UNIT-IV

MEMORY MANAGEMENT

MAIN MEMORY

The main purpose of a computer system is to execute programs.

During the execution of these programs together with the data they access must be stored in main memory.

Memory consists of a large array of bytes. Each Byte has its own address.

CPU fetches instructions from memory according to the value of the program counter.

BASIC HARDWARE

CPU can access data directly only from Main memory and processor registers.

Main memory and the Processor registers are called **Direct Access Storage Devices**.

Any instructions in execution and any data being used by the instructions must be in one of these direct-access storage devices.

If the data are not in memory, then the data must be moved to main memory before theCPU can operate on them.

Registers that are built into the CPU are accessible within one CPU clock cycle.

Completing a memory access from main memory may take many CPU clock cycles. Memory access from main memory is done through memory bus.

In such cases, the processor needs to **stall**, since it does not have the required data to complete the instruction that it is executing.

To avoid **memory stall**, we need to implement Cache memory in between Main memory and CPU.

BASE REGISTER & LIMIT REGISTER

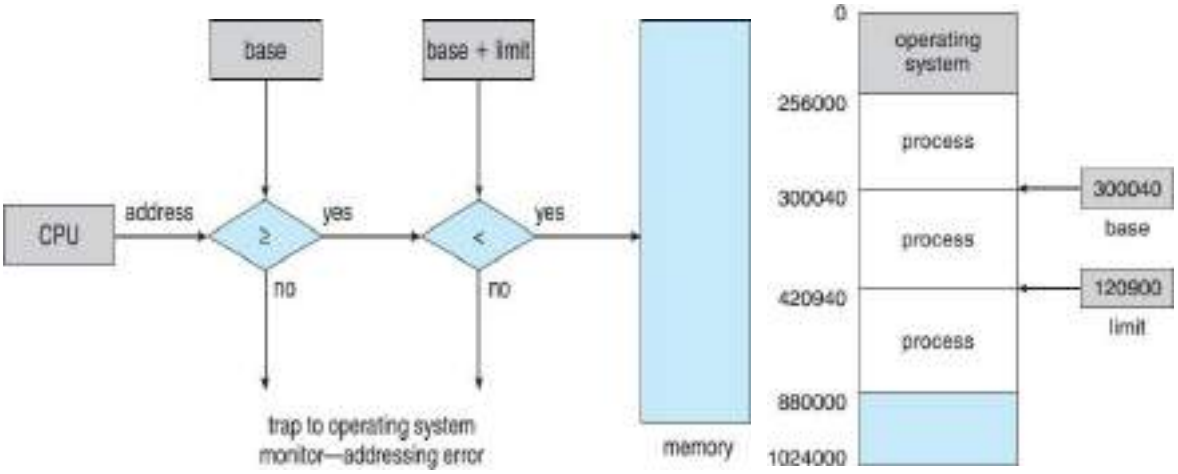
Each process has a separate memory space that protects the processes from each other. It is fundamental to having multiple processes loaded in memory for concurrent execution.

There are two register that provides protection: Base register and Limit register

Base Register holds the smallest legal physical memory address.

Limit register specifies the size of the range (i.e. process size).

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).



The base and limit registers can be loaded only by the operating system by using a special privileged instruction that can be executed only in kernel mode.

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 either accidentally or deliberately modifying the code or data structures of other users and the operating system.

Operating system executing in kernel mode is given unrestricted access to both operating-system memory and users' memory. This provision allows the operating system to do certain tasks such as:

Load users' programs into users' memory

To dump out those programs in case of errors

To access and modify parameters of system calls

To perform I/O to and from user memory etc.

Example: A Multiprocessing Operating system must execute context switches, storing the state of one process from the registers into main memory before loading the next process's context from main memory into the registers.

Address Binding

A program resides on a disk as a binary executable file. The program must be brought into memory and placed within a process for execution.

The process may be moved between disk and memory during its execution.

The processes on the disk that are waiting to be brought into memory for execution are put into the **Input Queue**.

Addresses may be represented in different ways during these steps.

Addresses in the source program are generally symbolic, such as the variable **count**.

A compiler typically **binds** these symbolic addresses to **Relocatable addresses** such as "14 bytes from the beginning of this module".

The **Linkage** editor or **Loader** in turn binds **Relocatable addresses** to **Absolute addresses** such as **74014** (i.e. 74000+14=74014).

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

Binding of instructions and data to memory addresses can be done at any of following steps:

Compile time.

If you know at compile time where the process will reside in memory then **Absolute code** can be generated.

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.

After some time, if the starting location has been changed then it will be necessary to recompile this code.

The MS-DOS .COM-format programs are bound at compile time.

Load time

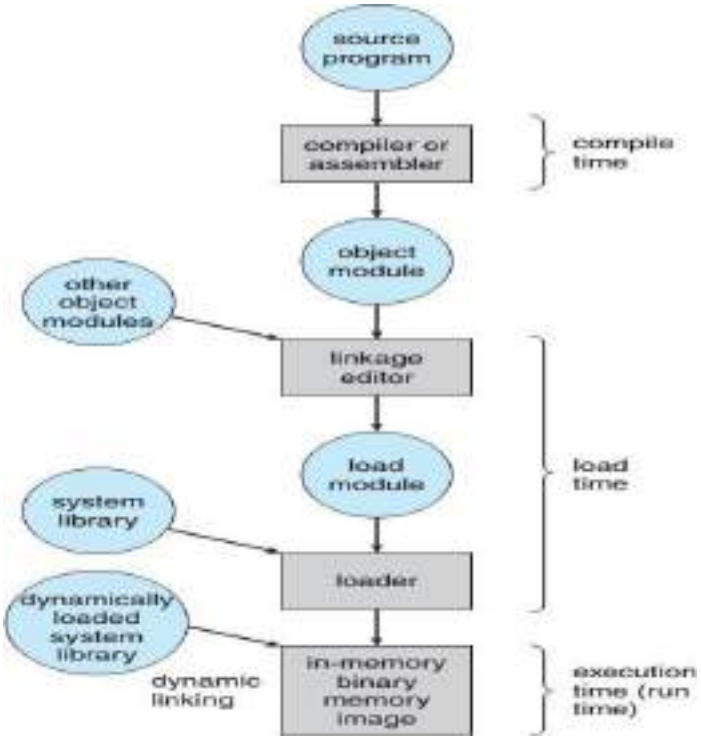
If it is not known at compile time where the process will reside in memory, then the compiler must generate **Relocatable code**.

In this case, final binding is delayed until load time. If the starting address changes, we need to reload only the user code to incorporate this changed value.

Execution time

If the process can be moved during its execution from one memory segment to another, then binding must be delayed until run time.

Most general-purpose operating systems use this method.



Logical Versus Physical Address Space

Logical address is the address generated by the CPU.

Physical address is the address that is loaded into the **Memory-Address Register** of the memory.

The set of all logical addresses generated by a program is a **Logical Address Space**.

The set of all physical addresses corresponding to these logical addresses is a **Physical** A**ddress Space**.

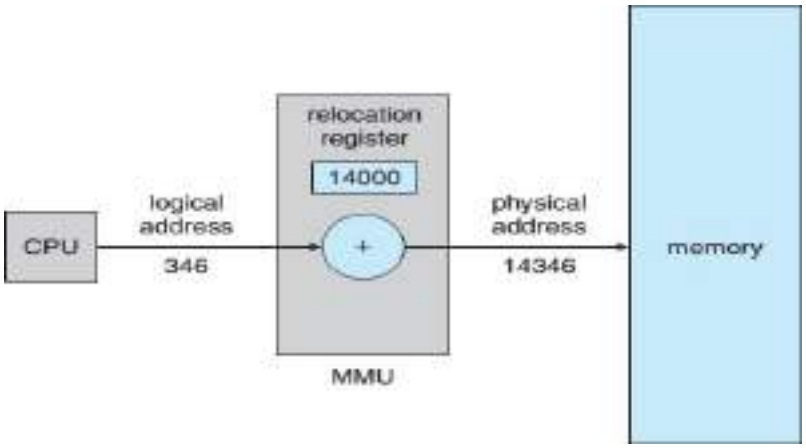
The Compile-time and Load-time address-binding methods generate identical logical and physical addresses.

The execution-time address binding scheme results in different logical and physical addresses. At this time, we call logical address as **Virtual address**.

The run-time mapping from virtual address to physical addresses is done by a hardware device called the **Memory-Management Unit (MMU)**.

Base register is now called a **Relocation Register**. Value in the relocation register is added to every address generated by a user process at the time the address is sent to memory.

Example: If the base is at 14000, then an attempt by the user to address location 0 is dynamically relocated to location 14000. An access to location 346 is mapped to location 14346.



The user program never sees the real Physical addresses. The program can create a pointer to location 346, store it in memory, manipulate it and compare it with other addresses all as the number 346.

Only when it is used as a memory address, it is relocated relative to the base register.

The user program deals with logical addresses. The Memory-mapping hardware converts logical addresses into physical addresses.

Final location of a referenced memory address is not determined until the reference is made.

Example: Logical addresses in the range **0 to max** and Physical addresses in the range **(R+0)** to (**R + max**) for a base value **R**.

The user program generates only logical addresses and thinks that the process runs in locations 0 to max.

These logical addresses must be mapped to physical addresses before they are used.

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 it 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 loaded, 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.

Advantage: It is 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 may be much smaller.

Note: It is the responsibility of the users to design their programs to support Dynamic linking. Operating systems may help the programmer by providing library routines to implement dynamic loading.

Dynamic Linking

Dynamically linked libraries are system libraries that are linked to user programs when the programs are running.

In static linking system libraries are treated like any other object module and they are combined by the loader into the binary program image.

In Dynamic linking, the linking is postponed until execution time.

This feature is usually used with system libraries, such as language subroutine libraries.

Without dynamic linking, each program on a system must include a copy of its language library in the executable image. This will waste both disk space and main memory.

With dynamic linking, a **stub** is included in the image for each library routine reference.

The stub is a small piece of code that indicates how to locate the appropriate memory-resident library routine or how to load the library if the routine is not already present.

When the stub is executed, it checks to see whether the needed routine is already in memory. If it is not, the program loads the routine into memory.

The stub replaces itself with the address of the routine and executes the routine.

Thus, the next time that particular code segment is reached, the library routine is executed directly, incurring no cost for dynamic linking.

Under this scheme, all processes that use a language library execute only one copy of the library code.

Shared Libraries

A library may be replaced by a new version and all programs that reference the library will automatically use the new version.

Without dynamic linking, all such programs would need to be relinked to gain access to the new library.

So that programs will not accidentally execute new or incompatible versions of libraries.

Version information is included in both the program and the library.

More than one version of a library may be loaded into memory and each program uses its version information to decide which copy of the library to use.

Versions with minor changes retain the same version number, whereas versions with major changes increment the number.

Thus, only programs that are compiled with the new library version are affected by any incompatible changes incorporated in it.

Other programs linked before the new library was installed will continue using the older library. This system is also known as **shared libraries**.

Note: Dynamic linking and shared libraries require help from the operating system.

SWAPPING

A process must be in **Main memory** to be executed. A process can be **swapped** temporarily out of main memory to a **backing store** and then brought back into main-memory for continued execution.

Swapping makes it possible for the total physical address space of all processes to exceed the real physical memory of the system, thus increasing the degree of multiprogramming in a system.

Standard Swapping

Standard swapping involves moving processes between main memory and a backing store.

The backing store is commonly a fast disk (i.e. Hard Disk). It must be large enough to accommodate copies of all memory images for all users and it must provide direct access to these memory images.

The system maintains a **Ready Queue** consisting of all processes whose memory images are on the backing store or in memory and the processes are ready to run.

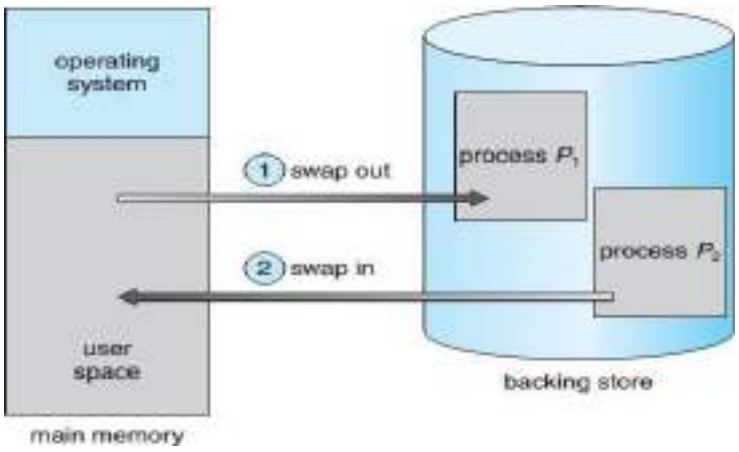
Whenever the CPU scheduler decides to execute a process, it calls the dispatcher.

The dispatcher checks to see whether the next process in the queue is in main memory.

If it is not in main memory and if there is no free memory region, the dispatcher **swaps out** a process currently in main memory and **swaps in** the desired process. It then reloads registers and transfers control to the selected process.

The context-switch time in such a swapping system is fairly high. The major part of the swap time is transfer time. The total transfer time is directly proportional to the amount of memory swapped.

If we want to swap a process, we must be sure that the process is completely idle such as waiting for I/O.



Standard swapping is not used in modern operating systems. It requires too much swapping time and provides too little execution time to be a reasonable memory-management solution.

UNIX, Linux and Windows use modified versions of swapping as below:

Swapping is enabled only when the amount of free memory falls below a threshold amount.

Swapping is disabled when the amount of free memory increases.

Operating system swaps portions of processes rather than the entire process to decrease the swap time.

Note: This type of swapping works in conjunction with Virtualization.

Swapping on Mobile systems

Mobile systems such as iOS and Android do not support swapping.

Mobile devices generally use flash memory rather than more spacious Hard disks as their persistent storage.

Mobile operating-system designers avoid swapping because of the less space constraint.

Flash memory can tolerate only the limited number of writes before it becomes unreliable and the poor throughput between main memory and flash memory in these devices.

Alternative methods used in Mobile systems instead of swapping:

Apple's iOS asks applications to voluntarily relinquish allocated memory when free memory falls below a certain threshold.

Read-only data (i.e. code) are removed from the system and later reloaded from flash memory if necessary.

Data that have been modified such as the **stack** are never removed.

Any applications that fail to free up sufficient memory may be terminated by the operating system.

Android may terminate a process if insufficient free memory is available. Before terminating a process android writes its **Application state** to flash memory so that it can be quickly restarted.

CONTIGUOUS MEMORY ALLOCATION

Memory allocation can be done in two ways:

1. Fixed Partition Scheme (Multi-programming with Fixed Number of Tasks)
2. Variable partition scheme (Multi-programming with Variable Number of Tasks)

Fixed Partition Scheme (MFT)

The memory can be divided into several **Fixed-Sized** partitions.

Each partition may contain exactly one process. Thus, the degree of multiprogramming is bound by the number of partitions.

In this **Multiple-Partition** method, when a partition is free, a process is selected from the input queue and is loaded into the free partition.

When the process terminates, the partition becomes available for another process.

Note: This method was originally used by the IBM OS/360 operating system (called MFT) but is no longer in use.

Variable partition scheme (MVT)

In the variable-partition scheme, the operating system keeps a table indicating which parts of memory are available and which are occupied.

Initially, all memory is available for user processes and it is considered one large block of available memory called as **Hole**.

Eventually the memory contains a set of holes of various sizes.

As processes enter the system, they are put into an **Input Queue**.

The operating system takes into account the memory requirements of each process and the amount of available memory space in determining which processes are allocated memory.

When a process is allocated space, it is loaded into memory and it can then compete for CPU time.

When a process terminates, it releases its memory. The operating system may use this free fill with another process from the input queue.

Memory is allocated to processes until the memory requirements of the next process cannot be satisfied (i.e.) there is no available block of memory is large enough to hold that process.

Then operating system can wait until a large block is available for the process or it can skip the process and moves down to the input queue to see whether the smaller memory requirements of some other process can be met.

The memory blocks available comprise a **set** of holes of various sizes scattered throughout main memory.

When a process arrives and needs memory, the system searches the set for a hole that is large enough for this process.

If the hole is too large, it is split into two parts. One part is allocated to the arriving process and the other part is returned to the set of holes.

When a process terminates, it releases its block of memory, which is then placed back in the set of holes.

If the new hole is adjacent to other holes, these adjacent holes are merged to form one larger hole.

At this point, the system may need to check whether there are processes waiting for memory and whether this newly freed and recombined memory could satisfy the demands of any of these waiting processes.

Dynamic Storage Allocation Problem:

The above procedure leads to Dynamic storage allocation problem which concerns how to satisfy a request of size **n** from a list of free holes.

There are 3-solutions for this problem: First fit, Best fit, worst fit.

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

Best fit. It allocates the smallest hole that is big enough. We must search the entire list, unless the list is ordered by size. This strategy produces the smallest leftover hole.

Worst fit. It allocates the largest hole. Again, we must search the entire list, unless it is sorted by size. This strategy produces the largest leftover hole, which may be more useful than the smaller leftover hole from a best-fit approach.

