Deadlocks

8.1 CONCEPT OF DEADLOCK

A deadlock is a condition where two or more users are waiting for data, locked by each other. Oracle automatically detects a deadlock and resolves them.

- Deadlock occurs when transactions executing at the same time lock each other out of data that they need to complete their logical units of work.
- Deadlock is a situation where a group of processes are all blocked and none of them can become unblocked until one of the other becomes unblocked.

In a multiprogramming environment, several processes may compete for a finite number of resources. 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. This situation is called a deadlock.

Consider a case where two different processes want to be allocated on the same resource (say printer) at a particular time. If both the processes requests for the same resource then the system will come under the state of deadlock because a single resource can attend only one process at a time. In other words, a printer can print only one process (document) at a time.

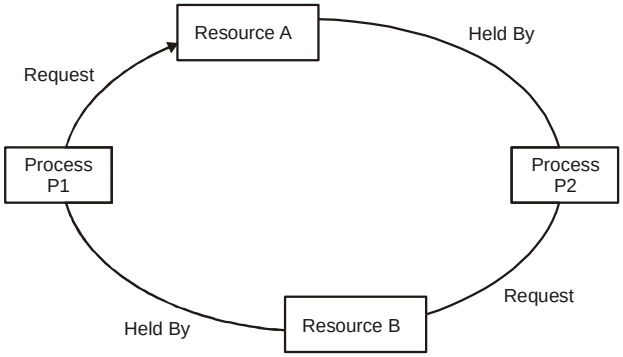


Fig. *Deadlock*

Some basic points on deadlock

Permanent blocking of a set of processes that either compete for system resources or communicate with each other

1. Involves conflicting needs for resources by two or more processes
2. No efficient general solution.

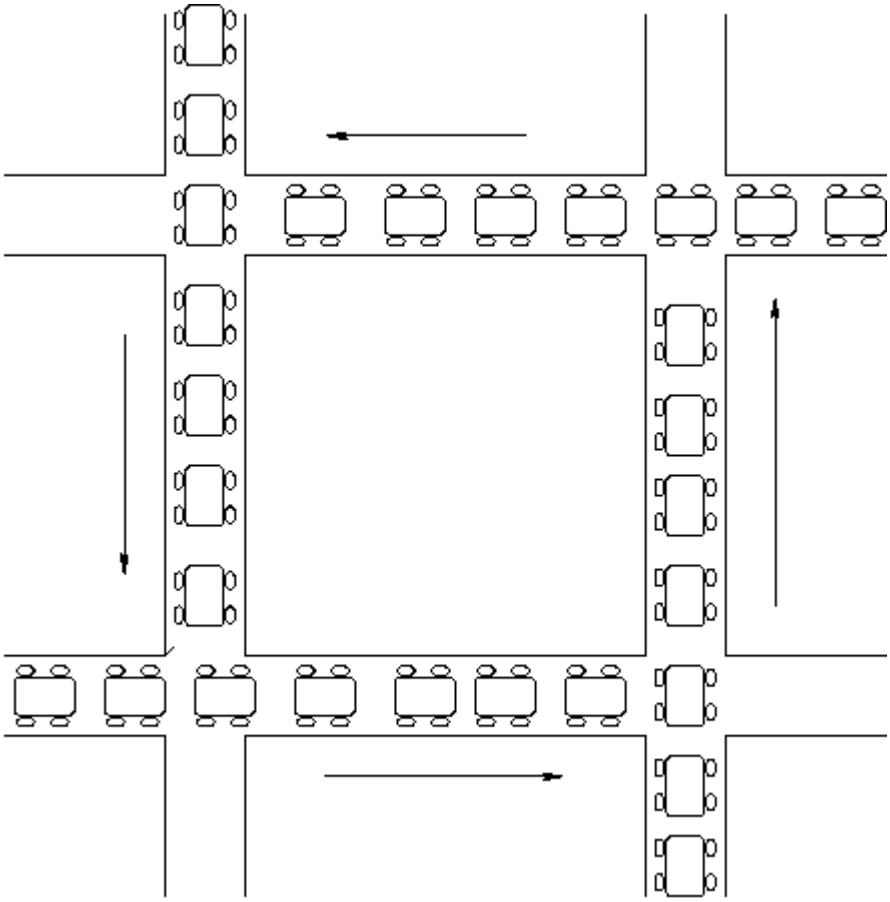
Necessary Conditions for Deadlock

The 4 Necessary Conditions for Deadlock

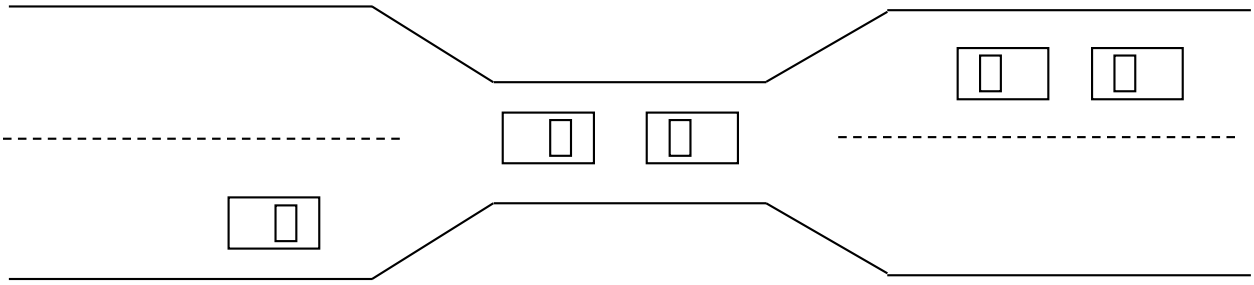
1. Exclusive access (mutual exclusion): only one process may use a resource at a time
2. Wait while holding (hold-and-wait):A process can continue to hold a resource while requesting another.
3. No preemption: A process cannot be forced to give up resources before it chooses to give them up.
4. Circular wait: There is a cycle of hold-and-wait relationships.

In order for there to be a deadlock, all of the above conditions must be true. You can observe that they are true in the examples we considered. Richard Holt, in a PhD dissertation published in the 1970's, showed that these conditions must apply in any deadlock. Informally, by looking at each condition and convincing yourself that if the condition is not true, there is no deadlock.

