UNIT I INTRODUCTION 7

Computer System – Elements and organization; Operating System Overview – Objectives and Functions – Evolution of Operating System; Operating System Structures – Operating System Services – User Operating System Interface – System Calls – System Programs – Design and Implementation–Structuring methods.

INTRODUCTION

Operating System is a program that acts as an interface between a user of a computer and the computer hardware. Eg. Windows (Microsoft), iOS (Apple), Android (Google), Linux/Ubuntu (open source)

Goals of Operating System:

- ❖ Execute user programs and make solving the user problems easier
- ❖ Make the computer system convenient to use
- ❖ Use the computer hardware in an efficient manner

1. COMPUTER SYSTEM

A computer system can be divided roughly into four components: the **hardware,** the **operating system,** the **application programs,** and a **user**

Hardware: The Central Processing Unit (CPU), the Memory, and the Input/output (I/O) devices provides the basic computing resources for the system.

Application Programs: Such as word processors, spreadsheets, compilers, and web browsers define the ways in which these resources are used to solve users' computing problems.

Operating System: Controls the hardware and coordinates its use among the various application programs for the various users.

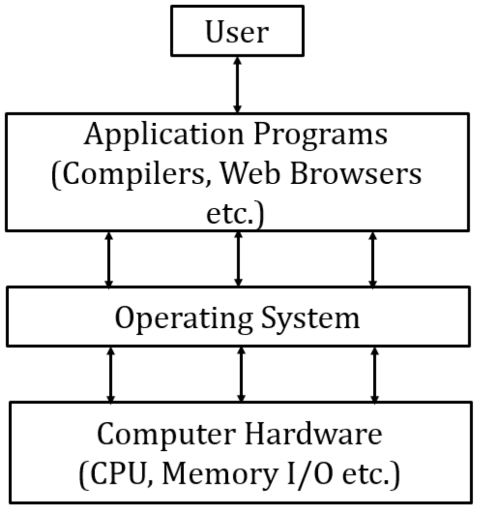

User
Application Programs (Compilers, Web Browsers etc.)
Operating System
Computer Hardware (CPU, Memory I/O etc.)

Figure : Computer System

The operating system provides the means for proper use of these resources in the operation of the computer system.
Operating Systems from two viewpoints:

- User View
- System View

- **User View**

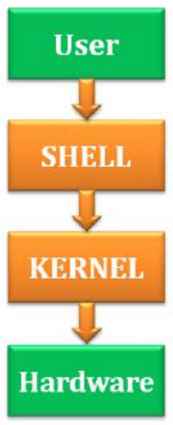
The user's view of the computer varies according to the interface being used. Many computer users sit with a laptop or in front of a PC consisting of a monitor, keyboard, and mouse. Such a system is designed for one user to control its resources. For this, the operating system is designed mostly for **ease of use**, performance and security and not to **resource utilization.**

Many users interact with mobile devices such as smartphones and tablets. These devices are connected to networks through cellular or other wireless technologies. The user interface for mobile computers generally features a touch screen, where the user interacts with the system by pressing and swiping fingers across the screen rather than using a physical keyboard and mouse. Many mobile devices also allow users to interact through a voice recognition interface, such as Apple's Siri.

- **System View**

From the computer's point of view, the operating system is the program most intimately involved with the hardware. It is a **resource allocator**. A computer system has many resources that may be required to solve a problem: CPU time, memory space, storage space, I/O devices, and so on. The operating system acts as the manager of these resources. The operating system must decide how to allocate the resources to specific programs and users so that it can operate the computer system efficiently and fairly.

Components of Operating System



Shell :

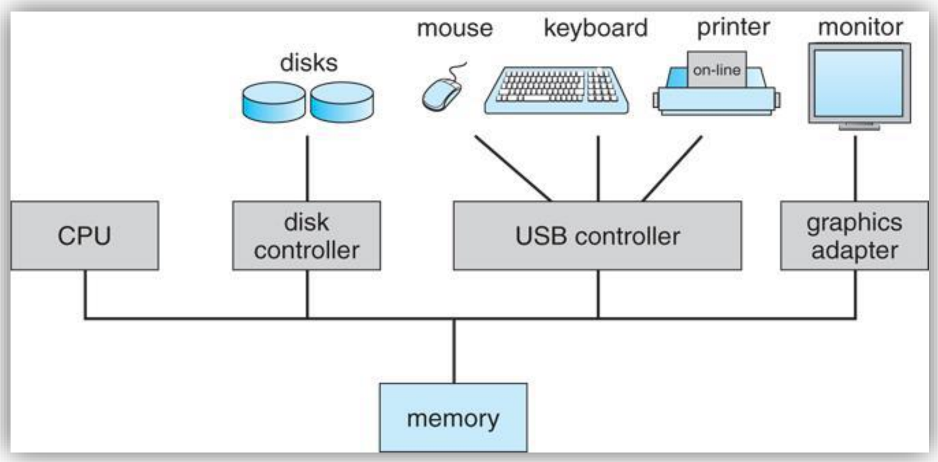
- ❖ Environment that gives a user an interface to access the operating system’s services.
- ❖ Launch Applications
- ❖ User Mode

Kernel :

- ❖ Keep track of the hardware and computer’s operations.
- ❖ Kernel Mode

Examples of Operating System

2. COMPUTER SYSTEM – ELEMENTS AND ORGANIZATION



A computer system consists of CPU and a number of device controllers connected through a common **bus** that provides access between components and shared memory. Each device controller is in charge of a specific type of device, more than one device may be attached. For example, one system USB port can connect to a USB hub, to which several devices can be connected. A device controller maintains some local buffer storage and a set of special purpose registers. The device controller is responsible for moving the data between the peripheral devices that it controls and its local buffer storage.

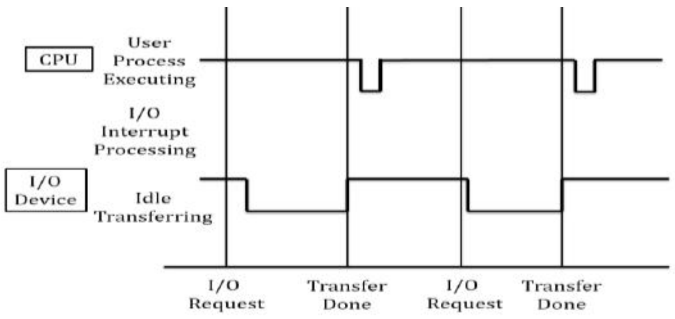
Operating systems have a **device driver** for each device controller. This device driver understands the device controller and provides the rest of the operating system with a uniform interface to the device. The CPU and the device controllers can execute in parallel.

Three key aspects of the system are

- ❖ Interrupts
- ❖ Storage Structure
- ❖ I/O Structure

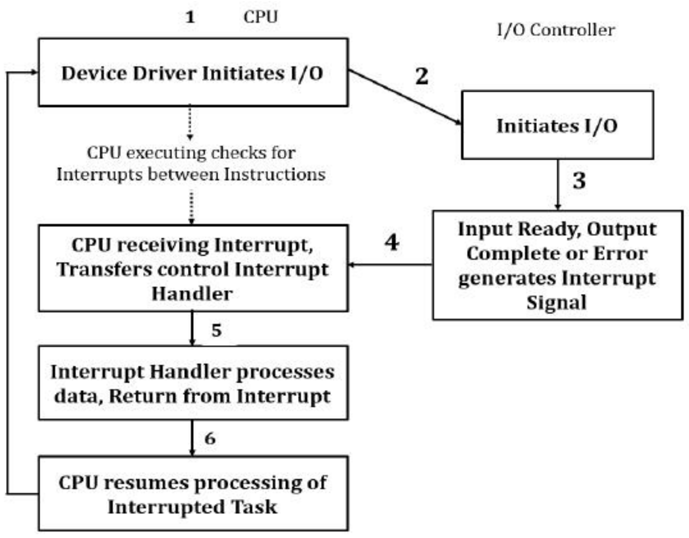
❖ Interrupts

Hardware may trigger an interrupt by sending a signal to the CPU. Software may trigger an interrupt by executing a special operation called a **system call.** When the CPU is interrupted, it stops what it is doing and immediately transfers execution to a fixed location. The fixed location usually contains the starting address where the service routine for the interrupt is located. The interrupt service routine executes; on completion, the CPU resumes the interrupted computation. A time line of this operation is shown below



The interrupt routine is called indirectly through the table, The table holds the addresses of the interrupt service routines for the various devices. This array, or **interrupt vector,** of addresses is then indexed by a unique device number, given with the interrupt request, to provide the address of the interrupt service routine for the interrupting device.

The interrupt architecture must also save the address of the interrupted instruction. After the interrupt is serviced, the saved return address is loaded into the program counter, and the interrupted computation resumes as though the interrupt had not occurred. The device controller raises an interrupt by asserting a signal on the interrupt request line, the CPU catches the interrupt and dispatches it to the interrupt handler, and the handler clears the interrupt by servicing the device.


1
CPU
I/O Controller
Device Driver Initiates I/O
2
Initiates I/O
CPU executing checks for Interrupts between Instructions
3
4
Input Ready, Output Complete or Error generates Interrupt Signal
CPU receiving Interrupt, Transfers control Interrupt Handler
5
Interrupt Handler processes data, Return from Interrupt
6
CPU resumes processing of Interrupted Task

Most CPUs have two interrupt request lines. One is the **nonmaskable interrupt**, which is reserved for events such as unrecoverable memory errors. The second interrupt line is **maskable**: it can be turned off by the CPU before the execution of critical instruction sequences that must not be interrupted. The maskable interrupt is used by device controllers to request service.

❖ Storage Structure

- KB : 1024 Bytes
- MB : 1024 ^{2} Bytes
- GB : 1024 ^{3} Bytes (1 Million Bytes)
- TB : 1024 ^{4} Bytes (1 Billion Bytes)
- PB : 1024 ^{5} Bytes

The CPU load the instructions from memory. General-purpose computers run most of their programs from rewritable memory, called main memory (**RAM**).

The first program to run on computer to power ON is a **bootstrap program**, which then loads the operating system. Since RAM is **volatile,** loses its content when power is turned off or otherwise lost. So the bootstrap program cannot be stored in RAM. So the computer uses electrically erasable programmable read-only memory (EEPROM) and other forms of **firmware,** storage that is infrequently written to and is nonvolatile. EEPROM can be changed but cannot be changed frequently. In addition, it is low speed, and so it contains mostly static programs and data that aren't frequently used. For example, the iPhone uses EEPROM to store serial numbers and hardware information about the device.

All forms of memory provide an array of bytes. Each byte has its own address. Interaction is achieved through a sequence of load or store instructions to specific memory addresses. The load instruction moves a byte or word from main memory to an internal register within the CPU, whereas the store instruction moves the content of a register to main memory.

The CPU automatically loads instructions from main memory for execution from the location stored in the program counter. First fetches an instruction from memory and stores that instruction in the **instruction register**. The instruction is then decoded and may cause operands to be fetched from memory and stored in some internal register. After the instruction on the operands has been executed, the result may be stored back in memory.

The programs and data must be in main memory permanently. This arrangement is not possible on most systems for two reasons:

- ❖ Main memory is usually too small to store all needed programs and data permanently.
- ❖ Main memory, is volatile, it loses its contents when power is turned off or otherwise lost.

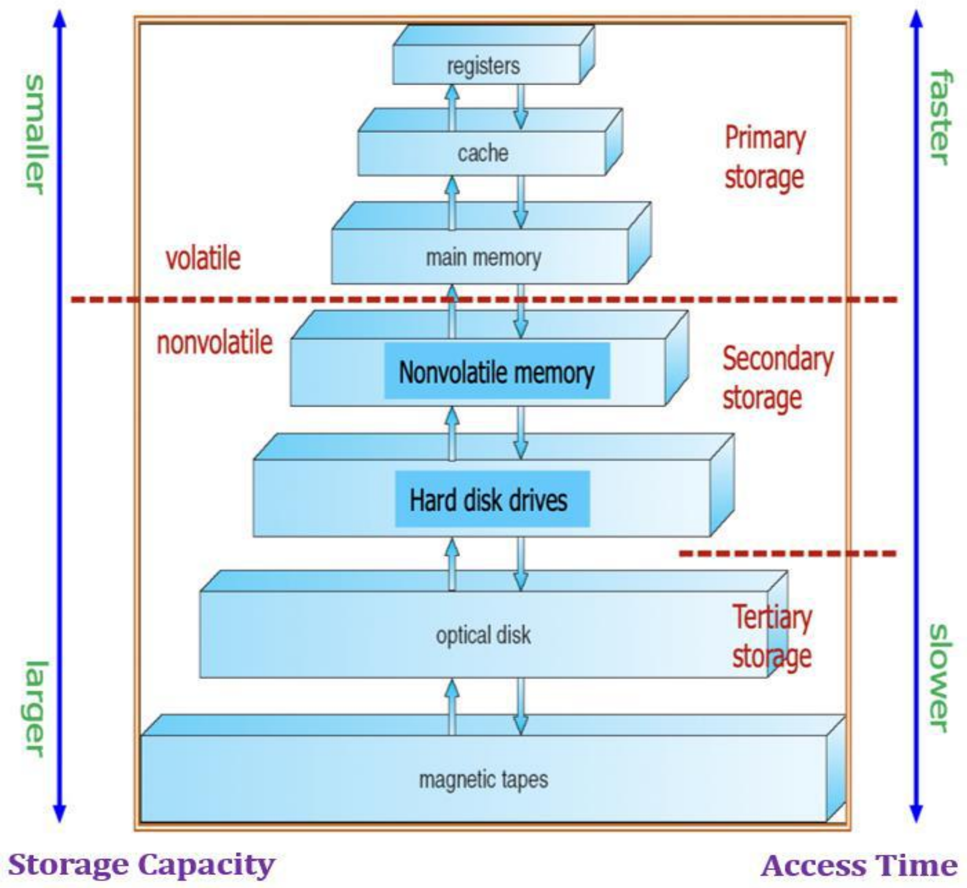
Most computer systems provide **secondary storage** as an extension of main memory. The main requirement for secondary storage is that it be able to hold large quantities of data permanently. The most common secondary-storage devices are **hard-disk drives** (**HDDs**) and **nonvolatile memory** (**NVM**) **devices**, which provide storage for both programs and data. Most programs are stored in secondary storage until they are loaded into memory.

Secondary storage is also much slower than main memory. Other possible components include cache memory, CD-ROM, magnetic tapes, and so on. Those that are slow enough and large enough that they are used only for special purposes, to store backup copies of material stored on other devices, are called **tertiary storage**.

The wide variety of storage systems can be organized in a hierarchy according to storage capacity and access time.

Smaller and faster memory closer to the CPU. Various storage systems are either volatile or nonvolatile. Volatile storage, loses its contents when the power to the device is removed, so data must be written to nonvolatile storage for safekeeping.

The top four levels of memory are constructed using **semiconductor memory**, which consists of semiconductor based electronic circuits. Non Volatile Memory devices, at the fourth level, are faster than hard disks. The most common form of NVM device is flash memory, which is popular in mobile devices such as smartphones and tablets. Increasingly, flash memory is being used for long term storage on laptops, desktops, and servers as well.



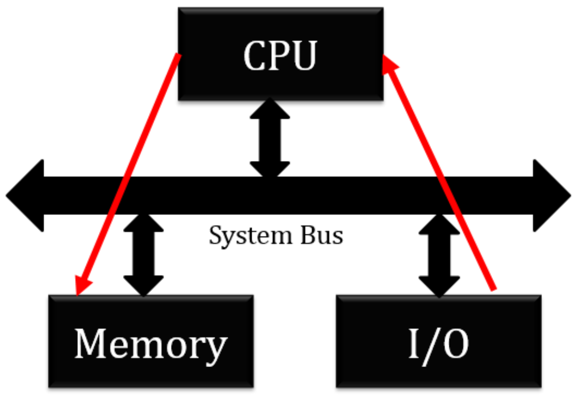
Volatile storage will be referred to simply as **memory**. Nonvolatile storage retains its contents when power is lost. This type of storage can be classified into two distinct types:

- **Mechanical** : Storage systems are HDDs, optical disks, holographic storage, and magnetic tape.
- **Electrical : S**torage systems are flash memory, FRAM, NRAM, and SSD.

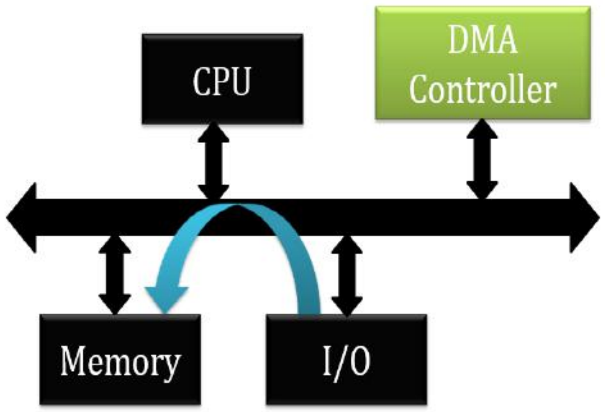
Mechanical storage is generally larger and less expensive per byte than electrical storage. Conversely, electrical storage is typically costly, smaller, and faster than mechanical storage.

❖ **I/O Structure**

A large portion of operating system code is dedicated to manage I/O. General purpose computer system consists of multiple devices, all of which exchange data via a common bus. The form of interrupt driven I/O is fine for moving small amounts of data, **direct memory access** (**DMA**) is used for bulk data transfer.



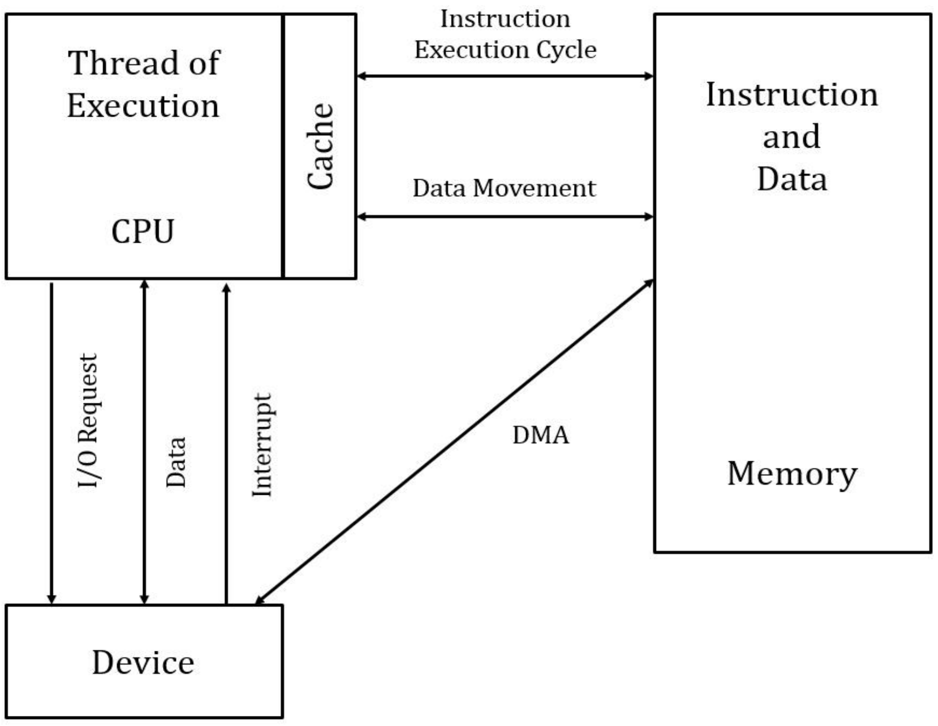
Normal Data Transfer



DMA Data Transfer

After setting up buffers, pointers, and counters for the I/O device, the device controller transfers an entire block of data directly to or from the device and main memory, with no intervention by the CPU. Only one interrupt is generated per block, to tell the device driver that the operation has completed, rather than the one interrupt per byte generated for low speed devices. While the device controller is performing these operations, the CPU is available to accomplish other work.

Some high-end systems use switch rather than bus architecture. On these systems, multiple components can talk to other components concurrently, rather than competing for cycles on a shared bus.



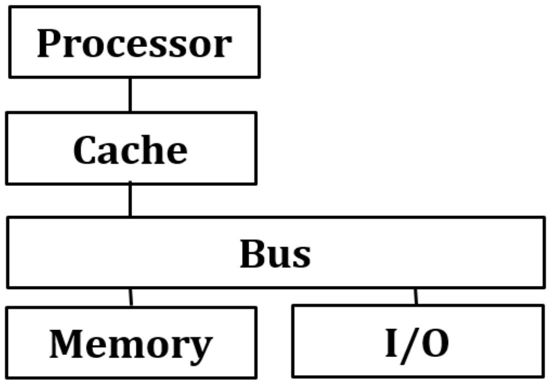
3. EVOLUTION OF OPERATING SYSTEM

CPU	: The hardware that executes instructions.
Processor	: A physical chip that contains one or more CPUs.
Core	: The basic computation unit of the CPU.
Multicore	: Including multiple computing cores on the same CPU.
Multiprocessor	: Including multiple processors.

According to the number of general-purpose processors used it is divided into

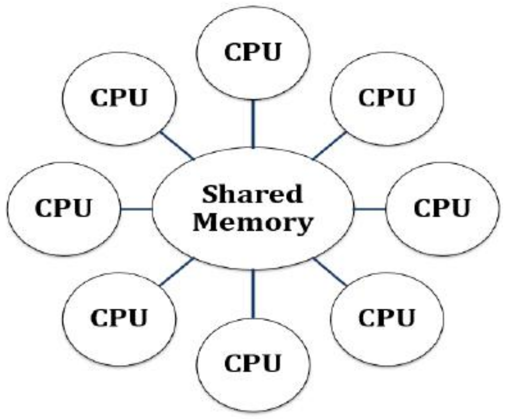
- ❖ Single Processor Systems
- ❖ Multiprocessor Systems
- ❖ Clustered Systems

❖ Single Processor Systems



On a single-processor system, there is one main CPU capable of executing the instructions. Most computer systems use a single processor containing one CPU with a single processing core. The **core** is the component that executes instructions and registers for storing data locally. The one main CPU with its core is capable of executing a general-purpose instruction set, including instructions from processes.

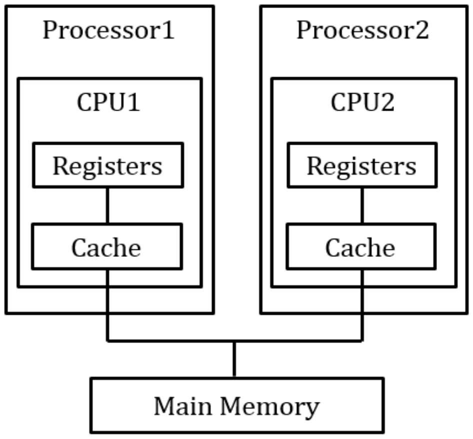
❖ Multiprocessor Systems



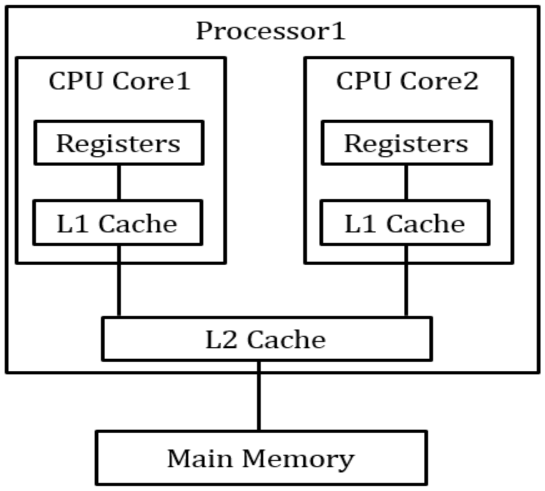
They have more processors, each with a single-core CPU. The processors share the computer bus and sometimes the clock, memory, and peripheral devices. The main advantages are

- Increased throughput
- Economy of scale
- Increased reliability – graceful degradation or fault tolerance

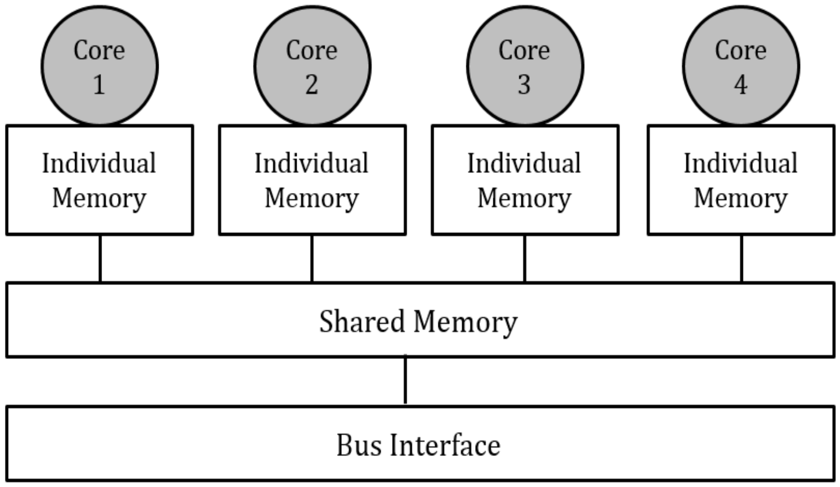
The advantage of multiprocessor systems is increased throughput. By increasing the number of processors, more work will be done in less time. The most common multiprocessor systems use **symmetric multiprocessing** (**SMP**), in which each peer CPU processor performs all tasks, including operating-system functions and user processes.



The Figure illustrates a typical SMP architecture with two processors, each with its own CPU. Each CPU processor has its own set of registers, and local cache. Main Memory is shared. The benefit is that many processes can run simultaneously *N* processes can run if there are *N* CPUs without causing performance to deteriorate significantly. However, since the CPUs are separate, one may be sitting idle while another is overloaded, resulting in inefficiencies. These inefficiencies can be avoided if the processors share certain data structures. The definition of **multiprocessor** has evolved over time and now includes **multicore** systems, in which multiple computing cores reside on a single chip. Multicore systems can be more efficient than multiple chips with single cores.

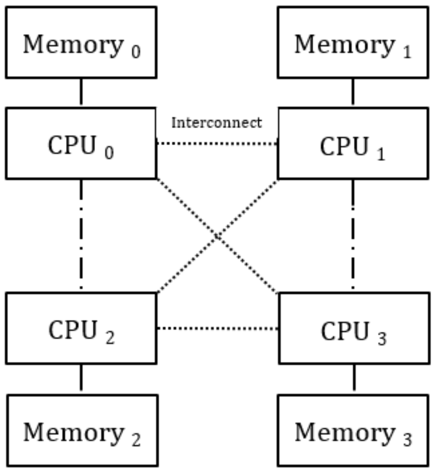


In a dual-core design with two cores on the same processor chip. In this design, each core has its own register set, as well as its own local cache, often known as a level 1, or L1, cache. Notice, too, that a level 2 (L2) cache is local to the chip but is shared by the two processing cores.



NUMA Multiprocessing Architecture

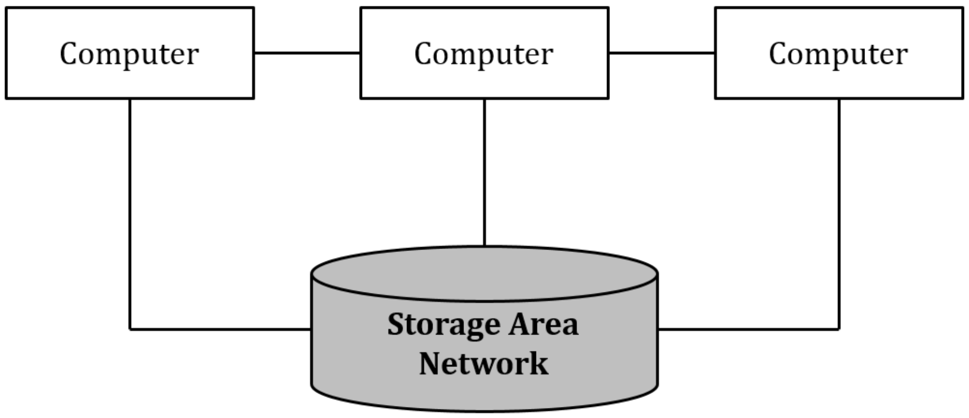
Adding additional CPUs to a multiprocessor system will increase computing power; and adding too many CPUs, becomes a bottleneck and performance begins to degrade. Instead to provide each CPU (or group of CPUs) with its own local memory that is accessed via a small, fast local bus. The CPUs are connected by a **shared system interconnect**, so that all CPUs share one physical address space. This approach known as **Non-Uniform Memory Access**, or **NUMA**



The advantage is that, when a CPU accesses its local memory, it is fast, and no contention over the system interconnect. Thus, NUMA systems can scale more effectively as more processors are added. A drawback is **increased latency**. For example, CPU0 cannot access the local memory of CPU3 as quickly as it can access its own local memory, slowing down performance.

Because NUMA systems can scale to accommodate a large number of processors, they are becoming increasingly popular on servers as well as high-performance computing systems.

❖ Clustered Systems

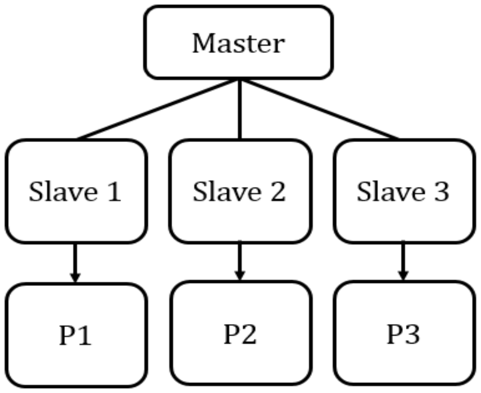


Clustered computers share storage and are closely linked via a local-area network LAN or a faster interconnect. Clustering is usually used to provide **high availability service** that is, service that will continue even if one or more systems in the cluster fail.

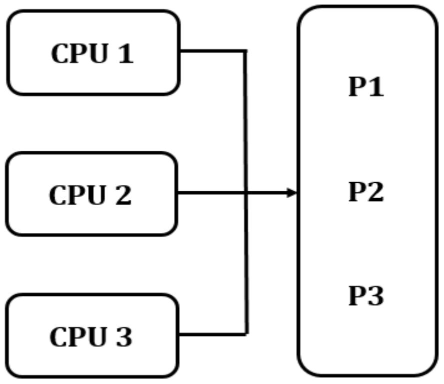
A layer of cluster software runs on the cluster nodes. Each node can monitor one or more of the others (over the network). If the monitored machine fails, the monitoring machine can take ownership of its storage and restart the applications that were running on the failed machine. High availability provides increased reliability, which is crucial in many applications. The ability to continue providing service proportional to the level of surviving hardware is called **graceful degradation**. Some systems go beyond graceful degradation and are called **fault tolerant**, because they can suffer a failure of any single component and still continue operation. Fault tolerance requires a mechanism to allow the failure to be detected, diagnosed, and, if possible, corrected.

Clustering can be structured asymmetrically or symmetrically.

In **Asymmetric Clustering**, one machine is in hot standby mode while the other is running the applications. The hot standby host machine does nothing but monitor the active server. If that server fails, the hot standby host becomes the active server.



Asymmetric Clustering



Symmetric Clustering

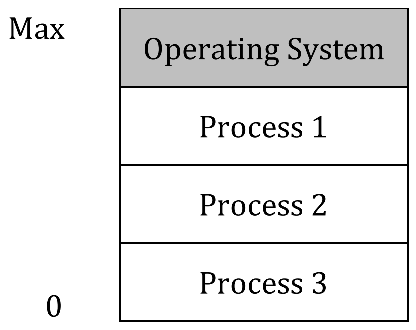
In **symmetric clustering**, two or more hosts are running applications and are monitoring each other. This structure is obviously more efficient, as it uses all of the available hardware.

4. OPERATING SYSTEM OPERATIONS

- ❖ Multiprogramming and Multitasking
- ❖ Dual-Mode and Multimode Operation
- ❖ Timer

❖ Multiprogramming

Multiprogramming increases CPU utilization, so that the CPU always has one to execute. In a multiprogrammed system, a program in execution is termed as **process**.



Memory Layout of a Multiprogramming System

The operating system picks and begins to execute one of these processes. Eventually, the process may have to wait for some task, such as an I/O operation, to complete. In a non-multiprogrammed System, the CPU would sit idle. In a multiprogrammed system, the operating system simply switches to, and executes, another process. When **that** process needs to wait, the CPU switches to **another** process, and so on. Eventually, the first process finishes waiting and gets the CPU back. As long as at least one process needs to execute, the CPU is never idle.

❖ Multitasking

CPU switches jobs so frequently that users can interact with each job while it is running, creating **interactive** computing

- **Response time** should be < 1 second
- Each user has at least one program executing in memory
- If several jobs ready to run at the same time [**CPU scheduling]**
- If processes don't fit in memory, **swapping** moves them in and out to run
- **Virtual memory** allows execution of processes not completely in memory

❖ Dual-Mode and Multimode Operation

The following are the modes

- **User Mode:**

Operating system running a user application such as handling a text editor. The transition from user mode to kernel mode occurs when the application requests the help of operating system or an interrupt or a system call occurs. The mode bit is set to 1 in the user mode. It is changed from 1 to 0 when switching from user mode to kernel mode.

- **Kernel Mode**

The system starts in kernel mode when it boots and after the operating system is loaded, it executes applications in user mode. There are some privileged instructions that can only be executed in kernel mode. These are interrupt instructions, input output management etc. If the privileged instructions are executed in user mode, it is illegal and a trap is generated. The mode bit is set to 0 in the kernel mode. It is changed from 0 to 1 when switching from kernel mode to user mode.

The concept of modes of operation in operating system can be extended beyond the dual mode. Known as the **multimode** system. In those cases more than 1 bit is used by the CPU to set and handle the mode.

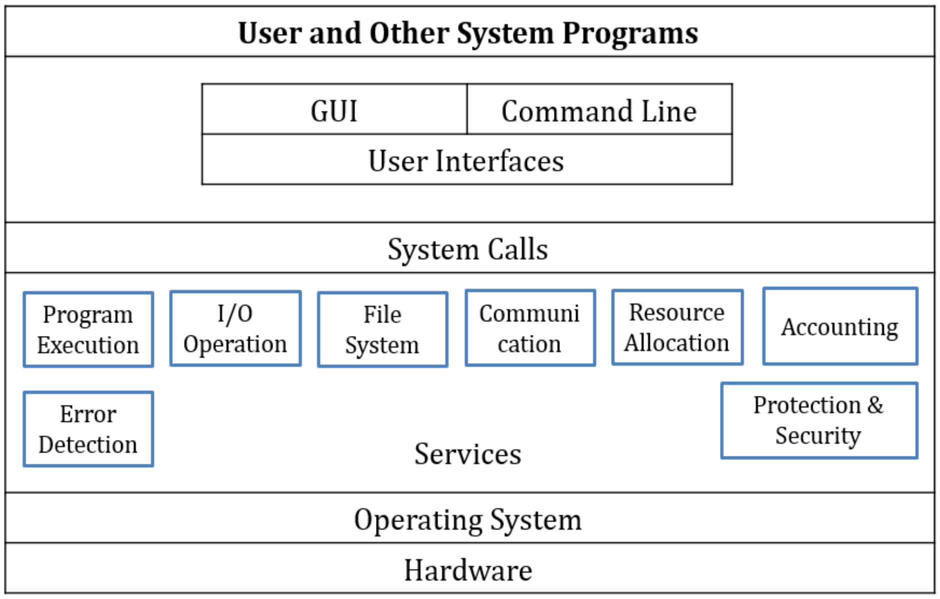
❖ **Timer**

A user program cannot get stuck in an infinite loop or to fail to call system services and never return control to the operating system. A timer is used to accomplish this goal. A timer can be set to interrupt the computer after a specified period. A **variable timer** is generally implemented by a fixed-rate clock and a counter. The operating system sets the counter. Every time the clock ticks, the counter is decremented. When the counter reaches 0, an interrupt occurs.

5. OPERATING SYSTEM SERVICES

Furthermore, the operating system, in one form or another, provides certain services to the computer system.

- ❖ User Interface
- ❖ Program Execution
- ❖ I/O Operations
- ❖ File System Manipulation
- ❖ Communications
- ❖ Error Detection
- ❖ Resource Allocation
- ❖ Accounting
- ❖ Protection and Security

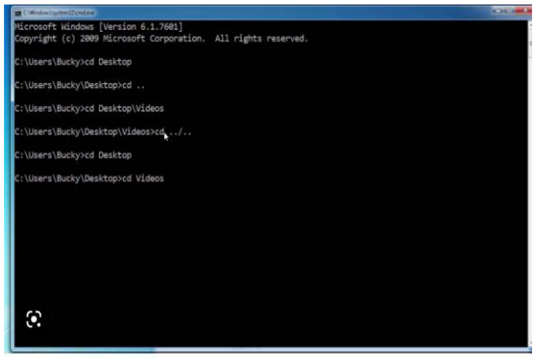


❖ **User Interface**

Almost all operating systems have a **user interface** (**UI**). Two types of User Interface are Command Based Interface and Graphical User Interface

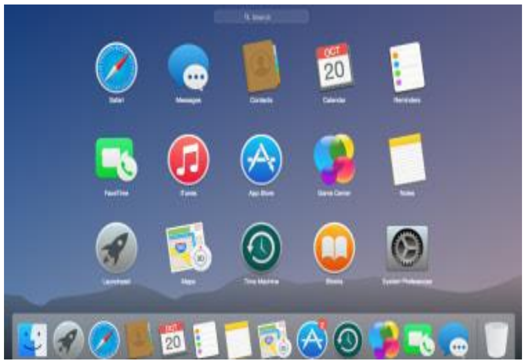
Command Based Interface

Requires a user to enter the commands to perform different tasks like creating, opening, editing or deleting a file, etc. The user has to remember the names of all such programs or specific commands which the operating system supports. The primary input device used by the user for command based interface is the keyboard. Command-based interface is often less interactive and usually allows a user to run a single program at a time. Examples of operating systems with command-based interfaces include MS-DOS and Unix.



```
Microsoft Windows [Version 6.1.7601]
Copyright (c) 2009 Microsoft Corporation.  All rights reserved.

C:\Users\Bucky>cd Desktop

C:\Users\Bucky\Desktop>cd ..

C:\Users\Bucky>cd Desktop\Videos

C:\Users\Bucky\Desktop\Videos>cd ../..

C:\Users\Bucky>cd Desktop

C:\Users\Bucky\Desktop>cd Videos
```

Command Based Interface



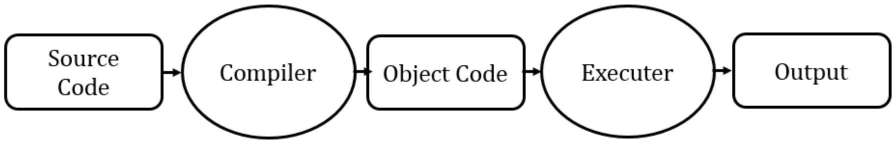
Graphical User Interface

Graphical User Interface (**GUI**)

The interface is a window system with a mouse that serves as a pointing device to direct I/O, choose from menus, and make selections and a keyboard to enter text. Mobile systems such as phones and tablets provide a **touch-screen interface**, enabling users to slide their fingers across the screen or press buttons on the screen to select choices.

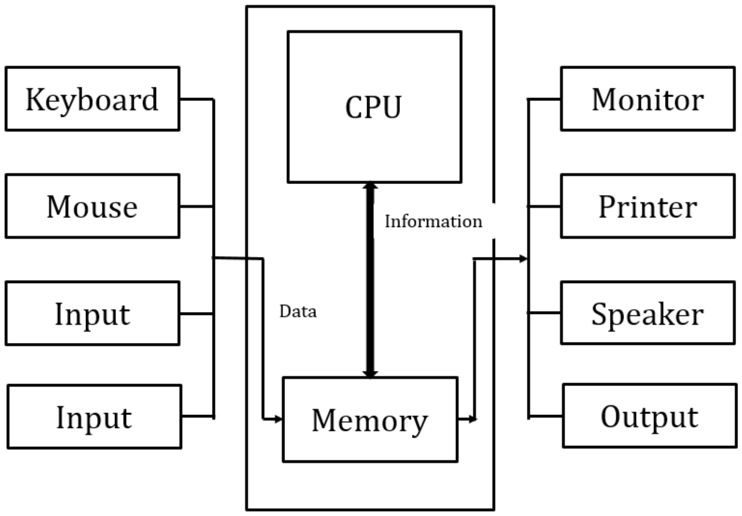
❖ **Program Execution**:

The OS is in charge of running all types of programs, whether they are user or system programs. The operating system makes use of a variety of resources to ensure that all types of functions perform smoothly.



❖ **Input/Output Operations**:

The operating system is in charge of handling various types of inputs, such as those from the keyboard, mouse, and desktop. Regarding all types of inputs and outputs, the operating system handles all interfaces in the most appropriate manner.



For instance, the nature of all types of peripheral devices, such as mice or keyboards, differs, and the operating system is responsible for transferring data between them.

❖ **File System Manipulation**:

The OS is in charge of deciding where data or files should be stored, such as on a floppy disk, hard disk, or pen drive. The operating system determines how data should be stored and handled.

❖ **Communications :**

There are many circumstances in which one process needs to exchange information with another process. Such communication may occur between processes that are executing on the same computer or between processes that are executing on different computer systems tied together by a network. Communications may be implemented via **shared memory**, in which two or more processes read and write to a shared section of memory, or **message passing**, in which packets of information in predefined formats are moved between processes by the operating system.

- **Error Detection:**

The operating system needs to be detecting and correcting errors constantly. Errors may occur in the CPU and memory hardware (such as a memory error or a power failure), in I/O devices (such as a parity error on disk, a connection failure on a network, or lack of paper in the printer), and in the user program (such as an arithmetic overflow or an attempt to access an illegal memory location). For each type of error, the operating system should take the appropriate action to ensure correct and consistent computing. Sometimes, it has no choice but to halt the system. At other times, it might terminate an error-causing process or return an error code to a process for the process to detect and possibly correct.

- **Resource Allocation:**

The operating system guarantees that all available resources are properly utilized by determining which resource should be used by whom and for how long. The operating system makes all of the choices.

- **Accounting:**

The operating system keeps track of all the functions that are active in the computer system at any one time. The operating system keeps track of all the facts, including the types of mistakes that happened.

- **Protection and Security :**

The operating system is in charge of making the most secure use of all the data and resources available on the machine. Any attempt by an external resource to obstruct data or information must be foiled by the operating system.

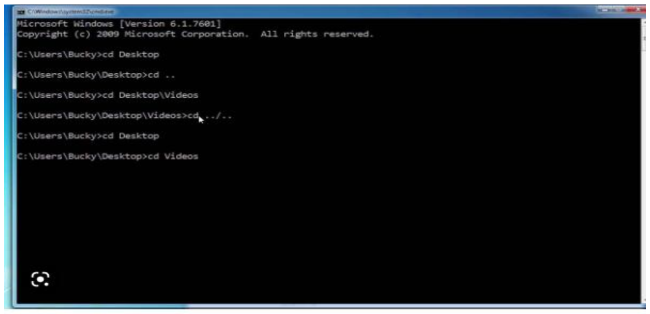
6. USER AND OPERATING SYSTEM INTERFACE

There are different types of user interfaces each of which provides a different functionality:

- Command Based Interface
- Graphical User Interface
- Touch Based Interface
- Voice Based Interface
- Gesture Based Interface

- **Command Based Interface**

Command based interface requires a user to enter the commands to perform different tasks like creating, opening, editing or deleting a file, etc. The user has to remember the names of all such programs or specific commands which the operating system supports. The primary input device used by the user for command based interface is the keyboard. Command-based interface is often less interactive and usually allows a user to run a single program at a time. Examples of operating systems with command-based interfaces include MS-DOS and Unix.



```
Microsoft Windows [Version 6.1.7601]
Copyright (c) 2009 Microsoft Corporation.  All rights reserved.

C:\Users\Bucky>cd Desktop

C:\Users\Bucky\Desktop>cd ..

C:\Users\Bucky>cd Desktop\Videos

C:\Users\Bucky\Desktop\Videos>cd ../..

C:\Users\Bucky>cd Desktop

C:\Users\Bucky\Desktop>cd Videos
```

Command Based Interface

Graphical User Interface

❖ **Graphical User Interface**

Users run programs or give instructions to the computer in the form of icons, menus and other visual options. Icons usually represent files and programs stored on the computer and windows represent running programs that the user has launched through the operating system. The input devices used to interact with the GUI commonly include the mouse and the keyboard. Examples of operating systems with GUI interfaces include Microsoft Windows, Ubuntu, Fedora and Macintosh, among others.

❖ **Touch Based Interface**

Today smartphones, tablets, and PCs allow users to interact with the system simply using the touch input. Using the touchscreen, a user provides inputs to the operating system, which are interpreted by the OS as commands like opening an app, closing an app, dialing a number, scrolling across apps, etc.
Examples of popular operating systems with touch-based interfaces are Android and iOS. Windows 8.1 and 10 also support touch-based interfaces on touchscreen devices.

Touch Based Interface

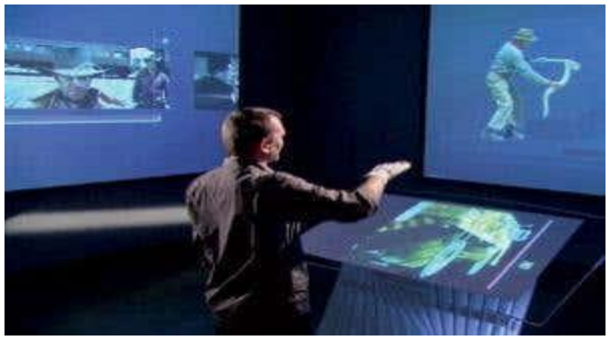
Voice Based Interface

❖ **Voice Based Interface**

Modern computers have been designed to address the needs of all types of users including people with special needs and people who want to interact with computers or smartphones while doing some other task. For users who cannot use input devices like the mouse, keyboard, and touchscreens, modern operating systems provide other means of human-computer interaction. Users today can use voice-based commands to make a computer work in the desired way. Some operating systems which provide voice-based control to users include iOS (Siri), Android (Google Now or "OK Google"), Microsoft Windows 10 (Cortana), and so on.

❖ **Gesture Based Interface**

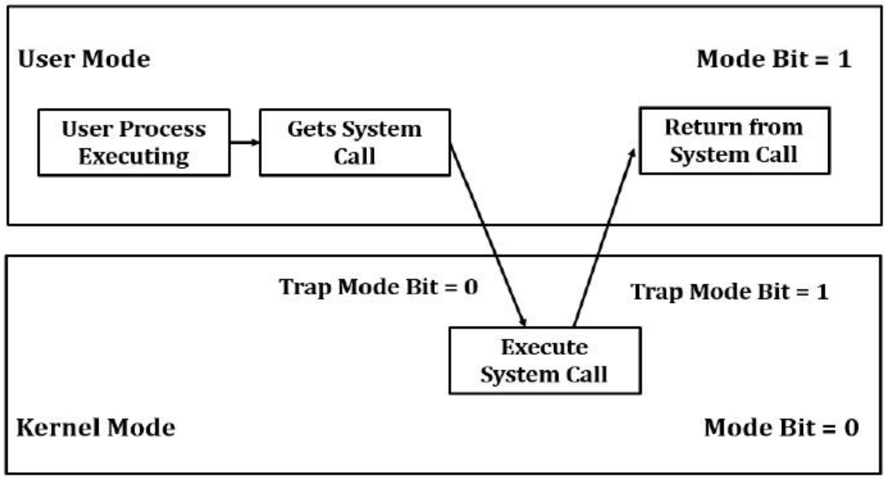
Some smartphones based on Android and iOS as well as laptops let users interact with the devices using gestures like waving, tilting, eye motion, and shaking. This technology is evolving faster and it has promising potential for application in gaming, medicine, and other areas.



7. SYSTEM CALLS

A **system call** is a mechanism that provides the interface between a process and the operating system. It is a programmatic method in which a computer program requests a service from the kernel of the OS. System call offers the services of the operating system to the user programs via API (Application Programming Interface). System calls are the only entry points for the kernel system.

Working of System Call



Step 1) The processes executed in the user mode till the time a system call interrupts it.
Step 2) After that, the system call is executed in the kernel-mode on a priority basis.
Step 3) Once system call execution is over, control returns to the user mode.,
Step 4) The execution of user processes resumed in Kernel mode.

Need of System Call

- Reading and writing from files demand system calls.
- If a file system wants to create or delete files, system calls are required.
- System calls are used for the creation and management of new processes.
- Network connections need system calls for sending and receiving packets.
- Access to hardware devices like scanner, printer, need a system call.

Example of how system calls are used.

For Example
Writing a simple program to read data from one file and copy them to another file.
The first input that the program will need is the names of the two files: the input file and the output file. These names can be specified in many ways, depending on the operating-system design.
One approach is to pass the names of the two files as part of the command
For example, the UNIX cp command:

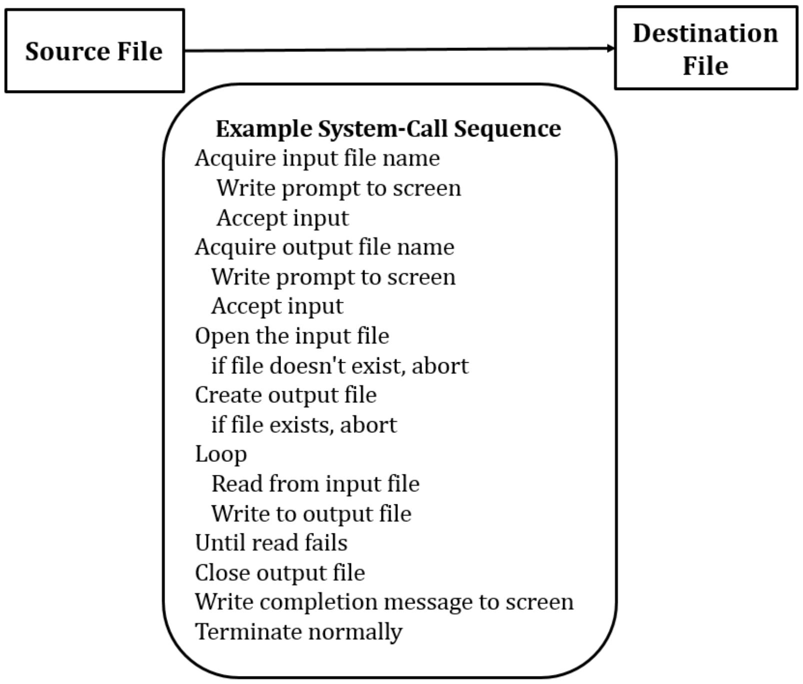
```
cp in.txt out.txt
```

This command copies the input file in.txt to the output file out.txt.

A second approach is for the program to ask the user for the names.

In an interactive system, this approach will require a sequence of system calls, first to write a prompting message on the screen and then to read from the keyboard the characters that define the two files. On mouse-based and icon-based systems, a menu of file names is usually displayed in a window. The user can then use the mouse to select the source name, and a window can be opened for the destination name to be specified. This sequence requires many I/O system calls. Once the two file names have been obtained, the program must open the input file and create and open the output file. Each of these operations requires another system call. Possible error conditions for each system call must be handled. For example, when the program tries to open the input file, it may find that there is no file of that name or that the file is protected against access.

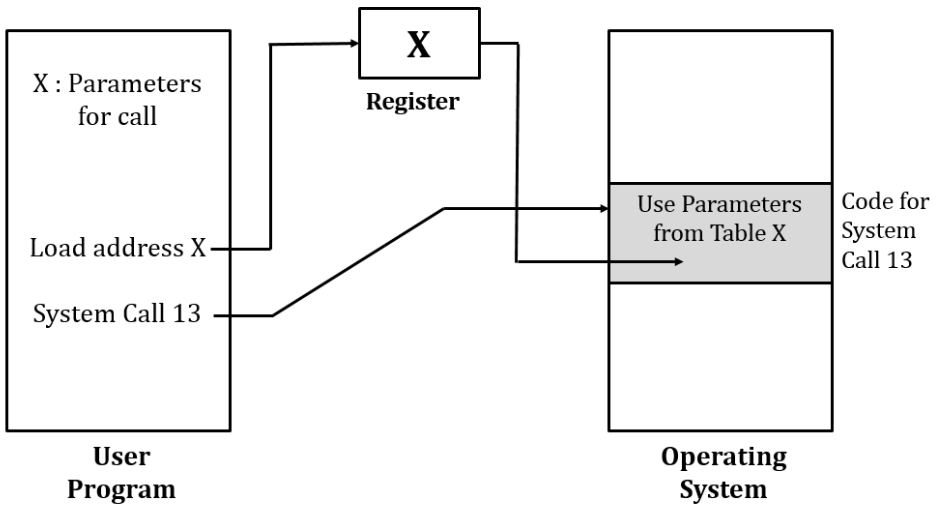
In these cases, the program should output an error message (another sequence of system calls) and then terminate abnormally (another system call). If the input file exists, then we must create a new output file. We may find that there is already an output file with the same name. This situation may cause the program to abort (a system call), or we may delete the existing file (another system call) and create a new one (yet another system call). Another option, in an interactive system, is to ask the user (via a sequence of system calls to output the prompting message and to read the response from the terminal) whether to replace the existing file or to abort the program.



When both files are set up, we enter a loop that reads from the input file (a system call) and writes to the output file (another system call). Each read and write must return status information regarding various possible error conditions. On input, the program may find that the end of the file has been reached or that there was a hardware failure in the read (such as a parity error). The write operation may encounter various errors, depending on the output device (for example, no more available disk space).

Finally, after the entire file is copied, the program may close both files (two system calls), write a message to the console or window (more system calls), and finally terminate normally (the final system call).

Passing of Parameters as a table



Rules for passing Parameters for System Call

Here are general common rules for passing parameters to the System Call:

- Parameters should be pushed on or popped off the stack by the operating system.
- Parameters can be passed in registers. For five or fewer parameters, registers are used.
- More than five parameters, the block method is used. The parameters are generally stored in a block, or table, in memory, and the address of the block is passed as a parameter in a register

Types of System calls

Here are the five types of System Calls in OS:

- ❖ Process Control
- ❖ File Management
- ❖ Device Management
- ❖ Information Maintenance
- ❖ Communications

Process Control

This system calls perform the task of process creation, process termination, etc.
Functions:

- End and Abort
- Load and Execute
- Create Process and Terminate Process
- Wait and Signal Event
- Allocate and free memory

File Management

File management system calls handle file manipulation jobs like creating a file, reading, and writing, etc.

Functions:

- Create a file
- Delete file
- Open and close file
- Read, write, and reposition
- Get and set file attributes

Device Management

Device management does the job of device manipulation like reading from device buffers, writing into device buffers, etc.

Functions:

- Request and release device
- Logically attach/ detach devices
- Get and Set device attributes

Information Maintenance

It handles information and its transfer between the OS and the user program.

Functions:

- Get or set time and date
- Get process and device attributes

Communication:

These types of system calls are specially used for interprocess communications.

Functions:

- Create, delete communications connections
- Send, receive message
- Help OS to transfer status information
- Attach or detach remote devices

Important System Calls Used in OS

wait()

A process needs to wait for another process to complete its execution. This occurs when a parent process creates a child process, and the execution of the parent process remains suspended until its child process executes. The suspension of the parent process automatically occurs with a wait() system call. When the child process ends execution, the control moves back to the parent process.

fork()

Processes use this system call to create processes that are a copy of themselves. With the help of this system Call parent process creates a child process, and the execution of the parent process will be suspended till the child process executes.

exec()

This system call runs when an executable file in the context of an already running process that replaces the older executable file. However, the original process identifier remains as a new process is not built, but stack, data, head, data, etc. are replaced by the new process.

kill():

The kill() system call is used by OS to send a termination signal to a process that urges the process to exit. However, a kill system call does not necessarily mean killing the process and can have various meanings.

exit():

The exit() system call is used to terminate program execution. Specially in the multi-threaded environment, this call defines that the thread execution is complete. The OS reclaims resources that were used by the process after the use of exit() system call.

8. SYSTEM PROGRAMS

System programs provide a convenient environment for program development and execution. It can be divided into:

- ❖ File manipulation
- ❖ Status information
- ❖ File modification
- ❖ Programming language support
- ❖ Program loading and execution
- ❖ Communications
- ❖ Application programs

❖ **File management**.

These programs create, delete, copy, rename, print, list, and generally access and manipulate files and directories.

❖ **Status information**.

Some programs simply ask the system for the date, time, amount of available memory or disk space, number of users, or similar status information. Others are more complex, providing detailed performance, logging, and debugging information. Typically, these programs format and print the output to the terminal or other output devices or files or display it in a window of the GUI. Some systems also support a **registry**, which is used to store and retrieve configuration information.

❖ **File modification:** .

Several text editors may be available to create and modify the content of files stored on disk or other storage devices. There may also be special commands to search contents of files or perform transformations of the text.

❖ **Programming-language support**:

Compilers, assemblers, debuggers, and interpreters for common programming languages (such as C, C++, Java, and Python) are often provided with the operating system or available as a separate download.

❖ **Program loading and execution**:

Once a program is assembled or compiled, it must be loaded into memory to be executed. The system may provide absolute loaders, relocatable loaders, linkage editors, and overlay loaders. Debugging systems for either higher-level languages or machine language are needed as well.

❖ **Communications**:

These programs provide the mechanism for creating virtual connections among processes, users, and computer systems. They allow users to send messages to one another's screens, to browse web pages, to send e-mail messages, to log in remotely, or to transfer files from one machine to another.

❖ **Background services**:

All general-purpose systems have methods for launching certain system-program processes at boot time. Some of these processes terminate after completing their tasks, while others continue to run until the system is halted. Constantly running system-program processes are known as services, subsystems, or daemons

9. OPERATING SYSTEM DESIGN AND IMPLEMENTATION

DESIGN GOALS

The first problem in designing a system is to define goals and specifications. At the highest level, the design of the system will be affected by the choice of hardware and the type of system: traditional desktop/laptop, mobile, distributed, or real time. Beyond this highest design level, the requirements may be much harder to specify.
The requirements can, however, be divided into two basic groups:

User goals and **system goals**.

❖ User goals
Convenience and efficiency , Easy to learn , Reliable , Safe and Fast

❖ System goals
Easy to design, implement, and maintain , Flexible, reliable, error-free, and efficient

Mechanisms and Policies

Mechanisms determine ***how*** to do something; policies determine ***what*** will be done.
For example, the timer construct is a mechanism for ensuring CPU protection, but deciding how long the timer is to be set for a particular user is a policy decision.

- Early OS in assembly language, Now C, C++
- Using emulators of the target hardware, particularly if the real hardware is unavailable (e.g. not built yet), or not a suitable platform for development, (e.g. smart phones, game consoles, or other similar devices.)
- Android
  Library : C, C++
  Application Frameworks : JAVA

10. OPERATING SYSTEM STRUCTURE

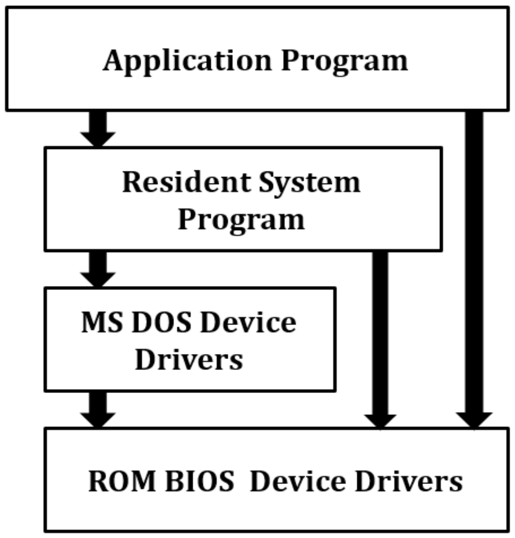
Operating system can be implemented with the help of various structures. The structure of the OS depends mainly on how the various common components of the operating system are interconnected and melded into the kernel.

Depending on this we have following structures of the operating system:

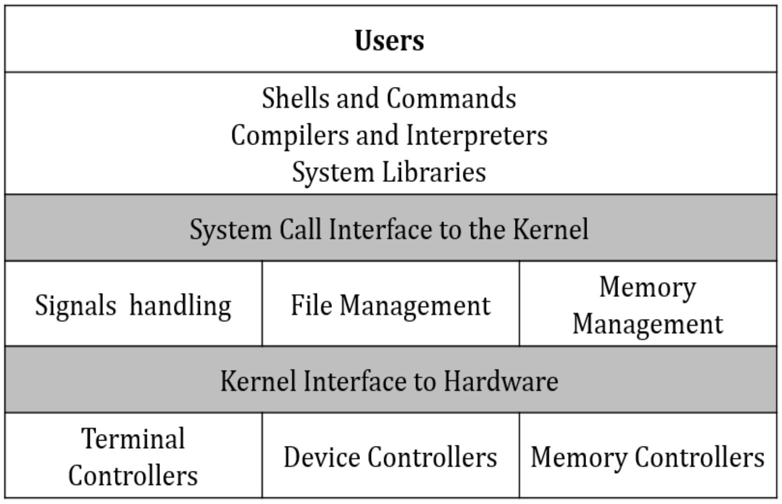
- Monolithic Structure
- Layered Approach
- Microkernels
- Modules
- Hybrid Systems
  - macOS and iOS
  - Android

❖ Monolithic structure:

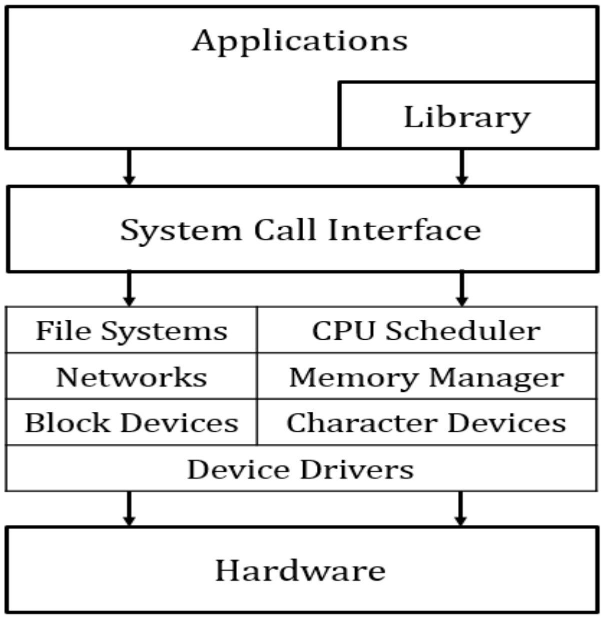
Such operating systems do not have well defined structure and are small, simple and limited systems. The interfaces and levels of functionality are not well separated. MS-DOS is an example of such operating system. In MS-DOS application programs are able to access the basic I/O routines. These types of operating system cause the entire system to crash if one of the user programs fails. Diagram of the structure of MS-DOS is shown below.



An example of such limited structuring is the original UNIX operating system, which consists of two separable parts: the kernel and the system programs. The kernel is further separated into a series of interfaces and device drivers, which have been added and expanded over the years as UNIX has evolved. Everything below the system-call interface and above the physical hardware is the kernel. The kernel provides the file system, CPU scheduling, memory management, and other operating system functions through system calls. Taken in sum, that is an enormous amount of functionality to be combined into one single address space. UNIX Structure is shown below

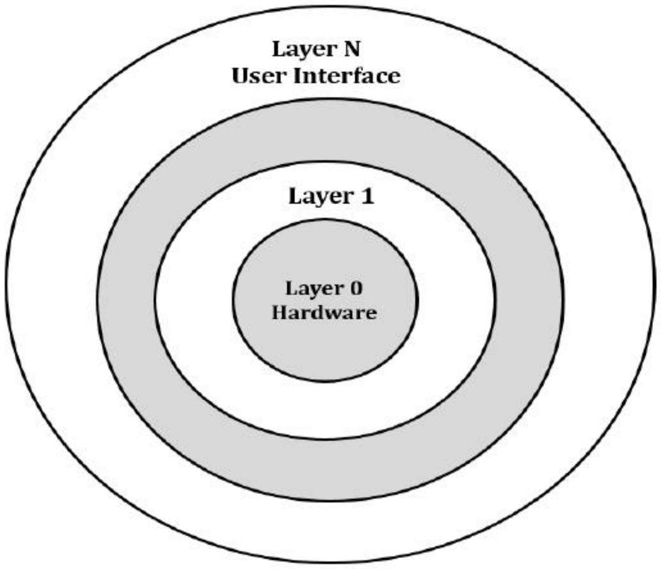


The Linux operating system is based on UNIX shown in the figure below. Applications typically use the glibc standard C library when communicating with the system call interface to the kernel. The Linux kernel is monolithic in that it runs entirely in kernel mode in a single address space, it does have a modular design that allows the kernel to be modified during run time.



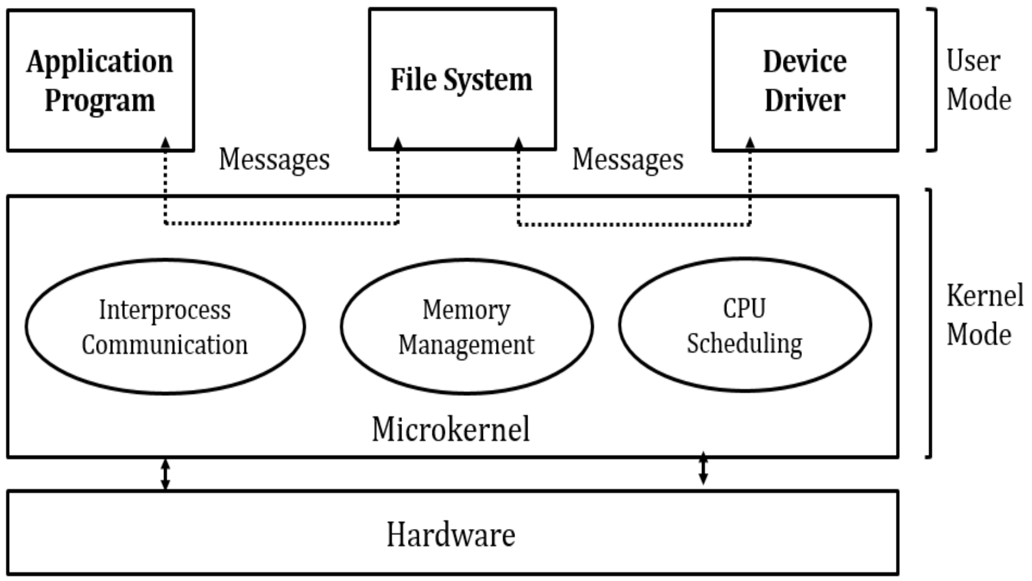
❖ Layered Approach

An OS can be broken into pieces and retain much more control on system. In this structure the OS is broken into number of layers (levels). The bottom layer (layer 0) is the hardware and the topmost layer (layer N) is the user interface. These layers are so designed that each layer uses the functions of the lower level layers only. This simplifies the debugging process as if lower level layers are debugged and an error occurs during debugging then the error must be on that layer only as the lower level layers have already been debugged. The main disadvantage of this structure is that at each layer, the data needs to be modified and passed on which adds overhead to the system. Moreover careful planning of the layers is necessary as a layer can use only lower level layers. UNIX is an example of this structure.



❖ **Micro-kernel:**

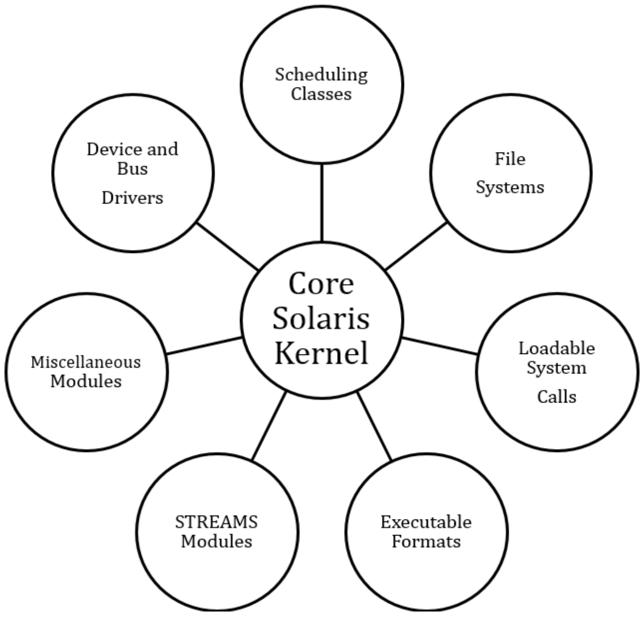
This structure designs the operating system by removing all non-essential components from the kernel and implementing them as system and user programs. This result in a smaller kernel called the micro-kernel.



Advantages of this structure are that all new services need to be added to user space and does not require the kernel to be modified. Thus it is more secure and reliable as if a service fails then rest of the operating system remains untouched. Mac OS is an example of this type of OS.

❖ Modular structure or approach:

The best approach for an OS. It involves designing of a modular kernel. The kernel has only set of core components and other services are added as dynamically loadable modules to the kernel either during run time or boot time. It resembles layered structure due to the fact that each kernel has defined and protected interfaces but it is more flexible than the layered structure as a module can call any other module. For example Solaris OS is organized as shown in the figure.



❖ Hybrid Systems

The Apple macOS operating system and the two mobile operating systems—iOS and Android.

macOS and iOS

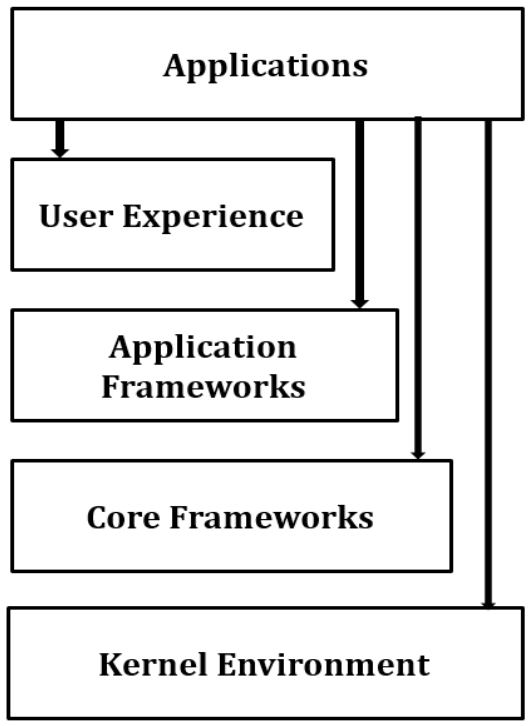
Apple's macOS operating system is designed to run primarily on desktop and laptop computer systems, whereas iOS is a mobile operating system designed for the iPhone smartphone and iPad tablet computer. Highlights of the various layers include the following:

- **User experience layer**. This layer defines the software interface that allows users to interact with the computing devices. macOS uses the ***Aqua*** user interface, which is designed for a mouse or trackpad, whereas iOS uses the ***Springboard*** user interface, which is designed for touch devices.

- **Application frameworks layer**. This layer includes the ***Cocoa*** and ***Cocoa Touch*** frameworks, which provide an API for the Objective-C and Swift programming languages. The primary difference between Cocoa and Cocoa Touch is that the former is used for developing macOS applications, and the latter by iOS to provide support for hardware features unique to mobile devices, such as touch screens.

- **Core frameworks**. This layer defines frameworks that support graphics and media including, Quicktime and OpenGL.

- **Kernel environment**. This environment, also known as **Darwin**, includes the Mach microkernel and the BSD UNIX kernel.



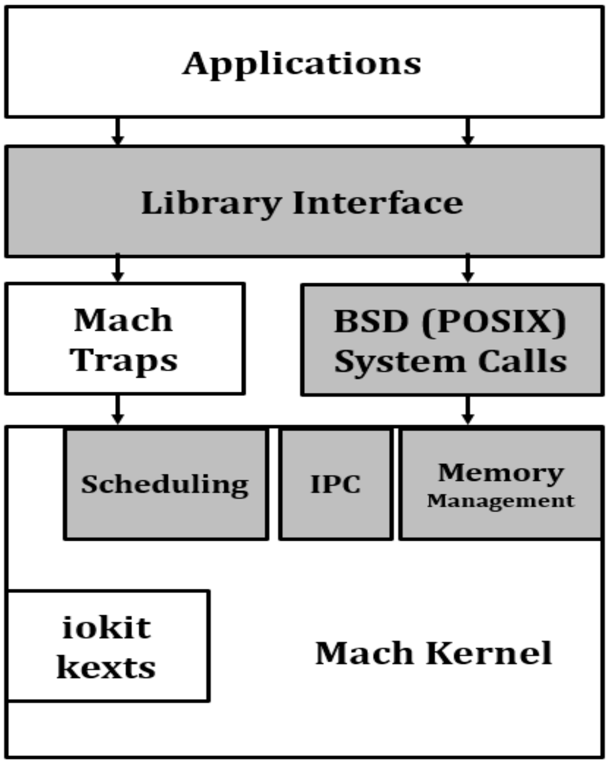
Applications can be designed to take advantage of user-experience features or to bypass them and interact directly with either the application framework or the core framework. Additionally, an application can forego frameworks entirely and communicate directly with the kernel environment.
Some significant distinctions between macOS and iOS include the following:

- Because macOS is intended for desktop and laptop computer systems, it is compiled to run on Intel architectures. iOS is designed for mobile devices and thus is compiled for ARM-based architectures. Similarly, the iOS kernel has been modified somewhat to address specific features and needs of mobile systems, such as power management and aggressive memory management. Additionally, iOS has more stringent security settings than macOS.
- The iOS operating system is generally much more restricted to developers than macOS and may even be closed to developers. For example, iOS restricts access to POSIX and BSD APIs on iOS, whereas they are openly available to developers on macOS.

Darwin OS

Darwin OS is a layered system that consists primarily of the Mach microkernel and the BSD UNIX kernel. Darwin's structure is shown below

Darwin provides ***two*** system-call interfaces: Mach system calls (known as **traps**) and BSD system calls (which provide POSIX functionality). The interface to these system calls is a rich set of libraries that includes not only the standard C library but also libraries that provide networking, security, and programming language support.



Beneath the system-call interface, Mach provides fundamental operating system services, including memory management, CPU scheduling, and inter process communication (IPC) facilities such as message passing and remote procedure calls (RPCs). Much of the functionality provided by Mach is available through **kernel abstractions**, which include tasks (a Mach process), threads, memory objects, and ports (used for IPC). As an example, an application may create a new process using the BSD POSIX fork() system call. Mach will, in turn, use a task kernel abstraction to represent the process in the kernel.

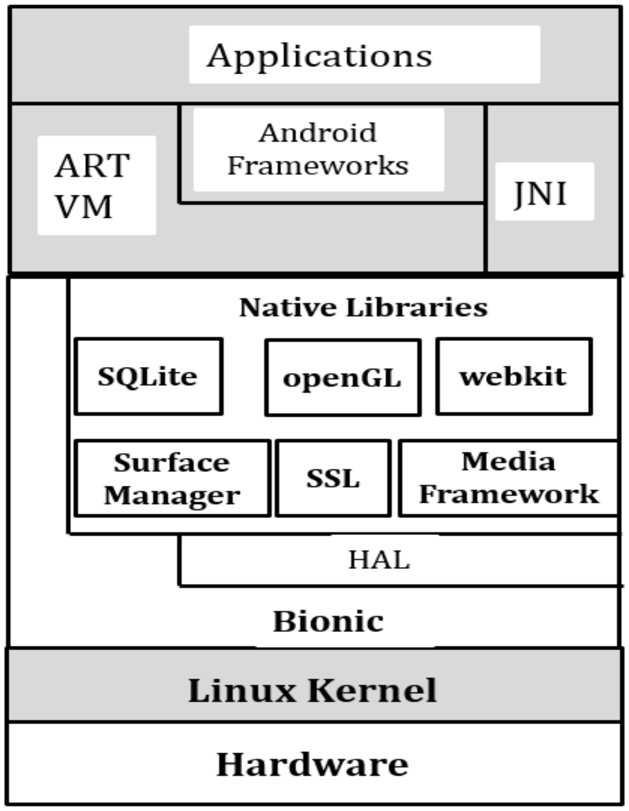
In addition to Mach and BSD, the kernel environment provides an I/O kit for development of device drivers and dynamically loadable modules (which macOS refers to as **kernel extensions**, or **kexts**).

