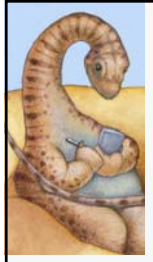
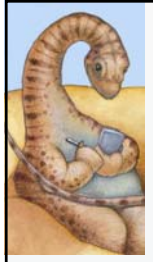
Chapter 7: Deadlocks



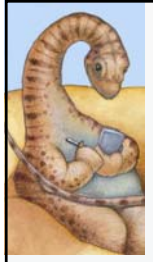
When two trains approach each other at a crossing, both shall come to a full stop and neither shall start up again until the other has gone.

-- A law passed by the Kansas legislature



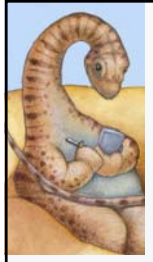
Chapter Objectives

- 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.



Chapter 7: Deadlocks

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

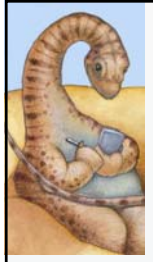


System Model

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

  CPU cycles, memory space, I/O devices

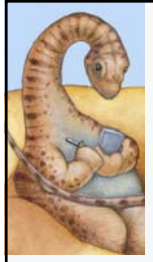
- Each resource type R_i has W_i instances.
- Each process utilizes a resource as follows:
  - request
    - If the request cannot be granted immediately, then the requesting process must wait until it can acquire the resource.
  - use
  - release



Deadlock Example

- Let S and Q be two semaphores initialized to 1

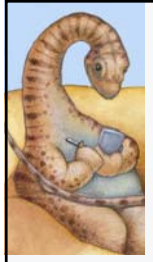
P_0	P_1
wait (S);	wait (Q);
wait (Q);	wait (S);
.	.
.	.
.	.
signal (S);	signal (Q);
signal (Q);	signal (S);



Deadlock Characterization

Deadlock can arise if four conditions hold simultaneously.

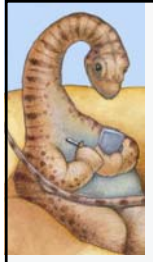
- **Mutual exclusion:** only one process at a time can use a resource instance.
- **Hold and wait:** a process holding at least one resource is waiting to acquire additional resources held by other processes.
- **No preemption:** a resource can be released only voluntarily by the process holding it, after that process has completed its task.
- **Circular wait:** there exists a sequence $\{P_0, P_1, \ldots, P_0\}$ of waiting processes such that P_0 is waiting for a resource that is held by P_1, P_1 is waiting for a resource that is held by $P_2, \ldots, P_{n-1}$ is waiting for a resource that is held by P_n, and P_0 is waiting for a resource that is held by P_0.



Resource-Allocation Graph

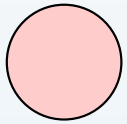
A set of vertices V and a set of edges E.

- 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 $R_j \rightarrow P_i$

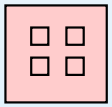


Resource-Allocation Graph (Cont.)

