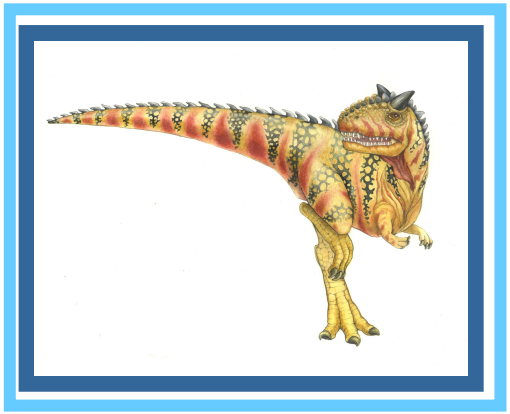
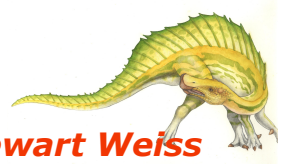
Chapter 7: Deadlocks



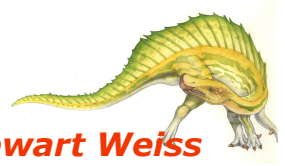
Chapter 7: Deadlocks

- The Deadlock Problem
- System Model
- Deadlock Characterization
- Methods for Handling Deadlocks
- Deadlock Prevention
- Deadlock Avoidance (briefly)
- Deadlock Detection (briefly)



Chapter Objectives

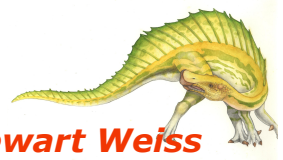
- To develop a description of deadlocks, which prevent sets of concurrent processes from completing their tasks
- To present a number of different methods for preventing or avoiding deadlocks in a computer system



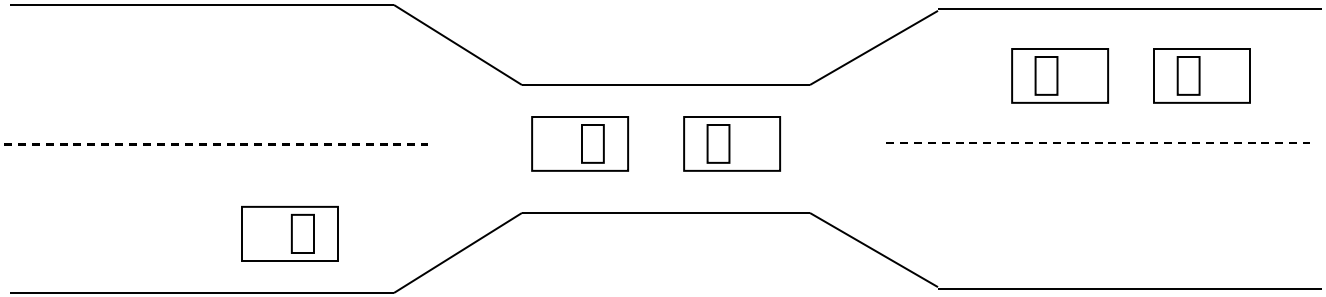
The Deadlock Problem

- 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 disk drives
  - P_1 and P_2 each hold one disk drive and each needs another one
- Example
  - semaphores *A* and *B*, initialized to 1

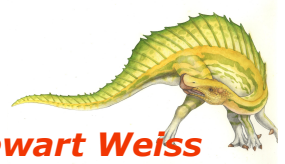
P_0	P_1
wait (A);	wait(B)
wait (B);	wait(A)



Bridge Crossing Example

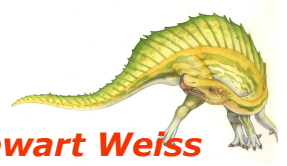


- Traffic only in one direction
- Each section of a bridge can be viewed as a resource
- If a deadlock occurs, it can be resolved if one car backs up (preempt resources and rollback)
- Several cars may have to be backed up if a deadlock occurs
- Starvation is possible
- Note – Most OSes do not prevent or deal with deadlocks



