Syllubus:

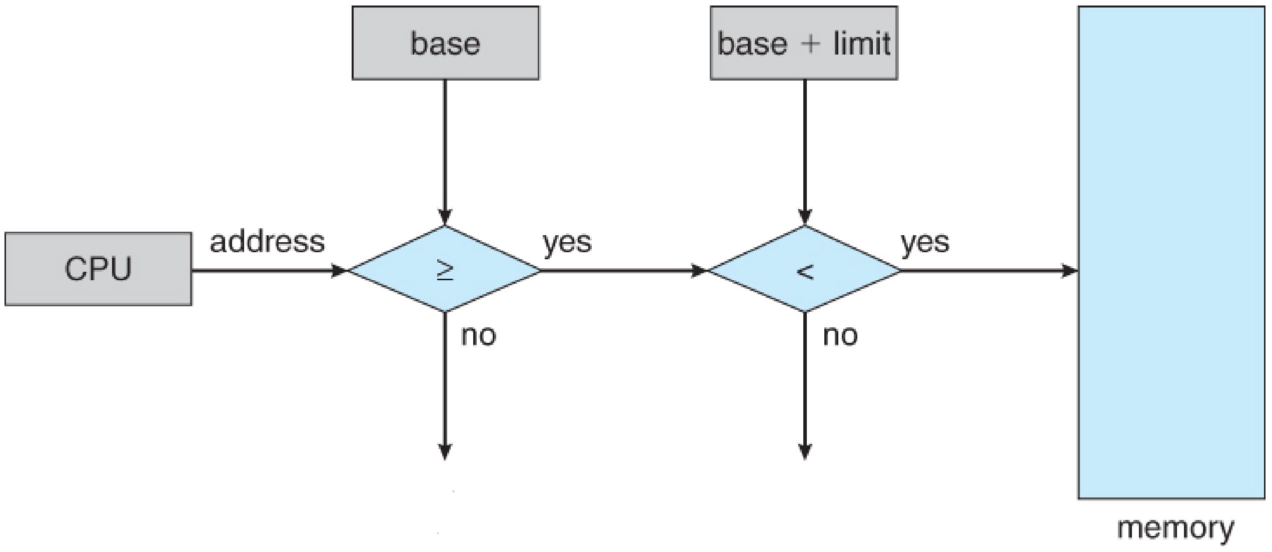
Memory Management: Swapping, contiguous memory allocation, paging, structure of the page table, segmentation

Virtual Memory Management: virtual memory, demand paging, page-Replacement algorithms, Thrashing

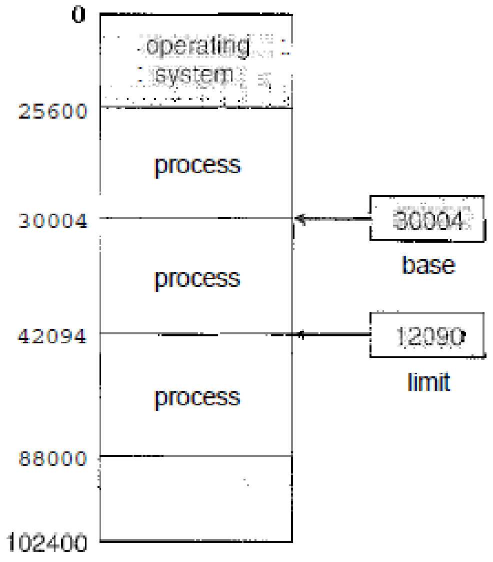
Memory Management : In a uniprogramming system, main memory is divided into two parts: one part for the operating system (resident monitor, kernel) and one part for the program currently being executed. In a multiprogramming system, the "user" part of memory must be further subdivided to accommodate multiple processes. The task of subdivision is carried out dynamically by the operating system and is known as **memory management**.

One impartant function of memory management is protection. This can be done by using Base and Limit registers. The base register holds the smallest legal physical memory address; the **limit register** specifies the size of the range.

Any attempt by a program executing in user mode to access operating-system memory or other users' memory results trap to the operating system, which treats the attempt as a fatal error which is shown in the below diagram.



Hardware address protection with base and limit registers.



A base and a limit register define a logical address space.

Address Binding: Address Binding is a process of mapping Logical address or virtual address to its corresponding physical address in main memory. The binding of instructions and data to memory addresses can be done at any step along the way:

Compile Time: If you know at compile time where the process will reside in memory, then **absolute code** can be generated. For example, if you know that a user process will reside starting at location *R*, then the generated compiler code will start at that location and extend up from there.

Load **time:** If it is not known at compile time where the process will reside in memory, then the compiler must generate **relocatable code. In** this case, final binding is delayed until load time.

Execution time. If the process can be moved during its execution from one memory segment to another, then binding must be delayed until run time.

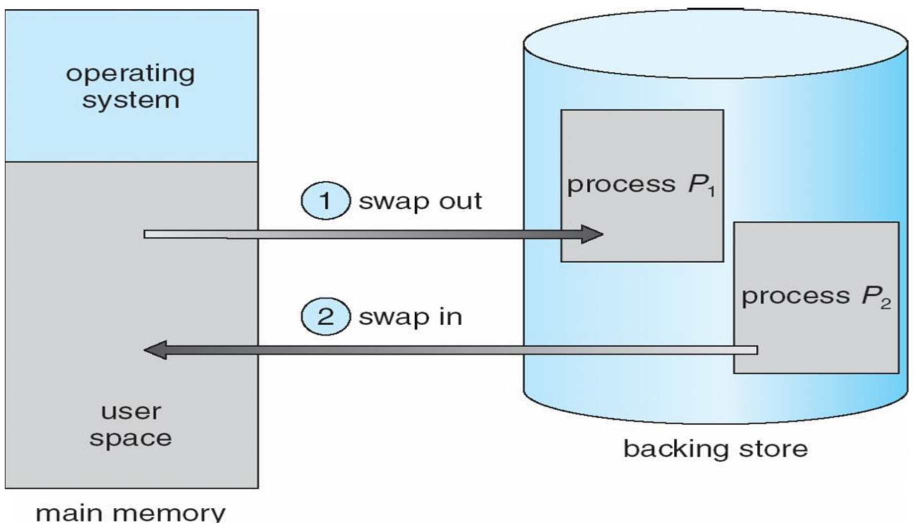
Logical Versus Physical Address Space: An address generated by the CPU is commonly referred to as a **logical address,** whereas an address seen by the memory unit—that is, the one loaded into the **memory-address register** of the memory—is commonly referred to as a **physical address.**

The compile-time and load-time address-binding methods generate identical logical and physical addresses. However, the execution-time addressbinding scheme results in differing logical and physical addresses. The set of all logical addresses generated by a program is a **logical** address **space;** the set of all physical addresses corresponding to these logical addresses is a **physical** address **space.**

The run-time mapping from virtual to physical addresses is done by a hardware device called the **memory-management unit (MMU).**

Swapping: A process must be in memory to be executed. A process, however, can be swapped temporarily out of memory to a **backing store** and then brought back into memory for continued execution. For example, assume a multiprogramming environment with a round-robin CPU-scheduling algorithm. When a quantum expires, the memory manager will start to swap out the process that just finished and to swap another process into the memory space that has been freed. In the meantime, the CPU scheduler will allocate a time slice to some other process in memory.

A variant of this swapping policy is used for priority-based scheduling algorithms. If a higher-priority process arrives and wants service, the memory manager can swap out the lower-priority process and then load and execute the higher-priority process. When the higher-priority process finishes, the lower-priority process can be swapped back in and continued. This variant
of swapping is sometimes called **roll out, roll in.**



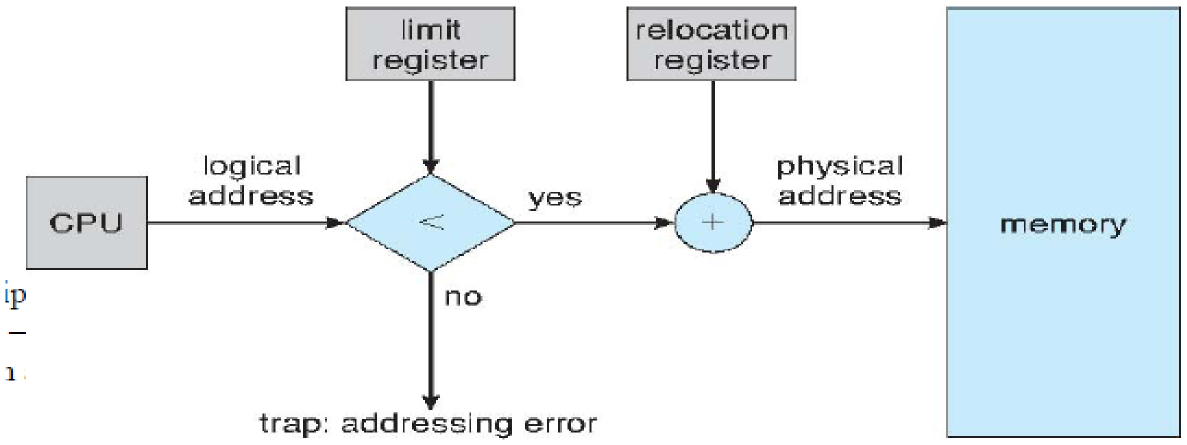
Normally, a process that is swapped out will be swapped back into the same memory space it occupied previously. This depends on the method of address binding. If binding is done at assembly or load time, then the process cannot be easily moved to a different location. If execution-time binding is being used, however, then a process can be swapped into a different memory space.

Swapping requires a backing store. The backing store is commonly a fast disk. It must be large enough to accommodate copies of all memory images for all users. The system maintains a ready queue consisting of all processes whose memory images are on the backing store or in memory and are ready to run. Whenever the CPU scheduler decides to execute a process, it calls the dispatcher. The dispatcher checks to see whether the next process in the queue is in memory. If it is not, and if there is no free memory region, the dispatcher swaps out a process currently in memory and swaps in the desired process. It then reloads registers and transfers control to the selected process.

The context-switch time in such a swapping system is fairly high. If we want to swap a process, we must be sure that it is completely idle.

Contiguous Memory Allocation: The memory is usually divided into two partitions: one for the resident operating system and one for the user processes. The operating system may be in either low memory or high memory depends on location of interrupt vector.

Memory Mapping and Protection: Mapping is the process of converting Logical address into Physical address. This can be done by using Base or relocation register and limit registers. Base register contains the value of the smallest physical address; the limit register contains the range of logical addresses. With relocation and limit registers, each logical address must be less than the limit register; the VIMU maps the logical address *dynamically* by adding the value in the relocation register. This mapped address is sent to memory.



