

# **Chapter 7: Deadlocks**

肖卿俊

办公室：九龙湖校区计算机楼212室

电邮：[csqjxiao@seu.edu.cn](mailto:csqjxiao@seu.edu.cn)

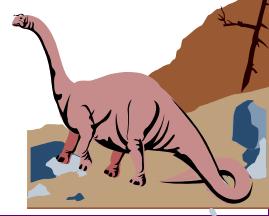
主页：<https://csqjxiao.github.io/PersonalPage>

电话：025-52091022



# Chapter 7: Deadlocks

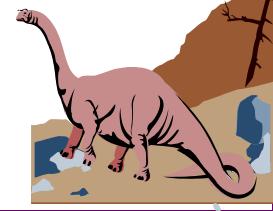
- System Model
- Deadlock Characterization
- Methods for Handling Deadlocks
- Deadlock Prevention (死锁预防)
- Deadlock Avoidance (死锁避免)
- Deadlock Detection (死锁检测)
- Recovery from Deadlock (死锁恢复)





# What Is a Deadlock?

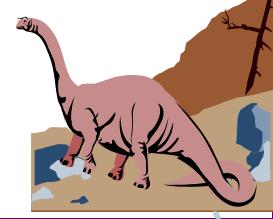
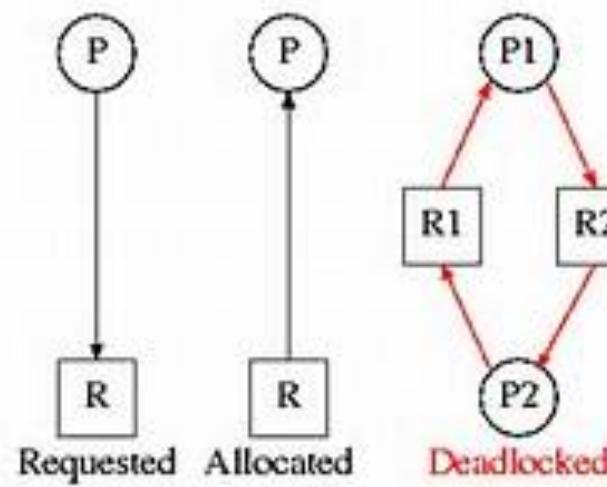
- Deadlock (死锁) is a special phenomenon of resource scarcity among a group of processes (or threads)
- A set of blocked processes each holding a resource and waiting to acquire a resource held by another process in the set.
- A Simple Example
  - ◆ System has 2 tape drives.
  - ◆  $P_1$  and  $P_2$  each hold one tape drive and each needs another one.





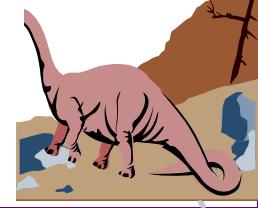
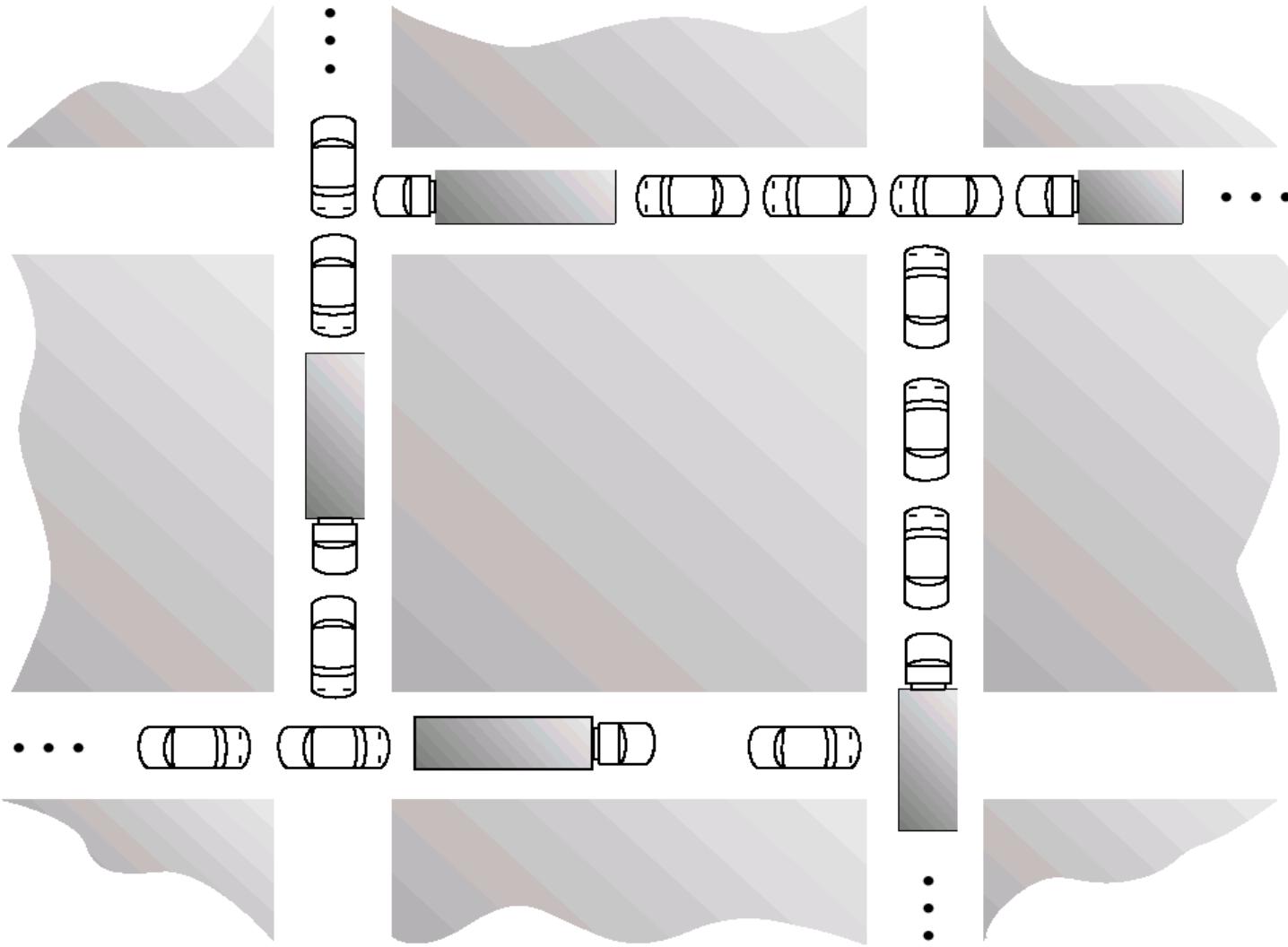
# Formalize the Simple Example of Deadlock

- A simple example of deadlock between two processes P1 and P2
  - ◆ P1 holds R1 and needs R2
  - ◆ P2 holds R2 and needs R1





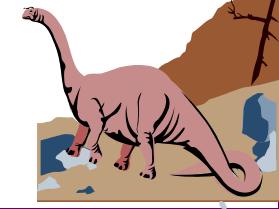
# Deadlock can be of a much larger scale





# System Model

- Resource types  $R_1, R_2, \dots, R_m$   
*CPU cycles, memory space, I/O devices, etc.*
- Each resource type  $R_i$  has  $W_i$  instances.
- Each process utilizes a resource as follows:
  - ◆ Request
  - ◆ Use
  - ◆ Release





# System Model

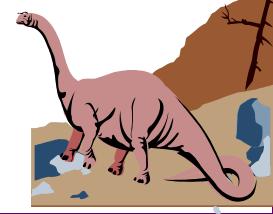
- System resources are used in the following way:
  - ◆ **Request:** If a process makes a request to use a system resource which cannot be granted immediately, then the requesting process must block until it can acquire the resource.
  - ◆ **Use:** The process can operate on the resource.
  - ◆ **Release:** The process releases the resource.
- **Deadlock:** A set of processes is in a deadlock state when every process in the set is waiting for an event that can only be caused by another process in the set.





# Chapter 7: Deadlocks

- System Model
- Deadlock Characterization
- Methods for Handling Deadlocks
- Deadlock Prevention (死锁预防)
- Deadlock Avoidance (死锁避免)
- Deadlock Detection (死锁检测)
- Recovery from Deadlock (死锁恢复)

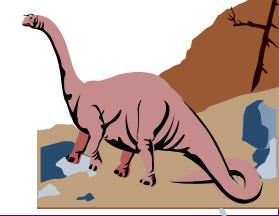




# Four Necessary Conditions for a Deadlock Situation

■ ***For a deadlock to occur, each of the following four conditions must hold.***

- ◆ **Mutual exclusion:** only one process at a time can use a resource.
- ◆ **Hold and wait:** A process must be holding a resource and waiting for another.
- ◆ **No preemption:** A resource can be released only voluntarily by the process holding it, after that process has completed its task.
- ◆ **Circular wait:** A waits for B, B waits for C, C waits for A.