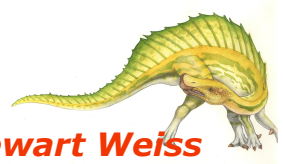
System Model

- An abstract model is used to better understand deadlock. In this model, a system consists of:
  - A finite set of m resource types, R_1, R_2, . . ., R_m
    - A resource type is a collection of identical, interchangeable units: e.g. memory blocks, printers, disk blocks
    - Each resource type R_i has W_i instances.
  - A finite set of processes,
  - A set of rules by which processes use resources.



Acquiring Resources

- To use a resource a process must:
  - request the resource, and if it is not available, it waits
  - acquire and use the resource
  - release the resource
- Steps 1 and 3 are usually system calls. The operating system must verify and service these.



Necessary Conditions for Deadlock

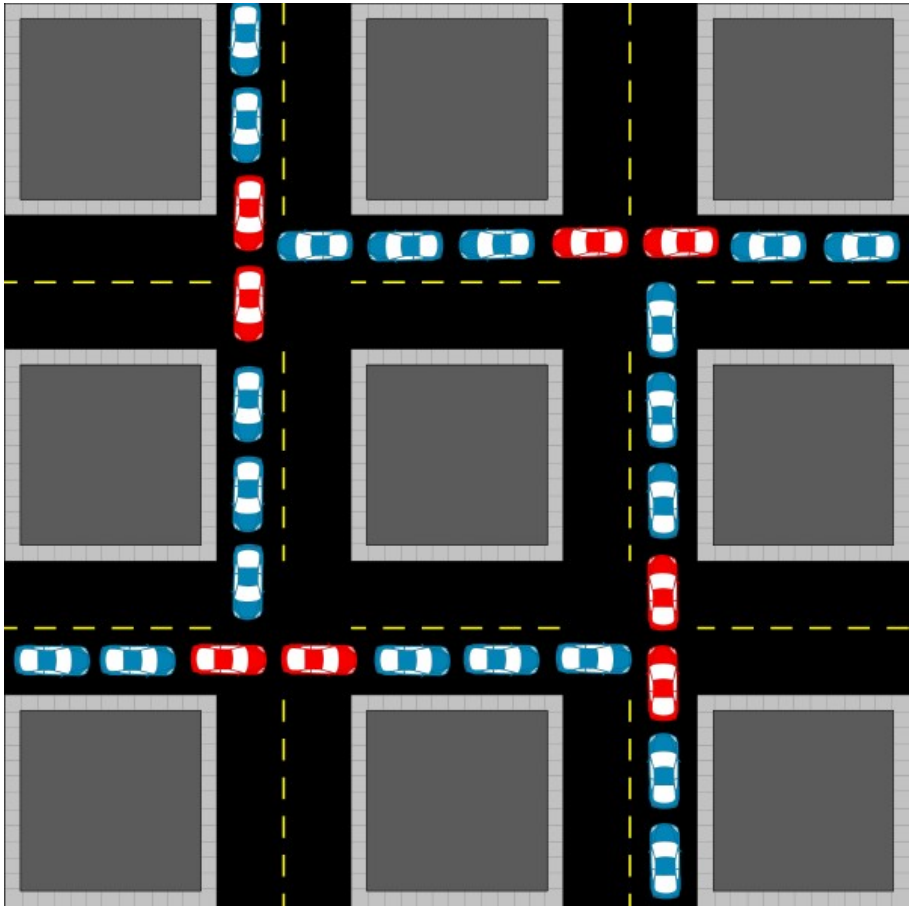
Deadlock can arise only if four conditions hold simultaneously.

