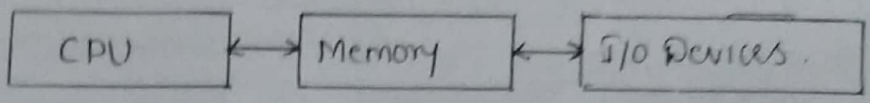
3 MEMORY MANAGEMENT

Introduction:

Memory is central to the operation of Modern computer system.



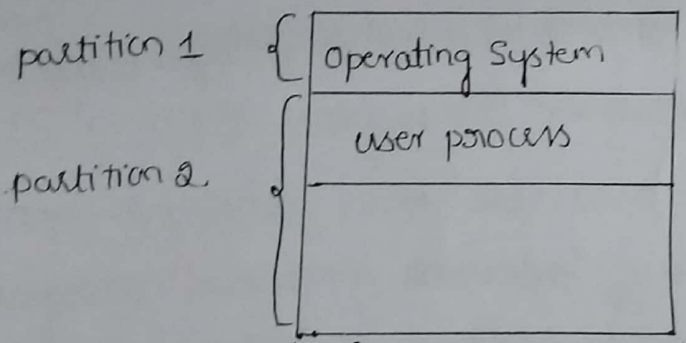
Basic structure of computer system.

→ CPU & I/O devices interact with memory. Memory is large array of words or bytes with its own address.

→ In a mono programming or uni-programming system, the main memory is divided into two parts.

(1) one part for operating system

(2) Another for a job is currently executing.



→ partition 2 is allowed for user process. But some part of the partition 2 wasted. It is indicated by blocked lines in the fig.

→ In multiprogramming environment the user space is divided into number of partitions.

→ Each partition is for one process.

→ The task of subdivision is carried out dynamically by the OS. This task is known as Memory Management.

→ The efficient memory management is possible with multiprogramming

→ A few process are in main memory, the remaining process are waiting for I/O and the process will be idle.
→ Thus memory needs to be allocated efficiently to pack as many process into memory as possible.

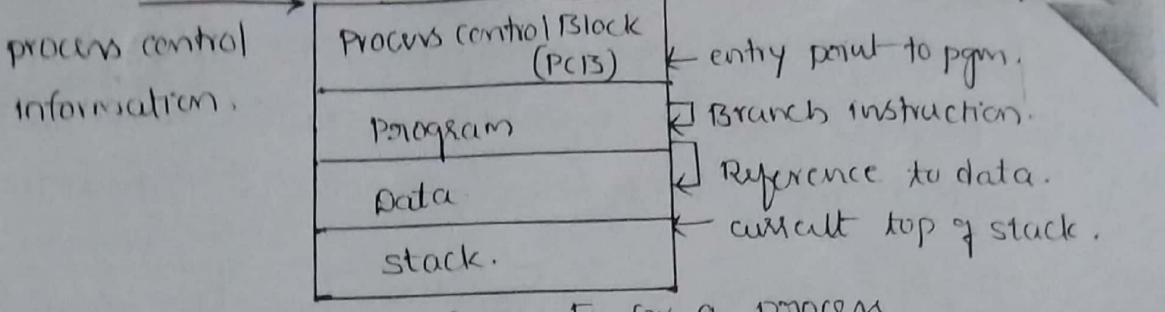
⇒ FUNCTIONS OF MEMORY MANAGEMENT:

→ To know the status of each memory location, whether the memory location is allocated or free.
→ Determining the factors of memory policy.
→ Allocation of memory
→ Deallocation of memory.

⇒ Memory Management Requirements:

→ There are several requirements to be satisfied to handle memory management effectively:
(i) Relocation (ii) protection (iii) sharing
(iv) Logical Organisation (v) physical Organisation.

⇒ (i) Relocation: The available main memory is generally shared among a number of processes in a multiprogramming system
→ It is not possible for programmer to know in advance which other programs will be resident into main memory at the time of execution of a program.
→ In addition we would like to swap active processes in & out of main memory to maximize processor use by providing a large pool of ready processes to execute.
→ Once a program has been swapped out of disk, it would be quit limiting to declare that when it is next swapped back in & must be placed in the same main memory.



Addressing requirements for a process.

→ For all this same addressing is required.

→ Each program will have branches and may need to access same data also. All these addresses must be known to the OS.

→ Some processor hardware and operating system software must be able to translate these memory reference to actual physical locations.

→ During all this swapping in and out of memory the process image automatically must be swapped in & out.

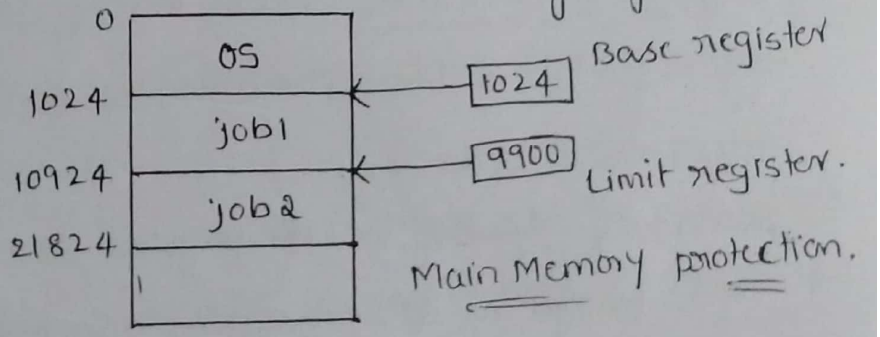
(ii) Protection: The word protection means provide security from unauthorized usage of memory.

→ The Operating system can protect the memory with the help of base & limit registers.

→ Base registers consisting the starting address of the process.

→ Limit registers specifies the boundary of that job.

→ so the limit registers is also said to be fencing registers.



Main Memory protection.

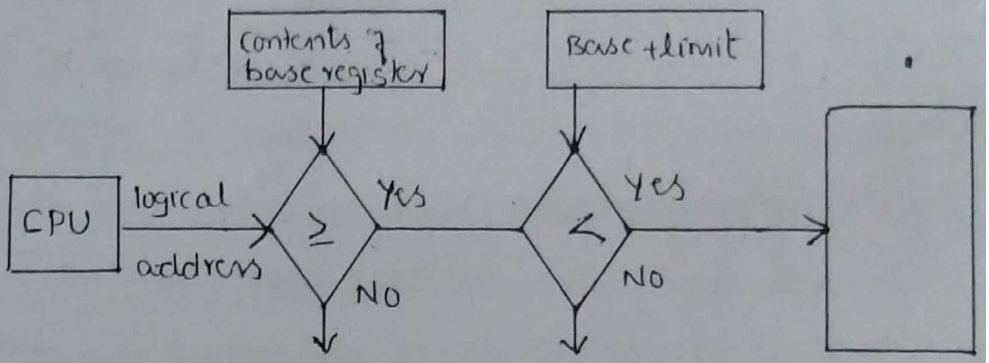
→ The base register holds the smallest legal physical memory address, the limit registers contain the size of the process.

→ The below figure explains the H/w protection mechanism with base & limit registers.

→ If the logical address is greater than the contents of a base

register, it is authorized access, otherwise it is trap O.S

→ The physical address (logical + base) is less than the base + limit it causes no problem otherwise it is trap to be OS.



Memory protection with base & limit

⇒ (iii) Sharing: protection mechanism are required at the time of access. the same portion of main memory by several processes

→ Accessing in the same portion of main memory by number of process is said to be sharing.

→ suppose to say it number of process are executing same program. It is advantageous to allow each process to access same copy of the program rather than its have its own copy.

→ If each process maintain a seperate copy of that program a lot of main memory will waste.

⇒ (iv) Logical organisation: Main memory is a computer system is organized as a linear, address space that consists of a sequence of bytes.

→ Secondary Memory at its physical level is similarly organised

→ Most programs are organised into modules. some of which are unmodifiable and some of which contain data that may be modified.

→ If the operating system & computer hardware can effectively deal with user programs & data.

(a) Physical organisation: A computer memory is organized in atleast two levels.
(a) Main memory (2) secondary memory.

→ (a) Main memory: It provides fast access at relatively high cost. It is volatile. It means it does not provide permanent storage.

→ (b) Secondary Memory: It is slower and cheaper than main memory. & it is usually non volatile.

→ In this memory a large capacity can be provided to allow for long term shortage of programs and data.

→ while a smaller main memory holds programs & data currently in use

⇒ ADDRESS BINDING:

→ programs & data items stores on a secondary storage disk as a binary executable file.

→ For executing the programs, it is bought into the memory and placed within a process.

→ The collection of processes on the disk that is waiting to be bought into memory for execution forms the input queue.

→ Input Queue is used for selecting one of the process for execution and loads into memory.

→ As a process executed, its access data from memory & process will terminate

→ The memory space become free. Most of the OS allows the user process to reside in any part of the physical memory.

→ user program consists of various types of instructions & data.

→ It also accesses various library files while executing.

→ source program uses symbolic addresses.

→ A compiler will bind these symbolic addresses to relocatable addresses.

→ Loader will in turn bind these relocatable addresses to absolute addresses.

→ Each binding is a mapping from one address space to another.

→ Binding of each instruction or data to memory addresses can be done at any step.

→ (1) Compile time: process will reside in memory at compile time.

→ It generates absolute code at compile time binding.

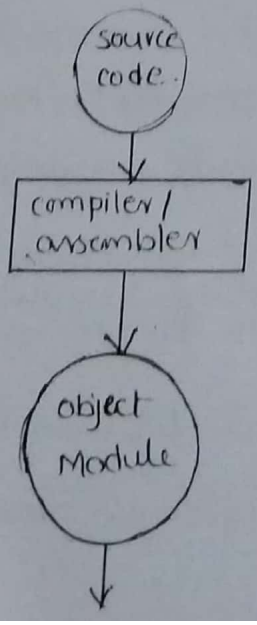
→ After compiling the process, if the ~~process~~ memory location is changed, then it is necessary that the code must be recompiled.

→ (2) Load time: Final binding is delayed until load time.

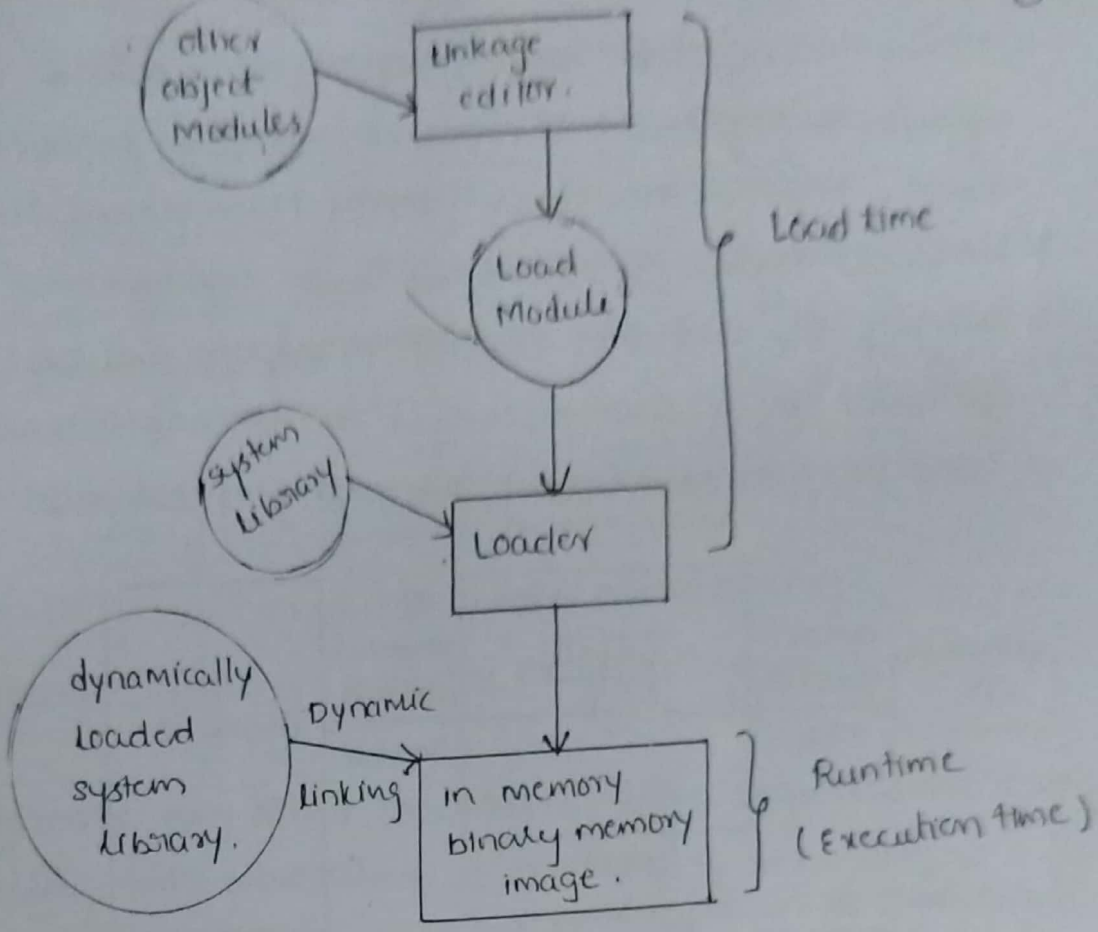
→ If the starting address changes, it needs only to reload the user code to incorporate this changed value.

