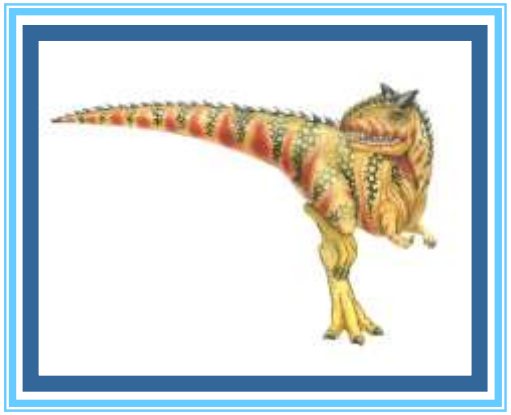
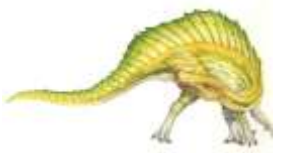
Chapter 7: Deadlocks



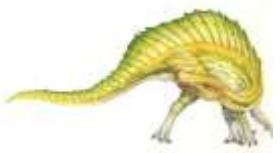
Chapter 7: Deadlocks

- System Model
- Deadlock Characterization
- Methods for Handling Deadlocks
- Deadlock Prevention
- Deadlock Avoidance
- Deadlock Detection
- Recovery from Deadlock



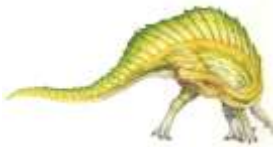
Chapter Objectives

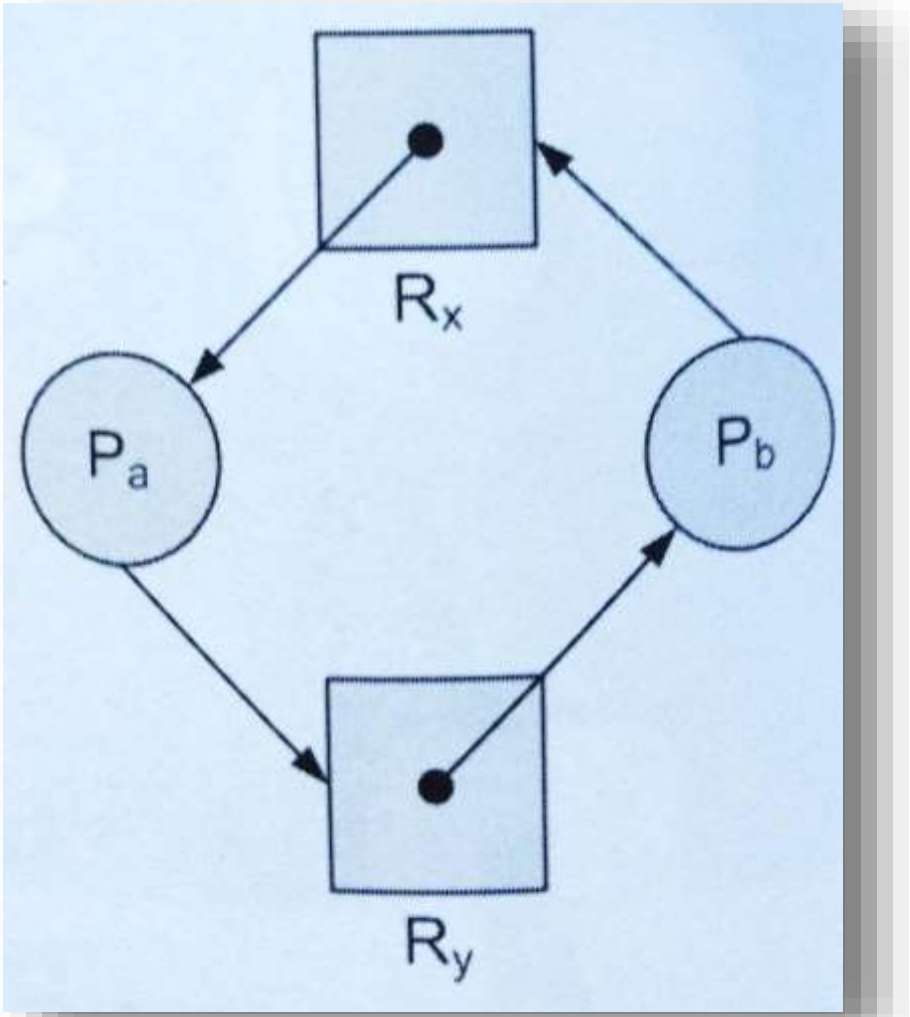
- To develop a description of deadlocks, which prevent sets of concurrent processes from completing their tasks
- To present a number of different methods for preventing or avoiding deadlocks in a computer system



เดสล็อกคืออะไร

- คือ การรอต่อเนื่องเป็นวงกลม circular wait จะเกิดขึ้นเมื่อโพรเซสมากกว่า 2 ตัวขึ้นไป กำลังรอทรัพยากรที่อีกฝ่ายถือครองอยู่ เช่น โพรเซส a ถือครองทรัพยากร x และต้องการใช้ทรัพยากร y แต่โพรเซส b ถือครองทรัพยากร y และต้องการใช้ทรัพยากร x
- ซึ่งสภาพนี้จะทำให้เกิดการรอเป็นวง (รอกันไปรอกันมา) เกิดเป็นวงล็อก หรือที่เรียกว่าเด็สล็อกนั่นเอง

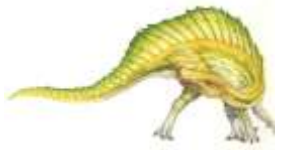



R_x
P_a
P_b
R_y

System Model

- System consists of resources
- Resource types $R_1, R_2, \ldots, R_m$

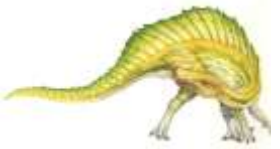
  CPU cycles, memory space, I/O devices
- Each resource type R_i has W_i instances.
- Each process utilizes a resource as follows:
  - **request** ร้องขอ
  - **use** ครอบครอง
  - **Release** ปล่อย



Deadlock Characterization

การเกิดเดสล็อกต้องมีสภาพทั้ง 4 อย่างนี้ครบ

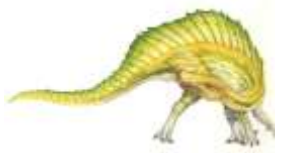
- **Mutual exclusion**: กีดกัน ทรัพยากรจะถูกถือครองได้ทีละ 1 โพรเซสเท่านั้น
- **Hold and wait**: ครองและรอพร้อมกัน โพรเซสสามารถใช้ทรัพยากรพร้อมกันได้มากกว่า 1 ตัว
- **No preemption**: ไม่มีการแทรก ทรัพยากรเมื่อถูกโพรเซสถือครอง จะไม่สามารถไปทำงานกับโพรเซสอื่น แล้วค่อยกลับมาทำงานค้างของโพรเซสเดิมได้
- **Circular wait**:**การรอคอยแบบวนรอบ แต่ละโพรเซสครอบครองทรัพยากรแล้วส่วนหนึ่ง แต่ร้องขอทรัพยากรเพิ่มจากโพรเซสอื่นที่ครอบครองทรัพยากรในลักษณะลูกโซ่**



Resource-Allocation Graph

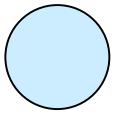
A set of vertices V and a set of edges E.

