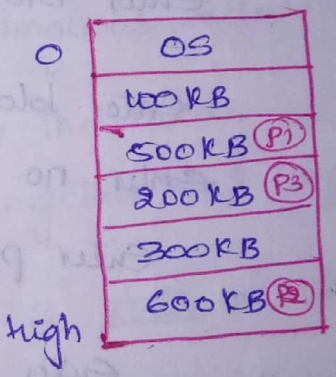
Strategies for Memory Allocation: The most popular strategies are: First fit, Best fit, & Worst fit. They select a free hole from the set of available holes.

1) **First fit:** Allocate the first hole that is big enough. Searching can start either at the beginning of the set of holes or at the location where the previous first-fit search ended. We can stop searching as soon as we find a free hole that is large enough.

eg: Given 5 mem. partitions of 100 KB, 500 KB, 200 KB, 300 KB, 600 KB (in order), how would the first fit, best fit & worst fit algs. place the processes of 212 KB, 417 KB, 112 KB & 426 KB (in order).



Ans: P1 – 212 KB — 500 KB
P2 – 417 KB — 600 KB
P3 – 112 KB — 200 KB
P4 – 426 KB — Must Wait

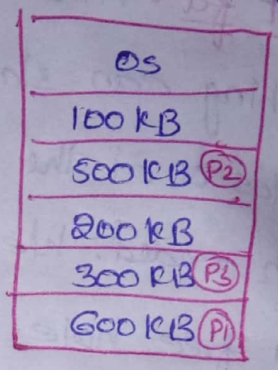
Note: We have 2 free mem. blocks 100 KB & 300 KB but we are not able allocate mem. for 426 KB process.

2) **Best fit:** Allocate the smallest hole that is big enough. We must search for the entire list, unless the list is ordered by size. This strategy produces the smallest leftover hole.

Ans: P1 – 212 KB — 300 KB
P2 – 417 KB — 500 KB
P3 – 112 KB — 200 KB
P4 – 426 KB — 600 KB

3) Worst Fit: Allocate the largest hole. Again we need to search the entire list unless it is sorted by size. This strategy produces the largest leftover hole, which may be more useful than the smallest left over hole from the best fit approach.

P1 — 212 KB — 600 KB
P2 — 417 KB — 500 KB
P3 — 112 KB — 300 KB
P4 — 426 KB — Must wait



Implementation in lab:

Ilp: Enter no. of blocks: 5
Enter block 0 size:
Enter block 1 size:
:
Enter block 4 size:
Enter no. of processes: 4
Enter process 0 size:
:
Enter process 3 size:

Olp: The process 0 is allocated to block:
:
The process 3 is not allocated.

Fragementation: There are mainly 2 types of framentation we have 1) External Fragmentation
2) Internal Fragmentation

1) External Fragmentation: As processes are loaded & removed from mem., the free mem. space is broken into little pieces. external fragmentation exists when there is enough

total mem. space to satisfy a request but the available space is not continguous.

eg: First fit & best fit strategies

50% Rule: statistical analysis of First fit reveals that, even with some optimization, given N allocated blocks, another 0.5N blocks will be lost to fragmentation i.e., one third (1/3rd) of mem. is unusable! this property is known as 50% (50 percent) rule.

eg: 100 blocks

if 40 blocks are allocated ⇒ 20 blocks will be lost for fragmentation.

Soln for External fragmentation: 1) Compaction

2) Non-Contiguous allocation

i) Compaction: The goal here is to shuffle the mem. contents so as to place all free mem. together in 1 large block.

* Compaction can't be done if relocation is static or load time & its only done if relocation is dynamic.

disadv: Could be expensive.

ii) Non-Contiguous mem. allocation: Allowing a process to be allocated physical mem. wherever such mem. is available. Two complementary techniques achieves this Soln. 1) Paging 2) Segmentation.

2) Internal Fragmentation: Consider a multiple-partitioned allocation scheme with a hole of 2000 bytes. Suppose that the next process requests 1998 bytes. If we allocate

exactly the requested block, we are left with 2 bytes of hole. This wasted/unused mem. that is internal to a block is called as Internal fragmentation.

* "Mem. fragmentation occurs when a sys contains mem. that is technically free but the computer can't utilise"

* ii) Non-Contiguous Mem. Allocation: Mem. for a process can be allocated in a non-contiguous manner. There are 2 techniques to do this a) Paging, b) Segmentation

a) Paging: It is a mem-mgnt scheme that permits the phy address space of a process to be non-contiguous. Paging avoids external fragmentation & the need for compaction.

It also solves the considerable prblm of fitting mem. chunks of varying sizes onto the backing store; most mem. mngt schemes used before the instruction of paging suffered from this prblm. The prblm arises because some code fragments or data residing in main mem. need to be swapped out, space must be found on the backing store. The backing store has the same fragmentation prblms, but access is much slower, so compaction is impossible.

Because of its advantages, over earlier methods, paging in its various forms is used in most OS.

Basic Method: The basic method for paging involves breaking physical mem. into fixed sized blocks called

frames & breaking logical mem into same size called pages. When a process is to be executed, its pages are loaded into any available mem. frames from their source (a file sys or a backing store). The backing store is divided into fixed-sized blocks that are of the same size as the mem. frames.

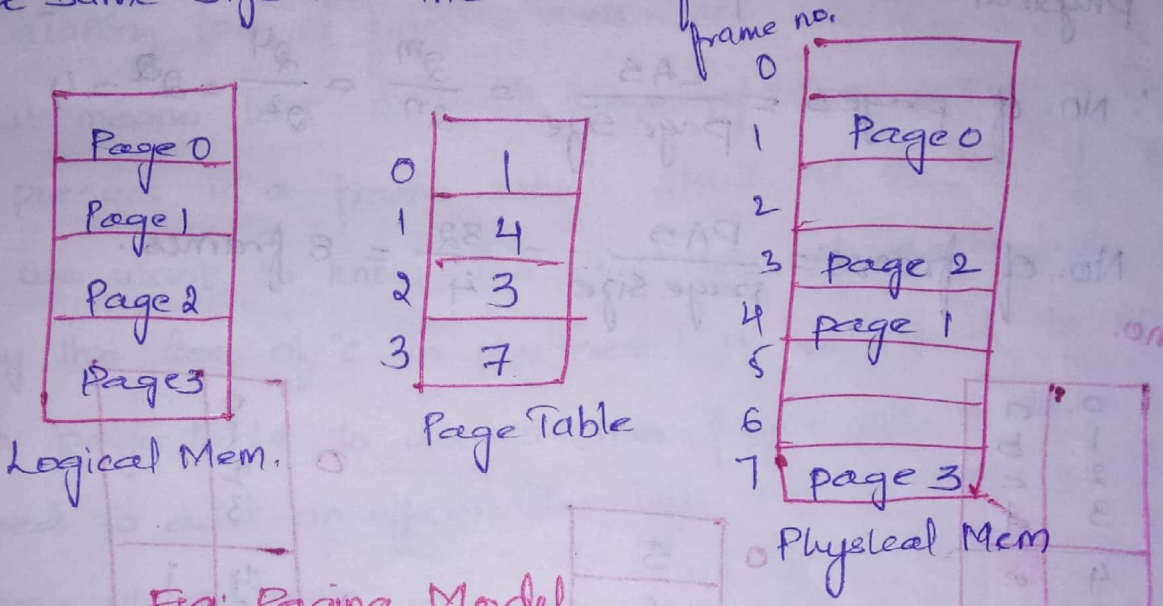


Fig: Paging Model

✓ When CPU executes a process, it knows only the logical address space of a process & hence there shld be some mapping of logical address to the physical address of a process & as process is allocated with non-contiguous mem locations, the base address is not enough. Hence we need a mapping to all the frames/pages to the physical addr space. This can be achieved by using page table. Page table stores the physical address of frames.

✓ If the CPU needs physical address of a particular page it will check the index of the page table & the indexed location consists of correct/actual phy addr of a page.

Eg: Consider the logical addr space as 16 bytes (2^4 gen represented by 2^m, where m=4 here). & page size as 4 bytes (2^n rep, where n=2), also phy mem. as 32 bytes.

page size $= 2^n = 2^2 = 4$ Bytes

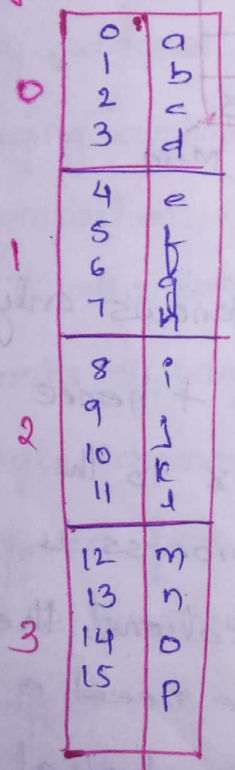
Logical addr. space $= 2^m = 2^4 = 16$ B

physical addr. space = 32 B.