→ (3) Execution time: process may be moved from one memory to another memory segment in execution. then binding must be delayed until run time.

→ Special hardware must be available. Most general purpose operating systems use execution time binding method.



7sep



Different steps for processing user program.

⇒ LOGICAL VERSUS PHYSICAL ADDRESS SPACE

→ Logical address is generated by CPU.

→ physical address is only the address of main memory & it is loaded into the memory address register.

→ Compile & load time address - binding methods generate same logical address & physical address.

→ Execution - time address - binding scheme results in different logical & physical addresses.

→ Generally logical address is referred as a virtual address

→ Logical address space set of all logical addresses generated by programs is a logical address space.

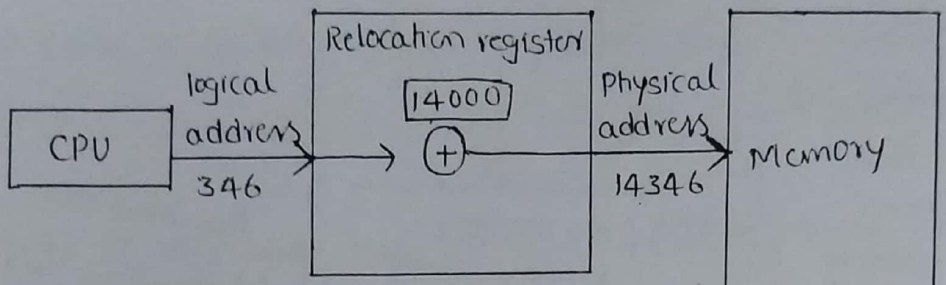
→ physical address space is a set of all physical addresses corresponding to base logical addresses.

→ Memory management unit mapping at run time from virtual to physical addresses is done by a hardware device. This H/w device is Memory Management Unit.

→ Relocation register is also called base register.

→ Value of the relocation register is added to every address generated by a user process at the time it is sent to memory.

→ User program never sees the real physical addresses.



Dynamic relocation using a relocation register.

→ Dynamic relocation implies that mapping from the virtual address space to the physical address space is performed at run time. usually with some hardware assistance.

→ Relocation is performed by hardware and is invisible to user

→ Dynamic relocation makes it possible to move a partially executed process from one area of memory into another without affecting.

⇒ Dynamic Loading: It is used for better memory space utilization

→ User program size is large as compared to the memory size.

→ program / size process are dynamically loaded into memory as per required.

→ With dynamic loading, a routine is not loaded until it is called.

→ All routines are kept on storage disk in a relocatable load form

→ The main program is loaded into memory & is executed.

→ Advantage of dynamic loading is that an unused routine

is never loaded. This method is particularly useful when large amount of code are needed to handle infrequently occuring cases.

⇒ Dynamic loading does not require special support to the OS. OS may help the programmer by providing library routines to implement dynamic loading.

⇒ Dynamic LINKing & Shared Libraries:-

→ Some operating system support only static linking.

→ In a dynamic Linking, only the main program is initially loaded.

→ If it reference any other procedures, those segments are loaded & addressing link established at the time of reference.

→ The linkage of subroutine is postponed until a reference is explicitly made

→ There are several ways to accomplish dynamic linking with dynamic linking a stub is included in the image for each library routine reference.

→ This stub is small piece of code that indicates how to locate the appropiate memory resident library routine.

⇒ SWAPPING:

→ Swapping is a technique of temporarily removing inactive program from the memory of a system.

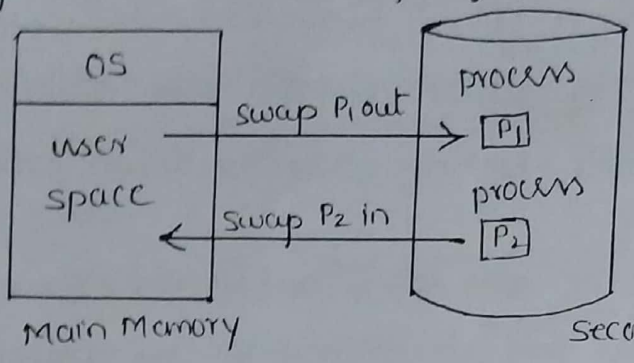
→ It removes the process from the primary memory when it is blocked & deallocating the memory.

→ This free memory is allocated to other processes.

→ For example when process P_1 requests an I/O operation, it becomes blocked & will not return to the ready state for a relatively long period of time.

→ Process manager places the process P_1 into a blocked state, then the memory manager will decide whether to swap the process from primary memory to secondary memory. Process P changes the state from blocked to the ready state then process P_2 is swapped out.

→ Memory manager can swap the address space back into primary memory either immediately ~~if~~ primary memory is available.



→ When process is swapped out, its executable image is copied to secondary memory

→ when the process is swapped back into available primary memory, the executable image that ~~was~~ has swapped out is copied into the new block allocated by the memory manager.

→ swapped out process will be swapped back into the same memory space that is occupied previously.

→ It also depends upon method of binding.

(1) If the address binding is done at load time, then the process is moved to same location of previous one.

(2) If the address binding is done at execution time, then a process can be swapped into a different memory space. This is because of the physical address which is computed during execution time.

Contigous Memory Allocation:

→ Single contigeuous allocation is a simple memory management scheme that requires no special hardware features.

→ For allocating memory it is ~~advised~~ divided into a no of fixed

sizes partition.

→ Multiprogramming depends on the no of partitions.

→ Each partition contains exactly only one process.

→ Multiple partition method is used in multiprogramming environment

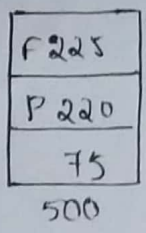
→ If the partition is free, process is selected from the input queue & is ~~selected~~ loaded into the free partition of memory.

→ when the process terminates the memory partition become available for another process.

→ The OS keeps the record of memory allocation & deallocation in the form of table.

→ Memory is available for user processes.

For Example: If there are 500 K bytes of memory a simple operating system may require 75K bytes. Leaving 425 K bytes allocated for user jobs (process)



→ If a typical process requires only 200 K bytes. Then 225 K bytes of memory are unused (free)

→ This unused memory is used for newly arrived process. This process is continued upto switch off the computer system.

→ The set of holes is searched to determine which hole is best to allocate.

→ Memory Management uses three algorithms for selecting free holes:-

1) first fit
2) Best fit
3) worst fit.

1) First fit:- Allocate first hole that is big enough. first fit begins to scan memory from the beginning & chooses the first available block that is large enough.

2) Best fit: Best fit chooses the block that is closest in size to the request. It allocates the smallest hole that is big enough.

(3) Worst Fit: Allocate the largest hole. It searches the entire list.

→ first & best fit are most popular algorithms for dynamic memory allocation.

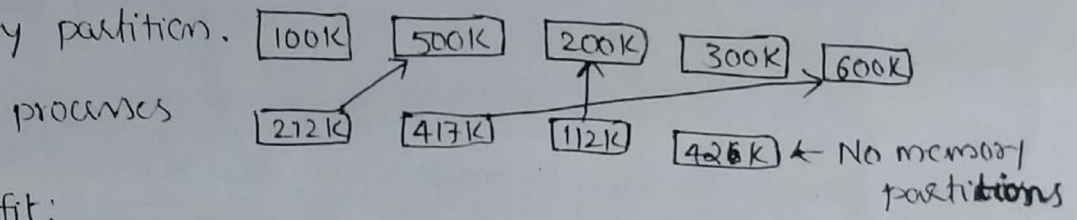
→ first fit is generally faster.

→ Best fit searches the entire free list to find the smallest free block.

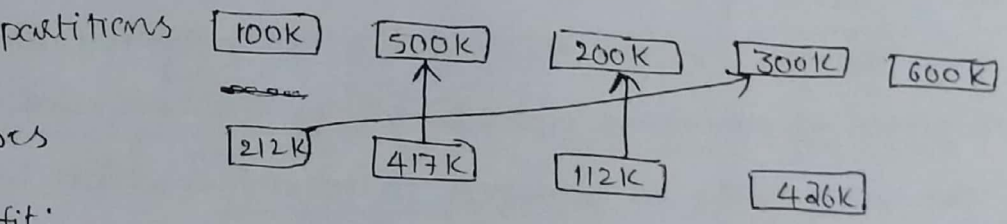
→ worst fit reduces the rate of production of small holes.

Example: Given memory partitions of 100K, 500K, 200K, 300K & 60K how would each of the first hit & best fit & worst fit algorithms place processes of 212K, 417K, 112K & 426K which algorithm makes the most efficient use of memory.

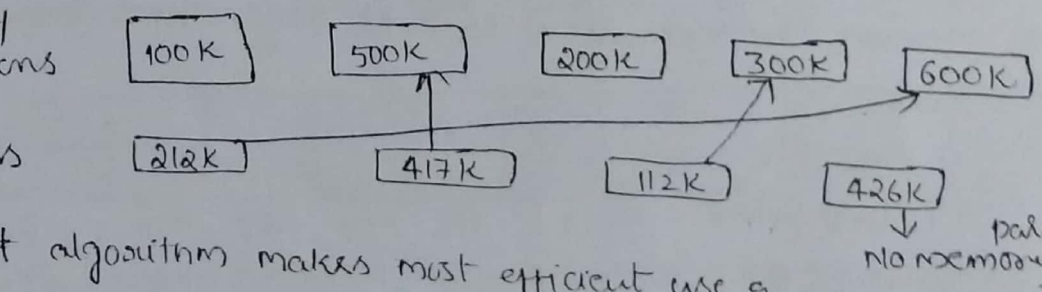
Sol: i) First fit



ii) Best fit:



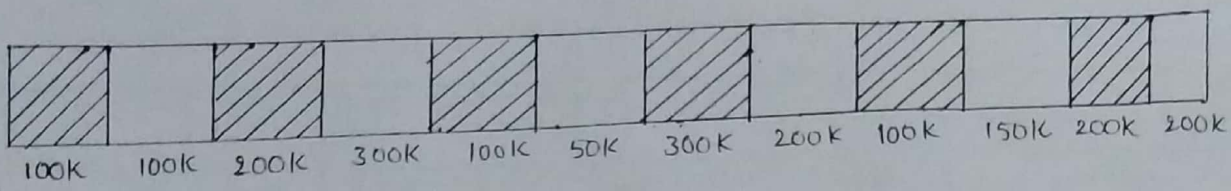
iii) worst fit:



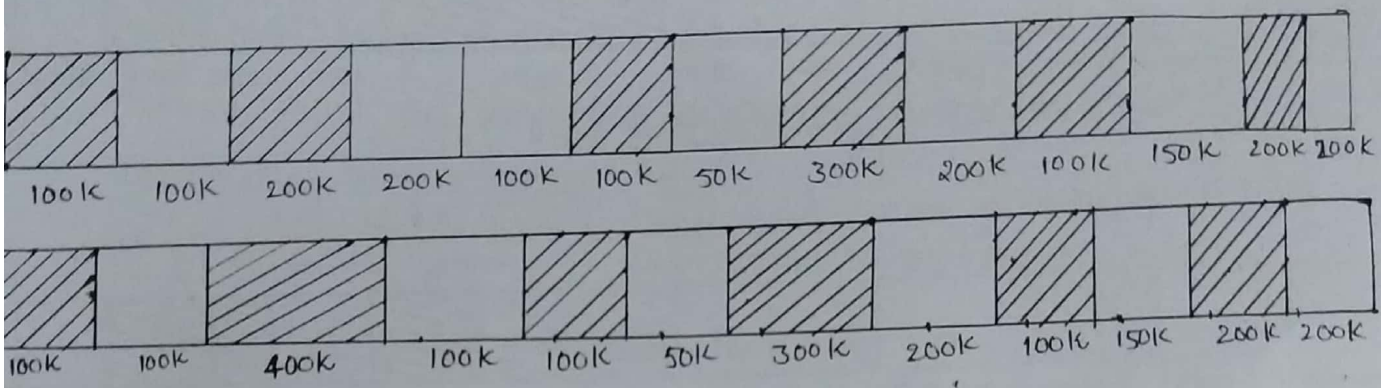
Best fit algorithm makes most efficient use of memory.

Example: 2: Memory partitions with used and unused block are given in the following. Additional requests 200K, 100K & 250K are received. At what starting address will each of the additional requests be allocated for the following allocation.

1) first fit 2) Best fit.

