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

II Year IIIrd Semester BSC CS (2020-2023)

Syllabus with Course objectives and Course outcomes

COURSE OBJECTIVES :

- To learn how computers work and to learn about system design (performance vs. simplicity, HW vs. SW, etc.)
- To have an overview of different types of operating systems.
- To learn and implement the concept of process management.
- To observe the concept of storage management.
- To understand the concept of I/O and file systems.
- To learn the basics of Linux Programming

UNIT 1 **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 2 **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 3 **9 Hrs.**

Deadlocks: Characterization- Methods for Handling Deadlocks - Deadlock Prevention - Avoidance - Detection - Recovery.

UNIT 4 **9 Hrs.**

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

UNIT 5 **9 Hrs.**

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

Max. 45 hours

COURSE OUTCOMES :

CO1 - Understand the fundamental components of a computer operating system and how computing resources are managed by the operating system.

CO2 - Apply the concepts of CPU scheduling, synchronization and deadlocks in real computing problems. CO3 - Demonstrate the different memory and I/O management techniques used in Operating Systems.

CO4 - Have practical exposure to the concepts of semaphores and monitors for process synchronization.

CO5 - Create design and construct the following OS components: Schedulers, Memory management systems in the modern operating system.

CO6 - Understand file system structure and implement a file system such as FAT.

Text /Reference Book:

1. A. Silberschatz P.B.Galvin, Gange., Operating System Concepts, 6th Edition., Addison-Wesley Publishing Co., 2002.
2. William Stallings, Operating Systems, Fourth Edition, PHI.
3. Andrew S Tanenbaum, Operating Systems: Design and Implementation, Third Edition,

UNIT – I - Operating Systems – SBSA 1301

UNIT 1

INTRODUCTION

OS is a program that acts as an intermediary between a user of a computer and the computer hardware

Operating system goals:

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

COMPUTER SYSTEM ARCHITECTURE

- Computer system can be divided into four components:
- Hardware – provides basic computing resources
    - CPU, memory, I/O devices
- Operating system
    - Controls and coordinates use of hardware among various applications and users
- Application programs – define the ways in which the system resources are used to solve the computing problems of the users
    - Word processors, compilers, web browsers, database systems, video games
- Users
    - People, machines, other computers

- Depends on the point of view
- Users want convenience, ease of use and good performance
    - Don't care about resource utilization
- But shared computer such as mainframe or minicomputer must keep all users happy
- Users of dedicate systems such as workstations have dedicated resources but frequently use shared resources from servers
- Handheld computers are resource poor, optimized for usability and battery life
- Some computers have little or no user interface, such as embedded computers in devices and automobiles
- OS is a resource allocator
    - Manages all resources

  - Decides between conflicting requests for efficient and fair resource use
- OS is a control program
  - Controls execution of programs to prevent errors and improper use of the computer

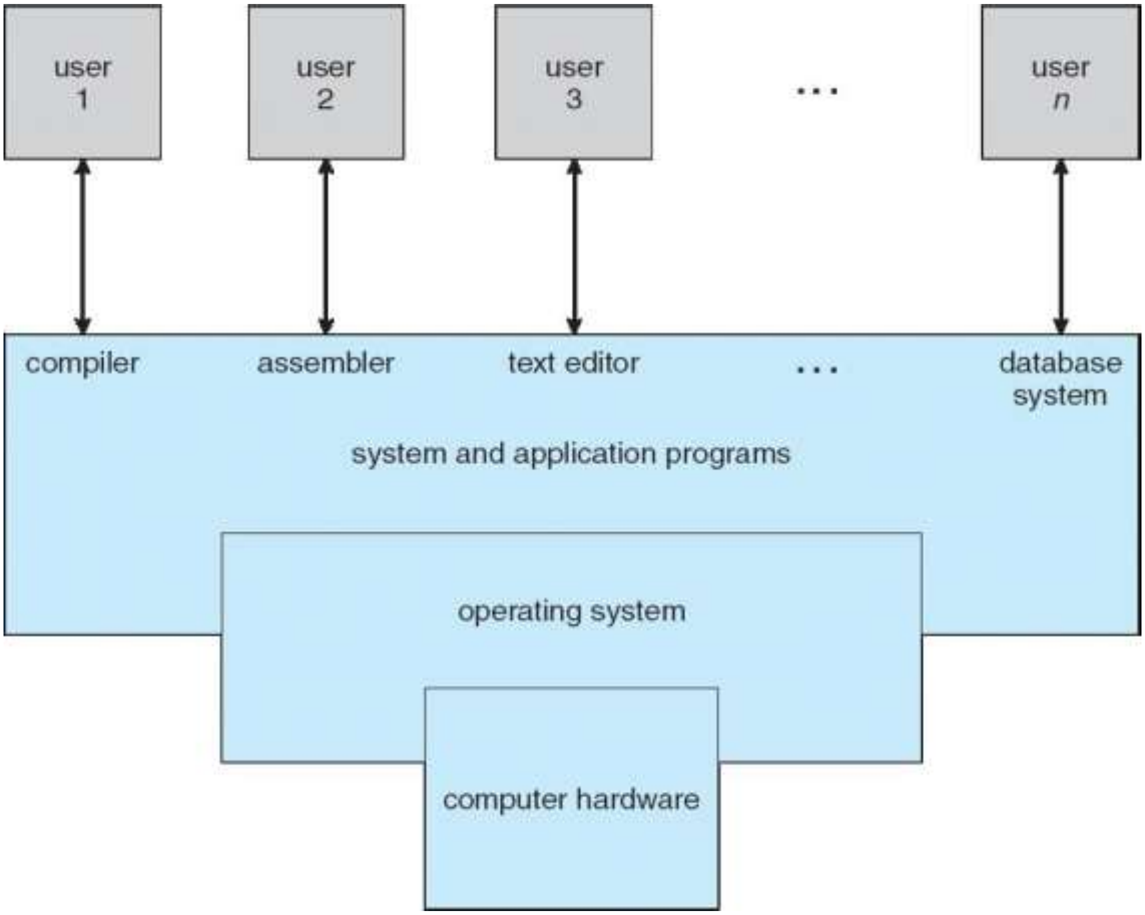


Figure 1: Abstract View of System Component

No universally accepted definition

- –Everything a vendor ships when you order an operating system| is a good approximation
  - But varies wildly
- –The one program running at all times on the computer| is the **kernel**.
- Everything else is either
  - a system program (ships with the operating system) , or
  - an application program.

COMPUTER STARTUP

- **Bootstrap program** is loaded at power-up or reboot
  - Typically stored in ROM or EPROM, generally known as **firmware**
  - Initializes all aspects of system
  - Loads operating system kernel and starts execution

COMPUTER-SYSTEM OPERATION

- One or more CPUs, device controllers connect through common bus providing access to shared memory

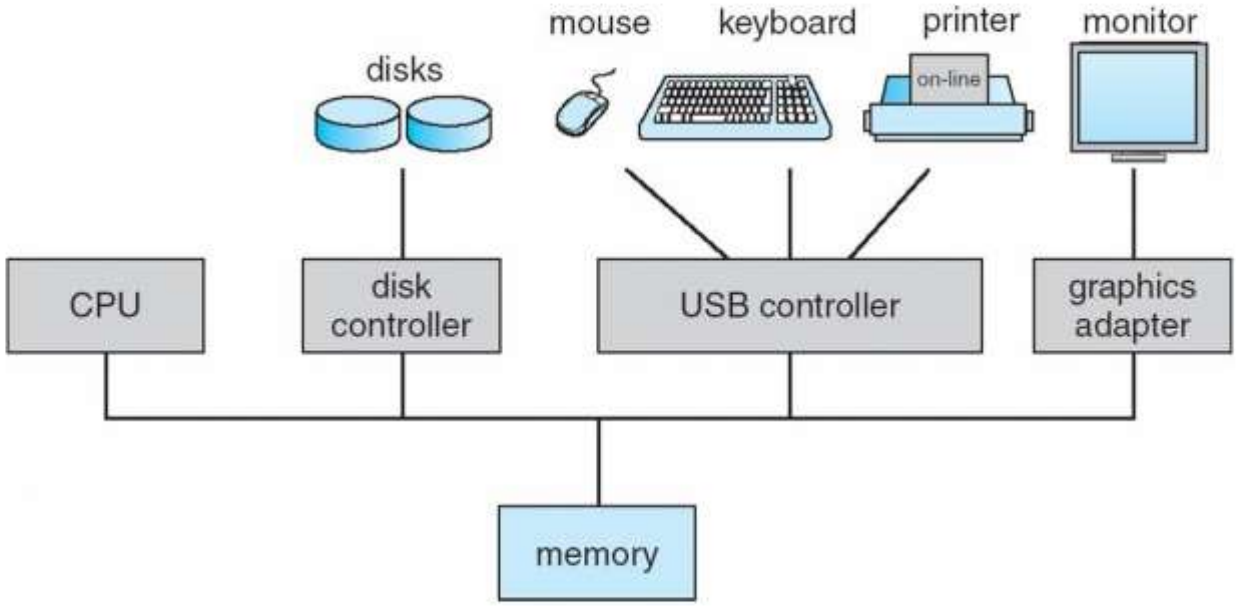


Figure 2 : **A Modern Computer System**

- I/O devices and the CPU can execute concurrently
- Each device controller is in charge of a particular device type
- Each device controller has a local buffer

- CPU moves data from/to main memory to/from local buffers
- I/O is from the device to local buffer of controller

VIEWS AND GOALS

Operating system goals:

- Execute user programs and make solving user problems easier
- Make the computer system convenient to use

Use the computer hardware in an efficient manner

User View vs System View in Operating System An operating system is a construct that allows the user application programs to interact with the system hardware. hardware. Operating system by itself does not provide any function but it provides an atmosphere in which different applications and programs can do useful work. The operating system can be observed from the point of view of the user or the system. system. This is known as the user view and the system view respectively. More details about these are given as follows –

User View The user view depends on the system interface that is used by the users. The different types of user view experiences can be explained as follows − If the user is using a personal computer, the operating system is largely designed to make the interaction easy. Some attention is also paid to the performance of the system, but there is no need for the operating system to worry about resource utilization.

