Deadlocks (UNIT-6 Part-II)

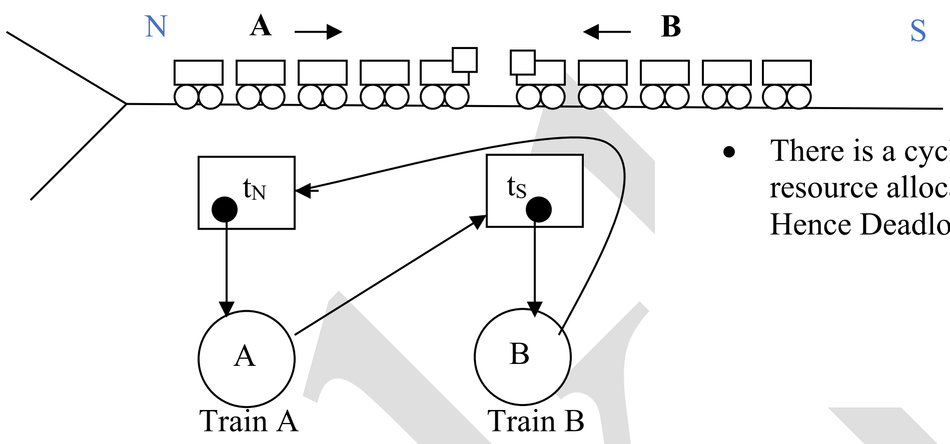
INDEX

Deadlocks

Definition of Deadlock: A set of processes is in a deadlock state when every process in the set is waiting for an event that can be caused by only other process in the set. (Events: Resource acquisition and release).

In a deadlock processes never finish executing and system resources are tied up, preventing other jobs from even starting.

Example of a deadlock: *Two trains approaching each other on a single track.*



- There is a cycle in resource allocation graph. Hence Deadlock.

Fig. Trains Deadlocked

Live lock / Indefinite postponement / Starvation: It is a situation in which some processes are making progress towards completion, but one or more other processes are locked out of the resource, because there is always another (higher priority) process that is using it. These processes are referred to be as starved.

Deadlock characterization:

Necessary conditions: a deadlock situation can arise if the following four conditions hold simultaneously in a system.

1. Mutual exclusion: at least one resource must be held in a non sharable mode, i.e. only one process at a time can use the resource.
2. Hold and wait: There must exist a process that is holding at least one resource and is waiting to acquire additional resources that are currently being held by other processes.
3. No Preemption: Resources cannot be preempted, i.e. a resource can be released only voluntarily by the process holding it after that process has completed its task.
4. Circular wait: there must exist a set $\{P_0, P_1, P_2, \ldots, P_n\}$ 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 held by P_0.

Qu. How the trains deadlock can be resolved?

Classification of Resources

- **Sharable resources:**

    - Can be used by several users at the same time.
    - Program code (if reentrant or pure)
    - Data areas if read only.
- **Non-sharable resources:**

    - Can only be used by one user at a time (Requires mutually exclusive access)
    - Data areas that need to be written.
    - Most external devices like printers.
    - The processor.

Non-sharable Resources can be further classified as

• **preemtable:**
-- Use of resource may be preempted and restarted later on.
-- e.g. The processor, The primary memory

• **non-preemptable:**
-- Use of resource can not be preempted.
-- e.g. Printers. (A new document cannot be printed until the previous is completed.)

Q: Give examples of deadlock, which are not related to a computer system **environment:**
Ans.
a. River crossing by two goats
b. Traffic deadlock example

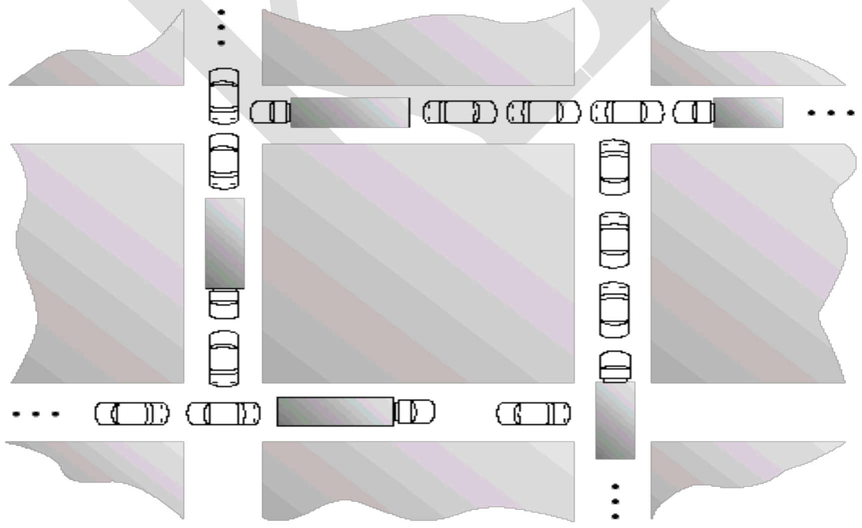


Figure: Traffic Deadlock Example :(Vehicle Gridlock)

Here each stream of cars holds one intersection but needs another intersection before it can proceed. Consider each intersection as a resource and each stream of cars as a process.

