DEADLOCKS

Deadlock: A deadlock involving a set of process Pi in D is a situation in which

1. Every process *Pi in D blocked on some event ei.*

2. Every event ei can only caused by some process in D.

Or

Deadlock is situation in which a set process unknowingly waiting for the resources which is not available.

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. The events with which we are mainly concerned here are resource acquisition and release. The resources may be either physical resources (for example, printers, tape drives, memory space, and CPU cycles) or logical resources (for example, files, and compiler). However, other types of events may result in deadlocks

Deadlocks may also involve different resource types. For example, consider a system with one printer and one tape drive. Suppose that process *Pi* is holding the tape drive and process *Pj* is holding the printer. If *Pi* requests the printer and *Pj* requests the tape drive, a deadlock occurs.

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 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 (Po, P1, ..., *Pn)* of waiting processes must exist such that Po is waiting for a resource that is held by P1, Pl is waiting for a resource that is held by *P2,* ..., Pn-1 is waiting for a resource that is held by *Pn,* and *Pn* is waiting for a resource that is held by Po.

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 of vertices V and a set of edges E. The set of vertices V is partitioned into two different types of nodes

P = {PI, P2, ..., Pn}, the set consisting of all the active processes in the system, and R = {Rl, R2, ..., Rm}, the set consisting of all resource types in the system.

A directed edge from process Pi to resource type Rj is denoted by Pi Rj; it signifies that process Pi requested an instance of resource type Rj and is currently waiting for that resource. A directed edge from resource type Rj to process Pi is denoted by Rj Pi; it signifies that an instance of resource type Rj has been allocated to process Pi. A directed edge Pi Rj is called a **request edge;** a directed *edge Rj Pi* is called an assignment **edge.**

Pictorially, we represent each process Pi as a circle, and each resource type Rj as a square. Since resource type Rj may have more than one instance, we represent each such instance as a dot within the square. Note that a *request* edge points to only the square Rj, whereas an assignment edge must also designate one of the dots in the square.

When process Pi *requests* an instance of resource type Rj, a *request* edge is inserted in the resource-allocation graph. When this *request* can be fulfilled, the *request* edge is *instantaneously* transformed to an assignment edge. .When the process no longer needs access to the resource it releases the resource, and as a result the assignment edge is deleted.

The resource-allocation graph shown in Figure depicts the following situation.

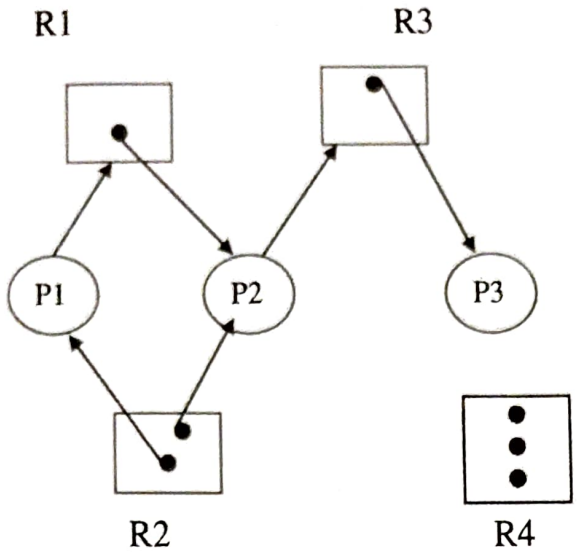
- **The sets P, R, and E:**
  P = {P1, P2, P3}
  R = {Rl, R2, R3, R4}
  E = {P1 Rl, P2 R3, Rl P2, R2 P2, R2 P1, R3 P3}

- **Resource instances:**
  One instance of resource type Rl
  Two instances of resource type R2
  One instance of resource type R3
  Three instances of resource type R4

- **Process states:**
  Processes P1 is holding an instance of resource type *R2,* and is waiting for an instance of resource type R1.

Process P2 is holding an instance of Rl and *R2,* and is waiting for an instance of resource type *R3*

Process P3 is holding an instance of *R3.*



Resource-allocation graph

Given the definition of a resource-allocation graph, it can be shown that,

If the graph contains no cycles, then no process in the system is deadlocked. If the graph does contain a cycle, then a deadlock may exist.

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.

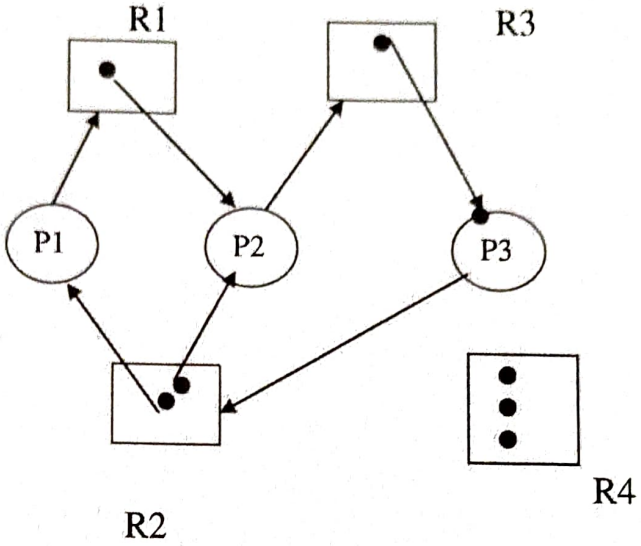
If one or more of the resource types involved in the cycle have more than one unit a knot is sufficient conditions for a deadlock occur.

Reachable Set: The reachable set of a node A is the set of all nodes B such that a path exists from A to B.

Knot: A knot is a nonempty set K of nodes such that the reachable set of each node in K is exactly the set K. A knot always contain one or more cycle.