Android

Developed for Android smartphones and tablet computers. Whereas iOS is designed to run on Apple mobile devices and is close-sourced, Android runs on a variety of mobile platforms and is open sourced, partly explaining its rapid rise in popularity. Android is similar to iOS in that it is a layered stack of software that provides a rich set of frameworks supporting graphics, audio, and hardware features. These features, in turn, provide a platform for developing mobile applications that run on a multitude of Android-enabled devices.

Software designers for Android devices develop applications in the Java language, but they do not generally use the standard Java API. Google has designed a separate Android API for Java development. Java applications are compiled into a form that can execute on the Android RunTime ART, a virtual machine designed for Android and optimized for mobile devices with limited

memory and CPU processing capabilities. Java programs are first compiled to a Java bytecode .class file and then translated into an executable .dex file. Whereas many Java virtual machines perform just-in-time (JIT) compilation to improve application efficiency, ART performs **ahead-of-time** (**AOT**) compilation
The structure of Android appears is shown below



.dex files are compiled into native machine code when they are installed on a device, from which they can execute on the ART. AOT compilation allows more efficient application execution as well as reduced power consumption, features that are crucial for mobile systems.

Programs written using Java native interface JNI are generally not portable from one hardware device to another. The set of native libraries available for Android applications includes frameworks for developing web browsers (webkit), database support (SQLite), and network support, such as secure sockets (SSLs). Android can run on an almost unlimited number of hardware devices, Google has chosen to abstract the physical hardware through the hardware abstraction layer, or HAL. By abstracting all hardware, such as the camera, GPS chip, and other sensors, the HAL provides applications with a consistent view independent of specific hardware. This feature, of course, allows developers to write programs that are portable across different hardware platforms.

The standard C library used by Linux systems is the GNU C library (glibc). Google instead developed the **Bionic** standard C library for Android. Not only does Bionic have a smaller memory footprint than glibc, but it also has been designed for the slower CPUs that characterize mobile devices. At the bottom of Android's software stack is the Linux kernel. Google has modified the Linux kernel used in Android in a variety of areas to support the special needs of mobile systems, such as power management. It has also made changes in memory management and allocation.

UNIT-II
PROCESS MANAGEMENT

1) Processes
2) Process concept
3) Process scheduling
4) Operations on process
5) Inter-process communication
6) CPU scheduling
7) Scheduling criteria
8) Scheduling algorithm
9) Multiple-processor scheduling.
10) Real-time scheduling
11) Threads
12) overview
13) Multithreading models
14) Threading issues
15) Process synchronization
16) The critical-section Problem.
17) Synchronization Hardware.
18) Mutex locks
19) Semaphores.
20) Classic problems of synchronization.
21) critical regions.
22) Monitors
23) Deadlocks
24) System models
25) Deadlock characterization
26) Methods for handling deadlocks
27) Deadlock Prevention
28) Deadlock avoidance
29) Deadlock detection
30) Recovery from deadlock.

(1) Process :

Early computer systems allowed only one Program to be executed at a time. This Program had complete control of the system and had access to all the System's resources.

In contrast, current-day computer systems allow multiple Programs to be loaded into memory and executed concurrently. Process is a program which is in execution. A Process is the unit of work in a modern time-sharing system. By switching the cpu between processes, the operating system can make the computer more productive.

(2) Process Concept :

An operating system executes a variety of Programs.

* Batch system - jobs
* Time Shared systems - User Programs (or) tasks.

We will use the terms job and Process almost interchangeably.

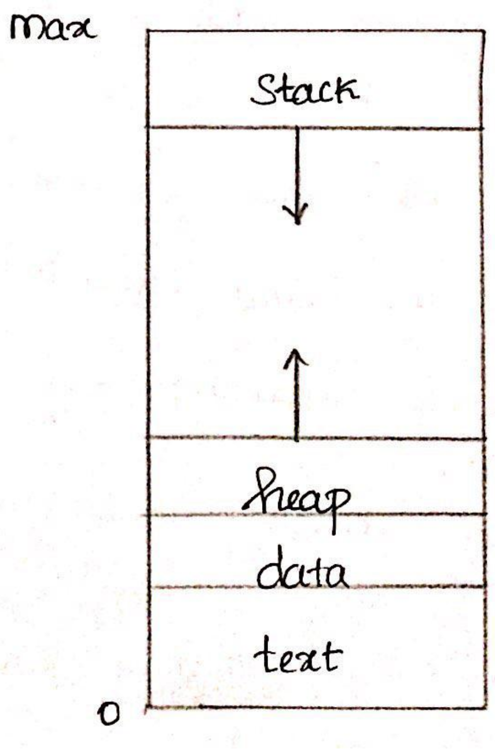
Process - It is a Program in execution (informal definition)

Program is passive entity stored on disk (executable file), Process is active.

Program becomes process when executable file loaded into memory.

* Execution of Program started via GUI, Command line entry of its name, etc.
* One program can be several processes.
* Consider multiple users executing the same program.
* In memory, a process consists of multiple parts:
    * Program code, also called text section
    * Current activity including
      Program counter
      Processor registers
    * Stack containing temporary data
      function parameters return address, local variables.
    * Data Section containing global variables.
    * Heap containing memory dynamically allocated during run time.

Process in memory.



We emphasize that a program by itself is not a Process; a Program is a passive entity, such as a file containing a list of instructions stored on disk (often called an executable file), whereas a Process is an active entity, with a program counter specifying the next instruction to execute and a set of associated resources.

A program becomes process, when an executable file is loaded into memory. Two common techniques for loading executable files are double-clicking an icon representing the executable file and entering the name of the executable file on the command line (as in prog.exe or a.out)

* Process state :

As a process executes, it changes state. The state of a process is defined in part by the current activity of that process. Each process may be in one of the following states;

New - The Process is being created.

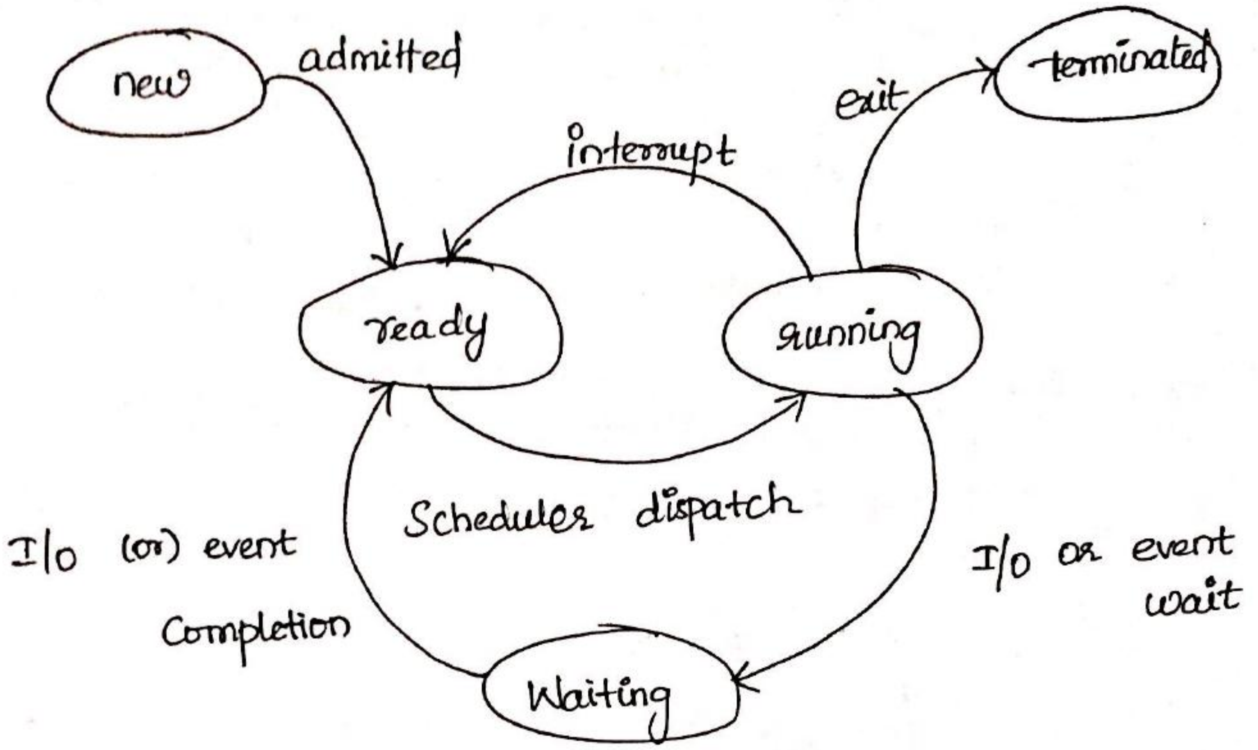
Running - Instructions are being executed.

Waiting - The Process is waiting for some event to occur.

Ready - The Process is waiting to be assigned to a processor.

Terminated - The Process has finished execution.

Diagram of Process state.



It is important to realize that only one Process can be running on any Processor at any instant. Many Processes may be ready and waiting, however.

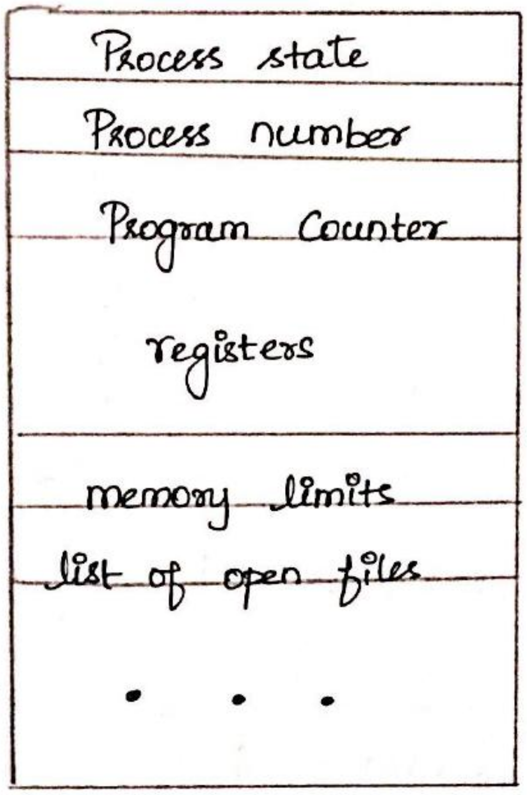
* Process control Block (PCB) :

Each Process is represented in the operating System by a Process control Block (PCB) - also called a task control block. It contains many pieces of information associated with a specific Process, Including these :

* Process state
* Program counter
* cpu register
* cpu - scheduling information.
* Memory - Management information.
* Accounting information.
* I/o status information.

"Data Structure maintained by OS for every Process is called PCB"

Process control Block - PCB



* Each information of Process is controlled by PCB, and each Process information is Stored in PCB, and Process ID.

* PCB is identified by an Integer called Process ID (PID)

Program State : The state may be new, ready, running, waiting, halted and so on.,

Program Counter : The counter indicates the address of the next instruction to be executed for this Process.

CPU registers : The registers vary in number and type, depending on the computer architecture. They include accumulators, index registers, stack pointers, and general-purpose registers, plus any condition-code information. Along with the Program counter, this state information must be saved when an interrupt occurs, to allow the Process to be continued correctly afterward. (temporary storage device)

CPU - Scheduling information :

This information includes a process priority, pointers to scheduling queues, and any other scheduling parameters.

Memory - Management information :

This information may include such information as the value of the base and limit registers, the page tables, or the segment tables, depending on the memory system used by the operating system.

Accounting information :

This information includes the amount of CPU and real time used, time limits, account numbers, job (or) process numbers, and so on.

I/O Status information :

This information includes the list of I/O devices allocated to the Process, a list of open files, and So on.

In brief, the PCB simply Serves as the repository for any information that may vary from Process to process.

*Threads :

The Process model discussed so far has implied that a process is a program that performs a single thread of execution. for eg., when a process is running a word - Processor Program, a single thread of instructions is being executed. Many morden OS have extended the Process concept to allow a Process to have multiple threads of execution and thus to perform more than one task at a time.

(3) Process scheduling :

The objective of multiprogramming is to have some Process running at all times, to maximize (cpu utilization.)

The objective of timesharing is to switch the cpu among processes so frequently that users can interact with each Program while it is running. To meet these objectives, the Process scheduler selects an available Process for program execution on the cpu. for a single-Processor system, there will never be more than one running process. If there are more Processes, the rest will have to wait until the cpu is free and can be rescheduled.

- ✓ * Scheduling Queues
- ✓ * Schedulers
- ✓ * Context switch

* Scheduling Queues :

Job queue - set of all process in the system.

Ready queue - set of all process residing in main memory, ready and waiting to execute.

Device queue - set of processes waiting for an I/o device process migrate among the various queues.

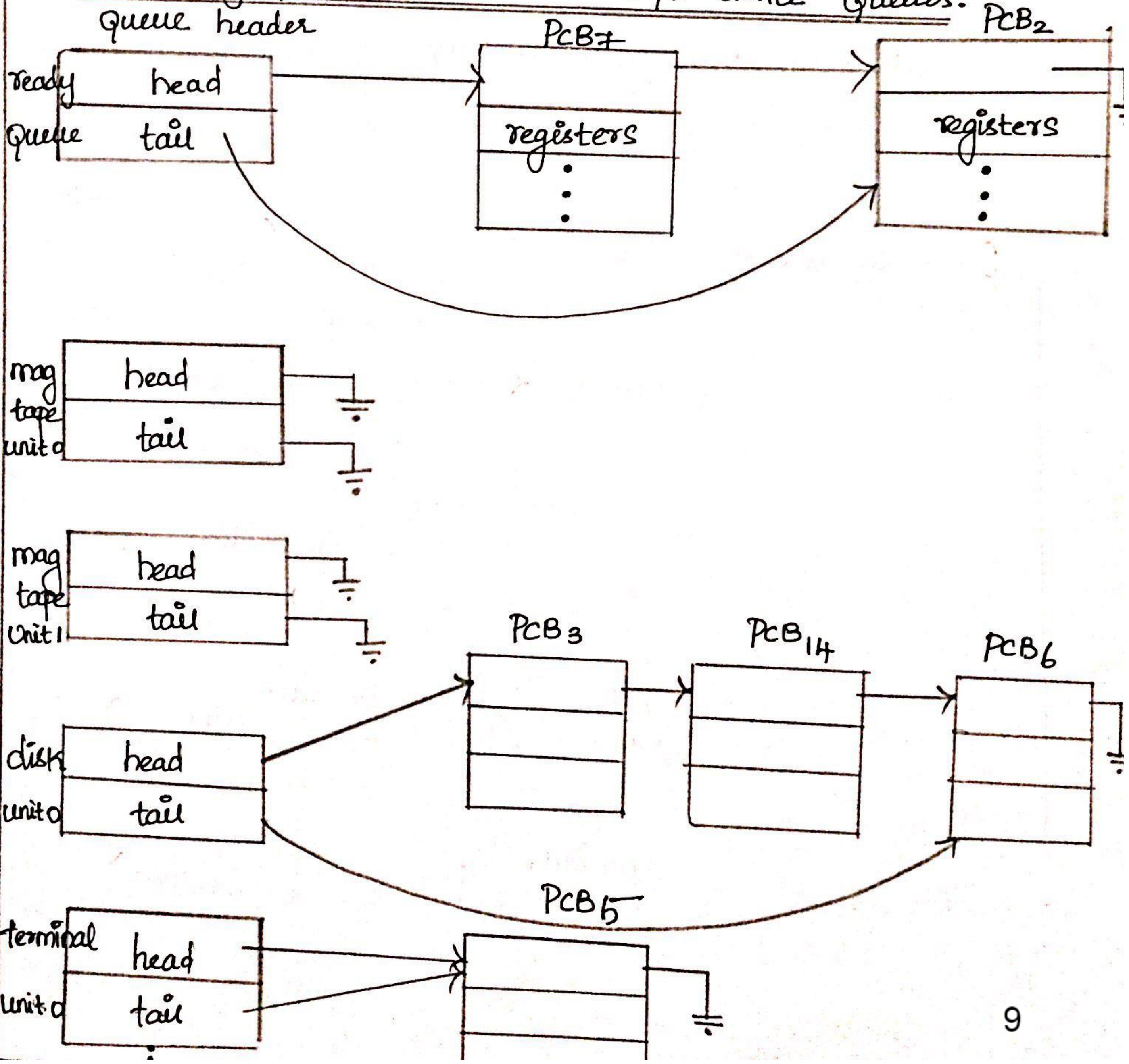
A common representation of Process scheduling is a Queueing diagram. Two types of Queues are Present : the ready queue and a set of device Queues. The Circles represent the resources that serve the queues, and the arrows indicate the flow of Processes in the System. A new Process is initially put in the ready Queue. It waits there until it is selected for execution (or) dispatched. Once the Process is allocated the CPU and is executing, one of several events could occur.

* The Process could issue an I/O request and then be placed in an I/O queue.

* The Process could create a new child process and wait for the child's termination.

* The Process could be removed forcibly from the CPU, as a result of an interrupt, and be put back in the ready Queue.

The ready Queue and various I/O device Queues.



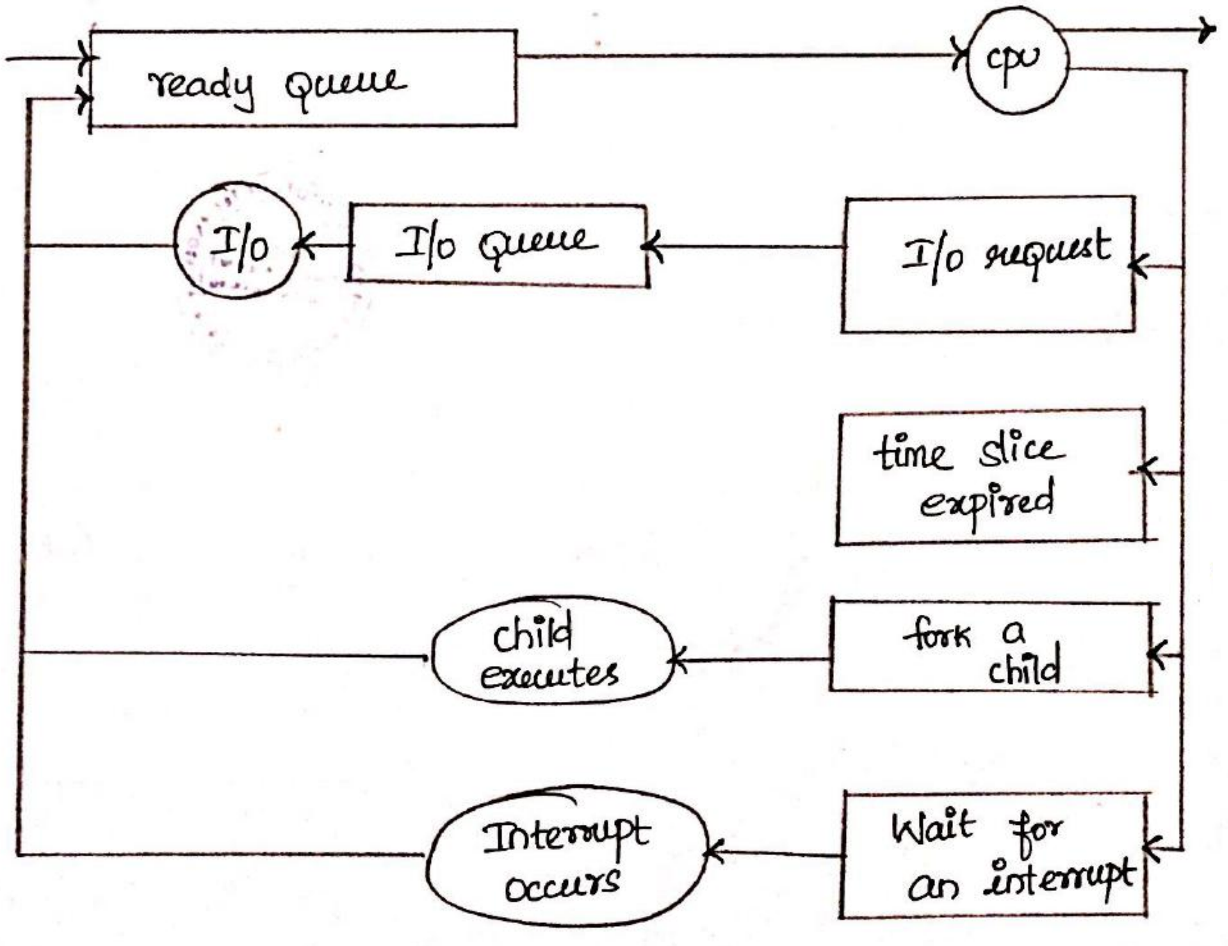
* Schedulers : [N/D-19]

⊛ Long-term scheduler (or job scheduler) - selects which Processes should be brought into the ready Queue.

⊛ Short-term scheduler (or cpu scheduler) - Selects which Process should be executed next and allocates cpu.

The operating system, must select for scheduling Purposes, Processes from these queues in some fashion. The selection Process is carried out by the appropriate scheduler.

Queueing diagram representation of Process scheduling.



The Primary distinction between these two schedulers lies in frequency execution. The short-term schedulers must select a new Process a new Process for the CPU frequently.

A Process may execute for only a few milliseconds before waiting for an I/O request. Often, the short-term scheduler executes at least once every 100 milliseconds. Because of the short time between executions, the short-term scheduler must be fast. If it takes 10 milliseconds to decide to execute a Process for 100 milliseconds, then 10/(100+10) = 9 Percent of the CPU is being used (wasted) simply for scheduling the work.

The long-term scheduler executes much less frequently; minutes may separate the creation of one new process and the next. The long-term scheduler controls the degree of multiprogramming (the number of Processes in memory).

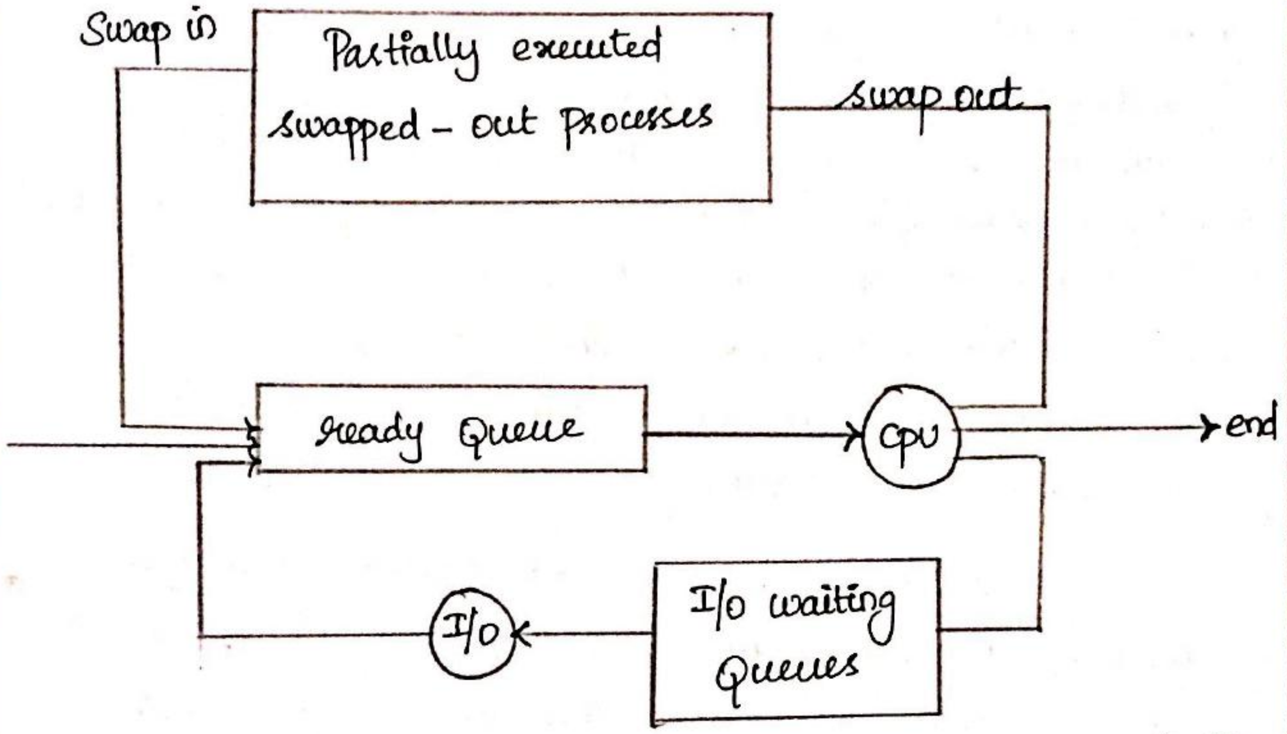
If the degree of multiprogramming is stable, then the average rate of process Creation must be equal to the average departure rate of processes leaving the system.

Processes can be described as either:

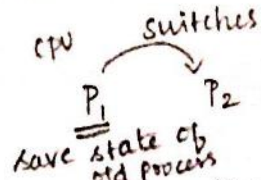
- I/O bound process - spends more time doing I/O than Computations, many short CPU bursts.

cpu - bound process - Spends more time doing computations, few very long cpu bursts.

Addition of medium - term scheduling to the queueing diagram.



Some operating systems, such as time - sharing systems, may introduce an additional, intermediate level of scheduling. The key idea behind a medium - term scheduler is that sometimes it can be advantageous to remove a process from memory (and from active contention for the cpu) and thus reduce the degree of multiprogramming.



* Context Switch :

* When cpu switches to another process, the system must save the state of the old process and load the saved state for the new process.

Execution :

* Parent and children execute ~~&~~ concurrently.

✓* Parent ~~and~~ ~~ch~~ waits until children terminate.

Address space :

* child duplicate of parent.

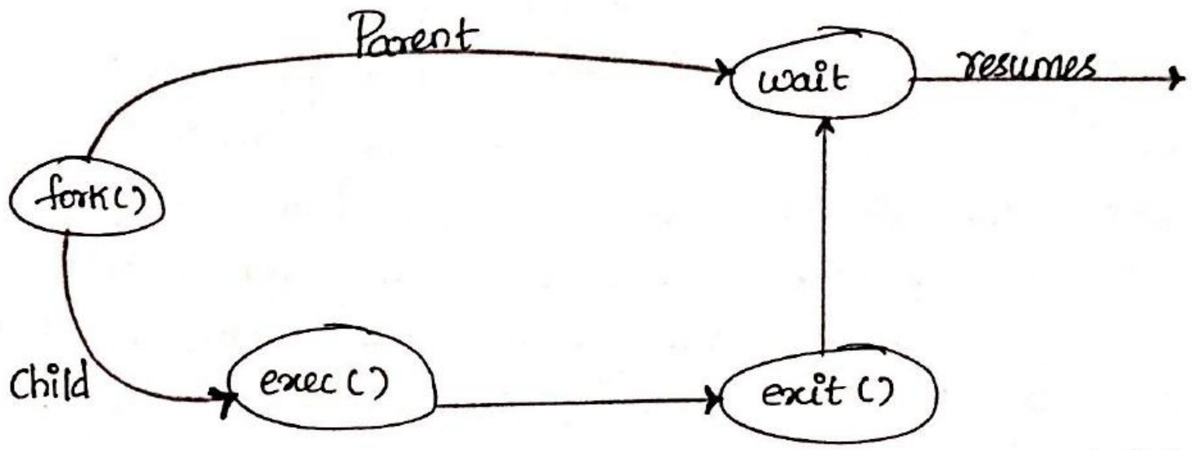
* child has a program loaded into it.

UNIX examples :

* fork system call create new process.

* exec system call used after a fork to replace the process' memory space with a new program.

Process creation using fork () system call.



When a process creates a new process, two possibilities exist in terms of execution.

1) The parent continues to execute concurrently with its children.

2) The Parent waits until some (or) all of its children have terminated.

* Context - switch time is overhead; the system does no useful work while switching.
* Time dependent on hardware support.

(4) Operations on Processes.

The Processes in most systems can execute concurrently, and they may be created and deleted dynamically. Thus, these systems must provide a mechanism for process creation and termination.

✓ * Process Creation (fork())
✓ * Process termination (exit())
*

* Process creation:

A Process may create Several new processes, via a Create - process system call, during the course of execution. The creating Process is called a parent process, and the new processes are called the children of that process. Each of these new processes may in turn create other processes, forming a tree of Processes.

Most operating systems (including UNIX and the Windows family of operating systems) identify Processes according to a unique Process identifier (or pid), which is typically an integer number.

Resource sharing:

→ Parent and children share all resources.
→ Children share subset of Parent's resources.
→ Parent and child share no resources.

There are also two possibilities in terms of the address space of the new process :

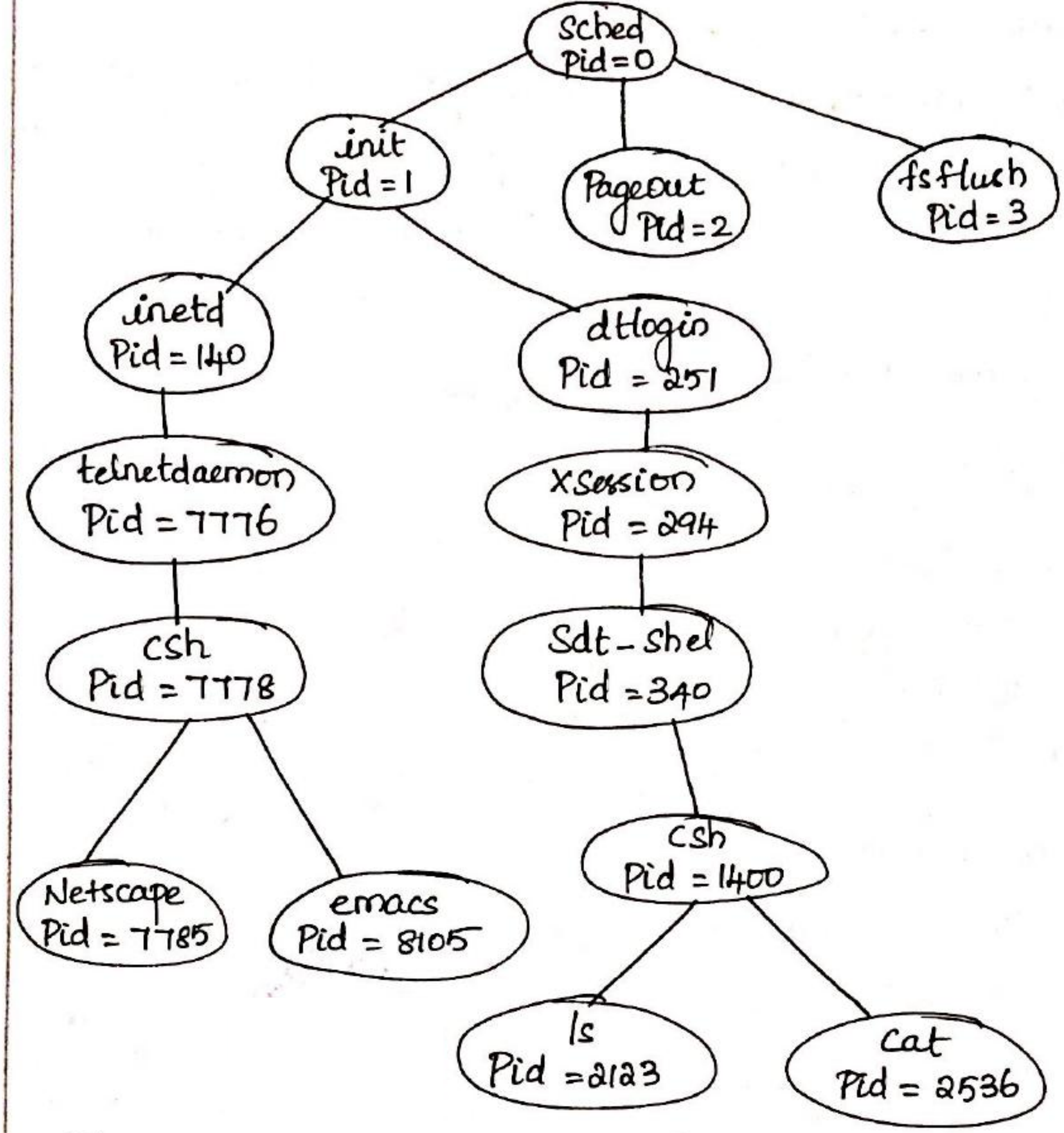
1) The child Process is a duplicate of the parent process (it has the same program and data as the parent).

2) The child Process has a new program loaded into it.

<u>C Program Forking Separate Process.</u>

```
int main ()
{
    Pid_t pid;
   /* fork another Process */
  Pid = fork ();
  if (Pid <0)
{
 /* error occurred */
 fprintf (stderr, "fork failed");
}  exit (-1);
  else if (Pid ==0)
{  ~~error~~ /* Child Process */
execlp ("bin /ls", "ls", NULL);
}
  else
 {  /* Parent process */
 /* Parent will wait for the child to complete */
  wait (NULL);
  Printf ("child Complete");
   exit (0);
 }
}
```

A tree of Processes on a typical solaris system.



In general, a process will need certain resources (cpu time, memory, files, I/o devices), to accomplish its task. When process creates a subprocess, that subprocess may be able to obtain its resources directly from the operating system, or it may be constrained to a subset of the resources of the parent Process.

The Parent may have to partition its resources among its children, or it may be able to share some resources among several of its children.

* Process Termination : Parent - exit ( ) / child - wait ( )

A process terminates when it finishes executing its final statement and asks the operating system to delete it by using the (exit ( ) system call.) At that point, the process may return a status value (typically an integer) to its parent process via the wait ( ) system call.

A process can cause the termination of another process via an appropriate system call. Usually, such a system call can be invoked only by the parent of the process that is to be terminated. Note that a parent needs to know the identities of its children. Thus, when one process creates a new process, the identity of the newly created process is passed to the parent.

✓ A parent may terminate the execution of one of its children for a variety of reasons, such as these;

* The child has exceeded its usage of ~~some of~~ the resources that it has been allocated.

* The task assigned to the child is no longer required.

* The parent is exiting, and the operating system does not allow a child to continue if its parent terminates.

✓ If a process terminates either normally or abnormally then all its children must also be terminated. This phenomenon, referred to as cascading termination, is normally initiated by the operating system.

Eg., To illustrate process execution and termination, consider that, in UNIX, we can terminate a process by using the exit () system call; its parent process may wait for the termination of a child process by using the wait () system call.

The wait () system call returns the process identifier of a terminated child so that the parent can tell which of its children has terminated.

5) Interprocess communication (IPC)

Processes executing concurrently in the operating system may be either independent processes (or) cooperating processes.

A process is independent if it cannot affect (or) be affected by the other processes executing in the system. Any process that does not share data with any other process is independent. A process is cooperating if it can affect or be affected by the other processes executing in the system.

✓ Clearly, any process that shares data with other processes is a cooperating process.

There are several reasons for providing an environment that allows process cooperation:

* Information sharing.
* Computation speedup.
* Modularity.
* Convenience.

Information sharing :

Since several users may be interested in the same piece of information (for instance, a shared file), we must provide an environment to allow concurrent access to such information.

Computation speedup :

If we want a particular task to run faster, we must break it into subtasks, each of which will be executing in parallel with the others. Notice that such a speedup can be achieved only if the computer has multiple processing elements (such as CPUs or I/O channels).

Modularity :

We may want to construct the system in a modular fashion, dividing the system functions into separate processes or threads.

convenience :

Even an individual user may work on many tasks at the same time. for instance, a user may be editing, Printing, and compiling in parallel.

Mechanism for processes to communicate and to Synchronize their actions.

Message system – Process Communicate with each other without resorting to shared variables.

Ipc facility provides two operations :

Send (message) - message size fixed (or) variable.

receive (message).

✓If P and Q wish to communicate, they need to :

→ establish a communication link between them.

→ exchange messages via send / receive.

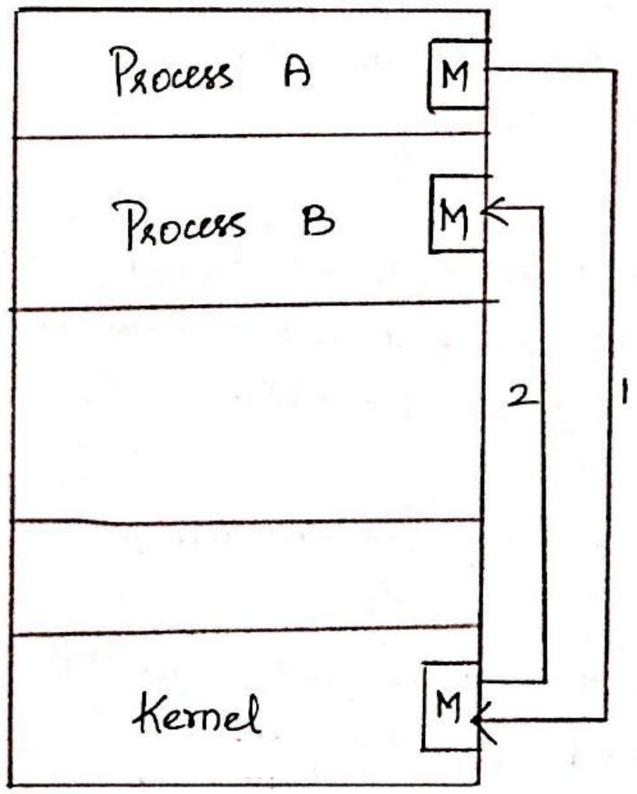
Implementation of communication link

→ Physical (eg., shared memory, hardware bus)

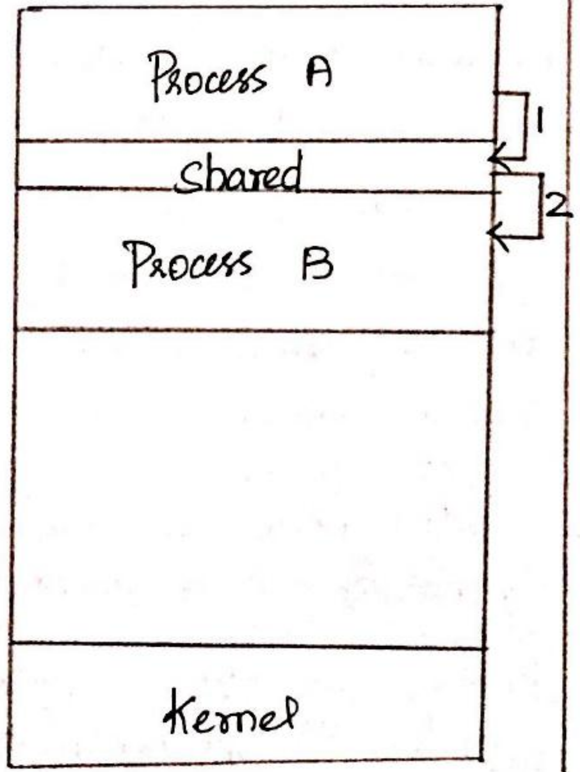
→ Logical (eg., logical properties).

Communication models :-

(a) Message passing



(b) shared Memory



Ipc mechanism will allow them to exchange data and information. These are two fundamentals models of Ipc

* Shared memory
* Message passing.

In shared-memory model, a region of memory that is shard by cooperating process is established.

In message passing model, communication takes place by means of messages exchanged between the cooperating processes.

Ipc is a mechanism by which two (or) more process communicate with each other through message passing mechanism without using shared addressed space.

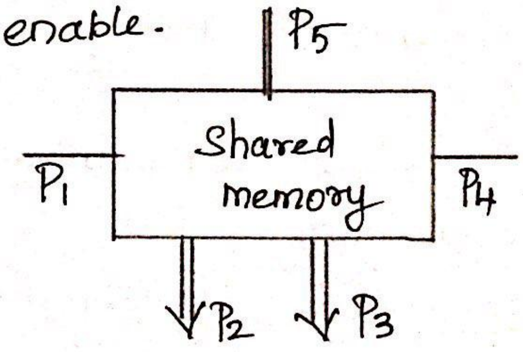
"Ipc means how two process are communicate to each other".

✓* Sharing of data (Memory)

✓* Message passing Mechanism.

Sharing of Data :-

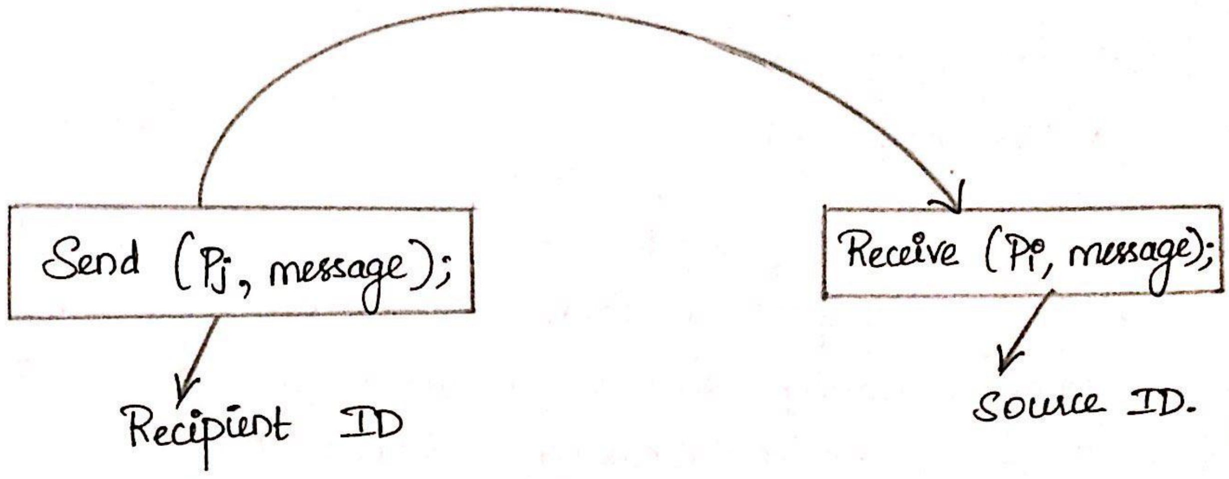
Two ~~are~~ (or) more Processes share their data with each other by which Ipc is enable.



The disadvantage of this mechanism is that your data is shared among all the user.

Message Passing Mechanism :

It send the message to the recipient and recipient receive the message from the source.



Message passing communication :

There are two types of message passing communication are used.

→ Direct Communication.

→ Indirect Communication.

Direct communication :

Sometime is called " Symmetric Naming convention". Here sender and receiver both knows the identity (ID) of each other.

P1 (Sender)
P2 (Receiver)
Send (P2, message)
Receiver (P1, message)

Direct Communication use two types of naming convention.

1) Asymmetric Naming Convention

2) Symmetric Naming convention.

Symmetric Naming Convention :

Both sender & receiver ID's are known.

Eg.,

Send (recipient ID, Message); // sender end

Receiver (source ID, message) ; // Receiver end.

Asymmetric Naming convention :

Sender knows the ID of recipient but receiver only receive the message without knowing sender ID.

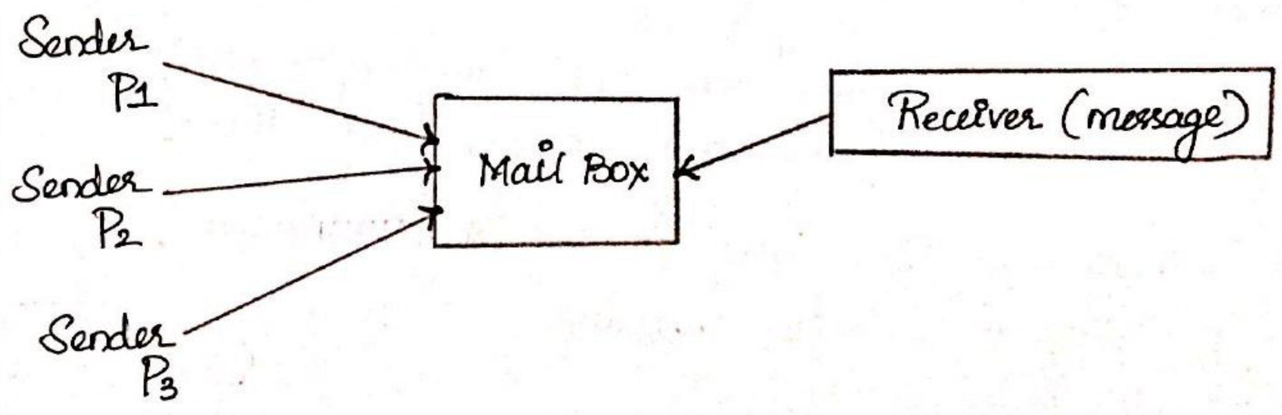
Eg., send (recipient ID, message); // sender end

Receiver (~~[illegible]~~, Message); // receiver end.

Indirect communication :

In this communication we doesn't know the identity of receiver and sender. Receiver receive the message from that mail box. The system calls are as following.

Mailbox own by the receiver.



Synchronous and Asynchronous Message passing.

* It is based on blocking and non blocking method.

* Block refers synchronous message passing and non blocking refers asynchronous message passing.

There are 4 types of message passing.

1) Blocking send : Sender process is block until the receiver receive the message.

2) Non-Blocking Send : Sender process continuously send the message without looking whether receiver receive the message or not.

3) Blocking Receive : Receiver process is block till sender process send the message.

4) Non Blocking receive : Receiver continuously receive the message without any restriction.

* Buffering in IPC :

Buffering is a storage facility which are work in between communication between two (or) more devices.

In IPC messages are not directly transfer to the receiver, it transfer through buffer mechanism.

There are '3' types of buffering techniques;

(1) Zero capacity buffering.

(2) Bounded buffer.

(3) Unbounded buffer.

Buffering Techniques :

(1) Zero capacity : There is no memory used take and deliver the message file that sending process is block.

(2) These type of buffering has buffer size =0.

Bounded Buffer : These type of buffer have finite length.

Unbounded Buffer : These type of buffer have infinite length. thus, any number of messages can wait in it. The Sender never blocks.

The zero-capacity case is sometimes referred to as a message system with no buffering; the other cases are referred to as systems with automatic buffering.

(6) CPU Scheduling :

→ CPU Scheduling is the basis of multiprogrammed Operating systems.

→ The objective of multiprogramming is to have some Process running at all times, in order to maximize CPU Utilization.

→ Scheduling is a fundamental operating-system function.

→ Almost all computer resources are scheduled before use.

CPU scheduling is a process which allows one Process to use the CPU while execution of another Process is on hold (in waiting state) due to Unavailability of any resources like I/O etc., Thereby making full use of CPU, The aim of CPU scheduling is to make the system efficient, fast and fair.

Cpu- I/o Burst cycle.
Cpu Scheduler
Pre emptive scheduling
Non-Preemptive scheduling
Dispatcher.

CPU- I/o Burst cycle:

* Process execution consists of a cycle of cpu execution and I/o wait.

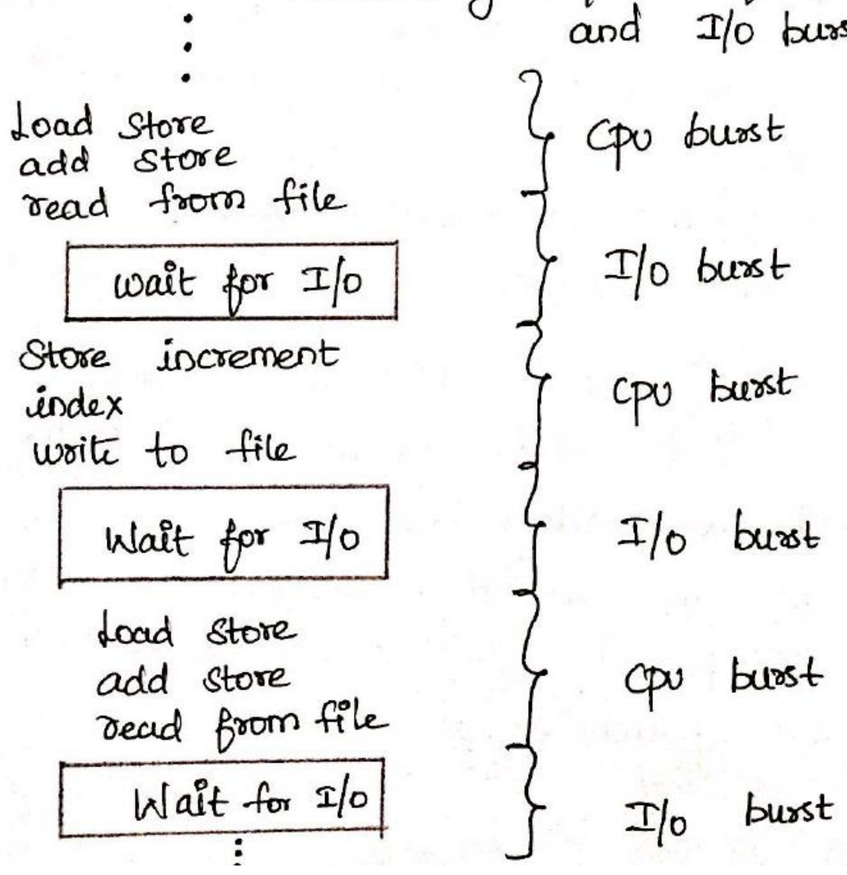
* Processes alternate between these two states.

* Processes execution begins with a cpu burst.

* That is followed by an I/o burst, then another cpu burst, then another I/o burst, and so on.

* Eventually, the last cpu burst will end with a system request to terminate execution, rather than with another I/o burst.

Alternating Sequence of cpu and I/o bursts.



CPU scheduler :

Whenever the CPU becomes idle, the operating system must select one of the processes in the ready queue to be executed.

The selection Process is carried out by the short-term scheduler or CPU scheduler. The ready queue is not necessarily a first-in, first-out (FIFO) Queue. It may be a FIFO Queue, a priority queue, a tree, or simply an Unordered linked list.

Preemptive scheduling :

CPU scheduling decisions may take place under the following four circumstances :

1) When a Process switches from the running state to the waiting state.

2) When a Process switches from the running state to the ready state.

3) When a Process switches from the waiting state to the ready state.

4) When a Process terminates.

Under 1 & 4 scheduling scheme is non-Preemptive. Otherwise the scheduling scheme is Preemptive

Non-Preemptive scheduling :

→ In non Preemptive scheduling, once the CPU has been allocated a Process, the Process keeps the CPU until it releases the CPU either by termination or by switching to the waiting state.

→ This scheduling method is used by the Microsoft windows environment.

Dispatcher :

The dispatcher is the module that gives control of the cpu to the process selected by the short-term Scheduler. The time it takes for the dispatcher to stop one process and start another running is known as dispatch Latency.

This function involves :

1) Switching context
2) switching to user mode
3) Jumping to the proper location in the user Program to restart that Program.

(7) Scheduling criteria :

Many criteria have been suggested for comparing cpu- Scheduling algorithms. The criteria include the following.

✓ 1) cpu Utilization
✓ 2) Throughput
✓ 3) Turnaround time
✓ 4) Waiting time
✓ 5) Response time

cpu Utilization :

We want to keep the cpu as busy as possible. conceptually, cpu Utilization can range from 0 to 100 Percent. In a real system, it should range from 40 percent (for a lighty loaded system) to 90 Percent (for a heavily used system).

Throughput :

If the cpu is busy executing Processes, then work is being done. One measure of work is the number of Processes that are completed per time unit, called throughput. For long Processes, this rate may be

One process per hour; for short transactions, it may be ten processes per second.

Turnaround time :

from the point of view of a particular Process, the important criterion is how long it takes to execute that process. The interval from the time of submission of a process to the time of completion is the turnaround time.

Turnaround time is the sum of the periods spent waiting to get into memory, waiting in the ready queue, executing on the cpu, and doing I/o.

Waiting time :

The cpu scheduling algorithm does not affect the amount of time during which a Process executes or does I/o ; it affects only the amount of time that a Process spends waiting in the ready queue.

Waiting time is the sum of the periods spent waiting in the ready Queue.

Response time :

In an interactive system, turnaround time may not be the best criterion. The time from the submission of a request until the first response is produced.

Response time is the time it takes to start responding, not the time it takes to output the response.

(8) Scheduling algorithms :

cpu scheduling deals with the problem of deciding which of the processes in the ready Queue is to be allocated the cpu. There are many different cpu-scheduling algorithms.

* First-come, First-Served Scheduling (FCFS)
* Shortest - Job - First Scheduling
* Priority scheduling
* Round - Robin scheduling.
* Multilevel Queue scheduling.
* Multilevel feedback queue scheduling.

* First - come, First - Served scheduling :-

The Process that requests the cpu first is allocated the cpu first. FCFS policy is easily managed by FIFO queue.

Consider the following set of processes that arrive at time 0, with the length of the cpu burst given in milliseconds.

Process	Burst time
P_1	24
P_2	3
P_3	3

Arrival time : Time at which the Process arrives in the ready Queue.

completion time : Time at which Process completes its execution.

Burst time : Time required by a Process for cpu execution

Turn around time : Time difference between completion time and arrival time.

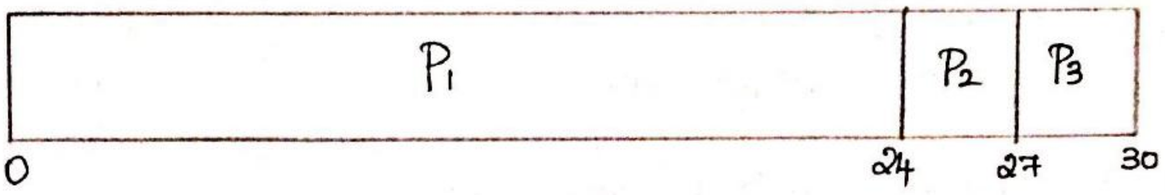
Turn around time = Completion time - Arrival time.

Waiting time : Time difference between turn around time and burst time.

Waiting time = Turn around time - Burst time.

If the processes arrive in the order P_1, P_2, P_3, and are served in FCFS order, we get the result shown in the following Gantt chart.

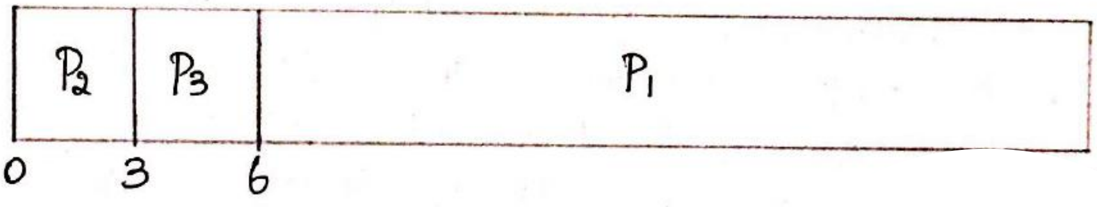
Gantt chart is a bar chart that illustrates a particular schedule, including the start and finish times of each of the participating processes.



The waiting time is 0 milliseconds for process P_1, 24 ms for P_2, 27 ms for P_3.

$\therefore$ Average waiting time $= (0+24+27)/3 = 51/3$
$= 17$ ms.

Eg., If the process arrives in the order P_2, P_3, P_1, then



Waiting time :- $P_2 \rightarrow 0$ ms
$P_3 \rightarrow 3$ ms
$P_1 \rightarrow 6$ ms.

Average waiting time :- $(0+3+6)/3 = 9/3$
$= 3$ milliseconds.

Convoy effect is phenomenon associated with the First come First serve (FCFS) algorithm, in which the whole OS slows down due to few slow processes.

FCFS algorithm is non-preemptive in nature, that is once cpu time has been allocated to a process, other processes can get cpu time only after the current process has finished. This property of FCFS scheduling leads to the situation called convoy effect.

To avoid convoy effect, preemptive scheduling like Round Robin scheduling can be used - as the smaller processes don't have to wait much for cpu time - making their execution faster and leading to less resources sitting idle.

* Shortest - Job - First Scheduling :

A different approach to cpu scheduling is the SJF scheduling algorithm. This algorithm associates with each process the length of the process's next cpu burst.

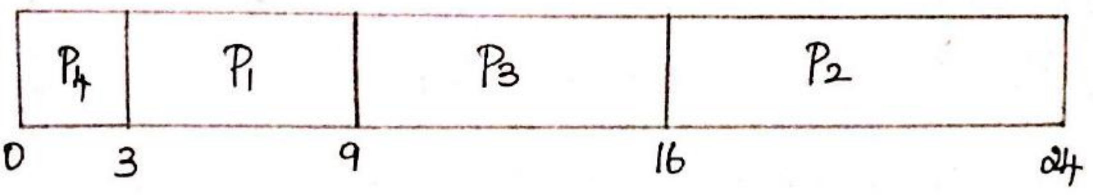
When the cpu is available, it is assigned to the process. that has the smallest next cpu burst. If the next cpu bursts of two processes are the same, FCFS scheduling is used to break the tie.

Note that a more appropriate term for this scheduling method would be the "shortest - next - cpu - burst algorithm", because scheduling depends on the length of the next cpu burst of a process, rather than its total length.

Eg ; with process and cpu burst time in milliseconds.

Process	Burst time
P_1	6
P_2	8
P_3	7
P_4	3

Using SJF Scheduling, the Gantt chart will be as.,



Waiting time :

$P_1 \rightarrow 3$ milliseconds

$P_2 \rightarrow 16$ milliseconds

$P_3 \rightarrow 9$ milliseconds

$P_4 \rightarrow 0$ millisecond.

Average waiting time ;

$\Rightarrow (3+16+9+0)/4 = 28/4 = 7$ milliseconds.

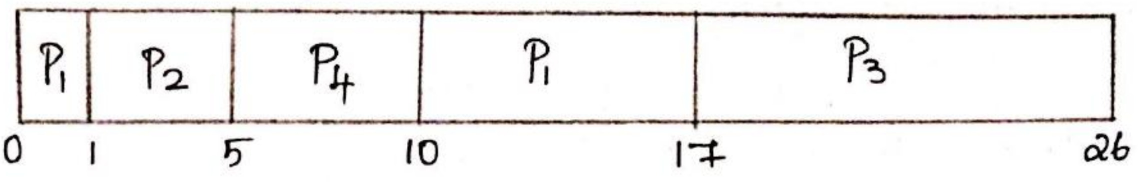
A Preemptive SJF algorithm will preempt the currently executing process, whereas a non-preemptive SJF algorithm will allow the currently running process to finish its cpu burst.

Preemptive SJF scheduling is sometimes called Shortest - remaining - time - first scheduling.

Eg., Consider the following 4 Processes, with the length of the cpu burst given in milliseconds.

Process	Arrival Time	Burst Time
P_1	0	8
P_2	1	4
P_3	2	9
P_4	3	5

If the Processes arrive at the ready Queue at the times shown and need the indicated burst times, then the resulting Preemptive SJF schedule is as depicted in the following Gantt chart.



Process P_1 is started at time 0, since it is the only process in the queue.

Process P_2 arrives at time 1. The remaining time for process P_1 (8-1 = 7 milliseconds) is larger than the time required by process P_2 (4 milliseconds), so Process P_1 is preempted, and P_2 is scheduled. The average waiting time for this example is
$[(10-1) + (1-1) + (17-2) + (5-3)] = 26/4 = 6.5$ ms.

Non-preemptive SJF scheduling would result in an average waiting time of 7.75 milliseconds.

* Priority Scheduling :

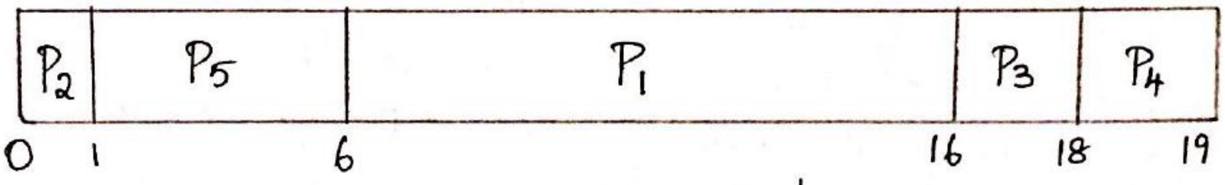
The SJF algorithm is a special case of the general priority scheduling algorithm. A priority is associated with each process, and the cpu is allocated to the process with the highest priority. Equal-priority Processes are scheduled in FCFS order. The larger the cpu burst, the lower the priority, and vice-versa.

Priorities are generally indicated by some fixed range of numbers, such as 0 to 7 (or) 0 to 4095. However, there is no general agreement on whether 0 is the highest (or) lowest Priority. Some system use low numbers to represent low priority and others use low numbers for high priority.

But we assume the low numbers represent highest priority.

Process	Burst time	Priority
P_1	10	3
P_2	1	1
P_3	2	4
P_4	1	5
P_5	5	2

Gantt chart :



Waiting time $= (6+0+16+18+1)/5$

Avg $= 41/5 = 8.2$ milliseconds.

Priority scheduling can be either preemptive (or) non-preemptive. When a process arrives at the ready queue, its priority is compared with the priority of the currently running process.

A preemptive priority scheduling algorithm will preempt the cpu if the priority of the newly arrived process is higher than the priority of the currently running process.

A non preemptive priority scheduling algorithm will simply put the new process at the head of the ready queue.

* Round - Robin Scheduling:

The RR scheduling algorithm is designed especially for time-sharing systems. It is similar to FCFS scheduling, but preemption is added to enable the system to switch between processes. A small unit of time, called a time quantum (or) time slice is defined. A time quantum is generally from 10 to 100 milliseconds in length.

To implement RR scheduling, we keep the ready queue as a FIFO queue of processes. New processes are added to the tail of the ready queue. The cpu scheduler picks the first process from the ready queue, sets a timer to interrupt after 1 time quantum, and dispatches the processes.

Only two things happen. The process may have a cpu burst of less than 1 time quantum. In this case, the process itself will release the cpu voluntarily.

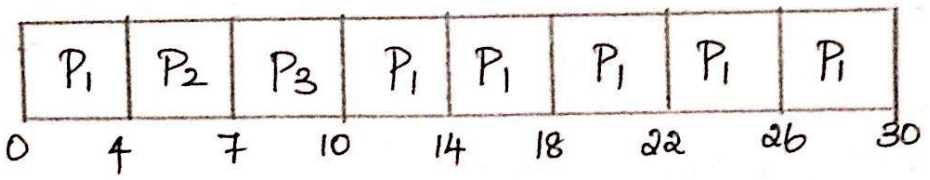
Otherwise, if the cpu burst of the currently running process is longer than 1 time quantum, the timer will go off and will cause an interrupt to the operating system.

Eg., Consider the following set of processes that arrive at time 0, with the length of the cpu burst given in milliseconds:

Process	Burst time
P_1	24
P_2	3
P_3	3

If we use a time Quantum of 4 milliseconds, then Process P1 gets the 1st 4 milliseconds. Since it requires another 20 milliseconds, it is preempted after the first time quantum, and the CPU is given to the next process in the Queue, Process P2. It doesn't need 4 ms, so it quits before its time quantum expires, and so on..

The resulting RR schedule is as follows;



Waiting time :-

$P_1 \rightarrow$ 6 ms (10-4)

$P_2 \rightarrow$ 4 ms

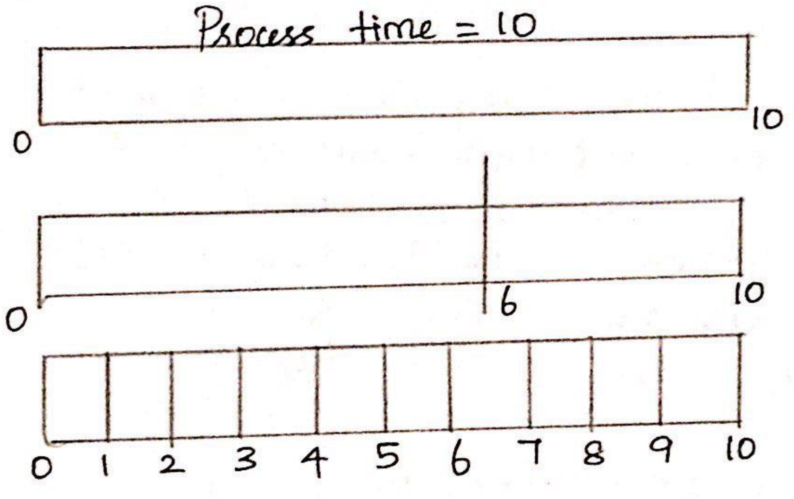
$P_3 \rightarrow$ 7 ms.

Average Waiting time is;

$(6+4+7)/3 = 17/3$

$= 5.66$ ms//.

How a smaller time Quantum increases context switches.



Quantum	Context switches
12	0
6	1
1	9

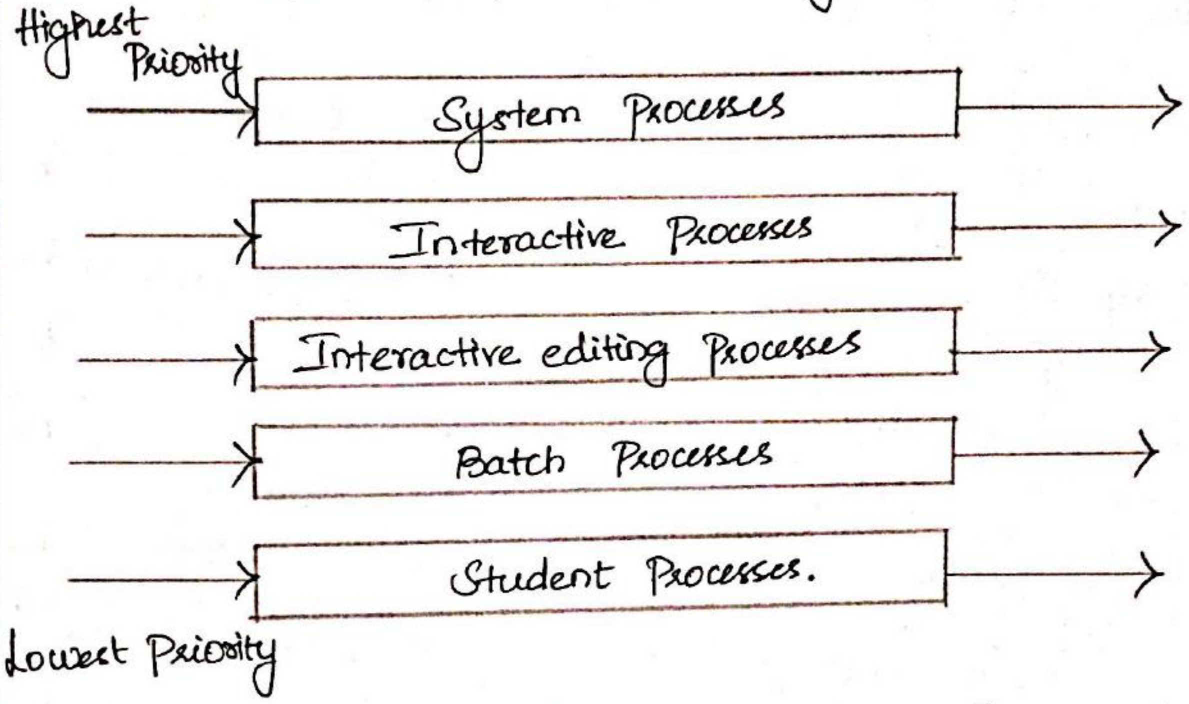
for eg., that we have only one process of 10 time units. If the quantum is 12 time units, the process finishes in less than 1 time quantum, with no overhead.

If the quantum is 6 time units, however, the process requires 2 quanta, resulting in a context switch. If the time quantum is 1 time unit, then 9 context switches will occur, slowing the execution of the process.

* Multilevel Queue scheduling:

It partitions the ready Queue into several separate Queues. The Processes are Permanently assigned to one Queue, generally based on some property of the Process, such as memory size, Process Priority, or Process type. Each Queue has its own scheduling algorithm.

Multilevel Queue scheduling:

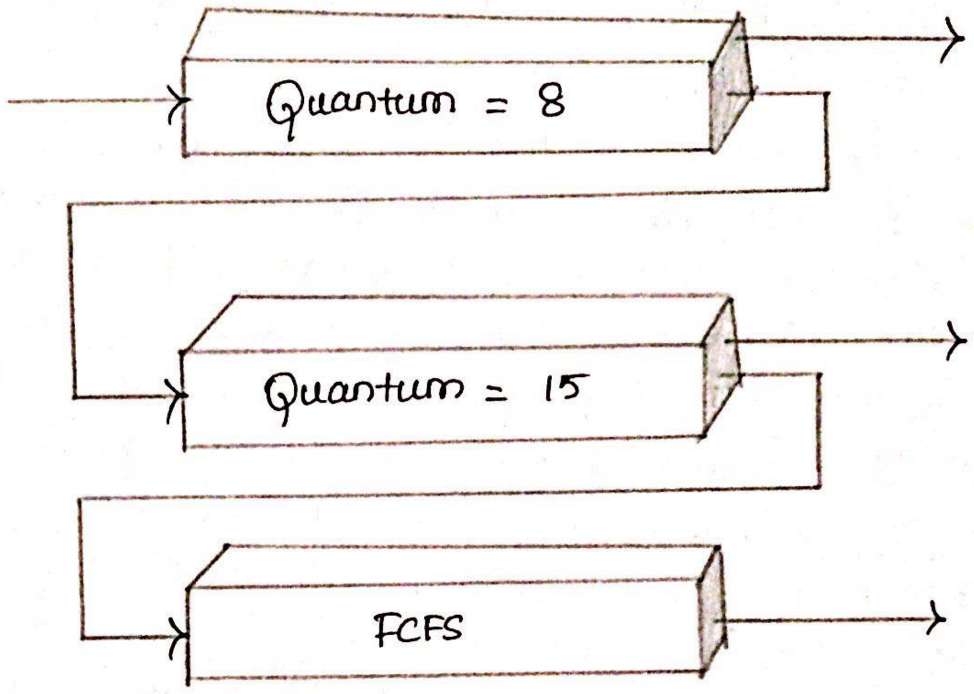


for instance, in the foreground (interactive) Queue – background (batch) Queue example, the foreground Queue can be given 80 percent of the cpu time for RR scheduling among its Processes, whereas the background Queue receives 20 percent of the cpu to give to its Processes on an FCFS basis.

* Multilevel feedback Queue scheduling:

It allows a Process to move between Queues. The idea is to separate Processes according to the characteristics of their cpu bursts. If a Process uses too much cpu time, it will be moved to a lower – Priority Queue. This schemes leaves I/o bound and interactive Processes in the higher – Priority queues.

Multilevel feedback Queues.



In general, a multilevel feedback queue scheduler is defined by the following parameters.

1) The number of queues.

2) The scheduling algorithm for each queue.

3) The method used to determine when to upgrade a Process to a higher Priority Queue.

4) The method used to determine when to demote a Process to a lower Priority Queue.

5) The method used to determine when queue a process will enter when that Process needs service.

Eg., Consider a multilevel feedback Queue scheduler with three Queues, numbered from 0 to 2. The scheduler 1st executes all Process in queue 0. Only when queue 0 is empty will it execute Processes in Queue 1. Similarly, Processes in Queue 2 will only be executed if Queues 0 and 1 are empty. A Process that arrives for Queue 1 will

Preempt a Process in Queue 2. A Process in Queue 1 will Preempt a Process ~~in Queue 2.~~ arriving for Queue 0.

A Process entering the ready queue is put in Queue 0. A Process in Queue 0 is given a time quantum of 8 milliseconds. If it does not finish within this time, it is moved to the tail of Queue 1. If Queue 0 is empty, the Process at the head of queue 1 is given a Quantum of 16 milliseconds. If it does not complete, it is Preempted and is put into Queue 2. Processes in Queue 2 are run on as FCFS basis, but are run only when queues 0 and 1 are empty.

(9) Multiple - Processor scheduling :

In multiple - Processor scheduling multiple CPU's are available and hence Load sharing becomes Possible. However multiple Processor scheduling is more complex as compared to single processor scheduling. In multiple processor scheduling there are cases when the Processors are identical i.e., Homogeneous, in terms of their functionality, we can use any processor available to run any Process in the Queue.

Approaches to Multiple - Processor scheduling.

- Processor affinity.
- Load Balancing.
- Multicore Processors.
- Virtualization and scheduling.

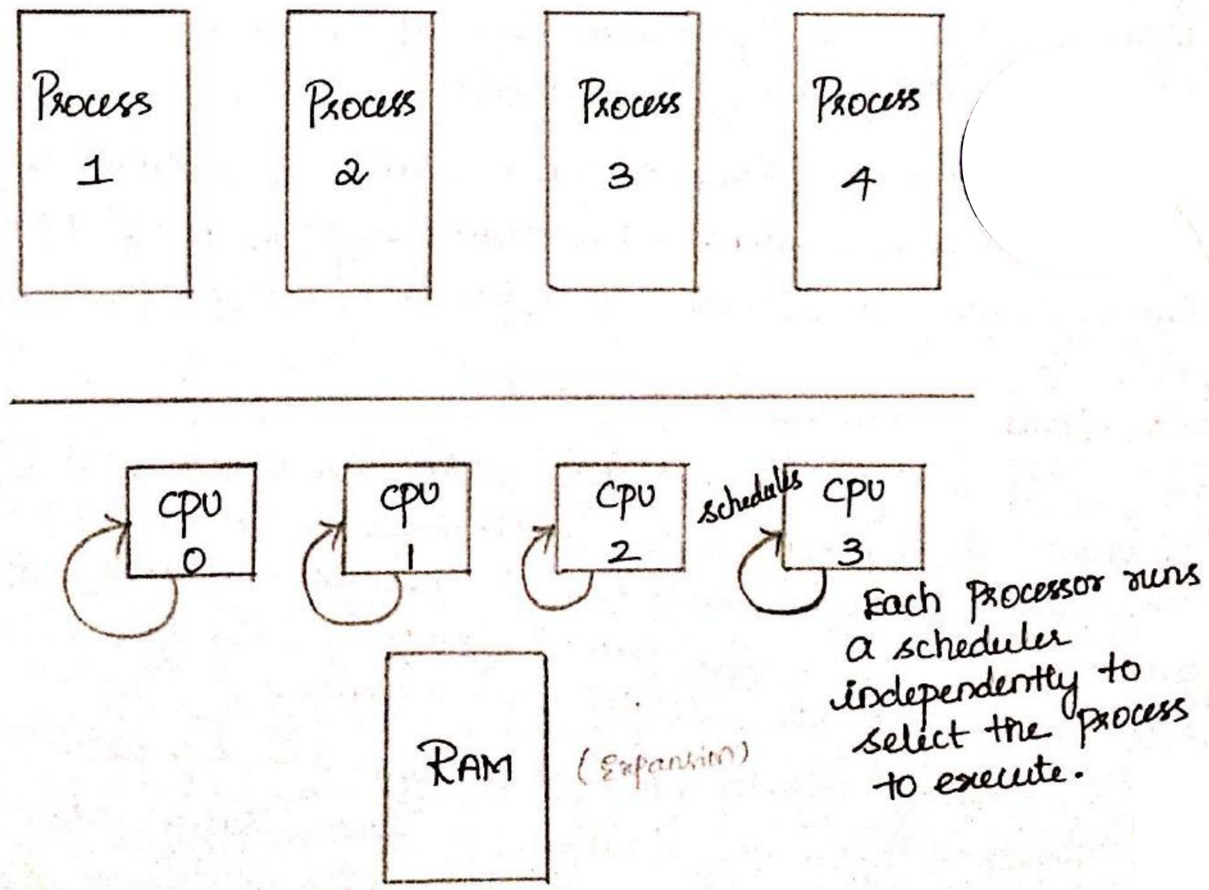
* Approaches to Multiple - Processor scheduling :-

One approach is when all the scheduling decisions and I/O processing are handled by a single processor which is called the Master Server and the other processors executes only the user code. This entire scenario is called Asymmetric multiprocessing.

A Second approach uses Symmetric multiprocessing where each processor is self scheduling. All processes may be in a common ready Queue (or) each processor may have its own private Queue for ready processes.