Example: Traffic gridlock is an everyday example of a deadlock situation.



BRIDGE CROSSING EXAMPLE



- Traffic only in one direction.
- Each section of a bridge can be viewed as a resource.
- If a deadlock occurs, it can be resolved if one car backs up (preempt resources and rollback).
- Several cars may have to be backed up if a deadlock occurs.
- Starvation is possible.

8.2 DEADLOCK 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 consisting of some number of identical instances. Memory space, CPU cycles, files, and I/O devices (such as printers and DVD drives) are examples of resource types. If a system has two CPUs, then the resource type CPU has two instances. Similarly, the resource type *printer* may have five instances.

A process must request a resource before using it and must release the resource after using it. A process may request as many resources as it require

es to carry out its designated task. Obviously, the number of resources requested may not excceed the total number of resources available in the system. In other words, a process cannot request three printers if the system has only two.

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

1. **Request:** The process requests the resource. 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.

8.3 DEADLOCK CHARACTERIZATION

In a deadlock, processes never finish executing, and system resources are tied up, preventing other jobs from starting. Before we discuss the various methods for dealing with the deadlock problem, we look more closely at features that characterize deadlocks.

8.3.1 Necessary Conditions

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 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 $\{P_0, P_1, \ldots, P_n\}$ of waiting processes must exist such that P_0 is waiting for a resource held by P_1, P_1 is waiting for a resource held by P_2,..., P_{n-1} is waiting for a resource held by P_n, and P_n is waiting for a resource held by P_0.

8.3.2 Resource-Allocation Graph

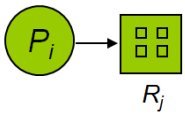
Deadlocks can be described more precisely in terms of a directed graph called a system resource-allocation graph. This graph consists of a set vertices V and a set of edges E. The set of vertices V is partitioned into two different types of nodes: $P = \{P_1, P_2, \ldots, P_n\}$, the set consisting of all the active processes in the system, and $R = \{R_1, R_2, \ldots, R_m\}$, the set consisting of all resource types in the system.

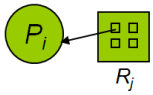
A set of vertices V and a set of edges E.

- n V is partitioned into two types:
- l $P = \{P_1, P_2, \ldots, P_n\}$, the set consisting of all the processes in the system.
- l $R = \{R_1, R_2, \ldots, R_m\}$, the set consisting of all resource types in the system.
- n request edge – directed edge $P_1 \rightarrow R_j$
- n assignment edge – directed edge $R_j \rightarrow P_i$

