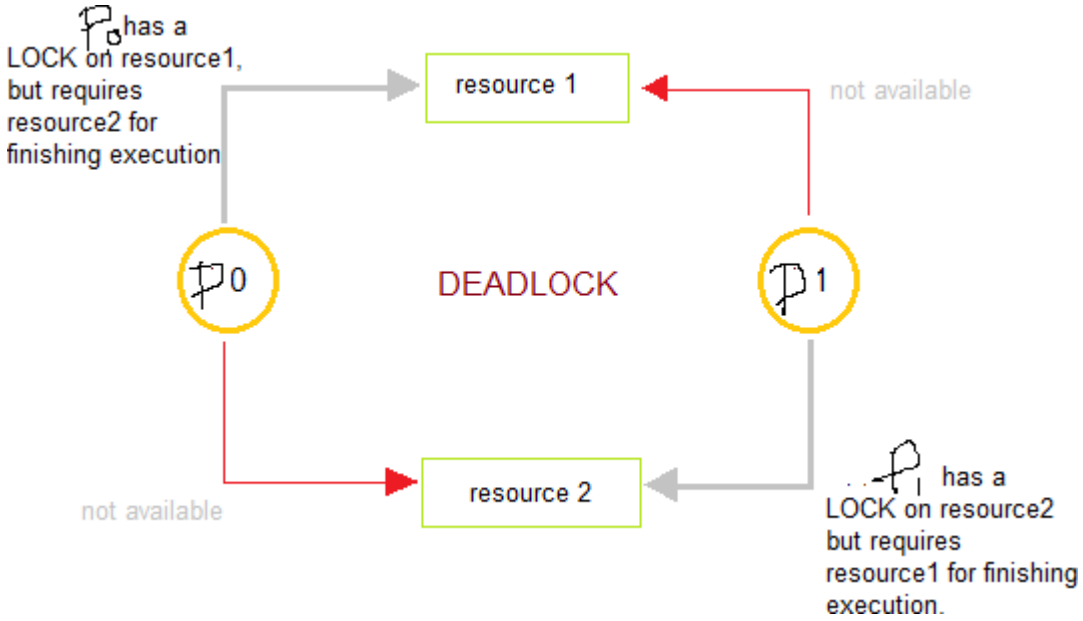
Module – 3
Deadlocks

- System model;
- Deadlock characterization;
- Methods for handling deadlocks;
- Deadlock prevention;
- Deadlock avoidance;
- Deadlock detection and recovery from deadlock.

In a multiprogramming system, numerous processes get competed for a finite number of resources. Any process requests resources and as the resources aren't available at that time, the process goes into a waiting state. At times, a waiting process is not at all able again to change its state as the resources it has requested are detained by other waiting processes. That condition is termed as deadlock.

Deadlocks are a set of blocked processes each holding a resource and waiting to acquire a resource held by another process.



4.1 System Model

A system consists of finite number of resources and is distributed among number of processes. A process must **request** a resource before using it and it must **release** the resource after using it. It can request any number of resources to carry out a designated task. The amount of resource requested may not exceed the total number of resources available.

- ✓ A process may utilize the resources in only the following **sequence**,
  1. **Request:** If the request is not granted immediately then the requesting process must wait it can acquire the resources.
  2. **Use:** The process can operate on the resource.
  3. **Release:** The process releases the resource after using it.

- ✓ To illustrate deadlock, consider a system with one printer and one tape drive. If a process Pi currently holds a printer and a process Pj holds the tape drive. If process Pi request a tape drive and process Pj request a printer then a deadlock occurs.
- ✓ Multithread programs are good candidates for deadlock because they compete for shared resources.

4.2 Deadlock Characterization

- **Necessary Conditions**

  A deadlock situation can occur if the following **4 conditions** occur simultaneously in a system.
  - **Mutual Exclusion:** At least one resource must be held in a non-sharable mode; that is, only one process at a time can use the resource. If another process requests for the resource, the requesting process must be delayed until the resource has been released.
  - **Hold and Wait:** A process must be holding at least one resource and waiting to acquire additional resources that are currently being held by the other process.
  - **No Preemption:** Resources cannot be preempted i.e., only the process holding the resources must release it after the process has completed its task.
  - **Circular Wait:** A set $\{P_0, P_1 ..... P_n\}$ of waiting process 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 resource held by process P_n and P_n is waiting for the resource held by P_0.

- **Resource Allocation Graph**

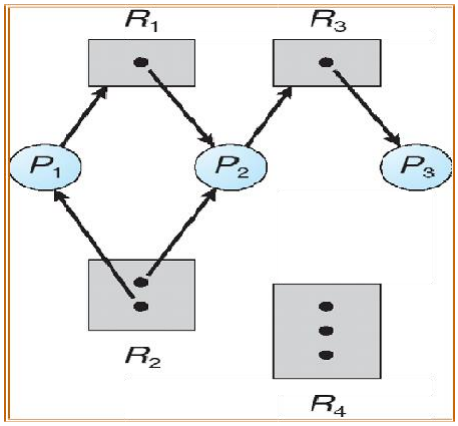
  - ✓ Deadlocks are described by using a directed graph called **system resource allocation graph**. The graph consists of set of **vertices (V)** and set of **edges (E)**.
  - ✓ The set of vertices (V) can be described into two different types of nodes,
  - ✓ $P=\{ P_1, P_2 ..... P_n\}$ a set consisting of all active processes and $R=\{R_1, R_2 ..... R_n\}$ a 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 indicates that P_i has requested an instance of resource type R_j and is waiting for that resource. This edge is called **Request edge.**
  - ✓ A directed edge $R_j \rightarrow P_i$ signifies that an instance of resource type R_j has been allocated to process P_i. This is called **Assignment edge.**
  - ✓ Process P_i is represented as **circle** and each resource type R_j as a **rectangle**.
  - ✓ Since resource type R_j may have more than one instance, we represent each such instance as a **dot** within the rectangle.
  - ✓ Request edge points to only the rectangle R_j, whereas an assignment edge must also designate one of the dots in the rectangle.
  - ✓ The below **figure 7.1** shows the Resource allocation graph which denotes,
  - ✓ The sets P, R and E:

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

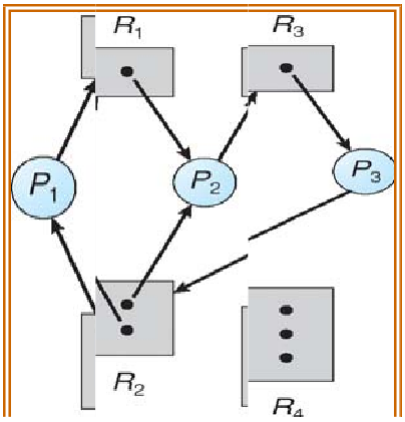
    $R = \{R_1, R_2, R_3, R_4\}$

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

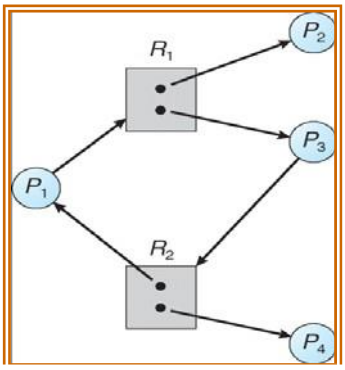
- Process P_1 is holding an instance of resource type R_2 and is waiting for an instance of resource type R_1.
- Process P_2 is holding an instance of R_1 and an instance of R_2 and is waiting for an instance of R_3.
- Process P_3 is holding an instance of R_3.



- ✓ If the graph contains no cycle, then no process in the system is deadlocked. If the graph contains 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 each resource type has several instances then a cycle do not necessarily implies that a deadlock has occurred.
- ✓ Consider the resource-allocation graph shown in **figure 7.1**.
- ✓ Suppose, process P_3 requests an instance of resource type R_2. Since no resource instance is currently available, a request edge $P_3 \circledcirc R_2$ is added to the graph which results in below **figure 7.2**.



- ✓ Now, two minimal cycles exist in the system:
  - $P_1 \circledcirc R_1 \circledcirc P_2 \circledcirc R_3 \circledcirc P_3 \circledcirc R_2 \circledcirc P_1$
  - $P_2 \circledcirc R_3 \circledcirc P_3 \circledcirc R_2 \circledcirc P_2$
- ✓ Processes P_1, P_2, and P_3 are deadlocked.
- ✓ Process P_2 is waiting for the resource R_3, which is held by process P_3.
- ✓ Process P_3 is waiting for either process P_1 or process P_2 to release resource R_2.
- ✓ In addition, process P_1 is waiting for process P_2 to release resource R_1.
- ✓ Consider the resource-allocation graph in below **figure 7.3**, which also have a cycle $P_1 \circledcirc R_1 \circledcirc P_3 \circledcirc R_2 \circledcirc P_1$, but there is no deadlock.



- ✓ P_4 may release its instance of resource type R_2. That resource can then be allocated to P_3, breaking the cycle.
- ✓ If a resource-allocation graph does not have a cycle, then the system is not in a deadlocked state. If there is a cycle, then the system may or may not be in a deadlocked state.

4.3 Methods for Handling Deadlocks

- ✓ **There are three ways to deal with deadlock problem:**
  - **Protocol can be implemented** to prevent or avoid deadlocks, ensuring that the system will never enter into the deadlock state.
  - Allow a system to enter into deadlock state, **detect it and recover** from it.
  - **Ignore** the problem and pretend that the deadlock never occur in the system. This is used by most OS including UNIX.

- ✓ To ensure that the deadlock never occurs, the system can use either deadlock avoidance or deadlock prevention.
- ✓ Deadlock prevention is a set of method for ensuring that at least one of the necessary conditions does not occur.
- ✓ Deadlock avoidance requires the OS is given advance information about which resource a process will request and use during its lifetime.
- ✓ If a system does not use either deadlock avoidance or deadlock prevention then a deadlock situation may occur. In this situation the system can provide an algorithm that examines the state of the system to determine whether a deadlock has occurred and an algorithm to recover from deadlock.
- ✓ Undetected deadlock will result in deterioration of the system performance.

4.4 Deadlock Prevention

- ✓ For a deadlock to occur each of the four necessary conditions must hold. If at least one of these conditions **does not hold** then we can prevent occurrence of deadlock.

  - **Mutual Exclusion:** This holds for non-sharable resources. For ex, A printer can be used by only one process at a time. Mutual exclusion is not possible in sharable resources and thus they cannot be involved in deadlock. Read-only files are good examples for sharable resources.

  - **Hold and Wait:** This condition can be eliminated by forcing a process to release all its resources held by it when it requests a resource. Two possible **solutions(protocols)** to achieve this are,
    - One protocol can be used is that each process is allocated with all of its resources before it starts execution.

- Another protocol that can be used is to allow a process to request a resource when the process has none.
- To illustrate the difference between these two protocols, we consider a process that copies data from a DVD drive to a file on disk, sorts the file, and then prints the results to a printer. If all resources must be requested at the beginning of the process, then the process must initially request the DVD drive, disk file, and printer. It will hold the printer for its entire execution, even though it needs the printer only at the end.
- The second method allows the process to request initially only the DVD drive and disk file. It copies from the DVD drive to the disk and then releases both the DVD drive and the disk file. The process must then again request the disk file and the printer. After copying the disk file to the printer, it releases these two resources and terminates.
- Both protocols have two main **disadvantages**.
    1. First, resource utilization is low, since resources may be allocated but unused for a long period.
    2. Second, starvation is possible. A process that needs several popular resources may have to wait indefinitely, because at least one of the resources that it needs is always allocated to some other process.

- **No Preemption:** To ensure that this condition never occurs the resources must be preempted. The following protocols can be used.
    - If a process is holding some resource and request another resource that cannot be immediately allocated to it, then all the resources currently held by the requesting process are preempted and added to the list of resources for which other processes may be waiting. The process will be restarted only when it regains the old resources and the new resources that it is requesting.
    - When a process request resources, we check whether they are available or not. If they are available we allocate them else we check that whether they are allocated to some other waiting process. If so we preempt the resources from the waiting process and allocate them to the requesting process. Otherwise, the requesting process must wait.

