ALLOCATING KERNEL MEMORY

When a process running in user mode requests additional memory, pages are allocated from the list of free page frames maintained by the kernel. This list is typically populated using a page-replacement algorithm and most likely contains free pages scattered throughout physical memory. If a user process requests a single byte of memory, internal fragmentation will result, as the process will be granted an entire page frame.

Kernel memory is often allocated from a free-memory pool different from the list used to satisfy ordinary user-mode processes.

There are two primary reasons for this:

1. The kernel requests memory for data structures of varying sizes, some of which are less than a page in size. As a result, the kernel must use memory conservatively and attempt to minimize waste due to fragmentation. This is especially important because many operating systems do not subject kernel code or data to the paging system.

2. Pages allocated to user-mode processes do not necessarily have to be in contiguous physical memory. However, certain hardware devices interact directly with physical memory —without the benefit of a virtual memory interface —and consequently may require memory residing in physically contiguous pages.

In the following sections, we examine two strategies for managing free memory that is assigned to kernel processes: the “buddy system” and slab allocation.

1. Buddy System

The buddy system allocates memory from a fixed-size segment consisting of physically contiguous pages. Memory is allocated from this segment using a **power-of-2 allocator**, which satisfies requests in units sized as a power of 2 (4 KB, 8 KB, 16 KB, and so forth). A request in units not appropriately sized is rounded up to the next highest power of 2. For example, a request for 11 KB is satisfied with a 16-KB segment.

Let’s consider a simple example. Assume the size of a memory segment is initially 256 KB and the kernel requests 21 KB of memory. The segment is initially divided into

two **buddies**—which we will call *AL* and *AR*—each 128 KB in size. One of these buddies is further divided into two 64-KB buddies —
BL and *BR*. However, the next-highest power of 2 from 21 KB is 32 KB so either *BL* or *BR* is again divided into two 32-KB buddies, *CL* and *CR*. One of these buddies is used to satisfy the 21-KB request. This scheme is illustrated in Figure 9.26, where *CL* is the segment allocated to the 21-KB request.

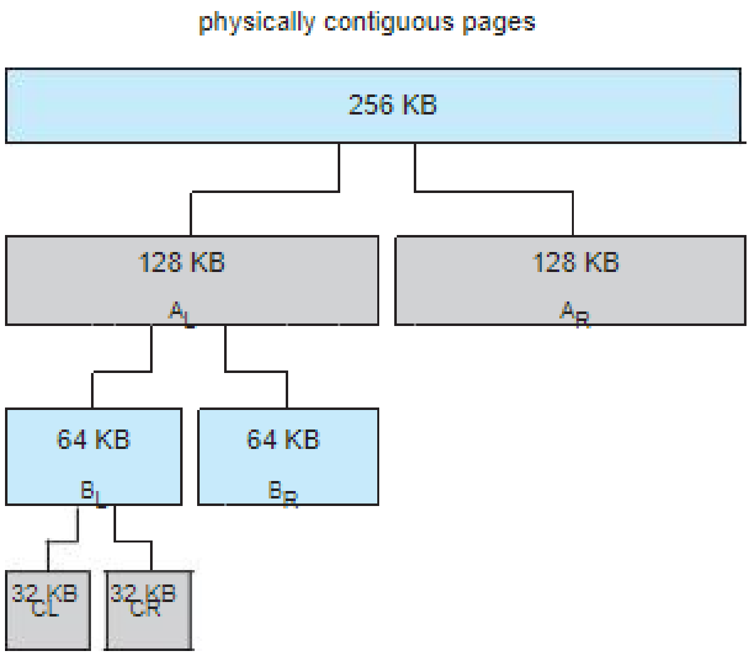


Fig : Buddy System Allocation

An advantage of the buddy system is how quickly adjacent buddies can be combined to form larger segments using a technique known as coalescing. In the above Figure for example, when the kernel releases the *CL* unit it was allocated, the system can coalesce *CL* and *CR* into a 64-KB segment.

This segment, *BL* , can in turn be coalesced with its buddy *BR* to form a 128-KB segment. Ultimately, we can end up with the original 256-KB segment.

The obvious drawback to the buddy system is that rounding up to the next highest power of 2 is very likely to cause fragmentation within allocated segments. For example, a 33-KB request can only be satisfied with a 64- KB segment.

In fact, we cannot guarantee that less than 50 percent of the allocated unit will be wasted due to internal fragmentation. In the following section, we explore a memory allocation scheme where no space is lost due to fragmentation.

2. Slab Allocation

A second strategy for allocating kernel memory is known as slab allocation. A slab is made up of one or more physically contiguous pages. A cache consists of one or more slabs. There is a single cache for each unique kernel data structure —for example, a separate cache for the data structure representing process descriptors, a separate cache for file objects, a separate cache for semaphores, and so forth.

Each cache is populated with objects that are instantiations of the kernel data structure the cache represents.

For example, the cache representing semaphores stores instances of semaphore objects, the cache representing process descriptors stores instances of process descriptor objects, and so forth. The relationship among slabs, caches, and objects is shown in Figure

The figure shows two kernel objects 3 KB in size and three objects 7 KB in size, each stored in a separate cache.

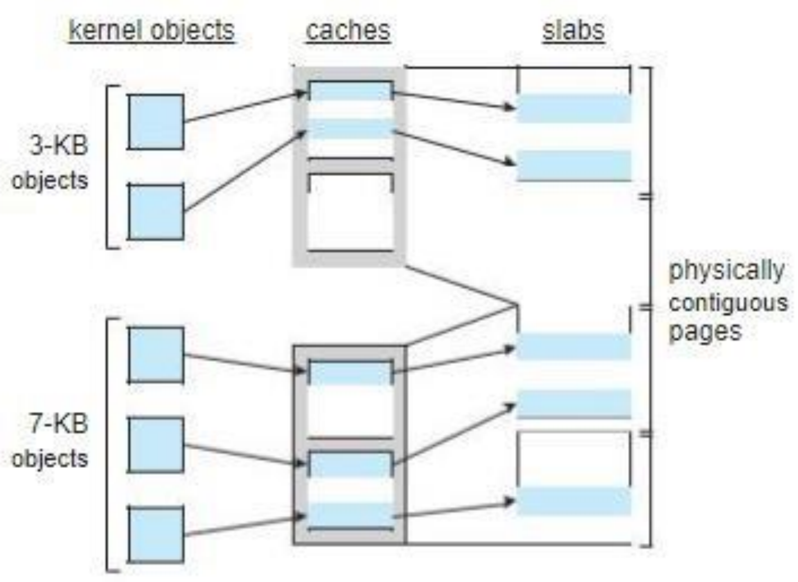


Fig : Slab Alocation

The slab-allocation algorithm uses caches to store kernel objects. When a cache is created, a number of objects which are initially marked as free are allocated to the cache. The number of objects in the cache depends on the size of the associated slab.

For example, a 12-KB slab (made up of three continguous 4-KB pages) could store six 2-KB objects. Initially, all objects in the cache are marked as free. When a new object for a kernel data structure is needed, the allocator can assign any free object from the cache to satisfy the request. The object assigned from the cache is marked as used.

Let's consider a scenario in which the kernel requests memory from the slab allocator for an object representing a process descriptor. In Linux systems, a process descriptor is of the type struct task struct, which requires approximately 1.7 KB of memory.

When the Linux kernel creates a new task, it requests the necessary memory for the struct task struct object from its cache. The cache will fulfill the request using a struct task struct object that has already been allocated in a slab and is marked as free.

In Linux, a slab may be in one of three possible states:

1. Full. All objects in the slab are marked as used.

2. Empty. All objects in the slab are marked as free.

3. Partial. The slab consists of both used and free objects.

The slab allocator first attempts to satisfy the request with a free object in a partial slab. If none exists, a free object is assigned from an empty slab. If no empty slabs are available, a new slab is allocated from contiguous physical pages and assigned to a cache; memory for the object is allocated from this slab.

The slab allocator provides two main benefits:

1. No memory is wasted due to fragmentation. Fragmentation is not an issue because each unique kernel data structure has an associated cache, and each cache is made up of one or more slabs that are divided into chunks the size of the objects being represented.

Thus, when the kernel requests memory for an object, the slab allocator returns the exact amount of memory required to represent the object.

2. Memory requests can be satisfied quickly. The slab allocation scheme is thus particularly effective for managing memory when objects are frequently allocated and deallocated, as is often the case with requests from the kernel. The act of allocating —and releasing — memory can be a time-consuming process.

However, objects are created in advance and thus can be quickly allocated from the cache. Furthermore, when the kernel has finished with an object and releases it, it is marked as free and returned to its cache, thus making it immediately available for subsequent requests from the kernel.

The slab allocator first appeared in the Solaris 2.4 kernel. Because of its general-purpose nature, this allocator is now also used for certain user-mode memory requests in Solaris. Linux originally used the buddy system; however, beginning with Version 2.2, the Linux kernel adopted the slab allocator.

Recent distributions of Linux now include two other kernel memory allocators — the SLOB and SLUB allocators. (Linux refers to its slab implementation as SLAB.)

The SLOB allocator is designed for systems with a limited amount of memory, such as embedded systems. SLOB (which stands for Simple List of Blocks) works by maintaining three lists of objects: ***small*** (for objects less than 256 bytes), ***medium*** (for objects less than 1,024 bytes), and ***large*** (for objects less than 1,024 bytes). Memory requests are allocated from an object on an appropriately sized list using a first-fit policy.

Beginning with Version 2.6.24, the SLUB allocator replaced SLAB as the default allocator for the Linux kernel. SLUB addresses performance issues with slab allocation by reducing much of the overhead required by the SLAB allocator.

One change is to move the metadata that is stored with each slab under SLAB allocation to the page structure the Linux kernel uses for each page. Additionally, SLUB removes the per-CPU queues that the SLAB allocator maintains for objects in each cache.

For systems with a large number of processors, the amount of memory allocated to these queues was not insignificant. Thus, SLUB provides better performance as the number of processors on a system increases.

UNIT III

MEMORY MANAGEMENT: BACKGROUND

In general, to run a program, it must be brought into memory.

Input queue – collection of processes on the disk that are waiting to be brought into memory to run the program.

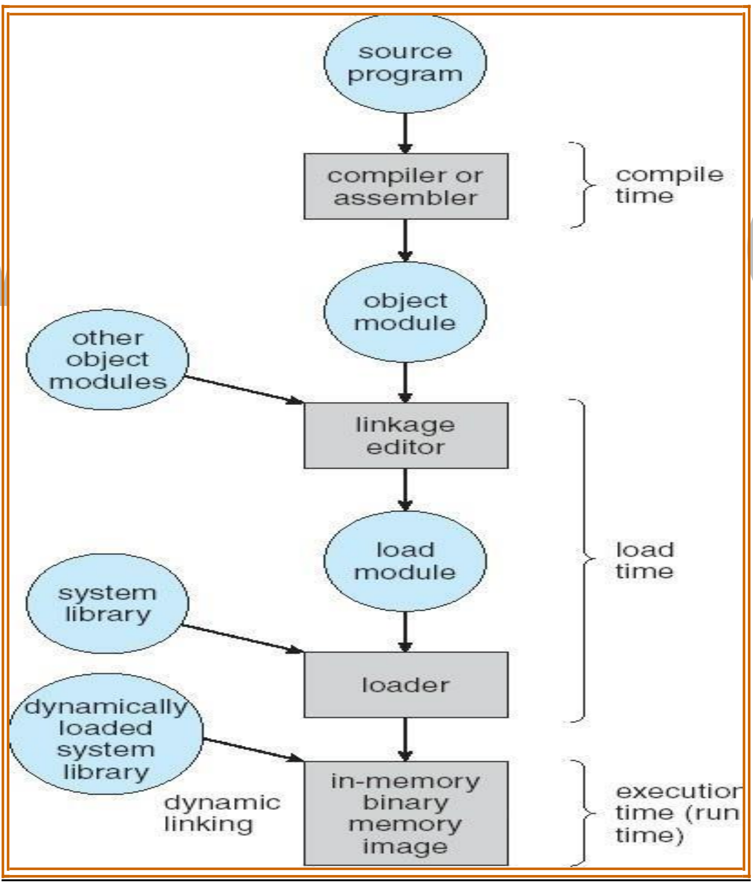
User programs go through several steps before being run

Address binding: Mapping of instructions and data from one address to another address in memory.

Three different stages of binding:

1. **Compile time**: Must generate absolute code if memory location is known in prior.
2. **Load time**: Must generate relocatable code if memory location is not known at compile time
3. **Execution time**: Need hardware support for address maps (e.g., base and limit registers).

Multistep Processing of a User Program



Logical vs. Physical Address Space

- **Logical address** – generated by the CPU; also referred to as **"virtual address"**
- **Physical address** – address seen by the memory unit.
- 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

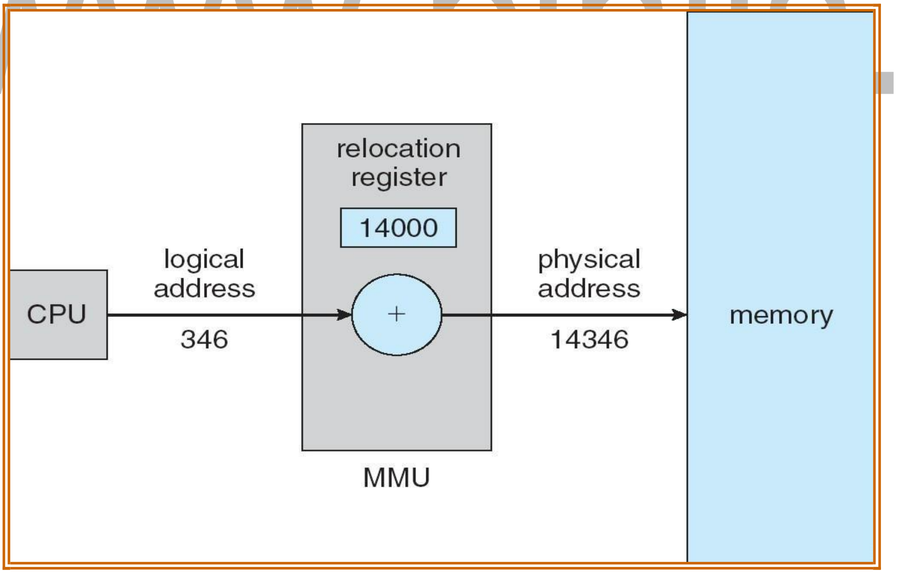
Memory-Management Unit (MMU)

It is a hardware device that maps virtual / Logical address to physical address

- In this scheme, the relocation register's value is added to Logical address generated by a user process.
- The user program deals with *logical* addresses; it never sees the *real* physical addresses
- Logical address range: 0 to max
- Physical address range: R+0 to R+max, where R—value in relocation register

Note: relocation register is a base register.

Dynamic relocation using relocation register



Dynamic Loading

- Through this, the routine is not loaded until it is called.

- Better memory-space utilization; unused routine is never loaded
- Useful when large amounts of code are needed to handle infrequently occurring cases
- No special support from the operating system is required implemented through programdesign

Dynamic Linking

- Linking postponed until execution time & is particularly useful for libraries
- Small piece of code called stub, used to locate the appropriate memory-resident library routine or function.
- 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

Shared libraries: Programs linked before the new library was installed will continue using the older library

Overlays:

- Enable a process larger than the amount of memory allocated to it.
- At a given time, the needed instructions & data are to be kept within a memory.

Swapping

A process can be swapped temporarily out of memory to a backing store (SWAP OUT) and then brought back into memory for continued execution (SWAP IN).

Backing store

Fast disk large enough to accommodate copies of all memory images for all users & it must provide direct access to these memory images

Roll out, roll in

Swapping variant used for priority-based scheduling algorithms; lower-priority process is swapped out so higher-priority process can be loaded and executed

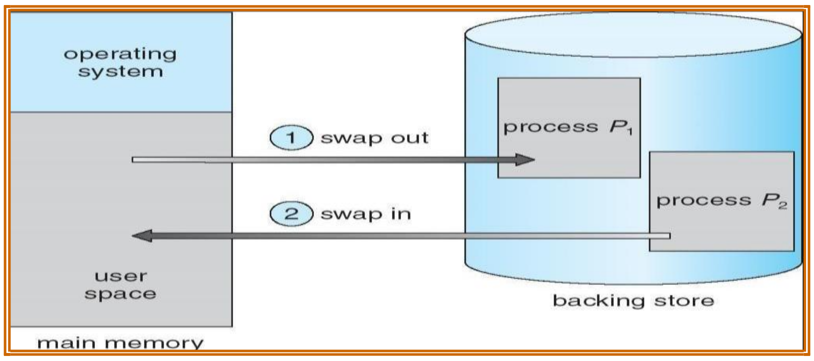
Transfer time :

- Major part of swap time is transfer time
- Total transfer time is directly proportional to the amount of memory swapped.

Example: Let us assume the user process is of size 1MB & the backing store is a standard hard disk with a transfer rate of 5MBPS.

Transfer time = 1000KB/5000KB per second

= 1/5 sec = 200ms



(i) Memory Protection:

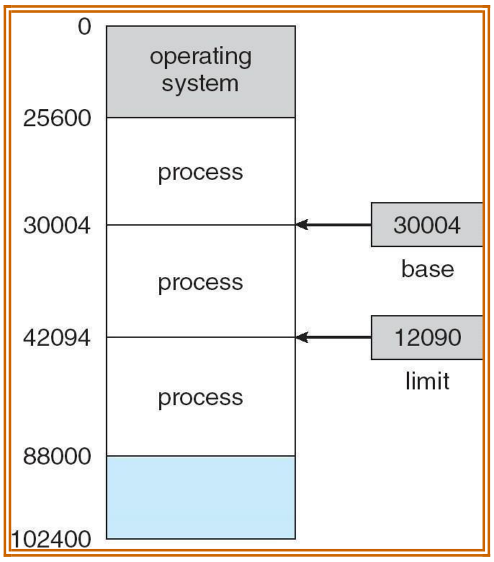
Memory protection is needed for

a) Protecting the OS from user process.

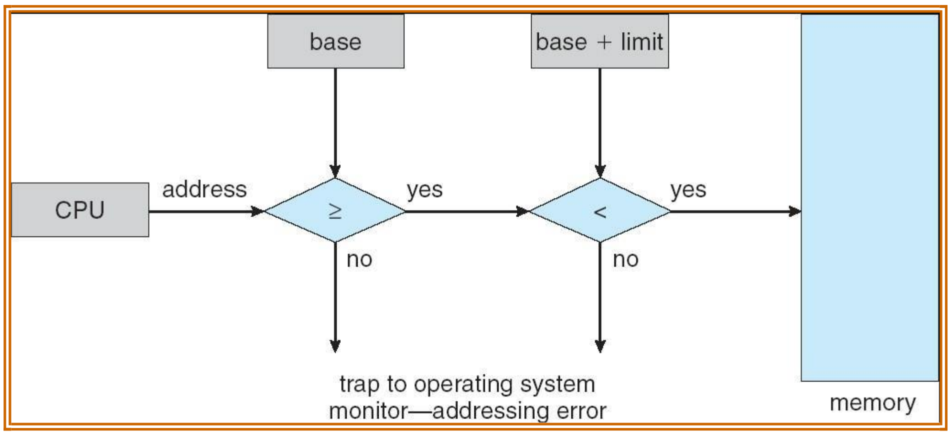
b) Protecting user processes from one another.

- The above protection is done by **"Relocation-register & Limit-register scheme** —
- Relocation register contains value of smallest physical address i.e base value.
- Limit register contains range of logical addresses – each logical address must be less than the limit register

A base and a limit register define a logical address space



HW address protection with base and limit registers



Contiguous Allocation

Each process is contained in a single contiguous section of memory.

There are two methods namely :

- Fixed – Partition Method
- Variable – Partition Method

Fixed - Partition Method :

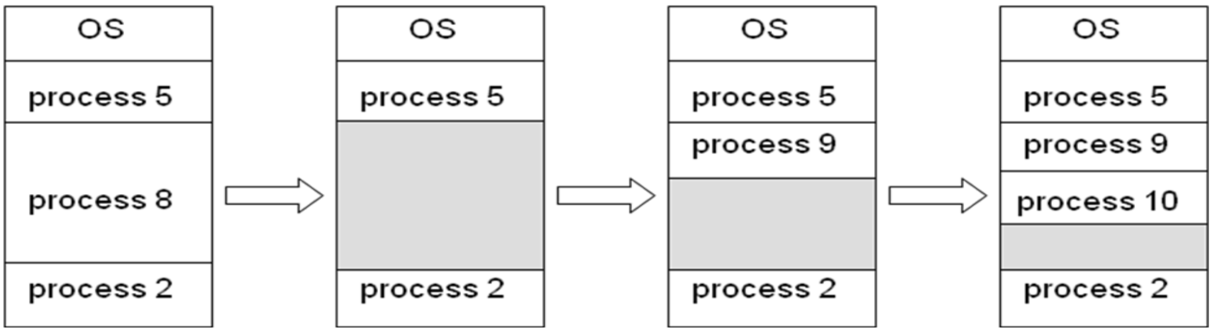
- Divide memory into fixed size partitions, where each partition has exactly one process.
- The drawback is memory space unused within a partition is wasted.

(eg.when process size < partition size)

Variable-partitionmethod:

- Divide memory into variable size partitions, depending upon the size of the incoming process.
- When a process terminates, the partition becomes available for another process.
- As processes complete and leave they create holes in the main memory.

Hole – block of available memory; holes of various size are scattered throughout memory.



Dynamic Storage-Allocation Problem:

How to satisfy a request of size ‗n‘ from a list of free holes?

Solution:

First-fit: Allocate the *first* hole that is big enough.

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

Worst-fit: Allocate the *largest* hole; must also search entire list. Produces the largest leftover hole.

NOTE: First-fit and best-fit are better than worst-fit in terms of speed and storage utilization

Fragmentation:

External Fragmentation

– This takes place when enough total memory space exists to satisfy a request, but it is not contiguous i.e, storage is fragmented into a large number of small holes scattered throughout the main memory.

Internal Fragmentation

– Allocated memory may be slightly larger than requested memory.

Example: hole = 184 bytes

Process size = 182 bytes. We are left with a hole of 2 bytes.

Solutions:

1. **Coalescing :** Merge the adjacent holes together.
2. **Compaction:** Move all processes towards one end of memory, hole towards other end of memory, producing one large hole of available memory. This scheme is expensive as it can be done if relocation is dynamic and done at execution time.
3. Permit the logical address space of a process to be **non-contiguous**.

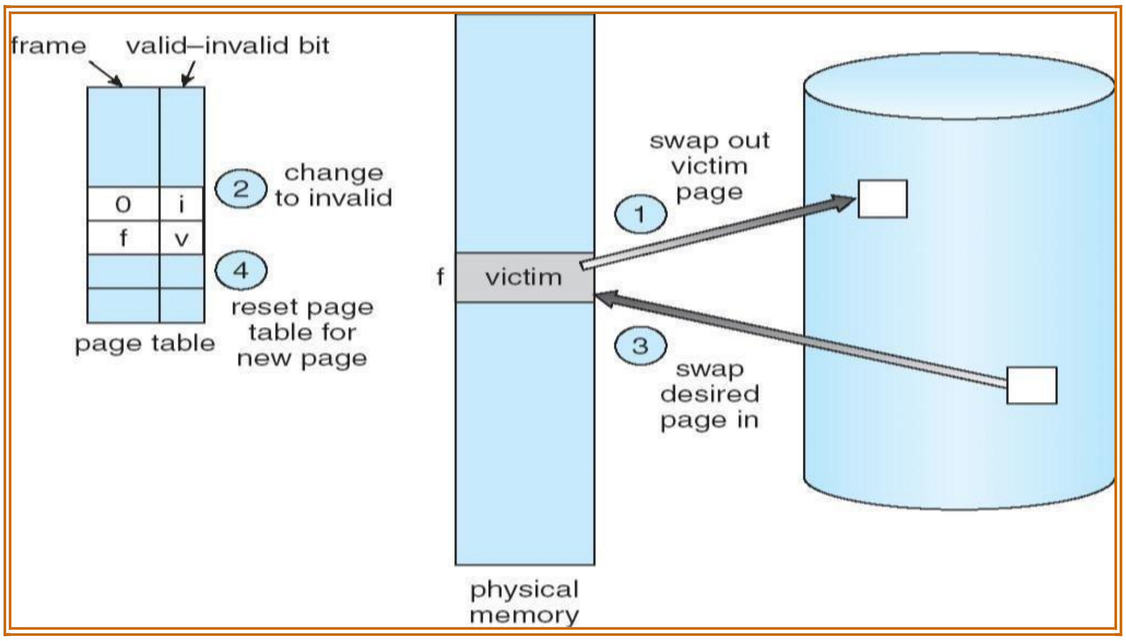
This is achieved through two memory management schemes namely **paging** and **segmentation**.

PAGE REPLACEMENT

- If no frames are free, we could find one that is not currently being used & free it.
- We can free a frame by writing its contents to swap space & changing the page table to indicate that the page is no longer in memory.
- Then we can use that freed frame to hold the page for which the process faulted.

Basic Page Replacement

1. Find the location of the desired page on disk
2. Find a free frame
   - If there is a free frame , then use it.
   - If there is no free frame, use a page replacement algorithm to select a **victim** frame
   - Write the victim page to the disk, change the page & frame tables accordingly.
3. Read the desired page into the (new) free frame. Update the page and frame tables.
4. Restart the process



Note:

If no frames are free, two page transfers are required & this situation effectively doubles the page- fault service time.

Modify (dirty) bit:

It indicates that any word or byte in the page is modified.

When we select a page for replacement, we examine its modify bit.

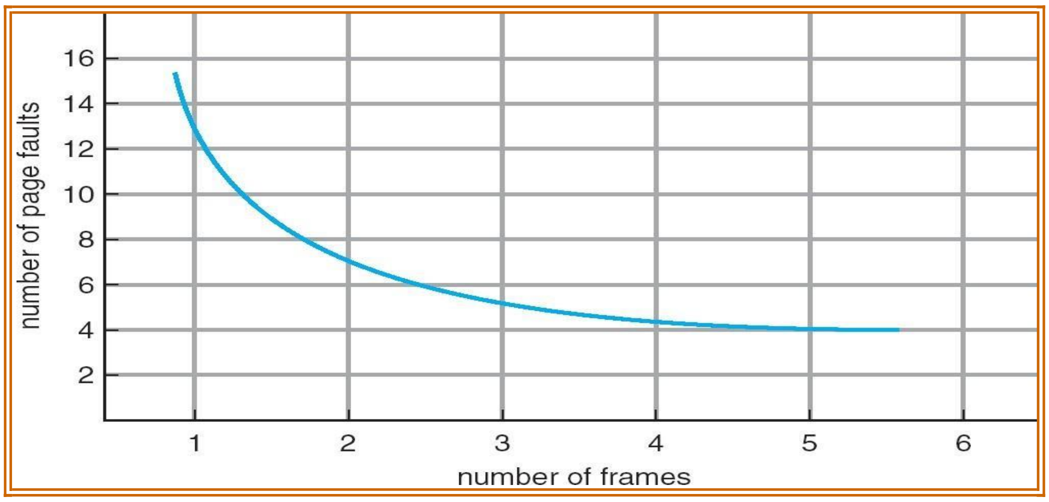
- If the bit is set, we know that the page has been modified & in this case we must write that page to the disk.
- If the bit is not set, then if the copy of the page on the disk has not been overwritten, then we can avoid writing the memory page on the disk as it is already there.

Page Replacement Algorithms

1. FIFO Page Replacement
2. Optimal Page Replacement
3. LRU Page Replacement
4. LRU Approximation Page Replacement
5. Counting-Based Page Replacement

We evaluate an algorithm by running it on a particular string of memory references & computing the number of page faults. The string of memory reference is called a "reference string" The algorithm that provides less number of page faults is termed to be a good one.

As the number of available frames increases , the number of page faults decreases. This is shown in the following graph:



(a) FIFO page replacement algorithm

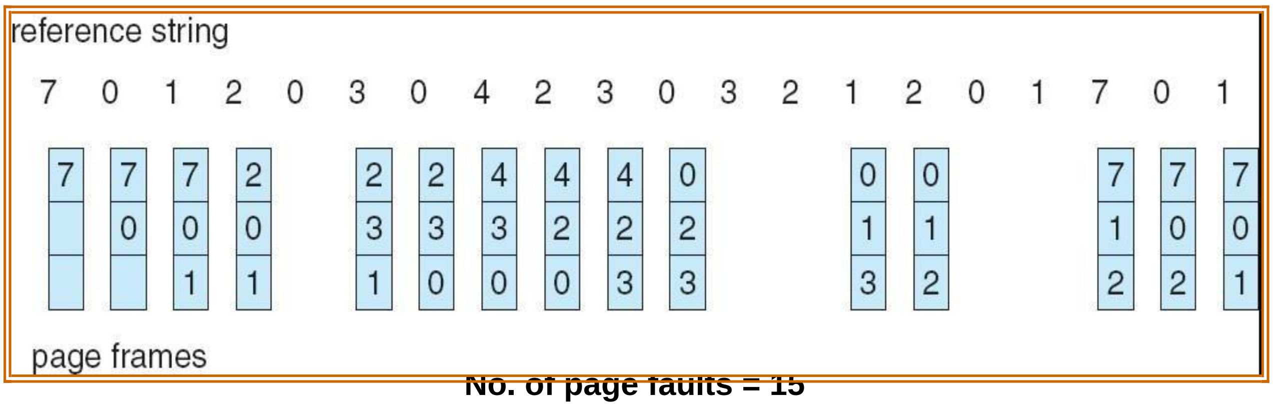
Replace the oldest page.

This algorithm associates with each page , the time when that page was brought in.

Example:

Reference string: 7,0,1,2,0,3,0,4,2,3,0,3,2,1,2,0,1,7,0,1

No.of available frames = 3 (3 pages can be in memory at a time per process)



No. of page faults = 15

Drawback:

FIFO page replacement algorithm's performance is not always good.

To illustrate this, consider the following example:

Reference string: 1, 2, 3, 4, 1, 2, 5, 1, 2, 3, 4, 5

- If No.of available frames -= 3 then the no.of page faults =9
- If No.of available frames =4 then the no.of page faults =10
- Here the no. of page faults increases when the no.of frames increases .This is called as **Belady's Anomaly.**

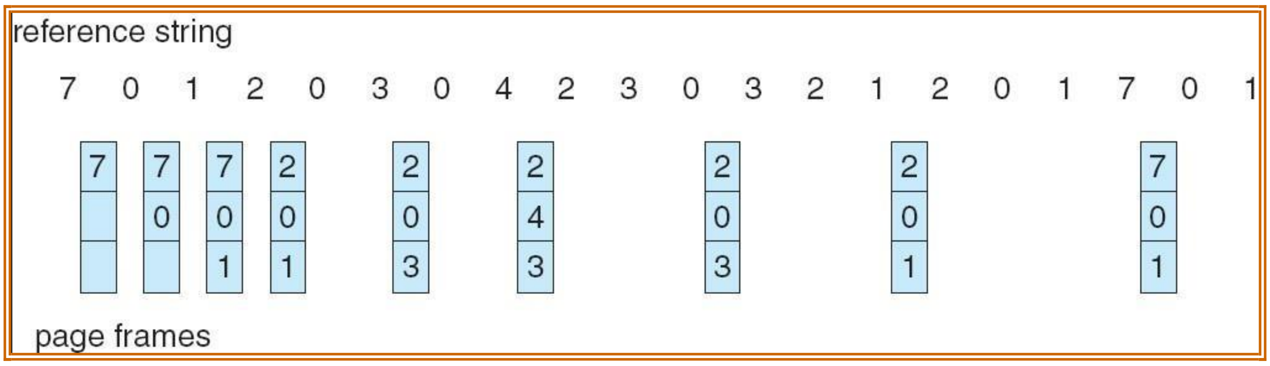
(b) Optimal page replacement algorithm

Replace the page that will not be used for the longest period of time.

Example:

Reference string: 7,0,1,2,0,3,0,4,2,3,0,3,2,1,2,0,1,7,0,1

No.of available frames = 3



No. of page faults = 9

Drawback:

It is difficult to implement as it requires future knowledge of the reference string.

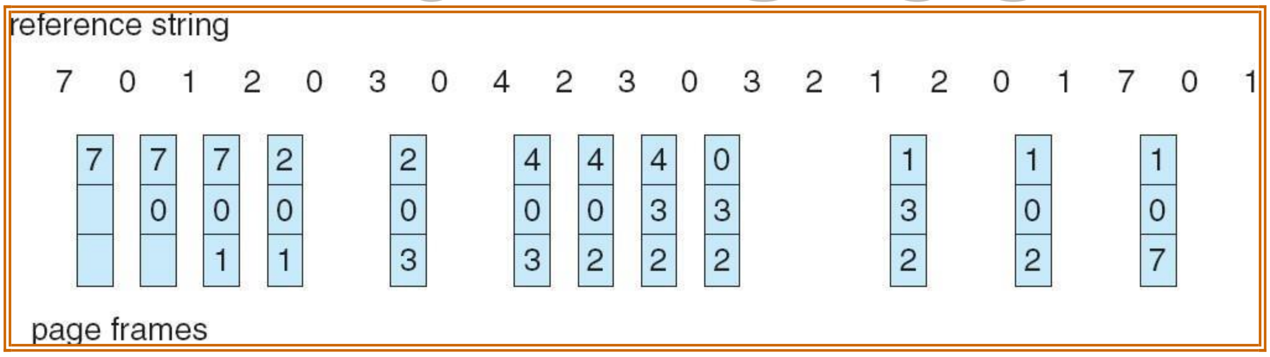
(c) LRU(Least Recently Used) page replacement algorithm

Replace the page that has not been used for the longest period of time.

Example:

Reference string:7,0,1,2,0,3,0,4,2,3,0,3,2,1,2,0,1,7,0,1

No.of available frames = 3



No. of page faults = 12

LRU page replacement can be implemented using

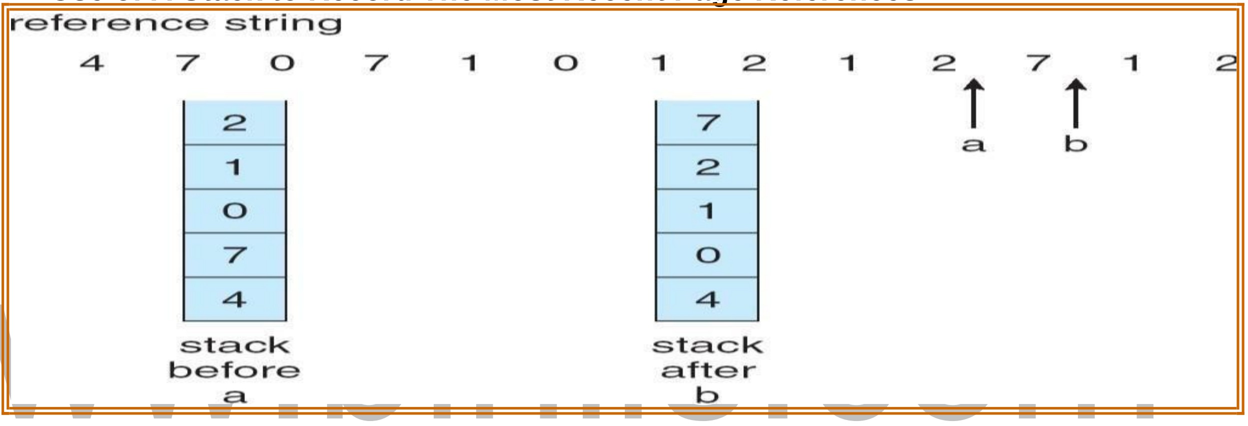
1. **Counters**
   - Every page table entry has a time-of-use field and a clock or counter is associated with the CPU.
   - The counter or clock is incremented for every memory reference.
   - Each time a page is referenced , copy the counter into the time-of-use field.

- When a page needs to be replaced, replace the page with the smallest counter value.

2. **Stack**

  - Keep a stack of page numbers
  - Whenever a page is referenced, remove the page from the stack and put it on top of the stack.
  - When a page needs to be replaced, replace the page that is at the bottom of the stack.(LRU page)

Use of A Stack to Record The Most Recent Page References



(d) LRU Approximation Page Replacement

Reference bit

- With each page associate a reference bit, initially set to 0
- When page is referenced, the bit is set to 1

- When a page needs to be replaced, replace the page whose reference bit is 0
- The order of use is not known , but we know which pages were used and which were not used.

(i) Additional Reference Bits Algorithm

- Keep an 8-bit byte for each page in a table in memory.
- At regular intervals , a timer interrupt transfers control to OS.
- The OS shifts reference bit for each page into higher- order bit shifting the other bits right 1 bit and discarding the lower-order bit.

Example:

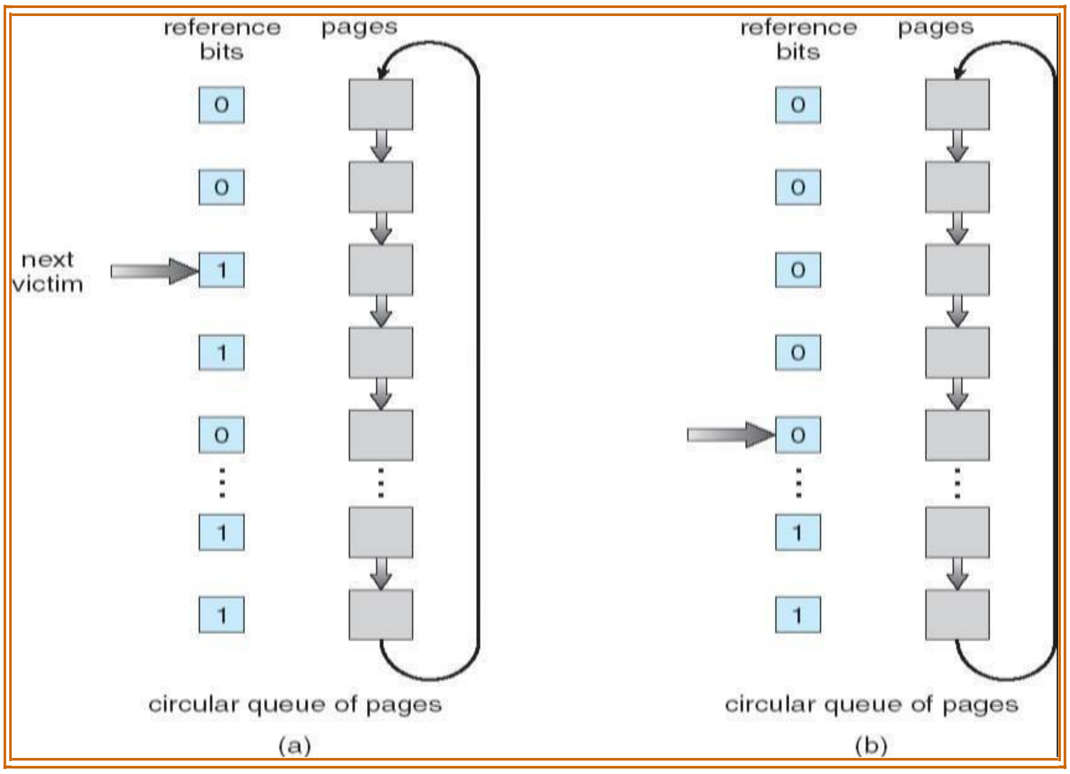
- If reference bit is 00000000 then the page has not been used for 8 time periods.
- If reference bit is 11111111 then the page has been used atleast once each time period.
- If the reference bit of page 1 is 11000100 and page 2 is 01110111 then page 2 is the LRU page.

(ii) Second Chance Algorithm

Basic algorithm is FIFO

When a page has been selected , check its reference bit.

- If 0 proceed to replace the page
- If 1 give the page a second chance and move on to the next FIFO page.
- When a page gets a second chance, its reference bit is cleared and arrival time is reset to current time.
- Hence a second chance page will not be replaced until all other pages are replaced.



(iii) Enhanced Second Chance Algorithm

Consider both reference bit and modify bit o There are four possible classes

1. (0,0) – neither recently used nor modified Best page to replace
2. (0,1) – not recently used but modified page has to be written out before replacement.
3. (1,0) - recently used but not modified page may be used again
4. (1,1) – recently used and modified page may be used again and page has to be written to disk

(e) Counting-Based Page Replacement

Keep a counter of the number of references that have been made to each page

1. **Least Frequently Used (LFU)Algorithm**: replaces page with smallest count
2. **Most Frequently Used (MFU)Algorithm**: replaces page with largest count
   -It is based on the argument that the page with the smallest count was probably just brought in and has yet to be used.

Page Buffering Algorithm

These are used along with page replacement algorithms to improve their performance

Technique 1:

- A pool of free frames is kept.
- When a page fault occurs, choose a victim frame as before.
- Read the desired page into a free frame from the pool
- The victim frame is written onto the disk and then returned to the pool of free frames.

Technique 2:

- Maintain a list of modified pages.
- Whenever the paging device is idles, a modified is selected and written to disk and its modify bit is reset.

Technique 3:

- A pool of free frames is kept.
- Remember which page was in each frame.
- If frame contents are not modified then the old page can be reused directly from the free frame pool when needed

Allocation of Frames

There are two major allocation schemes

- Equal Allocation
- Proportional Allocation

Equal allocation

If there are n processes and m frames then allocate m/n frames to each process.

Example: If there are 5 processes and 100 frames, give each process 20 frames.

Proportional allocation

Allocate according to the size of process

Let s_i be the size of process i.

Let m be the total no. of frames

Then $S = \sum s_i$

$a_i = s_i / S * m$where a_i is the no.of frames allocated to process i.

Global vs. Local Replacement

Global replacement – each process selects a replacement frame from the set of all frames; one process can take a frame from another.

Local replacement – each process selects from only its own set of allocated frames.

PAGING

- It is a memory management scheme that permits the physical address space of a process to be noncontiguous.
- It avoids the considerable problem of fitting the varying size memory chunks on to the backing store.

(i) Basic Method:

- Divide logical memory into blocks of same size called **"pages"**.
- Divide physical memory into fixed-sized blocks called **"frames"**
- Page size is a power of 2, between 512 bytes and 16MB.

Address Translation Scheme

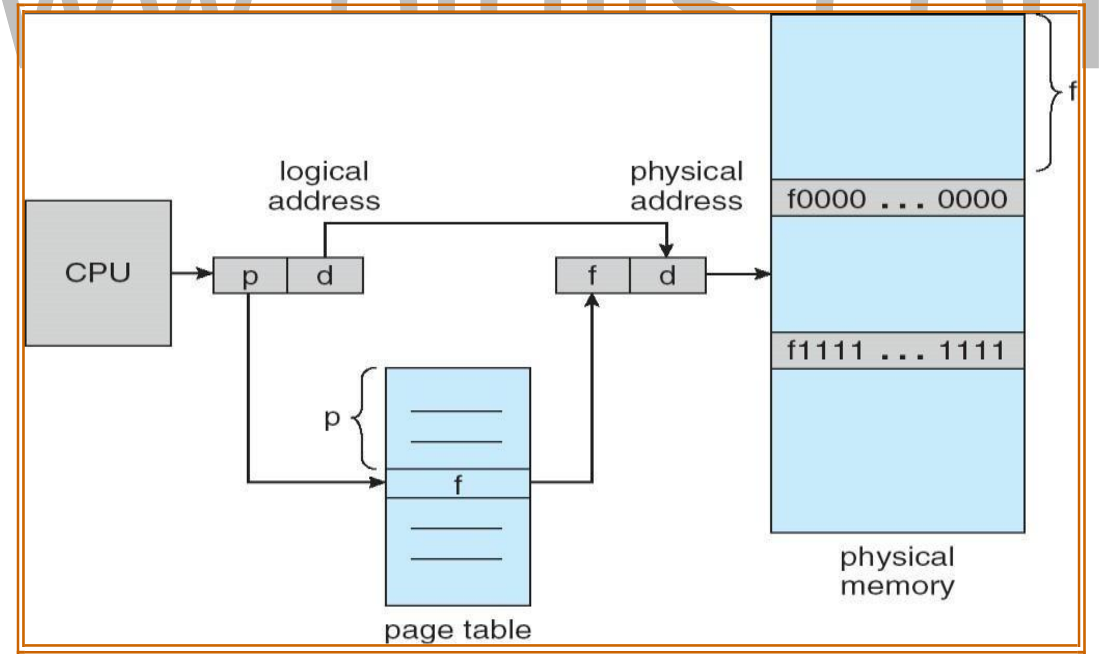
Address generated by CPU(logical address) is divided into:

Page number *(p)* – used as an index into a page table which contains base address of each page in physical memory

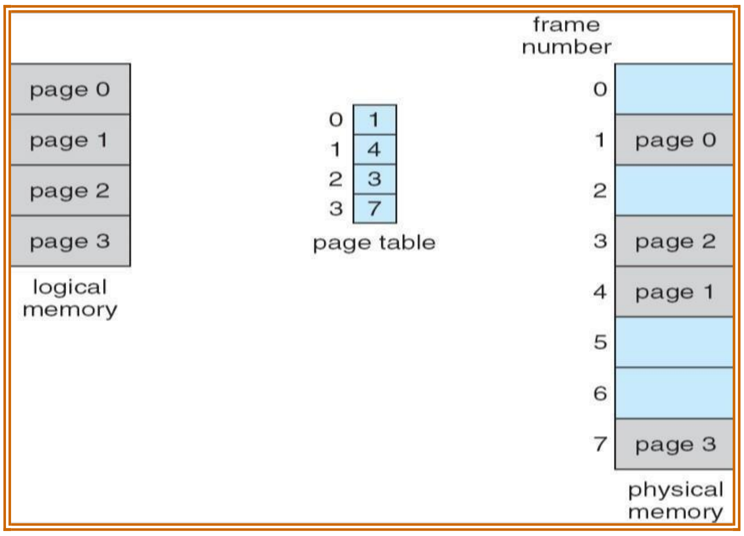
Page offset *(d)* – combined with base address to define the physical address i.e.,

Physical address = base address + offset

Paging Hardware



Paging model of logical and physical memory



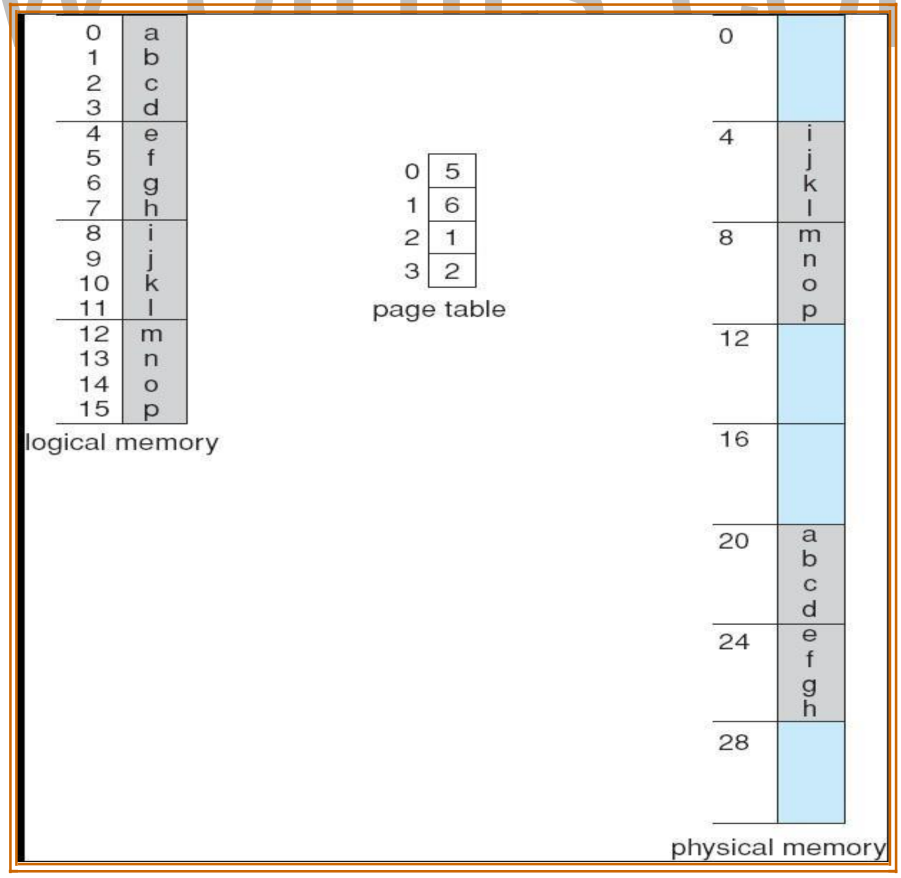
Paging example for a 32-byte memory with 4-byte pages

Page size = 4 bytes

Physical memory size = 32 bytes i.e (4 X 8 = 32 so, 8 pages) Logical address "0' maps to physical address 20 i.e ((5 X 4) +0) Where Frame no = 5,

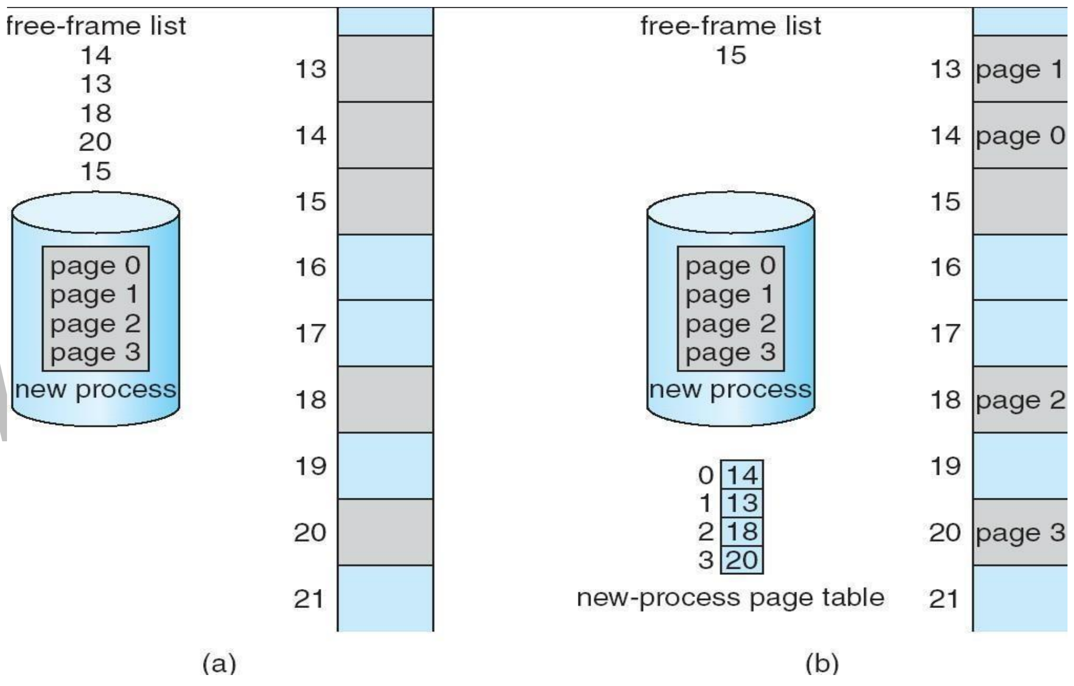
Page size = 4,

Offset = 0



Allocation

- When a process arrives into the system, its size (expressed in pages) is examined.
- Each page of process needs one frame. Thus if the process requires _n' pages, at least _n' frames must be available in memory.
- If _n' frames are available, they are allocated to this arriving process.
- The 1st page of the process is loaded into one of the allocated frames & the frame number is put into the page table.
- Repeat the above step for the next pages & so on.



(a) Before Allocation (b) After Allocation

Frame table: It is used to determine which frames are allocated, which frames are available, how many total frames are there, and so on.(ie) It contains all the information about the frames in the physical memory.

(ii) Hardware implementation of Page Table

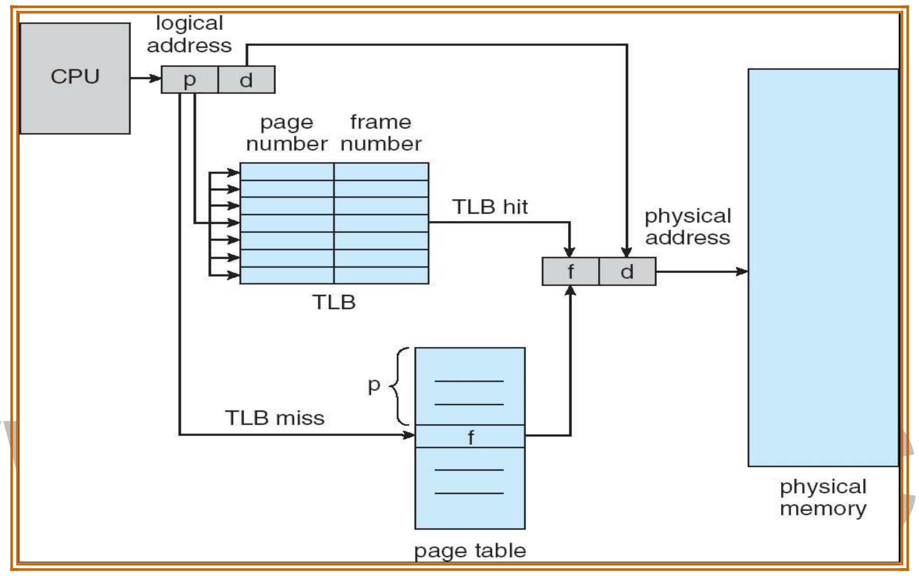
This can be done in several ways :

1. Using PTBR
2. TLB

The simplest case is **Page-table base register (PTBR)**, is an index to point the page table.

1. TLB (Translation Look-aside Buffer)

- It is a fast lookup hardware cache.
- It contains the recently or frequently used page table entries.
- It has two parts: Key (tag) & Value.
- More expensive.



Paging Hardware with TLB

When a logical address is generated by CPU, its page number is presented to TLB.

TLB hit: If the page number is found, its frame number is immediately available & is used to access memory

TLB miss: If the page number is not in the TLB, a memory reference to the page table must be made.

Hit ratio: Percentage of times that a particular page is found in the TLB.

For example hit ratio is 80% means that the desired page number in the TLB is 80% of the time.

Effective Access Time:

Assume hit ratio is 80%.

- If it takes 20ns to search TLB & 100ns to access memory, then the memory access takes 120ns(TLB hit)
- If we fail to find page no. in TLB (20ns), then we must 1st access memory for page table (100ns) & then access the desired byte in memory (100ns).
- Therefore Total = 20 + 100 + 100

  = 220 ns(TLB miss).
- Then Effective Access Time (EAT) = 0.80 X (120 + 0.20) X 220.

  =140ns.

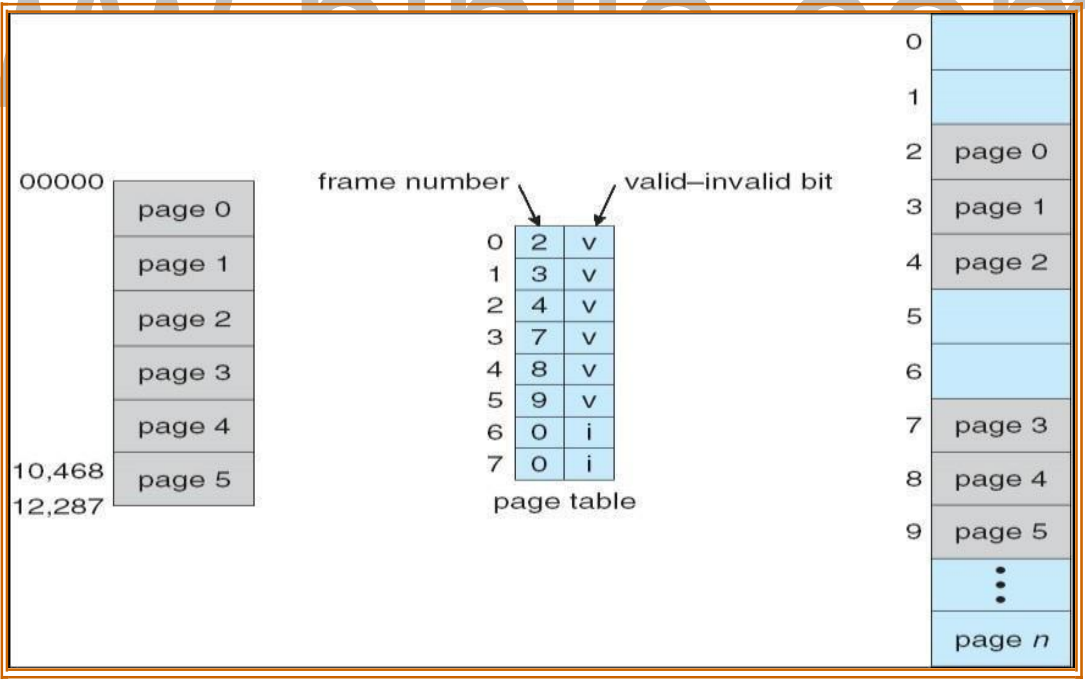
(iii) Memory Protection

Memory protection implemented by associating protection bit witheach frame

Valid-invalid bit attached to each entry in the page table:

"valid (v)" indicates that the associated page is in the process' logical address space, and is thus a legal page

"invalid (i)" indicates that the page is not in the process' logicaladdress space



(iv) Structures of the Page Table

a) Hierarchical Paging

b) Hashed Page Tables

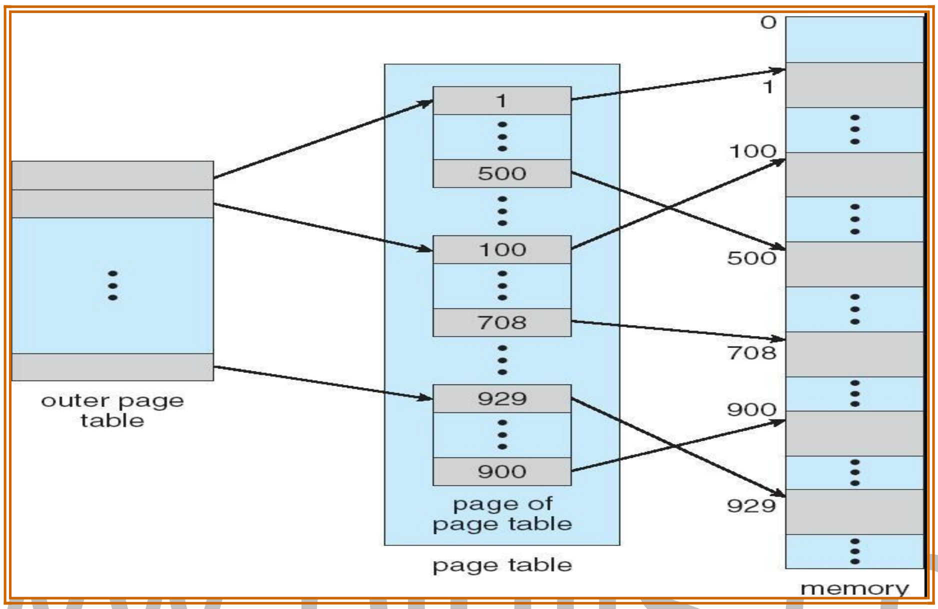
c) Inverted Page Tables

a) Hierarchical Paging

Break up the Page table into smaller pieces. Because if the page table is too large then it is quit difficult to search the page number.

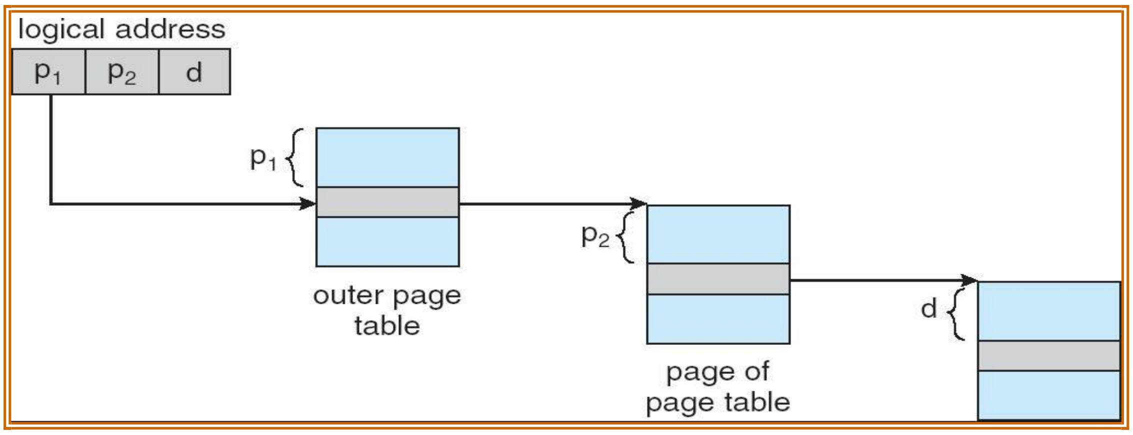
Example: "Two-Level Paging "

page number		page offset
p_i	p_2	d
10	10	12



Address-Translation Scheme

Address-translation scheme for a two-level 32-bit paging architecture



It requires more number of memory accesses, when the number of levels is increased.

(b) Hashed Page Tables

Each entry in hash table contains a linked list of elements that hash tothe same location.

Each entry consists of;

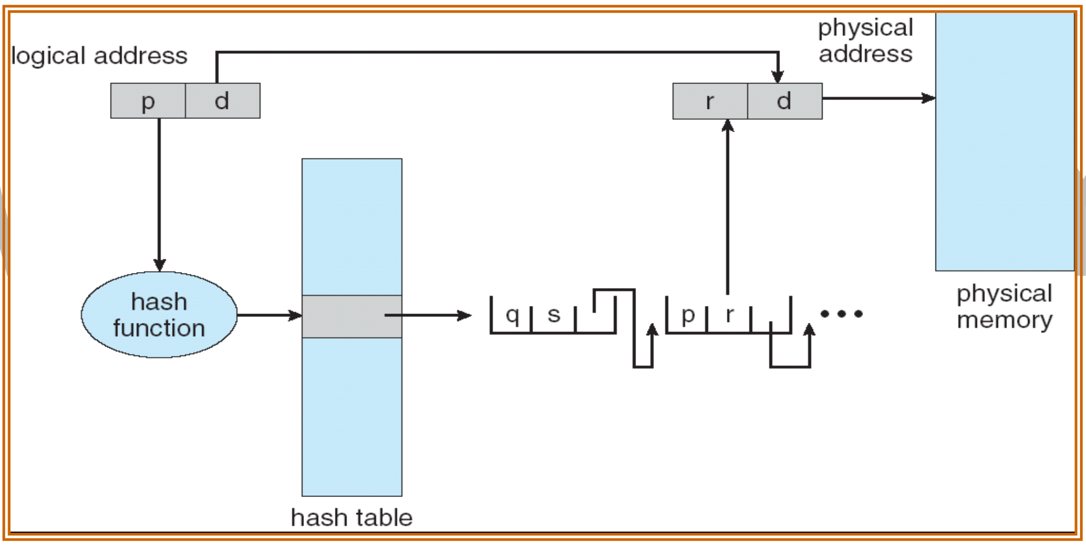
(a) Virtual page numbers

(b) Value of mapped page frame.

(c) Pointer to the next element in the linked list.

Working Procedure:

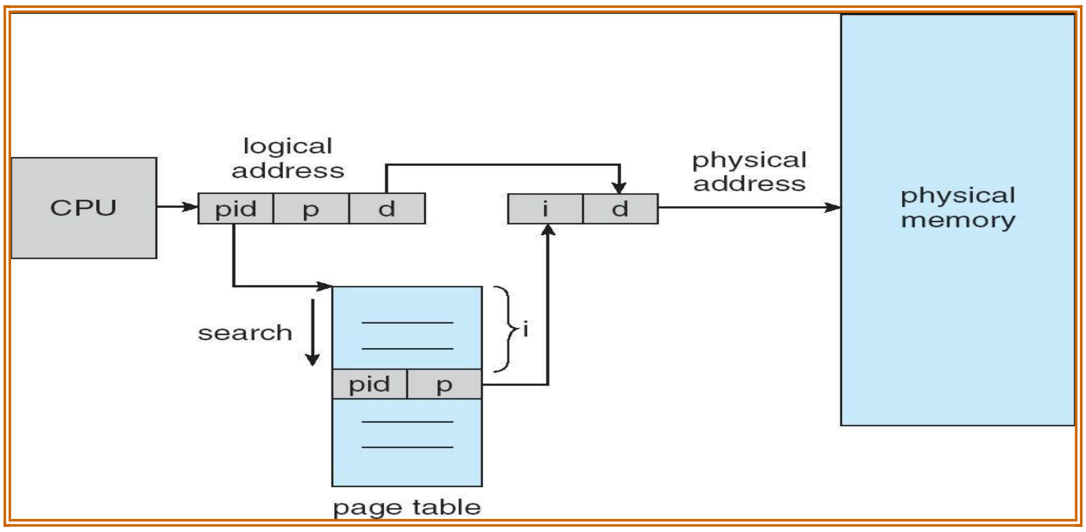
- The virtual page number in the virtual address is hashed into thehash table.
- Virtual page number is compared to field (a) in the 1st element in the linked list.
- If there is a match, the corresponding page frame (field (b)) is used to form the desired physical address.
- If there is no match, subsequent entries in the linked list are searched for a matching virtual page number.



Clustered page table: It is a variation of hashed page table & is similar to hashed page table except that each entry in the hash table refers to several pages rather than a single page.

(c) **Inverted Page Table**

It has one entry for each real page (frame) of memory & each entry consists of the virtual address of the page stored in that real memory location, with information about the process that owns that page. So, only one page table is in the system.



- When a memory reference occurs, part of the virtual address ,consisting of <Process-id, Page-no> is presented to the memory sub-system.
- Then the inverted page table is searched for match:

(i) If a match is found, then the physical address is generated.

(ii) If no match is found, then an illegal address access has been attempted.

Merit: Reduce the amount of memory needed.

Demerit: Improve the amount of time needed to search the table when a page reference oocurs.

(v) Shared Pages

One advantage of paging is the possibility of sharing common code.

Shared code

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

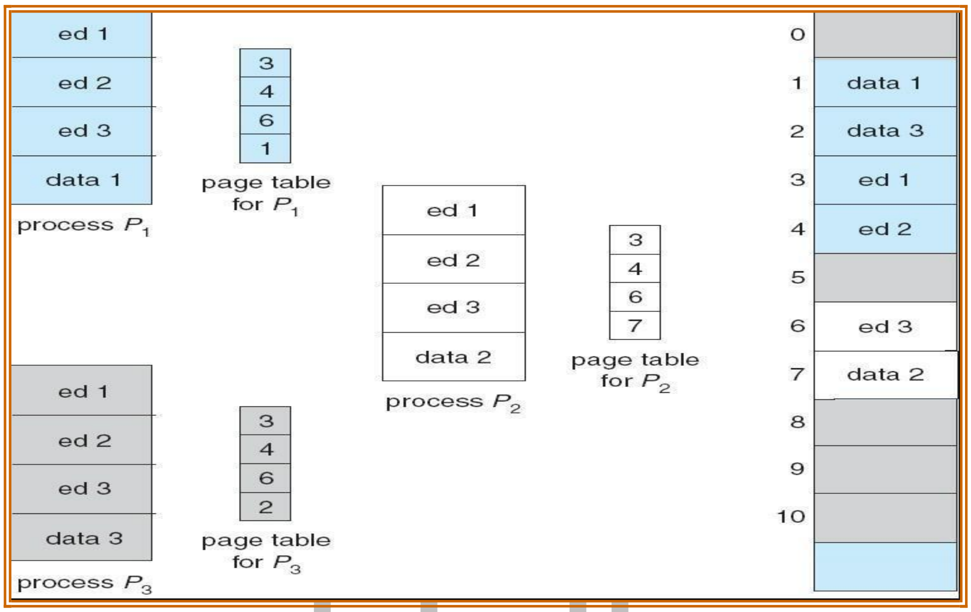
Shared code must appear in same location in the logical address space of all processes

Reentrant code (Pure code): Non-self modifying code. If the code is reentrant, then it never changes during execution. Thus two or more processes can execute the same code at the same time.

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

EXAMPLE:



Drawback of Paging – Internal fragmentation

In the worst case a process would need n pages plus one byte. It would be allocated n+1 frames resulting in an **internal fragmentation** of almost an entire frame.

Example:

Page size = 2048 bytes

Process size= 72766 bytes

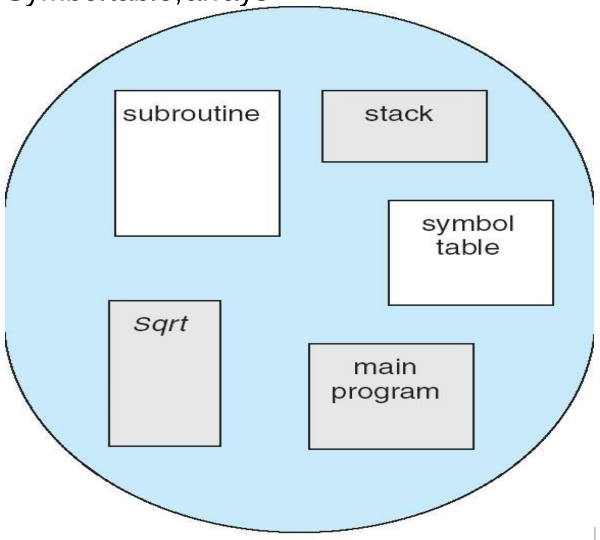
Process needs 35 pages plus 1086 bytes.

It is allocated 36 frames resulting in an internal fragmentation of 962 bytes.

SEGMENTATION

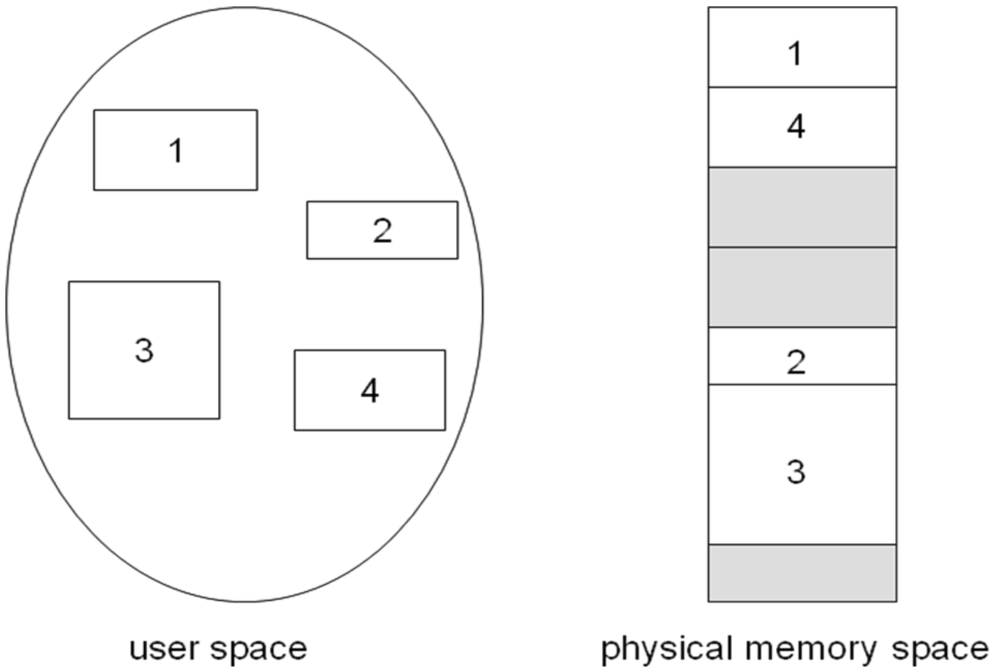
Memory-managementscheme that supports user view of memory

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



logical address

Logical View of Segmentation



Segmentation Hardware

Logical address consists of a two tuple :

<Segment-number, offset>

Segment table – maps two-dimensional physical addresses; each table entry 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;

Segment number_s is legal, if *s* < STLR

Relocation.

Dynamic

By segment table

Sharing.

- shared segments
- same segment number

Allocation.

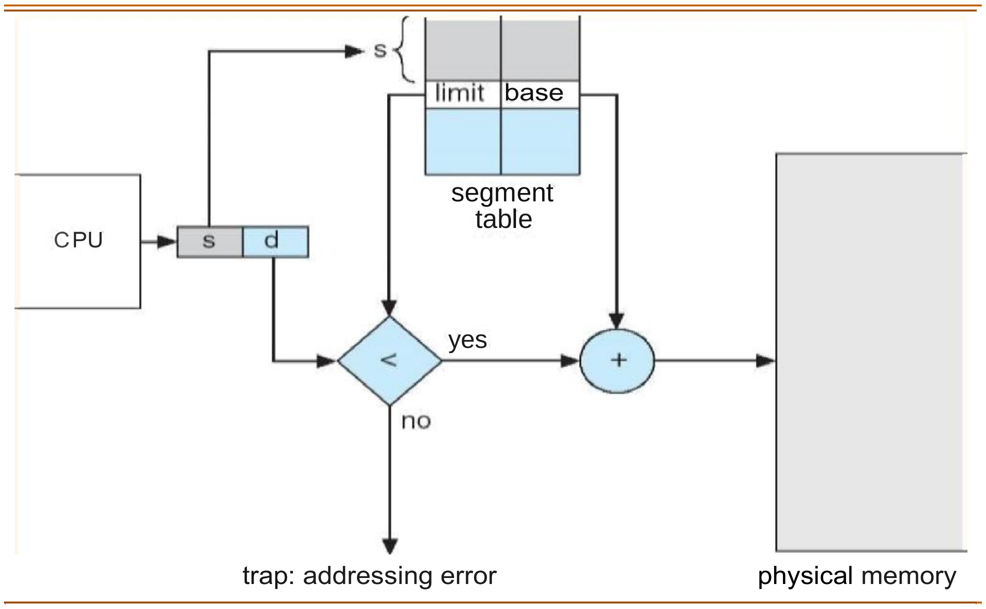
- first fit/best fit
- external fragmentation

Protection: With each entry in segment table associate:

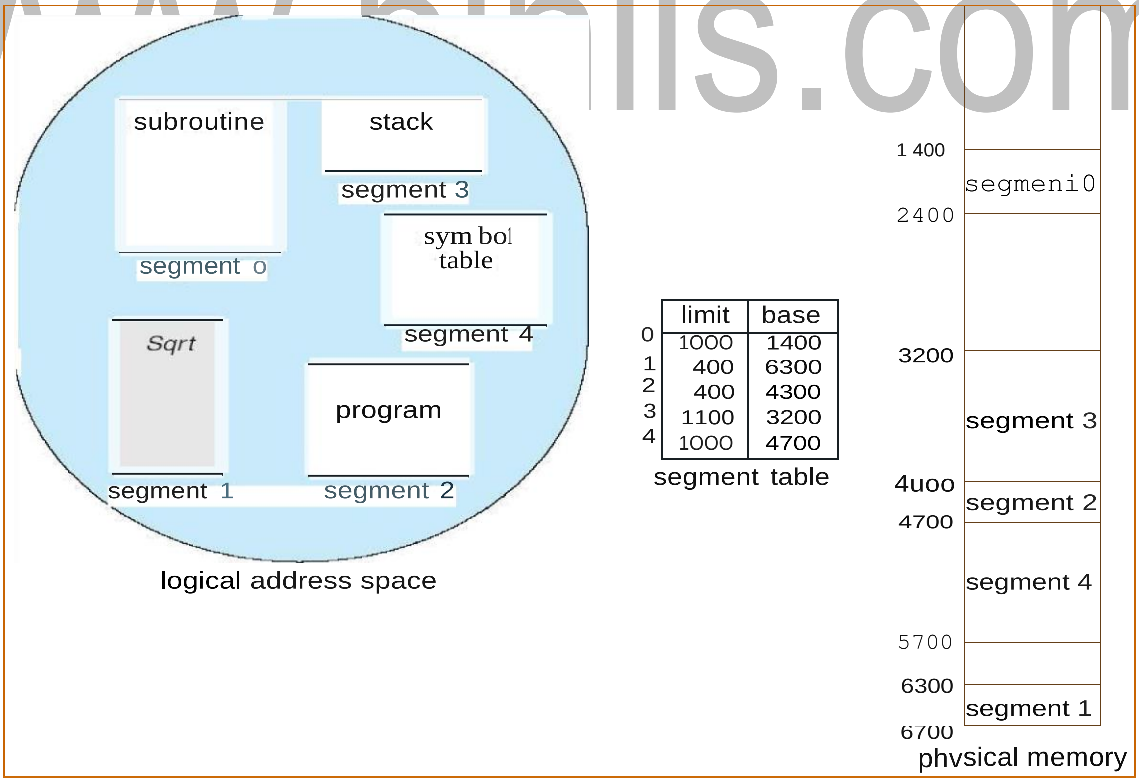
- validation bit = 0   illegal segment
- read/write/execute privileges

- Protection bits associated with segments; code sharing occurs at segment level
- Since segments vary in length, memory allocation is a dynamic storage- allocation problem
- A segmentation example is shown in the followingdiagram

Address Translation scheme

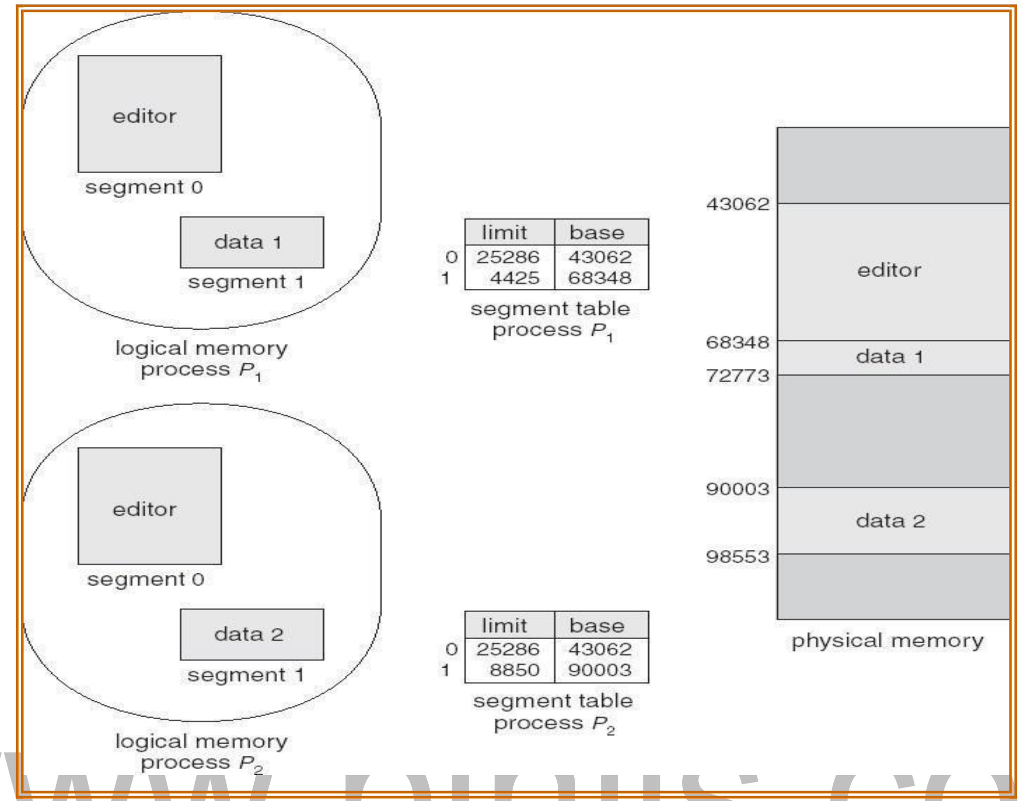


EXAMPLE:



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

Sharing of Segments



- Another advantage of segmentation involves the sharing of code or data.
- Each process has a segment table associated with it, which the dispatcher uses to define the hardware segment table when this process is given the CPU.
- Segments are shared when entries in the segment tables of two different processes point to the same physical location.

Segmentation withpaging

The IBM OS/ 2.32 bit version is an operating system running on top of the Intel386 architecture. The 386 uses segmentation with paging for memory management. The maximum number of segments per process is 16 KB, and each segment can be as large as 4 gigabytes.

The local-address space of a process is divided into two partitions.

- The first partition consists of up to 8 KB segments that are private to that process.
- The second partition consists of up to 8KB segments that are shared among all the processes.

- Information about the first partition is kept in the **local descriptor table (LDT)**, information about the second partition is kept in the **global descriptor table (GDT)**.
- Each entry in the LDT and GDT consist of 8 bytes, with detailed information about a particular segment including the base location and length of the segment.
- The logical address is a pair (selector, offset) where the selector is a 16-bit number:

s	g	p
13	1	2

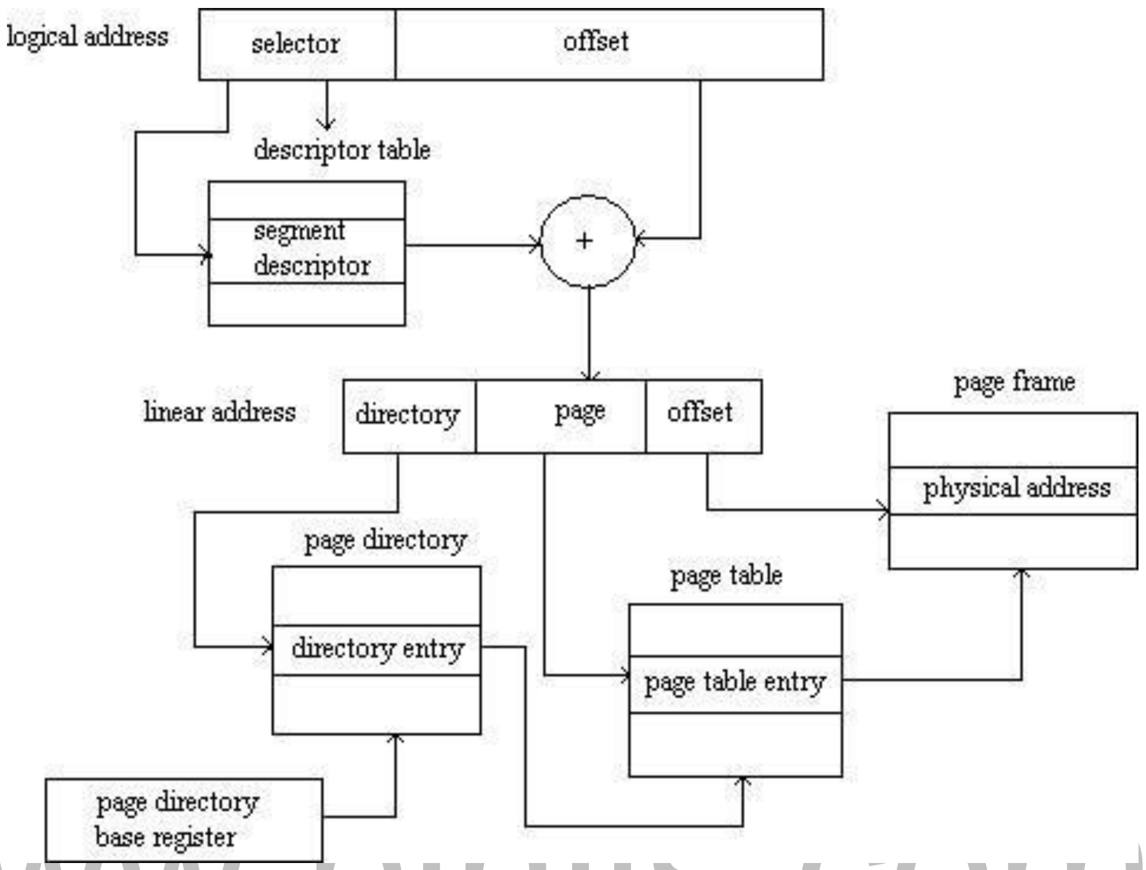
Where s designates the segment number, g indicates whether the segment is in the GDT or LDT, and p deals with protection. The offset is a 32-bit number specifying the location of the byte within the segment in question.

- The base and limit information about the segment in question are used to generate a linear-address.

- First, the limit is used to check for address validity. If the address is not valid, a memory fault is generated, resulting in a trap to the operating system. If it is valid, then the value of the offset is added to the value of the base, resulting in a 32-bit linear address. This address is then translated into a physical address.

- The linear address is divided into a page number consisting of 20 bits, and a page offset consisting of 12 bits. Since we page the page table, the page number is further divided into a 10-bit page directory pointer and a 10-bit page table pointer. The logical address is as follows.

p1	p2	d
10	10	12



- To improve the efficiency of physical memory use. Intel 386 page tables can be swapped to disk. In this case, an invalid bit is used in the page directory entry to indicate whether the table to which the entry is pointing is in memory or on disk.

- If the table is on disk, the operating system can use the other 31 bits to specify the disk location of the table; the table then can be brought into memory on demand.

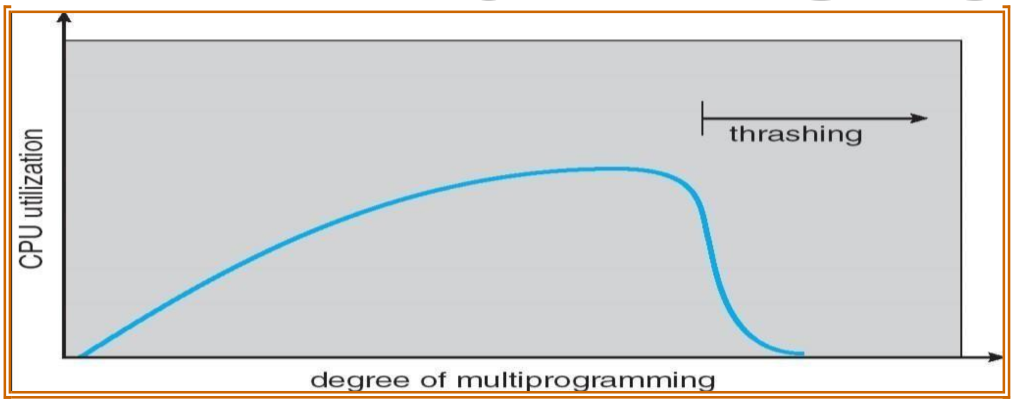
THRASHING

High paging activity is called **thrashing**.

If a process does not have –enough pages, the page-fault rate is very high.

This leads to:

- Low CPU utilization
- Operating system thinks that it needs to increase the degree of multiprogramming
- Another process is added to the system
- When the CPU utilization is low, the OS increases the degree of multiprogramming.
- If global replacement is used then as processes enter the main memory they tend to steal frames belonging to other processes.
- Eventually all processes will not have enough frames and hence the page fault rate becomes very high.
- Thus swapping in and swapping out of pages only takes place.
- This is the cause of thrashing.



To **limit thrashing**, we can use a **local replacement** algorithm.

To prevent thrashing, there are two methods namely ,

- Working Set Strategy
- Page Fault Frequency

1. Working-Set Strategy

It is based on the assumption of the model of locality.

Locality is defined as the set of pages actively used together.

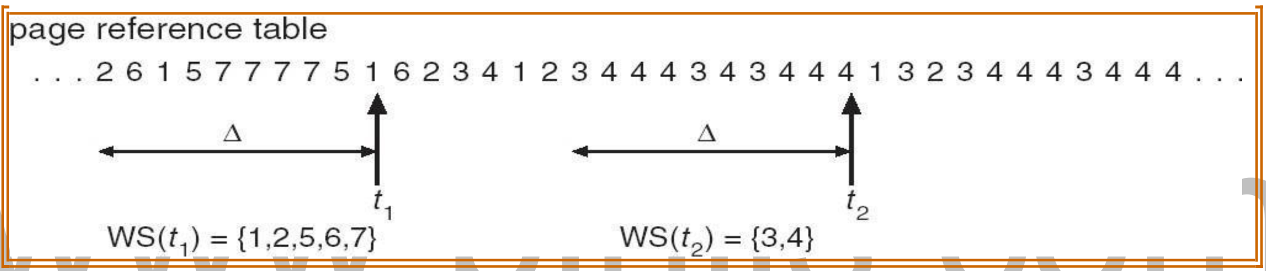
Working set is the set of pages in the most recent Δ page references is the working set window.

- if Δ too small , it will not encompass entire locality
- if Δ too large ,it will encompass several localities
- if Δ= it will encompass entire program

$D = \Sigma\ WSSi$

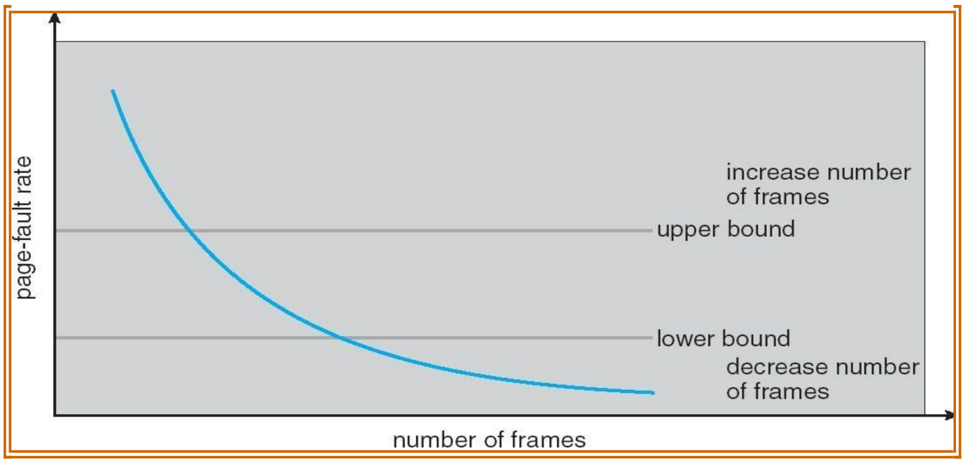
- Where WSSi is the working set size for process i.
- D is the total demand of frames

if $D > m$ then Thrashing will occur.



2. Page-Fault Frequency Scheme

- If actual rate too low, process loses frame
- If actual rate too high, process gains frame



Other Issues

1. **Prepaging**

- To reduce the large number of page faults that occurs at process startup
- Prepage all or some of the pages a process will need, before they are referenced
- But if prepaged pages are unused, I/O and memory are wasted

2. Page Size

Page size selection must take into consideration:

- Fragmentation
- Table size
- I/O overhead
- Locality

3. TLB Reach

- TLB Reach - The amount of memory accessible from the TLB
- TLB Reach = (TLB Size) X (Page Size)
- Ideally, the working set of each process is stored in the TLB, Otherwise there is a high degree of page faults.
- Increase the Page Size. This may lead to an increase in fragmentation as not all applications require a large page size
- Provide Multiple Page Sizes. This allows applications that require larger page sizes the opportunity to use them without an increase in fragmentation.

4. I/O interlock

- Pages must sometimes be locked into memory
- Consider I/O. Pages that are used for copying a file from a device must be
  - locked from being selected for eviction by a page replacement algorithm.

VIRTUAL MEMORY

It is a technique that allows the execution of processes that may not be completely in main memory.

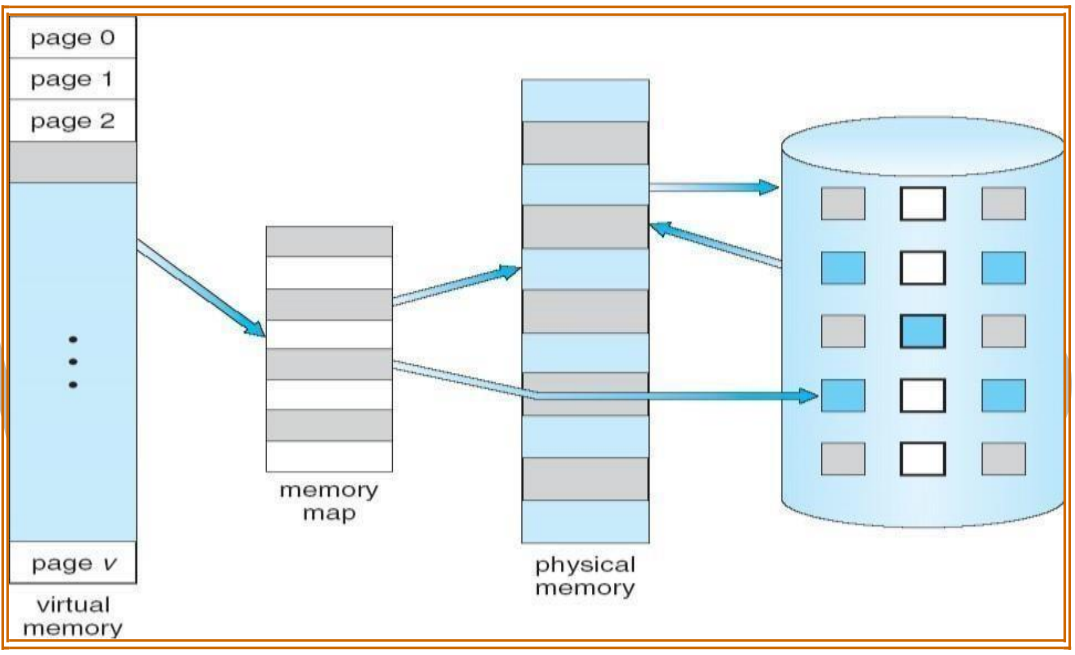
Advantages:

- Allows the program that can be larger than the physical memory.
- Separation of user logical memory from physical memory
- Allows processes to easily share files & address space.
- Allows for more efficient process creation.

Virtual memory can be implemented using

- Demand paging
- Demand segmentation

Virtual Memory that is Larger than Physical Memory



Demand Paging

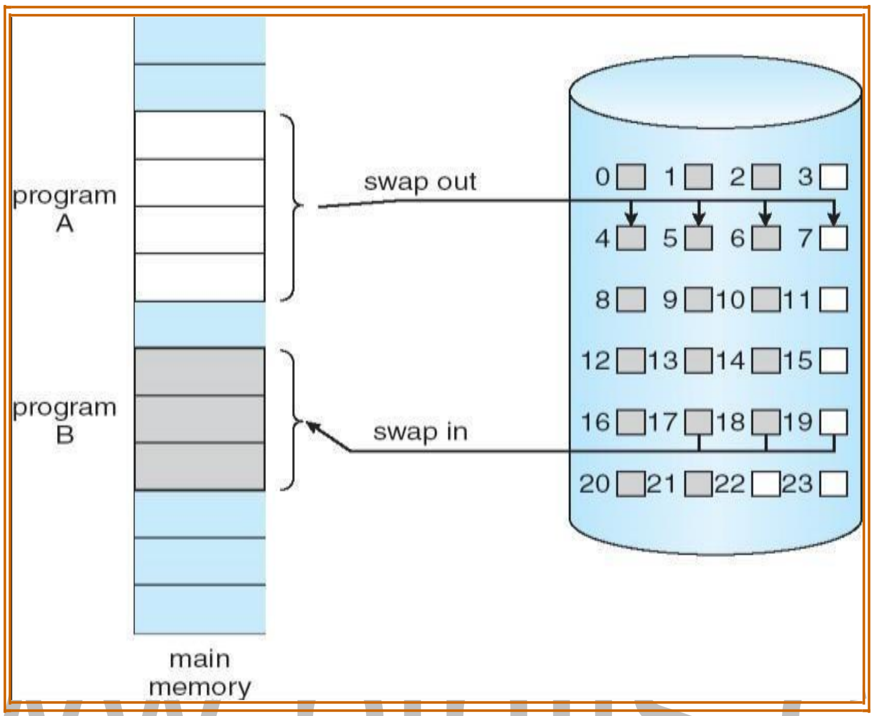
- It is similar to a paging system with swapping.
- Demand Paging - Bring a page into memory only when it is needed
- To execute a process, swap that entire process into memory. Rather than swapping the entire process into memory however, we use —Lazy Swapper

Lazy Swapper - Never swaps a page into memory unless that page will be needed.

Advantages

- Less I/O needed
- Less memory needed
- Faster response
- More users

Transfer of a paged memory to contiguous disk space



Basic Concepts:

Instead of swapping in the whole processes, the pager brings only those necessary pages into memory. Thus,

1. It avoids reading into memory pages that will not be used anyway.
2. Reduce the swap time.
3. Reduce the amount of physical memory needed.

To differentiate between those pages that are in memory & those that are on the disk we use the **Valid-Invalid bit**

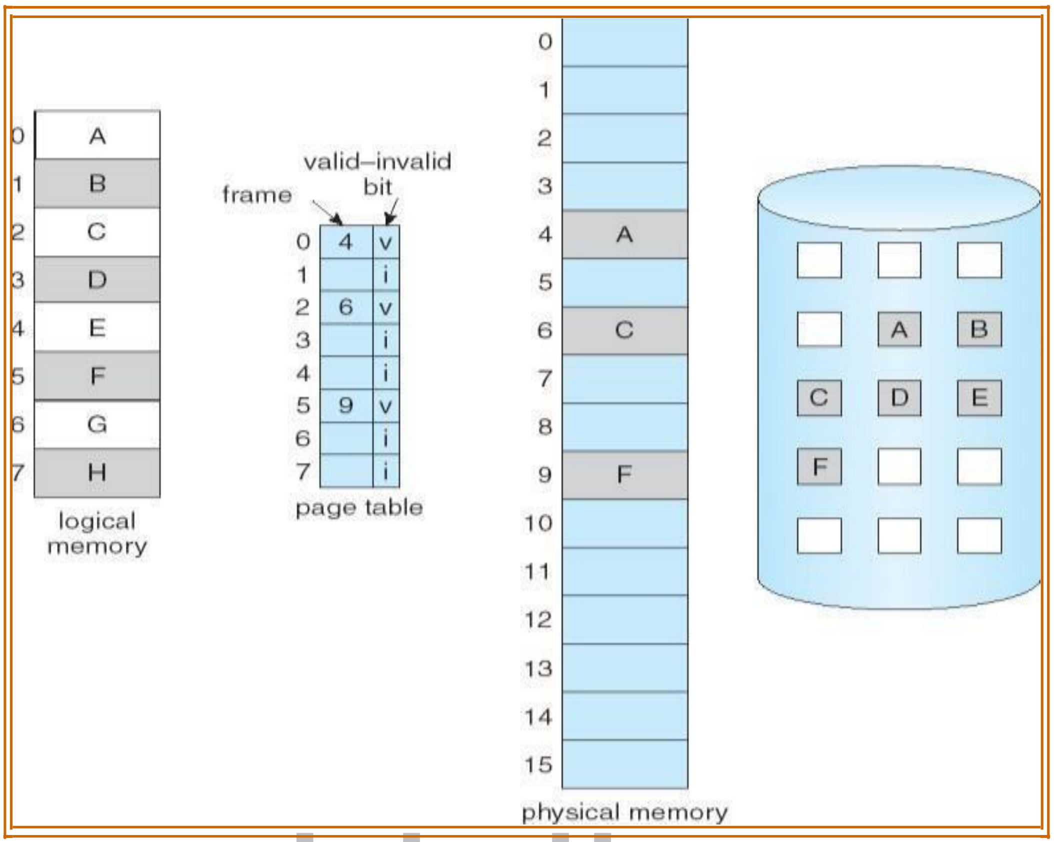
Valid-Invalid bit

A valid – invalid bit is associated with each page table entry.

Valid - > associated page is in memory.

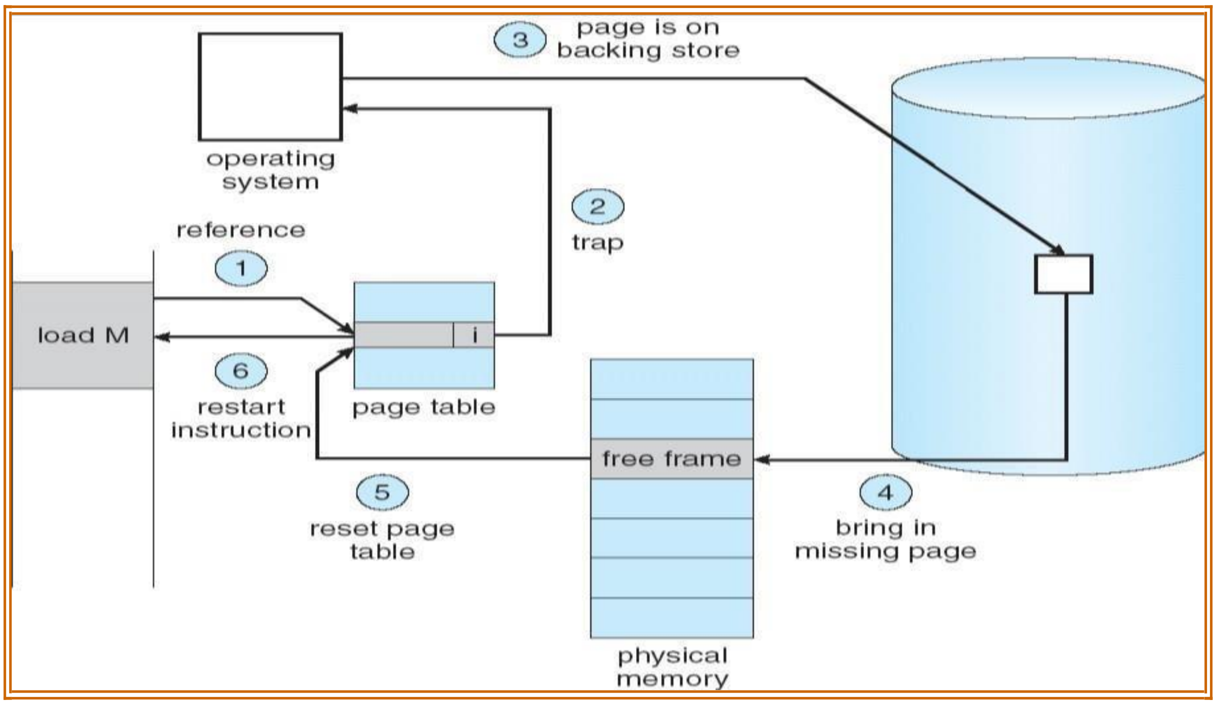
In-Valid -> invalid page, valid page but is currently on the disk

Page table when some pages are not in main memory



Page Fault

Access to a page marked invalid causes a page fault trap.



Steps in Handling a Page Fault

1. Determine whether the reference is a valid or invalid memory access
2. a) If the reference is invalid then terminate the process.

   b) If the reference is valid then the page has not been yet brought into main memory.
3. Find a free frame.
4. Read the desired page into the newly allocated frame.
5. Reset the page table to indicate that the page is now in memory.
6. Restart the instruction that was interrupted .

Pure demand paging

- Never bring a page into memory until it is required.
- We could start a process with no pages in memory**.**
- When the OS sets the instruction pointer to the 1st instruction of the process, which is on the non-memory resident page, then the process immediately faults for the page.
- After this page is bought into the memory, the process continue to execute, faulting as necessary until every page that it needs is in memory.

Performance of demand paging

Let p be the probability of a page fault 0 <*p* >1

Effective Access Time (EAT)

EAT = (1 – *p*) x ma + *p* x page fault time.

Where ma -> memory access, p ->Probability of page fault ($0 \leq p \leq 1$)

The memory access time denoted ma is in the range 10 to 200 ns.

If there are no page faults then EAT = ma.

To compute effective access time, we must know how much time is needed to service a page fault.

A page fault causes the following sequence to occur:

1. Trap to the OS
2. Save the user registers and process state.
3. Determine that the interrupt was a page fault.
4. Check whether the reference was legal and find the location of page on disk.
5. Read the page from disk to free frame.

a. Wait in a queue until read request is serviced.

b. Wait for seek time and latency time.

c. Transfer the page from disk to free frame.

6. While waiting ,allocate CPU to some other user.
7. Interrupt from disk.
8. Save registers and process state for other users.
9. Determine that the interrupt was from disk.
7. Reset the page table to indicate that the page is now in memory.
8. Wait for CPU to be allocated to this process again.
9. Restart the instruction that was interrupted .

Process Creation

Virtual memory enhances the performance of creating and running processes:

- Copy-on-Write
- Memory-Mapped Files

a) Copy-on-Write

fork()

Creates a child process as a duplicate of the parent process & it worked by creating copy of the parent address space for child, duplicating the pages belonging to the parent.

Copy-on-Write (COW)

Allows both parent and child processes to initially *share* the same pages in memory. These shared pages are marked as Copy-on- Write pages, meaning that if either process modifies a shared page, a copy of the shared page is created.

vfork():

With this the parent process is suspended & the child process uses the address space of the parent.

- Because vfork() does not use Copy-on-Write, if the child process changes any pages of the parent‘s address space, the altered pages will be visible to the parent once it resumes.
- Therefore, vfork() must be used with caution, ensuring that the child process does not modify the address space of the parent.

(b) Memory - mapped files:

Sequential read of a file on disk uses open() , read() and write()

Every time a file is accessed it requires a system call and disk access.

Alternative method: **“Memory - mapped files”**

Allowing a part of virtual address space to be logically associated with file

Mapping a disk block to a page in memory.