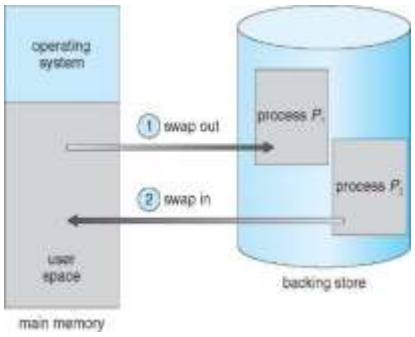
Unit IV:
Memory Background
Content:

- Swapping
- Contiguous Memory Allocation
- Paging
- Segmentation
- virtual memory

1. Swapping

A process must be in memory to be executed. A process 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 time slice 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 as shown following figure,



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 restriction is dictated by the method of address binding. Swapping requires a backing store. The backing store is commonly a fast disk. 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.

2. Contiguous Memory Allocation

The memory is usually divided into two partitions: one for the operating system and one for the user processes.

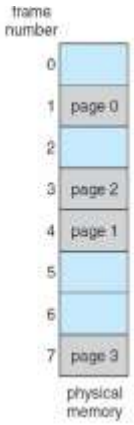
a) Memory Allocation –

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 depends on the number of partitions.

Page No	Frame No	Base Address
0	1	100
1	4	400
2	3	300
3	7	700

Table – Page Map Table (PMT)

Note – Each frame is 100 bytes long



In this multiple partition method, when a partition is free, a process is selected from the ready queue and is loaded into the free partition. When the process terminates, the partition becomes available for another process.

In the variable partition scheme, the operating system keeps a table indicating which parts of memory are available and which are occupied. Initially, all memory is available for user processes and is considered one large block of available memory called as a hole.

When a process arrives and needs memory, the system searches the set for a hole that is large enough for this process. If the hole is too large, it is split into two parts. One part is allocated to the arriving process; the other is returned to the set of holes.

When a process terminates, it releases its block of memory, which is then placed back in the set of holes. If the new hole is adjacent to other holes, these adjacent holes are merged to form one larger hole.

b) Fragmentation –

There are two main fragmentation issues in contiguous memory allocation.

1) Internal Fragmentation –

Allocated memory may be slightly larger than requested memory; this size difference is memory internal to a partition, but not being used.

Solution to the problem of internal fragmentation is variable size policy. The size of partition depends on size of process in dynamic allocation.

2) External Fragmentation –

Total free memory space exists to satisfy a request, but it is not contiguous. One possible solution to the external fragmentation problem is to permit the logical address space of the processes to be noncontiguous.

Two techniques achieve this solution:

a) Paging

b) Segmentation

3. Paging

Paging is a memory management scheme that permits the physical address space of a process to be noncontiguous. Paging avoids external fragmentation.

a) 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.

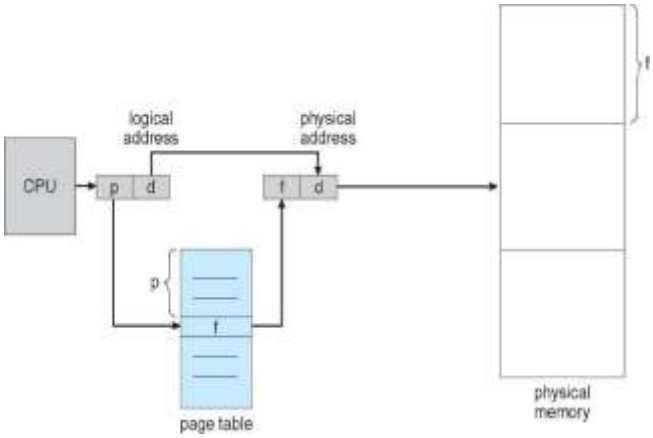
When a process is to be executed, its pages are loaded into any available memory frames. The paging model of memory is shown in following figure,

Hardware Support –

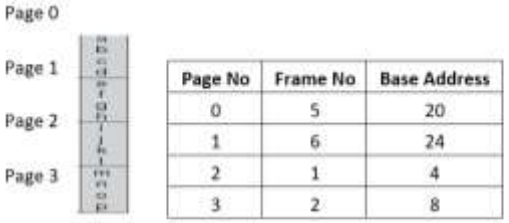
The hardware support for paging is shown in following figure. Every address generated by the CPU is divided into two parts:

1) Page number (p)
2) 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.