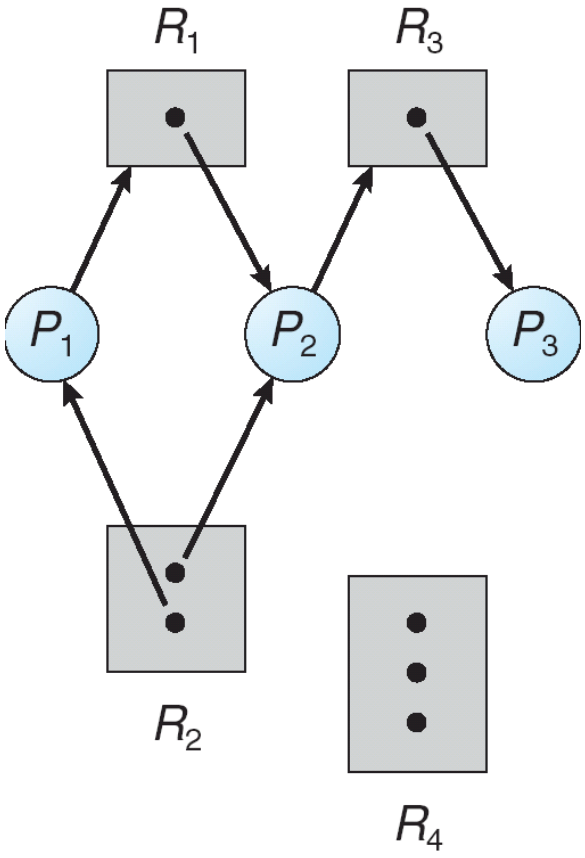
Process 

Resource Type with 4 instances 

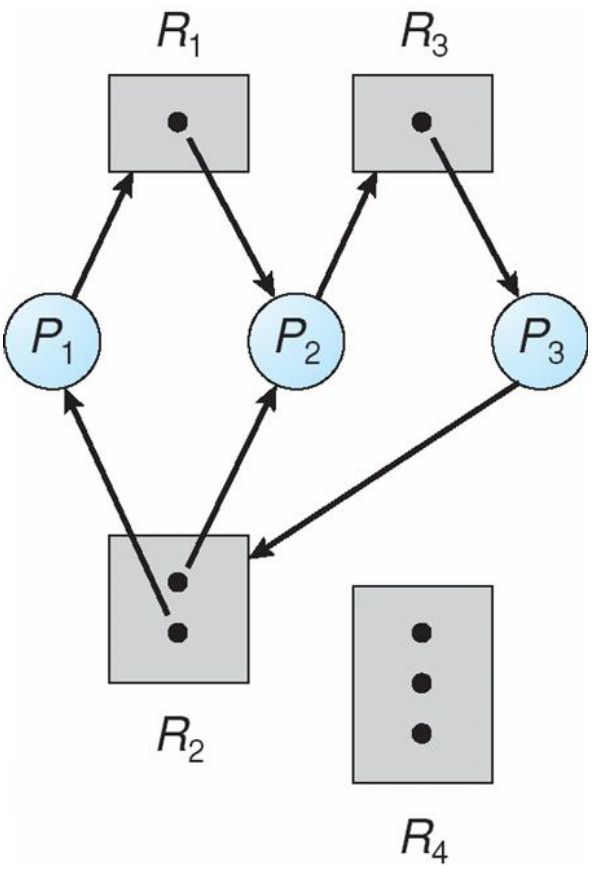
P_i requests instance of R_j 

P_i is holding an instance of R_j 

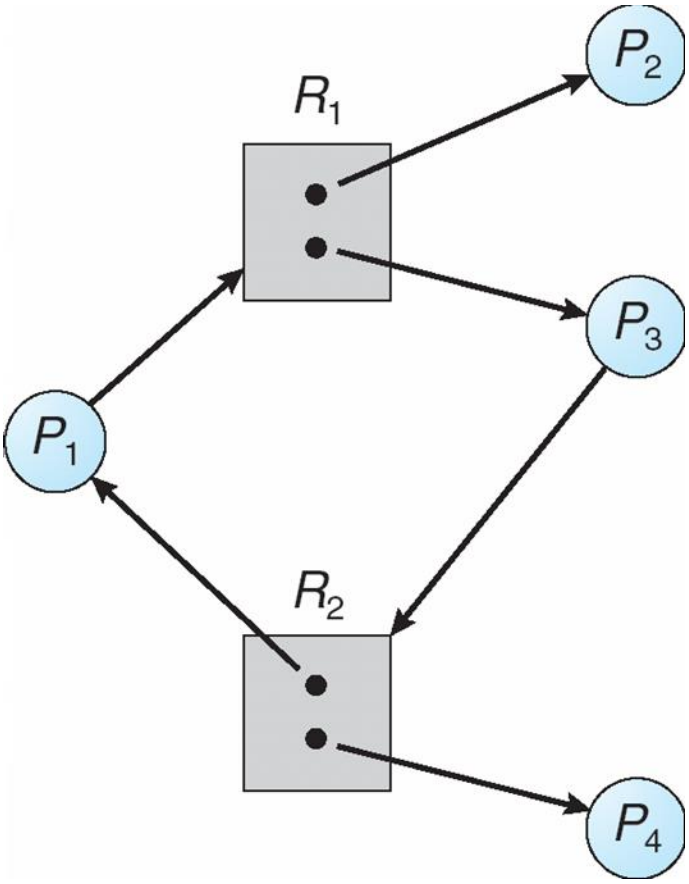
Example of a Resource Allocation Graph

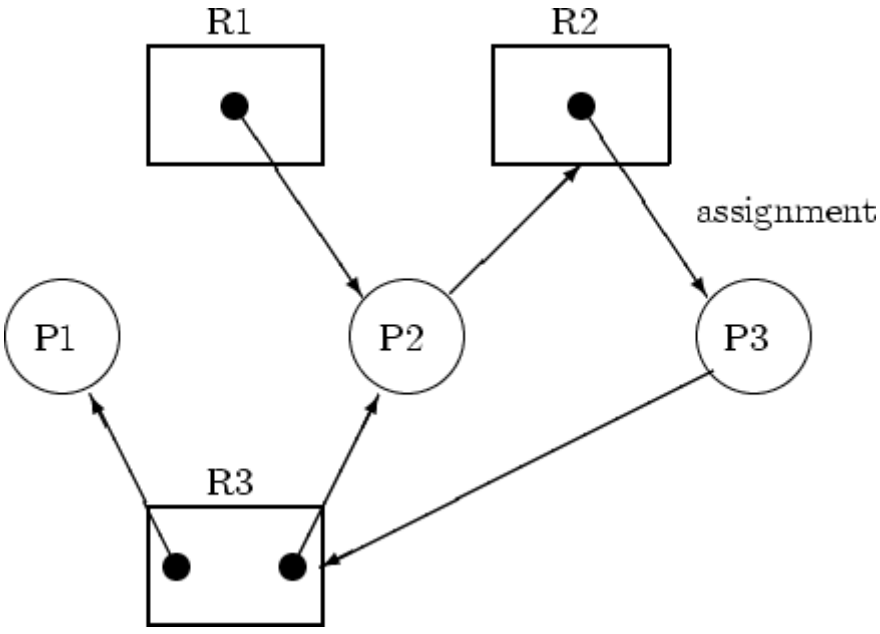


Resource Allocation Graph With A Deadlock



Graph With A Cycle But No Deadlock


P_2
R_1
P_3
P_1
R_2
P_4


R1
R2
assignment
P1
P2
P3
R3

Note:

- If graph contains no cycles ⇒ no deadlock
- If graph contains a cycle ⇒
  - if only one instance per resource type, then deadlock
  - if several instances per resource type, possibility of deadlock

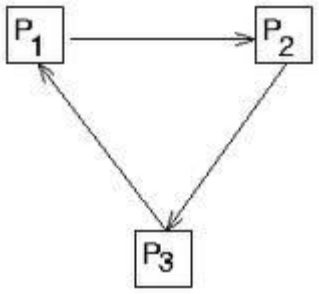
1. Wait-For Graphs (WFG)

Nodes correspond to processes (only).

There is an edge from process P_1 *to process* P_2 *if* P_1 *is blocked waiting for* P_2 *to release some resource.*

This type of graph models the wait-for relationships among processes, at a high level of abstraction. A cycle in this graph is a necessary condition for deadlock. Whether existence of a cycle is sufficient for deadlock depends on the kinds of wait-for relationships among processes we allow.

The following forms of graphs provide more detail, showing enough about the specifics of the wait-for relation that we can talk about deadlock detection.



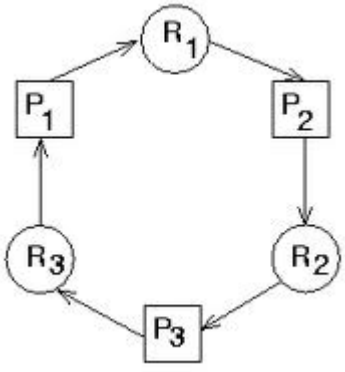
2. Single-Unit Resource Allocation Graphs

In this model there are two types of nodes, corresponding to processes and resources, respectively. Each resource is a single-unit reusable resource.

There is a request *edge from process* P *to resource* R *if* P *is blocked waiting for an allocation of* R.

There is an assignment *edge from resource* R *to process* P *if* P *is holding an allocation of* R.