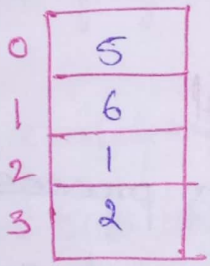
$\therefore$ No. of pages $= \frac{\text{LAS}}{\text{page size}} = \frac{2^m}{2^n} = \frac{2^4}{2^2} = 2^2 = 4$

No. of frames $= \frac{\text{PAS}}{\text{page size}} = \frac{32}{4} = 8$ frames.

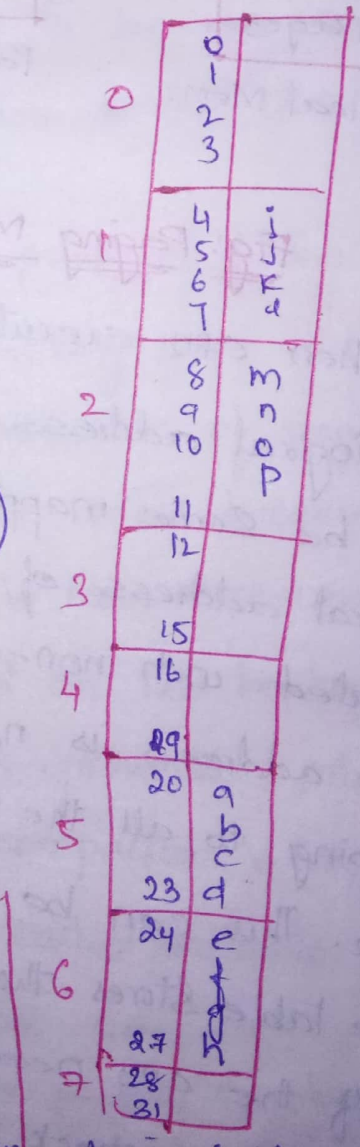
page no.



Logical Mem.



Page Table (contains frame Number)



physical Mem

Fig. Paging Example

If we want to know the physical address of a corresponding page, then we need to consider page table. As an eg., the first page

of the logical addr. space of process, (abcd) is mapped to 5th frame in physical address.

LA → 0 → 5th frame → PA

i) To determine the actual phys. addresses, we have to multiply page size with the frame no. i.e., Frame no. x page no.

* starting loc. of frame = 5×4 = 20.

It means that the 1st page of the process (LAS) is present in a frame which starts at a loc. of 20. But if we want to know the exact loc. of each byte in phy. mem say the loc. of 'c' in phy mem, 1st we need to consider the page table to access the frame no. & then we need to add an offset (the byte no. in LA) to the frame no. address.

i.e., (5×4) + 2 = 22 (PA of 'c').

In this way, the logical address is divided into 2 parts page no.(p) & page offset (d). We have to have the inf. of these 2 variables to correctly identify the location of any instruction/data in the physical address space.

eg1: page 0, offset 0 (p=0 & d=0) ⟺ LA = 0

⇒ PA = (5×4) + 0 = 20

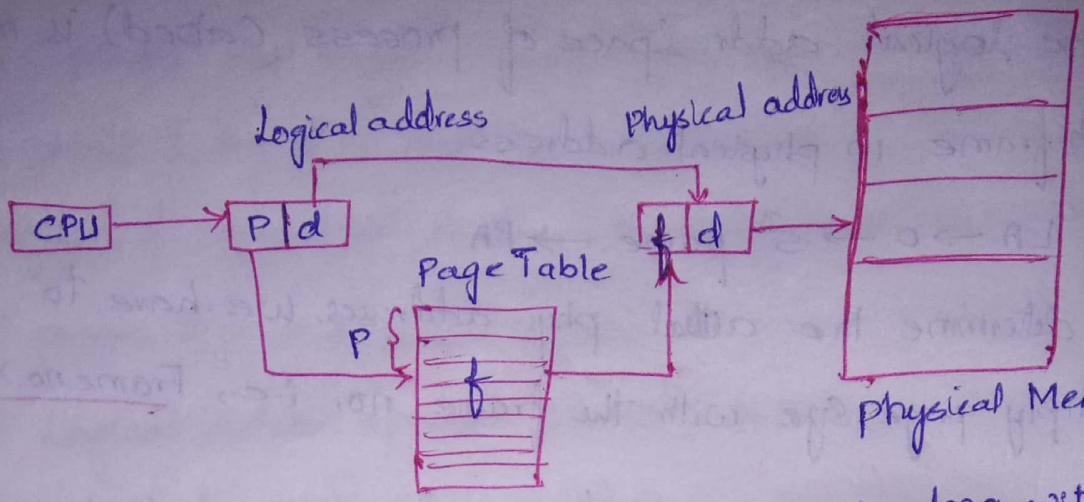
eg2: Page 1, offset 2 (p=1 & d=2) ⟺ LA = 6

⇒ PA = (6×4) + 2 = 26

LA: ↑ page size, ↑ No of

* We need a special hw to perform this mapping.

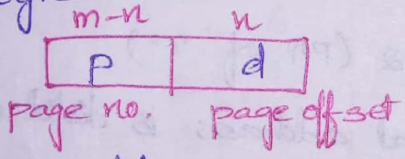
The address generated by CPU is divided into 2 parts, the 1st part denotes the page no. & is indexed into page table



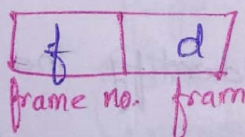
which gives frame no.. The frame no. & along with the page offset (2nd part) is used to find out the location of physical address.

If the logical address space = 2^m & page size = 2^n, then higher order m-n bits of a logical address designate the page no. & the n lower-order bits designate the page offset.

Thus, the logical address is as follow:



The physical address space is as follows:



* Organization of Logical Address Space (LAS):
* LAS is divided into equal no. of pages.
* Page Size is always a power of 2 (i.e., 2^K, $K>0$)
* No. of pages (N) = $\frac{LAS}{PS} = \frac{2^{LA}}{2^K} = 2^{LA-K}$
* No. of bits needed to locate all pages $\Rightarrow p = \log_2 N$
* When we use 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. Notice that frames are allocated as units. If the mem. requirements of a process do not happen to coincide may not be completely full. In the worst case, a process would need 'n' pages plus 1 byte. It would be allocated n+1 frames, resulting in internal fragmentation of almost an entire frame.

If process size is independent of page size, we expect internal fragmentation to avg one-half page per process. This consideration suggests that small page sizes are desirable However, overhead is involved in each page-table entry. & this overhead is reduced as the size of the pages increase Also disk I/O is efficient when the amt of data being transferred is larger. Generally page sizes have grown overtime as processes, data sets & main mem. have become larger. Today, pages typically are blw 4KB & 8KB in size, & some systems support even larger page sizes.

2 GB RAM ⟶ 4KB = page size

$$\text{No of frames} = \frac{2GB}{4KB} = \frac{2^{30}B}{2^{12}B} = 2^{30-12} = 2^{18}$$

When a process arrives in the sys to be executed, its size, expressed in pages, is examined. Each page of the process needs 1 frame. Thus a process requires 'n' pages, atleast 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, & the frame no. is put in the page table for that

process. The next page is loaded into another frame, its frame no. is put into the page table & so on.

✓ An important aspect of paging is the clear separation bw the user's view of mem & the actual physical mem. The user prg views mem as a single space, containing only this prg. Infact the user prg is scattered throughout phy mem, which also holds other progs. The mapping from logical to phy is hidden from the user & done by MMU (OS).

* Since the OS is managing phy mem., it must be aware of the allocation details of phy mem - which frames are allocated & which frames are available, how many total frames are there & so on. This inf. is generally kept in a data structure known as frame Table. The frame table has 1 entry for each physical page frame, indicating whether the latter is free or allocated, & if it is allocated, to which page of which process (or) processes.