- 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 $P_i \rightarrow R_j$
- **assignment edge** – directed edge $R_j \rightarrow P_i$

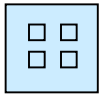


Resource-Allocation Graph (Cont.)

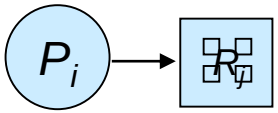
- สัญลักษณ์แทนโพรเซส



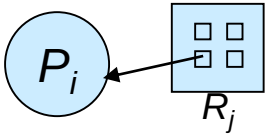
- สัญลักษณ์แทนทรัพยากร (จุดด้านใน 4 จุด แสดงว่าทรัพยากรสามารถถูกครอบครองโดย 4 โพรเซสพร้อมกันได้)



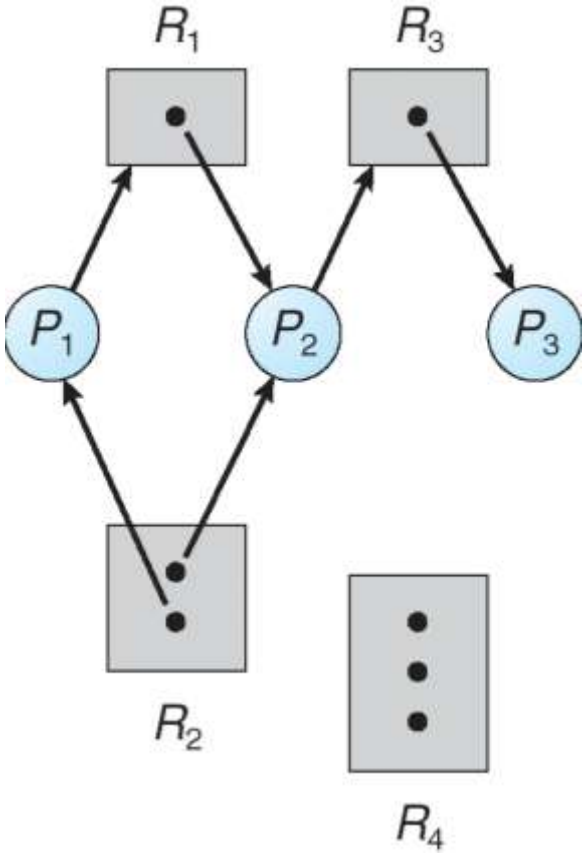
- หมายความว่า โพรเซส i ขอทรัพยากร j

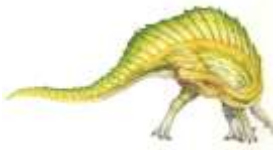


- หมายความว่า โพรเซส i ครอบครองทรัพยากร j อยู่ 1 ส่วน

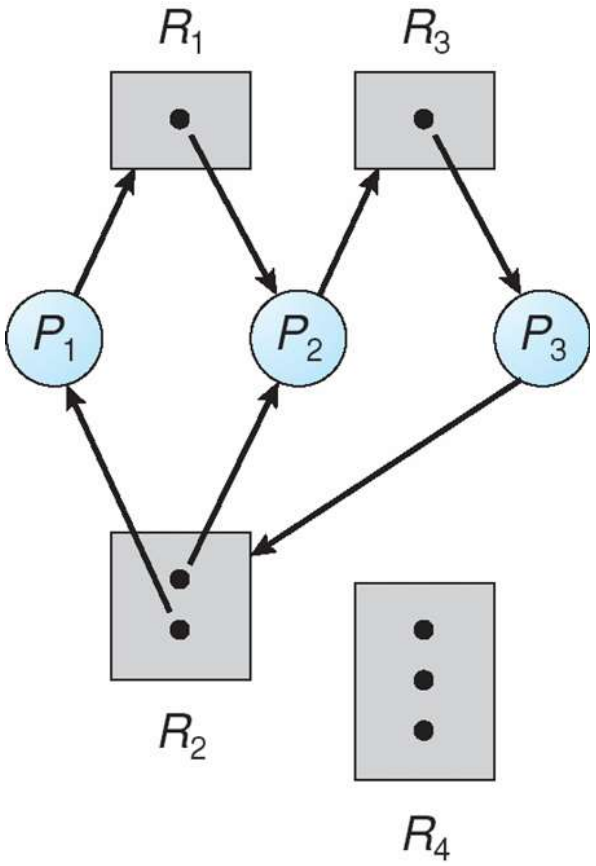


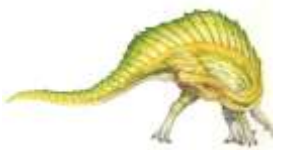
Example of a Resource Allocation Graph



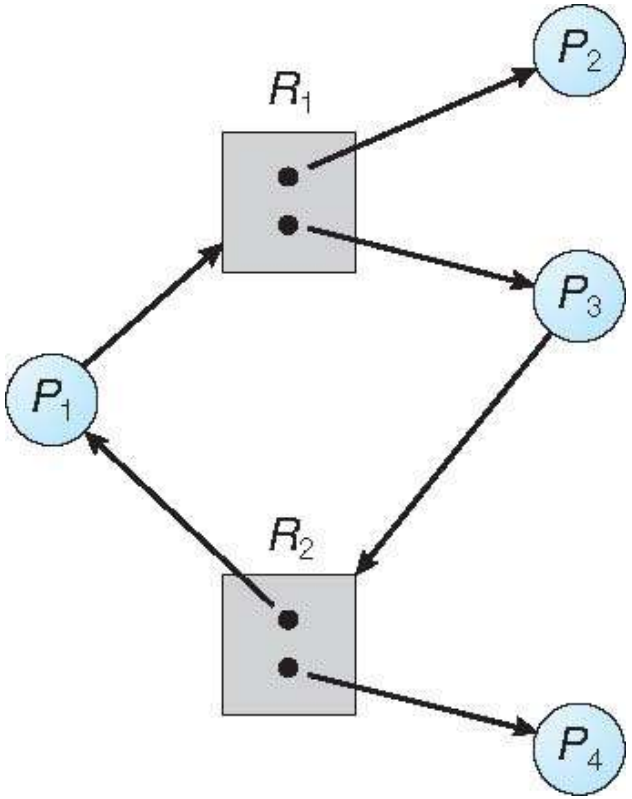


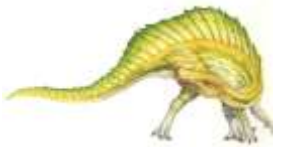
Resource Allocation Graph With A Deadlock





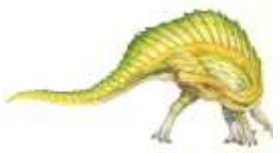
Graph With A Cycle But No Deadlock





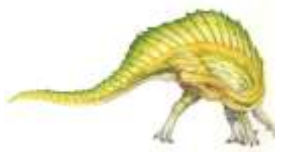
Basic Facts

- If graph contains no cycles $\Rightarrow$ no deadlock
- If graph contains a cycle $\Rightarrow$
  - if only one instance per resource type, then deadlock
  - if several instances per resource type, possibility of deadlock