In this model, the existence of a cycle is a necessary and sufficient *condition for deadlock.*



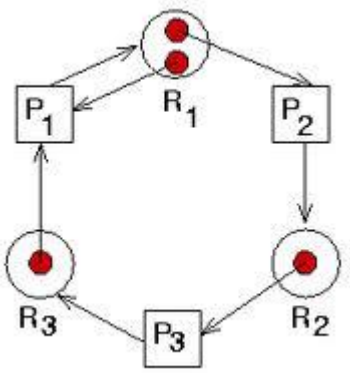
3. Multiunit Resource Allocation Graphs

In this form of graph the resource nodes are complex.

There is an outer-level node, which represents a collection of resources, and one or more nodes within, which represent units of that resource. A request is represented by an edge from a process to a resource collection. An allocation is represented by an edge from a unit of a resource to a process.

The graph shown in the figure is a multi-unit reusable resource graph. There are three processes (P_1, P_2, and P_3) and three collections of resources (R_1, R_2, and R_3). Resource collections R_2, and R_3 each have only a single unit. Resource collection R_1 has two units. Process P_1 is holding one unit of resource R_1 and one unit of resource R_3 and waiting on a request for the second unit of resource R_1. Process P_2 is holding one unit of resource R_1 and is waiting on a request R_2. Process P_3 is holding one unit of resource R_2 and is waiting on a request R_2. This is a deadlock, since every process is blocked on a resource request, and none of the requests can be satisfied.

The example can be modified slightly and used to show that the existence of a cycle in a multiunit resource graph is not a sufficient condition for deadlock. Just remove the edge that goes from P_1 to R_1, i.e., suppose process P_1 has not (yet) made its request for a second unit of R_1. There would still be a cycle in the graph, but since P_1 would not be blocked, there would be no deadlock.



8.4 DEADLOCK PREVENTION

The deadlock prevention and deadlock-avoidance algorithms presented in this chapter can also be used in a distributed system, provided that appropriate modifications are made.

1. We can use the resource-ordering deadlock-prevention technique by simply defining a global among the system resources. That is, all resources in the entire system are assigned unique numbers, and a process may request a resource (at any processor) with unique number if only if it is holding a resource with a unique number greater than i.
2. We can use the banker's algorithm in a distributed system by designating one of the processes in the system as the process that maintains the information necessary to carry out the banker's algorithm. Every resource request must be channeled through the banker's algorithm.

   These two schemes can be used in dealing with the deadlock problem in a distributed environment. The first scheme is simple to implement and requires little overhead. The second scheme can also be implemented easily, but if many require too many overheads. Since the number of messages to and from the process executing banker's algorithm may be large, therefore, the banker's scheme does not seem to be of practical use in a distributed system.
3. Another scheme based on time stamp-ordering approach is being mentioned as under:

   To control the preemption, we assign a unique priority number of each process. These numbers are used to decide whether a process A should wait for process B. For example, we can let A wait for B if A has a priority higher than that of B; otherwise A is rolled back. This scheme prevents deadlock

because, for every edge A $\rightarrow$ B in the wait-for graph, A has a higher priority than B. Thus, A can wait for B but B cannot wait for A and therefore, a cycle cannot exist.

One difficulty with this scheme is the possibility of starvation. Some processes with extremely low priority may always be rolled back. This difficulty can be avoided through the use of timestamps. Each process in the system is assigned a unique timestamp when it is created. Two deadlock-prevention schemes using timestamps have been explained below:

The wait and die scheme: This approach is based on a non-preemptive technique. When process A requests a resource currently held by B, A is allowed to wait only if it has a smaller timestamp than that of B. Otherwise A is rolled back (dies). For example, suppose that processes P_1, P_2 and P_3 have timestamps 2, 5 and 10 respectively. If P_1 requests a resource held by P_2, P_1 will wait. If P_3 requests a resource held by P_2, P_3 will be rolled back. In this scheme, an older process must wait for a younger one to released its resource. Thus, the older the process gets the more it tends to wait.

The wound and wait scheme: The approach is based on a preemptive technique and is a counterpart to the wait and die system. When process A requests a resource currently held by B, A is allowed to wait only if it has a larger timestamp than does B (that is, A is younger than B). Otherwise, B is rolled back. Looking at our previous example, with processes P_1, P_2 and P_3 if P_1 requests a resource held by P_2, then the resource will be preempted from P_2 and P_3 will be rolled back. If P_3 requests a resource held by P_2, then P_2 will wait. In this scheme, an older process never waits for a younger process.

The major problem with these two schemes is that unnecessary rollbacks may occur.

8.4.1 Mutual Exclusion

An effort should be made to prevent the mutual exclusion. This is possible by making all the sharable resources sharable. For example, if several processes would like to access a read-only file, then allow that to happen. No need for Mutual Exclusion in such a case. But few resources cannot be changed from non-sharable to sharable. For example, if you have several processes that want to update a file, and you allow them to have simultaneous write privileges, then your data might be inconsistent. This is a condition where mutual exclusion is must. As mentioned in the previous chapter, mutual exclusion is one of the requirements of the critical section problem's solution also, thus preventing it may sometimes lead to undesired results. Therefore, overall conclusion is that denying mutual exclusion can create more problems than it solves.

8.4.2 Hold and Wait

To prevent hold-and-wait condition from happening, we can have a rule that says, "a process may not request a resource if it is holding another resource". So, to take the print out of the contents of a file, you first request the disk, then you get it, use it and release it. Then you request the printer, you get it, you use it, and then you release it.

Thus, it implies that a process should have released all its resources before it requests for additional resources. Or another rule can be there that "a process should request and acquire all the requested resources before its execution begins". For example, to take the print out of the contents of a file, the disk and printer should be requested before hand.

In either case, there are two problems:

1. **Low Resource Utilization:** If a process follows the second rule, it will acquire all the resources like disk, tape and printer at the very beginning of its execution, no matter that it might need the printer at the end of its execution. This will stop other processes from accessing the printer and thus printer utilization will be very low.
2. **Starvation:** A process that needs many resources to start its execution, may be waiting for one or the other resource (allocated to other processes) for an indefinite time.

8.4.3 No pre-emption

If a process that is holding some resources request another resources that cannot be immediately allocated to it then resources currently being hold are pre-empted. The process will restart only when it can regain its old resources as well as the new ones it is requesting.