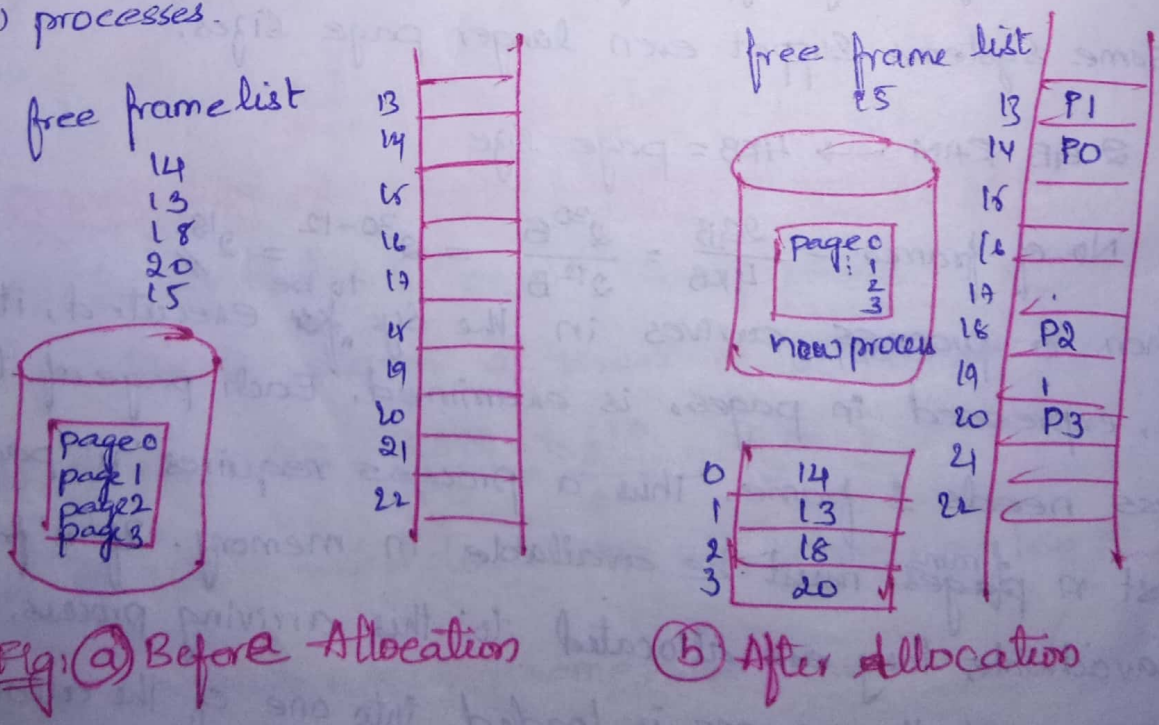


Fig: (a) Before Allocation (b) After Allocation

Paging Hardware: The hardware implementation of paging can be done in several ways.

Page table can be implemented as a set of dedicated registers. Registers are faster but if the page table is large (i.e., 1 million entries) then use of registers may not be feasible. Instead, a page table is kept in main memory, & a page Table Base Register (PTBR) points to the page table changing the page table requires changing only 1 register, substantially decreasing the context switch time.

The prblm with this approach is the time required to access a user memory location. If we want to access location i, we must first index into the page table, using the value in PTBR offset by the page no. of i. This task requires a memory access. It provides us with a frame no., which is combined with the page offset to produce the actual address. We can then access the desired place in mem. With this scheme, 2 mem accesses are needed to access a byte (one for page table entry. 1 for the byte). Thus memory access is slowed by a factor of 2. This delay would be intolerable.

Soln: The standard soln is to use a special, small, fast lookup hw cache, called a translation lookaside Buffer (TLB) The TLB is associative, high speed mem.. Each entry in TLB consists of 2 parts: a key (tag) & a value. When the associative mem. 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 search is fast; the hlw however, is expensive. Typically the no. of entries in TLB is small, often numbering blw 64 & 1024.

The TLB contains only a few of the page table entries. When a logical address is generated by the CPU, its page no. is presented to the TLB. If the page no. is found, its frame no is immediately available & is used to access mem. The whole task may take less than 10% longer than it would be if an unmapped mem. reference is used.

If the page no. is not in the TLB (TLB Miss), a mem. reference to the page table must be made. When the frame no. is obtained, we can use it to access mem. In addition we add the page no & frame no. to the TLB, so that they will be found quickly on the next reference. If the TLB is already full of entires, the OS must select one for replacement. Replacement policies range from least recently used (LRU) to Random. Furthermore some TLBs allow certain entries to be wired down, meaning that they cannot be removed from the TLB. Typically TLB entries for kernel code are wired down.

The percentage of times that a particular page no. is found in the TLB is called the hit ratio. An 80% hit ratio, for example, means that we find the desired page no. in the TLB 80% of the time. If it takes 20ns

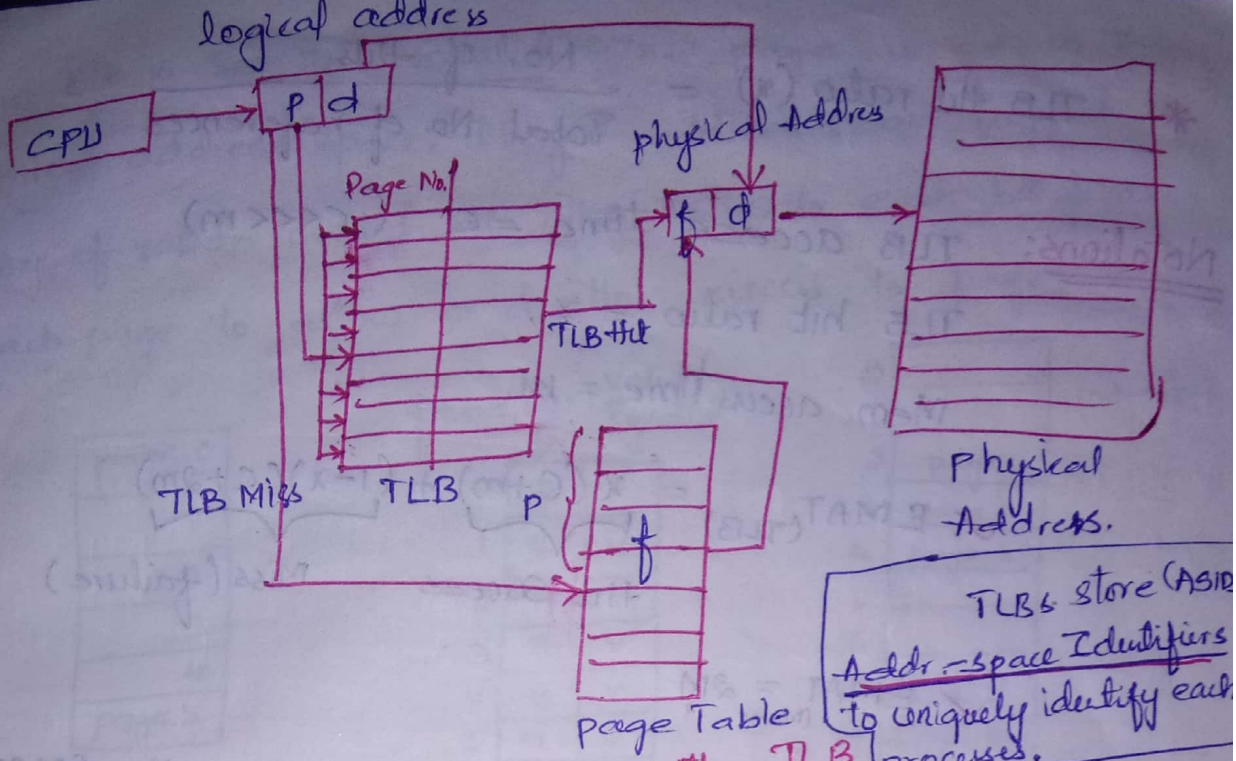


Fig: Paging Hardware with TLB

to search the TLB & 100 ns to access memory, then a mapped-mem access takes 20ns + 100ns = 120ns, When the page no. is in the TLB. If we fail to find the page no. in the TLB (20ns), then we must access mem. for the page table & frame no. (100ns) & then access the desired byte in mem (100ns), for a total of 220ns (20+100+100).

To find Effective Mem. acces Time, we wight the case by its probability

$$EAT = (0.80 * 120) + (0.20 * 220)$$

$$= 140 ns //$$

In this example, we suffer a 40% slowdown in mem. access Time (from 100ns to 140 ns).

For a 98% hit ratio, we have

$$EAT = (0.98 * 120) + (0.02 * 220)$$

$$= 122 ns //$$

This 1ed hit ratio produces only a 22% slowdown in Access Time.

* TLB Hit ratio (x) = $\frac{\text{No. of Hits}}{\text{Total No. of references}}$

Notations: TLB access time = C ($C \ll m$)

TLB hit ratio = x.

Mem. access time = m

✓ $EMAT_{(TLB)} = \underbrace{x(C+m)}_{\text{Hit Sucess}} + \underbrace{(1-x)(C+2m)}_{\text{Miss (failure)}}$

✓ $EMAT_{(Mem)} = 2M$

Eg: What hit ratio is required to reduce the EMAT from 300ns (without TLB) to 190 ns (with TLB). Assume Cache access time is 50 ns.

Ans: Given that $C = 50$ ns

$x = ?$

$2m = 300$ ns

$m = 150$ ns //

$EMAT_{TLB} = 190$ ns.

$$190 = x(50+150) + (1-x)(300+50)$$

$$= 200x + 350 - 350x$$

$$190 = 350 - 150x$$

$$x = 160/150 = 1.06 //$$

Protection: Memory protection in a paged environment is accomplished by protection bits associated with each frame. 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 & thus a legal (or valid) page. When

the bit is set to "invalid". The page is not in the logical address space. Illegal addresses are trapped by use of valid-invalid bit. The OS sets each bit from each page to allow or disallow access to page.

