MODULE-III

Deadlocks

A set of process is in a deadlock state if each process in the set is waiting for an event that can be caused by only another process in the set. In other words, each member of the set of deadlock processes is waiting for a resource that can be released only by a deadlock process. None of the processes can run, none of them can release any resources, and none of them can be awakened.

The resources may be either physical or logical. Examples of physical resources are Printers, Hard Disc Drives, Memory Space, and CPU Cycles. Examples of logical resources are Files, Semaphores, and Monitors.

The simplest example of deadlock is where process 1 has been allocated non-shareable resources A (say a Hard Disc drive) and process 2 has be allocated non-sharable resource B (say a printer). Now, if it turns out that process 1 needs resource B (printer) to proceed and process 2 needs resource A (Hard Disc drive) to proceed and these are the only two processes in the system, each is blocked the other and all useful work in the system stops. This situation is termed deadlock. The system is in deadlock state because each process holds a resource being requested by the other process neither process is willing to release the resource it holds.

Preemptable and Nonpreemptable Resources

Resources come in two flavors: preemptable and nonpreemptable.

- ➲ A preemptable resource is one that can be taken away from the process with no ill effects. Memory is an example of a preemptable resource.
- ➲ A nonpreemptable resource is one that cannot be taken away from process (without causing ill effect). For example, CD resources are not preemptable at an arbitrary moment.
- ➲ Reallocating resources can resolve deadlocks that involve preemptable resources. Deadlocks that involve nonpreemptable resources are difficult to deal with.

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

1. **Request:** If the request cannot be granted immediately, then the requesting process must wait until it can acquire the resource.
2. **Use:** The process can operate on the resource.
3. **Release:** The process releases the resource.

Necessary Conditions for Deadlock

Coffman (1971) identified four conditions that must hold simultaneously for there to be a deadlock.

1. **Mutual Exclusion Condition**: The resources involved are non-shareable.
   Explanation: At least one resource must be held in a non-shareable mode, that is, only one process at a time claims exclusive control of the resource. If another process requests that resource, the requesting process must be delayed until the resource has been released.
2. **Hold and Wait Condition:** Requesting process hold already the resources while waiting for requested resources.
   Explanation: There must exist a process that is holding a resource already allocated to it while waiting for additional resource that are currently being held by other processes.

4. **No-Preemptive Condition:** Resources already allocated to a process cannot be preempted.
   Explanation: Resources cannot be removed from the processes are used to completion or released voluntarily by the process holding it.

4. **Circular Wait Condition**: The processes in the system form a circular list or chain where each process in the list is waiting for a resource held by the next process in the list. There exists a set {P0, P1, …, P0} of waiting processes such that P0 is waiting for a resource that is held by P1, P1 is waiting for a resource that is held by P2, …, Pn–1 is waiting for a resource that is held by Pn, and P0 is waiting for a resource that is held by P0.

Note: It is not possible to have a deadlock involving only one single process. The deadlock involves a circular “hold-and-wait” condition between two or more processes, so “one” process cannot hold a resource, yet be waiting for another resource that it is holding. In addition, deadlock is not possible between two threads in a process, because it is the process that holds resources, not the thread that is, each thread has access to the resources held by the process.

Resource-Allocation Graph

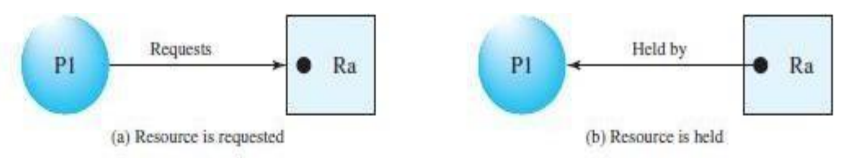
Deadlocks can be described in terms of a directed graph called a system resource allocation graph

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 = {R1, 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 Pi →Rj is called a request edge.

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 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. A request edge points to only the square Rj, whereas an assignment edge must designate one of the dots in the square.



The resource-allocation graph shown below depicts the following situation.

The sets P, R, and E:

P={P1,P2,P3}

R={R1,R2,R3,R4}

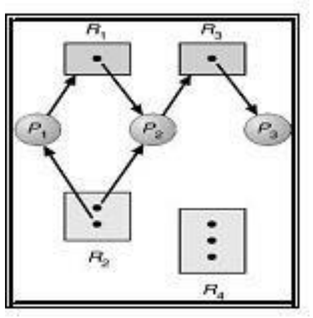
E={P1→R1, P2→R3, R1→P2, R2→P2, R2→P1, R3→P3}

Resource instances:

- One instance of resource type R1
- Two instances of resource type R2
- One instance of resource type R3
- Three instances of resource type R4

Process states:

- Process PI 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 R1 and R2, and is waiting for an instance of resource type R3.
- Process P3 is holding an instance of R3.



Example of a Resource Allocation Graph

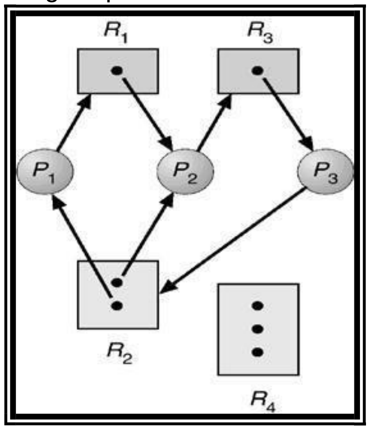
Given the definition of a resource-allocation graph, it can be shown that, if the graph contains no cycles, then no process in the system is deadlocked. If the graph does contain a cycle, then a deadlock may exist.

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.

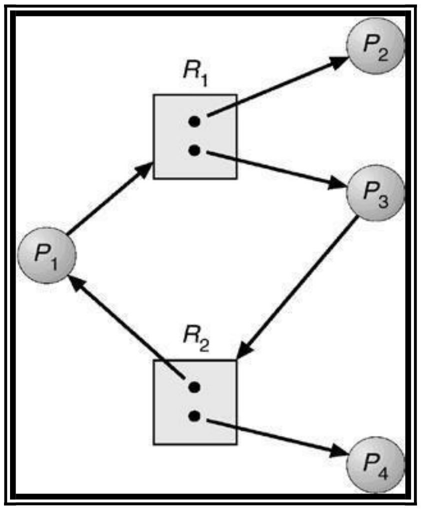
To illustrate this concept, let us return to the resource-allocation graph depicted in Figure. Suppose that process P3 requests an instance of resource type R2. Since no resource instance is currently available, a request edge P3→ R2 is added to the graph. At this point, two minimal cycles exist in the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

P1→R1→ P2→ R3→ P3→ R2→
P1 P2→ R3 → P3→ R2→ P2
Processes PI, 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 PI or process P2 to release resource R2. In addition, process PI is waiting for process P2 to release resource R1.



