UNIT – 3 : DEADLOCK AND DEVICE MANAGEMENT

PART – 1 : DEADLOCK

System Model:

- For the purposes of deadlock discussion, a system can be modelled as a collection of limited resources, which can be partitioned into different categories, to be allocated to a number of processes, each having different needs.
- Resource categories may include memory, printers, CPUs, open files, tape drives, CD-ROMS, etc.
- By definition, all the resources within a category are equivalent, and a request of this category can be equally satisfied by any one of the resources in that category. If this is not the case (i.e. if there is some difference between the resources within a category), then that category needs to be further divided into separate categories. For example, "printers" may need to be separated into "laser printers" and "color inkjet printers".
- Some categories may have a single resource.
- In normal operation a process must request a resource before using it, and release it when it is done, in the following sequence:

1. **Request** - If the request cannot be immediately granted, then the process must wait until the resource(s) it needs become available. For example the system calls open( ), malloc( ), new( ), and request( ).
2. **Use** - The process uses the resource, e.g. prints to the printer or reads from the file.
3. **Release** - The process relinquishes the resource. so that it becomes available for other processes. For example, close( ), free( ), delete( ), and release( ).

- For all kernel-managed resources, the kernel keeps track of what resources are free and which are allocated, to which process they are allocated, and a queue of processes waiting for this resource to become available. Application-managed resources can be controlled using mutexes or wait( ) and signal( ) calls, (i.e. binary or counting semaphores.)
- A set of processes is deadlocked when every process in the set is waiting for a resource that is currently allocated to another process in the set (and which can only be released when that other waiting process makes progress.)

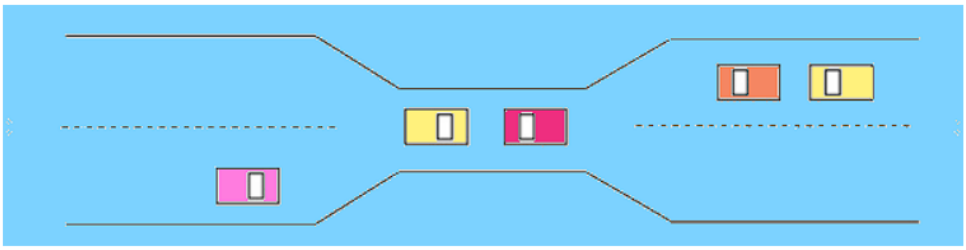
Deadlock Characterization:

What is Deadlock?

Deadlock is a situation that occurs in OS when any process enters a waiting state because another waiting process is holding the demanded resource. Deadlock is a common problem in multi-processing where several processes share a specific type of mutually exclusive resource known as a soft lock or software.

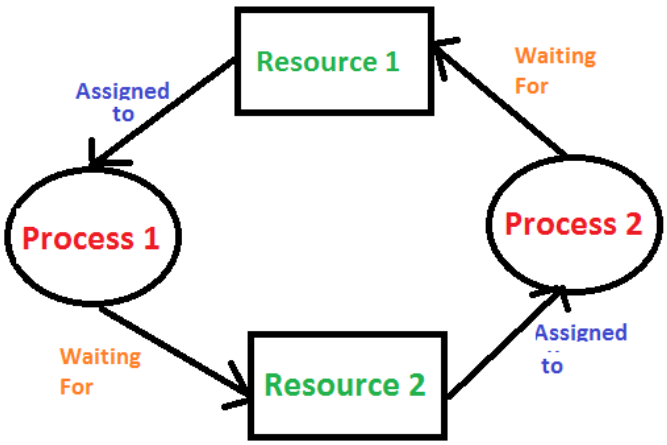
Example of Deadlock:

- A real-world example would be traffic, which is going only in one direction.
- Here, a bridge is considered a resource.
- So, when Deadlock happens, it can be easily resolved if one car backs up (Preempt resources and rollback).
- Several cars may have to be backed up if a deadlock situation occurs.
- So, starvation is possible.



Example of deadlock

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.

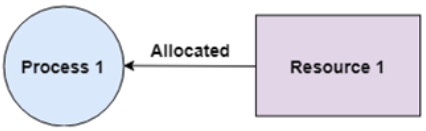


Necessary Conditions for Deadlock:

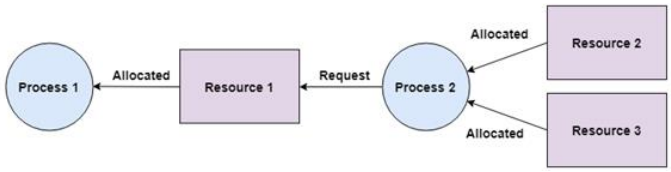
A deadlock occurs if the four **Coffman conditions** hold true. But these conditions are not mutually exclusive.

The Coffman conditions are given as follows −

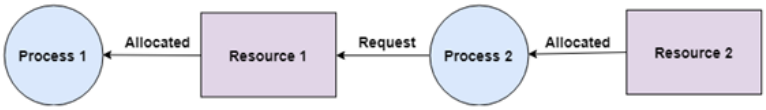
1. **Mutual Exclusion:** There should be a resource that can only be held by one process at a time. In the diagram below, there is a single instance of Resource 1 and it is held by Process 1 only.



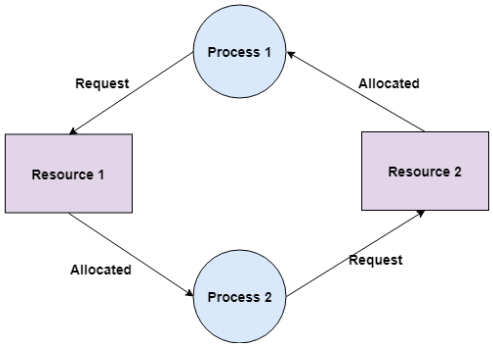
2. **Hold and Wait:** A process can hold multiple resources and still request more resources from other processes which are holding them. In the diagram given below, Process 2 holds Resource 2 and Resource 3 and is requesting the Resource 1 which is held by Process 1.



3. **No Preemption:** A resource cannot be preempted from a process by force. A process can only release a resource voluntarily. In the diagram below, Process 2 cannot preempt Resource 1 from Process 1. It will only be released when Process 1 relinquishes it voluntarily after its execution is complete.



4. **Circular Wait:** A process is waiting for the resource held by the second process, which is waiting for the resource held by the third process and so on, till the last process is waiting for a resource held by the first process. This forms a circular chain. 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 (Ostrich Method):** According to this method, it is assumed that deadlock would never occur. This approach is used by many operating systems where they assume that deadlock will never occur which means operating systems simply ignores the deadlock. This approach can be beneficial for those systems that are only used for browsing and for normal tasks. Thus, ignoring the deadlock method can be useful in many cases but it is not perfect in order to remove the deadlock from the operating system.

2. **Deadlock Prevention:** As we have discussed in the above section, that all four conditions: Mutual Exclusion, Hold and Wait, No preemption, and circular wait if held by a system then causes deadlock to occur. The main aim of the deadlock prevention method is to violate any one condition among the four; because if any of one condition is violated then the problem of deadlock will never occur. As the idea behind this method is simple but the difficulty can occur during the physical implementation of this method in the system.

3. **Deadlock Avoidance:** This method is used by the operating system in order to check whether the system is in a safe state or in an unsafe state. This method checks every step performed by the operating system. Any process continues its execution until the system is in a safe state. Once the system enters into an unsafe state, the operating system has to take a step back.

   Basically, with the help of this method, the operating system keeps an eye on each allocation, and make sure that allocation does not cause any deadlock in the system.

4. **Deadlock Detection and Recovery:** With this method, the deadlock is detected first by using some algorithms of the resource-allocation graph. This graph is mainly used to represent the allocations of various resources to different processes. After the detection of deadlock, a number of methods can be used in order to recover from that deadlock.

   One way is **preemption** by the help of which a resource held by one process is provided to another process.

   The second way is to **roll back**, as the operating system keeps a record of the process state and it can easily make a process roll back to its previous state due to which deadlock situation can be easily eliminate.

   The third way to overcome the deadlock situation is by killing one or more processes.

Difference Between Starvation and Deadlock:

Basis	Starvation	Deadlock
Definition	It is a process wherein resource-allocation is never done to low-priority resources and subsequently, they are never executed.	It is a situation where more than one process taking place in a system is prevented from getting executed because no resource is allocated to it.
Resource allocation	Here, resources are allocated to high-priority processes only.	Resource allocation is not done.
Ways to handle	A way to handle starvation is via aging.	By avoiding any of the four necessary conditions for deadlock.
Execution	Because of resource allocation, only high-priority resources are executed.	No process ever gets executed.
Other names	Starvation in OS is also termed as “lived lock”.	Also termed as “circular wait”.

Advantages of Deadlock:

- **Perfect for a single task:** The deadlock method is suitable for performing single tasks.
- **No preemptions:** There is no need to block a process from accessing that resource and preempting it to some other process.
- **Lucrative method:** If your resource can get stored as well as restored quickly, then this is the method for you.
- **No computations required:** Sincere thanks to all the problem-solving done in the system design, you can now bid adieu to run-time algorithmic computations.

Disadvantages of Deadlock:

- **Late initiation:** One of the biggest flaws of the deadlock method is that it lags in process initiation.
- **Forestall losses:** There is no iota of doubt that the deadlock method has inherent forestalled losses.
- **Preemptions are encountered frequently:** This method cannot be deemed good since it results in taking away the resource from the process making the output fetched up till now totally inconsistent.
- **No piecemeal resources:** The deadlock method doesn’t approve any request asking for gradational resources.
- **Unawareness about future needs:** Usually, processes ain’t aware of their future resource requirements.

Deadlock Prevention:

If we simulate deadlock with a table which is standing on its four legs then we can also simulate four legs with the four conditions which when occurs simultaneously, cause the deadlock.

However, if we break one of the legs of the table then the table will fall definitely. The same happens with deadlock, if we can be able to violate one of the four necessary conditions and don't let them occur together then we can prevent the deadlock.

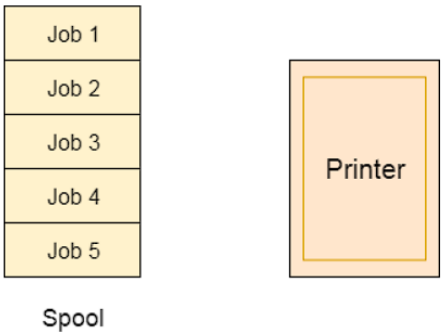
Let's see how we can prevent each of the conditions -

1. Mutual Exclusion: Mutual section from the resource point of view is the fact that a resource can never be used by more than one process simultaneously which is fair enough but that is the main reason behind the deadlock. If a resource could have been used by more than one process at the same time then the process would have never been waiting for any resource.

However, if we can be able to violate resources behaving in the mutually exclusive manner then the deadlock can be prevented.

Spooling:

For a device like printer, spooling can work. There is a memory associated with the printer which stores jobs from each of the process into it. Later, Printer collects all the jobs and print each one of them according to FCFS. By using this mechanism, the process doesn't have to wait for the printer and it can continue whatever it was doing. Later, it collects the output when it is produced.