The scheduling proceeds further by having the scheduler for each processor examine the ready Queue and select a process to execute.

Multiprocessor scheduling.


Process 1
Process 2
Process 3
Process 4
CPU 0
CPU 1
CPU 2
schedules
CPU 3
RAM
(Expansion)
Each processor runs a scheduler independently to select the process to execute.

* Processor Affinity : It means a Processes has an affinity for the Processor on which it is currently running.

When a Process runs on a specific Processor there are certain effects on the ~~ch~~· cache memory. If the Process migrates to another Processor, the contents of the cache memory must be invalidated for the first Processor and the cache for the second Processor must be repopulated. Because of high cost of invalidating and repopulating caches, most of the SMP (symmetric multiprocessing) systems try to avoid migration of Processes from one processor to another and try to keep a Process running on the same Processor. This is known as Processor affinity.

— Soft Affinity.

\ Hard Affinity.

Soft Affinity : When an operating system has a policy of attempting to keep a Process running on the same Processor but not guaranteeing it will do so, this situation is called soft affinity.

Hard Affinity : Some systems such as Linux also Provide some system calls that support Hard Affinity which allows a Process to migrate between Processors.

* Load Balancing :

It is the phenomena which keeps the workload evenly distributed across all processors in an SMP systems. Load Balancing is necessary only on systems where each processor has its own Private Queue of Process which are eligible to execute.

Load Balancing is unnecessary because once a Processor becomes idle it immediately extracts a runnable Process from the Common run Queue.

On SMP it is important to keep the workload balanced among all Processors to fully utilize the benefits of having more than one processor else one (or) more than one Processor ~~else one~~ will sit idle while other Processors have high workloads along with lists of Processors awaiting the cpu.

There are two general approaches to load balancing ;

* Push migration
* pull migration.

In Push migration a task routinely checks the load on each processor and if it finds an imbalance then it evenly distributes load on each processors by moving the Processes from overloaded to idle or less busy Processors.

In pull migration when an idle Processor pulls a waiting task from a busy Processor for its execution.

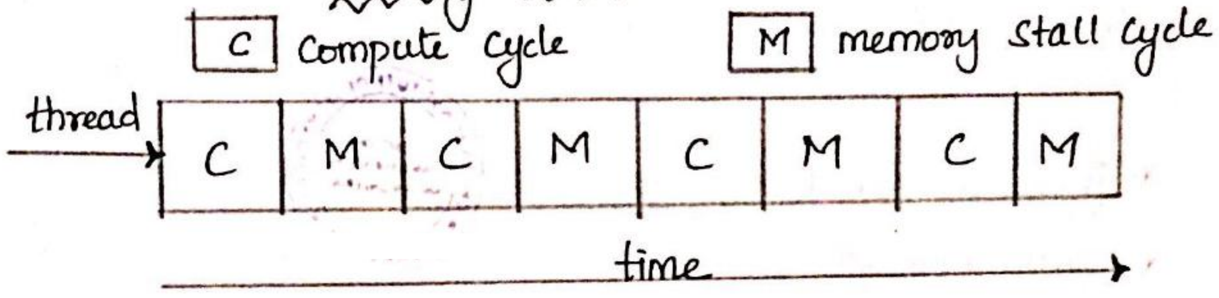
* Multi-core Processors :-

In multi-core processors multiple processor cores are places on the same physical chip. Each core has a register set to maintain its architectural state and thus appears to the operating system as a separate physical processor.

SMP systems that use multicore Processors are faster and consume less power than systems in which each processor has its own physical chip.

However multicore Processors may complicate the scheduling problems. When processor accesses memory then it spends a significant amount of time waiting for the data to become available. This situation is called Memory stall.

Memory stall.



It occurs for various reasons such as cache miss, which is accessing the data that is not in the cache memory. In such cases the processor can spend upto fify percent of its time waiting for data to become available from the memory.

To solve this problem recent hardware designs have implemented multithreaded processor cores in which two or more hardware threads are assigned to each core. Therefore if one thread stalls while waiting for the memory, core can switch to another thread.

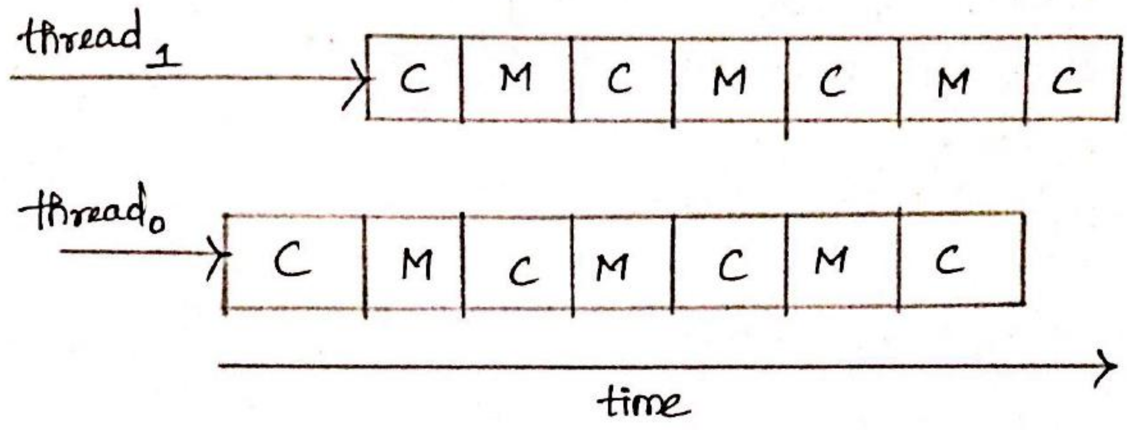
'2' Ways to multi-thread a processor.

→ Coarse - Grained Multithreading.
→ Fine - Grained Multithreading.

Coarse - Grained Multithreading :- A thread executes on a processor until a long latency event such as a memory stall occurs.

Fine - Grained Multithreading :- It switches between threads at a much finer level mainly at the boundary of an instruction cycle.

Multithreaded multicore system.



Virtualization and ~~Threading~~ Scheduling :-

In this type of multiple - Processor scheduling even a single cpu system acts like a multiple-Processor system. Most virtualized environments have one host or operating system and many guest operating systems.

The Host OS creates and manages the virtual machines and each virtual machine has a guest OS installed and applications running within that guest.

Each guest OS may be assigned for specific use cases, applications, and users, including time sharing or even real-time operation.

Virtualization can thus undo the good scheduling - algorithm efforts of the operating systems within virtual machines.

(10) Real-time scheduling :-

Tasks or processes attempt to control or react to events that take place in the outside world. These event occur in real time and process must be able to keep up with them.

Real time systems examples are

a) control of laboratory experiments.

b) Process control in industrial plants.

c) Robotics.

d) Air traffic control.

e) Telecommunications.

f) Military command and control systems.

Classification of real time task :

1) Hard real time
2) Soft real time
3) Aperiodic task
4) Periodic task

A hard real time task must meet its deadline, Otherwise it will cause undesirable damage or a fatal error to the system.

A Soft real time : Deadline is desirable but not mandatory, work is continued even if deadline missed.

Aperiodic task has deadline or constraint for Start (Or) finish times (or) both.

Periodic task : Requirement may be stated as Once per period T (Or) exactly T units apart.

Real time OS characteristics :

(i) Deterministic
(ii) Responsiveness
(iii) User Control
(iv) Reliability
(v) Fail soft operation.

(i) Deterministic :

* Operations are performed at fixed, predetermined times or within predetermined time intervals.

* Concerned with how long the operating system delays before acknowledging an interrupt and there is sufficient capacity to handle all the requests within the required time.

(ii) Responsiveness :

* How long, after acknowledgement, it takes the operating system to service interrupt.

* Includes amount of time to begin execution of the interrupt.

* Includes the amount of time to perform the interrupt.

(iii) User Control :

* It is essential to allow the user fine grained control over task priority. The user should be able to distinguish between hard and softs tasks and to specify relative priorities within each class.

* It also allow the user to specify the use of paging or process swapping.

(iv) Reliability :

* A Real time system is responding to and controlling events in real time. So loss or degradation of performance may have catastrophic consequences, ranging from financial loss to major equipment damage and even loss of life.

* A processor failure in a multiprocessor nonreal time system may result in a reduced level of service until the failed processor is repaired (or) replaced.

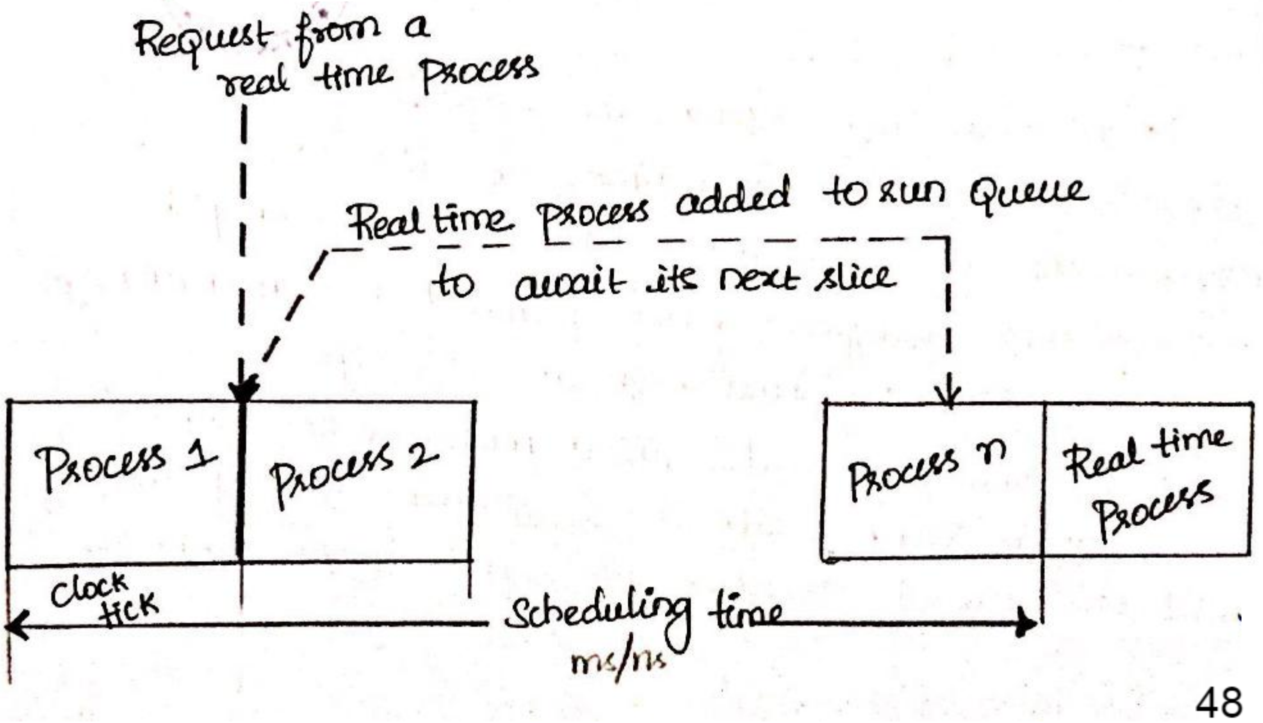
Features of Real time OS:-

1) Fast context switch
2) small size.
3) Ability to respond to external interrupts quickly.
4) Multitasking with Ipc tools such as semaphores, signals and events.
5) Preemptive scheduling based on priority.
6) Special alarms and timeouts.
7) Minimization of intervals during which interrupts are disabled.
8) Use of special sequential files that can accumulate data at a fast rate.

Classes of Real time scheduling:-

In a preemptive scheduler that uses simple round robin scheduling; a real time task would be added to the ready queue to awaits its next time slice.

RR preemptive scheduler.


Request from a real time process
Real time process added to run Queue to await its next slice
Process 1
Process 2
Process n
Real time Process
Clock tick
Scheduling time
ms/ns

Classes of Algorithms :-

1) Static table driven approaches.
2) Static priority driven Preemptive approaches.
3) Dynamic planning based approaches.
4) Dynamic best effort approaches.

⇒ Static table driven approach is applicable to tasks that are periodic. These perform a static analysis of feasible schedules of dispatching. The result of the analysis is scheduled that determines, at run time, when a task must begin execution.

⇒ In static priority driven preemptive scheduling, Static analysis is performed. But no schedule is drawn up. The analysis is used to assign priority to tasks, so that a traditional Priority driven Preemptive scheduler can be used. One example of this approach is the rate monotonic algorithm, which assigns static priorities to tasks based on their periods.

⇒ In dynamic Planning based scheduling, feasibility is determined at run time rather than off line prior to the start of execution.

Dynamic best effort approaches : No feasibility analysis is performed. The system tries to meet all deadlines and aborts any started process whose deadline is missed.

⑪ Threads : [N/D-19]

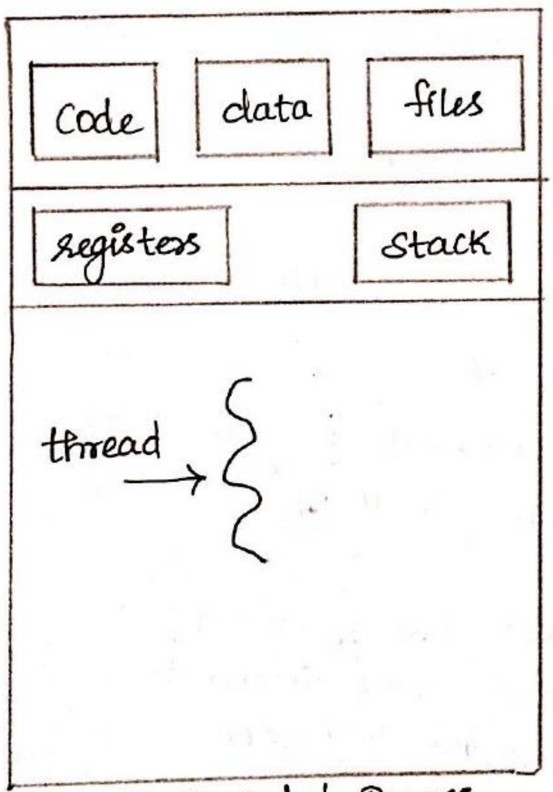
Overview.

* Motivation
* Benefits
* Multicore Programming

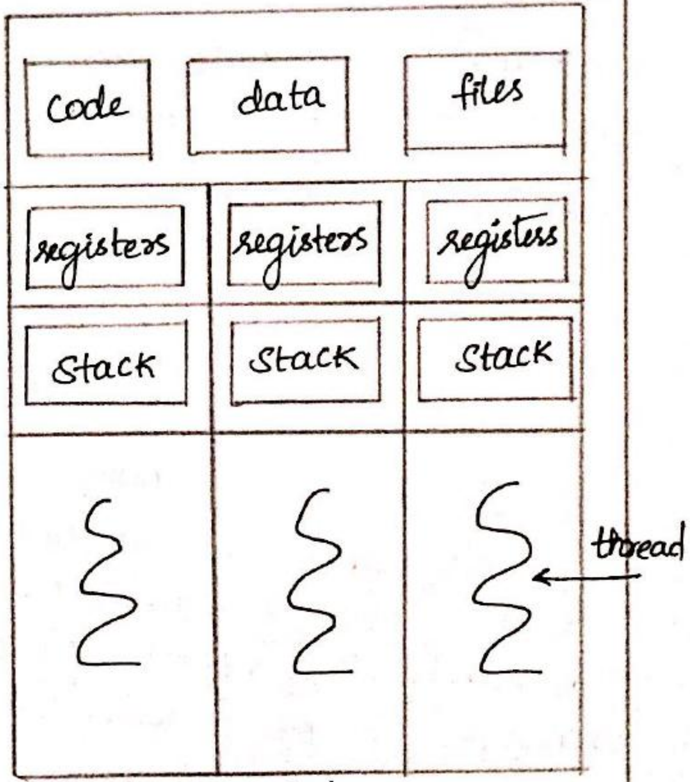
A thread is a basic unit of CPU Utilization; it Comprises a thread ID, a Program counter, a register set, and a Stack. It shares with other threads belonging to the same Process its code section, data section, and other Operating system resources, such as open files and signals.

A traditional (or) Heavy weight Process has a single thread of control. If a process has multiple threads of control, it can perform more than one task at a time.

Single threaded and multithreaded Process.



Single-threaded Process

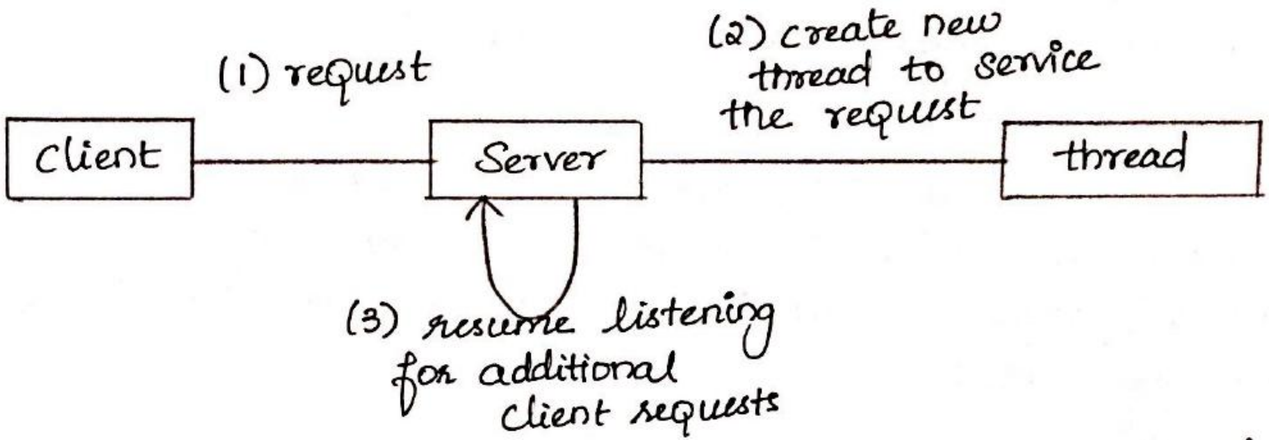


Multithreaded Process

* Motivation :

Many Software packages that run on modern desktop PCs are multithreaded. Threads also play vital role in remote Procedure call (RPC), systems. finally, most operating system Kernels are now multithreaded; several threads operate in the kernel, and each thread Performs a specific task, such as managing devices (or) interrupt handling.

Multithreaded Server architecture.


(1) request
(2) create new thread to service the request
client
Server
thread
(3) resume listening for additional client requests

Eg., Solaris creates a set of threads in the kernel Specifically for interrupt handling. Linux uses a kernel thread for managing the amount of free memory in the system.

* Benefits :-

The benefits of multithreaded programming can be broken down into four major categories.

(1) Responsiveness
(2) Resource sharing
(3) Economy
(4) Scalability.

(1) Responsiveness :

Multithreading an interactive application may allow a program to continue running even if part of it is blocked (or) is Performing a lengthy operation, thereby increasing responsiveness to the user.

(2) Resource sharing :

Processes can only share resources through techniques such as shared memory and message passing.

(3) Economy :

Allocating memory and resources for process creation is costly. Because threads share the resources of the Process to which they belong, it is more economical to create and context-switch threads.

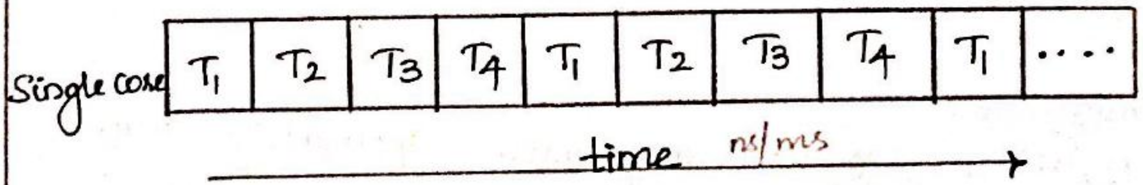
(4) Scalability :

The benefits of multithreading can be even greater in a multiprocessor architecture, where threads may be running in parallel on different Processing cores.

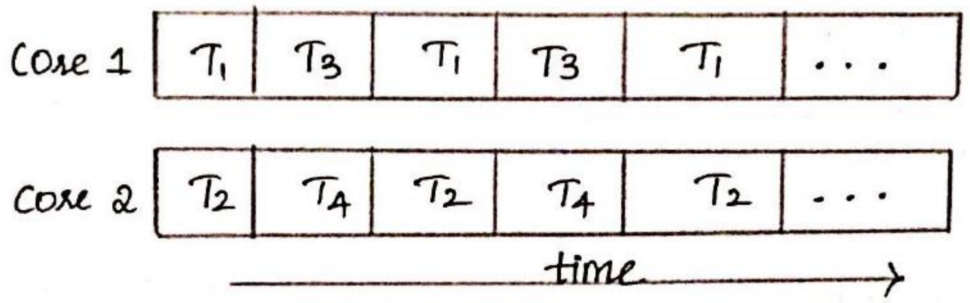
* Multicore Programming :

Earlier in the history of Computer design, in response to the need for more computing Performance, single-cpu systems evolved into multi-cpu systems. A more recent, similar trend in system design is to Place multiple computing cores on a single chip. Each core appears as a separate processor to the operating whether the cores appear across cpu chips or within cpu chips, we call these systems as multicore (or) multiprocessor systems.

Concurrent execution on a single-core system.



Parallel execution on a multicore system.



Multithreaded Programming provides a mechanism for more efficient use of these multiple computing cores and improved concurrency. Consider an application with four threads. On a system with a single computing core, concurrency merely means that the execution of the threads will be interleaved over time because the processing core is capable of executing only one thread at a time. On a system with multiple cores, however, concurrency means that the threads can run in parallel, because the system can assign a separate thread to each core. Notice the distinction between parallelism and concurrency in this discussion.

A system is parallel if it can perform more than one task simultaneously. In contrast, a concurrent system supports more than one task by allowing all the tasks to make progress.

(13) Multithreading models:

A relationship must exist between user threads and kernel threads. In this section, we look at three common ways of establishing such a relationship.

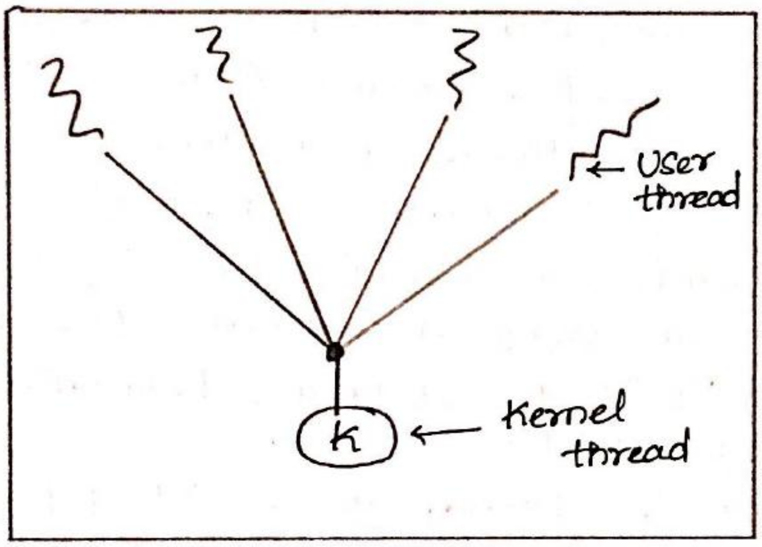
* Many - to - one
* one - to - one
* many - to - many.

Many - to - one :

→ Many User - level threads mapped to single kernel thread.

→ Examples :

→ Solaris Green Threads.
→ GNU portable Threads.



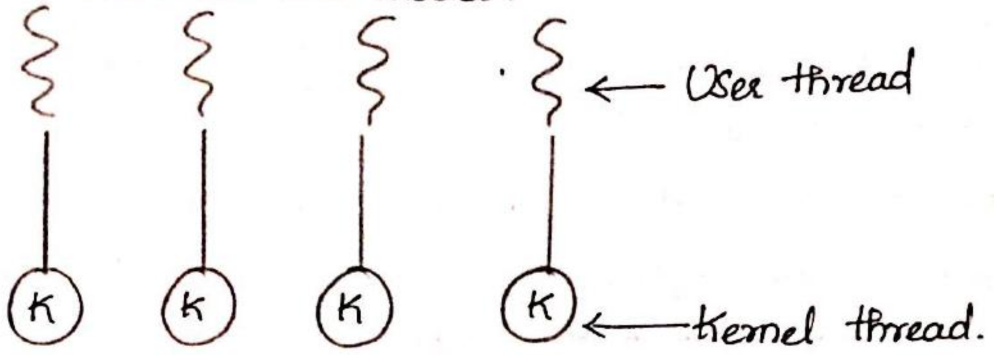
Thread management is done by the thread library in user space, so it is efficient; but the entire Process will block if a thread makes a blocking system call. Also, because (only one thread can access the kernel at a time,) multiple threads are unable to run in parallel on multiprocessors.

* one - one model :

Each user - level thread maps to kernel thread.

Eg ; Windows NT/XP/2000
Linux
Solaris 9 and later

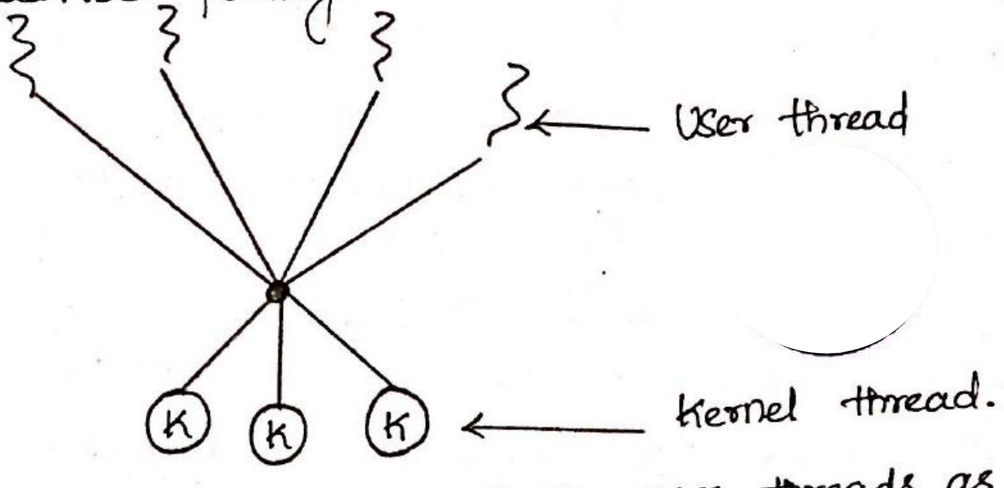
One-to-one model :



(It allows multiple threads to run in Parallel on multiprocessors.) The only drawback to this model is that creating a user thread requires creating the corresponding kernel thread.

* Many-to-many model :

(Allows many user level threads to be mapped to many kernel threads.) Allows the operating system to create a sufficient number of kernel threads. Solaris prior to version 9. Windows NT/2000 with the ThreadFiber package.



Developers can create as many user threads as necessary, and the corresponding kernel threads can run in parallel on a multiprocessor. Also called as two-level model.

14) Threading issues :

There are a Varity of issues to consider with multithreaded Programming.

- Semantics of fork() and exec() system calls.
- Thread cancellation.
  * Asynchronous (or) deferred.
- Signal handling.
  * Synchronous and asynchronous
- Thread pooling.
- Thread - specific data.
  * Create facility needed for data private to thread.

- Semantics of fork() and exec() :-

• Recall that when fork() is called, a separate, duplicate process is created.

• How should fork() behave in a multithreaded Program?

- Should all threads be duplicated?
- should only the threads that made the call to fork() be duplicated?

• In some systems, different versions of fork() exist depending on the desired behavior.

- Some UNIX systems have fork1() and forkall()

✓ fork1() only duplicates the calling thread.

✓ forkall() duplicates all of the threads in a Process.

- In a posix - compliant system, fork() behaves the same as fork1().

Semantics of fork() and exec()

- The exec() system call continues to behave as expected.

Replaces the entire process that called it, including all threads.

If planning to call exec() after fork(), then there is no need to duplicate all of the threads in the calling process.

✓* All threads in the child process will be terminated when exec() is called.

* Use fork1(), rather than forkall() if using in conjunction with exec()

* Thread Cancellation :

(It is the act of terminating a thread before it has completed.)

Eg., clicking the stop button on your web browser will stop the thread that is rendering the web page.

The thread to be cancelled is called the target thread. Threads can be cancelled in a couple of ways

✓* Asynchronous cancellation :

Terminates the target thread immediately. Thread may be in the middle of writing data... not so good.

✓* Deferred Cancellation : It allows the target thread to periodically check if it should be cancelled. Allows thread to terminate itself in an orderly fashion.

* Signal Handling :

Signals are used in UNIX systems to notify a process that a particular event has occurred.

CTRL-C is an example of an asynchronous signal that might be sent to a process.

* An asynchronous signal is one that is generated from outside the process that receives it.

Divide by 0 is an example of a synchronous signal that might be sent to a process.

* A synchronous signal is delivered to the same process that caused the signal to occur.

All signals follows the same basic pattern :

* A signal is generated by particular event.
* The signal is delivered to a process.
* The signal is handled by a signal handler (all signals are handled exactly once).

Signal handling is straightforward in a single-threaded process.

* The one (and only) thread in the process receives and handles the signal.

In a multithreaded program, where should signals be delivered?

Options :

(1) Deliver the signal to the thread to which the signal applies.

(2) Deliver the signal to every thread in the process.

(3) Deliver the signal only to certain threads in the process.

(4) Assign a specific thread to receive all signals for the process.

Option 1: Deliver the signal to the thread to which the signal applies.

Most likely option when handling synchronous signals (eg., only the thread that attempts to divide by zero needs to know of the error).

Option 2: Deliver the signal to every thread in the process.

Likely to be used in the event that the Process is being terminated. (eg., a CTRL-C is sent to terminate the Process, all threads need to receive this signal and terminate).

* Thread pools:

⇒ In applications where threads are repeatedly being created / destroyed "thread pools" might provide a performance benefit.

Eg., A server that spawns a new thread each time a client connects to the system and discards that thread when the client disconnects.

⇒ A thread pool is a group of threads that have been pre-created and are available to do work as needed.

✓ - Threads may be created when process starts.

✓ - A thread may be kept in a Queue until it is needed.

✓ - After a thread finishes, it is placed back into a Queue until it is needed again.

- Avoids the extra time needed to spawn new threads when they are needed.

Advantages of thread pools:

– Typically faster to service a request with an existing thread than create a new thread (Performance benefit).

– Bounds the number of threads in a process.

* Thread-specific data: In some applications it may be useful for each thread to have its own copy of data.

→ Also referred as Thread-local storage (or) Thread-static variables.

In C#

```
Class FooBar {
        [Threadstatic] static int foo;
}
```

Eg., Windows XP.

Implements threads using the one-to-one thread model.

Also implements a fiber that uses a many-to-many model.

Linux oftentimes uses the term task rather than Process or thread.

(15) Process Synchronization:

1) Concurrent access to shared data may result in data inconsistency.

2) Maintaining data consistency requires mechanisms to ensure the orderly execution of cooperating processes.

3) Shared memory solution to bounded-buffer problem allows atmost n-1 items in buffer at the same time. A solution, where all N buffers are used is not simple.

4) Suppose that we modify the Producer - consumer code by adding a variable counter, initialized to 0 and increment it each time a new item is added to the buffer.

5) Race condition : The situation where several Processes access - and manipulate shared data Concurrently. The final value of the shared data depends upon which process finishes last.

6) To Prevent race conditions, concurrent Processes must be synchronized.

(16) critical Section problem :

Consider a system consisting of n Processes {$P_0, P_1, \dots P_{n-1}$}. Each process has a segment of code, called a critical section, in which the Process may be changing common variables, updating a table, writing a file and so on.

The important feature of the system is that, when one process is executing in its critical section, no other process is to be allowed to execute in its critical section. ie., no two Processes are executing in their critical sections at the same time.

Each process must request Permission to enter into critical section. The section of code implementing this request is the entry section, and followed by an exit section. The remaining code is the remainder section.

The general structure of a typical process P_1 is shown.

```
{ do
  | entry section |
      critical section
  | exit section |
      remainder section
} while (TRUE);
```

A Solution to the critical section problem must satisfy the following three requirements.

* Mutual Exclusion
* Progress
* Bounded waiting.

Mutual Exclusion : If process P_1 is executing in its critical Section, then no other processes can be executing in their critical sections.

Progress : If no process is executing in its critical section and some processes wish to enter their critical sections, then only those processes that are not executing in their remainder sections can Participate in deciding which will enter its critical section next, and this selection cannot be postponed indefinitely.

Bounded Waiting : There exists a bound, or limit, on the number of times that other processes are allowed to enter their critical sections after a Process has made a request to enter its critical section and before that request is granted.)

Two general approaches are used to handle critical Sections in operating systems:

* Preemptive kernels
* non-preemptive kernels.

A Preemptive Kernel allows a process to be Preempted while it is running in kernel mode.

A non-preemptive kernel does not allow a process running in kernel mode to be preempted; a kernel-mode process will run until it exits kernel mode, blocks, or voluntarily yields control of the cpu.

17) Synchronization Hardware:

Many systems Provide hardware support for critical sections. Many systems provide special hardware instructions that allows us either to test and modify the content of a word set. If two test and set instructions are executed simultaneously, they will be executed sequentially in some arbitrary order. The Process enter its critical section only if waiting [i] == false and key == false.

Eg;

```
While (true)
{
   waiting [i] = true;
   key = true;
   while (waiting [i] && key)
   key = testandset (lock);
   waiting [i] = false;
   critical section
   j = j+1;
   //waiting [j] = false;
}
```

Uniprocessors :

- could disable interrupts.
- Currently running code would execute without Preemption.
- generally too inefficient on multiprocessor system.

* Modern machine provide special atomic hardware instructions.

* Atomic - Non interruptable.

- Either test memory word and set value.
- (or) swap contents of two memory words.

✓
```
do {
    acquire lock
        critical section
    release lock
        remainder section
} while (TRUE);
```

(18) Mutex Locks :

In computer Programming, a muteral exclusion object (mutex) is a Program object that allows multiple program threads to share the same resources, such as file access, but not simultaneously.

Strictly speaking, a mutex is locking mechanism used to synchronize access to a resource. Only one task (can be a thread or process based on OS abstraction) can acquire the mutex. It means there is ownership associated with mutex, and only the owner can release the lock (mutex).

The mutex is set to unlock when the data is no longer needed or the routine is finished.

(19) Semaphores :

The hardware - based solutions to the critical - section problem Presented are complicated for application programmers to use. To Overcome this difficulty, we can use a synchronization tool called a Semaphore.

A Semaphore S is an integer Variable that, apart from initialization, is accessed only through two standard atomic operations :

✓ wait () and Signal ().

The wait () operation was Originally termed "P"
The ~~ss~~ Signal () operation was Originally termed "V".

The definition of wait () is as follows ;

```
Wait (s) {
        while S <= 0
          ; // no-op
        S--;
    }
```

The definition of signal () is as follows :

```
Signal (s) {
    S++;
}
```

When one process modifies the semaphore value, no other process can simultaneously modify that same semaphore value.

* Usage
* Implementation
* Deadlocks and Starvation
* Priority Inversion

* Usage: operating systems often distinguish between counting and binary semaphores. The value of a counting semaphores can range over an unrestricted domain. The value of a binary semaphores can range only between 0 and 1. On some systems, binary semaphores are known as mutex locks, as they are locks that provide mutual exclusion.

Semaphores also used to solve various synchronization. Problems, for example, consider two concurrently running processes: P_1 with a statement S_1 and P_2 with a statement S_2. Suppose we require that S_2 be executed only after S_1 has completed.

We can implement this scheme readily by letting P_1 and P_2 share a common semaphore synch, initialized to 0, and by inserting the statements.

```
S1 ;
signal (synch);
```

in process P_1 and the statements

```
wait (synch);
S2 ;
```

in process P_2. Because synch is initialized to 0, P_2 will execute S_2 only after P_1 has invoked signal (synch), which is after statement S_1 has been executed.

* Implementation :

The main disadvantage of the semaphore definition given here is that it requires busy waiting. While a process is in its critical section, any other process that tries to enter its critical section must loop continuously in the entry code. This conditional looping is clearly a problem in a real multiprogramming system, where a single cpu is shared by many processes.

Mutual-exclusion implementation with semaphores.

```
do {
    wait (mutex);
      // critical section
    signal (mutex);
      // remainder section
   } while (TRUE);
```

The semaphore discussed so far requires a busy waiting. That is if a process is in critical-section, the other process that tries to enter its critical-section must loop continuously in the entry code.

To overcome the busy waiting problem, the definition of the semaphore operations wait and signal should be modified.

⇒ When a process executes the wait operation and finds that the semaphore value is not positive, the process can block itself. The block

Operation places the Process into a waiting Queue associated with the semaphore.

⇒ A Process that is blocked waiting on a semaphore should be restarted when some other Process executes a signal operation. The blocked Process should be restarted by a wakeup operation which put that process into ready Queue.

To implement the semaphore, we define a semaphore as a record as;

```
typedef struct {
    int value;
    struct process *L;
  } Semaphore;
```

* Deadlock and Starvation:

The implementation of a semaphore with a waiting Queue may result in a situation where two (or) more processes are waiting indefinitely for an event that can be caused only by one of the waiting Processes.

The event in question is the execution of a signal () operation. When such a state is reached, these Processes are said to be deadlocked.

To illustrate this, we consider a system consisting of two processes, P_0 and P_1, each accessing two semaphores, S and Q, set to the value 1;

```
        P0                  P1
     wait (s);          wait (Q);
     wait (Q);          wait (s);
        .                  :
        .
     signal (s);        signal (Q);
     signal (Q);        signal (s);
```

Suppose that P_0 executes wait (S) and then P_1 executes wait (Q). When P_0 executes wait (Q), it must wait until P_1 executes signal (Q). Similarly, when P_1 executes wait (S), it must wait until P_0 executes signal (S). Since these signal () operations cannot be executed, P_0 and P_1 are deadlocked.

Another problem related to deadlock is indefinite blocking or starvation, a situation where a process wait indefinitely within the semaphore. Indefinite blocking may occur if we add or remove processes from the list associated with a semaphore in LIFO order.

Types of Semaphores.

* Counting Semaphore - any positive integer Value.
* Binary Semaphore - integer value can range only between 0 and 1.

* Priority Inversion :

A scheduling challenge arises when a higher-priority process needs to read (or) modify kernel data that are currently being accessed by a lower-priority process - or a chain of lower-priority processes. This problem is known as priority Inversion.

It occurs only in systems with more than two priorities, so one solution is to have only two priorities.

(20) Classic problems of Synchronization : [N|D-19]

Here, we Present a number of synchronization Problems as examples of a large class of concurrency control problems. These Problems are used for testing nearly every newly proposed synchronization scheme. In our solutions to the problems, we use semaphores for synchronization.

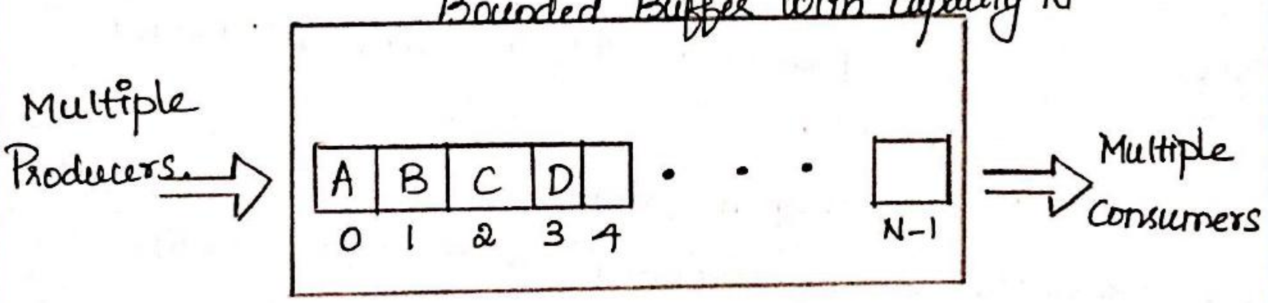
* The Bounded - Buffer problem.
* The Readers - Writers problem.
* The Dining - philosophers problem.

* The Bounded - Buffer problem :

The bounded buffer Problems (ie the producer - consumer problem) is a classic example of Concurrent access to a shared resource. A bounded buffer lets multiple Producers and multiple Consumers Share a single buffer.

Producers must block if the "Buffer is full".

Consumers must block if the "Buffer is Empty".



We assume that the ~~pool~~ consists of n buffers, each capable of holding one item.

The empty and full semaphores count the number of empty and full buffers. The semaphore empty is initialized to the value n; the semaphore full is initialized to the value 0.

The structure of the Producer process :-

```
do {
    . . .
    // Produce an item in nextp.
    . . .
   wait (empty);
   Wait (mutex);
   . . .
 // add nextp to buffer
   . . .
   signal (mutex);
   Signal (full);
 } while (TRUE);
```

We can interpet this code as the Producer Producing full buffers for the Consumer (or) as the Consumer Producing Empty buffers for the Producer.

The Structure of the Consumer process :

```
do {
    wait (full);
    wait (mutex);
    . . . .
 // remove an item from buffer to nextc
   . . .
 signal (mutex);
 signal (Empty);
   . . .
 // Consumes the item in nextc
   . . .
 } while (TRUE);
```

* The Readers – Writers Problem :

Readers : Only reads the data set they do not perform any updates.

Writers : Can both read and write.

Problem : Whenever allows multiple readers to read at the same time. Only one single writer can access the shared data at same time.

R–W and W–W ⇒ leads to ~~da~~ crash.

Shared data :-

```
    dat set ;
    Semaphore mutex = 1
    Semaphore wrt   = 1
    Integer readcount = 0.
```

Structure of writer Process : ⟨ read / write

```
    do {
        wait (wrt);
        . . .
        // writing is performed
        . . .
        signal (wrt);
    } while (TRUE);
```

Structure of reader process :

```
    do {
        wait (mutex);
        readcount ++ ;
        if (readcount = = 1)
          wait (wrt);
          signal (mutex);
            . . .
        // reading is Performed
            . . .
```

```
wait (mutex);
readcount --;
 if (readcount ==0)
   signal (wrt);
  signal (mutex);
 }while (TRUE);
```

Reader-Writer locks are most useful in the following situations :

(i) In applications where it is easy to identify which processes only read shared data and which processes only write shared data.

(ii) In applications that have more readers than writers. This is because reader-writer locks generally require more overhead to establish than semaphores (or) mutual exclusion locks. The increased concurrency of allowing multiple readers compensates for the overhead involved in setting up the reader-writer lock.

* The Dining-philosophers problem :

Consider 5 philosophers who spend their lives thinking and eating. The philosophers share a circular table surrounded by five chairs, each belonging to one philosopher.

In center of table is a bowl of rice, and the table is laid with 5 single chopsticks.

→ When philospher thinks, she does not interact with her colleagues.

→ from time to time, a philosopher gets hungry and tries to pick up the two chopsticks that are closet to her (the chopsticks that are between her and her left and right neighbors).

A philosopher may pick up only one chopstick at a time. obviously, she cannot pick up a chopstick that is already in the hand of a neighbor. When a hungry philosopher has both her chopsticks at the same time, she eats without releasing her chopsticks. When she is finished eating, she puts down both of her chopsticks and starts thinking again.

Problem : Develop an algorithm where no philosopher Staves. ie, every Philosopher should eventually get a chance to eat.

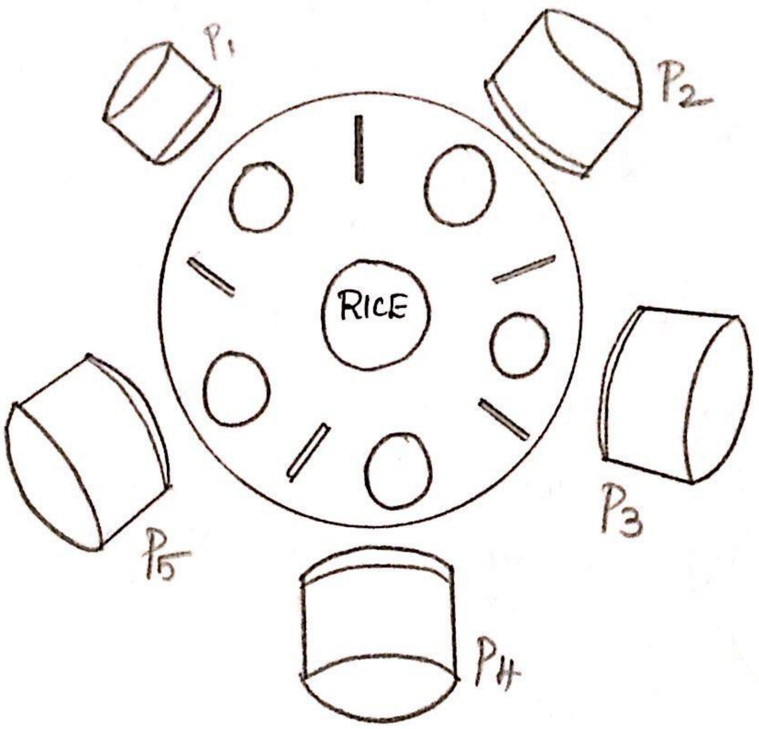
The structure of philosopher i.

Chopstick [5] initialized to 1

Structure of philosopher i;

```
do {
    wait (chopstick [i]);
    wait (chopstick [(i+1) % 5]);
    . . .
   // eat
    signal (chopstick [i]);
    signal (chopstick [(i+1) % 5]);
    . . .
   // think
    . . .
} while (TRUE);
```

The situation of the dining philosophers :



One simple solution is to represent each chopstick with a semaphore.

```
Shared data : State [5] (thinking, hungry, eating)
              Self [5] : Semaphore : { init to all 0}
              mutex : Semaphore { init to 1}
              left : (i+4) % 5 { left neighbour}
              right : (i+1) % 5, {right neighbour}
```

```
Philosopher (i)
{
  State [i] = hungry;
  wait (mutex);
  test[i];
  Signal (mutex);
  if (state [i] ! = Eating)
  wait (self [i]);
  . . . . . .
                          . . . Eating . . .
                          wait (mutex);
                          Putdown (i);
```

```
void test (int i)
{ if ((state (left)! = Eating) && (state[i] ==hungry)
              && (state (right)! = eating)
  {
     State [i] = Eating;
     Signal ( ~~self~~ self (i));
  }
void putdown (int i)
 {
   State [i] = thinking;
   wait (self (i));
  // test left & right ~~height~~ neighbours
   test (left);
   test (right);
 }
```

Several possible remedies to the deadlock problem are listed next.

* Allow at most four philosophers to be sitting simultaneously at the table.

* Allow a philosopher to pick up her chopsticks only if both chopsticks are available.

* Use an asymmetric Solution; that is, an odd Philosopher picks up first her left chopsticks and then her right chopsticks, Whereas an even philosopher picks up her right chopstick and then her left chopstick.

(21) Critical regions :

→ Regions referring to the same shared variable excluded each other in time.

→ When a process tries to execute the region statement, the Boolean expression B is evaluated. If B is true, statement S is executed. If it is false, the process is delayed until B becomes true and no other process is in the region associated with v.

Limitations :

→ Conditional critical regions are still distributed among the program code.

→ They are more difficult to implement efficiently than semaphores.

(22) Monitors :-

Monitors are based on abstract data types. A monitor is a programming language construct that provides equivalent functionality to that of semaphores but is easier to control. A monitor consists of procedures, the shared object and administrative data.

Characteristics of a monitor :-

(1) only one process can be active within the monitor at a time.

(2) The local data variables are accessible only by the monitor's procedures and not by any external procedure.

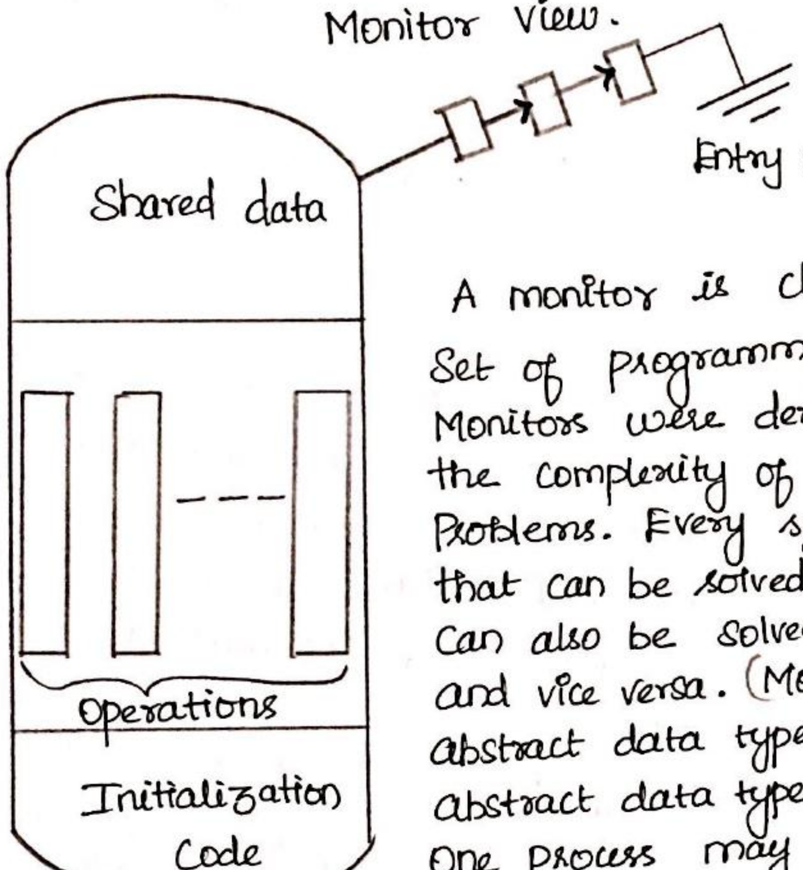
(3) A process enters the monitors by invoking one of its procedures.

(4) Monitor provides high-level of synchronization. The synchronization of process is accomplished via two special operations namely, wait and signal, which are executed within the monitors procedures.

(5) Monitors are a high level data abstraction tool combining three features:

(1) Shared data
(2) operation on data
(3) synchronization, scheduling.

Monitor view.



A monitor is characterised by a set of programmer defined operators. Monitors were derived to simplify the complexity of synchronization problem Problems. Every synchronization problem that can be solved with monitors can also be solved with semaphores, and vice versa. (Monitors are based on abstract data types. Monitor is an abstract data type for which only one process may be executing a procedure at any given time. Processes during enter to monitor when it is already in use must wait. This waiting is automatically managed by the monitor.) The above diagram shows the schematic view of a monitor.

A monitor is a software module consisting of one (or) more procedures, an initialization sequence and local data.

Syntax of monitor ;

```
Monitor  monitor-name
{
   declaration of shared variable
   Procedure body P1 ()
   {
      ---
      ---
   }
   P2 ()
  {
    procedure body
      ---
      ---
  }
      :
      :
  Pn ()
{
     Procedure body
  }  ---
       { initialization code
       }
  }
```

The monitor construct has been implemented in a number of programming languages. since monitors are a language feature, they are implemented with the help of a compiler. In response to the keywords monitor, condition, signal, wait and notify, the compiler insert little bits of code in the Program. The data variables in the monitor can be accessed by only One process at a time. A shared data structure can be protected by placing it in a monitor. The data inside the monitor may be either global to all Procedures within the monitors (or) local to a specific procedure.

A monitor supports synchronization by the use of condition variables that are contained within the monitor and accessible only within the monitor. Two condition variables are;

1) X. wait () : Suspend execution of the calling Process on condition X. The monitor is now available for use by another Process.

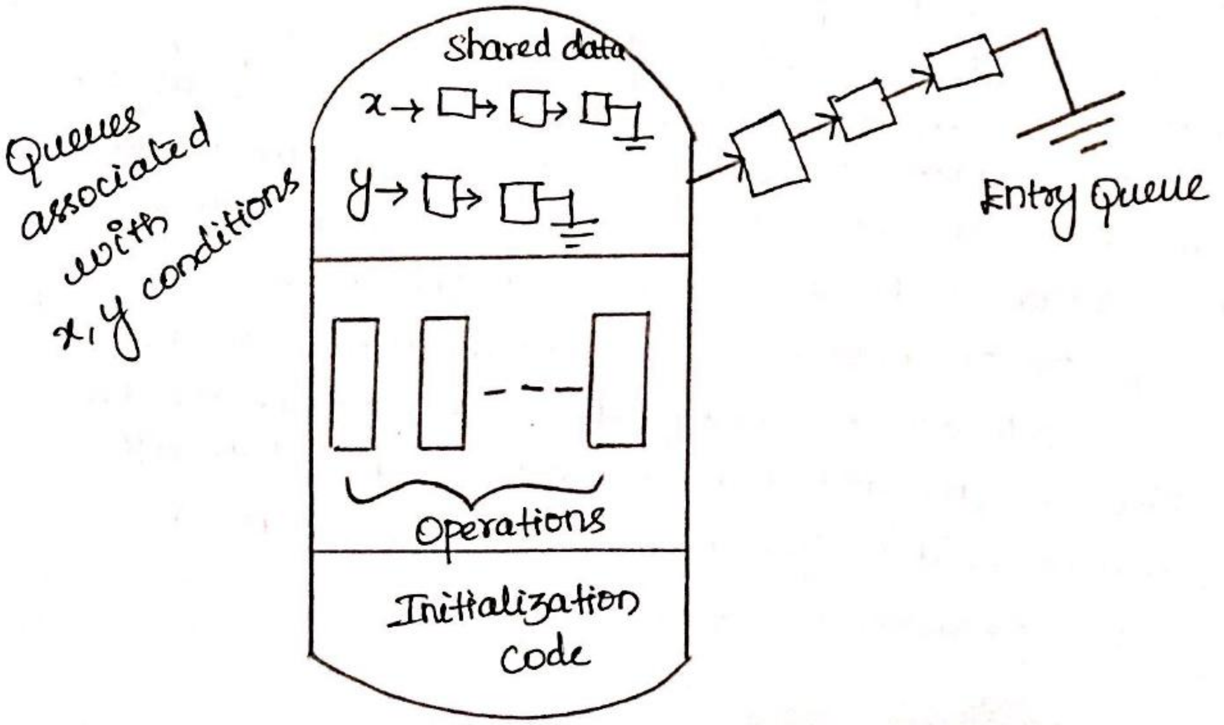
2) X. signal() : Resume execution of some process suspended after a X.wait on the same condition. This operation resumes exactly one suspended Process.

A condition variable is like a semaphore, with two differences;

(1) A semaphore counts the number of excess up operations, but a signal operation on a condition variable has no effect unless some Process is waiting. A wait on a condition variable always blocks the calling Process.

(2) A wait on a condition variable automatically does an up on the monitor mutex and blocks the caller.

Monitor with condition variables.



Monitor with Condition variables :-

Syntax:

```
Interface condition {
        Public void x.signal()
        Public void x.wait();
}
```

Each condition variable is associated with some logical condition on the state of the monitor. Condicler what happens when a Consumer is blocked on the nonempty condition variable and Producer calls add.

(1) The Producer adds the item to the buffer and calls nonempty signal().

(2) The Producer is immediately blocked and the consumer is allowed to continue.

(3) the consumer removes the item from the buffer and leaves the monitor.

(4) The Producer wakes up and since the signal operation wait the last statement is add, leaves the monitor.

They are easier and safer to use but less flexible. Many languages are not supported by the monitor. Java is making monitors much more popular and well known.

By making the mutual exclusion of critical regions automatic, monitors make parallel programming much less error prone than with semaphores.

Drawbacks of monitors:

(1) Major weakness of monitors is the absence of concurrency if a monitor encapsulates the resource, since only one process can be active within a monitor at a time.

(2) There is the possibility of deadlocks in the case of nested monitors calls.

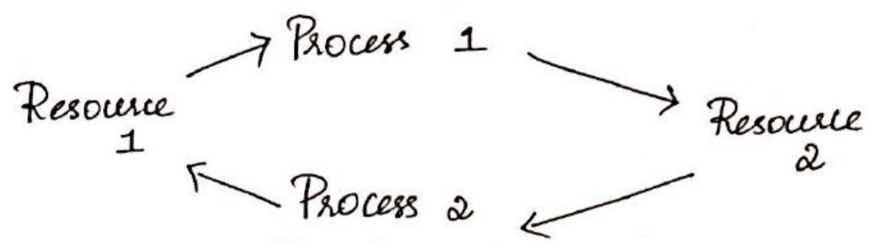
(3) Monitor Concept is its lack of implementation most commonly used programming languages.

(4) Monitors cannot easily be added if they are not natively supported by the language.

Monitors	Semaphore.
1) Monitors are based on abstract data type.	1) Semaphore is an operating system abstract data type.
2) Monitors were derived to simplify the complexity of synchronization problems by abstructing away details.	2) Semaphore - level synchronization Primitives are difficult to use for complex synchronization situations.
3) A monitor is a programming language construct that guarantee appropriate access to critical sections.	3) Semaphores provide a general-purpose mechanism for controlling access to critical sections.
4) Monitor Uses condition Variables.	4) A condition Variable is like a semaphore, with two differences.

(23) Deadlocks : [N/D 19]

A deadlock happens in operating system when two or more processes need some resource to complete their execution that is held by the other process.



Deadlock in OS.

In the above diagram, the process 1 has resource 1 and (needs to acquire) resource 2. Similarly process 2 has resource 2 and needs to acquire resource 1. Process 1 and Process 2 are in deadlock as each of them needs the other's resource to complete their execution but neither of them is willing to relinquish their resources.

(24) System models :

A System Consists of a finite number of resources to be distributed among a number of competing processes.

A Process must request a resource before using it and must release the resource after using it. A process may request as many resources ~~requested may not ex~~ as it requires to carry out its designated task. A process cannot request three printers, if the system has only two.

Under the normal mode of operation, a process may utilize a resource in only the following sequence.

Request :

The process requests the resource. If the request cannot be granted immediately (for eg., if the resource is being used by another process), then the requesting process must wait until it can acquire the resource.

Use :

The process can operate on resource (for example, if the resource is a printer), the process can print on the printer.

Release : The process releases the resource.

(25) <u>Deadlock characterization</u> :

In a deadlock, processes never finish executing, and system resources are tied up, preventing other jobs from starting.

* Necessary conditions
* Resource - Allocation graph

* Necessary Conditions : [N/D-19]

A deadlock situation can arise if the following four conditions hold simultaneously in a system.

* Mutual Exclusion
* Hold and wait
* No preemption
* circular wait

* Mutual Exclusion :

Atleast one resource must be held in a nonsharable mode ; that is, only one process at a time can use the resource. If another process requests that resource, the requesting process must be delayed until the resource has been released.

* Hold and wait :

A process must be holding atleast one resource and waiting to acquire additional resources that are currently being held by other processes.

* No Preemption :

Resources cannot be preempted ; that is, a resource can be released only voluntarily by the process holding it, after that process has completed its task.

* Circular wait :

A set $\{P_0, P_1, \ldots P_n\}$ of waiting processes must exist such that P_0 is waiting for a resource held by P_1, P_1 is waiting for a resource held by $P_2, \ldots, P_{n-1}$ is waiting for a resource held by P_n, and P_n is waiting for a resource held by P_0.

Resource allocation graph :

Deadlocks can be described more precisely in terms of a directed graph called a system resource - allocation graph.

This graph consists of a set of vertices V and set of edges E. The set of vertices V is partitioned into two different types of nodes :

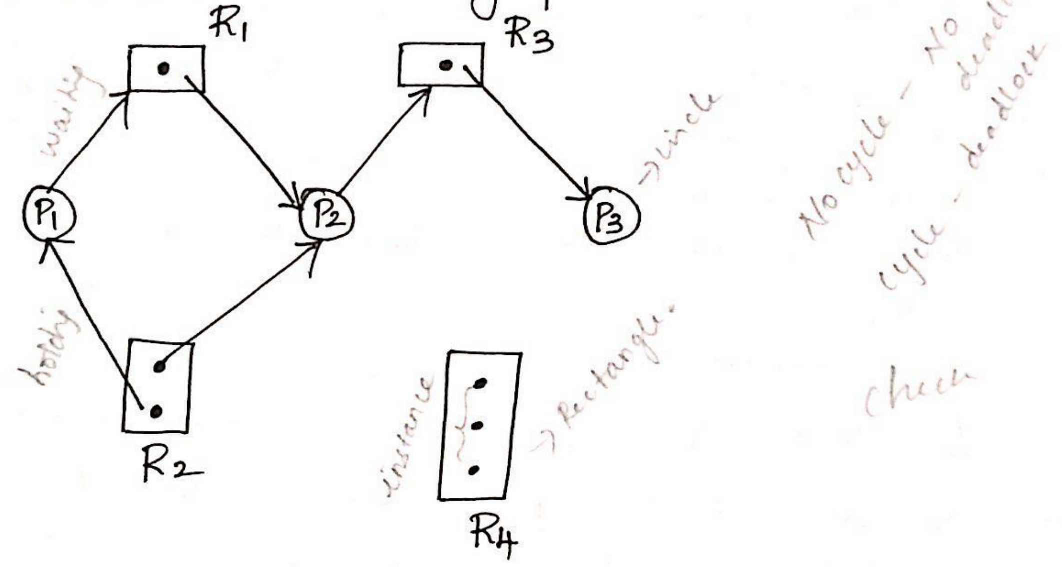
$P = \{P_1, P_2, \ldots, P_n\}$, the set consisting of all the active processes in the system, and

$R = \{ R_1, R_2, \ldots, R_m \}$, the set consisting of all resource types in the system.

A directed edge from process P_i to resource type R_j is denoted by $P_i \rightarrow R_j$; it signifies that process P_i has requested an instance of resource type R_j and is currently waiting for that resource.

A directed edge from resource type R_j to process P_i is denoted by $R_j \rightarrow P_i$; it signifies that an instance of resource type R_j has been allocated to process P_i. A directed edge $P_i \rightarrow R_j$ is called a request edge; and a directed edge $R_j \rightarrow P_i$ is called assignment edge.

Resource - allocation graph.



Pictorially, we represent each process P_i as a circle and each resource type R_j as a rectangle. Since resource type R_j may have more than one instance, we represent each such instance as a dot within the rectangle.

Note that a request edge points to only the rectangle R_j, whereas an assignment edge must also designate one of the dots in the rectangle.

* The sets P, R and E.

$P = \{P_1, P_2, P_3\}$

$R = \{R_1, R_2, R_3, R_4\}$

$E = \{P_1 \to R_1,\ P_2 \to R_3,\ R_1 \to P_2,\ R_2 \to P_2,\ R_2 \to P_1,\ R_3 \to P_3\}$.

* Resource instances :

✓ One instance of resource type R_1.

✓ Two instance of resource type R_2.

✓ One instance of resource type R_3.

✓ Three instance of resource type R_4.

* Process states :

(i) Process P_1 is holding an instance of resource type R_2 and is waiting for an instance of resource type R_1.

(ii) Process P_2 is holding an instance of R_1 and an instance of R_2 and is waiting for an instance of R_3.

(iii) Process P_3 is holding an instance of R_3.

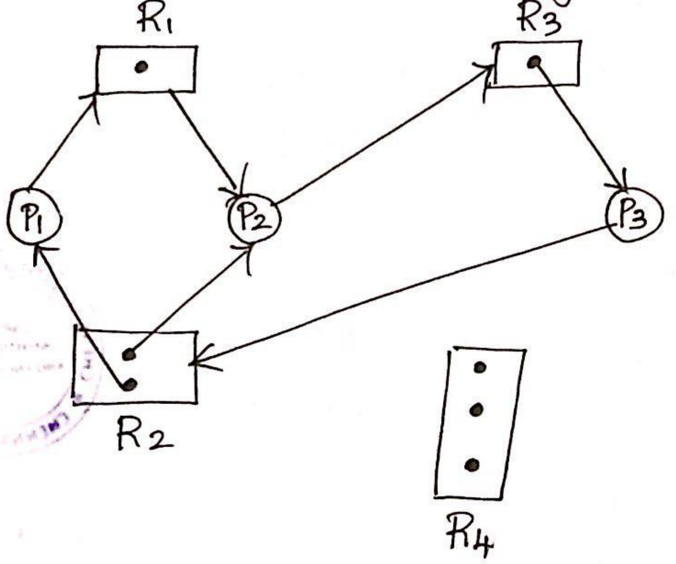
If the graph contains no cycles, then no process in the system is deadlocked. If the graph does contain a cycle, then a deadlock may exist.

If each resource type has exactly one instance, then a cycle implies that a deadlock has occurred. If the cycle involves only a set of resource types, each of which has only a single instance, then a deadlock has occurred.

Each process involved in the cycle is deadlocked. In this case, a cycle in the graph is both a necessary and a sufficient condition for the existence of deadlock.

In the dig., suppose that process P_3 requests an instance of resource type R_2.

Resource - allocation graph with a deadlock.



Since no resource instance is currently available, a request edge $P_3 \rightarrow R_2$ is added to the graph shown above.

Two minimal cycles exist in the system;

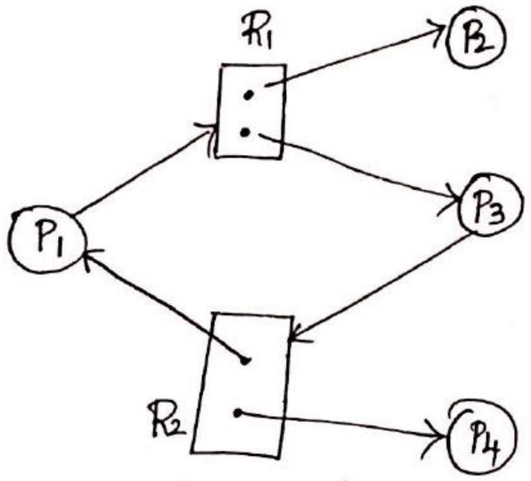
$$P_1 \rightarrow R_1 \rightarrow P_2 \rightarrow R_3 \rightarrow P_3 \rightarrow R_2 \rightarrow P_1$$

$$P_2 \rightarrow R_3 \rightarrow P_3 \rightarrow R_2 \rightarrow P_2.$$

(Processes P_1, P_2, and P_3 are deadlocked.) Process P_2 is waiting for the resource R_3, which is held by Process P_3. Process P_3 is waiting for either Process P_1 or process P_2 to release resource R_2. In addition, Process P_1 is waiting for process P_2 to release resource R_1.

Now consider the resource - allocation graph, in this example we have a cycle.

Resource - allocation graph with a cycle but no deadlock.



$P_1 \rightarrow R_1 \rightarrow P_3 \rightarrow R_2 \rightarrow P_1$.

However there is no deadlock. Observe that Process P_4 may release its instance of resource type R_2. That resource can then be allocated to P_3, breaking the cycle.

If a resource - allocation graph does not have a cycle, then the system is not in a deadlocked state. If there is a cycle, then the system may or may not be in a deadlocked state. This observation is important when we deal with the deadlock problem.

(26) Methods for Handling deadlocks :

Generally Speaking, we can deal with the deadlock Problem in one of 3 ways;

→ We can use a Protocol to Prevent or avoid deadlocks, ensuring that the system will never enter a deadlock state.

→ We can allow the system to enter a deadlocked State, detect it, and recover.

→ We can ignore the Problem altogether and Pretend that deadlocks never occur in the system.

27) Deadlock Prevention:

We can Prevent deadlock by eliminating any of the four conditions.

* Mutual Exclusion.
* Hold and wait
* No Preemption
* Circular wait

* Mutual Exclusion :

→ Not required for sharable resources (Read only files)

→ Must hold for non sharable resources.

→ cannot prevent deadlocks by denying mutual-exclusion condition (these are non sharable resources).

* Hold and Wait :

→ Must guarantee that whenever a Process requests a resources, it does not hold any other resources.

→ Requires Process to request and be allocated all its resources before it begins execution.

→ System calls requesting resources precede all Other system calls.

→ Allow process to request resources only when the process has none.

Eg; Process copying data from a tape drive to a disk file, sort the disk file and print the results to a printer.

Low resource utilization; starvation possible.

Here., the resources are

* tape drive
* disk
* Printer. All the 3 resources must be used by the process. It hold the Printer for its entire execution, even through it needs the printer only at the end. (Process that copies data from Tape drive to a disk, sorts the file and prints the results to a printer).

The second method allows the Process to request initially only the DVD drive and disk file. It copies from drive and to the disk then releases both the tape drive and disk file. Process then again request the disk and the printer. After copying the disk file to the printer, it releases these two resources and terminates.

So both protocols has two disadvantages.

* Resource Utilization is low.
* Starvation is possible.

* No Preemption :

The third necessary condition for deadlocks is that there Should be no Preemption of resources that have already been allocated.

- If a Process that is holding some resources requests another resource that cannot be immediately allocated to it, then all resources currently being held are released. It won't lead to deadlock but delay some time to complete.

- Preempted resources are added to the list of resources for which the Process is waiting.

- Process will be restarted only when it can regain its old resources, as well as the new ones that it is requesting.

- If a process requests resources, check if resources are available.
- If available, then allocate.
- If not available, check if allocated to some other process that is waiting.
- If so, preempt the resources from the waiting process.
- Allocate to the requesting process.

* Circular wait :

  - Impose a total ordering of all resource types.
  - Require that each process requests resources in an increasing order of enumeration.
  - $R = \{R_1, R_2, \dots R_n\}$ be the set of resource types.
  - Each resource type is assigned a number.

$F : R \rightarrow N$ where,

$N \rightarrow$ Set of Natural numbers

Eg: $\left.\begin{array}{l} F(\text{tape drive}) = 1 \\ F(\text{disk drive}) = 5 \\ F(\text{Printer}) = 12. \end{array}\right\}$ The numbers are in increasing order.

Protocol 1 : Each process requests only in increasing order of enumeration. After requesting R_i, can request R_j only if $F(R_j) > F(R_i)$.

Eg ; disk drive and then printer.

For several instances of same resource type, a single request for all of them must be issued.

Protocol 2 : When an instance of R_j is to be got, release all instances of R_i, if $F(R_i) \geq F(R_j)$

Suppose if circular wait exists in a system, then let $P_0, P_1, \dots P_n$ be the processes involved in the circular wait, P_i waiting for a resource held by P_{i+1}.

$$\begin{tikzcd}
 & P_1 \arrow[dr] & \\
P_5 \arrow[ur] & & P_2 \arrow[dl] \\
P_4 \arrow[u] & P_3 \arrow[l] &
\end{tikzcd}$$

$$F(R_0) < F(R_1) < \ldots < F(R_n) < F(R_0)$$

which is impossible.

$P_1 \to P_1 \to \boxed{P_2} \to$ waiting for another resource

(28) Deadlock Avoidance :

Deadlock is a state in which a Process is waiting for the resource that is already used by another Process and that another Process is waiting for another resource.

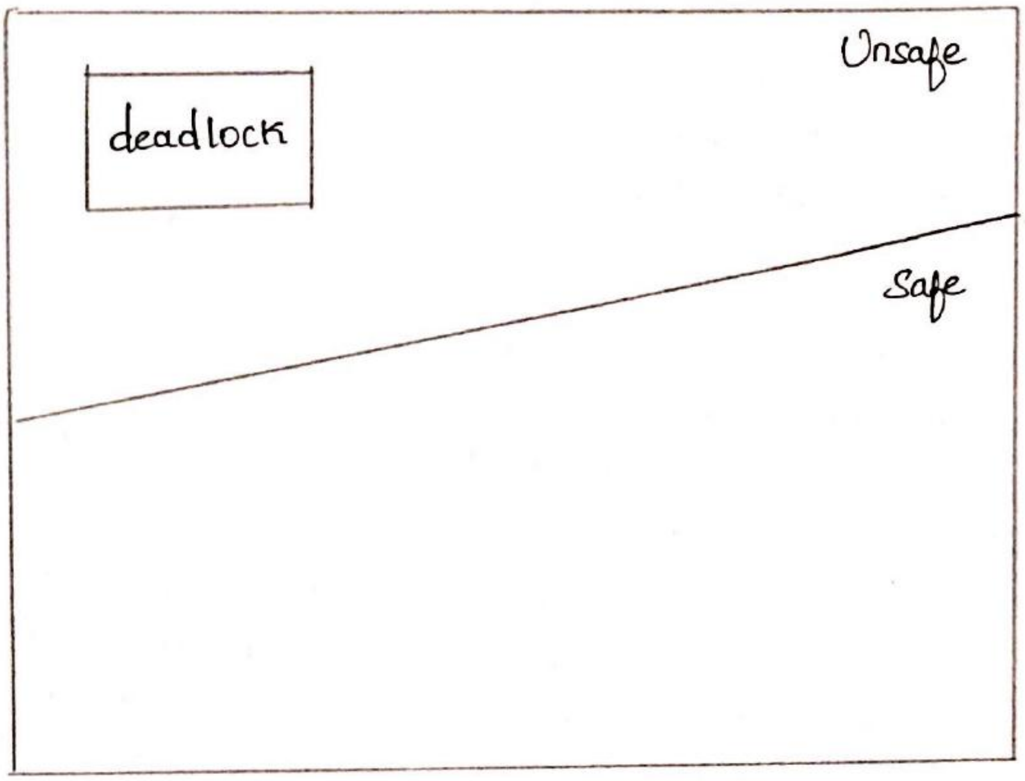
Deadlock can be avoided if certain information about processes are available to the operating system before allocation of resources. For every resource request will cause the system to enter an unsafe State that means this state Could result in deadlock. The system then only grants the requests that will lead ~~the~~ to safe states.

* Safe State
* Resource - allocation graph algorithm.
* Banker's algorithm.
    - Safety algorithm
    - Resource - Request algorithm
    - Example.

Safe State :

A State is safe if it is has a bunch of processes and there are enough resources for the 1st process to be finished and after it releases it's resources there are enough resources for the next process to be Proceed. There is no chance of deadlock.

Safe, Unsafe, and deadlocked State spaces.



Unsafe State : A State that may allow deadlock. It is possible for a process to be in an unsafe state, but for this not to result in a deadlock.

Deadlock : No further progress is possible.

In order to determine the condition of next state (safe / unsafe / deadlock) the system must know these information in advance.

✓* Resources currently available.

✓* Resources currently allocated to each process.

* Resources that will be required and released by these processes in the future.

To avoid deadlock we have to follow a simple rule. If a request causes the next state into unsafe state or deadlock then we shouldn't proceed the request.

When a process requests available resource, system must decide if immediate allocation leaves the system in a safe state. System is in safe state if there exists a sequence <P_1, P_2, ... P_n> of all the processes in the systems such that for each P_i, the resource that P_i can still request can be satisfied by currently available resources + resources held by all the P_j, with $j < i$.

ie., If P_i resource needs are not immediately available, then P_i can wait until all P_j has finished. When P_j is finished, P_i can obtain needed resources, execute, return allocated resources, and terminate. When P_i terminates, P_{i+1} can obtain its needed resources, and so on.

// Avoidance Algorithms :

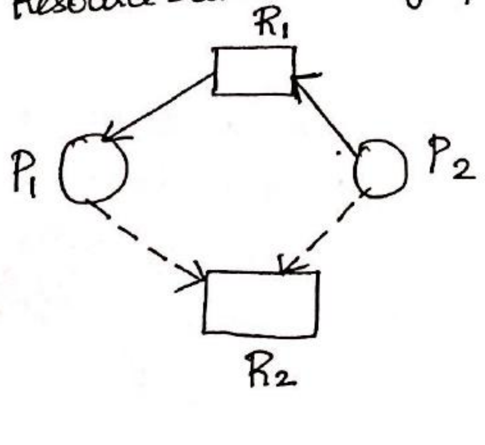
* Single Instance of a resource type. Use a resource-allocation graph.
* Multiple instance of a resource type. Use the banker's algorithm.

* Resource - Allocation - Graph Algorithm :-

In addition to request and assignment edges already described, and a new edge is introduced called a claim edge.

A claim edge $P_i \rightarrow R_j$ indicates that process P_i may request resource R_j at some time in the future. It is represented in graph by a dashed line.

Resource-allocation graph for deadlock avoidance.

