**Unit I :** **8 hours**

**Introduction to System Software , Overview of all system software's :**

Operating system , I/O manager, Assembler , Compiler, Linker , Loader.

Introductory Concepts : Operating system functions and characteristics.

Historical evolution of operating systems.

Real time systems.

Distributed systems.

An operating system (OS) is a collection of software that manages computer hardware resources and provides common services for computer programs. The operating system is a vital component of the system software in a computer system.

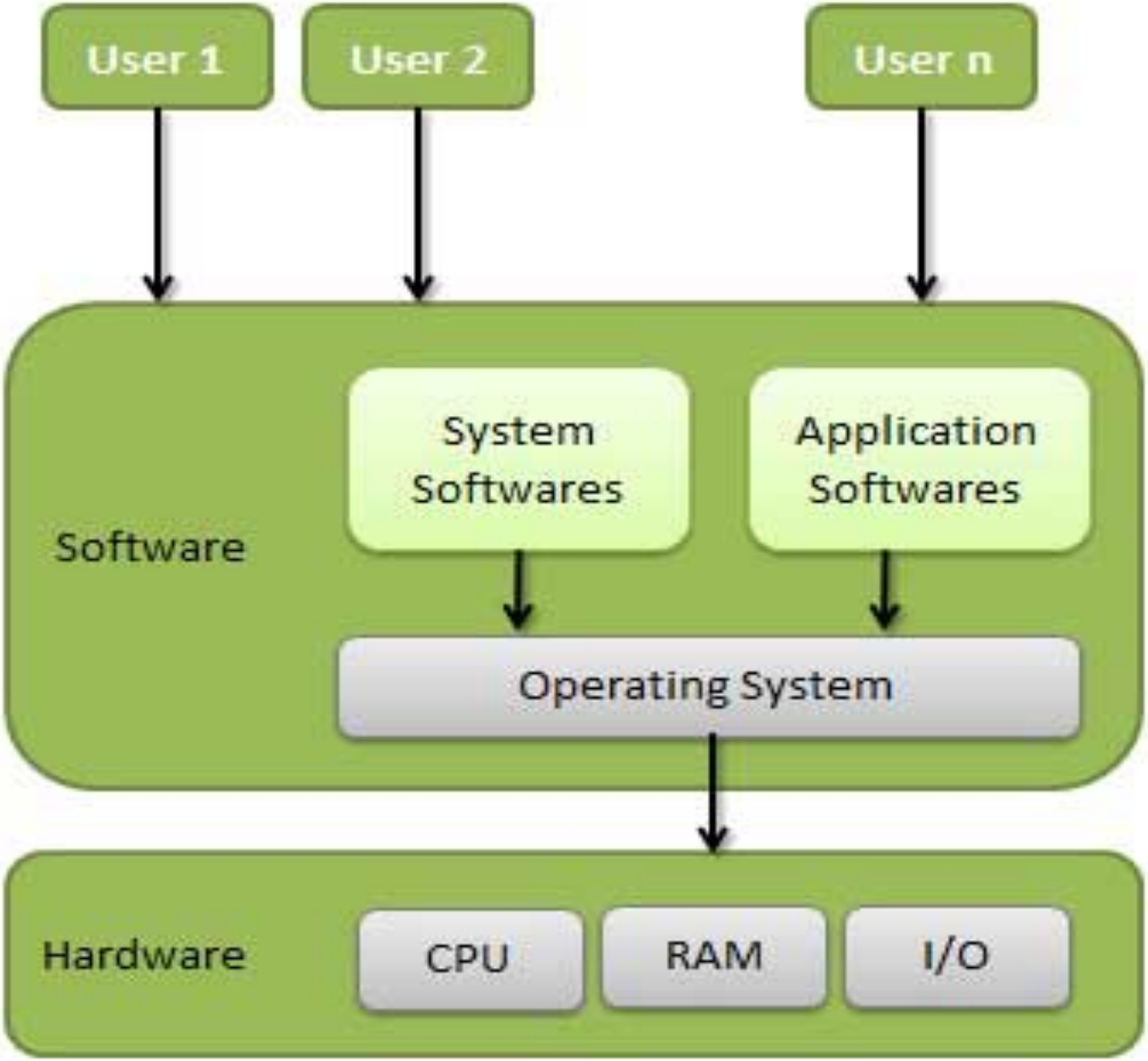
An operating System (OS) is an intermediary between users and computer hardware. It provides users an environment in which a user can execute programs conveniently and efficiently.

In technical terms, It is a software which manages hardware. An operating System controls the allocation of resources and services such as memory, processors, devices and information.

An operating system is a program that acts as an interface between the user and the computer hardware and controls the execution of all kinds of programs.

Following are some of important functions of an operating System.

a) Memory Management
b) Processor Management
c) Device Management
d) File Management
e) Security
f) Control over system performance
g) Job accounting
h) Error detecting aids
i) Coordination between other software and users


User 1
User 2
User n
Software
System Softwares
Application Softwares
Operating System
Hardware
CPU
RAM
I/O

Computer software can be divided into two main categories:

**Application Software & System Software.**

**Application software** consists of the programs for performing tasks particular to the machine's utilization. This software is designed to solve a particular problem for users.

e.g. spreadsheets, database systems, desktop publishing systems, program development software, and games etc.

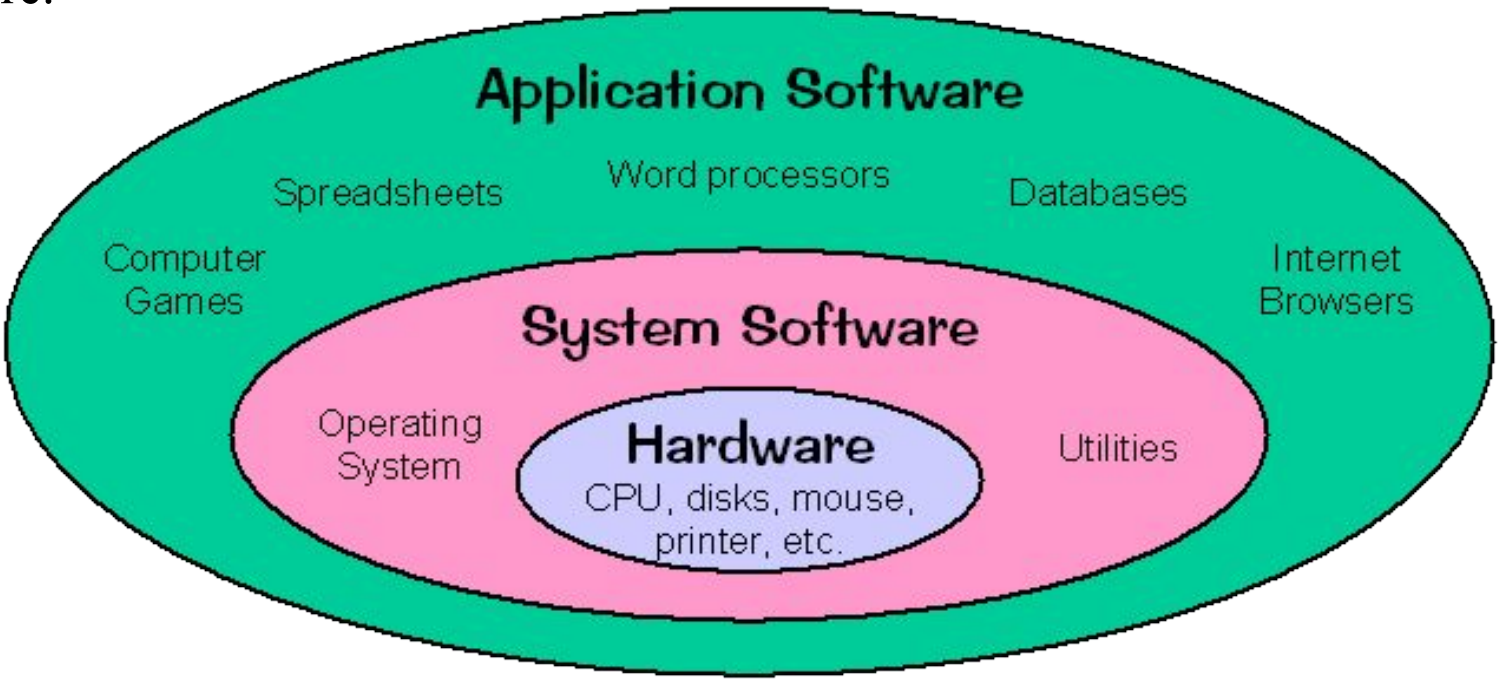
**System Software** is more transparent and less noticed by the typical computer user. This software provides a general programming environment in which programmers can create specific applications to suit their needs.

This environment provides new functions that are not available at the hardware level and performs tasks related to executing the application program.

System software acts as an interface between the hardware of the computer and the application software that users need to run on the computer.

The diagram below illustrates the relationship between application software and system software.

The most important type of system s/w is the OS




An operating system has three main responsibilities :

1) Perform basic tasks, such as recognizing i/p from the k/b, sending o/p to the display screen,
2) Keeping track of files and directories on the disk, and
3) Controlling peripheral devices such as disk drives and printers.

Ensure that different programs and users running at the same time do not interfere with each other.

Provide a software platform on top of which other programs (i.e. application software) can run.

The first two responsibilities address the need for managing the computer hardware and the application programs that use the hardware.

The third responsibility focuses on providing an interface between application software and hardware so that application software can be efficiently developed.

Since the operating system is already responsible for managing the hardware, it should provide a programming interface for application developers.

There are four common types of operating system strategies on which modern operating systems are built :

a) Batch b) Timesharing c) personal computing d) dedicated.

| | | |
|---|---|---|
| Batch | | This strategy involves reading a series of jobs (called a batch) into the machine and then executing the programs for each job in the batch. This approach does not allow users to interact with programs while they operate. |
| Timesharing | | This strategy supports multiple interactive users. Rather than preparing a job for execution ahead of time, users establish an interactive session with the computer and then provide commands, programs and data as they are needed during the session. |
| Personal Computing | | This strategy supports a single user running multiple programs on a dedicated machine. Since only one person is using the machine, more attention is given to establishing predictable response times from the system. This strategy is quite common today because of the popularity of personal computers. |
| Dedicated | | This strategy supports real-time and process control systems. These are the types of systems which control satellites, robots, and air-traffic control. The dedicated strategy must guarantee certain response times for particular computing tasks or the application is useless |

A computer consists of various devices that provide input and output (I/O) to and from the outside world. Typical devices are keyboards, mice, audio controllers, video controllers, disk drives, networking ports, and so on. Device drivers provide the software connection between the devices and the operating system. For this reason, I/O is very important to the device driver writer.

# I/O Manager

The Windows kernel-mode I/O manager manages the communication between applications and the interfaces provided by device drivers. Because devices operate at speeds that may not match the operating system, the communication between the operating system and device drivers is primarily done through **I/O Request Packets (IRPs).** These packets are similar to network packets or Windows message packets. They are passed from operating system to specific drivers and from one driver to another.

The Windows I/O system provides a layered driver model called stacks. Typically IRPs go from one driver to another in the same stack to facilitate communication.

e.g. a joystick driver would need to communicate to a USB hub, which in turn would need to communicate to a USB host controller, which would then need to communicate through a PCI bus to the rest of the computer hardware. The stack consists of joystick driver, USB hub, USB host controller, and the PCI bus.

This communication is coordinated by having each **driver in the stack** send and receive IRPs.

The I/O manager has two subcomponents: the Plug and Play Manager and Power Manager. They manage the I/O functionality for the technologies of Plug and Play and power management.

Plug and Play (PnP) is a combination of hardware technology and software techniques that enables a PC to recognize when a device is added to the system. With PnP, the system configuration can change with little or no input from the user. For example, when a USB thumb drive is plugged in, Windows can detect the thumb drive and add it to the file system automatically.

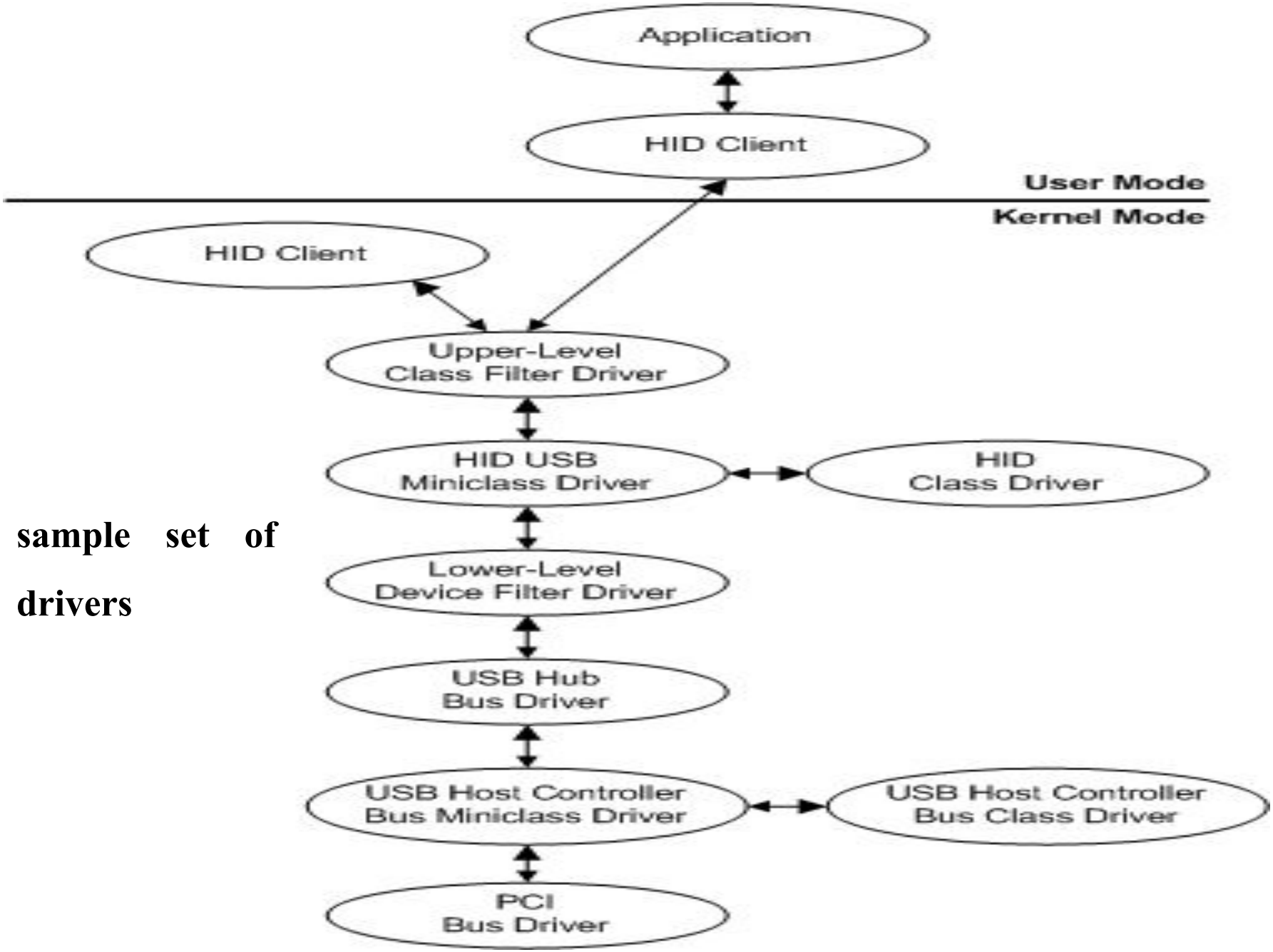
The PnP manager is actually a subsystem of the I/O manager.

A mini- driver uses an OS supplied library to abstract from the OS requirements.

Miniclass works in place of a device class driver

Class mini-driver implements device-specific details of storage class, such as tape and disk. It links against classpnp.sys. classpnp.sys implements common PNP and power support functionality.

**sample set of drivers**



Starting at the bottom of the previous figure, the drivers in the sample stack include:

A **PCI driver** that drives the PCI bus. This is a PnP bus driver. The PCI bus driver is provided with the system by Microsoft.

The **bus driver for the USB host controller** is implemented as a class/miniclass driver pair. The USB host controller class and miniclass drivers are provided with the system by Microsoft.

The **USB hub bus driver** that drives the USB hub. The USB hub driver is provided with the system by Microsoft.

Three drivers for the joystick device; one of them is a class / miniclass pair.

**The function driver**, the **main driver for the joystick device**, is the HID class driver/HID USB miniclass driver pair. (HID represents "Human Interface Device".) The HID USB miniclass driver supports the USB-specific semantics of HID devices, relying on the HID class driver DLL for general HID support.

A function driver can be specific to a particular device, or, as in the case of HID, (**Human Interface Device**) a function driver can service a group of devices.

In this example, the HID class driver/HID USB miniclass driver pair services any HID-compliant device in the system on a USB bus.

A HID class driver/HID 1394 miniclass driver pair would service any HID-compliant device on a 1394 bus.

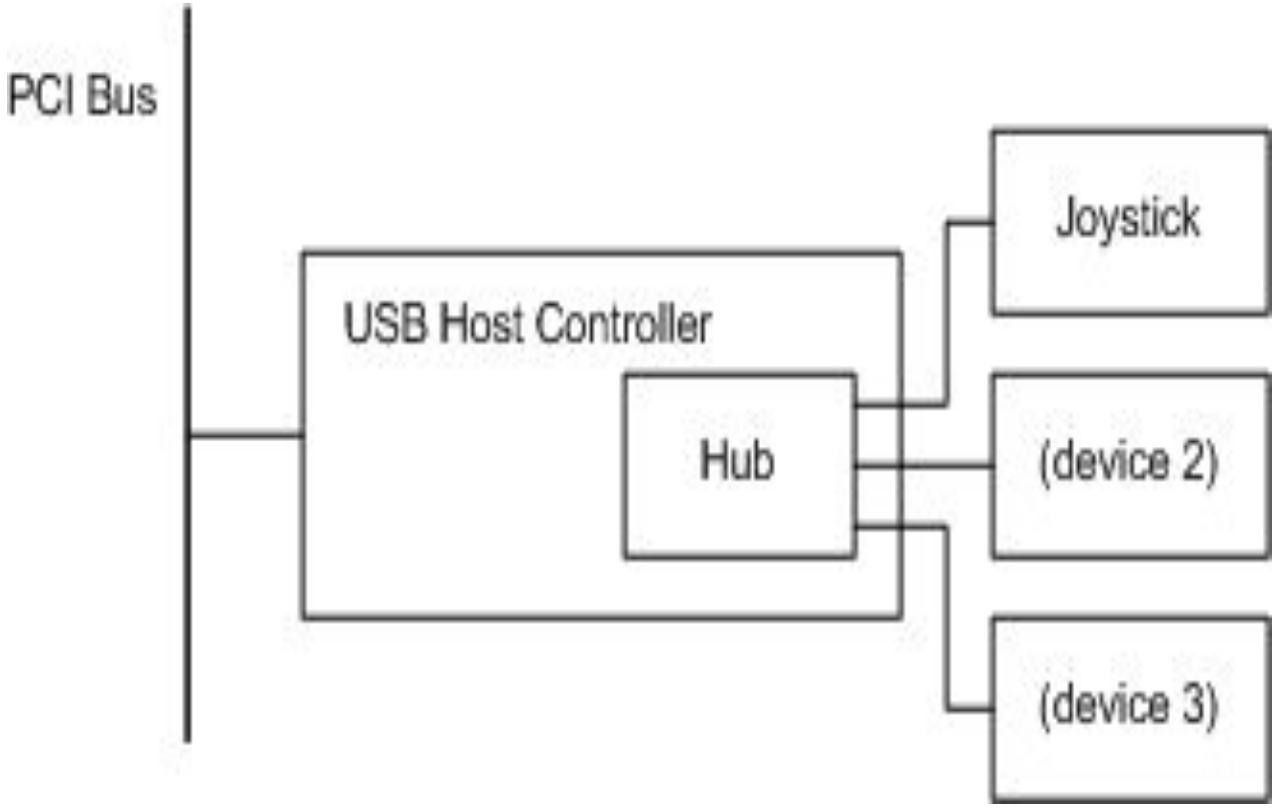
A function driver can be written by the device vendor or by Microsoft.

In this example, the function driver (the HID class/HID USB miniclass driver pair) is written by Microsoft.

There are two filter drivers for the joystick device in this example: an upper-level class filter that adds a macro button feature and a lower-level device filter that enables the joystick to emulate a mouse device.

The upper-level filter is written by someone who needs to filter the joystick I/O and the lower-level filter driver is written by the joystick vendor.

The kernel-mode and user-mode HID clients and the application are not drivers but are shown for completeness.




The figure shows a sample PnP hardware configuration for a USB joystick.

## Illustration of WDM (Windows Driver Model) Driver Layers

In this figure, the USB joystick plugs into a port on a USB hub. The USB hub in this example resides on the USB Host Controller board and is plugged into the single port on the USB host controller board. The USB host controller plugs into a PCI bus. From a PnP perspective, the USB hub, the USB host controller, and the PCI bus are all bus devices because they each provide ports. The joystick is not a bus device.

# KERNEL

In computing, the **kernel** is a computer program that manages I/O (input/output) requests from software, and translates them into data processing instructions for the central processing unit and other electronic components of a computer.

The kernel is a fundamental part of a modern computer's operating system.

i.e. A kernel connects the application software to the hardware of a computer.

**Kernel mode**, also referred to as system mode, is one of the two distinct modes of operation of the CPU (central processing unit) in Linux.

The other is **user mode**, a non-privileged mode for user programs, that is, for everything other than the kernel.

When the CPU is in kernel mode, it is assumed to be executing trusted software, and thus it can execute any instructions and reference any memory addresses (i.e., locations in memory).

The kernel (which is the core of the operating system and has complete control over everything that occurs in the system) is *trusted* software, but all other programs are considered *un-trusted* software.

Thus, all user mode software must request use of the kernel by means of a *system call* in order to perform privileged instructions, such as *process* creation or *input/output* operations.

The critical code of the kernel is usually loaded into a *protected area* of memory, which prevents it from being overwritten by other, less frequently used parts of the operating system or by applications.

The kernel performs its tasks, such as executing processes and handling interrupts, in *kernel space*, whereas everything a user normally does, such as writing text in a text editor or running programs in a GUI (graphical user interface), is done in *user space*.

This separation is made in order to prevent user data and kernel data from interfering with each other and thereby diminishing performance or causing the system to become unstable (and possibly crashing).

## The OS Kernel

- The internal part of the OS is often called the ***kernel***
- Kernel Components
  - File Manager
  - Device Drivers
  - Memory Manager
  - Scheduler
  - Dispatcher

## OS File Manager

- Maintains information about the files that are available on the system
- Where files are located in mass storage, their size and type and their protections, what part of mass storage is available.
- Files usually allowed to be grouped in ***directories*** or ***folders***. Allows hierarchical organization.

## OS Device Drivers

- Software to communicate with peripheral devices or controllers
- Each driver is unique.
- Translates general requests into specific steps for that device.

## OS Memory Manager

- Responsible for coordinating the use of the machine's main memory.
- Decides what area of memory is to be allocated for a program and its data.
- Allocates and deallocates memory for different programs and always knows what areas are free.

## OS Scheduler

- Maintains a record of processes that are present, adds new processes, removes completed processes
  - memory area(s) assigned

- priority
- state of readiness to execute (ready/wait)

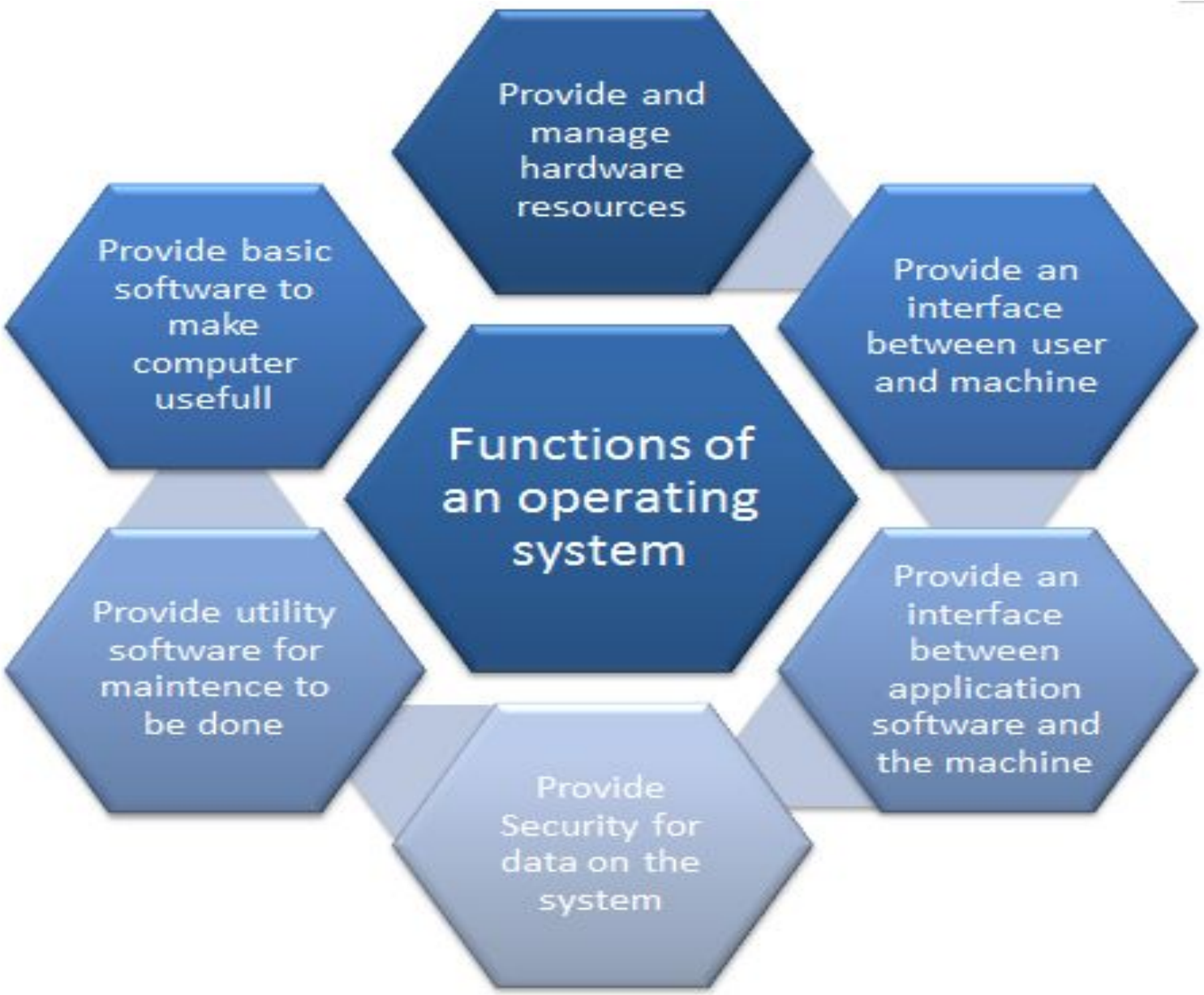
## OS Dispatcher

- Ensures that processes that are ready to run are actually executed.
- Time is divided into small (50 ms) segments called a ***time slice***
- When the time slice is over, the dispatcher allows scheduler to update process state for each process, then selects the next process to run.

## OS Summary

- Shell -- Interface to user.
- File Manager -- Manages Mass (bulk) Memory.
- Device Drivers -- Communicate with Peripherals.
- Memory Manager -- Manages Main Memory.
- Scheduler & Dispatcher -- Manage Processes.

# Functions of OS



There are Many Functions those are Performed by the Operating System But the Main Goal of Operating System is to **Provide the Interface between the user and the hardware** i.e. Provides the Interface for Working on the System by the user.

Operating System will **Manages all the Resources** of the Computer System those are attached to the System.

e.g. Memory , Processor , all the I/O Devices etc.

The Operating System will identify at which Time the CPU will perform which Operation and in which Time the Memory is used by which Programs.

Also which Input Device will respond to which Request of the user means When the Input and Output Devices are used by the which Programs.

# Storage Management

**Operating System also Controls all the Storage Operations .**

Means how the data or files will be stored into the computers and how the Files will be accessed by the users etc.

All the operations those are responsible for storing and accessing the files is determined by the Operating System .

Operating System also allows Creation of Files, Creation of Directories and Reading and Writing the data of Files and Directories and also copy the contents of the Files and the Directories from One Place to Another Place.

## Process Management

The Operating System also treats the process management means all the Processes those are given by the user or the Process those are System ‘s own Process are handled by the Operating System .

The Operating System will create the priorities for the user and also start or stops the execution of the process.

Also makes the Child Process after dividing the Large Processes into the Small Processes.

## Memory Management

Operating System also manages the memory of the Computer System means provide the memory to the process .

Also de-allocate the memory from the Process if a Process gets completed .

**Extended Machine :** Operating System also behaves like an Extended Machine means Operating system also Provides us Sharing of Files between Multiple Users, also Provides Some Graphical Environments and also Provides Various Languages for Communications and also Provides Many Complex Operations like using Many Hardware's and Software's.

**Mastermind:** Operating System also performs many functions and for those reasons we can say that Operating System is a Mastermind. It provides Booting without an Operating System and provides facility to increase the Logical Memory of the Computer System by using the Physical Memory of the Computer System and also provides various Types of Formats Like NTFS and FAT File Systems.

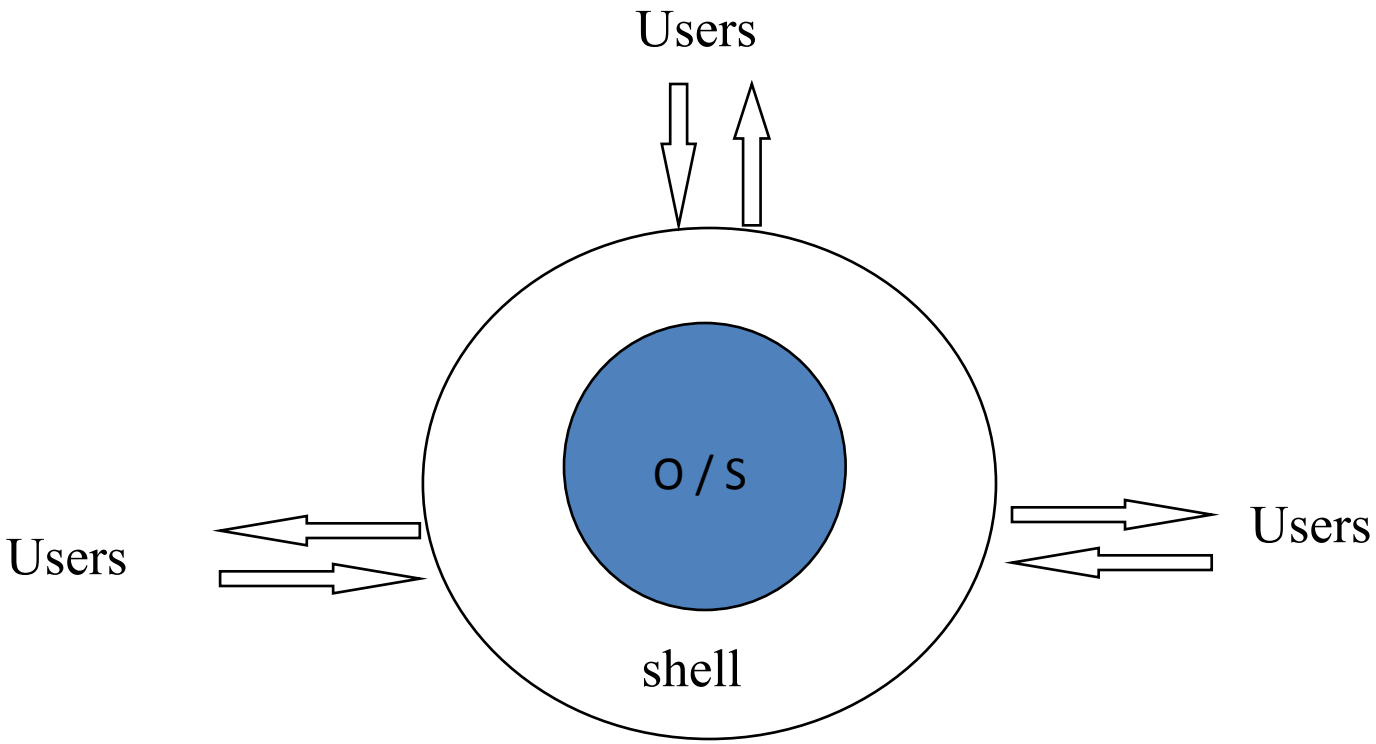
Operating System also controls the errors those have been occurred into the program and also provides recovery of the system when the system gets damaged. Means when due to some Hardware Failure , if system doesn't works properly then this recover the system and also correct the system and also provides us the Backup Facility. And Operating System also breaks the large program into the Smaller Programs those are also called as the threads. And execute those threads one by one.

Computer applications today require a single machine to perform many operations and the applications may compete for the resources of the machine.

This demands a high degree of coordination

This coordination is handled by system software known as the ***operating system***

**OS Shell interface**



**OS for batch jobs**

- Program execution required may require equipment.
- OS was a system to simplify program setup & simplify transition between jobs.
- Physical separation of users and equipment led to computer operators.

  (responsible for)
- Users left jobs with the operator and came back the next day (batch jobs).
- Users had no interaction with computer during program execution.
- Some applications may require interaction.

**OS for Interactive Processing**

- Allowed programs to carry on dialogue with user via remote terminals

  (workstations) Real-time processing
- Users demand timely response
- Machines too expensive to serve only one user.
- Common for several users to want interactive services at the same time.

A real-time operating system (**RTOS**) is an operating system (OS) intended to serve real-time application process data **as it comes in**, typically **without buffering delays**. Processing time requirements (including any OS delay) are measured in tenths of seconds or shorter.

**OS for time-sharing**

- To accommodate multiple real-time users, the OS rotates its various jobs in and out of execution via ***time-sharing.***
- Each job gets a predetermined "time slice"
- At end of time slice current job is set aside and a new one starts.
- By rapidly shuffling jobs, illusion (feeling) of several jobs executing simultaneously is created.
- Without time slicing, a computer spends most of its time waiting for peripheral devices or users.
- A collection of tasks can be completed in less time with time-sharing than when completed sequentially.

A system is said to be **Real Time** if it is required to complete it's work & deliver it's services on time. A key characteristic of an RTOS is the level of its consistency concerning the amount of time it takes to accept and complete an application's task.

**e.g. – Flight Control System [All tasks in that system must execute on time.]**

A real-time operating system **(RTOS)** is a multitasking operating system designed for real-time applications. Such applications include embedded systems, industrial robots, scientific research equipment and others.

- **Hard Real Time System**
  - Failure to meet deadlines is fatal (dangerous)
  - example : Flight Control System
- **Soft Real Time System**
  - Late completion of jobs is undesirable but not fatal.
  - System performance degrades as more & more jobs miss deadlines
  - Online Databases

## Scheduling Algorithms in RTOS

- **Clock Driven Scheduling**
- **Weighted Round Robin Scheduling**
- **Priority Scheduling**

  (Greedy / List / Event Driven)
- **Clock Driven**
  - All parameters about jobs (release time/ execution time/deadline) known in advance.
  - Schedule can be computed offline or at some regular time instances.
  - Minimal runtime overhead.
  - Not suitable for many applications.

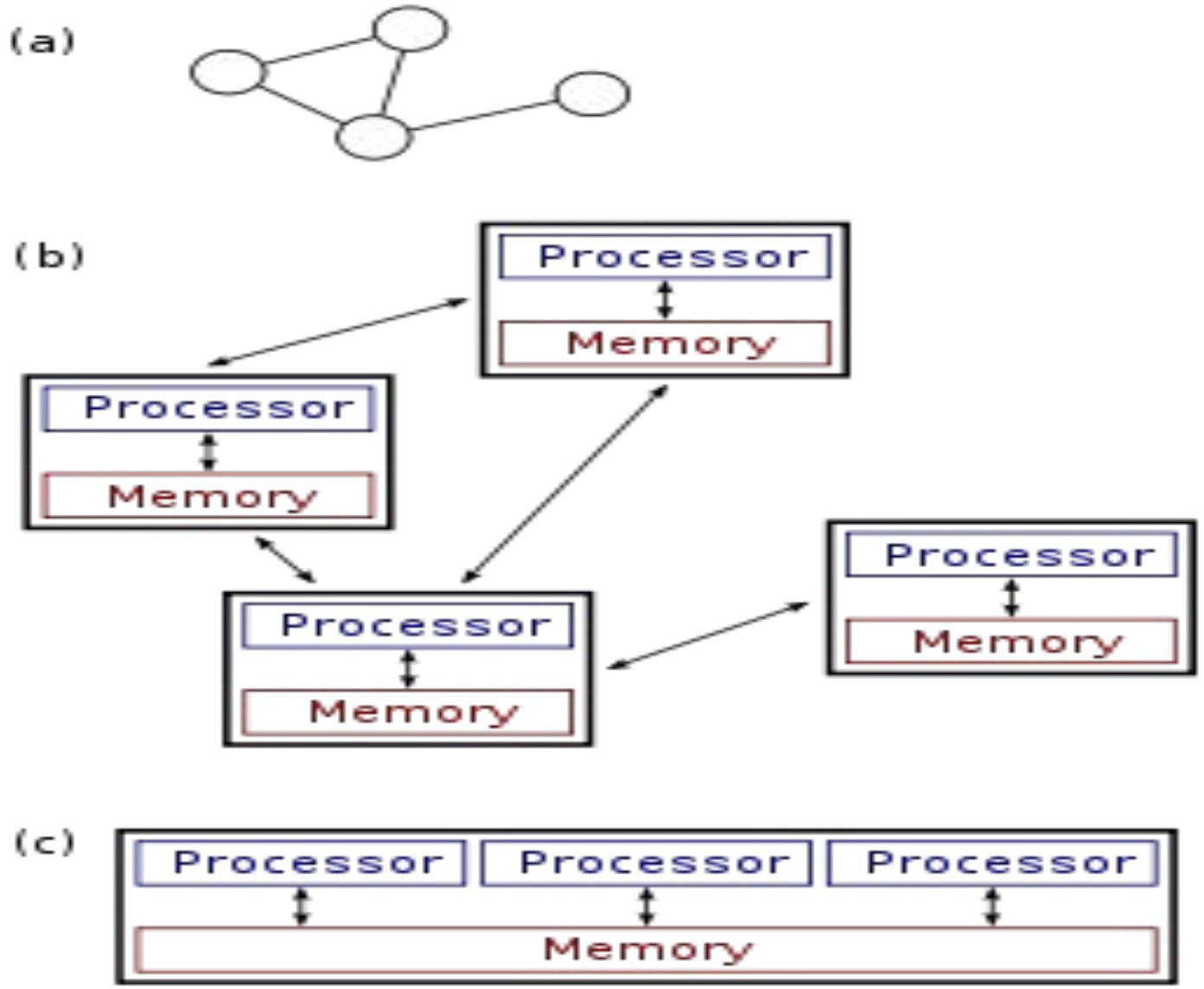
- **Weighted Round Robin**
  - Jobs scheduled in FIFO manner
  - Time quantum given to jobs is proportional to it's weight
  - Example use : High speed switching network
    - QOS guarantee.
  - Not suitable for precedence constrained jobs.
    - Job A can run only after Job B. No point in giving time quantum to Job B before Job A.

