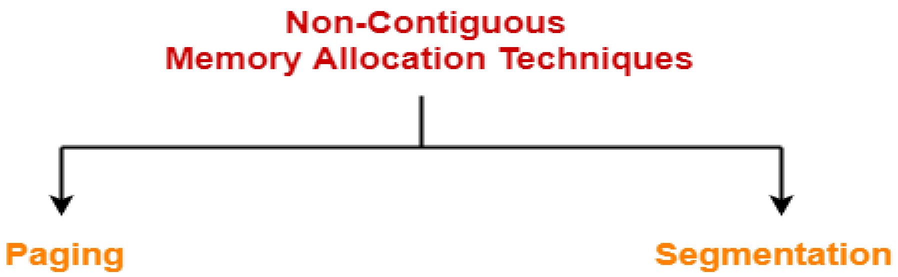
Unit 4

Topic 3: Paging-I

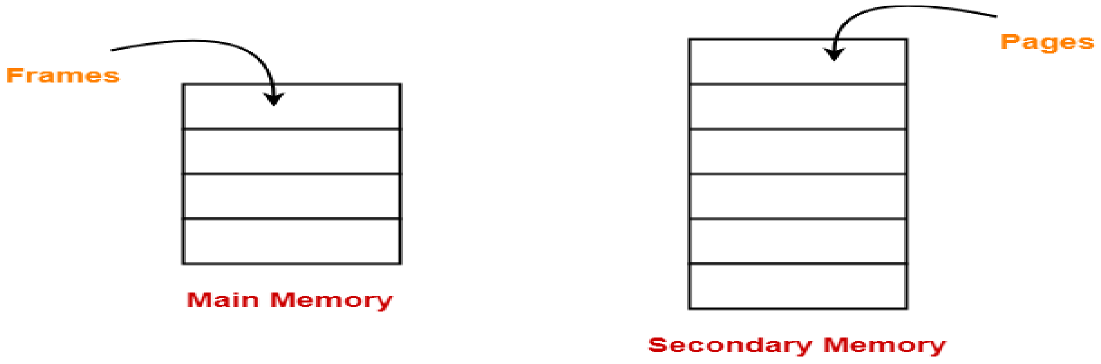
To overcome the problem of External fragmentation in contiguous memory allocation is we go for non-contiguous memory allocation. It allows to store parts of a single process in a non-contiguous fashion. Thus, different parts of the same process can be stored at different places in the main memory.

There are two popular techniques used for non-contiguous memory allocation-



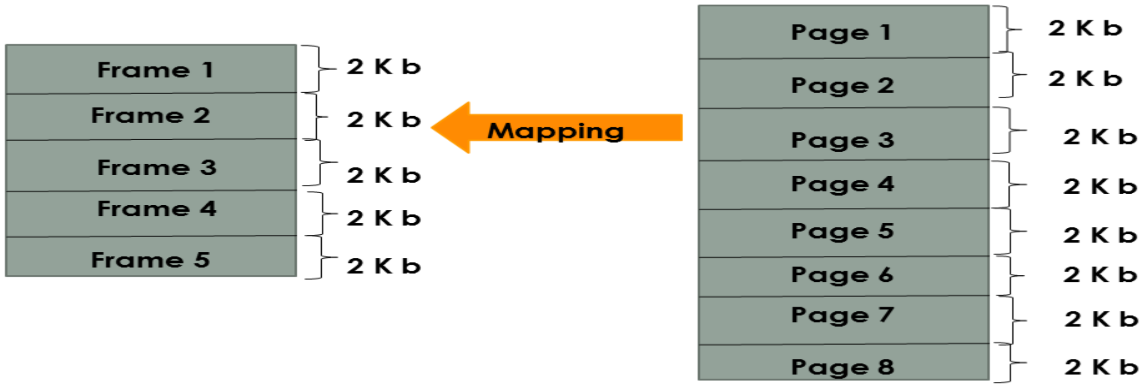
Paging

The paging technique divides the physical memory(main memory) into fixed-size blocks that are known as **Frames** and also divide the **logical memory(secondary memory) into blocks of the same size** that are known as **Pages.** Meaning that the process residing in secondary memory is divided in to pages.