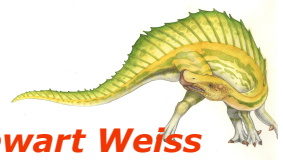
- **Mutual exclusion:** At least one resource is held in nonsharable mode.
- **Hold and wait:** There is a process that holds one resource and is waiting to acquire additional resources held by other processes
- **No preemption:** A resource can be released only voluntarily by the process holding it, after that process has finished using it.
- **Circular wait:** There exists a set $\{P_0, P_1, \ldots, P_0\}$ 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.

Gridlock Example

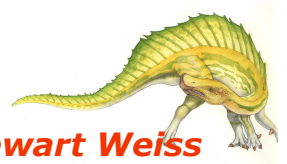
- Gridlock as an example of deadlock
- The shared resource type is the intersection; it has units the size of a car-length.
- Verify the four conditions:
  - mutual exclusion,
  - hold-and-wait,
  - non-preemptive allocation,
  - circular waiting





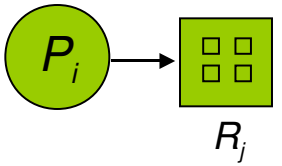
Resource Allocation Graph

- ***A Resource allocation graph*** is a mathematical abstraction to help reasoning about deadlock and resource allocation policies.
- It is a directed graph G = <V,E> in which V, the set of vertices, is partitioned into two sets:
  - $P = \{P_1, P_2, \ldots, P_n\}$, 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
- and E, the edge set, is partitioned into two types of edges:
  - request edge – directed edge $P_1 \rightarrow R_j$ *means process has requested resource*
  - assignment edge – directed edge $R_j \rightarrow P_i$ *resource is held by process*

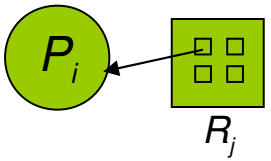


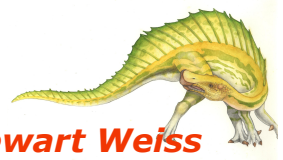
Resource-Allocation Graph (Cont.)

- Process is drawn as a circle 
- Resource Type is drawn as a square with instances inside: e.g. 4 instances 
- *A request edge:* P_i requests one instance of R_j