Hardware support for relocation and limit registers.

When the CPU scheduler selects a process for execution, the dispatcher loads the relocation and limit registers with the correct values as part of the context switch. Because every address generated by the CPU is checked against these registers, we can protect both the operating system and the other users programs and data from being modified by this running process.

Memory Allocation: One of the simplest methods for allocating memory is to divide memory into several fixed-sized partitions. Each partition may contain exactly one process. Thus, the degree of multiprogramming is bound by the number of partitions. In this multiplepartition method, when a partition is free, a process is selected from the input queue and is loaded into the free partition. When the process terminates, the partition becomes available for another process. This method was originally used by the IBM OS/360 operating system. The following are two different partitioning methods:

1. **Fixed partitioning**
2. **Dynamic sized partitioning (or) Variable sized Partitioning**

1. **Fixed partitioning:** In this, Single contiguous memory location is a simple memory management scheme that requires no special hardware features. For allocating memory, it is divided into number of fixed sized partitions. Each partition contains exactly one process. If the partition is free, process is selected from the input queue and is loaded into the free partition of memory. When the process terminates, the memory partition becomes available for another process. Batch operating systems uses this partition scheme. The operating system maintains the record of allocating and deallocating memory to processes.

   When the process arrives in the systemand needs memory, operating system search large enough space for this process. If available, use it. Otherwise, wait.

   The following diagrams shows the examples of two alternatives for fixed partitioning: one possibility is to make use of equal size partitions.

8MB
8MB
8MB
8MB
8MB
8MB
8MB
8MB

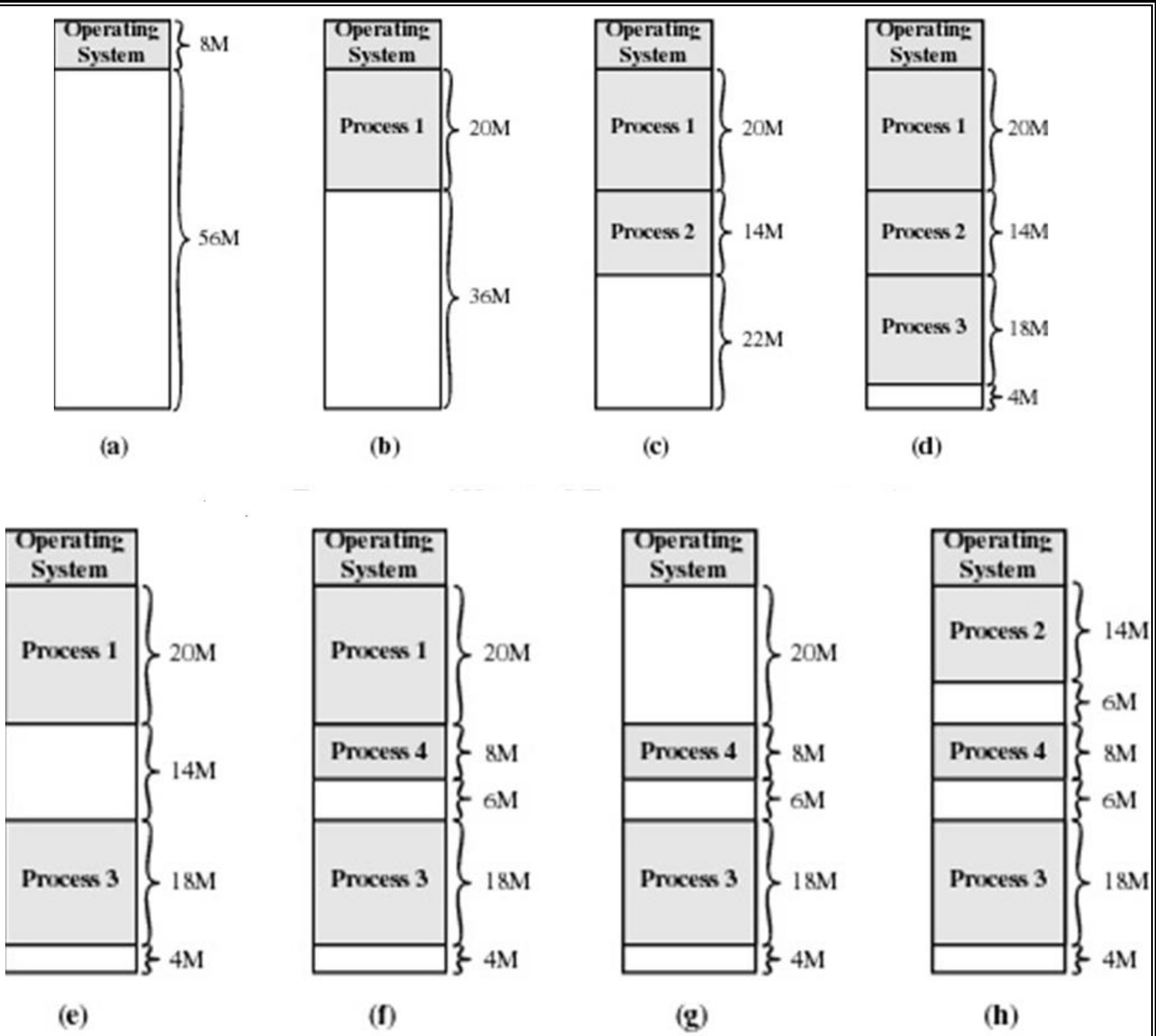
Equal Size Partitions

8MB
6MB
5MB
3MB
4MB
4MB
6MB
12MB
16MB

Unequal size partitions

Any process whose size is lessthan or equal to the partition size can be loaded into any available partition.
If all memory partitions are full and no process is in the ready or running state, the operating system can swap a process out of any of the partitions and load in another process.
Therer are two difficulties in with the use of equal size fixed partitions:
→ A program may be too big to fit into a partition.
→ Main memory utilization is extremely inefficient.
Advantages and disadvantages of Fixed Partition size:
Advantages:
→ Simple to implement
→ Less overhead
Disadvantages:
→ Memory is not efficiently used
→ Poor utilization of Processors
→ User's process is limited to the size of available main memory

2. **Dynamic sized partitioning:** In this technique, Memory Prtitions are of variable length. When process is brought into memory, it is allocated exactly as much memory as it requires and no more. Initially, main memory is empty, except for the operating system as shown below.
   The first three processes are loaded in, starting where the operating system ends and occupying just enough space for each process. This leaves a "hole" at end of memory that is too small for a fourth process.
   The operating system swaps out process2 , which leaves sufficient room to load a new process, process 4. Since, process 4 is smaller than process 2,a nother small hole is created. At certain point of time, none of the processes in mamn memory is ready, but process 2 is the ready suspend state, is available. Because there is insufficient space in memory for process 2, the operating system swaps out process 1 and swaps in process 2 for exectution.



The Effect of Dynamic Partitioning

The **first-fit, best-fit,** and **worst-fit** strategies are the ones most commonly used to select a free hole from the set of available holes.

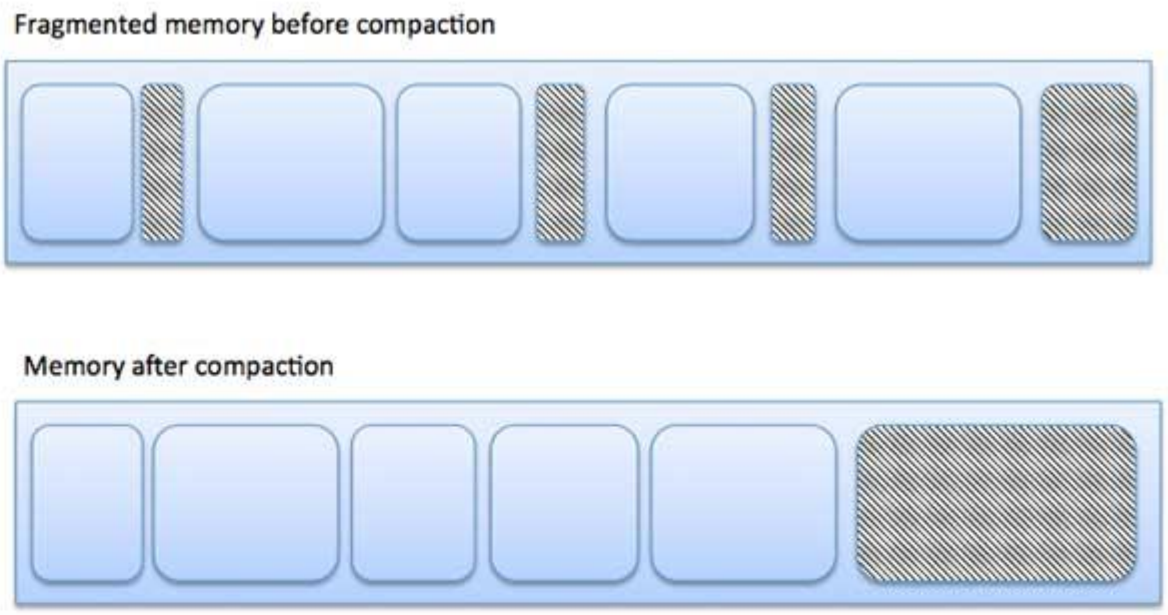
- First fit: Allocate the *first* hole that is big enough. Searching can start either at the beginning of the set of holes or where the previous first-fit search ended.
- Best fit: Allocate the *smallest* hole that is big enough. Search the entire list, unless the list is ordered by size. This strategy produces the smallest leftover hole.
- Worst fit: Allocate the *largest* hole. Again, search the entire list, unless it is sorted by size. This strategy produces the largest leftover hole, which may be more useful than the smaller leftover hole from a best-fit approach.

Fragmentation: As processes are loaded and removed from memory, the free memory space is broken into little pieces. It happens after sometimes that processes cannot be allocated to memory blocks considering their small size and memory blocks remains unused. This problem is known as Fragmentation. Fragmentation is of two types –

- **Internal Fragmentation:** Unused memory which is internal to a prtition is called as Internal Fragmentation. The internal fragmentation can be reduced by effectively assigning the smallest partition but large enough for the process.
- **External Fragmentation:** External fragmentation exists when there is enough total memory space to satisfy a request, but the available spaces are not contiguous. One solution to the problem of external

fragmentation is **Compaction**. **Compaction** is the process of shuffling the memory contents so as to place all free memory together in one large block. Compaction is not always possible, however. If relocation is static and is done at assembly or load time, compaction cannot be done; compaction is possible *only* if relocation is dynamic and is done at execution time.