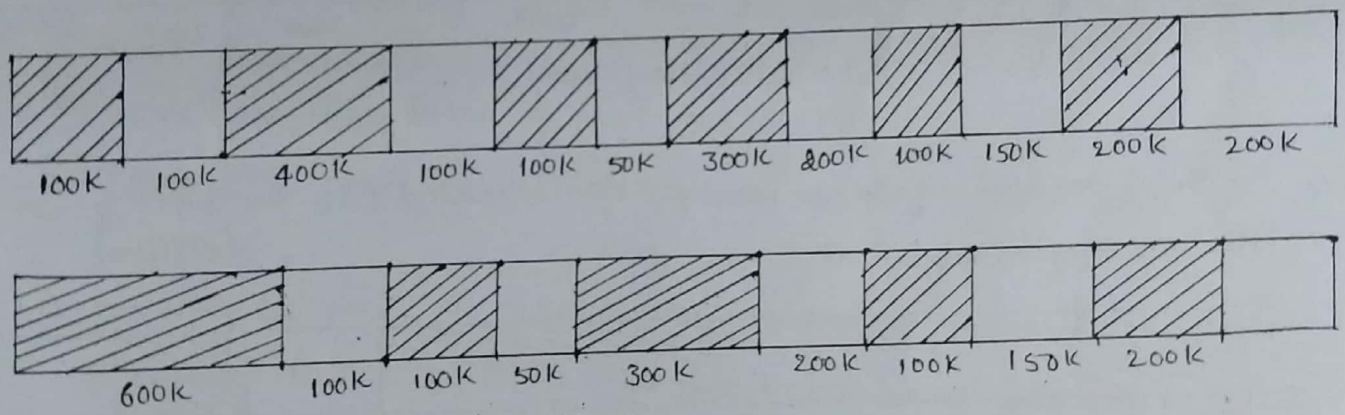


Sol: 1) first fit:



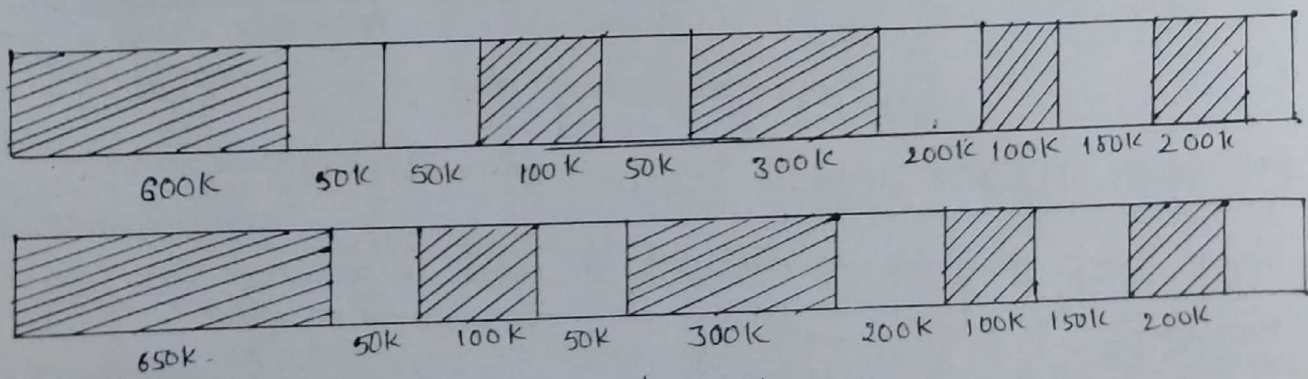
After allocating 200K

→ for second request of 100K it allocates in the first hole of size 100K



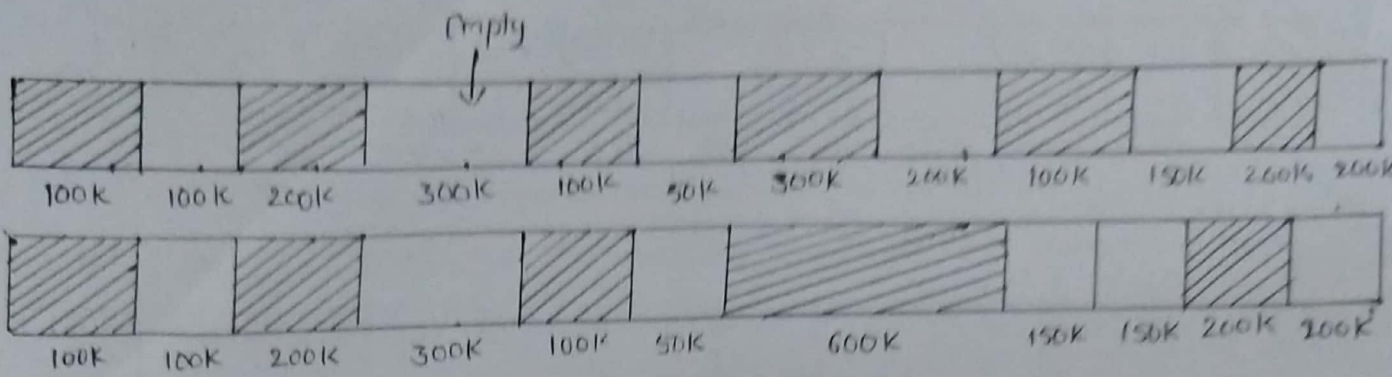
After allocating 100K.

→ for 3rd request 50K memory would now look as follows.



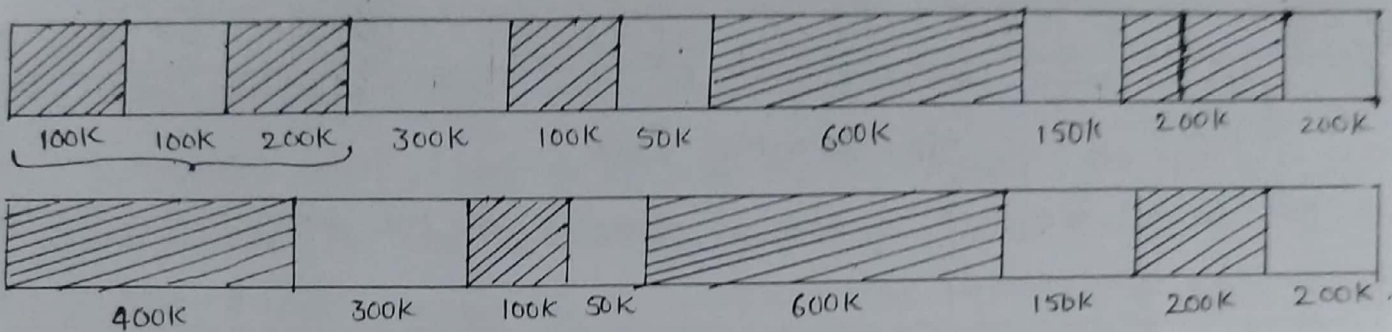
After allocating 50K.

ii) Best fit: It searches for the smallest hole larger than or equal to 200K. The fourth hole is exactly 200K, making it best fit.



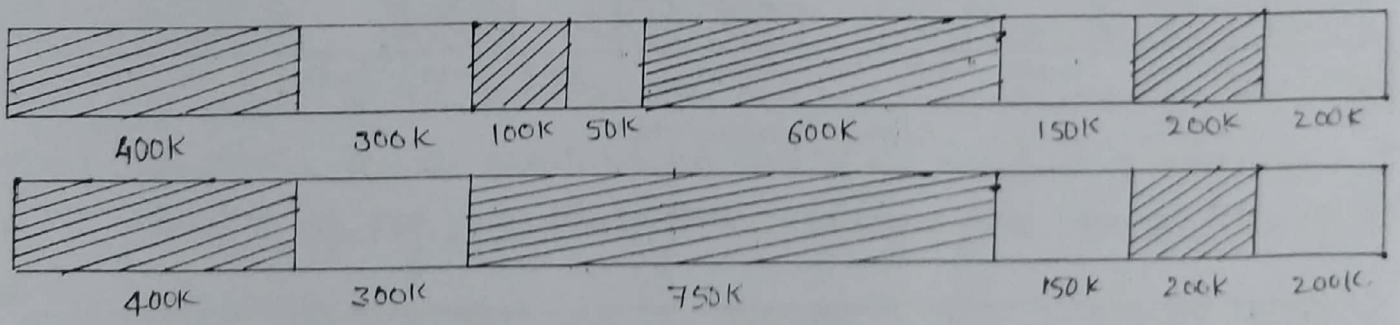
after allocating 200K.

→ Second request of 100K, first hole is best fit for it.



after allocating 100K.

→ The final request of 50K would be allocated to the hole of 50K
The memory looks like.



after allocating 150K.

MEMORY ALLOCATION:

→ Before any program is executed, it must be loaded into main memory.

→ Bringing of programs from secondary memory to main memory is known as loading.

→ The main task of memory management is to bring or load

programs into main memory for execution by the processor.
There are 2 kinds of loading. They are:

(1) Compile time loading: The process routines are loaded before execution. This is also known as static loading.

(2) Run time loading: The process routines are loaded during execution itself. This is also known as dynamic loading.

Memory must be ~~partioned~~ partitioned to facilitate loading
Partitioning is of 2 types.

(1) fixed partitioning (2) Dynamic partitioning.

(1) fixed partitioning: We must remember the the operating system occupies some fixed portion of main program & the rest of it is available for user programs.

→ The simplest scheme is to partition the memory into regions with fixed boundaries.

→ There are two forms of fixed partitioning.

(a) Equal size partitions: This contains memory with equal size partitions.

→ Loading is done in such a way that any process less than or equal to the partition size can be loaded into any available partition, if all partitions are full & no resident process is in the ready/running state, then the OS can swap a process out & load in another process.

Advantages:

→ supports multiprogramming
→ Requires no special H/w
→ Simple & easy to implement.

Disadvantage:

→ If a program is too large to set into that partition i.e., if program's size is larger than partition size then we cannot

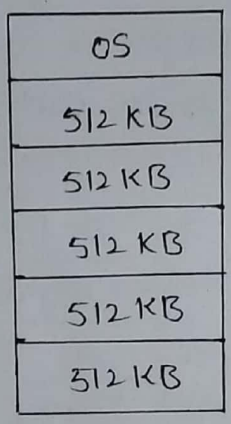
that programs into memory.

for example: Let us assume a program of back B, partition size is 512KB due to which we cannot load a 60KB program

(2) Any program through too small occupies an entire partition. Thus resulting in wastage of memory & loading to inefficient usage of memory.

for example: Let us take a program of size 112Kb if we load this program into a partition of size 512KB in the following fig results is wastage of 400KB memory which is undesirable.

→ In this case there is a wasted space internal to a partition is referred to as internal fragmentation.



Equal size partitions.

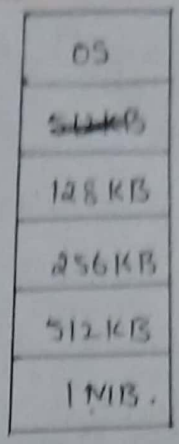
(b) Unequal size partitions: The problem of equal size partition can be partially overcome using unequal size partitioning.

→ The main memory is divided into partitions of unequal sizes. Here a process is loaded into suitable partitions such that it minimizes wastage of memory.

for example: If we take a process of size 120KB, it can be loaded into 1MB or 512KB / 256KB.

→ Objective is to reduce memory wastage so we load that process into 128KB partition as it results in 8KB of memory wastage, which is more in other cases.

unequal size partition.

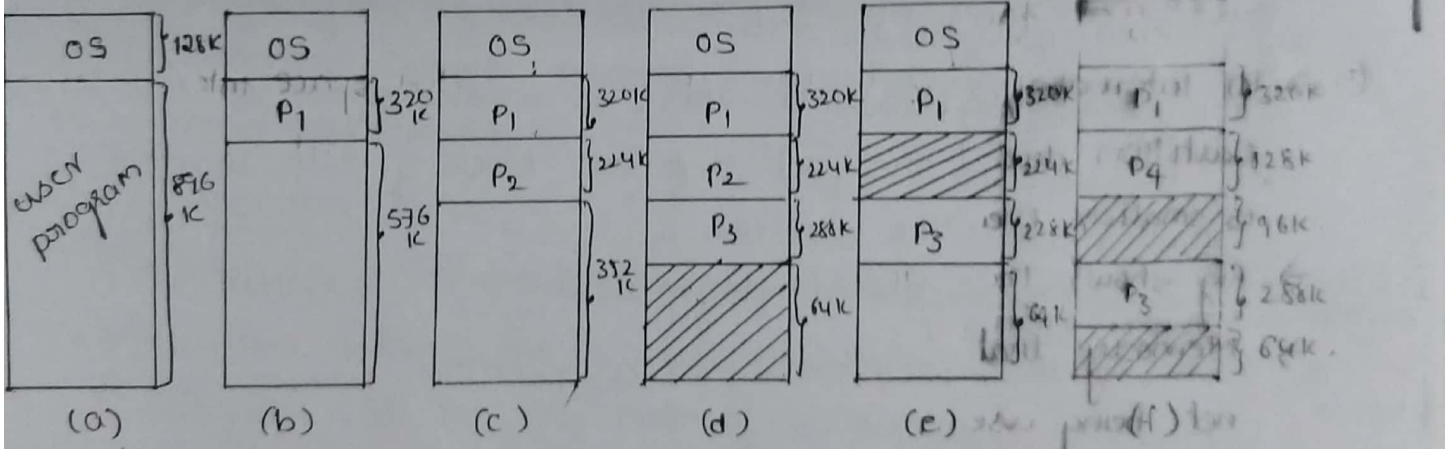


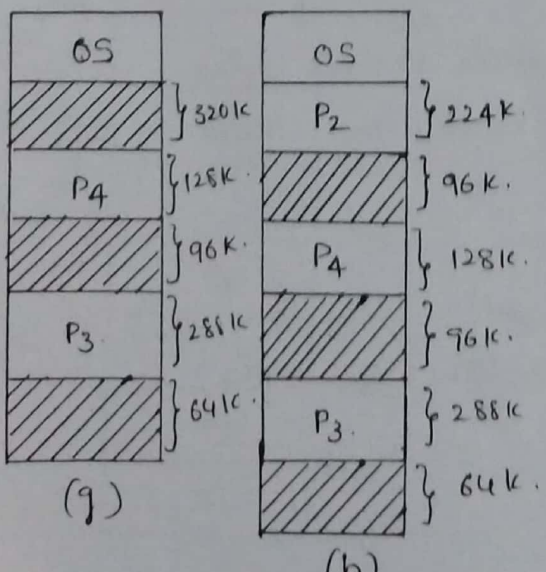
partition size are present at the time of system generation small jobs do not use partition space efficiently.

