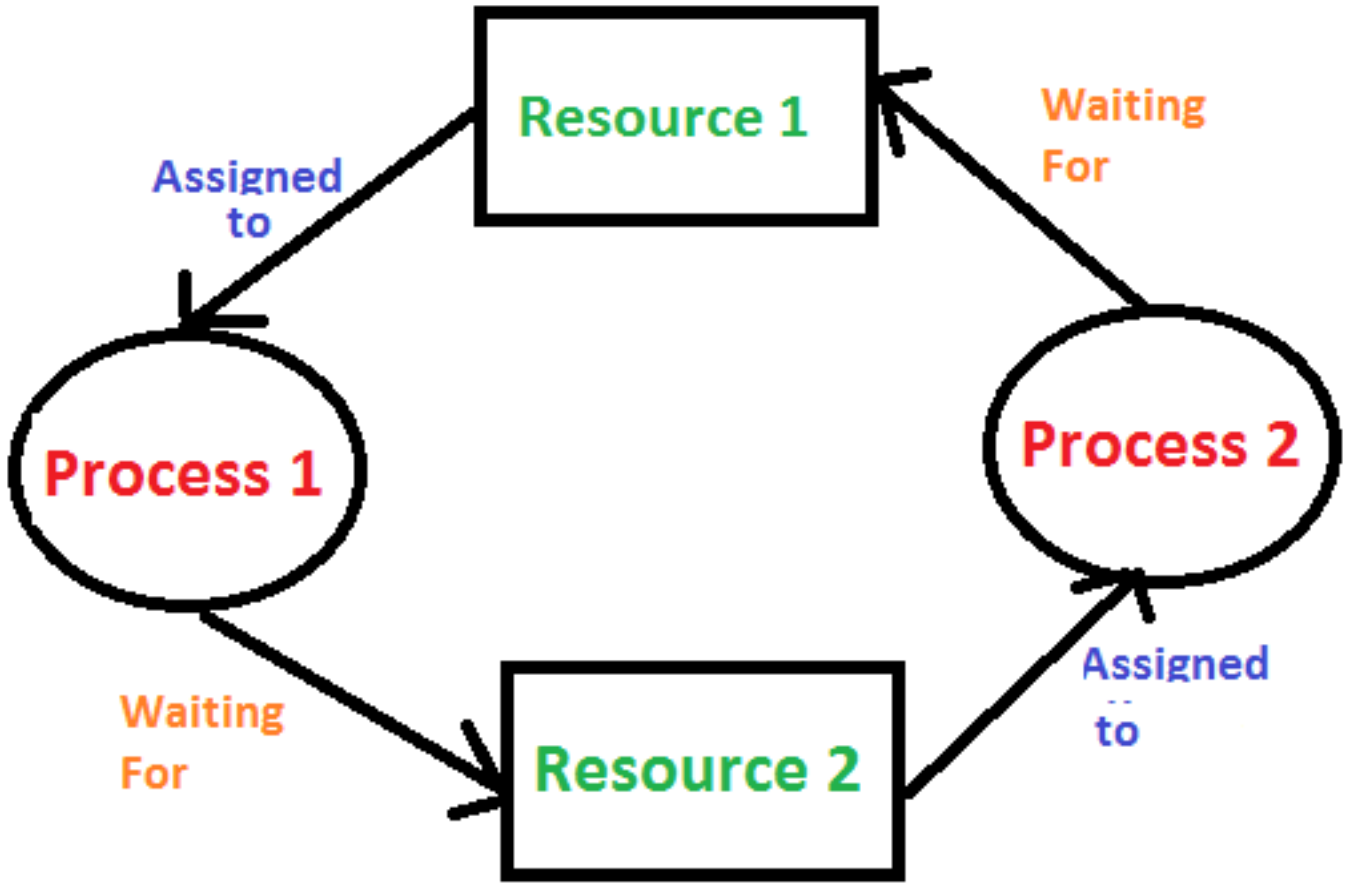
UNIT 3: DEADLOCK

DEADLOCK

- ***Deadlock*** is a situation where a set of processes are blocked because each process is holding a resource and waiting for another resource acquired by some other process.
- For example, in the below diagram, Process 1 is holding Resource 1 and waiting for resource 2 which is acquired by process 2, and process 2 is waiting for resource 1.
- A process in operating systems uses different resources and uses resources in the following way.
  1) Requests a resource
  2) Use the resource
  2) Releases the resource