The following diagram shows how fragmentation can cause waste of memory and a compaction technique can be used to create more free memory out of fragmented memory –



Another possible solution to the external-fragmentation problem is to permit the logical address space of the processes to be noncontiguous, thus allowing a process to be allocated physical memory wherever the latter is available. Two complementary techniques achieve this solution: paging and Segmentation.

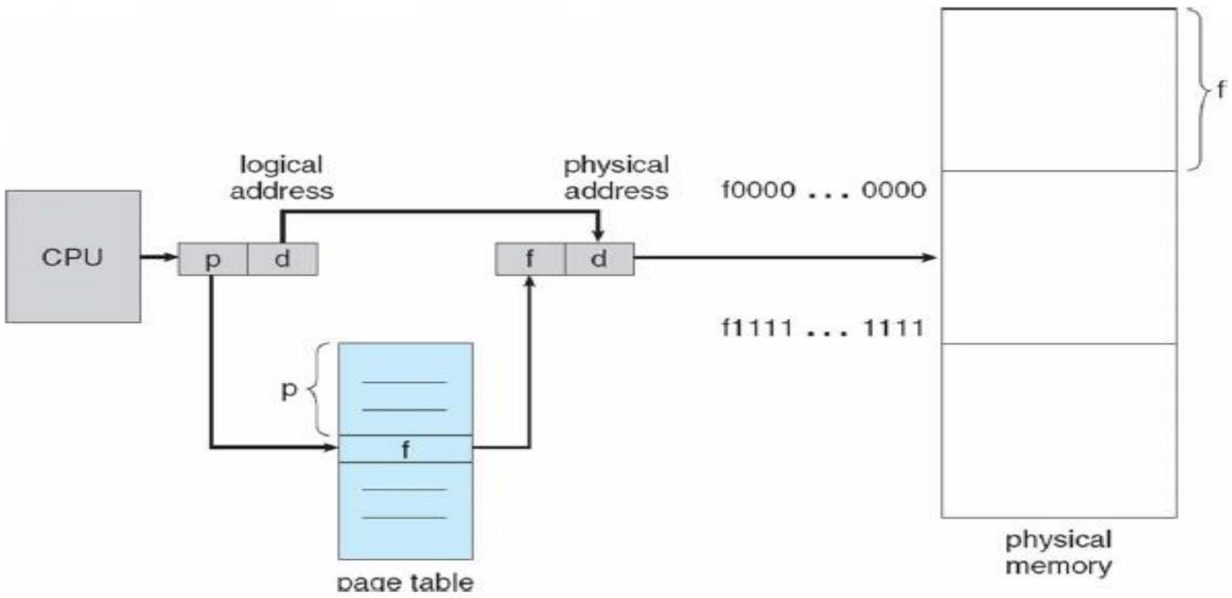
Differences between Internal and External Fragmentation:

S.No	Internal Fragmentation	External Fragmentation
1	Memory allocated to a process may be slightly larger than the requested memory.	External fragmentation exists when there is enough total memory space to satisfy a request, but the available spaces are not contiguous.
2	First fit and best fit memory allocation does not suffer from internal fragmentation.	First fit and best fit memory allocation suffers from internal fragmentation.
3	Systems with fixed sized allocation units, such as the single sized partition scheme and paging suffer from internal fragmentation.	Systems with variable sized allocation units, such as the multiple partition scheme and segmentation suffer from internal fragmentation.

Paging: Paging is a memory-management scheme that permits the physical address. Space of a process to be noncontiguous. Paging avoids external fragmentation and use of Compaction.

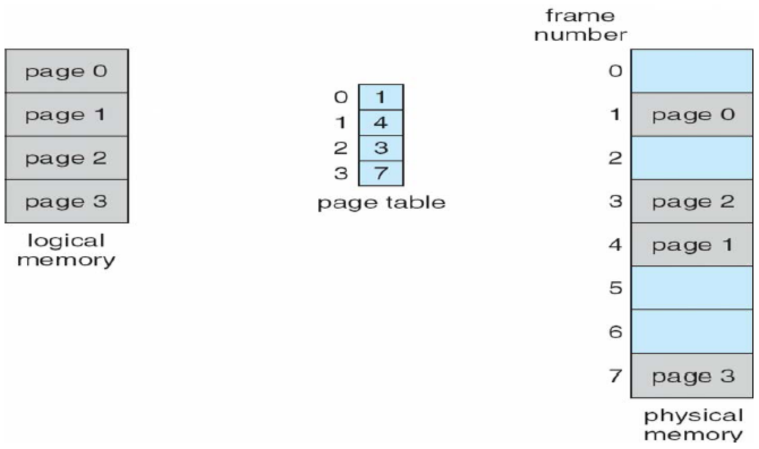
Basic Method: The basic method for implementing paging involves breaking physical memory into fixed-sized blocks called **frames** and breaking logical memory into blocks of the same size called **pages.** When a process is to be executed, its pages are loaded into any available memory frames from the backing store. The backing store is divided into fixed-sized blocks that are of the same size as the memory frames.

The hardware support for paging is illustrated in the following figure.



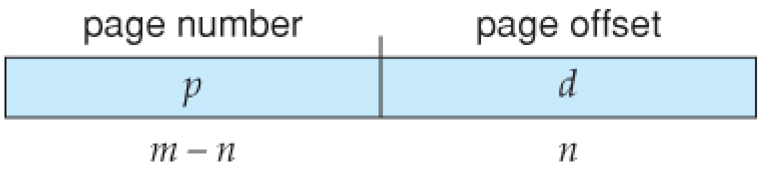
Paging hardware.

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



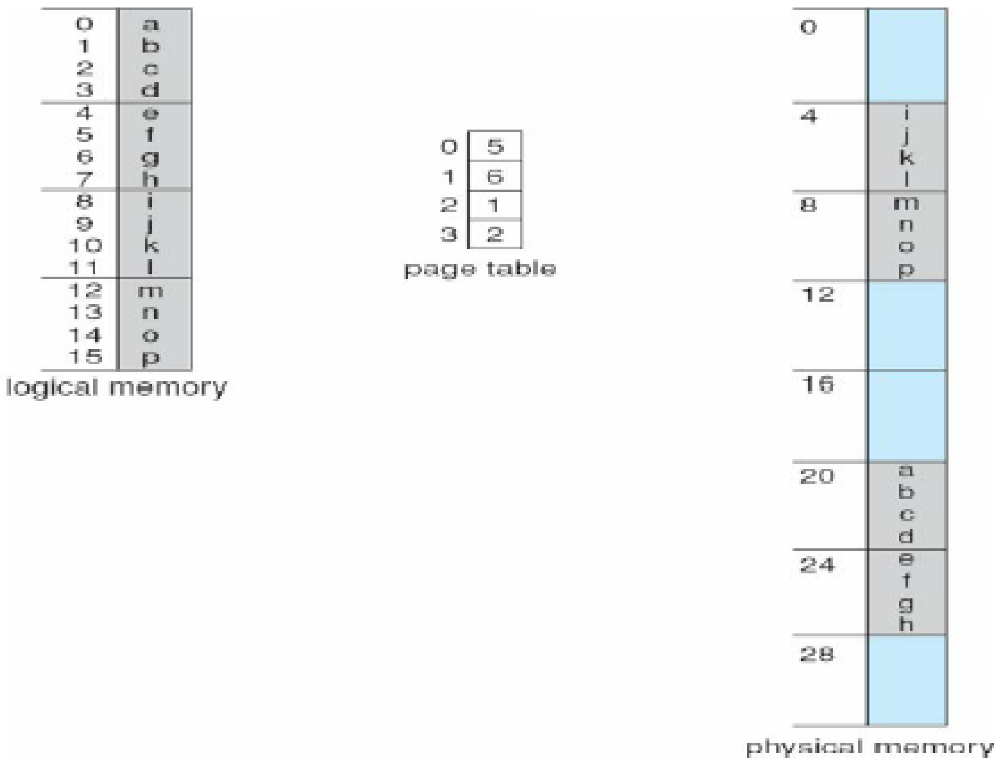
Paging model of logical and physical memory.

Paging model of logical and physical memory. The page size (like the frame size) is defined by the hardware. The size of a page is typically a power of 2, varying between 512 bytes and 16 MB per page, depending on the computer architecture. The selection of a power of 2 as a page size makes the translation of a logical address into a page number and page offset particularly easy. If the size of logical address space is 2^m and a page size is 2^n addressing units (bytes or words), then the high-order $m – n$ bits of a logical address designate the page number, and the n low-order bits designate the page offset. Thus, the logical address is as follows:



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

For example, consider the memory in the following figure. Using a page size of 4 bytes and a physical memory of 32 bytes (8 pages)..



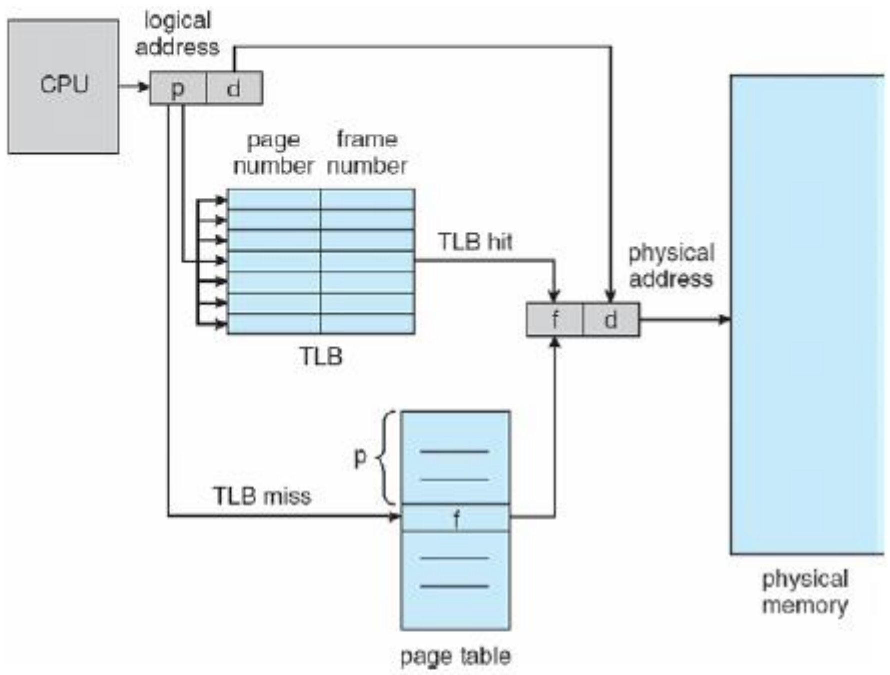
Paging example for a 32-byte memory with 4-byte pages.

