SATHYABAMA
INSTITUTE OF SCIENCE AND TECHNOLOGY
(DEEMED TO BE UNIVERSITY)
Accredited "A" Grade by NAAC | 12B Status by UGC | Approved by AICTE
www.sathyabama.ac.in

SCHOOL OF COMPUTING

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

UNIT – I – Operating System – SBS1206

OPERATING SYSTEM:

UNIT I **(9 Hrs)**

Introduction: Operating system –Views and Goals - Types of System- OS Structure - Components - Services - System Structure - Layered Approach - Process Management: Process - Process Scheduling - Cooperating Process - Threads - Inter-process Communication.

UNIT 1

INTRODUCTION

INTRODUCTION TO THE OS :

- A computer system has many resources (hardware and software), which may be require to complete a task.
- Operating System is a system software that acts as an intermediary between a user and ComputerHardware to enable convenient usage of the system and efficient utilization of resources.
- The commonly required resources are input/output devices, memory, file storage space, CPU etc.
- The operating system acts as a manager of the above resources and allocates them to specific programs and users, whenever necessary to perform a particular task.
- Therefore operating system is the resource manager i.e. it can manage the resource of a computer system internally.
- The resources are processor, memory, files, and I/O devices. **In simple terms, an operating system is the interface between the user and the machine.**

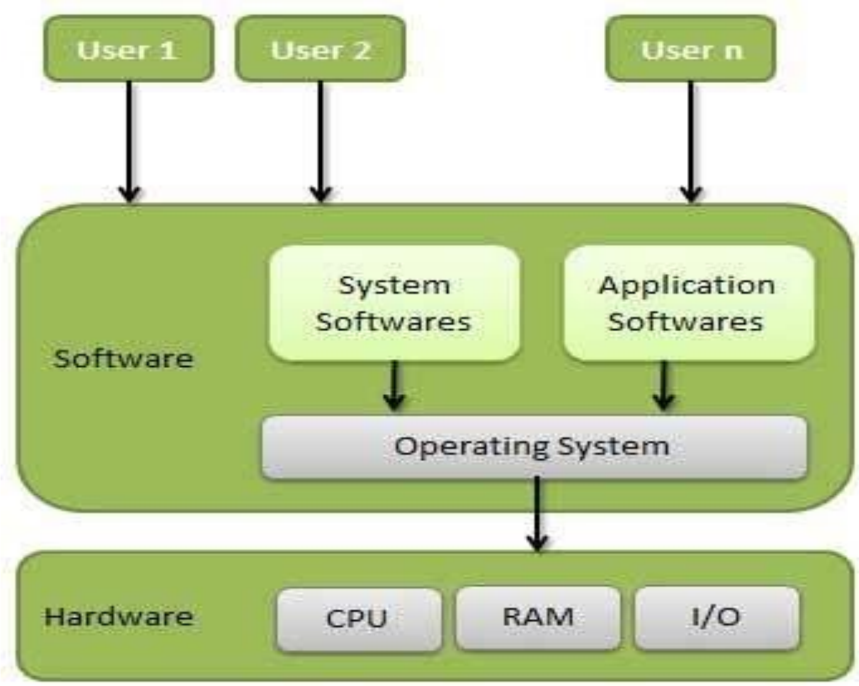


Fig:1.1 Architecture of an Operating System

- Operating system is the most important program that runs on a computer. OS is considered as thebackbone of a computer, managing both software and hardware resources.
- They are responsible foreverything from the control and allocation of memory to recognizing input from external devices andtransmitting output to computer displays.
- They also manage files on computer hard drives and control peripherals, like printers and scanners.
- Operating systems monitor different programs and users, making sure everything runs smoothly,without interference, despite the fact that numerous devices and programs are used simultaneously.
- An operating system also has a vital role to play in security. Its job includes preventing unauthorizedusers from accessing the computer system.

GOALS OF OPERATING SYSTEM :

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

FUNCTIONS OF OPERATING SYSTEM / SYSTEM COMPONENTS:

- Main Memory Management
- Processor Management
- Device Management
- File Management
- I/O System Management
- Secondary Management
- Networking
- Protection System
- Command Interpreter System

MEMORY MANAGEMENT:

- Memory management refers to management of Primary Memory or Main Memory. Main memory is a large array of words or bytes where each word or byte has its own address.
- Main memory provides a fast storage that can be accessed directly by the CPU.

- ACTIVITIES for memory management :
  - Keeps tracks of primary memory, i.e., what part of it are in use by whom, what part is not in use.
  - In multiprogramming, the OS decides which process will get memory when and how much.
  - Allocates the memory when a process requests it to do so.
  - De-allocates the memory when a process no longer needs it or has been terminated.

PROCESSOR MANAGEMENT:

- In multiprogramming environment, the OS decides which process gets the processor when and for how much time. This function is called process scheduling.
- ACTIVITIES of processor management
  - Keeps tracks of processor and status of process. The program responsible for this task is known as traffic controller.
  - Allocates the processor (CPU) to a process.
  - De-allocates processor when a process is no longer required.

DEVICE MANAGEMENT:

- An Operating System manages device communication via their respective drivers.
- ACTIVITIES of device management
  - Keeps tracks of all devices. Program responsible for this task is known as the I/O controller.
  - Decides which process gets the device when and for how much time.
  - Allocates the device in the efficient way.
  - De-allocates devices.

FILE MANAGEMENT:

- A file system is normally organized into directories for easy navigation and usage. These directories may contain files and other directions.
- ACTIVITIES of file management
  - Keeps track of information, location, uses, status etc. The collective facilities are often known as file system.
  - Decides who gets the resources.
  - Allocates the resources.
  - De-allocates the resources

I/O SYSTEM MANAGEMENT:

- OS hides the peculiarities of specific hardware devices from the user.
- It consists of
  - A memory management component that includes buffering, caching and spooling.
  - A general device-driver interface
  - Drivers for specific hardware devices.
  - Only the device driver knows the peculiarities of the specific device to which it is assigned.

SECONDARY STORAGE MANAGEMENT:

- The main purpose of a computer system is to execute programs. These programs, with the data they access ,must be in main memory, or primary storage.
- Systems have several levels of storage, including primary storage, secondary storage and cache storage.

- Since main memory (primary storage) is volatile and too small to accommodate all data and programs permanently, the computer system must provide secondary storage to back up main memory.
- Most modern computer systems use disks as the principle on-line storage medium, for both programs and data.
- ACTIVITIES are,
  - Free-space management (paging/swapping)
  - Storage allocation (what data goes where on the disk)
  - Disk scheduling (Scheduling the requests for memory access).

NETWORKING:

- A distributed systems are a collection of processors that do not share memory, peripheral devices, or a clock.
- The processors in a distributed system vary in size and function. They may include small processors,workstations, minicomputers and large, general-purpose computer systems.
- The processors in the system are connected through a communication-network ,which are configuredin a number of different ways i.e.., Communication takes place using a protocol.The network may be fully or partially connected .
- The communication-network design must consider routing and connection strategies, and theproblems of contention and security.
- A distributed system provides user access to various system resources.
- Access to a shared resource allows:
  - Computation Speed-up
  - Increased functionality
  - Increased data availability
  - Enhanced reliability

PROTECTION SYSTEM:

- If a computer system has multiple users and allows the concurrent execution of multiple processes, then the various processes must be protected from one another's activities.
- Protection refers to mechanism for controlling the access of programs, files, memory segments, processes(CPU) only by the users who have gained proper authorization from the OS.
- The protection mechanism must:

  - Distinguish between authorized and unauthorized usage.
  - Specify the controls to be imposed.
  - Provide a means of enforcement.

COMMAND INTERPRETER SYSTEM:

- A command interpreter is one of the important system programs for an OS. It is an interface of the operating system with the user. The user gives commands, which are executed by Operating system (usually by turning them into system calls).
- The main function of a command interpreter is to get and execute the next user specified command.
- Many commands are given to the operating system by control statements which deal with:
  - process creation and management
  - I/O handling
  - secondary-storage management
  - main-memory management
  - file-system access
  - protection
  - networking

CLASSIFICATION OF OPERATING SYSTEM:

- **Multi-user OS:**
  - Allows two or more users to run programs at the same time. This type of operating system may be used for just a few people or hundreds of them. In fact, there are some operating systems that permit hundreds or even thousands of concurrent users.
- **Multiprocessing OS:**
  - Support a program to run on more than one central processing unit (CPU) at a time. This can come in very handy in some work environments, at schools, and even for some home-computing situations.
- **Multitasking OS :**
  - Allows to run more than one program at a time.
- **Multithreading OS:**
  - Allows different parts of a single program to run concurrently (simultaneously or at the same time).

- **Real time OS:**
    - These are designed to allow computers to process and respond to input instantly. Usually, generalpurposeoperating systems, such as disk operating system (DOS), are not considered real time, as theymay require seconds or minutes to respond to input. Real-time operating systems are typically usedwhen computers must react to the consistent input of information without delay.
    - General-purpose operating systems, suchas DOS and UNIX, are not real-time. Today's operating systems tend to have graphical user interfaces(GUIs) that employ pointing devices for input. A mouse is an example of such a pointing device, as is astylus. Commonly used operating systems for IBM-compatible personal computers include MicrosoftWindows, Linux, and Unix variations. For Macintosh computers, Mac OS X, Linux, BSD, and someWindows variants are commonly used.

OPERATING SYSTEM STRUCTURES:

- An OS provides the environment within which programs are executed. Internally, Operating Systems vary greatly in their makeup, being organized along many different lines. The design of a new OS is a major task. The goals of the system must be well defined before the design begins. The type of system desired is the basis for choices among various algorithms and strategies. An OS may be viewed from several vantage ways.
    - By examining the services that it provides.
    - By looking at the interface that it makes available to users and programmers.
    - By disassembling the system into its components and their interconnections.

OPERATING SYSTEM SERVICES:

- An OS provides environment for the execution of programs. It provides certain services to programsand to the users of those programs. The specific services provided differs from one OS to another,but we can identify common classes.
- These OS services are provided for the convenience of the programmer, to make the programming task easier. One set of operating-system services provides functions that are helpful to the userare,

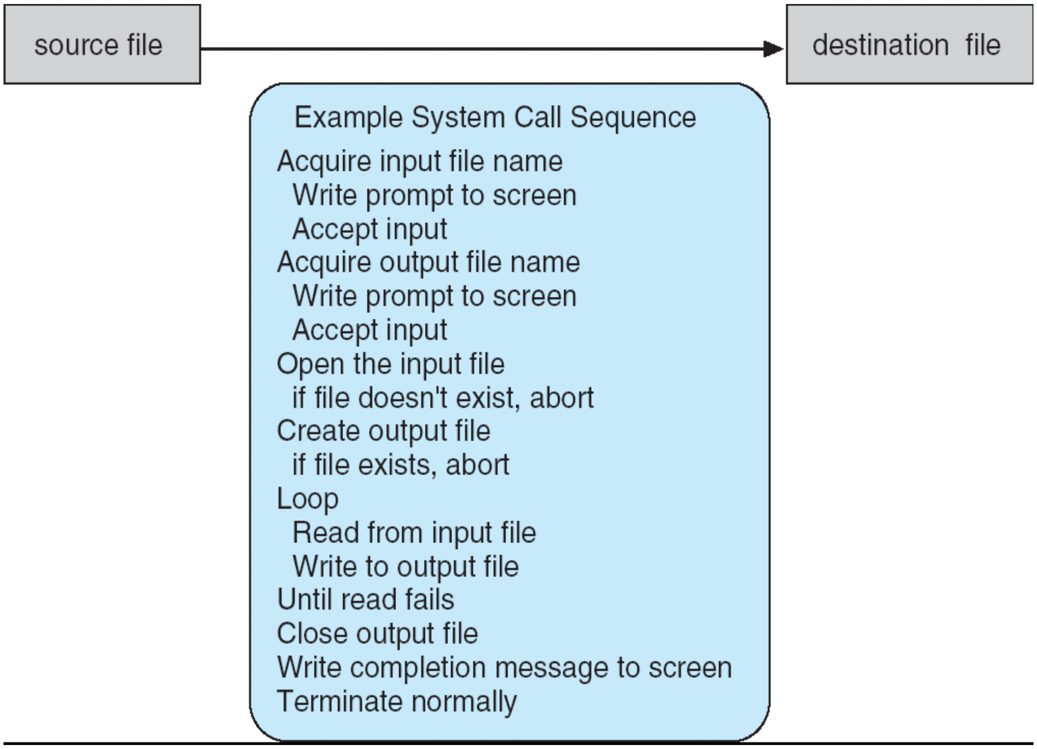
- **Program execution**
  - The system must be able to load a program into memory and to run that program.
  - The program must be able to end its execution, either normally or abnormally (indicating error).
- **I/O operations**
  - A running program may require I/O, which may involve a file or an I/O device.
  - For specific devices, special functions may be desired (rewind a tape drive, or to blank a CRT).
  - For efficiency and protection, users usually cannot execute I/O operations directly.Therefore Operating system must provide some means to perform I/O.
- **File-system manipulation**
  - The file system is of particular interest.
  - Obviously, programs need to read and write files and directories, create and delete them, searchthem, list file Information, permission management.
- **Communications**
  - One process needs to exchange information with another process. Such communication can occurin two ways:
  - The first takes place between processes that are executing on the same computer.
  - The second takes place between processes that are executing on different computers over anetwork.
  - Communications may be implemented via shared memory or through message passing, in whichpackets of information moved between the processes by the OS.
- **Error detection**
  - OS needs to be constantly aware of possible errors. Errors may occur
  - In the CPU and memory hardware (such as a memory error or power failure)
  - In I/O devices (such as a parity error on tape, a connection failure on a network or lack ofpower in the printer).
  - And in user program (such as arithmetic overflow, an attempt to access illegal memorylocation, or a too-great use of CPU time).

        - For each type of error, OS should take the appropriate action to ensure correct andconsistent computing.
        - Debugging facilities can greatly enhance the user's and programmer's abilities to efficientlyuse the system
    - **Resource allocation**
        - When multiple users logged on the system or multiple jobs running at the same time, resourcesmust be allocated to each of them.
        - Many types of resources are managed by OS. Some (such as CPU cycles, main memory, and filestorage) may have special allocation code, whereas others (such as I/O devices) may have generalrequest and release code.
    - **Protection and security**
        - The owners of information stored in a multi-user or networked computer system may want to control the use of that information.
        - Protection involves ensuring that all access to system resources is controlled. Security of the system from outsiders requires user authentication, extends to defending external I/O devices from invalid access attempts.
        - If a system is to be protected and secure, precautions must be instituted throughout. A chain is only as strong as its weakest link.

SYSTEM CALLS:

- System calls provide the interface between a process and the operating system.
- These calls are generally available as assembly-language instructions.
- Some systems also allow to make system calls from a high level language, such as C, C++and Perl (have been defined to replace assembly language for systems programming). Asan example of how system calls are used,consider writing a simple program to read datafrom one file and to copy them to another file.
- The first input that the program will need is the names of the two files:
    - The input file
    - The output file
- Once the two file names are obtained, the program must open the input file and create theoutput file.
- Each of these operations requires another system call and may encounter possible errorconditions.

  - When the program tries to open the file, it may find that no file of that name exists orthat the file is protected against access.
- If the input file exists, then we must create a new output file.
- We may find an output file with the same name.
  - This situation may cause the program to abort (a system call), or
  - We may delete the existing file (another system call).
- In an interactive system another option is to ask the user (a sequence of systemcalls to output the prompting message and to read response from the keyboard)whether to replace the existing file or to abort the program.



```
Acquire input file name
  Write prompt to screen
  Accept input
Acquire output file name
  Write prompt to screen
  Accept input
Open the input file
  if file doesn't exist, abort
Create output file
  if file exists, abort
Loop
  Read from input file
  Write to output file
Until read fails
Close output file
Write completion message to screen
Terminate normally
```

Fig:1.2 Example of System Calls

- Now that both the files are setup, we enter a loop that reads from the input file (a system call)and writes to the output file (another system call).
- Each read and write must return status information regarding various possible error conditions.
  - **On input,**
    - the program may find that the end of file has been reached, or
    - that a hardware failure occurred in the read (such as a parity error).
  - **On output,**
    - Various errors may occur, depending on the output device (such as no moredisk space, physical end of tape, printer out of paper).

- Finally, after the entire file is copied, The program may close both files (another system call), writes a message to theconsole(more system calls), and finally terminates normal (the final system call).
- System calls occur in different ways, depending on the computer in use.
- Three general methods are used to pass parameters between a running program and the operating system.
  - Simplest approach is to pass parameters in registers.
  - Store the parameters in a table in memory, and the table address is passed as aparameter in a register (in the cases where parameters are more than registers).
  - Push (store) the parameters onto the stack by the program, and pop off the stack byoperating system.

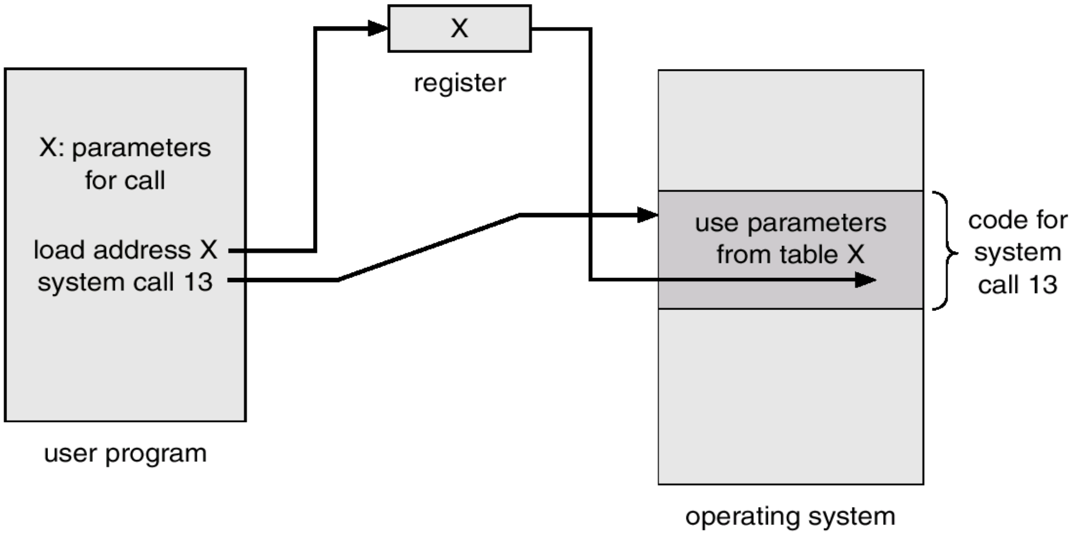
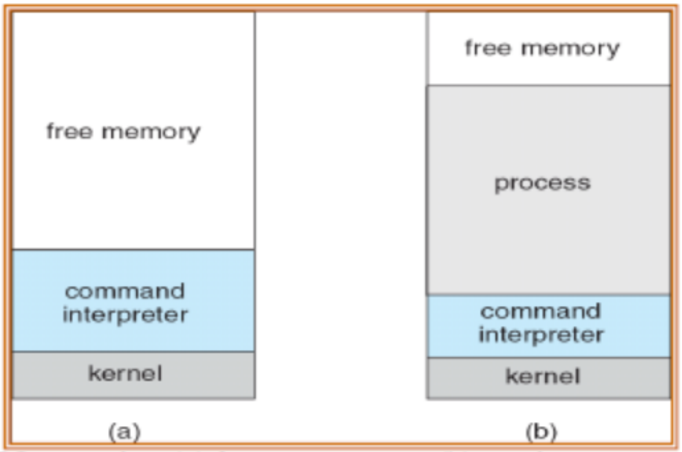


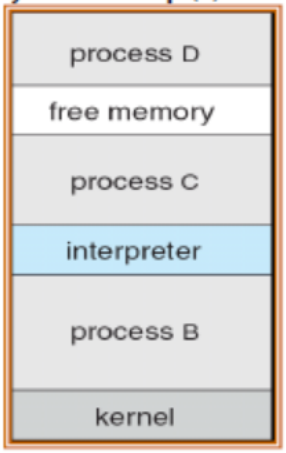
Fig:1.3 Parameter Passing via Table

TYPES OF SYSTEM CALLS

- System calls can be grouped roughly in to five categories:
  - **Process control:**load, execute, abort, end, create process, terminate process, get process attributes, set process attributes, allocate and free memory, wait event, signal event.



MS-DOS execution. (a) **At system startup** (b) **running a program**



UNIX Running Multiple Programs

- **File management :**create file, delete file, open, close, read, write,reposition, get file attribute,set fileattributes.
- **Device management :**request device, release device, read, reposition,write,get device attributes,setdevice attributes, logically attach or detach device.
- **Information maintenance :**get time and date, set time and date, get system data, set system data, get process file or device attributes, set process file or device attributes.
- **Communications :**create, close communication connection, send, receive messages, transfer status information, attach or detach remote devices. Communication may take place using:
  - message passing model or
  - shared memory model

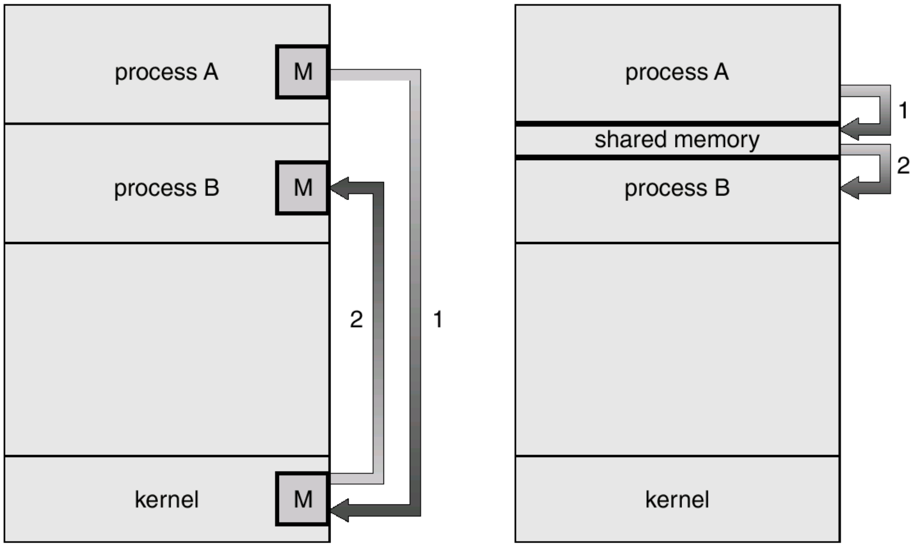


Fig:1.4 Communication models: (a) message passing model (b) shared memory model

SYSTEM STRUCTURE:

- A system as large and complex as a modern operating system must beengineered carefully if it is to function properly and be modified easily.
- A common approach is to partition the task into small components rather than have one monolithic system.
- There are four different structures that have shown in this document in order to get some idea of the spectrum of possibilities.
- These are by no means exhaustive, but they give an idea of somedesigns that have been tried in practice.

SIMPLE STRUCTURE:

- Many commercial systems do not have well-defined structures. Frequently,such operating systemsstarted as small, simple, and limited systems and then grew beyond their original scope.
- MS-DOS is an example of such a system. It was originally designed and implemented by a few people who had no idea that it would become so popular.

MS-DOS:

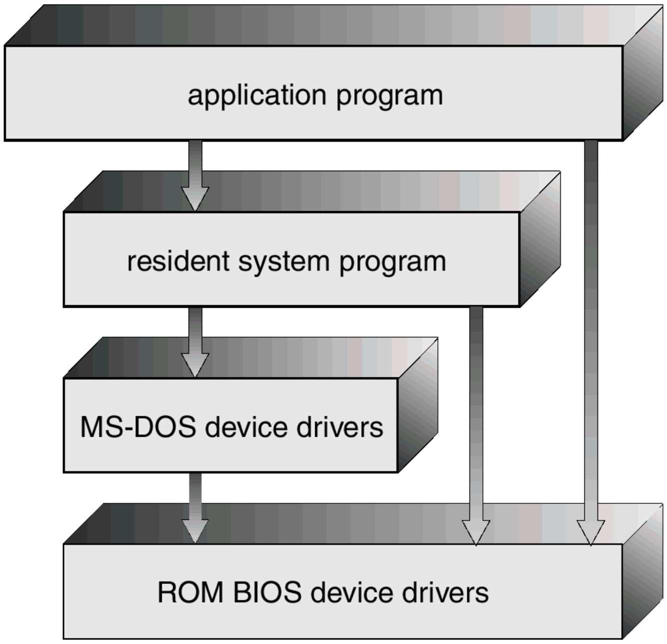


Fig:1.5 MS DOS LAYER STRUCTURE

- It was written to provide the most functionality in the least space(because of the limitedhardware on which it ran)
- So it was not divided into modules carefully.
- MS-DOS has some structure, its interfaces and levels of functionality arenot well separated

UNIX:

- UNIX is the another system limited by hardware functionality. It consists of two separable parts
    - System programs
    - Kernel

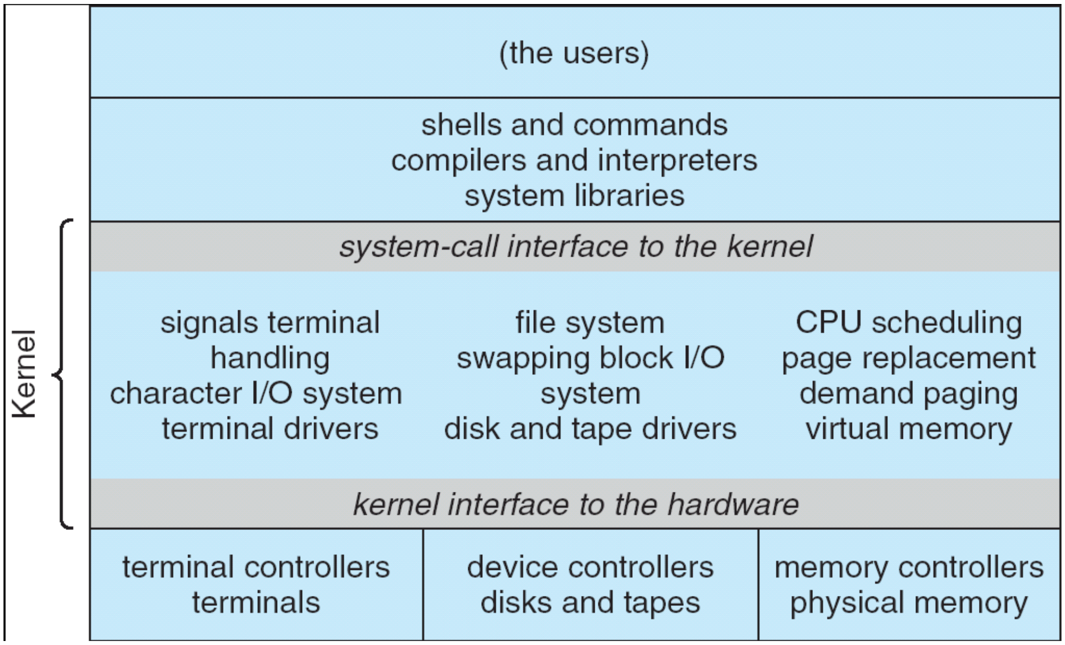


Fig:1.6 UNIX System Structures

The Kernel :

- The kernel is further separated into a series of interfaces and devicedrivers, which have been added and expanded over the years as UNIX has evolved.
- Everything below the system call interface and above the physicalhardware is the kernel.
- The kernel provides the file system, CPU scheduling,memorymanagement, and other operating-system functions through systemcalls.
- Taken in sum, that is an enormous amount of functionality to becombined into one level
- New versions of UNIX are designed to use more advanced hardware.
- With proper hardware support, operating systems can be broken into pieces that are smallerand more appropriate than those allowed by the original MS-DOS or UNIX systems.
- The operating system can then retain much greater control over the computer and over theapplications that make use of that computer.
- Implementers have more freedom in changing the inner workings of the system and in creatingmodular operating systems.

LAYERED APPROACH:

- A system can be made modular in many ways. One method is the layered approach, in which the operating system is broken up into a number of layers (levels), each built on top of lower layers.
- The bottom layer (layer 0) is the hardware; the highest (layer N) is the user interface as shownin below. With modularity, layers are selected such that each uses functions (operations) and services of only lower-level layers.

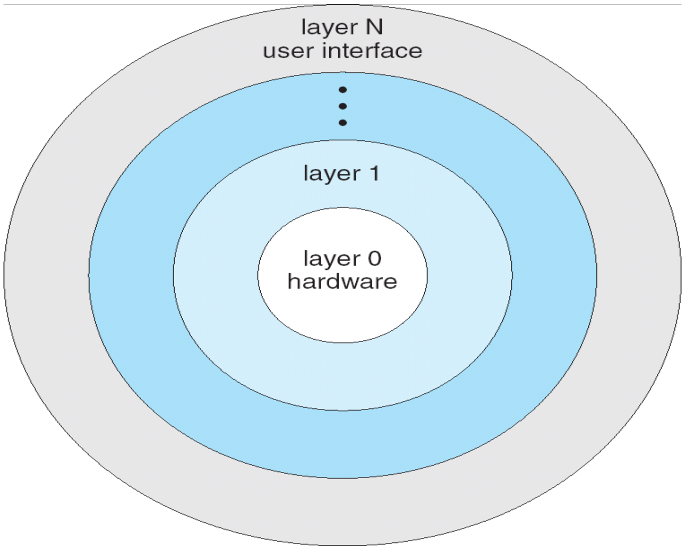


Fig:1.7 A Layered Operating System

- A layered design was first used in the operating system. Its six layers are as follows:

LAYER 5	User Programs
LAYER 4	Buffering for Input and Output
LAYER 3	Operator Console Device Driver
LAYER 2	Memory Management
LAYER 1	CPU Scheduling
LAYER 0	Hardware

- An operating-system layer is an implementation of an abstract object made up of data, and ofthe operations that can manipulate those data.

- A typical operating-system layer—say, layer M consists of data structures and a set of routinesthat can be invoked by higher-level layers. Layer M, in turn, can invoke operations on lowerlevellayers.

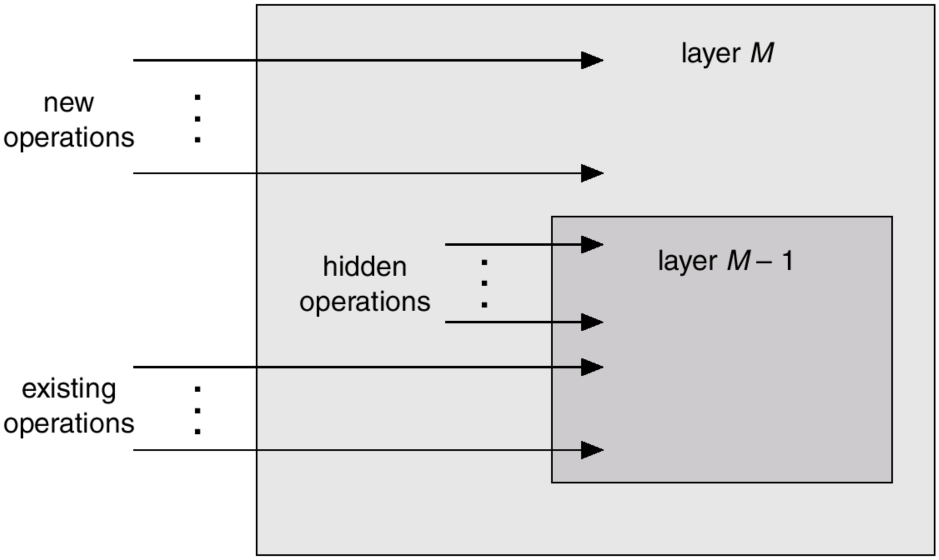


Fig:1.8 A typical Operating System Layer

- **ADVANTAGE :**
  - The main advantage of the layered approach is modularity (simplicity of construction and debugging). The layers are selected so that each uses functions (operations) and services of only lower-level layers.
- **DISADVANTAGE :**
  - The major difficulty with the layered approach involves appropriately defining the various layers. Because a layer can use only lower-level layers, careful planning is necessary.
  - A final problem with layered implementations is that they tend to be less efficient than othertypes.
  - Fewer layers with more functionality are being designed, providing most of the advantages ofmodularized code while avoiding the difficult problems of laver definition and interaction.

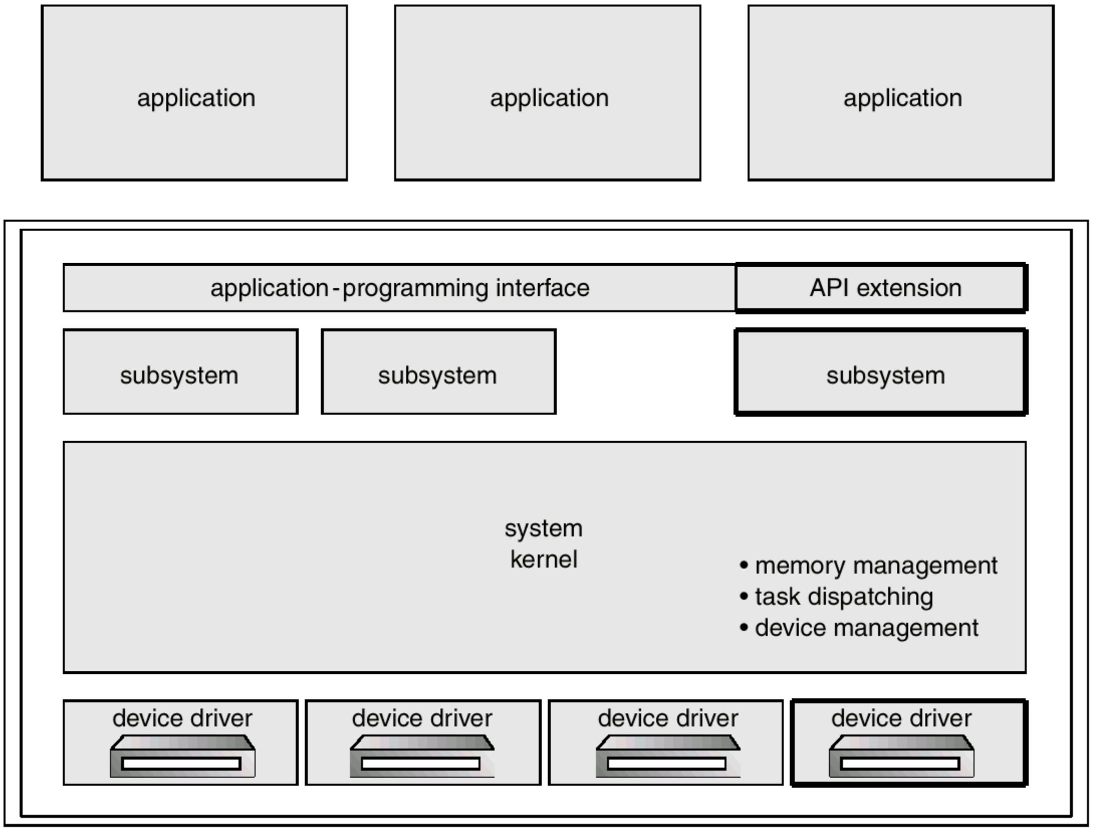


Fig:1.9 OS/2 Layer Structure

- OS/2 layer structure shown in above figure is a descendent of MS-DOS that adds multitasking and dualmodeoperation, as well as other new features.

KERNELS:

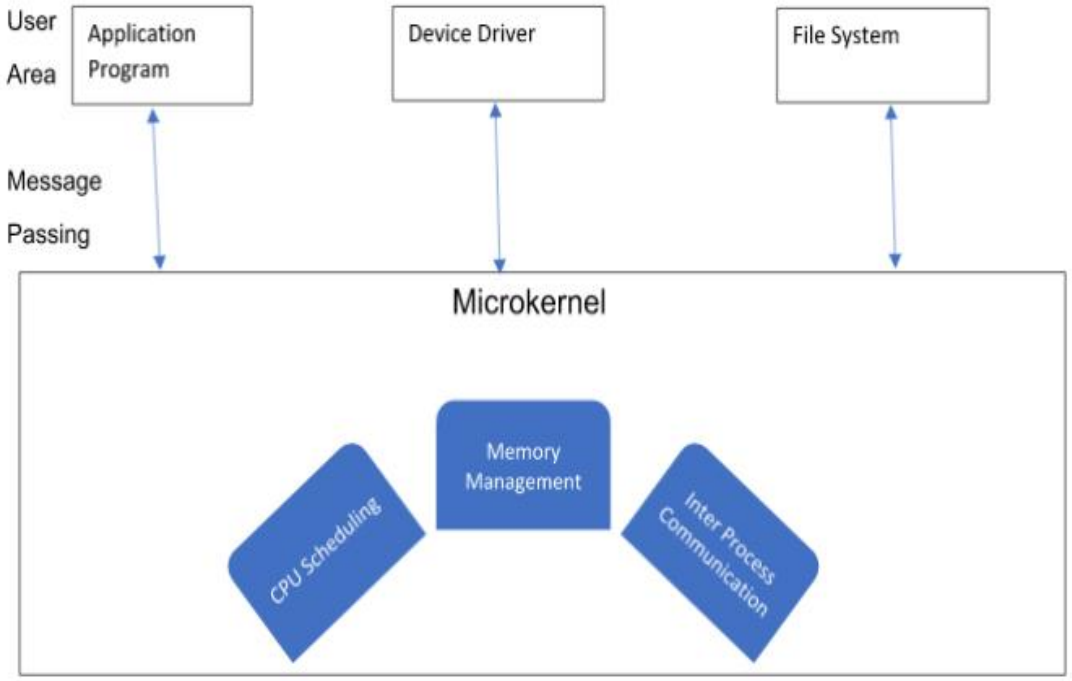


Fig:1.10 Kernels

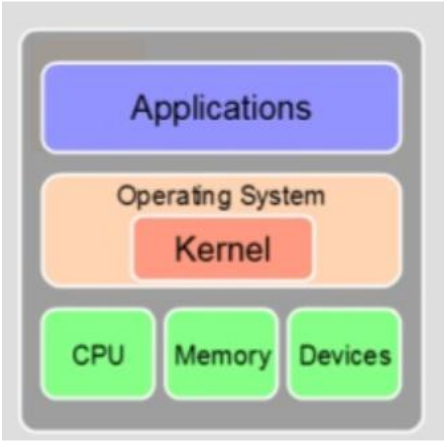


Fig: 1.11 Overview of a Kernel

- The fundamental part of an Operating system
- Responsible for providing secure access to the machine's hardware for various programs
- Responsible for deciding when and how long a program can use a certain hardware.
- **TYPES:**
  - Monolithic kernel
  - Micro kernel
  - Hybrid kernels
  - Nano kernels
  - Exo kernels

MICRO KERNELS:

- This method structures the operating system by removing all nonessential components from the kernel and implementing them as system and user-level programs i.e., moves as much from the kernel into "user" space which results is a smaller kernel.
- Microkernels typically provide minimal process and memory management, in addition to a communication facility.
- The main function of themicrokernel is to provide a communication facility between the client program and the variousservices that are also running in user space.
- Communication takes place between user modules using message passing. They communicate indirectly by exchangingmessages with the microkernel.
- One benefit of the microkernel approach is ease of extending the operating system. All newservices are added to user space and consequently do not require modification of the kernel.
- The microkernel also provides more security and reliability, since most services are running asuser—rather than kernel—processes.
- This micro kernel is a smaller kernel, which allows a only fewer changes in the operating system.
- **BENEFITS :**
  - Easier to extend a microkernel
  - Easier to port the operating system to new architectures
  - More reliable (less code is running in kernel mode)
  - More secure
- **DRAWBACKS:**
  - Microkernels can suffer from performance decreases due to increased system function overhead.

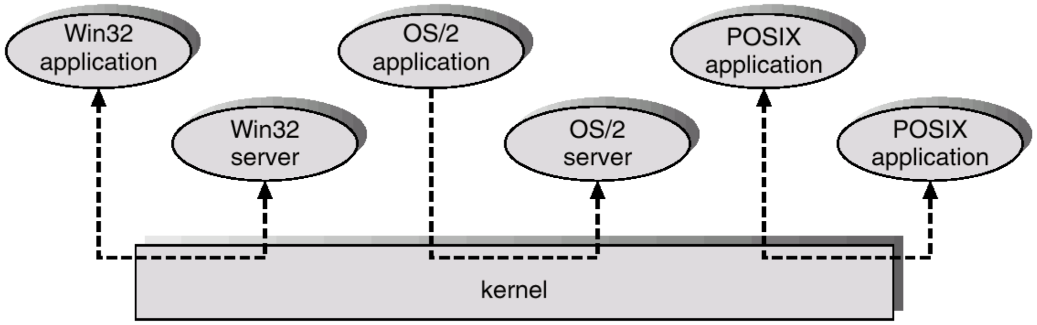


Fig:1.12 Micro Kernels

RESOURCES:

- The OS treats an entity as a resource if it satisfies the below characteristics:
    - A process must request it from the OS.
    - A process must suspend its operation until the entity is allocated to it.
- The most common source is a file. A process must request a file before it can read it orwrite it.
- **FILES:**
    - A sequential file is a named,linear stream bytes of memory.
    - You can store information by opening a file

PROCESS:

- A program in execution; process execution must progress in sequential fashion.
- A process is a program execution in OS
- A process is more than the program code, which is sometimes known as thetext section.
- A process includes:
    - **Program counter --** The current activity, as represented by the value of the Program counter and the contents of the processor's registers.
    - **Stack --** The process Stack contains temporary data (such as function parameters, return addresses, and local variables)
    - **Data section --** Data section, which contains global variables.
- An operating system executes a variety of programs:
    - Batch system – jobs
    - Time-shared systems – user programs or tasks
- A process may also include a heap, which is memory that is dynamically allocated duringprocess run time

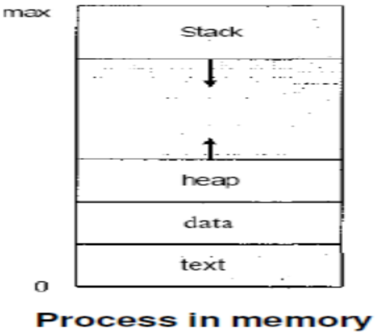
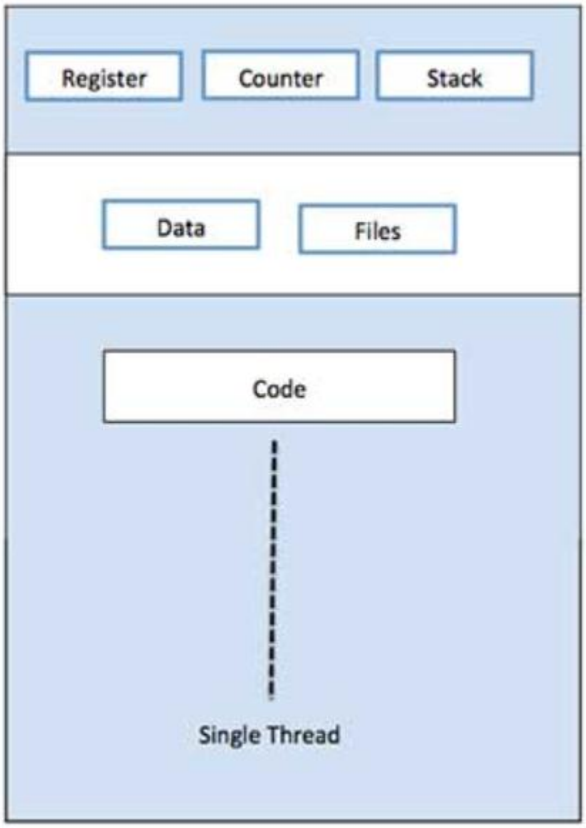


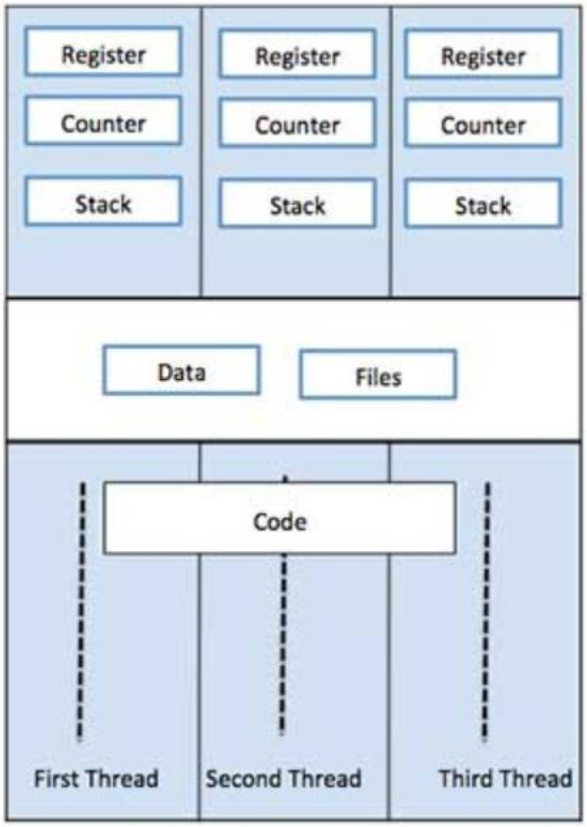
Fig:1.13 Process in Memory

THREADS:

- A thread is a flow of execution through the process code, with its own program counter that keeps track of which instruction to execute next, system registers which hold its current working variables, and a stack which contains the execution history.
- A thread shares with its peer threads few information like code segment, data segment and open files.
- A thread is a basic unit of CPU utilization. A thread,sometimes called as **light weight process** whereasa process is a **heavyweight process**.
- Thread comprises:
    - A thread ID
    - A program counter
    - A register set
    - A stack.
- A process is a program that performs a single thread of execution i.e., a process is a executingprogram with a single thread of control.
- This single thread of control allows the process to perform only one task at one time.