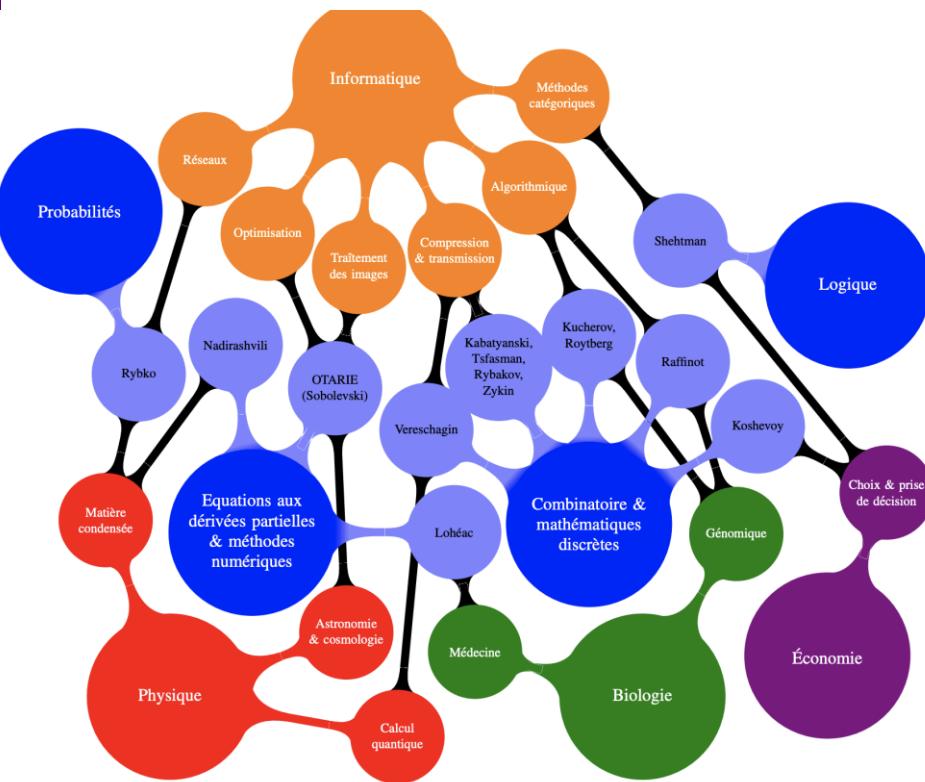




# Many real-world phenomenon can be modelled as a graph

## ■ Social Networks

### ◆ Scientific interaction



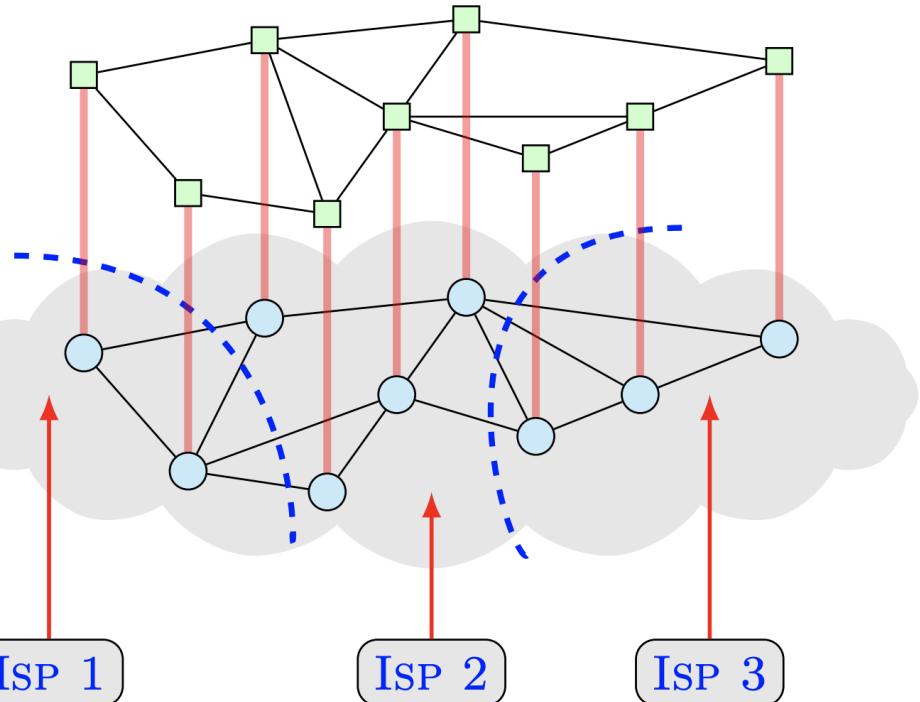
Operating System Concepts

7

## ■ Communication Ntwk

### ◆ P2P system

P2P SYSTEM TOPOLOGY

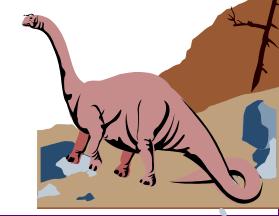




# Model process resource request & allocation relations as a graph

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

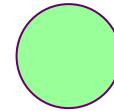
- Vertices  $V$  is partitioned into two types:
  - ◆  $P = \{P_1, P_2, \dots, P_n\}$ , the set consisting of all the processes in the system.
  - ◆  $R = \{R_1, R_2, \dots, R_m\}$ , the set consisting of all resource types in the system.
- Edges  $E$  is also partitioned into two types:
  - ◆ Resource request edge
    - directed edge  $P_i \rightarrow R_j$
  - ◆ Resource assignment edge
    - directed edge  $R_j \rightarrow P_i$



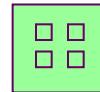


# Resource-Allocation Graph (Cont.)

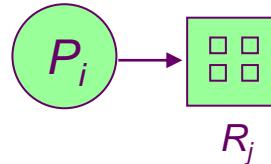
- Process



- Resource Type with 4 instances

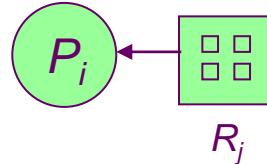


- $P_i$  requests instance of  $R_j$



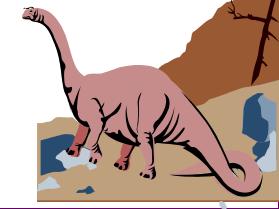
How to draw a directed graph using latex tikz?  
<http://www.texample.net/tikz/examples/graph/>

- $P_i$  is holding instance of  $R_j$



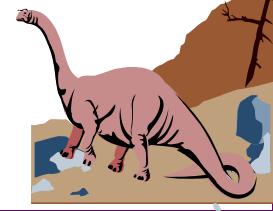
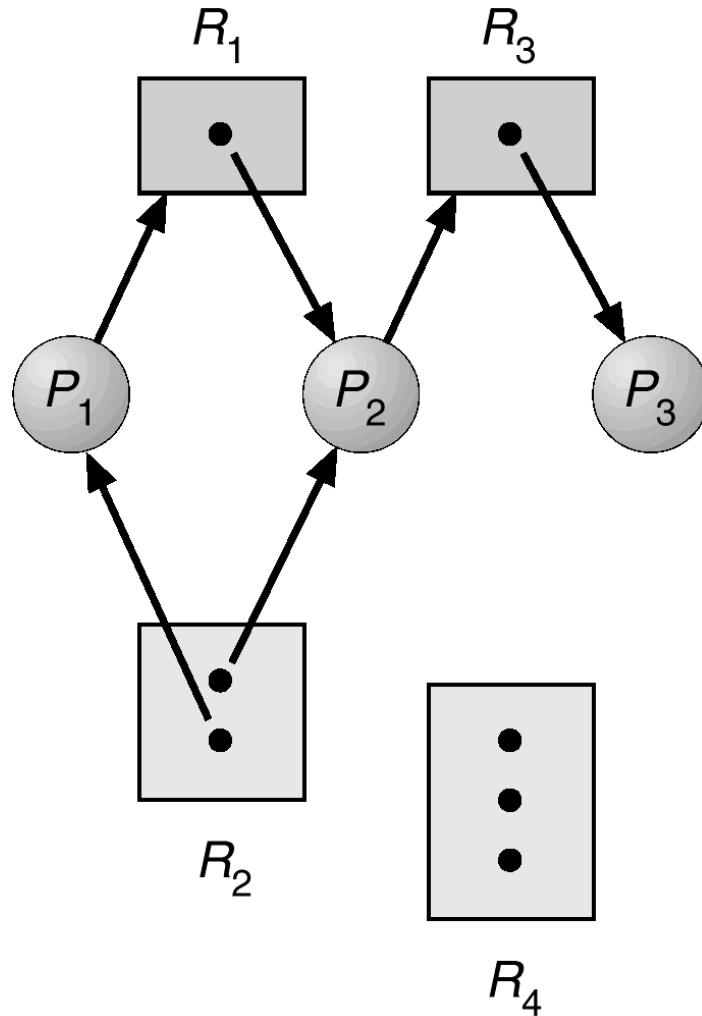
- There can be weights on directed edges

- ◆  $P_i$  requests/holds  $w_{ij}$  instances of  $R_j$



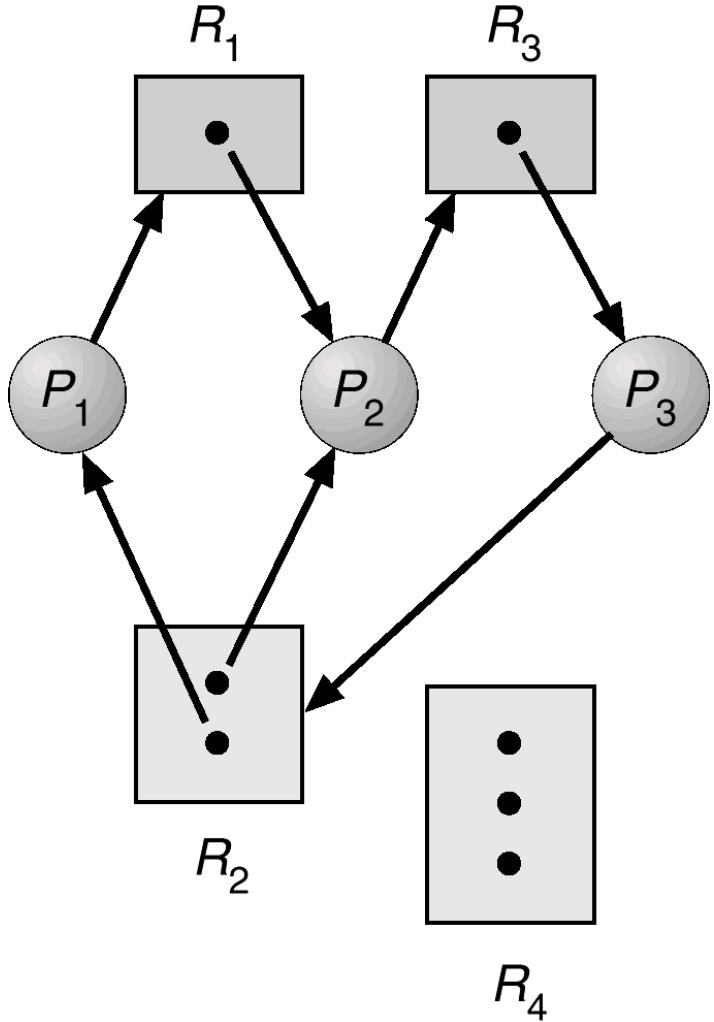


# A Simple Example of Resource Allocation Graph





# Resource Allocation Graph With A Deadlock

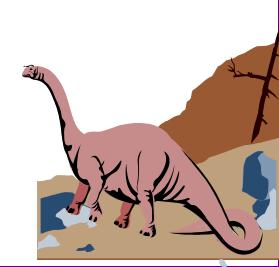


Problem: **Detection of a Cycle in an Directed Graph.**

**Two Methods** to solve this

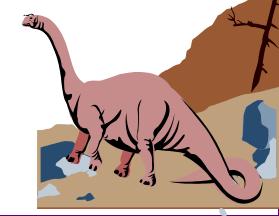
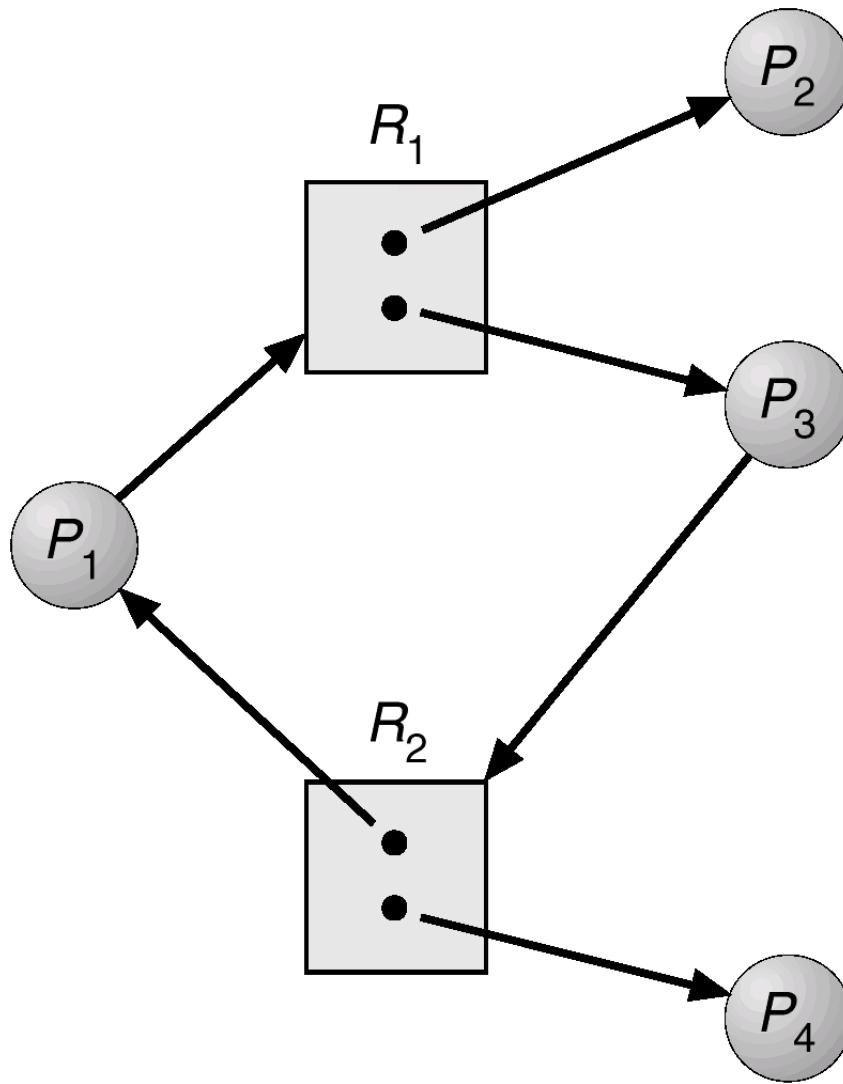
- DFS: <https://www.geeksforgeeks.org/detect-cycle-in-a-graph/>
- Kahn's Algorithm (BFS):  
<https://www.geeksforgeeks.org/detect-cycle-in-a-directed-graph-using-bfs/>

