Unit-4

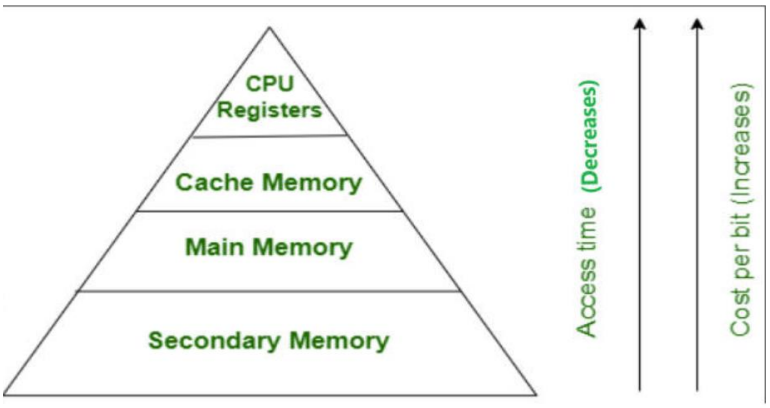
Memory Management and Virtual Memory

Topic 1

The term memory can be defined as a collection of data in a specific format. It is used to store instructions and process data. The memory comprises a large array or group of words or bytes, each with its own location. The primary purpose of a computer system is to execute programs. These programs, along with the information they access, should be in the main memory during execution. The CPU fetches instructions from memory according to the value of the program counter.

Basic Hardware

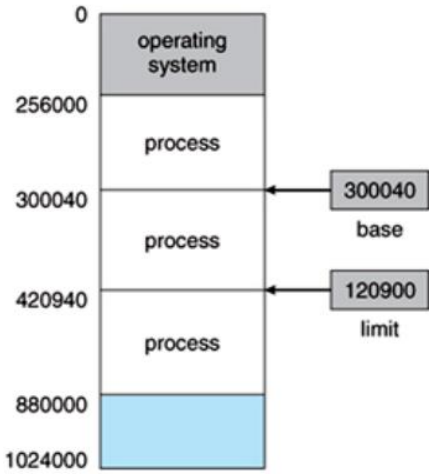
- The CPU can only access its registers and main memory. It cannot, for example, make direct access to the hard drive, so any data stored there must first be transferred into the main memory chips before the CPU can work with it.
- Memory accesses to registers are very fast, generally one clock tick, and a CPU may be able to execute more than one machine instruction per clock tick.
- Memory accesses to main memory are comparatively slow, and may take a number of clock ticks to complete. In such cases, the processor normally needs to stall, since it does not have the data required to complete the instruction that it is executing. This situation is intolerable because of the frequency of memory accesses. The remedy is to add fast memory between the CPU and main memory. A memory buffer used to accommodate a speed differential, called a cache memory.



Computer Memory

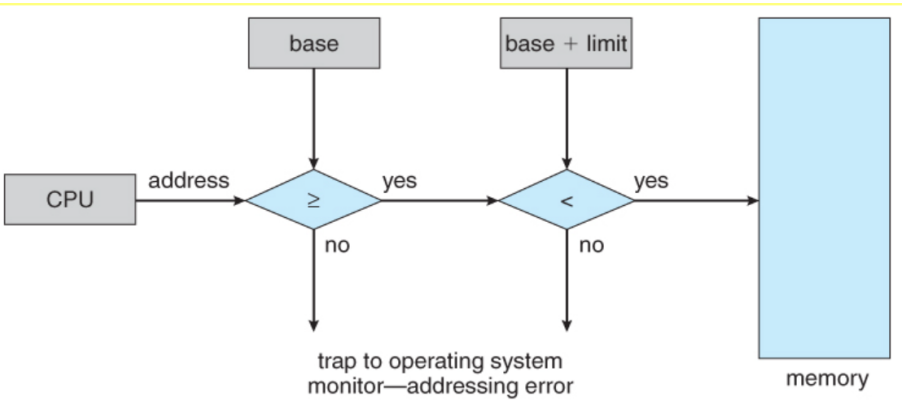
Not only are we concerned with the relative speed of accessing physical memory, but we also must ensure correct operation to protect the operating system from access by user processes and, in

addition, to protect user processes from one another. This protection must be provided by the hardware. We first need to make sure that each process has a separate memory space. To do this, we need the ability to determine the range of legal addresses that the process may access and to ensure that the process can access only these legal addresses. We can provide this protection by using two registers, usually a base and a limit.



The base register holds the smallest legal physical memory address; the limit register specifies the size of the range. For example, if the base register holds 300040 and the limit register is 120900, then the program can legally access all addresses from 300040 through 420939 (inclusive).

Protection of memory space is accomplished by having the CPU hardware compare every address generated in user mode with the registers. Any attempt by a program executing in user mode to access operating-system memory or other users' memory results in a trap to the operating system, which treats the attempt as a fatal error .



This scheme prevents a user program from (accidentally or deliberately) modifying the code or data structures of either the operating system or other users. The base and limit registers can be loaded only by the operating system, which uses a special privileged instruction. Since privileged instructions can be executed only in kernel mode, and since only the operating system executes in kernel mode, only the operating system can load the base and limit registers.

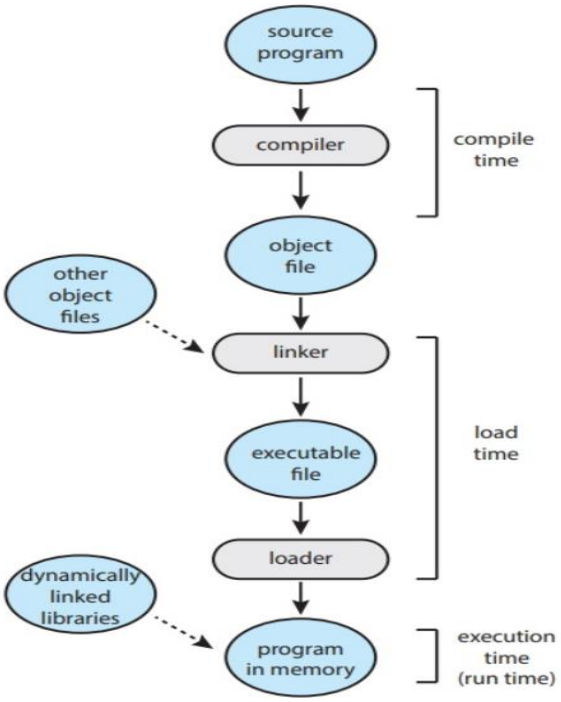
Adress Binding

Address Binding is the association of program instructions and data to the actual physical memory location. There are various types of address binding in the operating system.

There are 3 types of Address Binding:

1. Compile Time Address Binding: If you know at compile time where the process will reside in memory, then absolute code can be generated. For example, if you know that a user process will reside starting at location R, then the generated compiler code will start at that location and extend up from there. If, at some later time, the starting location changes, then it will be necessary to recompile this code. The MS-DOS .COM-format programs are bound at compile time.
2. Load Time Address Binding: If it is not known at compile time where the process will reside in memory, then the compiler must generate relocatable code. In this case, final binding is delayed until load time. If the starting address changes, we need only reload the user code to incorporate this changed value
3. Execution Time Address Binding: If the process can be moved during its execution from one memory segment to another, then binding must be delayed until run time. Special hardware must be available for this scheme to work.

Fig : Multistep processing of a user program.



Logical versus Physical Address Space

Logical Address

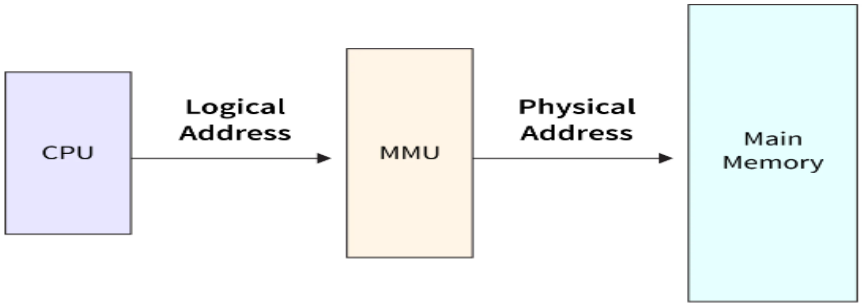
- Logical address is generated by CPU while a program is running.
- The logical address is virtual address as it does not exist physically it is also known as Virtual Address.
- This address is used as a reference to access the physical memory location by CPU.
- Logical Address Space is set of all logical addresses generated by a program.

Physical Address:

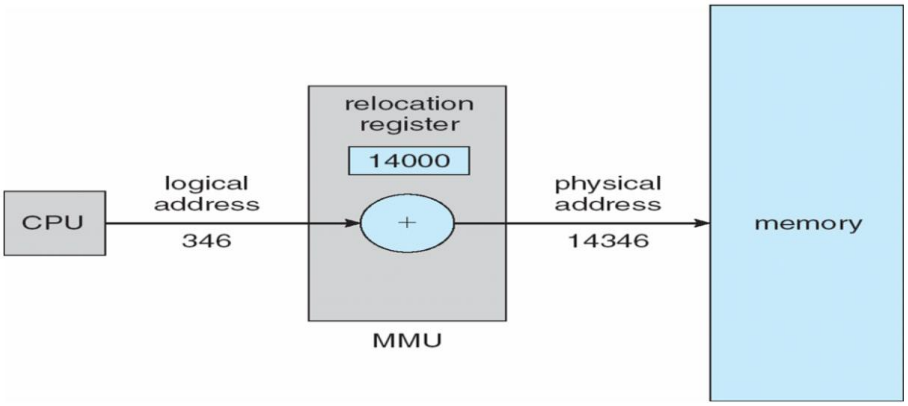
- Physical address identifies a physical location of required data in a memory.
- The user never directly deals with the physical address but can access by its corresponding logical address.
- The set of all physical addresses corresponding to these logical addresses is physical address space

The user program generates the logical address and thinks that the program is running in this logical address but the program needs physical memory for its execution, therefore the logical address must

be mapped to the physical address by MMU before they are used. Memory Management Unit or address translation unit is a hardware device used for mapping logical address to its corresponding physical address.



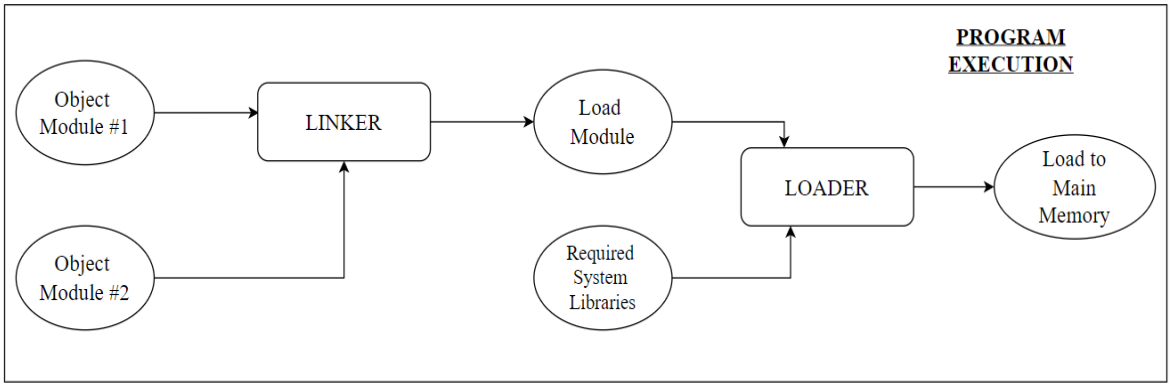
Let us understand the concept of mapping with the help of a simple MMU scheme and that is a **base-register scheme**.



In the above diagram, the base register is termed the **Relocation register.** The value in the relocation register is added to every address that is generated by the user process at the time when the address is sent to the memory.

Suppose the **base is at 14000**, then an attempt by the user to address location **0** is relocated dynamically to **14000**; thus access to location **346** is mapped to **14346.**

Linking and Loading



Working of loading and linking

Linking is a process of collecting and maintaining pieces of code and data into a single file. It is performed by a linker. A linker is special program that combines the object files, generated by compiler/assembler and other pieces of code to originate an executable file has .exe extension.

In the context of operating systems, linking can be done in two main ways: static linking and dynamic linking.

Static Linking: *: Combines all necessary library code with the program code at compile time.*

Process: The linker takes all the object files and libraries and merges them into a single executable file. All external references are resolved during compilation.

Advantages:

- The executable is self-contained, with no dependencies on external libraries at runtime.
- Simplifies distribution as all necessary code is bundled.

Disadvantages:

- Increases the size of the executable file.
- Wastes disk space and memory as each program has its own copy of the library code.
- Updates to libraries require recompilation of the program

Dynamic Linking: *Links the necessary library code to the program at runtime rather than at compile time.*

Process:

- The program includes stubs (With dynamic linking, **we use references to the library in the system where function definitions are available**. The information about the library in which a function definition exists is stored in stubs) instead of the actual library routines
- When the program runs, the stubs locate and load the required routines from shared libraries.
- If a required routine is already in memory, the stub links to it; otherwise, it loads the routine into memory.
- The stub replaces itself with the address of the routine, allowing direct execution in future calls.

Advantages:

- Reduces the size of the executable file.
- Saves disk space and memory as multiple programs can share the same library code in memory.
- Facilitates easy updates to libraries, as programs automatically use the new version without recompilation.
- Multiple versions of a library can coexist, with programs using the appropriate version based on version information.

Disadvantages:

- Requires runtime support from the operating system.
- Potentially introduces runtime overhead during the initial linking process.

Loading

To bring the program from secondary memory to main memory is called Loading. It is performed by a loader. It is a special program that takes the input of executable files from the linker, loads it to the main memory, and prepares this code for execution by a computer. There are two types of loading in the operating system:

Static Loading: Loading the entire program into the main memory before the start of the program execution is called static loading.If static loading is used then accordingly static linking is applied.

- **Dynamic Loading:** Loading the program into the main memory on demand is called dynamic loading. If dynamic loading is used then accordingly dynamic linking is applied. To obtain better memory-space utilization, we can use dynamic loading .With dynamic loading, a routine is not loaded until it is called. All routines are kept on disk in a relocatable load format. The main program is loaded into memory and is executed. When a routine needs to call another routine, the calling routine first checks to see whether the other routine has been loaded. If it has not, the relocatable linking loader

is called to load the desired routine into memory and to update the program's address tables to reflect this change. Then control is passed to the newly loaded routine.

- The advantage of dynamic loading is that an unused routine is never loaded. This method is particularly useful when large amounts of code are needed to handle infrequently occurring cases, such as error routines. In this case, although the total program size may be large, the portion that is used (and hence loaded) may be much smaller.
- Dynamic loading does not require special support from the operating system. It is the responsibility of the users to design their programs to take advantage of such a method. Operating systems may help the programmer, however, by providing library routines to implement dynamic loading.

Unit-IV

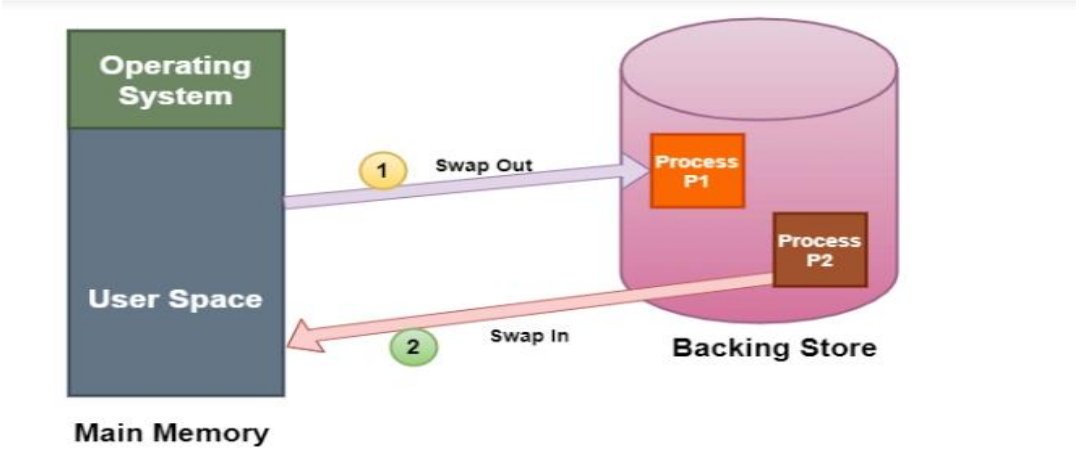
Topic 2: Swapping and Contiguous Allocation

Swapping

Swapping in OS is a memory management technique that temporarily swaps processes from main memory to secondary memory or vice versa which helps to increase the degree of multi-programming and increase main memory utilisation.

There are two steps in the process of Swapping in the operating system:

1. **Swap In:** Swap-in is the process of bringing a process from secondary storage/hard disc to main memory (RAM).
2. **Swap Out:** Swap-out takes the process out of the Main memory and puts it in the secondary storage (backing store)



In the above diagram, process P1 is swap out so the process with more memory requirement or higher priority will be executed to increase the overall efficiency of the operating system. While the process P2 is swap in for its execution from the secondary memory to the main memory(RAM).

▪ In a multi-programming system with a *round-robin scheduling algorithm*, when a time slice expires, memory manager starts to swap out the process that have finished and swap in another process to the memory.

▪ Another swapping policy is algorithm. When any higher priority process arrives for execution, the memory management swaps out the lower priority process so that it can load and execute the higher priority process. After completing the execution of higher priority process, the lower priority

process is swap back and continues its execution. This type of swapping is also called **roll out** (swap out) and **roll in** (swap in).

▪ Normally a previously swapped out process will be swapped back into the same memory space but it depends on the method. If address binding method is execution time binding then a process can be swapped to a different memory space as physical address are computed during execution time.

▪ Swapping requires a fast backing store like a disk. It must be large enough to accommodate copies of all memory images for all users.

▪ When CPU scheduler decides to execute a process, it calls dispatcher that check whether the next process in the queue is in main memory. If it is not and there is not enough free memory available, the dispatcher swaps out any current process and swaps in the desired process.

▪ If we want to swap a process it must be completely idle. A process waiting for an I/O operation should never be swapped to free-up its memory. If we want to swap out process p1 and swap in process p2, the I/O operation of process p1 might attempt to use memory that belongs to process p2. Main solution of this problem is never to swap a process with pending I/O .

Example: Suppose the user process's size is 2048KB and is a standard hard disk where swapping has a data transfer rate of 1Mbps. Now we will calculate how long it will take to transfer from main memory to secondary memory.

User process size is 2048Kb

Data transfer rate is 1Mbps = 1024 kbps

Time = process size / transfer rate

= 2048 / 1024 = 2 seconds = 2000 milliseconds

Now taking swap-in and swap-out time, the process will take 4000 milliseconds.

Advantages of Swapping

1. It helps the CPU to manage multiple processes within a single main memory.
2. It helps to create and use virtual memory.
3. Swapping allows the CPU to perform multiple tasks simultaneously. Therefore, processes do not have to wait very long before they are executed.
4. It improves the main memory utilization.

Disadvantages of Swapping

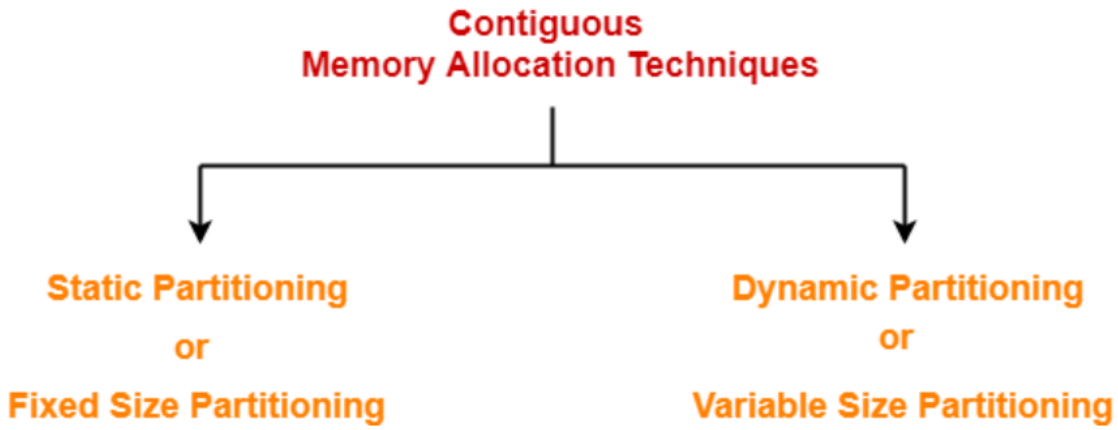
1. If the computer system loses power, the user may lose all information related to the program in case of substantial swapping activity.
2. If the swapping algorithm is not good, the composite method can increase the number of Page Fault and decrease the overall processing performance.

Contiguous Memory Allocation

When a program or process is to be executed, it needs some space in the memory. For this reason, some part of the memory has to be allotted to a process according to its requirements. This process is called memory allocation.One such memory allocation technique is contiguous memory allocation. As the name implies, we allocate contiguous blocks of memory to each process using this technique.

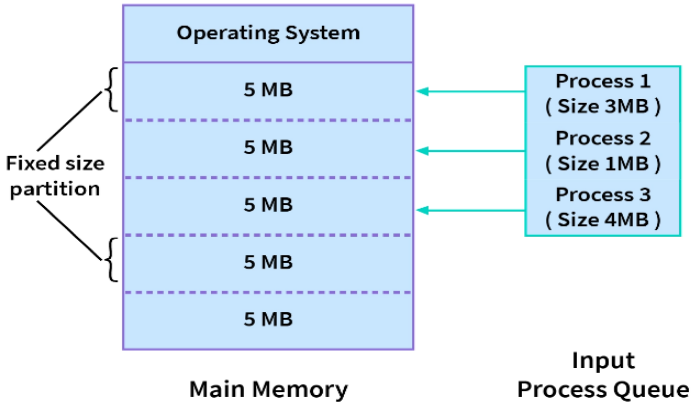
Contiguous Memory Allocation Techniques

Whenever a process has to be allocated space in the memory, following the contiguous memory allocation technique, we have to allot the process a continuous empty block of space to reside. This allocation can be done in two ways:

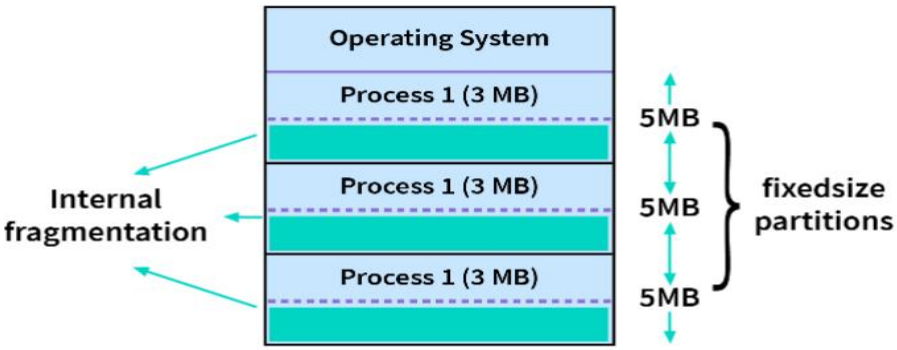


Fixed-size/Static Partitioning

In this type of contiguous memory allocation technique, **each process is allotted a fixed-size continuous block in the main memory.** That means there will be continuous blocks of fixed size into which the complete memory will be divided, and each time a process comes in, it will be allotted one of the free blocks. Because irrespective of the size of the process, each is allotted a block of the same size memory space. This technique is also called static partitioning.



In the diagram above, we have 3 processes in the input queue that have to be allotted space in the memory. As we are following the fixed-size partition technique, the memory has fixed-sized blocks.



The first process, which is of size 3MB is also allotted a 5MB block, and the second process, which is of size 1MB, is also allotted a 5MB block, and the 4MB process is also allotted a 5MB block. So, the process size doesn't matter. Each is allotted the same fixed-size memory block.

Note: The number of processes that can stay in the memory at once is called the degree of multiprogramming. Hence, the degree of multiprogramming of the system is decided by the number of blocks created in the memory.

Advantages

1. Because all of the blocks are the same size, this scheme is simple to implement. All we have to do now is divide the memory into fixed blocks and assign processes to them.
2. It is easy to keep track of how many blocks of memory are left, which in turn decides how many more processes can be given space in the memory.

3. As at a time multiple processes can be kept in the memory, this scheme can be implemented in a system that needs multiprogramming.

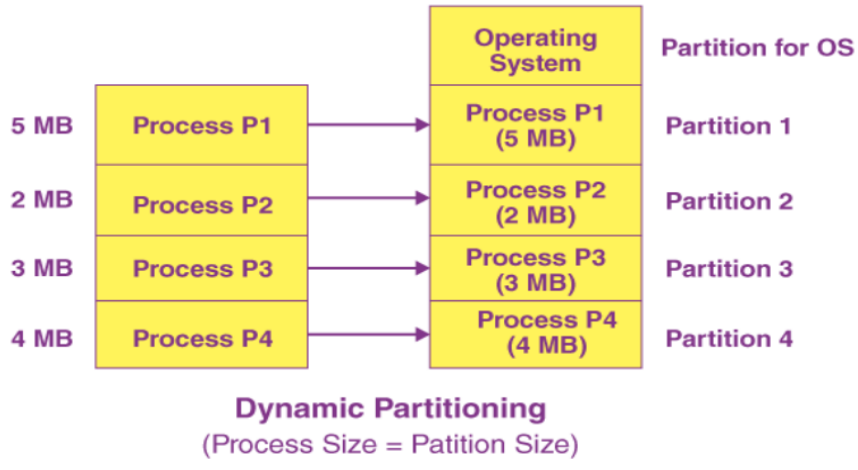
Disadvantages

1. As the size of the blocks is fixed, we will not be able to allot space to a process that has a greater size than the block.
2. The size of the blocks decides the degree of multiprogramming, and only that many processes can remain in the memory at once as the number of blocks.
3. If the size of the block is greater than the size of the process, we have no other choice but to assign the process to this block, but this will lead to much empty space left behind in the block. This empty space could've been used to accommodate a different process. This is called internal fragmentation. Hence, this technique may lead to space wastage.