FRAGMENTATION

There are 2-problems with Memory allocation

1. Internal Fragmentation
2. External Fragmentation

Internal Fragmentation

Consider a multiple-partition allocation scheme with a hole of 18,464 bytes.

Suppose that the next process requests 18,462 bytes. If we allocate exactly the requested block, we are left with a hole of 2 bytes.

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

The general approach to avoiding this problem is to break the physical memory into fixed-sized blocks and allocate memory in units based on block size.

With this approach, the memory allocated to a process may be slightly larger than the requested memory.

The difference between these two numbers is **Internal Fragmentation**. It is unused memory that is internal to a partition.

External Fragmentation

Both the first-fit and best-fit strategies for memory allocation suffer from **External Fragmentation**.

As processes are loaded and removed from main memory, the free memory space is broken into small pieces.

External fragmentation exists when there is enough total memory space to satisfy a request but the available spaces are not contiguous, the storage is fragmented into a large number of small holes.

External fragmentation problem can be severe. In the worst case, we could have a block of free memory between every two processes that is wasted.

If all these small pieces of memory were in one big free block instead, we might be able to run several more processes.

Solution to External fragmentation

One solution to the problem of external fragmentation is **Compaction**.

The goal is to shuffle the memory contents so as to place all free memory together in one large block.

Compaction is possible only if relocation is dynamic and is done at execution time.

If addresses are relocated dynamically, relocation requires only moving the program and data and then changing the base register to reflect the new base address.

If relocation is static and is done at assembly or load time, compaction cannot be done.

Note: Compaction can be expensive, because it moves all processes toward one end of memory. All holes move in the other direction and produces one large hole of available memory.

Other solutions to External fragmentation are **Segmentation** and **Paging.** They allow a process to be allocated physical memory wherever such memory is available. These are non-contiguous memory allocation techniques.

Memory Protection

OS can prevent a process from accessing other process memory. We use two registers for this purpose: Relocation register and Limit register.

Relocation register contains the value of the smallest physical address such as 100040.

Limit register contains the range of logical addresses such as 74600.

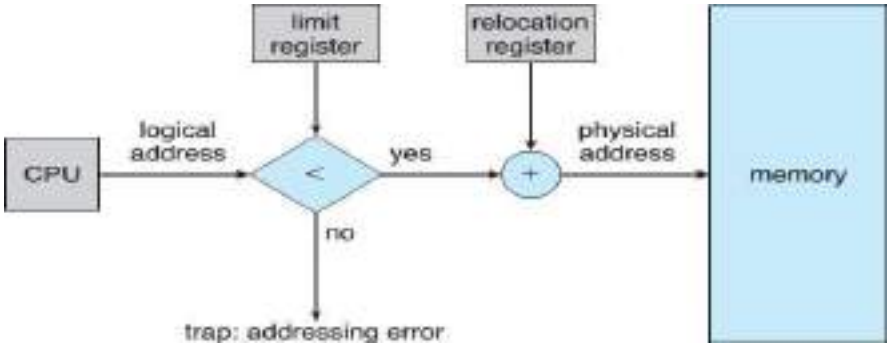
Each logical address must be within the range specified by the limit register.

The relocation-register scheme provides an effective way to allow the operating system's size to change dynamically.

Memory Management Unit maps the logical address dynamically by adding the value in the relocation register. This mapped address is sent to memory

When the CPU scheduler selects a process for execution, the dispatcher loads the relocation register and limit registers with the correct values as part of the context switch.

Because every address generated by a CPU is checked against these registers, we can protect both the operating system and the other users' programs and data from being modified by this running process.



SEGMENTATION

Segmentation is a memory-management scheme that permits the physical address space of a process to be noncontiguous.

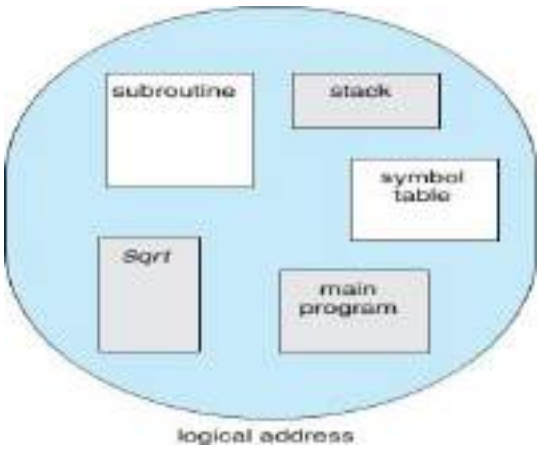
A logical address space is a collection of segments. Each segment has a name and a length.

Logical addresses specify both the segment name and the offset within the segment.

The programmer specifies each address by two quantities: a segment name and an offset.

The segments are referred to by Segment Number.

A logical address consisting of two tuples: **<Segment Number, offset>**



A C compiler might create separate segments for the following:

The code

Global variables

The heap, from which memory is allocated

The stacks used by each thread

The standard C library

Note: Libraries that are linked in during compile time might be assigned separate segments. The loader would take all these segments and assign them segment numbers.

Segmentation Hardware

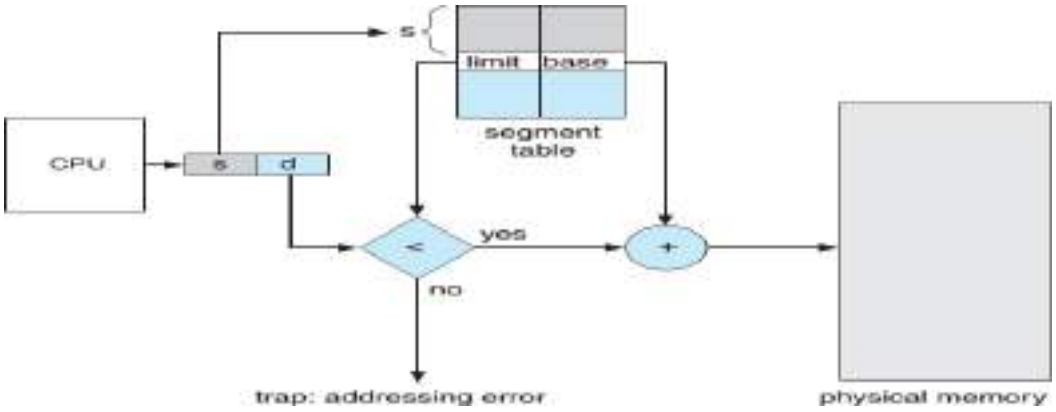
Logical address can be viewed by a programmer as a two-dimensional address and whereas actual Physical address is a one-dimensional address.

The Memory Management Unit (MMU) maps two-dimensional user-defined addresses into one-dimensional physical address.

This mapping is affected by a **Segment table**.

Each entry in the segment table has a **segment base** and a **segment limit**.

The segment base contains the starting physical address where the segment resides in memory and the segment limit specifies the length of the segment.



A logical address consists of two parts: segment number **s** and an offset into that segment d.

The segment number is used as an index to the segment table.

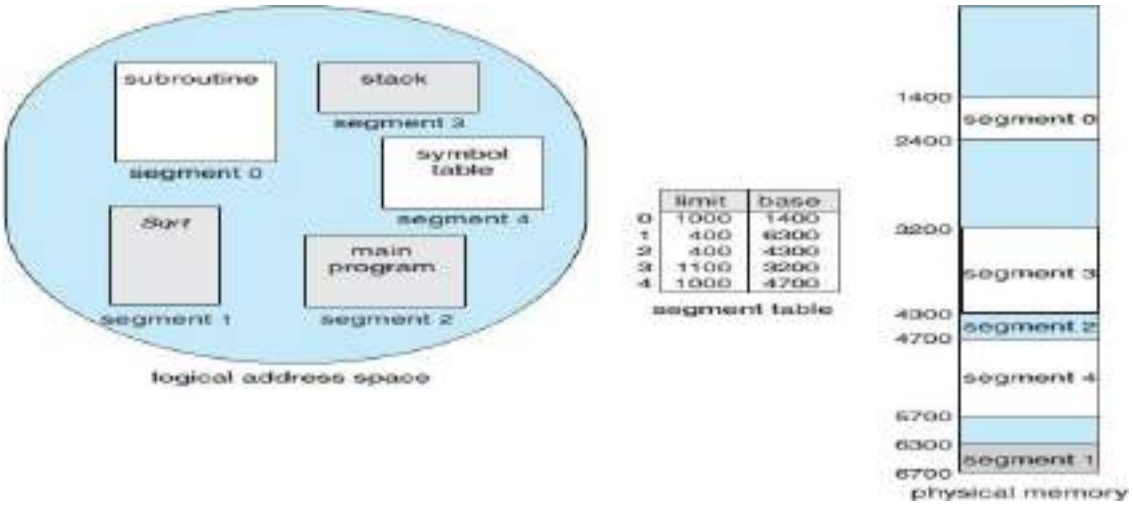
The offset d of the logical address must be between 0 and the segment limit.

When an offset is legal, it is added to the segment base to produce the address in physical memory of the desired byte.

If d>=segment limit, it is illegal then an addressing error trap will be generated that indicates logical addressing attempt beyond end of segment.

The segment table is essentially an array of base–limit register pairs.

Example: Consider the below diagram that have five segments numbered from 0 to 4. The segments are stored in physical memory.



The segment table has a separate entry for each segment, giving the beginning address of the segment in physical memory (i.e. base) and the length of that segment (i.e. limit).

1. Segment 2 is 400 bytes long and begins at location 4300. Thus, a reference to byte 53 of segment 2 is mapped onto location 4300 + 53 = 4353.
2. A reference to segment 3, byte 852 is mapped to 3200 (base of segment 3) + 852 = 4052.
3. A reference to byte 1222 of segment 0 would result in a trap to the operating system, as this segment is only 1,000 bytes long.

PAGING

Paging also permits the physical address space of a process to be noncontiguous.

Paging avoids External fragmentation and need for compaction.

Paging is implemented through cooperation between the operating system and the computer hardware.

Physical memory is divided into fixed-sized blocks called **Frames.**

Logical memory is divided into blocks of the same size called **Pages**.

Frame size is equal to the Page size.

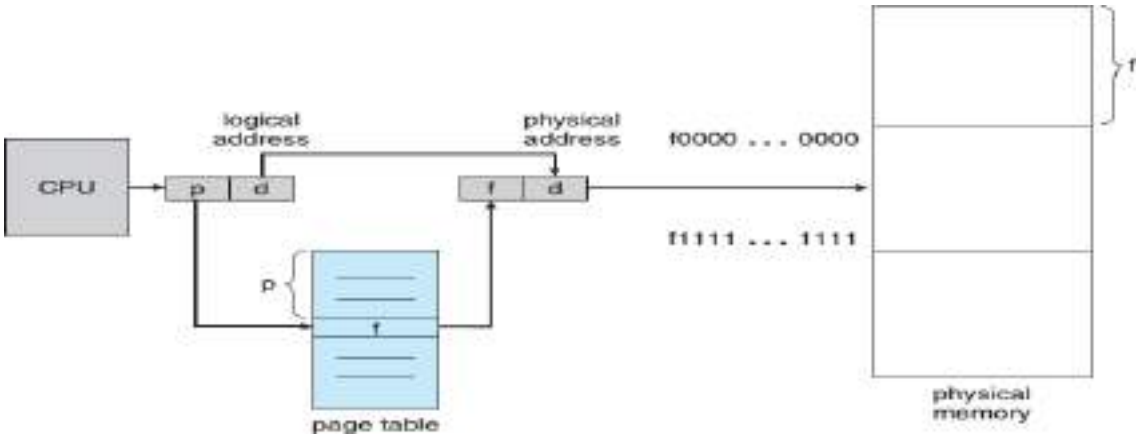
When a process is to be executed, its pages are loaded into any available memory frames from their source such as a file system or backing store.

The backing store is divided into fixed-sized blocks that are the same size as the memory frames or clusters of multiple frames.

Frame table maintains list of frames and the allocation details of the frames (i.e.) A frameis free or allocated to some page.

Each process has its own page table. When a page is loaded into main memory the corresponding page table is active in the system and all other page tables are inactive.

Page tables and Frame tables are kept in main memory. A **Page-Table Base Register (PTBR)** points to the page table.



Every address generated by the CPU is divided into two parts: a **page number (p)** and a **page-offset (d)**.

The page number is used as an index into a **Page table**. The page table contains the base address of each page in physical memory.

This base address is combined with the page offset to define the physical memory address that is sent to the memory unit.

The below shows the paging of Logical and Physical memory:

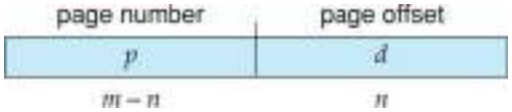
The page size is defined by the hardware. The size of a page is a power of 2.

Depending on the computer architecture the page size varies between 512 bytes and 1 GB per page.

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.

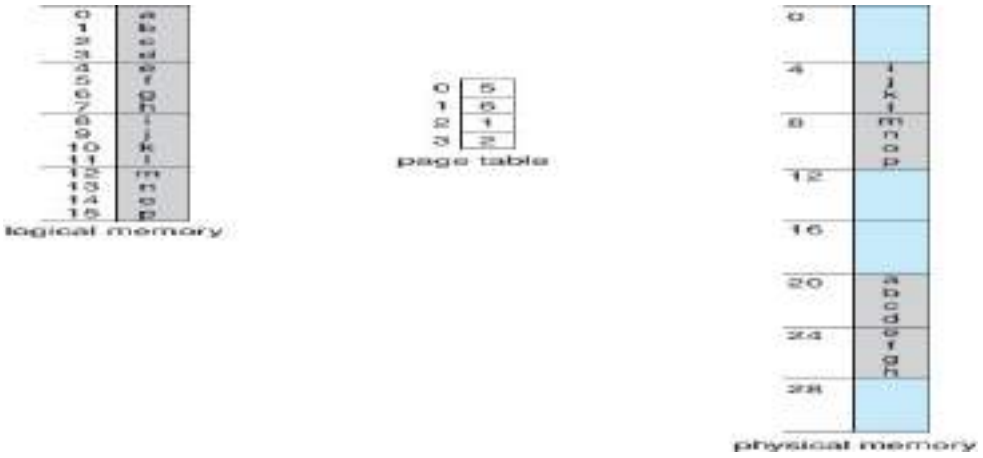
If the size of the logical address space is 2^m and a page size is 2^n bytes, then the high-order **(m–n)** bits of a logical address designate the page number and the n low-order bits designate the page offset.

The logical address contains: **p** is an index into the page table and **d** is the displacement within the page.



Example: consider the memory in the below figure where *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 programmer'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.

Paging scheme avoids external fragmentation but it creates internal fragmentation because of fixed size pages.

If page size is **2048** bytes, a process of **20489** bytes will need **10** pages plus **9** bytes.

It will be allocated **11** frames, resulting in internal fragmentation of **2048−9 = 2037** bytes.

In the worst case, a process would need ***n*** pages plus **1** byte. It would be allocated ***n*** **+ 1** frames resulting in internal fragmentation of almost an entire frame.

If the page size is small then the number of entries in page table is more this will leads to huge number of context switches.

If the page size is large then the number of entries in page table is less and the number of context switches is less.

Paging separates the programmer's view of memory and the actual physical memory

The programmer views memory as one single space, containing only this one program.

In fact, the user program is scattered throughout physical memory, which also holds other programs.

The difference between the programmer's view of memory and the actual physical memory is reconciled by the **Address-Translation Hardware**. The logical addresses are translated into physical addresses.

This mapping is hidden from the programmer and is controlled by the operating system.

User process is unable to access memory that it does not own (i.e. other process memory).

It has no way of addressing memory outside of its page table and the table includes only those pages that the process owns.

Problem: Slow access of a user memory location

If we want to access location ***i***, we must first index into the page table using the value in the PTBR offset by the page number for ***i***. This task requires a memory access.

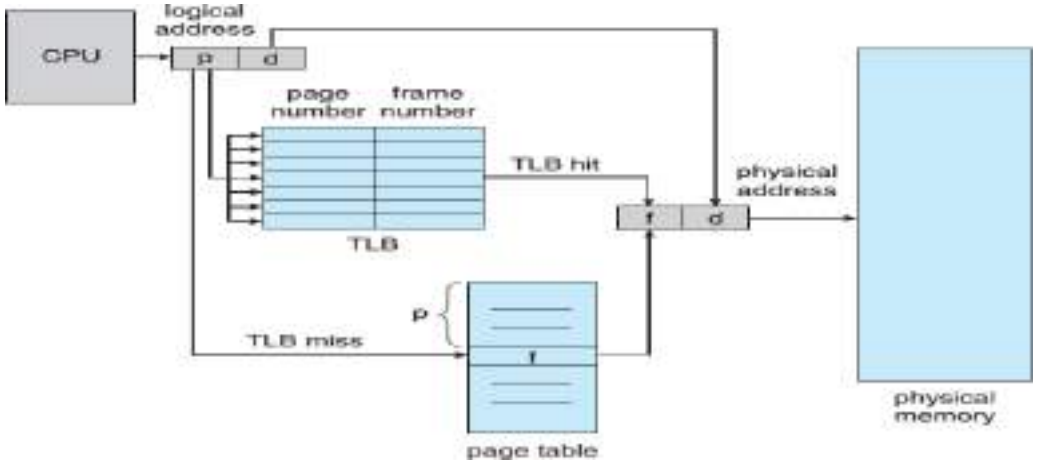
It provides us with the frame number, which is combined with the page offset to produce the actual address. We can then access the desired place in memory.

