UNIT-3 PART-A

System Model

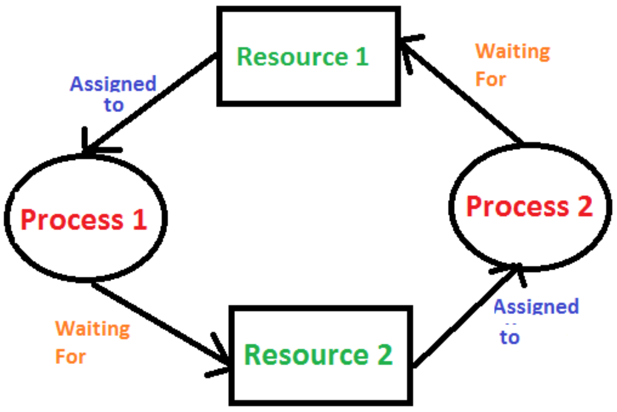
A system can be modeled as a collection of limited resources, which can be partitioned into different categories, to be allocated to a number of processes, each having different needs. Resource categories may include memory, printers, CPUs, open files, tape drives, CD-ROMS, etc. Some categories may have a single resource.

In normal operation a process must request a resource before using it, and release it when it is done, in the following sequence:

- Request - If the request cannot be immediately granted, then the process must wait until the resource(s) it needs become available. For example the system calls open( ), malloc( ), new( ), and request( ).
- Use - The process uses the resource, e.g. prints to the printer or reads from the file.
- Release - The process relinquishes the resource. so that it becomes available for other processes. For example, close( ), free( ), delete( ), and release( ).

A set of processes is deadlocked when every process in the set is waiting for a resource that is currently allocated to another process in the set (and which can only be released when that other waiting process makes progress.)

Deadlock is a situation where a set of processes are blocked because each process is holding a resource and waiting for another resource acquired by some other process. Consider an example when two trains are coming toward each other on same track and there is only one track, none of the trains can move once they are in front of each other. Similar situation occurs in operating systems when there are two or more processes hold some resources and wait for resources held by other(s). For example, in the below diagram, Process 1 is holding Resource 1 and waiting for resource 2 which is acquired by process 2, and process 2 is waiting for resource 1.



Necessary Conditions

There are four conditions that are necessary to achieve deadlock:

- **Mutual Exclusion** - At least one resource must be held in a non-sharable mode; If any other process requests this resource, then that process must wait for the resource to be released.
- **Hold and Wait** - A process must be simultaneously holding at least one resource and waiting for at least one resource that is currently being held by some other process.
- **No preemption** - Once a process is holding a resource (i.e. once its request has been granted), then that resource cannot be taken away from that process until the process voluntarily releases it.
- **Circular Wait** - A set of processes { P_0, P_1, P_2, . . ., P_N } must exist such that every P[i] is waiting for P[(i + 1) % (N + 1)].

Methods for Handling Deadlocks

Deadlock prevention

Recovery from Deadlock

Deadlock detection and avoidance

Deadlock ignorance.

- Deadlocks can be prevented by preventing at least one of the four required conditions:

Mutual Exclusion

- Shared resources such as read-only files do not lead to deadlocks.
- Unfortunately some resources, such as printers and tape drives, require exclusive access by a single process.

Hold and Wait

To prevent this condition processes must be prevented from holding one or more resources while simultaneously waiting for one or more others. There are several possibilities for this:

- Require that all processes request all resources at one time. This can be wasteful of system resources if a process needs one resource early in its execution and doesn't need some other resource until much later.
- Require that processes holding resources must release them before requesting new resources, and then re-acquire the released resources along with the new ones in a single new request. This can be a problem if a process has partially completed an operation using a resource and then fails to get it re-allocated after releasing it.
- Either of the methods described above can lead to starvation if a process requires one or more popular resources.

No Preemption *Preemption of process resource allocations can prevent this condition of deadlocks, when it is possible.*

- One approach is that if a process is forced to wait when requesting a new resource, then all other resources previously held by this process are implicitly released, (preempted), forcing this process to re-acquire the old resources along with the new resources in a single request, similar to the previous discussion.
- Another approach is that when a resource is requested and not available, then the system looks to see what other processes currently have those resources *and* are them blocked waiting for some other resource. If such a process is found, then some of their resources may get preempted and added to the list of resources for which the process is waiting.
- Either of these approaches may be applicable for resources whose states are easily saved and restored, such as registers and memory, but are generally not applicable to other devices such as printers and tape drives.

Circular Wait

- One way to avoid circular wait is to number all resources, and to require that processes request resources only in strictly increasing (or decreasing) order.
- In order to request resource Rj, a process must first release all Ri such that i>= j.
- Big challenge in this scheme is determining the relative ordering of different resource

Recovery from Deadlock

There are three basic approaches to recovery from deadlock:

Process Termination

Two basic approaches, both of which recover resources allocated to terminate processes:

- Terminate all processes involved in the deadlock. This definitely solves the deadlock, but at the expense of terminating more processes than would be absolutely necessary.
- Terminate processes one by one until the deadlock is broken. This is more conservative, but requires doing deadlock detection after each step.
- In the latter case there are many factors that can go into deciding which processes to terminate next:

    1. Process priorities.
    2. How long the process has been running, and how close it is to finishing.
    3. How many and what type of resources is the process holding. (Are they easy to preempt and restore?)
    4. How many more resources does the process need to complete.
    5. How many processes will need to be terminated
    6. Whether the process is interactive or batch.
    7. Whether or not the process has made non-restorable changes to any resource.

Resource Preemption

When preempting resources to relieve deadlock, there are three important issues to be addressed:

- **Selecting a victim** - Deciding which resources to preempt from which processes involves many of the same decision criteria outlined.
- **Rollback** - Ideally one would like to roll back a preempted process to a safe state prior to the point at which that resource was originally allocated to the process. Unfortunately it can be difficult or impossible to determine what such a safe state is, and so the only safe rollback is to roll back all the way back to the beginning. (i.e. abort the process and make it start over.)

Deadlock detection and avoidance

Detection:

Resource-Allocation Graph *In some cases deadlocks can be understood more clearly through the use of* ***Resource-Allocation Graphs****, having the following properties:*

A set of resource categories, { R_1, R_2, R_3, . . ., R_N }, which appear as square nodes on the graph. Dots inside the resource nodes indicate specific instances of the resource. (E.g. two dots might represent two laser printers.)

A set of processes, { P_1, P_2, P_3, . . ., P_N }

Request Edges - A set of directed arcs from P_i to Rj, indicating that process Pi has requested Rj, and is currently waiting for that resource to become available.

Assignment Edges - A set of directed arcs from Rj to Pi indicating that resource Rj has been allocated to process Pi, and that Pi is currently holding resource Rj.

Note that a **request edge** can be converted into an **assignment edge** by reversing the direction of the arc when the request is granted. For example:

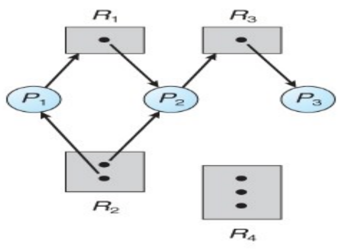


Figure : Resource allocation graph

- If a resource-allocation graph contains no cycles, then the system is not deadlocked. If a resource-allocation graph does contain cycles **AND** each resource category contains only a single instance, then a deadlock exists.
- If a resource category contains more than one instance, then the presence of a cycle in the resource-allocation graph indicates the *possibility* of a deadlock, but does not guarantee one. Consider, for example,
- **Wait-For Graph**
- The Wait-For Graph Algorithm is a deadlock detection algorithm used to detect deadlocks in a system where resources can have multiple instances. The algorithm works by constructing a Wait-For Graph, which is a directed graph that represents the dependencies between processes and resources.
-

Single Instance of Each Resource Type

- If each resource category has a single instance, then we can use a variation of the resource-allocation graph known as a ***wait-for graph***.
- A wait-for graph can be constructed from a resource-allocation graph by eliminating the resources and collapsing the associated edges, as shown in the figure below.
- An arc from Pi to Pj in a wait-for graph indicates that process Pi is waiting for a resource that process Pj is currently holding.

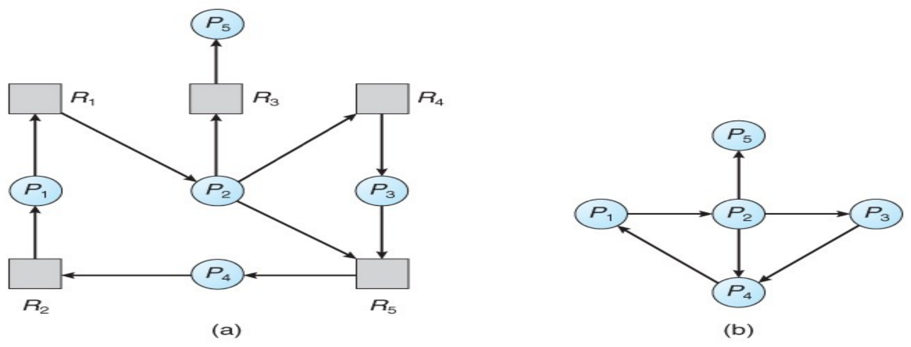


Figure - (a) Resource allocation graph. (b) Corresponding wait-for graph

- As before, cycles in the wait-for graph indicate deadlocks.
- This algorithm must maintain the wait-for graph, and periodically search it for cycles.

Multiple instances RAG: Detection of the cycle is necessary but not a sufficient condition for deadlock detection, in this case, the system may or may not be in deadlock varies according to different situations.

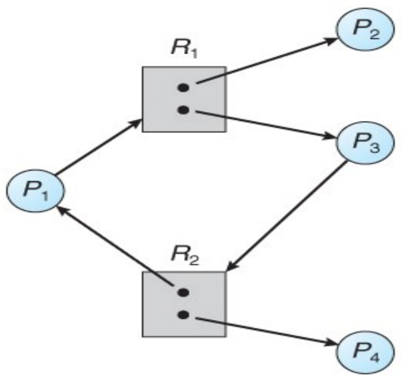


Figure - Resource allocation graph with a cycle but no deadlock

AVOIDANCE:

Safe State

A state is ***safe*** if the system can allocate all resources requested by all processes (up to their stated maximums) without entering a deadlock state. A state is safe if there exists a ***safe sequence*** of processes { P_0, P_1, P_2, ..., P_N } such that all of the resource requests for Pi can be granted using the resources currently allocated to Pi and all processes Pj where j <i. (i.e. if all the processes prior to Pi finish and free up their resources, then Pi will be able to finish also, using the resources that they have freed up.)

If a safe sequence does not exist, then the system is in an unsafe state, which ***MAY*** lead to deadlock. (All safe states are deadlock free, but not all unsafe states lead to deadlocks.)

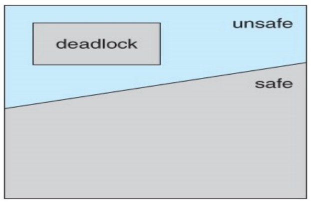


Figure - Safe, unsafe, and deadlocked state spaces.

For example, consider a system with 12 tape drives, allocated as follows. Is this a safe state? What is the safe sequence?

