Subject Name: Operating Systems

Unit:4 Unit Name: Deadlocks

Faculty Name: Ms. Puja Padiya

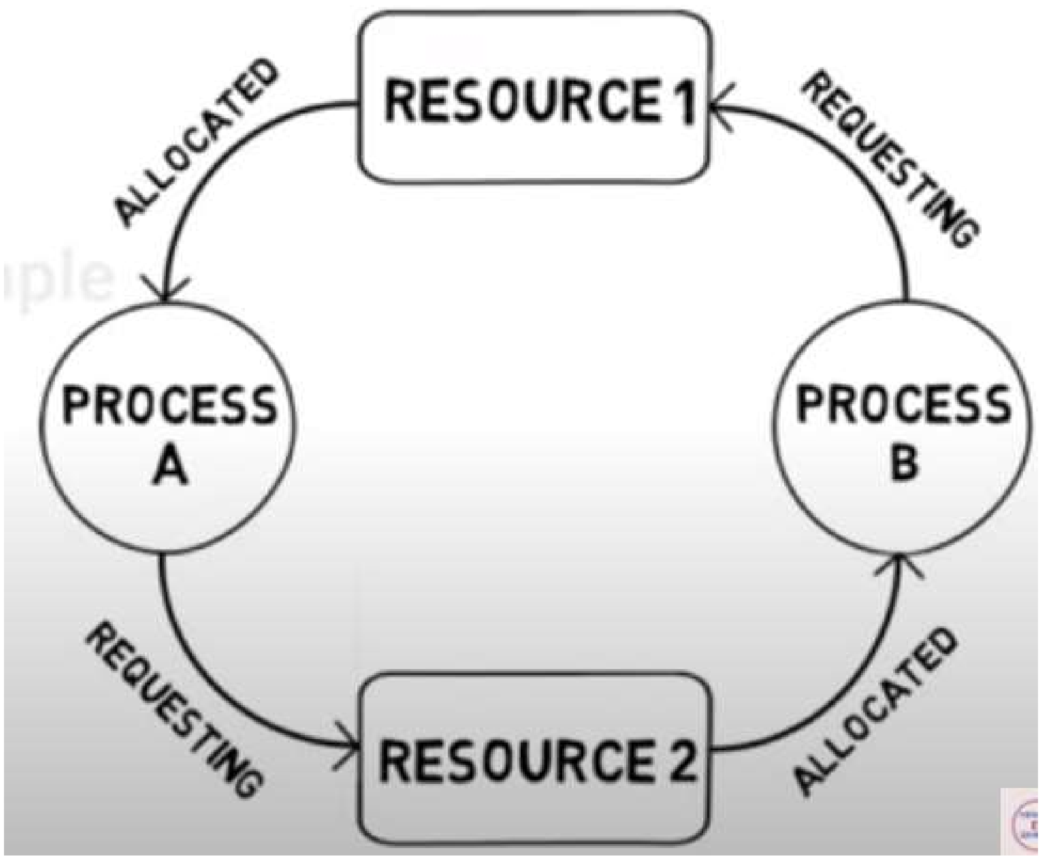
Index

Unit No: 4 **Unit Name: Deadlocks**

Lecture:

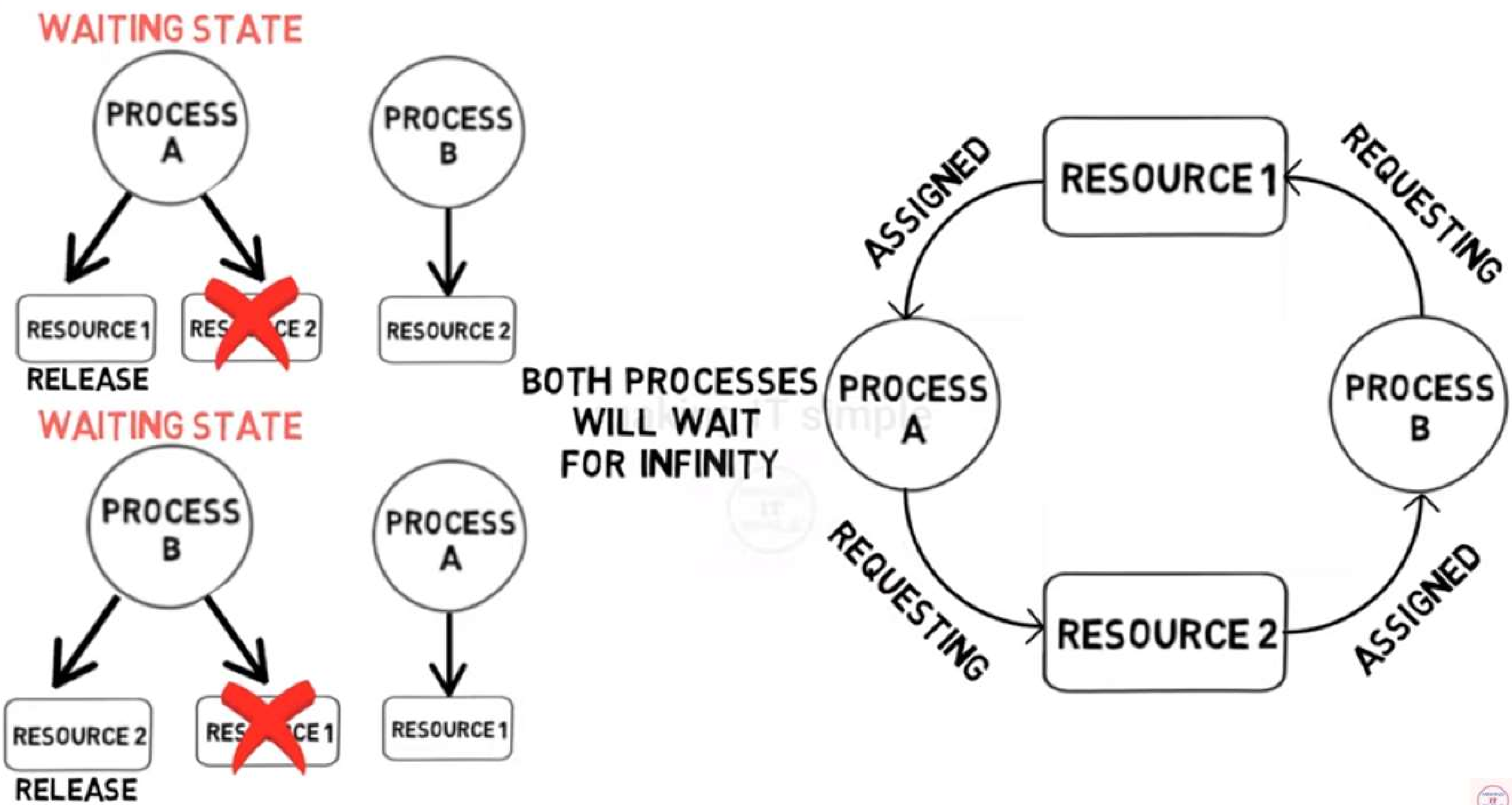
Principal of Deadlock Condition and Resource Allocation Graph

What is Deadlock?



- A deadlock is a situation in which two computer programs sharing the same resource are effectively preventing each other from accessing the resource, resulting in both programs ceasing to function.
- A set of processes is deadlocked when every process in the set is waiting for the resource that is currently allocated to the another process in the set.

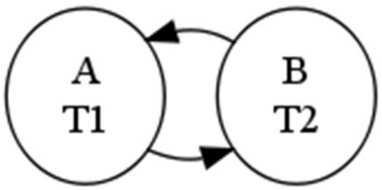
What is Deadlock?



Defination and Basics

Deadlock: A set of blocked processes each *holding* a resource and *waiting* to acquire a resource held by another process in the set.

Example 1



System has 2 disk drives

Processes P1 and P2 each hold one disk drive and each needs another one

Example 2

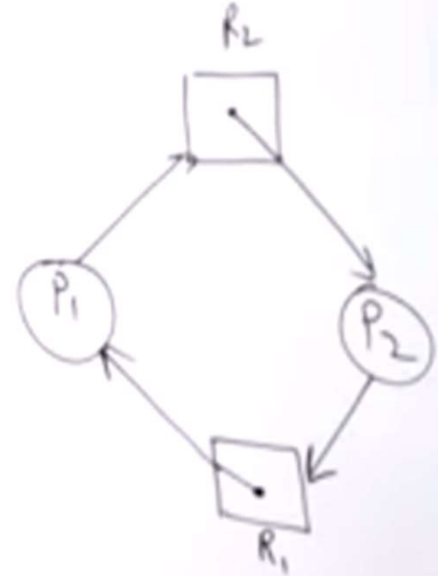
Semaphores **A** and **B**, initialized to 1

P0	**P1**
wait (A);	wait(B)
wait (B);	wait(A)

Dead Lock

→ In a multiprogramming system, a number of process compete for limitted number of resources and if a resource is not available at that instance then process enters into waiting state