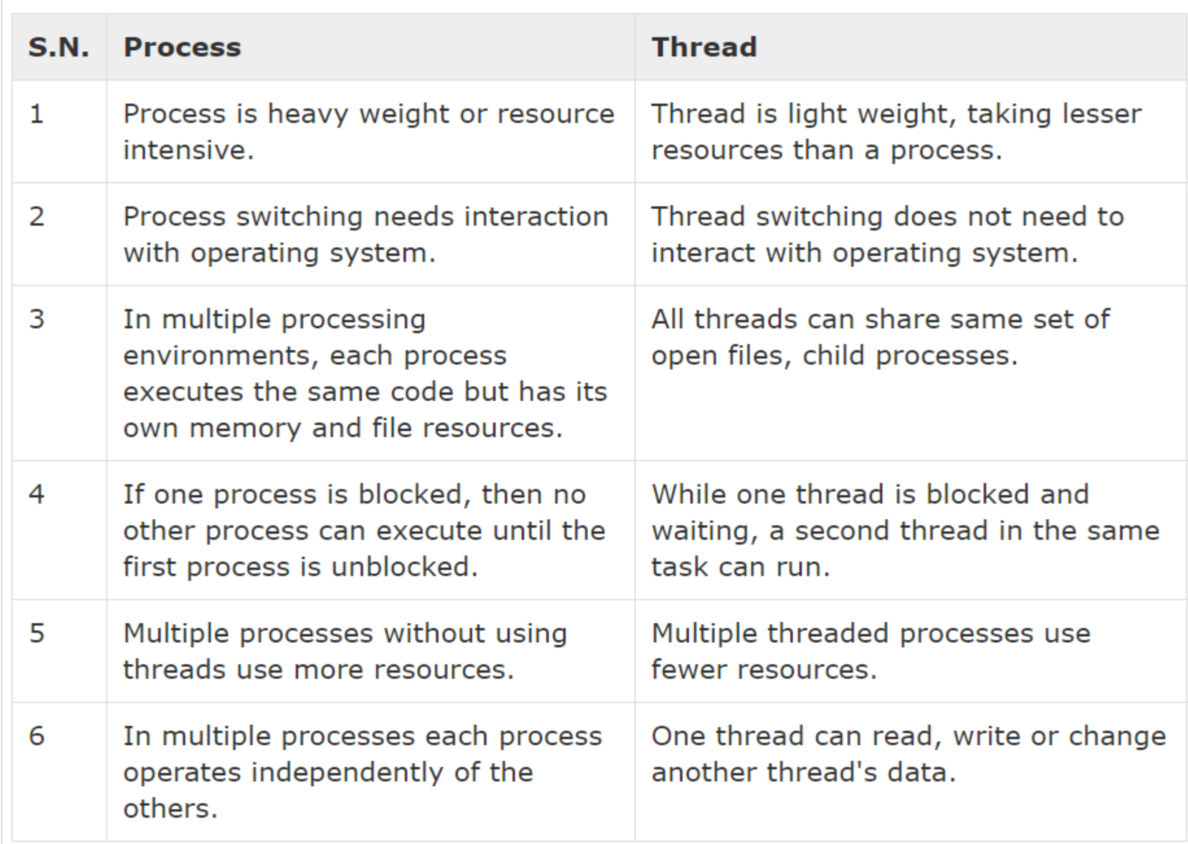
Single Process P with single thread



Single Process P with three threads

Fig:1.14 Single & Multithreaded Processes

DIFFERENCE BETWEEN PROCESS AND THREADS:



S.N.	Process	Thread
1	Process is heavy weight or resource intensive.	Thread is light weight, taking lesser resources than a process.
2	Process switching needs interaction with operating system.	Thread switching does not need to interact with operating system.
3	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.
4	If one process is blocked, then no other process can execute until the first process is unblocked.	While one thread is blocked and waiting, a second thread in the same task can run.
5	Multiple processes without using threads use more resources.	Multiple threaded processes use fewer resources.
6	In multiple processes each process operates independently of the others.	One thread can read, write or change another thread's data.

ADVANTAGES OF THREAD:

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

TYPES OF THREADS:

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

USER LEVEL THREADS:

- User threads are supported above the kernel and are managed without kernel support i.e., theyare implemented by thread library at the user level.

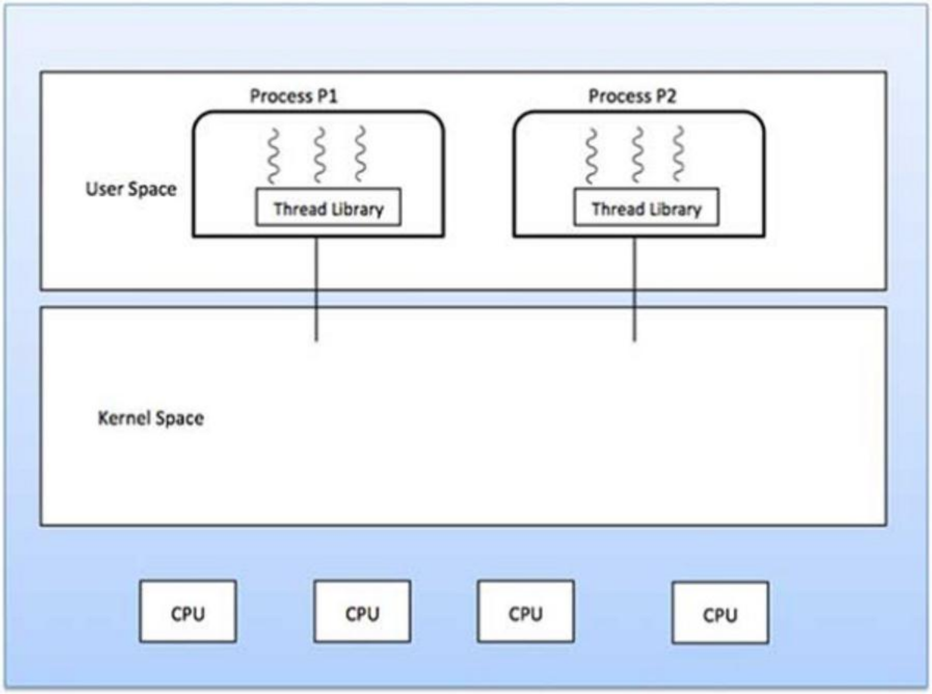


Fig:1.15 User Level Thread

- 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 starts with a single thread.
- **ADVANTAGES OF USER THREADS:**
  - 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:

- Kernel threads are supported and managed 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 performs thread creation, scheduling and management in Kernel space. Kernel threads are generally slower to create and manage than the user threads.

- **ADVANTAGES OF KERNEL LEVEL THREADS:**
    - 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 be multithreaded.
- **DISADVANTAGES:**
    - Kernel threads are generally slower to create and manage than the user threads.
    - Transfer of control from one thread to another within the same process requires a mode switch to the Kernel.

MULTI THREADING MODELS:

- Many systems provide support for both user and kernel threads, resulting in different multithreading models. Three common ways of establishing this relationship are:
    - Many to many relationship.
    - Many to one relationship.
    - One to one relationship.

MANY TO MANY MODEL:

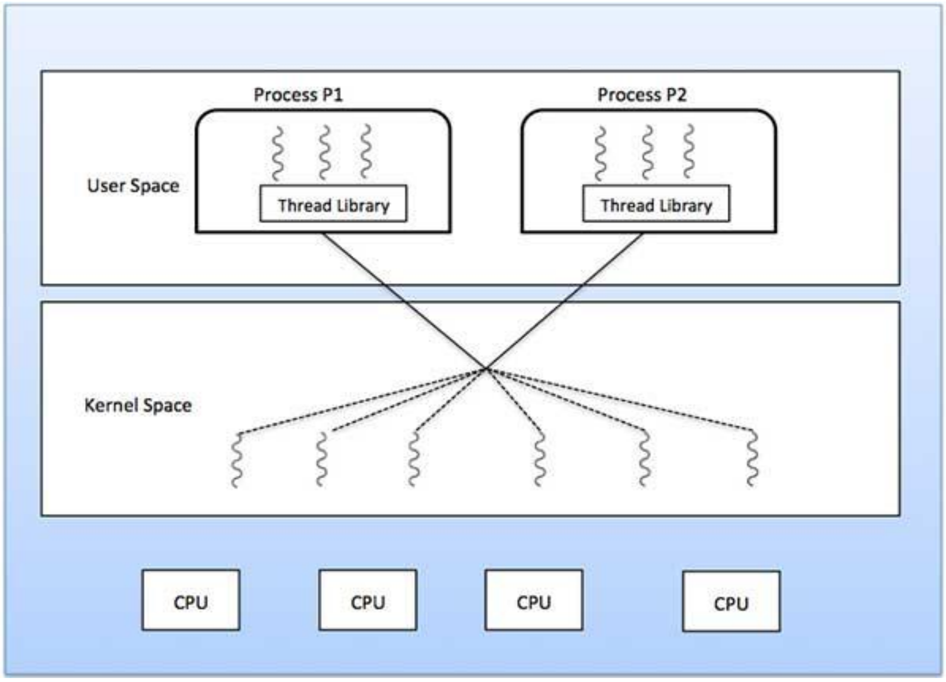


Fig:1.16 Thread-Many to Many Model

- The many-to-many model multiplexes many user-level threads to a smaller or equal number of kernel threads.
- The many-to-many threading model where 6 user level threads are multiplexing with 6 kernel level threads.
- In this model, developers can create as many user threads as necessary and the corresponding Kernel threads can run in parallel on a multiprocessor machine.
- This model provides the best accuracy on concurrency and when a thread performs a blocking system call, the kernel can schedule another thread for execution.

MANY TO ONE MODEL:

- The many-to-one model maps many user-level threads to one kernel thread. Thread management is done by the thread library in user space, so it is efficient; but the entire process will block if a thread makes a blocking system call.
- In addition, user level thread libraries implemented on Operating systems that do notsupport kernel threads use the many-to-one model.

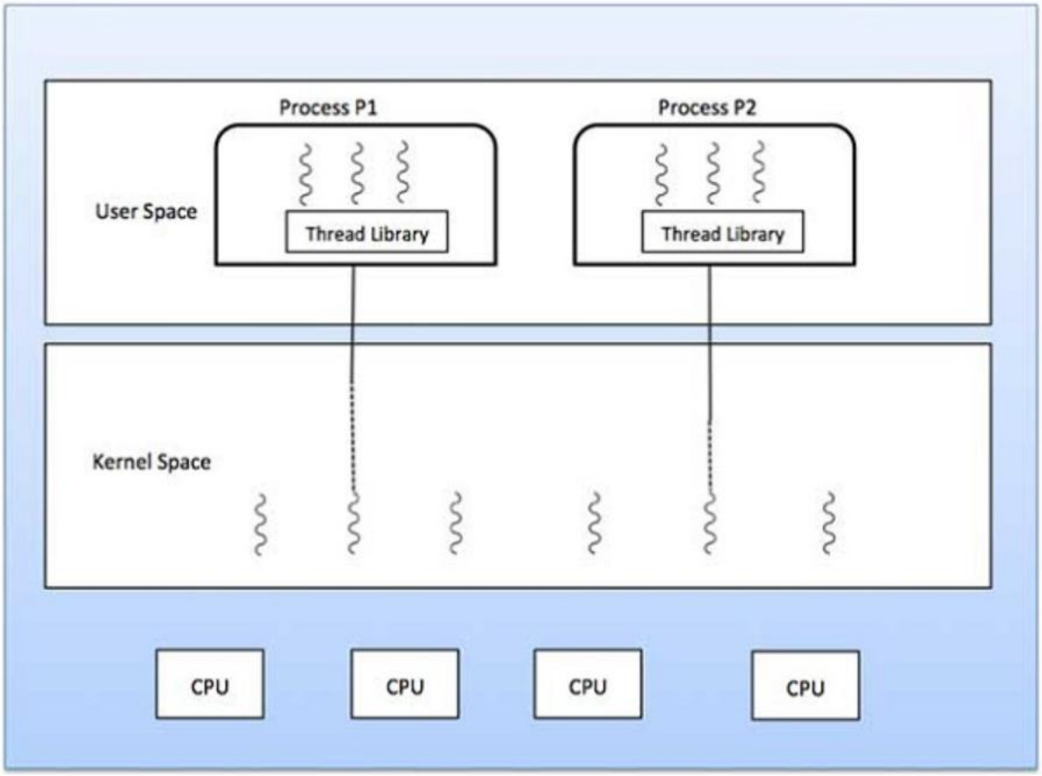


Fig:1.17 Thread-Many to One Model

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 allows another thread to run when a thread makes a blocking system call.
- It supports multiple threads 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.

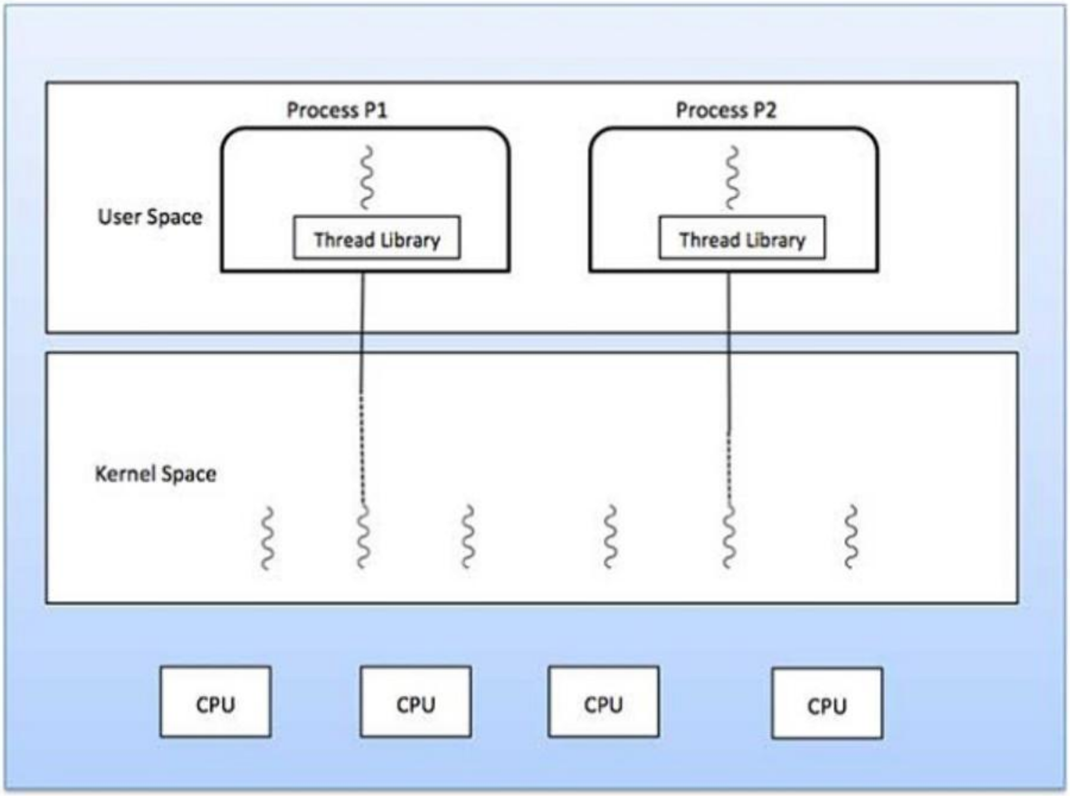
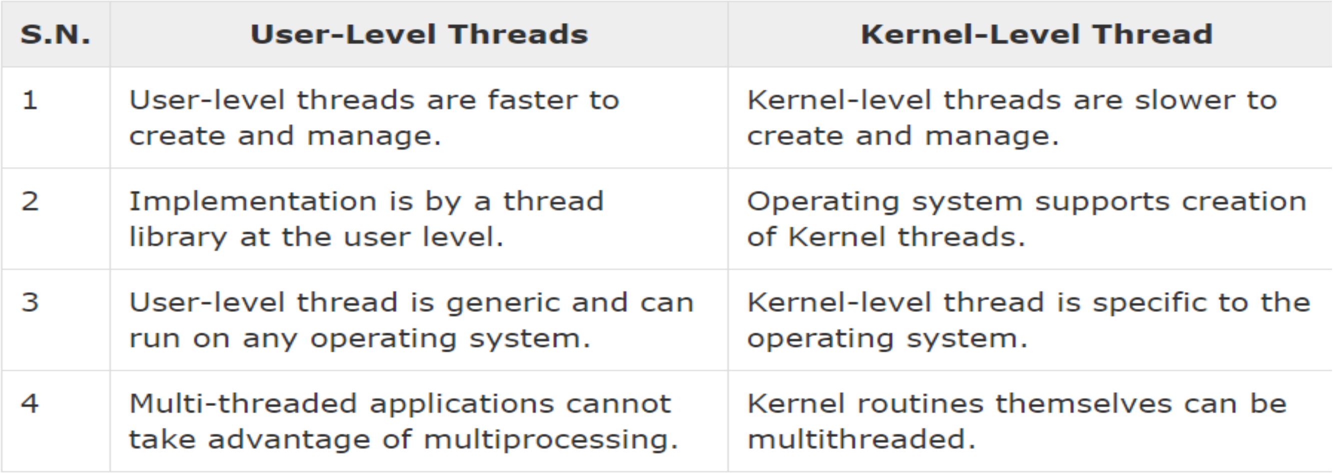


Fig:1.18 Thread-One to One Model

DIFFERENCE BETWEEN USER LEVEL AND KERNEL LEVEL THREADS:



S.N.	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 and can run on any operating system.	Kernel-level thread is specific to the operating system.
4	Multi-threaded applications cannot take advantage of multiprocessing.	Kernel routines themselves can be multithreaded.

OBJECTS:

- Objects are the basic run time entities in an object-oriented system.
- They may represent a person, a place, a bank account,a table of data or any item that the program has to handle.
- Programming problem is analyzed in terms of objects and the nature of communication between them.
- When a program is executed, the objects interact by sending messages to one another.

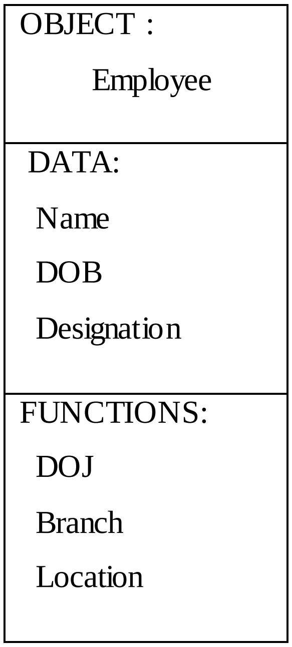


Fig:1.19 Object Model

- Real-world objects share two characteristics: They all have state and behavior.

- Desktop lamp may have only two possible states (on and off) and two possible behaviors (turnon, turn off), but your desktop radio might have additional states (on, off, current volume, current station) and behavior (turn on, turn off, increase volume, decrease volume, seek, scan, and tune).
- You may also notice that some objects, in turn, will also contain other objects.
- These real-world observations all translate into the world of object-oriented programming.

DEVICE MANAGEMENT:

- This component of operating system manages hardware devices via their respective drivers.
- The operating system performs the following ACTIVITIES for device management.
    - It keeps track of all the devices. The program responsible for keeping track of all the devices is known as I/O controller.
    - It provides a uniform interface to access devices with different physical characteristics.
    - It allocates the devices in an efficient manner.
    - It de-allocates the devices after usage.
    - It decides which process gets the device and how much time it should be used.
    - It optimizes the performance of each individual device.

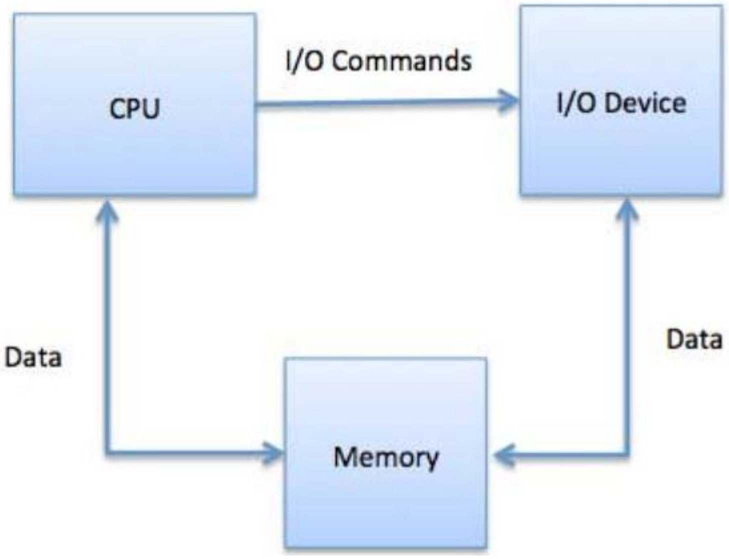


Fig:1.20 Device Management Diagram

- **Direct I/O** – CPU software explicitly transfer data to and from the controller’s data registers

- Direct I/O with polling – the device management software polls the device controllerstatus register to detect completion of the operation; device management isimplemented wholly in the device driver, if interrupts are not used
- Interrupt driven direct I/O – interrupts simplify the software's responsibility for detectingoperation completion; device management is implemented through the interaction of adevice driver and interrupt routine

- **Memory mapped I/O** – device addressing simplifies the interface (device seen as a range ofmemory locations)
  - Memory mapped I/O with polling – the device management software polls the devicecontroller status register to detect completion of the operation; device management is
  - implemented wholly in the device driver.
  - Interrupt driven I/O – interrupts simplify the software's responsibility for detectingoperation completion; device management is implemented through the interaction of adevice driver and interrupt routine
- **Direct memory access**– involves designing of hardware to avoid the CPU perform the transferof information between the device (controller's data registers) and the memory).

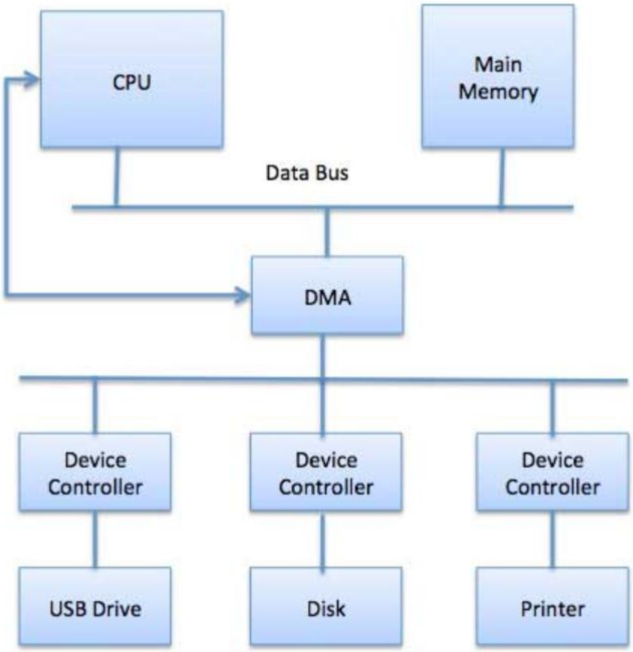


Fig:1.21 Overall Structure of Device Management

I/O SYSTEM ORGANIZATION:

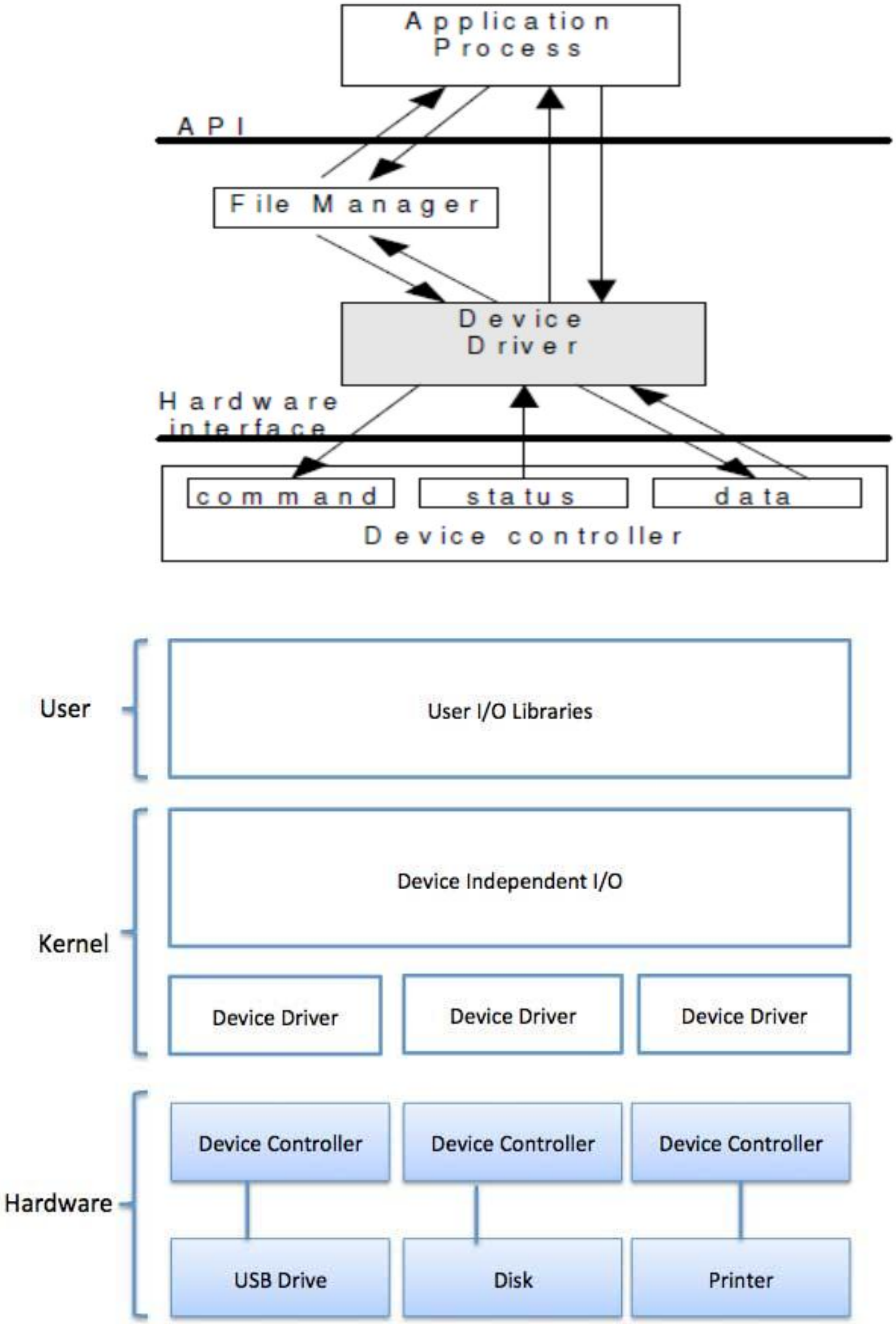


Fig:1.22 I/O System Organization

- An application process uses a device by issuing commands and exchanging data with the device management (device driver).
- Device drivers are software modules that can be plugged into an OS to handle a particular device.

- Operating System takes help from device drivers to handle all I/O devices.
- A device driver performs the following jobs –
  - To accept request from the device independent software above to it.
  - Interact with the device controller to take and give I/O and perform required error handling
  - Making sure that the request is executed successfully
- Since each device controller is specific to a particular device, the device driver implementation will be device specific,
  - Provide correct commands to the controller
  - Interpret the controller status register (CSR) correctly
  - Transfer data to and from device controller data registers as required for correct deviceoperation

DEVICE I/O Vs MEMORY MAPPED I/O

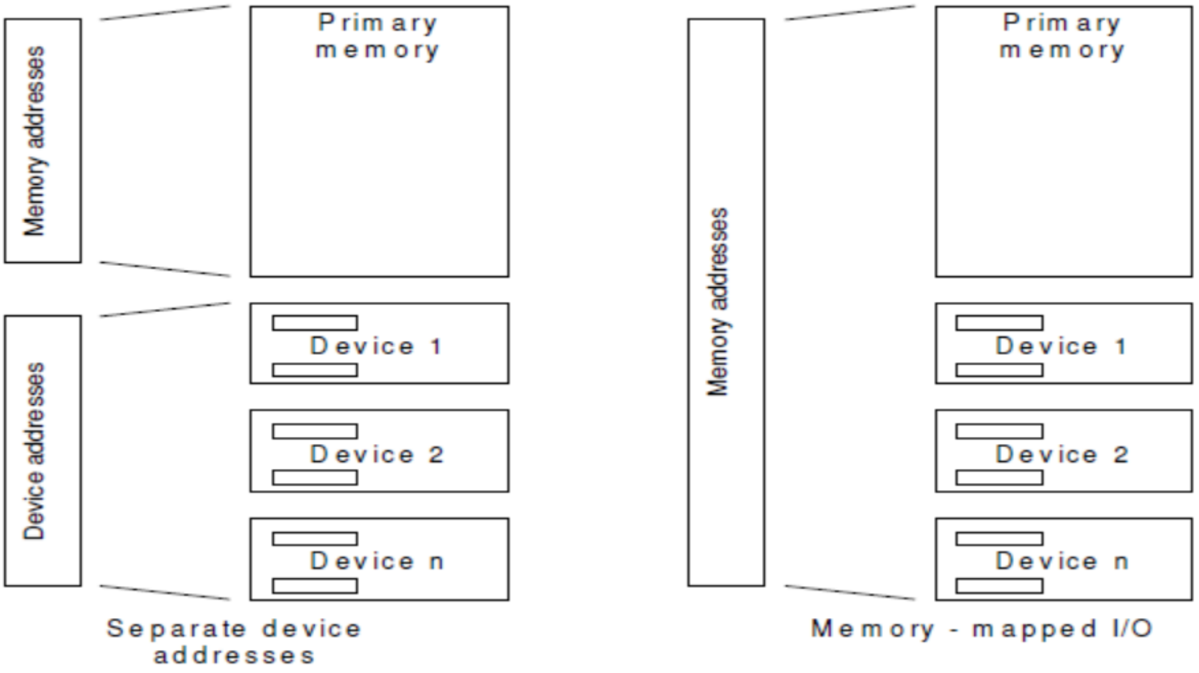


Fig:1.23 DEVICE I/O Vs MEMORY MAPPED I/O

I/O WITH POLLING:

- Each I/O operation requires that the software and hardware coordinate their operations to accomplish desired effect
- In direct I/O pooling this coordination is done in the device driver;

- While managing the I/O, the device manager will poll the busy/done flags to detect the operation’s completion; thus, the CPU starts the device, then polls the CSR to determine when the operation has completed
- With this approach is difficult to achieve high CPU utilization, since the CPU must constantly check the controller status;

I/O with polling – read

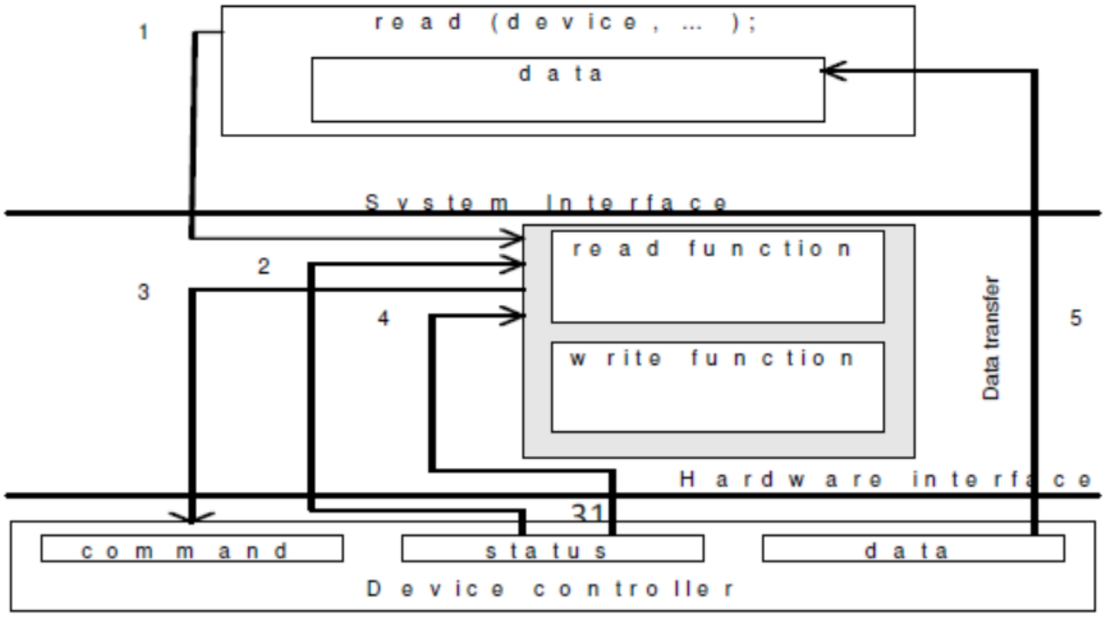


Fig:1.24 I/O with Polling-Reading Operation

- Application process requests a read operation
- The device driver queries the CSR to determine whether de device is idle; if device is busy, thedriver waits for it to become idle
- The driver stores an input command into the controller’s command register, thus starting thedevice
- The driver repeatedly reads the content of CSR to detect the completion of the read operation
- The driver copies the content of the controller's data register(s) into the main memory user’s processes space.

I/O with polling – write

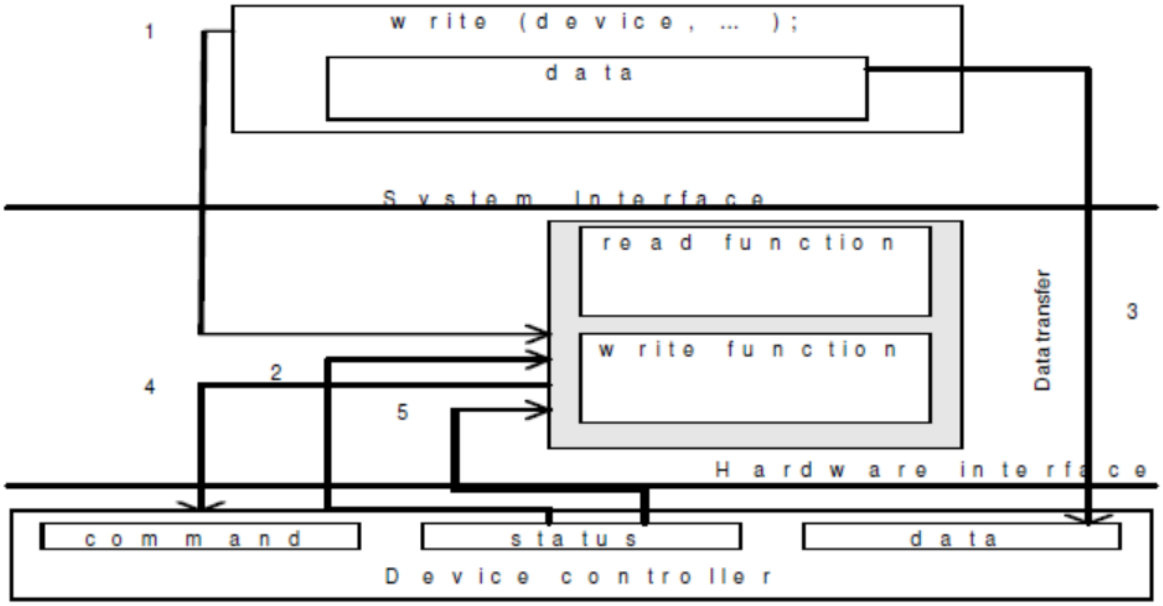


Fig:1.25 I/O with Polling-Writing Operation

- The application process requests a write operation
- The device driver queries the CSR to determine if the device is idle; if busy, it will wait tobecome idle
- The device driver copies data from user space memory to the controller's data register(s).
- The driver stores an output command into the command register, thus starting the device.
- The driver repeatedly reads the CSR to determine when the device completed its operation.

INTERUPT DRIVEN I/O:

- The application process requests a read operation
- The device driver queries the CSR to find out if the device is idle; if busy, then it waits until the device becomes idle
- The driver stores an input command into the controller's command register, thus starting the device
- The device completes the operation and interrupts the CPU, thereforecausing an interrupt handler to run
- The interrupt handler determines which device caused the interrupt; it thenbranches to the device handler for that device

- The device driver retrieves the pending I/O status information from thedevice status table
- The device driver copies the content of the controller's data register(s)into the user process's space
- The device handler returns the control to the application process (knowing thereturn address from the device status table). Same sequence (or similar) ofoperations will be accomplished for an output operation.

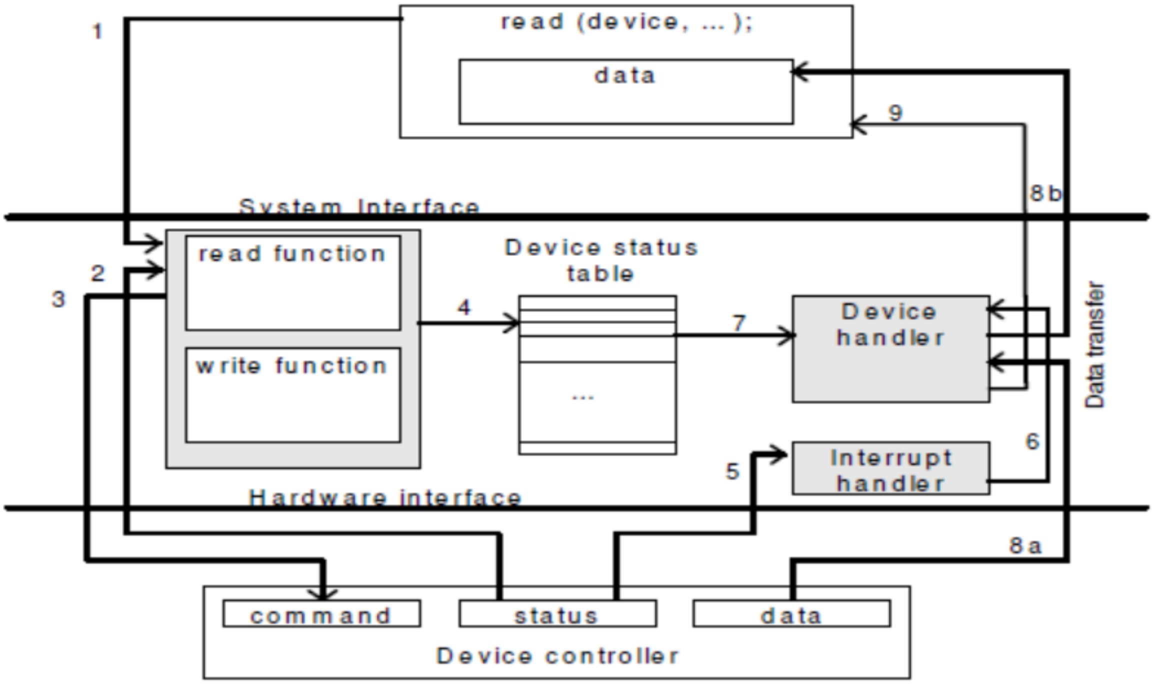


Fig:1.26 INTERUPT DRIVEN I/O:

DMA – DIRECT MEMORY ACCESS:

- DMA controllers are able to read and write information directly from /to primary memory, with no software intervention
- The I/O operation has to be initiated by the driver
- DMA hardware enables the data transfer to be accomplished without using the CPU at all
- The DMA controller must include an address register (and address generation hardware loaded by the driver with a pointer to the relevant memory block; this pointer is used by the DMA hardware to locate the target block in primary memory
- Traditional I/O
  - Polling approach:

        - CPU transfer data between the controller data registers and the primary memory
        - Output operations - device driver copies data from the application process dataarea to the controller; vice versa for input operations
    - Interrupt driven I/O approach:
        - The interrupt handler is responsible for the transfer task.

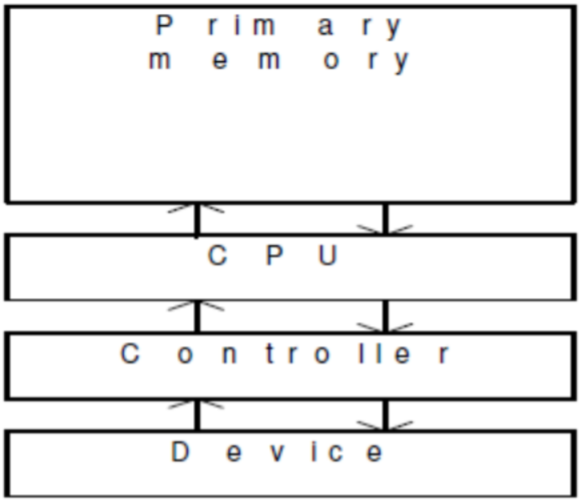
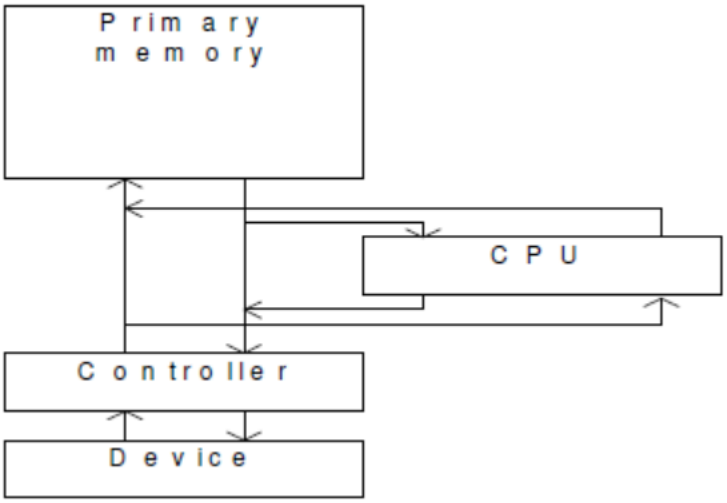


Fig:1.27: Interrupt driven I/O approach:



TYPICAL DMA:

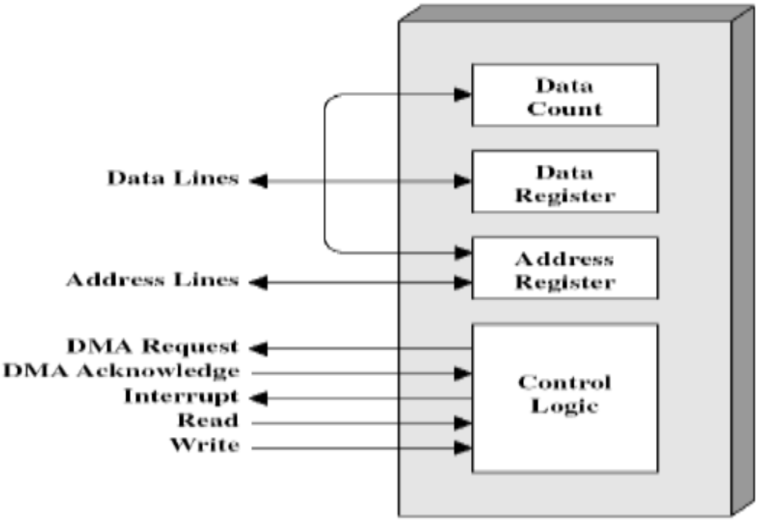
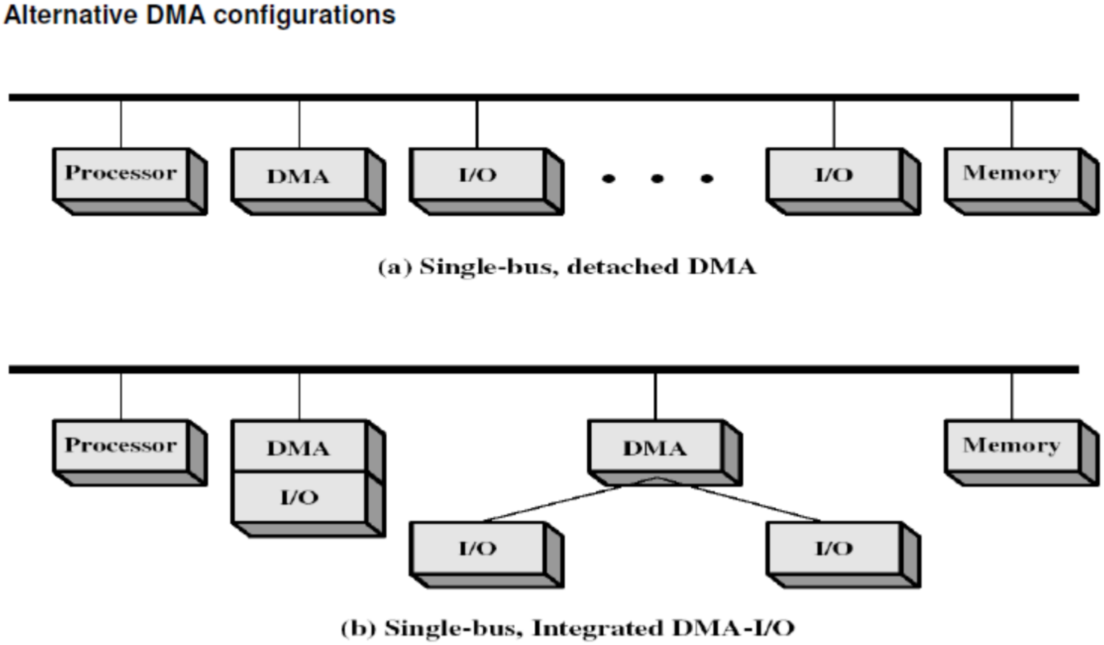


Fig:1.28: Direct Memory Access

- Mimics the processor. Transfers data to/from memory over system bus


Alternative DMA configurations
Processor
DMA
I/O
• • •
I/O
Memory
(a) Single-bus, detached DMA
Processor
DMA
I/O
DMA
I/O
I/O
Memory
(b) Single-bus, Integrated DMA-I/O

BUFFERING:

- Buffering is a technique by which a device manager keeps the slower I/O devices busy when aprocess is not requiring I/O operations.
- **Input buffering** is the process of reading the data into the primary memory before the processrequests it.
- **Output buffering** is the process of saving the data in the memory and then writing it to thedevice while the process continues its execution.