	Maximum Needs	**Current Allocation**
P0	10	5
P1	4	2
P2	9	2

- What happens to above table if process P2 requests & is granted one more tape drive?
- Key to the safe state approach is that when a request is made for resources, the request is granted only if the resulting allocation state is a safe one.

Resource-Allocation Graph Algorithm

- If resource categories have only single instances of their resources, then deadlock states can be detected by cycles in the resource-allocation graphs.
- In this case, unsafe states can be recognized and avoided by augmenting the resource-allocation graph with ***claim edges***, noted by dashed lines, which point from a process to a resource that it may request in the future.
- In order for this technique to work, all claim edges must be added to the graph for any particular process before that process is allowed to request any resources. (Alternatively, processes may only make requests for resources for which they have already established claim edges, and claim edges cannot be added to any process that is currently holding resources.)

- When a process makes a request, the claim edge Pi->Rj is converted to a request edge. Similarly when a resource is released, the assignment reverts back to a claim edge.
- This approach works by denying requests that would produce cycles in the resource-allocation graph, taking claim edges into effect.

Consider for example what happens when process P2 requests resource R2:

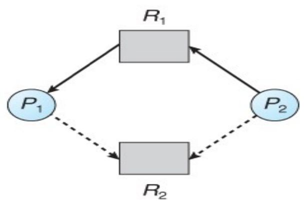


Figure - Resource allocation graph for deadlock avoidance

- The resulting resource-allocation graph would have a cycle in it, and so the request cannot be granted.

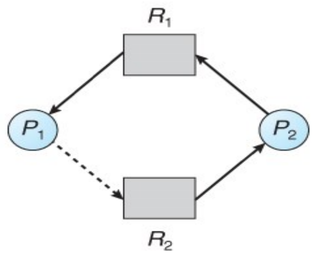


Figure - An unsafe state in a resource allocation graph

Banker's Algorithm

For resource categories that contain more than one instance the resource-allocation graph method does not work, and more complex (and less efficient) methods must be chosen.

- The Banker's Algorithm gets its name because it is a method that bankers could use to assure that when they lend out resources they will still be able to satisfy all their clients.
- When a process starts up, it must state in advance the maximum allocation of resources it may request, up to the amount available on the system.When a request is made, the scheduler determines whether granting the request would leave system

in a safe state. If not, then the process must wait until the request can be granted safely.

The banker's algorithm relies on several key data structures: (where n is the number of processes and m is the number of resource categories.)

- Available[m] indicates how many resources are currently available of each type.
- Max[n][m] indicates the maximum demand of each process of each resource.
- Allocation[n][m] indicates the number of each resource category allocated to each process.
- Need[n][m] indicates the remaining resources needed of each type for each process. (Note that Need[i][j] = Max[i][j] - Allocation[i][j] for all i, j.)

For simplification of discussions, we make the following notations / observations:

- One row of the Need vector, Need[i], can be treated as a vector corresponding to the needs of process i, and similarly for Allocation and Max.
- A vector X is considered to be <= a vector Y if X[i] <= Y[i] for all i.

Safety Algorithm

In order to apply the Banker's algorithm, we first need an algorithm for determining whether or not a particular state is safe.This algorithm determines if the current state of a system is safe, according to the following steps:

1) Let Work and Finish be vectors of length m and n respectively.

- Work is a working copy of the available resources, which will be modified during the analysis.
- Finish is a vector of Boolean indicating whether a particular process can finish. (or has finished so far in the analysis.)
- Initialize Work to Available, and Finish to false for all elements.

2) Find an i such that both (A) Finish[i] == false, and (B) Need[i] < Work. This process hasn’t finished,but could with the given available working set. If no such i exists, go to step4.

3) Set Work = Work + Allocation[i], and set Finish[i] to true. This corresponds to process i finishing up and releasing its resources back into the work pool.Then loop back to step 2.

4) If finish[i] == true for all i, then the state is a safe state, because a safe sequence is found.

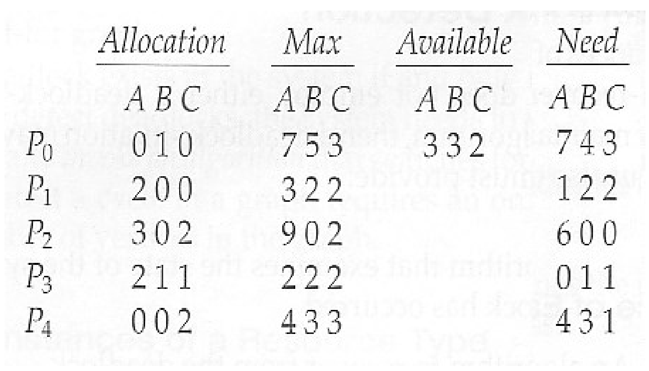
Resource-Request Algorithm (The Bankers Algorithm)

Now that we have a tool for determining if a particular state is safe or not, we are now ready to look at the Banker's algorithm itself. This algorithm determines if a new request is safe, and grants it only if it is safe to do so.

- When a request is made (that does not exceed currently available resources), pretend it has been granted, and then see if the resulting state is a safe one. If so, grant the request, and if not, deny the request, as follows:
- Let Request[n][m] indicate the number of resources of each type currently requested by processes. If Request[i] > Need[i] for any process i, raise an error condition.
- If Request[i] > Available for any process i, then that process must wait for resources to become available. Otherwise the process can continue to step 3.
- Check to see if the request can be granted safely, by pretending it has been granted and then seeing if the resulting state is safe. If so, grant the request, and if not, then the process must wait until its request can be granted safely. The procedure for granting a request (or pretending to for testing purposes) is:
    - Available = Available - Request
    - Allocation = Allocation + Request
    - Need = Need - Request

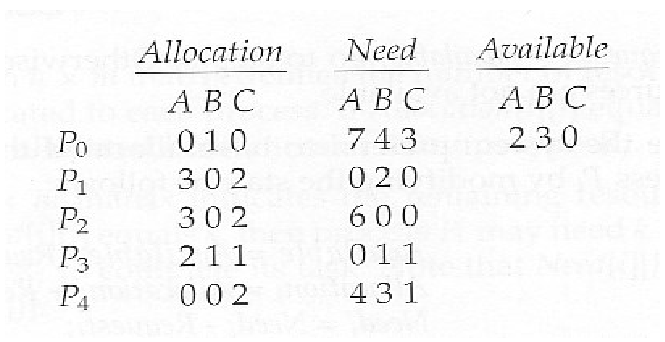
An Illustrative Example

- Consider the following situation:



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

- And now consider what happens if process P1 requests 1 instance of A and 2 instances of C. (Request[1] = (1, 0, 2))

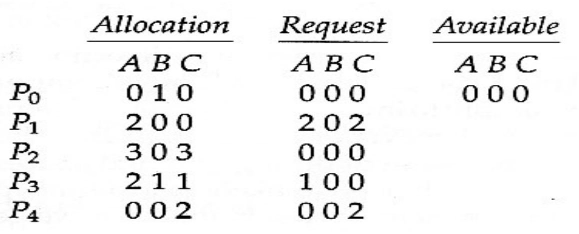


	Allocation	Need	Available
	A B C	A B C	A B C
P_0	0 1 0	7 4 3	2 3 0
P_1	3 0 2	0 2 0	
P_2	3 0 2	6 0 0	
P_3	2 1 1	0 1 1	
P_4	0 0 2	4 3 1	

- What about requests of (3, 3,0) by P4? or (0, 2, 0) by P0? Can these be safely granted? Why or why not?

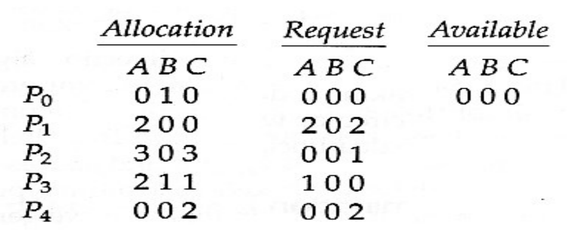
Deadlock Detection

- If deadlocks are not avoided, then another approach is to detect when they have occurred and recover somehow.
- In addition to the performance hit of constantly checking for deadlocks, a policy / algorithm must be in place for recovering from deadlocks, and there is potential for lost work when processes must be aborted or have their resources preempted.



	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	

- Now suppose that process P2 makes a request for an additional instance of type C, yielding the state shown below. Is the system now deadlocked?



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

Unit -3 part B

memory management

Basic concept

Program must be brought into memory and placed within

a process for it to be run.

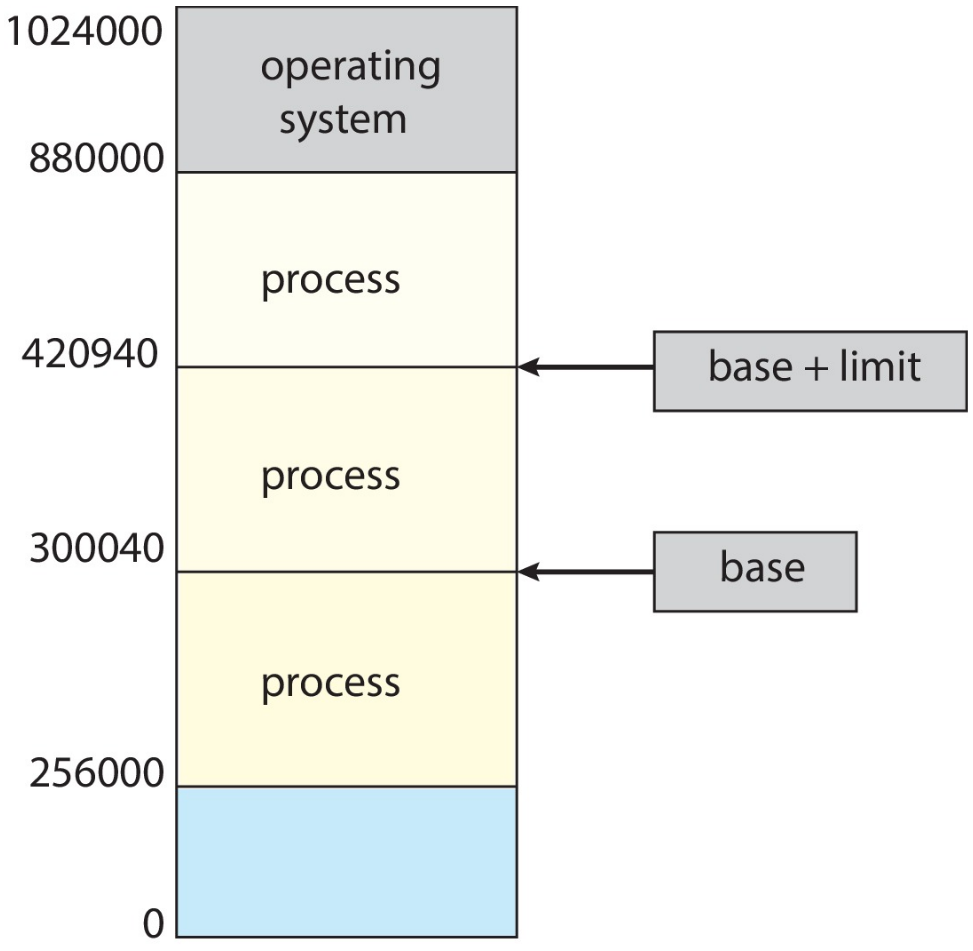
Base and Limit registers

To protect processes from each; each process has its own memory space.