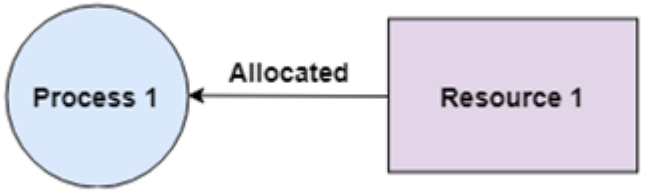

Resource 1
Assigned
to
Waiting
For
Process 1
Process 2
Waiting
For
Resource 2
Assigned
to

Deadlock Characterization

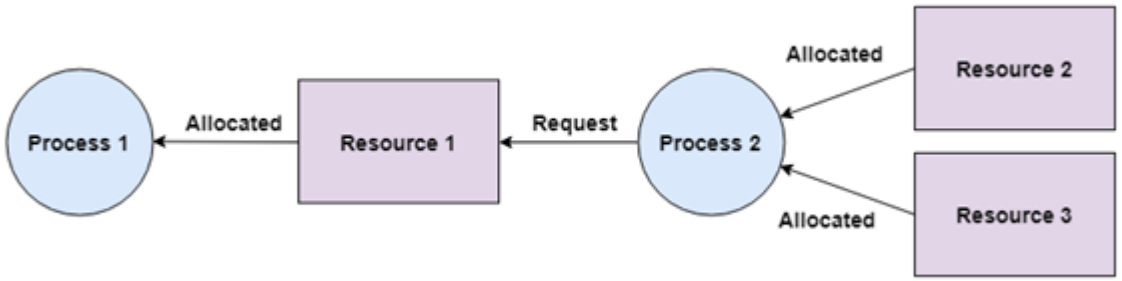
- In a deadlock, processes never finish executing and system resources are tied up preventing other jobs from starting.
- A deadlock occurs if the four Coffman conditions hold true. But these conditions are not mutually exclusive. They are given as follows −

- ✓ Mutual exclusion
- ✓ Hold and wait
- ✓ No pre-emption
- ✓ Circular Wait

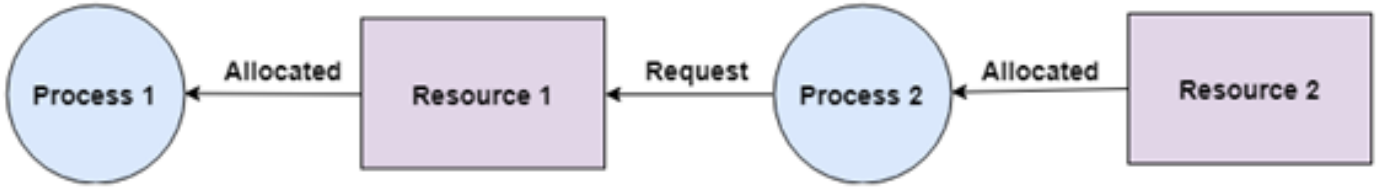
1.Mutual Exclusion : A resource can only be shared in mutually exclusive manner. It implies, if two process cannot use the same resource at the same time.



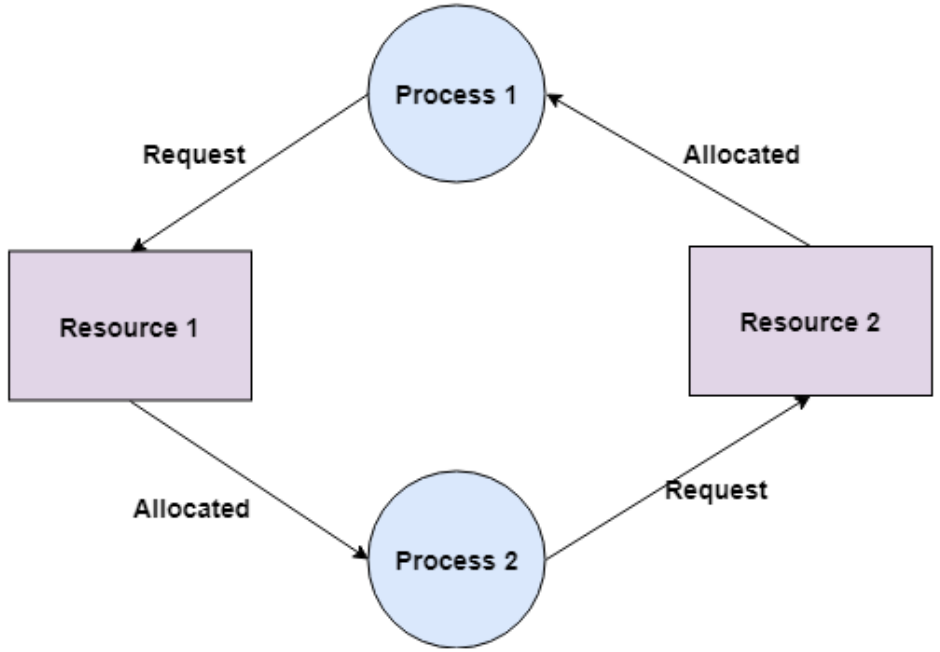
2. Hold and Wait: A process waits for some resources while holding another resource at the same time. In the diagram given below, Process 2 holds Resource 2 and Resource 3 and is requesting the Resource 1 which is held by Process 1.