The process pages are stored at a different location in the main memory. The thing to note here is that the size of the page and frame will be the same. A page is mapped into a frame

Basic Method



CPU always generates a logical address consisting of two parts-

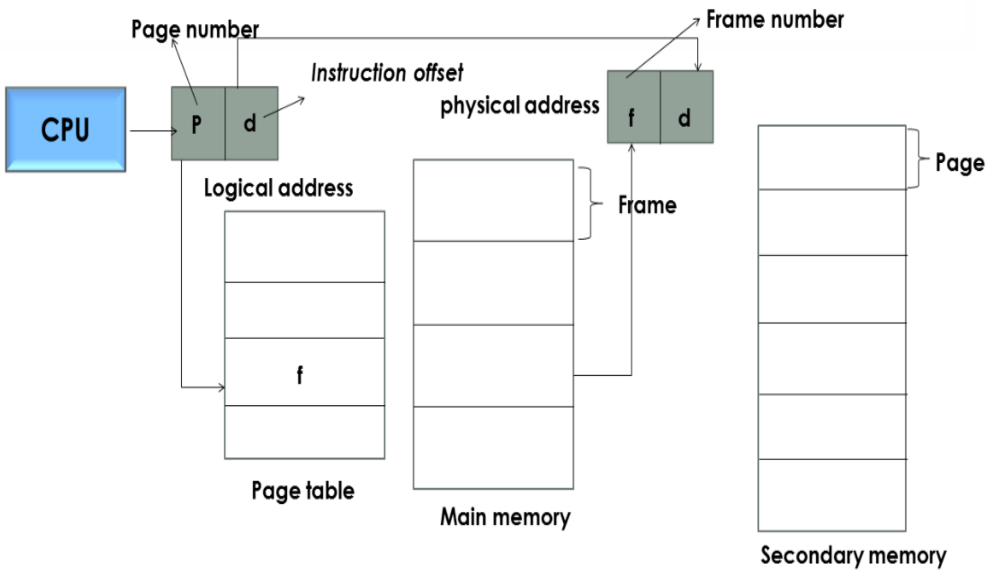
Page Number: specifies the specific page of the process from which CPU wants to read the data.

Page Offset : specifies the specific word/instruction on the page that CPU wants to read.

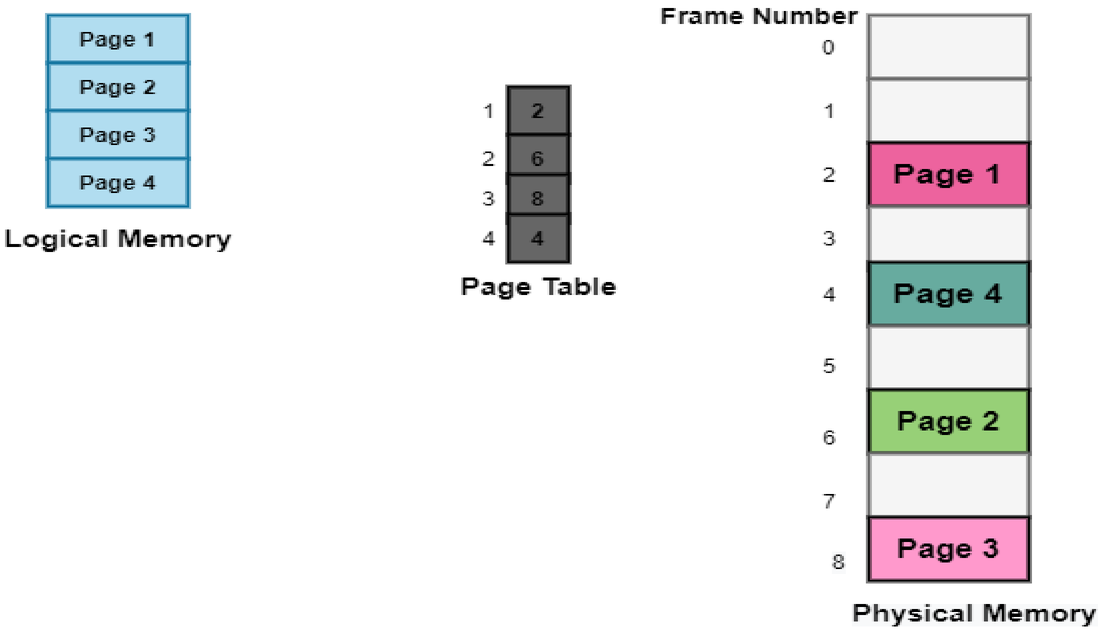
A physical address is needed to access the main memory and consists of and offset .

frame number : specifies the specific frame where the required page is stored.