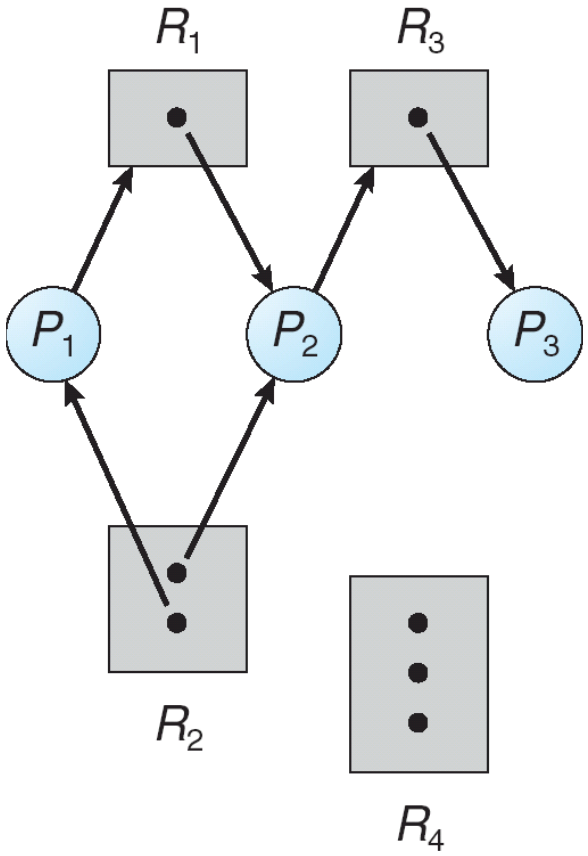


- *An assignment edge:* P_i is holding an instance of R_j

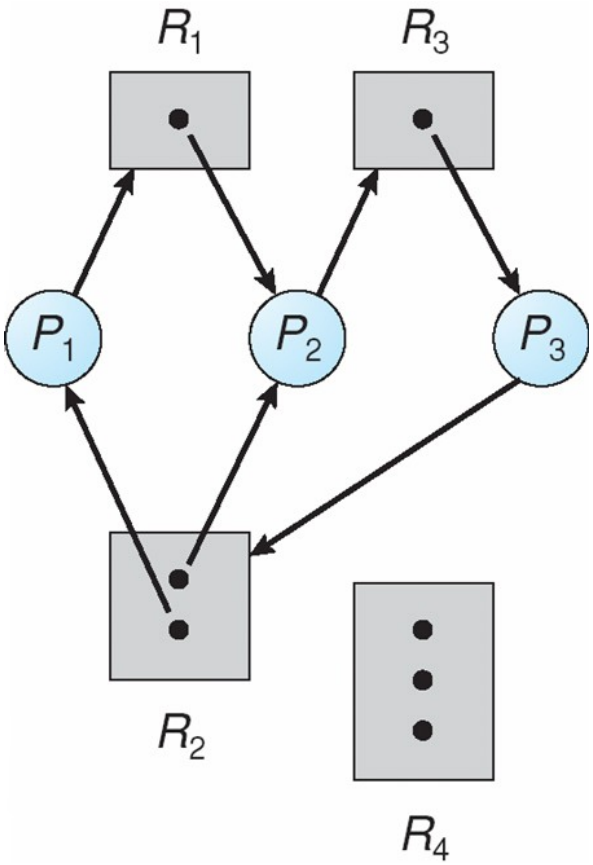




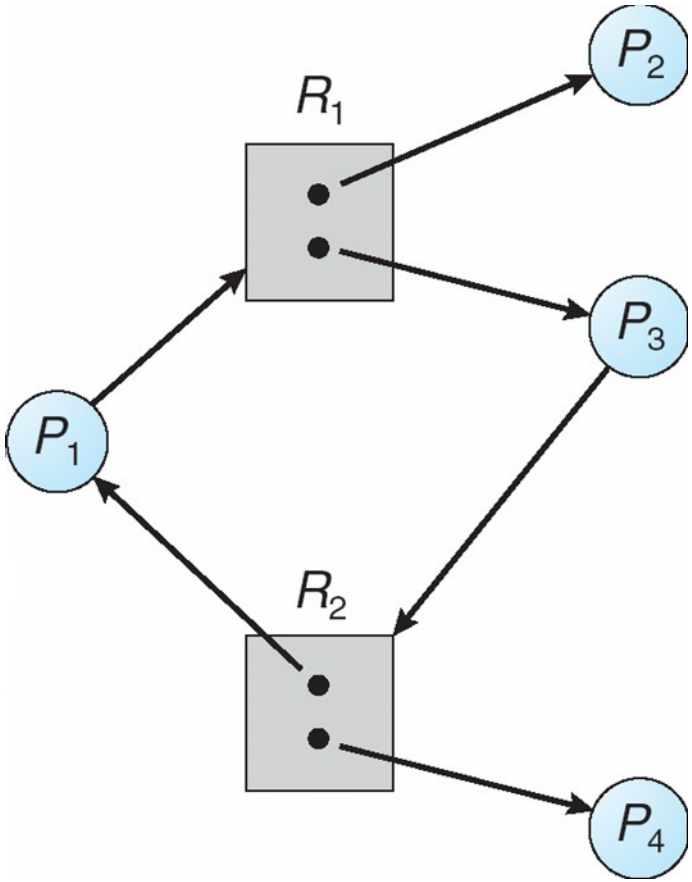
Example of a Resource Allocation Graph



Resource Allocation Graph With A Deadlock

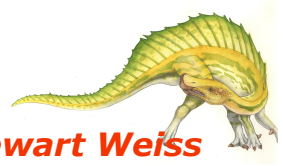


Graph With A Cycle But No Deadlock



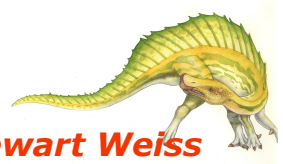
Basic Facts

- If graph contains no cycles, the state that it represents is not a deadlock state.
- If graph contains a cycle, then
  - if each resource type has only one instance, then the state is a deadlock state
  - if there is a resource with multiple instances, it may not be a deadlock state.