- **Priority Scheduling**

  (Greedy/List/Event Driven)

  - Processor never left idle when there are ready tasks
  - Processor allocated to processes according to priorities
  - Priorities — Static - at design time; Dynamic - at runtime

- **Earliest Deadline First (EDF)**
  - Process with earliest deadline given highest priority
- **Least Slack Time First (LSF)**
  - slack = relative deadline – execution left
- **Rate Monotonic Scheduling (RMS)**
  - For periodic tasks
  - Tasks priority inversely proportional to it's period




**a)–(b) A distributed system.**
**(c) A parallel system.**

A **distributed system is a collection of autonomous** computers linked by a computer network that appear to the users of the system as a single computer.

**Definition :** A distributed system consists of a collection of autonomous computers, connected through a network and distribution middleware, which enables computers to coordinate their activities and to share the resources of the system, so that users perceive the system as a single, integrated computing facility.

By running a distributed system software the computers are enabled to:

- coordinate their activities
- share resources: hardware, software, data.

## Centralised System Characteristics

- One component with non-autonomous parts.
- Component shared by users all the time.
- All resources accessible.
- Software runs in a single process.
- Single Point of control.
- Single Point of failure.

## Distributed System Characteristics

- Multiple autonomous components.
- Components are not shared by all users.
- Resources may not be accessible.
- Software runs in concurrent processes on different processors.
- Multiple Points of control.
- Multiple Points of failure.

## Common Characteristics

- Resource Sharing
- Openness
- Concurrency
- Scalability
- Fault Tolerance
- Transparency

## Resource Sharing

- Ability to use any hardware, software or data anywhere in the system.
- Resource manager controls access, provides naming scheme and controls concurrency.
- Resource sharing model (e.g. client / server or object-based) describing how resources are provided, they are used & provider and user interact with each other.

## Openness

- Openness is concerned with extensions & improvements of distributed systems.
- Detailed interfaces of components need to be published.
- New components have to be integrated with existing components.
- Differences in data representation of interface types on different processors (of different vendors) have to be resolved.

## Concurrency

- Components in distributed systems are executed in concurrent processes.
- Components access and update shared resources (e.g. variables, databases, device drivers).
- Integrity of the system may be violated if concurrent updates are not coordinated.
- Lost updates.
- Inconsistent analysis.

## Scalability

- Adaption of distributed systems to
  - accommodate more users.
  - respond faster (this is the hard one)
- Usually done by adding more and/or faster processors.
- Components should not need to be changed when scale of a system increases.
- Design components to be scalable!

## Fault Tolerance

- Hardware, software and networks fail!
- Distributed systems must maintain availability even at low levels of hardware/software/network reliability.
- Fault tolerance is achieved by
  - recovery
  - redundancy

## Transparency

- Distributed systems should be perceived by users and application programmers as a whole rather than as a collection of cooperating components.
- Transparency has different dimensions that were identified by ANSA.
- These represent various properties that distributed systems should have.

## Access Transparency

- Enables local and remote information objects to be accessed using identical operations.
- Example: File system operations in NFS.
- Example: Navigation in the Web.
- Example: SQL Queries

## Location Transparency

- Enables information objects to be accessed without knowledge of their location.
  e.g. File system operations in NFS
- Example: Pages in the Web
- Example: Tables in distributed databases.

## Concurrency Transparency

- Enables serveral processes to operate concurrently using shared information objects without interference between them. e.g. NFS , Automatic teller machine network , Database management system

## Replication Transparency

- Enables multiple instances of information objects to be used to increase reliability and performance without knowledge of the replicas by users or application programs
- e.g. Distributed DBMS
- e.g. Mirroring Web Pages.

## Failure Transparency

- Enables the concealment of faults
- Allows users and applications to complete their tasks despite the failure of other components.

e.g. : Database Management System

## Migration Transparency

- Allows the movement of information objects within a system without affecting the operations of users or application programs

e.g. NFS , Web Pages

## Performance Transparency

- Allows the system to be reconfigured to improve performance as loads vary.

e.g. Distributed make.

## Scaling Transparency

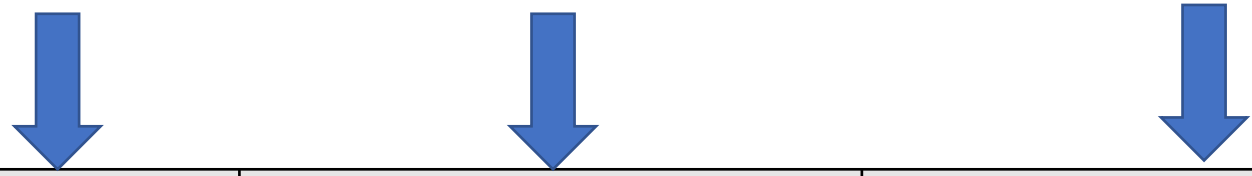
- Allows the system and applications to expand in scale without change to the system structure or the application algorithms.

e.g. World-Wide-Web , Distributed Database

# OPERATING SYSTEM

By :- Shripad Bhide

**Execution Environment :- No. 1** **No. 2** **No. 3**



| **Processes** | **Processes** | **Processes** |
|---|---|---|
| | | |
| Kernel - 1 | Kernel - 2 | Kernel - 3 |
| (Fedora) | (Ubuntu) | (Windows) |
| Virtual Machine 1 | Virtual Machine 2 | Virtual Machine 3 |
| | | |
| Virtual Machine Implementation | | |
| (Base Windows - VM) | | |
| Hardware (of Physical Machine) | | |

When Hardware (CPU, Memory, NIC , Disk Drive etc…) of a single machine / computer used in several different execution environment , creating illusion (seems)  that each separate execution environment is running its own private execution.

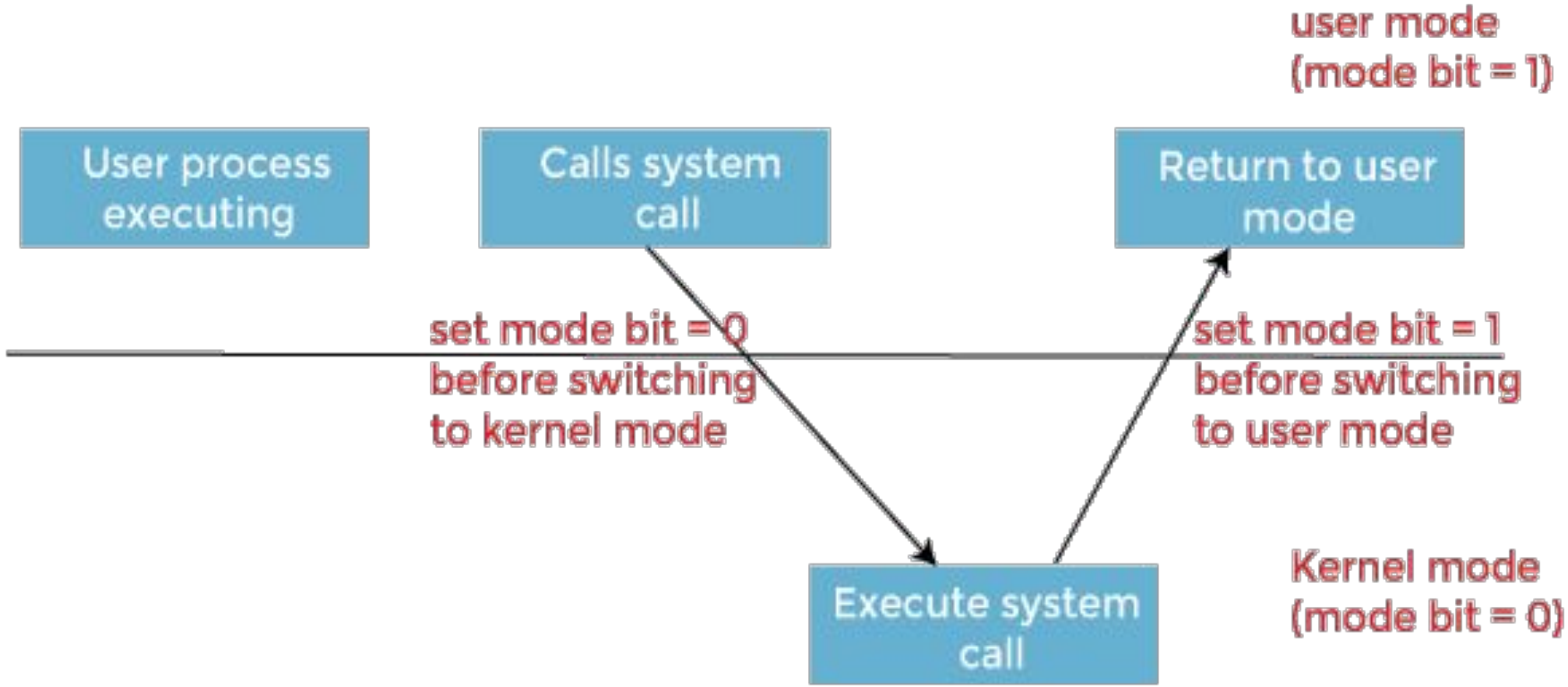
**User Mode**

When we are working with user applications like  word, excel, power-point or any application program, then the system is in user mode.

**Kernel Mode**

When the user application is in demand for a service from the operating system or an interrupt occurs or system call, then there will be a transition from user to kernel mode to fulfill the requests.  When the system boots, the hardware starts in kernel mode and when the operating system is loaded, it starts user application in user mode.

To provide protection to the hardware, we have privileged instructions which execute only in kernel mode.

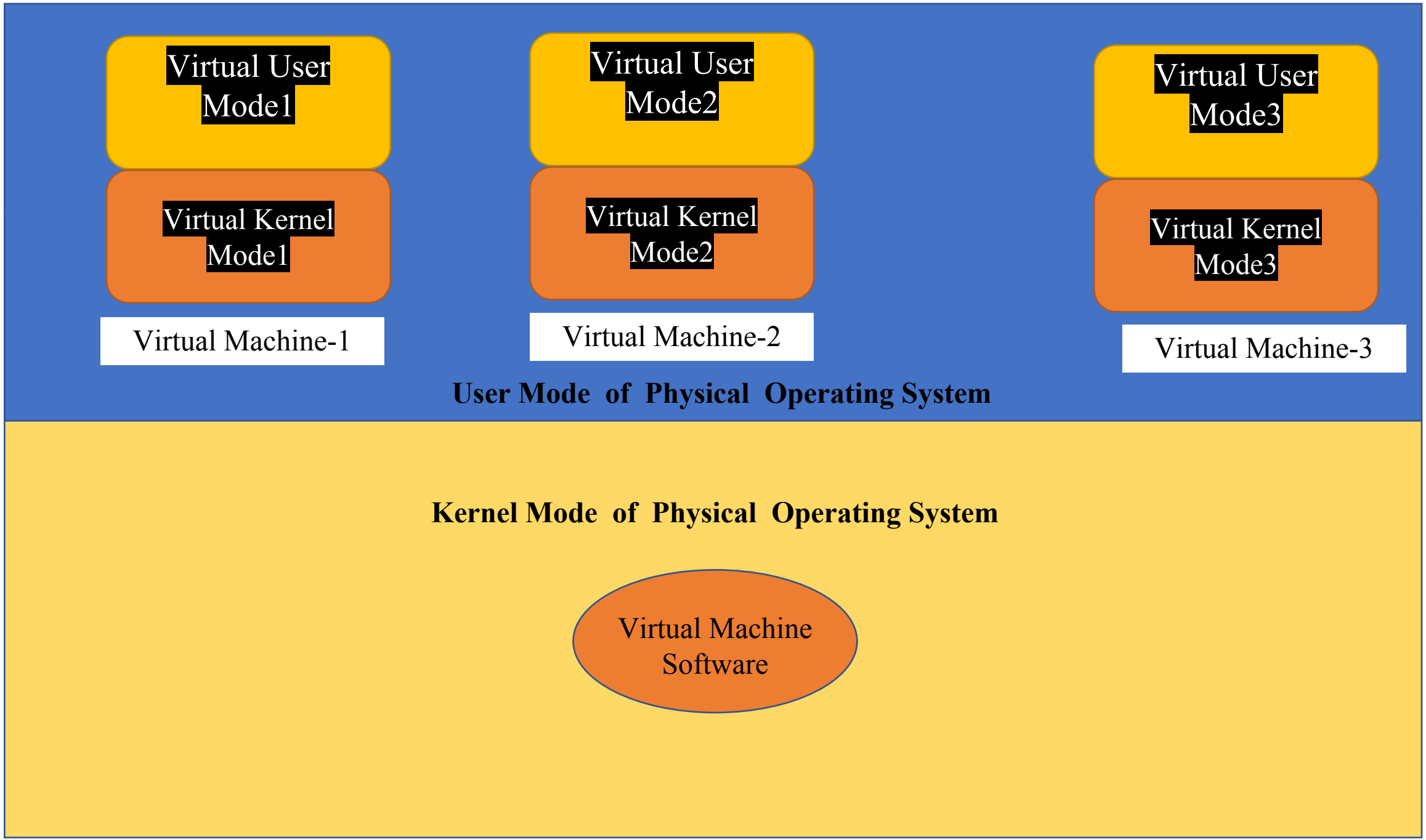

user mode
(mode bit = 1)
User process executing
Calls system call
Return to user mode
set mode bit = 0
before switching
to kernel mode
set mode bit = 1
before switching
to user mode
Execute system call
Kernel mode
(mode bit = 0)

Virtual Machine Software runs in Kernal Mode. But Virtual Machine Itself runs in User Mode. (of Physical system)

Each Virtual Machine having its own Virtual User Mode and Virtual Kernel Mode and both runs in Physical User Mode.

Isolation of each virtual machine w.r.t basic hardware is maintained. Part of a disk is created is called Mini Disk. So parallel working with disk is possible.

e.g. Virtual Box , Vmware etc.


Virtual User Mode1
Virtual Kernel Mode1
Virtual Machine-1
Virtual User Mode2
Virtual Kernel Mode2
Virtual Machine-2
Virtual User Mode3
Virtual Kernel Mode3
Virtual Machine-3
User Mode of Physical Operating System
Kernel Mode of Physical Operating System
Virtual Machine Software

**Kernel :-** Central component of an OS, Heart of an OS , Interface between user and software system.

It is an interface between Hardware components and software applications.

Applications like games , video, photos or audio editing , Network analysis etc. We install different softwares. e.g. Audio file require software for altering speakers , Gams required joysticks to work , for Photo editing photo-shop i.e. software demands hardware to function.

Kernel allows data to flow to the memory. It also decides order of program execution.

Separate space in memory is for kernel functions.

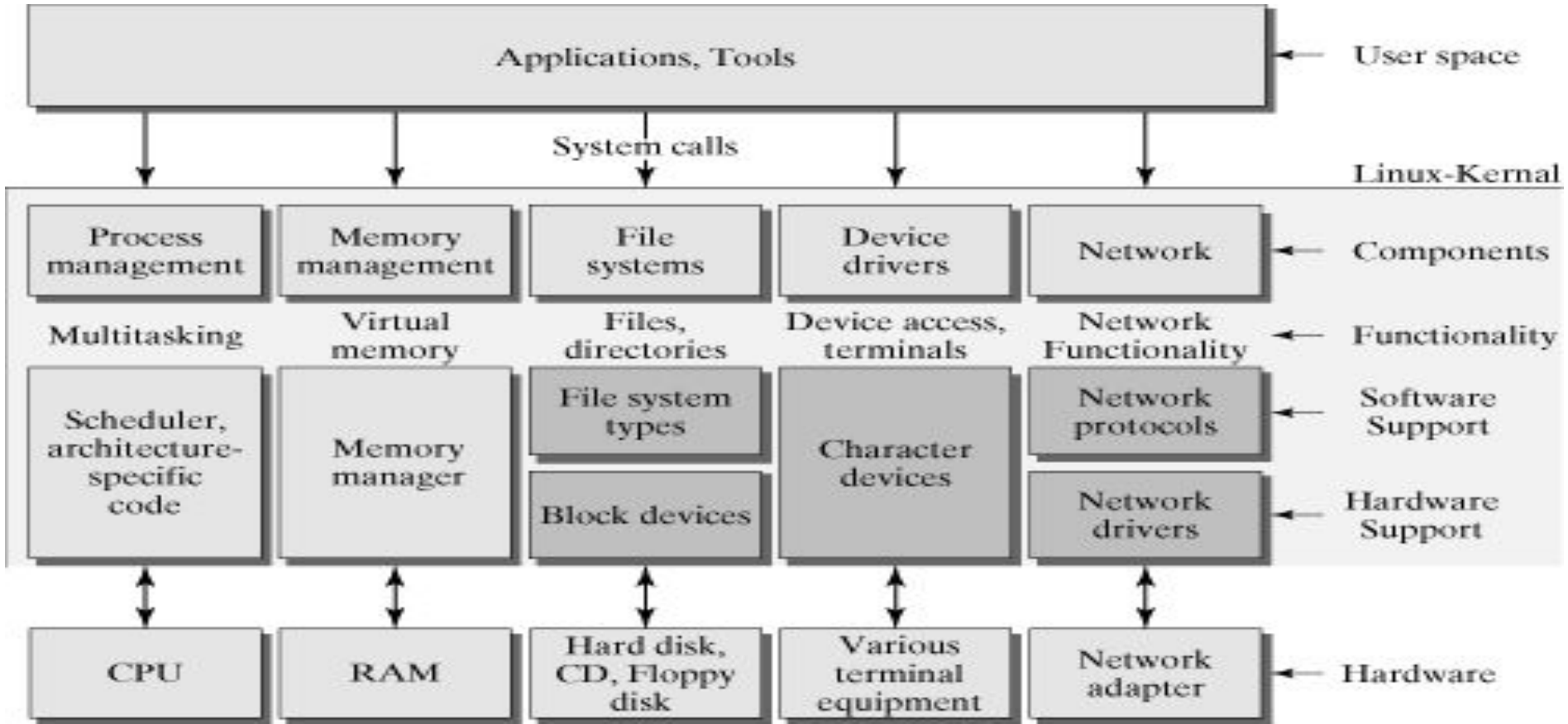
If user data **interferes** with kernel data System fails.

**Process demands resources from kernel is system call.**

A kernel is a central component of an operating system. It acts as an interface between the user applications and the hardware. The sole aim of the kernel is to manage the communication between the software (user level applications) and the hardware (CPU, disk memory etc). The main tasks of the kernel are :

## Types Of Kernels

Kernels may be classified mainly in two categories :- i) Monolithic ii) Micro Kernel



There is a difference between kernel and OS. Kernel as described above is the heart of OS which manages the core features of an OS while if some useful applications and utilities are added over the kernel, then the complete package becomes an OS. So, it can easily be said that an operating system consists of a kernel space and a user space.

We can say that Linux is a kernel as it does not include applications like file system utilities, windowing systems and graphical desktops, system administrator commands, text editors, compilers etc. So , various companies add these kind of applications over linux kernel and provide their operating system like ubuntu, suse, centOS, redHat etc.

**Monolithic Kernel**

In monolithic Kernels both user services and the kernel services are implemented in the same memory space. By doing this, the size of the Kernel as well as the size of the Operating System is increased. As there is no separate User Space and Kernel Space, so the execution of the process will be faster in Monolithic Kernels.

**Advantages :-**

- It provides CPU scheduling, memory scheduling, file management through System calls.
- Execution of the process is fast as there is no separate space.

**Disadvantages :-**

- If the service fails, then the system failure happens.
- If you try to add new services then the entire Operating System needs to be modified.

**Microkernel :-**

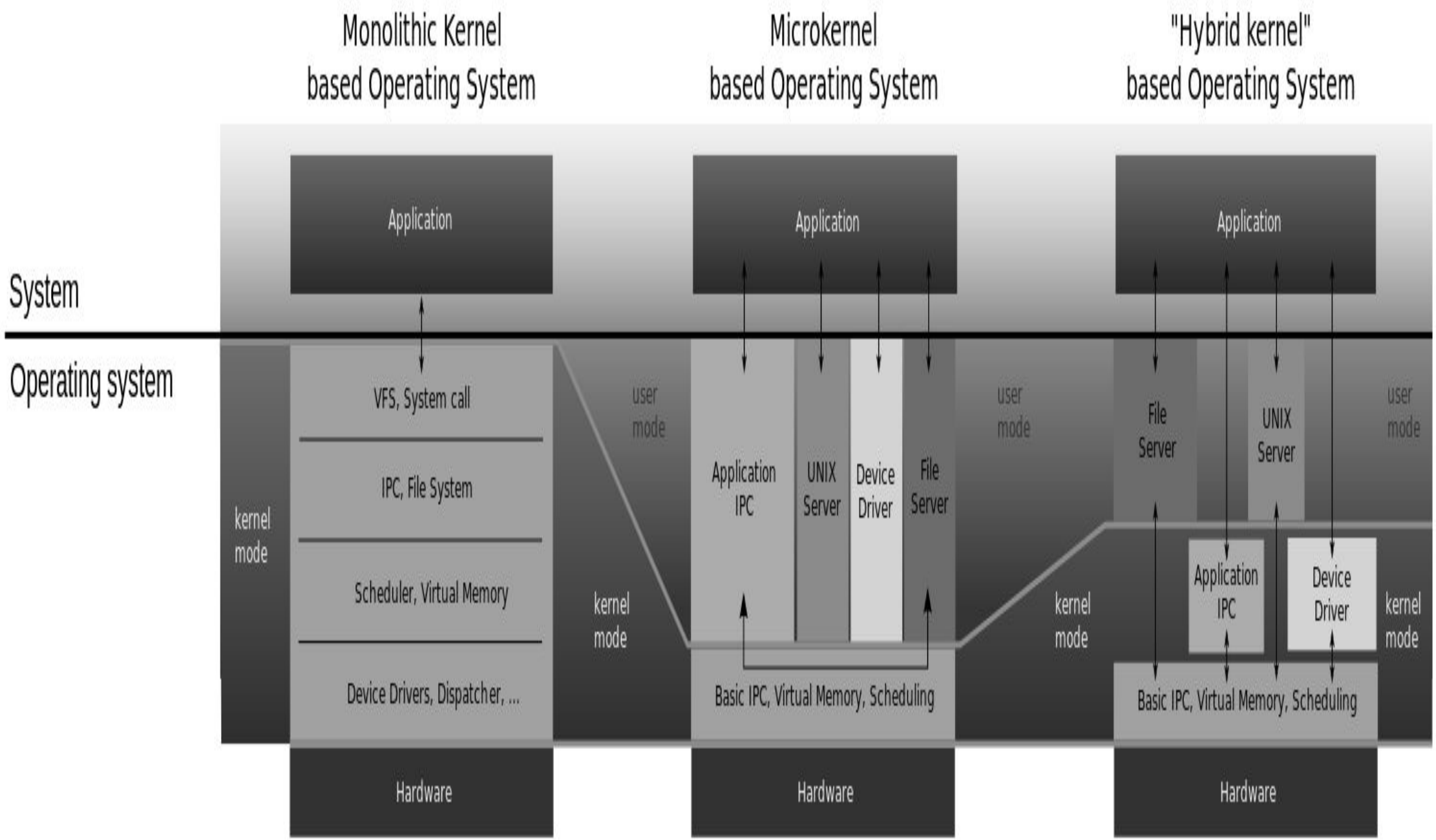
A Microkernel is not the same as the Monolithic kernel. It is a little bit different because in a Microkernel, the user services and kernel services are implemented into different spaces. Because of using User Space and Kernel Space separately, it reduces the size of the Kernel and in turn, reduces the size of the Operating System.

As we are using different spaces for user and kernel service, the communication between application and services is done with the help of message parsing because of this it reduces the speed of execution.

The **advantage** of microkernel is that it can easily add new services at any time.

The **disadvantage** of microkernel is that here we are using User Space and Kernel Space separately. So, the communication between these can reduce the overall execution time.

**Remaining types - Hybrid Kernel , Nanokernel , Exokernel**

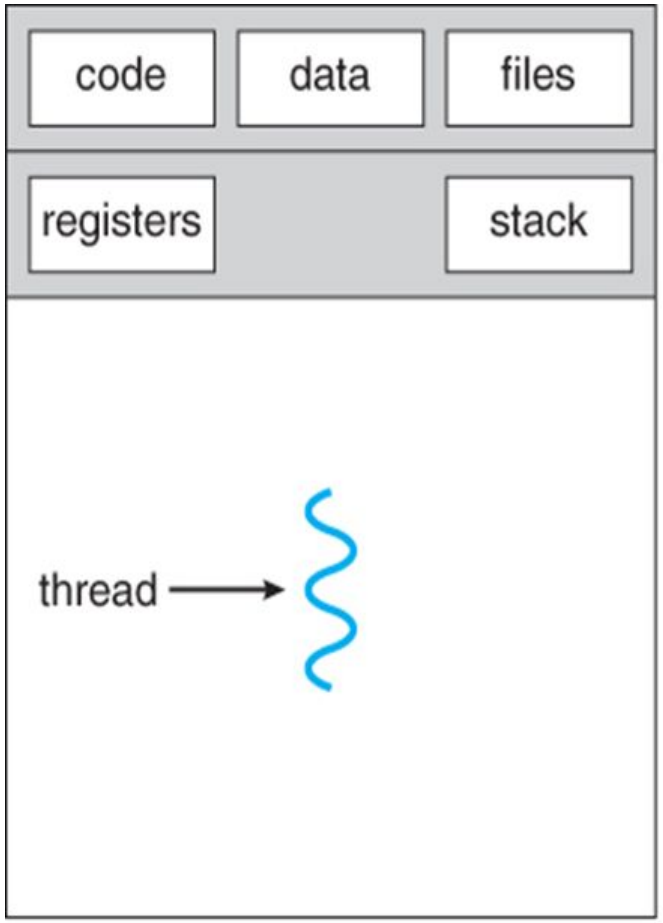

Monolithic Kernel
based Operating System
Microkernel
based Operating System
"Hybrid kernel"
based Operating System
System
Operating system
Application
Application
Application
VFS, System call
IPC, File System
Scheduler, Virtual Memory
Device Drivers, Dispatcher, ...
kernel mode
user mode
kernel mode
Application IPC
UNIX Server
Device Driver
File Server
Basic IPC, Virtual Memory, Scheduling
user mode
kernel mode
File Server
UNIX Server
Application IPC
Device Driver
Basic IPC, Virtual Memory, Scheduling
user mode
kernel mode
Hardware
Hardware
Hardware

A process has a self contained execution environment that means it has a complete, private set of basic run time resources particularly each process has its own memory space. Threads exist within a process and every process has at least one thread

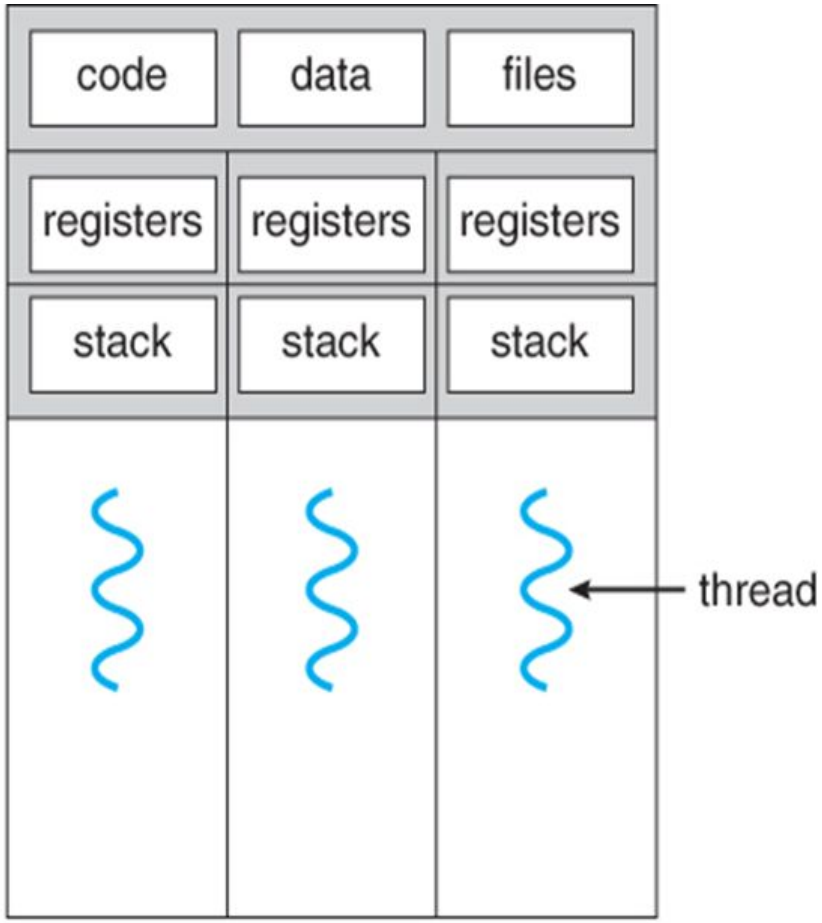
Each process provides the resources needed to execute a program. Each process is started with a single thread, known as the primary thread. A process can have multiple threads in addition to the primary thread.

On a multiprocessor system, multiple processes can be executed in parallel. Multiple threads of control can exploit the true parallelism possible on multiprocessor systems.

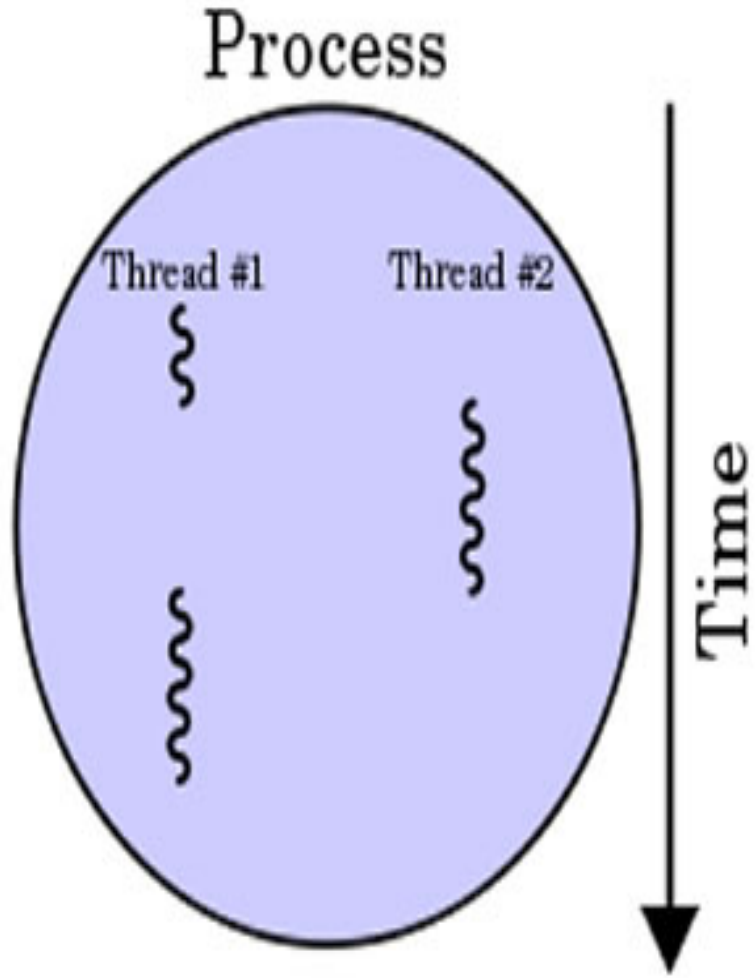
Processes are heavily dependent on system resources available while threads require minimal amounts of resource, so a process is considered as heavyweight while a thread is termed as a lightweight process.



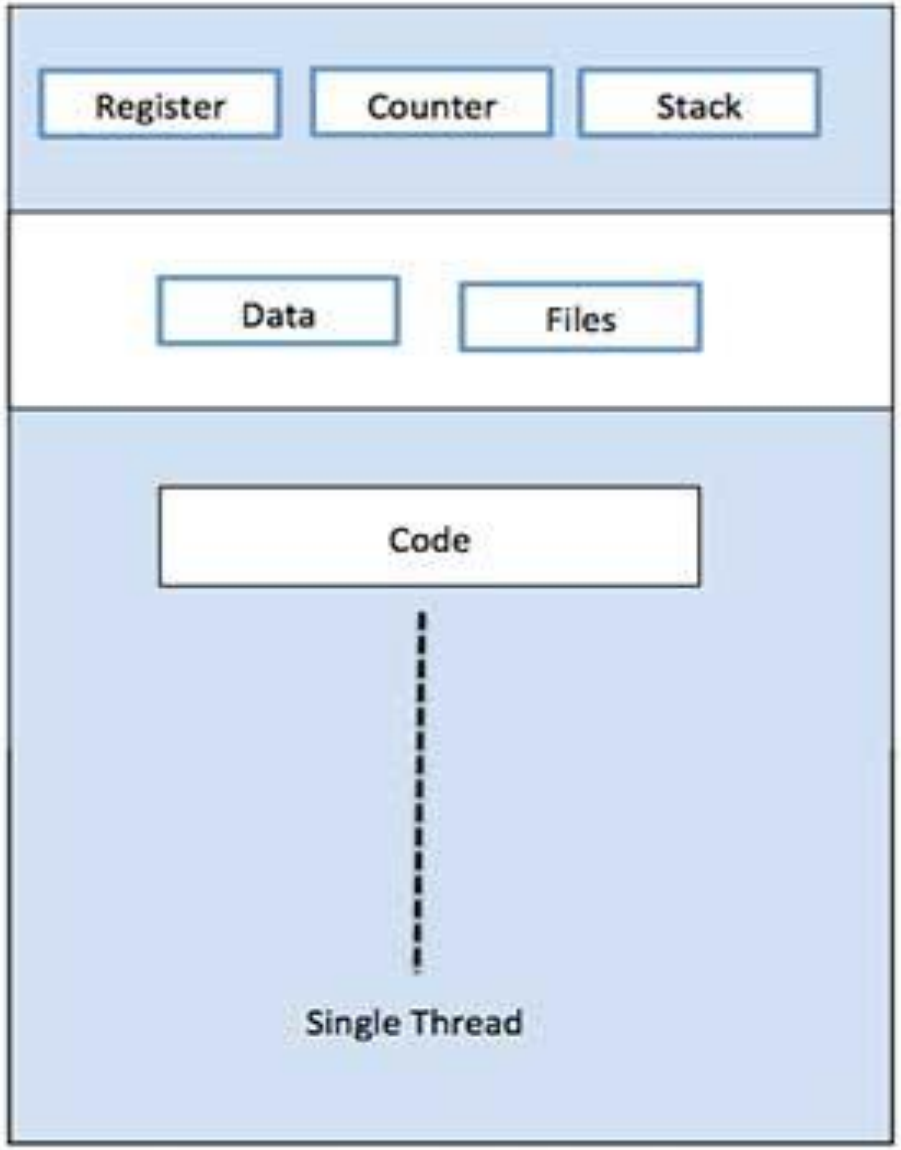

single-threaded process



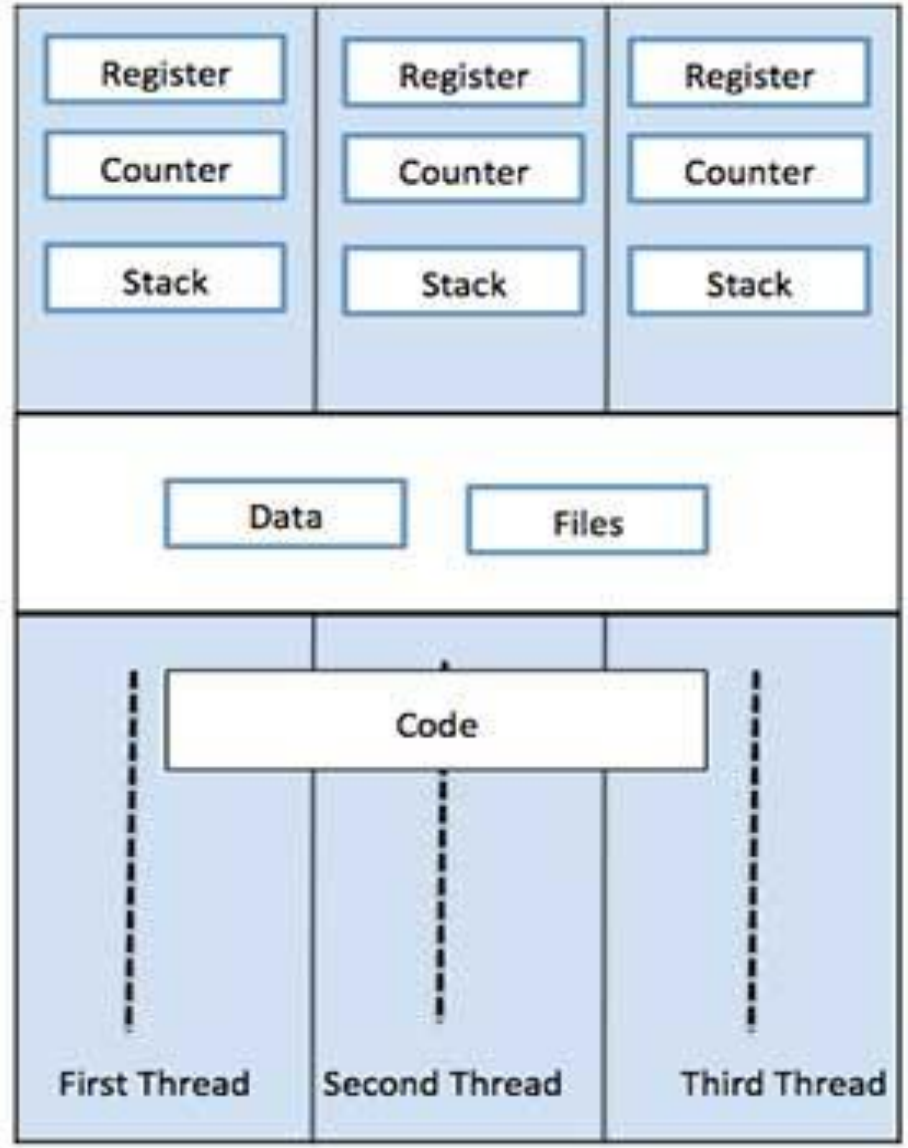

multithreaded process



- A thread is a basic unit of CPU utilization, consisting of a program counter, a stack, and a set of registers, ( and a thread ID. )
- Traditional ( heavyweight ) processes have a single thread of control - There is one program counter, and one sequence of instructions that can be carried out at any given time.
- Multi-threaded applications have multiple threads within a single process, each having their own program counter, stack and set of registers, but sharing common code, data, and certain structures such as open files.