With this scheme, ***two*** memory accesses are needed to access a byte (i.e.) one for the page-table entry, one for the byte.

Thus, memory access is slowed by a factor of 2. This delay is intolerable.

Solution: Translation Look-aside Buffer (TLB)

TLB is a special, small, fast lookup hardware cache. It is associative, high-speed memory.



Each entry in the TLB consists of two parts: **a key (or tag)** and **a value**.

The size of TLB is between 32 and 1024 entries.

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.

Multiple levels of TLBs are maintained if the system is having multiple levels of Cache.

The TLB contains only a few of the page-table entries.

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

If the page number is found, its frame number is immediately available and is used to access memory. This is called **TLB Hit.**

These TLB lookup steps are executed as part of the instruction pipeline within the CPU, which does not add any performance penalty compared with a system that does not implement paging.

If the page number is not in the TLB is known as a **TLB miss**. At the time of TLB miss, a memory reference to the page table must be made.

Depending on the CPU, this may be done automatically in hardware or via an interrupt to the **OS**. When the frame number is obtained, we can use it to access memory.

Then 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, an existing entry must be selected for replacement by using any of page replacement algorithms.

Wired Down entries: These are the entries that cannot be removed from the TLB. Examples for these are **Key Kernel Code entries.**

Address Space Identifiers (ASID's) in TLB

TLBs store Address-Space Identifiers in each TLB entry.

An ASID uniquely identifies each process and is used to provide address-space protection for that process.

When the TLB attempts to resolve virtual page numbers, it ensures that the ASID for the currently running process matches the ASID associated with the virtual page.

If the ASIDs do not match, the attempt is treated as a TLB miss.

In addition to providing address-space protection, an ASID allows the TLB to contain entries for several different processes simultaneously.

If the TLB does not support separate ASIDs, then every time a new page table is selected, the TLB must be **flushed** or erased to ensure that the next executing process does not use the wrong translation information.

Otherwise, the TLB could include old entries that contain valid virtual addresses but have incorrect or invalid physical addresses left over from the previous process.

TLB Hit Ratio/ Miss Ratio

Percentage of times that the page number is found in the TLB is called the **Hit ratio**.

Percentage of times that the page number is not found in the TLB is called the **Miss ratio**.

TLB Miss ratio=1-Hit ratio

Effective Memory Access Time

It is the sum of time taken for a page to access for TLB hit ratio and TLB miss ratio. Example: An 80-percent hit ratio means that we find the desired page number in the TLB 80 percent of the time. If it takes 100 nanoseconds to access memory, then a mapped-memory access takes 100 nanoseconds when the page number is in the TLB.

If we fail to find the page number in the TLB then we must first access memory for the page table and frame number for 100 nanoseconds and then access the desired byte in memory for 100 nanoseconds with a total of 200 nanoseconds.

Effective Memory Access Time = (0.80 × 100 ns) + (0.20 × 200 ns)

= 120 nanoseconds

Memory Protection in Paging Environment

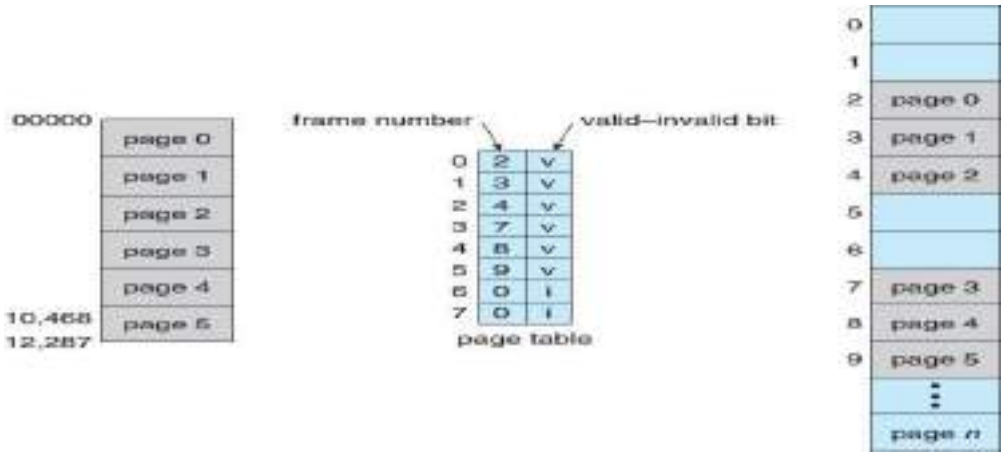
Memory protection in a paged environment is accomplished by protection bits associated with each frame. These bits are kept in the page table.

A one bit **valid–invalid** bit is attached to each entry in the page table.

When this bit is set to *valid,* the associated page is in the process's logical address space and it is 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 operating system sets this bit for each page to allow or disallow access to the page.



Consider the above figure: A system with a 14-bit address space (0 to 16383), we have a program that should use only addresses 0 to 10468. Each page size is of 2 KB.

Addresses in pages 0, 1, 2, 3, 4 and 5 are mapped normally through the page table.

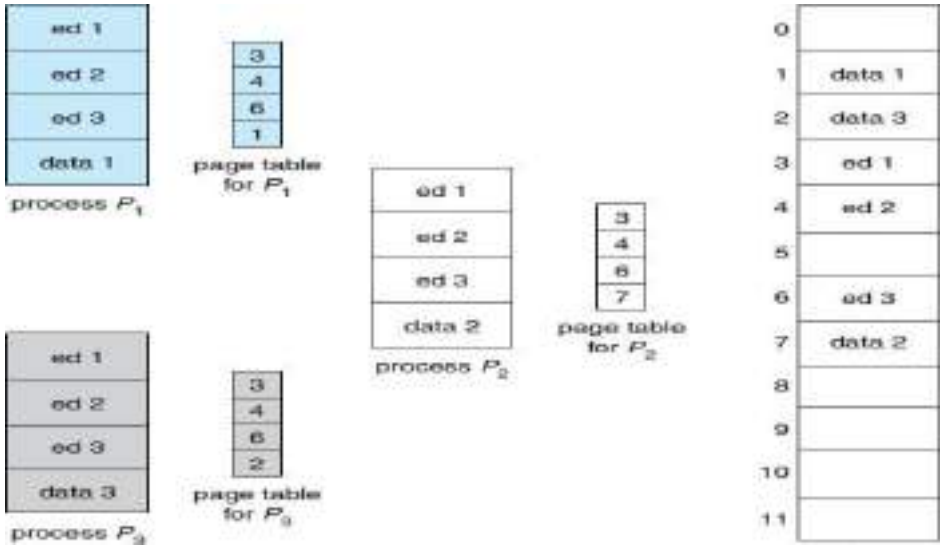
Any attempt to generate an address in pages 6 or 7 will find that the valid–invalid bit is set to invalid and the computer will trap to the operating system indicating that **Invalid** page reference.

Problem: The program extends only to address 10468, any reference beyond that address is illegal. But references to page 5 are classified as valid, so accesses to addresses up to 12287 are valid. Only the addresses from 12288 to 16383 are invalid.

Solution: To avoid this problem we use a register **Page-Table Length Register (PTLR)** to indicate the size of the page table. This PLTR value is checked against every logical address to verify that the address is in the valid range for the process.

Shared Pages

Paging has an advantage of **Sharing Common Code** This is important in Time sharing Environment.



Consider the above figure that shows 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** or **Reusable code** it can be shared.

Each process has its own data page. All three processes sharing a three-page editor each page 50 KB in size.

Reentrant code is non-self-modifying code (i.e.) 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 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 2,150 KB instead of 8,000 KB by saving 5850 KB.

Other heavily used programs can also be shared such as compilers, window systems, run-time libraries, database systems and so on.

VIRTUAL MEMORY

INTRODUCTION

Virtual Memory is a technique that allows the execution of processes that are not completely in Main-memory.

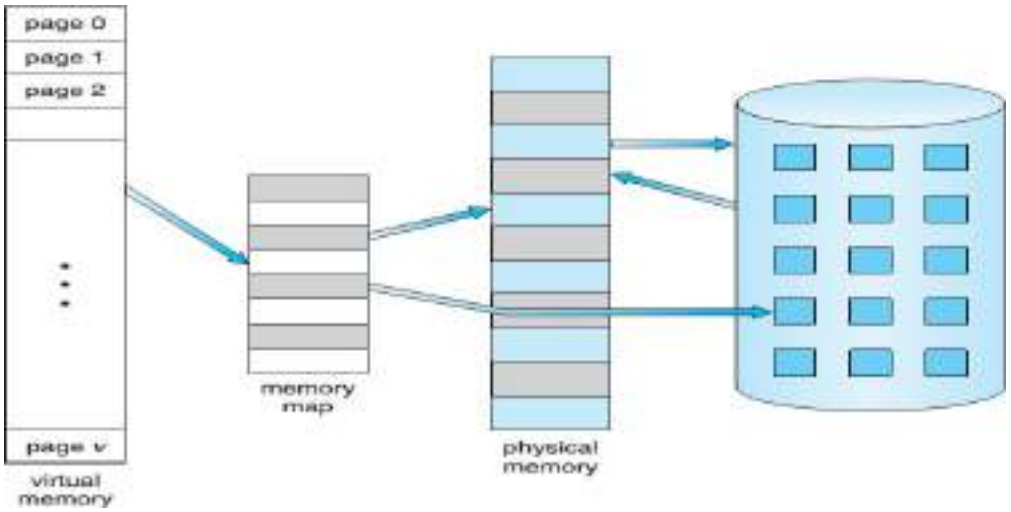
Virtual memory involves the separation of Logical Memory as perceived by users from Physical Memory.

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

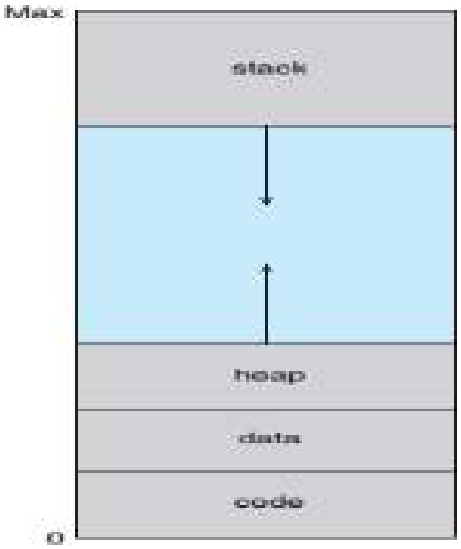
A program would no longer be constrained by the amount of physical memory that is available. Users would be able to write programs for a large ***virtual*** address space.

Because each user program could take less physical memory, more programs could be run at the same time with a corresponding increase in CPU utilization and throughput but it will not increase the Response time or Turnaround time.

Less I/O would be needed to load or swap user programs into memory, so each user program would run faster. This process will benefit both system and user.



Virtual Address Space of a process refers to the logical (or virtual) view of how a process is stored in memory. The below figure shows a virtual address space:



In the above figure, a process begins at a certain logical address (say address 0) and exists in contiguous memory.

Physical memory organized in page frames and the physical page frames assigned to a process may not be contiguous.

Memory Management Unit (**MMU**) maps logical pages to physical page frames in Main memory.

In the above figure, we allow the heap to grow upward in memory as it is used for dynamic memory allocation.

We allow for the stack to grow downward in memory through successive function calls.

The large blank space (i.e., hole) between the heap and the stack is part of the Virtual address space but will require actual physical pages only if the heap or stack grows. Virtual address spaces that include holes are known as **Sparse Address Spaces**.

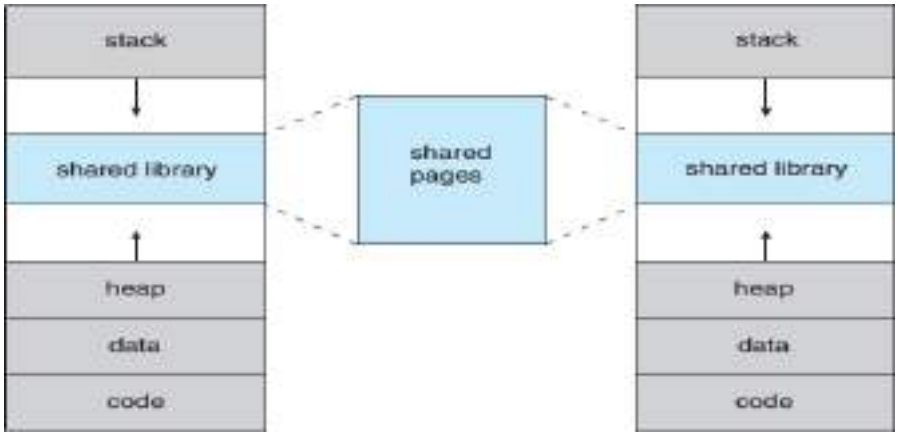
Using a Sparse Address Space is beneficial because the holes can be filled as the stack or heap segments grow or if we wish to dynamically link libraries or possibly other shared objects during program execution.

Shared Library using Virtual Memory

System libraries can be shared by several processes through mapping of the shared object into a virtual address space.

Each process considers the libraries to be part of its virtual address space, the actual pages where the libraries reside in physical memory are shared by all the processes.

A library is mapped read-only into the space of each process that is linked with it.



Two or more processes can communicate through the use of shared memory.

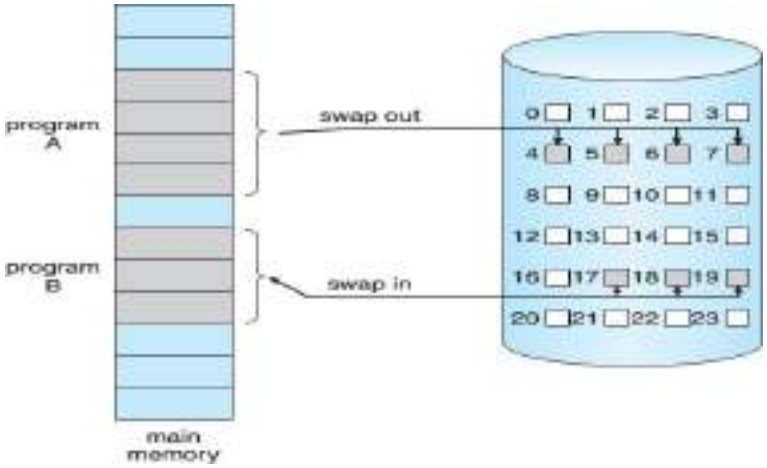
Virtual memory allows one process to create a region of memory that it can share with another process.

Processes sharing this region consider it is part of their virtual address space, yet the actual physical pages of memory are shared.

Pages can be shared during process creation with the fork( ) system call, thus speeding up process creation.

DEMAND PAGING

With Demand-paged virtual memory, pages are loaded only when they are demanded during program execution. Pages that are never accessed are never loaded into physical memory.



A demand-paging system is similar to a paging system with swapping, where processes reside in secondary memory (i.e. disk).

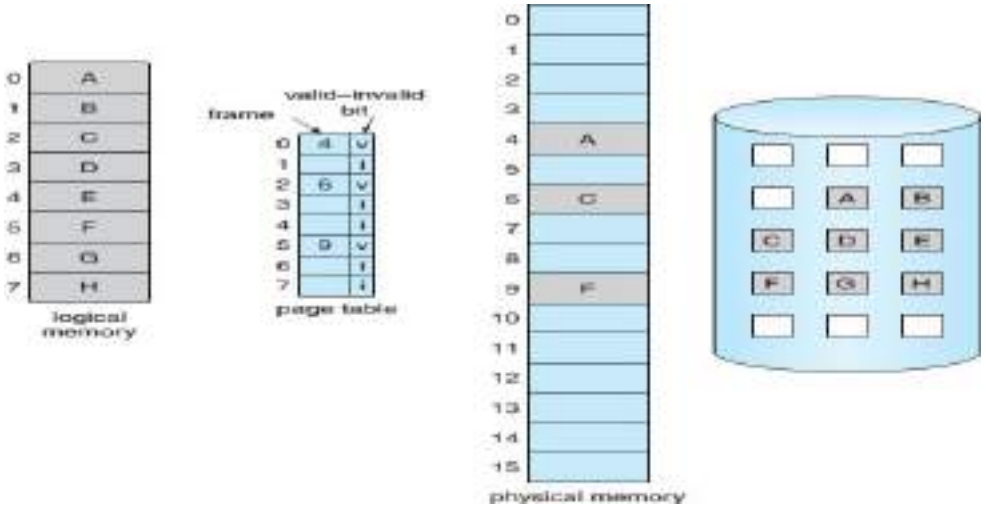
Demand paging uses the concept of **Lazy Swapper** or **Lazy Pager**. A lazy swapper or pager never swaps a page into memory unless that page will be needed.

Valid–Invalid bit is used to distinguish between the pages that are in memory and the pages that are on the disk.