1. Mutual exclusion condition applies, since only one stream of vehicles can be on an intersection of the street at a time.
2. Hold-and-wait condition applies, since each stream of vehicles is occupying an intersection, and waiting for another intersection.
3. No-preemptive condition applies, since an intersection that is occupied by a stream of vehicles cannot be taken away from it.
4. Circular wait condition applies, since each stream of vehicles is waiting on the next stream of vehicles to move. That is, each stream of vehicles in the traffic is waiting for an intersection held by the next stream of vehicles in the traffic in circular manner.

Q: Give examples of deadlock in a Computer System:

ANS:

(A) Deadlock involving single resource type: -
System with 4 tape drives and two processes, if each process holds two tape drives but needs three, deadlock occurs.

(B) Deadlock involving different resource types: -
System with one printer and one card reader and two processes. Suppose that process P is holding the card reader and process Q is holding the printer if P now requests the printer and Q requests the card reader a deadlock occurs.

***Qu.**: a) Show that the four necessary conditions for deadlock indeed hold in the examples (A,B) given above.*
b) Come up with a simple rule that will avoid deadlocks.

***ANS:** Allow each process to acquire all the resources it needs in advance.*

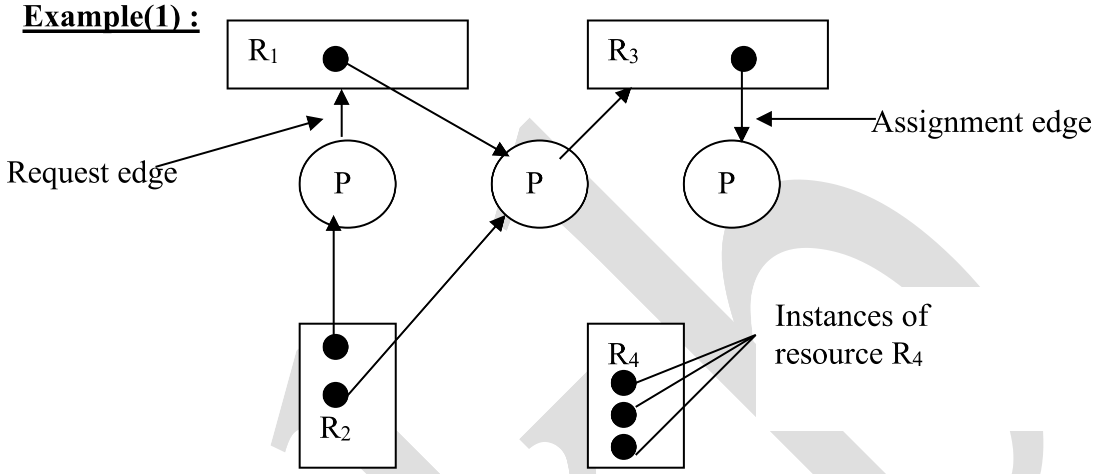
Q: How process should use a resource? (Resource use protocol)
A: Request ->Use -> Release.

Q: How OS Manages Resources?
A: The request and release of resources are system calls.
e.g. Device: request device, release device.
File: open, close.
Memory: allocate, free.
Others: with wait and signal operations on semaphores.

System wide resources allocation table is maintained by OS wherein it records whether each resource is free or allocated, and if a resource is allocated to which process. If a process requests a resource that is currently allocated to another process, it can be added to a queue of processes waiting for this resource.

Resource Allocation graph(RAG): It is a directed graph used to describe deadlocks more precisely. It consists of set of vertices V and a set of edges E. the set of vertices is partitioned into two types.
P = {P_1, P_2, …, P_n}, the set containing all the processes in the system and R = {R_1, R_2,…, R_m}, the set consisting of all the resource types in the system.

Example(1) :



Above resource allocation graph depicts the following situation: -

- The sets P, R and E

  P = {P1,P2,P3}
  R = {R1,R2,R3,R4}
  E = {$\underbrace{\text{P1->R1, P2->R3,}}_{\text{Request Edges}}$ $\underbrace{\text{R1->P2, R2->P1, R3->P3}}_{\text{Assignment edges}}$}
- Resource instances:
  One instance of resource type R1
  Two instances of resource type R2
  One instances of resource type R3
  Three instances of resource type R4
- Process states:
  Process P1 holding one instance of type R2 and waiting for an instance of type R1
  Process P2 holding one instance of type R2 and waiting for an instance of type R3
  Process P3 holding one instance of type R3

RAG and Deadlock condition:

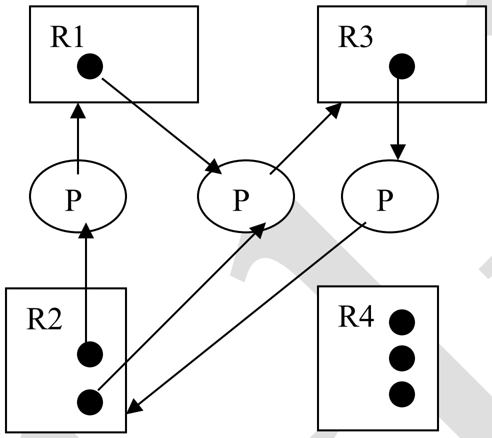
(1) If graph contains no cycles, then no process in the system is deadlocked.
(2) Existence of a cycle is necessary and sufficient condition for deadlock if each resource type has exactly one instance.

(3) If each resource type has several instances, then a cycle does not necessarily imply that a deadlock occurred. In this case a cycle in the graph is necessary but not a sufficient condition for the existence of a deadlock.