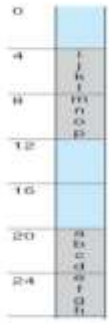


Physical address of value = Base address of frame + Page Offset



Page No	Frame No	Base Address
0	5	20
1	6	24
2	1	4
3	2	8

Note – Each frame is 4 bytes long



Consider the memory in following figure, using a page size of 4 bytes and a physical memory of 28 bytes. Suppose CPU want to find physical address of "b". CPU generated address of required value into two parts: page 0, offset 1 i.e. value – "b". Indexing into the page table, we find that page 0 is in frame 5.

Physical address of value = (Frame Number x Frame Size) + Page Offset
Thus, physical address of value "b" is 21 = (5 x 4) + 1

4. Segmentation

a) Basic Method –

Segmentation is a memory management scheme that supports user view of memory. 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.

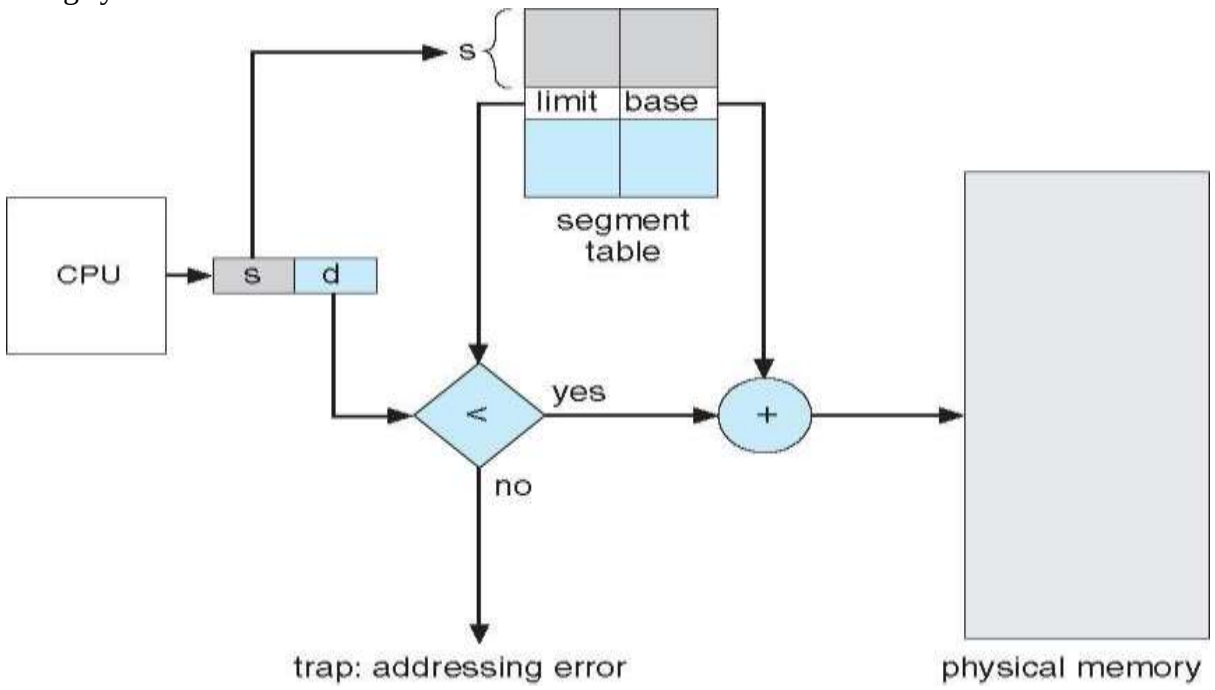
The user therefore specifies each address by two quantities: a segment name and an offset. For simplicity, segments are referred by a segment number, rather than by a segment name. Thus, a logical address consists of a, <Segment-number, offset>

b) Hardware Support –

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.