Resource Allocation Graph with a deadlock

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



Resource Allocation Graph with a Cycle but No Deadlock

METHODS OF HANDLING DEADLOCK

In general, there are four strategies of dealing with deadlock problem:

1. **Deadlock Prevention**: Prevent deadlock by resource scheduling so as to negate at least one of the four conditions.
2. **Deadlock Avoidance**: Avoid deadlock by careful resource scheduling.
3. **Deadlock Detection and Recovery**: Detect deadlock and when it occurs, take steps to recover.
4. **The Ostrich Approach:** Just ignore the deadlock problem altogether.

DEADLOCK PREVENTION

A deadlock may be prevented by denying any one of the conditions.

- **Elimination of "Mutual Exclusion" Condition:** The mutual exclusion condition must hold for non-sharable resources. That is, several processes cannot simultaneously share a single resource. This condition is difficult to eliminate because some resources, such as the Hard disc drive and printer, are inherently non-shareable. Note that shareable resources like read-only-file do not require mutually exclusive access and thus cannot be involved in deadlock.
- **Elimination of "Hold and Wait" Condition:** There are two possibilities for elimination of the second condition. The first alternative is that a process request be granted all of the resources it needs at once, prior to execution. The second alternative is to disallow a process from requesting resources whenever it has previously allocated resources. This strategy requires that all of the resources a process will need must be requested at once. The system must grant resources on "all or none" basis. If the complete set of resources needed by a process is not currently available, then the process must wait until the complete set is available. While the process waits, however, it may not hold any resources. Thus the "wait for" condition is denied and deadlocks cannot occur. This strategy can lead to serious waste of resources.
- **Elimination of "No-preemption" Condition:** The non-preemption condition can be alleviated by forcing a process waiting for a resource that cannot immediately be allocated to relinquish all of its currently held resources, so that other processes may use them to finish. This strategy requires that when a process that is holding some resources is denied a request for additional resources. The process must release its held resources and, if necessary, request them again together with additional resources. Implementation of this strategy denies the "no-preemptive" condition effectively.
- **Elimination of "Circular Wait" Condition:** The last condition, the circular wait, can be denied by imposing a total ordering on all of the resource types and then forcing, all processes to request the

resources in order (increasing or decreasing). This strategy impose a total ordering of all resources types, and to require that each process requests resources in a numerical order (increasing or decreasing) of enumeration. With this rule, the resource allocation graph can never have a cycle.

For example, provide a global numbering of all the resources, as shown

1	≡	Card reader
2	≡	Printer
3	≡	Optical driver
4	≡	HDD
5	≡	Card punch

Now the rule is this: processes can request resources whenever they want to, but all requests must be made in numerical order. A process may request first printer and then a HDD(order: 2, 4), but it may not request first a optical driver and then a printer (order: 3, 2). The problem with this strategy is that it may be impossible to find an ordering that satisfies everyone.

DEADLOCK AVOIDANCE

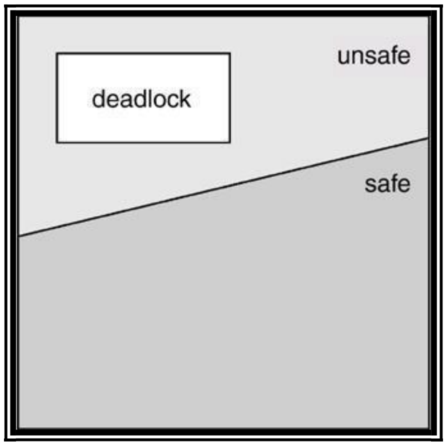
This approach to the deadlock problem anticipates deadlock before it actually occurs. This approach employs an algorithm to access the possibility that deadlock could occur and acting accordingly. If the necessary conditions for a deadlock are in place, it is still possible to avoid deadlock by being careful when resources are allocated. It employs the most famous deadlock avoidance algorithm that is the Banker's algorithm.

A deadlock-avoidance algorithm dynamically examines the resource-allocation state to ensure that a circular wait condition can never exist. The resource-allocation *state* is defined by the number of available and allocated resources, and the maximum demands of the processes.

Safe and Unsafe States

A system is said to be in a **Safe State**, if there is a safe execution sequence. An execution sequence is an ordering for process execution such that each process runs until it terminates or blocked and all request for resources are immediately granted if the resource is available.

A system is said to be in an **Unsafe State**, if there is no safe execution sequence. An unsafe state may not be deadlocked, but there is at least one sequence of requests from processes that would make the system deadlocked.



(Relation between Safe, Unsafe and Deadlocked States)

Resource-Allocation Graph Algorithm

The deadlock avoidance algorithm uses a variant of the resource-allocation graph to avoid deadlocked state. It introduces a new type of edge, called a **claim edge.** A claim edge Pi →Rj indicates that process Pi may request resource Rj at some time in the future. This edge resembles a request edge in direction, but is represented by a dashed line. When process Pi requests resource Rj, the claim edge Pi →Rj is converted to a request edge. Similarly, when a resource Rj is released by Pi**,** the assignment edge Rj →Pi is reconverted to a claim edge Pi→ Rj.

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 that does not result in the formation of a cycle in the resource-allocation graph. An algorithm for detecting a cycle in this graph is called cycle detection algorithm.

If no cycle exists, then the allocation of the resource will leave the system in a safe state. If a cycle is found, then the allocation will put the system in an unsafe state. Therefore, process *Pi* will have to wait for its requests to be satisfied.

Banker's algorithm

The **Banker's algorithm** is a resource allocation & deadlock avoidance algorithm developed by Edsger Dijkstra that test for safety by simulating the allocation of pre-determined maximum possible amounts of all resources. Then it makes a "safe-state" check to test for possible deadlock conditions for all other pending activities, before deciding whether allocation should be allowed to continue.

The Banker's algorithm is run by the operating system whenever a process requests resources. The algorithm prevents deadlock by denying or postponing the request if it determines that accepting the request could put the system in an unsafe state (one where deadlock could occur).

For the Banker's algorithm to work, it needs to know three things:

- How much of each resource could possibly request by each process.
- How much of each resource is currently holding by each process.
- How much of each resource the system currently has available.

Resources may be allocated to a process only if it satisfies the following conditions:

1. request ≤ max, else set error as process has crossed maximum claim made by it.
2. request ≤ available, else process waits until resources are available.

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[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 *Ri* to complete its task. Note that *Need[i,j] = Max[i,j] - Allocafion[i,j].*

These data structures vary over time in both size and value. The vector *Allocation$_i$* specifies the resources currently allocated to process *Pi;* the vector *Need$_i$* specifies the additional resources that process *Pi* may still request to complete its task.

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 *Finisk[i] :=false* for *i* = 1,2, ..., *n.*
2. Find an *i* such that both a. *Finisk[i] =false*
   b. *Need$_i$* ≤ *Work.*
   If no such ***i*** exists, go to step 4.
3. *Work* := *Work* + *Allocation$_i$*
   Finisk[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.