Single Process P with single thread



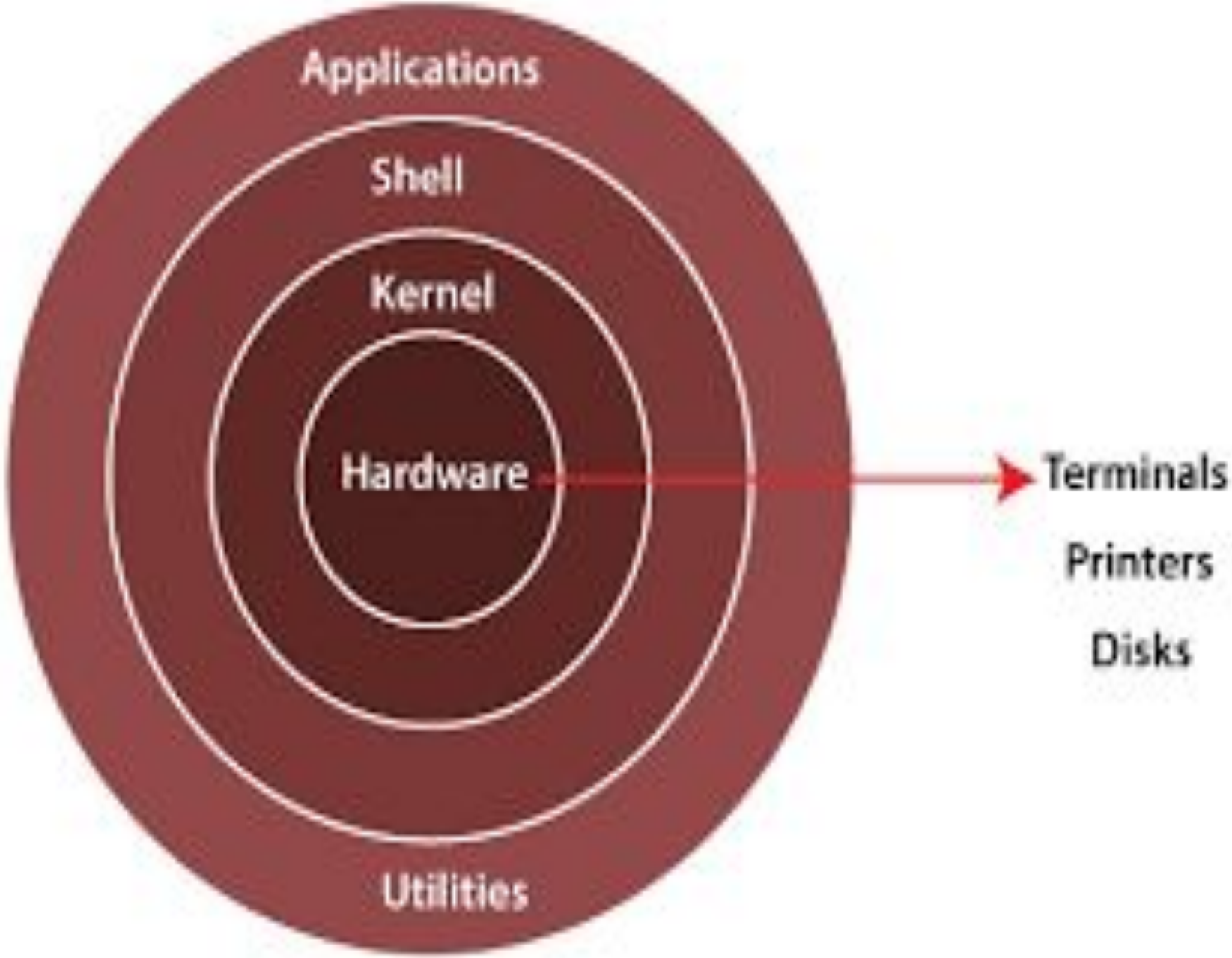

Single Process P with three threads

The shell (computer program) is the Linux command line interpreter. It provides an interface between the user and the kernel and executes programs called commands. Shells allow users to communicate efficiently and directly with their operating systems.

e.g. If a user enters [ls] then the shell executes the [ls] command.

**Different Types of Shells in Linux**

- The Bourne Shell (sh)
- The GNU Bourne-Again Shell (bash) - invented by Steven Bourne
- The C Shell (csh)
- The Korn Shell (ksh)
- The Z Shell (zsh)


Applications
Shell
Kernel
Hardware
Utilities
Terminals
Printers
Disks

The root directory denoted by the slash

An **Absolute Path** is a full path specifying the location of a file or directory from the root directory or start of the actual filesystem.

**Example:** /home/javatpoint/Desktop/CollegeStudent

**The relative path** of a file is its location relative to the current working directory. It never starts with a slash (/). It begins with the ongoing work directory.

**Desktop/CollegeStudent**

```
#!/bin/bash
# A shell script to find the factorial of a number

read -p "Enter a number" num

fact = 1
for((i=2;i<=num;i++))
{
  fact=$((fact*i))
}

echo $fact
```

```
#!/bin/bash
# A shell script to find the factorial of a number

read -p "Enter a number" num
fact=1
while [ $num -gt 1 ]
do
  fact=$((fact*num))
  num=$((num-1))
done

echo $fact
```

Bash is a command-line interpreter or Unix Shell and it is widely used in GNU/Linux Operating System. It is written by Brian Jhan Fox.

The first line of our script file will be – #!/bin/bash

**This will tell, the system to use Bash for execution. Then we can write our own scripts.**

#!/bin/bash echo "Hello, GeeksforGeeks"

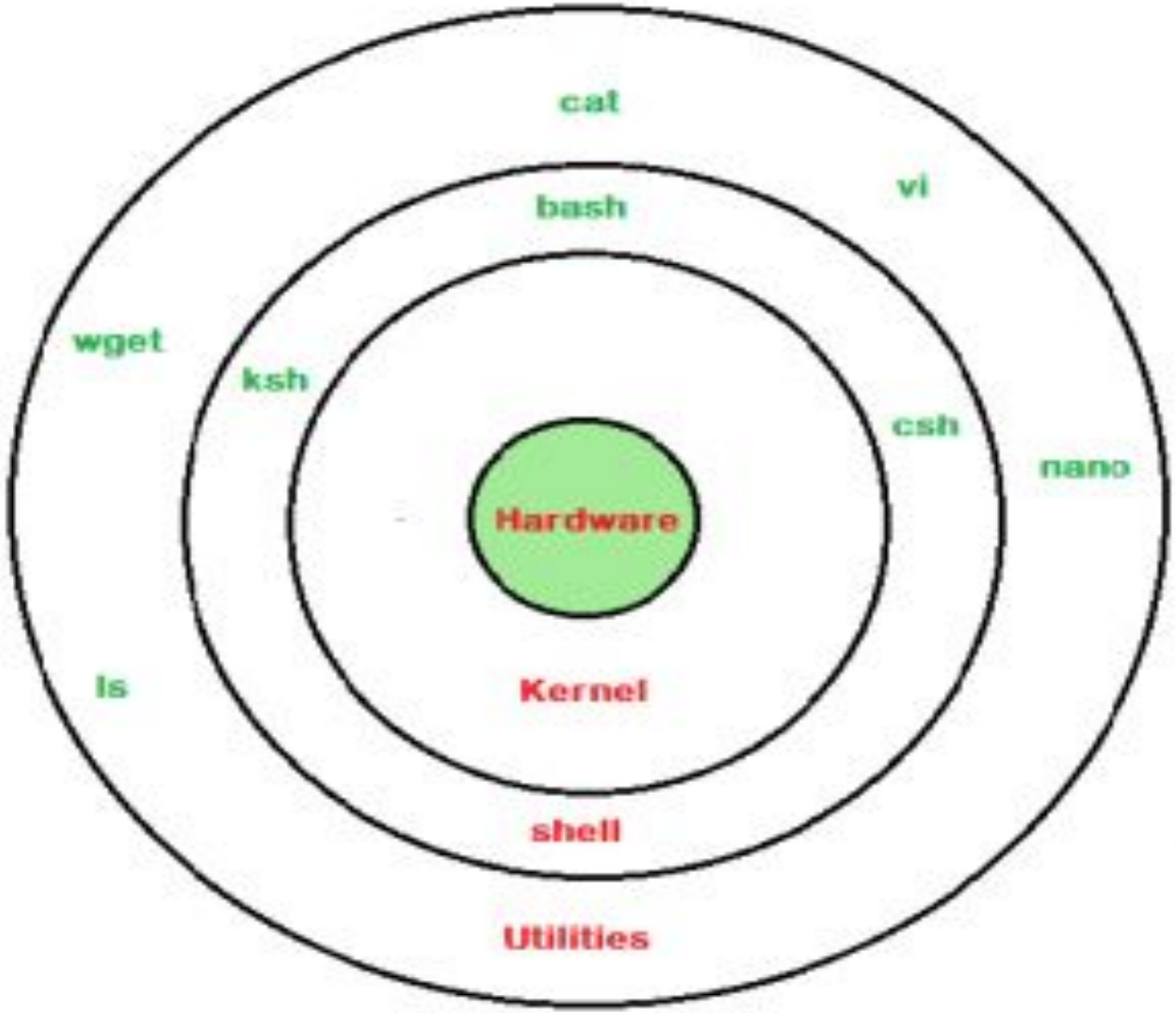
```
#!/bin/bash
Age = 17
if [ "$Age" -ge 18 ] ; then
    echo "You can vote"
else
    echo "You cannot vote" fi
```

To create and write a file with the .sh extension we can use **gedit text editor**.

The command for it will be – gedit scriptname.sh

| Character | Meaning |
| --- | --- |
| ~ | Home directory |
| ` | Command substitution (archaic) |
| # | Comment |
| $ | Variable expression |
| & | Background job |
| * | String wildcard |
| ( | Start subshell |
| ) | End subshell |
| \ | Quote next character |
| \| | Pipe |
| [ | Start character-set wildcard |
| ] | End character-set wildcard |

| Character | Meaning |
| --- | --- |
| { | Start command block |
| } | End command block |
| ; | Shell command separator |
| ' | Strong quote |
| <"> | Weak quote |
| < | Input redirect |
| > | Output redirect |
| / | Pathname directory separator |
| ? | Single-character wildcard |
| ! | Pipeline logical NOT |


cat
vi
bash
wget
ksh
csh
nano
Hardware
ls
Kernel
shell
Utilities

**Unit II** **(8 Hrs)**

**Operating Systems : Methodologies for implementation of O/S service system calls,**

system programs, Interrupt mechanisms.

**Process - Concept of process and threads, Process states, Process management, Context switching**

Interaction between processes and OS Multithreading

Process Control, Job schedulers, Job Scheduling,

scheduling criteria, scheduling algorithms

A **system call** is how a program requests a service from an operating system's kernel.

This may include hardware-related services (for example, accessing a hard disk drive), creation and execution of new processes, and communication with integral kernel services such as process scheduling.

System calls provide an essential interface between a process and the operating system.

System calls can be roughly grouped into five major categories:

1. Process Control

    a) load  b) execute  c) end, abort  d) create process

    e) terminate process  f) get/set process attributes

    g) wait for time, wait event, signal event

    h) allocate, free memory

2. File management
   a) create file, delete file  b) open, close  c) read, write, reposition
   d) get/set file attributes
3. Device Management
   a) request device, release device  b) read, write, reposition
   c) get/set device attributes  d) logically attach or detach devices
4. Information Maintenance
   a) get/set time or date  b) get/set system data
   c) get/set process, file, or device attributes
5. Communication
   a) create, delete communication connection
   b) send, receive messages
   c) transfer status information
   d) attach or detach remote devices

System calls provide an interface between the process and the operating system. System calls allow user-level processes to request some services from the operating system which process itself is not allowed to do.

In handling the trap, the operating system will enter in the kernel mode, where it has access to privileged instructions, and can perform the desired service on the behalf of user-level process.

It is because of the critical nature of operations that the operating system itself does them every time they are needed.

e.g. for I/O a process involves a system call telling the operating system to read or write particular area and this request is satisfied by the operating system.

System programs provide basic functioning to users so that they do not need to write their own environment for program development (editors, compilers) and program execution (shells).

# Process States

A process is a program in execution.

The execution of a process must progress in a sequential fashion.

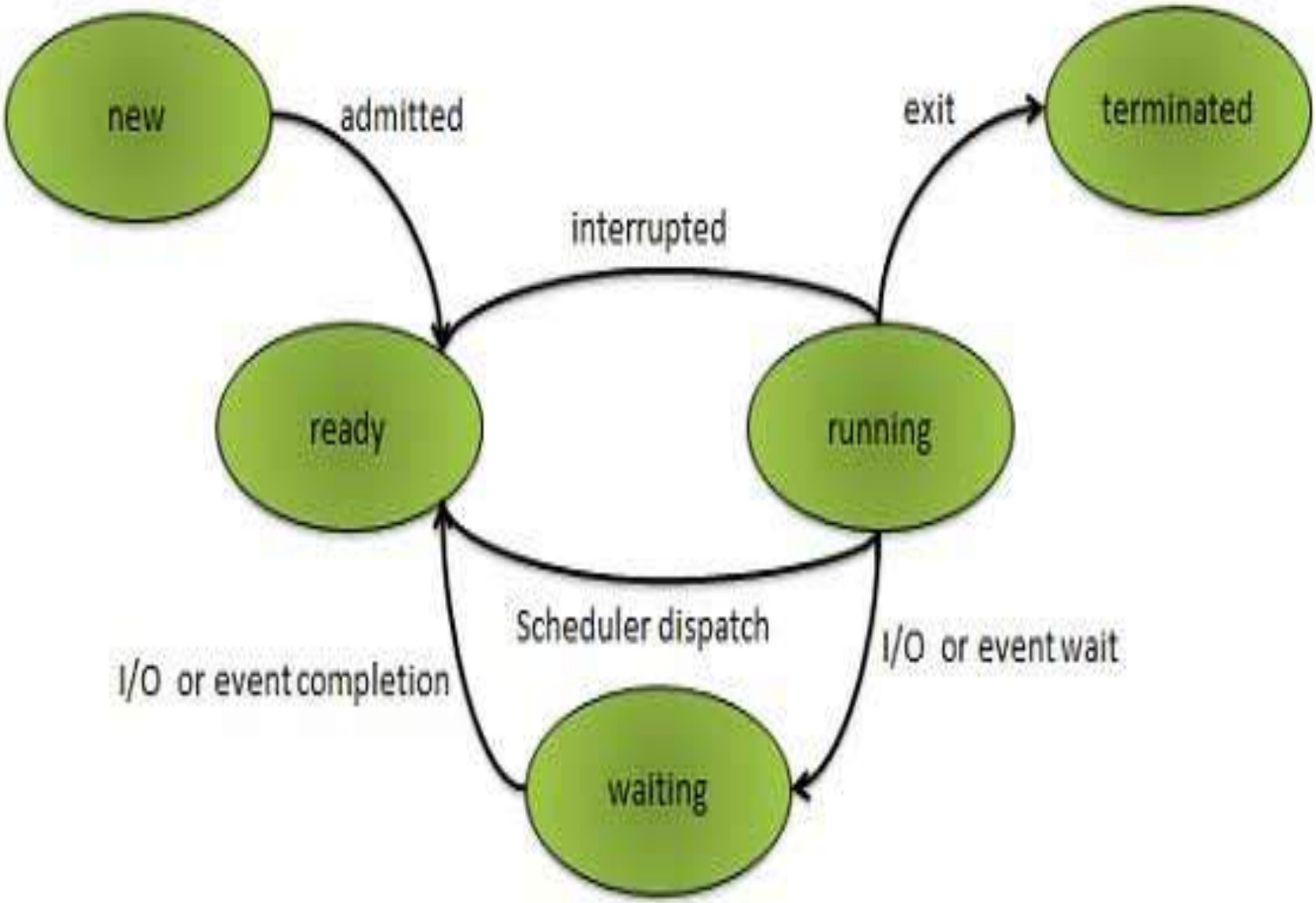
A process is defined as an entity which represents the basic unit of work to be implemented in the system.

**Components of Process :-**

1. Object Program :- Code to be executed.
2. Data :- Data to be used for executing the program.
3. Resources :- While executing the program, it may require some resources.
4. Status :- Verifies the status of the process execution. A process can run to completion only when all requested resources have been allocated to the process. Two or more processes could be executing the same program, each using their own data and resources.

A process goes through a series of discrete process states.

1) **New State** The process being created.
2) **Ready State** .A The process is waiting to be assigned to a processor. Ready processes are waiting to have the processor allocated to them by the operating system so that they can run. It is runnable but temporarily stopped to let another process run. Logically, the 'Running' and 'Ready' states are similar.
3) **Running State** A process is said t be running if it currently has the CPU, i.e. - actually using the CPU at that particular instant.
4) **Blocked (waiting) State** When a process blocks, it does so because logically it cannot continue, typically because it is waiting for input that is not yet available. Formally, a process is said to be blocked if it is waiting for some event to happen (such as an I/O completion) before it can proceed. In this state a process is unable to run until some external event happens.


new
admitted
exit
terminated
interrupted
ready
running
Scheduler dispatch
I/O or event completion
I/O or event wait
waiting

**5. Terminated State** The process has finished execution.

In the case of 'Ready' state, there is temporarily no CPU available for it.

The 'Blocked' state is different from the 'Running' and 'Ready' states in that the process cannot run, even if the CPU is available.

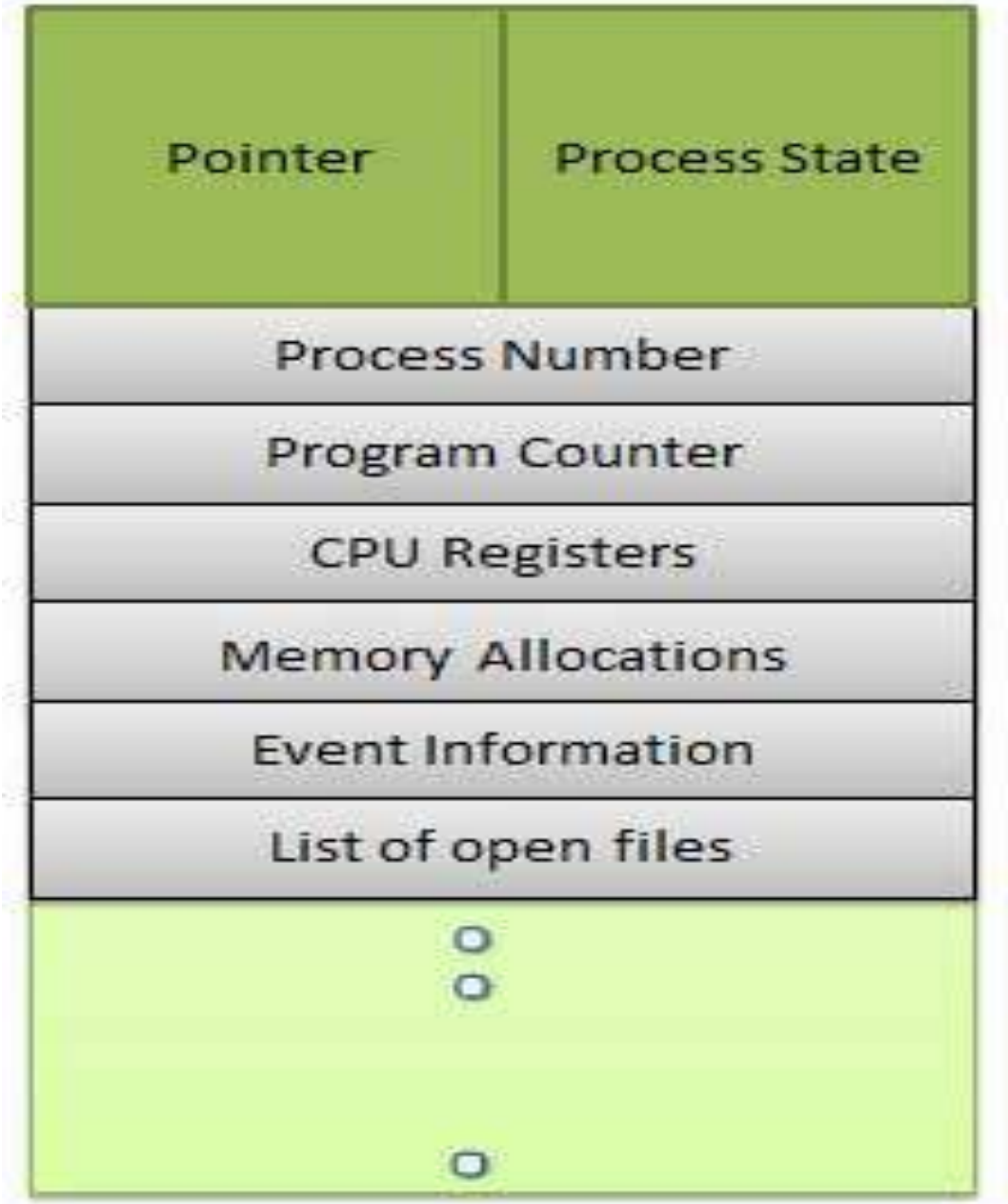
## Process Control Block, PCB

Each process is represented in the operating system by a process control block (PCB) also called a task control block.

PCB is the data structure used by the operating system.

Operating system groups all information that needs about particular process.

PCB contains many pieces of information associated with a specific process which are described below.




**Process Control Block, PCB**

**Pointer**

Pointer points to another process control block. Pointer is used for maintaining the scheduling list.

**Process State**

Process state may be new, ready, running, waiting and so on.

**Program Counter**

Program Counter indicates the address of the next instruction to be executed for this process.

**CPU registers**

CPU registers include general purpose register, stack pointers, index registers and accumulators etc. number of register and type of register totally depends upon the computer architecture.

**Memory management information**

This information may include the value of base and limit registers, the page tables, or the segment tables depending on the memory system used by the operating system. This information is useful for de-allocating the memory when the process terminates.

**Accounting information**

This information includes the amount of CPU and real time used, time limits, job or process numbers, account numbers etc.

**Process Control Block includes :-**

CPU scheduling, I/O resource management, File management information etc..

The PCB serves as the repository for any information which can vary from process to process.

Loader/linker sets flags and registers when a process is created.

If that process get suspended, the contents of the registers are saved on a stack and the pointer to the particular stack frame is stored in the PCB.

By this technique, the hardware state can be restored so that the process can be scheduled to run again.

**In Short - it's a data structure holding:**

· PC, CPU registers,

· memory management information,

· accounting ( time used, ID, ... )

· I/O status ( such as file resources ),

· scheduling data ( relative priority, etc. )

· Process State (so running, suspended, etc. is simply a field in the PCB ).

# SCHEDULING

The process scheduling is the activity of the process manager that handles the removal of the running process from the CPU and the selection of another process on the basis of a particular strategy.

Process scheduling is an essential part of a **Multiprogramming operating system**. Such operating systems allow more than one process to be loaded into the executable memory at a time and loaded process shares the CPU using time multiplexing.

The act of **Scheduling a process means changing the active PCB pointed to by the CPU.** Also called a **context switch.**

A context switch is essentially the same as a process switch - it means that the memory, as seen by one process is changed to the memory seen by another process.

# Scheduling Queues

Scheduling queues refers to queues of processes or devices. When the process enters into the system, then this process is put into a job queue. This queue consists of all processes in the system. The operating system also maintains other queues such as device queue. Device queue is a queue for which multiple processes are waiting for a particular I/O device. Each device has its own device queue.

(Process is driven by events that are triggered by needs and availability )

·Ready queue = contains those processes that are ready to run.

·I/O queue (waiting state ) = holds those processes waiting for I/O service.

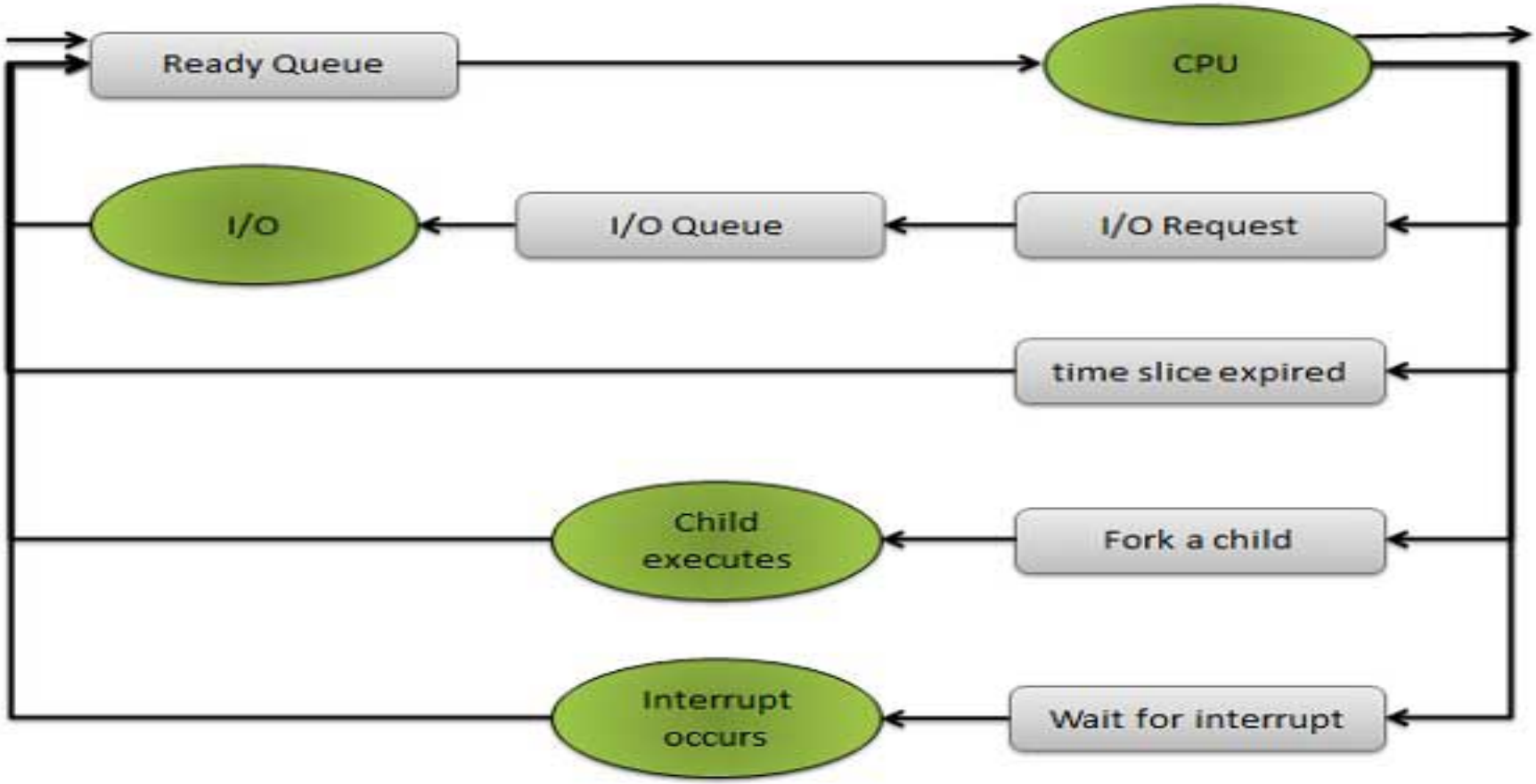
What do the queues look like? They can be implemented as single or double linked.

This figure shows the queuing diagram of process scheduling.

•Queue is represented by rectangular box.

•The circles represent the resources that serve the queues.

•The arrows indicate the process flow in the system



**Queues are of two types :-** **1) Ready queue 2) Device queue**

A newly arrived process is put in the ready queue.

Processes waits in ready queue for allocating the CPU.

Once the CPU is assigned to a process, then that process will execute.

While executing the process, any one of the following events can occur.

The process could issue an I/O request and then it would be placed in an I/O queue.

The process could create new sub process and will wait for its termination.

The process could be removed forcibly from the CPU, as a result of interrupt and put back in the ready queue.

**Schedulers**

Schedulers are special system softwares which handles process scheduling in various ways. Their main task is to select the jobs to be submitted into the system and to decide which process to run.

Schedulers are of three types

- Long Term Scheduler
- Short Term Scheduler
- Medium Term Scheduler

## LONG TERM SCHEDULER

It is also called job scheduler. Long term scheduler determines which programs are admitted to the system for processing. Job scheduler selects processes from the queue and loads them into memory for execution. Process loads into the memory for CPU scheduling.

The primary objective of the job scheduler is to provide a balanced mix of jobs, such as I/O bound and processor bound. It also controls the degree of multiprogramming. 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.

On some systems, the long term scheduler may not be available or minimal. Time-sharing operating systems have no long term scheduler. When process changes the state from new to ready , then there is use of long term scheduler.

## LONG TERM SCHEDULER …

· Run seldom (rarely) ( when job comes into memory )

· Controls degree of multiprogramming

· Tries to balance arrival and departure rate through an appropriate job mix.

## SHORT TERM SCHEDULER

It is also called CPU scheduler. Main objective is increasing system performance in accordance with the chosen set of criteria. It is the change of ready state to running state of the process. CPU scheduler selects process among the processes that are ready to execute and allocates CPU to one of them.

Short term scheduler also known as dispatcher, execute most frequently and makes the fine grained decision of which process to execute next. Short term scheduler is faster than long term scheduler.

## SHORT TERM SCHEDULER…

Contains three functions:

· Code to remove a process from the processor at the end of its run.

a) Process may go to ready queue or to a wait state.

· Code to put a process on the ready queue –

a) Process must be ready to run.

b) Process placed on queue based on priority.

· Code to take a process off the ready queue and run that process (also called dispatcher).

a) Always takes the first process on the queue (no intelligence required)

b) Places the process on the processor.

This code runs frequently and so should be as short as possible.

## MEDIUM TERM SCHEDULER

Medium term scheduling is part of the swapping. It removes the processes from the memory. It reduces the degree of multiprogramming. The medium term scheduler is in-charge of handling the swapped out-processes.

## MEDIUM TERM SCHEDULER ….

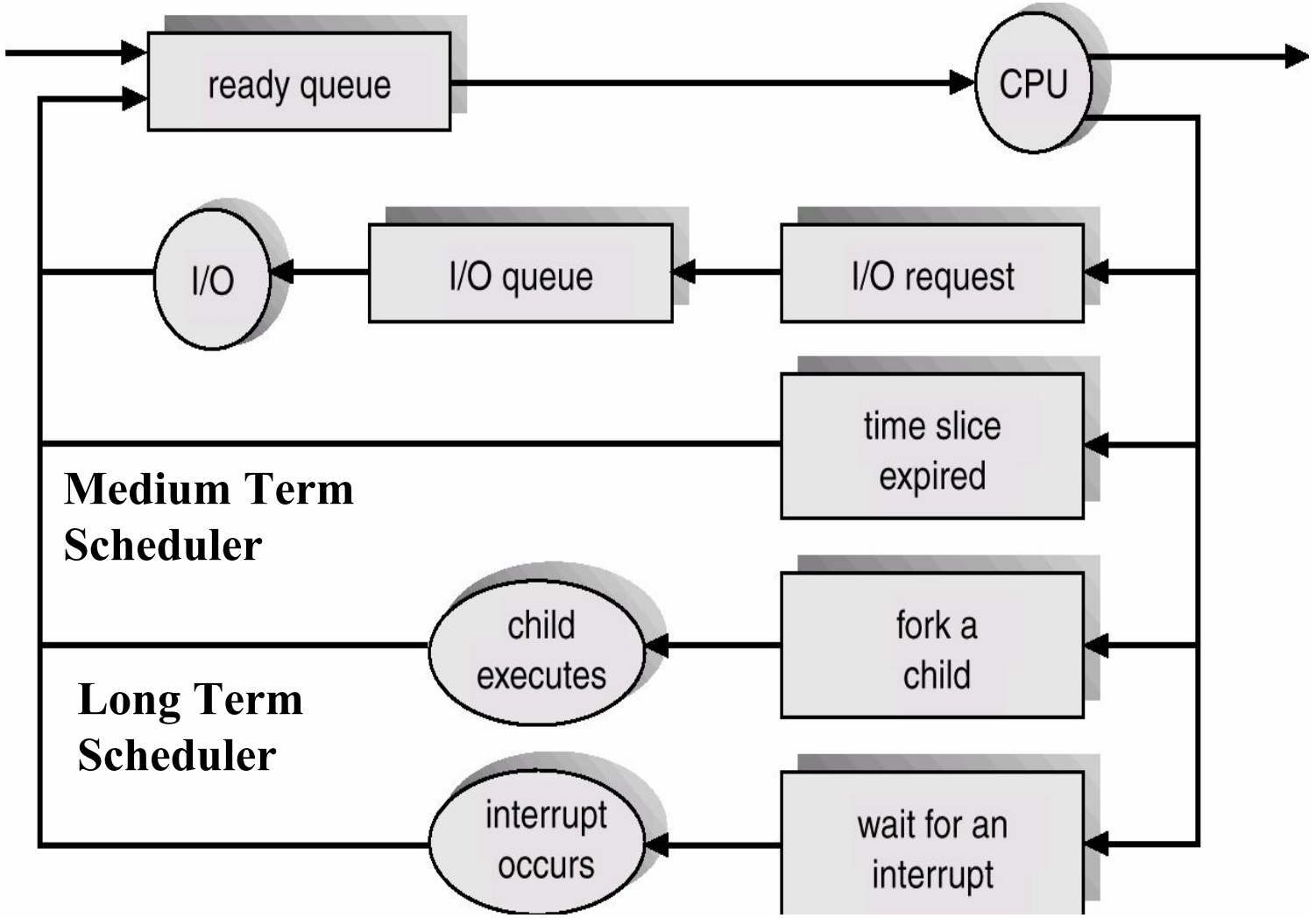
Mixture of CPU and memory resource management.

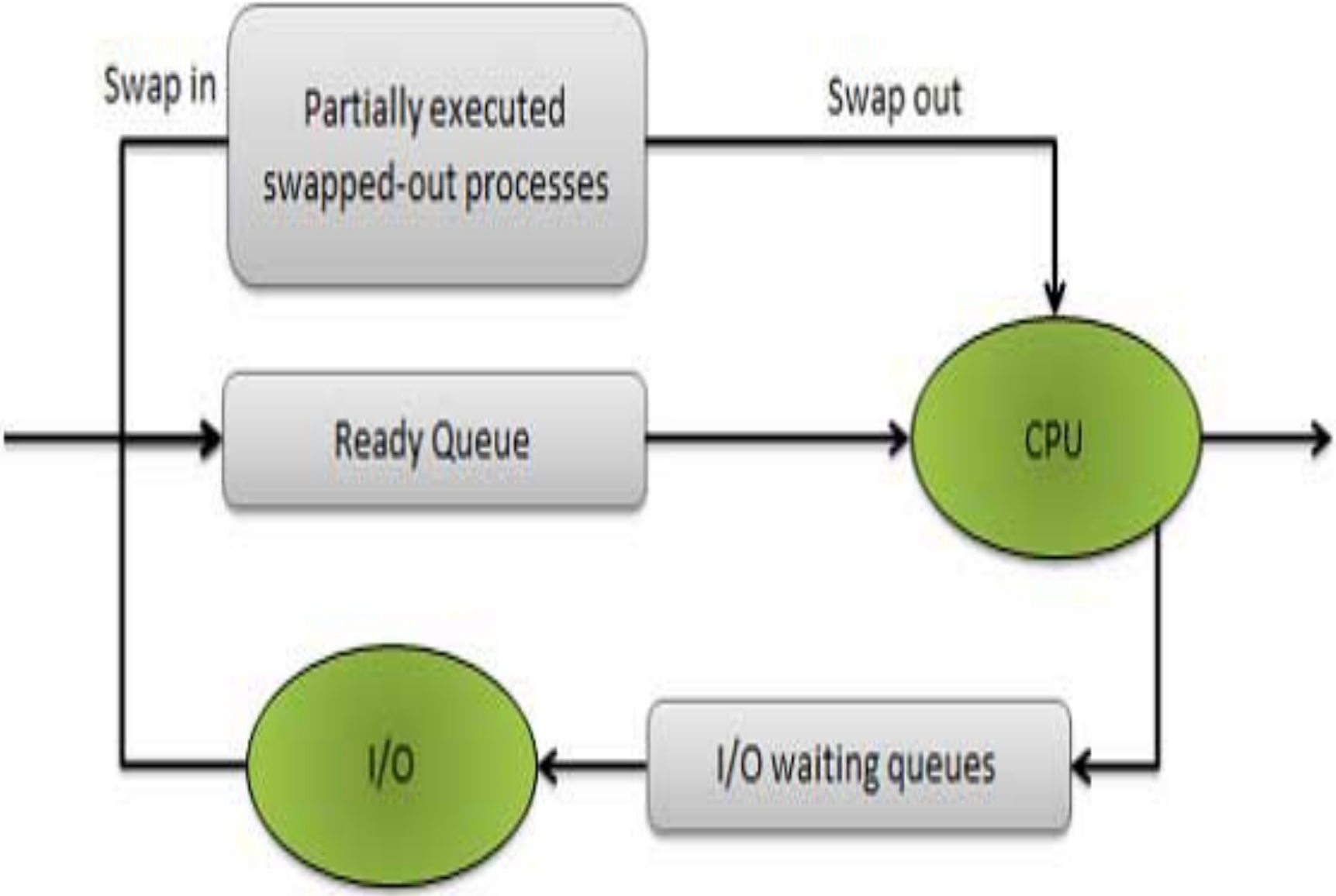
Swap out/in jobs to improve mix and to get memory.

Controls change of priority.

| Sr. No. | Long Term Scheduler | Short Term Scheduler | Medium Term Scheduler |
|---|---|---|---|
| 1 | It is a job scheduler | It is a CPU scheduler | It is a process swapping scheduler. |
| 2 | Speed is lesser than short term scheduler | Speed is fastest among other two | Speed is in between both short and long term scheduler. |
| 3 | It controls the degree of multiprogramming | It provides lesser control over degree of multiprogramming | It reduces the degree of multiprogramming. |
| 4 | It is almost absent or minimal in time sharing system | It is also minimal in time sharing system | It is a part of Time sharing systems. |
| 5 | It selects processes from pool and loads them into memory for execution | It selects those processes which are ready to execute | It can re-introduce the process into memory and execution can be continued |

**Short Term Scheduler**




Swap in
Partially executed swapped-out processes
Swap out
Ready Queue
CPU
I/O
I/O waiting queues

# Two State Process Model

Two state process model refers to running and non-running states which are described below.

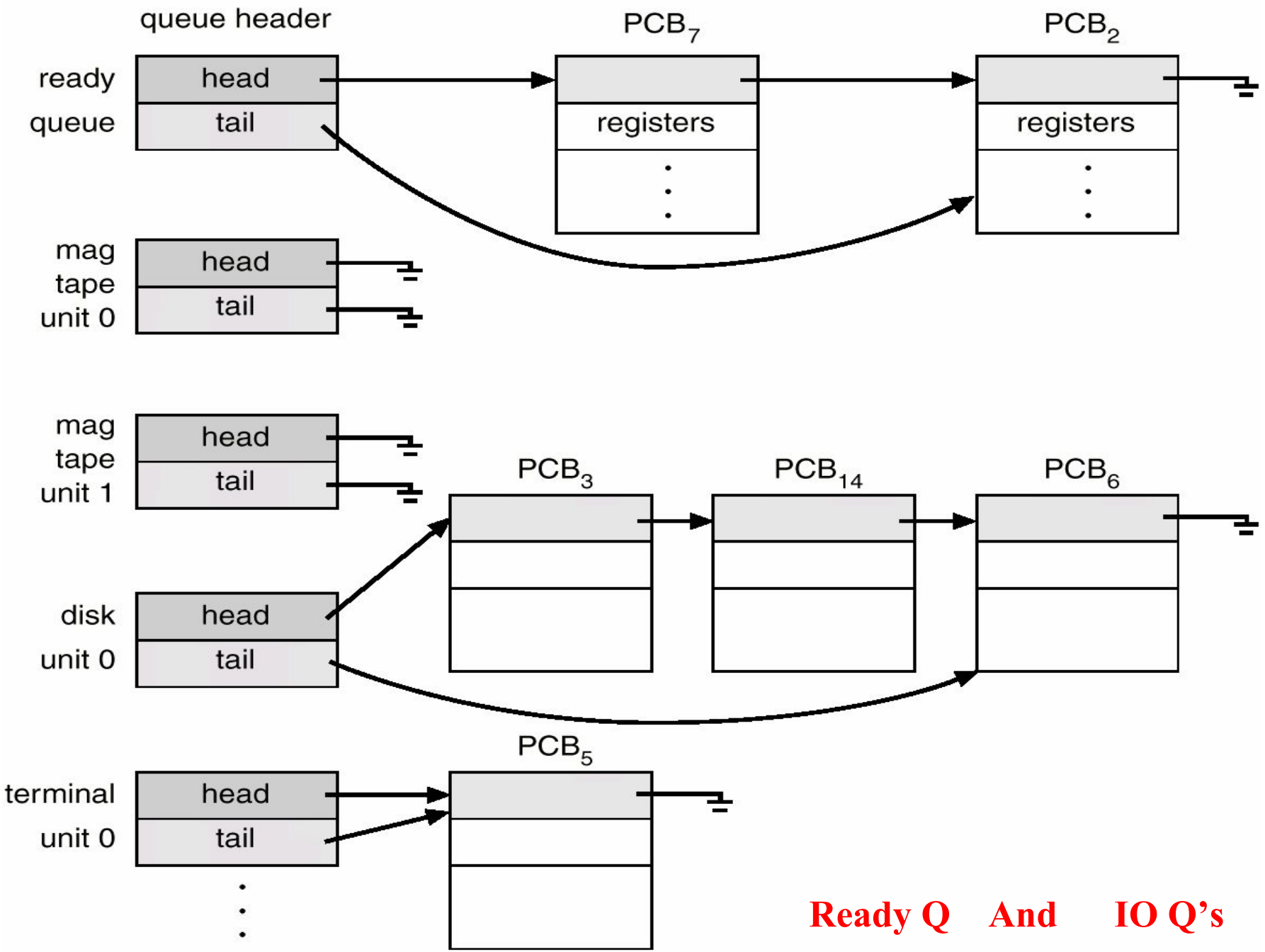
**Running**

1 When new process is created by Operating System that process enters into the system as in the running state.

**Not Running**

Processes that are not running are kept in queue, waiting for their turn to execute. Each entry in the queue is a pointer to a particular process. Queue

2 is implemented by using linked list. Use of dispatcher is as follows. When a process is interrupted, that process is transferred in the waiting queue. If the process has completed or aborted, the process is discarded. In either case, the dispatcher then selects a process from the queue to execute.




