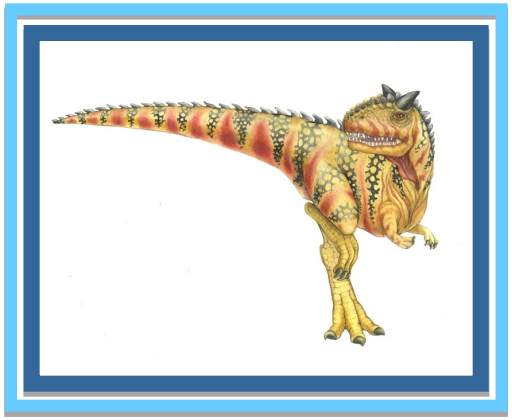
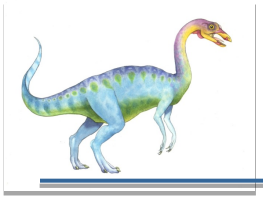
Chapter 7: Deadlocks

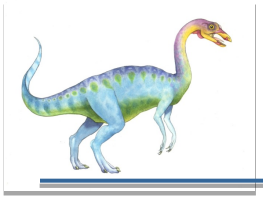




Chapter 7: Deadlocks

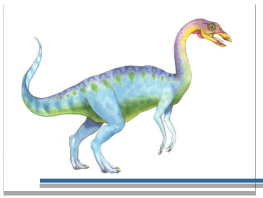
- System Model
- Deadlock Characterization
- Methods for Handling Deadlocks
- Deadlock Prevention
- Deadlock Avoidance

- Sections from the textbook: 7.1, 7.2, 7.3, 7.4, and 7.5



Chapter Objectives

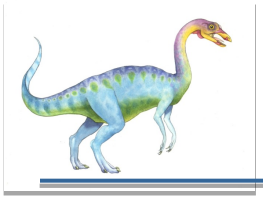
- To develop a description of deadlocks, which prevent sets of concurrent processes from completing their tasks.
- To present a number of different methods for preventing or avoiding deadlocks in a computer system.



Deadlock

- In a multiprogramming environment, several processes may compete for a finite number of resources.
- A process requests resources.
  if the resources are not available → the process enters a waiting state
- A waiting process is never again able to change state.
  →because the requested resources are held by other waiting processes.

This situation is called a ***deadlock.***



System Model

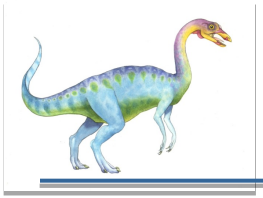
- A system consists of a finite number of resources to be distributed among a number of competing processes.

- Resource types R_1, R_2, . . ., R_m
  Resources can be either:
  - *(physical) CPU cycles, memory space, I/O devices(printers, DVD drivers)*
  - *(logical) semaphores, mutex locks, Files.*

- The resources may be partitioned into several types (or classes), each consisting of some number of identical instances
  - Each resource type R_i has W_i instances.
  - The resource classes/types must be defined properly.

- Each process utilizes a resource as the following sequence:
  - **request**
  - **use**
  - **Release**

- A deadlock may involve the same resource type or different resource types.

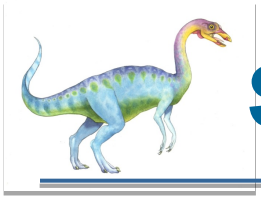


Deadlock Characterization

Deadlock can arise if four conditions hold simultaneously.

All four conditions must hold for a deadlock to occur

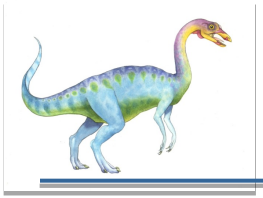
- **Mutual exclusion**: only one process at a time can use a resource
- **Hold and wait**: a process holding at least one resource is waiting to acquire additional resources held by other processes
- **No preemption**: a resource can be released only voluntarily by the process holding it, after that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



System Resource-Allocation Graph

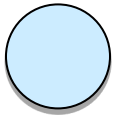
A directed graph that has a set of vertices V and a set of edges E.