Alternatively: If a process request some resources we first check whether they are available, if they are, we allocate them. If they are not available we check if they are allocated to some other process that is waiting for some other resources. If so we prompt the desined resources from waiting process and allocate them to requesting process. If the resources are not either available or hold by a waiting process the requesting process must wait and while it is waiting some of the resources may be pre-empted if another process request them. A process can be restarted only when it is allocated the new resources it is requesting and recovers any resource that was pre-empted.

8.4.4 Circular Wait

To avoid circular wait total ordering of all resource type is imposed and each process is required to request resources in increasing order of enumeration.

Let $R = [R_1, R_2 \ldots R_n]$ be set of resources types. Each resource type is allocated a unique integer number. A one to one function is defined.

$F : R \rightarrow N$ where 'N' is set of natural number

F(tape derive) = 1

F(disk drive) = 5

F(printer) = 12

Each process can request resources in increasing order i.e. if as initially has requested instance, of resource R_i then it can request instances of resources R_j if $F(R_j) > F(R_i)$

If several instances of some resource type is needed a single request for all of them must be issued.

Note that function (F) must be defined in normal order of usage of resource in the system.

All of the four conditions are necessary for deadlock to occur.

Hence, by preventing any one of them we prevent deadlock.

1. *Exclusive access (mutual exclusion*: redesign to eliminate the need for mutual exclusion.
2. *Wait while holding (hold-and-wait)*: If a process holding resources is denied a further request, the process must release all its resources and rerequest them.
   - Require that a process request all of its required resources at one time.
3. *No preemption*: If a process requests a resource that is currently held by another process, the OS preempts the second process and requires it to release its resources
4. *Circular wait*: Define a linear ordering of resources and require allocations be requested only in this order

If we look at each of the four necessary conditions for deadlock, we can see how deadlock might be prevented by denying that condition. Whether one of these prevention strategies can apply to a given resource depends on the nature of the resource and how it is used. For example, ordered allocation works well for mutexes, but preemption of a mutex is not acceptable.

8.5 DEADLOCK AVOIDENCE

The methods of handling deadlock through prevention results in low device utilization and reduced throughput.

Another alternative method for avoiding deadlock is to require additional information about how the resources are to be requested with complete knowledge of sequence of request and release for each process

we can decide for each request whether the current requesties satisfied or must wait to avoid a future deadlock.

The simplest algorithm requires that each process should declare maximum number of resources of each type that may need. A deadlock avoidence algorithm dynamically examine the resource allocation state to ensure that there can never be a circular wait condition.

A state is safe if system can allocate resources to each process in some order and still to system avoid deadlock avoid deadlock

e.g. to illustrate the above concept.

In the condition the system	P_0	10 max	5 current
is safe. Now the sequence	P_1	4	2 allocation
	P_2	9	2
(P_1, P_0, P_2) statisfies the safety	Total tapes = 12		
algorithm. Since P_1 can be	Available = 3		

immediately allocated all its tap need finishes and releases 4 tapes. Now available tapes are 5 which can be allocate to P_0 and after completing P_0, all the 10 tapes, will be released and seven be allocated to P_2. If can make to unsafe state if request of P_2 for one tape is granted. In this condition the system is no longer in safe state. At this point only process P_1 can be allocate all its tape when it returns them the system will have or 4 tapes and system will never complete any process.

A decision is made dynamically whether the current resource allocation request will, if granted, potentially lead to a deadlock. Requires knowledge of worst-case future process requests Approaches:

1. Postpone starting a process if its demands might lead to deadlock, i.e. *while resources it may need are held by others*
2. Postpone granting an incremental resource request to a process if granting the allocation might lead to deadlock

Deadlock avoidance requires foreknowledge of the worst-case resource requests a process may make.

The two logical points at which a deadlock avoidance decision can be made are the point of process admission and the point of an incremental resource request.

Requirements for Deadlock Avoidance

- Maximum resource requirement must be stated in advance
- Processes under consideration must be independent; no synchronization requirements
- There must be a fixed number of resources to allocate
- No process may exit while holding resources

Maximum resource requirement information may not be available, except for certain kinds of resources and certain restricted types of processes. The availability of this kind of information was typical of traditional batch processing systems, and is true for some real-time operating systems.

The kinds of resources this method may be applied to are also limited. It has been used for allocating tape drives to jobs in a batch-oriented operating system. Another example of an application is to the allocation of signal processors to a main-CPU process in a real-time signal processing system.

8.5.1 Banker's Algorithm

(Several instances of a resource type)

When a new process enters the system it must declare maximum number of instance of each resource type it may need.

When a user request a set of resources, the system must determine whether the allocation of these resources will leave the system in safe state. If it will, then the resources are allocated, otherwise the process must wait untill some other process releases enough resources.

Several data structure must be maintained to implement the banker's algorithm. The data structures encode the state of resource allocation.

let n = number of processes in the system.

m = number of resources type then following data structure is defined

Available: A vector of length m indicating number of available resources of each type

Available [1] = k, there are 'k' instances of resources type R, are available

Maximum: An $n \times m$ matrix defining maximum demand of each process

max[i, 1] = k, process P_i may request at most k-instances of resources type R_j.

Allocation: An $n \times m$ matrix defining the number of resources of each type currently allocated to each process.

Allocation[i, J] = k, process P_i is currently allocated k instance of R_j.

Need: An $n \times m$ matrix indicating remaining resource need of each process.

need[i, j] = k, process P_i may need k more instances of R_j in order to complete its task.

need[i, j] = max[i, j] – Allocation[i, 1]

Each row in matrix allocation and need are treated as vectors and referred as allocations i and need i respectively.

Allocation $i \rightarrow$ Specified the resources currently allocated to process P_i

Need $i \rightarrow$ Specifies additional resources that P_i may still request in order to complete its task.

Let request i be the request vector for process P_i.

If request i[j] = k

process P_i wants 'k' instances of resources type R_j, when request for resource is made by process P_i following action are taken.

