Deadlock

In multiprogramming environment, a system consists of a finite number of resources to be distributed among a number of competing processes.

A process requests resources; if the resources are not available at that time, the process enters a waiting state.

Waiting process may never again change state, because the resources it has requested are held by other waiting processes.

This situation is called a deadlock.

The resources may be partitioned into several types, each consisting of some number of identical instances.

CPU cycles, files, and I/O devices (such as Printer) are examples of resource types.

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, then the requesting process must wait until it can acquire the resource.

2. **Use**: The process can operate on the resource.

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

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

Multithreaded programs are the good candidates for deadlock because multiple threads can complete for shared resources.

Deadlock Characterization

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.

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 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.

All of these four conditions must hold for a deadlock to occur.

Resource-Allocation Graph

Deadlocks can be described in terms of a directed graph called **system resourceallocation graph**.

This graph consists 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: $P = \{P_1, P_2, \ldots, P_n\}$,and $R = \{R_1, R_2, ..., R_m\}$, the set consisting of all resource types in the system.

A directed edge from process P_i to resource type R_j is denoted by $P_i \rightarrow R_j$; it signifies that process P_i has requested an instance of resource type R_j and is currently waiting for that resource.

A directed edge from resource type R_j to process P_i is denoted by $R_j \rightarrow P_i$; it signifies that an instance of resource type R_j has been allocated to process P_i. 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**.

Pictorially, we represent each process P_i as a circle and each resource type R_j as a rectangle.

A resource type may have more than one instance; we represent each such instance as a dot within the rectangle.

A request edge points only to the rectangle R_j, whereas an assignment edge must also designate one of the dots in the rectangle.

When a process P_i requests an instance of resource type R_j, 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.

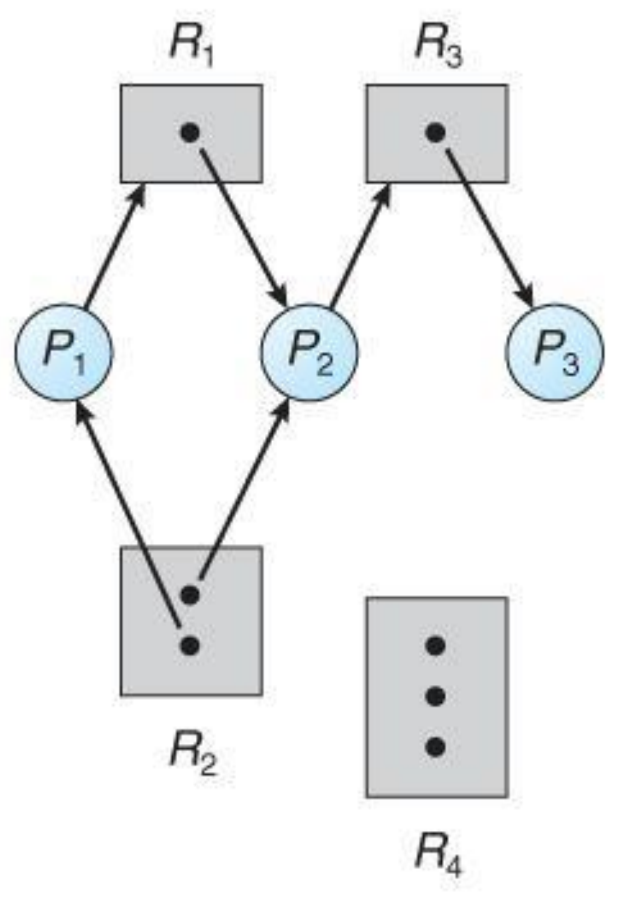

Fig 1: Resource allocation graph.

The resource-allocation graph shown in Fig 1 represents the following situation:

The sets P, R, and E:

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

$R = \{R_1, R_2, R_3, R_4\}$

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

Process P_1 is holding an instance of resource type R_2 and is waiting for an instance of resource type R_1.

Process P_2 is holding an instance of R_1 and an instance of R_2 and is waiting for an instance of R_3.

Process P_3 is holding an instance of R_3.

If the resource-allocation graph contains no cycles, then the system is not in deadlock state. But if there is a cycle, then the system may or may not be in deadlock state.

If each resource type has exactly one instance, then a cycle in the graph is both necessary and a sufficient condition for the existence of deadlock.

If each resource type has several instances, then a cycle in the graph is a necessary but not a sufficient condition for the existence of deadlock.

In our example, let consider that the 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 (as shown in Fig 2).