3. No pre-emption: The process which once scheduled will be executed till the completion. No other process can be scheduled by the scheduler meanwhile. In the diagram below, Process 2 cannot pre-empt Resource 1 from Process 1. It will only be released when Process 1 relinquishes it voluntarily after its execution is complete.



4. Circular Wait: All the processes must be waiting for the resources in a cyclic manner so that the last process is waiting for the resource which is being held by the first process. For example: Process 1 is allocated Resource2 and it is requesting Resource 1. Similarly, Process 2 is allocated Resource 1 and it is requesting Resource 2. This forms a circular wait loop.



Methods for Handling Deadlocks

1. Deadlock Ignorance
2. Deadlock Prevention
3. Deadlock Avoidance
4. Deadlock Detection and Recovery

1. Deadlock Ignorance

- It is the most popular method and it acts as if no deadlock and the user will restart. As handling deadlock is expensive to be called of a lot of codes need to be altered which will decrease the performance so for less critical jobs deadlock are ignored.
- Ostrich algorithm is used in deadlock Ignorance.
- The Ostrich algorithm: “Put your head in the sand approach”
- i. If the likelihood of a deadlock is sufficient small and the cost of avoiding a deadlock is sufficient high it might be batter to ignore the problem.
- ii. Clearly not a good philosophy for nuclear missile launchers.
- iii. For embedded system the program runs are fixed in advanced.

2. Deadlock Prevention

- The strategy of deadlock prevention is to design a system in such a way that the possibility of deadlock is excluded a prior.
- There are two methods for deadlock prevention:

a) Indirect method: This indirect method of deadlock prevention is to prevent the occurrence of one of the three necessary conditions.

b) Direct method: This direct method of deadlock prevention is to prevent the occurrence of circular wait condition.

- Deadlock prevention strategies are very conservative; they solve the problem of deadl ock by limiting access to resources and by imposing restrictions on processes.

Mutual Exclusion

Mutual exclusion condition must hold for non-sharable resources. If access to a resource requires mutual exclusion, then mutual exclusion must be supported by the operating system. Some resources, such as files, may allow multiple accesses for reads but only exclusive access for writes. In this case, deadlock can occur if more than one process requires write permission.

Hold and Wait

The hold and wait condition can be eliminated by forcing a process to release all resources held by it whenever it requests a resource that is not available. For example, process copies data from a floppy disk to a hard disk, sort a disk file and then prints the results to a printer. If all the resources must be requested at the beginning of the process, then the process must initially request the floppy disk, hard disk and a printer. It will hold the printer for its entire execution, even though it needs the printer only at the end. In these two method, resource utilization is low in the first method and second method is affected by the starvation.

No Pre-emption

This condition is also caused by the nature of the resource. This condition can be prevented in several way.

- If a process holding certain resources is denied a further request. That process must release its original resources and if necessary request them again, together with additional resource.
- If a process requests a resource that is currently held by another process, the operating system may pre-empt the second process and require it to release its resources.
- In general, sequential I/O devices cannot be pre-empted.
- Pre-emption is possible for certain types of resources, such as CPU and main memory.

Circular Wait

One way to prevent the circular wait condition is by linear ordering of different types of system resources. In this, system resources are divided into different classes. If a process has been allocated resources of type R, then it may subsequently request only those resource types following R in the ordering.

