OPERATING SYSTEMS (THEORY)

LECTURE - 8

K.ARIVUSELVAN

Assistant Professor (Senior) – (SITE)

VIT University

DEADLOCKS

The Deadlock Problem

❑ Processes each holding a resource and waiting to acquire a resource held by another process

- Example1
  - ➔System has 2 tape drives
  - ➔P1 and P2 each hold one tape drive and each needs another one

Deadlock Characterization

Deadlock can arise if four conditions hold simultaneously:

(1) Mutual exclusion: **only one process at a time can use a resource (i.e. resource is non-sharable).**

(2) Hold and wait: a process **holding at least one** resource is **waiting to acquire additional resources** held by other processes.

Deadlock Characterization

(3) No preemption: a resource can be **released only voluntarily** by the process holding it, after that process has completed its task.

(4) Circular wait: there exists a set {$P0$, $P1$, …, Pn} of waiting processes such that:

=> $P0$ is waiting for a resource that is held by $P1$,
=> $P1$ is waiting for a resource that is held by $P2$, …,
=> $Pn–1$ is waiting for a resource that is held by Pn,
and
=> Pn is waiting for a resource that is held by $P0$.

Methods for Handling Deadlocks

- Deadlock Prevention

- Deadlock Avoidance

Deadlock Prevention

=> Ensure atleast one of the necessary conditions cannot hold.

Deadlock Avoidance

=> Given additional information in advance, concerning which resources a process will request and use during its life time

NOTE:

=> A System does not employ either a deadlock prevention or deadlock avoidance algorithm , it leads to a deadlock situation.

Deadlock Prevention

(1) Mutual Exclusion:

=> Hold for non-sharable resource (Printer can't share simultaneously)

=> Sharable resource mutual exclusion not required (Read only file access by several process simultaneously). No dead lock

=> In general it is not possible to prevent deadlock by denying the mutual exclusion condition.

Deadlock Prevention

(2) Hold & Wait:

1st Protocol :

=> Allot all requested resource before begins execution

Example:

Copying data from Tape drive to Disk file, Sort Disk file, Print

Drawback:

Hold printer entire execution even printer need at last

Deadlock Prevention

(2) Hold & Wait:

2nd Protocol :

=> Request some resource and use them

=> Before request, release additional resource currently allotted.

Example:

- **Request Tape drive & Disk**
- **Release**
- **Request Disk & Printer**

Deadlock Prevention

(3) No Preemption:

Protocol :

=> Check for resource availability

if (available)

{ allocate resource}

Else

{Check other process hold resource and request additional resource }

If (Yes) then PREEMPT

Deadlock Prevention

(4) Circular Wait:

Protocol :

=>Assign unique integer to resource type

Example:

Tape Drive = 1, Disk = 4, Printer = 10

Constraint :

=> Process request the resource only in increasing order

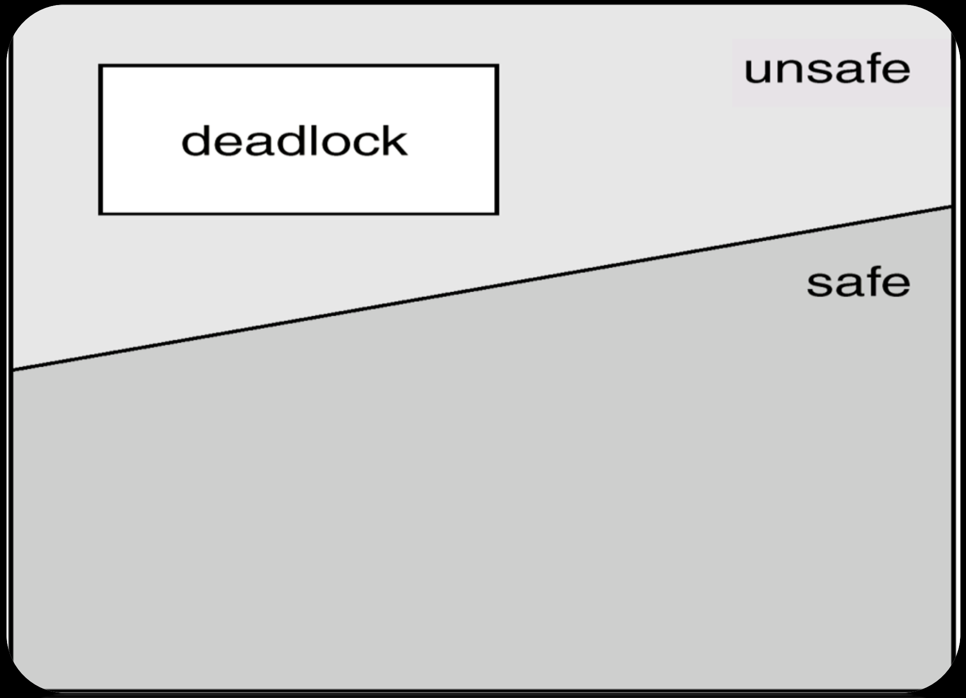
=> Process that want to use both Tape & Printer at the same time must first request Tape & then Printer

Deadlock Avoidance

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 **safe sequence** of all processes.



Example – Safe State

- System with

  12 Tape Drives &

  3 Processes (P_0 , P_1 , P_2)

Time T_0:

	Max.Needs	Allocation	Available
P0	10	5	3
P1	4	2	
P2	9	2	

- Sequence <P1, P0, P2> is in Safe State or Unsafe Time T_0

Banker's Algorithm

- A resource allocation system with Multiple instances

- Each process must a priori claim maximum use.

Constraint:

- 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:** Number of available resources. If Available [j] = k, there are k instances of resource type R_j available.

- **Max:** *Maximum demand of each process.* If *Max* [i,j] = k, then process P_i may request at most k instances of resource type R_j.

- **Allocation:** *Number of resource currently allocated to each process.* If Allocation[i,j] = k then P_i is currently allocated k instances of R_j.

- **Need:** Remaining resource need of each process. 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

- 5 processes P0 through P4;
- 3 resource types A (10 instances), B (5 instances) and C (7 instances).

- Snapshot at time *T*0:

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

Example (Cont.)

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

	Need A	B	C
P0	7	4	3
P1	1	2	2
P2	6	0	0
P3	0	1	1
P4	4	3	1

Resource-Request Algorithm for Process *Pi*

- **Request $_i$ = request vector for process Pi.**

- **If Request $_i$ [j] = k then process P_i wants k instances of resource type R_j.**

ACTION TAKEN:

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:`

How?

Check if this new state is Safe. i.e.if a safe sequence exists

Available = Available – $Request_i$;

$Allocation_i$ = $Allocation_i$ + $Request_i$;

$Need_i$ = $Need_i$ – $Request_i$;

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

Safety Algorithm

- **Finding out whether or not a system is in safe state**

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

Work* = *Available
***Finish* [*i*] = *false* for *i* = 1,2,3, …, *n*.**

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.b

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

Example (Cont.)

Seq	Avail	Need	New=Avail-Need	Max	Max + New
P_1	3 3 2	1 2 2	2 1 0	3 2 2	5 3 2
P_3					
P_4					
P_2					
P_0					

Resource-allocation Graph

A visual (mathematical) way to determine if a deadlock has, or may occur.