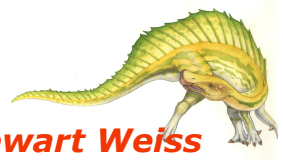
Methods for Handling Deadlocks

- Ensure that the system will *never* enter a deadlock state
- 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



Preventing and Avoiding Deadlock

- Two ways to ensure that the system will *never* enter a deadlock state:
- Preventing deadlock by guaranteeing that one of the four necessary conditions does not hold. They all work by restricting resource requests in one way or another. These are called ***prevention methods***.
- Avoiding deadlock without restricting requests, but instead by dynamically determining when it is safe to grant resources to processes that request them. These are called ***avoidance methods***.

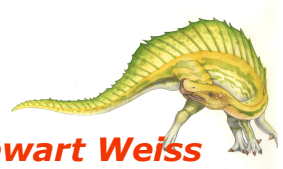


Deadlock Prevention

•Deadlock prevention restrains the ways that requests can be made.

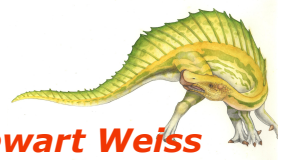
- **Mutual Exclusion** – Cannot in general remove this condition because some resources are inherently non-sharable and must be used in mutual exclusion. (printers, disk drives, tape drives, etc.)

- **Hold and Wait** – To remove this condition means that a process cannot hold any resource if it is waiting for another. Therefore, it implies that whenever a process requests a resource, it does not hold any other resources. Only two possibilities:
  - Require process to ***request and be allocated all of its resources*** before it begins execution, or
  - Allow process to request resources ***only when the process has none***
  - Problems: poor resource utilization; starvation possible



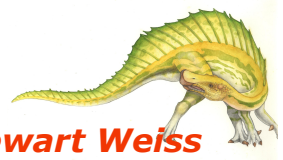
Deadlock Prevention (Cont.)

- **No Preemption** – Removing this implies being able to forcibly take away resource from a process. Two possibilities:
  - 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, and 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
  - If a process requests resources that are unavailable but held by a process waiting for other resources, the requested resources are taken away from the waiting process and given to the one that requested them. If the resources are held by a ready process, the requesting process has to wait instead (and may lose resources it currently holds if another process requests them.)
  - Used for resources whose state can be recovered such as CPU, memory.



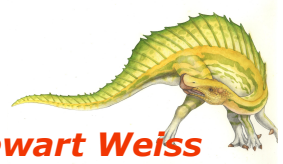
Deadlock Prevention (Cont.)

- **Circular Wait** – To remove this condition implies that a linear (i.e., total) order must be imposed on all resource types, that processes must request resources in an increasing order of enumeration.
  - Havender's scheme: Assume resources are ordered in some way:
    - R1, R2, R3, ...Rm
    - A process can request resources only in increasing order of subscript: if a process holds Ri, it can only request Rj if j > i.
    - If a process wants Rj, it must release all Ri for which i >= j.
    - This eliminates circular waiting.



Deadlock Avoidance

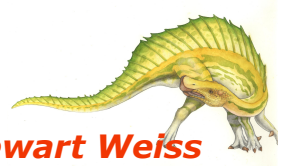
- In prevention schemes, requests are restricted. Result is poor utilization and throughput.
- In ***deadlock avoidance***, all requests are allowed, but operating system decides whether to grant them based on additional information.
  - Simplest and most useful model requires that each process declare 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.
- Dijkstra's ***Banker's Algorithm*** is most well known deadlock avoidance method, and ituses this information.