Paging example for a 32-byte memory with 4-byte pages. Logical address 0 is page 0, offset 0. Indexing into the page table, we find that page 0 is in frame 5. Thus, logical address 0 maps to physical address 20 (= (5 x 4) + 0). Logical address 3 (page 0, offset 3) maps to physical address 23 (= (5x4) + 3). Logical address 4 is page 1, offset 0; according to the page table, page 1 is mapped to frame 6. Thus, logical address 4 maps to physical address 24 (= (6x4) + 0). Logical address 13 maps to physical address 9.

Hardware Support: As size of the page table increases, memory access time increases. Such that performance of a computer decreases. To overcome this problem, use a special, small, fastlookup hardware cache, called a translation look-aside buffer (TLB). The TLB is associative, high-speed memory. Each entry in the TLB consists of two parts: a key (or tag) and a value. When the associative memory is presented with an item, the item is compared with all keys simultaneously. If the item is found, the corresponding value field is returned.

The TLB is used with page tables in the following way. The TLB contains only a few of the page-table entries. When a logical address is generated by the CPU, its page number is presented to the TLB. If the page number is found(TLB Hit), its frame number is immediately available and is used to access memory.

If the page number is not in the TLB (known as a TLB miss), a memory reference to the page table must be made. When the frame number is obtained, use it to access memory. In addition, add the page number and frame number to the TLB, so that they will be found quickly on the next reference. If the TLB is already full of entries, the operating system must select one for replacement.



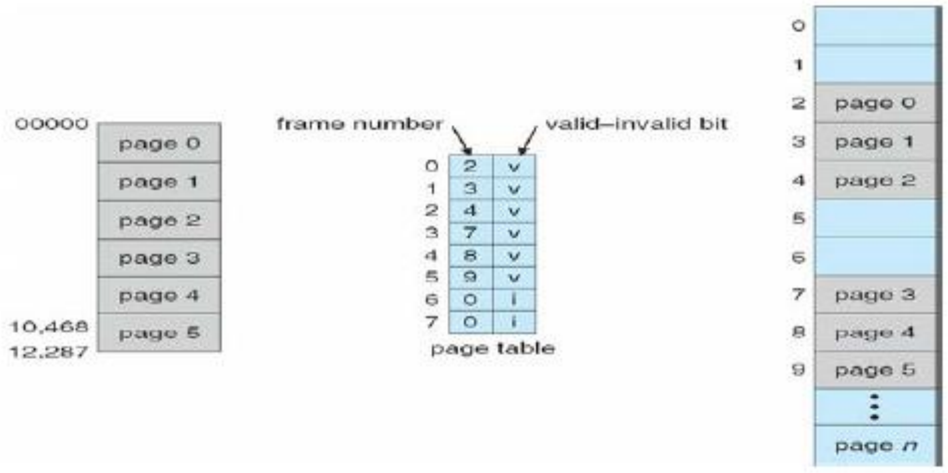
Paging hardware with TLB.

The percentage of times that a particular page number is found in the TLB is called the **hit ratio.** An 80-percent hit ratio means that we find the desired page number in the TLB 80 percent of the time. If it takes 20 nanoseconds to search the TLB and 100 nanoseconds to access memory, then a mapped-memory access takes 120 nanoseconds when the page number is in the TLB. If we fail to find the page number in the TLB (20 nanoseconds), then we must first access memory for the page table and frame number (100 nanoseconds) and then access the desired byte in memory (100 nanoseconds), for a total of 220 nanoseconds. To find the **effective memory-access time,** we weight each case by its probability:

effective access time = 0.80 x 120 + 0.20 x 220

= 140 nanoseconds.

Protection: Memory protection in a paged environment is accomplished by protection bits associated with each frame. Normally, these bits are kept in the page table. One additional bit is generally attached to each entry in the page table: a **valid-invalid** bit. When this bit is set to "valid," the associated page is in the process's logical address space and is thus a legal (or valid) page. When the bit is set to"invalid,"' the page is not in the process's logical address space. The operating system
sets this bit for each page to allow or disallow access to the page.

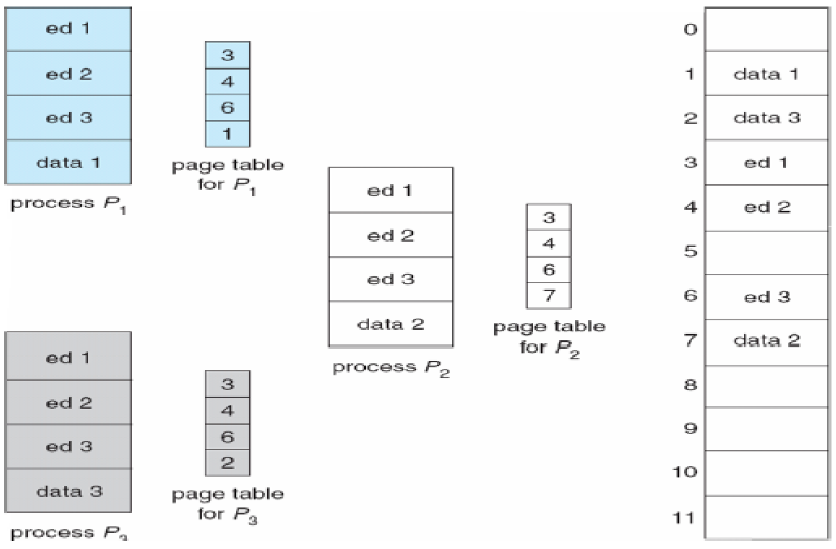


Valid (v) or invalid (i) bit in a page table.

Here, Addresses in pages 0,1, 2,3, 4, and 5 are mapped normally through the page table. Any attempt to generate an address in pages 6 or 7, however, will find that the valid-invalid bit is set to invalid, and the computer will trap to the operating system.

Shared Pages: An advantage of paging is the possibility of *sharing* common code. This consideration is particularly important in a time-sharing environment. Consider a system that supports 40 users, each of whom executes a text editor. If the text editor consists of 150 KB of code and 50 KB of data space, we need 8,000 KB to support the 40 users.

In this case, Only one copy of the editor need be kept in physical memory. Each user's page table maps onto the same physical copy of the editor, but data pages are mapped onto different frames. Thus, to support 40 users, we need only one copy of the editor (150 KB), plus 40 copies of the 50 KB of data space per user. The total space required is now 2,150 KB instead of 8,000 KB which is shown in the figure below.

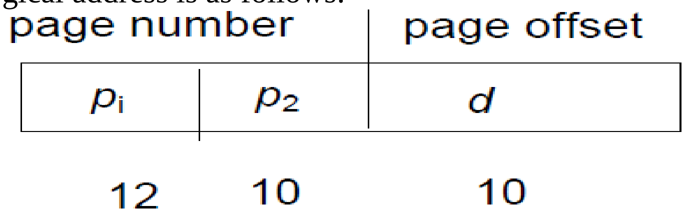


Sharing of code in a paging environment.

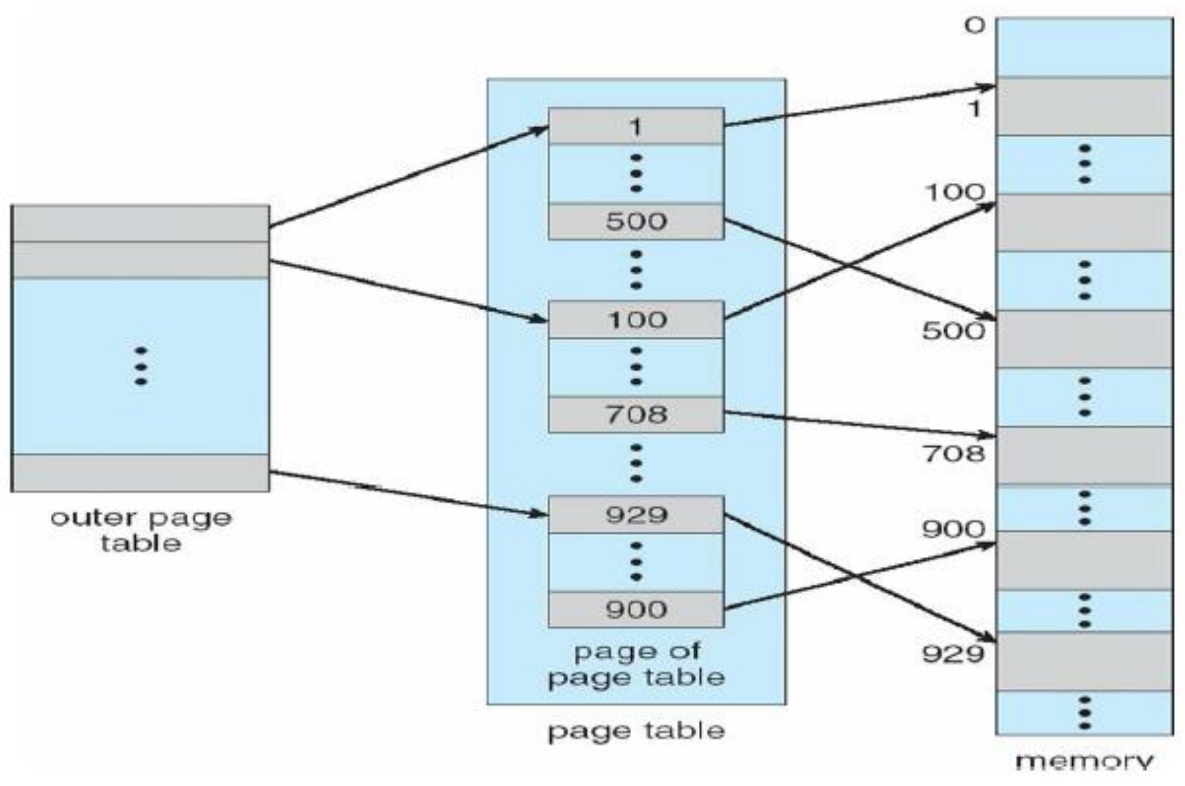
Structure of the Page Table:

Some of the most common techniques for structuring the page table are : Hierarchical Paging, Hashed page tables, and Inverted page tables.