Methods for Handling Deadlocks

- Ensure that the system will ***never*** enter a deadlock state:
  - Deadlock prevention
  - Deadlock avoidence
- Allow the system to enter a deadlock state and then recover
- Ignore the problem and pretend that deadlocks never occur in the system; used by most operating systems, including UNIX



การป้องกันการเกิดเดสล็อก

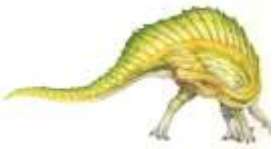
Deadlock Prevention

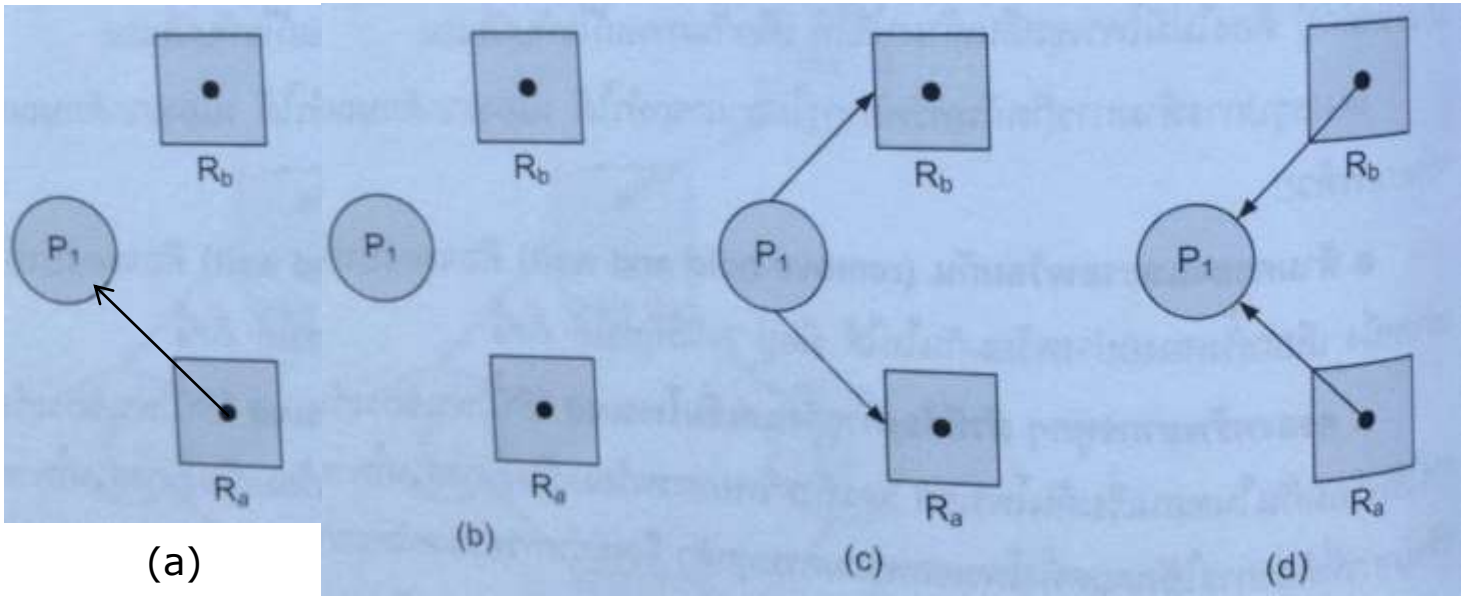
Restrain the ways request can be made

- **Mutual Exclusion –** แก้โดยห้ามกีดกันทรัพยากร แต่ทำจริงไม่ได้ เนื่องจากงานบางครั้งต้องมีการกีดกันทรัพยากรด้วย

  not required for sharable resources (e.g., read-only files);
  must hold for non-sharable resources

- **Hold and Wait –** แก้โดย ห้ามครองและรอพร้อมกัน
  - ครองทรัพยากรทุกตัวที่ต้องการใช้ในตอนเริ่มต้นโพรเซส ถ้าทรัพยากรพร้อมเพียงบางส่วนโพรเซสก็จะรอต่อไป จนกว่าทรัพยากรที่ต้องการใช้ตลอดทั้งโพรเซสครบทุกตัว จึงจะทำการครองทรัพยากรนั้นทั้งหมดและทำงาน.
  - ขอทรัพยากรเพิ่มได้ แต่ต้องคืนของเก่าก่อน





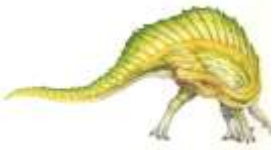
a) โพรเซสครองทรัพยากร Ra

b) โพรเซสต้องการทรัพยากร Rb เพิ่ม แต่ไม่ว่าง จึงต้องปล่อยทรัพยากร Ra

c) โพรเซสรอทรัพยากร Ra และ Rb (มีรอ ไม่มีครอง)

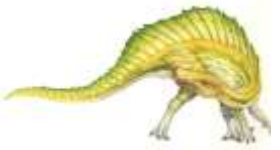
d) โพรเซสครองทรัพยากร Ra และ Rb (มีครอง ไม่มีรอ)

- การปิดโอกาส starvation :โพรเซสที่ใช้ทรัพยกรมากจะถูกละเลยไม่ถูกกระทำเลย จนกว่าจะมีทรัพยากรที่ต้องการทั้งหมดพร้อมในคราวเดียว ซึ่งโอกาสเช่นนั้นถึงจะมีแต่ก็ไม่บ่อยโพรเซสอาจจะถูกแช่แข็งไม่สามารถทำให้เสร็จได้ตลอดไป
- >>การห้ามเกิดการครองและการรอขึ้นพร้อมกันสามารถทำได้ แต่ต้องแลกกับข้อเสีย คือ ประสิทธิภาพของระบบจะสูญหายไป และทรัพยากรถูกใช้อย่างสิ้นเปลือง ไม่คุ้มค่า



Deadlock Prevention (Cont.)

