Deadlocks

In a multiprogramming environment, a finite number of resources are distributed among several competing processes. A process requests resources: if the resources are not available at that time, the process enters a waiting state. Sometimes, a waiting process is never again able to change state, because the resources it has requested are held by other waiting processes. Similarly, while processes communicate with each other, for synchronizing and mutual exclusive access to shared resources, or improper use of communication primitives some processes are placed in a permanent wait state. This situation where a set of processes are in a permanent wait state is called a deadlock.

A set of processes is in a deadlocked state when every process in the set is waiting for an event (resource acquisition and release) that can be caused only by another process in the set. The resources may be either physical resources (for example, printers, tape drives, memory space, and CPU cycles) or logical resources (for example, semaphores, mutex locks, and files).

We describe a deadlock example involving the same type of resources:
A system with three CD RW drives, and each of three processes hold one of these CD RW drives. If each process now requests another drive, the three processes will be in a deadlocked state. Each is waiting for the event "CD RW is released," which can be caused only by one of the other waiting processes.
Another example of deadlock involving different resource types:
A system with one printer and one DVD drive. Suppose that process Pi is holding the DVD and process Pj is holding the printer. If Pi requests the printer and Pj requests the DVD drive, a deadlock occurs.

Necessary Conditions for Deadlock

In a deadlock, processes never finish executing, and system resources are tied up, preventing other jobs from starting. A deadlock situation can arise if the following four conditions hold simultaneously in a system:
Mutual Exclusion: At least one resource must be held in a non-sharable mode; that is, only one process at a time can use the resource. If another process requests that resource, the requesting process must be delayed until the resource has been released.
Hold and Wait: A process must be holding at least one resource and waiting to acquire additional resources that are currently being held by other processes.
No Preemption: Resources cannot be preempted; that is, a resource can be released only voluntarily by the process holding it, after that process has completed its work with it.
Circular Wait: A set {P0, P1, …, Pn} of waiting processes must exist such that P0 is waiting for a resource held by P1, P1 is waiting for a resource held by P2, …, Pn−1 is waiting for a resource held by Pn, and Pn is waiting for a resource held by P0.

Methods for Handling Deadlocks

We can handle the deadlock problem in one of three ways:

- To ensure that the system will never enter a deadlocked state, use a protocol to prevent or avoid deadlocks.
- Allow the system to enter a deadlocked state, detect it, and recover.
- Ignore the problem altogether and pretend that deadlocks never occur in the system.

Deadlock Prevention

Deadlocks can be prevented by using a set of methods to ensure that at least one of the necessary conditions of deadlock occurrence cannot hold. The methods to prevent deadlocks by constraining how requests for resources can be made.

Mutual Exclusion: The mutual exclusion condition must hold. We cannot prevent deadlocks by denying the mutual-exclusion condition, due to exclusive access to certain non-shareable resources.

Hold and Wait: To ensure that the hold-and-wait condition never occurs in the system, we must guarantee that, whenever a process requests a resource, it does not hold any other resources. One method is that each process is disciplined to request all required resources before it begins execution and if available are allocated. Another method allows a process to request resources only when it has none. A process may request some resources and use them. Before it can request any additional resources, it must release all the resources that it is currently allocated. We can further explain the two approaches using a simple example:

> A process that copies data from a DVD drive to a file on disk, sorts the file, and then prints the results to a printer.

In first method if all resources must be requested at the beginning of the process, then the process must initially request the DVD drive, disk file, and printer. It will hold the printer for its entire execution, even though it needs the printer only at the end.

The second method allows the process to request initially only the DVD drive and disk file. It copies from the DVD drive to the disk and then releases both the DVD drive and the disk file. The process must then request the disk file and the printer. After copying the disk file to the printer, it releases these two resources and terminates.