**Time Complexity:**  $O(V + E)$ ,





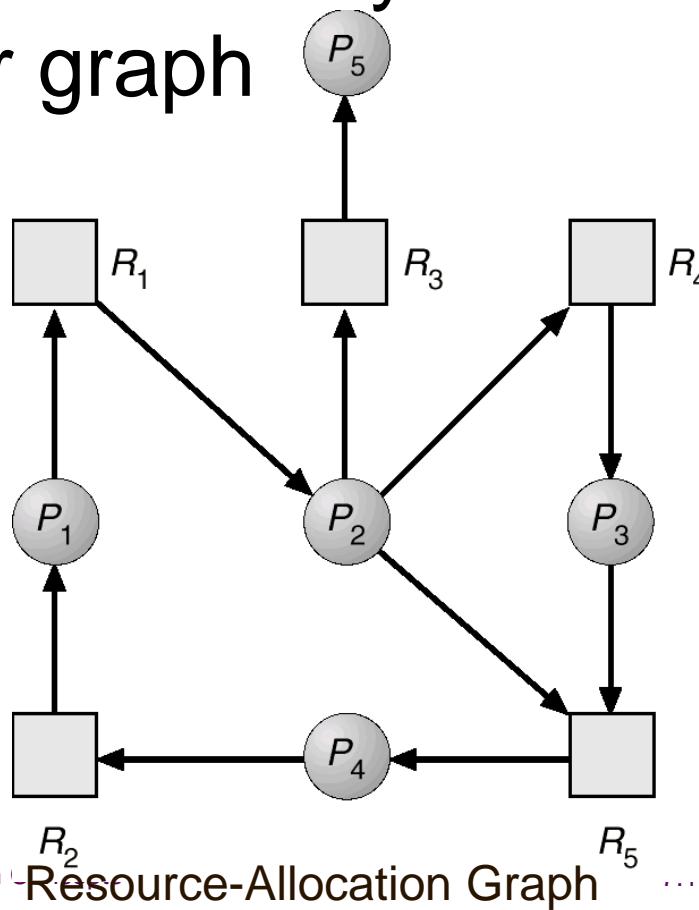
# Resource Allocation Graph With A Cycle But No Deadlock



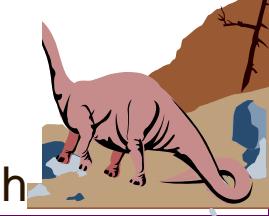
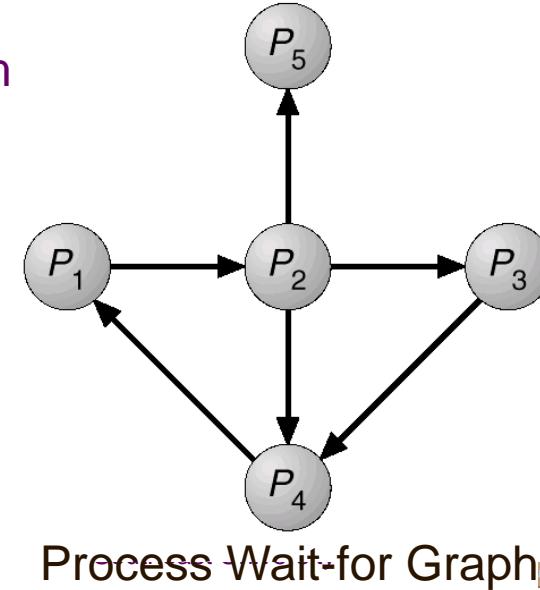


# Resource-Allocation Graph and Wait-for Graph (1)

- When there is only one instance per resource type, a resource allocation graph can be easily transformed to a process-wait-for graph



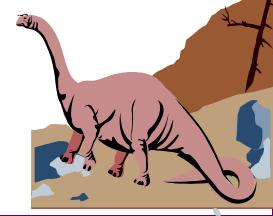
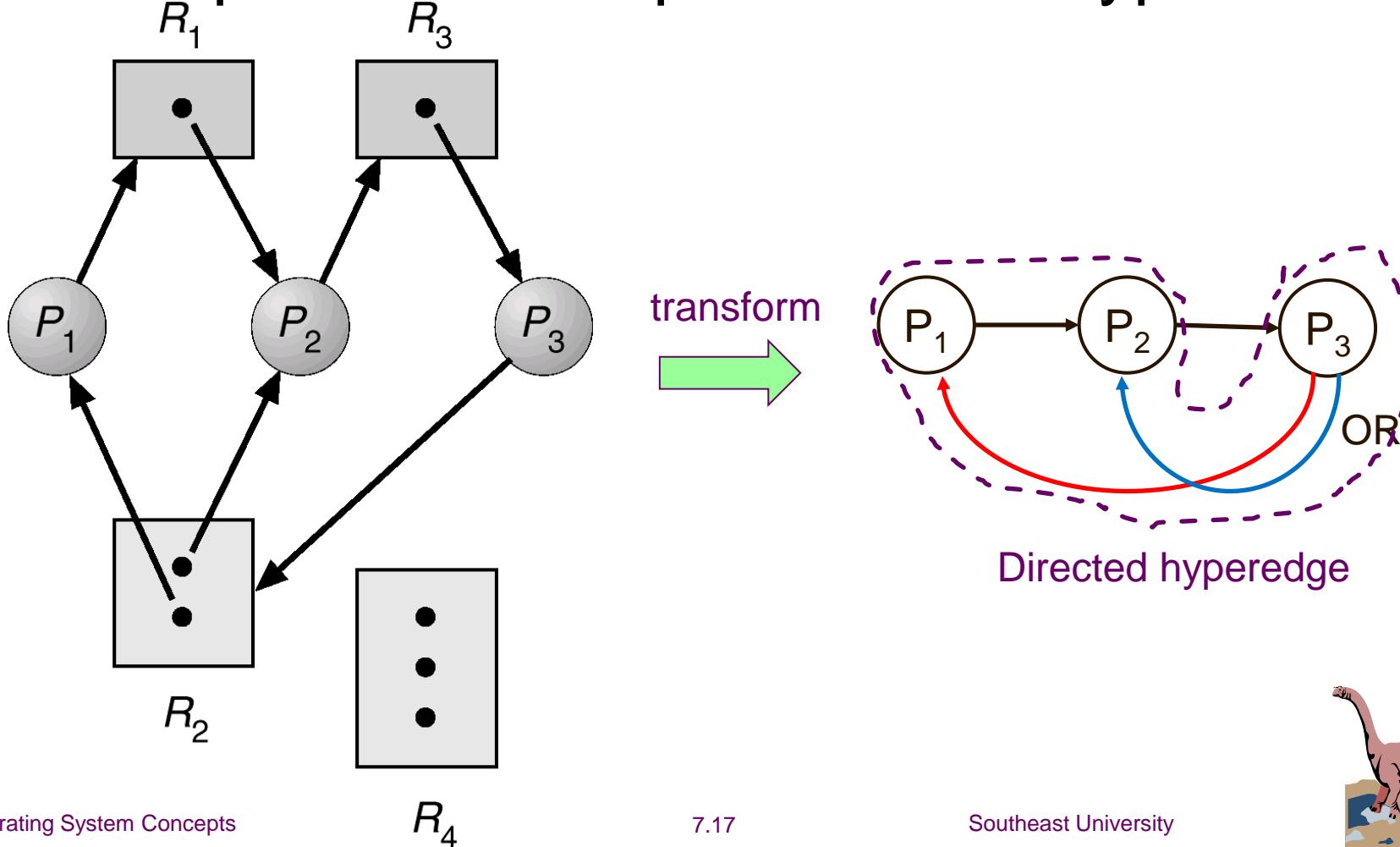
transform  
→





# Resource-Allocation Graph and Wait-for Graph (2)

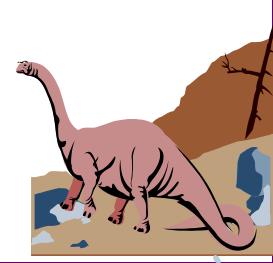
- The transformation is difficult, when there are multiple instances per resource type





# Basic Facts about Deadlock Detection

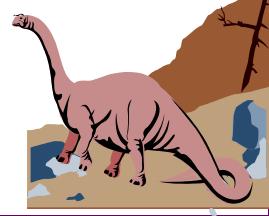
- If a resource allocation graph contains no cycles  $\Rightarrow$  no deadlock.
- If a resource allocation graph contains a cycle  $\Rightarrow$ 
  - ◆ if only one instance per resource type, then deadlock.
  - ◆ if several instances per resource type, possibility of deadlock.





# Chapter 7: Deadlocks

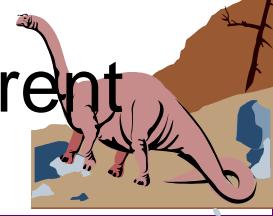
- System Model
- Deadlock Characterization
- **Methods for Handling Deadlocks**
- Deadlock Prevention (死锁预防)
- Deadlock Avoidance (死锁避免)
- Deadlock Detection (死锁检测)
- Recovery from Deadlock (死锁恢复)





# Methods for Handling Deadlocks

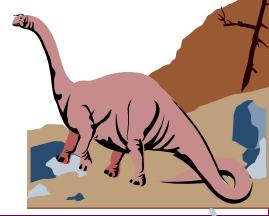
1. **Ignore** the problem and pretend that deadlocks never occur in the system.
2. **Allow** the system to enter a deadlock state, (事后) **detect** it, and recover from it, typically by killing the processes that hold popular resources
3. (事先) Ensure that the system will **never** enter a deadlock state.
  - ◆ **Prevention** (预防) : Ensure one of the four necessary conditions fails.
  - ◆ **Avoidance** (避免) : The OS needs more information so that it can determine if the current request can be satisfied or delayed.





# Chapter 7: Deadlocks

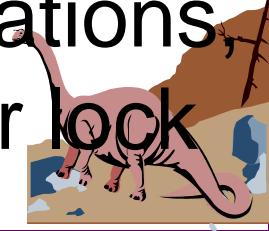
- System Model
- Deadlock Characterization
- Methods for Handling Deadlocks
- Deadlock Prevention (死锁预防)
- Deadlock Avoidance (死锁避免)
- Deadlock Detection (死锁检测)
- Recovery from Deadlock (死锁恢复)





# Deadlock Prevention (死锁预防): Mutual Exclusion

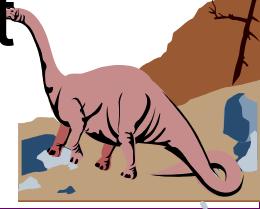
- By ensuring that at least one of the four conditions cannot hold, we can prevent the occurrence of a deadlock.
- **Mutual Exclusion:** Some sharable resources must be accessed exclusively (e.g., printer), which means we cannot deny the mutual exclusion condition.
- Some OS mechanisms may bring inspirations, e.g., CPU time sharing and reader-writer lock





# Deadlock Prevention (死锁预防): Hold and Wait

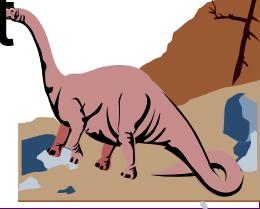
- Strictly forbid a process to hold some resources and then request for other resources.
- Two strategies are possible:
  - ◆ A process must acquire *all* resources before it runs.
  - ◆ When a process requests for resources, it must hold none (*i.e.*, return all resources before requesting for more).
- **Resource utilization** may be low, since many resources will be held and unused for long time
- **Starvation** is possible. A process that needs some popular resources may have to wait indefinitely.