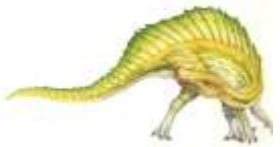
- **No Preemption –**แก้โดยทำให้มีการแทรกได้ ซึ่งถ้าเป็นซีพียูจะสามารถย้อนคืนสภาพก่อนการแทรกงานได้ หรือที่เรียกว่า **context switch** แต่ถ้าเป็นการแทรกงานในทรัพยากร ถึงแม้จะทำได้แต่ก็ไม่ง่ายเพราะต้องเตรียมพื้นที่
  ไว้สำหรับบันทึกสภาพก่อนการแทรกงานของแต่ละทรัพยากร เสี่ยงต่อการคำนวณผิดพลาด
  เพราะไม่ได้คำนวณรวดเร็ว
  มีการพักเนื่องจากมีการแทรกงาน
- Circular Wait – แก้โดยห้ามไม่ให้เกิดวงคอย โดยการจัดเรียงลำดับของทรัพยากร โดยเรียงยึดตามหลักการใช้งานจริง เลขลำดับต้องไม่ซ้ำ และมีหลักสำคัญคือ ห้ามรอทรัพยากรลำดับที่ต่ำกว่าทรัพยากรที่ถือครองอยู่

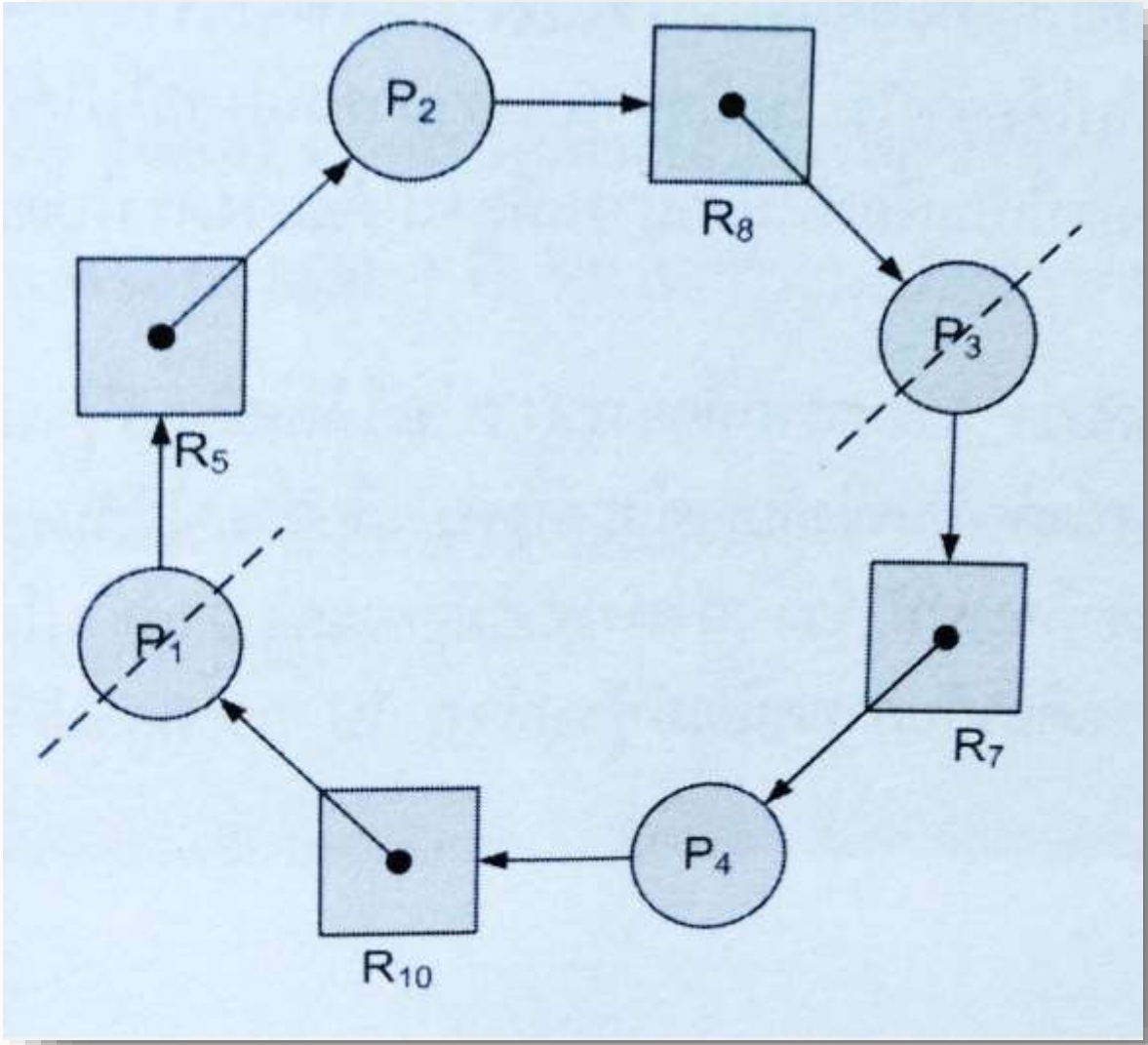


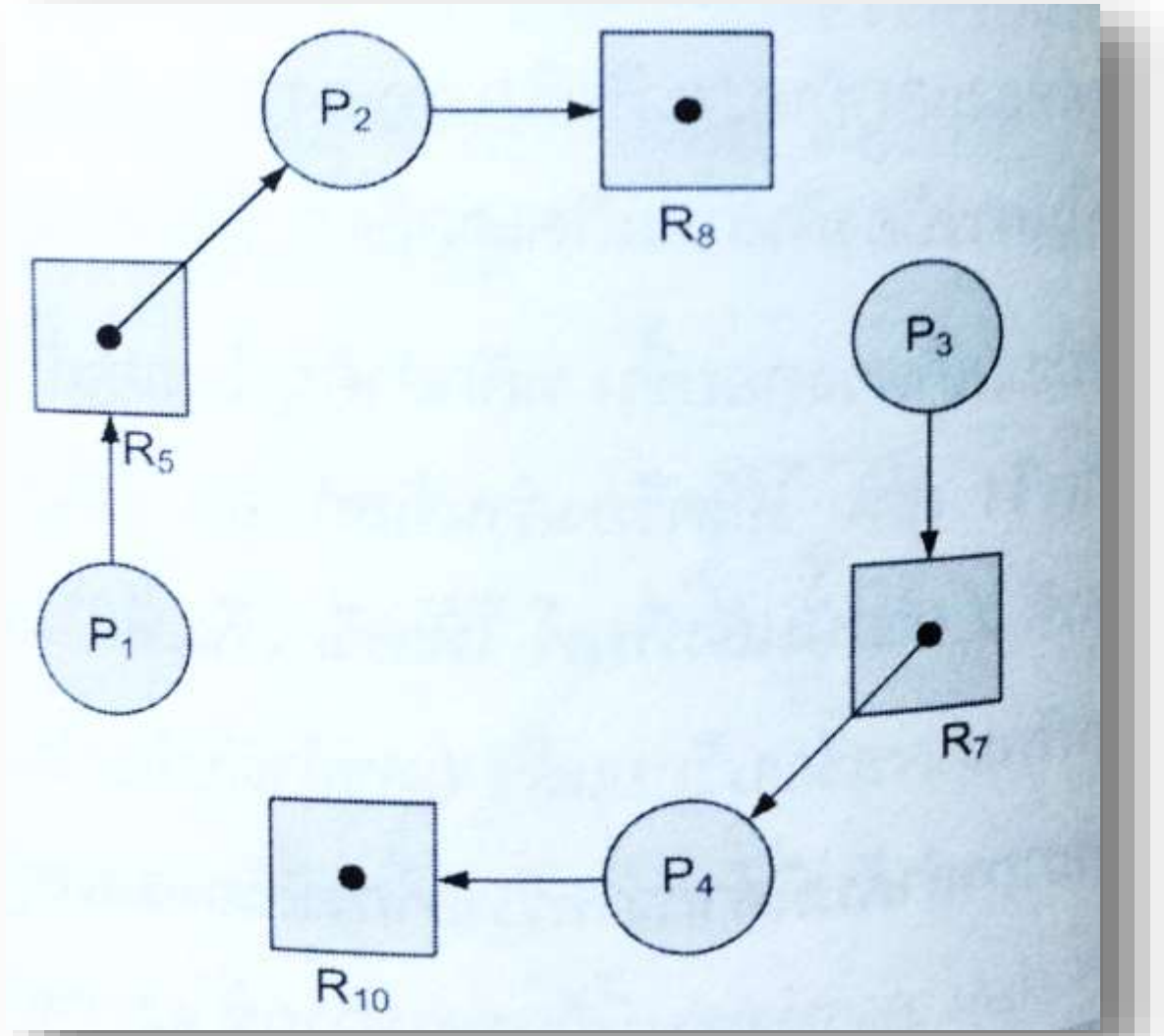
Deadlock Avoidance

Requires that the system has some additional ***a priori*** 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