Although, Spooling can be an effective approach to violate mutual exclusion but it suffers from two kinds of problems.

1. This cannot be applied to every resource, only be used for resources with associated memory, like a Printer.
2. After some point of time, there may arise a race condition between the processes to get space in that spool.

We cannot force a resource to be used by more than one process at the same time since it will not be fair enough and some serious problems may arise in the performance. Therefore, we cannot violate mutual exclusion for a process practically.

2. Hold and Wait:

Hold and wait condition lies when a process holds a resource and waiting for some other resource to complete its task. Deadlock occurs because there can be more than one process which are holding one resource and waiting for other in the cyclic order.

However, we have to find out some mechanism by which a process either doesn't hold any resource or doesn't wait. That means, a process must be assigned all the necessary resources before the execution starts. A process must not wait for any resource once the execution has been started.

!(Hold and wait) = !hold or !wait (negation of hold and wait is, either you don't hold or you don't wait)

This can be implemented practically if a process declares all the resources initially. However, this sounds very practical but can't be done in the computer system because a process can't determine necessary resources initially.

Process is the set of instructions which are executed by the CPU. Each of the instruction may demand multiple resources at the multiple times. The need cannot be fixed by the OS.

The problem with the approach is:

1. Practically not possible.
2. Possibility of getting starved will be increases due to the fact that some process may hold a resource for a very long time.

3. No Preemption:

Preemption is temporarily interrupting an executing task and later resuming it.

For example, if process P1 is using a resource and a high-priority process P2 requests for the resource, process P1 is stopped and the resources are allocated to P2.

There are two ways to eliminate this condition by preemption:

1. If a process is holding some resources and waiting for other resources, then it should release all previously held resources and put a new request for the required resources again. The process can resume once it has all the required resources.

   For example: If a process has resources R1, R2, and R3 and it is waiting for resource R4, then it has to release R1, R2, and R3 and put a new request of all resources again.

2. If a process P1 is waiting for some resource, and there is another process P2 that is holding that resource and is blocked waiting for some other resource. Then the resource is taken from P2 and allocated to P1. This way process P2 is preempted and

it requests again for its required resources to resume the task. The above approaches are possible for resources whose states are easily restored and saved, such as memory and registers.

Challenges:

- These approaches are problematic as the process might be actively using these resources and halting the process by preempting can cause inconsistency.

  For example: If a process is writing to a file and its access is revoked for the process before it completely updates the file, the file will remain unusable and in an inconsistent state.

- Putting requests for all resources again is inefficient and time-consuming.

4. Circular Wait:

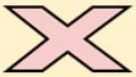
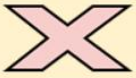
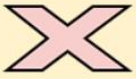
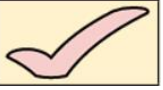
To violate circular wait, we can assign a priority number to each of the resource. A process can't request for a lesser priority resource. This ensures that not a single process can request a resource which is being utilized by some other process and no cycle will be formed.

Challenges:

- It is difficult to assign a relative priority to resources, as one resource can be prioritized differently by different processes.

  For Example: A media player will give a lesser priority to a printer while a document processor might give it a higher priority. The priority of resources is different according to the situation and use case.

Feasibility of Deadlock Prevention:

Condition	Approach	Is Practically Possible?
Mutual Exclusion	Spooling	
Hold and Wait	Request for all the resources initially	
No Preemption	Snatch all the resources	
Circular Wait	Assign priority to each resources and order resources numerically	

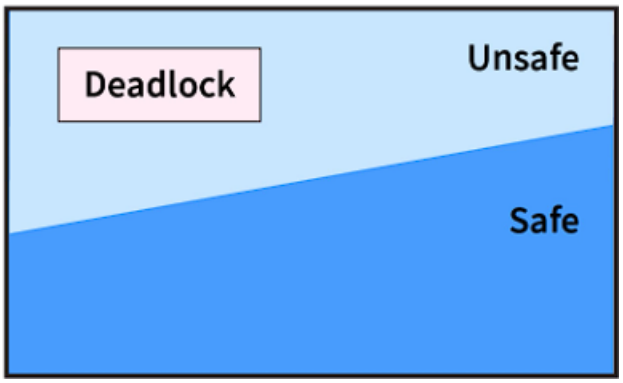
Among all the methods, violating Circular wait is the only approach that can be implemented practically.

Deadlock Avoidance:

- The general idea behind deadlock avoidance is to prevent deadlocks from ever happening, by preventing at least one of the aforementioned conditions.
- This requires more information about each process, AND tends to lead to low device utilization. (i.e. it is a conservative approach.)
- In some algorithms the scheduler only needs to know the *maximum* number of each resource that a process might potentially use. In more complex algorithms the scheduler can also take advantage of the *schedule* of exactly what resources may be needed in what order.
- When a scheduler sees that starting a process or granting resource requests may lead to future deadlocks, then that process is just not started or the request is not granted.
- A resource allocation ***state*** is defined by the number of available and allocated resources, and the maximum requirements of all processes in the system.

Safe State and Unsafe State:

- A state is ***safe*** if the system can allocate all resources requested by all processes (up to their stated maximums) without entering a deadlock state.
- More formally, a state is safe if there exists a ***safe sequence*** of processes {P0, P1, P2, ..., PN} such that all of the resource requests for Pi can be granted using the resources currently allocated to Pi and all processes Pj where j < i. (i.e. if all the processes prior to Pi finish and free up their resources, then Pi will be able to finish also, using the resources that they have freed up.)
- If a safe sequence does not exist, then the system is in an **unsafe state**, which ***MAY*** lead to deadlock. (All safe states are deadlock free, but not all unsafe states lead to deadlocks.)



The above Figure shows the Safe, unsafe, and deadlocked state spaces

- Let us consider a system having 12 magnetic tapes and three processes P1, P2, P3. Process P1 requires 10 magnetic tapes, process P2 may need as many as 4 tapes, process P3 may need up to 9 tapes. Suppose at a time to, process P1 is holding 5 tapes, process P2 is holding 2 tapes and process P3 is holding 2 tapes. (There are 3 free magnetic tapes). Is this a safe state? What is the safe sequence?

	Maximum Needs	**Current Allocation**
P0	10	5
P1	4	2
P2	9	2

So at time t0, the system is in a safe state.The sequence is <P2,P1,P3> satisfies the safety condition.Process P2 can immediately be allocated all its tape drives and then return them. After the return the system will have 5 available tapes, then process P1 can get all its tapes and return them (the system will then have 10 tapes); finally, process P3 can get all its tapes and return them (The system will then have 12 available tapes).

A system can go from a safe state to an unsafe state. Suppose at time t1, process P3 requests and is allocated one more tape. The system is no longer in a safe state. At this point, only process P2 can be allocated all its tapes. When it returns them the system will then have only 4 available tapes.Since P1 is allocated five tapes but has a maximum of ten so it may request 5 more tapes. If it does so, it will have to wait because they are unavailable. Similarly, process P3 may request its additional 6 tapes and have to wait which then results in a deadlock.

The mistake was granting the request from P3 for one more tape. If we made P3 wait until either of the other processes had finished and released its resources, then we could have avoided the deadlock

Note: In a case, if the system is unable to fulfill the request of all processes then the state of the system is called unsafe.

The main key of the deadlock avoidance method is whenever the request is made for resources then the request must only be approved only in the case if the **resulting state** is a **safe state**.

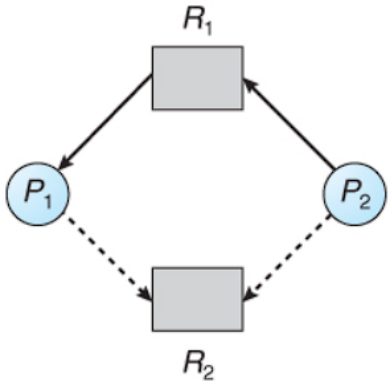
Deadlock Avoidance Solution:

Deadlock Avoidance can be solved by two different algorithms:

- **Resource allocation Graph**
- **Banker's Algorithm**

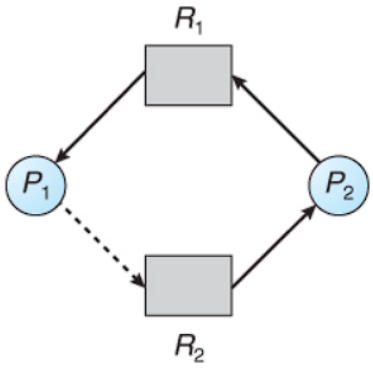
Resource-Allocation Graph Algorithm:

- If resource categories have only single instances of their resources, then deadlock states can be detected by cycles in the resource-allocation graphs.
- In this case, unsafe states can be recognized and avoided by augmenting the resource-allocation graph with ***claim edges***, noted by dashed lines, which point from a process to a resource that it may request in the future.
- In order for this technique to work, all claim edges must be added to the graph for any particular process before that process is allowed to request any resources. (Alternatively, processes may only make requests for resources for which they have already established claim edges, and claim edges cannot be added to any process that is currently holding resources.)
- When a process makes a request, the claim edge Pi->Rj is converted to a request edge. Similarly, when a resource is released, the assignment reverts back to a claim edge.
- This approach works by denying requests that would produce cycles in the resource-allocation graph, taking claim edges into effect.
- Consider for example what happens when process P2 requests resource R2:



Resource allocation graph for deadlock avoidance

- The resulting resource-allocation graph would have a cycle in it, and so the request cannot be granted.



An unsafe state in a resource allocation graph

Banker's Algorithm:

The banker's algorithm is a resource allocation and deadlock avoidance algorithm that tests for safety by simulating the allocation for the 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.

Why Banker's Algorithm is Named So?

The banker's algorithm is named so because it is used in the banking system to check whether a loan can be sanctioned to a person or not. Suppose there are n number of account holders in a bank and the total sum of their money is S. If a person applies for a loan, then the bank first subtracts the loan amount from the total money that the bank has and if the remaining amount is greater than S then only the loan is sanctioned. It is done because if all the account holders come to withdraw their money, then the bank can easily do it.

It also helps the OS to successfully share the resources between all the processes. It is called the banker's algorithm because bankers need a similar algorithm- they admit loans that collectively exceed the bank's funds and then release each borrower's loan in installments. The banker's algorithm uses the notation of a safe allocation state to ensure that granting a resource request cannot lead to a deadlock either immediately or in the future.
In other words, the bank would never allocate its money in such a way that it can no longer satisfy the needs of all its customers. The bank would try to be in a safe state always.

The 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 resource 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 **Rj**

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 **Pi** may request at most **'k'** instances of resource type **Rj.**

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 **Pi** is currently allocated **'k'** instances of resource type **Rj**

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 **Pi** currently needs **'k'** instances of resource type **Rj**
- Need [i, j] = Max [i, j] – Allocation [i, j]

Allocation specifies the resources currently allocated to process Pi and Needi specifies the additional resources that process Pi may still request to complete its task.