# Deadlock Prevention (死锁预防): Hold and Wait

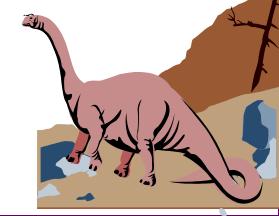
- Strictly forbid a process to hold some resources and then request for other resources.
- Two strategies are possible:
  - ◆ A process must acquire *all* resources before it runs.
  - ◆ We can wrap the taking of left chopstick and the taking of right chopstick as an atomic operation. WHY?
- Resources will be held and unused for long time
- Starvation is possible. A process that needs some popular resources may have to wait indefinitely.





# Deadlock Prevention (死锁预防): No Preemption

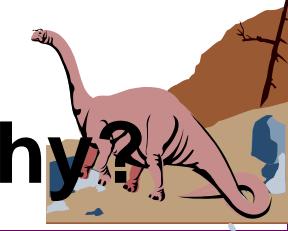
- If a process that is holding some resources requests another resource that cannot be immediately allocated to it, then **all resources currently being held are released**.
- If the requested resources are not available:
  - ◆ If they are being held by processes that are waiting for additional resources, these resources are preempted and given to the requesting process.
  - ◆ Otherwise, the requesting process waits until the requested resources become available. While it is waiting, its resources may be preempted.





# Deadlock Prevention (死锁预防): Circular Wait

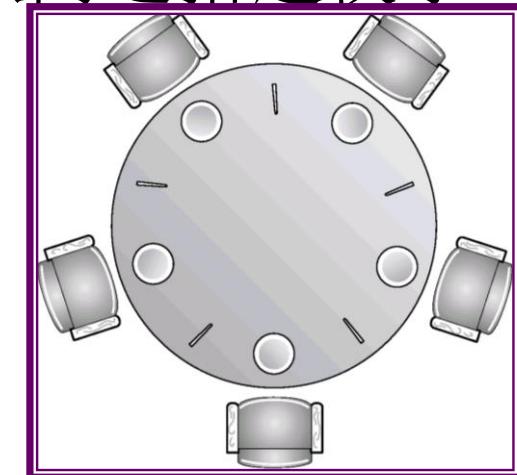
- To break the circular waiting condition, we order all resource types (e.g., tapes, printers)
- A process can only request resources higher than the resource types it holds.
- Suppose the ordering of tapes, disks, and printers are 1, 4, and 8. If a process holds a disk (4), it can only ask a printer (8) and cannot request a tape (1). A process must release some higher order resources to request a lower order resource. To get tapes (1), a process must release its disk (4).
- In this way, there will be no cycle. Why?





# 哲学家问题的死锁预防

- 假设有5个哲学家，共享一张放有五把椅子的桌子，每人分得一把椅子。
- 桌子上总共只有5支筷子，在每人两边分开各放一支
- 哲学家们在肚子饥饿时才试图分两次从两边拾起筷子就餐。
- 条件：
  1. 只有拿到两支筷子时，哲学家才能吃饭。
  2. 如果筷子已在他人手上，则该哲学家必须等待到他人吃完之后才能拿到筷子。
  3. 任一哲学家在自己未拿到两支筷子吃饭之前，决不放下自己手中的筷子。
- 试用信号量解决该哲学家用餐问题，要预防死锁问题





■ **解法1：**将抓左筷子和抓右筷子的动作捆绑成一个原子操作（只有当左右筷子都拿到时，才释放mutex）

```
semaphore mutex = 1 ;
```

```
semaphore chopstick[5]={1, 1, 1, 1, 1};
```

```
void philosopher(int i)
```

```
{
```

```
do {
```

```
    think();
```

```
wait(mutex);
```

```
wait(chopstick[(i+1)%5]);
```

```
wait(chopstick[i]);
```

```
signal(mutex);
```

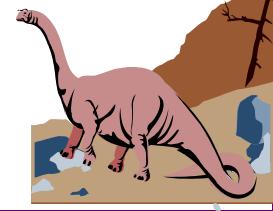
```
eat();
```

```
signal(chopstick[(i+1)%5]);
```

```
signal(chopstick[i]);
```

```
} while(true);
```

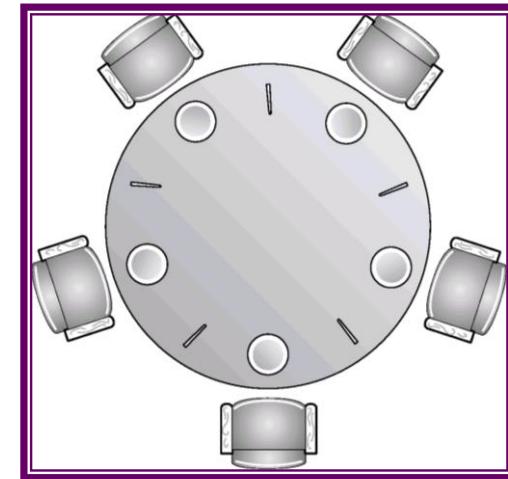
```
}
```





# A Quiz

■ 该解法有瑕疵。一个好的解法应该允许不相邻的没有资源冲突的哲学家同时进餐。



◆ 比如，哲学家P1和哲学家P3座位不相邻，那么他们就不共用任何筷子。如果P1和P3都处于饥饿状态，好的解法应当允许P1和P3同时进餐。

■ 请设想如何才能出现下面的场景 --- 桌面上只有一位哲学家（比如P1）正在进餐，同时另一位非邻座的哲学家进程P3尽管饥饿，但是被阻塞无法进餐，除非哲学家P1结束进餐。



■ 解法2：最多允许四个哲学家同时进餐，以保证至少有一个哲学家能够拿起两只筷子，最终总会释放出他使用的两支筷子，从而可使更多的哲学家进餐。

```
semaphore chopstick[5]={1, 1, 1, 1, 1};
```

```
semaphore room=4;
```

```
void philosopher(int i)
```

```
{
```

```
do {
```

```
    think();
```

```
    wait(room); //请求进入房间进餐
```

```
    wait(chopstick[i]); //请求左手边的筷子
```

```
    wait(chopstick[(i+1)%5]); //请求右手边的筷子
```

```
    eat();
```

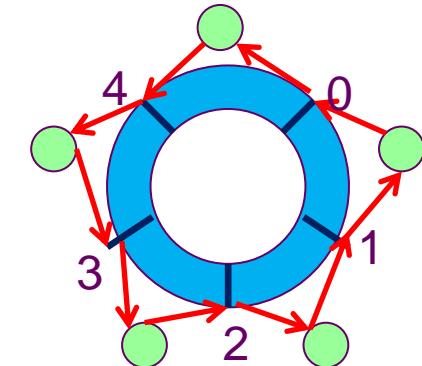
```
    signal(chopstick[(i+1)%5]); //释放右手边的筷子
```

```
    signal(chopstick[i]); //释放左手边的筷子
```

```
    signal(room); //退出房间释放信号量room
```

```
} while(true);
```

每个人都要**wait(room)**和**signal(room)**，有一定性能损失



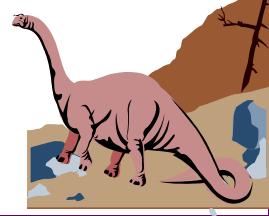


```
semaphore chopstick[5] = {1, 1, 1, 1, 1};  
void philosopher(int i) {  
    do {  
        think();  
        if(i%2 == 0) { // 偶数哲学家，先右后左。  
            wait (chopstick[(i+1)%5]);  
            wait (chopstick[i]);  
            eat();  
            signal (chopstick[(i+1)%5]);  
            signal (chopstick[i]);  
        } else { // 奇数哲学家，先左后右。  
            wait (chopstick[i]);  
            wait (chopstick[(i+1)%5]);  
            eat();  
            signal (chopstick[i]);  
            signal (chopstick[(i+1)%5]);  
        }  
    } while(true);  
}
```



# Chapter 7: Deadlocks

- System Model
- Deadlock Characterization
- Methods for Handling Deadlocks
- Deadlock Prevention (死锁预防)
- Deadlock Avoidance (死锁避免)
- Deadlock Detection (死锁检测)
- Recovery from Deadlock (死锁恢复)





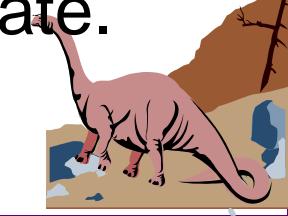
# Deadlock Avoidance (死锁避免)

- When only one instance per resource type, deadlock-avoidance algorithm can examine the resource-allocation state dynamically to ensure that there will never be a circular-wait condition.
- Otherwise, simplest and most useful model requires that each process **declares the *maximum number of resources of each type* that it may need.** Requires that the system has some additional *a priori* information available.
- Resource-allocation **state** is defined by the number of available and allocated resources, and the maximum demands of the processes





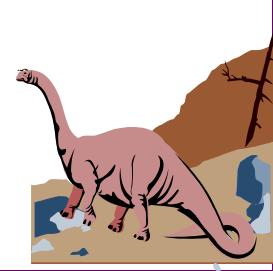
# Safe State

- 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 safe sequence** of all processes to run to the end.
  - Sequence  $\langle P_1, P_2, \dots, P_n \rangle$  is safe if 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  $j < i$ .
    - 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.
- 



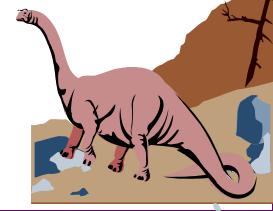
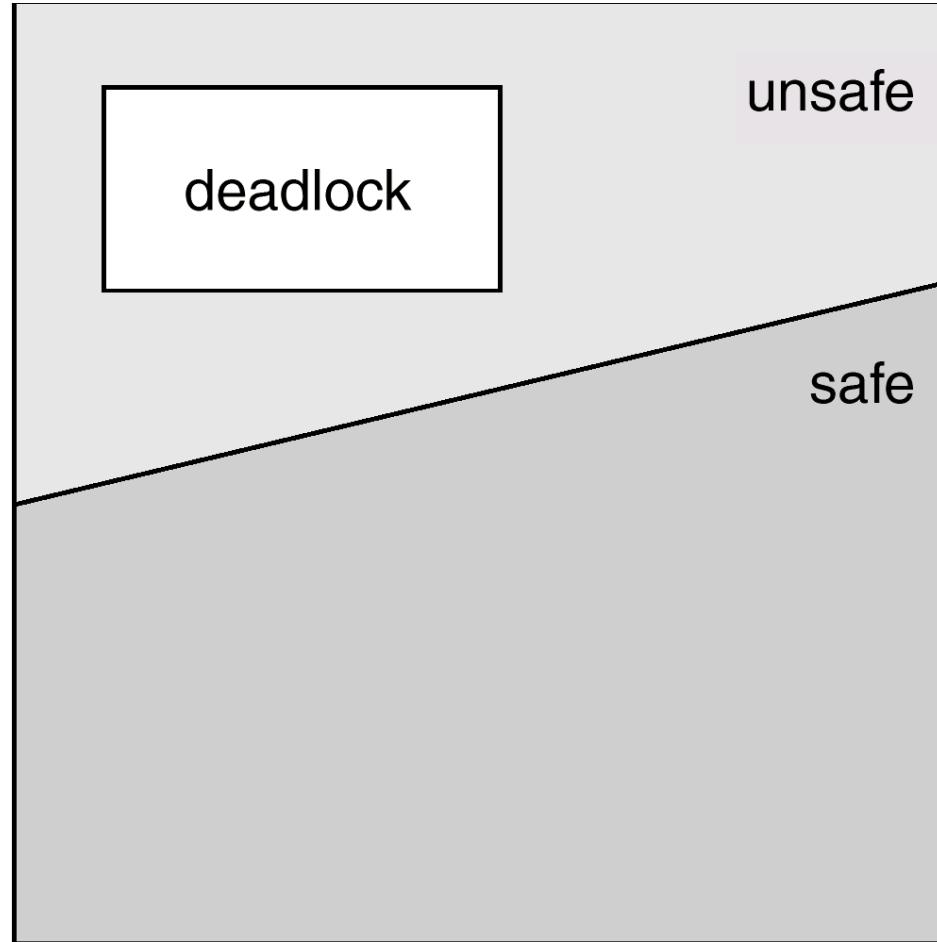
# Basic Facts

- If a system is in safe state  $\Rightarrow$  definitely not in deadlock states.
- If a system is in unsafe state  $\Rightarrow$  possibility of deadlock.
- Avoidance  $\Rightarrow$  ensure that a system will never enter an unsafe state.