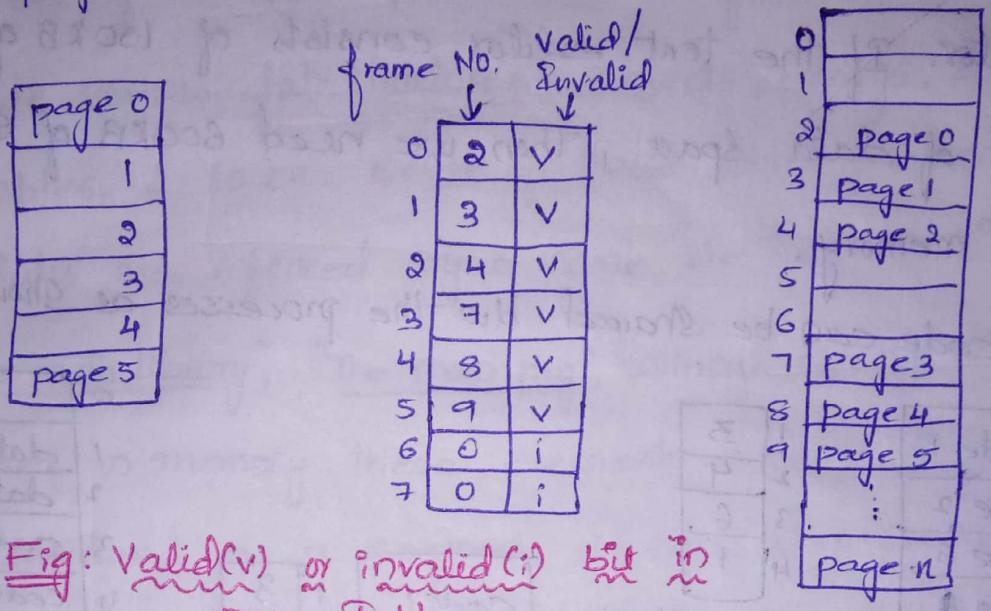


Fig: Valid(v) or invalid(i) bit in Page Table

* In this fig., the pages 0, 1, 2, 3, 4, & 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, & the computer will trap to the OS.

✓ Many processes use only a small fraction of the address space available to them. It would be wasteful in this case to create a page table with entries for every page in address range. Most of this table would be unused but would take up valuable mem. space. Some systems provide hlw, in the form of a page table length register (PTLR), to indicate the size of the page table. This value is checked against every logical address to verify that the address is in the valid range for the process.

Shared Pages: An advantage of paging is the possibility of sharing common code. This consideration is particularly important in a time-sharing environment.

* Assume 3 users/processes each of whom executes a text editor. If the text editor consists of 150KB of code & 50KB of data space, then we need 600KB of space of main memory.

The code can be shared blw the processes as shown below

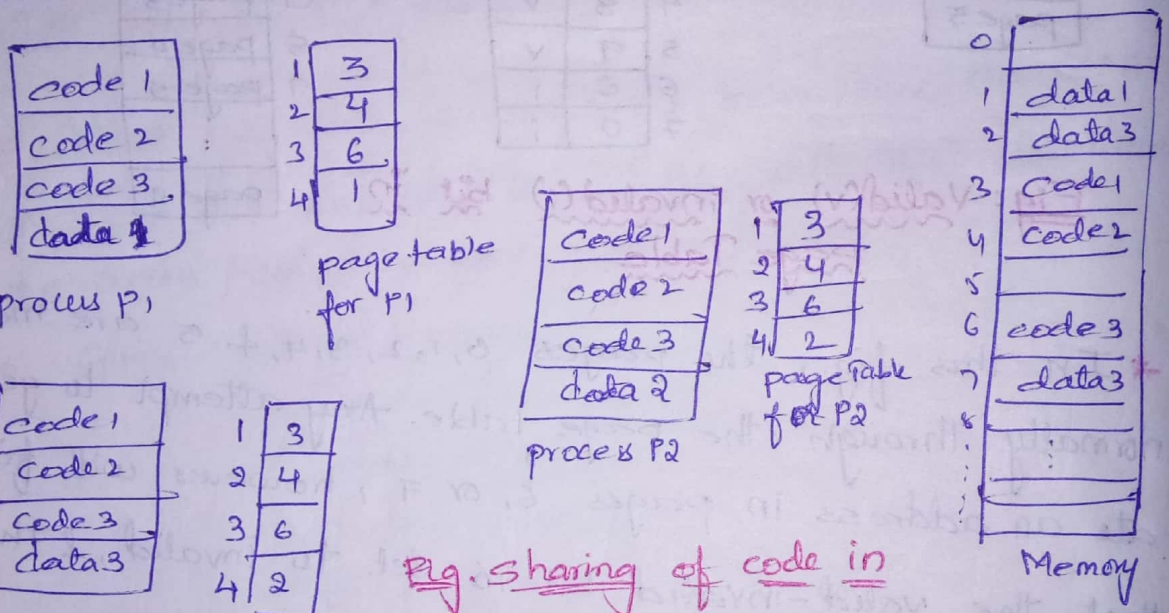


Fig. Sharing of code in Paging environment.

* The code that is being shared should be Reentrant (or pure code) (Non self modifying code) ; i.e., should be read only code
* Only one copy is kept in physical mem. which saves memory.
* Other heavily used programs like - compilers, window systems, libraries, databases & so on can also be shared.

Disadvantage of paging: Paging differentiates the user's view of mem. from the actual physical mem. The user's view of mem. (logical address space) is not same as the physical mem. The logical mem/user's view is mapped onto actual

physical memory. This mapping allows differentiation b/w logical mem & physical memory.

⑤ **Segmentation:** Users generally thk of a prg as a main prg with set of methods, procedures or funs. It may also include various datastructures : objects, arrays, stacks, variables, & so on. Each of these data elements or modules are referred by a name. We talk abt "the stack" "the math library", "The main prg", without caring what addr-esses in memory these elements occupy. Each of these is considered as a segment & each segment is of variable length, which is defined by the purpose of the segment in the prg. Elements within a segment are identified by their offset from the beginning of the segment

✓ Segmentation is a mem. mngt scheme that supports the users view of mem.. A logical address space is a collection of segments. Each segment has a name & a length. The addresses specify both the segment name & the offset within the segment. for simplicity of implementation, Segments are numbered & are referred to by a segment no., rather than by a segment name. Thus logical address consists of 2 tuples: <segment number, offset>

✓ Normally, the user prg is compiled, & the compiler automatically constructs segments reflecting the i/p prg.

* A 'c' compiler might create separate segments for the

following. 1) The code

2) Global variables

3) The heap, from which mem. is allocated

4) The stacks used by each thread

5) The standard c Library

Libraries that are linked during compile time might be assigned separate segments. The loader would take all these segments & assign them segment nos.

Eg: consider 5 segments numbered from 0 through 4. The segments are stored in physical memory as shown below. The segment table has a separate entry ofor each segment, giving the beginning address of the segment in physical mem. (or Base) & length of that segment (or limit).

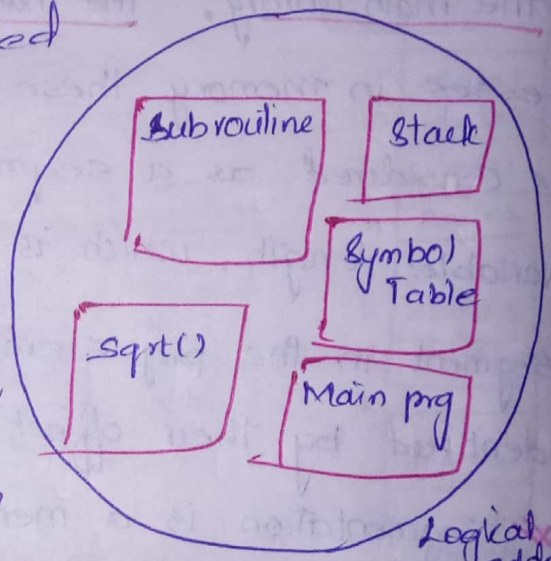
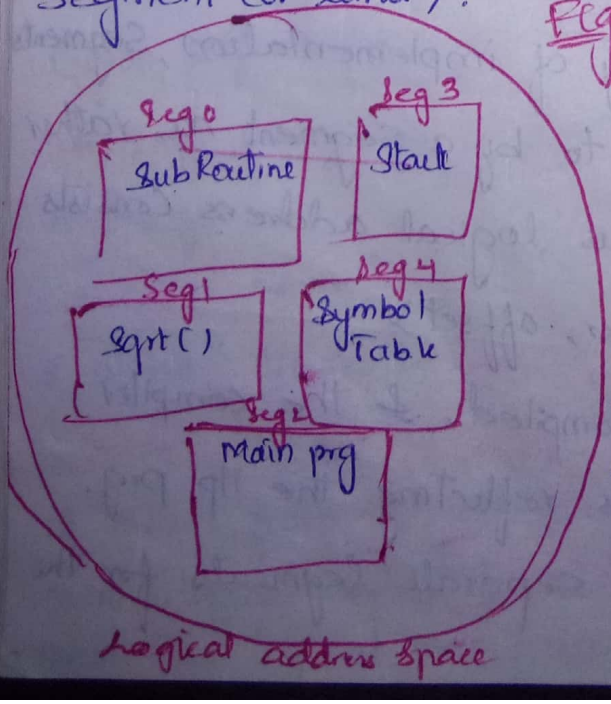


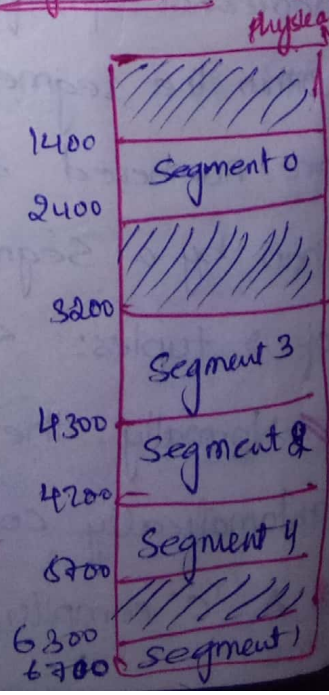
Fig: User's view of a prg.

