OPERATING SYSTEM (DEAD LOCK)

By

Amar Panchal

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*

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

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 $\{P_0, P_1, \ldots, P_n\}$ 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.

Handlind Deadlock

- Detection
- Avoidance
- Prevention

DEADLOCK PREVENTION VERSUS DEADLOCK AVOIDANCE

DEADLOCK PREVENTION	DEADLOCK AVOIDANCE
Mechanism to ensure that at least one of the necessary conditions for deadlock can never occur	Mechanism to ensure that the system does not enter an unsafe state
System does not require information on the existing resources, resource availability and resource requests	System requires information on the existing resources, resource availability and resource requests to find whether the system is in safe or unsafe state
Non-blocking synchronization algorithms and serializing tokens are some deadlock prevention algorithms	Banker's algorithm is the most common deadlock avoidance algorithm
All resources are requested at once	Requests for resources are manipulated until at least one safe path is found

Visit www.PEDIAA.com

Prevention

- Collective Request
  - *Give one take other*
- Ordered Request
  - *Take only in increasing order*
- Preemption
  - *If block leave resouces and take when u start*

Avoidance

- Banker's Algorithm

Banker's Algorithm

• Banker's Algorithm is used to determine whether a process's request for allocation of resources be safely granted immediately.

or

• The grant of request be deferred to a later stage.

• For the banker's algorithm to operate, each process has to a priori specify its maximum requirement of resources.

• A process is admitted for execution only if its maximum requirement of resources is within the system capacity of resources.

•The Banker's algorithm is an example of resource allocation policy that avoids deadlock.

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

Example:- Consider the following table of a system:

Process	Allocated				Max				Available			
	R1	R2	R3	R4	R1	R2	R3	R4	R 1	R 2	R 3	R 4
P1	0	0	1	2	0	0	1	2	2	1	0	0
P2	2	0	0	0	2	7	5	0				
P3	0	0	3	4	6	6	5	0				
P4	2	3	5	4	4	3	5	6				
P5	0	3	3	2	0	6	5	2				

1. Compute NEED Matrix.
2. Is the system in safe state? Justify.

Solution:- Consider the following table of the system:

Process	Allocated				Max				Available			
	R1	R2	R3	R4	R1	R2	R3	R4	R1	R2	R3	R4
P1	0	0	1	2	0	0	1	2	2	1	0	0
P2	2	0	0	0	2	7	5	0				
P3	0	0	3	4	6	6	5	0				
P4	2	3	5	4	4	3	5	6				
P5	0	3	3	2	0	6	5	2				

1. **Compute NEED Matrix = ?**

Need [i] = Max[i] - Allocated[i],

Therefore,

Need Matrix

NEED MATRIX	R1	R2	R3	R4
P1	0	0	0	0
P2	0	7	5	0
P3	6	6	2	2
P4	2	0	0	0
P5	0	3	2	0

2. Is the system is Safe State?

By applying the Banker's Algorithm:

Let **Avail** = Available; i.e . Avail = {2,1,0,0}

Iteration 1. Check all processes from P1 to P5.

For P1:→

if (**P1 Need < Avail**)→TRUE

then calculate

Avail= Avail + Allocated [P1]

= {2,1,0,0} + = {0,0,1,2}

Avail = **{2,1,1,2}**

2. Is the system is Safe State?

By applying the Banker's Algorithm:

Iteration 1.

For P2:→

if (**P2 Need < Avail**)→FALSE

//then Check for next process.

For P3:→

if (**P3 Need < Avail**)→ FALSE

//then Check for next process.

2. Is the system is Safe State?

<u>By applying the Banker's Algorithm:</u>

Iteration 1.

<u>For P4:→</u>

if (**P4 Need < Avail**)→TRUE

then calculate

Avail= Avail + Allocated [P4]

= {2,1,1,2} + = {2,3,5,4}

Avail = **{4,4,6,6}**

2. Is the system is Safe State?

<u>By applying the Banker's Algorithm:</u>

Iteration 1.

<u>For P5:→</u>

if (**P5 Need < Avail**)→TRUE

then calculate

Avail= Avail + Allocated [P5]

= {4,4,6,6} + = {0,3,3,2}

Avail = **{4,7,9,8}**

2. Is the system is Safe State?

<u>By applying the Banker's Algorithm:</u>

Iteration 2. Check only process P2 to P3.

<u>For P2:→</u>

if (**P2 Need < Avail**)→TRUE

then calculate

Avail= Avail + Allocated [P2]

= {4,7,9,8} + = {2,0,0,0}

Avail = **{6,7,9,8}**

2. Is the system is Safe State?

<u>By applying the Banker's Algorithm:</u>

Iteration 2. Check only process P2 to P3.

<u>For P3:→</u>

if (**P3 Need < Avail**)→TRUE

then calculate

Avail= Avail + Allocated [P3]

= {6,7,9,8} + = {0,0,3,4}

Avail = **{6,7,12,12} =System Capacity**

Since, all the processes got TRUE marked, no further iterations are required.

Therefore, **Safe Sequence = P1, P4, P5, P2 , P3**

Therefore, the System is in the Safe State.

