Deadlock

A set of process is in a deadlock state if each process in the set is waiting for an event that can be caused by only another process in the set. In other words, each member of the set of deadlock processes is waiting for a resource that can be released only by a deadlock process. None of the processes can run, none of them can release any resources, and none of them can be awakened. It is important to note that the number of processes and the number and kind of resources possessed and requested are unimportant.

The resources may be either physical or logical. Examples of physical resources are Printers, Tape Drivers, Memory Space, and *CPU* Cycles. Examples of logical resources are Files, Semaphores, and Monitors.

The simplest example of deadlock is where process 1 has been allocated non-shareable resources *A*, say, a tap drive, and process 2 has be allocated non-sharable resource *B*, say, a printer. Now, if it turns out that process 1 needs resource *B* (printer) to proceed and process 2 needs resource *A* (the tape drive) to proceed and these are the only two processes in the system, each is blocked the other and all useful work in the system stops. This situation is termed deadlock. The system is in deadlock state because each process holds a resource being requested by the other process neither process is willing to release the resource it holds.

Preemptable and Nonpreemptable Resources

Resources come in two flavors: preemptable and nonpreemptable. A preemptable resource is one that can be taken away from the process with no ill effects. Memory is an example of a preemptable resource. On the other hand, a nonpreemptable resource is one that cannot be taken away from process (without causing ill effect). For example, *CD* resources are not preemptable at an arbitrary moment.

Reallocating resources can resolve deadlocks that involve preemptable resources. Deadlocks that involve nonpreemptable resources are difficult to deal with.

Necessary and Sufficient Deadlock Conditions

Coffman (1971) identified **four (4) conditions** that must hold simultaneously for there to be a deadlock.

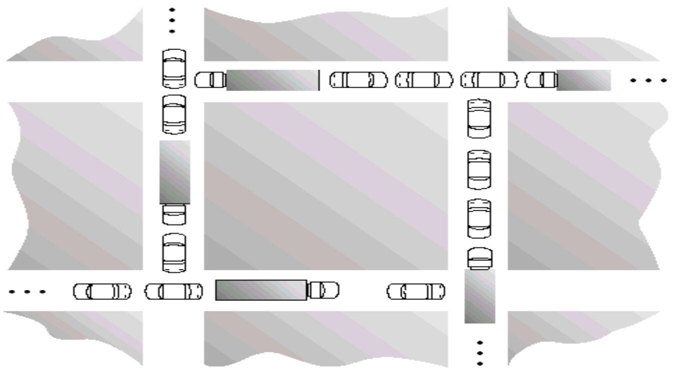
1. Mutual Exclusion Condition
The resources involved are non-shareable.
Explanation: At least one resource (thread) must be held in a non-shareable mode, that is, only one process at a time claims exclusive control of the resource. If another process requests that resource, the requesting process must be delayed until the resource has been released.

2. Hold and Wait Condition
Requesting process hold already, resources while waiting for requested resources.
Explanation: There must exist a process that is holding a resource already allocated to it while waiting for additional resource that are currently being held by other processes.

3. No-Preemptive Condition
Resources already allocated to a process cannot be preempted.
Explanation: Resources cannot be removed from the processes are used to completion or released voluntarily by the process holding it.

4. Circular Wait Condition
The processes in the system form a circular list or chain where each process in the list is waiting for a resource held by the next process in the list.

As an example, consider the traffic deadlock in the following figure



Consider each section of the street as a resource.

1. Mutual exclusion condition applies, since only one vehicle can be on a section of the street at a time.
2. Hold-and-wait condition applies, since each vehicle is occupying a section of the street, and waiting to move on to the next section of the street.
3. No-preemptive condition applies, since a section of the street that is a section of the street that is occupied by a vehicle cannot be taken away from it.
4. Circular wait condition applies, since each vehicle is waiting on the next vehicle to move. That is, each vehicle in the traffic is waiting for a section of street held by the next vehicle in the traffic.

The simple rule to avoid traffic deadlock is that a vehicle should only enter an intersection if it is assured that it will not have to stop inside the intersection.