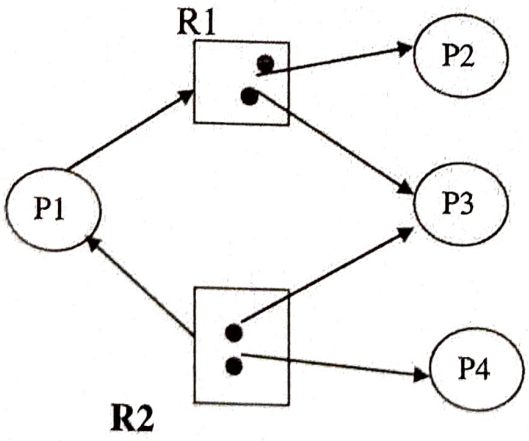
To illustrate this concept, let us return to the resource-allocation graph depicted in Figure. Suppose that process P3 requests an instance of resource type *R2*. Since no resource instance is currently available, a request edge P3 R2 is added to the graph (Figure). At this point, two minimal cycles exist in the system:

P1 - R1 - P2 - R3 - P3 - R2 - P1
P2 - R3 - P3 - R2 - P2



Resource-allocation graph with a deadlock.

Processes P1, *P2,* and P3 are deadlocked. Process P2 is waiting for the resource *R3,* which is held by process *P3.* Process *P3,* on the other hand, is waiting for either process P1 or process P2 to release resource *R2.* In addition, process P1 is waiting for process P2 to release resource R1.



Resource-allocation graph with a cycle but no deadlock.

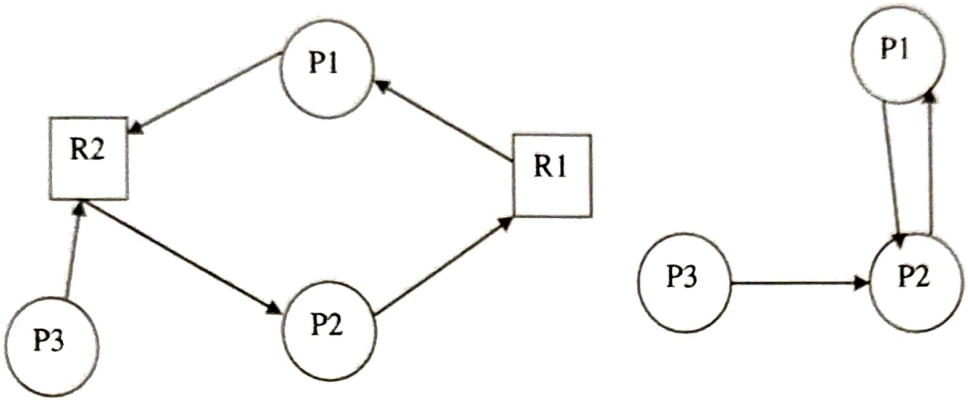
Now consider the resource-allocation graph in Figure. In this example, we also have a cycle

P1 R1 P3 R2 P1

However; there is no deadlock. Observe that process P4 may release its instance of resource type *R2.* That resource Can then be allocated to *P3,* breaking the cycle.

Wait-For Graph

When all the resource types have only a single unit each, a simplified from of resource allocation graph is normally used. The simplified graph is obtained from the original resource allocation graph by removing the resource nodes and collapsing the appropriate edges.



The simplified graph is commonly known as a wait-for a graph (WFG) because it clearly shows which process are waiting for which other process. For instance in the above WFG, process P1 and P3 waiting for P2 and process P2 is waiting for P1. Since WFG is constructed only when each resource type has only a single unit, a cycle is both necessary and sufficient condition for a deadlock in a WFG.

Methods for Handling Deadlocks

Principally, we can deal with the deadlock problem in one of three ways:

- We can use a protocol to prevent or avoid deadlocks, ensuring that the system will *never* enter a deadlock state.
- We can allow the system to enter a deadlock state, detect it, and recover.
- We can ignore the problem altogether, and pretend that deadlocks never occur in the system.

To ensure that deadlocks never occur, the system can use either deadlock prevention or a deadlock-avoidance scheme. **Deadlock prevention is a set of methods for ensuring that at least one of the necessary conditions cannot hold. These methods prevent deadlocks by constraining how requests for resources can be made.**

Deadlock avoidance, on the other hand, requires that the operating system be given in advance additional information concerning which resources a process will request and use during its lifetime. With this additional knowledge, we can decide for each request whether or not the process should wait. To decide whether the current request can be satisfied or must be delayed, the system must consider the

resources currently available, the resources currently allocated to each process, and the future requests and releases of each process.

Deadlock Prevention

By ensuring that at least one of these conditions cannot hold, we can *prevent* the occurrence of a deadlock. Let us elaborate on this approach by examining each of the four necessary conditions separately.

1. Mutual Exclusion

The mutual-exclusion conditions hold for non-sharable resources. For example, a printer cannot be simultaneously shared by several processes. **Sharable resources, on the other hand, do not require mutually exclusive access, and thus cannot be involved in a deadlock. Read only file are a good example of a sharable resource.** If several processes attempt to open a read-only file at the same time, they can be granted simultaneous access to the file. A process never needs to wait for a sharable resource. In general, however, we cannot prevent deadlocks by denying the mutual-exclusion condition: Some resources are intrinsically non-sharable.

2. Hold and Wait (All Request together or Collective requests)

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. We can implement this provision by requiring that system calls requesting resources for a process precede all other system calls.**

The validity constraints on a resource requests is that a process must make its entire request together-typically at the starts of the execution.

These protocols have two main disadvantages. **First, resource utilization may be low**, since many of the resources may be allocated but unused for a long period. Consider two process P1 and P2 such that P1 needs a tape device and a printer, while P2 needs only printer. It is possible that P1 require a tape at the beginning of the execution and printer at the end of the execution. However it will forced to request both tape and printer at the beginning of the execution. This will simply idle the printer at the beginning of the execution.

Second, starvation is possible. A process that needs several popular resources may have to wait indefinitely, because at least one of the, resources that it needs is always allocated to some other process.

