DEADLOCKS

A process requests resources; if the resources are not available at that time, the process enters a wait state. Waiting processes may never again change state, because the resources they have requested are held by other waiting processes. This situation is called a deadlock.

Deadlock problems can only become more common, given current trends, including larger numbers of processes, multithreaded programs, many more resources within a system, and an emphasis on long-lived file and database servers rather than batch systems.

SYSTEM MODEL

A system consists of a finite number of resources to be distributed among a number of competing processes. The resources are partitioned into several types, each of which consists of some number of identical instances.

Under the normal mode of operation, a process may utilize a resource in only the following sequence:

1. *Request:* If the request cannot be granted immediately (for example, if the resource is being used by another process), then the requesting process must wait until it can acquire the resource.

2. *Use:* The process can operate on the resource (for example, if the resource is a printer, the process can print on the printer).

3. *Release:* The process releases the resource.

Examples are the request and release device, open and close file, and allocate and free memory system calls.

A set of processes is in a deadlock state when every process in the set is waiting for an event that can be caused only by another process in the set.

DEADLOCK CHARACTERIZATION

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:

1. *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.

2. *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.

3. *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 task.

4. *Circular wait:* A set $\{P_0, P_1, ..., P_n\}$ of waiting processes must exist such that P_0 is waiting for a resource that is held by P_1, P_1 is waiting for a resource that is held by P_2, ..., P_{n-1} is waiting for a resource that is held by P_n, and P_n is waiting for a resource that is held by P_0.

RESOURCE ALLOCATION GRAPH

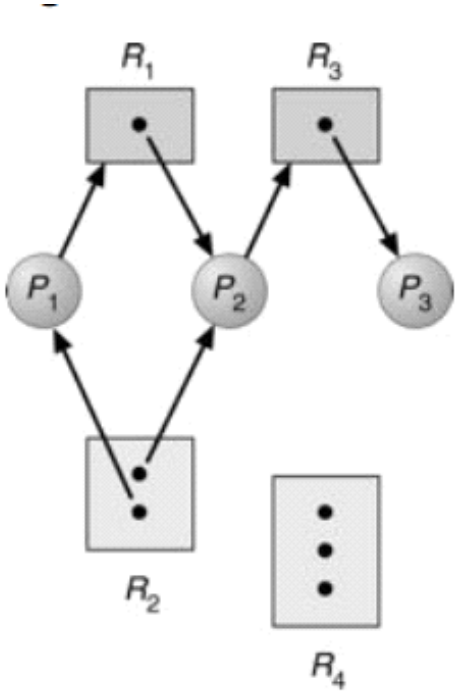
Deadlocks can be described in terms of directed graph called a system resource allocation graph which consists of set of vertices V and set of edges E.

V is partitioned into two types of nodes

1. $P=\{P_1,P_2,...P_n\}$ consisting active processes
2. $R=\{R_1,R_2,...R_m\}$ consisting all resource types.

A directed edge $P_i \rightarrow R_j$ is called a request edge;
A directed edge $R_j \rightarrow P_i$ is called an assignment edge.

Resource-allocation graph



3. The sets P, R, and E:

   $P = \{P_1, P_2, P_3\}$

   $R = \{R1, R2, R3, R4\}$

4. $E = \{P_1 \rightarrow R_1, P_2 \rightarrow R_3, R_1 \rightarrow P_2, R_2 \rightarrow P_2, R_2 \rightarrow P_1, R_3 \rightarrow P_3\}$

- Resource instances:
  - One instance of resource type R_1
  - Two instances of resource type R_2
  - One instance of resource type R_3
  - Three instances of resource type R_4

- Process states:

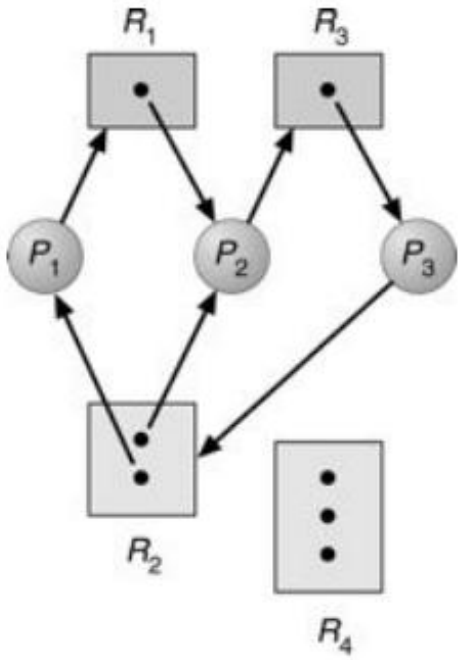
1. Process P_1 is holding an instance of resource type R_2 and is waiting for an instance of resource type R_1.
2. Process P_2 is holding an instance of R_1 and R_2 and is waiting for an instance of resource type R_3.
3. Process P_3 is holding an instance of R_3.

If each resource type has exactly one instance, then a cycle implies that a deadlock has occurred. If the cycle involves only a set of resource types, each of which has only a single instance, then a deadlock has occurred. Each process involved in the cycle is deadlocked. In this case, a cycle in the graph is both a necessary and a sufficient condition for the existence of deadlock.

If each resource type has several instances, then a cycle does not necessarily imply that a deadlock has occurred. In this case, a cycle in the graph is a necessary but not a sufficient condition for the existence of deadlock.

Suppose that process P_3 requests an instance of resource type R_2. Since no resource instance is currently available, a request edge $P_3 \rightarrow R_2$ is added to the graph

Resource-allocation graph with a deadlock



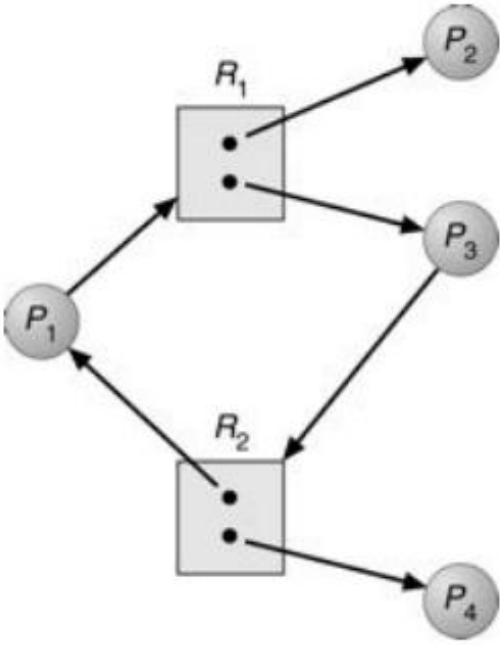
At this point, two minimal cycles exist in the system:

$$P_1 \rightarrow R_1 \rightarrow P_2 \rightarrow R_3 \rightarrow P_3 \rightarrow R_2 \rightarrow P_1$$

$$P_2 \rightarrow R_3 \rightarrow P_3 \rightarrow R_2 \rightarrow P_2$$

Processes P_1, P_2, and P_3 are deadlocked.

Resource-allocation graph with a cycle but no deadlock



we have a cycle

$$P_1 \rightarrow R_1 \rightarrow P_3 \rightarrow R_2 \rightarrow P_1$$

However, there is no deadlock. Observe that process P_4 may release its instance of resource type R_2. That resource can then be allocated to P_3, breaking the cycle.

If a resource-allocation graph does not have a cycle, then the system is not in a deadlocked state. If there is a cycle, then the system may or may not be in a deadlocked state.

Methods for handling deadlocks

1. We can use a protocol to prevent or avoid deadlocks, ensuring that the system will never enter a deadlock state.
2. We can allow the system to enter a deadlock state, detect it, and recover.
3. We can ignore the problem altogether and pretend that deadlocks never occur in the system. This solution is used by most operating systems including unix.