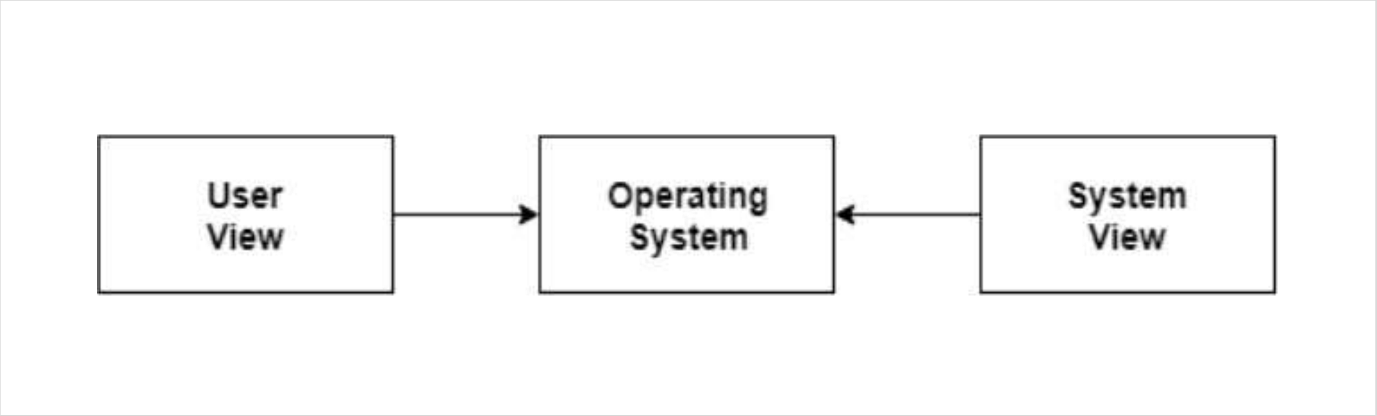


Figure 3 : **Views of Operating System**

is because the personal computer uses all the resources available and there is no sharing. If the user is using a system connected to a mainframe or a minicomputer, the operating system is largely concerned with resource utilization. This is because there may be multiple terminals connected to the mainframe and the operating system makes sure that all the resources such as CPU, memory, I/O devices etc. are divided uniformly between them. If the user is sitting on a workstation connected to other workstations through networks, then the operating system needs to focus on both individual usage of resources and sharing though the network. This happens because the workstation exclusively uses its own resources but it also needs to share files etc. with other workstations across the network. If the user is using a handheld computer such as a mobile, then the operating system handles the usability of the device including a few remote operations. The battery level of the device is also taken into account. There are some devices that contain very less or no user view because because there is no interaction interaction with the users. Examples are embedded computers in home devices, automobiles etc

System View

According to the computer system, system, the operating system is the bridge between applications and hardware. It is most intimate with the hardware and is used to control it as required. The different types of system view for operating system can be explained as follows: The system views the operating system as a resource allocator. There are many resources such as CPU time, memory space, file storage space, I/O devices etc. that are required by processes for execution. It is the duty of the operating system to allocate these resources judiciously to the processes so that the computer system can run as smoothly as possible. The operating system can also work as a control program. It manages all the processes and I/O devices so that the computer system works smoothly and there are no errors. It makes sure that the I/O devices work in a proper manner without creating problems. Operating systems can also be viewed as a way to make using hardware easier. Computers were required to easily solve user problems. However it is not easy to work directly with the computer hardware. So, operating systems were developed to easily communicate with the hardware. An operating system can also be considered as a program running at all times in the background of a computer system (known as the kernel) and handling all the application programs. This is the definition of the operating system that is generally followed.

TYPES OF SYSTEM

- Most systems use a single general-purpose processor
  - Most systems have special-purpose processors as well
- **Multiprocessors** systems growing in use and importance
  - Also known as **parallel systems**, **tightly-coupled systems**
  - Advantages include:

Increased throughput

1. **Economy of scale**
2. **Increased reliability** – graceful degradation or fault tolerance

- Two types:
  - **Asymmetric Multiprocessing** – each processor is assigned a specific task
  - **Symmetric Multiprocessing** – each processor performs all tasks

SYMMETRIC MULTIPROCESSING ARCHITECTURE

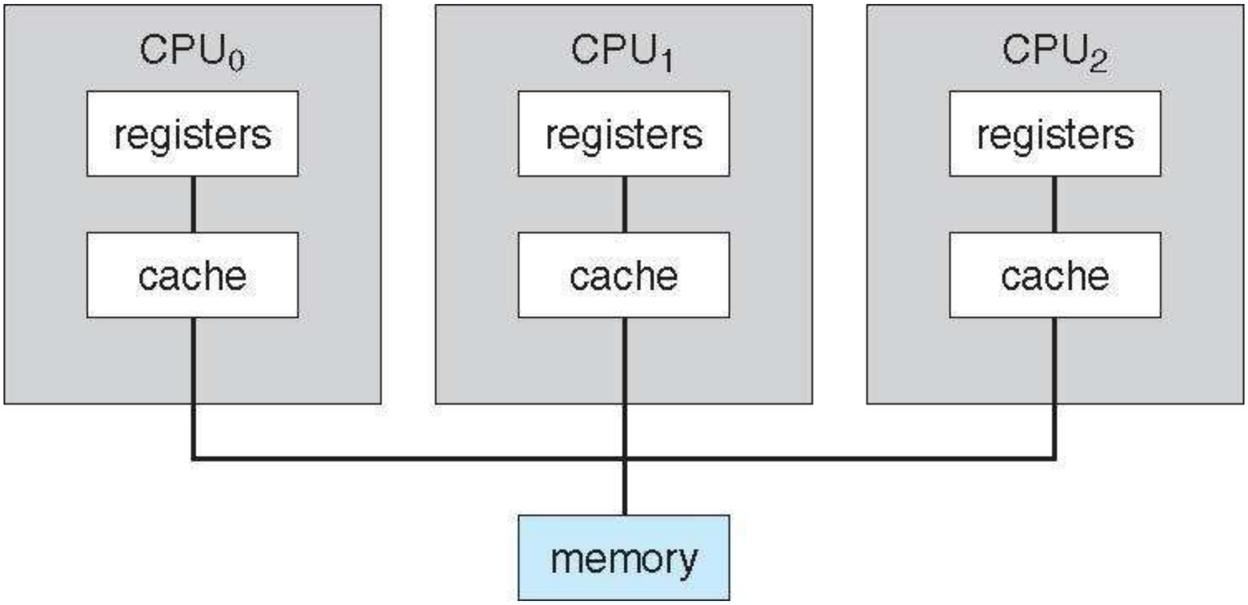


Figure 4 : **Symmetric multiprocessing architecture**

DUAL CORE DESIGN

- *Multicore*
  - Several cores on a single chip
  - On chip communication is faster than between-chip
  - Less power used

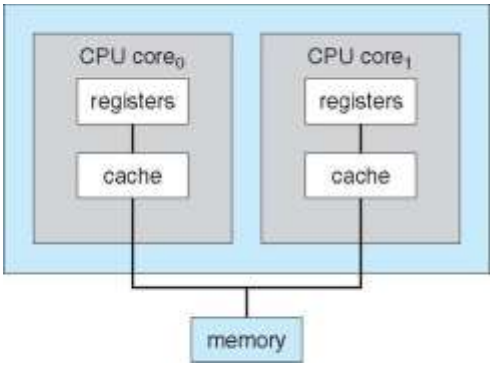


Figure 5 : Dual core design

CLUSTERED SYSTEMS

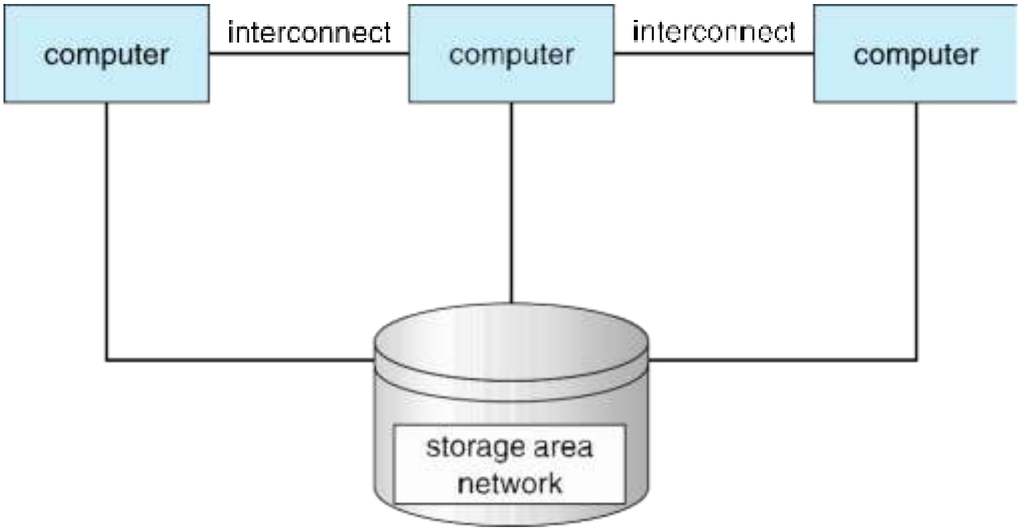


Figure 6 : Clustered systems

- Like multiprocessor systems, but multiple systems working together
- Provides a **high-availability** service which survives failures
- **Asymmetric clustering** has one machine in hot-standby mode
- **Symmetric clustering** has multiple nodes running applications, monitoring each other
- Some clusters are for **high-performance computing (HPC)**
- Applications must be written to use **parallelization**