HARDWARE LEVEL BUFFERING:

- Consider a simple character device controller that reads a single byte form a modem for each input operation.
- **Two operations**
    - Normal operation
    - Buffered operation

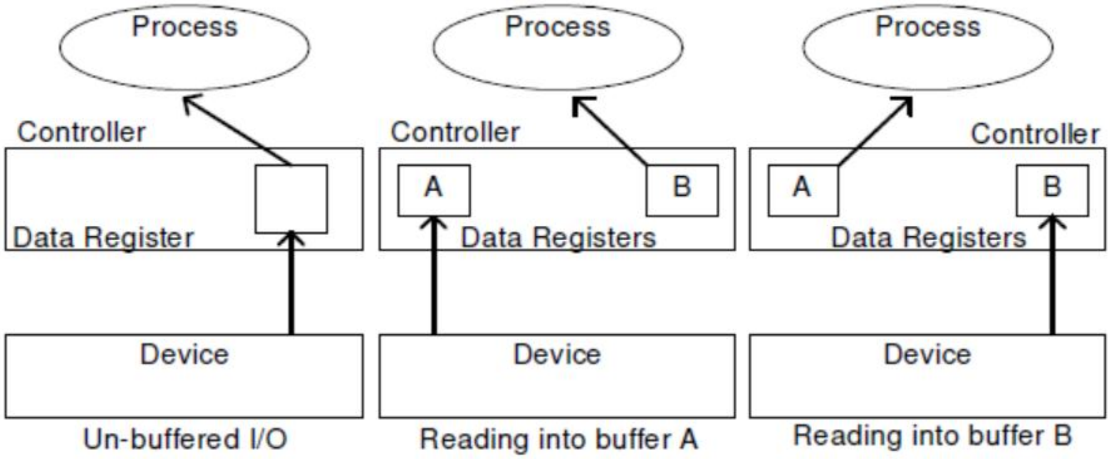


Fig:1.29 HARDWARE LEVEL BUFFERING:

DRIVER LEVEL BUFFERING:

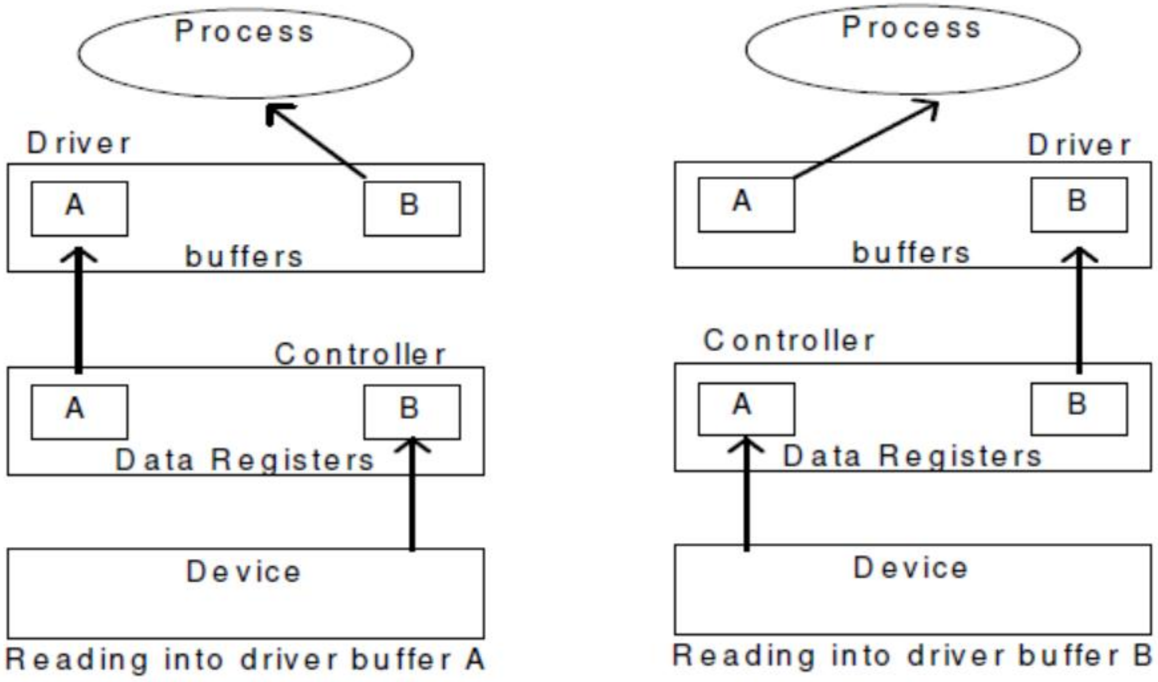
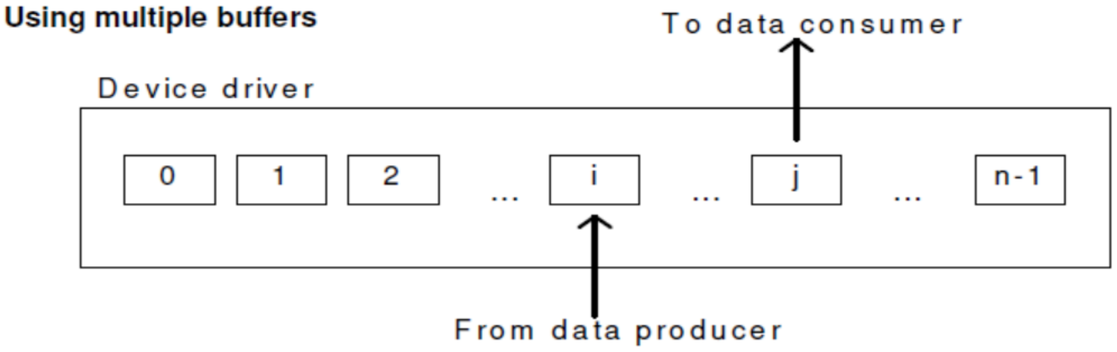


Fig:1.30 DRIVER LEVEL BUFFERING:



DEVICE DRIVER:

- It is a software program that controls a particular type of device attached to thecomputer.

- It provides an interface to the hardware devices without the requirement toknow the precise information about the hardware.
- A device driver communicates with the device through a bus or communication subsystem.
- **RESPONSIBLITIES:**
    - Initialize devices
    - Interpreting the commands from the operating system
    - Manage data transfers
    - Accept and process interrupts
    - Maintain the integrity of driver and kernel data structures

DEVICE DRIVER INTERFACE:

- Each operating system defines an architecture for its device management system. The designsare different from operating system to operating system; there is no universal organization
- Each operating system has two major interfaces to the device manager:
    - The driver API
    - The interface between a driver and the operating system kernel.

DRIVER API

- Provides a set of functions that an programmer can call to manage a device (usually communicates orstorage). The device manager
    - Must track the state of the device: when it is idle, when is being used and by whichprocess
    - Must maintain the information in the device status table
    - May maintain a device descriptor to store other information about the device
- OPEN/CLOSE functions to allow initiate/terminate of the device's use
    - **open** – allocates the device and initializes the tables and the device for use
    - **close** – releases dynamic tables entries and releases the device

THE DRIVER - KERNEL INTERFACE

- The device driver must execute privileged instructions when it starts the device; this means that the device driver must be executed as part of the operating system rather than part of a user program.
- The driver must also be able to read/write info from/to the address spaces of differentprocesses, since same device driver can be used by different processes
- Two ways of dealing with the device drivers
  - **Old way:** Driver is part of the operating system, to add a new device driver, the whole OSmust have been complied
  - **Modern way:**Drivers installation is allowed without re-compilation of the OS by usingreconfigurable device drivers; the OS dynamically binds the OS code to the driverfunctions.

Reconfigurable device drivers

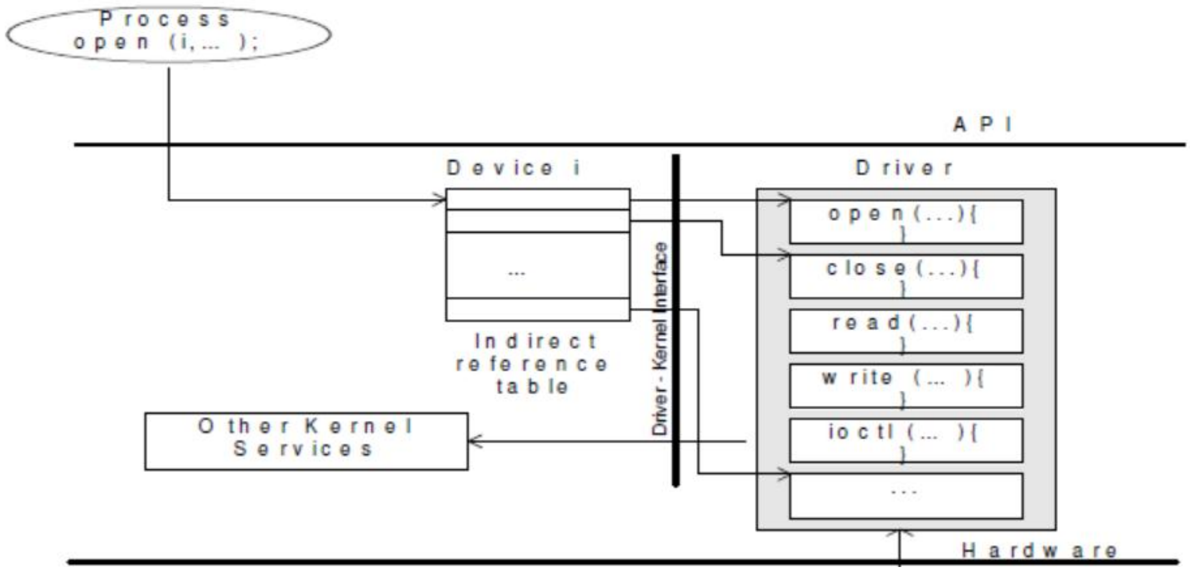


Fig:1.31 Reconfiguring Device Drivers

SATHYABAMA
INSTITUTE OF SCIENCE AND TECHNOLOGY
(DEEMED TO BE UNIVERSITY)
Accredited "A" Grade by NAAC | 12B Status by UGC | Approved by AICTE
www.sathyabama.ac.in

SCHOOL OF COMPUTING

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

UNIT – II – Operating System – SBS1206

OPERATING SYSTEM:

UNIT II **(9 Hrs)**

CPU Scheduling: CPU Schedulers - Scheduling Criteria - Scheduling Algorithms.Process Synchronization: Critical-Section Problem - Synchronization Hardware - Semaphores Classical Problems of Synchronization - Critical Region - Monitors.

UNIT 2

PROCESS MANAGEMENT

INTRODUCTION TO PROCESSES:

- Early computer systems allowed only one program to be executed at a time.
- This program had complete control of the system and had access to all the system's resources.
- In contrast, current-day computer systems allow multiple programs to be loaded into memory and executed concurrently.
- This evolution required firmer control and more compartmentalization of the various programs; and these needs resulted in the notion of a process, which is a program in execution.
- A process is basically a program in execution. The execution of a process must progress in a sequential fashion.
- A process will need certain resources—such as CPU time, memory, files, and I/O devices —to accomplish its task.
- These resources are allocated to the process either when it is created or while it is executing.
- **Definition:**A process is defined as an entity which represents the basic unit of work to be implemented in the system.
- A process is the unit of work in most systems.
- Systems consist of a collection of processes:
    - Operating-system processes execute system code, and
    - User processes execute user code.
- Although traditionally a process contained only a single thread of control as it ran, most modern operating systems now support processes that have multiple threads.
- A process is the unit of work in a modern time-sharing system.
- The more complex the operating system is, the more it is expected to do on behalf of its users.
- Although its main concern is the execution of user programs, it also needs to take care of various system tasks that are better left outside the kernel itself.

- A system therefore consists of a collection of processes: operatingsystem processes executing system code and user processes executing user code.

PROCESSES:

- A process is mainly a program in execution where the execution of a process must progress in a sequential order or based on some priority or algorithms.
- In other words, it is an entity that represents the fundamental working that has been assigned to a system.
- When a program gets loaded into the memory, it is said to as process. This processing can be categorized into 4 sections. These are:
    - Heap
    - Stack
    - Data
    - Text

PROCESS CONCEPTS:

- A question that arises in discussing operating systems involves what to call all the CPU activities.
- A batch system executes jobs, whereas a time-shared system has user programs, or tasks.
- Even on a single-user system such as Microsoft Windows, a user may be able to run several programs at one time: a word processor, a web browser, and an e-mail package.
- Even if the user can execute only one program at a time, the operating system may need to support its own internal programmed activities, such as memory management.
- In many respects, all these activities are similar, so we call all of them processes.
- The terms job and process are used almost interchangeably used.
- Although we personally prefer the term process, much of operating-system theory and terminology was developed during a time when the major activity of operating systems was job processing.
- It would be misleading to avoid the use of commonly accepted terms that include the word job (such as job scheduling) simply because process has superseded job.

THE PROCESS:

- An operating system executes a variety of programs:
  - Batch system – jobs
  - Time-shared systems – user programs or tasks
- We use the terms job and process almost interchangeably.
- Process – a program in execution; process execution must progress in sequential fashion.
- A process is a program in execution. A process is more than the program code, which is sometimes known as the text section.
- A process includes:
  - Counter
  - Program stack
  - Data section
    - The current activity, as represented by the value of the program counter and the contents of the processor's registers.
    - The process stack contains temporary data (such as function parameters, return addresses, and local variables)
    - A data section, which contains global variables.
- Process may also include a heap, which is memory that is dynamically allocated during process run time.

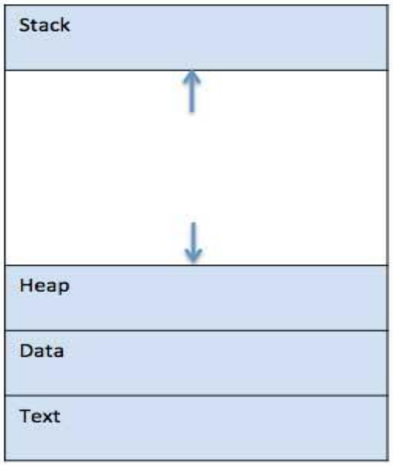


Fig:2.1 The Process

- **STACK** - The process Stack contains the temporary data such as method/function parameters, return address and local variables.
- **HEAP** - This is dynamically allocated memory to a process during its run time.
- **TEXT -** This includes the current activity represented by the value of Program Counter and the contents of the processor's registers**.**
- **DATA -** This section contains the global and static variables.
- We emphasize that a program by itself is not a process; a program is a passive entity, such as a file containing a list of instructions stored on disk (often called an executable file).
- Whereas a process is an active entity, with a program counter specifying the next instruction to execute and a set of associated resources.
- A program becomes a process when an executable file is loaded into memory.

PROCESS STATE / PROCESS LIFE CYCLE:

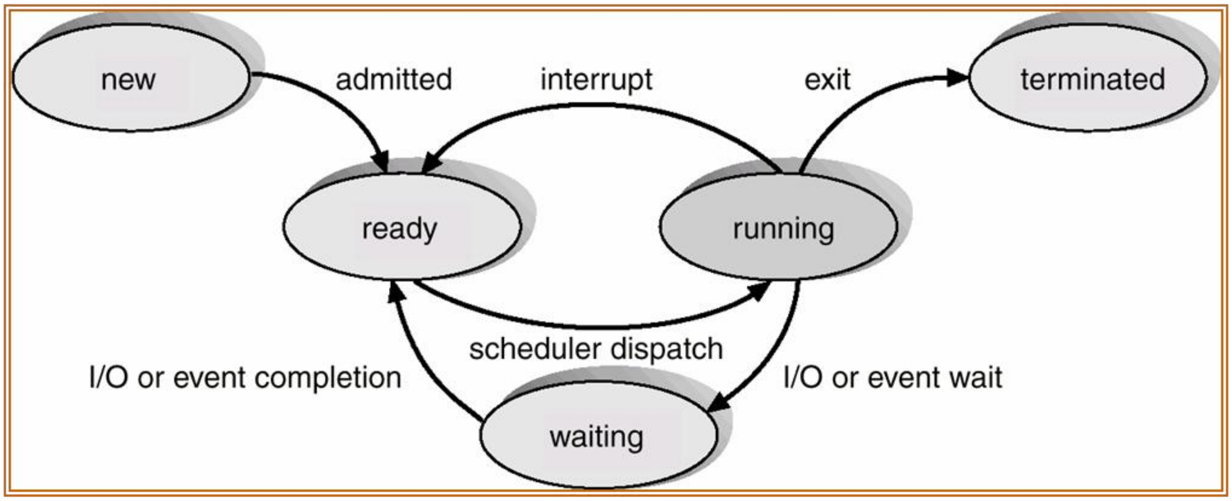


Fig: 2.2-Process State

- When a process executes, it passes through different states.
- These stages may differ in different operating systems, and the names of these states are also not standardized.
- In general, a process can have one of the following five states at a time.
  - New/Start
  - Running
  - Waiting
  - Ready

    - Terminated

- **START / NEW –**
    - This is the initial state when a process is first started/created.
- **READY–**
    - 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.
    - Process may come into this state after Start state or while running it by but interrupted by the scheduler to assign CPU to some other process.
- **RUNNING –**
    - Once the process has been assigned to a processor by the OS scheduler, the process state is set to running and the processor executes its instructions.
- **WAITING –**
    - Process moves into the waiting state if it needs to wait for a resource, such as waiting for user input, or waiting for a file to become available.
- **TERMINATED OR EXIT –**
    - Once the process finishes its execution, or it is terminated by the operating system, it is moved to the terminated state where it waits to be removed from main memory.
- These names are arbitrary, and they vary across operating systems.

PROCESS CONTROL BLOCK (PCB):

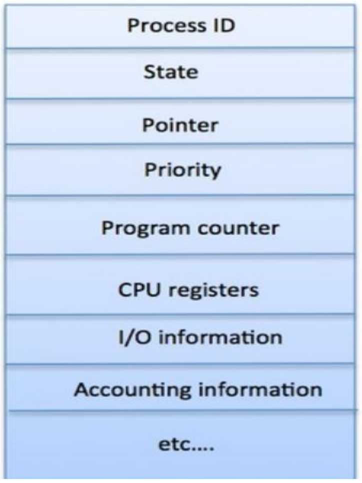


Fig:2.3 Process Control Block

- A Process Control Block is a data structure maintained by the Operating System for every process.
- The PCB is identified by an integer process ID (PID).
- A PCB keeps all the information needed to keep track of a process as listed below ,
- **PROCESS STATE**
    - The current state of the process i.e., whether it is ready, running, waiting, or whatever.
- **PROCESS PRIVILEGES**
    - This is required to allow/disallow access to system resources.
- **PROCESS ID**
    - Unique identification for each of the process in the operating system**.**
- **POINTER**
    - A pointer to parent process.
- **PROGRAM COUNTER**
    - Program Counter is a pointer to the address of the next instruction to be executed for this process**.**
- **CPU REGISTERS**
    - Various CPU registers where process need to be stored for execution for running state.
- **CPU SCHEDULING INFORMATION**
    - Process priority and other scheduling information which is required to schedule the process.
- **MEMORY MANAGEMENT INFORMATION**
    - This includes the information of page table, memory limits, Segment table depending on memory used by the operating system.
- **ACCOUNTING INFORMATION**
    - This includes the amount of CPU used for process execution, time limits, execution ID etc.
- **IO STATUS INFORMATION**

    - This includes a list of I/O devices allocated to the process.
- The architecture of a PCB is completely dependent on Operating System and may contain different information in different operating systems. (shown in above diagram)
- The PCB is maintained for a process throughout its lifetime, and is deleted once the process terminates.

DIAGRAM SHOWING CPU SWITCH PROCESS TO PROCESS

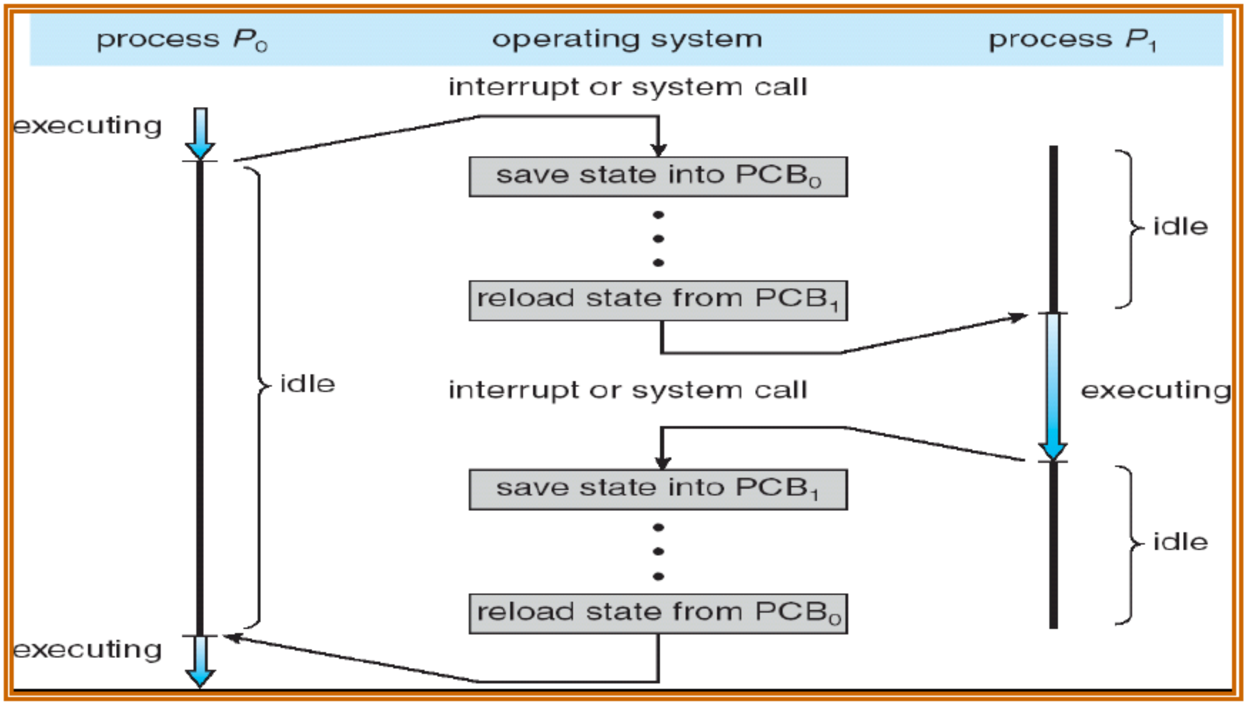


Fig:2.4 Diagram Showing CPU Switch Process To Process

PROCESS SCHEDULING:

- A uniprocessor system can have only one running process. If more process exist, the rest must wait until the CPU is free and can be rescheduled.
- The objective of multiprogramming is to have some process running at all times, to maximize CPU utilization.

- The objective of time sharing is to switch the CPU among processes so frequently that users can interact with each program while it is running.
- To meet these objectives, the process scheduler selects an available process (possibly from a set of several available processes) for program execution on the CPU.

PROCESS SCHEDULING QUEUES:

- The OS maintains all PCBs in Process Scheduling Queues.
- The OS maintains a separate queue for each of the process states and PCBs of all processes in the same execution state are placed in the same queue.
- When the state of a process is changed, its PCB is unlinked from its current queue and moved to its new state queue.
- Process migration between the various queues:
  - Job queue
  - Ready queue
  - Device queues

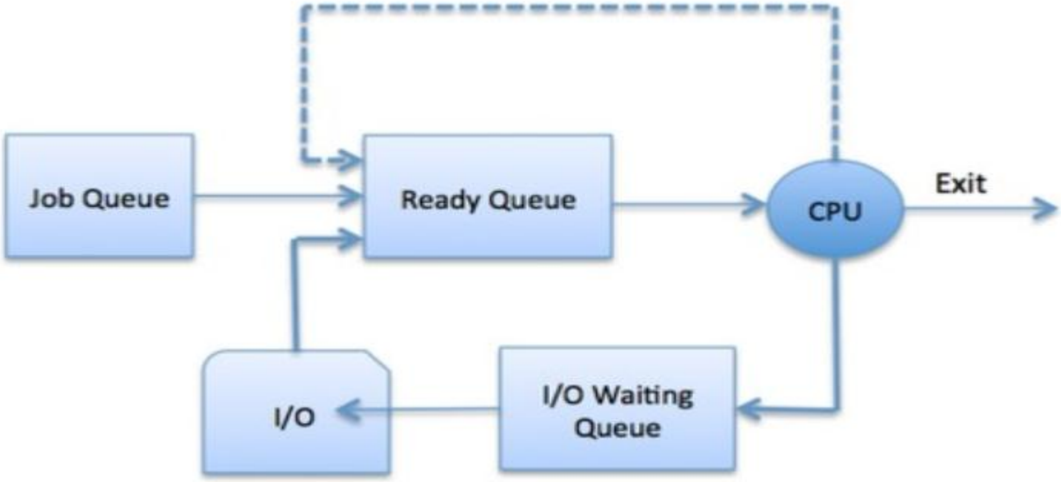


Fig:2.5 Process Scheduling Queue

- **JOB QUEUE**−This queue keeps all the processes in the system.
- **READY QUEUE** − This queue keeps a set of all processes residing in main memory, ready and waiting to execute. A new process is always put in this queue.
- **DEVICE QUEUES** −The processes which are blocked due to unavailability of an I/O device constitute this queue.

- The OS can use different policies to manage each queue (FIFO, Round Robin, Priority, etc.).
- The OS scheduler determines how to move processes between the ready and run queues which can only have one entry per processor core on the system; in the above diagram, it has been merged with the CPU.

TWO STATE PROCESS MODEL:

- Two-state process model refers to running and non-running states which are described below
- **RUNNING**
  - When a new process is created, it 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 is implemented by using linked list.

QUEUING DIAGRAM REPRESENTATION FOR PROCESS SCHEDULING

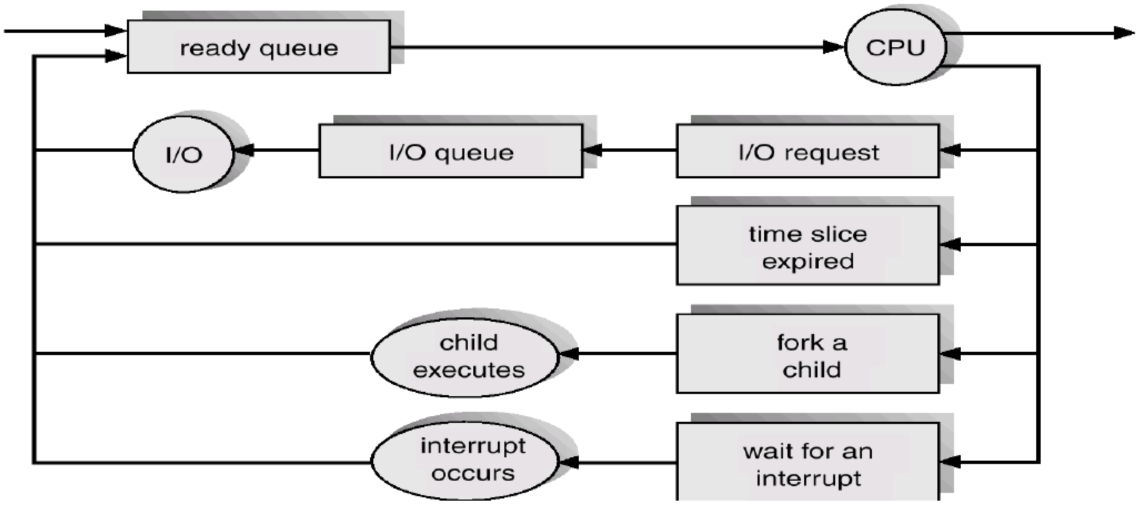


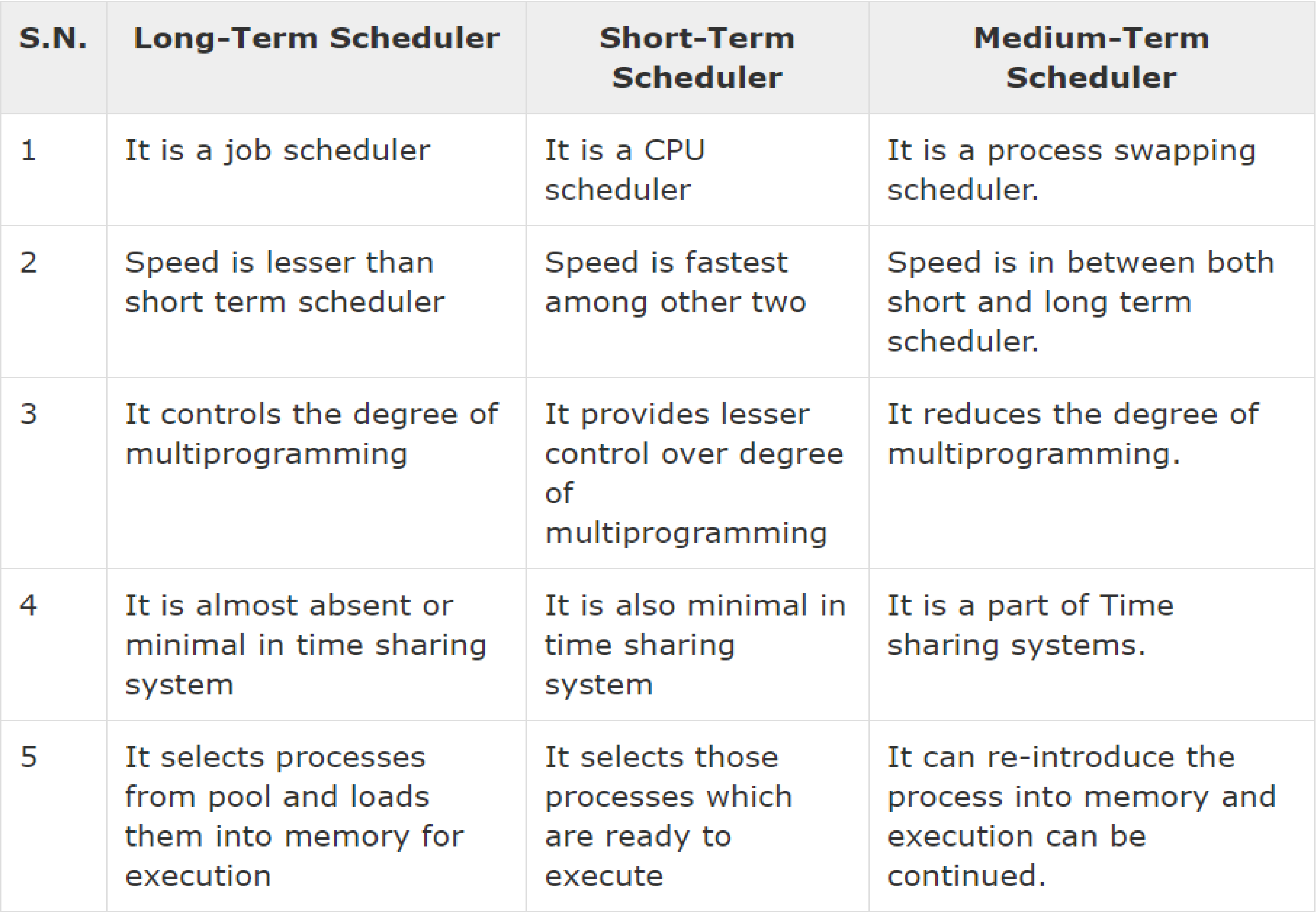
Fig:2.6 Queuing Diagram Representation For Process Scheduling

- Each rectangular box represents a queue.
- Two types of queues are present:
  - the ready queue and
  - a set of device queues

- The circles represent the resources that serve the queues, and the arrows indicate the flow of processes in the system.
- A new process is initially put in the ready queue.
- It waits there until it is selected for execution, or is dispatched.
- Once the process is allocated the CPU and is executing, one of several events could occur:
  - The process could issue an I/O request and then be placed in an I/O queue.
  - The process could create a new subprocess and wait for the subprocess's termination.
  - The process could be removed forcibly from the CPU, as a result of an
  - Interrupt, and be put back in the ready queue.
- In the first two cases, the process eventually switches from the waiting state to the ready state and is then put back in the ready queue.
- A process continues this cycle until it terminates, at which time it is removed from all queues and has its PCB and resources de-allocated.

SCHEDULERS:

- Schedulers are special system software which handles the 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



S.N.	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.

- **LONG TERM SCHEDULER:**
  - It is also called a job scheduler.
  - A long-term scheduler determines which programs are admitted to the system for processing.
  - It 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.

- **SHORT TERM SCHEDULER:**
  - It is also called as CPU scheduler.
  - Its main objective is to increase 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 a process among the processes that are ready to execute and allocates CPU to one of them.
  - Short-term schedulers, also known as dispatchers, make the decision of which process to execute next.
  - Short-term schedulers are faster than long-term schedulers.
- **MEDIUM TERM SCHEDULER:**
  - Medium-term scheduling is a part of 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.
  - A running process may become suspended if it makes an I/O request.
  - A suspended process cannot make any progress towards completion.
  - In this condition, to remove the process from memory and make space for other processes, 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.

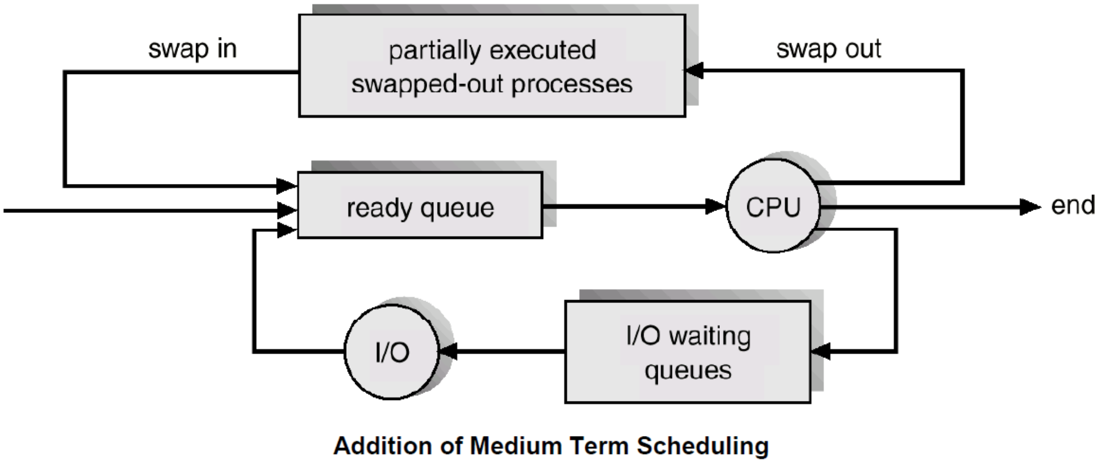

swap in
partially executed swapped-out processes
swap out
ready queue
CPU
end
I/O
I/O waiting queues
Addition of Medium Term Scheduling

Fig:2.7 Medium Term Scheduler

CONTEXT SWITCH:

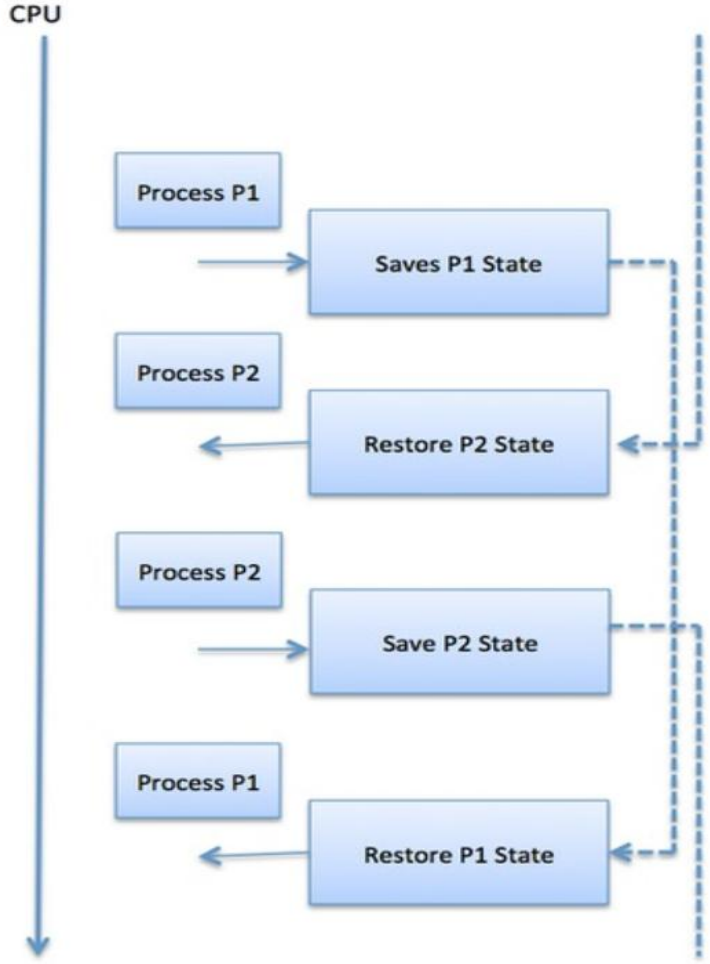


Fig:2.8 Context Switch

- 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.
- Context switches are computationally intensive since register and memory state must be saved and restored.
- To avoid the amount of context switching time, some hardware systems employ two or more sets of processor registers.

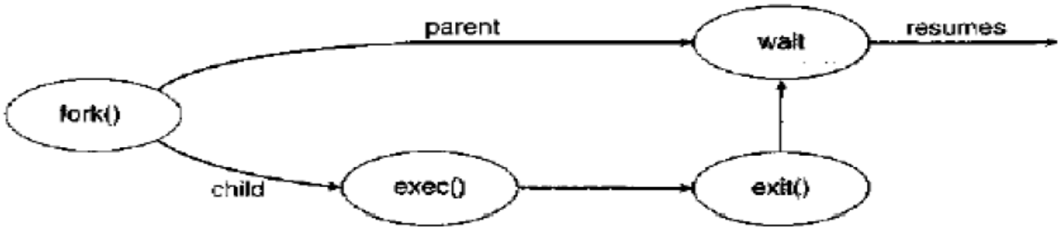
- When the process is switched, the following information is stored for later use.
    - Program Counter
    - Scheduling information
    - Base and limit register value
    - Currently used register
    - Changed State
    - I/O State information
    - Accounting information

OPERATIONS ON PROCESSES:

- The processes in the system can execute concurrently, and they must be created and deleted dynamically.
- Thus, the operating system must provide a mechanism (or facility) for process creation and termination.
- In this section, we explore the mechanisms involved in creating processes and illustrate process creation on UNIX and Windows systems.

PROCESS CREATION:

- Parent process creates children processes, which, in turn create other processes, forming a tree of processes.



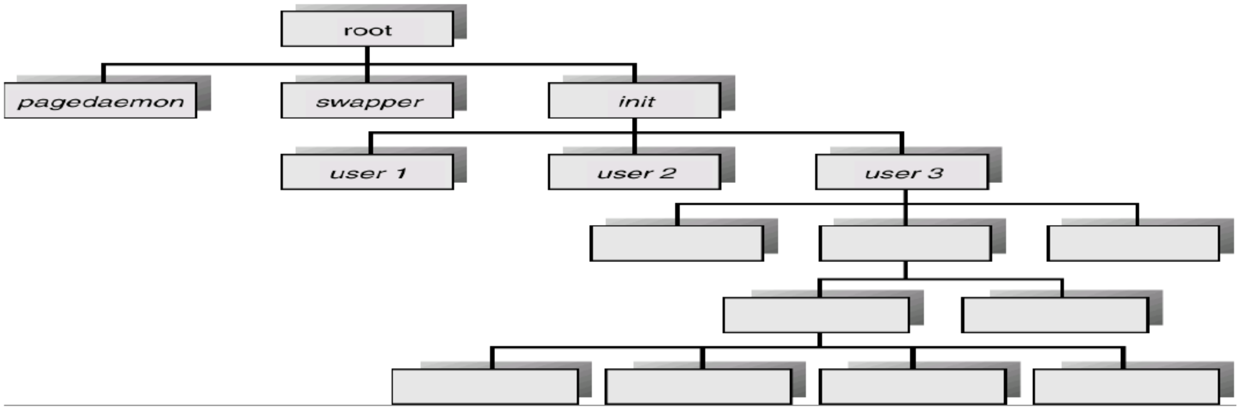
Process creation

Fig:2.9 Process Creation Diagram

- Resource sharing
    - Parent and children share all resources.
    - Children share subset of parent’s resources.

- Parent and child share no resources.

- Execution
  - Parent and children execute concurrently.
  - Parent waits until children terminate.
- A process may create several new processes, via a create-process system call, during the course of execution.
- The creating process is called a parent process, and the new processes are called the children of that process.



A Tree of Processes On A Typical UNIX System

Fig:2.10 A tree of Processes on a Typical UNIX System

- Each of these new processes may in turn create other processes, forming a tree of processes.
- When a process creates a subprocess, that subprocess may be able to obtain its resources directly from the operating system, or it may be constrained to a subset of the resources of the parent process.
- When a process creates a new process, two possibilities exist in terms of execution:
  - The parent continues to execute concurrently with its children.
  - The parent waits until some or all of its children have terminated.
- There are also two possibilities in terms of the address space of the new process:
  - The child process is a duplicate of the parent process (it has the same program and data as the parent).

COOPERATING PROCESSES:

- Processes executing concurrently in the operating system may be either independent processes or cooperating processes.
- A process is independent if it cannot affect or be affected by the other processes executing in the system.
- Any process that does not share data with any other process is independent.
- A process is cooperating if it can affect or be affected by the other processesexecuting in the system.
- There are several reasons for providing an environment that allows process cooperation:
- **INFORMATION SHARING:**
  - Since several users may be interested in the same piece of information (for instance, a shared file), we must provide an environment to allow concurrent access to such information.
- **COMPUTATION SPEEDUP:**
  - If we want a particular task to run faster, we must break it into subtasks, each of which will be executing in parallel with the others.
- **MODULARITY:**
  - We may want to construct the system in a modular fashion, dividing the system functions into separate processes or threads.
- **CONVENIENCE:**
  - Even an individual user may work on many tasks at the same time.
- Concurrent execution of cooperating processes requires mechanisms that allow processes to communicate with one another and to synchronize their actions.
- To illustrate the concept of cooperating processes, let's consider the producer-consumer problem, which is a common paradigm for cooperating processes.
- A producer process produces information that is consumed by a consumer process.
- To allow producer and consumer processes to run concurrently, we must have available **A BUFFER** of items that can be filled by the producer and emptied by the consumer.
- This **BUFFER** will reside in a region of memory that is shared by the producer and consumer processes.
- The producer and consumer must be synchronized, so that the consumer does not try to consume an item that has not yet been produced.

- Two types of buffers can be used:
    - Unbounded buffer
    - Bounded buffer
- **UNBOUNDED BUFFER** –
    - Places no practical limit on the size of the buffer.
    - The consumer may have to wait for new items, but the producer can always produce new items.
- **BOUNDED BUFFER –**
    - It assumes a fixed buffer size.
    - In this case, the consumer must wait if the buffer is empty, and the producer must wait if the buffer is full.

CPU SCHEDULING:

- CPU scheduling is a process which allows one process to use the CPU while the execution of another process is on hold(in waiting state) due to unavailability of any resource like I/O etc, thereby making full use of CPU
- Whenever the CPU becomes idle, the operating system must select one of the processes in the ready queue to be executed.
- The selection process is carried out by the short-term scheduler (or CPU scheduler).

CPU SCHEDULING: DISPATCHER

- Another component involved in the CPU scheduling function is the Dispatcher. The dispatcher is the module that gives control of the CPU to the process selected by the short-term scheduler. This function involves:
    - Switching context
    - Switching to user mode
    - Jumping to the proper location in the user program to restart that program from where it left last time.
- The dispatcher should be as fast as possible, since it is invoked during every process switch.
    - The time it takes for the dispatcher to stop one process and start another running is known as the **DISPATCH LATENCY.**

TYPES OF CPU SCHEDULING:

- CPU scheduling decisions may take place under the following four circumstances:
- When a process switches from the **runningstate** to the **waiting** state(for I/O request or invocation of wait for the termination of one of the child processes).
- When a process switches from the **running state** to the **ready state** (for example, when an interrupt occurs).
- When a process switches from the **waiting state** to the **ready state**(for example, completion of I/O).
- When a process terminates.