Dynamic Partitioning: It was introduced inorder to overcome the draw backs of fixed partitioning.

→ partitions are of variable length & number.

→ when a process is bought into main memory, it is allocated exactly as much memory as it requires & not more than that.





→ The process space is empty. Then a process 1 of size 320KBps is being loaded

→ After the process 1, P2, P3 of size 224KB, 288KB are loaded into the memory.

→ This leaves a hole at the end of memory.

→ This leaves at the end of memory

that is too small for the next process, i.e., P_4 of size 128 KB which is waiting to be executed.

→ As all the processes are met in the running state, P_3 of size 224 KB is swapped out & P_4 of size 128 KB is loaded.

→ By this 96 KB is left unused in (f).

→ Again P_2 is loaded P_1 will be swapped out & P_2 will be swapped in leaving again 96 KB.

→ As the time goes on & notice a lot of small holes in memory.

→ Thus memory becomes more & more fragmented & its usage declines.

→ This phenomenon is known as external fragmentation.

Fragmentation:

Memory fragmentation can be two types.

i) Internal fragmentation

nal fragmentation.

(1) → In internal fragmentation, there is wasted space internal to a partition due to the fact that the block of data loaded is smaller than the partition.

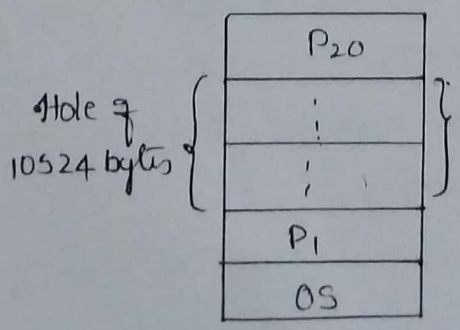
→ Fig shows the internal fragmental.

→ Memory that is internal to a partition scheme but is not being used.

→ For example in multiple partition scheme hole of 10,524 bytes

→ Suppose that the next process requests 10 520 bytes.

→ If we allocate exactly the request block, then there is left with a hole of 4 bytes.



→ The overhead to keep track of this hole will be substantially larger than the hole itself.

(2) External fragmentation:

→ External fragmentation exists when enough total memory space exists to satisfy a request, but it is not contiguous storage is fragmented into a large number of small holes.

→ There is a hole of 200k & 500k in multiple partition allocation scheme

→ Next process request for 700k of memory.

→ Actually 700k of memory is free which satisfy the request but hole is not contiguous.

→ So there is an external fragmentation of memory.

→ The selection of first fit versus best fit can affect the amount of fragmentation.

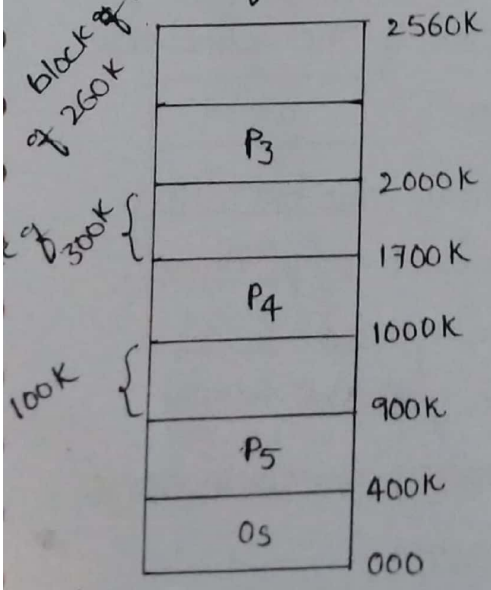
→ Depending on the total amount of memory storage, process size, external external fragmentation minor / major problem.

Compaction:

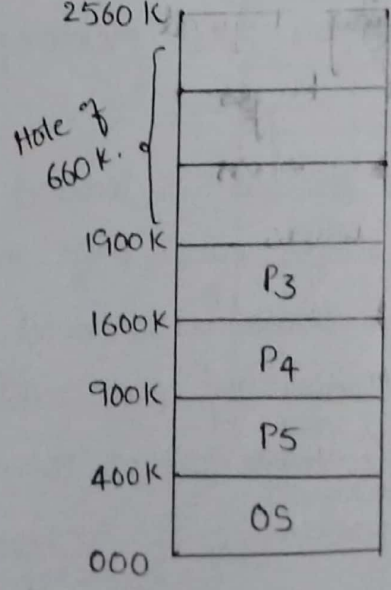
→ One technique for overcoming external fragmentation is compaction.

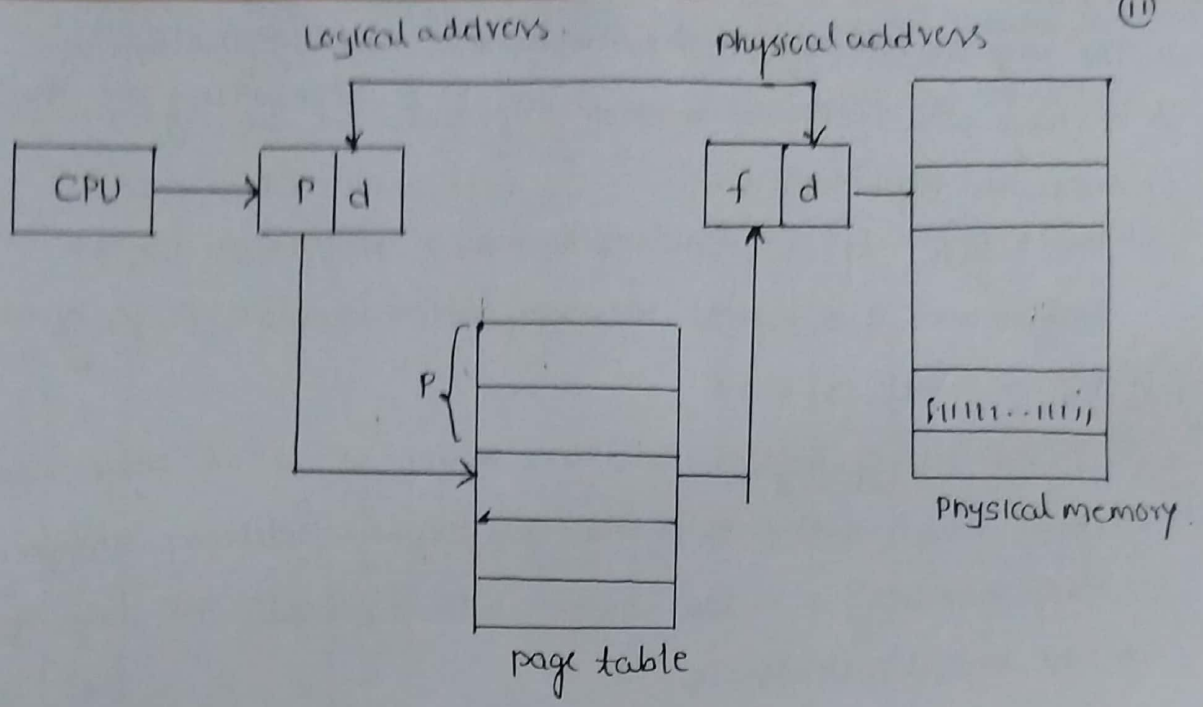
→ From time to time the OS shifts the processes so that they are contiguous & all the free memory is together in one block.

Before compaction.



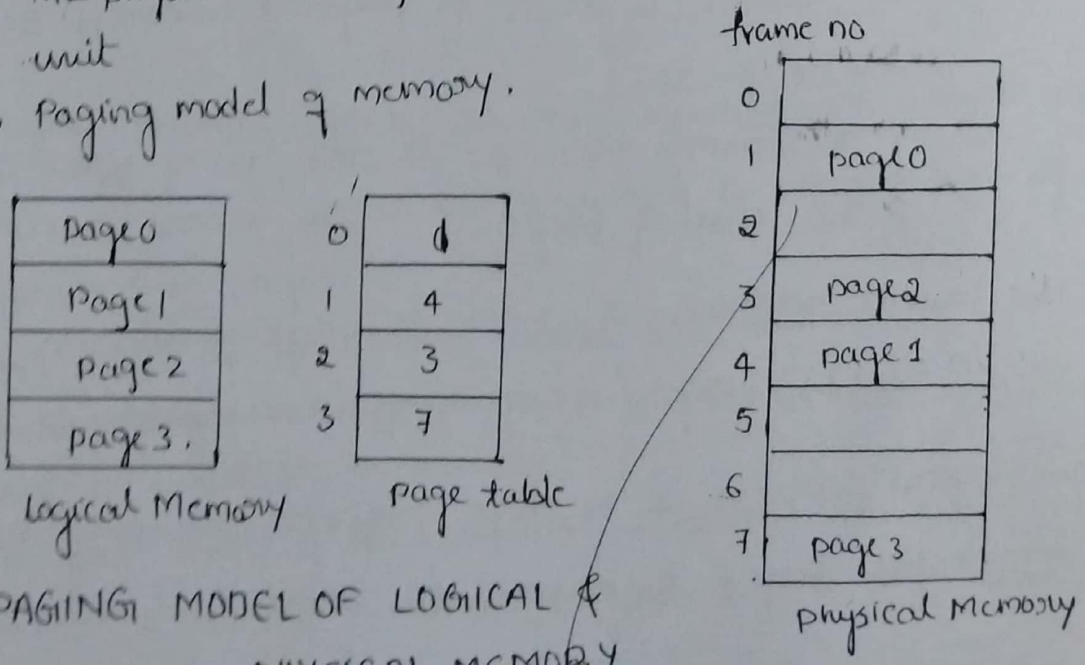
After





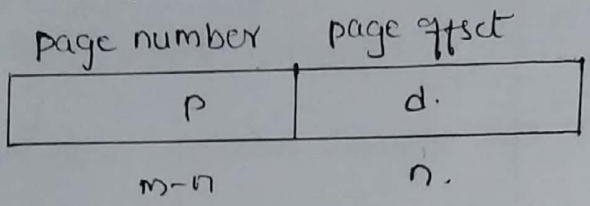
PAGING HARDWARE.

→ Cpu generates the logical address into two parts.
(1) page number (P) (2) Page offset (d).
→ Page table ~~stores~~ stores the number of the page frame allocated for each page.
→ 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
→ Paging model of memory.