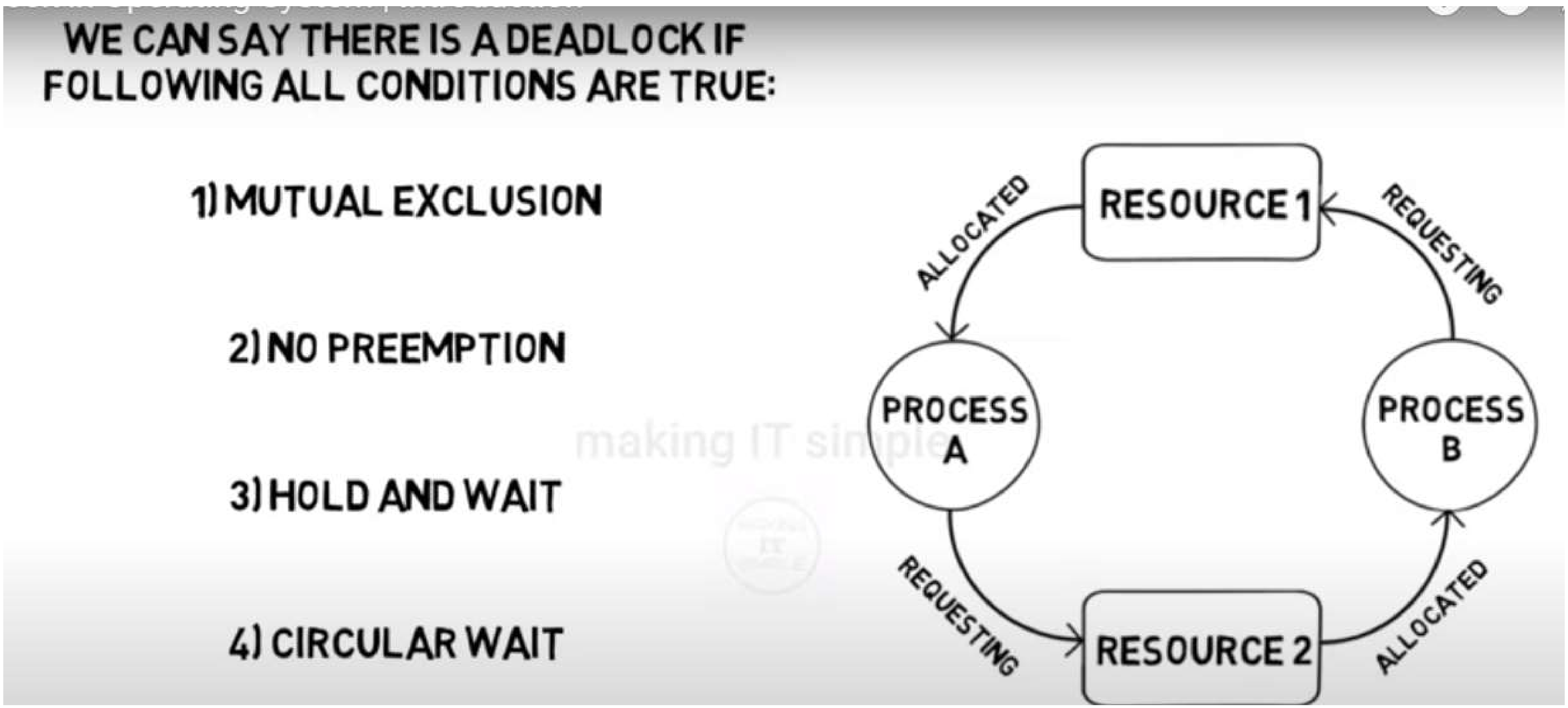
→ If a process unable to change its waiting state indefinitely because the resources requested by it are held by another waiting process. then system is said to be in deadlock.



System model:-

- → Every process will request for the resource
- → If entertained then, process will use the resource
- → Process must release the resource after use

Necessary Condition for Deadlock



Necessary Condition for Deadlock

Deadlock can arise if *four conditions* hold simultaneously:

1. **Mutual exclusion:** only one process at a time can use a resource
2. **Hold and wait:** holding at least one resource and is waiting to acquire additional resources held by others
3. **No preemption:** a resource can be released only voluntarily by the process holding it.
4. **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.

Methods for Handling Deadlock

- The following are **methods for addressing** the possibility of deadlock: ensure that the **system never enters a deadlocked** state:
  - deadlock prevention
  - deadlock avoidance
- deadlock detection and recovery: allow the system to **enter a deadlocked state**, **then deal with and eliminate** the problem
- ignore the problem: **approached used by many operating** systems including UNIX and Windows, and the Java VM

Deadlock Prevention

- Deadlock can be prevented , by making sure that **one of the four** necessary condition for deadlock **should not met**.

Restrain the ways request can be made. Any of the following polices will prevent deadlock:

1. **Mutual Exclusion -** cannot be prevented, since multiple processes shares a resource.

2. **Hold and Wait** –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.

3. **No Preemption** – If a process holding some resources requests another resource that cannot be immediately allocated to it, all resources currently being held are released. Process will be restarted only when it can regain its old resources, as well as the new ones that it is requesting.

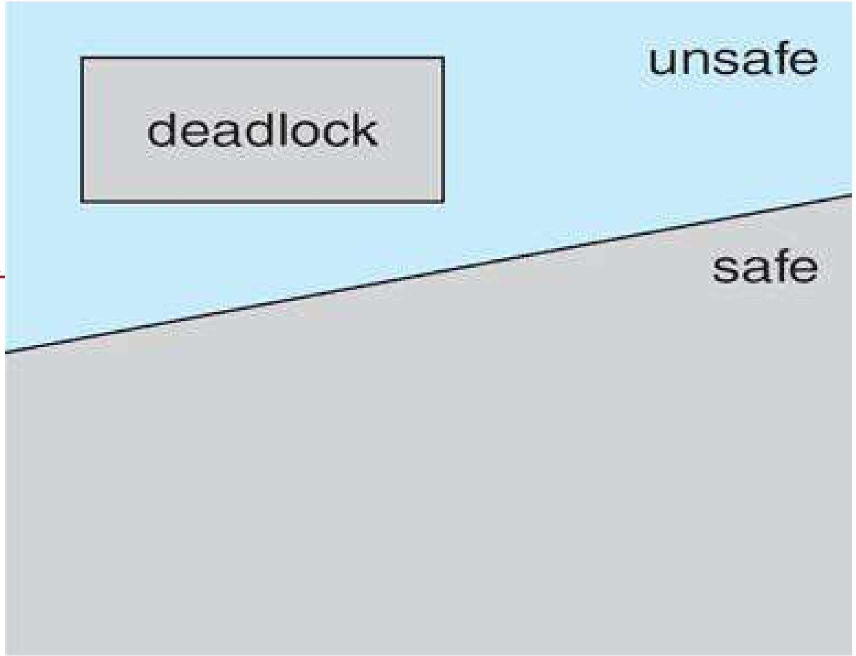
4. **Circular Wait** – impose a total ordering of all resource types, and require that each process requests resources in an ***increasing order*** of enumeration.

Deadlock Avoidance

Requires that the system has ***a priori information*** available.

- Simplest 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.

Safe, Unsafe , Deadlock



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 <**P1, P2, …, Pn**> of ALL the processes in the systems such that:

For each **Pi** , the resources that **Pi** can still request can be satisfied by currently available resources **+** resources held by all the **Pj** , with ***j < i***

- 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**.

Resource Allocation graph

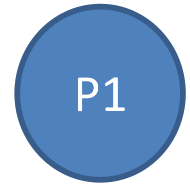
A set of **vertices *V*** and a set of **edges *E***

- **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

- **E** is partitioned into two types:
  - ***request*** edge – directed edge $P_i \rightarrow R_j$
  - ***assignment*** edge – directed edge $R_j \rightarrow P_i$

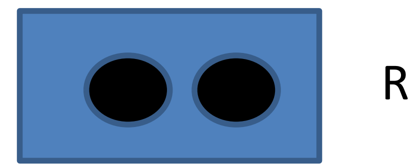
Resource Allocation graph

- The process are represented as circular nodes

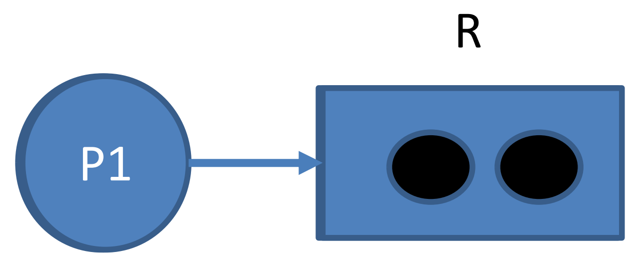


- The resources are represented with a rectangle nodes

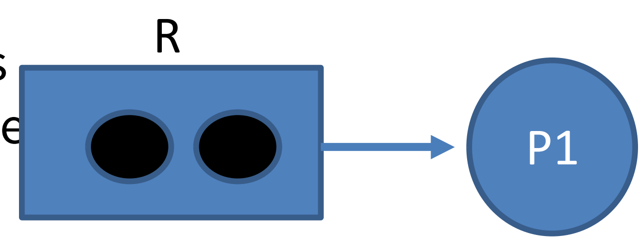
- The resource node contain some dots to represent the number of instance of that resource type. The number of dots is equal to the number of instance.



- An edge, from process to resource, indicates that the process has requested this resource but it has not been allocated. This is known as a request edge.

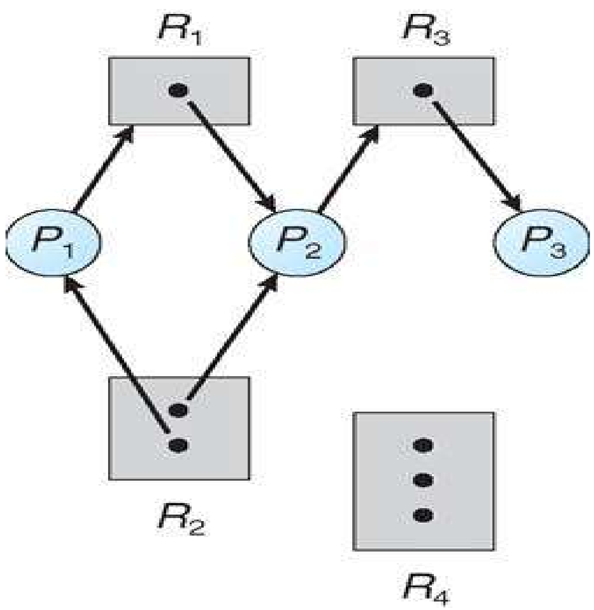


- An edge, from resource node dot to a process, indicates that one instance of this resource type has been allocated to the process. This is known as a assignment edge.