The problem of internal fragmentation may arise due to the fixed sizes of the memory blocks. It may be solved by assigning space to the process via dynamic partitioning. Dynamic partitioning allocates only the amount of space requested by the process. As a result, there is no internal fragmentation.

Variable-size /Dynamic Partitioning

In this type of contiguous memory allocation technique, **no fixed blocks or partitions are made in the memory.** Instead, each process is allotted a variable-sized block depending upon its requirements. That means, that whenever a new process wants some space in the memory, if available, this amount of space is allotted to it. Hence, the size of each block depends on the size and requirements of the process which occupies it.



Dynamic Partitioning
(Process Size = Patition Size)

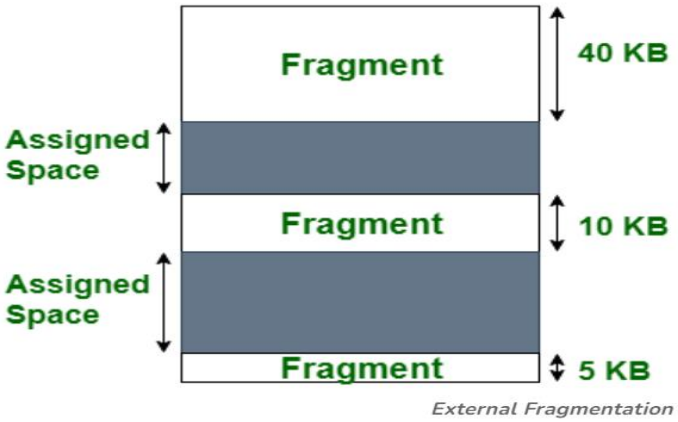
The very first partition has to be reserved for the OS. The left space gets divided into various parts. The actual size of every partition would be equal to the process size. The size of the partition varies according to the requirement of the process. This way, internal fragmentation can be easily avoided.

Advantages

1. As the processes have blocks of space allotted to them as per their requirements, there is no internal fragmentation. Hence, there is no memory wastage in this scheme.
2. The number of processes that can be in the memory at once will depend upon how many processes are in the memory and how much space they occupy. Hence, it will be different for different cases and will be dynamic.
3. As there are no blocks that are of fixed size, even a process of big size can be allotted space.

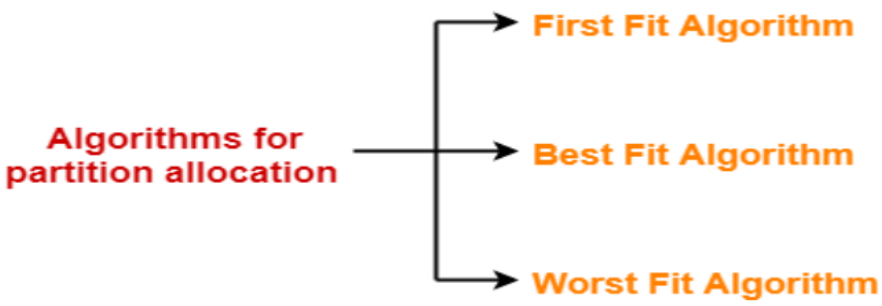
Disadvantages

1. Because this approach is dynamic, a variable-size partition scheme is difficult to implement.
2. It is difficult to keep track of processes and the remaining space in the memory.
3. Since there is no internal fragmentation, it doesn't mean there would be no external fragmentation either.Now, let us consider three processes, i.e. P1 (1 MB), P2 (3 MB) and P3 (1 MB), that are being loaded in their respective partitions of the main memory. P1 ,P2 and P3 get completed after some time, and the assigned space gets freed. Here, we have three unused partitions available (40K,10K,5K) in the main memory. However, we can't use them to load 55K process P4 in the memory because they aren't located contiguously.



External Fragmentation

Partition Allocation Strategies/Algorithms

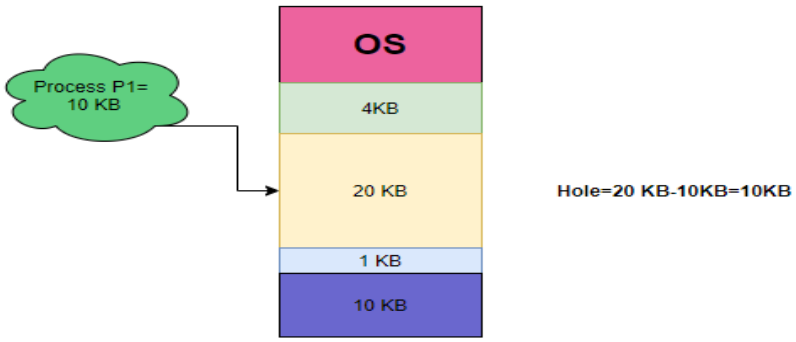


1.First Fit Allocation

According to this strategy, allocate the **first hole or first free partition** to the process that is big enough. This searching can start either from the beginning of the set of holes or from the location where the previous first-fit search ended.Searching can be stopped as soon as we find a free hole that is large enough.

Let us take a look at the example given below:

Process P1 of size 10KB has arrived and then the first hole that is enough to meet the requirements of size 20KB is chosen and allocated to the process.

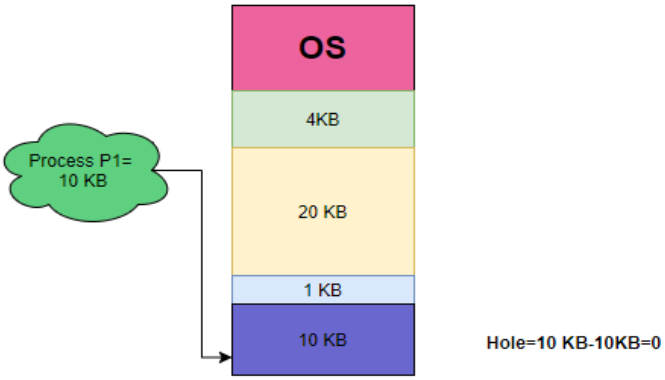


2. Best Fit Allocation

With this strategy, the smallest free partition/ hole that is big enough and meets the requirements of the process is allocated to the process. This strategy searches the entire list of free partitions/holes in order to find a hole whose size is either greater than or equal to the size of the process.

Let us take a look at the example given below:

Process P1 of size 10KB is arrived and then the smallest hole to meet the requirements of size 10 KB is chosen and allocated to the process.

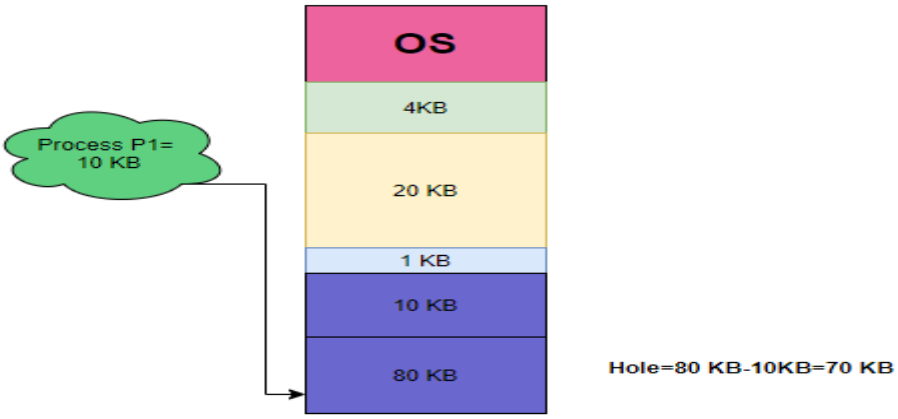


3. Worst Fit Allocation

With this strategy, the Largest free partition/ hole that meets the requirements of the process is allocated to the process. It is done so that the portion that is left is big enough to be useful. This strategy is just the opposite of Worst Fit.This strategy searches the entire list of holes in order to find the largest hole and then allocate the largest hole to process.

Let us take a look at the example given below:

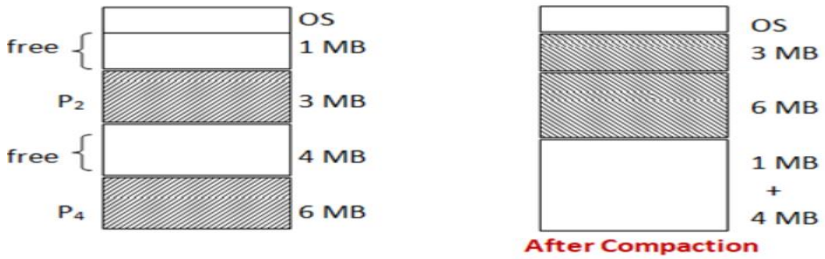
Process P1 of size 10KB has arrived and then the largest hole of size 80 KB is chosen and allocated to the process.



Compaction

In memory management, swapping creates multiple fragments in the memory because of the processes moving in and out. Compaction is a method for removing external fragmentation.

Compaction is a technique to collect all the free memory present in the form of fragments into one large chunk of free memory, which can be used to run other processes.It does that by moving all the processes towards one end of the memory and all the available free space towards the other end of the memory so that it becomes contiguous.



The compaction process usually consists of two steps:

1. Copying all pages that are not in use to one large contiguous area.
2. Then, write the pages that are in use into the newly freed space.

Advantages of Compaction

- Reduces external fragmentation.
- Make memory usage efficient.
- Memory becomes contiguous.
- Since memory becomes contiguous more processes can be loaded to memory, thereby increasing scalability of OS.
- Improves memory utilization as their is less gap between memory blocks.

Disadvantages of Compaction

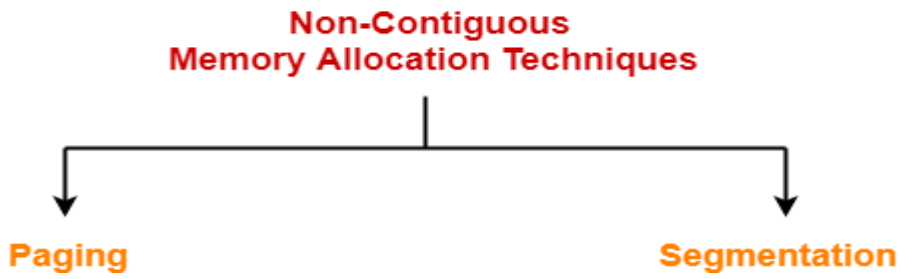
- System efficiency reduces and latency is increased.
- A huge amount of time is wasted in performing compaction.
- CPU sits idle for a long time.
- Not always easy to perform compaction.
- It may cause deadlocks since it disturbs the memory allocation process.

Unit 4

Topic 3: Paging-I

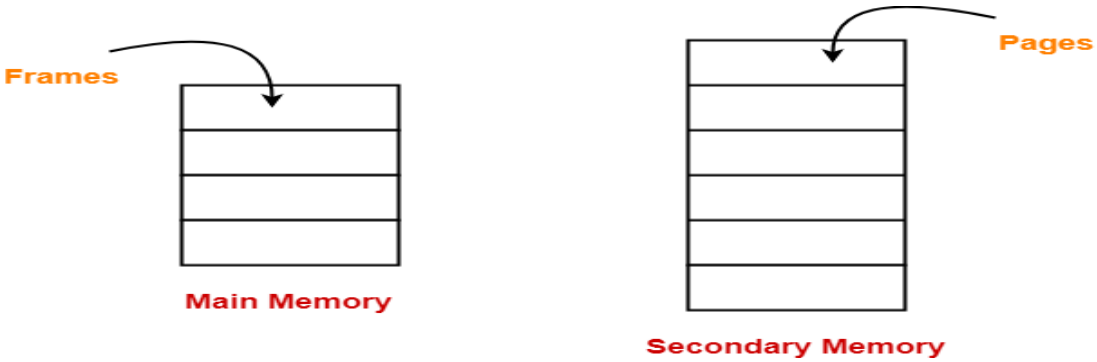
To overcome the problem of External fragmentation in contiguous memory allocation is we go for non-contiguous memory allocation. It allows to store parts of a single process in a non-contiguous fashion. Thus, different parts of the same process can be stored at different places in the main memory.

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



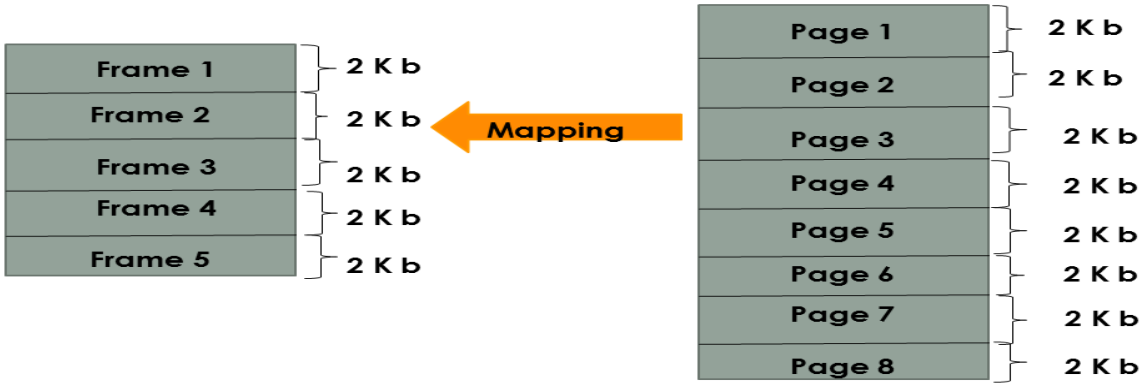
Paging

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



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

Basic **Method**



CPU always generates a logical address consisting of two parts-

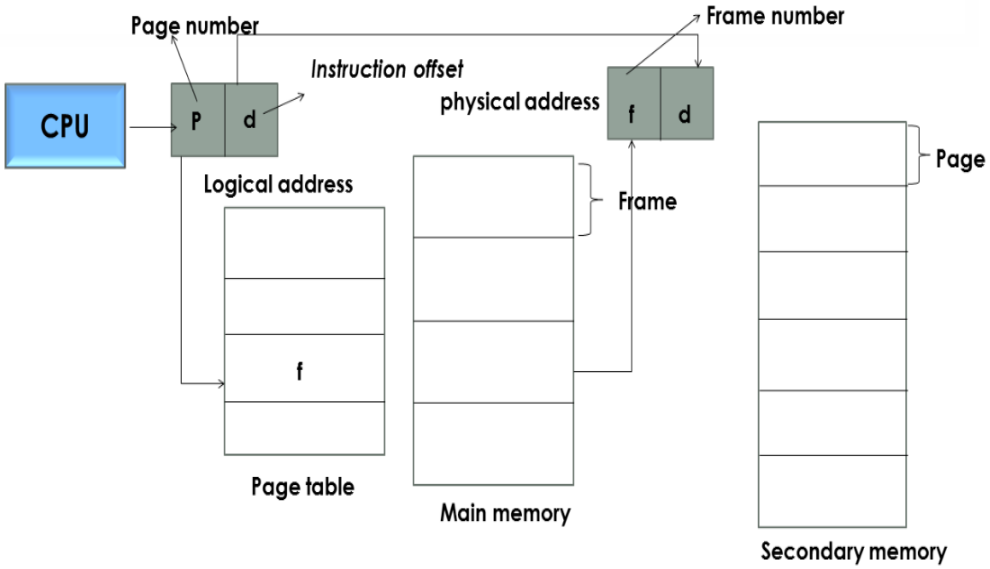
Page Number: specifies the specific page of the process from which CPU wants to read the data.

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

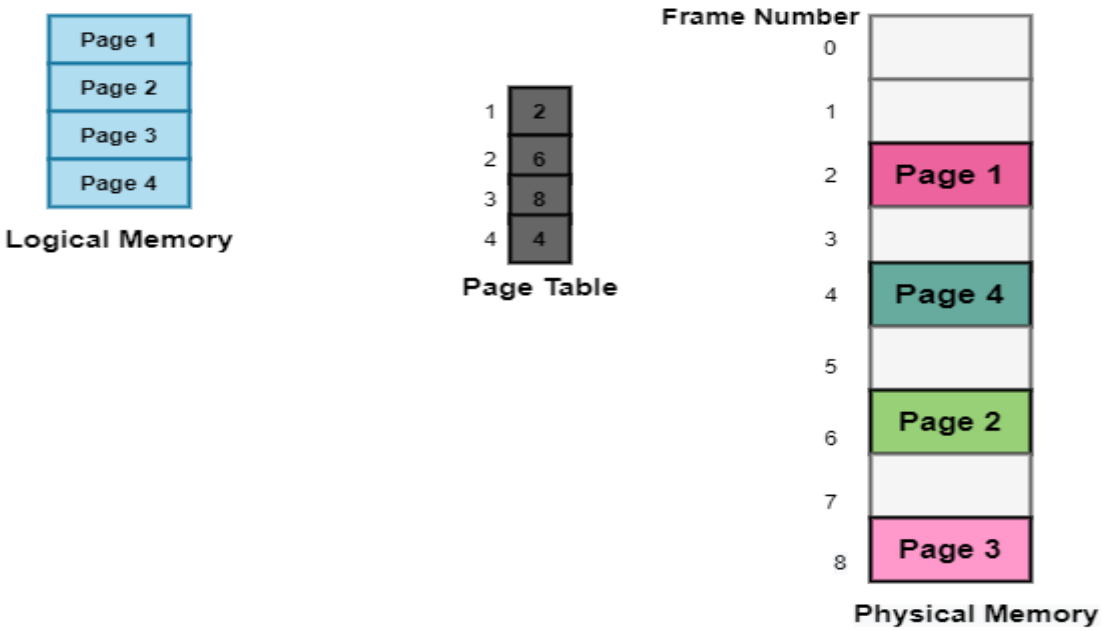
A physical address is needed to access the main memory and consists of and offset .

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

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



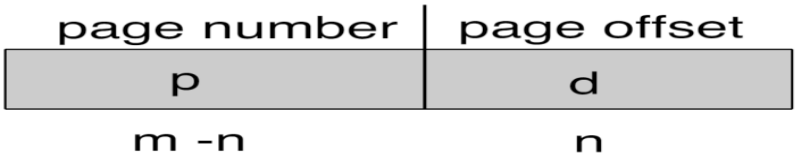
The logical address that is generated by the CPU is translated into the physical address with the help of the page table.Thus page table mainly provides the corresponding frame number where that page is stored in the main memory.



The above diagram shows the paging model of Physical and logical memory.

The page size (like the frame size) is defined by the hardware. The size of a page is typically a power of 2, varying between 512 bytes and 16 MB per page, depending on the computer architecture. The selection of a power of 2 as a page size makes the translation of a logical address into a page number and page offset particularly easy.

If the size of the logical address space is 2^m, and a page size is 2^n addressing units (bytes or words) then the high-order m- n bits of a logical address designate the page number, and the n low-order bits designate the page offset. Thus, the logical address is as follows:



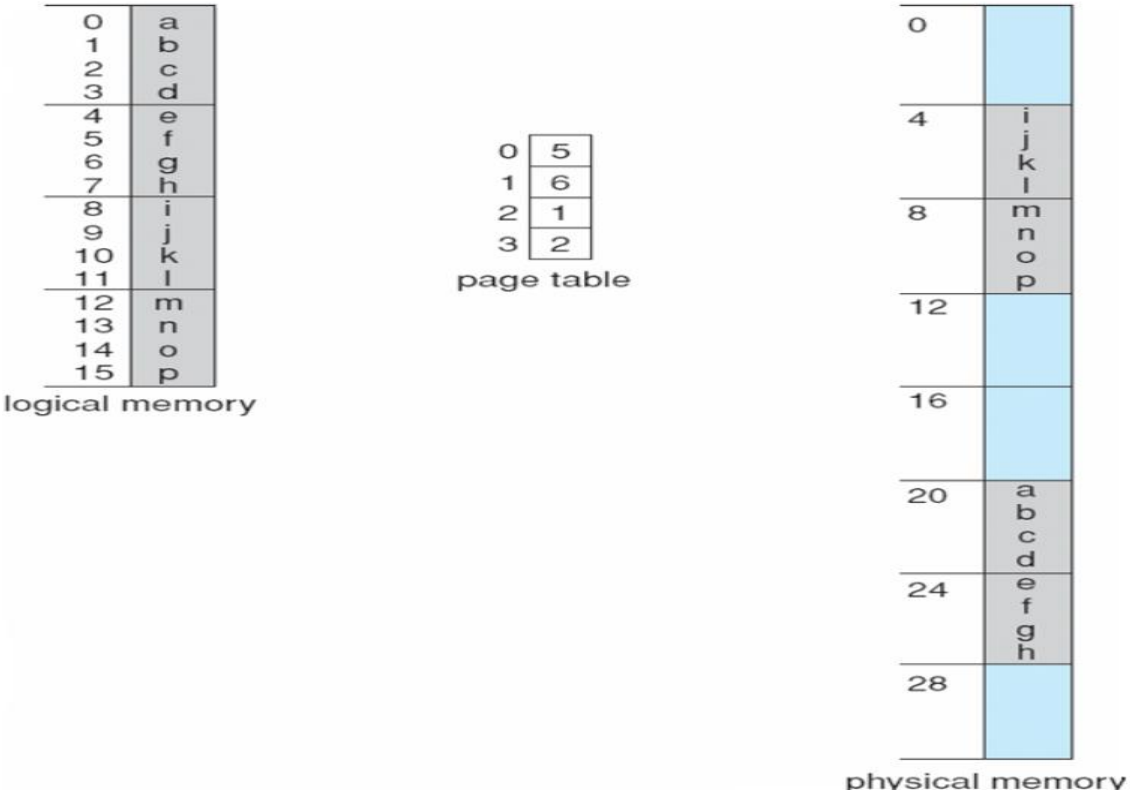
Example, consider the memory in Figure below. Here, in the logical address, n= 2 and m = 4. Using a page size of 4 bytes and a physical memory of 32 bytes (8 pages), we show how the user's view of memory can be mapped into physical memory.

♣ Logical address 0 is page 0, offset 0. Indexing into the page table, we find that page 0 is in frame 5. Thus, logical address 0 maps to physical address 20 [= (5 × 4) + 0].

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

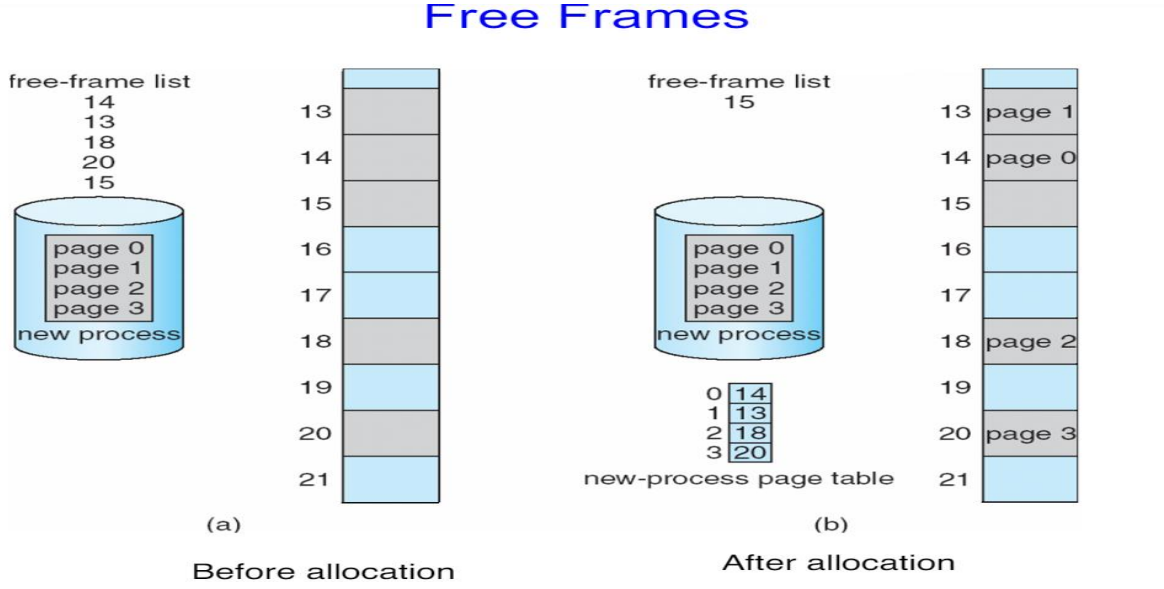
♣ Logical address 4 is page 1, offset 0; according to the page table, page 1 is mapped to frame 6. Thus, logical address 4 maps to physical address 24 [= (6 × 4) + 0].

♣ Logical address 13 maps to physical address 9.