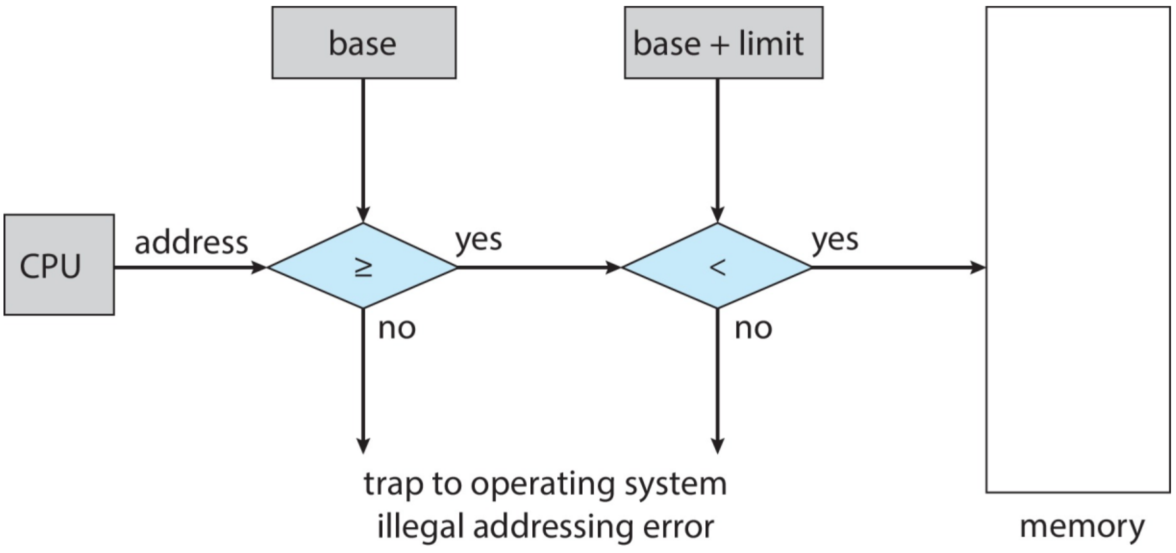
For each process, the base and limit registers define its logical address space.

Base register holds the smallest legal physical memory address(starting address of process).

limit register specifies the size of the range.



Hardware Address protection with Base and Limit registers

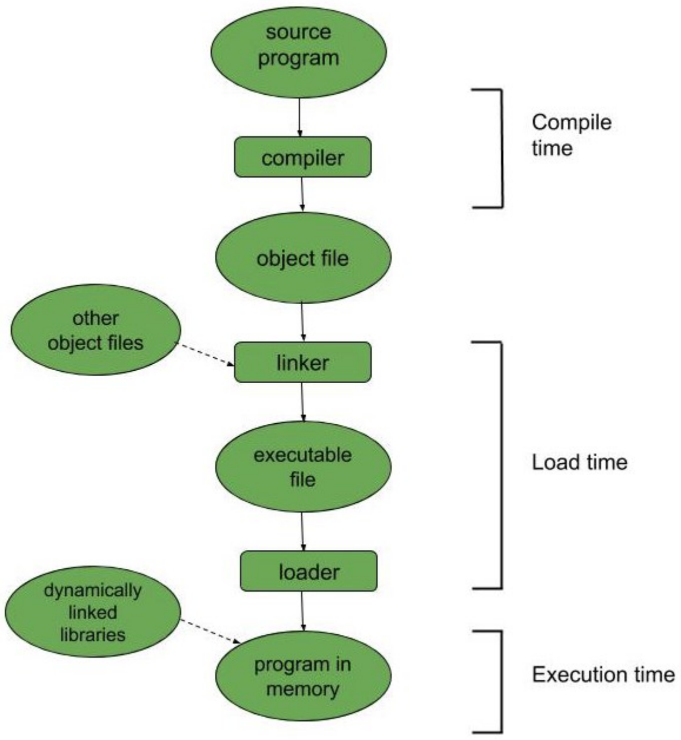


CPU must check every memory access generated in user mode to be sure it is between base and limit for that user; to protect a process's memory space.

Base and limit registers loaded only by the OS through a privileged instruction.

Multistep Processing of a User Program

The Binding of instructions and data to memory addresses can be done at any step



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

Compile time: If memory location known a priori, absolute code can be generated must recompile code if starting location changes.

Load time: Must generate relocatable code if memory location is not known at compile time.

Execution time: 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).

Logical vs Physical Address :

Logical Address is the address generated by the CPU whereas Physical Address is the one seen by the memory unit.

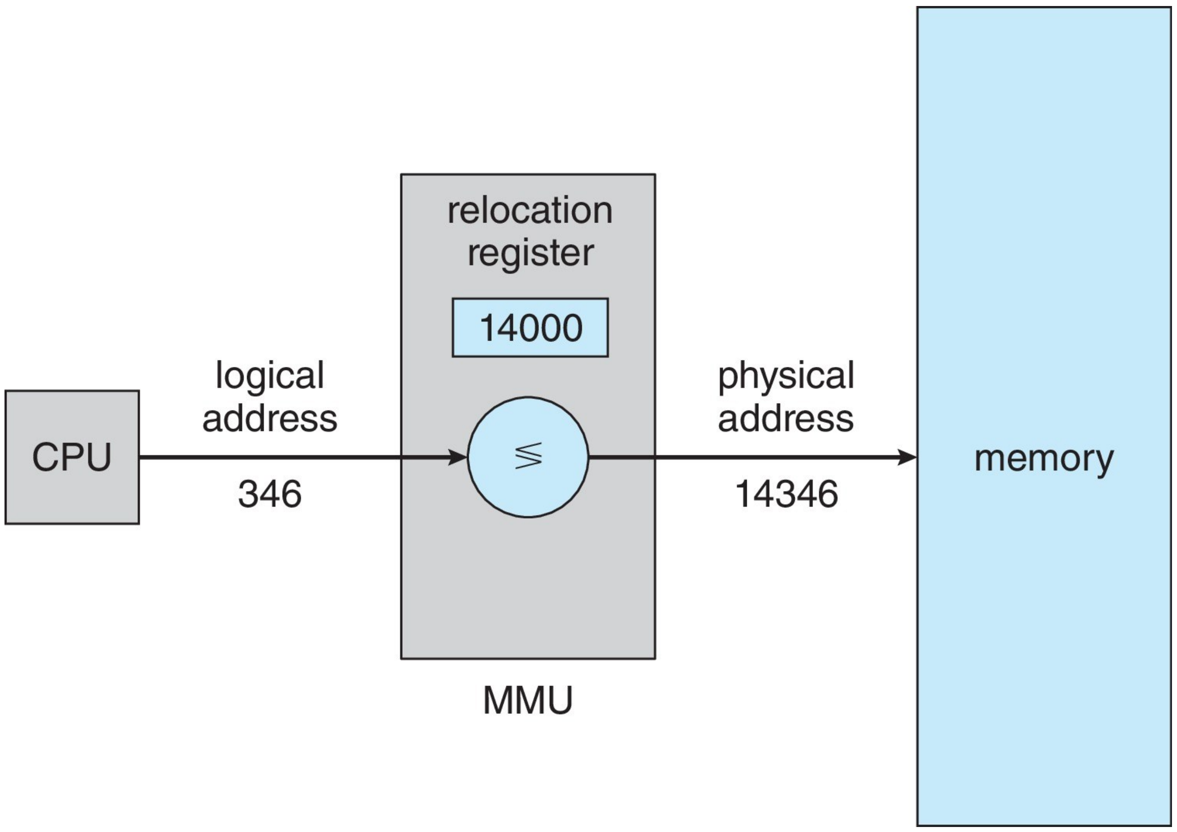
The user program deals with the logical addresses and one never sees the physical addresses.

S.No	Logical Address	Physical Address
1.	Users can access the **logical address** of the Program.	User can never access the **physical address** of the Program
2.	The logical address is generated by the CPU.	The physical address is located in the memory unit.
3.	The user can access the physical address with the help of a logical address.	A physical address can be accessed by a user indirectly b ut not directly.
4.	The logical address does not exist physically in the memory and thus termed as a Virtual address.	On the other hand, the physical address is a location in the memory. Thus it can be accessed physically.
5.	The set of all logical addresses that are generated by any program is referred to as **Logical Address Space.**	The set of all **physical addresses** corresponding to the **Logical addresses** is commonly known as **Physical Address Space.**
6.	This address is generated by the CPU.	It is computed by the Memory Management Unit(MMU).

Memory-Management Unit (MMU)

MMU is the hardware that does virtual-to-physical addr mappings at run-time.

Base register now called relocation register.



Dynamic Loading :

Dynamic Loading is a concept in which program is loaded only when it is required.

Logic address space and Physical address space:

Logical Address Space is set of all logical addresses generated by CPU in reference to a program.

Physical Address is set of all physical addresses mapped to the corresponding logical addresses.

9.1.5 Dynamic Linking and Shared Libraries

The in-memory program text originally contains a stub for each reference that the program has to a library routine. The stub is a piece of code that tells where in memory or on disk to locate the library routine.

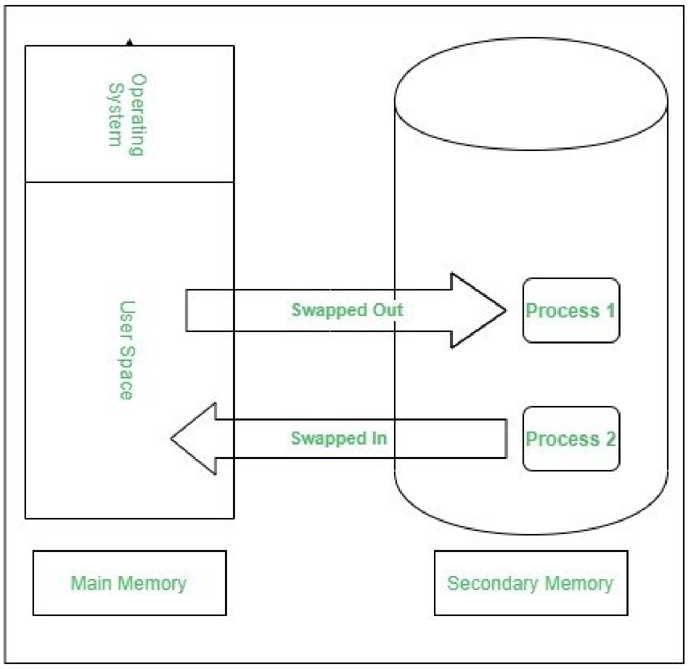
When the program first executes the stub, the stub replaces itself with with the address of the routine and executes it. (If need be, it first loads the routine.)

All processes share the same copy of each library routine.

Because of memory management, user processes need help from the OS to check on the memory locations of routines. The sharing of library routines requires help from the hardware and the OS - shared memory.