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 $Request_i \le Need_i$ go to step 2. Otherwise, raise error condition, since process has exceeded its maximum claim*
2. *If $Request_i \le Available$, 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:*

$$Available = Available - Request;$$

$$Allocation_i = Allocation_i + Request_i;$$

$$Need_i = Need_i - Request_i;$$

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

Dead Lock detection

- Resource Allocation Graph
- Hierarchical Technique
- Fully Distributed

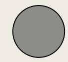
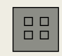
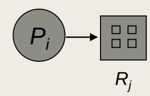
Resource-Allocation Graph

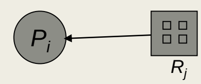
A set of vertices *V* and a set of edges *E*.

- 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 $P_i \rightarrow R_j$
- assignment edge – directed edge $R_j \rightarrow P_i$

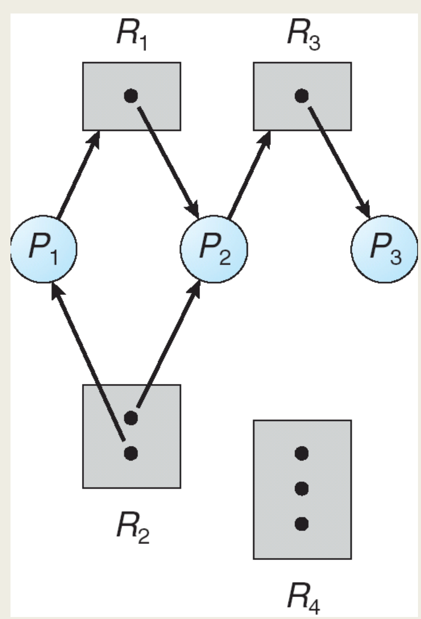
Prof.Amar Panchal Amarpanchal.com 9821601163

Resource-Allocation Graph (Cont.)

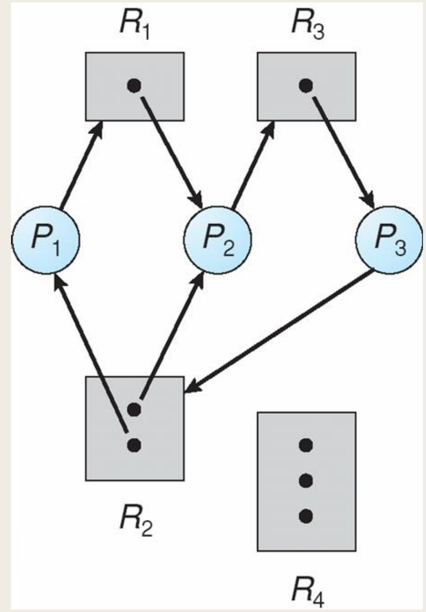
- Process 
- Resource Type with 4 instances 
- P_i requests instance of R_j 

- P_i is holding an instance of R_j 

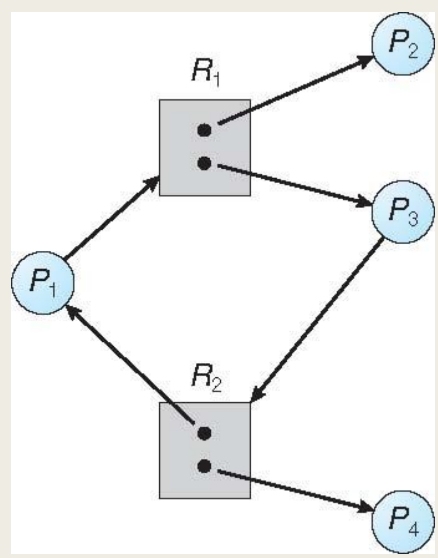
Example of a Resource Allocation Graph



Resource Allocation Graph With A Deadlock



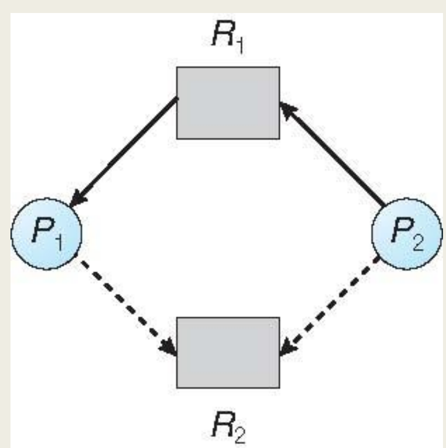
Graph With A Cycle But No Deadlock



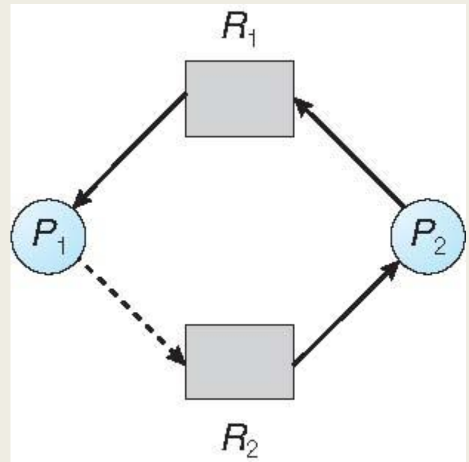
Resource-Allocation Graph Scheme

- **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
- 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
- Resources must be claimed *a priori* in the system

Resource-Allocation Graph



Unsafe State In Resource-Allocation Graph



Deadlock Detection

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