CPU SCHEDULING: 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.
  - Conceptually, CPU utilization can range from 0 to 100 percent.
  - In a real system, it should range from 40 percent (for a lightly loaded system) to 90 percent (for a heavily used system).
- **THROUGHPUT**
  - It is the total number of processes completed per unit time or rather say total amount of work done in a unit of time. This may range from 10/second to 1/hour depending on the specific processes**.**
- **TURNAROUND TIME**
  - It is the amount of time taken to execute a particular process, i.e. The interval from time of submission of the process to the time of completion of the process(Wall clock time).

- **WAITING TIME**
    - The sum of the periods spent waiting in the ready queue amount of time a process has been waiting in the ready queue to acquire get control on the CPU.
- **LOAD AVERAGE**
    - It is the average number of processes residing in the ready queue waiting for their turn to get into the CPU.
- **RESPONSE TIME**
    - Amount of time it takes from when a request was submitted until the first response is produced. Remember, it is the time till the first response and not the completion of process execution (final response).

SCHEDULING ALGORITHMS:

- CPU scheduling deals with the problem of deciding which of the processes in the ready queue is to be allocated the CPU.
- There are many different CPU scheduling algorithms:
    - First Come First Serve(FCFS) Scheduling
    - Shortest-Job-First(SJF) Scheduling
    - Priority Scheduling
    - Shortest Remaining Time (SRT) Scheduling
    - Round Robin(RR) Scheduling
    - Multilevel Queue Scheduling
    - Multilevel Feedback Queue Scheduling

FCFS - FIRST COME FIRST SERVE SCHEDULING:

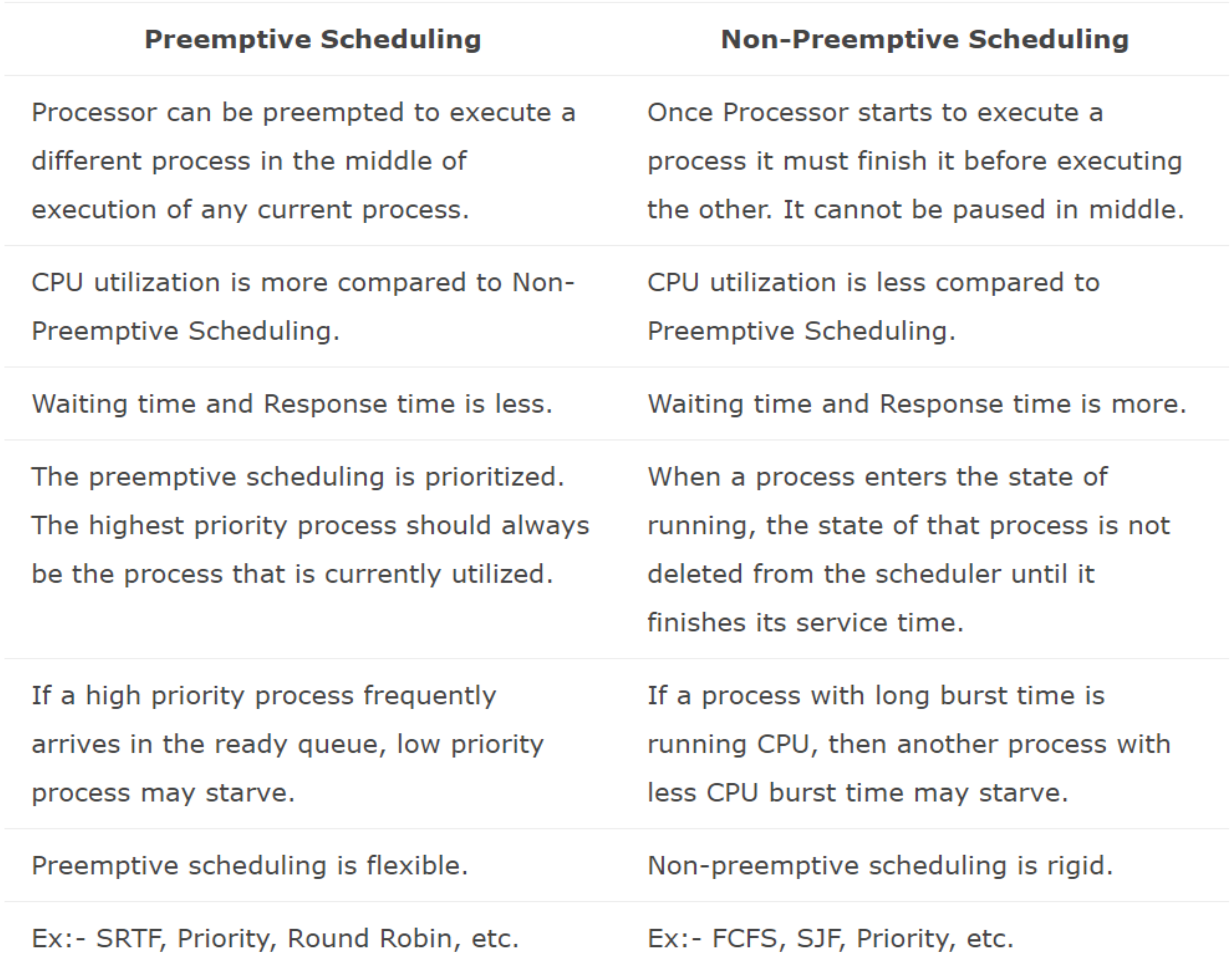
- In the "First come first serve" scheduling algorithm, as the name suggests, the process which arrives first, gets executed first, or we can say that the process which requests the CPU first, gets the CPU allocated first.
- Jobs are executed on first come, first serve basis.
- It is a non-preemptive, pre-emptive scheduling algorithm.
- Easy to understand and implement.
- Its implementation is based on FIFO queue.
- Poor in performance as average wait time is high.

- CHECK THE ANOTHER PDF FOR THE CALCULATION OF AVERAGE Turnaround Time – TAT & Waiting Time – WT

COMMON DEFINITIONS INVOLVED IN CALCULATION PROCESS

- **ARRIVAL TIME:** Time taken for the arrival of each process in the CPU Scheduling Queue.
- **COMPLETION TIME:** Time taken for the execution to complete, starting from arrival time.
- **TURN AROUND TIME:** Time taken to complete after arrival. In simple words, it is the difference between the Completion time and the Arrival time.
- **WAITING TIME:** Total time the process has to wait before it's execution begins. It is the difference between the Turn Around time and the Burst time of the process**.**

DIFFERENCE BETWEEN PREEMPTIVE AND NON – PREEMPTIVVE



Preemptive Scheduling	Non-Preemptive Scheduling
Processor can be preempted to execute a different process in the middle of execution of any current process.	Once Processor starts to execute a process it must finish it before executing the other. It cannot be paused in middle.
CPU utilization is more compared to Non-Preemptive Scheduling.	CPU utilization is less compared to Preemptive Scheduling.
Waiting time and Response time is less.	Waiting time and Response time is more.
The preemptive scheduling is prioritized. The highest priority process should always be the process that is currently utilized.	When a process enters the state of running, the state of that process is not deleted from the scheduler until it finishes its service time.
If a high priority process frequently arrives in the ready queue, low priority process may starve.	If a process with long burst time is running CPU, then another process with less CPU burst time may starve.
Preemptive scheduling is flexible.	Non-preemptive scheduling is rigid.
Ex:- SRTF, Priority, Round Robin, etc.	Ex:- FCFS, SJF, Priority, etc.

SJF – SHORTEST JOB FIRST SCHEDULING:

- This is also known as shortest job next, or SJN
- This is a non-preemptive, pre-emptive scheduling algorithm.
- Best approach to minimize waiting time.
- Easy to implement in Batch systems where required CPU time is known in advance.
- Impossible to implement in interactive systems where required CPU time is not known.
- The processer should know in advance how much time process will take.

PRIORITY BASED SCHEDULING:

- Priority scheduling is a non-preemptive algorithm and one of the most common scheduling algorithms in batch systems.
- 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 served basis.
- Priority can be decided based on memory requirements, time requirements or any other resource requirement**.**

SRT – SHORTEST REMAINING TIME SCHEDULING:

- Shortest remaining time (SRT) is the preemptive version of the SJN algorithm.
- The processor is allocated to the job closest to completion but it can be preempted by a newer ready job with shorter time to completion.
- Impossible to implement in interactive systems where required CPU time is not known.
- It is often used in batch environments where short jobs need to give preference.

ROUND ROBIN SCHEDULING:

- Round Robin is the preemptive process scheduling algorithm.
- Each process is provided a fix time to execute, it is called a quantum.
- Once a process is executed for a given time period, it is preempted and other process executes for a given time period.
- Context switching is used to save states of preempted processes.

MULTIPLE-LEVEL QUEUES SCHEDULING:

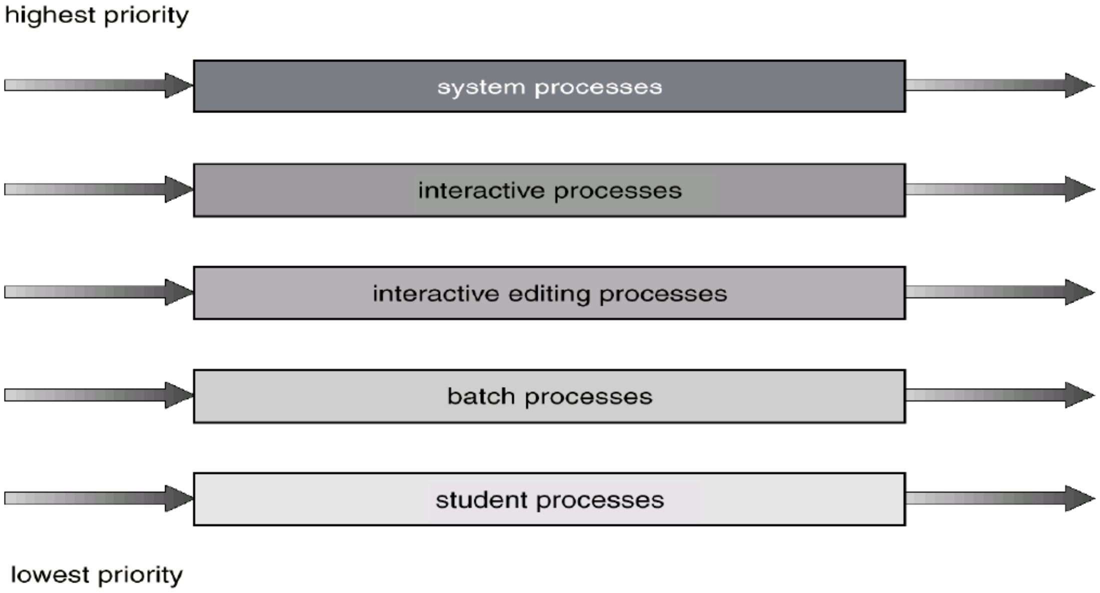
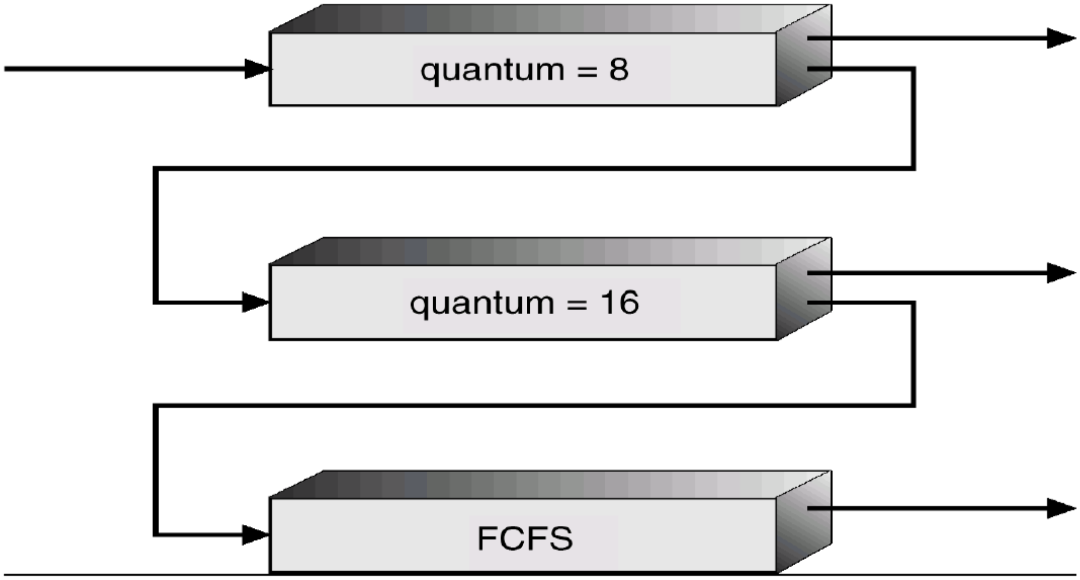


Fig:2.11 Multiple-Level Queue Scheduling

- Multiple-level queues are not an independent scheduling algorithm. They make use of other existing algorithms to group and schedule jobs with common characteristics.
- Multiple queues are maintained for processes with common characteristics.
- Each queue can have its own scheduling algorithms.
- Priorities are assigned to each queue**.**

MULTILEVEL FEEDBACK QUEUE SCHEDULING:

- Normally, in the multilevel queue scheduling algorithm, processes are permanently assigned to a queue when they enter to the system.
- The multilevel feedback-queue scheduling algorithm, in contrast, allows a process to move between queues.
- The idea is to separate processes according to the characteristics of their CPU bursts.
- If a process uses too much CPU time, it will be moved to a lower-priority queue.
- This scheme leaves I/O-bound and interactive processes in the higher-priority queues.
- Similarly, a process that waits too long in a lower-priority queue may be moved to a higher-priority queue. This form of **AGING** prevents **STARVATION.**



Multilevel feedback queues

Fig:2.12 Multilevel Feedback Queues

SATHYABAMA
INSTITUTE OF SCIENCE AND TECHNOLOGY
(DEEMED TO BE UNIVERSITY)
Accredited "A" Grade by NAAC | 12B Status by UGC | Approved by AICTE
www.sathyabama.ac.in

SCHOOL OF COMPUTING

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

UNIT – III – Operating System – SBS1206

UNIT 3

UNIT III	**(9 Hrs)**
Deadlocks: Characterization- Methods for Handling Deadlocks - Deadlock Prevention - Avoidance - Detection - Recovery.	

THE CRITICAL SECTION PROBLEM:

- A Critical Section is a code segment that accesses shared variables and has to be executed as an atomic action.
- It means that in a group of cooperating processes, at a given point of time, only one process must be executing its critical section.
- If any other process also wants to execute its critical section, it must wait until the first one finishes.

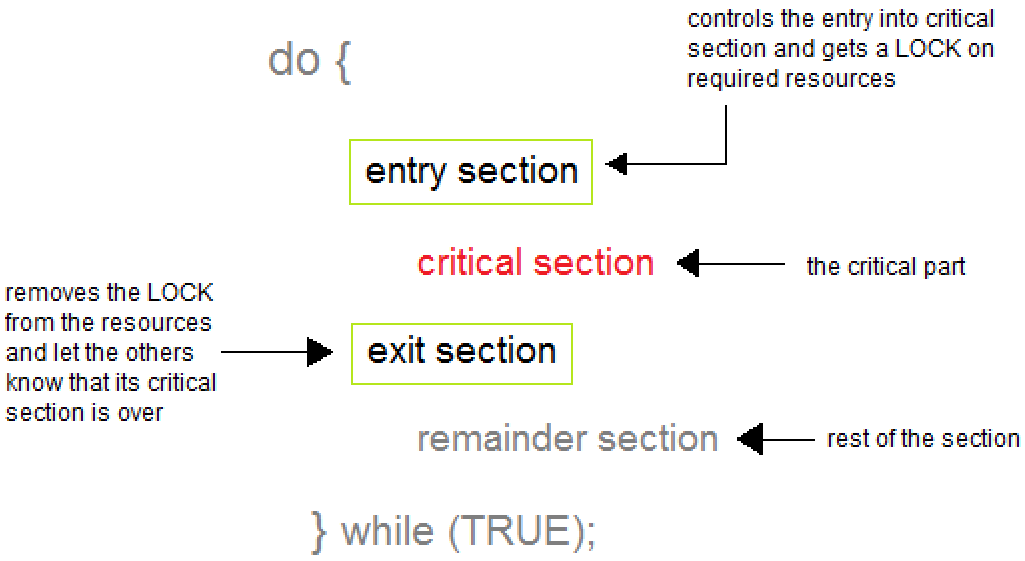

do {
controls the entry into critical section and gets a LOCK on required resources
entry section
critical section
the critical part
removes the LOCK from the resources and let the others know that its critical section is over
exit section
remainder section
rest of the section
} while (TRUE);

Fig:3.1 Critical section problem

THE SOLUTION TO THE CRITICAL SECTION PROBLEM:

The main solution for the critical section problem is based on the three main ways.

1. **MUTUAL EXCLUSION:**

    - Out of a group of cooperating processes, only one process can be in its critical section at a given point of time.

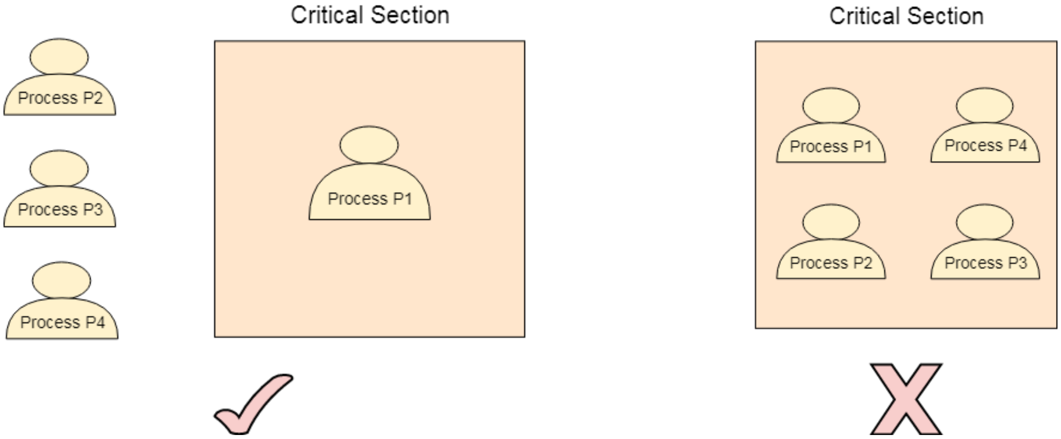


Fig:3.2 Mutual Exclusion

    - If process P1 is executing in its critical section, then no other processes can be executing in their critical sections.
    - IMPLICATIONS:
        - Critical sections better be focused and short.
        - Better not get into an infinite loop in there.
        - If a process somehow halts/waits in its critical section, it must not interfere with other processes.

2. **PROGRESS:**

    - If no process is in its critical section, and if one or more threads want to execute their critical section then any one of these threads must be allowed to get into its critical section.
    - If only one process wants to enter, it should be able to.
    - If two or more want to enter, one of them should succeed.

3. **BOUNDED WAITING:**

    - After a process makes a request for getting into its critical section, there is a limit for how many other processes can get into their critical section, before this process's request is granted.

- So after the limit is reached, system must grant the process permission to get into its critical section.

TWO PROCESS SOLUTIONS:

ALGORITHM 1:

```
do {
while (turn != 1);
critical section
turn = j;
remainder section
}while (1);
```

This Algorithm satisfies Mutual exclusion whereas it fails to satisfy progress requirement since it requires strict alternation of processes in the execution of the critical section.

For example, if turn == 0 and p1 is ready to enter its critical section,p1 cannot do so, even though p0 may be in its remainder section.

ALGORITHM 2:

```
do{
flag[i] =true;
while (flag[j]);
critical section
flag[i]= false;
remainder section

}while(1);
```

In this solution, the mutual exclusion is met. But the progress is not met. To illustrate this problem, we consider the following

Execution Sequence:

To: P0 sets Flag[0] = true

T1: P1 sets Flag[1] = true

Now P0 and P1 are looping forever in their respective while statements.

ALGORITHM 3:

By combining the key ideas of algorithm 1 and 2, we obtain a correct solution.

```
Do{

Flag[i] = true;

Turn = j;

While (flag[j] && turn == j);

Critical section

Flag[i] = false;

Remainder section

}while(1);
```

The algorithm does satisfy the three essential criteria to solve the critical section problem. The three criteria are mutual exclusion, progress, and bounded waiting.

SYNCHRONIZATION HARDWARE:

- Many systems provide hardware support for critical section code.

- The critical section problem could be solved easily in a single-processor environment if we could disallow interrupts to occur while a shared variable or resource is being modified.
- In this manner, we could be sure that the current sequence of instructions would be allowed to execute in order without pre-emption. Unfortunately, this solution is not feasible in a multiprocessor environment.
- Disabling interrupt on a multiprocessor environment can be time consuming as the message is passed to all the processors.

MUTEX LOCKS:

- As the synchronization hardware solution is not easy to implement for everyone, a strict software approach called Mutex Locks.

```
do{
Acquire Lock
Critical Section
Release Lock
Remainder Section
}while(TRUE);
```

MUTEX:

- Mutex is a program object that allows multiple program threads to share the same resource, such as file access, but not simultaneously.
- When a program is started a mutex is created with a unique name.
- After this stage, any thread that needs the resource must lock the mutex from other threads while it is using the resource.
- The mutex is set to unlock when the data is no longer needed or the routine is finished.

SOLUTION TO THE CRITICAL - SECTION PROBLEM USING LOCKS

- Hardware features can make any programming task easier and improve system efficiency.
- The critical section problem can be solved simply in a uniprocessor environment if we could prevent interrupts from occurring while a shared variable was being modified.
- Then we can ensure that the current sequence of instructions would be allowed to execute in order without pre-emption.

- No other instructions would be run, so no unexpected modifications could be made to the shared variable.
- This solution is not feasible in a multiprocessor environment.
- Disabling interrupts on a multiprocessor can be time consuming, as the message is passed to all the processors.

- **Definition of TestAndSet:**

```
boolean TestAndSet (boolean *target)
{
 boolean rv = *target;
*target = TRUE;
 return rv:
 }
```

- **Shared boolean variable lock., initialized to false.**
- **Solution:**

```
 do {
while ( TestAndSet (&lock ))
; /* do nothing
// critical section
lock = FALSE;
// remainder section
} while ( TRUE);
```

- **Solution using Swap:**
- **Definition of Swap:**

```
void Swap (boolean *a, boolean *b) {
boolean temp = *a;

*a = *b;

*b = temp:

}
```

- **Shared Boolean variable lock initialized to FALSE; Each process has a local Boolean variable key.**
- **Solution:**

```
do {
 key = TRUE;
 while  ( key == TRUE)
 Swap (&lock,  &key );
 // critical  section
 lock  = FALSE;
 // remainder  section
 } while  ( TRUE);
```

- **Solution with TestAndSet and bounded wait**
- **boolean waiting[n]; boolean lock; initialized to false Pi can enter critical section iff waiting[i] == false or key == false**

```
do {
waiting[i]  = TRUE;
key = TRUE;
while  (waiting[i]  && key)
key = TestAndSet  (&lock);
waiting[i]  = FALSE;
// critical  section
j = (i + 1) % n;
while  ((j != i) && !waiting[j])
j = (j + 1) % n;
if (j == i)
lock  = FALSE;
else
waiting[j]  = FALSE;
// remainder  section
} while  (TRUE);
```

SEMAPHORES

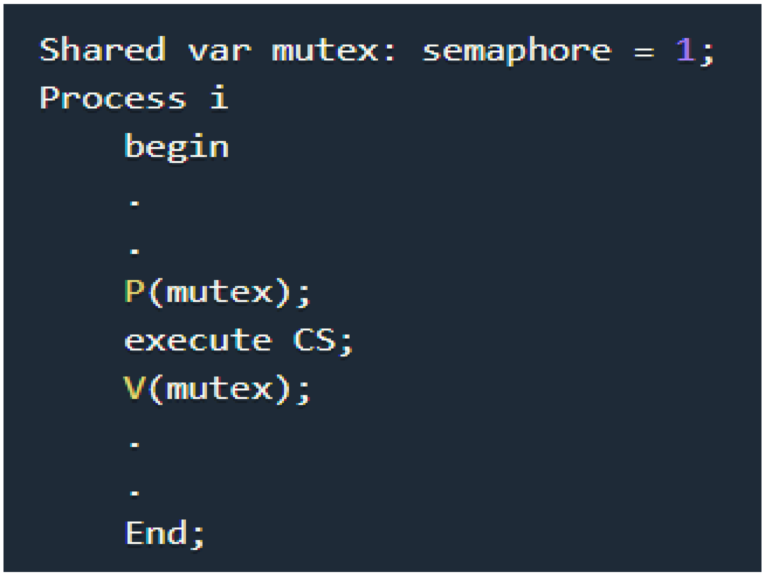
- It’s a very significant technique for managing concurrent processes by using the value of a simple integer variable to synchronize the progress of interacting processes. This integer variable is called semaphore.
- In very simple words, semaphore is a variable which can hold only a non-negative Integer value, shared between all the threads, with operations wait and signal,
    - P(S): if S ≥ 1 then S := S - 1
      else <block and enqueue the process>;
    - V(S): if <some process is blocked on the queue>
      then <unblock a process>
      else S := S + 1;
- Wait: Decrements the value of its argument S, as soon as it would become non-negative (greater than or equal to 1).
- Signal: Increments the value of its argument S, as there is no more process blocked on the queue.

PROPERTIES OF SEMAPHORES

- It's simple and always have a non-negative Integer value.
- Works with many processes.
- Can have many different critical sections with different semaphores.
- Each critical section has unique access semaphores.
- Can permit multiple processes into the critical section at once, if desirable.

TYPES OF SEMAPHORES

- **BINARY SEMAPHORE:**
    - It is a special form of semaphore used for implementing mutual exclusion, hence it is often called a **MUTEX**.
    - A binary semaphore is initialized to 1 and only takes the values 0 and 1 during execution of a program.
- **COUNTING SEMAPHORES:**
    - These are used to implement bounded concurrency.
- **EXAMPLE:**



```
Shared var mutex: semaphore = 1;
Process i
    begin
    .
    .
    P(mutex);
    execute CS;
    V(mutex);
    .
    .
    End;
```

LIMITATIONS OF SEMAPHORES

- Priority Inversion is a big limitation of semaphores.
- Their use is not enforced, but is by convention only.
- With improper use, a process may block indefinitely. Such a situation is called Deadlock. We will be studying deadlocks in details in coming lessons.

BINARY SEMAPHORE:

USAGE:

- OS's distinguish between counting and binary semaphores.
- The value of a counting semaphore can range over an unrestricted domain.
- The value of a binary semaphore can range only between 0 and 1.
- Binary semaphores are known as mutex locks as they are locks that provide mutual exclusion.
- Binary semaphores are used to deal with the critical section problem for multiple processes.
- Counting semaphores can be used to control access to a given resource consisting of a finite number of instances.
- The semaphore is initialized to the number of resources available.
- Each process that wishes to use a resource performs a wait() operation on the semaphore.
- When a process releases a resource, it performs a signal() operation.

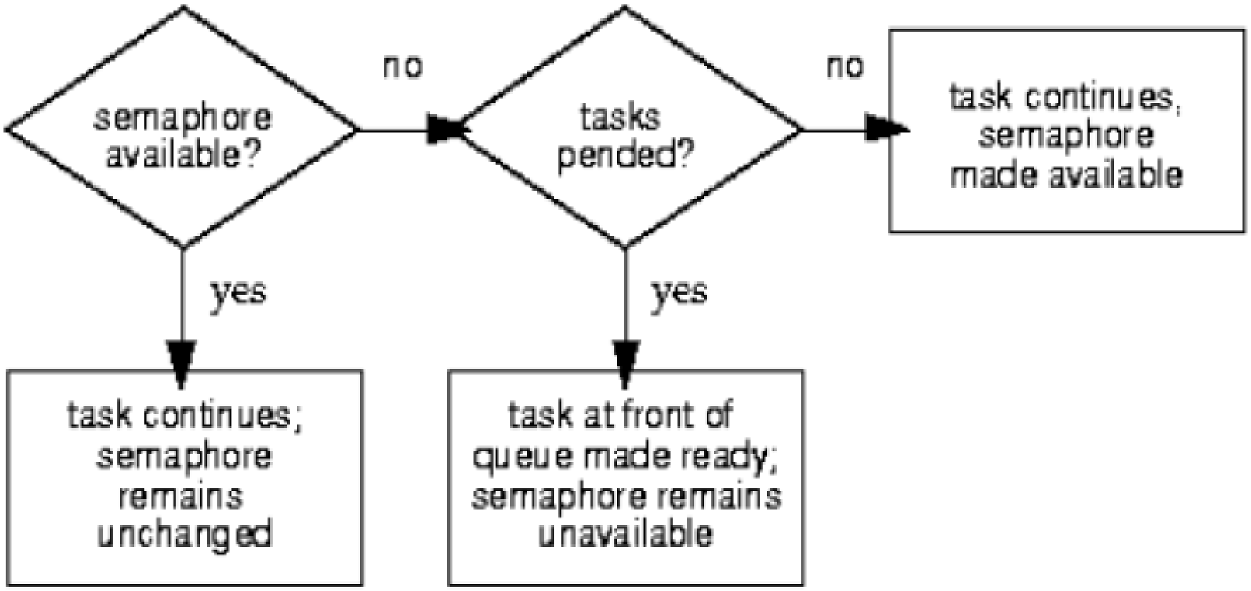

semaphore available?
no
tasks pended?
no
task continues, semaphore made available
yes
yes
task continues; semaphore remains unchanged
task at front of queue made ready; semaphore remains unavailable

Fig:3.3 Binary Semaphore Usage

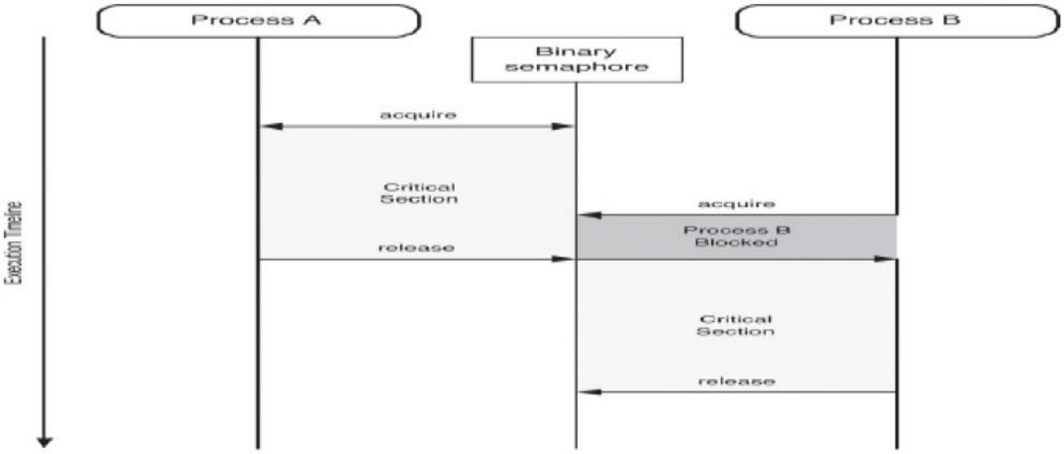


Fig:3.4 Binary Semaphore

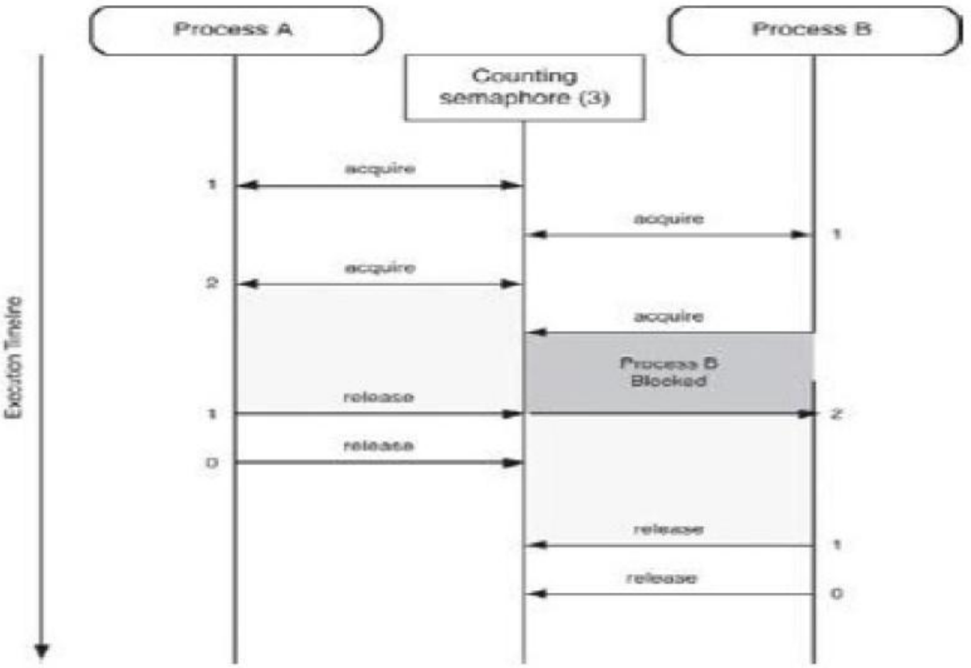
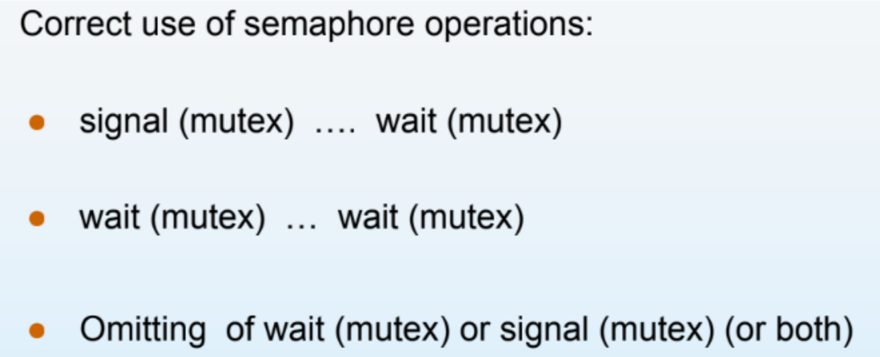


Fig:3.5 Counting Semaphore

IMPLEMENTATION

- The main disadvantage of the semaphore is that it requires busy waiting. While a process is in its critical section, any other process that tries to enter its critical section must loop continuously in the entry code.
- Busy waiting wastes CPU cycles that some other process might be able to use productively. This type of semaphore is called a **SPINLOCK** because the process spins while waiting for the lock.
- To overcome, the need for busy waiting the definition of wait () and signal() semaphore operations can be modified.
- When a process executes the wait() operation and finds that the semaphore value is not positive, it must wait.

PROBLEMS WITH SEMAPHORE:



Correct use of semaphore operations:

- signal (mutex) …. wait (mutex)
- wait (mutex) … wait (mutex)
- Omitting of wait (mutex) or signal (mutex) (or both)

DEADLOCKS AND STARVATION:

DEADLOCK:

- Deadlocks are a set of blocked processes each holding a resource and waiting to acquire a resource held by another process.

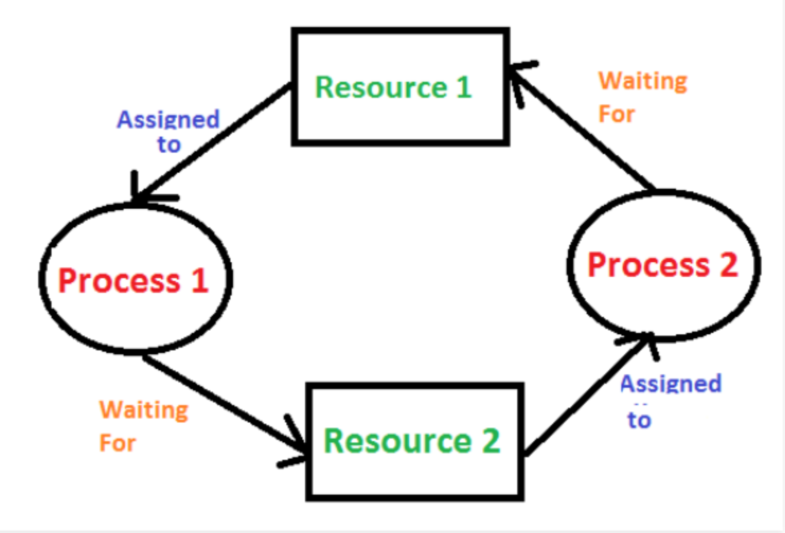


Fig:3.6 Deadlocks

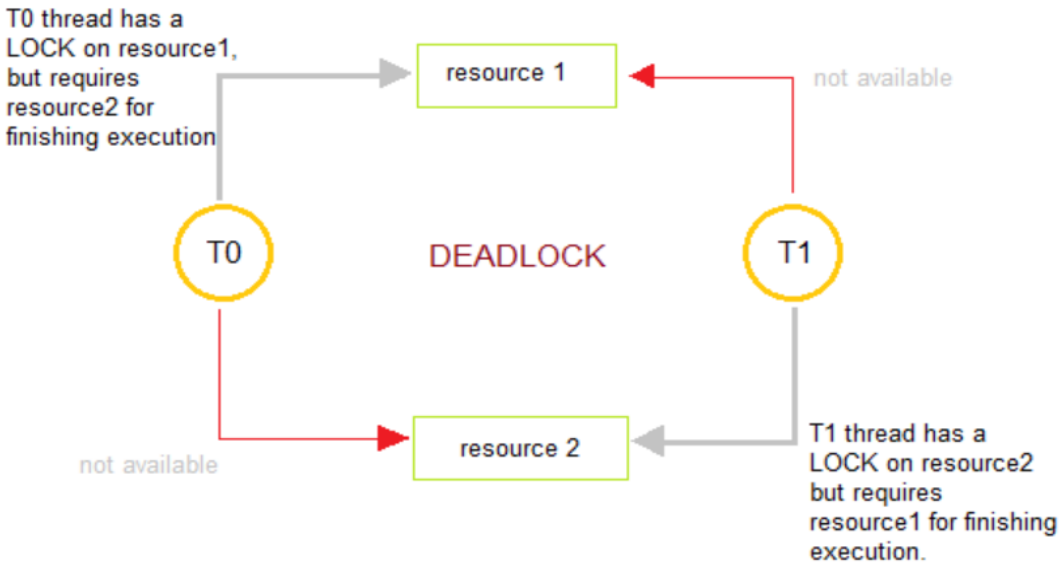

T0 thread has a LOCK on resource1, but requires resource2 for finishing execution
resource 1
not available
T0
DEADLOCK
T1
not available
resource 2
T1 thread has a LOCK on resource2 but requires resource1 for finishing execution.

Fig:3.7 Deadlock Example with T0 and T1 Thread

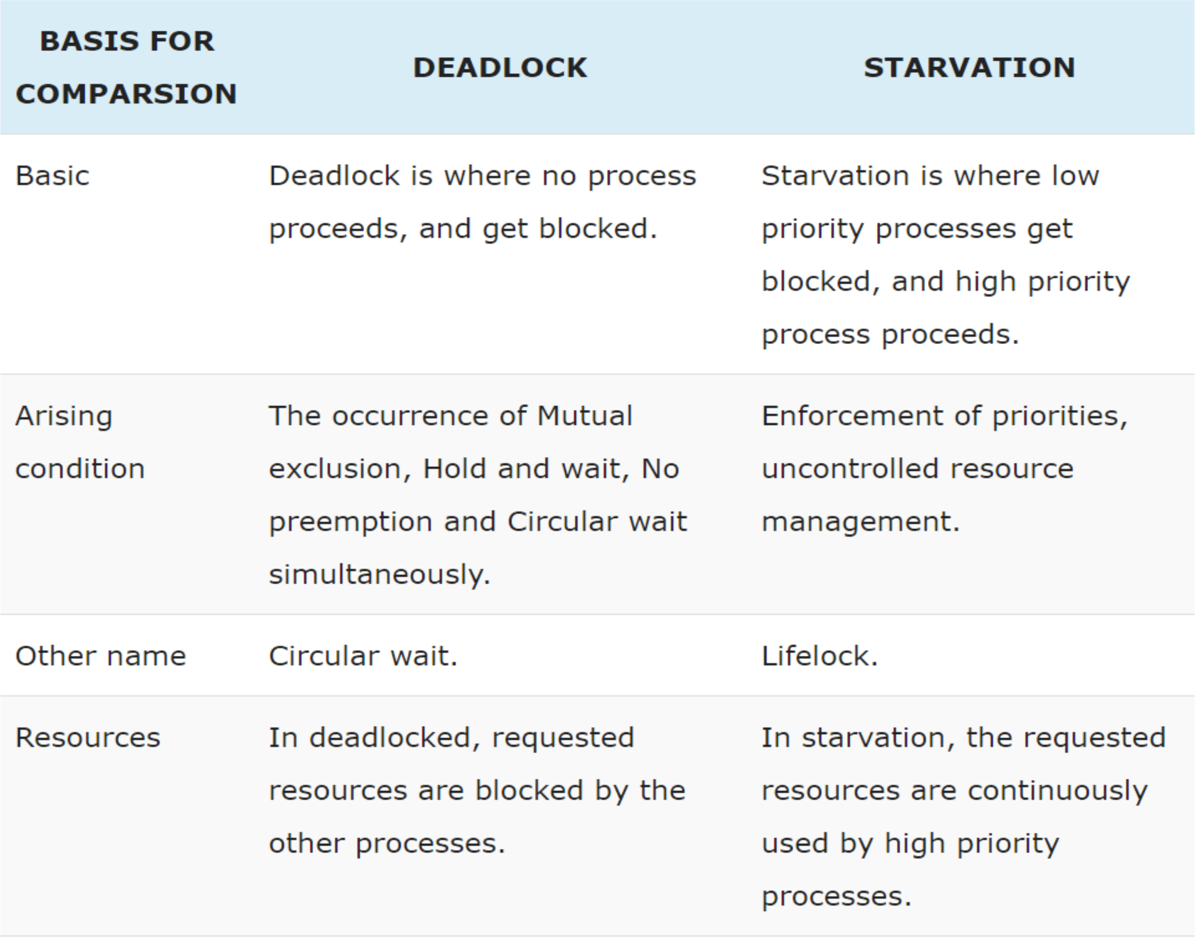
STARVATION

- Starvation, process with high priorities continuously uses the resources preventing low priority process to acquire the resources.
- Starvation can be defined as when a process request for a resource and that resource has been continuously used by the other processes then the requesting process faces starvation.
- In starvation, a process ready to execute waits for CPU to allocate the resource. But the process has to wait indefinitely as the other processes continuously block the requested resources.

AGING

- Aging can resolve the problem of starvation. Aging gradually increases the priority of the process that has been waiting long for the resources. Aging prevents a process with low priority to wait indefinitely for a resource.

DIFFERENCE BETWEEN DEADLOCKS & STARVATION



BASIS FOR COMPARSION	DEADLOCK	STARVATION
Basic	Deadlock is where no process proceeds, and get blocked.	Starvation is where low priority processes get blocked, and high priority process proceeds.
Arising condition	The occurrence of Mutual exclusion, Hold and wait, No preemption and Circular wait simultaneously.	Enforcement of priorities, uncontrolled resource management.
Other name	Circular wait.	Lifelock.
Resources	In deadlocked, requested resources are blocked by the other processes.	In starvation, the requested resources are continuously used by high priority processes.

CLASSIC PROBLEMS OF SYNCHRONIZATION

- These synchronization problems are examples of large class of concurrency control problems. In solutions to these problems, we use semaphores for synchronization.
    - **Bounded buffer problem / Producer consumer problem**
    - **Readers-writer problem**
    - **Dining-philosophers problem**

BOUNDED BUFFER / PRODUCER CONSUMER PROBLEM

- This problem is generalized in terms of the Producer Consumer problem, where a finite buffer pool is used to exchange messages between producer and consumer processes.