Banker's Algorithm:

1. **Active**:= Running U Blocked;

for k=1…r

New_ request[k]:= Requested_ resources[requesting_ process, k];

2. **Simulated_ allocation**:= Allocated_ resources;

for k=1…..r //Compute projected allocation state

Simulated_ allocation [requesting _process, k]:= Simulated_ allocation [requesting _process, k] + New_ request[k];

3. **feasible**:= true;

for k=1….r // Check whether projected allocation state is feasible

if Total_ resources[k]< Simulated_ total_ alloc [k] then feasible:= false;

4. **if feasible**= true

then // Check whether projected allocation state is a safe allocation state

while set Active contains a process P1 such that

For all k, Total _resources[k] – Simulated_ total_ alloc[k]>= Max_ need [l ,k]- Simulated_ allocation[l, k]

Delete Pl from Active;

for k=1…..r

Simulated_ total_ alloc[k]:= Simulated_ total_ alloc[k]- Simulated_ allocation[l, k];

5. I**f set Active is empty**

then // Projected allocation state is a safe allocation state

for k=1….r // Delete the request from pending requests

Requested_ resources[requesting_ process, k]:=0;

for k=1….r // Grant the request

Allocated_ resources[requesting_ process, k]:= Allocated_ resources[requesting_ process, k] + New_ request[k];

Total_ alloc[k]:= Total_ alloc[k] + New_ request[k];

Banker's algorithm consists of a Safety algorithm and a Resource request algorithm.

Safety Algorithm:

The algorithm for finding out whether or not a system is in a safe state can be described as follows:

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 Pi. $Request_i$ [j] = k means process Pi wants k instances of resource type Rj. When a request for resources is made by process Pi, the following actions are taken:

1) If $Request_i$ <= $Need_i$
Goto step (2); otherwise, raise an error condition, since the process has exceeded its maximum claim.

2) If $Request_i$ <= Available
Goto step (3); otherwise, Pi must wait, since the resources are not available.

3) Have the system pretend to have allocated the requested resources to process Pi by modifying the state as
follows:
Available = Available – $Request_i$
$Allocation_i$ = $Allocation_i$ + $Request_i$
$Need_i$ = $Need_i$– $Request_i$

Example:

Considering a system with five processes P0 through P4 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 t0 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	

Q.1: What will be the content of the Need matrix?

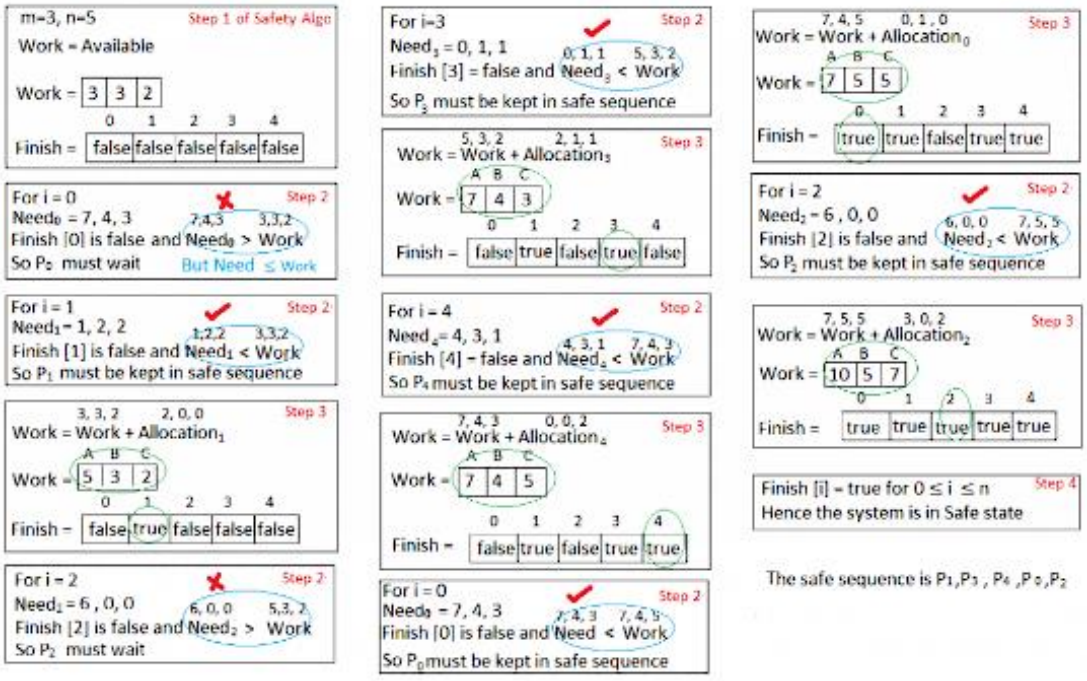
Need [i, j] = Max [i, j] – Allocation [i, j]

So, the content of Need Matrix is:

Process	Need		
	A	B	C
P_0	7	4	3
P_1	1	2	2
P_2	6	0	0
P_3	0	1	1
P_4	4	3	1

Q.2: Is the system in a safe state? If Yes, then what is the safe sequence?

Applying the Safety algorithm on the given system,



Q.3: What will happen if process P1 requests one additional instance of resource type A and two instances of resource type C?

A B C

$Request_1$ = 1, 0, 2

To decide whether the request is granted we use Resource Request algorithm

Step 1

1, 0, 2 < 1, 2, 2

$Request_1 < Need_1$

Step 2

1, 0, 2 < 3, 3, 2

$Request_1 < Available$

Step 3

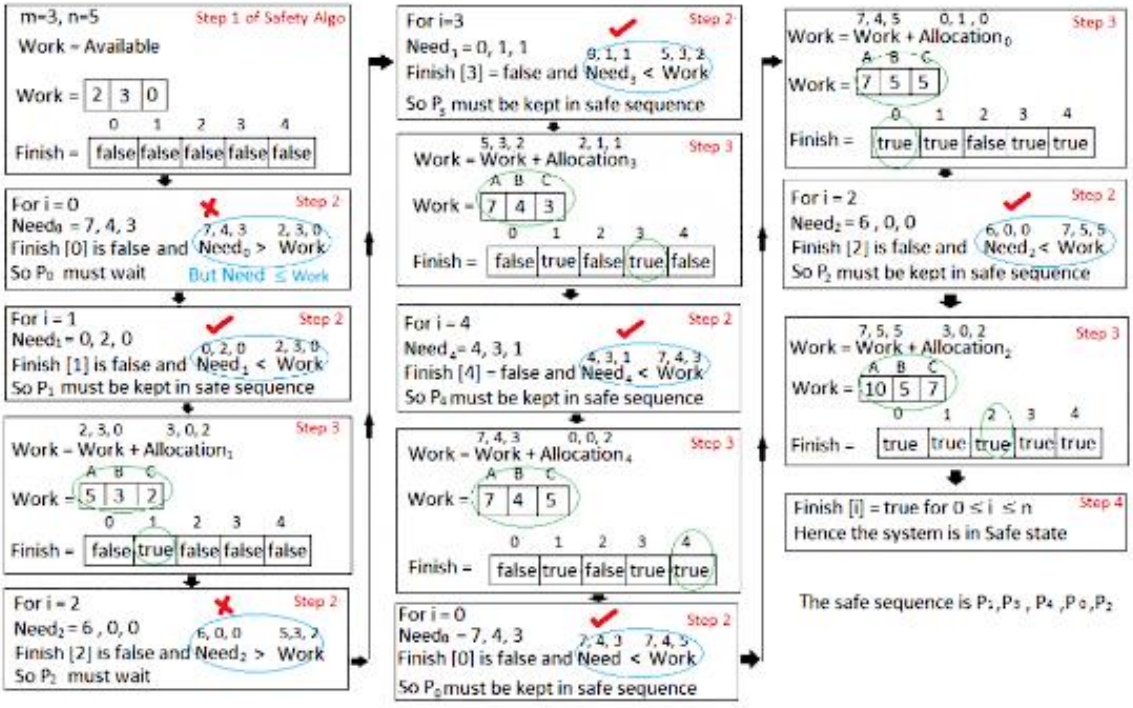
$Available = Available - Request_1$

$Allocation_1 = Allocation_1 + Request_1$

$Need_1 = Need_1 - Request_1$

Process	Allocation	Need	Available
	A B C	A B C	A B C
P_0	0 1 0	7 4 3	2 3 0
P_1	3 0 2	0 2 0	
P_2	3 0 2	6 0 0	
P_3	2 1 1	0 1 1	
P_4	0 0 2	4 3 1	

We must determine whether this new system state is safe. To do so, we again execute Safety algorithm on the above data structures.



Hence the new system state is safe, so we can immediately grant the request for process **P1.**

Weakness of Banker's Algorithm:

1. It requires that there be a fixed number of resources to allocate.

2. The algorithm requires that users state their maximum needs (request) in advance.

3. Number of users must remain fixed.

4. The algorithm requires that the bankers grant all requests within a finite time. 5. Algorithm requires that process returns all resource within a finite time.

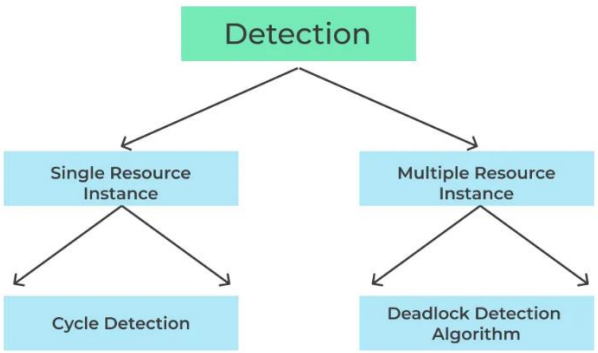
Deadlock Detection:

Deadlock detection involves identifying when a deadlock has occurred in the system. This can be done through the use of algorithms, such as the Banker's Algorithm or the Wait-For Graph Algorithm, which check for the presence of the necessary conditions for deadlock.

Detection:

There are 2 different cases in case of Deadlock detection –

1. **Deadlock detection using a centralized algorithm:** In this approach, a single entity, such as the operating system, is responsible for detecting and resolving deadlocks in the system. This entity periodically checks for the presence of deadlocks and takes appropriate action to resolve them.
2. **Deadlock detection using a distributed algorithm:** In this approach, multiple entities, such as processes or nodes in a distributed system, work together to detect and resolve deadlocks. Each entity maintains information about the resources it holds and requests, and communicates with other entities to detect and resolve deadlocks. This approach can be more efficient and scalable than a centralized approach, but it can also be more complex to implement.