- V is partitioned into two types:
  - $P = \{P_1, P_2, \ldots, P_n\}$, the set consisting of all the processes in the system (Circles)
  - $R = \{R_1, R_2, \ldots, R_m\}$, the set consisting of all resource types in the system (Rectangles)
- E can be divided into:
  - **request edge** – directed edge $P_i \rightarrow R_j$
    (a request edge points to only the rectangle R_j)
  - **assignment edge** – directed edge $R_j \rightarrow P_i$
    (an assignment edge must also designate one of the dots in the rectangle)



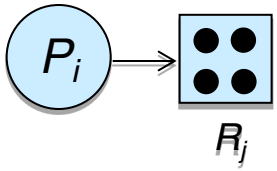
Resource-Allocation Graph (Cont.)

- Process

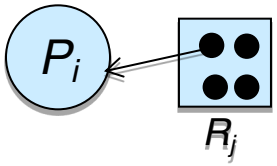


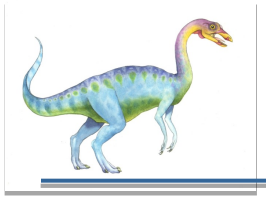
- Resource Type with 4 instances

- P_i requests instance of R_j

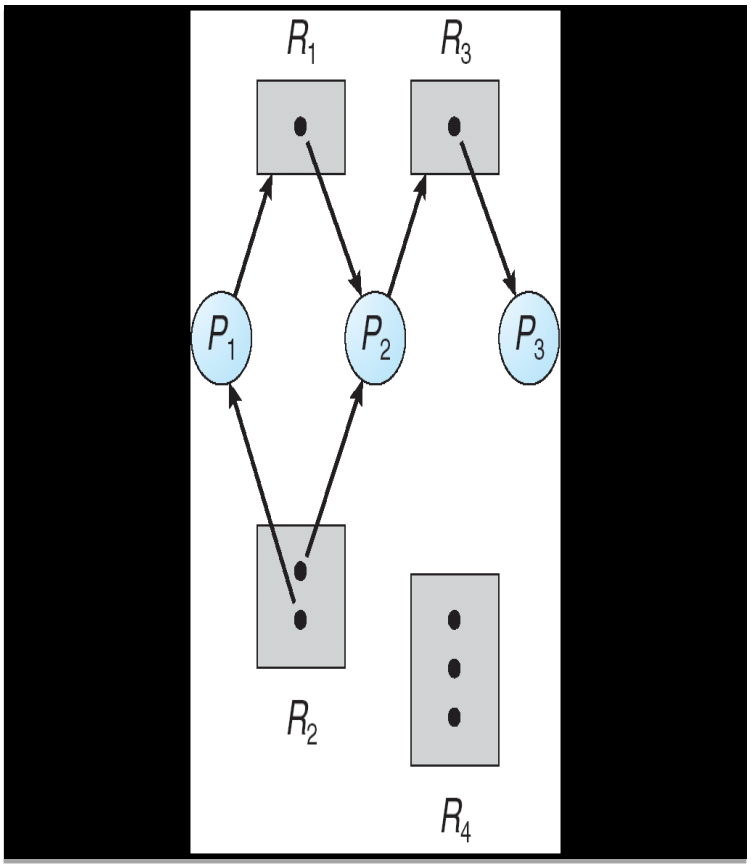


- P_i is holding an instance of R_j





Example of a Resource Allocation Graph



The sets *P, R,* and *E*:

$P=\{P_1, P_2, P_3\}$

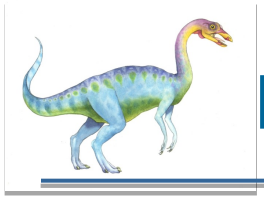
$R=\{R_1, R_2, R_3, R_4\}$

$E=\{P_1 \rightarrow R_1, P_2 \rightarrow R_3, R_1 \rightarrow P_2, R_2 \rightarrow P_2, R_2 \rightarrow P_1, R_3 \rightarrow P_3\}$

Is there a deadlock?

If the graph contains no cycles, then no process in the system is deadlocked.

If the graph contains a cycle, then a deadlock **may** or **may not** exist.