Fig: Example for Segmentation



	Limit	Base
0	1000	1400
1	400	6300
2	400	4300
3	1100	3200
4	1000	4700

Segment table



Mapping addresses: For eg. a reference to 53 byte of segment 2 is mapped onto location 4300+53=4353.

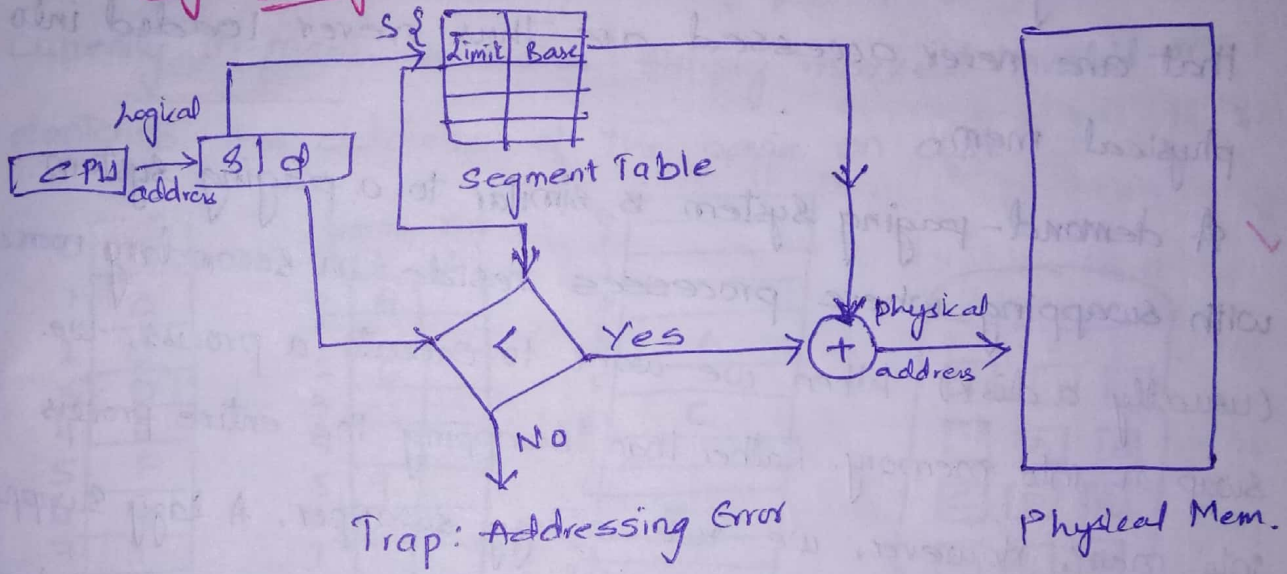
*A reference to segment 3, byte 852 ⇒ 3200+852=4052.

→Physical address = Base address of segment + offset.

And offset must be checked against limit.

∴ offset < Limit.

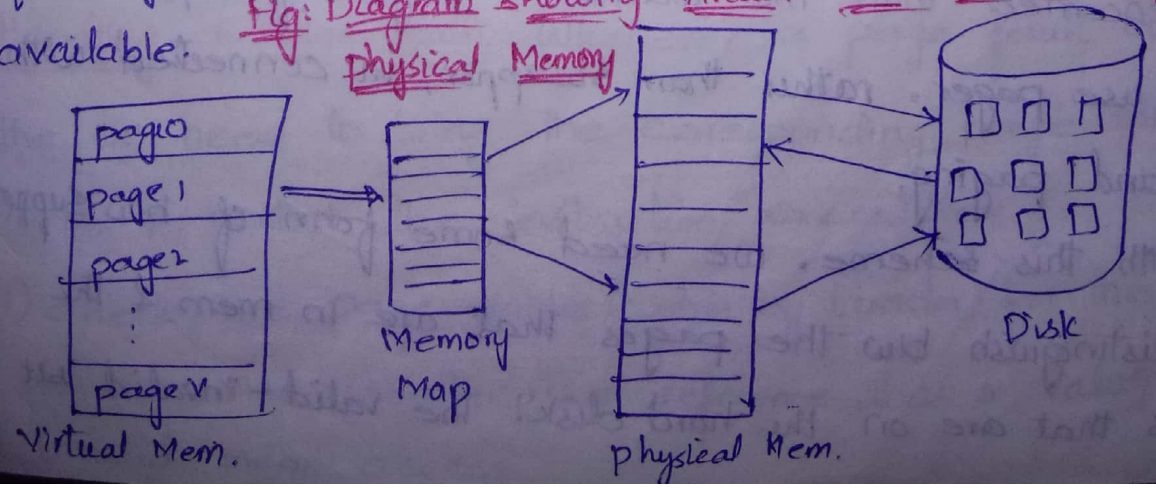
Fig: Segmentation Hardware



*Virtual Memory Management:

Virtual Mem. involves the separation of logical mem. as perceived by users from physical mem. This separation allows an extremely large virtual memory to be provided for programmers when only a smaller physical mem. is available.

Fig: Diagram showing Virtual Mem. that is larger than physical Memory



* OS creates an illusion of having larger mem. by taking advantage of disk. This illusioned mem. can be called as virtual memory.

Demand paging: Load the pages only when they are needed (Load on Demand)

✓ This is the common technique used in virtual mem. systems.

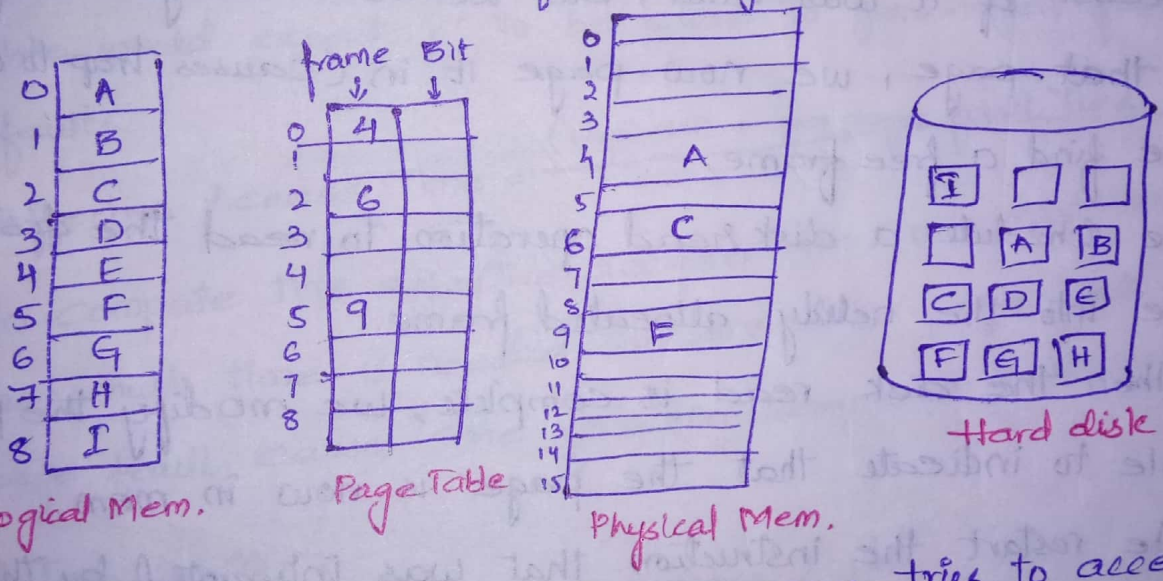
- With demand paged virtual mem., pages are only loaded when they are demanded during prg execution; pages that are never accessed are thus never loaded into physical mem.

✓ A demand-paging system is similar to a paging system with swapping where processes reside in secondary memory (usually a disk). When we want to execute a process, we swap it into memory. Rather than swapping the entire process into mem. However, we use a lazy swapper. A lazy swapper never swaps a page into mem. unless that page will be needed. Since we are now viewing a process as a seq-uence of pages, rather than as 1 large contingous addr. space, use of the term swapper is technically incorrect. A swapper manipulates entire processes, whereas a pager is concerned with the individual pages of a process. We thus use pager, rather than swapper, in connected with demand paging.