**Ready Q And IO Q's**

# Context Switch

A context switch is the mechanism to store and restore the state or context of a CPU in Process Control block so that a process execution can be resumed from the same point at a later time.

Using this technique a context switcher enables multiple processes to share a single CPU. Context switching is an essential part of a multitasking operating system features.

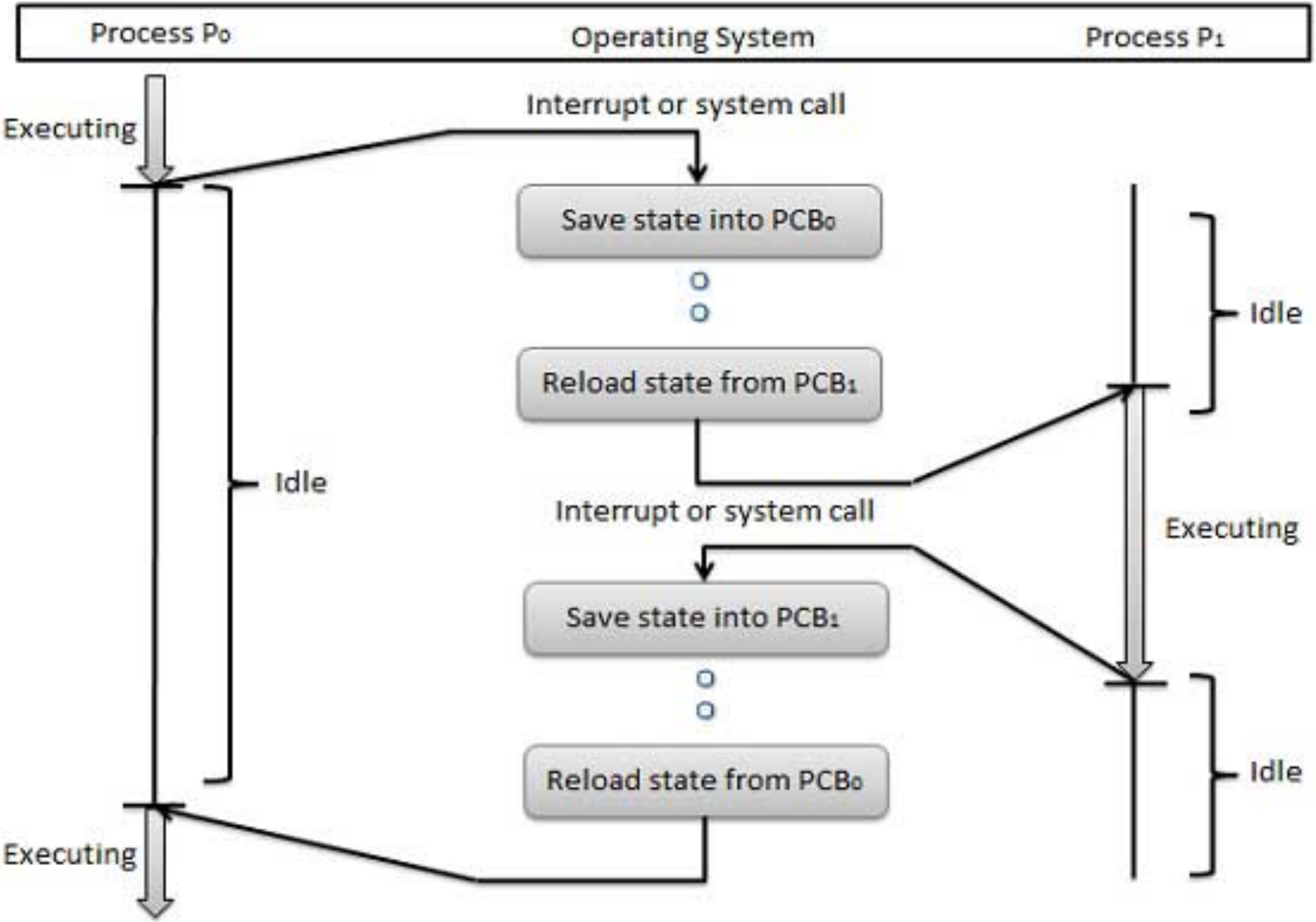
When the scheduler switches the CPU from executing one process to execute another, the context switcher saves the content of all processor registers for the process being removed from the CPU, in its process descriptor. The context of a process is represented in the process control block of a process.

Context switch time is pure overhead. Context switching can significantly affect performance as modern computers have a lot of general and status registers to be saved. Content switching times are highly dependent on hardware support.

Context switch requires ( n + m ) bxK time units to save the state of the processor with **n** general registers, assuming b are the store operations are required to save **n** and **m** registers of two process control blocks and each store instruction requires K time units.

Some hardware systems employ two or more sets of processor registers to reduce the amount of context switching time. When the process is switched, the following information is stored.

- Program
- Scheduling Information
- Base and limit register value
- Currently used register
- Changed State
- I/O State
- Accounting




**The CPU switching from one process to another.**

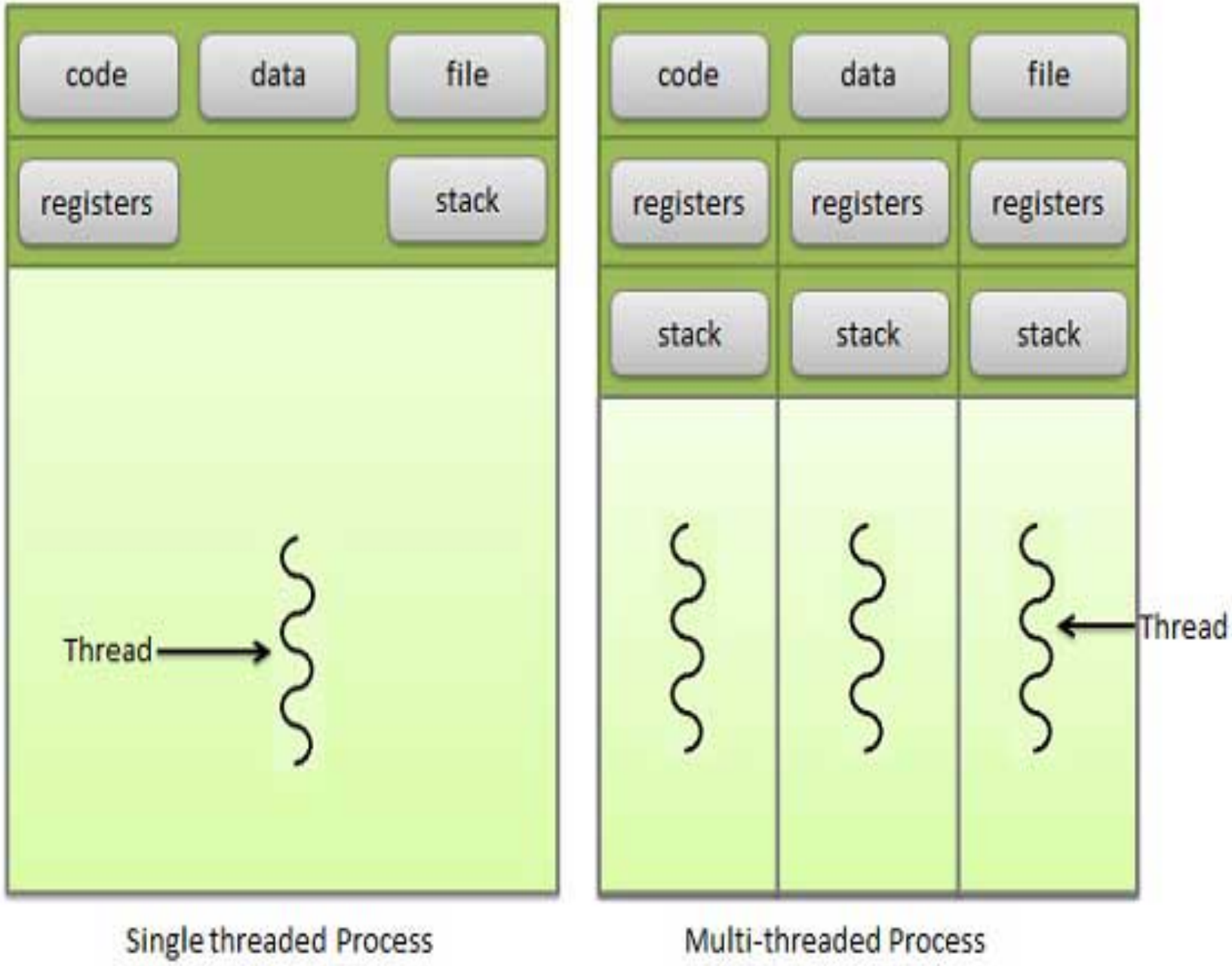
Running process may become suspended if it makes an I/O request. Suspended processes cannot make any progress towards completion. In this condition, to remove the process from memory and make space for other process, the suspended process is moved to the secondary storage. This process is called swapping, and the process is said to be swapped out or rolled out. Swapping may be necessary to improve the process mix.

## What is Thread?

A thread is a flow of execution through the process code, with its own program counter, system registers and stack. A thread is also called a light weight process. Threads provide a way to improve application performance through parallelism. Threads represent a software approach to improving performance of operating system by reducing the overhead thread is equivalent to a classical process.

Each thread belongs to exactly one process and no thread can exist outside a process. Each thread represents a separate flow of control.Threads have been successfully used in implementing network servers and web server. They also provide a suitable foundation for parallel execution of applications on shared memory multiprocessors.

Following figure shows the working of the single and multithreaded processes.


code
data
file
registers
stack
Thread
code
data
file
registers
registers
registers
stack
stack
stack
Thread
Single threaded Process
Multi-threaded Process

| Sr. No. | Process | Thread |
|---|---|---|
| 1 | Process is heavy weight or resource intensive. | Thread is light weight taking lesser resources than a process. |
| 1 | Process switching needs interaction with operating system. | Thread switching does not need to interact with operating system. |
| 1 | In multiple processing environments each process executes the same code but has its own memory and file resources. | All threads can share same set of open files, child processes. |
| 1 | If one process is blocked then no other process can execute until the first process is unblocked. | While one thread is blocked and waiting, second thread in the same task can run. |
| 1 | Multiple processes without using threads use more resources. | Multiple threaded processes use fewer resources. |
| 1 | In multiple processes each process operates independently of the others. | One thread can read, write or change another thread's data. |

## Adntages of Thread

- Thread minimize context switching time.
- Use of threads provides concurrency within a process.
- Efficient communication.
- Economy- It is more economical to create and context switch threads.
- Utilization of multiprocessor architectures to a greater scale and efficiency.

## Types of Thread

Threads are implemented in following two ways

- **User Level Threads** -- User managed threads
- **Kernel Level Threads** -- Operating System managed threads acting on kernel, an operating system core.

## User Level Threads

In this case, application manages thread management kernel is not aware of the existence of threads. The thread library contains code for creating and destroying threads, for passing message and data between threads, for scheduling thread execution and for saving and restoring thread contexts. The application begins with a single thread and begins running in that thread.

## Advantages

Thread switching does not require Kernel mode privileges.

User level thread can run on any operating system.

Scheduling can be application specific in the user level thread.

User level threads are fast to create and manage.

## Disadvantages

In a typical operating system, most system calls are blocking.

Multithreaded application cannot take advantage of multiprocessing.

## Kernel Level Threads

In this case, thread management done by the Kernel. There is no thread management code in the application area. Kernel threads are supported directly by the operating system. Any application can be programmed to be multithreaded. All of the threads within an application are supported within a single process.

The Kernel maintains context information for the process as a whole and for individuals threads within the process. Scheduling by the Kernel is done on a thread basis. The Kernel performs thread creation, scheduling and management in Kernel space. Kernel threads are generally slower to create and manage than the user threads.

## Advantages

Kernel can simultaneously schedule multiple threads from the same process on multiple processes.

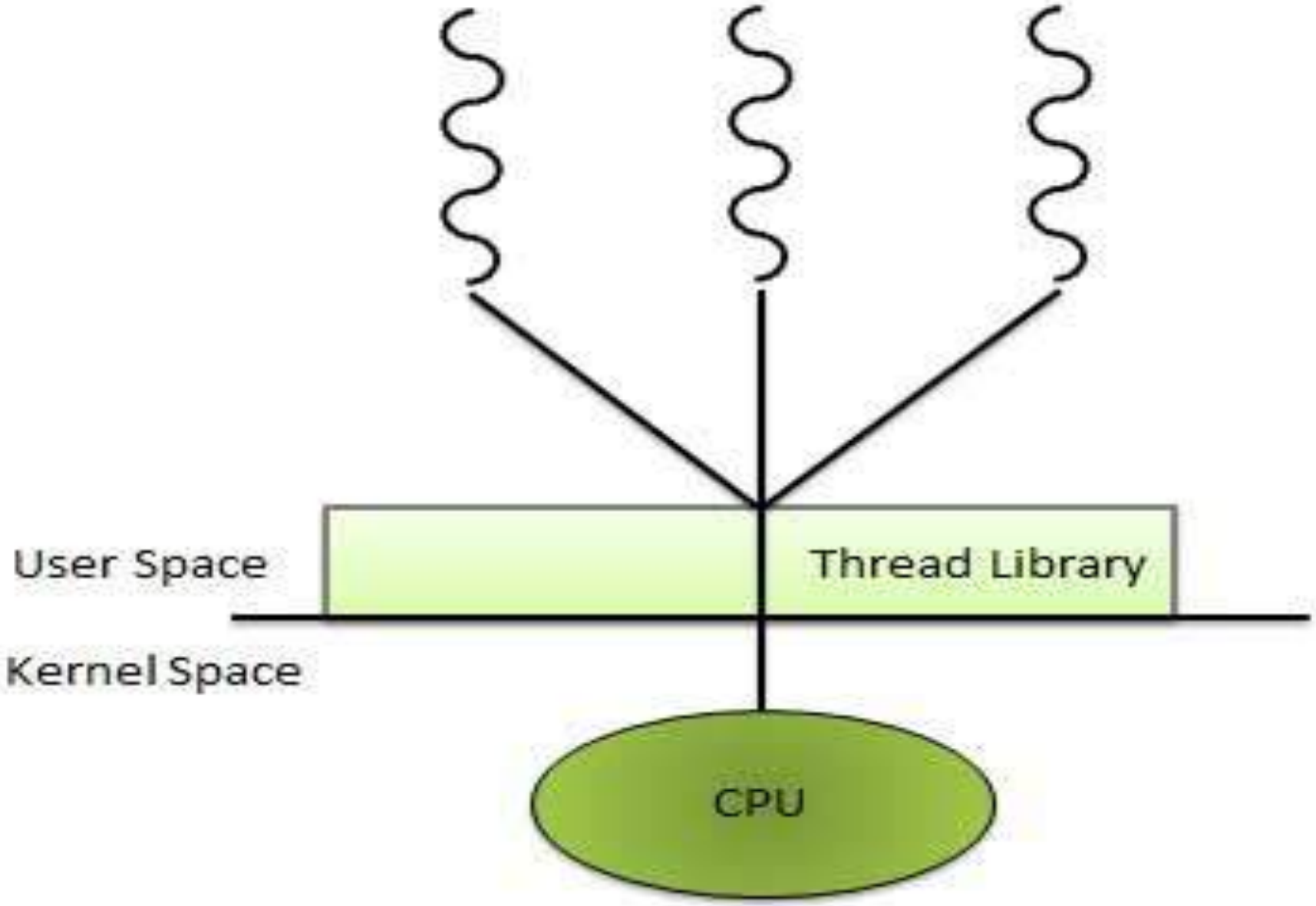
If one thread in a process is blocked, the Kernel can schedule another thread of the same process.

Kernel routines themselves can multithreaded.

## Disadvantages

Kernel threads are generally slower to create and manage than the user threads.

Transfer of control from one thread to another within same process requires a mode switch to the Kernel.


User Space
Thread Library
Kernel Space
CPU

# Multithreading Models

Some operating system provide a combined user level thread and Kernel level thread facility. Solaris is a good example of this combined approach. In a combined system, multiple threads within the same application can run in parallel on multiple processors and a blocking system call need not block the entire process. Multithreading models are three types

Many to many relationship.

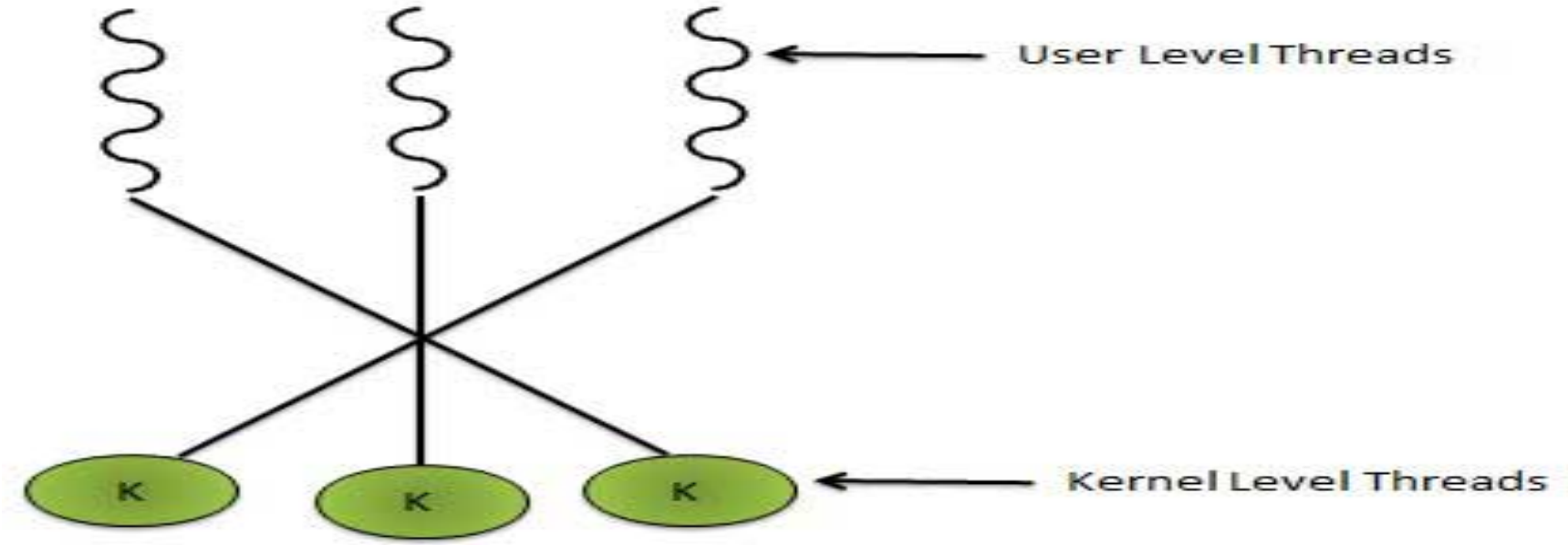
Many to one relationship.

One to one relationship.

# Many to Many Model

In this model, many user level threads multiplexes to the Kernel thread of smaller or equal numbers. The number of Kernel threads may be specific to either a particular application or a particular machine.

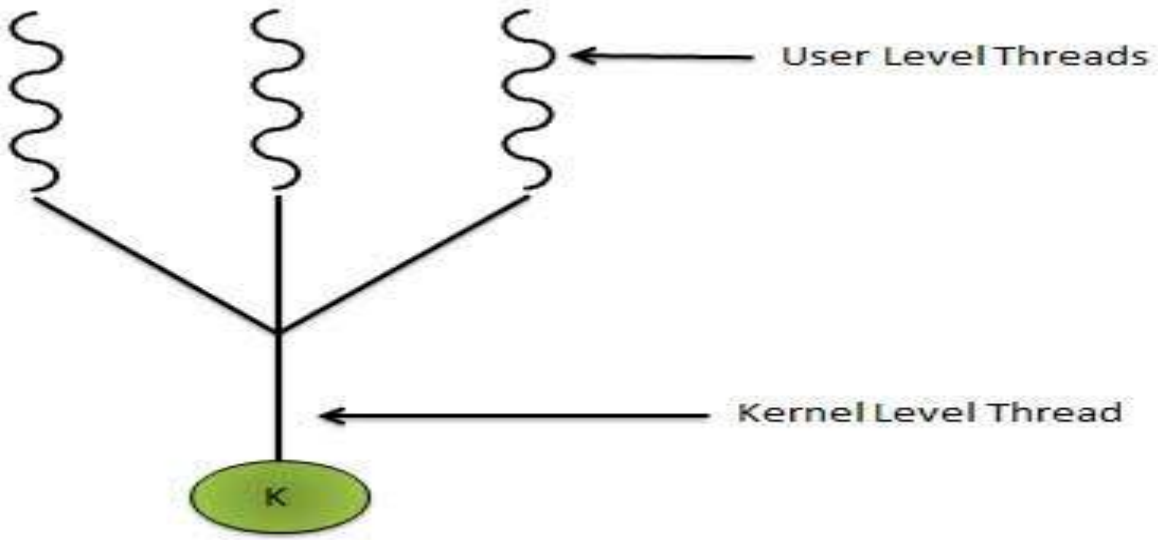
Following diagram shows the many to many model. In this model, developers can create as many user threads as necessary and the corresponding Kernel threads can run in parallels on a multiprocessor.



# Many to One Model

Many to one model maps many user level threads to one Kernel level thread. Thread management is done in user space. When thread makes a blocking system call, the entire process will be blocks. Only one thread can access the Kernel at a time, so multiple threads are unable to run in parallel on multiprocessors.

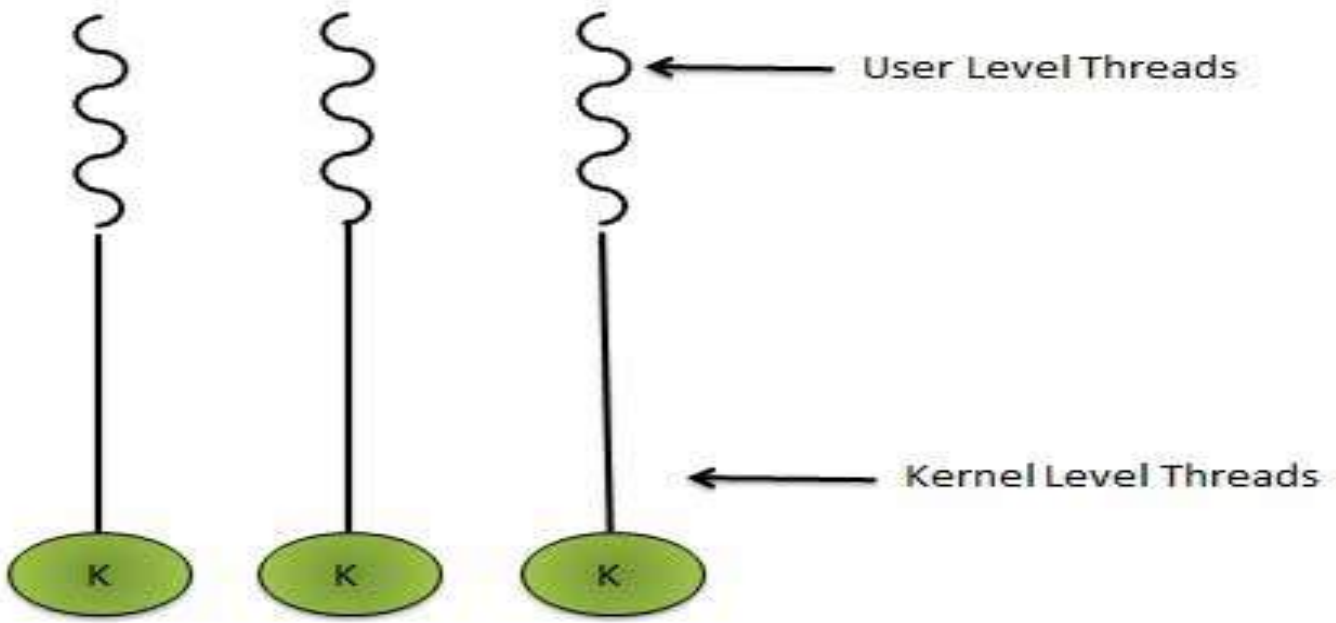
If the user level thread libraries are implemented in the operating system in such a way that system does not support them then Kernel threads use the many to one relationship modes



# One to One Model

There is one to one relationship of user level thread to the kernel level thread. This model provides more concurrency than the many to one model. It also another thread to run when a thread makes a blocking system call. It support multiple thread to execute in parallel on microprocessors.

Disadvantage of this model is that creating user thread requires the corresponding Kernel thread. OS/2, windows NT and windows 2000 use one to one relationship model



## Difference between User Level & Kernel Level Thread

| Sr. No. | User Level Threads | Kernel Level Thread |
|---|---|---|
| 1 | User level threads are faster to create and manage. | Kernel level threads are slower to create and manage. |
| 2 | Implementation is by a thread library at the user level. | Operating system supports creation of Kernel threads. |
| 3 | User level thread is generic & can run on any OS | Kernel level thread is specific to the operating system. |
| 4 | Multi-threaded application can't take advantage of multiprocessing. | Kernel routines themselves can be multithreaded. |

## Scheduling Criteria

**CPU utilization** – keep the CPU as busy as possible

**Throughput** – No of processes that complete their execution per time unit .

**Turnaround Time** – amount of time to execute a particular process

**Waiting Time** – amount of time a process has been waiting in the ready queue

**Response Time** – amount of time it takes from when a request was submitted until the first response is produced, not output (for time-sharing environment)

## Optimization Criteria

Max CPU utilization

Max throughput

Min turnaround time

Min waiting time

Min response time

## Scheduling Criteria

Different CPU scheduling algorithms have different properties, and the choice of a particular algorithm may favor one class of processes over another. In choosing which algorithm to use in a particular situation, we must consider the properties of the various algorithms. Many criteria have been suggested for comparing CPU scheduling algorithms. Which characteristics are used for comparison can make a substantial difference in which algorithm is judged to be best. The criteria include the following:

CPU Utilization. We want to keep the CPU as busy as possible.

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 10 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 the 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 periods spend waiting in the ready queue.

Response time. In an interactive system, turnaround time may not be the best criterion. Often, a process can produce some output fairly early and can continue computing new results while previous results are being output to the user. Thus, another measure is the time from the submission of a request until the first response is produced.

This measure, called response time, is the time it takes to start responding, not the time it takes to output the response. The turnaround time is generally limited by the speed of the output device.

It is desirable to maximize CPU utilization and throughput and to minimize turnaround time, waiting time, and response time. In most cases, we optimize the average measure. However, under some circumstances, it is desirable to optimize the minimum or maximum values rather than the average. For example, to guarantee that all users get good service, we may want to minimize the maximum response time. Investigators have suggested that, for interactive systems, it is more important to minimize the variance in the response time than to minimize the average response time. A system with reasonable and predictable response time may be considered more desirable than a system that is faster on the average but is highly variable. However, little work has been done on CPU-scheduling algorithms that minimize variance

## Scheduling Algorithms

First-Come, First-Served (FCFS) Scheduling

Shortest-Job-First (SJR) Scheduling

Priority Scheduling

Round Robin (RR)

Multilevel queue scheduling

## How to choose a scheduling algorithm

When designing an operating system, a programmer must consider which scheduling algorithm will perform best for the use the system is going to see. There is no universal “best” scheduling algorithm, and many operating systems use extended or combinations of the scheduling algorithms above. For example, Windows NT/XP/Vista uses a Multilevel feedback queue, a combination of fixed priority preemptive scheduling, round-robin, and first in first out.

In this system, processes can dynamically increase or decrease in priority depending on if it has been serviced already, or if it has been waiting extensively. Every priority level is represented by its own queue, with round-robin scheduling amongst the high priority processes and FIFO among the lower ones. In this sense, response time is short for most processes, and short but critical system processes get completed very quickly. Since processes can only use one time unit of the round robin in the highest priority queue, starvation can be a problem for longer high priority processes.

## Types of Scheduling Algorithm

**Circumstances that scheduling may take place :-**

A process switches from the running state to the waiting state (e.g., doing for I/O, invocation of wait for the termination of one of the child processes)

A process switches from the running state to the ready state

(e.g., an interrupt occurs)

A process switches from the waiting state to the ready state (e.g., I/O completion)

A process terminates

**Non-preemptive scheduling** :-

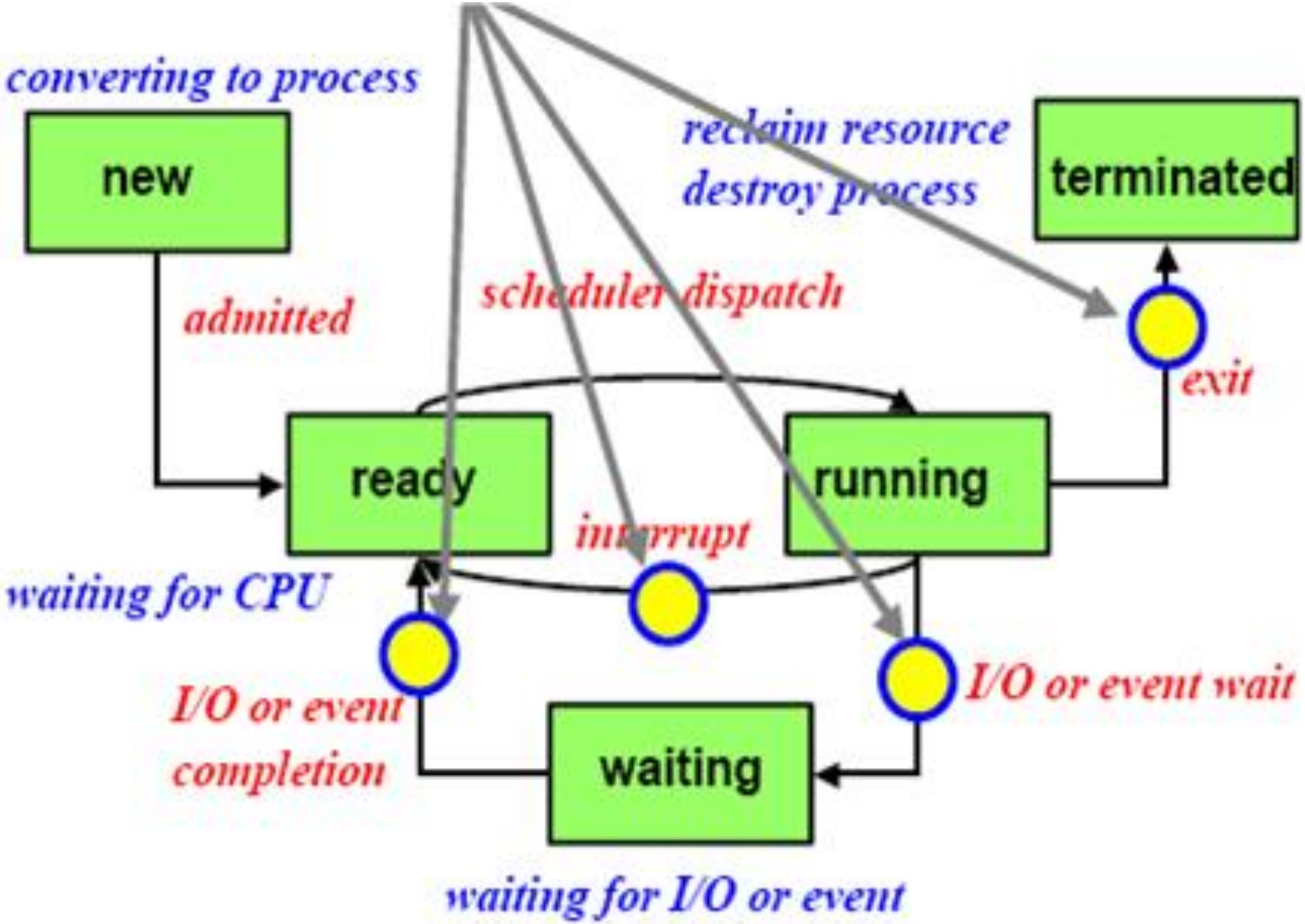
Scheduling occurs only when a process voluntarily enter the wait state or terminates

Simple, but very inefficient.

It is the only method that can be used on certain hardware platforms, because it does not require the special hardware (for example, a timer) needed for preemptive scheduling.

**Preemptive Scheduling :-** scheduling occurs in all possible cases. What if the kernel is in its critical section modifying some important data? Mutual exclusion may be violated.

The kernel must pay special attention to this situation and, hence, is more complex.


converting to process
new
admitted
reclaim resource
destroy process
terminated
scheduler dispatch
exit
ready
running
interrupt
waiting for CPU
I/O or event wait
I/O or event
completion
waiting
waiting for I/O or event

## First Come First Serve

First Come, First Served (FCFS), is the simplest scheduling algorithm, FIFO simply queues processes in the order that they arrive in the ready queue.

Since context switches only occur upon process termination, and no reorganization of the process queue is required, scheduling overhead is minimal.

Throughput can be low, since long processes can hog the CPU

Turnaround time, waiting time and response time can be high for the same reasons above

No prioritization occurs, thus this system has trouble meeting process deadlines.

The lack of prioritization means that as long as every process eventually completes, there is no starvation. In an environment where some processes might not complete, there can be starvation.

It is based on Queuing

| Process | Arrival Time | Execute Time | Service Time |
|---|---|---|---|
| P0 | 0 | 5 | 0 |
| P1 | 1 | 3 | 5 |
| P2 | 2 | 8 | 8 |
| P3 | 3 | 6 | 16 |


Wait time of each process is following

| Process | Wait Time : Service Time - Arrival Time |
|---|---|
| P0 | 0 - 0 = 0 |
| P1 | 5 - 1 = 4 |
| P2 | 8 - 2 = 6 |
| P3 | 16 - 3 = 13 |

Average Wait Time: (0+4+6+13) / 4 = 5.55

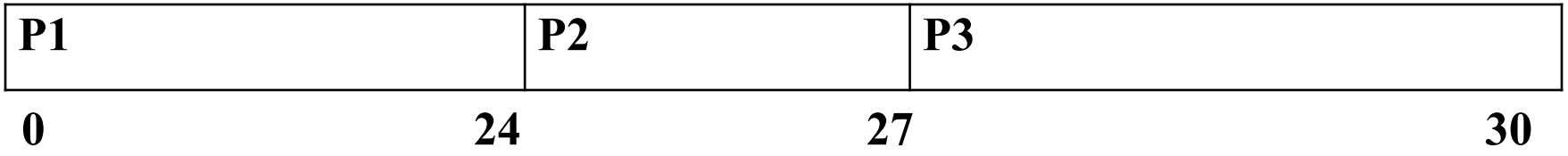
## First-Come, First-Served (FCFS) Scheduling

| Process | Burst Time (time required for completion) |
|---|---|
| P1 | 24 |
| P2 | 3 |
| P3 | 3 |

**Suppose that the processes arrive in the order : *P1* , *P2* , *P3***

**The Gantt Chart for the schedule is:**

a chart in which a series of horizontal lines shows the amount of work done or production completed in certain periods of time in relation to the amount planned for those periods.

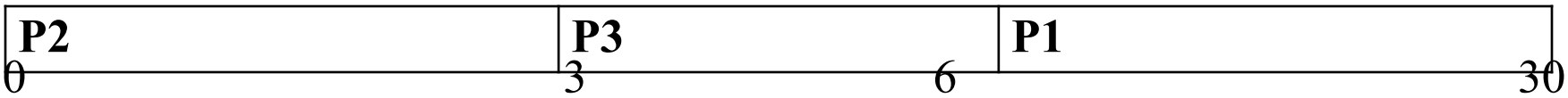



**Waiting time for *P1* = 0; *P2* = 24; *P3* = 27**

**Average waiting time: (0 + 24 + 27)/3 = 51/3 = 17**

Suppose that the processes arrive in the order :- *P2* , *P3* , *P1* .

The Gantt chart for the schedule is:




Waiting time for *P1* = 6*;* *P2* = 0*;* *P3* = 3

Average waiting time: (6 + 0 + 3)/3 = 9/3 = 3

Much better than previous case.

*Due to Convoy effect* short process behind long process

| **Process** | **Burst Time (time required for completion)** |
|---|---|
| P2 | 03 |
| P3 | 03 |
| P1 | 24 |

# Convoy effect

Consider : $P1$ : **CPU-bound** $P2$, $P3$, $P4$ : **I/O-bound**

$P2$, $P3$ and $P4$ could quickly finish their IO request → ready queue, waiting for CPU.

**Note : IO devices are idle then.**

then $P1$ finishes its CPU burst and move to an IO device.

$P2$, $P3$, $P4$, which have short CPU bursts, finish quickly → back to IO queue.

**Note: CPU is idle then.**

$P1$ moves then back to ready queue is gets allocated CPU time.

Again $P2$, $P3$, $P4$ wait behind $P1$ when they request CPU time.

One cause : FCFS is non-preemptive

$P1$ keeps the CPU as long as it needs

## Shortest-Job-First (SJR) Scheduling

Associate with each process the length of its next CPU burst. Use these lengths to schedule the process with the shortest time.