- **Circular Wait:** One way to ensure that this condition never holds is to impose total ordering of all resource types and each process requests resource in an increasing order. For ex, Let $R=\{R_1,R_2,\ldots\ldots R_m\}$ be the set of resource types. We assign each resource type with a unique integer value. This allows us to compare two resources and determine whether one precedes the other in ordering. We can define a one to one function F:R©N as follows,

  F(disk drive)=5, F(printer)=12, F(tape drive)=1

  Deadlock can be prevented by using the following protocols.
    - Each process can request the resource in **increasing order**. A process can request any number of instances of resource type say R_i and it can request instances of resource type R_j only $F(R_j) > F(R_i)$.
    - Alternatively when a process requests an instance of resource type R_j, it has released any resource R_i such that $F(R_i) >= F(R_j)$.

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

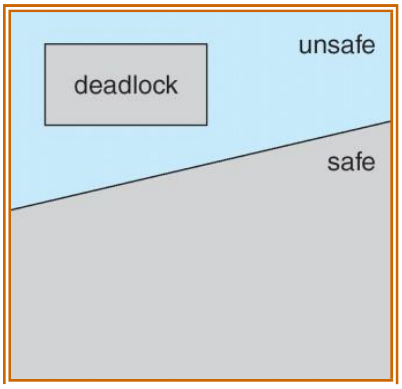
4.5 Deadlock Avoidance

- ✓ Deadlock prevention algorithm may lead to low device utilization and reduces system throughput.
- ✓ Avoiding deadlocks requires additional information about how resources are to be requested. With the knowledge of the complete sequences of requests and releases we can decide for each requests whether the process should wait or not.

- ✓ For each requests it requires checking of the resources **currently available**, resources that are **currently allocated** to each processes, future **requests and release** of each process to decide whether the current requests can be satisfied or must wait to avoid future possible deadlock.
- ✓ A deadlock avoidance algorithm dynamically examines the resources allocation state to ensure that a circular wait condition never exists. The resource allocation state is defined by the number of available and allocated resources and the maximum demand of each process.

- **Safe State**

  - ✓ A state is a **safe state** in which there exists at least one order in which all the process will run completely without resulting in a deadlock.
  - ✓ A system is in safe state if there exist a **safe sequence**.
  - ✓ A sequence of processes $<P_1,P_2,\ldots\ldots P_n>$ is a safe sequence for the current allocation state if, for each P_i, the resources requests that P_i can still make and it can be satisfied by the currently available resources plus the resources held by all p_j, with $j<i$.
  - ✓ If the resources that P_i requests are not currently available then P_i can wait until all P_j have finished. When they have finished, P_i can obtain all of its needed resource to complete its designated task.
  - ✓ A safe state is not a deadlocked state. But, a deadlocked state is an unsafe state.
  - ✓ Not all unsafe states are deadlocked. An unsafe state may lead to a deadlock. It is shown in below **figure 7.4.**



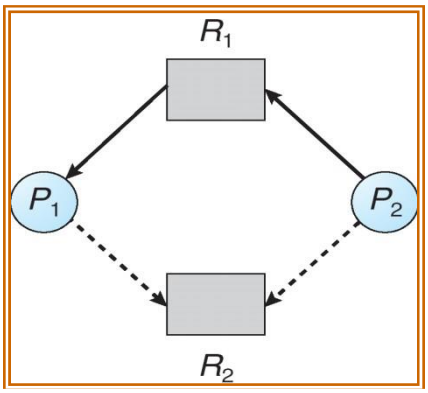
  - ✓ **For Ex**, Consider a system with 12 magnetic tape drives and three processes P_o, P_1, and P_2. Process P_o requires ten tape drives, process P_1may need as many as four tape drives, and process P_2 may need up to nine tape drives. Suppose that, at time t_o, process P_o is holding five tape drives, process P_1 is holding two tape drives, and process P_2 is holding two tape drives. (there are 3 free tape drives.)

	Maximum Needs	Current Needs (allocated)
P0	10	5
P1	4	2
P2	9	2

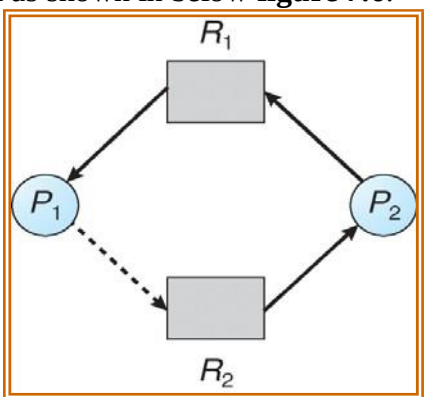
  - ✓ At time t_0, the system is in a safe state. The sequence $<P_1, P_0, P_2>$ satisfies the safety condition.
  - ✓ A system can go from a safe state to an unsafe state. Suppose that, at time t_1, process P_2 requests and is allocated one more tape drive. The system is no longer in a safe state. Only process P_1 can be allocated all its tape drives.
  - ✓ Whenever a process request a resource that is currently available, the system must decide whether resources can be allocated immediately or whether the process must wait. The request is granted only if the allocation leaves the system in safe state.

Resource Allocation Graph Algorithm

- ✓ This algorithm is used only if we have **one instance** of a resource type. In addition to the request edge and the assignment edge a new edge called **claim edge** is used. A claim edge P_i Ⓞ R_j indicates that process P_i may request R_j in future. The claim edge is represented by a **dotted line**.
- ✓ When a process P_i requests the resource R_j, the claim edge is converted to a request edge. When resource R_j is released by process P_i, the **assignment edge** R_j Ⓞ P_i is replaced by the claim edge P_i Ⓞ R_j.
- ✓ When a process Pi requests resource R_j the request is granted only if converting the request edge P_i Ⓞ R_j to as assignment edge R_j Ⓞ P_i do not result in a cycle.
- ✓ Cycle detection algorithm is used to detect the cycle. If there are no cycles then the allocation of the resource to process leave the system in safe state.
- ✓ To illustrate this algorithm, we consider the resource-allocation graph shown in below **figure 7.5**.



- ✓ Suppose that P_2 requests R_2 but we cannot allocate it to P_2 even if R_2 is currently free, because this will create a cycle in the graph as shown in below **figure 7.6**.



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

- **Banker's Algorithm**

- ✓ This algorithm is applicable to the system with **multiple instances** of each resource types, but this is less efficient than the resource allocation graph algorithm.
- ✓ When a new process enters the system it must declare the maximum number of resources that it may need. This number may not exceed the total number of resources in the system. The system must determine that whether the allocation of the resources will leave the system in a safe state or not. If it is so resources are allocated else it should wait until the process release enough resources.

- ✓ Several **data structures** are used to implement the banker's algorithm. Let 'n' be the number of processes in the system and 'm' be the number of resources types. The following data structures are needed.
  - **Available**: A vector of length m indicates the number of available resources. If Available[j]=k, then k instances of resource type R_j is available.
  - **Max**: An n*m matrix defines the maximum demand of each process. If Max[i][j]=k, then P_i may request at most k instances of resource type R_j.
  - **Allocation**: An n*m matrix defines the number of resources of each type currently allocated to each process. If Allocation[i][j]=k, then P_i is currently allocated k instances of resource type R_j.
  - **Need**: An n*m matrix indicates the remaining resources need of each process. If Need[i][j]=k, then P_i may need k more instances of resource type R_j to complete its task. So Need[i][j]=Max[i][j]-Allocation[i][j].

- **Safety Algorithm**

  - ✓ This algorithm is used to find out whether or not a system is in safe state or not. The algorithm can be described as follows,

    Step 1. Let Work and Finish be two vectors of length m and n respectively. Initialize work = available and Finish[i]=false for i=1,2,3,…….n

    Step 2. Find i such that both

    Finish[i]==false

    $Need_i$ <= Work

    If no such i exists, then go to step 4

    Step 3. Work = Work +Allocation

    Finish[i]=true

    Go to step 2

    Step 4. If Finish[i]==true for all i, then the system is in safe state.
  - ✓ This algorithm may require an order of $m*n^2$ operation to decide whether a state is safe.

- **Resource Request Algorithm**

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

    - If $Request_i$ <= $Need_i$, go to step 2. Otherwise raise an error condition, since the process has exceeded its maximum claim.
    - If $Request_i$ <= Available, go to step 3. Otherwise P_i must wait, since the resources are not available.
    - The system pretend to have allocated the requested resources to process P_i, then modify 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 complete and P_i is allocated its resources. If the new state is unsafe, then P_i must wait for $Request_i$ and old resource allocation state is restored.

- **An Illustrative Example**

  - ✓ To illustrate the use of the banker's algorithm, consider a system with five Processes P_o through P_4 and three resource types A, B, and C. Resource type A has ten instances, resource type B has five instances, and resource type C has seven instances. Suppose that, at time T_0 , the following snapshot of the system has been taken.

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

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

	Need
	A B C
P_0	7 4 3
P_1	1 2 2
P_2	6 0 0
P_3	0 1 1
P_4	4 3 1

P_0 --◎ 7 4 3 <= 3 3 2 is false,
P_1 --◎ 1 2 2 <= 3 3 2 is true, so work=work + allocation
work=3 3 2 + 2 0 0=5 3 2
P_2 --◎ 6 0 0 <= 5 3 2 is false,
P_3 --◎ 0 1 1 <= 5 3 2 is true, so work=5 3 2 + 2 1 1=7 4 3
P_4 --◎ 4 3 1<= 7 4 3 is true, so work=7 4 3 + 0 0 2=7 4 5
P_2 --◎ 6 0 0 <= 7 4 5 is true, so work=7 4 5 + 3 0 2=10 4 7
P_0 --◎ 7 4 3 <= 10 4 7 is true, so work=10 4 7 + 0 1 0=10 5 7

- ✓ We claim that the system is currently in a safe state.
- ✓ The sequence < P_l, P_3, P_4, P_2, P_o> satisfies the safety criteria.
- ✓ Suppose now the process P_1 requests one additional instance of resource type A and two instances of resource type C, so $Request_1$ = (1,0,2).
- ✓ To decide whether this request can be immediately granted, we first check that **$Request_1$ <= $Need_1$**, that is, **(1,0,2) <= (1,2,2)**, which is true then, **$Request_1$ <= $Available_1$**, that is, **(1,0,2) <= (3,3,2)**, which is true. Then we arrive at the following new state:

	Allocation	**Need**	**Available**
	A B C	**A B C**	**A B C**
P_0	0 1 0	7 4 3	2 3 0
P_1	3 0 2	0 2 0	
P_2	3 0 2	6 0 0	
P_3	2 1 1	0 1 1	

P_4	0 0 2	4 3 1

- ✓ Now we must determine whether this new system state is safe. We execute safety algorithm and find that the sequence <P_1, P_3, P_4, P_o, P_2> satisfies the safety requirement. Hence the request can be immediately granted.
- ✓ From Resource Request Algorithm, we must also see 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,

  i.e., (3,3,0)<=(4,3,1)……true

  (3,3,0)<=(2,3,0)……false
- ✓ Similarly, a request for (0,2,0) by P_o cannot be granted, even though the resources are available, because the resulting state is unsafe.

  i.e., (0,2,0)<=(7,4,3)……true

  (0,2,0)<=(2,3,0)…… true
- ✓ Now the snapshot changes as follows,

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

- ✓ Then we are supposed to apply safety algorithm to this snapshot, but no safe sequence is generated, hence the request for (0,2,0) by P_o cannot be granted.

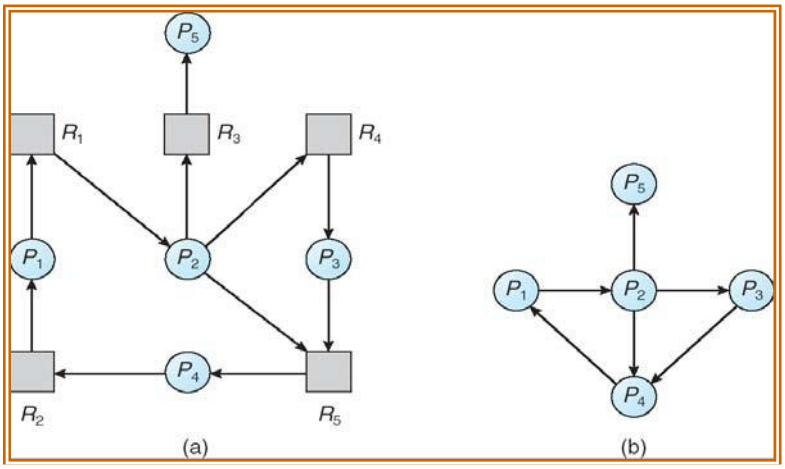
4.6 Deadlock Detection

If a system does not employ either 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.