When this bit is set to "valid," the associated page is both legal and is in main memory.

If the bit is set to "invalid," the page either is not valid (i.e. not in the logical address space of the process) or is valid but is currently on the disk.

The page-table entry for a page that is brought into main memory is set to valid, but the page-table entry for a page that is not currently in main memory is either marked as invalid or contains the address of the page on disk.

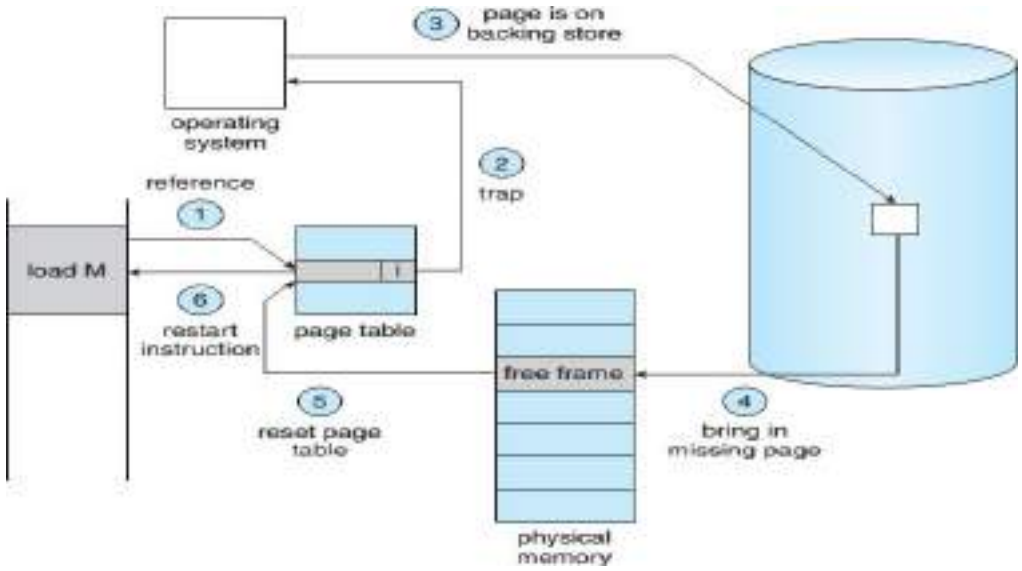


Note: The process executes and accesses pages that are **Main Memory Resident** then the execution proceeds normally.

PAGE FAULT

Access to a page marked invalid causes a **Page Fault**. The paging hardware, in translating the address through the page table will notice that the invalid bit is set that causes a trap to the operating system. This trap is the result of the operating system's failure to bring the desired page into memory.

Procedure for Handling Page Fault



1. We check an internal page table kept with the **Process Control Block** for this process to determine whether the reference was a valid or an invalid memory access.
2. If the reference was invalid, we terminate the process. If it was valid but we have not yet brought in that page, we now page it in.
3. We find a free frame (Example: by taking one from the free-frame list).
4. We schedule a disk operation to read the desired page into the newly allocated frame.
5. When the disk read is complete, we modify the internal table kept with the process and also the page table to indicate that the page is now in memory.
6. We restart the instruction that was interrupted by the trap. The process can now access the page in main memory.

Pure Demand Paging

System can start executing a process with ***no*** pages in memory.

When the operating system sets the instruction pointer to the first instruction of the process and that process is not resides in main memory then the process immediately faults for the page.

After this page is brought into memory, the process continues to execute and faulting as necessary until every page that it needs is in memory.

At that point, it can execute with no more faults. This scheme is called **Pure Demand Paging**.

Pure Demand Paging never brings a page into memory until it is required.

Hardware support for Demand Paging

The hardware to support demand paging is the same as the hardware for paging and swapping:

Page table has the ability to mark an entry as invalid through a valid–invalid bit or a special value of protection bits.

Secondary Memory holds those pages that are not present in main memory. The secondary memory is usually a high-speed disk. It is known as the swap device and the section of disk used for this purpose is known as **Swap Space**.

Performance of Demand Paging

Demand paging can significantly affect the performance of a computer system.

As long as we have no page faults, the effective access time is equal to the memory access time.

If a page fault occurs, we must first read the relevant page from disk and then access the desired word.

$$\textbf{Effective Access Time} = (1 - p) \times ma + p \times \textbf{page fault time}$$

Where ma denotes Memory Access Time

p be probability of a page fault ($0 \leq p \leq 1$). If p is closer to zero then there are few page faults.

Sequence of Steps followed by Page Fault

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 the page reference was legal and determines the location of the page on the disk.
5. 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.
6. While waiting, allocate the CPU to some other user by using CPU scheduling.
7. Receive an interrupt from the disk I/O subsystem (I/O completed).
8. Save the registers and process state for the other user.
9. Determine that the interrupt was from the disk.
10. Correct the page table and other tables to show that the desired page is now in memory.
11. Wait for the CPU to be allocated to this process again.
12. Restore the user registers, process state and new page table and then resume the interrupted instruction.

Example: With an average page-fault service time of 8 milliseconds and a memory access time of 200 nanoseconds, the effective access time in nanoseconds is:

$$\begin{aligned}\text{Effective Access Time} &= (1 - p) \times (200) + p \text{ (8 milliseconds)} \\ &= (1 - p) \times 200 + p \times 8{,}000{,}000 \\ &= 200 + 7{,}999{,}800 \times p.\end{aligned}$$

Note: The effective access time is directly proportional to the **Page-Fault rate.**

PAGE REPLACEMENT ALGORITHMS

If no frame is free, we find one that is not currently being used and free it.

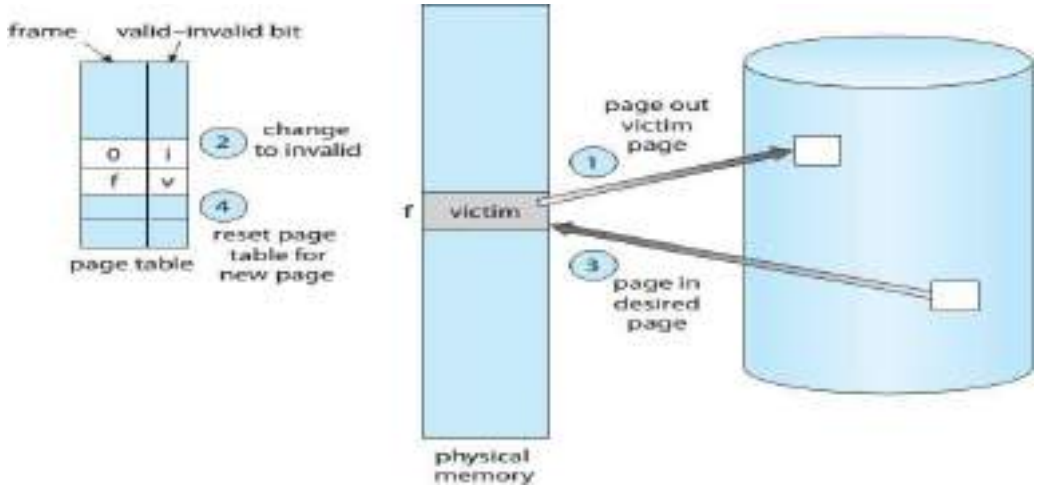
We can free a frame by writing its contents to swap space and changing the page table to indicate that the page is no longer in main memory.

We can now use the freed frame to hold the page for which the process faulted.

We modify the page-fault service routine to include Page Replacement:

1. Find the location of the desired page on the disk.
2. Find a free frame:
    a. If there is a free frame, use it.
    b. If there is no free frame, use a Page-Replacement algorithm to select a **Victim frame**.
    c. Write the victim frame to the disk and change the page table and frame table.

3. Read the desired page into newly freed frame and change the page table and frame table.
4. Continue the user process from where the page fault occurred.



Modify bit or Dirty bit

Each page or frame has a modify bit associated with it in the hardware.

The modify bit for a page is set by the hardware whenever any byte in the page has been modified.

When we select a page for replacement, we examine its modify bit.

If the bit is set, the page has been modified since it was **read in** from the disk. Hence we must write the page to the disk.

If the modify bit is not set, the page has ***not*** been modified since it was read into memory. Hence there is no need for write the memory page to the disk, because it is already there.

Note: To determine the number of page faults for a particular reference string and page-replacement algorithm, we also need to know the number of page frames available. As the number of frames available increases, the number of page faults decreases.

There is several Page Replacement Algorithms are in use:

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

<u>First-In-First-Out Page Replacement Algorithm</u>

FIFO algorithm associates with time of each page when it was brought into main memory.

When a page must be replaced, the oldest page is chosen.

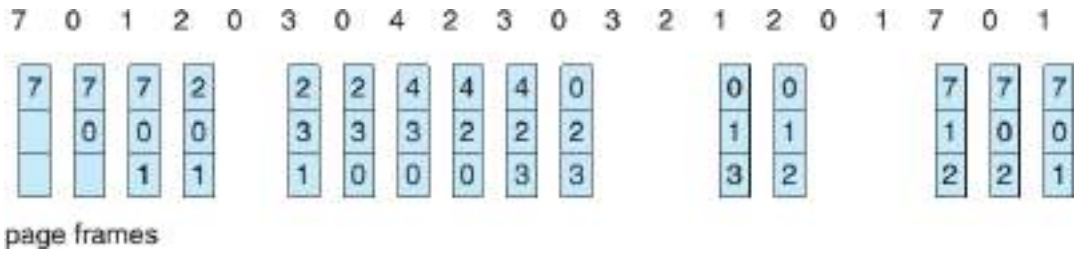
We can create a FIFO queue to hold all pages in memory.

We replace the page at the **Head** of the queue.

When a page is brought into memory, we insert it at the tail of the queue.

<u>Example</u>: Consider the below reference string and memory with three frames

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



First **3-references** (7, 0, 1) cause **3-Page faults** and are brought into these empty frames.

The next reference (2) replaces page 7, because page 7 was brought in first.

Since 0 is the next reference and 0 is already in memory, we have no fault for this reference.

The first reference to 3 results in replacement of page 0, since it is now first in line. Because of this replacement, the next reference to 0, will fault. Page 1 is then replaced by page 0.

By the end, there are **Fifteen-page** faults altogether.

Problem: Belady's Anomaly

Belady's Anomaly states that: the page-fault rate may ***increase*** as the number of allocated frames increases. Researchers identifies that Belady's anomaly is solved by using Optimal Replacement algorithm.

Optimal Page Replacement Algorithm (OPT Algorithm)

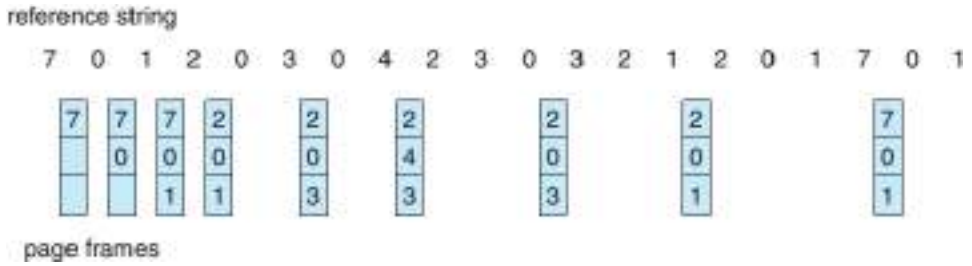
It will never suffer from Belady's anomaly.

OPT states that: Replace the page that will not be used for the longest period of time.

OPT guarantees the lowest possible page fault rate for a fixed number of frames.

Example: Consider the below reference string and memory with three frames

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



The first **3-references** cause faults that fill the **3-empty** frames.

The reference to page 2 replaces page 7, because page 7 will not be used until reference number 18, whereas page 0 will be used at 5 and page 1 at 14.

The reference to page 3 replaces page 1 because page 1 will be the last of the three pages in memory to be referenced again.

At the end there are only **9-page faults** by using optimal replacement algorithm which is much better than a FIFO algorithm with **15-page faults**.

Note: No replacement algorithm can process this reference string in **3-frames** with fewer than **9-faults**.

Problem with Optimal Page Replacement algorithm

The optimal page-replacement algorithm is difficult to implement, because it requires future knowledge of the reference string. The optimal algorithm is used mainly for comparison studies (i.e. performance studies).

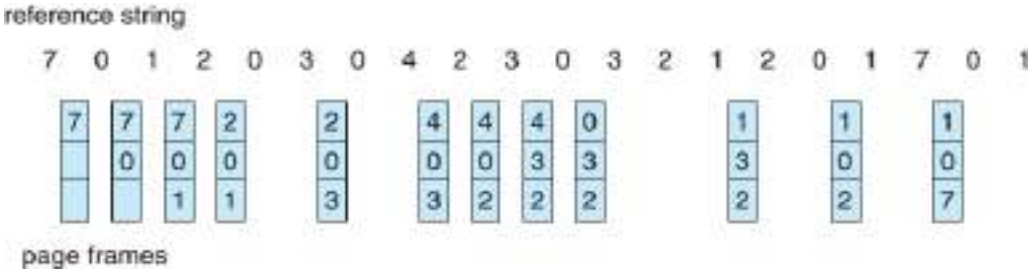
LRU Page Replacement Algorithm

In LRU algorithm, the page that has **not** been used for the **longest** period of time will be replaced (i.e.) we are using the recent past as an approximation of the near future.

LRU replacement associates with each page the time of that page's last use.

Example: Consider the below reference string and memory with three frames

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



The first five faults are the same as those for optimal replacement.

When the reference to page 4 occurs LRU replacement sees that out of the three frames in memory, page 2 was used least recently.

Thus, the LRU algorithm replaces page 2, not knowing that page 2 is about to be used.

When it then faults for page 2, the LRU algorithm replaces page 3, since it is now the least recently used of the three pages in memory.

The total number of page faults with LRU is 12 which is less as compared to FIFO.

LRU can be implemented in two ways: **1. Counters** and **2. Stack**

Counters

Each page-table entry is associated with a time-of-use field and add to the CPU a logical clock or counter. The clock is incremented for every memory reference.

Whenever a reference to a page is made, the contents of the clock register are copied to the time-of-use field in the page-table entry for that page.

We replace the page with the smallest time value. This scheme requires a search of the page table to find the LRU page and a write to memory to the time-of-use field in the page table for each memory access.

The times must also be maintained when page tables are changed due to CPU scheduling.

Stack

LRU replacement is implemented by keeping a stack of page numbers.

Whenever a page is referenced, it is removed from the stack and put on the top.

In this way, the most recently used page is always at the top of the stack and the least recently used page is always at the bottom.

Because entries must be removed from the middle of the stack, it is best to implement this approach by using a doubly linked list with a Head pointer and a Tail pointer.

Removing a page and putting it on the top of the stack then requires changing six pointers at worst.

Each update is a little more expensive, but there is no search for a replacement.

The tail pointer points to the bottom of the stack, which is the LRU page.

This approach is particularly appropriate for software or microcode implementations of LRU replacement.

CHAPTER – 2 - FILE SYSTEM INTERFACE

FILE

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

A file is the smallest allotment of logical secondary storage (i.e.) data cannot be written to secondary storage unless they are within a file.

Files represent programs and data. Data files may be numeric, alphanumeric or binary.

The information in a file is defined by its creator.

Different types of information may be stored in a file such as source or executable programs, numeric or text data, photos, music, video and so on.

A file structure depends on its type:

Text file is a sequence of characters organized into lines.

Source file is a sequence of functions, each of which is further organized as declarations followed by executable statements.

Executable file is a series of code sections that the loader can bring into memory and execute.

File Attributes

A file is referred to by its name. The following are the list of file attributes:

Name: The symbolic file name is the only information kept in human-readable form.

Identifier: This is a unique number that identifies the file within the file system. It is the non-human-readable name for the file.

Type: This information is needed for systems that support different types of files.

Location: It is a pointer to a device and to the location of the file on that device.

Size: The current size of the file and the maximum size are included in this attribute.

Protection: It is access-control information determines who can do reading, writing, executing and so on.

Time, Date and User Identification: This information kept for creation, last modification and last use. These data can be useful for protection, security and usage monitoring.

Directory structure keeps information about all files. It resides on secondary storage.

A directory entry consists of the file's name and its unique identifier.

The identifier locates the other file attributes.

It may take more than a kilobyte to record this information for each file.

File Operations

There are 6 basic operations performed on file and corresponding System call are:

1. **Creating a file: create ()** system call is used to create a file. To create a file Operating system checks whether there is enough space in the system. If yes, then a new entry will be made in the directory structure.
2. **Repositioning within a file**. The directory is searched for the appropriate entry and the current-file-position pointer is repositioned to a given value. Repositioning within a file need not involve any actual I/O. This file operation is also known as a **File-Seek**.

3. **Deleting a file: delete( )** system call is used to delete a file. To deete a file, we search the directory for the named file. If we found the associated directory entry, we release all file space and erase the directory entry.
4. **Truncating a file**. The user erases all the contents of a file but keep its attributes. The length of the file will be reset to zero.
5. **Writing a file**. write( ) system call is used to write a file. It specifies both the name of the file and the information to be written to the file. The system searches the filename in the directory to find the file's location.
6. **Reading a file**. read( ) system call is used to read from a file. It specifies the name of the file and where the next block of the file should be put. The directory is searched for the associated entry.