✓ With this scheme, we need some form of hlw support to distinguish blw the pages that are in mem. & the pages that are on the hard disk. The valid-invalid bit

scheme can be used for this purpose. This time, when the bit is set to "valid", the associated page is both legal & in mem. If the bit is set to "invalid", the associated page either is not valid (i.e., not in logical address space of process) or is valid but is currently on disk.

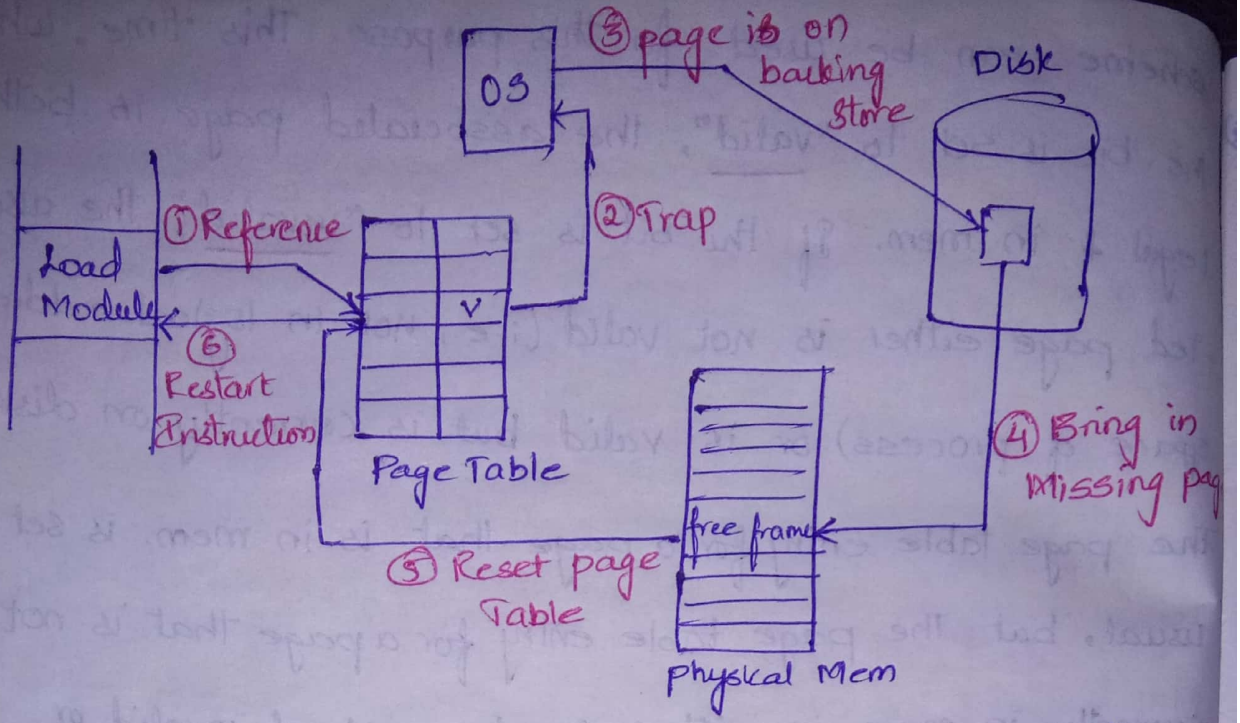
The page table entry for a page that is in mem. is set as usual, but the page table entry for a page that is not currently in mem. is either simply marked invalid or contains the address of the page on disk.



* Page Fault: Whenever the process tries to access a page that is not in mem. (marked as invalid) causes a page fault.

Handling a page fault: Whenever a page fault occurs the os need to bring the corresponding page into mem. & restart the instruction execution.

1) Check the page Table (Done by Loader) for the process to determine whether the reference was a valid or an invalid mem. access.



2) If the reference was invalid, we terminate the process. If it was valid, but we have not yet brought in that page, we now page it in (causes trap to os).

3) We find a free frame.

4) We schedule a disk read operation to read the desired page into the newly allocated frame.

5) When the disk read is complete, we modify the page table to indicate that the page is now in mem.

6) We restart the instruction that was interrupted by the trap. The process can now access the page as though it had always been in mem.

✓ Pure Demand Paging :- In the extreme case, we can start executing a process with no pages in mem. causes page fault & then the 1st page is brought to mem.

-* Performance of Demand paging : Demand paging can significally affect the performance of a computer sys.

To see why, lets compute the effective access time for a demand-paged mem. For most computer systems, mem access time ranges from 10-200ns. As long as we have no page faults, the effective access time is equa to the mem. access time. If however, a page fault occurs, we must first read the relevant page from disk & then access the desired word.

Let 'p' be the probability of a page fault ($0 \le p \le 1$), & 'm' is the mem. access time.

We would expect 'p' to be close to zero - i.e., few page faults.

Effective Access Time = $(1-p) * m + p *$ page fault time

To compute the effective access time, we must know how much time is needed to service a page fault. A page fault causes the following sequence to occur:

1. Trap to OS.
2. Save the user registers & process state.
3. Determine the interrupt was a page fault.
4. Check the page reference was legal & determine the location of the page on disk.
5. Issue a read from the disk to a free frame:
   a) Wait in a queue for this device until a read request is serviced.
   b) Wait for device seek & or latency time.
   c) Begin the transfer of the page to a free frame.
6. While waiting, allocate the CPU to some other process/ user.

⑦ Receive an interrupt from I/O disk subsystem. (I/O comp
⑧ Save the registers & process state for other process/user.
⑨ Determine that interrupt was from the disk.
⑩ Correct the page table & other tables to show that the desired page is now in mem.
⑪ Wait for the CPU to be allocated to this process again.
⑫ Restore the user registers, process state, & new page table & then resume the interrupted instruction.

* Note that all these steps may not be necessary.

✓ Generally, a page-switch time is close to 8ms & also consider mem. access time as 100ns.

Then Effective access Time = $(1-p) * 100ns + p(8)ms$

$= 100 - 100p + 8,000,000ns$

$= 100 + 7,999,900 * p$ ns

EAT is directly proportional to the page fault rate.

✓ If one access out of 1,000 causes a page fault, the Effective Access time is

$EAT = 100 + 7999900 * \frac{1}{1000}$

$= 8199.9 ns$

$= 8.2$ milliseconds //

---

* By Taking $m = 200$

$EAT = (1-p) * 200 + p(8)$ ms

$= (1-p) * 200 + 8000000 * p$

$= 200 + 7,999,800p$

$EAT \propto P$

If 1 access out of 1000 causes a page fault,

$$EAT = 200 + 7999800 * \frac{1}{1000}$$

$$= 8199.8 \text{ nanosec.}$$

$$\approx 8.2 \text{ microseconds}$$

Without demand paging = 200ns $= \frac{200}{1000}$ micro sec

$= 0.2$ microsec.

The computer which uses demand paging will be slowed down by a factor of 40.

✓ Percentage slow down.

Actual access time = 200ns

with demand paging = 8.2 μs

$$\text{Degradation percentage} = \frac{8.2\,\mu s}{200\,ns} \times 100$$

$$= \frac{8.2 \times 1000}{200} \times 100$$

$$= 4100 \text{ i.e., } 41\%$$

✓ If we want to bring down the degradation % to less than 10% (find demand paging access time).

$$10 = \frac{x * 100}{200} = \frac{x * 1000}{200} \times 100 \text{ (μs)}$$

$x = 0.02$ μs

Degradation in Demand paging access time.

* Total access time = Mem. access time + Degradation

$= 0.2 + 0.02$ μs

$= 0.22$ μs $= 220$ ns

Lets find out the page fault ratio

$200 + 7999800 * p < 220$

$7999800p < 20$

$p < 0.0000025$

Page Replacement: Page replacement takes the following approach. If no frame is free, we find one that is not currently being used & free it. We can free a frame by writing its contents to swap space & changing the page table to indicate that the page is no longer in memory. We can use a free frame to hold the page for which the process faulted.

Basic Algorithm:

1) Find the location of the desired page on the disk.
2) Find a free frame:
   a. If frame is free use it.
   b. If there is no free frame, use a page replacement algorithm to select a victim frame
   c. Write the victim frame to the disk, change the page & frame tables accordingly.
3) Read the desired page into the newly freed frame; change the page & frame tables.
4) Restart the user process.

* Notice that, if no frames are free, 2 page transfers (one out & one in) are required. This situation effectively doubles the page-fault service time & increases the effective access time accordingly.

Solution: We can reduce the overhead by using a modify bit (or dirty bit). When this scheme is used, each page or frame has a modify bit associated with it in the hw. The modify bit for a page is set by

the hlw whenever any word or byte in the page is written into, indicating that the page has been modified. When we select a page for replacement, we examine its modify bit. If the bit is set, we know that the page has been modified. since it was read in from the disk. In this case, we must write the page to the disk. If the modify bit is not set, however the page has not been modified since it was read into mem. In this case, we need not write the mem. page to the disk, its already there.