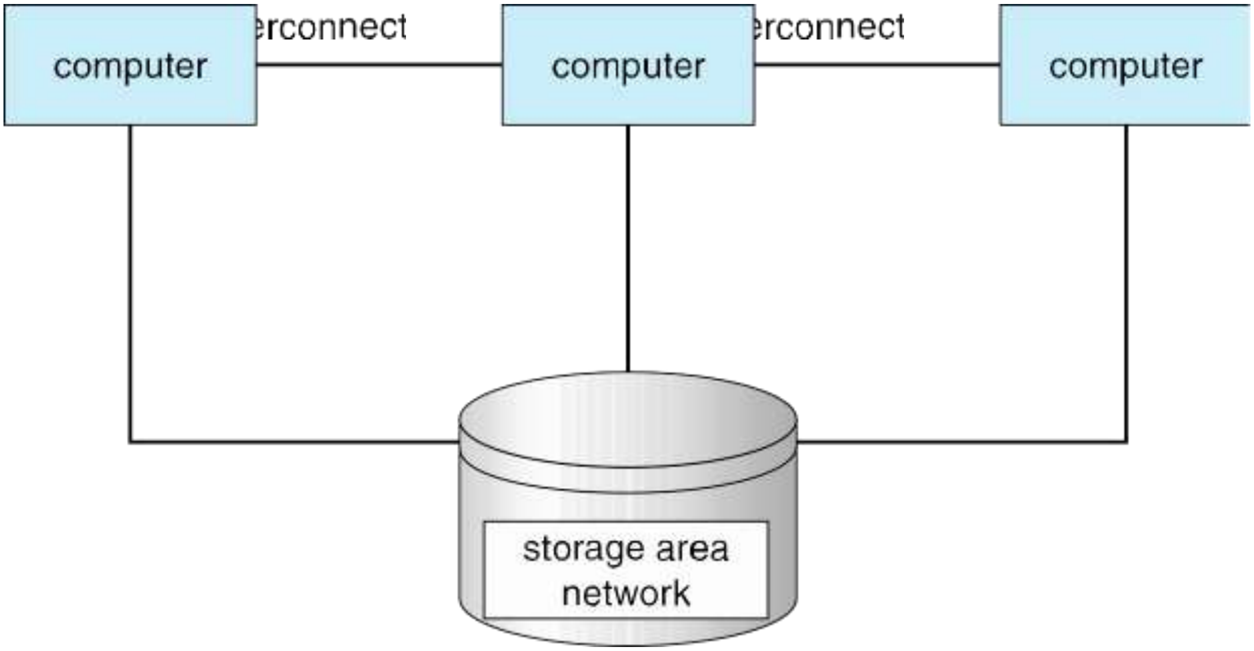


Figure7: **Clustered systems**

MEMORY LAYOUT FOR MULTIPROGRAMMED SYSTEMS

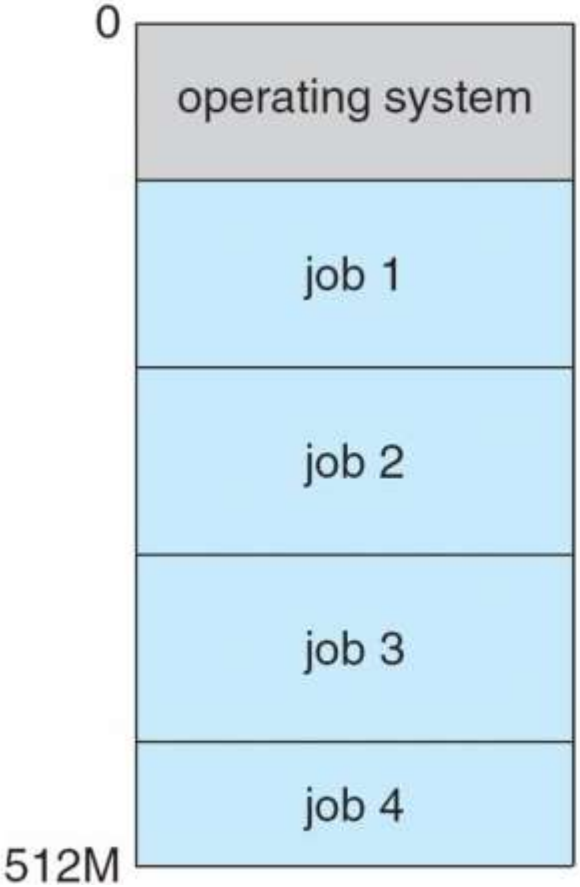


Figure 8: Memory layout for multiprogrammed systems

OPERATING SYSTEM STRUCTURES

- In general, various ways are used to structure OSes
- Many OS'es don't have well-defined structures
    - Not one pure model: Hybrid systems
    - Combine multiple approaches to address performance, security, usability needs
- Simple structure – MS-DOS
- More complex structure - UNIX
- Layered OSes
- Microkernel OSes

SIMPLE STRUCTURE

- MS-DOS was created to provide the most functionality in the least space
- Not divided into modules
- MS-DOS has some structure
    - But its interfaces and levels of functionality are not well separated
- No dual mode existed for Intel 8088
    - MS-DOS was developed for 8088
    - Direct access to hardware is allowed

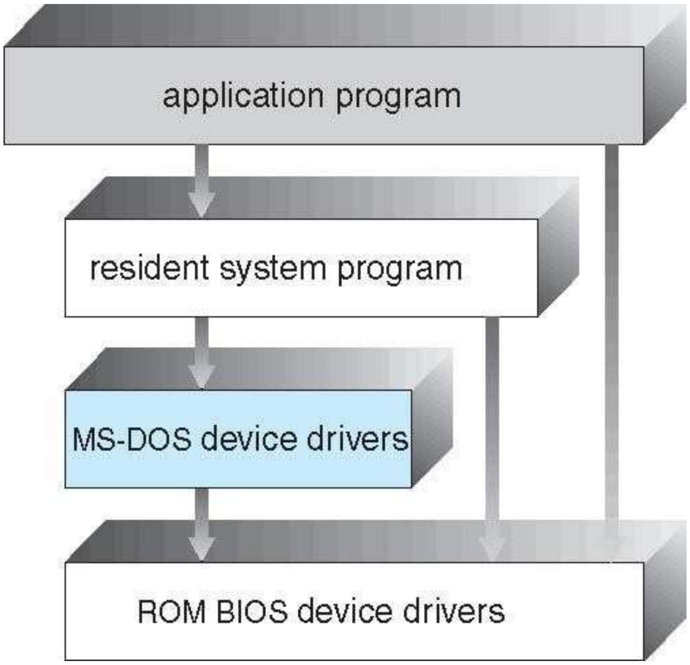


Figure 9 :MS DOS Layer Structure

- Traditional UNIX has limited structuring
- UNIX consists of 2 separable parts:
    - Systems programs
    - Kernel
- UNIX Kernel
    - Consists of everything that is
        - below the system-call interface and
        - above the physical hardware
    - Kernel provides
        - File system, CPU scheduling, memory management, and other operating-system functions
        - This is a lot of functionality for just 1 layer
        - Rather monolithic
        - But fast -- due to lack of overhead in communication inside kernel

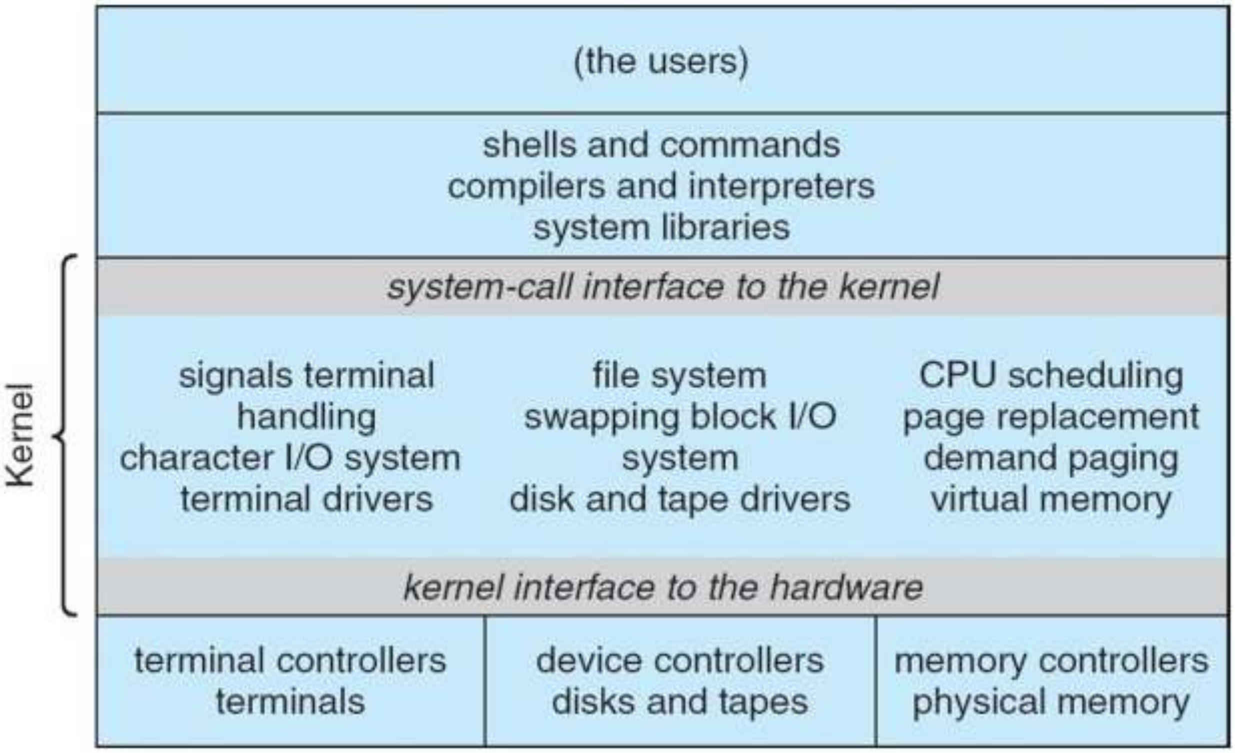


Figure 10 :Unix System Structure

□

SYSTEM COMPONENTS

We can create a system as large and complex as an operating system by partitioning it into smaller pieces. Each piece should be a well-delineated (represented accurately or precisely) portion of the system with carefully defined inputs, outputs and functions. Even though, not all systems have the same structure. However, many modern operating systems share the same goal of supporting the following types of system components:

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

Process Management

The operating system manages many kinds of activities ranging from user programs to system programs like printer spooler, name servers, file server etc. Each of these activities is encapsulated in a process. A program by itself is not a process but a program in execution. For example

− A batch job is a process

− A time-shared user program is a process

− A system task (e.g. spooling output to printer) is a process.