Safe State

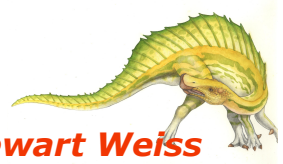
- When a process requests an available resource, system must decide if immediate allocation leaves the system in a safe state

- System is in safe state if there exists a sequence $<P_1, P_2, \ldots, P_n>$ of ALL the processes in the systems such that for each P_i, the resources that P_i can still request can be satisfied by currently available resources + resources held by all the P_j, with $j < i$
- That is:
  - If P_i resource needs are not immediately available, then P_i can wait until all P_j have finished
  - When P_j is finished, P_i can obtain needed resources, execute, return allocated resources, and terminate
  - When P_i terminates, P_{i+1} can obtain its needed resources, and so on

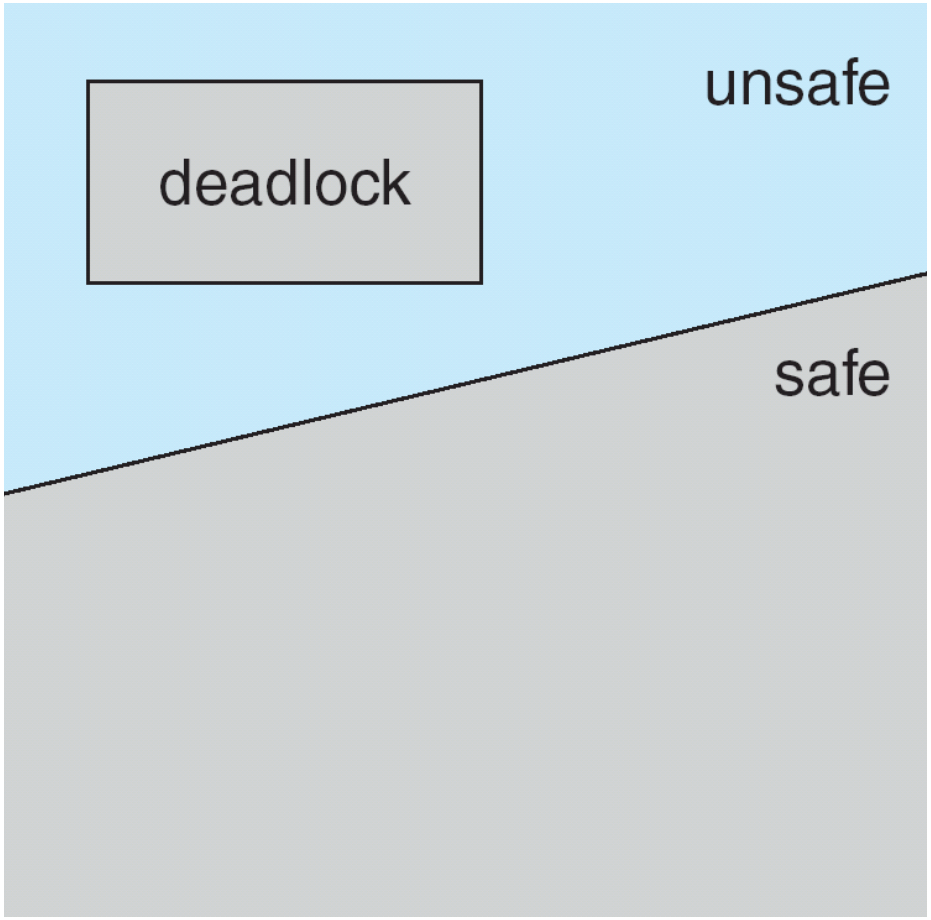


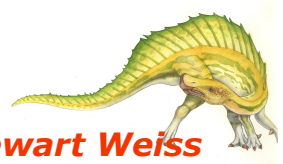
Basic Facts

- If a system is in safe state then deadlock is impossible, because over time all needed resources can be given to processes that have requested them.
- If a system is in unsafe state then there is possibility of deadlock
- Avoidance methods ensure that a system will never enter an unsafe state.