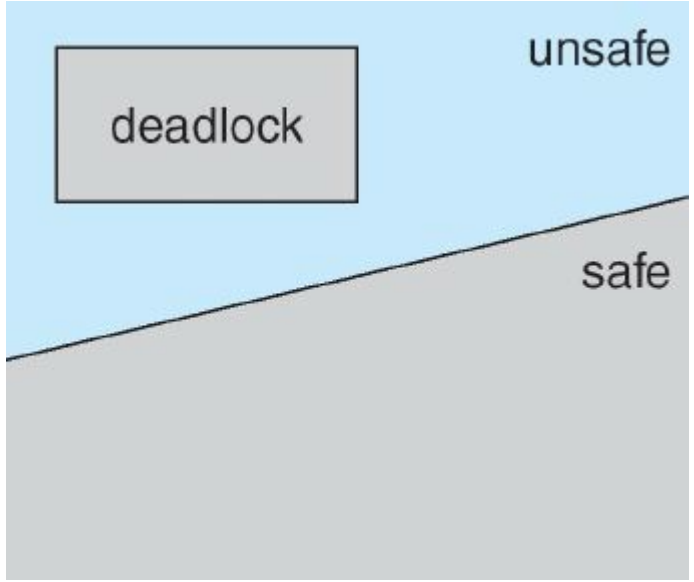
For Example, if P1 process is allocated R5 resources, now next time if P1 ask for R4, R3 lesser than R5 such request will not be granted, only request for resources more than R5 will be granted.

3. Deadlock Avoidance

- In deadlock avoidance, the request for any resource will be granted if the resulting state of the system doesn't cause deadlock in the system. The state of the system will continuously be checked for safe and unsafe states.
- In order to avoid deadlocks, the process must tell OS, the maximum number of resources a process can request to complete its execution.
- The simplest and most useful approach states that the process should declare the maximum number of resources of each type it may ever need.
- The Deadlock avoidance algorithm examines the resource allocations so that there can never be a circular wait condition.

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
•That is:
•If Pi resource needs are not immediately available, then Pi can wait until all Pj have finished
•When Pj is finished, Pi can obtain needed resources, execute, return allocated resources, and terminate
•When Pi terminates, Pi +1 can obtain its needed resources, and so on



Facts

- 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.

AVOIDANCE ALGORITHM

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

Banker's Algorithm in Operating System

- The banker's algorithm is a resource allocation and deadlock avoidance algorithm that tests for safety by simulating the allocation for predetermined maximum possible amounts of all resources, then makes an "s-state" check to test for possible activities, before deciding whether allocation should be allowed to continue.
- Following **Data structures** are used to implement the Banker's Algorithm:
- Let **'n'** be the number of processes in the system and **'m'** be the number of resources types.
- **Available :**
  - It is a 1-d array of size **'m'** indicating the number of available resources of each type.
  - Available[j] = k means there are **'k'** instances of resource type $\mathbf{R_j}$
- **Max :**
  - It is a 2-d array of size '**n*m'** that defines the maximum demand of each process in a system.
  - Max[i, j] = k means process $\mathbf{P_i}$ may request at most **'k'** instances of resource type $\mathbf{R_{j.}}$

- **Allocation :**
- It is a 2-d array of size **'n*m'** that defines the number of resources of each type currently allocated to each process.
- Allocation[i, j] = k means process $\mathbf{P_i}$ is currently allocated **'k'** instances of resource type $\mathbf{R_j}$
- **Need :**
- It is a 2-d array of size **'n*m'** that indicates the remaining resource need of each process.
- Need [i, j] = k means process $\mathbf{P_i}$ currently need **'k'** instances of resource type $\mathbf{R_j}$ for its execution.
- **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=1, 2, 3, 4….n
- *2) Find an i such that both*
  a) Finish[i] = false
  b) $Need_i$ <= Work
  if no such i exists goto step (4)
- *3) Work = Work + Allocation[i]*
  Finish[i] = true
  goto step (2)
- *4) if Finish [i] = true for all i*
  then the system is in a safe state

Resource-Request Algorithm

Let $Request_i$ be the request array for process P_i.
$Request_i[j] = k$ means process P_i wants k instances of resource type R_j. When a request for resources is made by process P_i, 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, P_i must wait, since the resources are not available.
3) Have the system pretend to have allocated the requested resources to process Pi by modifying the state as follows:
Available = Available – Requesti
$Allocation_i = Allocation_i + Request_i$
$Need_i = Need_i – Request_i$

Example:

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	

Question

Process	Allocation		Max		Available	
	R_1	R_2	R_1	R_2	R_1	R_2
P_1	7	2	9	5	2	1
P_2	1	3	2	6		
P_3	1	1	2	2		
P_4	3	0	5	0		

i) Calculate content of need matrix.

ii) System is safe or unsafe.