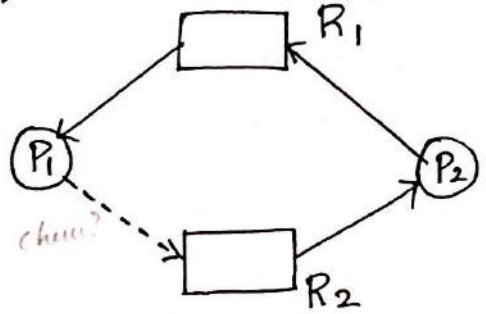


When process P_i requests resources R_j, the claim edge $P_i \to R_j$ is converted to a request edge.
Similarly, when a resource R_j is released by P_i, the assignment edge $R_j \to P_i$ is reconverted to a claim edge $P_i \to R_j$.

We note that the resources must be claimed a priori in the system. That is before process P_i starts executing, all its claim edges must already appear in the resource-allocation graph. We can relax this condition by allowing a claim edge $P_i \to R_j$ to be added to the graph only if all the edges associated with process P_i are claim edges.

If no cycle exists, then the allocation of the resource will leave the system in a safe state. If a cycle is found, then the allocation will put the system in an unsafe state. In that case, process P_i will have to wait for its requests to be satisfied.

An Unsafe State in a resource-allocation graph.



Suppose that P_2 requests R_2. Although R_2 is currently free, we cannot allocate it to P_2, since this action will create a cycle in the graph. A cycle, as mentioned, indicates that the system is in an unsafe state. If P_1 requests R_2, and P_2 requests R_1, then a deadlock will occur.

Banker's Algorithm :

This algorithm is used in a banking system to ensure that the bank never allocated its available cash in such a way that it could no longer satisfy the needs of all its customers.

Several data structures must be maintained to implement the banker's algorithm.

$n \rightarrow$ the number of processes in the system.

$m \rightarrow$ the number of resource types.

Available : Vector of length m. If available [j] = k, there are k instances of resource type R_j, available.

Max : n×m matrix. If Max [i, j] = k, then process P_i may request at most k instance of resource type R_j.

Allocation : n×m matrix. If Allocation [i, j] = k then, P_i is currently allocated k instances of R_j.

Need : n×m matrix. If need [i, j] = k, then P_i may need k more instances of R_j to complete its tasks.

$$Need [i, j] = Max [i, j] - Allocation [i, j].$$

Safety Algorithm :

1) Let work and Finish be vectors of length m and n, respectively. Initialize work = Available and Finish [i] = false, for i = 0, 1, . . , n-1.

2) Find an index i such that both

a) Finish [i] = = false

b) $Need_i \leq work$.

If no such i exists, go to step 4.

3) Work = Work + $Allocation_i$

Finish [i] = true

Goto step 2.

4) If Finish [i] = = true for all i, then the system is in a safe state.

Resource - Request Algorithm :

Let Request i be the request vector for Process P_i. If $Request_i[j] == k$, then Process P_i wants k instances of resource type R_j. When a request for resources is made by process P_i, the following actions are taken.

1) If $Request_i \leq Need_i$, go to step 2. Otherwise, raise an error condition, since the process has exceeded its maximum claim.

2) If $Request_i \leq Available$, go to Step 3. Otherwise, P_i must wait, since the resource are not available.

3) Have the system pretend to have allocated the requested resources to process P_i by modifying the state as follows;

$$Available = Available + Request_i ;$$
$$Allocation = Allocation_i + Request_i ;$$
$$Need_i = Need_i - Request_i ;$$

Eg; consider a system with five processes P_0 through P_4 and three resource types A, B and C.

Resource type	Instance
A	10
B	5
C	7.

suppose that, at time T_0 the following snapshot of the system has been taken :

	Allocation			Max			Available		
	A	B	C	A	B	C	A	B	C
P_0	0	1	0	7	5	3	3	3	2
P_1	2	0	0	3	2	2			
P_2	3	0	2	9	0	2			
P_3	2	1	1	2	2	2			
P_4	0	0	2	4	3	3			

Need : Max - Allocation.

Need.

	A	B	C
P_0	7	4	3
P_1	1	2	2
P_2	6	0	0
P_3	0	1	1
P_4	4	3	1

We claim that the system is currently in a safe state. Indeed the sequence $\langle P_1, P_3, P_4, P_2, P_0 \rangle$ Satisfies the safety criteria.

Suppose now that process P_1 requests one additional instance of resource type A and two instances of resource type C, so $Request_1 = (1, 0, 2)$. To decide whether this request can be immediately granted, we first check that $Request_1 \leq Available$ - that is, that $(1, 0, 2) \leq (3, 3, 2)$ which is true. We arrive at the following new state.

	Allocation			Need			Available		
	A	B	C	A	B	C	A	B	C
P_0	0	1	0	7	4	3	2	3	0
P_1	3	0	2	0	2	0			
P_2	3	0	2	6	0	0			
P_3	2	1	1	0	1	1			
P_4	0	0	2	4	3	1			

To check whether the new system is safe, find $\langle P_1, P_3, P_4, P_0, P_2 \rangle$ satisfies the safety requirement.

(3, 3, 0) by P_4 can't be granted &

(0, 2, 0) by P_0 can't be granted even though the resources are available, since the resulting state is Unsafe.

(29) Deadlock detection :

* Single Instance of each Resource type.
* Several Instance of a Resource type.
* Detection – Algorithm Usage.

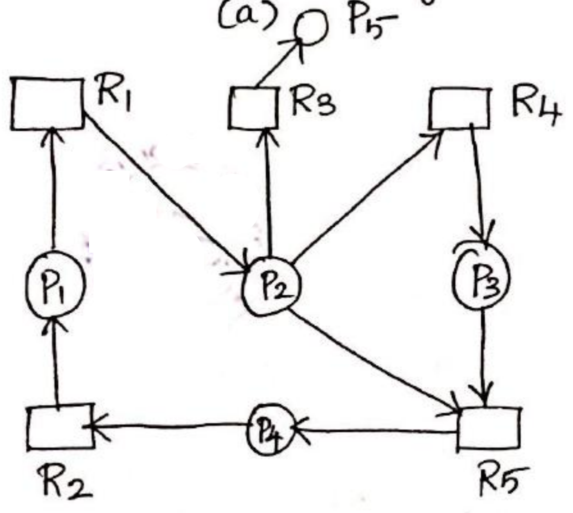
* Single Instance of each Resource type :

If all resources have only a single Instance, then we can define a deadlock-detection algorithm, called wait-for graph.

Edge from P_i to P_j is a wait-for graph implies that process P_i is waiting for process P_j to release a resource that P_i needs. An edge $P_i \rightarrow P_j$ exists in a wait-for graph if and only if the corresponding resource allocation graph contains two edges $P_i \rightarrow R_q$ and $R_q \rightarrow P_j$ for some resource R_q.

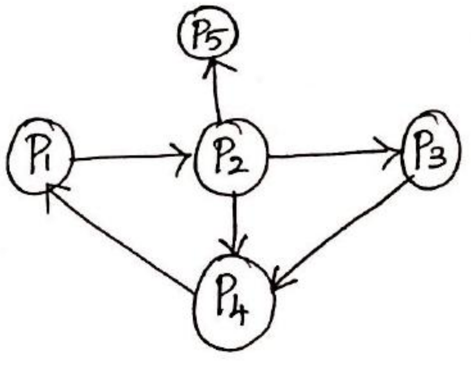
Resource allocation graph.

(a)



Corresponding wait-for graph

(b)



Deadlock exists if and only if the wait-for-graph contains a cycle.

* Several Instances of a Resource Type :

Available : A vector of length m indicates the number of available resources of each type.

Allocation : An $n \times m$ matrix defines the number of resources of each type currently allocated to each process.

Request : An $n \times m$ matrix indicates the current request of each process. If Request $[ij] = k$, then Process P_i is requesting k more instances of resource type. R_j.

Detection algorithm:

1) Let work and Finish be vectors of length m and n, respectively Initialize;

(a) work = Available

(b) for $i = 1, 2, \ldots, n$, if Allocation$_i \neq 0$, then Finish [i] = false; Otherwise, Finish [i] = true.

2) find an index i such that both :

(a) Finish [i] = = false

(b) Request$_i \leq$ work.

If no such i exists, go to step 4.

3) Work = Work + Allocation$_i$

Finish [i] = true

goto step 2.

4) If Finish [i] == false, for some i, $1 \leq i \leq n$, then the system is in deadlock state. Moreover, if Finish [i] == false, then P_i is deadlocked.

Eg., 5 processes P_0 through P_4 ; 3 resource types.

A (7 instances), B (2 instances), C (6 instances).

Snapshot at time T_0 :

	Allocation	Request	Available
	A B C	A B C	A B C
P_0	0 1 0	0 0 0	0 0 0
P_1	2 0 0	2 0 2	
P_2	3 0 3	0 0 0	
P_3	2 1 1	1 0 0	
P_4	0 0 2	0 0 2	

Sequence $\langle P_0, P_2, P_3, P_1, P_4 \rangle$ will result in Finish [i] = true for all i.

P_2 requests an additional instance of type C.

	Request
	A B C
P_0	0 0 0
P_1	2 0 1
P_2	0 0 1
P_3	1 0 0
P_4	0 0 2

State of system?

→ can reclaim resources held by process P_0, but insufficient resources to fulfill other processes, requests.

→ Deadlock exists, consisting of processes P_1, P_2, P_3 and P_4.

* Detection - Algorithm Usage:

1) How often is a deadlock likely to occur?

2) How many processes will be affected by deadlock when it happens?

(30) Recovery from deadlock.

* Process Termination

* Resource Preemption

Process Termination

* Abort all deadlocked processes.
* Abort one process at a time until the deadlock cycle is eliminated.

In which order should we choose to abort?

- Priority of the process.
- How long process has computed, and how much longer to completion.
- Resources process needs to complete.
- How many processes will need to be terminated.
- Is process interactive (or) batch?

Resource Preemption :

If Preemption is required to deal with deadlocks, then three issues need to be addressed.

Selecting a victim
Rollback
Starvation.

Selecting a victim : Which resources and which processes are to be preempted? As in process termination, we must determine the order of preemption to minimize cost.

Cost factors:-
1) Number of resources a deadlocked process is holding.
2) Amount of time the process consumed during its execution.

Rollback :

We must rollback the process to some safe state and restart it from that state.

Rollback – Abort the process & restart it.

Starvation :

Same process may always be picked as victim, include number of rollback is cost factor.

UNIT - III

STORAGE MANAGEMENT

1) Main Memory.
2) Background
3) Swapping
4) Contiguous Memory Allocation.
5) Paging
6) Segmentation
7) Segmentation with paging.
8) 32 and 64 bit architecture
9) Examples
10) virtual memory
11) Background
12) Demand paging
13) Page Replacement
14) Allocation
15) Thrashing
16) Allocating kernel Memory
17) OS Examples.

(1) Main Memory :

Memory management is the functionality of an operating system which handles (or) manages primary memory and moves processes back and forth between main memory and disk during execution.

It keeps track of each and every memory location, regardless of either it is allocated to Processes. It decides which Process will get memory at what time. It tracks whenever some memory gets freed (or) Unallocated and correspondingly it updates the status.

(2) Background :

* Basic Hardware
* Address Binding
* Logical Vs physical Address space
* Dynamic Loading
* Dynamic Linking & shared Libraries

In general, to run a Program, it must be brought into memory.

Input Queue - Collection of Processes on the disk that are waiting to be brought into memory to run the program.

User Programs go through Several steps before being run.

Address binding : Mapping of instructions and data from one address to another address in memory.

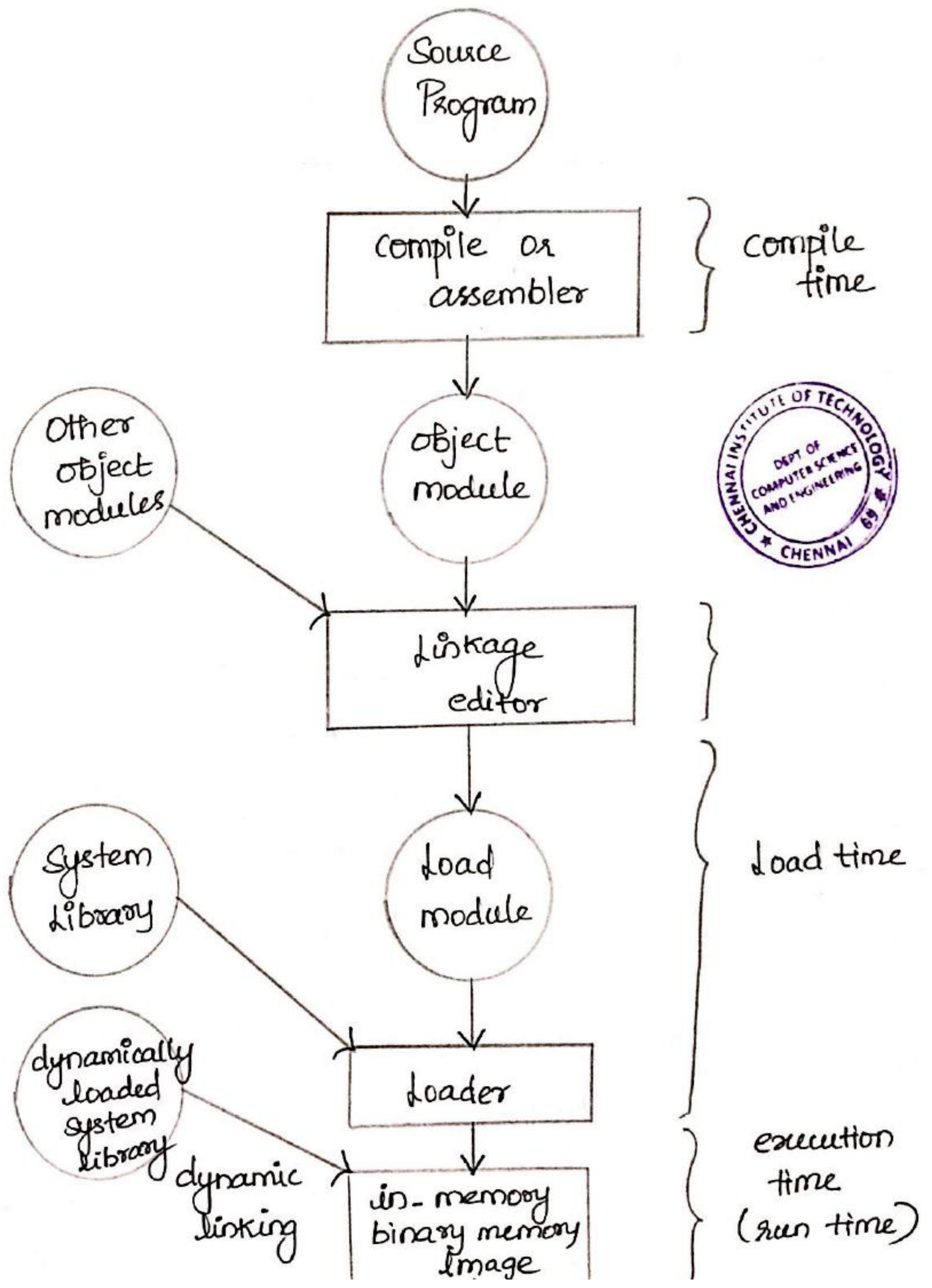
Basic Hardware and Address Binding :

1) compile time : Must generate absolute code if memory location is known in prior.

2) Load time : Must generate relocatable code if memory location is not known at compile time.

3) Execution time : Need hardware support for address maps (eg., base and limit registers).

Multistep Processing of a User Program.



* Logical Vs Physical address space : [N/D - 2019]

An address generated by the cpu is commonly referred to as a logical address, whereas an address seen by the memory unit - that is, the one loaded into the memory - address register of the memory - is commonly referred to as a physical address.

Logical and physical addresses are the same in compile - time and load - time address - binding schemes. Logical (virtual) and physical addresses differ in execution - time address - binding scheme.

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 address space.

Memory - Management Unit (MMU) :

✓ The run time mapping from virtual to physical addresses is done by a hardware device called the memory - management unit (MMU).

✓ It is a hardware device that maps virtual / logical address to physical address.

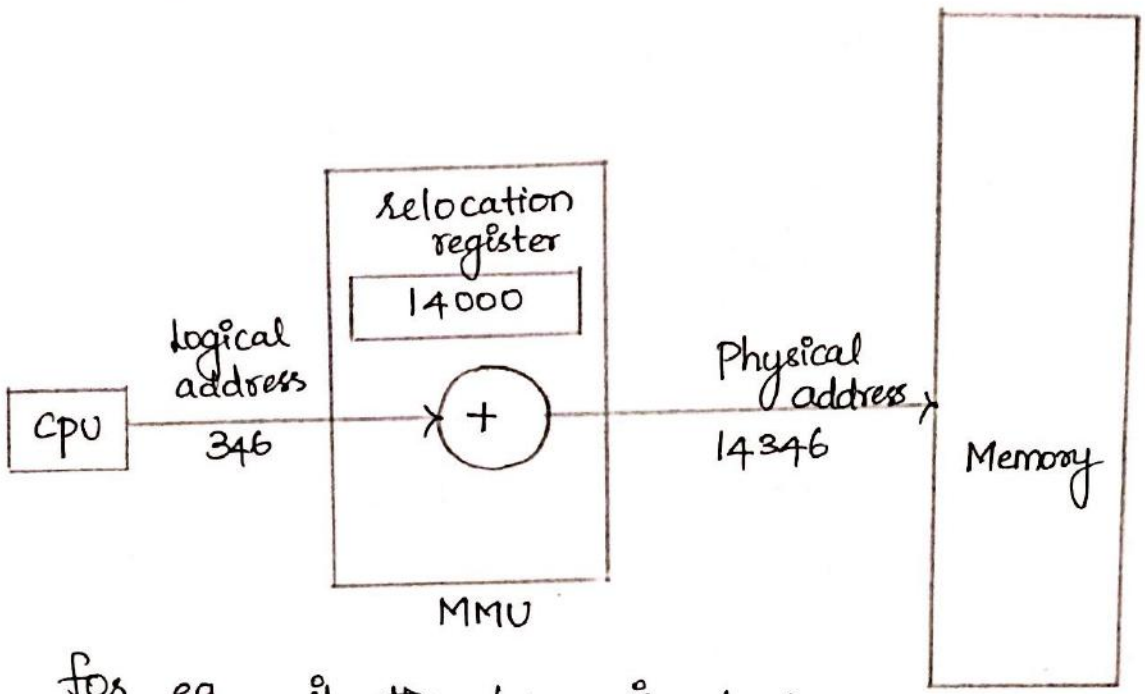
✓ In this scheme, the relocation register's value is added to logical address generated by a user process.

✓ The user program deals with logical addresses; it never sees the real physical addresses.

✓ Logical address range : 0 to max.

✓ Physical address range : R+0 to R+ max, where R - value in relocation register.

Dynamic relocation using a relocation register.



for eg., if the base is at 14000, then an attempt by the user to address location 346 is mapped to location 14346. The User Program deals with logical addresses. The memory-mapping hardware converts logical addresses into physical addresses.

Dynamic Loading :

Through this, the routine is not loaded until it is called.

* Better memory-space utilization ; Unused routine is never loaded.

* Useful when large amounts of code are needed to handle infrequently occuring cases.

* No special support from the operating system is required implemented through program design.

Dynamic Linking and Shared Libraries :

✓ Linking postponed until execution time and is particularly useful for libraries.

✓ Small piece of code called stub, used to locate the appropriate memory – resident library routine (or) function.

✓ Stub replaces itself with the address of the routine, and executes the routine.

✓ operating system needed to check if routine is in processes' memory address.

Shared libraries : Programs linked before the new library was installed will continue using the older library.

(3) Swapping : [N|D-19]

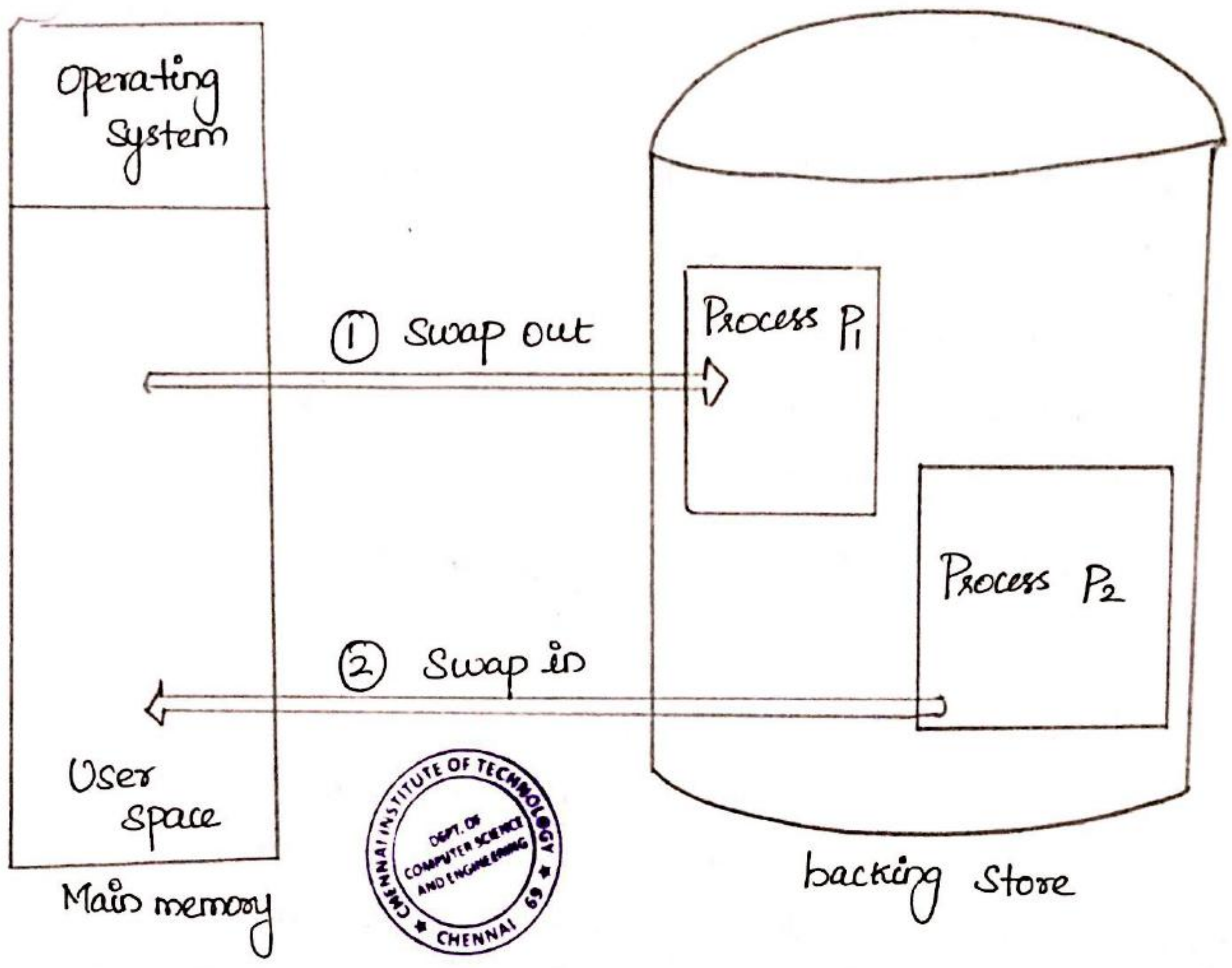
A process must be in memory to be executed. A process can be swapped temporarily out of memory to a backing store (SWAP OUT) and then brought back into memory for continued execution (SWAP IN).

✓ Backing Store – fast disk large enough to accommodate copies of all memory images for all users and it must provide direct access to these memory images.

✓ Roll out, Roll in – swapping variant used for priority – based scheduling algorithms; lower – priority process is swapped out so higher – priority process can be loaded and executed.

✓ Transfer time – Major part of swap time is transfer time. Total transfer time is directly proportional to the amount of memory swapped.

Swapping of two processes using a disk as a backing store.



* Swapping requires a backing store (normally a fast disk)

* The backing store must be big enough to accommodate all copies of memory images for all users, and must Provide direct access.

* The system has a ready queue with all processes, whose memory images are on the backing store or in memory and ready to run.

→ The cpu scheduler calls the dispatcher before running a process.

→ The dispatcher checks if the next process in queue is in memory.

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

* The context - switch time in such a swapping System is fairly high.

Eg; Let us assume the user process is of size 1 MB and the backing store is a standard hard disk with a transfer rate of 5 MBps.

Transfer time = 1000 KB / 5000 KB per second.
= 1/5 sec = 200 ms.

(4) Contiguous Memory Allocation :

It is a classical memory allocation model that assigns a process consecutive memory blocks (that is, memory blocks having consecutive addresses).

Contiguous memory allocation is one of the oldest memory allocation schemes. When a process needs to execute, memory is requested by the process. Each process is contained in a single contiguous Section of memory.

There are two methods namely;

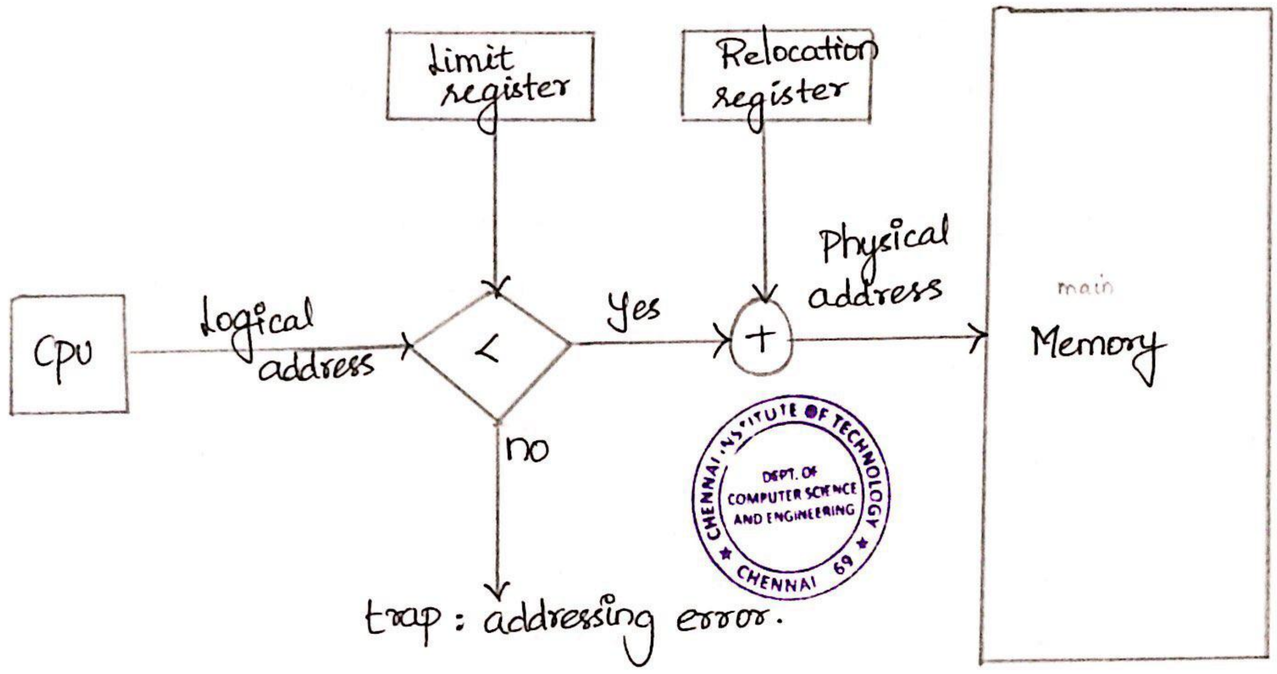
* fixed - partition method.
* Variable - partition method.

fixed - partition Method :

⇒ Divide memory into fixed size partitions, where each partition has exactly one process.

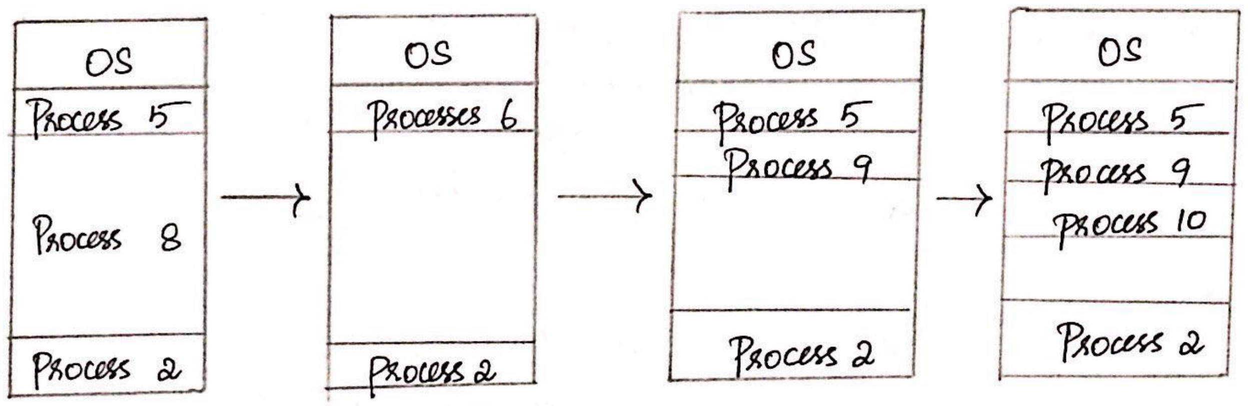
The drawback is memory space unused within a partition is wasted. Eg., When Process size < Partition size.

Hardware support to relocation and limit registers.



Variable - Partition Method :

* Divide memory into variable size partitions, depending upon the size of the incoming Process. When a Process terminates, the partition becomes available for another process. As processes complete and leave they create holes in the main memory.



Hole – Block of available memory ; holes of various size are scattered throughout memory.

Dynamic storage - Allocation Problem

How to satisfy a request of size 'n' from a list of free holes?

Solution :

✓ first - fit : Allocate the first hole that is big enough.

✓ Best - fit : Allocate the smallest hole that is big enough ; must Search entire list ; Unless Ordered by size. Produces the smallest leftover hole.

✓ Worst - fit : Allocate the largest hole ; must also search entire list. Produces the largest leftover hole.

Note : first - fit and best - fit are better than worst-fit in terms of speed and Storage utilization.

Fragmentation :

* External fragmentation
* Internal fragmentation.

External fragmentation :

This takes place when enough total memory space exists to satisfy a request, but it is not contiguous ie., Storage is fragmented into a large number of small holes scattered throughout the main memory.

Internal fragmentation :

Allocated memory may be slightly larger than requested memory. Eg : hole = 184 bytes.
Process size = 182 bytes.

We are left with a hole of 2 bytes.

Solutions :

1) Coalescing : Merge the adjacent holes together.

2) Compaction : Move all processes towards one end of memory, hole towards other end of memory, Producing one large hole of available memory. This scheme is expensive as it can be done if relocation is dynamic and done at execution time.

3) Permit the logical address space of a process to be non-contiguous. This is achieved through two memory management schemes namely paging and segmentation.

(5) Paging : [N|D-19]

- ✓ Basic Method
- ✓ Hardware Support
- ✓ Protection
- ✓ Shared pages
- ✓ Structure of Page table

Paging : It is a memory management scheme that Permits the physical address space of a process to be noncontiguous. It avoids the considerable problem of fitting the varying size memory chunks on to the backing store.

Basic Method :-

→ Divide logical memory into blocks of same size called "pages"

⇒ Divide physical memory into fixed-sized blocks called "frames"

⇒ Page size is a power of 2, between 512 bytes and 16 MB.

Address Translation scheme :

Address generated by cpu (logical address) is divided into ;

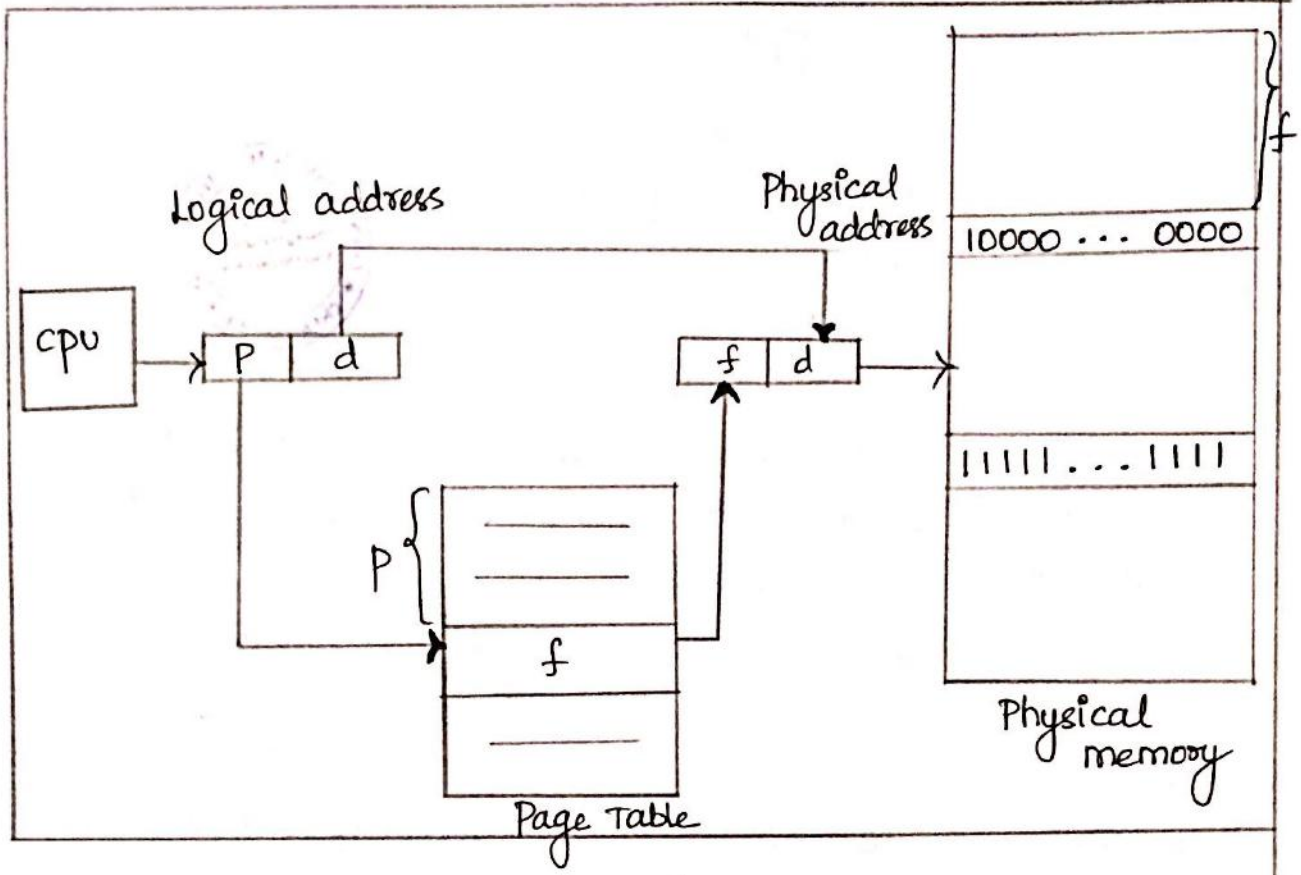
✓ Page number (P) - Used as an index into a page table which contains base address of each page in Physical memory.

✓ Page offset (d) - Combined with base address to define the physical address ie.,

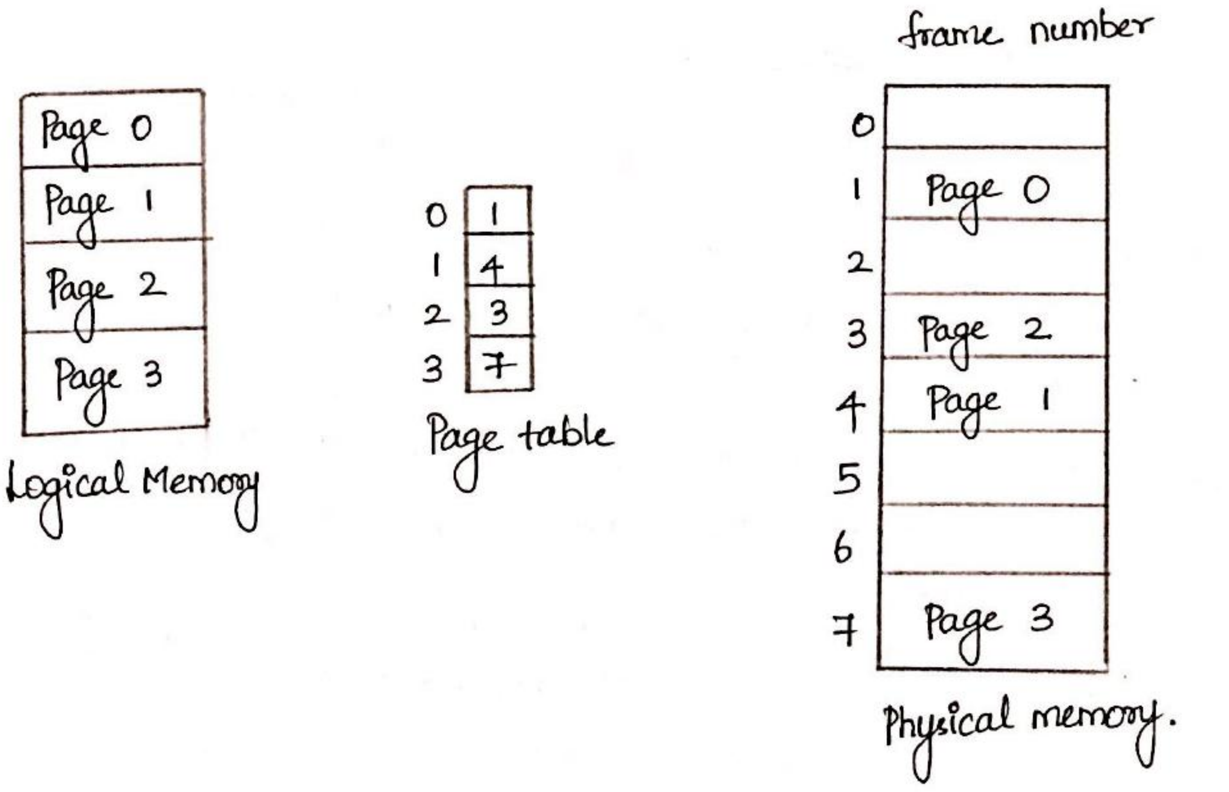
Physical address = base address + offset.

* Hardware support :

Paging Hardware :



Paging model of logical and physical memory.



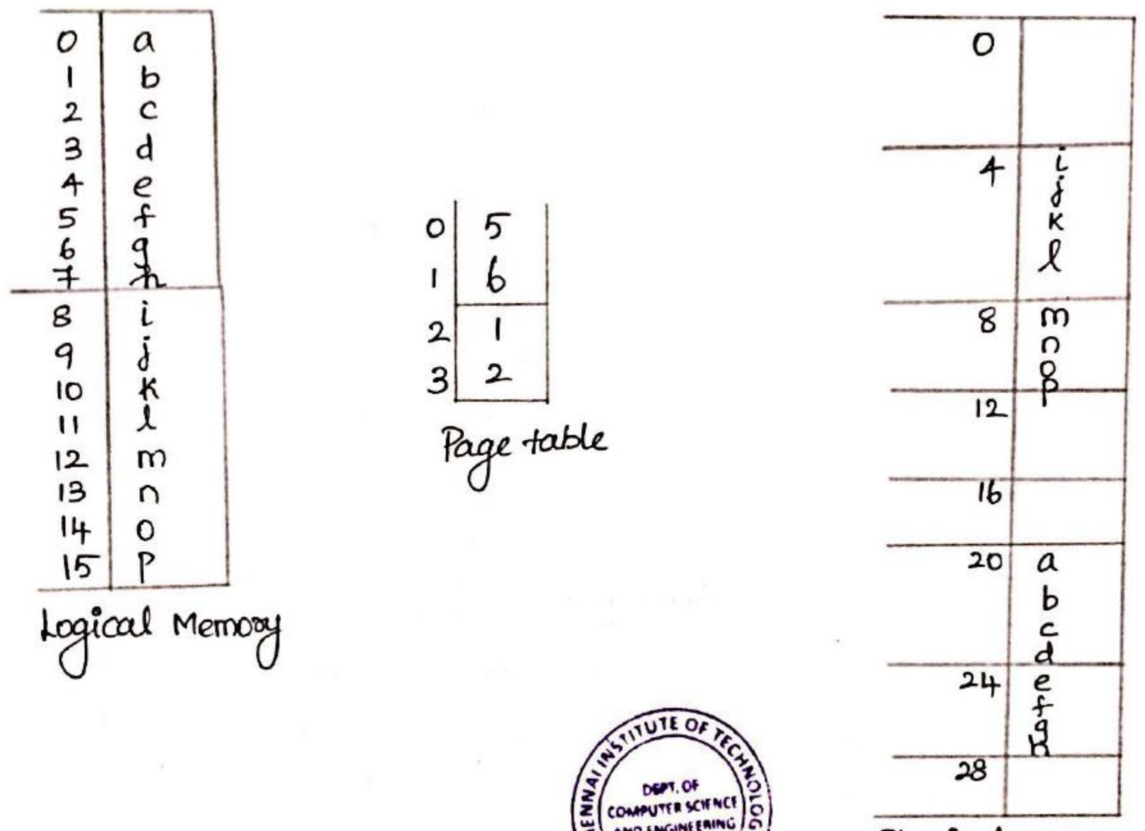
Paging example for a 32-byte memory with 4-byte pages:-

Page size = 4 bytes.

Physical memory size = 32 bytes ie., (4 × 8 = 32 so, 8 pages)

Logical address = 0 (maps to physical address 20 ie., (5×4)+0) when Frame no = 5, Page size = 4, offset = 0.

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



0	a
1	b
2	c
3	d
4	e
5	f
6	g
7	h
8	i
9	j
10	k
11	l
12	m
13	n
14	o
15	p

Logical Memory

0	5
1	6
2	1
3	2

Page table

0	
4	i j k l
8	m n o p
12	
16	
20	a b c d
24	e f g h
28	

Physical Memory

Allocation:

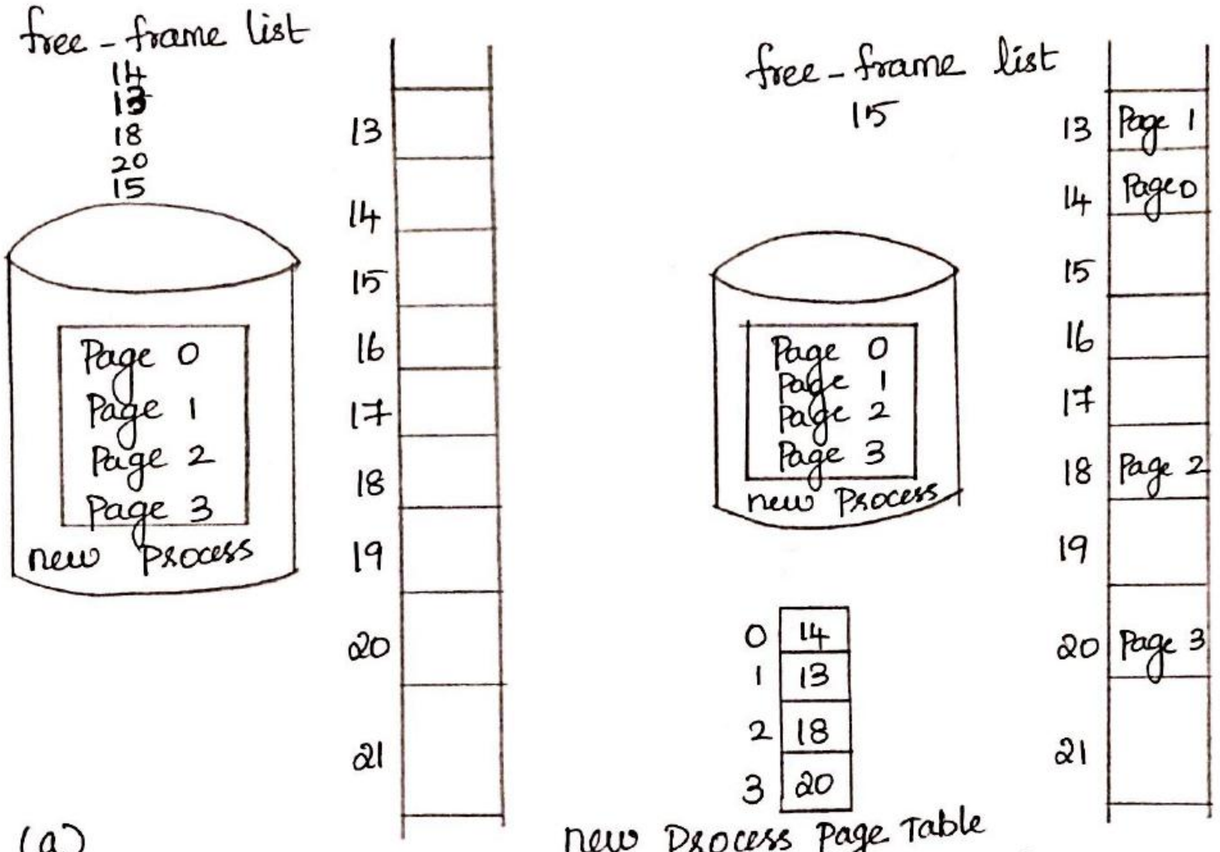
✓ When a process arrives into the system, its size (expressed in pages) is examined.

✓ Each page of process needs one frame. Thus if the Process requires n pages, at least n' frames must be available in memory.

✓ If n' frames are available, they are allocated to this arriving process.

* The 1st page of the Process is loaded into one of the allocated frames and the frame number is put into the page table.
* Repeat the above step for the next pages and so on.

Free frames (a) Before allocation (b) After Allocation



Frame table : It is used to determine which frames are allocated, which frames are available, how many total frames are there, and so on. (ie) It contains all the information about the frames in the Physical memory.

Hardware implementation of Page Table :

This can be done in several ways :

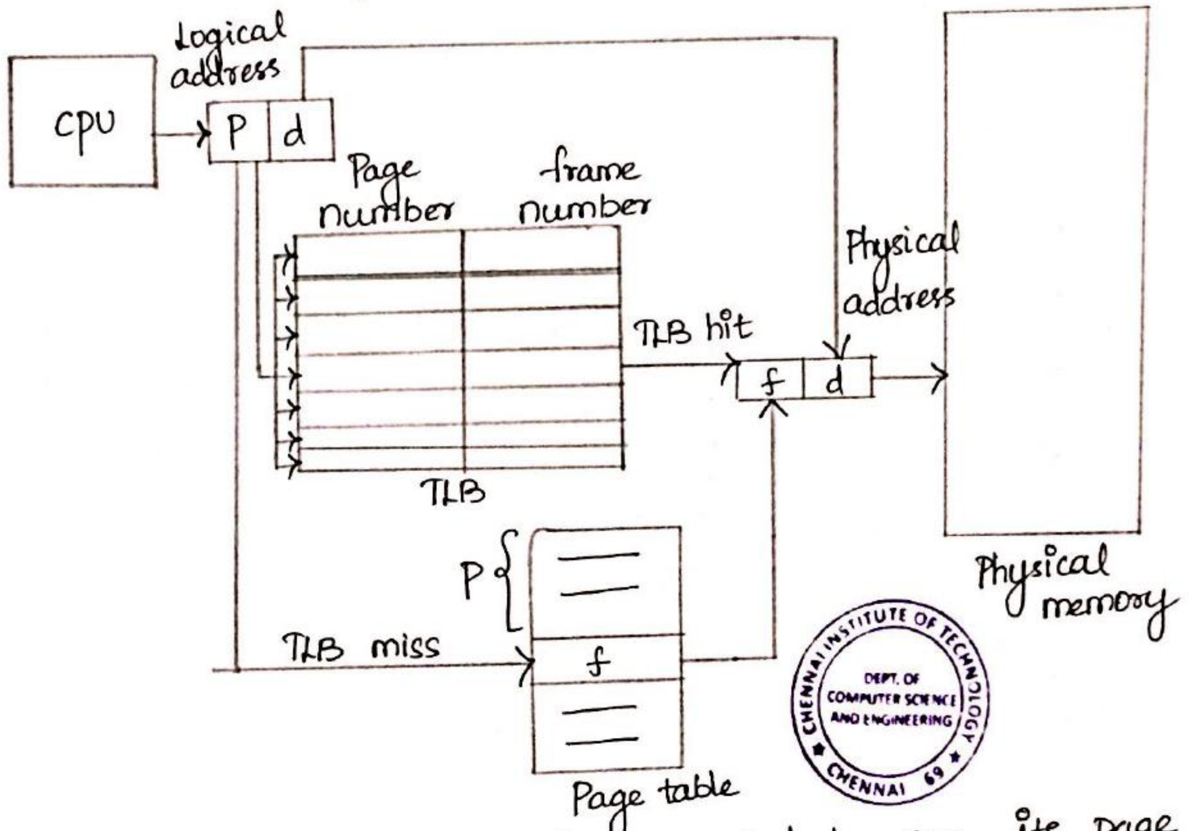
1) Using PTBR
2) TLB.

The simplest case is page - table base register (PTBR), is an index to point the page table.

TLB (Translation Look-aside Buffer)

- It is a fast lookup hardware cache.
- It contains the recently or frequently used page table entries.
- It has two parts : Key (tag) & Value.
  More expensive.

Paging Hardware with TLB.



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

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

TLB miss : If the page number is not in the TLB, a memory reference to the page table must be made.

Hit ratio : Percentage of times that a particular page is found in the TLB.

⇒ for Eg., Hit ratio is 80% means that the desired page number in the TLB is 80% of the time.

Effective Access time :

- Assume hit ratio is 80%.
- If it takes 20 ns to search TLB and 100 ns to access memory, then the memory access takes 120 ns (TLB hit)
- If we fail to find page no. in TLB (20 ns), then we must 1st access memory for page table (100 ns) and then access the desired byte in memory (100 ns).

$\therefore$ Total = 20 + 100 + 100
= 220ns (TLB miss)

Then Effective Access Time (EAT) = 0.80 x (120 + 0.20) x 220
= 140 ns.

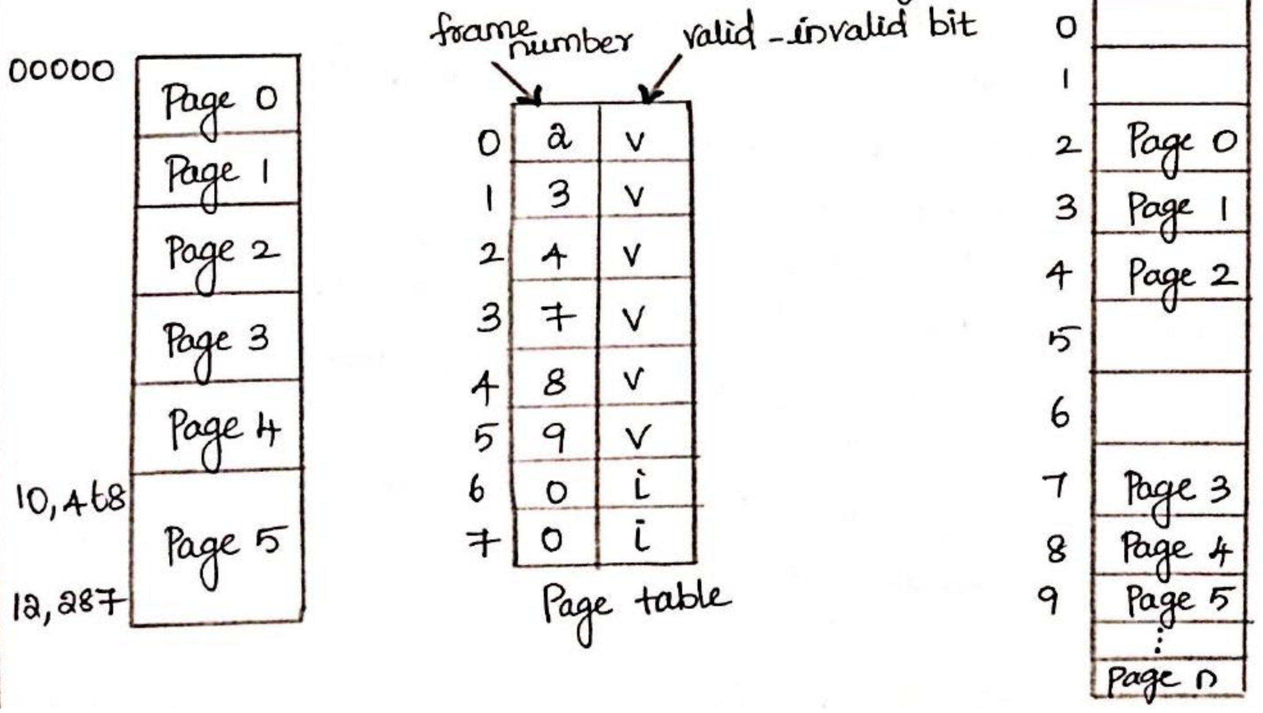
* Protection :

Memory protection implemented by associating Protection bit with each frame. Valid - invalid bit attached to each entry in the page table:

✓ Valid (v) " indicates that the associated page is in the process' logical address space, and is thus a legal page.

✓ Invalid (i) " indicates that the page is not in the Process' logical address spaces.

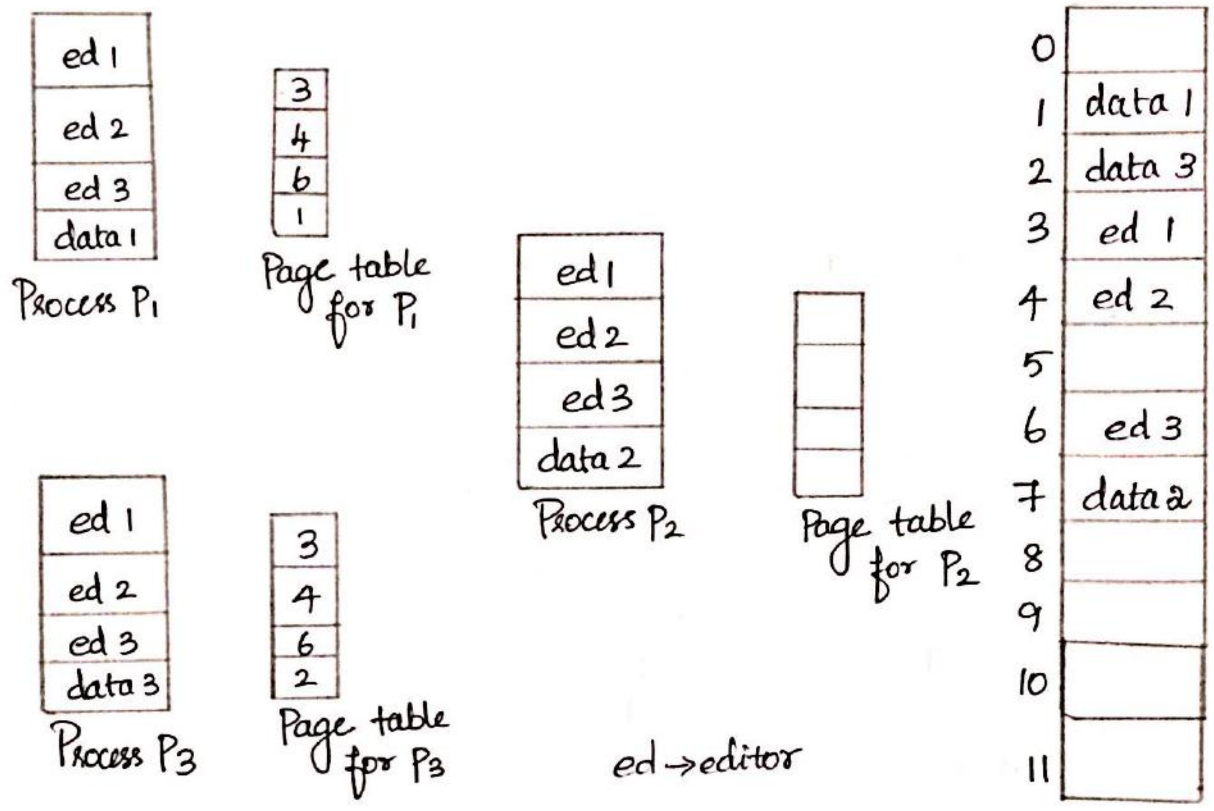
Valid (v) or invalid (i) bit in a page table.



* Shared pages :

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

Sharing of code in a paging environment.

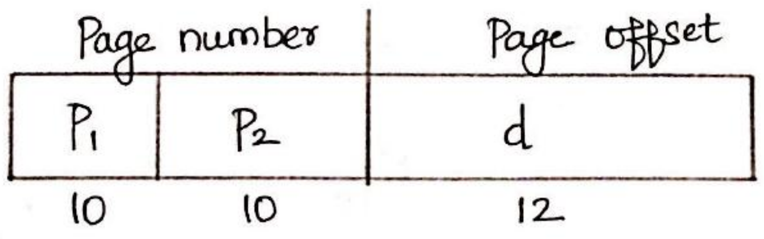


Structures of the page Table :

Hierarchical paging :

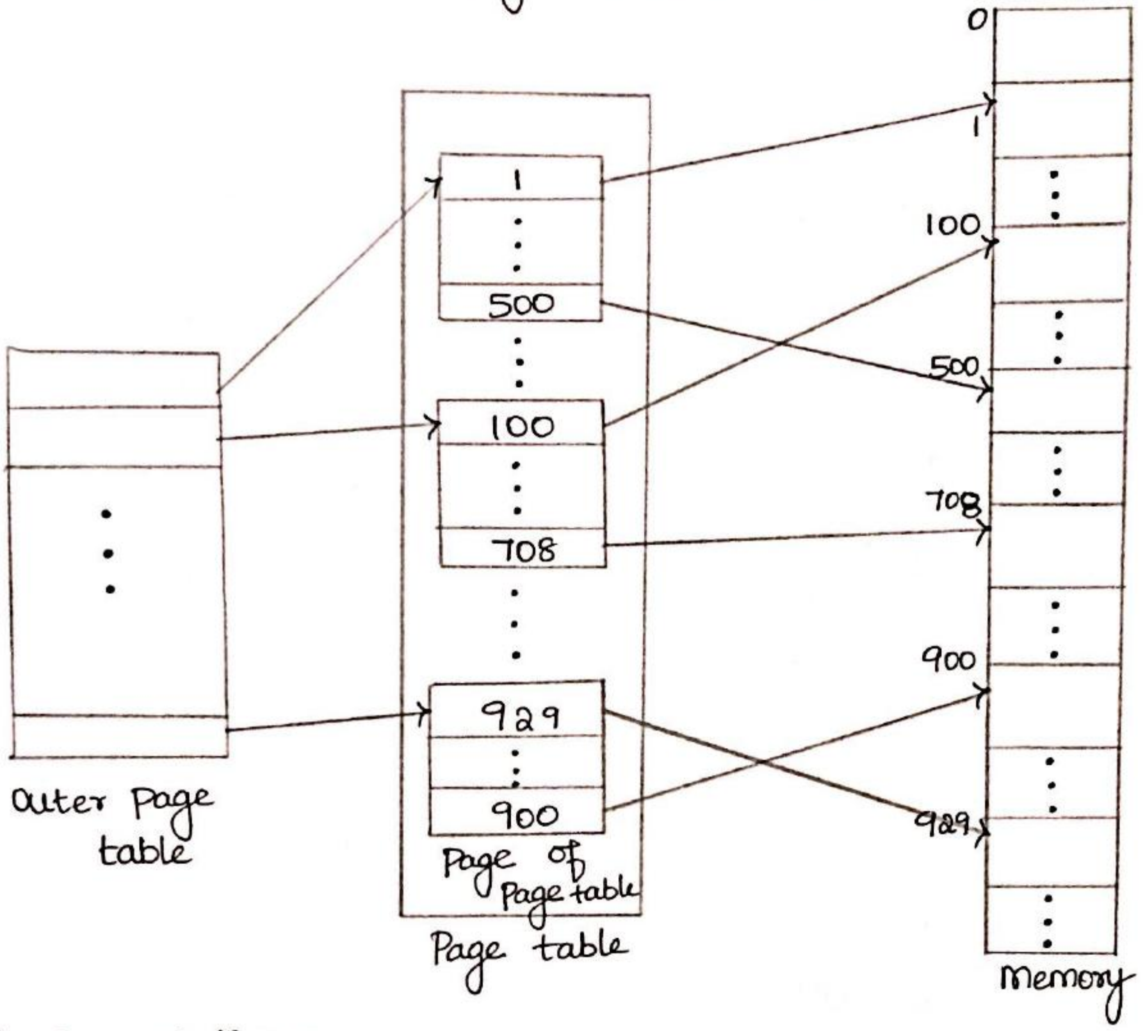
Break up the page table into smaller pieces. Because if the page table is too large then it is Quite difficult to search the page number.

Eg: "Two-level paging".



CHENNAI INSTITUTE OF TECHNOLOGY DEPT. OF COMPUTER SCIENCE AND ENGINEERING CHENNAI 69

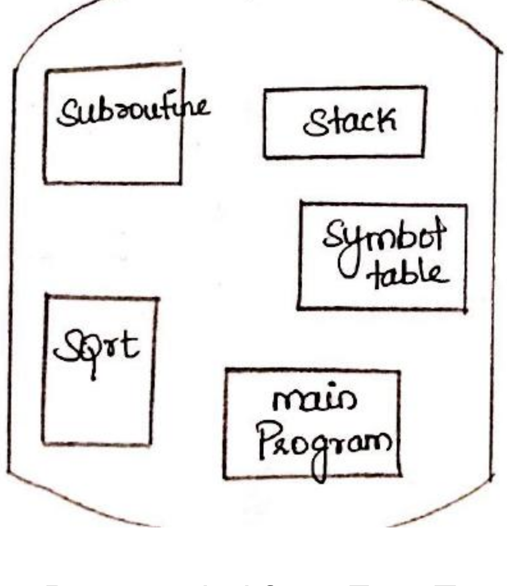
A two-level page table scheme.



(6) Segmentation :

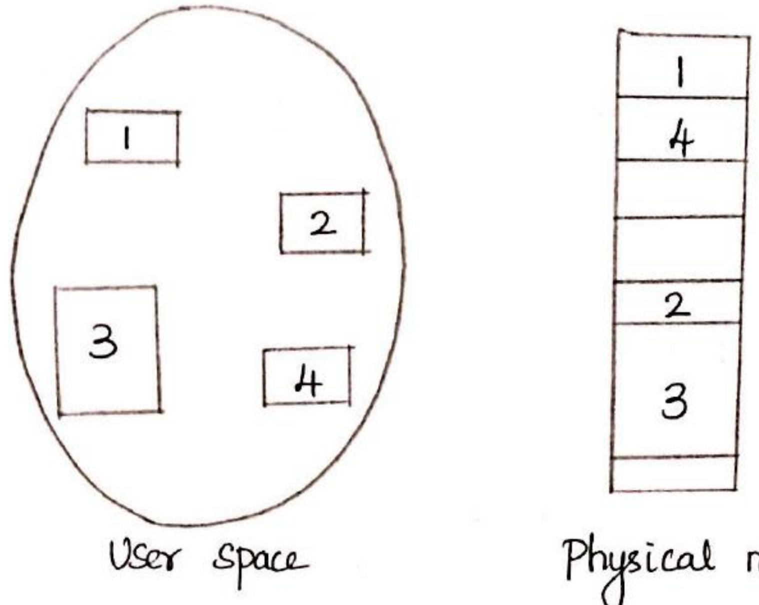
- Memory management scheme that supports user view of memory.
- A program is a collection of segments. A segment is a logical unit such as: Main program, Procedure, function, Method, object, Local variables, global variables, common block, Stack, symbol table, arrays.

User's view of a program.



logical address

Logical View of Segmentation :



Segmentation Hardware :

✓ Logical address consists of a two tuple :

<Segment - number, offset>

✓ Segment table - maps two dimensional physical addresses ; each table entry has :

✓ Base : contains the starting physical address where the segments reside in memory.

✓ Limit : specifies the length of the segment.

✓ Segment - table base register (STBR) : points to the segment table's location in memory.

✓ Segment - table length register (STLR) : Indicates number of segments used by a program ;

Segment number = S' is legal, if S < STLR.

✓ Relocation.

✓ dynamic

✓ by segment table.

✓ Sharing.

✓ shared segments

✓ Same segment number.

✓ Allocation

- ✓ first fit / best fit
- ✓ external fragmentation.

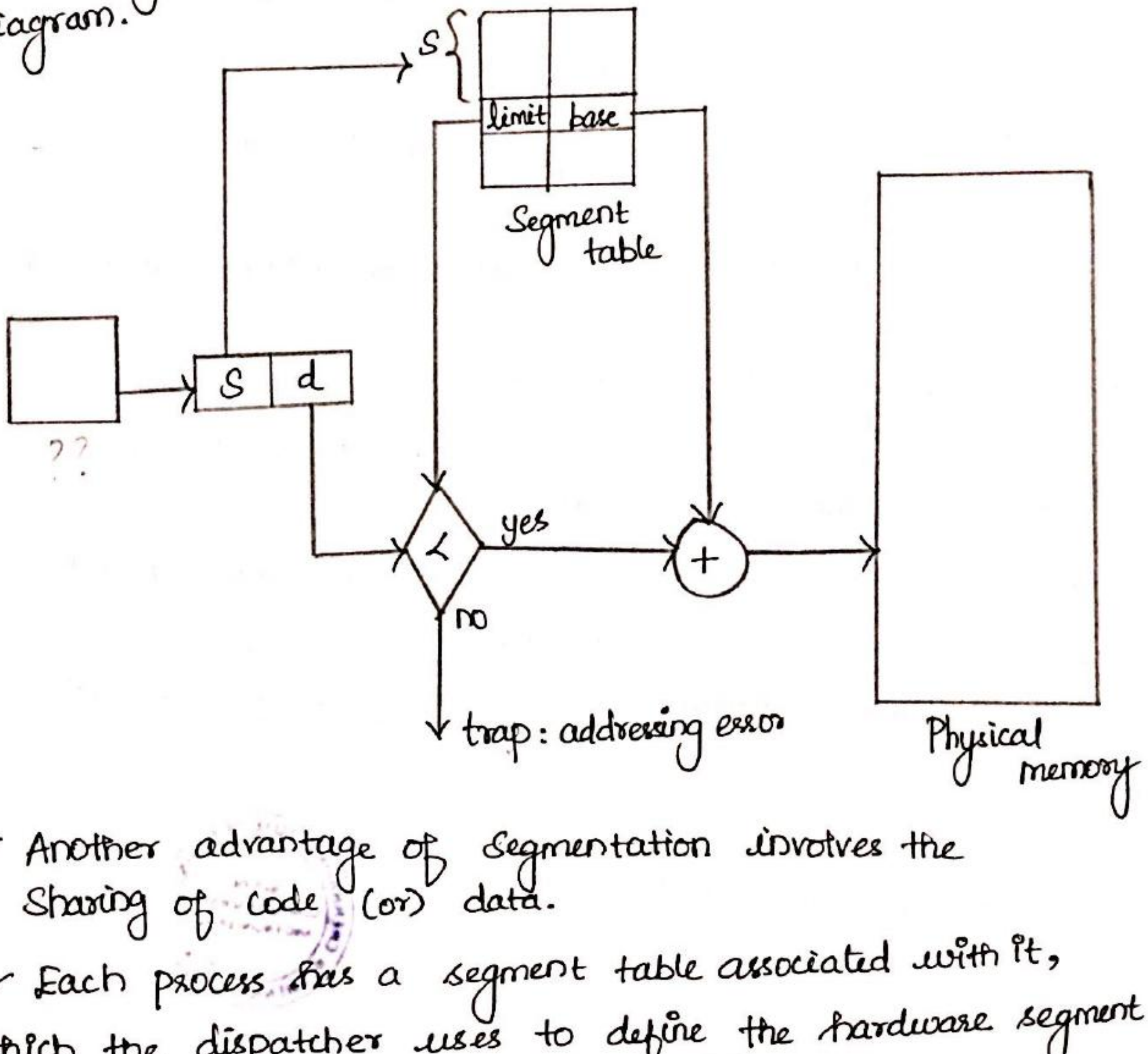
✓ Protection : With each entry in segment table associate:

- ✓ Validation bit = 0
- ✓ illegal segment
- ✓ read / write / execute Privileges.

✓ Protection bits associated with segments ; code sharing Occurs at segment level.

✓ Since segments vary in length, memory allocation is a dynamic storage - allocation problem.

✓ A segmentation example is shown in the following diagram.

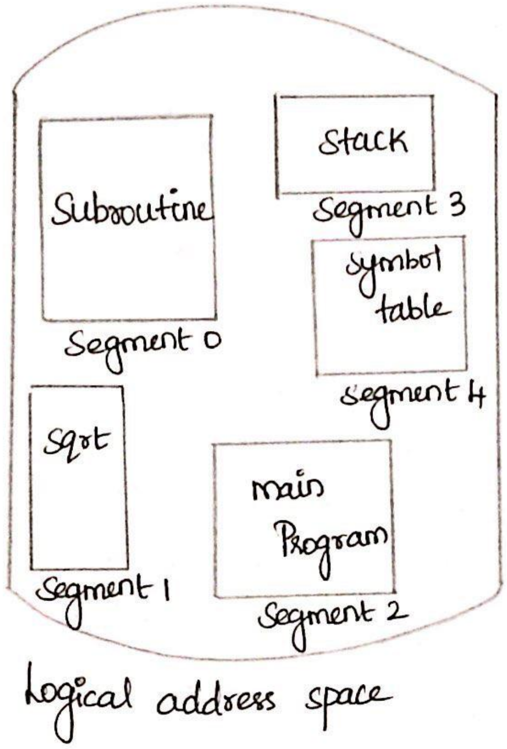


✓ Another advantage of segmentation involves the Sharing of code (or) data.

✓ Each process has a segment table associated with it, which the dispatcher uses to define the hardware segment table when this process is given the CPU.

Segments are shared when entries in the segment tables of two different processes point to the same physical location.

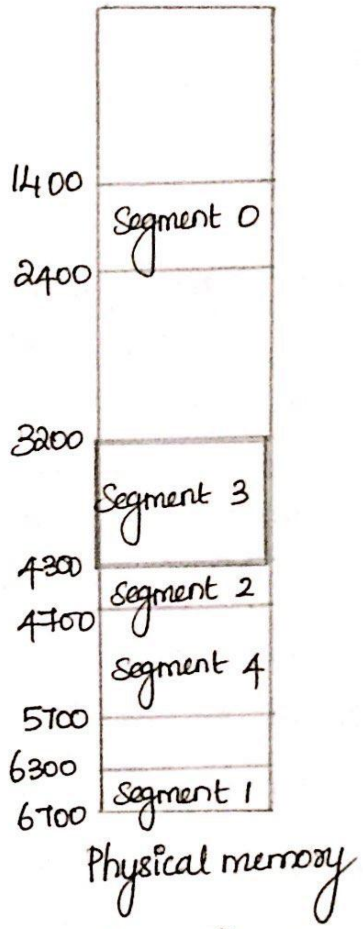
Example of Segmentation:-



Logical address space

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

Segment table



Physical memory

We have 5 segments numbered from 0 through 4. The segments are stored in physical memory. for eg., 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. A reference to segment 3, byte 852, is mapped to 3200 (the base of segment 3) + 852 = 4052.

A reference to byte 1222 of segment 0 would result in a trap to the operating system, as this segment is only 1000 bytes long.

CHENNAI INSTITUTE OF TECHNOLOGY
DEPT. OF COMPUTER SCIENCE AND ENGINEERING
CHENNAI 69

(7) Segmentation with Paging:

✓ The IBM OS/2.32 bit Version is an operating System running on top of the Intel 386 architecture. The 386 Uses Segmentation with Paging for memory management. The maximum number of segments per Process is 16 KB, and each segment can be as large as 4 gigabytes.

✓ The local address space of a process is divided into two partitions.

⇒ The first partition consists of up to 8 KB segments that are private to that process.

⇒ The Second Partition consists of up to 8 KB Segments that are shared among all the processes. Information about the first partition is kept in the local descriptor table. (LDT), information about the second Partition is kept in the global descriptor table (GDT).

Each entry in the LDT and GDT consists of 8 bytes, with detailed information about a particular segment including the base location and length of the segment. The logical address is a pair (selector, offset) where the selector is a 16-bit number:

s	g	p
13	1	2

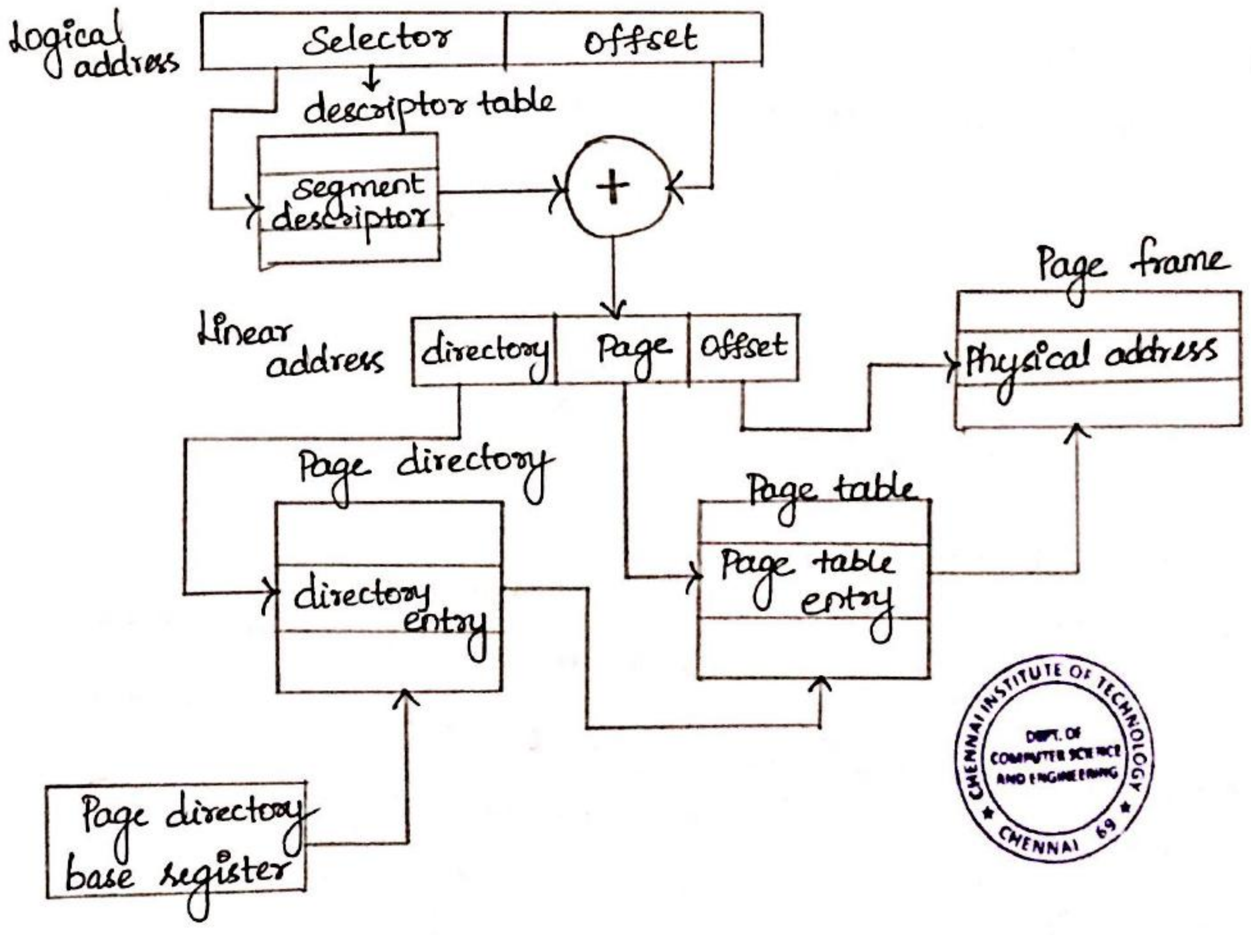
Where s designates the segment number, g indicates whether the segment is in the GDT (or) LDT, and p deals with protection. The offset is a 32-bit number specifying the location of the byte within the segment in Question.