Resource-Request Algorithm

Let *Request$_i$* be the request vector for process *Pi.* If *Request$_i$[j]* = *k,* then process *Pi* 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, *Pi* must wait, since the resources are not available.
3. Have the system pretend to have allocated the requested resources to process *Pi* by modifying the state as follows:
   Available := Available - Request$_i$;
   Allocation$_i$:= Allocation$_i$ + Request$_i$;
   Need$_i$:= Need$_i$ - Request$_i$;

If the resulting resource-allocation state is safe, the transaction is completed and process *Pi* is allocated its resources. However, if the new state is unsafe, then *Pi* must wait for *Request$_i$* and the old resource-allocation state is restored.

Consider a system with five processes P0 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 T0, the following snapshot of the system has been taken:

	Allocation	Max	Available
	A B C	A B C	A B C
P0	0 1 0	7 5 3	3 3 2
PI	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
P0	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 <PI, P3, P4, P2, P0> satisfies the safety criteria. Suppose now that process P1 requests one additional instance of resource type A and two instances of resource type C, so Request1 = (1,0,2). To decide whether this request can be immediately granted, we first check that Request1≤ Available (that is, (1,0,2) ≤ (3,3,2)), which is true. We then pretend that this request has been fulfilled, and we arrive at the following new state:

	Allocation	Need	Available
	A B C	A B C	A B C
P0	0 1 0	7 4 3	2 3 0
PI	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 <PI, P3, P4, P0, P2> satisfies our safety requirement. Hence, we can immediately grant the request of process PI.

However, that when the system is in this state, a request for (3,3,0) by P4 cannot be granted, since the resources are not available. 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

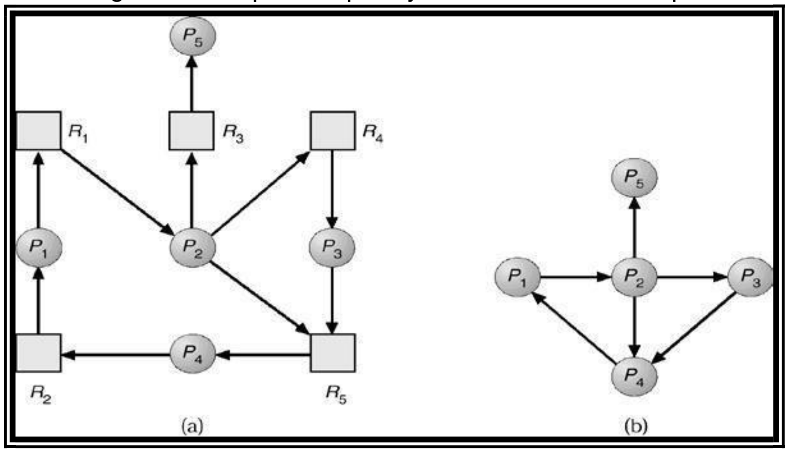
If a system does not employ either a deadlock-prevention or a deadlock avoidance algorithm, then a deadlock situation may occur. In this environment, the system must provide:

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

According to number of instances in each resource type, the Deadlock Detection algorithm can be classified into two categories as follows:

1. **Single Instance of Each Resource Type:** If all resources have only a single instance, then it can define a deadlock detection algorithm that uses a variant of the resource-allocation graph (is called a *wait-for* graph). A wait–for graph can be draw by removing the nodes of type resource and collapsing the appropriate edges from the resource-allocation graph.

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 a wait-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:



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

A deadlock exists in the system if and only if the wait-for graph contains a cycle. To detect deadlocks, the system needs to maintain the wait-for graph and periodically to invoke an algorithm that searches for a cycle in the graph. An algorithm to detect a cycle in a graph requires an order of n^2 operations, where *n* is the number of vertices in the graph.

2. **Several Instances of a Resource Type:** The following deadlock-detection algorithm is applicable to several instance of a resource type. The algorithm employs several time-varying data structures:
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 *Pi* is requesting *k* more instances of resource type Rj.