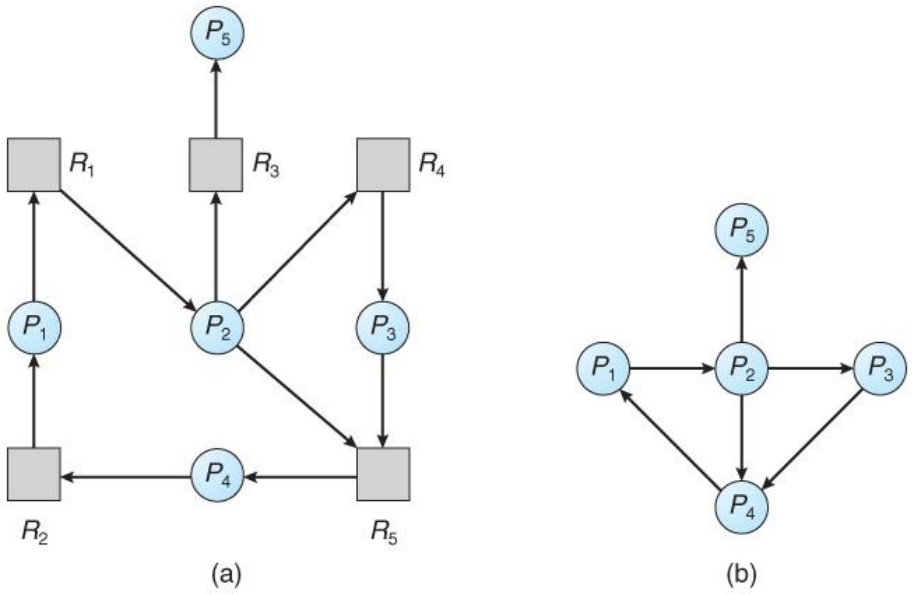


Both centralized and distributed algorithms have their own advantages and disadvantages. Centralized algorithms are simpler to implement but may have scalability issues, while distributed algorithms can be more efficient and handle large systems, but they can be more complex to implement.

If Resources has Single Instance:

- If each resource category has a single instance, then we can use a variation of the resource-allocation graph known as a **wait-for graph**.
- A wait-for graph can be constructed from a resource-allocation graph by eliminating the resources and collapsing the associated edges, as shown in the figure below.

- An arc from Pi to Pj in a wait-for graph indicates that process Pi is waiting for a resource that process Pj is currently holding.



(a) Resource allocation graph. *(b) Corresponding wait-for graph*

- As before, cycles in the wait-for graph indicate deadlocks.
- This algorithm must maintain the wait-for graph, and periodically search it for cycles.

If there are multiple instances of resources –

- The detection algorithm outlined here is essentially the same as the Banker's algorithm, with two subtle differences:
  - In step 1, the Banker's Algorithm sets Finish[i] to false for all i. The algorithm presented here sets Finish[i] to false only if Allocation[i] is not zero. If the currently allocated resources for this process are zero, the algorithm sets Finish[i] to true. This is essentially assuming that IF all of the other processes can finish, then this process can finish also. Furthermore, this algorithm is specifically looking for which processes are involved in a deadlock situation, and a process that does not have any resources allocated cannot be involved in a deadlock, and so can be removed from any further consideration.
  - Steps 2 and 3 are unchanged
  - In step 4, the basic Banker's Algorithm says that if Finish[i] == true for all i, that there is no deadlock. This algorithm is more specific, by stating that if Finish[i] == false for any process Pi, then that process is specifically involved in the deadlock which has been detected.

- **Note:** An alternative method was presented above, in which Finish held integers instead of Booleans. This vector would be initialized to all zeros, and then filled with increasing integers as processes are detected which can finish. If any processes are left at zero when the algorithm completes, then there is a deadlock, and if not, then the integers in finish describe a safe sequence. To modify this algorithm to match this section of the text, processes with allocation = zero could be filled in with N, N - 1, N - 2, etc. in step 1, and any processes left with Finish = 0 in step 4 are the deadlocked processes.
- Consider, for example, the following state, and determine if it is currently deadlocked:

	Allocation	*Request*	*Available*
	A B C	*A B C*	*A B C*
P_0	0 1 0	0 0 0	0 0 0
P_1	2 0 0	2 0 2	
P_2	3 0 3	0 0 0	
P_3	2 1 1	1 0 0	
P_4	0 0 2	0 0 2	

- Now suppose that process P2 makes a request for an additional instance of type C, yielding the state shown below. Is the system now deadlocked?

	Allocation	*Request*	*Available*
	A B C	*A B C*	*A B C*
P_0	0 1 0	0 0 0	0 0 0
P_1	2 0 0	2 0 2	
P_2	3 0 3	0 0 1	
P_3	2 1 1	1 0 0	
P_4	0 0 2	0 0 2	

Detection-Algorithm Usage:

- When should the deadlock detection be done? Frequently, or infrequently?
- The answer may depend on how frequently deadlocks are expected to occur, as well as the possible consequences of not catching them immediately. (If deadlocks are not removed immediately when they occur, then more and more processes can "back up" behind the deadlock, making the eventual task of unblocking the system more difficult and possibly damaging to more processes.)
- There are two obvious approaches, each with trade-offs:

1. Do deadlock detection after every resource allocation which cannot be immediately granted. This has the advantage of detecting the deadlock right away, while the minimum number of processes are involved in the deadlock. (One might consider that the process whose request triggered the deadlock condition is the "cause" of the deadlock, but realistically all of the processes in the cycle are equally responsible for the resulting deadlock.) The down side of this approach is the extensive overhead and performance hit caused by checking for deadlocks so frequently.
2. Do deadlock detection only when there is some clue that a deadlock may have occurred, such as when CPU utilization reduces to 40% or some other magic number. The advantage is that deadlock detection is done much less frequently, but the down side is that it becomes impossible to detect the processes involved in the original deadlock, and so deadlock recovery can be more complicated and damaging to more processes.

Recovery From Deadlock:

There are three basic approaches to recovery from deadlock:

1. Inform the system operator, and allow him/her to take manual intervention.
2. Terminate one or more processes involved in the deadlock
3. Preempt resources.

Process Termination:

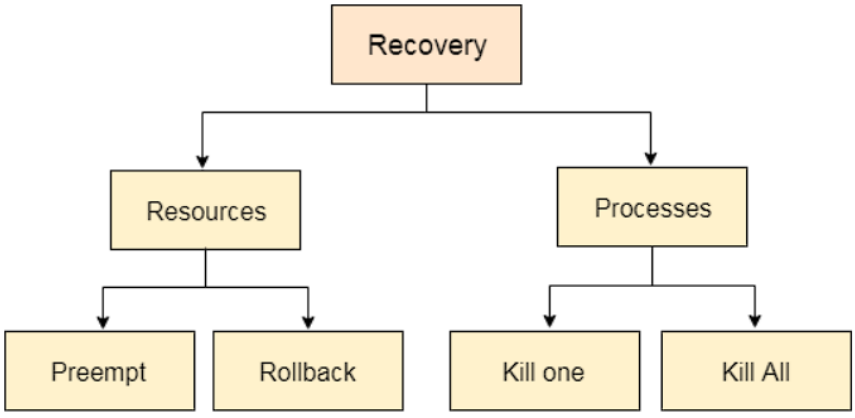
- Two basic approaches, both of which recover resources allocated to terminated processes:
  - Terminate all processes involved in the deadlock. This definitely solves the deadlock, but at the expense of terminating more processes than would be absolutely necessary.
  - Terminate processes one by one until the deadlock is broken. This is more conservative, but requires doing deadlock detection after each step.
- In the latter case there are many factors that can go into deciding which processes to terminate next:
  1. Process priorities.
  2. How long the process has been running, and how close it is to finishing.
  3. How many and what type of resources is the process holding. (Are they easy to preempt and restore?)
  4. How many more resources does the process need to complete.

5. How many processes will need to be terminated
6. Whether the process is interactive or batch.
7. Whether or not the process has made non-restorable changes to any resource.

Resource Preemption:

When preempting resources to relieve deadlock, there are three important issues to be addressed:

1. **Selecting a victim** - Deciding which resources to preempt from which processes involves many of the same decision criteria outlined above.
2. **Rollback** - Ideally one would like to roll back a preempted process to a safe state prior to the point at which that resource was originally allocated to the process. Unfortunately, it can be difficult or impossible to determine what such a safe state is, and so the only safe rollback is to roll back all the way back to the beginning. (i.e. abort the process and make it start over.)
3. **Starvation** - How do you guarantee that a process won't starve because its resources are constantly being preempted? One option would be to use a priority system, and increase the priority of a process every time its resources get preempted. Eventually it should get a high enough priority that it won't get preempted any more.

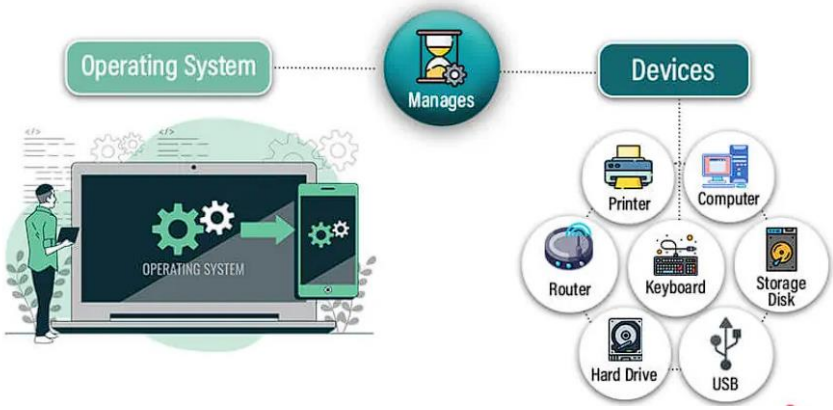


PART – 2 : DEVICE MANAGEMENT

Introduction:

Device management in operating system controls the hardware or virtual device devices in a computer/PC to make sure everything functions properly. The device management allocates the input/output devices such as discs, microphones, keyboards, printers, magnetic tape, USB ports, scanners, and various other devices. They could also be virtual, such as virtual machines or virtual switches. During the execution of a program, it may require various computer resources (devices) for its complete execution.

The operating system has the onus upon it to provide judiciously the resources required. It is the sole responsibility of the operating system to check if the resource is available or not. It is not just concerned with the allocation of the devices but also with the deallocation, i.e., once the requirement of a device/resource is over by a process, it must be taken away from the process.



Devices and Their Characteristics:

Devices refer to hardware components that are connected to the computer.

Fundamental Types of I/O Devices:

In the realm of computing, Input/Output (I/O) devices serve as the essential conduits that connect our digital world to the physical one. They come in various forms, each tailored to specific functions and purposes.

Here, we explore three fundamental categories:

- **Boot Devices:** Boot devices are the digital gateways to your computer's soul. They house the initial firmware or bootloader that kicks off the operating system's startup process. These devices include hard drives, SSDs, USB drives, and network boot options. When you power up your machine, it searches for the bootloader in these devices, initiating the boot sequence that brings your computer to life.

- **Character Devices:** Character devices are the unsung heroes of I/O. They handle data character-by-character, making them ideal for real-time interactions. This category encompasses devices like keyboard and mouse, which send individual keystrokes and mouse movements to the computer. Character devices enable you to interact with your computer, shaping the user experience with every key press and mouse click.
- **Network Devices:** In our interconnected world, network devices reign supreme. They facilitate computer data exchange, enabling communication and collaboration across vast distances. Network devices encompass network interface cards (NICs), routers, modems, and switches. They are the linchpins of the internet and local networks, ensuring seamless data transfer and connectivity.

These fundamental I/O device types are the building blocks of modern computing, and understanding their roles is essential for both users and system administrators. As technology advances, the boundaries between these categories continue to blur, giving rise to new, hybrid devices that further enrich our digital experiences.