3. No Preemption

The third necessary condition is that there be no preemption of resources that have already been allocated. 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.** 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

4 Circular Wait (Resource Ranking)

The fourth and final condition for deadlocks is the circular-wait condition. **Wait 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 = {R1, R2, ..., *Rm}* be the set of resource types. We assign to each resource type a unique integer number, which allows us to compare two resources and to determine whether one precedes another in our ordering.

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, say *Ri.* After that, the process can request instances or resource type Rj if and only if *F (Rj) > F (Ri).* If several instances of the same recourse type are needed, a *single* request for all of them must be issued. For example, using the function defined previously, a process that wants to use the tape drive and printer at the same time must first request the tape drive and then request the printer.

Alternatively, we can require that, whenever a process requests an instance of resource type *Rj,* it has released any resources *Ri* such that *F (Ri) >= F (Rj).*

We can demonstrate this fact by assuming that a circular wait exists (proof by contradiction). Let the set of processes involved in the circular wait be {Po, P1, ..., *Pn},* where *Pi* is waiting for a resource *Ri,* which is held by process Pi+l. Then, since process Pi+l is holding resource *Ri* while requesting resource Ri+l, we must have *F (Ri)* < F (Ri+l), for all *i.* But this condition means that F (Ro) < F (Rl) < ... < *F (Rn)* < F (Ro). By transitivity, *F (Ro)* < F (Ro), which is impossible. Therefore, there can be no circular wait.

Deadlock Avoidance

Deadlock-prevention algorithms, by restraining how requests can be made. The restraints ensure that at least one of the necessary conditions for deadlock cannot

occur, and, hence, that deadlocks cannot hold. **Possible side effects of preventing deadlocks by this method however, are low device utilization and reduced system throughput.**

An alternative method for avoiding deadlock is to require additional information about how resources are to be requested. For example, in a system with one tape drive and one printer, we might be told that process *P* will request first the tape drive, and later the printer, before releasing both resources. Process Q, on the other hand, will request first the printer, and then the tape drive. With this 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. Each request requires that 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.

1. Safe State

A request is safe request at time T if after granting a request at time T there exists at least one sequence of allocation and deal location all the process in the system can complete

A state is *safe* if the system can allocate resources to each process (up to its maximum) some order arid still avoid a deadlock. More formally, a system is in a safe state only if there exists a safe sequence. A sequence of processes <P1, *P2*, ..., *Pn*> is a safe sequence for current allocation state if for each *Pi*,. the resources that *Pi* can still request can be satisfied by the currently available resources plus the resources held by all the *Pj*, with $j < i$, In this situation, if the resources that process *Pi* needs are not immediately available, then *Pi* can wait until all *Pj* have finished. When they have finished, Pi can obtain all of its needed resources, complete its designated task, return its allocated resources, and terminate. When *Pi* terminates, *Pi+1 can* obtain its needed resources, and so on. If no such sequence exists, then the system state is said to *be unsafe*

A safe state is not a deadlock state. Conversely, a deadlock state is an unsafe state. Not all unsafe states are deadlocks; however unsafe state *may* lead to a deadlock. As long as the state is safe, the operating system cannot avoid unsafe (and deadlock) states. In an unsafe state, the operating system cannot prevent processes from requesting resources such that a deadlock occurs: The behavior of the processes controls unsafe states.

To illustrate, we consider a system with 12 magnetic tape drives and 3 processes: *Po, P1,* and *P2*. Process P0 requires 10 tape drives, process *P1* may need as many as 4, and process *P2* need up to 9 tape drives: Suppose that, at time *to,* process *Po* is holding 5 tapes drives, process *P1* is holding 2, and process *P2* is holding 2 tape drives. (Thus, there are 3 free tape drives.)

	Maximum Needs	Current Needs
Po	10	5
P1	4	2
P2	9	2

At time *to*, the system in a safe state... The sequence <*PI, Po, P2*> satisfies the safety condition, since process *P1* can immediately be allocated all its tape drives and then return them (the system will then have 5 available tape drives), then process *Po* can get all its tape drives and return them (the system will then have 10 available tape drives), and finally process *P2* could get all its tape drives and return them (the system will then have all 12 tape drives available).

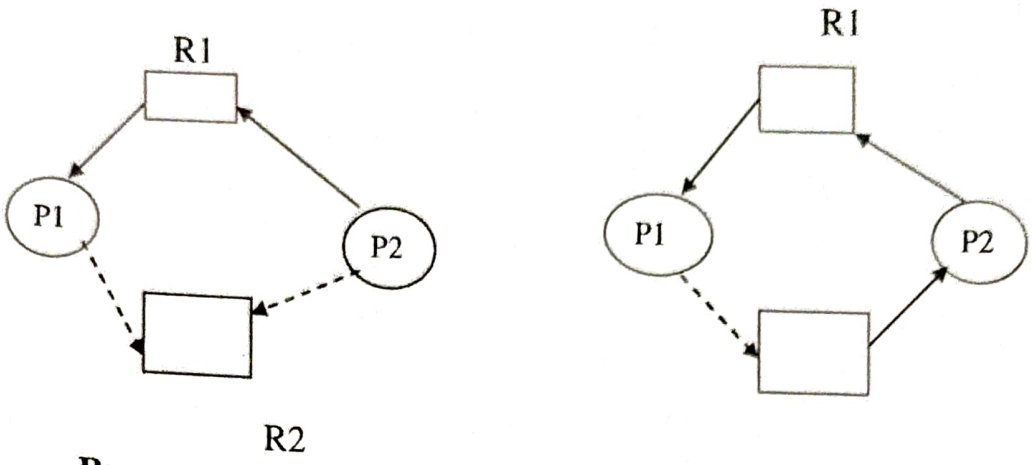
A system may go from a safe state to an unsafe state. Suppose that, at time tl, process P2 requests and is allocated, 1 more tape drive. The system is no longer in a safe state. At this point, only process P1 can be allocated all its tape drives. When it returns them, the system will have only 4 available tape drives. Since process Po is allocated 5 tape drives, but has a maximum of 10, it may then request 5 more tape device they are unavailable; process Po must wait. Similarly, process P2 may request an additional 6 tape drives and have to wait, resulting in a deadlock. .

