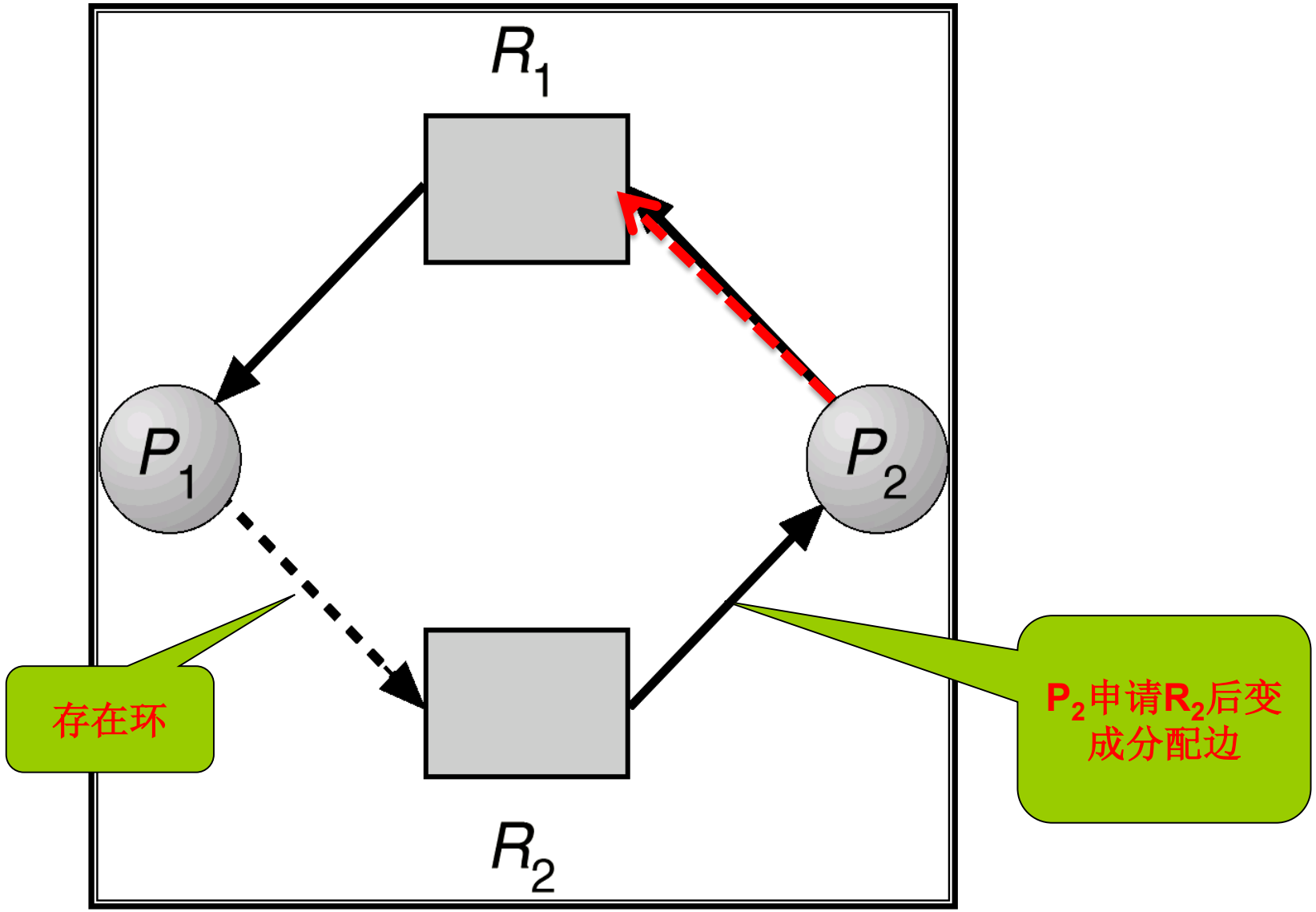


Fig 7.6 Unsafe State In Resource-Allocation Graph



7.5.3 Banker's Algorithm（银行家算法）

- *Multiple instances*
  - *最具代表的避免死锁算法是Dijkstra的银行家算法，这是由于该算法用于银行系统现金贷款的发放而得名。*
  - *例：设银行家有10万贷款，P、Q、R分别需要8,3,9万元搞项目，如果P已申请到了4万，Q要申请2万，显然，如果满足Q的申请，有安全序列<P,Q,R>/<Q,P,R>。但如果R要申请4万，若满足R的申请，显然不存在安全序列。*
- *Each process must a prior claim maximum use.*
- *When a process requests a resource it may have to wait.*
- *When a process gets all its resources it must return them in a finite amount of time.*

Data Structures for the Banker's Algorithm

Let *n* = number of processes, and *m* = number of resources types.

- *Available:* Vector of length *m*. If available [*j*] = *k*, there are *k* instances of resource type R_j available.
- *Max:* *n* x *m* matrix. If *Max* [*i,j*] = *k*, then process P_i may request at most *k* instances of resource type R_j.
- *Allocation:* *n* x *m* matrix. If Allocation[*i,j*] = *k* then P_i is currently allocated *k* instances of R_j.

Data Structures for the Banker's Algorithm (Cont.)

- *Need*: *n* x *m* matrix. If *Need*[*i*,*j*] = *k*, then P_i may need *k* more instances of R_j to complete its task.

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

7.5.3.1 Safety Algorithm安全算法

1. Let *Work* and *Finish* be vectors of length m and n, respectively. Initialize:

   Work = Available

   Finish [i] = false for i = 1,2,3, …, n.

2. Find an index i such that both:

   (a). Finish [i] = false

   (b). $Need_i \leq Work$

   If no such i exists, go to step 4.

3. Work = Work + $Allocation_i$

   Finish[i] = true

   go to step 2.

4. If Finish [i] == true for all i, then the system is in a safe state.