Page Offset : Specifies the specific word that has to be read from that page



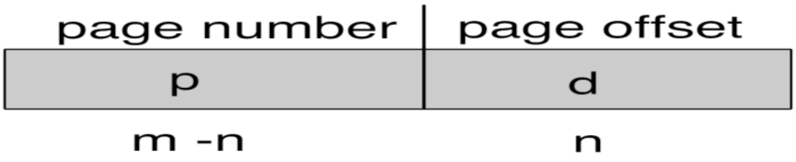
The logical address that is generated by the CPU is translated into the physical address with the help of the page table.Thus page table mainly provides the corresponding frame number where that page is stored in the main memory.



The above diagram shows the paging model of Physical and logical 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 the 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:



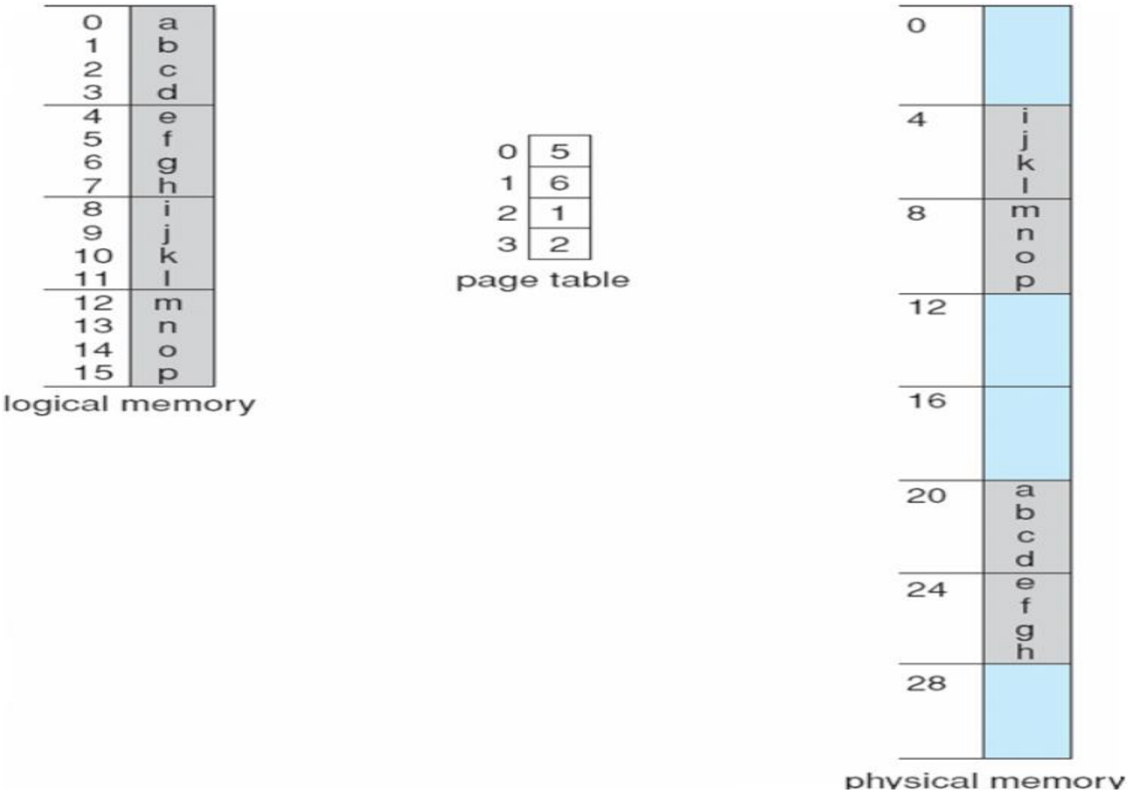
Example, consider the memory in Figure below. Here, in the logical address, n= 2 and m = 4. Using a page size of 4 bytes and a physical memory of 32 bytes (8 pages), we show how the user's view of memory can be mapped into physical memory.

♣ 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 × 4) + 0].

