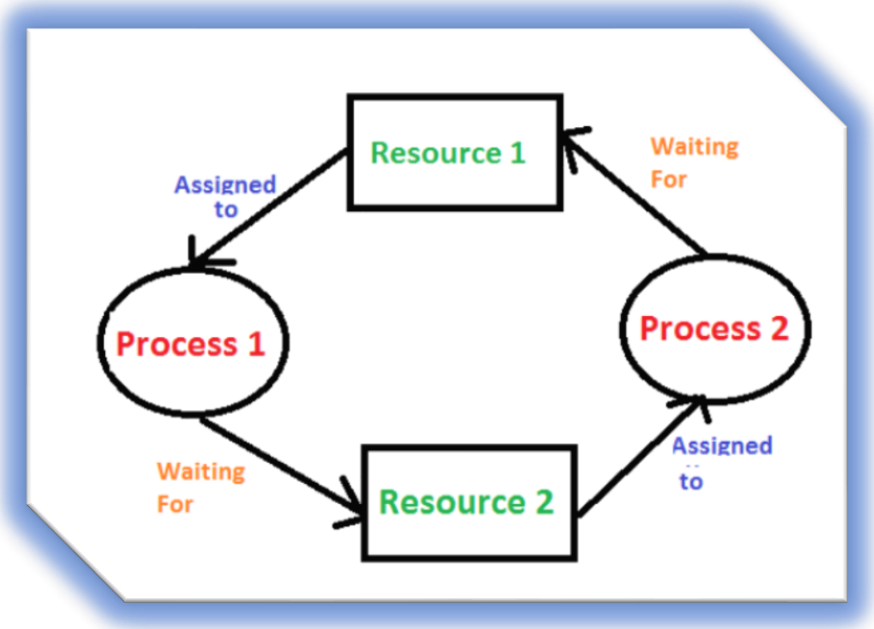
DEADLOCK IN OS

Deadlock

A **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 the **same track** and there is only **one track**, none of the trains can move once they are in front of each other.
- ✓ A similar situation occurs in operating systems when there are two or more processes that 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.**



Examples Of Deadlock

The system has **2 tape** drives. **P1** and **P2** each hold one tape drive and each needs another one.

Semaphores A and **B**, initialized to **1, P0**, and **P1** are in deadlock as follows:

P0 executes **wait(A)** and preempts.

P1 executes **wait(B).**

Now **P0** and **P1** enter in deadlock.

P0	P1
wait(A);	wait(B)
wait(B);	wait(A)

Deadlock can **arise** if the following **four conditions** hold simultaneously (Necessary Conditions)

- **Mutual Exclusion:** Two or more resources are non-shareable (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 waiting for each other in circular form.

Methods of Handling Deadlock:

Deadlock Detection

Deadlock detection in OS is a critical concept in computer science and operating systems that deals with the potential problem of deadlocks in concurrent systems.

A deadlock occurs when two or more processes are unable to proceed with their execution because each process is waiting for a resource that is held by another process within the same system.

This situation results in a standstill where no process can make progress, effectively halting the system's functionality.

Causes of Deadlocks:

Mutual Exclusion:

At least one resource must be held in a non-sharable mode. This means that only one process can use the resource at a time.

Hold and Wait:

A process must be holding at least one resource and waiting to acquire additional resources at the same time.

No Preemption:

Resources cannot be forcibly taken away from a process; they can only be released voluntarily.

Circular Wait:

A circular chain of two or more processes exists, where each process is waiting for a resource held by the next process in the chain.

Deadlock Detection in OS Strategies:

Resource Allocation Graph:

- ✓ This method represents the resources and processes as nodes in a graph, with edges representing resource requests and allocations.
- ✓ Deadlocks can be detected by identifying cycles in the graph.

Wait-Die and Wound-Wait Schemes:

- ✓ These schemes are used in resource management systems where older processes can preempt resources from younger processes (Wound-Wait) or vice versa (Wait-Die).
- ✓ These strategies help to prevent potential deadlocks.

Banker's Algorithm:

- ✓ This algorithm is used to determine if a resource allocation will lead to a safe state, where a safe state ensures that all processes can complete their execution without getting stuck in a deadlock.

Timeout Mechanisms:

- ✓ Processes are given a certain time limit to complete their execution.
- ✓ If they fail to complete within the time limit, their allocated resources are released, preventing potential deadlocks.

Periodic Checking:

- ✓ In this approach, the system periodically checks for potential deadlocks by analyzing the state of resources and processes.
- ✓ If a deadlock is detected, appropriate actions are taken.

Advantages of Deadlock Detection in OS:

Prevents Hangs:

- Detects and resolves deadlocks, preventing system freezes.

Optimizes Allocation:

- Identifies inefficient resource usage and reallocates for efficiency.

Dynamic Allocation:

- Responds to real-time demands, improving responsiveness.

Minimizes Impact:

- Reduces disruptions, enhancing user experience.

Predictable Behavior:

- Enables better management and predictability.

Monitors Health:

- Provides insights, aids performance optimization.

Reclaims Resources:

- Frees locked resources, increasing availability.

Limitations of Deadlock Detection:

Detection Overhead:

- Adds computational burden due to continuous monitoring.

Delayed Resolution:

- Only responds after deadlock occurs, not preventing it.

Resource Usage:

- Consumes additional system resources for monitoring.

Complexity:

- Requires careful design and tuning, adding complexity.

Potential False Positives:

- Can detect non-deadlock situations as deadlocks.

Deadlock Recovery

- ✓ Deadlock recovery is a process in computer science and operating systems that aims to resolve or mitigate the effects of a deadlock after it has been detected.
- ✓ Deadlocks are situations where multiple processes are stuck and unable to proceed because each process is waiting for a resource held by another process.
- ✓ Recovery strategies are designed to break this deadlock and allow the system to continue functioning.

Recovery Strategies:

Process Termination:

One way to recover from a deadlock is to terminate one or more of the processes involved in the deadlock.

By releasing the resources held by these terminated processes, the remaining processes may be able to continue executing.

However, this approach should be used cautiously, as terminating processes could lead to loss of data or incomplete transactions.

Resource Preemption:

- ✓ Resources can be forcibly taken away from one or more processes and allocated to the waiting processes.
- ✓ This approach can break the circular wait condition and allow the system to proceed. However, resource preemption can be complex and needs careful consideration to avoid disrupting the execution of processes.

Process Rollback:

- ✓ In situations where processes have checkpoints or states saved at various intervals, a process can be rolled back to a previously saved state.
- ✓ This means that the process will release all the resources acquired after the saved state, which can then be allocated to other waiting processes.

- ✓ Rollback, though, can be resource-intensive and may not be feasible for all types of applications.

Wait-Die and Wound-Wait Schemes:

- ✓ As mentioned in the Deadlock Detection in OS section, these schemes can also be used for recovery.
- ✓ Older processes can preempt resources from younger processes (**Wound-Wait),** or younger processes can be terminated if they try to access resources held by older processes (Wait-Die).

Kill the Deadlock:

- ✓ In some cases, it might be possible to identify a specific process that is causing the deadlock and terminate it.
- ✓ This is typically a last resort option, as it directly terminates a process without any complex recovery mechanisms.

Advantages of Deadlock Recovery:

Resumes Processes:

- ✓ Terminates deadlock-involved processes, allowing others to continue.

Reclaims Resources:

- ✓ Releases resources, improving allocation.

Minimizes Downtime:

- ✓ Swiftly resolves deadlocks, reducing system downtime.

User Transparent:

- ✓ Shields users from deadlock complexities.

Ensures Stability:

- ✓ Prevents prolonged hangs, enhances system stability.

Optimizes Utilization:

Redistributes resources for efficient use.

Automated Resolution:

Swift, automated recovery from deadlocks.

Limitations of Deadlock Recovery:

Process Termination:

- ✓ May terminate processes, affecting user tasks.

Resource Waste:

- ✓ Terminated processes release resources, causing waste.

User Impact:

- ✓ Interruption to users due to process termination.

Unfairness:

- ✓ Selecting processes to terminate may seem arbitrary.

Automated Risks:

- ✓ Automated recovery decisions might not be optimal.

Banker ‘s Algorithm:

- ✓ **Banker's algorithm** is a deadlock avoidance algorithm.
- ✓ It is named so because this algorithm is used in banking systems to determine whether a loan can be granted or not.
- ✓ Consider there are **n** account holders in a bank and the sum of the money in all of their accounts is **S.**
- ✓ Every time a loan has to be granted by the bank, it subtracts the loan amount from the total money the bank has.
- ✓ Then it checks if that difference is greater than **S.**
- ✓ It is done because, only then, the bank would have enough money even if all the n account holders draw all their money at once.
- ✓ Banker's algorithm works in a similar way in computers.
- ✓ Whenever a new process is created, it must specify the maximum instances of each resource type that it needs, exactly.

Characteristics of Banker's Algorithm

The characteristics of Banker's algorithm are as follows:

- If any process requests for a resource, then it has to wait.
- This algorithm consists of advanced features for maximum resource allocation.
- There are limited resources in the system we have.
- In this algorithm, if any process gets all the needed resources, then it is that it should return the resources in a restricted period.
- Various resources are maintained in this algorithm that can fulfill the needs of at least one client.

Let us assume that there are **n** processes and **m** resource types.

Data Structures used to implement the Banker's Algorithm

Some data structures that are used to implement the banker's algorithm are:

Available

- It is an array of length **m.** It represents the number of available resources of each type. If Available**[j] = k,** then there are k instances available, of resource type **Rj.**

Max

- It is an **n x m** matrix which represents the maximum number of instances of each resource that a process can request. If **Max[i][j] = k**, then the process **Pi** can request atmost **k** instances of resource type **Rj.**

Allocation

- It is an **n x m** matrix which represents the number of resources of each type currently allocated to each process. If Allocation**[i][j] = k,** then process Pi is currently allocated k instances of resource type **Rj.**

Need

- It is a two-dimensional array. It is an **n x m** matrix which indicates the remaining resource needs of each process. If **Need[i][j] = k,** then process **Pi** may need **k** more instances of resource type **Rj** to complete its task.

```
Need[i][j] = Max[i][j] - Allocation [i][j]
```

Example:

Let us consider the following snapshot for understanding the **banker's algorithm:**

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

- ❖ calculate the content of the need matrix?
- ❖ Check if the system is in a safe state?
- ❖ Determine the total sum of each type of resource?

Solution:

1. The Content of the need matrix can be calculated by using the formula given below:

Need = Max – Allocation

Process	Need		
	A	B	C
P_0	3	2	1
P_1	1	1	0
P_2	5	0	1
P_3	7	3	3
P_4	0	0	0

2. Let us now check for the safe state.

Safe sequence:

For process P0, Need = (3, 2, 1) and

Available = (2, 1, 0)

Need <=Available = False

So, the system will move to the next process.

2. For Process P1, Need = (1, 1, 0)

Available = (2, 1, 0)

Need <= Available = True

Request of P1 is granted.

Available = Available +Allocation

= (2, 1, 0) + (2, 1, 2)

= (4, 2, 2) (New Available)

3. For Process P2, Need = (5, 0, 1)

Available = (4, 2, 2)

Need <=Available = False

So, the system will move to the next process.

4. For Process P3, Need = (7, 3, 3)

Available = (4, 2, 2)

Need <=Available = False

So, the system will move to the next process.

5. For Process P4, Need = (0, 0, 0)

Available = (4, 2, 2)

Need <= Available = True

Request of P4 is granted.

Available = Available + Allocatio

= (4, 2, 2) + (1, 1, 2)

= (5, 3, 4) now, (New Available)

6. Now again check for Process P2, Need = (5, 0, 1)

Available = (5, 3, 4)

Need <= Available = True

Request of P2 is granted.

Available = Available + Allocation

= (5, 3, 4) + (4, 0, 1)

= (9, 3, 5) now, (New Available)

7. Now again check for Process P3, Need = (7, 3, 3)

Available = (9, 3, 5)

Need <=Available = True

The request for P3 is granted.

Available = Available +Allocation

= (9, 3, 5) + (0, 2, 0) = (9, 5, 5)

8. Now again check for Process P0, = Need (3, 2, 1)

= Available (9, 5, 5)

Need <= Available = True

So, the request will be granted to P0.

Safe sequence: < P1, P4, P2, P3, P0>

The system allocates all the needed resources to each process. So, we can say that the system is in a safe state.

9.The total amount of resources will be calculated by the following formula:

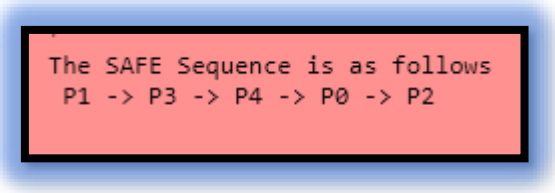
The total amount of resources= sum of columns of allocation + Available

= [8 5 7] + [2 1 0] = **[10 6 7]**

Implementation of Banker's Algorithm in C:

```c
//C program for Banker's Algorithm
#include <stdio.h>
int main()
{
    // P0, P1, P2, P3, P4 are the names of Process

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

    int max[5][3] = { { 7, 6, 3 }, // P0 // MAX Matrix
                      { 3, 2, 2 }, // P1
                      { 8, 0, 2 }, // P2
                      { 2, 1, 2 }, // P3
                      { 5, 2, 3 } }; // P4

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

    int f[n], ans[n], ind = 0;
    for (k = 0; k < n; k++) {
        f[k] = 0;
    }
    int need[n][r];
    for (i = 0; i < n; i++) {
        for (j = 0; j < r; 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 < r; j++) {
                    if (need[i][j] > avail[j]){
                        flag = 1;
                        break;
                    }
                }

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

    printf("Th SAFE Sequence is as follows\n");
    for (i = 0; i < n - 1; i++)
        printf(" P%d ->", ans[i]);
    printf(" P%d", ans[n - 1]);

    return (0);

}
```

Output:



Difference between Deadlock and Starvation :

Sr.	Deadlock	Starvation
1	Deadlock is a situation where no process got blocked and no process proceeds	Starvation is a situation where the low priority process got blocked and the high priority processes proceed.
2	Deadlock is an infinite waiting.	Starvation is a long waiting but not infinite.
3	Every Deadlock is always a starvation.	Every starvation need not be deadlock.
4	The requested resource is blocked by the other process.	The requested resource is continuously be used by the higher priority processes.
5	Deadlock happens when Mutual exclusion, hold and wait, No preemption and circular wait occurs simultaneously.	It occurs due to the uncontrolled priority and resource management.