Our mistake was in granting the request from process P2 for 1 more tape drive. If we had made P2 wait until either of the other processes had finished and released its resources, then-we could have avoided the deadlock.

2. Resource-Allocation Graph Algorithms

If we have a resource-allocation system with only one instance of each resource type, a variant of the resource-allocation graph can be used for deadlock avoidance.
In addition to the request and assignment edges, we introduce a new type of edge, **called a claim edge.** A claim edge Pi *Ri* indicates that process Pi may request resource Rj at some time in the future. This edge resembles request edge in direction, but is represented by a dashed line. When process *Pi* requests resource *Rj*, the claim edge Pi *Rj* is concerted to a request edge. Similarly, when a resource *Rj* is released by *Pi, a*ssignment edge *Rj* *Pi is* reconverted to a claim edge *Pi* *Rj*. We note that the resources must be claimed a priori in the system. That is, before process *Pi* starts executing, all its claim edges must already appear in the resource-allocation graph. We can relax this condition by allowing a claim edge. *Pi* *Rj* to be added to the graph only if all the edges associated with process *Pi* is claim edges.

Suppose that process *Pi* requests resource *Rj*. The request can be granted only if converting the request edge *Pi* *Rj* to an assignment edge *Rj* *Pi* does not result in the formation of a cycle in the resource-allocation graph. Note that we check for safety by using a cycle-detection algorithm. An algorithm for detecting a cycle in this graph requires an order of n^2 operations, where n is the number of processes in the system.



Resource-allocation graph for deadlock avoidance. **R2**

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. Therefore, process *Pi* will have to wait for its requests to be satisfied.

To illustrate this algorithm, we consider the resource-allocation graph of Figure 1. Suppose that P2 requests *R2*. Although R2 is currently free, we cannot allocate it to *P2*, since this action will create a cycle in the graph (Figure 2). A cycle indicates that the system is in an unsafe state. If P1 requests *R2*, and P2 requests R1, then a deadlock will occur.

3 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 this 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 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 structures 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 types. We need the following data structures:

Available: A vector of length *m* indicates the number of available resources of each type. If *Available[i,j]* = k there are *k* instances of resource type Rj available.

. **Max:** An *n x m* matrix defines the maximum demand of each process. If *Max [i,j] = k,* then process Pi; may request at most *k* instances of resource type *Rj.*

Allocation: An *n* x *m* matrix defines the number of resources of each type currently allocated to each process. If *Allocation [i,j] = k,* then process Pi is currently allocated *k* instances of resource type *Rj*

. **Need**: An *n* x *m* matrix indicates the remaining resource need of each process. If *Need [i, j] = k,* then process Pi may need *k* more instances of resource type Rj to complete its task.

Note that ***Need [i,j] = Max[i,j] +Allocation [i,j].***

1. 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* and *Finish[i]:= false* for i = 1 to n.

2. Find an i such that both
   a. *Finish[i] = false*
   b. *Need <= Work.*
   If no such *i* exists, go to step 4.

3. *Work: = Work + Allocation*
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* x n^2 operations to decide whether a state is safe.

2 Resource-Request Algorithms

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

1. If *Request i* <= *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, Pj 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.

An Illustrative Example

Consider a system with five processes Po through P4 and three resource types *A*, B, C. Resource type *A* has 10 instances, resource type *B* has 5 instances, and resource type C has 7 instances. Suppose that, at time *To*, the following snapshot of the system has been taken:

	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 **<P1, *P3*,**

***P4, P2,* Po>** 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, (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 <P1, *P3, P4,* Po, *P2*> satisfies our safety requirement. Hence, we can immediately grant the request of process Pl.

You should be able to see, however, that when the system is in this state, a request for (3, 3, 0) by P 4 cannot be granted, since the resources are not available.

A request for (0, 2, 0) by Po cannot be granted, even though the resources are available, since the resulting state is unsafe.

Deadlock Detection

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

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

1. Single Instance of Each Resource Type

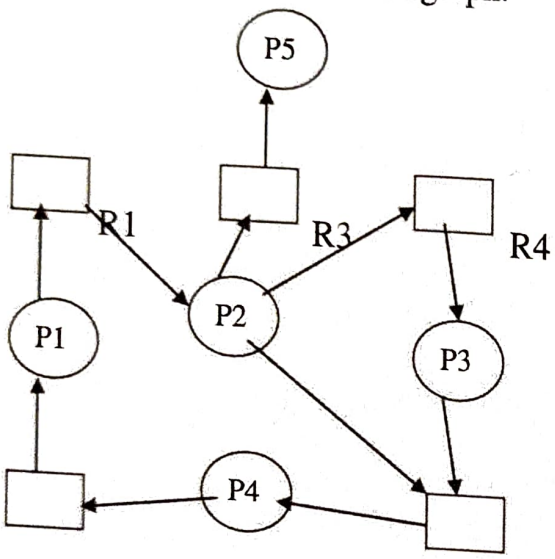
If all resources have only a single instance, then we can define a deadlock detection algorithm that uses a variant of the resource-allocation graph, called a ***wait-for* graph**. We obtain this graph from the resource allocation graph by removing the nodes of type resource and collapsing the appropriate edges.