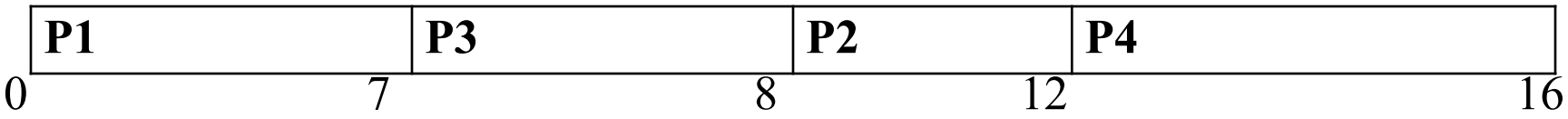
Two schemes:

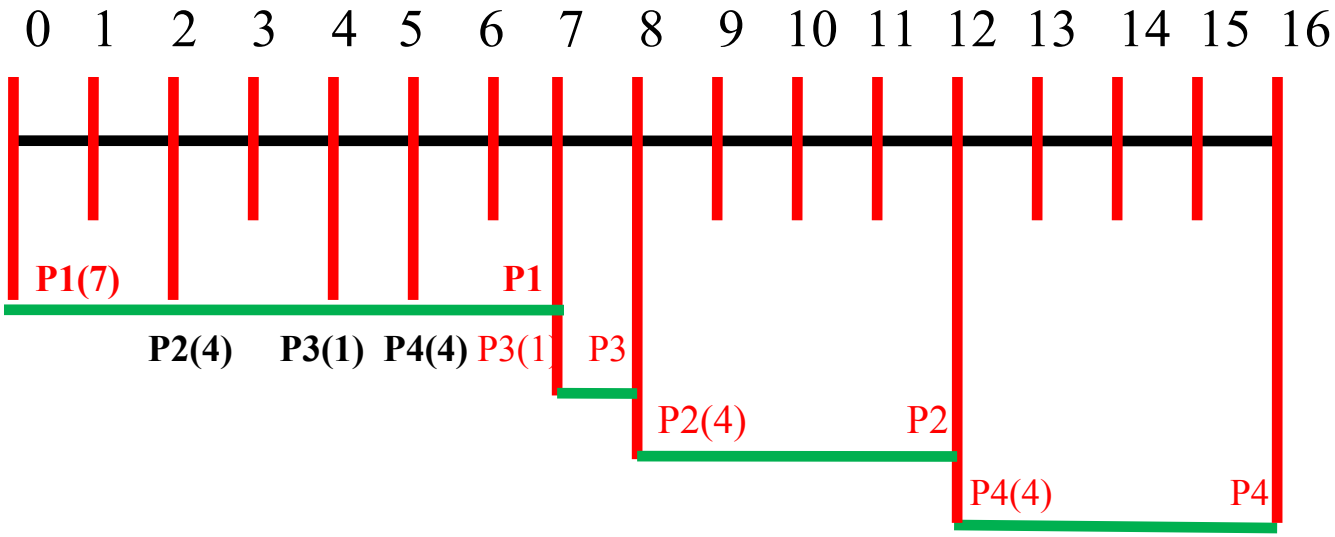
1. non pre- emptive – once CPU given to the process it cannot be preempted until completes its CPU burst.
2. preemptive – if a new process arrives with CPU burst length less than remaining time of current executing process, preempt. This scheme is know as the Shortest-Remaining-Time-First (SRTF).

SJF is optimal – gives minimum average waiting time for a given set of processes.

| Process | Arrival Time | Burst Time |
|---|---|---|
| *P1* | 0.0 | 7 |
| *P2* | 2.0 | 4 |
| *P3* | 4.0 | 1 |
| *P4* | 5.0 | 4 |

## SJF (non-preemptive)

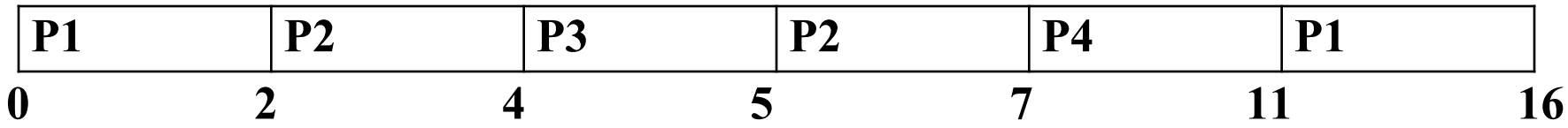



| Process | Arrival Time | Burst Time | Waiting Time |
|---|---|---|---|
| P1 | 0.0 | 7 | 0 |
| P2 | 2.0 | 4 | 08 - 02 = 6 |
| P3 | 4.0 | 1 | 07 - 04 = 3 |
| P4 | 5.0 | 4 | 12 - 05 = 7 |

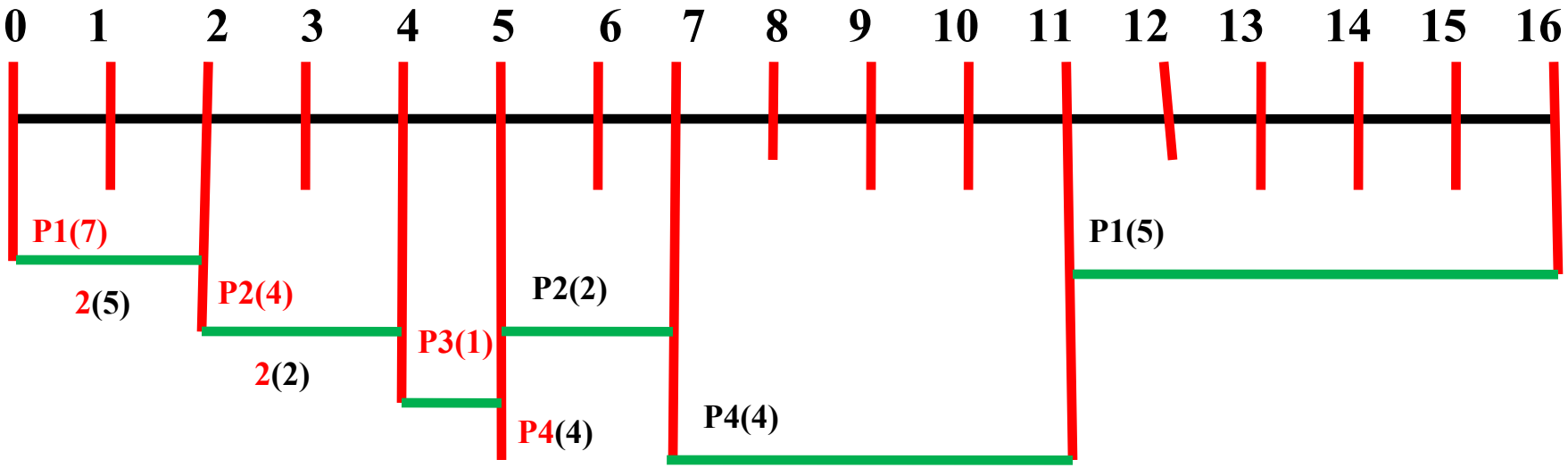
Average waiting time = [0 +(8-2) + (7-4) + (12-5)] /4 = 16/4 = 4

# Example of Preemptive SJF




## Shortest Job First (SJF)

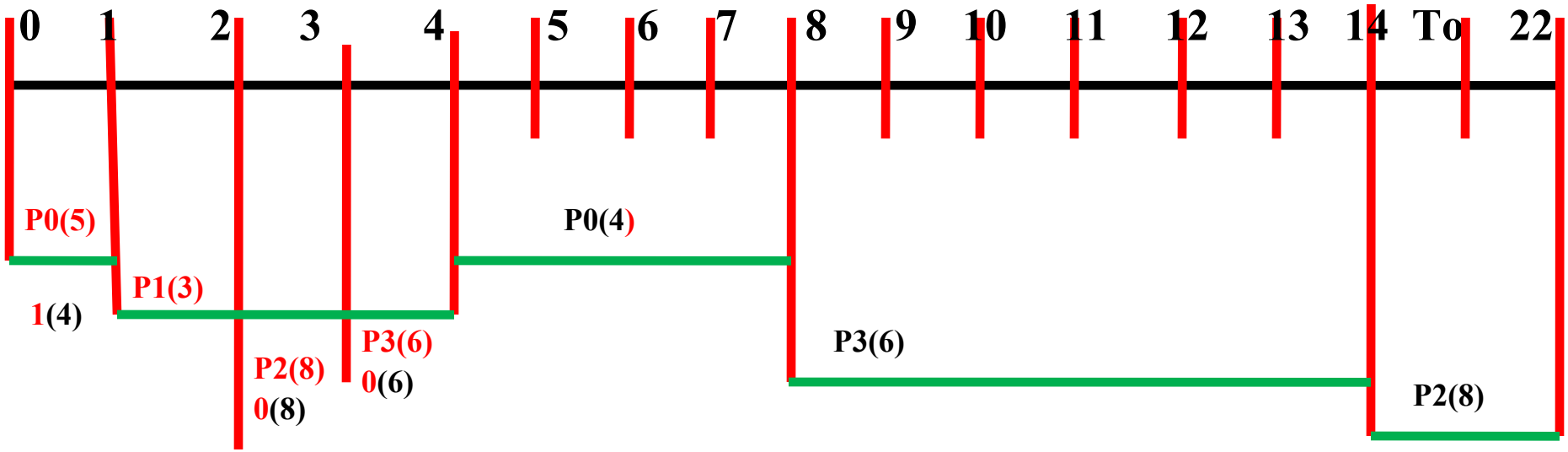
- Best approach to minimize waiting time.
- Impossible to implement
- Processer should know in advance how much time process will take.



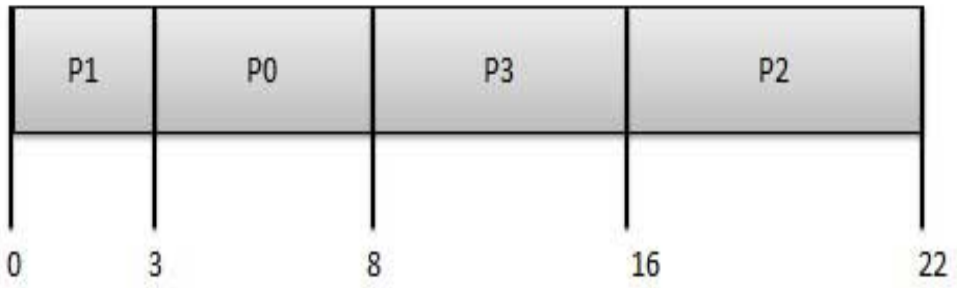

| Process | Arrival Time | Burst Time | Waiting Time |
|---|---|---|---|
| P1 | 0.0 | 7 | 11 - 02 = 09 |
| P2 | 2.0 | 4 | 05 – 02 = 03 – 02= 01 |
| P3 | 4.0 | 1 | 04 – 04 = 00 |
| P4 | 5.0 | 4 | 07 – 05 = 02 |

**SJF (preemptive)**

**Average waiting time = (9 + 1 + 0 +2) /4 = 12/4 = 3**




| Process | Arrival Time | Execute Time | Service Time |
|---|---|---|---|
| P0 | 0 | 5 | 0 |
| P1 | 1 | 3 | 3 |
| P2 | 2 | 8 | 8 |
| P3 | 3 | 6 | 16 |




| Process | Wait Time : Service Time - Arrival Time |
|---|---|
| P0 | 3 - 0 = 3 (4 – 1 = 3) |
| P1 | 0 - 0 = 0 (1-1 = 0) |
| P2 | 14 -02 = 12 |
| P3 | 8 - 3 = 5 |

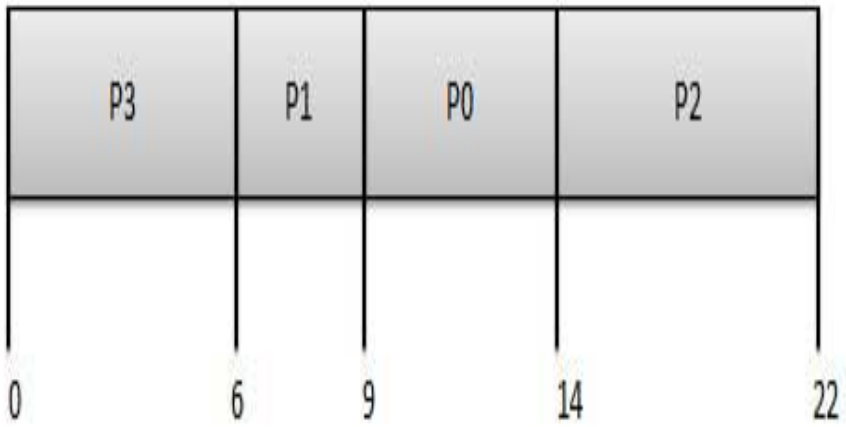
**Average waiting time = (3 + 0 + 12 +5) /4 = 20/4 = 5**

## Priority Based Scheduling

- Each process is assigned a priority. Process with highest priority is to be executed first and so on.
- Processes with same priority are executed on first come first serve basis.
- Priority can be decided based on memory requirements, time requirements or any other resource requirement.

Wait time of each process is following

| Process | Arrival Time | Execute Time | Priority | Service Time |
|---|---|---|---|---|
| P0 | 0 | 5 | 1 | 0 |
| P1 | 1 | 3 | 2 | 3 |
| P2 | 2 | 8 | 1 | 8 |
| P3 | 3 | 6 | 3 | 16 |




| Process | Wait Time : Service Time - Arrival Time |
|---|---|
| P0 | 9 - 0 = 9 |
| P1 | 6 - 1 = 5 |
| P2 | 14 - 2 = 12 |
| P3 | 0 - 0 = 0 |

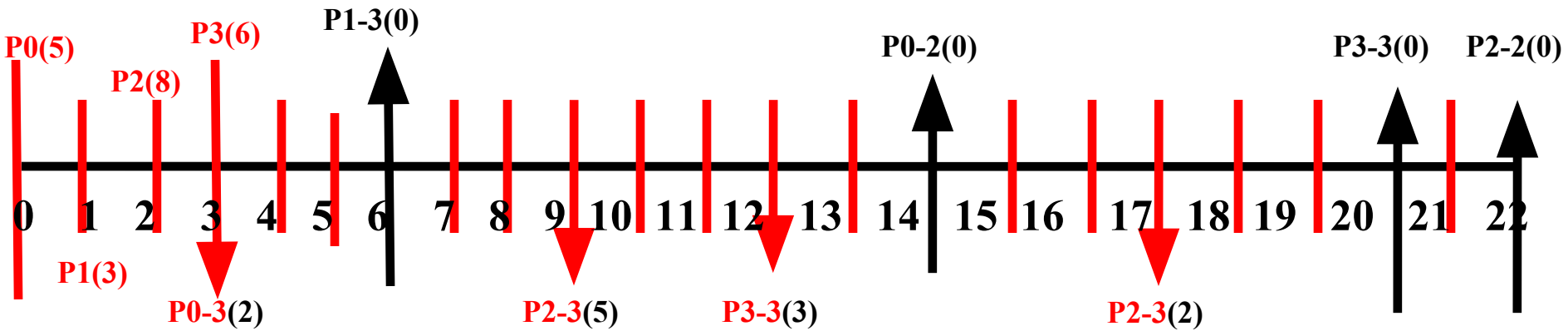
Average Wait Time : (9+5+12+0) / 4 = 6.5

## Round Robin Scheduling

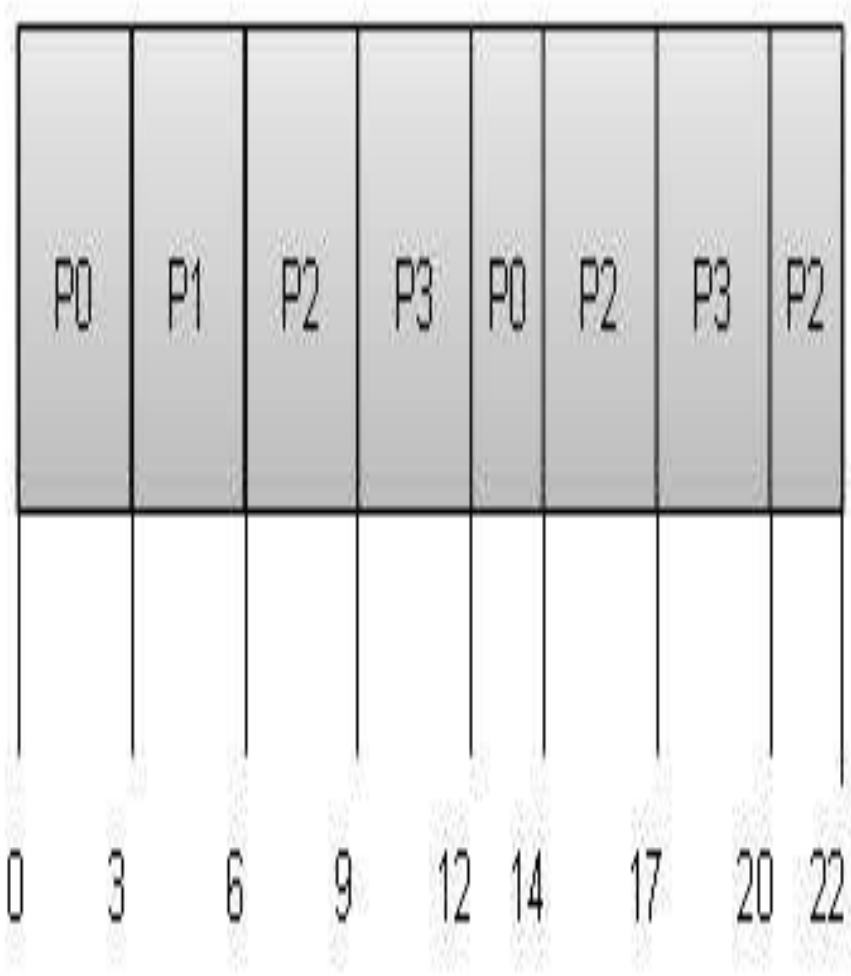
- Each process is provided a fix time to execute called quantum.
- Once a process is executed for given time period. Process is preempted and other process executes for given time period.
- Context switching is used to save states of preempted processes.

Wait time of each process is following

Average Wait Time: (9+2+12+11) / 4 = 8.5



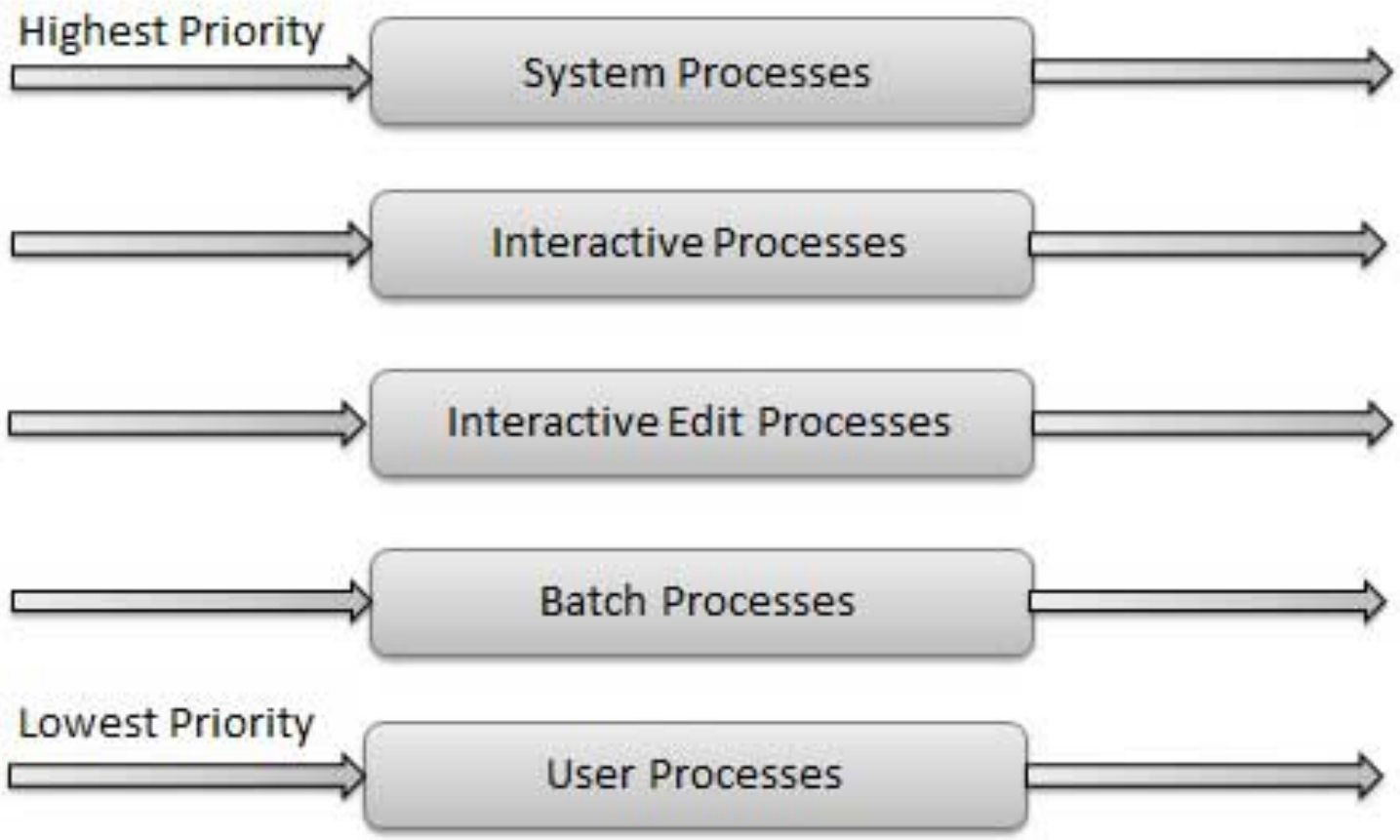
Quantum = 3




| Process | Wait Time : Service Time - Arrival Time |
|---|---|
| P0 | (0-0 ) + (12-3) = 9 |
| P1 | (3-1) = 2 |
| P2 | (6-2) + (14-9) + (20-17) = 12 |
| P3 | (9-3) + (17-12) = 11 |

## Multi Queue Scheduling

- Multiple queues are maintained for processes.
- Each queue can have its own scheduling algorithms.
- Priorities are assigned to each queue.



## Multitasking

Multitasking has the same meaning of multiprogramming but in a more general sense, as it refers to having multiple (programs, processes, tasks, threads) running at the same time. This term is used in modern operating systems when multiple tasks share a common processing resource (e.g., CPU and Memory). At any time the CPU is executing one task only while other tasks waiting their turn. The illusion of parallelism is achieved when the CPU is reassigned to another task (i.e. *process* or *thread context switching*).

There are subtle differences between multitasking and multiprogramming. A *task* in a multitasking operating system is not a whole application program but it can also refer to a "thread of execution" when one process is divided into sub-tasks. Each smaller task does not hijack the CPU until it finishes like in the older multiprogramming but rather a fair share amount of the CPU time called quantum.

Just to make it easy to remember, both multiprogramming and multitasking operating systems are **(CPU) time sharing** systems. However, while in multiprogramming (older OSs) one program as a whole keeps running until it blocks, in multitasking (modern OSs) time sharing is best manifested because each running process takes only a fair quantum of the CPU time.

## Multithreading

Up to now, we have talked about multiprogramming as a way to allow multiple programs being resident in main memory and (apparently) running at the same time. Then, multitasking refers to multiple tasks running (apparently) simultaneously by sharing the CPU time. Finally, multiprocessing describes systems having multiple CPUs. So, where does multithreading come in? Multithreading is an execution model that allows a single process to have multiple code segments (i.e., *threads*) run concurrently within the "context" of that process. You can think of threads as child processes that share the parent process resources but execute independently. Multiple threads of a single process can share the CPU in a single CPU system or (purely) run in parallel in a multiprocessing system Why should we need to have multiple threads of execution within a single process context?

Well, consider for instance a GUI application where the user can issue a command that require long time to finish (e.g., a complex mathematical computation). Unless you design this command to be run in a separate execution thread you will not be able to interact with the main application GUI (e.g., to update a progress bar) because it is going to be unresponsive while the calculation is taking place. Of course, designing multithreaded/concurrent applications requires the programmer to handle situations that simply don't occur when developing single-threaded, sequential applications. For instance, when two or more threads try to access and modify a shared resource (*race conditions*), the programmer must be sure this will not leave the system in an inconsistent or deadlock state. Typically, this thread synchronization is solved using OS primitives, such as **mutexes** and **sempaphores**.

## Multiprogramming

In a multiprogramming system there are one or more programs loaded in main memory which are ready to execute. Only one program at a time is able to get the CPU for executing its instructions (i.e., there is at most one process running on the system) while all the others are waiting their turn.

The main idea of multiprogramming is to maximize the use of CPU time. Indeed, suppose the currently running process is performing an I/O task (which, by definition, does not need the CPU to be accomplished). Then, the OS may interrupt that process and give the control to one of the other in-main-memory programs that are ready to execute (i.e. *process context switching*). In this way, no CPU time is wasted by the system waiting for the I/O task to be completed, and a running process keeps executing until either it voluntarily releases the CPU or when it blocks for an I/O operation. Therefore, the ultimate goal of multiprogramming is to keep the CPU busy as long as there are processes ready to execute.

## Multiprocessing

Multiprocessing sometimes refers to executing multiple processes (programs) at the same time. This might be misleading because we have already introduced the term "multiprogramming" to describe that before.

In fact, multiprocessing refers to the *hardware* (i.e., the CPU units) rather than the *software* (i.e., running processes). If the underlying hardware provides more than one processor then that is multiprocessing. Several variations on the basic scheme exist, e.g., multiple cores on one die or multiple dies in one package or multiple packages in one system.

Anyway, a system can be both multiprogrammed by having multiple programs running at the same time and multiprocessing by having more than one physical processor.

**Unit III** **(8 Hrs)**

**Concurrency Control : Concurrency and Race Conditions, Mutual exclusion requirements**

Software and hardware solutions, Semaphores, Monitors, Classical IPC problems and solutions.

Deadlock : Characterization, Detection, Recovery, Avoidance and Prevention.

## What is Concurrency?

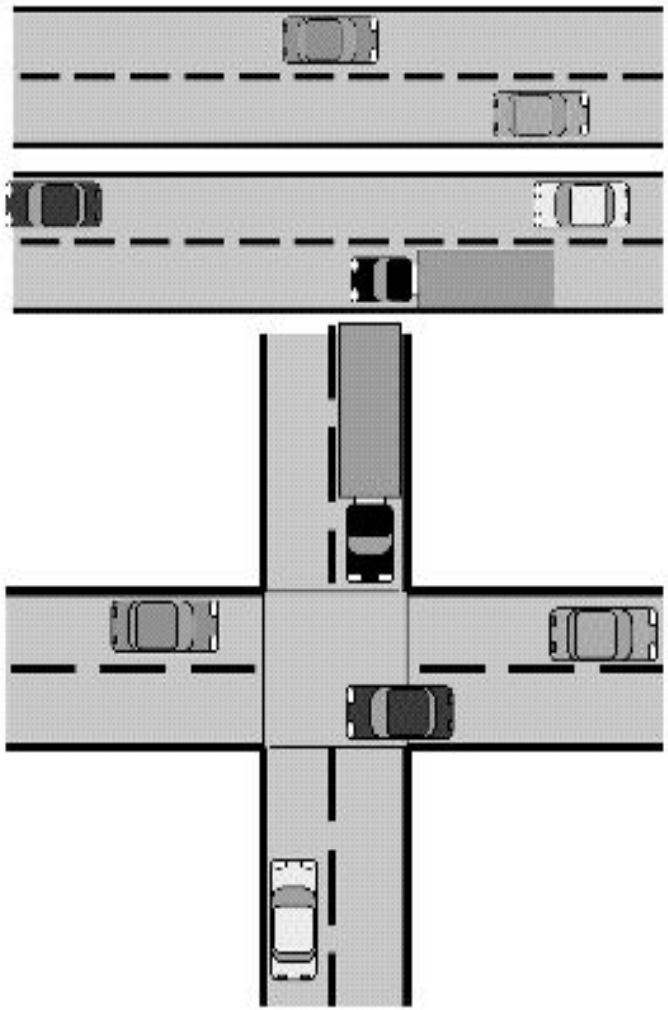
Concurrency is the tendency for things to happen at the same time in a system. Concurrency is a natural phenomenon.

In the real world, at any given time, many things are happening simultaneously. When we design software to monitor and control real-world systems, we must deal with this natural concurrency.

When dealing with concurrency issues in software systems, there are generally two aspects that are important: **being able to detect and respond to external events occurring in a random order**, and **ensuring that these events are responded to in some minimum required interval.**

If each concurrent activity evolved independently, in a truly parallel fashion, this would be relatively simple: we could simply create separate programs to deal with each activity.

The challenges of designing concurrent systems arise mostly because of the interactions which happen between concurrent activities. When concurrent activities interact, some sort of coordination is required.



e.g. - parallel activities that do not interact have simple concurrency issues. It is when parallel activities interact or share the same resources that concurrency issues become important.

Vehicular traffic provides a useful analogy. Parallel traffic streams on different roadways having little interaction cause few problems. Parallel streams in adjacent lanes require some coordination for safe interaction, but a much more severe type of interaction occurs at an intersection, where careful coordination is required.

Process management in operating systems can be classified broadly into three categories:

**Multiprogramming** involves *multiple processes* on *a system* with a *single processor*.

**Multiprocessing** involves *multiple processes* on *a system* with *multiple processors*.

**Distributed processing** involves multiple processes on *multiple systems*.

All of these involve cooperation, competition, and communication between processes that either run simultaneously or are interleaved in arbitrary ways to give the appearance of running simultaneously.

Concurrent processing is thus central to operating systems and their design.

## Mutual Exclusion

Mutual exclusion is in many ways the fundamental issue in concurrency. It is the requirement that when a process P is accessing a shared resource R, no other process should be able to access R until P has finished with R. Examples of such resources include files, I/O devices such as printers, and shared data structures.

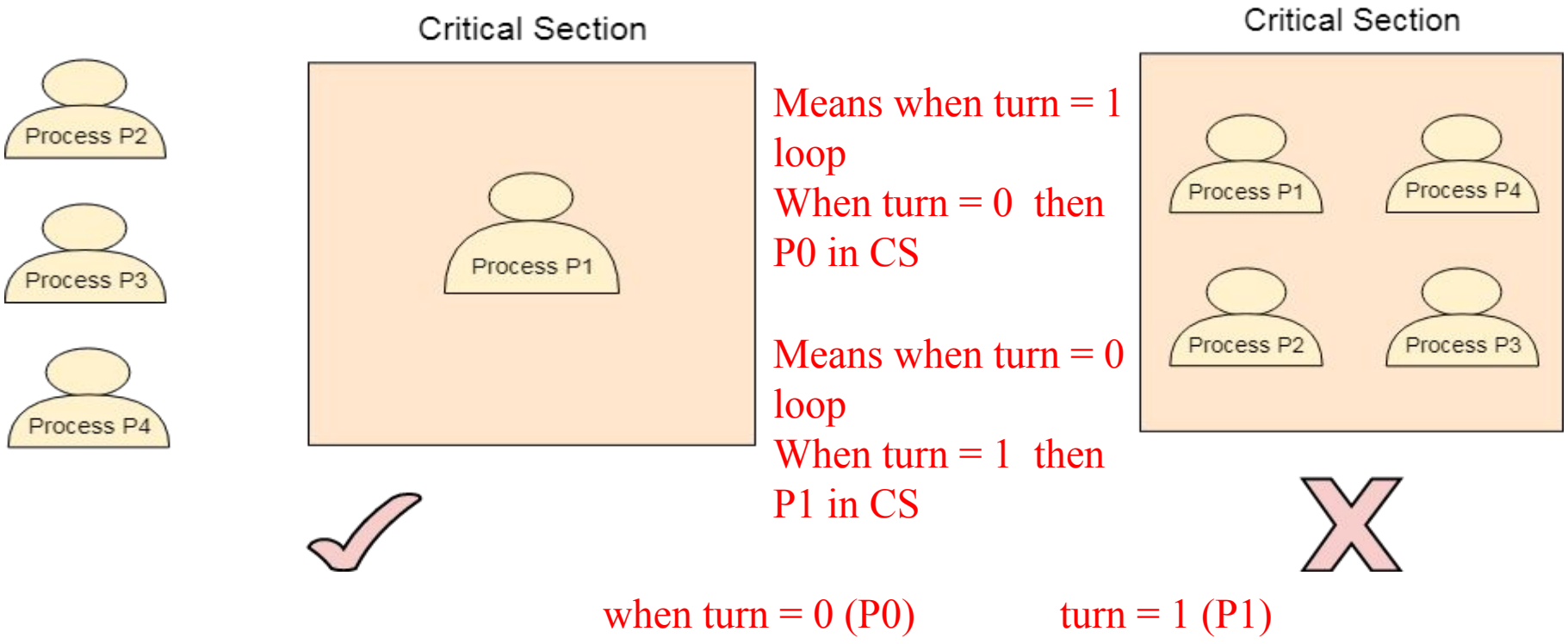
There are essentially three approaches to implementing mutual exclusion.

• Leave the responsibility with the processes themselves: this is the basis of most software approaches. These approaches are usually highly error-prone and carry high overheads.

• Allow access to shared resources only through special-purpose machine instructions:

i.e. a hardware approach. These approaches are faster but still do not offer a complete solution to the problem, e.g. they cannot guarantee the absence of deadlock and starvation.

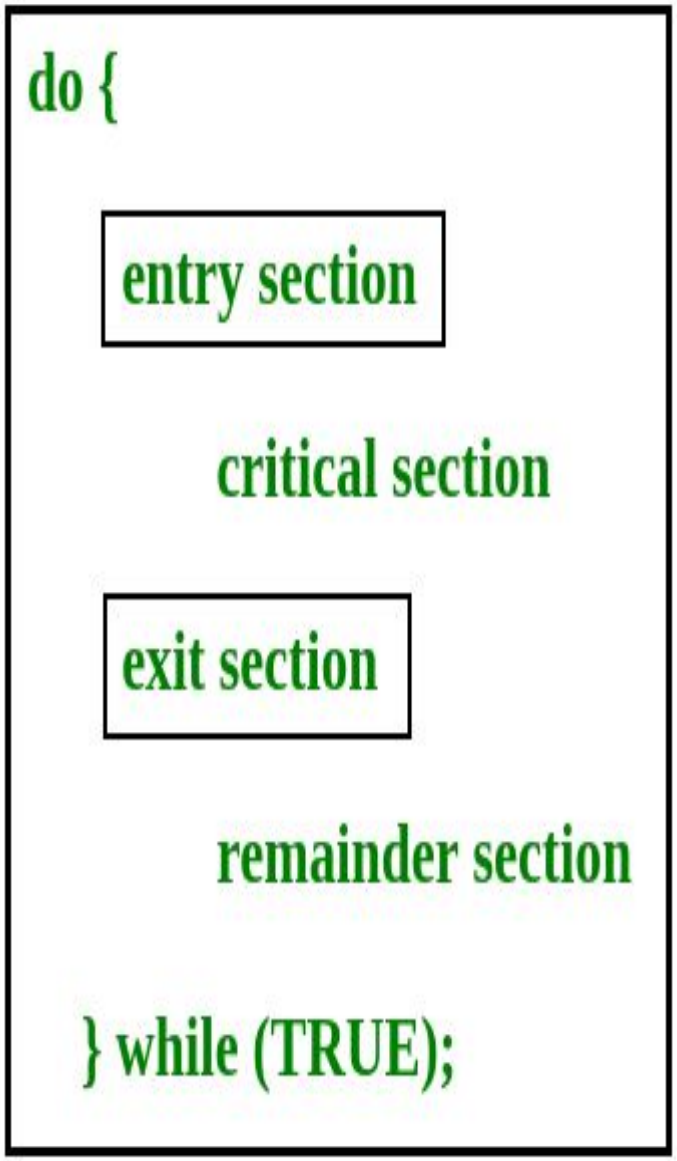
• Provide support through the operating system, or through the programming language. We shall outline three approaches in this category: semaphores, monitors, and message passing.




**Mutual Exclusion :-** It implies that only one process can be inside the critical section at any time. If any other processes require the critical section, they must wait until it is free.

**Progress :-** Means that if a process is not using the critical section, then it should not stop any other process from accessing it. In other words, any process can enter a critical section if it is free.

**Bounded Waiting :-** It means that each process must have a limited waiting time. It should not wait endlessly to access the critical section.




```
P0
while (1)
{
    while(turn!=0);
    Critical Section
    turn = 1
    Reminder section
}
```

```
P1
while (1)
{
    while(turn!=1);
    Critical Section
    turn =0
    Reminder section
}
```

turn = 0
**Or**
turn = 1

turn = 0
**Or**
turn = 1

| P0 | P1 |
|---|---|
| F | F |

turn = 1

turn = 0

```
P0
while (1)
{
      flag[0]= T;
      while(flag[1]);
      Critical Section
      flag[0]=F;
      Reminder section
}
```

```
P1
while (1)
{
      flag[1]= T;
      while(flag[0]);
      Critical Section
      flag[1]=F;
      Reminder section
}
```

| P0 | P1 |
|---|---|
| F | F |
| flag[0] | flag[1] |

```
P0
while (1)
{
      flag[0]= T;
      turn = 1;
      while(turn==1 && flag[1]==T);
      Critical Section
      flag[0]=F;
      Reminder section
}
```

```
P1
while (1)
{
      flag[1]= T;
      turn=0;
      while(turn==0  &&  flag[0]==T);
      Critical Section
      flag[1]=F;
      Reminder section
}
```

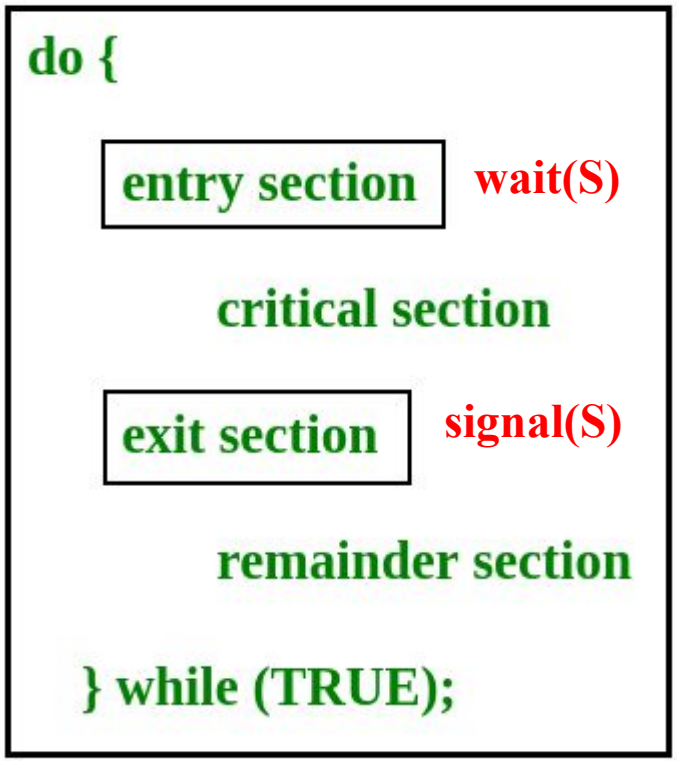
| P0<br>flag[0] | P1<br>flag[1] |
|---|---|
| T / F | T / F |

| Turn |
|---|
| F / T |

## Semaphore – Integer Variable (int S)

Which restrict access to shared resources in a multi processing environment. The two most common kinds of semaphore are counting semaphore and binary semaphore.

It is assessed through standard atomic operations - wait () & signal()




```
wait(S)
{
    while(S<=0);
     S = S – 1 ;
}
```

```
signal(S)
{
    S = S + 1 ;
}
```

# Producer & Consumer Problem

S = semaphore = 1 E = Empty Slots = n F = Full Slots = 0

| 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 |
|---|---|---|---|---|---|---|---|---|---|
| | | | | | | | | | |

Now
S = 1 E = 10 F = 00

**Conditions :-**

1) Consumer / producer should not work at a time.

2) Consumer should consume if any product is available.

3) Producer should produce if empty cell is there to store.

