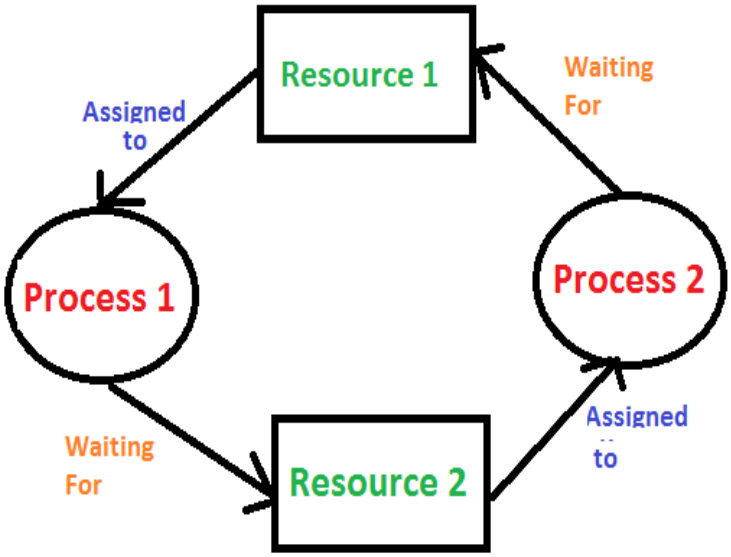
Introduction of Deadlock in Operating System

A process in operating systems uses different resources and uses resources in following way.
1) Requests a resource
2) Use the resource
2) Releases the resource

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.
Consider an example when two trains are coming toward each other on same track and there is only one track, none of the trains can move once they are in front of each other. Similar situation occurs in operating systems when there are two or more processes hold some resources and wait for resources held by other(s). 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.



Deadlock can arise if following four conditions hold simultaneously (<u>Necessary Conditions</u>)

Mutual Exclusion: One or more than one resource are non-sharable (Only one process can use at a time)
Hold and Wait: A process is holding at least one resource and waiting for resources.

No Preemption: A resource cannot be taken from a process unless the process releases the resource.
Circular Wait: A set of processes are waiting for each other in circular form.

Methods for handling deadlock
There are three ways to handle deadlock
1) Deadlock prevention or avoidance: The idea is to not let the system into deadlock state.
One can zoom into each category individually, Prevention is done by negating one of above mentioned necessary conditions for deadlock.
Avoidance is kind of futuristic in nature. By using strategy of "Avoidance", we have to make an assumption. We need to ensure that all information about resources which process WILL need are known to us prior to execution of the process. We use Banker's algorithm (Which is in-turn a gift from Dijkstra) in order to avoid deadlock.
2) Deadlock detection and recovery: Let deadlock occur, then do preemption to handle it once occurred.

3) Ignore the problem all together: If deadlock is very rare, then let it happen and reboot the system. This is the approach that both Windows and UNIX take.

Exercise:
1) Suppose n processes, P1, …. Pn share m identical resource units, which can be reserved and released one at a time. The maximum resource requirement of process Pi is Si, where Si > 0. Which one of the following is a sufficient condition for ensuring that deadlock does not occur? (GATE CS 2005)

(a) $\forall i, s_i < m$ (b) $\forall i, s_i < n$

(c) $\sum_{i=1}^{n} s_i < (m+n)$ (d) $\sum_{i=1}^{n} s_i < (m*n)$

Answer (c)
In the extreme condition, all processes acquire Si-1 resources and need 1 more resource. So following condition must be true to make sure that deadlock never occurs.
$\sum_{i=1}^{n}(Si-1)$ < m The above expression can be written as following. $\sum_{i=1}^{n} Si$ < (m + n)

2) Consider the following snapshot of a system running n processes. Process i is holding Xi instances of a resource R, 1 <= i <= n. currently, all instances of R are occupied. Further, for all i, process i has placed a request for an additional Yi instances while holding the Xi instances it already has. There are exactly two processes p and q such that Yp = Yq = 0. Which one of the following can serve as a necessary condition to guarantee that the system is not approaching a deadlock?

(a) min (Xp, Xq) < max (Yk) where k != p and k != q

(b) Xp + Xq >= min (Yk) where k != p and k != q

(c) max (Xp, Xq) > 1

(d) min (Xp, Xq) > 1

Answer(b)

Since both p and q don't need additional resources, they both can finish and release Xp + Xq resources without asking for any additional resource. If the resources released by p and q are sufficient for another process waiting for Yk resources, then system is not approaching deadlock.

3) Which of the following is NOT true of deadlock prevention and deadlock avoidance schemes?

(a)In deadlock prevention, the request for resources is always granted if the

resulting state is safe

(b)In deadlock avoidance, the request for resources is always granted if the result

state is safe

(c)Deadlock avoidance is less restrictive than deadlock prevention

(d)Deadlock avoidance requires knowledge of resource requirements a priori

ANSWER(a)

Recovery from Deadlock in Operating System

Prerequisite – Deadlock Detection And Recovery
When a Deadlock Detection Algorithm determines that a deadlock has occurred in the system, the system must recover from that deadlock. There are two approaches of breaking a Deadlock:

1. Process Termination:

To eliminate the deadlock, we can simply kill one or more processes. For this, we use two methods:

- **(a). Abort all the Deadlocked Processes:**
  Aborting all the processes will certainly break the deadlock, but with a great expenses. The deadlocked processes may have computed for a long time and the result of those partial computations must be discarded and there is a probability to recalculate them later.

- **(b). Abort one process at a time untill deadlock is eliminated:**
  Abort one deadlocked process at a time, untill deadlock cycle is eliminated from the system. Due to this method, there may be considerable overhead, because after aborting each process, we have to run deadlock detection algorithm to check whether any processes are still deadlocked.

2. Resource Preemption:
To eliminate deadlocks using resource preemption, we preepmt some resources from processes and give those resources to other processes. This method will raise three issues –

- **(a). Selecting a victim:**
  We must determine which resources and which processes are to be preempted and also the order to minimize the cost.
- **(b). Rollback:**
  We must determine what should be done with the process from which resources are preempted. One simple idea is total rollback. That means abort the process and restart it.
- **(c). Starvation:**
  In a system, it may happen that same process is always picked as a victim. As a result, that process will never complete its designated task. This situation is called **Starvation** and must be avoided. One solution is that a process must be picked as a victim only a finite number of times.

Deadlock Detection Algorithm in Operating System

If a system does not employ either a deadlock prevention or deadlock avoidance algorithm then a deadlock situation may occur. In this case-

- Apply an algorithm to examine state of system to determine whether deadlock has has occurred or not.
- Apply an algorithm to recover from the deadlock. For more refer- Deadlock Recovery

Deadlock Detection Algorithm/ Bankers Algorithm:
The algorithm employs several time varying data structures:

- **Available-** A vector of length m indicates the number of available resources of each type.
- **Allocation-** An n*m matrix defines the number of resources of each type currently allocated to a process. Column represents resource and resource represent process.
- **Request-** An n*m matrix indicates the current request of each process. If request[i][j] equals k then process P_i is requesting k more instances of resource type R_j.

We treat rows in the matrices Allocation and Request as vectors, we refer them as $Allocation_i$ and $Request_i$.

Steps of Algorithm:

1. Let *Work* and *Finish* be vectors of length m and n respectively. Initialize *Work= Available*. For *i=0, 1, …., n-1*, if $Request_i$ = 0, then *Finish[i]* = true; otherwise, *Finish[i]*= false.
2. Find an index i such that both
   a) *Finish[i] == false*
   b) $Request_i$ <= *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, 0<=i<n, then the system is in a deadlocked state. Moreover, if *Finish[i]==false* the process P_i is deadlocked.

For example,

	Allocation			Request			Available		
	A	B	C	A	B	C	A	B	C
P0	0	1	0	0	0	0	0	0	0
P1	2	0	0	2	0	2			
P2	3	0	3	0	0	0			
P3	2	1	1	1	0	0			
P4	0	0	2	0	0	2			

1. In this, Work = [0, 0, 0] &
   Finish = [false, false, false, false, false]
2. *i=0* is selected as both Finish[0] = false and [0, 0, 0]<=[0, 0, 0].
3. Work =[0, 0, 0]+[0, 1, 0] =>[0, 1, 0] &
   Finish = [true, false, false, false, false].
4. *i=2* is selected as both Finish[2] = false and [0, 0, 0]<=[0, 1, 0].
5. Work =[0, 1, 0]+[3, 0, 3] =>[3, 1, 3] &
   Finish = [true, false, true, false, false].
6. *i=1* is selected as both Finish[1] = false and [2, 0, 2]<=[3, 1, 3].
7. Work =[3, 1, 3]+[2, 0, 2] =>[5, 1, 5] &
   Finish = [true, true, true, false, false].
8. *i=3* is selected as both Finish[3] = false and [1, 0, 0]<=[5, 1, 5].
9. Work =[5, 1, 5]+[2, 1, 1] =>[7, 2, 6] &
   Finish = [true, true, true, true, false].
10. *i=4* is selected as both Finish[4] = false and [0, 0, 2]<=[7, 2, 6].
11. Work =[7, 2, 6]+[0, 0, 2] =>[7, 2, 8] &
    Finish = [true, true, true, true, true].
12. Since Finish is a vector of all true it means **there is no deadlock** in this example.

```
// Banker's Algorithm
#include <iostream>
using namespace std;

int main()
{
    // P0, P1, P2, P3, P4 are the Process names here

    int n, m, i, j, k;
    n = 5; // Number of processes
    m = 3; // Number of resources
    int alloc[5][3] = { { 0, 1, 0 }, // P0 // Allocation Matrix
                        { 2, 0, 0 }, // P1
                        { 3, 0, 2 }, // P2
                        { 2, 1, 1 }, // P3
                        { 0, 0, 2 } }; // P4

    int max[5][3] = { { 7, 5, 3 }, // P0 // MAX Matrix
                      { 3, 2, 2 }, // P1
                      { 9, 0, 2 }, // P2
                      { 2, 2, 2 }, // P3
                      { 4, 3, 3 } }; // P4

    int avail[3] = { 3, 3, 2 }; // Available Resources
```

```
    int f[n], ans[n], ind = 0;
    for (k = 0; k < n; k++) {
        f[k] = 0;
    }
    int need[n][m];
    for (i = 0; i < n; i++) {
        for (j = 0; j < m; j++)
            need[i][j] = max[i][j] - alloc[i][j];
    }
    int y = 0;
    for (k = 0; k < 5; k++) {
        for (i = 0; i < n; i++) {
            if (f[i] == 0) {

                int flag = 0;
                for (j = 0; j < m; j++) {
                    if (need[i][j] > avail[j]){
                        flag = 1;
                        break;
                    }
                }

                if (flag == 0) {
                    ans[ind++] = i;
                    for (y = 0; y < m; y++)
                        avail[y] += alloc[i][y];
                    f[i] = 1;
                }
            }
        }
    }

    cout << "Following is the SAFE Sequence" << endl;
    for (i = 0; i < n - 1; i++)
        cout << " P" << ans[i] << " ->";
    cout << " P" << ans[n - 1] <<endl;

    return (0);
}
```

```
Following is the SAFE Sequence
 P1 -> P3 -> P4 -> P0 -> P2
```