1. If request $i \leq$ need i go to step 2, otherwise an error condition will be raised since the process has executed. Its maximum claim.
2. If request $i \leq$ Available goto step 3, otherwise P_i must wait since the resources are not available.
3. The system pretend to have allocated the resource to P_i by modifying the state as

   Available = Available – Request i

   Allocation i = Allocation i + Request i

   Need i = Need i – Request i

If the resulting allocation state is safe the transaction is completed and process P_i is allocated its resource. However if the new state is unsafe then P_i must wait for the request and old allocation state is restored.

Concept of Banker's Algorithm

The resource-allocation-graph algorithm is not applicable to a resource-allocation system with multiple instances of each resource type. The deadlock-avoidance algorithm that we describe next is applicable to such a system but 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 allocated its available cash in such a way that it could 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. 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.

Several data structure must be maintained to implement the banker's algorithm. These data structures encode the state of the resource-allocation system. Let *n* be the number of processes in the system and *m* be the number of resource type. 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 $n \times m$ 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 $n \times m$ 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 $n \times m$ 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 R_j to complete its task. Note that Need[*i*][*j*] equals Max [*i*][*j*]-Allocation[*i*][*j*].

These data structures vary over time in both size and value.

To simplify the presentation of the banker's algorithm, we next establish some notation. Let X and Y be vectors of length *n*. we say that $X \le Y$ if and only if $X[i] \le Y[i]$ for all $i = 1, 2, \ldots n$. for example id X= (1, 7, 3, 2) and Y = (0, 3, 2, 1) then $Y \le X$. $Y < X$ if $Y \rightarrow X$ and $Y \rightarrow X$.

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 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.

An Illustrative Example

Finally, to illustrate the use of the banker's algorithm, consider a system with five process P0 through P4 and three resource types A, B, and C. Resource type A has 10 instances, resource type B has 5 instances, and resource type C has 7 instances. Suppose that, at time T_0, the following snapshot of the system has been taken:

	Allocation	Max	Available
	A B C	A B C	A B C
P0	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 as follows:

	Need
	A B C
P0	7 4 3
P1	1 2 2
P2	6 0 0
P3	0 1 1

P4	4 3 1	

When 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 Request1 < Available-that is, that (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
P0	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 <P1, P3, P4, P0, P2> satisfies the safety requirement. Hence, we can immediately grant the request of process P1.

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.

We leave it as a programming exercise to implement the banker's algorithm.

8.5.2 Safety Algorithm

1. Look for a row whose unmet resource needs are all smaller than available. If no such row exists the system is deadlocked since no process will run to completion.
2. Assume the process of row choose a requests all the resources it needs which is guaranteed to be possible and finishes. Mark that process as terminated and all its resources to available vector.
3. Repeat step (1) and (2) untill all the process marked terminated in which case the initial state was safe or untill a deadlocks occurs in which it was unsafe.

1. System has 5 processes $P_0 - P_4$.

   Allocation

   Maximum

   Need [*i*, *j*] = Maximum [*i*, *j*] – Allocation [*i*, *j*]

   =

2. An O.S. contains 2 resources process to start. The number of resources units are (4 and 5) respecting. The current resource allocation is as

	Allocated		Maximum need	
	R_1	R_2	R_1	R_2
P_1	1	3	1	5
P_2	2	1	3	2

would the following request be guaranted in

(a) P_2 request (1, 0)

(b) P_2 request (0, 1)

(c) P_2 request (1, 1) **Ans.**

(d) P_1 request (1, 0)

(e) P_1 request (0, 1)

need (*i*, *j*) =

Various method for deadlock prevention involues that one of the four necessary conditions.

- Each process must request all its required resources at once and do not proceed untill all the have been granted.
- If a process is holding certain resources is devied further request that process must release it origin resources and if necessary request than again together with the additional resources
- Impose a linear ordering of resources type on all processes i.e. if a process has been allocated resources of given type, if may subsequently request only those resources of types later in ordering.

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, I, …, *n*–1.
2. Find an *i* such that both
   (a) Finish[i] = = false
   (b) Need d" 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]= =time for all I, then the system is in a safe state.
5.

This algorithm may require an order of $m \times n^2$ operations to determine whether a state is safe.

8.5.3 Resource-Request Algorithm

We now describe the algorithm which determines if requests can be safely granted.

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

1. If $request_i < Need_i$, go to step 2. Otherwise, raise an error condition, since the process has exceeded its maximum claim.
2. If $Request_i$<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.

8.6 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 may provide:

- An algorithm that examines the state of the system to determine whether a deadlock has occurred.

- An algorithm to recover from the deadlcok

8.6.1 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. We obtain this graph from the resource-allocation graph by removing the resource nodes and collapsing the appropriate edges.

More precisely, an edge from P_i to P_j in a wait-for graph implies that process P_i is waiting for process P_j to release a resource that P_i needs. An edge $P_i \rightarrow P_j$ exists in a wait-for graph if and only if the corresponding resource-allocation graph contains two edges $P_i \rightarrow R_q$ and $R_q \rightarrow P_j$ for some resource R_q.

As 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. An algorithm to detect a cycle in a graph requires an order of n^2 operations, where n is the number of vertices in the graph.

8.6.2 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. We turn now to a deadlock-detection algorithm that is applicable to such a system. The algorithm employs several time-varying data structures that are similar to those used in the banker's algorithm (Section 7.5.3):

- **Available:** A vector of length m indicates the number of available resources of each type.
- **Allocation:** An $n \times m$ matrix defines the number of resources of each type currently allocated to each process.

8.6.3 DETECTION-ALGORITHM USAGE

When should we invoke the detection algorithm? The answer depends on two factors:

1. How *often* is a deadlock likely to occur?
2. How *many* processes will be affected by deadlock when it happens?

If deadlocks occur frequently, then the detection algorithm should be invoked frequently. Resources allocated to deadlocked processes will be idle until the deadlock can be broken. In addition, the number of processes involved in the deadlock cycle may grow. Deadlocks occur only when some process makes a request that cannot be granted immediately. This request may be the final request that completes a chain of waiting processes. In the extreme, then, we can invoke the deadlock-detection algorithm every time a request for allocation cannot be granted immediately.

8.7 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. Another 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 deadlock processes.

8.7.1 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 along 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.

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, rest the printer to a correct state before printing the next job.