Both these methods have two main disadvantages. First, resource utilization may be low, since resources may be allocated but unused for a long period. Second, 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: To ensure that this condition does not hold, we can use the following protocol. If a process is holding some resources and requests another resource that cannot be immediately allocated to it (that is, the process must wait), then all resources the process is currently holding are preempted. In other words, these resources are implicitly released. The preempted resources are added to the list of resources for which the process

is waiting. The process will be restarted only when it can regain its old resources, as well as the new ones that it is requesting.
Alternatively, if a process requests some resources, we first check whether they are available. If they are, we allocate them. If they are not, 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. This protocol is often applied to resources whose state can be easily saved and restored later, such as CPU registers and memory space.
Circular Wait: One way to ensure that this condition 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.
If R = {R1, R2, ..., Rm} be the set of resource types. We assign to each resource type a unique integer number, which allows us to compare two resources and to determine whether one precedes another in our ordering. Formally, we define a one-to-one function F: R→N, where N is the set of natural numbers. For example, if the set of resource types R includes tape drives, disk drives, and printers, then the function F might be defined as follows:

F(tape drive) = 1
F(disk drive) = 3
F(printer) = 5

To deny circular wait condition, following protocol is used:
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). For example, using the function defined previously, a process that wants to use the tape drive and printer at the same time must first request the tape drive and then request the printer. Alternatively, we can require that a process requesting an instance of resource type Rj must have released any resources Ri such that F(Ri) ≥ F(Rj). Note also that if several instances of the same resource type are needed, a single request for all of them must be issued. If these two protocols are used, then the circular-wait condition cannot hold.
Deadlocks are prevented by limiting how requests can be made by process to ensure that at least one of the necessary conditions for deadlock cannot occur at the cost low device utilization and reduced system throughput.

Deadlock Avoidance

Deadlocks can be avoided by requiring additional information about how resources are to be requested by processes. For example, in a system with one tape drive and one printer, the system might need to know that process P will request first the tape drive and then the

printer before releasing both resources, whereas process Q will request first the printer and then the tape drive. With this knowledge of the complete sequence of requests and releases for each process, the system can decide for each request whether or not the process should wait in order to avoid a possible future deadlock. Each request requires that in making this decision the system consider the resources currently available, the resources currently allocated to each process, and the future requests and releases of each process.

The simplest and most useful model requires that each process declare the maximum number of resources of each type that it may need. Given this a priori information, it is possible to construct an algorithm that ensures that the system will never enter a deadlocked state. 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. More formally, 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 resource requests that Pi can still make can be satisfied by the currently available resources plus the resources held by all Pj, with $j < i$. In this situation, if the resources that 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 such sequence exists, then the system state is said to be unsafe.

A safe state is not a deadlocked state. Conversely, a deadlocked state is an unsafe state. Not all unsafe states are deadlocks, however an unsafe state may lead to a deadlock. As long as the state is safe, the operating system can avoid unsafe (and deadlocked) states.

We can further explain by considering a system with ten magnetic tape drives and three processes: P0, P1, and P2. Process P0 requires 8 tape drives, process P1 may need as many as four tape drives, and process P2 may need up to six tape drives.

Suppose that, at time t0, process P0 is holding four tape drives, process P1 is holding two tape drives, and process P2 is holding two tape drives. Thus, there are two free tape drives. At time t0, the system is in a safe state. The sequence <P1, P0, P2> and <P1, P2, P0> fulfills the safety condition. Process P1 can immediately be allocated all its tape drives and then return them (the system will then have four available tape drives); then process P0/P2 can get all its tape drives and return them (the system will then have eight available tape drives); and finally process P2/P0 can get all its tape drives and return them and the system will then have all ten tape drives available.

A system can go from a safe state to an unsafe state. Suppose that, at time t1, process P2 requests and is allocated one more tape drive. The system is no longer in a safe state, no sequence of processes fulfills safety condition. In this case to avoid the unsafe state, process P2 is not allocated the resource and is placed in a wait state. Only

P1's request to allocate resources (2 tapes) can be fulfilled, and it will return four tapes after run to completion. Now the system has six tapes and can be used to fulfill requirements of either process P0 or process P2.
In this approach 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 otherwise be.

Resource-Allocation-Graph Algorithm
Using the concept of a safe state, avoidance algorithms can be defined to ensure that the system will never deadlock. The idea is simply to 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 a safe state.