4) https://www.google.com/search?q=mutex+in+os&rlz=1C1FKPE_en-GB&tbm=vid&ei=dTiAZMWzAfWKseMPn_Kr2Aw&start=10&sa=N&ved=2ahUKEwiFwt7r17D_AhV1RWwGHR_5CssQ8NMDegQIGBAW&biw=1024&bih=657&dpr=1#fpstate=ive&vld=cid:3a2ccbb1,vid:8wcuLCvMmF8

| 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 |
|---|---|---|---|---|---|---|---|---|---|
| | | | | | | | | | |

```
void Producer()
{
    while(T)
    {
        produce();
        wait(E);
        wait(S);
        append();
        signal(S);
        signal(F);
    }
}
```

```
void Consumer()
{
    while(T)
    {
        wait(F);
        wait(S);
        take();
        signal(S);
        signal(E);
        Use();
    }
}
```

## Reader Writer Problem

**red = 0**
**wrt = 0**
**readcount=0**

### For Reader

```
wait (red)
read-count ++
if (readcount==1)
    wait (wrt)
signal red
```

### Read Operation

```
wait(red)
 read-count—
 if(readcount==0)
        signal (wrt)
 signal (red)
```

### For Writer

```
wait (wrt)
```

### Write Operation

```
signal (wrt)
```

**Reader – Reader** **Writer**

**red=1 wrt=1** **wrt = 0**

**readcount=0**

**red=0** **red=0**

**rc=1** **rc=2**

**wrt=0**

**red=1** **red=1**

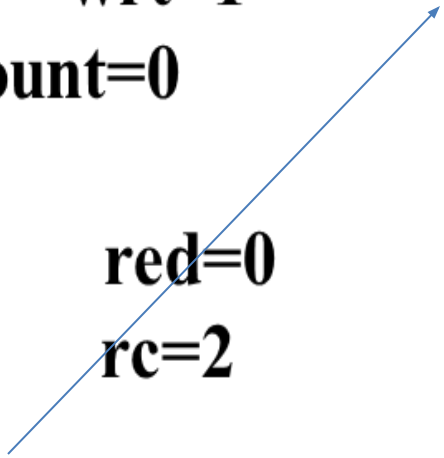
**R1** **R2**

**Exit Part**

**red=0**

**rc=1**

**red=1**



## Semaphores

A semaphore is hardware or a software tag variable whose value indicates the status of a common resource.

Its purpose is to lock the resource being used. A process which needs the resource will check the semaphore for determining the status of the resource followed by the decision for proceeding.

In multitasking operating systems, the activities are synchronized by using the semaphore techniques.

A semaphore is a variable. There are 2 types of semaphores:

1) Binary semaphores  2) Counting semaphores

Binary semaphores have 2 methods associated with it. (up, down / lock, unlock)

Binary semaphores can take only 2 values (0/1).

They are used to acquire locks.

When a resource is available, the process in charge set the semaphore to 1 else 0.

Counting Semaphore may have value to be greater than one, typically used to allocate resources from a pool of identical resources.

The fundamental idea of semaphores is that processes “communicate” via global counters that are initialized to a positive integer and that can be accessed only through two atomic operations (many different names are used for these operations: the following names are those used in Stallings Page 216):

semSignal(x) increments the value of the semaphore x.

semWait(x) tests the value of the semaphore x: if x > 0, the process decrements x and continues; if x = 0, the process is blocked until some other process performs a semSignal, then it proceeds as above.

A critical code section is then protected by bracketing it between these two operations:

semWait (x); <critical code section> semSignal (x);

In general the number of processes that can execute this critical section simultaneously is determined by the initial value given to x. If more than this number try to enter the critical section, the excess processes will be blocked until some processes exit. Most often, semaphores are initialized to one.

**MUTEX**

A mutex and the binary semaphore are essentially the same. Both can take values: 0 or 1. However, there is a significant difference between them that makes mutexes more efficient than binary semaphores.

A mutex can be unlocked only by the thread that locked it. Thus a mutex has an owner concept. Mutex is the short form for 'Mutual Exclusion object'. A mutex allows multiple threads for sharing the same resource. The resource can be file. A mutex with a unique name is created at the time of starting a program. A mutex must be locked from other threads, when any thread that needs the resource. When the data is no longer used / needed, the mutex is set to unlock.

# Monitors

The principal problem with semaphores is that calls to semaphore operations tend to be distributed across a program, and therefore these sorts of programs can be difficult to get correct, and very difficult indeed to prove correct!

Monitors address this problem by imposing a higher-level structure on accesses to semaphore variables. A monitor is essentially an object (in the Java sense) which has the semaphore variables as internal (private) data and the semaphore operations as (public) operations. Mutual exclusion is provided by allowing only one process to execute the monitor's code at any given time.

Monitors are significantly easier to validate than "bare" semaphores for at least two reasons:

- all synchronization code is confined to the monitor; and
- once the monitor is correct, any number of processes sharing the resource will operate correctly.

Concurrency issues (expressed using processes):

- **Atomic.** An operation is atomic if the steps are done as a unit. Operations that are not atomic, but interruptible and done by multiple processes can cause problems. For example, an lseek followed by a write is not atomic. A process is likely to lose its time quantum between the lseek (a slow operation if the distance seeked is large!) and the write. If another process has the file open and does a write then the result is not what is intended.
- **Race conditions.** A race condition occurs if the outcome depends on which of several processes gets to a point first. For example, fork( ) can generate a race condition if the result depends on whether the parent or the child process runs first. Other race conditions can occur if two processes are updating a global variable.

- **Blocking** and **starvation**. While neither of these problems is unique to concurrent processes, their effects must be carefully considered. Processes can *block* waiting for resources. A process could be blocked for a long period of time waiting for input from a terminal. If the process is required to periodically update some data, this would be very undesirable. *Starvation* occurs when a process does not obtain sufficient CPU time to make meaniful progress.
- **Deadlock**. Deadlock occurs when two processes are blocked in such a way that neither can proceed. The typical occurrence is where two processes need two non-shareable resources to proceed but one process has acquired one resource and the other has acquired the other resource. Acquiring resources in a specific order can resolve some deadlocks.

## Deadlock

Deadlock is defined as the permanent blocking of a set of processes that either compete for global resources or communicate with each other. It occurs when each process in the set is blocked awaiting an event that can be triggered only by another blocked process in the set.

Consider Figure 6.2 (all figures are taken from Stallings' web-site), in which both processes P and Q need both resources A and B simultaneously to be able to proceed. Thus P has the form get A, ... get B, ..., release A, ..., release B, and Q has the form get B, ... get A, ..., release B, ..., release A.

# Banker's Algorithm – Deadlock Avoidance

| Allocation Matrix | A | B | C |
|---|---|---|---|
| P0 | 0 | 1 | 0 |
| P1 | 2 | 0 | 0 |
| P2 | 3 | 0 | 2 |
| P3 | 2 | 1 | 1 |
| P4 | 0 | 0 | 2 |

| Maximum Resource Types | A | B | C |
|---|---|---|---|
| | 10 | 5 | 7 |

| Currently Available | A | B | C |
|---|---|---|---|
| | 3 | 3 | 2 |

| MAX Required | A | B | C |
|---|---|---|---|
| P0 | 7 | 5 | 3 |
| P1 | 3 | 2 | 2 |
| P2 | 9 | 0 | 2 |
| P3 | 2 | 2 | 2 |
| P4 | 4 | 3 | 3 |

**1) Find the need**

**(Max- Allocation)**

**2) Find safe sequence**

| Need Matrix | A | B | C |
|---|---|---|---|
| P0 | 7 | 4 | 3 |
| P1 | 1 | 2 | 2 |
| P2 | 6 | 0 | 0 |
| P3 | 0 | 1 | 1 |
| P4 | 4 | 3 | 1 |

**P0**

**Need > Available**

**743 > 332 (True)**

**Don't Execute**

**P1**

**Need > Available**

**122 > 332 (False)**

**Executed**

**New available resources after P1**

**332+ 200 = 532**

**P2**

**Need > Available**

**600 > 532 (True)**

**Don't Execute**

**P3**

**Need > Available**

**011 > 532 (False)**

**Executed**

**New available resources after P3**

**532+ 211 = 743**

**P4**

**Need > Available**

**431 > 743 (False)**

**Executed**

**New available resources after P4**

**743+ 002 = 745**

**Again P0 (Second Try)**

**Need > Available**

**743 > 745 (False)**

**Executed**

**New available resources after P0**

**745+ 010 = 755**

**Again P2 (Second Try)**

**Need > Available**

**600 > 755 (False)**

**Executed**

**New available resources after P2**

**755+ 302 = 10 5 7**

| Maximum Resource Types | A | B | C |
|---|---|---|---|
| | 10 | 5 | 7 |

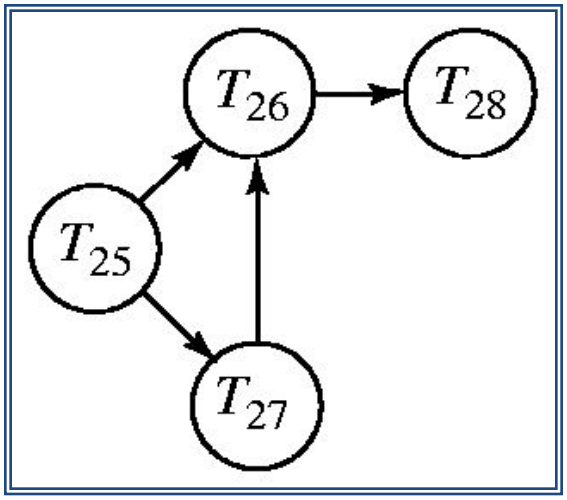
**Lastly available resources should be match with Resources Given in example.**

**So safe sequence of processes to be adopted to avoid deadlock is**

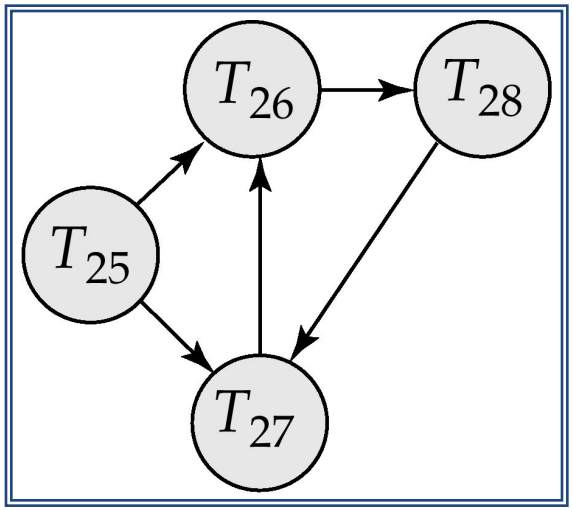
**< P1 – P3 - P4 – P0 – P2 >**

# Deadlock Detection

- Deadlocks can be described as a *wait-for graph* where:
  - vertices are all the transactions in the system
  - There is an edge $T_i \rightarrow T_k$ in case $T_i$ is waiting for $T_k$
- When $T_i$ requests a data item currently being held by $T_k$, then the edge $T_i \rightarrow T_k$ is inserted in the wait-for graph. This edge is removed only when $T_k$ is no longer holding a data item needed by $T_i$.
- The system is in a deadlock state if and only if the wait-for graph has a cycle. Must invoke a deadlock-detection algorithm periodically to look for cycles.




Wait-for graph without a cycle




Wait-for graph with a cycle

The six paths depicted are as following.

1. Q acquires both resources, then releases them. P can operate freely later.
2. Q acquires both resources, then P requests A. P is blocked until the resources are released, but can then operate freely.
3. Q acquires B, then P acquires A, then each requests the other resource. Deadlock is now inevitable. **(expected)**
4. P acquires A, then Q acquires B, then each requests the other resource. Deadlock is now inevitable.
5. P acquires both resources, then Q requests B. Q is blocked until the resources are released, but can then operate freely.
6. P acquires both resources, then releases them. Q can operate freely later.

**The differences between binary semaphore and mutex are:**

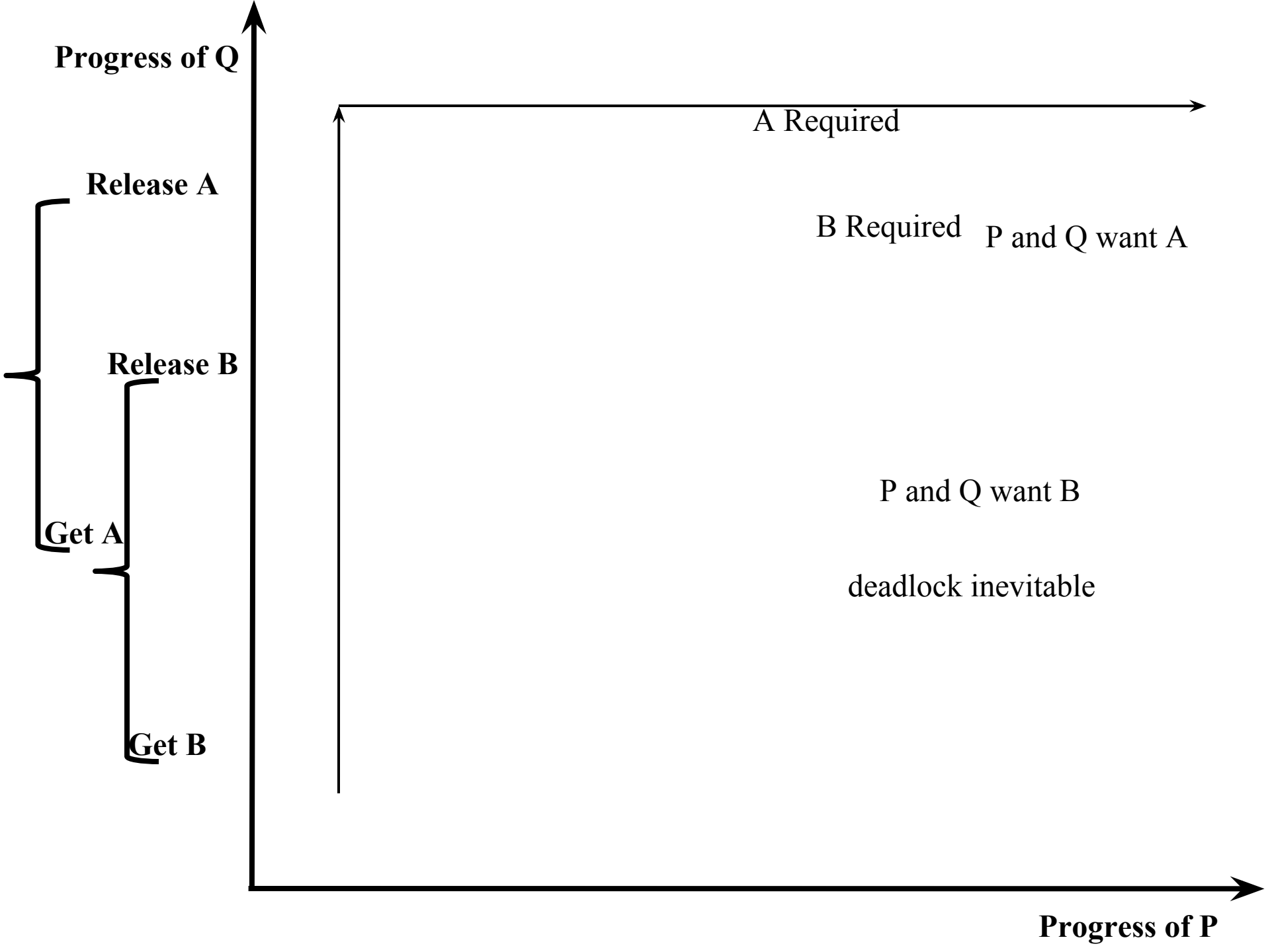
Mutex is used exclusively for mutual exclusion. Both mutual exclusion and synchronization can be used by binary.

A task that took mutex can only give mutex.

From an ISR a mutex can not be given.

Recursive taking of mutual exclusion semaphores is possible. This means that a task that holds before finally releasing a semaphore, can take the semaphore more than once.

Options for making the task which takes as DELETE_SAFE are provided by Mutex, which means the task deletion is not possible when holding the mutex.


Progress of Q
A Required
Release A
B Required
P and Q want A
Release B
P and Q want B
Get A
deadlock inevitable
Get B
Progress of P

**Unit IV** **(8 Hrs)**

**Memory management :** Contiguous and non-contiguous, Swapping, Paging, Segmentation and demand Paging, Virtual Memory, Management of Virtual memory: allocation, fetch and replacement

## Memory Management Requirements

### 1) Relocation :-

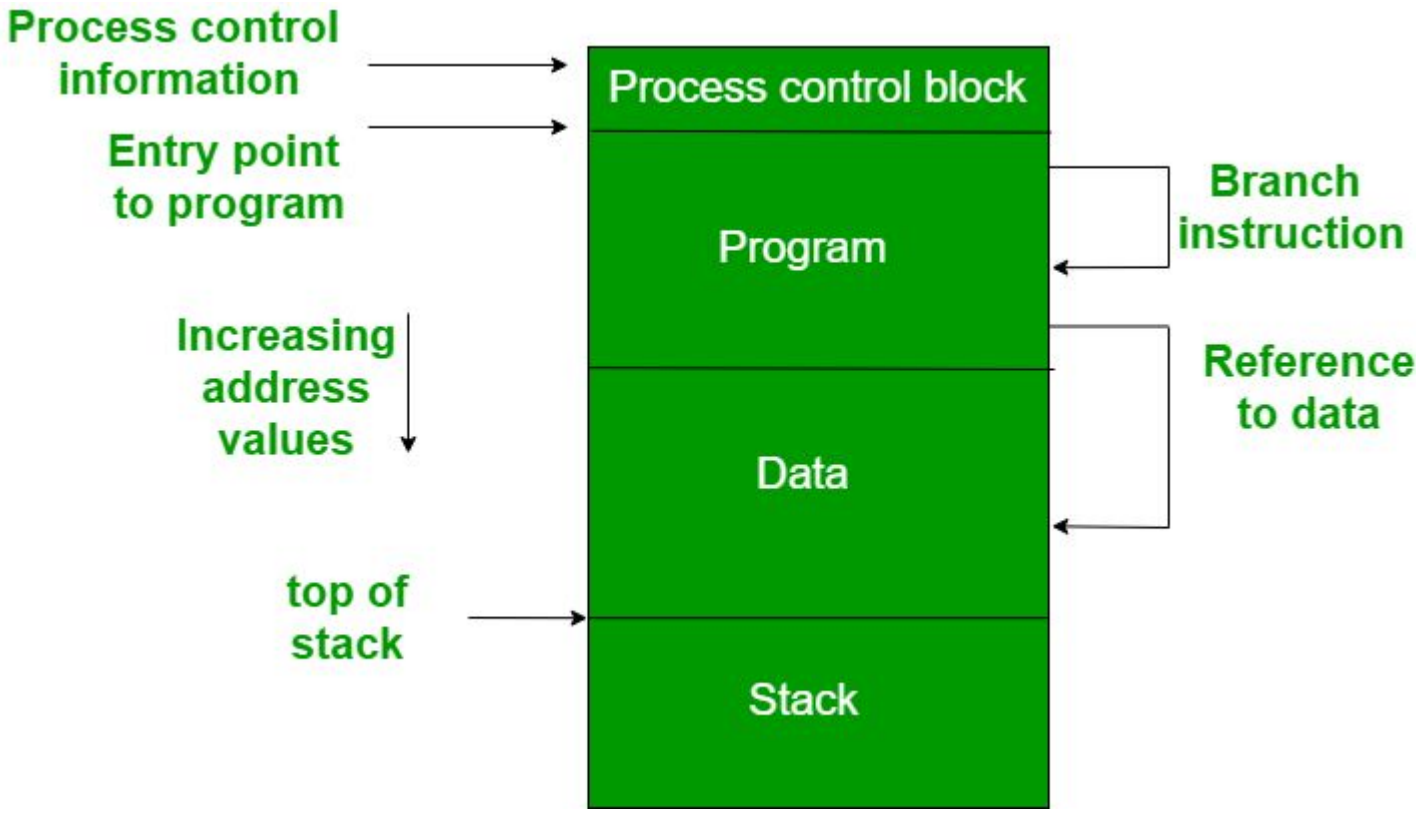
The available memory is generally shared among the number of processes in a multiprogramming system, so it is not possible to know in advance which other programs will be resident in main memory at the time of execution of his program. Due to Swapping the active process may swapped with other. Also due to swapping out , the operating system have a larger pool of ready-to-execute process.

When a program gets swapped out to a disk memory, then it is not always possible that when it is swapped back into main memory then it occupies the previous memory location, since the location may still be occupied by another process. We may need to relocate the process to a different area of memory. Thus there is a possibility that program may be moved in main memory due to swapping.

All the time the operating system will need to know many things including the location of process control information, the execution stack, and the code entry.

Within a program, there are memory references in various instructions and these are called logical addresses.

After loading of the program into main memory, the processor and the operating system must be able to translate logical addresses into physical addresses. Branch instructions contain the address of the next instruction to be executed.



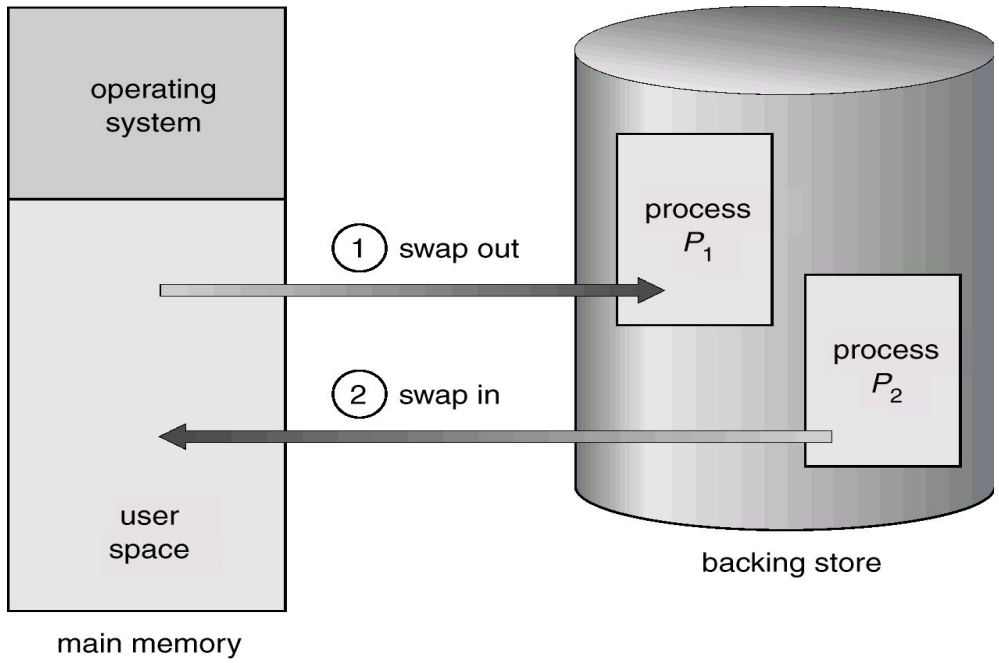
**Swapping :-**

A process can be *swapped* temporarily out of memory to a *backing store*, and then brought back into memory for continued execution.

Backing store – fast disk large enough to accommodate copies of all memory images for all users; 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.

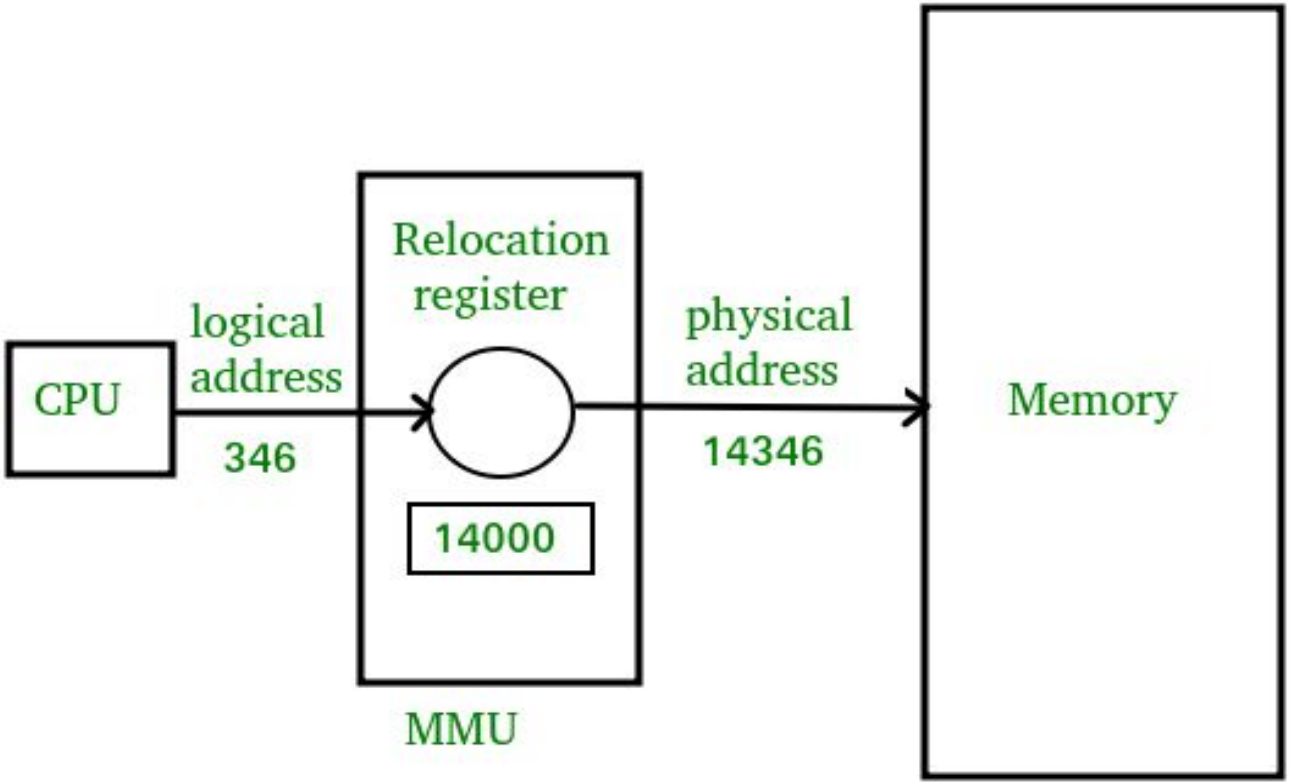
Major part of swap time is transfer time; total transfer time is directly proportional to the *amount* of memory swapped.



**Logical Address** is generated by CPU while a program is running. The logical address is virtual address as it does not exist physically, therefore, it is also known as Virtual Address. This address is used as a reference to access the physical memory location by CPU.
The hardware device called Memory-Management Unit is used for mapping logical address to its corresponding physical address.

**Physical Address** identifies a physical location of required data in a memory. The user never directly deals with the physical address but can access by its corresponding logical address. The user program generates the logical address and thinks that the program is running in this logical address but the program needs physical memory for its execution, therefore, the logical address must be mapped to the physical address by MMU before they are used.

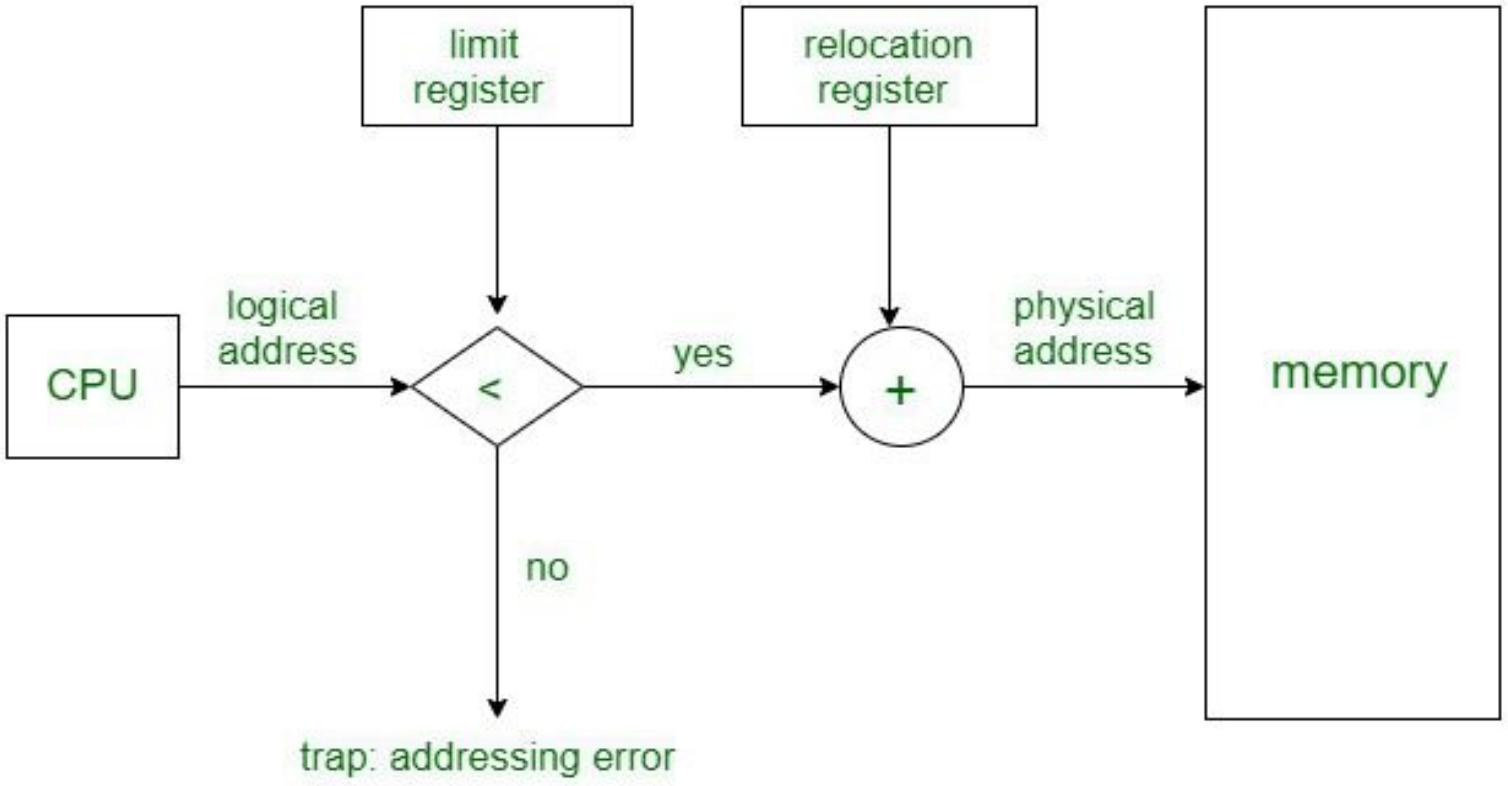
Program must be brought into memory and placed within a process for it to be executed.
Input queue – collection of processes on the disk that are waiting to be brought into memory for execution.
User programs go through several steps before being executed.




| OS |
| --- |
| 0000 |
| |
| |
| |
| |
| 13999 |
| 14000 |
| --- |
| ---- |
| 14346 |
| ----- |
| ----- |
| 15000 |
| |
| |

In MMU (Hardware device that maps virtual to physical address.) scheme, the value in the relocation register is added to every address generated by a user process at the time it is sent to memory.

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




The Memory Management Unit is a combination of 2 registers –

1) Base Register (Relocation Register) 2) Limit Register.

**Base Register –** contains the starting physical address of the process.

**Limit Register** -mentions the limit relative to the base address on the region occupied by the process.

The logical address generated by the CPU is first checked by the limit register, If the value of the logical address generated is less than the value of the limit register, the base address stored in the relocation register is added to the logical address to get the physical address of the memory location.

If the logical address value is greater than the limit register, then the CPU traps to the OS, and the OS terminates the program by giving fatal error.

**In Non Contiguous Memory allocation, processes can be allocated anywhere in available space. The address translation in non-contiguous memory allocation is difficult.**

| Column 1 | Column 2 | Column3 | Column 4 | Column 5 |
|---|---|---|---|---|
| Desk11 01 | Desk21 08 | Desk31 15 | Desk41 22 | Desk51 29 |
| Desk12 02 | Desk22 09 | Desk32 16 | Desk42 23 | Desk52 30 |
| Desk13 03 | Desk23 10 | Desk33 17 | Desk43 24 | Desk53 31 |
| Desk14 04 | Desk24 11 | Desk34 18 | Desk44 25 | Desk54 32 |
| Desk15 05 | Desk25 12 | Desk35 19 | Desk45 26 | Desk55 33 |
| Desk16 06 | Desk26 13 | Desk36 20 | Desk46 27 | Desk56 34 |
| Desk17 07 | Desk27 14 | Desk37 21 | Desk47 28 | Desk57 35 |

Logical Address Desk 36 (It indicates Column 3 and desk No. 6)

But actual desk no is the physical place or Physical address which will be calculated by MMU

Every column contains 7 desk. As given address is from column no 3. So **7+7+6** is the

**physical no i.e. 20**

**2) Sharing :-** Protection mechanism must have the flexibility to allow several processes to access the same portion of the main memory. e.g. if the number of process are executing the same program then it is advantageous to allow each process to access the same copy of the program rather than its own separate copy.

**3) Protection:** When we have two program at the same time there is a danger that one program can write to the address space of another program so in the manner every process should be protected against unwanted interference by other processes. Satisfaction of the relocation requirement increase the difficulty of satisfying the protection requirement. CPU tend to support absolute addressing which means that code runs differently when loaded in different places. This is not possible to check the absolute address at the compile time. Most of the programming languages allow the dynamic calculation of the address at run time.

**4) Logical organization :-** As user write program in modules with different characteristics then logical organization be like instruction modules are execute-only, data modules are either read-only or read/write and some modules are private other are public. To effective deal with the user program, the operating system and hardware must support a basic form of module to provide the required protection and sharing.

**5) Physical organization :-** As we know computer memory is organized into two levels main and secondary memory. Main memory is a volatile which provide fast access at relatively high cost and secondary memory is non-volatile which is going to provide slower and cheaper access than main memory. The flow of information is mainly between main memory and secondary memory which is major system concern but sometimes it is impractical for the programmers understand how. As main memory is available for the program and for data also which may be insufficient for that programmer must use mechanism of overlaying that hold the

data and program that are needed at the given time. There is also another factor that going to affect programmer concern that is multiprogramming environment. In such environment, programmer does not know how much space is available.

The main memory can be broadly allocated in two ways –

**1) Contiguous Memory Allocation**

a) Fixed partition scheme b) Variable partition scheme.

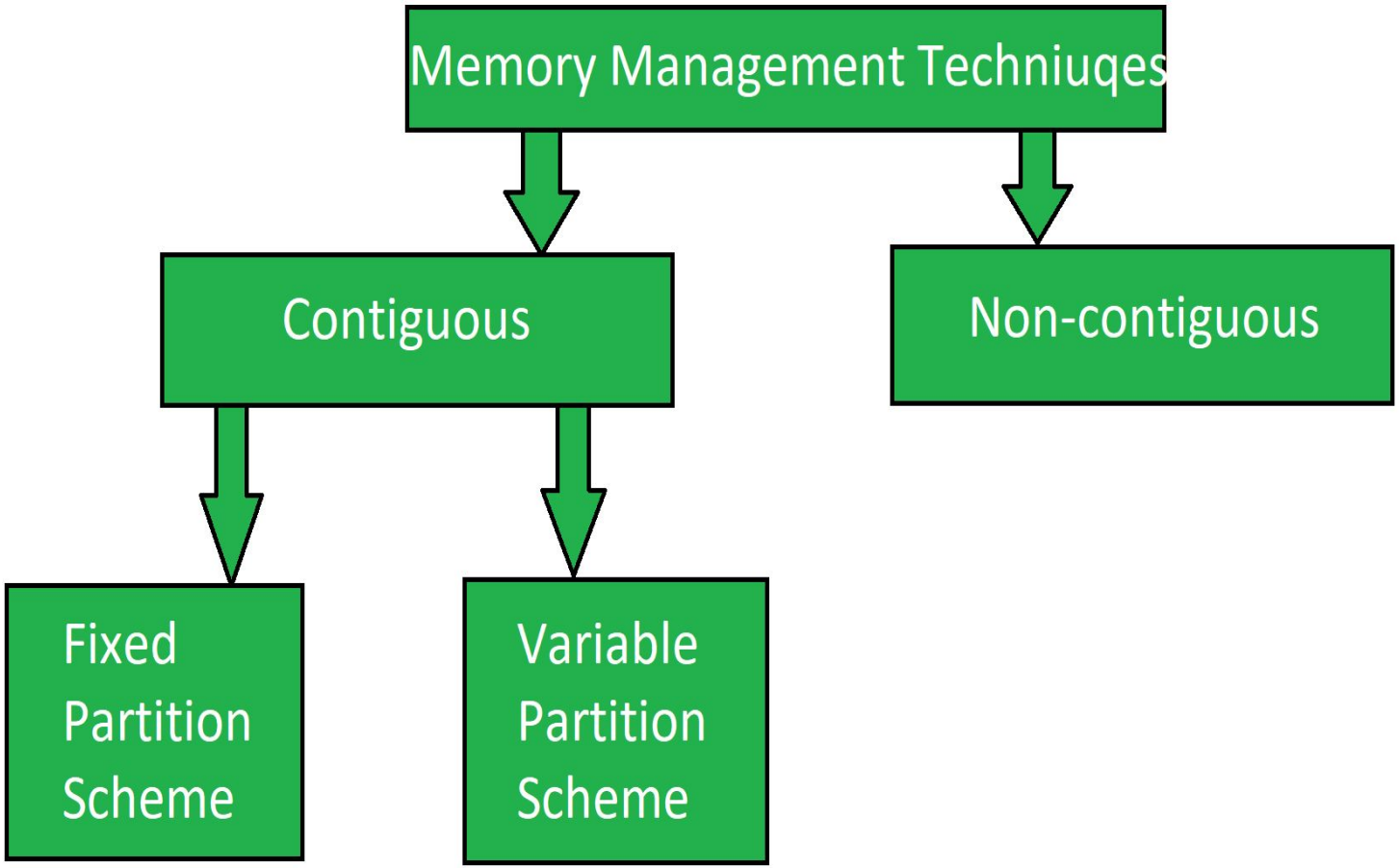
**Different Partition Allocation methods are used in Contiguous memory allocations**

a) First Fit b) Best Fit c) Worst Fit d) Next Fit

**2) Non-Contiguous Memory Allocation**

a) Paging b) Multitlevel Paging c) Inverted Paging

d) Segmentation e) Segmented Paging


Memory Management Techniuqes
Contiguous
Non-contiguous
Fixed Partition Scheme
Variable Partition Scheme

In the fixed partition scheme :-

i) memory is divided into fixed number of partitions.

ii) number of partitions are fixed in the memory.

iii) in every partition only one process will be accommodated.

iv) Degree of multi-programming is restricted by number of partitions in the memory.

Maximum size of the process is restricted by maximum size of the partition. Every partition is associated with the *limit registers*.