Deadlock prevention is a set of methods for ensuring that at least one of the necessary conditions cannot hold.

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.

DEADLOCK PREVENTION

By ensuring that at least one of these conditions cannot hold, we can prevent the occurrence of a deadlock.

MUTUAL EXCLUSION :

The mutual-exclusion condition must hold for non-sharable resources.

Sharable resources, in contrast, do not require mutually exclusive access and thus cannot be involved in a deadlock. Read-only files are a good example of a sharable resource .

We cannot prevent deadlocks by denying the mutual exclusion condition, because some resources are intrinsically non-sharable.

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 protocol that can be used requires each process to request and be allocated all its resources before it begins execution. An alternative protocol allows a process to request resources only when it has none. Both these protocols 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.

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 currently being held are preempted.

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. We define a one-to-one function F: R → N.

F(tape drive) = 1

F(disk drive) = 5

F(printer) = 12

We can now consider the following protocol to prevent deadlocks: 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. The process can request instances of resource type Rj if and only if $F(R_j) > F(R_i)$.

Alternatively, we can require that, whenever a process requests an instance of resource type R_j, it has released any resources R_i such that $F(R_i) \geq F(R_j)$.

DEADLOCK AVOIDANCE

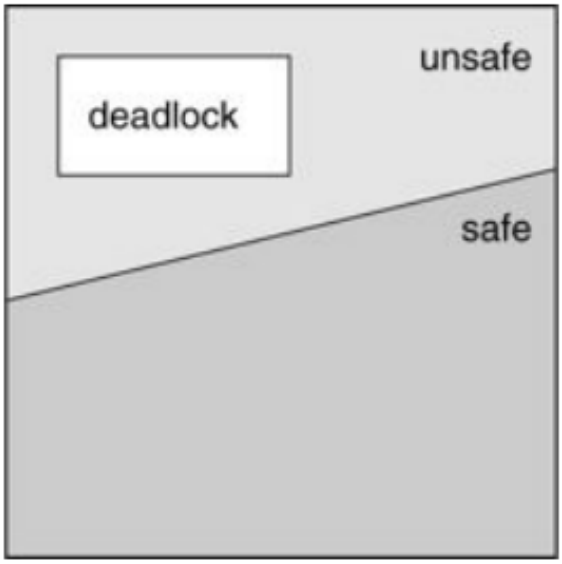
An alternative method for avoiding deadlocks is to require additional information about how resources are to be requested. With the knowledge of the complete sequence of requests and releases for each process, we can decide for each request whether or not the process should wait in order to avoid a possible 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. Such an algorithm defines the deadlock-avoidance approach

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.

Safe, unsafe, and deadlock state spaces



An unsafe state may lead to a deadlock. As long as the state is safe, the operating system can avoid unsafe (and deadlocked) states. In an unsafe state, the operating system cannot prevent processes from requesting resources such that a deadlock occurs

Example

we consider a system with 12 magnetic tape drives and 3 processes: P_0,P_1, and P_2. Process P_0 requires 10 tape drives, process P_1 may need as many as 4,and process P_2 may need up to 9 tape drives. Suppose that, at time t_0, process P_0 is holding 5 tape drives, process P_1 is holding 2, and process P2 is holding 2 tape drives.

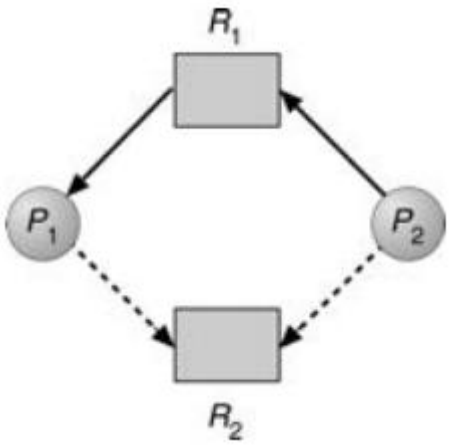
	Maximum Needs	Current Needs
P_0	10	5
P_1	4	2
P_2	9	2

At time t0, the system is in a safe state. The sequence <P_1, P_0, P_2> satisfies the safety condition.