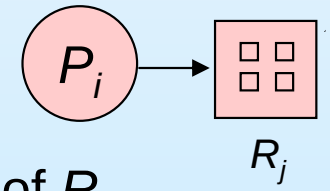
- Process



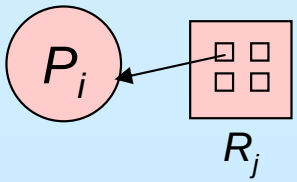
- Resource Type with 4 instances

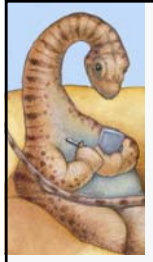


- P_i requests instance of R_j

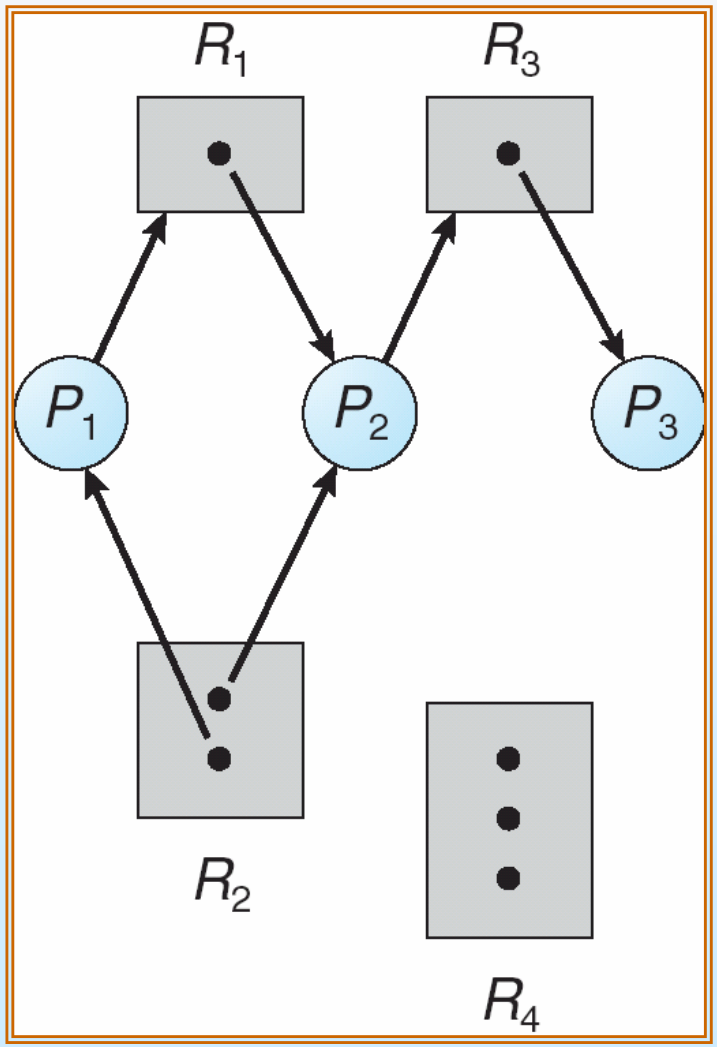


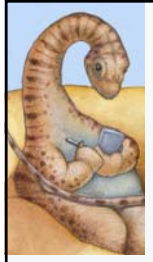
- P_i is holding an instance of R_j





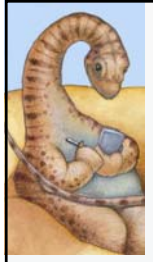
Example of a Resource Allocation Graph



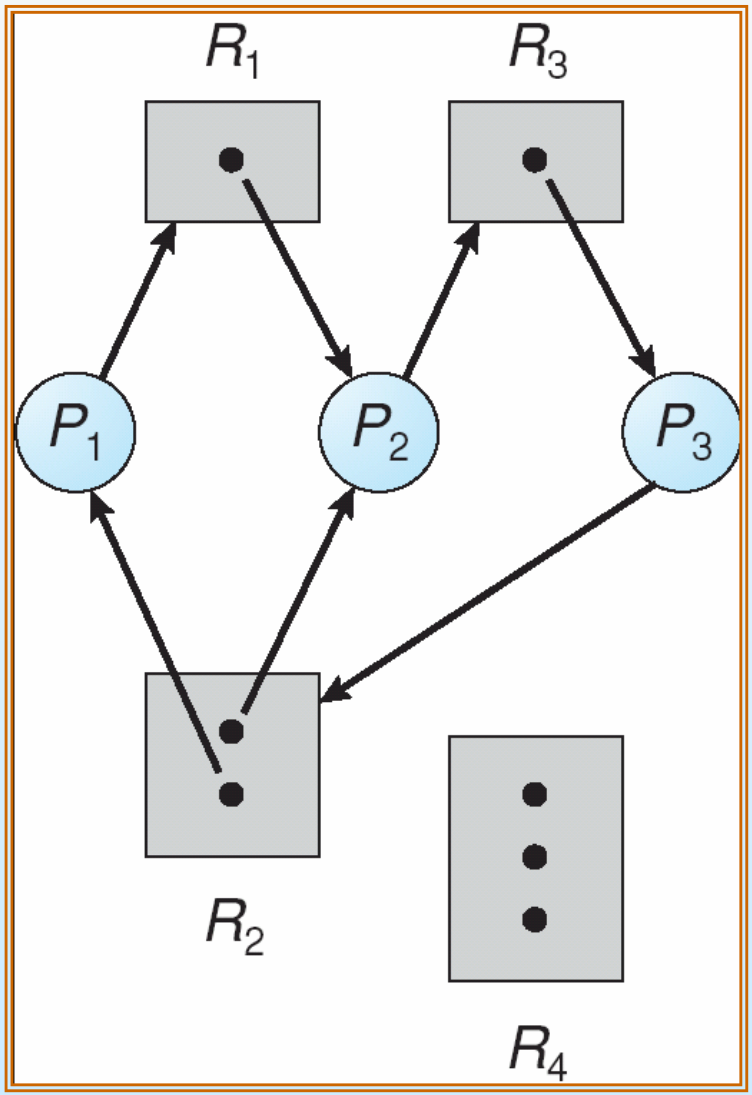


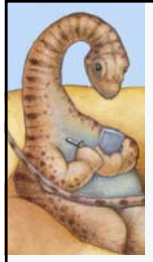
Basic Facts

- 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.