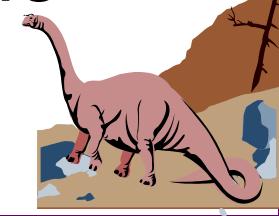


# Safe, Unsafe , Deadlock State



# Single Instance per Resource Type: Resource-Allocation Graph Algorithm

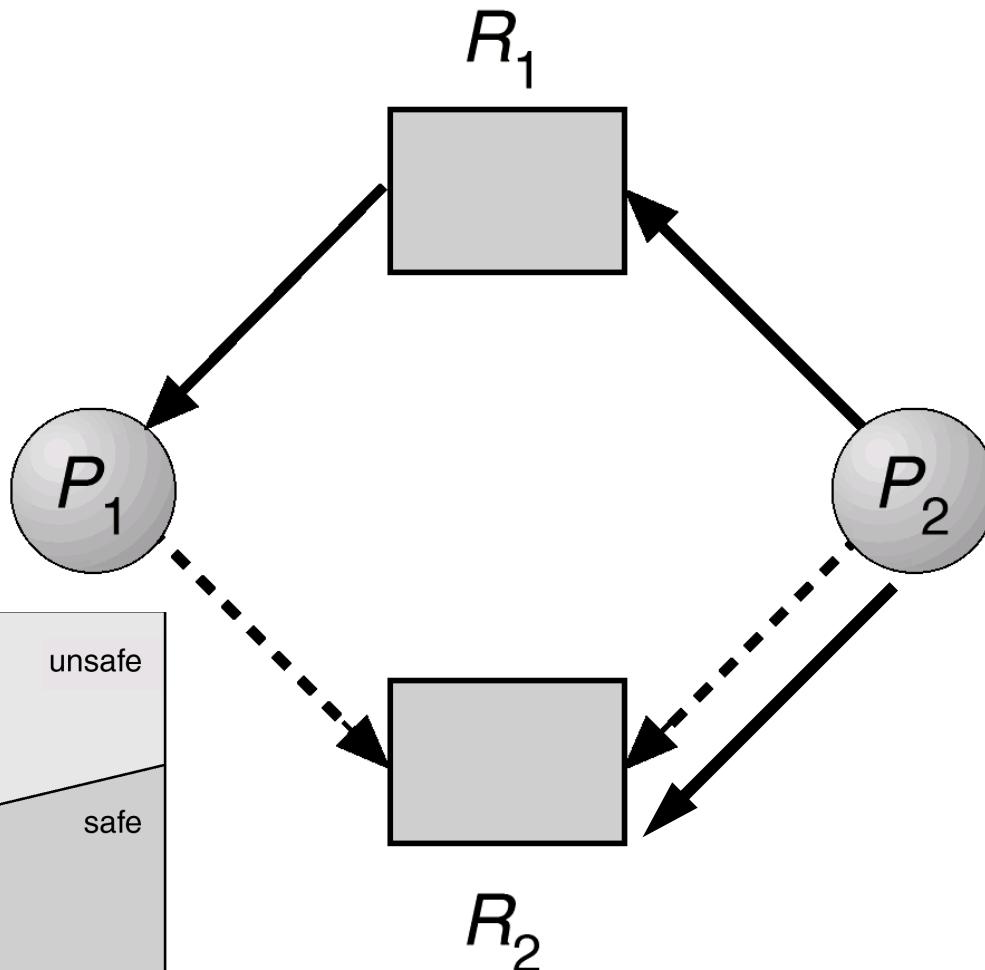
- *Claim edge*  $P_i \rightarrow R_j$  indicated that process  $P_i$  may request resource  $R_j$ ; represented by a dashed line.
- Claim edge converts to request edge when a process requests a resource.
- When a resource is released by a process, assignment edge reconverts to a claim edge.
- Resources must be claimed apriori in the system.



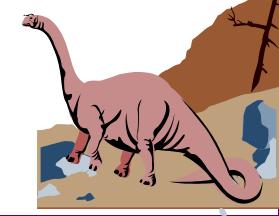


# Resource-Allocation Graph

## For Deadlock Avoidance: Example 1

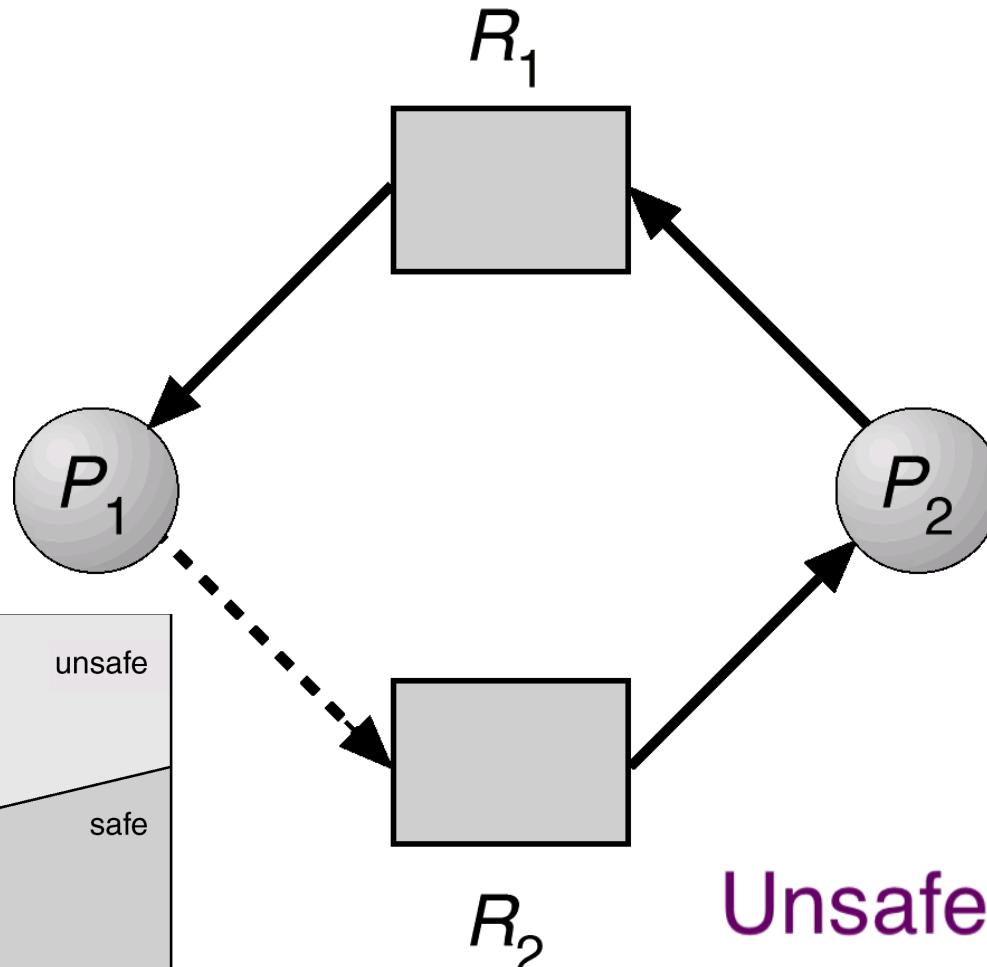


Deadlock?  
Safe?  
Unsafe?



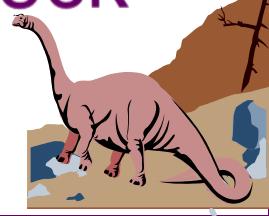


# Resource-Allocation Graph For Deadlock Avoidance: Example 2



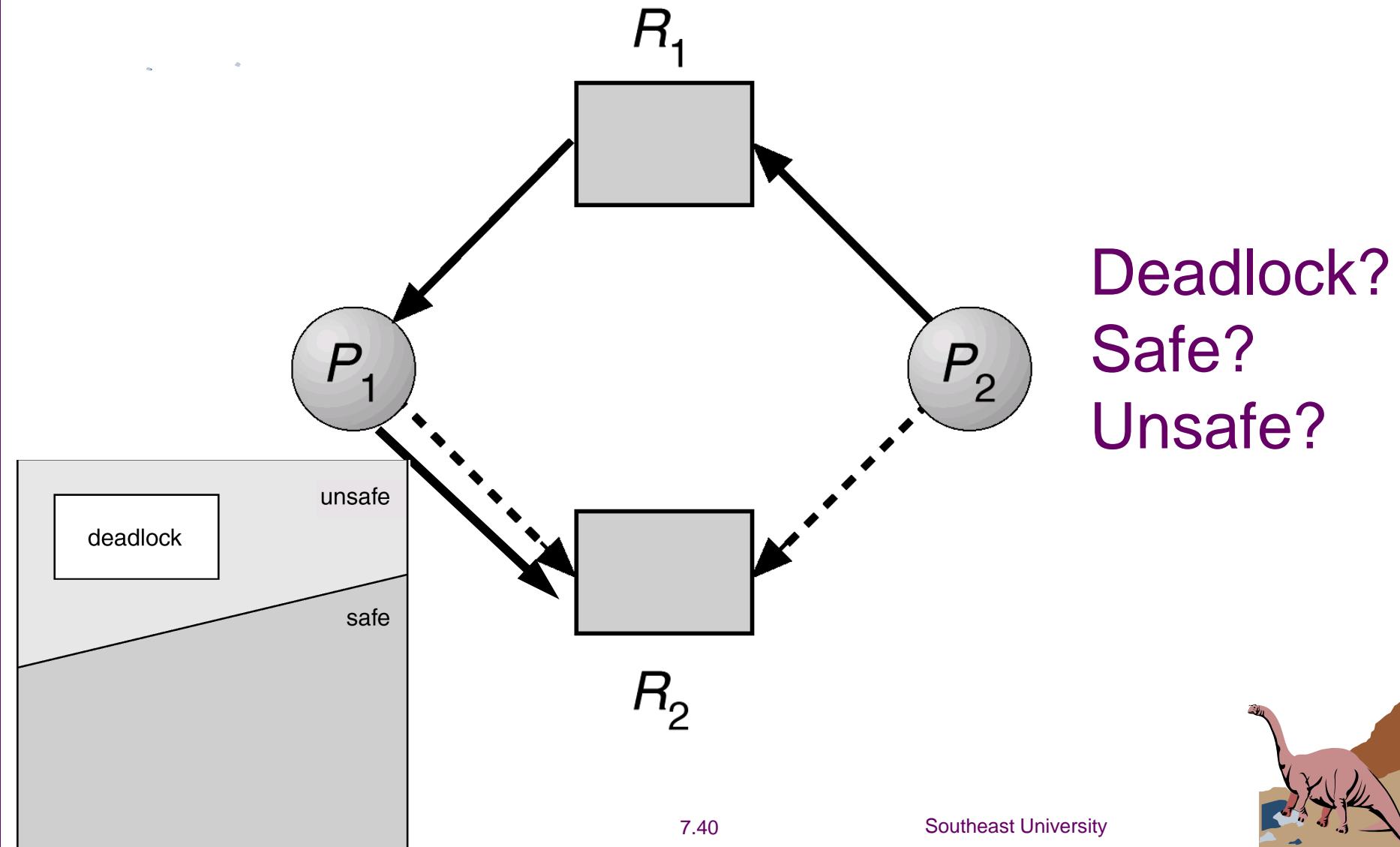
Deadlock?  
Safe?  
Unsafe?