♣ Logical address 3 (page 0, offset 3) maps to physical address 23 [= (5 × 4) + 3].

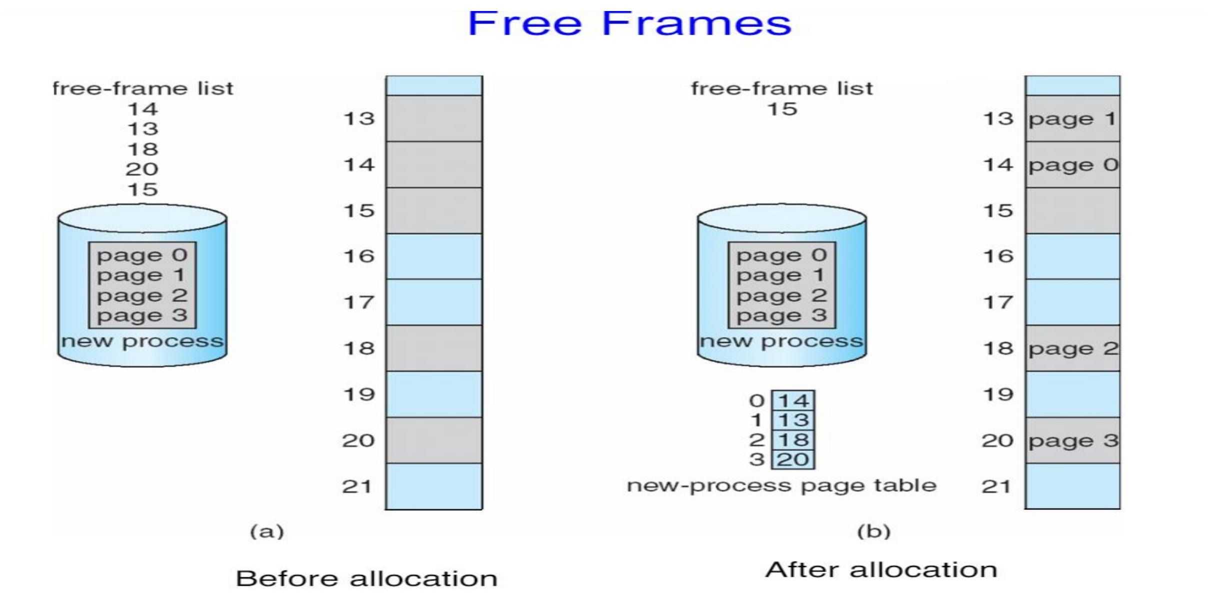
♣ Logical address 4 is page 1, offset 0; according to the page table, page 1 is mapped to frame 6. Thus, logical address 4 maps to physical address 24 [= (6 × 4) + 0].

♣ Logical address 13 maps to physical address 9.



When we use a paging scheme, we have no external fragmentation: any free frame can be allocated to a process that needs it. However, we may have some internal fragmentation. For example, if page size is 2,048 bytes, a process of 72,766 bytes will need 35 pages plus 1,086 bytes. It will be allocated 36 frames, resulting in internal fragmentation of 2,048 - 1,086 = 962 bytes. In the worst case, a process would need 11 pages plus 1 byte. It would be allocated 11 + 1 frames, resulting in internal fragmentation of almost an entire frame.

When a process arrives in the system to be executed, its size, expressed in pages, is examined. Each page of the 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 first page of the process is loaded into one of the allocated frames, and the frame number is put in the page table for this process. The next page is loaded into another frame, its frame number is put into the page table, and so on



Hardware Support

In Operating System, for each process page table will be created, which will contain a Page Table Entry (PTE). This PTE will contain information like frame number which will tell where in the main memory the actual page is residing.

Now the question is where to place the page table, such that overall access time (or reference time) will be less. The problem initially was to fast access the main memory content based on the address generated by the CPU (i.e. logical/virtual address). Initially, some people thought of using registers to store page tables, as they are high-speed memory so access time will be less.

The idea used here is, to place the page table entries in registers, for each request generated from the CPU (virtual address), it will be matched to the appropriate page number of the page table, which will now tell where in the main memory that corresponding page resides. Everything seems right here, but the problem is registered size is small (in practice, it can accommodate a maximum of 0.5k to 1k page table entries) and the process size may be big hence the required page table will also be big (let's say this page table contains 1M entries), so registers may not hold all the entries of the Page table. So, this is not a practical approach.