It is not possible to have a deadlock involving only one single process. The deadlock involves a circular "hold-and-wait" condition between two or more processes, so "one" process cannot hold a resource, yet be waiting for another resource that it is holding. In addition, deadlock is not possible between two threads in a process, because it is the process that holds resources, not the thread that is, each thread has access to the resources held by the process.

Dealing with Deadlock Problem

In general, there are four strategies of dealing with deadlock problem:

1. **The Ostrich Approach**
   Just ignore the deadlock problem altogether.
2. **Deadlock Detection and Recovery**
   Detect deadlock and, when it occurs, take steps to recover.
3. **Deadlock Avoidance**
   Avoid deadlock by careful resource scheduling.
4. **Deadlock Prevention**
   Prevent deadlock by resource scheduling so as to negate at least one of the four conditions.

Deadlock Prevention

Havender in his pioneering work showed that since all four of the conditions are necessary for deadlock to occur, it follows that deadlock might be prevented by denying any one of the conditions.

Elimination of “Mutual Exclusion” Condition

The mutual exclusion condition must hold for non-sharable resources. That is, several processes cannot simultaneously share a single resource. This condition is difficult to eliminate because some resources, such as the tap drive and printer, are inherently non-shareable. Note that shareable resources like read-only-file do not require mutually exclusive access and thus cannot be involved in deadlock.

Elimination of “Hold and Wait” Condition

There are two possibilities for elimination of the second condition. The first alternative is that a process request be granted all of the resources it needs at once, prior to execution. The second alternative is to disallow a process from requesting resources whenever it has previously allocated resources. This strategy requires that all of the resources a process will need must be requested at once. The system must grant resources on “all or none” basis. If the complete set of resources needed by a process is not currently available, then the process must wait until the complete set is available. While the process waits, however, it may not hold any resources. Thus the “wait for” condition is denied and deadlocks simply cannot occur. This strategy can lead to serious waste of resources. For example, a program requiring ten tap drives must request and receive all ten derives before it begins executing. If the program needs only one tap drive to begin execution and then does not need the remaining tap drives for several hours. Then substantial computer resources (9 tape drives) will sit idle for several hours. This strategy can cause indefinite postponement (starvation). Since not all the required resources may become available at once.

Elimination of “No-preemption” Condition

The nonpreemption condition can be alleviated by forcing a process waiting for a resource that cannot immediately be allocated to relinquish all of its currently held resources, so that other processes may use them to finish. Suppose a system does allow processes to hold resources while requesting additional resources. Consider what happens when a request cannot be satisfied. A process holds resources a second process may need in order to proceed while second process may hold the resources needed by the first process. This is a deadlock. This strategy require that when a process that is holding some resources is denied a request for additional resources. The process must release its held resources and, if necessary, request them again together with additional resources. Implementation of this strategy denies the “no-preemptive” condition effectively.

High Cost When a process release resources the process may lose all its work to that point. One serious consequence of this strategy is the possibility of indefinite postponement (starvation). A process might be held off indefinitely as it repeatedly requests and releases the same resources.

Elimination of “Circular Wait” Condition

The last condition, the circular wait, can be denied by imposing a total ordering on all of the resource types and than forcing, all processes to request the resources in order (increasing or decreasing). This strategy impose a total ordering of all resources types, and to require that each process requests resources in a numerical order (increasing or decreasing) of enumeration. With this rule, the resource allocation graph can never have a cycle.

For example, provide a global numbering of all the resources, as shown

1	≡	Card reader
2	≡	Printer
3	≡	Plotter
4	≡	Tape drive
5	≡	Card punch

Now the rule is this: processes can request resources whenever they want to, but all requests must be made in numerical order. A process may request first printer and then a tape drive (order: 2, 4), but it may not request first a plotter and then a printer (order: 3, 2). The problem with this strategy is that it may be impossible to find an ordering that satisfies everyone.

Deadlock Avoidance

This approach to the deadlock problem anticipates deadlock before it actually occurs. This approach employs an algorithm to access the possibility that deadlock could occur and acting accordingly. This method differs from deadlock prevention, which guarantees that deadlock cannot occur by denying one of the necessary conditions of deadlock.