⇒ The base and limit information about the segment in question are used to generate a linear-address.

⇒ first, the limit is used to check for address validity. If the address is not valid, a memory fault is generated, resulting in a trap to the operating system. If it is valid, then the value of the offset is added to the value of the base, resulting in a 32-bit linear address. This address is then translated into a physical address.

⇒ The linear address is divided into a page number consisting of 20 bits, and a page offset consisting of 12 bits. Since we page the page table, the page number is further divided into a page table pointer. The logical address is as follows.

P_1	P_2	d
10	10	12



CHENNAI INSTITUTE OF TECHNOLOGY
DEPT. OF COMPUTER SCIENCE AND ENGINEERING
CHENNAI 69

* To improve the efficiency of Physical memory use. Intel 386 page tables can be swapped to disk. In this case, an invalid bit is used in the page directory entry to indicate whether the table to which the entry is pointing is in memory or on disk.

* If the table is on disk, the operating system can use the other 31 bits to specify the disk location of the table; the table then can be brought into memory on demand.

(10) Virtual Memory :-

Segmentation Advantages :-

1) Segmentation eliminates fragmentation
2) It Provides virtual memory.
3) Allows dynamic segment growth.
4) Segmentation assists dynamic linking.
5) Segmentation is Visible.

Disadvantages :-

1) Maximum size of a segment is limited by the size of main memory.

2) Difficulty to manage variable size segments on Secondary storage.

Advantages of paging :

1) Paging eliminates fragmentation.
2) Support higher degree of multiprogramming.
3) Paging increases memory and processor utilization.
4) compaction overhead required for the relocatable Partition scheme is also eliminated.

Disadvantages of Paging:

1) Page address mapping hardware usually increases the cost of the computer.

2) Memory must be used to store the various tables like page table, memory map table etc.,

Advantages of segmentation with paging:

Combines all advantages of paging & segmentation.

Disadvantages:

1) It increases hardware cost.
2) It increases processor overheads.
3) Dangers of thrashing.

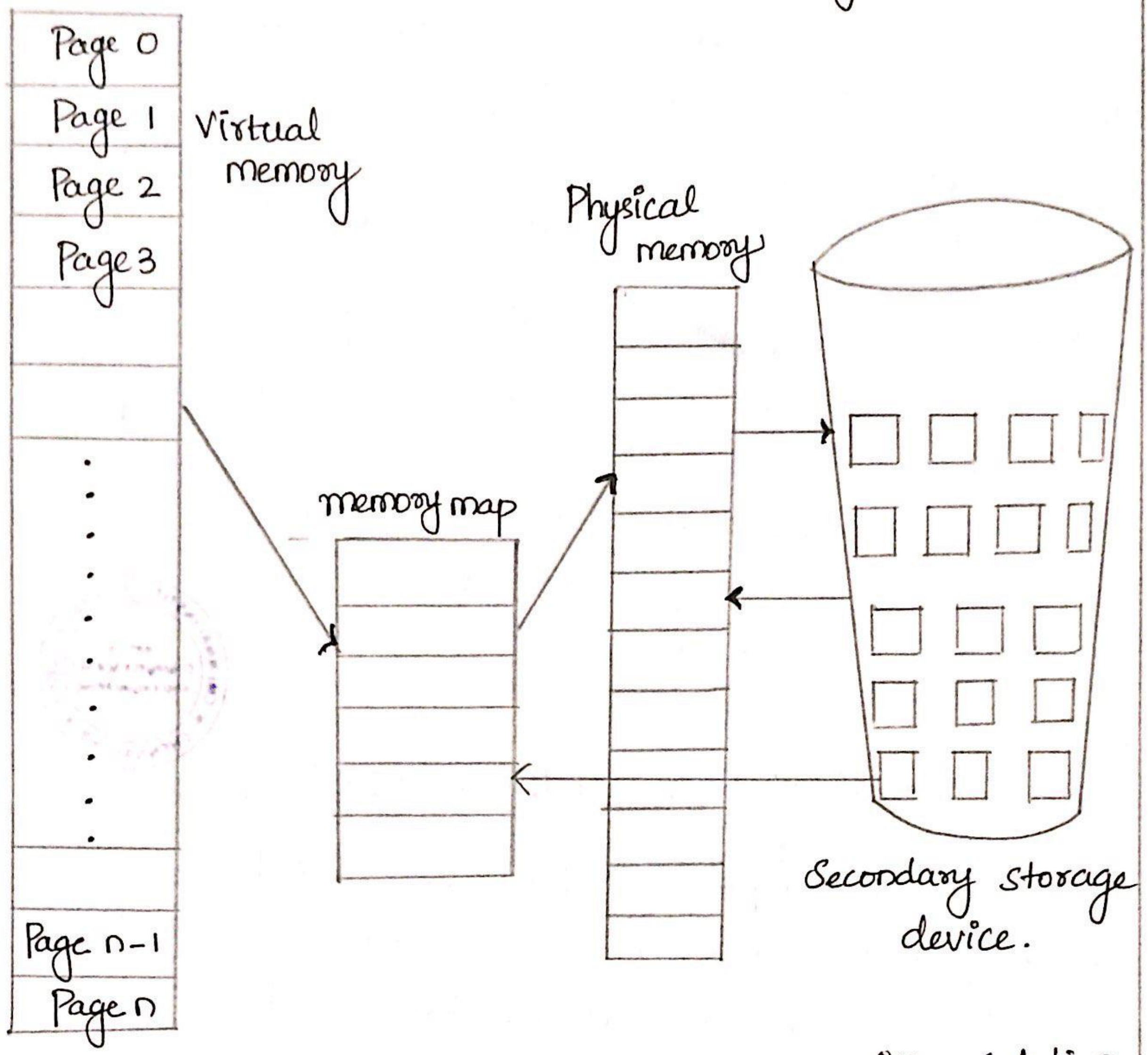
CHENNAI INSTITUTE OF TECHNOLOGY DEPT. OF COMPUTER SCIENCE AND ENGINEERING CHENNAI 69

(10) Virtual Memory :- (Background)

In many computers, Programmers often realize that some of their large Programs cannot fit in main memory for execution. Even if there is enough main memory for one Program, the main memory may be shared with other users, causing any one Program to execute. The usual solution is to introduce management schemes that intelligently allocate portions of memory to users as necessary for the efficient running of their programs. The use of virtual memory is to achieve this goal.

* Virtual memory allows execution of partially loaded Process. The ability to execute a partially loaded Process is also advantageous from the operating system point of view.

Virtual memory with other memory :



The fig., shows the virtual memory with relation to physical memory.

In fact, an examination of real programs shows us that, in many cases, the entire program is not needed.

Following are the situations, when entire program is not required to load fully.

1) User written error handling routines are used only when an error occurs in the data (or) computation.

2) Certain options and features of a program may be used rarely.

3) Many tables are assigned a fixed amount of address space even though only a small amount of the table is actually used.

4) Many routines are commonly used at mutually exclusive times during a run.

The ability to execute a program that is only Partially in memory would confer many benefits.

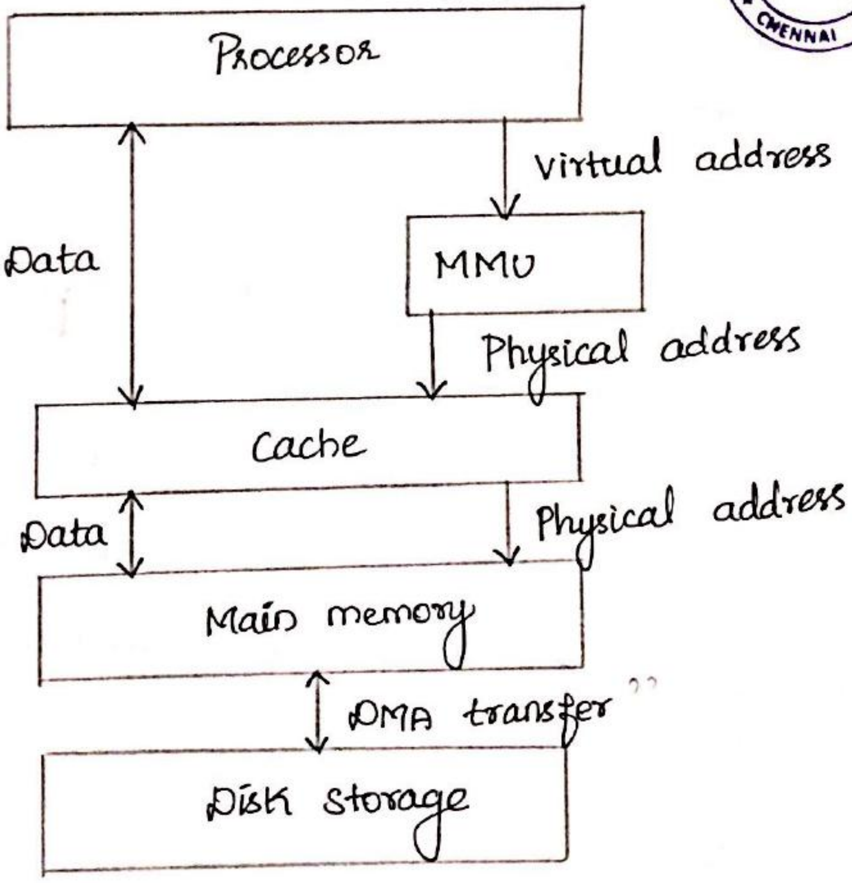
1) Less number of I/O would be needed to load (or) swap each user program into memory.

2) A program would no longer be constrained by the amount of physical memory that is available.

3) 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.

Virtual memory organization

CHENNAI INSTITUTE OF TECHNOLOGY DEPT OF COMPUTER SCIENCE AND ENGINEERING CHENNAI 69



Virtual memory makes the task of Programming much easier. Virtual memory is commonly implemented by demand paging.

Virtual memory mechanism bridges the size and speed gaps between the main memory and secondary Storage and is usually implemented in part by software technique.

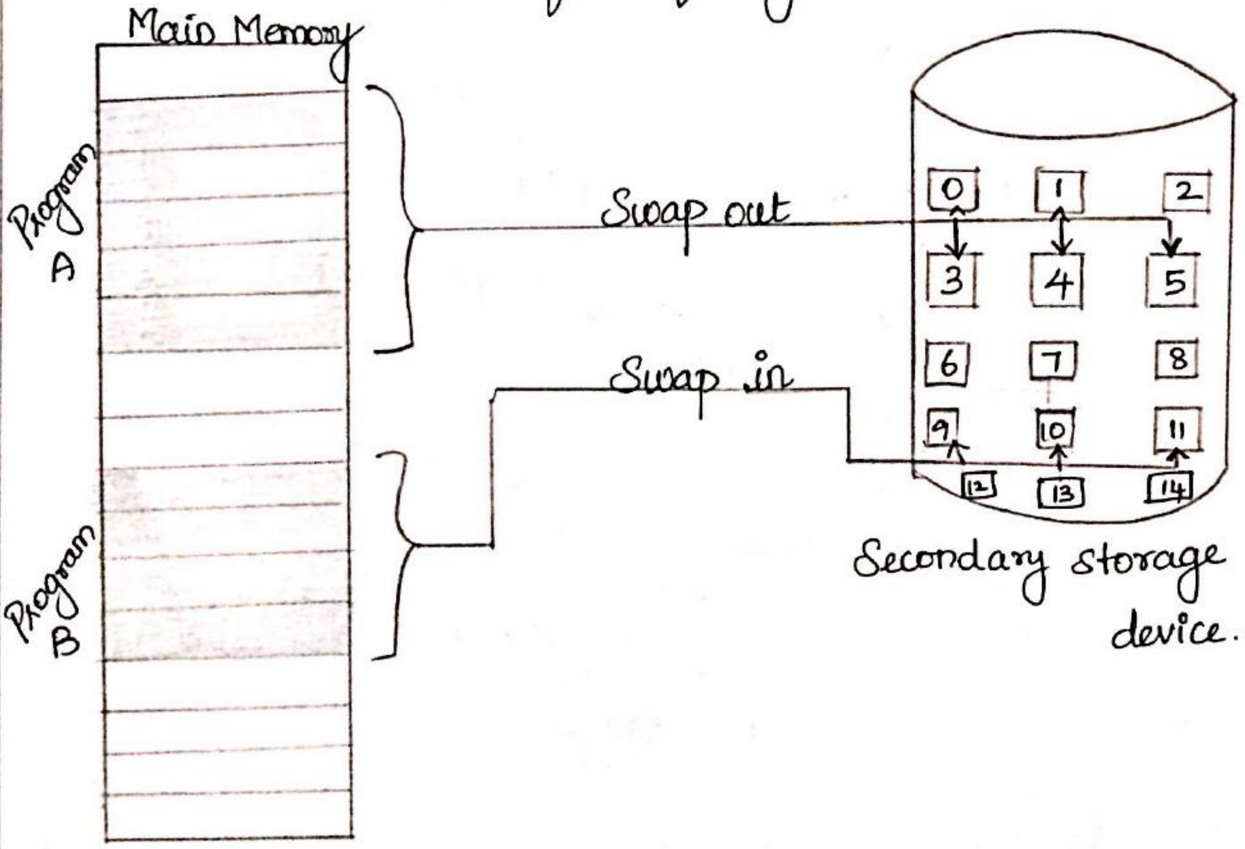
(12) Demand Paging:

Demand paging = Paging system + swapping.

* With demand paging, a page is brought into main memory only when a reference is made to a location.

* Lazy swapper concept is used in used in demand paging. A Lazy swapper never swaps a page into a memory unless that page will be needed.

Transfer of page.



* Demand paging combines the features of simple Paging and overlaying to implement virtual memory.

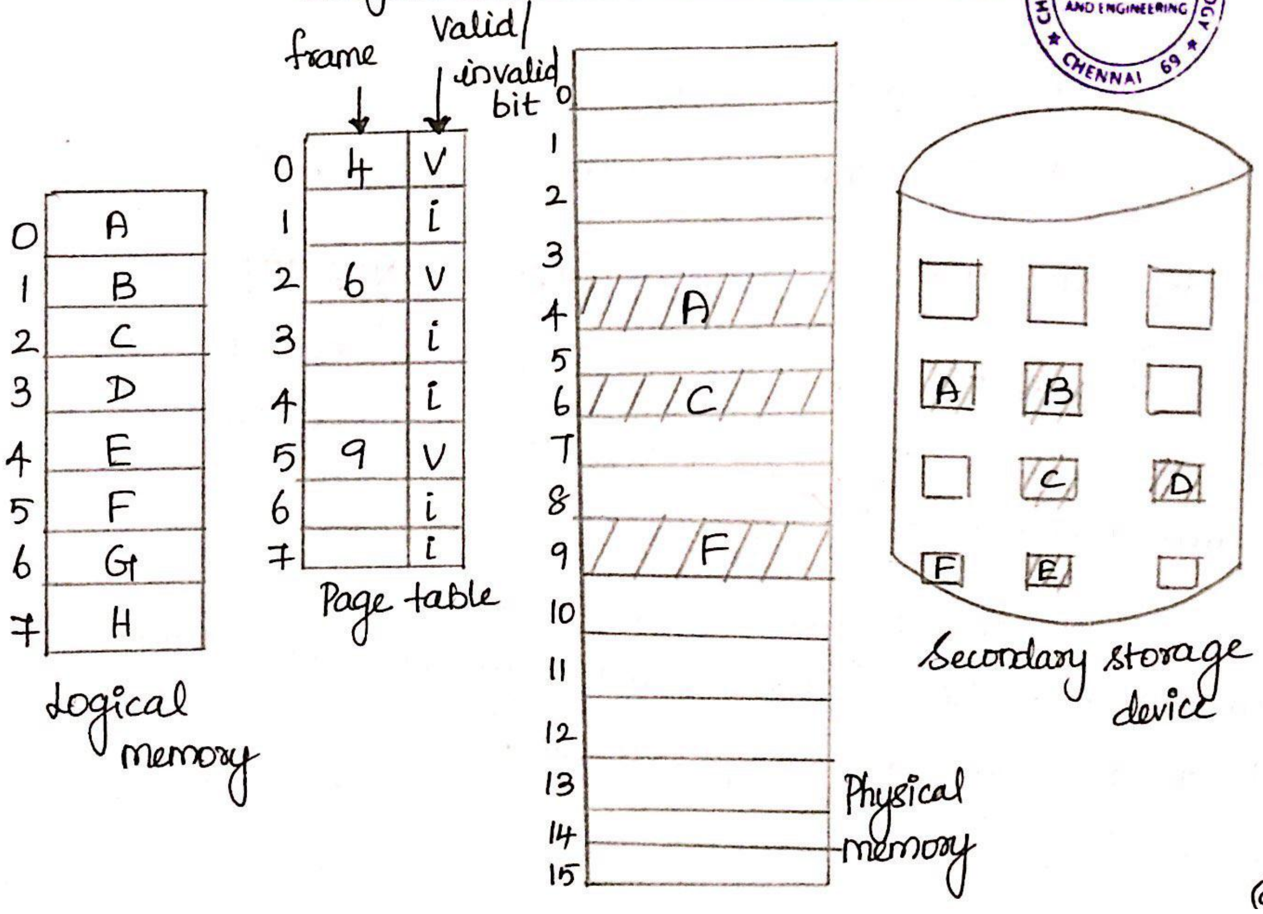
* In this, each page of a program is stored contiguously in the paging swap space on a secondary Storage.

* As locations in pages are referenced, the pages are copied into memory page frames.

* Once the page is in the memory, it is accessed as in simple Paging. The previous diagram Shows the transfer of page from secondary storage to main memory.

* Some form of hardware support is required to distinguish between those pages that are in memory and those pages that are on the disk. The valid/invalid bit scheme can be used for this purpose. Each entry in the page table has at a minimum two fields. Page frame and Valid - invalid bit.

Page table with Valid invalid bits



* When a valid bit is set, then the associated page is legal and present in the memory. Whenever a virtual address is generated, the memory management hardware extracts the page number from the address, and the appropriate entry in the page table is accessed.

The valid-invalid bit is checked. If the page is not in memory, a page fault occurs, transferring control to the page fault routine in the operating system.

* When the running process experiences a page fault, it must be suspended until the missing page is brought into main memory.

* The disk address of the faulted page is usually provided in the File Map Table (FMT). This table is parallel to the page map table. Thus, when processing a page number provided by the mapping hardware of index the FMT and to obtain the related disk address.

* Steps in Handling a page fault
* Hardware support
* Software Algorithm
* Performance of demand paging.

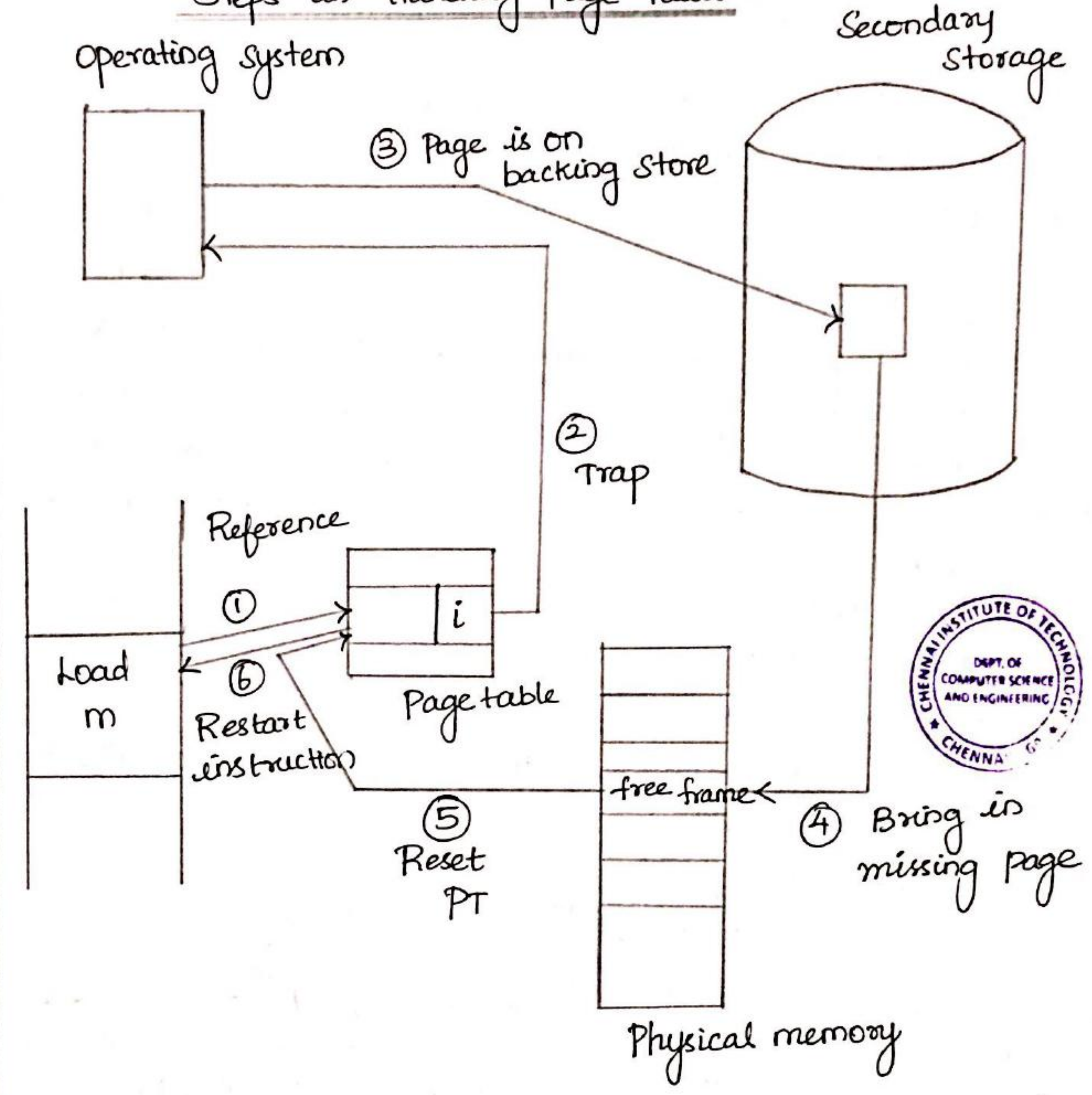
* Steps in Handling a page fault : [N|D-19]

1) Check an internal table for this process, to determine whether the reference was a valid or invalid memory access.

2) If the reference was invalid, terminate the process. If it was valid, but not yet brought in that page, now page it in.

* find a free frame.
* Schedule a disk operation to read the desired page into the newly allocated frame.

Steps in Handling page fault :



CHENNAI INSTITUTE OF TECHNOLOGY
DEPT. OF COMPUTER SCIENCE AND ENGINEERING
CHENNAI

* Schedule a disk operation to read the desired Page into the newly allocated frame.
* When disk read is complete, modify the internal table kept with the Process and the page table to indicate that the page is now in memory. Restart the instruction that was interrupted by the illegal address trap. The Process can now access the page as though it had always been in memory.

Hardware support

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

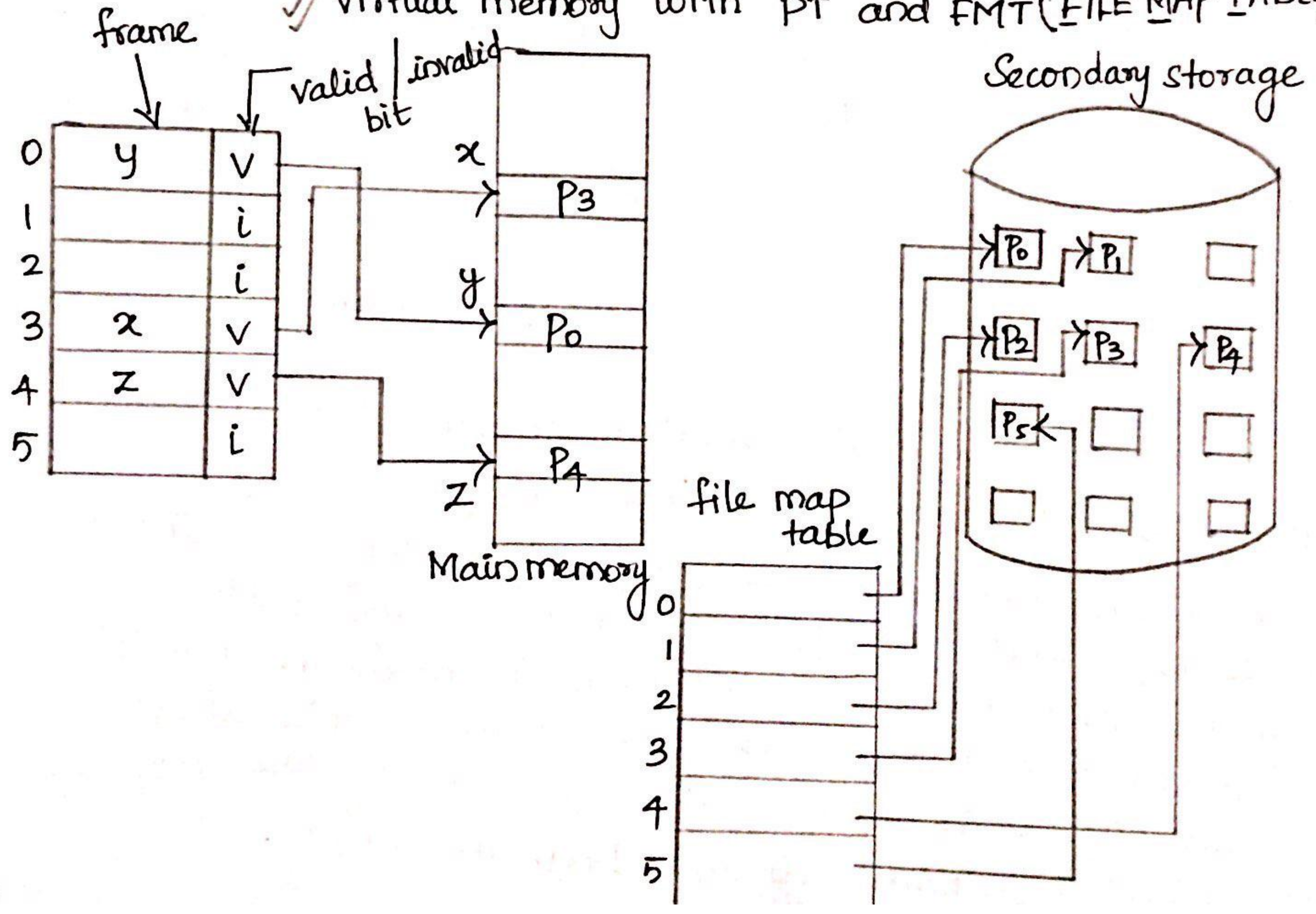
Page table : It has the ability to mark an entry invalid through a valid - invalid bit.

Secondary memory : This memory holds those pages that are not present in main memory. It is usually high speed disk.

Software Algorithm :

Demand page memory management provides tremendous flexibility for the operating system. It must interact with information management to access and store copies of the jobs address space on secondary storage. file map table is used to store the information regarding a file. FMT is not used by the hardware. It is usually accessed by software. A possible format and use of the file map table is shown in fig.,

✓ Virtual memory with PT and FMT (FILE MAP TABLE)



Performance of demand Paging :-

Demand paging can have a significant effect on the performance of a computer system. Let us calculate effective access time for a demand paged memory. Let P be the Probability of a page fault $(0 \leq P \leq 1)$.

Effective access time $= (1-P) \times ma + P \times$ Page fault time

where., $P \rightarrow$ page fault ; ma = Memory access time.

Effective access time is directly proportional to the Page fault rate.

It is important to keep the page-fault rate low in a demand paging system. Otherwise, the effective access time increases, slowing process execution dramatically.

Advantages of demand Paging :

1) Large virtual memory.

2) More efficient use of memory.

3) Unconstrained multiprogramming. There is no limit on degree of multiprogramming.

Disadvantages of demand paging :

1) Number of tables and amount of Processor over head for handling page interrupts are greater than in the case of the simple paged management techniques.

2) Due to the lack of an explicit constraints on a job, address space size.

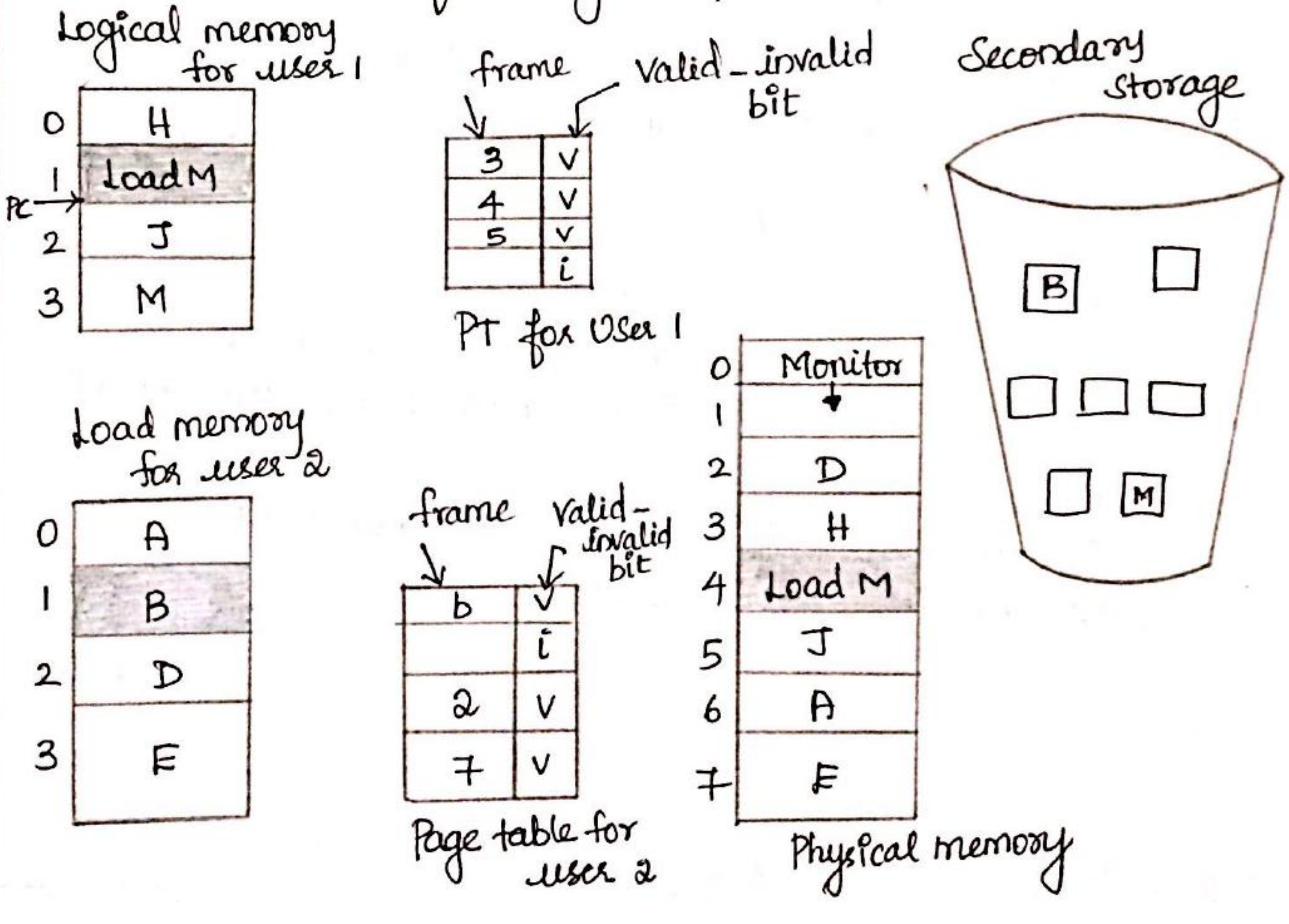
(13) Page Replacement Algorithms :- [N|D-19]

* Page replacement policy deals with the selection of a page in memory to be replaced when a new page must be brought in. While a user process is executing, a page fault occurs.

* The hardware traps to the operating system, which checks its internal tables to see that this page fault is a genuine one rather than an illegal memory access.

* The operating system determines where the desired Page is residing on the disk, but then finds that there are no free frames, on the free frame list. All memory is in use. When all the frames in main memory are occupied and It is necessary to bring in a new page to satisfy a page fault, replacement Policy is concerned with selecting a page currently in memory to be replaced.

Need for Page replacement.



* All the Policies have as their objective that the Page that is removed should be the page least likely to be referenced in the near future.

* Working of Page replacement Algorithm :-

1) Find the location of the desired page on the disk.

2) Find a free frame

a) If there is a free frame, use it.

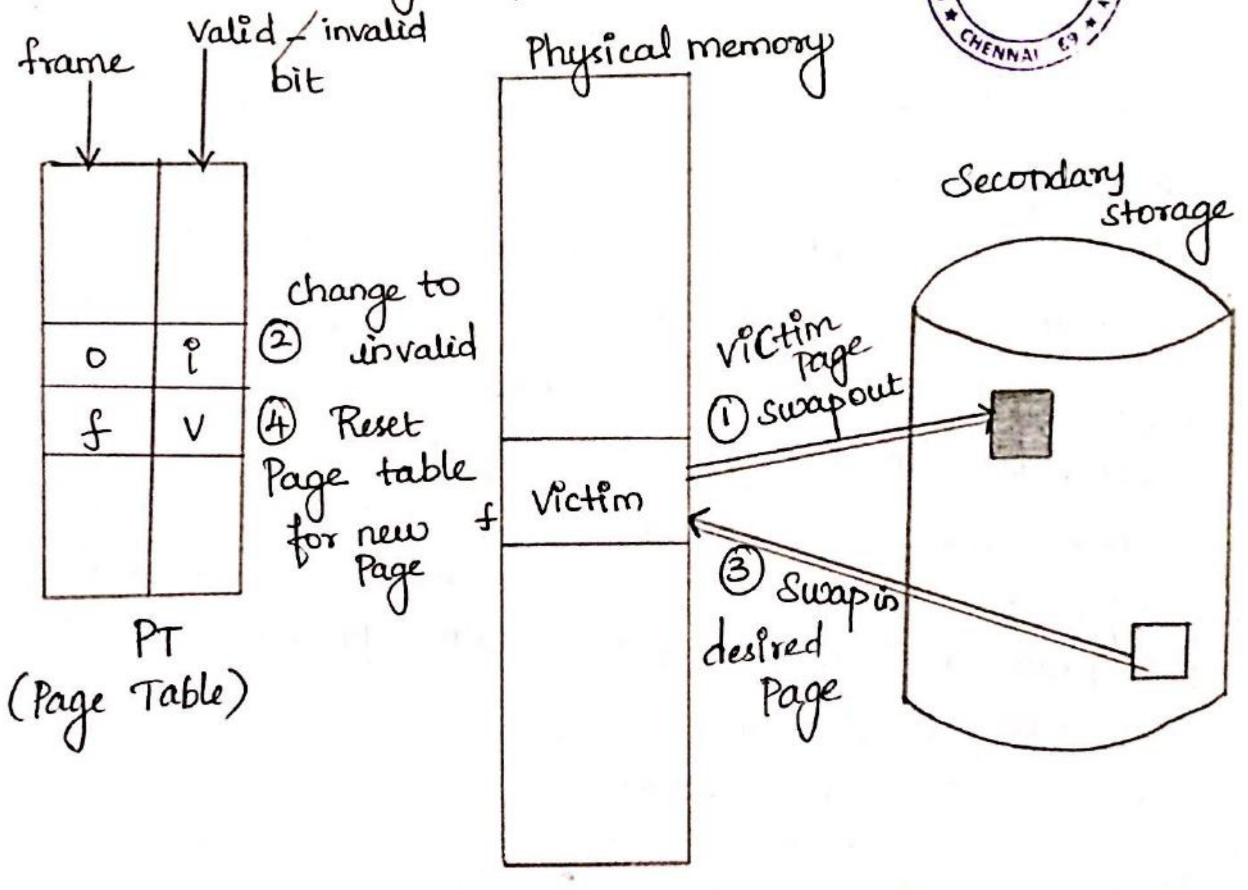
b) If there is no free frame, use a page replacement algorithm to select a victim frame.

c) Write a Victim page to the disk, change the Page and frame tables accordingly.

3) Read the desired Page into the free frame, Change the page and frame tables.

4) Restart the user process.

Page replacement.



Memory Reference String :

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

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

When analyzing page replacement algorithms, we are interested only in the pages being referenced.

Replacement Algorithms.

1) First In First Out (FIFO)   2) Least Recently Used (LRU)   3) optimal

1) First In First out (FIFO) Page Replacement :

⇒ It is one of the simplest method. It selects the page that has been in memory the longest. When a page must be replaced, the oldest page is chosen.

⇒ To implement, the memory manager must keep track of the relative order of the loading of pages into the memory.

⇒ When a page is brought into memory, it is inserted at the tail of the Queue.

⇒ It is easy to understand and program. FIFO is not the 1st choice of the operating system designess for page replacement algorithm, since its performance is not always good.

Eg., Let us consider the reference string.

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

Page frame : 3.

Solution :

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

∴ No. of Page fault is : 16.

If Page frame : 4

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

∴ No. of Page fault is : 8

Belady's Anomaly :- As the page fault rate may increase as the number of allocated frames increases. FIFO page replacement algorithm may space this problem. (for some page replacement algorithm)

2) LRU Pagereplacement :-

⇒ It replaces the page in memory that has not been referenced for the longest time.

⇒ It performs better than FIFO.

⇒ It do not suffer from Belady's Anomaly.

Let us apply LRU algorithm to the reference string with frame 3.

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

Solution : ←

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

∴ No. of page fault : 16

If Page frame = 4, then ←

frame	0	1	2	3	0	1	2	3	0	1	2	3	4	5	6	7
0	0	0	0	0	0	0	0	0	0	0	0	0	4	4	4	4
1		1	1	1	1	1	1	1	1	1	1	1	1	5	5	5
2			2	2	2	2	2	2	2	2	2	2	2	2	6	6
3				3	3	3	3	3	3	3	3	3	3	3	3	7
Page fault	*	*	*	*	[illegible]	Hit							*	*	*	*

∴ No. of page fault : 8

Implement using
- counters
- Stack.

Counters :

1) Each time a page is referenced, copy the counter into the time-of-use field.

2) When a page needs to be replaced, replace the Page with the smallest counter value.

Stack: 1) Whenever a page is referenced, remove the Page from the stack and put it on top of the stack.

2) When a page need to be replaced, replace the Page that is at the bottom of the stack (LRU page)

LRU Approximation :

* A reference bit is associated with each memory block and this bit is automatically set to 1 by the hardware whenever that page is referenced. The single reference bit per clock can be used to approximate LRU removal.

* The page removal software periodically resets the reference bit to 0, while the execution of the users job causes some reference bits to be set to 1. If the reference bit is 0, then that page has not been referenced since the last time the reference bit was reset to 0.

3) optimal Page Replacement :

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

Eg., Ref. frame : 3

Ref string : 0 1 2 3 0 1 2 3 0 1 2 3 4 5 6 7
$\longrightarrow$

Frame	0	1	2	3	0	1	2	3	0	1	2	3	4	5	6	7
0	0	0	0	0	0	0	0	0	0	1	1	1	4	4	4	7
1		1	1	1	1	1	2	2	2	2	2	2	2	5	5	5
2			$\textcircled{2}$	3	3	3	3	3	3	3	3	3	3	3	6	6
Page fault	*	*	*	*			*			*			*	*	*	*

∴ Number of Page fault : 10.

With Page Frame = 4, for some reference string.

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

No. of Page fault : 8

4) Counting based Page replacement :-

⇒ It depends on the counter of the no. of references.

(LFU) Least Frequently Used.

⇒ It Selects a page for replacement if the Page has not been used often in the past. LFU tends to react slowly to change in locality. LFU uses frequency counts from the beginning of the Page reference stream.

Eg; Ref. String :- 0, 1, 2, 3, 0, 1, 2, 3, 0, 1, 2, 3, 4, 5, 6, 7.
Ref. frame :- 3.

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

∴ No. of page fault = 12.

14) Allocation of Frames :

The allocation policy in a virtual memory system governs the operating system decisions regarding the amount of real memory to be allocated to each active Process.

In Paging System,

⇒ If more real pages are allocated to a process, it reduces page fault frequency and improved turnaround time.

⇒ If too few pages are allocated to a process, its page fault frequency and turnaround time may decreases to unacceptable levels.

The minimum no. of frames per process is defined by the architecture, the maximum number is defined by the amount of available physical memory.

$$\begin{tikzcd}
& \text{Allocation Schemes} \arrow[dl] \arrow[dr] & \\
\text{Equal Allocation} & & \text{Proportional Allocation}
\end{tikzcd}$$

Equal Allocation :

If there are 'n' processes and 'm' frames, then allocate m/n frames to each process.

Eg ; If there are '5' processes and '100' frames, give each process '20' frames.

Proportional Allocation : Allocate according to size of process.

Let S_i be the size of Process i.

Let m be the total no. of frames.

then ., $S = \Sigma S_i$

$a_i = S_i / S * m$

where a_i is the no. of frames allocated to process i

With multiple Processes competing for frames,

$$\begin{tikzcd}[column sep=small] & \text{Page replacement} \arrow[dl, dash] \arrow[dr, dash] & \\ \text{Local Replacement} & & \text{Global Replacement.} \end{tikzcd}$$

~~Local~~ Global Replacement : Each Process selects a replacement frame from the set of all frames; One Process can take a frame from another.

Local Replacement : Each process selects from only its own set of allocated frames.

(15) Thrashing :

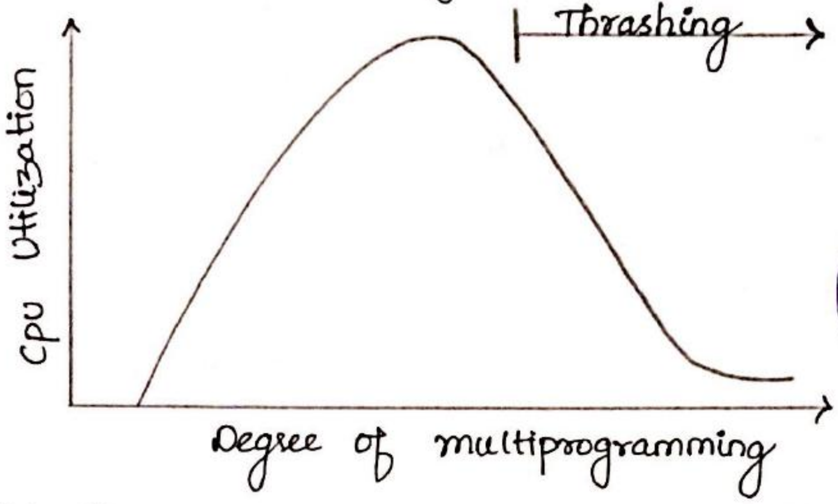
High paging activity is called thrashing. If the number of frames allocated to a low priority process falls below the minimum number required by the computer architecture, Process must be suspend for execution.

The phenomenon of excessively moving pages back and forth between memory and secondary storage is called thrashing.

It consumes lot of computer energy but accomplishes very little useful results. A process is thrashing if it is spending more time paging than executing. Thrashing results in several performance problems.

The operating system monitors cpu utilization. If cpu utilization is too low, we increase the degree of multiprogramming by introducing new Process to the system. As processes wait for the Paging device, cpu utilization decreases. The cpu scheduler sees the decreasing cpu utilization and increases the degree of multiprogramming as a result.

Thrashing.



At this point, to increase CPU utilization and stop thrashing, we must decrease the degree of multiprogramming.

We can limit the effect of thrashing by using a local replacement algorithm. To prevent thrashing, we must provide a process as many frames as it needs. But how to calculate the required frame. Solution to this is by using working set theory.

Locality of Reference :

The locality model states that, as a process executes, it moves from locality to locality.
A locality is a set of pages that are actively used together.

$$\begin{tikzcd} & \text{Locality} \arrow[dl, dash] \arrow[dr, dash] & \\ \text{Spatial locality} & & \text{Temporal locality.} \end{tikzcd}$$

Spatial locality : It refers to the fact that if a memory location is accessed, it is likely that a location near it will be accessed in the next instruction.
Temporal locality : If a memory location has been referenced, there is good chance it will be referenced again in a short period of time.

To limit thrashing, we can use a local replacement algorithm. To prevent thrashing, there are two methods namely.,

* Working set strategy
* Page fault frequency.

* Working set Strategy:

* It is based on the assumption of the model of locality.
* Locality is defined as the set of pages actively used together.
* Working set is the set of pages in the most recent, Δ page references.
* Δ is the working set window.

Δ if Δ too small, it will not encompass entire locality.

Δ if Δ too large, it will encompass several localities.

Δ if Δ = ΔΔ it will encompass entire program.

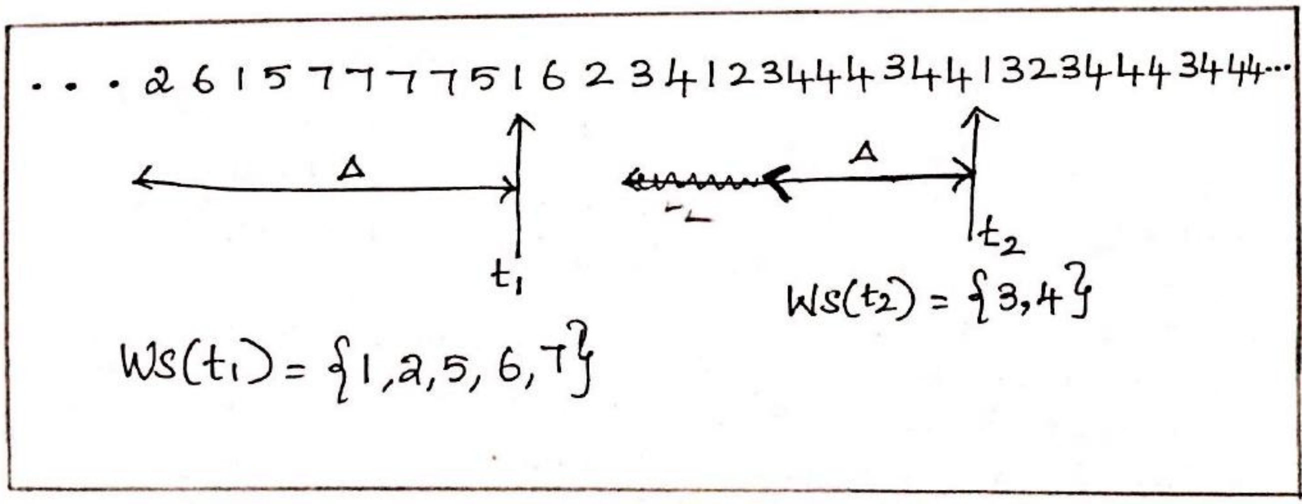
$D = \Delta WSS_i$.

Δ where WSS_i is the working set size for Process i.

Δ D is the total demand of frames.

* if $D > m$ then, Thrashing will occur.

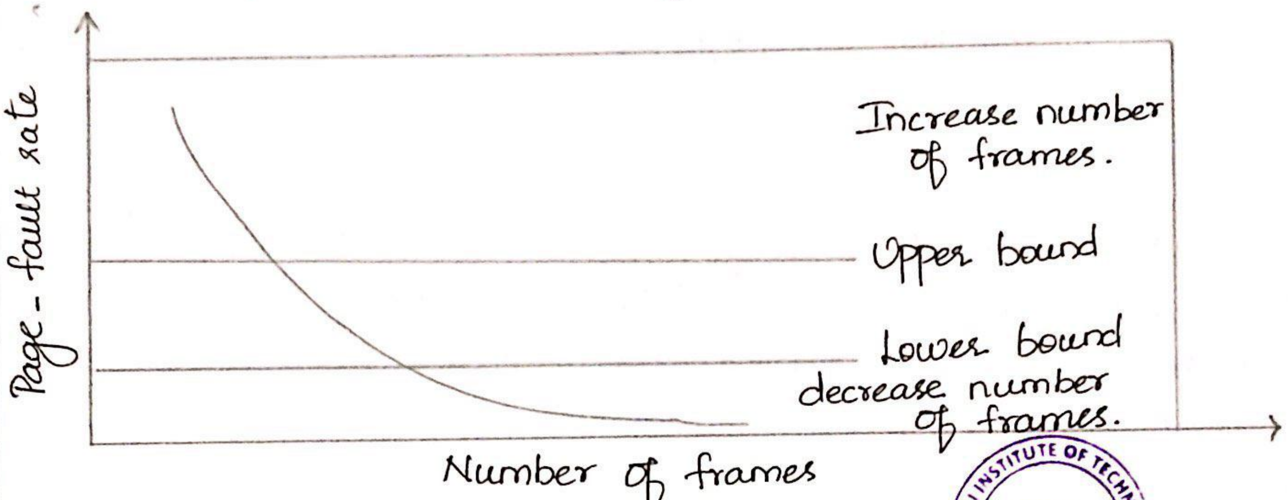
Page reference table :- [Working set model]



* Page - fault frequency Scheme :

→ If actual rate too low, process loses frame.

→ If actual rate too high, Process gains frame.



(16) Allocating Kernel Memory :

When a process running in user mode requests additional memory, pages are allocated from the list of free Page frames maintained by the kernel.

This list is typically populated using a page replacement algorithm such as those most likely contains free pages scattered throughout physical memory. ~~Internal fragmentation will~~ If a user process requests a single byte of memory, internal fragmentation will result, as the process will be granted an entire Page frame.

Kernel memory is often allocated from a free-memory pool different from the list used to satisfy ordinary user-mode processes. There are two primary reasons for this :

(1) The Kernel requests memory for data structures of varying sizes, some of which are less than a page in size.

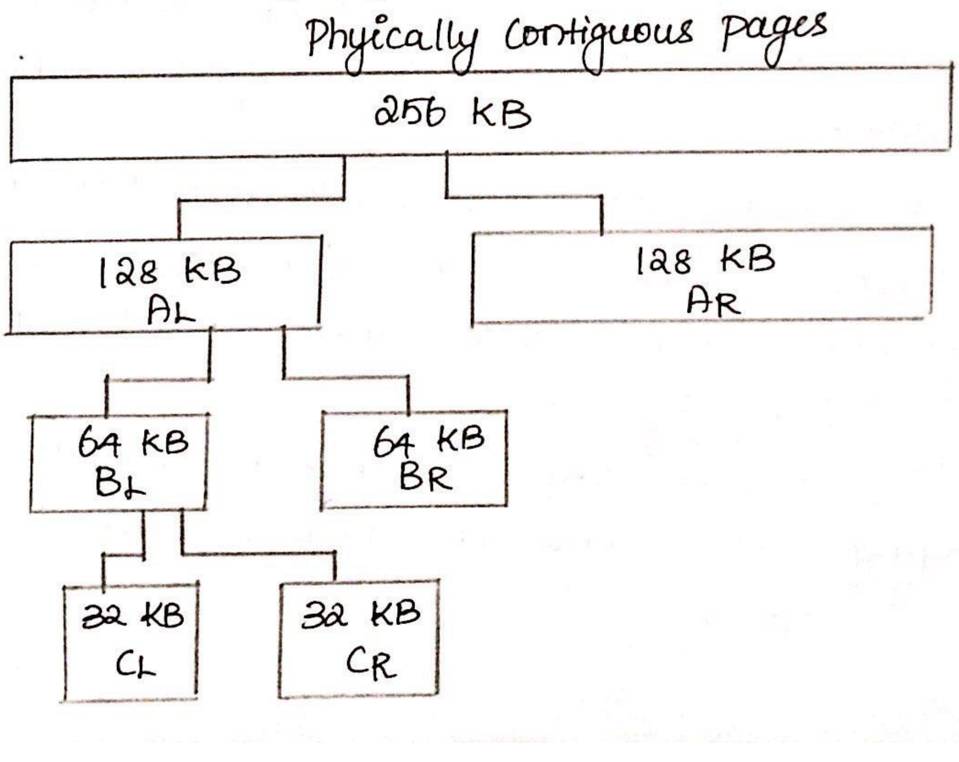
As a result, the kernel must use memory conservatively and attempt to minimize waste due to fragmentation. This is especially important because many operating systems do not subject kernel code or data to the paging system.

(2) page allocated to user-mode processes do not necessarily have to be in contiguous physical memory. However certain hardware devices interact directly with physical memory - without the benefit of a virtual memory interface - and consequently may require memory residing in physically contiguous pages.

Buddy system:

The buddy system allocates memory from a fixed - size consisting of physically contiguous pages. Memory is allocated from this segment using a power- of- 2 allocator, which satisfies requests in units sized as a power of 2 (4 KB, 8KB, 16KB, and so forth). A request in units not appropriately sized is rounded up to the next highest power of 2. for eg., a request for 11 KB is satisfied with a 16K segment.

Buddy system allocation.



(17) OS Examples :-

Windows : Windows implements virtual memory using demand paging with clustering. clustering handles Page faults by bringing in not only the faulting Page but also several pages following the faulting Page. When a process is first created, it is assigned a working set minimum and maximum. The working-set minimum is the minimum number of Pages the process is guaranteed to have in memory.

If sufficient memory is available, process may be assigned as many pages as its working-set maximum. The virtual memory manager maintains a list of free page frames. Associated with this list is a threshold value that is used to indicate whether sufficient free memory is available.

If a page fault occurs for a process that is below its working-set maximum, the virtual memory manager allocates a page from this list of free pages. If a process that is at its working set maximum incurs a page fault, it must select a page for replacement using local LRU page-replacement policy.

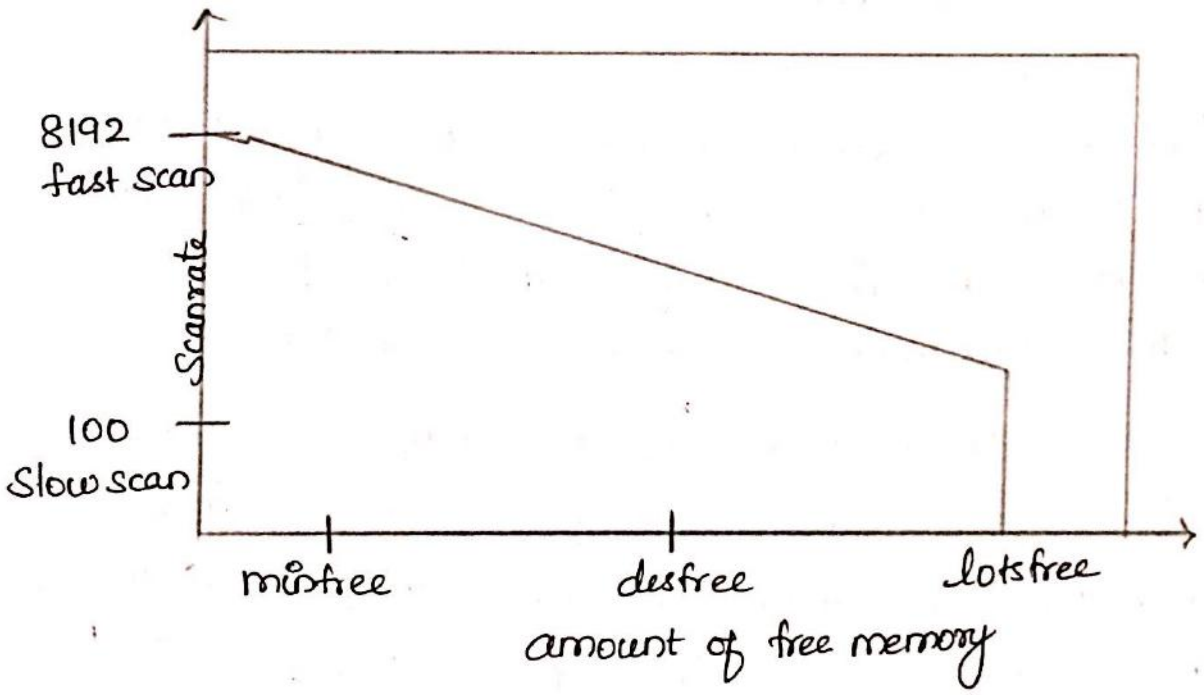
Solaris :-

In solaris, when a thread incurs a page fault, the kernel assigns a page to the faulting thread from the list of free pages it maintains.

Therefore, it is imperative that the kernel keep a sufficient amount of free memory available.

Associated with this list of free pages is a Parameter - lotsfree - that represents a threshold to begin Paging. The lotsfree parameter is typically set to 1/64 the size of the physical memory. Four times per second, the kernel checks whether the amount of free memory is less than lotsfree. If the number of free pages falls below lotsfree, a process known as a pageout starts up.

Solaris page scanner:-



K. [illegible] 24/12/18
Prepared By

[illegible] 10/01/19
Verified By

[illegible] 22/1/19
Approved By

CHENNAI INSTITUTE OF TECHNOLOGY
DEPT. OF COMPUTER SCIENCE AND ENGINEERING
CHENNAI 69

UNIT-IV

FILE SYSTEMS AND I/O SYSTEMS

Mass Storage system – Overview of Mass Storage Structure, Disk Structure, Disk Scheduling and Management, swap space management; File-System Interface – File concept, Access methods, Directory Structure, Directory organization, File system mounting, File Sharing and Protection; File System Implementation- File System Structure, Directory implementation, Allocation Methods, Free Space Management, Efficiency and Performance, Recovery; I/O Systems – I/O Hardware, Application I/O interface, Kernel I/O subsystem, Streams, Performance.

Mass Storage system: Overview of Mass Storage Structure

Mass Storage system:

Mass Storage refers to systems meant to store large amounts of data. Mass storage system is where the operating system is stored, where all our PC programs are kept and where we keep the stuff we create and collect.

Magnetic Disks

Magnetic disk provides bulk of secondary storage for modern computer systems.

Traditional magnetic disks have the following basic structure:

One or more ***platters*** in the form of disks covered with magnetic media. ***Hard disk*** platters are made of rigid metal, while "***floppy***" disks are made of more flexible plastic.Common platter diameters range from **1.8 to 5.25 inches.**

The two surfaces of a platter are covered with a magnetic material. We store information by recording it magnetically on the platters.

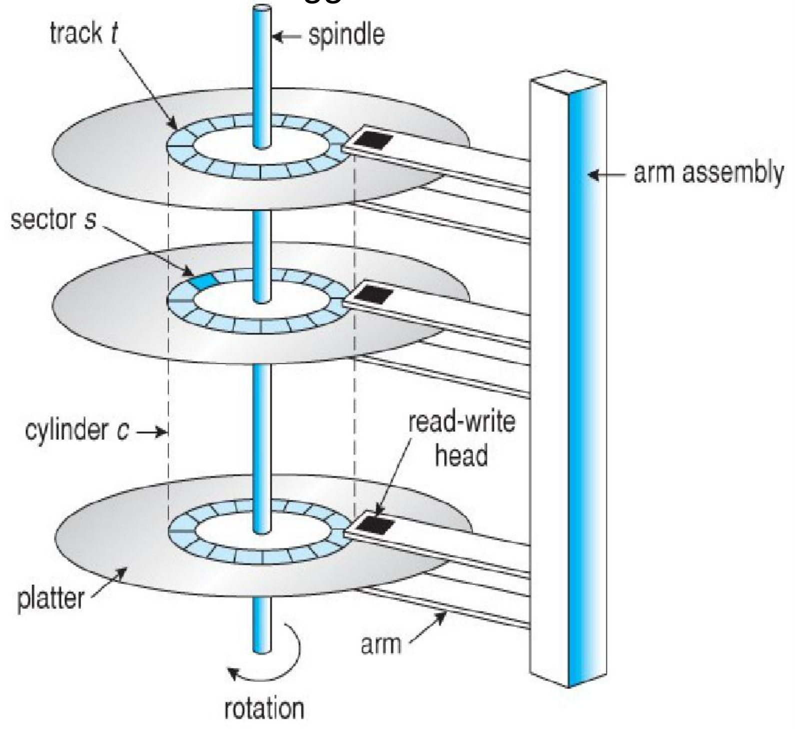
Each platter has two working ***surfaces.*** Older hard disk drives would sometimes not use the very top or bottom surface of a stack of platters, as these surfaces were more susceptible to potential damage.

Each working surface is divided into a number of concentric rings called ***tracks.*** The collection of all tracks that are the same distance from the edge of the platter, (i.e. all tracks immediately above one another in the following diagram) is called a ***cylinder***.

Each track is further divided into ***sectors,*** traditionally containing **512 bytes** of data each, although some modern disks occasionally use larger sector sizes. The data on a hard drive is read by read-write ***heads.*** The standard configuration (shown below) uses one head per surface, each on a separate ***arm***, and controlled by a common ***arm assembly*** which moves all heads simultaneously from one cylinder to another.

The storage capacity of a traditional disk drive is equal to the number of heads(i.e. the number of working surfaces), times the number of tracks per surface, times the number of sectors per track, times the number of bytes per sector.

In operation the disk rotates at high speed, such as 7200 rpm (120 revolutions per second.) The rate at which data can be transferred from the disk to the computer is composed of several steps:



Moving-head disk mechanism.

The ***positioning time***, the ***seek time*** or ***random access time*** is the time required to move the heads from one cylinder to another, and for the heads to settle down after the move. This is typically the slowest step in the process and the predominant bottleneck to overall transfer rates.

The ***rotational latency*** is the amount of time required for the desired sector to rotate around and come under the read-write head. This can range anywhere from zero to one full revolution, and on the average will equal one-half revolution. The ***transfer rate***, which is the time required to move the data electronically from the disk to the computer. Disk heads "fly" over the surface on a very thin cushion of air. If they should accidentally contact the disk, then a ***head crash*** occurs, which may or may not permanently damage the disk or even destroy it completely.

Floppy disks are normally ***removable***.Hard drives can also be removable, and some are even ***hot-swappable***, meaning they can be removed while the computer is running, and a new hard drive inserted in their place.

Disk drives are connected to the computer via a cable known as the ***I/O Bus.*** Some of the common interface formats include Enhanced Integrated Drive Electronics, EIDE; Advanced Technology Attachment, ATA; Serial ATA, SATA, Universal Serial Bus, USB; Fiber Channel, FC, and Small Computer Systems Interface, SCSI. The ***host controller*** is at the computer end of the I/O bus, and the ***disk controller*** is built into the disk itself.

The CPU issues commands to the host controller via I/O ports. Data is transferred between the magnetic surface and onboard ***cache*** by the disk controller, and then the data is transferred from that cache to the host controller and the motherboard memory at electronic speeds.

Solid-State Disks

Sometimes old technologies are used in new ways as economics change or the technologies evolve. An example is the growing importance of **Solid-State Disks**, or **SSDs**. Simply described, an SSD is non-volatile memory that is used like a hard drive. There are many variations of this technology, from DRAM with a battery to allow it to maintain its state in a power failure through flash-memory technologies like single-level cell (SLC) and multilevel cell (MLC) chips.

Magnetic Tapes

Magnetic tape was used as an early secondary-storage medium. Although it is relatively permanent and can hold large quantities of data, its access time is slow compared with that of main memory and magnetic disk. In addition, random access to magnetic tape is about a thousand times slower than random access to magnetic disk, so tapes are not very useful for secondary storage.

Tapes are used mainly for backup, for storage of infrequently used information, and as a medium for transferring information from one system to another. Some tapes have built-in compressions that can more than double the effective storage. Tapes and their drivers are usually categorized by width, including **4, 8, and 19 millimeters and 1/4 and 1/2 inch.** Some are named according to technology, such as LTO-5 and SDLT.

Disk Structure:

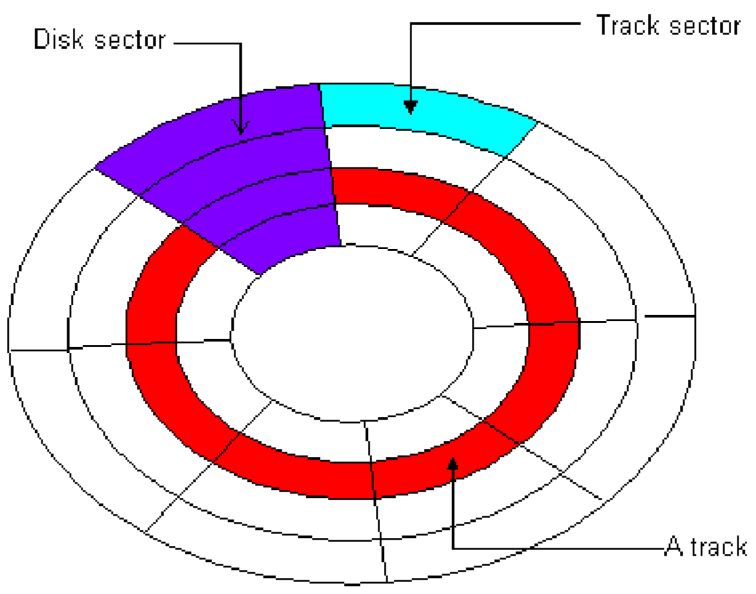
A hard disk is a memory storage device which looks like this:

The disk is divided into **tracks**. Each track is further divided into **sectors**. The point to be noted here is that outer tracks are bigger in size than the inner tracks but they contain the same number of sectors and have equal storage capacity.

This is because the storage density is high in sectors of the inner tracks whereas the bits are sparsely arranged in sectors of the outer tracks.

Some space of every sector is used for formatting. So, the actual capacity of a sector is less than the given capacity. Read-Write(R-W) head moves over the rotating hard disk.

It is this Read-Write head that performs all the read and write operations on the disk and hence, position of the R-W head is a major concern. To perform a read or write operation on a memory location, we need to place the R-W head over that position.



Some important terms must be noted here:

1. **Seek time –** The time taken by the R-W head to reach the desired track from it's current position.
2. **Rotational latency –** Time taken by the sector to come under the R-W head.
3. **Data transfer time –** Time taken to transfer the required amount of data. It depends upon the rotational speed.
4. **Controller time –** The processing time taken by the controller.
5. **Average Access time –** seek time + Average Rotational latency + data transfer time + controller time.

In questions, if the seek time and controller time is not mentioned, take them to be zero. If the amount of data to be transferred is not given, assume that no data is being transferred. Otherwise, calculate the time taken to transfer the given amount of data.

The average of rotational latency is taken when the current position of R-W head is not given. Because, the R-W may be already present at the desired position or it might take a whole rotation to get the desired sector under the R-W head. But, if the current position of the R-W head is given then the rotational latency must be calculated.

Example –
Consider a hard disk with:

4 surfaces
64 tracks/surface
128 sectors/track
256 bytes/sector

1. **What is the capacity of the hard disk?**
   Disk capacity = surfaces * tracks/surface * sectors/track * bytes/sector
   Disk capacity = 4 * 64 * 128 * 256
   Disk capacity = 8 MB

2. **The disk is rotating at 3600 RPM, what is the data transfer rate?**
   60 sec -> 3600 rotations
   1 sec -> 60 rotations
   Data transfer rate = number of rotations per second * track capacity * number of surfaces (since 1 R-W head is used for each surface)
   Data transfer rate = 60 * 128 * 256 *
   4 Data transfer rate = 7.5 MB/sec

3. **The disk is rotating at 3600 RPM, what is the average access time?**
   Since, seek time, controller time and the amount of data to be transferred is not given, we consider all the three terms as 0.
   Therefore, Average Access time = Average rotational
   delay Rotational latency => 60 sec -> 3600 rotations 1 sec
   -> 60 rotations
   Rotational latency = (1/60) sec = 16.67 msec.
   Average Rotational latency = (16.67)/2
   = 8.33 msec.
   Average Access time = 8.33 msec.

Disk Scheduling and Management

Disk scheduling is done by operating systems to schedule I/O requests arriving for disk. Disk scheduling is also known as I/O scheduling.
Disk scheduling is important because:

Multiple I/O requests may arrive by different processes and only one I/O request can be served at a time by disk controller. Thus other I/O requests need to wait in waiting queue and need to be scheduled. Two or more request may be far from each other so can result in greater disk arm movement.
Hard drives are one of the slowest parts of computer system and thus need to be accessed in an efficient manner.

There are many Disk Scheduling Algorithms but before discussing them let’s have a quick look at some of the important terms:

Seek Time: Seek time is the time taken to locate the disk arm to a specified track where the data is to be read or write. So the disk scheduling algorithm that gives minimum average seek time is better.

Rotational Latency: Rotational Latency is the time taken by the desired sector of disk to rotate into a position so that it can access the read/write heads. So the disk scheduling algorithm that gives minimum rotational latency is better.

Transfer Time: Transfer time is the time to transfer the data. It depends on the rotating speed of the disk and number of bytes to be transferred.

Disk Access Time: Disk Access Time is:

Disk Access Time = Seek Time + Rotational Latency + Transfer Time

Disk Response Time: Response Time is the average of time spent by a request waiting to perform its I/O operation. *Average Response time* is the response time of the all requests. *Variance Response Time* is measure of how individual request are serviced with respect to average response time. So the disk scheduling algorithm that gives minimum variance response time is better.

Disk Scheduling Algorithms

1. **FCFS:** FCFS is the simplest of all the Disk Scheduling Algorithms. In FCFS, the requests are addressed in the order they arrive in the disk queue.

Advantages:

Every request gets a fair chance
No indefinite postponement

Disadvantages:

Does not try to optimize seek time
May not provide the best possible service

2. **SSTF:** In SSTF (Shortest Seek Time First), requests having shortest seek time are executed first. So, the seek time of every request is calculated in advance in queue and then they are scheduled according to their calculated seek time. As a result, the request near the disk arm will get executed first. SSTF is certainly an improvement over FCFS as it decreases the average response time and increases the throughput of system.

Advantages:

Average Response Time
Decreases Throughput increases

Disadvantages:

Overhead to calculate seek time in advance
Can cause Starvation for a request if it has higher seek time as compared to incoming
requests High variance of response time as SSTF favours only some requests

3. **SCAN:** In SCAN algorithm the disk arm moves into a particular direction and services the requests coming in its path and after reaching the end of disk, it reverses its direction and again services the request arriving in its path. So, this algorithm works like an elevator and hence also known as **elevator algorithm.** As a result, the requests at the midrange are serviced more and those arriving behind the disk
arm will have to wait.

Advantages:

High throughput
Low variance of response
time Average response time

Disadvantages:

Long waiting time for requests for locations just visited by disk arm

4. **CSCAN**: In SCAN algorithm, the disk arm again scans the path that has been scanned, after reversing its direction. So, it may be possible that too many requests are waiting at the other end or there may be zero or few requests pending at the scanned area.

These situations are avoided in *CSAN* algorithm in which the disk arm instead of reversing its direction goes to the other end of the disk and starts servicing the requests from there. So, the disk arm moves in a circular fashion and this algorithm is also similar to SCAN algorithm and hence it is known as C-SCAN (Circular SCAN).

Advantages:

Provides more uniform wait time compared to SCAN

5. **LOOK:** It is similar to the SCAN disk scheduling algorithm except the difference that the disk arm in spite of going to the end of the disk goes only to the last request to be serviced in front of the head and then reverses its direction from there only. Thus it prevents the extra delay which occurred due to unnecessary traversal to the end of the disk.

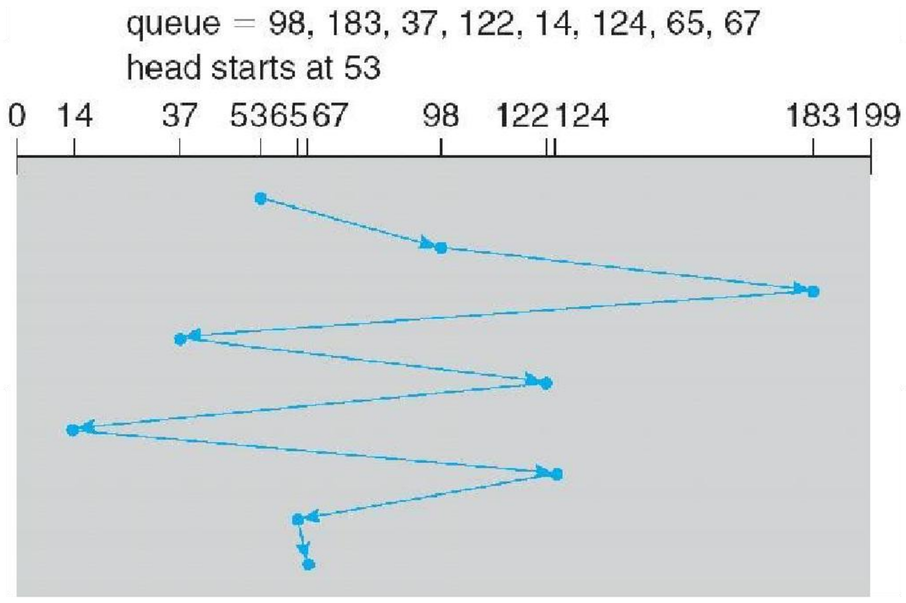
6. **CLOOK:** As LOOK is similar to SCAN algorithm, in similar way, CLOOK is similar to CSCAN disk scheduling algorithm. In CLOOK, the disk arm inspite of going to the end goes only to the last request to be serviced in front of the head and then from there goes to the other end's last request. Thus, it also prevents the extra delay which occurred due to unnecessary traversal to the end of the disk.

1. FCFS Scheduling:

The simplest form of disk scheduling is, of course, the first-come, first-served (FCFS) algorithm. This algorithm is intrinsically fair, but it generally does not provide the fastest service. Consider, for example, a disk queue with requests for I/O to blocks on cylinders

I/O to blocks on cylinders

98, 183, 37, 122, 14, 124, 65, 67.

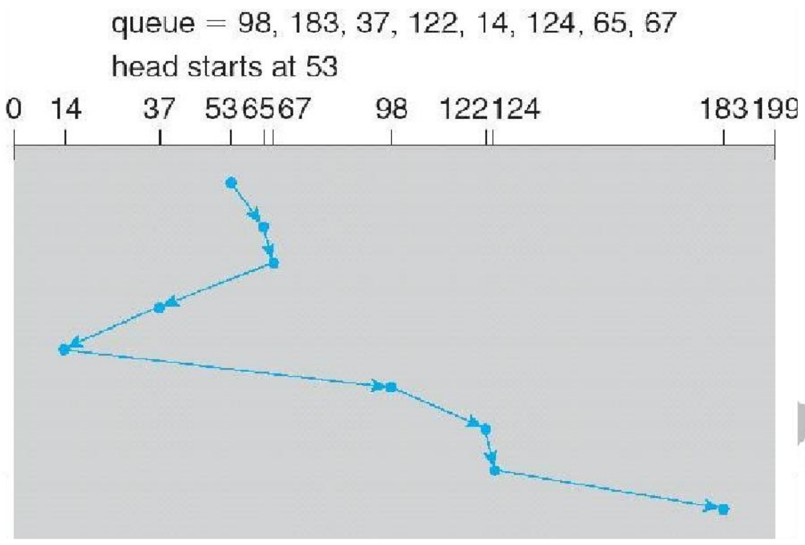


FCFS disk scheduling.

If the disk head is initially at cylinder 53, it will first move from 53 to 98, then to 183, 37, 122, 14, 124, 65, and finally to 67, for **a total head movement of 640 cylinders**. The wild swing from 122 to 14 and then back to 124 illustrates the problem with this schedule. If the requests for cylinders 37 and 14 could be serviced together, before or after the requests for 122 and 124, the total head movement could be decreased substantially, and performance could be thereby improved.

2. **SSTF(shortest-seek-time-first)Scheduling**

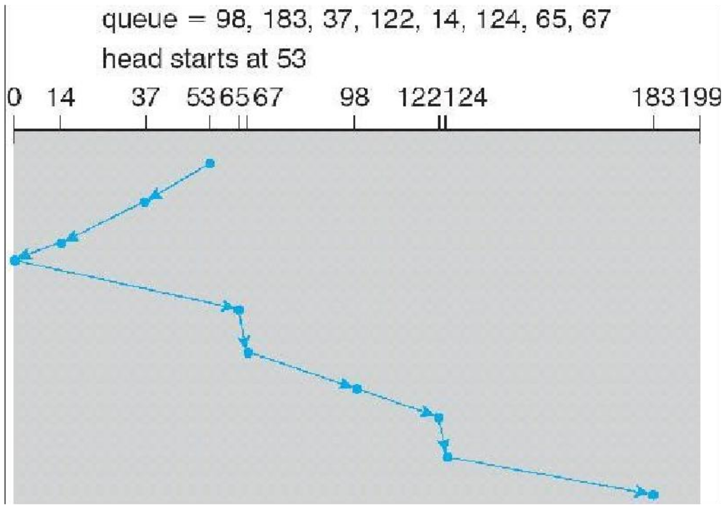
Service all the requests close to the current head position, before moving the head far away to service other requests. That is selects the request with the minimum seek time from the current head position.