- **Hierarchical Paging:** In this, page table will be divided into smaller pieces. Example for Hierarchical Paging is two-level paging algorithm, in which the page table itself is also paged. Consider a 32 bit machine with a page size of 4 KB. A logical address is divided into a page number consisting of 20 bits and a page offset consisting of 12 bits. Because we page the page table, the page number is further divided into a 10-bit page number and a 10-bit page offset. Thus, a logical address is as follows:



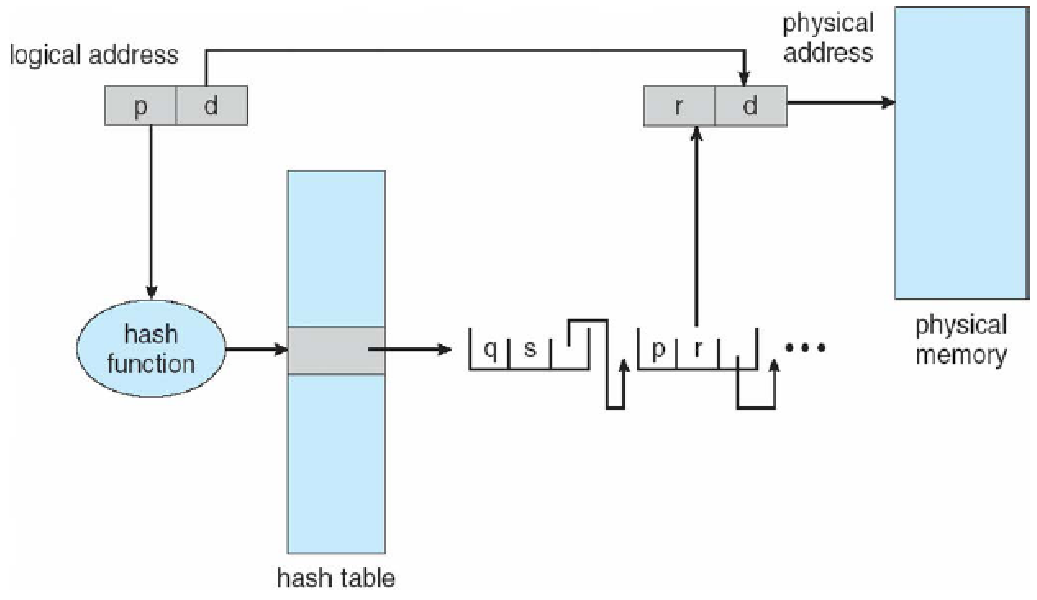
Where P_1 is an index into the outer page table and *p2* is the displacement within the page of the outer page table.



A two-level page-table scheme

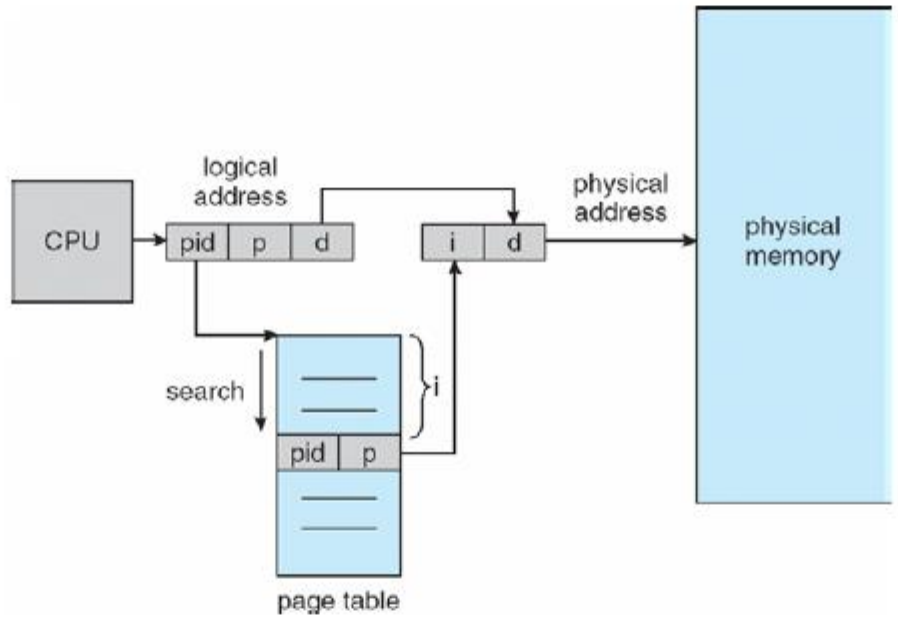
- **Hashed page tables:** A common approach for handling address spaces larger than 32 bits is to use a **hashed** page table, with the hash value being the virtual page number. Each entry in the hash table contains a linked list of elements that hash to the same location (to handle collisions). Each element consists of three fields: (1) the virtual page number, (2) the value of the mapped page frame, and (3) a pointer to the next element in the linked list.

  The algorithm works as follows: The virtual page number in the virtual address is hashed into the hash table. The virtual page number is compared with field 1 in the first element in the linked list. If there is a match, the corresponding page frame (field 2) is used to form the desired physical address. If there is no match, subsequent entries in the linked list are searched for a matching virtual page number. This scheme is shown in the following figure.



Hashed page table.

- **Inverted page tables:** An inverted page table has one entry for each real page (or frame) of memory. Each entry consists of the virtual address of the page stored in that real memory location, with information about the process that owns that page. Thus, only one page table is in the system, and it has only one entry for each page of physical memory. The following diagram shows the operation of an inverted page table.



Inverted page table.

Each virtual address in the system consists of a triple

<process-id, page-number, offset>.

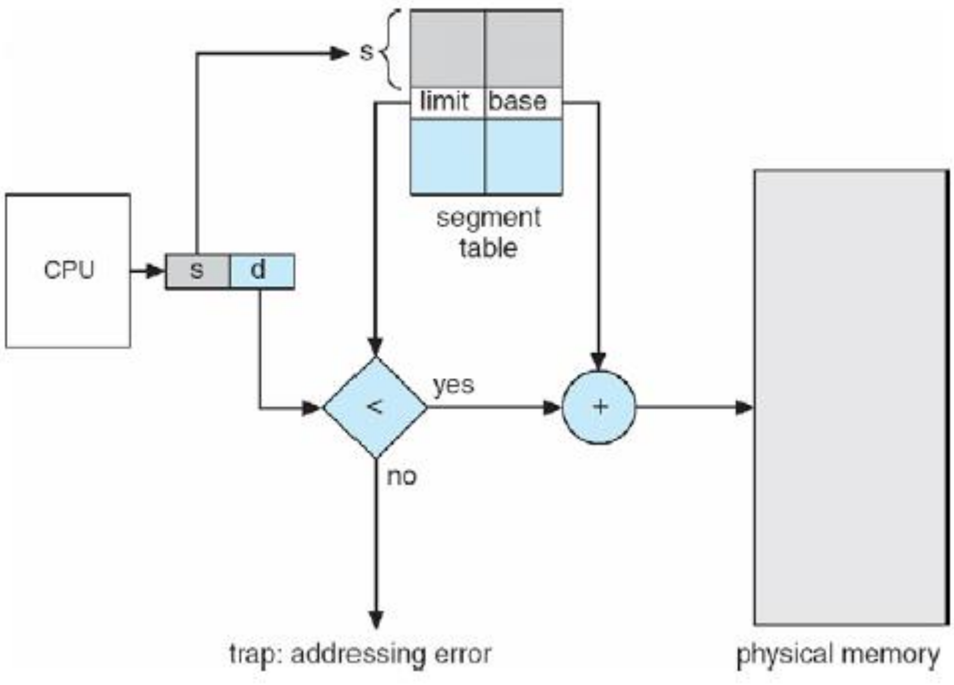
Each inverted page-table entry is a pair <process-id, page-number> where the process-id assumes the role of the address-space identifier. When a memory reference occurs, part of the virtual address, consisting of <process-id, pagenumber>, is presented to the memory subsystem. The inverted page table is then searched for a match. If a match is found—say; at entry i—then the physical address <*i*, offset> is generated. If no match is found, then an illegal address access has been attempted.

Segmentation: Segmentation is the process of dividing a program into variable sized blocks called segments. A Segment is called as a logical grouping of information such as subroutines, arrays, structues etc…

Basic Method: A logical address space is a collection of segments. Each segment has a name and a length. The addresses specify both the segment name and the offset within the segment. segments are numbered and are referred to by a segment number, rather than by a segment name. Thus, a logical address consists of a *two tuple:*

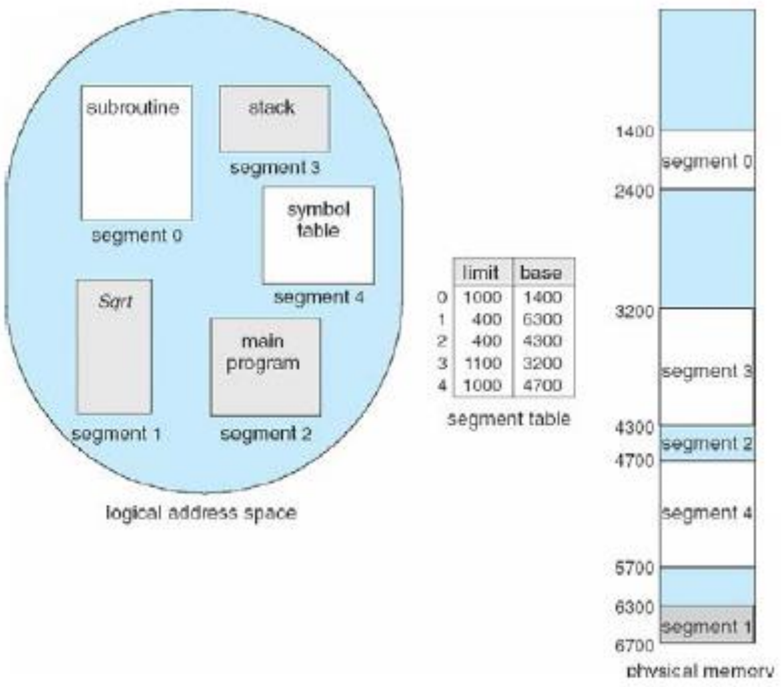
< segment-number, offset >.

Hardware: the following diagram represents segmentation hardware.



Segmentation hardware.

A logical address consists of two parts: a segment number, *s*, and an offset into that segment, *d*. The segment number is used as an index to the segment table. Each entry in the segment table has a *segment base* and a *segment limit.* The segment base contains the starting physical address where the segment resides in memory, whereas the segment limit specifies the length of the segment.
The offset *d* of the logical address must be between 0 and the segment limit. If it is not, a trap message will be sent to the operating system. When an offset is legal, it is added to the segment base to produce the address in physical memory of the desired byte.
For example, We have five segments numbered from 0 through 4. The segments are stored in physical memory as shown below.