A system resource-allocation graph is a directed graph consisting of a set of vertices V and a set of edges E. The set of vertices V is partitioned into two different types of nodes: P = {P1, P2, ..., Pn}, the set consisting of all the active processes in the system, and R = {R1, R2, ..., Rm}, the set consisting of all resource types in the system. A directed edge from process Pi to resource type Rj is denoted by Pi → Rj ; it signifies that process Pi has requested an instance of resource type Rj and is currently waiting for that resource. A directed edge from resource type Rj to process Pi is denoted by Rj → Pi; it signifies that an instance of resource type Rj has been allocated to process Pi. A directed edge Pi → Rj is called a request edge; a directed edge Rj → Pi is called an assignment edge. When process Pi requests an instance of resource type Rj, a request edge is inserted in the resource-allocation graph. When this request can be fulfilled, the request edge is instantaneously transformed to an assignment edge. When the process no longer needs access to the resource, it releases the resource. As a result, the assignment edge is deleted.
Pictorially, we represent each process Pi as a circle and each resource type Rj as a rectangle. Since resource type Rj may have more than one instance, we represent each such instance as a dot within the rectangle A request edge points to only the rectangle Rj, whereas an assignment edge must also designate one of the dots in the rectangle.

In resource allocation graph by adding another type of edge, called a claim edge, it can be used to avoid unsafe sate with one instance of resources type. A claim edge Pi → Rj indicates that process Pi may request resource Rj at some time in the future. This edge resembles a request edge in direction but is represented in the graph by a dashed line. When process Pi requests resource Rj, the claim edge Pi → Rj is converted to a request edge. Similarly, when a resource Rj is released by Pi, the assignment edge Rj → Pi is reconverted to a claim edge Pi → Rj. The resources must be claimed in advance in the system. That is, before process Pi starts executing, all its claim edges must already appear in the resource-allocation graph. We can relax this condition by allowing a claim edge Pi → Rj to be added to the graph only if all the edges associated with process Pi are claim edges.

Now suppose that process Pi requests resource Rj. The request can be granted only if converting the request edge Pi → Rj to an assignment edge Rj → Pi does not result in the formation of a cycle in the resource-allocation graph. We check for safety by using a cycle-detection algorithm. If no cycle exists, then the allocation of the resource will leave the system in a safe state. If a cycle is found, then the allocation will put the system in an unsafe state. In that case, process Pi will have to wait for its requests to be satisfied.

We illustrate by using an example with two processes P1 and P2. Two resources R1 and R2, where R1 is allocated to P1, and P2 has requested for R1. P1 and P2 also indicated that may require R2, so we have claim edges P1 → R2, and P2 → R2 as shown in Figure 1.

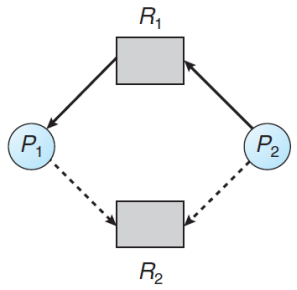
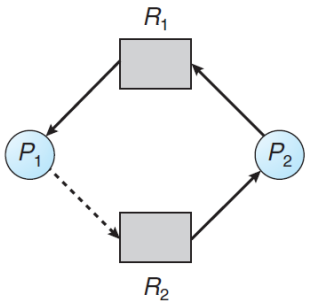


Figure 1: Resource allocation graph

Suppose that P2 requests R2. Although R2 is currently free, we cannot allocate it to P2, since this action will create a cycle (as mentioned, cycle indicates that the system is in an unsafe state) in the graph as shown in the following figure. If P1requests R2, and P2



and P2 requests R1, then a deadlock will occur. The resource-allocation-graph algorithm is not applicable to a resource allocation system with multiple instances of each resource type and mostly banker's algorithm is used.

Banker's Algorithm

When a new process enters the system, it must declare the maximum number of instances of each resource type that it may need. This number may not exceed the total number of resources in the system. When a user requests a set of resources, the system must determine whether the allocation of these resources will leave the system in a safe state. If it will, the resources are allocated; otherwise, the process must wait until some other process releases enough resources.

