UNIT - III

DEADLOCK

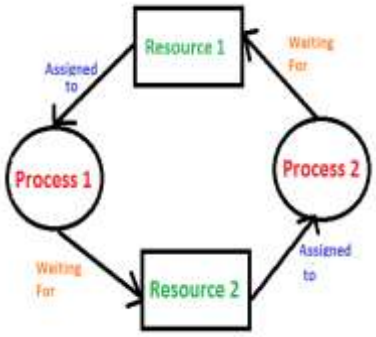
A process in operating system uses resources in the following way.

(i) Requests a resource
(ii) Use the resource
(iii) Releases the resource

*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, Process1 is holding Resource1 and waiting for Rsource2 which is acquired by Process2, and Process2 is waiting for Resource1.



Examples of Deadlock

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

3. Assume the space is available for allocation of 200K bytes, and the following sequence of events occurs.

P0	P1
Request 80KB;	Request 70KB;
Request 60KB;	Request 80KB;

NECESSARY CONDITIONS FOR DEADLOCK

- ***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 Pre-emption***

  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.

Resource Allocation Graph

The resource allocation graph is the pictorial representation of the state of a system. As its name suggests, the resource allocation graph is the complete information about all the processes which are holding some resources or waiting for some resources.

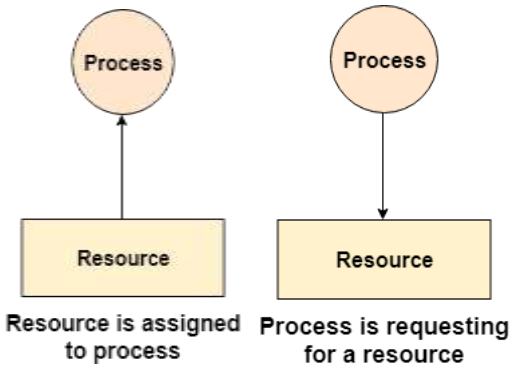
It also contains the information about all the instances of all the resources whether they are available or being used by the processes.

In Resource allocation graph, the process is represented by a Circle while the Resource is represented by a rectangle.

Vertices are mainly of two types, Resource and Process. Each of them will be represented by a different shape. Circle represents process while rectangle represents resource. A resource can have more than one instance. Each instance will be represented by a dot inside the rectangle.

Edges in RAG are also of two types, one represents **Assignment Edge** and other represents the wait of a process for a resource ie.**Request Edge**.

A resource is shown as assigned to a process if the tail of the arrow is attached to an instance to the resource and the head is attached to a process.



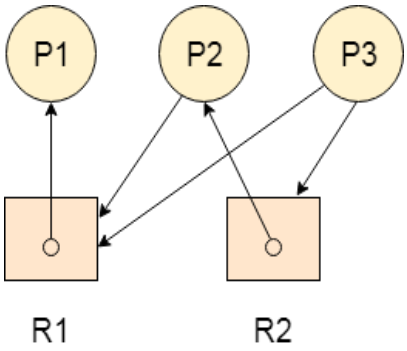
A process is shown as waiting for a resource if the tail of an arrow is attached to the process while the head is pointing towards the resource.

Example

Consider 3 processes P1, P2 and P3 and two types of resources R1 and R2. The resources are having 1 instance each.

According to the graph, R1 is being used by P1, P2 is holding R2 and waiting for R1, P3 is waiting for R1 as well as R2.

The graph is deadlock free since no cycle is being formed in the graph.



Using Resource Allocation Graph, it can be easily detected whether system is in a Deadlock state or not. The rules are

Rule-01: In a Resource Allocation Graph where all the resources are single instance,

- If a cycle is being formed, then system is in a deadlock state.
- If no cycle is being formed, then system is not in a deadlock state.

Rule-02: In a Resource Allocation Graph where all the resources are **NOT** single instance,

- If a cycle is being formed, then system may be in a deadlock state.
- **Banker's Algorithm** is applied to confirm whether system is in a deadlock state or not.
- If no cycle is being formed, then system is not in a deadlock state.
- Presence of a cycle is a necessary but not a sufficient condition for the occurrence of deadlock.

METHODS FOR HANDLING DEADLOCK

There are three ways to handle deadlock

1) Deadlock prevention or avoidance

PREVENTION

The idea is to not let the system into a deadlock state. This system will make sure that above mentioned four conditions will not arise. These techniques are very costly so we use this in cases where our priority is making a system deadlock-free.

One can zoom into each category individually, Prevention is done by negating one of the four necessary conditions for deadlock.

Eliminate mutual exclusion

It is not possible to dis-satisfy the mutual exclusion because some resources, such as the tape drive and printer, are inherently non-shareable.

Solve hold and Wait

Allocate all required resources to the process before the start of its execution, this way hold and wait condition is eliminated but it will lead to low device utilization. for example, if a process requires a printer at a later time and we have allocated a printer before the start of its execution printer will remain blocked till it has completed its execution. The process will make a new request for resources after releasing the current set of resources. This solution may lead to starvation.

Allow pre-emption

Preempt resources from the process when resources are required by other high-priority processes.

Circular wait Solution

Each resource will be assigned a numerical number. A process can request the resources to increase/decrease. order of numbering. For Example, if the P1 process is allocated R5 resources, now next time if P1 asks for R4, R3 lesser than R5 such a request will not be granted, only a request for resources more than R5 will be granted.

AVOIDANCE

Avoidance is kind of futuristic. By using the strategy of “Avoidance”, we have to make an assumption. We need to ensure that all information about resources that the process will need is known to us before the execution of the process.

Resource Allocation Graph

The resource allocation graph (RAG) is used to visualize the system's current state as a graph. The Graph includes all processes, the resources that are assigned to them, as well as the resources that each Process requests. Sometimes, if there are fewer processes, we can quickly spot a deadlock in the system by looking at the graph rather than the tables we use in Banker's algorithm.

