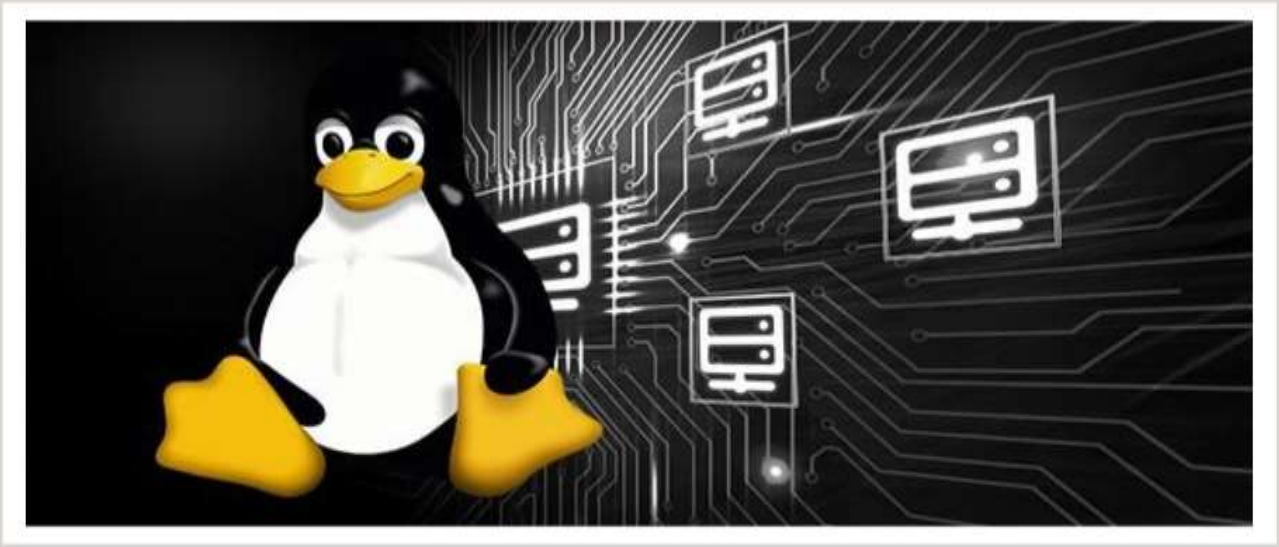


Operating systems

By
I Ravindra kumar, B.Tech, M.Tech,(Ph.D.)
Assistant professor,
Dept of CSE, VNR VJIET

Deadlocks

Deadlock

A system can be modeled as

- a collection of limited resources, which can be partitioned into different categories,
- to be allocated to a number of processes, each having different needs.
- Resource categories may include memory, printers, CPUs, open files, tape drives, CD-ROMS, etc.

In normal operation a process must request a resource in the following sequence

- **Request** -
  - If the request cannot be immediately granted, then the process must wait until the resource(s) it needs become available.
  - For example the system calls open( ), malloc( ), new( ), and request( ).
- **Use** - The process uses the resource, e.g. prints to the printer or reads from the file.
- **Release** -
  - The process relinquishes the resource. so that it becomes available for other processes.
  - For example, close( ), free( ), delete( ), and release( ).

all kernel-managed resources,

- the kernel keeps track of what resources are free
- which are allocated, to which process they are allocated,
- a queue of processes waiting for this resource to become available.

Application-managed resources can be controlled using mutexes or wait( ) and signal( ) calls

DeadLock:

when every process in the set is waiting for a resource that is currently allocated to another process in the set (and which can only be released when that other waiting process makes progress.)

Necessary Conditions

- **Mutual Exclusion** - At least one resource must be held in a non-sharable mode; If any other process requests this resource, then that process must wait for the resource to be released.
- **Hold and Wait** - A process must be simultaneously holding at least one resource and waiting for at least one resource that is currently being held by some other process.
- **No preemption** - Once a process is holding a resource (i.e. once its request has been granted), then that resource cannot be taken away from that process until the process voluntarily releases it.
- **Circular Wait** - A set of processes { P0, P1, P2, . . ., PN } must exist such that every P[i] is waiting for P[(i + 1) % (N + 1)]. (Note that this condition implies the hold-and-wait condition, but it is easier to deal with the conditions if the four are considered separately.)

Resource-Allocation Graph

Resource- Allocation Graphs, having the following properties:

- A set of resource categories, { R1, R2, R3, . . ., RN }, which appear as square nodes on the graph.

- Dots inside the resource nodes indicate specific instances of the resource.