- **Single Instances of each Resource Type**

  - ✓ If all the resources have only a single instance then we can define deadlock detection algorithm that uses a **variant of resource allocation graph** as shown in below **figure 7.7(a)** called a **wait-for graph** as shown in below **figure 7.7(b)**. This graph is obtained by removing the resource nodes and collapsing appropriate edges.



- ✓ An edge from P_i to P_j in wait for graph implies that P_i is waiting for P_j to release a resource that P_i needs.
- ✓ An edge from P_i to P_j exists in wait for graph if and only if the corresponding resource allocation graph contains the edges P_i ◎R_q and R_q◎ P_j.
- ✓ Deadlock exists within the system if and only if there is a cycle. To detect deadlock the system needs an algorithm that searches for cycle in a graph.

- **Several Instances of 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 includes following time-varying data structures.
    - **Available**. A vector of length m indicates the number of available resources of each type.
    - **Allocation.** An n*m matrix defines the number of resources of each type currently allocated to each process.
    - **Request**. An n*m matrix indicates the current request of each process. If Request[i][j]=k then P_i is requesting k more instances of resources type R_j.

  - ✓ The deadlock detection algorithm can be defined as follows,

    Step 1. Let Work and Finish be vectors of length m and n respectively. Initialize Work= Available. For i=0,1,2…….....n-1, if $allocation_i$!= 0 then Finish[i]=false, else Finish[i]=true.
    Step 2. Find an index i such that both
    Finish[i]= false
    $Request_i$<=Work
    If no such i exists, go to step 4.
    Step 3. Work = Work + $Allocation_i$
    Finish[i] = true
    Go to step 2.
    Step 4. If Finish[i] == false, for some i where 0<=i<n, then a system is in a deadlock state.

  - ✓ To illustrate this algorithm, we consider a system with five processes P_0 through P_4 and three resource types A, B, and C. Resource type A has seven instances, resource type B has two instances, and resource type C has six instances. Suppose that, at time T_0, we have the following resource-allocation state:

    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 0	
P_3	2 1 1	1 0 0	
P_4	0 0 2	0 0 2	

- ✓ From the above algorithm, the sequence <P_0, P_2, P_3, P_1, P_4> will result in Finish[i]== true for all i.
- ✓ If P_2 requests an additional instance of type C, the Request matrix is modified as follows,

	Request A B C
P_0	0 0 0
P_1	2 0 1
P_2	0 0 1
P_3	1 0 0
P_4	0 0 2

- ✓ The system is now deadlocked. Even though we can reclaim resources held by process P_0, but number of available resources is not sufficient to fulfill the requests of other processes. Thus, deadlock exists, consisting of processes P_1, P_2, P_3, and P_4.

- **Detection algorithm usage**

- ✓ This algorithm helps to find,
  - How **often** a deadlock is likely to occur?
  - How **many** processes will be affected by deadlock when it happens?
- ✓ We can invoke the deadlock detection algorithm when a request for allocation cannot be granted immediately.
- ✓ If detection algorithm is invoked for every resource request, this will cause a computational time overhead. So the algorithm must be invoked less frequently.

4.7 Recovery from Deadlock

There are **two** options for breaking a deadlock,
- One is 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, **one of two methods** can be used. In both methods, the system reclaims all resources allocated to the terminated processes.
  - **Abort all deadlocked processes.** This method breaks the deadlock cycle, but at great expense.
  - **Abort one process at a time until the deadlock cycle is eliminated.** This method causes 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 is not a easy task. If the process was in the middle of updating a file, terminating it will leave that file in an incorrect state.
- ✓ If the **partial termination method** is used, then we must determine which deadlocked process (or processes) should be terminated.

- ✓ We should abort those processes whose termination will incur the minimum cost. The following **factors** are considered to **select** the process.

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 types of resources the process has used?
4. How many more resources the process needs in order to complete?
5. How many processes will need to be terminated?
6. Whether the process is interactive or batch?

- **Resource Preemption**

- ✓ To eliminate deadlocks using resource preemption, we successively preempt some resources from processes and give these resources to other processes until the deadlock cycle is broken.
- ✓ The **three issues** need to be addressed are,
  - **Selecting a victim.** Which resources and which processes are to be preempted? We must determine the order of preemption to minimize cost.
  - **Rollback.** We must roll back the preempted process to some safe state and restart it from that state.
  - **Starvation.** We must ensure that a process can be picked as a victim only a small number of times. The most common solution is to include the number of rollbacks in the cost factor.

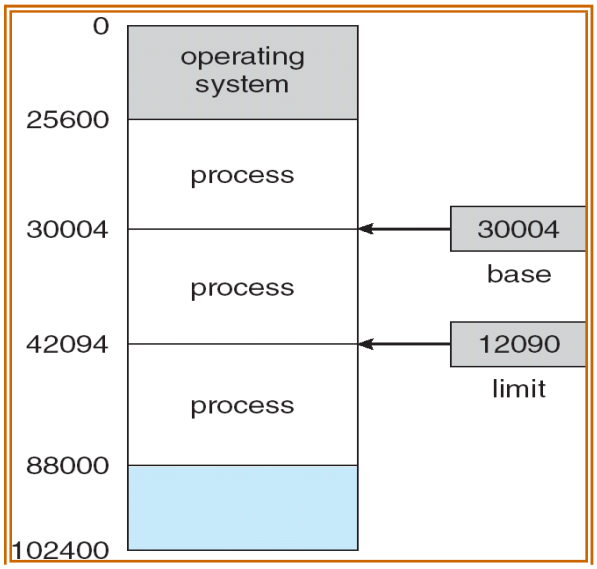
MEMORY MANAGEMENT STRATEGIES

➢ Background

- ✓ Memory management is concerned with managing the primary memory.
- ✓ Memory consists of array of bytes or words each with its own address.
- ✓ We can ignore how a program generates a memory address. We are interested only in the sequence of memory addresses generated by the running program.

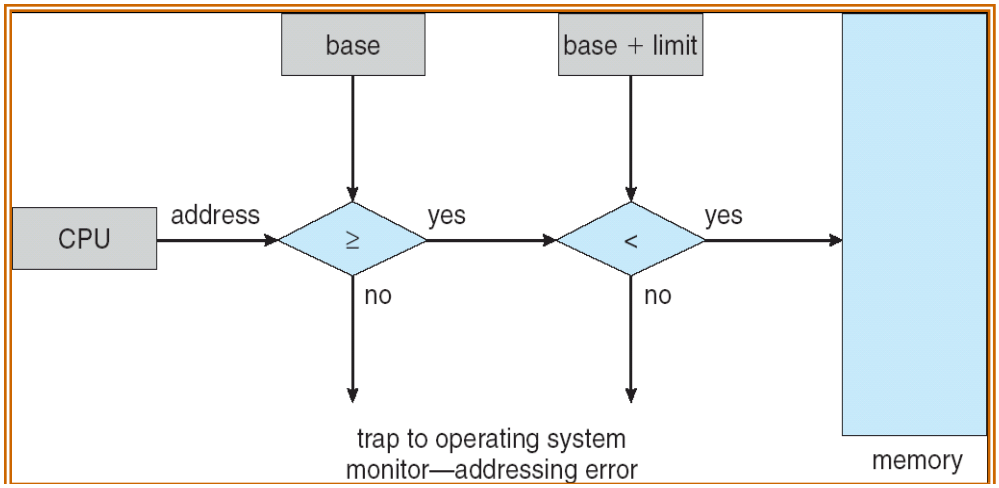
• Basic Hardware

- ✓ **Main memory and the registers** in the processor are the only storage that the CPU can access directly. Hence the program and data must be brought from disk into main memory for CPU to access.
- ✓ Registers can be accessed in **one CPU** clock cycle. But main memory access can take **many CPU** clock cycles. Hence processor needs to stall, since it does not have the data required to complete the instruction that is executing.
- ✓ To overcome above situation a fast memory called **cache** is placed between main memory and CPU registers.
- ✓ We must ensure correct operation to **protect the operating system** from access by user processes and also to protect user processes from one another. This protection must be provided by the hardware. It can be implemented in several ways and **one such possible implementation** is,
  - We first need to make sure that each process has a separate memory space.
  - To do this, we need the ability to determine the range of legal addresses that the process may access and to ensure that the process can access only these legal addresses.
  - We can provide this protection by using two registers, **a base and a limit**, as illustrated in below **figure.**



  - The **base register** holds the smallest legal **physical memory address**; the **limit register** specifies the size of **the range**. **For example**, if the base register holds 300040 and the limit register is 120900, then the program can legally access all addresses from 300040 through 420940.

- ✓ **Protection of memory space** is accomplished by having the CPU hardware compare every address generated in user mode with the registers.
- ✓ Any attempt by a program executing in user mode to access operating-system memory or other users' memory results in a **trap to the operating system**, which treats the attempt as a **fatal error** as shown in below **figure**.
- ✓ This prevents a user program from (accidentally or deliberately) modifying the code or data structures of either the operating system or other users.
- ✓ The **base and limit registers** uses a special privileged instructions which can be executed only in **kernel mode**, and since only the operating system executes in kernel mode, **only the operating system can load the base and limit registers**.

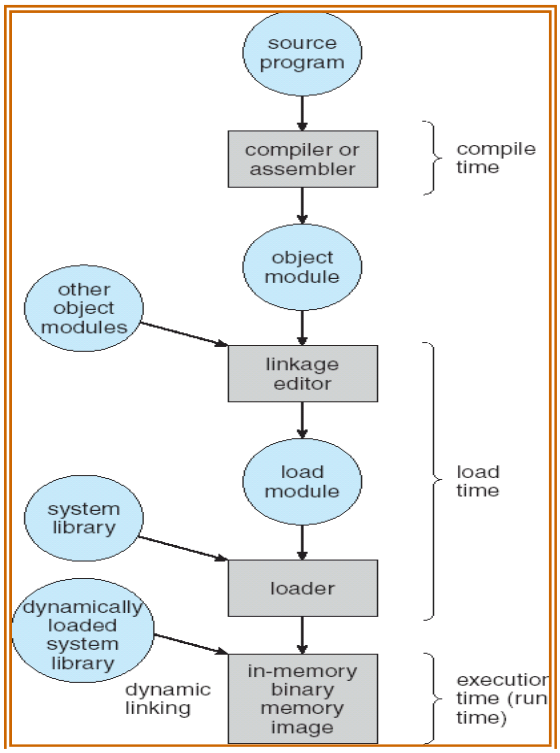


- **Address Binding**

- ✓ Programs are stored on the secondary storage disks as binary executable files.
- ✓ When the programs are to be executed they are brought in to the main memory and placed within a process.
- ✓ The collection of processes on the disk waiting to enter the main memory forms the **input queue.**
- ✓ One of the processes which are to be executed is fetched from the queue and is loaded into main memory.
- ✓ During the execution it fetches instruction and data from main memory. After the process terminates it returns back the memory space.
- ✓ During execution the process will go through several steps as shown in below **figure** and in each step the address is represented in different ways.
- ✓ In source program the address is symbolic. The compiler **binds** the symbolic address to re-locatable address. The loader will in turn bind this re-locatable address to absolute address.