Banker's Algorithm

Bankers's Algorithm is a resource allocation and deadlock avoidance algorithm which test all the request made by processes for resources, it checks for the safe state, and after granting a request system remains in the safe state it allows the request, and if there is no safe state it doesn't allow the request made by the process.

In prevention and avoidance, we get the correctness of data but performance decreases.

2) Deadlock detection and recovery

If Deadlock prevention or avoidance is not applied to the software then we can handle this by deadlock detection and recovery, which consist of two phases.

In the first phase, we examine the state of the process and check whether there is a deadlock or not in the system.

If found deadlock in the first phase then we apply the algorithm for recovery of the deadlock.

3) Deadlock ignorance:

If a deadlock is very rare, then let it happen and reboot the system. This is the approach that both Windows and UNIX take. We use the ostrich algorithm for deadlock ignorance.

In Deadlock, ignorance performance is better than the above two methods but not the correctness of data.

SAFE STATE

A safe state can be defined as a state in which there is no deadlock. It is achievable if:

- If a process needs an unavailable resource, it may wait until the same has been released by a process to which it has already been allocated. if such a sequence does not exist, it is an unsafe state.
- All the requested resources are allocated to the process.

BANKER'S ALGORITHM

It is a banker algorithm used to **avoid deadlock** and **allocate resources** safely to each process in the computer system. The '**S-State'** examines all possible tests or activities before deciding whether the allocation should be allowed to each process. It also helps the operating system to successfully share the resources between all the processes.

The banker's algorithm is named because it checks whether a person should be sanctioned a loan amount or not to help the bank system safely simulate allocation resources.

Suppose the number of account holders in a particular bank is 'n', and the total money in a bank is 'T'. If an account holder applies for a loan; first, the bank subtracts the loan amount from full cash and then estimates the cash difference is greater than T to approve the loan amount. These steps are taken because if another person applies for a loan or withdraws some amount from the bank, it helps the bank manage and operate all things without any restriction in the functionality of the banking system.

Similarly, it works in an **operating system**. When a new process is created in a computer system, the process must provide all types of information to the operating system like upcoming processes, requests for their resources, counting them, and delays.

Based on these criteria, the operating system decides which process sequence should be executed or waited so that no deadlock occurs in a system. Therefore, it is also known as **deadlock avoidance algorithm** or **deadlock detection** in the operating system.

When working with a banker's algorithm, it requests to know about three things:

1. How much each process can request for each resource in the system. It is denoted by the [**MAX**] request.
2. How much each process is currently holding each resource in a system. It is denoted by the [**ALLOCATED**] resource.
3. It represents the number of each resource currently available in the system. It is denoted by the [**AVAILABLE**] resource.

Following are the important data structures terms applied in the banker's algorithm as follows:

Suppose n is the number of processes, and m is the number of each type of resource used in a computer system.

1. **Available**: It is an array of length 'm' that defines each type of resource available in the system. When Available[j] = K, means that 'K' instances of Resources type R[j] are available in the system.
2. **Max:** It is a [n x m] matrix that indicates each process P[i] can store the maximum number of resources R[j] (each type) in a system.
3. **Allocation:** It is a matrix of m x n orders that indicates the type of resources currently allocated to each process in the system. When Allocation [i, j] = K, it means that process P[i] is currently allocated K instances of Resources type R[j] in the system.
4. **Need:** It is an M x N matrix sequence representing the number of remaining resources for each process. When the Need[i] [j] = k, then process P[i] may require K more instances of resources type Rj to complete the assigned work.

   Need[i][j] = Max[i][j] - Allocation[i][j].
5. **Finish**: It is the vector of the order **m**. It includes a Boolean value (true/false) indicating whether the process has been allocated to the requested resources, and all resources have been released after finishing its task.

The Banker's Algorithm is the combination of the safety algorithm and the resource request algorithm to control the processes and avoid deadlock.

Safety Algorithm

It is a safety algorithm used to check whether or not a system is in a safe state or follows the safe sequence in a banker's algorithm:

Step1:

There are two vectors **Wok** and **Finish** of length m and n in a safety algorithm.

Initialize: Work = Available

Finish[i] = false; for I = 0, 1, 2, 3, 4… n - 1.

Step2:

Check the availability status for each type of resources [i], such as:

Need[i] <= Work

Finish[i] == false

If the i does not exist, go to step 4.

Step3:

Work = Work +Allocation(i) // to get new resource allocation

Finish[i] = true

Go to step2 to check the status of resource availability for the next process.

Step4:

If Finish[i] == true; it means that the system is safe for all processes.

Resource Request Algorithm

Let create a resource request array R[i] for each process P[i].

Step1:

When the number of **requested resources** of each type is less than the **Need** resources, go to step2 and if the condition fails, which means that the process P[i] exceeds its maximum claim for the resource. As the expression suggests:

If Request(i) <= Need, then go to step2, Else raise an error message.

Step2:

And when the number of requested resources of each type is less than the available resource for each process, go to step (3). As the expression suggests:

If Request(i) <= Available, then go to step3.

Else Process P[i] must wait for the resource.

Step3:

When the requested resource is allocated to the process by changing state:

Available = Available – Request

Allocation(i) = Allocation(i) + Request (i)

$Need_i = Need_i - Request_i$

When the resource allocation state is safe, its resources are allocated to the process P(i). And if the new state is unsafe, the Process P (i) has to wait for each type of Request R(i) and restore the old resource-allocation state.

Example:

Consider a system that contains five processes P1, P2, P3, P4, P5 and the three resource types A, B and C. Following are the resources types: A has 10, B has 5 and the resource type C has 7 instances.