(E.g. two dots might represent two laser printers.)

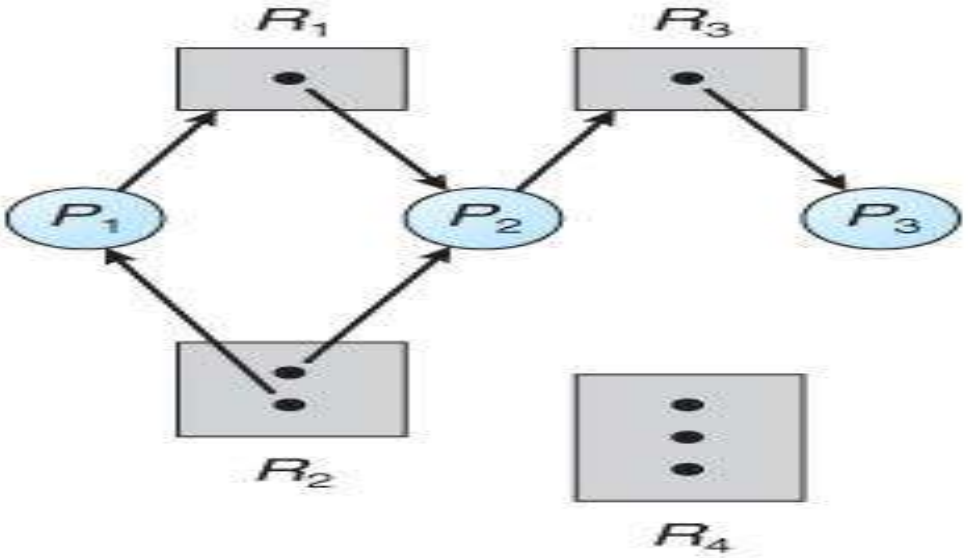
- A set of processes, { P1, P2, P3, . . ., PN }

- **Request Edges** - A set of directed arcs from Pi to Rj, indicating that process Pi has requested Rj, and is currently waiting for that resource to become available.

- **Assignment Edges** - A set of directed arcs from Rj to Pi indicating that resource Rj has been allocated to process Pi, and that Pi is currently holding resource Rj.

Note that a request edge can be converted into an assignment edge by reversing the direction of the arc when the request is granted.

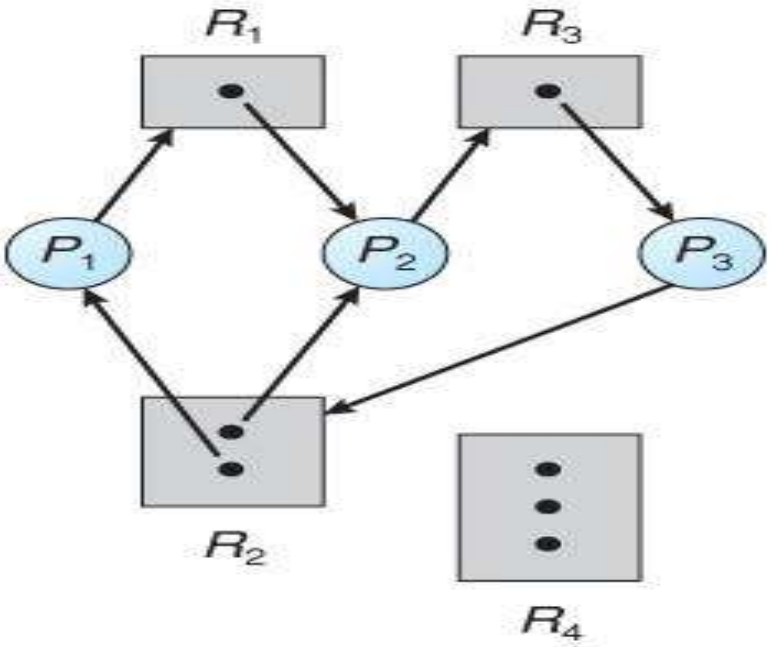
(However note also that request edges point to the category box, whereas assignment edges emanate from a particular instance dot within the box.)