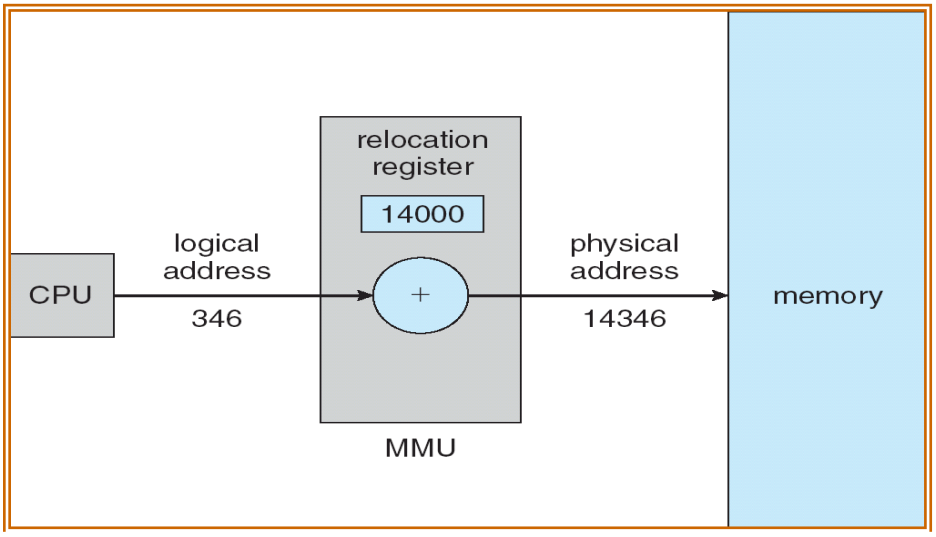
- ✓ Binding of instructions and data to memory addresses can be done at **any step** along the way:
  - **Compile time:** If we know at compile time where the process resides in memory, then **absolute code** can be generated. **For example**, if we know that a user process will reside starting at location *R*, then the generated compiler code will start at that location and extend up from there. If, at some later time, the starting location changes, then it will be necessary to recompile this code.

- **Load time:** If it is not known at compile time where the process will reside in memory, then the compiler must generate **relocatable code**. In this case, final binding is delayed **until load time.**
- **Execution time:** If the process is moved during its execution from one memory segment to another then the binding is delayed until run time. Special hardware is used for this. Most of the general purpose operating system uses this method.



- ## Logical versus physical address

✓ The address generated by the CPU is called **logical address or virtual address**.
✓ The address seen by the memory unit i.e., the one loaded in to the memory register is called the **physical address**.
✓ Compile time and load time address binding methods generate **same logical and physical address**. The execution time addressing binding generate **different logical and physical address**.
✓ Set of logical address space generated by the programs is the **logical address space**. Set of physical address corresponding to these logical addresses is the **physical address space.**
✓ The **mapping** of virtual address to physical address during run time is done by the hardware device called **Memory Management Unit (MMU)**.
✓ The **base register** is now called **re-location register**.
✓ Value in the re-location register is added to every address generated by the user process at the time it is sent to memory as shown in below **figure 8.4**.
✓ **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.



- **Dynamic Loading**

  - ✓ For a process to be executed it should be loaded in to the physical memory. The size of the process is limited to the size of the physical memory. Dynamic loading is used to obtain better memory utilization.
  - ✓ In dynamic loading the routine or procedure will not be loaded **until it is called**.
  - ✓ Whenever a routine is called, the calling routine first checks whether the called routine is already loaded or not. If it is not loaded it calls the loader to load the desired program in to the memory and updates the programs address table to indicate the change and control is passed to newly **invoked or called** routine.
  - ✓ The **advantages** are ,
    - Gives better memory utilization.
    - Unused routine is never loaded.
    - Do not need special operating system support.
    - Useful when large amount of codes are needed to handle infrequently occurring cases, such as error routines.

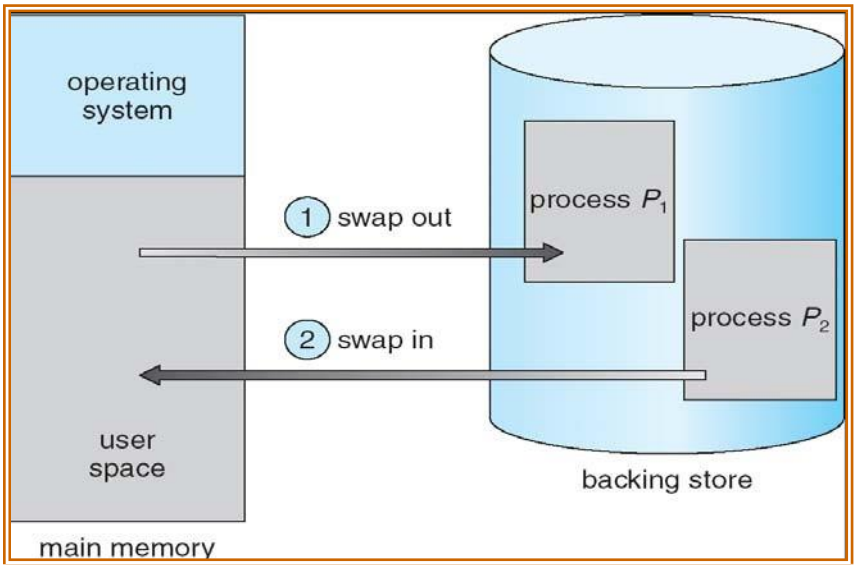
- **Dynamic linking and Shared libraries**

  - ✓ Some operating system supports only the **static linking**.
  - ✓ In dynamic linking only the main program is loaded in to the memory. If the main program requests a procedure, the procedure is loaded and the link is established at the time of references. This linking is postponed until the execution time.
  - ✓ With dynamic linking a **"stub"** is used in the image of each library referenced routine. A **"stub"** is a **piece of code** which is used to **indicate how to locate the appropriate memory resident library routine or how to load library if the routine is not already present.**
  - ✓ When "stub" is executed it checks whether the routine is present is memory or not. If not it loads the routine in to the memory.
  - ✓ This feature can be used to update libraries i.e., library is replaced by a new version and all the programs can make use of this library.
  - ✓ More than one version of the library can be loaded in memory at a time and each program uses its version of the library. Only the program that is compiled with the new version is affected by

the changes incorporated in it. Other programs linked before new version was installed will continue using older library. This system is called **"shared libraries"**.

➢ Swapping

- ✓ A process can be **swapped** temporarily out of the memory to a **backing store** and then brought back in to the memory for continuing the execution. This process is called swapping. **Ex.** In a multi-programming environment with a round robin CPU scheduling whenever the time quantum expires then the process that has just finished is swapped out and a new process swaps in to the memory for execution as shown in below **figure**.



- ✓ A variant of this swapping policy is priority based scheduling. When a low priority is executing and if a high priority process arrives then a low priority will be swapped out and high priority is allowed for execution. This process is also called as **Roll out and Roll in**.
- ✓ Normally the process which is swapped out will be swapped back to the same memory space that is occupied previously and this depends upon address binding.
- ✓ The system maintains a **ready queue** consisting of all the processes whose memory images are on the backing store or in memory and are ready to run.
- ✓ The **context-switch time** in a swapping system is **high**. **For ex,** assume that the user process is 10 MB in size and the backing store is a standard hard disk with a transfer rate of 40MB per second. The actual transfer of the 40MB process to or from main memory takes,

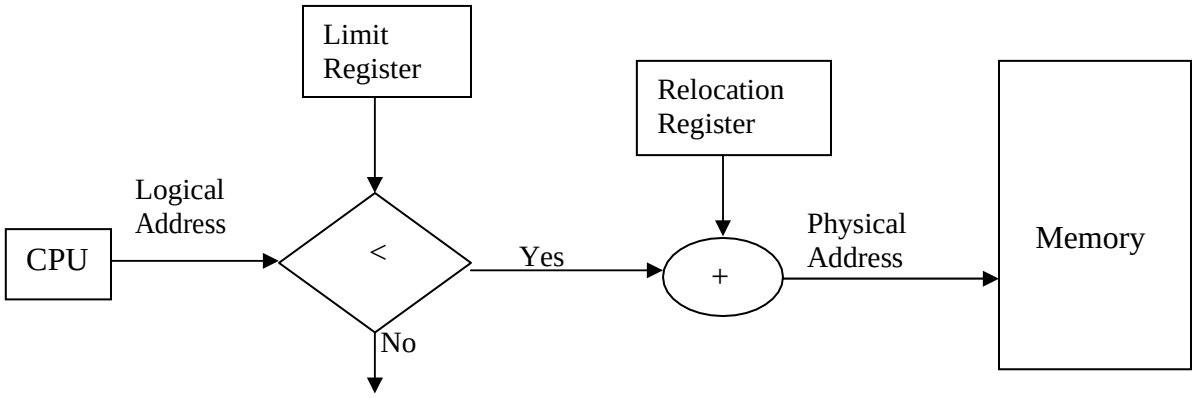
  10MB(10000KB)/40MB(40000KB) per second
  =1/4 second =250 miliseconds
- ✓ Assuming an **average latency** of **8 milliseconds**, the swap time is **258 milliseconds**. Since we must both swap out and swap in, the total swap time is about **516 milliseconds.**
- ✓ Swapping is constrained by other factors,
  - o To swap a process, it should be completely idle.
  - o If a process is waiting for an I/O operation, then the process cannot be swapped.

➢ Contiguous Memory Allocation

- ✓ The main memory must accommodate both the operating system and the various user processes. One common **method** to allocate main memory in the most efficient way is **contiguous memory allocation**.
- ✓ The memory is divided into two partitions, **one for the resident of operating system and one for the user processes.**

- **Memory mapping and protection**

  - ✓ Relocation registers are used to protect user processes from each other, and to protect from changing OS code and data.
  - ✓ The relocation register contains the value of the smallest physical address and the limit register contains the range of logical addresses.
  - ✓ With relocation and limit registers, each logical address must be less than the limit register.
  - ✓ The MMU maps the logical address dynamically by adding the value in the relocation register. This mapped address is sent to main memory as shown in below f**igure**.
  - ✓ The relocation-register scheme provides an effective way to allow the operating system's size to change dynamically.



Trap: Addressing error

- **Memory Allocation**

  - ✓ One of the simplest methods for memory allocation is to divide memory in to several **fixed partition**. Each partition contains exactly one process. The degree of multi-programming depends on the number of partitions.
  - ✓ In **multiple partition method**, when a partition is free, process is selected from the input queue and is loaded in to free partition of memory. When process terminates, the memory partition becomes available for another process.
  - ✓ The OS keeps a table indicating which part of the memory is free and is occupied.
  - ✓ Initially, all memory is available for user processes and is considered one large block of available memory called a **hole.**
  - ✓ When a process requests, the OS searches for large hole for this process. If the hole is too large it is **split** in to two. One part is allocated to the requesting process and other is returned to the set of holes.
  - ✓ The set of holes are searched to determine which hole is best to allocate.
  - ✓ **Dynamic storage allocation problem** is one which concerns about how to satisfy a request of size n from a list of free holes. There are three **strategies/solutions** to select a free hole,

- **First fit:** Allocates first hole that is big enough. This algorithm scans memory from the beginning and selects the first available block that is large enough to hold the process.
- **Best fit:** It chooses the hole i.e., closest in size to the request. It allocates the smallest hole i.e., big enough to hold the process.
- **Worst fit:** It allocates the largest hole to the process request. It searches for the largest hole in the entire list.