Note: A process is usually either reading from or writing to a file, the current operation location can be kept as a per-process **Current-File-Position Pointer**. Both the read and write operations use this same pointer.

open( ) and close( ) system calls

System calls open( ) and close( ) are used to open and to close a file respectively.
OS maintains information about all open files in **Open-File Table.**
When a file operation is requested, the file is indexed into Open File Table.
When a file is closed by a process then the OS removes its entry from the open-file table.

Operating system uses **2-levels** of Internal tables:

1. **Per-Process Table** The per-process table tracks all files that a process has open. This table stores information regarding the process's use of the file. Each entry in the per-process table in turn points to a system-wide open-file table.
2. **System-Wide Table:** It contains process-independent information, such as the location of the file on disk, access dates and file size. Once a file has been opened by one process, the system-wide table includes an entry for the file. When another process executes an open( ) call, a new entry is added to the process's open-file table pointing to the appropriate entry in the system-wide table.

An open file is associated with following information:

File pointer: This pointer is unique to each process operating on the file. It must be kept separate from the on-disk file attributes. On systems that do not include a file offset as part of the read( ) and write( ) system calls, the system must track the last read– write location as a current-file-position pointer.

File-open count: The open-file table also has an **open count** associated with each file to indicate how many processes have opened that file. Each close( ) decreases the open count. When the open count reaches zero, the file is no longer in use and the file's entry is removed from the open-file table.

Disk location of the file: Most file operations require the system to modify data several times within the file. The information needed to locate the file on disk is kept in main-memory so that the system does not have to read it from disk for each operation.

Access rights: Each process opens a file in an access mode. This information is stored on the per-process table so the operating system can allow or deny subsequent I/O requests.

File Types

File types are generally included as part of file name. The file name is split into two parts: a name and an extension usually separated by a dot.

The system uses the extension to indicate the type of the file and the type of operations that can be done on that file.

Example: resume.docx, server.c and ReaderThread.cpp.

The below table shows the common file types:

File Type	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

Internal File Structure

Locating an offset within a file can be complicated for the operating system.

Disk systems have a well-defined block size determined by the size of a sector.

All disk I/O is performed in units of one block (physical record). All blocks are the same size.

Problem: It is unlikely that the physical record size will exactly match the length of the desired logical record. Logical records may have different lengths.

Solution: Packing a number of logical records into physical blocks solves this problem.

Example: The UNIX operating system defines all files to be streams of bytes.

Each byte is individually addressable by its offset from the beginning of the file.

Logical record size, physical block size and packing technique determine how many logical records are in each physical block.

Packing can be done either by the user's application program or by the operating system.

Note: All file systems suffer from internal fragmentation. The larger the block size, the greater the internal fragmentation.

ACCESS METHODS

Files store information. When a file is used, this information must be accessed and read into computer memory. The information in the file can be accessed in several ways:

1. Sequential Access
2. Direct Access
3. Indexed Access

Sequential Access

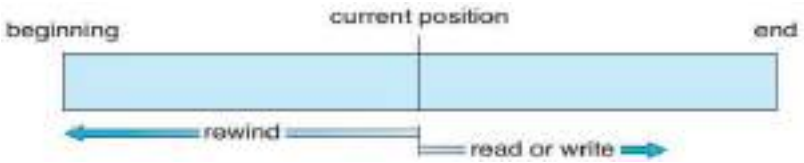
Information in the file is processed in order, one record after the other record.

Example: editors and compilers usually access files in sequential order.

Reads and writes make up the bulk of operations on a file:

read_next( ) operation reads the next portion of the file and automatically advances a file pointer, which tracks the I/O location.

write_next( ) operation appends to the end of the file and advances to the end of the newly written material.



Sequential access is based on a tape model of a file and works on sequential-access devices.

Direct Access or Relative Access

In direct access method, the file is viewed as a numbered sequence of blocks or records.

There are no restrictions on the order of reading or writing for a direct-access file.

Thus, we may read block 14, then read block 53 and then write block 7.

A file is made up of fixed-length **logical records** that allow programs to read and write records rapidly in no particular order.

The direct-access method is based on a disk model of a file, since disks allow random access to any file block.

Databases are direct access type. When a query concerning a particular **subject** arrives,

we compute which block contains the **answer** and then read that block directly to provide the desired information.

Example: Airline-reservation system

We might store all the information about a particular flight 713 in the block identified by the flight number.

The number of available seats for flight 713 is stored in block 713 of the reservation file.

To store information about a larger set, such as people, we might compute a hash function on the people's names to determine a block to read and search.

File operation in Direct Access Method

read(n) and write(n) are the read and write operation performed in Direct Access method where **n** represent the block number.

The block number provided by the user to the operating system is a **Relative Block Number**.

A relative block number is an index relative to the beginning of the file.

Thus, the first relative block of the file is 0, the next is 1 and so on.

Relative block numbers allow the **OS** to decide where the file should be placed and helps to prevent user from accessing portions of the file system that may not be part of its file.

Indexed Access

The **index** is like an index in the back of a book that contains pointers to the various blocks.

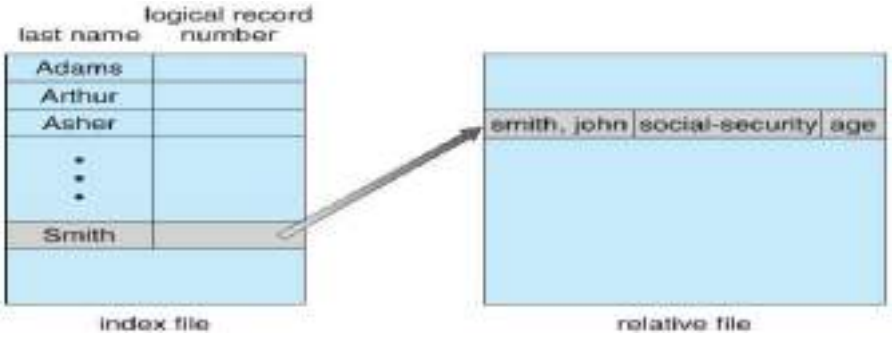
To find a record in the file, we first search the index and then use the pointer to access the file directly and to find the desired record.

To find a record we can make a binary search of the index. This search helps us to know exactly which block contains the desired record and access that block.

This structure allows us to search a large file doing little I/O.

With large files, the index file itself may become too large to be kept in memory.

One solution is to create an index for the index file. The primary index file contains pointers to secondary index files, which point to the actual data items.



DIRECTORY STRUCTURE

Files are stored on random-access storage devices such as Hard-disks, Optical-disks and Solid-state disks.

A storage device can be used for a file system. It can be subdivided for finer-grained control.

Ex: A disk can be **partitioned** into quarters. Each quarter can hold a separate file system.

Partitioning is useful for limiting the sizes of individual file systems, putting multiple file-system types on the same device or leaving part of the device available for other uses, such as swap space or unformatted (raw) disk space.

A file system can be created on each of these disk partitions. Any entity containing a file system is generally known as a **Volume**.

Volume may be a subset of a device, a whole device. Each volume can be thought of as a virtual disk.

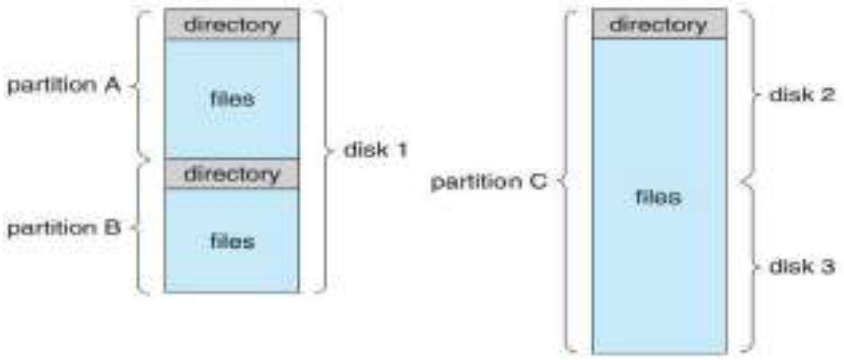
Volumes can also store multiple operating systems. Volumes allow a system to boot and run more than one operating system.

Each volume contains a file system maintains information about the files in the system.

This information is kept in entries in a **Device directory** or **Volume table of contents**.

The device directory (**directory**) records information such as name, location, size and type for all files on that volume.

The below figure shows the typical file system organization:



Storage Structure in Solaris OS

The file systems of computers can be extensive. Even within a file system, it is useful to

segregate files into groups and manage those groups. This organization involves the use of directories.

Consider the types of file systems in the Solaris Operating system:

Tmpfs is a ―temporary‖ file system that is created in volatile main memory and has its contents erased if the system reboots or crashes

objfs is a ―virtual‖ file system that gives debuggers access to kernel symbols

ctfs is a virtual file system that maintains ―contract‖ information to manage which processes start when the system boots and must continue to run during operation

lofs is a ―loop back‖ file system that allows one file system to be accessed in place of another file system.

procfs is a virtual file system that presents information on all processes as a file system

ufs, zfs are general-purpose file systems.

Operations on Directory

Different operations performed on a directory are:

Search for a file. This operation searches a directory structure to find the entry for a particular file. It finds all files whose names match with a particular pattern.

Create a file. When a new file is created, its entry is added to the directory.

Delete a file. When a file is no longer needed, we can remove it from the directory.

List a directory. We need to be able to list the files in a directory and the contents of the directory entry for each file in the list.

Rename a file. A file can be renamed, when the contents or use of the file changes (i.e.) csec.txt to cse.txt or cse.txt to cse.c file etc.

Traverse the file system. We may wish to access every directory and every file within a directory structure.

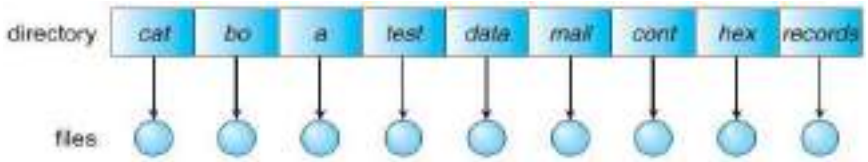
LOGICAL STRUCTURE OF A DIRECTORY

The different directory structures are:

1. Single Level Directory
2. Two-Level Directory
3. Tree Structured Directory
4. Acyclic-Graph Directories
5. General Graph Directory

Single-Level Directory

In Single-Level Directory structure, all files are contained in the same directory.



A single-level directory has significant limitations that when the number of files increases or when the system has more than one user, all files are in the same directory, they must have unique names.

Even a single user on a single-level directory may find it difficult to remember the names of all the files as the number of files increases.

It is common for a user to have hundreds of files on one computer system.

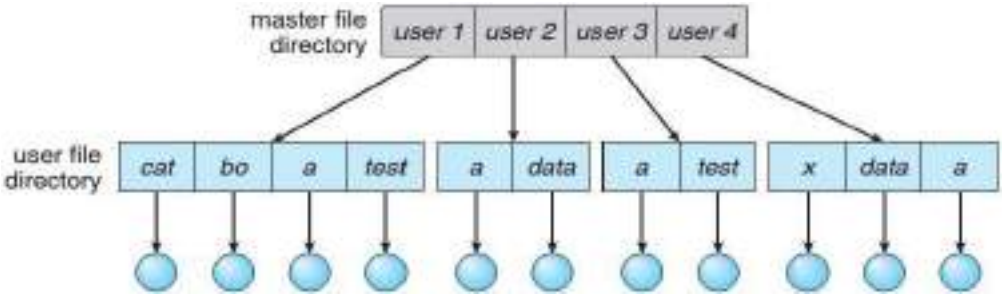
Keeping track of so many files is a difficult task.

Two-Level Directory

In the two-level directory structure, each user has his own **user file directory (UFD)**.

Each UFD lists only the files of a single user.

When a user job starts or a user log in, the system's **Master File Directory (MFD)** is searched. MFD is indexed by user name or account number and each entry points to the UFD for that user.



When a user refers to a particular file, only his own UFD is searched. Thus, different users may have files with same name as long as all the file names within each UFD are unique.

To create a file for a user, the operating system searches only that user's UFD to check whether another file of that name exists.

To delete a file, the operating system confines its search to the local UFD; thus, it cannot accidentally delete another user's file that has the same name.

This way the two-level directory structure solves the name-collision problem.

To name a particular file uniquely in a two-level directory, we must give both the user name and the file name.

A two-level directory can be thought of as a tree or an inverted tree, of height 2.

The root of the tree is the MFD.

MFD's direct descendants are the UFDs.

The descendants of the UFDs are the files.

The files are the leaves of the tree.

Specifying a user name and a file name defines a path in the tree from the root (MFD) to a leaf (a file).

A user name and a file name define a **path name**.

Every file in the system has a path name.

To name a file uniquely, a user must know the path name of desired file.

The user directories themselves must be created and deleted as necessary.

A special system program is run with the appropriate user name and account information.

The program creates a new UFD and adds an entry in the MFD.

The execution of this program might be restricted to system administrators.

Disadvantages:

This structure effectively isolates one user from another.

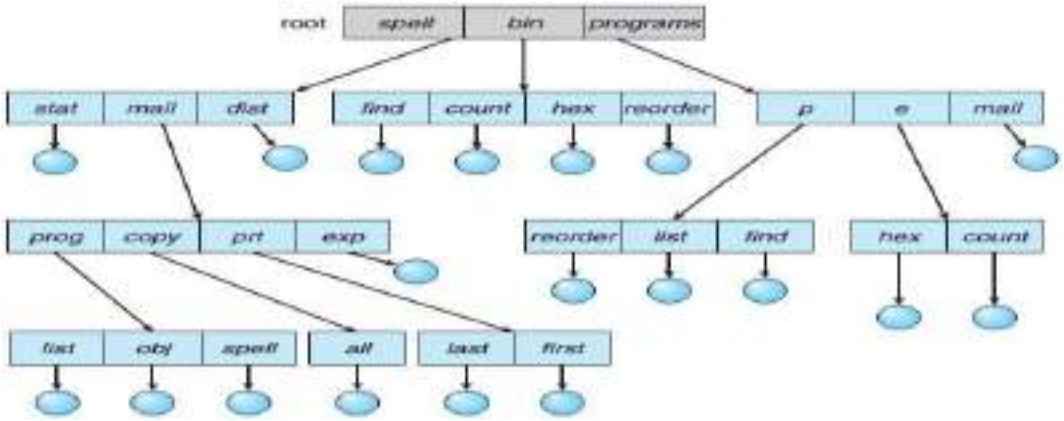
Isolation is an advantage when the users are completely independent but it is a disadvantage when the users want to cooperate on some task and to access others files.

Some systems simply do not allow one local user files to be accessed by other users.

Tree-Structured Directories

Tree Structure allows users to create their own subdirectories and to organize their files

accordingly.



The tree has a root directory and every file in the system has a unique path name.

A directory (or) subdirectory contains a set of files or subdirectories.

A directory is simply another file, but it is treated in a special way. All directories have the same internal format.

One bit in each directory entry defines the entry as a file (0) or as a subdirectory (1).

Each process has a current directory. The **current directory** should contain most of the files that are of current interest to the process.

When reference is made to a file, the current directory is searched.

If a file is needed that is not in the current directory, then the user must specify a path name or change the current directory to the directory holding that file.

To change directories, a system call is provided that takes a directory name as a parameter and uses it to redefine the current directory.

Thus, the user can change the current directory whenever the user wants.

Path names can be of **2-types:** Absolute and Relative path name.

1. An **absolute path name** begins at the root and follows a path down to the specified file, giving the directory names on the path.
2. A **relative path name** defines a path from the current directory.

Example: If the current directory is **root/spell/mail** then the relative path name **prt/first** refers to the same file as does the absolute path name **root/spell/mail/prt/first.**

Deletion of Directory

If a directory is empty, its entry will be deleted form corresponding the directory.

If the directory to be deleted is not empty but contains several files or subdirectories then one of the two approaches can be followed:

1. Some systems will not delete a directory unless it is empty. Thus, to delete a directory, the user must first delete all the files in that directory.
2. When a request is made to delete a directory, all the directory's files and subdirectories

are also to be deleted. Example: UNIX rm command used for this purpose.

Note: With a tree-structured directory system, users can be allowed to access the files of other users. Example: user B can access a file of user A by specifying its path names.

Acyclic-Graph Directories

The acyclic graph is a natural generalization of the tree-structured directory scheme.

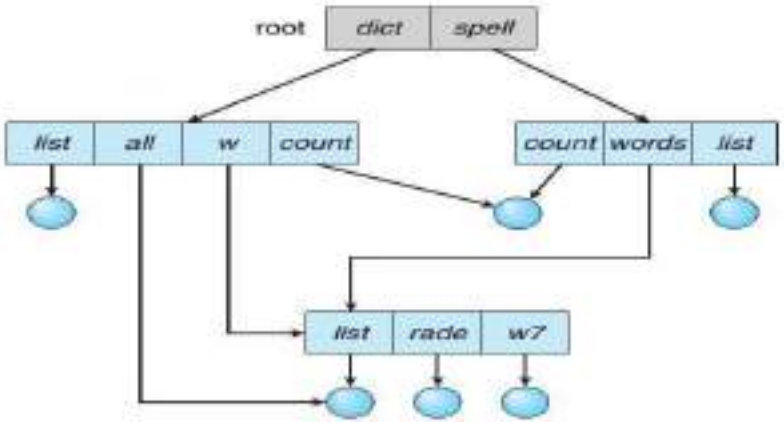
A tree structure prohibits the sharing of files or directories.