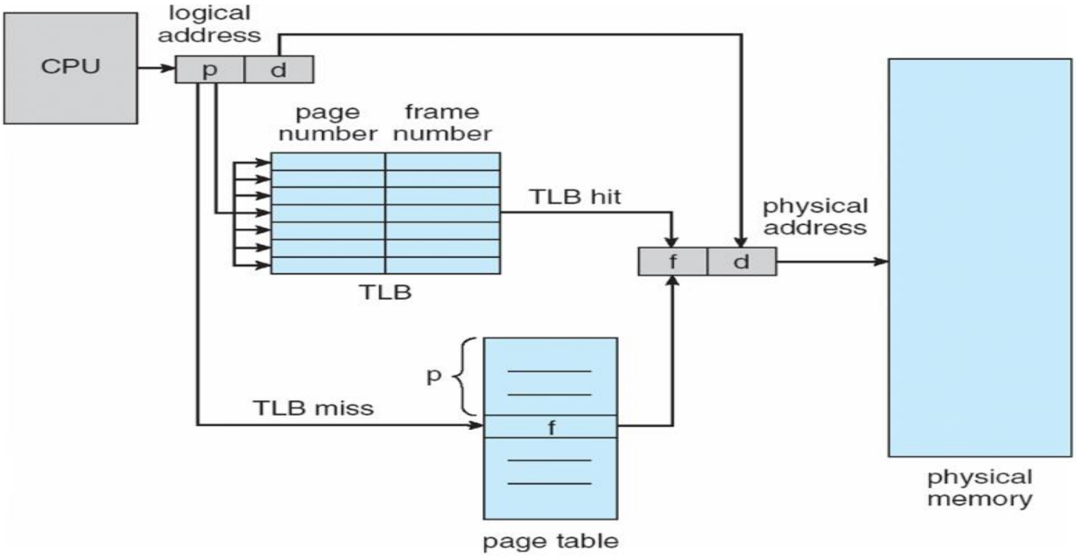
The entire page table was kept in the main memory to overcome this size issue. but the problem here is two main memory references are required:

1. To find the frame number
2. To go to the address specified by frame number

The two-memory access problem can be solved by the use of a special fast-lookup hardware cache called *associative memory* or *translation look-aside buffers (TLBs)*. 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. Typically, the number of entries in a **TLB is small**, often numbering between 64 and 1,0241.

The TLB is used with pages tables in the following way :

Paging Hardware With TLB



- 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 (known as **TLB hit**), its frame number is immediately available and is used to access memory.
- If the page number is not found in the TLB (known as **TLB miss**) a memory reference to the page table must be made.
- when the frame number is obtained, we can use it to access memory.
- In addition, we 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.
- Replacement policies range from **least recently used** (LRU) to random.

The percentage of times that a particular page number is found in the TLB is called TLB Hit Ratio.Let us now see an example of how effective memory access time is calculated: Assume it takes 20 nsecs to search TLB and 100 nsecs to access memory, what is the effective memory access time, if the hit ratio is 80%?

If the page number is present in the TLB, memory access time = 120 nsecs (20+100)

If page number is not present in the TLB, memory access time = 220 nsecs (20+100+100)

If 80 percent hit ratio, effective memory-access time = 0.80 x 120 + 0.20 x 220 = 140 nsecs.