More precisely, an edge from *Pi* to Pj in a wait-for graph implies that process *Pi* is waiting for process Pj to release a resource that *Pi* needs. An edge *Pi* ---> Pj exists in await-for graph if and only if the corresponding resource allocation graph contains

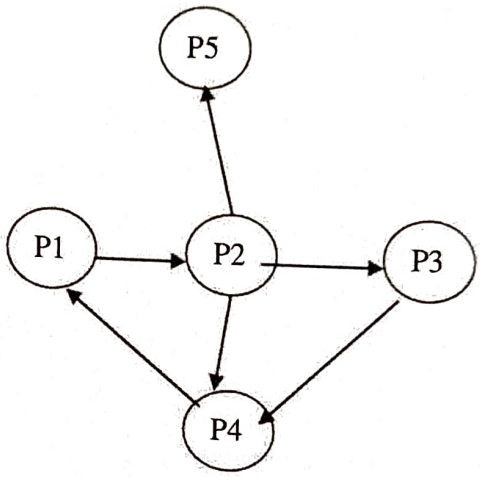
two edges *Pi* ---> *Rq* and *Rq* ---> Pj for some resource *Rq*. For example, in Figure we present a resource-allocation graph and the corresponding wait-for graph.

As before, 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 to *invoke 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.



(a) Resource-allocation graph.



(b) Corresponding wait-for graph.

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. The deadlock-detection algorithm that we describe next 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:

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

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

- **Request:** An *n* x *m* matrix indicates the current request of each process. If *Request [i,j] = k*, then process Pj is requesting *k* more instances of resource type *Rj*.

1. Let *Work* and *Finish* be vectors of length *m* and *n*, respectively.
Initialize *Work: = Available*. For $i = 1, 2, ..., n$,

If *Allocation i* <> *0* then *Finish[i] := false;*
Otherwise, *Finish[i]:= true.*

2. Find an index *i* such that both

a. *Finish[i] = false.*

b. *Request i<= Work.*

If no such *i* exists, go to step 4.

3. *Work: = Work + Allocation;*
Finish[i]:= true
Go to step 2.

4. If *Finish[i]* = false, for some *i*, 1<= *i*<= *n*, then the system is in a deadlock state. Moreover, if *Finish[i] = false*, then process Pi is deadlocked.

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

To illustrate this algorithm, we consider a system with five processes Po through P4 and three resource types *A*, B, C. Resource type *A* has 7 instances, resource type B has 2 instances, and resource type C has 6 instances. Suppose that, at time *To*, we have the following resource-allocation state:

	Allocation	*Request*	*Available*
	A B C	*A B C*	*A B C*
Po	*0 1 0*	*0 0 0*	0 0 0
P1	*2 0 0*	2 0 2	
P2	3 0 3	*0 0 0*	
P3	2 1 1	*1 0 0*	
P4	0 0 2	0 0 2	

We claim that the system is not in a deadlocked state. Indeed, if we execute our algorithm, we will find that the sequence **<Po, *P2, P3,* P1, *P4*>** will result in *Finish[i]* = true for all i.

Suppose now that process P2 makes one additional request for an instance of type C. The *Request* matrix is modified as follows:

	Request		
	A	B	C
P0	0	0	0
P1	2	0	2
P2	0	0	1
P3	1	0	0
P4	0	0	2

We claim that the system is now deadlocked. Although we can reclaim the resources held by process Po, 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 P 4.

Recovery from Deadlock

When a detection algorithm determines that a deadlock exists, several alternatives exist. One possibility is to inform the operator that a deadlock has occurred, and to let the operator deal with deadlock manually. The other possibility is to let the system *recover* from the deadlock automatically. **There are two options for breaking a deadlock. One solution is simply to abort one or more processes to break the circular wait. The second option is to preempt some resources from one or more of the deadlocked processes.**

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.

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

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

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

Many factors may determine 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?

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. Selecting a victim: Which resources and which processes are to be preempted.

2. Rollback.: When a resource is preempted the process cannot continue its normal execution, hence rollback the process to some safe state and restart it from that state.

3. Starvation: 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 needs to be dealt with in any practical system

Memory Management

Logical- Versus Physical-Address Space

An address generated by the CPU is commonly referred to as a logical address, whereas an address seen by the memory unit-that is, the one loaded into the memory-address register of the **memory-is commonly referred to as a physical address.**

The compile-time and load-time address-binding methods generate identical logical and physical addresses. However, the execution-time addresses binding scheme results in differing logical and physical addresses. In this case, we usually refer to the logical address as a virtual address. The set of all logical addresses generated by a program is a **logical-address space**; the **set of all physical addresses corresponding to these logical addresses is a physical-address space**. Thus, in the execution-time address-binding scheme, the logical- and physical-address spaces differ.

The run-time mapping **from virtual to physical addresses is done by a hardware device called the memory-management unit (MMU)**

The base register is now called a **relocation register**. The value in the relocation register is *added* to every address generated by a user process at the time it is sent to memory. For example, if the base is at 14000, then an attempt by the user to address location 0 is dynamically relocated to location 14000; an access to location 346 is mapped to location 14346.

The user program never sees the *real* physical addresses. The program can create a pointer to location 346, store it in memory, manipulate it, and compare it to other addresses-all as the number 346. Only when it is used as a memory address is it relocated relative to the base register. The user program deals with *logical* addresses. **The memory-mapping hardware converts logical addresses into physical addresses.**

We now have two different types of addresses: logical addresses (in the range 0 to *max)* and physical addresses (in the range R + 0 to R + *max* for a base value *R).* The user generates only logical addresses and thinks that the process runs in locations 0 to *max.* The user program supplies logical addresses; these logical addresses must be mapped to physical addresses before they are used.