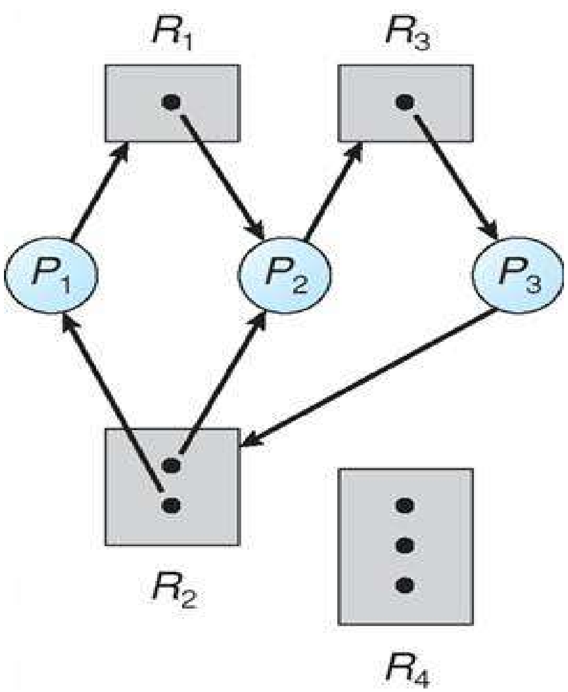
Example of a Resource Allocation Graph

(a) No cycle, No Deadlock



R2->P1->R1->P2->R3->P3
R2->P2->R3->P3

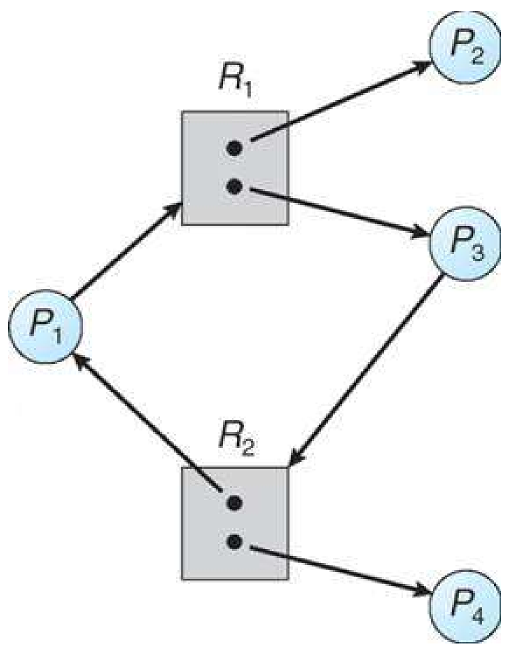
(b)Cycle and Deadlock



R2->P1->R1->P2->R3->P3->R2
R2->P2->R3->P3->R2

Example of a Resource Allocation Graph

(c) cycle, But No Deadlock



R1->P2
R1->P3->R2->P4
R1->P3->R2->P1->R1

Basic Facts

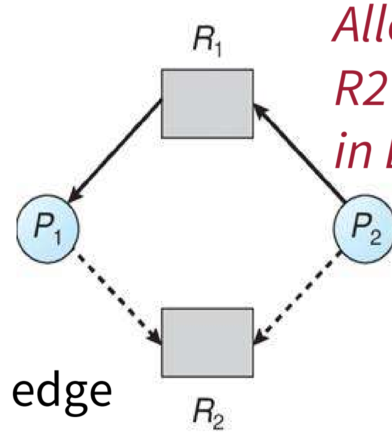
- If graph contains no cycles → no deadlock
- If graph contains a cycle →
  - ➢ if only one instance per resource type, then deadlock
  - ➢ if several instances per resource type, possibility of deadlock

Resource Allocation Graph

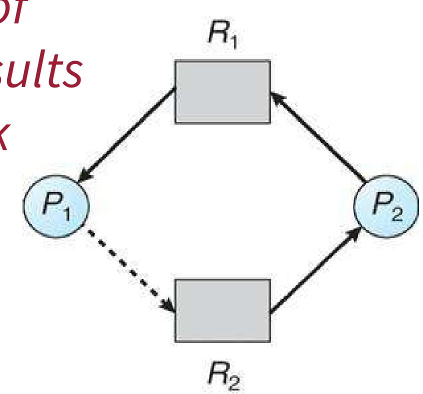
Resource-Allocation Graph Scheme: Used for **Single** instance of resource types

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

- 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



Allocation of $R2 \rightarrow p2$ Results in Deadlock



Resource-Allocation Graph Algorithm

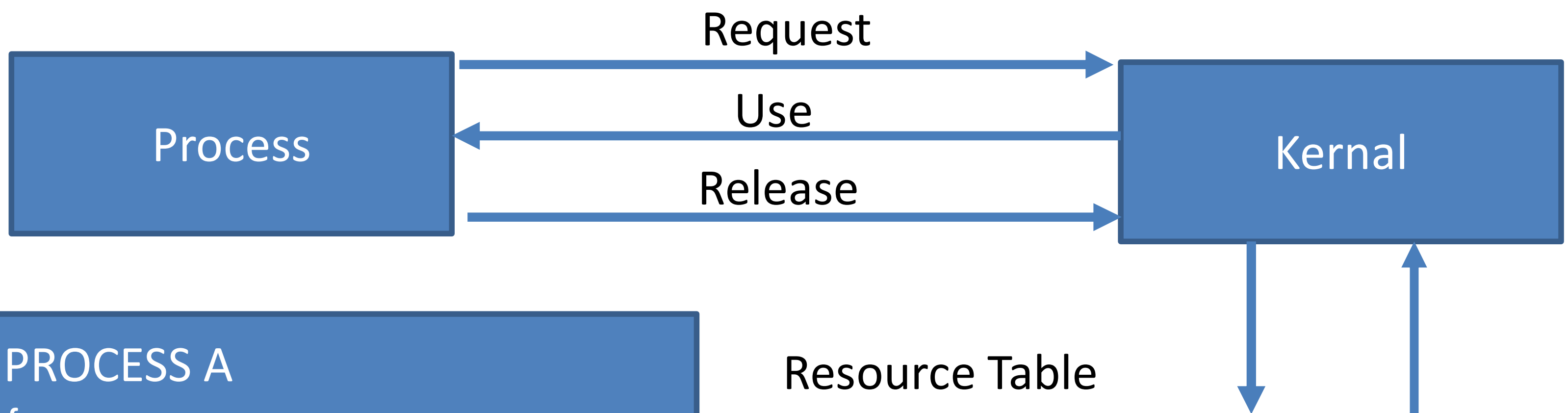
Suppose that process $\boldsymbol{P_i}$ requests a resource $\boldsymbol{R_j}$

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

Resource Allocation in system




```
PROCESS A
{
Semaphore sem_resource;

Wait(sem_resource);

Use the resource; //CS

Signal(sem_resource);

}
```

Resource Table

Resource ID	Resource Name	Status
1	Printer	Free
2	Memory	Allocated
3	Database x	Free
4	…..	……

Resource Allocation in system

```
PROCESS A
{
Semaphore sem_CD;
Semaphore sem_PRINTER;

Wait(sem_CD);
Wait(sem_PRINTER);

Use the resource; //CS

Signal(sem_PRINTER);
Signal(sem_CD);

}
```

```
PROCESS B
{
Semaphore sem_CD;
Semaphore sem_PRINTER;

Wait(sem_CD);
Wait(sem_PRINTER);

Use the resource; //CS

Signal(sem_PRINTER);
Signal(sem_CD);

}
```

```
PROCESS B'
{
Semaphore sem_CD;
Semaphore sem_PRINTER;

Wait(sem_PRINTER);
Wait(sem_CD);

Use the resource; //CS

Signal(sem_CD);
Signal(sem_PRINTER);

}
```

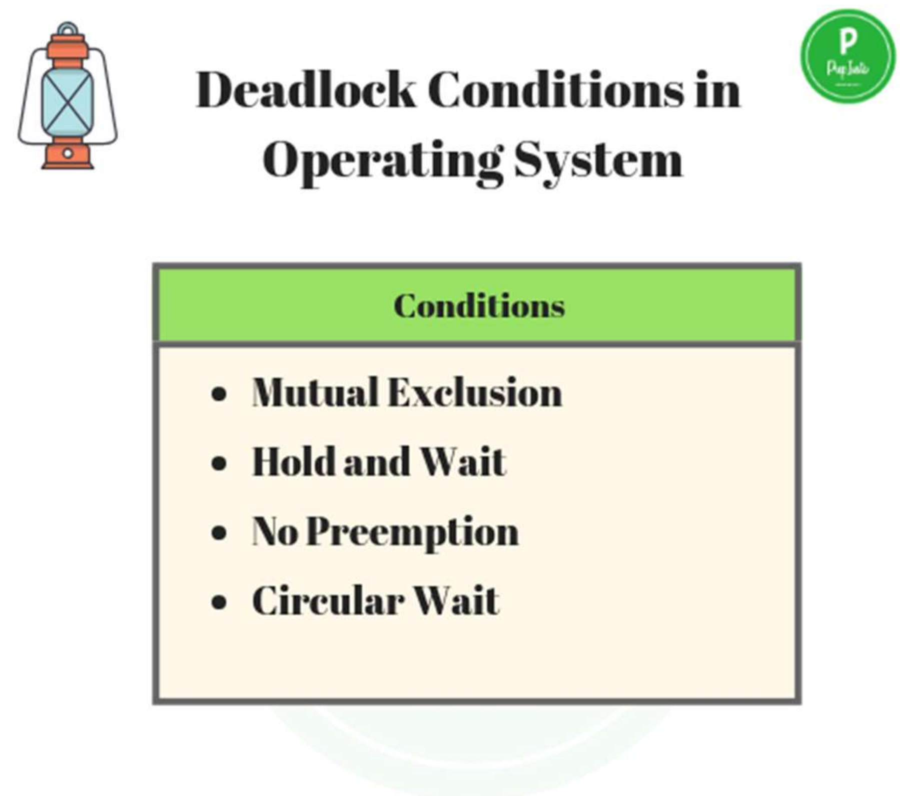
Unit No: 4 Unit Name: Deadlocks

Lecture:
Operating Deadlock Prevention

Deadlock Prevention

Deadlock can occur if all the given 4 conditions (Coffman Conditions) are satisfied, that is if all are true :

- **Mutual Exclusion**
- **Hold and Wait**
- **No Preemption**
- **Circular wait**



Hence , the way to prevent deadlock is to ensure that at least one of these conditions do not hold :