RESOURCE ALLOCATION GRAPH

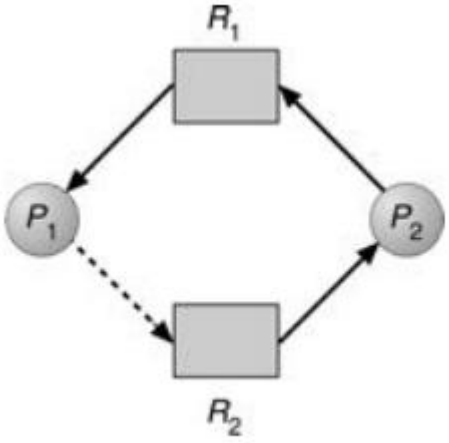
In addition to the request and assignment edges, we introduce a new type of edge, called a claim edge. A claim edge $P_i \rightarrow R_j$ indicates that process P_i may request resource R_j at some time in the future. This edge resembles a request edge in direction but is represented by a dashed line.

When process P_i requests resource R_j, the claim edge $P_i \rightarrow R_j$ is converted to a request edge. Similarly, when a resource R_j is released by P_i, the assignment edge $R_j \rightarrow P_i$ is reconverted to a claim edge $P_i \rightarrow R_j$



Suppose that process P_i requests resource R_j. The request can be granted only if converting the request edge $P_i \rightarrow R_j$ to an assignment edge $R_j \rightarrow P_i$ does not result in the formation of a cycle in the resource-allocation graph. Note that 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.



A cycle indicates that the system is in an unsafe state. If P_1 requests R_2, and P_2 requests R_1, then a deadlock will occur.

BANKER'S ALGORITHM

The deadlock-avoidance algorithm that we describe is less efficient than the resource-allocation graph scheme. This algorithm is commonly known as the banker's algorithm.

The name was chosen because the algorithm could be used in a banking system to ensure that the bank never allocates its available cash such that it can no longer satisfy the needs of all its customers.

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.

Let n be the number of processes in the system and m be the number of resource types. We need the following data structures:

1. *Available:* A vector of length m indicates the number of available resources of each type. If Available[j] equals k, there are k instances of resource type Rj available.
2. *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.
3. *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.
4. *Need:* An n × m matrix indicates the remaining resource need of each process. If Need[i][j] equals k, then process P_i 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].

SAFETY ALGORITHM

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 i such that both

   a. Finish[i] == false

   b. Needi ≤ Work

   If no such i exists, go to step 4.

3. Work = Work + $Allocation_i$

   Finish[i] = true

   Go to step 2.

5. If Finish[i] == true for all i, then the system is in a safe state.

RESOURCE REQUEST ALGORITHM

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

1. If $Request_i \leq Needi$, 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, P_i 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:

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

An Illustrative Example

	Allocation	Max	Available
	A B C	A B C	A B C
Po	0 1 0	7 5 3	3 3 2
P1	2 0 0	3 2 2	
P2	3 0 2	9 0 2	
P3	2 1 1	2 2 2	
p4	0 0 2	4 3 3	

The content of the matrix Need is defined to be Max - Allocation and is

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

We claim that the system is currently in a safe state. Indeed, the sequence <P_1, P_3, P_4, P_2, P_0> satisfies the safety criteria. Suppose now that process P_1 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 $Request_1$ ≤ Available—that is, (1,0,2) ≤ (3,3,2), which 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
Po	0 1 0	7 4 3	2 3 0
P1	3 0 2	0 2 0	
P2	3 0 2	6 0 0	
P3	2 1 1	0 1 1	
P4	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 our safety requirement.