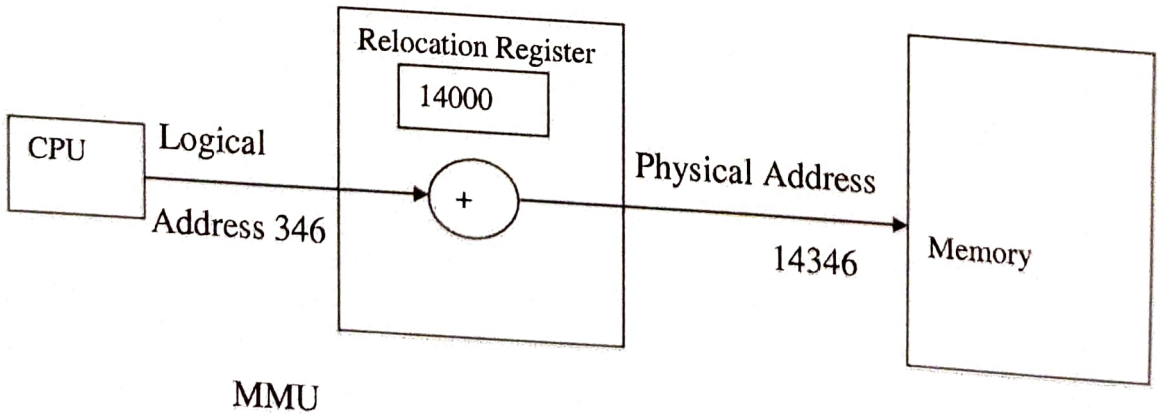


Figure **Dynamic relocation using relocation register.**

Swapping

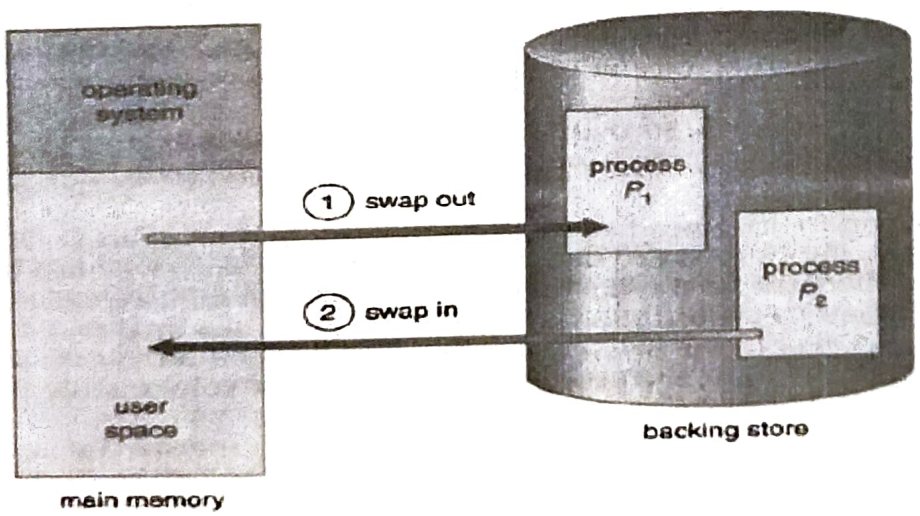
A process needs to be in memory to be executed. **A process, however, can be swapped temporarily out of memory to a backing store, and then brought back into memory for continued execution.**

Or

Swapping is the technique of temporarily removing inactive programs from the memory of a computer system.

Inactive program is one which neither executing on the CPU nor performing an I/O operation.

For example, assume a Time sharing environment with a round-robin CPU-scheduling algorithm. When a quantum expires, the memory manager will start to swap out the process that just finished, and to swap in another process to the memory space that has been freed (Figure). In the meantime, the CPU scheduler will allocate a time slice to some other process in memory. When each process finishes its quantum, it will be swapped with another process. Ideally, the memory manager can swap processes fast enough that some processes will be in memory, ready to execute, when the CPU scheduler wants to reschedule the CPU. The quantum must also be sufficiently large that reasonable amounts of computing are done between swaps.



Figure

Swapping of two processes using a disk as a backing store.

A variant of this swapping policy is used for priority-based scheduling algorithms. If a higher-priority process arrives and wants service, the memory manager can swap out the lower-priority process so that it can load and execute the higher-priority process. When the higher-priority process finishes, the lower-priority process can be swapped back in and continued. **This variant of swapping is sometimes called roll out, roll in.**

Swapping requires a **backing store. The backing store is commonly a fast disk. It must be large enough to accommodate copies of all memory images for all users, and it must provide direct access to these memory images.** The system maintains a ready queue consisting of all processes whose memory images are on the backing store or in memory and are ready to run. Whenever the CPU scheduler decides to execute a process, it calls the dispatcher. The dispatcher checks to see whether the next process in the queue is in memory. If not, and there is no free memory region, the dispatcher swaps out a process currently in memory and swaps in the desired process. It then reloads registers as normal and transfers control to the selected process.

Contiguous Memory Allocation

The main memory must accommodate both the operating system and the various user processes. We therefore need to allocate different parts of the main memory in the most efficient way possible. This section will explain one method, contiguous memory allocation.

We usually want several user processes to reside in memory at the same time. We therefore need to consider how to allocate available memory to the processes that are in the input queue waiting to be brought into memory. **In this contiguous memory allocation, each process is contained in a single contiguous section of memory. In contiguous memory allocation all the set of code and data about a process is stored in a single contiguous area.**

Memory Allocation

Now we are ready to turn to memory allocation. One of the simplest methods for memory allocation is to divide memory into several fixed-sized partitions Each partition may contain exactly one-process, Thus, the degree of multiprogramming is bound by the number of partitions. In this multiple-partition method, when a partition is free, a process is selected from the input queue and is loaded into the free partition. When the process terminates, the partition becomes available for another process. The method described next is a generalization of the fixed-partition scheme (called MVT); it is used primarily in a batch environment.

The operating system keeps a table indicating which parts of memory are available and which are occupied. Initially, all memory is available for user processes, and is considered as one large block of available memory, a hole. When a process arrives and needs memory, we search for a hole large enough for this process. If we find one, we allocate only as much memory as is needed, keeping the rest available to satisfy future requests,

As processes enter the system, they are put into an input queue. The operating system takes into account the memory requirements of each process and the amount of available memory space in determining which processes are allocated memory. When a process is allocated space, it is loaded into memory and it can then compete for the CPU. When a process terminates, it releases its memory, which the operating system may then fill with another process from the input queue.