Total head movement = 236 cylinders

3. SCAN Scheduling

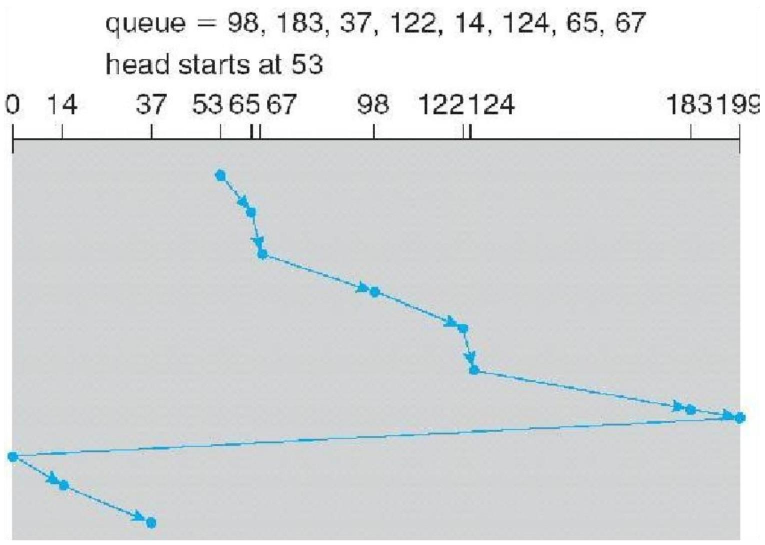
The disk head starts at one end of the disk, and moves toward the other end, servicing requests as it reaches each cylinder, until it gets to the other end of the disk. At the other end, the direction of head movement is reversed, and servicing continues.



SCAN disk scheduling.

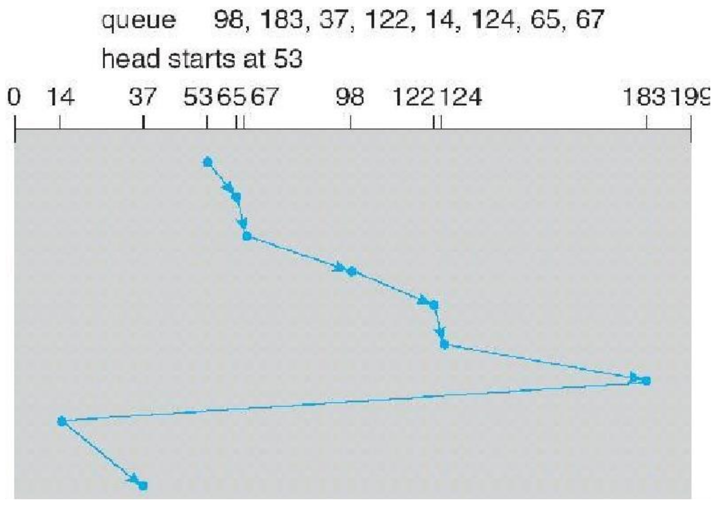
4. C-SCAN Scheduling

Variant of SCAN designed to provide a more uniform wait time. It moves the head from one end of the disk to the other, servicing requests along the way. When the head reaches the other end, however, it immediately returns to the beginning of the disk, without servicing any requests on the return trip.



5. LOOK Scheduling

Both SCAN and C-SCAN move the disk arm across the full width of the disk. In this, the arm goes only as far as the final request in each direction. Then, it reverses direction immediately, without going all the way to the end of the disk. LOOK scheduling improves upon SCAN by looking ahead at the queue of pending requests, and not moving the heads any farther towards the end of the disk than is necessary. The following diagram illustrates the circular form of LOOK:



C-LOOK disk scheduling.

Disk Management

1. Disk Formatting:

Before a disk can store data, the sector is divided into various partitions. This process is called low- level formatting or physical formatting. It fills the disk with a special data structure for each sector. The data structure for a sector consists of

✓ Header,

✓ Data area (usually 512 bytes in size),and

✓ Trailer.

Error-Correcting Code (ECC).

This formatting enables the manufacturer to

1. Test the disk and

2. To initialize the mapping from logical block numbers

To use a disk to hold files, the operating system still needs to record its own data structures on the disk. It does so in two steps.

(a) The first step is **Partition** the disk into one or more groups of cylinders. Among the partitions, one partition can hold a copy of the OS's executable code, while another holds user files.

(b) The second step is **logical formatting** .The operating system stores the initial file-system data structures onto the disk. These data structures may include maps of free and allocated space and an initial empty directory.

2. Boot Block:

For a computer to start running-for instance, when it is powered up or rebooted-it needs to have an initial program to run. This initial program is called bootstrap program & it should be simple. It initializes all aspects of the system, from CPU registers to device controllers and the contents of main memory, and then starts the operating system.

To do its job, the bootstrap program

1. Finds the operating system kernel on disk,

2. Loads that kernel into memory, and

3. Jumps to an initial address to begin the operating system execution.

The bootstrap is stored in read-only memory **(ROM).**

Advantages:

1. ROM needs no initialization.
2. It is at a fixed location that the processor can start executing when powered up or reset.
3. It cannot be infected by a computer virus. Since, ROM is read only.

The full bootstrap program is stored in a partition called the **boot blocks**, at a fixed location on the disk. A disk that has a boot partition is called a **boot disk or system disk**. The code in the boot ROM instructs the disk controller to read the boot blocks into memory and then starts executing that code. **Bootstrap loader**: load the entire operating system from a non-fixed location on disk, and to start the operating system running.

3. Bad Blocks:

The disk with defected sector is called as bad block.

Depending on the disk and controller in use, these blocks are handled in a variety of ways;

Method 1: "Handled manually''

If blocks go bad during normal operation, a **special program** must be run manually to search for the bad blocks and to lock them away as before. Data that resided on the bad blocks usually are lost.

Method 2: "sector sparing or forwarding"

The controller maintains a list of bad blocks on the disk. Then the controller can be told to replace each bad sector logically with one of the spare sectors. This scheme is known as sector sparing or forwarding.

A typical bad-sector transaction might be as follows:

1. The operating system tries to read logical block87.
2. The controller calculates the ECC and finds that the sector is bad.
3. It reports this finding to the operating system.
4. The next time that the system is rebooted, a special command is run to tell the controller to replace the bad sector with a spare.
5. After that, whenever the system requests logical block 87, the request is translated into the replacement sector's address by the controller.

Method 3: "sector slipping"

For an example, suppose that logical block 17 becomes defective, and the first available spare follows sector 202. Then, sector slipping would remap all the sectors from 17 to 202, moving them all down one spot. That is, sector 202 would be copied into the spare, then sector 201 into 202, and then 200 into 201, and so on, until sector 18 is copied into sector 19. Slipping the sectors in this way frees up the space of sector 18, so sector 17 can be mapped to it.

Swap Space Management:

Modern systems typically swap out pages as needed, rather than swapping out entire processes. Hence the swapping system is part of the virtual memory management system.

Managing swap space is obviously an important task for modern OS.

Swap-Space Use

The amount of swap space needed by an OS varies greatly according to how it is used. Some systems require an amount equal to physical RAM; some want a multiple of that; some want an amount equal to the amount by which virtual memory exceeds physical RAM, and some systems use little or none at all!

Some systems support multiple swap spaces on separate disks in order to speed up the virtual memory system.

The interchange of data between virtual memory and real memory is called as swapping and **space** on disk as "**swap space**".

Swap-Space Location

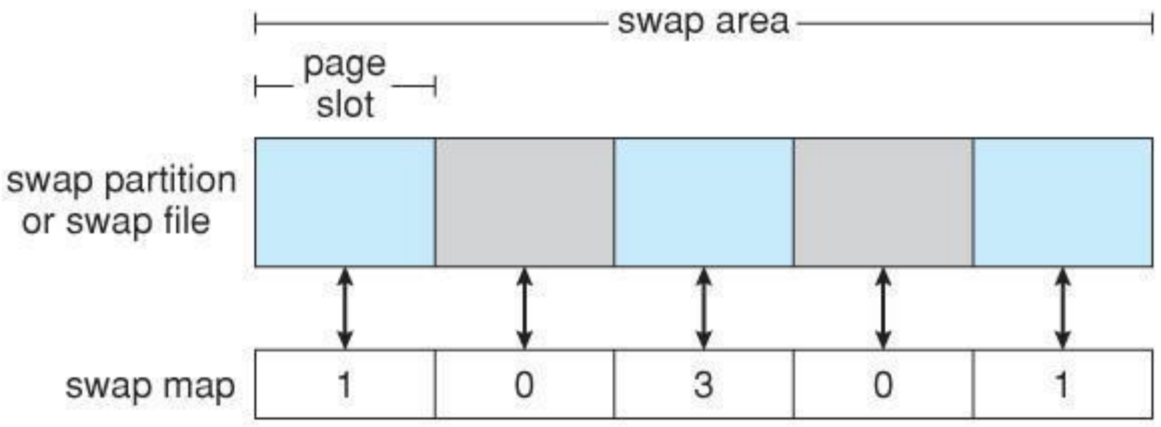
Swap space can be physically located in one of two locations:

As a large file which is part of the regular file system. This is easy to implement, but inefficient. Not only must the swap space be accessed through the directory system, the file is also subject to fragmentation issues. Caching the block location helps in finding the physical blocks, but that is not a complete fix.

As a raw partition, possibly on a separate or little-used disk. This allows the OS more control over swap space management, which is usually faster and more efficient. Fragmentation of swap space is generally not a big issue, as the space is re-initialized every time the system is rebooted. The downside of keeping swap space on a raw partition is that it can only be grown by repartitioning the hard drive.

Swap-Space Management: An Example

Historically OS swapped out entire processes as needed. Modern systems swap out only individual pages, and only as needed. In the mapping system shown below for Linux systems, a map of swap space is kept in memory, where each entry corresponds to a 4K block in the swap space. Zeros indicate free slots and non-zeros refer to how many processes have a mapping to that particular block (>1 for shared pages only.)



The data structures for swapping on Linux systems.

<u>File-System Interface – File concept</u>

<u>**File:**</u> A file is a named collection of related information that is recorded on secondary storage such as magnetic disks, magnetic tapes and optical disks. In general, a file is a sequence of bits, bytes, lines or records whose meaning is defined by the files creator and user.

File Attributes

Different OS keep track of different file attributes, including:

Name - Some systems give special significance to names, and particularly extensions (.exe, .txt, etc.), and some do not. Some extensions may be of significance to the OS (.exe), and others only to certain applications (.jpg)
Identifier
Type - Text, executable, other binary, etc.
Location - on the hard drive.
Size
Protection
Time & Date
User ID

File Operations

The file **ADT** supports many common operations:

- **Creating a file**
- **Writing a file**
- **Reading a file**

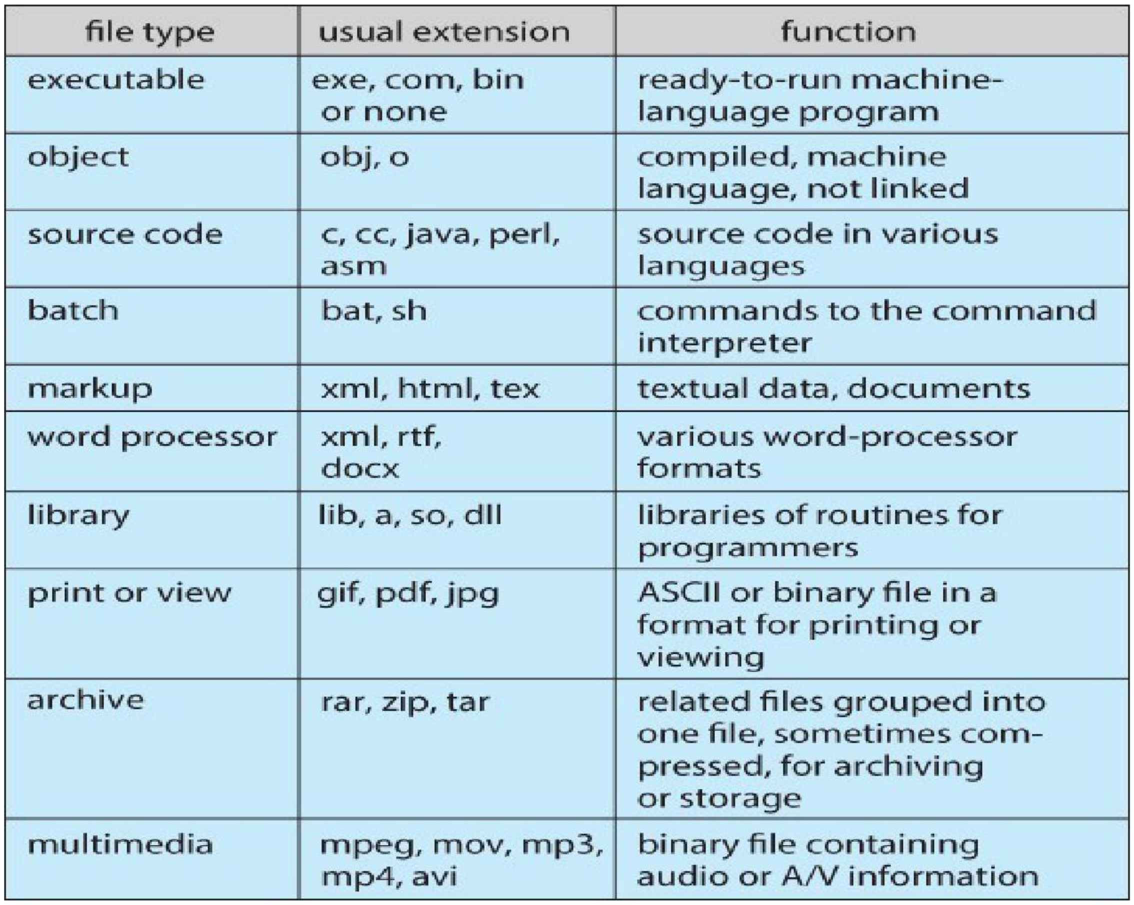
- **Repositioning within a file**
- **Deleting a file**
- **Truncating a file.**

Some systems provide support for *file locking.*

- A ***shared lock*** is for reading only.
- A ***exclusive lock*** is for writing as well as reading.
- An ***advisory lock*** is informational only, and not enforced. (A "Keep Out" sign, which may be ignored.)
- A ***mandatory lock*** is enforced. (A truly locked door.)
- UNIX used advisory locks, and Windows uses mandatory locks.

File Types

Windows (and some other systems) use special file extensions to indicate the type of each file:



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

File Structure

Some files contain an internal structure, which may or may not be known to the OS.

For the OS to support particular file formats increases the size and complexity of the OS.

UNIX treats all files as sequences of bytes, with no further consideration of the internal structure. (With the exception of executable binary programs, which it must know how to load and find the first executable statement, etc.)

Macintosh files have **two *forks* - a *resource fork*, and a *data fork***. The resource fork contains information relating to the UI, such as icons and button images, and can be modified independently of the data fork, which contains the code or data as appropriate.

A File Structure should be according to a required format that the operating system can understand.

A **file** has a certain defined structure according to its type. A

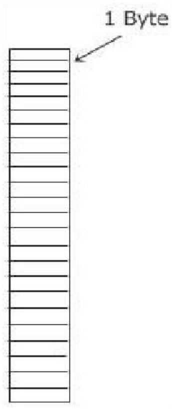
text file is a sequence of characters organized into lines. A

source file is a sequence of procedures and functions.

An **object file** is a sequence of bytes organized into blocks that are understandable by the machine.

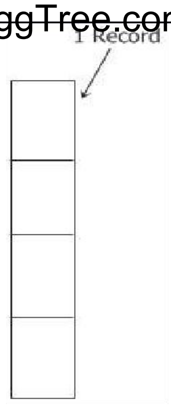
Files can be structured in several ways in which three common structures are given in this tutorial with their short description one by one.

File Structure 1



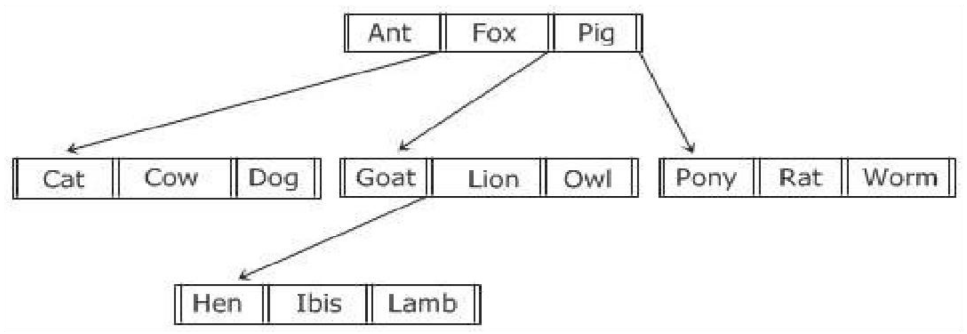
Here, as you can see from the above figure, the file is an unstructured sequence of bytes. Therefore, the OS doesn't care about what is in the file, as all it sees are bytes.

File Structure 2



Now, as you can see from the above figure that shows the second structure of a file, where a file is a sequence of fixed-length records where each with some internal structure. Central to the idea about a file being a sequence of records is the idea that read operation returns a record and write operation just appends a record.

File Structure 3



Now in the last structure of a file that you can see in the above figure, a file basically consists of a tree of records, not necessarily all the same length, each containing a key field in a fixed position in the record. The tree is stored on the field, just to allow the rapid searching for a specific key.

Access Methods

File access mechanism refers to the manner in which the records of a file may be accessed. There are several ways to access files −

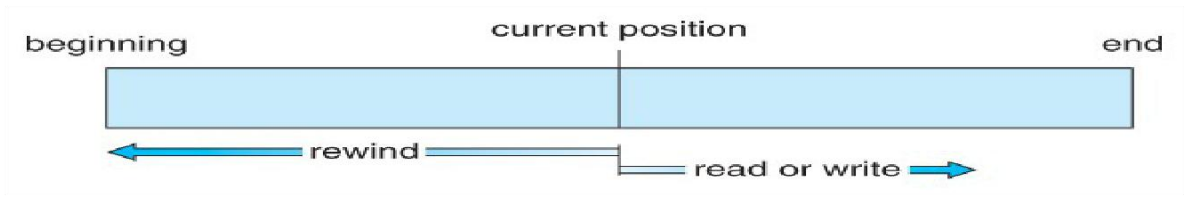
Sequential access

Direct/Random access

Indexed sequential access

Sequential access

A sequential access is that in which the records are accessed in some sequence, i.e., the information in the file is processed in order, one record after the other. This access method is the most primitive one. Example: Compilers usually access files in this fashion.



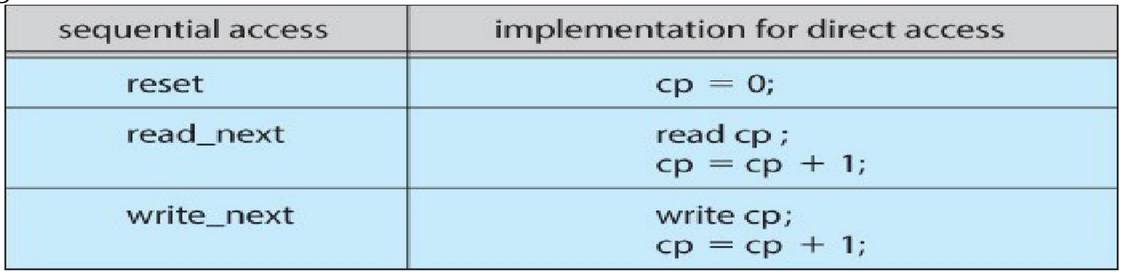
Sequential-access file.

Direct/Random access

Random access file organization provides, accessing the records directly.

Each record has its own address on the file with by the help of which it can be directly accessed for reading or writing.

The records need not be in any sequence within the file and they need not be in adjacent locations on the storage medium.



sequential access	implementation for direct access
reset	cp = 0;
read_next	read cp ; cp = cp + 1;

write_next	write cp;
	cp = cp + 1;

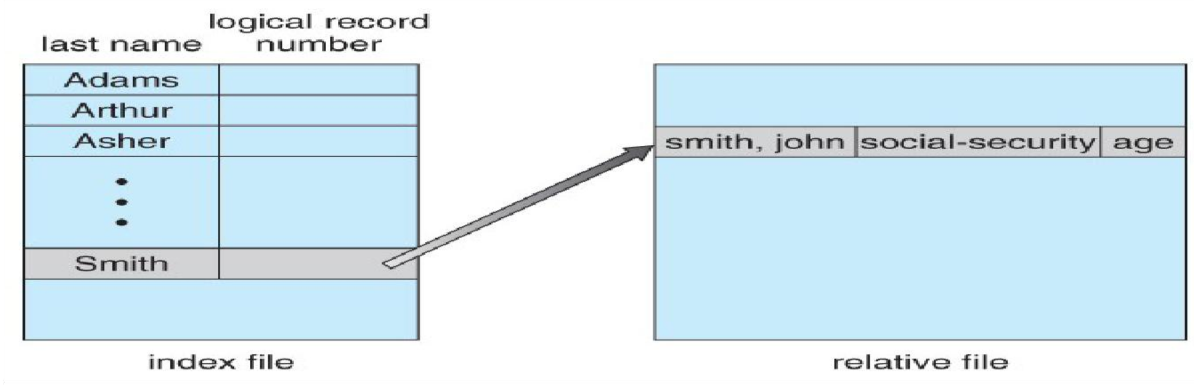
Simulation of sequential access on a direct-access file.

Indexed sequential access

This mechanism is built up on base of sequential access.

An index is created for each file which contains pointers to various blocks.

Index is searched sequentially and its pointer is used to access the file directly.



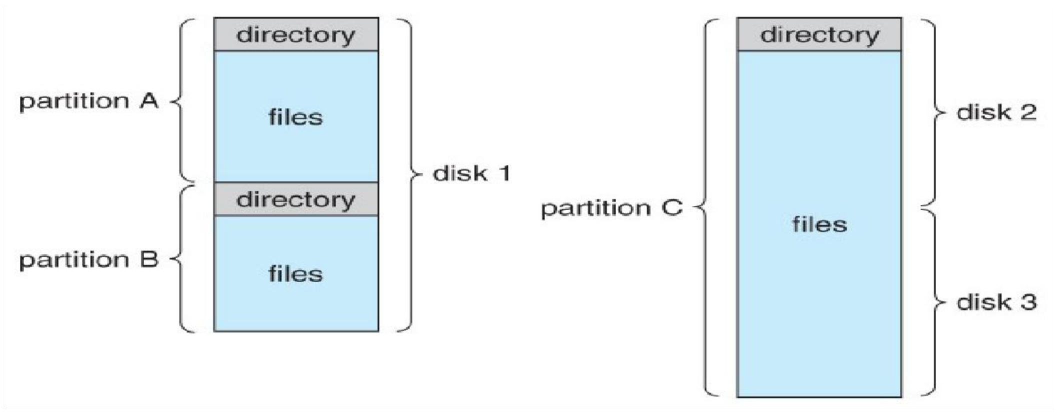
Example of index and relative files.

Directory Structure:

A directory is a container that is used to contain folders and file. It organizes files and folders into hierarchical manner.

Storage Structure

A disk can be used in its entirety for a file system. Alternatively a physical disk can be broken up into multiple ***partitions, slices, or mini-disks***, each of which becomes a virtual disk and can have its own file system(or be used for raw storage, swap space, etc.)Or, multiple physical disks can be combined into one *volume*, i.e. a larger virtual disk, with its own file system spanning the physical disks.



A typical file-system organization.

Directory Overview

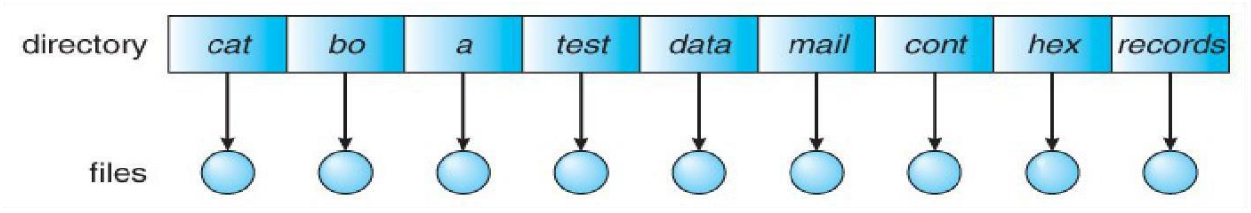
Directory operations to be supported include:

- Search for a file
- Create a file - add to the directory
- Delete a file - erase from the directory
- List a directory - possibly ordered in different ways.
- Rename a file - may change sorting order
- Traverse the file system.

Directory organization

Single-Level Directory

Simple to implement, but each file must have a unique name. The simplest method is to have one big list of all the files on the disk. The entire system will contain only one **directory** which is supposed to mention all the files present in the file system. The **directory** contains one entry per each file present on the file system.



Single-level directory.

Two-Level Directory

Each user gets their own directory space.

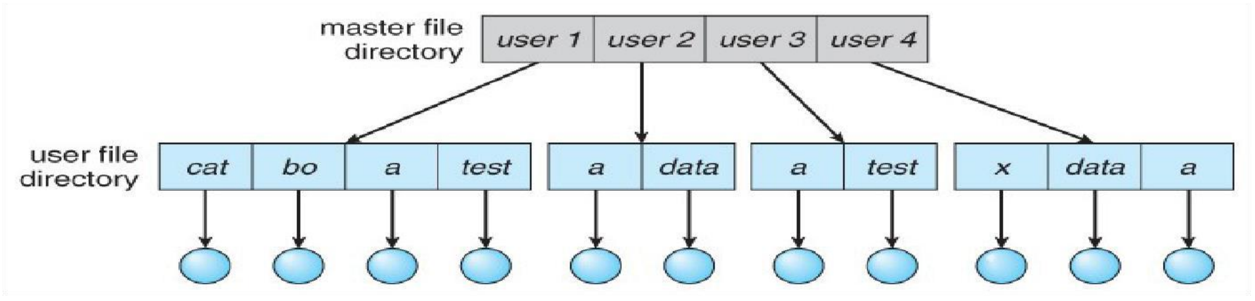
File names only need to be unique within a given user's directory.

A master file directory is used to keep track of each users directory, and must be maintained when users are added to or removed from the system.

A separate directory is generally needed for system (executable) files.

Systems may or may not allow users to access other directories besides their own

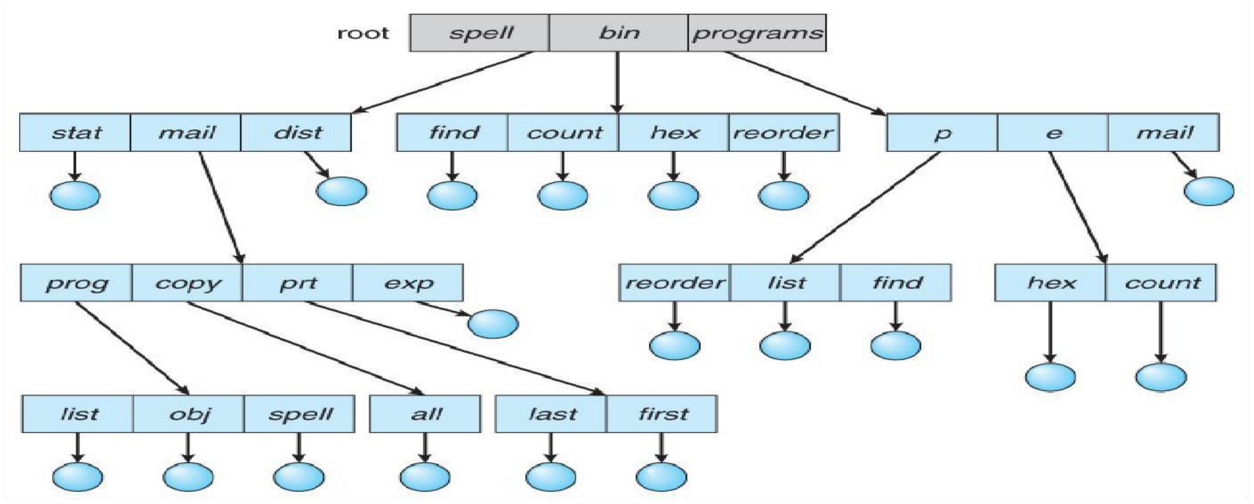
- If access to other directories is allowed, then provision must be made to specify the directory being accessed.
- If access is denied, then special consideration must be made for users to run programs located in system directories. A *search path* is the list of directories in which to search for executable programs, and can be set uniquely for each user.



Two-level directory structure.

Tree-Structured Directories

An obvious extension to the two-tiered directory structure, and the one with which we are all most familiar. Each user / process has the concept of a ***current directory*** from which all (relative) searches take place. Files may be accessed using either absolute pathnames (relative to the root of the tree) or relative pathnames (relative to the current directory.)Directories are stored the same as any other file in the system, except there is a bit that identifies them as directories, and they have some special structure that the OS understands. One question for consideration is whether or not to allow the removal of directories that are not empty - Windows requires that directories be emptied first, and UNIX provides an option for deleting entire sub-trees.



Tree-structured directory structure.

Acyclic-Graph Directories

When the same files need to be accessed in more than one place in the directory structure (e.g. because they are being shared by more than one user / process), it can be useful to provide an acyclic-graph structure. (Note the *directed* arcs from parent to child.)

UNIX provides two types of *links* for implementing the acyclic-graph structure. (See "man ln" for more details.)

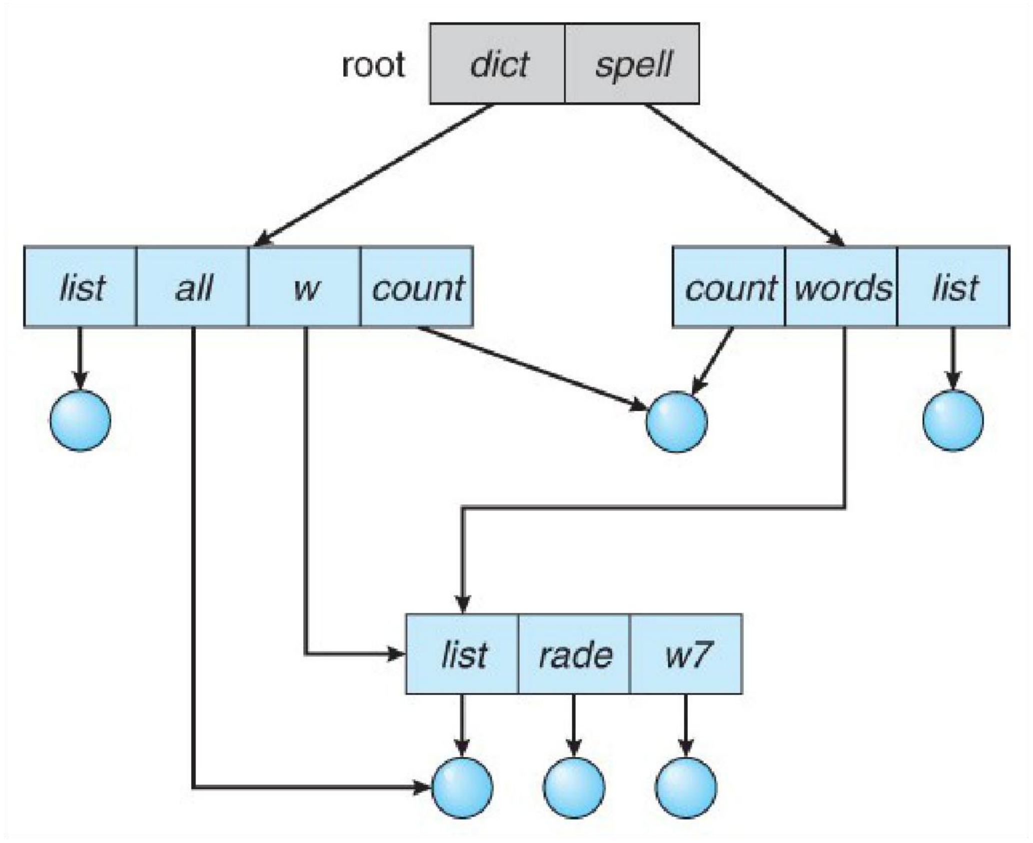
- **A *hard link*** (usually just called a link) involves multiple directory entries that both refer to the same file. Hard links are only valid for ordinary files in the same file system.
- **A *symbolic link*** that involves a special file, containing information about where to find the linked file. Symbolic links may be used to link directories and/or files in other file systems, as well as ordinary files in the current file system.

Windows only supports symbolic links, termed *shortcuts.*

Hard links require a *reference count*, or *link count* for each file, keeping track of how many directory entries are currently referring to this file. Whenever one of the references is removed the link count is reduced, and when it reaches zero, the disk space can be reclaimed.

For symbolic links there is some question as to what to do with the symbolic links when the original file is moved or deleted:

- One option is to find all the symbolic links and adjust them also.
- Another is to leave the symbolic links dangling, and discover that they are no longer valid the next time they are used.
- What if the original file is removed, and replaced with another file having the same name before the symbolic link is next used?



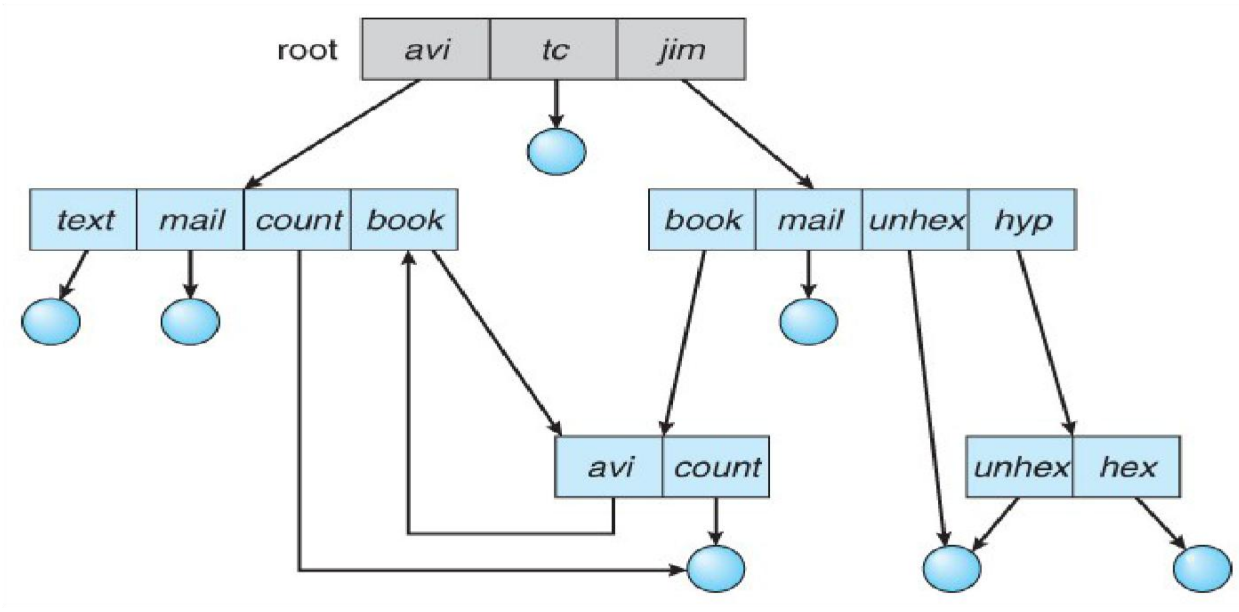
Acyclic-graph directory structure.

General Graph Directory

If cycles are allowed in the graphs, then several problems can arise:

- Search algorithms can go into infinite loops. One solution is to not follow links in search algorithms. (Or not to follow symbolic links, and to only allow symbolic links to refer to directories.)

- Sub-trees can become disconnected from the rest of the tree and still not have their reference counts reduced to zero. Periodic garbage collection is required to detect and resolve this problem.



General graph directory.

File system mounting:

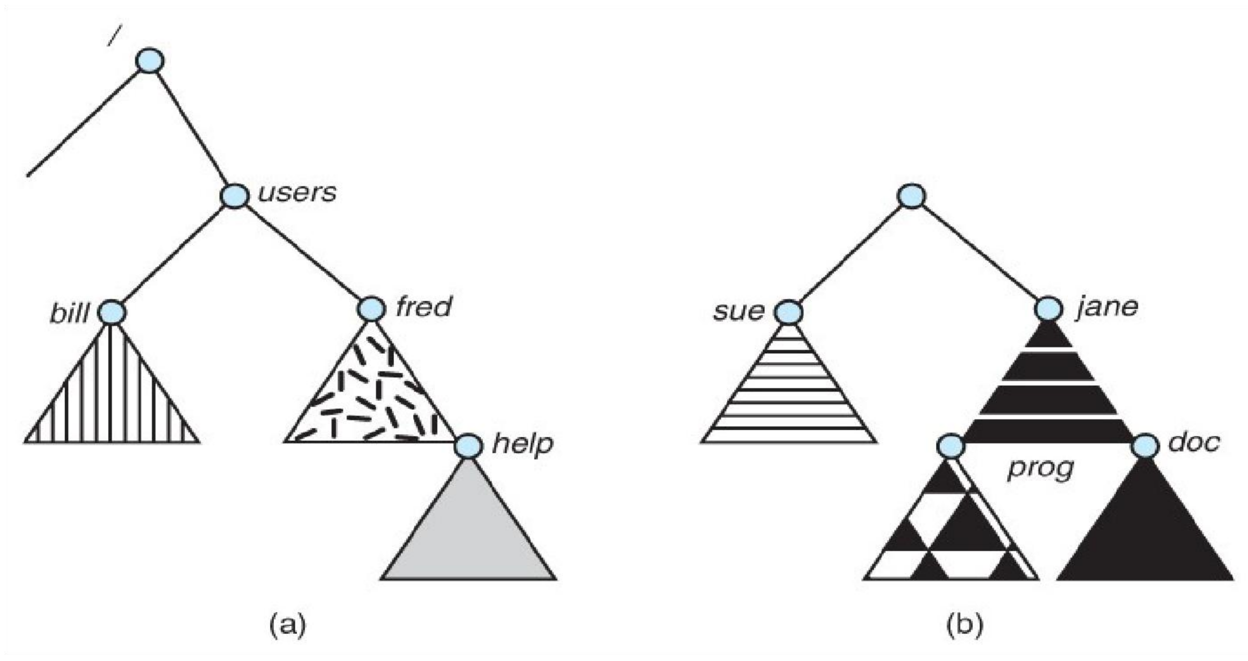
- The basic idea behind mounting file systems is to combine multiple file systems into one large tree structure.
- The mount command is given a file system to mount and a ***mount point*** (directory) on which to attach it.
- Once a file system is mounted onto a mount point, any further references to that directory actually refer to the root of the mounted file system.

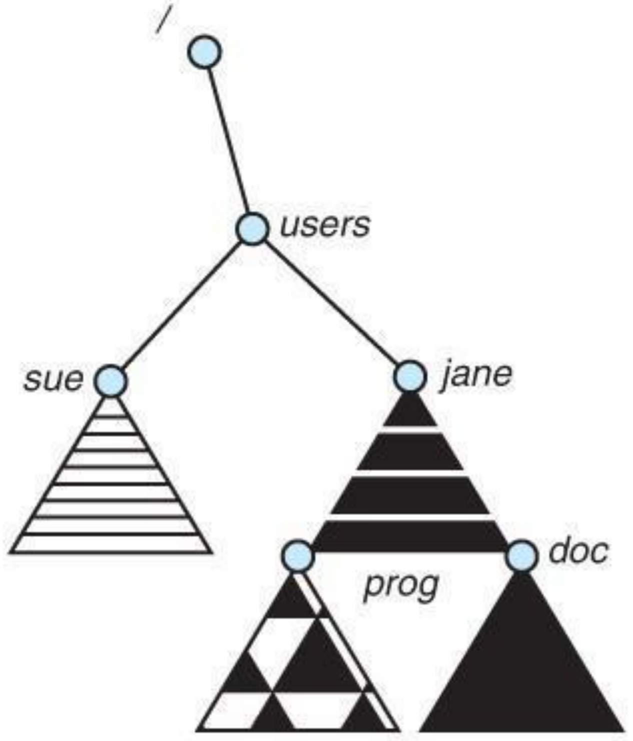
Any files (or sub-directories) that had been stored in the mount point directory prior to mounting the new file system are now hidden by the mounted file system, and are no longer available. For this reason some systems only allow mounting onto empty directories.

File systems can only be mounted by root, unless root has previously configured certain filesystems to be mountable onto certain pre-determined mount points. (E.g. root may allow users to mount floppy filesystems to /mnt or something like it.) Anyone can run the mount command to see what file systems is currently mounted.

Filesystems may be mounted read-only, or have other restrictions imposed.

(a) Existing System **(b) Unmounted Volume**





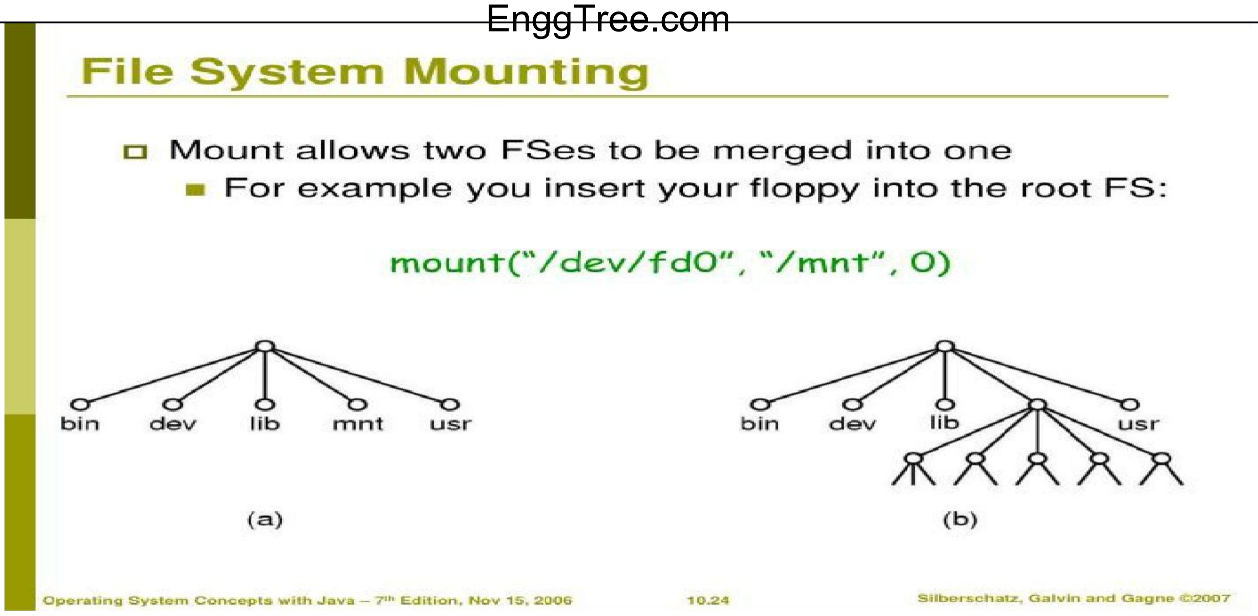
.

Mount point.

The traditional Windows OS runs an extended two-tier directory structure, where the first tier of the structure separates volumes by drive letters, and a tree structure is implemented below that level.

Macintosh runs a similar system, where each new volume that is found is automatically mounted and added to the desktop when it is found.

More recent Windows systems allow filesystems to be mounted to any directory in the filesystem, much like UNIX.

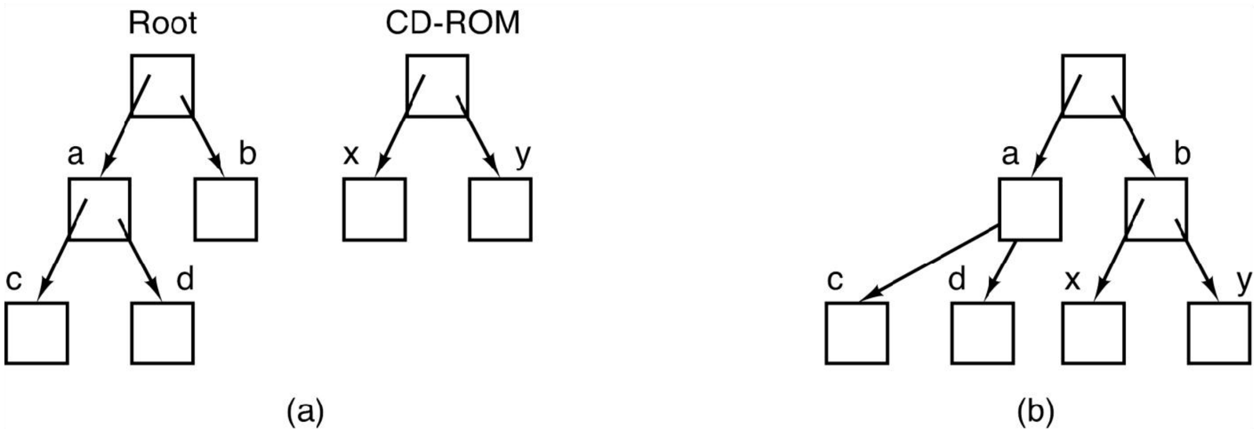


File System Mounting

- Mount allows two FSes to be merged into one
  - For example you insert your floppy into the root FS:

```
mount("/dev/fd0", "/mnt", 0)
```

bin
dev
lib
mnt
usr
(a)
bin
dev
lib
usr
(b)
Operating System Concepts with Java – 7th Edition, Nov 15, 2006
10.24
Silberschatz, Galvin and Gagne ©2007



File Sharing and Protection:

File Sharing

Multiple Users

On a multi-user system, more information needs to be stored for each file:

- The owner (user) who owns the file, and who can control its access.
- The group of other user IDs that may have some special access to the file.
- What access rights are afforded to the owner (User), the Group, and to the rest of the world (the universe, a.k.a. Others.)

- Some systems have more complicated access control, allowing or denying specific accesses to specifically named users or groups.

Remote File Systems

The advent of the Internet introduces issues for accessing files stored on remote computers

- The original method was ftp, allowing individual files to be transported across systems as needed. Ftp can be either account or password controlled, or *anonymous*, not requiring any user name or password.
- Various forms of ***distributed file systems*** allow remote file systems to be mounted onto a local directory structure, and accessed using normal file access commands. (The actual files are still transported across the network as needed, possibly using ftp as the underlying transport mechanism.)
- The WWW has made it easy once again to access files on remote systems without mounting their filesystems, generally using (anonymous) ftp as the underlying file transport mechanism.

The Client-Server Model

When one computer system remotely mounts a file system that is physically located on another system, the system which physically owns the files acts as a *server*, and the system which mounts them is the *client.*

User IDs and group IDs must be consistent across both systems for the system to work properly. (I.e. this is most applicable across multiple computers managed by the same organization, shared by a common group of users.)

The same computer can be both a client and a server. (E.g. cross-linked file systems.

) There are a number of security concerns involved in this model:

- Servers commonly restrict mount permission to certain trusted systems only. Spoofing (a computer pretending to be a different computer) is a potential security risk.
- Servers may restrict remote access to read-only.
- Servers restrict which filesystems may be remotely mounted. Generally the information within those subsystems is limited, relatively public, and protected by frequent backups.

The NFS (Network File System) is a classic example of such a system.

Distributed Information Systems

The ***Domain Name System***, *DNS,* provides for a unique naming system across all of the Internet.

Domain names are maintained by the ***Network Information System, NIS***, which unfortunately has several security issues. NIS+ is a more secure version, but has not yet gained the same widespread acceptance as NIS.

Microsoft's ***Common Internet File System**, **CIFS,*** establishes a *network login* for each user on a networked system with shared file access. Older Windows systems used *domains*, and newer systems (XP, 2000), use *active directories.* User names must match across the network for this system to be valid.

A newer approach is the ***Lightweight Directory-Access Protocol***, ***LDAP***, which provides a *secure single sign-on* for all users to access all resources on a network. This is a secure system which is gaining in popularity, and which has the maintenance advantage of combining authorization information in one central location.

Failure Modes

When a local disk file is unavailable, the result is generally known immediately, and is generally non-recoverable. The only reasonable response is for the response to fail.

However when a remote file is unavailable, there are many possible reasons, and whether or not it is unrecoverable is not readily apparent. Hence most remote access systems allow for blocking or delayed response, in the hopes that the remote system (or the network) will come back up eventually.

Consistency Semantics

Consistency Semantics deals with the consistency between the views of shared files on a networked system. When one user changes the file, when do other users see the changes?

At first glance this appears to have all of the synchronization issues discussed in Chapter 6. Unfortunately the long delays involved in network operations prohibit the use of atomic operations as discussed in that chapter.

UNIX Semantics

The UNIX file system uses the following semantics:

- Writes to an open file are immediately visible to any other user who has the file open.
- One implementation uses a shared location pointer, which is adjusted for all sharing users.

The file is associated with a single exclusive physical resource, which may delay some accesses.

Session Semantics

The Andrew File System, AFS uses the following semantics:

- Writes to an open file are not immediately visible to other users.

- When a file is closed, any changes made become available only to users who open the file at a later time.

According to these semantics, a file can be associated with multiple (possibly different) views. Almost no constraints are imposed on scheduling accesses. No user is delayed in reading or writing their personal copy of the file.

AFS file systems may be accessible by systems around the world. Access control is maintained through (somewhat) complicated access control lists, which may grant access to the entire world (literally) or to specifically named users accessing the files from specifically named remote environments.

Immutable-Shared-Files Semantics

Under this system, when a file is declared as *shared* by its creator, it becomes immutable and the name cannot be re-used for any other resource. Hence it becomes read-only, and shared access is simple.

Protection

The processes in an operating system must be **protected** from one another's activities. To provide such **protection**, we can use various mechanisms to ensure that only processes that have gained proper authorization from the operating system can operate on the **files**, memory segments, CPU, and other resources of a system.

Goals of Protection

Obviously to prevent malicious misuse of the system by users or programs. See chapter 15 for a more thorough coverage of this goal.

To ensure that each shared resource is used only in accordance with system *policies*, which may be set either by system designers or by system administrators.

To ensure that errant programs cause the minimal amount of damage possible.

Note that protection systems only provide the *mechanisms* for enforcing policies and ensuring reliable systems. It is up to administrators and users to implement those mechanisms effectively.

Principles of Protection

The ***principle of least privilege*** dictates that programs, users, and systems be given just enough privileges to perform their tasks.

This ensures that failures do the least amount of harm and allow the least of harm to be done.

For example, if a program needs special privileges to perform a task, it is better to make it a SGID program with group ownership of "network" or "backup" or some other pseudo group, rather than SUID with root ownership. This limits the amount of damage that can occur if something goes wrong.

Typically each user is given their own account, and has only enough privilege to modify their own files.

The root account should not be used for normal day to day activities - The System Administrator should also have an ordinary account, and reserve use of the root account for only those tasks which need the root privileges.

Domain of Protection

A computer can be viewed as a collection of *processes* and *objects* (both HW & SW).

The ***need to know principle*** states that a process should only have access to those objects it needs to accomplish its task, and furthermore only in the modes for which it needs access and only during the time frame when it needs access.

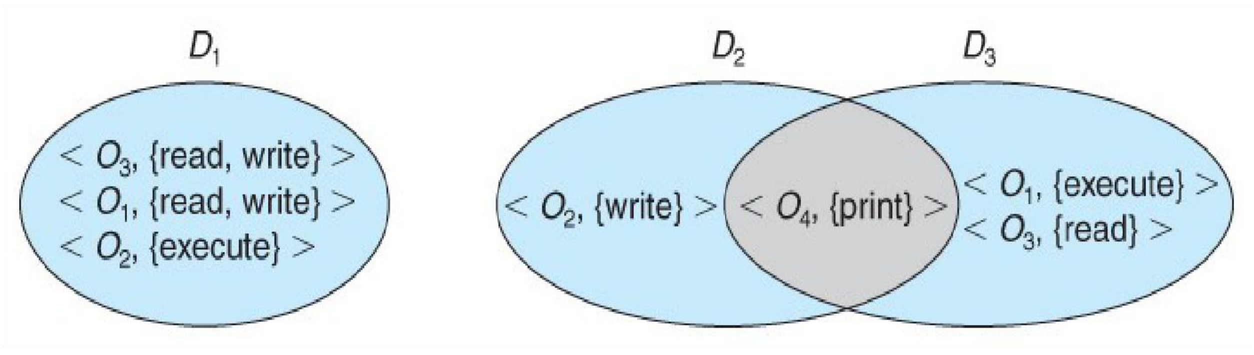
The modes available for a particular object may depend upon its type.

Domain Structure

A ***protection domain*** specifies the resources that a process may access.

Each domain defines a set of objects and the types of operations that may be invoked on each object. An ***access right*** is the ability to execute an operation on an object.

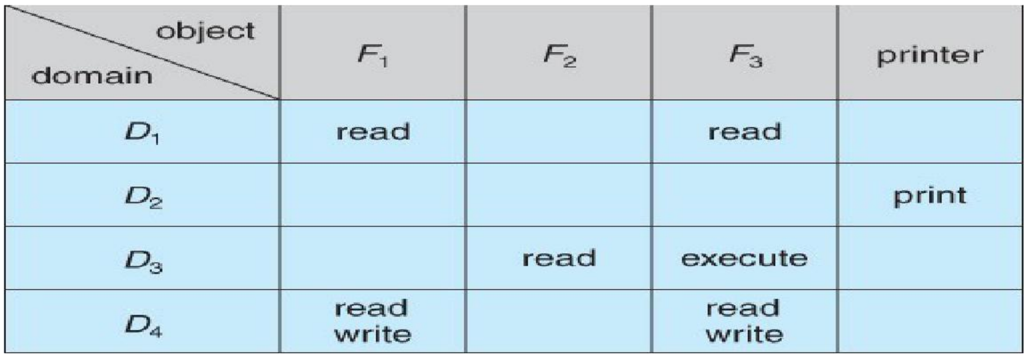
A domain is defined as a set of < object, { access right set } > pairs, as shown below. Note that some domains may be disjoint while others overlap.



System with three protection domains.

The association between a process and a domain may be ***static* or *dynamic.***

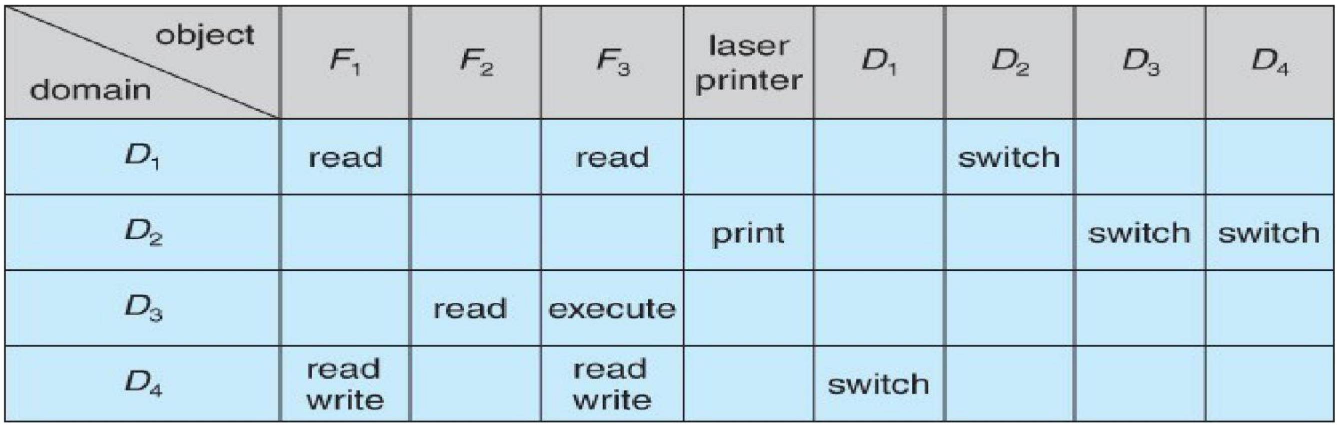
If the association is static, then the need-to-know principle requires a way of changing the contents of the domain dynamically. If the association is dynamic, then there needs to be a mechanism for ***domain switching.*** Domains may be realized in different fashions - as users, or as processes, or as procedures. E.g. if each user corresponds to a domain, then that domain defines the access of that user, and changing domains involves changing user ID. The model of protection that we have been discussing can be viewed as an ***access matrix,*** in which columns represent different system resources and rows represent different protection domains. Entries within the matrix indicate what access that domain has to that resource.



object / domain	F_1	F_2	F_3	printer
D_1	read		read	
D_2				print
D_3		read	execute	
D_4	read write		read write	

Access matrix.

Domain switching can be easily supported under this model, simply by providing "switch" access to other domains:



object / domain	F_1	F_2	F_3	laser printer	D_1	D_2	D_3	D_4
D_1	read		read			switch		
D_2				print			switch	switch
D_3		read	execute					
D_4	read write		read write		switch			

Access matrix of above Figure with domains as objects.

Types of Access

The following low-level operations are often controlled
Read - View the contents of the file
Write - Change the contents of the file.
Execute - Load the file onto the CPU and follow the instructions contained therein.
Append - Add to the end of an existing file.
Delete - Remove a file from the system.
List -View the name and other attributes of files on the system.
Higher-level operations, such as copy, can generally be performed through combinations of the above.

Access Control

One approach is to have complicated ***Access Control Lists, ACL***, which specify exactly what access is allowed or denied for specific users or groups.

The AFS uses this system for distributed access.

Control is very finely adjustable, but may be complicated, particularly when the specific users involved are unknown. (AFS allows some wild cards, so for example all users on a certain remote system may be trusted, or a given username may be trusted when accessing from any remote system.)

UNIX uses a set of 9 access control bits, in three groups of three. These correspond to R, W, and X permissions for each of the Owner, Group, and Others. (See "man chmod" for full details.) The RWX bits control the following privileges for ordinary files and directories:

bit	Files	Directories
R	Read (view) file contents.	Read directory contents. Required to get a listing of the directory.
W	Write (change) file contents.	Change directory contents. Required to create or delete files.
X	Execute file contents as a program.	Access detailed directory information. Required to get a long listing, or to access any specific file in the directory. Note that if a user has X but not R permissions on a directory, they can still access specific files, but only if they already know the name of the file they are trying to access.

File System Structure

Hard disks have two important properties that make them suitable for secondary storage of files in file systems: (1) Blocks of data can be rewritten in place, and (2) they are direct access, allowing any block of data to be accessed with only (relatively) minor movements of the disk heads and rotational latency.

Disks are usually accessed in physical blocks, rather than a byte at a time. Block sizes may range from **512 bytes to 4K or larger.**

File systems organize storage on disk drives, and can be viewed as a layered design:

- At the lowest layer are the physical devices, consisting of the magnetic media, motors & controls, and the electronics connected to them and controlling them. Modern disk put more and more of the electronic controls directly on the disk drive itself, leaving relatively little work for the disk controller card to perform.

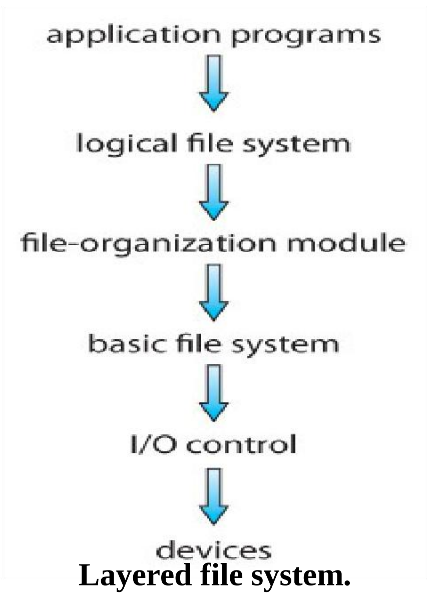
- ***I/O Control*** consists of ***device drivers***, special software programs (often written in assembly) which communicate with the devices by reading and writing special codes directly to and from memory addresses corresponding to the controller card's registers. Each controller card (device) on a system has a different set of addresses (registers, a.k.a. ***ports***) that it listens to, and a unique set of command codes and results codes that it understands.

- The ***basic file system*** level works directly with the device drivers in terms of retrieving and storing raw blocks of data, without any consideration for what is in each block. Depending on the system, blocks may be referred to with a single block number, (e.g. block # 234234), or with head-sector-cylinder combinations.

- The ***file organization module*** knows about files and their logical blocks, and how they map to physical blocks on the disk. In addition to translating from logical to physical blocks, the file organization module also maintains the list of free blocks, and allocates free blocks to files as needed.

- The ***logical file system*** deals with all of the meta data associated with a file (UID, GID, mode, dates, etc), i.e. everything about the file except the data itself. This level manages the directory structure and the mapping of file names to ***file control blocks, FCBs***, which contain all of the meta data as well as block number information for finding the data on the disk.

The layered approach to file systems means that much of the code can be used uniformly for a wide variety of different file systems, and only certain layers need to be file system specific. Common file systems in use include the UNIX file system, UFS, the Berkeley Fast File System, FFS, Windows systems FAT, FAT32, NTFS, CD-ROM systems ISO 9660, and for Linux the extended file systems ext2 and ext3 (among 40 others supported.)



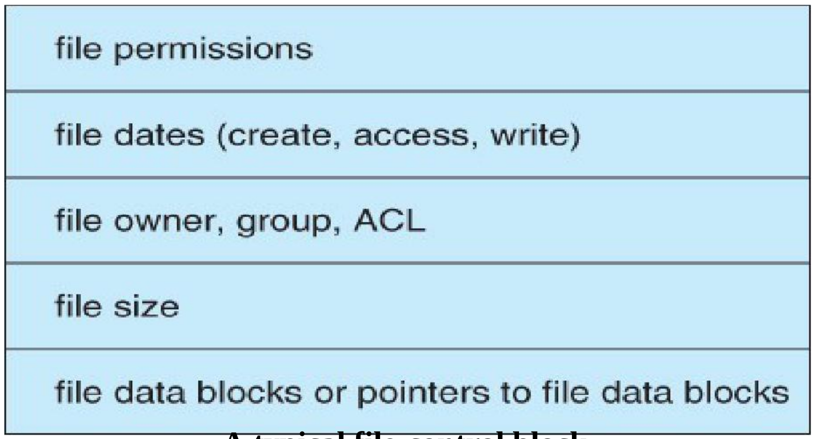
Layered file system.

File System Implementation:

Overview

File systems store several important data structures on the disk:

- A ***boot-control block***, (per volume) a.k.a. the *boot block* in UNIX or the *partition boot sector* in Windows contains information about how to boot the system off of this disk. This will generally be the first sector of the volume if there is a bootable system loaded on that volume, or the block will be left vacant otherwise.
- A ***volume control block***, (per volume) a.k.a. the *master file table* in UNIX or the *superblock* in Windows, which contains information such as the partition table, number of blocks on each file system, and pointers to free blocks and free FCB blocks.
- A directory structure (per file system), containing file names and pointers to corresponding FCBs. UNIX uses inode numbers, and NTFS uses a *master file table.*
- The ***File Control Block***, *FCB*, (per file) containing details about ownership, size, permissions, dates, etc. UNIX stores this information in inodes, and NTFS in the master file table as a relational database structure.



file permissions
file dates (create, access, write)
file owner, group, ACL
file size
file data blocks or pointers to file data blocks

A typical file-control block.

There are also several key data structures stored in memory:

- An in-memory mount table.
- An in-memory directory cache of recently accessed directory information.
- *A* ***system-wide open file table***, containing a copy of the FCB for every currently open file in the system, as well as some other related information.

- *A* ***per-process open file table***, containing a pointer to the system open file table as well as some other information. (For example the current file position pointer may be either here or in the system file table, depending on the implementation and whether the file is being shared or not.)

Figure illustrates some of the interactions of file system components when files are created and/or used:

- When a new file is created, a new FCB is allocated and filled out with important information regarding the new file. The appropriate directory is modified with the new file name and FCB information.
- When a file is accessed during a program, the open ( ) system call reads in the FCB information from disk, and stores it in the system-wide open file table. An entry is added to the per-process open file table referencing the system-wide table, and an index into the per-process table is returned by the open( ) system call. UNIX refers to this index as a *file descriptor*, and Windows refers to it as a *file handle*.
- If another process already has a file open when a new request comes in for the same file, and it is sharable, then a counter in the system-wide table is incremented and the per-process table is adjusted to point to the existing entry in the system-wide table.
- When a file is closed, the per-process table entry is freed, and the counter in the system-wide table is decremented. If that counter reaches zero, then the system wide table is also freed. Any data currently stored in memory cache for this file is written out to disk if necessary.

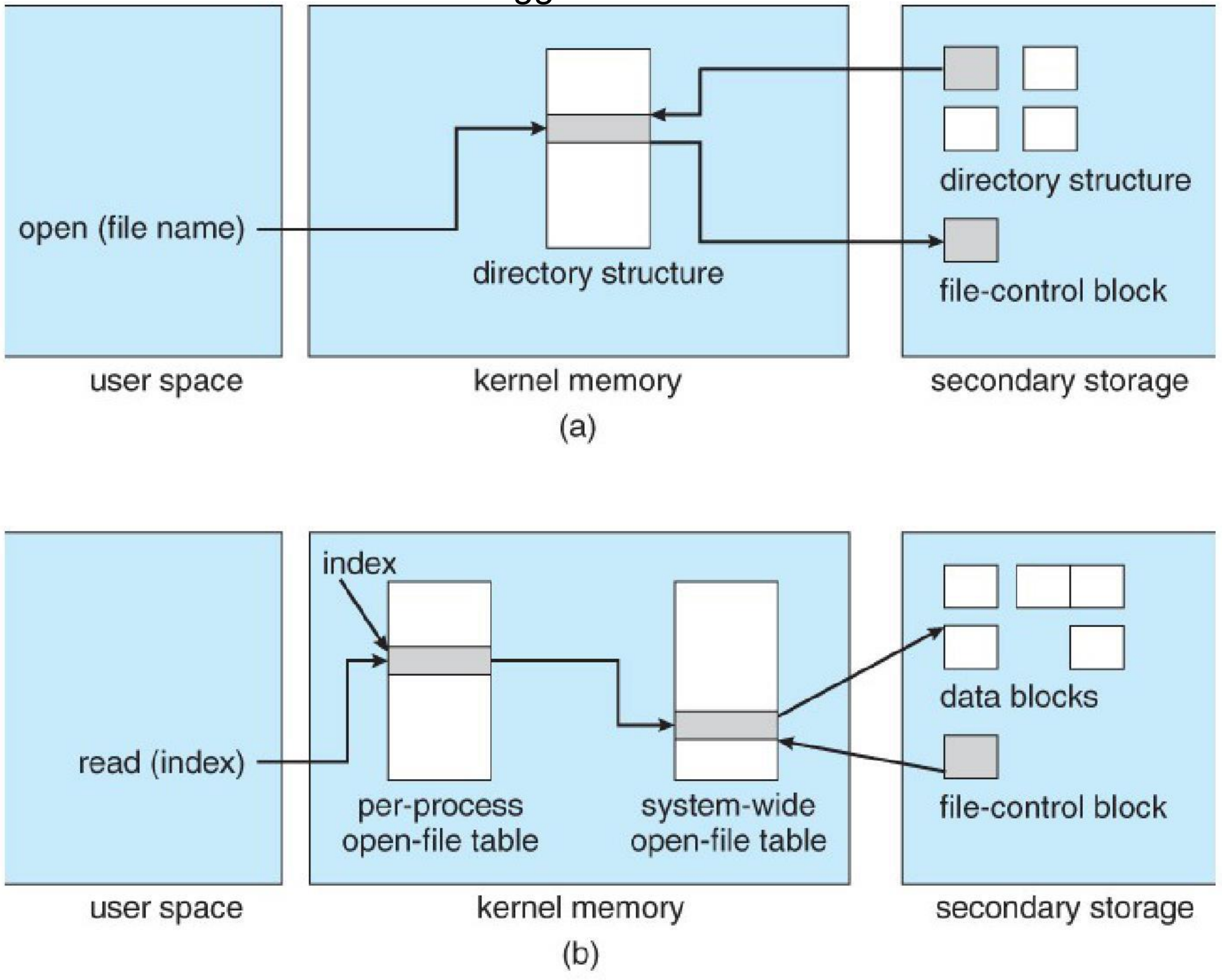


Figure 12.3 - In-memory file-system structures. (a) File open. (b) File read.

12.2.2 Partitions and Mounting

Physical disks are commonly divided into smaller units called partitions. They can also be combined into larger units, but that is most commonly done for RAID installations and is left for later chapters.

Partitions can either be used as raw devices (with no structure imposed upon them), or they can be formatted to hold a file system (i.e. populated with FCBs and initial directory structures as appropriate.) Raw partitions are generally used for swap space, and may also be used for certain programs such as databases that choose to manage their own disk storage system. Partitions containing filesystems can generally only be accessed using the file system structure by ordinary users, but can often be accessed as a raw device also by root.

The boot block is accessed as part of a raw partition, by the boot program prior to any operating system being loaded. The ***root partition*** contains the OS kernel and at least the key portions of the OS needed to complete the boot process. At boot time the root partition is mounted, and control is transferred from the boot program to the kernel found there. (Older systems required that the root partition lie completely within the first 1024 cylinders of the disk, because that was as far as the boot program could reach. Once the kernel had control, then it could access partitions beyond the 1024 cylinder boundary.)

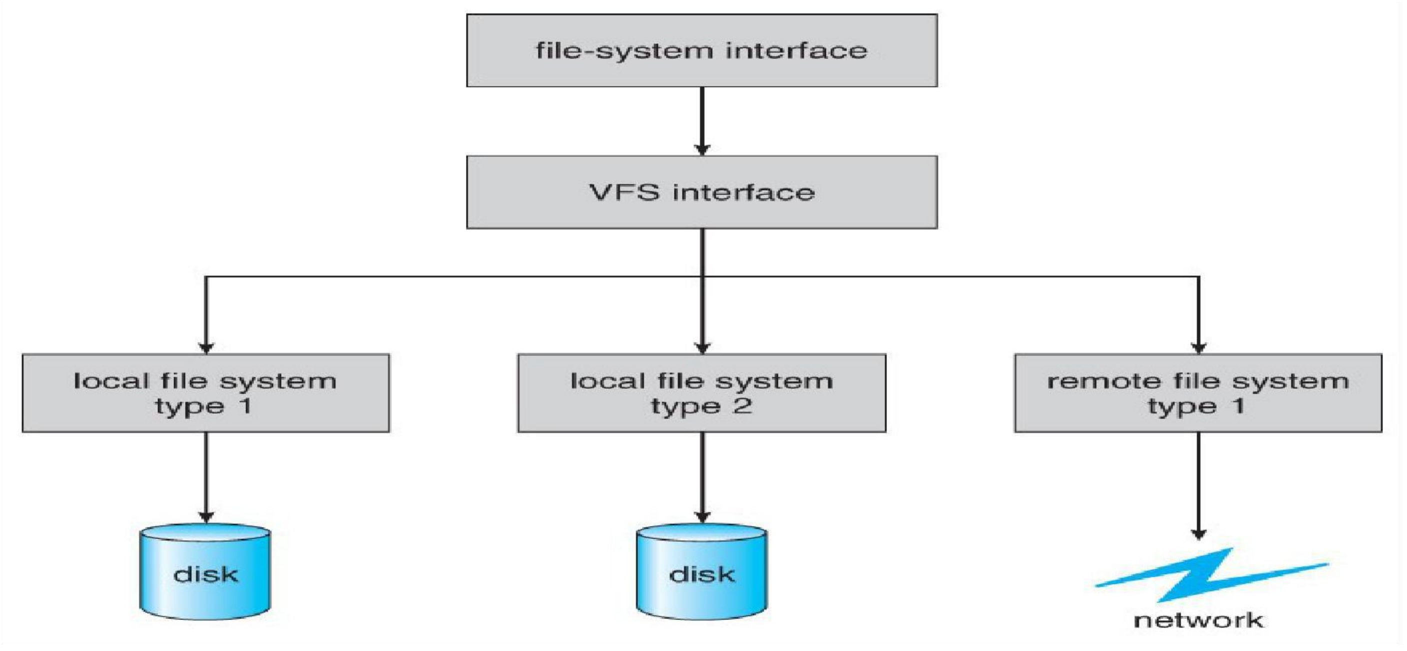
Virtual File Systems

Virtual File Systems, VFS, provide a common interface to multiple different file system types. In addition, it provides for a unique identifier (vnode) for files across the entire space, including across all file systems of different types. (UNIX inodes are unique only across a single file system, and certainly do not carry across networked file systems)

The VFS in Linux is based upon four key object types:

- The *inode* object, representing an individual file
- The *file* object, representing an open file.
- The *superblock* object, representing a file system.
- The *dentry* object, representing a directory entry.

Linux VFS provides a set of common functionalities for each file system, using function pointers accessed through a table. The same functionality is accessed through the same table position for all file system types, though the actual functions pointed to by the pointers may be file system-specific. Common operations provided include open( ), read( ), write( ), and mmap( ).



Schematic view of a virtual file system.

Directory implementation:

Directories need to be fast to search, insert, and delete, with a minimum of wasted disk space.

Linear List

A linear list is the simplest and easiest directory structure to set up, but it does have some drawbacks.

Finding a file (or verifying one does not already exist upon creation) requires a linear search.

Deletions can be done by moving all entries, flagging an entry as deleted, or by moving the last entry into the newly vacant position.

Sorting the list makes searches faster, at the expense of more complex insertions and deletions.

A linked list makes insertions and deletions into a sorted list easier, with overhead for the links.

More complex data structures, such as B-trees, could also be considered.

Hash Table

A hash table can also be used to speed up searches.

Hash tables are generally implemented *in addition to* a linear or other structure

Allocation Methods:

The allocation methods define how the files are stored in the disk blocks. There are three main disk space or file allocation methods.

Contiguous Allocation

Linked Allocation

Indexed Allocation

The main idea behind these methods is to provide:

Efficient disk space utilization.

Fast access to the file blocks.

All the **three methods** have their own advantages and disadvantages as discussed below:

Contiguous Allocation

Contiguous Allocation requires that all blocks of a file be kept together contiguously.

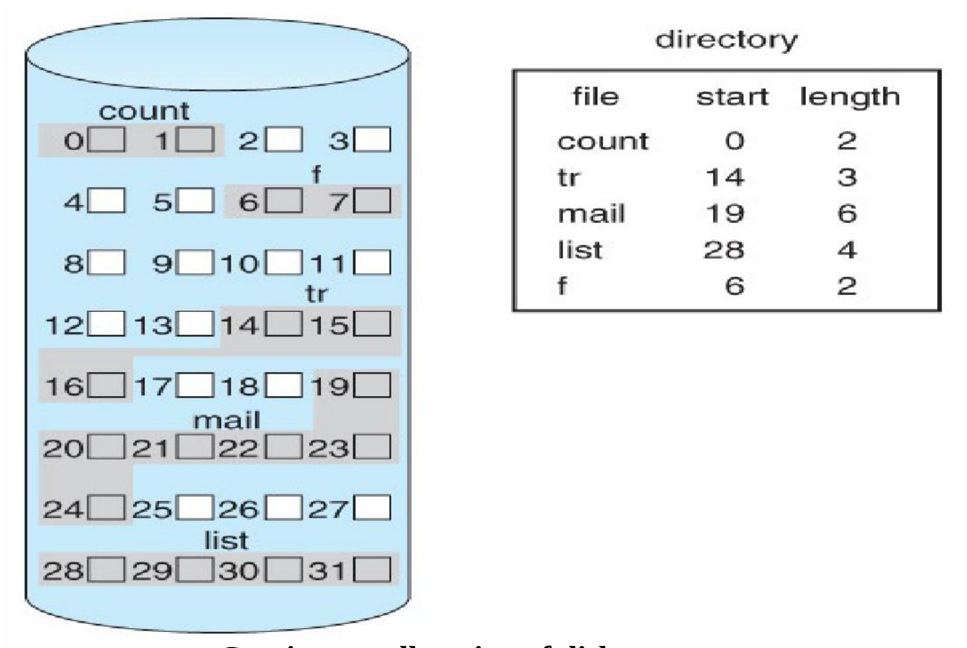
Performance is very fast, because reading successive blocks of the same file generally requires no movement of the disk heads, or at most one small step to the next adjacent cylinder.