To encode the state of the resource-allocation system following data structures are needed, where ***n*** is the number of processes in the system and ***m*** is the number of resource types:

> ***Available:*** A vector of length ***m*** indicates the number of available resources of each type. If Available[j] equals k, then k instances of resource type Rj are available.
> ***Max:*** An *n* × *m* matrix defines the maximum demand of each process. If Max[i][j] equals k, then process Pi may request at most k instances of resource type Rj.
> ***Allocation:*** An n × m matrix defines the number of resources of each type currently allocated to each process. If Allocation[i][j] equals k, then process Pi is currently allocated k instances of resource type Rj.
> ***Need:*** An n × m matrix indicates the remaining resource need of each process. If Need[i][j] equals k, then process Pi may need k more instances of resource type Rj to complete its task. Note that Need[i][j] equals Max[i][j] − Allocation[i][j].

To simplify presentation of the banker's algorithm, we next establish following notation:

> Let X and Y be vectors of length n.
> We say that X ≤ Y if and only if X[i] ≤ Y[i] for all i = 1, 2, ..., n.
> For example, if X = (1,7,3,2) and Y = (0,3,2,1), then Y ≤ X.
> In addition, Y < X if Y ≤ X and Y≠ X.

We can treat each row in the matrices Allocation and Need as vectors and refer to them as Allocation_i and Need_i . The vector Allocation_i specifies the resources currently allocated to process Pi ; the vector Need_i specifies the additional resources that process Pi may still request to complete its task.

Safety Algorithm
To find out whether or not a system is in a safe state safety algorithm is described as follows:

1. Let Work and Finish be vectors of length m and n, respectively. Initialize
   Work = Available and Finish[i] = false for i = 0, 1, ..., n − 1.
2. Find an index i such that both
   a. Finish[i] == false
   b. $\text{Need}_i \le \text{Work}$
   If no such i exists, go to step 4.
3. Work = Work + Allocation_i
   Finish[i] = true
   Go to step 2.
4. If Finish[i] == true for all i, then the system is in a safe state.

Resource-Request Algorithm

To determining whether requests can be safely granted the algorithm is described as:
Let $Request_i$ be the request vector for process Pi. If $Request_i[j] == k$, then 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 $Request_i \leq Need_i$, go to step 2. Otherwise, raise an error condition, since the process has exceeded its maximum claim.
2. If $Request_i \leq Available$, go to 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:

$Available = Available – Request_i$;
$Allocation_i = Allocation_i + Request_i$;
$Need_i = Need_i – Request_i$;

If the resulting resource-allocation state is safe, the transaction is completed, and process Pi is allocated its resources. However, if the new state is unsafe, then Pi must wait for $Request_i$, and the old resource-allocation state is restored.

To illustrate the use of the banker's algorithm, consider a system with five processes P0 through P4 and three resource types A, B, and C. Resource type A has ten instances, resource type B has five instances, and resource type C has seven instances. Suppose that, at time T0, the following snapshot of the system has been taken:

	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	

The content of the matrix Need is defined to be Max − Allocation and is as follows:

	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

We claim that the system is currently in a safe state. Indeed, the sequence <P1, P3, P4, P2, P0> satisfies the safety criteria. Suppose now that process P1 requests one additional instance of resource type A and two instances of resource type C, so Request1 = (1,0,2). To decide whether this request can be immediately granted, we first check that $Request_1 \leq Available$ [(1,0,2) ≤ (3,3,2)], that is true. We then pretend that this request has been fulfilled, and we arrive at the following new state:

	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 execute our safety algorithm and find that the sequence <*P*1, *P*3, *P*4, *P*0, *P*2> satisfies the safety requirement. Hence, we can immediately grant the request of process *P*1.

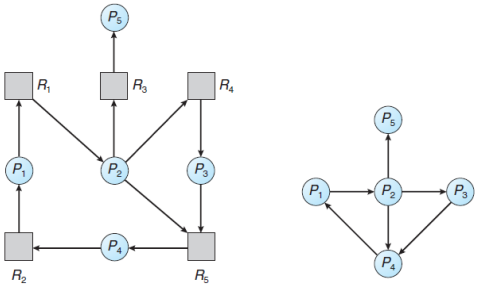
You should be able to see, however, that when the system is in this state, a request for (3,3,0) by P4 cannot be granted, since the resources are not available. Furthermore, a request for (0,2,0) by P0 cannot be granted, even though the resources are available, since the resulting state is unsafe.