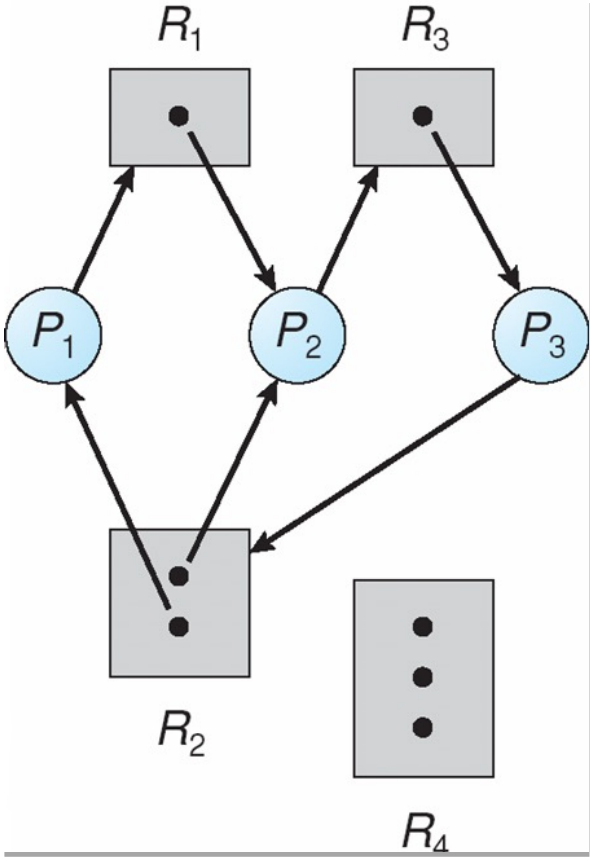


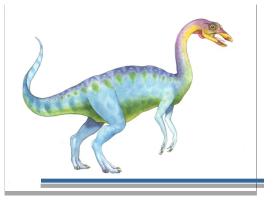
Resource Allocation Graph With A Deadlock

In this example, two cycles exist in the system:

- $P_1 \rightarrow R_1 \rightarrow P_2 \rightarrow R_3 \rightarrow P_3 \rightarrow R_2 \rightarrow P_1$
- $P_2 \rightarrow R_3 \rightarrow P_3 \rightarrow R_2 \rightarrow P_2$

Processes P_1, P_2, and P_3 are deadlocked

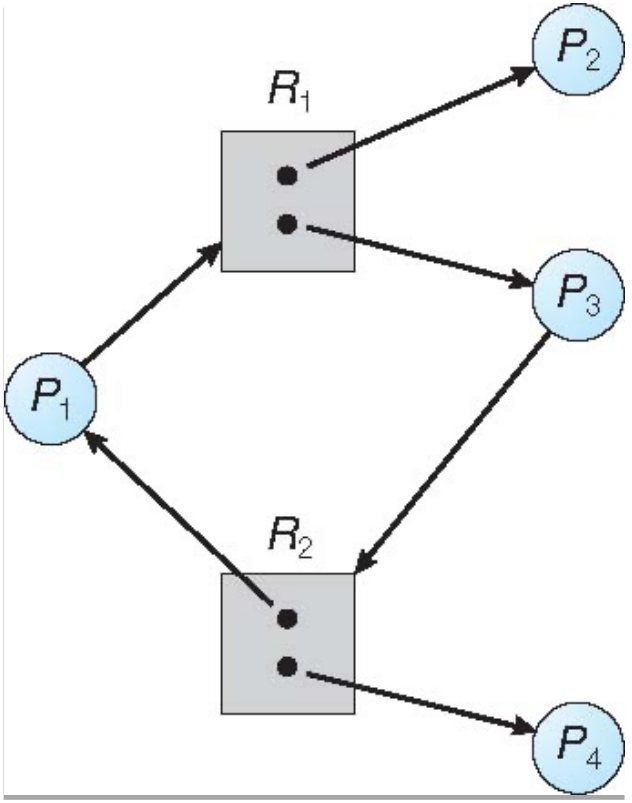


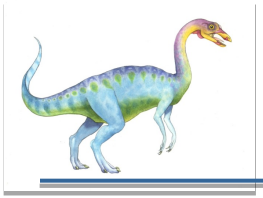


Graph With A Cycle

Here, we also have a cycle:
$P_1 \rightarrow R_1 \rightarrow P_3 \rightarrow R_2 \rightarrow P_1$

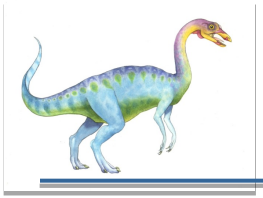
Is there a deadlock?