7.5.3.2 Resource-Request Algorithm

$Request_i$ = request vector for process P_i. If $Request_i[j] = k$ then process P_i wants k instances of resource type R_j.

1. If $Request_i \leq Need_i$ go to step 2. Otherwise, raise error condition, since process has exceeded its maximum claim.
2. If $Request_i \leq Available$, go to step 3. Otherwise P_i must wait, since resources are not available.
3. Pretend to allocate requested resources to P_i by modifying the state as follows:

$$Available = Available - Request_i;$$

$$Allocation_i = Allocation_i + Request_i;$$

$$Need_i = Need_i - Request_i;$$

- Call Safety Algorithm
- If safe $\Rightarrow$ the resources are allocated to P_i.
- If unsafe $\Rightarrow$ P_i must wait, and the old resource-allocation state is restored

Example of Banker's Algorithm

- 5 processes P_0 through P_4; 3 resource types *A* (10 instances), *B* (5instances, and *C* (7 instances).
- Snapshot at time T_0:

	Allocation	*Max*	*Available*
	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	

问：
(1) State of system?
(2) Can request for (1,0,2) by P1 be granted?
(3) Can request for (3,3,0) by P4 be granted?
(4) Can request for (0,2,0) by P0 be granted?

(1) State of system?

	Max	Allocation	Need	Available	work	分配前	释放后	Finish
	A B C	A B C	A B C	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	3 3 2	④ 7 4 5	7 5 5	
P1	3 2 2	2 0 0	1 2 2			① 3 3 2	5 3 2	
P2	9 0 2	3 0 2	6 0 0			⑤ 7 5 5	10 5 7	
P3	2 2 2	2 1 1	0 1 1			② 5 3 2	7 4 3	
P4	4 3 3	0 0 2	4 3 1			③ 7 4 3	7 4 5	

- The system is in a safe state since the sequence $< P_1, P_3, P_4, P_0, P_2 >$ satisfies safety criteria.