There can be many processes running the same program. A program does nothing unless its instructions are executed by a CPU. The execution of a process must be sequential. The five major activities of an operating system in regard to process management are:

- Creation and deletion of user and system processes.

• Suspension and resumption (Block/Unblock) of processes.

• Providing mechanism for process Synchronization.

• Providing mechanism for process Communication.

• Providing mechanism for process deadlock handling.

Main Memory Management

Main memory is central to the operation of a modern computer system. Primary-Memory or Main Memory is a large array of words or bytes ranging in size from hundreds of thousands to billion. Each word or byte has its own address. Main-memory provides storage that can be accessed directly by the CPU. The main memory is only large storage device that the CPU is able to address and access directly for a program to be executed, that must in the main memory. To improve both the utilisation of the CPU and the speed of the computer's response to its users, we must keep several programs in memory.

The major activities of an operating system in regard to memory-management are:

- Monitoring which part of memory are currently being used and by whom.
- Deciding which process are loaded into memory when memory space becomes available.
- Allocating and deallocating memory space as needed.

File Management

File management is one of the most visible components of an OS. Computers can store information on several different types of physical media (e.g. magnetic tap, magnetic disk, CD etc). Each of these media has its own properties like speed, capacity, data transfer rate and access methods. For convenient use of the computer system, the OS provides a uniform logical view of information storage.

• A file is a logical storage unit, which abstracts away the physical properties of its storage device. A file is a collection of related information defined by its creator. Commonly, files represent programs (both source and object forms) and data.

• The operating system is responsible for the following activities in connection with file management:

o Creation and deletion of files.

o Creation and deletion of directions.

o Support of primitives for manipulating files and directions.

o Mapping of files onto secondary storage.

o Backing up of files on stable (non volatile) storage media.

I/O System Management

OS hides the peculiarities of specific hardware devices from the user. I/O subsystem consists of:

o A memory management component that includes buffering, caching and spooling.

o A general device-driver interface

o 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.

• The operating system is responsible for the following activities in connection with disk management:

o Free-space management (paging/swapping)

o Storage allocation (what data goes where on the disk)

o Disk scheduling (Scheduling the requests for memory access).

Networking

A distributed systems is a collection of processors that do not share memory, peripheral devices, or a clock. Instead, each processor has its own local memory and clock, and the processors communicate with one another through various communication lines such as network or high-speed buses.

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 configured in 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 the problems of contention and security.
- A distributed system provides user access to various system resources.
- Access to a shared resource allows:
  - o Computation Speed-up
  - o Increased functionality
  - o Increased data availability
  - o 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:
  - o Distinguish between authorized and unauthorized usage.
  - o Specify the controls to be imposed.
  - o 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:

o process creation and management

o I/O handling

o secondary-storage management

o main-memory management

o file-system access

o protection

o networking

OPERATING SYSTEM SERVICES

- Operating System Services
- User Operating System Interface
- System Calls
- Types of System Calls
- System Programs
- Operating System Design and Implementation
- Operating System Structure
- Operating System Debugging
- Operating System Generation
- System Boot
- Operating systems provide an environment for execution of programs and services (helpful functions) to programs and users

- **User services:**
    - **User interface**
        - **No UI**, **Command-Line (CLI)**, **Graphics User Interface (GUI)**, **Batch**
    - **Program execution** - Loading a program into memory and running it, end 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
    - User services (Cont.):
    - **File-system manipulation** - Programs need to read and write files and directories, create and delete them, search them, list file Information, permission management.
    - **Communications** – Processes may exchange information, on the same computer or between computers over a network
        - Communications may be via shared memory or through message passing (packets moved by the OS)
    - **Error detection** – OS needs to be constantly aware of possible errors
        - May occur in the CPU and memory hardware, in I/O devices, in user program
        - For each type of error, OS should take the appropriate action to ensure correct and consistent computing
        - Debugging facilities can greatly enhance the user's and programmer's abilities to efficiently use the system

- **System services:**
    - For ensuring the efficient operation of the system itself via resource sharing
    - **Resource allocation -** When multiple users or multiple jobs running concurrently, resources must be allocated to each of them
        - Many types of resources - CPU cycles, main memory, file storage, I/O devices.
    - **Accounting -** To keep track of which users use how much and what kinds of computer resources
    - **Protection and security -** The owners of information stored in a multiuser or networked computer system may want to control use of that information,

concurrent processes should not interfere with each other

- **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

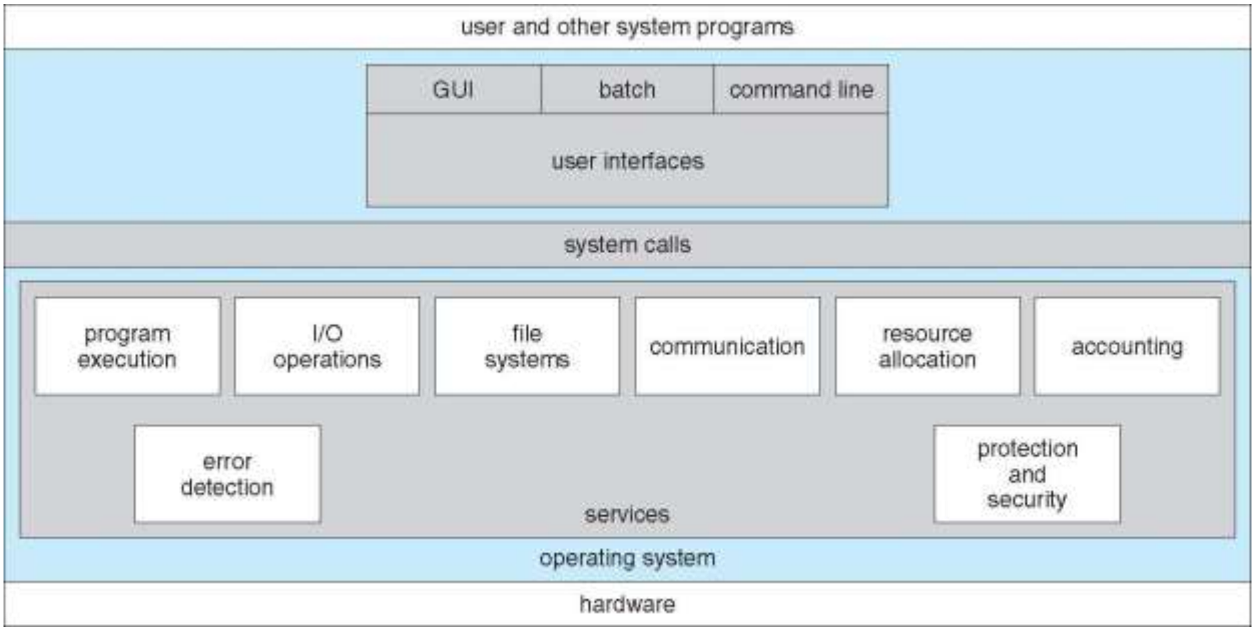
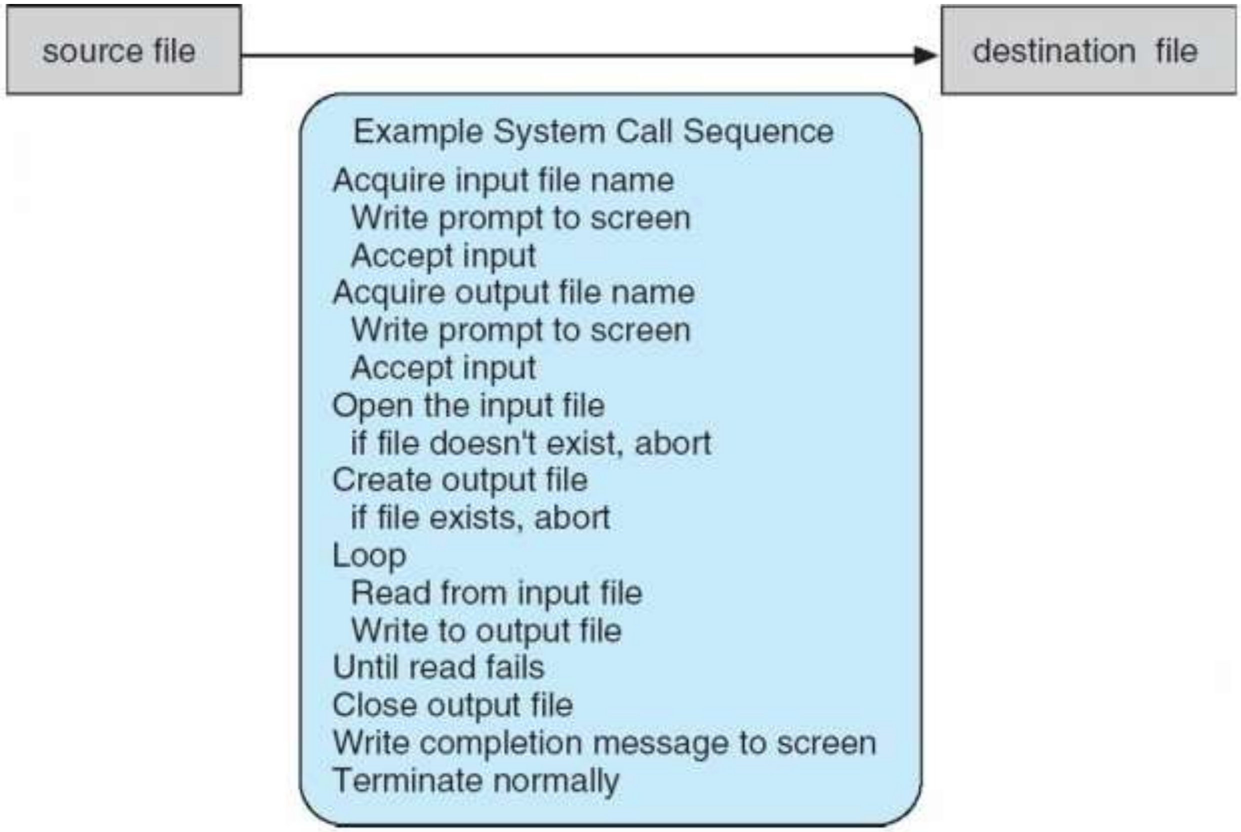