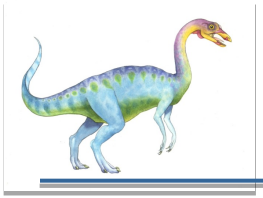
Basic Facts

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



Methods for Handling Deadlocks

- **Methods for Handling Deadlocks:**
  - Ensure that the system will *never* enter a deadlock state:
    - Deadlock prevention
    - 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



Deadlock Prevention

- A deadlock to occur, when each of the four necessary conditions holds.
  1.Mutual exclusion 2.Hold and wait 3. No preemption 4.Circular wait.
- By ensuring that at least one of these conditions **cannot hold**, we can *prevent* the occurrence of a deadlock.

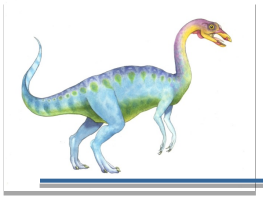
- **Mutual Exclusion** – must hold for non-sharable resources, e.g., a printer
  ➔not required for sharable resources (e.g., read-only files);

- **Hold and Wait** – must guarantee that whenever a process requests a resource, it does not hold any other resources

To achieve that, two protocols:

1. Require a process to request and be allocated all its resources before it begins execution, (no wait)
2. Allow a process to request resources only when the process has none allocated to it. (no hold)

⌦ Low resource utilization; starvation possible



Deadlock Prevention (Cont.)

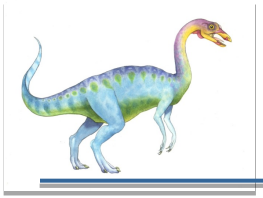
- **No Preemption** –
  - If a process that is holding some resources requests another resource that cannot be immediately allocated to it, then all resources currently being held are released
  - Preempted resources are added to the list of resources for which the process is waiting
  - Process will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

  ⌦ This protocol is often applied to resources whose state can be easily saved and restored later, such as CPU registers and memory space.

- **Circular Wait** – impose a total ordering of all resource types, and require that each process requests resources in an increasing order of enumeration.

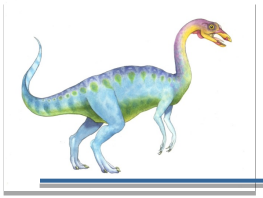
Deadlock Avoidance

- It requires that the operating system be given additional information in advance concerning which resources a process will request and use during its lifetime.
- For example, in a system with one CD drive and one printer, the system might need to know that process *P* will request first the CD drive and then the printer before releasing both resources, whereas process *Q* will request first the printer and then the CD drive. With this knowledge of the complete sequence of requests and releases for each process, the system can decide for each request whether or not the process should wait in order to avoid a possible future deadlock.
- 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

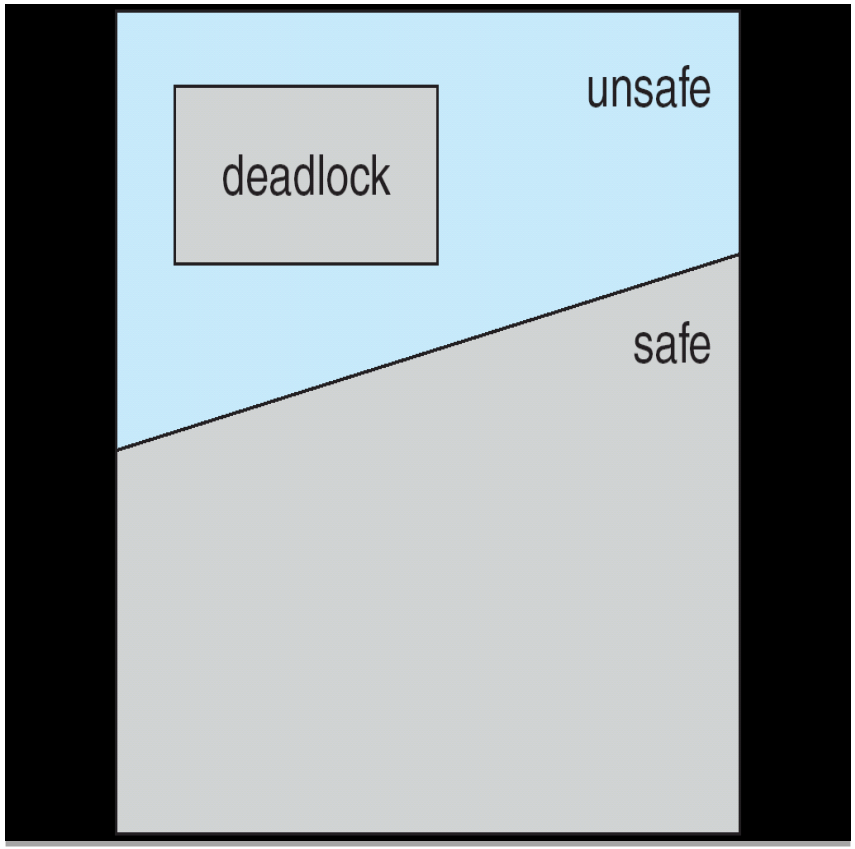
- A state is **safe** if the system can allocate resources to each process (up to its maximum) in some order and still avoid a deadlock.
- System is in **safe state** if there exists a sequence $<P_1, P_2, \ldots, P_n>$ of ALL the processes in the systems such that for each P_i, the resources that P_i can still request can be satisfied by currently available resources + resources held by all the P_j, with $j < i$
- That is:
  - If P_i resource needs are not immediately available, then P_i can wait until all P_j have finished
  - When P_j is finished, P_i can obtain needed resources, execute, return allocated resources, and terminate
  - When P_i terminates, P_{i+1} can obtain its needed resources, and so on
  - If no such sequence exists, then the system state is said to be ***unsafe.***