Storage allocation involves the same issues discussed earlier for the allocation of contiguous blocks of memory (first fit, best fit, fragmentation problems, etc.) The distinction is that the high time penalty required for moving the disk heads from spot to spot may now justify the benefits of keeping files contiguously when possible. In this scheme, each file occupies a contiguous set of blocks on the disk. For example, if a file requires n blocks and is given a block b as the starting location, then the blocks assigned to the file will be: b, b+1, b+2,……b+n-1.This means that given the starting block address and the length of the file (in terms of blocks required), we can determine the blocks occupied by the file.

The directory entry for a file with contiguous allocation contains

Address of starting block
Length of the allocated portion.

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



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

Contiguous allocation of disk space.

Advantages:

Both the Sequential and Direct Accesses are supported by this. For direct access, the address of the kth block of the file which starts at block b can easily be obtained as (b+k).

This is extremely fast since the number of seeks are minimal because of contiguous allocation of file blocks.

Disadvantages:

This method suffers from both internal and external fragmentation. This makes it inefficient in terms of memory utilization.

Increasing file size is difficult because it depends on the availability of contiguous memory at a particular instance.

Linked Allocation

Disk files can be stored as linked lists, with the expense of the storage space consumed by each link. (E.g. a block may be 508 bytes instead of 512.)

Linked allocation involves no external fragmentation, does not require pre-known file sizes, and allows files to grow dynamically at any time.

Unfortunately linked allocation is only efficient for sequential access files, as random access requires starting at the beginning of the list for each new location access.

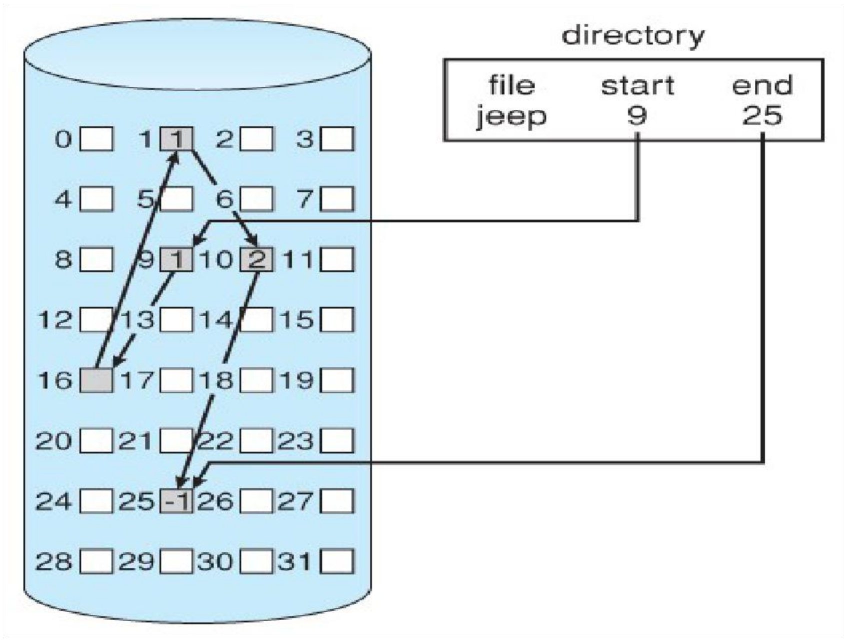
Allocating *clusters* of blocks reduces the space wasted by pointers, at the cost of internal fragmentation.

Another big problem with linked allocation is reliability if a pointer is lost or damaged. Doubly linked lists provide some protection, at the cost of additional overhead and wasted space.

In this scheme, each file is a linked list of disk blocks which need not be contiguous. The disk blocks can be scattered anywhere on the disk.

The directory entry contains a pointer to the starting and the ending file block. Each block contains a pointer to the next block occupied by the file.

The file 'jeep' in following image shows how the blocks are randomly distributed. The last block (25) contains -1 indicating a null pointer and does not point to any other block.



Linked allocation of disk space.

Advantages:

This is very flexible in terms of file size. File size can be increased easily since the system does not have to look for a contiguous chunk of memory.

This method does not suffer from external fragmentation. This makes it relatively better in terms of memory utilization.

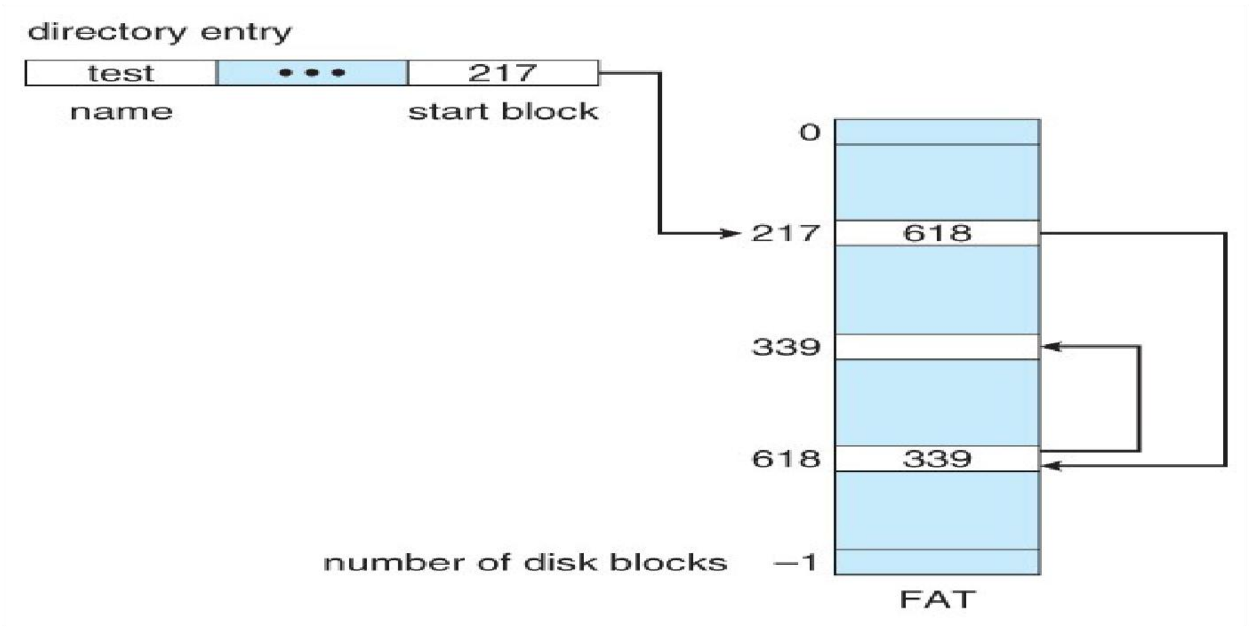
Disadvantages:

Because the file blocks are distributed randomly on the disk, a large number of seeks are needed to access every block individually. This makes linked allocation slower.

It does not support random or direct access. We cannot directly access the blocks of a file. A block k of a file can be accessed by traversing k blocks sequentially (sequential access) from the starting block of the file via block pointers.

Pointers required in the linked allocation incur some extra overhead.

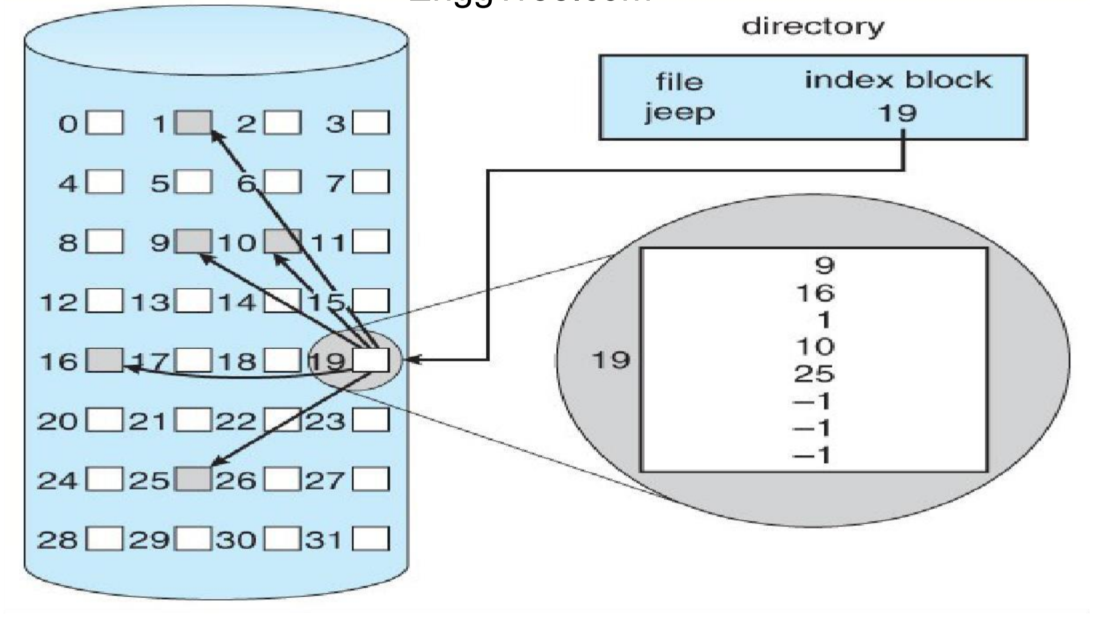
The ***File Allocation Table, FAT,*** used by DOS is a variation of linked allocation, where all the links are stored in a separate table at the beginning of the disk. The benefit of this approach is that the FAT table can be cached in memory, greatly improving random access speeds.



File-allocation table.

Indexed Allocation

Indexed Allocation combines all of the indexes for accessing each file into a common block (for that file), as opposed to spreading them all over the disk or storing them in a FAT table.



Indexed allocation of disk space.

Advantages:

This supports direct access to the blocks occupied by the file and therefore provides fast access to the file blocks.

It overcomes the problem of external fragmentation.

Disadvantages:

The pointer overhead for indexed allocation is greater than linked allocation.

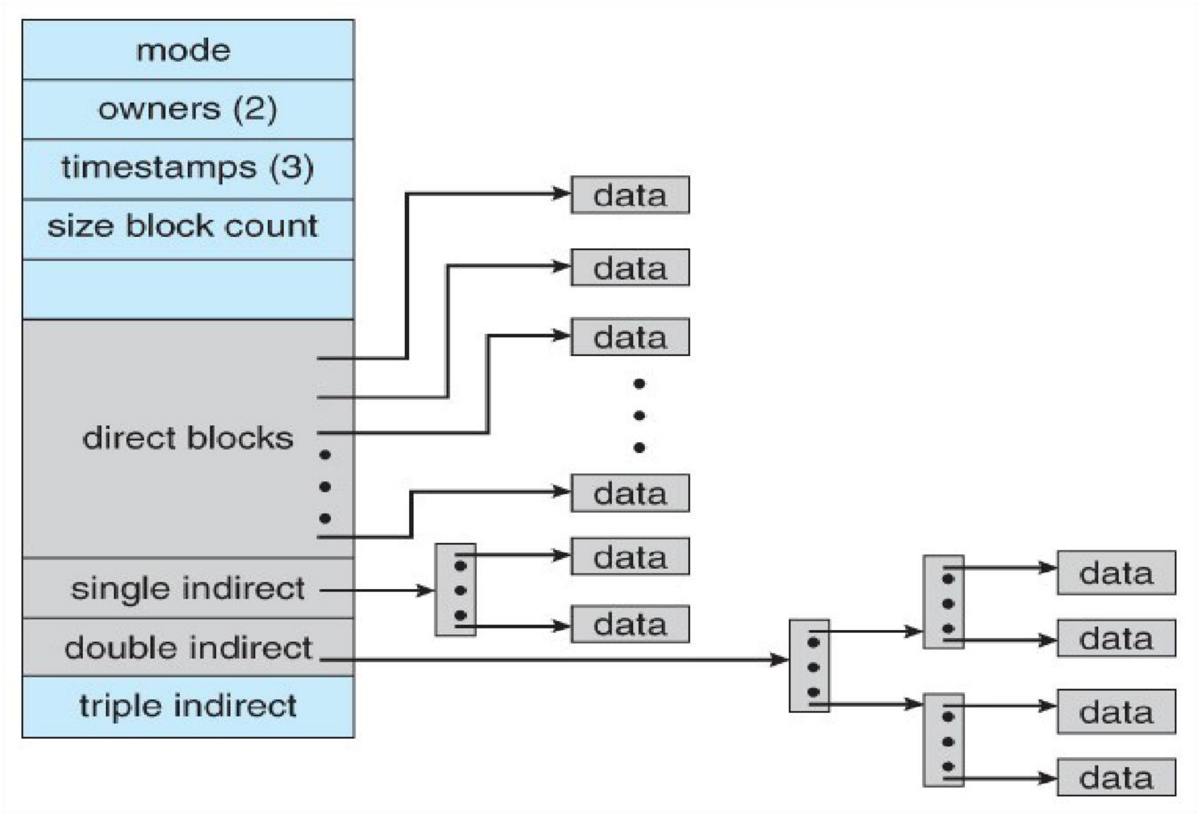
For very small files, say files that expand only 2-3 blocks, the indexed allocation would keep one entire block (index block) for the pointers which is inefficient in terms of memory utilization. However, in linked allocation we lose the space of only 1 pointer per block.

Some disk space is wasted (relative to linked lists or FAT tables) because an entire index block must be allocated for each file, regardless of how many data blocks the file contains. This leads to questions of how big the index block should be, and how it should be implemented. There are several approaches:

Linked Scheme - An index block is one disk block, which can be read and written in a single disk operation. The first index block contains some header information, the first N block addresses, and if necessary a pointer to additional linked index blocks.

Multi-Level Index - The first index block contains a set of pointers to secondary index blocks, which in turn contain pointers to the actual data blocks.

Combined Scheme - This is the scheme used in UNIX inodes, in which the first 12 or so data block pointers are stored directly in the inode, and then singly, doubly, and triply indirect pointers provide access to more data blocks as needed. (See below) The advantage of this scheme is that for small files (which many are), the data blocks are readily accessible (up to 48K with 4K block sizes); files up to about 4144K (using 4K blocks) are accessible with only a single indirect block (which can be cached), and huge files are still accessible using a relatively small number of disk accesses (larger in theory than can be addressed by a 32-bit address, which is why some systems have moved to 64-bit file pointers.)



The UNIX inode.

Performance

The optimal allocation method is different for sequential access files than for random access files, and is also different for small files than for large files.

Some systems support more than one allocation method, which may require specifying how the file is to be used (sequential or random access) at the time it is allocated. Such systems also provide conversion utilities.

Some systems have been known to use contiguous access for small files, and automatically switch to an indexed scheme when file sizes surpass a certain threshold.

And of course some systems adjust their allocation schemes (e.g. block sizes) to best match the characteristics of the hardware for optimum performance.

Free Space Management:

Another important aspect of disk management is keeping track of and allocating free space.

Bit Vector

One simple approach is to use a *bit vector*, in which each bit represents a disk block, set to 1 if free or 0 if allocated.

Fast algorithms exist for quickly finding contiguous blocks of a given size

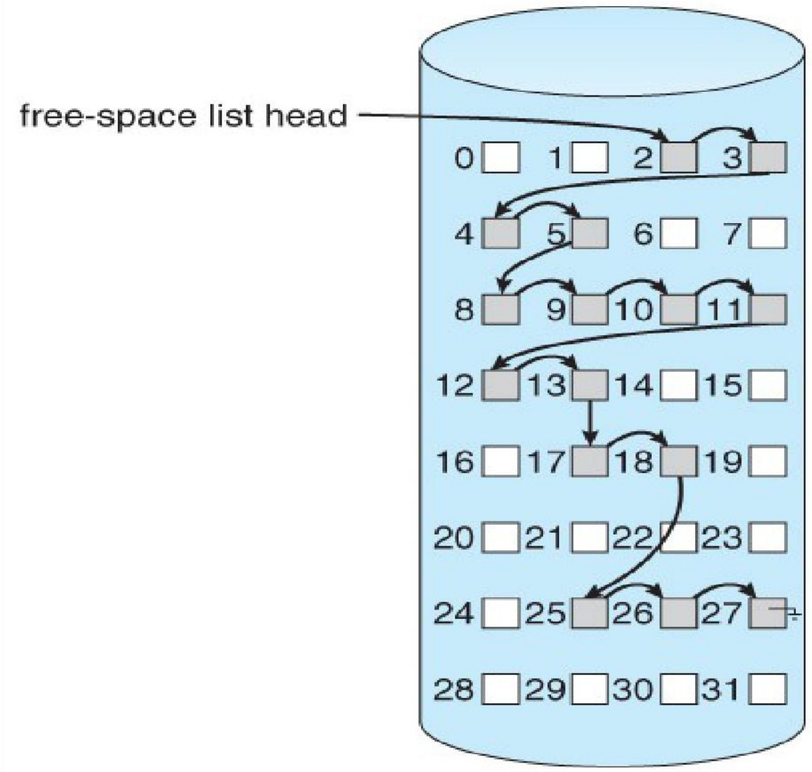
The down side is that a 40GB disk requires over 5MB just to store the bitmap. (For example.)

Linked List

A linked list can also be used to keep track of all free blocks.

Traversing the list and/or finding a contiguous block of a given size are not easy, but fortunately are not frequently needed operations. Generally the system just adds and removes single blocks from the beginning of the list.

The **FAT** table keeps track of the free list as just one more linked list on the table.



Linked free-space list on disk.

Grouping

A variation on linked list free lists is to use links of blocks of indices of free blocks. If a block holds up to N addresses, then the first block in the linked-list contains up to N-1 addresses of free blocks and a pointer to the next block of free addresses.

Counting

When there are multiple contiguous blocks of free space then the system can keep track of the starting address of the group and the number of contiguous free blocks. As long as the average length of a contiguous group of free blocks is greater than two this offers a savings in space needed for the free list. (Similar to compression techniques used for graphics images when a group of pixels all the same color is encountered.)

Space Maps (New)

Sun's ZFS file system was designed for HUGE numbers and sizes of files, directories, and even file systems.

The resulting data structures could be VERY inefficient if not implemented carefully. For example, freeing up a 1 GB file on a 1 TB file system could involve updating thousands of blocks of free list bit maps if the file was spread across the disk.

ZFS uses a combination of techniques, starting with dividing the disk up into (hundreds of) *metaslabs* of a manageable size, each having their own space map.

Free blocks are managed using the counting technique, but rather than write the information to a table, it is recorded in a log-structured transaction record. Adjacent free blocks are also coalesced into a larger single free block.

An in-memory space map is constructed using a balanced tree data structure, constructed from the log data.

The combination of the in-memory tree and the on-disk log provide for very fast and efficient management of these very large files and free blocks.

Efficiency and Performance:

Efficiency

UNIX pre-allocates inodes, which occupies space even before any files are created.

UNIX also distributes inodes across the disk, and tries to store data files near their inode, to reduce the distance of disk seeks between the inodes and the data.

Some systems use variable size clusters depending on the file size.

The more data that is stored in a directory (e.g. last access time), the more often the directory blocks have to be re-written.

As technology advances, addressing schemes have had to grow as well.

Sun's ZFS file system uses 128-bit pointers, which should theoretically never need to be expanded. (The mass required to store 2^128 bytes with atomic storage would be at least 272 trillion kilograms!) Kernel table sizes used to be fixed, and could only be changed by rebuilding the kernels. Modern tables are dynamically allocated, but that requires more complicated algorithms for accessing them.

Performance

Disk controllers generally include on-board caching. When a seek is requested, the heads are moved into place, and then an entire track is read, starting from whatever sector is currently under the heads (reducing latency.) The requested sector is returned and the unrequested portion of the track is cached in the disk's electronics.

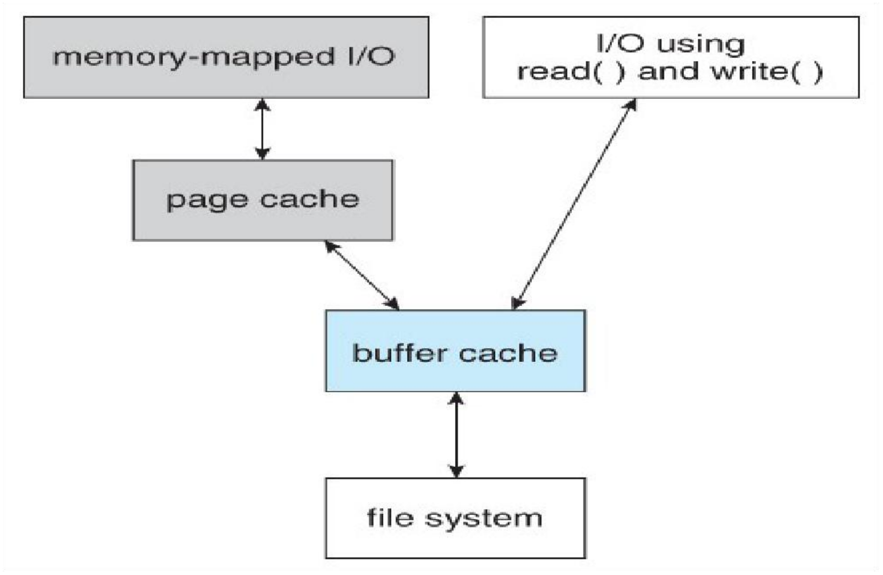
Some OSes cache disk blocks they expect to need again in a ***buffer cache.***

A ***page cache*** connected to the virtual memory system is actually more efficient as memory addresses do not need to be converted to disk block addresses and back again.

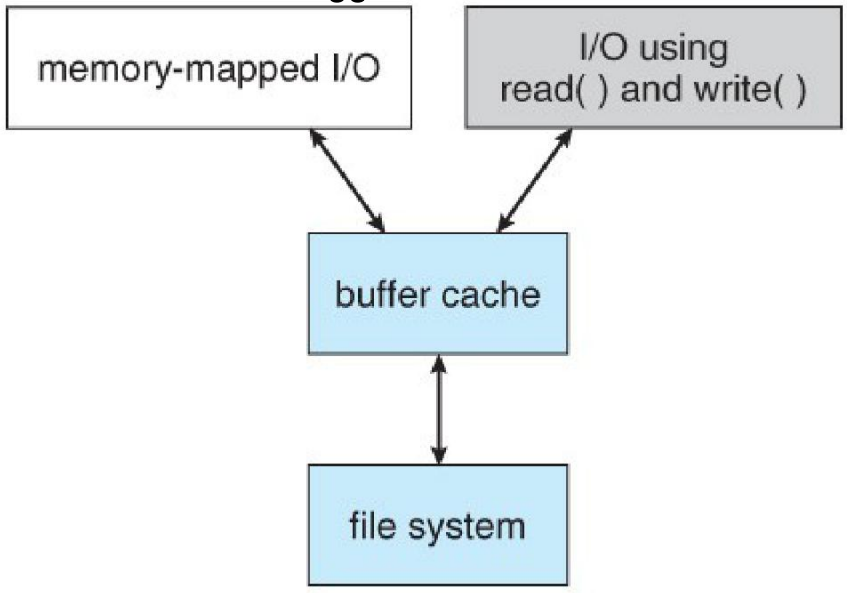
Some systems (Solaris, Linux, Windows 2000, NT, XP) use page caching for both process pages and file data in a ***unified virtual memory.***

Figures show the advantages of the *unified buffer cache* found in some versions of UNIX and Linux

- Data does not need to be stored twice, and problems of inconsistent buffer information are avoided.



I/O without a unified buffer cache.



I/O using a unified buffer cache.

Page replacement strategies can be complicated with a unified cache, as one needs to decide whether to replace process or file pages, and how many pages to guarantee to each category of pages. Solaris, for example, has gone through many variations, resulting in *priority paging* giving process pages priority over file I/O pages, and setting limits so that neither can knock the other completely out of memory.

Another issue affecting performance is the question of whether to implement *synchronous writes* or *asynchronous writes.* Synchronous writes occur in the order in which the disk subsystem receives them, without caching; Asynchronous writes are cached, allowing the disk subsystem to schedule writes in a more efficient order

The type of file access can also have an impact on optimal page replacement policies. For example, LRU is not necessarily a good policy for sequential access files. For these types of files progression normally goes in a forward direction only, and the most recently used page will not be needed again until after the file has been rewound and re-read from the beginning, (if it is ever needed at all.)

On the other hand, we can expect to need the next page in the file fairly soon. For this reason sequential access files often take advantage of two special policies:

- ***Free-behind*** frees up a page as soon as the next page in the file is requested, with the assumption that we are now done with the old page and won't need it again for a long time.
- ***Read-ahead*** reads the requested page and several subsequent pages at the same time, with the assumption that those pages will be needed in the near future. This is similar to the track caching that is already performed by the disk controller, except it saves the future latency of transferring data from the disk controller memory into motherboard main memory.

The caching system and asynchronous writes speed up disk writes considerably, because the disk subsystem can schedule physical writes to the disk to minimize head movement and disk seek times.

Reads, on the other hand, must be done more synchronously in spite of the caching system, with the result that disk writes can counter-intuitively be much faster on average than disk reads.

Recovery:

Consistency Checking

The storing of certain data structures (e.g. directories and inodes) in memory and the caching of disk operations can speed up performance, but what happens in the result of a system crash? All volatile memory structures are lost, and the information stored on the hard drive may be left in an inconsistent state.

A ***Consistency Checker*** is often run at boot time or mount time, particularly if a filesystem was not closed down properly. Some of the problems that these tools look for include:

Disk blocks allocated to files and also listed on the free list.
Disk blocks neither allocated to files nor on the free list.
Disk blocks allocated to more than one file.
The number of disk blocks allocated to a file inconsistent with the file's stated size. Properly allocated files / inodes which do not appear in any directory entry.

Log-Structured File Systems

Log-based transaction-oriented filesystems borrow techniques developed for databases, guaranteeing that any given transaction either completes successfully or can be rolled back to a safe state before the transaction commenced:

All metadata changes are written sequentially to a log.
A set of changes for performing a specific task (e.g. moving a file) is a *transaction*.
As changes are written to the log they are said to be *committed*, allowing the system to return to its work. In the meantime, the changes from the log are carried out on the actual filesystem, and a pointer keeps track of which changes in the log have been completed and which have not yet been completed.
When all changes corresponding to a particular transaction have been completed, that transaction can be safely removed from the log.
At any given time, the log will contain information pertaining to uncompleted transactions only, e.g. actions that were committed but for which the entire transaction has not yet been completed.
From the log, the remaining transactions can be completed,or if the transaction was aborted, then the partially completed changes can be undone.

Other Solutions (New)

Sun's ZFS and Network Appliance's WAFL file systems take a different approach to file system consistency.
No blocks of data are ever over-written in place. Rather the new data is written into fresh new blocks, and

after the transaction is complete, the metadata (data block pointers) is updated to point to the new blocks.
The old blocks can then be freed up for future use.
Alternatively, if the old blocks and old metadata are saved, then a *snapshot* of the system in its original state is preserved. This approach is taken by WAFL.
ZFS combines this with check-summing of all metadata and data blocks, and RAID, to ensure that no inconsistencies are possible, and therefore ZFS does not incorporate a consistency checker.

Backup and Restore

In order to recover lost data in the event of a disk crash, it is important to conduct backups regularly.

Files should be copied to some removable medium, such as magnetic tapes, CDs, DVDs, or external removable hard drives.

A full backup copies every file on a file system.

Incremental backups copy only files which have changed since some previous time.

A combination of full and incremental backups can offer a compromise between full recoverability, the number and size of backup tapes needed, and the number of tapes that need to be used to do a full restore. For example, one strategy might be:

- At the beginning of the month do a full backup.
- At the end of the first and again at the end of the second week, backup all files which have changed since the beginning of the month.
- At the end of the third week, backup all files that have changed since the end of the second week.
- Every day of the month not listed above, do an incremental backup of all files that have changed since the most recent of the weekly backups described above.

Backup tapes are often reused, particularly for daily backups, but there are limits to how many times the same tape can be used.

Every so often a full backup should be made that is kept "forever" and not overwritten.

Backup tapes should be tested, to ensure that they are readable!

For optimal security, backup tapes should be kept off-premises, so that a fire or burglary cannot destroy both the system and the backups. There are companies (e.g. Iron Mountain) that specialize in the secure off-site storage of critical backup information.

Keep your backup tapes secure - The easiest way for a thief to steal all your data is to simply pocket your backup tapes!

Storing important files on more than one computer can be an alternate though less reliable form of backup.

Note that incremental backups can also help users to get back a previous version of a file that they have since changed in some way.

Beware that backups can help forensic investigators recover e-mails and other files that users had though they had deleted!

I/O Systems:

Overview

Management of I/O devices is a very important part of the operating system - so important and so varied that entire I/O subsystems are devoted to its operation. (Consider the range of devices on a modern computer, from mice, keyboards, disk drives, display adapters, USB devices, network connections, audio I/O, printers, special devices for the handicapped, and many special-purpose peripherals.)

I/O Subsystems must contend with two (conflicting?) trends: (1) The gravitation towards standard interfaces for a wide range of devices, making it easier to add newly developed devices to existing systems, and (2) the development of entirely new types of devices, for which the existing standard interfaces are not always easy to apply.

Device drivers are modules that can be plugged into an OS to handle a particular device or category of similar devices.

I/O Hardware:

I/O devices can be roughly categorized as storage, communications, user-interface, and

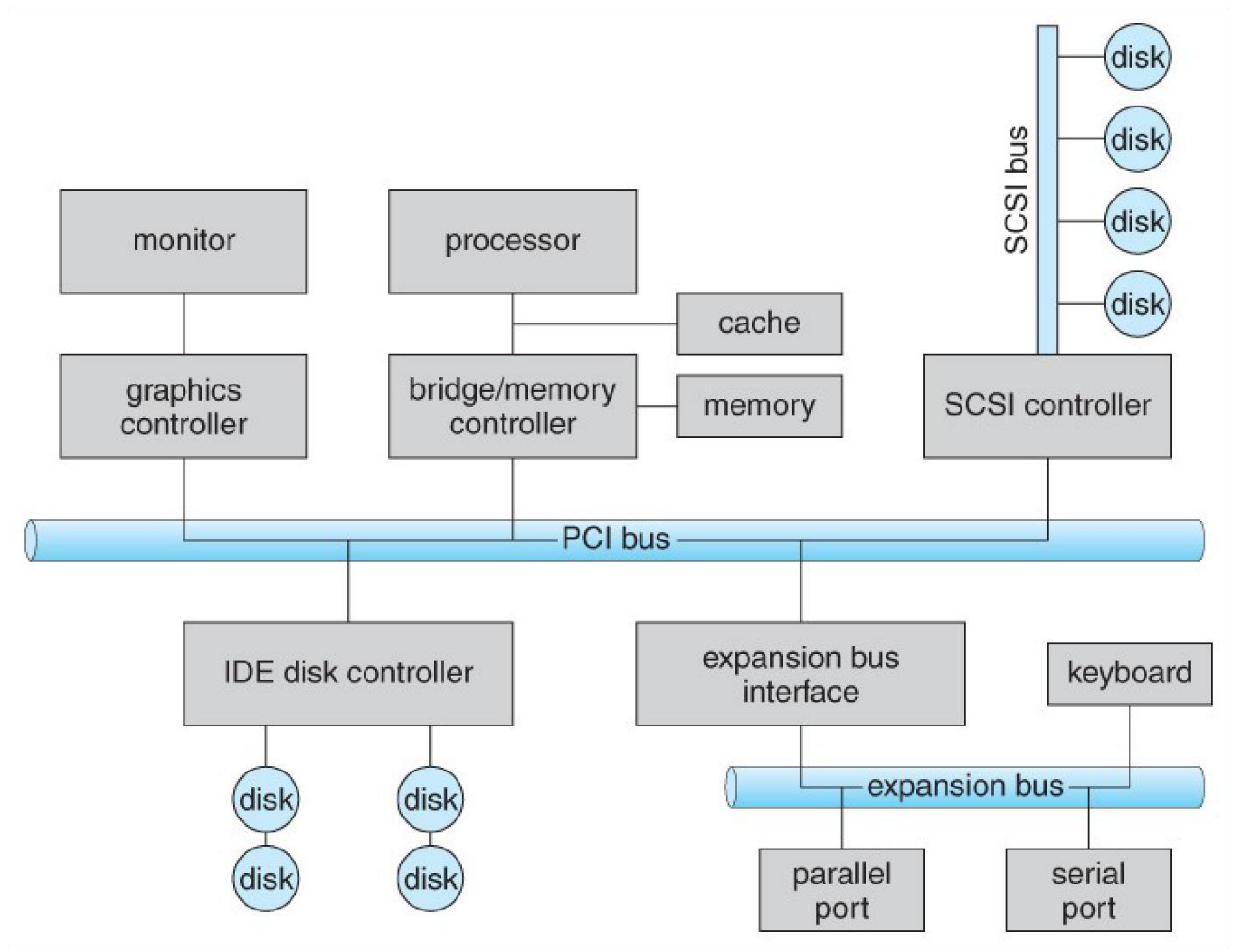
other Devices communicate with the computer via signals sent over wires or through the air.

Devices connect with the computer via ***ports***, e.g. a serial or parallel port. A

common set of wires connecting multiple devices is termed a ***bus.***

- Buses include rigid protocols for the types of messages that can be sent across the bus and the procedures for resolving contention issues.
- Figure below illustrates three of the four bus types commonly found in a modern PC:

1. The ***PCI bus*** connects high-speed high-bandwidth devices to the memory subsystem (and the CPU.)
2. The ***expansion bus*** connects slower low-bandwidth devices, which typically deliver data one character at a time (with buffering.)
3. The ***SCSI bus*** connects a number of SCSI devices to a common SCSI controller.
4. A ***daisy-chain bus,*** (not shown) is when a string of devices is connected to each other like beads on a chain, and only one of the devices is directly connected to the host.



A typical PC bus structure.

One way of communicating with devices is through ***registers*** associated with each port. Registers may be one to four bytes in size, and may typically include (a subset of) the following four:

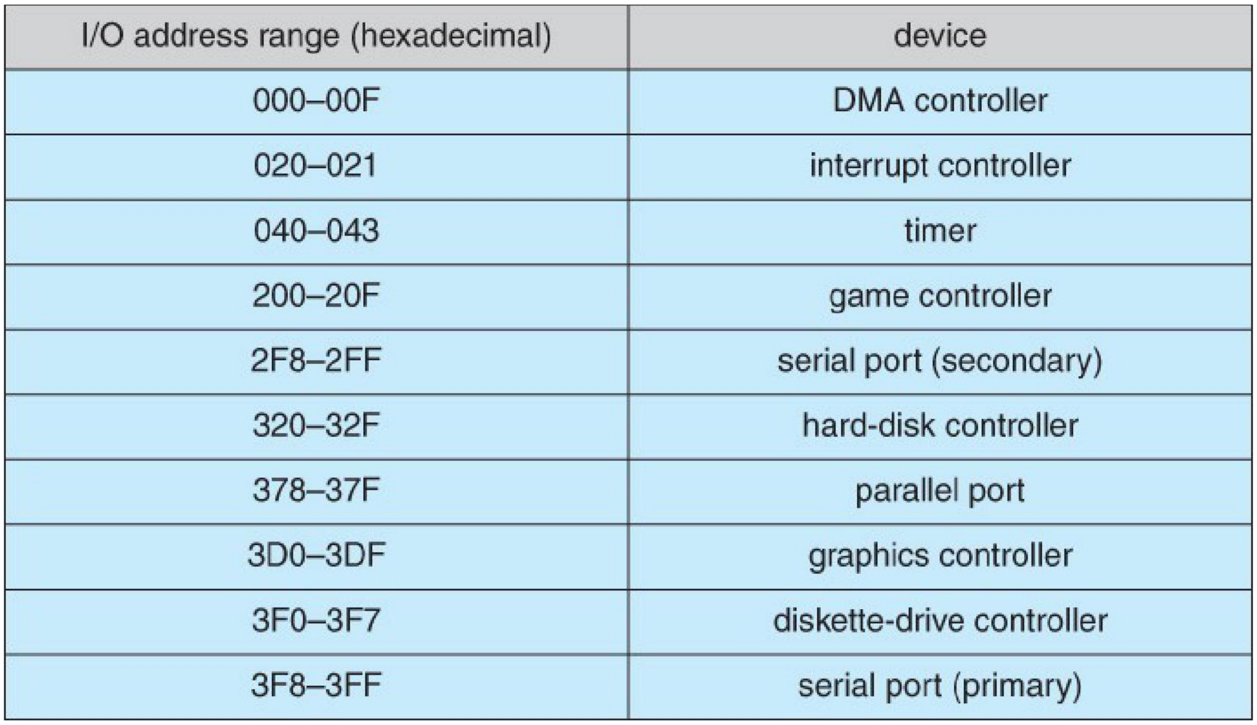
1. The ***data-in register*** is read by the host to get input from the device.

2. The ***data-out register*** is written by the host to send output.

3. The ***status register*** has bits read by the host to ascertain the status of the device, such as idle, ready for input, busy, error, transaction complete, etc.

4. The ***control register*** has bits written by the host to issue commands or to change settings of the device such as parity checking, word length, or full- versus half-duplex operation.

Figure shows some of the most common I/O port address ranges.



I/O address range (hexadecimal)	device
000–00F	DMA controller
020–021	interrupt controller
040–043	timer
200–20F	game controller
2F8–2FF	serial port (secondary)
320–32F	hard-disk controller
378–37F	parallel port
3D0–3DF	graphics controller
3F0–3F7	diskette-drive controller
3F8–3FF	serial port (primary)

Device I/O port locations on PCs (partial).

Another technique for communicating with devices is ***memory-mapped I/O.***

- In this case a certain portion of the processor's address space is mapped to the device, and communications occur by reading and writing directly to/from those memory areas.

- Memory-mapped I/O is suitable for devices which must move large quantities of data quickly, such as graphics cards.

- Memory-mapped I/O can be used either instead of or more often in combination with traditional registers. For example, graphics cards still use registers for control information such as setting the video mode.

- A potential problem exists with memory-mapped I/O, if a process is allowed to write directly to the address space used by a memory-mapped I/O device.
- (Note: Memory-mapped I/O is not the same thing as direct memory access, DMA.)

Polling

One simple means of device *handshaking* involves polling:

1. The host repeatedly checks the *busy bit* on the device until it becomes clear.
2. The host writes a byte of data into the data-out register, and sets the *write bit* in the command register (in either order.)
3. The host sets the *command ready bit* in the command register to notify the device of the pending command.
4. When the device controller sees the command-ready bit set, it first sets the busy bit.
5. Then the device controller reads the command register, sees the write bit set, reads the byte of data from the data-out register, and outputs the byte of data.
6. The device controller then clears the *error bit* in the status register, the command-ready bit, and finally clears the busy bit, signalling the completion of the operation.

Polling can be very fast and efficient, if both the device and the controller are fast and if there is significant data to transfer. It becomes inefficient, however, if the host must wait a long time in the busy loop waiting for the device, or if frequent checks need to be made for data that is infrequently there.

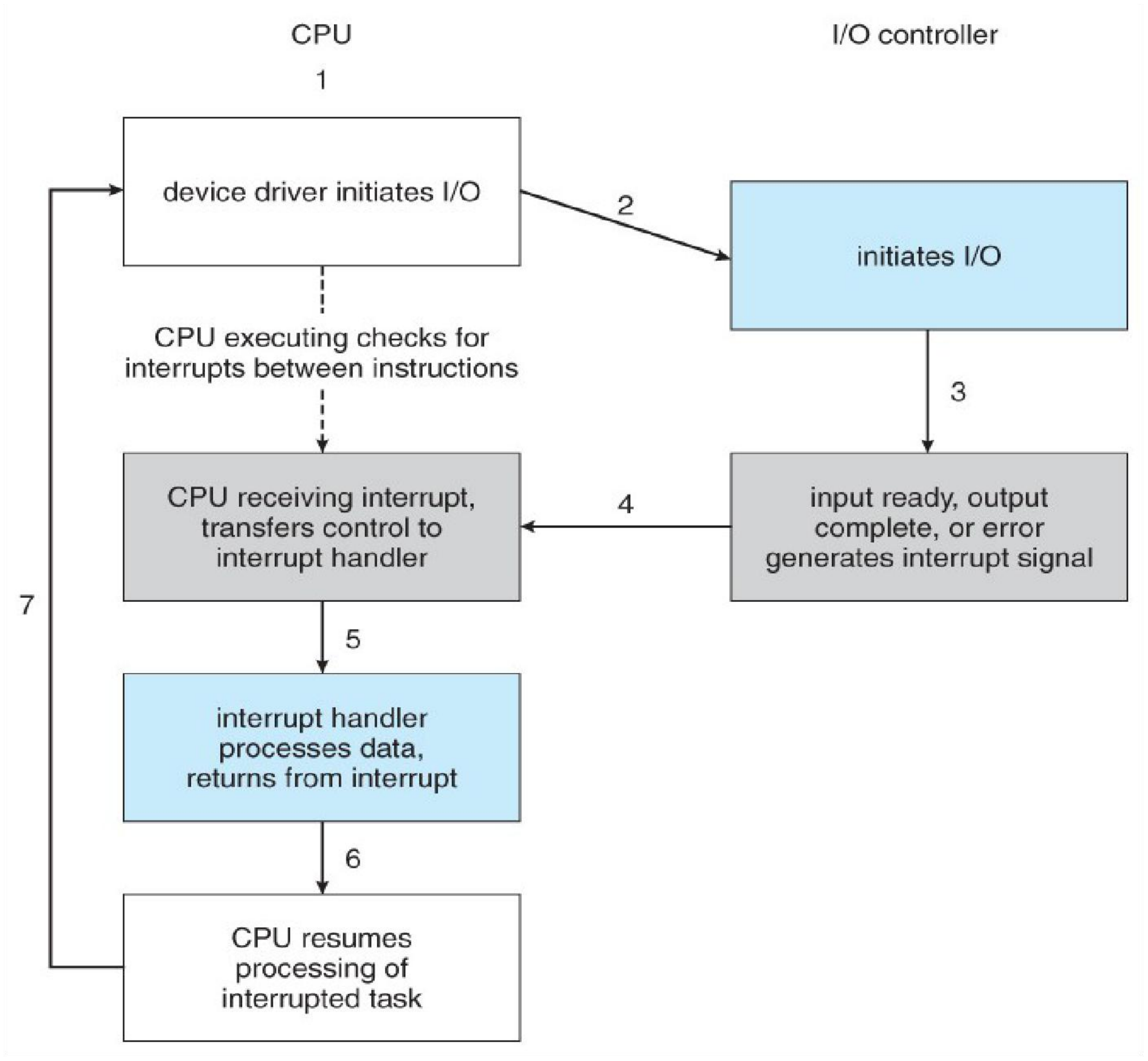
Interrupts

Interrupts allow devices to notify the CPU when they have data to transfer or when an operation is complete, allowing the CPU to perform other duties when no I/O transfers need its immediate attention.

The CPU has an *interrupt-request line* that is sensed after every instruction.

- A device's controller *raises* an interrupt by asserting a signal on the interrupt request line.
- The CPU then performs a state save, and transfers control to the *interrupt handler* routine at a fixed address in memory. (The CPU *catches* the interrupt and *dispatches* the interrupt handler.)
- The interrupt handler determines the cause of the interrupt, performs the necessary processing, performs a state restore, and executes a *return from interrupt* instruction to return control to the CPU. (The interrupt handler *clears* the interrupt by servicing the device.)

(Note that the state restored does not need to be the same state as the one that was saved when the interrupt went off. See below for an example involving time-slicing.)


CPU
I/O controller
1
device driver initiates I/O
2
initiates I/O
CPU executing checks for interrupts between instructions
3
CPU receiving interrupt, transfers control to interrupt handler
4
input ready, output complete, or error generates interrupt signal
7
5
interrupt handler processes data, returns from interrupt
6
CPU resumes processing of interrupted task

Interrupt-driven I/O procedure

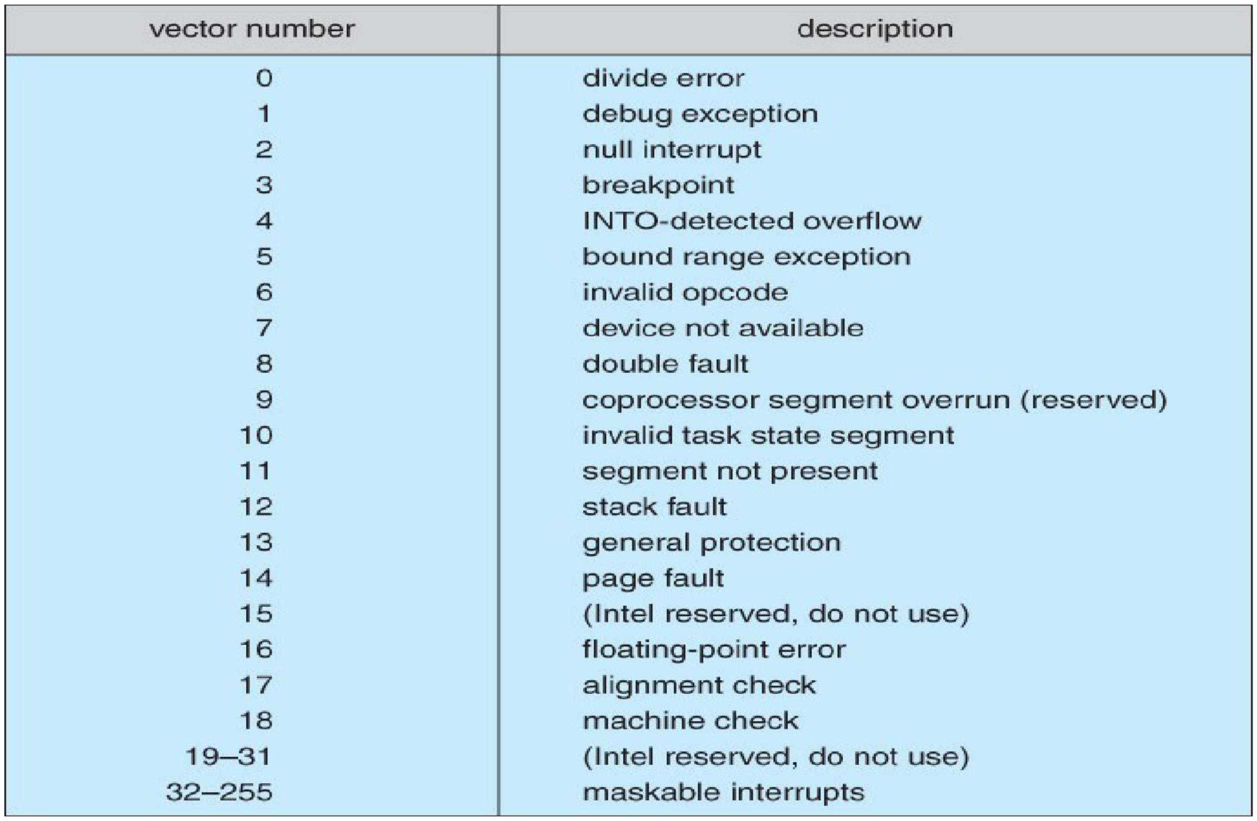
<u>Interrupt-driven I/O cycle.</u>

The above description is adequate for simple interrupt-driven I/O, but there are three needs in modern computing which complicate the picture:

1. The need to defer interrupt handling during critical processing,
2. The need to determine *which* interrupt handler to invoke, without having to poll all devices to see which one needs attention, and
3. The need for multi-level interrupts, so the system can differentiate between high- and low-priority interrupts for proper response.

These issues are handled in modern computer architectures with *interrupt-controller* hardware.

- Most CPUs now have two interrupt-request lines: One that is *non-maskable* for critical error conditions and one that is *maskable,* that the CPU can temporarily ignore during critical processing.
- The interrupt mechanism accepts an *address,* which is usually one of a small set of numbers for an offset into a table called the *interrupt vector.* This table (usually located at physical address zero ?) holds the addresses of routines prepared to process specific interrupts.
- The number of possible interrupt handlers still exceeds the range of defined interrupt numbers, so multiple handlers can be *interrupt chained*. Effectively the addresses held in the interrupt vectors are the head pointers for linked-lists of interrupt handlers.
- Figure shows the Intel Pentium interrupt vector. Interrupts 0 to 31 are non-maskable and reserved for serious hardware and other errors. Maskable interrupts, including normal device I/O interrupts begin at interrupt 32.Modern interrupt hardware also supports *interrupt priority levels,* allowing systems to mask off only lower-priority interrupts while servicing a high-priority interrupt, or conversely to allow a high-priority signal to interrupt the processing of a low-priority one.



vector number	description
0	divide error
1	debug exception
2	null interrupt
3	breakpoint
4	INTO-detected overflow
5	bound range exception
6	invalid opcode
7	device not available
8	double fault
9	coprocessor segment overrun (reserved)
10	invalid task state segment
11	segment not present
12	stack fault
13	general protection
14	page fault
15	(Intel reserved, do not use)
16	floating-point error
17	alignment check
18	machine check
19–31	(Intel reserved, do not use)
32–255	maskable interrupts

Intel Pentium processor event-vector table.

At boot time the system determines which devices are present, and loads the appropriate handler addresses into the interrupt table.

During operation, devices signal errors or the completion of commands via interrupts.

Exceptions, such as dividing by zero, invalid memory accesses, or attempts to access kernel mode instructions can be signalled via interrupts.

Time slicing and context switches can also be implemented using the interrupt mechanism.

- The scheduler sets a hardware timer before transferring control over to a user process.
- When the timer raises the interrupt request line, the CPU performs a state-save, and transfers control over to the proper interrupt handler, which in turn runs the scheduler.
- The scheduler does a state-restore of a *different* process before resetting the timer and issuing the return-from-interrupt instruction.

A similar example involves the paging system for virtual memory - A page fault causes an interrupt, which in turn issues an I/O request and a context switch as described above, moving the interrupted process into the wait queue and selecting a different process to run. When the I/O request has completed (i.e. when the requested page has been loaded up into physical memory), then the device interrupts, and the interrupt handler moves the process from the wait queue into the ready queue, (or depending on scheduling algorithms and policies, may go ahead and context switch it back onto the CPU.)

System calls are implemented via *software interrupts,* a.k.a. *traps.* When a (library) program needs work performed in kernel mode, it sets command information and possibly data addresses in certain registers, and then raises a software interrupt. (E.g. 21 hex in DOS.) The system does a state save and then calls on the proper interrupt handler to process the request in kernel mode. Software interrupts generally have low priority, as they are not as urgent as devices with limited buffering space.

Interrupts are also used to control kernel operations, and to schedule activities for optimal performance. For example, the completion of a disk read operation involves two interrupts:

- A high-priority interrupt acknowledges the device completion, and issues the next disk request so that the hardware does not sit idle.
- A lower-priority interrupt transfers the data from the kernel memory space to the user space, and then transfers the process from the waiting queue to the ready queue.

The Solaris OS uses a multi-threaded kernel and priority threads to assign different threads to different interrupt handlers. This allows for the "simultaneous" handling of multiple interrupts, and the assurance that high-priority interrupts will take precedence over low-priority ones and over user processes.

Direct Memory Access

For devices that transfer large quantities of data (such as disk controllers), it is wasteful to tie up the CPU transferring data in and out of registers one byte at a time.

Instead this work can be off-loaded to a special processor, known as the *Direct Memory Access, DMA, Controller.*

The host issues a command to the DMA controller, indicating the location where the data is located, the location where the data is to be transferred to, and the number of bytes of data to transfer. The DMA controller handles the data transfer, and then interrupts the CPU when the transfer is complete.

A simple DMA controller is a standard component in modern PCs, and many *bus-mastering* I/O cards contain their own DMA hardware.

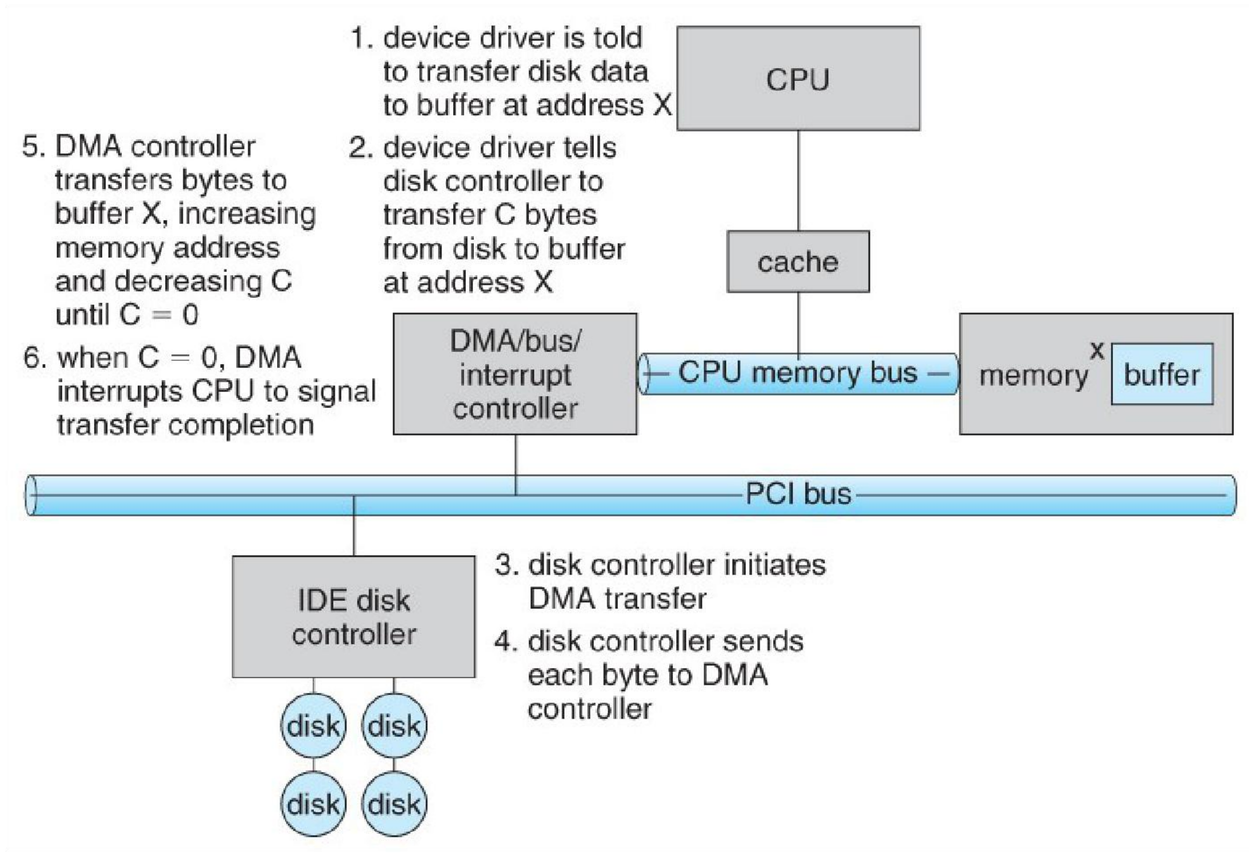
Handshaking between DMA controllers and their devices is accomplished through two wires called the DMA-request and DMA-acknowledge wires.

While the DMA transfer is going on the CPU does not have access to the PCI bus (including main memory), but it does have access to its internal registers and primary and secondary caches.

DMA can be done in terms of either physical addresses or virtual addresses that are mapped to physical addresses. The latter approach is known as ***Direct Virtual Memory Access, DVMA***, and allows direct data transfer from one memory-mapped device to another without using the main memory chips.

Direct DMA access by user processes can speed up operations, but is generally forbidden by modern systems for security and protection reasons. (i.e. DMA is a kernel-mode operation.)

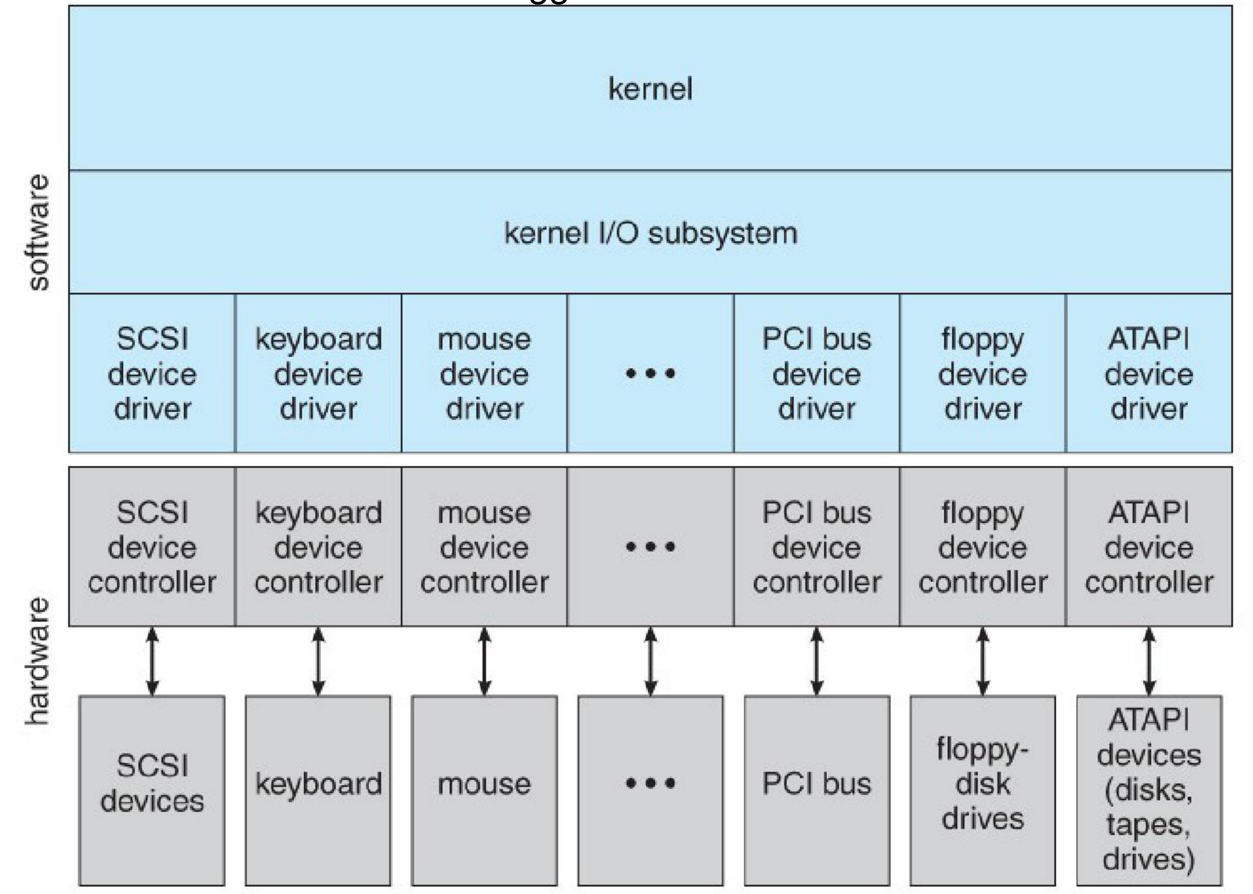
Below illustrates the **DMA** process.


1. device driver is told to transfer disk data to buffer at address X
CPU
5. DMA controller transfers bytes to buffer X, increasing memory address and decreasing C until C = 0
2. device driver tells disk controller to transfer C bytes from disk to buffer at address X
cache
6. when C = 0, DMA interrupts CPU to signal transfer completion
DMA/bus/ interrupt controller
CPU memory bus
memory
x
buffer
PCI bus
IDE disk controller
3. disk controller initiates DMA transfer
4. disk controller sends each byte to DMA controller
disk
disk
disk
disk

Steps in a DMA transfer

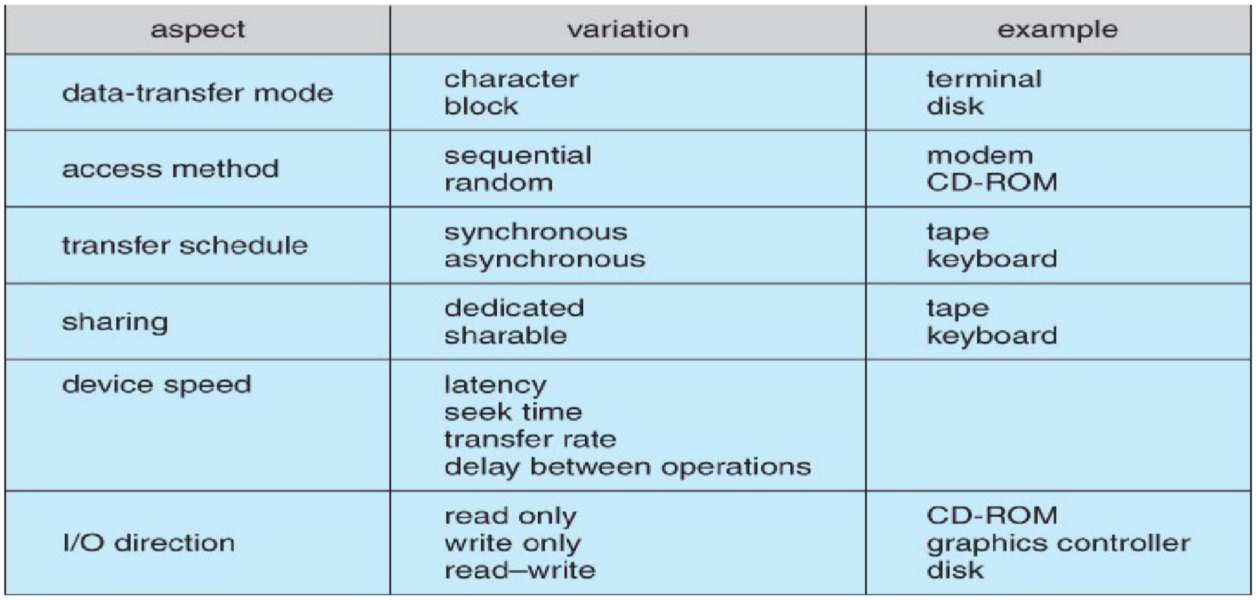
<u>Application I/O interface:</u>

User application access to a wide variety of different devices is accomplished through layering, and through encapsulating all of the device-specific code into ***device drivers***, while application layers are presented with a common interface for all (or at least large general categories of) devices.



A kernel I/O structure.

Devices differ on many different dimensions, as outlined



aspect	variation	example
data-transfer mode	character block	terminal disk
access method	sequential random	modem CD-ROM
transfer schedule	synchronous asynchronous	tape keyboard
sharing	dedicated sharable	tape keyboard
device speed	latency seek time transfer rate delay between operations	
I/O direction	read only write only read–write	CD-ROM graphics controller disk

Characteristics of I/O devices.

Block and Character Devices

Block devices are accessed a block at a time, and are indicated by a "b" as the first character in a long listing on UNIX systems. Operations supported include read( ), write( ), and seek( ).

- Accessing blocks on a hard drive directly (without going through the filesystem structure) is called ***raw I/O***, and can speed up certain operations by bypassing the buffering and locking normally conducted by the OS. (It then becomes the application's responsibility to manage those issues.)
- A new alternative is ***direct I/O,*** which uses the normal filesystem access, but which disables buffering and locking operations.

Character devices are accessed one byte at a time, and are indicated by a "c" in UNIX long listings. Supported operations include get( ) and put( ), with more advanced functionality such as reading an entire line supported by higher-level library routines.

Network Devices

Because network access is inherently different from local disk access, most systems provide a separate interface for network devices.

One common and popular interface is the ***socket*** interface, which acts like a cable or pipeline connecting two networked entities. Data can be put into the socket at one end, and read out sequentially at the other end. Sockets are normally full-duplex, allowing for bi-directional data transfer.

The select( ) system call allows servers (or other applications) to identify sockets which have data waiting, without having to poll all available sockets.

Clocks and Timers

Three types of time services are commonly needed in modern

systems: o Get the current time of day.

- Get the elapsed time (system or wall clock) since a previous event.
- Set a timer to trigger event X at time T.

Unfortunately time operations are not standard across all systems.

A ***programmable interrupt timer, PIT*** can be used to trigger operations and to measure elapsed time. It can be set to trigger an interrupt at a specific future time, or to trigger interrupts periodically on a regular basis.

- The scheduler uses a PIT to trigger interrupts for ending time slices.
- The disk system may use a PIT to schedule periodic maintenance cleanup, such as flushing buffers to disk.
- Networks use PIT to abort or repeat operations that are taking too long to complete. I.e. resending packets if an acknowledgement is not received before the timer goes off.
- More timers than actually exist can be simulated by maintaining an ordered list of timer events, and setting the physical timer to go off when the next scheduled event should occur.

On most systems the system clock is implemented by counting interrupts generated by the PIT. Unfortunately this is limited in its resolution to the interrupt frequency of the PIT, and may be subject to some drift over time. An alternate approach is to provide direct access to a high frequency hardware counter, which provides much higher resolution and accuracy, but which does not support interrupts.

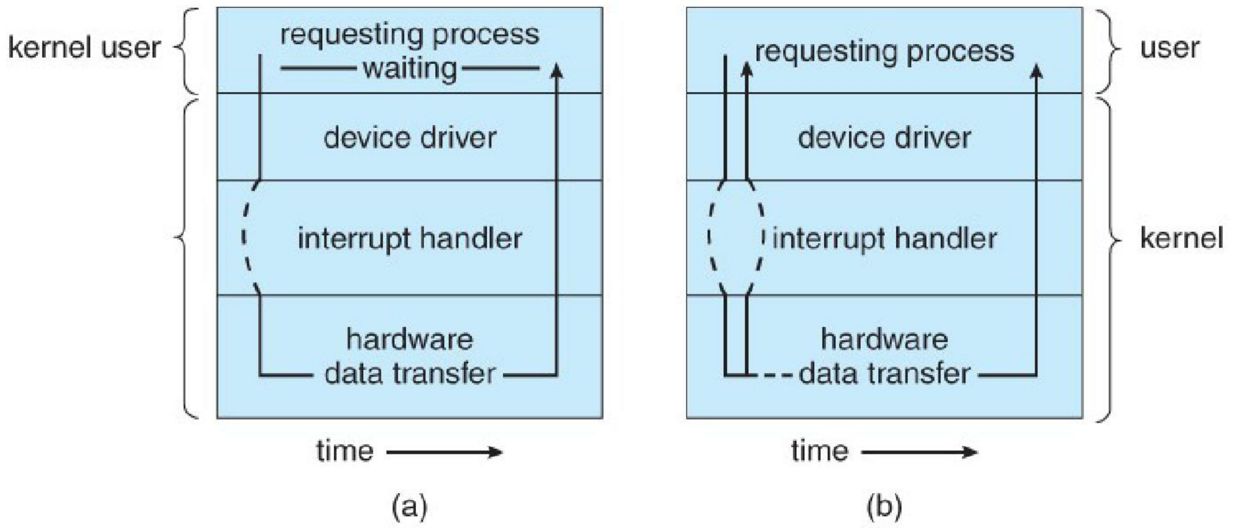
Blocking and Non-blocking I/O

With ***blocking I/O*** a process is moved to the wait queue when an I/O request is made, and moved back to the ready queue when the request completes, allowing other processes to run in the meantime.

With ***non-blocking I/O*** the I/O request returns immediately, whether the requested I/O operation has (completely) occurred or not. This allows the process to check for available data without getting hung completely if it is not there.

One approach for programmers to implement non-blocking I/O is to have a multi-threaded application, in which one thread makes blocking I/O calls (say to read a keyboard or mouse), while other threads continue to update the screen or perform other tasks.

A subtle variation of the non-blocking I/O is the ***asynchronous I/O,*** in which the I/O request returns immediately allowing the process to continue on with other tasks, and then the process is notified (via changing a process variable, or a software interrupt, or a callback function) when the I/O operation has completed and the data is available for use. (The regular non-blocking I/O returns immediately with whatever results are available, but does not complete the operation and notify the process later.)



Two I/O methods: (a) synchronous and (b) asynchronous.

Kernel I/O subsystem:

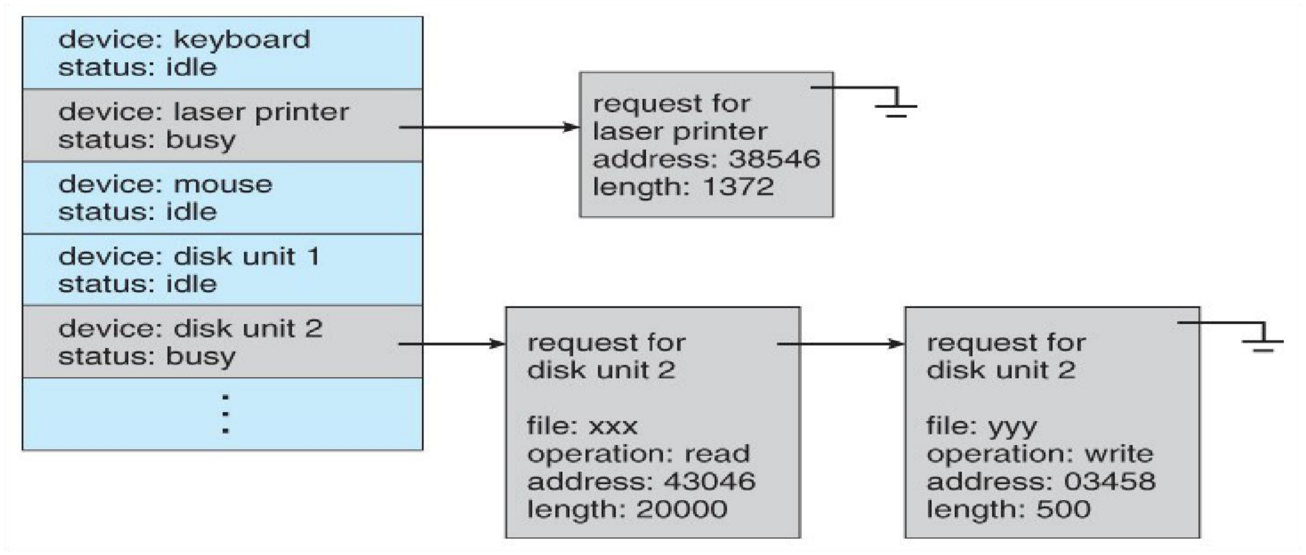
I/O Scheduling

Scheduling I/O requests can greatly improve overall efficiency. Priorities can also play a part in request scheduling.

The classic example is the scheduling of disk accesses, as discussed in detail.

Buffering and caching can also help, and can allow for more flexible scheduling options.

On systems with many devices, separate request queues are often kept for each device:



Device-status table.

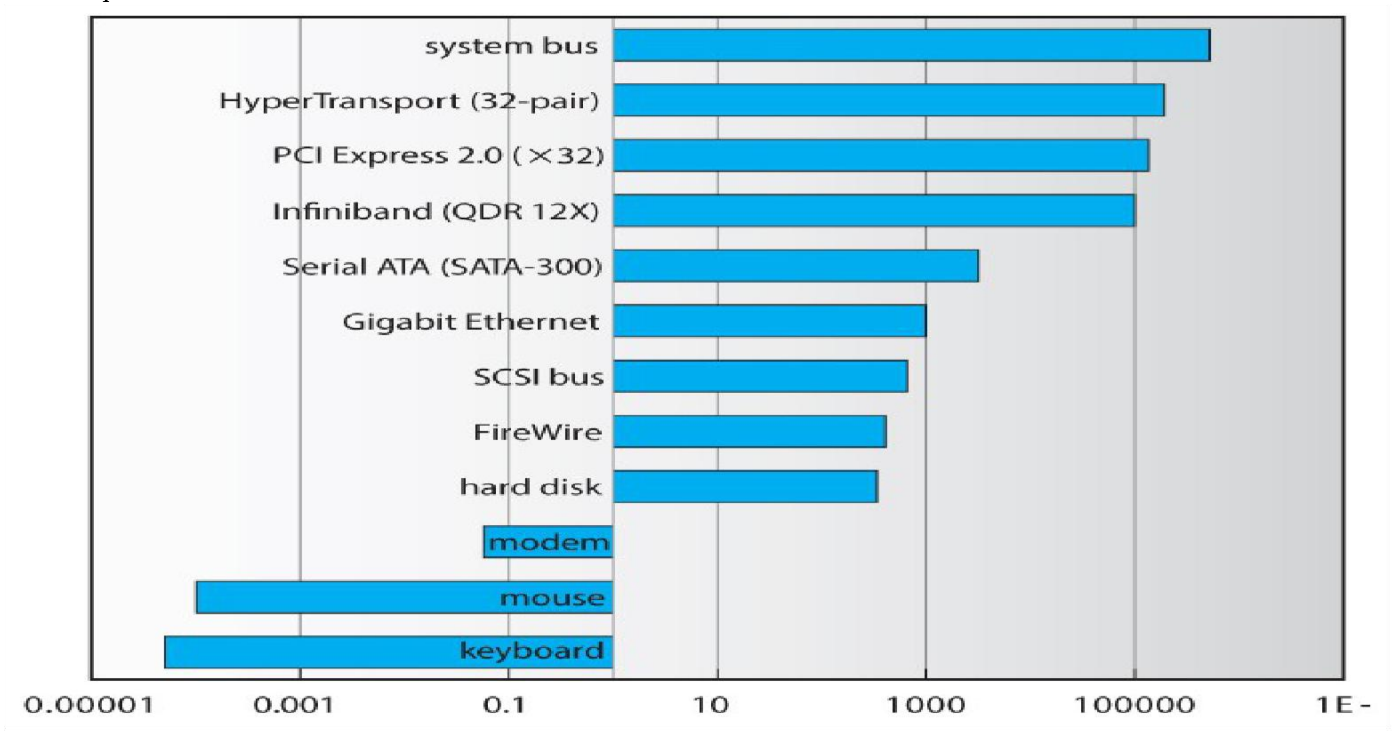
Buffering

Buffering of I/O is performed for (at least) 3 major reasons:

Speed differences between two devices. (See Figure below.) A slow device may write data into a buffer, and when the buffer is full, the entire buffer is sent to the fast device all at once. So that the slow device still has somewhere to write while this is going on, a second buffer is used, and the two buffers alternate as each becomes full. This is known as *double buffering*. (Double buffering is often used in (animated) graphics, so that one screen image can be generated in a buffer while the other (completed) buffer is displayed on the screen. This prevents the user from ever seeing any half-finished screen images.)

Data transfer size differences. Buffers are used in particular in networking systems to break messages up into smaller packets for transfer, and then for re-assembly at the receiving side.

To support ***copy semantics***. For example, when an application makes a request for a disk write, the data is copied from the user's memory area into a kernel buffer. Now the application can change their copy of the data, but the data which eventually gets written out to disk is the version of the data at the time the write request was made.



Sun Enterprise 6000 device-transfer rates (logarithmic).

Caching

Caching involves keeping a copy of data in a faster-access location than where the data is normally stored.

Buffering and caching are very similar, except that a buffer may hold the only copy of a given data item, whereas a cache is just a duplicate copy of some other data stored elsewhere.

Buffering and caching go hand-in-hand, and often the same storage space may be used for both purposes.

For example, after a buffer is written to disk, then the copy in memory can be used as a cached copy, (until that buffer is needed for other purposes)

Spooling and Device Reservation

A *spool (Simultaneous Peripheral Operations On-Line)* buffers data for (peripheral) devices such as printers that cannot support interleaved data streams.

If multiple processes want to print at the same time, they each send their print data to files stored in the spool directory. When each file is closed, then the application sees that print job as complete, and the print scheduler sends each file to the appropriate printer one at a time.

Support is provided for viewing the spool queues, removing jobs from the queues, moving jobs from one queue to another queue, and in some cases changing the priorities of jobs in the queues.