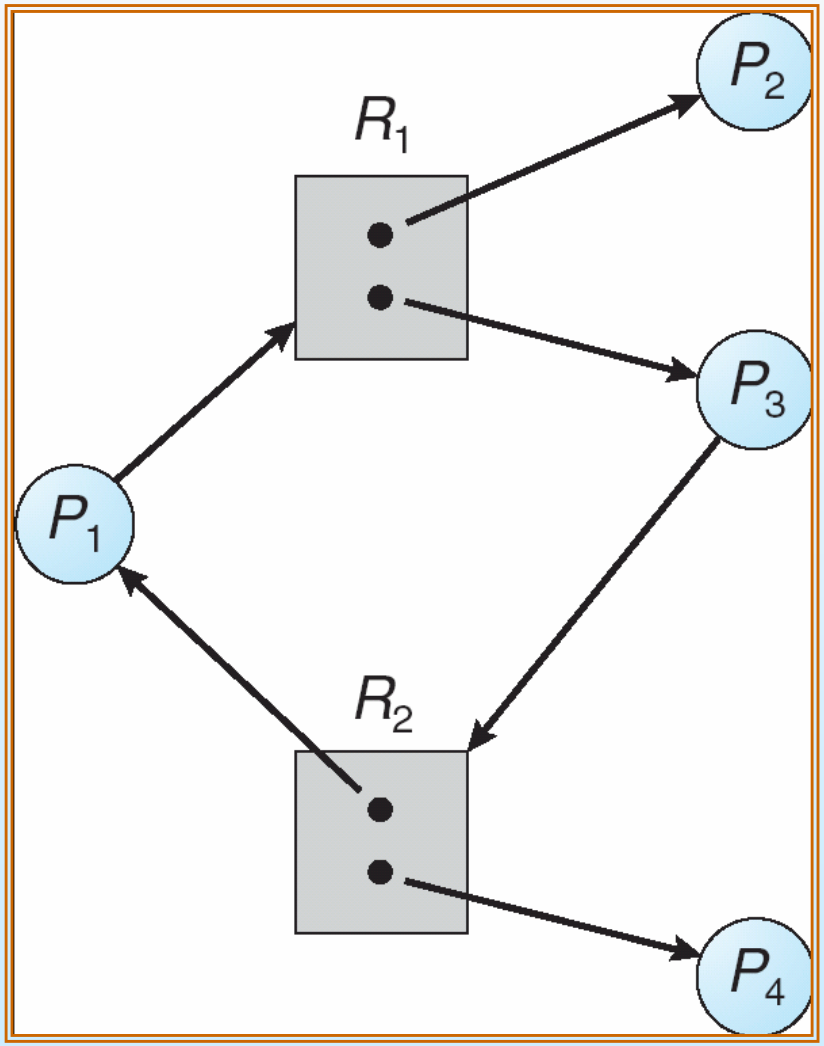


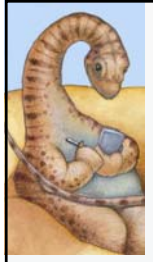
Resource Allocation Graph With A Deadlock





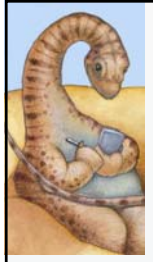
Resource Allocation Graph With A Cycle But No Deadlock





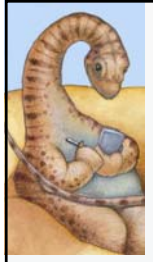
Methods for Handling Deadlocks

- Methods:
  - Ensure that the system will *never* enter a deadlock state.
    - Deadlock prevention or deadlock avoidance
  - 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.
    - It is up to the application developer to write programs that handle deadlocks.
- Some argued that none of basic approaches alone is appropriate for the entire spectrum of resource-allocation problems in OS.
  - The basic approaches can be combined.



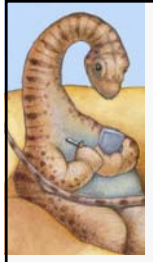
Deadlock Prevention

- Restrain the ways request can be made to ensure at least one of the necessary conditions cannot hold.



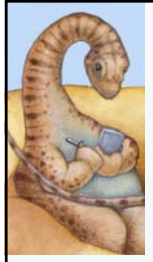
Mutual Exclusion

- Not required for sharable resources; must hold for nonsharable resources.
  - We cannot prevent deadlocks by denying the mutual exclusion condition for nonsharable resources.



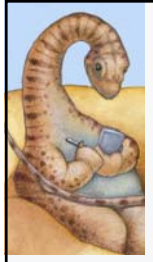
Hold and Wait

- Must guarantee that whenever a process requests a resource, it does not hold any other resources.
  - Require process to request and be allocated all its resources before it begins execution
    - System calls requesting resources precede all other system calls.
  - Allow process to request resources only when the process has none.
    - Before the process can request any additional resources, it must release all its allocated resources.
  - Low resource utilization; starvation possible.



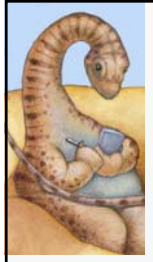
No Preemption

- If a process that is holding some resources requests another resource that cannot be immediately allocated to it, then all resources currently being held are released.
  - Preempted resources are added to the list of resources for which the process is waiting.
- If a process requests some resources, first check whether they are available.
  - If not and held by other process, preempt the desired resources from the waiting process and allocate them to the requesting process.
- A process can be restarted only when it has all resources.