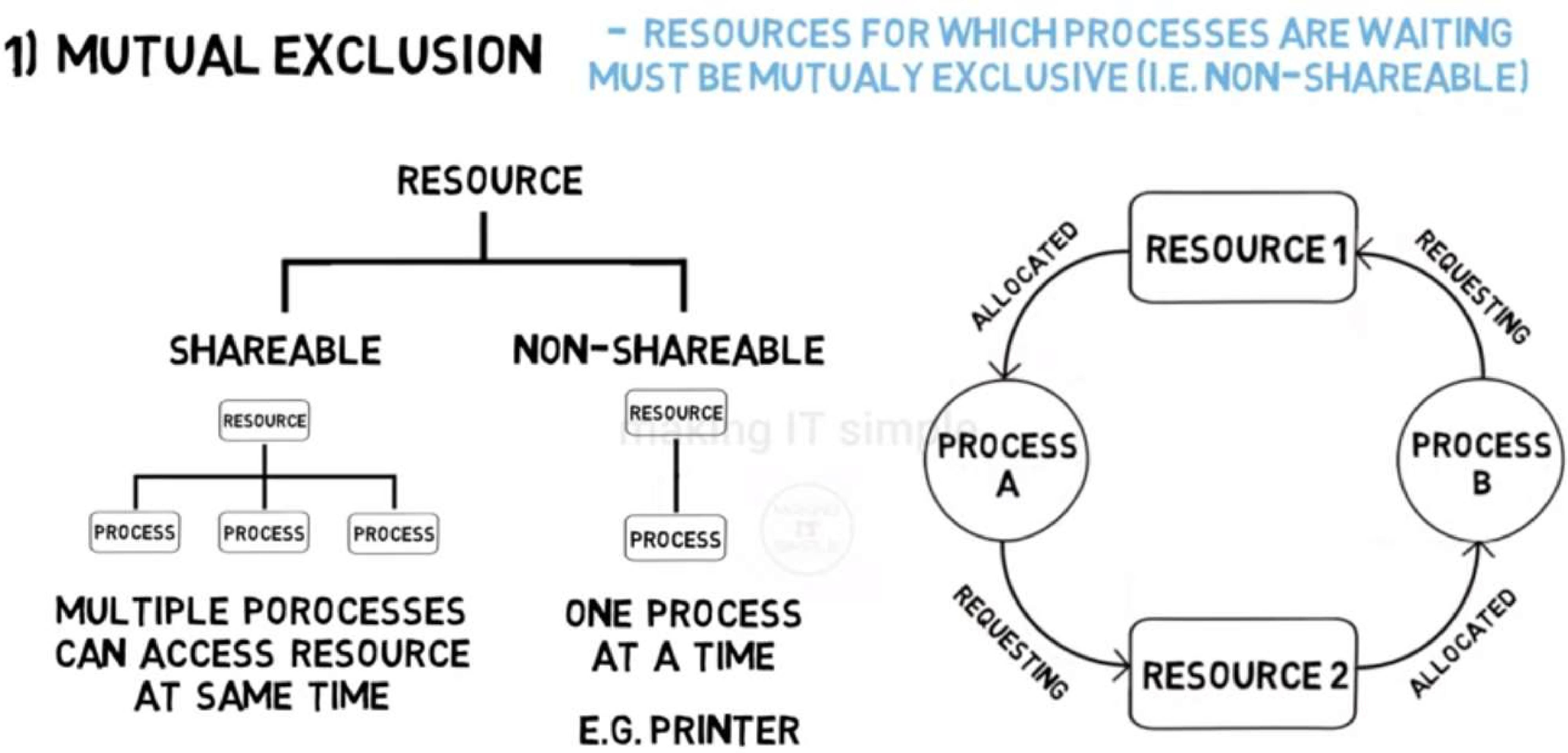
D Y PATIL DEEMED TO BE UNIVERSITY RAMRAO ADIK INSTITUTE OF TECHNOLOGY NAVI MUMBAI

Difference between Deadlock Prevention and Avoidance

Deadlock prevention :- we restrict the processes in which they use the resources(try to break any one condition)

Deadlock avoidance:-we are not restrict the processes in which they use the resources. In this we handle allocation of the resource by analysing the condition of deadlock.

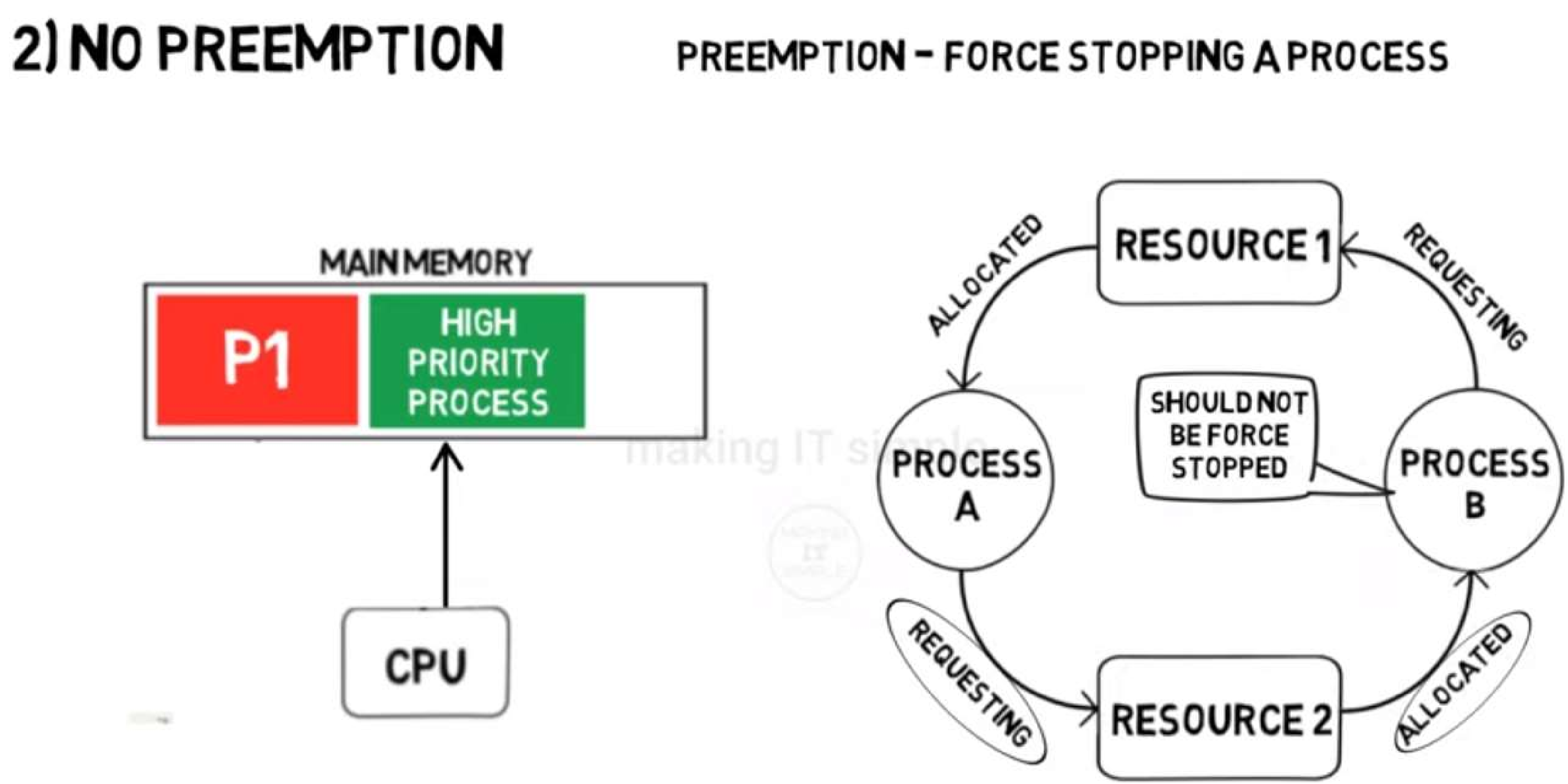
Deadlock Prevention: Mutual Exclusion



Deadlock Prevention: Mutual Exclusion

- The mutual exclusion condition must hold. That is, at least one resource must be non-sharable. Sharable resources, in contrast, do not require mutually exclusive access and thus cannot be involved in a deadlock. Read-only files are a good example of a sharable resource.
- If several processes attempt to open a read-only file at the same time, they can be granted simultaneous access to the file.
- A process never needs to wait for a sharable resource. In general, however, we cannot prevent deadlocks by denying the mutual- exclusion condition, because some resources are intrinsically non- sharable. For example, a mutex lock cannot be simultaneously shared by several processes.

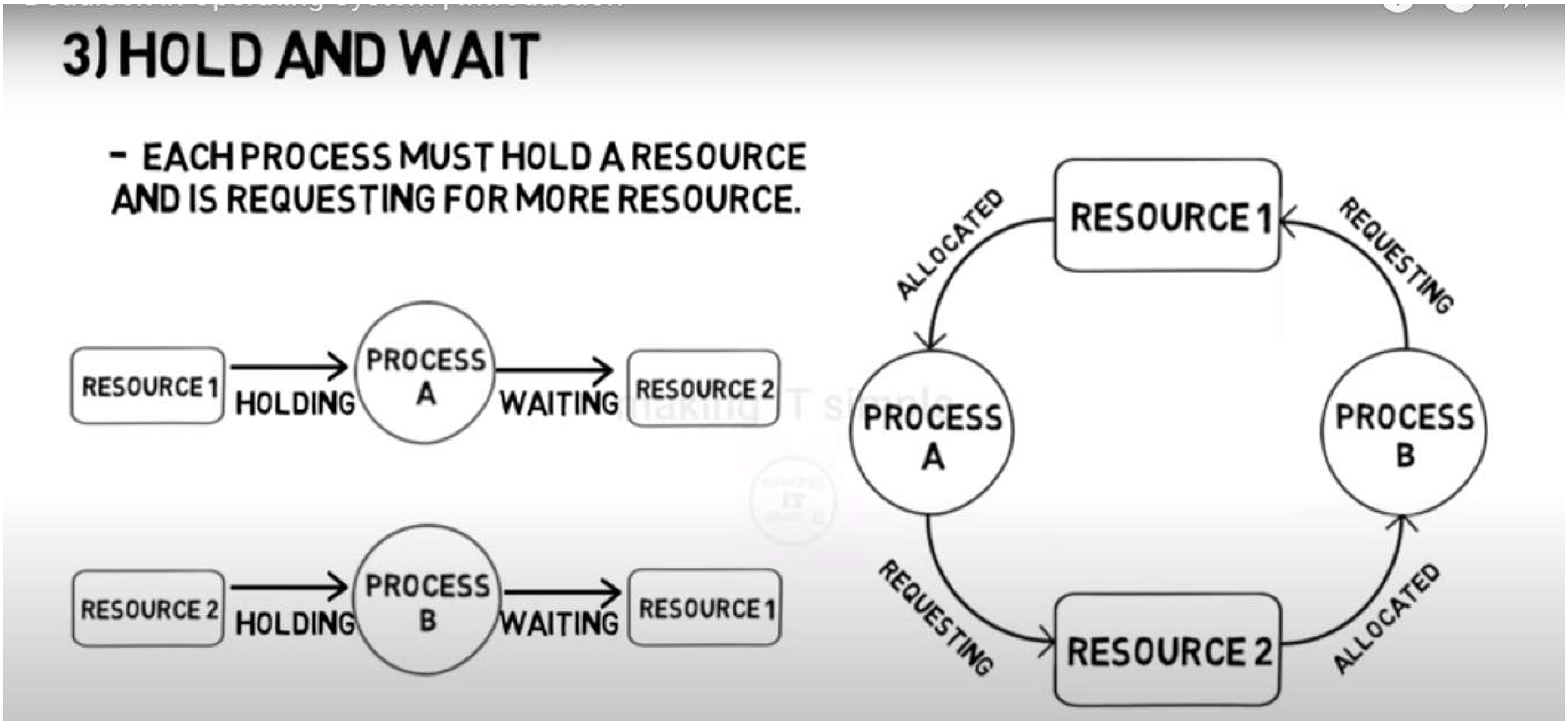
Deadlock Prevention: No Preemption



Deadlock Prevention: No Preemption

- To ensure that this condition does not hold, we can use the following protocol,
- Protocol: If a process is holding some resources and requests another resource that cannot be immediately allocated to it (that is, the process must wait), then all resources the process is currently holding are preempted.
- In other words, these resources are implicitly released. The preempted resources are added to the list of resources for which the process is waiting. The process will be restarted only when it can regain its old resources, as well as the new ones that it is requesting.