Spool queues can be general (any laser printer) or specific (printer number 42.)

Error Handling

I/O requests can fail for many reasons, either transient (buffers overflow) or permanent (disk crash). I/O requests usually return an error bit (or more) indicating the problem. UNIX systems also set the global variable *errno* to one of a hundred or so well-defined values to indicate the specific error that has occurred. (Seeerrno.h for a complete listing, or man errno.)

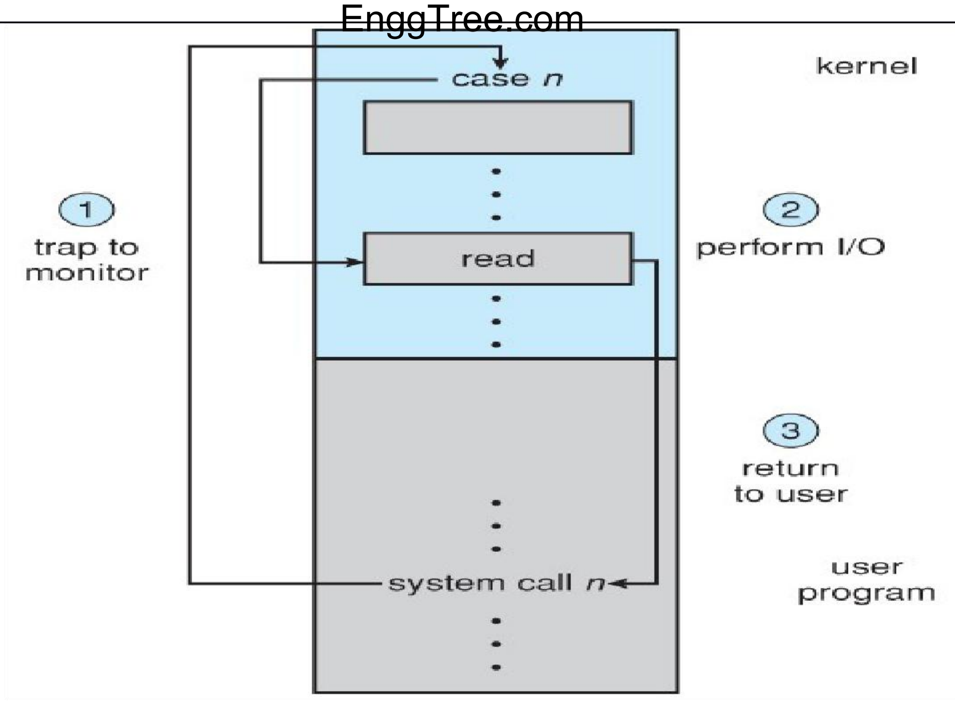
Some devices, such as SCSI devices, are capable of providing much more detailed information about errors, and even keep an on-board error log that can be requested by the host.

I/O Protection

The I/O system must protect against either accidental or deliberate erroneous I/O.

User applications are not allowed to perform I/O in user mode - All I/O requests are handled through system calls that must be performed in kernel mode.

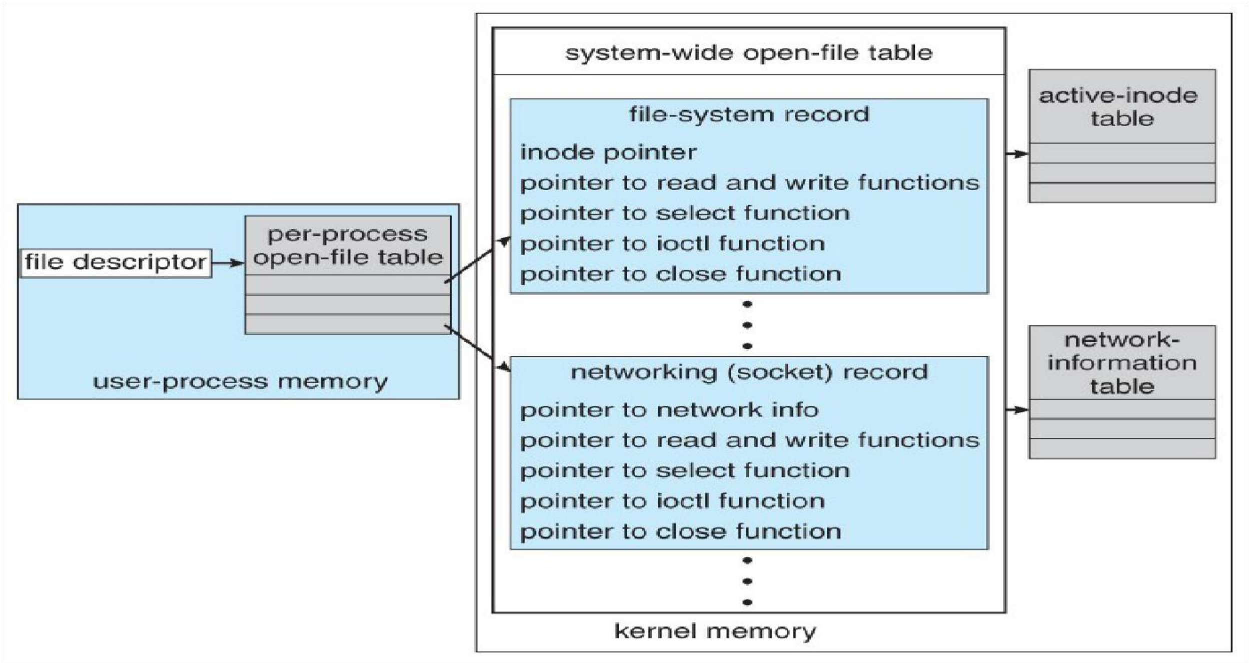
Memory mapped areas and I/O ports must be protected by the memory management system, but access to these areas cannot be totally denied to user programs. (Video games and some other applications need to be able to write directly to video memory for optimal performance for example.) Instead the memory protection system restricts access so that only one process at a time can access particular parts of memory, such as the portion of the screen memory corresponding to a particular window.



Use of a system call to perform I/O.

Kernel Data Structures

The kernel maintains a number of important data structures pertaining to the I/O system, such as the open file table. These structures are object-oriented, and flexible to allow access to a wide variety of I/O devices through a common interface. (See Figure below.)Windows NT carries the object-orientation one step further, implementing I/O as a message-passing system from the source through various intermediaries to the device.



system-wide open-file table
file-system record
inode pointer
pointer to read and write functions
pointer to select function
pointer to ioctl function
pointer to close function
active-inode table
file descriptor
per-process open-file table
user-process memory
networking (socket) record
pointer to network info
pointer to read and write functions
pointer to select function
pointer to ioctl function
pointer to close function
network-information table
kernel memory

UNIX I/O kernel structure.

Streams

The ***streams*** mechanism in UNIX provides a bi-directional pipeline between a user process and a device driver, onto which additional modules can be added.

The user process interacts with the ***stream head.***

The device driver interacts with the ***device end.***

Zero or more ***stream modules*** can be pushed onto the stream, using ioctl( ). These modules may filter and/or modify the data as it passes through the stream.

Each module has a ***read queue*** and a ***write queue.***

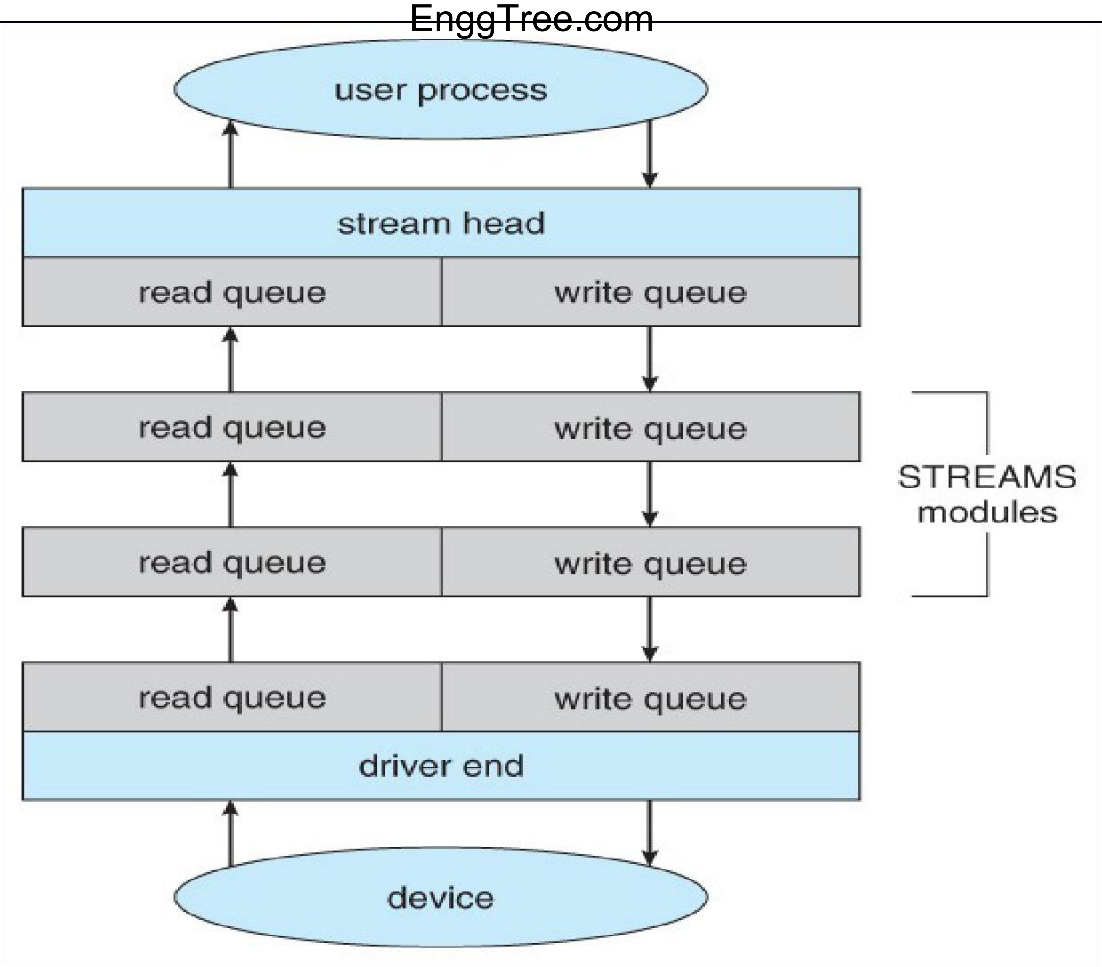
Flow control can be optionally supported, in which case each module will buffer data until the adjacent module is ready to receive it. Without flow control, data is passed along as soon as it is ready.

User processes communicate with the stream head using either read( ) and write( ) (or putmsg( ) and getmsg( ) for message passing.)

Streams I/O is asynchronous (non-blocking), except for the interface between the user process and the stream head.

The device driver **must** respond to interrupts from its device - If the adjacent module is not prepared to accept data and the device driver's buffers are all full, and then data is typically dropped.

Streams are widely used in UNIX, and are the preferred approach for device drivers. For example, UNIX implements sockets using streams.



The SREAMS structure.

Performance:

The I/O system is a major factor in overall system performance, and can place heavy loads on other major components of the system (interrupt handling, process switching, memory access, bus contention, and CPU load for device drivers just to name a few.)
Interrupt handling can be relatively expensive (slow), which causes programmed I/O to be faster than interrupt-driven I/O when the time spent busy waiting is not excessive.
Network traffic can also put a heavy load on the system. Consider for example the sequence of events that occur when a single character is typed in a telnet session, as shown in figure (And the fact that a similar set of events must happen in reverse to echo back the character that was typed) Sun uses in-kernel threads for the telnet daemon, increasing the supportable number of simultaneous telnet sessions from the hundreds to the thousands.

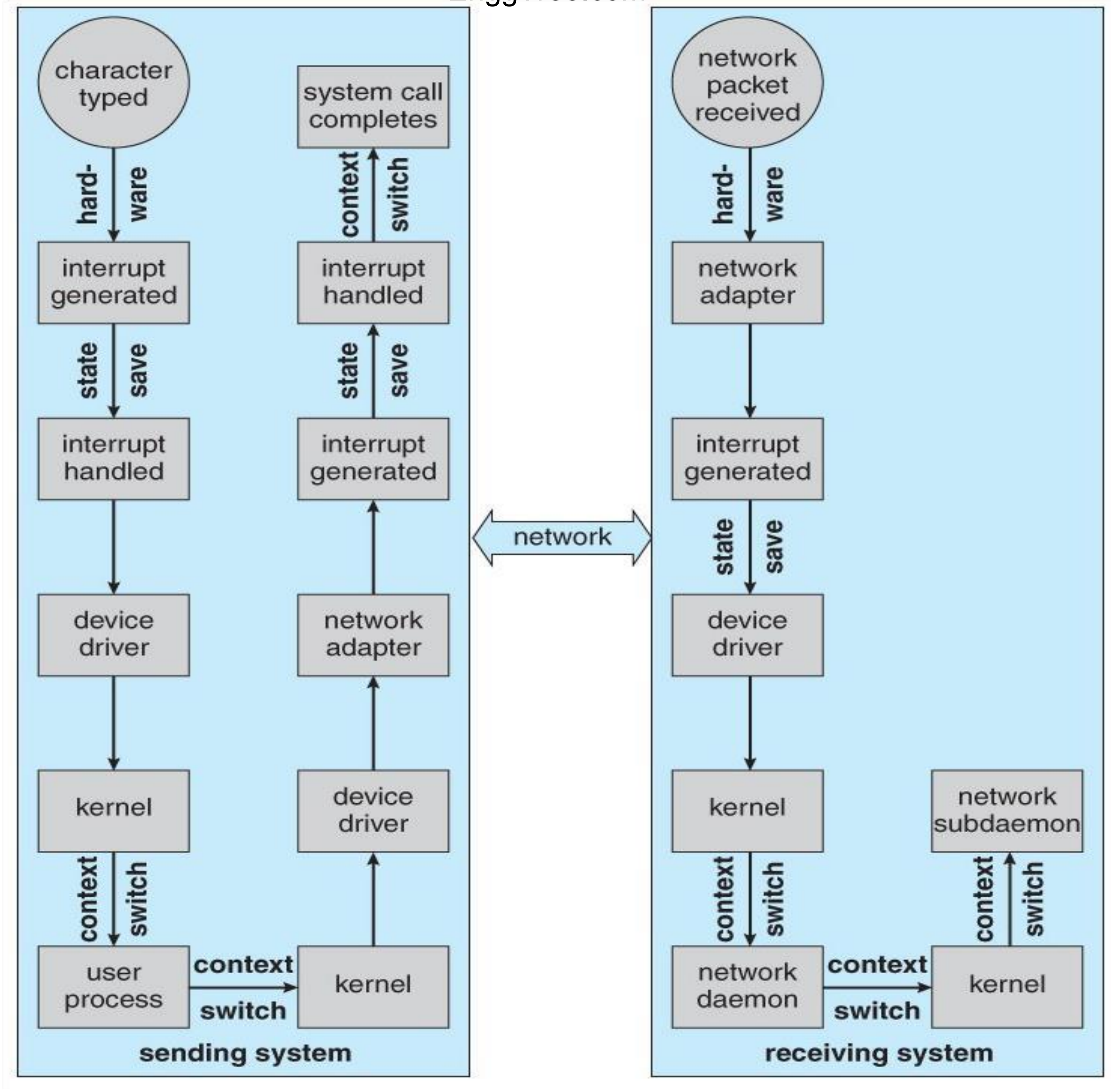


Figure Intercomputer communications.

Other systems use ***front-end processors*** to off-load some of the work of I/O processing from the CPU. For example a ***terminal concentrator*** can multiplex with hundreds of terminals on a single port on a large computer.

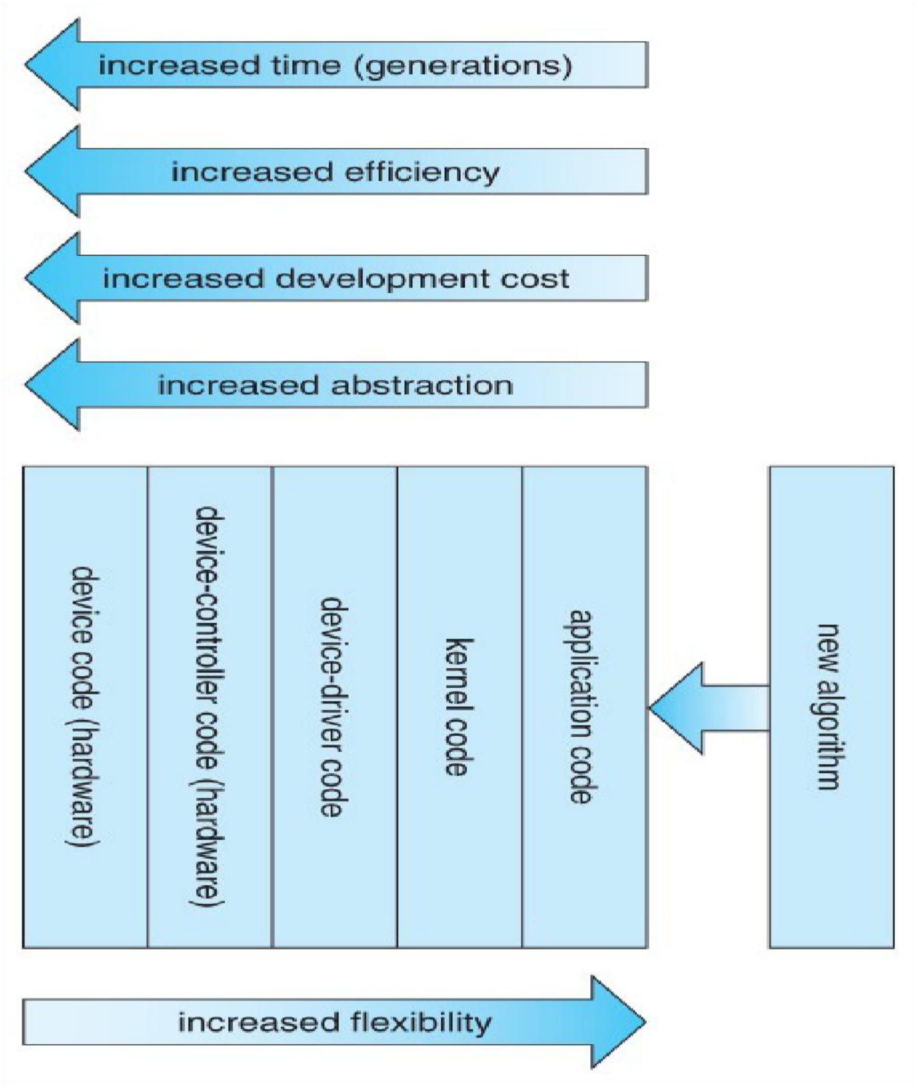
Several principles can be employed to increase the overall efficiency of I/O processing:

1. Reduce the number of context switches.

2. Reduce the number of times data must be copied.

3. Reduce interrupt frequency, using large transfers, buffering, and polling where appropriate.
4. Increase concurrency using DMA.
5. Move processing primitives into hardware, allowing their operation to be concurrent with CPU and bus operations.
6. Balance CPU, memory, bus, and I/O operations, so a bottleneck in one does not idle all the others.

The development of new I/O algorithms often follows a progression from application level code to on-board hardware implementation, as shown in Figure 13.16. Lower-level implementations are faster and more efficient, but higher-level ones are more flexible and easier to modify. Hardware-level functionality may also be harder for higher-level authorities (e.g. the kernel) to control.



Device functionality progression.

UNIT 5

Virtual Machine abstracts the hardware of our personal computer such as CPU, disk drives, memory, NIC (Network Interface Card) etc, into many different execution environments as per our requirements, hence giving us a feel that each execution environment is a single computer. For example, VirtualBox.

When we run different processes on an operating system, it creates an illusion that each process is running on a different processor having its own virtual memory, with the help of CPU scheduling and virtual-memory techniques. There are additional features of a process that cannot be provided by the hardware alone like system calls and a file system. The virtual machine approach does not provide these additional functionalities but it only provides an interface that is same as basic hardware. Each process is provided with a virtual copy of the underlying computer system.

We can create a virtual machine for several reasons, all of which are fundamentally related to the ability to share the same basic hardware yet can also support different execution environments, i.e., different operating systems simultaneously.

The main drawback with the virtual-machine approach involves disk systems. Let us suppose that the physical machine has only three disk drives but wants to support seven virtual machines. Obviously, it cannot allocate a disk drive to each virtual machine, because virtual-machine software itself will need substantial disk space to provide virtual memory and spooling. The solution is to provide virtual disks.

Users are thus given their own virtual machines. After which they can run any of the operating systems or software packages that are available on the underlying machine. The virtual-machine software is concerned with multi-programming multiple virtual machines onto a physical machine, but it does not need to consider any user-support software. This arrangement can provide a useful way to divide the problem of designing a multi-user interactive system, into two smaller pieces.

Advantages:

1. There are no protection problems because each virtual machine is completely isolated from all other virtual machines.
2. Virtual machine can provide an instruction set architecture that differs from real computers.
3. Easy maintenance, availability and convenient recovery.

Disadvantages:

1. When multiple virtual machines are simultaneously running on a host computer, one virtual machine can be affected by other running virtual machines, depending on the workload.
2. Virtual machines are not as efficient as a real one when accessing the hardware.

HISTORY

Both system virtual machines and process virtual machines date to the 1960s and continue to be areas of active development.

System virtual machines grew out of time-sharing, as notably implemented in the Compatible Time-Sharing System (CTSS). Time-sharing allowed multiple users to use a computer concurrently: each program appeared to have full access to the machine, but only one program was executed at the time, with the system switching between programs in time slices, saving and restoring state each time. This evolved into virtual machines, notably via IBM's research systems: the M44/44X, which used partial virtualization, and the CP-40 and SIMMON, which used full virtualization, and were early examples of hypervisors. The first widely available virtual machine architecture was the CP-67/CMS (see History of CP/CMS for details). An important distinction was between using multiple virtual machines on one host system for time-sharing, as in M44/44X and CP-40, and using one virtual machine on a host system for prototyping, as in SIMMON. Emulators, with hardware emulation of earlier systems for compatibility, date back to the IBM System/360 in 1963,[6][7] while the software emulation (then-called "simulation") predates it.

Process virtual machines arose originally as abstract platforms for an intermediate language used as the intermediate representation of a program by a compiler; early examples date to around 1966. An early 1966 example was the O-code machine, a virtual machine that executes O-code (object code) emitted by the front end of the BCPL compiler. This abstraction allowed the compiler to be easily ported to a new architecture by implementing a new back end that took the existing O-code and compiled it to machine code for the underlying physical machine. The Euler language used a similar design, with the intermediate language named *P* (portable).[8] This was popularized around 1970 by Pascal, notably in the Pascal-P system (1973) and Pascal-S compiler (1975), in which it was termed p-code and the resulting machine as a p-code machine.

This has been influential, and virtual machines in this sense have been often generally called p-code machines. In addition to being an intermediate language, Pascal p-code was also executed directly by an interpreter implementing the virtual machine, notably in UCSD Pascal (1978); this influenced later interpreters, notably the Java virtual machine (JVM). Another early example was SNOBOL4 (1967), which was written in the SNOBOL Implementation Language (SIL), an assembly language for a virtual machine, which was then targeted to physical machines by transpiling to their native assembler via a macro assembler.[9] Macros have since fallen out of favor, however, so this approach has been less influential. Process virtual machines were a popular approach to implementing early microcomputer software, including Tiny BASIC and adventure games, from one-off implementations such as Pyramid 2000 to a general-purpose engine like Infocom's z-machine, which Graham Nelson argues is "possibly the most portable virtual machine ever created".

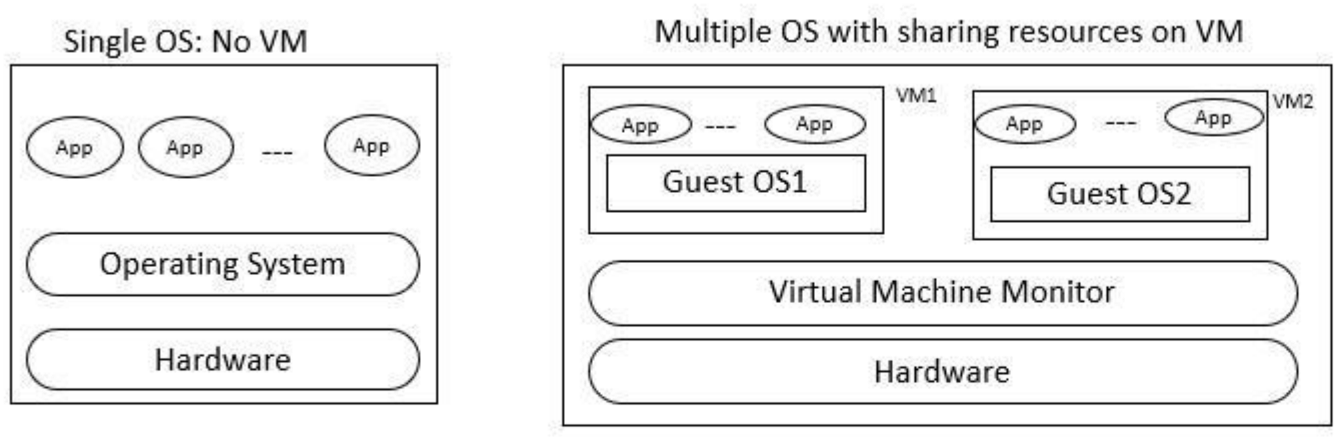
Significant advances occurred in the implementation of Smalltalk-80, particularly the Deutsch/Schiffmann implementation which pushed just-in-time (JIT) compilation forward as an implementation approach that uses process virtual machine. Later notable Smalltalk VMs were VisualWorks, the Squeak Virtual Machine, and Strongtalk. A related language that produced a lot of virtual machine innovation was the Self programming language, which pioneered adaptive optimization[17] and generational garbage collection. These techniques proved commercially successful in 1999 in the HotSpot Java virtual machine.[18] Other innovations include having a register-based virtual machine, to better match the underlying hardware, rather than a stack-based virtual machine, which is a closer match for the programming language; in 1995, this was pioneered by the Dis virtual machine for the Limbo language. OpenJ9 is an alternative for HotSpot JVM in OpenJDK and is an open source eclipse project claiming better startup and less resource consumption compared to HotSpot.

FEATURES OF VIRTUAL MACHINES

The features of the virtual machines are as follows −

- Multiple OS systems use the same hardware and partition resources between virtual computers.
- Separate Security and configuration identity.
- Ability to move the virtual computers between the physical host computers as holistically integrated files.

The below diagram shows you the difference between the single OS with no VM and Multiple OS with VM −



BENEFITS

Let us see the major benefits of virtual machines for operating-system designers and users which are as follows −

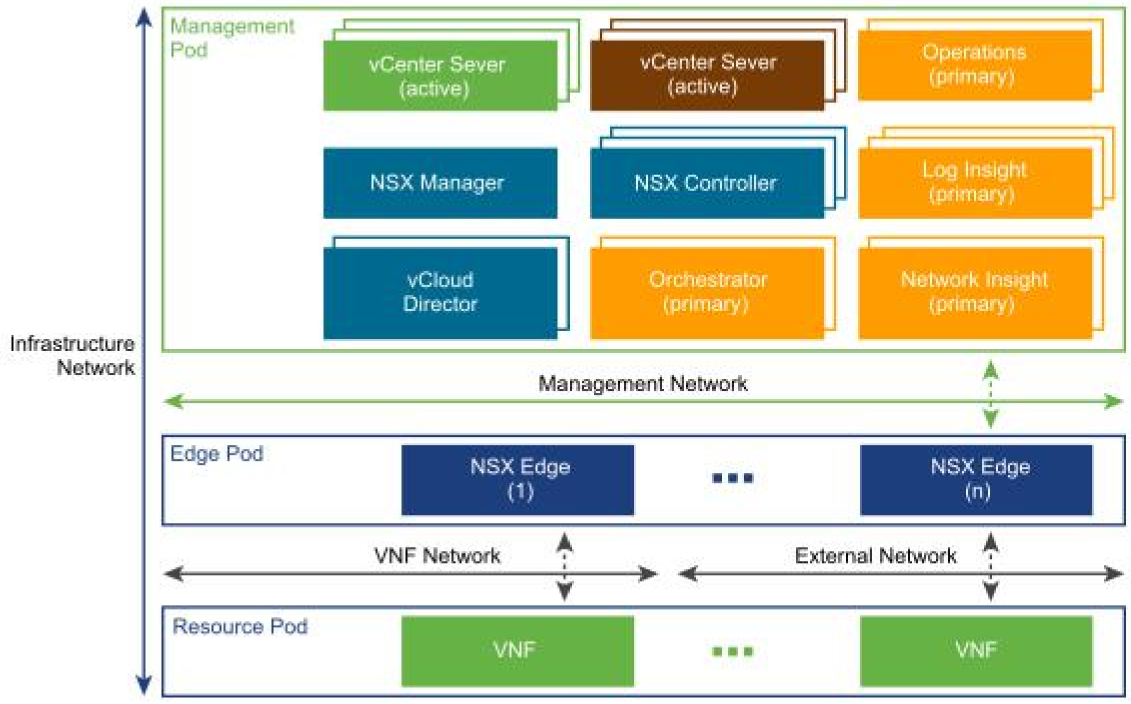
- The multiple Operating system environments exist simultaneously on the same machine, which is isolated from each other.
- Virtual machine offers an instruction set architecture which differs from real computer.
- Using virtual machines, there is easy maintenance, application provisioning, availability and convenient recovery.

Virtual Machine encourages the users to go beyond the limitations of hardware to achieve their goals.

The operating system achieves virtualization with the help of a specialized software called a hypervisor, which emulates the PC client or server CPU, memory, hard disk, network and other hardware resources completely, enabling virtual machines to share resources.

The hypervisor can emulate multiple virtual hardware platforms that are isolated from each other allowing virtual machines to run Linux and window server operating machines on the same underlying physical host.

VIRTUAL BUILDING BLOCKS



Storage Design

This reference architecture uses a shared storage design that is based on vSAN. vCloud NFV also supports certified third-party shared storage solutions, as listed in the VMware Compatibility Guide.

vSAN is a software feature built in the ESXi hypervisor that allows locally attached storage to be pooled and presented as a shared storage pool for all hosts in a vSphere cluster. This simplifies the storage configuration with a single datastore per cluster for management and VNF workloads. With vSAN, VM data is stored as objects and components. One object consists of multiple components, which are distributed across the vSAN cluster based on the policy that is assigned to the object. The policy for the object ensures a highly available storage backend for the cluster workload, with no single point of failure.

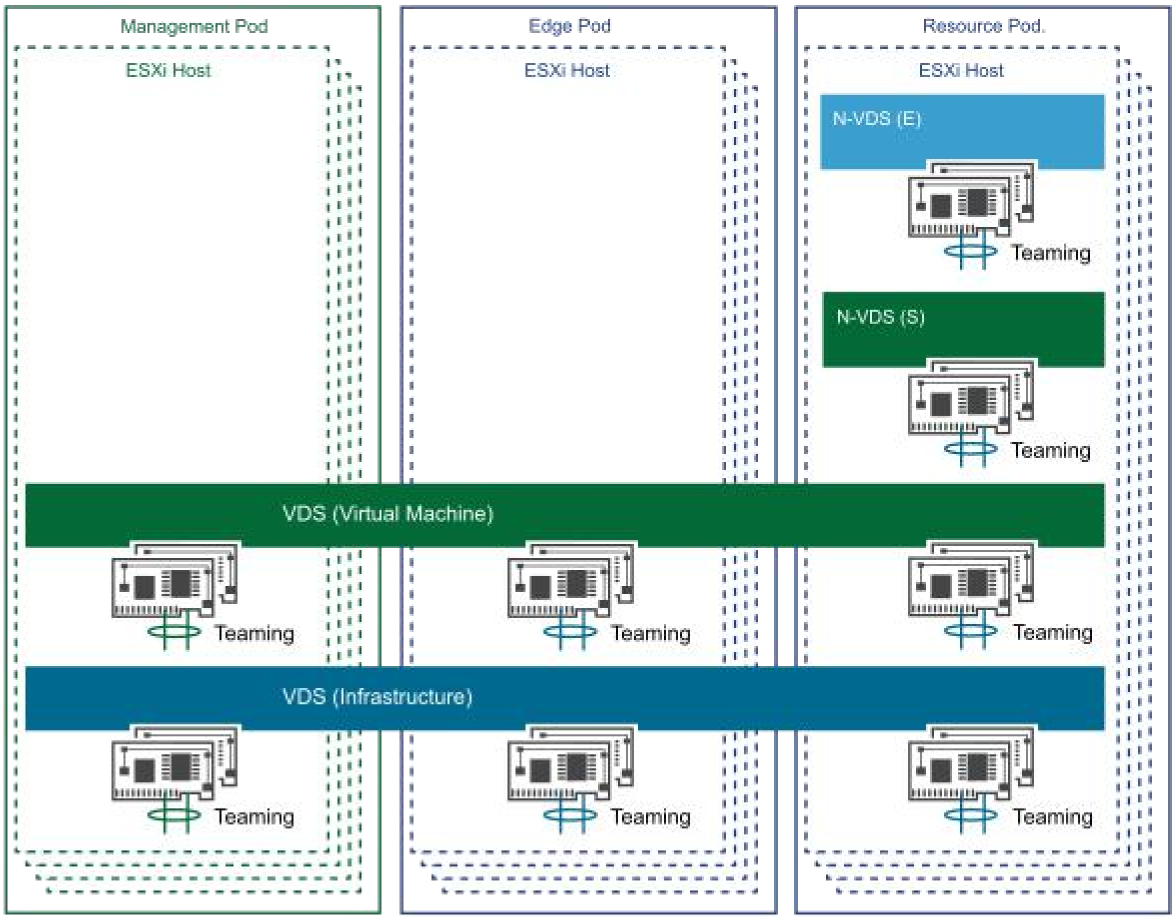
vSAN is a fully integrated hyperconverged storage software. Creating a cluster of server hard disk drives (HDDs) and solid-state drives (SSDs), vSAN presents a flash-optimized, highly resilient, shared storage datastore to ESXi hosts and virtual machines. This allows for the control of capacity, performance, and availability through storage policies, on a per VM basis.

Network Design

Thev Cloud NFV platform consists of infrastructure networks and VM networks. Infrastructure networks are host level networks that connect hypervisors to physical networks. Each ESXi host has multiple port groups configured for each infrastructure network.

The hosts in each Pod are configured with VMware vSphere® Distributed Switch™ (VDS) devices that provide consistent network configuration across multiple hosts. One vSphere Distributed Switch is used for VM networks and the other one maintains the infrastructure networks. Also, the N-VDS switch is used as the transport for telco workload traffic.

Virtual Network Design



Infrastructure networks are used by the ESXi hypervisor for vMotion, VMware vSphere Replication, vSAN traffic, and management and backup. The Virtual Machine networks are used by VMs to communicate with each other. For each Pod, the separation between infrastructure and VM networks ensures security and provides network resources where needed. This separation is implemented by two vSphere Distributed Switches, one for infrastructure networks and another one for VM networks. Each distributed switch has separate uplink connectivity to the physical data center network, completely separating its traffic from other network traffic. The uplinks are mapped to a pair of physical NICs on each ESXi host, for optimal performance and resiliency.

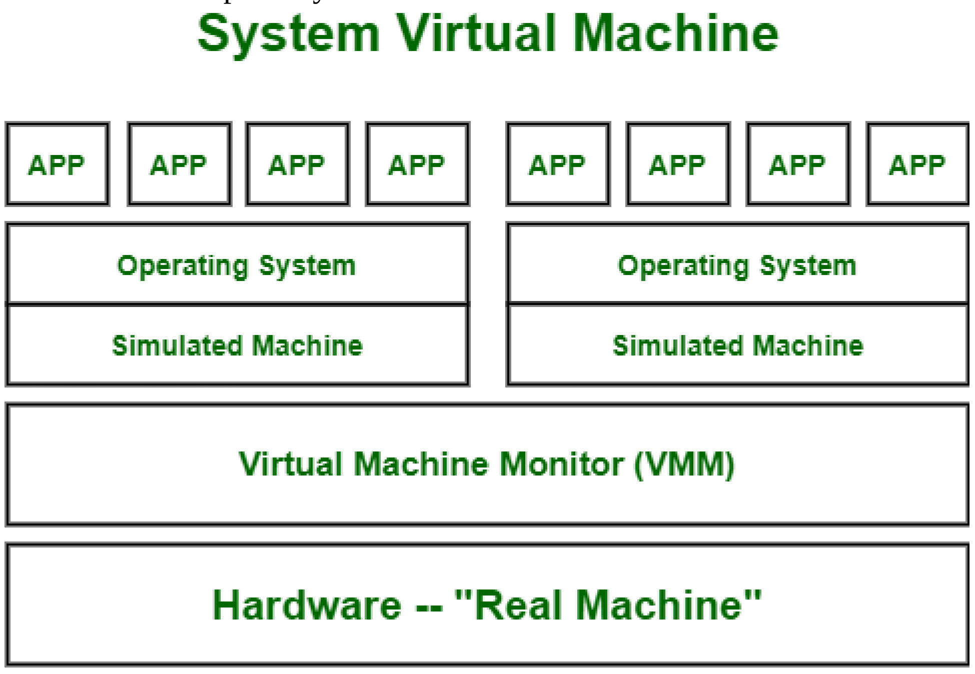
VMs can be connected to each other over a VLAN or over Geneve-based overlay tunnels. Both networks are designed according to the requirements of the workloads that are hosted by a specific Pod. The infrastructure vSphere Distributed Switch and networks remain the same regardless of the Pod function. However, the VM networks depend on the networks that the specific Pod requires. The VM networks are created by NSX-T Data Center to provide enhanced networking services and performance to the Pod workloads. The ESXi host's physical NICs are used as uplinks to connect the distributed switches to the physical network switches. All ESXi physical NICs connect to layer 2 or layer 3 managed switches on the physical network. It is common to use two switches for connecting to the host physical NICs for redundancy purposes.

TYPES OF VIRTUAL MACHINES

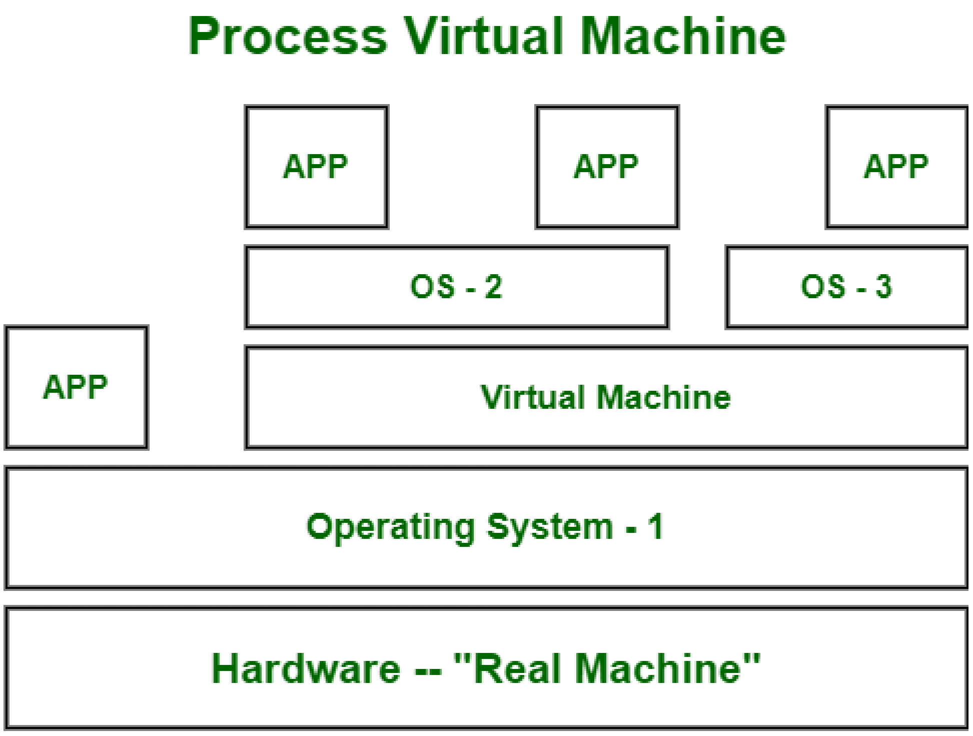
Virtual Machine is like fake computer system operating on your hardware. It partially uses the hardware of your system (like CPU, RAM, disk space, etc.) but its space is completely separated from your main system. Two virtual machines don't interrupt in each other's working and functioning nor they can access each other's space which gives an illusion that we are using totally different hardware system. More detail at Virtual Machine.

Types of Virtual Machines : Virtual machines are classified into two types:

1. **System Virtual Machine:** These types of virtual machines gives us complete system platform and gives the execution of the complete virtual operating system. Just like virtual box, system virtual machine is providing an environment for an OS to be installed completely. We can see in below image that our hardware of Real Machine is being distributed between two simulated operating systems by Virtual machine monitor. And then some programs, processes are going on in that distributed hardware of simulated machines separately.



2. Process Virtual Machine : While process virtual machines, unlike system virtual machine, does not provide us with the facility to install the virtual operating system completely. Rather it creates virtual environment of that OS while using some app or program and this environment will be destroyed as soon as we exit from that app. Like in below image, there are some apps running on main OS as well some virtual machines are created to run other apps. This shows that as those programs required different OS, process virtual machine provided them with that for the time being those programs are running. **Example** – Wine software in Linux helps to run Windows applications.



Virtual Machine Language : It's type of language which can be understood by different operating systems. It is platform-independent. Just like to run any programming language (C, python, or java) we need

Types of VMs – Type 0 Hypervisor

Old idea, under many names by HW manufacturers

- "partitions","domains"
- A HW feature implemented by firmware
- OS need to nothing special, VMM is in firmware
- Smaller feature set than other types
- Each guest has dedicated HW
- I/O a challenge as difficul to have enough devices, controlers to dedicate to each guest
- Sometimes VMM implements a **control partition** running daemons that other guests communicate with for shared I/O
- Can provide virtualization-within-virtualization(guest itself can be a VMM with guests
- Other types have difficulty in doing this.

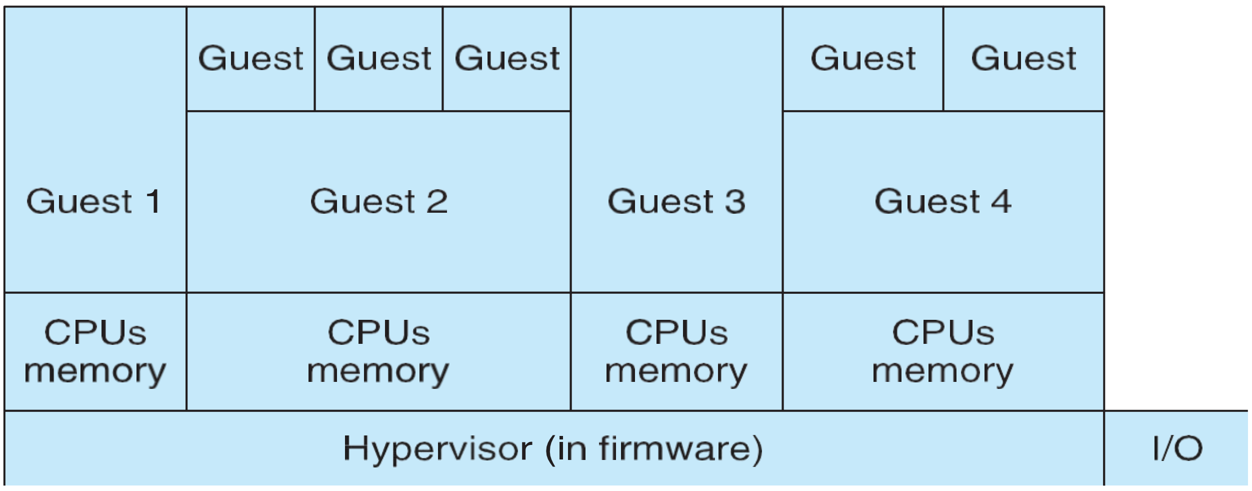
A **virtual machine (VM)** is a virtual environment which functions as a virtual computer system with its own CPU, memory, network interface, and storage, created on a physical hardware system.

Types of VMs – Type 1 Hypervisor

Commonly found in company data centers

- Special purpose operating systems that run natively on HW

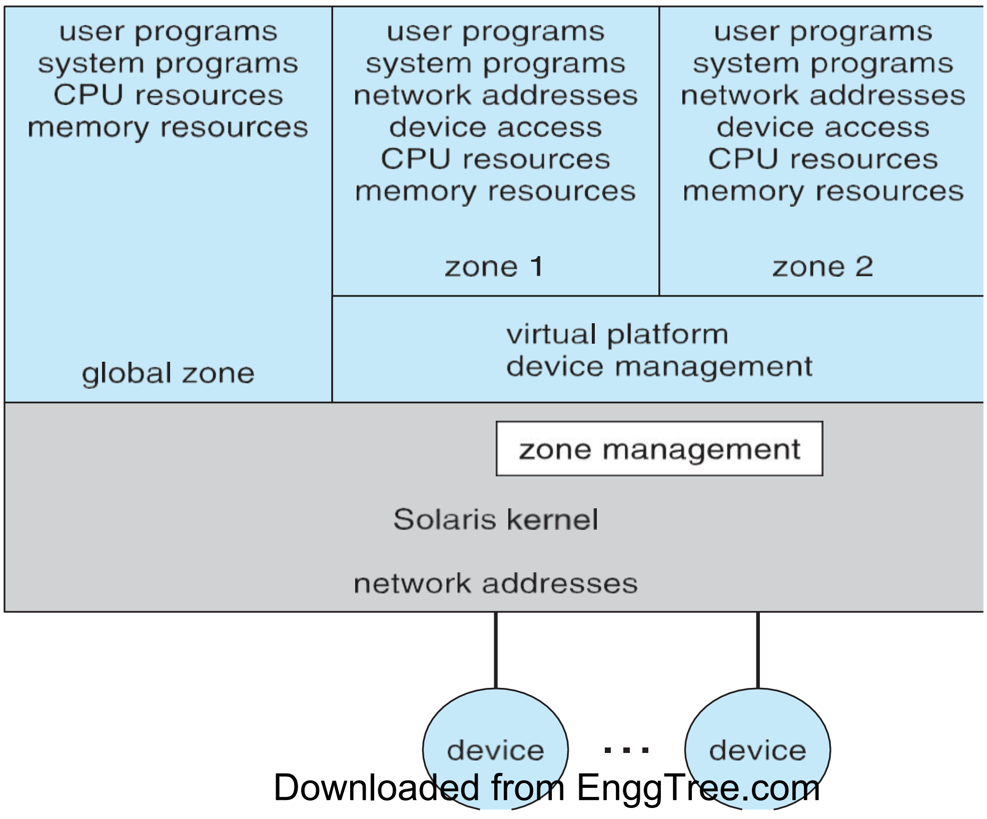
- Rather than providing system call interface, creater unand manage guest OSes.
- Can run on Type0 hypervisors but not on other Type1s
- Run in kernel mode
- Guests generally don't know they are running in a VM
- Implement device drivers for host HW because no other component can
- Also provide other traditional OS services like CPU and memory management



- Very little OS involvement in virtualization
- VMM is simply another process, run and managed by host
- ◗ Even the host doesn't know they are a VMM running guests
  - ◗ Tend to have poorer overall performance because can take advantage of some HW features
  - ◗ But also a benefit because require no changes to host OS

    - ◗ Student could have Type2 hypervisor on native host, run
    - ◗ Multiple guests, all on standard host OS such as Windows, Linux, MacOS

Solaris 10 with Two Zones



VIRTUALIZATION AND OPERATING-SYSTEM COMPONENTS

Now look at operating system aspects of virtualization

- CPU scheduling, memory management, I/O, storage, and unique VM migration feature

◗ How do VMM sschedule CPU use when guests believe they have dedicated CPUs?

- ◗ How can memory management work when many guests

Require large amounts of memory?

OS Component – CPU Scheduling

Even single-CPU systems act like multiprocessor ones when virtualized

- One or more virtual CPUs per guest
- Generally VMM has one or more physical CPUs and number of threads to run on them.
- Guests configured with certain number of VCPUs
  - ◗ Can be adjusted throughout life of VM

When enough CPUs for all guests->VMM can allocate dedicated CPUs, each guest much like native operating system managing its CPUs

Usually not enough CPUs->CPU **over commitment**

VMM can use standard scheduling algorithms to put threads on CPUs

Some add fairness aspect

Cycle stealing by VMM and oversubscription of CPUs means guests don't get CPU cycles they expect.

- ◗ Consider timesharing scheduler in a guest trying to schedule 100ms time slices -> each may take 100ms, 1 second, or longer
- ◗ Poor response times for users of guest
- ◗ Time-of-day clocks incorrect
- ◗ Some VMMs provide application to run in each guest to fix time-of-day and provide other integration features

OS Component – Memory Management

Also suffers from over subscription -> requires extra management efficiency from VMM

For example, VMware ESX guests have a configured amount of physical memory, then ESX uses 3 methods of memory management

1. Double-paging, in which the guest page table indicates a page is in a physical frame but the VMM moves some of those pages to backing store

2. Install a pseudo-device driver in each guest (it looks like a device driver to the guest kernel but really just adds kernel-mode code to the guest)

- ◗ Balloon memory manager communicates with VMM and is told to allocate or deallocate memory to decrease or increase physical memory use of guest, causing guest OS to free or have more memory available

4. Deduplication by VMM determining if same page loaded more than once, memory mapping the same page into multiple guests

OS Component – I/O

Easier for VMMs to integrate with guests because I/O has lots of variation

Already somewhat segregated / flexible via device drivers

VMM can provide new devices and device drivers

But overall I/O is complicated for VMMs

- ◗ Many short paths for I/O in standard OSes for improved performance
- ◗ Less hypervisor needs to do for I/O for guests, the better
- ◗ Possibilities include direct device access, DMA pass-through, direct interrupt delivery
- o Again, HW support needed for these

Networking also complex as VMM and guests all need network access

- o VMM can bridge guest to network (allowing direct access)
- o And / or provide network address translation (NAT)
- o NAT address local to machine on which guest is running, VMM provides address translation to guest to hide its address.

OS Component – Storage Management

- Both boot disk and general data access need be provided by VMM
- Need to support potentially dozens of guests per VMM (so standard disk partitioning not sufficient)
- Type 1 – storage guest root disks and config information within file system provided by VMM as a disk image
- Type 2 – store as files in file system provided by host OS
- Duplicate file -> create new guest
- Move file to another system -> move guest
- **Physical-to-virtual (P-to-V)** convert native disk blocks into VMM format
- **Virtual-to-physical (V-to-P)** convert from virtual format to native or disk format

VMM also needs to provide access to network attached storage (just networking) and other disk images, disk partitions, disks, etc.

OS Component – Live Migration

Taking advantage of VMM features leads to new functionality not found on general operating systems such as live migration

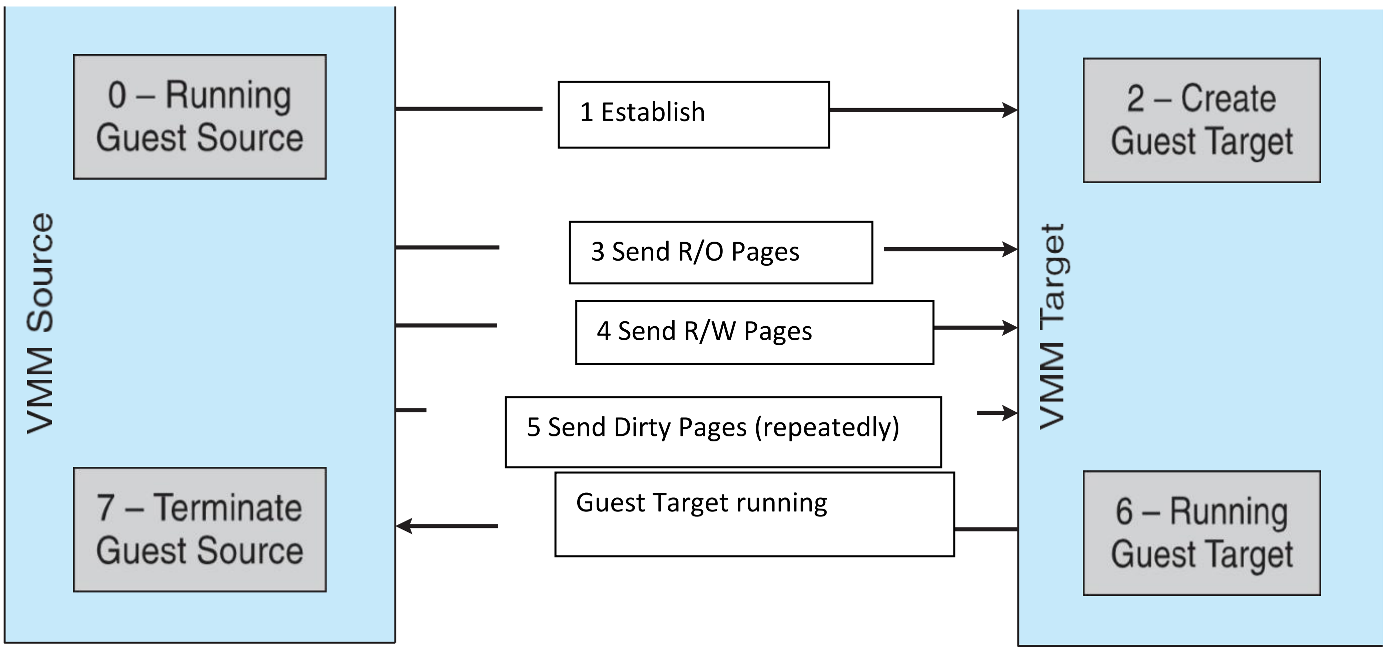
Running guest can be moved between systems, without interrupting user access to the guest or its apps

Very useful for resource management, maintenance downtime windows, etc

1. The source VMM establishes a connection with the target VMM
2. The target creates a new guest by creating a new VCPU, etc
3. The source sends all read-only guest memory pages to the target
4. The source sends all read-write pages to the target, marking them as clean
5. The source repeats step 4, as during that step some pages were probably modified by the guest and are now dirty
6. When cycle of steps 4 and 5 becomes very short, source VMM freezes guest, sends VCPU's final state, sends other state details, sends final dirty pages, and tells target to start running the guest

Once target acknowledges that guest running, source terminates guest.

Live Migration of Guest Between Servers



BASIS FOR DEVELOPING THE OS

Create the illusion of having one or more objects to emulate the real object. It is closely related to abstraction. In developing the OS, abstraction provides simplification by combining multiple simple objects into a single complex object

Virtualization provides diversification and replication by creating the illusion of objects with desired characteristics.

The virtual infrastructure design comprises the design of the software components that form the virtual infrastructure layer. This layer supports running telco workloads and workloads that maintain the business continuity of services. The virtual infrastructure components include the virtualization platform hypervisor, virtualization management, storage virtualization, network virtualization, and backup and disaster recovery components.

This section outlines the building blocks for the virtual infrastructure, their components, and the networking to tie all the components together.

MOBILE OPERATING SYSTEM

A mobile operating system is an operating system that helps to run other application software on mobile devices. It is the same kind of software as the famous computer operating systems like Linux and Windows, but now they are light and simple to some extent.

The operating systems found on smartphones include Symbian OS, iPhone OS, RIM's BlackBerry, Windows Mobile, Palm WebOS, Android, and Maemo. Android, WebOS, and Maemo are all derived from Linux. The iPhone OS originated from BSD and NeXTSTEP, which are related to Unix.

It combines the beauty of computer and hand use devices. It typically contains a cellular built-in modem and SIM tray for telephony and internet connections. If you buy a mobile, the manufacturer company chooses the OS for that specific device.

Popular platforms of the Mobile OS

1. Android OS: The Android operating system is the most popular operating system today. It is a mobile OS based on the **Linux Kernel** and **open-source software**. The android operating system was developed by **Google**. The first Android device was launched in **2008**.

2. Bada (Samsung Electronics): Bada is a Samsung mobile operating system that was launched in 2010. The Samsung wave was the first mobile to use the bada operating system. The bada operating system offers many mobile features, such as 3-D graphics, application installation, and multipoint-touch.

3. BlackBerry OS: The BlackBerry operating system is a mobile operating system developed by **Research In Motion** (RIM). This operating system was designed specifically for BlackBerry handheld devices. This operating system is beneficial for the corporate users because it provides synchronization with Microsoft Exchange, Novell GroupWise email, Lotus Domino, and other business software when used with the BlackBerry Enterprise Server.

4. iPhone OS / iOS: The iOS was developed by the Apple inc for the use on its device. The iOS operating system is the most popular operating system today. It is a very secure operating system. The iOS operating system is not available for any other mobiles.

5. Symbian OS: Symbian operating system is a mobile operating system that provides a high-level of integration with communication. The Symbian operating system is based on the java language. It combines middleware of wireless communications and personal information management (PIM) functionality. The Symbian operating system was developed by **Symbian Ltd** in **1998** for the use of mobile phones. **Nokia** was the first company to release Symbian OS on its mobile phone at that time.

6. Windows Mobile OS: The window mobile OS is a mobile operating system that was developed by **Microsoft**. It was designed for the pocket PCs and smart mobiles.

7. Harmony OS: The harmony operating system is the latest mobile operating system that was developed by Huawei for the use of its devices. It is designed primarily for IoT devices.

8. Palm OS: The palm operating system is a mobile operating system that was developed by **Palm Ltd** for use on personal digital assistants (PADs). It was introduced in **1996**. Palm OS is also known as the **Garnet OS**.

9. WebOS (Palm/HP): The WebOS is a mobile operating system that was developed by **Palm**. It based on the **Linux Kernel**. The HP uses this operating system in its mobile and touchpads.

What is Apple iOS?

Apple iOS is a proprietary mobile operating system that runs on mobile devices such as the iPhone and iPad. Apple iOS stands for iPhone operating system and is designed for use with Apple's multitouch devices. The mobile OS supports input through direct manipulation and responds to various user gestures, such as pinching, tapping and swiping. The iOS developer kit provides tools that allow for iOS app development.

Apple iOS market share

As of 2022, the Apple iOS market share was 18.8% worldwide, making it the second most popular brand behind Samsung, according to IDC.

Apple iOS market share

As of 2019, the Apple iOS market share was 13.4% worldwide, making it the second most popular mobile OS behind Google Android, according to IDC.

Apple iOS features

- Wi-Fi, Bluetooth and cellular connectivity, along with VPN support.
- Integrated search support, which enables simultaneous search through files, media, applications and email.
- Gesture recognition supports -- for example, shaking the device to undo the most recent action.

- Push email.
- Safari mobile browser.
- Integrated front- and rear-facing cameras with video capabilities.
- Direct access to the Apple App Store and the iTunes catalog of music, podcasts, television shows and movies available to rent or purchase. iOS is also designed to work with Apple TV.
- Compatibility with Apple's cloud service, iCloud.
- Siri is Apple's virtual assistant that can set reminders, offer suggestions or interact with certain third-party apps. Siri's voice has been modified recently to make it sound more natural.
- Cross-platform communications between Apple devices through AirDrop.
- Support for Apple Watch, runs watchOS 9 but requires iPhone 8 or later running iOS 16 or later.
- Apple Pay, which stores users' credit card data and allows them to pay for goods and services directly with an iOS device.
- CarPlay allows users to interact with an iOS device while driving. CarPlay supports Siri voice controls, and users can access apps through a connected vehicle's touchscreen. CarPlay provides access to maps, phone, calendar, messaging, and music apps.
- The HomePod feature allows Siri to identify family members by voice, giving everyone a personalized experience. HomePod's handoff feature allows users to hand off music, podcasts and phone calls so that they can listen on another device.
- HomeKit allows iOS to be used as a tool for controlling home automation. HomeKit accessories include routers, lights, security cameras and more. The Home app allows you to control these devices from iOS.

What are the security and privacy features of Apple iOS?

iOS includes the following security features:

- **Apple ID support.** Users can sign into websites and apps using their existing Apple ID. Additionally, iOS supports signing in using Face ID or Touch ID, which use biometric authentication methods. Apple IDs are protected with two-factor authentication.

- **Privacy and security.** iOS supports fine-grained controls that prevent apps from gaining location information or accepting AirDrop content from unknown senders. Apps can also be blocked from using Wi-Fi or Bluetooth without users' permission. Additionally, iOS devices use a secure boot chain to ensure that only trusted (signed) code is executed during the boot process. This allows iOS devices to verify the integrity of any code running on the device.
- **Secure Enclave Support.** Secure Enclave is a hardware-based feature that stores cryptographic keys in an isolated location to prevent those keys from being compromised. Secure Enclave is not exclusive to iOS devices. It also works with Apple TV, Apple Watch, Mac computers and other Apple products.

Apple iOS version history

Apple iOS was originally known as iPhone OS. The company released three versions of the mobile OS under that name before iOS 4 debuted in June 2010. Apple released iOS 2 on July 11, 2008. It premiered alongside Apple's iPhone 3G. This operating system was followed on June 17, 2009 by iOS 3. The fourth version of iOS was released on June 21, 2010, along with the iPhone 4.

On Oct. 12, 2011, Apple released iOS 5, which expanded the number of available applications to over 500,000. This iOS version also added the Notification Center, a camera app, Siri and more.Unveiled on June 11, 2012, iOS 6 included a Maps application and the Passbook ticket storage and loyalty program application.

Released on Sept. 18, 2013, iOS 7 featured an entirely redesigned user interface. In September 2014, iOS 8 introduced Continuity, a cross-platform system that allows users of multiple Apple devices to pick up on one where they left off from another. Other new features included the Photos app and Apple Music.

ANDROID OPERATING SYSTEM

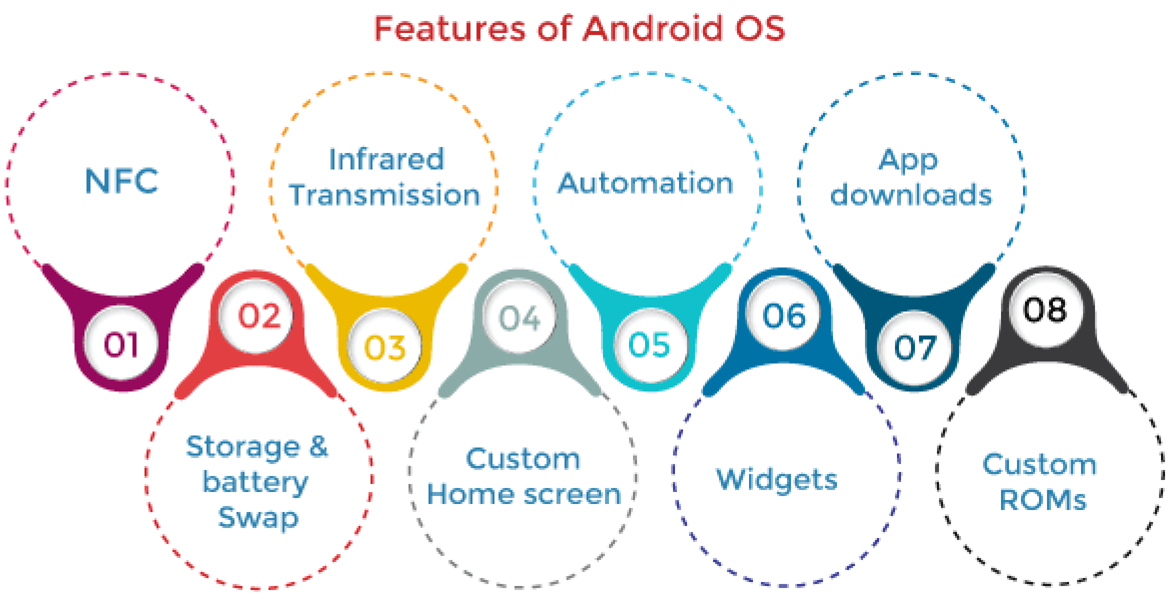
Android is a mobile operating system based on a modified version of the Linux kernel and other open-source software, designed primarily for touchscreen mobile devices such as smartphones and tablets. Android is developed by a partnership of developers known as the Open Handset Alliance and commercially sponsored by Google. It was disclosed in November 2007, with the first commercial Android device, the HTC Dream, launched in September 2008.

It is free and open-source software. Its source code is Android Open Source Project (AOSP), primarily licensed under the Apache License. However, most Android devices dispatch with additional proprietary software pre-installed, mainly Google Mobile Services (GMS), including core apps such as Google Chrome, the digital distribution platform Google Play and the associated Google Play Services development platform.

- About 70% of Android Smartphone runs Google's ecosystem, some with vendor-customized user interface and some with software suite, such as ***TouchWiz***and later ***One UI*** by Samsung, and ***HTC Sense***.
- Competing Android ecosystems and forksinclude Fire OS (developed by Amazon) or LineageOS. However, the "Android" name and logo are trademarks of Google which impose standards to restrict "uncertified" devices outside their ecosystem to use android branding.

Features of Android Operating System

Below are the following unique features and **characteristics of the android operating system, such as:**



1. Near Field Communication (NFC)

Most Android devices support NFC, which allows electronic devices to interact across short distances easily. The main goal here is to create a payment option that is simpler than carrying cash or credit cards, and while the market hasn't exploded as many experts had predicted, there may be an alternative in the works, in the form of Bluetooth Low Energy (BLE).

2. Infrared Transmission

The Android operating system supports a built-in infrared transmitter that allows you to use your phone or tablet as a remote control.

3. Automation

The ***Tasker*** app allows control of app permissions and also automates them.

4. Wireless App Downloads

You can download apps on your PC by using the Android Market or third-party options like ***AppBrain***. Then it automatically syncs them to your Droid, and no plugging is required.

5. Storage and Battery Swap

Android phones also have unique hardware capabilities. Google's OS makes it possible to upgrade, replace, and remove your battery that no longer holds a charge. In addition, Android phones come with SD card slots for expandable storage.

6. Custom Home Screens

While it's possible to hack certain phones to customize the home screen, Android comes with this capability from the get-go. Download a third-party launcher like ***Apex, Nova***, and you can add gestures, new shortcuts, or even performance enhancements for older-model devices.

7. Widgets

Apps are versatile, but sometimes you want information at a glance instead of having to open an app and wait for it to load. Android widgets let you display just about any feature you choose on the home screen, including weather apps, music widgets, or productivity tools that helpfully remind you of upcoming meetings or approaching deadlines.

8. Custom ROMs

Because the Android operating system is open-source, developers can twist the current OS and build their versions, which users can download and install in place of the stock OS. Some are filled with features, while others change the look and feel of a device. Chances are, if there's a feature you want, someone has already built a custom ROM for it.

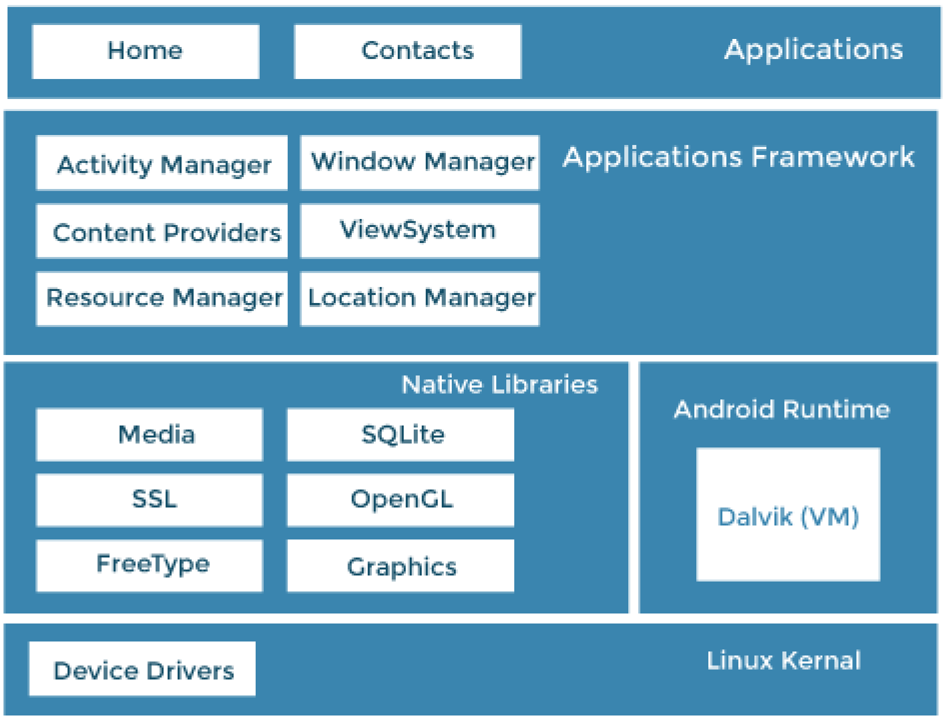
Architecture of Android OS

The android architecture contains a different number of components to support any android device needs. Android software contains an open-source Linux Kernel with many C/C++ libraries exposed through application framework services.

Among all the components, Linux Kernel provides the main operating system functions to Smartphone and Dalvik Virtual Machine (DVM) to provide a platform for running an android application. An android operating system is a stack of software components roughly divided into five sections and four main layers, as shown in the below architecture diagram.

- Applications
- Application Framework
- Android Runtime
- Platform Libraries

- Linux Kernel



1. Applications

An application is the top layer of the android architecture. The pre-installed applications like camera, gallery, home, contacts, etc., and third-party applications downloaded from the play store like games, chat applications, etc., will be installed on this layer.

It runs within the Android run time with the help of the classes and services provided by the application framework.

2. Application framework

Application Framework provides several important classes used to create an Android application. It provides a generic abstraction for hardware access and helps in managing the user interface with application resources. Generally, it provides the services with the help of which we can create a particular class and make that class helpful for the Applications creation.

It includes different types of services, such as activity manager, notification manager, view system, package manager etc., which are helpful for the development of our application according to the prerequisite.

The Application Framework layer provides many higher-level services to applications in the form of Java classes. Application developers are allowed to make use of these services in their applications. The Android framework includes the following key services:

- **Activity Manager:** Controls all aspects of the application lifecycle and activity stack.

- **Content Providers:** Allows applications to publish and share data with other applications.
- **Resource Manager:** Provides access to non-code embedded resources such as strings, colour settings and user interface layouts.
- **Notifications Manager:** Allows applications to display alerts and notifications to the user.
- **View System:** An extensible set of views used to create application user interfaces.

3. Application runtime

Android Runtime environment contains components like core libraries and the Dalvik virtual machine (DVM). It provides the base for the application framework and powers our application with the help of the core libraries.

Like ***Java Virtual Machine*** (JVM), ***Dalvik Virtual Machine*** (DVM) is a register-based virtual machine designed and optimized for Android to ensure that a device can run multiple instances efficiently.

It depends on the layer Linux kernel for threading and low-level memory management. The core libraries enable us to implement android applications using the standard ***JAVA*** or ***Kotlin*** programming languages.

4. Platform libraries

The Platform Libraries include various C/C++ core libraries and Java-based libraries such as Media, Graphics, Surface Manager, OpenGL, etc., to support Android development.

- **app:** Provides access to the application model and is the cornerstone of all Android applications.
- **content:** Facilitates content access, publishing and messaging between applications and application components.
- **database:** Used to access data published by content providers and includes SQLite database, management classes.
- **OpenGL:** A Java interface to the OpenGL ES 3D graphics rendering API.
- **os:** Provides applications with access to standard operating system services, including messages, system services and inter-process communication.
- **text:** Used to render and manipulate text on a device display.
- **view:** The fundamental building blocks of application user interfaces.
- **widget:** A rich collection of pre-built user interface components such as buttons, labels, list views, layout managers, radio buttons etc.

- **WebKit:** A set of classes intended to allow web-browsing capabilities to be built into applications.
- **media:** Media library provides support to play and record an audio and video format.
- **surface manager:** It is responsible for managing access to the display subsystem.
- **SQLite:** It provides database support, and FreeType provides font support.
- **SSL:** Secure Sockets Layer is a security technology to establish an encrypted link between a web server and a web browser.

5. Linux Kernel

Linux Kernel is the heart of the android architecture. It manages all the available drivers such as display, camera, Bluetooth, audio, memory, etc., required during the runtime.

The Linux Kernel will provide an abstraction layer between the device hardware and the other android architecture components. It is responsible for the management of memory, power, devices etc. The features of the Linux kernel are:

- **Security:** The Linux kernel handles the security between the application and the system.
- **Memory Management:** It efficiently handles memory management, thereby providing the freedom to develop our apps.
- **Process Management:** It manages the process well, allocates resources to processes whenever they need them.
- **Network Stack:** It effectively handles network communication.
- **Driver Model:** It ensures that the application works properly on the device and hardware manufacturers responsible for building their drivers into the Linux build.

Android Applications

Android applications are usually developed in the Java language using the Android Software Development Kit. Once developed, Android applications can be packaged easily and sold out either through a store such as ***Google Play, SlideME, Opera Mobile Store, Mobango, F-droid*** or the ***Amazon Appstore***.

Android powers hundreds of millions of mobile devices in more than 190 countries around the world. It's the largest installed base of any mobile platform and growing fast. Every day more than 1 million new Android devices are activated worldwide.

Smartphones
Tablets
android Operating System
Android Smart TV
Smart watches
Smart TVs

Android Emulator

The Emulator is a new application in the Android operating system. The Emulator is a new prototype used to develop and test android applications without using any physical device.

The android emulator has all of the hardware and software features like mobile devices except phone calls. It provides a variety of navigation and control keys. It also provides a screen to display your application. The emulators utilize the android virtual device configurations. Once your application is running on it, it can use services of the android platform to help other applications, access the network, play audio, video, store, and retrieve the data.

Advantages of Android Operating System

We considered every one of the elements on which Android is better as thought about than different platforms. Below are some important advantages of Android OS, such as:

- **Android Google Developer:** The greatest favourable position of Android is Google. Google claims an android operating system. Google is a standout amongst the most trusted and rumoured item on the web. The name Google gives trust to the clients to purchase Android gadgets.
- **Android Users:** Android is the most utilized versatile operating system. More than a billion individuals clients utilize it. Android is likewise the quickest developing operating system in the world. Various clients increment the number of applications and programming under the name of Android.
- **Android Multitasking:** The vast majority of us admire this component of Android. Clients can do heaps of undertakings on the double. Clients can open a few applications on the double and oversee them very. Android has incredible UI, which makes it simple for clients to do multitasking.

- **Google Play Store App:** The best part of Android is the accessibility of many applications. Google Play store is accounted for as the world's largest mobile store. It has practically everything from motion pictures to amusements and significantly more. These things can be effortlessly downloaded and gotten to through an Android phone.
- **Android Notification and Easy Access:** Without much of a stretch, one can access their notice of any SMS, messages, or approaches their home screen or the notice board of the android phone. The client can view all the notifications on the top bar. Its UI makes it simple for the client to view more than 5 Android notices immediately.
- **Android Widget:** Android operating system has a lot of widgets. This gadget improves the client encounter much and helps in doing multitasking. You can include any gadget relying on the component you need on your home screen. You can see warnings, messages, and a great deal more use without opening applications.

Disadvantages of Android Operating System

We know that the Android operating system has a considerable measure of interest for users nowadays. But at the same time, it most likely has a few weaknesses. Below are the following disadvantages of the android operating system, such as:

- **Android Advertisement pop-ups:** Applications are openly accessible in the Google play store. Yet, these applications begin demonstrating tons of advertisements on the notification bar and over the application. This promotion is extremely difficult and makes a massive issue in dealing with your Android phone.
- **Android require Gmail ID:** You can't get to an Android gadget without your email ID or password. Google ID is exceptionally valuable in opening Android phone bolts as well.
- **Android Battery Drain:** Android handset is considered a standout amongst the most battery devouring operating systems. In the android operating system, many processes are running out of sight, which brings about the draining of the battery. It is difficult to stop these applications as the lion's share of them is system applications.
- **Android Malware/Virus/Security:** Android gadget is not viewed as protected when contrasted with different applications. Hackers continue attempting to take your data. It is anything but difficult to target any Android phone, and each day millions of attempts are done on Android phones.