Classification of Operating System Peripheral Devices:

In the intricate world of operating systems, the management and classification of peripheral devices is a crucial task. These devices serve as the vital links between the digital realm and the physical hardware, and they come in diverse forms, each necessitating unique handling.

In this context, we delve into three distinct categories:

- **Dedicated Devices:** Dedicated devices are the specialists of the peripheral world. These are hardware components exclusively reserved for a specific task or application. Examples include- graphics processing units (GPUs) that focus on rendering graphics and sound cards that handle audio processing. Dedicated devices are optimized for their intended roles, delivering high performance and efficiency.
- **Shared Devices:** Shared devices, as the name suggests, are versatile team players. These devices can be accessed and utilized by multiple applications or processes concurrently. Printers, for instance, can be shared among various software applications, allowing several programs to print documents simultaneously. Shared devices are the multitaskers of the peripheral landscape.
- **Virtual Devices:** Virtual devices introduce a layer of abstraction and flexibility into the mix. These are not physical components but software emulations of hardware devices. Virtual devices are instrumental in virtualization technology, allowing multiple virtual machines to interact with emulated hardware as if it were real. This dynamic category enables efficient resource allocation, isolation, and management in virtualized environments.

Functions of the Device Management:

Device management in an operating system is the silent conductor that orchestrates the intricate interplay between hardware and software. Its functions are critical in ensuring seamless operation and resource utilization. Here are the key roles it plays:

- **Device Detection:** Device management starts with the identification of hardware components. It scans the system to detect connected devices, verifying their presence and characteristics.
- **Driver Communication:** Once devices are identified, device management interacts with device drivers. Drivers act as translators, enabling the OS to communicate with specific hardware. Device management ensures the right driver is loaded for each device.
- **Resource Allocation:** Operating systems often manage multiple applications concurrently. Device management allocates resources like CPU time, memory, and I/O bandwidth, ensuring each device receives its fair share.
- **Device Configuration:** Devices often require specific settings or configurations to operate optimally. Device management handles these configurations, including resolution settings for monitors, network settings for network adapters, and more.
- **Error Handling:** When issues arise, device management plays a crucial role in error detection and recovery. It interprets error codes, initiates error recovery mechanisms, and, in some cases, can even hot-swap faulty devices.
- **Security Management:** Security is paramount. Device management ensures that only authorized applications can access specific devices, guarding against unauthorized access or potential security breaches.
- **Power Management:** Modern devices often require power management, allowing them to enter low-power states when not in use. Device management regulates these power states to conserve energy and extend the device's lifespan.
- **Plug-and-Play Support:** Device management enables the seamless addition and removal of devices, ensuring new devices are recognized and operational without requiring system reboots.

Features of Device Management:

We have seen the functions of the Device management system in the Operating system in the above article now let's look at the features it provides:

1. To communicates with the device controllers to allocate the device to the various processes running on the computer the device management operating system takes the help of the device drivers.

2. Device drivers are the software programs that work as the translator between the hardware signals and the high-level programming language of the operating system and are used to control the functioning of devices in the proper or expected manner.
3. Command, status, and data registers are mainly used by the device controller in the device management operating system.
4. As we have discussed above the device drivers are mediators between the software and hardware.
5. Application Programming Interface or API is the set of programs or functions that allows the application to interact with the software and the data is handled by the device management operating system.

What are the Various Techniques for Accessing a Device?

- **Polling:** In this instance, a CPU keeps an eye on the status of the device to share data. Busy-waiting is a drawback, but simplicity is a plus. In this scenario, when an input/output operation is needed, the computer simply keeps track of the I/O device's status until it's ready, at which time it is accessed. Stated differently, the computer waits for the device to be ready.
- **Interrupt-Driven I/O:** Notifying the associated driver of the device's availability is the device controller's job. One interrupt for each keyboard input results in slower data copying and movement for character devices, but the advantages include more effective use of CPU cycles. A block of bytes is created from a serial bit stream by a device controller. It also does error correction if needed. It consists of two primary parts: a data buffer that an operating system can read or write to, and device registers for communication with the CPU.
- **DMA (Direct Memory Access):** Data motions are carried out by using a second controller. This approach has the benefit of not requiring the CPU to duplicate data, but it also has the drawback of preventing a process from accessing data that is in transit.
- **Double Buffering:** This mode of access has two buffers. One fills up while the other is utilised, and vice versa. In order to hide the line-by-line scanning from the viewer, this technique is frequently employed in animation and graphics.

Device Drivers:

They are the intermediaries between your operating system and the myriad hardware components within your computer. These small but mighty pieces of software serve two fundamental roles.

Role of Device Drivers:

- **Translation and Communication:** Device drivers act as interpreters. They translate generic commands from the operating system into language that specific hardware components understand. They facilitate seamless communication between software applications and hardware devices.
- **Optimization:** Device drivers are tailored to extract the best performance from hardware. They unlock the full potential of your devices, ensuring they work efficiently and effectively.

Types of Device Drivers:

There are several types of device drivers, including:

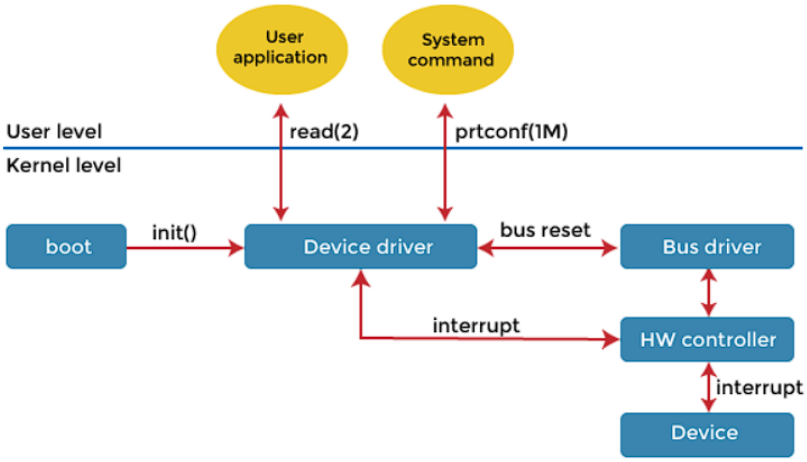
- **Kernel Drivers:** These reside within the core of the operating system and provide direct communication with hardware. Nestled within the kernel, the central component of the OS, these drivers provide direct communication with hardware components. They enable the operating system to harness the full potential of your devices, ensuring smooth, low-level interactions. Kernel drivers operate at the system's core, making them integral to your computer's performance and stability. They are the guardians of your hardware, ensuring that every command and data transfer happens with precision and efficiency, making them the unsung heroes of your digital experience.
- **Virtual Device Drivers:** Used in virtualization environments to create and manage emulated hardware. These drivers don't control physical components but create software emulations of hardware devices. They are crucial in virtualization technology, facilitating the operation of multiple virtual machines (VMs) on a single physical host. Virtual device drivers allow VMs to interact with emulated hardware as if it were real, abstracting the underlying physical components and enhancing the flexibility and isolation of virtualized environments. They are pivotal in optimizing resource allocation and ensuring efficient operation within the ever-evolving virtualization landscape.
- **Filter Drivers:** These intercept data as it passes between the operating system and the device, often for purposes like encryption or compression. These drivers serve diverse purposes, from data encryption and compression to firewall filtering and antivirus scans. Filter drivers enable real-time data analysis and transformation, safeguarding data integrity and security. They play a pivotal role in shaping the quality, privacy, and safety of data flows, ensuring that information passes through a series of critical checkpoints before reaching its destination. In this way, filter drivers act as the guardians protecting your data from harm.

- **Plug and Play Drivers:** These enable the system to configure and manage newly connected devices automatically. These drivers automatically detect new hardware connections, configure settings, and ensure the hardware is ready for use without requiring a system reboot. PnP drivers transform the once cumbersome task of installing and configuring hardware into a hassle-free, user-friendly experience. They play a pivotal role in the modern era of effortless hardware connectivity, where you can plug in a new device, and it's instantly recognized and put to work, simplifying your tech experience.

How does Device Driver work?

When you get a peripheral device such as a printer, scanner, keyboard or modem, the device comes together with a driver CD which needs to be installed before the device starts working. As soon we install the driver software into the computer, it detects and identifies the peripheral device, and we become able to control it.

A device driver is a piece of software that allows your computer's operating system to communicate with a hardware device the driver is written for. Generally, a driver communicates with the device through the ***computer bus***, which connects the device with the computer. Device Drivers depend upon the Operating System's instruction to access the device and performing any particular action. After the action, they also show their reactions by delivering output or message from the hardware device to the Operating system.



Device drivers work within the ***kernel*** layer of the operating system. The kernel is the part of the operating system that directly interacts with the system's physical structure. Instead of accessing a device directly, an operating system loads the device drivers and calls the specific functions in the driver software to execute specific tasks on the device. Each driver contains the device-specific codes required to carry out the actions on the device.

Card reader, controller, modem, network card, sound card, printer, video card, USB devices, RAM, Speakers etc., need Device Drivers to operate.

For example, a printer driver tells the printer which format to print after getting instructions from OS. Similarly, A sound card driver is there because the 1's and 0's data of an MP3 file is converted to audio signals, and you enjoy the music.

Importance of Device Drivers:

1. **Hardware Support**: Device drivers are crucial for enabling the OS to support a wide range of hardware devices. Without proper drivers, hardware components would be unrecognized or non-functional.
2. **System Stability**: Well-designed and updated drivers contribute to the stability of the system. They help prevent crashes, freezes, and other issues related to hardware communication.
3. **Performance**: Optimized drivers can improve the performance of hardware devices. For example, a graphics driver optimized for a specific GPU can enhance gaming performance and video rendering.
4. **Security**: Updated drivers often include security patches and improvements. Outdated drivers may have vulnerabilities that could be exploited by malicious software.
5. **Plug-and-Play**: Device drivers enable plug-and-play functionality, allowing users to connect new devices to their computers and have them recognized and configured automatically.

Installing and Updating Drivers:

- **Automatic Updates**: Modern operating systems often include mechanisms to automatically download and install updated drivers from the internet.
- **Manual Installation**: Users can also manually install drivers provided by hardware manufacturers. This is common when specific configurations or optimizations are needed.
- **Device Manager (Windows)**: In Windows, Device Manager allows users to view, update, uninstall, and manage device drivers. It's a central location for driver management.
- **System Preferences (macOS)**: macOS users can manage drivers through System Preferences. For example, adding printers or configuring network adapters.
- **Package Managers (Linux)**: Linux distributions often use package managers to install and update drivers. Commands like **apt** (Debian/Ubuntu), **yum** (Red Hat/CentOS), or **pacman** (Arch Linux) can be used to manage drivers.

How Device Driver Handles a Request?

How a device driver handles a request in the operating system is as follows:

Suppose a request comes to read a block N. If the driver is idle when a request arrives, it starts carrying out the request immediately. Otherwise, if the driver is already busy with some other request, it places the new request in the queue of pending requests.