Circular Wait

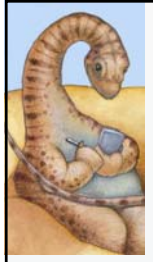
- Impose a total ordering of all resource types
  - Assign a unique integer to each resource type
- Require that each process requests resources in an increasing order of enumeration
  - If several instances of the same resource type are needed, a single request for all of them must be issued.
- Alternatively, require that whenever a process requests an instance of the resource type, it has released any resources higher than the requesting resource.
- Ordering itself does not prevent deadlock; it is up to application developers to write programs that follow the ordering.
  - Certain software can be used to verify that locks are acquired in the proper order.
    - **witness** on BSD



Deadlock Avoidance

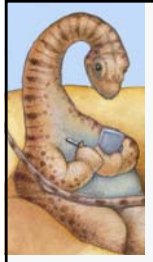
Requires that systems are given in advance additional information concerning which resources a process will request during its lifetime.

- 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.



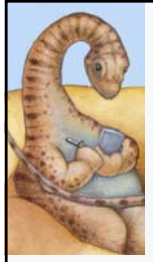
Safe State

- 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 for the current allocation state 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.

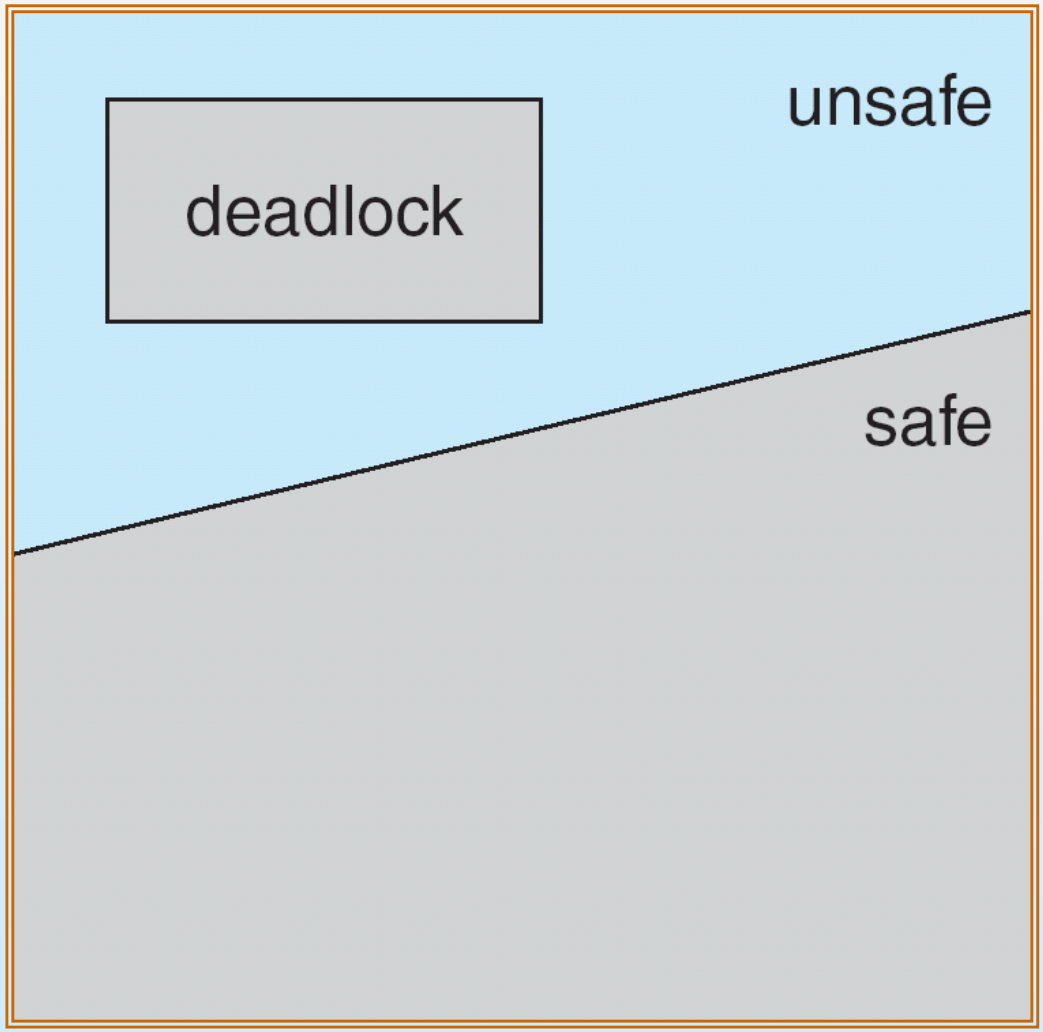


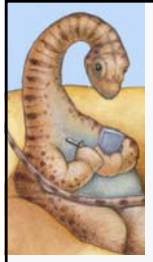
Basic Facts

- If a system is in safe state ⇒ 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



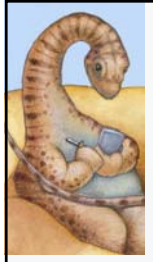


Safe State Example

- 12 magnetic tape drivers and three processes: P_0, P_1, and P_2.