If the necessary conditions for a deadlock are in place, it is still possible to avoid deadlock by being careful when resources are allocated. Perhaps the most famous deadlock avoidance algorithm, due to Dijkstra [1965], is the Banker's algorithm. So named because the process is analogous to that used by a banker in deciding if a loan can be safely made.

Banker's Algorithm

In this analogy

Customers	≡	processes
Units	≡	resources, say, tape drive
Banker	≡	Operating System

Customers	**Used**	**Max**	
A	0	6	
B	0	5	Available Units
C	0	4	= 10
D	0	7	

Fig. 1

In the above figure, we see four customers each of whom has been granted a number of credit nits. The banker reserved only 10 units rather than 22 units to service them. At certain moment, the situation becomes

Customers	**Used**	**Max**	
A	1	6	
B	1	5	Available Units
C	2	4	= 2
D	4	7	

Fig. 2

Safe State The key to a state being safe is that there is at least one way for all users to finish. In other analogy, the state of figure 2 is safe because with 2 units left, the banker can delay any request except *C's*, thus letting *C* finish and release all four resources. With four units in hand, the banker can let either *D* or *B* have the necessary units and so on.

Unsafe State Consider what would happen if a request from B for one more unit were granted in above figure 2.

We would have following situation

Customers	**Used**	**Max**	
A	1	6	
B	2	5	Available Units
C	2	4	= 1
D	4	7	

Fig. 3

This is an unsafe state.

If all the customers namely *A, B, C,* and *D* asked for their maximum loans, then banker could not satisfy any of them and we would have a deadlock.

Important Note: It is important to note that an unsafe state does not imply the existence or even the eventual existence a deadlock. What an unsafe state does imply is simply that some unfortunate sequence of events might lead to a deadlock.

The Banker's algorithm is thus to consider each request as it occurs, and see if granting it leads to a safe state. If it does, the request is granted, otherwise, it postponed until later. Haberman [1969] has shown that executing of the algorithm has complexity proportional to N^2 where N is the number of processes and since the algorithm is executed each time a resource request occurs, the overhead is significant.

Deadlock Detection

Deadlock detection is the process of actually determining that a deadlock exists and identifying the processes and resources involved in the deadlock.
The basic idea is to check allocation against resource availability for all possible allocation sequences to determine if the system is in deadlocked state a. Of course, the deadlock detection algorithm is only half of this strategy. Once a deadlock is detected, there needs to be a way to recover several alternatives exists:

- Temporarily prevent resources from deadlocked processes.

- Back off a process to some check point allowing preemption of a needed resource and restarting the process at the checkpoint later.
- Successively kill processes until the system is deadlock free.

These methods are expensive in the sense that each iteration calls the detection algorithm until the system proves to be deadlock free. The complexity of algorithm is $O(N^2)$ where N is the number of proceeds. Another potential problem is starvation; same process killed repeatedly.

Recovery from Deadlock

1. **Recovery through preemption**

   Deadlock can be recovered by temporarily taking a resource away from its current owner and give it to another process. The ability to take a resource away from a process, have another process use it, and then give it back without the process noticing it is highly dependent on the nature of resource.

2. **Recovery through rollback**

   Processes are check pointed periodically. Check pointing would write the state of a process to a file so that it can be restarted later. The checkpoint contains the memory image and resource states. New checkpoints should not overwrite old ones but should be written to new files. When a deadlock is detected, the process that owns the needed resources is rolled back to a point in time before it acquired some other resources by starting one of its earlier check point.

3. Recovery through killing process

   It is the simplest way to break a deadlock. One possibility is killing a process in the cycle. If it doesn't help, it can be continued until the cycle is broken. Alternatively, a process not in the cycle can be chosen as a victim in order to release its resources.

   Eg. Process A holds a printer and wants a plotter and process B holds a plotter and wants a printer. These two are deadlocked. A third process C may hold another identical printer and plotter and be happily running. Killing the third process will release the resources and break the deadlock involving the first two.