Figure 11 :OS Layer Structure

- CLI or **command interpreter** allows direct command entry
  - Sometimes implemented in kernel, sometimes by systems program
  - Sometimes multiple flavors implemented – **shells**
  - Primarily fetches a command from user and executes it
  - Sometimes commands built-in, sometimes just names of programs
    - If the latter, adding new features doesn't require shell modification
- Systems calls: programming interface to the services provided by the OS
- Typically written in a high-level language (C or C++)
- Mostly accessed by programs via a high-level **Application Programming Interface (API)** rather than direct system call use
- Three most common APIs are
  - Win32 API for Windows,
  - POSIX API for POSIX-based systems
    - including virtually all versions of UNIX, Linux, and Mac OS X,

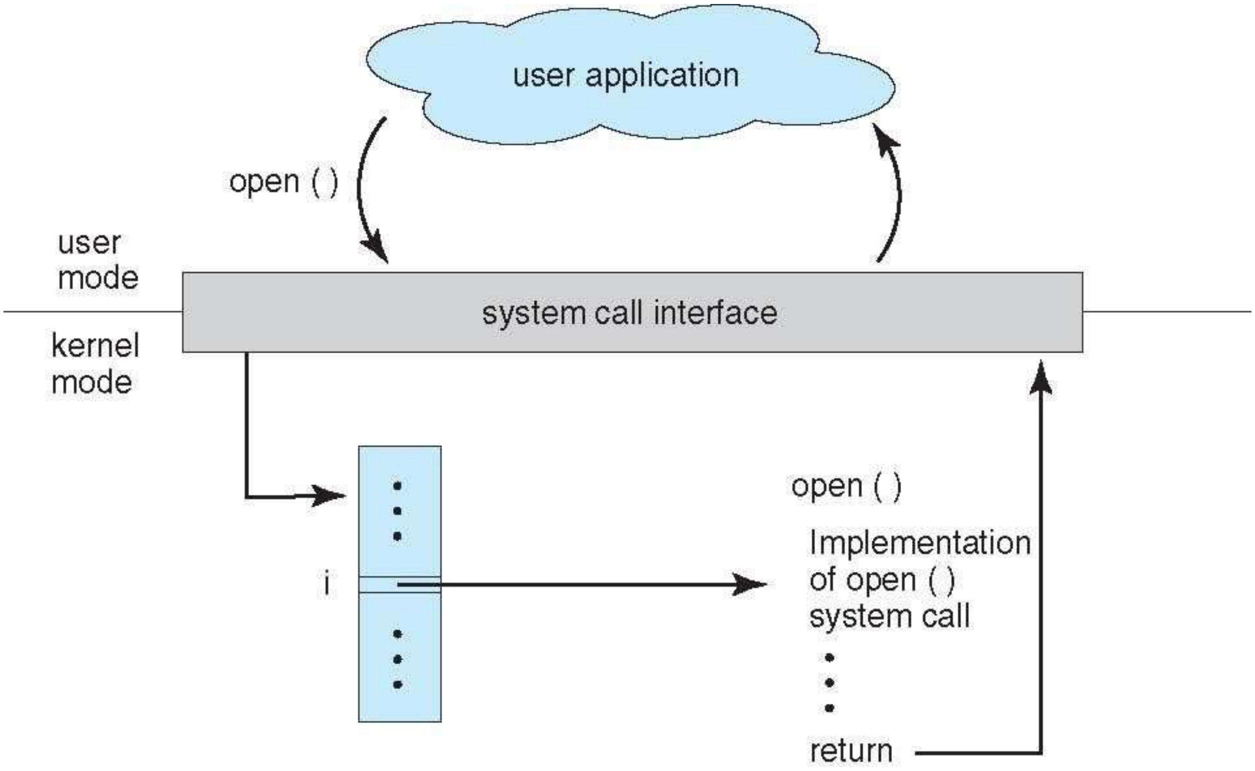
  - Java API for the Java virtual machine (JVM)



```
Example System Call Sequence
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
```

Figure 12: Example of System Call Sequence

- Typically, a number associated with each system call
  - **System-call interface** maintains a table indexed according to these numbers
- The system call interface invokes the intended system call in OS kernel and returns status of the system call and any return values
- The caller need know nothing about how the system call is implemented
  - Just needs to obey API and understand what OS will do as a result call
  - Most details of OS interface hidden from programmer by API
    - Managed by run-time support library (set of functions built into libraries included with compiler



□

Figure 13 : **Example of System Call Interface**

- *Three* general methods used to pass parameters to the OS in system calls
  - Simplest: in registers
    - In some cases, may be more parameters than registers
  - Parameters stored in a block, or table, in memory, and address of block passed as a parameter in a register
    - This approach taken by Linux and Solaris
  - Parameters placed, or **pushed**, onto the **stack** by the program and **popped** off the stack by the operating system
  - Block and stack methods do not limit the number or length of parameters being passed

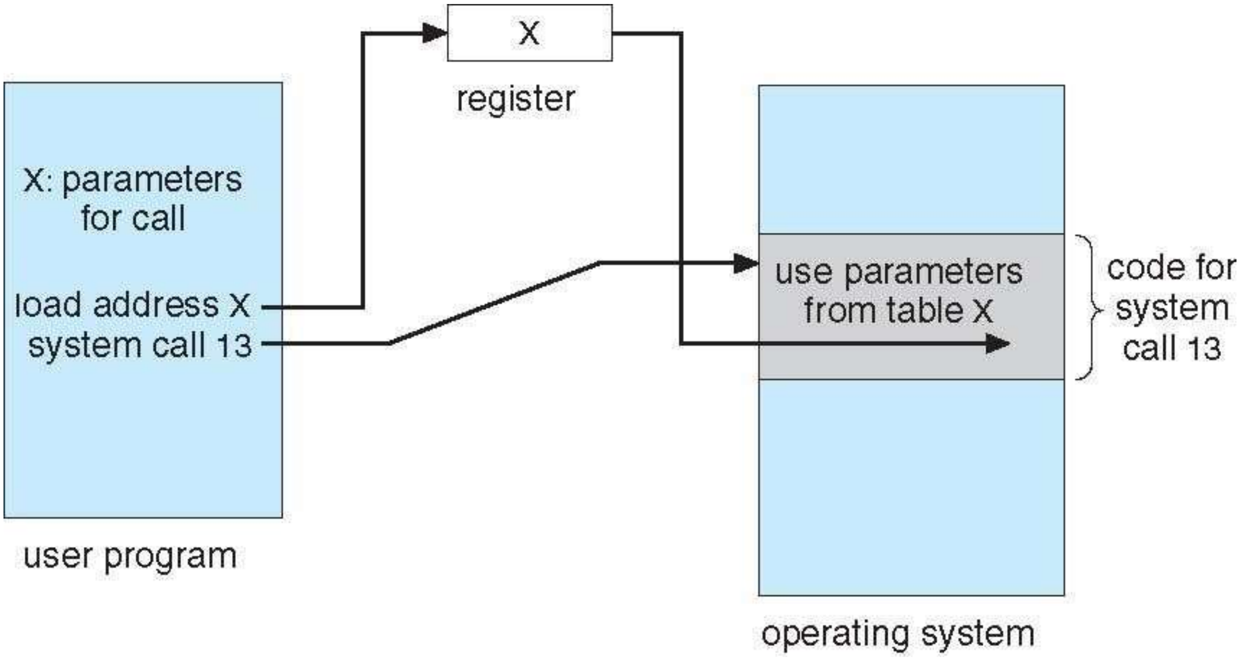


Figure 14 :**Passing of Parameters as a table**

SYSTEM STRUCTURE

I/O STRUCTURE

- Synchronous (blocking) I/O
    - Waiting for I/O to complete
    - Easy to program, not always efficient
    - Wait instruction idles the CPU until the next interrupt
    - At most one I/O request is outstanding at a time
        - no simultaneous I/O processing
- Asynchronous (nonblocking) I/O
    - After I/O starts, control returns to user program without waiting for I/O completion
    - Harder to program, more efficient
    - **System call** – request to the OS to allow user to wait for I/O completion (polling periodically to check busy/done)
    - **Device-status table** contains entry for each I/O device indicating its type, address, and state

STORAGE HIERARCHY

The basic unit of computer storage is the **bit**. A bit can contain one of two values, 0 and 1. All other storage in a computer is based on collections of bits. Given enough bits, it is amazing how many things a computer can represent: numbers, letters, images, movies, sounds, documents, and programs, to name a few. A **byte** is 8 bits, and on most computers it is the smallest convenient chunk of storage. For example, most computers don't have an instruction to move a bit but do have one to move a byte. A less common term is **word**, which is a given computer architecture's native unit of data. A word is made up of one or more bytes. For example, a computer that has 64-bit registers and 64-bit memory addressing typically has 64-bit (8-byte) words. A computer executes many operations in its native word size rather than a byte at a time.