Wait for graph: We obtain this graph from the resource allocation graph by removing the nodes of type resource and collapsing the appropriate edges. In this graph we show the processes waiting on each other.

Q1*: show that request from process P3 for an instance of a resource type R2 in example 1 leads to a deadlock.*
Ans.: Adding the request edge P3->R2 in example 1.

There are two cycles in the graph hence there is a chance that the given system is deadlock.



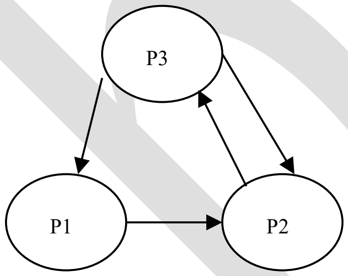
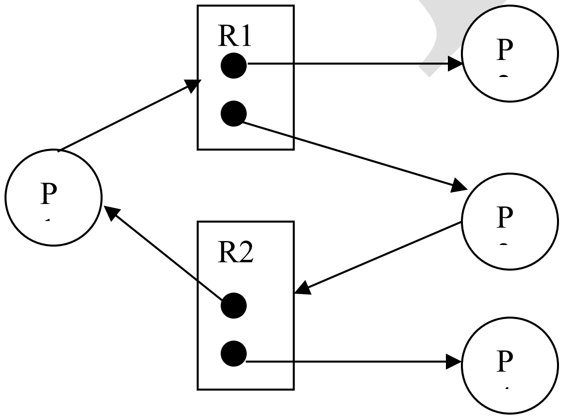


Fig. Wait for graph for RAG on the left

The four necessary conditions for deadlock hold for the above graph hence the above system is in a deadlock state.

Q2: Show that the resource allocation graph shown below is not representing a deadlock system OR find whether the following system is deadlocked. Justify your answer.
Ans: As there is a cycle in the graph the system is unsafe. But it is not deadlocked because the process P4 may release its instance of resource type R2. That resource can then be allocated to P3, causing the circular wait/cycle to break.



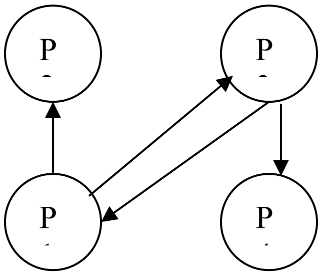


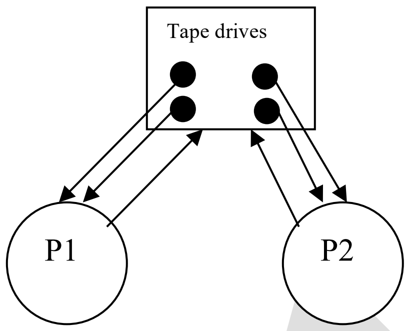
Fig. Wait for graph for RAG on the left

Q: Draw resource allocation graphs for deadlock examples A & B

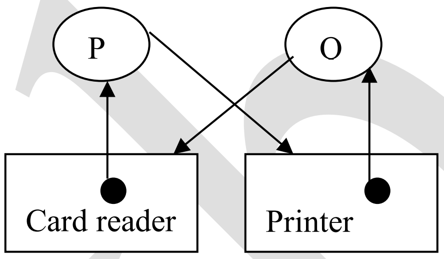
(A) Deadlock involving single resource type: -
System with 4 tape drives and two processes, if each process holds two tape drives but needs three, deadlock occurs.

(B) Deadlock involving different resource types: -
System with one printer and one card reader and two processes. Suppose that process P is holding the card reader and process Q is holding the printer if P now requests the printer and Q requests the card reader a deadlock occurs.

Example A



Example B



Methods for handling deadlocks:

Sr. No.	Method	How
1.	Prevention (To ensure that system will never enter the deadlock state)	• Deny one of the necessary conditions
2.	Avoidance (To ensure that system will never enter the deadlock state)	• Banker's Algorithm (Requires knowledge of request and release of resources by each process), • Resource allocation graph algorithm (with only one instance of each resource type.)
3.	Detection and recovery	Allow the system to enter the deadlock state and then recover.
4.	Combined approach	Divide resources into different resource classes. For each resource class employ different approach .
5.	Ignore the problem altogether. (Ostrich Algorithm)	Used by most OS, Including UNIX and MS Windows

Note: Methods 1 & 2 see that a deadlock does not occur.

Deadlock Prevention:

For a deadlock to occur each of the four necessary conditions must hold. By ensuring that at least one of these conditions cannot hold, we can prevent the occurrence of a deadlock.

1) Mutual Exclusion:

i) Shareable resources: Mutually exclusive access is not required; hence these resources cannot be involved in a deadlock. (e.g. read only files)

ii) Non-Shareable Resources: Some resources like printer are intrinsically non shareable. Hence mutually exclusive access is required.
Hence in general it is not possible to prevent deadlocks by denying mutual exclusion condition.

2) Hold and wait: To deny this condition we must guarantee that whenever a process requests a resource it does not hold any other resources. The different protocols to achieve this are

a) Each process to request and be allocate all of its resources before it begins execution.

b) Allow a process to request only when it has none. Here before requesting an additional resource the process has to release any previously hold resource.

Disadvantages of denying hold and wait:

i) Resource utilization may be very low, since many of the resources may be allocated but unused for a long period of time.

ii) Starvation is possible. A process that needs several popular resources may have to wait indefinitely because at least one of the resources that it needs is always allocated to some other process.