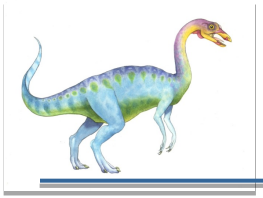


Basic Facts

- 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**.
- Initially, the system is in a **safe state**.
  - Whenever a process requests a resource that is currently available, the system must decide whether the resource can be allocated immediately or whether the process must wait.
  - The request is granted only if the allocation leaves the system in a safe state.

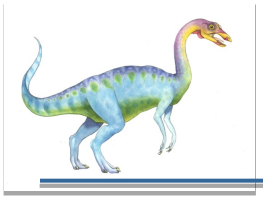
Safe, Unsafe, Deadlock State





Avoidance Algorithms

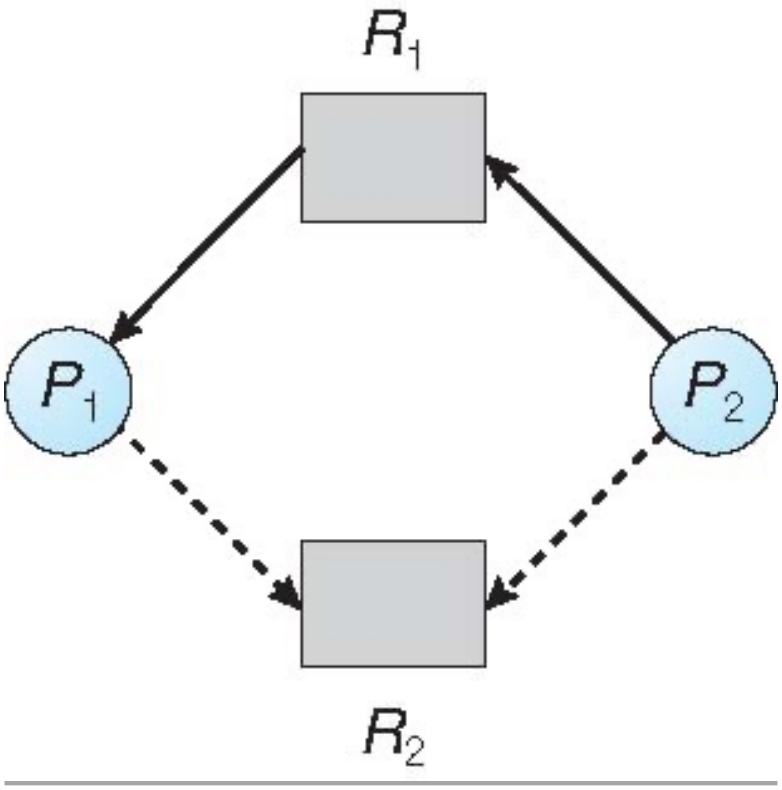
- Two algorithms:
  - Single instance of a resource type
    - Use a resource-allocation graph algorithm.
  - Multiple instances of a resource type
    - Use the banker's algorithm

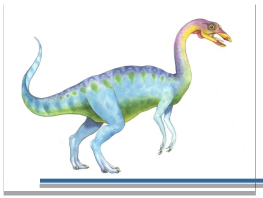


Resource-Allocation Graph Scheme

- **Single resource instance.**
- **Claim edge** $P_i \dashrightarrow R_j$ 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
  - Request edge converted to an assignment edge when the resource is allocated to the process
  - When a resource is released by a process, assignment edge reconverts to a claim edge again.
- Resources must be claimed *a priori* in the system