PAGING MODEL OF LOGICAL & PHYSICAL MEMORY

→ The page size is defined by the H/W.
→ A page size is typically a power of 2 depending on the computer architecture.
→ The selection of a power of 2 as a page size makes the translation of a logical ~~Memory~~ address into a page number & page offset is easy.
→ If the size of logical address space is 2^m & page size is 2^n, then high order m-n bits of a logical address designate page number & n low-order bits designate the page offset.
→ The logical address is



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

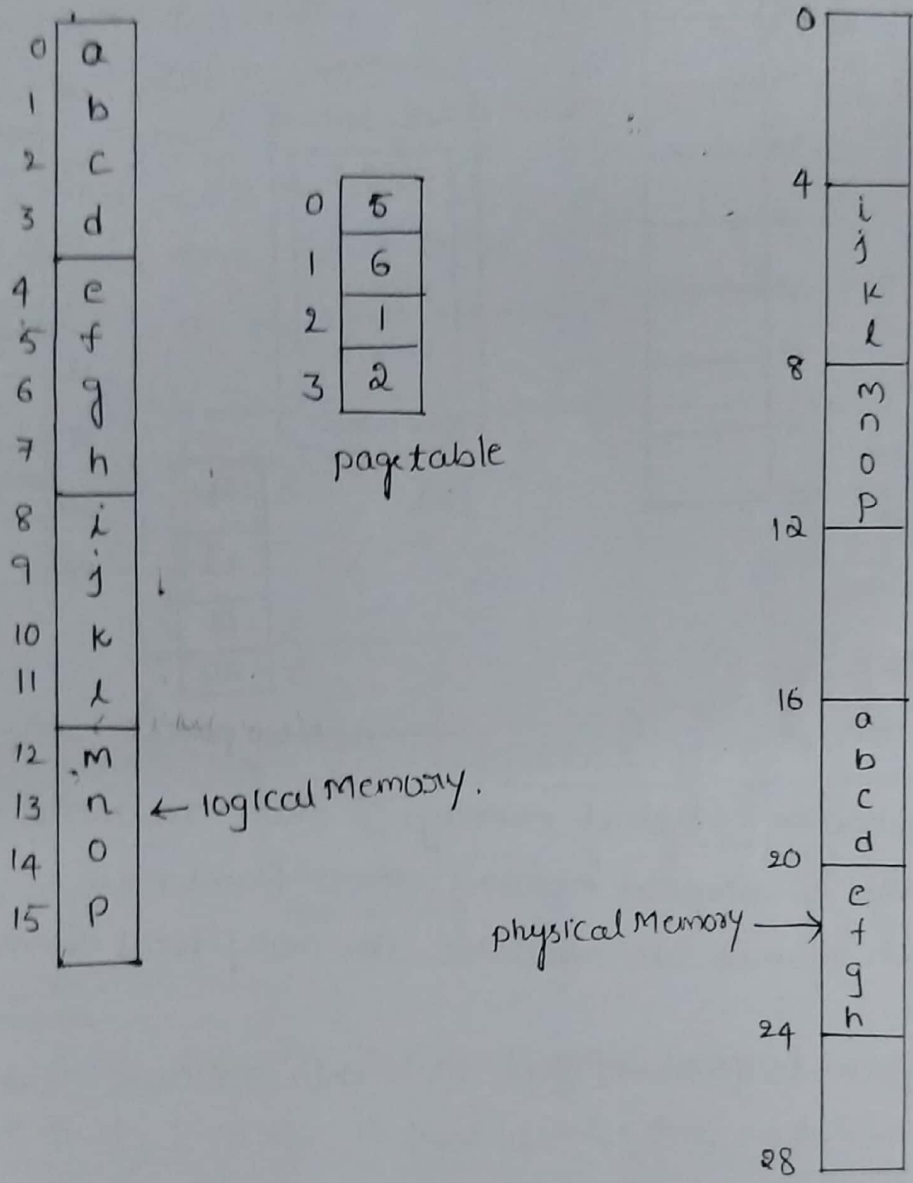
Ex: A page size of 4 bytes & a physical memory of 32 bytes (8 pages)

→ Logical address 0 is page 0. Indexing into the page table find that Page 0 is in frame 5.
→ Thus logical address 0 maps to physical address 20.
i.e., $((5 \times 4) + 0) = 20$.
5 is the frame no, 4 is the no of bytes in page
0 is the offset of page 0.
→ The logical address 3 maps to physical address as 23.
i.e., $((5 \times 4) + 3) = 23$. (page 0, offset 3, No of bytes 4)
→ The logical address 4 is page 1, offset 0, according to the page table Page 1 is mapped to frame 6.

→ Similarly logical address 13 to physical address 9. (12)

→ When we use a paging scheme we have no external fragmentation.

→ Any free frame can be allocated to a process that needs it



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

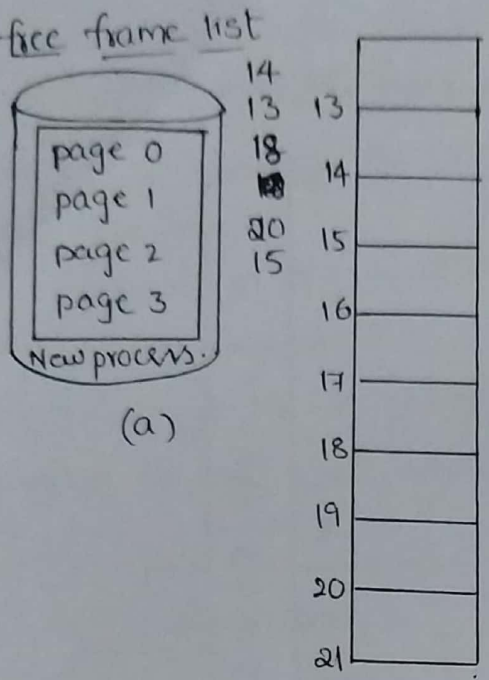
→ When a process arrives in the system to be executed. its size, expressed in pages

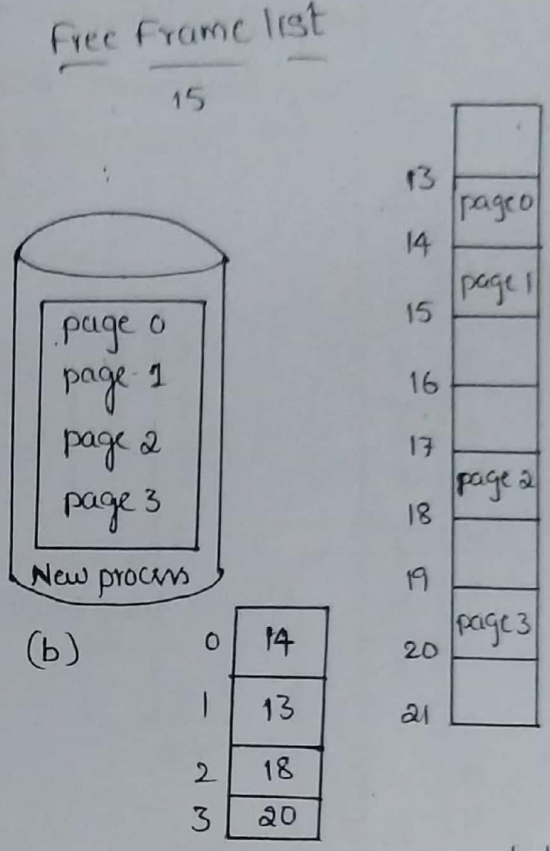
→ Each page of the process needs one frame.

→ If the ~~frame~~ process requires N pages, atleast N frames must be available in memory.

→ The first page of the process is loaded into one of the allocated frames & 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.





New process page table

Free frames
a) Before Allocation
b) After Allocation.

→ The OS is managing physical memory. It must be aware of the allocation details of physical memory which frames are allocated which frames are available. How many total frames there are....

→ This information is generally kept in a data structure called frame table.

→ The OS must be aware that user processes operate in user space. & all logical addresses must be mapped to procedure physical addresses.

→ Item is compared with all keys.

→ If the item is found the corresponding value field is returned.

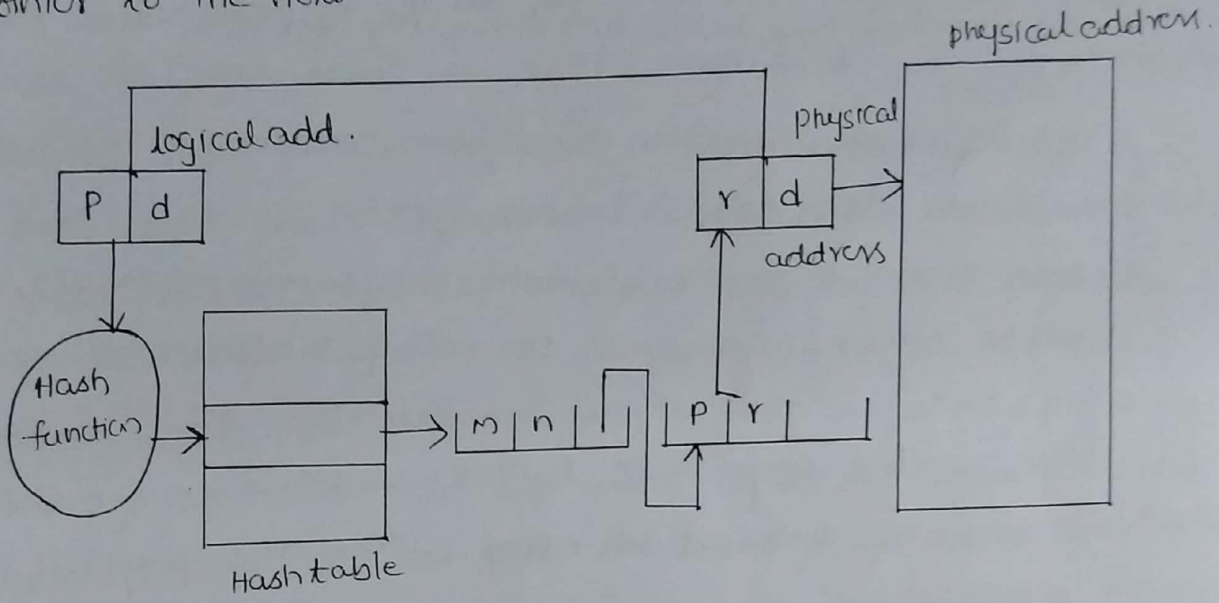
→ The search is fast, the Hardware is expensive.

→ The TLB contains only a few of the page table entries

→ when a logical address is generated by the CPU, its page

HASHED PAGE TABLE:

→ It handles be address space larger than 32 bits.
→ The virtual page no is used as based value
→ Linked list is used in the hash table
→ Each entry contains the linked list of element that hash to the same location.
→ Each element in the hash table contains.
1) virtual page number
2) Mapped page frame value
3) pointer to the next element in the linked list.



WORKING:

→ virtual page no is taken from virtual address.
→ virtual page no is hashed into the hash table
→ virtual page no is compared with the first element of linked list.
→ Both the value is matched, that value is used for calculating physical address
→ If the both value is not matched. that value is used for calculating physical address.
→ clustered page table is same as hashed page table but only difference is that each entry in the hash table refers to several pages.

INVERTED PAGE TABLE :

→ As address space has grown to 64 bits, the size of traditional page table becomes a problem

→ OS must then translate from logical to physical memory address

→ Table is sorted by virtual address. The OS is able to calculate where the table associated physical address entry is & to use that value directly.

→ One drawback of this method is that each page table may consist of million of entries.

→ To solve this problem we can use an inverted page table

→ An inverted page table has one entry for each real page of memory

→ 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-No> where process-id assumes the role of the address-space identifier.

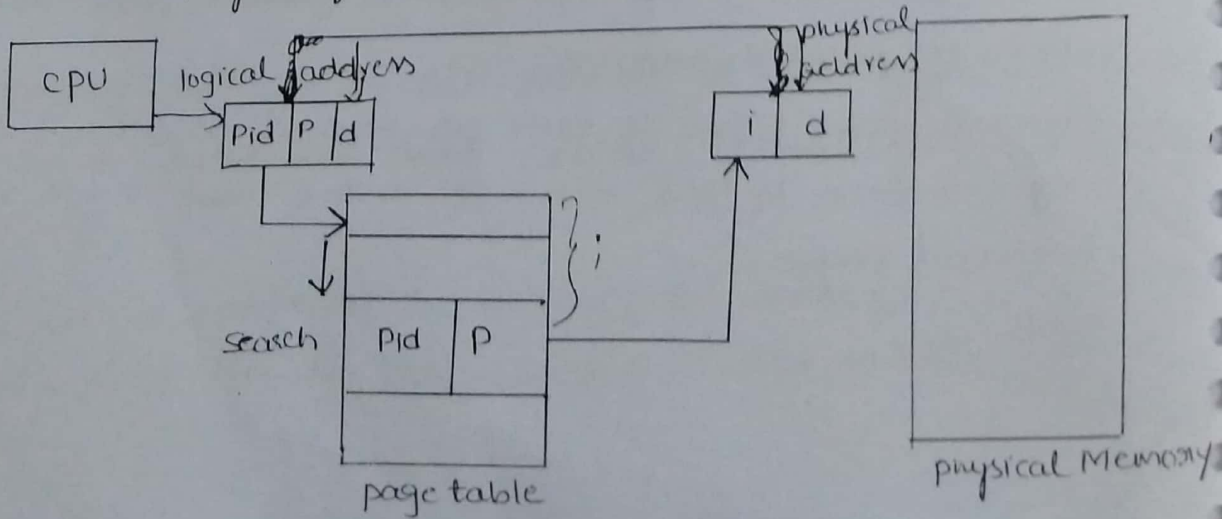
→ The inverted page table is then searched for a match.

→ If a match is found at entry i, then the physical address <i, offset>

→ If no match is found, then an illegal address access has been attempted.

→ Although this scheme decreases the amount of memory needed to store each page table.

→ It increases the amount of time needed to search the table when a page reference occurs.



Hardware support :-

Each OS has its own methods for storing page tables.
The hardware implementation of the page table can be done in several ways.
The simplest case the page table is implemented as a set of dedicated registers.
These registers should be built with very high-speed logic to make the paging-address translation efficient.
The drawback with the registers. The use of registers of the page table is satisfactory if the page table is reasonably small. when the page table is large registers are complement. To overcome that we place the page table in main memory, and a page-table base register (PTBR) points to the page table. Changing page-tables requires changing only this one register reducing context-switch time.
The problem with this approach is the time required to access a user memory allocation. If we want to access location i, we must first index into the page table, using the value of PTBR offset by the page number for i. This task requires a memory access.
It provides us the frame number, which is combined with the page offset to produce the actual address. we can access the desired place in memory.
The drawback is two memory accesses are needed to access a byte. Thus memory access is slowed by a factor of 2.
The standard soln to this problem is use a special, small, fast lookup hardware cache called Translation look-aside buffer (TLB).
⇒ The TLB is associative, high-speed memory.
⇒ Each entry in the TLB consists of two parts key & value.