Computer storage, along with most computer throughput, is generally measured and manipulated in bytes and collections of bytes.

A **kilobyte**, or **KB**, is 1,024 bytes

a **megabyte**, or **MB**, is $1,024^2$ bytes
a **gigabyte**, or **GB**, is $1,024^3$ bytes
a **terabyte**, or **TB**, is $1,024^4$ bytes

a **petabyte**, or **PB**, is $1,024^5$ bytes

Computer manufacturers often round off these numbers and say that a megabyte is 1 million bytes and a gigabyte is 1 billion bytes. Networking measurements are an exception to this general rule; they are given in bits (because networks move data a bit at a time).

STORAGE STRUCTURE

- Main memory – only large storage media that the CPU can access directly

Random access

  - Typically **volatile**

- Secondary storage – extension of main memory that provides large **nonvolatile** storage capacity
- Hard disks – rigid metal or glass platters covered with magnetic recording material
  - Disk surface is logically divided into **tracks**, which are subdivided into **sectors**
  - The **disk controller** determines the logical interaction between the device and the computer
- **Solid-state disks** – faster than hard disks, nonvolatile
  - Various technologies
  - Becoming more popular

- Storage systems organized in hierarchy
  - Speed
  - Cost (per byte of storage)
  - Volatility
- **Device Driver** for each device controller to manage I/O
  - Provides uniform interface between controller and kernel

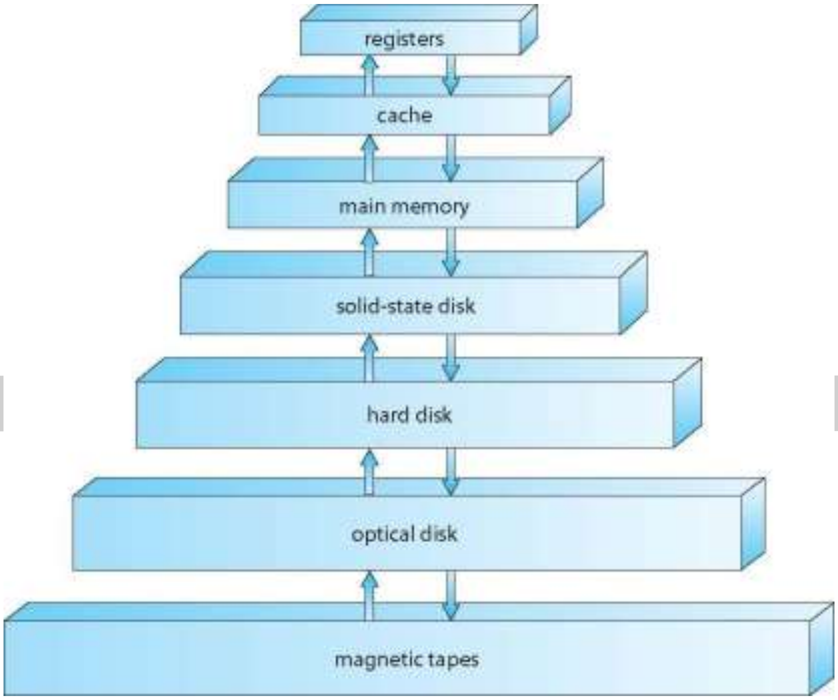
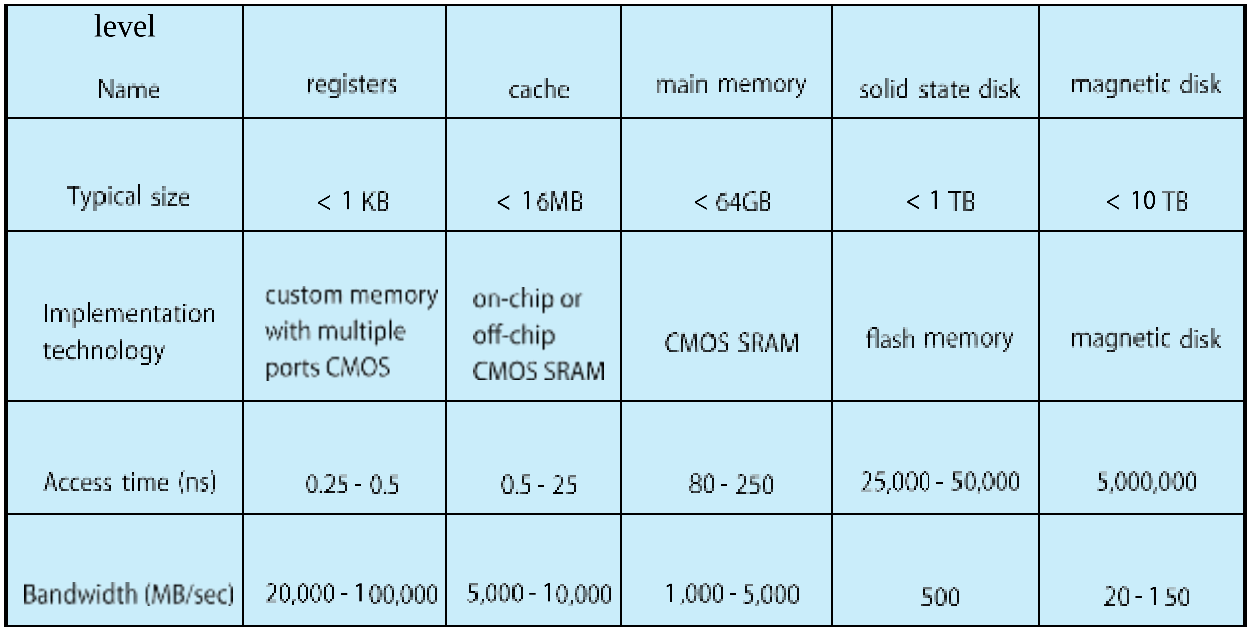
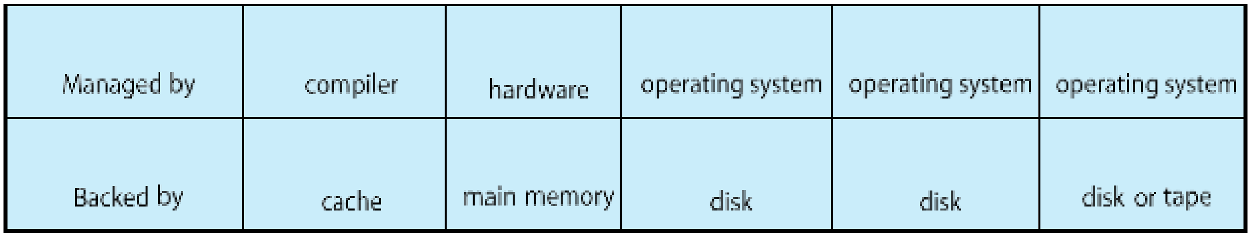


Figure 15 :Storage device hierarchy



level Name	registers	cache	main memory	solid state disk	magnetic disk
Typical size	< 1 KB	< 16MB	< 64GB	< 1 TB	< 10 TB
Implementation technology	custom memory with multiple ports CMOS	on-chip or off-chip CMOS SRAM	CMOS SRAM	flash memory	magnetic disk
Access time (ns)	0.25 - 0.5	0.5 - 25	80 - 250	25,000 - 50,000	5,000,000
Bandwidth (MB/sec)	20,000 - 100,000	5,000 - 10,000	1,000 - 5,000	500	20 - 150



Managed by	compiler	hardware	operating system	operating system	operating system
Backed by	cache	main memory	disk	disk	disk or tape

Figure 16 :Comparative analysis of Components of Operating System

CACHING

- Important principle
- Performed at many levels in a computer
  - in hardware,
  - operating system,
  - software
- Information in use copied from slower to faster storage temporarily
  - Efficiency
- Faster storage (cache) checked first to determine if information is there
  - If it is, information used directly from the cache (fast)
  - If not, data copied to cache and used there
- Cache smaller than storage being cached
  - Cache management important design problem
  - Cache size and replacement policy

DIRECT MEMORY ACCESS STRUCTURE

- Typically used for I/O devices that generate data in blocks, or generate data fast
- Device controller transfers blocks of data from buffer storage directly to main memory without CPU intervention
- Only one interrupt is generated per block, rather than the one interrupt per byte

HOW A MODERN COMPUTER SYSTEM WORKS

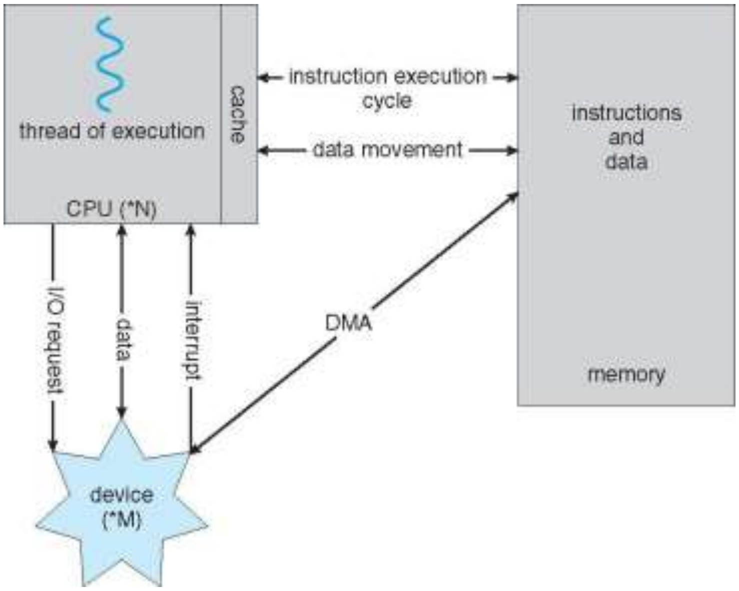


Figure 17: Working of modern computer system

LAYERED APPROACH

Layered Operating System

Layered Structure is a type of system structure in which the different services of the operating system are split into various layers, where each layer has a specific well-defined task to perform. It was created to improve the pre-existing structures like the Monolithic structure

(UNIX) and the Simple structure (MS-DOS).