	Maximum Needs	Current Needs
P_0	10	5
P_1	4	2
P_2	9	2

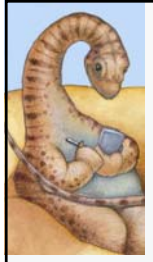
- At current moment, the system is in a safe state. The sequence $<P_1, P_0, P_2>$ satisfies safety condition.
- P_1 can immediately be allocated all its tape drives and then return them – then 5 tape drives available
- Then P_0 can immediately be allocated all its tape drives and then return them – then 10 tape drives available
- Finally P_2 can be allocated all its tape drives and then return them – then 12 tape drives available



Unsafe State Example

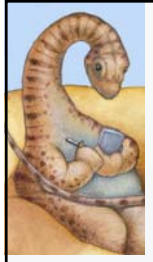
	Maximum Needs	Current Needs
P_0	10	5
P_1	4	2
P_2	9	3

- At this moment, the system is no longer in a safe state.
- Only P_1 can be allocated all its tape drives.
  - After it returns them – then only 4 tape drives available
  - Not enough for maximum needs of P_0 and P_2, and they must wait.
  - Deadlock occurs.
- It is possible to define avoidance algorithms that ensure that the system never deadlock.
  - That is, ensure that the system will always remain in a safe state.

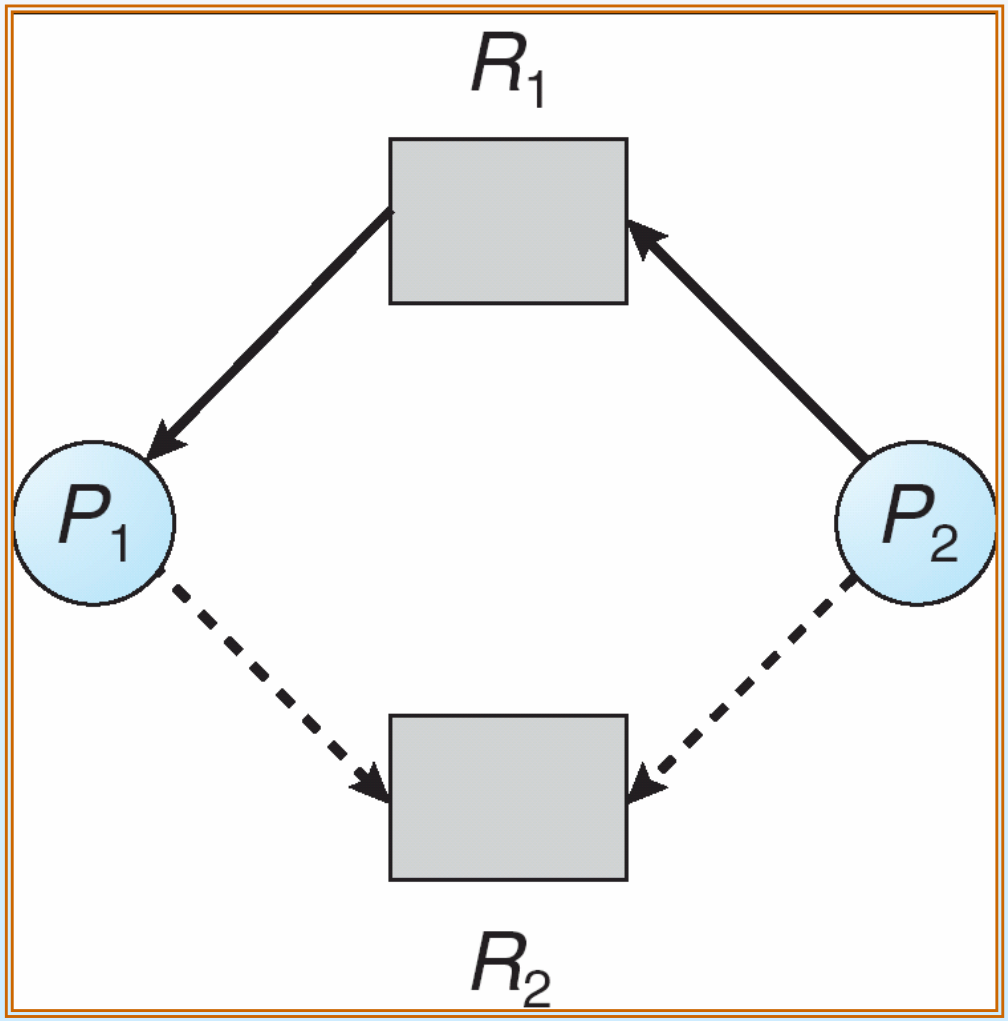


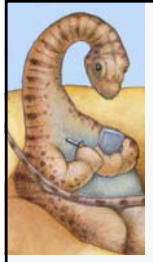
Resource-Allocation Graph Algorithm

- A variant of earlier RAG for deadlock avoidance:
  - one instance for each resource + a new type of edge called a **claim edge**
- *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 *a priori* in the system.