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

When a process arrives in the system to be executed, its size, expressed in pages, is examined. Each page of the process needs one frame. Thus, if the process requires n pages, at least n frames must be available in memory. If n frames are available, they are allocated to this arriving process. The first page of the process is loaded into one of the allocated frames, and the frame number is put in the page table for this process. The next page is loaded into another frame, its frame number is put into the page table, and so on



Hardware Support

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

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

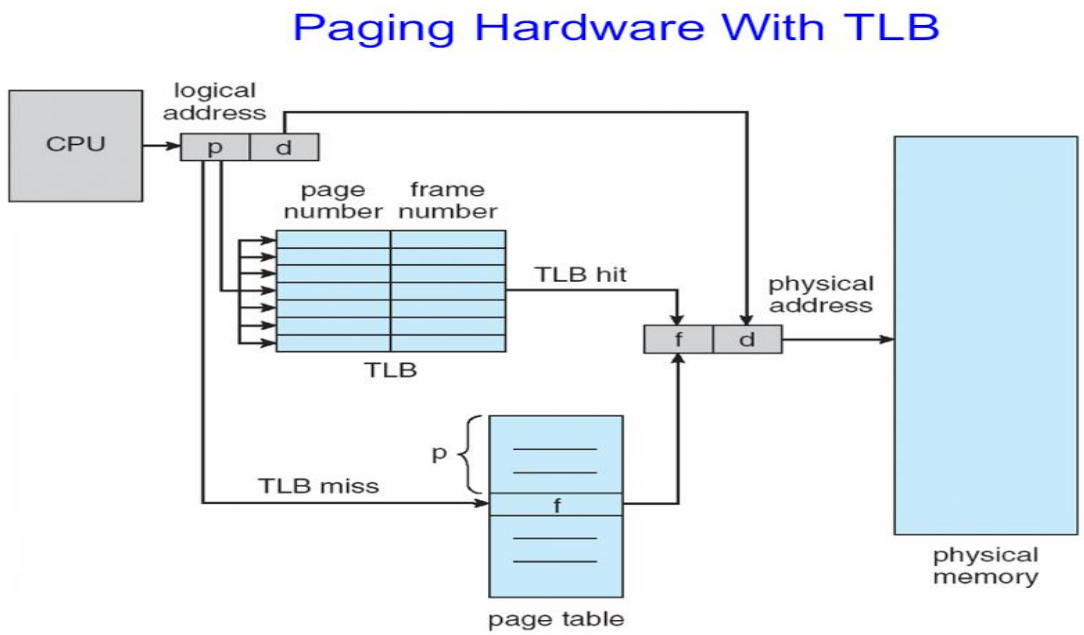
The idea used here is, to place the page table entries in registers, for each request generated from the CPU (virtual address), it will be matched to the appropriate page number of the page table, which will now tell where in the main memory that corresponding page resides. Everything seems right here,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

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

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

The two-memory access problem can be solved by the use of a special fast-lookup hardware cache called *associative memory* or *translation look-aside buffers (TLBs)*. Each entry in the TLB consists of two parts: **a key** (or tag) and **a value**. When the **associative memory** is presented with an item, the item is compared with all keys simultaneously. If the item is found, the corresponding **value field** is returned. Typically, the number of entries in a **TLB is small**, often numbering between 64 and 1,0241.

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



- The TLB contains only a few of the **page-table** entries. When a logical address is generated by the CPU, its page number is presented to the TLB.
- if the page number is found (known as **TLB hit**), its frame number is immediately available and is used to access memory.
- If the page number is not found in the TLB (known as **TLB miss**) a memory reference to the page table must be made.
- when the frame number is obtained, we can use it to access memory.
- In addition, we add the page number and frame number to the TLB, so that they will be found quickly on the **next reference**.
- If the TLB is already full of entries, the operating system must **select one** for replacement.
- Replacement policies range from **least recently used** (LRU) to random.

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

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

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

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

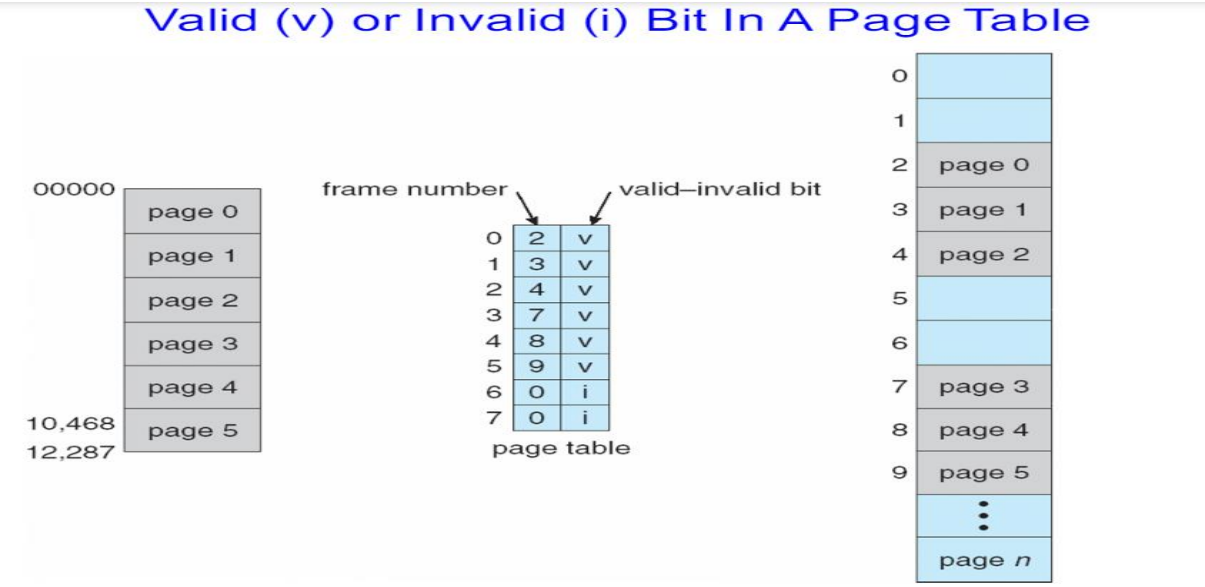
Protection

- Memory protection in a paged environment is accomplished by **protection bits** associated with each frame. Normally, these bits are kept in the page table. One bit can define a page to be read-write or read-only.
- Every reference to memory goes through the page table to find the correct frame number. At the same time that the physical address is being computed, the protection bits can be checked to verify that no writes are being made to a read-only page.
- An attempt to write to a read-only page causes a hardware trap to the operating system (or memory-protection violation).
- One additional bit is generally attached to each entry in the page table: a **valid-invalid** bit.

When this bit is set to ``valid'', the associated page is in the process's logical address space and is thus a legal (or valid) page.

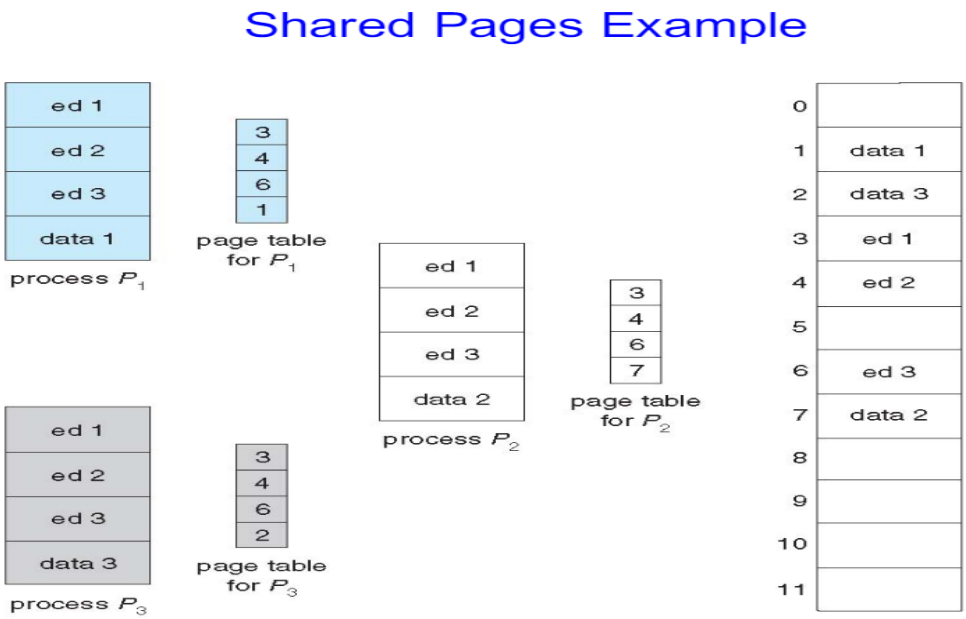
When the bit is set to ``invalid'', the page is not in the process's logical address space.

- Illegal addresses are trapped by use of the valid-invalid bit. The OS sets this bit for each page to allow or disallow access to the page.
- Suppose, for example, that in a system with a 14-bit address space (0 to 16383), we have a program that should use only addresses 0 to 10468.
  - item Addresses in pages 0, 1, 2, 3, 4, and 5 are mapped normally through the page table.
  - Any attempt to generate an address in pages 6 or 7, however, will find that the valid-invalid bit is set to invalid, and the computer will trap to the OS (invalid page reference).



Shared Pages

An advantage of paging is the possibility of sharing common code. This consideration is particularly important in a time-sharing environment. Consider a system that supports 40 users, each of whom executes a text editor. If the text editor consists of 150 KB of code and 50 KB of data space, we need 8,000 KB to support the 40 users. If the code is reentrant code(or pure code however, it can be shared, as shown in Figure)



Here we see a three-page editor-each page 50 KB in size (the large page size is used to simplify the figure)-being shared among three processes. Each process has its own data page. Re-entrant code is non-self-modifying code: it never changes during execution. Thus, two or more processes can execute

the same code at the same time. Each process has its own copy of registers and data storage to hold the data for the process's execution. The data for two different processes will be of course, be different. Only one copy of the editor need be kept in physical memory.

Each user's page table maps onto the same physical copy of the editor, but data pages are mapped onto different frames. Thus, to support 40 users, we need only one copy of the editor (150 KB), plus 40 copies of the 50 KB of data space per user. The total space required is now 2)50 KB instead of 8,000 KB-a significant savings. Other heavily used programs can also be shared -compilers, window systems, run-time libraries, database systems, and so on. To be sharable, the code must be reentrant. The read-only nature of shared code should not be left to the correctness of the code; the operating system should enforce this property.

Unit-IV

Topic 4: Paging-II (Structure of Page Table)

Page tables can consume a significant amount of memory.

• For example: A 32-bit virtual address space using 4 kB pages.

Page size = 4* 2^{10}(4 KB) =2^2*2^{10}=2^{12} bytes

Space for page numbers (Logical Address space -page size) = 2^{32} - 2^{12}= 2^{20} bytes

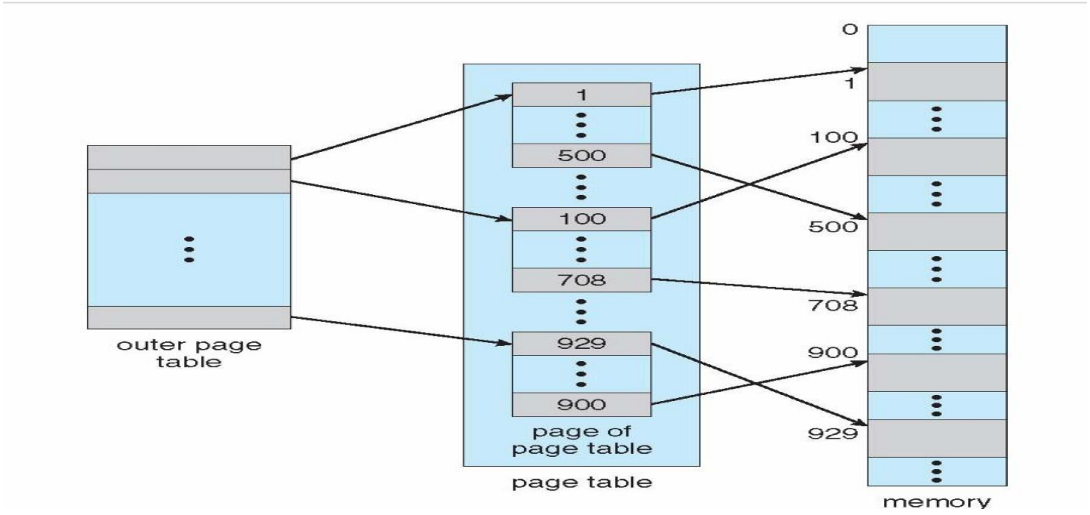
• So page table may consists of up to 1 million entries, which cannot be stored continuously in main memory. There are many techniques used for storing these page tables

Some of the common techniques that are used for structuring the Page table are as follows:

1. Hierarchical Paging
2. Hashed Page Tables
3. Inverted Page Tables

Hierarchical Paging

- Another name for Hierarchical Paging is multilevel paging.
- There might be a case where the page table is too big to fit in a contiguous space, so we may have a hierarchy with several levels.
- In this type of Paging the logical address space is broke up into Multiple page tables.
- Hierarchical Paging is one of the simplest techniques and for this purpose, a two-level page table and three-level page table can be used.



Two Level Page Table

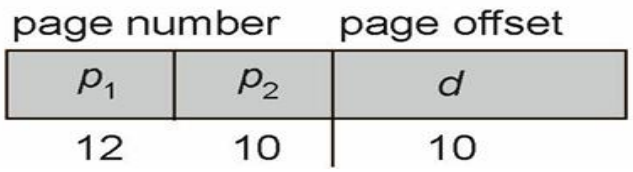
Consider a system having 32-bit logical address space and a page size of 1 KB and it is further divided into:

Page size = $1 * 2^{10}$(1 KB)=$2^0 * 2^{10}=2^{10}$ bytes=10 bits for (displacement / Offset)

Total bits for Logical address=32

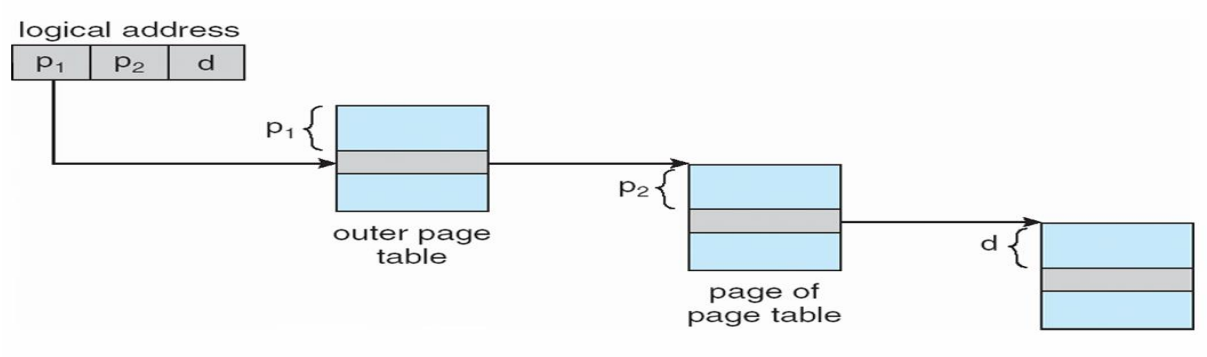
No.of bits for page number=bits for logical address-offset bits=(32-10)=22

22 is split up into two parts=12+10



In the above diagram, P1 is an index into the **Outer Page** table. P2 indicates the displacement within the page of the **Inner page** Table. As address translation works from outer page table inward so is known as **forward-mapped Page Table**.

Below given figure below shows the Address Translation scheme for a two-level page table



Three Level Page Table

For a system with 64-bit logical address space, a two-level paging scheme is not appropriate. Let us suppose that the page size, in this case, is 4KB.If in this case, we will use the two-page level scheme then the addresses will look like this:

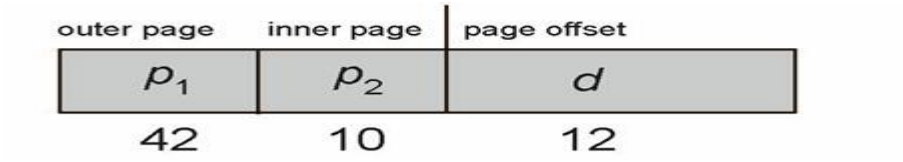
Page size = $4 * 2^{10}$(4 KB)=$2^2 * 2^{10}=2^{12}$ bytes=12 bits for (displacement / Offset)

Total bits for Logical address=64

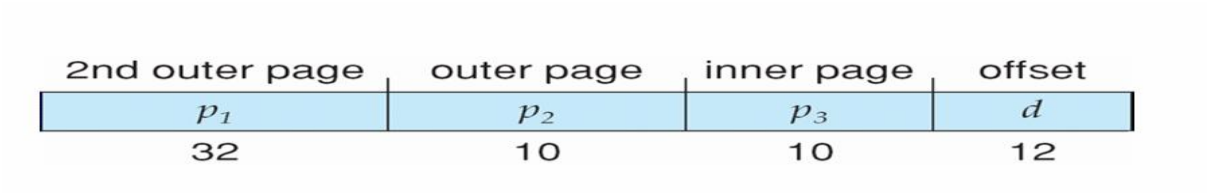
No.of bits for page number=bits for logical address-offset bits=(64-12)=52

52 is split up into two parts=42+10

The outer page table consists of 242 entries, or 244 bytes. The obvious way to avoid such a large table is to divide the outer page table into smaller pieces. (This approach is also used on some 32-bit processors for added flexibility and efficiency.)



We can divide the outer page table in various ways. We can page the outer page table, giving us a three-level paging scheme. Suppose that the outer page table is made up of standard-size pages (210 entries, or 212 bytes). In this case, a 64-bit address space is still daunting:



Hashed Page Table

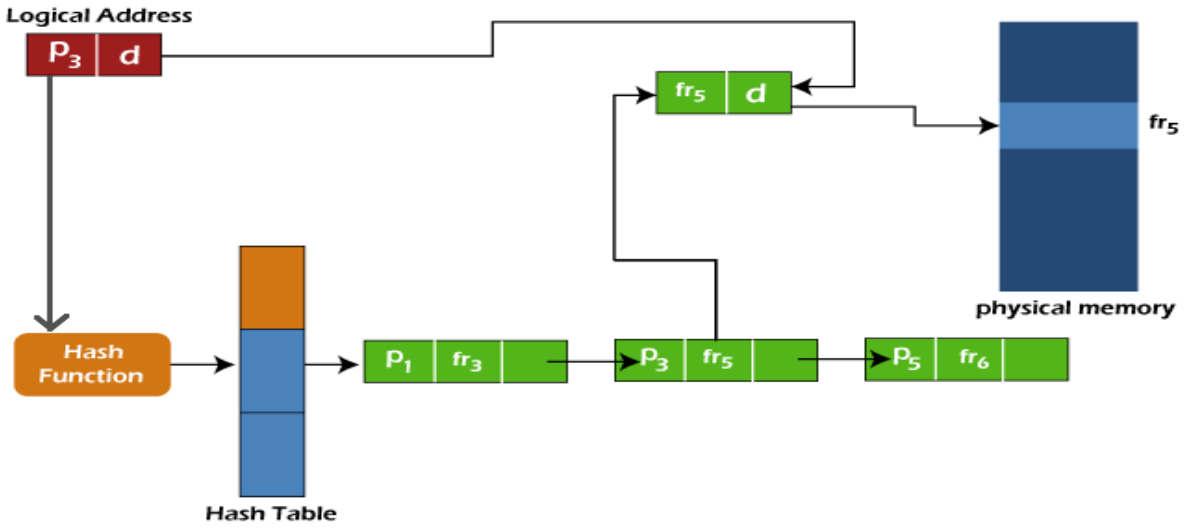
Hashed page tables are a technique for structuring page tables in memory. Hashed page tables are common in address spaces greater than 32 bits.In a hashed page table, the virtual addresses are hashed into the hash table. Each element in the table comprises a linked list of elements to avoid collisions. The hash value used is the virtual page number, i.e., all the bits that are not part of the page offset.

For each element in the hash table, there are three fields available,

1. The virtual Page Number (which is the hash value).
2. The value of the mapped page frame.
3. A pointer to the next element in the linked list.

Working of Hashed Page Table

We would understand the working of the hashed page table with the help of an example. The CPU generates a logical address for the page it needs. Now, this logical address needs to be mapped to the physical address. This logical address has two entries, i.e., a page number (P_3) and an offset, as shown below.



- The page number from the logical address is directed to the hash function.
- The hash function produces a hash value corresponding to the page number.
- This hash value directs to an entry in the hash table.
- As we have studied earlier, each entry in the hash table has a link list. Here the page number is compared with the first element's first entry. If a match is found, then the second entry is checked.

In this example, the logical address includes page number P_3 which does not match the first element of the link list as it includes page number P_1. So we will move ahead and check the next element; now, this element has a page number entry, i.e., P3, so further, we will check the frame entry of the element, which is fr_5. We will append the offset provided in the logical address to this frame number to reach the page's physical address. So, this is how the hashed page table works to map the logical address to the physical address.

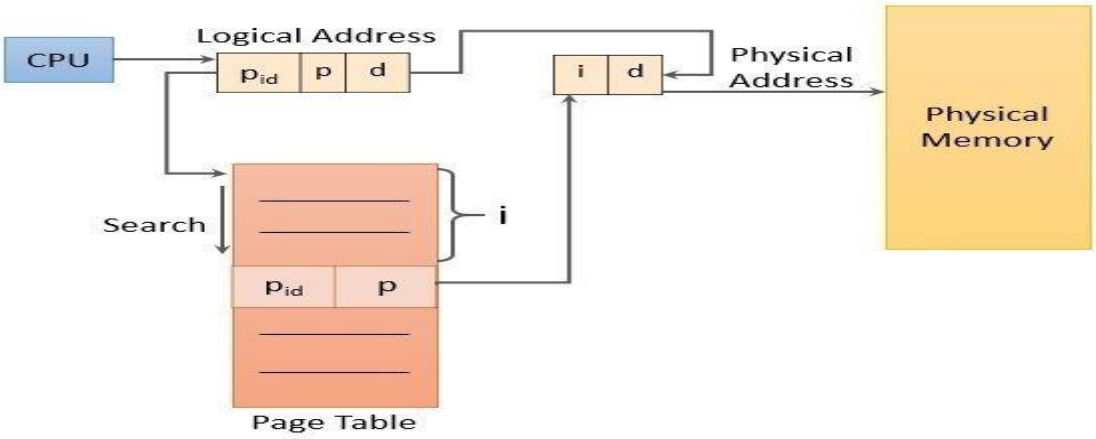
A variation of this scheme that is favorable for 64-bit address spaces has been proposed. This variation uses clustered page table which are similar to hashed page tables except that each entry in the hash table refers to several pages (such as 16) rather than a single page. Therefore, a single page-table entry can store the mappings for multiple physical-page frames. Clustered page tables are particularly useful for sparse address spaces, where memory references are noncontiguous and scattered throughout the address space.

Inverted Page Table

The concept of normal paging says that every process maintains its own page table which includes the entries of all the pages belonging to the process. The large process may have a page table with millions of entries. Such a page table consumes a large amount of memory.

Consider we have six processes in execution. So, six processes will have some or the other of their page in the main memory which would compel their page tables also to be in the main memory consuming a lot of space. This is the drawback of the paging concept.

The inverted page table is the solution to this wastage of memory. The concept of an inverted page table involves the existence of single-page table which has entries of all the pages (may they belong to different processes) in main memory along with the information of the process to which they are associated. To get a better understanding consider the figure below of inverted page table.



Structure of Inverted Page Table

The CPU generates the logical address for the page it needs to access. This time the logical address consists of three entries process id, page number and the offset. The process id identifies the process, of which the page has been demanded, page number indicates which page of the process has been asked for and the offset value indicates the displacement required.

The match of process id along with associated page number is searched in the page table and say if the search is found at the i^{th} entry of page table then i and offset together generates the physical address for the requested page. This is how the logical address is mapped to a physical address using the inverted page table.

Though the inverted page table reduces the wastage of memory but it increases the search time. This is because the entries in an inverted page table are sorted on the basis of physical address whereas the lookup is performed using logical address. It happens sometimes that the entire table is searched to find the match.

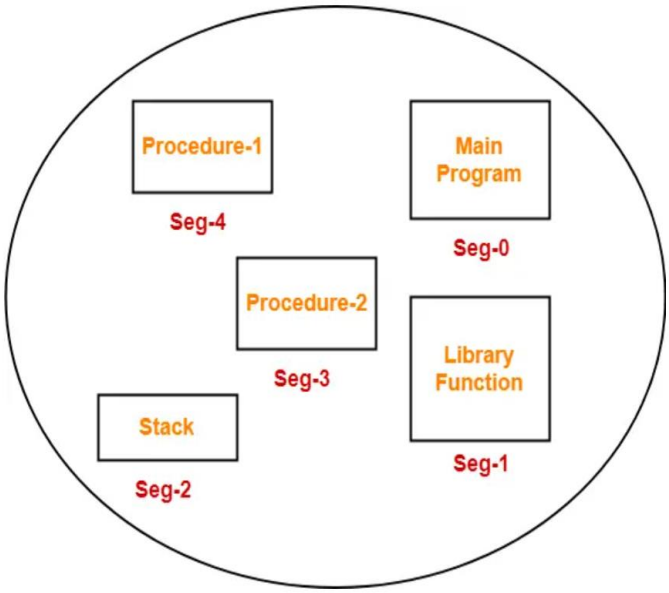
Unit-IV

Topic 5: Segmentation

Like Paging, Segmentation is another non-contiguous memory allocation technique. In Paging, the process was divided into equal-sized pages irrespective of the fact that what is inside the pages. It also divides some relative parts of a process into different pages which should be loaded on the same page. It decreases the efficiency of the system and doesn't give the user's view of a process. In Segmentation, similar modules are loaded in the same segments. It gives the user's view of a process and also increases the efficiency of the system as compared to Paging.

Basic Method

A Program is basically a collection of segments. And a segment is a logical unit such as: main program, procedure, function, method, object, local variable and global variables, symbol table, common block, stack and arrays. A computer system that is using segmentation has a logical address space that can be viewed as multiple segments.



And the size of the segment is of the variable that is it may grow or shrink. As we had already told you that during the execution each segment has a name and length. And the address mainly specifies both thing name of the segment and the displacement within the segment. Therefore the user specifies each address with the help of two quantities: segment name and offset.For simplified Implementation segments are numbered; thus referred to as segment number rather than segment name.Thus the logical address consists of two tuples:**<segment-number,offset>** where,

Segment Number(s):

Segment Number is used to represent the number of bits that are required to represent the segment.

Offset(d)

Segment offset is used to represent the number of bits that are required to represent the size of the segment.

Hardware

The details about each segment are stored in a table called a segment table. Segment table is stored in one (or many) of the segments.In the segment table each entry has :

Segment Base/base address:

The segment base mainly contain**s** the starting physical address where the segments reside in the memory.