Swapping

A process needs to be in memory for execution. But sometimes there is not enough main memory to hold all the currently active processes in a timesharing system. So, the excess process is kept on disk and brought in to run dynamically. Swapping is the process of bringing in each process in the main memory, running it for a while, and then putting it back to the disk.



Swapping has been subdivided into two concepts: swap-in and swap-out.

Swap-out is a technique for moving a process from RAM to the hard disc.

Swap-in is a method of transferring a program from a hard disc to main memory, or RAM.

Contiguous Memory Allocation

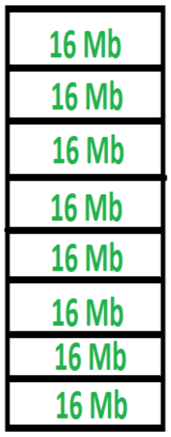
There are two primary allocation methods in os: contiguous and non-contiguous allocation methods.

Contiguous memory allocation is a technique where the operating system allocates a contiguous block of memory to a process.

Contiguous memory allocation can be implemented by 2 ways

MFT (Multiprogramming with a Fixed number of Tasks) is one of the old memory management techniques in which the memory is partitioned into fixed size partitions and each job is assigned to a partition.

Multi-programming with fixed partitioning is a contiguous memory management technique in which the main memory is divided into fixed sized partitions which can be of equal or unequal size. Whenever we have to allocate a process memory then a free partition that is big enough to hold the process is found. Then the memory is allocated to the process.If there is no free space available then the process waits in the queue to be allocated memory. It is one of the most oldest memory management technique which is easy to implement.



Advantages of Fixed-size Partition Scheme

- This scheme is simple and is easy to implement
- It supports multiprogramming as multiple processes can be stored inside the main memory.
- Management is easy using this scheme

Disadvantages of Fixed-size Partition Scheme

Some disadvantages of using this scheme are as follows:

1. Internal Fragmentation

Suppose the size of the process is lesser than the size of the partition in that case some size of the partition gets wasted and remains unused. This wastage inside the memory is generally termed as Internal fragmentation

As we have shown in the above diagram the 70 KB partition is used to load a process of 50 KB so the remaining 20 KB got wasted.

2. Limitation on the size of the process

If in a case size of a process is more than that of a maximum-sized partition then that process cannot be loaded into the memory. Due to this, a condition is imposed on the size of the process and it is: the size of the process cannot be larger than the size of the largest partition.

3. External Fragmentation

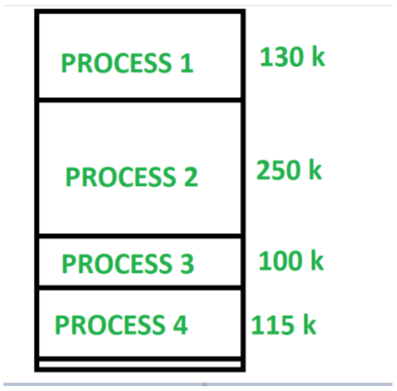
It is another drawback of the fixed-size partition scheme as total unused space by various partitions cannot be used in order to load the processes even though there is the availability of space but it is not in the contiguous fashion.

4. Degree of multiprogramming is less

In this partition scheme, as the size of the partition cannot change according to the size of the process. Thus the degree of multiprogramming is very less and is fixed.

MVT (Multiprogramming with a Variable number of Tasks) is the memory management technique in which each job gets just the amount of memory it needs.

Multi-programming with variable partitioning is a contiguous memory management technique in which the main memory is not divided into partitions and the process is allocated a chunk of free memory that is big enough for it to fit. The space which is left is considered as the free space which can be further used by other processes. It also provides the concept of compaction. In compaction the spaces that are free and the spaces which not allocated to the process are combined and single large memory space is made.



Fragmentation

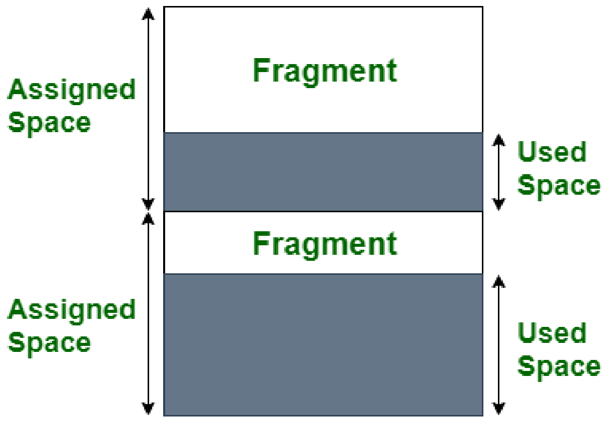
An unwanted problem with operating systems is fragmentation, which occurs when processes load and unload from memory and divide available memory.

This can happen when a file is too large to fit into a single contiguous block of free space on the storage medium, or when the blocks of free space on the medium are insufficient to hold the file.

Types of Fragmentation

There are two main types of fragmentation:

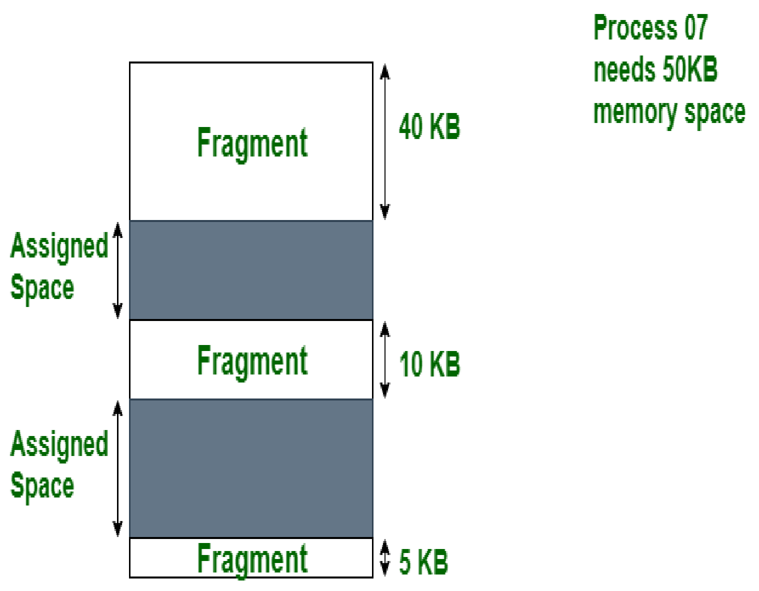
1. Internal fragmentation:Internal fragmentation occurs when there is unused space within a memory block. For example, if a system allocates a 64KB block of memory to store a file that is only 40KB in size, that block will contain 24KB of internal fragmentation. When the system employs a fixed-size block allocation method, such as a memory allocator with a fixed block size, this can occur.



Internal Fragmentation

2**. External fragmentation**

External fragmentation occurs when a storage medium, such as a hard disc or solid-state drive, has many small blocks of free space scattered throughout it. This can happen when a system creates and deletes files frequently, leaving many small blocks of free space on the medium. When a system needs to store a new file, it may be unable to find a single contiguous block of free space large enough to store the file and must instead store the file in multiple smaller blocks. This can cause external fragmentation and performance problems when accessing the file.



Advantages of Variable-size Partition Scheme

- **No Internal Fragmentation** As in this partition scheme space in the main memory is allocated strictly according to the requirement of the process thus there is no chance of internal fragmentation. Also, there will be no unused space left in the partition.
- **Degree of Multiprogramming is Dynamic** As there is no internal fragmentation in this partition scheme due to which there is no unused space in the memory. Thus more processes can be loaded into the memory at the same time.
- **No Limitation on the Size of Process** In this partition scheme as the partition is allocated to the process dynamically thus the size of the process cannot be restricted because the partition size is decided according to the process size.

Disadvantages of Variable-size Partition Scheme

- **External Fragmentation** As there is no internal fragmentation which is an advantage of using this partition scheme does not mean there will no external fragmentation. Let us understand this with the help of an example: In the above diagram- process P1(3MB) and process P3(8MB) completed their execution. Hence there are two spaces left i.e. 3MB and 8MB. Let's there is a Process P4 of size 15 MB comes. But the empty space in memory cannot be allocated as no spanning is allowed in contiguous allocation. Because the rule says that process must be continuously present in the main memory in order to get executed. Thus it results in External Fragmentation.
- **Difficult Implementation** The implementation of this partition scheme is difficult as compared to the Fixed Partitioning scheme as it involves the

allocation of memory at run-time rather than during the system configuration. As we know that OS keeps the track of all the partitions but here allocation and deallocation are done very frequently and partition size will be changed at each time so it will be difficult for the operating system to manage everything.

MEMORY ALLOCATION ALGORITHMS:

1. First Fit

In the first fit approach is to allocate the first free partition or hole large enough which can accommodate the process. It finishes after finding the first suitable free partition.

Advantage

Fastest algorithm because it searches as little as possible.

Disadvantage

The remaining unused memory areas left after allocation become waste if it is too smaller. Thus request for larger memory requirement cannot be accomplished.

2. Best Fit

The best fit deals with allocating the smallest free partition which meets the requirement of the requesting process. This algorithm first searches the entire list of free partitions and considers the smallest hole that is adequate. It then tries to find a hole which is close to actual process size needed.

Advantage

Memory utilization is much better than first fit as it searches the smallest free partition first available.

Disadvantage

It is slower and may even tend to fill up memory with tiny useless holes.

3. Worst fit

In worst fit approach is to locate largest available free portion so that the portion left will be big enough to be useful. It is the reverse of best fit.

Advantage

Reduces the rate of production of small gaps.

Disadvantage

If a process requiring larger memory arrives at a later stage then it cannot be accommodated as the largest hole is already split and occupied.

Solution of External Fragmentation:Compaction

By applying this technique, we can store the bigger processes in the memory. The free partitions are merged which can now be allocated according to the needs of new processes.

NonContiguous Memory Allocation:

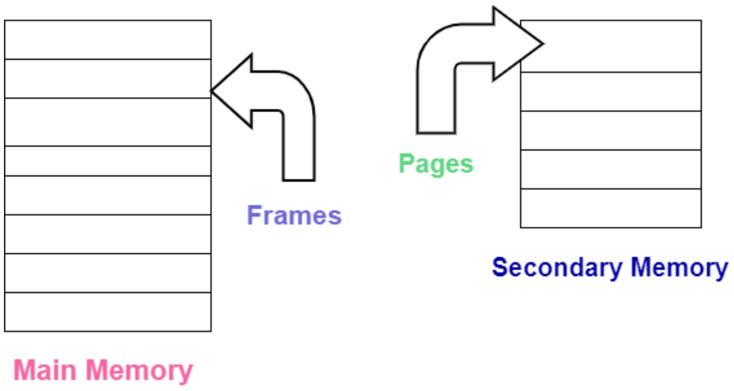
this can be implemented with paging and segmentation.

Paging:

The paging technique divides the physical memory(main memory) into fixed-size blocks that are known as **Frames** and also divide the **logical memory(secondary memory) into blocks of the same size** that are known as **Pages.**

This technique keeps the track of all the free frames.

The Frame has the same size as that of a Page. A frame is basically a place where a (logical) page can be (physically) placed.

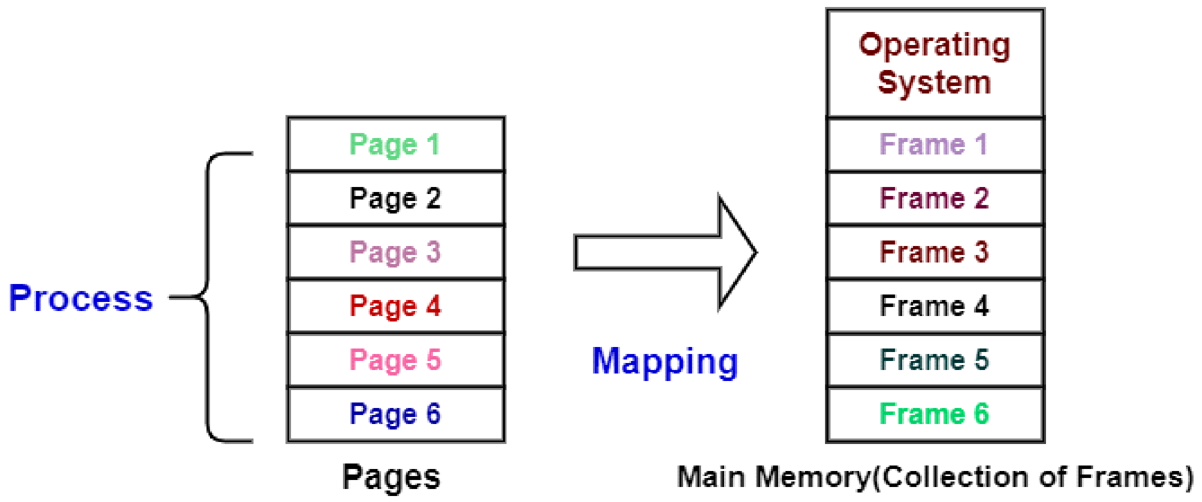


Each process is mainly divided into parts where the size of each part is the same as the page size.

There is a possibility that the size of the last part may be less than the page size.

- Pages of a process are brought into the main memory only when there is a requirement otherwise they reside in the secondary storage.

- One page of a process is mainly stored in one of the frames of the memory. Also, the pages can be stored at different locations of the memory but always the main priority is to find contiguous frames.



Let us now cover the concept of **translating a logical address into the physical address:**

Translation of Logical Address into Physical Address

Before moving on further there are some important points to note:

- The CPU always generates a logical address.
- In order to access the main memory always a physical address is needed.

The **logical address generated by CPU always consists of two parts:**

- Page Number(p)
- Page Offset (d)

where,

Page Number is used to specify the specific page of the process from which the CPU wants to read the data. and it is also used as an index to the page table.

and **Page offset** is mainly used to specify the specific word on the page that the CPU wants to read.

Now let us understand **what is Page Table?**

Page Table in OS

The Page table mainly **contains the base address of each page** in the Physical memory. The base address is then combined with the page offset in order to define the **physical memory address** which is then sent to the memory unit.

Thus page table mainly provides the corresponding frame number (base address of the frame) where that page is stored in the main memory.

As we have told you above that the **frame number** is combined with the **page offset** and forms the **required physical address.**

So, **The physical address consists of two parts:**

- Page offset(d)
- Frame Number(f)

where,

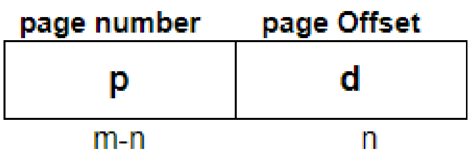
The **Frame number** is used to indicate the specific frame where the required page is stored.

and **Page Offset** indicates the specific word that has to be read from that page.

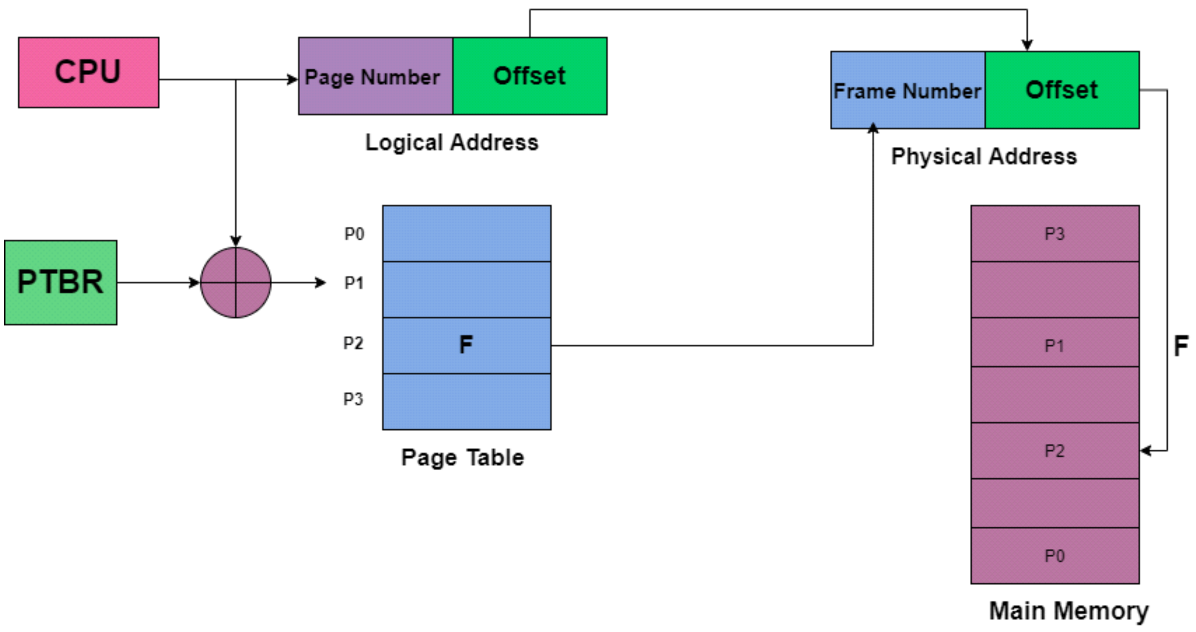
The Page size (like the frame size) is defined with the help of hardware. It is important to note here that the size of the page is typically the power of 2 that varies between 512 bytes and 16 MB per page and it mainly depends on the architecture of the computer.

If the size of **logical address space** is 2 raised to the power m and **page size** is 2 raised to the power n addressing units then the high order m-n bits of **logical address designates the page number** and the n low-order **bits designate the page offset**.

The logical address is as follows:



where **p** indicates the index into the page table, and **d** indicates the displacement within the page.



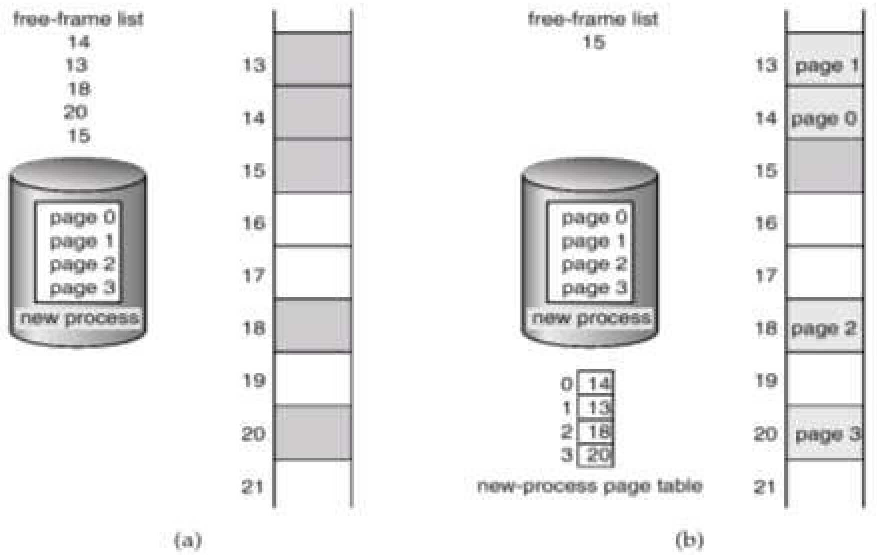
The above diagram indicates the translation of the Logical address into the Physical address. The **PTBR** in the above diagram means page table base register and it basically holds the base address for the page table of the current process.

The **PTBR** is mainly a processor register and is managed by the operating system. Commonly, each process running on a processor needs its own logical address space.

But there is a problem with this approach and that is with the time required to access a user memory location. Suppose if we want to find the location i, we must first find the index into the page table by using the value in the PTBR offset by the page number for I. And this task requires memory access. It then provides us the frame number which is combined with the page offset in order to produce the actual address. After that, we can then access the desired place in the memory.

With the above scheme, two memory accesses are needed in order to access a byte(one for the page-table entry and one for byte). Thus memory access is slower by a factor of 2 and in most cases, this scheme slowed by a factor of 2.

Free frame list



When a page is to be placed in the physical memory, it is necessary to know which frames in the physical memory are free. For this, a free frame list is maintained in a frame table by the operating system. This frame table gives details about which frames are allocated, which frames are available and how many total frames are there. The frame table has one entry for each physical page frame. Each entry gives details on whether the frame is free or allocated, if allocated, to which page of which process. Figure 21.2 shows a free-frame list with frames 14, 13, 18, 20 and 15 as free frames. Frames 16, 17, 19 and 21 are already filled with pages of some other processes. A new process with four pages (page 0, 1, 2, 3) arrives. From the free-frame list, the operating system knows that frame 14 is free and places page 0 in frame 14

Translation of look-aside buffer(TLB)

There is the standard solution for the above problem that is to use a special, small, and fast-lookup hardware cache that is commonly known as Translation of look-aside buffer(TLB).

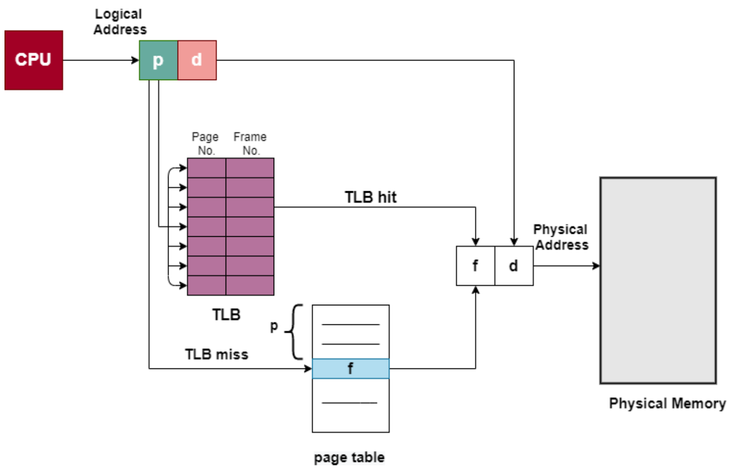
- TLB is associative and high-speed memory.
- Each entry in the TLB mainly consists of two parts: a key(that is the tag) and a value.
- When associative memory is presented with an item, then the item is compared with all keys simultaneously. In case if the item is found then the corresponding value is returned.
- The search with TLB is fast though the hardware is expensive.
- The number of entries in the TLB is small and generally lies in between 64 and 1024.

TLB is used with **Page Tables** in the following ways:

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

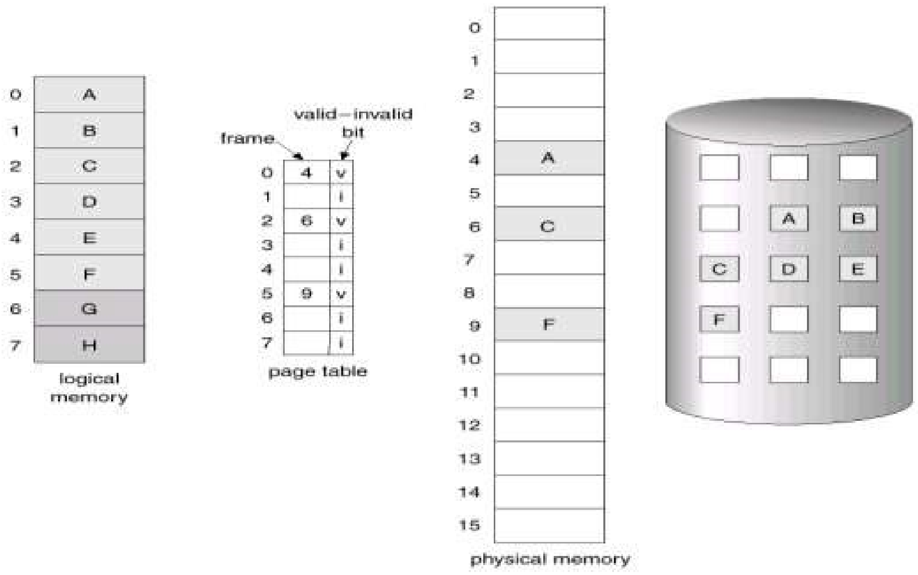
- If the page number is found, then its frame number is immediately available and is used in order to access the memory. The above whole task may take less than 10 percent longer than would if an unmapped memory reference were used.
- In case if the page number is not in the TLB (which is known as **TLB miss**), then a memory reference to the Page table must be made.
- When the frame number is obtained it can be used to access the memory. Additionally, page number and frame number is added to the TLB so that they will be found quickly on the next reference.
- In case if the **TLB is already full of entries** then the Operating system must select o**ne for replacement.**
- TLB allows some entries to **be wired down**, which means they cannot be removed from the TLB. Typically TLB entries for the kernel code are **wired down.**

Paging Hardware With TLB

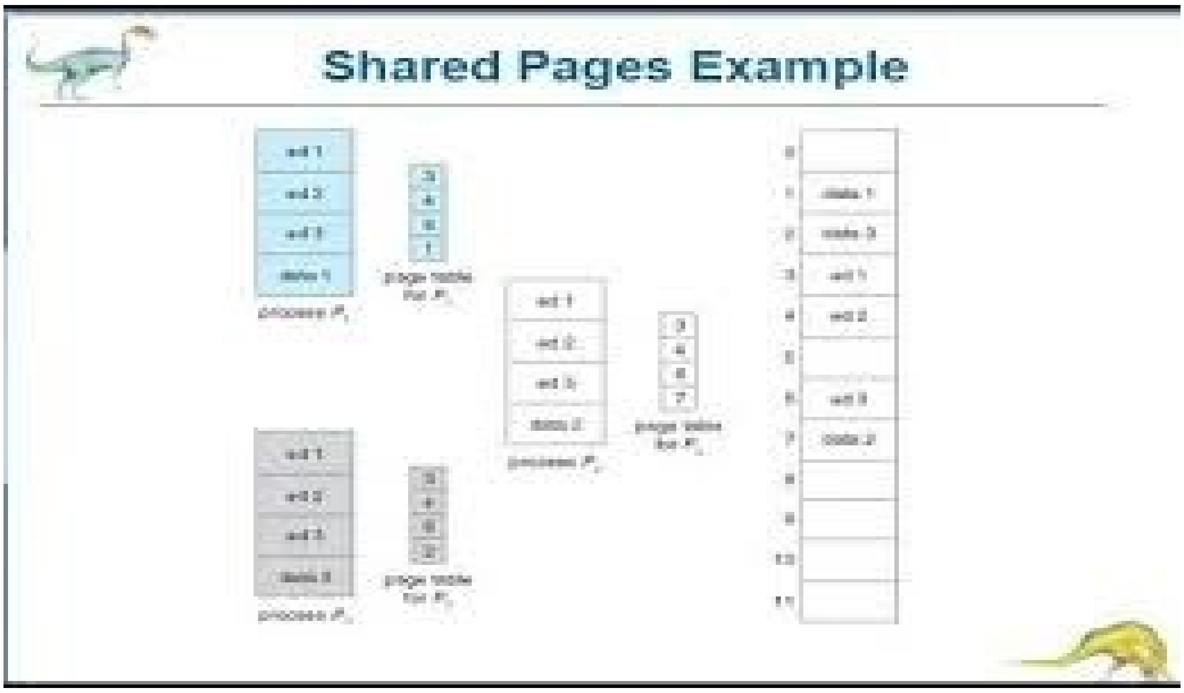


Protection:

Valid/Invalid bit: The paging process should be protected by using the concept of insertion of an additional bit called Valid/Invalid bit. This bit is used to indicate whether a page is currently in memory or not. If the bit is set to valid, the page is in memory, and if it is set to invalid, the page is not in memory.



shared pages:



common pages are shared among processes.

structure of pagetable:

Techniques used for Structuring the Page Table

Some of the common techniques that are used for structuring the Page table are as follows:

- Hierarchical Paging
- Hashed Page Tables
- Inverted Page Tables

Let us cover these techniques one by one;

Hierarchical Paging

Another name for Hierarchical Paging is multilevel paging.

- There might be a case where the page table is too big to fit in a contiguous space, so we may have a hierarchy with several levels.
- In this type of Paging the logical address space is broke up into Multiple page tables.
- Hierarchical Paging is one of the simplest techniques and for this purpose, a

two-level page table and three-level page table can be used.

Two Level Page Table

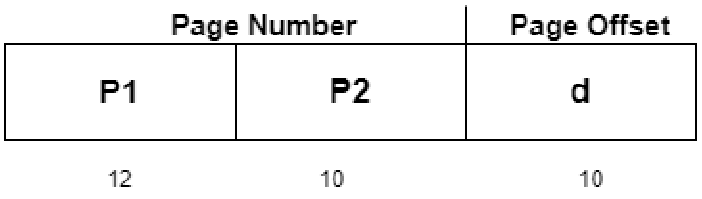
Consider a system having 32-bit logical address space and a page size of 1 KB and it is further divided into:

- Page Number consisting of 22 bits.
- Page Offset consisting of 10 bits.

As we page the Page table, the page number is further divided into :

- Page Number consisting of 12 bits.
- Page Offset consisting of 10 bits.

Thus the Logical address is as follows:



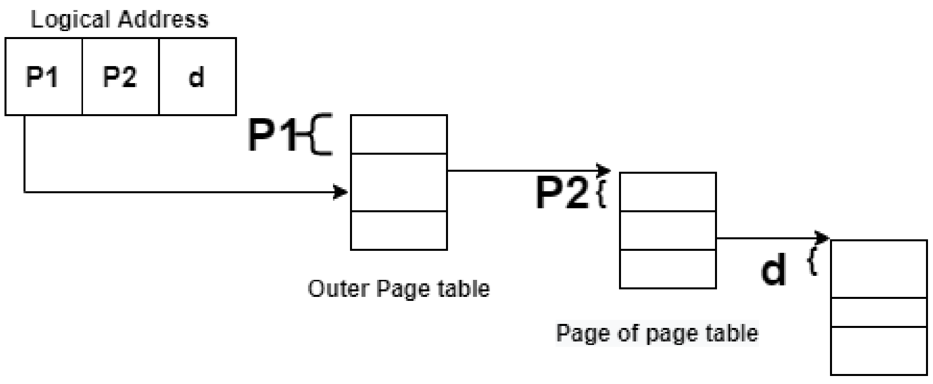
In the above diagram,

P1 is an index into the **Outer Page** table.

P2 indicates the displacement within the page of the **Inner page** Table.

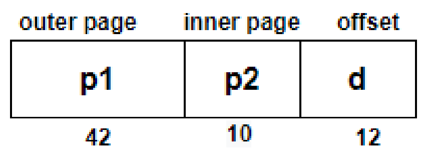
As address translation works from outer page table inward so is known as **forward-mapped Page Table**.

Below given figure below shows the Address Translation scheme for a two-level page table

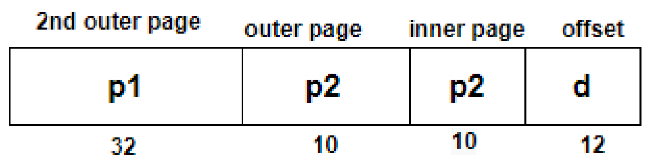


Three Level Page Table

For a system with 64-bit logical address space, a two-level paging scheme is not appropriate. Let us suppose that the page size, in this case, is 4KB.If in this case, we will use the two-page level scheme then the addresses will look like this:



Thus in order to avoid such a large table, there is a solution and that is to divide the outer page table, and then it will result in a **Three-level page table:**



Hashed Page Tables

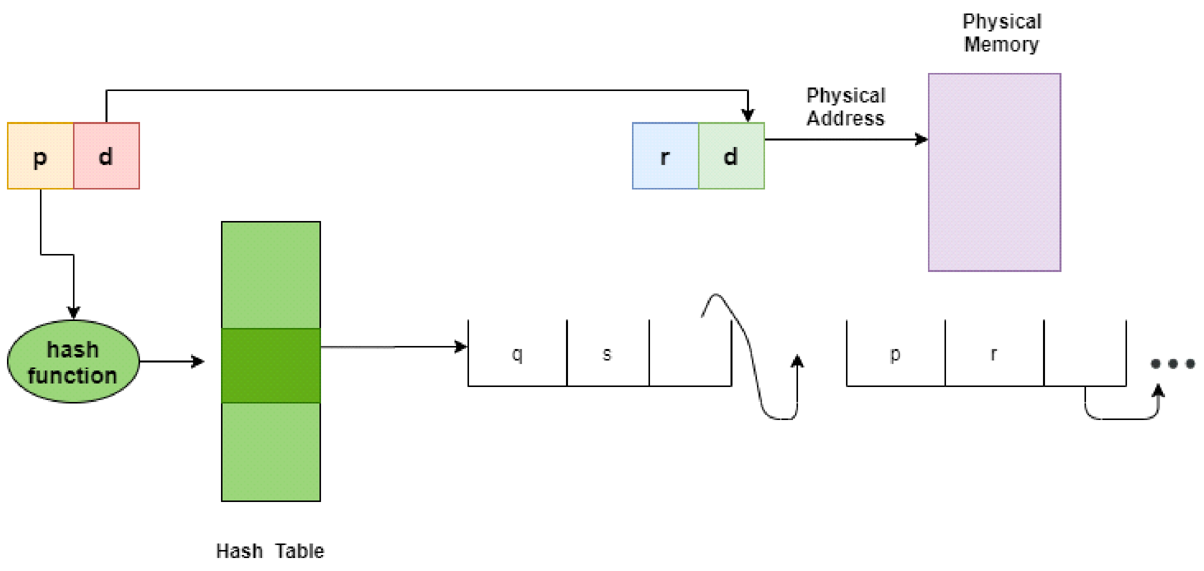
This approach is used to handle address spaces that are larger than 32 bits.

- In this virtual page, the number is hashed into a page table.
- This Page table mainly contains a chain of elements hashing to the same elements.

Each element mainly consists of :

- The virtual page number
- The value of the mapped page frame.
- A pointer to the next element in the linked list.

Given below figure shows the address translation scheme of the Hashed Page Table:



The above Figure shows Hashed Page Table

The Virtual Page numbers are compared in this chain searching for a match; if the match is found then the corresponding physical frame is extracted.

In this scheme, a variation for 64-bit address space commonly uses **clustered page tables**.

Clustered Page Tables

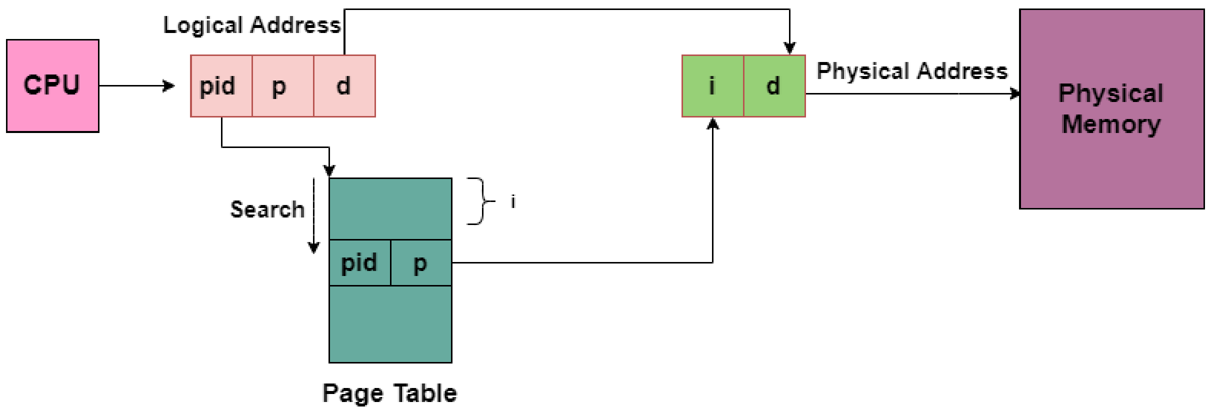
- These are similar to hashed tables but here each entry refers to several pages (that is 16) rather than 1.
- Mainly used for sparse address spaces where memory references are non-contiguous and scattered

Inverted Page Tables

The Inverted Page table basically combines A page table and A frame table into a single data structure.

- There is one entry for each virtual page number and a real page of memory
- And the entry mainly consists of the virtual address of the page stored in that real memory location along with the information about the process that owns the page.
- Though this technique decreases the memory that is needed to store each page table; but it also increases the time that is needed to search the table whenever a page reference occurs.

Given below figure shows the address translation scheme of the Inverted Page Table:



In this, we need to keep the track of process id of each entry, because many processes may have the same logical addresses.

Also, many entries can map into the same index in the page table after going through the hash function. Thus chaining is used in order to handle this

Segmentation in Operating Systems:

Basic Method

A computer system that is using segmentation has a logical address space that can be viewed as multiple segments. And the size of the segment is of the variable that is it may grow or shrink. As we had already told you that during the execution each segment has a name and length. And the address mainly specifies both thing name of the segment and the displacement within the segment.

Therefore the user specifies each address with the help of two quantities: segment name and offset.

For simplified Implementation segments are numbered; thus referred to as segment number rather than segment name.

Thus the logical address consists of two tuples:

<segment-number,offset>

where,

Segment Number(s): Segment Number is used to represent the number of bits that are required to represent the segment.

Offset(d) Segment offset is used to represent the number of bits that are required to represent the size of the segment.

Segmentation Architecture

Segment Table

A Table that is used to store the information of all segments of the process is commonly known as Segment Table. Generally, there is no simple relationship between logical addresses and physical addresses in this scheme.

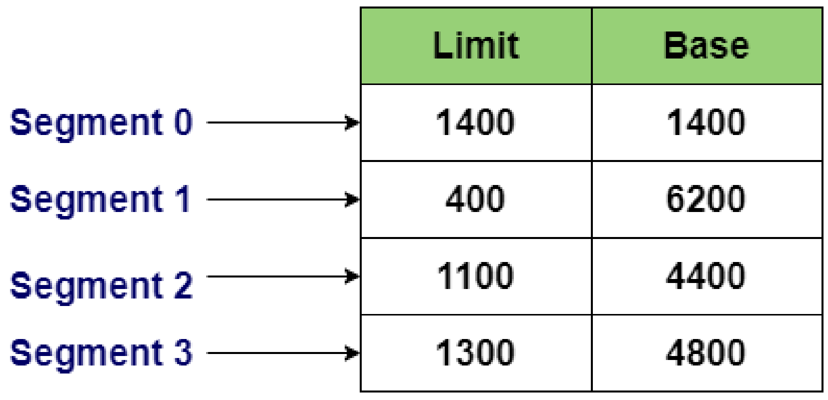
- The mapping of a two-dimensional Logical address into a one-dimensional Physical address is done using the segment table.
- This table is mainly stored as a separate segment in the main memory.
- The table that stores the base address of the segment table is commonly known as the Segment table base register (STBR)

In the segment table each entry has :

- **Segment Base/base address:** The segment base mainly contain**s** the starting physical address where the segments reside in the memory.
- **Segment Limit:** The segment limit is mainly used to specify the length of the segment.

Segment Table Base Register(STBR) The STBR register is used to point the segment table's location in the memory.

Segment Table Length Register(STLR) This register indicates the number of segments used by a program. The segment number s is legal if **s<STLR**

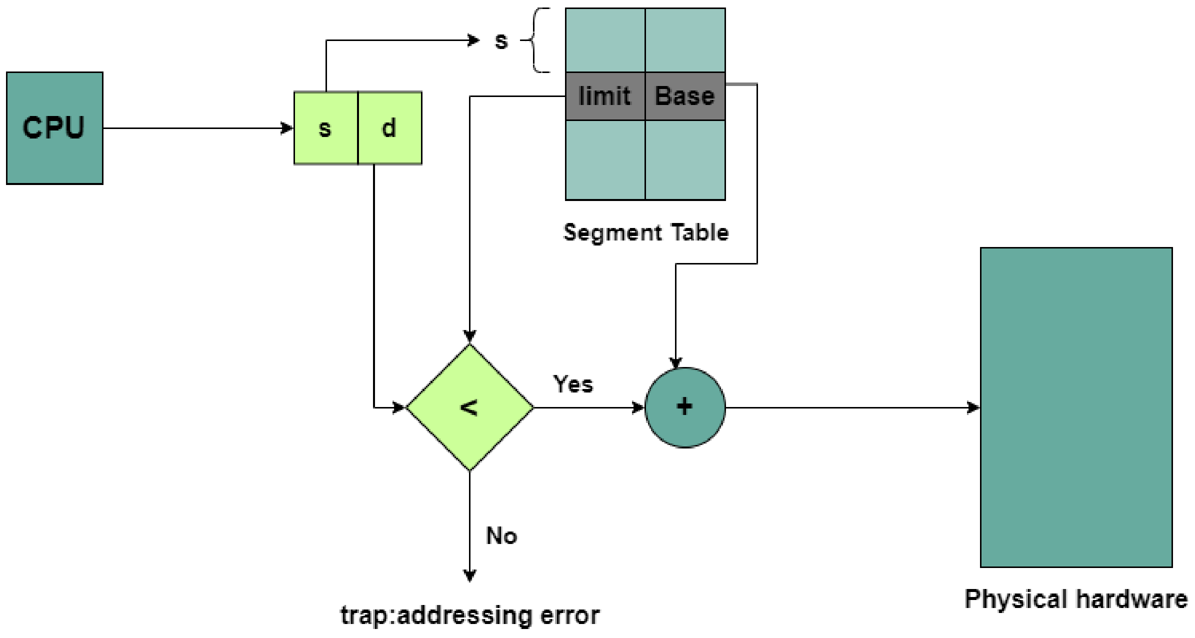


	Limit	Base
Segment 0	1400	1400
Segment 1	400	6200
Segment 2	1100	4400
Segment 3	1300	4800

Segment Table

Segmentation Hardware

Given below figure shows the segmentation hardware :



The logical address generated by CPU consist of two parts:

Segment Number(s): It is used as an index into the segment table.

Offset(d): It must lie in between '0' and 'segment limit'.In this case, if the Offset exceeds the segment limit then the trap is generated.

Thus; **correct offset+segment base= address in Physical memory** and segment table is basically an array of base-limit register pair.

Advantages of Segmentation

The Advantages of the Segmentation technique are as follows:

- In the Segmentation technique, the segment table is mainly used to keep the record of segments. Also, the segment table occupies less space as compared to the paging table.
- There is no Internal Fragmentation.
- Segmentation generally allows us to divide the program into modules that provide better visualization.
- Segments are of variable size.

Disadvantages of Segmentation

Some disadvantages of this technique are as follows:

- Maintaining a segment table for each process leads to overhead
- This technique is expensive.
- The time is taken in order to fetch the instruction increases since now two memory accesses are required.
- Segments are of unequal size in segmentation and thus are not suitable for swapping.
- This technique leads to external fragmentation as the free space gets broken down into smaller pieces along with the processes being loaded and removed from the main memory then this will result in a lot of memory waste.

Paging	**Segmentation**
Paging is a memory management technique where memory is partitioned into fixed-sized blocks that are commonly known as **pages.**	Segmentation is also a memory management technique where memory is partitioned into variable-sized blocks that are commonly known as **segments**.
With the help of Paging, the logical address is divided into	With the help of Segmentation, the logical address is divided into **section**

a **page number** and **page offset**.	**number** and **section offset**.
This technique may lead to **Internal Fragmentation**.	Segmentation may lead to **External Fragmentation**.
In Paging, the page size is decided by the hardware.	While in Segmentation, the size of the segment is decided by the user.
In order to maintain the page data, the page table is created in the Paging	In order to maintain the segment data, the segment table is created in the Paging
The page table mainly contains the base address of each page.	The segment table mainly contains the segment number and the offset.
This technique is faster than segmentation.	On the other hand, segmentation is slower than paging.
In Paging, a list of free frames is maintained by the Operating system.	In Segmentation, a list of holes is maintained by the Operating system.
In this technique, in order to calculate the absolute address page number and the offset both are required.	In this technique, in order to calculate the absolute address segment number and the offset both are required.

Virtual Memory in Operating Systems

Virtual Memory is a space where large programs can store themselves in form of pages while their execution and only the required pages or portions of processes are loaded into the main memory. This technique is useful as a large virtual memory is provided for user programs when a very small physical memory is there. Thus Virtual memory is a technique that allows the execution of processes that are not in the physical memory completely.

Virtual Memory mainly gives the illusion of more physical memory than there really is with the help of Demand Paging.