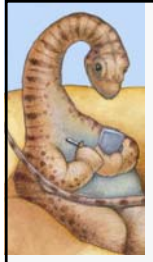
Resource-Allocation Graph For Deadlock Avoidance



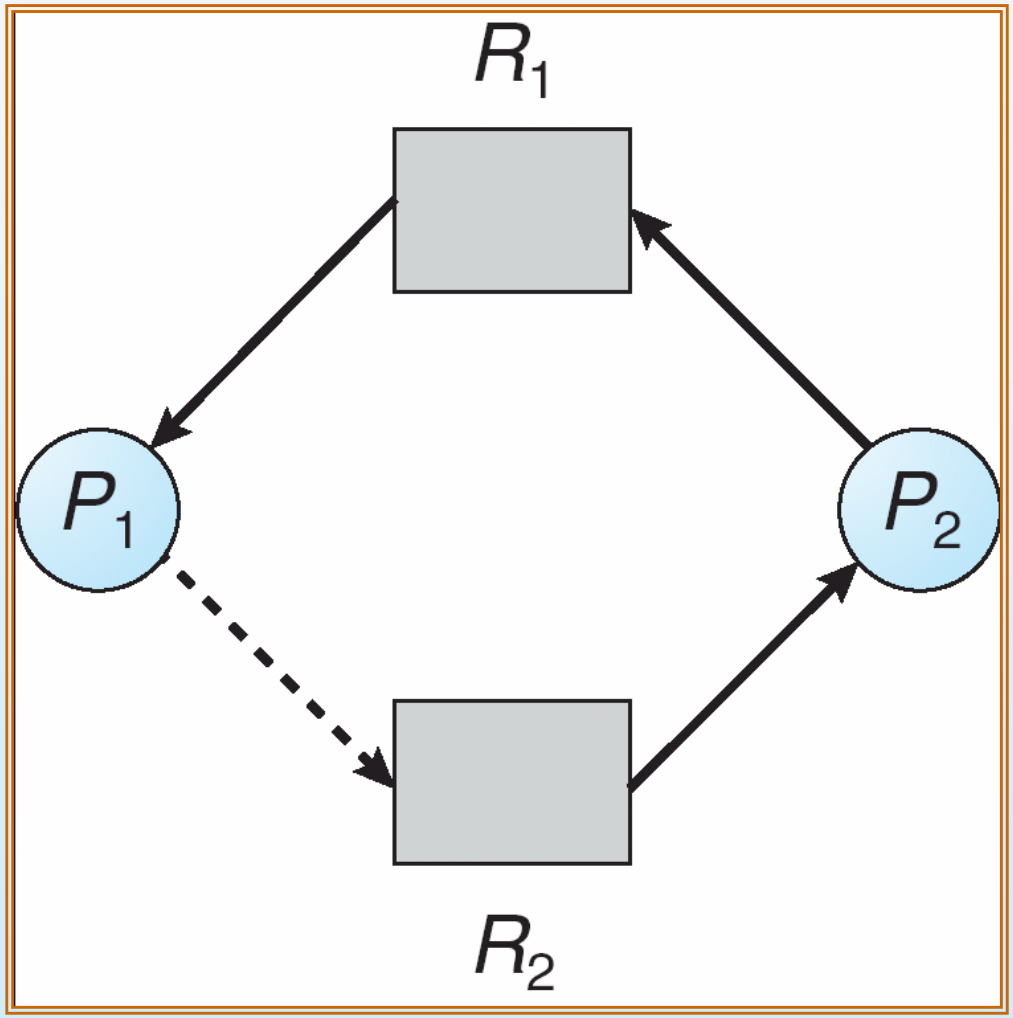


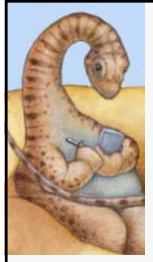
Cycle-Detection Algorithm

- The request $P_i \rightarrow R_j$ can be granted only converting $P_i \rightarrow R_j$ to $R_j \rightarrow P_i$ does not result in the formation of a cycle.
  - Cycle detection is $O(n^2)$, n is number of processes.
- If no cycle exists, then the allocation of the resource $R_j \rightarrow P_i$ will leave the system in a safe state.
- If found, $R_j \rightarrow P_i$ will put the system in an unsafe state.
  - Thus, P_i will have to wait for its requests to be satisfied.



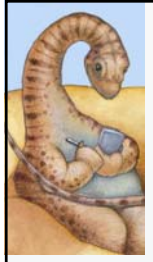
Unsafe State In Resource-Allocation Graph





Banker's Algorithm

- Multiple instances.
- Each process must a priori claim maximum use.
- When a process requests a resource it may have to wait if the allocation will put the system in an unsafe state.
- When a process gets all its resources it must return them in a finite amount of time.