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

Example of segmentation.

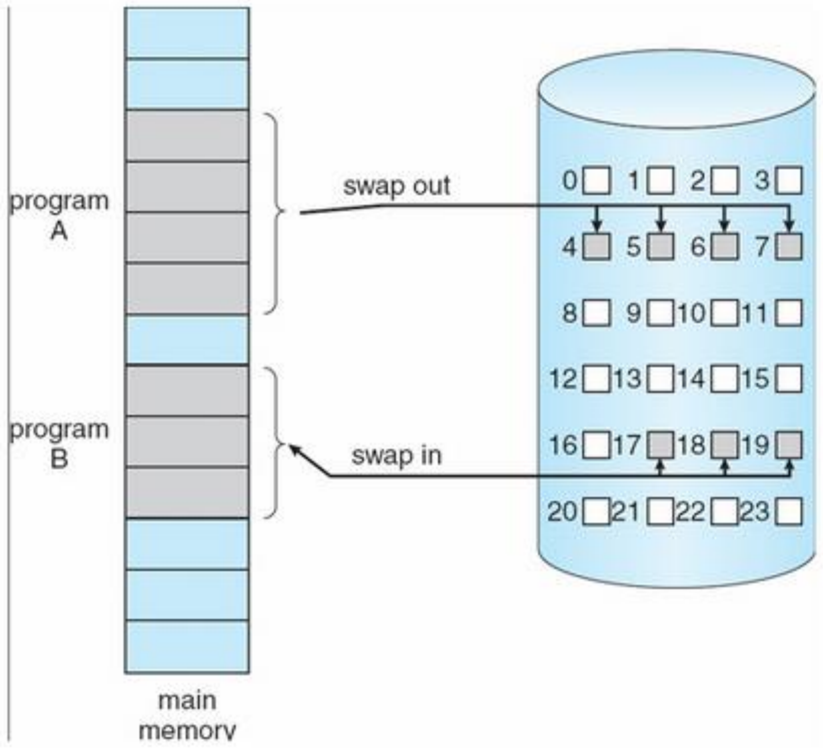
The segment table has a separate entry for each segment, giving the beginning address of the segment in physical memory (or base) and the length of that segment (or limit). For example, segment 2 is 400 bytes long and begins at location 4300. Thus, a reference to byte 53 of segment 2 is mapped onto location 4300 + 53 = 4353. A reference to segment 3r byte 852, is mapped to 3200 (the base of segment 3) + 852 = 4052. A reference to byte 1222 of segment 0 would result in a trap to the operating system, as this segment is only 1,000 bytes long.

Differences between Segmentation and Paging:

S.No	Segmentation	Paging
1	Program is divided into variable size segments.	Program is divided into fixed size pages.
2	User is responsible for dividing program.	Operating system take care of dividing program.
3	Segmentation is slower than paging.	Paging is faster than segmentation.
4	Segmentation is visible to the user	Paging is invisible to the user.
5	Segmentation eliminates internal fragmentation.	Paging suffers from internal fragmentation.
6	Segmentation suffers from external fragmentation	Paging eliminates external fragmentation.
7	Processor uses segment number, offset to calculate absolute address.	Processor uses page number, offset to calculate absolute address.

Virtual Memory: Virtual Memory allows execution of partially loaded processes. Virtual memory can be implemented by using Demand Paging.

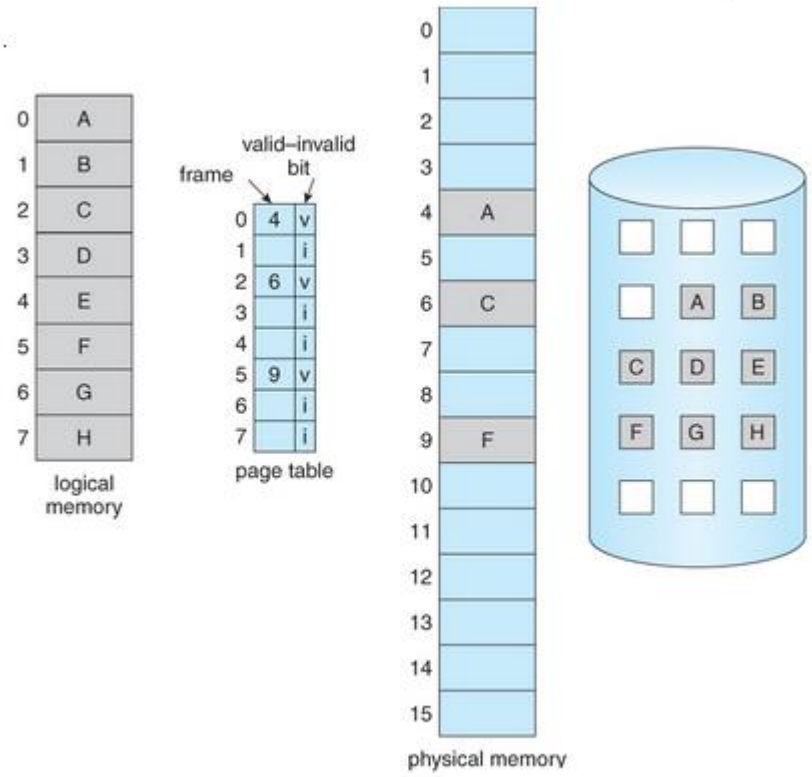
Demand Paging: In Demand Paging, pages are only loaded when they are demanded during program execution; pages that are never accessed are thus never loaded into physical memory. A demand-paging system is similar to a paging system with swapping as shown below. Here processes reside in secondary memory (usually a disk). To execute a process, swap it into memory. Rather than swapping the entire process into memory, however, use a **lazy swapper.** A lazy swapper never swaps a page into memory unless that page will be needed.



Transfer of a paged memory to contiguous disk space.

When a process is to be swapped in, the pager guesses which pages will be used before the process is swapped out again. Instead of swapping in a whole process, the pager brings only those necessary pages into memory. Thus, it avoids reading into memory pages that will not be used anyway, decreasing the swap rime and the amount of physical memory needed.

With this scheme, we need some form of hardware support to distinguish between the pages that are in memory and the pages that are on the disk. The valid-invalid bit scheme is used as shown below.



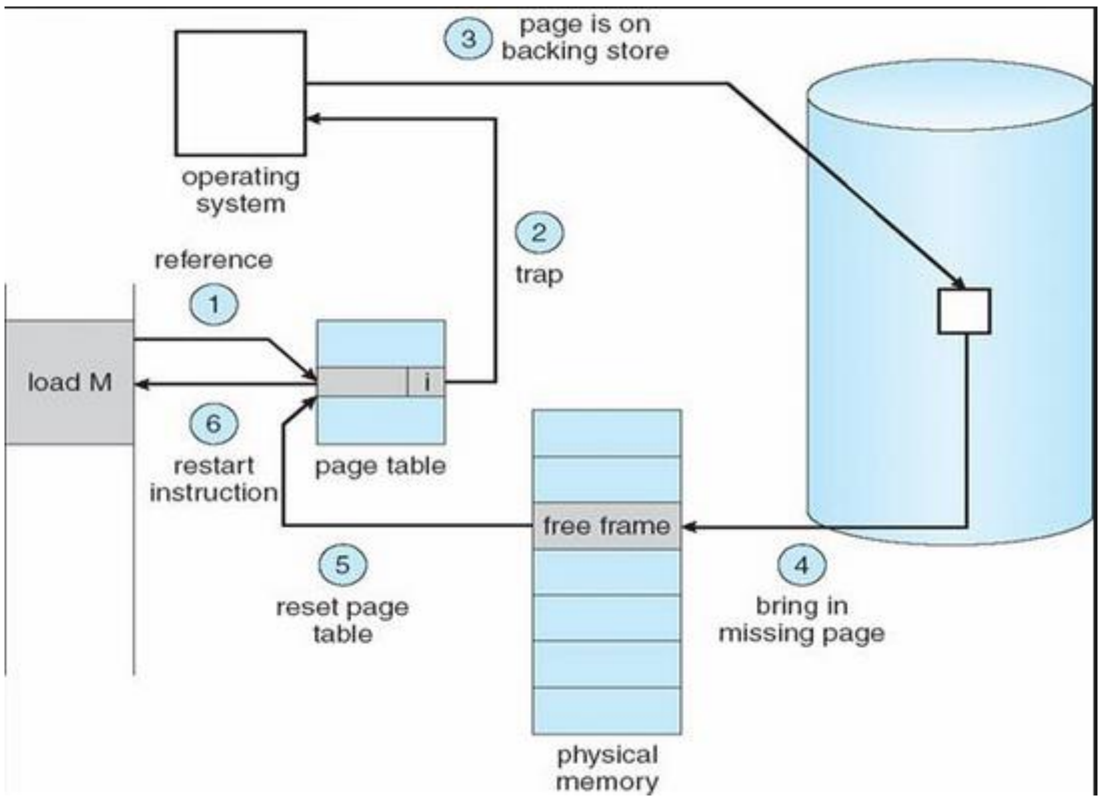
<u>Page table when some pages are not in main memory.</u>

When this bit is set to "valid/" the associated page is both legal and in memory. If the bit is set to "invalid," the page either is not valid (that is, not in the logical address space of the process) or is valid but is currently on the disk.

Access to a page marked invalid causes a **page-fault trap. T**his trap is the result of the operating system's failure to bring the desired page into memory. The procedure for handling this page fault is as shown below.

- First check whether the reference was a valid or an invalid memory access.
- If the reference was invalid, terminate the process. If it was valid, but not yet brought in that page, now page it in.
- Find a free frame
- Schedule a disk operation to read the desired page into the newly allocated frame.
- When the disk read is complete, modify the internal table kept with the process and the page table to indicate that the page is now in memory.
- Restart the instruction that was interrupted by the trap. The process can now access the page as though it had always been in memory.

The following diagram shows steps for handling Page fault.



Steps in handling a page fault.

A crucial requirement for demand paging is the need to be able to restart any instruction after a page fault. A page fault may occur at any memory reference. If the page fault occurs on the instruction fetch, restart byfetching the instruction again. If a page fault occurs during fetching an operand, fetch and decode the instruction again and then fetch the operand.

Performance of Demand Paging: Demand paging can significantly affect the performance of a computer system. This can be calculated in terms of effective access time.

Let *p* be the probability of a page fault ($0 \le p \le 1$). The effective access time is then

effective access time = $(1 - p) * ma + p *$ page fault time.

Where, ma is memory access time.

Pafefault time is the time waited by CPU due to page fault.