Deadlock Prevention: Hold and Wait



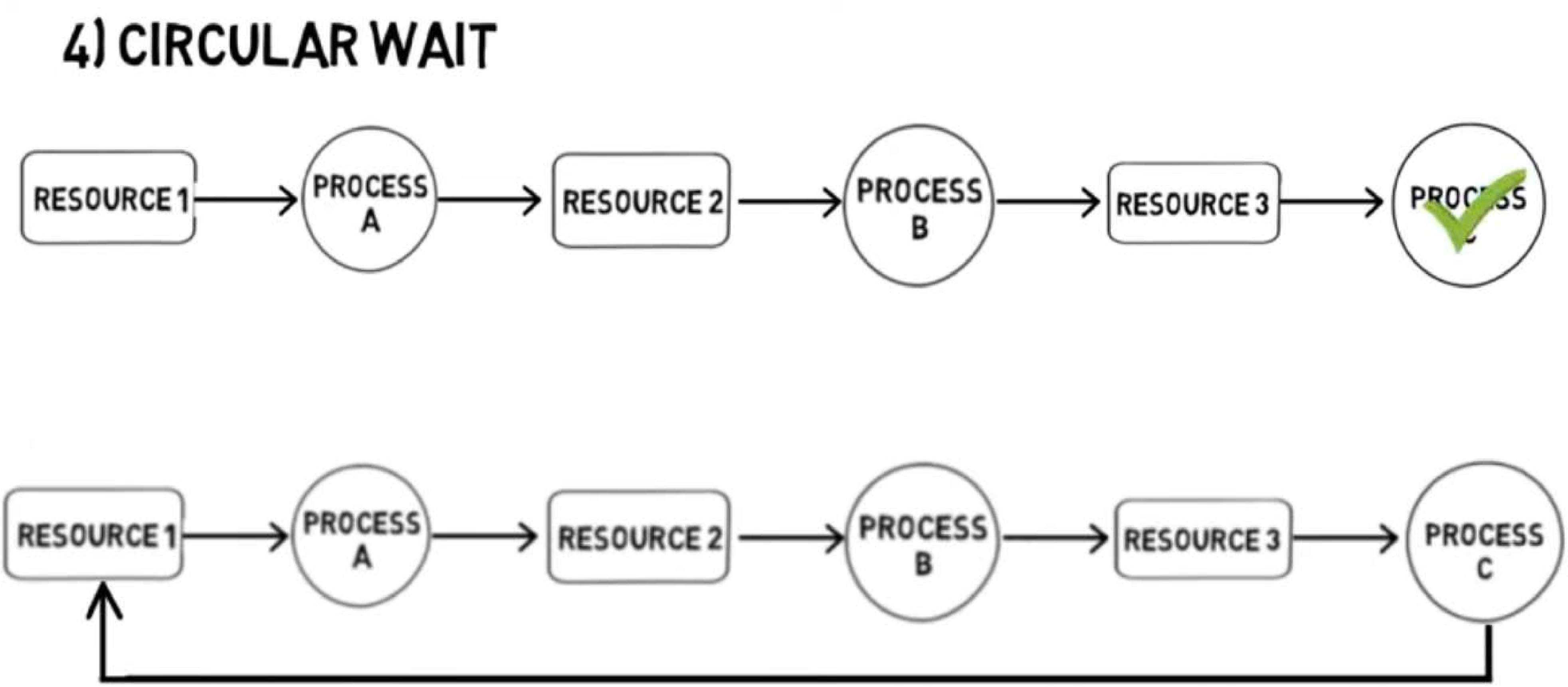
D Y PATIL DEEMED TO BE UNIVERSITY RAMRAO ADIK INSTITUTE OF TECHNOLOGY NAVI MUMBAI

Deadlock Prevention: Hold and Wait

- To ensure that the hold-and-wait condition never occurs in the system,
- we must guarantee that, whenever a process requests a resource, it does not hold any other resources.

- One protocol that we can use requires each process to request and be allocated all its resources before it begins execution.
- We can implement this provision by requiring that system calls requesting resources for a process precede all other system calls.
- An alternative protocol allows a process to request resources only when it has none.
- A process may request some resources and use them. Before it can request any additional resources, it must release all the resources that it is currently allocated.

Deadlock Prevention: Circular Wait



Deadlock Prevention: Circular Wait

- One way to ensure that this condition never holds is to impose a total ordering of all resource types and to require that each process requests resources in an increasing order of enumeration.

Deadlock Prevention

Deadlock can be prevented , by making sure that **one of the four** necessary condition for deadlock **should not met**.

Restrain the ways request can be made. Any of the following polices will prevent deadlock:

1. **Mutual Exclusion -** cannot be prevented, since multiple processes shares a resource.

2. **Hold and Wait** –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.(poor utilization of resources)

3. **No Preemption** – If a process holding some resources requests another resource that cannot be immediately allocated to it, all resources currently being held are released. Process will be restarted only when it can regain its old resources, as well as the new ones that it is requesting.

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

F:R->N

F(ri)<f(rj)

Unit No: 3 Unit Name: Process Synchronization and Deadlocks

Lecture:

Deadlock Avoidance: Banker’s Algorithm for Single & Multiple Resources

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

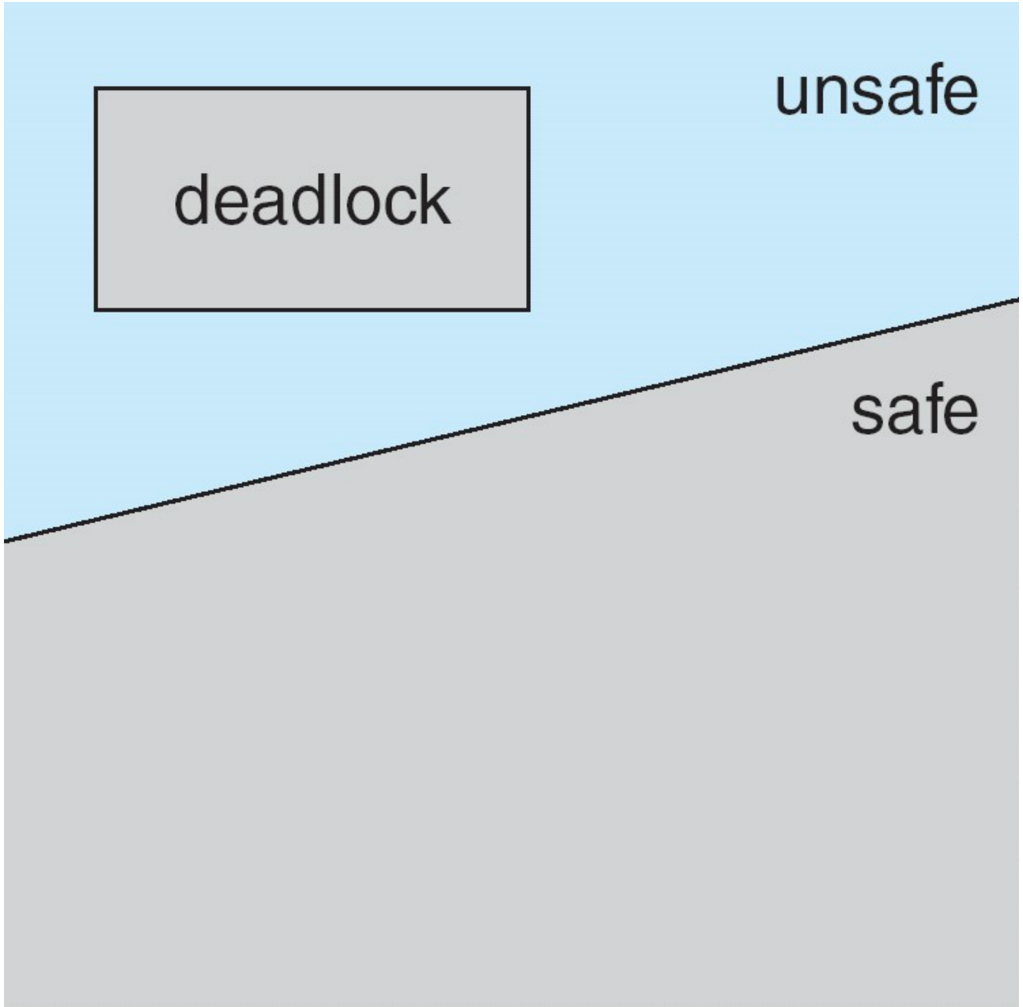
Safe State

- 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, …, P_n> of ALL the processes is 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 $\Rightarrow$ 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