✓ First fit and best fit are the most popular algorithms for dynamic memory allocation. **All these algorithms suffer from fragmentation.**

- **Fragmentation**

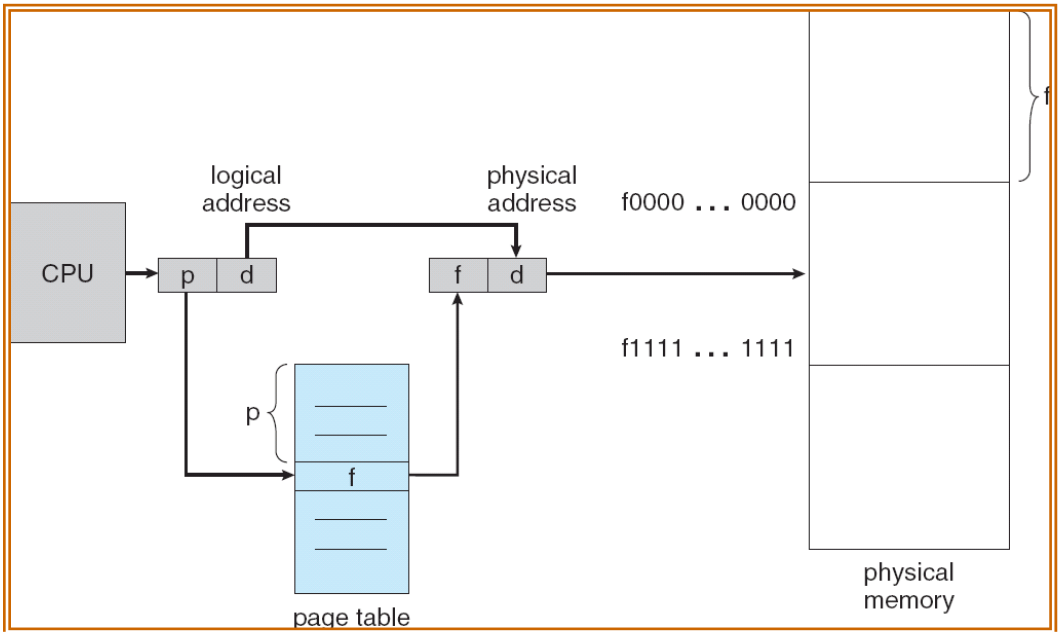
✓ **External Fragmentation** exists when there is enough memory space exists to satisfy the request, but it is not contiguous. Storage is fragmented into a large number of small holes.
✓ External Fragmentation may be either minor or a major problem.
✓ Statistical analysis of first fit reveals that, even with some optimization, given *N* allocated blocks, another 0.5 *N* blocks will be lost to fragmentation. That is, one-third of memory may be unusable. This property is known as the **50-percent rule.**
✓ **Internal fragmentation:** Memory fragmentation can be internal as well as external. Consider a multiple-partition allocation scheme with a hole of 18,464 bytes. Suppose that the next process requests 18,462 bytes. If we allocate exactly the requested block, we are left with a hole of 2 bytes. The difference between these two numbers is **internal fragmentation** that is internal to a partition.
✓ The overhead to keep track of this hole will be substantially larger than the hole itself.
✓ The general approach to avoid this problem is to break the physical memory into fixed-sized blocks and allocate memory in units based on block size. With this approach, the memory allocated to a process may be slightly larger than the requested memory.
✓ One solution to over-come external fragmentation is **compaction**. The goal is to move all the free memory together to form a large block. Compaction is possible only if the re-location is dynamic and done at execution time.
✓ **Another solution** to the external fragmentation problem is to permit the logical address space of a process to be non-contiguous.

➢ Paging

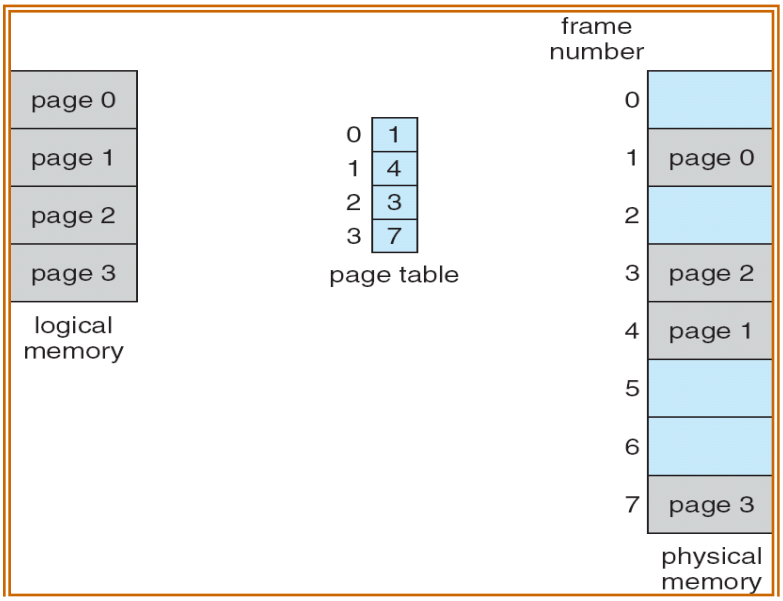
✓ Paging is a **memory management scheme** that permits the physical address space of a process to be **non-contiguous**. Support for paging is handled by **hardware.**
✓ Paging avoids the considerable problem of fitting the varying sized memory chunks on to the backing store.

- **Basic Method**

✓ Physical memory is broken in to fixed sized blocks called **frames (f)** and Logical memory(secondary memory) is broken in to blocks of same size called **pages (p).**
✓ When a process is to be executed its pages are loaded in to available frames from the backing store. The backing store is also divided in to **fixed-sized blocks of same size as memory frames.**
✓ The below **figure** shows paging hardware.



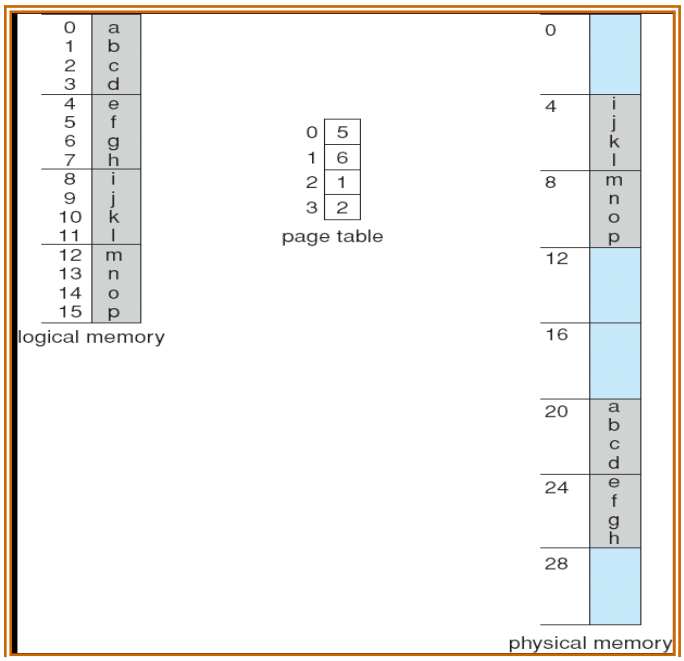
- ✓ Logical address generated by the CPU is divided in to **two parts: page number (p) and page offset (d)**.
- ✓ The page number (p) is used as **index** to the page table. The page table contains base address of each page in physical memory. This base address is combined with the page offset to define the physical memory i.e., sent to the memory unit. The paging model memory is shown in below **figure.**



- ✓ The page size is defined by the hardware. The size is the power of 2, varying between **512 bytes and 16Mb per page.**
- ✓ If the size of logical address space is $\mathbf{2^m}$ address unit and page size is 2^n, then high order **m-n** designates the **page number** and **n** low order bits represents **page offset**. Thus logic address is as follows.

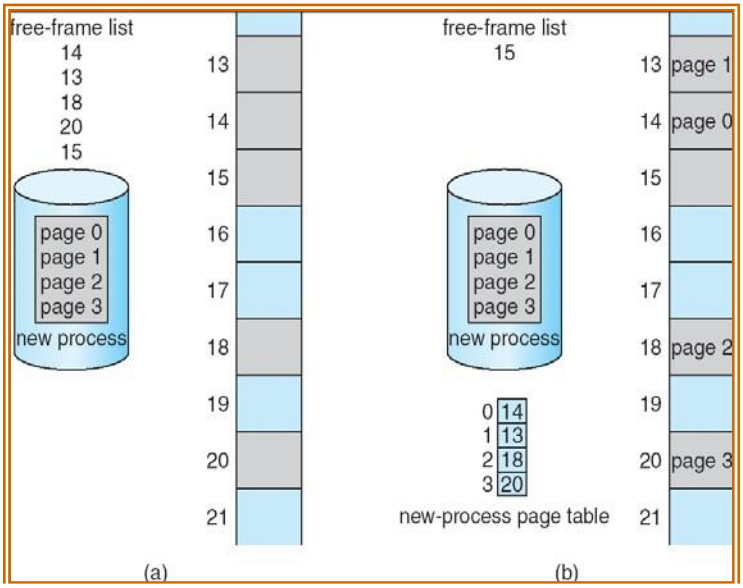
page number	page offset
p	d
m-n	n

- ✓ **Ex:** To show how to map logical memory in to physical memory, consider a page size of 4 bytes and physical memory of 32 bytes (8 pages) as shown in below **figure 8.9**.
  - **a.** Logical address 0 is page 0 and offset 0 and Page 0 is in frame 5. The **logical address 0** maps to physical address **[(5*4) + 0]=20.**
  - **b.** **Logical address 3** is page 0 and offset 3 and Page 0 is in frame 5. The **logical address 3** maps to **physical address [(5*4) + 3]= 23.**
  - **c.** **Logical address 4** is page 1 and offset 0 and page 1 is mapped to frame 6. So logical address 4 maps to **physical address [(6*4) + 0]=24**.
  - **d.** **Logical address 13** is page 3 and offset 1 and page 3 is mapped to frame 2. So logical address 13 maps to **physical address [(2*4) + 1]=9.**



- ✓ In paging scheme, we have **no external fragmentation**. Any free frame can be allocated to a process that needs it. But we may have some **internal fragmentation**.
- ✓ If the memory requirements of a process do not happen to coincide with page boundaries, the last frame allocated may not be completely full.
- ✓ **For example**, if page size is 2,048 bytes, a process of 72,766 bytes will need 35 pages plus 1,086 bytes. It will be allocated 36 frames, resulting in **internal fragmentation** of 2,048 - 1,086= **962 bytes**.
- ✓ When a process arrives in the system to be executed, its size expressed in pages is examined. Each page of the process needs one frame. Thus, if the process requires **n** pages, at least **n** frames must be available in memory. If n frames are available, they are allocated to this arriving process.
- ✓ The first page of the process is loaded in to one of the allocated frames, and the frame number is put in the page table for this process. The next page is loaded into another frame and its frame

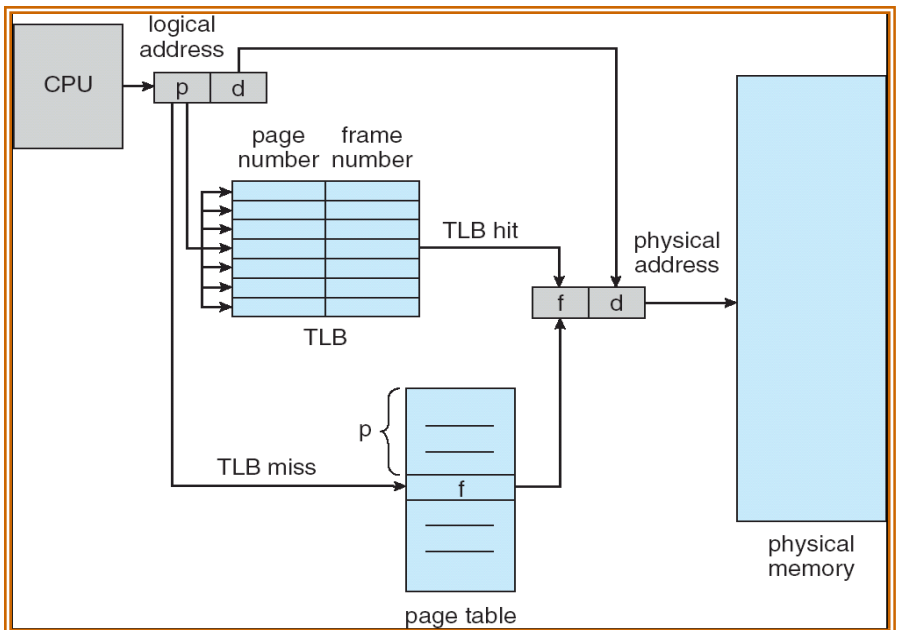
number is put into the page table and so on, as shown in below **figure (a) before allocation, (b) after allocation**.