(2) P_1 Request (1,0,2)

① Check that $Request_1(1,0,2) \leq Need_1(1,2,2)$, (that is *true*)

② Check that $Request_1(1,0,2) \leq Available(3,3,2)$, (that is *true*)

③ 试探把资源分配给进程P1并修改有关数据结构的数值：

$Available = Available(3,3,2) - Request_1(1,0,2) \Rightarrow Available(2,3,0)$;

$Need_1 = Need_1(1,2,2) - Request_1(1,0,2) = \Rightarrow Need_1(0,2,0)$;

$Allocation_1 = Allocation_1(2,0,0) + Request_1(1,0,2) \Rightarrow Allocation_1(3,0,2)$;

④ *Call* Safety Algorithm

(2) P_1 Request (1,0,2)

	Max	Allocation	Need	Available	work	分配前	释放后
	A B C	A B C	A B C	A B C	A B C	A B C	A B C
P0	7 5 3	0 1 0	7 4 3	2 3 0	2 3 0	④ 7 4 5	7 5 5
P1	3 2 2	3 0 2	0 2 0			① 2 3 0	5 3 2
P2	9 0 2	3 0 2	6 0 0			⑤ 7 5 5	10 5 7
P3	2 2 2	2 1 1	0 1 1			② 5 3 2	7 4 3
P4	4 3 3	0 0 2	4 3 1			③ 7 4 3	7 4 5

- Executing safety algorithm shows that sequence <P_1, P_3, P_4, P_0, P_2> satisfies safety requirement.

(3) Can request for (3,3,0) by P_4 be granted?

① $Request_4(3,3,0) \leq Need_4(4,3,1)$，P4请求在最大需求范围内。

② Request4(3,3,0)≤Available(2,3,0)不成立，即可用资源暂不能满足P4请求资源需要，P4阻塞等待。

(4) Can request for (0,2,0) by P_0 be granted?

① Check that $Request_0(0,2,0) \leq Need_0(7,4,3)$, (that is *true*)

② Check that $Request_0(0,2,0) \leq Available(2,3,0)$, (that is *true*)

③ 试探把资源分配给进程P0并修改有关数据结构的数值：

$Available = Available(2,3,0) - Request_0(0,2,0) \Rightarrow Available(2,1,0)$;

$Need_0 = Need_0(7,4,3) - Request_0(0,2,0) \Rightarrow Need_0(7,2,3)$;

$Allocation_0 = Allocation_0(0,1,0) + Request_0(0,2,0) \Rightarrow Allocation_0(0,3,0)$;

④ Call Safety Algorithm

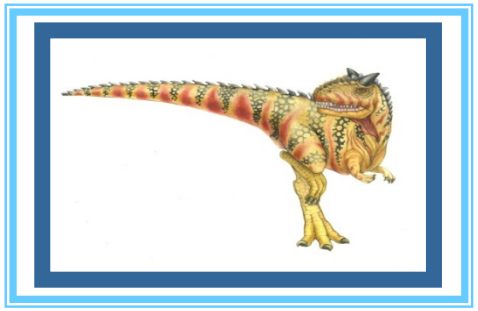
(4) P_0 Request (0,2,0)

	Max	Allocation	Need	Available	work	分配前	释放后
	A B C	A B C	A B C	A B C	A B C	A BC	A B C
P0	7 5 3	0 3 0	7 2 3	2 1 0	2 1 0		
P1	3 2 2	3 0 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				