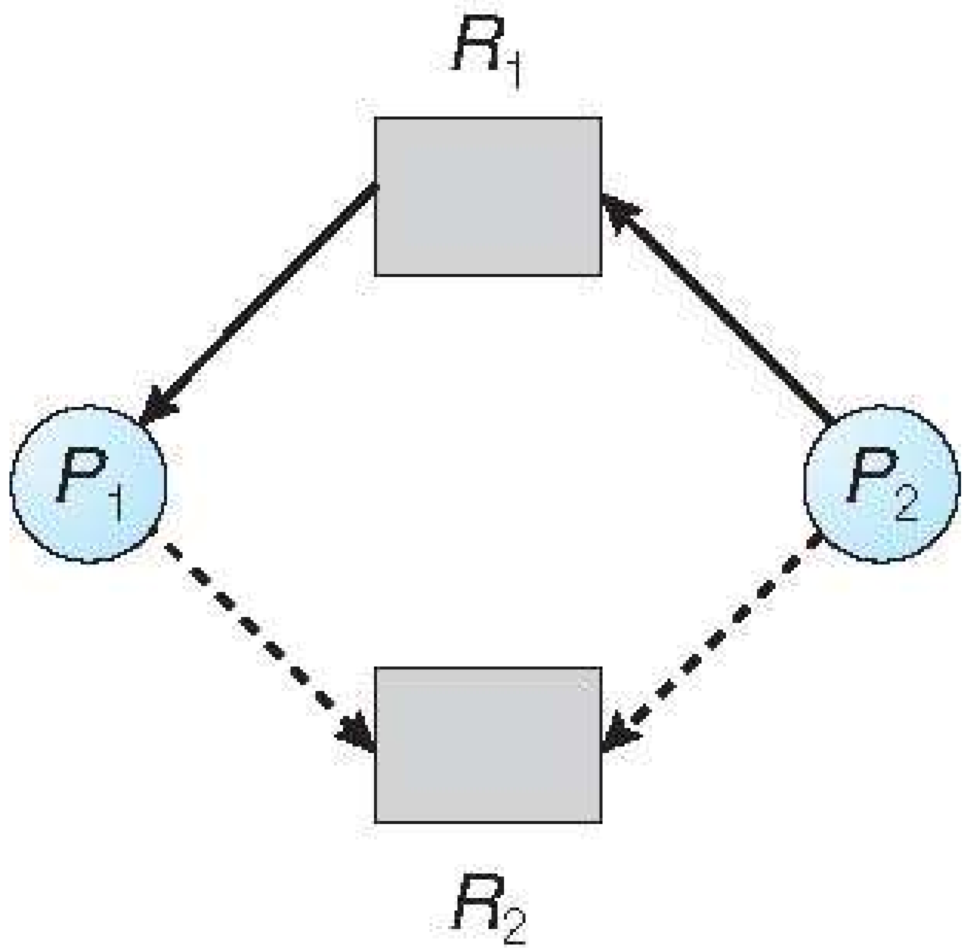


- Single instance of a resource type
  - Use a resource-allocation graph
- Multiple instances of a resource type
  - Use the banker's algorithm

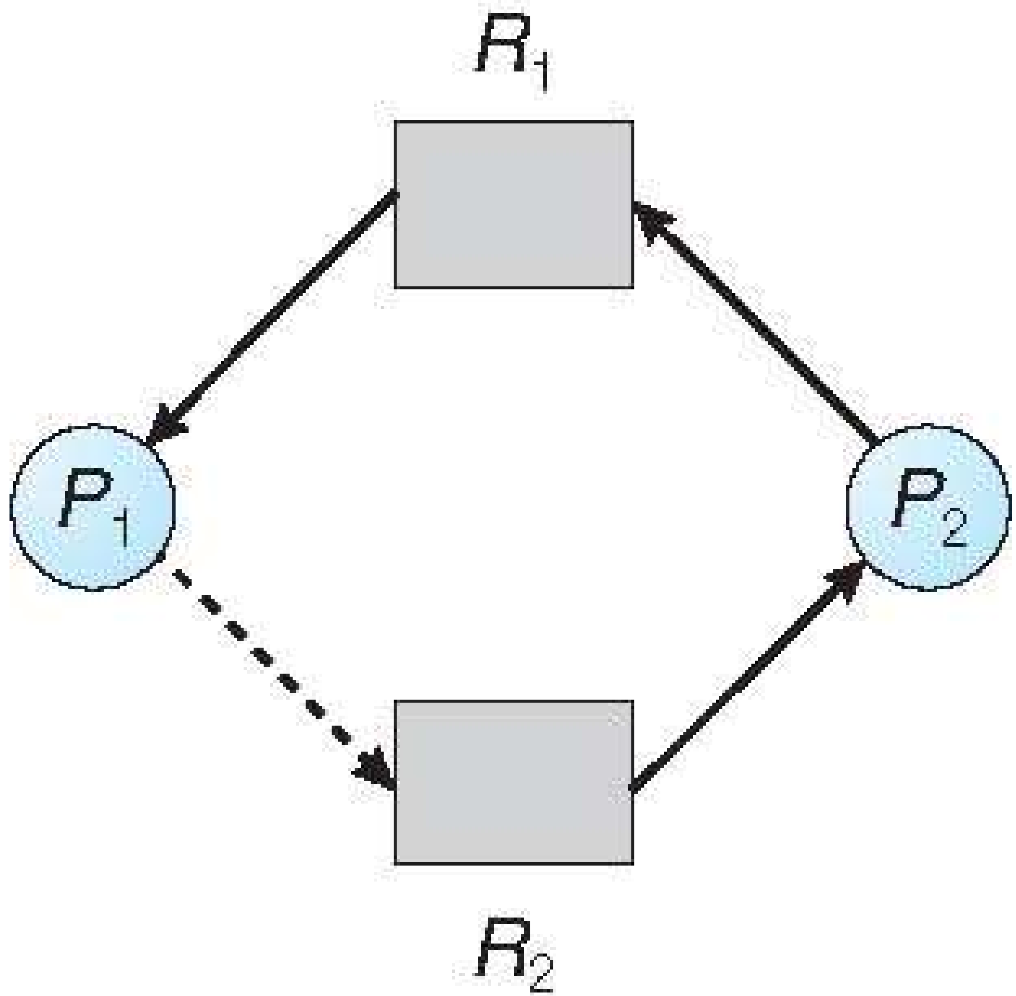
Resource-Allocation Graph Scheme

- **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



Resource-Allocation Graph Algorithm

- Suppose 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

Banker’s Algorithm

- 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 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

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

Safety Algorithm

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

   $Work = Available$

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

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

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 $Request_i \leq Need_i$ 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:

$$Available = Available - Request;$$

$$Allocation_i = Allocation_i + Request_i;$$

$$Need_i = Need_i - Request_i;$$

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

Example 1:

- A system has 12 tape drives and 3 processes: P0, P1, P2.

	Max needs	Currently holding
P0	10	5
P1	4	2
P2	9	2

- ✓ At time t0, 12 – 9 = 3 units are free and system is in a safe state. The sequence <P1, P0, P2> satisfies the safety condition.
- ✓ P1 can request 2 more units since its max = 4. Now P1 holds 4 units and 1 is free. P1 terminates, so now 5 units are free. These are then assigned to P0 and now 0 units are free. Then P0 releases its 10 units. P2 can acquire 7 more units and return them. Now 12 units are free.

Example 1:

- This system can go from safe state to unsafe state as follows:

1. At t1, P2 requests 1 more unit and is granted the unit (now 2 units are free).
2. Now only P1 (needs 2 more at max) can be granted all its units (now 0 units free).
3. When P1 terminates, 4 units are free.
4. P0 has 5 units already and could request the remaining 5 units. But we have only 4 units, so P0 waits.
5. P2 has 3 units and may request the remaining 6 (9 – 3 = 6). So P2 waits => deadlock.

- So we need to stick to the sequence <P1, P0, P2>. So if P2 requests any units, it must wait until previous processes complete. So a request is granted only if allocation leaves the system in a safe state.

Example 2:

Free resources : 3

PROCESS	Allocated	Needed resources
P1	4	10
P2	2	4
P3	2	7

Safe State < P2, P3, P1>

Example 2:

Free resources : 1

PROCESS	Allocated	Needed resources
P1	4	10
P2	4	4
P3	2	7

Example 2:

Free resources : 5

PROCESS	Allocated	Needed resources
P1	4	10
P2	0	0
P3	2	7

Example 2:

Free resources : 0

PROCESS	Allocated	Needed resources
P1	4	10
P2	0	0
P3	7	7

Example 2:

Free resources : 7

PROCESS	Allocated	Needed resources
P1	4	10
P2	0	0
P3	0	0

Example 2:

Free resources : 1

PROCESS	Allocated	Needed resources
P1	10	10
P2	0	0
P3	0	0

Example 2:

Free resources : 11

PROCESS	Allocated	Needed resources
P1	0	0
P2	0	0
P3	0	0

Result analysis

Now, after the termination of processes, we can see following results;

1. P1 have 0 resources and 0 resources required because P1 is no more in system.
2. P2 have 0 resources and 0 resources required because P2 is no more in system.
3. P3 have 0 resources and 0 resources required because P3 is no more in system.

Result: All processes execute successfully, so there is no deadlock and the system is in a safe state

Example3

Considering a system with five processes P_0 through P_4 and three resources of type A, B, C. Resource type A has 10 instances, B has 5 instances and type C has 7 instances. Suppose at time t_0 following snapshot of the system has been taken:

Process	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			

Example3

- **Question1. What will be the content of the Need matrix?**
  Need [i, j] = Max [i, j] – Allocation [i, j]

Example3

Need [i, j] = Max [i, j] – Allocation [i, j]

So, the content of Need Matrix is:

Process	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

Example3

- **Question2. Is the system in a safe state? If Yes, then what is the safe sequence?**
  Applying the Safety algorithm on the given system

m=3, n=5 — Step 1 of Safety Algo

Work = Available

Work = | 3 | 3 | 2 |

	0	1	2	3	4
Finish =	false	false	false	false	false

For i = 0 — Step 2

$Need_0$ = 7, 4, 3

Finish [0] is false and $\overset{7,4,3}{Need_0} > \overset{3,3,2}{Work}$ ✗

So P_0 must wait — But Need ≤ Work

For i = 1 — Step 2

$Need_1$ = 1, 2, 2

Finish [1] is false and $\overset{1,2,2}{Need_1} < \overset{3,3,2}{Work}$ ✓

So P_1 must be kept in safe sequence

Step 3

$Work = \overset{3,3,2}{Work} + \overset{2,0,0}{Allocation_1}$

	A	B	C
Work =	5	3	2

	0	1	2	3	4
Finish =	false	true	false	false	false

For i = 2 — Step 2

$Need_2$ = 6 , 0, 0

Finish [2] is false and $\overset{6,0,0}{Need_2} > \overset{5,3,2}{Work}$ ✗

So P_2 must wait

For i=3 — Step 2

$Need_3$ = 0, 1, 1

Finish [3] = false and $\overset{0,1,1}{Need_3} < \overset{5,3,2}{Work}$ ✓

So P_3 must be kept in safe sequence

Step 3

$Work = \overset{5,3,2}{Work} + \overset{2,1,1}{Allocation_3}$

	A	B	C
Work =	7	4	3

	0	1	2	3	4
Finish =	false	true	false	true	false

For i = 4 — Step 2

$Need_4$ = 4, 3, 1

Finish [4] = false and $\overset{4,3,1}{Need_4} < \overset{7,4,3}{Work}$ ✓

So P_4 must be kept in safe sequence

Step 3

$Work = \overset{7,4,3}{Work} + \overset{0,0,2}{Allocation_4}$

	A	B	C
Work =	7	4	5

	0	1	2	3	4
Finish =	false	true	false	true	true

For i = 0 — Step 2

$Need_0$ = 7, 4, 3

Finish [0] is false and $\overset{7,4,3}{Need} < \overset{7,4,5}{Work}$ ✓

So P_0 must be kept in safe sequence

Step 3

$Work = \overset{7,4,5}{Work} + \overset{0,1,0}{Allocation_0}$

	A	B	C
Work =	7	5	5

	0	1	2	3	4
Finish =	true	true	false	true	true

For i = 2 — Step 2

$Need_2$ = 6 , 0, 0

Finish [2] is false and $\overset{6,0,0}{Need_2} < \overset{7,5,5}{Work}$ ✓

So P_2 must be kept in safe sequence

Step 3

$Work = \overset{7,5,5}{Work} + \overset{3,0,2}{Allocation_2}$

	A	B	C
Work =	10	5	7

	0	1	2	3	4
Finish =	true	true	true	true	true

Step 4

Finish [i] = true for $0 \le i \le n$

Hence the system is in Safe state

The safe sequence is P_1, P_3, P_4, P_0, P_2

Example3

- **Question3. What will happen if process P_1 requests one additional instance of resource type A and two instances of resource type C?**