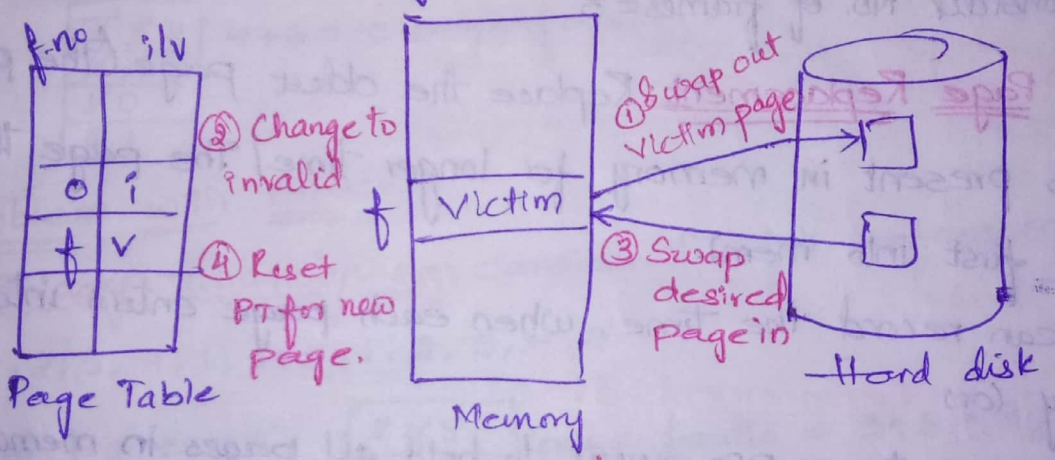


Fig. Page Replacement

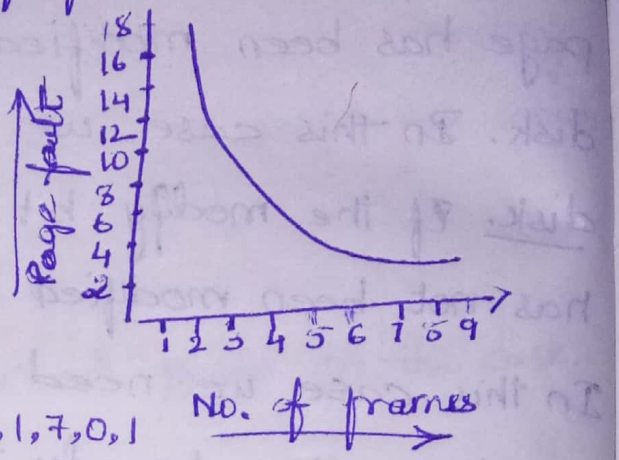
Problems in implementing Demand paging:

1) Frame-Allocation Algs: If we have multiple processes in mem., we must decide how many frames to allocate to each process.

2) Page Replacement Algs: When free frames are not available, we must select the frames that are to be replaced.

* We have to select a page replacement alg. in such a way that it gives lowest page fault rate

Page faults Vs No. of frames: Generally, when the no. of frames allocated to a process is increased, then the page faults will be fed. The following is the general graph which represents page faults vs No. of frames.



Page Replacement Algorithms:

We illustrate the page replacement algorithms with the help of following reference string.

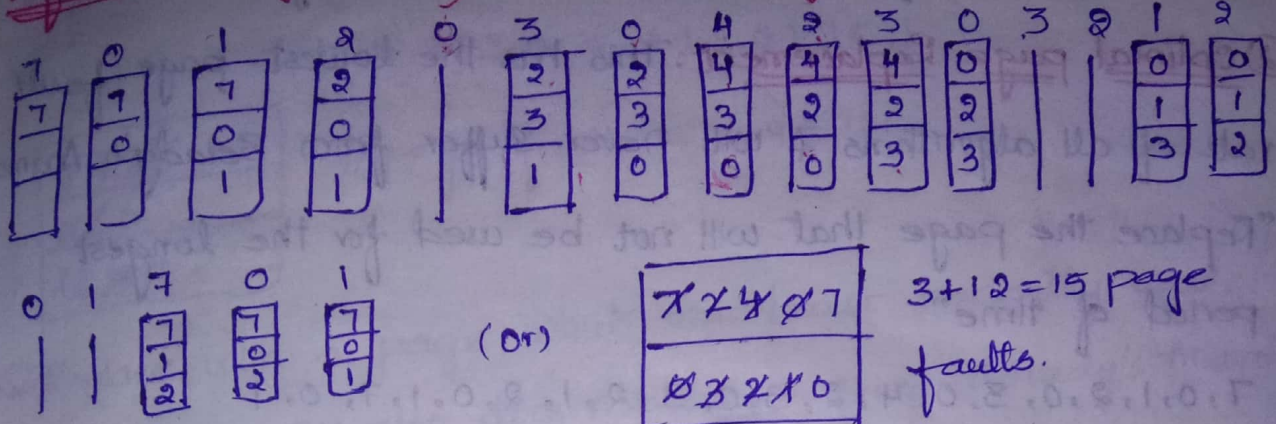
7, 0, 1, 2, 0, 3, 0, 4, 2, 3, 0, 3, 2, 1, 2, 0, 1, 7, 0, 1

We consider No. of frames = 3

1) **FIFO Page Replacement:** Replace the oldest page (The page that is present in memory for longer time / The page that arrived first into mem).

- ✓ We can record the time, when each page enters into memory (or)
- ✓ We can create a FIFO queue to hold all pages in memory. We replace the page at the head of queue, when the page is brought into the mem, we insert it at the tail of the queue.
- ✓ For our example reference string, our 3 frames are initially empty. The first 3 references (7, 0, 1) causes page faults & are brought into mem. (these empty frames). The next reference (2) replaces 7, because 7 was brought in first. Since (0) is the next reference & (0) is already in mem., we have no fault for this reference.

Reference String



(or)

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

3+12=15 page faults.

✓ The FIFO page replacement algorithm is easy to understand & prg. However its performance is not always good.

* Repeat above with 4 frames.

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

4+6=10 page faults.

Problem with FIFO:

* Belady's Anamoly: Eg: consider the foll. reference string.

1, 2, 3, 4, 1, 2, 5, 1, 2, 3, 4, 5

i) # frames = 3

1	2	3	4	1	2	5	1	2	3	4	5
	1	2	3	4	1	2	5	1	2	3	4
		1	2	3	4	1	2	5	1	2	3

page faults = 3+6 = 9 //

ii) # frames = 4

1	2	3	4	1	2	5	1	2	3	4	5
	1	2	3	4	1	2	5	1	2	3	4
		1	2	3	4	1	2	5	1	2	3
			1	2	3	4	1	2	5	1	2

page faults = 4+6 = 10 //

Note that no. of page faults for 4 frames (10) is greater than the no. of page faults for 3 frames (9). This most unexpected result is known as Belady's Anamoly.

For some page replacement algs, the page fault rate may increase as the no. of allocated frames increases.

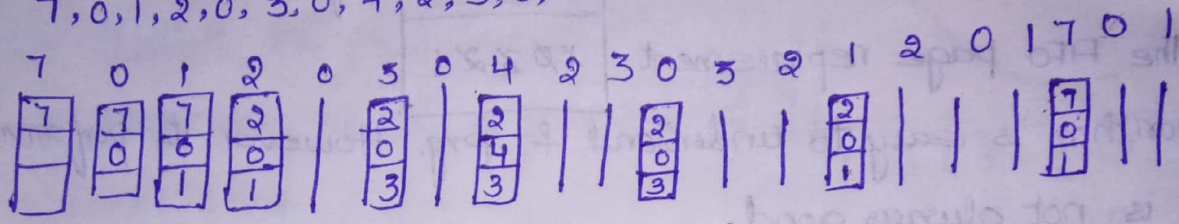
We would expect that giving more mem. to a process would

improve its performance but its not always true.

② Optimal page Replacement: This has the lowest page fault rate of all algorithms & will never suffer from Belady's Anamol

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

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



# of page faults = 9 (or) = 3 + 6 = 9 page faults.

* Disadvantage / Drawback:

The optimal page replacement alg. is difficult to implement because it requires the future knowledge of the reference string. Hence it is mainly used for comparison studies.

* LRU Page Replacement: It's an approximation of optimal page replacement algorithm. (Least Recently Used)

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

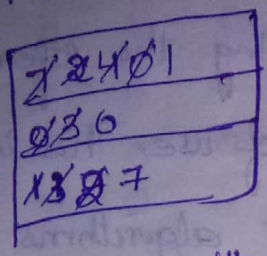
✓ We can think of this strategy as the OPT algorithm looking backward in time, rather than forward.

✓ If we let S^R be the reverse of a reference string S, then the page fault rate for the OPT alg. on S is the same as the page fault rate for the OPT alg on S^R.

Similary, page fault rate of LRU on S = page fault rate of LRU on S^R.

Reference String:

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



page faults = 3+9 = 12 //

- Replace with a page which was least recently used.
- Like OPT, LRU also does not suffer from Belady's Anamoly.
- OPT & LRU both are called as stack algorithms.

A stack alg is a alg for which it can be shown that the set of pages in mem for n frames is always a subset of the set of pages that would be in mem. with n+1 frames. For LRU, the set of pages in mem. would be the most recently referenced pages. If the no. of frames is fixed, these n pages will still be the most recently referenced & so will still be in mem.