3) No Preemptions: To deny this condition following protocol observed-

a) If a process that is holding some resources, requests additional resources that cannot be immediately allocated to it (i.e. the process must wait) then all resources currently being held are preempted. These resources are added to the list of resources. The process will only be restarted when it can regain its old resources as well as the new one that it is requesting.

b) If a process requests some resources that are not available, we check whether they are allocated to some other process that is waiting for additional resources. If so we preempt the desired resources from the waiting process and allocate them to the requesting process.

Some type of resources such as partial updated files, printers cannot be preempted without corrupting the system. Therefore, preemption is possible only for certain type of resources (such as CPU & Memory), and it can be applied when the benefits of deadlock prevention outweigh the cost of state save restore operations.

***Qu.**: Certain system (THE system) allows printer to be preempted from a process. Explain*

How can this system eliminate the problem of corrupted printout?

Ans: It can finish the current page and remember the next page number to be printed for a process. OS can maintain a table storing process IDs and page numbers to be printed next.

4) Circular wait: In order to ensure that the circular wait condition never holds we may impose *a total ordering of different classes of system resources*. In this approach system resources are divided into different classes Cj where j=1, . . ., n. Deadlocks are prevented by requiring all processes to request and acquire their resources in a strictly increasing order of the specified system resource classes. Moreover, acquisition of all resources within a given class must be made with a single request, and not incremental. In other words, once a process acquires a resource belonging the class Cj, it can only request resources of class C_{j+1} or higher thereafter.

Ex. Two processes require one tape drive and one printer.

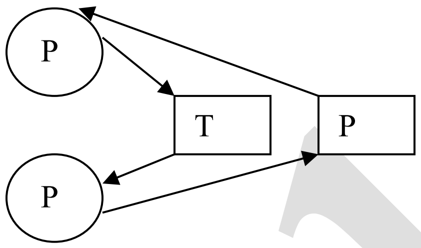


Fig. Without resource ordering

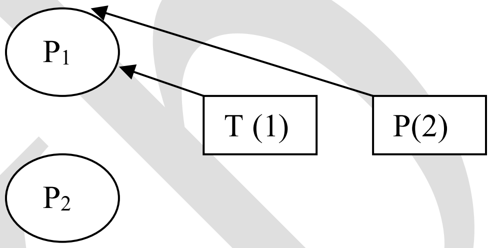
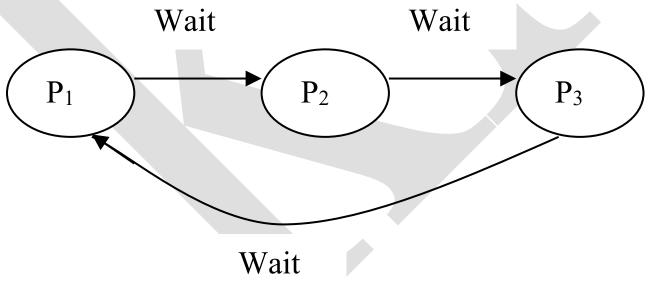


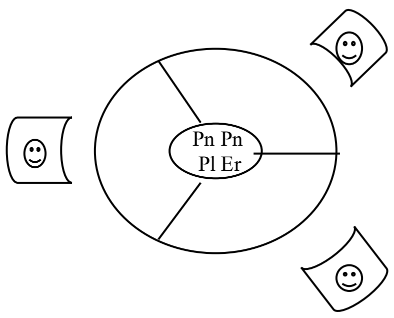
Fig. with resource ordering

In the RAG below assume that P_i is a process holding a resource in class Ci. Process P_i cannot possibly wait for any process that is itself waiting for a resource in class C_i or lower. Hence circular wait condition possibility is eliminated.



The type of wait shown in above figure is impossible since a process holding class 3 resources can't wait for class 1 resource (since 3>1)

Example: 3 students are writing records with 2 ink pens 1 pencils and 1 erasure, on a partitioned table with the slot containing these four items open for access to three persons, as shown in figure below.



	Class number	Number of instances
Ink pen	1	2
Pencil	2	1
Erasure	3	1

Let the system state without resource ordering is

Processes	Max. Demand			Allocation			Need = MAX - Alloc		
	R1	R2	R3	R1	R2	R3	R1	R2	R3
S1	1	1	1	1	0	1	0	1	0
S2	1	1	1	1	0	0	0	1	1
S3	0	1	1	0	1	0	0	0	1

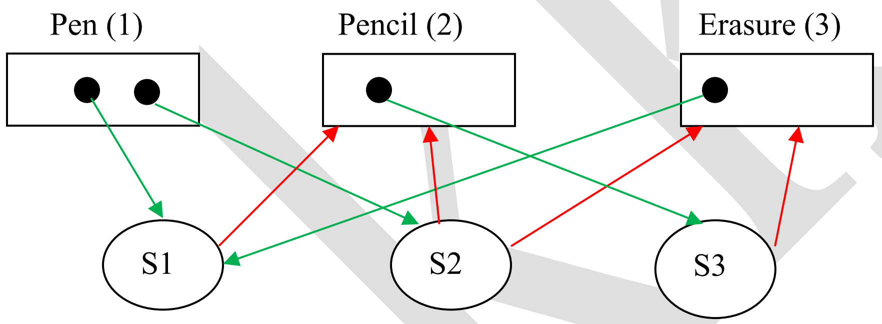


Fig. Resource Allocation graph for the above example <u>without</u> resource ordering