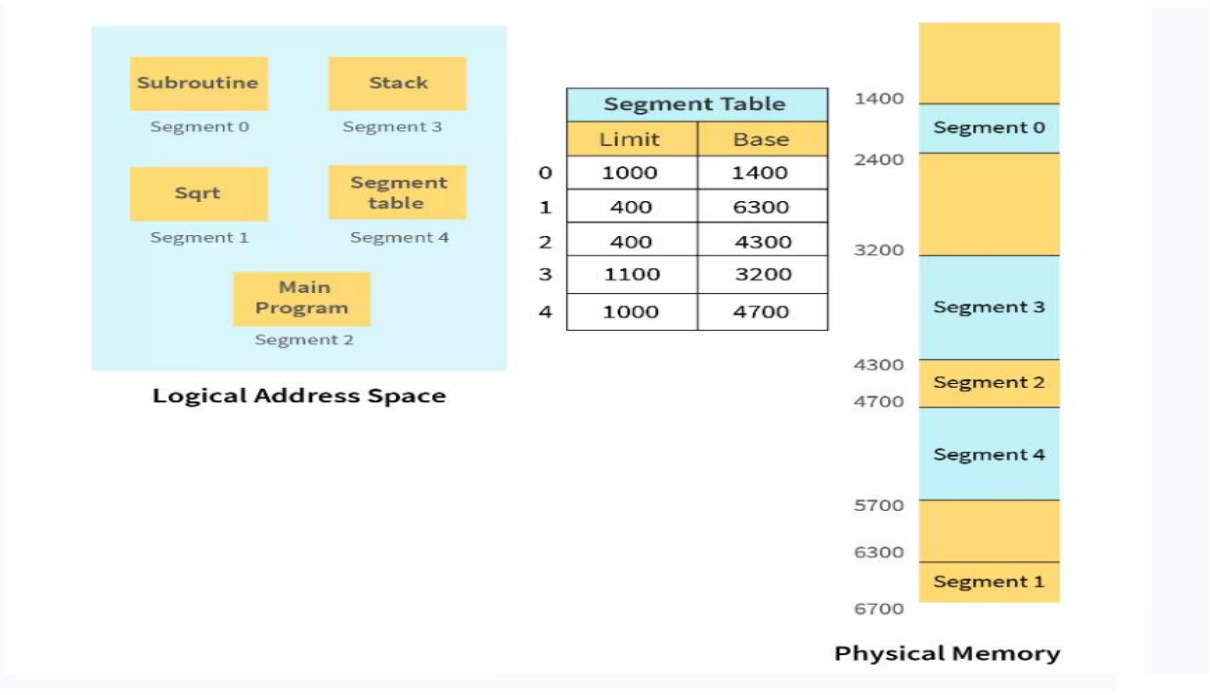
Segment Limit:

The segment limit is mainly used to specify the length of the segment.

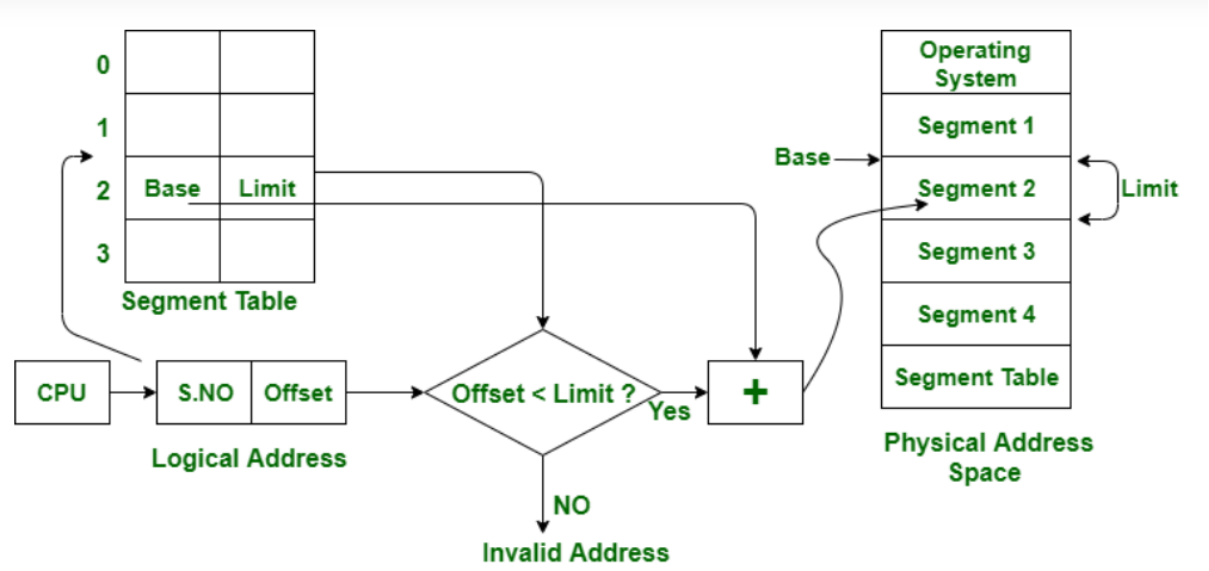
Let's take the example of segmentation to understand how it works.

Let us assume we have five segments namely: Segment-0, Segment-1, Segment-2, Segment-3, and Segment-4. Initially, before the execution of the process, all the segments of the process are stored in the physical memory space. We have a segment table as well. The segment table contains the beginning entry address of each segment (denoted by **base**). The segment table also contains the length of each of the segments (denoted by **limit**).

As shown in the image below, the base address of Segment-0 is 1400 and its length is 1000, the base address of Segment-1 is 6300 and its length is 400, the base address of Segment-2 is 4300 and its length is 400, and so on.The pictorial representation of the above segmentation with its segment table is shown below.



With the help of segment table and hardware assistance, the operating system can easily translate a logical address into physical address on execution of a program. The **Segment number** is mapped to the segment table. The limit of the respective segment is compared with the offset. If the offset is less than the limit then the address is valid otherwise it throws an error as the address is invalid. In the case of valid addresses, the base address of the segment is added to the offset to get the physical address of the actual word in the main memory.



Advantages of Segmentation

1. No internal fragmentation
2. Average Segment Size is larger than the actual page size.
3. Less overhead
4. It is easier to relocate segments than entire address space.
5. The segment table is of lesser size as compared to the page table in paging.

Disadvantages

1. It can have external fragmentation.
2. it is difficult to allocate contiguous memory to variable sized partition.
3. Costly memory management algorithms.

Paging Vs Segmentation

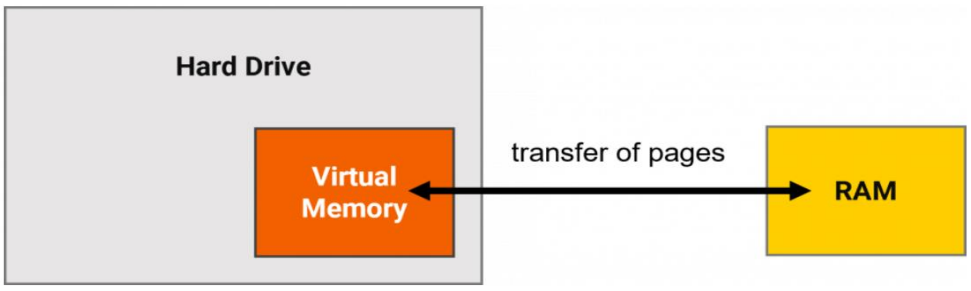
Paging	Segmentation
A page is a physical unit of information.	A segment is a logical unit of information.
Frames on main memory are required	No frames are required
The page is of the fixed block size	The page is of the variable block size
It leads to internal fragmentation	It leads to external fragmentation
The page size is decided by hardware in paging	Segment size is decided by the user in segmentation
It does not allow logical partitioning and protection of application components	It allows logical partitioning and protection of application components
Paging involves a page table that contains the base address of each page	Segmentation involves the segment table that contains the segment number and offset

Unit-IV

Topic 6 : Virtual Memory

Suppose you have a furniture shop that is not big enough to accommodate all the furniture, then you will keep it somewhere else, like in a warehouse. And you will keep that furniture in your shop which will be in demand. And if you don't require some particular furniture, you will keep that in the warehouse. And according to trend or demand, you will keep exchanging the furniture items from the warehouse. The same concept is for virtual memory.(warehouse is main memory and furniture's are process).

Virtual Memory is a storage scheme that provides user an illusion of having a very big main memory. This is done by treating a part of secondary memory as the main memory. In this scheme, User can load the bigger size processes than the available main memory by having the illusion that the memory is available to load the process.



Instead of loading one big process in the main memory, the Operating System loads the different parts of more than one process in the main memory. By doing this, the degree of multiprogramming will be increased and therefore, the CPU utilization will also be increased.

In practice, most real processes do not need all their pages, or at least not all at once, for several reasons:

- Error handling code is not needed unless that specific error occurs, some of which are quite rare.
- Arrays are often over-sized for worst-case scenarios, and only a small fraction of the arrays are actually used in practice.

The ability to load only the portions of processes that were actually needed (and only when they were needed) has several benefits:

- Programs could be written for a much larger address space (virtual memory space) than physically exists on the computer.
- Because each process is only using a fraction of their total address space, there is more memory left for other programs, improving CPU utilization and system throughput.
- Less I/O is needed for swapping processes in and out of RAM, speeding things up.

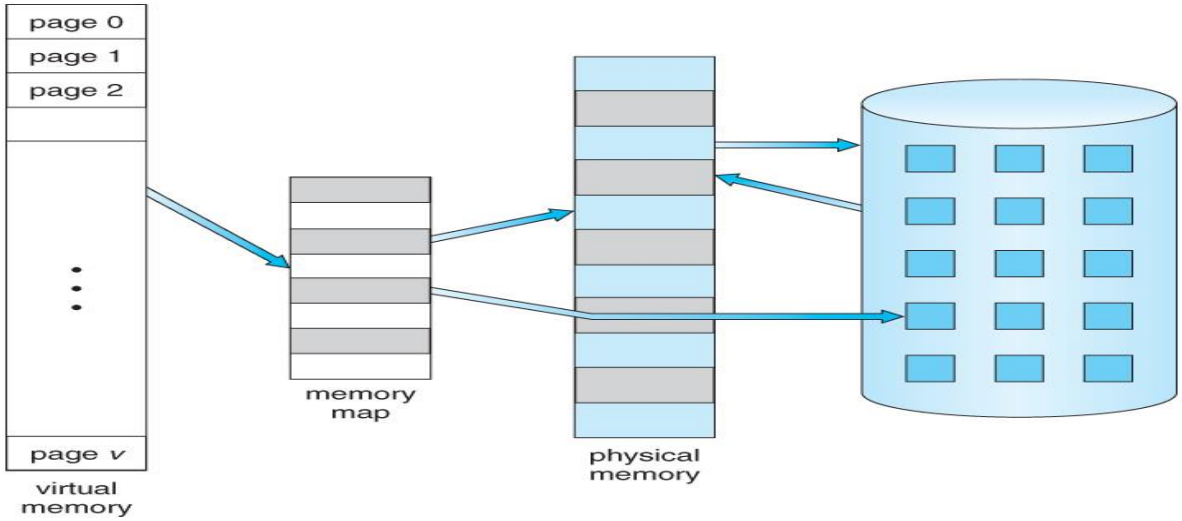
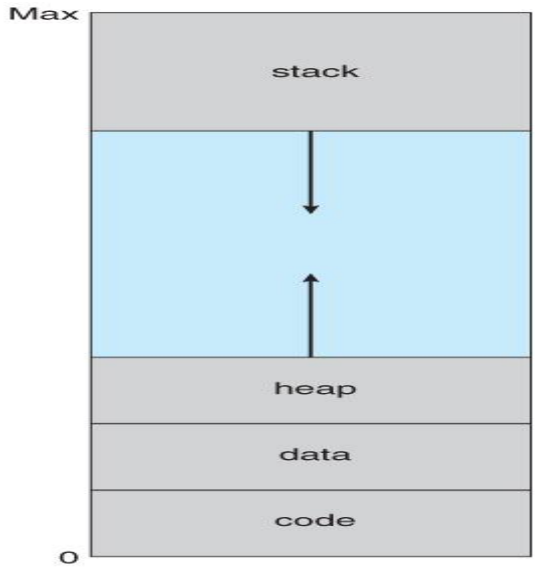


Diagram showing virtual memory that is larger than physical memory

- Figure below shows ***virtual address space***, which is the programmers logical view of process memory storage. The actual physical layout is controlled by the process's page table.



Note that the address space shown in Figure is ***sparse*** - A great hole in the middle of the address space is never used, unless the stack and/or the heap grow to fill the hole.

Virtual memory also allows the sharing of files and memory by multiple processes, with several benefits:

System libraries can be shared by mapping them into the virtual address space of more than one process.

Processes can also share virtual memory by mapping the same block of memory to more than one process.

Process pages can be shared during a fork( ) system call, eliminating the need to copy all of the pages of the original (parent) process.

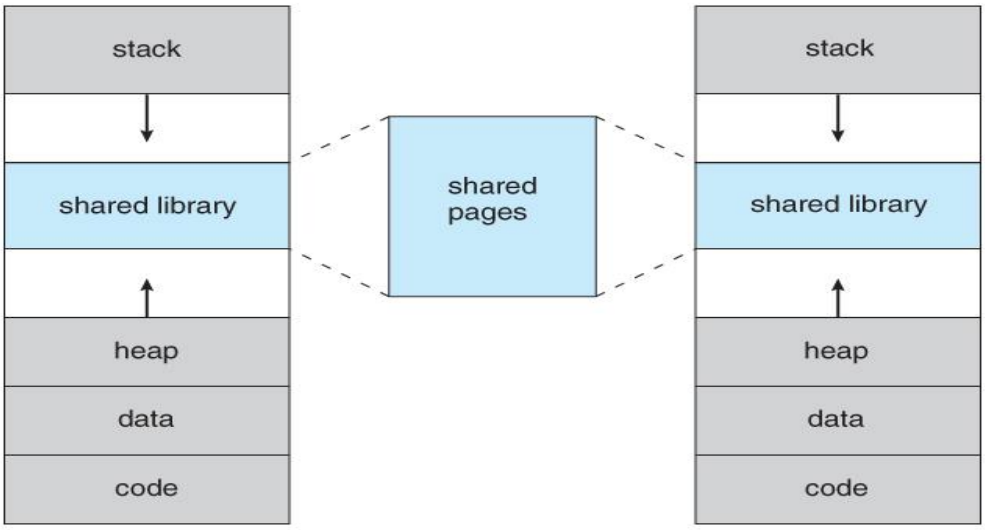


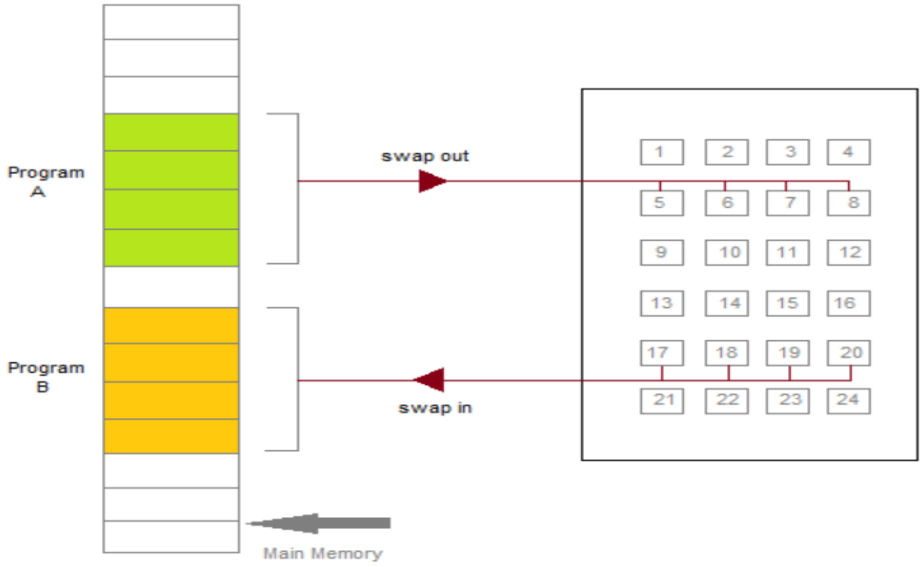
Figure : Shared library using virtual memory

Demand Paging

- According to the concept of Virtual Memory, in order to execute some process, only a part of the process needs to be present in the main memory which means that only a few pages will only be present in the main memory at any time.
- However, deciding, which pages need to be kept in the main memory and which need to be kept in the secondary memory, is going to be difficult because we cannot say in advance that a process will require a particular page at particular time.
- Therefore, to overcome this problem, there is a concept called Demand Paging is introduced. It suggests keeping all pages of the frames in the secondary memory until they are required. In other words, it says that do not load any page in the main memory until it is required. Whenever any page

is referred for the first time in the main memory, then that page will be found in the secondary memory.

- The demand paging system is somehow similar to the paging system with swapping where processes mainly reside in the main memory(usually in the hard disk). Thus demand paging is the process that solves the above problem only by swapping the pages on Demand.
- This is also known as **lazy swapper**(It never swaps the page into the memory unless it is needed).Swapper that deals with the individual pages of a process are referred to as **Pager.**



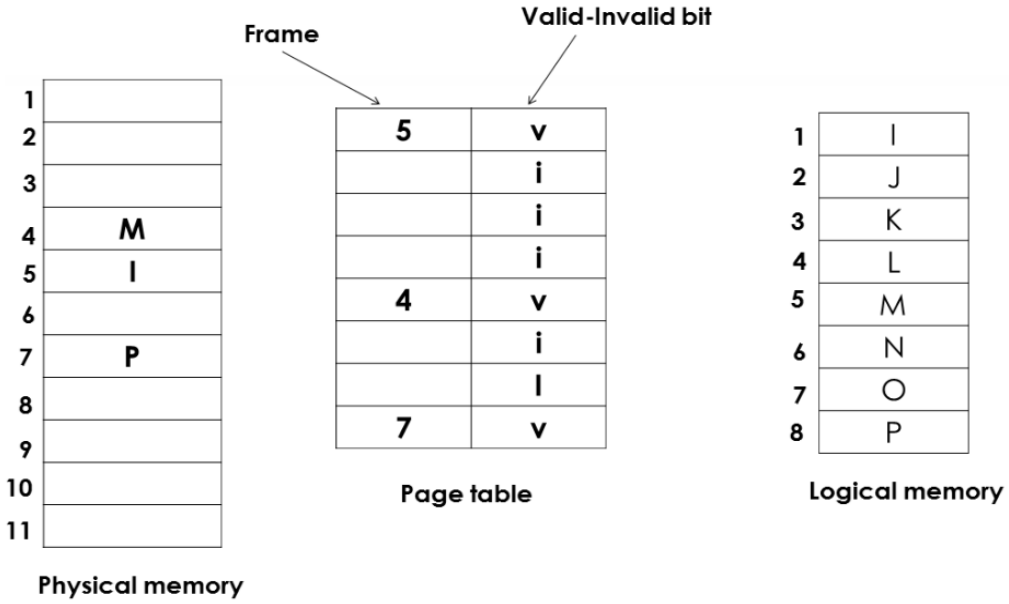
Some form of hardware support is used to distinguish between the pages that are in the memory and the pages that are on the disk. Thus for this purpose Valid-Invalid scheme is used:

- With each page table entry, a valid-invalid bit is associated(where **1** indicates **in the memory** and **0** indicates **not in the memory**)
- Initially, the valid-invalid bit is set to 0 for all table entries.

1. If the bit is set to "**valid**", then the associated page is **both legal and is in memory.**
2. If the bit is set to "**invalid**" then it indicates that the **page is either not valid** or the page is valid **but is currently not on the disk**.

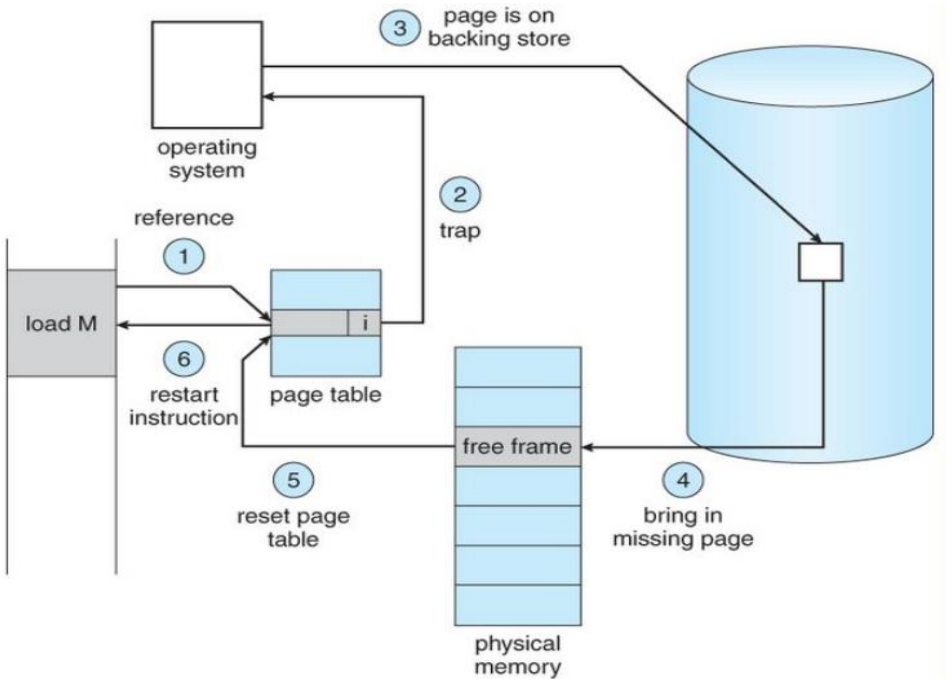
- For the **pages** that are **brought into the memory**, **the page table is set as usual.**
- But for the **pages** that are **not currently in the memory**, the **page table** is either **simply marked as invalid** or it contains the **address of the page on the disk**.

During the translation of address, if the valid-invalid bit in the page table entry is 0 then it leads to **page fault.**



The above figure is to indicates the page table when some pages are not in the main memory.

If the referred page is not present in the main memory then there will be a miss and the concept is called Page miss or page fault. Let us understand the procedure to handle the page fault as shown with the help of the below diagram:



1. First of all, internal table(that is usually the process control block) for this process in order to determine whether the reference was valid or invalid memory access.
2. If the reference is invalid, then we will terminate the process. If the reference is valid, but we have not bought in that page so now we just page it in.
3. Then we **locate the free frame lis**t in order to find the free frame.
4. Now a disk operation is scheduled in order to read the **desired page into the newly allocated fram**e.
5. When the disk is completely read, then the **internal table is modified** that is kept with the process, and the page table that mainly indicates the page is now in memory.
6. Now we will restart the instruction that was interrupted due to the trap. Now the process can access the page as though it had always been in memory.

In some cases when initially no pages are loaded into the memory, pages in such cases are only loaded when are demanded by the process by generating page faults. It is then referred to as Pure Demand Paging.

Performance of Demand Paging

To understand the impact of demand paging on system performance, we compute the effective access time (EAT) for memory. Let:

- *ma* be the memory access time (10 to 200 ns).
- *p* be the probability of a page fault ($0 \le p \le 1$).

When no page faults occur, EAT is simply *ma*. If a page fault occurs, additional time is required to handle it.

Let p be the probability of a page fault (0<= p <=1). We would expect p to be close to zero-that is, we would expect to have only a few page faults. Then EAT =$(1-p)\times ma + p\times$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 the operating system.

2.Save the user registers and process state.

3. Determine that the interrupt was a page fault.

4. Check that the page reference was legal and determine the location of the page on the disk Issue a read from the disk to a free frame:

a. Wait in a queue for this device until the read request is serviced.

b. Wait for the device seek and/ or latency time.

c. Begin the transfer of the page to a free frame.

5.While waiting, allocate the CPU to some other user (CPU scheduling, optional).

6.Receive an interrupt from the disk I/0 subsystem (I/0 completed).

7.Save the registers and process state for the other user (if step 6 is executed).

8.Determine that the interrupt was from the disk

9.Correct the page table and other tables to show that the desired page is now in memory.

10.Wait for the CPU to be allocated to this process again.

11. Restore the user registers, process state, and new page table, and then resume the interrupted instruction.

Not all of these steps are necessary in every case. In any case, we are faced with three major components of the page-fault service time

Service the page-fault interrupt.

Read in the page.

Restart the process.

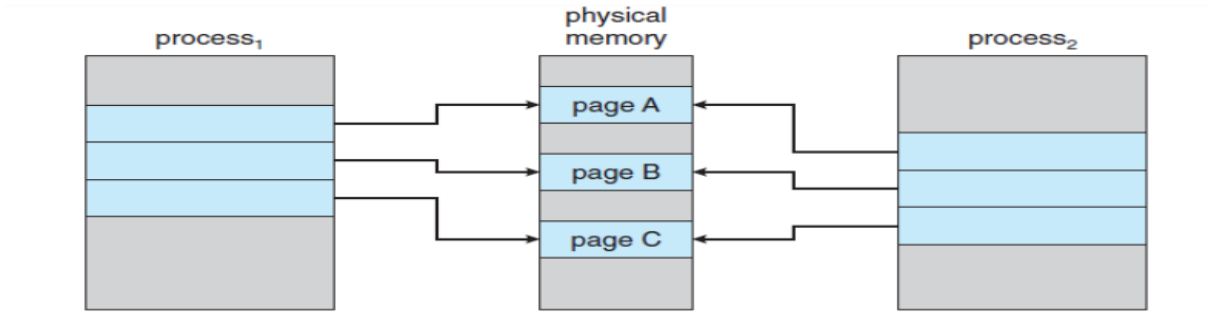
An additional aspect of demand paging is the handling and overall use of swap space.Swap Space Optimization can be done by

- Fast Swap Space: Allocated in large blocks, faster than file system.
- File System Paging: Initially demand pages from the file system; subsequently use swap space for better throughput.
- Hybrid Approach: Pages binary files from the file system, overwrites when replaced, and uses swap space for stack/heap. Used in Solaris and BSD UNIX.