In real scenarios, most processes never need all their pages at once, for the following reasons :

- Error handling code is not needed unless that specific error occurs, some of which are quite rare.
- Arrays are often over-sized for worst-case scenarios, and only a small fraction of the arrays are actually used in practice.
- Certain features of certain programs are rarely used.

In an Operating system, the memory is usually stored in the form of units that are known as **pages**. Basically, these are atomic units used to store large programs.

Virtual memory can be implemented with the help of:-

- Demand Paging
- Demand Segmentation

The Need for Virtual Memory

Following are the reasons due to which there is a need for Virtual Memory:

- In case, if a computer running the Windows operating system needs more memory or RAM than the memory installed in the system then it uses a small portion of the hard drive for this purpose.
- Suppose there is a situation when your computer does not have space in the physical memory, then it writes things that it needs to remember into the hard disk in a swap file and that as virtual memory.

Benefits of having Virtual Memory

- Large programs can be written, as the virtual space available is huge compared to physical memory.
- Less I/O required leads to faster and easy swapping of processes.
- More physical memory available, as programs are stored on virtual memory, so they occupy very little space on actual physical memory.
- Therefore, the Logical address space can be much larger than that of physical address space.
- Virtual memory allows address spaces to be shared by several processes.
- During the process of creation, virtual memory allows: **copy-on-write** and **Memory-mapped files**

Demand Paging:

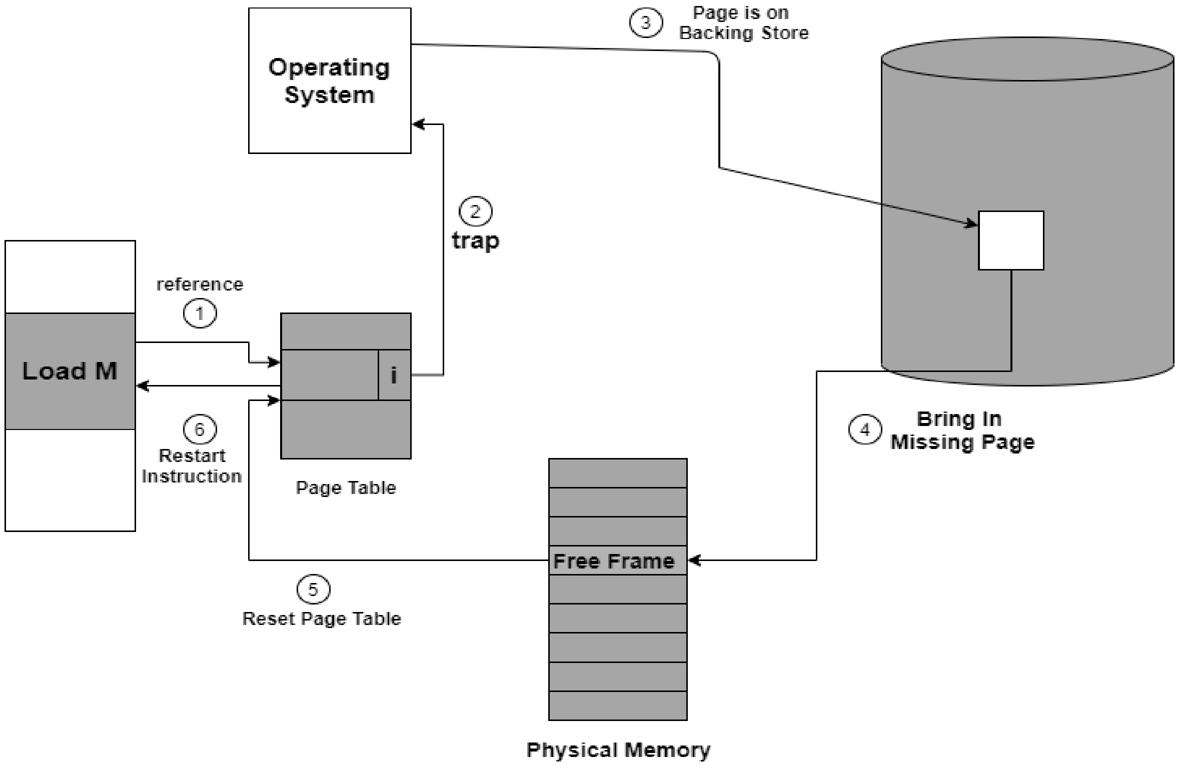
In some cases when initially no pages are loaded into the memory, pages in such cases are only loaded when are demanded by the processor.

In the case of pure demand paging, there is not even a single page that is loaded into the memory initially.

When the execution of the process starts with no pages in the memory, then the operating system sets the instruction pointer to the first instruction of the process.

Page Fault:

- So basically when the page referenced by the CPU is not found in the main memory then the situation is termed as Page Fault.
- Whenever any page fault occurs, then the required page has to be fetched from the secondary memory into the main memory.



- First of all, internal table(that is usually the process control block) for this process in order to determine whether the reference was valid or invalid memory access.
- If the reference is invalid, then we will terminate the process. If the reference is valid, but we have not bought in that page so now we just page it in.
- Then we **locate the free frame lis**t in order to find the free frame.
- Now a disk operation is scheduled in order to read the **desired page into the newly allocated fram**e.
- When the disk is completely read, then the **internal table is modified** that is kept with the process, and the page table that mainly indicates the page is now in memory.
- Now we will restart the instruction that was interrupted due to the trap. Now the process can access the page as though it had always been in memory.

Page Replacement Algorithms:

Basic page replacement algorithms are:

1.First In First Out(FIFO)

2.Optimal page replacement algorithm.

3.Least Recently Used page replacement algorithm(LRU)

1.FCFS

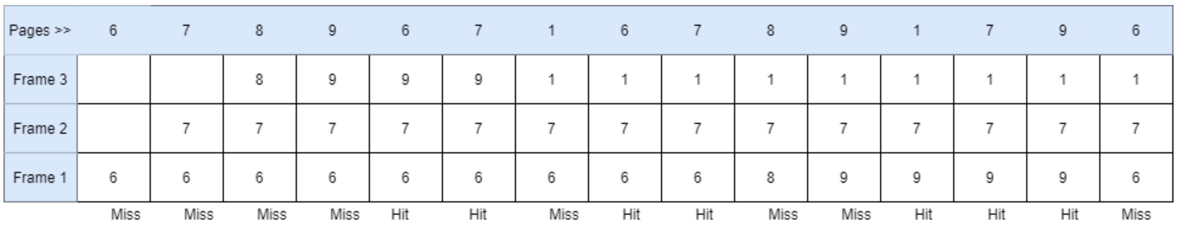
This algorithm mainly replaces the oldest page that has been present in the main memory for the longest time.

Example: Consider the Pages referenced by the CPU in the order are 6, 7, 8, 9, 6, 7, 1, 6, 7, 8, 9, 1

Pages >>	6	7	8	9	6	7	1	6	7	8	9	1
Frame 3			8	8	8	7	7	7	7	7	9	9
Frame 2		7	7	7	6	6	6	6	6	8	8	8
Frame 1	6	6	6	9	9	9	1	1	1	1	1	1
	Miss	Miss	Miss	Miss	Miss	Miss	Miss	Hit	Hit	Miss	Miss	Hit

2.Optimal Page Replacement Algorithm:

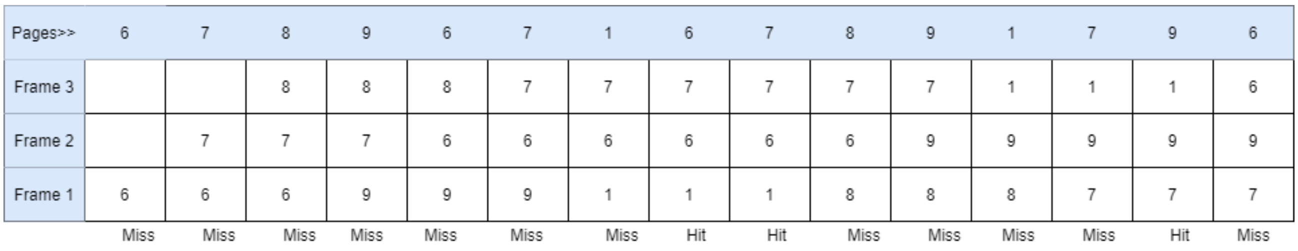
This algorithm mainly replaces the page that will not be used for the longest time in the future.



Pages >>	6	7	8	9	6	7	1	6	7	8	9	1	7	9	6
Frame 3			8	9	9	9	1	1	1	1	1	1	1	1	1
Frame 2		7	7	7	7	7	7	7	7	7	7	7	7	7	7
Frame 1	6	6	6	6	6	6	6	6	6	8	9	9	9	9	6
	Miss	Miss	Miss	Miss	Hit	Hit	Miss	Hit	Hit	Miss	Miss	Hit	Hit	Hit	Miss

3. LRU Page Replacement Algorithm:

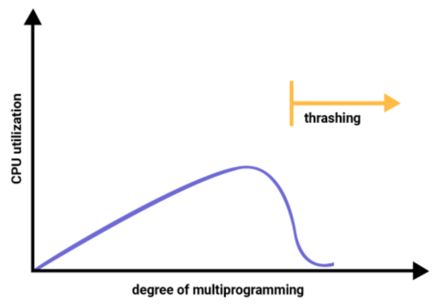
- The page that has not been used for the longest time in the main memory will be selected for replacement.
- Example: Consider the Pages referenced by the CPU in the order are 6, 7, 8, 9, 6, 7, 1, 6, 7, 8, 9, 1, 7, 9, 6
-



Pages>>	6	7	8	9	6	7	1	6	7	8	9	1	7	9	6
Frame 3			8	8	8	7	7	7	7	7	7	1	1	1	6
Frame 2		7	7	7	6	6	6	6	6	6	9	9	9	9	9
Frame 1	6	6	6	9	9	9	1	1	1	8	8	8	7	7	7
	Miss	Miss	Miss	Miss	Miss	Miss	Miss	Hit	Hit	Miss	Miss	Miss	Miss	Hit	Miss

Thrashing:

If the CPU spends less time on some actual productive work spend more time on pagereplacement.



Techniques used to handle the thrashing

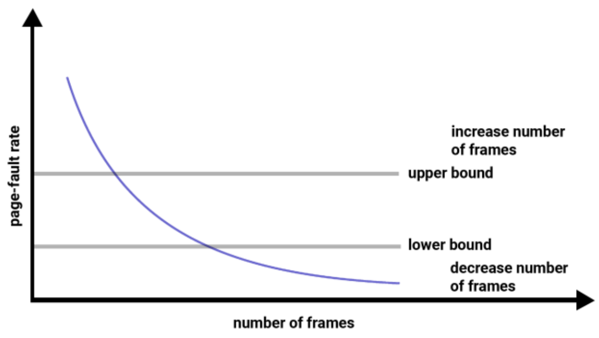
Working-Set Model

Page Fault Frequency

Working-Set Model:

The set of the pages in the most recent? page reference is known as the working set. If a page is in active use, then it will be in the working set. In case if the page is no longer being used then it will drop from the working set.

Page Fault Frequency:



When the Page fault is too high, then we know that the process needs more frames. Conversely, if the page fault-rate is too low then the process may have too many frames.