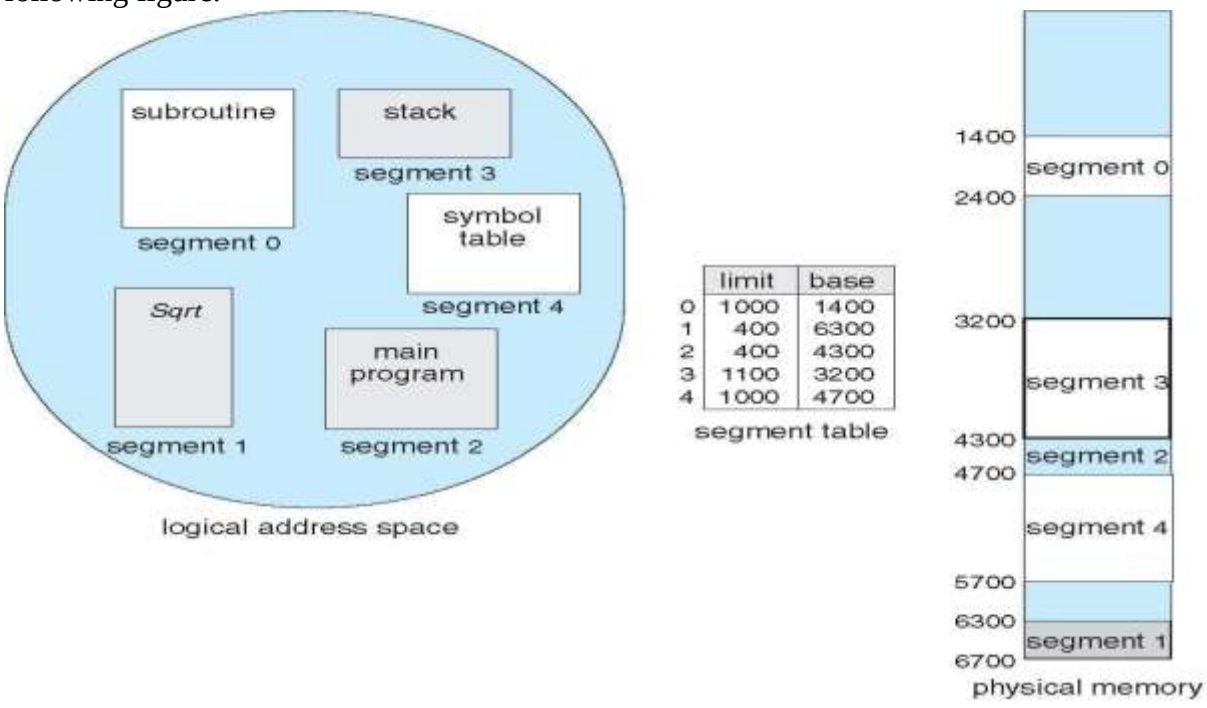
The use of a segment table is shown in following figure. A logical address consists of two parts: a segment number, "s", and an offset into that segment, "d".

The segment number is used as an index to the segment table. The offset "d" of the logical address must be between 0 and the segment limit. If it is not, we trap to the operating system.



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, consider the situation shown in following figure. We have five segments numbered from 0 through 4. The segments are stored in physical memory as shown in following figure.



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.

5. virtual memory

a) Background –

Virtual memory is a technique that allows the execution of processes that are not completely in memory. One major advantage of this scheme is that programs can be larger than physical memory.

This technique frees programmers from the concerns of memory storage limitations. Virtual memory involves the separation of logical memory as perceived by users from physical memory.

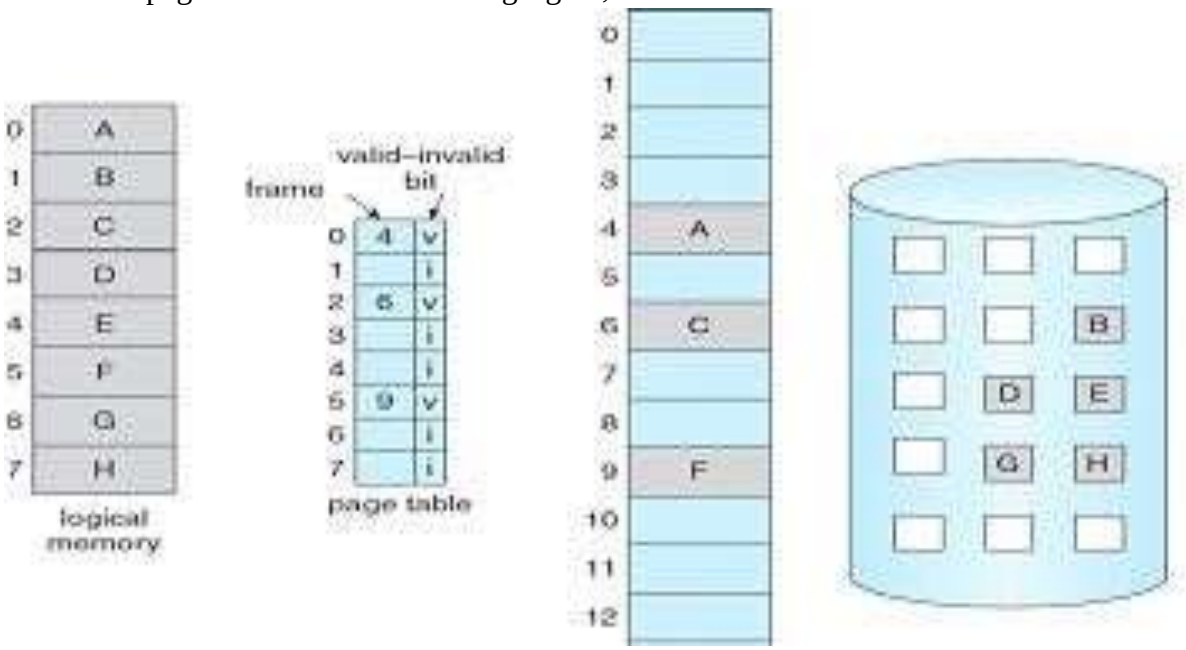
This separation, allows an extremely large virtual memory to be provided for programmers when only a smaller physical memory is available.