Deadlock Detection

If a system does not employ either a deadlock-prevention or a deadlock avoidance algorithm, then a deadlock situation may occur. Deadlock detection strategies do not limit resource access or restrict process actions. In this environment, the system may provide an algorithm that examines the state of the system to determine whether a deadlock has occurred.

If a system has single instance of each resource type a variant of the resource-allocation graph, called a wait-for graph is used to detect deadlocks. Wait-for graph is obtained from the resource-allocation graph by removing the resource nodes and collapsing the appropriate edges.

More precisely, an edge from Pi to Pj in a wait-for graph implies that process Pi is waiting for process Pj to release a resource that Pi needs. An edge Pi → Pj exists in a wait-for graph if and only if the corresponding resource allocation graph contains two edges Pi → Rq and Rq → Pj for some resource Rq. A resource-allocation graph and the corresponding wait-for graph is depicted below:



Deadlock exists in the system if and only if the wait-for graph contains a cycle. To detect deadlocks, the system needs to maintain the wait-for graph and periodically invoke an algorithm that searches for a cycle in the graph.

If a system has multiple instances of each resource type, graph reduction technique can be applied on resource-allocation graph. In considering deadlock detection we often need to (conceptually) remove non-deadlocked processes from the system.

Graph Reduction: We can reduce a graph by removing a process ("reduce graph G by process pi") if we can show that process pi can continue execution.

A graph can be reduced by a process p if the process can acquire all the resource units it is requesting.

Reduction by a process pi simulates:

- the acquisition of all of pi's outstanding requests,
- the return of all units of reusable resources allocated to pi, and
- if pi is a producer of a consumable resource, the production of a "sufficient" number of units to satisfy all subsequent requests by consumers.

Obviously, if we can reduce a graph by all of its processes, there is no deadlock.

Deadlock Recovery

When a detection algorithm determines that a deadlock exists, system should recover from the deadlock. There are two options for breaking a deadlock:

- Simply to abort one or more processes to break the circular wait.
- To preempt some resources from one or more of the deadlocked processes.

Process Termination: To eliminate deadlocks by aborting a process, we use one of two methods. In both methods, the system reclaims all resources allocated to the terminated processes.

Abort all deadlocked processes: This method clearly will break the deadlock cycle, but at great expense. The deadlocked processes may have computed for a long time, and the results of these partial computations must be discarded and probably will have to be recomputed later.

Abort one process at a time until the deadlock cycle is eliminated: This method incurs considerable overhead, since after each process is aborted, a deadlock-detection algorithm must be invoked to determine whether any processes are still deadlocked.

Aborting a process may not be easy. If the process was in the midst of updating a file, terminating it will leave that file in an incorrect state. Similarly, if the process was in the midst of printing data on a printer, the system must reset the printer to a correct state before printing the next job.

Resource Preemption: To eliminate deadlocks using resource preemption, we successively preempt some resources from processes and give these resources to other processes until the deadlock cycle is broken. If preemption is required to deal with deadlocks, then three issues need to be addressed:
Selecting a Victim: Which resources and which processes are to be preempted? As in process termination, we must determine the order of preemption to minimize cost. Cost factors may include such parameters as the number of resources a deadlocked process is holding and the amount of time the process has thus far consumed.
Rollback: If we preempt a resource from a process, what should be done with that process? Clearly, it cannot continue with its normal execution; it is missing some needed resource. We must roll back the process to some safe state and restart it from that state. Since, in general, it is difficult to determine what a safe state is, the simplest solution is a total rollback: abort the process and then restart it. Although it is more effective to roll back the process only as far as necessary to break the deadlock, this method requires the system to keep more information about the state of all running processes.
Starvation: How do we ensure that starvation will not occur? That is, how can we guarantee that resources will not always be preempted from the same process?

In a system where victim selection is based primarily on cost factors, it may happen that the same process is always picked as a victim. As a result, this process never completes its designated task, a starvation situation any practical system must address. Clearly, we must ensure that a process can be picked as a victim only a (small) finite number of times. The most common solution is to include the number of rollbacks in the cost factor.