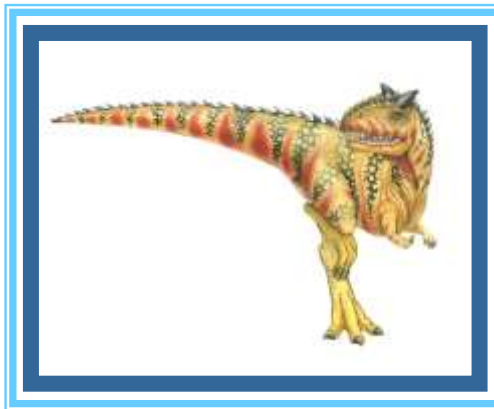


# OS-3

---





# System Model

---

- System consists of resources
- Resource types  $R_1, R_2, \dots, R_m$ 
  - 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**

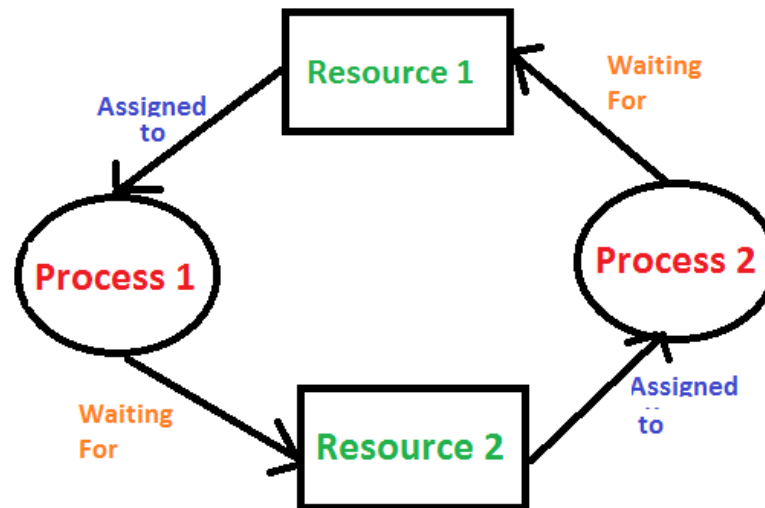




# What is deadlock

**Deadlock** is a situation where a set of processes are blocked because each process is holding a resource and waiting for another resource acquired by some other process.

Consider an example when two trains are coming toward each other on same track and there is only one track, none of the trains can move once they are in front of each other.





# Deadlock Characterization

Deadlock can arise if four conditions hold simultaneously.

- **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  $\{P_0, P_1, \dots, P_n\}$  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  $P_0$ .





# Resource-Allocation Graph

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

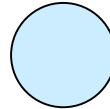
- $V$  is partitioned into two types:
  - $P = \{P_1, P_2, \dots, P_n\}$ , the set consisting of all the processes in the system
  - $R = \{R_1, R_2, \dots, R_m\}$ , the set consisting of all resource types in the system
- **request edge** – directed edge  $P_i \rightarrow R_j$
- **assignment edge**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