The detection algorithm is described as follows:

```
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. $Request_i \leq Work$.
   If no such i exists, go to step 4.
3. Work := Work + $Allocation_i$
   Finish[i] := true
   go to step 2.
4. If Finish[i] = false, for some i, $1 \leq i \leq n$, then the system is in a deadlock
   state. 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.

RECOVERY FROM DEADLOCK

When a detection algorithm determines that a deadlock exists, then the system or operator is responsible for handling deadlock problem. There are two options for breaking a deadlock.

1. Process Termination
2. Resource

preemption 3.

Process Termination

There are two method to eliminate deadlocks by terminating a process as follows:

1. **Abort all deadlocked processes:** This method will break the deadlock cycle clearly by terminating all process. This method is cost effective. And it removes the partial computations completed by the processes.
2. **Abort one process at a time until the deadlock cycle is eliminated:** This method terminates one process at a time, and invokes a deadlock-detection algorithm to determine whether any processes are still deadlocked.

Resource Preemption

In resource preemption, the operator or system preempts 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:** The system or operator selects which resources and which processes are to be preempted based on cost factor.
2. **Rollback:** The system or operator must roll back the process to some safe state and restart it from that state.
3. **Starvation:** The system or operator should ensure that resources will not always be preempted from the same process?

MEMORY MANAGEMENT

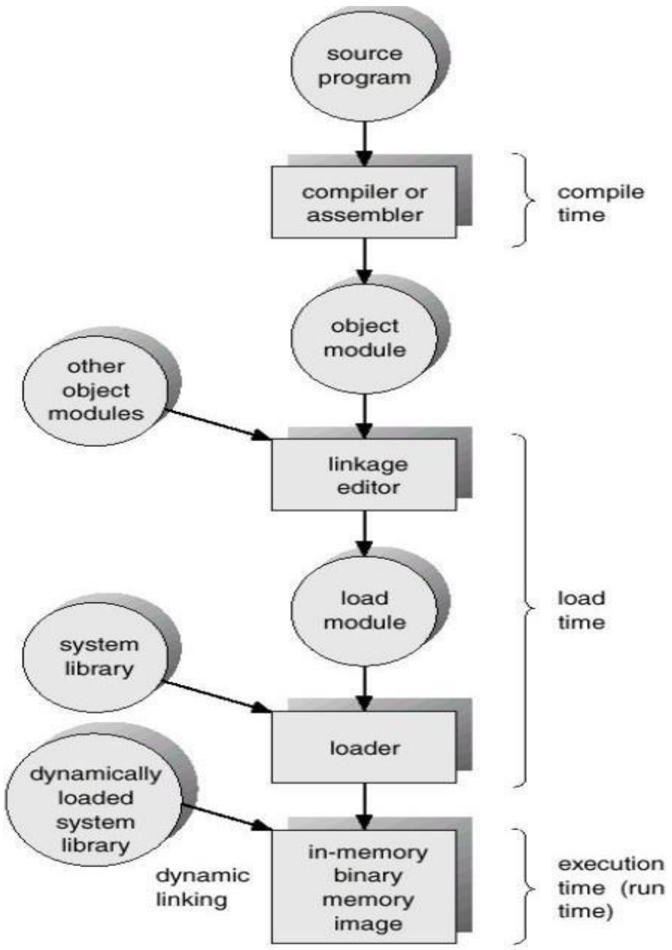
In a uni-programming system, main memory is divided into two parts: one part for the operating system (resident monitor, kernel) and one part for the user program currently being executed.

In a multiprogramming system, the “user” part of memory must be further subdivided to accommodate multiple processes. The task of subdivision is carried out dynamically by the operating system and is known as **memory management**.

Binding of Instructions and Data to Memory

Address binding of instructions and data to memory addresses can happen at three different stages.

1. **Compile time**: The compile time is the time taken to compile the program or source code. During compilation, if memory location known a priori, then it generates absolute codes.
2. **Load time**: It is the time taken to link all related program file and load into the main memory. It must generate relocatable code if memory location is not known at compile time.
3. **Execution time**: It is the time taken to execute the program in main memory by processor. Binding delayed until run time if the process can be moved during its execution from one memory segment to another. Need hardware support for address maps (e.g., base and limit registers).



(Multistep processing of a user program.)

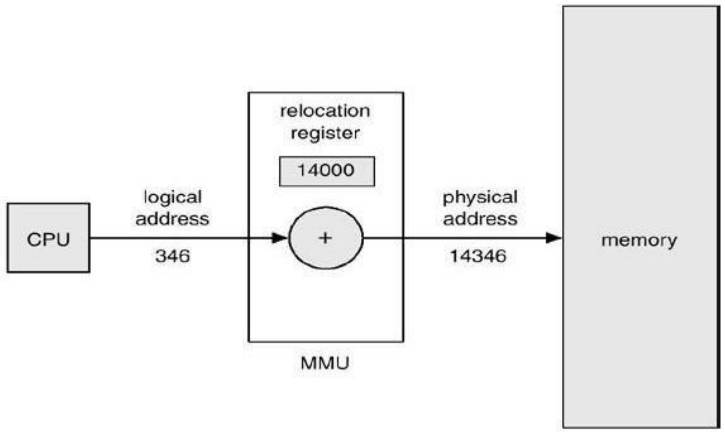
Logical- Versus Physical-Address Space

⇒ An address generated by the CPU is commonly referred to as a **logical address or** a **virtual address** whereas an address seen by the main memory unit is commonly referred to as a **physical address.**

⇒ The set of all logical addresses generated by a program is a **logical-address space whereas** the set of all physical addresses corresponding to these logical addresses is a **physical- address space.**

⇒ Logical and physical addresses are the same in compile-time and load-time address-binding schemes; logical (virtual) and physical addresses differ in execution-time address- binding scheme.

⇒ The Memory Management Unit is a hardware device that maps virtual to physical address. In MMU scheme, the value in the relocation register is added to every address generated by a user process at the time it is sent to memory as follows:



(Dynamic relocation using a relocation register)

Dynamic Loading

⇒ It loads the program and data dynamically into physical memory to obtain better memory-space utilization.

⇒ With dynamic loading, a routine is not loaded until it is called.

⇒ The advantage of dynamic loading is that an unused routine is never loaded.

⇒ This method is useful when large amounts of code are needed to handle infrequently occurring cases, such as error routines.

⇒ Dynamic loading does not require special support from the operating system.

Dynamic Linking

⇒ Linking postponed until execution time.

⇒ Small piece of code (stub) used to locate the appropriate memory-resident library routine.

⇒ Stub replaces itself with the address of the routine and executes the routine.

⇒ Operating system needed to check if routine is in processes memory address.

⇒ Dynamic linking is particularly useful for libraries.

Overlays

⇒ Keep in memory only those instructions and data that are needed at any given time.

⇒ Needed when process is larger than amount of memory allocated to it.

⇒ Implemented by user, no special support needed from operating system, programming design of overlay structure is complex.

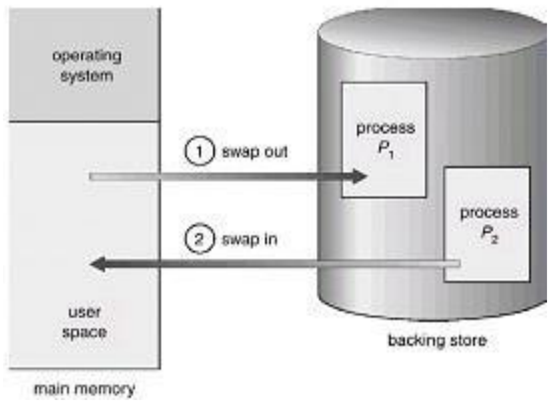
Swapping

⇒ A process can be swapped temporarily out of memory to a backing store (large disc), and then brought back into memory for continued execution.

⇒ **Roll out, roll in**: 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 called **roll out, roll in.**

⇒ Major part of swap time is transfer time; total transfer time is directly proportional to the amount of memory swapped.

⇒ Modified versions of swapping are found on many systems (UNIX, Linux, and Windows).



(Schematic View of Swapping)

MEMORY ALLOCATION

The main memory must accommodate both the operating system and the various user
processes. We need to allocate different parts of the main memory in the most efficient way possible.

The main memory is usually divided into two partitions: one for the resident operating system, and one for the user processes. We may place the operating system in either low memory or high memory. The major factor affecting this decision is the location of the interrupt vector. Since the interrupt vector is often in low memory, programmers usually place the operating system in low memory as well.

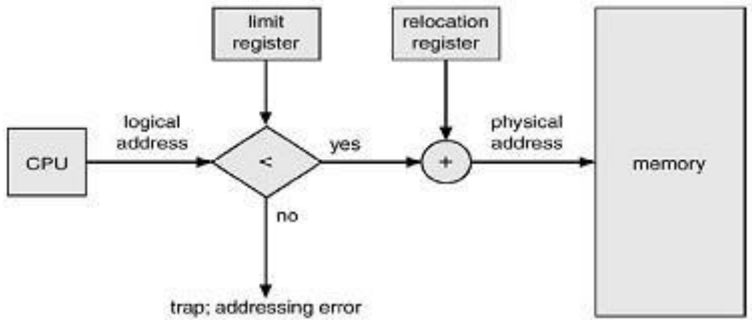
There are following two ways to allocate memory for user processes:

1. Contiguous memory allocation
2. Non contiguous memory allocation

1. Contiguous Memory Allocation

Here, all the processes are stored in contiguous memory locations. To load multiple processes into memory, the Operating System must divide memory into multiple partitions for those processes.

Hardware Support: The relocation-register scheme used to protect user processes from each other, and from changing operating system code and data. Relocation register contains value of smallest physical address of a partition and limit register contains range of that partition. Each logical address must be less than the limit register.



(Hardware support for relocation and limit registers)

According to size of partitions, the multiple partition schemes are divided into two types:

i. Multiple fixed partition/ multiprogramming with fixed task(MFT)
ii. Multiple variable partition/ multiprogramming with variable task(MVT)

i. <u>Multiple fixed partitions:</u> Main memory is divided into a number of static partitions at system generation time. In this case, any process whose size is less than or equal to the partition size can be loaded into any available partition. If all partitions are full and no process is in the Ready or Running state, the operating system can swap a process out of any of the partitions and load in another process, so that there is some work for the processor.