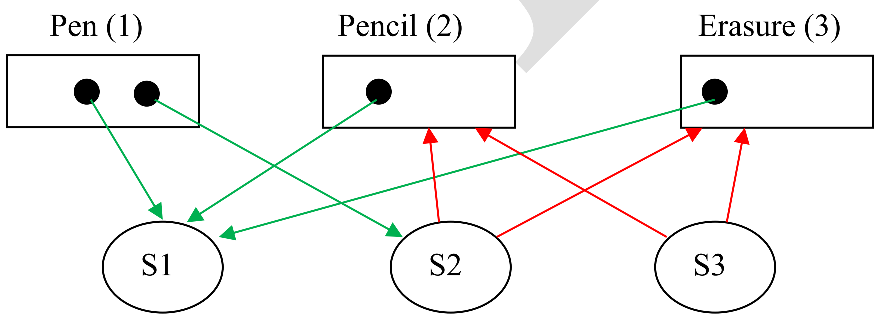


Fig. Resource Allocation graph for the above example <u>with</u> resource ordering

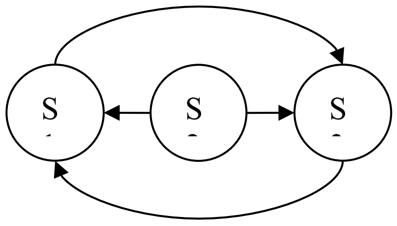


Fig. Wait for graph without resource ordering (Note cycle in this graph: Hence Unsafe state)

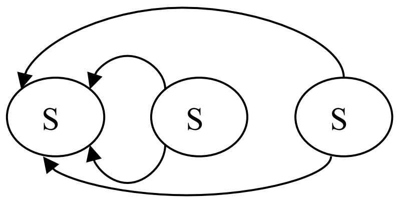


Fig. Wait for graph with resource ordering (No cycles; hence safe state)

Deadlock avoidance:

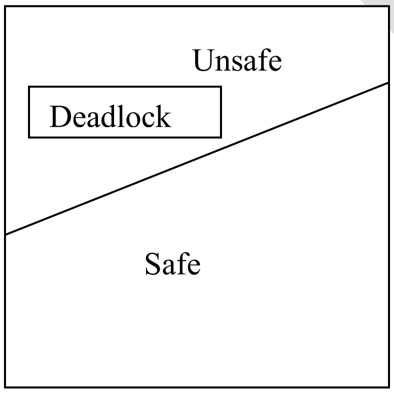
i) This approach requires additional information about how resources are to be requested.
ii) With this knowledge of the complete sequence of requests and release for each process we can decide for each request whether request can be satisfied or whether the process must wait to avoid a possible future deadlock.

Whenever a process requests a resource that is currently available, the system must decide if the resource can be immediately allocated or if the process must wait. The request is granted only if it leaves the system in a safe state.

In this scheme if a process requests a resource, which is currently available, it may still have to wait. Thus, resource utilization may be lower than without a deadlock avoidance algorithm.

Resource allocation state: It is defined by number of available and allocated resources, and the maximum demands of the processes.

Safe, Unsafe and deadlock state spaces: -



i) Deadlock state is an unsafe state.
ii) Not all unsafe states are deadlocks.
iii) In an unsafe state the OS can not prevent processes from requesting resources in such a way that a deadlock occurs (i.e. behavior of the process controls the unsafe state.)
iv) It is possible to go from a safe state to unsafe state.

Safe state: A state is safe if the system can allocate resources to each process (up to it's maximum) in some order and still avoid a deadlock.

Example: Consider a system with total 12 magnetic tape drives and three processes P0, P1, P2. Resource allocation state of these processes is as follows (at time T0)

Process	Maximum demand	Allocations	Available = Total - $\sum$ Alloc = 12 - (5+2+2)	Need = Max- Alloc (Remaining)
P_0	10	5	3	5
P_1	4	2		2
P_2	9	2		7

i) Find out whether the system is safe. (Safety Algo.)

ii) Find out (at time $T_1 > T_0$) whether the system will grant the request by process P_2 for one more tape drive. (Resource Request Algo.)

Ans : The content of the work vector is given as processes are allocated additional tapes.
Work=available (initially)

i) Safety Algorithm:

Work=Work + Allocation	Remarks
3	Initially (At time T_0)
5	P_1 finished
10	P_0 finished
12	P_2 finished

There exists a safe sequence <P_1, P_0, P_2>. Hence the given resource allocation state is safe.

ii) Applying resource request algorithm :

Pretend that the request by P_2 for one more tape drive is granted. The modified state is shown below

Process	Maximum demand	Allocations	Available =Total-$\sum$ Alloc =12-(5+2+3)	Need=Max- Alloc (Remaining)
P_0	10	5	2	5
P_1	4	2		2
P_2	9	3		6

Now applying safety algorithm on the above resource allocation state

Work	**Remarks**
2	Initially (At time T_0)
4	• P_1 finishes. • Now no process can finish hence unsafe state, hence the request can't be granted.)

Banker's Algorithm: A deadlock avoidance algorithm:

The Banker's algorithm is named after its analogy to the way a bank handles its resources. The algorithm was originally introduced by Edsger Dijkstra (a Dutch Scientist) in 1965 and was later popularized in the context of resource allocation in operating systems.