# Resource-Allocation Graph (Cont.)

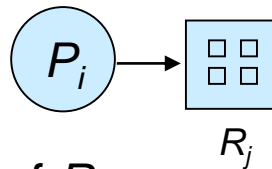
- Process



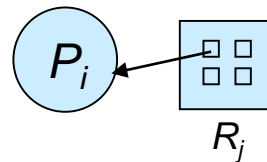
- Resource Type with 4 instances



- $P_i$  requests instance of  $R_j$

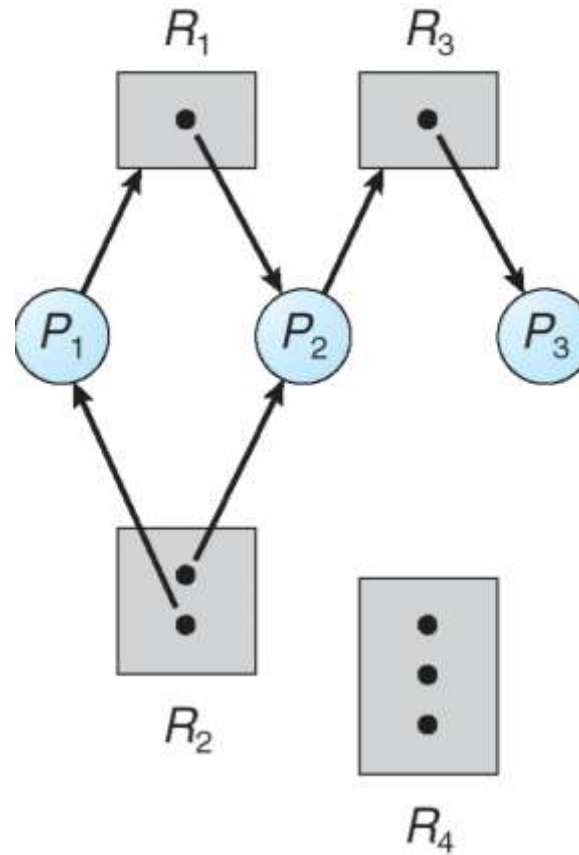


- $P_i$  is holding an instance of  $R_j$



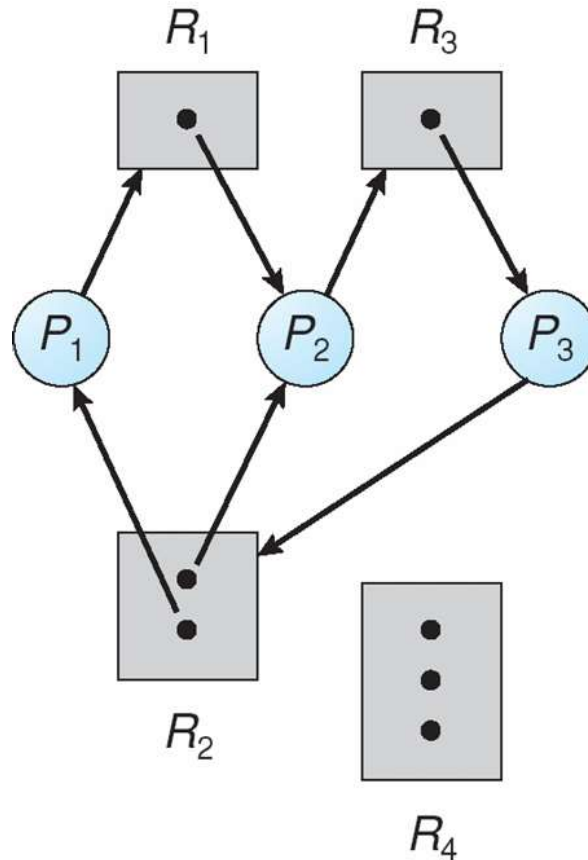


# Example of a Resource Allocation Graph





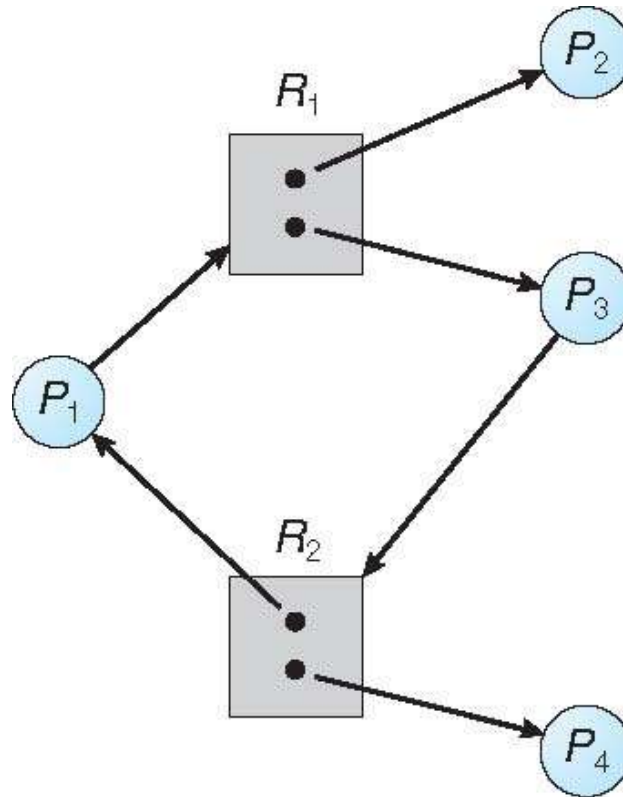
# Resource Allocation Graph With A Deadlock







# Graph With A Cycle But No Deadlock

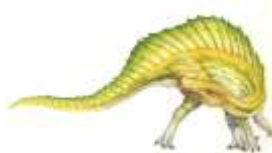




# Basic Facts

---

- 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:
  - 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

---

Restrain the ways request can be made

- **Mutual Exclusion** – not required for sharable resources (e.g., read-only files); must hold for non-sharable 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, or allow process to request resources only when the process has none allocated to it.
  - 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