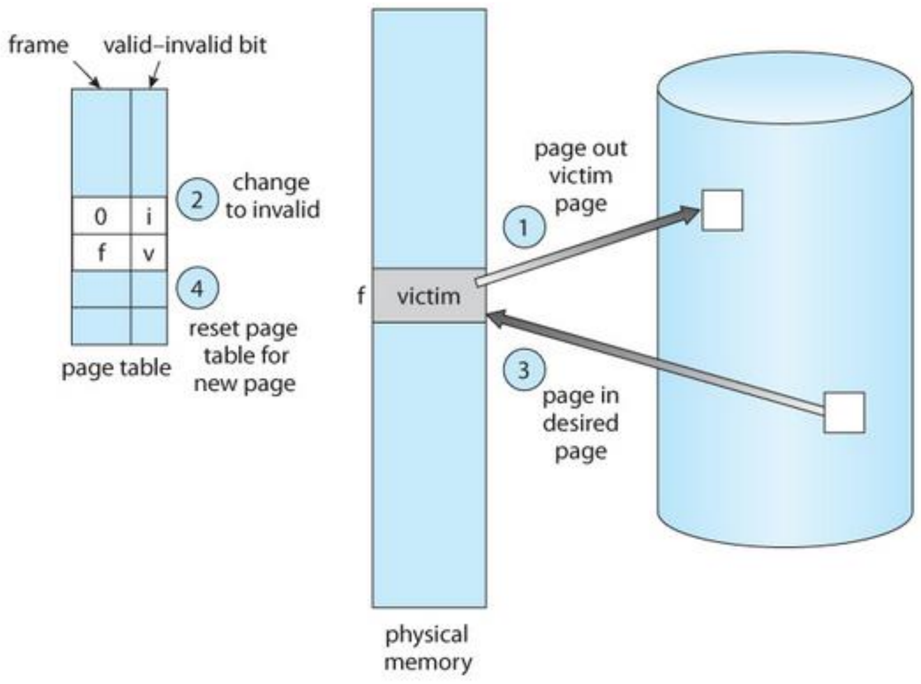
Effective access time is directly promotional to the page fault rate. It is important to keep the page fault rate low in a demand paging system. Otherwise, the effective access time increases, skowing process execution dramatically.

Advantages of Demand Paging:

1. Large virtual memory
2. More efficient use of memory
3. There is no limit on degree of multiprogramming.

Page Replacement Algorithms: Page replacement policy deals with the selection of a page in memory to be replaced when a new page must be brought in. When a page fault occurs, operating system performs the following.

- Find the location of the desired page on the disk.
- Find a free frame:
  - If there is a free frame, use it.
  - If there is no free frame, use a page-replacement algorithm toselect a victim frame.
  - Write the victim frame to the disk; change the page and frame tables accordingly.
- Read the desired page into the newly freed frame; change the page and frame tables.
- Restart the user process.



Page replacement

The following are different types of Page Replacement algorithms used to select victim page.

1. **FIFO Page Replacement:** The simplest page-replacement algorithm is a first-in, first-out (FIFO) algorithm. A FIFO replacement algorithm associates with each page the time when that page was brought into memory. When a page must be replaced, the oldest page is chosen. This technique, creates a FIFO queue to hold all pages in memory. We replace the page at the head of the queue. When a page is brought into memory, we insert it at the tail of the queue.
For example, consider the following reference string with 3 frames.

7, 0,1, 2, 0, 3, 0, 4, 2, 3, 0, 3, 2,1, 2, 0, 1, 7, 0,1

Initial	E E E	---------
After inserting 7	7 E E	Page Fault
After inserting 0	7 0 E	Page Fault
After inserting 1	7 0 1	Page Fault
After inserting 2	2 0 1	Page Fault
After inserting 0	2 0 1	--------
After inserting 3	2 3 1	Page Fault
After inserting 0	2 3 0	Page Fault
After inserting 4	4 3 0	Page Fault
After inserting 2	4 2 0	Page Fault
After inserting 3	4 2 3	Page Fault
After inserting 0	0 2 3	Page Fault
After inserting 3	0 2 3	------------
After inserting 2	0 2 3	------------
After inserting 1	0 1 3	Page Fault
After inserting 2	0 1 2	Page Fault
After inserting 0	0 1 2	------------
After inserting 1	0 1 2	------------
After inserting 7	7 1 2	Page Fault
After inserting 0	7 0 2	Page Fault
After inserting 1	7 0 1	Page Fault

Total number of page faults: 15

The first three references (7,0,1) cause page faults and are brought into these empty frames. The next reference (2) replaces page 7, because page 7 was brought in first. Since 0 is the next reference and 0 is already in memory, we have no fault for this reference. The first reference to 3 results in replacement of page 0, since it is now first in line. Because of this replacement, the next reference, to 0, will fault. Page 1 is then replaced by page 0. This process continues till last page.

2. **LRU Page Replacement**: The key distinction between the FIFO and OPT algorithms (other than looking backward versus forward in time) is that the FIFO algorithm uses the time when a page was brought into memory, whereas the OPT algorithm uses the time when a page is to be *used.* LRU replacement associates with each page the time of that page's last use. When a page must be replaced, LRU chooses the page that has not been used for the longest period of time.
For example, consider the following reference string with 3 frames.

7, 0,1, 2, 0, 3, 0, 4, 2, 3, 0, 3, 2,1, 2, 0, 1, 7, 0,1

Initial	E E E	---------
After inserting 7	7 E E	Page Fault
After inserting 0	7 0 E	Page Fault
After inserting 1	7 0 1	Page Fault
After inserting 2	2 0 1	Page Fault
After inserting 0	2 0 1	-----------
After inserting 3	2 0 3	Page Fault
After inserting 0	2 0 3	-----------
After inserting 4	4 0 3	Page Fault
After inserting 2	4 0 2	Page Fault
After inserting 3	4 3 2	Page Fault
After inserting 0	0 3 2	Page Fault
After inserting 3	0 3 2	------------
After inserting 2	0 3 2	------------
After inserting 1	1 3 2	Page Fault
After inserting 2	1 3 2	
After inserting 0	1 0 2	Page Fault
After inserting 1	1 0 2	------------
After inserting 7	1 0 7	Page Fault
After inserting 0	1 0 7	------------
After inserting 1	1 0 7	------------

Total number of page faults: 12

The LRU algorithm produces 12 faults. Notice that the first 5 faults are the same as those for optimal replacement. When the reference to page 4 occurs, however, LRU replacement sees that, of the three frames in memory, page 2 was used least recently. Thus, the LRU algorithm replaces page 2, not knowing that page 2 is about to be used. When it then faults for page *2*, the LRU algorithm replaces page 3, since it is now the least recently used of the three pages in memory. Despite these problems, LRU replacement with 12 faults is much better than FIFO replacement with 15.

3. **Optimal Page Replacement:** An optimal page-replacement algorithm has the lowest page-fault rate of all algorithms. In this technique, Replace the page that will not be used for the longest period of time. Unfortunately, the optimal page-replacement algorithm is difficult to implement, because it requires future knowledge of the reference string.
For example, consider the following reference string with 3 frames.
7, 0,1, 2, 0, 3, 0, 4, 2, 3, 0, 3, 2,1, 2, 0, 1, 7, 0,1

Initial	E E E	---------
After inserting 7	7 E E	Page Fault
After inserting 0	7 0 E	Page Fault
After inserting 1	7 0 1	Page Fault
After inserting 2	2 0 1	Page Fault

After inserting 0	2 0 1	-----------
After inserting 3	2 0 3	Page Fault
After inserting 0	2 0 3	-----------
After inserting 4	2 4 3	Page Fault
After inserting 2	2 4 3	------------
After inserting 3	2 4 3	------------
After inserting 0	2 0 3	Page Fault
After inserting 3	2 0 3	------------
After inserting 2	2 0 3	------------
After inserting 1	2 0 1	Page Fault
After inserting 2	2 0 1	------------
After inserting 0	2 0 1	------------
After inserting 1	2 0 1	------------
After inserting 7	7 0 1	Page Fault
After inserting 0	7 0 1	------------
After inserting 1	7 0 1	------------

Total number of page faults: 09

The first three references cause faults that fill the three empty frames. The reference to page 2 replaces page 7, because 7 will not be used until reference 18, whereas page 0 will be used at 5, and page 1 at 14. The reference to page 3 replaces page 1, as page 1 will be the last of the three pages in memory to be referenced again. With only nine page faults, optimal replacement is much better than other algorithms.

4. **Counting-Based Page Replacement(LFU):** The **least frequently used (LFU) page-replacement algorithm** requires that the page with the smallest count be replaced. The reason for this selection is that an actively used page should have a large reference count. A problem arises, however, when a page is used heavily during the initial phase of a process but then is never used again. Since it was used heavily, it has a large count and remains in memory even though it is no longer needed.
   For example, consider the following reference string with 3 frames.

7, 0,1, 2, 0, 3, 0, 4, 2, 3, 0, 3, 2,1, 2, 0, 1, 7, 0,1

Initial	E E E	---------
After inserting 7	7 E E	Page Fault
After inserting 0	7 0 E	Page Fault
After inserting 1	7 0 1	Page Fault
After inserting 2	2 0 1	Page Fault
After inserting 0	2 0 1	-----------
After inserting 3	2 0 3	Page Fault
After inserting 0	2 0 3	-----------
After inserting 4	4 0 3	Page Fault
After inserting 2	4 0 2	Page Fault
After inserting 3	3 0 2	Page Fault
After inserting 0	3 0 2	------------
After inserting 3	3 0 2	------------
After inserting 2	3 0 2	------------
After inserting 1	3 1 2	Page Fault
After inserting 2	3 1 2	------------
After inserting 0	0 1 2	Page Fault
After inserting 1	0 1 2	------------
After inserting 7	0 1 7	Page Fault
After inserting 0	0 1 7	------------

After inserting 1	0 1 7	------------

Total number of page faults: 11

Thrashing:

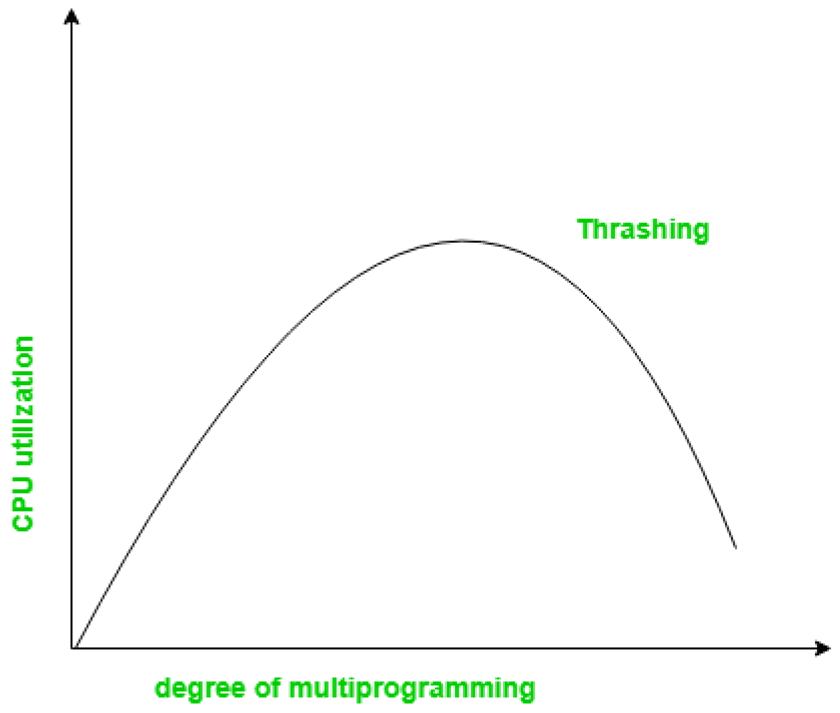
When a program need space larger than RAM or it need space when RAM is full, Operating System will try to allocate space from secondary memory and behaves like it has that much amount of memory by serving to that program. This concept is called virtual memory. To know about thrashing we first need to know what is page fault and swapping.