Applications of Device Drivers:

Because of the diversity of modern hardware and operating systems, drivers operate in many different environments. Device drivers may interface with Printers, Video adapters, Network cards, Sound cards, Local buses of various sorts, Image scanners, Digital cameras, Digital terrestrial television tuners, IrDA adapters, and Implementing support for different file systems. It also interfaces with:

- Low-bandwidthI/O buses of various sorts for pointing devices such as mice, keyboards, etc.
- Computer storagedevices such as hard disk, CD-ROM, and floppy disk buses (ATA, SATA, SCSI, SAS)
- The radio-frequency communication transceiver adapters are used for short-distance and low-rate wireless communication in home automation, such as Bluetooth Low Energy (BLE), Thread, ZigBee, and Z-Wave).

Choosing and installing the correct device drivers for given hardware is often a key component of computer system configuration. Common levels of abstraction for device drivers include:

1. For hardware:

- Interfacing directly
- Writing to or reading from a device control register
- Using some higher-level interface (e.g. Video BIOS)
- Using another lower-level device driver (e.g. file system drivers using disk drivers)
- Simulating work with hardware while doing something entirely different.

2. For software:

- Allowing the operating system direct access to hardware resources
- Implementing only primitives
- Implementing an interface for non-driver software (e.g. TWAIN)
- Implementing a language, sometimes quite high-level (e.g. PostScript)

Device Handling:

Device handling in an operating system involves the management of hardware devices connected to the system. This includes tasks such as allocation, detection, configuration, control, and monitoring of devices to ensure they function properly.

Device Allocation:

Device allocation is the art of distributing and managing resources in a computer system, a critical task for efficient operation. It comes in various flavors, with static and dynamic allocation methods and the choice between contiguous and non-contiguous allocation:

Static Allocation: Static allocation is like assigning dedicated parking spaces. Resources are fixed and pre-allocated, making them always available for specific devices. This method is simple and predictable but can be inefficient if devices don't consistently utilize their allocated resources.

Dynamic Allocation: Dynamic allocation is akin to flexible office space. Resources are assigned as needed, optimizing utilization. It's more versatile but may require extra management and can lead to resource contention in heavily loaded systems.

Contiguous and Non-contiguous Allocation: Contiguous allocation groups resources together, providing a single, uninterrupted chunk. Non-contiguous allocation scatters resources across the system. Contiguous is efficient but limiting, while non-contiguous offers flexibility at the cost of fragmentation.

I/O Request Handling:

I/O request handling keeps your computer's data flowing seamlessly. It involves managing input and output requests, queuing, and scheduling algorithms, as well as buffering and caching techniques:

- **Input and Output Requests:** These requests are the data exchanges between your software and hardware. Handling them efficiently is crucial for system performance. These digital couriers are the messages between your software and hardware. Input requests deliver commands to hardware (e.g., keystrokes), while output requests bring results back to software (e.g., displaying text). They are the data bridge connecting your digital world with the physical one.
- **Queuing and Scheduling Algorithms:** Queuing algorithms prioritize incoming I/O requests, while scheduling algorithms determine the order in which they're executed, optimizing data flow. They are essential for multitasking and resource allocation, ensuring your computer juggles numerous requests efficiently, just like a well-organized traffic system keeps vehicles moving seamlessly. These algorithms

play a vital role in maintaining order and optimizing system performance, ensuring that critical tasks get the green light while preventing gridlock in the digital highway.

- **Buffering and Caching:** Buffering and caching techniques temporarily store data in high-speed memory to reduce data transfer bottlenecks and enhance overall system responsiveness. Buffering temporarily stores data in high-speed memory, bridging the gap between slow and fast components and preventing data bottlenecks. Caching, on the other hand, stores frequently accessed data for quick retrieval, reducing the need to fetch it from slower sources. Together, they create a dynamic duo that accelerates data transfer, reducing wait times and providing a smoother user experience. These techniques are like having quick-access notes handy, saving you from having to retrieve information from your bookshelf repeatedly.

Device Initialization and Configuration:

Device initialization is like the curtain rising on a tech performance. It prepares hardware for action during system startup. Configuration, the fine-tuning, ensures devices work smoothly with the operating system. Modern plug-and-play systems make this a seamless, user-friendly process, simplifying the tech symphony.

- **Bootstrapping Devices:** They initiate the essential processes, load necessary drivers, and set the stage for all other hardware and software components to come alive. Bootstrapping devices are the foundation upon which the entire system is built, like the ship's captain guiding it through uncharted territory.
- **Configuration Processes:** During these processes, system settings are fine-tuned to optimize device interactions with the operating system. It's like a conductor adjusting the instruments to create a harmonious performance. These processes encompass various tasks, such as setting display resolutions, configuring network connections, and tailoring device behavior.
- **Plug and Play Systems:** These systems make the once-complicated task of installing and configuring hardware a breeze. They automatically detect and identify newly connected devices, configure their settings, and ensure they are ready for immediate use, all without requiring a system reboot. PnP systems have revolutionized the tech experience, making it easy for users to expand and enhance their hardware without needing technical expertise. These systems are the ultimate facilitators of effortless hardware integration, turning the once daunting task of device configuration into a user-friendly and seamless process, bringing true plug-and-play convenience to technology.

Device Policies and Access Control:

Device policies and access control are the conductors, ensuring every instrument plays its part harmoniously. These fundamental components steer the course of resource allocation, access control, and device sharing.

- **Resource Allocation Policies:** Think of resource allocation policies as the architects of a balanced orchestra. They manage the distribution of system resources to devices, preventing a single overzealous instrument from drowning out the others. These policies establish rules that guide allocating CPU time, **memory**, and bandwidth, promoting fair and efficient resource utilization.
- **Access Control Mechanisms:** Like vigilant gatekeepers, they determine who has the privilege to interact with specific devices. These mechanisms protect against unauthorized access, preserving system integrity and **data security**. They establish user or application permissions, ensuring that only those with the proper credentials can access devices, thereby preventing unauthorized tampering or data breaches.
- **Device Sharing:** Device sharing is the choreographer that enables multiple musicians to share the stage simultaneously. This strategy allows several users or processes to access and utilize devices concurrently, like an ensemble of musicians collaborating in perfect sync. Device sharing optimizes resource utilization and promotes a collaborative environment, facilitating a dynamic and efficient digital experience.

Disk Scheduling Algorithms:

"**Disk Scheduling Algorithms**" in an operating system can be referred to as a manager of a grocery store that manages all the incoming and outgoing requests for goods of that store. He keeps a record of what is available in-store, what we need further, and manages the timetable of transaction of goods. Disk scheduling is also known as **I/O Scheduling**.

The 'Disk Structure in OS' is made in such a way that there are many layers of storage blocks on the disk. When we need to access these data from disk, we initialize a 'request' to the system to give us the required data. These requests are done on a large scale. So, there is a large number of input and output requests coming to the disk.

The operating system manages the timetable of all these requests in a proper algorithm. This is called a "Disk Scheduling Algorithm in OS". This algorithm helps OS to maintain an efficient manner in which input-output requests can be managed to execute a process. It manages a proper order to deal with sequential requests to the disk and provide the required data.

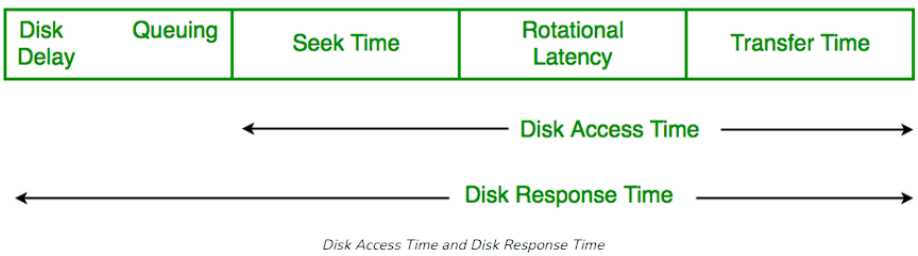
Since multiple requests are approaching the disk, this algorithm also manages the upcoming requests of the future and does a lining up of requests. So, it is also known as the "**Request Scheduling Algorithm**".

Importance of Disk Scheduling:

- Multiple I/O requests may arrive by different processes and only one I/O request can be served at a time by the disk controller. Thus, other I/O requests need to wait in the waiting queue and need to be scheduled.
- Two or more requests may be far from each other so this can result in greater disk arm movement.
- Hard drives are one of the slowest parts of the computer system and thus need to be accessed in an efficient manner.

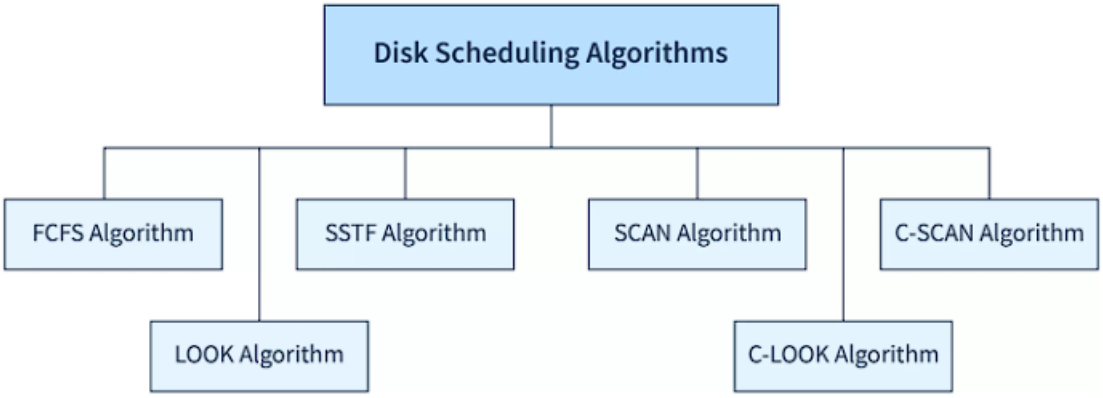
Key Terms Associated with Disk Scheduling:

1. **Seek Time**: As we know, the data may be stored on various blocks of disk. To access these data according to the request, the disk arm moves and finds the required block. The time taken by the arm in doing this search is known as "Seek Time".
2. **Rotational Latency**: The required data block needs to move at a particular position from where the read/write head can fetch the data. So, the time taken in this movement is known as "Rotational Latency". This rotational time should be as less as possible so, the algorithm that will take less time to rotate will be considered a better algorithm.
3. **Transfer Time**: When a request is made from the user side, it takes some time to fetch these data and provide them as output. This taken time is known as "Transfer Time".
4. **Disk Access Time**: It is defined as the total time taken by all the above processes. ***Disk access time = (seek time + rotational latency time + transfer time)***
5. **Disk Response Time**: The disk processes one request at a single time. So, the other requests wait in a queue to finish the ongoing process of request. The average of this waiting time is called "Disk Response Time".
6. **Starvation**: Starvation is defined as the situation in which a low-priority job keeps waiting for a long time to be executed. The system keeps sending high-priority jobs to the disk scheduler to execute first.
7. **Total Seek Time = Total head Movement * Seek Time**



Disk Access Time and Disk Response Time

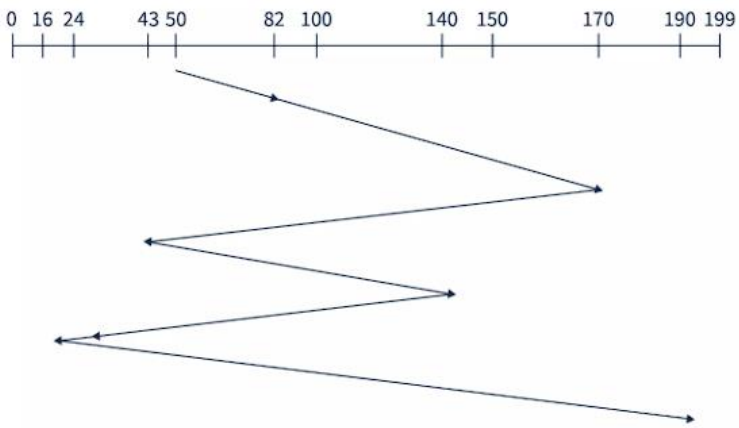
Types of Disk Scheduling Algorithm:



1. FCFS Disk Scheduling Algorithm:

It stands for **'first-come-first-serve'**. As the name suggests, the request that comes first will be processed first and so on. The requests coming to the disk are arranged in a proper sequence as they arrive. Since every request is processed in this algorithm, so there is no chance of 'starvation'.