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

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

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

Here the process P_1, P_2 and P_3 are deadlocked.

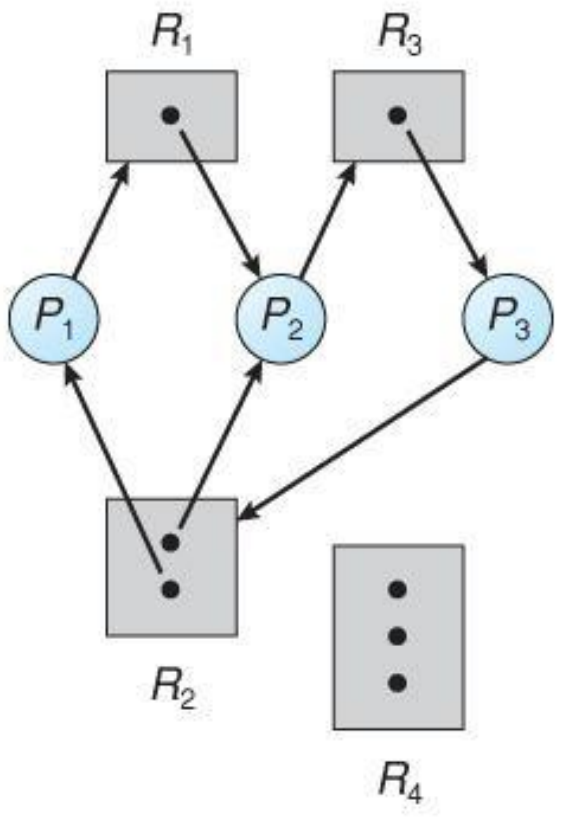

Fig 2: Resource-allocation graph with a deadlock.

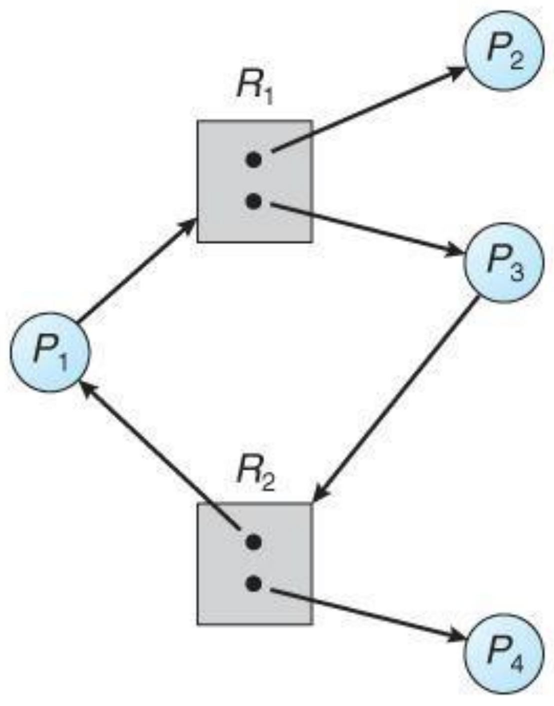

Fig 3: Resource-allocation graph with a cycle but no deadlock.

Now consider the resource-allocation graph in Fig 3.

In this example, we also have a cycle:

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

However, there is no deadlock.

Methods for Handling Deadlocks

The deadlock problem can be handled with one of three ways:

i) We can use a protocol to prevent or avoid deadlocks, ensuring that the system will never enter a deadlocked state.

ii) We can allow the system to enter a deadlocked state, detect it, and recover.

iii) We can ignore the problem altogether and pretend that deadlocks never occur in the system.

Deadlock Prevention

For a deadlock to occur, each of the four necessary conditions must hold.

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 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.

However, 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 can use following protocols.

i) Each process should allocate all its resources before it begins execution. ii) A process

should request resources only when the process has none.

Both these protocols have two main disadvantages.

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

Secondly, 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, 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 database transactions.

It cannot generally be applied to resources like printer.

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.

Let $R = \{R_1, R_2, ..., R_m\}$ be the set of resource types.

We assign to each resource type a unique integer number, which allows us to compare two resources.

Formally, we define a one-to-one function $F: R \rightarrow N$, where N is the set of natural numbers.

For example,

$$F(Disk\ drive)=5 \quad F(Printer)=12$$

A process can initially request an instance of a resource—say, R_i.

After that, the process can request an instance of resource R_j if and only if $F(R_j) > F(R_i)$.

Alternatively, a process requesting an instance of resource R_j must have released any resources R_i such that $F(R_i) \geq F(R_j)$.

If these two protocols are used, then the circular-wait condition cannot hold.

The function F should be defined according to the normal order of usage of resources in a system.

For example, *Disk drive* is usually needed before *Printer*, so

$$F(Disk\ drive) < F(Printer)$$

Deadlock Avoidance

The side effects of the deadlock prevention algorithm are low device utilization and reducing system throughput.

An alternative method for avoiding deadlocks is to require additional information about how resources are to be requested.

Each request requires that the system consider the resources currently available, the resources currently allocated to each process, and the future requests and releases of each process, to decide whether the current request can be satisfied or must wait to avoid a possible future deadlock.

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.

This 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.

A system is in a safe state only if there exists a safe sequence.

A sequence of process <P_1, P_2, ..., P_n> is a safe sequence for the current allocation state if, for each P_i, the resource requests that P_i can still make can be satisfied by the currently available resources plus the resources held by all P_j, with $j < i$.

In this situation, if the resources that P_i needs are not immediately available, then P_i can wait until all P_j have finished.

When they have finished, P_i can obtain all of its needed resources, complete its designated task, return its allocated resources, and terminate.

When P_i terminates, P_{i+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.

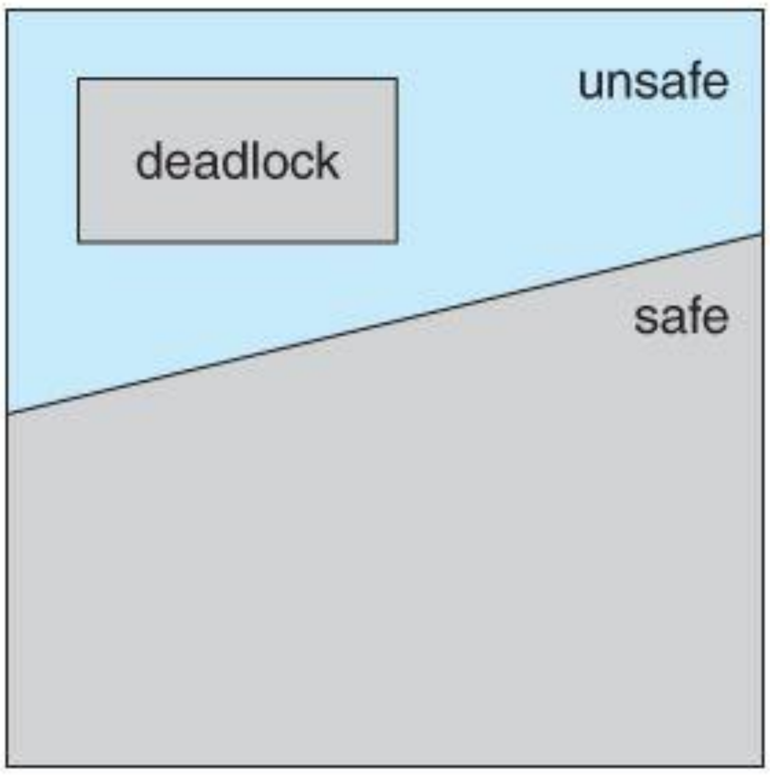

Fig 4: Safe, unsafe, and deadlocked state spaces.

Let consider a system with 12 resources and 3 processes: P_0, P_1, and P_2.

The processes P_0, P_1, and P_2 needs maximum 10, 4 and 9 resources, respectively. Suppose that, at time t_0, the processes P_0, P_1, and P_2.holding 5, 2 and 2 resources, respectively.

Therefore, there are 3 free resources.

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

At time t_0, the system is in a safe state.

The sequence < P_1, P_0, P_2> satisfies the safety condition.

Process P_1 can immediately be allocated all its resources and then return them (the system will then have 5 available resources); then process P_0 can get all its resources and return them (the system will then have 10 available resources); and finally process P_2 can get all its resources and return them (the system will then have all 12 resources available).

A system can go from a safe state to an unsafe state.

Suppose that, at time t_1, the process P_2 requests and is allocated 1 more resource.

Now the situation is as follows:

Process	Maximum Needs	Current Needs
P_0	10	5
P_1	4	2
P_2	9	3

Now the system is no longer in a safe state.

At this point, only process P_1 can be allocated all its resources.

When it returns them, the system will have only 4 available resources.

Since process P_0 is allocated 5 resources but has a maximum of 10, it may request 5 more resources.

If it does so, it will have to wait, because they are unavailable.
Similarly, process P_2 may request 6 additional resources and have to wait, resulting in a deadlock.