Page fault and swapping: We know every program divided into some pages. When a program need a page which is not in RAM that is called page fault. Whenever a page fault happens, operating system will try to fetch that page from secondary memory and try to swap it with one of the page in RAM. This is called swapping.

What is Thrashing ?
If this page fault and then swapping happening very frequently at higher rate, then operating system has to spend more time to swap these pages. This state is called thrashing. Because of this, CPU utilization is going to be reduced.

Cause of Thrashing: The operating system monitors CPU utilization. If CPU utilization is too low, we increase the degree of multiprogramming by introducing a new process to the system. A global page-replacement algorithm is used; it replaces pages without regard to the process to which they belong. Now suppose that a process enters a new phase in its execution and needs more frames. It starts faulting and taking frames away from other processes.

The CPU scheduler sees the decreasing CPU utilization and *increases* the degree of multiprogramming as a result. The new process tries to get started by taking frames from running processes, causing more page faults and a longer queue for the paging device. As a result, CPU utilization drops even further, and the CPU scheduler tries to increase the degree of multiprogramming even more. Thrashing has occurred, and system throughput plunges. The pagefault rate increases tremendously As a result, the effective memory-access time increases. No work is getting done, because the processes are spending all their time paging.



To prevent thrashing, provide a process with as many frames as it needs. But how do we know how many frames it "needs'? There are several techniques. The working-set strategy starts by looking at how many frames a process is actually using.

Effect of Thrashing:

Whenever thrashing starts, operating system tries to apply either **Global page replacement** Algorithm or **Local page replacement** algorithm.

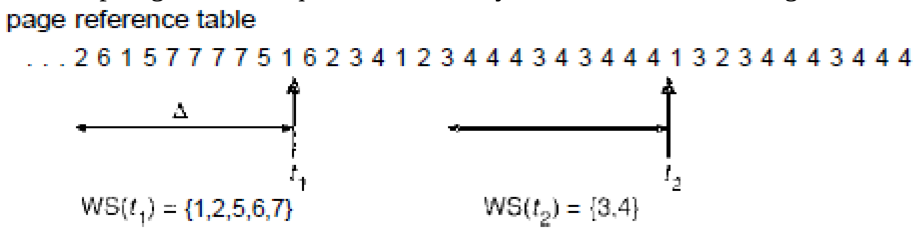
Global Page Replacement Since global page replacement can access to bring any page, it tries to bring more pages whenever thrashing found. But what actually will happen is, due to this, no process gets enough frames and by result thrashing will be increase more and more. So global page replacement algorithm is not suitable when thrashing happens.

Local Page Replacement Unlike global page replacement algorithm, local page replacement will select pages which only belongs to that process. So there is a chance to reduce the thrashing. But it is proven that there are many disadvantages if we use local page replacement. So local page replacement is just alternative than global page replacement in thrashing scenario.

Techniques to handle:

1. **Working-Set Model:** The idea is to examine the most recent ▲ page references. The set of pages in the most recent ▲ page references is the working set. If a page is in,active use, it will be in the working set. If it is no longer being used, it will drop from the working set ▲ time units after its last reference.

   For example, given the sequence of memory references shown in Figure



If ▲= 10 memory references, then the working set at time *t1* is {1, *2,* 5, 6, *7}.* By time *h,* the working set has changed to {3, 4}. The accuracy of the working set depends on the selection of ▲. If ▲ is too small, it will not encompass the entire locality; if ▲ is too large, it may overlap several localities. In the extreme, if ▲ is infinite, the working set is the set of pages touched during the process execution.

The most important property of the working set, then, is its size. If we compute the working-set size, *WSSj,* for each process in the system, we can then consider that

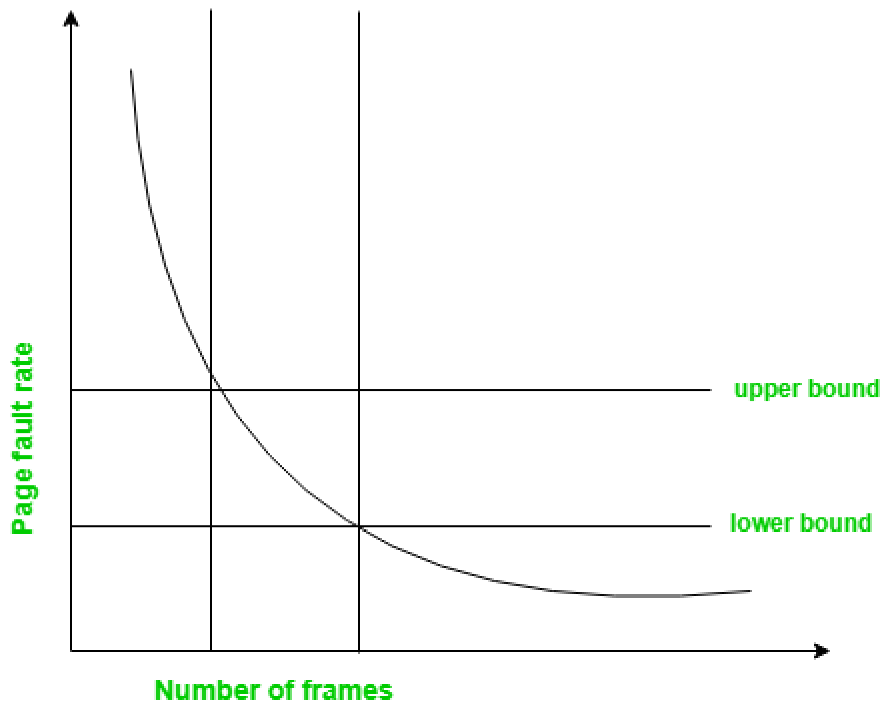
$$D = \sum WSS_i$$

where *D* is the total demand for frames. Each process is actively using the pages in its working set. Thus, process *i* needs WSS_i frames. If the total demand is greater than the total number of available frames (D > *m),* thrashing will occur, because some processes will not have enough frames.

This technique prevents thrashing along with ensuring the highest degree of multiprogramming possible. Thus, it optimizes CPU utilisation.

2. **Page Fault Frequency –**

   A more direct approach to handle thrashing is the one that uses Page-Fault Frequency concept.



The problem associated with Thrashing is the high page fault rate and thus, the concept here is to control the page fault rate.

If the page fault rate is too high, it indicates that the process has too few frames allocated to it. On the contrary, a low page fault rate indicates that the process has too many frames.

Upper and lower limits can be established on the desired page fault rate as shown in the diagram. If the page fault rate falls below the lower limit, frames can be removed from the process. Similarly, if the page fault rate exceeds the upper limit, more number of frames can be allocated to the process.
In other words, the graphical state of the system should be kept limited to the rectangular region formed in the given diagram.

Here too, if the page fault rate is high with no free frames, then some of the processes can be suspended and frames allocated to them can be reallocated to other processes. The suspended processes can then be restarted later.

FREQUENTLY ASKED QUESTIONS

1. Distinguish between Logical and Physical address space?
2. What is the purspose of Paging and page tables?
3. What is Paging? Discuss the Paging model of logical and physical memory?
4. What is a Virtual Memory? Discuss the benefits of virtual memory technique?
5. What is Thrashing? What can the system do to eliminate this problem?
6. What is a Pagefault? Explain the steps involved in handling a pagefault with a neat sketch?
   OR
   Discuss the procedure for handling page fault in Demand Paging?
7. Explain the need of page replacement and explain with an example?
8. Suggest the best algorithm for the given reference string.
   7, 0, 1, 2, 0, 3, 0, 4, 2, 3, 0, 3, 2, 1, 2, 0, 1, 7, 0, 1 with three page frames.
9. Compute the number of page faults for optimal page replacement strategy for the given reference string 1, 2, 3, 4, 5, 3, 4, 1, 6, 7, 8, 7, 8, 9, 7, 8, 9, 5, 4, 5, 4, 2 with 4 page frames.
10. Consider the following reference string:
    1, 2, 3, 4, 2, 1, 5, 6, 2, 1, 2, 3, 7, 6, 3, 2, 1, 2, 3, 6

How many page faults would occur for the optimal page replacement algorithm, assuming three frames and all frames are initially empty.

11. Discuss the hardware support required to support demand paging?
12. What factors effecting the performance of demand paging? Compute the performance of demand paging when page fault service rate is 8 m/sec and memory access time is 200nsec.
13. Explain how demand paging effects the performance of a computer system?
14. Explain the difference between internal and external fragmentation?
15. Discuss various issues related to the allocation of frames to processes?
16. Why Segmentation and Paging combined into one scheme?
    Ans: Segmentation and paging are often combined inorder to improve upon each other. Segmented paging is helpful when the page table becomes very large. A large contiguous section of the page table that is unused can be collapsed into a single segmented table entry with a page table address of zero. Paged segmentation handles the case of having very long segments that require a lot of time for allocation. By paging the segments, we reduce wasted memory due to external fragmentation as well as simplify the allocation.
17. What is the cause of Thrashing? How does the system detect Thrashing? How to eliminate this problem?