Example – The Windows NT operating system uses this layered approach as a part of it.

Design Analysis :

The whole Operating System is separated into several layers (from 0 to n) as the diagram shows. Each of the layers must have its own specific function to perform. There are some rules in the implementation of the layers as follows.

1. The outermost layer must be the User Interface layer.
2. The innermost layer must be the Hardware layer.
3. A particular layer can access all the layers present below it but it cannot access the layers present above it. That is layer n-1 can access all the layers from n-2 to 0 but it cannot access the nth layer.

Thus if the user layer wants to interact with the hardware layer, the response will be travelled through all the layers from n-1 to 1. Each layer must be designed and implemented such that it will need only the services provided by the layers below it.

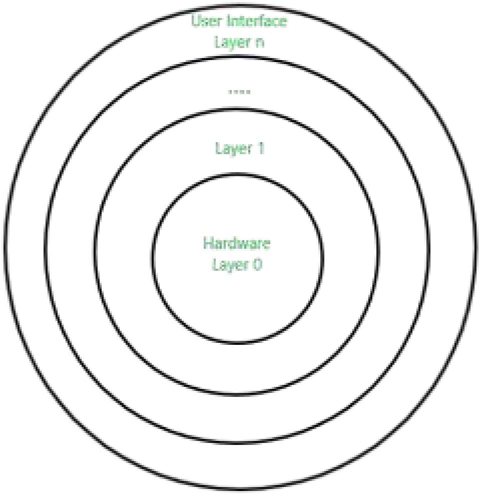


Figure:18: Layered Operating System

Layered OS Design

Advantages :

There are several advantages to this design :

1. Modularity :

This design promotes modularity as each layer performs only the tasks it is scheduled to perform.

2. Easy debugging :

As the layers are discrete so it is very easy to debug. Suppose an error occurs in the CPU scheduling layer, so the developer can only search that particular layer to debug, unlike the Monolithic system in which all the services are present together.

3. Easy update :

A modification made in a particular layer will not affect the other layers.

4. No direct access to hardware :

The hardware layer is the innermost layer present in the design. So a user can use the services of hardware but cannot directly modify or access it, unlike the Simple system in which the user had direct access to the hardware.

5. Abstraction :

Every layer is concerned with its own functions. So the functions and implementations of the other layers are abstract to it.

Disadvantages :

Though this system has several advantages over the Monolithic and Simple design, there are also some disadvantages as follows.

1. Complex and careful implementation :

As a layer can access the services of the layers below it, so the arrangement of the layers must be done carefully. For example, the backing storage layer uses the services of the memory management layer. So it must be kept below the memory management layer. Thus with great modularity comes complex implementation.

2. Slower in execution :

If a layer wants to interact with another layer, it sends a request that has to travel through all the layers present in between the two interacting layers. Thus it increases response time, unlike the Monolithic system which is faster than this. Thus an increase in the number of layers may lead to a very inefficient design.

PROCESS MANAGEMENT

  - CPU, memory, I/O, files
  - Initialization data
- Typically system has many processes, some user, some operating system running concurrently on one or more CPUs
  - Concurrency by multiplexing the CPUs among the processes / threads

ACTIVITIES

- Creating and deleting both user and system processes
- Suspending and resuming processes
- Providing mechanisms for process synchronization
- Providing mechanisms for process communication
- Providing mechanisms for deadlock handling

MEMORY MANAGEMENT

- To execute a program all (or part) of the instructions must be in memory
- All (or part) of the data that is needed by the program must be in memory.
- Memory management determines what is in memory and when
  - Optimizing CPU utilization and computer response to users
- Memory management activities
  - Keeping track of which parts of memory are currently being used and by whom
  - Deciding which processes (or parts thereof) and data to move into and out of memory
  - Allocating and deallocating memory space as needed

STORAGE MANAGEMENT

- OS provides uniform, logical view of information storage
    - Different devices, same view
    - Abstracts physical properties to logical storage unit - **file**
    - Each medium is controlled by device (i.e., disk drive, tape drive)
        - Varying properties include access speed, capacity, data-transfer rate, access method (sequential or random)
        - File-System management

- Files usually organized into directories
- Access control on most systems to determine who can access what
- OS activities include
  - Creating and deleting files and directories
  - Primitives to manipulate files and directories
  - Mapping files onto secondary storage
  - Backup files onto stable (non-volatile)

storage media MASS STORAGE MANAGEMENT

- Usually disks used to store
- data that does not fit in main memory, or
- data that must be kept for a ―long‖ period of time
- Proper management is of central importance
- Entire speed of computer operation hinges on disk subsystem and its algorithms
- Disk is slow, its I/O is often a bottleneck
- OS activities
- Free-space management
- Storage allocation
- Disk scheduling
- Multitasking environments must be careful to use most recent value, no matter where it is stored in the storage hierarchy
- Multiprocessor environment must provide **cache coherency** in hardware such that all CPUs have the most recent value in their cache
- Distributed environment situation even more complex

Several copies of a datum can exist

PROCESS

PROCESSES - PROCESS

CONCEPT

- An operating system executes a variety of programs:
  - Batch system – ‑**jobs”**
  - Time-shared systems – ‑**user programs”** or ‑**tasks”**
- We will use the terms ***job*** and ***process*** almost interchangeably
- **Process** – is a program in execution (informal definition)
- Program is ***passive*** entity stored on disk (**executable file**), process is ***active***

  - Program becomes process when executable file loaded into memory
- Execution of program started via GUI, command line entry of its name, etc
- One program can be several processes
  - Consider multiple users executing the same program
- In memory, a process consists of **multiple parts:**
  - **Program code**, also called **text section**
  - **Current activity** including
    - **program counter**
      - processor registers
  - **Stack** containing temporary data
      - Function parameters, return addresses, local variables
  - **Data section** containing global variables
  - **Heap** containing memory dynamically allocated during run time

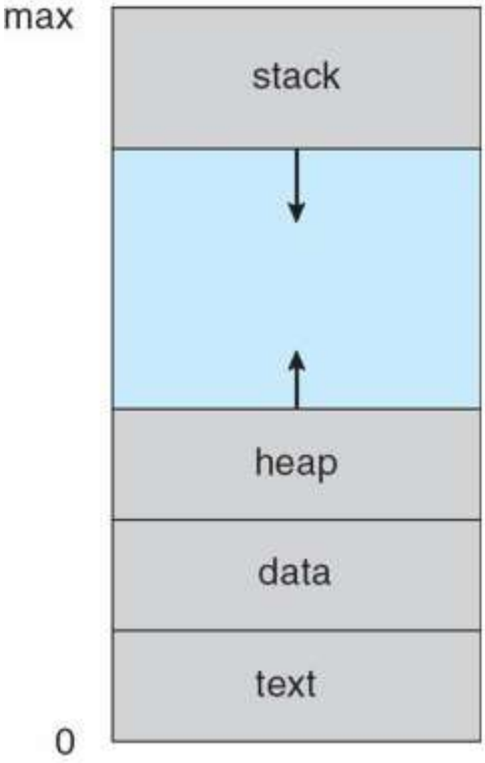


Figure 19: Allocation in Memory

DIAGRAM OF PROCESS STATE

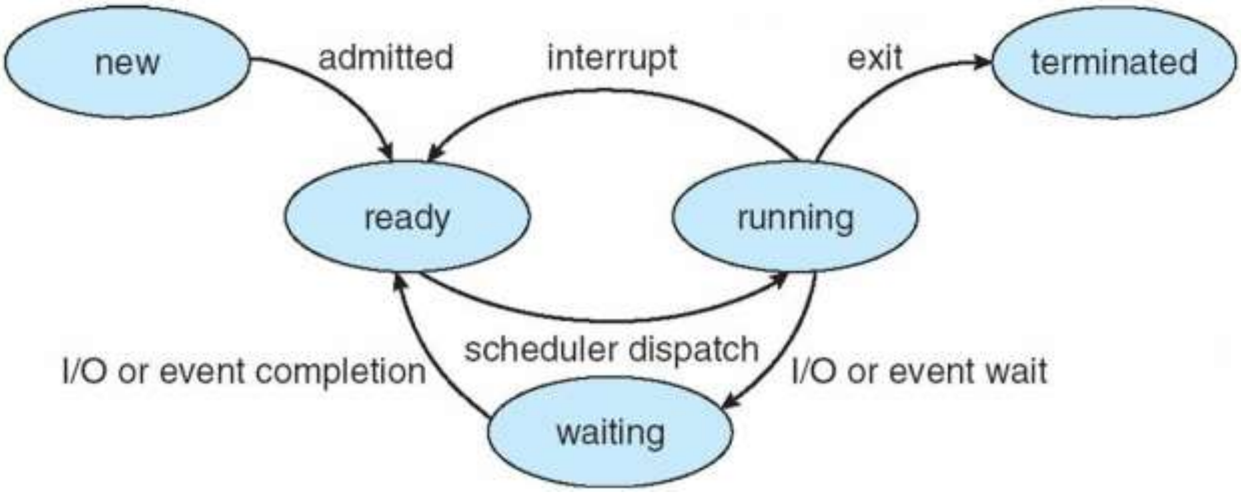


Figure 20: **Process State Representation**

- As a process executes, it changes **state**
  - **new**: The process is being created
  - **ready**: The process is waiting to be assigned to a processor
  - **running**: Instructions are being executed
  - **waiting**: The process is waiting for some event to occur
  - **terminated**: The process has finished

execution PROCESS CONTROL BLOCK(PCB)