- Because the buffer pool has a maximum size, this problem is often called the Bounded buffer problem.
- Solution to this problem is, creating two counting semaphores "full" and "empty" to keep track of the current number of full and empty buffers respectively.
- The code below can be interpreted as the producer producing full buffers for the consumer or as the consumer producing empty buffers for the producer.

```
do
{
// produce an item in nextp
wait(empty);
wait(mutex);
// add the item to the buffer
signal(mutex);
signal(full);
}while(TRUE);
```

- The structure of the Producer Process

```
do
{
wait(full);
wait(mutex);
// remove an item from buffer into nextc
signal(mutex);
signal(empty);
// consume the item in nextc
}while(TRUE);
```

- The structure of the Consumer Process

EXAMPLE CODE:

```
#include<stdio.h>
#include<conio.h>
int main()
{
int s,n,b=0,p=0,c=0;
clrscr();
printf("\n producer and consumer problem");
do
{
printf("\n menu");
printf("\n 1.producer an item");
printf("\n 2.consumer an item");
printf("\n 3.add item to the buffer");
printf("\n 4.display status");
printf("\n 5.exit");
printf("\n enter the choice");
scanf("%d",&s);
switch(s)
{
case 1:
p=p+1;
printf("\n item to be produced");
break;
case 2:
if(b!=0)
{
```

```
c=c+1;

b=b-1;

printf("\n  item to be consumed");

}

else

{

printf("\n  the buffer  is empty please wait...");

}

break;

case 3:

if(b<n)

{

if(p!=0)

{

b=b+1;

printf("\n  item added to buffer");

}

else

printf("\n  no.of items  to add...");

}

else

printf("\n  buffer  is  full,please  wait");

break;

case 4:

printf("no.of  items  produced :%d",p);

printf("\n  no.of consumed  items:%d",c);

printf("\n  no.of buffered  item:%d",b);

break;

case 5:exit(0);
```

```
}
}
while(s<=5);
getch();
return 0;
}
```

THE READERS WRITERS PROBLEM

- In this problem there are some processes(called readers) that only read the shared data, and never change it, and there are other processes(called writers) who may change the data in addition to reading, or instead of reading it.
- There are various type of readers-writers problem, most centered on relative priorities of readers and writers.

SOLUTION:

- From the above problem statement, it is evident that readers have higher priority than writer. If a writer wants to write to the resource, it must wait until there are no readers currently accessing that resource.
- Here, we use one mutex m and a semaphore w. An integer variable **sread** is used to maintain the number of readers currently accessing the resource. The variable **swrite** is initialized to 0. A value of 1 is given initially to m and w.
- Instead of having the process to acquire lock on the shared resource, we use the mutex m to make the process to acquire and release lock whenever it is updating the **sread** variable.

EXAMPLE CODE

```
#include<stdio.h>
#include<conio.h>
#include<process.h>
void main()
{
typedef int semaphore;
semaphore sread=0, swrite=0;
```

```
int ch,r=0;
clrscr();
printf("\nReader  writer");
do
{
printf("\nMenu");
printf("\n\t  1.Read from file");
printf("\n  \t2.Write to file");
printf("\n  \t 3.Exit the reader");
printf("\n  \t 4.Exit the writer");
printf("\n  \t 5.Exit");
printf("\nEnter  your choice:");
scanf("%d",&ch);
switch(ch)
{
case 1: if(swrite==0)
{
sread=1;
r+=1;
printf("\nReader  %d reads",r);
}
else
{
printf("\n  Not possible");
}
break;
case 2: if(sread==0  && swrite==0)
{
swrite=1;
printf("\nWriter in Progress");
}
else if(swrite==1)
{
printf("\nWriter writes the files");
}
else if(sread==1)
{
printf("\nCannot write while reader reads the file");
}
else
printf("\nCannot write file");
break;
case 3: if(r!=0)
{
printf("\n The reader %d closes the file",r);
r-=1;
}
else if(r==0)
{
printf("\n Currently no readers access the file");
sread=0;
}
else if(r==1)
{
printf("\nOnly 1 reader file");
}
else
printf("%d reader are reading the file\n",r);
```

```
break;
case 4: if (swrite==1)
{
printf("\nWriter close the file");
swrite=0;
}
else
printf("\nThere is no writer in the file");
break;
case 5: exit(0);
}
}
while(ch<6);
getch();
}
```

DINING PHILOSOPHERS PROBLEM:

- The dining philosopher's problem involves the allocation of limited resources to a group of processes in a deadlock-free and starvation-free manner.
- There are five philosophers sitting around a table, in which there are five chopsticks/forks kept beside them and a bowl of rice in the center.
- When a philosopher wants to eat, he uses two chopsticks - one from their left and one from their right.
- When a philosopher wants to think, he keeps down both chopsticks at their original place.

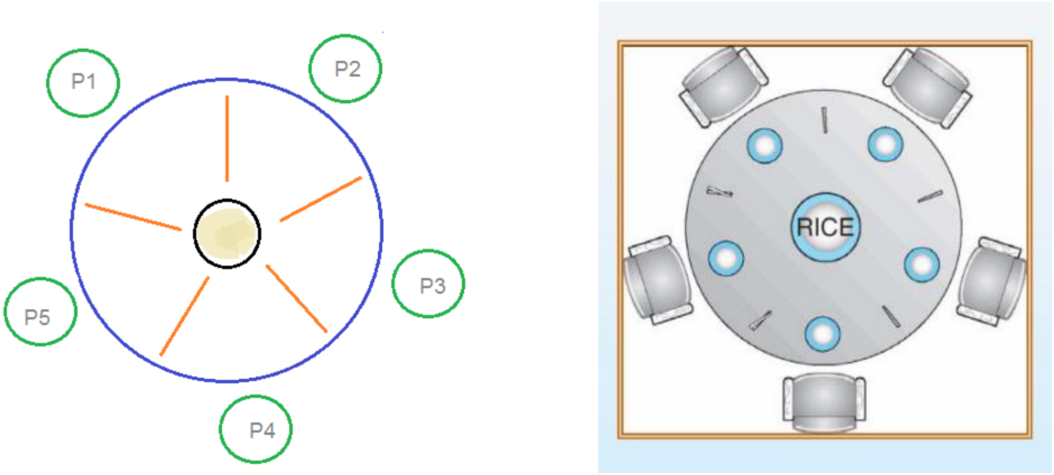


Fig:3.8 Dining Philosophers Problem

SOLUTION:

- From the problem statement, it is clear that a philosopher can think for an indefinite amount of time.
- But when a philosopher starts eating, he has to stop at some point of time.
- The philosopher is in an endless cycle of thinking and eating.
- An array of five semaphores, stick[5], for each of the five chopsticks.

EXAMPLE CODE:

```
#include<stdio.h>

#include<conio.h>

#define LEFT (i+4) %5

#define RIGHT (i+1) %5

#define THINKING 0

#define HUNGRY 1

#define EATING 2

int state[5];

void put_forks(int);

void test(int);
void take_forks(int);
void philosopher(int i)
{
if(state[i]==0)
{
take_forks(i);
if(state[i]==EATING)
printf("\n Eating in process....");
put_forks(i);
}
}
void put_forks(int i)
{
state[i]=THINKING;
printf("\n philosopher %d completed its works",i);
test(LEFT);
test(RIGHT);
}
void take_forks(int i)
{
state[i]=HUNGRY;
test(i);
}
void test(int i)
{
if(state[i]==HUNGRY && state[LEFT]!=EATING && state[RIGHT]!=EATING)
{
printf("\n philosopher %d can eat",i);
state[i]=EATING;
}
}
void main()
{
int i;
clrscr();
for(i=1;i<=5;i++)
state[i]=0;
printf("\n\t\t\t Dining Philosopher Problem");
printf("\n\t\t..........");
for(i=1;i<=5;i++)
{
printf("\n\n the philosopher %d falls hungry\n",i);
philosopher(i);
}
getch();
}
```

CRITICAL REGIONS:

- A critical region is a section of code that is always executed under mutual exclusion.
- Critical regions shift the responsibility for enforcing mutual exclusion from the programmer (where it resides when semaphores are used) to the compiler.
- They consist of two parts:
  - Variables that must be accessed under mutual exclusion.
  - A new language statement that identifies a critical region in which the variables are accessed.

EXAMPLE

```
var
v : shared T;
...
region v do
```

```
begin
...
end;
```

- All critical regions that are ‘tagged’ with the same variable have compiler-enforced mutual.
- Exclusion so that only one of them can be executed at a time:

```
Process A:
region V1 do
begin
{ Do some stuff. }
end;
region V2 do
begin
{ Do more stuff. }
end;

Process B:
region V1 do
begin
{ Do other stuff. }
end;
```

- Here process A can be executing inside its V2 region while process B is executing inside its V1 region, but if they both want to execute inside their respective V1 regions only one will be permitted to proceed.
- Each shared variable (V1 and V2 above) has a queue associated with it. Once one process is executing code inside a region tagged with a shared variable, any other processes that attempt to enter a region tagged with the same variable are blocked and put in the queue.

CONDITIONAL CRITICAL REGIONS:

- Critical regions aren’t equivalent to semaphores.
- As described so far, they lack condition synchronization.
- We can use semaphores to put a process to sleep until some condition is met (e.g. see the bounded-buffer Producer-Consumer problem), but we can’t do this with critical regions.

- Conditional critical regions provide condition synchronization for critical regions

```
region v when B do
begin
...
end;
```

where B is a boolean expression (usually B will refer to v).

- Conditional critical regions work as follows:
  - A process wanting to enter a region for v must obtain the mutex lock. If it cannot, then it is queued.
  - Once the lock is obtained the boolean expression B is tested. If B evaluates to true then the process proceeds, otherwise it releases the lock and is queued. When it next gets the lock it must retest B.

IMPLEMENTATION:

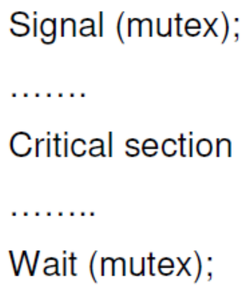
- Each shared variable now has two queues associated with it.
- The **MAIN QUEUE** is for processes that want to enter a critical region but find it locked.
- The **EVENT QUEUE** is for the processes that have blocked because they found the condition to be false.
- When a process leaves the conditional critical region the processes on the event queue join those in the main queue.

LIMITATIONS:

- Conditional critical regions are still distributed among the program code.
- There is no control over the manipulation of the protected variables — no information hiding or encapsulation.
- Once a process is executing inside a critical region it can do whatever it likes to the variables it has exclusive access to.
- Conditional critical regions are more difficult to implement efficiently than semaphores.

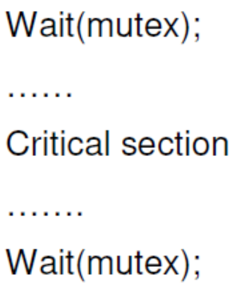
MONITORS:

- Consist of private data and operations on that data.
- Can contain types, constants, variables and procedures.
- Only the procedures explicitly marked can be seen outside the monitor.
- The monitor body allows the private data to be initialized.
- The compiler enforces mutual exclusion on a particular monitor.
- Each monitor has a boundary queue, and processes wanting to call a monitor routine join this queue if the monitor is already in use.
- Monitors are an improvement over conditional critical regions because all the code that accesses the shared data is localized.



```
Signal (mutex);
…….
Critical section
……..
Wait (mutex);
```

- Although semaphores provide a convenient and effective mechanism for process synchronization, using them incorrectly can result in timing errors that are difficult to detect since these errors happen only if some particular execution sequences take place and these sequences do not always occur.
- Suppose that a process interchanges the order in which the wait () and signal () operations on the semaphore mutex are executed.
- Here several processes may be executing in their critical sections simultaneously, violating the mutual exclusion requirement.
- Suppose that a process replaces signal (mutex) with wait (mutex) that is it executes



```
Wait(mutex);
......
Critical section
.......
Wait(mutex);
```

- Here a deadlock will occur.
- Suppose that a process omits the wait(mutex), or the signal(mutex) or both. Here, either mutual exclusion is violated or a dead lock will occur.
- To deal with such errors, a fundamental high level synchronization construct called **MONITORS** type is used.

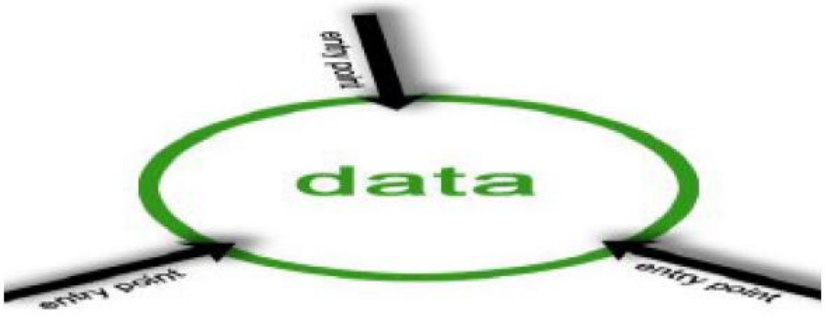
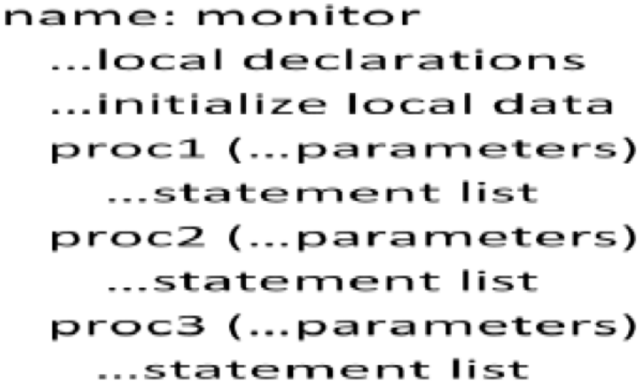


Fig:3.9 Monitors

USAGE OF MONITORS:

- A monitor type presents a set of programmer defined operations that are provided mutual exclusion within the monitor.
- The monitor type also contains the declaration of variables whose values define the state of an instance of that type, along with the bodies of the procedures or functions that operate on those variables.
- The representation of a monitor type cannot be used directly by the various processes.
- Thus, a procedure defined within a monitor can access only those variables declared locally within the monitor and its formal parameters.

- The local variables of a monitor can be accessed by only the local procedures.



```
name: monitor
  ...local declarations
  ...initialize local data
  proc1 (...parameters)
    ...statement list
  proc2 (...parameters)
    ...statement list
  proc3 (...parameters)
    ...statement list
```

- When x.signal() operation is invoked by a process P, there is a suspended process Q associated with condition x.
- If suspended process Q is allowed to resume its execution, the signaling process P must wait.
- Otherwise, both P and Q would be active simultaneously within the monitor.

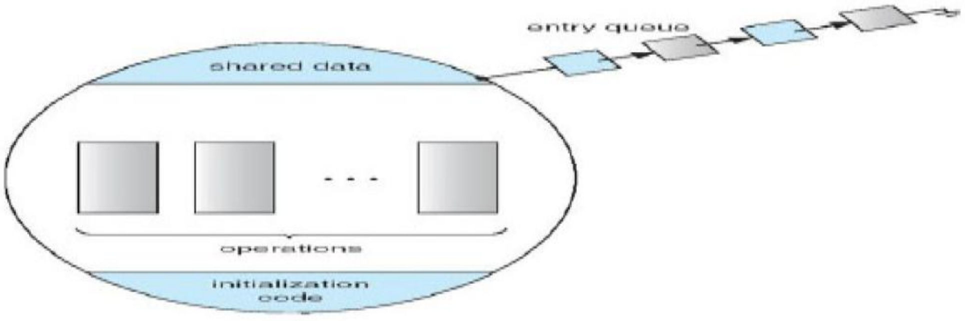


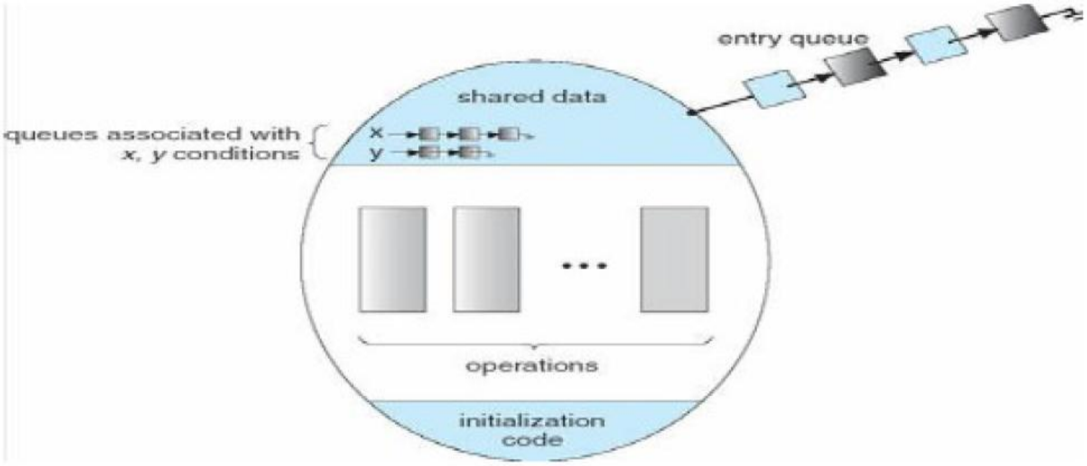
Fig:3.10 Monitors with shared data

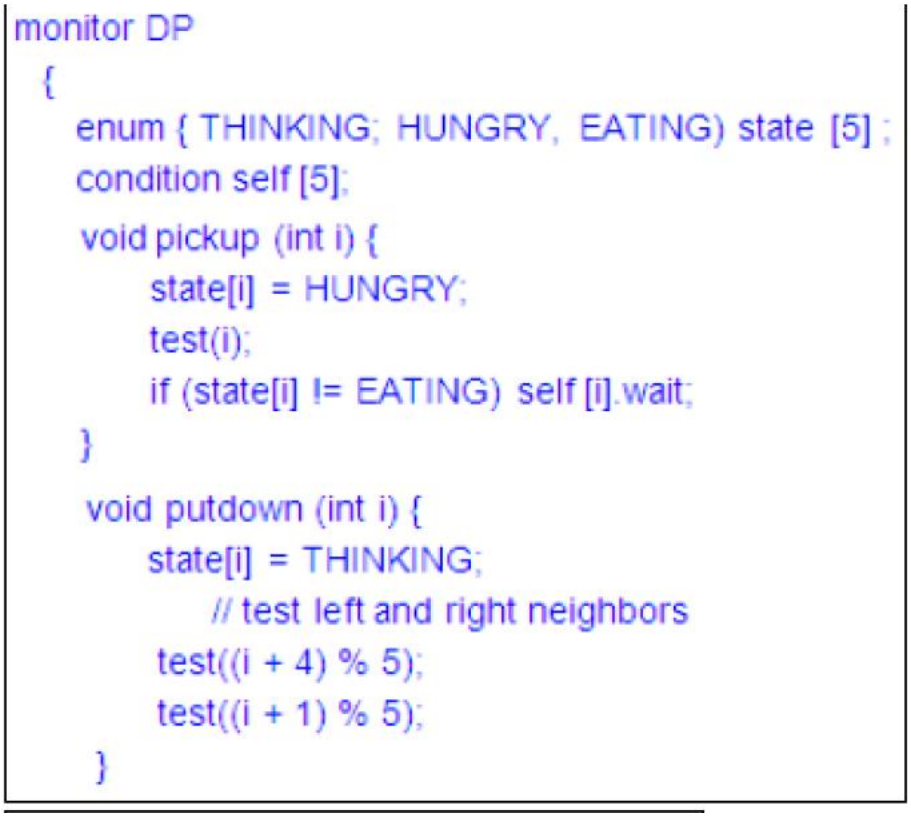
- However, both processes can conceptually continue with their execution. Two possibilities exist:
    - **Signal and wait** – P either waits until Q leaves the monitor or waits for another condition.
    - **Signal and condition** – Q either waits until P leaves the monitor or waits for another condition.

DINING PHILOSOPHERS SOLUTION USING MONITORS:

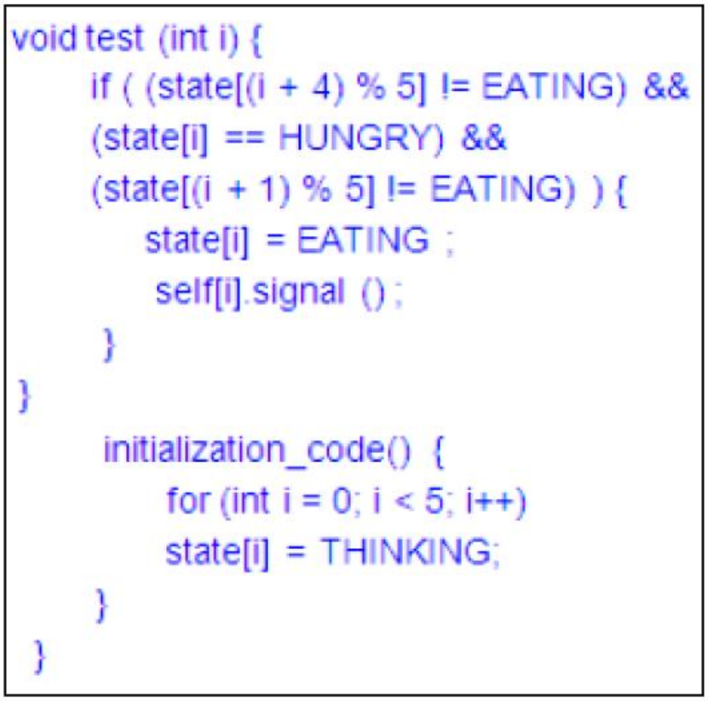
- This solution imposes the restriction that a philosopher may pick up her chopsticks only if both of them are available.
- To code this solution, we need to distinguish among three states in which we may find a philosopher.
- For this purpose, we use this data structure **enum {thinking, hungry, eating } state[5];**

enum {thinking, hungry, eating } state[5];





```
monitor DP
  {
     enum { THINKING; HUNGRY, EATING) state [5] ;
     condition self [5];
     void pickup (int i) {
          state[i] = HUNGRY;
          test(i);
          if (state[i] != EATING) self [i].wait;
     }
      void putdown (int i) {
          state[i] = THINKING;
               // test left and right neighbors
           test((i + 4) % 5);
           test((i + 1) % 5);
      }
```

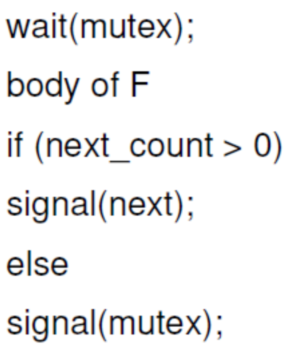


```
void test (int i) {
      if ( (state[(i + 4) % 5] != EATING) &&
      (state[i] == HUNGRY) &&
      (state[(i + 1) % 5] != EATING) ) {
          state[i] = EATING ;
           self[i].signal () ;
      }
}
       initialization_code() {
           for (int i = 0; i < 5; i++)
           state[i] = THINKING;
      }
 }
```

Monitor solution to Dining Philosophers problem

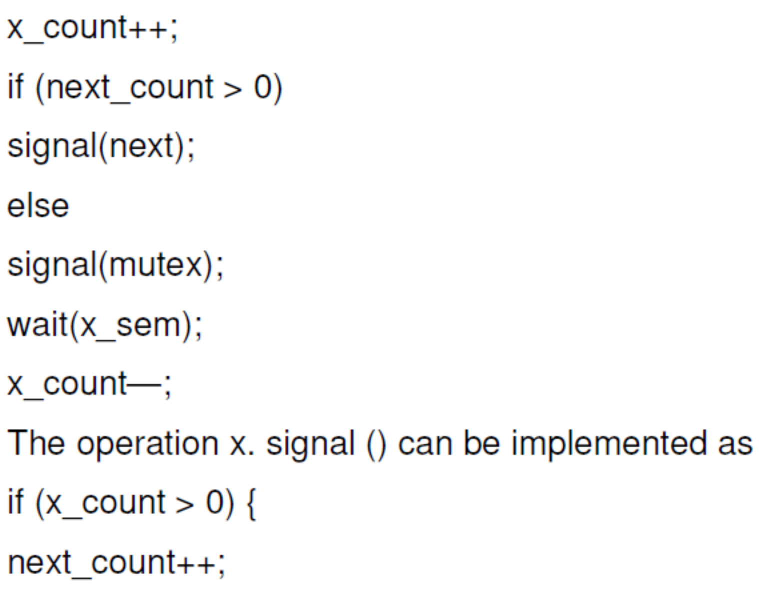
IMPLEMENTING A MONITOR USING SEMAPHORES

- For each monitor, a semaphore mutex initialized to 1 is provided.
- A process must execute wait(mutex) before entering the monitor and must execute(signal) after leaving the monitor.
- Since a signaling process must wait until the resumed process either leaves or waits, an additional semaphore next initialized to 0, on which the signaling processes may suspend themselves.
- An integer variable next_count is used to count the number of processes suspended on next.



```
wait(mutex);
body of F
if (next_count > 0)
signal(next);
else
signal(mutex);
```

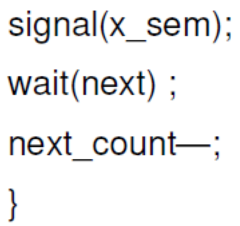
- We can now describe how condition variables are implemented.
- For each condition x, we introduce a semaphore x_sem and an integer variable x_count, both initialized to 0.
- The operation x. wait () can now be implemented as



```
x_count++;
if (next_count > 0)
signal(next);
else
signal(mutex);
wait(x_sem);
x_count—;
```

The operation x. signal () can be implemented as

```
if (x_count > 0) {
next_count++;
```



```
signal(x_sem);
wait(next) ;
next_count—;
}
```

RESUMING PROCESSES WITHIN A MONITOR

- If several processes are suspended on condition x, and an x.signal() operation is executed by some process, then for determining which suspended process should be resumed next, we use FCFS ordering so that the process waiting the longest is resumed first.
- Or conditional wait construct() can be used as x.wait(c); where c is an integer expression that is evaluated when the wait () operation is executed.
- The value of c, which is called a **PRIORITY NUMBER**, is then stored with the name of the process that is suspended.
- When x. signal () is executed, the process with the smallest associated priority number is resumed next.

A MONITOR TO ALLOCATE A SINGLE RESOURCE

```
monitor ResourceAllocator
boolean busy;
condition x;
void acquire(int time)
if (busy)
x.wait(time);
busy = TRUE;
void release() {
busy = FALSE;
x.signal();
initialization_code
busy = FALSE;
```

- A process that needs to access the resource in question must observe the following sequence:

```
R.acquire(t);
access the resource;
R. release O ;
```

- where R is an instance of type Resource Allocator.
- Unfortunately, the monitor concept cannot guarantee that the preceding access sequence will be observed. In particular, the following problems can occur:
  - A process might access a resource without first gaining access permission to the resource.
  - A process might never release a resource once it has been granted access to the resource.
- **Synchronization Example**
  - A process might attempt to release a resource that it never request
  - A process might request the same resource twice (without first releasing the resource).

DEADLOCKS:

- The Deadlock Problem
- System Model
- Deadlock Characterization
- Methods for Handling Deadlocks
- Deadlock Prevention
- Deadlock Avoidance
- Deadlock Detection
- Recovery from Deadlock

THE DEADLOCK PROBLEM

- A set of blocked processes each holding a resource and waiting to acquire a resource held by another process in the set.
- **Example**
  - System has 2 tape drives.
  - P0 and P1 each hold one tape drive and each needs another one.
- Example
  - semaphores A and B, initialized to 1

P0	P1
wait (A);	wait(B)
wait (B);	wait(A)

Bridge Crossing Example

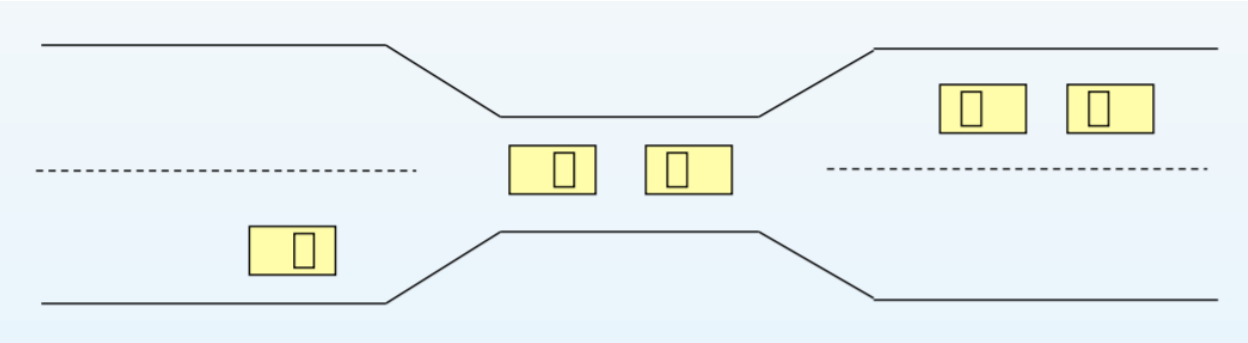


Fig:3.11 Bridge Crossing Example for Deadlock

- Traffic only in one direction.
- Each section of a bridge can be viewed as a resource.
- If a deadlock occurs, it can be resolved if one car backs up (preempt resources and rollback).
- Several cars may have to be backed up if a deadlock occurs.
- Starvation is possible.

SYSTEM MODEL

- Resource types R1, R2, . . ., Rm
- CPU cycles, memory space, I/O devices
- Each resource type Ri has Wi instances.
- Each process utilizes a resource as follows:
    - request
    - use
    - release

DEADLOCK CHARACTERIZATION

- Deadlock can arise if four conditions hold simultaneously.
- **Mutual exclusion**:

    Only one process at a time can use a resource.
- **Hold and wait:**

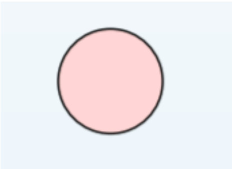
    A process holding at least one resource is waiting to acquire additional resource held by other processes.
- **No preemption:**

    A resource can be released only voluntarily by the process holding it, after that process has completed its task.
- **Circular wait:**

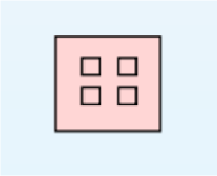
    There exists a set {P0, P1, …, P0} of waiting processes such that P0 is waiting for a resource that is held by P1, P1 is waiting for a resource that is held by P2, …, Pn–1 is waiting for a resource that is held by Pn, and P0 is waiting for a resource that is held by P0.

RESOURCE-ALLOCATION GRAPH

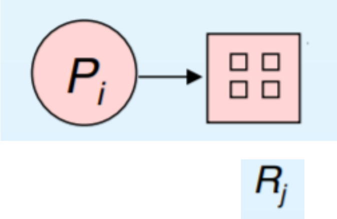
- A set of vertices V and a set of edges E.
- V is partitioned into two types:
  - P = {P1, P2, …, Pn}, the set consisting of all the processes in the system.
  - R = {R1, R2, …, Rm}, the set consisting of all resource types in the system.
- Request edge – directed edge P1 → Rj
- Assignment edge – directed edge Rj → Pi
- Process



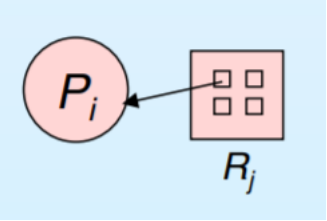
- Resource Type with 4 instances



- Pi requests instance of Rj



- Pi is holding an instance of Rj



EXAMPLE OF A RESOURCE ALLOCATION GRAPH

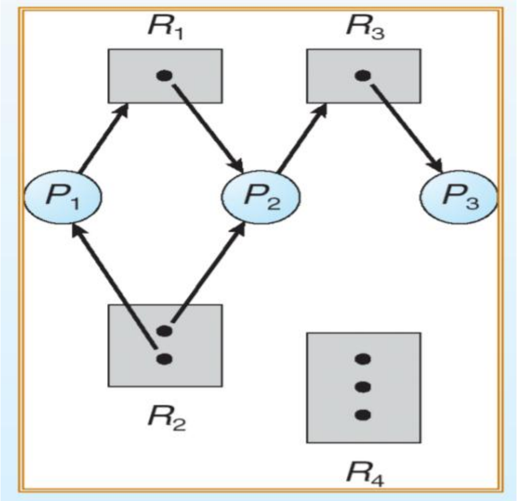


Fig:3.12 RAG with multiple instances

Will there be a deadlock here?

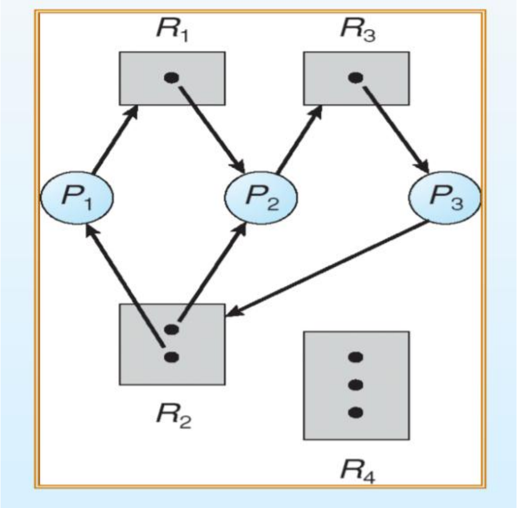


Fig:3.13 RAG with Deadlock

Resource Allocation Graph With A Cycle But No Deadlock

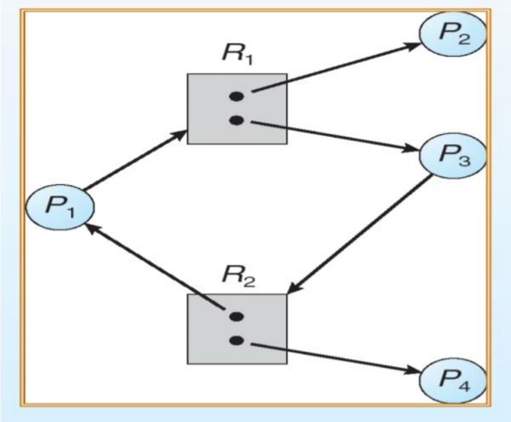


Fig:3.14 RAG with a cycle but No Deadlock

Basic Facts

- If graph contains no cycles ⇒ no deadlock.
- If graph contains a cycle ⇒
  - if only one instance per resource type, then deadlock.
  - if several instances per resource type, possibility of deadlock.

METHODS FOR HANDLING DEADLOCKS

- Deadlock Prevention
- Deadlock Avoidance
- Deadlock Prevention

DEADLOCK DETECTION

- Restrain the ways request can be made.
- **Mutual Exclusion** – not required for sharable resources; must hold for non-sharable resources.
- **Hold and Wait** – must guarantee that whenever a process requests a resource, it does not hold any other resources. Low resource utilization; starvation possible.

- **No Preemption** – If a process that is holding some resources requests another resource that cannot be immediately allocated to it, then all resources currently being held are released. Preempted resources are added to the list of resources for which the process is waiting. Process will be restarted only when it can regain its old resources, as well as the new ones that it is requesting.
- **Circular Wait** – impose a total ordering of all resource types, and require that each process requests resources in an increasing order of enumeration.

DEADLOCK AVOIDANCE

- Requires that the system has some additional a priori information available.
- Simplest and most useful model requires that each process declare the maximum number of resources of each type that it may need.
- The deadlock-avoidance algorithm dynamically examines the resource-allocation state to ensure that there can never be a circular-wait condition.
- Resource-allocation state is defined by the number of available and allocated resources, and the maximum demands of the processes.

DEADLOCK DETECTION

- Allow system to enter deadlock state
- Detection algorithm
- Recovery scheme

Basic Facts

- If a system is in safe state ⇒ no deadlocks.
- If a system is in unsafe state ⇒ possibility of deadlock.
- Avoidance ⇒ ensure that a system will never enter an unsafe state.

Safe, Unsafe , Deadlock State

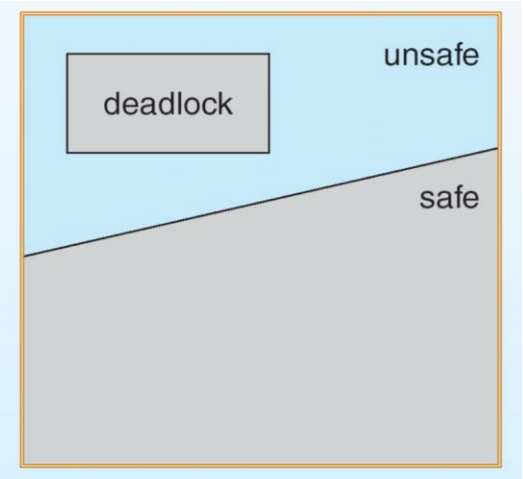


Fig:3.15 Deadlock with safe and unsafe state

RESOURCE-ALLOCATION GRAPH ALGORITHM

- Claim edge Pi → Rj indicated that process Pj may request resource Rj; represented by a dashed line.
- Claim edge converts to request edge when a process requests a resource.
- When a resource is released by a process, assignment edge reconverts to a claim edge.
- Resources must be claimed a priori in the system.

Resource-Allocation Graph For Deadlock Avoidance

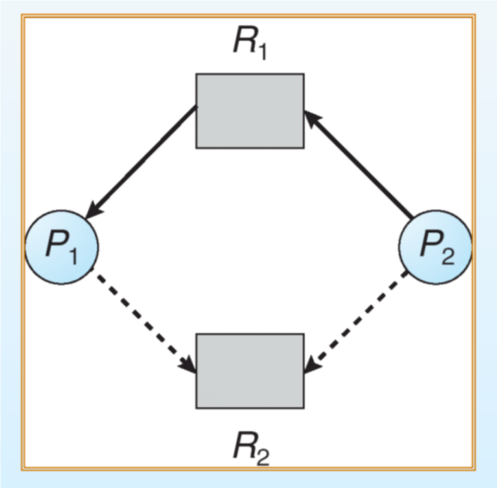


Fig:3.16 RAG for Deadlock Avoidance

Unsafe State In Resource-Allocation Graph

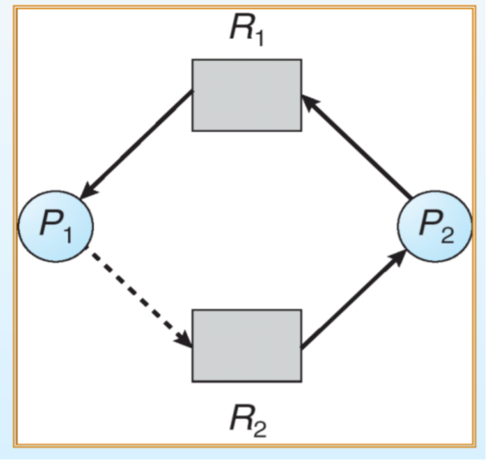


Fig:3.17 Unsafe state in RAG

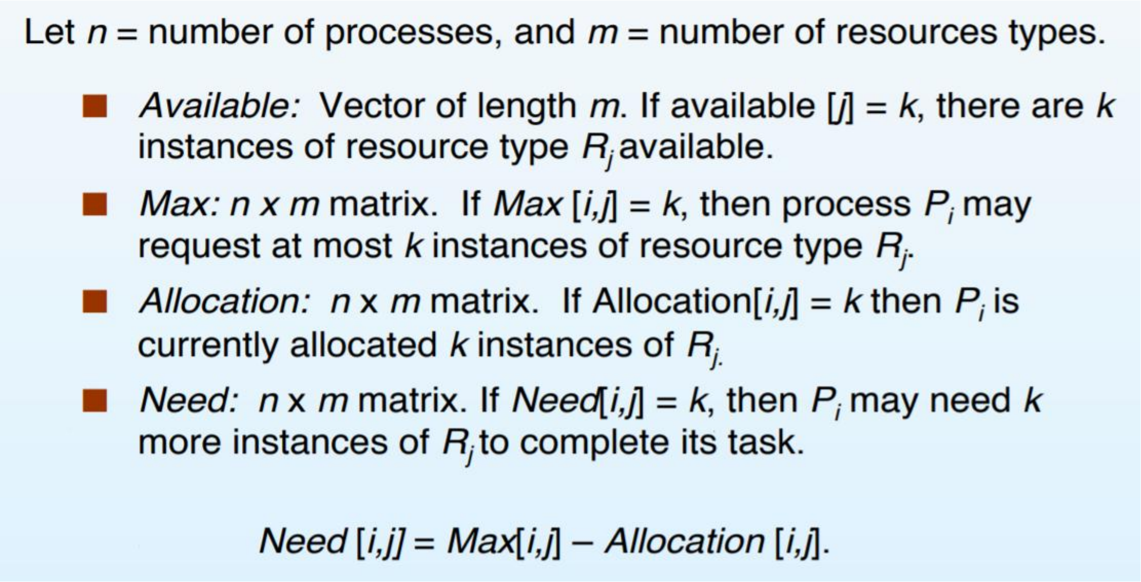
EXAMPLE:

- Banker's Algorithm
- Resource-Request Algorithm

BANKER'S ALGORITHM

- Multiple instances.
- Each process must a priori claim maximum use.
- When a process requests a resource it may have to wait.
- When a process gets all its resources it must return them in a finite amount of time.

Data Structures for the Banker's Algorithm

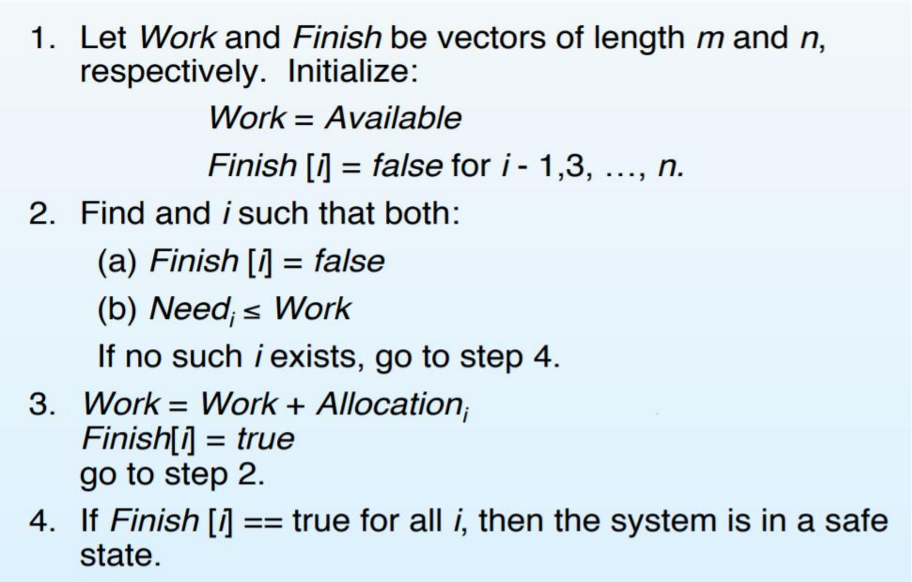


Let n = number of processes, and m = number of resources types.

- *Available:* Vector of length m. If available $[j] = k$, there are k instances of resource type R_j available.
- *Max:* n x m matrix. If *Max* $[i,j] = k$, then process P_i may request at most k instances of resource type R_j.
- *Allocation:* n x m matrix. If Allocation$[i,j] = k$ then P_i is currently allocated k instances of $R_{j.}$
- *Need:* n x m matrix. If *Need*$[i,j] = k$, then P_i may need k more instances of R_j to complete its task.

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

Safety Algorithm



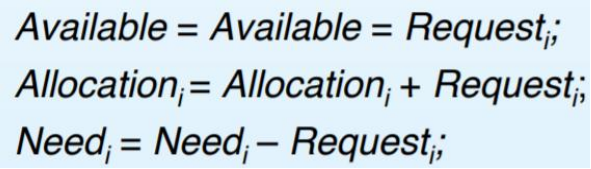
```
1. Let Work and Finish be vectors of length m and n,
   respectively. Initialize:
         Work = Available
         Finish [i] = false for i - 1,3, …, n.
2. Find and i such that both:
    (a) Finish [i] = false
    (b) Need_i ≤ Work
    If no such i exists, go to step 4.
3. Work = Work + Allocation_i
   Finish[i] = true
   go to step 2.
4. If Finish [i] == true for all i, then the system is in a safe
   state.
```

Resource-Request Algorithm for Process P_i

Request = request vector for process P_i

- If **Request$_i$ [j]** = k then process P_i wants k instances of resource type R_j.
- If **Request$_i$ ≤ Need$_i$** go to step 2. Otherwise, raise error condition, since process has exceeded its maximum claim.

- If **Request$_i$ ≤ Available**, go to step 3. Otherwise P_i must wait, since resources are not available.