The name "Banker's algorithm" is derived from the analogy that the operating system acts as a "banker" that allocates resources to different processes. Just as a banker needs to ensure that it can satisfy the financial needs of its customers, the operating system using the Banker's algorithm aims to allocate resources in a way that avoids deadlock and ensures the system's stability.

Algorithm:

i) When a new process enters the system, it must declare the max number of instances of each resource type that it may need. This number may not exceed the total number of resources in the system.

ii) When a user process requests a set of resources, it must be determined whether the allocation of these resources will leave the system in safe state (using safety algorithm). If so, the resources are allocated otherwise the process must wait until some other process release enough resources.

Implementation: Let n be the number of processes in the system, let m be the number of resource types.

Data structure required:

- Available: A vector of length m, indicating number of available resources of each type.
- Max: An n x m matrix defining the max demand of each process.
- Allocation: An n x m matrix defining the no. of resources of each type currently allocated to each process.
- Need: An n x m matrix indicating the remaining resource needs of each process.
  Need [i, j] = max [i,j] – Allocation [i,j]
- Each matrix has first dimension as process number and second as resource number

Qu. What do you mean by the following?

i) Available[j] = k ii) Max[i,j] = k iii) Allocation[i,j] = k
iv) Need[i,j] = k

Example: - Consider a system with five processors {p_0, p_1, p_2, p_3, p_4} and three resource types {A, B, C}. Resource type A has 10 instances, B has 5, C has 7 instances.
Suppose that at time T_0 following snapshot of system is taken.
Total = [10 5 7], Available=[10 5 7] – [7 2 5] = [3 3 2]
Find out whether the system is in a safe state.

Process	Max demand A B C	Allocations A B C	Available = Total - ∑ Alloc A B C	Need = Max - Alloc (Remaining)	Work	Comment
p_0	7 5 3	0 1 0	3 3 2	7 4 3	3 3 2	= Available
p_1	3 2 2	2 0 0		1 2 2	5 3 2	p_1 finished
p_2	9 0 2	3 0 2		6 0 0	7 4 3	p_3 finished
p_3	2 2 2	2 1 1		0 1 1	7 4 5	p_4 finished
p_4	4 3 3	0 0 2		4 3 1	7 5 5	p_0 finished
	Total	**7 2 5**			10 5 7	p_2 finished

As finish[i]=true for all i , the system is in a safe state. (See the algorithm below)

Banker's Algorithm

Safety Algorithm:	**Resource request algorithm**
1) Let work and finish be vectors of length m and n respectively. Initialize work = Available and Finish[i] = false for i=1,2,…,n. 2) Find an i such that both a. a)Finish[i]=false b)Need[i] <= Work if no such i exists then go to step 4. 3) Work=Work+Allocation$_i$ Finish[i]=true Goto step 2 4) If Finish[i] = true for all i then the system is in safe state, else it is in unsafe state (Time Complexity m*n^2)	If Request$_i$ be the request vector for process P$_i$ If Request$_i$ [j] = k, then process Pi wants k instances of resource type Rj, when a request for resource is made by process Pi, the following actions are taken. 1) If Request$_i$ <= Need$_i$, goto step 2 otherwise raise an error condition since the process has exceeded its maximum claim. 2) If Request$_i$ <= Available goto step 3 otherwise Pi must wait since the resources are not available. 3) Let the system pretend that the requested resources are allocated to process P$_i$ by modifying the state, as follows: Available$_i$ = Available$_i$-Request$_i$; Allocation$_i$ = Allocation$_i$ + Request$_i$; Need$_i$ = Need$_i$ - Request$_i$ 4) Now **apply the safety algorithm** to find whether the system is in safe state. If the resulting resource allocation state is safe the transaction is completed and process P$_i$ is allocated its resources. However if the new state is unsafe, then P$_i$ must wait for Request$_i$ and the old state is restored.

Programming Assignment:
Simulate the Banker's Algorithm for deadlock avoidance using a computer program

Deadlock avoidance for resource allocation system with only one instance of each resource type(using resource allocation graph algorithm).

In addition to the request and assignment edges, we introduce a new type of edge, called the claim edge.
Claim edge A claim edge Pi->Rj indicates that process P1 may request resource Rj at some time in the future. This edge resembles the request edge in direction but is represented by a dashed line.

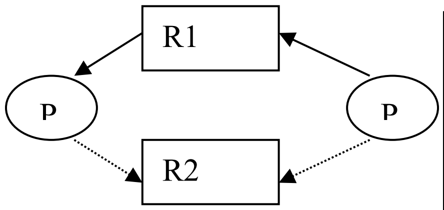


Fig. Resource allocation graph for deadlock avoidance

Suppose P2 requests R2, claim edge is P2->R2 converted to assignment edge R2->P2, as this results in a cycle the request is not granted.