Advantages: Simple to implement and little operating system overhead.
Disadvantage: * Inefficient use of memory due to internal fragmentation.
* Maximum number of active processes is fixed.

ii. <u>Multiple variable partitions:</u> With this partitioning, the partitions are of variable length and number. When a process is brought into main memory, it is allocated exactly as much memory as it requires and no more.

Advantages: No internal fragmentation and more efficient use of main memory.
Disadvantages: Inefficient use of processor due to the need for compaction to counter external fragmentation.

Partition Selection policy:

When the multiple memory holes (partitions) are large enough to contain a process, the operating system must use an algorithm to select in which hole the process will be loaded. The partition selection algorithm are as follows:

⇒ **First-fit**: The OS looks at all sections of free memory. The process is allocated to the first hole found that is big enough size than the size of process.

⇒ **Next Fit:** The next fit search starts at the last hole allocated and The process is allocated to the next hole found that is big enough size than the size of process.

⇒ **Best-fit**: The Best Fit searches the entire list of holes to find the smallest hole that is big enough size than the size of process.

⇒ **Worst-fit**: The Worst Fit searches the entire list of holes to find the largest hole that is big enough size than the size of process.

Fragmentation: The wasting of memory space is called fragmentation. There are two types of fragmentation as follows:

1. **External Fragmentation:** The total memory space exists to satisfy a request, but it is not contiguous. This wasted space not allocated to any partition is called external fragmentation. The external fragmentation can be reduce by compaction. The goal is to shuffle the memory contents to place all free memory together in one large block. Compaction is possible only if relocation is dynamic, and is done at execution time.
2. **Internal Fragmentation:** The allocated memory may be slightly larger than requested memory. The wasted space within a partition is called internal fragmentation. One method to reduce internal fragmentation is to use partitions of different size.

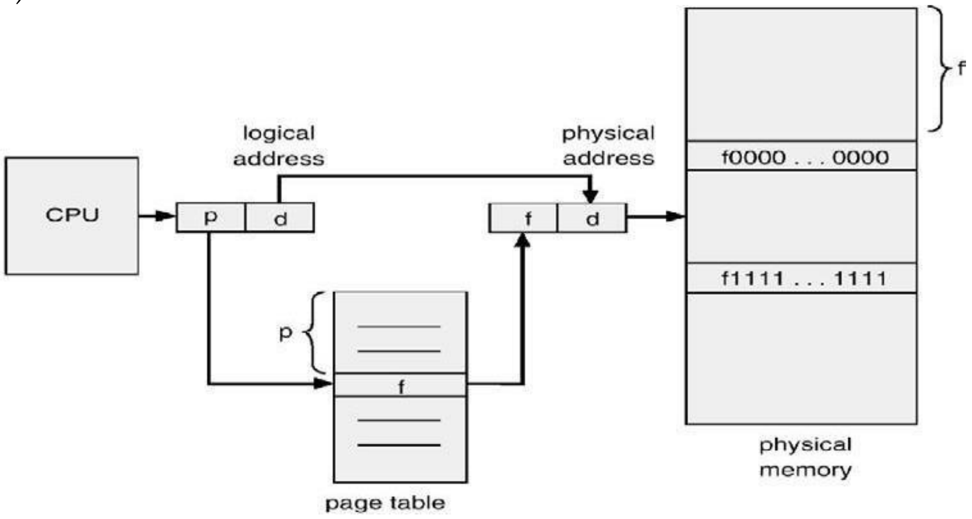
2. Noncontiguous memory allocation

In noncontiguous memory allocation, it is allowed to store the processes in non contiguous memory locations. There are different techniques used to load processes into memory, as follows:

1. Paging
2. Segmentation
3. Virtual memory paging(Demand paging) etc.

PAGING

Main memory is divided into a number of equal-size blocks, are called **frames**. Each process is divided into a number of equal-size block of the same length as frames, are called **Pages**. A process is loaded by loading all of its pages into available frames (may not be contiguous).



(Diagram of Paging hardware)

Process of Translation from logical to physical addresses

⇒ 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.

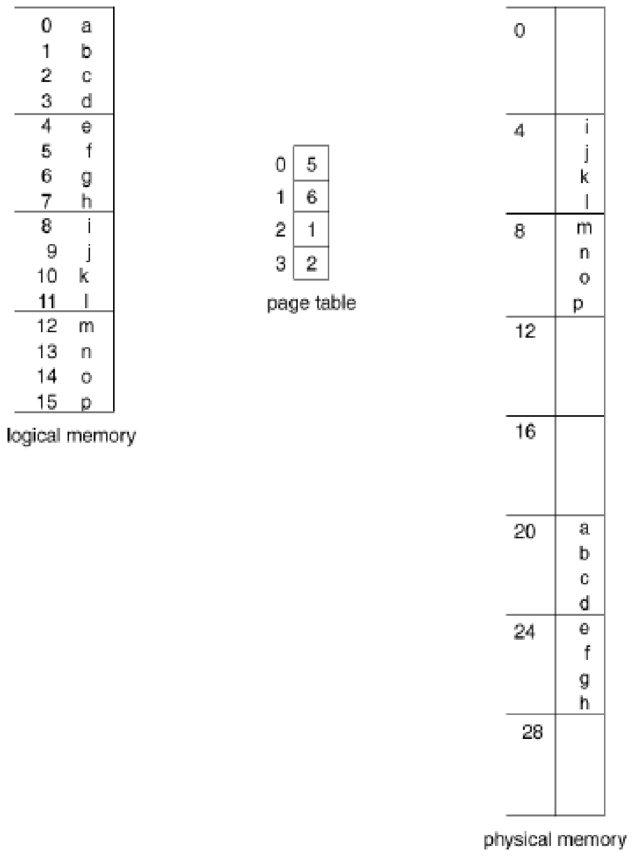
⇒ If the size of logical-address space is 2^m and a page size is 2^n addressing units (bytes or words), then the high-order (m – n) bits of a logical address designate the page number and the n low-order bits designate the page offset. Thus, the logical address is as follows:

page number	page offset
p	d
$m - n$	n

Where p is an index into the page table and d is the displacement within the page.

Example:

Consider a page size of 4 bytes and a physical memory of 32 bytes (8 pages), we show how the user's view of memory can be mapped into physical memory. Logical address 0 is page 0, offset 0. Indexing into the page table, we find that page 0 is in frame 5. Thus, logical address 0 maps to physical address 20 (= (5 x 4) + 0). Logical address 3 (page 0, offset 3) maps to physical address 23 (= (5 x 4) + 3). Logical address 4 is page 1, offset 0; according to the page table, page 1 is mapped to frame