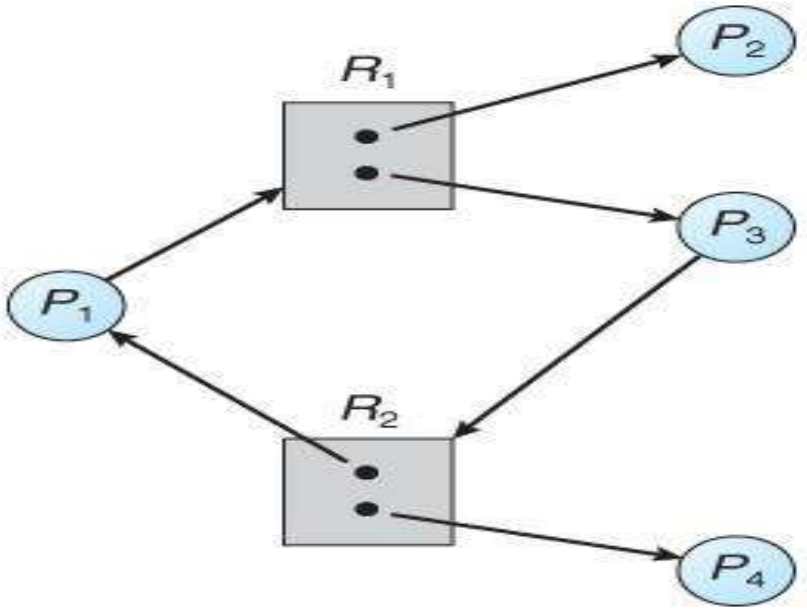


●If a resource-allocation graph contains no cycles, then the system is not deadlocked.

(When looking for cycles, remember that these are directed graphs.)

●If a resource-allocation graph does contain cycles AND each resource category contains only a single instance, then a deadlock exists.





Methods for Handling Deadlocks:

- Deadlock prevention or avoidance -
- Deadlock detection and recovery -
- Ignore the problem all together

Deadlock Prevention:

by preventing at least one of the four required conditions:

Mutual Exclusion

- Shared resources such as read-only files do not lead to deadlocks.
- Unfortunately some resources, such as printers and tape drives, require exclusive access by a single process.

Hold and Wait

- processes must be prevented from holding one or more resources while simultaneously waiting for one or more others
- Require that all processes request all resources at one time.
- Require that processes holding resources must release them before requesting new resources

No Preemption:

Approaches are:

- if a process is forced to wait when requesting a new resource, then all other resources previously held by this process are implicitly released
- when a resource is requested and not available,
    - then the system looks to see what other processes currently have those resources and are themselves blocked waiting for some other resource.
    - If such a process is found, then some of their resources may get preempted and added to the list of resources for which the process is waiting.

Circular Wait:

- number all resources, and to require that processes request resources only in strictly increasing (or decreasing) order.
- in order to request resource Rj, a process must first release all Ri such that i >= j.

Deadlock Avoidance

To prevent deadlocks from ever happening, by preventing at least one of the afore mentioned conditions.

● requires more information about each process, AND tends to lead to low device utilizatithe scheduler only needs to know the maximum number of each resource that a process might potentially use on

When a scheduler sees that starting a process or granting resource requests may lead to future deadlocks, then that process is just not started or the request is not granted.

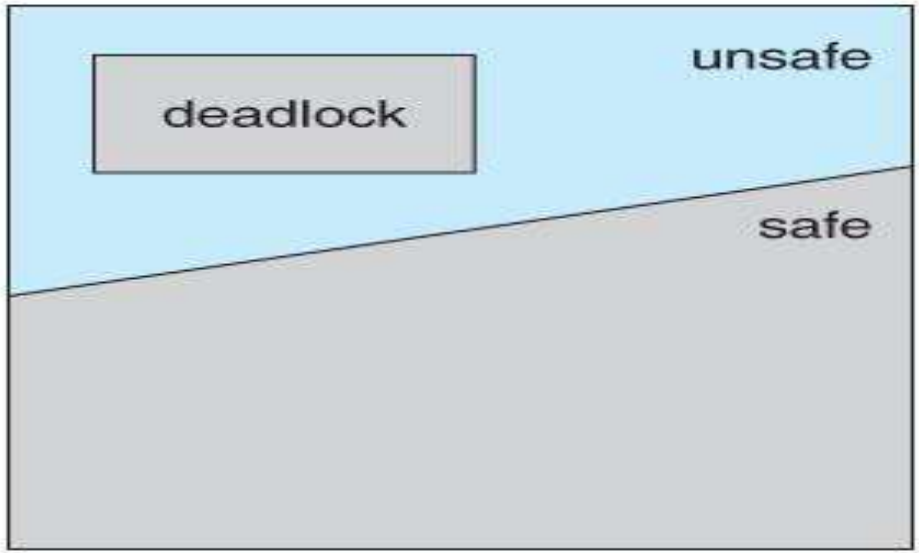
A resource allocation state is defined by the number of available and allocated resources, and the maximum requirements of all processes in the system.