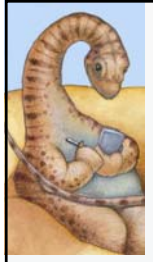


Data Structures for the Banker's Algorithm

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

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

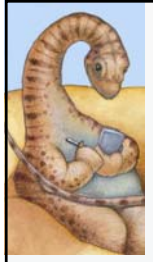
$$Need[i][j] = Max[i][j] - Allocation[i][j].$$



Notation

- Let X and Y be vectors of length n
  - $X \leq Y$ iff $X[i] \leq Y[i]$ for all $i=1, 2, 3, \ldots, n$
  - $X<Y$ if $X \leq Y$ and $X \neq Y$

- Each row in the matrix M is treated as a vector and denoted as M_i.
  - $Allocation_i$ specifies the resources currently allocated to P_i
  - $Need_i$ specifies the addition resources that P_i may still request to complete its task



Safety Algorithm

- To find out whether a system is in a safe state

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

   Work = *Available*

   Finish[*i*] = *false* for *i* = 0,1,2,3, …, *n-1.*

2. Find an *i* such that both:

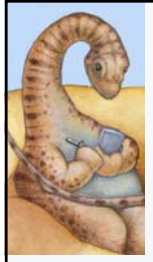
   (a) *Finish*[*i*] = *false*

   (b) $Need_i \leq Work$

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

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

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



Resource-Request Algorithm for Process P_i

- To determine if requests can be safely granted

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

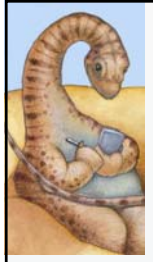
1. If $Request_i \leq Need_i$ go to step 2. Otherwise, raise error condition, since process has exceeded its maximum claim.
2. If $Request_i \leq Available$, go to step 3. Otherwise P_i must wait, since resources are not available.
3. Pretend to allocate requested resources to P_i by modifying the state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

$$Available = Available - Request_i;$$

$$Allocation_i = Allocation_i + Request_i;$$

$$Need_i = Need_i - Request_i;$$

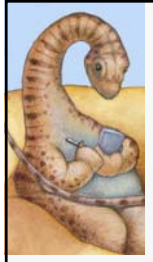
   - If safe $\Rightarrow$ the resources are allocated to P_i
   - If unsafe $\Rightarrow$ P_i must wait, and the old resource-allocation state is restored



Example of Banker's Algorithm

- 5 processes P_0 through P_4
- 3 resource types *A* (10 instances), *B* (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	

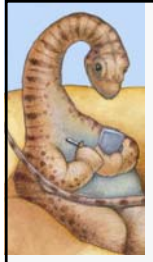


Example (Cont.)

- 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

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

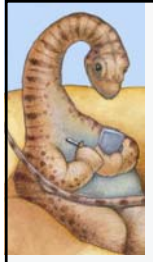


Example P_1 Request (1,0,2) (Cont.)

- Check that $Request_1 \leq Available$; that is, (1,0,2) ≤ (3,3,2) ⇒ true.

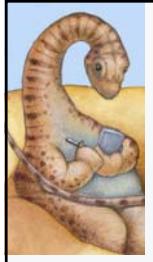
	Allocation	*Need*	*Available*
	A B C	*A B C*	*A B C*
P_0	0 1 0	7 4 3	2 3 0
P_1	3 0 2	0 2 0	
P_2	3 0 1	6 0 0	
P_3	2 1 1	0 1 1	
P_4	0 0 2	4 3 1	

- Executing safety algorithm shows that sequence < 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?