If the partial termination method is used, then we must determine which deadlocked process (or processes) should be terminated. This determination is a policy decision, similar to CPU-scheduling decisions. The question is basically an economic one; we should abort those processes whose termination will incur the minimum cost. Unfortunately, the term minimum cost is not a precise one. 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. Whether the process is interactive or batch

8.7.2 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. **Selection 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 during its execution.
2. **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 if 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.
3. **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 that must be dealt with in any practical system. 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.

SUMMARY

Discuss the methods of** Deadlock: Prevention**?

1. break **mutual exclusion**:
2. read-only files are shareable
3. but some resources are intrinsically nonshareable (printers)
4. break **hold and wait**:
5. request all resources in advance
6. request (tape, disk, printer)
7. release all resources before requesting new batch
8. request (tape, disk), release (tape, disk), request (disk, printer)

Disadvantages: low resource utilization, starvation

Deadlock

You may need to write code that acquires more than one lock. This opens up the

Safe State and what is its use in deadlock avoidance

When a process requests an available resource, system must decide if immediate allocation leaves the system in a safe state. System is in safe state if there exists a safe sequence of all processes. Deadlock Avoidance ensure that a system will never enter an unsafe state. Sequence is safe if for each Pi, the resources that Pi can still request can be satisfied by currently available resources + resources held by all the Pj, with j.

Q.1. How do we know a state is safe?

- Start in the given state.
- Simulate running each process to completion, by allocating its maximum resource requirements and then releasing all resources.
- If all processes can complete, the state is safe.

Q.2 When is a system in safe state?

Ans. The set of dispatchable processes is in a safe state if there exists at least one temporal order in which all processes can be run to completion without resulting in a deadlock.

Q.3 How to Detect Deadlock?

Ans.

- Similar to detecting an unsafe state
- Simulate execution of unblocked processes, assuming they will complete and release all resources

Q.4 What to do when Deadlock is Detected?

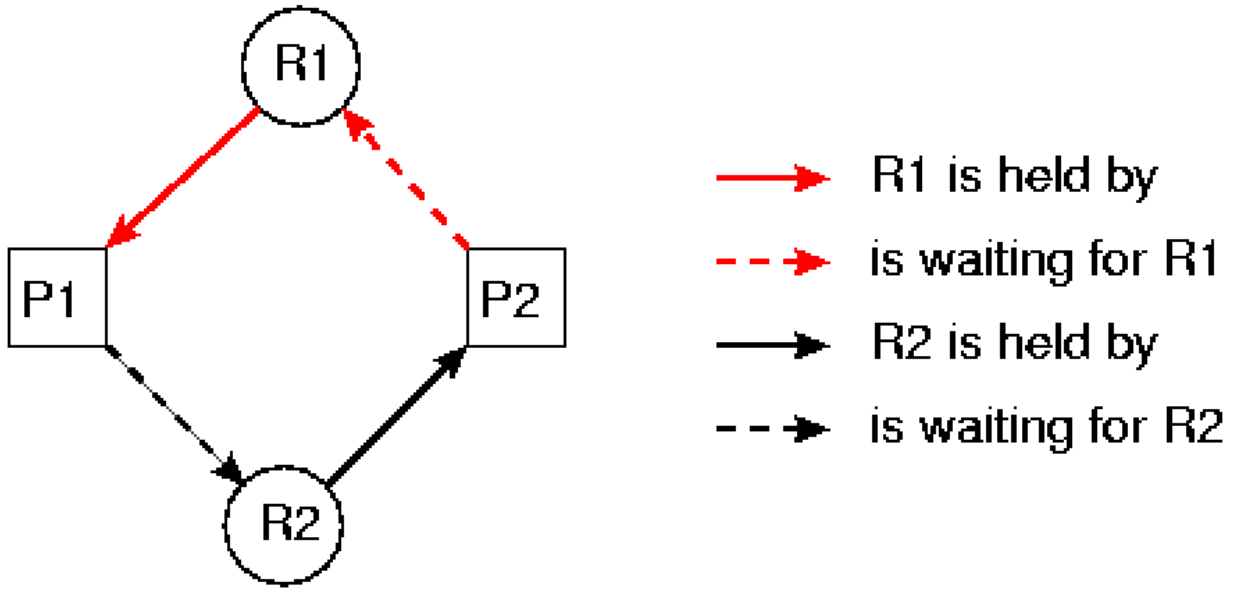
Ans:

1. Abort all deadlocked processes
2. Back up each deadlocked process to some previously defined checkpoint, and restart it from the checkpoint of original deadlock may reoccur
3. Kill deadlocked processes until deadlock no longer exists
4. Preempt resources until deadlock no longer exists

Q.5 How to Choose which Process to Abort?

- Least amount of processor time consumed so far
- Least number of lines of output produced so far
- Most estimated time remaining
- Least total resources allocated so far
- Lowest priority

Concept of Resource Allocation Graph of a Deadlocked System



The figure shows the deadlock that is possible for the two pseudo code processes above, modeled by a (single-unit) resource allocation graph.

There are two types of nodes: process nodes (show as squares) and resource nodes (show as circles).

There are two types of edges: held-by edges (shown as solid arrows) go from a resource to the process that is holding it; wait-for edges (shown as dashed arrows) go from a process to the resource it is waiting for. Notice that there is a cycle in the graph. A cycle in the resource allocation graph is a necessary condition for deadlock (in every model). For single-unit resources (but not in the more general models) it is also a sufficient condition.

Q.6 How can we deal with deadlock?

Ans. There are only three approaches to dealing with deadlock. In that case we need to be able to detect it when it has happened, and then deal with it. Alternatively, we can try to make sure deadlock does not happen. There are two general approaches to that.

Q.7 Name the three Strategies for Dealing with Deadlock?

Ans. The 3 Approaches/Strategies for Dealing with Deadlock

- Prevention - apply design rules to insure it can never occur
- Avoidance - dynamically steer around deadlocks
- Detection - hope deadlocks will not occur, but recover when one does
- possibility of deadlock. Consider the following piece of code:
- Lock *l1, *l2;
- void p() {
- l1->Acquire();
- l2->Acquire();
- code that manipulates data that l1 and l2 protect
- l2->Release();
- l1->Release();
- }
- void q() {
- l2->Acquire();

- l1->Acquire();
- code that manipulates data that l1 and l2 protect
- l1->Release();
- l2->Release();
- }

  If p and q execute concurrently, consider what may happen. First, p acquires l1 and q acquires l2. Then, p waits to acquire l2 and q waits to acquire l1. How long will they wait? Forever.
- This case is called deadlock.