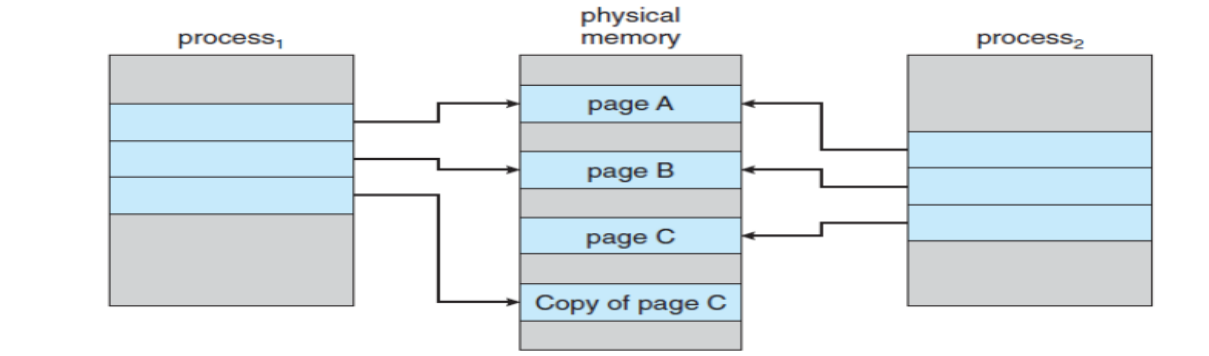
Copy-on-Write

Copy-on-Write Recall that the fork( ) system call creates a child process that is a duplicate of its parent. Traditionally, fork( ) worked by creating a copy of the parent's address space for the child, duplicating the pages belonging to the parent. However, considering that many child processes invoke the exec() system call immediately after creation, the copying of the parent's address space may be unnecessary. Instead, we can use a technique known as copy-on-write, which works by allowing the parent and child processes initially to share the same pages. These shared pages are marked as copy-on-write pages, meaning that if either process writes to a shared page, a copy of the shared page is created. Copy-on-

write is illustrated in Figures below, which show the contents of the physical memory before and after process 1 modifies page C



Before process 1 modifies page C.



After process 1 modifies page C.

When it is determined that a page is going to be duplicated using copy on-write, it is important to note the location from which the free page will be allocated. Many operating systems provide a pool of free pages for such requests. These free pages are typically allocated when the stack or heap for a process must expand or when there are copy-on-write pages to be managed. Operating systems typically allocate these pages using a technique known as Zero-fill-on-demand. Zero-fill-on-demand pages have been zeroed-out before being allocated, thus erasing the previous contents.

Unit-IV

Topic 7: Page Replacement Algorithms

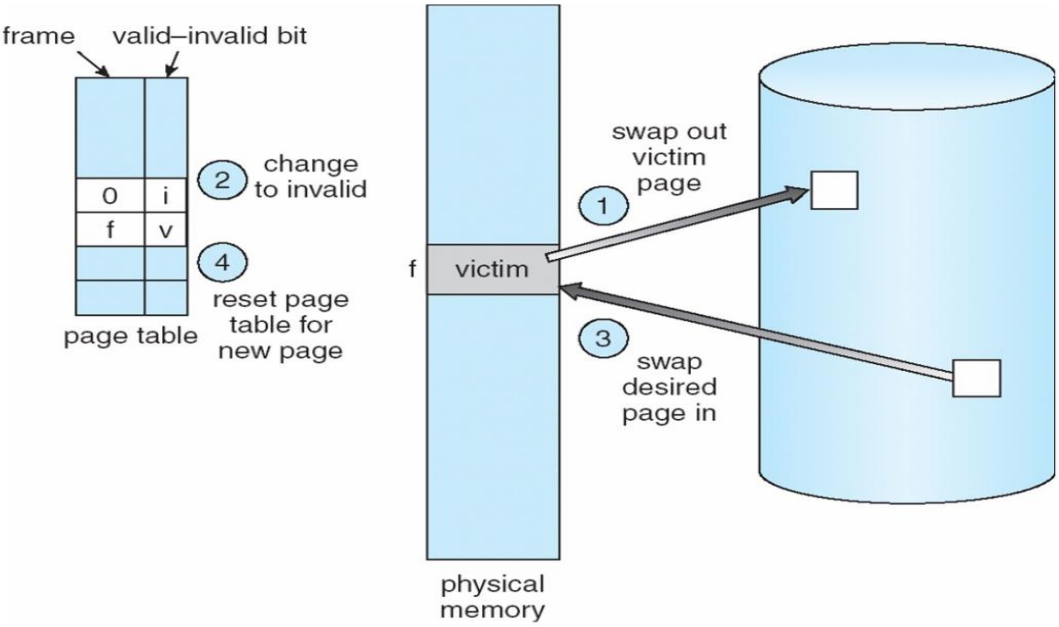
Need for Page Replacement

As studied in Demand Paging, only certain pages of a process are loaded initially into the memory. This allows us to get more processes into memory at the same time. but what happens when a process requests for more pages and no free memory is available to bring them in then the Operating system needs to decide which page to replace. The operating system must use any page replacement algorithm in order to select the victim frame.

Basic Page Replacement

If there is no free frame follow the steps as depicted in the diagram

1. Use a page replacement algorithm to select a victim frame. Write the victim page to the disk.
2. change the page & frame tables accordingly.
3. Read the desired page into the free frame.
4. Update the page and frame tables and restart the process



Notice that, if no frames are free, two page transfers (one out and one in) are required. This situation effectively doubles the page-fault service time and increases the effective access time accordingly. We can reduce this overhead by using a modify or a dirty bit. Each page or frame has a modify bit associated with it in the hardware. It indicates that any word or byte in the page is 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 & in this case we must write that page to the disk.
- If the bit is not set, then if the copy of the page on the disk has not been overwritten, then we can avoid writing the memory page on the disk as it is already there.

To implement demand paging, two problems should be solved:

  - Frame-allocation algorithm
  - Page replacement algorithm

When multiple processes are in main memory, the operating system should decide how many frames should be allocated to each process. Suppose there are 10 processes and there are 50 frames, a decision has to be made as to how many frames should be given to each process. This is called frame allocation.

Whenever page replacement is needed, the operating system should decide which page should be selected for replacement. This is done by a page replacement algorithm.

Page Replacement Algorithms

A page replacement algorithm determines how the victim page (the page to be replaced) is selected when a page fault occurs. Page replacement algorithms in an operating system require specific inputs to function correctly. These inputs allow the algorithm to make decisions about which page to replace when a page fault occurs. Here are the primary inputs needed:

Reference String: A sequence of memory addresses or page numbers that represent the order in which pages are accessed by a process.

Number of Frames: The number of available frames in the physical memory.

In the context of page replacement algorithms and memory management in operating systems, "page hit" and "page miss" (or page fault) are key concepts that describe the success or failure of accessing a page in memory.

Page Hit: A page hit occurs when the page requested by the process is already present in physical memory. No replacement is needed

Page Miss (Page Fault): A page miss, also known as a page fault, occurs when the page requested by the process is not present in physical memory. The operating system must load the requested page from secondary storage (like a hard disk or SSD) into physical memory.

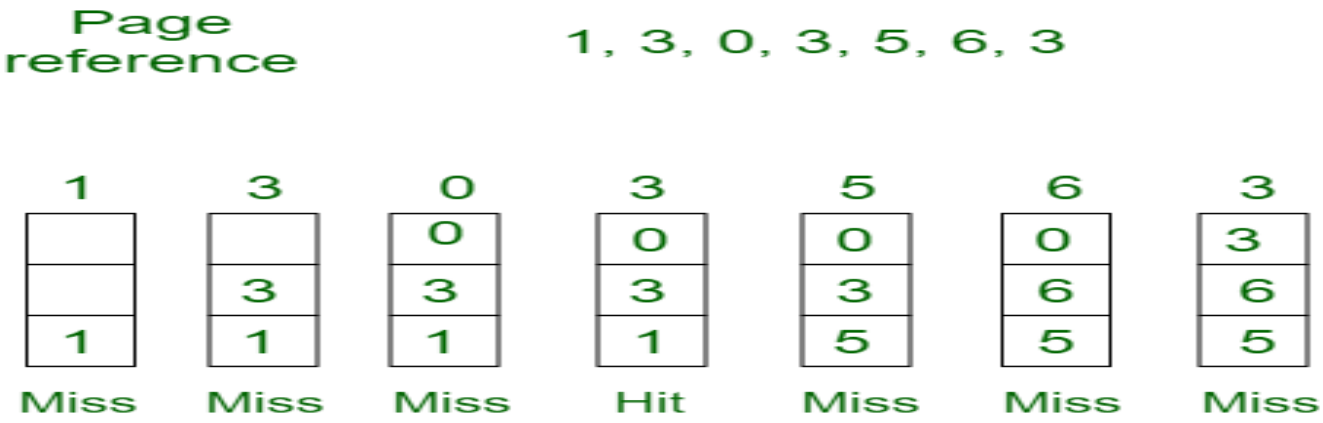
The aim of any replacement algorithm is to minimize the page fault rate.

Page Replacement Algorithms

1. FIFO Page Replacement
2. Optimal Page Replacement
3. LRU Page Replacement

1. First In First Out (FIFO): This is the simplest page replacement algorithm. In this algorithm, the operating system keeps track of all pages in the memory in a queue, the oldest page is in the front of the queue. When a page needs to be replaced page in the front of the queue is selected for removal.

Example 1: Consider page reference string 1, 3, 0, 3, 5, 6, 3 with 3-page frames. Find the number of page faults.



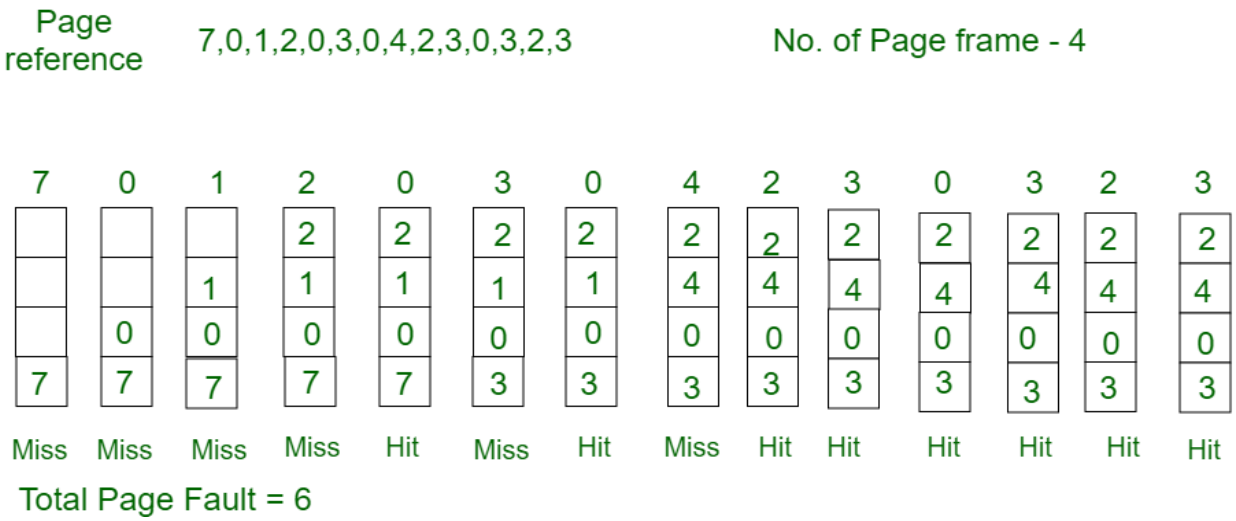
Initially, all slots are empty, so when 1, 3, 0 came they are allocated to the empty slots —> **3 Page Faults.**

when 3 comes, it is already in memory so —> **0 Page Faults.** Then 5 comes, it is not available in memory so it replaces the oldest page slot i.e 1. —>**1 Page Fault.** 6 comes, it is also not available in memory so it replaces the oldest page slot i.e 3 —>**1 Page Fault.** Finally, when 3 come it is not available so it replaces 0 **1 page fault.**

Belady's anomaly proves that it is possible to have more page faults when increasing the number of page frames while using the First in First Out (FIFO) page replacement algorithm. For example, if we consider reference strings 3, 2, 1, 0, 3, 2, 4, 3, 2, 1, 0, 4, and 3 slots, we get 9 total page faults, but if we increase slots to 4, we get 10-page faults.

2. Optimal Page replacement: In this algorithm, pages are replaced which would not be used for the longest duration of time in the future.

- **Example-2:** Consider the page references 7, 0, 1, 2, 0, 3, 0, 4, 2, 3, 0, 3, 2, 3 with 4 page frame. Find number of page fault.



Initially, all slots are empty, so when 7 0 1 2 are allocated to the empty slots —> **4 Page faults** 0 is already there so —> **0 Page fault.** when 3 came it will take the place of 7 because it is not used for the longest duration of time in the future.—>**1 Page fault.** 0 is already there so —> **0 Page fault.** 4 will takes place of 1 —> **1 Page Fault.**

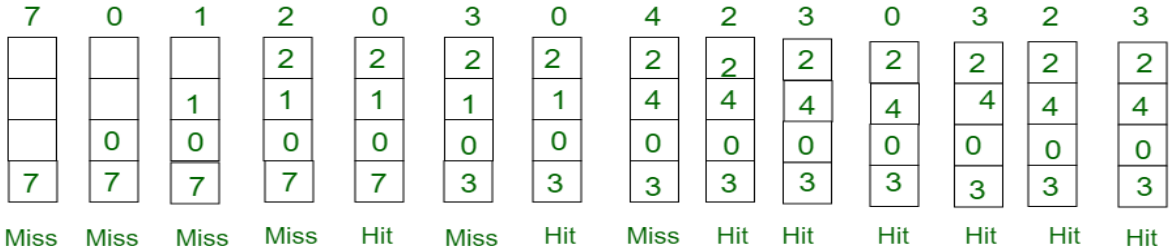
Now for the further page reference string —> **0 Page fault** because they are already available in the memory.

Optimal page replacement is perfect, but not possible in practice as the operating system cannot know future requests. The use of Optimal Page replacement is to set up a benchmark so that other replacement algorithms can be analyzed against it.

3. LRU: The LRU stands for the **Least Recently Used**. It keeps track of page usage in the memory over a short period of time. It works on the concept that pages that have been highly used in the past are likely to be significantly used again in the future. It removes the page that has not been utilized in the memory for the longest time. LRU is the most widely used algorithm because it provides fewer page faults than the other methods.

- **Example-3:** Consider the page reference string 7, 0, 1, 2, 0, 3, 0, 4, 2, 3, 0, 3, 2, 3 with 4 page frames. Find number of page faults.

Page reference 7,0,1,2,0,3,0,4,2,3,0,3,2,3 No. of Page frame - 4



Total Page Fault = 6

Here LRU has same number of page fault as optimal but it may differ according to question.

- Initially, all slots are empty, so when 7 0 1 2 are allocated to the empty slots —> **4 Page faults**

0 is already their so —> **0 Page fault.** when 3 came it will take the place of 7 because it is least recently used —>**1 Page fault**

0 is already in memory so —> **0 Page fault**.

4 will takes place of 1 —> **1 Page Fault**

Now for the further page reference string —> **0 Page fault** because they are already available in the memory.

Implementing the Least Recently Used (LRU) page replacement algorithm can be done using either a counter-based approach or a stack-based approach.

LRU Counter-Based Approach: In this method, each page table entry is associated with a counter that keeps track of the last time the page was accessed.

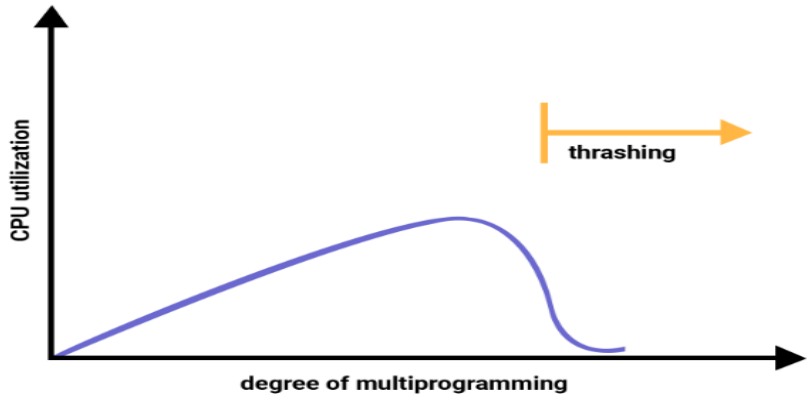
- Let a global counter to zero.
- Each page has a corresponding counter initialized to zero. Each time a page is accessed, increment the global counter and update the page's counter with the current value of the global counter.
- When a page fault occurs and a page needs to be replaced, choose the page with the smallest counter value (indicating it was used the longest time ago).

LRU Stack-Based Approach: In this method, a stack (or a doubly linked list) is used to maintain the order of page accesses. The most recently accessed page is moved to the top of the stack, and the least recently accessed page is at the bottom.

- Start with an empty stack.
- Each time a page is accessed, if it is already in the stack, move it to the top. If it is not in the stack and the stack is full, remove the bottom page and add the new page to the top.
- The page at the bottom of the stack is the least recently used and should be replaced when necessary

Thrashing

Thrashing in operating systems refers to a situation where the system spends a significant amount of time swapping pages in and out of memory, rather than executing application code. This typically occurs when the system is under heavy load and the working set of all processes cannot fit into the physical memory available.



Causes of Thrashing

1. **Insufficient Memory**: When the system's RAM is too small to handle the current load, it relies heavily on virtual memory, causing frequent page swaps.
2. **High Multiprogramming Level**: Running too many processes simultaneously can lead to an excessive demand for memory.
3. **Poor Locality of Reference**: If the processes do not exhibit good locality of reference, meaning they frequently access pages scattered throughout memory, it can increase the rate of page faults.
4. **Over-committing Memory**: Allocating more memory to processes than the physical memory available, relying on virtual memory to compensate.

Effect of Thrashing

At the time, when thrashing starts then the operating system tries to apply either the **Global page replacement** Algorithm or the **Local page replacement** algorithm.

Global Page Replacement

The Global Page replacement has access to bring any page, whenever thrashing found it tries to bring more pages. Actually, due to this, no process can get enough frames and as a result, the thrashing will increase more and more. Thus the global page replacement algorithm is not suitable whenever thrashing happens.

Local Page Replacement

Unlike the Global Page replacement, the local page replacement will select pages which only belongs to that process. Due to this, there is a chance of a reduction in the thrashing. As it is also proved that there are many disadvantages of Local Page replacement. Thus local page replacement is simply an alternative to Global Page replacement.

Techniques used to handle the thrashing

As we have already told you the Local Page replacement is better than the Global Page replacement but local page replacement has many disadvantages too, so it is not suggestible. Thus, given below are some other techniques that are used:

1.Working-Set Model

- ✓ This model is based on the above-stated concept of the Locality Model. The basic principle states that if we allocate enough frames to a process to accommodate its current locality, it will only fault whenever it moves to some new locality. But if the allocated frames are lesser than the size of the current locality, the process is bound to thrash. According to this model, based on parameter A, the working set is defined as the set of pages in the most recent 'A' page references. Hence, all the actively used pages would always end up being a part of the working set.

  The accuracy of the working set is dependent on the value of parameter A. If A is too large, then working sets may overlap. On the other hand, for smaller values of A, the locality might not be covered entirely.

  If D is the total demand for frames and WSS_i is the working set size for process i,

  $$D = \sum WSS_i$$

  Now, if 'm' is the number of frames available in the memory, there are 2 possibilities:

- (i) D>m i.e. total demand exceeds the number of frames, then thrashing will occur as some processes would not get enough frames.
- (ii) D<=m, then there would be no thrashing.

2. Page Fault Frequency

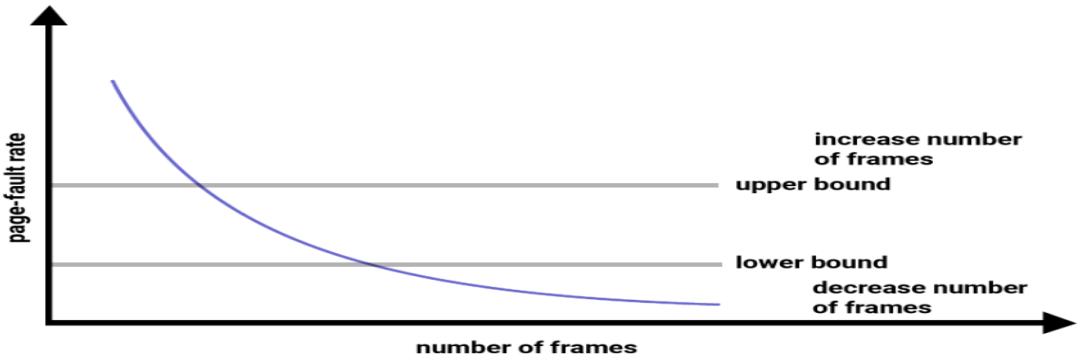


Figure: Page fault frequency

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

When the Page fault is too high, then we know that the process needs more frames. Conversely, if the page fault-rate is too low then the process may have too many frames.

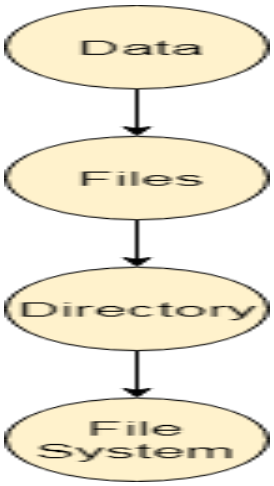
We can establish upper and lower bounds on the desired page faults. If the actual page-fault rate exceeds the upper limit then we will allocate the process to another frame. And if the page fault rate falls below the lower limit then we can remove the frame from the process.

Thus with this, we can directly measure and control the page fault rate in order to prevent thrashing.

Unit-IV

Topic 8: File System Interface and Operations

A file can be defined as a data structure which stores the sequence of records. Files are stored in a file system, which may exist on a disk or in the main memory. Files can be simple (plain text) or complex (specially-formatted).The collection of files is known as Directory. The collection of directories at the different levels, is known as File System.



The information in a file is defined by its creator. Many different types of information may be stored in a file-source programs, object programs, executable programs, numeric data, text, payroll records, graphic images, sound recordings, and so on. A file has a certain defined which depends on its type. A text file is a sequence of characters organized into lines (and possibly pages). A source file is a sequence of subroutines and functions, each of which is further organized as declarations followed by executable statements. An object file is a sequence of bytes organized in.to blocks understandable by the system's linker. An executable file is a series of code sections that the loader can bring into memory and execute.

Attributes of the File

1.Name

Every file carries a name by which the file is recognized in the file system. One directory cannot have two files with the same name.

2.Identifier

Along with the name, Each File has its own extension which identifies the type of the file. For example, a text file has the extension **.txt,** A video file can have the extension **.mp4.**

3.Type

In a File System, the Files are classified in different types such as video files, audio files, text files, executable files, etc.

file type	usual extension	function
executable	exe, com, bin or none	ready-to-run machine-language program
object	obj, o	compiled, machine language, not linked
source code	c, cc, java, perl, asm	source code in various languages
batch	bat, sh	commands to the command interpreter
markup	xml, html, tex	textual data, documents
word processor	xml, rtf, docx	various word-processor formats
library	lib, a, so, dll	libraries of routines for programmers
print or view	gif, pdf, jpg	ASCII or binary file in a format for printing or viewing
archive	rar, zip, tar	related files grouped into one file, sometimes compressed, for archiving or storage
multimedia	mpeg, mov, mp3, mp4, avi	binary file containing audio or A/V information

4.Location

In the File System, there are several locations on which, the files can be stored. Each file carries its location as its attribute.

5.Size

The Size of the File is one of its most important attribute. By size of the file, we mean the number of bytes acquired by the file in the memory.

6.Protection

The admin of the computer may want the different protections for the different files. Therefore, each file carries its own set of permissions to the different group of Users.

7.Time and Date

Every file carries a time stamp which contains the time and date on which the file is last modified

Operations on the File

A file is a collection of logically related data that is recorded on the secondary storage in the form of sequence of operations. The content of the files are defined by its creator who is creating the file. The various operations which can be implemented on a file such as read, write, open and close etc. are called

file operations. These operations are performed by the user by using the commands provided by the operating system. Some common operations are as follows:

1.Create operation:

This operation is used to create a file in the file system. It is the most widely used operation performed on the file system. To create a new file of a particular type the associated application program calls the file system. This file system allocates space to the file. As the file system knows the format of directory structure, so entry of this new file is made into the appropriate directory.

2. Open operation:

This operation is the common operation performed on the file. Once the file is created, it must be opened before performing the file processing operations. When the user wants to open a file, it provides a file name to open the particular file in the file system. It tells the operating system to invoke the open system call and passes the file name to the file system.

3. Write operation:

This operation is used to write the information into a file. A system call write is issued that specifies the name of the file and the length of the data has to be written to the file. Whenever the file length is increased by specified value and the file pointer is repositioned after the last byte written.

4. Read operation:

This operation reads the contents from a file. A Read pointer is maintained by the OS, pointing to the position up to which the data has been read.

5. Re-position or Seek operation:

The seek system call re-positions the file pointers from the current position to a specific place in the file i.e. forward or backward depending upon the user's requirement. This operation is generally performed with those file management systems that support direct access files.

6. Delete operation:

Deleting the file will not only delete all the data stored inside the file it is also used so that disk space occupied by it is freed. In order to delete the specified file the directory is searched. When the directory entry is located, all the associated file space and the directory entry is released.

7. Truncate operation:

Truncating is simply deleting the file except deleting attributes. The file is not completely deleted although the information stored inside the file gets replaced.