- **Hardware Support**

  - ✓ The **hardware implementation** of the page table can be done in several ways. The simplest method is that the page table is implemented as a set of **dedicated registers.**
  - ✓ The use of registers for the page table is satisfactory if the page table is reasonably small (for example, 256 entries). But most computers, allow the page table to be very large (for example, 1 million entries) and for these machines, the use of fast registers to implement the page table is not feasible.
  - ✓ So the page table is kept in the main memory and a **page table base register (PTBR)** points to the page table and **page table length register (PTLR)** indicates size of page table. Here two memory accesses are needed to access a byte and thus memory access is slowed by a factor of 2.
  - ✓ The only solution is to use a special, fast lookup hardware **cache** called **Translation look aside buffer (TLB)**. TLB is associative, with high speed memory. Each entry in TLB contains **two** parts, **a key and a value.** When an associative register is presented with an item, it is compared with all the key values, if found the corresponding value field is returned. Searching is fast but hardware is expensive.

  - ✓ TLB is used with the page table **as follows:**
    - o TLB contains only few page table entries.
    - o When a logical address is generated by the CPU, its page number is presented to TLB. If the page number is found its frame number is immediately available and is used to access the actual memory. If the page number is not in the TLB **(TLB miss)** the memory reference to the page table must be made.
    - o When the frame number is obtained we can use it to access the memory as shown in below **figure**. The page number and frame number are added to the TLB, so that they will be found quickly on the next reference.
    - o If the TLB is full of entries the OS must select anyone for **replacement.**
    - o Some TLBs allow entries to be **wired down** meaning that they cannot be removed from the TLB.



- ✓ Some TLBs store **Address Space Identifiers (ASIDs)** in each TLB entry. An ASID uniquely identifies each process and is used to provide address-space protection for that process.
- ✓ The percentage of time that a page number is found in the TLB is called **hit ratio**.
- ✓ **For example,** an **80-percent hit ratio** means that we find the desired page number in the TLB 80 percent of the time. If it takes **20 nanoseconds** to search the TLB and **100 nanoseconds to access memory**, then a mapped-memory access takes **120 nanoseconds when the page number is in the TLB**. If we fail to find the page number in the TLB (20 nanoseconds), then we must first access memory for the page table and frame number (100 nanoseconds) and then access the desired byte in memory (100 nanoseconds), for a total of **220 nanoseconds**. Thus the effective access time is,

  Effective Access Time (EAT) = 0.80 x 120 + 0.20 x 220
  = **140 nanoseconds.**

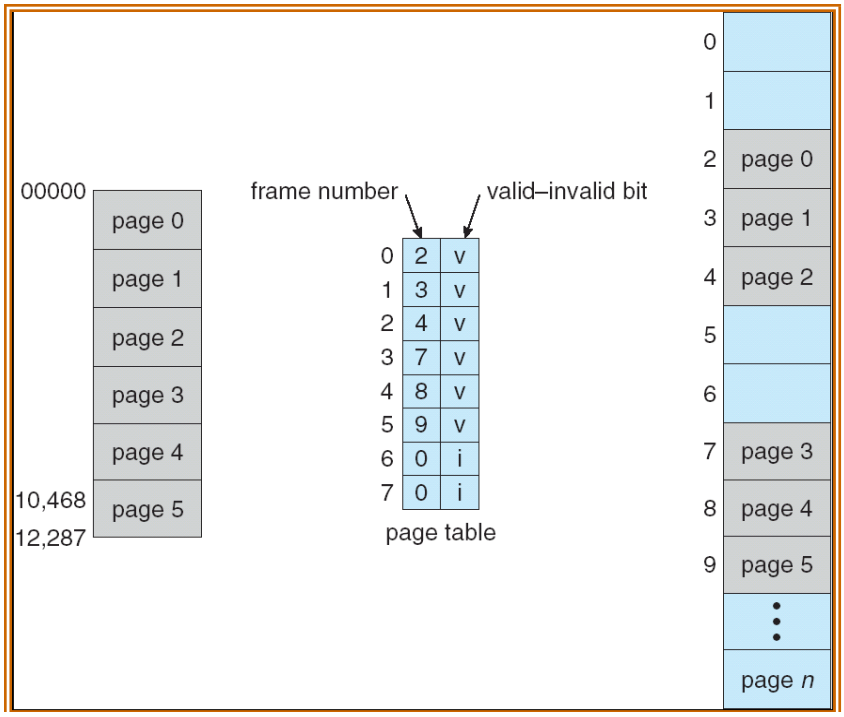
  In this example, we suffer a **40-percent slowdown** in memory-access time (from 100 to 140 nanoseconds).
- ✓ For a **98-percent hit ratio** we have

  Effective Access Time (EAT) = 0.98 x 120 + 0.02 x 220
  = **122 nanoseconds.**
- ✓ This increased hit rate produces only a **22 percent slowdown** in access time.

- **Protection**

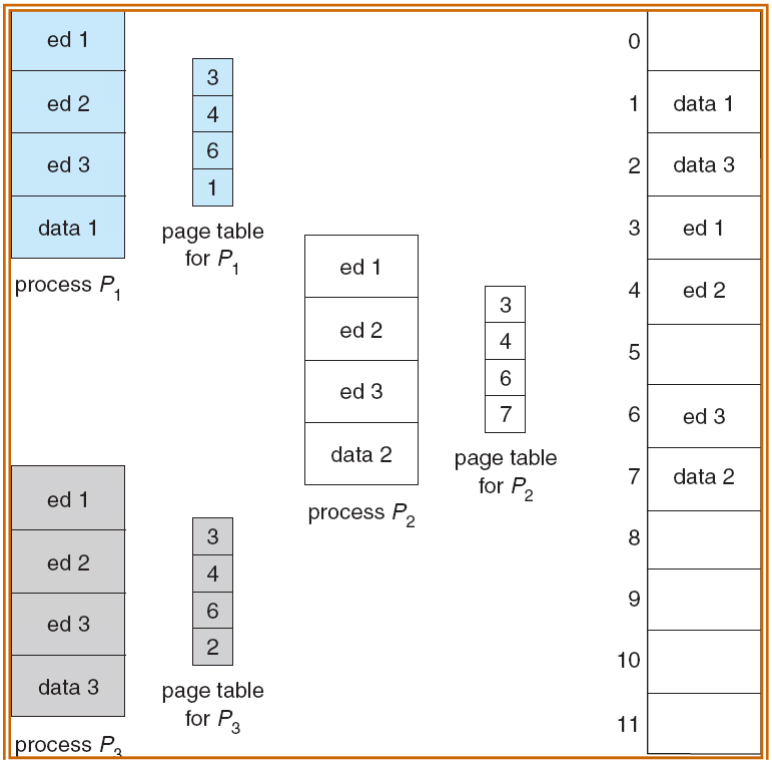
  - ✓ Memory protection in paged environment is done by **protection bits** that are associated with each frame. These bits are kept in page table.
  - ✓ One bit can define a page to be read-write or read-only.
  - ✓ One more bit is attached to each entry in the page table, a **valid-invalid** bit.
  - ✓ A valid bit indicates that associated page is in the process's logical address space and thus it is a legal or valid page.
  - ✓ If the bit is invalid, it indicates the page is not in the process's logical address space and is illegal. Illegal addresses are trapped by using the valid-invalid bit.
  - ✓ The OS sets this bit for each page to allow or disallow accesses to that page.

- ✓ **For example,** in a system with a 14-bit address space (0 to 16383), we have a program that should use only addresses 0 to 10468. Given a page size of 2 KB, we have the situation shown in below **figure**. Addresses in pages 0, 1, 2, 3, 4, and 5 are mapped normally through the page table. Any attempt to generate an address in pages 6 or 7, we find that the valid -invalid bit is set to **invalid**, and the computer will trap to the operating system (**invalid page reference**).



- **Shared Pages**

  - ✓ An advantage of paging is the possibility of **sharing common code**. This consideration is particularly important in a time-sharing environment.
  - ✓ Consider a system that supports 40 users, each of whom executes a text editor. If the text editor consists of 150 KB of code and 50 KB of data space, we need 8,000 KB to support 40 users. If the code is **reentrant code (or pure code)** it can be shared as shown in below **figure**. Here there are three-page editor-each page 50 KB in size and are being shared among three processes. Each process has its own data page.
  - ✓ Reentrant code is **non-self-modifying** code **(Read only).** It never changes during execution. Thus, two or more processes can execute the same code at the same time.
  - ✓ Each process has its own copy of registers and data storage to hold the data for the process's execution.
  - ✓ Thus, to support 40 users, we need only one copy of the editor (150 KB), plus 40 copies of the 50 KB of data space per user. The total space required is now 2150 KB instead of 8,000 KB. **(i.e., 150+40*50= 2150 KB).**

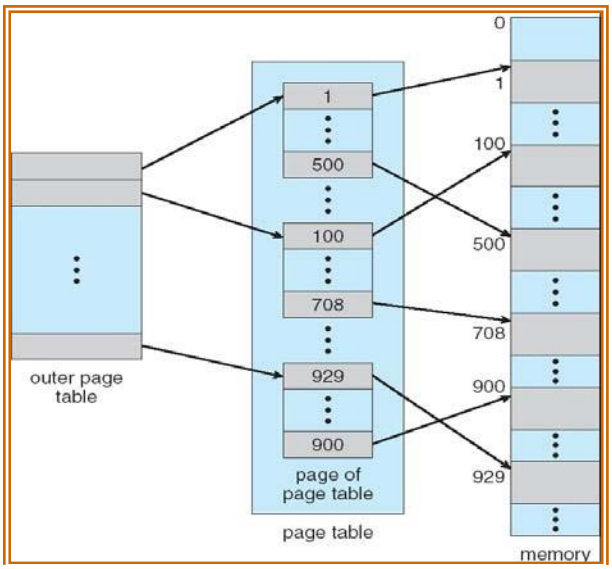


➢ Structure of the Page Table

- **Hierarchical paging**

  ✓ Recent computer system support a large logical address apace from **2^{32} to 2^{64}** and thus page table becomes large. So it is very difficult to allocate contiguous main memory for page table. One **simple solution** to this problem is to divide page table in to smaller pieces.

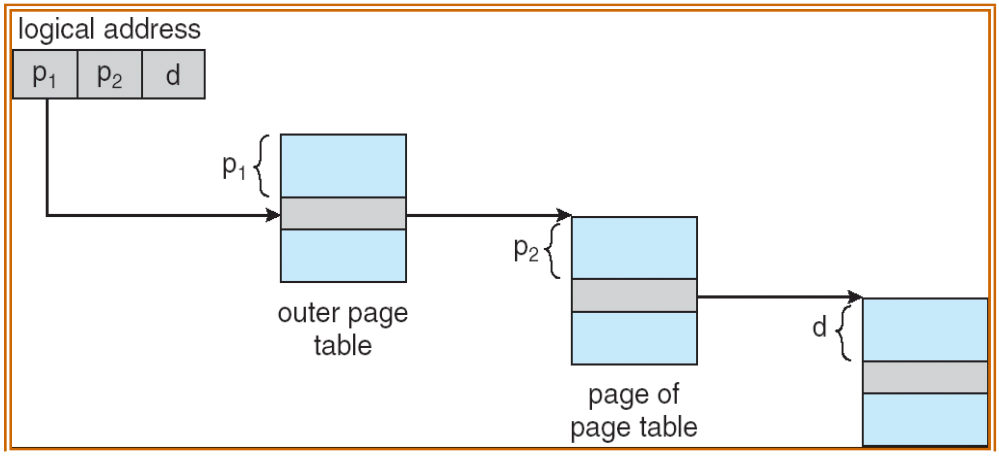
  ✓ One way is to use **two-level paging algorithm** in which the page table itself is also paged as shown in below **figure.**



- ✓ **Ex.** In a 32- bit machine with page size of 4kb, a logical address is divided in to a page number consisting of 20 bits and a page offset of 12 bit(page size 4kb=$2^{12)}$. The page table is further divided since the page table is paged, the page number is further divided in to 10 bit page number and a 10 bit offset. So the logical address is,

Page number		**page offset**
P_1	**P_2**	**d**
10	10	12

- ✓ **P_1** is an index into the outer page table and **P_2** is the displacement within the page of the outer page table. The **address-translation method** for this architecture is shown in below **figure 8.15**. Because address translation works from the outer page table inward, this scheme is also known as a **forward-mapped page table.**