b) Demand paging –

Consider how an executable program might be loaded from disk into memory. One option is to load the entire program in physical memory at program execution time. Another way is to initially load pages only as they are needed.

With demand paged virtual memory, pages are only loaded when they are demanded during program execution memory.

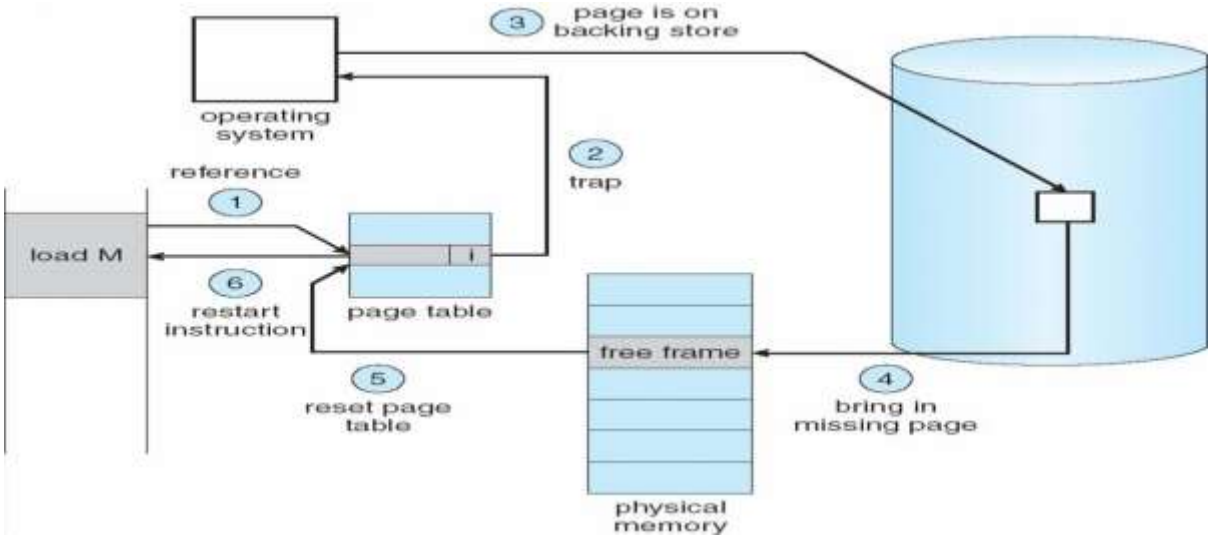
A demand paging system is similar to a paging system. Breaking physical memory into fixed sized blocks called frames and breaking logical memory into blocks of the same size called pages as shown on following figure,



When we want to execute a process, we swap it into memory. A lazy swapper never swaps a page into memory unless that page will be needed.

The valid-invalid bit is used to identify whether page is in main memory or virtual memory. When bit is set to "valid" the associated page is both legal and in memory. If the bit is set to "invalid," the page is currently on the disk i.e. virtual memory.

But what happens if the required page is not available in main memory? The page fault is occurred. The procedure for handling page fault as shown in following figure,



1. We check a page table to determine whether the reference was a valid or an invalid.
2. If the reference was invalid, terminate the process. If it was valid, we now page it in.
3. We find a free frame.
4. We schedule a disk operation to read the desired page into the newly allocated frame.
5. When the disk read is complete, we modify the page table and the page table to indicate that the page is now in memory.
6. We restart the instruction that was interrupted by the page fault.