At any given time, we have a list of available block sizes and the input queue. The operating system can order the input queue according to a scheduling algorithm. Memory is allocated to processes until, finally, the memory requirements of the next process cannot be satisfied; no available block of memory (or hole) is large enough to hold that process. The operating system can then wait until a large enough blocks is available, or it can skip down the input queue to see whether the smaller memory requirements of some other process can be met.

In general, a *set* of holes, of various sizes, is scattered throughout memory at any given time. When a process arrives and needs memory, the system searches this set for a hole that is large enough for this process. If the hole is too large, it is split into two: One part is allocated to the arriving process; the other is returned to the set of holes. When a process terminates, it releases its block of memory, which is then placed back in the set of holes. If the new hole is adjacent to other holes, these adjacent holes are merged to form one larger hole. At this point, the system may need to check whether there are processes waiting for memory and whether this newly freed and recombined memory could satisfy the demands of any of these waiting processes.

This procedure is a particular instance of the general dynamic storage allocation problem, which is how to satisfy a request of size *n* from a list of free holes. There are many solutions to this problem. The set of holes is searched to determine which hole is best to allocate. The first-fit, best-fit, and worst-fit strategies are the most common ones used to select a free hole from the set of available holes.

. ***First fit:*** Allocate the *first* hole that is big enough. Searching can start either at the beginning of the set of holes or where the previous first-fit search ended. We can stop searching as soon as we find a free hole that is large enough.

. ***Best fit:*** Allocate the *smallest* hole that is big enough. We must search the entire list, unless the list is kept ordered by size.

. ***Worst fit:*** Allocate the *largest* hole. Again, we must search the entire list, unless it is sorted by size. This strategy produces the largest leftover hole, which may be more useful than the smaller leftover hole from a best-fit approach.

As processes enter the system, they are put into an input queue. The operating system takes into account the memory requirements of each process and the amount of available memory space in determining which processes are allocated memory. When a process is allocated space, it is loaded into memory and it can then compete for the CPU. When a process terminates, it releases its memory, which the operating system may then fill with another process from the input queue.

At any given time, we have a list of available block sizes and the input queue. The operating system can order the input queue according to a scheduling algorithm. Memory is allocated to processes until, finally, the memory requirements of the next process cannot be satisfied; no available block of memory (or hole) is large enough to hold that process. The operating system can then wait until a large enough blocks is available, or it can skip down the input queue to see whether the smaller memory requirements of some other process can be met.

In general, a *set* of holes, of various sizes, is scattered throughout memory at any given time. When a process arrives and needs memory, the system searches this set for a hole that is large enough for this process. If the hole is too large, it is split into two: One part is allocated to the arriving process; the other is returned to the set of holes. When a process terminates, it releases its block of memory, which is then placed back in the set of holes. If the new hole is adjacent to other holes, these adjacent holes are merged to form one larger hole. At this point, the system may need to check whether there are processes waiting for memory and whether this newly freed and recombined memory could satisfy the demands of any of these waiting processes.

This procedure is a particular instance of the general dynamic storage allocation problem, which is how to satisfy a request of size *n* from a list of free holes. There are many solutions to this problem. The set of holes is searched to determine which hole is best to allocate. The first-fit, best-fit, and worst-fit strategies are the most common ones used to select a free hole from the set of available holes.

- ***First fit:*** Allocate the *first* hole that is big enough. Searching can start either at the beginning of the set of holes or where the previous first-fit search ended. We can stop searching as soon as we find a free hole that is large enough.

- ***Best fit:*** Allocate the *smallest* hole that is big enough. We must search the entire list, unless the list is kept ordered by size.

- ***Worst fit:*** Allocate the *largest* hole. Again, we must search the entire list, unless it is sorted by size. This strategy produces the largest leftover hole, which may be more useful than the smaller leftover hole from a best-fit approach.

Fragmentation

Early Multiprogramming systems used a partitioned approach to memory management. Each memory partitioned was a single contiguous area in the memory. Memory allocation is at the beginning of the execution. Thus initiating a process the OS allocate a memory partitioned larger than the size of the process. Use of this memory management model often gave rise the problem of fragmentation.

Memory fragmentation implies the existence of unusable memory areas in a computer system.

There are two kinds of fragmentation –internal and external fragmentation

There are three jobs A, B and C exits in a system's allocate 100KB of memory to Job But Job A utilized only 80KB of memory through out its execution.20KB of memory allocated to Job A becomes unused . It cannot be allocate to any other jobs. This is internal fragmentation.

Internal fragmentation arises in the system when memory area allocated to a program is not fully utilized by it.

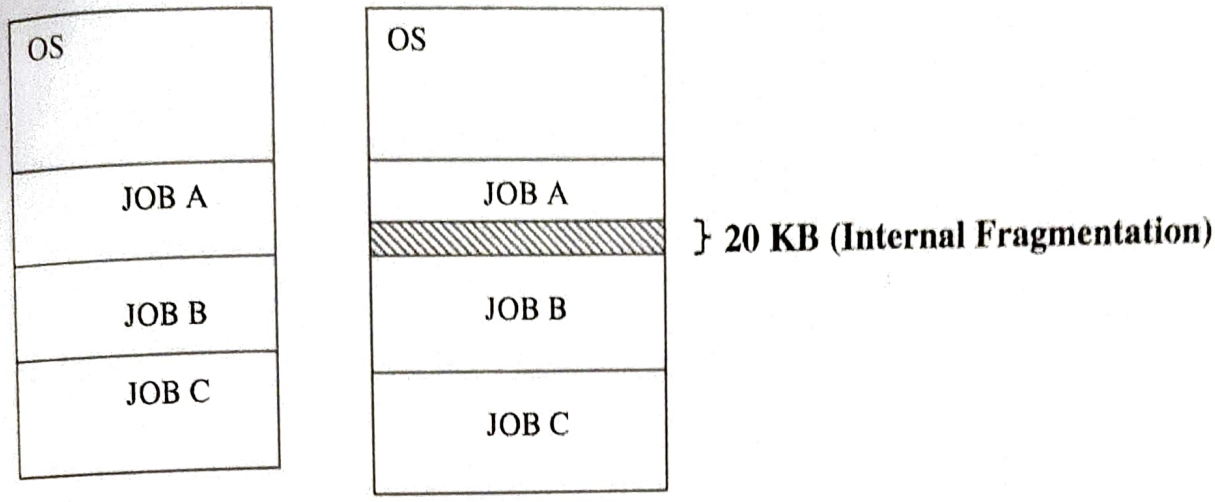
Let Job A complete and Job D whose memory requirement is 90KB is placed in its place. An area of 10Kb of free memory area exists between Job D and B. However this cannot be used because it is too small to accommodate a job. Now let Job C complete and let Job E be placed in its location. This leads another free area.(20KB). A total of 30KB of unallocated memory now exists in the system but a job, say F requiring 25KB cannot be initiated because a single contiguous free area 25KB is not available. Hence 10Kb and 20KB are unusable. This is called external fragmentation.