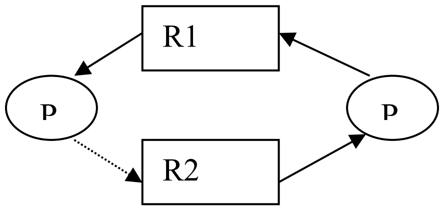


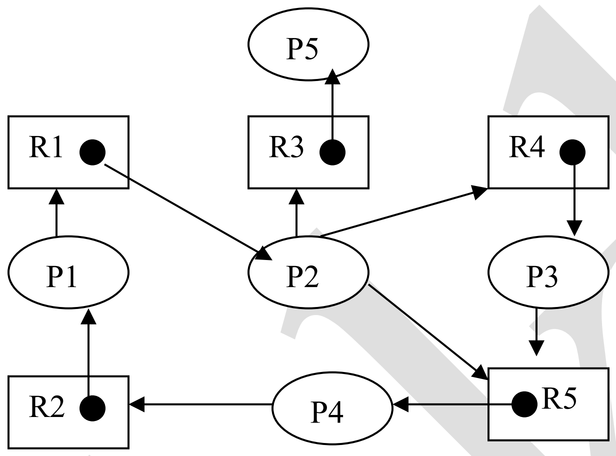
Fig. An unsafe state in resource allocation graph

Deadlock Detection: If a system does not employ either a deadlock prevention or deadlock avoidance algorithm then a deadlock situation may occur. In this environment, the system may provide:

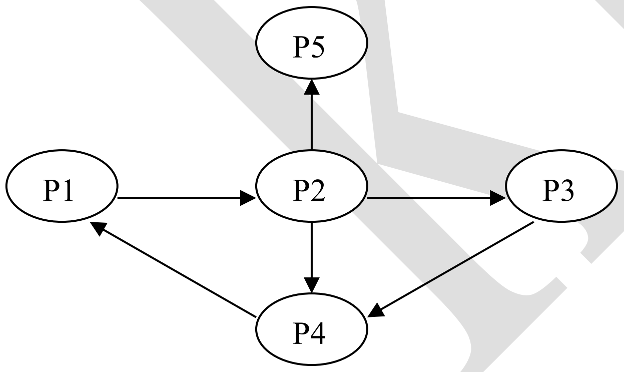
- An algorithm that examines the state of the system to determine whether a deadlock has occurred.
- An algorithm to recover from the deadlock**.**

Deadlock Detection for resource allocation system with single instance of each resource type:

Q3: Show that the following system is in the deadlock state.



Ans: Converting the above graph to **wait for graph.**



Note: An algorithm to detect a cycle in a graph requires an order of n^2 operations, where n is number of vertices in the graph.

This is an example of resource allocation system with single instance of each resource type. Hence the presence of cycle in the graph shows that the system is deadlocked.

Deadlock avoidance algorithm Vs deadlock detection algorithm:

- Deadlock Detection Algorithm: The primary objective of a deadlock detection algorithm is to identify whether a deadlock has occurred in the system.
- Deadlock Avoidance Algorithm: The primary objective of a deadlock avoidance algorithm is to prevent the occurrence of deadlocks by carefully allocating resources.

Deadlock detection in system with several instances of a resource type

The algorithm uses additional data structure ‘Request which is an n*m matrix. It indicates the *current request of each process*.

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

Example (a): Consider a system with resources A, B, C with instances 7,2 and 6 respectively. Suppose that at time T0 we have the following resource allocation state.

State	Allocations A B C	Requests A B C	Available / Work A B C	Comments
P_0	0 1 0	0 0 0	0 0 0	Initially
P_1	2 0 0	2 0 2	0 1 0	P_0 finished
P_2	3 0 3	0 0 0	3 1 3	P_2 finished
P_3	2 1 1	1 0 0	5 2 4	P_3 finished
P_4	0 0 2	0 0 2	5 2 6	P_4 finished
Total	7 2 6		7 2 6	P_1 finished

Applying detection algorithm, we now claim that the resource allocation state is not in a deadlock state. Indeed, if we execute our algorithm, we will find that the sequence $< P_0, P_2, P_3, P_4, P_1 >$ will result in Finish[i] = true for all.

(b) Suppose that process P_2 makes one additional request for an instance of type C. The request matrix is modified as follows:

	Request A B C	Available/Work A B C	
P_0	0 0 0	0 0 0	Initially
P_1	2 0 2	0 1 0	P_0 finished
P_2	0 0 1		
P_3	1 0 0		
P_4	0 0 2		

After this for no process $Request_i <= Work$ Hence we claim that system is now deadlocked and processes involved are P_1, P_2, P_3 and P_4

The algorithm (Deadlock Detection)

1. Let Work and Finish be vectors of length m and n respectively.
   Initialize Work := Available. For i = 1,2,…,n, if Allocation_i != 0, then Finish[i] := false; otherwise Finish [i] = true
   /*only processes holding resources (Allocation_i != 0) can be involved in a deadlock hence making Finish[i] := false only for such processes.*/
2. Find an index i such that both
   a) Finish[i] = false
   b) $\text{Request}_{i <=}$ Work
   If no such i exists goto step 4
3. Work := Work + Allocation_i
   Finish [i]: =true
   Goto step 2.
4. If finish[i] = false, for some i, 1<=i<=n, then the system is in a deadlock state. Moreover, if Finish [i] = false, then process Pi is deadlocked.