Here the mistake was in granting the request from process P_2 for 1 more resource.

Given the concept of a safe state, we can define the deadlock avoidance algorithms that 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 the process must wait.

The request is granted only if the allocation leaves the system in a safe state.

Resource-Allocation-Graph Algorithm

If a resource-allocation system has only one instance of each resource type, then a variant of resource-allocation graph can be used for deadlock avoidance.

In addition to the request and assignment edges, this graph has 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.

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 $P_i \rightarrow R_j$.

Before the process P_i starts executing, all its claim edges must be appear in the resource-allocation graph.

If the process P_i requests resource R_j; then 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.

Here 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, otherwise the allocation will put the system in an unsafe state.

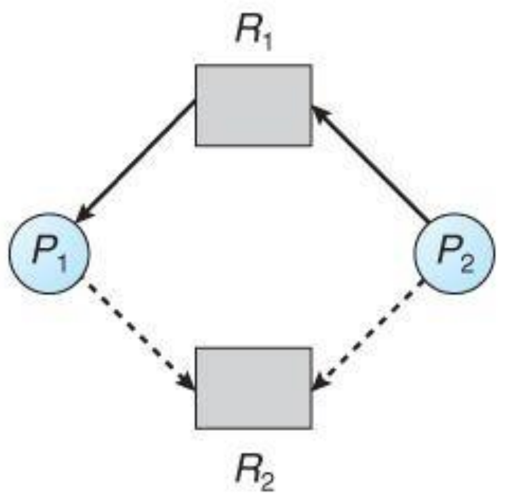

Fig 5: Resource-allocation graph for deadlock avoidance.

In our example, suppose P_2 requests R_2.

Although R_2 is currently free, but we cannot allocate it to P_2, since this action will create a cycle in the graph.

A cycle, as mentioned, 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 resource-allocation-graph algorithm is not applicable to a resource allocation system with multiple instances of each resource type.

The Banker's algorithm is a deadlock-avoidance algorithm and is applicable to such a system.

But this algorithm is less efficient than the resource-allocation graph algorithm.

This algorithm is 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 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.

Let n is the number of processes in the system, and m is the number of resource types.

We need following data structures:

Available:

A vector of length m indicates the number of available resources of each type. $Available[j]=k$: k instances of resource type R_j are available.

Max:

An $n \times m$ matrix defines the maximum demand of each process.
$Max[i][j]=k$: The process P_i may request at most k instances of resource type R_j.

Allocation:

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

$Allocation[i][j]=k$: The process P_i is currently allocated k instances of resource type R_j.

Need:

An $n \times m$ matrix indicates the remaining resource need of each process.

$Need[i][j]=k$: The process P_i may need k more instances of resource type R_j to complete its task.

It may be noted that $Need[i][j]=Max[i][j]-Allocation[i][j]$.

Safety Algorithm

The algorithm for finding out whether or not a system is in a safe state can be described as follows:

```
1. Let $Work$ and $Finish$ be vectors of length $m$ and $n$, respectively.
   Initialize
   $Work = Available$
   $Finish[i] = false$ ; for $i = 0, 1, ..., n - 1$.
2. Find an index $i$ such that both
    a. $Finish[i] == false$
    b. $Need_i \leq Work$
    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.
```

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

Resource-Request Algorithm

Let $Request_i$ be the request vector for process P_i .

If $Request_i[j] == k$, then the process P_i wants k instances of resource type R_j.

When a request for resources is made by process P_i , 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, P_i must wait, since the resources are not available.

3. Have the system pretend to have allocated the requested resources to process P_i 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 P_i is allocated its resources.

However, if the new state is unsafe, then P_i must wait for $Request_i$, and the old resource-allocation state is restored.

Example:

Consider the system with five processes P_0, P_1, P_2, P_3 and P_4; 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 at time t_0 the resource allocation and maximum resource requirement by the processes are as follows:

Process	***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			

Find out whether the system is in safe state or not. If the system is in safe state then find out the safe sequence.

The content of the matrix ***Need*** is defined to be ***Max*** – ***Allocation***.

At time t_0 the state of system is as follows:

Process	***Allocation***			***Max***			***Need***			***Available***		
	A	***B***	***C***	***A***	***B***	***C***	***A***	***B***	***C***	***A***	***B***	***C***
P_0	**0**	**1**	**0**	**7**	**5**	**3**	**7**	**4**	**3**	3	3	2
P_1	**2**	**0**	**0**	**3**	**2**	**2**	**1**	**2**	**2**			
P_2	**3**	**0**	**2**	**9**	**0**	**2**	**6**	**0**	**0**			
P_3	**2**	**1**	**1**	**2**	**2**	**2**	**0**	**1**	**1**			
P_4	**0**	**0**	**2**	**4**	**3**	**3**	**4**	**3**	**1**			

With the help of Resource-request algorithm and safety algorithm we can find out the safe sequence if any as follows:

System situation	***Available***		
	A	***B***	***C***
At time t_0	3	3	2
After execution of P_1	3+2=5	3	2
After execution of P_3	5+2=7	3+1=4	2+1=3
After execution of P_4	7	4	3+2=5
After execution of P_0	7	4+1=5	5
After execution of P_2	7+3=10	5	5+2=7

We claim that the system is currently in a safe state.

Indeed, the sequence <P_1, P_3, P_4, P_0 *and* P_2> satisfies the safety criteria.

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:

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

ii) An algorithm to recover from the deadlock.

It can be noted that a detection-and-recovery scheme requires overhead that includes not only the run-time costs of maintaining the necessary information and executing the deadlock detection algorithm, but also the potential losses inherent in recovering from a deadlock.

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.

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.

A deadlock exists in the system if and only if the ***wait-for graph contains a cycle***.

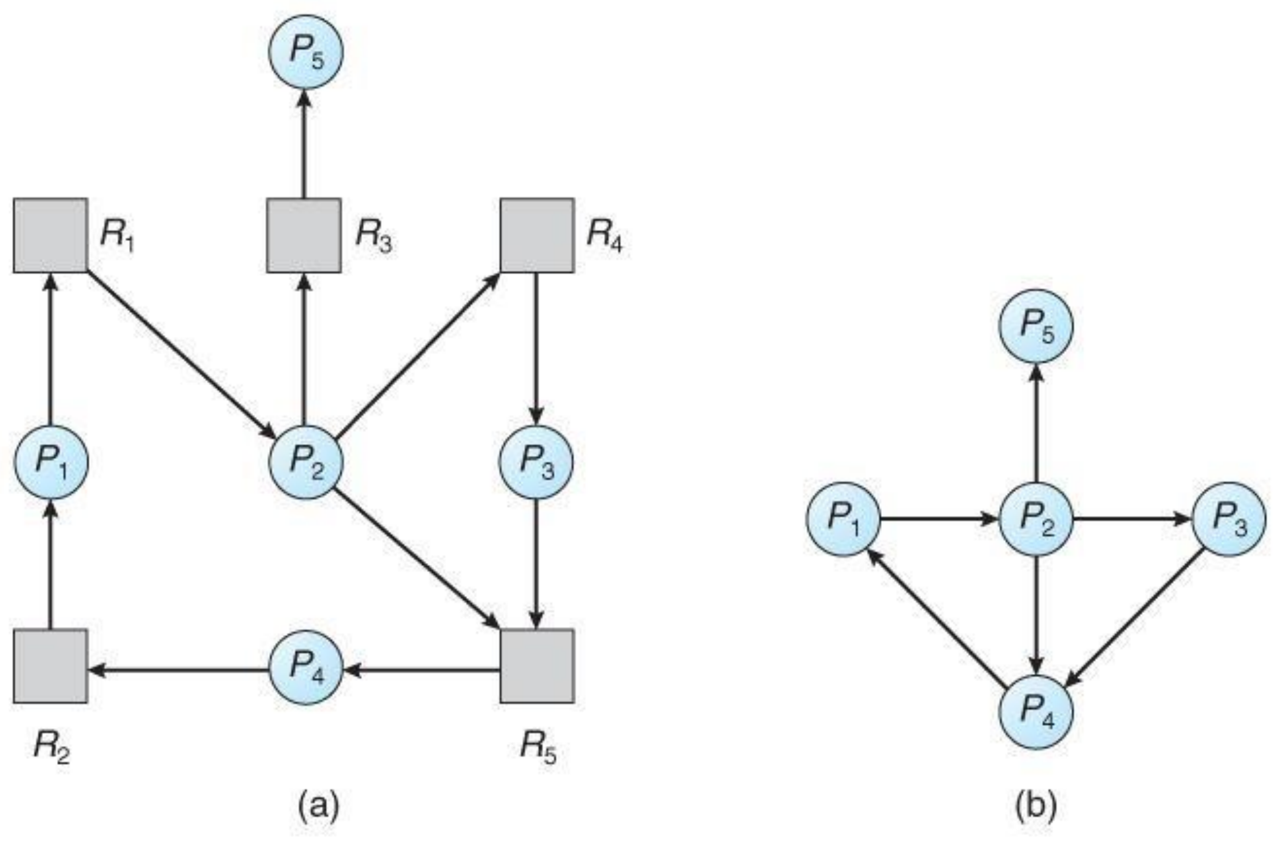

Fig 5: (a) Resource-allocation graph. (b) Corresponding wait-for graph.

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.

We need several deadlock detection algorithm (which is very similar to the banker's algorithm) to solve this problem.

The data structure of the algorithm is as follows:

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.

Request:
An $n \times m$ matrix indicates the current request of each process.

If *Request*[*i*][*j*]=*k*, then process P_i is requesting *k* more instances of resource type R_j.

The algorithm is as follows:

1. Let *Work* and *Finish* be vectors of length *m* and *n*, respectively.
   Initialize *Work* = *Available*

For $i = 0, 1, ..., n–1$ if $Allocation_i \neq 0$, then $Finish[i] = false$ Otherwise, $Finish[i] = true$

2. Find an index i such that both
   a. $Finish[i] == false$
   b. $Request_i \leq Work$

   If no such i exists, go to step 4.

3. $Work = Work + Allocation_i$ $Finish[i] = true$ Go to step 2.

4. If $Finish[i] == false$ for some i, $0 \leq i < n$, then the system is in a deadlocked state.

   If $Finish[i] == false$, then the process P_i is deadlocked.

This algorithm requires an order of $m \times n^2$ operations to detect whether the system is in a deadlocked state.

Example 1:

Consider that the state of a system in time t_0 as follows:

Process	***Allocation***			***Request***			***Available***		
	A	***B***	***C***	***A***	***B***	***C***	***A***	***B***	***C***
P_0	0	1	0	0	0	0	0	0	0
P_1	2	0	0	0	2	0			
P_2	3	0	3	0	0	0			
P_3	2	1	1	1	0	0			
P_4	0	0	2	0	0	2			

In this system, the number of instances of resource type A, B and C are 7, 2 and 6 respectively.

Justify whether the system is in a deadlock state or not.

We have applied the deadlock detection in the example.

System situation	***Available***		
	A	***B***	***C***
At time t_0	0	0	0
After execution of P_0	0	(0+1)=1	0
After execution of P_2	3	1	3
After execution of P_3	(3+2)=5	(1+1)=2	(3+1)=4
After execution of P_4	5	2	(4+2)=6
After execution of P_1	(5+2)=7	2	6

So, we can claim that the system is not in a deadlock state.

The sequence <P_0, P_2, P_3, P_4, P_1> results in *Finish*[*i*] == *true* for all *i*.

Example 2:

Consider that the state of a system in time t_0 as follows:

Process	***Allocation***			***Request***			***Available***		
	A	***B***	***C***	***A***	***B***	***C***	***A***	***B***	***C***
P_0	0	1	0	0	0	0	0	0	0
P_1	2	0	0	2	0	2			
P_2	3	0	3	0	0	1			
P_3	2	1	1	1	0	0			
P_4	0	0	2	0	0	2			

In this system, the number of instances of resource type A, B and C are 7, 2 and 6 respectively.

Justify whether the system is in a deadlock state or not.

We claim that the system is now deadlocked.

Although we can reclaim the resources held by process P_0, the number of available resources is not sufficient to fulfill the requests of the other processes.

Thus, a deadlock exists, consisting of processes P1, P2, P3, and P4.

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.

Deadlocks occur only when some process makes a request that cannot be granted immediately.

We can invoke the deadlock detection algorithm every time a request for allocation cannot be granted immediately.

In this case, we can identify not only the deadlocked set of processes but also the specific process that caused the deadlock.But invoking the deadlock-detection algorithm for every resource request will incur considerable overhead in computation time. A less expensive alternative is simply to invoke the algorithm at defined intervals—for example, once per hour or whenever CPU utilization drops below 40 percent.

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 deadlocked processes.

Process Termination

To eliminate deadlocks by aborting a process, we use one of two methods.

i) 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.

ii) 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.

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

We have to determine which resources and which processes are to be preempted.

To terminate a process, 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.

2. Rollback

We have to determine what should be done after we preempt a resource from a process.

One solution is that, we must roll back the process to some safe state and restart it from that state.

But it is difficult to determine what a safe state is, and 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.

Starvation:

We have to also notice 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, and results starvation.

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.