6. Thus, logical address 4 maps to physical address 24 (= (6 x 4) + 0). Logical address 13 maps to physical address 9(= (2 x 4)+1).



Hardware Support for Paging:

Each operating system has its own methods for storing page tables. Most operating systems allocate a page table for each process. A pointer to the page table is stored with the other register values (like the instruction counter) in the process control block. When the dispatcher is told to start a process, it must reload the user registers and define the correct hardware page table values from the stored user page table.

Implementation of Page Table

⇒ Generally, Page table is kept in main memory. The Page Table Base Register (PTBR) points to the page table. And Page-table length register (PRLR) indicates size of the page table.

⇒ In this scheme every data/instruction access requires two memory accesses. One for the page table and one for the data/instruction.

⇒ The two memory access problem can be solved by the use of a special fast-lookup hardware cache called **associative memory** or **translation look-aside buffers (TLBs)**.

Paging Hardware With TLB

The TLB is an associative and high-speed memory. Each entry in the TLB consists of two parts: a key (or tag) and a value. The TLB is used with page tables in the following way.

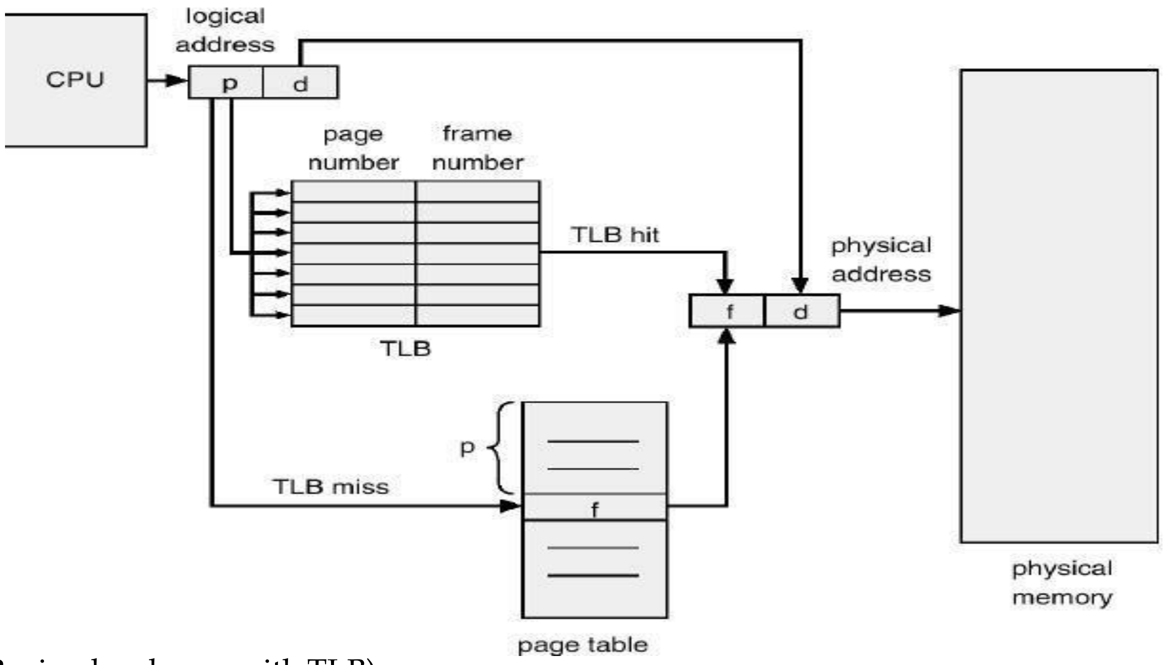
⇒ The TLB contains only a few of the page-table entries. When a logical address is generated by the CPU, its page number is presented to the TLB.

⇒ If the page number is found (known as a **TLB Hit**), its frame number is immediately available and is used to access memory. It takes only one memory access.

⇒ If the page number is not in the TLB (known as a **TLB** miss), a memory reference to the page table must be made. When the frame number is obtained, we can use it to access memory. It takes two memory accesses.

⇒ In addition, it stores the page number and frame number to the TLB, so that they will be found quickly on the next reference.

⇒ If the TLB is already full of entries, the operating system must select one for replacement by using replacement algorithm.



(Paging hardware with TLB)

The percentage of times that a particular page number is found in the TLB is called the

hit ratio. The effective access time (EAT) is obtained as follows:

EAT= HR x (TLBAT + MAT) + MR x (TLBAT + 2 x MAT)

Where HR: Hit Ratio, TLBAT: TLB access time, MAT: Memory access time, MR: Miss Ratio.

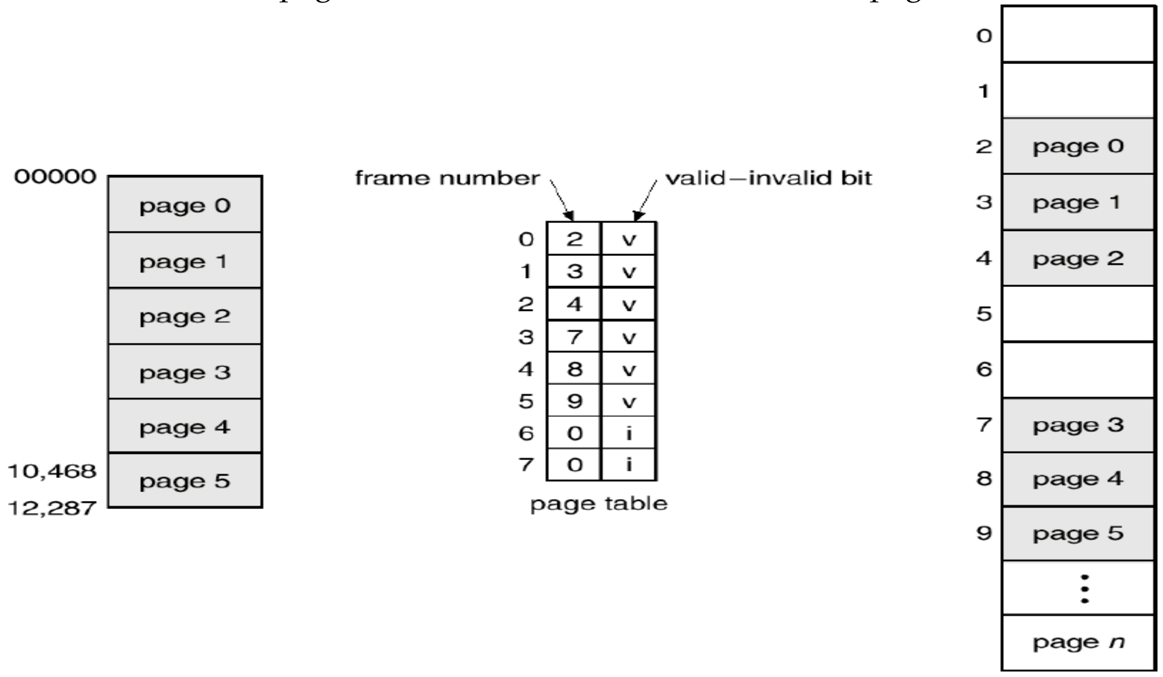
Memory protection in Paged Environment:

⇒ Memory protection in a paged environment is accomplished by protection bits that are associated with each frame. These bits are kept in the page table.

⇒ One bit can define a page to be read-write or read-only. This protection bit can be checked to verify that no writes are being made to a read-only page. An attempt to write to a read- only page causes a hardware trap to the operating system (or memory-protection violation).

⇒ One more bit is attached to each entry in the page table: a **valid-invalid** bit. When this bit is set to "valid," this value indicates that the associated page is in the process' logical- address space, and is a legal (or valid) page. If the bit is set to "invalid," this value indicates that the page is not in the process' logical-address space.

⇒ Illegal addresses are trapped by using the valid-invalid bit. The operating system sets this bit for each page to allow or disallow accesses to that page.

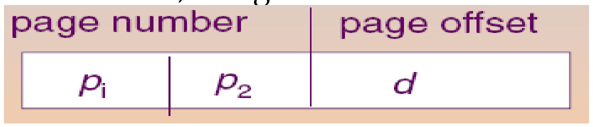


(Valid (v) or invalid (i) bit in a page table)

Structure of the Page Table