**Limit Registers:** It has two limit:

**Lower Limit:** Starting address of the partition.

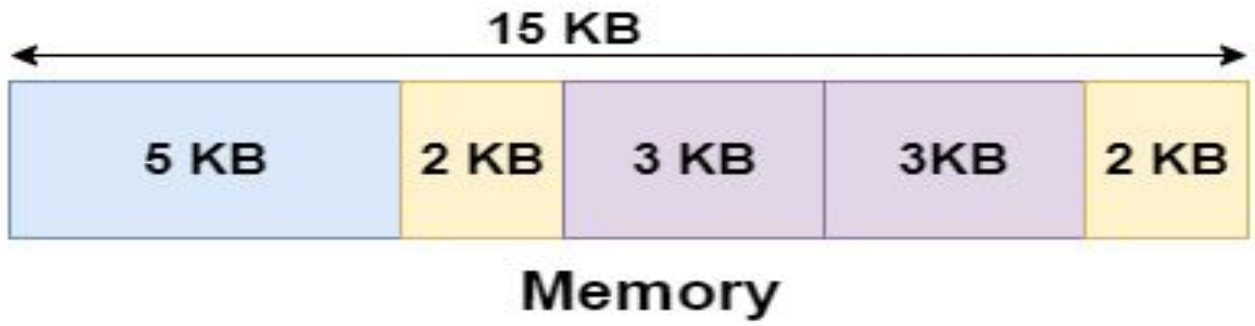
**Upper Limit:** Ending address of the partition.

**Fixed-size Partition Scheme :-**

This technique is also known as **Static partitioning**. In this scheme, the system divides the memory into fixed-size partitions. The partitions may or may not be the same size. The size of each partition is fixed as indicated by the name of the technique and it cannot be changed.

In this partition scheme, each partition may contain exactly one process. There is a problem that this technique will limit the degree of multiprogramming because the number of partitions will basically decide the number of processes.

Whenever any process terminates then the partition becomes available for another process.



## Advantages of Fixed-size Partition Scheme :-

This scheme is simple and is easy to implement

It supports multiprogramming as multiple processes can be stored inside the main memory.

Management is easy using this scheme

## Disadvantages of Fixed-size Partition Scheme :-

**1. Internal Fragmentation**

Suppose the size of the process is lesser than the size of the partition in that case some size of the partition gets wasted and remains unused. This wastage inside the memory is generally termed as **Internal fragmentation**

**2. Limitation on the size of the process**

If in a case size of a process is more than that of a maximum-sized partition then that process cannot be loaded into the memory. Due to this, a condition is imposed on the size of the process and it is: the size of the process cannot be larger than the size of the largest partition.

**3. Degree of multiprogramming is less**

In this partition scheme, as the size of the partition cannot change according to the size of the process. Thus the degree of multiprogramming is very less and is fixed.

**Variable-size Partition Scheme :-**

This scheme is also known as Dynamic partitioning and is came into existence to overcome the drawback i.e internal fragmentation that is caused by Static partitioning. In this partitioning, scheme allocation is done dynamically.

The size of the partition is not declared initially. Whenever any process arrives, a partition of size equal to the size of the process is created and then allocated to the process. Thus the size of each partition is equal to the size of the process.

As partition size varies according to the need of the process so in this partition scheme there is no internal fragmentation.

**Advantages :-**

1. **No Internal Fragmentation** :-

As in this partition scheme space in the main memory is allocated strictly according to the requirement of the process thus there is no chance of internal fragmentation. Also, there will be no unused space left in the partition.

**2. Degree of Multiprogramming is Dynamic :-**

As there is no internal fragmentation in this partition scheme due to which there is no unused space in the memory. Thus more processes can be loaded into the memory at the same time.

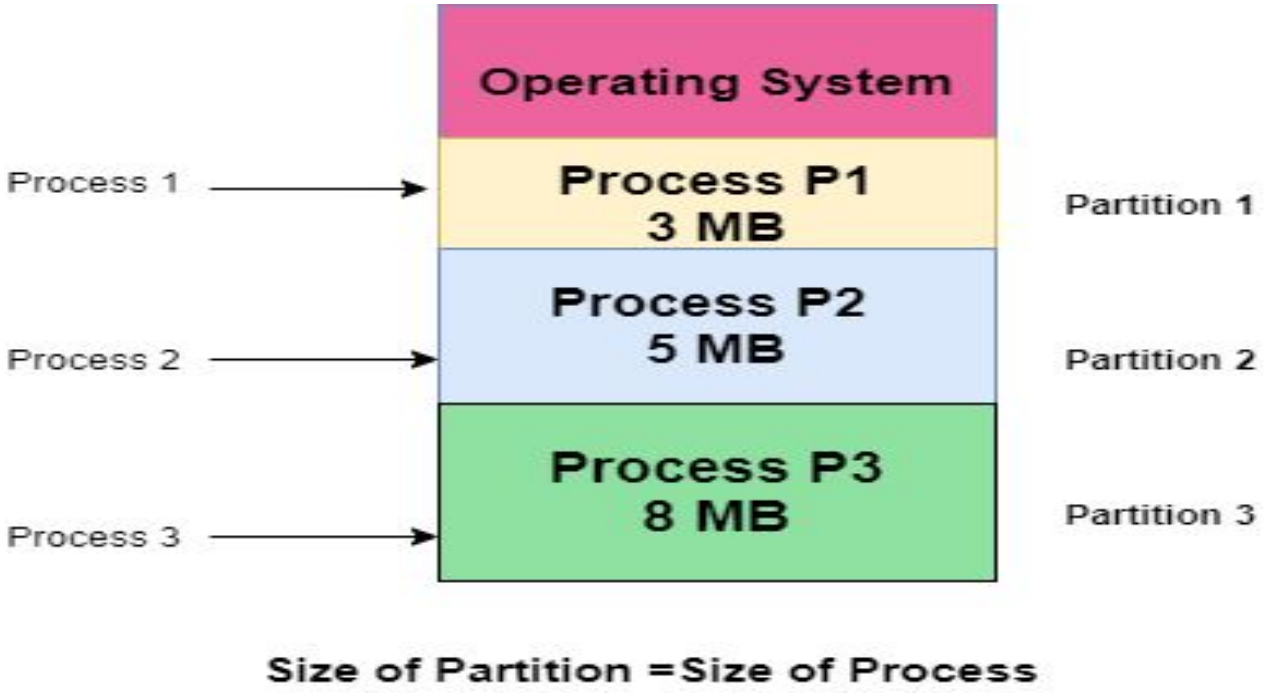
**3.No Limitation on the Size of Process :-**

In this partition scheme as the partition is allocated to the process dynamically thus the size of the process cannot be restricted because the partition size is decided according to the process size.

**Disadvantages :-**

1) **External Fragmentation** :- As there is no internal fragmentation which is an advantage of using this partition scheme does not mean there will no external fragmentation.

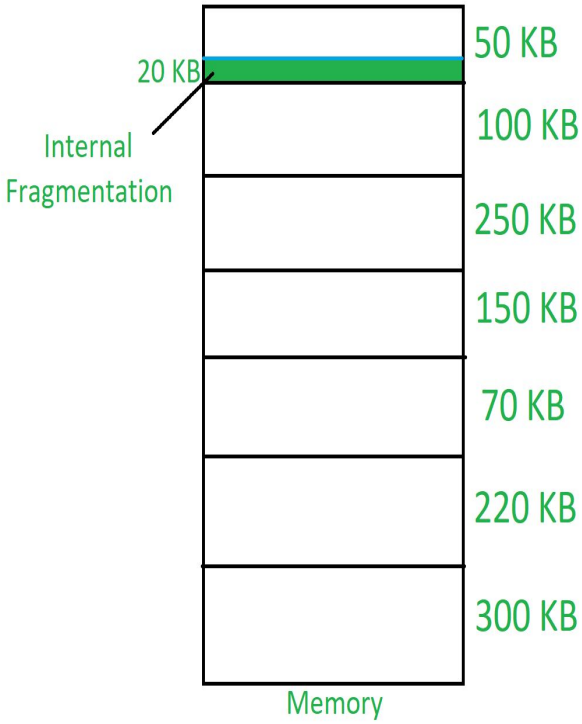
e.g.: In diagram- process P1(3MB) and process P3(8MB) completed their execution. Hence there are two spaces left i.e. 3MB and 8MB. Let's there is a Process P4 of size 15 MB comes. But the empty space in memory cannot be allocated as no spanning is allowed in contiguous allocation.




Because the rule says that process must be continuously present in the main memory in order to get executed. Thus it results in External Fragmentation.

**2. Difficult Implementation** :-

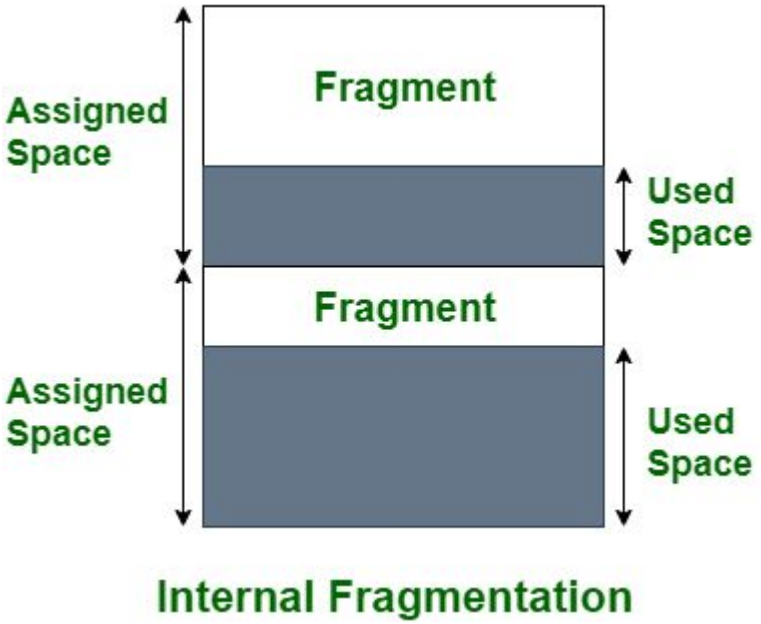
The implementation of this partition scheme is difficult as compared to the Fixed Partitioning scheme as it involves the allocation of memory at run-time rather than during the system configuration. As we know that OS keeps the track of all the partitions but here allocation and deallocation are done very frequently and partition size will be changed at each time so it will be difficult for the operating system to manage everything.




P = 30 KB

**In the given figure 50 KB partition is used to load a process of 30 KB. So the remaining 20 KB is wasted.**

Internal fragmentation



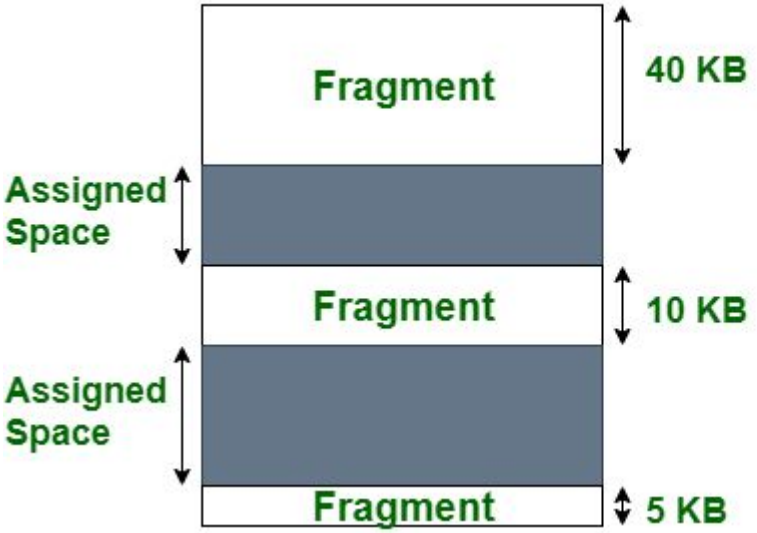
**Internal fragmentation found into fixed sized blocks.**

Whenever a process request for the memory, the fixed sized block is allotted to the method. The memory allotted to the process is somewhat larger than the memory requested, then the distinction (difference) between allotted and requested memory is that the **Internal fragmentation.**

**External Fragmentation is found in variable partition scheme.**

Initially the memory is a single continuous free block. Whenever the request by the process arrives, accordingly partition will be made in the memory. If the smaller processes keep on coming then the larger partitions will be made into smaller partitions.
It happens when there's a sufficient quantity of area within the memory to satisfy the memory request of a method. however the process's memory request cannot be fulfilled because the memory offered is during a non-contiguous manner.

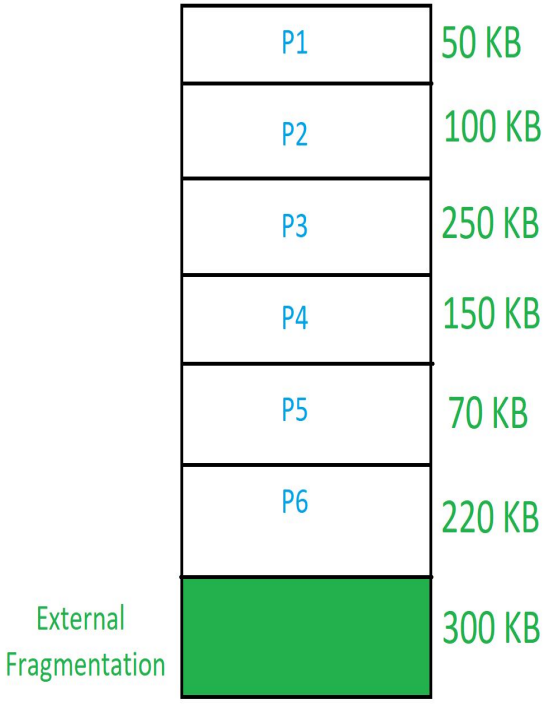
To overcome the problem of external fragmentation, compaction technique is used or non-contiguous memory management techniques are used.



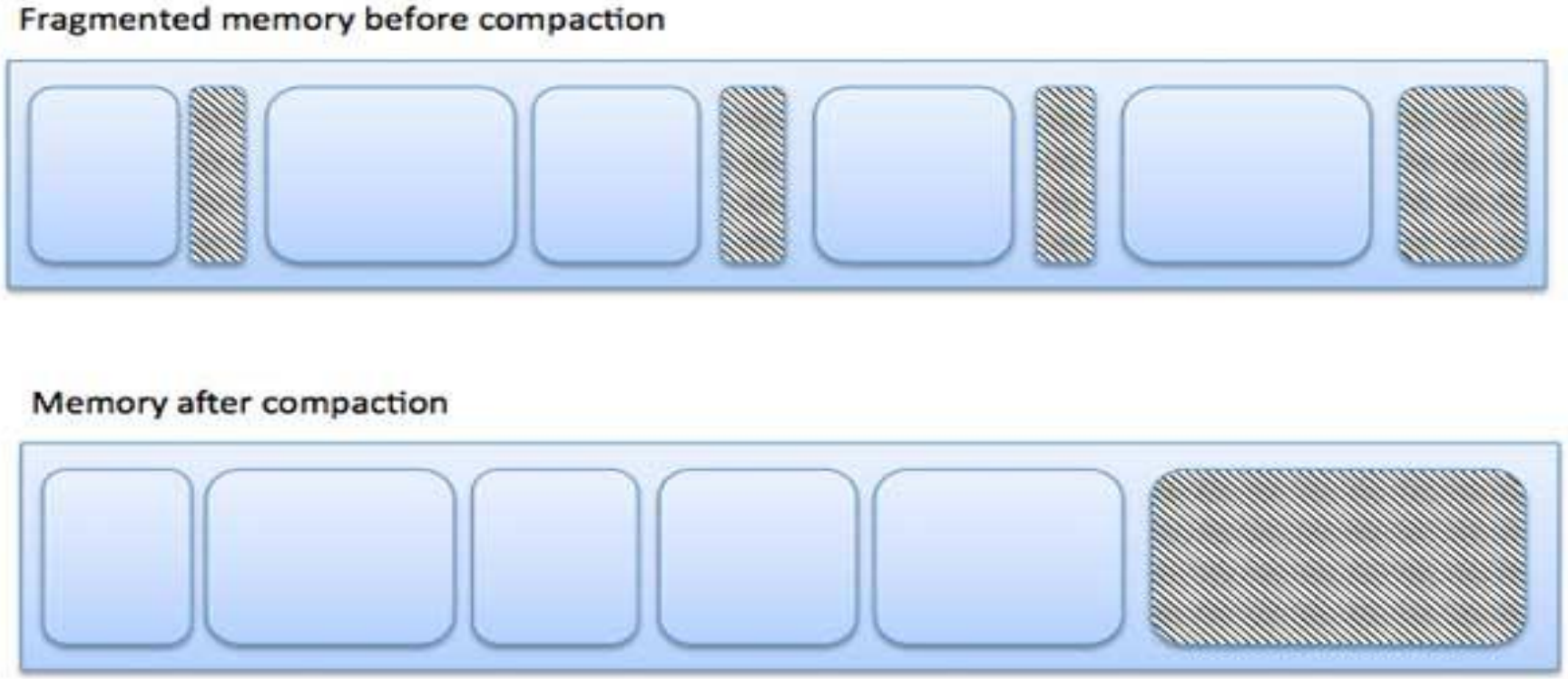

Process 07 needs 50KB memory space

Here is enough space (55 KB) to run a process-07 (required 50 KB) but the memory (fragment) is not contiguous. Here, we use compaction, paging or segmentation to use the free space to run a process.

Moving all the processes toward the top or towards the bottom to make free available memory in a single continuous place is called as compaction. Compaction is undesirable to implement because it interrupts all the running processes in the memory.



| Sr. No. | Internal fragmentation | External fragmentation |
|---|---|---|
| 1. | In internal fragmentation fixed-sized memory, blocks square measure appointed to process. | In external fragmentation, variable-sized memory blocks square measure appointed to method. |
| 2. | Internal fragmentation happens when the method or process is larger than the memory. | External fragmentation happens when the method or process is removed. |
| 3. | The solution of internal fragmentation is best-fit block. | Solution of external fragmentation is compaction, paging and segmentation. |
| 4. | Internal fragmentation occurs when memory is divided into fixed sized partitions. | External fragmentation occurs when memory is divided into variable size partitions based on the size of processes. |
| 5. | The difference between memory allocated and required space or memory is called Internal fragmentation. | The unused spaces formed between non-contiguous memory fragments are too small to serve a new process, is called External fragmentation . |




**External fragmentation**

Total memory space is enough to satisfy a request or to reside a process in it, but it is not contiguous, so it cannot be used.

**Internal fragmentation**

Memory block assigned to process is bigger. Some portion of memory is left unused, as it cannot be used by another process.

Memory allocation is a process by which computer programs are assigned memory or space.

It is of three types :

**First Fit Allocation :-** The first hole that is big enough is allocated to the program.

**Best Fit Allocation :-** The smallest hole that is big enough is allocated to the program.

**Worst Fit Allocation :-** The largest hole that is big enough is allocated to the program.

**Different Partition Allocation methods are used in Contiguous memory allocations**

**1) First Fit :-**

In the first fit approach is to allocate the first free partition or hole large enough which can accommodate the process. It finishes after finding the first suitable free partition.

**Advantage**

Fastest algorithm because it searches as little as possible.

**Disadvantage**

The remaining unused memory areas left after allocation become waste if it is too smaller. Thus request for larger memory requirement cannot be accomplished.

Input : Memory Block Size = {100, 500, 200, 300, 600};

Process Size = {212, 417, 112, 426};

| Process No. | Process Size | Wasted Size | Block no. |
|---|---|---|---|
| 1 | 212 | 288 | 2 |
| 2 | 417 | 183 | 5 |
| 3 | 112 | 88 | 2 |
| 4 | 426 | Not Allocated | |

**Remaining Space 288+183+88+100+300 = 959**


**2) Best Fit :-**

The best fit deals with allocating the smallest free partition which meets the requirement of the requesting process. This algorithm first searches the entire list of free partitions and considers the smallest hole that is adequate. It then tries to find a hole which is close to actual process size needed.

**Advantage**

Memory utilization is much better than first fit as it searches the smallest free partition first available.

**Disadvantage**

It is slower and may even tend to fill up memory with tiny useless holes.

Input : Memory Block Size = {100, 500, 200, 300, 600};

Process Size = {212, 417, 112, 426};

| Process No. | Process Size | Wasted Size | Block no. |
|---|---|---|---|
| 1 | 212 | 88 | 4 |
| 2 | 417 | 83 | 2 |
| 3 | 112 | 88 | 3 |
| 4 | 426 | 174 | 5 |

Remaining Space **88+83+88+174+100 = 533**

**3) Worst fit :-**

In worst fit approach is to locate largest available free portion so that the portion left will be big enough to be useful. It is the reverse of best fit.

**Advantage**

Reduces the rate of production of small gaps.

**Disadvantage**

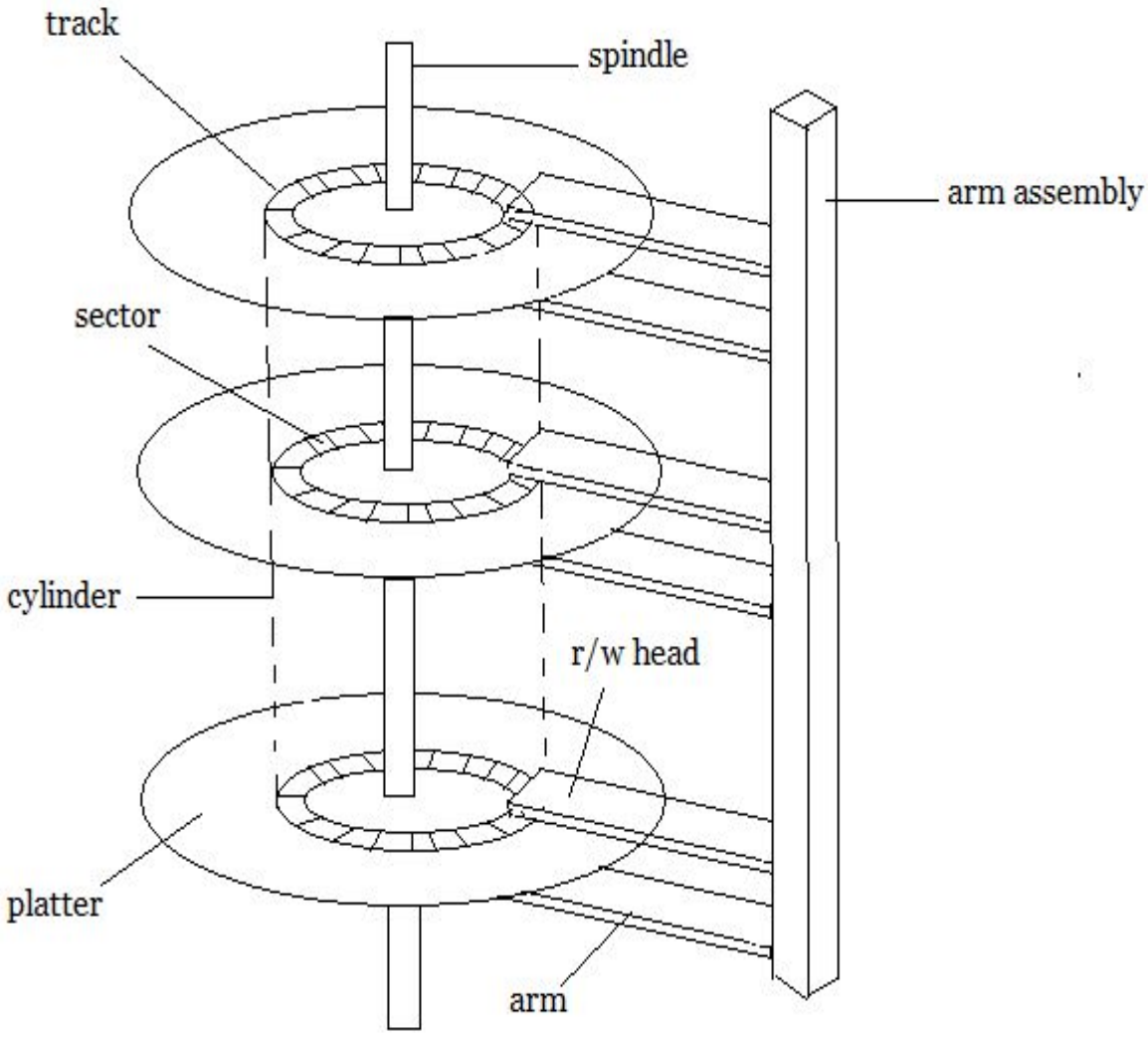
If a process requiring larger memory arrives at a later stage then it cannot be accommodated as the largest hole is already split and occupied.

Input : Memory Block Size = {100, 500, 200, 300, 600};

Process Size = {212, 417, 112, 426};

| Process No. | Process Size | Wasted Size | Block no. |
|---|---|---|---|
| 1 | 212 | 383 | 5 |
| 2 | 417 | 83 | 2 |
| 3 | 112 | 188 | 4 |
| 4 | 426 | **Not Accommodated** | |

Remaining Space **383+83+188+100+200 = 766**


track
spindle
arm assembly
sector
cylinder
r/w head
platter
arm

A magnetic disk contains several **platters**. Each platter is divided into circular shaped **tracks**. The length of the tracks near the centre is less than the length of the tracks farther from the centre. Each track is further divided into **sectors**,

Tracks of the same distance from centre form a cylinder. A read-write head is used to read data from a sector of the magnetic disk.

**Transfer Rate :-** This is the rate at which the data moves from disk to the computer.

**Random Access Time :-** It is the sum of the seek time and rotational latency.

**Seek Time :-** Is the time taken by the arm to move to the required track.

**Rotational Latency** :- It is defined as the time taken by the arm to reach the required sector in the track.

Even though the disk is arranged as sectors and tracks physically, the data is logically arranged and addressed as an array of blocks of fixed size. The size of a block can be 512 or 1024 bytes. Each logical block is mapped with a sector on the disk, sequentially. In this way, each sector in the disk will have a logical address.

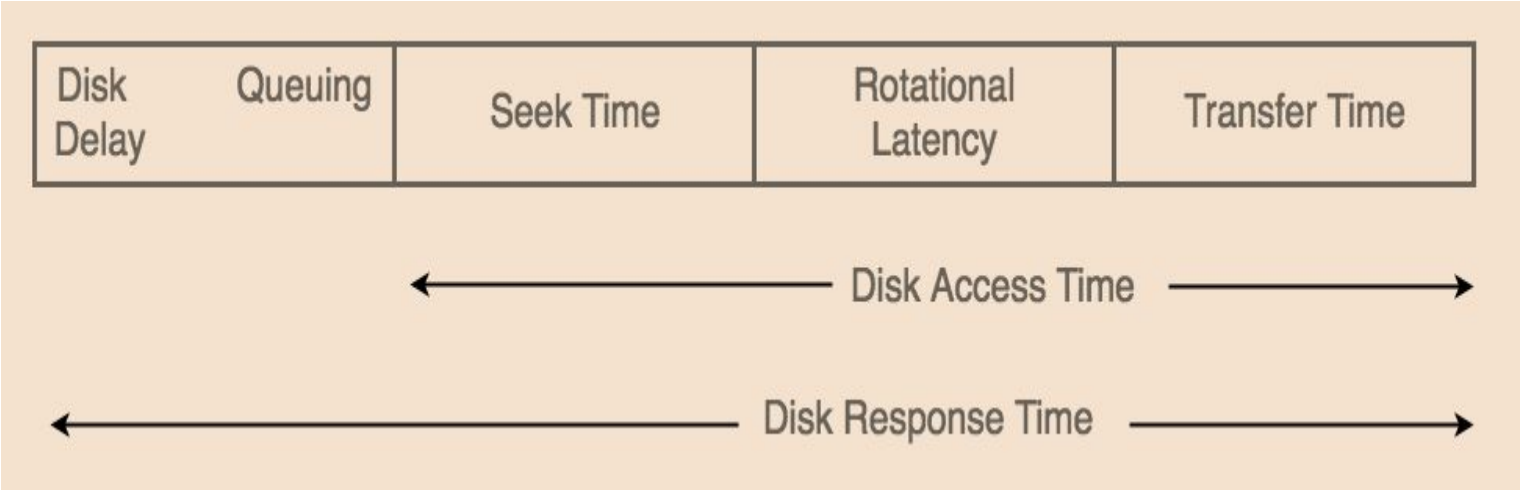
- **Seek Time :-** Seek time is the time taken to locate the disk arm to a specified track where the data is to be read or write. So the disk scheduling algorithm that gives minimum average seek time is better.
- **Rotational Latency :-** Rotational Latency is the time taken by the desired sector of 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 :- 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.



## 1. First Come First Serve :-

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.

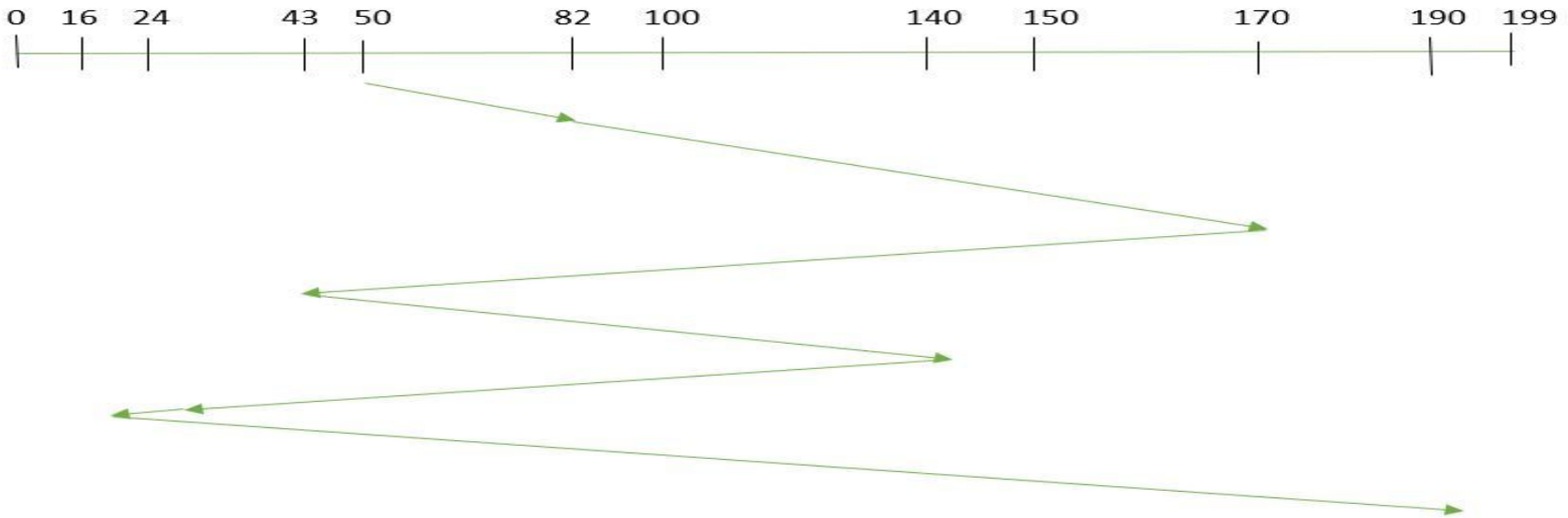
**Example :-**

Suppose the order of request is – (82,170,43,140,24,16,190)

Current position of Read/Write head is : 50

So, total seek time : = (82-50)+(170-82)+(170-43)+(140-43)+(140-24)+(24-16)+(190-16)

= 642



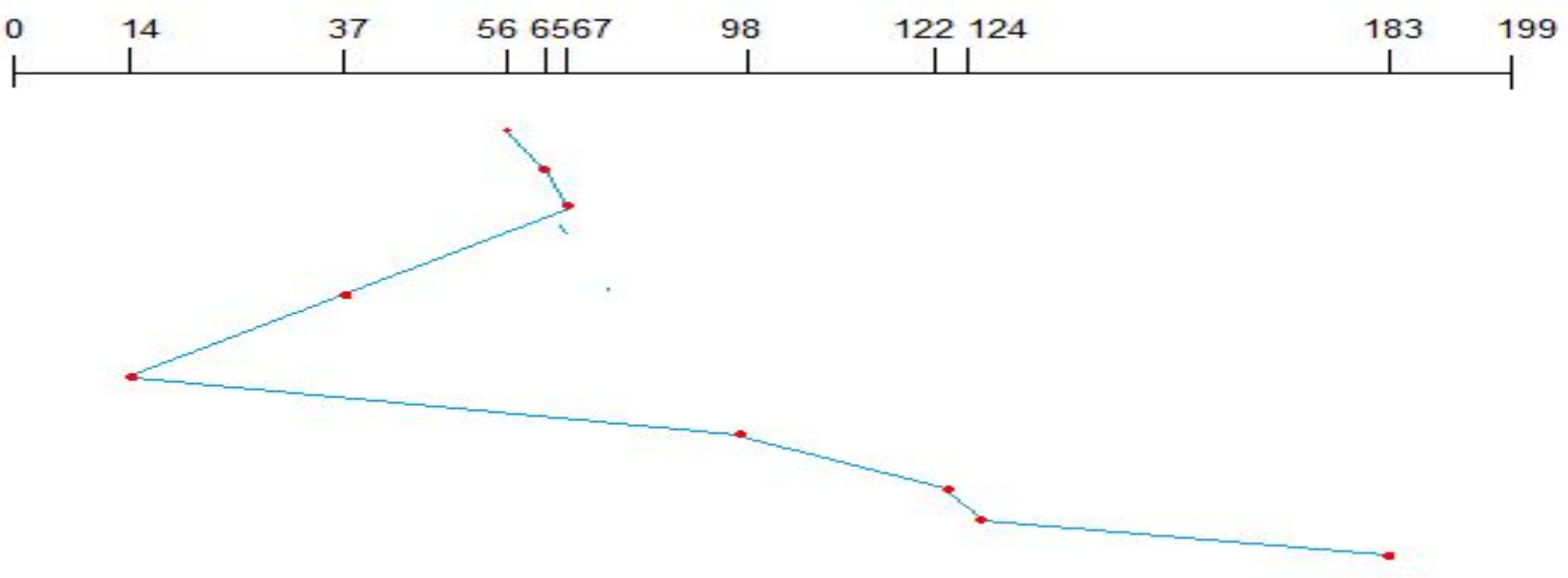

Every request gets a fair chance. But this algorithm does not try to optimize seek time & may not provide the best possible service

## 2) Shortest Seek Time First (SSTF) :-

The position which is closest to the current head position is chosen first.

Consider the example where disk order request queue is - **98, 183, 37, 122, 14, 124, 65, 67**

Assume the head is initially at cylinder **56**. The next closest cylinder to **56** is **65**, and then the next nearest one is **67**, then **37**, **14**, so on.




Total seek time := (65-56)+(67-65)+(67-37)+(37-14)+(98-14)+(122-98)+(124-122)+(183-124)
= (9+2+30+23+84+24+02+59)
= 233

**Advantages :-**

1) Average Response Time decreases

2) Throughput increases

**Disadvantages :-**

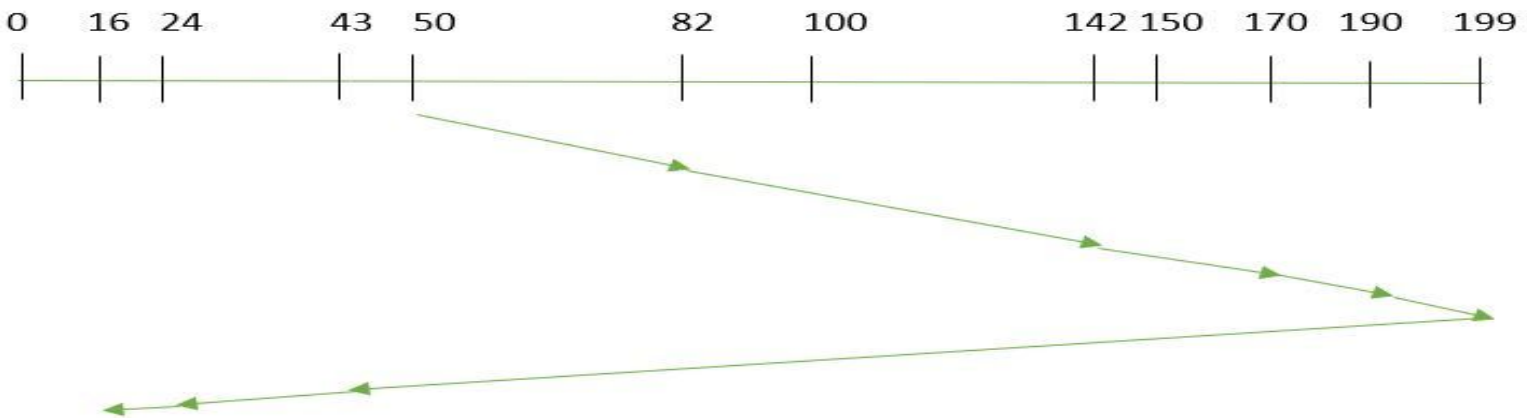
1) Overhead to calculate seek time in advance

2) Can cause Starvation for a request if it has higher seek time as compared to incoming requests.

3) High variance of response time as SSTF favours only some requests

**3) SCAN :-** In this 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 as 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.