8. Close operation:

When the processing of the file is complete, it should be closed so that all the changes made permanent and all the resources occupied should be released. On closing it deallocates all the internal descriptors that were created when the file was opened.

9. Append operation:

This operation adds data to the end of the file.

10. Rename operation:

This operation is used to rename the existing file.

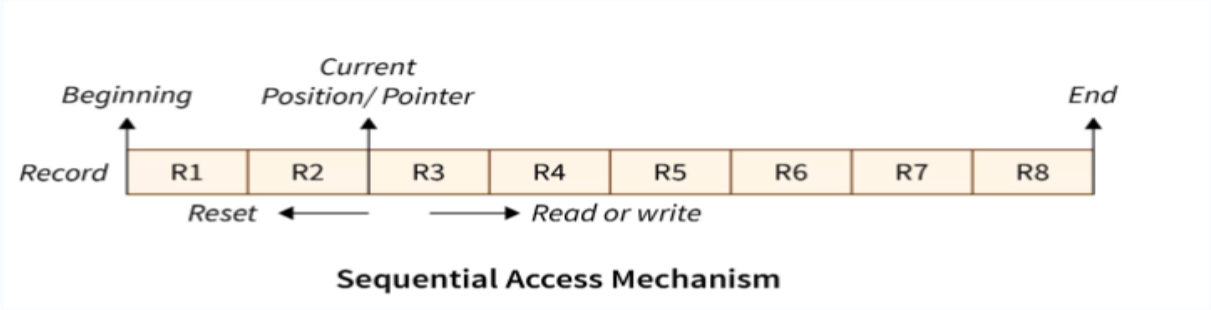
File Access Method

File access methods in OS refer to the techniques used to access read and write data from and to a file. There are various file access methods in os like:

- Sequential Access
- Direct Access
- Indexed Sequential Access

Sequential Access

The operating system reads the file word by word in a sequential access method of file accessing. A pointer is made, which first links to the file's base address. If the user wishes to read the first word of the file, the pointer gives it to them and raises its value to the next word. This procedure continues till the file is finished. It is the most basic way of file access. The data in the file is evaluated in the order that it appears in the file and that is why it is easy and simple to access a file's data using a sequential access mechanism. For example, editors and compilers frequently use this method to check the validity of the code.



Sequential Access Mechanism

Examples of devices that use sequential access – Sequential access is commonly used in devices such as tape drives, which require reading or writing data in a linear or sequential order.

Advantages of Sequential Access:

- The sequential access mechanism is very easy to implement.

- It uses lexicographic order to enable quick access to the next entry.

Disadvantages of Sequential Access:

- Sequential access will become slow if the next file record to be retrieved is not present next to the currently pointed record.
- Adding a new record may need relocating a significant number of records of the file.

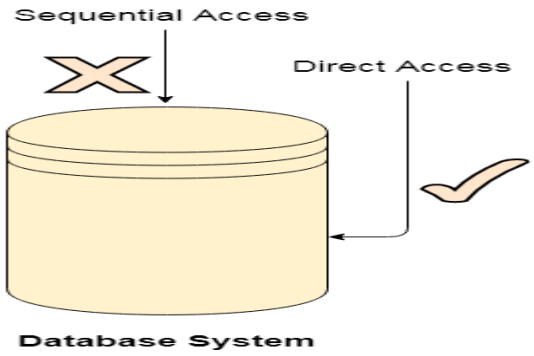
Direct Access

The Direct Access is mostly required in the case of database systems. In most of the cases, we need filtered information from the database. The sequential access can be very slow and inefficient in such cases.

Suppose every block of the storage stores 4 records and we know that the record we needed is stored in 10th block. In that case, the sequential access will not be implemented because it will traverse all the blocks in order to access the needed record.

In direct access, data is read or written directly to the physical location in the file. The data can be accessed by using the record number, byte position, or block number. This allows for fast and efficient access to specific data within the file.

For the direct-access method, the file operations must be modified to include the block number as a parameter. Thus, we have read n, where n is the block number, rather than read next, and ·write n rather than write next. An alternative approach is to retain read next and write next, as with sequential



Examples of devices that use direct access – Direct access is commonly used in devices such as magnetic disk drives, optical disk drives, and flash memory. These devices require fast and efficient access to specific data, which makes direct access an ideal file access method. Direct access is also commonly used in database systems, where fast access to specific records is required.

Advantages of Direct/Relative Access:

- The files can be retrieved right away with a direct access mechanism, reducing the average access time of a file.
- There is no need to traverse all of the blocks that come before the required block to access the record.

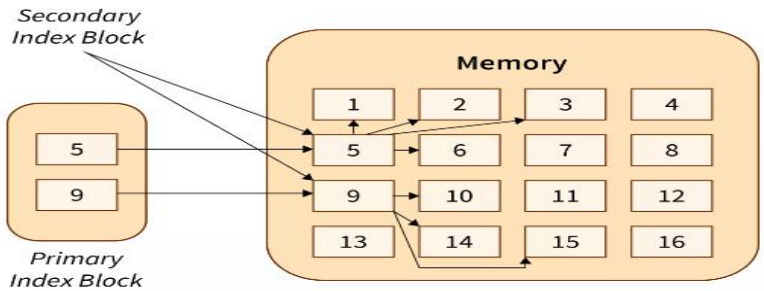
Disadvantages of Direct/Relative Access:

- The direct access mechanism is typically difficult to implement due to its complexity.
- Organizations can face security issues as a result of direct access as the users may access/modify the sensitive information. As a result, additional security processes must be put in place.

Indexed Sequential Access

It's the other approach to accessing a file that's constructed on top of the sequential access mechanism. This method is practically similar to the pointer-to-pointer concept in which we store the address of a pointer variable containing the address of some other variable/record in another pointer variable. The indexes, similar to a book's index (pointers), contain a link to various blocks present in the memory. To locate a record in the file, we first search the indexes and then use the pointer-to-pointer concept to navigate to the required file.

Primary index blocks contain the links of the secondary inner blocks which contain links to the data in the memory.



Indexed Sequential Access Of File

Advantages of Indexed Sequential Access:

- If the index table is appropriately arranged, it accesses the records very quickly.
- Records can be added at any position in the file quickly.

Disadvantages of Indexed Sequential Access:

- When compared to other file access methods, it is costly and less efficient.
- It needs additional storage space.

Unit-IV

Topic 9: DIRECTORY AND DISK STRUCTURE

Systems stores millions of files on disk. Therefore, it is necessary to organize the disk to manage this large number of files. Directory can be defined as the listing of the related files on the disk. The directory may store some or the entire file attributes. To get the benefit of different file systems on the different operating systems, A hard disk can be divided into the number of partitions of different sizes. The partitions are also called volumes or mini disks.

Each partition must have at least one directory in which, all the files of the partition can be listed. A directory entry is maintained for each file in the directory which stores all the information related to that file.

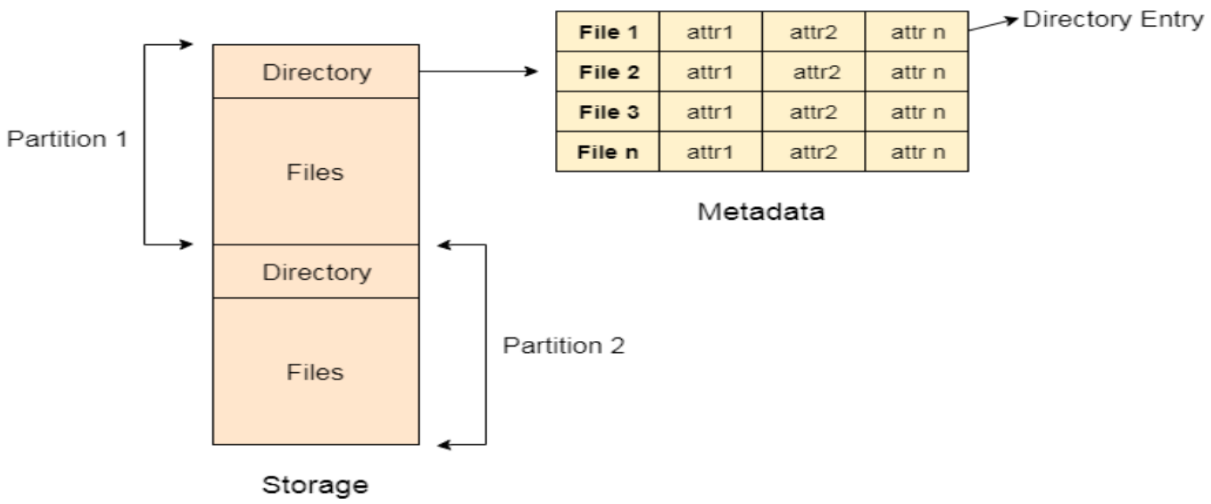


Fig: A typical file-system organization

Storage Structure

- A general-purpose computer system has multiple storage devices, and those devices can be sliced up into volumes that hold file systems.
- Computer systems may have zero or more file systems, and the file systems may be of varying types.
- For example, a typical Solaris system may have dozens of file systems of a dozen different types

Consider the types of file systems in the Solaris

- **tmpfs**—a "temporary" file system that is created in volatile main memory and has its contents erased if the system reboots or crashes
- **lofs**—a "loop back" file system that allows one file system to be accessed in place of another one
- **procfs**—a virtual file system that presents information on all processes as a file system
- **ufs, zfs**—general-purpose file systems

Operations Performed on Directory

Similar to how operations can be performed on files, there are operations that can be performed on directories. Some of the operations that can be performed on directories are given below:

- Search for a file – A directory has information about all the files present in the directory. To search for a file, it is necessary to search the directory.
- Create a file – To create a file, an entry is created in the directory. For this, it is necessary to write into the directory.
- Delete a file – To delete a file, it is necessary to remove the name of the file and all other details about the file from the directory. This again needs write permissions in the directory.
- List a directory – For listing the contents of a directory, it is necessary to read from the directory.
- Rename a file – To change the name of the file, it is necessary to read, write and search in the directory. Note that renaming a file may change the position of the file name in the directory.
- Traverse the file system – This also needs reading and searching operations on the directory.

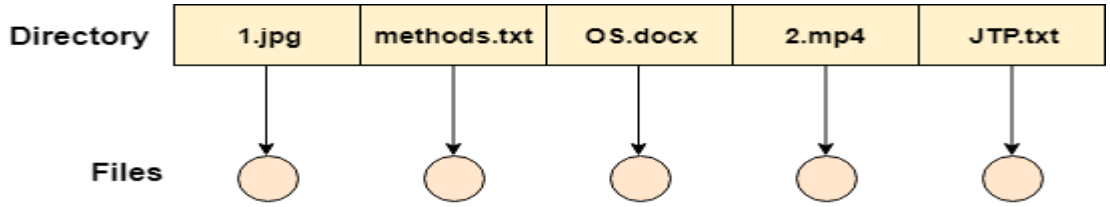
The Logical Directory Structures in OS

We have mainly five different types of directory structures in OS

- Single level directory
- Two-level directory
- Tree structure or hierarchical directory
- Acyclic graph directory
- General graph directory structure

Single Level Directory

The simplest method is to have one big list of all the files on the disk. The entire system will contain only one directory which is supposed to mention all the files present in the file system. The directory contains one entry per each file present on the file system.



Single Level Directory

This type of directories can be used for a simple system.

Advantages

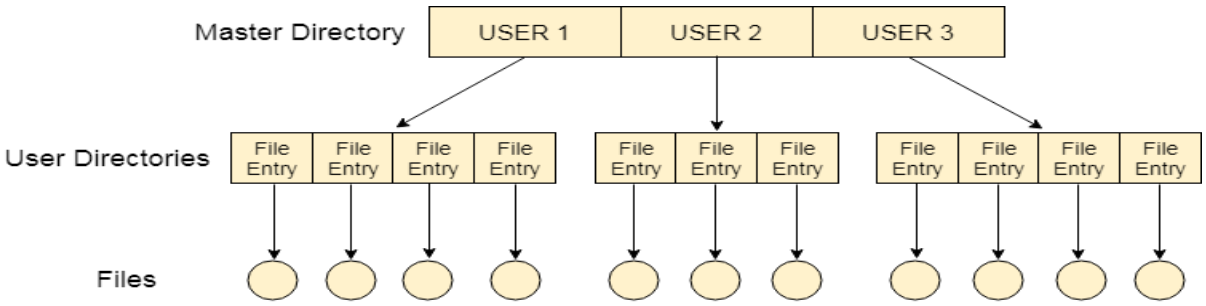
1. Implementation is very simple.
2. If the sizes of the files are very small then the searching becomes faster.
3. File creation, searching, deletion is very simple since we have only one directory.

Disadvantages

1. We cannot have two files with the same name.
2. The directory may be very big therefore searching for a file may take so much time.
3. Protection cannot be implemented for multiple users.
4. There are no ways to group same kind of files.
5. Choosing the unique name for every file is a bit complex and limits the number of files in the system because most of the Operating System limits the number of characters used to construct the file name.

Two Level Directory

In two level directory systems, we can create a separate directory for each user. There is one master directory which contains separate directories dedicated to each user. For each user, there is a different directory present at the second level, containing group of user's file. The system doesn't let a user to enter in the other user's directory without permission.



Two Level Directory

Path Name

- In two-level directory, this tree structure has (Master File Directory)MFD as root of path through(User file Directory) UFD touser file name at leaf.
- Path name :: username + filename
- Standard syntax -- /user/file.ext

Advantages

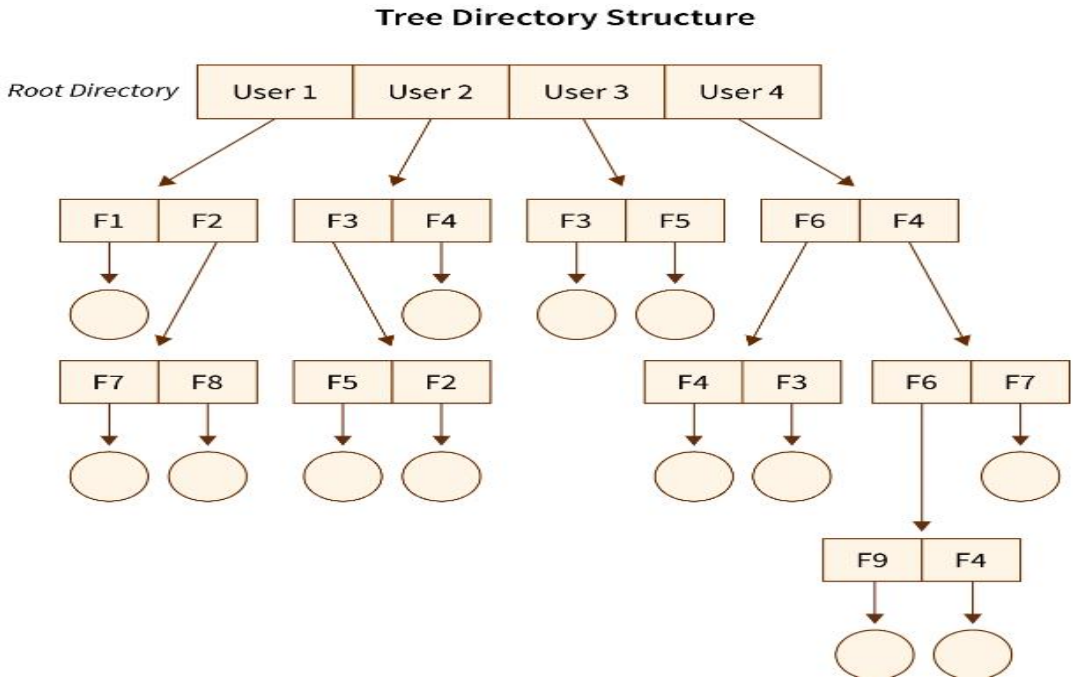
- Searching is very easy.
- There can be two files with the same name in two different user directories. Since they are not in the same directory, the same name can be used.
- Grouping is easier.
- A user cannot enter another user's directory without permission.
- Implementation is easy.

Disadvantages

- One user cannot share a file with another user.
- Even though it allows multiple users, still a user cannot keep two same type files in a user directory.
- It does not allow users to create subdirectories.

Tree-structured directory

The two-level directory was extended to have multiple levels and the tree-structured directory was formed. The tree is the most common directory structure. **Figure below shows an example of a tree-structured directory.**



In this structure, directory itself is a file. A directory and sub directory contains a set of files. Internal format is same for all directories.Commonly used directory structure is tree structure. Tree has a root directory. All files in disk have a unique path name.

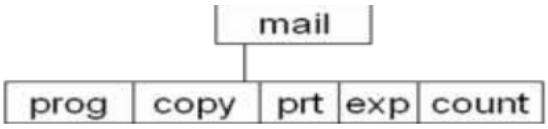
• A path name describes the path that the operating system must take to get from some starting point in the file system to the destination object. Each process has its own working directory. Path names are of two types : relative and absolute.

• Relative path name : Its path starts from current working directory (may start with

• Absolute path name: Its path starts from root directory (starts from "/").

• Tree structured directory differentiate file and subdirectory by using one bit representation. Zero (0) represents file and one (1) represents subdirectory.

- Whenever a file is newly created, it is created in the current directory. A new subdirectory is also created in the current directory. The command used for creating a subdirectory is **mkdir** <dir-name> where dir-name is the name of the subdirectory that is created. For example, if in current directory **/mail**, a subdirectory called *count* is to be created, then the command **mkdir** *count* is used. ***Figure below shows that a subdirectory count is created beneath the directory mail.***



• When process makes reference to a file for opening, file system search this file in the current directory. If needed file is not found in the current directory, then user either change the directory or give the full path of the file. System calls are used for performing any operation on the files or directory.

• Empty directory and its entry are easily deleted by operating system. If directory contains files and subdirectory, i.e. non-empty directory, some operating systems not delete the directory.

• To delete a directory, user must first delete all the files and subdirectory in that directory.

Advantages:

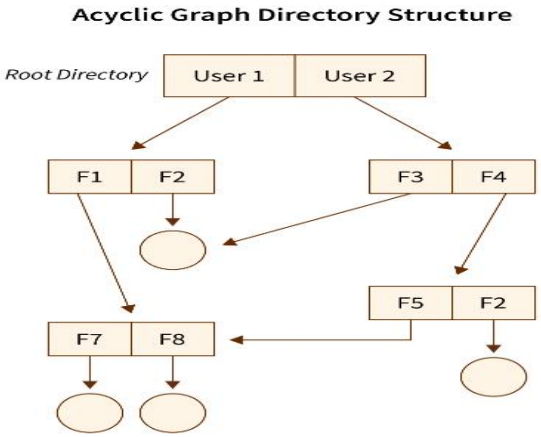
- This directory structure allows subdirectories inside a directory.
- The searching is easier.
- File sorting of important and unimportant becomes easier.
- This directory is more scalable than the other two directory structures explained.

Disadvantages:

- As the user isn't allowed to access other user's directory, this prevents the file sharing among users.
- As the user has the capability to make subdirectories, if the number of subdirectories increase the searching may become complicated.
- Users cannot modify the root directory data.

The Acyclic Graph Directory Structure

Overcoming the drawbacks posed by the tree-structured directory structure,i.e, the restriction that it cannot have multiple parent directories and also cannot share files between users - The Acyclic Graph directory structure in OS came into the picture.Mostly, this is used in situations such as, when two users or two programmers are collaborating on a project and they need to access the files.



- It is very interesting to note that a shared directory or file is not the same as two copies of the file. When there are two copies of files, each user can view the copy rather than the original, but if one user changes the file content, the changes will not appear in the other's copy.
- Only one original file exists for shared copy. Any changes made by one user are immediately visible to the other user. When user create file in shared directory, it automatically appear in all the shared subdirectories.

Implementation of shared files and directory is done through links

a. **The Hard Link:** This is also called as the physical link. If we want to delete the files in the acyclic graph directory structures then we need to remove the actual files only if all the references to the files are deleted that is, no link that even references the main file should be established. Here we don't leave a suspended link.

b. **The Symbolic/Soft-Link:** This is also called as the logical link. If we want to delete the files in the acyclic graph directory structures then we need to simply delete the files and need to keep in mind that only a dangling/ hanging point is left. Here we leave a suspended link.

Advantages:

- In the **Acyclic Graph** directory structure in OS we can share files between users.
- Here we can search the files easily as compared to the tree-structured directory structure as here we have different-different paths to one file.

Disadvantages:

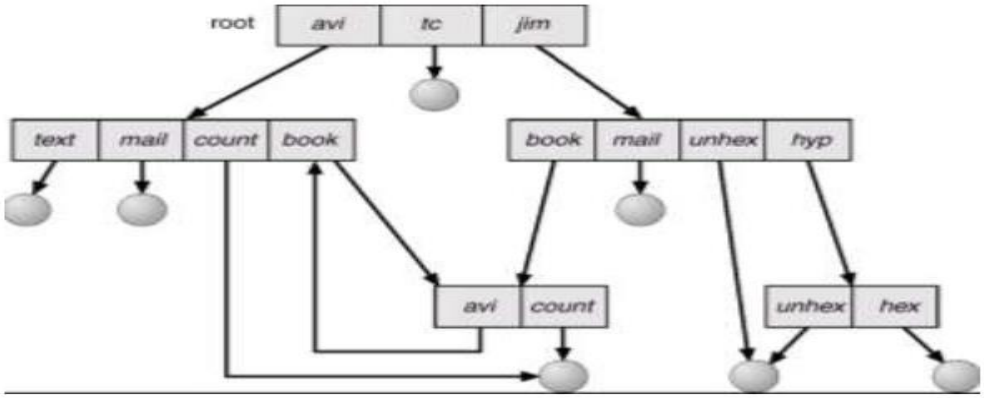
- In the Acyclic Graph directory structure in OS as we can share the files via linking, so there are chances that in the case when we want to delete a file in a directory it may create a problem.

The General Graph Directory Structure

As observed in the acyclic-graph directory structure in OS the drawback of links that are established and need to be either terminated or suspended to reach the files/directory is resolved with The General Graph directory structure in OS taking a vital place in the different types of a directory structure in OS.

In this type of structure, the users are free to create different sub-directories for different file types.Adding to the above, With the help of the file paths if the users feel the need to access the location of the files then that is made possible by the General Graph Directory Structure.

Below is the pictorial representation of The General Graph *directory structure* ***in OS :***



But if cycles are allowed to exist in directories, there is a possibility of searching a subdirectory twice, that is, it is possible to go from a parent directory to a subdirectory and from the subdirectory to the parent directory and so on. This may result in an infinite loop. To avoid this situation, one way is to limit the number of directories accessed during a search .The second way is to allow only links to file and not to subdirectories.

Let us now see how deletion of a file is handled in a general graph directory. Since cycles are allowed, if there is a self-referencing directory, there is a possibility to have a non-zero reference count even after deletion. To overcome this, garbage collection is to be done. During garbage collection, the file system is traversed and everything that can be accessed is marked in the first pass. In second pass, everything that is not marked is collected onto a list of free space. But this method is extremely time-consuming for a disk-based file system.

Advantages:

- The General Graph directory structure allows the cycle or creation of a directory within a directory.
- This directory structure is known to be a flexible version compared to other directory structures.

Disadvantages:

- As this directory structure allows the creation of multiple sub-directories a lot of garbage collection can be required.
- If compared to the other directory structure in OS the General Graph directory structure is a costly structure to be chosen.

Unit-IV

Topic 10 : Protection and File System Structure

It is necessary to keep files safe from physical damage (reliability) and from improper access (protection). For keeping the files reliable, it is necessary to have duplicate copies of files. We can also periodically copy disk files to tape at regular intervals. The owner/creator of the file should be able to control what can be done on a file and by whom.

Types of Access :

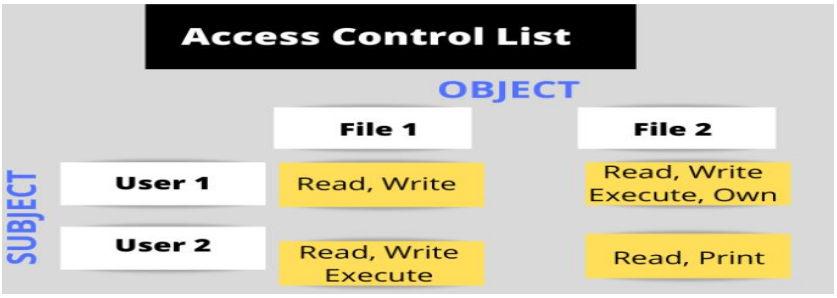
The files which have direct access of the any user have the need of protection. The files which are not accessible to other users doesn't require any kind of protection. The mechanism of the protection provide the facility of the controlled access by just limiting the types of access to the file. Access can be given or not given to any user depends on several factors, one of which is the type of access required. Several different types of operations can be controlled:

- **Read –** Reading from a file.
- **Write –** Writing or rewriting the file.
- **Execute –** Loading the file and after loading the execution process starts.
- **Append –** Writing the new information to the already existing file, editing must be end at the end of the existing file.
- **Delete –** Deleting the file which is of no use and using its space for the another data.
- **List –** List the name and attributes of the file.

One way in which access can be controlled is to have access control lists and groups.

Access Control Lists and Groups

With each file and directory an access-control list (ACL) is attached. The ACL has the names of users and the types of access allowed for each user. When a user requests access to a particular file, the access list is checked. If the user is listed for that particular access, access is allowed. Else, user is denied access.



This technique has two undesirable consequences:

- Constructing such a list may be a tedious and unrewarding task, especially if we do not know in advance the list of users in the system.
- The directory entry, previously of fixed size, now must be of variable size, resulting in more complicated space management.

These problems can be resolved by use of a condensed version of the access list. To condense the length of the access-control list, many systems recognize three classifications of users in connection with each file:

Owner: The user who created the file is the owner.

Group: A set of users who are sharing the file and need similar access is a group, or work group.

Universe/others: All other users in the system constitute the universe.