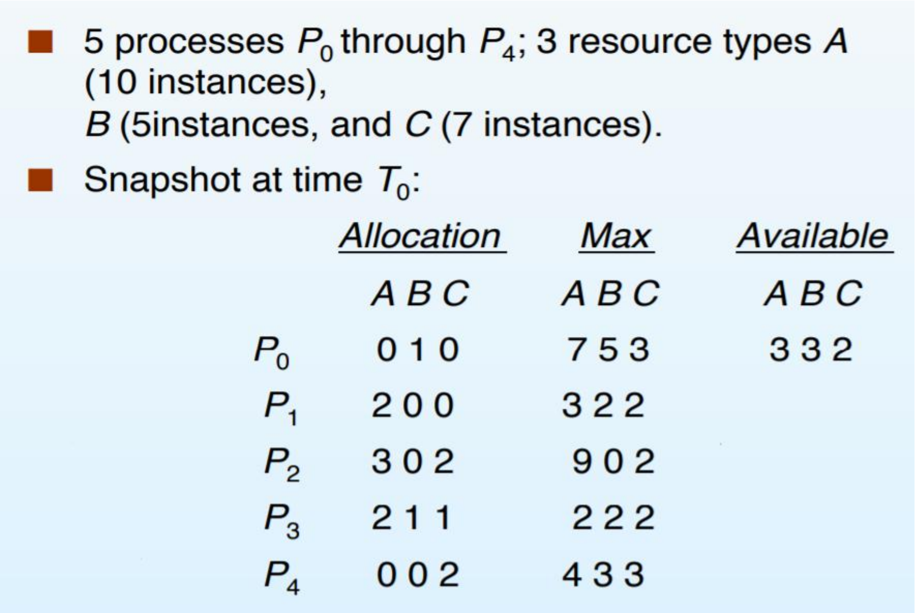
$Available = Available = Request_i;$

$Allocation_i = Allocation_i + Request_i;$

$Need_i = Need_i - Request_i;$

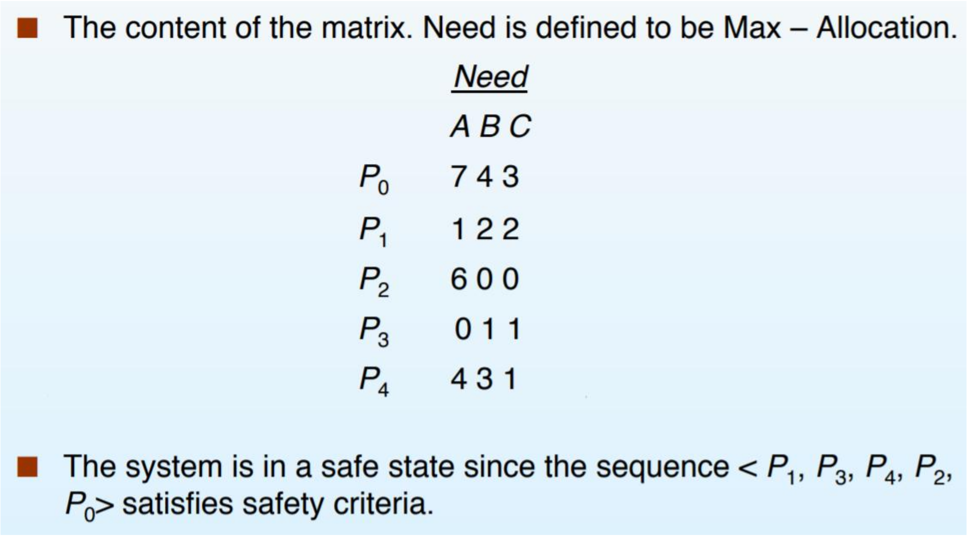
- If safe ⇒ the resources are allocated to Pi.
- If unsafe ⇒ Pi must wait, and the old resource-allocation state is restored

Example of Banker's Algorithm



- 5 processes P_0 through P_4; 3 resource types A (10 instances), B (5instances, and C (7 instances).
- Snapshot at time T_0:

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

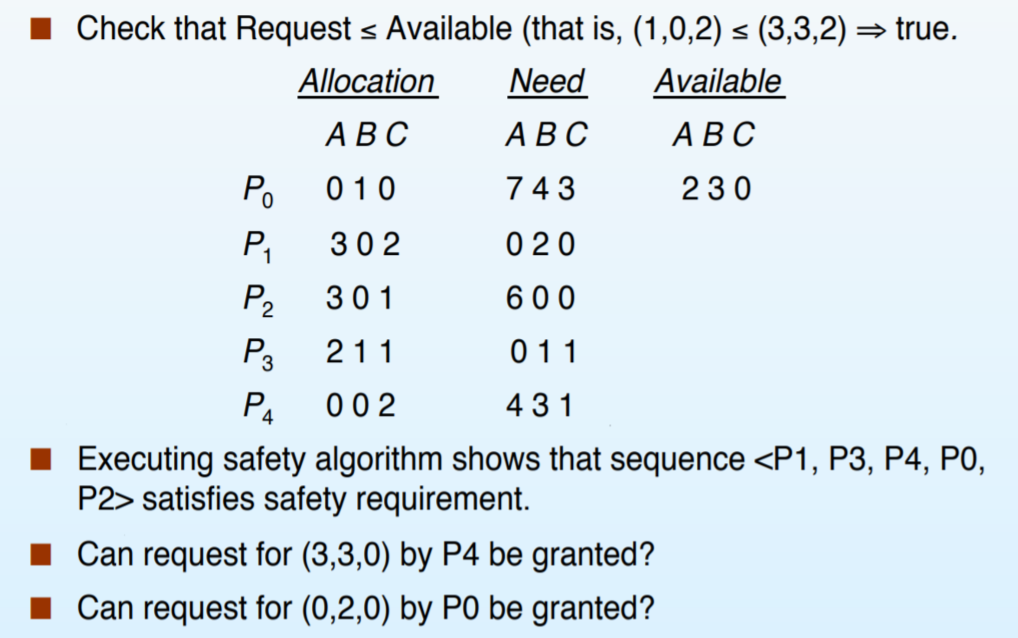


- The content of the matrix. Need is defined to be Max – Allocation.

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

- The system is in a safe state since the sequence < P_1, P_3, P_4, P_2, P_0> satisfies safety criteria.

Example request (contd)



- Check that Request ≤ Available (that is, (1,0,2) ≤ (3,3,2) ⇒ true.

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

- Executing safety algorithm shows that sequence <P1, P3, P4, P0, P2> satisfies safety requirement.
- Can request for (3,3,0) by P4 be granted?
- Can request for (0,2,0) by P0 be granted?

DEADLOCK DETECTION

- Allow system to enter deadlock state
- Detection algorithm
- Recovery scheme

Resource-Allocation Graph and Wait-for Graph

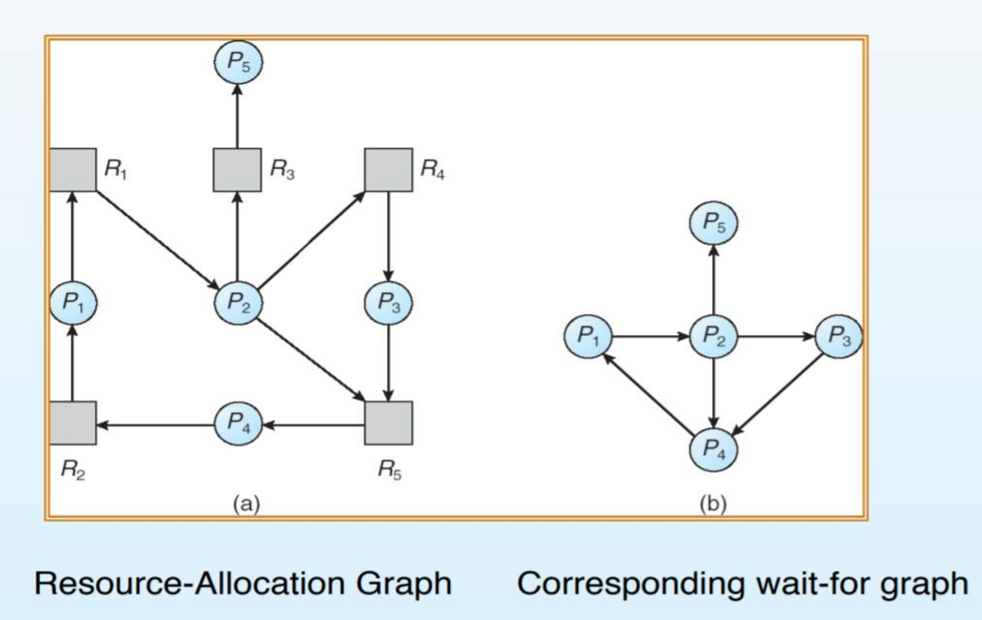
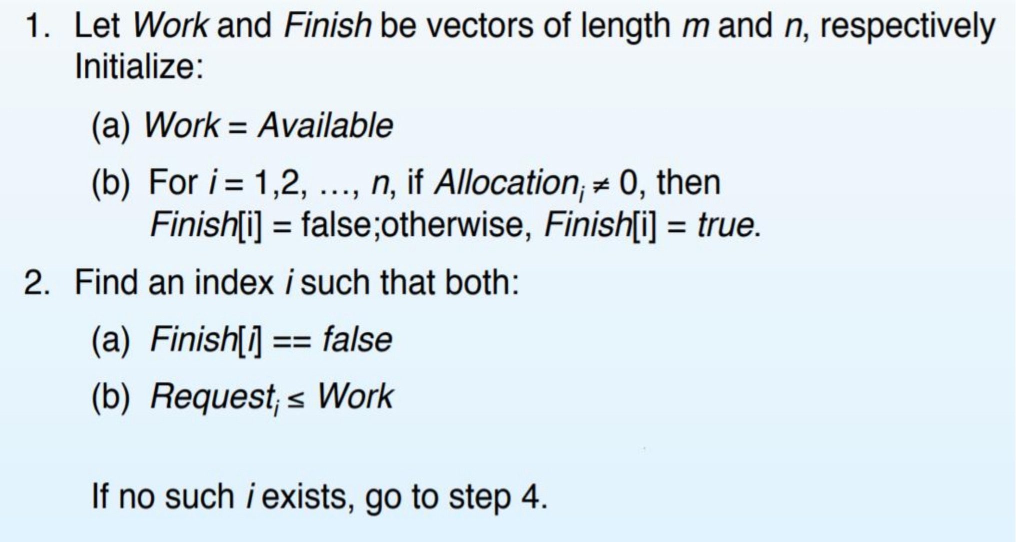


Fig:3.18 Wait-For Graph

Several Instances of a Resource Type

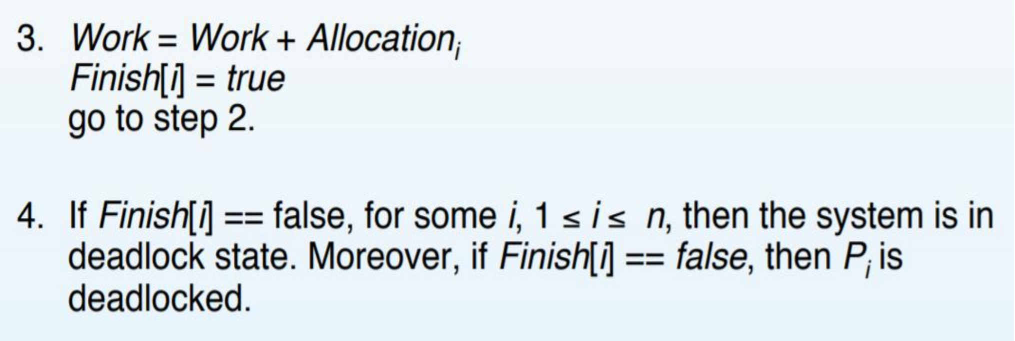
- **Available**: A vector of length m indicates the number of available resources of each type.
- **Allocation**: An n x m matrix defines the number of resources of each type currently allocated to each process.
- **Request**: An n x m matrix indicates the current request of each process. If Request [i_j] = k, then process P_i is requesting k more instances of resource type. R_j.

DETECTION ALGORITHM



1. Let *Work* and *Finish* be vectors of length *m* and *n*, respectively Initialize:
   (a) *Work = Available*
   (b) For $i = 1,2, \ldots, n$, if $Allocation_i \neq 0$, then *Finish*[i] = false;otherwise, *Finish*[i] = *true*.
2. Find an index *i* such that both:
   (a) *Finish*[*i*] == *false*
   (b) $Request_i \leq Work$

   If no such *i* exists, go to step 4.



3. $Work = Work + Allocation_i$
   Finish[*i*] = *true*
   go to step 2.

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

DETECTION-ALGORITHM USAGE

- When, and how often, to invoke depends on:
- How often a deadlock is likely to occur?
- How many processes will need to be rolled back? one for each disjoint cycle If detection algorithm is invoked arbitrarily, there may be many cycles in the resource graph and so we would not be able to tell which of the many deadlocked processes “caused” the deadlock.

RECOVERY FROM DEADLOCK: PROCESS TERMINATION

- Abort all deadlocked processes.
- Abort one process at a time until the deadlock cycle is eliminated.
- In which order should we choose to abort?
- Priority of the process.
- How long process has computed, and how much longer to completion.
- Resources the process has used. Resources process needs to complete.

- How many processes will need to be terminated.
- Is process interactive or batch?

RECOVERY FROM DEADLOCK: RESOURCE PREEMPTION

- Selecting a victim – minimize cost.
- **Rollback** – return to some safe state, restart process for that state.
- **Starvation** – same process may always be picked as victim, include number of rollback in cost factor.

SATHYABAMA
INSTITUTE OF SCIENCE AND TECHNOLOGY
(DEEMED TO BE UNIVERSITY)
Accredited "A" Grade by NAAC | 12B Status by UGC | Approved by AICTE
www.sathyabama.ac.in

SCHOOL OF COMPUTING

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

UNIT – IV – Operating System – SBS1206

OPERATING SYSTEM:

UNIT IV **(9 Hrs)**

Memory Management: Address Binding - Dynamic Loading and Linking - Overlays - Logical and Physical Address Space - Contiguous Allocation- Non-Contiguous Allocation.

UNIT 4

STORAGE / MEMORY MANAGEMENT

STORAGE / MEMORY MANAGEMENT:

- Memory management is the functionality of an operating system which handles or manages primary memory.
- Memory management keeps track of each and every memory location either itis allocated to some process or it is free.
- It checks how much memory is to be allocated toprocesses. It decides which process will get memory at what time.
- It tracks whenever somememory gets freed or unallocated and correspondingly it updates the status.
- Memory management provides protection by using two registers, a base register and a limitregister.
- The base register holds the smallest legal physical memory address and the limitregister specifies the size of the range.
- For example, if the base register holds 300000 and thelimit register is 1209000, then the program can legally access all addresses from 300000through 411999

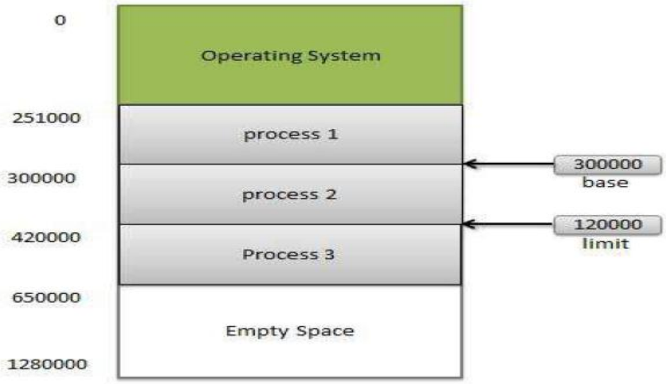


Fig:4.1 Storage/Memory Management

GOALS OF MEMORY MANAGEMENT:

- Allocate available memory efficiently to multiple processes
- Main functions
- Allocate memory to processes when needed
- Keep track of what memory is used and what is free
- Protect one process's memory from another

MEMORY ALLOCATION STRATEGIES:

- **CONTIGUOUS ALLOCATION :**
    - In contiguous memory allocation each process is contained in a single contiguous block of memory.
    - Memory is divided into several fixed size partitions.
    - Each partition contains exactly one process.
    - When a partition is free, a process is selected from the input queue and loaded into it.
    - The free blocks of memory are known as holes.
    - The set of holes is searched to determine which hole is best to allocate.

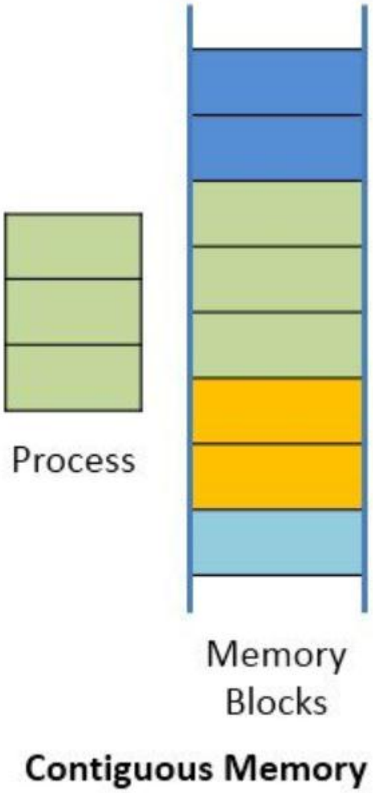


Fig:4.2 Continuous Memory Management

- **NON-CONTIGUOUS ALLOCATION:**
  - Parts of a process can be allocated noncontiguous chunks of memory.
  - In context to memory organization, non contiguous memory allocation means the available memory space is scattered here and there it means all the free available memory space is not together at one place.

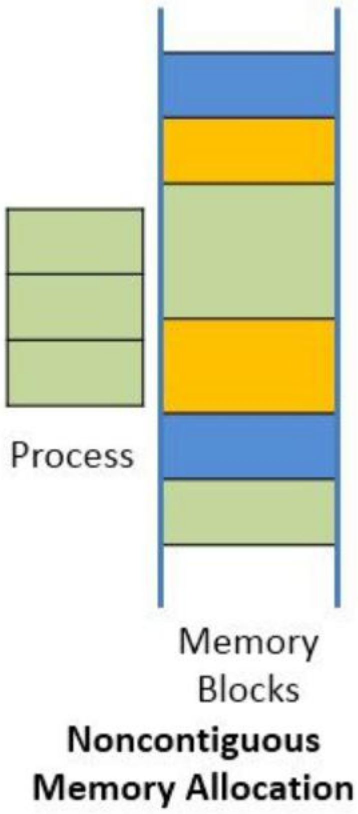


Fig:4.3 Non-Continuous Memory Management

CONTAGIOUS Vs NON CONTAGIOUS

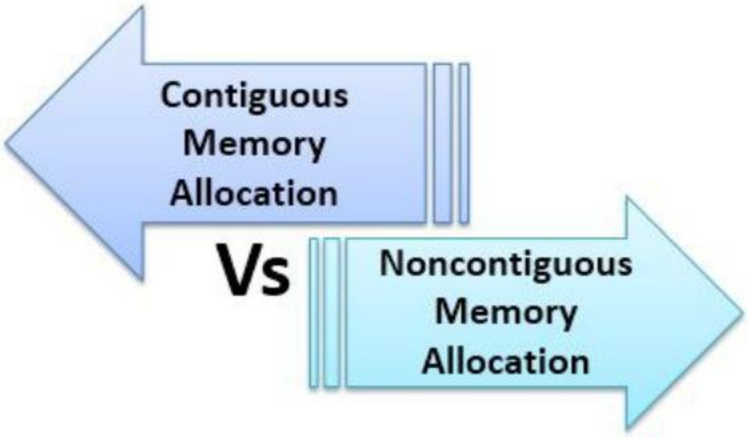
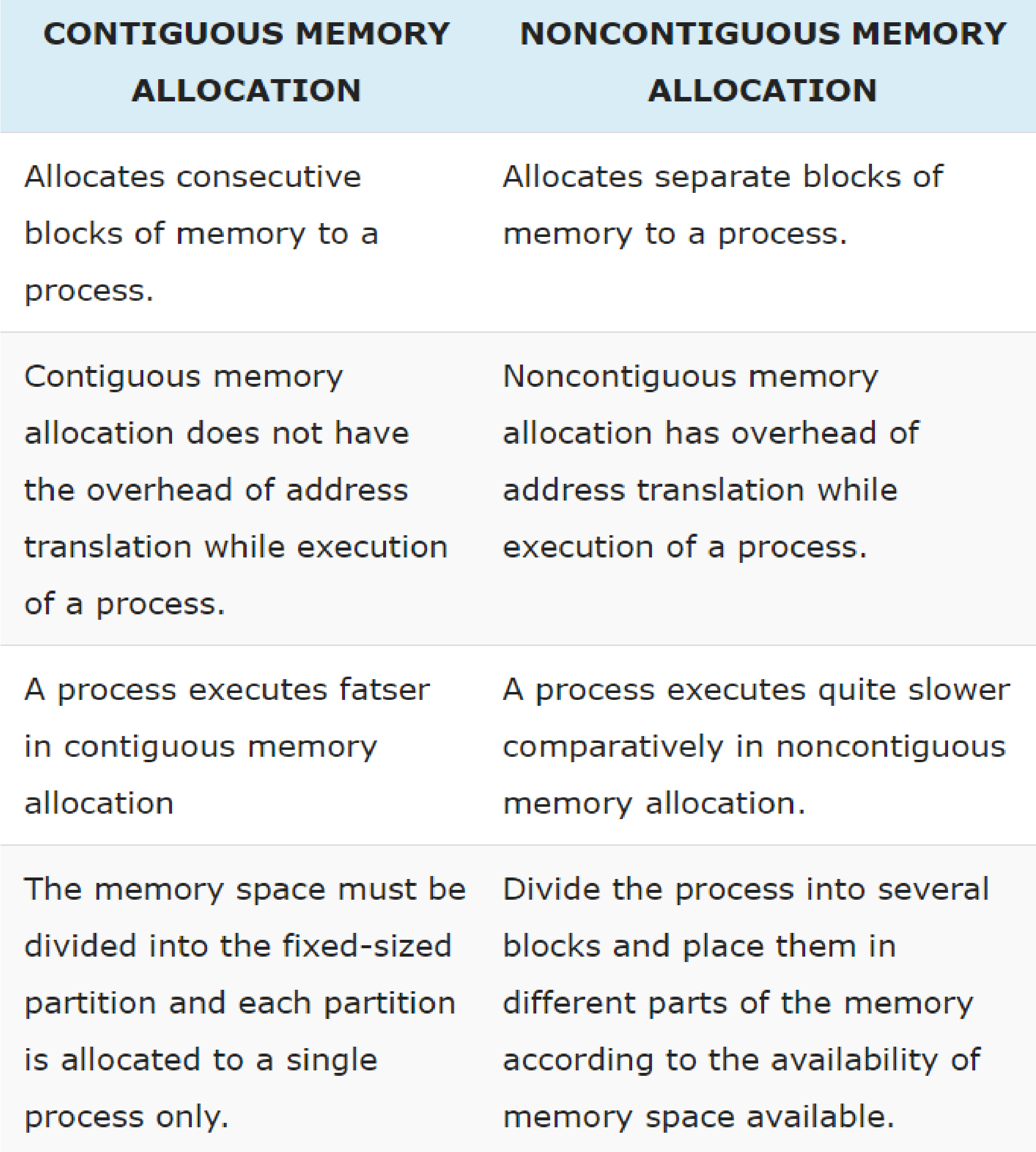


Fig:4.4 Continuous Vs Non Continuous Memory Management



CONTIGUOUS MEMORY ALLOCATION	NONCONTIGUOUS MEMORY ALLOCATION
Allocates consecutive blocks of memory to a process.	Allocates separate blocks of memory to a process.
Contiguous memory allocation does not have the overhead of address translation while execution of a process.	Noncontiguous memory allocation has overhead of address translation while execution of a process.
A process executes fatser in contiguous memory allocation	A process executes quite slower comparatively in noncontiguous memory allocation.
The memory space must be divided into the fixed-sized partition and each partition is allocated to a single process only.	Divide the process into several blocks and place them in different parts of the memory according to the availability of memory space available.

FIXED PARTITION SCHEME:

- Memory is broken up into fixed size partitions
- But the size of two partitions may be different
- Each partition can have exactly one process
- When a process arrives, allocate it a free partition
- Easy to manage
- Problems - Maximum size of process bound by max. partition size, Largeinternal fragmentation possible

VARIABLE PARTITION SCHEME

- Hole – block of available memory; holes of various size are scattered throughout memory
- When a process arrives, it is allocated memory from a hole large enough to accommodate it
- Operating system maintains information about:
  - allocated partitions
  - free partitions (hole)

MULTIPROGRAMMING:

- To overcome the problem of underutilization of CPU and main memory, the multiprogramming was introduced.
- The multiprogramming is interleaved execution of multiple jobs by the samecomputer.
- In multiprogramming system, when one program is waiting for I/O transfer, there isanother program ready to utilize the CPU.
- SO it is possible for several jobs to share the time ofthe CPU.
- But it is important to note that multiprogramming is not defined to be the execution ofjobs at the same instance of time.
- Rather it does mean that there are a number of jobsavailable to the CPU (placed in main memory) and a portion of one is executed then a segmentof another and so on.

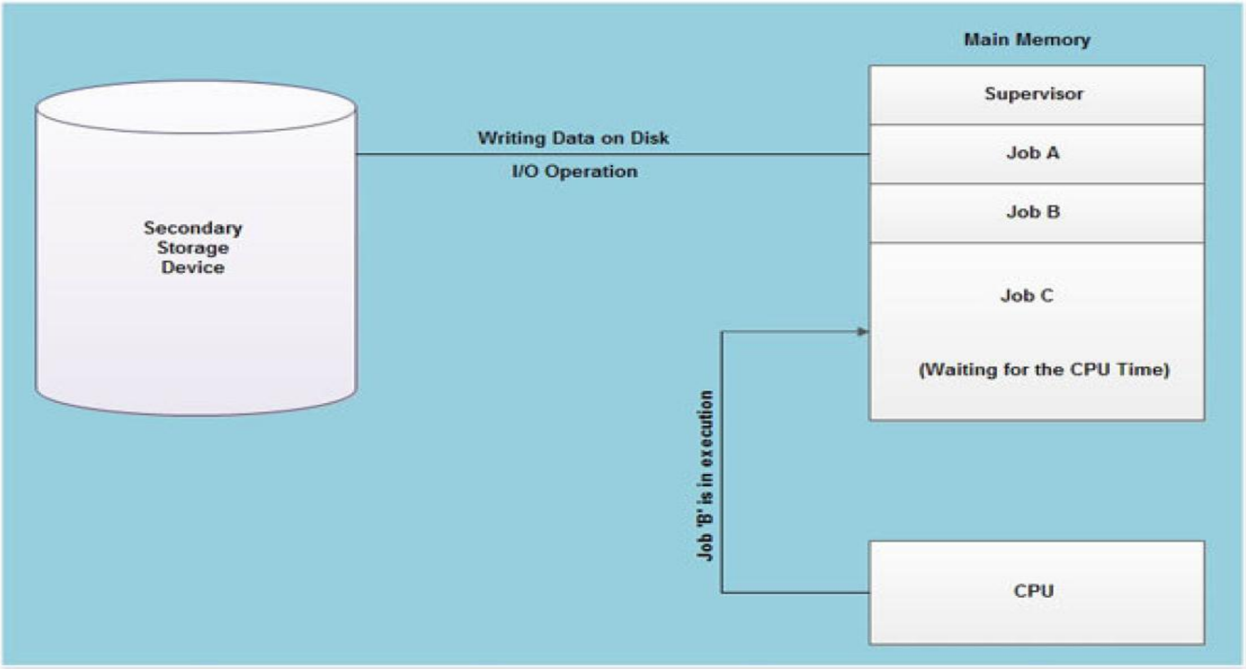


Fig:4.5 Multiprogramming Memory Management

- At the particular situation, job' A' is not utilizing the CPU time because it is busy in I/ 0 operations.
- Hence the CPU becomes busy to execute the job 'B'. Another job C is waiting for the CPU for getting its execution time.
- So in this state the CPU will never be idle and utilizes maximum of its time.
- A program in execution is called a "Process", "Job" or a "Task".
- The concurrent execution of programs improves the utilization of system resources and enhances the system throughput as compared to batch and serial processing.
- In this system, when a process requests some I/O to allocate; meanwhile the CPU time is assigned to another ready process.
- So, here when a process is switched to an I/O operation, the CPU is not set idle.
- Multiprogramming is a common approach to resource management.
- The essential components of a single-user operating system include a command processor, an input/ output control system, a file system, and a transient area.

- A multiprogramming operating system builds on this base, subdividing the transient area to hold several independent programs and adding resource management routines to the operating system's basic functions.

STORAGE HIERARCHY

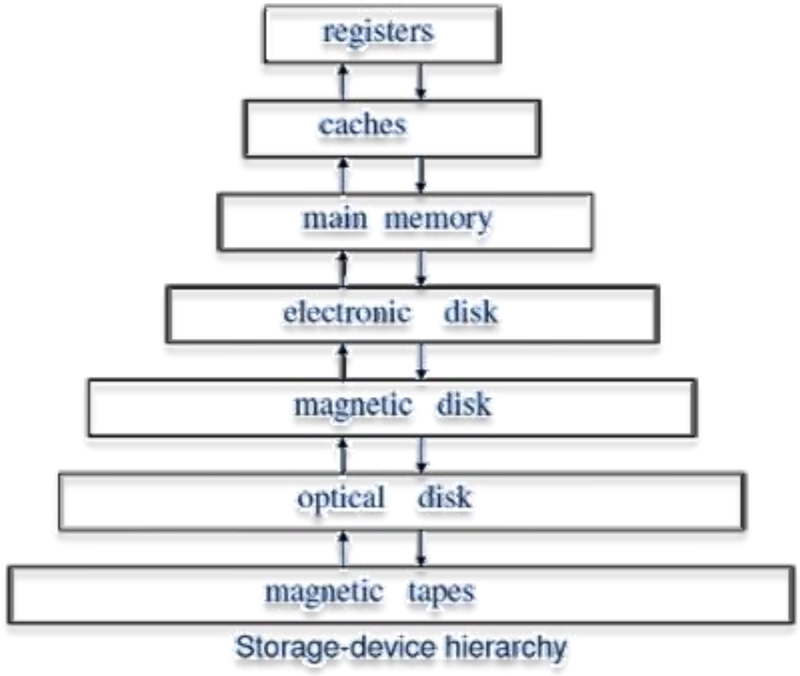


Fig:4.6 Storage Hierarchy

- The wide variety of storage systems in a computer system can be organized in a hierarchyaccording to their speed and their cost.
- The higher levels are expensive but fast. As we movedown the hierarchy, the cost per bit decreases, whereas the access time increases.
- The reasonfor using the slower memory decives is that they are cheaper than the faster ones.
- Many early storage devices, including paper tape and core memories, are found only in museums now that magnetic tape and semiconductor memory have become faster and cheaper.

DYNAMIC LOADING

- In dynamic loading, a routine of a program is not loaded until it is called by the program.

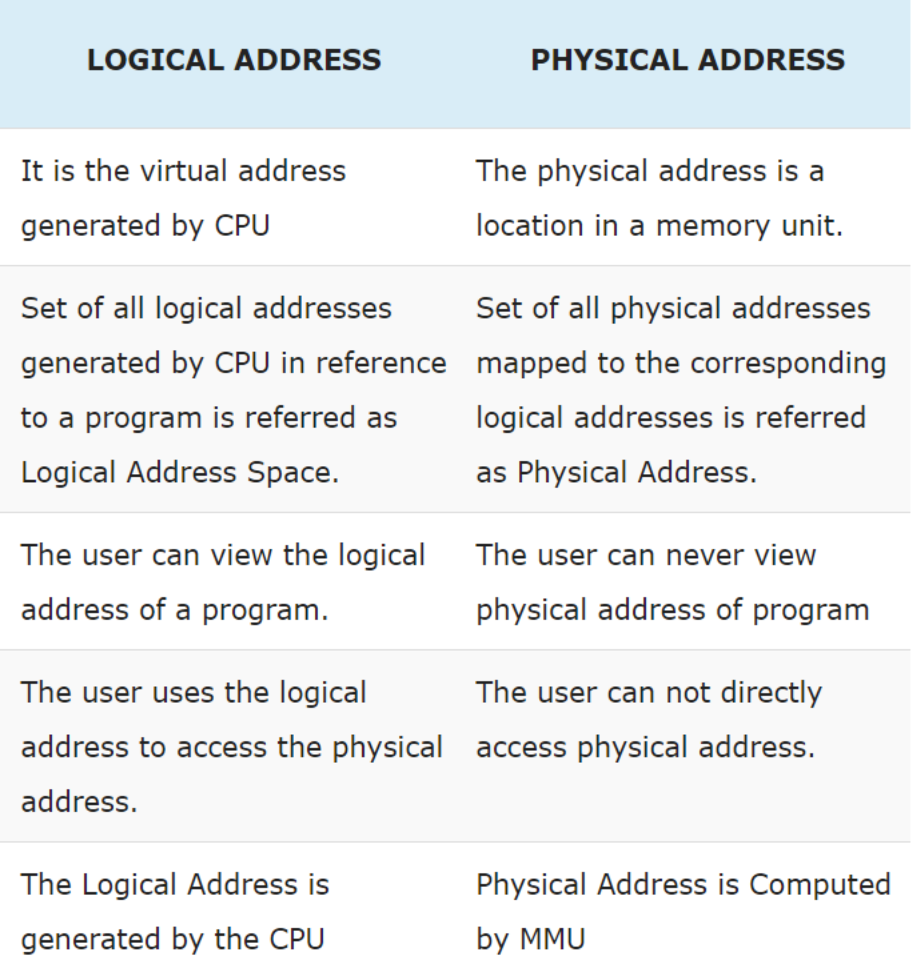
- Allroutines are kept on disk in a re-locatable load format.
- The main program is loaded intomemory and is executed.
- Other routines methods or modules are loaded on request.
- Dynamicloading makes better memory space utilization and unused routines are never loaded.

DYNAMIC LINKING

- Linking is the process of collecting and combining various modules of code and data into aexecutable file that can be loaded into memory and executed.
- Operating system can linksystem level libraries to a program.
- When it combines the libraries at load time, the linking iscalled static linking and when this linking is done at the time of execution, it is called asdynamic linking.

LOGICAL vs PHYSICAL ADDRESS SPACE:

- An address generated by the CPU is a logical address whereas address actually available onmemory unit is a physical address.
- Logical address is also known a Virtual address.
- Virtual and physical addresses are the same in compile-time and load-time address-binding schemes.
- Virtual and physical addresses differ in execution-time address-binding scheme.



LOGICAL ADDRESS	PHYSICAL ADDRESS
It is the virtual address generated by CPU	The physical address is a location in a memory unit.
Set of all logical addresses generated by CPU in reference to a program is referred as Logical Address Space.	Set of all physical addresses mapped to the corresponding logical addresses is referred as Physical Address.
The user can view the logical address of a program.	The user can never view physical address of program
The user uses the logical address to access the physical address.	The user can not directly access physical address.
The Logical Address is generated by the CPU	Physical Address is Computed by MMU

SWAPPING

- Swapping is a mechanism in which a process can be swapped temporarily out of main memoryto a backing store, and then brought back into memory for continued execution.
- Backing storeis a usually a hard disk drive or any other secondary storage which fast in access and largeenough to accommodate copies of all memory images for all users.
- It must be capable ofproviding direct access to these memory images.
- Major time consuming part of swapping is transfer time.
- Total transfer time is directlyproportional to the amount of memory swapped.
- Let us assume that the user process is of size100KB and the backing store is a standard hard disk with transfer rate of 1 MB per second.
- The actual transfer of the 100K process to or from memory will take
- 100KB / 1000KB per second = 1/10 second = 100 milliseconds

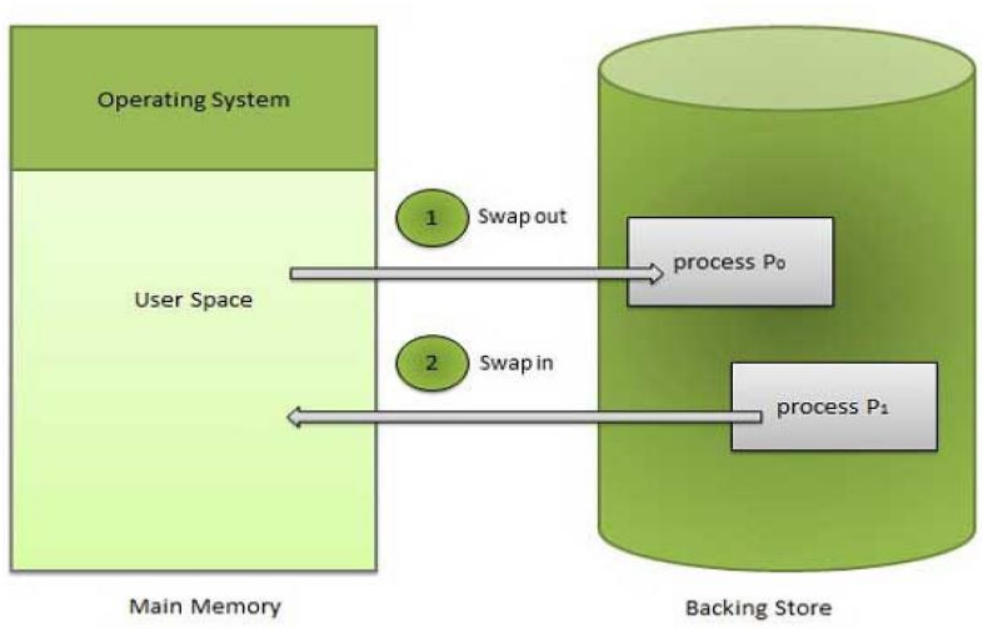


Fig:4.7 Swapping Process

MULTIPLE-PARTITION ALLOCATION

FIXED PARTITION ALLOCATION

- One memory allocation method is to divide the memory into a number of fixed-size partitions.
- Each partition may contain exactly one process. Thus the degree of multiprogramming is boundby the number of partitions.
- When a partition is free, a process is selected from the input queueand is loaded into the free partition.
- When the process terminates, the partition becomesavailable for another process.

DYNAMIC ALLOCATION

- The operating system keeps a table indicating which parts of memory is available (called holes)are available and which are occupied.
- Initially, all memory is available for user processes (i.e.,there is one big hole).
- When a process arrives, we select a hole which is large enough to holdthis process.

- We allocate as much memory is required for the process and the rest is kept as a hole which can be used for later requests.

Selection of a hole to hold a process can follow the following algorithms

- **FIRST-FIT:**
  - Allocate the first hole that is big enough. Searching can start at the beginning ofthe set of holes or where the previous first-fit search ended. There may be many holes in the memory, so the operating system, to reduce the amount of time it spends analyzing the available spaces, begins at the start of primary memory and allocates memory from the first hole it encounters large enough to satisfy the request.
- **BEST-FIT:**
  - Allocate the smallest hole that is big enough. We must search the entire list,unless the list is kept ordered by size. This strategy produces the smallest leftover hole.
- **WORST-FIT:**
  - Allocate the largest hole that is big enough. We must search the entire list,unless the list is kept ordered by size. This strategy produces the largest leftover hole.
  - The idea is that this placement will create the largest hold after the allocations, thus increasing the possibility that compared to best fit, another process can use the remaining space.

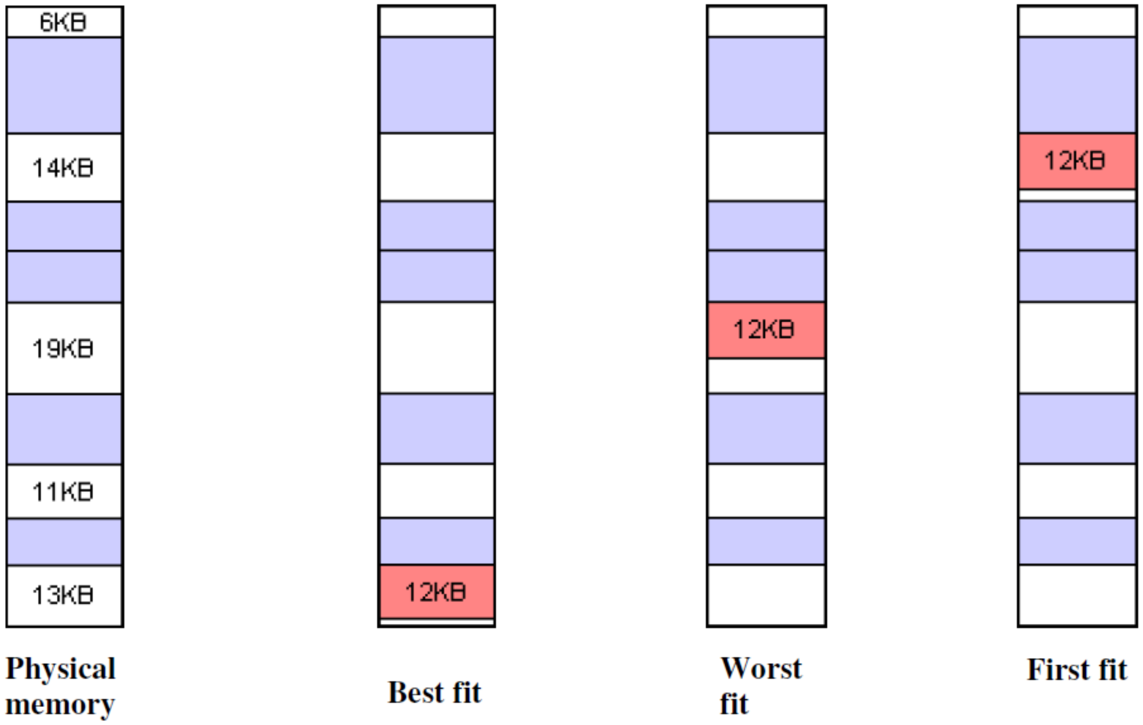
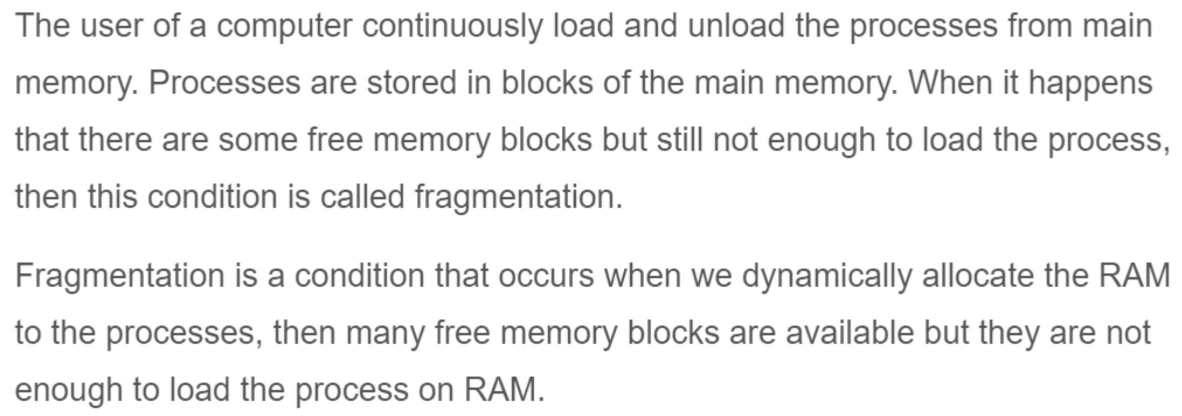


Fig:4.8 Dynamic Allocation

FRAGMENTATION:

- As processes are loaded and removed from memory, the free memory space is broken into little pieces.
- It happens after sometimes that processes cannot be allocated to memory blocks considering their small size and memory blocks remains unused. This problem is known as Fragmentation.

  OR



The user of a computer continuously load and unload the processes from main memory. Processes are stored in blocks of the main memory. When it happens that there are some free memory blocks but still not enough to load the process, then this condition is called fragmentation.

Fragmentation is a condition that occurs when we dynamically allocate the RAM to the processes, then many free memory blocks are available but they are not enough to load the process on RAM.

TYPES:

- **External fragmentation**
- **Internal fragmentation**

EXTERNAL FRAGMENTATION:

- External Fragmentation happens when a dynamic memory allocation algorithm allocates some memory and a small piece is left over that cannot be effectively used.
- If too much external fragmentation occurs, the amount of usable memory is drastically reduced.
- Total memory space exists to satisfy a request, but it is not contiguous.
- The problem of fragmentation can be solved by **COMPACTION.**
- The goal is to shuffle the memorycontents to place all free memory together in one large block.
- For a relocated process to beable to execute in its new location, all internal addresses (e.g., pointers) must be relocated.
- Ifthe relocation is static and is done at assembly or load time, compaction cannot be done.
- Compaction is possible only if relocation is dynamic and is done at execution time.
- If addressesare relocated dynamically, relocation requires only moving the program and data, and thenchanging the base register to reflect the new base address.

There may be many compaction algorithms:

- Simply move all processes toward one end of the memory; all holes move in the other direction producing one large hole of available memory.

- Create a large hole big enough anywhere to satisfy the current request.

INTERNAL FRAGMENTATION

- Internal fragmentation is the space wasted inside of allocated memory blocks because of restriction on the allowed sizes of allocated blocks.
- Allocated memory may be slightly larger than requested memory; this size difference is memory internal to a partition, but not being used.