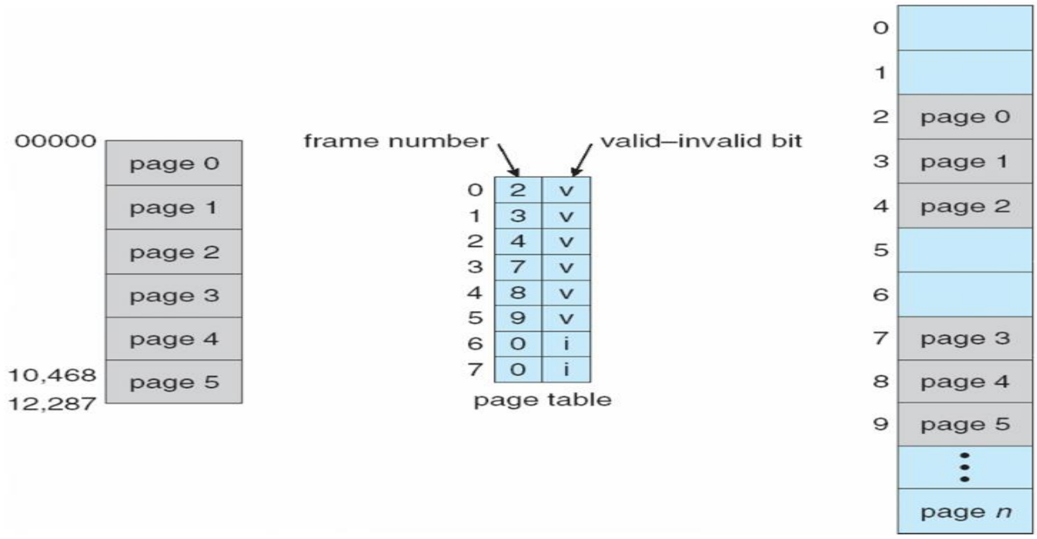
Protection

- 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 bit can define a page to be read-write or read-only.
- Every reference to memory goes through the page table to find the correct frame number. At the same time that the physical address is being computed, the protection bits can be checked to verify that no writes are being made to a read-only page.
- An attempt to write to a read-only page causes a hardware trap to the operating system (or memory-protection violation).
- 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.
- Illegal addresses are trapped by use of the valid-invalid bit. The OS sets this bit for each page to allow or disallow access to the page.
- Suppose, for example, that in a system with a 14-bit address space (0 to 16383), we have a program that should use only addresses 0 to 10468.
  - item 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 OS (invalid page reference).

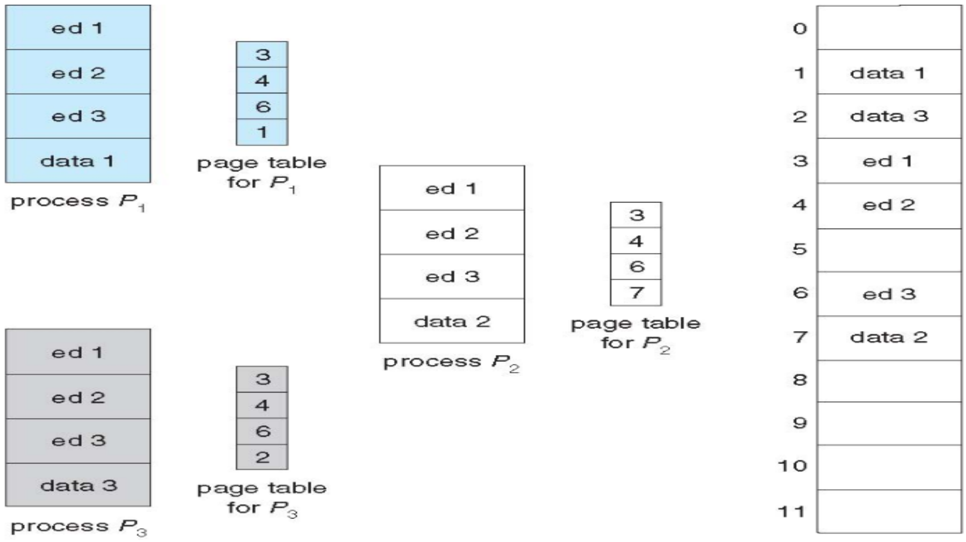
Valid (v) or Invalid (i) Bit In A Page Table



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. If the code is reentrant code(or pure code however, it can be shared, as shown in Figure)

Shared Pages Example



Here we see a three-page editor-each page 50 KB in size (the large page size is used to simplify the figure)-being shared among three processes. Each process has its own data page. Re-entrant code is non-self-modifying code: it never changes during execution. Thus, two or more processes can execute the same code at the same time. Each process has its own copy of registers and data storage to hold the data for the process's execution. The data for two different processes will be of course, be different. 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)50 KB instead of 8,000 KB-a significant savings. Other heavily used programs can also be shared -compilers, window systems, run-time libraries, database systems, and so on. To be sharable, the code must be reentrant. The read-only nature of shared code should not be left to the correctness of the code; the operating system should enforce this property.