Safe State

A state is safe if the system can allocate all resources requested by all processes (up to their stated maximums) without entering a deadlock state.

a state is safe

if there exists a safe sequence of processes { P0, P1, P2, ..., PN } such that all of the resource requests for Pi can be granted using the resources currently allocated to Pi and all processes Pj where j < i.



consider a system with 12 tape drives, allocated as follows. Is this a safe state? What is the safe sequence?

	Maximum Needs	Current Allocation
P0	10	5
P1	4	2
P2	9	2

Resource-Allocation Graph Algorithm:

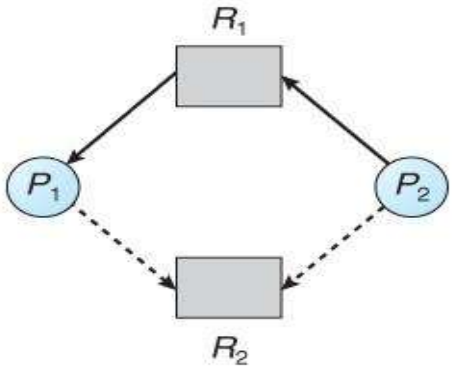
If resource categories have only single instances of their resources,

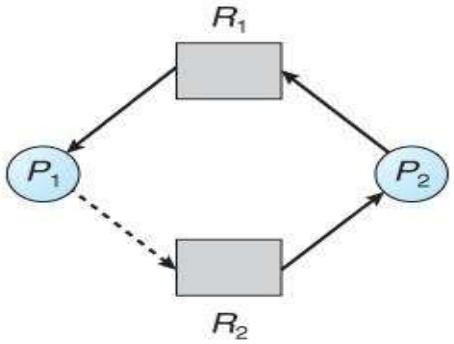
then deadlock states can be detected by cycles in the resource-allocation graphs.

unsafe states can be recognized and avoided by augmenting the resource-allocation graph with claim edges,

noted by dashed lines, which point from a process to a resource that it may request in the future.

In order for this technique to work, all claim edges must be added to the graph for any particular process before that process is allowed to request any resources.





Banker's Algorithm

- For resource categories that contain more than one instance
- the resource-allocation graph method does not work, and more complex (and less efficient) methods
- The Banker's Algorithm gets its name because it is a method that bankers could use to assure that when they lend out resources they will still be able to satisfy all their clients.

Relies on several key data structures:

- Available[m]
- Max[n][m]
- Allocation[n][m
- Need[n][m]

To apply the Banker's algorithm,

Safety Algorithm determines if the current state of a system is safe, according to the following steps:

- Let Work and Finish be vectors of length m and n respectively.
- Work is a working copy of the available resources, which will be modified during the analysis.
- Finish is a vector of booleans indicating whether a particular process can finish. (or has finished so far in the analysis.)

Step1:Initialize Work to Available, and Finish to false for all elements.

Step2:Find an i such that both (A) Finish[i] == false, and (B) Need[i] < Work.

- This process has not finished, but could with the given available working set. If no such i exists, go to step 4.

Step3: Set Work = Work + Allocation[i], and set Finish[i] to true.

- This corresponds to process i finishing up and releasing its resources back into the work pool. Then loop back to step 2.

Step4: If finish[i] == true for all i, then the state is a safe state, because a safe sequence has been found.

When a request is made (that does not exceed currently available resources):

- pretend it has been granted, and then see if the resulting state is a safe one.
- If so, grant the request, and if not, deny the request with resource request algorithm, as follows:

Let Request[n][m] indicate the number of resources of each type currently requested by processes.

Step1: If Request[i] > Need[i] for any process i, raise an error condition.

Step2: If Request[i] > Available for any process i, then that process must wait for resources to become available.

- Otherwise the process can continue to step 3.