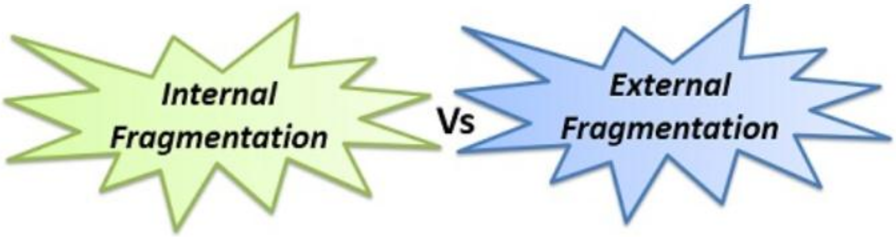
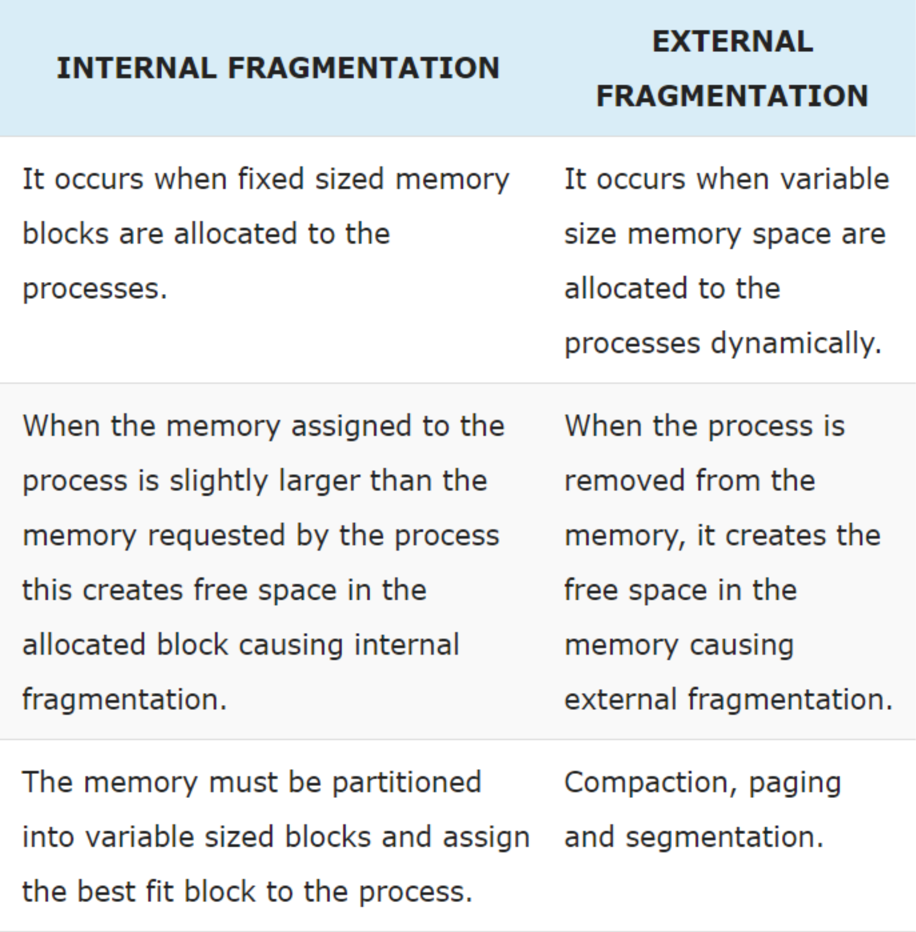


Fig:4.9 Internal Vs External Fragmentation

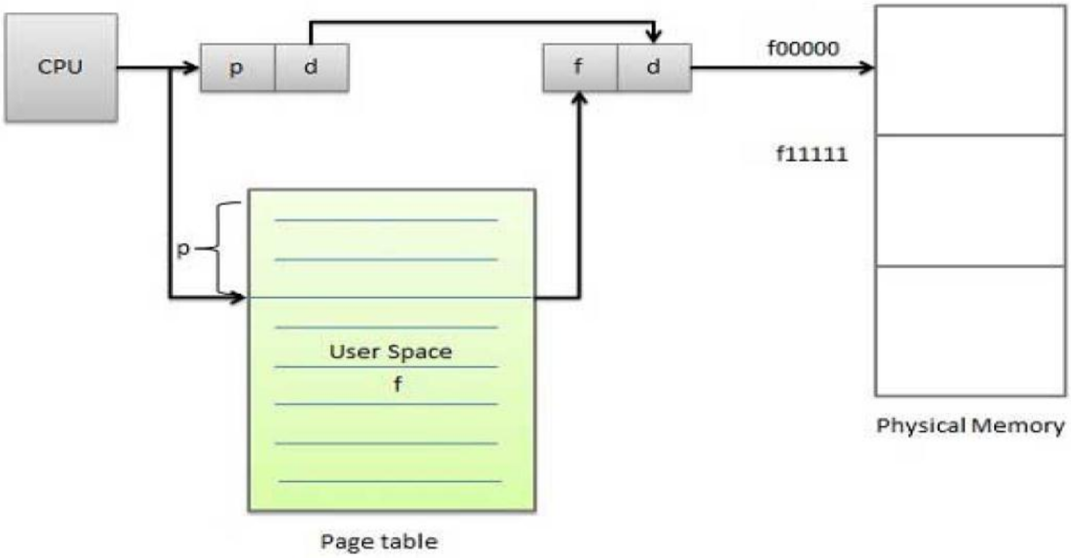


INTERNAL FRAGMENTATION	EXTERNAL FRAGMENTATION
It occurs when fixed sized memory blocks are allocated to the processes.	It occurs when variable size memory space are allocated to the processes dynamically.
When the memory assigned to the process is slightly larger than the memory requested by the process this creates free space in the allocated block causing internal fragmentation.	When the process is removed from the memory, it creates the free space in the memory causing external fragmentation.
The memory must be partitioned into variable sized blocks and assign the best fit block to the process.	Compaction, paging and segmentation.

PAGING

- Paging is a memory management scheme that eliminates the need for contiguous allocation of physical memory.
- This scheme permits the physical address space of a process to be non – contiguous.
- Logical Address or Virtual Address (represented in bits): An address generated by the CPU
- Logical Address Space or Virtual Address Space(represented in words or bytes): The set of all logical addresses generated by a program
- Physical Address (represented in bits): An address actually available on memory unit
- Physical Address Space (represented in words or bytes): The set of all physical addresses corresponding to the logical addresses
- Physical memory is broken into fixed-size blocks called **FRAMES**.
- Logical memory is also brokeninto blocks of the same size called **PAGES.**

- When a process is to be executed, its pages (whichare in the backing store) are loaded into any available memory frames. Thus the pages of aprocess may not be contiguous.



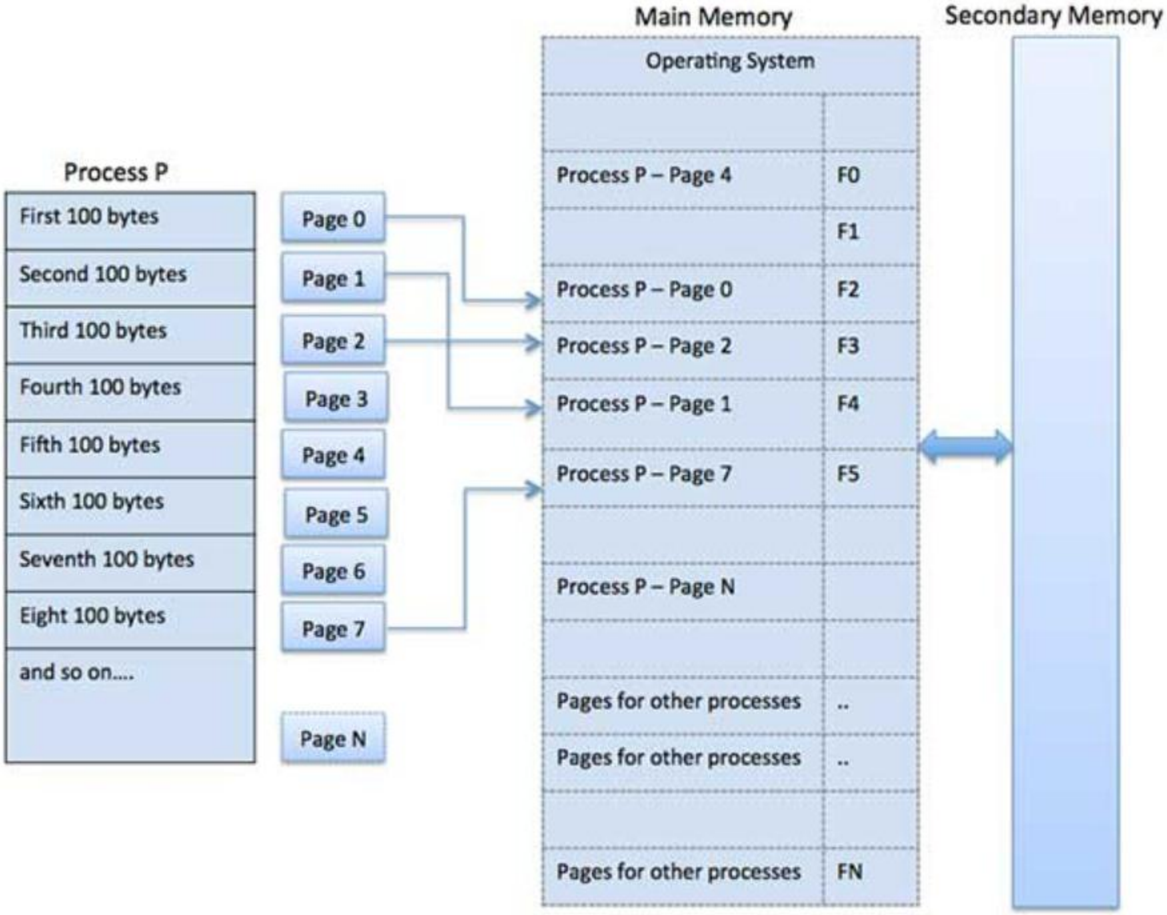


Fig:4.10 Paging

Address Translation

- Page address is called **logical address** and represented by **page number** and the **offset**.
    - Logical Address = Page number + page offset
- Frame address is called **physical address** and represented by a **frame number** and the **offset**.
    - Physical Address = Frame number + page offset
- Paging eliminates external fragmentation altogether but there may be a little internalfragmentation.

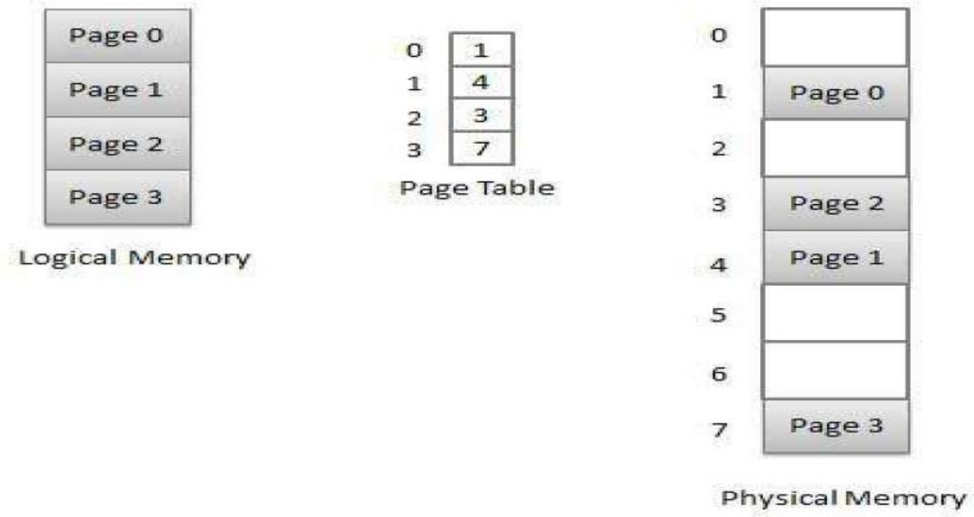
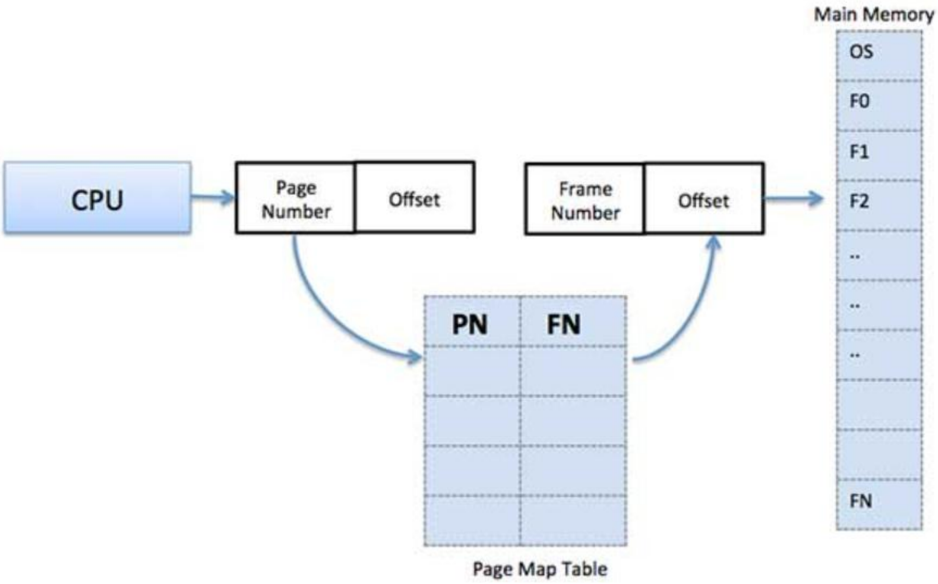


Fig:4.11 Address Translation



ADVANTAGES AND DISADVANTAGES OF PAGING

- Paging reduces external fragmentation, but still suffer from internal fragmentation.
- Paging is simple to implement and assumed as an efficient memory management technique.
- Due to equal size of the pages and frames, swapping becomes very easy.
- Page table requires extra memory space, so may not be good for a system having small RAM.

MULTILEVEL PAGING

- Most computer systems support a very large logical address space (232 to 264). In such a case, the page table itself becomes very very large.
- E.g., consider a 32-bit logical address space. If the page size is 4K bytes (212), then a page table may consist of up to (232 / 212) = 1 millionentries. If each entry consists of 4 bytes, each process may need 4 megabytes of physicaladdress alone for the page table
- **DISADVANTAGE**: Page tables consume a large amount of physical memory because each page table canhave millions of entries.

INVERTED PAGE TABLE

- To overcome the disadvantage of page tables given above, an inverted page table could be used.An inverted page table has one entry for each (frame) of memory.

- Each entry consists ofthe logical (or virtual) address of the page stored in that memory location, with information aboutthe process that owns it.
- Thus there is only one inverted page table in the system, and it hasonly one entry for each frame of physical memory.
- **DISADVANTAGE:**
  - The complete information about the logical address space of a process, which is required if a referenced page is not currently in memory is no longer kept.
  - To overcome this, anexternal page table (one per process) must be kept. Each such table looks like thetraditional per-process page table, containing information on where each logical page islocated.
  - These external page tables need not be in the memory all the time, because theyare needed only when a page fault occurs.

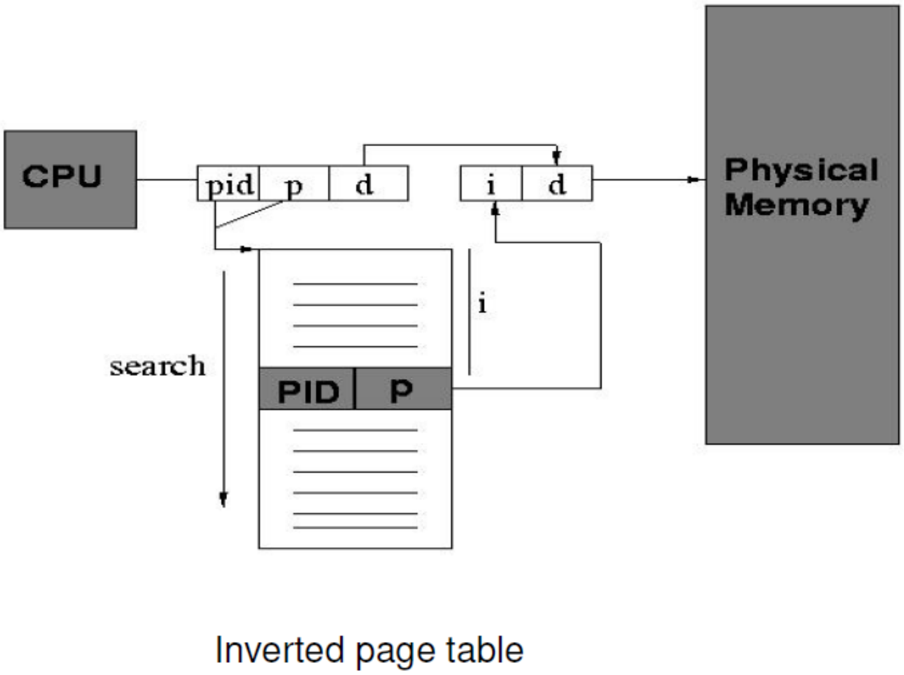


Fig:4.12 Inverted Page Table

PROTECTION

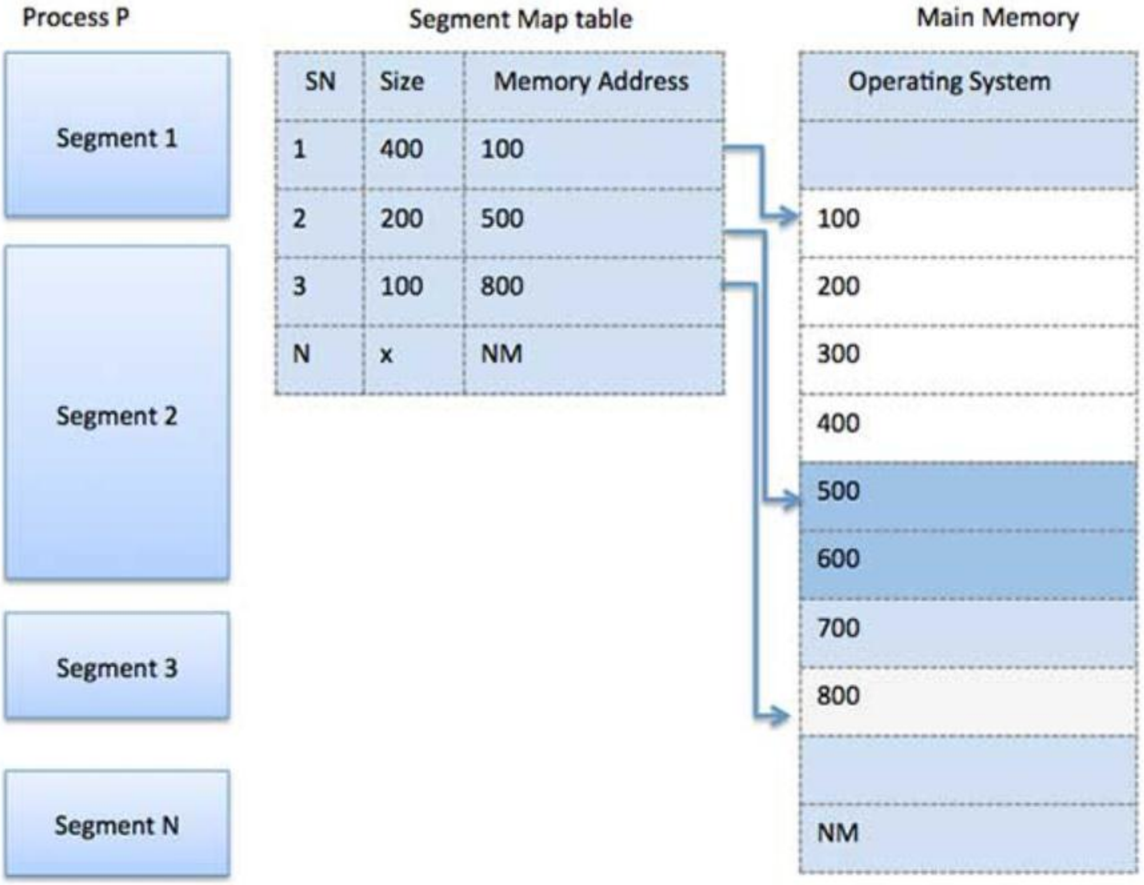
- Memory protection in a paged environment is accomplished by protection bits that areassociated with each frame. Normally, they are kept in the page table.
- One bit can define apage to be read-and-write or read-only.

SHARED PAGES

- Another advantage of paging is the possibility of sharing common code.
- Consider a system that supports 40 users, each of which executes a text editor.
- If the text editor consists of 150K of code and 50K of data space, then we would need 8000K to support the 40 users.
- If the code is reentrant (non-self-modifying), then it can be shared between the 40 users.

SEGMENTATION:

- Segmentation is a memory management technique in which each job is divided into several segments of different sizes, one for each module that contains pieces that perform related functions.
- Each segment is actually a different logical address space of the program.
- When a process is to be executed, its corresponding segmentations are loaded into non-contiguous memory though every segment is loaded into a contiguous block of available memory.
- Segmentation memory management works very similar to paging but here segments are of variable-length where as in paging pages are of fixed size.
- A program segment contains the program's main function, utility functions, data structures, and so on.
- The operating system maintains a segment map table for every process and a list of free memory blocks along with segment numbers, their size and corresponding memory locations in main memory.



SN	Size	Memory Address
1	400	100
2	200	500
3	100	800
N	x	NM

Fig:4.13 Segmentation

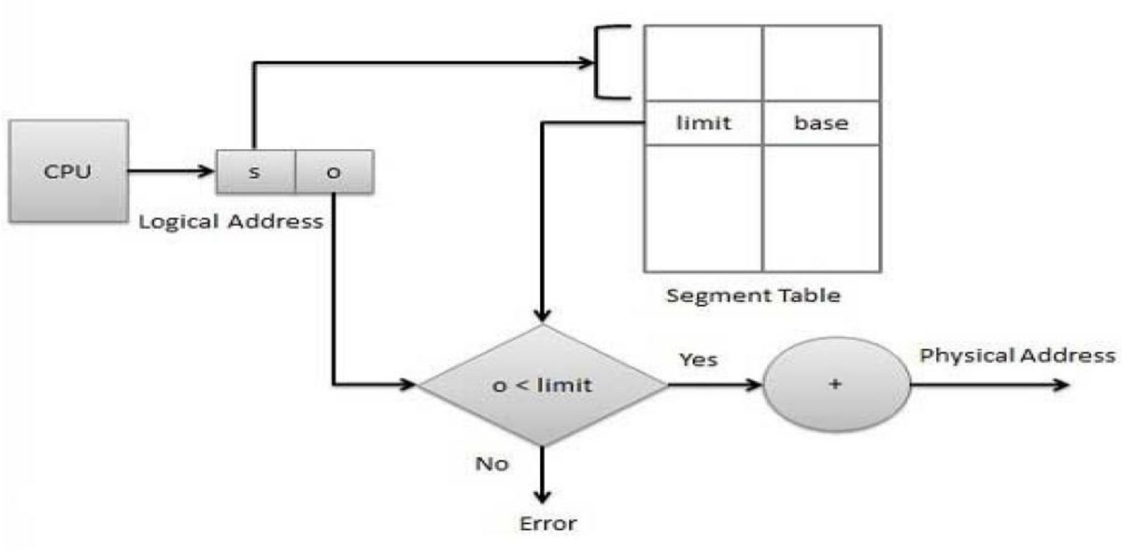
- Segmentation is a memory-management scheme that suggests that a logical address space bedivided into a collection of segments. Each segment has a name and a length.
- Addressesspecify both the segment name and the offset within the segment.
- The user therefore specifieseach address by two quantities: a segment name (or segment number) and an offset.
- **SEGMENT NUMBER (S)** -- segment number is used as an index into a segment tablewhich contains base address of each segment in physical memory and a limit ofsegment.
- **SEGMENT OFFSET (O)** -- segment offset is first checked against limit and then is combinedwith base address to define the physical memory address.

- Therefore logical addresses consist of **two tuples**:
    - **<segment-number, offset>**

TUPLES:

- In the context of relational databases, a tuple is one record (one row).
- The information in a database can be thought of as a spreadsheet, with columns (known as fields or attributes) representing different categories of information, and tuples (rows) representing all the information from each field associated with a single record.

SEGMENTATION HARDWARE:



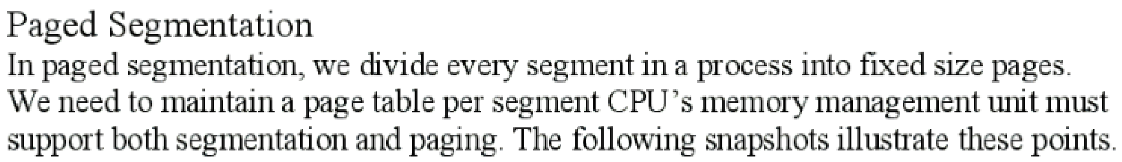
Segmentation Hardware

Fig:4.14 Segmentation Hardware

- Each entry of the segment table has a segment base and segment limit.
- The segment basecontains the starting physical address where the segment resides in the main memory, whereasthe segment limit specifies the length of the segment.
- The main difference between the segmentation and multi-partition schemes is that one programmay consist of several segments.
- The segment table can be kept either in fast registers or inmemory.

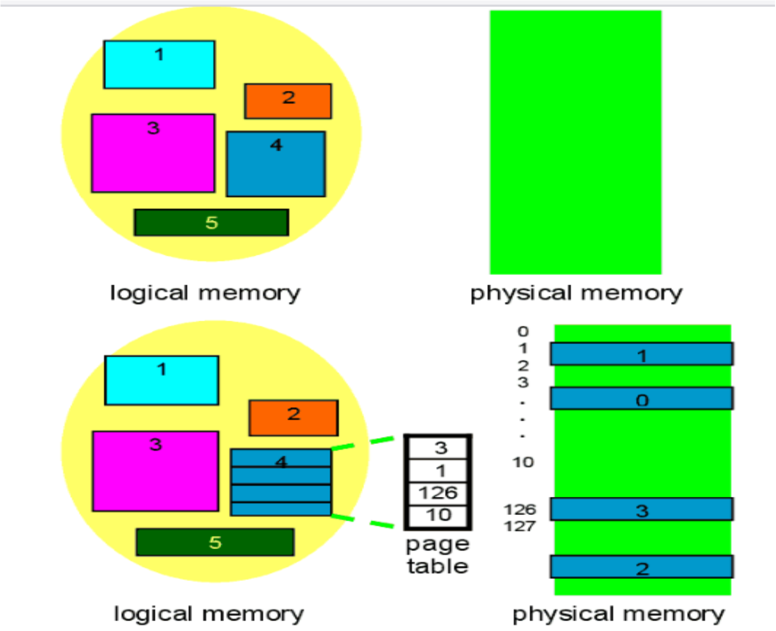
- In case a program consists of several segments, we have to keep them in the memory and a**segment-table base register (STBR)** points to the segment table.Moreover, because thenumber of segments used by a program may vary widely, a **segment-table length register (STLR)** is used.
- One advantage of segmentation is that it automatically provides protection of memory becauseof the segment-table entries (base and limit tuples).
- Segments also can be shared in asegmented memory system. Segmentation may cause external fragmentation.

PAGED-SEGMENTATION



Paged Segmentation

In paged segmentation, we divide every segment in a process into fixed size pages. We need to maintain a page table per segment CPU's memory management unit must support both segmentation and paging. The following snapshots illustrate these points.



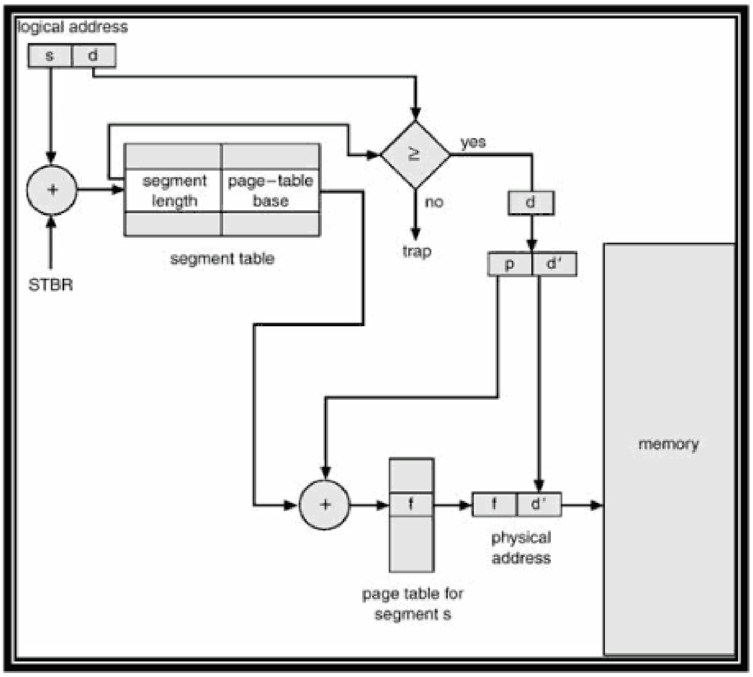
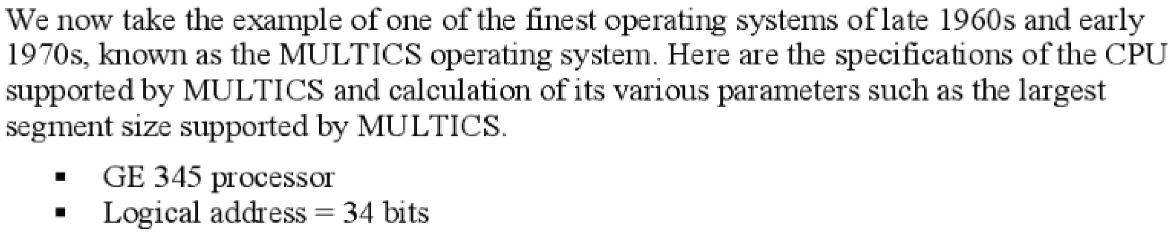


Fig:4.15 Paged Segmentation

MULTICS:


We now take the example of one of the finest operating systems of late 1960s and early 1970s, known as the MULTICS operating system. Here are the specifications of the CPU supported by MULTICS and calculation of its various parameters such as the largest segment size supported by MULTICS.

- GE 345 processor
- Logical address = 34 bits

ADVANTAGES OF SEGMENTATION

- No Internal fragmentation.
- Segment Table consumes less space in comparison to Page table in paging.

DISADVANTAGE OF SEGMENTATION

- As processes are loaded and removed from the memory, the free memory space is broken into little pieces, causing External fragmentation.

PAGE REPLACEMENT

- In a operating systems that use paging for memory management, page replacement algorithm are needed to decide which page needed to be replaced when new page comes in.
- Whenever a new page is referred and not present in memory, page fault occurs and Operating System replaces one of the existing pages with newly needed page.
- Different page replacement algorithms suggest different ways to decide which page to replace. The target for all algorithms is to reduce number of page faults.

PAGE FAULT

- A page fault is a type of interrupt, raised by the hardware when a running program accesses a memory page that is mapped into the virtual address space, but not loaded in physical memory

PAGE REPLACEMENT ALGORITHMS

- There are many page-replacement algorithms. We select a particular replacement algorithm based on the one with the lowest page-fault rate.
- An algorithm is evaluated by running it on a particular string of memory references (called a reference string) and computing the number of page-faults.
- Reference strings are generatedartificially (by a random-number generator, for example) or by tracing a given system andrecording the address of each memory reference.

TYPES OF REPLACEMENT ALGORITHMS

- **FIFO** – First In First Out
- **LRU** – Least Recently Used
- **Optimal Algorithm**
- **LFU -** Least Frequently Used
- **MFU -** Least Frequently Used

FIRST THREE ALGORITHMS - EXAMPLES ARE GIVEN IN ANOTHER PDF

LFU (Least Frequently Used) ALGORITHM

- Replace the least frequently used page
- **Disadvantages:**
  - This algorithm suffers from the situation in which a page is used heavily during the initial phaseof a process, but then is never used again. Since it was used heavily, it has a large count andremains in memory even though it is no longer needed.

MFU (Least Frequently Used) ALGORITHM

- Replace the most frequently used page
- **Disadvantage:**
  - This algorithm is based on the argument that the page with the smallest count was probablyjust brought in and has yet to be used. Neither MFU nor LFU replacement is common. Theirimplementation is fairly expensive.

VIRTUAL MEMORY

- Virtual memory is a technique that allows the execution of processes that may not becompletely in memory.
- In many cases, in the course of execution of a program, some part of theprogram may never be executed.
- The advantages of virtual memory are:
  - Users would be able to write programs whose logical address space is greater than thephysical address space.
  - More programs could be run at the same time thus increasing the degree ofmultiprogramming.
  - Less I/O would be needed to load or swap each user program into memory, so eachprogram would run faster.

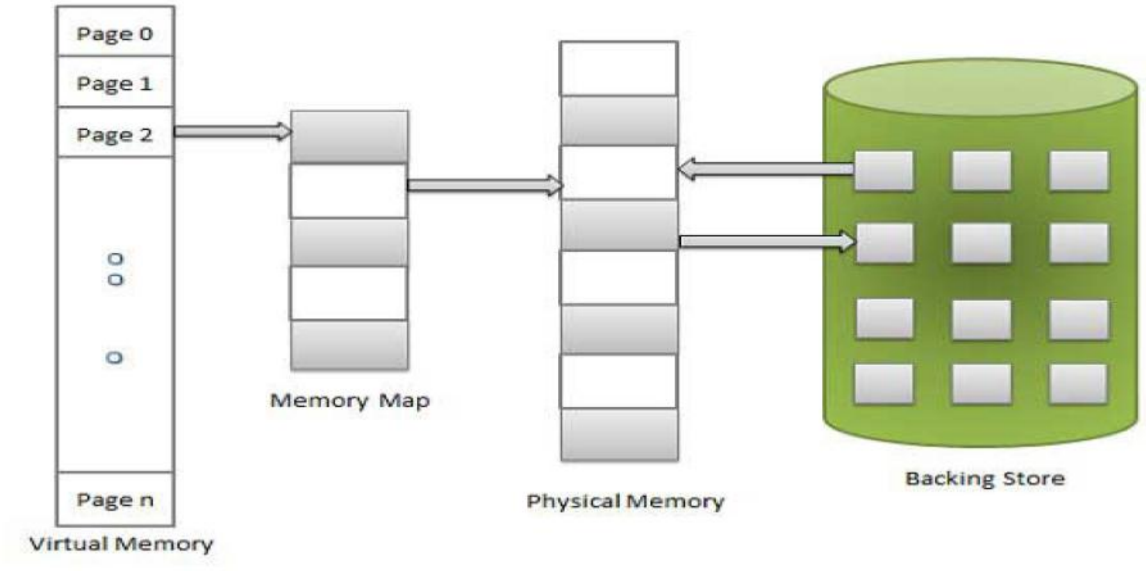


Fig:4.16 Virtual Memory Management

- Virtual memory is the separation of logical memory from physical memory.
- Virtual memory iscommonly implemented by demand paging. It can also be implemented in a segmentationsystem.

DEMAND PAGING

- A demand-paging system is similar to a paging system with swapping.
- When a process is to be swapped in, the pager guesses which pages will be used before the process is swapped out again.
- Instead of swapping in a whole process, the pager brings only those necessary pages into the memory.
- Thus, it avoids reading into memory pages that will not be used anyway, decreasing the swap time and the amount of physical memory needed.

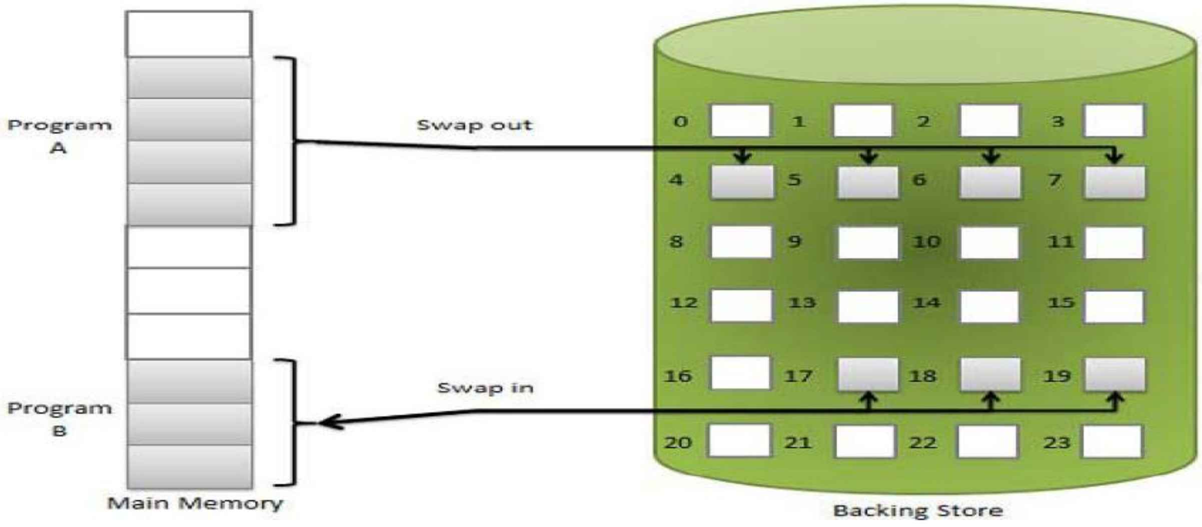


Fig:4.17 Demand Paging –Swap-in and out

- To distinguish between those pages that are in memory and those that are on disk, we use an invalid-valid bit which is a field in each page-table entry.
- When this bit is set to "valid", this value indicates that the associated page is both legal and in memory.
- If the bit is set to "invalid", itindicates that the page is either not valid (i.e., not in logical address space of the process), or isvalid but is currently on the disk.
- The page-table entry for a page that is not currently in memoryis simple marked invalid, or contains the address of the page on disk.
- Access to a page markedinvalid causes a page-fault trap. The procedure for handling page fault is given below:

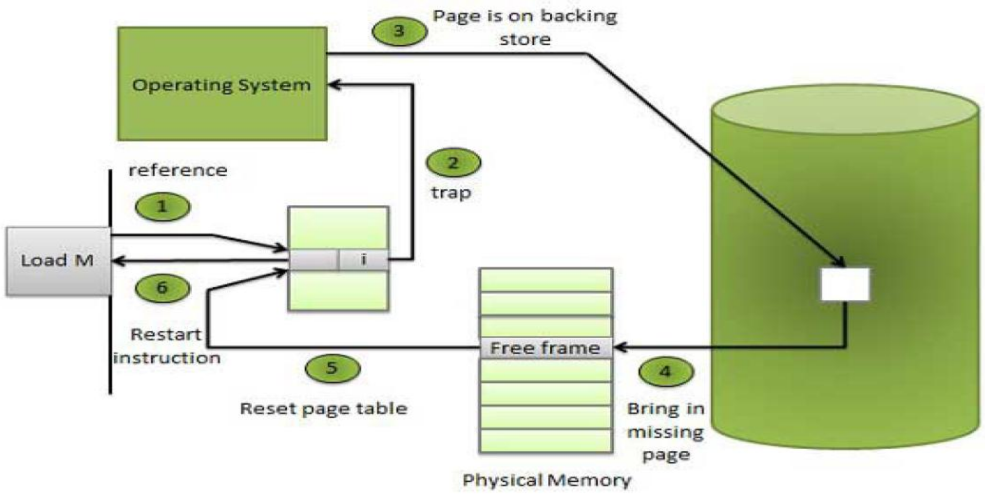


Fig:4.18 Demand Paging Process

ADVANTAGES

- Large virtual memory.
- More efficient use of memory.
- Unconstrained multiprogramming.
- There is no limit on degree of multiprogramming.

DISADVANTAGES

- Number of tables and amount of processor overhead for handling page interrupts aregreater than in the case of the simple paged management techniques.
- Due to the lack of explicit constraints on jobs address space size.

PURE DEMAND PAGING

- In the extreme case, we could start executing a process with no pages in memory.
- When theoperating system set the instruction pointer to the first instruction of the process which is on a non-memory-resident page, the process would immediately fault for the page.
- After this page was brought into memory, the process would continue to execute, faulting as necessary until every page that is needed was actually in memory.
- Then, it could execute with no more faults. This scheme is called **pure demand paging.**

DEMAND PAGING vs ANTICIPATORY PAGING

DEMAND PAGING

- When a process first executes, the system loads into main memory the page that contains its first instruction
- After that, the system loads a page from secondary storage to main memory onlywhen the process explicitly references that page
- Requires a process to accumulate pages one at a time

ANTICIPATORY PAGING

- Operating system attempts to predict the pages a process will need and preloads these pages when memory space is available
- Anticipatory paging strategies must be carefully designed so that overheadincurred by the strategy does not reduce system performance.

SATHYABAMA
INSTITUTE OF SCIENCE AND TECHNOLOGY
(DEEMED TO BE UNIVERSITY)
Accredited "A" Grade by NAAC | 12B Status by UGC | Approved by AICTE
www.sathyabama.ac.in

SCHOOL OF COMPUTING

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

UNIT – V – Operating System – SBS1206

OPERATING SYSTEM:

UNIT V **(9 Hrs)**

File System: File Concepts - Access Methods - Directory Structures - Protection Consistency Semantics - File System Structures - Allocation Methods - Free Space Management.

UNIT 5

FILE SYSTEMS

FILE CONCEPT

- A file is a named collection of related information that is recorded on secondary storage.
- Data can NOT be written to secondary storage unless they are within a file.

FILE STRUCTURE

- A text file is a sequence of characters organized into lines.
- A source file is a sequence of subroutines and function.
- An object file is a sequence of bytes organized into blocks understandable by thesystem's linker
- An executable file is a series of code sections that the loader can bring into memory andexecute.

FILE ATTRIBUTES

- **Name** – only information kept in human-readable form
- **Identifier** – unique tag (number) identifies file within file system
- **Type** – needed for systems that support different types
- **Location** – pointer to file location on device
- **Size** – current file size
- **Protection** – controls who can do reading, writing, executing
- **Time, date, and user identification** – data for protection, security, and usage monitoring
- Information about files are kept in the directory structure, which is maintained on the disk

FILE OPERATIONS

- The operating system provides system calls to create, write, read, reposition, delete, and truncate files. We look at what the operating system must do for each of these basic file operations.

- **CREATING A FILE**
  - Firstly, space in the file system must be found for the file. Secondly, an entryfor the new file must be made in the directory. The directory entry records the name of thefile and the location in the system.
- **WRITING A FILE.**
  - In the system call for writing in a file, we have to specify the name of the fileand the information to be written to the file.
  - The operating system searches the directory tofind the location of the file.
  - The system keeps a write pointer to the location in the file wherethe next write is to take place.
  - The write pointer must be updated whenever a write occurs.
- **READING A FILE**
  - To read a file, we make a system call that specifies the name of the file andwhere (in memory) the next block of the file should be put.
  - As in writing, the systemsearches the directory to find the location of the file, and the system keeps a read pointer tothe location in the file where the next read is to take place.
  - The read pointer must beupdated whenever a read occurs.
- **REPOSITIONING WITHIN A FILE**
  - In this operation, the directory is searched for the named file, and the current-file-position is set to a given value. This file operation is called a **FILE SEEK**.
- **DELETING A FILE**
  - We search the directory for the appropriate entry.
  - Having found it, we release all the space used by this file and erase the directory entry**.**

- **TRUNCATING A FILE**
  - This operation is used when the user wants to erase the contents of the file but keep the attributes the file intact

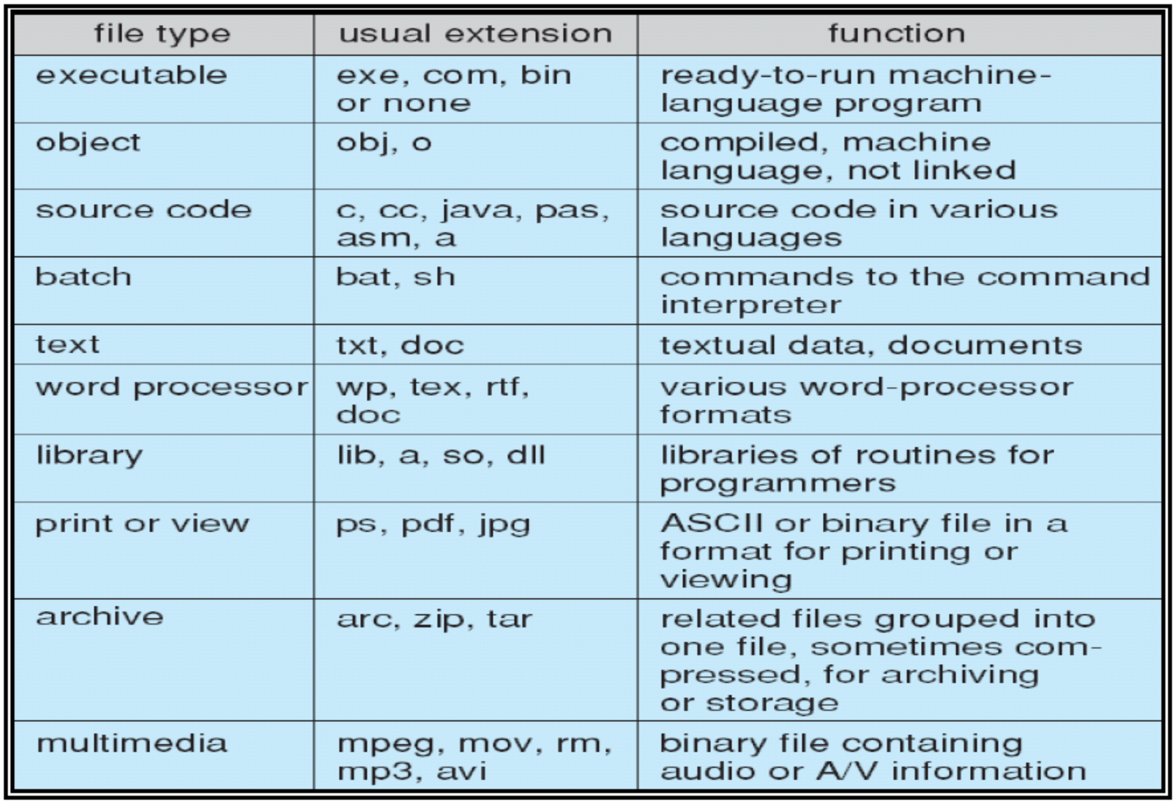
MEMORY-MAPPED FILES

- Some operating systems allow mapping sections of file into memory on virtual-memory systems.
- It allows part of the virtual address space to be logically associated with a section of a file.
- Reads and writes to that memory region are then treated as reads and writes to the file,greatly simplifying the file use.
- Closing the file results in all the memory-mapped data beingwritten back to the disk and removed from the virtual memory of the process.