In UNIX, directory and file protection are handled similarly. Each file has three fields – owner, group and others. For each field there are three bits, r, w and x, corresponding to read, write and execute permissions. Suppose, the owner is provided read, write and execute permissions, r, w and x are set to 1 respectively. The octal number corresponding to the permissions allowed is 7 (111 in binary). Suppose the group is provided permissions for read and write, but no permissions for execution, the octal number for permissions is 6 (110 in binary). If others (universe, public) are provided only execute permissions, the octal number for permissions is 1 (001 in binary).

r w x

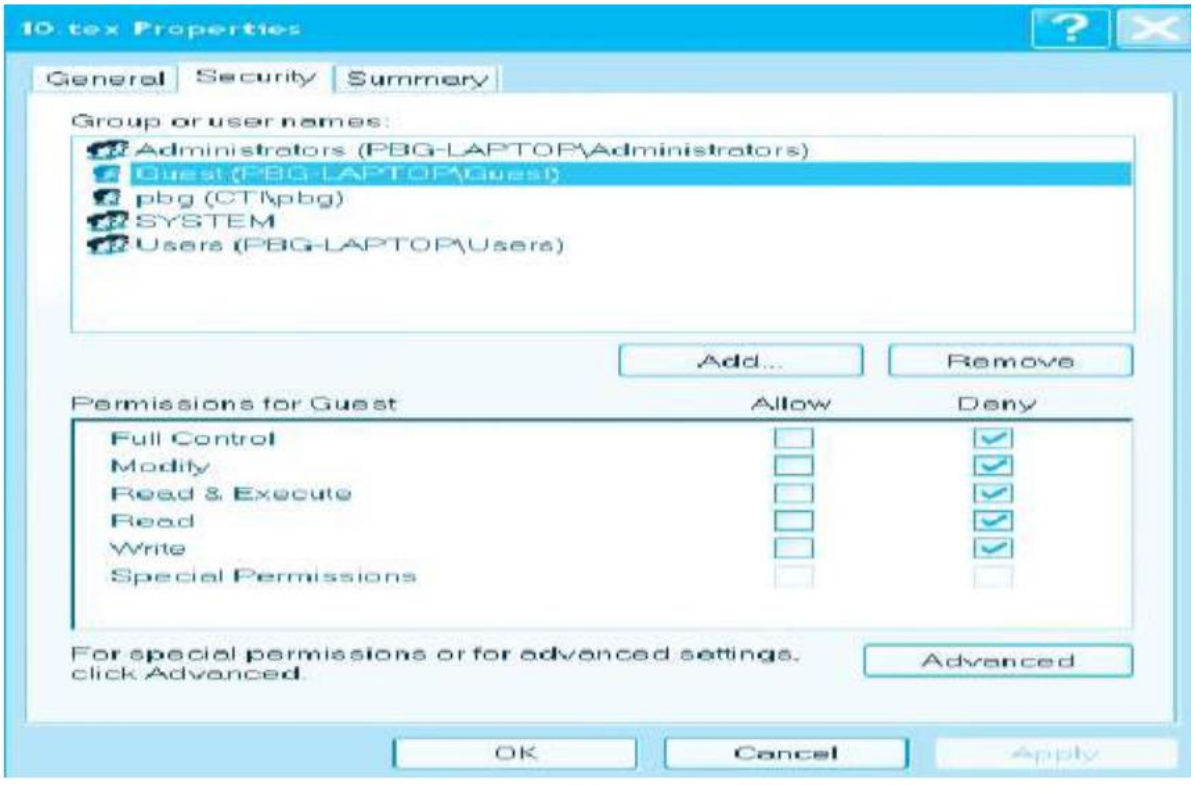
a) owner access	7	⇒	1 1 1
b) group access	6	⇒	1 1 0
c) public access	1	⇒	0 0 1

Thus, the access word for the file is 761 (rwx).

-rw-rw-r--	1 pbg	staff	31200	Sep 3 08:30	intro.ps
drwx------	5 pbg	staff	512	Jul 8 09.33	private/
drwxrwxr-x	2 pbg	staff	512	Jul 8 09:35	doc/
drwxrwx---	2 pbg	student	512	Aug 3 14:13	student-proj/
-rw-r--r--	1 pbg	staff	9423	Feb 24 2003	program.c
-rwxr-xr-x	1 pbg	staff	20471	Feb 24 2003	program
drwx--x--x	4 pbg	faculty	512	Jul 31 10:31	lib/
drwx------	3 pbg	staff	1024	Aug 29 06:52	mail/
drwxrwxrwx	3 pbg	staff	512	Jul 8 09:35	test/

Figure above shows a sample UNIX directory listing. The first file in the list is intro.ps. This file has read and write permissions for owner, read and write permissions for group and read permissions for others. The second row corresponds to a subdirectory called private in the listed directory. The letter 'd' before the access permissions indicates that 'private' is a directory. This directory 'private' has read, write and execute permissions for owner, but no permissions for group and others.

Figure below shows a example of access control provided for guest user in Windows XP. The guest user is denied all permissions.



Other Approaches

Another approach to the protection problem is to associate a password with each file. Just as access to the computer system is often controlled by a password, access to each file can be controlled in the

same way. If the passwords are chosen randomly and changed often, this scheme may be effective in limiting access to a file.

The use of passwords has a few disadvantages, however.

First, the number of passwords that a user needs to remember may become large, making the scheme impractical.

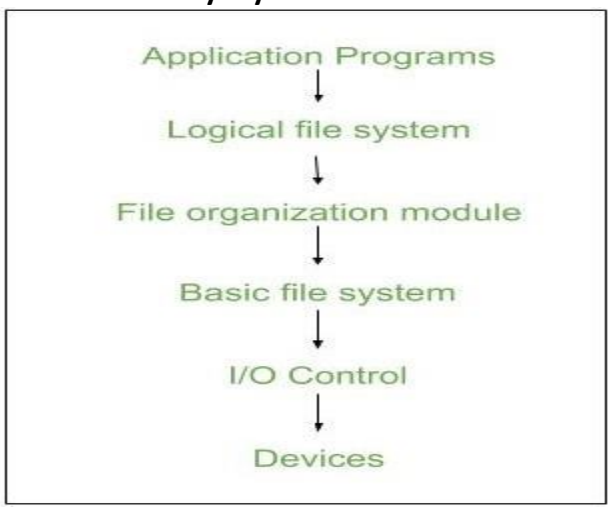
Second, if only one password is used for all the files, then once it is discovered, all files are accessible; protection is on an all-or-none basis.

Some systems allow users to associate password with a subdirectory. In this case, it is not necessary to assign a password for each and every file within this directory. It is enough to assign a single password for the entire directory.

File System Struture

The **file system structure** defines how information in key directories and files (existing and future) are organized and how they are stored in an operating system.

The file system is organized into many layers:



I/O control level

- This layer is placed just above the I/O devices. This level comprises of the device drivers and the interrupt handlers. The device drivers act as an interface between the devices and the operating system. They help to transfer information between the main memory and the disk.
- An example of input given to a device driver is – 'retrieve block 123'. In response to this, the device driver has to send low-level hardware specific instructions to the disk controller. The disk controller assists in reading block 123 from the disk.

Basic file system

- This layer issues generic commands to the device driver to read and write physical blocks on the disk. The input received from the I/O control layer is sent from this layer. Memory buffers and caches are maintained by the operating system in the main memory.
- The basic file system layer is responsible for managing the memory buffers and caches. Each memory buffer can hold contents equal to the size of a block. Each buffer holds the contents of a disk block. The contents that are read from the disk are copied to the buffers and can even be used later.
- The cache holds frequently used file-system metadata. This can be the contents of the file or attributes of the file such as the owner of the file, size of the file and so on. If the file is used frequently, the metadata can be used from the cache itself. It is not necessary to read from the disk each time.

File-organization module

- This layer uses the functionalities of the basic file system layer. This layer knows about files and their logical and physical blocks. The logical blocks are with respect to a particular file. The logical blocks are numbered from 0 to N for a particular file.
- Physical blocks do not match the logical numbers. Logical block *i* need not be kept in block number *i* in the physical memory. It can be kept in any physical disk block. Therefore, it is necessary to know the location of the location of the file in the disk (That is the locations of the disk blocks where the contents of the file are kept in the disk).
- Therefore, appropriate data structures are maintained by the file-organization module to know the mapping between the logical block number and the physical bock number. The file-organization module also has a free-space manager, which tracks unallocated disk blocks.

Logical file system

- This layer lies above the file-organization module. This layer manages the metadata information of a file. The metadata information includes all details about a file except the actual contents of the file, for example, the name of the file, the size of the file and so on.
- This layer also manages the directory structure. It maintains the file-structure via file-control blocks. A File-control block (FCB) (inode in UNIX) has information about a file – owner, size, permissions, time of access, location of file contents and so on.

Advantages of layered file system

- Duplication of code is minimized. The code for I/O control and basic file-system layers can be used by multiple file systems.
- The layers above these two layers can be modified for different file systems. That is, each file system can then have its own logical file system and file-organization modules, while the I/O control and basic file-system layers being the same.

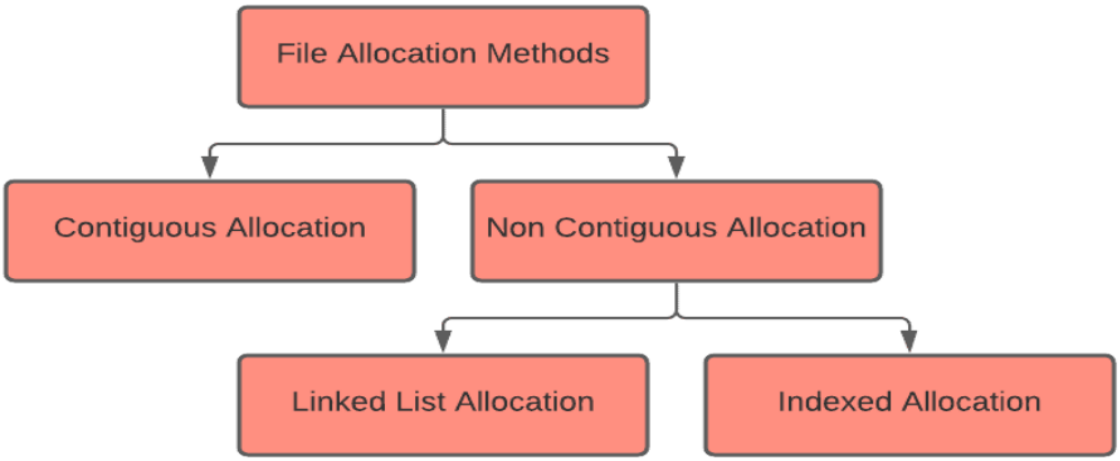
Disadvantages

- Having a layered file system can introduce more operating-system overhead, resulting in decreased performance. The decision about how many layers to use, what each layer should do is a challenge.
- Many file-systems are in use today – UNIX file system, FAT, FAT32, NTFS, ext3, ext4, Google. The FAT, FAT32 and NTFS are used in Windows operating systems. Ext3 and ext4 are used in Linux operating systems. Google has its own distributed file system called the Google File System (GFS).

Unit-IV

Topic 11(File Allocation Methods)

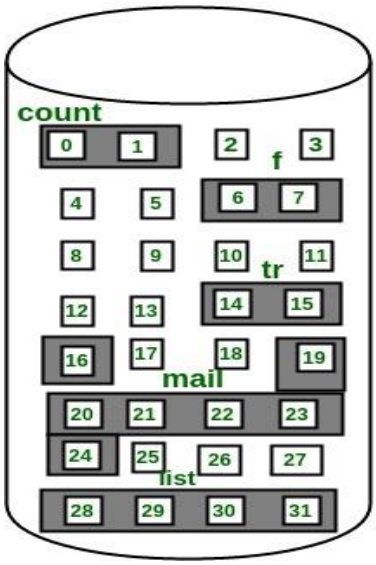
- Generally, files are stored in secondary storage devices such as hard disks. So to manage the space of hard disks efficiently and store the files fastly the Operating_System uses file allocation methods.
- The file system divides the file logically into multiple blocks. These blocks now need to be stored in a secondary storage device such as a hard disk. The hard disk contains multiple sectors or physical blocks in which the data actually reside. Now the file system will decide where to store these blocks of a file in the sectors of the hard disk using file allocation methods.



Contiguous Allocation

- In this scheme, each file occupies a contiguous set of blocks on the disk. For example, if a file requires n blocks and is given a block b as the starting location, then the blocks assigned to the file will be: *b, b+1, b+2,……b+n-1.*
- This means that given the starting block address and the length of the file (in terms of blocks required), we can determine the blocks occupied by the file. The directory entry for a file with contiguous allocation contains
    - Address of starting block
    - Length of the allocated portion.

The *file 'mail'* in the following figure starts from the block 19 with length = 6 blocks. Therefore, it occupies *19, 20, 21, 22, 23, 24* blocks.



Directory

file	start	length
count	0	2
tr	14	3
mail	19	6
list	28	4
f	6	2

- Contiguous allocation also suffers from external fragmentation. Small free disk spaces are created after allocation free block and deleting files.
- External fragmentation means there will require free space available but that is not contiguous. To solve the problem of external fragmentation, compaction method is used.
- One more problem is that how to calculate the space needed for a file. It is difficult to estimate the required space.

Advantages

- It is very easy to implement.
- There is a minimum amount of seek time.
- The disk head movement is minimum.
- Memory access is faster.
- It supports sequential as well as direct access.

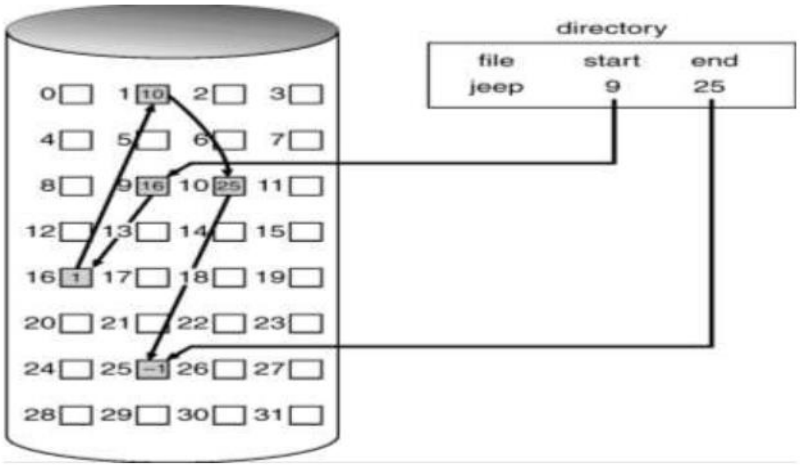
Disadvantages

- At the time of creation, the file size must be initialized.
- As it is pre-initialized, the size cannot increase.
- Due to its constrained allocation, it is possible that the disk would fragment internally or externally.

Linked File Allocation

The linked allocation solves all problems of contiguous allocation. In linked allocation, each file is a linked list of disk blocks. The allocated disk blocks may be scattered anywhere on the disk. The directory entry has a pointer to the first and the last blocks of the file.

Figure below shows an example of linked allocation.



There is a file named *jeep*. The starting disk block number 9 is given in the directory entry. Disk block 9 has the address of the next disk block, 16. Disk block 16 points to disk block 1 and so on. The last disk block of the file, disk block 25 has a pointer value of -1 indicating that it is the last disk block of the file.

Advantages

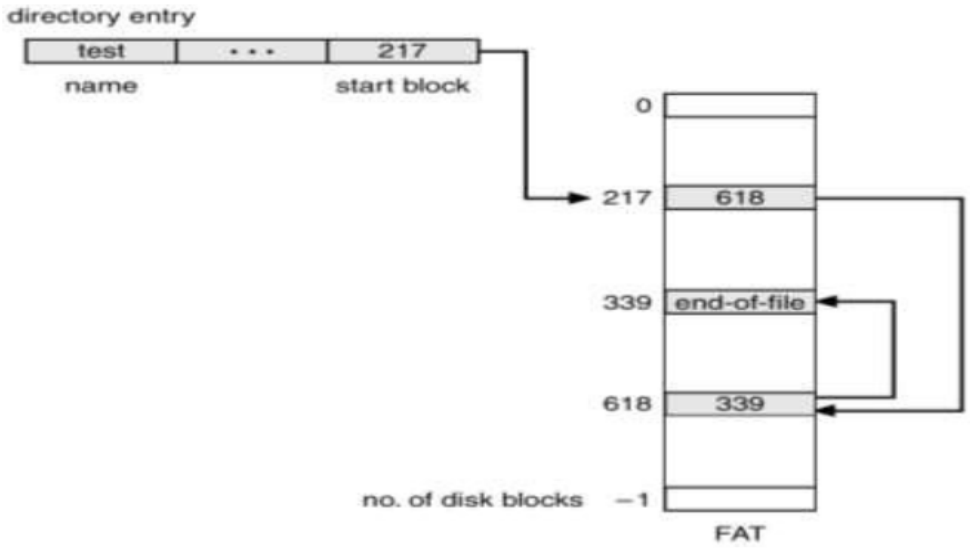
- There is no external fragmentation.
- The directory entry just needs the address of starting block.
- The memory is not needed in contiguous form, it is more flexible than contiguous file allocation.

Disadvantages

- It does not support random access or direct access.
- If pointers are affected so the disk blocks are also affected.(no reliability)
- Extra space is required for pointers in the block.

A variation of linked allocation scheme is used in the File-allocation table (FAT). This is the disk-space allocation scheme used by MS-DOS. A section of the disk at the beginning of each volume contains this table (FAT).

Figure below shows an example file allocation table. The directory entry has the address of the first disk block number (217). This 217 is used as an index into the FAT to get the entry where 618 is stored. This means that the next disk block number is 618. While using 618 as an index into the FAT, 339 is got, which means that, 339 is the next disk block. When 339 is used as index, the entry shows that end of file is reached.



Indexed Allocation

- The indexed allocation solves the external fragmentation and size-declaration problem of contiguous allocation. It also solves the direct access problem in linked allocation.
- In indexed allocation all pointers are brought together into one block called the *index block*. Each file has an index block, which is an array of disk-block addresses. The *i*th entry in the index block points to the *i*th block of the file.

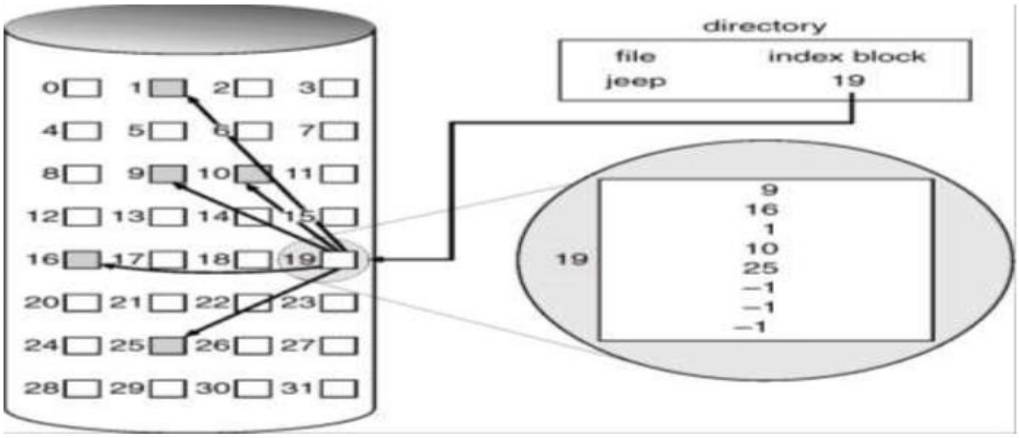


Figure above shows an example of indexed allocation. The directory entry has the address of the index block (disk block 19). The index block has the addresses of the disk blocks (9, 16, 1, 10 and 25) of the file.

Advantages:

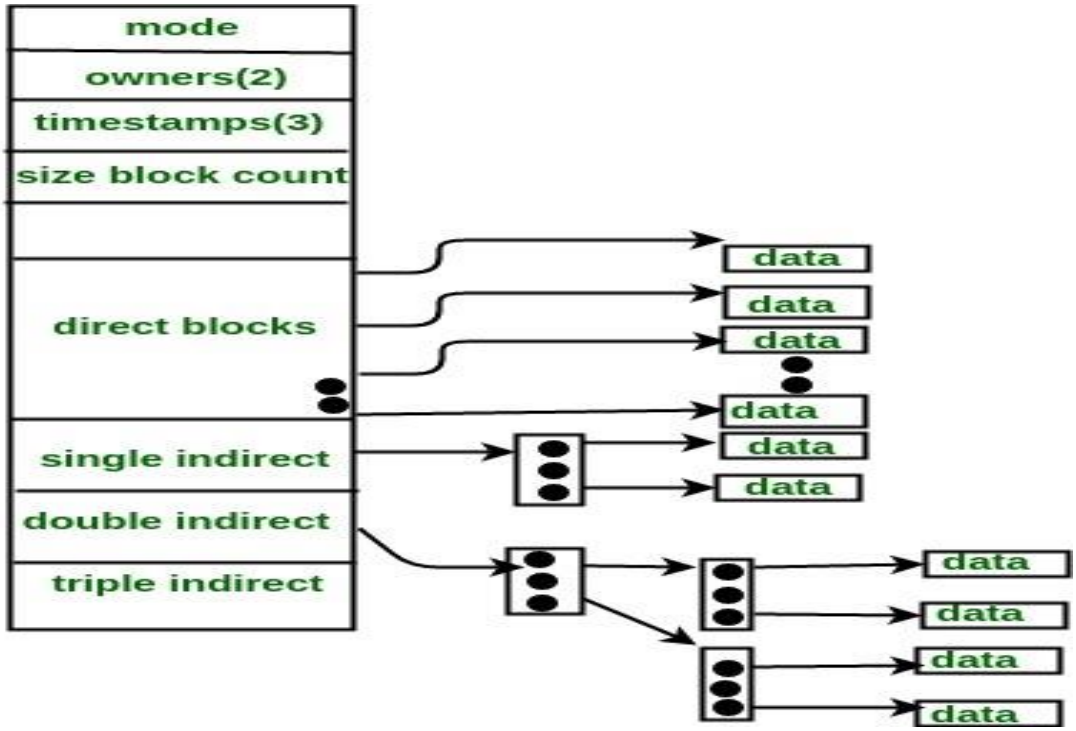
- This supports direct access to the blocks occupied by the file and therefore provides fast access to the file blocks.
- It overcomes the problem of external fragmentation.

Disadvantages:

- The pointer overhead for indexed allocation is greater than linked allocation.
- For very small files, say files that expand only 2-3 blocks, the indexed allocation would keep one entire block (index block) for the pointers which is inefficient in terms of memory utilization. However, in linked allocation we lose the space of only 1 pointer per block.

For files that are very large, single index block may not be able to hold all the pointers. Following mechanisms can be used to resolve this:

1. **Linked scheme:** This scheme links two or more index blocks together for holding the pointers. Every index block would then contain a pointer or the address to the next index block.
2. **Multilevel index:** In this policy, a first level index block is used to point to the second level index blocks which in turn points to the disk blocks occupied by the file. This can be extended to 3 or more levels depending on the maximum file size.
3. **Combined Scheme:** In this scheme, a special block called the **Inode (information Node)** contains all the information about the file such as the name, size, authority, etc and the remaining space of Inode is used to store the Disk Block addresses which contain the actual file *as shown in the image below.* The first few of these pointers in Inode point to the **direct blocks** i.e the pointers contain the addresses of the disk blocks that contain data of the file.
    - The next few pointers point to indirect blocks. Indirect blocks may be single indirect, double indirect or triple indirect.
    - **Single Indirect block** is the disk block that does not contain the file data but the disk address of the blocks that contain the file data.
    - Similarly, **double indirect blocks** do not contain the file data but the disk address of the blocks that contain the address of the blocks containing the file data.


mode
owners(2)
timestamps(3)
size block count
direct blocks
single indirect
double indirect
triple indirect
data
data
data
data
data
data
data
data
data
data

Unit-IV

Topic 12: Free Space Management

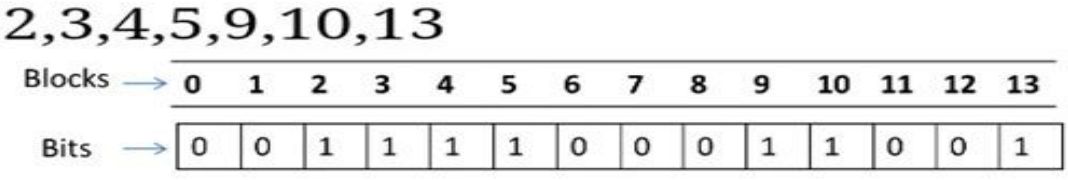
- Free space management is a critical aspect of operating systems as it involves managing the available storage space on the hard disk or other secondary storage devices
- The system maintains a free space list by keep track of the free disk space. The free space list contains all the records of the free space disk block.
- When we create a file, we first search for the free space in the memory and then check in the free space list for the required amount of space that we require for our file.
- if the free space is available then allocate this space to the new file. After that, the allocating space is deleted from the free space list. Whenever we delete a file then its free memory space is added to the free space list.

There are some methods or techniques to implement a free space list. These are as follows:

1. Bitmap
2. Linked list
3. Grouping
4. Counting
5. Space Maps

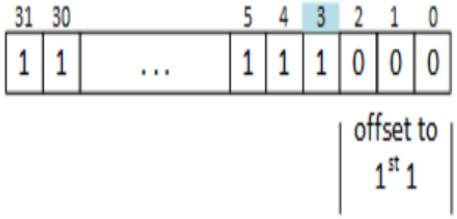
Bitmap

- This technique is used to implement the free space management. When the free space is implemented as the bitmap or bit vector then each block of the disk is represented by a bit. When the block is free its bit is set to 1 and when the block is allocated the bit is set to 0.
- The main advantage of the bitmap is it is relatively simple and efficient in finding the first free block and also the consecutive free block in the disk. Many computers provide the bit manipulation instruction which is used by the users.