**Example :-** Suppose the requests to be addressed are - 82,170,43,140,24,16,190. And the Read/Write arm is at 50, and it is also given that the disk arm should move **" towards the larger value". (It may start towards nearest end) It always hits upper or bottom value**




Seek Time = (199-50)+(199-16) = 332

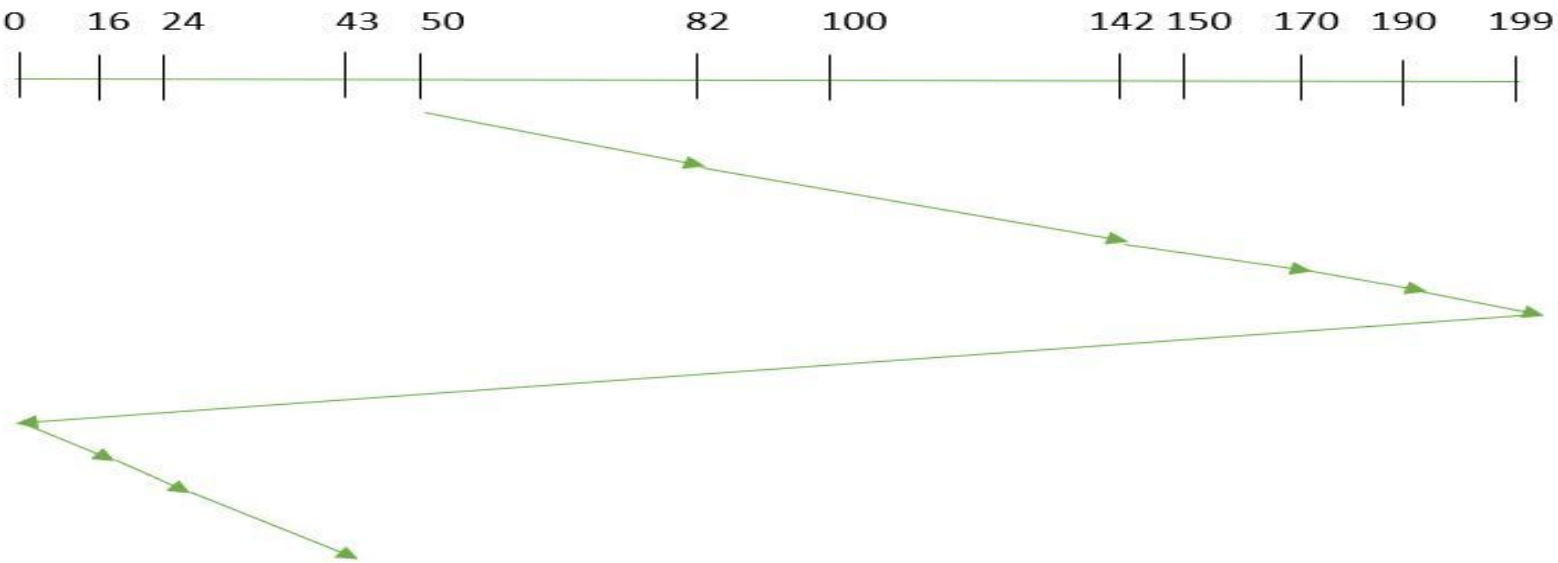
**Advantages :-** High throughput Low variance of response time Average response time

**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 *CSCAN* 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).

Suppose the requests to be addressed are - 82,170,43,140,24,16,190. And the Read/Write arm is at 50, and it is also given that the disk arm should move **"towards the larger value".**




Seek time is calculated as :- = (199-50)+(199-0)+(43-0) = 391

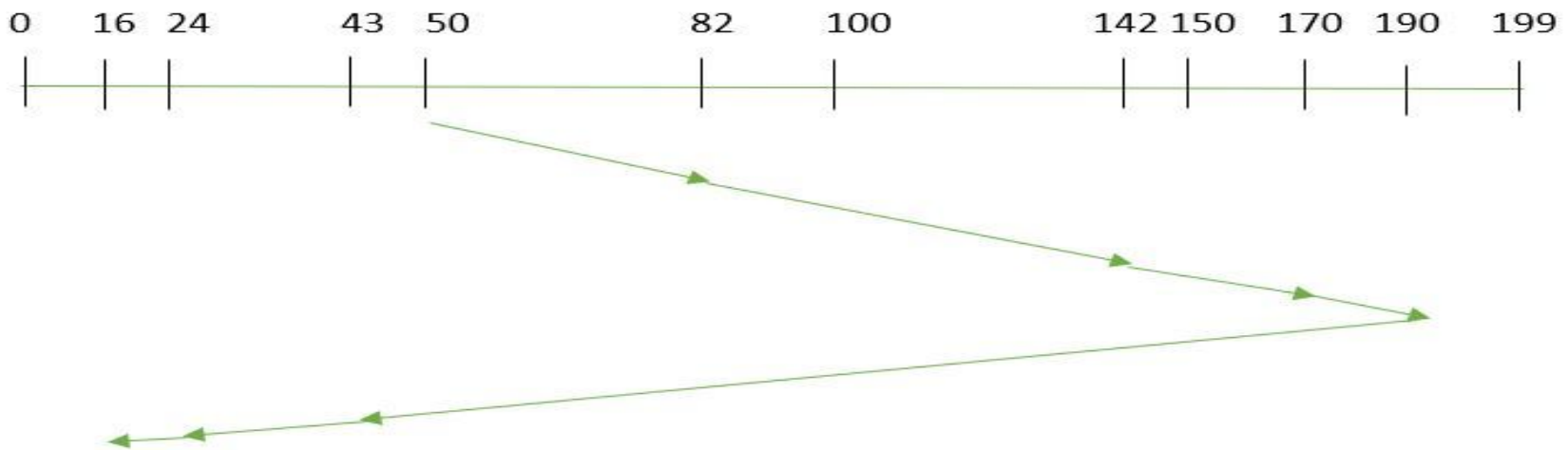
**Advantages :-**

Provides more uniform wait time compared to SCAN

**5) LOOK :-** It is similar to the SCAN disk scheduling algorithm except for 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.

**Example :-**

Suppose the requests to be addressed are - 82,170,43,140,24,16,190. And the Read/Write arm is at 50, and it is also given that the disk arm should move **"towards the larger value".**



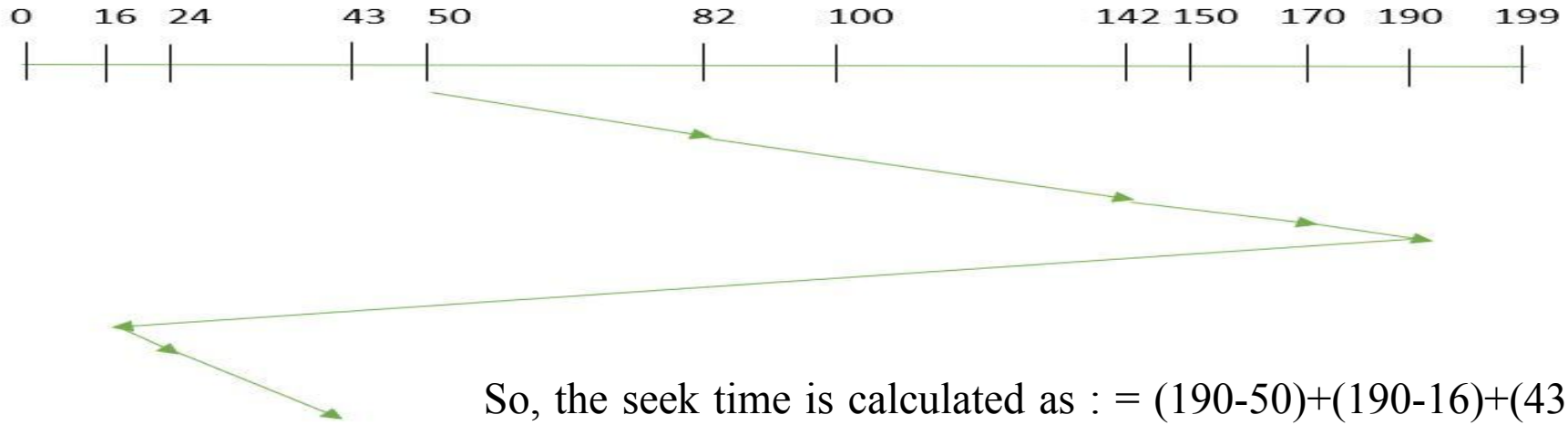

So, the seek time is calculated as = (190-50)+(190-16)

= 314

**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 in spite 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.

**Example :-**

Suppose the requests to be addressed are - 82,170,43,140,24,16,190. And the Read/Write arm is at 50, and it is also given that the disk arm should move **"towards the larger value"**




So, the seek time is calculated as : = (190-50)+(190-16)+(43-16)
= 341

# Process Management in Linux

A process means program in execution. There are basically 2 types of processes.

**1) Foreground processes :-** Such kind of processes are also known as **interactive processes**. These are the processes which are to be executed or initiated by the user or the programmer, they can not be initialized by system services. Such processes take input from the user and return the output. While these processes are running we can not directly initiate a new process from the same terminal.

**2) Background processes:** Such kind of processes are also known as **non interactive processes**. These are the processes that are to be executed or initiated by the system itself or by users, though they can even be managed by users. These processes have a unique PID or process if assigned to them and we can initiate other processes within the same terminal from which they are initiated.

**Foreground Process :-**

1) To stop a process in between of its execution , use **sleep command** . To force stop a foreground process in between of its execution - ***CTRL+Z***

**sleep NUMBER[SUFFIX]... (by default time in seconds)**

**suffix may be (s - seconds , m – minutes , h - hours , d - days)**

**2)** To get the list of jobs that are either running or stopped - **jobs command** is used .

It will display the stopped processes in this terminal and even the pending ones.

**JOB Job name or number.**

-l Lists process IDs in addition to the normal information.

-n List only processes that have changed status since the last notification.

-p Lists process IDs only.

-r Restrict output to running jobs.

-s Restrict output to stopped jobs.

a) To display the process ID or jobs for the job whose name begins with "p,"

$ jobs -p %p OR $ jobs %p

**O/p**

[4]- Stopped ping cybercity.iz

b) Only **-p option to jobs command used to display PIDs only**

$ jobs -p

**O/p**

7897
7905
7950
8046

c) Only **-r option to jobs command used to display only running jobs**

$ jobs –r

**O/p**

[1] Running gpass &
[2] Running gnome-calculator &
[3] -Running gedit fetch-stock-prices.py &

3) To run all the pending and force stopped jobs in the background **bg command** is used. This will start the stopped and pending processes in the background.

4) To get details of a process running in background.

ps -ef | grep name of process or pattern to search

ps - list processes

-e - show all processes, not just those belonging to the user

-f - show processes in full format (more detailed than default)

command 1 | command 2 - pass output of command 1 as input to command 2

grep find lines containing a pattern

5) To run processes with priority.

nice -n 5 **process name**

The top priority is -20 but as it may affect the system processes so we have used the priority 5.

**Background Processes**

A background process runs without being connected to your keyboard. If the background process requires any keyboard input, it waits.

The advantage of running a process in the background is that you can run other commands; you do not have to wait until it completes to start another!

The simplest way to start a background process is to add an ampersand (**&**) at the end of the command.

**There are five types of Process in Linux**

1. **Parent process :-** The process created by the user on the terminal. All processes have a parent process, If it was created directly by user then the parent process will be the kernel process.
2. **Child process :-** The process created by another process (by its parent process). All child processes have a parent process.
3. **Orphan process :-** Sometimes when the parent gets executed before its own child process then the child process becomes an orphan process. The orphan process have "Init" process (PID 0) as their PPID (parent process ID)
4. **Zombie process :-** The processes which are already dead but shows up in process status is called Zombie process. Zombie processes have Zero CPU consumption.

   When a process ends the execution, then it will have an exit status to report to its master process. Because of that little bit of information, the process will remain in the OS process table as a zombie process, which indicates that it is not to be scheduled for future, but this process cannot be completely removed or the process ID will not be used until the exit has been determined and no longer needed.

When a child completes the process, the master process will receive a SIGCHLD signal to indicate that one of its child process has finished the executing; the parent process will typically call the wait() system status at this point. That status will provide the parent with the child's process exit status, and will cause the child process to be removed from the process table.

**5. Daemon process :-** These are system-related processes that run in the background. A daemon is a program with a unique purpose. They are utility programs that run silently in the background to monitor and take care of certain subsystems to ensure that the operating system runs properly.

# Operations Performed on a Directory

- To understand the requirements for a file structure, it is helpful to consider the types of operations that may be performed on the directory:


# Two-Level Scheme

There is one directory for each user and a master directory

Master directory has an entry for each user directory providing address and access control information

Each user directory is a simple list of the files of that user

Names must be unique only within the collection of files of a single user

File system can easily enforce access restriction on directories

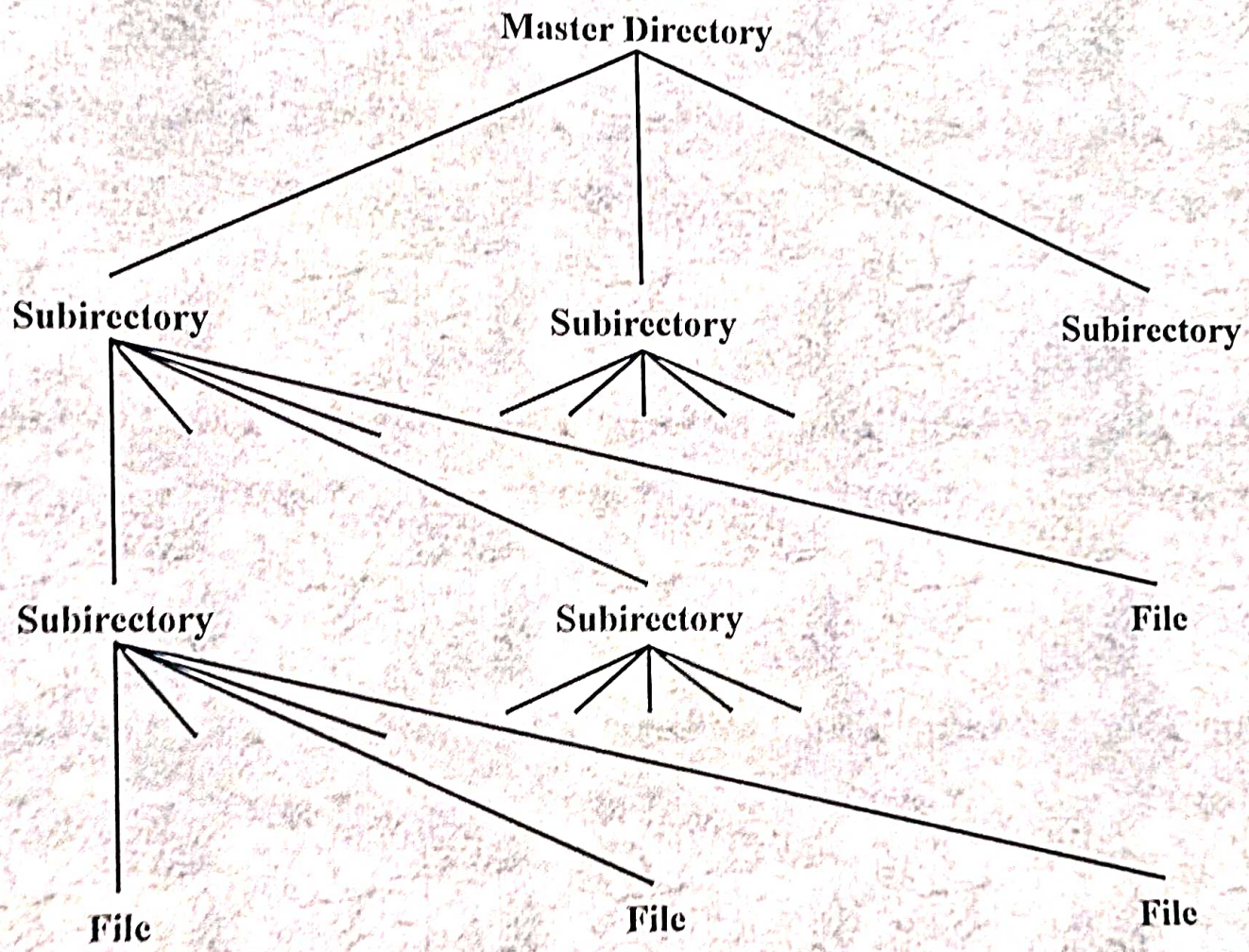



Figure 12.6 Tree-Structured Directory

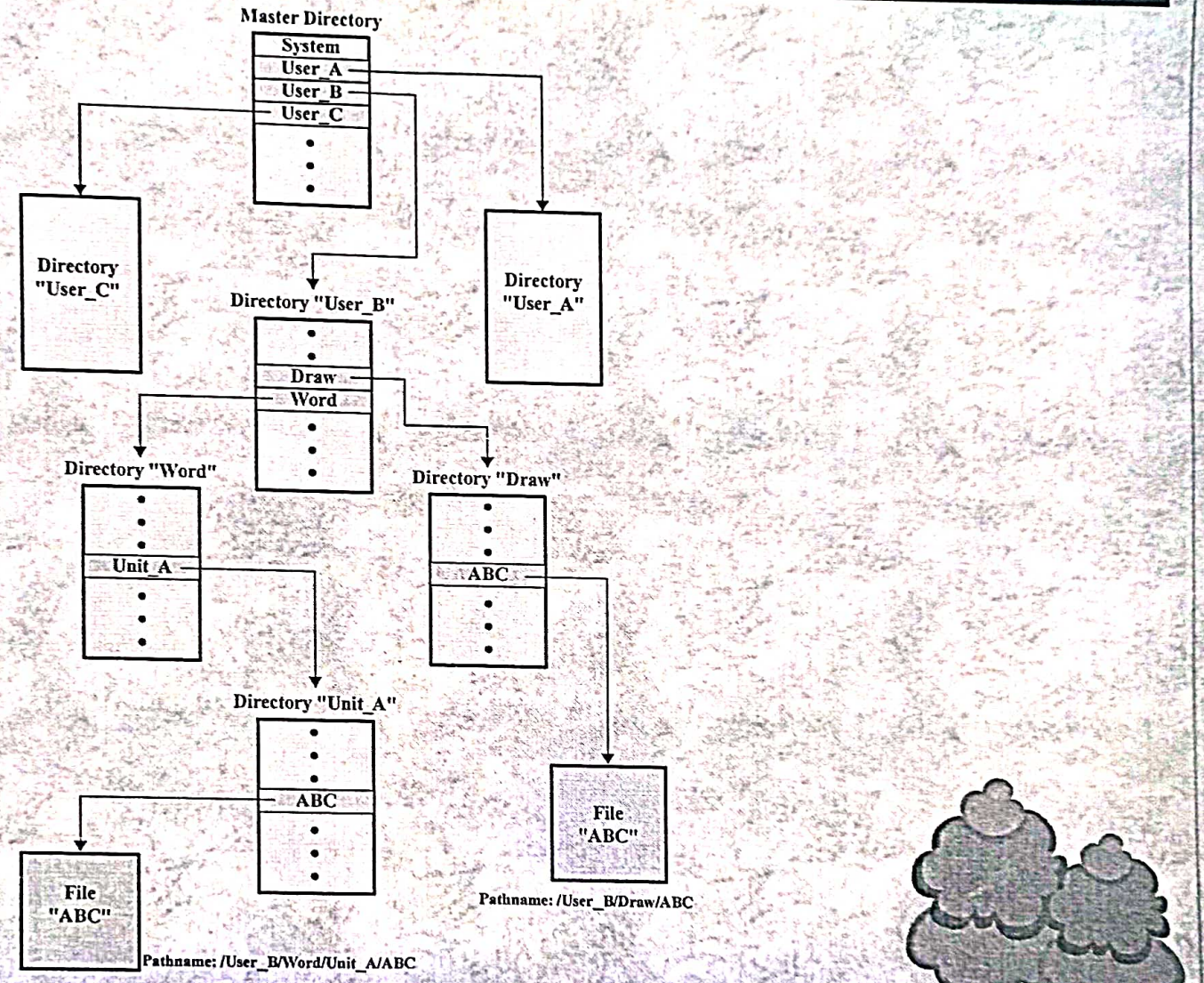
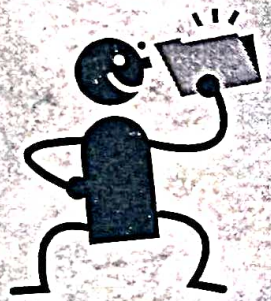



**Figure 12.7 Example of Tree-Structured Directory**

# File Sharing



Two issues arise when allowing files to be shared among a number of users:

- access rights
- management of simultaneous access

# Access Rights

- *None*
  - the user would not be allowed to read the user directory that includes the file
- *Knowledge*
  - the user can determine that the file exists and who its owner is and can then petition the owner for additional access rights
- *Execution*
  - the user can load and execute a program but cannot copy it
- *Reading*
  - the user can read the file for any purpose, including copying and execution
- *Appending*
  - the user can add data to the file but cannot modify or delete any of the file's contents
- *Updating*
  - the user can modify, delete, and add to the file's data
- *Changing protection*
  - the user can change the access rights granted to other users
- *Deletion*
  - the user can delete the file from the file system

# User Access Rights

## Owner

- usually the initial creator of the file
- has full rights
- may grant rights to others

## Specific Users

- individual users who are designated by user ID

## User Groups

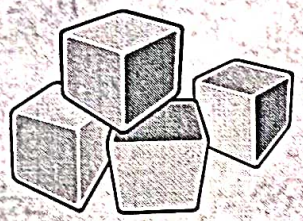
- a set of users who are not individually defined

## All

- all users who have access to this system
- these are public files

# Record Blocking

- Blocks are the unit of I/O with secondary storage
  - for I/O to be performed records must be organized as blocks



- Given the size of a block, three methods of blocking can be used:

1) **Fixed-Length Blocking** – fixed-length records are used, and an integral number of records are stored in a block

   ***Internal fragmentation*** – unused space at the end of each block

2) **Variable-Length Spanned Blocking** – variable-length records are used and are packed into blocks with no unused space

3) **Variable-Length Unspanned Blocking** – variable-length records are used, but spanning is not employed

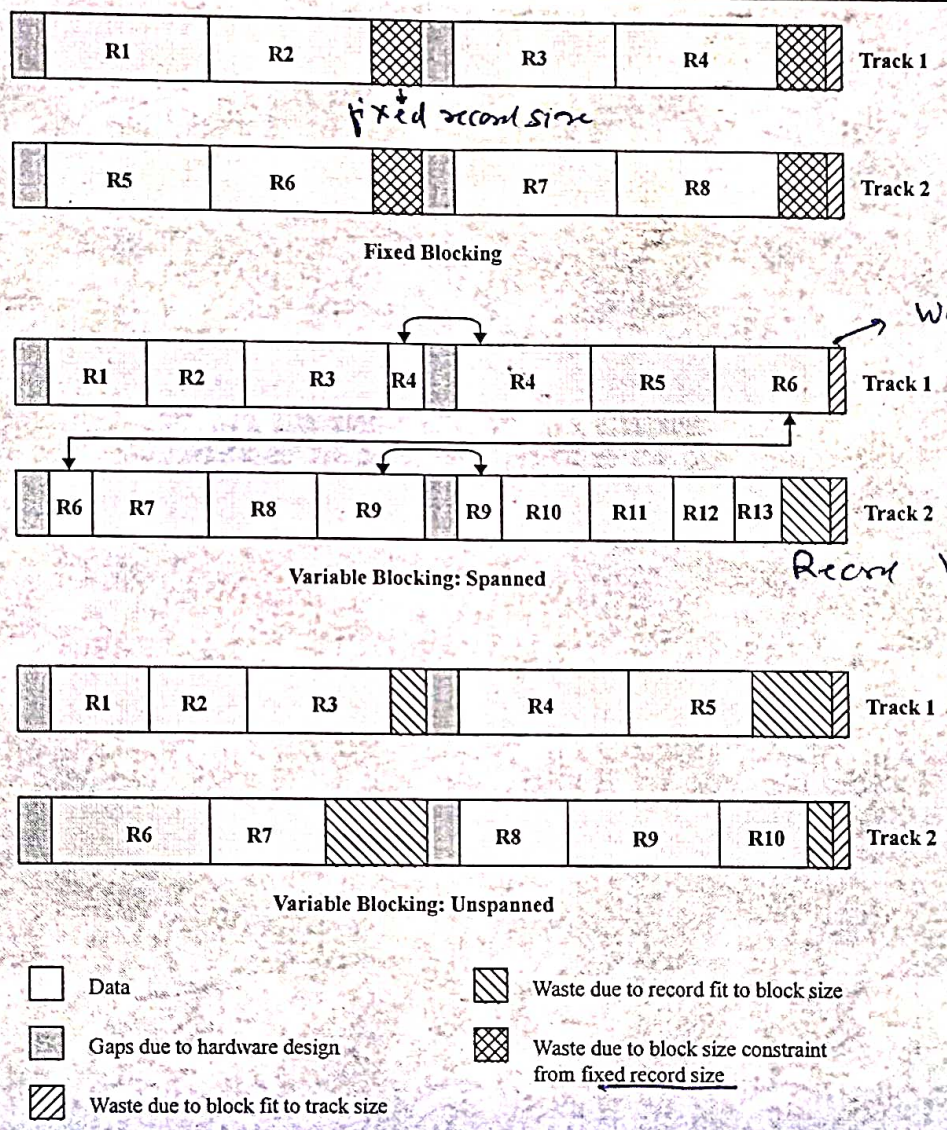
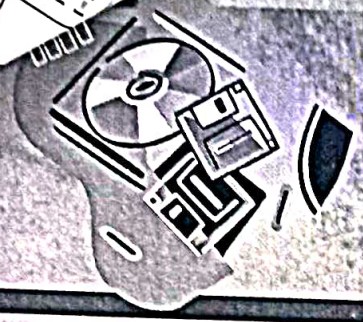



**Figure 12.8 Record Blocking Methods [WIED87]**

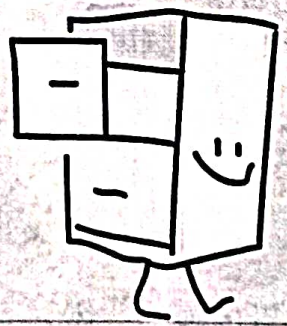


# File Allocation

- On secondary storage, a file consists of a collection of blocks
- The operating system or file management system is responsible for allocating blocks to files
- The approach taken for file allocation may influence the approach taken for free space management
- Space is allocated to a file as one or more ***portions*** (contiguous set of allocated blocks)
- ***File allocation table (FAT)***
  - data structure used to keep track of the portions assigned to a file

# Preallocation vs Dynamic Allocation

- A preallocation policy requires that the maximum size of a file be declared at the time of the file creation request
- For many applications it is difficult to estimate reliably the maximum potential size of the file
  - tends to be wasteful because users and application programmers tend to overestimate size
- Dynamic allocation allocates space to a file in portions as needed



# Portion Size

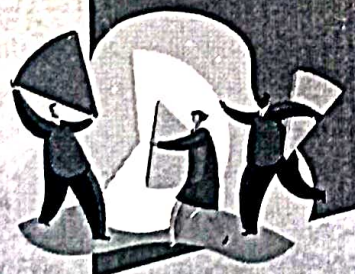
- In choosing a portion size there is a trade-off between efficiency from the point of view of a single file versus overall system efficiency
- Items to be considered:
  1) contiguity of space increases performance, especially for Retrieve_Next operations, and greatly for transactions running in a transaction-oriented operating system
  2) having a large number of small portions increases the size of tables needed to manage the allocation information
  3) having fixed-size portions simplifies the reallocation of space
  4) having variable-size or small fixed-size portions minimizes waste of unused storage due to overallocation

# Alternatives

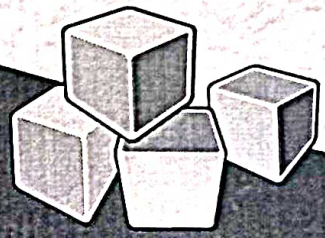
Two major alternatives:

## Variable, large contiguous portions

- provides better performance
- the variable size avoids waste
- the file allocation tables are small



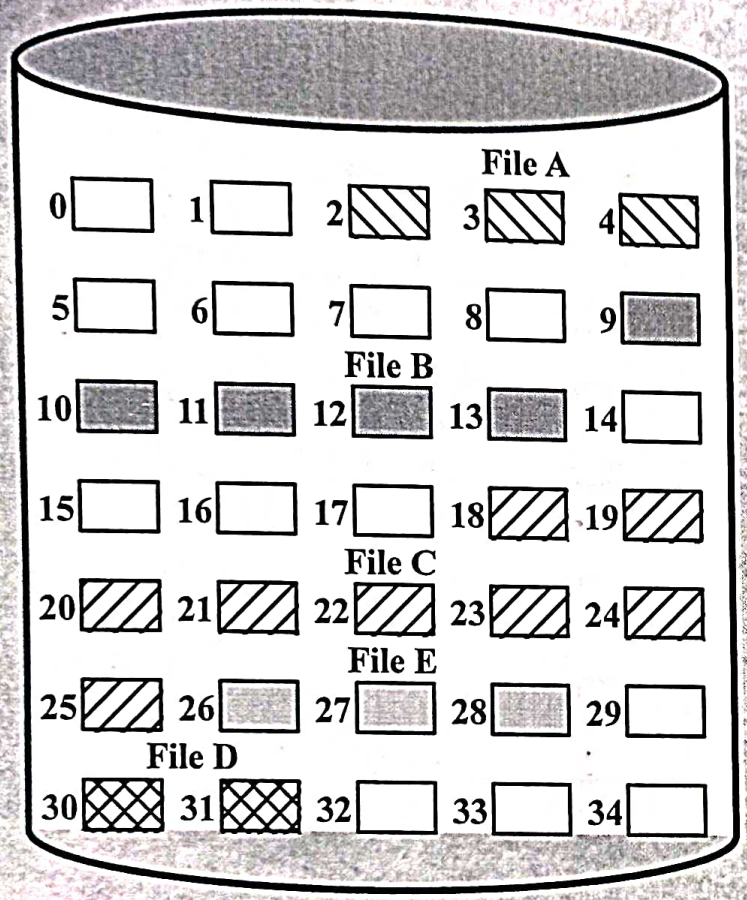
## Blocks



- small fixed portions provide greater flexibility
- they may require large tables or complex structures for their allocation
- contiguity has been abandoned as a primary goal
- blocks are allocated as needed

# Table 12.2
# File Allocation Methods

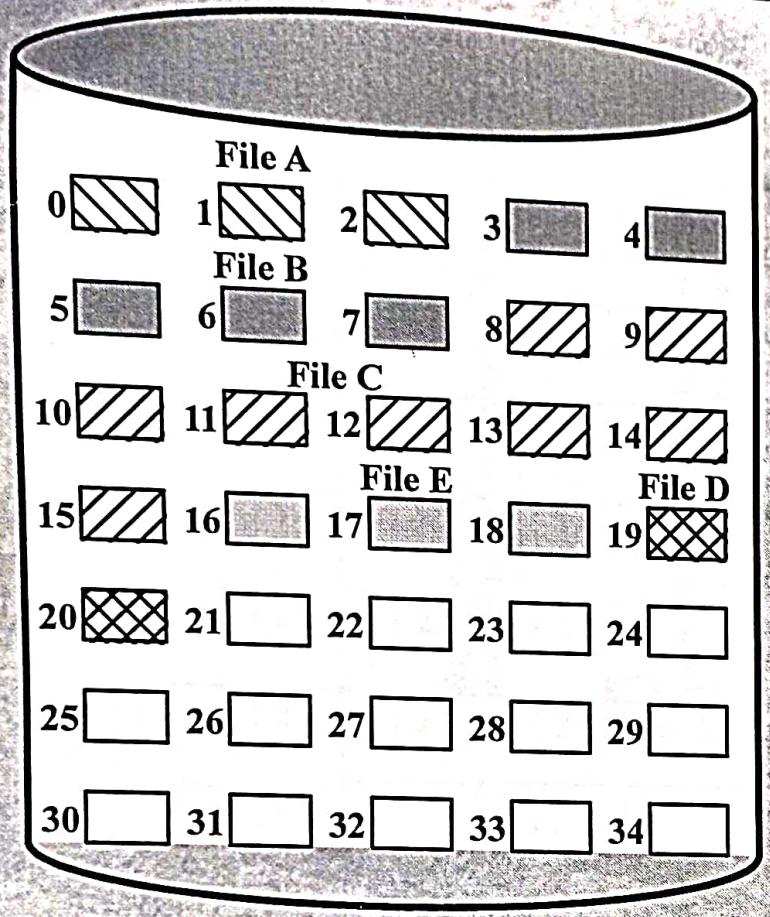
| | Contiguous | Chained | Indexed | |
|---|---|---|---|---|
| **Preallocation?** | Necessary | Possible | Possible | |
| **Fixed or variable size portions?** | Variable | Fixed blocks | Fixed blocks | Variable |
| **Portion size** | Large | Small | Small | Medium |
| **Allocation frequency** | Once | Low to high | High | Low |
| **Time to allocate** | Medium | Long | Short | Medium |
| **File allocation table size** | One entry | One entry | Large | Medium |




File Allocation Table

| File Name | Start Block | Length |
|---|---|---|
| File A | 2 | 3 |
| File B | 9 | 5 |
| File C | 18 | 8 |
| File D | 30 | 2 |
| File E | 26 | 3 |

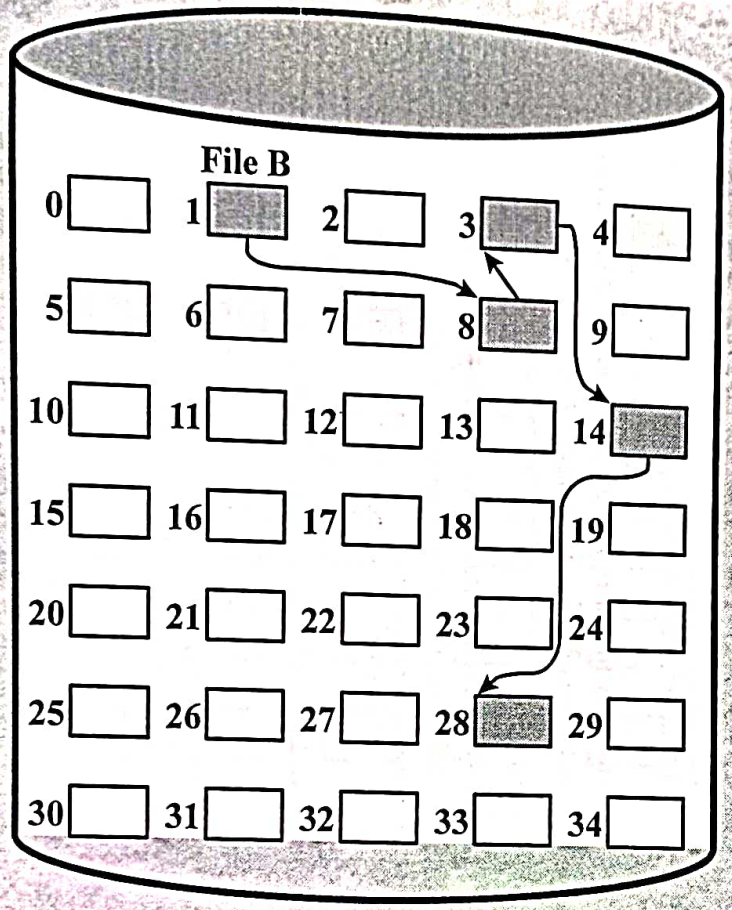
**Figure 12.9 Contiguous File Allocation**




File Allocation Table

| File Name | Start Block | Length |
|---|---|---|
| File A | 0 | 3 |
| File B | 3 | 5 |
| File C | 8 | 8 |
| File D | 19 | 2 |
| File E | 16 | 3 |

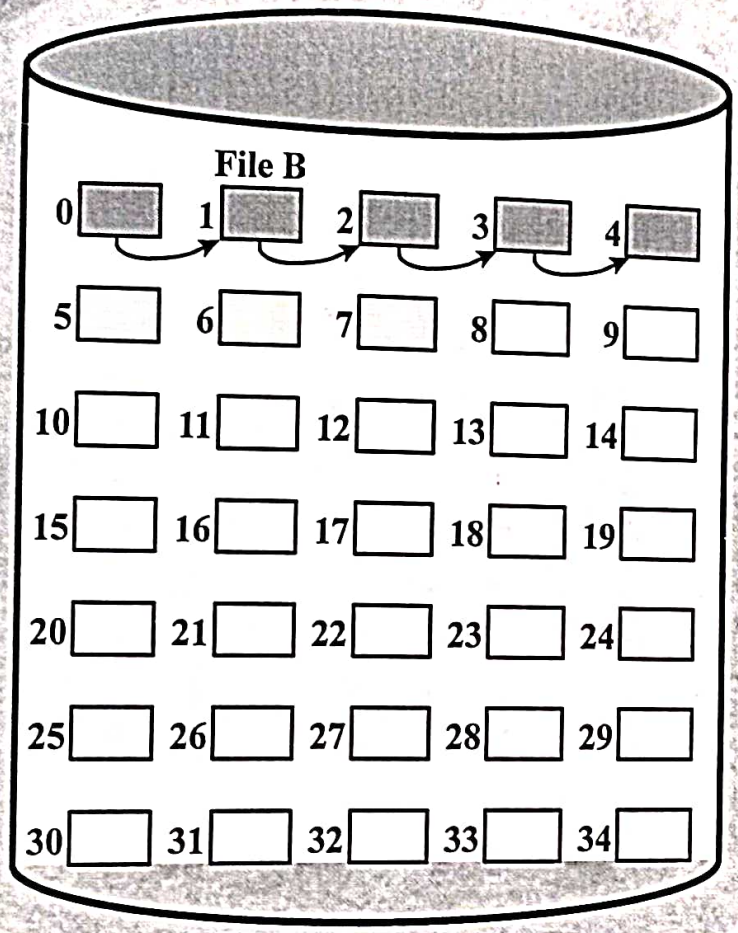
**Figure 12.10 Contiguous File Allocation (After Compaction)**




File Allocation Table

| File Name | Start Block | Length |
|---|---|---|
| • • • | • • • | • • • |
| File B | 1 | 5 |
| • • • | • • • | • • • |

**Figure 12.11 Chained Allocation**




File Allocation Table

| File Name | Start Block | Length |
|---|---|---|
| ... | ... | ... |
| File B | 0 | 5 |
| ... | ... | ... |

**Figure 12.12 Chained Allocation (After Consolidation)**

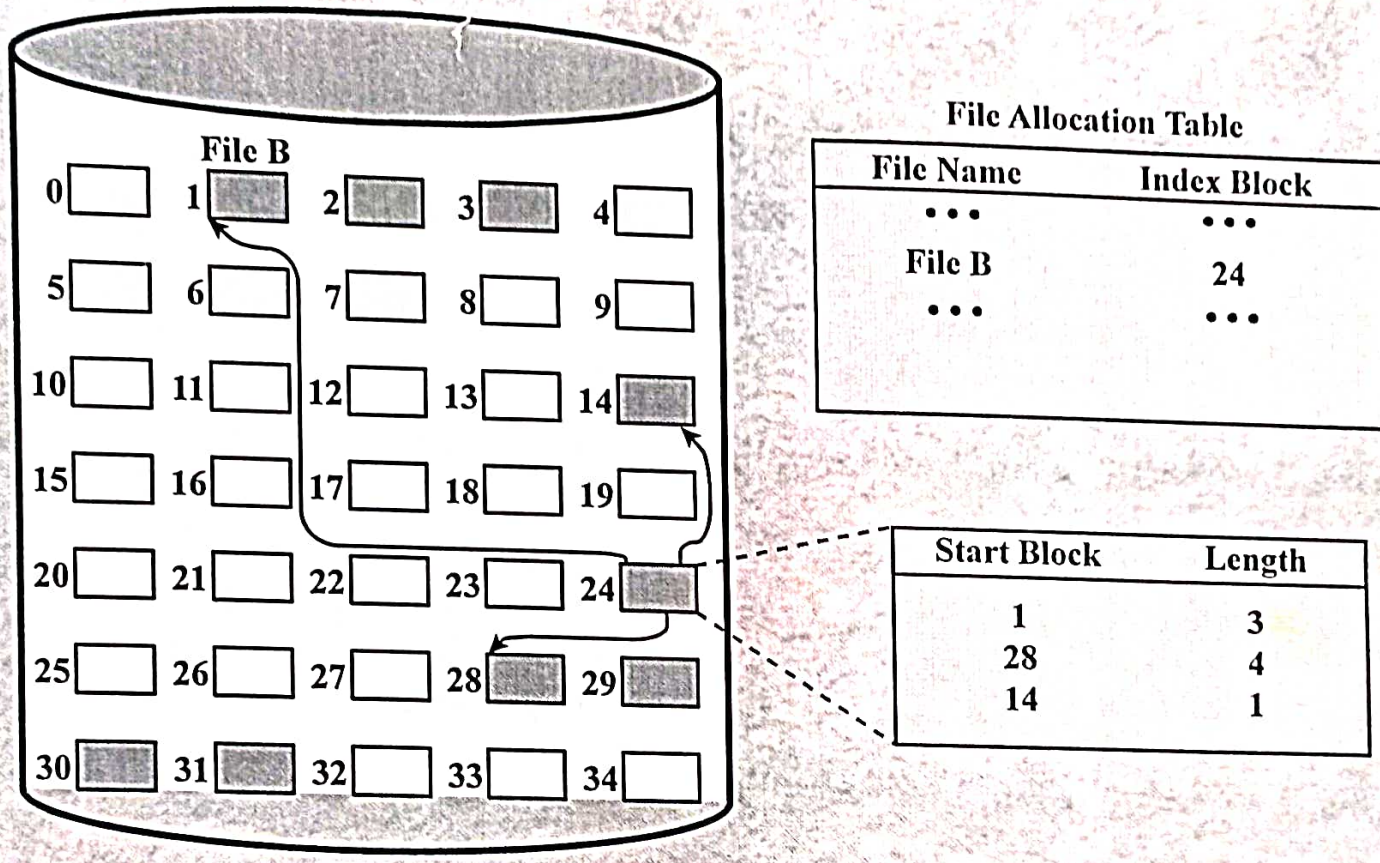



**Figure 12.14 Indexed Allocation with Variable-Length Portions**