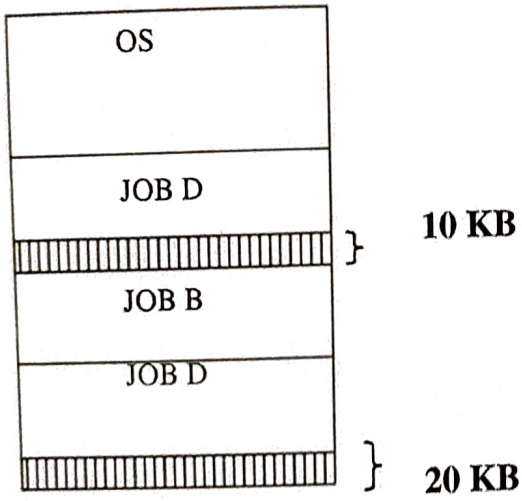
External fragmentation arises in the system when free memory area existing in a system is too small to be allocating to a job.

One solution to the problem of external fragmentation is compaction. The goal is to shuffle the memory contents to place all free memory together in one large block.

Compaction is the process of physical shifting of the process from one part of the memory to another part of the memory.

Another possible solution to the external-fragmentation problem is to permit the logical-address space of a process to be noncontiguous, thus allowing a process to be allocated physical memory wherever the latter is available. **Two complementary techniques achieve this solution: paging and segmentation.**





(10 +20)KB External Fragmentation

Paging

Paging is a memory-management scheme that permits the physical-address space of a process to be noncontiguous. Paging avoids the considerable problem of fitting the varying-sized memory chunks onto the backing store, from which most of the previous memory-management schemes suffered. When some code fragments or data residing in main memory need to be swapped out, space must be found on the backing store. The fragmentation problems discussed in connection with main memory are also prevalent with backing store, except that access is much slower, so compaction is impossible.

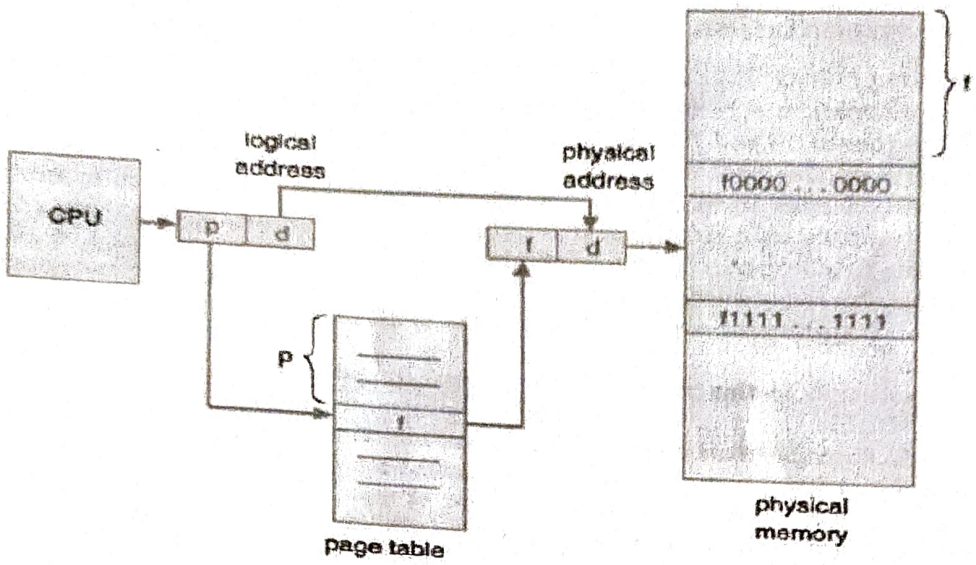


Figure 9.6 Paging hardware.

Physical memory is broken into fixed-sized blocks called frames. Logical memory is also broken into blocks of the same size called pages. When a process is to be executed, its pages are loaded into any available memory frames from the backing store. The backing store is divided into fixed-sized blocks that are of the same size as the memory frames.

The hardware support for paging is illustrated in Figure. Every address generated by the CPU is divided into two parts: a **page number (p) and a page offset (d).** The page number is used as an index into a page table. The page table contains the base address of each page in physical memory. This base address is combined with the page offset to define the physical memory address that is sent to the memory unit. The paging model of memory is shown in Figure.

The page size (like the frame size) is defined by the hardware. The size of a page is typically a power of 2, varying between 512 bytes and 16 MB per page, depending on the computer architecture. The selection of a power of 2 as a page size makes the translation of a logical address into- a page number and page offset particularly easy. If the size of logical-address space is 2^m, and a page size is 2^n addressing units (bytes or words), then the high-order ***m - n* bits of a logical address** designate the page number, and the ***n* low-order bits designate the page offset**. Thus, the logical address is as follows: