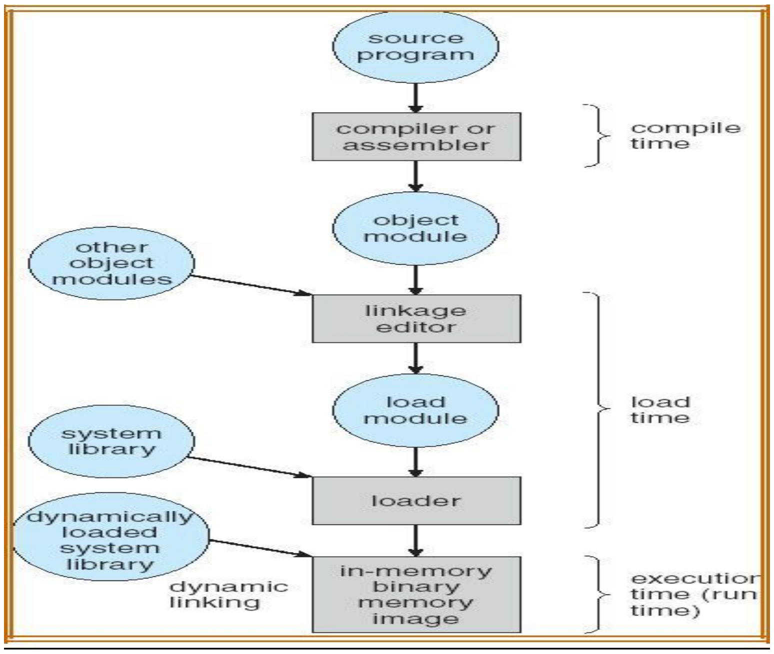
## MAIN MEMORY

- In general, to execute 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 for the process (where it will reside in main memory) 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

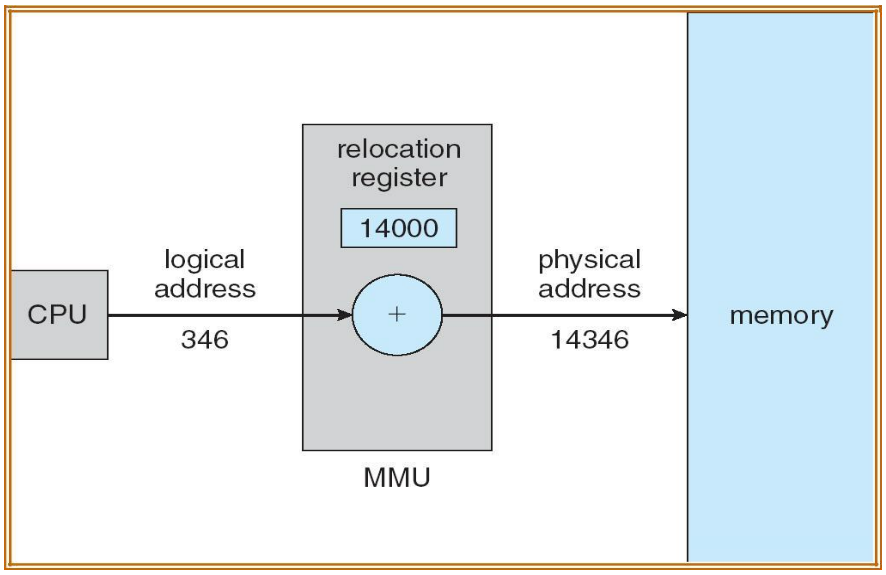
- **Logical address** – generated by the CPU; also referred to as **"virtual address"**
- **Physical address** – address seen by the memory unit.
- Logical and physical addresses are the **same** in ―compile-time and load-time address-binding schemes‖
- Logical (virtual) and physical addresses **differ** 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

## 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 program design

## 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

- **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 must be in memory for execution.
- 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 100MB & the backing store is a standard hard disk with a transfer rate of 50MBPS.

Transfer time = 100MB/50KB per second

= 2 sec = 200ms

Since we must swap both out and in, the total swap time is about 400 milliseconds

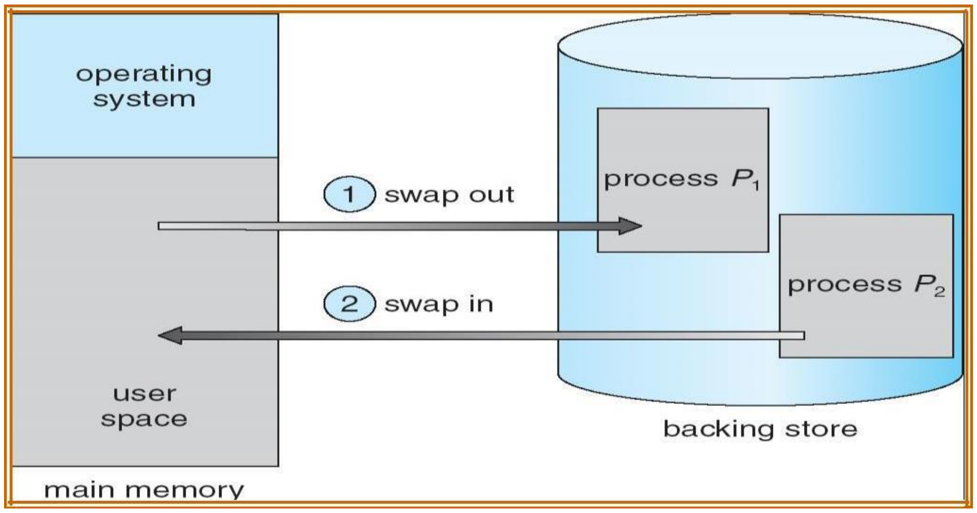


Fig : Swapping of two processes using a disk as a backing store

## CONTIGUOUS MEMORY ALLCOATION

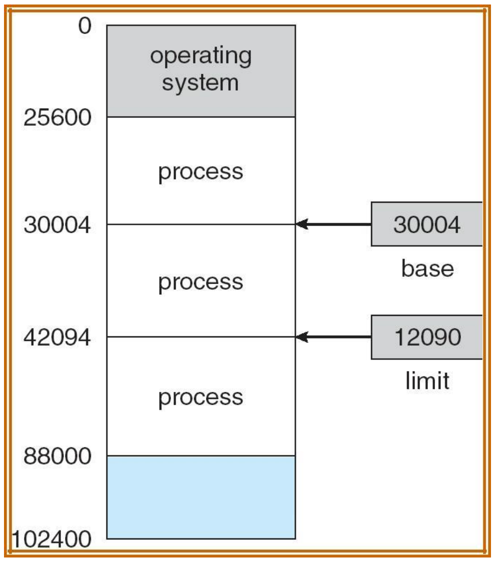
Contiguous memory allocation is a classical memory allocation model. Here, a system assigns consecutive memory blocks (that is, memory blocks having consecutive addresses) to a process. Contiguous memory allocation is one of the oldest memory allocation methods.

The memory is usually divided into two partitions: one for the resident operating system and one for the user processes. We can place the operating system in either low memory or high memory. Since the interrupt vector is often in low memory, programmers usually place the operating system in low memory.

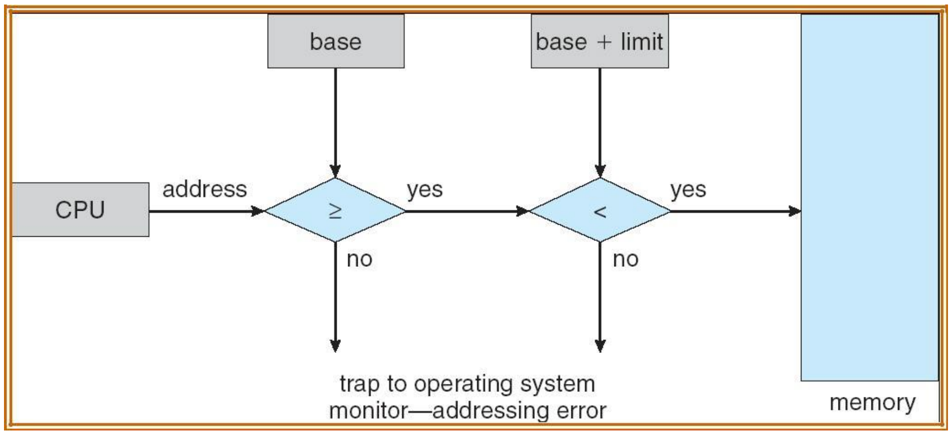
### (i) Memory Protection:

- It should consider;
    - 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

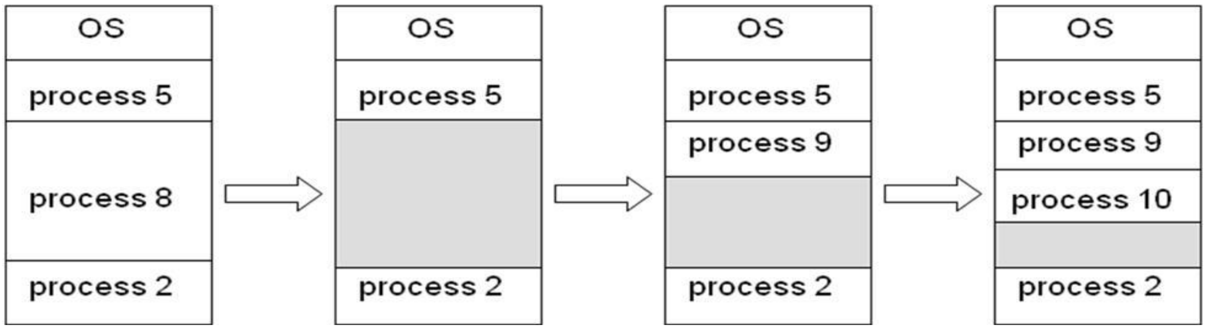


**HW address protection with base and limit registers**



**ii) Memory Allocation**

- Each process is contained in a single contiguous section of memory.
- There are two methods namely :
    - Fixed – Partition Method
    - Variable – Partition Method
- **<u>Fixed – Partition Method</u>** :
    - 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)
- **<u>Variable-partition method:</u>**
    - 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.

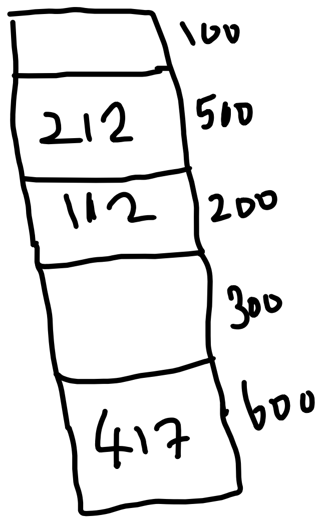
PROBLEM:

Given five memory partitions of 100Kb, 500Kb, 200Kb, 300Kb, 600Kb (in order) of fixed sized blocks, how would the first-fit, best-fit, and worstfit algorithms place processes of 212 Kb, 417 Kb, 112 Kb, and 426 Kb (in order)? Which algorithm makes the most efficient use of memory?

**ANSWER:**

**First-fit:**

212K is put in 500K partition
417K is put in 600K partition
112K is put in 200K partition
426K must wait



**Best-fit:**

212K is put in 300K partition
417K is put in 500K partition
112K is put in 200K partition
426K is put in 600K partition

**Worst-fit:**

212K is put in 600K partition
417K is put in 500K partition
112K is put in 300K partition
426K must wait

**In this example, best-fit turns out to be the best.**

| Job Number | Memory Requested |
|---|---|
| J1 | 20 K |
| J2 | 200 K |
| J3 | 500 K |
| J4 | 50 K |

| Memory location | Memory block size | Job number | Job size | Status | Internal fragmentation |
|---|---|---|---|---|---|
| 10567 | 200 K | J1 | 20 K | Busy | 180 K |
| 30457 | 30 K | | | Free | 30 |
| 300875 | 700 K | J2 | 200 K | Busy | 500 K |
| 809567 | 50 K | J4 | 50 K | Busy | None |
| Total available : | 980 K | Total used : | 270 K | | 710 K |

**iii) 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**.

## 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.

### 3.1 Basic Method:

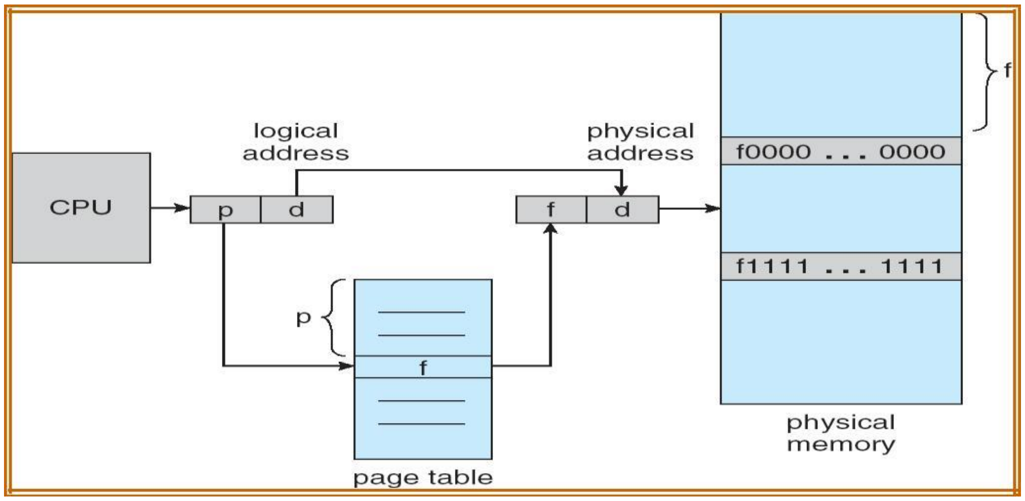
- Divide logical memory into blocks of same size called **"pages"**.
- Divide physical memory into fixed-sized blocks called **"frames"**
- Page size is a power of 2, between 512 bytes and 16MB.

### Address Translation Scheme

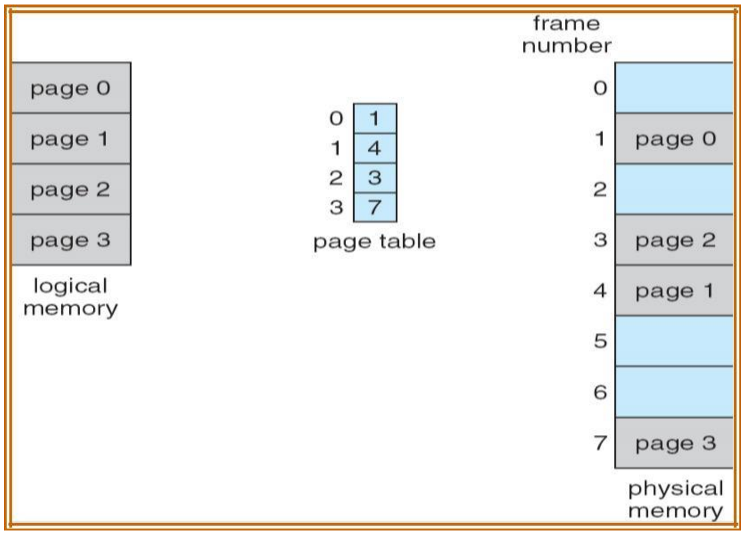
- Address generated by CPU(logical address) is divided into:
  - ✓ **Page number *(p)*** – used as an index into a page table which contains base address of each page in physical memory
  - ✓ **Page offset *(d)*** – combined with base address to define the physical address i.e.,

Physical address = base address + offset

**Paging Hardware**



**Paging model of logical and physical memory**



**<u>Paging example for a 32-byte memory with 4-byte pages</u>**

Page size = 4 bytes

Physical memory size = 32 bytes i.e ( 4 X 8 = 32 so, 8 pages)

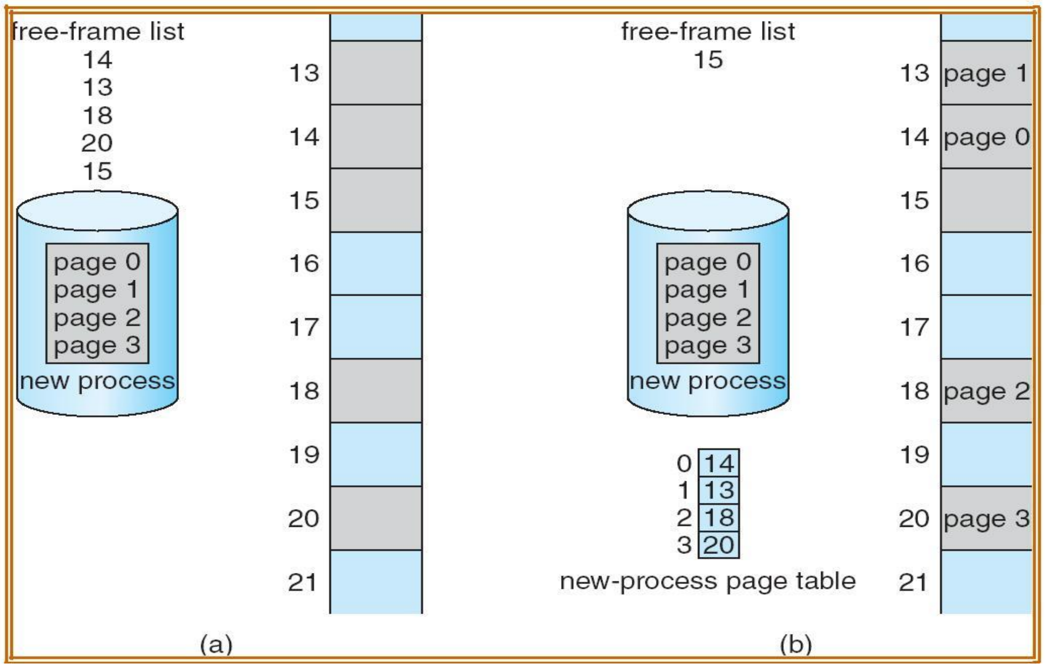
Logical address ‗0' maps to physical address 20 i.e ( (5 X 4) +0)

Where Frame no = 5, Page size = 4, Offset = 0

**<u>Allocation</u>**

- 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.



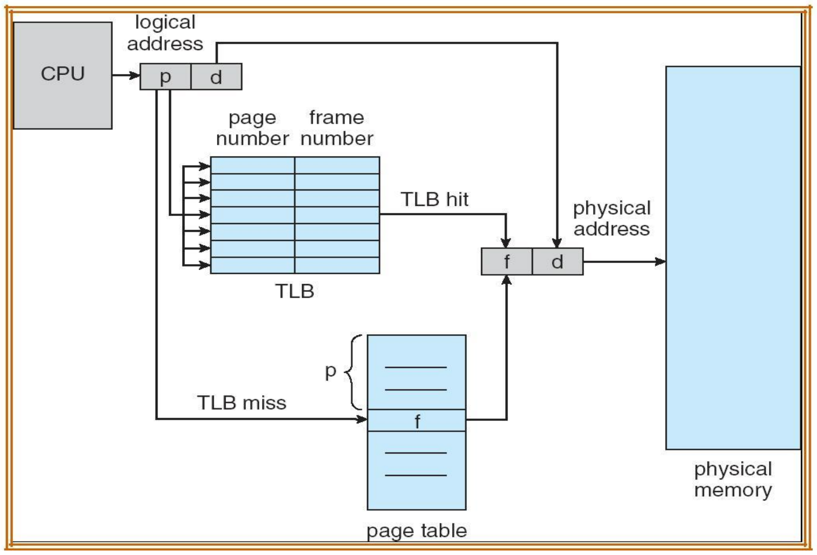
(a) Before Allocation (b) After Allocation

**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.

**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.
- **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.

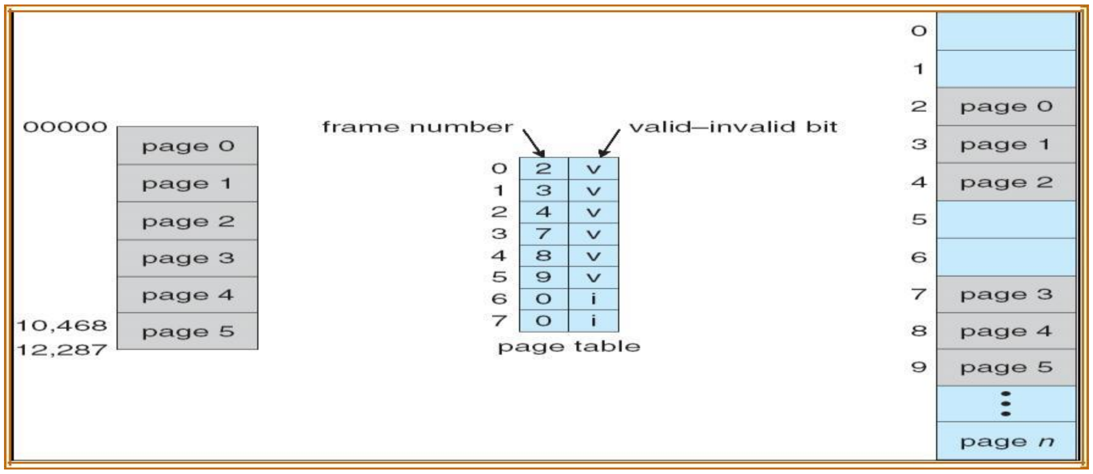
## Paging Hardware with TLB



- 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.

**(ii) Protection**

- Memory protection implemented by associating protection bit with each 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' logical address space



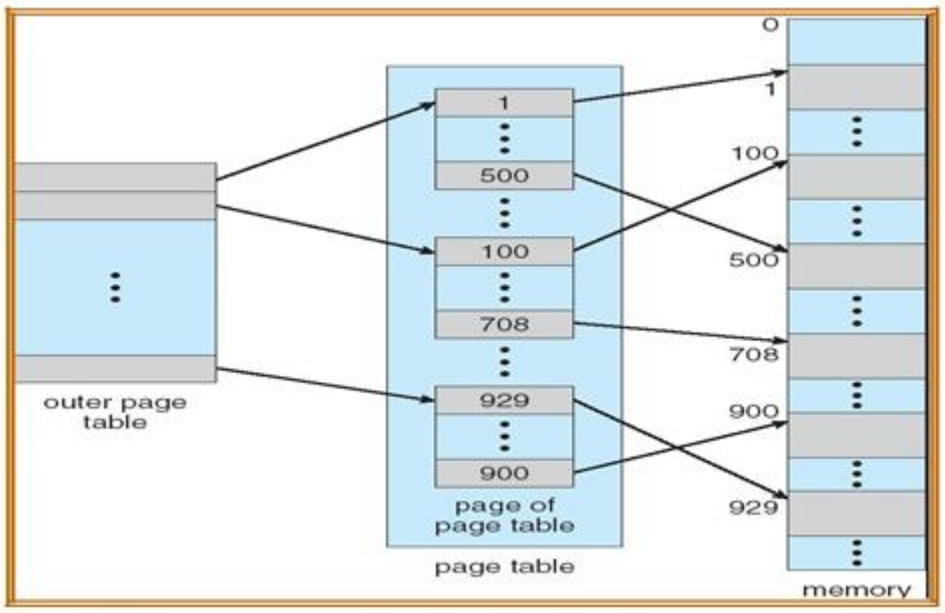
## Structures of the Page Table

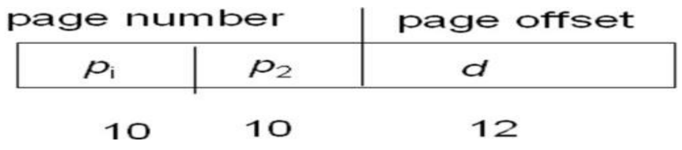
a) Hierarchical Paging
b) Hashed Page Tables
c) Inverted Page Tables

### 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"**



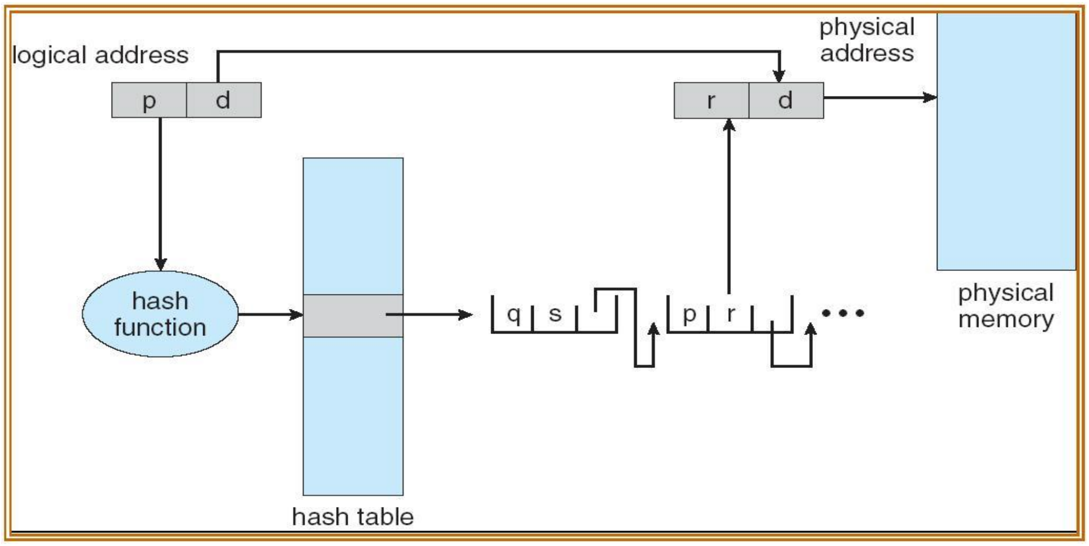


## (b) Hashed Page Tables

- Each entry in hash table contains a linked list of elements that hash to the 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.

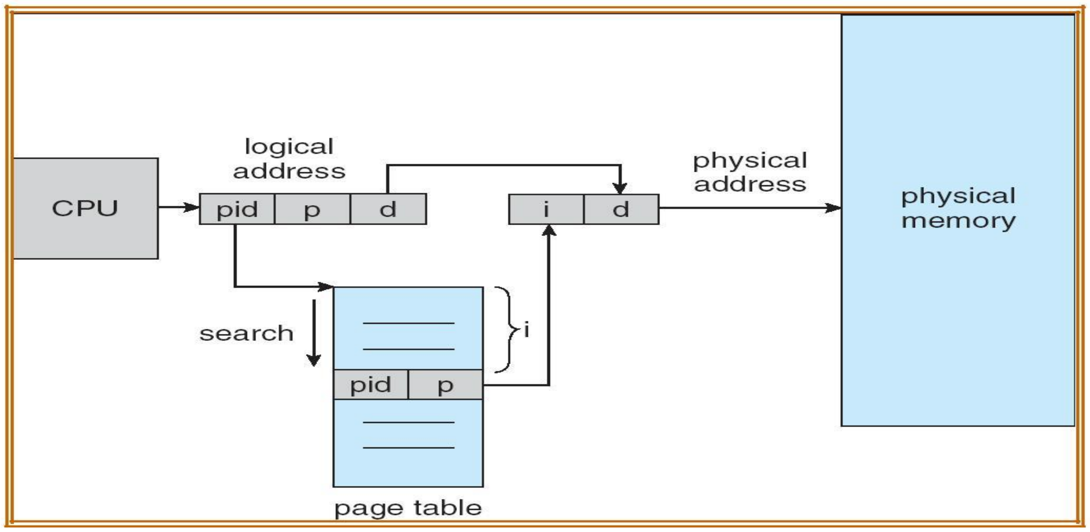
- Working Procedure:
  - The virtual page number in the virtual address is hashed into the hash table.
  - Virtual page number is compared to field (a) in the 1st element in the linked list.
  - If there is a match, the corresponding page frame (field (b)) is used to form the desired physical address.
  - 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.


CPU
logical address
pid
p
d
i
d
physical address
physical memory
search
i
pid
p
page table

- 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.