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

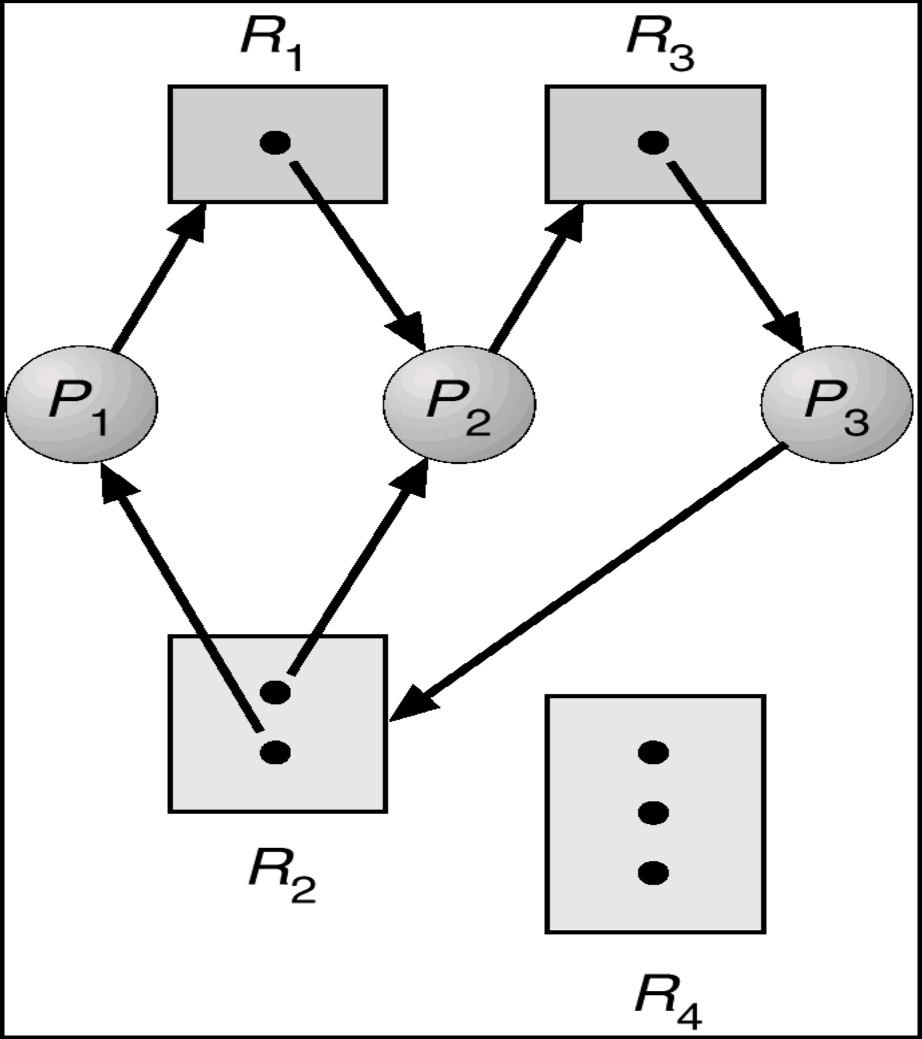
- **V is partitioned into two types:**

=> ***P* = {*P*1, *P*2, …, *Pn*},** the set consisting of all the **processes** in the system.

=> ***R* = {*R*1, *R*2, …, *Rm*}**, the set consisting of all **resource types** in the system

- **Request edge** – directed edge $P_i \rightarrow R_j$
- **Assignment edge** – directed edge $R_j \rightarrow P_i$
- **If graph contains cycles $\Rightarrow$ deadlock**