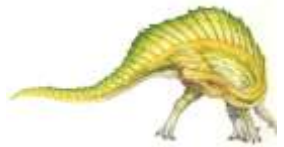



P2
R8
P3
R5
P1
R7
P4
R10


P2
R8
P3
R5
P1
R7
P4
R10

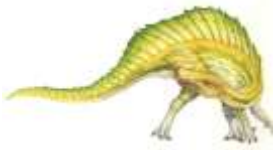
Safe State

- When a process requests an available resource, system must decide if immediate allocation leaves the system in a safe state
- System is in **safe state** if there exists a sequence $<P_1, P_2, \ldots, P_n>$ of ALL the processes in the systems such that for each P_i, the resources that P_i can still request can be satisfied by currently available resources + resources held by all the P_j, with $j < I$
- That is:
  - 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

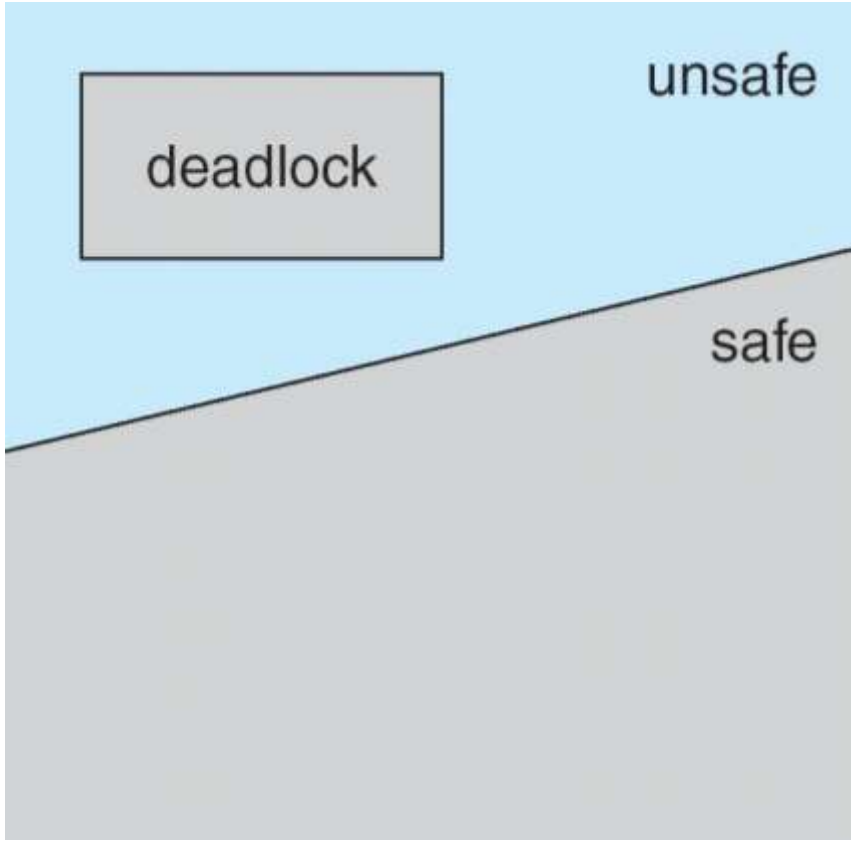


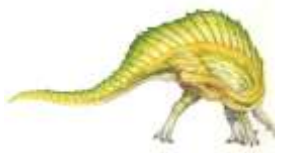
Basic Facts

- 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.