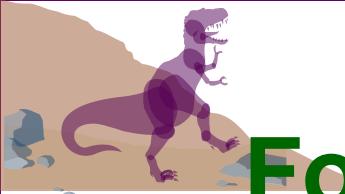
Unsafe ≠ Deadlock



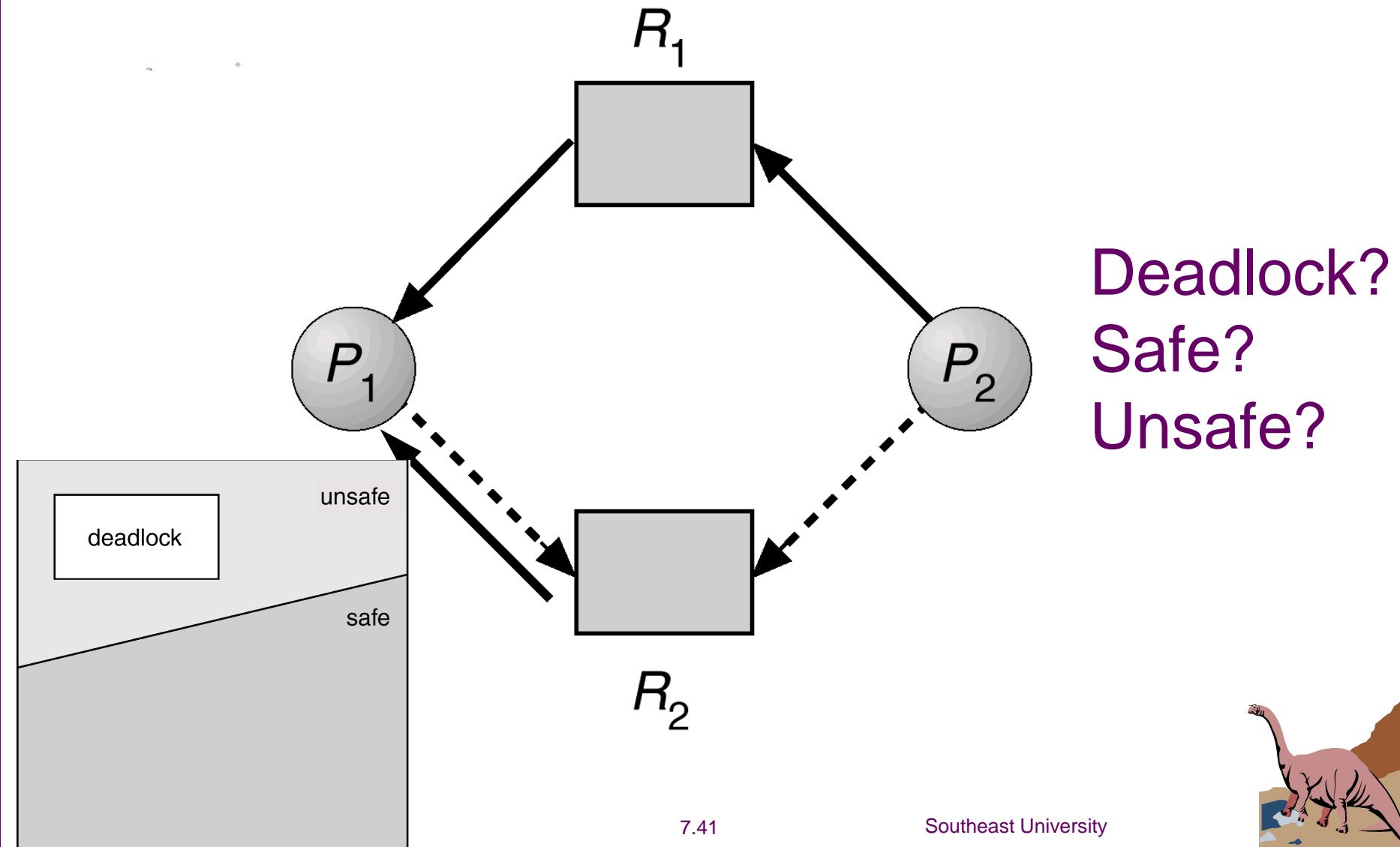


# Resource-Allocation Graph For Deadlock Avoidance: Example 3





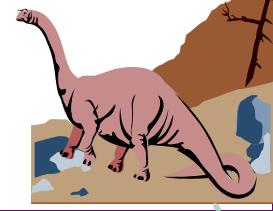
# Resource-Allocation Graph For Deadlock Avoidance: Example 3





# Multiple Instances per Resource Type: Banker's Algorithm

- Three Assumptions of Banker's Algorithm
  - ◆ Each process must apriori claim its maximum use of each resource type.
  - ◆ When a process requests for a particular amount of resources, it may have to wait, even if the system has the resources available.
  - ◆ When a process gets all its needed resources, it must return them in a finite amount of time.





# Data Structures for the Banker's Algorithm

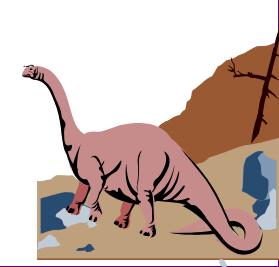
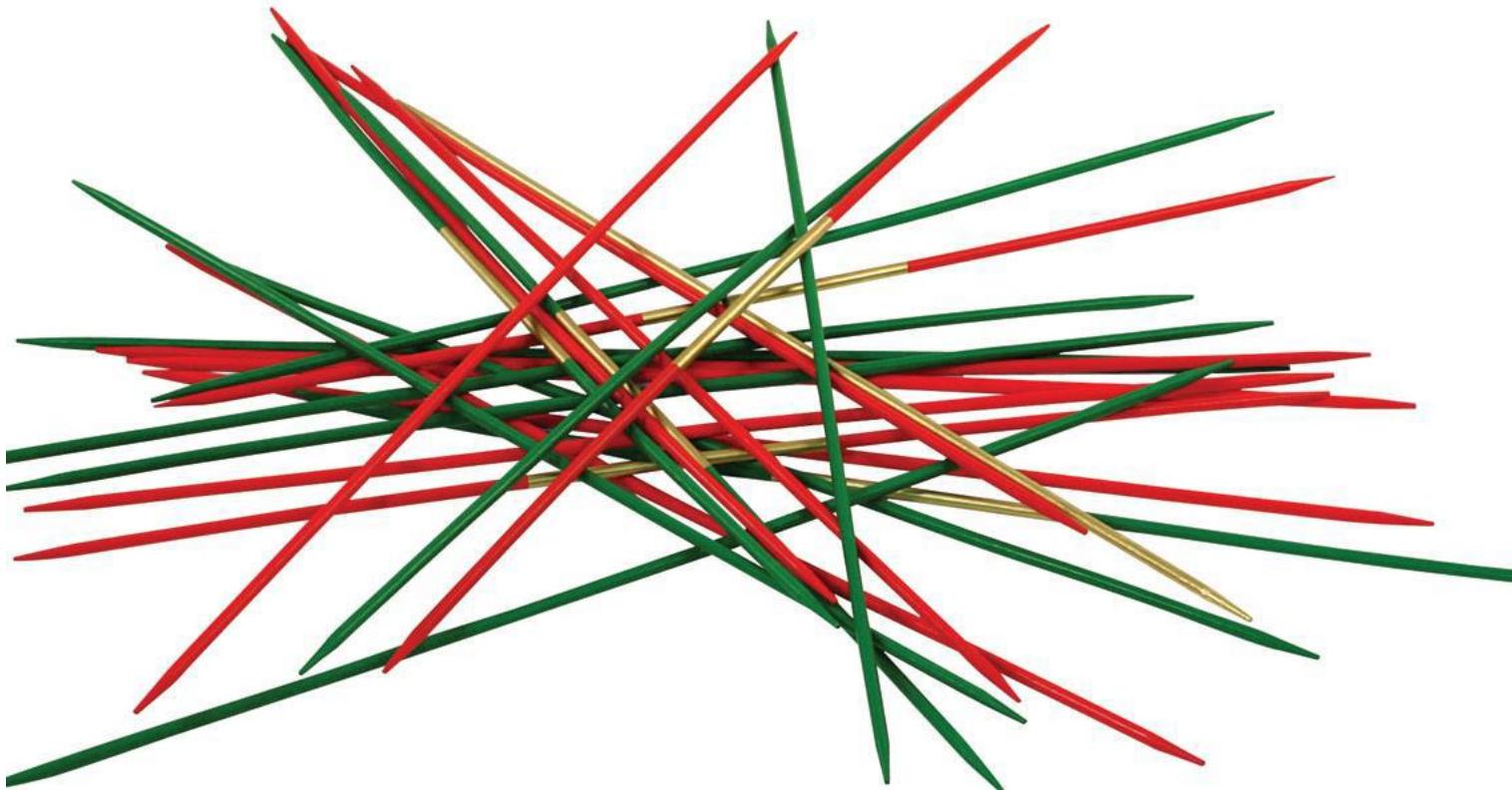
$n$  = number of processes,  $m$  = number of resources types

- **Available:** Vector of length  $m$ . If  $\text{Available}[j] = k$ , there are  $k$  instances of resource type  $R_j$  available.
- **Max:**  $n \times m$  matrix. If  $\text{Max}[i, j] = k$ , then process  $P_i$  may request at most  $k$  instances of resource type  $R_j$ .
- **Allocation:**  $n \times m$  matrix. If  $\text{Allocation}[i, j] = k$  then  $P_i$  is currently allocated  $k$  instances of  $R_j$ .
- **Need:**  $n \times m$  matrix. If  $\text{Need}[i, j] = k$ , then  $P_i$  may need  $k$  more instances of  $R_j$  to finish its task.





# Inspiration

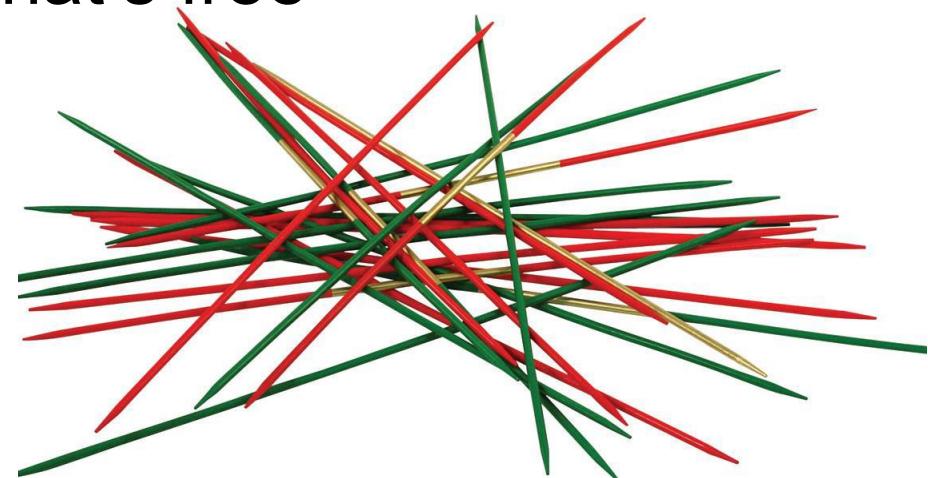




# Playing Pickup Sticks with Processes

## ■ Pickup

- ◆ Find a stick on top  
= Find a process that can finish with what it has plus what's free
- ◆ Remove a stick  
= Process releases its resources



## ■ Repeat

- ◆ Until all processes have finished, Answer: **safe**
- ◆ Or we get stuck,  
Answer: **unsafe**

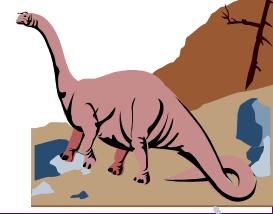




# An Example of Banker's Algorithm

- 5 processes:  $P_0$  through  $P_4$
- 3 resource types: A (10 instances), B (5 instances), and C (7 instances).
- System snapshot at the time  $T_0$ :

	<u>Allocation</u>			<u>Max</u>			<u>Available</u>		
	<i>A</i>	<i>B</i>	<i>C</i>	<i>A</i>	<i>B</i>	<i>C</i>	<i>A</i>	<i>B</i>	<i>C</i>
$P_0$	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			

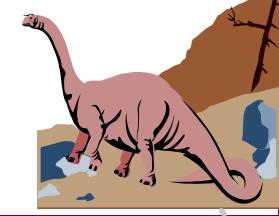




# An Example of Banker's Algorithm (Cont.)

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

	<u>Need</u>		
	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





# Resource-Request Algorithm for Process $P_i$

$\text{Request}$  = request vector for process  $P_i$ .