Resource Allocation Graph

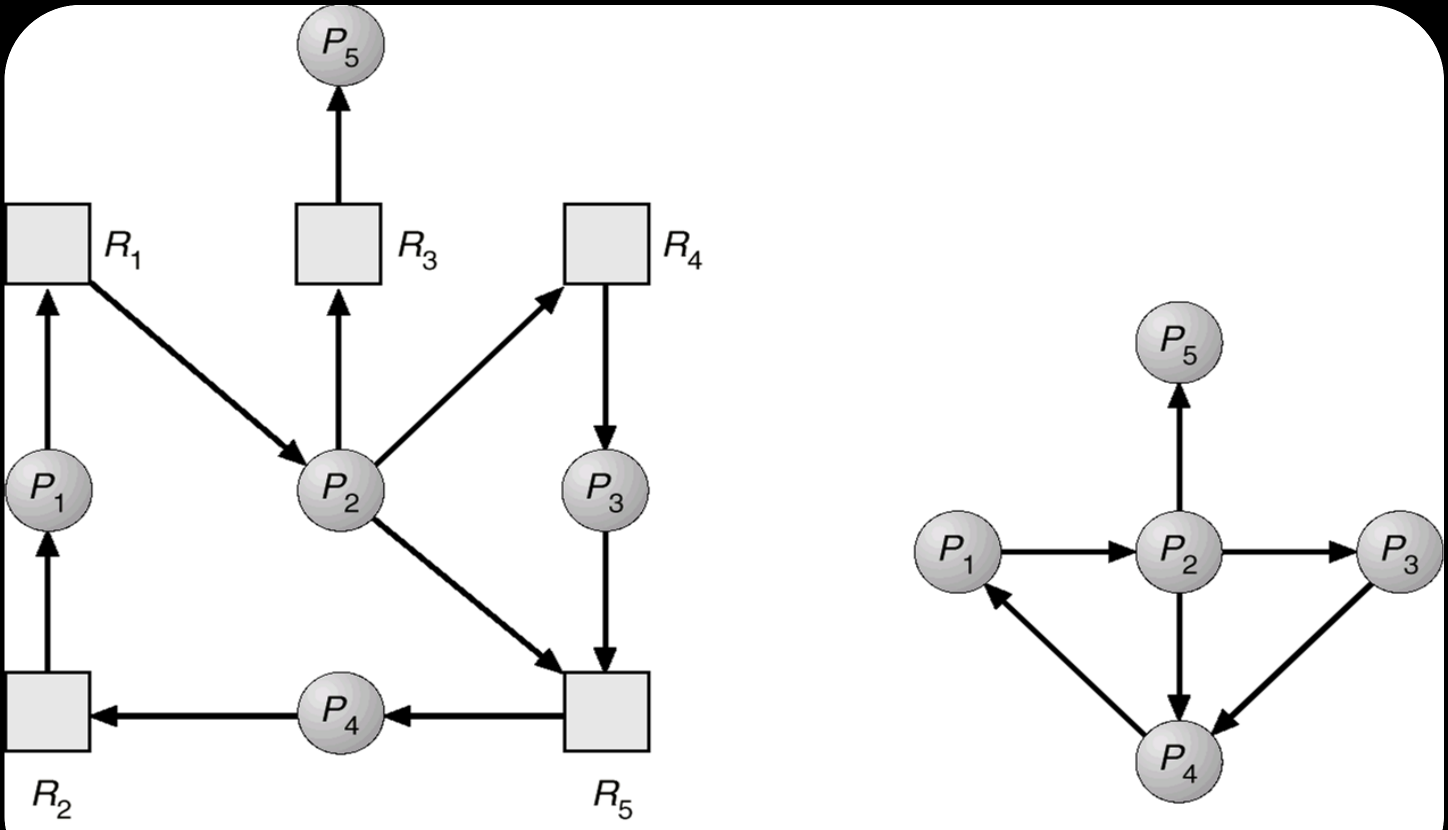


Wait-For Graph (Single Instance)

- **Maintain *wait-for* graph**
  - ➔**Nodes are processes**
  - ➔***Pi* → *Pj* if *Pi* is waiting for *Pj***

- **Periodically invoke an algorithm that searches for a cycle in the graph.**

Resource-Allocation Graph and Wait-for Graph



(a) Resource-Allocation Graph

(b) Corresponding wait-for graph

Deadlock Detection

- Allow system to enter deadlock state

(1) Detection algorithm

(2) Recovery scheme

Detection Algorithm

Several Instances of a Resource Type

Data Structures:

Available: **Number of available resources** of each type.

Allocation: The number of resources of each type currently allocated to each process.

Request: The current request of each process.

If Request [i,j] = k, then process Pi is requesting k more instances of resource type. Rj.

Detection Algorithm

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

(a) *Work = Available*

(b) For $i = 1,2, \ldots, n$,
if $Allocation_i \neq 0$, then Finish[i] = false;
otherwise, Finish[i] = true (i.e..$Allocation_i = 0$).

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

(a) *Finish*[*i*] == *false*
(b) *Requesti* ≤ *Work*

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

Detection Algorithm (Cont.)

3. Work = Work + Allocationi
   Finish[i] = true
   go to step 2.

4. If Finish[i] == false, for some i , $1 \leq i \leq n$, then the system is in deadlock state.

Moreover, if Finish[i] == false, then Pi is deadlocked.

Example of Detection Algorithm

- Five processes *P*0 through *P*4;
- Three resource types A (7 instances), *B* (2 instances), and *C* (6 instances)

- 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	

- Sequence <*P*0, *P*2, *P*3, *P*1, *P*4> will result in *Finish*[*i*] = true for all *i*.

Example (Cont.)

- *P*2 requests an additional instance of type *C*.

	Request
	A B C
*P*0	0 0 0
*P*1	2 0 1
*P*2	0 0 1
*P*3	1 0 0
*P*4	0 0 2

- State of system?

  → Can reclaim resources held by process *P*0, but insufficient resources to fulfill other processes requests.

  → Deadlock exists, consisting of processes *P*1, *P*2, *P*3, and *P*4.

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