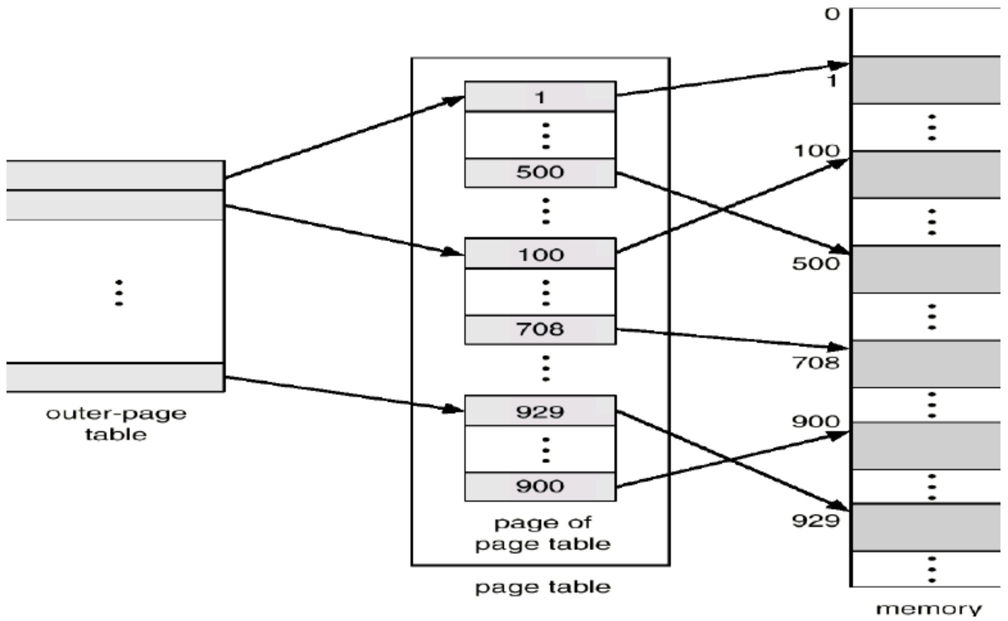
There are different structures of page table described as follows:

1. **Hierarchical Page table:** When the number of pages is very high, then the page table takes large amount of memory space. In such cases, we use multilevel paging scheme for reducing size of page table. A simple technique is a two-level page table. Since the page table is paged, the page number is further divided into parts: page number and page offset. Thus, a logical address is as follows:

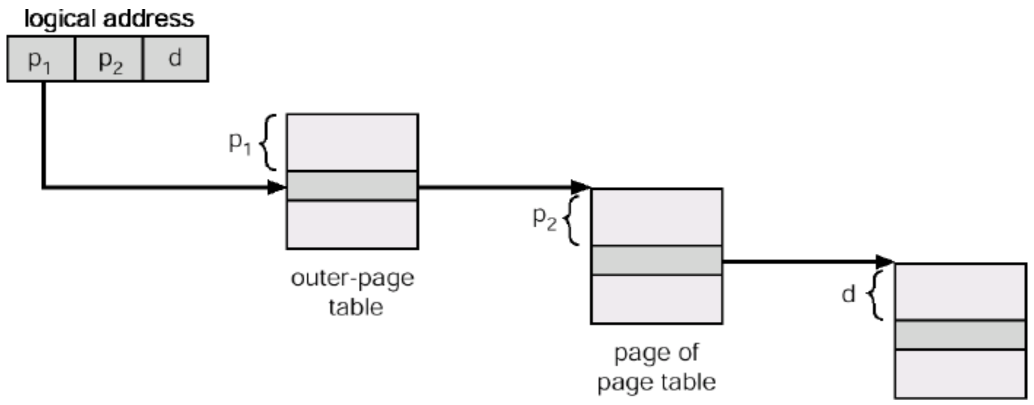


Where p_i is an index into the outer page table, and p2 is the displacement within the page of the outer page table.

Two-Level Page-Table Scheme:

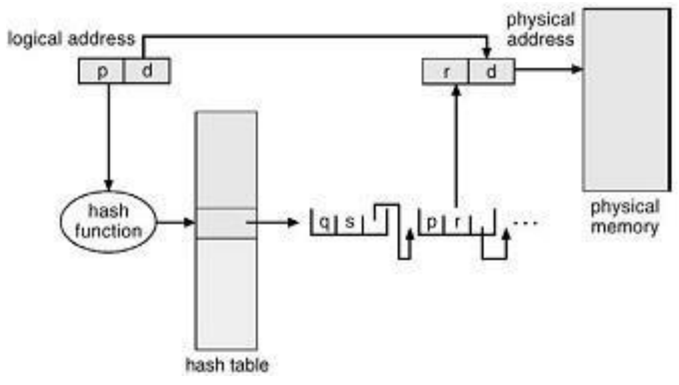


Address translation scheme for a two-level paging architecture:



2. Hashed Page Tables:

This scheme is applicable for address space larger than 32bits. In this scheme, the virtual page number is hashed into a page table. This page table contains a chain of elements hashing to the same location. Virtual page numbers are compared in this chain searching for a match. If a match is found, the corresponding physical frame is extracted.

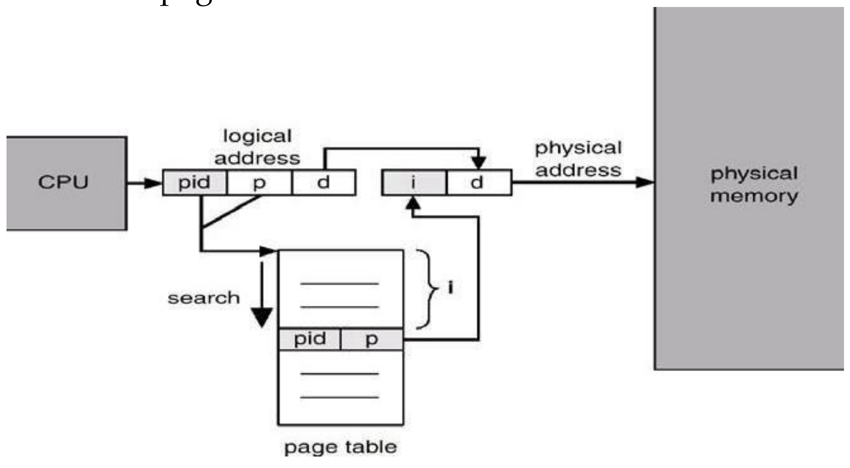


3. **Inverted Page Table:**

⇒ One entry for each real page of memory.

⇒ Entry consists of the virtual address of the page stored in that real memory location, with information about the process that owns that page.

⇒ Decreases memory needed to store each page table, but increases time needed to search the table when a page reference occurs.



Shared Pages

Shared code

⇒ One copy of read-only (reentrant) code shared among processes (i.e., text editors, compilers, window systems).

⇒ Shared code must appear in same location in the logical address space of all processes.

Private code and data

⇒ Each process keeps a separate copy of the code and data.

⇒ The pages for the private code and data can appear anywhere in the logical address space.

SEGMENTATION

Segmentation is a memory-management scheme that supports user view of memory. A program is a collection of segments. A segment is a logical unit such as: main program, procedure, function, method, object, local variables, global variables, common block, stack, symbol table, arrays etc.

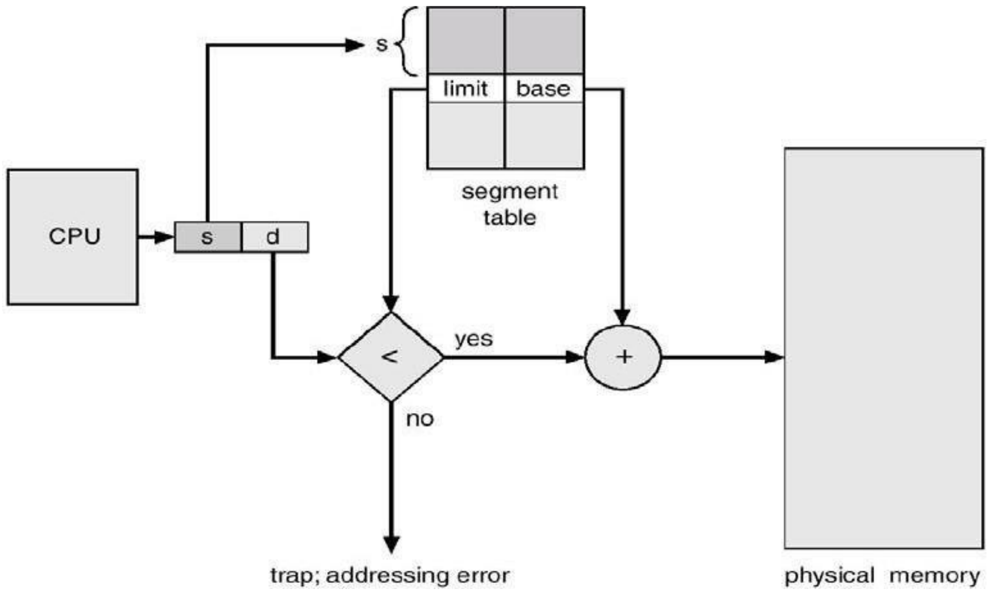
A logical-address space is a collection of segments. Each segment has a name and a length. The user specifies each address by two quantities: a segment name/number and an offset.

Hence, Logical address consists of a two tuple: <segment-number, offset>

Segment table maps two-dimensional physical addresses and each entry in table has: **base** – contains the starting physical address where the segments reside in memory. **limit** – specifies the length of the segment.

Segment-table base register (STBR) points to the segment table's location in memory.

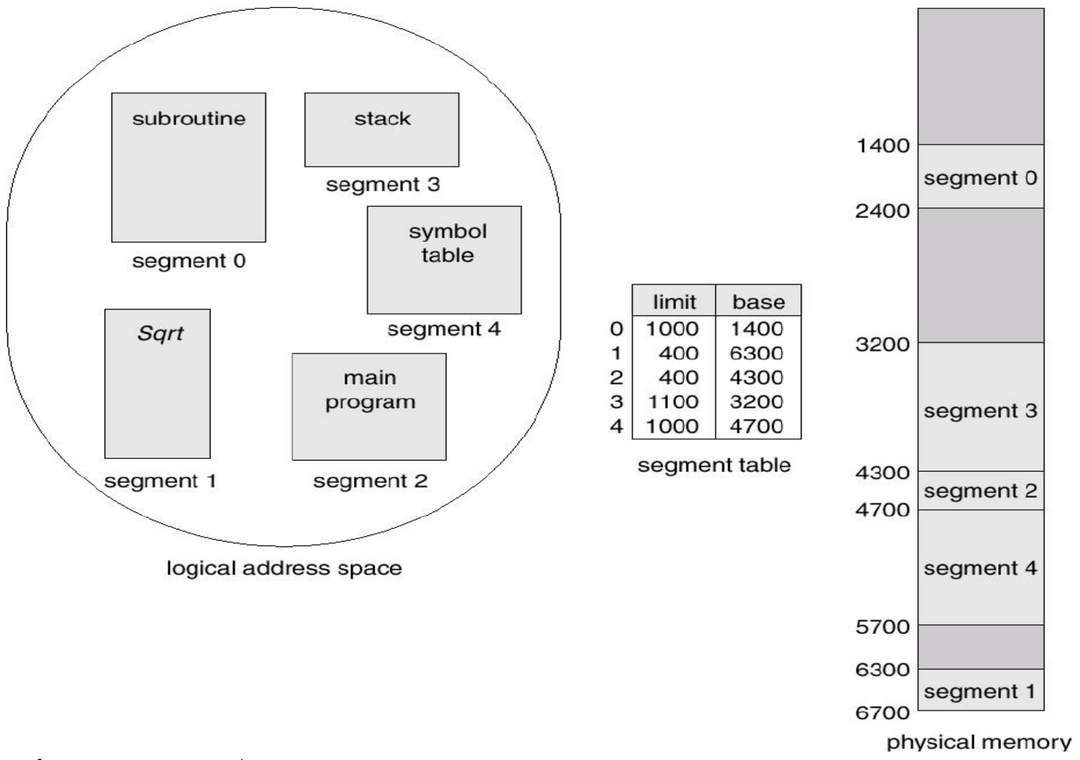
Segment-table length register (STLR) indicates number of segments used by a program.



(Diagram of Segmentation Hardware)

The segment number is used as an index into the segment table. The offset d of the logical address must be between 0 and the segment limit. If it is not, we trap to the operating system that logical addressing attempt beyond end of segment. If this offset is legal, it is added to the segment base to produce the address in physical memory of the desired byte.

Consider we have five segments numbered from 0 through 4. The segments are stored in physical memory as shown in figure. The segment table has a separate entry for each segment, giving start address in physical memory (or base) and the length of that segment (or limit). For example, 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.



	limit	base
0	1000	1400
1	400	6300
2	400	4300
3	1100	3200
4	1000	4700

(Example of segmentation)