- ✓ For a system with a 64-bit logical address space, a two-level paging scheme is no longer appropriate. Suppose the page size in such a system is **4 KB** the page table consists of up to **2^{52}** entries. If we use a two-level paging scheme, then the inner page tables can be one page long, or contain **2^{10}** 4-byte entries. The addresses look like this,

Outer page	**inner page**	**offset**
P_1	**P_2**	**d**
42	10	12

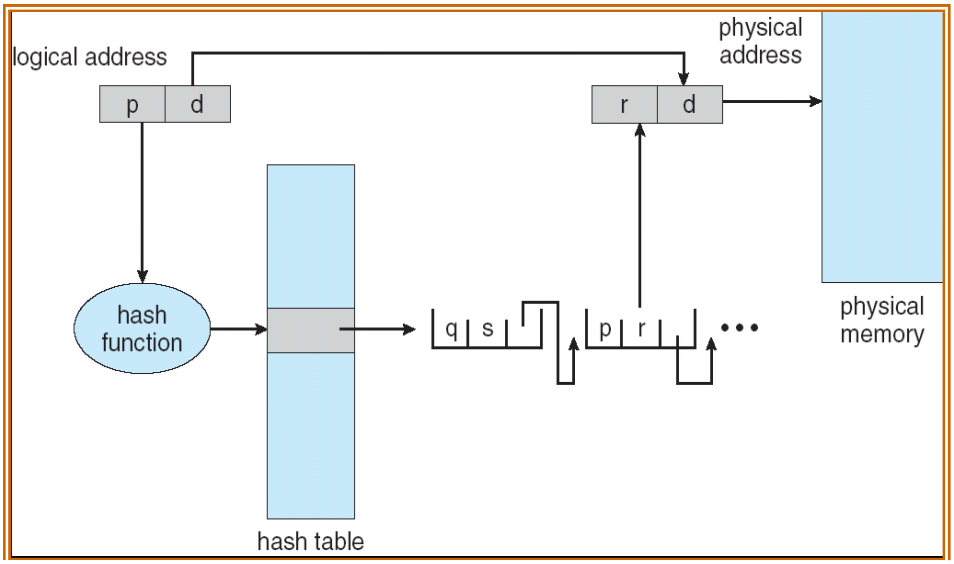
- ✓ The outer page table consists of 2^{42} entries, or 2^{44} bytes. The one way to avoid such a large table is to divide the outer page table into smaller pieces.
- ✓ We can avoid such a large table using **three-level paging scheme**.

2nd outer page	**outer page**	**inner page**	**offset**
P_1	**P_2**	**P_3**	**d**
32	10	10	12

- ✓ The outer page table is still 2^{34} bytes in size. The next step would be a **four-level paging scheme.**

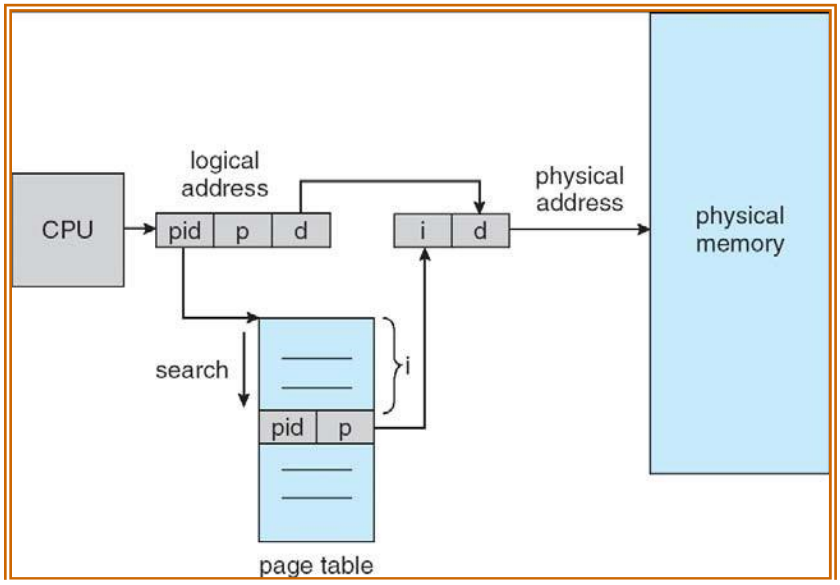
- **Hashed page table**

  - ✓ Hashed page table handles the address space larger than 32 bit. The virtual page number is used as **hash value**. Linked list is used in the hash table which contains a list of elements that hash to the same location.
  - ✓ Each element in the hash table contains the following three fields,
    - **Virtual page number**
    - **Mapped page frame value**
    - **Pointer to the next element in the linked list**
  - ✓ The algorithm works as follows,
    - Virtual page number is taken from virtual address and is hashed in to the hash table.
    - Virtual page number is compared with **field 1** in the first element in the linked list.
    - If there is a match, the corresponding page frame **(field 2)** is used to form the desired physical address. If there is no match, subsequent entries in the linked list are searched for a matching virtual page number. This scheme is shown in below **figure.**
    - **Clustered pages** are similar to hash table but one difference is that each entity in the hash table refer to several pages.



- **Inverted Page Tables**

  - ✓ Page tables may consume large amount of physical memory just to keep track of how other physical memory is being used.
  - ✓ To solve this problem, we can use an inverted page table has one entry for each real page (or frame) of memory. Each entry consists of the virtual address of the page stored in that real memory location with information about the process that owns the page.
  - ✓ Thus, only one page table is in the system, and it has only one entry for each page of physical memory.
  - ✓ The below **figure** shows the operation of an inverted page table.

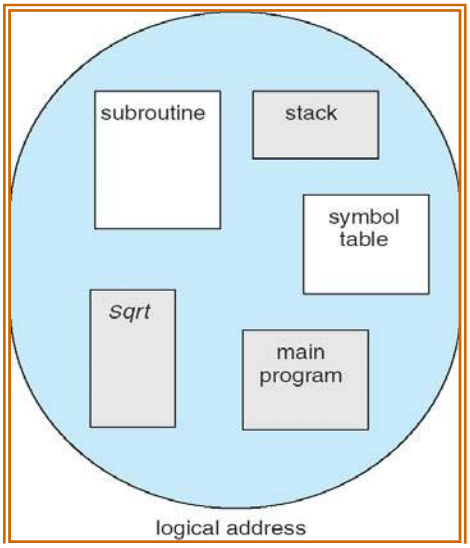


- ✓ The inverted page table entry is a pair **<process-id, page number>**. **Where process-id assumes the role of the address-space identifier**. When a memory reference is made, the part of virtual address consisting of **<process-id, page number>** is presented to memory sub-system.
- ✓ The inverted page table is searched for a match. If a match is found at **entry i,** then the physical address **<i, offset>** is generated. If no match is found then an illegal address access has been attempted.
- ✓ This scheme **decreases the amount of memory** needed to store each page table, but increases the amount of time needed to search the table when a page reference occurs.

➢ Segmentation

- **Basic method**

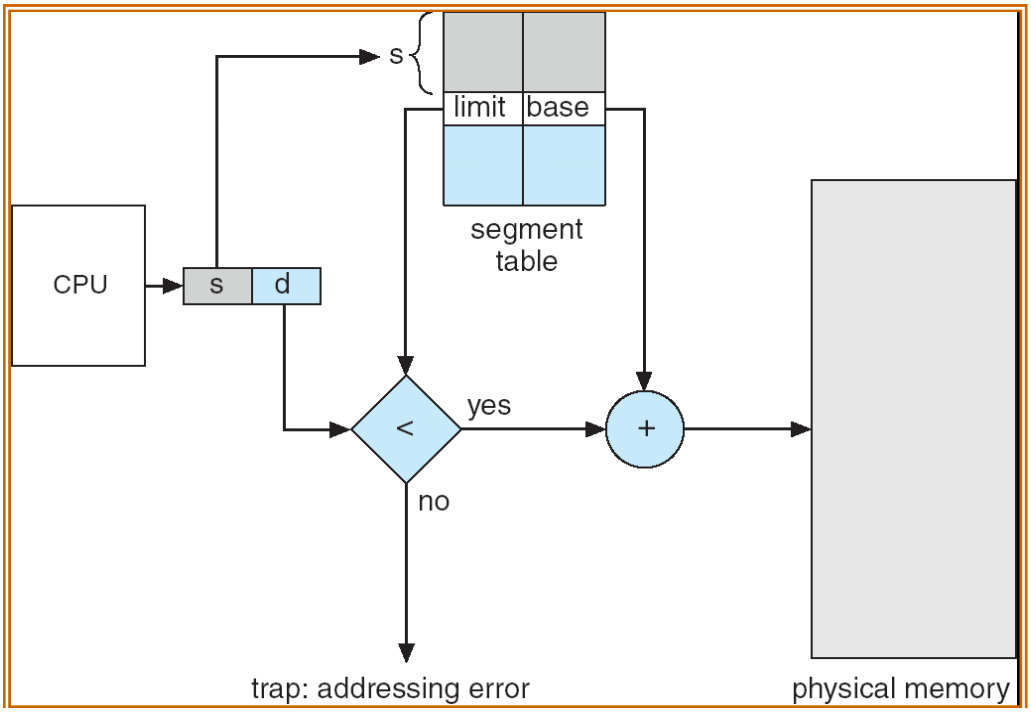
✓ Users prefer to view memory as a collection of **variable-sized segments**, **with no ordering among segments** as shown in below **figure**.



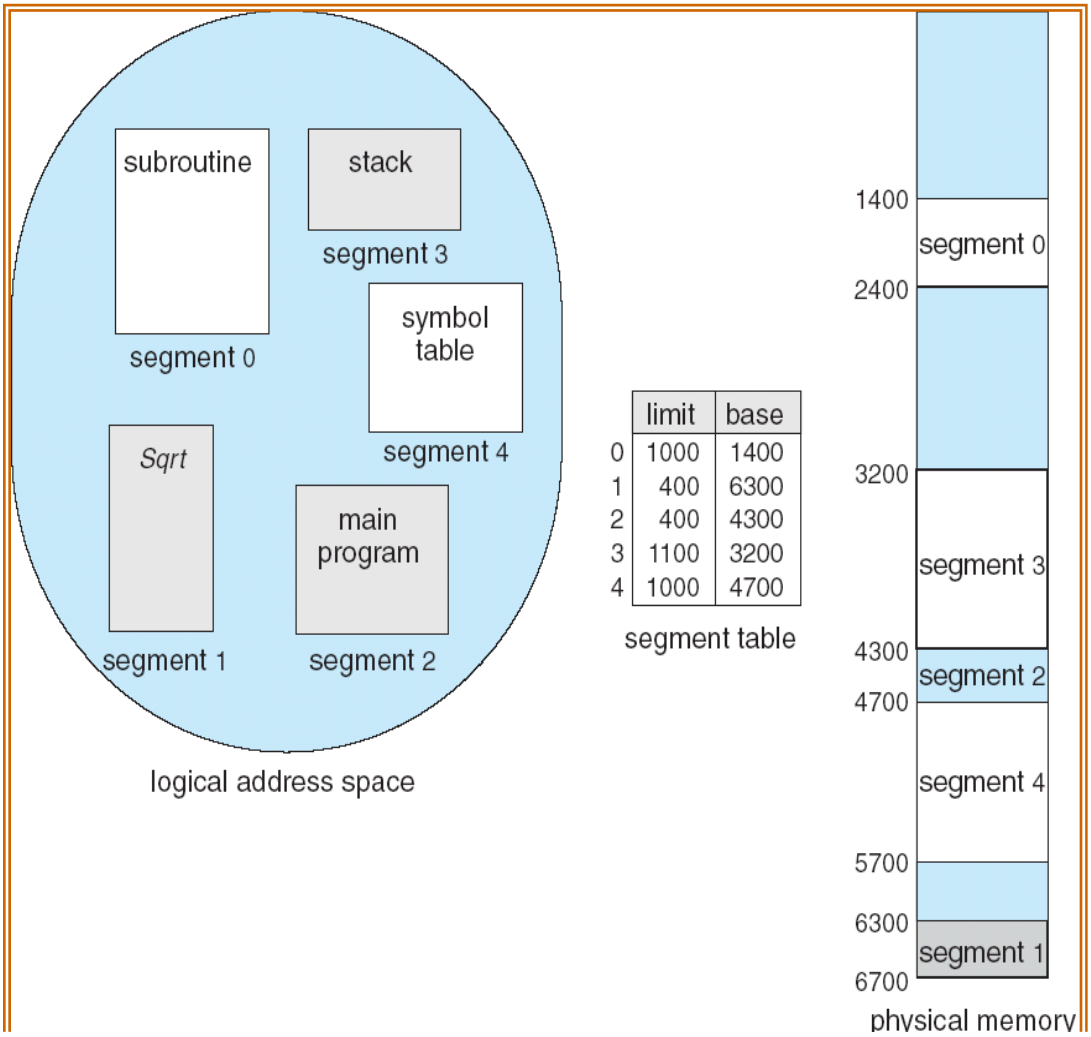
- ✓ **Segmentation** is a memory-management scheme that supports the user view of memory.
- ✓ A **logical address** is a collection of segments. Each segment has a **name and length.** The address specifies both the **segment name and the offset** within the segments.
- ✓ The segments are numbered and are referred by a segment number. So the logical address consists of <**segment number, offset**>.