Example: Suppose a disk having 200 tracks (0-199). The request sequence (82,170,43,140,24,16,190) of the disk is shown as in the given figure and the head start is at request 50.



Explanation: In the above image, we can see the head starts at position 50 and moves to request 82. After serving them the disk arm moves towards the second request which is 170 and then to the request 43 and so on. In this algorithm, the disk arm will serve the requests in arriving order. In this way, all the requests are served in arriving order until the process executes.

Suppose the order of request is- (82,170,43,140,24,16,190)

And current position of Read/Write head is: 50

So, total overhead movement (total distance covered by the disk arm) =

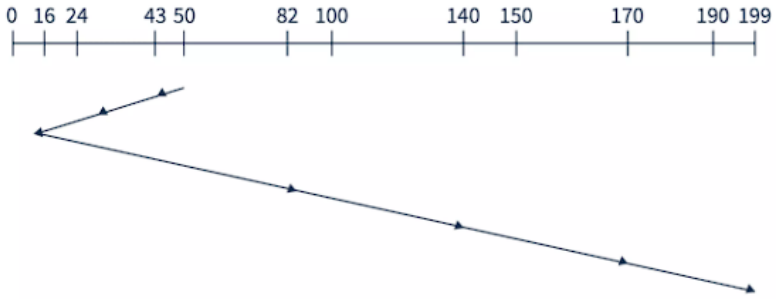
(82-50) + (170-82) + (170-43) + (140-43) + (140-24) + (24-16) + (190-16) =642

- **Advantages**:
    1. Implementation is easy.
    2. No chance of starvation.
- **Disadvantages**:
    1. 'Seek time' increases.
    2. Not so efficient.

2. SSTF Disk Scheduling Algorithm:

It stands for **'Shortest seek time first'**. As the name suggests, it searches for the request having the least 'seek time' and executes them first. This algorithm has less 'seek time' as compared to the FCFS Algorithm.

Example: Suppose a disk has 200 tracks (0-199). The request sequence (82,170,43,140,24,16,190) are shown in the given figure and the head position is at 50.



Explanation: The disk arm searches for the request which will have the least difference in head movement. So, the least difference is (50-43). Here the difference is not about the shortest value but it is about the shortest time the head will take to reach the nearest next request. So, after 43, the head will be nearest to 24, and from here the head will be nearest to request 16, After 16, the nearest request is 82, so the disk arm will move to serve to request 82 and so on.

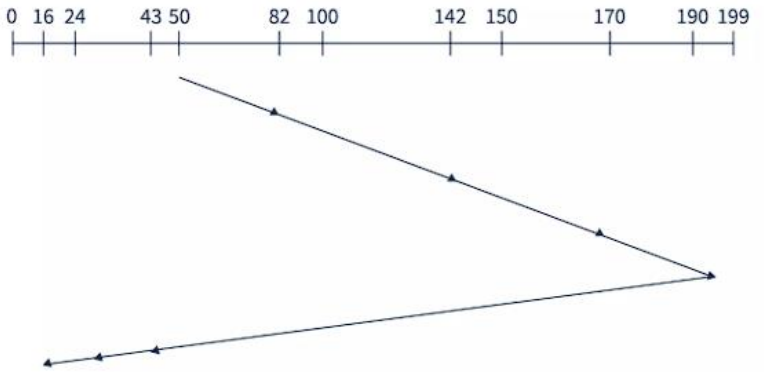
Hence, Calculation of Seek Time = (50-43) + (43-24) + (24-16) + (82-16) + (140-82) + (170-140) + (190-170) = 208

- **Advantages**:
    1. In this algorithm, disk response time is less.
    2. More efficient than FCFS.
- **Disadvantages**:
    1. Less speed of algorithm execution.
    2. Starvation can be seen.

3. SCAN Disk Scheduling Algorithm:

In this algorithm, the head starts to scan all the requests in a direction and reaches the end of the disk. After that, it reverses its direction and starts to scan again the requests in its path and serves them. Due to this feature, this algorithm is also known as the **"Elevator Algorithm".**

Example: Suppose a disk has 200 tracks (0-199). The request sequence (82,170,43,140,24,16,190) is shown in the given figure and the head position is at 50. The 'disk arm' will first move to the larger values.



Explanation: In the above image, we can see that the disk arm starts from position 50 and goes in a single direction until it reaches the end of the disk i.e.- request position 199. After that, it reverses and starts servicing in the opposite direction until reaches the other end of the disk. This process keeps going on until the process is executed.

Hence, the Calculation of 'Seek Time' will be like: (199-50) + (199-16) =332

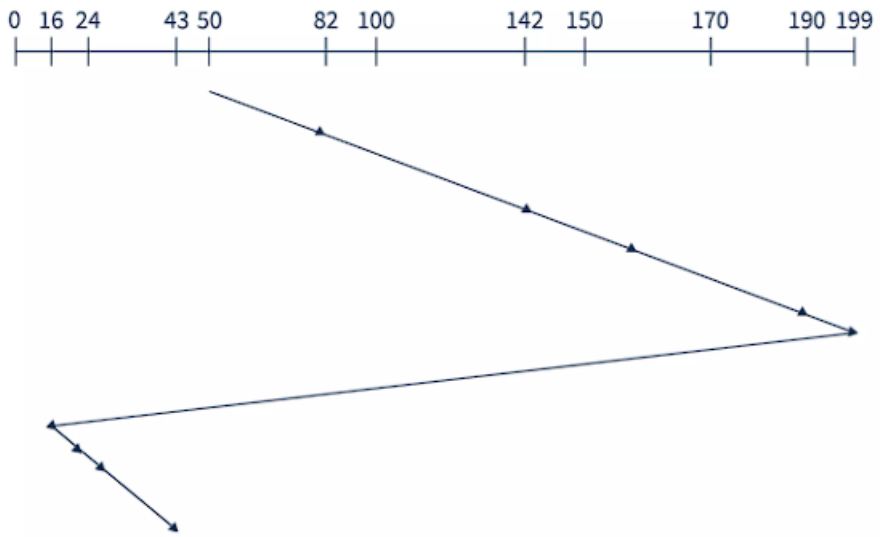
- **Advantages**:
    1. Implementation is easy.
    2. Requests do not have to wait in a queue.
- **Disadvantage**:
    1. The head keeps going on to the end even if there are no requests in that direction.

4. C-SCAN Disk Scheduling Algorithm:

It stands for "**Circular-Scan**". This algorithm is almost the same as the Scan disk algorithm but one thing that makes it different is that 'after reaching the one end and reversing the head direction, it starts to come back. The disk arm moves toward the end of the disk and serves the requests coming into its path.

After reaching the end of the disk it reverses its direction and again starts to move to the other end of the disk but while going back it does not serve any requests.

Example: Suppose a disk having 200 tracks (0-199). The request sequence (82,170,43,140,24,16,190) are shown in the given figure and the head position is at 50.



Explanation: In the above figure, the disk arm starts from position 50 and reached the end (199), and serves all the requests in the path. Then it reverses the direction and moves to the other end of the disk i.e.- 0 without serving any task in the path.

After reaching 0, it will again go move towards the largest remaining value which is 43. So, the head will start from 0 and moves to request 43 serving all the requests coming in the path. And this process keeps going.

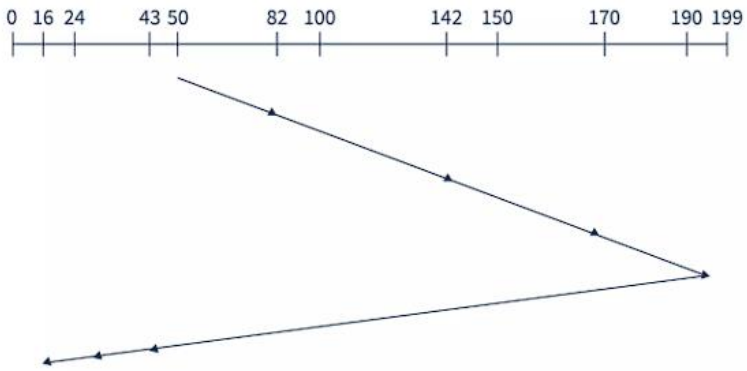
Hence, Seek Time will be = (199−50) + (199−0) + (43−0) = 391

- **Advantages**:
    1. The waiting time is uniformly distributed among the requests.
    2. Response time is good in it.
- **Disadvantages**:
    1. The time taken by the disk arm to locate a spot is increased here.
    2. The head keeps going to the end of the disk.

5. LOOK Disk Scheduling Algorithm:

In this algorithm, the disk arm moves to the 'last request' present and services them. After reaching the last requests, it reverses its direction and again comes back to the starting point. It does not go to the end of the disk, in spite, it goes to the end of requests.

Example a disk having 200 tracks (0-199). The request sequence (82,170,43,140,24,16,190) are shown in the given figure and the head position is at 50.



Explanation: The disk arm is starting from 50 and starts to serve requests in one direction only but in spite of going to the end of the disk, it goes to the end of requests i.e.-190. Then comes back to the last request of other ends of the disk and serves them. And again, starts from here and serves till the last request of the first side.

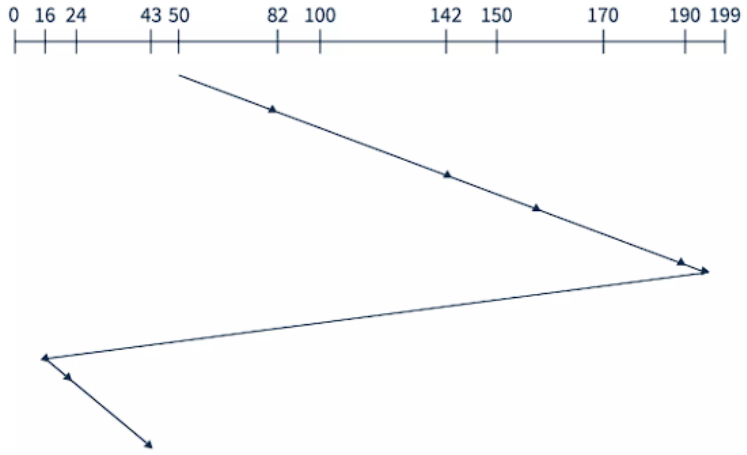
Hence, **Seek time** = (190-50) + (190-16) = 314

- **Advantages**:
    1. Starvation does not occur.
    2. Since the head does not go to the end of the disk, the time is not wasted here.
- **Disadvantage**:
    1. The arm has to be conscious to find the last request.