## ■ 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

---

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





# Basic Facts

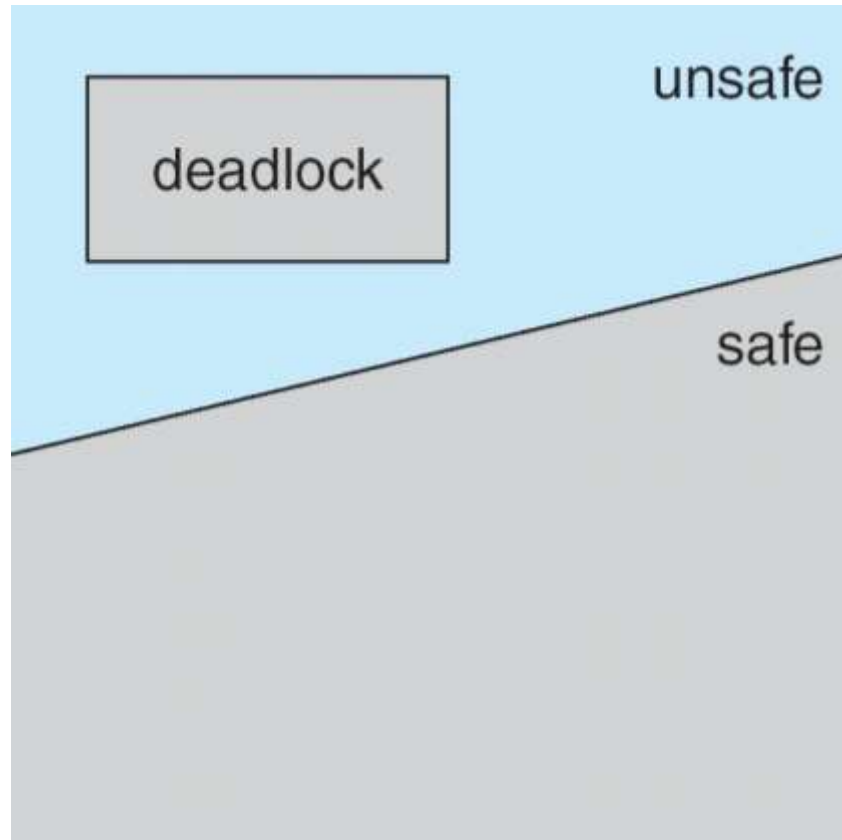
---

- 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







# Banker's Algorithm

---

It is a deadlock avoidance algorithm, this algorithm is applied when there are multiple instances of a resource type.

- Multiple instances
- Each process must a priori 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





# 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

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





## Example (Cont.)

Safety Algorithm

if,  
     $need \leq available$

Then,  
    Executes process  
    New  $available = available + allocation$

Else,  
    do not execute go forward





# 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  $T_0$ :

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

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

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





## Example (Cont.)

Total:  $A = 10, B = 5, C = 7$

Process	Allocation			Max Need			Available			Need = (Max - Allocation)		
	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	5	3	2	1	2	2
$P_2$	3	0	2	9	0	2	7	4	3	6	0	0
$P_3$	2	1	1	4	2	2	7	4	5	2	1	1
$P_4$	0	0	2	5	3	3	7	5	5	5	3	1
	7	2	5				10	5	7			

$$A = (10 - 7) = 3; B = (5 - 2) = 3; C = (7 - 5) = 2$$

$\therefore$  Safe Sequence :  $\{P_1, P_3, P_4, P_0, P_2\}$

• After  $P_1$  execution, Available resources =  $(3+2), (0+3), (0+2)$   
 $= (5, 3, 2)$





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





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

- Check that Request  $\leq$  Available (that is,  $(1,0,2) \leq (3,3,2) \Rightarrow$  true

	<u>Allocation</u>	<u>Need</u>	<u>Available</u>
	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  $\langle P_1, P_3, P_4, P_0, P_2 \rangle$  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?







# 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  $\langle P_1, P_2, \dots, P_n \rangle$  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





# Safety Algorithm

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

**Work = Available**

**Finish** [ $i$ ] = **false** for  $i = 0, 1, \dots, n-1$

2. Find an  $i$  such that both:

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

(b) **Need** <sub>$i$</sub>  ≤ **Work**

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

3. **Work = Work + Allocation** <sub>$i$</sub>   
**Finish** [ $i$ ] = **true**  
go to step 2

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