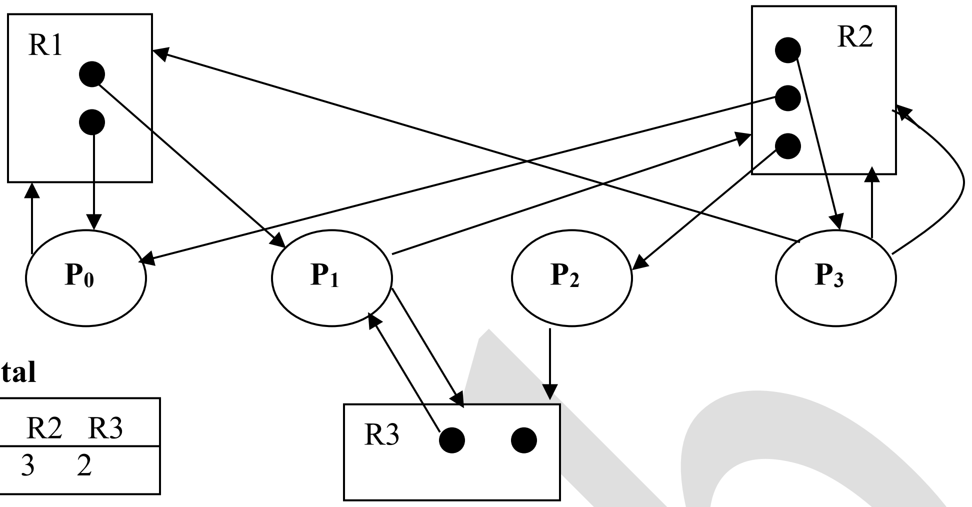
Ex. Consider a system consisting of four resources of the same type that are shared by three processes, each of which needs at most two resources. Show that the system is deadlock free.

Detection algorithm uses:

When should we invoke the detection algorithm?

1. If deadlocks occur frequently, then the detection algorithm should be invoked frequently.
2. In the extreme we can invoke the detection algorithm every time a request for allocation cannot be granted immediately
3. A less expensive alternative is simply to invoke the algorithm at less frequent intervals - for example once per hour or whenever CPU utilization drops below 40 %.

Example: (Deadlock detection)
Consider the given resource allocation graph.



Total

R1	R2	R3
2	3	2

Find if the system is in deadlock state.

Solution: Applying deadlock detection algorithm

Process R1 R2 R3	Allocation R1 R2 R3	Request R1 R2 R3	Available R1 R2 R3	Work R1 R2 R3	Comment
P_0	1 1 0	1 0 0	0 0 1	0 0 1	Initially
P_1	1 0 1	0 1 1		0 1 1	P_2 finished
P_2	0 1 0	0 0 1		1 1 2	P_1 finished
P_3	0 1 0	1 2 0		2 2 2	P_0 finished
				2 3 2	P_3 finished

From the above table it is clear that the system is not in a deadlock state, since all the processes have finished their execution.

Recovery from deadlock:

Recovering from deadlock involves taking appropriate actions to resolve the deadlock situation and restore the system to a functioning state. Here are some common ways to recover from deadlock:

Process Termination:
One approach is to terminate one or more processes involved in the deadlock. By removing one or more processes from the system, the resources they hold will be released, allowing other processes to continue execution. The terminated processes can then be restarted later.

Resource Preemption:
Resource preemption involves forcibly reclaiming resources from one or more processes to allocate them to other processes. This approach requires a system that supports resource preemption. By preempting resources from deadlocked processes, it becomes possible to break the circular wait condition and allow other processes to proceed.

Rollback:

In certain cases, it may be possible to roll back the state of some processes to a previous checkpoint, where they held fewer resources. This approach involves restoring the system to a state before the deadlock occurred and re-executing the affected processes from that point. The aim is to release resources and reallocate them in a different order to avoid deadlock.

Issues:

- Selecting a victim (no. of resources a process is holding, amount of time a process has consumed so far during its execution)
- Rollback (Total, Partial)
- Starvation (Some process is always picked up as a victim: Select a victim based on no. of rollbacks of a process.)

The Ostrich algorithm:

"Stick your head in the sand and pretend that there is no problem."

- The easiest way to deal with the problem
- How is it possible to use such an algorithm?
  - Other errors are much more frequent
  - To not add limitations to the operating system
- The ostrich algorithm is both Windows and UNIX approved, because most users would prefer an occasional deadlock to a very restrictive, inconvenient, complex, and slow system.
- However the recent versions of these OS provide deadlock detection support, that can be enabled or disabled by the user.

Ex. Consider the following resource-allocation policy. Requests and releases for resources are allowed at any time. If a request for resources cannot be satisfied because the resources are not available, then we check any processes that are blocked, waiting for resources. If they have the desired resources, then these resources are taken away from them and are given to the requesting process. The vector of resources for which the process is waiting is increased to include the resources that were taken away.

For example, consider a system with three resource types and the vector *Available* initialized to (4,2,2). If process P0 asks for (2,2,1), it gets them. If P1 asks for (1,0,1), it gets them. Then, if P0 asks for (0,0,1), it is blocked (resource not available). If P2 now asks for (2,0,0), it gets the available one (1,0,0) and one that was allocated to P0 (since P0 is blocked). P0's *Allocation* vector goes down to (1,2,1) and its *Need* vector goes up to (1,0,1).

a. Can deadlock occur? If you answer "yes", give an example. If you answer "no," specify which necessary condition cannot occur.

b. Can indefinite blocking occur? Explain your answer.

Ans:

a) Deadlock cannot occur because preemption exists.

b) Yes. A process may never acquire all the resources it needs if they are continuously preempted by a series of requests such as those of process P0.