The calculation of the block number is done by the formula:**(number of bits per words) X (number of 0-value word) + Offset of first 1 bit**

(d)




```
Number of bits per word = 32
number of 0-value words = 3
offset to the first 1 bit = 3

32 * 3 + 3 = 96 + 3 = 99
```

Advantages

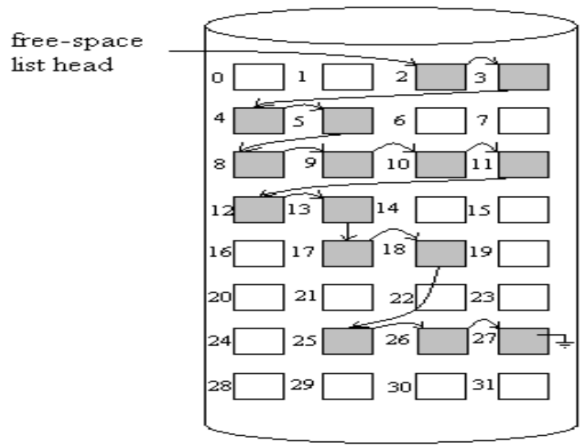
- This technique is relatively simple.
- This technique is very efficient to find the free space on the disk.

Disadvantages

- Again, we see hardware features driving software functionality. Unfortunately, bit vectors are inefficient unless the entire vector is kept in main memory (and is written to disk occasionally for recovery needs). Keeping it in main memory is possible for smaller disks but not necessarily for larger ones.

Linked list

- Another approach to free-space management is to link together all the free disk blocks, keeping a pointer to the first free block in a special location on the disk and caching it in memory. This first block contains a pointer to the next free disk block, and so on.
- If for example blocks 2, 3, 4, 5, 8, 9, 10, 11, 12, 13, 17, 18, 25, 26, and 27 were free and the rest of the blocks were allocated. In this situation, we would keep a pointer to block 2 as the first free block. Block 2 would contain a pointer to block 3, which would point to block 4, which would point to block 5, which would point to block 8, **as shown in figure below**



Advantages

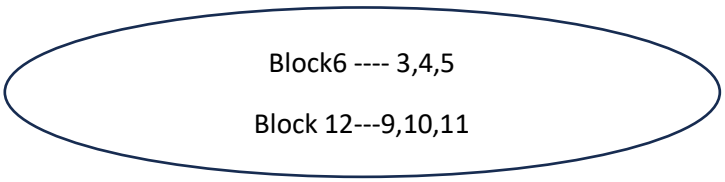
- Whenever a file is to be allocated a free block, the operating system can simply allocate the first block in free space list and move the head pointer to the next free block in the list.

Disadvantages

- Searching the free space list will be very time consuming; each block will have to be read from the disk, which is read very slowly as compared to the main memory.
- Not Efficient for faster access.

Grouping

- This is also the technique of free space management. In this, there is a modification of the free-list approach which stores the address of the n free blocks. In this the first n-1 blocks are free but the last block contains the address of the n blocks. When we use the standard linked list approach the addresses of a large number of blocks can be found very quickly.
- For example, consider a disk having 16 blocks where block numbers 3, 4, 5, 6, 9, 10, 11, 12, 13, and 14 are free, and the rest of the blocks, i.e., block numbers 1, 2, 7, 8, 15 and 16 are allocated to some files.
- If we apply the Grouping method considering *n* to be 4, Block 6 will store the addresses of Block 3, Block 4, and Block 5. Similarly, Block 12 will store the addresses of Block 9, Block 10, and Block 11. Block 11 will store the addresses of Block 12, Block 13, and Block 14. This is also represented in the following figure-



Advantage

1. By using this method, we can easily find addresses of a large number of free blocks easily and quickly.

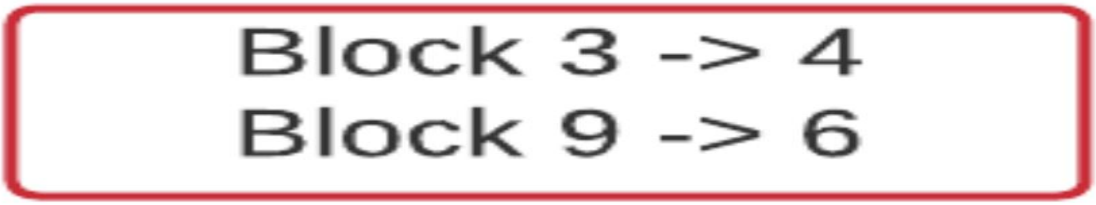
Disadvantage

1. We need to change the entire list if one block gets occupied.

Counting

- This is the fourth method of free space management in operating systems. This method is also a modification of the linked list method. This method takes advantage of the fact that several contiguous blocks may be allocated or freed simultaneously.
- In this method, a linked list is maintained but in addition to the pointer to the next free block, a count of free contiguous blocks that follow the first block is also maintained.
- Thus each free block in the disk will contain two things-
    1. A pointer to the next free block.
    2. The number of free contiguous blocks following it.

For example, consider a disk having 16 blocks where block numbers 3, 4, 5, 6, 9, 10, 11, 12, 13, and 14 are free, and the rest of the blocks, i.e., block numbers 1, 2, 7, 8, 15 and 16 are allocated to some files.If we apply the counting method, Block 3 will point to Block 4 and store the count 4 (since Block 3, 4, 5, and 6 are contiguous). Similarly, Block 9 will point to Block 10 and keep the count of 6 (since Block 9, 10, 11, 12, 13, and 14 are contiguous). This is also represented in the following figure-



Advantages

- Fast allocation of a large number of consecutive free blocks.
- Random access to the free block is possible.
- The overall list is smaller in size.

Disadvantages

- Each free block requires more space for keeping the count in the disk.
- For efficient insertion, deletion, and traversal operations. We need to store the entries in B-tree.

- The entire area is reduced.

Space map

ZFS (Zettabyte File System) is an advanced file system and logical volume manager designed by Sun Microsystems. One of its most notable features is its sophisticated free space management, which provides high performance, reliability, and efficiency.

Selects a Metaslab: *ZFS divides each virtual storage pool into metaslabs, which are chunks of storage (usually hundreds of MBs to a few GBs).*

ZFS uses a metaslab selection algorithm to choose the most appropriate metaslab for the allocation request. This can be based on factors like the amount of free space, recent usage patterns, and fragmentation levels.

Updates the Spacemap: *Spacemaps are stored in a log-structured format, recording free space transactions (allocations and deallocations) in a sequential manner*

- ZFS records the allocation in the metaslab's spacemap, adding an entry that specifies the start block and the length of the allocated space.

Updates Metadata: *Space maps track space allocation changes, recording the start block, length, and type of operation (allocation or deallocation*).

- The allocation is also reflected in the relevant metadata structures, ensuring that the file system remains consistent.

Advantages

- Space maps offer a powerful and efficient way to manage free space in large and dynamic storage systems. Their advantages include efficient space tracking, improved performance, consistency, and flexibility.

Disadvantages

Implementation Complexity: Managing space maps is more complex than simpler structures like bit vectors. This complexity can make the implementation more difficult and increase the potential for bugs.

- **Metadata Overhead:** While space maps reduce metadata overhead compared to tracking each block individually, they still consume memory to store the ranges and associated data. This overhead can become significant in extremely large storage systems.
- **Search Overhead:** Searching for free space in a highly fragmented space map can be slower compared to simpler structures, especially if the data structure used for the space map is not optimized for quick searches.

Unit-IV

Topic 13: System calls for File Management

These system call are responsible for all services related to file manipulation, such as writing into a file, reading from a file, creating a file, and so on.

*Some **system calls for file management** like*

create(): The create() function is used to create a new empty file in C. We can specify the permission and the name of the file which we want to create using the create() function.

syntax

int **create**(char **filename*, mode_t *mode*);

Parameter

filename: name of the file which you want to create

mode: indicates permissions of the new file.

The mode can be specified using a combination of the following constants, defined in <sys/stat.h>:

- S_IRUSR (or S_IREAD): Read permission for the owner.
- S_IWUSR (or S_IWRITE): Write permission for the owner.
- S_IXUSR (or S_IEXEC): Execute (or search, if it's a directory) permission for the owner.
- S_IRGRP: Read permission for the group.
- S_IWGRP: Write permission for the group.
- S_IXGRP: Execute (or search) permission for the group.
- S_IROTH: Read permission for others.
- S_IWOTH: Write permission for others.
- S_IXOTH: Execute (or search) permission for others.

These constants can be combined using the bitwise OR operator (|) to set multiple permissions.

Return Value

returns file descriptor)

return -1 when an error

open (): open() system call **is used to know the file descriptor of user-created files. Since read and write use file descriptor as their 1st parameter so to know the file descriptor open() system call is used.**

Syntax:

fd = open (file_name, mode, permission);

Example:

fd = open ("file", O_CREAT | O_RDWR, 0777);

Here,

file_name is the name to the file to open.

mode is used to define the file opening modes such as

O_RDONLY	Opens the file in read-only mode.
O_WRONLY	Opens the file in write-only mode.
O_RDWR	Opens the file in read and write mode.
O_CREAT	Create a file if it doesn't exist.
O_EXCL	Prevent creation if it already exists.
O_ APPEND	Opens the file and places the cursor at the end of the contents.
O_ASYNC	Enable input and output control by signal.
O_CLOEXEC	Enable close-on-exec mode on the open file.
O_NONBLOCK	Disables blocking of the file opened.
O_TMPFILE	Create an unnamed temporary file at the specified path.

permission is used to define the file permissions.

Return value: Function returns the file descriptor.

read (): read() system call is used to read the content from the file. It can also be used to read the input from the keyboard by specifying the **0** as file descriptor .

Syntax:

```
length = read(file_descriptor , buffer, max_len);
Example:
n = read(0, buff, 50);
```

Here,

- file_descriptor is the file descriptor of the file.
- buffer is the name of the buffer where data is to be stored.
- max_len is the number specifying the maximum amount of that data can be read

Return value: If successful read returns the number of bytes actually read.

write (): write() system call is used to write the content to the file.

Syntax:

```
length = write(file_descriptor , buffer, len);
Example:
n = write(fd, "Hello world!", 12);
```

Here,

- file_descriptor is the file descriptor of the file.
- buffer is the name of the buffer to be stored.
- len is the length of the data to be written.

Return value: If successful write() returns the number of bytes actually written.

close (): close() system call is used to close the opened file, it tells the operating system that you are done with the file and close the file.

Syntax:

```
int close(int fd);
```

Here,

- fd is the file descriptor of the file to be closed.

***Return value*:** If file closed successfully it returns 0, else it returns -1.

```
#include<unistd.h>
#include<fcntl.h>
#include<sys/stat.h>
#include<sys/types.h>
#include<stdio.h>
int main()
{
        int n,fd;
        char buff[50];
        printf("Enter text to write in the file:\n");
        n= read(0, buff, 50);
        fd=open("file",O_CREAT | O_RDWR, 0777);
        write(fd, buff, n);
        write(1, buff, n);
        close(int fd);
        return 0;
}
```

lseek(): lseek() system call repositions the read/write file offset i.e., it changes the positions of the read/write pointer within the file. In every file any read or write operations happen at the position pointed to by the pointer. lseek() system call helps us to manage the position of this pointer within a file.

e.g., let's suppose the content of a file F1 is "1234567890" but you want the content to be "12345hello". You simply can't open the file and write "hello" because if you do so then "hello" will be written in the very beginning of the file. This means you need to reposition the pointer after '5' and then start writing "hello". lseek() will help to reposition the pointer and write() will be used to write "hello"

The first parameter is the file descriptor of the file, which you can get using open() system call. the second parameter specifies how much you want the pointer to move and the third parameter is the

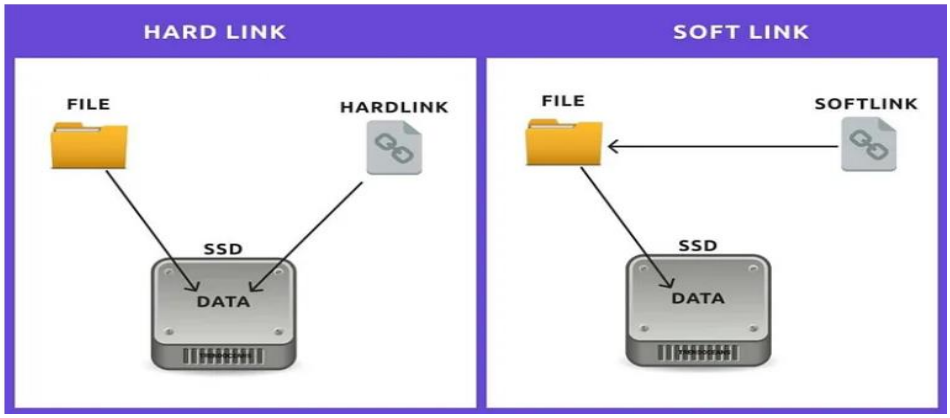
reference point of the movement i.e., beginning of file(SEEK_SET), current position(SEEK_CUR) of pointer or end of file(SEEK_END).

Examples:

- lseek(fd,5,SEEK_SET) – this moves the pointer 5 positions ahead starting from the beginning of the file
- lseek(fd,5,SEEK_CUR) – this moves the pointer 5 positions ahead from the current position in the file
- lseek(fd,-5,SEEK_CUR) – this moves the pointer 5 positions back from the current position in the file
- lseek(fd,-5,SEEK_END) -> this moves the pointer 5 positions back from the end of the file

On success, lseek() returns the position of the pointer within the file as measured in bytes from the beginning of the file. But, on failure, it returns -1.

A link in UNIX is a pointer to a file. Like pointers in any programming languages, links in UNIX are pointers pointing to a file or a directory. Creating links is a kind of shortcuts to access a file. Links allow more than one file name to refer to the same file. There are two types of links :



1. **Soft Link:**

   A soft link (also known as Symbolic link) acts as a pointer or a reference to the file name. It does not access the data available in the original file. If the earlier file is deleted, the soft link will be pointing to a file that does not exist anymore.

2. **Hard Link :**

   A hard link acts as a copy (mirrored) of the selected file. It accesses the data available in the original file.If the earlier selected file is deleted, the hard link to the file will still contain the data of that file.

link() & symlink()

In Unix-like operating systems, link() and symlink() are system calls used to create hard links and symbolic links (symlinks) respectively. Here's an explanation of each:

link(): The link() function creates a new link (hard link) to an existing file.

#include <unistd.h>

int link(const char *oldpath, const char *newpath);

- **Parameters:**

const char *oldpath: The path name of an existing file.

const char *newpath: The path name of the new link to be created.

- **Returns:**

0 on success.

-1 on failure, and errno is set to indicate the error

symlink(): The symlink() function creates a symbolic link (symlink) to a target file.

#include <unistd.h>

int symlink(const char *target, const char *linkpath);

- **Parameters:**

const char *target: The path name of the target file that the symlink will point to.

const char *linkpath: The path name where the symbolic link will be created.

- **Returns:**

0 on success.

-1 on failure, and errno is set to indicate the error.

Unlink(): The unlink() function in Unix-like operating systems is used to delete a name (unlink a file) from the file system. It removes the link to a file, and if that was the last link to the file and no processes have the file open, the file itself is deleted and the space it was using is made available for reuse.

#include <unistd.h>

int unlink(const char *pathname);

Parameters

const char *pathname: The path name of the file to be unlinked (deleted).

Return Value

Returns 0 on success.

Returns -1 on failure, and sets errno to indicate the error

```
#include <unistd.h>
#include <stdio.h>
#include <errno.h>
int main() {
   const char *sourcefile = "source.txt";
   const char *hardlink = "hardlink.txt";
   const char *symlinkfile = "shortcut";
   // Creating a hard link
   if (link(sourcefile, hardlink) == -1) {
      perror("link");
      return 1;    }
   printf("Hard link created: %s -> %s\n", hardlink, sourcefile);
   // Creating a symbolic link
   if (symlink(sourcefile, symlinkfile) == -1) {
      perror("symlink");
      return 1;    }
   printf("Symbolic link created: %s -> %s\n", symlinkfile, sourcefile);
   // Deleting the source file using unlink()
   if (unlink(sourcefile) == -1) {
      perror("unlink");
      return 1;    }
   printf("File '%s' deleted successfully.\n", sourcefile);
   return 0;}
```

stat(),fstat() and lstat(): The stat, fstat, and lstat functions in Unix-like operating systems are used to retrieve detailed information about files and directories. They are part of the system's API and are declared in the <sys/stat.h> header file. Here's an overview of each function, their similarities, and differences:

stat(): The stat function retrieves information about a file specified by its pathname.

```
#include <sys/stat.h>
int stat(const char *pathname, struct stat *buf);
```

Parameters:

const char *pathname: The path to the file.

struct stat *buf: A pointer to a stat structure where the information will be stored.

Returns:

0 on success.

-1 on failure, and errno is set to indicate the error.

fstat(): The fstat function retrieves information about an open file referred to by a file descriptor.

```
#include <sys/stat.h>
int fstat(int fd, struct stat *buf);
```

Parameters:

int fd: The file descriptor of the file.

struct stat *buf: A pointer to a stat structure where the information will be stored.

Returns:

0 on success.

-1 on failure, and errno is set to indicate the erro

lstat(): The lstat function is similar to stat, but it retrieves information about a symbolic link itself rather than the file it points to.

```
#include <sys/stat.h>
int lstat(const char *pathname, struct stat *buf);
```

Parameters:

const char *pathname: The path to the file or symbolic link.

struct stat *buf: A pointer to a stat structure where the information will be stored.

Returns:

0 on success.

-1 on failure, and errno is set to indicate the error.

stat Structure

All three functions use the stat structure to store information about the file. Here is a brief description of some of the fields in the stat structure:

```
struct stat {
   dev_t     st_dev;     // Device ID
   ino_t     st_ino;     // Inode number
   mode_t    st_mode;    // File type and mode
   nlink_t   st_nlink;   // Number of hard links
```

```
    uid_t     st_uid;     // User ID of owner
    gid_t     st_gid;     // Group ID of owner
    dev_t     st_rdev;    // Device ID (if special file)
    off_t     st_size;    // Total size, in bytes
    blksize_t st_blksize; // Block size for filesystem I/O
    blkcnt_t  st_blocks;  // Number of 512B blocks allocated
    time_t    st_atime;   // Time of last access
    time_t    st_mtime;   // Time of last modification
    time_t    st_ctime;   // Time of last status change
};
#include <sys/stat.h>
#include <fcntl.h>
#include <unistd.h>
#include <stdio.h>
#include <errno.h>
void print_stat_info(const char *label, const struct stat *fileStat) {
    printf("%s:\n", label);
    printf("File size: %ld bytes\n", fileStat->st_size);
    printf("Number of links: %lu\n", fileStat->st_nlink);
    printf("File inode: %lu\n", fileStat->st_ino);
    printf("File mode: %o\n", fileStat->st_mode);
    printf("File UID: %d\n", fileStat->st_uid);
    printf("File GID: %d\n", fileStat->st_gid);
    printf("Last access time: %ld\n", fileStat->st_atime);
    printf("Last modification time: %ld\n", fileStat->st_mtime);
    printf("Last status change time: %ld\n", fileStat->st_ctime);
    printf("\n");
}
int main() {
    const char *filename = "example.txt";
    const char *symlinkname = "example_symlink";
    // Using stat()
```

```
    struct stat statBuf;
    if (stat(filename, &statBuf) == -1) {
        perror("stat");
        return 1;    }
    print_stat_info("stat()", &statBuf);
    // Using fstat()
    int fd = open(filename, O_RDONLY);
    if (fd == -1) {
        perror("open");
        return 1;    }
    struct stat fstatBuf;
    if (fstat(fd, &fstatBuf) == -1) {
        perror("fstat");
        close(fd);
        return 1;    }
    print_stat_info("fstat()", &fstatBuf);
    close(fd);
    // Using lstat()
    struct stat lstatBuf;
    if (lstat(symlinkname, &lstatBuf) == -1) {
        perror("lstat");
        return 1;    }
    print_stat_info("lstat()", &lstatBuf);
    if (S_ISLNK(lstatBuf.st_mode)) {
        printf("%s is a symbolic link\n", symlinkname);
    } else {
        printf("%s is not a symbolic link\n", symlinkname);
    }    return 0;}
```

chmod(): The chmod command is used to change the permissions of a file or directory in Unix-like operating systems.

The command chmod(“newfiles”, 00444); sets the permissions of the directory or file named newfiles to 444.

Here's what 00444 means in terms of permissions:

- The leading 0 indicates that this is an octal number.
- The 444 indicates the permission settings for the owner, group, and others.

In octal notation: 4 corresponds to read-only permission.

chown(): The chown function in Unix-like operating systems is used to change the ownership of a file or directory. It allows you to specify a new owner and optionally a new group for a given file or directory

#include <unistd.h>

int chown(const char *pathname, uid_t owner, gid_t group);

Parameters

- const char *pathname: The path to the file or directory whose ownership you want to change.
- uid_t owner: The user ID (UID) of the new owner. If this is -1, the owner is not changed.
- gid_t group: The group ID (GID) of the new group. If this is -1, the group is not changed.

Return Value

- Returns 0 on success.
- Returns -1 on failure, and errno is set to indicate the error.

Example

```
#include <unistd.h>
#include <stdio.h>
#include <errno.h>
int main() {
   const char *path = "example.txt";
   uid_t new_owner = 1001; // New owner UID
   gid_t new_group = 1001; // New group GID
   if (chown(path, new_owner, new_group) == -1) {
      perror("chown");
      return 1;    }
   printf("Ownership of '%s' changed to UID %d and GID %d\n", path, new_owner, new_group);
   return 0;}
```

Unit-IV

Topic 14: Directory System calls

Opendir() : The opendir function opens a directory stream corresponding to the directory name, and returns a pointer to the directory stream.

#include <dirent.h>

DIR *opendir(const char *name);

Parameters:

- const char *name: A pointer to a string containing the name of the directory to be opened.

Returns:

- A pointer to a DIR structure on success.
- NULL on failure, and errno is set to indicate the error.

Example:

```
#include <dirent.h>
#include <errno.h>
DIR *dir = opendir("some_directory");
if (dir == NULL) {
   // Handle error
} else {
   // Directory opened successfully
}
```

readdir(): The readdir function reads the next directory entry from the directory stream.

#include <dirent.h>

struct dirent *readdir(DIR *dirp);

Parameters:

- DIR *dirp: A pointer to the directory stream (opened by opendir).

Returns:

- A pointer to a struct dirent representing the next directory entry.
- NULL at the end of the directory stream or on error (check errno to distinguish).

Example:

```
#include <dirent.h>
```

```
#include <stdio.h>
#include <errno.h>
DIR *dir = opendir("some_directory");
if (dir != NULL) {
   struct dirent *entry;
   while ((entry = readdir(dir)) != NULL) {
      printf("%s\n", entry->d_name);
   }
   closedir(dir);
} else {
   // Handle error
}
```

closedir() : The closedir function closes the directory stream.

```
#include <dirent.h>
int closedir(DIR *dirp);
```

Parameters:

- DIR *dirp: A pointer to the directory stream to be closed.

Returns:

- 0 on success.
- -1 on failure, and errno is set to indicate the error.

Example

```
#include <dirent.h>
#include <errno.h>
DIR *dir = opendir("some_directory");
if (dir != NULL) {
   // Read entries
   closedir(dir);
} else {
   // Handle error
}
```

mkdir(): The mkdir function creates a new directory with the specified name and permissions.

int mkdir(char* name, int mode);

Parameters:

- char* name: A pointer to a string that specifies the name of the directory to be created.
- int mode: An integer that specifies the permissions for the new directory, usually given in octal format (e.g., 0755).

Returns:

- 0 on success.
- -1 on failure, and errno is set to indicate the error.

Example:

```
#include <sys/stat.h>
#include <errno.h>
int result = mkdir("new_directory", 0755);
if (result == 0) {
   // Directory created successfully
} else {
   // Handle error
}
```

rmdir : The rmdir function removes an empty directory with the specified name.

int rmdir(char* name);

Parameters:

- char* name: A pointer to a string that specifies the name of the directory to be removed.

Returns:

- 0 on success.
- -1 on failure, and errno is set to indicate the error.

Example:

```
#include <unistd.h>
#include <errno.h>
int result = rmdir("new_directory");
if (result == 0) {
   // Directory removed successfully
} else {
```

```
    // Handle error
}
```

The directory must be empty before it can be removed using rmdir. If the directory is not empty or does not exist, rmdir will fail and set errno appropriately.

umask

The umask (short for "user file creation mask") is a function and command in Unix-like operating systems that sets the default permissions for newly created files and directories. The umask value is subtracted from the default permissions to determine the actual permissions assigned when files and directories are created.

Understanding umask

The default permissions before applying the umask are:

- **Directories:** 777 (rwxrwxrwx)
- **Files:** 666 (rw-rw-rw-)

The umask value is used to mask out certain permission bits so that they are not set when new files and directories are created.

Setting and Viewing umask

You can view the current umask value by running the umask command without arguments:

$ umask

output : 0022.

You can set the umask value by running the umask command with an argument:

$ umask 0022

How umask Affects Permissions

The umask value is subtracted from the default permissions to determine the actual permissions:

If the umask is 0022:

Directory

Default permissions: 777

umask: 0022

Resulting permissions: 777 - 022 = 755

Files

Default permissions: 666

umask: 0022

Resulting permissions: 666 - 022 = 644

```
#include <sys/stat.h>
#include <stdio.h>
#include <fcntl.h>
#include <unistd.h>
int main() {
   // Set umask to 0022
   umask(0022);
   // Create a new file with default permissions
   int fd = open("example.txt", O_CREAT | O_WRONLY, 0666);
   if (fd == -1) {
      perror("open");
      return 1;
   }
   // Write to the file
   write(fd, "Hello, World!\n", 14);
   // Close the file
   close(fd);
   // Check the permissions of the created file
   printf("File 'example.txt' created with umask 0022\n");
   return 0;
}
```