Q.8 List three examples of deadlocks that are not related to a computer system environment.

Ans:

- Two cars crossing a single-lane bridge from opposite directions.
- A person going down a ladder while another person is climbing up the ladder.
- Two trains traveling toward each other on the same track.
- Two carpenters who must pound nails. There is a single hammer and a single bucket of nails. Deadlock occurs if one carpenter has the hammer and the other carpenter has the nails.

Q.9 Suppose that a system is in an unsafe state. Show that it is possible for the processes to complete their execution without entering a deadlock state.

Ans: An unsafe state may not necessarily lead to deadlock, it just means that we cannot guarantee that deadlock will not occur. Thus, it is possible that a system in an unsafe state may still allow all processes to complete without deadlock occurring. Consider the situation where a system has 12 resources allocated among processes P0, P1, and P2. The resources are allocated according to the following policy:

	Max	**Current**	**Need**
P0	10	5	
P1	4	2	
P2	9	3	6

```
for (int i = 0; i < n; i++) {
// first find a thread that can finish
for (int j = 0; j < n; j++) {
if (!finish[j]) {
boolean temp = true;
for (int k = 0; k < m; k++) {
if (need[j][k] > work[k])
temp = false;
}
if (temp) { // if this thread can finish
finish[j] = true;
for (int x = 0; x < m; x++)
work[x] += work[j][x];
}
}
}
}
```

Currently there are two resources available. This system is in an unsafe state as process P1 could complete, thereby freeing a total of four resources. But we cannot guarantee that processes P0 and P2 can complete. However, it is possible that a process may release resources before requesting any further. For example, process P2 could release a resource, thereby increasing the total number of

resources to five. This allows process P0 to complete, which would free a total of nine resources, thereby allowing process P2 to complete as well.

Q.10 Consider a computer system that runs 5,000 jobs per month with no deadlock-prevention or deadlock-avoidance scheme. Deadlocks occur about twice per month, and the operator must terminate and rerun about 10 jobs per deadlock. Each job is worth about $2 (in CPU time), and the jobs terminated tend to be about half-done when they are aborted. A systems programmer has estimated that a deadlock-avoidance algorithm (like the banker's algorithm) could be installed in the system with an increase in the average execution time per job of about 10 percent. Since the machine currently has 30-percent idle time, all 5,000 jobs per month could still be run, although turnaround time would increase by about 20 percent on average.

(a) What are the arguments for installing the deadlock-avoidance algorithm?

(b) What are the arguments against installing the deadlock-avoidance algorithm?

Answer: An argument for installing deadlock avoidance in the system is that we could ensure deadlock would never occur. In addition, despite the increase in turnaround time, all 5,000 jobs could still run. An argument against installing deadlock avoidance software is that deadlocks occur infrequently and they cost little when they do occur.

Q.11 Can a system detect that some of its processes are starving? If you answer "yes," explain how it can. If you answer "no," explain how the system can deal with the starvation problem.

Ans: Starvation is a difficult topic to define as it may mean different things for different systems. For the purposes of this question, we will define starvation as the situation whereby a process must wait beyond a reasonable period of time—perhaps indefinitely—before receiving a requested resource. One way of detecting starvation would be to first identify a period of time—T—that is considered unreasonable. When a process requests a resource, a timer is started. If the elapsed time exceeds T, then the process is considered to be starved. One strategy for dealing with starvation would be to adopt a policy where resources are assigned only to the process that has been waiting the longest. For example, if process *Pa* has been waiting longer for resource X than process *Pb*, the request from process *Pb* would be deferred until process *Pa*'s request has been satisfied. Another strategy would be less strict than what was just mentioned. In this scenario, a resource might be granted to a process that has waited less than another process, providing that the other process is not starving. However, if another process is considered to be starving, its request would be satisfied first.

Q.12 Consider the following resource-allocation policy. Requests and releases for resources are allowed at any time. If a request for resources cannot be satisfied because the resources are not available, then we check any processes that are blocked, waiting for resources. If they have the desired resources, then these resources are taken away from them and are given to the requesting process. The vector of resources for which the process is waiting is increased to include the resources that were taken away.

For example, consider a system with three resource types and the vector Available initialized to (4, 2, 2). If process P0 asks for (2, 2, 1), it gets them. If P1 asks for (1, 0, 1), it gets them. Then, if P0 asks for (0, 0, 1), it is blocked (resource not available). If P2 now asks for (2, 0, 0), it gets the available one (1, 0, 0) and one that was allocated to P0 (since P0 is blocked). P0's Allocation vector goes down to (1, 2, 1) and its Need vector goes up to (1, 0, 1).

(a) Can deadlock occur? If you answer "yes", give an example. If you answer "no," specify which necessary condition cannot occur.

(b) Can indefinite blocking occur? Explain your answer.

Ans:

(a) Deadlock cannot occur because preemption exists.

(b) Yes. A process may never acquire all the resources it needs if they are continuously preempted by a series of requests such as those of process C.

Q.13 Suppose that you have coded the deadlock-avoidance safety algorithm and now have been asked to implement the deadlock-detection algorithm. Can you do so by simply using the safety algorithm code and redefining

Maxi = Waitingi + Allocationi, where Waitingi is a vector specifying the resources process i is waiting for, and Allocationi is as defined in Section 7.5? Explain your answer.

Ans: Yes. The Max vector represents the maximum request a process may make. When calculating the safety algorithm we use the Need matrix, which represents Max— Allocation. Another way to think of this is Max = Need + Allocation. According to the question, the Waiting matrix fulfills a role similar to the Need matrix, therefore Max = Waiting + Allocation.

Q.14 Is it possible to have a deadlock involving only one single process?

Explain your answer.

Ans: No. This follows directly from the hold-and-wait condition.