- Implementing LRU → ① counter, ② stack

4) LRU Approximation Page Replacement: Few computer systems provide sufficient hdw support for true LRU page replacement. Some systems provide no hdw support, & other page replacement algs must be used. Many systems provide some help in the form of reference bits

The reference bit for a page is set by the hdw whenever that page is referenced. (either a read or write to any byte in the page). Reference bits are associated with each entry in the page table.

Initially, all bits are cleared (set to zero) by the OS. As a user process executes, the bit associated with

each page reference is set (to one) by the hlw. After Sometime we can determine which pages have been used & which have not been used by examining the reference bits, although we do not known the order of use. This inf. is the basis for many page replacement algorithms that approximate LRU Replacement.

i) **Additional-Reference-bits Algorithm**: We can gain additional ordering inf. by recording the reference bits at regular intervals. We can keep an 8-bit byte for each page in page table in mem. At regular intervals (Say every 100 ms), a timer interrupt transfers control to the OS. The OS shifts the reference bit for each page into the high-order bit of its 8-bit byte, shifting the other bits right by 1 bit & discarding the low-order bit.

eg: Lets consider 4 bits (which includes one reference bit & 3 used bit (i.e., U3 U2 U1 U0) Also assume that there are 5 frames.

Reference String: 3,2,3 T 8,0,3 T 3,0,2 T 6,3,4,7

1) **Initial State**:

P	U3	U2	U1	U0
-	0	0	0	0
-	0	0	0	0
-	0	0	0	0
-	0	0	0	0
-	0	0	0	0

* During first time interval all the 'U' bits are equal to zero & can place pages anywhere

ii)

P	U3	U2	U1	U0
3	1	0	0	0
2	1	0	0	0
-	0	0	0	0
-	0	0	0	0
-	0	0	0	0

Place 3 & 2 pages in the 1st 2 frames & set U3 to 1 for both.

Third page reference is 3 & its already present in the frame hence no need to load again.

iii) After first time interval, the bits are shifted to right 1 position.

P	U3	U2	U1	U0
3	0	1	0	0
2	0	1	0	0
-	0	0	0	0
-	0	0	0	0
-	0	0	0	0

During 2nd time interval pages 8, 0, 3 are referenced. pages 8 & 0 are loaded into empty positions & their U3 bits are set to 1. Then the U3 bit for page 3 is also set to 1.

iv)

P	U3	U2	U1	U0
3	1	1	0	0
2	0	1	0	0
8	1	0	0	0
0	1	0	0	0
-	0	0	0	0

At the end of 2nd time interval all 'U' bits are shifted right by 1 position.

v)

P	U3	U2	U1	U0
3	0	1	1	0
2	0	0	1	0
8	0	01	0	0
0	0	01	0	0
-	0	0	0	0

During 3rd time interval pages 3, 0 & 2 are referenced, so U3 is set for pages 3, 0 & 2.

P	U3	U2	U1	U0
3	1	1	1	0
2	1	0	1	0
8	0	1	0	0
0	1	1	0	0
-	0	0	0	0

At the end of 3rd time interval, all 'U' bits are shifted by 1 positi

P	U3	U2	U1	U0
3	0	1	1	1
2	0	1	0	1
8	0	0	1	0
0	0	1	1	0
-	0	0	0	0

During 4th time interval pages 6, 3, 4 & 7 are refered. First page 6 is loaded & its U3 is set to 1. Then the U3 bit for page 3 is set to 1.

P	U3	U2	U1	U0
3	1	1	1	1
2	0	1	0	1
8	0	0	1	0
0	0	1	1	0
6	1	0	0	0

✓ Continuing in the same time interval, when page 4 is required mem. is full, so we choose the page with the lowest 'U' value, which is page 8. page 4 replaces page 8 & U bits are set to 1000. Nxt page 7 is required, replace it by lowest U value ie, 2.

P	U3	U2	U1	U0
3	1	1	1	1
2	0	0	0	1
4	1	0	0	0
0	0	1	1	0
6	1	0	0	0

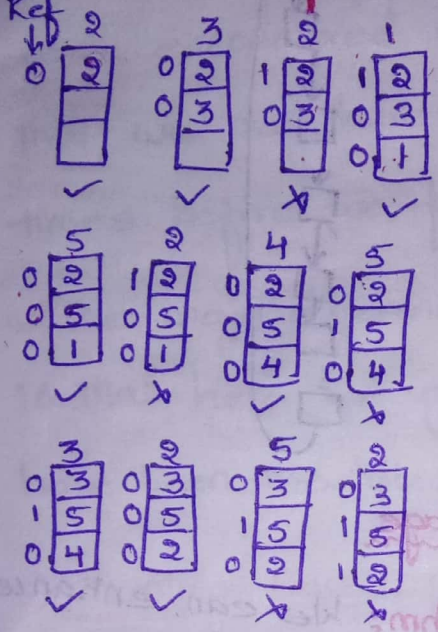
⟹

P	U3	U2	U1	U0
3	1	1	1	1
7	1	0	0	0
4	1	0	0	0
0	0	1	1	0
6	1	0	0	0

* Second chance Algorithm: FIFO Variant of 2nd chance Algorithm (SCA).

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

Frames = 3, reference bit is taken (No used bits)



✓ Since '2' is again referenced & its already in mem. we are giving a second chance/reward by setting reference bit to 1.

✓ We are not going replace 2 by 5 because its already got 2nd chance & it's set to 1 hence look at the pages which are having '0' as reference bit & follow FIFO & as '2' is getting its second chance change reference bit to '0'.

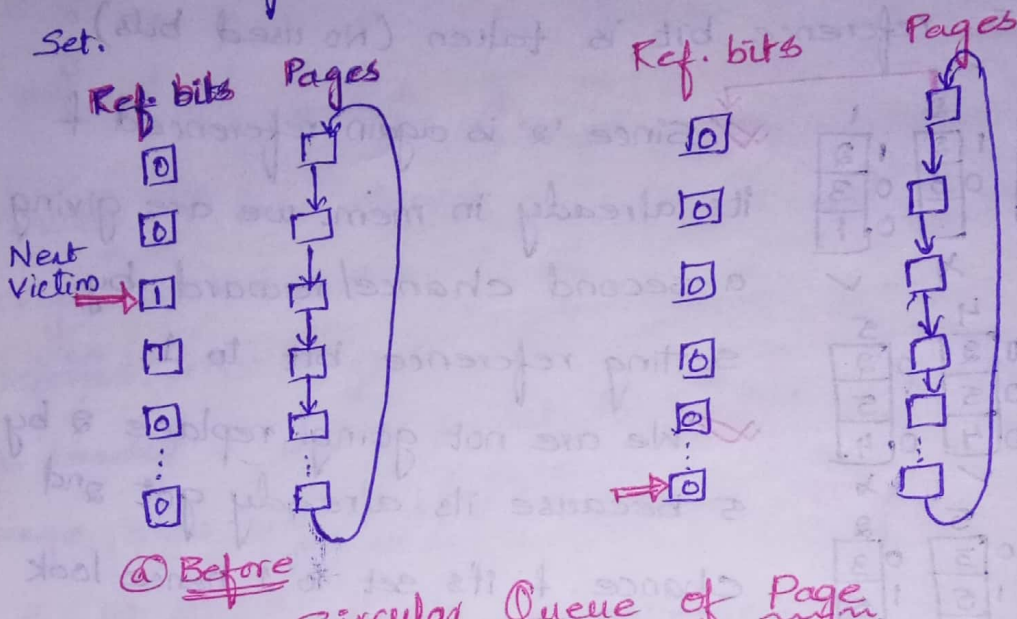
✓ As a reward for saving page fault we will set reference bit to 1.

* Implementing Second chance Algorithm: (Clock Algorithm)

✓ Circular queue can be used. A pointer (i.e., a hand on the clock) indicates which page is to be replaced next. When a frame is needed, the pointer advances until it finds a page with a '0' reference bit set. As it advances, it clears the reference bits (set to 1 to 0) i.e., giving second chance to those pages. Once a victim page is found, the page is replaced, & the new page is inserted in the circular queue in that position.

✓ In the worst case, when all the bits are set, the pointer cycles through the whole queue, giving each

page a second chance. It clears all the reference bi
before selecting the next page for replacement secon
chance degenerates to FIFO replacement if all bits are
set.



(a) Before

Circular Queue of Page

iii) Enhanced Second-chance Algorithm: We can enhance the second-chance algorithms by considering the reference bit & the modify bit as an ordered pair. With these 2 bits, we have the following 4 possible cases.

1) (0,0): Neither recently used nor modified - Best page to replace.

2) (0,1): Not recently used but modified - not quite as good, because the page will page need to be written out before replacement.

3) (1,0): Recently used ~~but modified~~ clean - Probably will be used again.

4) (1,1): Recently used & modified - Probably will be used again soon, & the page will be need to be written out to disk before it can replaced.

Each page will be in 1 of these 4 classes. When page replacement is called for, we use the same