Example3

$$\text{Request}_1 = \overset{A}{1}, \overset{B}{0}, \overset{C}{2}$$

To decide whether the request is granted we use Resource Request algorithm

Step 1

$\overset{1,0,2}{\text{Request}_1} < \overset{1,2,2}{\text{Need}_1}$

Step 2

$\overset{1,0,2}{\text{Request}_1} < \overset{3,3,2}{\text{Available}}$

Step 3

$\text{Available} = \text{Available} - \text{Request}_1$

$\text{Allocation}_1 = \text{Allocation}_1 + \text{Request}_1$

$\text{Need}_1 = \text{Need}_1 - \text{Request}_1$

Process	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	

Example3

- We must determine whether this new system state is safe. To do so, we again execute Safety algorithm on the above data structures.

m=3, n=5 — Step 1 of Safety Algo

Work = Available

Work =	2	3	0

	0	1	2	3	4
Finish =	false	false	false	false	false

For i = 0 — Step 2

Need₀ → $Need_0$ = 7, 4, 3

Finish [0] is false and $Need_0$ > Work (7, 4, 3 > 2, 3, 0)

So P_0 must wait

But Need ≤ Work

For i = 1 — Step 2

$Need_1$ = 0, 2, 0

Finish [1] is false and $Need_1$ < Work (0, 2, 0 < 2, 3, 0)

So P_1 must be kept in safe sequence

Step 3

Work = Work + $Allocation_1$ (2, 3, 0 + 3, 0, 2)

	A	B	C
Work =	5	3	2

	0	1	2	3	4
Finish =	false	true	false	false	false

For i = 2 — Step 2

$Need_2$ = 6 , 0, 0

Finish [2] is false and $Need_2$ > Work (6, 0, 0 > 5,3, 2)

So P_2 must wait

For i=3 — Step 2

$Need_3$ = 0, 1, 1

Finish [3] = false and $Need_3$ < Work (0, 1, 1 < 5, 3, 2)

So P_3 must be kept in safe sequence

Step 3

Work = Work + $Allocation_3$ (5, 3, 2 + 2, 1, 1)

	A	B	C
Work =	7	4	3

	0	1	2	3	4
Finish =	false	true	false	true	false

For i = 4 — Step 2

$Need_4$= 4, 3, 1

Finish [4] = false and $Need_4$ < Work (4, 3, 1 < 7, 4, 3)

So P_4 must be kept in safe sequence

Step 3

Work = Work + $Allocation_4$ (7, 4, 3 + 0, 0, 2)

	A	B	C
Work =	7	4	5

	0	1	2	3	4
Finish =	false	true	false	true	true

For i = 0 — Step 2

$Need_0$ = 7, 4, 3

Finish [0] is false and Need < Work (7, 4, 3 < 7, 4, 5)

So P_0 must be kept in safe sequence

Step 3

Work = Work + $Allocation_0$ (7, 4, 5 + 0, 1 , 0)

	A	B	C
Work =	7	5	5

	0	1	2	3	4
Finish =	true	true	false	true	true

For i = 2 — Step 2

$Need_2$ = 6 , 0, 0

Finish [2] is false and $Need_2$ < Work (6, 0, 0 < 7, 5, 5)

So P_2 must be kept in safe sequence

Step 3

Work = Work + $Allocation_2$ (7, 5, 5 + 3, 0, 2)

	A	B	C
Work =	10	5	7

	0	1	2	3	4
Finish =	true	true	true	true	true

Step 4

Finish [i] = true for 0 ≤ i ≤ n

Hence the system is in Safe state

The safe sequence is P_1, P_3 , P_4 , P_0 , P_2

Example3

- Hence the new system state is safe, so we can immediately grant the request for process **P_1**.
  Code for Banker’s Algorithm

Example 4 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	

Example4 (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

Example4: P_1 Request (1,0,2)

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

for P0 Request (7,4,3) <= Available (2,3,0) ???

No

Hence P0 cannot be granted resources

Example4: P_1 Request (1,0,2)

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

Check for P1:

Example4: P_1 Request (1,0,2)

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

Check for P2:

Example4: P_1 Request (1,0,2)

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

Check for P3:

Example4: P_1 Request (1,0,2)

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

- Check for P4:

Example4: P_1 Request (1,0,2)

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

Check for P0:

Check for P2:

Example4: P_1 Request (1,0,2)

- Check that Request $\leq$ Available (that is, $(1,0,2) \leq (3,3,2) \Rightarrow$ 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 $< P_1, P_3, P_4, P_2, P_0>$ 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?

Example5:

Let us consider the following snapshot for understanding the banker's algorithm:

Processes	Allocation A B C	Max A B C	Available A B C
P0	1 1 2	4 3 3	2 1 0
P1	2 1 2	3 2 2	
P2	4 0 1	9 0 2	
P3	0 2 0	7 5 3	
P4	1 1 2	1 1 2	

1.calculate the content of the need matrix?
2.Check if the system is in a safe state?
3.Determine the total sum of each type of resource?

Solution5:

1**. The Content of the need matrix can be calculated by using the formula given below:**

Need = Max – Allocation

Process	Need
P0	3 2 1
P1	1 1 0
P2	5 0 1
P3	7 3 3
P4	0 0 0

2. Let us now check for the safe state.
Safe sequence:

For process P0, Need = (3, 2, 1) and
Available = (2, 1, 0)

Need <=Available = False

So, the system will move to the next process.

Example5:

2. For Process P1, Need = (1, 1, 0)

Available = (2, 1, 0)

Need <= Available = True

Request of P1 is granted.

Available = Available +Allocation

= (2, 1, 0) + (2, 1, 2)

= (4, 2, 2) (New Available)

Example5:

3. For Process P2, Need = (5, 0, 1)

Available = (4, 2, 2)

Need <=Available = False

So, the system will move to the next process.

Example5:

4. For Process P3, Need = (7, 3, 3)

Available = (4, 2, 2)

Need <=Available = False

So, the system will move to the next process.

Example5:

5. For Process P4, Need = (0, 0, 0)

Available = (4, 2, 2)

Need <= Available = True

Request of P4 is granted.

Available = Available + Allocation

= (4, 2, 2) + (1, 1, 2)

= (5, 3, 4) now, (New Available)

Example5:

6. Now again check for Process P2, Need = (5, 0, 1)

Available = (5, 3, 4)

Need <= Available = True

Request of P2 is granted.

Available = Available + Allocation

= (5, 3, 4) + (4, 0, 1)

= (9, 3, 5) now, (New Available)

Example5:

7. Now again check for Process P3, Need = (7, 3, 3)

Available = (9, 3, 5)

Need <=Available = True

The request for P3 is granted.

Available = Available +Allocation

= (9, 3, 5) + (0, 2, 0) = (9, 5, 5)

Example5:

8. Now again check for Process P0, = Need (3, 2, 1)

= Available (9, 5, 5)

Need <= Available = True

So, the request will be granted to P0.

Safe sequence: < P1, P4, P2, P3, P0>

Example5:

The system allocates all the needed resources to each process. So, we can say that the system is in a safe state.

3. The total amount of resources will be calculated by the following formula:

The total amount of resources= sum of columns of allocation + Available

= [8 5 7] + [2 1 0] = [10 6 7]

Disadvantages of Banker's Algorithm

Some disadvantages of this algorithm are as follows:

1.During the time of Processing, this algorithm does not permit a process to change its maximum need.

2.Another disadvantage of this algorithm is that all the processes must know in advance about the maximum resource needs.

Resource Request Algorithm

- Now the next algorithm is a resource-request algorithm and it is mainly used to determine whether requests can be safely granted or not.

- Let Requesti be the request vector for the process Pi.

- If Requesti[j]==k, then process Pi wants k instance of Resource type Rj.

- When a request for resources is made by the process Pi, the following are the actions that will be taken:

Algorithm

1. If Request*i* <= Need*i*, then go to step 2;else raise an error condition, since the process has exceeded its maximum claim.

2.If Request*i* <= Available then go to step 3; else Pi must have to wait as resources are not available.

3.Now we will assume that resources are assigned to process Pi and thus perform the following steps:

Available= Available-Requesti;

Allocationi=Allocationi +Requesti;

$Need_i = Need_i - Request_i$;

If the resulting resource allocation state comes out to be safe, then the transaction is completed and, process Pi is allocated its resources.

But in this case, if the new state is unsafe, then Pi waits for Requesti, and the old resource-allocation state is restored.