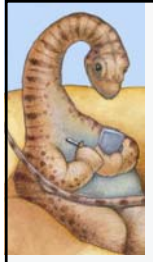
Deadlock Detection

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

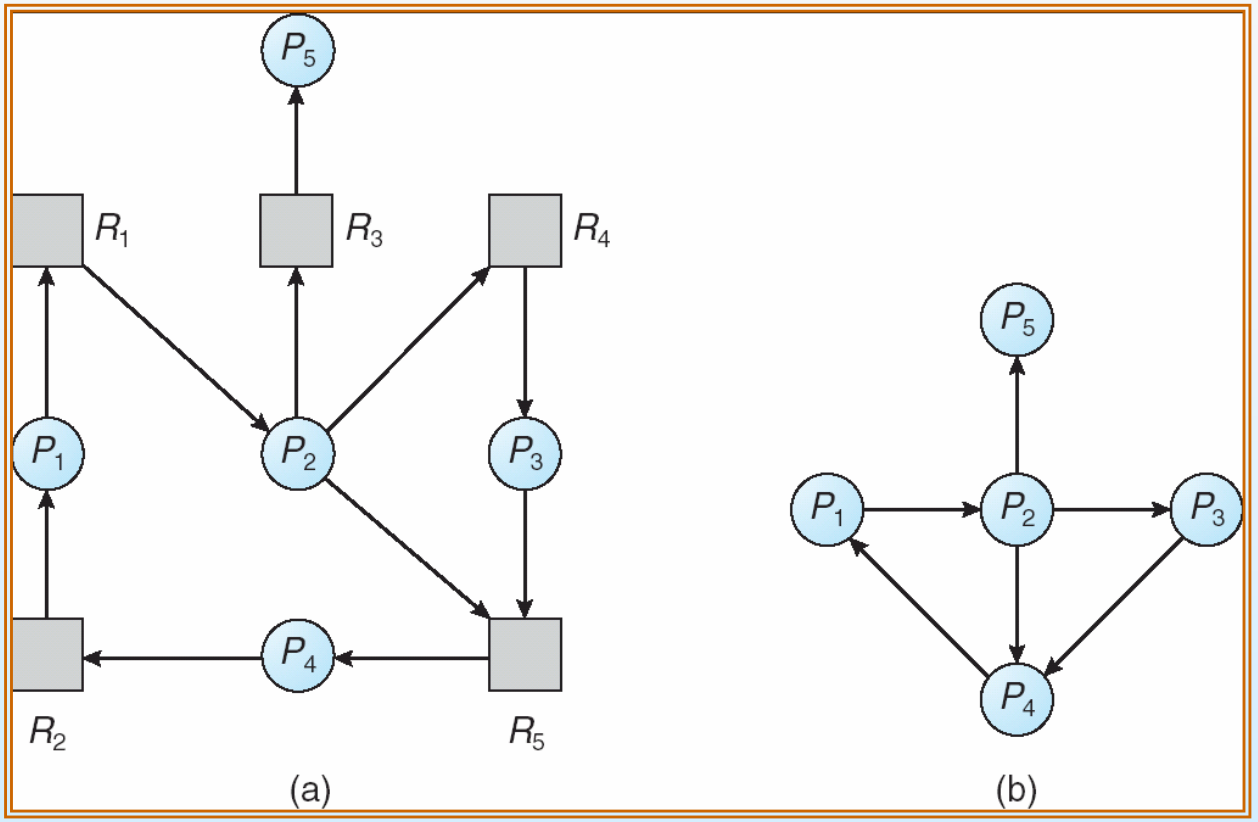


Single Instance of Each Resource Type

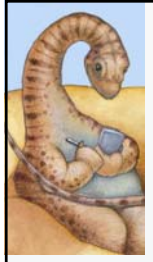
- Maintain *wait-for* graph
  - Nodes are processes.
  - $P_i \rightarrow P_j$ if P_i is waiting for P_j.
- Periodically invoke an algorithm that searches for a cycle in the graph.
- An algorithm to detect a cycle in a graph requires an order of n^2 operations, where n is the number of vertices in the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



Resource-Allocation Graph and Wait-for Graph

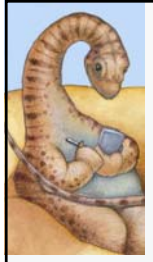


Resource-Allocation Graph Corresponding wait-for graph



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

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

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

   (b) $Request_i \leq Work$

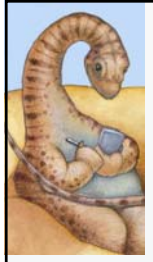
   If no such *i* exists, go to step 4.

3. $Work = Work + Allocation_i$

   Finish[*i*] = *true*

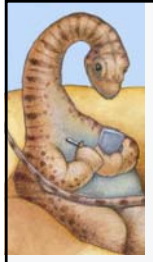
   go to step 2.

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.



Detection Algorithm (Cont.)

- The algorithm simply investigates every possible allocation sequence for the unfinished processes.
- Algorithm requires an order of $O(m \times n^2)$ operations to detect whether the system is in deadlocked state.

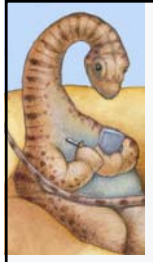


Example of Detection Algorithm

- 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_1, P_3, P_4> will result in *Finish*[*i*] = true for all *i*.

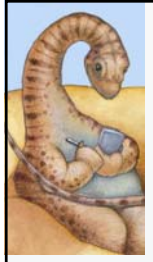


Example (Cont.)

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

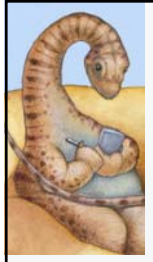
	Request
	A B C
P_0	0 0 0
P_1	2 0 2
P_2	0 0 1
P_3	1 0 0
P_4	0 0 2

- 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.



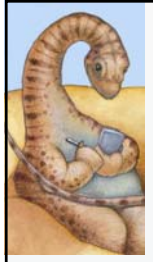
Detection-Algorithm Usage

- 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?
  - 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.

End of Chapter 7