If  $\text{Request}_i[j] = k$ , then the process  $P_i$  wants  $k$  instances of resource type  $R_j$ .

## Three-Step Algorithm

Step 1. If  $\text{Request}_i \leq \text{Need}_i$ , then go to step 2.  
Otherwise, raise error condition, since the process  $P_i$  has exceeded its maximum claim.

Step 2. If  $\text{Request}_i \leq \text{Available}$ , then go to step 3.  
Otherwise, the process  $P_i$  must wait, since the requested resources are not available.

Step 3. Pretend to allocate requested resources to  $P_i$  by simulating the resource allocation:





# Resource-Request Algorithm for Process $P_i$

## Explain the Step 3 in More Details

Step 3. Pretend to allocate requested resources to process  $P_i$  by modifying the state as follows:

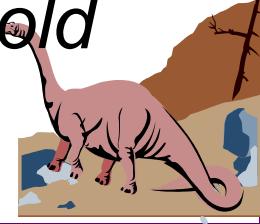
*For each  $j^{th}$  type of resource with  $0 \leq j < m$ ,*

$$\text{Available}_j = \text{Available}_j - \text{Request}_{i,j};$$

$$\text{Allocation}_{i,j} = \text{Allocation}_{i,j} + \text{Request}_{i,j};$$

$$\text{Need}_{i,j} = \text{Need}_{i,j} - \text{Request}_{i,j};$$

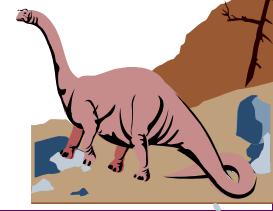
- If safe  $\Rightarrow$  the resources are allocated to process  $P_i$ , and  $P_i$  goes to Ready state
- If unsafe  $\Rightarrow$  process  $P_i$  must wait, and the old resource-allocation state is restored





# Safety Algorithm

- Purpose: Differentiate the safe and unsafe states
- Pessimistic Assumption: all processes will eventually attempt to acquire their stated maximum resources and terminate soon afterward
  - ◆ If a process terminates without acquiring its maximum resource, it only makes it easier on the system

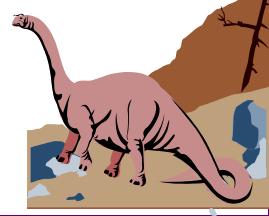




# Safety Algorithm (cont.)

## ■ How to differentiate between safe and unsafe system states?

- ◆ Determines if a state is **safe** by trying to find a hypothetical sequence of requests by the processes that would allow each to acquire its maximum resources and then terminate (returning its resources to the system).
- ◆ Any state where no such sequence exists is an **unsafe** state.





# Safety Algorithm (cont.)

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

*Work* = *Available* (copy the array of available resources)  
*Finish* [*i*] = *false*, for each *i* = 0, 1, ..., *n*-1.

Step 2. Find an *i* such that both conditions satisfy

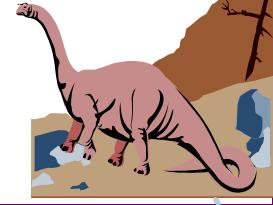
- (a) *Finish* [*i*] = *false*
- (b)  $\text{Need}_i \leq \text{Work}$

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

Step 3. *Finish*[*i*] = *true*;

*Work* = *Work* + *Allocation*; // reclaim resources  
go to step 2.

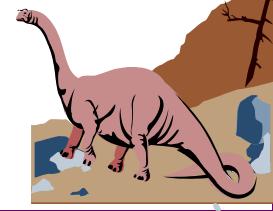
Step 4. If *Finish* [*i*] == *true* for all *i*,  
then the system is in a **safe state**.





# Notes for Safety Algorithm

- These requests and acquisitions are *hypothetical*. The algorithm generates them to check the safety of the state, but no resources are actually given and no processes actually terminate.
- The order in which these requests are generated – if several can be fulfilled – doesn't matter, since safety is checked for each resource request



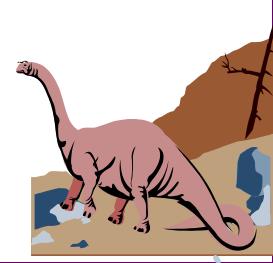


# An Example: $P_1$ Request for (1,0,2)

- Snapshot at time  $T_0$ :

	<u>Allocation</u>			<u>Max</u>			<u>Available</u>			<u>Need</u>		
	<i>A</i>	<i>B</i>	<i>C</i>	<i>A</i>	<i>B</i>	<i>C</i>	<i>A</i>	<i>B</i>	<i>C</i>	<i>A</i>	<i>B</i>	<i>C</i>
$P_0$	0	1	0	7	5	3	3	3	2	7	4	3
$P_1$	2	0	0	3	2	2				1	2	2
$P_2$	3	0	2	9	0	2				6	0	0
$P_3$	2	1	1	2	2	2				0	1	1
$P_4$	0	0	2	4	3	3				4	3	1

- Firstly, check that  $\text{Request} \leq \text{Need}_1$ .  
That is,  $(1,0,2) \leq (1,2,2) \Rightarrow \text{true}$ .

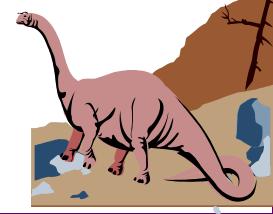




# An Example: $P_1$ Request for (1,0,2)

- Secondly, check that Request  $\leq$  Available.  
That is,  $(1,0,2) \leq (3,3,2) \Rightarrow \text{true.}$
- Thirdly, simulate the resource allocation

	<u>Allocation</u>			<u>Need</u>			<u>Available</u>		
	<i>A</i>	<i>B</i>	<i>C</i>	<i>A</i>	<i>B</i>	<i>C</i>	<i>A</i>	<i>B</i>	<i>C</i>
$P_0$	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			





# An Example: $P_1$ Request for (1,0,2)

- More details about the third step:  
Executing the safety algorithm shows that there exists an execution sequence  $\langle P_1, P_3, P_4, P_0, P_2 \rangle$  that can satisfy the safety requirement.
- Further Questions:
  - ◆ Can the request for (3,3,0) by  $P_4$  be granted?
  - ◆ Can the request for (0,2,0) by  $P_0$  be granted?



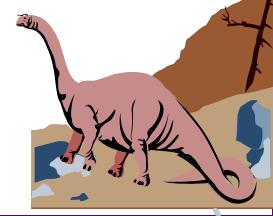


# An Example: $P_4$ Request for (3,3,0)

- Snapshot at time  $T_0$ :

	<u>Allocation</u>			<u>Max</u>			<u>Available</u>			<u>Need</u>		
	<i>A</i>	<i>B</i>	<i>C</i>	<i>A</i>	<i>B</i>	<i>C</i>	<i>A</i>	<i>B</i>	<i>C</i>	<i>A</i>	<i>B</i>	<i>C</i>
$P_0$	0	1	0	7	5	3	3	3	2	7	4	3
$P_1$	2	0	0	3	2	2				1	2	2
$P_2$	3	0	2	9	0	2				6	0	0
$P_3$	2	1	1	2	2	2				0	1	1
$P_4$	0	0	2	4	3	3				4	3	1

- Firstly, check that  $\text{Request} \leq \text{Need}_4$ .  
That is,  $(3,3,0) \leq (4,3,1) \Rightarrow \text{true}$ .



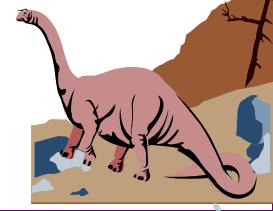


# An Example: $P_4$ Request for (3,3,0)

- Secondly, check that Request  $\leq$  Available.  
That is,  $(3,3,0) \leq (3,3,2) \Rightarrow \text{true.}$
- Thirdly, simulate the resource allocation

	<u>Allocation</u>			<u>Need</u>			<u>Available</u>		
	<i>A</i>	<i>B</i>	<i>C</i>	<i>A</i>	<i>B</i>	<i>C</i>	<i>A</i>	<i>B</i>	<i>C</i>
$P_0$	0	1	0	7	4	3	0	0	2
$P_1$	2	0	0	1	2	2			
$P_2$	3	0	2	6	0	0			
$P_3$	2	1	1	0	1	1			
$P_4$	3	3	2	1	0	1			

*Safe?*  
*Unsafe?*



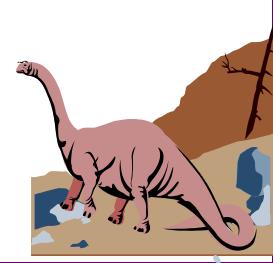


# An Example: $P_0$ Request for (0,2,0)

- Snapshot at time  $T_0$ :

	<u>Allocation</u>			<u>Max</u>			<u>Available</u>			<u>Need</u>		
	<i>A</i>	<i>B</i>	<i>C</i>	<i>A</i>	<i>B</i>	<i>C</i>	<i>A</i>	<i>B</i>	<i>C</i>	<i>A</i>	<i>B</i>	<i>C</i>
$P_0$	0	1	0	7	5	3	3	3	2	7	4	3
$P_1$	2	0	0	3	2	2				1	2	2
$P_2$	3	0	2	9	0	2				6	0	0
$P_3$	2	1	1	2	2	2				0	1	1
$P_4$	0	0	2	4	3	3				4	3	1

- Firstly, check that  $\text{Request} \leq \text{Need}_0$ .  
That is,  $(0,2,0) \leq (7,4,3) \Rightarrow \text{true}$ .



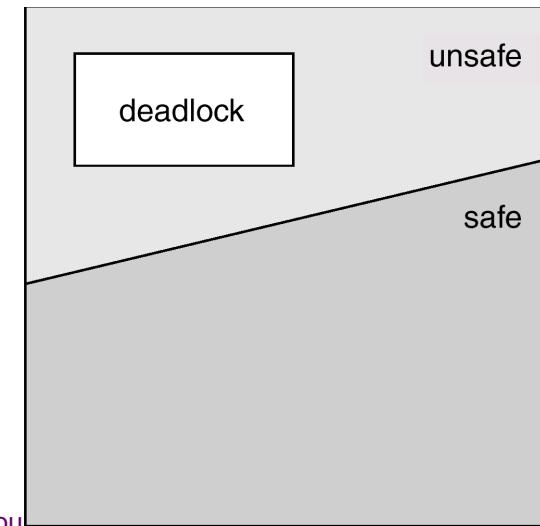


# An Example: $P_0$ Request for (0,2,0)

- Secondly, check that Request  $\leq$  Available.  
That is,  $(0,2,0) \leq (3,3,2) \Rightarrow \text{true.}$
- Thirdly, simulate the resource allocation

	<u>Allocation</u>			<u>Need</u>			<u>Available</u>		
	<i>A</i>	<i>B</i>	<i>C</i>	<i>A</i>	<i>B</i>	<i>C</i>	<i>A</i>	<i>B</i>	<i>C</i>
$P_0$	0	3	0	7	2	3	3	1	2
$P_1$	2	0	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			

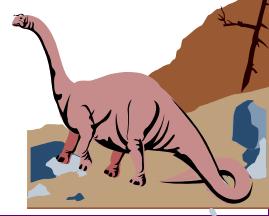
Safe?  
Unsafe?





# Chapter 7: Deadlocks

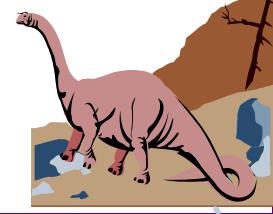
- System Model
- Deadlock Characterization
- Methods for Handling Deadlocks
- Deadlock Prevention (死锁预防)
- Deadlock Avoidance (死锁避免)
- Deadlock Detection (死锁检测) **(死锁检测)**
- Recovery from Deadlock (死锁恢复)





# Deadlock Detection (死锁检测)

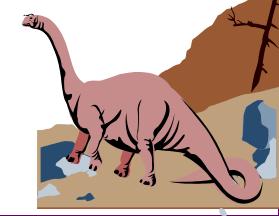
- Deadlock avoidance requires every process to apriori claim its maximum number of resources needed for each resource type
- However, sometimes such knowledge is not available in real systems
- Alternatively, we may adopt the deadlock detection mechanism
  - ◆ Allow the system to enter deadlock state
  - ◆ Run deadlock detection algorithm periodically
  - ◆ Recovery scheme upon the detection of deadlocks





# A Simpler Situation: Single Instance for Each Resource Type

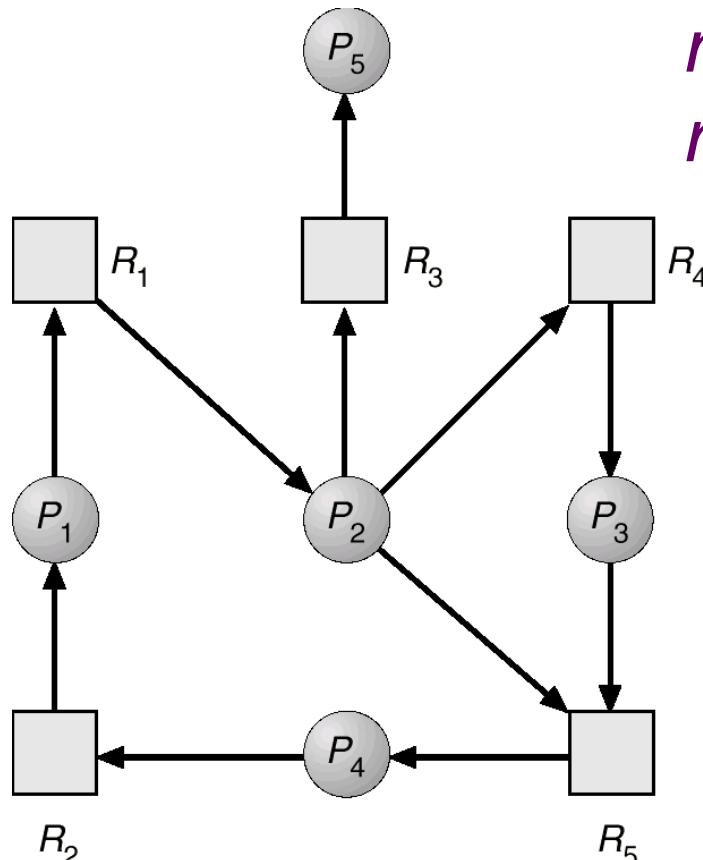
- Maintain *wait-for* graph
  - ◆ Nodes are processes.
  - $P_i \rightarrow P_j$  if  $P_i$  is waiting for  $P_j$ .
- Periodically invoke a deadlock detection algorithm that searches for a cycle in the graph.
- An algorithm to detect a cycle in a graph needs an order of  $n^*e$  operations, where  $n$  ( $e$ ) is the number of vertices (edges) in the graph.





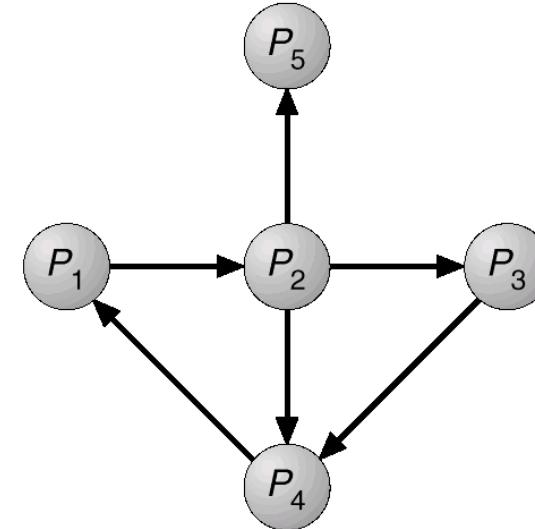
# Resource-Allocation Graph and Wait-for Graph

$m$  resource types  
 $n$  processes



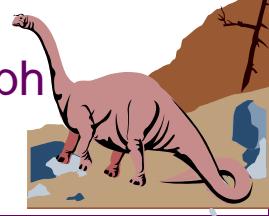
(a)

Resource-Allocation Graph



(b)

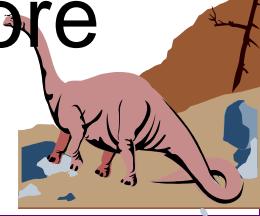
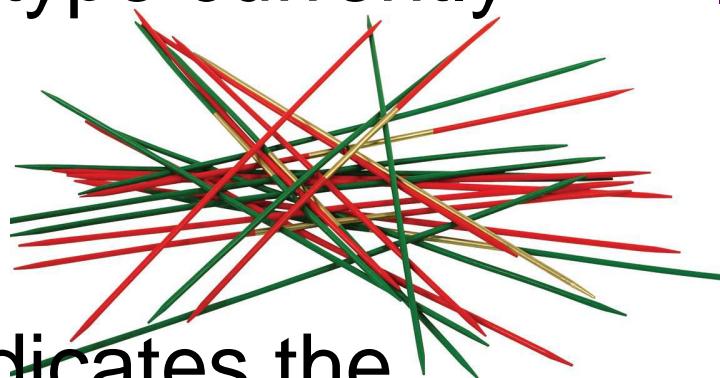
Corresponding Wait-for Graph





# A More Difficult Situation: Multiple Instances for a Resource Type

- **Available:** A vector of length  $m$  indicates the number of available resources of each type
- **Allocation:** An  $n \times m$  matrix defines the number of resources of each type currently allocated to each process.
  - Need matrix (pessimistic) → Request matrix (realistic)
- **Request:** An  $n \times m$  matrix indicates the current request of each process. If  $\text{Request } [i,j] = k$ , then process  $P_i$  is requesting  $k$  more instances of resource type  $R_j$ .





# Deadlock Detection Algorithm

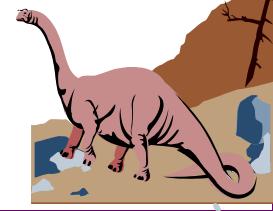
Step 1. Let  $Work$  and  $Finish$  be vectors of length  $m$  and  $n$ , respectively. Initialize:

- (a)  $Work = Available$
- (b) For  $i = 1, 2, \dots, n$ , if  $Allocation_i \neq 0$ , then  $Finish[i] = \text{false}$ ; otherwise,  $Finish[i] = \text{true}$ .

Step 2. Find an index  $i$  such that both:

- (a)  $Finish[i] == \text{false}$
- (b)  $Request_i \leq Work$

If no such  $i$  exists, go to step 4.





# Detection Algorithm (Cont.)

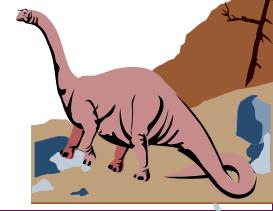
Step 3.  $\text{Finish}[i] = \text{true}$

$\text{Work} = \text{Work} + \text{Allocation}_i$  // reclaim resource  
go to step 2.

Step 4. If  $\text{Finish}[i] == \text{false}$ , for some  $i$ ,  $1 \leq i \leq n$ , then the system is in deadlock state.

Moreover, if  $\text{Finish}[i] == \text{false}$ , then  $P_i$  is deadlocked.

Algorithm requires an order of  $O(m \times n \times n)$  operations to detect whether the system is in deadlocked states.



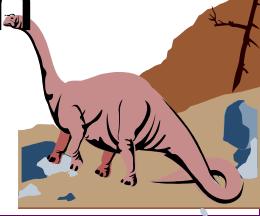


# An Example of Detection Algorithm

- Five processes  $P_{0 \sim 4}$ . Three resource types A (7 instances), B (2 instances) and C (6 instances).
- Snapshot at time  $T_0$ :

	<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			

- Sequence  $\langle P_0, P_2, P_3, P_1, P_4 \rangle$  will result in  $Finish[i] = \text{true}$  for all  $i$ . So, no deadlock.



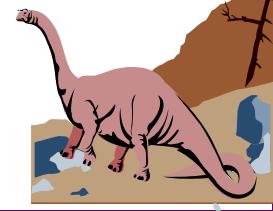


# An Example of Detection Algorithm

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

	<u>Request</u>	<u>Allocation</u>	<u>Available</u>
	A B C	A B C	A B C
$P_0$	0 0 0	0 1 0	0 0 0
$P_1$	2 0 1	2 0 0	
$P_2$	0 0 1	3 0 3	
$P_3$	1 0 0	2 1 1	
$P_4$	0 0 2	0 0 2	

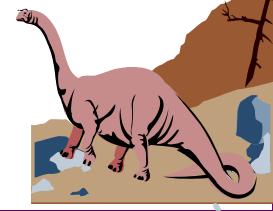
- What is the state of system? Deadlock or no deadlock?





# Detection-Algorithm Usage

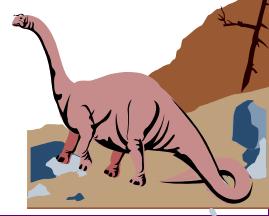
- When, and how often, to invoke depends on:
  - ◆ How often a deadlock is likely to occur?
  - ◆ How many processes will be affected by deadlock when it happens?
  
- Imagine: if detection algorithm is invoked arbitrarily, there may be many cycles in the resource graph and so we would not be able to tell which of the many deadlocked processes “caused” the deadlock.





# Chapter 7: Deadlocks

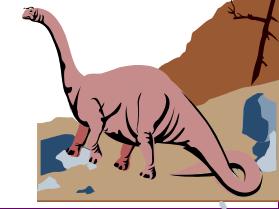
- System Model
- Deadlock Characterization
- Methods for Handling Deadlocks
- Deadlock Prevention (死锁预防)
- Deadlock Avoidance (死锁避免)
- Deadlock Detection (死锁检测)
- Recovery from Deadlock (死锁恢复)





# 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?
  - ◆ 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.
  - ◆ 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 the number of rollbacks when calculating the cost factor for victim selection.





# Concluding Notes

- In general, deadlock detection or avoidance is **expensive**, consuming much system resources
- Real systems must evaluate **cost and frequency of deadlock** against **costs of detection or avoidance**
- Deadlock avoidance and recovery may cause **indefinite postponement (starvation)**
- Unix, Windows use **Ostrich Algorithm** (do nothing)
- Typical apps use **deadlock prevention** (order locks)
- **Database transaction processing systems** (e.g., credit card systems) need to use deadlock detection/recovery/avoidance/prevention (why?)