→ When the associative memory is presented with an item the item is compared with all keys. If the item is found the corresponding value field is returned.

→ The search is fast, the hardware is expensive.

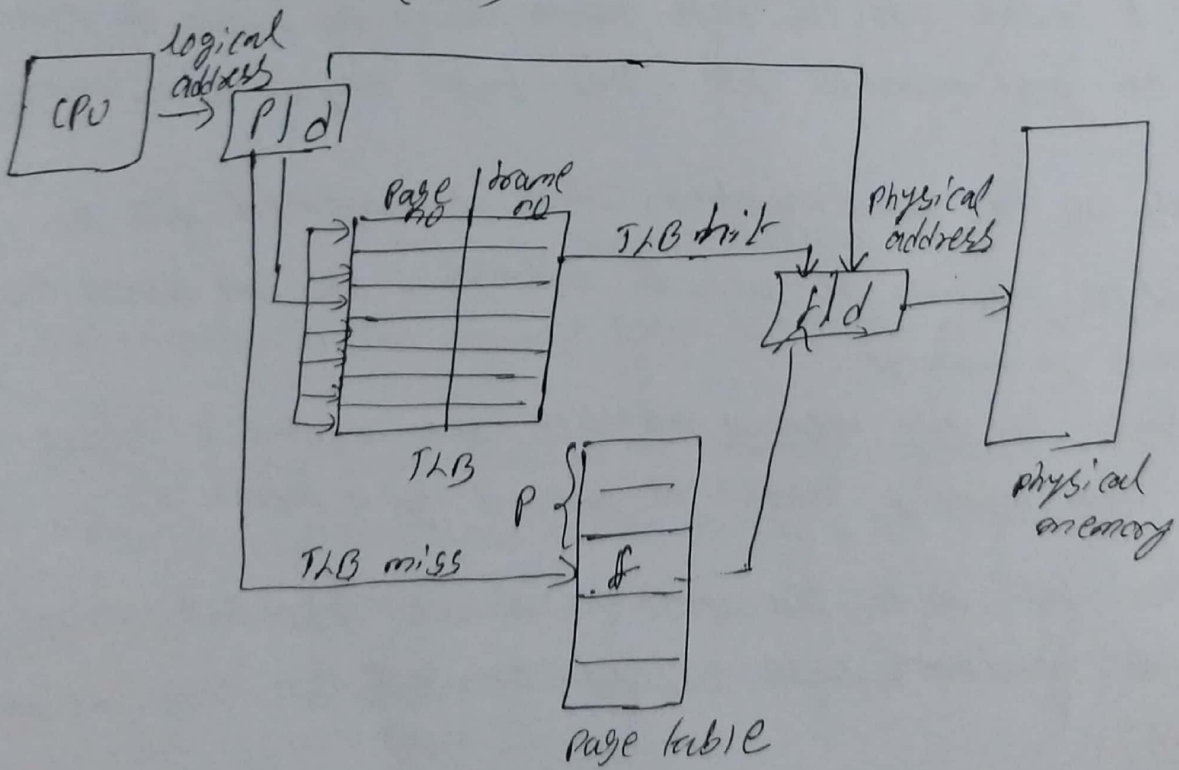
→ The TLB is used with page tables in the following way. The TLB contains only a few of the page-table entries.

→ When a logical address is generated by the CPU its page number is presented to the TLB.

→ If the page number is found, its frame number is immediately available and is used to access memory.

→ If the page number is not in the TLB (known as TLB miss) a memory reference to the page table must be made. 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 full of entries the OS must select one for replacement "police (LRU).



Paging hardware with TLB

SEGMENTATION:

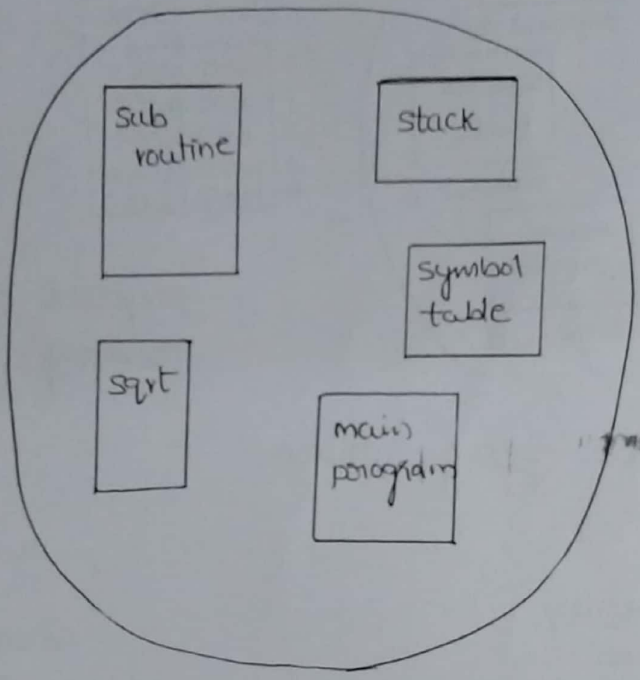
→ It is a memory-management scheme that supports this user view of memory.

→ A logical address space is a collection of segments. Each segment has a name and length.

→ The addresses specify both the segment name and the offset within the segment.

→ The user therefor specifies each address by two quantities. a segment name and an offset.

→ A logical address consists of 2 parts. < segment-number, offset >.



Logical Address.

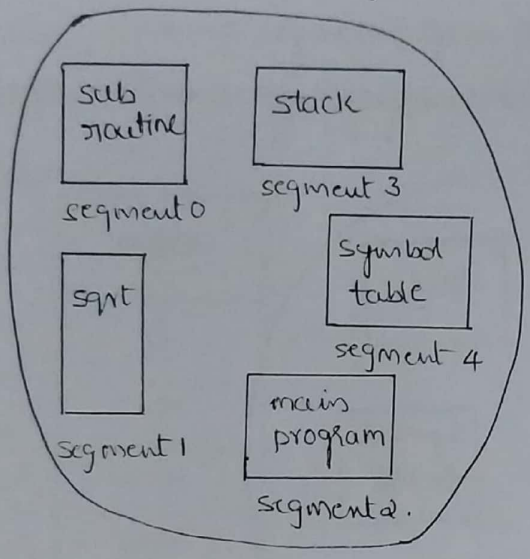
→ Each entry in the segment table has a segment base & a segment limit.

→ The segment base contains the starting physical address and the segment limit specifies the length of the segment.

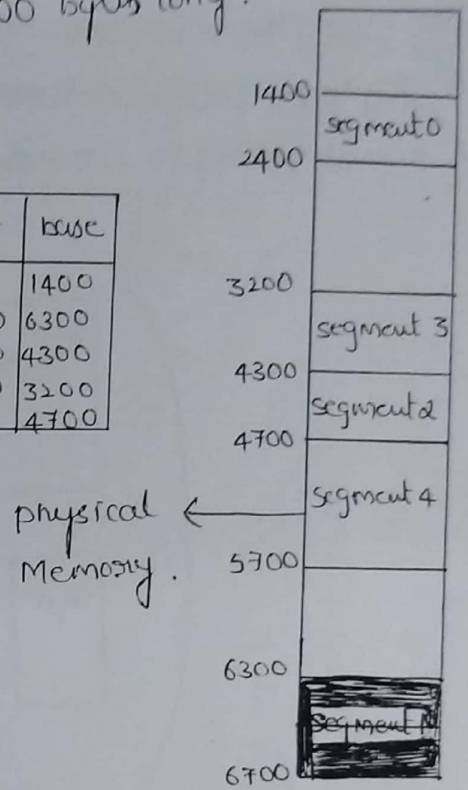
→ A logical address consists of two parts, a segment number(s) and an offset into that segment d. The segment is used as an index to the segment table.

→ we have five segments numbered from 0 through 4. The segments are stored in physical memory. The segment table has a

separate entry for each segment giving the beginning address of the segment in physical memory. and the length of the segment. for example, segment 2 is 400 bytes long and begins at location 4300+53 = 4353. A reference to segment 3, byte 852, is mapped to 3200 (the base of segment) + 852 = 4052. A reference to byte 1222 of segment 0 would result in a trap to the O.S, as this segment is only 1000 bytes long.



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



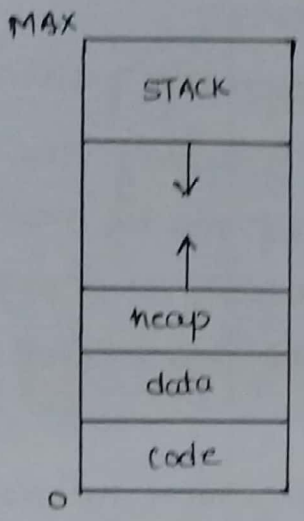
Example of segmentation.

Virtual MEMORY

Virtual memory involves the seperation of logical memory as perceived by users from physical Memory. This seperation allows an extremely large virtual Memory to be provided for programmers when only a smaller physical memory is available. Virtual memory makes the task of programming much easier, because the programmer no longer needs to worry about the amount of physical memory available. she can concentrate instead on the problem to be programmed.

The virtual address space of a process refers to the logical view of how a process is stored in memory. Typically, this view is that a process begins at a certain logical address - say, address 0 - and exists in contiguous memory.

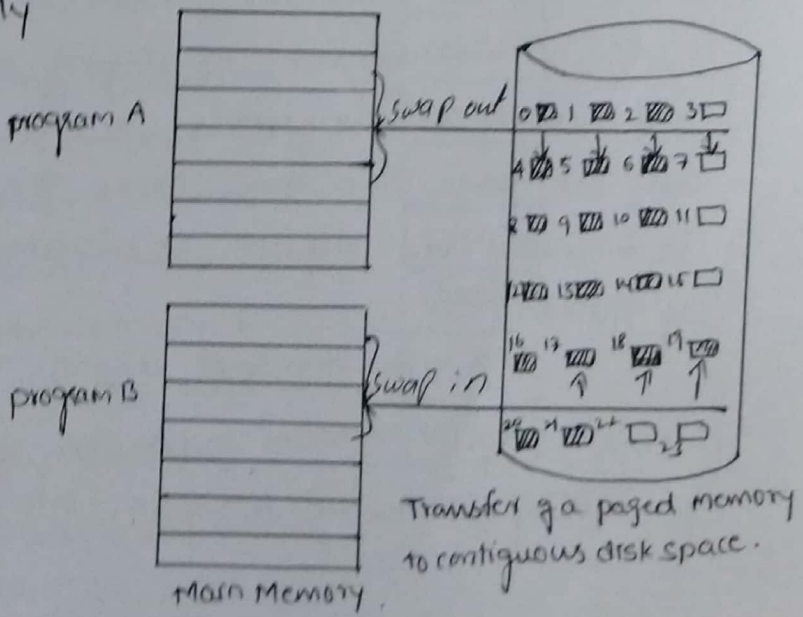
we allow for the heap to grow upward in memory as it is used for dynamic memory allocation. Similarly, we allow for the stack to grow downward in memory through successive function calls. The large blank space b/w the heap and the stack is part of the virtual address space but will require actual physical pages only if the heap or stack grows. virtual address spaces that include holes are known as sparse address spaces. Using a sparse address space is beneficial because the holes can be filled as the stack or heap segments grow or if we wish to dynamically link libraries during program execution.



Virtual Address Space.

DEMAND PAGING:

Consider executable program might be loaded from disk into memory. at program execution time. we may not initially need the entire program in memory initially load pages only as they are needed. This technique is known as "Demand Paging" and is commonly used in virtual memory system. With demand-paged virtual memory, pages are only loaded



Transfer of a paged memory to contiguous disk space.

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 in secondary memory. Swapping the entire process into memory, however, we use a lazy swapper. A lazy swapper never swaps a page into memory unless that page will be needed. Since we are now viewing a process as a sequence of pages, rather than as one large contiguous address 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.

