Methods for Handling Deadlocks

We can deal with the deadlock problem in one of the three ways.

> We can use a protocol to prevent or avoid deadlock, ensuring that the system will never enter a deadlock state.
>
> We can allow the system to enter a deadlock state, detect it, and recover.
>
> We can ignore the problem altogether and pretend that deadlock never occur in the system.

To ensure that deadlocks never occur, the system can use either a deadlock prevention or a deadlock avoidance scheme. Possible strategies to deal with deadlocks:

Deadlock Prevention - Is a set of methods for ensuring that at least one of the necessary conditions cannot hold. These methods prevent deadlocks by constraining how requests for resources can be made.

Deadlock Avoidance- Deadlock Avoidance requires that the operating system be given in advance additional information concerning which resources a process will request and use during its lifetime. With this knowledge, we can decide for each request whether or not the process should wait.

Deadlock Detection and Recovery - Here the system can provide an algorithm that examines the state of the system to determine whether a deadlock has occurred and an algorithm to recover from the deadlock.

Deadlock Prevention

We can prevent the occurrence of a deadlock by ensuring that at least one of the four necessary conditions can not hold.

Mutual Exclusion - Mutual exclusion condition must hold for non-sharable resources. Sharable resources (e.g. a printer or a read only file) do not require mutually exclusive access, and thus cannot be involved in a deadlock. A process never needs to wait for a sharable resource. In general, we cannot prevent deadlocks by denying the mutual-exclusion condition, because some resources are intrinsically non-sharable.

Hold and Wait - When a process requests a resource, it does not hold any other resources. Two protocols can be used: Requires each process to request and be allocated all its resources before it begins execution. Allows a process to request resources only when the process has none. A process may request some resources and use them. However, it must release all the resources that is currently allocated before it can request any additional resources.

Disadvantages:

Resource utilization may be low, since resources may be allocated but unused for a long period.

Starvation is possible. A process that needs several popular resources may have to wait indefinitely, because at least one of the resources that it needs is always allocated to some other process.

No Preemption - If a process is holding some resources and requests another resource that cannot be immediately allocated to it, then all resources currently being held are preempted. The process will be restarted only when it can regain its old resources, as well as the new ones that is requesting**.** If a process requests some resources, we first check whether they are available. If they are, we allocate them. If they are not available, we check whether they are allocated to some other process that is waiting for additional resources. If so, we preempt the desired resources from the waiting process and allocate them to the requesting process. If the resources are neither available nor held by a waiting process, the requesting process must wait. While it is waiting, some of its resources may be preempted, but only if another process requests them. A process can be restarted only when it is allocated the new resources it is requesting and recovers any resources that were preempted while it was waiting.

Circular Wait - One way to ensure that circular wait never holds is to impose a total ordering of all resource types, and to require that each process requests resources in an increasing order of enumeration.

Each process can request resources only in an increasing order of enumeration. That is, a process can initially request any number of instances of a resource type, say Ri. After that,

the process can request instances of resource type Rj if and only if F(Rj) > F(Ri). Alternatively, whenever a process requests an instance of resource type Rj, it has released any resources Ri such that F(Ri) >= F(Rj). If these two protocols are used, then the circular wait condition cannot hold.

Deadlock prevention algorithms prevent deadlocks by restraining how request can be made. This ensures that at least one of the necessary conditions for deadlock cannot occur. Possible side effects of preventing deadlocks by these methods are low utilization and reduced system throughput.

Deadlock Avoidance

Deadlock avoidance requires that the operating system be given in advance additional information concerning which resources a process will request and use during its lifetime. With this knowledge, we can decide for each request whether or not the process should wait.

Given a priori information about the maximum number of resources of each type that may be requested for each process, it is possible to construct an algorithm that ensures the deadlock-avoidance approach**.**

A deadlock-avoidance algorithm dynamically examines the resource-allocation state to ensure that a circular wait condition can never exist. The resource-allocation state is defined by the number of available and allocated resources, and the maximum demands of the processes.

Safe state

A state is safe if the system can allocate resources to each process (up to its maximum) in some order and still avoid a deadlock. A system is in a safe state only if there exists a safe sequence.

A sequence of processes <P1, P2, …, Pn> is a safe sequence for the current allocation state if, for each Pi, the resources that Pi can still request can be satisfied by the currently available resources plus the resources held by all the Pj, with j < i.

In this situation the resources that process Pi needs are not immediately available, then Pi can wait until all Pj have finished. When they have finished, Pi can obtain all of its needed resources, complete its designated task, return its allocated resources and terminate. When Pi terminates, Pi+1 can obtain its needed resources and so on. If no sequence exists, then the system state is said to be unsafe.

Consider a system with 12 magnetic tape drives and 3 processes, P0, P1 and P2. Process P0 requires 10 tape drives, P1 requires4 tape drives and P2 requires 9 tape drives. Suppose that at time t0, process P0 is holding 5 tape drives, P1 is holding 2 tape drives and P2 is holding 2 tape drives. Thus there are 3 free tape drives.

	Maximum need	Current needs
P0	10	5
P1	4	2
P2	9	2

At time t0, the system is in a safe state. The sequence <P1,P0,P2> satisfies the safety condition. Process P1 can immediately be allocated all its tape drives and then return them (the system then will have 5 available tape drives), then process P0 can get all its tape drives and return them (then available – 10 tape drives), and finally process P2 could get all its tape drives.

A system may go from a safe state to an unsafe state. Suppose that at time T1, process P2 requests and is allocated 1 more tape drive. Then the system is no longer in a safe state, because at this point only process P1 can be allocated all its tape drives and process P0 and P2 have to wait – resulting in a deadlock.

Avoidance algorithms ensure that the system will always remain in a safe state. Initially, the system is in a safe state. Whenever a process requests a resource that is currently available, the system must decide whether the resource can be allocated immediately or whether the process must wait. The request is granted only if the allocation leaves the system in safe state. In this scheme, if a process requests a resource that is currently available, it may still have to wait. Thus, resource utilization may be lower than it would be without deadlock-avoidance algorithm.