OPEN FILES

- **File pointer:** pointer to last read/write location, per process that has the file open
- **File-open count:** the counter tracks the number of opens and closes, and reaches zero on the last close. The system can then remove the entry.
- **Disk location of the file**: the info needed to locate the file on disk.
- **Access rights**: per-process access mode information so OS can allow or denysubsequent I/O request

FILE TYPES – NAME, EXTENSION



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

ACCESS METHODS

- **Sequential Access**

read next

write next

reset

no read after last write

(rewrite)

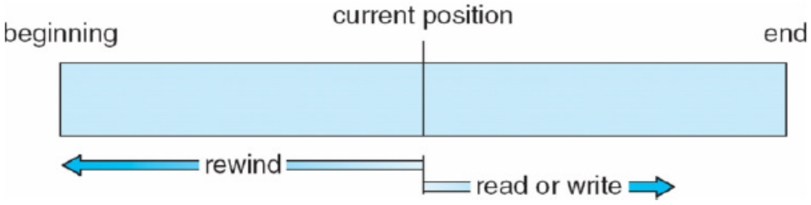
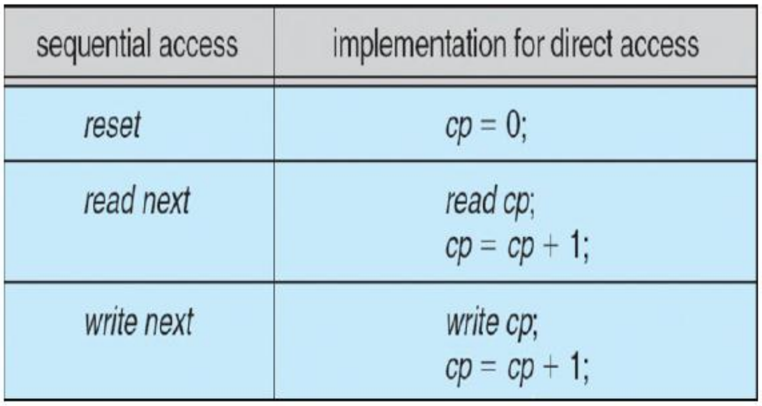


Fig:5.1 Sequential Access

- **Direct Access**

```
read n
write n
position to n
          read next
          write next
rewrite n
n = relative block number
```

- **Simulation of Sequential Access on Direct-access File**



sequential access	implementation for direct access
reset	*cp* = 0;
read next	*read cp*; *cp* = *cp* + 1;
write next	*write cp*; *cp* = *cp* + 1;

Cp=> current position

- **Index and Relative Files**

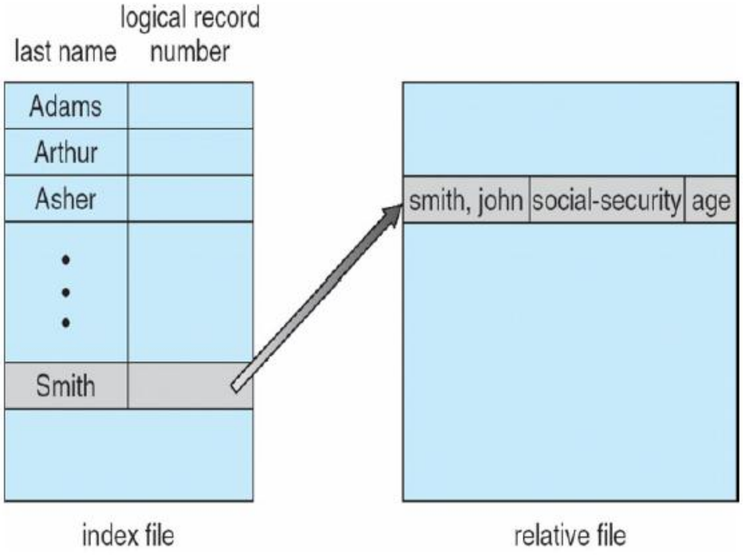


Fig:5.2 Indexing & Relative Files

- The index contains pointers to the various blocks.
- To find a record in the file, we first search the index and then use the pointer to access the file directly and to find the desired record

DISK STRUCTURE

- Disk can be subdivided into partitions
- Disks or partitions can be **Redundant Arrays of Independent Disks (RAID)** protectedagainst failure
- Disk or partition can be used raw – without a file system, or formatted with a file system
- Partitions also known as minidisks, slices
- Entity containing file system known as a volume
- Each volume containing file system also tracks that file system's info in device directoryor volume table of contents
- There are five commonly used directorystructures:
    - **Single-Level Directory**
    - **Two-Level Directory**
    - **Tree-Structure Directories**
    - **Acyclic-Graph Directories**

- **General Graph Directories**

A TYPICAL FILE-SYSTEM ORGANIZATION

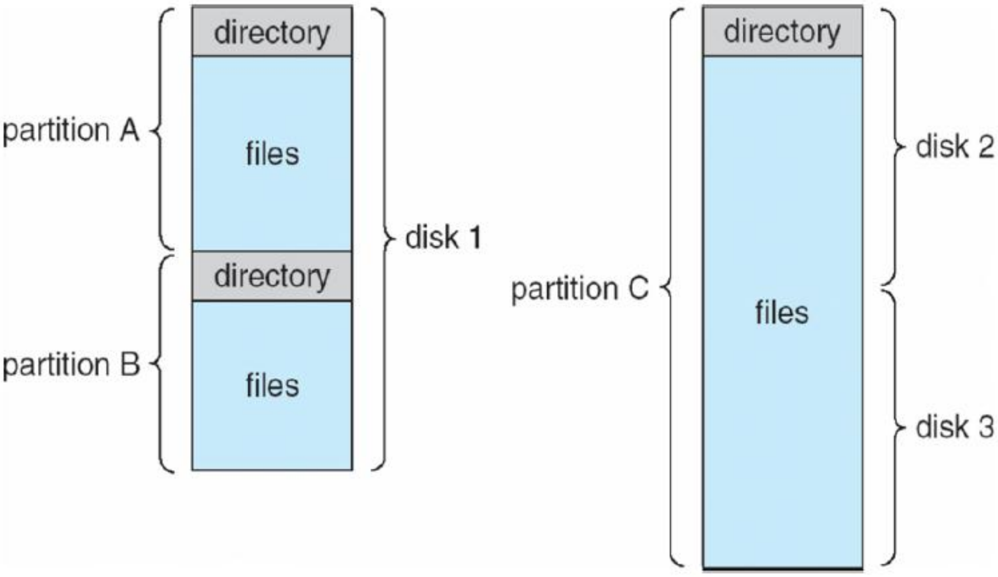


Fig:5.3 A Typical File-System Organization

Operations Performed on Directory

- Search for a file
- Create a file
- Delete a file
- List a directory
- Rename a file
- Traverse the file system

SINGLE-LEVEL DIRECTORY

- All files are contained in the same directory.
- It is difficult to maintain file name uniqueness.
- CP/M-80 and early version of MS-DOS use this directory structure.

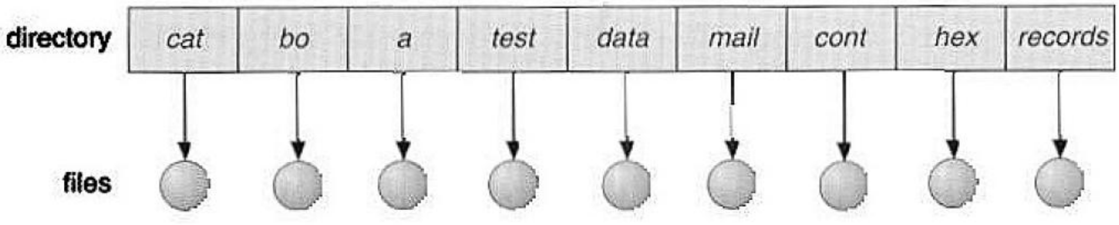


Fig:5.4 Single-Level Directory

TREE-STRUCTURED DIRECTORIES

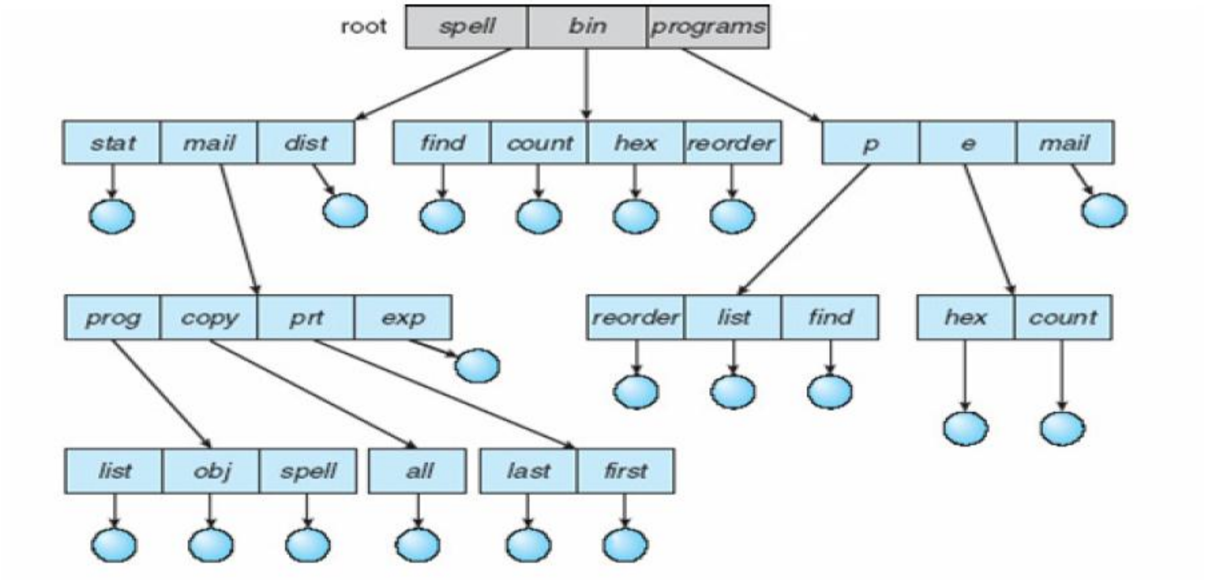


Fig:5.5 Tree-Structure Directory

- **ABSOLUTE PATH:** begins at the root and follows a path down to the specified file.
  - root/spell/mail/prt/first
- **RELATIVE PATH:** defines a path from the current directory.
  - prt/first given root/spell/mail ascurrent path

ACYCLIC-GRAPH DIRECTORY

- This type of directories allows a file/directory to be shared by multipledirectories.
- This is different from two copies of the same file or directory.
- An acyclic-graph directory is more flexible than a simple tree structure.

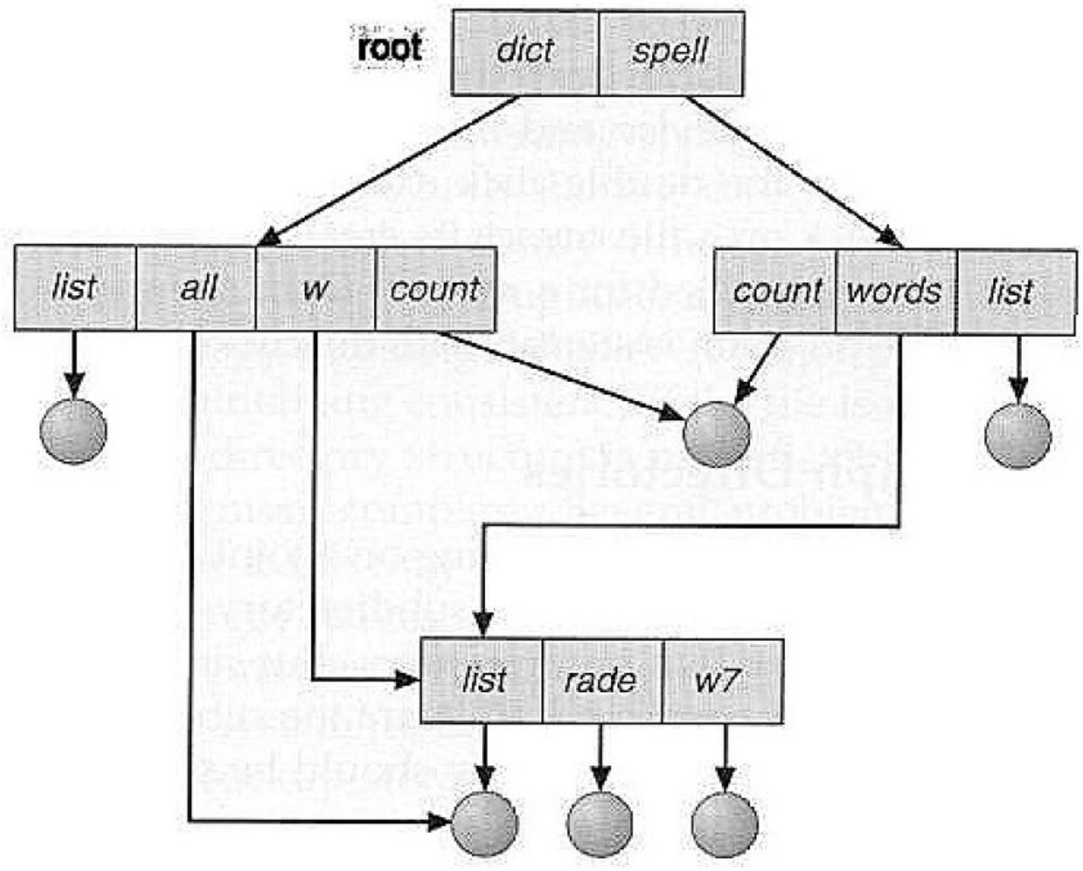


Fig:5.6 Acyclic Graph Directory

GENERAL GRAPH DIRECTORY

- It is easy to traverse the directories of a tree or an acyclic directory system.
- However, if links are added arbitrarily, the directory graph becomes arbitrary and maycontain cycles
- Creating a new file is done in current directory
  - Delete a file
    - rm<file-name>

- Creating a new subdirectory is done in current directory
    - mkdir<dir-name>

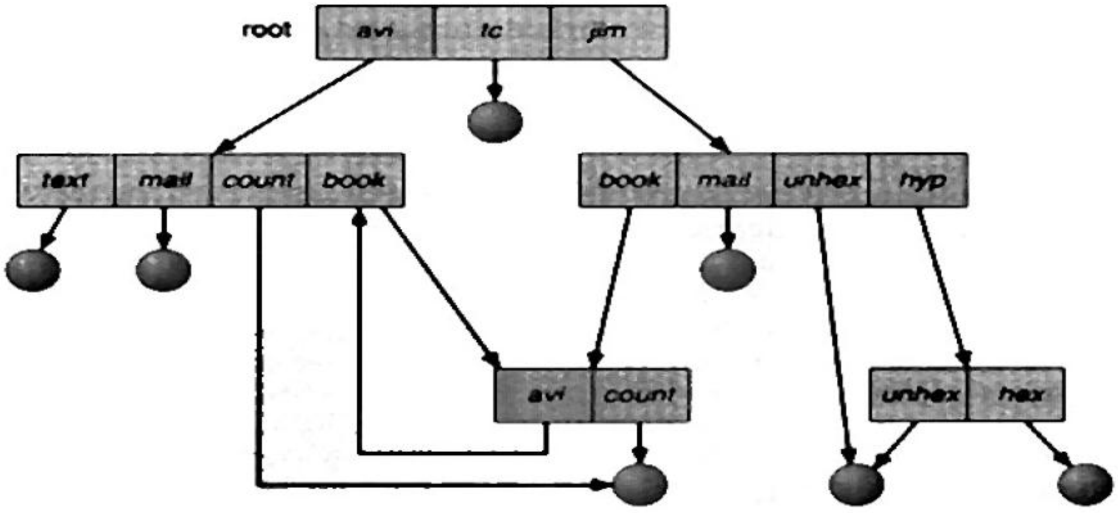
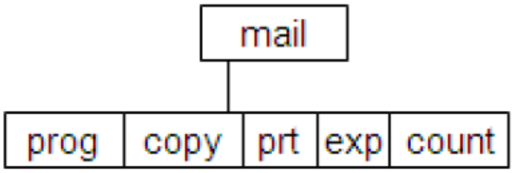


Fig:5.7 General Graph Directory

Example:

if in current directory /mail

mkdir count



Deleting “mail” ⇒ deleting the entire subtree rooted by “mail”

Option1: do not delete a directory unless it is empty, such as MS-DOS

Option2: delete all files in that directory, such as UNIX rm command with r option

Only one file exists. Any changes made by one person are immediately visible to the other.

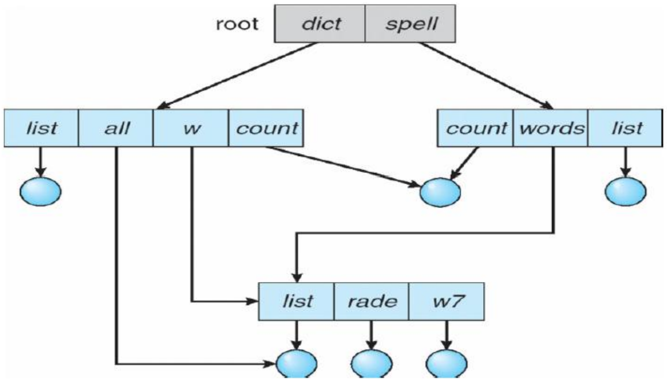


Fig:5.8 General Graph Directory

- Efficient searching
- Grouping Capability
  - pwd
  - cd /spell/mail/prog
  - New directory entry type
  - Link – another name (pointer) to an existing file
  - Resolve the link – follow pointer to locate the file

FILE SHARING

- When a file is shared by multiple users, how can we ensure its consistency
- If multiple users are writing to the file, should all of the writers be allowed to write?Or,should the operating system protect the user actions from each other?
- This is the file consistency semantics

FILE CONSISTENCY SEMANTICS

- Consistency semantics is a characterization of the system that specifies the semantics ofmultiple users accessing a shared file simultaneously.
- Consistency semantics is an important criterion for evaluating any file system that supports filesharing.
- There are three commonly used semantics

- Unix semantics
- Session Semantics
- Immutable-Shared-Files Semantics

Unix Semantics

- Writes to an open file by a user are visible immediately to other users have the file openat the same time.All users share the file pointer.
- A file has a single image that interleaves all accesses, regardless of their origin

Session Semantics

- Writes to an open file by a user are not visible immediately to other users that have thesame file open simultaneously
- Once a file is closed, the changes made to it are visible only insessions started later.
- Already-open instances of the file do not affect these changes
- Multiple users are allowed to perform both readand write concurrently on their image of the file without delay.
- The Andrew File System (AFS) uses this semantics.

Immutable-Shared-Files Semantics

- Once a file is declared as shared by its creator, it cannot be modified.
- An immutable file has two important properties**:**
  - **Its name may not be used**
  - **Its content may not be altered**

FILE PROTECTION

- We can keep files safe from physical damage (i.e.,reliability) and improper access (i.e.,protection).
- Reliability is generally provided by backup.
- The need for file protection is a directresult of the ability to access files.
- Access control may be a complete protection by denyingaccess. Or, the access may be controlled.

TYPES OF ACCESS

- **Read:** read from the file
- **Write:** write or rewrite the file
- **Execute**: load the file into memory and execute it
- **Append**: write new info at the end of a file
- **Delete:** delete a file
- **List:** list the name and attributes of the file

FILE-SYSTEM STRUCTURE

- Logical storage unit
- Collection of related information

Layered File System

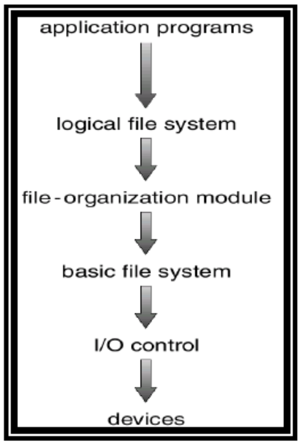
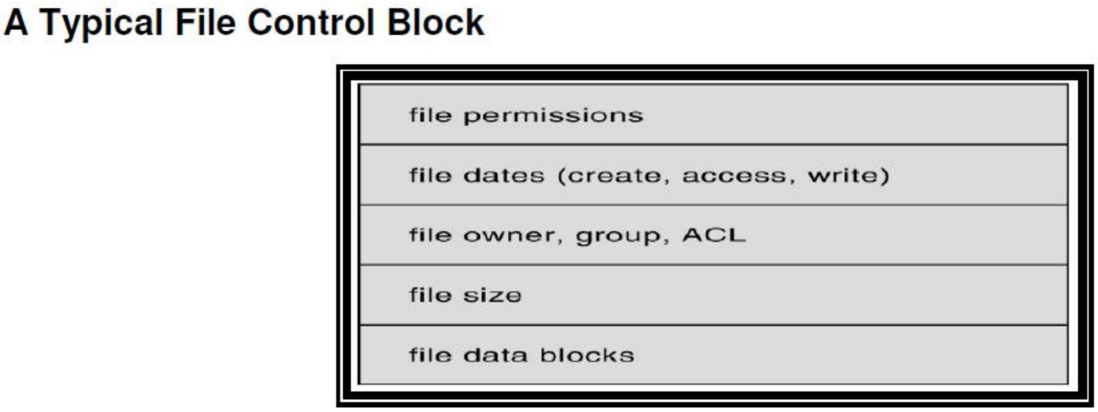


Fig:5.9 Layered File System



A Typical File Control Block

file permissions

file dates (create, access, write)

file owner, group, ACL

file size

file data blocks

Fig:5.10 A Typical File Control Block

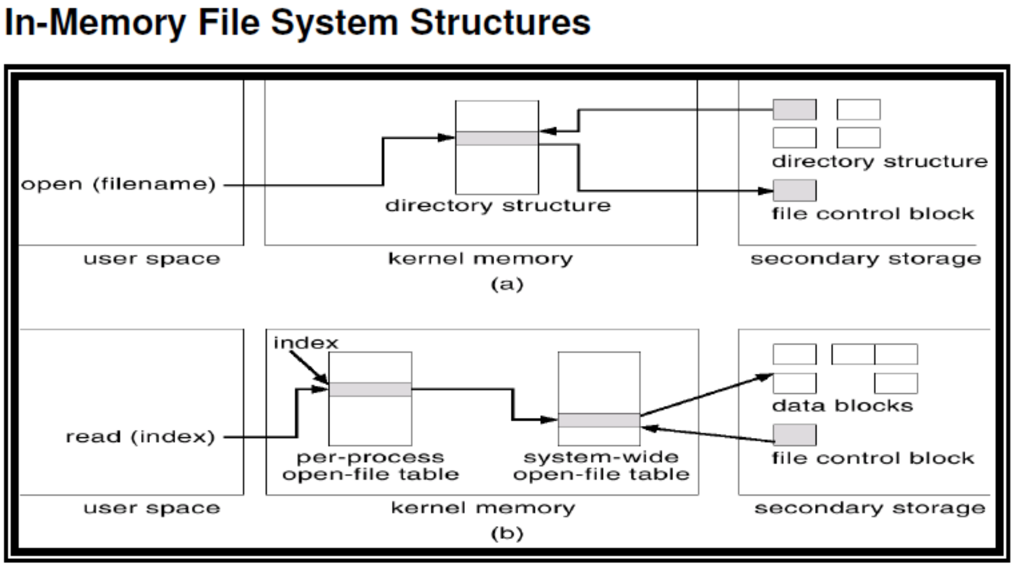


Fig:5.11 In-Memory File System Structures

VIRTUAL FILE SYSTEMS

- Virtual File Systems (VFS) provide an object-oriented way of implementing file systems.
- VFS allows the same system call interface (the API) to be used for different types of filesystems.

- The API is to the VFS interface, rather than any specific type of file system.

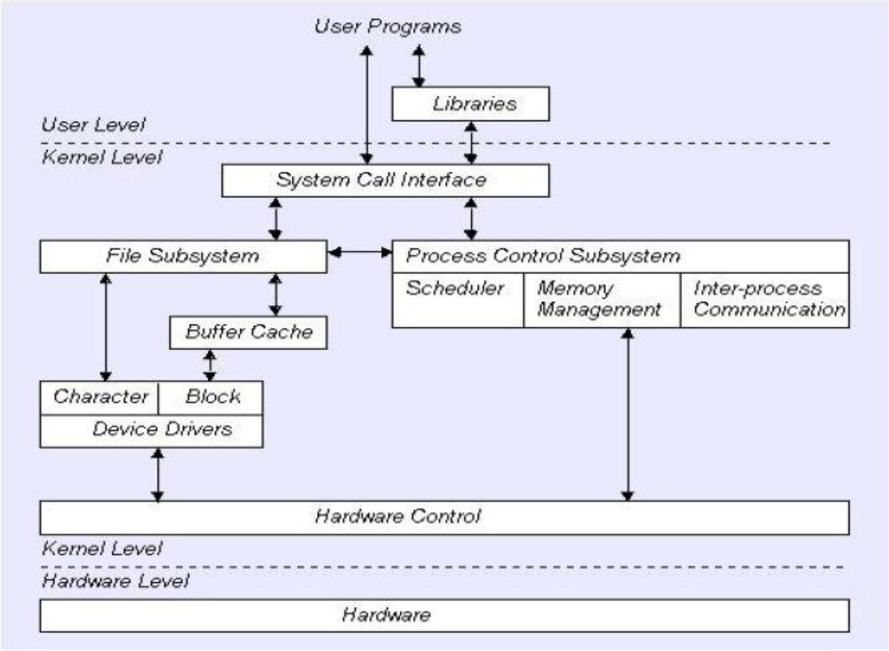


Fig:5.12 Virtual File System

Schematic View of Virtual File System

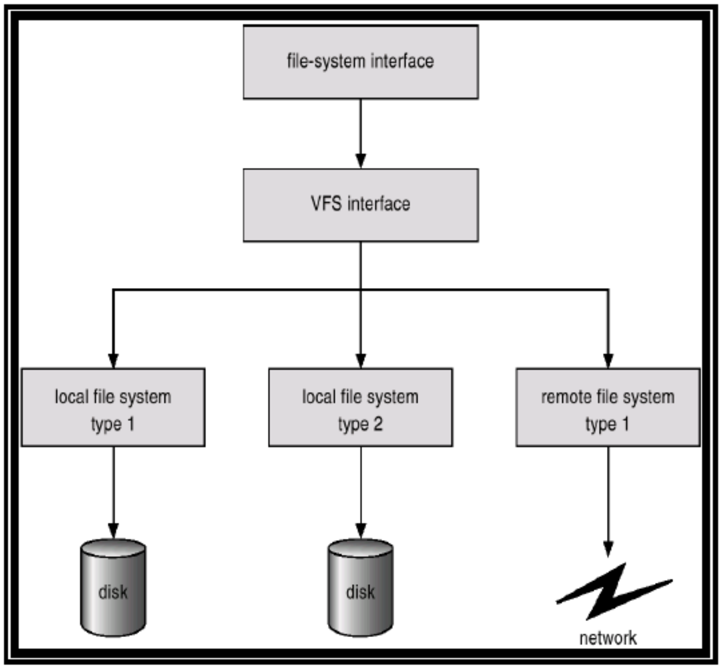


Fig:5.13 Schematic View of Virtual File System

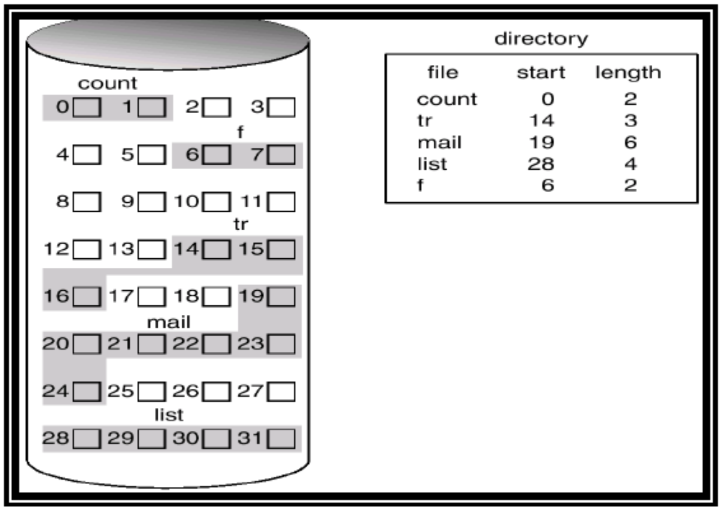
ALLOCATION METHODS

- An allocation method refers to how disk blocks are allocated for files:
    - Contiguous allocation
    - Linked allocation
    - Indexed allocation

Contiguous Allocation

- Each file occupies a set of contiguous blocks on the disk.
- Simple – only starting location (block #) and length (number of blocks) are required.
- Random access.
- Wasteful of space (dynamic storage-allocation problem).
- Files cannot grow.

Contiguous Allocation of Disk Space

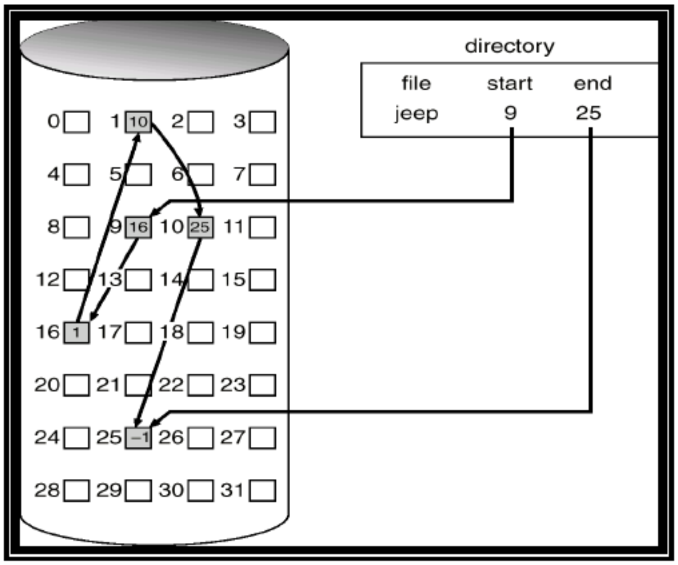


directory

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

Fig:5.14 Contiguous Allocation of Disk Space

Linked Allocation



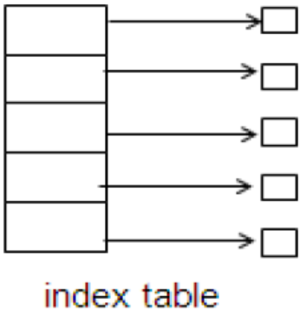
directory

file	start	end
jeep	9	25

Fig:5.15 Linked Allocation

- Simple – need only starting address
- Free-space management system – no waste of space
- No random access
- Space waste for pointer (e.g. 4byte of 512 B)
- Reliability

Indexed Allocation



- Brings all pointers together into the index block.

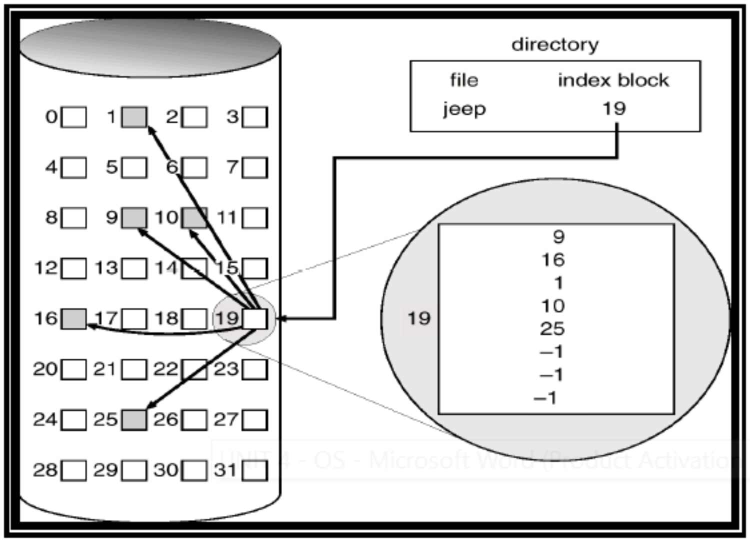


Fig:5.16 Indexed Allocation

- Need index table

- Random access
- Dynamic access without external fragmentation, but have overhead of index block

Indexed Allocation – Mapping

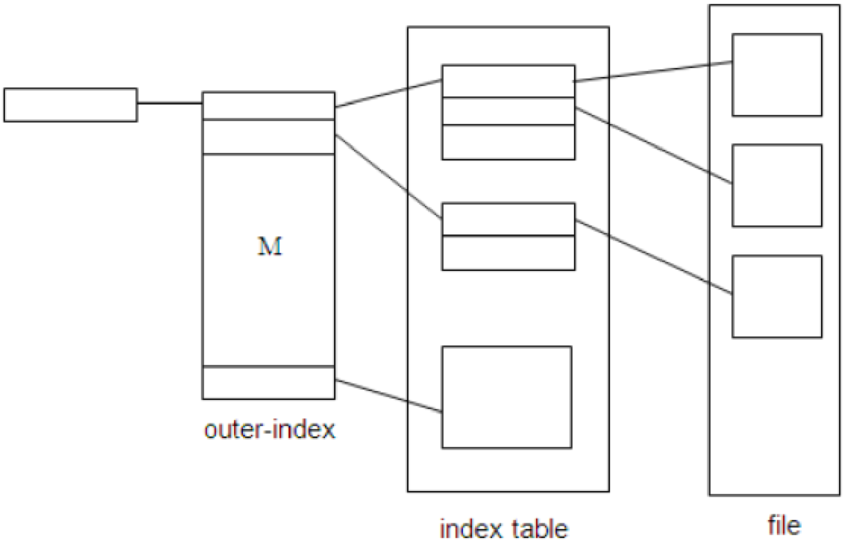
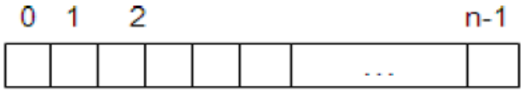


Fig:5.17 Indexed Allocation – Mapping

Free-Space Management

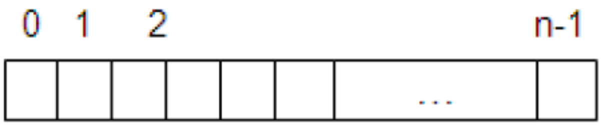
Bit vector (*n* blocks)



bit[*i*] = 678 0 ⇒ block[*i*] free
1 ⇒ block[*i*] occupied

Block number calculation

(number of bits per word) *
(number of 0-value words) +
offset of first 1 bit



bit[*i*] = 678 $0 \Rightarrow$ block[*i*] free

$1 \Rightarrow$ block[*i*] occupied

- Bit map requires extra space. Example:

  block size = 2^{12} bytes (4 K)

  disk size = 2^{30} bytes (1 gigabyte)

  $n = 2^{30}/2^{12} = 2^{18}$ bits (or 32K bytes)

  – Easy to get contiguous files
- Linked list (free list)
  – Cannot get contiguous space easily
  – No waste of space
- Grouping
  – Large free blocks can be quickly found
- Counting
- Need to protect:
  – Pointer to free list
  – Bit map

- Must be kept on disk
- Copy in memory and disk may differ.

- Linked Free Space List on Disk

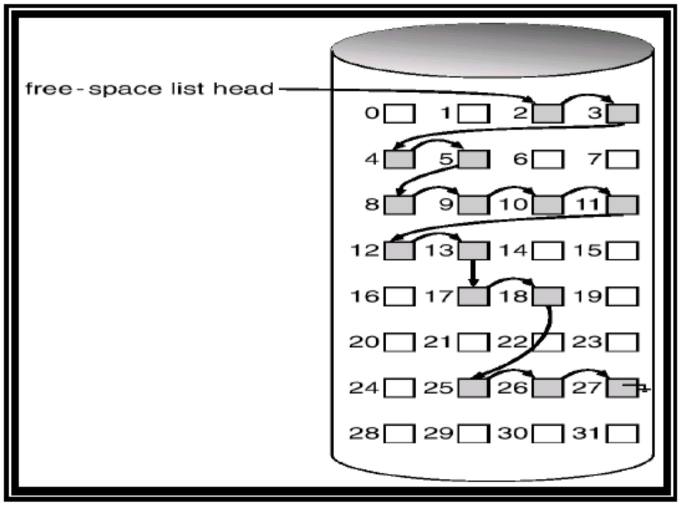


Fig:5.18 Linked Free Space List on Disk

EFFICIENCY AND PERFORMANCE

- **Efficiency dependent on:**

  disk allocation and directory algorithms

  types of data kept in file's directory entry

- **Performance**
  - disk cache – separate section of main memory for frequently used blocks
  - free-behind and read-ahead – techniques to optimize sequential access
  - improve PC performance by dedicating section of memory as virtual disk, orRAM disk.

Various Disk-Caching Locations

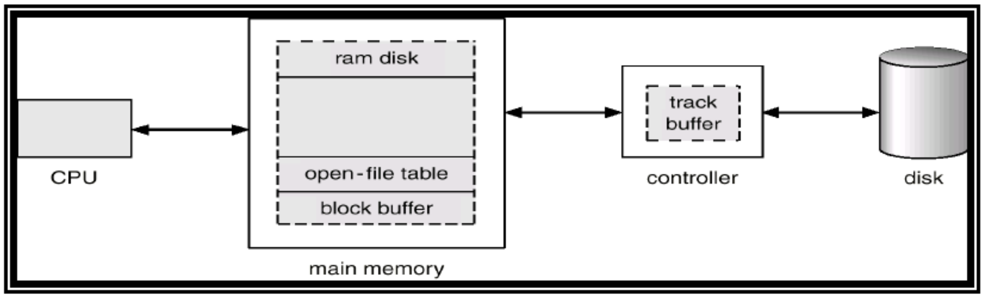


Fig:5.19 Various Disk-Caching Locations

PAGE CACHE

- A page cache caches pages rather than disk blocks using virtual memory techniques.
- Memory-mapped I/O uses a page cache.
- Routine I/O through the file system uses the buffer (disk) cache.
- This leads to the following figure.

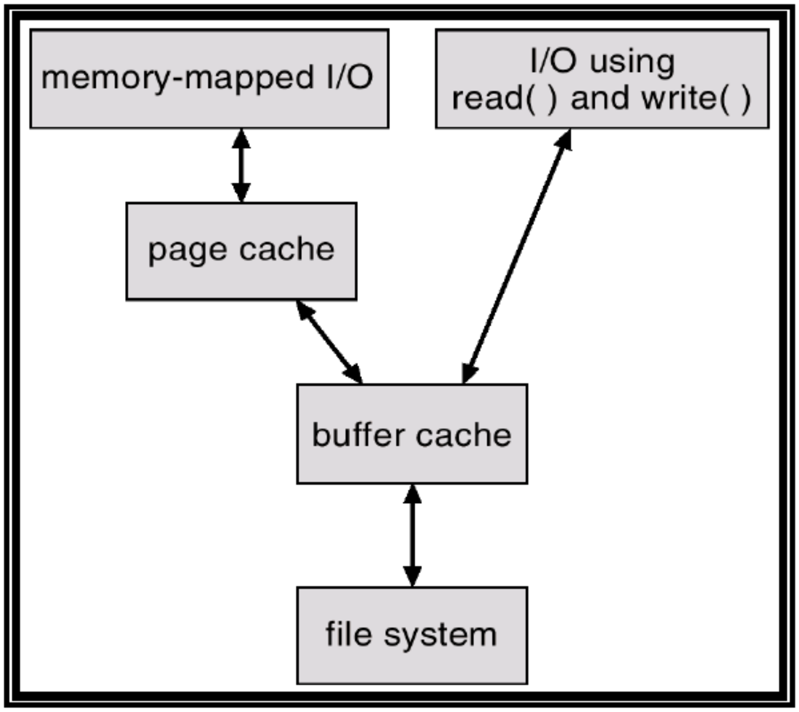


Fig:5.20 Page Cache

Unified Buffer Cache

- A unified buffer cache uses the same page cache to cache both memory-mapped pagesand ordinary file system I/O.

I/O Using a Unified Buffer Cache

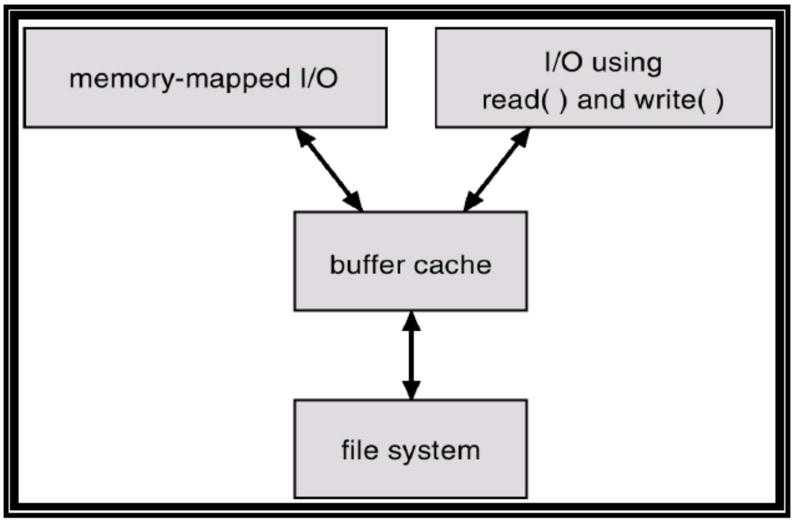


Fig:5.21 I/O Using a Unified Buffer Cache

RECOVERY

- Consistency checking – compares data in directory structure with data blocks on disk,and tries to fix inconsistencies.
- Use system programs to back up data from disk to another storage device (floppy disk,magnetic tape).
- Recover lost file or disk by restoring data from backup.

LOG STRUCTURED FILE SYSTEMS

- Log structured (or journaling) file systems record each update to the file system as atransaction.
- All transactions are written to a log. A transaction is considered committed once it iswritten to the log.
- The transactions in the log are asynchronously written to the file system. When the filesystem is modified, the transaction is removed from the log.
- If the file system crashes, all remaining transactions in the log must still be performed

THE SUN NETWORK FILE SYSTEM (NFS)

- An implementation and a specification of a software system for accessing remote filesacross LANs (or WANs).
- The implementation is part of the Solaris and SunOS operating systems running on Sunworkstations using an unreliable datagram protocol (UDP/IP protocol and Ethernet.
- NFS is designed to operate in a heterogeneous environment of different machines operating systems, and network architectures; the NFS specifications independent ofthese media.
- The NFS specification distinguishes between the services provided by a mountmechanism and the actual remote-file-access services.

Three Major Layers of NFS Architecture

- UNIX file-system interface (based on the open, read, write, and close calls, and filedescriptors).
- Virtual File System (VFS) layer – distinguishes local files from remote ones, and localfiles are further distinguished according to their file-system types.
- The VFS activates file-system-specific operations to handle local requests according totheir file-system types.
- Calls the NFS protocol procedures for remote requests.
- NFS service layer – bottom layer of the architecture; implements the NFS protocol

Schematic View of NFS Architecture

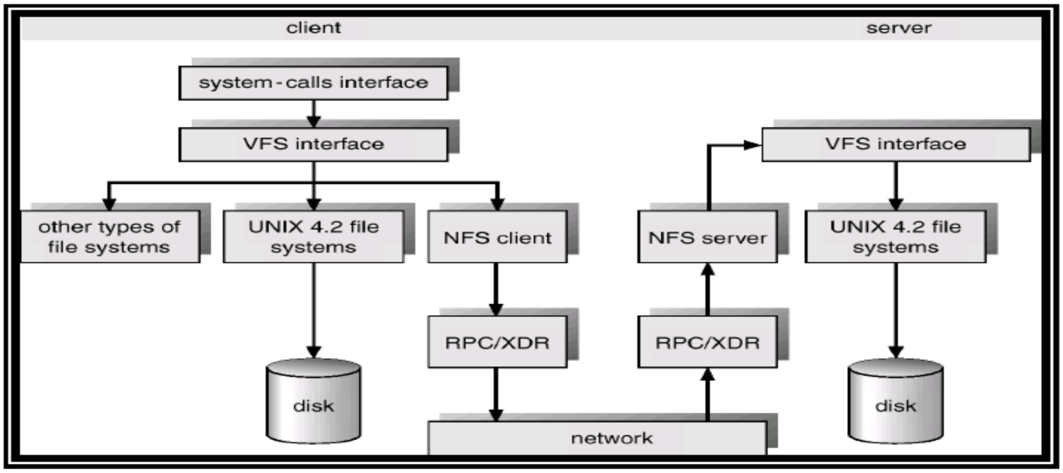


Fig:5.22 Schematic View of NFS Architecture

NFS PROTOCOL

- Provides a set of remote procedure calls for remote file operations. The proceduressupport the following operations:
  - searching for a file within a directory
  - reading a set of directory entries
  - manipulating links and directories
  - accessing file attributes
  - reading and writing files
- NFS servers are stateless; each request has to provide a full set of arguments.
- Modified data must be committed to the server's disk before results are returned to theclient (lose advantages of caching).
- The NFS protocol does not provide concurrency control mechanisms