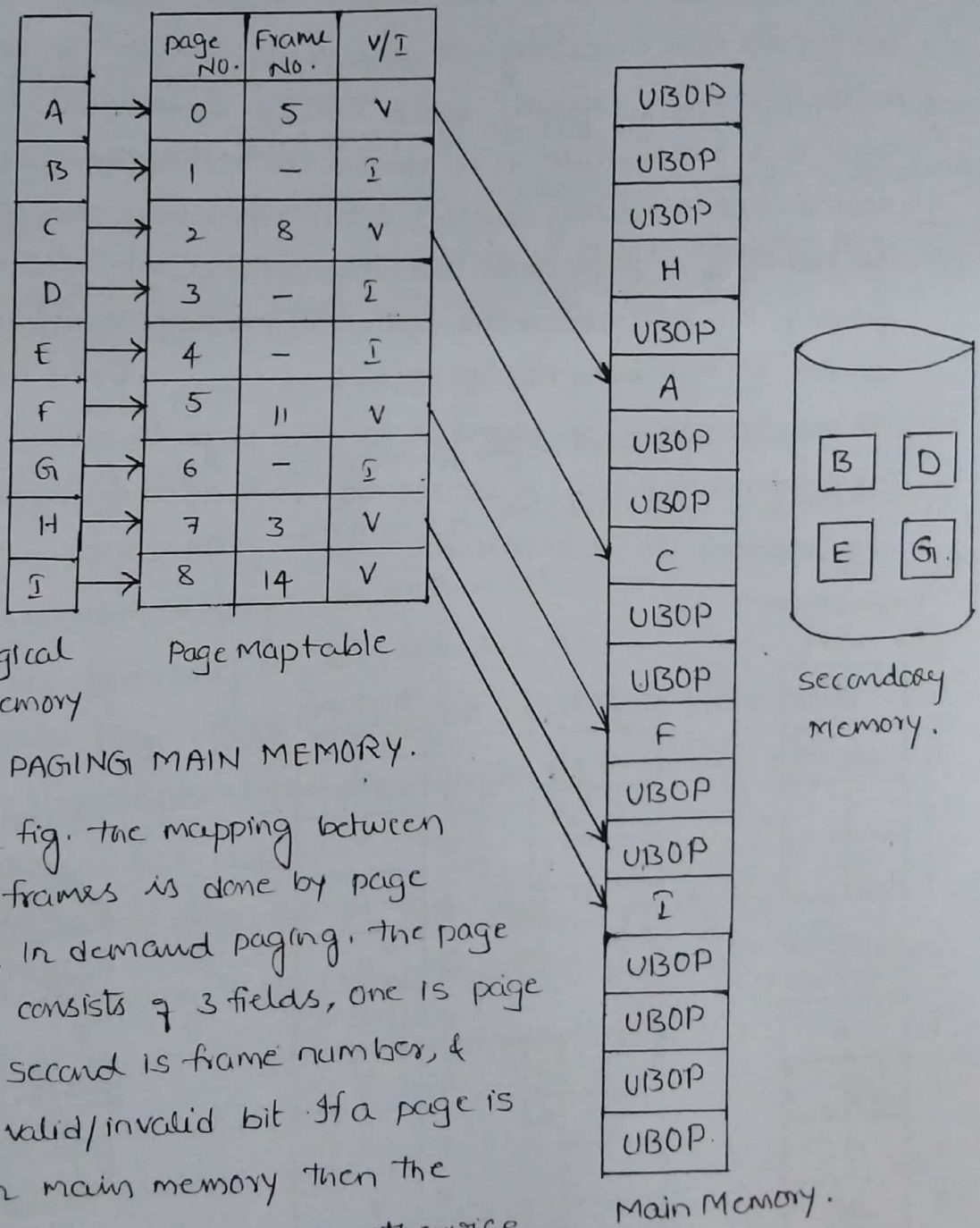
Demand paging, it is the application of virtual memory. It is the combination of paging and swapping. The criteria of this scheme is "a page is not loaded into the main memory from secondary memory, until it is needed". So a page is loaded into the main memory by demand, so this scheme is said to be "Demand Paging".

Demand paging is a concept in which the pages are loaded into memory only on demand i.e; whichever page needs to be executed, that page will be requested by the CPU. The operating system will bring that page from disk and transfers it into memory frames. Thus pages are loaded into memory only when they are required, the remaining pages are in disk.

However, in paging all the pages are loaded into memory at one stretch. But we know that CPU can execute only one page at a time, hence the remaining pages will sit in memory but will not be executed. This will cause wastage of frames.

In demand paging, the program is divided into several pages. Memory is divided into several frames. Instead of loading the entire program into memory, only essential

pages will be transferred to main memory. Pages are transferred to memory only when a request to that page is generated. Assume that the logical address space is 72KB. the page and frame size is 8KB, so the logical address space is divided into 9 pages, numbered from 0 to 8. The available main memory is 40KB. i.e., 5 frames are available, the remaining 4 pages are loaded in the secondary storage device, whenever those pages are required, the operating system swap-in those pages into main memory.



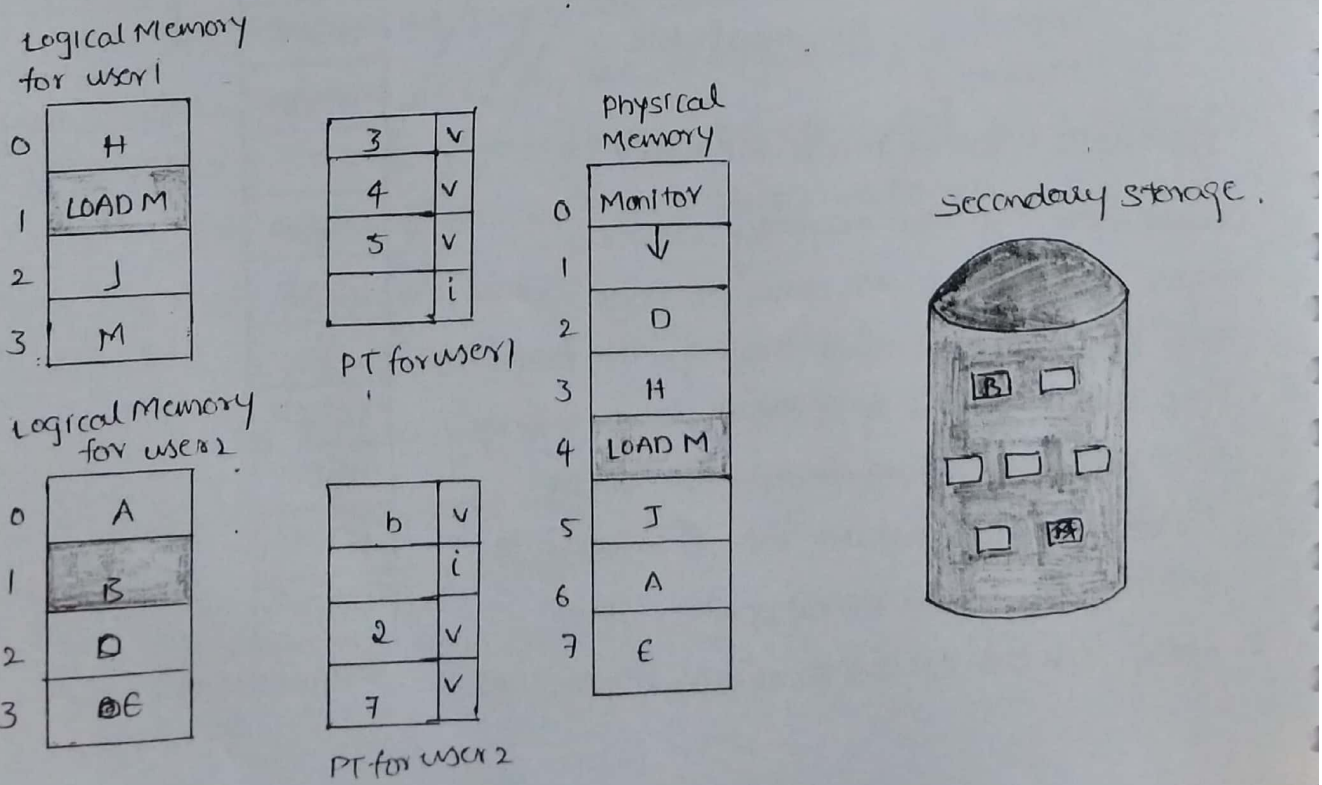
DEMAND PAGING MAIN MEMORY.

Consider the fig. the mapping between pages and frames is done by page map table. In demand paging, the page map table consists of 3 fields, one is page number, second is frame number, & third is valid/invalid bit. If a page is residing in main memory then the valid/invalid bit set to valid, otherwise

if page resides in secondary storage, then the bit sets to "invalid". In the fig, the page numbers 1, 3, 4, 6, are loaded in secondary memory. so those bits are set to invalid, remaining all pages are residing in main memory, so those bits are set to valid. The available free ~~memory~~ frames in main memory is 5, so 5 pages are loaded. remaining frames are used by other proces

PAGE REPLACEMENT:

Page replacement policy deals with the selection of a page in memory to be replaced when a new page must be brought in. while a user process is executing, a page fault occurs. The hardware traps to the operating system, which checks its internal tables to see that this page fault is a genuine one rather than an illegal memory access. The operating system determines where the desired page is residing on the disk, but then finds that there are no free frames on the free frame list. All memory is in use. This is shown in fig. When all the frames in main memory are occupied and it is necessary to bring in a new page to satisfy a page fault, replacement policy is concerned with selecting a page currently in memory to be replaced.



WORKING OF PAGE REPLACEMENT ALGORITHM:

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

MEMORY REFERENCE STRING:

The string of memory references is called a Reference string. A succession of memory references made by a program executing on a computer with 1MB of memory is given below in hex

Notation.

14489, 1448B, 14494, 14496, A1F8, 14497, 14499, 2638E, 1449A, ..

When analyzing page replacement algorithms, we are interested only in the pages being referenced. Assuming a 256 byte page size the referenced pages are obtained

.... 144, 144, 144, 144, A1, 144, 144, 263, 144, ...

The pattern of page reference can be compared into a reference string for page replacement ~~for~~ analysis as follows:

... 144, A1, 144, 263, 144, ...

A reference string obtained in this way is used to illustrate most of the following replacement algorithms.

Replacement algorithms are of following types:

1. First in First Out (FIFO)
2. Least Recently used (LRU)
3. optimal.

(1) FIFO page replacement:

- FIFO is one of the simplest method. FIFO page replacement algorithm selects the page that has been in memory the longest. When a page must be replaced, the oldest page is chosen. To implement FIFO page replacement algorithm, the memo manager must keep track of the relative order of the loading of pages into the memory. One way to accomplish this is to maintain a FIFO queue of pages. When a page is bought into memory, it is inserted at the tail of the queue.
- FIFO page replacement algorithm is easy to understand & program. Its performance is not always good. FIFO is ~~not~~ the first choice of the operating system designers for page replacement algorithm.

Let us consider the reference string.

0, 1, 2, 3, 0, 1, 2, 3, 0, 1, 2, 3, 4, 5, 6, 7.

- page frame is 3. Initially all the 3 frames are empty. The first three references (0,1,2) cause page faults and are bought into these empty frames. The next reference 3 is replace page 0, because page 0 was bought in first.

This process is as shown below.

Reference String:

0 1 2 3 0 1 2 3 0 1 2 3 4 5 6 7.

NAME																
0	0	0	0	3	3	3	2	2	2	1	1	1	4	4	4	7
1		1	1	1	0	0	0	3	3	3	2	2	2	5	5	5
2			2	2	2	1	1	1	0	0	0	3	3	3	6	6
ge ault	*	*	*	*	*	*	*	*	*	*	*	*	*	*	*	*

* ⇒ page fault.

This example incurs 16 page faults in FIFO algorithm.

(2) LRU page Replacement:

Least recently used policy replaces the page in memory that has not been referenced for the longest time. The LRU algorithm performs better than FIFO. The LRU algorithm belongs to a larger class of stack replacement algorithms. When more real memory is made available to the executing program, stack algorithms therefore do not suffer from Belady's anomaly. Let us apply LRU algorithm to the reference string with frame is 3.

0 1 2 3 0 1 2 3 0 1 2 3 4 5 6 7.

Reference string.

Frame	0	1	2	3	0	1	2	3	0	1	2	3	4	5	6	7
0	0	0	0	3	3	3	2	2	2	1	1	1	4	4	4	7
1		1	1	1	0	0	0	3	3	3	2	2	2	5	5	5
2			2	2	2	1	1	1	0	0	0	3	3	3	6	6
page-fault	*	*	*	*	*	*	*	*	*	*	*	*	*	*	*	*

Number of page-faults: 16.

Consider the same reference string with 4 page-faults.

Frame	0	1	2	3	0	1	2	3	0	1	2	3	4	5	6	7
0	0	0	0	0	0	0	0	0	0	0	0	0	4	4	4	4
1		1	1	1	1	1	1	1	1	1	1	1	1	5	5	5
2			2	2	2	2	2	2	2	2	2	2	2	2	6	6
3				3	3	3	3	3	3	3	3	3	3	3	3	7
Page fault	*	*	*	*									*	*	*	*

Number of page-fault: 8.

- Implementation of the LRU algorithm imposes too much overhead to be handled by software alone. Thus the implementation of a pure LRU replacement algorithm requires extensive and dedicated hardware support for the desired stack operation. Stack is one of the solution for implementing LRU algorithm. Whenever a page is referenced, it

removed from the stack and put on the top. In this way, the top of the stack is always the most recently used page and the bottom is the LRU page. It is best implemented by a doubly linked list, with head and tail pointer.

- Second method used for this is by using counter. Counter is incremented for every memory reference.

(3) OPTIMAL PAGE REPLACEMENT:

The optimal policy selects the replacement that page for which the time to the next reference is the longest. An optimal page replacement algorithm has the lowest page fault rate of all algorithms and will never suffer from Belady's anomaly. This algorithm is impossible to implement because it would require the operating system to have perfect knowledge of future events.