- **Hardware**

- ✓ **Segment table** maps **2-Dimensional user defined address** in to **1-Dimensional physical address.**
- ✓ Each entry in the segment table has a **segment base and segment limit**.
- ✓ The segment **base contains the starting physical address where the segment resides and limit specifies the length of the segment**.
- ✓ The use of segment table is shown in the below **figure.**



- ✓ Logical address consists of two parts, **segment number s and an offset d.**
- ✓ The segment number is used as an i**ndex** to segment table. The offset must be in between **0 and limit**, if not an error is reported to OS.
- ✓ If legal the offset is added to the base to generate the actual physical address.
- ✓ The segment table is an **array of base-limit register pairs**.
- ✓ **For example**, consider the below **figure**. We have **five segments** numbered from 0 through 4. Segment 2 is 400 bytes long and begins at location 4300. Thus, a reference to byte 53 of segment 2 is mapped onto location 4300 +53= 4353. A reference byte 852 of segment 3, is mapped to 3200 (the base of segment 3) + 852 = 4052. A reference to byte 1222 of segment 0 would result in a trap to the operating system, as this segment is only 1,000 bytes long.


subroutine
stack
segment 3
segment 0
symbol table
segment 4
Sqrt
main program
segment 1
segment 2
logical address space
limit base
0 1000 1400
1 400 6300
2 400 4300
3 1100 3200
4 1000 4700
segment table
1400
segment 0
2400
3200
segment 3
4300
segment 2
4700
segment 4
5700
6300
segment 1
6700
physical memory

QUESTION BANK

Process Synchronization

1. What are semaphores? Explain two primitive semaphore operations. What are its advantages?
2. Explain any one synchronization problem for testing newly proposed sync scheme.
3. Explain three requirements that a solution to critical –section problem must satisfy.
4. State dining philosopher's problem and give a solution using semaphores. Write structure of philosopher i.
5. What do you mean by binary semaphore and counting semaphore? With C struct, explain implementation of wait() and signal().
6. Describe term monitor. Explain solution to dining philosopher's problem using monitor.
7. What are semaphores? Explain its usage and implementation.
8. What are monitors? Solve Dinning philosophers problem by using monitors
9. Define spin lock.
10. Explain synchronization problem of bounded buffer and reader writers and give a solution code by using semaphores.
11. Explain Peterson's solution to solve critical section problem.
12. Give the need of priority inheritance protocol.
13. Explain synchronization hardware and mutex lock implementation with sample code.
14. What are monitors? Explain.
15. Give a solution for producer and consumer problem using monitors

File Implemetation

1. What is a file? Describe different access methods on files.
2. What is file mounting? Explain.
3. Draw neat diagram and explain fixed file allocation. Is FAT linked allocation?
4. Explain following: file types, file operations, file attributes.
5. Explain methods to implement directories .
6. What is free space list? With example, explain any two methods to implement free space list.
7. What are major methods to allocate disk space? Explain each with examples.
8. Explain different file access methods.
9. Explain various directory structures.
10. Explain different disk space allocation methods with example.
11. Explain the various storage mechanisms available to store files, with neat diagram.

Deadlocks:

1. Explain with an example how resource allocation graph is used to define deadlock?
2. What are two options for breaking deadlock?
3. What is wait-for graph? How is it useful for detection of deadlock?

4.

	Allocation	Request	Available
1.	ABC	ABC	ABC
ii. P_0	0 1 0	0 0 0	0 0 0
iii. P_1	2 0 0	2 0 2	
iv. P_2	3 0 3	0 0 0	
v. P_3	2 1 1	1 0 0	
vi. P_4	0 0 2	0 0 2	

Show the system is not deadlocked by one safe sequence. At t_2, p_2 makes one additional request for type C, show that system is deadlocked if request is granted.

5. Define hardware instructions test() , set() and swap(). Give algorithms to implement mutual exclusion with these instructions.
6. Describe necessary conditions for a deadlock situation to arise and how to handle.
7. List any 4 examples of deadlock that are not related to computer systems.
8. Explain the safety algorithm used in banker's algorithm, with suitable data structures.
9. List any 4 examples of deadlock that are not related to computer systems.
10. Explain the safety algorithm used in banker's algorithm, with suitable data structures.
11. Explain banker's algorithm for deadlock avoidance.
12. Explain process termination and resource preemption to recover from deadlock.
13. Consider given chart and answer i) what is content of matrix need? ii) is system safe?If yes give safe sequence. iii) if request comes from P_1, arrives for (0,4,2,0), can it be granted ?

	Allocation	**Max**	**Available**
	ABCD	**ABCD**	**ABCD**
a. P_0	0 0 1 2	0 0 1 2	1 5 2 0
b. P_1	1 0 0 0	1 7 5 0	
c. P_2	1 3 5 4	2 3 5 6	
d. P_3	0 6 3 2	0 6 5 2	
e. P_4	0 0 1 4	0 6 5 6	

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

i. Need

ii. A B C D

b. P_0 0 0 0 0
c. P_1 0 7 5 0
d. P_2 1 0 0 2
e. P_3 0 0 2 0
f. P_4 0 6 4 2

ii. P_0 0 0 0 0 <= 1 5 2 0 is true, so work=work + allocation
a. work= 1 5 2 0 + 0 0 1 2=1 5 3 2
iii. P_1 0 7 5 0 <= 1 5 3 2 is false,
iv. P_2 1 0 0 2 <= 1 5 3 2 is true, so work=1 5 3 2 + 1 3 5 4=2 8 8 6
v. P_3 0 0 2 0 <= 2 8 8 6 is true, so work=2 8 8 6 + 0 6 3 2=2 14 11 8
vi. P_4 0 6 4 2 <= 2 14 11 8 is true, so work=2 14 11 8+ 0 0 1 4=2 14 12 12
vii. P_1 0 7 5 0 <= 2 14 12 12 is true, so work=2 14 12 12+ 1 0 0 0=3 14 12 12

We claim that the system is currently in a safe state. The sequnce < P_o, P_2, P_3, P_4, P_l > satisfies the safety criteria. Suppose now the process P_1 requests for **(0,4,2,0)**. To decide whether this request can be immediately granted, we first check that

f. **Request$_1$ <= Need$_1$**, that is, **(0,4,2,0) <= (0,7,5,0)** which is true then,
g. **Request$_1$ <= Available$_1$**, that is, **(0,4,2,0)<= (1,5,2,0)**, which is true. Then we arrive at the following new state:

1. **Allocation Max Available**
2. **ABCD ABCD ABCD**

ii. P_0 0 0 1 2 0 0 1 2 1 1 0 0
iii. P_1 1 4 2 0 1 7 5 0
iv. P_2 1 3 5 4 2 3 5 6
v. P_3 0 6 3 2 0 6 5 2
vi. P_4 0 0 1 4 0 6 5 6

i. Need
ii. A B C D

b. P_0 0 0 0 0
c. P_1 0 3 3 0
d. P_2 1 0 0 2
e. P_3 0 0 2 0
f. P_4 0 6 4 2

Now we must determine whether this new system state is safe. We execute safety algorithm and find that the sequence <P_1, P_3, P_4, P_o, P_2> satisfies the safety requirement. Hence the request can be immediately granted.

14. Consider a system consisting of *m* resources of the same type being shared by *n* processes. A process can request or release only one resource at a time. Show that the system is deadlock free if the following two conditions hold:
    a. The maximum need of each process is between one resource and *m* resources.
    b. The sum of all maximum needs is less than *m* + *n*.

Solution:

The given conditions can be written as

$Max_i >= 1$ for all i

$$\sum_{i=1}^{n} Max_i < m+n$$

The need value can be calculated as, $Need_i = Max_i - Allocation_i$

If there exists a deadlock then,

$$\sum_{i=1}^{n} Allocation_i = m$$

Therefore, $\sum Need_i + \sum Allocation_i = \sum Max_i < m+n$

We get, $\sum Need_i + m < m+n$

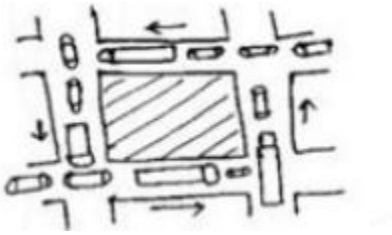
Hence, $\sum Need_i < n$

This implies there exist P_i such that $Need_i=0$. Since $Max_i >= 1$, P_i has atleast one resource to release. Hence no deadlocks.

For ex, consider m=5, n=3 and below snapshot then we can trace out with above steps and prove that no deadlock occurs

	Max	**Allocation**	**Available**	**Need**
P_0	2	1	0	1
P_1	3	2		1
P_2	2	2		0

15. Define deadlock. Consider the traffic deadlock depicted in the figure given below, Analyze the four necessary conditions for deadlock indeed hold in the example below.



16. What is wait for graph? Explain Deadlock detection algorithm for several instances of resource type.

Memory Management:

1. What do you mean by fragmentation? Explain difference between internal and external fragmentation.
2. Differentiate between internal and external fragmentation. How are they overcome?
3. What is paging and swapping?
4. What is address binding? Explain with necessary steps, binding instructions and data to memory addresses.
5. Mention the problem with simple paging scheme. How TLB is used to solve this problem? Explain with supporting h/w dig with example.
6. Draw and explain the multistep processing of a user program.
7. In the paging scheme with TLB it takes 20 ns to search the TLB and 100 ns to access memory. Find the effective access time and percentage slowdown in memory access time if
   i) Hit Ratio is 80%
   ii) Hit Ratio is 98%\
8. What are drawbacks of contiguous memory allocation? Given five memory partitions of 100KB, 500 KB, 200 KB,300 KB and 600 KB . How would each of first fir, best fir and worst fir algorithms work to place processes of 212KB,417 KB,112KB and 426KB(in order)? Which is more efficient?

Solution:

First Fit

100K	
	212K
500K	112K
200K	
300K	

Best Fit

100K	
500K	417K
200K	112K
300K	212K
600K	426K

Worst Fit

100K	
500K	417K
200K	
300K	212K
600K	112K

600K | 417K |

426 must wait

426 must wait

The Best Fit is the efficient algorithm.

9. What is locality of reference? Differentiate between paging and segmentation.
10. Explain the differences between. i) Logical and Physical address space.
    ii) Internal and External fragmentation.
11. What is paging? Give advantages and disadvantages.
12. What is segmentation? Give advantages & disadvantages.
13. What are the different methods of implementing page table?
14. How can you ensure protection & sharing in paging?
15. How to satisfy a request of size n from a list of free holes? Explain.
16. Explain the following:
    a) Privileged instruction
    b) Transient code
    c) 50-percent rule
    d) Roll in, roll out
    e) Compaction
17. How can hardware address protection be ensured with base & limit registers?
18. What is dynamic loading? Give advantages.
19. Explain the following:
    a) Frame table
    b) Hit ratio
    c) Re-entrant code
    d) Legal page
    e) TLB
20. Give a brief idea on dynamic linking & shared libraries.