Process	Allocation			Max			Available		
	A	B	C	A	B	C	A	B	C
P1	0	1	0	7	5	3	3	3	2
P2	2	0	0	3	2	2			
P3	3	0	2	9	0	2			
P4	2	1	1	2	2	2			
P5	0	0	2	4	3	3			

Answer the following questions using the banker's algorithm:

1. What is the reference of the need matrix?
2. Determine if the system is safe or not.
3. What will happen if the resource request (1, 0, 2) for process P1 can the system accept this request immediately?
4. What will happen if the resource request (3, 3, 0) for process P5?
5. What will happen if the resource request (0, 2, 0) for process P1?

Ans.1:

Context of the need matrix is as Need [i] = Max [i] - Allocation [i]

Need for P1: (7, 5, 3) - (0, 1, 0) = 7, 4, 3
Need for P2: (3, 2, 2) - (2, 0, 0) = 1, 2, 2
Need for P3: (9, 0, 2) - (3, 0, 2) = 6, 0, 0
Need for P4: (2, 2, 2) - (2, 1, 1) = 0, 1, 1
Need for P5: (4, 3, 3) - (0, 0, 2) = 4, 3, 1

Process	Need		
	A	B	C
P1	7	4	3
P2	1	2	2
P3	6	0	0
P4	0	1	1
P5	4	3	1

Ans.2: Apply the Banker's Algorithm:

Available Resources of A, B and C are 3, 3, and 2.

Now we check if each type of resource request is available for each process.

Step 1:

For Process P1:

Need <= Available

7, 4, 3 <= 3, 3, 2 condition is **false**.

So, we examine another process, P2.

Step 2:

For Process P2:

Need <= Available

1, 2, 2 <= 3, 3, 2 condition **true**

New available = available + Allocation

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

Similarly, we examine another process P3.

Step 3:

For Process P3:

P3 Need <= Available

6, 0, 0 < = 5, 3, 2 condition is **false**.

Similarly, we examine another process, P4.

Step 4:

For Process P4:

P4 Need <= Available

0, 1, 1 <= 5, 3, 2 condition is **true**

New Available resource = Available + Allocation

5, 3, 2 + 2, 1, 1 => 7, 4, 3

Similarly, we examine another process P5.

Step 5:

For Process P5:

P5 Need <= Available

4, 3, 1 <= 7, 4, 3 condition is **true**

New available resource = Available + Allocation

7, 4, 3 + 0, 0, 2 => 7, 4, 5

Now, we again examine each type of resource request for processes P1 and P3.

Step 6:

For Process P1:

P1 Need <= Available

7, 4, 3 <= 7, 4, 5 condition is **true**

New Available Resource = Available + Allocation

7, 4, 5 + 0, 1, 0 => 7, 5, 5

So, we examine another process P2.

Step 7:

For Process P3:

P3 Need <= Available

6, 0, 0 <= 7, 5, 5 condition is true

New Available Resource = Available + Allocation

7, 5, 5 + 3, 0, 2 => 10, 5, 7

Hence, we execute the banker's algorithm to find the safe state and the safe sequence like P2, P4, P5, P1 and P3.

Ans. 3:

For granting the Request (1, 0, 2), first we have to check that **Request <= Available**, that is (1, 0, 2) <= (3, 3, 2),

Since the condition is true, the process P2 may get the request immediately.

Allocation for P2 is (3,0,2) and new Available is (2, 3, 0)

Context of the need matrix is as follows:

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

Need for P1: (7, 5, 3) - (0, 1, 0) = 7, 4, 3
Need for P2: (3, 2, 2) - (3, 0, 2) = 0, 2, 0
Need for P3: (9, 0, 2) - (3, 0, 2) = 6, 0, 0
Need for P4: (2, 2, 2) - (2, 1, 1) = 0, 1, 1
Need for P5: (4, 3, 3) - (0, 0, 2) = 4, 3, 1

Process	Need		
	A	B	C
P1	7	4	3
P2	0	2	0
P3	6	0	0
P4	0	1	1
P5	4	3	1

Apply the Banker's Algorithm:

Available Resources of A, B and C are 2, 3, and 0.

Now we check if each type of resource request is available for each process.

Step 1:

For Process P1:

Need <= Available

7, 4, 3 <= 2, 3, 0 condition is **false**.

So, we examine another process, P2.

Step 2:

For Process P2:

Need <= Available

1, 2, 2 <= 2, 3, 0 condition **true**

New available = available + Allocation

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

Similarly, we examine another process P3.

Step 3:

For Process P3:

P3 Need <= Available

6, 0, 0 < = 5, 3, 2 condition is **false**.

Similarly, we examine another process, P4.

Step 4:

For Process P4:

P4 Need <= Available

0, 1, 1 <= 5, 3, 2 condition is **true**

New Available resource = Available + Allocation

5, 3, 2 + 2, 1, 1 => 7, 4, 3

Similarly, we examine another process P5.

Step 5:

For Process P5:

P5 Need <= Available

4, 3, 1 <= 7, 4, 3 condition is **true**

New available resource = Available + Allocation

7, 4, 3 + 0, 0, 2 => 7, 4, 5

Now, we again examine for processes P1 and P3.

Step 6:

For Process P1:

P1 Need <= Available

7, 4, 3 <= 7, 4, 5 condition is **true**

New Available Resource = Available + Allocation

7, 4, 5 + 0, 1, 0 => 7, 5, 5

So, we examine another process P2.

Step 7:

For Process P3:

P3 Need <= Available

6, 0, 0 <= 7, 5, 5 condition is true

New Available Resource = Available + Allocation

7, 5, 5 + 3, 0, 2 => 10, 5, 7

Hence, P2 granted immediately and the safe sequence like P2, P4, P5, P1 and P3.

Ans. 4:

For granting the Request (3, 3, 0) by P5, first we have to check that

Request <= Available, that is (3, 3, 0) <= (2, 3, 0),

Since the condition is false. So the request for (3, 3, 0) by process P5 cannot be granted.

Ans. 5:

For granting the Request (0, 2, 0) by P1, first we have to check that

Request <= Available, that is (0, 2, 0) <= (2, 3, 0),

Since the condition is true. So the request for (0, 2, 0) by process P1 may be granted.

Allocation for P1 is (0, 3, 0)

Context of the need matrix is as follows:

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

Need for P1: (7, 5, 3) - (0, 3, 0) = 7, 2, 3

Process	**Need** **A** **B** **C**		
P1	7	2	3
P2	0	2	0
P3	6	0	0
P4	0	1	1
P5	4	3	1

Apply the Banker's Algorithm:

Available Resources of A, B and C are 2, 1, and 0.

For Process P1: 7, 2, 3<= 2, 1, 0 condition is **false**.

For Process P2: 0, 2, 0<= 2, 1, 0 condition is **false.**

For Process P3: 6, 0, 0<= 2, 1, 0 condition is **false**.

For Process P4: 0, 1, 1 <= 2, 1, 0 condition is **false.**

For Process P5: 4, 3, 1 <= 2, 1, 0 condition is **false.**

Hence, the state is unsafe, P1 cannot be granted immediately.

DEADLOCK DETECTION

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 the system's state to determine whether deadlock has occurred.
- Apply an algorithm to recover from the deadlock.

A deadlock detection algorithm is a technique used by an operating system to identify deadlocks in the system. This algorithm checks the status of processes and resources to determine whether any deadlock has occurred and takes appropriate actions to recover from the deadlock.

The algorithm employs several times 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. The column represents resource and rows represent a 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.

The Bankers algorithm includes a **Safety Algorithm / Deadlock Detection Algorithm.** The algorithm for finding out whether a system is in a safe state can be described as follows:

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, 0] =>[5, 1, 3] &
   Finish = [true, true, true, false, false].

8. *i=3* is selected as both Finish[3] = false and [1, 0, 0]<=[5, 1, 3].
9. Work =[5, 1, 3]+[2, 1, 1] =>[7, 2, 4] &
   Finish = [true, true, true, true, false].

10. *i=4* is selected as both Finish[4] = false and [0, 0, 2]<=[7, 2, 4].
11. Work =[7, 2, 4]+[0, 0, 2] =>[7, 2, 6] &
    Finish = [true, true, true, true, true].

12. Since Finish is a vector of all true it means **there is no deadlock** in this example.

There are several algorithms for detecting deadlocks in an operating system, including:

1. Wait-For Graph:

A graphical representation of the system's processes and resources. A directed edge is created from a process to a resource if the process is waiting for that resource. A cycle in the graph indicates a deadlock.

2. Banker's Algorithm:

A resource allocation algorithm that ensures that the system is always in a safe state, where deadlocks cannot occur.

3. Resource Allocation Graph:

A graphical representation of processes and resources, where a directed edge from a process to a resource means that the process is currently holding that resource. Deadlocks can be detected by looking for cycles in the graph.

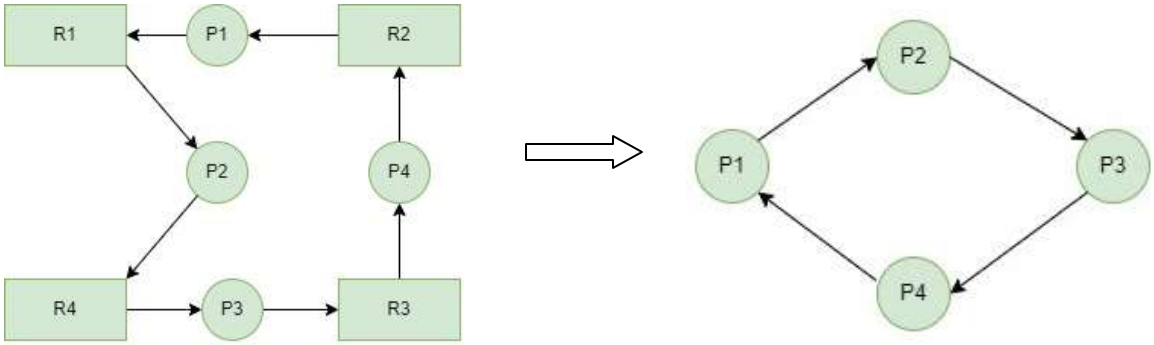
4. Detection by System Modeling:

A mathematical model of the system is created, and deadlocks can be detected by finding a state in the model where no process can continue to make progress.

5. Timestamping:

Each process is assigned a timestamp, and the system checks to see if any process is waiting for a resource that is held by a process with a lower timestamp.

These algorithms are used in different operating systems and systems with different resource allocation and synchronization requirements. The choice of algorithm depends on the specific requirements of the system and the trade-offs between performance, complexity and accuracy.



Resource Allocation Graph Wait-For graph

RECOVERY FROM DEADLOCK

The OS will use various recovery techniques to restore the system if it encounters any deadlocks. When a Deadlock Detection Algorithm determines that a deadlock has occurred in the system, the system must recover from that deadlock.

Approaches to Breaking a Deadlock

(a) Process Termination

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

1. Abort all the Deadlocked Processes:

Aborting all the processes will certainly break the deadlock but at a great expense. The deadlocked processes may have been computed for a long time, and the result of those partial computations must be discarded and there is a probability of recalculating them later.

2. Abort one process at a time until the deadlock is eliminated:

Abort one deadlocked process at a time, until the 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 a deadlock detection algorithm to check whether any processes are still deadlocked.

(b) Resource Preemption

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

1. Selecting a victim:

We must determine which resources and which processes are to be preempted and also in order to minimize the cost.

2. Rollback:

We must determine what should be done with the process from which resources are preempted. One simple idea is total rollback. That means aborting the process and restarting it.

3. Starvation:

In a system, it may happen that the 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.

PROCESS MANAGEMENT AND SYNCHRONIZATION

Process Synchronization is the coordination of execution of multiple processes in a multi-process system to ensure that they access shared resources in a controlled and predictable manner. It aims to resolve the problem of race conditions and other synchronization issues in a concurrent system.

The main objective of process synchronization is to ensure that multiple processes access shared resources without interfering with each other and to prevent the possibility of inconsistent data due to concurrent access. To achieve this, various synchronization techniques such as semaphores, monitors and critical sections are used.

On the basis of synchronization, processes are categorized as one of the following two types:

- **Independent Process**: The execution of one process does not affect the execution of other processes.
- **Cooperative Process**: A process that can affect or be affected by other processes executing in the system.

Process synchronization problem arises in the case of Cooperative processes also because resources are shared in Cooperative processes.

Race Condition

A race condition is a condition when there are many processes and every process shares the data with each other and accessing the data concurrently and the output of execution depends on a particular sequence in which they share the data and access.

(OR)

When more than one process is executing the same code or accessing the same memory or any shared variable in that condition there is a possibility that the output or the value of the shared variable is wrong so for that all the processes doing the race to say that my output is correct. This condition is known as **race condition**.

Several processes access and process the manipulations over the same data concurrently, then the outcome depends on the particular order in which the access takes place.

Example:

Let's say there are two processes P1 and P2 which share common variable (shared=10), both processes are present in ready – queue and waiting for its turn to be execute.

Suppose, Process P1 first come under execution, initialized as X=10 and increment it by 1 (ie.X=11), after then when CPU read line sleep(1), it switches from current process P1 to process P2 present in ready-queue. The process P1 goes in waiting state for 1 second.

Process 1	**Process 2**
int X = shared	int Y = shared
X++	Y--
sleep(1)	sleep(1)
shared = X	shared = Y

Now CPU execute the Process P2, initialized Y=10 and decrement Y by 1(ie.Y=9), after then when CPU read sleep(1), the current process P2 goes in waiting state and CPU remains idle for sometime as there is no process in ready-queue.

After completion of 1 second of process P1 when it comes in ready-queue, CPU takes the process P1 under execution and execute the remaining line of code and shared=11.

After completion of 1 second of Process P2, when process P2 comes in ready-queue, CPU start executing the further remaining line of Process P2 and shared=9.

Note:

We are assuming the final value of common variable(shared) after execution of Process P1 and Process P2 is 10 (as Process P1 increment variable by 1 and Process P2 decrement variable by 1 and finally it becomes shared=10). But we are getting undesired value due to lack of proper synchronization.

Actual meaning of race-condition

- If the order of execution of process (first P1 -> then P2) then we will get the value of common variable (shared) = 9.
- If the order of execution of process (first P2 -> then P1) then we will get the final value of common variable (shared) =11.

Basically, Here the (value1 = 9) and (value2=11) are racing , If we execute these two process in our computer system then sometime we will get 9 and sometime we will get 10 as final value of common variable(shared). This phenomenon is called **Race-Condition.**

CRITICAL SECTION PROBLEM

A critical section is a code segment that can be accessed by only one process at a time. The critical section contains shared variables that need to be synchronised to maintain the consistency of data variables. So the critical section problem means designing a way for cooperative processes to access shared resources without creating data inconsistencies.

In the entry section, the process requests for entry in the **Critical Section.** Any solution to the critical section problem must satisfy three requirements:

- **Mutual Exclusion**: If a process is executing in its critical section, then no other process is allowed to execute in the critical section.
- **Progress**: If no process is executing in the critical section and other processes are waiting outside the critical section, then only those processes that are not executing in their remainder section can participate in deciding which will enter in the critical section next, and the selection can't be postponed indefinitely.
- **Bounded Waiting**: A bound must exist on the number of times that other processes are allowed to enter their critical sections after a process has made a request to enter its critical section and before that request is granted.

PETERSON'S SOLUTION

Peterson's Solution is a classical software-based solution to the critical section problem. In Peterson's solution, we have two shared variables:

- **boolean flag[i]**: Initialized to FALSE, initially no one is interested in entering the critical section
- **int turn**: The process whose turn is to enter the critical section.

```
// code for producer i
do
{
 flag[i] = true;
 turn = j;
 while (flag[j] == true && turn == j);
        critical section
 flag[i] = false;
        reminder section
 }while(TRUE);
```

```
// code for consumer j
do
{
 flag[j] = true;
 turn = i;
 while (flag[i] == true && turn == i);
        critical section
 flag[i] = false;
        reminder section
 }while(TRUE);
```

In the solution, i represents the Producer and j represents the Consumer. Initially, the flags are false. When a process wants to execute it's critical section, it sets its flag to true and turn into the index of the other process. This means that the process wants to execute but it will allow the other process to run first. The process performs busy waiting until the other process has finished it's own critical section. After this, the current process enters its critical section and adds or removes a random number from the shared buffer. After completing the critical section, it sets it's own flag to false, indicating it does not wish to execute anymore.

Peterson's Solution preserves all three conditions:

- Mutual Exclusion is assured as only one process can access the critical section at any time.
- Progress is also assured, as a process outside the critical section does not block other processes from entering the critical section.
- Bounded Waiting is preserved as every process gets a fair chance.

Disadvantages of Peterson's Solution

- It involves busy waiting.
- It is limited to 2 processes.
- Peterson's solution cannot be used in modern CPU architectures.

SEMAPHORES

Semaphore is a Hardware Solution. This Hardware solution is written or given to critical section problem. The Semaphore is just a normal integer. The Semaphore cannot be negative. The least value for a Semaphore is zero (0). The Maximum value of a Semaphore can be anything. The Semaphores usually have two operations. The two operations have the capability to decide the values of the semaphores.

The two Semaphore Operations are:

1. Wait ( )
2. Signal ( )

Wait Semaphore Operation

The Wait operation works on the basis of Semaphore or Mutex Value. If the Semaphore value is greater than zero, then the Process can enter the Critical Section Area.

If the Semaphore value is equal to zero then the Process has to wait.

If the process exits the Critical Section, then have to reduce the value of Semaphore.

Definition of wait()

```
wait(Semaphore S)
{
  while (S<=0) ;        //no operation
  S--;
}
```

Signal Semaphore Operation

The most important part is that this Signal Operation or V Function is executed only when the process comes out of the critical section. The value of semaphore cannot be incremented before the exit of process from the critical section.

Definition of signal()

```
signal(S)
{
  S++;
}
```

There are two types of semaphores:

- **Binary Semaphores:**

  They can only be either 0 or 1. They are also known as mutex locks, as the locks can provide mutual exclusion. All the processes can share the same mutex semaphore that is initialized to 1. Then, a process has to wait until the lock becomes 0. Then, the process can make the mutex semaphore 1 and start its critical section. When it completes its critical section, it can reset the value of the mutex semaphore to 0 and some other process can enter its critical section.

- **Counting Semaphores:**

  They can have any value and are not restricted over a certain domain. They can be used to control access to a resource that has a limitation on the number of simultaneous accesses. The semaphore can be initialized to the number of instances of the resource. Whenever a process wants to use that resource, it checks if the number of remaining instances is more than zero, i.e., the process has an instance available. Then, the process can enter its critical section thereby decreasing the value of the counting semaphore by 1. After the process is over with the use of the instance of the resource, it can leave the critical section thereby adding 1 to the number of available instances of the resource.

CLASSICAL PROBLEMS OF SYNCHRONIZATION

The following problems of synchronization are considered as classical problems:

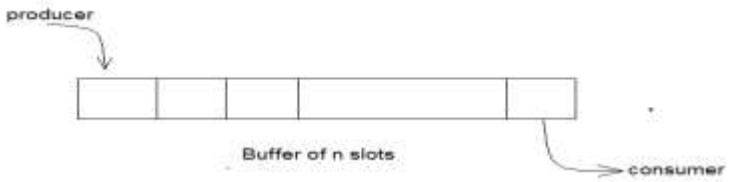
1. Bounded-buffer (or Producer-Consumer) Problem,

2. Dining-Philosophers Problem,

3. Readers and Writers Problem,

Bounded-buffer (or Producer-Consumer) Problem

Bounded Buffer problem is also called **producer consumer problem** and it is one of the classic problems of synchronization. This problem is generalized in terms of the Producer-Consumer problem. Solution to this problem is, creating two counting semaphores "full" and "empty" to keep track of the current number of full and empty buffers respectively. Producers produce a product and consumers consume the product, but both use of one of the containers each time.



A producer tries to insert data into an empty slot of the buffer. A consumer tries to remove data from a filled slot in the buffer. There needs to be a way to make the producer and consumer work in an independent manner.

One solution of this problem is to use semaphores. The semaphores which will be used here are:

- **m**, a **binary semaphore** which is used to acquire and release the lock.
- **empty**, a **counting semaphore** whose initial value is the number of slots in the buffer, since, initially all slots are empty.
- **full**, a **counting semaphore** whose initial value is 0.

At any instant, the current value of empty represents the number of empty slots in the buffer and full represents the number of occupied slots in the buffer.

The Producer Operation

```
do
{
   wait(empty);          // wait until empty > 0 and then decrement 'empty'
   wait(mutex);          // acquire lock

   /* perform the insert operation in a slot */

   signal(mutex);        // release lock
   signal(full);         // increment 'full'

 } while(TRUE);
```

- Looking at the above code for a producer, we can see that a producer first waits until there is atleast one empty slot.
- Then it decrements the **empty** semaphore because, there will now be one less empty slot, since the producer is going to insert data in one of those slots.
- Then, it acquires lock on the buffer, so that the consumer cannot access the buffer until producer completes its operation.
- After performing the insert operation, the lock is released and the value of **full** is incremented because the producer has just filled a slot in the buffer.

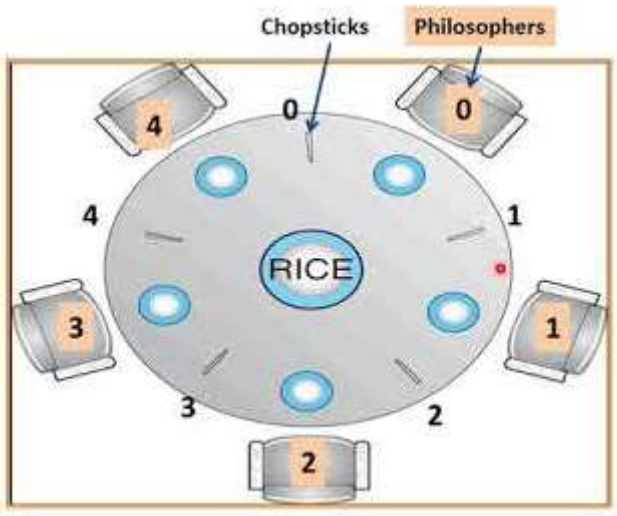
The Consumer Operation

```
do
{
   wait(full);              // wait until full > 0 and then decrement 'full'
   wait(mutex);             // acquire the lock

   /* perform the remove operation in a slot */

   signal(mutex);           // release the lock
   signal(empty);           // increment 'empty'

 } while(TRUE);
```

- The consumer waits until there is atleast one full slot in the buffer.
- Then it decrements the **full** semaphore because the number of occupied slots will be decreased by one, after the consumer completes its operation.
- After that, the consumer acquires lock on the buffer.
- Following that, the consumer completes the removal operation so that the data from one of the full slots is removed.
- Then, the consumer releases the lock.
- Finally, the **empty** semaphore is incremented by 1, because the consumer has just removed data from an occupied slot, thus making it empty.

Dining-Philosophers Problem

The Dining Philosopher Problem states that K philosophers seated around a circular table with one chopstick between each pair of philosophers. There is one chopstick between each philosopher. A philosopher may eat if he can pickup the two chopsticks adjacent to him. One chopstick may be picked up by any one of its adjacent followers but not both. This problem involves the allocation of limited resources to a group of processes in a deadlock-free and starvation-free manner.



The design of the problem was to illustrate the challenges of avoiding deadlock, a deadlock state of a system is a state in which no progress of system is possible. Consider a proposal where each philosopher is instructed to behave as follows:

- The philosopher is instructed to think till the left fork is available, when it is available, hold it.
- The philosopher is instructed to think till the right fork is available, when it is available, hold it.
- The philosopher is instructed to eat when both forks are available.
- then, put the right fork down first
- then, put the left fork down next
- repeat from the beginning.

The structure of Philosopher i is as follows.

```
do
 {
  Wait( take_chopstick[i] );
  Wait( take_chopstick[(i+1) % 5] ) ;
  …
  EAT
  …
  Signal( put_chopstick[i] );
  Signal( put_chopstick[ (i+1) % 5] ) ;
  …
  THINK
 }  while(TRUE);
```

In the above code, first wait operation is performed on take_chopstick[i] and take_chopstick[(i+1) % 5]. This shows philosopher i have picked up the chopsticks from its left and right. The eating function is performed after that.

On completion of eating by philosopher i the, signal operation is performed on take_chopstick[i] and take_chopstick[(i+1) % 5]. This shows that the philosopher i have eaten and put down both the left and right chopsticks. Finally, the philosopher starts thinking again.

Let value of i = 0(initial value), Suppose Philosopher P_0 wants to eat, it will enter in Philosopher() function, and execute **Wait(take_chopstick[i]);** by doing this it holds **C0 chopstick** and reduces semaphore C0 to 0**,** after that it execute **Wait(take_chopstick[(i+1) % 5]);** by doing this it holds **C1 chopstick** (since i =0, therefore (0 + 1) % 5 = 1) and reduces semaphore C1 to 0.

Similarly, suppose now Philosopher P1 wants to eat, it will enter in Philosopher() function, and execute **Wait(take_chopstick[i]);** by doing this it will try to hold **C1 chopstick** but will not be able to do that**,** since the value of semaphore C1 has already been set to 0 by philosopher P0, therefore it will enter into an infinite loop because of which philosopher P1 will not be able to pick chopstick C1 whereas if Philosopher P2 wants to eat, it will enter in Philosopher() function, and execute **Wait(take_chopstickC[i]);** by doing this it holds **C2 chopstick** and reduces semaphore C2 to 0, after that, it executes **Wait(take_chopstickC[(i+1) % 5]);** by doing this it holds **C3 chopstick**(since i =2, therefore (2 + 1) % 5 = 3) and reduces semaphore C3 to 0.

Hence the above code is providing a solution to the dining philosopher problem, A philosopher can only eat if both immediate left and right chopsticks of the philosopher are available else philosopher needs to wait. Also at one go two independent philosophers can eat simultaneously (i.e., philosopher **P0 and P2, P1 and P3 & P2 and P4** can eat simultaneously as all are the independent processes and they are following the above constraint of dining philosopher problem)

The drawback of the above solution of the dining philosopher problem

- No two neighbouring philosophers can eat at the same point in time.
- This solution can lead to a deadlock condition. This situation happens if all the philosophers pick their left chopstick at the same time, which leads to the condition of deadlock and none of the philosophers can eat.

To avoid deadlock, some of the solutions are as follows :

- Maximum number of philosophers on the table should not be more than four, in this case, chopstick C4 will be available for philosopher P3, so P3 will start eating and after the finish of his eating procedure, he will put down his both the chopstick C3 and C4, i.e. semaphore C3 and C4 will now be incremented to 1. Now philosopher P2 which was holding chopstick C2 will also have chopstick C3 available, hence similarly, he will put down his chopstick after eating and enable other philosophers to eat.
- A philosopher at an even position should pick the right chopstick and then the left chopstick while a philosopher at an odd position should pick the left chopstick and then the right chopstick.
- Only in case if both the chopsticks (left and right) are available at the same time, only then a philosopher should be allowed to pick their chopsticks
- All the four starting philosophers (P0, P1, P2, and P3) should pick the left chopstick and then the right chopstick, whereas the last philosopher P4 should pick the right chopstick and then the left chopstick. This will force P4 to hold his right chopstick first since the right chopstick of P4 is C0, which is already held by philosopher P0 and its value is set to 0, i.e C0 is already 0, because of which P4 will get trapped into an infinite loop and chopstick C4 remains vacant. Hence philosopher P3 has both left C3 and right C4 chopstick available, therefore it will start eating and will put down its both chopsticks once finishes and let others eat which removes the problem of deadlock.

Readers and Writers Problem

Suppose that a database is to be shared among several concurrent processes. Some of these processes may want only to read the database, whereas others may want to update (that is, to read and write) the database. We distinguish between these two types of processes by referring to the former as readers and to the latter as writers. Precisely in OS we call this situation as the readers-writers problem. Problem parameters:

- One set of data is shared among a number of processes.
- Once a writer is ready, it performs its write. Only one writer may write at a time.
- If a process is writing, no other process can read it.
- If at least one reader is reading, no other process can write.
- Readers may not write and only read.

There are four types of cases that could happen here.

Case	Process 1	Process 2	Allowed/Not Allowed
Case 1	Writing	Writing	Not Allowed
Case 2	Writing	Reading	Not Allowed
Case 3	Reading	Writing	Not Allowed
Case 4	Reading	Reading	Allowed

Three variables are used: **mutex, wrt, readcnt**

1. Semaphore **mutex** is used to ensure mutual exclusion when **readcnt** is updated i.e. when any reader enters or exit from the critical section.
2. Semaphore **wrt** is used by both readers and writers.
3. **readcnt** tells the number of processes performing read in the critical section, initially 0 amd it is integer variable.

Functions for semaphore

wait() : decrements the semaphore value.

signal() : increments the semaphore value.

Reader process

- Reader requests the entry to critical section.
- If allowed:
  - it increments the count of number of readers inside the critical section. If this reader is the first reader entering, it locks the **wrt** semaphore to restrict the entry of writers if any reader is inside.
  - It then, signals mutex as any other reader is allowed to enter while others are already reading.
  - After performing reading, it exits the critical section. When exiting, it checks if no more reader is inside, it signals the semaphore "wrt" as now, writer can enter the critical section.
- If not allowed, it keeps on waiting.

```
do
{
  wait(mutex);        // Reader wants to enter the critical section
  readcnt++;          // The number of readers has now increased by 1

  if (readcnt==1)     // there is atleast one reader in the critical section
    wait(wrt);        //  no writer can enter if there is even one reader

  signal(mutex);      // other readers can enter where otherer is inside

   ….. perform READING

  wait(mutex);        // a reader wants to leave
  readcnt--;

if (readcnt == 0)     // no reader is left in the critical section,
     signal(wrt);     // writers can enter

  signal(mutex);      // reader leaves

} while(true);
```

Writer process

1. Writer requests the entry to critical section.
2. If allowed i.e. wait() gives a true value, it enters and performs the write. If not allowed, it keeps on waiting.
3. It exits the critical section.

```
do
{
  wait(wrt);                    // writer requests for critical section

     …perform WRITING

  signal(wrt);                  // leaves the critical section

} while(true);
```

Thus, the semaphore '**wrt**' is queued on both readers and writers in a manner such that preference is given to readers if writers are also there. Thus, no reader is waiting simply because a writer has requested to enter the critical section.

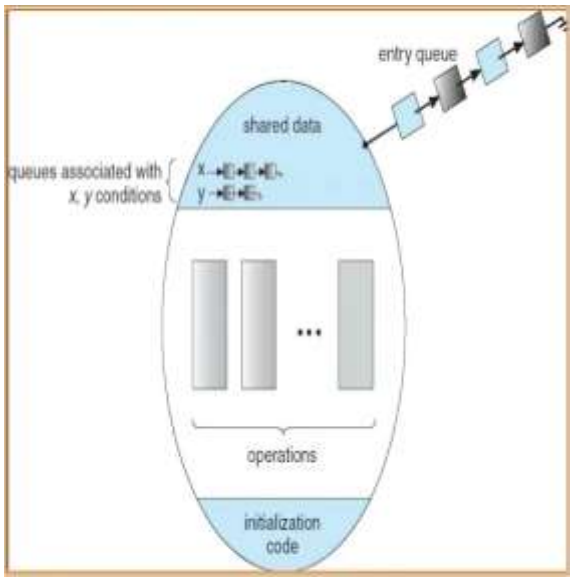
MONITOR

It is a synchronization technique that enables threads to mutual exclusion and the **wait()** for a given condition to become true. It is an abstract data type. It has a shared variable and a collection of procedures executing on the shared variable. A process may not directly access the shared data variables, and procedures are required to allow several processes to access the shared data variables simultaneously.

At any particular time, only one process may be active in a monitor. Other processes that require access to the shared variables must queue and are only granted access after the previous process releases the shared variables.

Syntax:

```
monitor
{
  //shared variable declarations
   data variables;
   Procedure P1() { ... }
   Procedure P2() { ... }
   .
   .
   .
   Procedure Pn() { ... }
   Initialization Code() { ... }
}
```



Advantages

- Mutual exclusion is automatic in monitors.
- Monitors are less difficult to implement than semaphores.
- Monitors may overcome the timing errors that occur when semaphores are used.
- Monitors are a collection of procedures and condition variables that are combined in a special type of module.

Disadvantages

- Monitors must be implemented into the programming language.
- The compiler should generate code for them.
- It gives the compiler the additional burden of knowing what operating system features is available for controlling access to crucial sections in concurrent processes.

Comparison between the Semaphore and Monitor

Features	Semaphore	Monitor
Definition	A semaphore is an integer variable that allows many processes in a parallel system to manage access to a common resource like a multitasking OS.	It is a synchronization process that enables threads to have mutual exclusion and the wait() for a given condition to become true.

Syntax	// Wait Operation
	wait(Semaphore S)
	{
	while (S<=0);
	S--;
	}
	// Signal Operation
	signal(Semaphore S)
	{
	S++;
	}
		Monitor
		{
		//shared variable declarations
		
		Procedure P1() { ... }
		Procedure P2() { ... }
		.
		.
		.
		Procedure Pn() { ... }
		Initialization Code() { ... }
		}

Basic	Integer variable	Abstract data type
Access	When a process uses shared resources, it calls the wait() method on S, and when it releases them, it uses the signal() method on S.	When a process uses shared resources in the monitor, it has to access them via procedures.
Action	The semaphore's value shows the number of shared resources available in the system.	The Monitor type includes shared variables as well as a set of procedures that operate on them.
Condition Variable	No condition variables.	It has condition variables.