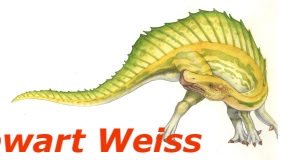
Safe, Unsafe , Deadlock State





Banker's Algorithm

- Works when resource types have multiple instances.
- Each process must a priori declare its maximum resource requirements for each resource type
- When a process requests a resource it may have to wait until the request can be granted to make the state safe
- When a process gets all its resources it must return them in a finite amount of time
- Idea: when a process requests resource, if units are available, pretend to allocate and see if resulting state is safe; if safe, allocate; if not, make process wait.

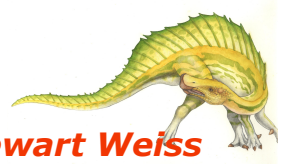


Data Structures for the Banker's Algorithm

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

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



Safety Algorithm

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

   Work[i] = Available[i] for 1 <= i <= m

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

2. Find *i* such that both:

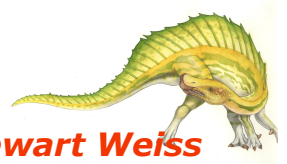
   (a) Finish [i] = false

   (b) Need[i] <= Work[i]

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

3. Work[i] = Work[i] + 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, otherwise unsafe



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

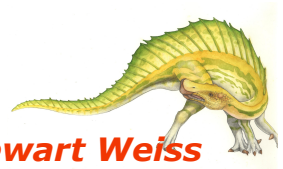
1. If $Request_i <= Need_i$ go to step 2. Otherwise, raise error condition, since process has exceeded its maximum claim
2. If $Request_i <= Available$, go to step 3. Otherwise P_i must wait, since resources are not available
3. Pretend to allocate requested resources to P_i by modifying the state as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

$$Available = Available - Request;$$

$$Allocation_i = Allocation_i + Request_i;$$

$$Need_i = Need_i - Request_i;$$

- *Check if this state is safe; If safe then the resources are allocated to Pi*
- *If unsafe then Pi must wait, and the old resource-allocation state is restored*



Example of Banker's Algorithm

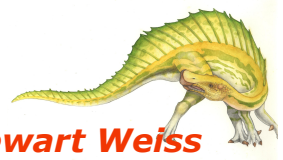
- 5 processes P_0 through P_4;

  3 resource types:

  A (10 instances), B (5instances), and C (7 instances)

Snapshot at time T_0:

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

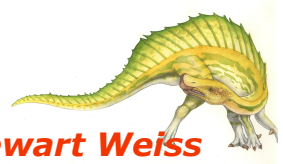


Example (Cont.)

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

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

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

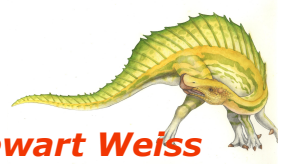


Example: P_1 Request (1,0,2)

- Check that Request <= Available (that is, (1,0,2) <= (3,3,2) is true

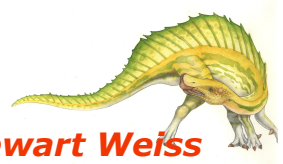
	Allocation	*Need*	*Available*
	A B C		*A B C*
P_0	0 1 0	7 4 3	2 3 0
P_1	3 0 2	0 2 0	
P_2	3 0 1	6 0 0	
P_3	2 1 1	0 1 1	
P_4	0 0 2	4 3 1	

- 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?



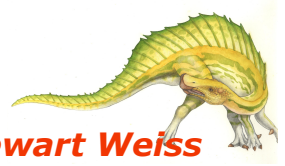
Deadlock Detection

- Allow system to enter deadlock state then check if it is a deadlock state
- Implies that system must continuously update required data structures and check whether state is deadlock state: large overhead.
- Systems tend not to do it because overhead is so great.



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