Resource-Allocation Graph

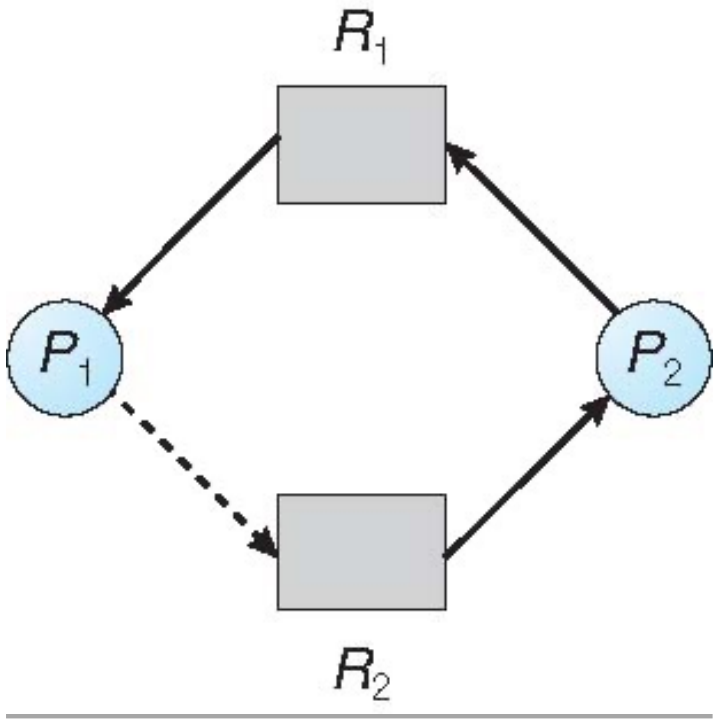




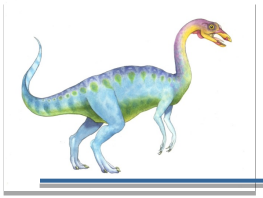
Resource-Allocation Graph Algorithm

- Suppose that process P_i requests a resource R_j
- The request can be granted only if converting the request edge to an assignment edge does not result in the formation of a cycle in the resource allocation graph

Unsafe State In Resource-Allocation Graph

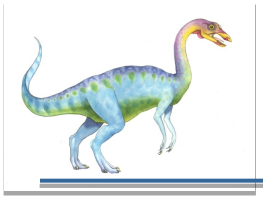


Although R_2 is currently free, we cannot allocate it to P_2, since this action will create a cycle in the graph



Banker's Algorithm

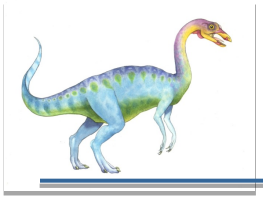
- Multiple resource instances
- Each process must a priori claim maximum use
- When a process requests a resource:
  - When a process requests a set of resources, the system must determine whether the allocation of these resources will leave the system in a safe state. If it will, the resources are allocated.
  - otherwise, the process must wait until some other process releases enough resources.
- When a process gets all its resources it must return them in a finite amount of time
- This algorithms is composed of two sub algorithms:
  - Safety algorithm (finding out whether or not a system is in a safe state)
  - Resource-Request Algorithm (determining whether requests can be safely granted).



Data Structures for the Banker's Algorithm

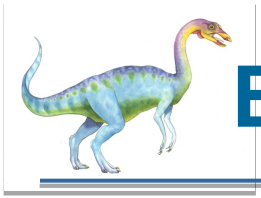
- Let n = number of processes, and m = number of resources types.
- Several data structures must be maintained to implement the banker's algorithm
  - **Available**: Vector of length m. If available [j] = k, there are k instances of resource type R_j available
  - **Max**: n x m matrix. If Max [i,j] = 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 (indicates the remaining resource need of each process).

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



Data Structures for the Banker's Algorithm

- ***For simplicity:***
  - ***Allocation***$_i$ → *a vector that* specifies the resources currently allocated to process P_i
  - ***Need***$_i$ → *a vector that* specifies the additional resources that process P_i may still request to complete its task.



Banker's Algorithm – Safety Algorithm

1. Let ***Work*** and ***Finish*** be vectors of length *m* and *n*, respectively. Initialize:
   Work* = *Available
   ***Finish* [*i*] = *false* for *i* = 0, 1, …, *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

Banker's Algorithm – Resource-Request Algorithm for Process P_i

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

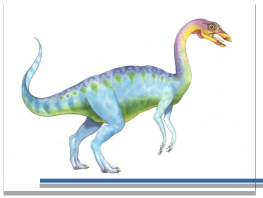
1. If $Request_i \leq Need_i$ go to step 2. Otherwise, raise error condition, since process has exceeded its maximum claim

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:

$$Available = Available - Request_i;$$
$$Allocation_i = Allocation_i + Request_i;$$
$$Need_i = Need_i - Request_i;$$

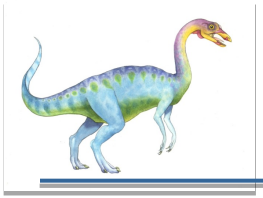
- If safe → the resources are allocated to P_i
- If unsafe → P_i must wait, and the old resource-allocation state is restored



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	

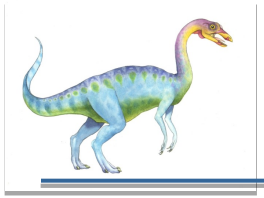


Example (Cont.)

- 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



Solution

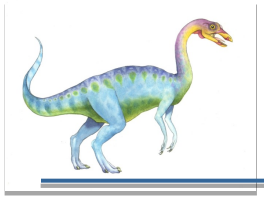
Step 1	m=3, n=5				
Work = Available					
Work =	3		3		2
Process	0	1	2	3	4
Finish =	false	false	false	false	false

Step 2	P_0	For i = 0	
$Need_0$ =	7	4	3
		7,4,3 > 3,3,2	
Finish [0] = false and $Need_0$ > Work			
P_0 must wait			

Step 2	P_1	For i = 1	
$Need_1$ =	1	2	2
		1,2,2 < 3,3,2	
Finish [1] = false and $Need_1$ < Work			
P_1 can be kept in safe sequence			

Step 2	P_2	For i = 2	
$Need_2$ =	6	0	0
		6,0,0 > 5,3,2	
Finish [2] = false and $Need_2$ > Work			
P_2 must wait			

Step 3	P_1				
Work =	Work + Allocation				
	3,3,2 + 2,0,0				
	5		3		2
Process	0	1	2	3	4
Finish =	false	true	false	false	false



Cont.

Step 2	P_3	For i = 3	
$Need_3$ =	0	1	1
		0,1,1 <	5,3,2
Finish [3] = false and $Need_3$ < Work			
P_3 can be kept in safe sequence			

Step 3	P_3				
Work =	Work + Allocation				
	5,3,2 + 2,1,1				
	7		4		3
Process	0	1	2	3	4
Finish =	false	true	false	true	false

Step 2	P_4	For i = 4	
$Need_4$ =	4	3	1
		4,3,1 <	7,4,3
Finish [4] = false and $Need_4$ < Work			
P_4 can be kept in safe sequence			

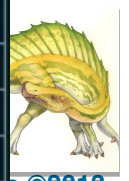
Step 3	P_4				
Work =	Work + Allocation				
	7,4,3 + 0,0,2				
	7		4		5
Process	0	1	2	3	4
Finish =	false	true	false	true	true

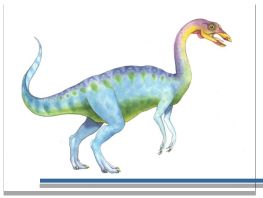
Step 2	P_0	For i = 0	
$Need_0$ =	7	4	3
		7,4,3 <	7,4,5
Finish [0] = false and $Need_0$ < Work			
P_0 can be kept in safe sequence			

Step 3	P_0				
Work =	Work + Allocation				
	7,4,5 + 0,1,0				
	7		5		5
Process	0	1	2	3	4
Finish =	true	true	false	true	true

Step 2	P_2	For i = 2	
$Need_2$ =	6	0	0
		6,0,0 <	7,5,5
Finish [0] = false and $Need_2$ < Work			
P_2 can be kept in safe sequence			

Step 3	P_2				
Work =	Work + Allocation				
	7,5,5 + 3,0,2				
	10		5		7
Process	0	1	2	3	4
Finish =	true	true	true	true	true





Example: P_1 Request (1,0,2)

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

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

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

End of Chapter 7