DEADLOCK DETECTION

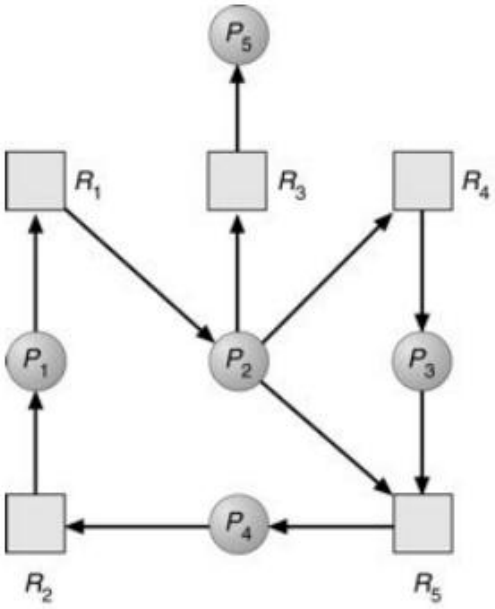
If a system does not employ either a deadlock-prevention or a deadlock-avoidance algorithm, then a deadlock situation may occur. In this environment, the system must provide:

1. An algorithm that examines the state of the system to determine whether a deadlock has occurred
2. An algorithm to recover from the deadlock.

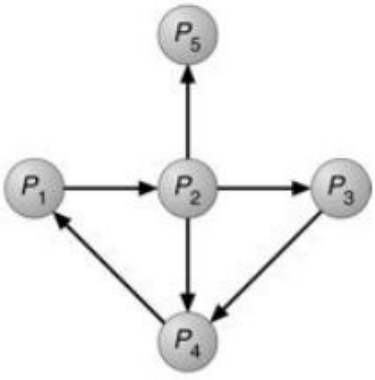
Two requirements as they pertain to systems with only a single instance of each resource type, as well as to systems with several instances of each resource type.

Single instance of each resource type:

If all resources have only a single instance, then we can define a deadlock-detection algorithm that uses a variant of the resource-allocation graph, called a wait-for graph.



Resource Allocation Graph



Corresponding wait-for graph

A 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.

Several instances of a resource type

The wait-for graph scheme is not applicable to a resource-allocation system with multiple instances of each resource type. The algorithm employs several time-varying data structures that are similar to those used in the banker's algorithm.

1. *Available:* A vector of length m indicates the number of available resources of each type.

2. *Allocation:* An n × m matrix defines the number of resources of each type currently allocated to each process.

3. *Request:* · An n × m matrix indicates the current request of each process. If Request[i][j] equals k, then process P_i is requesting k more instances of resource type R_j.

RECOVERY FROM DEADLOCK

When a detection algorithm determines that a deadlock exists, several alternatives are available. One possibility is to inform the operator that a deadlock has occurred and to let the operator deal with the deadlock manually. The other possibility is to let the system recover from the deadlock automatically.

There are two options for breaking a deadlock. One is simply to abort one or more processes to break the circular wait. The other is 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

1. *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.

2. *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.

Many factors may affect which process is chosen, including:

1. What the priority of the process is
2. How long the process has computed and how much longer the process will compute before completing its designated task
3. How many and what type of resources the process has used (for example, whether the resources are simple to preempt)
4. How many more resources the process needs in order to complete
5. How many processes will need to be terminated
6. Whether the process is interactive or batch

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:

1. *Selecting a victim:*
   As in process termination, we must determine the order of preemption to minimize cost

2. *Rollback :*
   If we preempt a resource from a process, 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.

3. *Starvation:*
   Ifprocess never completes its designated task, a starvation situation that needs to be dealt with in any practical system where a process that needs several popular resources may have to wait, because atleast one of the resources that it needs is always allocated to some other process. This situation is known as starvation.