An **acyclic graph** is a graph with no cycles allows directories to share subdirectories and files. The same file or subdirectory may be in two different directories.

With a shared file, only one actual file exists, so any changes made by one person are immediately visible to the other.

Sharing is particularly important for subdirectories; a new file created by one person will automatically appear in all the shared subdirectories.

UNIX implements Shared files and subdirectories by creating a new directory entry called a Link. A **link** is effectively a pointer to another file or subdirectory.



Example: A link may be implemented as an absolute or a relative path name.

When a reference to a file is made, we search the directory.

If the directory entry is marked as a link, then the name of the real file is included in the link information.

We **resolve** the link by using that path name to locate the real file.

Links are easily identified by their format in the directory entry and Links are effectively indirect pointers.

The operating system ignores these links when traversing directory trees to preserve the acyclic structure of the system.

Problems with Acyclic-Graph Directories

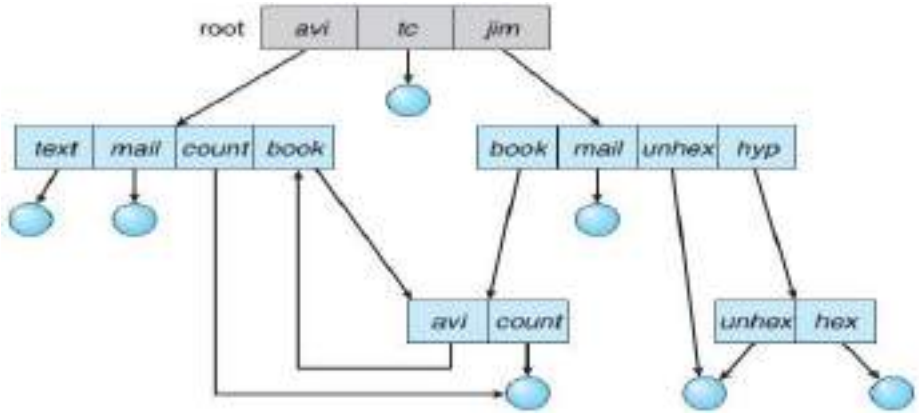
1. A file may now have multiple absolute path names. Consequently, distinct file names may refer to the same file.
2. Deletion of shared file is problematic. Because more than one user is using the file if one user deletes a shared file, it may leave dangling pointers to non-existence file for other users.

General Graph Directory

General Graph Directory structure is an acyclic graph with Cycles.

A problem with using an acyclic-graph structure is ensuring that there are no cycles.

In tree-structure directory we can add new files and subdirectories to an existing tree-structured directory preserves the tree-structured nature but if we add links, the tree structure is destroyed, resulting in a simple graph structure.



The primary advantage of an acyclic graph is the relative simplicity of the algorithms to traverse the graph and to determine when there are no more references to a file.

If cycles are allowed to exist in the directory, we likewise want to avoid searching any component twice, for reasons of correctness as well as performance.

Problem: A poorly designed algorithm might result in an infinite loop continually searching through the cycle and never terminating.

Solution: we can limit arbitrarily the number of directories that will be accessed during a search.

PROTECTION

The information is stored on the computer system. Protection deals with issue of improper access of information to the illegitimate users.

Protection provides controlled access by limiting the types of file access that can be made.

Several different types of operations may be controlled:

Read. Read from the file.

Write. Write or rewrite the file.

Execute. Load the file into memory and execute it.

Append. Write new information at the end of the file.

Delete. Delete the file and free its space for possible reuse.

List. List the name and attributes of the file.

Renaming, copying and editing the file, may also be controlled.

Access Control

Different users may need different types of access to a file or directory.

Systems uses A**ccess-control list (ACL) Scheme that** specifies user names and the types of access allowed for each user.

When a user requests access to a particular file, the operating system checks the access list associated with that file.

If that user is listed for the requested access, the access is allowed. Otherwise, a protection violation occurs and the user job is denied access to the file.

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 work group.
Universe: All other users in the system constitute the universe.

Protection in UNIX

In the UNIX system, groups can be created and modified only by the manager or super user.

With the more limited protection classification, only three fields are needed to define protection.

Each field is a collection of bits and each bit either allows or prevents the access associated with it.

The UNIX system defines three fields of 3 bits each—rwx, where **r** controls read access, **w** controls write access and **x** controls execution.

A separate field is kept for the file owner, for the file's group and for all other users.

In this scheme, 9 bits per file are needed to record protection information.

FILE-SYSTEM STRUCTURE

File systems are maintained on Secondary Storage Disks.

Two reasons for storing file systems on disk are:

1. A disk can be rewritten in place (i.e.) It is possible to read a block from the disk, modify the block and write it back into the same place.
2. A disk can access directly any block of information it contains. Thus, it is simple to access any file either sequentially or randomly and switching from one file to another requires only moving the read–write heads and waiting for the disk to rotate.

I/O transfers between memory and disk are performed in units of **blocks**.

Each block has one or more sectors.

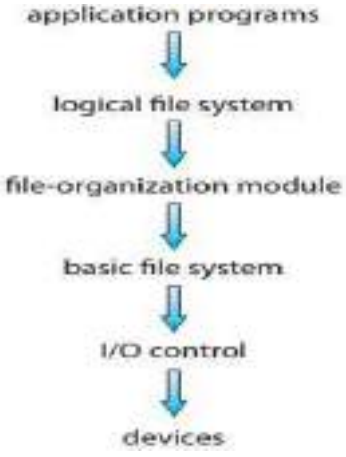
Depending on the disk drive, sector size varies from 32 bytes to 4,096 bytes; the usual size is 512 bytes.

File systems provide efficient and convenient access to the disk by allowing data to be stored, located and retrieved easily.

Design issues of File System

1. Defining how the file system should look to the user. This task involves defining a file and its attributes, the operations allowed on a file and the directory structure for organizing files.
2. Creating algorithms and data structures to map the logical file system onto the physical secondary-storage devices.

Layered Structured File System



I/O control level consists of device drivers and interrupt handlers to transfer information between the main memory and the disk system.

Basic File System needs only to issue generic commands to the appropriate device driver to read and write physical blocks on the disk.

Each physical block is identified by its numeric disk address

Example: drive 1, cylinder 73, track 2, sector 10.

Basic file system layer also manages the memory buffers and caches that hold various file-system, directory and data blocks.

F**ile-Organization Module** knows about files and their logical blocks, as well as physical blocks.

The file-organization module includes the free-space manager, which tracks unallocated blocks and provides these blocks to the file-organization module when requested.

Logical File System manages metadata information. Metadata includes all of the file-system structure except the actual data (or) contents of the files.

Logical File System maintains file structure via File-Control Blocks. A **File-Control Block (FCB)** contains information about the file, including ownership, permissions and location of the file contents. In UNIX FCB is called as an **inode**.

The logical file system is also responsible for protection.

Advantage: Layered structure minimizes the duplication of code. Code reusability is possible with this structure.

Disadvantage: Layering can introduce more operating-system overhead, which may result in decreased performance.

File systems supported by different Operating systems

1. **UNIX** uses the **UNIX File System (UFS)** is based on Berkeley Fast File System (FFS).
2. **Windows** supports disk file-system formats of **FAT**, **FAT32** and **NTFS** as well as CD-ROM and DVD file-system formats.
3. **Standard Linux file system** is known as the **Extended File System**, with the most common versions being ext3 and ext4.

FILE SYSTEM IMPLEMENTATION

Several on-disk and in-memory structures are used to implement a file system. These structures vary depending on the operating system and the file system.

On-Disk Structure

The file system may contain information about how to boot an operating system stored on disk, the total number of blocks, the number and location of free blocks, the directory structure and individual files.

Several On-Disk structures are given below:

Boot Control Block

A Boot Control Block (per volume) can contain information needed by the system to boot an operating system from that volume.

If the disk does not contain an operating system, this block can be empty.

It is typically the first block of a volume.

In UFS, it is called the **Boot Block**. In NTFS, it is the **Partition Boot Sector**.

Volume Control Block

A Volume Control Block (per volume) contains volume (or) partition details such as the number of blocks in the partition, the size of the blocks, a free-block count and free-block pointers and a free-FCB count and FCB pointers.

In UFS, this is called a **Super-Block**. In NTFS, it is stored in the **Master File Table**.

Directory Structure

A directory structure (per file system) is used to organize the files.

In UFS, this includes file names and associated inode numbers.

In NTFS, it is stored in the master file table.

A per-file FCB contains many details about the file.

It has a unique identifier number to allow association with a directory entry.

In NTFS, this information is actually stored within the master file table, which uses a relational database structure, with a row per file.

The in-memory information is used for both file-system management and performance improvement via caching. The data are loaded at mount time, updated during file-system operations and discarded at dismount.

Several in-memory structures are given below:

An in-memory **mount table** contains information about each mounted volume.

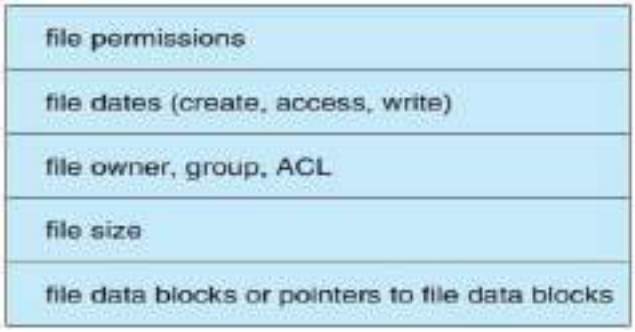
An in-memory directory-structure cache holds the directory information of recently accessed directories. For directories at which volumes are mounted, it can contain a pointer to the volume table.

The **system-wide open-file table** contains a copy of the FCB of each open file, as well as other information.

The **per-process open-file table** contains a pointer to the appropriate entry in the system-wide open-file table, as well as other information.

Buffers hold file-system blocks when they are being read from disk or written to disk.

The below figure shows the FCB



To create a new file, an application program calls the logical file system.

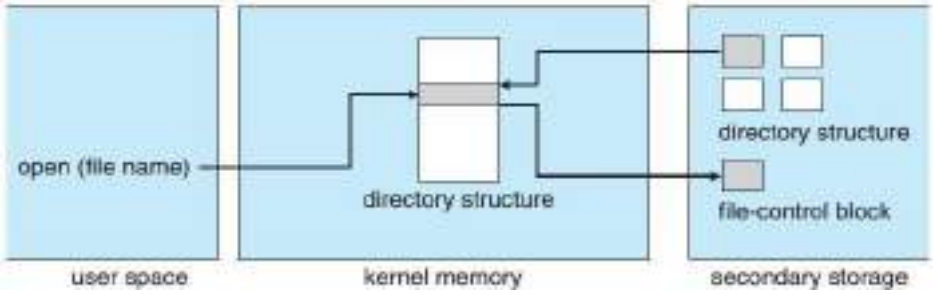
The logical file system knows the format of the directory structures.

To create a new file, it allocates a new FCB.

The system then reads the appropriate directory into memory, updates it with the new file name and FCB and writes it back to the disk.

Process of Opening a file

After a file has been created, it can be used for I/O.



To open a file we use a system call open( ). The open( ) call passes a file name to the logical file system.

The open( ) system call first searches the system-wide open-file table to see if the file is already in use by another process.

If the file is open, a per-process open-file table entry is created pointing to the existing system-wide open-file table.

If the file is not already open, the directory structure is searched for the given file name.

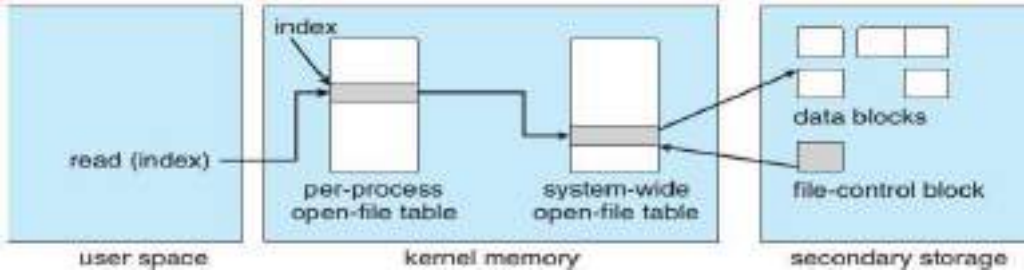
Once the file is found, the FCB is copied into a system-wide open-file table in memory.

This table not only stores the FCB but also tracks the number of processes that have the file open.

Process of Reading a File

After an entry has been made in the per-process open-file table, with a pointer to the entry in the system-wide open-file table and some other fields.

These other fields may include a pointer to the current location in the file for the next read( ) or write( ) operation and the access mode in which the file is open.



The open( ) call returns a pointer to the appropriate entry in the per-process file-system table. All file operations are then performed via this pointer.

The file name may not be part of the open-file table, as the system has no use for it once the appropriate FCB is located on disk.

FCB could be cached to save time on subsequent opens of the same file. The name given to the entry varies.

UNIX refers to FCB as a **File Descriptor**. Windows refers to FCB as a **File Handle.**

Process of Closing the File

When a process closes the file, the per-process table entry is removed and the system-wide entry's open count is decremented.

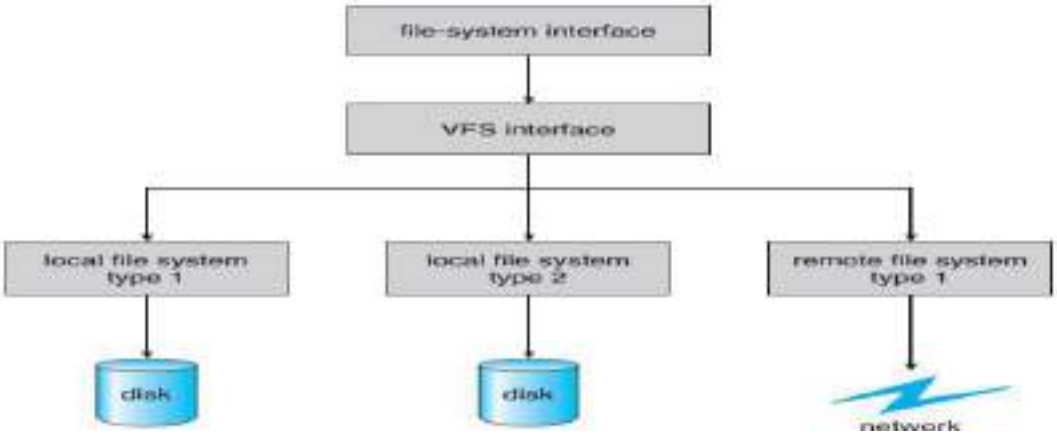
When all users that have opened the file close it, any updated metadata is copied back to the disk-based directory structure and the system-wide open-file table entry is removed.

Virtual File Systems

The first layer is the file-system interface, based on the open( ), read( ), write( ) and close( ) system calls and also based on file descriptors.

The second layer is called the **virtual file system (VFS)** layer. The VFS layer serves two important functions:

1. It separates file-system-generic operations from their implementation by defining a clean VFS interface. Several implementations for the VFS interface may coexist on the same machine, allowing transparent access to different types of file systems mounted locally.
2. It provides a mechanism for uniquely representing a file throughout a network. The VFS is based on a file-representation structure called a **vnode**, that contains a numerical designator for a network-wide unique file. This network-wide uniqueness is required for support of network file systems. The kernel maintains one vnode structure for each active node. A node may be a file or directory.



The VFS distinguishes local files from remote ones and local files are further distinguished according to their file-system types.

VFS activates file-system-specific operations to handle local requests according to their file-system types and calls the Network File System (NFS) protocol procedures for remote requests.

File handles are constructed from the relevant vnodes and are passed as arguments to these procedures.

The third layer implements the file-system type or the remote-file-system protocol.

ALLOCATION METHODS

Three major methods of allocating disk space are in wide use:

1. Contiguous Allocation
2. Linked Allocation
3. Indexed Allocation

Contiguous Allocation

Contiguous allocation requires that each file occupy a set of contiguous blocks on the disk.

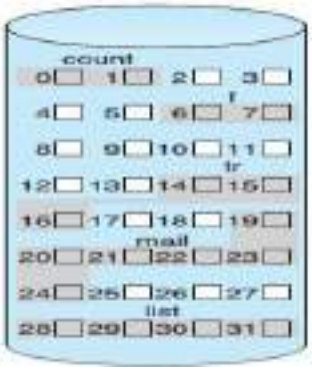
Disk addresses define a linear ordering on the disk. With this ordering, assuming that only one job is accessing the disk, accessing block $b + 1$ after block b normally requires no head movement.

When head movement is needed the head need only move from one track to the next.

Contiguous allocation of a file is defined by the disk address and length (in block units) of the first block.

If the file is n blocks long and starts at location b, then it occupies blocks $\boldsymbol{b, b + 1, b + 2, ..., b + n - 1}$.

The directory entry for each file indicates the address of the starting block and the length of the area allocated for this file.



directory

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

Accessing a file that has been allocated contiguously is easy.

For sequential access, the file system remembers the disk address of the last block referenced and when necessary, reads the next block.

For direct access to block ***i*** of a file that starts at block ***b***, it can immediately access block **b+i**.

Both sequential and direct access can be supported by contiguous allocation.

Two-Problems with Contiguous Allocation:

1. Finding space for a new file
2. Determining how much space is needed for a file.

Finding space for a new file

The contiguous-allocation problem occurs in dynamic storage-allocation that involves how to satisfy a request of size n from a list of free holes.

First fit and best fit are the most common strategies used to select a free hole from the set of available holes.

Both, First fit and Best fit algorithms suffer from the problem of **external fragmentation**.

As files are allocated and deleted, the free disk space is broken into little pieces.

External fragmentation exists whenever free space is broken into chunks.

It becomes a problem when the largest contiguous chunk is insufficient for a request.

The storage is fragmented into a number of holes, none of which is large enough to store the data.

Compaction solves external fragmentation by copying an entire file system onto another disk.

The original disk is then freed completely, creating one large contiguous free space.

We then copy the files back onto the original disk by allocating contiguous space from this one large hole.

The cost of compaction is very high when the size of the hard disk is huge. The time taken for compaction will be high as the size of the hard disk increases.

Determining how much space is needed for a file

When the file is created, the total amount of space it will need must be found and allocated.

If we allocate too little space to a file, we may find that the file cannot be extended.

Especially with a best-fit allocation strategy, the space on both sides of the file may be in use. Hence, we cannot make the file larger in place.

First, the user program can be terminated, with an appropriate error message.

The user must then allocate more space and run the program again.

These repeated runs may be costly.

To prevent them, the user will normally overestimate the amount of space needed, resulting in considerable wasted space.

The other possibility is to find a larger hole, copy the contents of the file to the new space and release the previous space.

All these are time consuming and system performance will be effected.

Linked Allocation

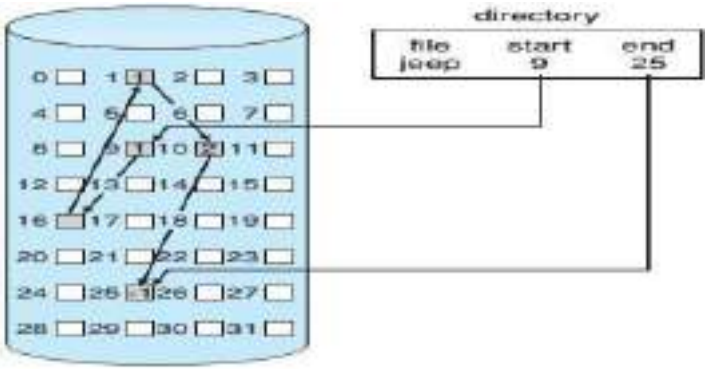
Linked allocation solves all problems of contiguous allocation.

With linked allocation, each file is a linked list of disk blocks.

Disk blocks are scattered anywhere on the disk.

The directory contains a pointer to the first and last blocks of the file.

Consider the below figure that shows linked list allocation:



A file of five blocks might start at block 9 and continue at block 16, then block 1, then block 10 and finally block 25.

Each block contains a pointer to the next block.

These pointers are not made available to the user.

Thus, if each block is 512 bytes in size and a disk address (the pointer) requires 4 bytes, then the user sees blocks of 508 bytes.

Advantage:

Linked List allocation avoids Compaction

To create a new file, we simply create a new entry in the directory.

With linked allocation, each directory entry has a pointer to the first disk block of the file.

This pointer is initialized to null (the end-of-list pointer value) to signify an empty file. The size field is also set to 0.

A write to the file causes the free-space management system to find a free block and this new block is written to and is linked to the end of the file.

To read a file, we simply read blocks by following the pointers from block to block.

There is no external fragmentation with linked allocation and any free block on the free-space list can be used to satisfy a request.

The size of a file need not be declared when the file is created. A file can continue to grow as long as free blocks are available. Hence by Linked List allocation avoids external fragmentation and it avoid need for compact disk space.

1. It is inefficient for Direct Access
2. Space for Pointers
3. Reliability

Linked list allocation can be used effectively only for sequential-access files.

To find the $\mathbf{i^{th}}$ block of a file, we must start at the beginning of that file and follow the pointers until we get to the $\mathbf{i^{th}}$ block.

Each access to a pointer requires a disk read and some require a disk seek.

It is inefficient to support a direct-access capability for linked-allocation files.

If a pointer requires 4 bytes out of a 512-byte block, then 0.78 percent of the disk is being used for pointers, rather than for information.

Each file requires slightly more space than it would otherwise.

Solutions to above problems: **Clustering**

Cluster is a collection multiple blocks and we allocate clusters rather than blocks.

Let the file system define a cluster as four blocks and operate on the disk only in cluster units. Pointers then use a much smaller percentage of the file's disk space.

This method improves disk throughput and decreases the space needed for block allocation and free-list management.

Clustering approach leads to the problem of internal fragmentation, because more space is wasted when a cluster is partially full than when a block is partially full.

Reliability issues will be arised

The files are linked together by pointers scattered all over the disk.

If a pointer were lost or damaged, this might result in picking up the wrong pointer.

This error could in turn result in linking into the free-space list or into another file.

FILE ALLOCATION TABLE (FAT)

Linked List allocation uses File Allocation Table.

FAT is very efficient method of disk-space allocation. It was used by the MS-DOS operating system.

A section of disk at the beginning of each volume contains the table.

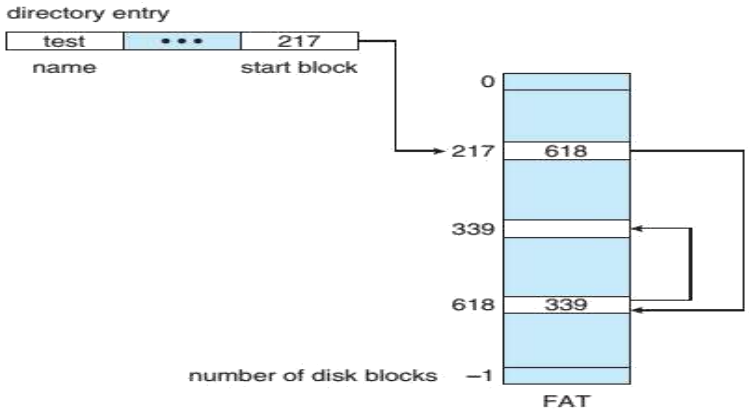
The table has one entry for each disk block and is indexed by block number.

The directory entry contains the block number of the first block of the file.

The table entry indexed by that block number contains the block number of the next block in the file.

This chain continues until it reaches the last block, which has a special end-of-file value as the table entry.

An unused block is indicated by a table value of 0. Allocating a new block to a file is to find the first 0-valued table entry and replacing the previous end-of-file value with the address of the new block. The 0 is then replaced with the end-of-file value.



The FAT allocation scheme can result in a significant number of disk head seeks, unless the FAT is cached.

The disk head must move to the start of the volume to read the FAT and find the location of the block in question, then move to the location of the block itself.

A benefit is that random-access time is improved, because the disk head can find the location of any block by reading the information in the FAT.

Indexed Allocation

In Indexed allocation, each file has its own index block. An index block is an array of disk-block addresses.

The **i**th entry in the index block points to the **i**th block of the file.

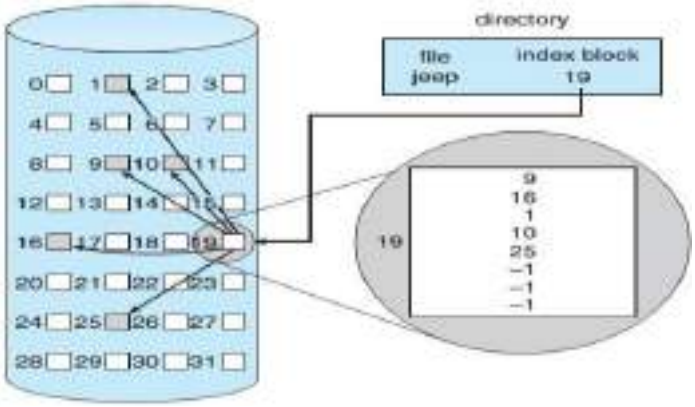
The directory contains the address of the index block.

To find and read the i^{th} block, we use the pointer in the i^{th} index-block entry.

When the file is created, all pointers in the index block are set to null.

When the i^{th} block is first written, a block is obtained from the free-space manager and its address is put in the i^{th} index-block entry.

Indexed allocation supports direct access, without suffering from external fragmentation, because any free block on the disk can satisfy a request for more space.



Indexed allocation suffers from wasted space and Pointer Overhead.

The pointer overhead of the index block is greater than the pointer overhead of linked allocation.

Consider we have a file of only one or two blocks.

With linked allocation, we lose the space of only one pointer per block.

With indexed allocation, an entire index block must be allocated, even if only one or two pointers will be non-null.

Determining size of the index block is a big issue in Indexed allocation. Several mechanisms are used for this purpose are:

1. Linked Scheme
2. Multilevel Index
3. Combined Scheme

Linked Scheme

An index block is normally one disk block. Thus, it can be read and written directly by itself.

To allow for large files, we can link together several index blocks.

Example: An index block might contain a small header giving the name of the file and a set of the first 100 disk-block addresses.

The next address (i.e.) the last word in the index block is null (for a small file) or is a pointer to another index block (for a large file).

Multilevel index

A variant of linked representation uses a first-level index block to point to a set of second-level index blocks, which in turn point to the file blocks.

To access a block, the operating system uses the first-level index to find a second-level index block and then uses that block to find the desired data block.

This approach could be continued to a third or fourth level, depending on the desired maximum file size.

With 4,096-byte blocks, we could store 1,024 four-byte pointers in an index block.

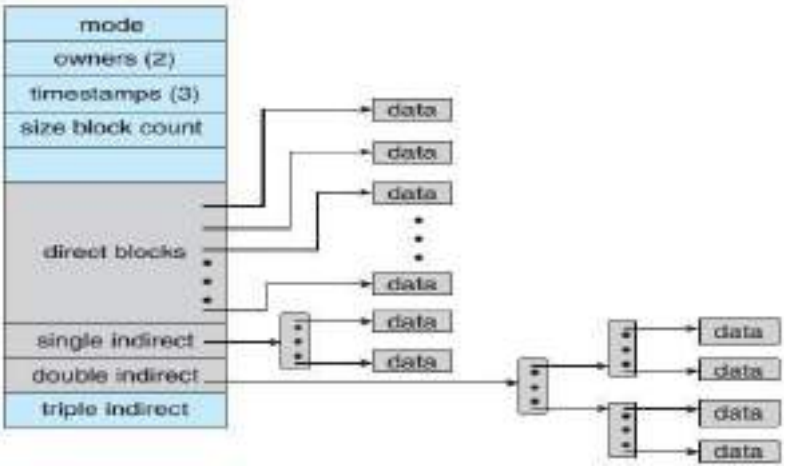
Two levels of indexes allow 1,048,576 data blocks and a file size of up to 4 GB.

Combined Scheme

It is used by the UNIX based file system that keeps the first 15 pointers of the index block in the file's inode.

The first 12 of these pointers point to **direct blocks** (i.e.) they contain addresses of blocks that contain data of the file.

Thus, the data for small files of no more than 12 blocks do not need a separate index block.



If the block size is 4 KB, then up to 48 KB of data can be accessed directly.

The next three pointers point to **Indirect blocks**.

The first points to a **Single indirect block**, which is an index block containing not data but the addresses of blocks that do contain data.

The second points to a **Double indirect block**, which contains the address of a block that contains the addresses of blocks that contain pointers to the actual data blocks.

The last pointer contains the address of a **Triple indirect block**.

FREE-SPACE MANAGEMENT

Since disk space is limited, we need to reuse the space from deleted files for new files.

To keep track of free disk space, the system maintains a **Free-Space List**.

Free-space list records all free disk blocks, those not allocated to some file or directory.

To create a file, we search the free-space list for the required amount of space and allocate that space to the new file. This space is then removed from the free-space list.

When a file is deleted, its disk space is added to the free-space list.

The free space can be managed in several ways:

1. Bit Vector
2. Linked List
3. Grouping
4. Counting
5. Space Maps

Bit Vector

The free-space list is implemented as a **Bit map** or **Bit vector**.

Each block is represented by one bit, the bit 1 represents block is free and bit 0 represents block is allocated.

Example: Consider a disk where blocks 2, 3, 4, 5, 8, 9, 10, 11, 12, 13, 17, 18, 25, 26 and 27 are free and the rest of the blocks are allocated. The free-space bit map would be

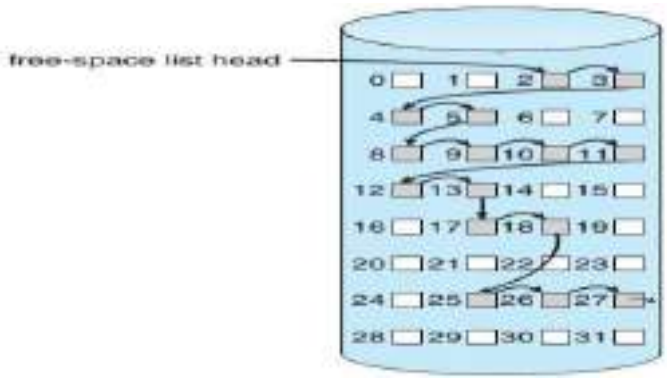
00111100111111000110000001110000 0

Advantage: Its relative simplicity and its efficiency in finding the first free block or *n* consecutive free blocks on the disk.

Disadvantage: Bit Vectors are kept in main memory is possible for smaller disks. For larger disks it is not efficient to keep it in Main memory because A 1-TB disk with 4-KB blocks requires 256 MB to store its bit map. So, as the disk size increases, the bit vector size is also increases.

Linked List

All the free disk blocks are linked together by 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.



Consider the above figure, that shows the set of free blocks 2, 3, 4, 5, 8, 9, 10, 11, 12, 13, 17, 18, 25, 26 and 27.

The system would keep a pointer to block 2 as the first free block. Block 2 would contain a pointer to block 3 and so on.

This scheme is not efficient; to traverse the list, we must read each block, which requires substantial I/O time.

Grouping

It stores the addresses of *n* free blocks in the first free block.

The first $n-1$ of these blocks are free blocks and the last block contains the addresses of another *n* free blocks and so on.

Addresses of a large number of free blocks can be found quickly than linked-list method.

Counting

When space is allocated with the contiguous-allocation algorithm or through clustering, several contiguous blocks may be allocated or freed simultaneously.

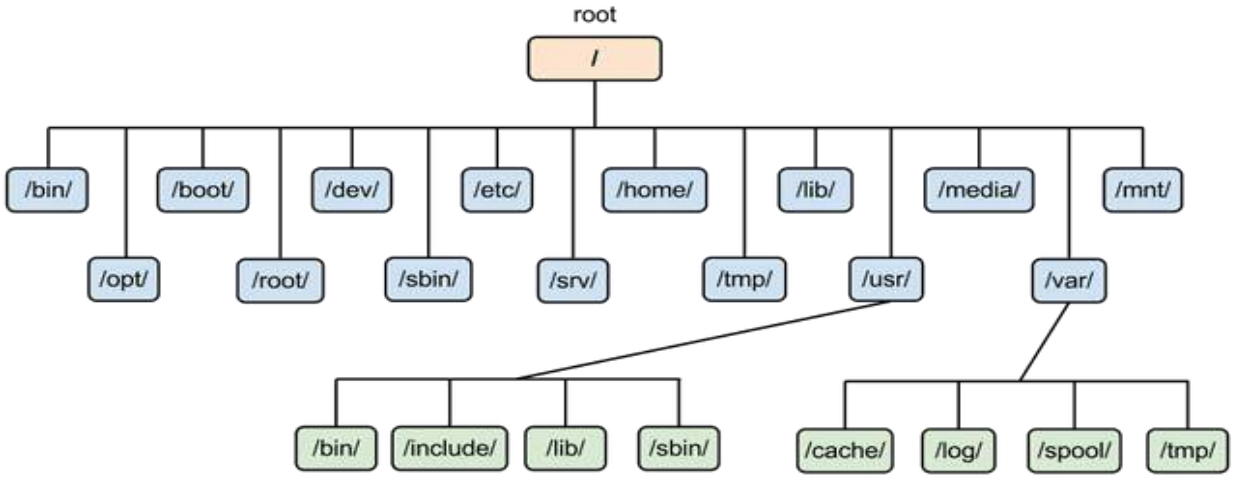
Here we keep the address of the first free block and the number (*n*) of free contiguous blocks that follow the first block.

Each entry in the free-space list then consists of a disk address and a count. Hence the overall disk entries are small.

CHAPTER – 3 – UNIX File System

UNIX File System

- In UNIX, the file system is a hierarchical structure of files and directories where users can store and retrieve information using the files.
- The file system consists of two main components which are files and directories.
- The highest-level directory in the entire hierarchical structure is known as the root.
- The root is denoted as ' / '.
- There can be many sub-directories under this directory.



- Directories and their descriptions are as follows:

- /bin: short for binaries, this is the directory where many commonly used executable commands reside
- /dev: contains device-specific files
- /etc: contains system configuration files
- /home: contains user directories and files
- /lib: contains all library files
- /mnt: contains device files related to mounted devices (like Hard drive, CD-ROM, etc.)

- /proc: contains files related to system processes
- /root: the root users' home directory (note this is different than ‘ / ‘ → / is the parent directory of all files including root, /root is home directory of user root→ user lands into this directory as soon as he/she logs on the machine)
- /sbin: binary files of the system reside here.
- /tmp: storage for temporary files that are periodically removed from the file system
- /usr: It is the directory holding user home directories, its use has changed, and it also contains executable commands
- /var: It is a short form for ‘variable’, a place for files that may often change

Types of UNIX File System

▶ The Unix File Systems are classified into six types, that are:

1. Ordinary Files
2. Directories
3. Special Files
4. Pipes
5. Sockets
6. Symbolic Links

Ordinary Files

▶ The files in Unix, which include program instructions, texts, and data, these files are known as ordinary files.

▶ Some of the characteristics of ordinary files are:

- There are no other files in ordinary files.
- They are always placed under the directory file.
- These files store the information of users. It may be some text or any image which we have to draw.
- We mainly work with ordinary files.

Directories

▶ Directories store both kinds of files: special as well as ordinary files. Some of the characteristics of directories are:

- Directories may contain other directories also.
- All these directories are the descendants of the root directory.
- We use directories to organize the collection of files.
- It doesn’t contain real information like text or images.

Special Files

▶ These files represent physical devices like printers, tape drivers, or a terminal. Some of its characteristics are:

- Special files represent input/output operations on Linux and Unix systems.

Pipes

▶ Pipes are used to link commands together to store both kinds. The following are the characteristics of pipes:

- It is a temporary file that only holds the data from one command until another reads it.
- Unix pipes provide the one-way flow of data.

Any command's output is assigned to its next command as an input. **Sockets**

- The files that enable advanced inter-process communication are called Sockets. It is also known as an inter-process communication socket. Some of the characteristics of sockets are:
- Unix sockets are used in the framework of client-server applications.
- It is essentially a data stream, exactly the same as the network stream and network socket, except where each transaction is local to the file system.

Symbolic Links

- It is used to reference another file in the system. Characteristics of symbolic links are:
- It involves a text representation of the path to the file it refers to.
- To an end-user, a symbolic link will seem to have its name, yet when we need to write or read data to this file, it will reference these tasks to the file it focuses on.
- If the source file is deleted or moved to a different location, the symbolic file will not function.

FILES

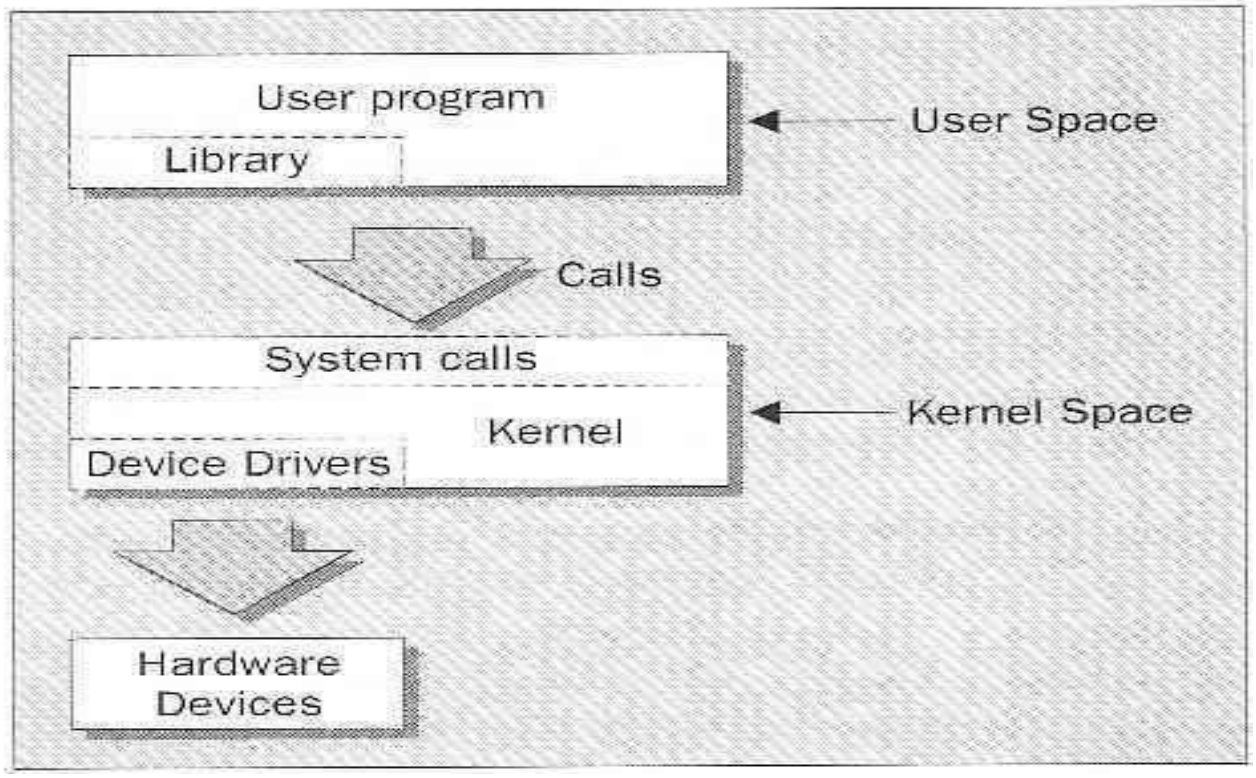
- In UNIX everything is a file.
- Programs use disk files, serial ports, printers and others in the same way as they would use a file.
- Directories are a special type of files.

File Attributes:

- File type-type of file
- Access Permissions-access permissions of the file
- dlinkcount-no of hard links of a file
- UID- file owner userID
- GID-the file groupID
- Inode number-inode number of a file
- FilesystemID-systemID where the file is stored.

System Calls and Device Drivers

- System call is a method for a computer program to request a service from the kernel of the OS.
- System calls are provided by UNIX to access and control files and devices.
- Drivers are used to help the hardware devices interact with the operating system. In Linux, even the hardware devices are treated like ordinary files, which makes it easier for the software to interact with the device drivers. When a device is connected to the system, a device file is created in /dev directory.
- System calls to access device drivers include:
- Open- open a file or device.
- Close- close a file or device.
- read-read from an open file or device.
- Write- write to a file or device.



File Descriptor

- File descriptor is an integer which is an index in the kernel on the opened files(Which is called file descriptor table). It is used to deal with the files . most of the functions like open,close,read using the file descriptors to deal with the files.
- We have three standard POSIX file descriptors, corresponding to the three standard streams:

Integer value	Name	<stdio.h> file stream
0	**Standard input**	**stdin**
1	**Standard output**	**stdout**
2	**Standard error**	**stderr**

Inode Number

- An Inode number is a uniquely existing number for all the files in Linux and all Unix type systems.
- When a file is created on a system, a file name and Inode number is assigned to it.
- Generally, to access a file, a user uses the file name but internally file name is first mapped with respective Inode number stored in a table.

Inode Contents

- An Inode is a data structure containing metadata about the files.
- User ID of file, Group ID of file
- Device ID
- File size
- Date of creation, Permission

- Owner of the file
- Link counter to determine number of hard links

File System Calls

- Open: To create a new file descriptor ,use the open system call.

```
#include <fcntl.h>
#include <sys/types.h>
#include <sys/stat.h>

int open(const char *path, int oflags);
int open(const char *path, int oflags, mode_t mode);
```

- **Open:** establishes an access path to a file or device.
- The name of the file or device to be opened is passed as a parameter ,**path**, and the **oflags** parameter is used to specify actions to be taken on opening the file.
- The **oflags** are specified as a bitwiseOR of a mandatory file access mode and other optional modes.The **open** call must specify one of the following file access modes:

Mode	Description
O_RDONLY	Open for read-only
O_WRONLY	Open for write-only
O_RDWR	Open for reading and writing

- The call may also include a combination (bitwise OR) of the following optional modes in the **oflags** parameter:

O_APPEND	Place written data at the end of the file.
O_TRUNC	Set the length of the file to zero, discarding existing contents.
O_CREAT	Creates the file, if necessary, with permissions given in **mode**.
O_EXCL	Used with O_CREAT, ensures that the caller creates the file. The **open** is atomic, i.e. it's performed with just one function call. This protects against two programs creating the file at the same time. If the file already exists, **open** will fail.

- **Initial Permissions**
- When we create a file using the **O_CREAT** flag with **open**, we must use the three parameter form. **mode**, the third parameter, is made form a bitwise OR of the flags defined in the header file **sys/stat.h**. These are:

S_IRUSR	Read permission, owner.
S_IWUSR	Write permission, owner.
S_IXUSR	Execute permission, owner.
S_IRGRP	Read permission, group.
S_IWGRP	Write permission, group.
S_IXGRP	Execute permission, group.
S_IROTH	Read permission, others.
S_IWOTH	Write permission, others.
S_IXOTH	Execute permission, others.

```
#include <unistd.h>
#include <sys/stat.h>
#include <fcntl.h>

int main()
{
    char c;
    int in, out;

    in = open("file.in", O_RDONLY);
    out = open("file.out", O_WRONLY|O_CREAT, S_IRUSR|S_IWUSR);
    while(read(in,&c,1) == 1)
        write(out,&c,1);

    exit(0);
}
```

- EXAMPLE:

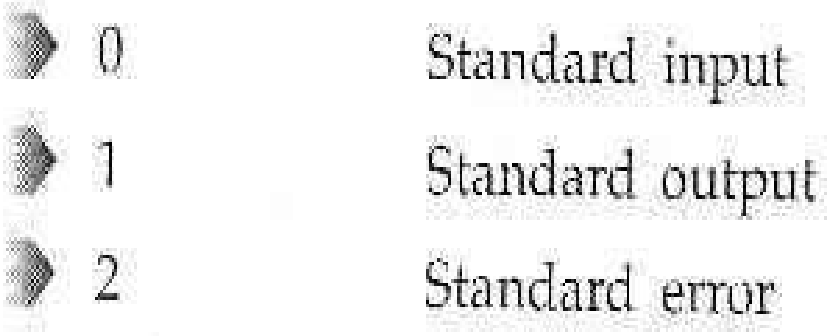
```
open ("myfile", O_CREAT, S_IRUSR|S_IXOTH);
```

Has the effect of creating a file called **myfile**, with read permission for the owner and execute permission for others, and only those permissions.

```
#include <unistd.h>

int close(int fildes);
```

- **Write**: Each running program, called a **process**, has associated with it a number of file descriptors. When a program starts, it usually has three of these descriptors already opened.




```
#include <unistd.h>

size_t write(int fildes, const void *buf, size_t nbytes);
```

The **write** system call arranges for the first **nbytes** bytes from **buf** to be written to the file associated with the file descriptor **fildes**.

```
#include <unistd.h>

size_t read(int fildes, void *buf, size_t nbytes);
```

The **read** system call reads up to **nbytes** of data from the file associated with the file descriptor **fildes** and places them in the data area **buf**.

- This program, **simple_read.c**, copies the first 128 bytes of the standard input to the standardoutput.

```
#include <unistd.h>

int main()
{
    char buffer[128];
    int nread;

    nread = read(0, buffer, 128);
    if (nread == -1)
        write(2, "A read error has occurred\n", 26);

    if ((write(1,buffer,nread)) != nread)
        write(2, "A write error has occurred\n",27);

    exit(0);
}
```

- lseek:

```
#include <unistd.h>

#include <sys/types.h>

off_t lseek(int fildes, off_t offset, int whence);
```

The **lseek** system call sets the read/write pointer of a file descriptor, **fildes**. You use it to set where in the file the next read or write will occur.
The **offset** parameter is used to specify the position and the **whence** parameter specifies how the offset is used.

- Whence can be one of the following:

SEEK_SET	**offset** is an absolute position
SEEK_CUR	**offset** is relative to the current position
SEEK_END	**offset** is relative to the end of the file

File Status Information-Stat Family: fstat, stat and lstat

```
#include <unistd.h>
#include <sys/stat.h>
#include <sys/types.h>

int fstat(int fildes, struct stat *buf);
int stat(const char *path, struct stat *buf);
int lstat(const char *path, struct stat *buf);
```

- The **fstat** system call returns status information about the file associated with an open file descriptor.

The members of the structure stat will include:

stat Member	Description
st_mode	File permissions and file type information.
st_ino	The inode associated with the file.
st_dev	The device the file resides on.
st_uid	The user identity of the file owner.
st_gid	The group identity of the file owner.
st_atime	The time of last access.
st_ctime	The time of last change to mode, owner, group or content.
st_mtime	The time of last modification to contents.
st_nlink	The number of hard links to the file.

- The permissions flags are the same as for the **open** system call above

- **S_IFBLK** Entry is a block special device.
- **S_IFDIR** Entry is a directory.
- **S_IFCHR** Entry is a character special device.
- **S_IFIFO** Entry is a FIFO (named pipe).
- **S_IFREG** Entry is a regular file.
- **S_IFLNK** Entry is a symbolic link.

FilePermission- chmod

- You can change the permissions on a file or directory using the **chmod** system call.

```
#include <sys/stat.h>

int chmod(const char *path, mode_t mode);
```

Chown

- A super user can change the owner of a file using the **chown** system call.

```
#include <unistd.h>

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

Links-soft link and hard link

- **Soft link(symbolic links):**Refer to a symbolic path indicating the abstract location of another file.
- Used to provide alternative means of referencing files.
- Users may create links for files using **ln** command by specifying–**s** option.
- **Hard links:** Refer to the specific location of physical data.
- A hard link is a UNIX pathname for a file.
- Most of the files have only one hardlink. However users may create additional hardlinks for files using **ln** command.

unlink, link, symlink

```
#include <unistd.h>

int unlink(const char *path);
int link(const char *path1, const char *path2);
int symlink(const char *path1, const char *path2);
```

- We can remove a file using **unlink**.
- The **unlink** system call decrements the link count on a file. The **link** system call creates a new link to an existing file.
- The **symlink** creates a symbolic link to an existing file.

Directories: mkdir, rmdir

We can create and remove directories using the mkdir and rmdir system calls.

```
#include <sys/stat.h>

int mkdir(const char *path, mode_t mode);
```

The mkdir system call makes a new directory with path as its name.

```
#include <unistd.h>

int rmdir(const char *path);
```

The rmdir system call removes an empty directory.

opendir, readdir

- The **opendir** function opens a directory and establishes a directory stream.

```
#include <sys/types.h>
#include <dirent.h>

DIR *opendir(const char *name);
```

- The **readdir** function returns a pointer toa structure detailing the next directory entry in the directory stream **dirp**.

```
#include <sys/types.h>
#include <dirent.h>

struct dirent *readdir(DIR *dirp);
```

- The **dirent** structure containing directory entry details included the following entries:

ino_t	d_ino	The inode of the file.
char	d_name[]	The name of the file.

Umask

- The umask command in Linux is used to set default permissions for files or directories the user creates.

- The umask command specifies the permissions that the user does not want to be given out to the newly created file or directory.
- umask works by doing a Bitwise AND with the bitwise complement (where the bits are inverted, i.e. 1 becomes 0 and 0 becomes 1) of the umask.
- The bits which are set in the umask value, refer to the permissions, which are not assigned by default, as these values are subtracted from the maximum permission for files/directories.
- On Linux and Unix operating systems, all new files are created with a default set of permissions. The umask utility allows you to view or to set the file mode creation mask, which determines the permissions bits for newly created files or directories.
- By default, on Linux systems, the default creation permissions are 666 for files, which gives read and write permission to user, group, and others, and to 777 for directories, which means read, write and execute permission to user, group, and others. Linux does not allow a file to be created with execute permissions.
- The default creation permissions can be modified using the umask utility.
- umask affects only the current shell environment. On most Linux distributions, the default system-wide umask value is set in the pam_umask.so or /etc/profile file.
- If you want to specify a different value on a per-user basis, edit the user's shell configuration files such as ~/.bashrc or ~/.zshrc. You can also change the current session umask value by running umask followed by the desired value.
- To view the current mask value, simply type umask without any arguments:
- umask
- The output will include the
- 022
- The umask value contains the permission bits that will NOT be set on the newly created files and directories.
- The default creation permissions for files are 666 and for directories 777. To calculate the permission bits of the new files, subtract the umask value from the default value.
- For example, to calculate how umask 022 will affect newly created files and directories, use:
- Files: 666 - 022 = 644. The owner can read and modify the files. Group and others can only read the files.

- Directories: 777 - 022 = 755.The owner can cd into the directory, and list, read, modify, create or delete the files in the directory. Group and others can cd into the directory and list and read the files.