Question

Example 6.4 : *The operating system contains 3 resources, the number of instance of each resource type are 7, 7, 10. The current resource allocation state is as shown below.*

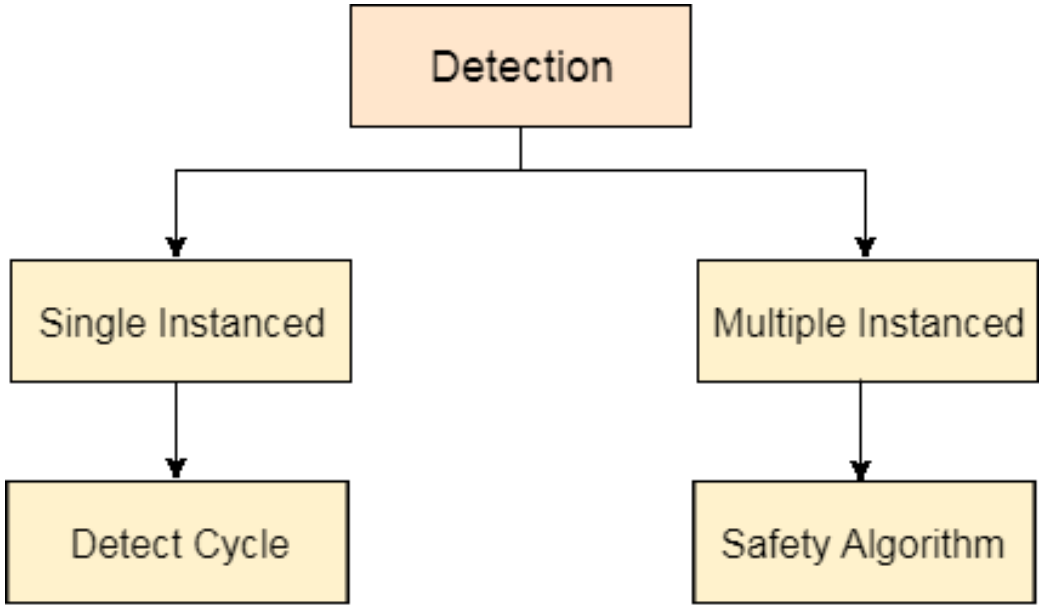
Process	Current allocation			Maximum need		
	R_1	R_2	R_3	R_1	R_2	R_3
P_1	2	2	3	3	6	8
P_2	2	0	3	4	3	3
P_3	1	2	4	3	4	4

i) Is the current allocation in a safe state?

P1(1 1 0) request granted?

Deadlock Detection and Recovery

- The main task of the OS is detecting the deadlocks. The OS can detect the deadlocks with the help of Resource allocation graph.
- In single instanced resource types, if a cycle is being formed in the system then there will definitely be a deadlock. On the other hand, in multiple instanced resource type graph, detecting a cycle is not just enough. We have to apply the safety algorithm on the system by converting the resource allocation graph into the allocation matrix and request matrix.



- In order to recover the system from deadlocks, either OS considers resources or processes.
- For Resource
- Preempt the resource
- We can snatch one of the resources from the owner of the resource (process) and give it to the other process with the expectation that it will complete the execution and will release this resource sooner. Well, choosing a resource which will be snatched is going to be a bit difficult.
- Rollback to a safe state
- System passes through various states to get into the deadlock state. The operating system can rollback the system to the previous safe state. For this purpose, OS needs to implement check pointing at every state.
- The moment, we get into deadlock, we will rollback all the allocations to get into the previous safe state.

For Process

Kill a process

Killing a process can solve our problem but the bigger concern is to decide which process to kill. Generally, Operating system kills a process which has done least amount of work until now.

Kill all process

This is not a suggestible approach but can be implemented if the problem becomes very serious. Killing all process will lead to inefficiency in the system because all the processes will execute again from starting.