Thank You

D Y PATIL
DEEMED TO BE
UNIVERSITY
RAMRAO ADIK
INSTITUTE OF TECHNOLOGY
NAVI MUMBAI

Unit No: 3 Unit Name: Process Synchronization and Deadlocks

Lecture:

Deadlock Detection and Recovery

Deadlock Detection- Multiple Instance

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:

Work = ***Available***

For *i* = 1,2, …, *n*:

if $\boldsymbol{Allocation_i}$ **!=** 0, then ***Finish*****[i]** = false; else ***Finish*****[i]** = *true*

Deadlock Detection- Multiple Instance

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

Finish[***i***] **==** ***false*** **AND** ***Request$_i$*** **<=** ***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 £ *i* £ *n*, then the system is in **deadlock state**.

and ***P$_i$*** is **deadlocked**

The Algorithm requires ***m* x *n*^{2}** operations to detect whether the system is in deadlocked state

Example of Deadlock Detection- Multiple Instance

Example of Detection Algorithm

- Five processes P_0 through P_4;
- Three resource types A B C

  7 2 6

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	

System is **not deadlocked**, sequence **<P_0, P_2, P_3, P_1, P_4>** will result in ***Finish*[*i*] = true** for all *i*

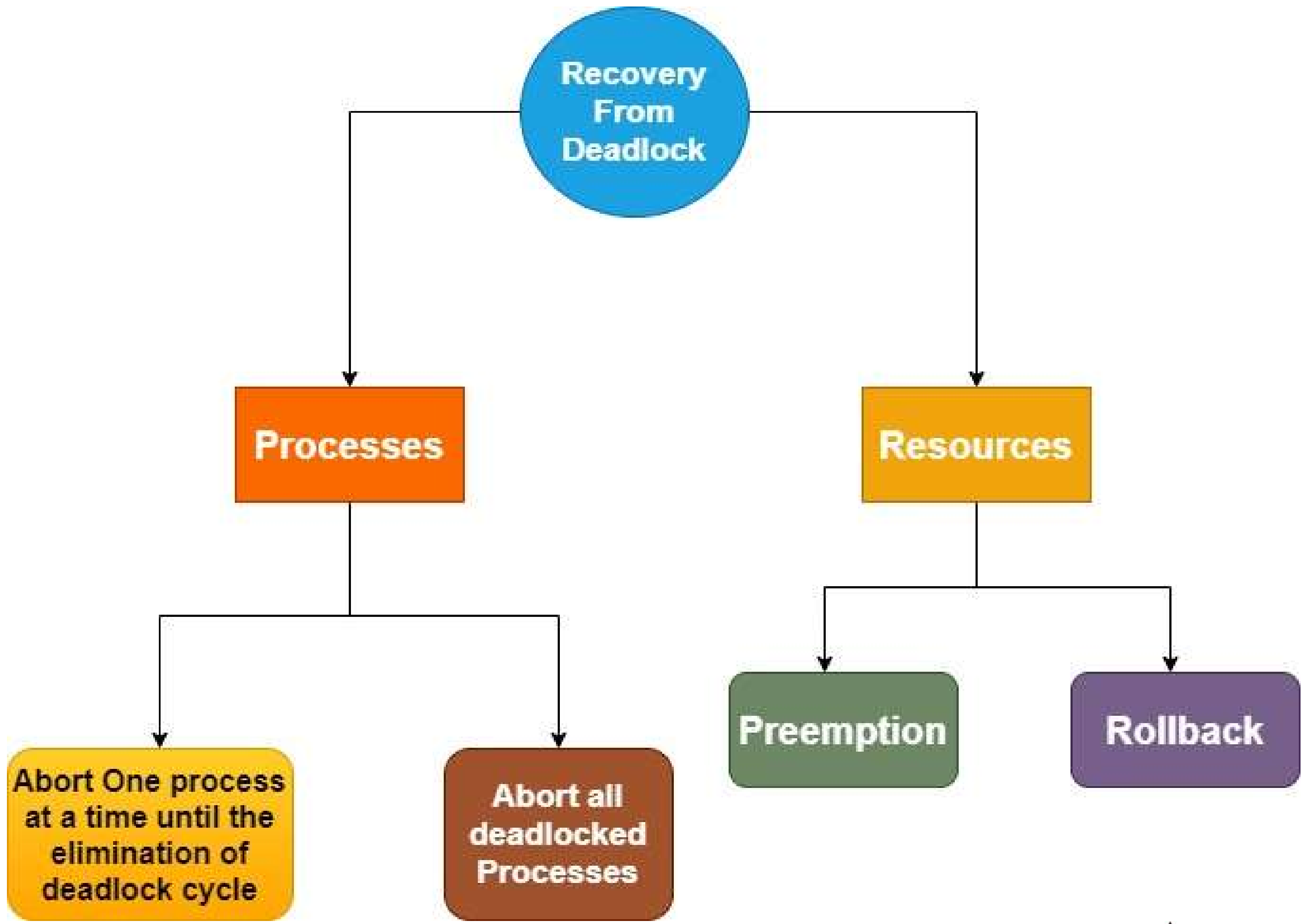
Example of Deadlock Detection- Multiple Instance

- 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?
  - o Can reclaim resources held by process P_0,
  but insufficient resources to fulfill other processes requests
  - o **Deadlock exists**, consisting of processes P_1, P_2, P_3, and P_4

Recovery From Deadlock



Process Termination

- In order to eliminate deadlock by aborting the process, we will use one of two methods given below. In both methods, the system reclaims all resources that are allocated to the terminated processes.

- **Aborting all deadlocked Processes**
  - Clearly, this method is helpful in breaking the cycle of deadlock, but this is an expensive approach. This approach is not suggestable but can be used if the problem becomes very serious. If all the processes are killed then there may occur insufficiency in the system and all processes will execute again from starting.

- **Abort one process at a time until the elimination of the deadlock cycle**
  - This method can be used but we have to decide which process to kill and this method incurs considerable overhead. The process that has done the least amount of work is killed by the Operating system firstly.

Resource Preemption

- In order to eliminate the deadlock by using resource preemption, we will successively preempt some resources from processes and will give these resources to some other processes until the deadlock cycle is broken and there is a possibility that the system will recover from deadlock.

- But there are chances that the system goes into starvation.

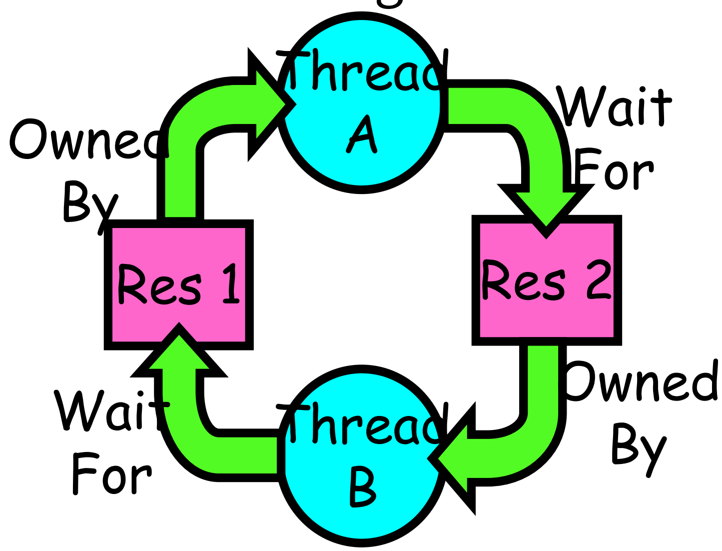
Starvation vs Deadlock

The occurrence of deadlock can be detected by the resource scheduler.

Starvation	Deadlock
When all the low priority processes got blocked, while the high priority processes execute then this situation is termed as Starvation.	Deadlock is a situation that occurs when one of the processes got blocked.
Starvation is a long waiting but it is not an infinite process.	Deadlock is an infinite process.
It is not necessary that every starvation is a deadlock.	There is starvation in every deadlock.
Starvation is due to uncontrolled priority and resource management.	During deadlock, preemption and circular wait does not occur simultaneously.

Starvation vs Deadlock

- Starvation: thread waits indefinitely
  - Example, low-priority thread waiting for resources constantly in use by high-priority threads
- Deadlock: circular waiting for resources
  - Thread A owns Res 1 and is waiting for Res 2
    Thread B owns Res 2 and is waiting for Res 1



Deadlock $\Rightarrow$ Starvation but not vice versa
Starvation can end (but doesn't have to)
Deadlock can't end without external intervention

Rules of the Game

- The philosophers are very logical
  - They want to settle on a shared policy that all can apply concurrently
  - They are hungry: the policy should let everyone eat (eventually)
  - They are utterly dedicated to the proposition of equality: the policy should be totally fair

Dining philosophers (cont)

- Each philosopher goes in a cycle
  - Think for a while
  - Get 2 chopsticks
  - Eat for a while
  - Put down the chopsticks
  - Repeat

  - Your task is to devise a scheme for the “Get 2 chopsticks” step

Solution 1 (Bad Solution)

```
Think();
Pick up left chopstick;
Pick up right chopstick;
Eat();
Put down right chopstick;
Put down left chopstick;
```

- Problem: Deadlock
- Why?

1. Each philosopher can pick up left fork before anyone picks up their right fork.
2. Now everyone is waiting for right fork.

Solution 2: Global lock

```
Think();
table.lock();
while(!both chopstick available)
  chopstickPutDown.await();
Pick up left chopstick;
Pick up right chopstick;
table.unlock();
Eat();
Put down right chopstick;
Put down left chopstick;
chopstickPutDown.signal();
```

Solution 3: Reactive

```
Think();
Pick up left chopstick;
if(right chopstick available) {
  Pick up right chopstick;
} else {
  Put down left chopstick;
  continue; //Go back to Thinking
}
Eat();
```

Solution 4: Global ordering

```
Think();
Pick up "smaller" chopstick from left and right;
Pick up "bigger" chopstick from left and right;
Eat();
Put down "bigger" chopstick from left and right;
Put down "smaller" chopstick from left and right;
```

- Why can't we deadlock?
  1. It is not possible for all philosophers to have a chopstick
  2. Two philosophers, A and B, must share a chopstick, X, that is "smaller" than all other chopsticks
  3. One of them, A, has to pick it up X first
  4. B can't pick up X at this point
  5. B can't pick up the bigger one until X is picked up
  6. SO, 4 philosophers left, 5 chopsticks total

     One philosopher must be able to have two chopsticks!

Dining Philosophers Solutions

- Allow only 4 philosophers to sit simultaneously
- Asymmetric solution
  - Odd philosopher picks left fork followed by right
  - Even philosopher does vice versa
- Pass a token
- Allow philosopher to pick fork only if both available

Solutions are less interesting than the problem itself!

- In fact the problem statement is why people like to talk about this problem!
- Rather than solving Dining Philosophers, we should use it to understand properties of solutions that work and of solutions that can fail!

Thank You

D Y PATIL
DEEMED TO BE
UNIVERSITY
RAMRAO ADIK
INSTITUTE OF TECHNOLOGY
NAVI MUMBAI