Step3: check the safety algorithm if it safe then,then the procedure for granting a request (or pretending to for testing purposes) is:

Available = Available - Request

Allocation = Allocation + Request

Need = Need - Request

Consider the following situation:

	Allocation	*Max*	*Available*	*Need*
	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		1 2 2
P_2	3 0 2	9 0 2		6 0 0
P_3	2 1 1	2 2 2		0 1 1
P_4	0 0 2	4 3 3		4 3 1

And now consider what happens if process P1 requests 1 instance of A and 2 instances of C. (Request[1] = (1, 0, 2))

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 $Need_0$ (7,4,3) > Work (3,3,2)

So P_0 must wait

But Need ≤ Work

For i = 1 — Step 2

$Need_1$ = 1, 2, 2

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

So P_1 must be kept in safe sequence

Step 3

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

	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 $Need_2$ (6, 0, 0) > Work (5,3, 2)

So P_2 must wait

For i=3 — Step 2

$Need_3$ = 0, 1, 1

Finish [3] = false and $Need_3$ (0, 1, 1) < Work (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 $Need_4$ (4, 3, 1) < Work (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 (7, 4, 3) < Work (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 $Need_2$ (6, 0, 0) < Work (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

Finish [i] = true for 0 ≤ i ≤ n — Step 4

Hence the system is in Safe state

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

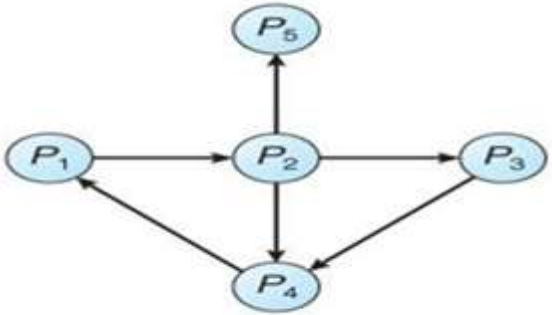
Deadlock Detection:

Single Instance of Each Resource Type

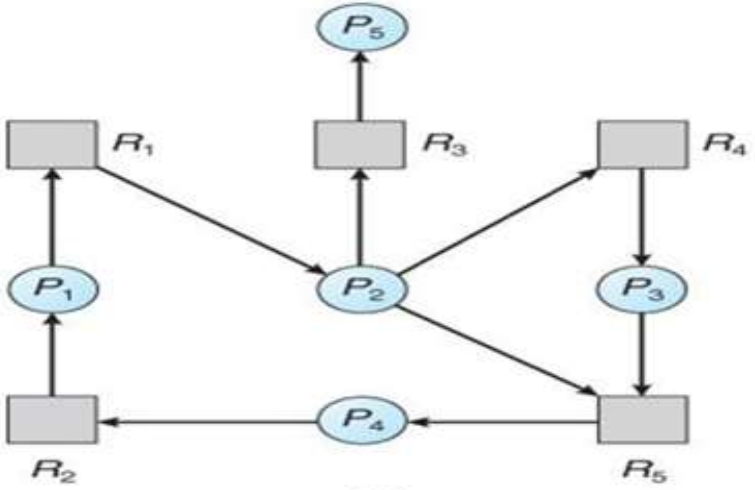
If each resource category has a single instance, then we can use a variation of the resource- allocation graph known as a wait-for graph.

A wait-for graph can be constructed from a resource-allocation graph by eliminating the resources and collapsing the associated edges

cycles in the wait-for graph indicate deadlocks.



(b)



(a)

Several Instances of a Resource Type

same as the Banker's algorithm, with two subtle differences:

Let Work and Finish be vectors of length m and n, respectively. Initialize Work = Available. For i = 0, 1, ..., n–1, if Allocationi ≠ 0, then Finish[i] = false. Otherwise, Finish[i] = true.

2. Find an index i such that both

   a. Finish[i] == false

   b. Requesti ≤ Work

   If no such i exists, go to step 4.

3. Work = Work + Allocationi

   Finish[i] = true

   Go to step 2.

4. If Finish[i] ==false for some i, 0≤i<n, then the system is in a deadlocked state. Moreover, if Finish[i] == false, then process Pi is deadlocked.

Consider, for example, the following state, and determine if it is currently deadlocked:

	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	

Now suppose that process P2 makes a request for an additional instance of type C, yielding the state shown below. Is the system now deadlocked?

Recovery From Deadlock

three basic approaches to recovery from deadlock:

- Inform the system operator, and allow him/her to take manual intervention.
- Terminate one or more processes involved in the deadlock
- Preempt resources.

Process Termination

Two basic approaches,

- Terminate all processes involved in the deadlock.
- Terminate processes one by one until the deadlock is broken.

Resource Preemption

When preempting resources to relieve deadlock, there are three important issues to be addressed:

- Selecting a victim -
- Rollback

  - Ideally one would like to roll back a preempted process to a safe state prior to the point at which that resource was originally allocated to the process.

- Starvation

  - How do you guarantee that a process won't starve because its resources are constantly being preempted?
  - One option would be to use a priority system, and increase the priority of a process every time its resources get preempted.
  - Eventually it should get a high enough priority that it won't get preempted any more.