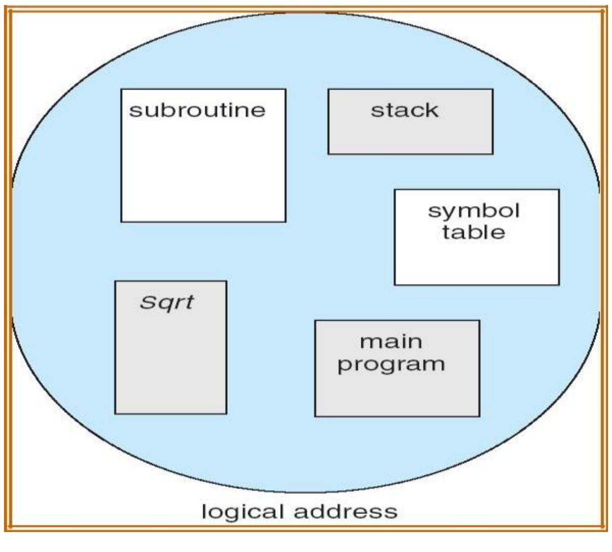
**(v) Shared Pages**

- 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

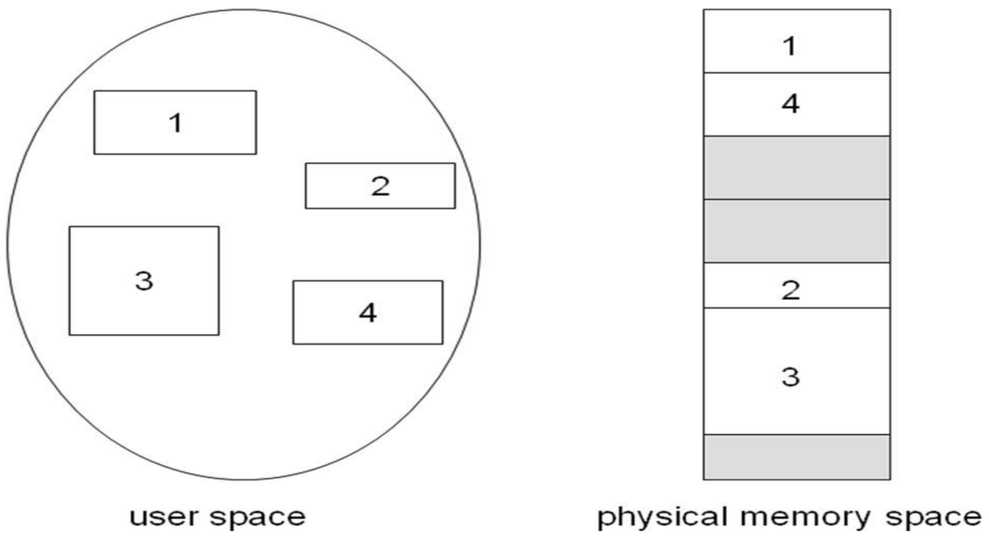
## <u>5. SEGMENTATION</u>

- Memory-management scheme that supports user view of memory
- A program is a collection of segments. A segment is a logical unit such as: Main program, Procedure, Function, Method, Object, Local variables, global variables, Common block, Stack, Symbol table, arrays

**User's View of a Program**



**Logical View of Segmentation**



**<u>5.1 Segmentation Hardware</u>**

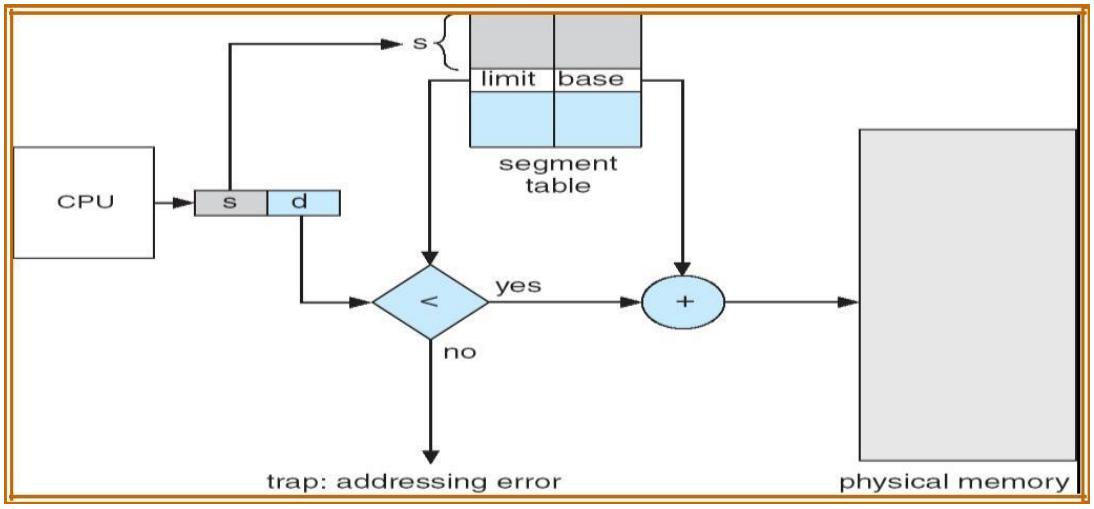
- Logical address consists of a two tuple :

  **<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;

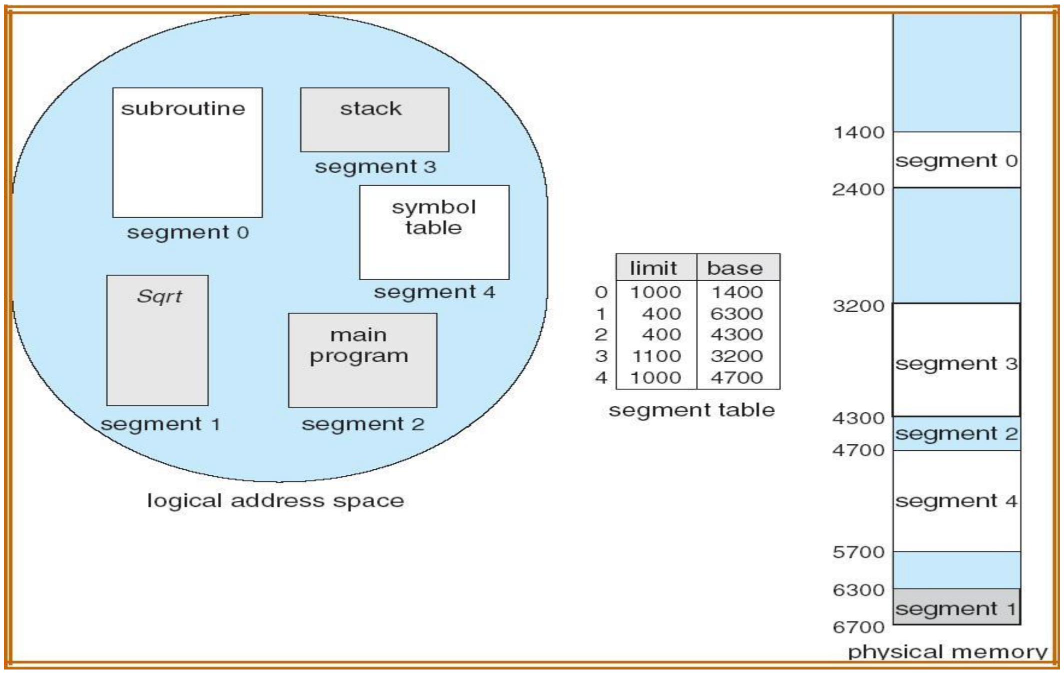
  Segment number_s‘ is legal, if *s* < STLR
- **Relocation**.
  - ✓ dynamic
  - ✓ by segment table
- **Sharing**.
  - ✓ shared segments
  - ✓ same segment number
- **Allocation**.
  - ✓ first fit/best fit
  - ✓ external fragmentation
- **Protection:** With each entry in segment table associate:
  - ✓ validation 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 following diagram

**Address Translation scheme**

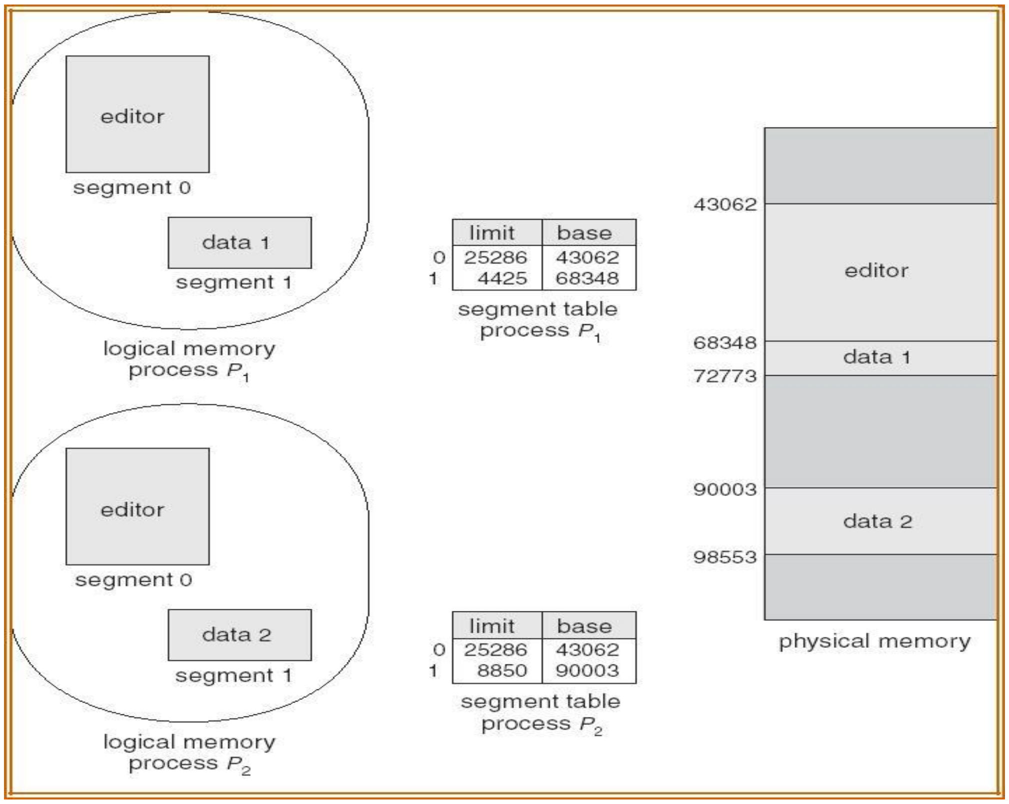


**EXAMPLE**:



| | 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.

**Paged segmentation Technique - IA-32 Architecture**

**<u>Segmentation with paging</u>**

- The IBM OS/2, 32 bit version is an operating system running on top of the Intel 386 architecture. The 386 uses segmentation with paging for memory management. The maximum number of segments per process is 16k, 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 a16-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 |

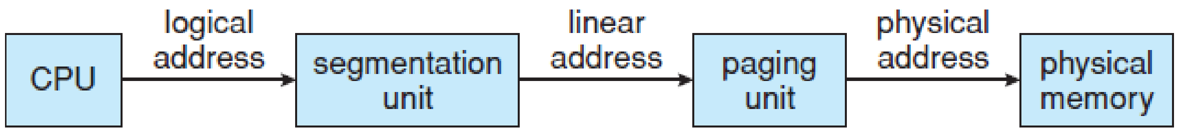


Fig: Logical to physical address translation in IA-32.

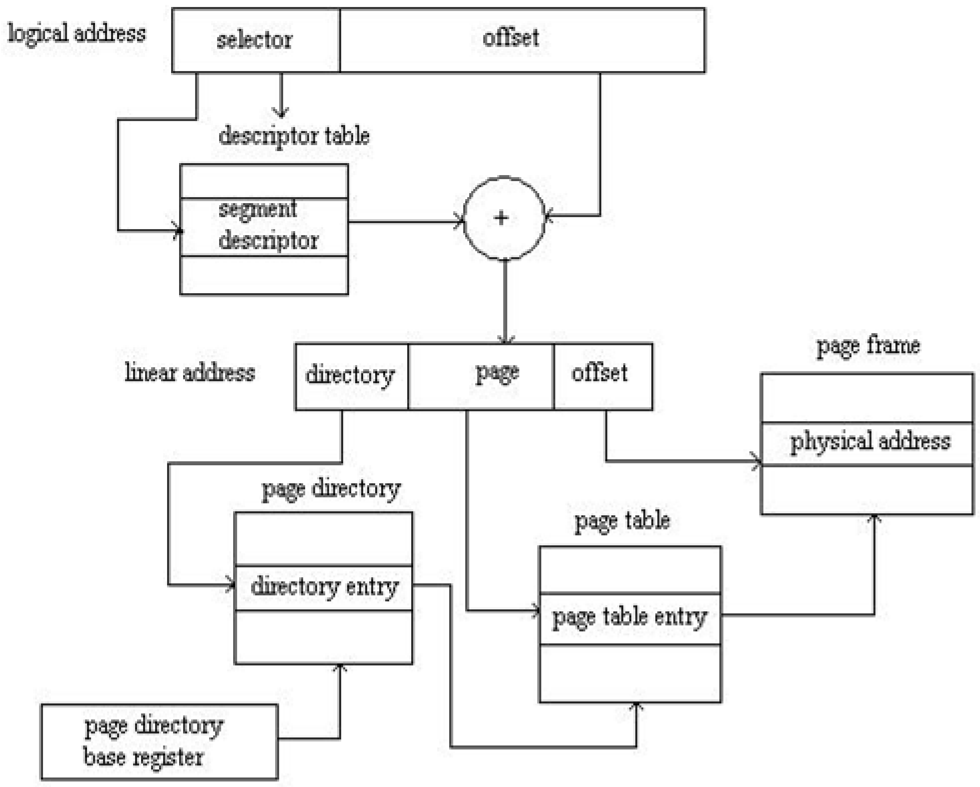
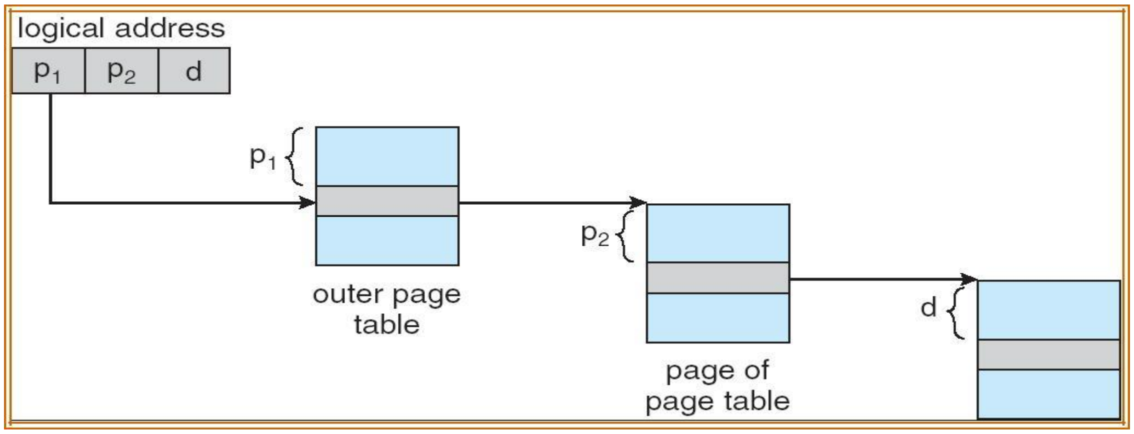


Fig: Segmented Paging in IA32 Architecture

- 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.

## Address-Translation Scheme/32 PAGING ARCHITECTURE

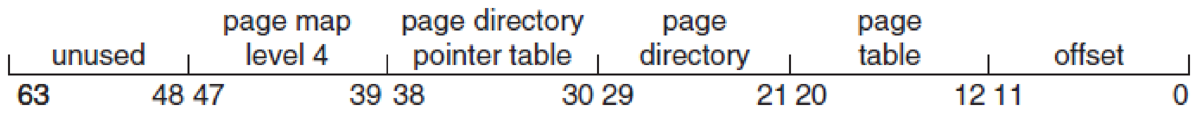
Address-translation scheme for a two-level 32-bit paging architecture



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

## Understanding the Paging in the Intel architectures

The x86-64 architecture currently provides a 48-bit virtual address with support for page sizes of 4 KB, 2 MB, or 1 GB using four levels of paging hierarchy. The representation of the linear address appears in Figure. Virtual addresses are 48 bits in size but support 52-bit physical addresses (4096 terabytes).



## Understanding the Paging with respect to ARM

Apple has licensed the ARM design for its iPhone and iPad mobile devices, and several Android-based smartphones use ARM processors as well.

The 32-bit ARM architecture supports the following page sizes
1. 4-KB and 16-KB pages
2. 1-MB and 16-MB pages (termed sections)

One-level paging is used for 1-MB and 16-MB sections; two-level paging is used for 4-KB and 16-KB pages. Address translation with the ARM MMU is shown in Figure

The ARM architecture also supports two levels of TLBs. At the outer level are two micro TLBs—a separate TLB for data and another for instructions. The micro TLB supports ASIDs as well. At the inner level is a single main TLB. Address translation begins at the micro TLB level. In the case of a miss, the main TLB is then checked. If both TLBs yield misses, a page table walk must be performed in hardware.

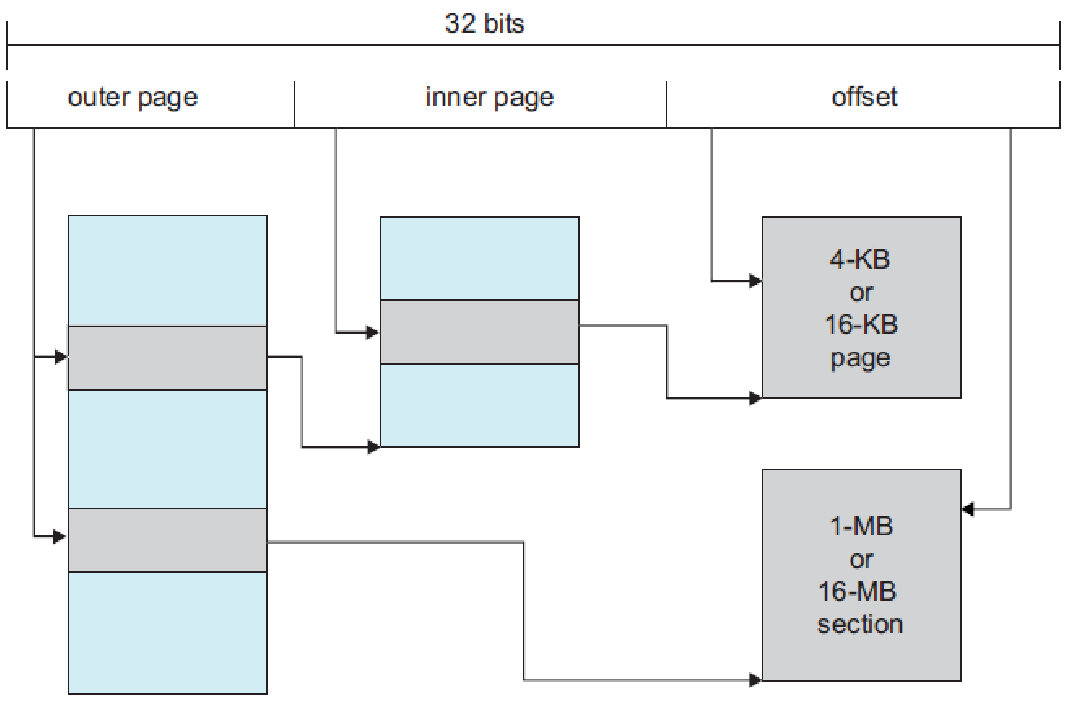


Figure: Logical address translation in ARM