① $Need_i > Work\ i=0,\ldots,4$ 不安全

② P0请求不能分配

7.6 Deadlock Detection



Deadlock Detection死锁检测

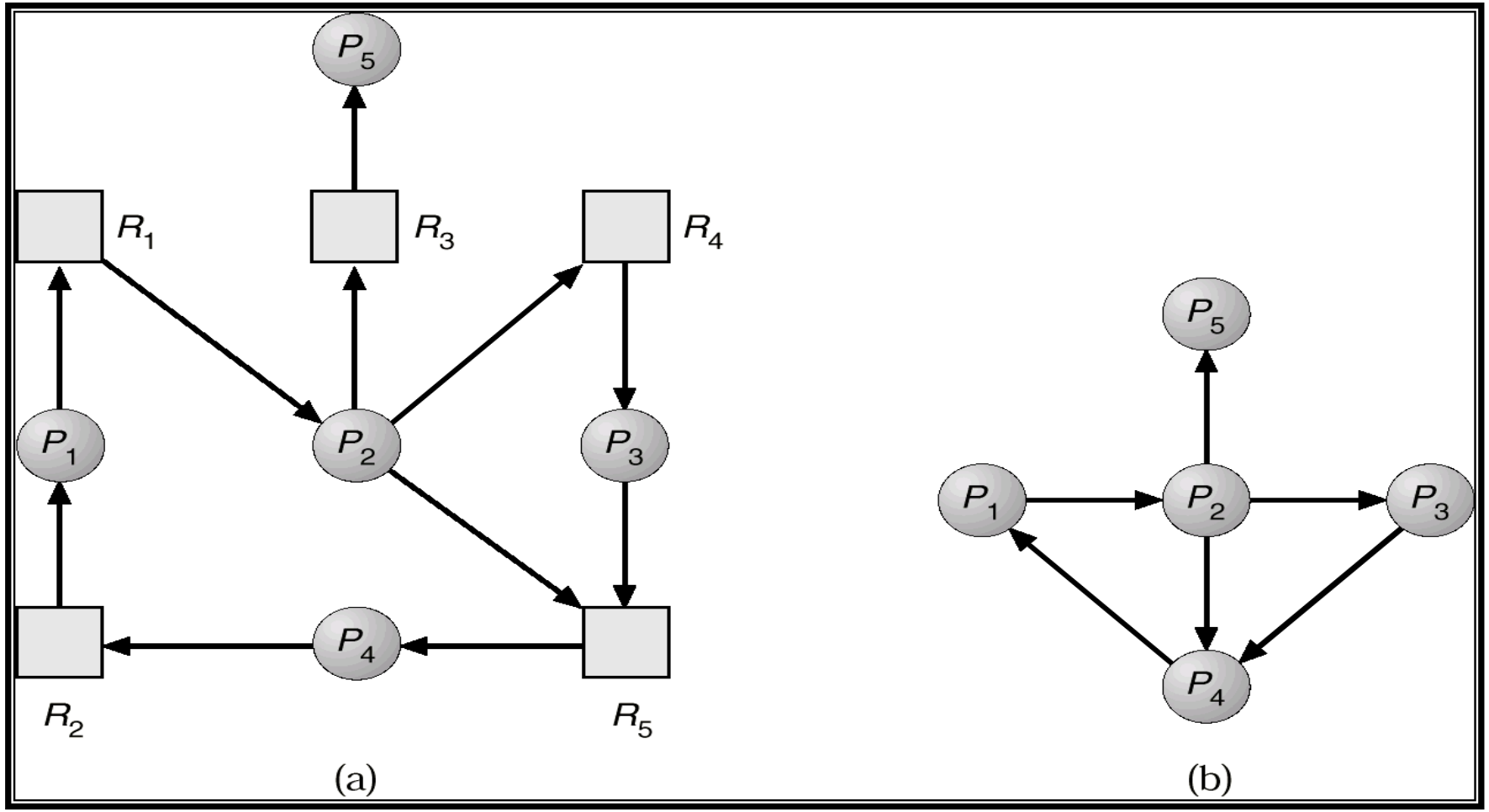
- Allow system to enter deadlock state

- Detection algorithm
- Recovery scheme

7.6.1 Single Instance of Each Resource Type

- Maintain *wait-for* graph 等待图—资源分配图的变形
- Nodes are processes.
  - $P_i \rightarrow P_j$ if P_i is waiting for P_j.
- Periodically invoke an algorithm that searches for a cycle in the graph.
- An algorithm to detect a cycle in a graph requires an order of n^2 operations, where n is the number of vertices in the graph.