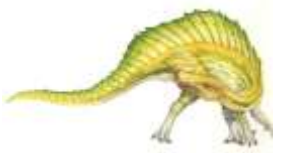
Safe, Unsafe, Deadlock State



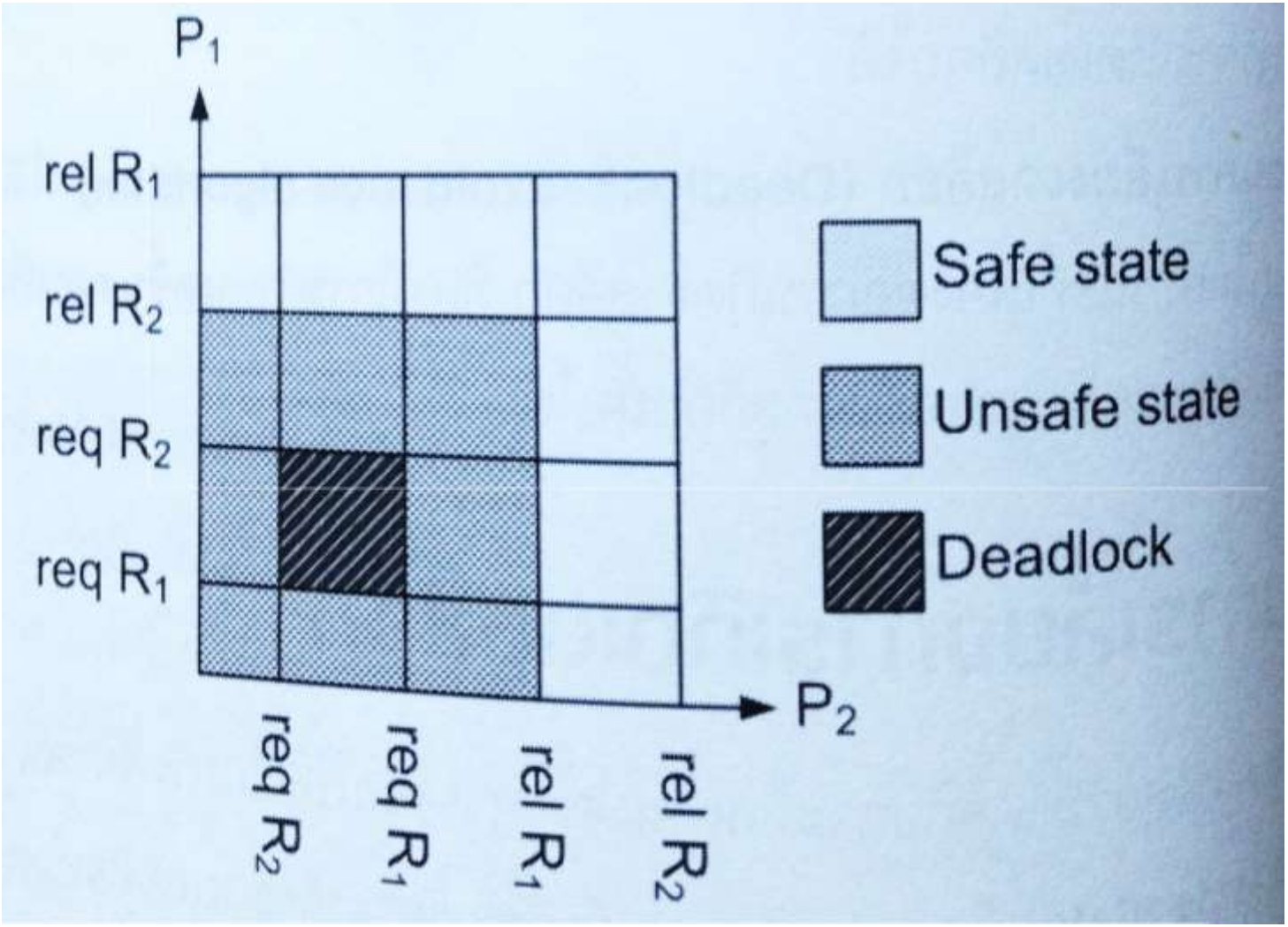


Avoidance Algorithms

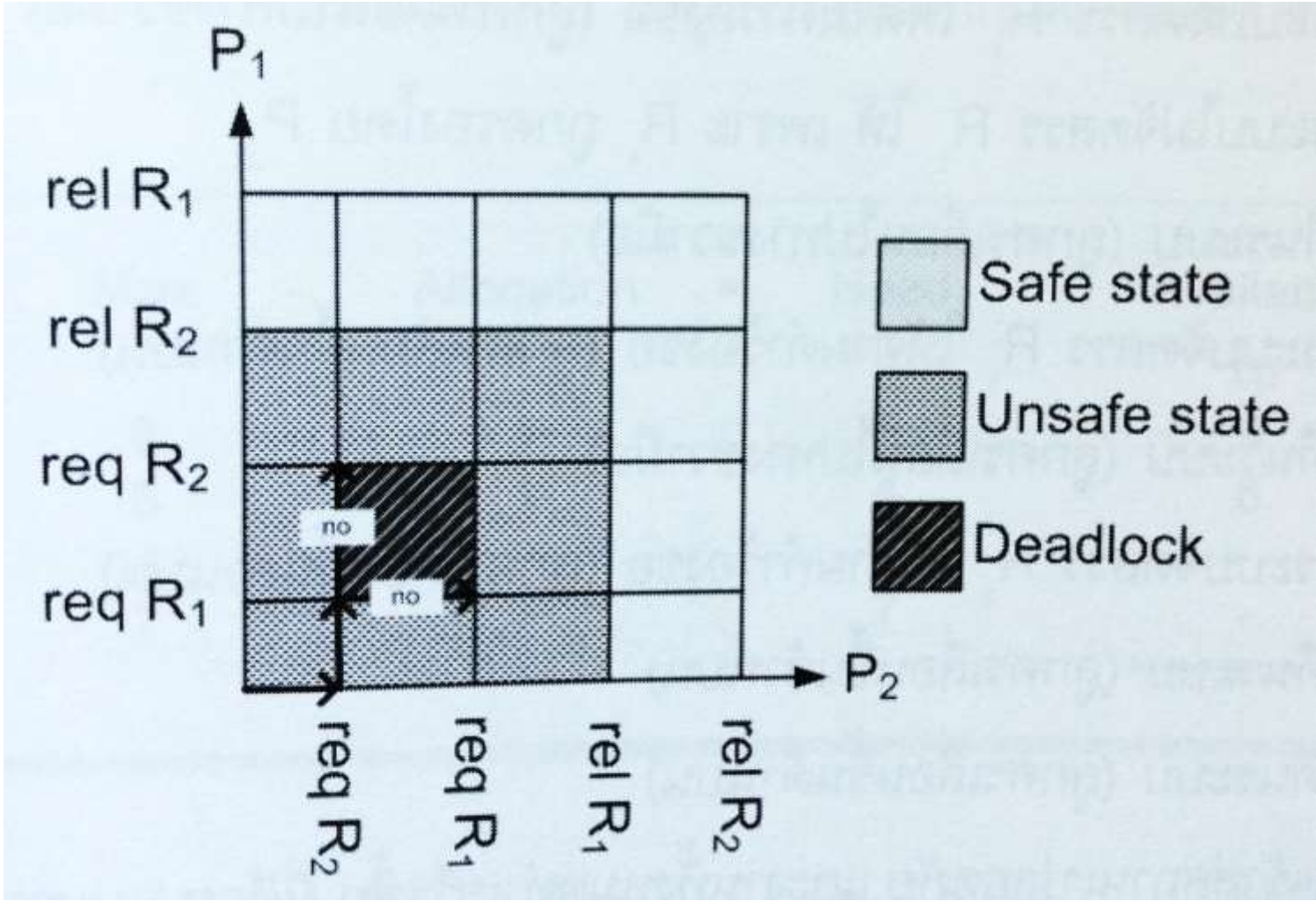
- Single instance of a resource type
  - Use a resource-allocation graph
- Multiple instances of a resource type
  - Use the banker's algorithm

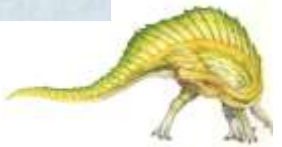


สถานะการจัดสรรทรัพยากร

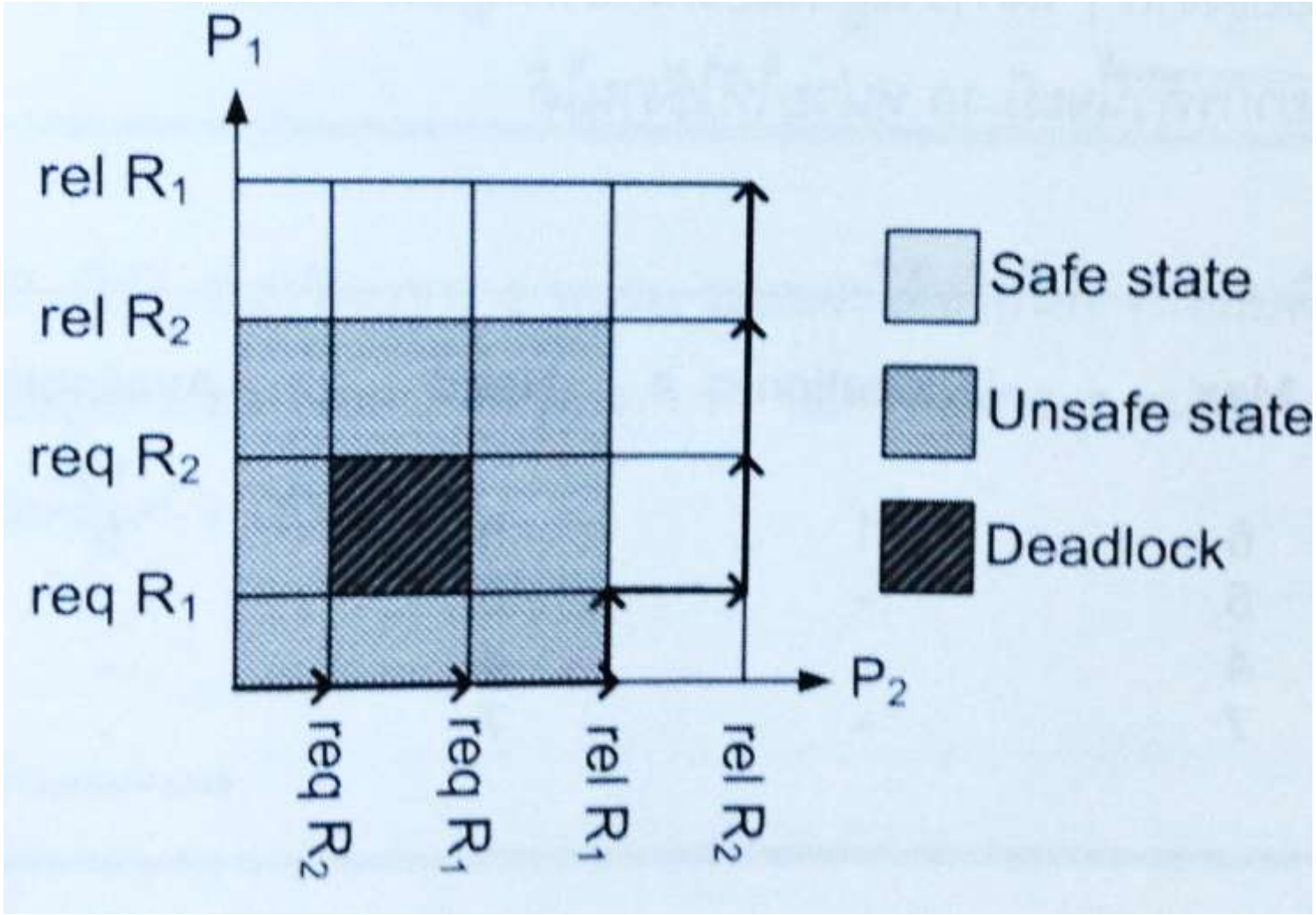


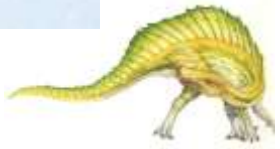
การพาระบบเข้าสู่สถานะเดสล็อก





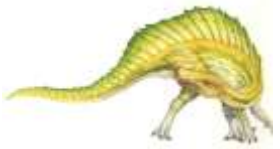
การพาระบบเลี่ยงสถานะเดสล็อก





Banker's Algorithm

- วีธีการเลี่ยงแบบนายธนาคาร โดยนำหลักการจากธนาคารในการสำรองจ่ายหมุนเวียน
  ให้เพียงพอตลอดเวลา ป้องกันการการจัดสรรเงินไปใช้ในช่องทางอื่น จนเหลือไม่พอให้ลูกค้าเบิกไปใช้จ่ายได้ วิธีการของแบงก์เกอร์ ก็คือ "การสำรองทรัพยากรให้เพียงพอสำหรับความต้องการที่ต่ำที่สุดเสมอ"



Data Structures for the Banker's Algorithm

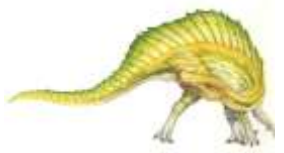
- **Available**: ทรัพยากรเริ่มต้นมีอยู่
  Vector of length m. If available $[j] = k$, there are k instances of resource type R_j available

- **Max**: โพรเซสต้องการทรัพยากรทั้งหมด
  $n \times m$ matrix. If $Max[i,j] = k$, then process P_i may request at most k instances of resource type R_j

- **Allocation**: โพรเซสใช้งานทรัพยากรจริง
  $n \times m$ matrix. If $Allocation[i,j] = k$ then P_i is currently allocated k instances of R_j

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

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



Example of Banker's Algorithm

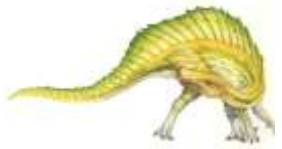
- 5 processes P_0 through P_4;

  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	

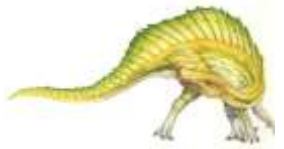


Example (Cont.)

- The content of the matrix ***Need*** is defined to be ***Max – Allocation***

	Need
	A B C
P_0	7 4 3
P_1	1 2 2
P_2	6 0 0
P_3	0 1 1
P_4	4 3 1

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

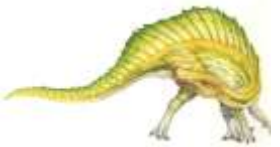


Example: P_1 Request (1,0,2)

- Check that Request ≤ Available (that is, (1,0,2) ≤ (3,3,2) ⇒ true

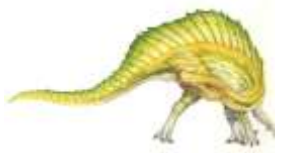
	Allocation	*Need*	*Available*
	A B C	*A B C*	*A B C*
P_0	0 1 0	7 4 3	2 3 0
P_1	3 0 2	0 2 0	
P_2	3 0 2	6 0 0	
P_3	2 1 1	0 1 1	
P_4	0 0 2	4 3 1	

- Executing safety algorithm shows that sequence < $\boldsymbol{P_1}$, $\boldsymbol{P_3}$, $\boldsymbol{P_4}$, $\boldsymbol{P_0}$, $\boldsymbol{P_2}$> satisfies safety requirement
- Can request for (3,3,0) by $\boldsymbol{P_4}$ be granted?
- Can request for (0,2,0) by $\boldsymbol{P_0}$ be granted?



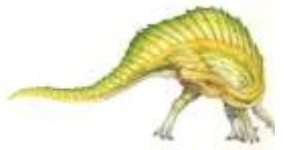
Deadlock Detection

- Allow system to enter deadlock state
- Detection algorithm
- Recovery scheme



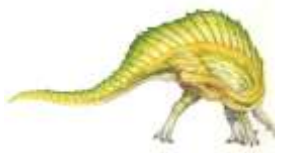
การตรวจหาเดสล็อก

- คือ การแจ้งเตือนให้ระบบทราบเมื่อเกิดเดสล็อกขึ้น
- หลักในการตรวจหาเดสล็อก
- 1. เมื่อเดสล็อกเกิดระบบต้องทราบทันที
- 2. ทำการแก้ไขเดสล็อก เพื่อทำให้ระบบสามารถทำงานต่อไปได้
- ตัวอย่างของการตรวจหาเดสล็อก “ถ้าความต้องการของทุกโพรเซสมีมากกว่าทรัพยากรที่เหลือเดสล็อกเกิดขึ้นแน่นอน”



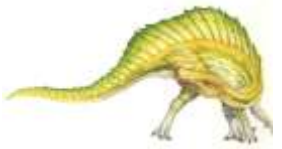
การแก้ไขปัญหาเดสล็อก

- 1. การยกเลิกโพรเซส
- 2. การแทรกกลางคัน



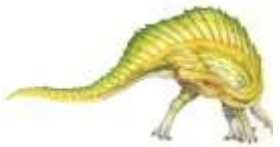
การยกเลิกโพรเซส

- คือการจบโพรเซสนั้น และคืนทรัพยากรให้ระบบ เพื่อนำไปใช้กับโพรเซสอื่นต่อไปมีอยู่ 2 แนวทาง
- 1. ยกเลิกทุกโพรเซสที่อยู่ในสถานะเดสล็อก
- 2. ยกเลิกบางโพรเซสที่อยู่ในสถานะเดสล็อก



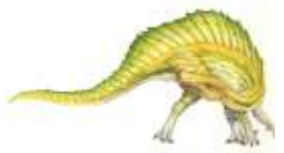
การแทรกกลางคัน

- คือการเอาทรัพยากรบางส่วนของโพรเซสที่ติดเดสล็อก เอาไปให้โพรเซสอื่นทำงานก่อน จนกว่าเดสล็อกจะหลุด มี 3 วิธี
- 1. เลือกผู้เสียสละ
- 2. ย้อนกลับ
- 3. ขังลืม

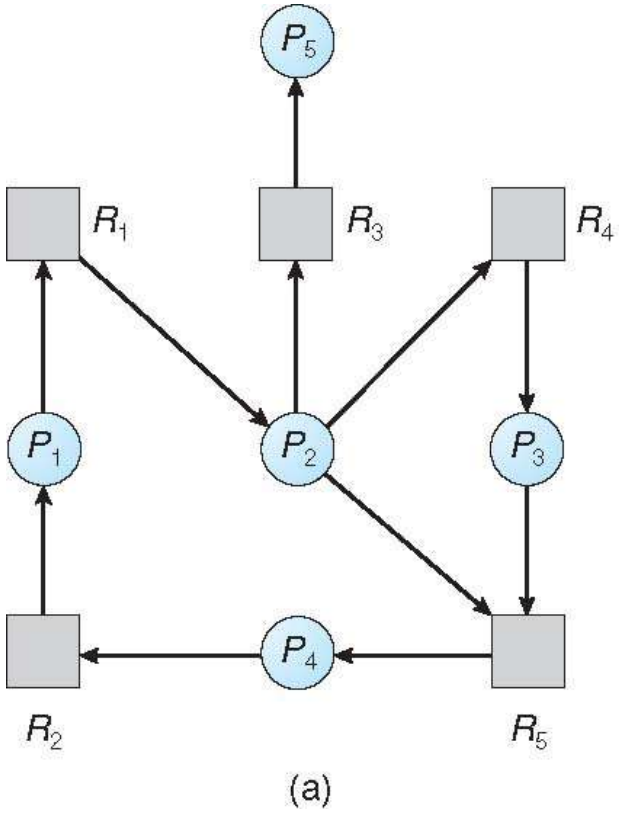


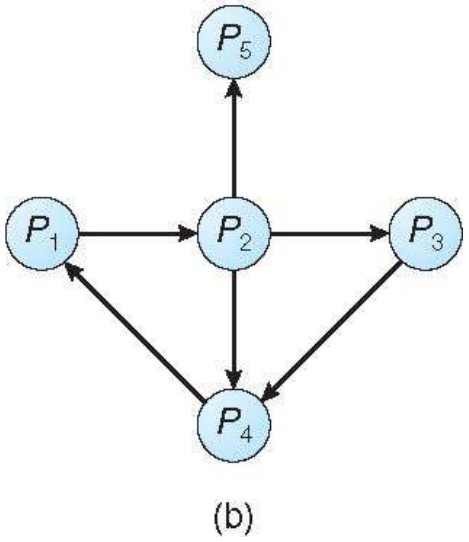
Single Instance of Each Resource Type

- Maintain **wait-for** graph
  - 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. If there is a cycle, there exists a deadlock
- 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



Resource-Allocation Graph and Wait-for Graph



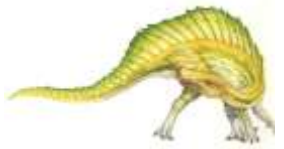


Resource-Allocation Graph

Corresponding wait-for graph

Several Instances of a Resource Type

- **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 R_j.



Detection Algorithm

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

   (a) ***Work = Available***

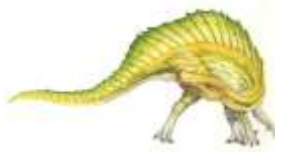
   (b) For ***i* = 1,2, …, *n***, if ***Allocation*$_i$ ≠ 0**, then ***Finish*[i] = *false***; otherwise, ***Finish*[i] = *true***

2. Find an index ***i*** such that both:

   (a) ***Finish*[*i*] == *false***

   (b) ***Request*$_i$ ≤ *Work***

   If no such ***i*** exists, go to step 4

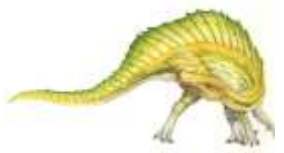


Detection Algorithm (Cont.)

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



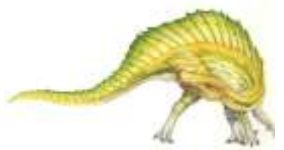
Example of Detection Algorithm

- 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***

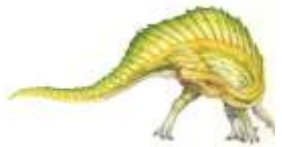


Example (Cont.)

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

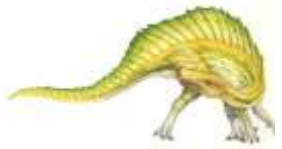
	Request
	A B C
P_0	0 0 0
P_1	2 0 2
P_2	0 0 1
P_3	1 0 0
P_4	0 0 2

- 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



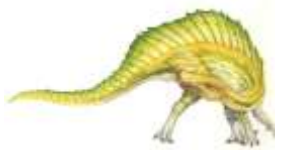
Detection-Algorithm Usage

- 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?
    - one for each disjoint cycle
- 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.



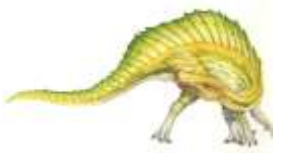
Recovery from Deadlock: Process Termination

- Abort all deadlocked processes
- Abort one process at a time until the deadlock cycle is eliminated
- In which order should we choose to abort?
  1. Priority of the process
  2. How long process has computed, and how much longer to completion
  3. Resources the process has used
  4. Resources process needs to complete
  5. How many processes will need to be terminated
  6. Is process interactive or batch?



Recovery from Deadlock: Resource Preemption

- **Selecting a victim** – minimize cost
- **Rollback** – 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



End of Chapter 7