6. C-LOOK Disk Scheduling Algorithm:

The C-Look algorithm is almost the same as the Look algorithm. The only difference is that after reaching the end requests, it reverses the direction of the head and starts moving to the initial position. But in moving back, it does not serve any requests.

Example: Suppose a disk having 200 tracks (0-199). The request sequence (82,170,43,140,24,16,190) are shown in the given figure and the head position is at 50.



Explanation: The disk arm starts from 50 and starts to serve requests in one direction only but in spite of going to the end of the disk, it goes to the end of requests i.e.-190. Then comes back to the last request of other ends of a disk without serving them. And again, starts from the other end of the disk and serves requests of its path.

Hence, Seek Time = (190−50) + (190−16) + (43−16) = 341

- **Advantages**:
    1. The waiting time is decreased.
    2. If there are no requests till the end, it reverses the head direction immediately.
    3. Starvation does not occur.
    4. The time taken by the disk arm to find the desired spot is less.
- **Disadvantage**:
    1. The arm has to be conscious about finding the last request.

7. RSS (Random Scheduling):

It stands for Random Scheduling and just like its name it is natural. It is used in situations where scheduling involves random attributes such as random processing time, random due dates, random weights, and stochastic machine breakdowns this algorithm sits perfectly. Which is why it is usually used for analysis and simulation.

8. LIFO (Last-In First-Out):

In LIFO (Last In, First Out) algorithm, the newest jobs are serviced before the existing ones i.e. in order of requests that get serviced the job that is newest or last entered is serviced first, and then the rest in the same order.

Advantages of LIFO (Last-In First-Out):

Here are some of the advantages of the Last In First Out Algorithm.

- Maximizes locality and resource utilization
- Can seem a little unfair to other requests and if new requests keep coming in, it cause starvation to the old and existing ones.

9. N-STEP SCAN:

It is also known as the **N-STEP LOOK** algorithm. In this, a buffer is created for N requests. All requests belonging to a buffer will be serviced in one go. Also once the buffer is full no new requests are kept in this buffer and are sent to another one. Now, when these N requests are serviced, the time comes for another top N request and this way all get requests to get a guaranteed service

Advantages of N-STEP SCAN:

Here are some of the advantages of the N-Step Algorithm.

- It eliminates the starvation of requests completely

10. F-SCAN:

This algorithm uses two sub-queues. During the scan, all requests in the first queue are serviced and the new incoming requests are added to the second queue. All new requests are kept on halt until the existing requests in the first queue are serviced.

Advantages of F-SCAN:

Here are some of the advantages of the F-SCAN Algorithm.

- F-SCAN along with N-Step-SCAN prevents “arm stickiness” (phenomena in I/O scheduling where the scheduling algorithm continues to service requests at or near the current sector and thus prevents any seeking)

Each algorithm is unique in its own way. Overall Performance depends on the number and type of requests.

Note: Average Rotational latency is generally taken as 1/2(Rotational latency).

Selecting a Disk-Scheduling Algorithm:

Having seen a number of disk scheduling algorithms, we now see which algorithm performs the best. SSTF is common and has a natural appeal because it increases performance over FCFS. SCAN and C-SCAN perform better for systems that place a heavy load on the disk – because less starvation. When there is heavy load in the disk, if SSTF is used, it will continue to serve the requests close to the current position of the disk. The requests that are farther away from the current position will starve.

The performance of the algorithms depends on the number and types of requests also. Suppose there is only one outstanding request. Then all the algorithms behave like FCFS.

The requests for disk service can also be influenced by the file-allocation method. For a contiguously allocated file, the contents of the file are stored in contiguous disk blocks and therefore, there is limited head movement.

For linked or indexed file, the contents of a file can be kept in any disk block in the disk. This may need greater head movement when compared to contiguous allocation.

The location of directories and index blocks is also important. The directories have the name of the file and the details present in the file control block (inode in the case of UNIX/Linux). The file control block has the addresses of the disk blocks where the contents of the file are stored. To open a file, it is necessary to search the directory structure, get the file control block, get the addresses of the disk blocks containing data, access the disk blocks and get the file's data. If directory entry is on the first cylinder and the file's data are on the final cylinder, greater head movement is needed. Therefore, caching directories and index blocks in main memory help to reduce disk-arm movement.

Disk-scheduling algorithm should be written as a separate module of the operating system, allowing it to be replaced with a different algorithm, if necessary.

Thus, it is seen that either SSTF or LOOK is a reasonable choice for the default algorithm. SSTF causes less head movement and LOOK works well when there is a heavy load.

Disk-scheduling algorithms discussed here consider only the seek distances. The rotational latency is not taken into account. For modern disks, rotational latency can also be as large as the seek time. Therefore, disk-scheduling algorithms are implemented in the controller hardware built into the disk drive. When the operating system sends requests, the controller queues the requests and schedules them to improve both the seek time and rotational latency.

Swap Space Management:

A computer has a sufficient amount of physical memory, but we need more, so we swap some memory on disk most of the time. **Swap space** is a space on a hard disk that is a substitute for physical memory. It is used as virtual memory, which contains process memory images. Whenever our computer runs short of physical memory, it uses its virtual memory and stores information in memory on a disk.

This interchange of data between virtual memory and real memory is called **swapping** and space on disk as swap space. Swap space helps the computer's operating system pretend that it has more RAM than it actually has. It is also called a **swap file**.

Virtual memory is a combination of RAM and disk space that running processes can use. **Swap space** is the **portion of virtual memory** on the hard disk, used when RAM is full. Swap space can be useful to computers in the following various ways, such as:

- It can be used as a single contiguous memory which reduces I/O operations to read or write a file.
- Applications that are not used or less used can be kept in a swap file.
- Having sufficient swap files helps the system keep some physical memory free all the time.
- The space in physical memory that has been freed due to swap space can be used by OS for other important tasks.

Operating systems such as Windows, Linux, etc. systems provide a certain amount of swap space by default which users can change according to their needs. If you don't want to use virtual memory, you can easily disable it together. Still, if you run out of memory, then the kernel will kill some of the processes to create a sufficient amount of space in physical memory so that it totally depends upon the user whether he wants to use swap space or not.

Swap-Space Management:

Swap-Space management is another low-level task of the operating system. Disk space is used as an extension of main memory by the virtual memory. As we know the fact that disk access is much slower than memory access, In the swap-space management we are using disk space, so it will significantly decreases system performance. Basically, in all our systems we require the best throughput, so the goal of this swap-space implementation is to provide the virtual memory the best throughput.

Swap-Space Use:

Swap-space is used by the different operating-system in various ways. The systems which are implementing swapping may use swap space to hold the entire process which may include image, code and data segments. Paging systems may simply store pages that have been pushed out of the main memory. The need of swap space on a system can vary from a megabytes to gigabytes but it also depends on the amount of physical memory, the virtual memory it is backing and the way in which it is using the virtual memory.

It is safer to overestimate than to underestimate the amount of swap space required, because if a system runs out of swap space it may be forced to abort the processes or may crash entirely. Overestimation wastes disk space that could otherwise be used for files, but it does not harm other.

Following table shows different system using amount of swap space:

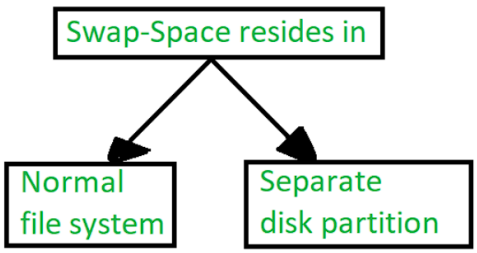
System	Swap-Space
1. Solaris	Equal amount of physical memory
2. Linux	Double the amount of physical memory

Explanation of above table:

Solaris, setting swap space equal to the amount by which virtual memory exceeds page-able physical memory. In the past Linux has suggested setting swap space to double the amount of physical memory. Today, this limitation is gone, and most Linux systems use considerably less swap space.

Including Linux, some operating systems; allow the use of multiple swap spaces, including both files and dedicated swap partitions. The swap spaces are placed on the disk so the load which is on the I/O by the paging and swapping will spread over the system's bandwidth.

Swap-Space Location:



Location of swap-space

A swap space can reside in one of the two places –

1. Normal file system
2. Separate disk partition

Let, if the swap-space is simply a large file within the file system. To create it, name it and allocate its space **normal file-system** routines can be used. This approach, through easy to implement, is inefficient. Navigating the directory structures and the disk-allocation data structures takes time and extra disk access. During reading or writing of a process image, **external fragmentation** can greatly increase swapping times by forcing multiple seeks.

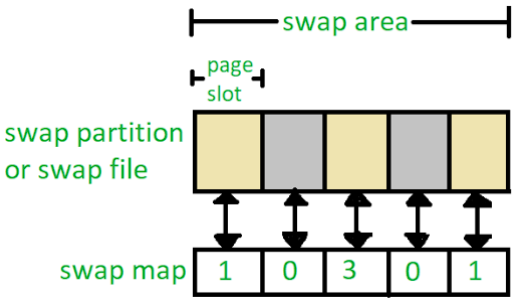
There is also an alternate to create the swap space which is in a separate **raw partition**. There is no presence of any file system in this place. Rather, a swap space storage manager is used to allocate and de-allocate the blocks. from the raw partition. It uses the algorithms for speed rather than storage efficiency, because we know the access time of swap space is shorter than the file system. By this **Internal fragmentation** increases, but it is acceptable, because the life span of the swap space is shorter than the files in the file system. Raw partition approach creates fixed amount of swap space in case of the **disk partitioning**.

Some operating systems are flexible and can swap both in raw partitions and in the file system space, example: **Linux**.

Swap-Space Management: An Example –

The traditional UNIX kernel started with an implementation of swapping that copied entire process between contiguous disk regions and memory. UNIX later evolve to a combination of swapping and paging as paging hardware became available. In Solaris, the designers changed standard UNIX methods to improve efficiency. More changes were made in later versions of Solaris, to improve the efficiency.

Linux is almost similar to Solaris system. In both the systems the swap space is used only for anonymous memory, it is that kind of memory which is not backed by any file. In the Linux system, one or more swap areas are allowed to be established. A swap area may be in either in a swap file on a regular file system or a dedicated file partition.



Data structure for swapping on Linux system

Each swap area consists of 4-KB **page slots**, which are used to hold the swapped pages. Associated with each swap area is a **swap-map-** an array of integers counters, each corresponding to a page slot in the swap area. If the value of the counter is 0 it means page slot is occupied by swapped page. The value of counter indicates the number of mappings to the swapped page. For example, a value 3 indicates that the swapped page is mapped to the 3 different processes.