Fig 7.7 Resource-Allocation Graph and Wait-for Graph



Resource-Allocation Graph **Corresponding wait-for graph**

7.6.2 Several Instances of a Resource Type

- Data structures:
  - *Available:* A vector of length *m* indicates the number of available resources of each type.
  - *Allocation:* An *n* x *m* matrix defines the number of resources of each type currently allocated to each process.
  - *Request:* An *n* x *m* matrix indicates the current request of each process. If *Request* [i,j] = *k*, then process P_i is requesting *k* more instances of resource type.

Detection Algorithm

1. Let Work and Finish be vectors of length m and n, respectively Initialize:

   (a) Work = Available

   (b) For i = 1,2, …, n, if $Allocation_i \neq 0$, then
   Finish[i] = false;otherwise, Finish[i] = true.

2. Find an index i such that both:

   (a) Finish[i] == false

   (b) $Request_i \leq Work$

   If no such i exists, go to step 4.

Detection Algorithm (Cont.)

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.

Algorithm requires an order of O(m x $n^{2)}$ operations to detect whether the system is in deadlocked state.

Example of Detection Algorithm

- Five processes P_0 through P_4; three resource types A (7 instances), B (2 instances), and C (6 instances).
- Snapshot at time T_0

	Allocation	*Request*	*Available*
	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	

- Sequence <P_0, P_2, P_3, P_1, P_4> will result in *Finish*[*i*] = true for all *i*.

Example (Cont.)

- P_2 requests an additional instance of type *C*.

	Allocation	*Request*	*Available*
	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 1	
P_3	2 1 1	1 0 0	
P_4	0 0 2	0 0 2	

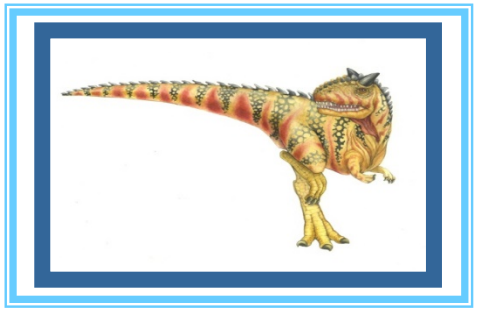
- State of system?
  - Can reclaim resources held by process P_0, but insufficient resources to fulfill other processes requests.
  - Deadlock exists, consisting of processes P_1, P_2, P_3, and P_4.

7.6.3 Detection-Algorithm Usage

- When, and how often, to invoke depends on:
  - How often a deadlock is likely to occur?
  - How many processes will need to be rolled back?

- If detection algorithm is invoked arbitrarily, there may be many cycles in the resource graph and so we would not be able to tell which of the many deadlocked processes “caused” the deadlock.

7.7 Recovery from Deadlock



Recovery from Deadlock死锁恢复

- 检测到死锁后采取措施：
  - 通知系统管理员
  - 系统自己恢复
- 打破死锁两种方法：
  - 进程终止
  - 抢占资源

7.7.1 Process Termination

- Abort all deadlocked processes.
- Abort one process at a time until the deadlock cycle is eliminated.

- Many factors may determine which process is chosen,include:
  - Priority of the process.
  - How long process has computed, and how much longer to completion.
  - Resources the process has used.
  - Resources process needs to complete.
  - How many processes will need to be terminated.
  - Whether the process is interactive or batch.

7.7.2 Resource Preemption

- Selecting a victim – minimize cost最小化代价
- Rollback（回退）– return to some safe state, restart process for that state.
- Starvation – same process may always be picked as victim, include number of rollback in cost factor.

习题分析

- ex_ch7.doc

Homework

- 学在浙大 c.zju.edu.cn

Reading Assignments

- Read for this week:
  - Chapters 7
    of the text book:
- Read for next week:
  - Chapters 8
    of the text book: