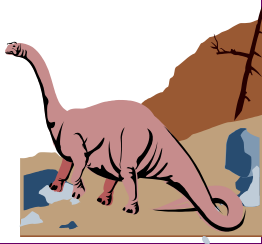
Chapter 7: Deadlocks

肖 卿 俊
办公室：计算机楼532室
电邮：csqjxiao@seu.edu.cn
主页：http://cse.seu.edu.cn/PersonalPage/csqjxiao
电话：025-52091023

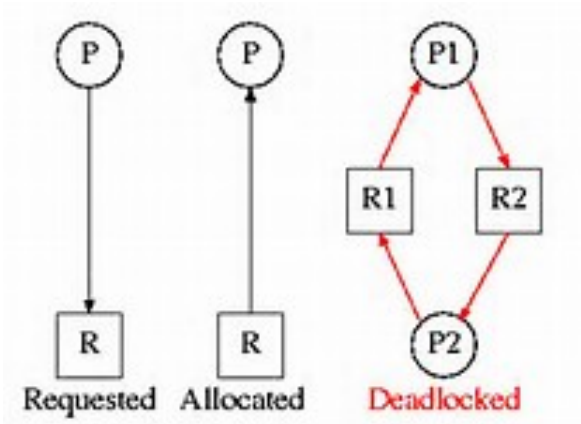
Chapter 7: Deadlocks

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

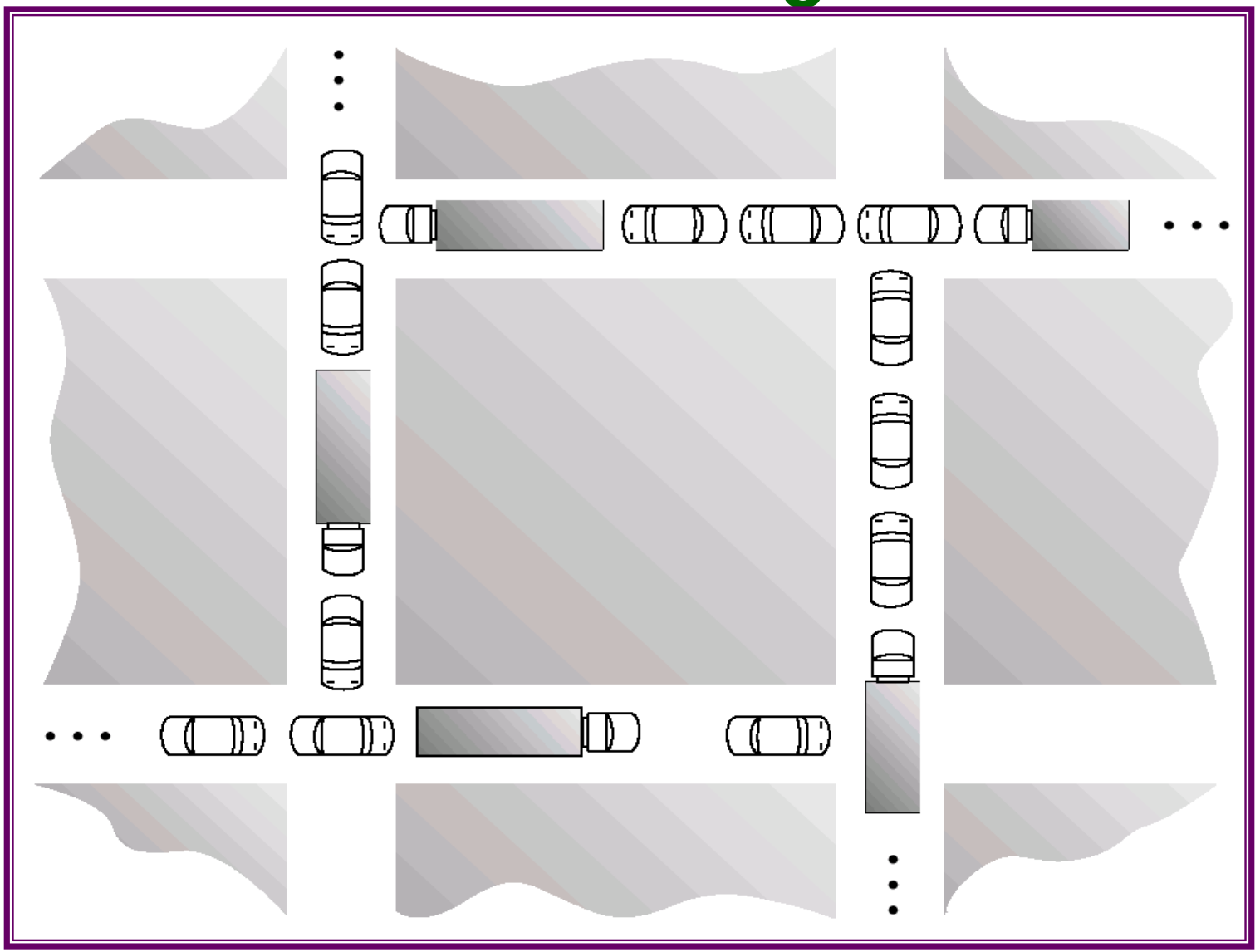


What Is a Deadlock?

- Deadlock (死锁) is a special phenomenon of resource scarcity among a group of processes (or threads)
- A simple example of deadlock between two processes P1 and P2
  - P1 holds R1 and needs R2
  - P2 holds R2 and needs R1



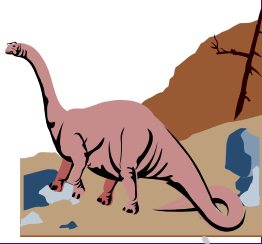
Deadlock can be of a much larger scale



Define the Deadlock Problem

- A set of blocked processes each holding a resource and waiting to acquire a resource held by another process in the set.

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

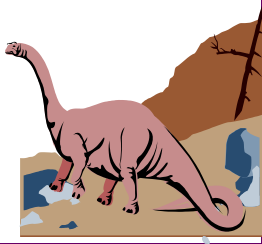


System Model

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

  CPU cycles, memory space, I/O devices

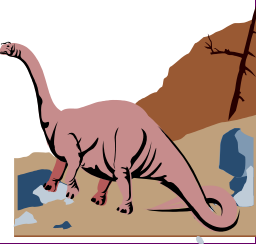
- Each resource type R_i has W_i instances.
- Each process utilizes a resource as follows:
  - request
  - use
  - release



- System resources are utilized 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 blocks until it can acquire the resource.
  - Use:The process can operate on the resource.
  - Release: The process releases the resource.
- *Deadlock*: A set of process 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.

Deadlock Characterization

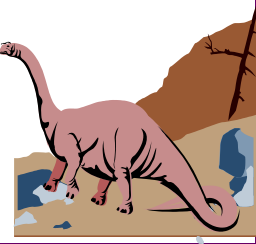
- ***For a deadlock to occur, each of the following four conditions must hold.***
  - **Mutual exclusion:** only one process at a time can use a resource.
  - **Hold and wait:** A process must be holding a resource and waiting for another.
  - **No preemption:** a resource can be released only voluntarily 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.



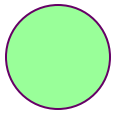
Resource-Allocation Graph

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

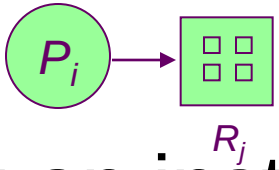
- V is partitioned into two types:
  - $P = \{P_1, P_2, \ldots, P_n\}$, the set consisting of all the processes in the system.
  - $R = \{R_1, R_2, \ldots, R_m\}$, the set consisting of all resource types in the system.
- request edge – directed edge $P_1 \rightarrow R_j$
- assignment edge – directed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

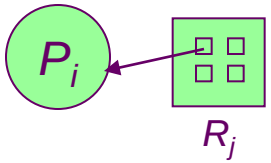


Resource-Allocation Graph (Cont.)

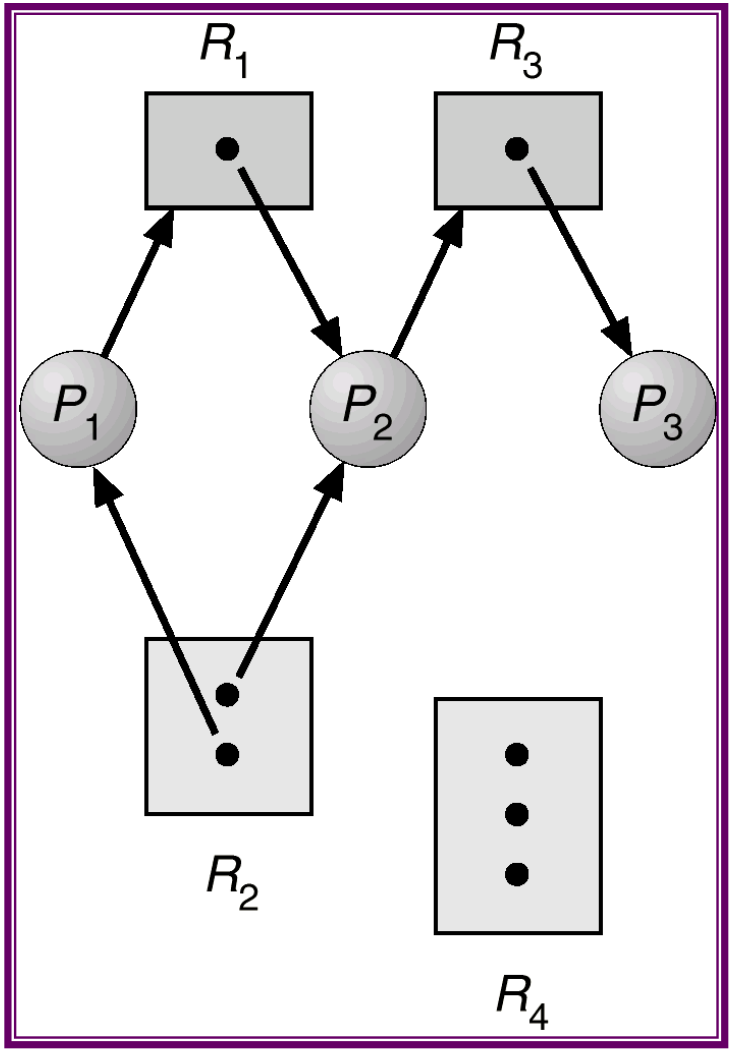
- Process 

  Resource Type with 4 instances 

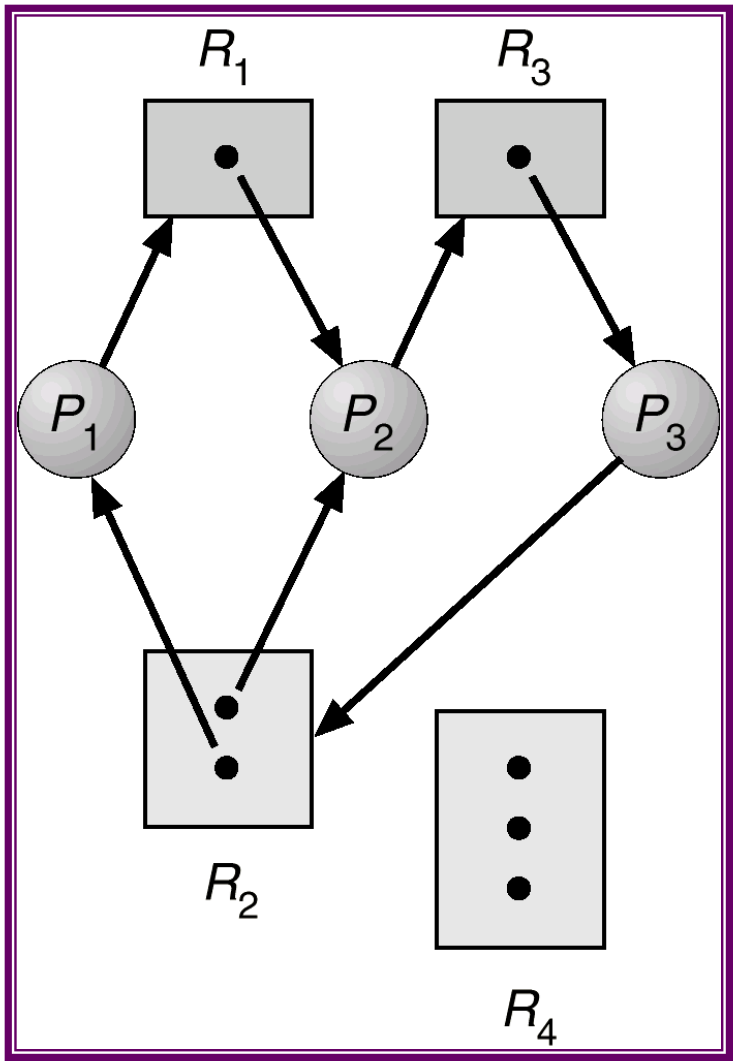
- P_i requests instance of R_j 

- P_i is holding an instance of R_j 

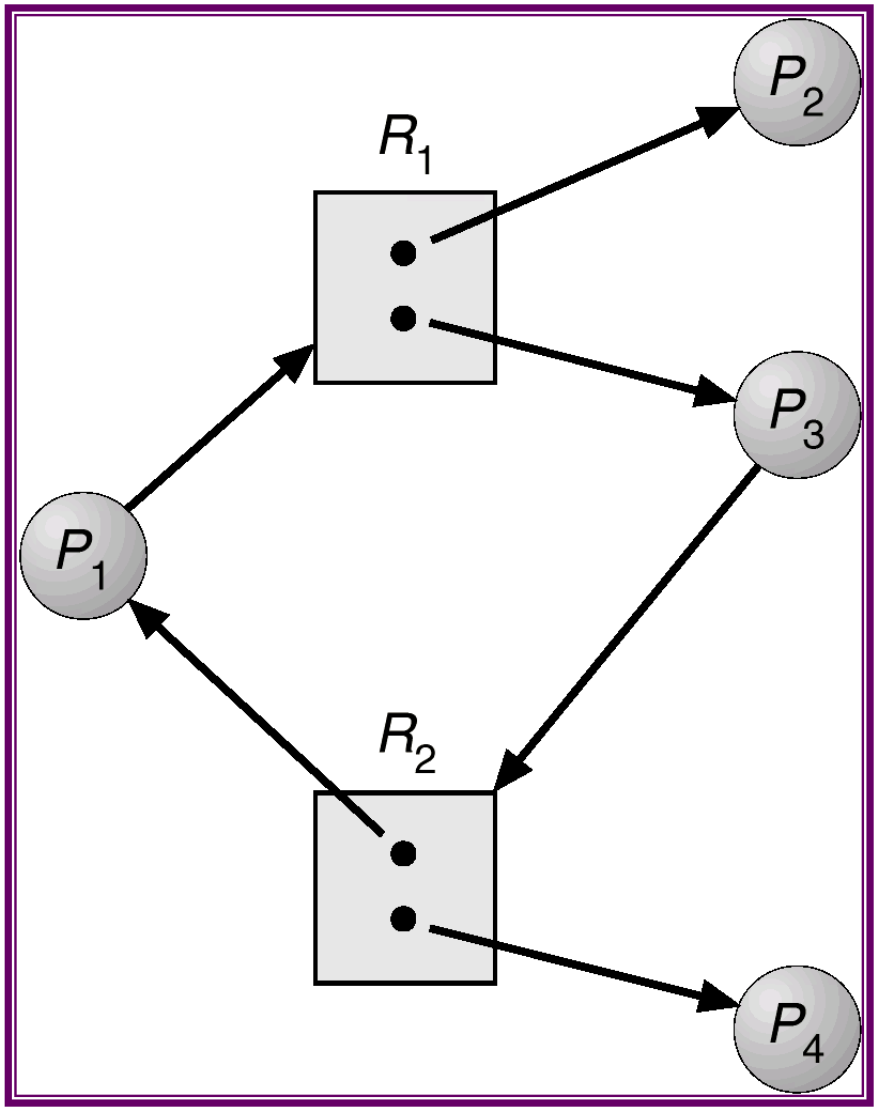
Example of a Resource Allocation Graph



Resource Allocation Graph With A Deadlock

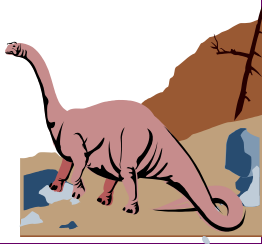


Resource Allocation Graph With A Cycle But No Deadlock



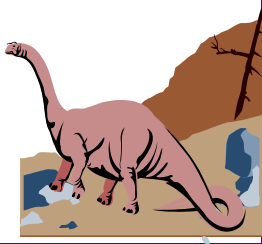
Basic Facts

- If graph contains no cycles $\Rightarrow$ 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.



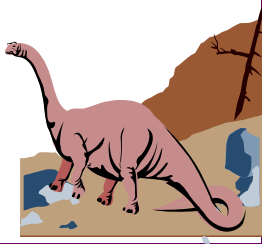
Methods for Handling Deadlocks

- Ensure that the system will *never* enter a deadlock state.
  - **Prevention**: Ensure one of the four conditions fails.
  - **Avoidance**: The OS needs more information so that it can determine if the current request can be satisfied or delayed.
- Allow the system to enter a deadlock state, detect it, and recover.
- Ignore the problem and pretend that deadlocks never occur in the system



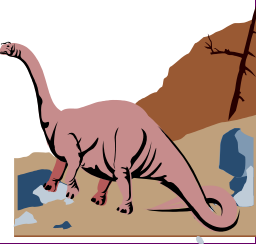
Deadlock Prevention: Mutual Exclusion

- 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.



Deadlock Prevention: Hold and Wait

- No process can 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.*, returning resources before requesting for more).
- Resource utilization may be low, since many resources will be held and unused for a long time.
- Starvation is possible. A process that needs some popular resources may have to wait indefinitely.

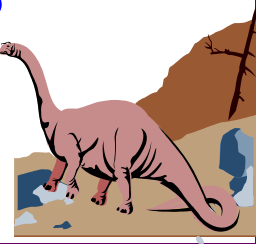


Deadlock Prevention: No Preemption

- 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: Circular Wait

- To break the circular waiting condition, we can 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, no deadlock is possible. Why?**



Deadlock Avoidance

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

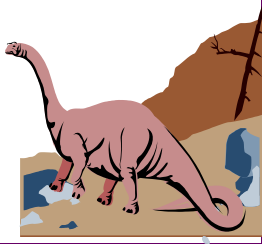
- Simplest and most useful model requires that each process declare the *maximum number* of resources of each type that it may need.
- The deadlock-avoidance algorithm dynamically examines the resource-allocation state to ensure that there can never be a circular-wait condition.
- Resource-allocation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

Safe State

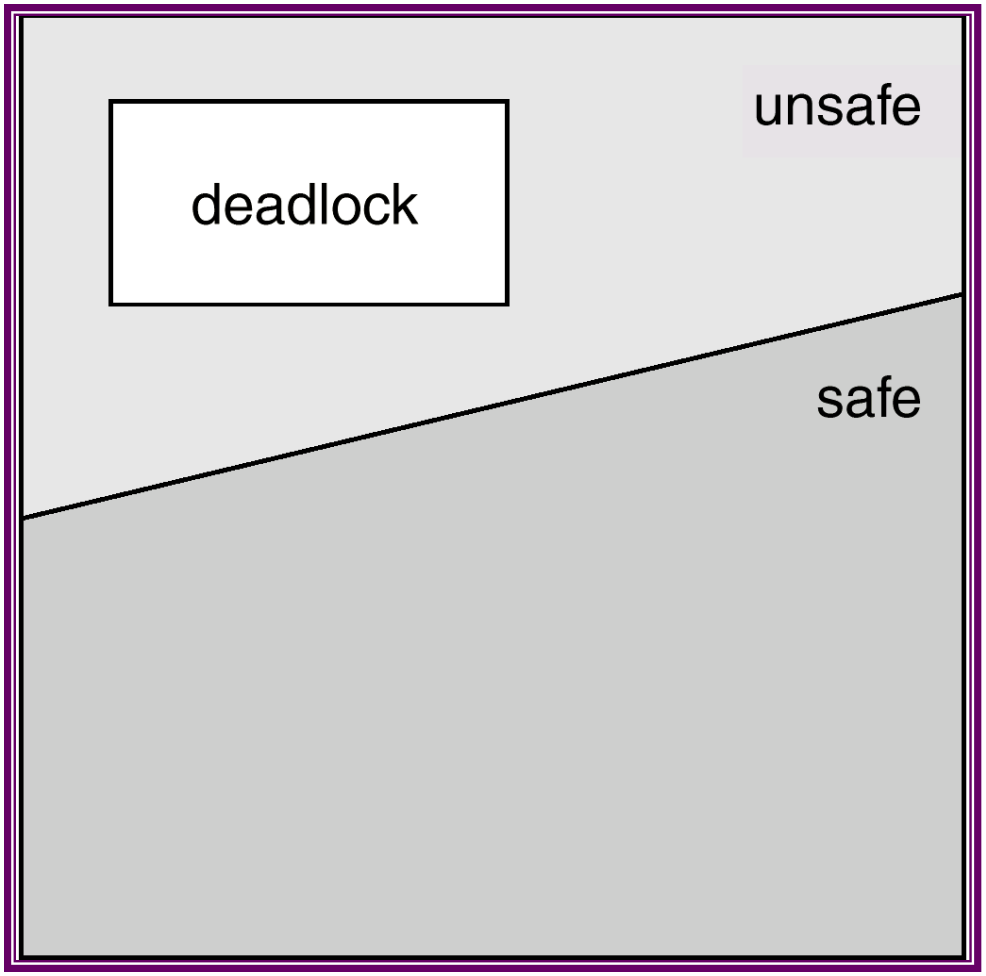
- When a process requests an available resource, system must decide if immediate allocation leaves the system in a *safe state*.
- System is in safe state if there exists a safe sequence of all processes.
- Sequence $<P_1, P_2, \ldots, P_n>$ is safe if for each P_i, the resources that P_i can still request 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 $\Rightarrow$ no deadlocks.
- If a system is in unsafe state $\Rightarrow$ possibility of deadlock.
- Avoidance $\Rightarrow$ ensure that a system will never enter an unsafe state.

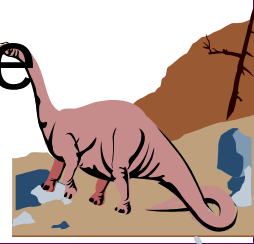


Safe, Unsafe , Deadlock State

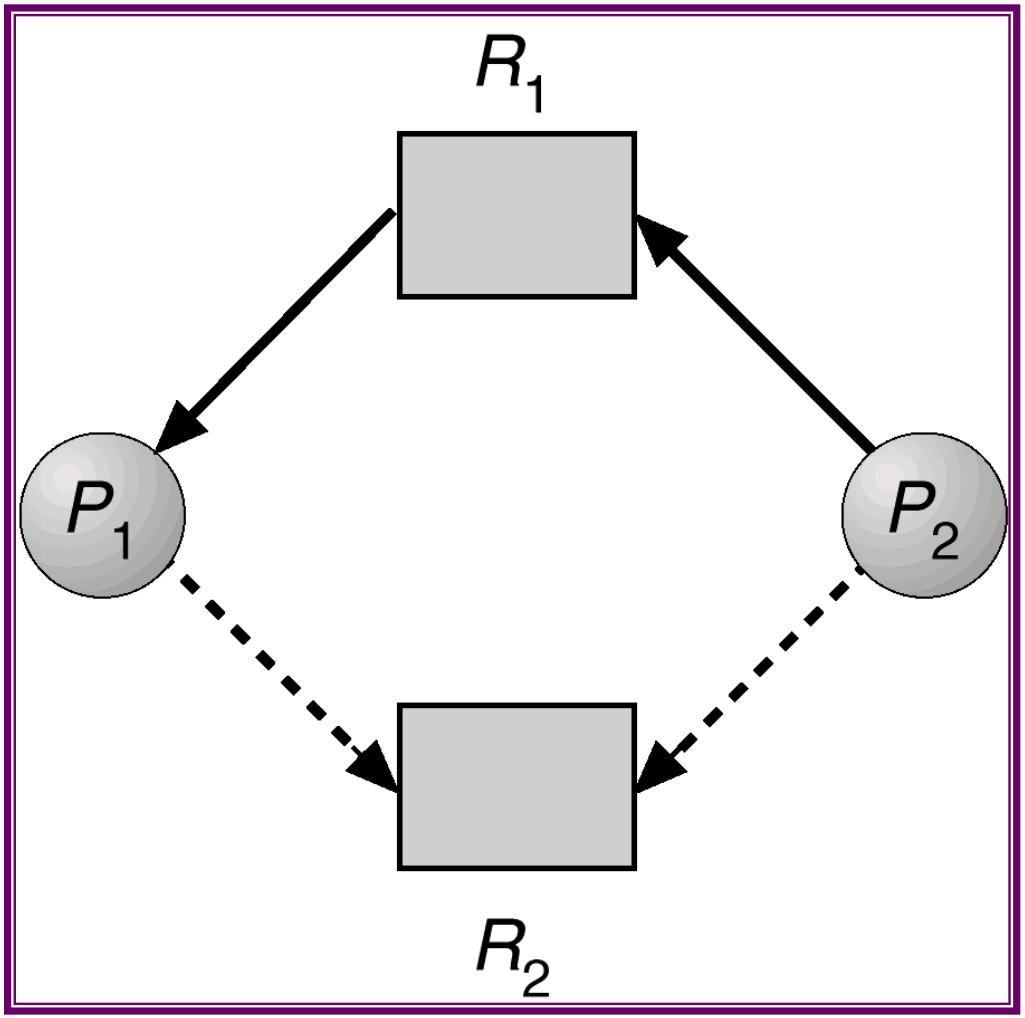


Resource-Allocation Graph Algorithm

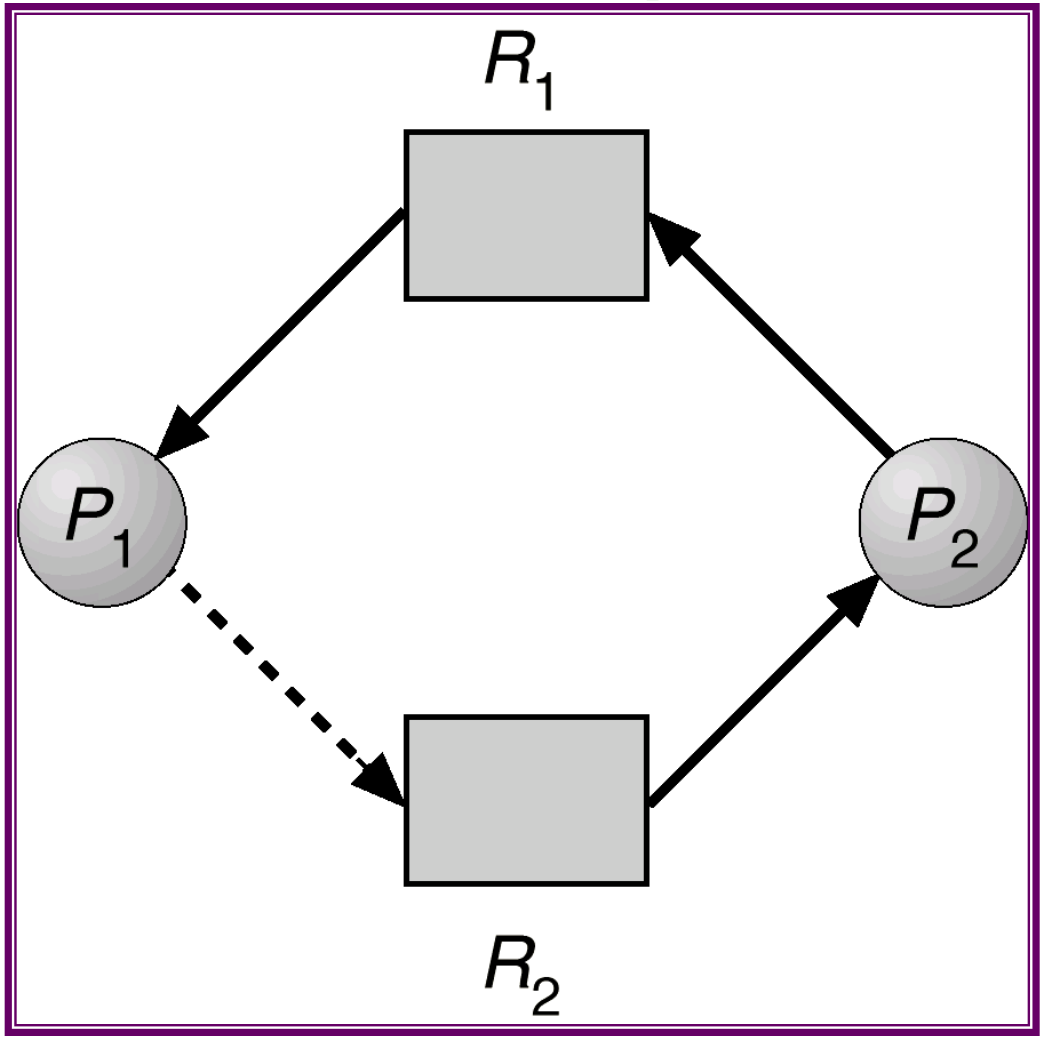
- *Claim edge* $P_i \rightarrow R_j$ indicated that process P_j may request resource R_j; represented by a dashed line.
- Claim edge converts to request edge when a process requests a resource.
- When a resource is released by a process, assignment edge reconverts to a claim edge.
- Resources must be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



Resource-Allocation Graph For Deadlock Avoidance



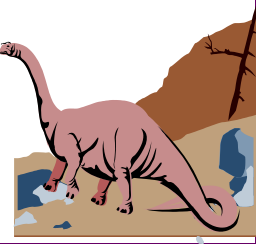
Unsafe State In Resource-Allocation Graph



Banker's Algorithm

- Multiple instances.
- Each process must apriori 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.

http://en.wikipedia.org/wiki/Banker%27s_algorithm



Data Structures for the Banker's Algorithm

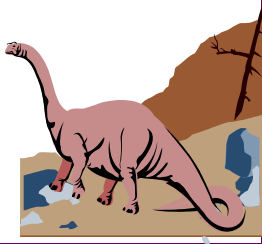
Let n = number of processes, and m = number of resources types.

- *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.
- *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.

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).
- Snapshot at time T_0:

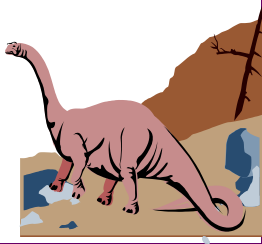
	Allocation	*Max*	*Available*
	A B C	*A B C*	*A B C*
P_0	0 1 0	7 5 3	3 3 2
P_1	2 0 0	3 2 2	
P_2	3 0 2	9 0 2	
P_3	2 1 1	2 2 2	
P_4	0 0 2	4 3 3	



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



Resource-Request Algorithm for Process P_i

Request = request vector for process P_i.

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

Step 1. If $Request_i \leq Need_i$, then go to step 2. Otherwise, raise error condition, since process has exceeded its maximum claim.

Step 2. If $Request_i \leq Available$, go to step 3. Otherwise P_i must wait, since 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 (Cont.)

Explain the Step 3 in 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 $0 \leq j < m$,

$Available_j = Available_j - Request_i[j];$

$Allocation_i[j] = Allocation_i[j] + Request_i[j];$

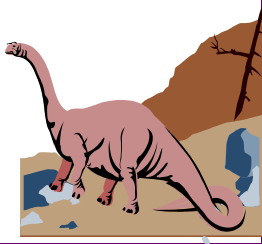
$Need_i[j] = Need_i[j] - Request_i[j]$;

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

Safety Algorithm

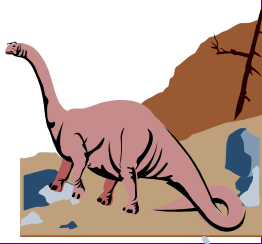
- Purpose: Differentiate the safe and unsafe states

- Assumes that 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??
  - Determines if a state is **safe** by trying to find a hypothetical set 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 set exists is an **unsafe** state.



Safety Algorithm (cont.)

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

Work = *Available* *(copy the array of available resources)*

Finish [*i*] = *false* for *i* = 0, 1, …, *n-1.*

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

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

(b) $Need_i \leq Work$

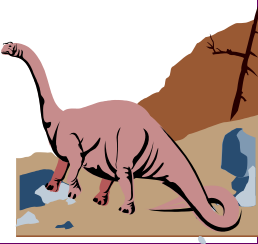
If no such *i* exists, go to step 4.

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

$Work = Work + Allocation_i$ *//return resources*

go to step 2.

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



Safety Algorithm (cont.)

- 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.

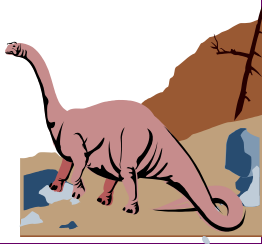
- The order in which these requests are generated – if several can be fulfilled – doesn't matter, since safety is checked for each resource request

Example P_1 Request (1,0,2)

- Check that Request $\leq$ $Need_1$ (that is, (1,0,2) $\leq$ (1,2,2) $\Rightarrow$ *true.*

- Snapshot at time T_0:

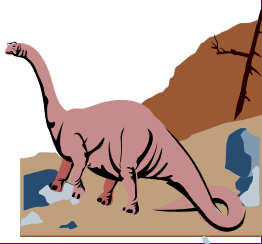
	Allocation	*Max*	*Available*	*Need*
	A B C	*A B C*	*A B C*	*A B C*
P_0	0 1 0	7 5 3	3 3 2	7 4 3
P_1	2 0 0	3 2 2		1 2 2
P_2	3 0 2	9 0 2		6 0 0
P_3	2 1 1	2 2 2		0 1 1
P_4	0 0 2	4 3 3		4 3 1



Example P_1 Request (1,0,2) (Cont.)

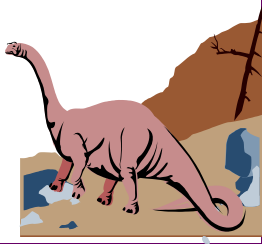
- Check that Request ≤ Available (that is, (1,0,2) ≤ (3,3,2) ⇒ *true.*
- Simulate the resource allocation

	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 1	6 0 0	
P_3	2 1 1	0 1 1	
P_4	0 0 2	4 3 1	



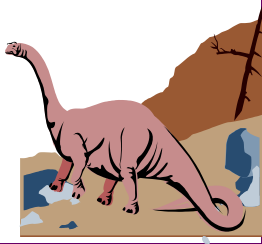
Example P_1 Request (1,0,2) (Cont.)

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



Deadlock Detection

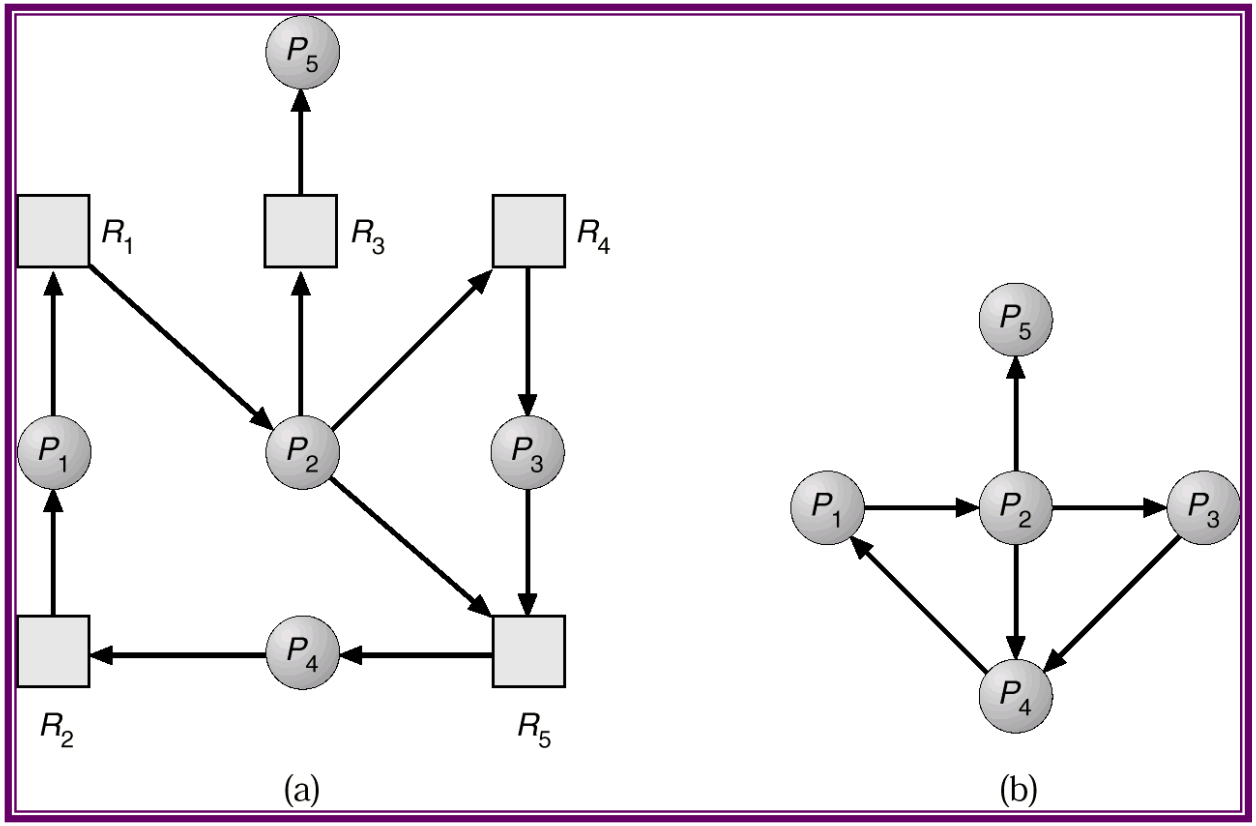
- Allow system to enter deadlock state
- Detection algorithm
- Recovery scheme



Assumption: 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.
- 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

Assumption: 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.

Deadlock Detection Algorithm

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

(a) *Work* = *Available*

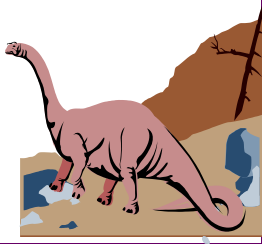
(b) For $i = 1,2, \ldots, n$, if $Allocation_i \neq 0$, then *Finish*[i] = false; otherwise, *Finish*[i] = *true*.

Step 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.

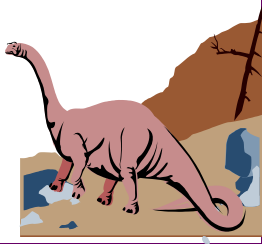


Detection Algorithm (Cont.)

Step 3. *Finish*[*i*] = *true*
Work = *Work* + *Allocation*$_i$
go to step 2.

Step 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 states.

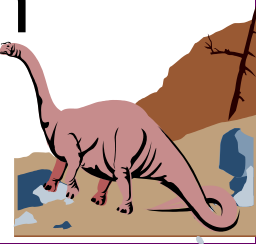


Example of Detection Algorithm

- 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*.

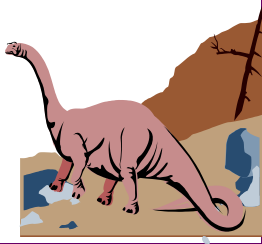


Detection Example (Cont.)

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

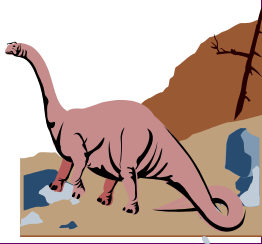
	Request		
	A	*B*	*C*
P_0	0	0	0
P_1	2	0	1
P_2	0	0	1
P_3	1	0	0
P_4	0	0	2

- State of system?



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?
- 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?
  - 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 number of rollback in cost factor.