Let us consider the reference string with frame equal to 3.

0 1 2 3 0 1 2 3 0 1 2 3 4 5 6 7

Reference String.

Frame	0	1	2	3	0	1	2	3	0	1	2	3	4	5	6	7
0	0	0	0	0	0	0	0	0	0	1	1	1	4	4	4	7
1		1	1	1	1	1	2	2	2	2	2	2	2	5	5	5
2			2	3	3	3	3	3	3	3	3	3	3	3	6	6
page fault	*	*	*	*			*			*			*	*	*	*

Number of page fault = 10.

ALLOCATION OF FRAMES:

How do we allocate the fixed amount of free memory among the various processes? If we have 93 free frames and two processes, how many frames does each process get?

Here Allocation Algorithms:

Here allocation algorithms can be done in 2 ways,

i) equal allocation algorithms ii) Proportional allocation algorithms.

The easiest way to split m frames among n processes is to give everyone an equal share, m/n frames. For instance, if there are 93 frames and five processes, each will get 18 frames. This scheme is called "equal allocation". Consider a system with a 1-KB frame size. If a student process of 10 KB and an interactive database of 127 KB are the only two processes running in a system with 62 free frames. It doesnot make much sense to give to each process 31 frames. The student process does not need more than 10 frames, so the other 21 are, strictly speaking, wasted.

To solve this problem, we can use "proportional allocation". Let the size of the virtual memory for process P_i be s_i, and define,

$$S = \Sigma s_i.$$

Then if the total number of available frames is m, we allocate a_i frames to process P_i, where a_i is approximately,

$$a_i = s_i / S \times m.$$

we must adjust each a_i to be an integer that is greater than the minimum number of frames required by the instruction set, with a sum not exceeding m. For proportional allocation, we would split 62 frames b/w two processes, one of 10 pages and one of 127 pages, by allocating 4 frames and 57 frames respectively, since

$$\left(\frac{10}{10+127}\right) \times 62 = \frac{10}{137} \times 62 \approx 4 \text{ and}$$

$$\left(\frac{127}{10+127}\right) \times 62 = \frac{127}{137} \times 62 \approx 57.$$

Both processes share the available frames according to their "needs", rather than equally.

With multiple processes competing for frames, we can classify page replacement algorithms into two broad categories: Global Replacement and Local Replacement.

Global Replacement allows a process to select a replacement frame from the set of all frames, even if that frame is currently allocated to some other process. Local Replacement requires that each process select from only its own set of allocated frames.

With a local replacement strategy, the number of frames

4. FILE SYSTE INTERFACE.

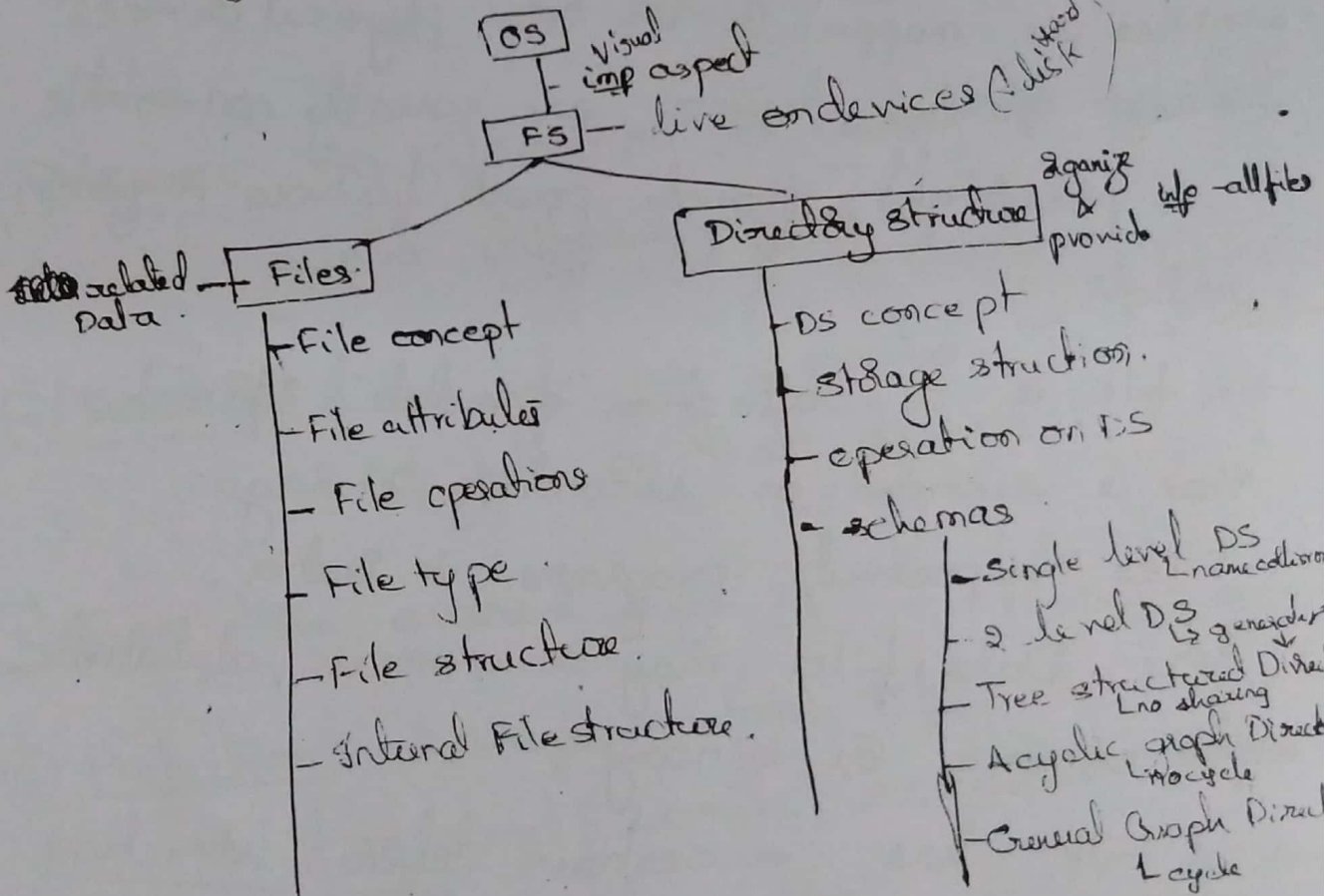
→ File system is the most visible aspect of OS

→ File system consist of 2 parts

→ collection of files.
which contains related data.

→ Directory structure
which organize & provides information about all the files in the system.

→ File system live on devices.



File concept:

→ Computers can store info on various storage media such as magnetic disks, magnetic taps & optical disks

→ To make computer system convenient to use, the OS provides a uniform logical view of information storage

→ Logical storage unit is called File

→ Files are mapped by OS onto physical devices.

→ these storage devices are usually non-volatile & persistent though power failures & system reboots.

→ A file is a collection of related information that is recorded on secondary storage.

→ Files represent programs & Data.

Data files may be numeric, alphabetic, alpha numeric & binary.

→ A File has a certain defined structure which depends on its type

→ Text file – Sequence of characters

→ source file – sequence of functions & subroutines

→ object file – sequence of bytes

→ executable file – series of code sections

File Attributes:

→ Attributes are properties & characteristics that describe a file.

→ file attributes vary from one os to another.

→ File attributes consist of

(1) Name: human readable form.

(2) Identifier: Unique tag mostly number, identifies the file within the FS.

(3) Type: This information is needed for sys that supp different types of files.

(4) Location: This info is a pointer. To the location of file on that device.

(5) Size: the current size of the file.

(6) Protection: Access control info ie who can read, write, execute & so on.

(7) Time, Date, User identification:

Information contains creation, last modification & last use. This is useful for protection, security & usage monitoring.

3) File operations:

→ A file is an abstract data type.

→ OS provide system calls to create, write, read, reposition, delete & truncate files.

→ These are six basic file operations.

① Creating a file:

→ Find space in FS for the file.

→ Add entry in Directory for the new file.

② Writing a file:

→ Specify file name & info to be written to the file.

→ sys. call searches for file's location in dir

→ sys keeps write pointer to the location of file where the next write is to take place.

→ write pointer is updated.

③ Reading a file:

→ specify filename

→ sys searches for file's location in dir

→ sys keeps read pointer

→ read pointer is updated.

④ Repositioning within a file:

→ current-file-position pointer is repositioned to a given value.

→ this file operation is known as File seek.

(5) Deleting a file :

→ searches for file's location in directory to delete.

→ Having found, we release all file space so that it can be reused by other files & erase the directory entry.

(6) Truncating a file :

→ Erase the contents of a file but keep its attributes.

→ Rather than forcing a user to delete the file & recreate it.

Other common operations

(1) Appending :- New info is added to the existing file. ~~to renaming an existing file.~~

(2) Renaming :- Renaming an existing file.

File Operations involve.

a) open() :- open() sys call be made before a file is used actively.

b) close() :- close() sys call is used after using a file

→ some os provide facilities for locking an open file.

→ File locks allow one process to lock a file & prevent other processes from gaining access to it.

→ File locks are useful for files that are shared by several processes

Ex:- Sys log file — used & accessed by no of process in the sys.

→ There are 2 types of locks.

→ shared lock (or) reades lock.

→ Exclusive lock (or) writes lock.

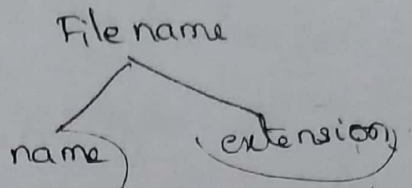
→ shared lock: several processes can acquire the lock concurrently

→ Exclusive lock: - Only one process can acquire the lock at a time.

(1) File Type:

→ A common technique for implementing file types is to include the type as part of the filename.

→ File name is split into 2 parts



→ Usually separated by a period character.

→ In this way, the user & OS can tell what type of file it is.

→ Example are resume.doc ✓
xyz.c ✓
server.java.

File Types	Usual extension	Function
1) Text file	.doc, .txt	textual data & documentation.
2) Source file	.c, .java, .pas	programs written in different programming languages
3) Object file	.obj, .o	machine language not linked.
4) Executable file	.exe, .com, .bin	machine language ready to run
5) Multimedia file	.mpeg, .mp3, .jpg	Binary file containing audio & video file.
6) Print & view file	.pdf, .jpg	Binary file in a format for printing & viewing
7) Archive	.zip, .arc	Related files grouped into one file
8) Batch	.bat	commands to the interpreter.
9) word process	.doc, .wp	various word process format
10) Library	.lib, .a,	libraries of routines for programmer

3) File structure:

→ File type is used to indicate the internal structure of the file

→ Certain files must be confirm to a required structure that is understood by OS

⇒ For ex:- Assume the system supports 2 files
Text file
Executable binary file.

→ if the user want to define an encrypted file to protect the contents from being read by unauthorized people

→ Users find neither file type to be appropriate

⇒ UNIX considers each file to be a sequence of 8-bit bytes.

⇒ All OS must support at least one structure that of an executable file - so that the sys is able to load & run programs

⇒ Macintosh OS support a min no. of file structures

→ It expects files to contain 2 parts

File

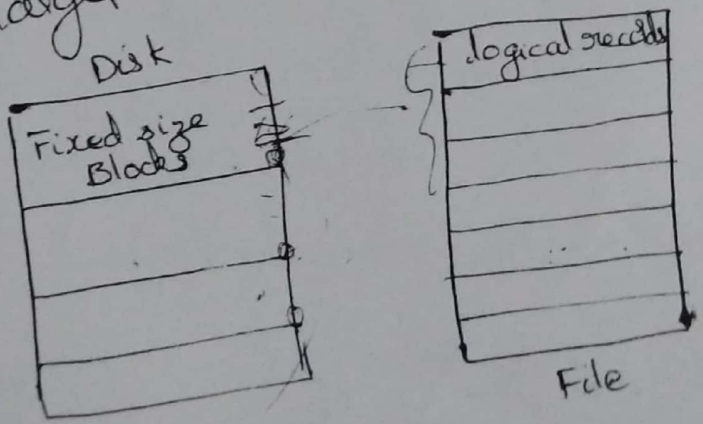
resource fork Data fork.

Resource fork: contains information of interest to the user.

Data fork: contains program code & data.

(6) Internal File Structure:

→ Disk have block size (physical record)
→ All blocks are the same size.
→ It is unlikely that physical record size will exactly match the length of the logical record.
→ logical records may vary in length.
→ Packing a no of logical records into physical blocks is a common sol to the problem.
→ But some portion of the last block of each file is generally wasted.
→ This is called Internal fragmentation.
→ All FS suffer from "
→ larger block size - greater the internal fragmentation



Directory structure.

> File System is an imp aspect of operating system.

→ File system live on device & is divided into 2 parts
- File — which contain related data.
- Directory structure — Organize & provide information about all files.

Storage Structure:

→ The complete disk can be used to store FS.

→ (or) some partition of disk can be used to store FS

→ The portion of storage disk used to store the FS is termed as Volume

→ Volume contains FS & information about the files — name, type, location size — which is called as device directory (or) Volume table of content (or) simply a directory.