Each process is represented in OS by PCB

- PCB - info associated with the process
- Also called **task control block**
- Process state – running, waiting, etc
- Program counter – location of instruction to next execute
- CPU registers – contents of all process-centric registers
- CPU scheduling information- priorities, scheduling queue pointers
- Memory-management information – memory allocated to the process
- Accounting information – CPU used, clock time elapsed since start, time limits
- I/O status information – I/O devices allocated to process, list of open files

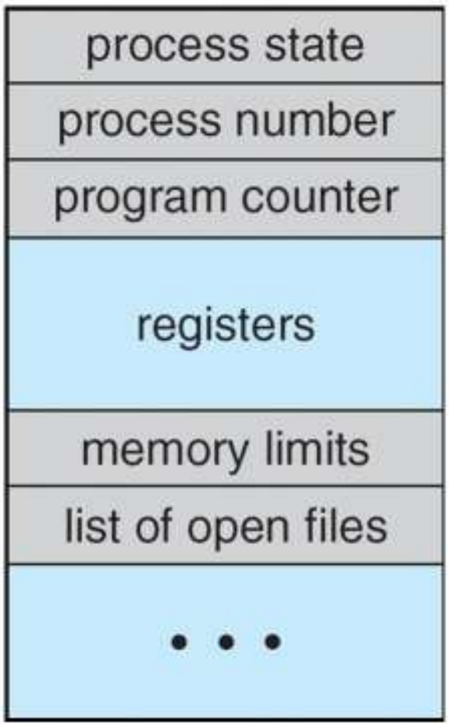


Figure 21: Process Control Block

CPU SWITCH FROM PROCESS TO PROCESS

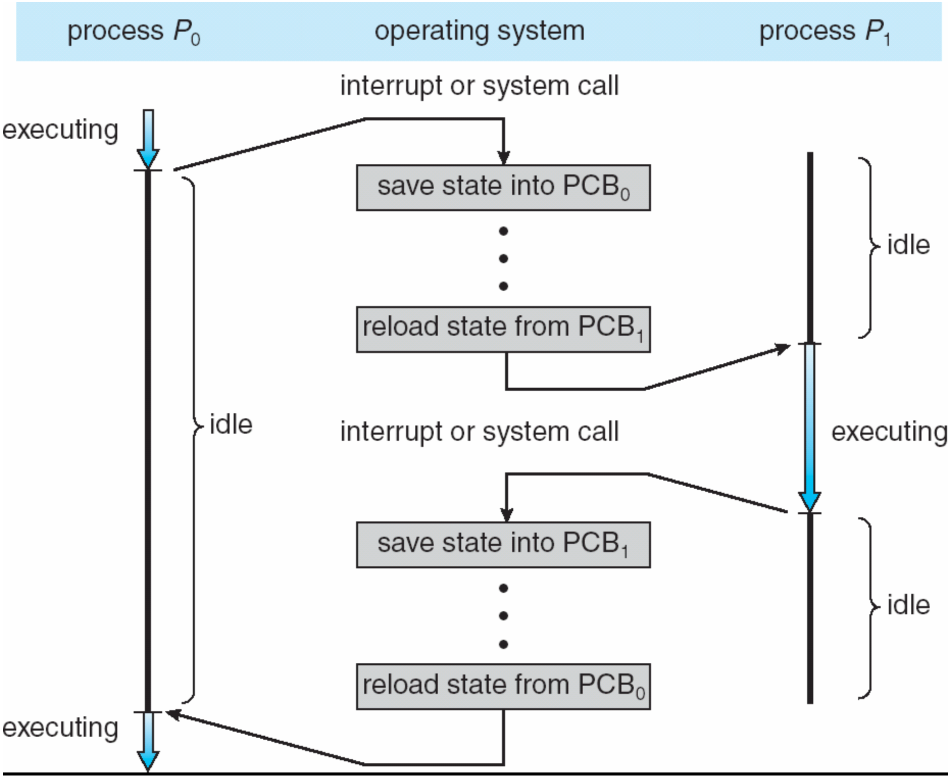


Figure 22: Diagram showing CPU switch from process to process

THREADS

- So far, process has a single thread of execution
  - One task at a time
- Consider having multiple program counters per process
  - Multiple locations can execute at once
  - Multiple tasks at a time
  - Multiple threads of control -> **threads**
- PCB must be extended to handle threads:
  - Store thread details
  - Multiple program

counters PROCESS

REPRESENTATION IN LINUX

Represented by the C structure

task_struct

pid t_pid; /* process

identifier */ long state; /*

state of the process */

unsigned int time_slice /* scheduling

information */ struct task_struct *parent; /*

this process's parent */ struct list_head

children; /* this process's children */

struct files_struct *files; /* list of open files */

struct mm_struct *mm; /* address space of this process */

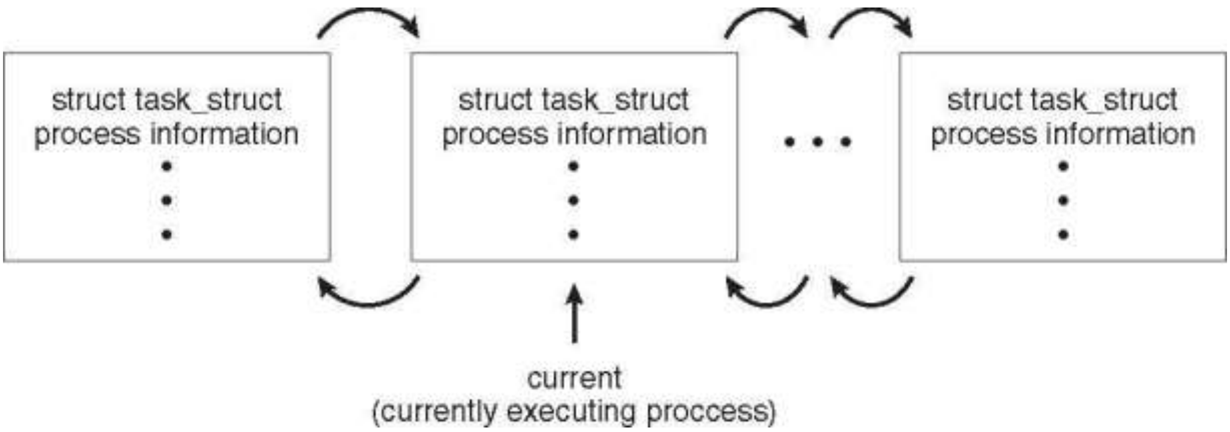


Figure 23: Representation in linux

PROCESS SCHEDULING

- Goal of multiprogramming:
  - Maximize CPU use
- Goal of time sharing:
  - Quickly switch processes onto CPU for time sharing
- **Process scheduler** – needed to meet these goals
  - Selects 1 process to be executed next on CPU
  - Among available processes
- Maintains **scheduling queues** of processes
  - **Job queue** – set of all processes in the system
  - **Ready queue** – set of all processes residing in main memory, ready and waiting to execute
  - **Device queues** – set of processes waiting for an I/O device
  - Processes migrate among the various queues

READY QUEUE AND VARIOUS QUEUES

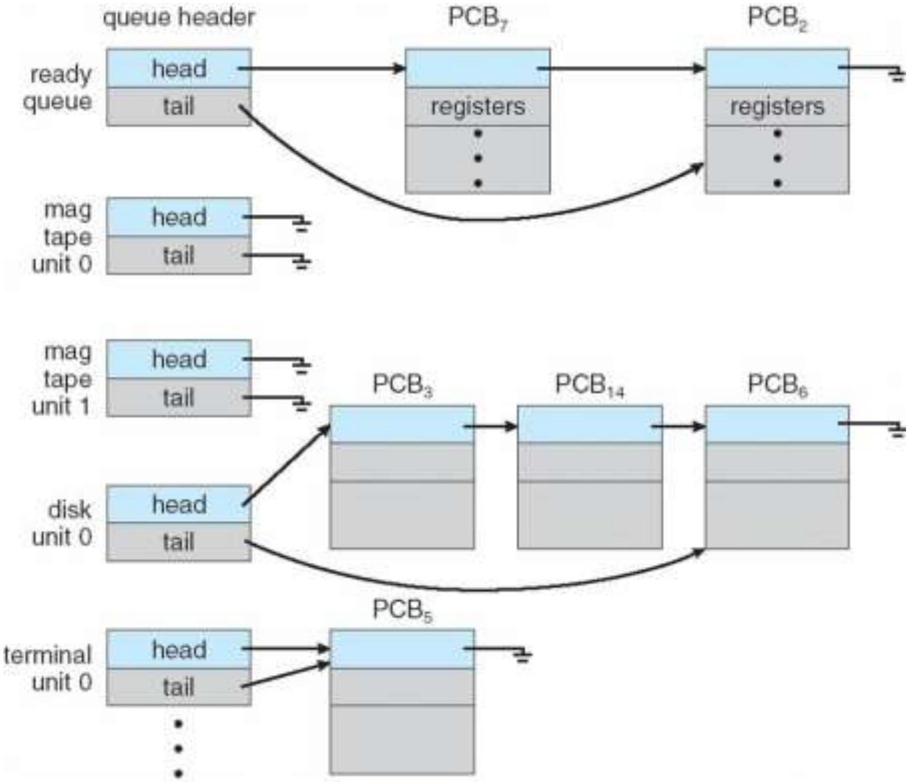


Figure 24: The ready queue and various input output device queue

REPRESENTATION OF PROCESS SCHEDULING

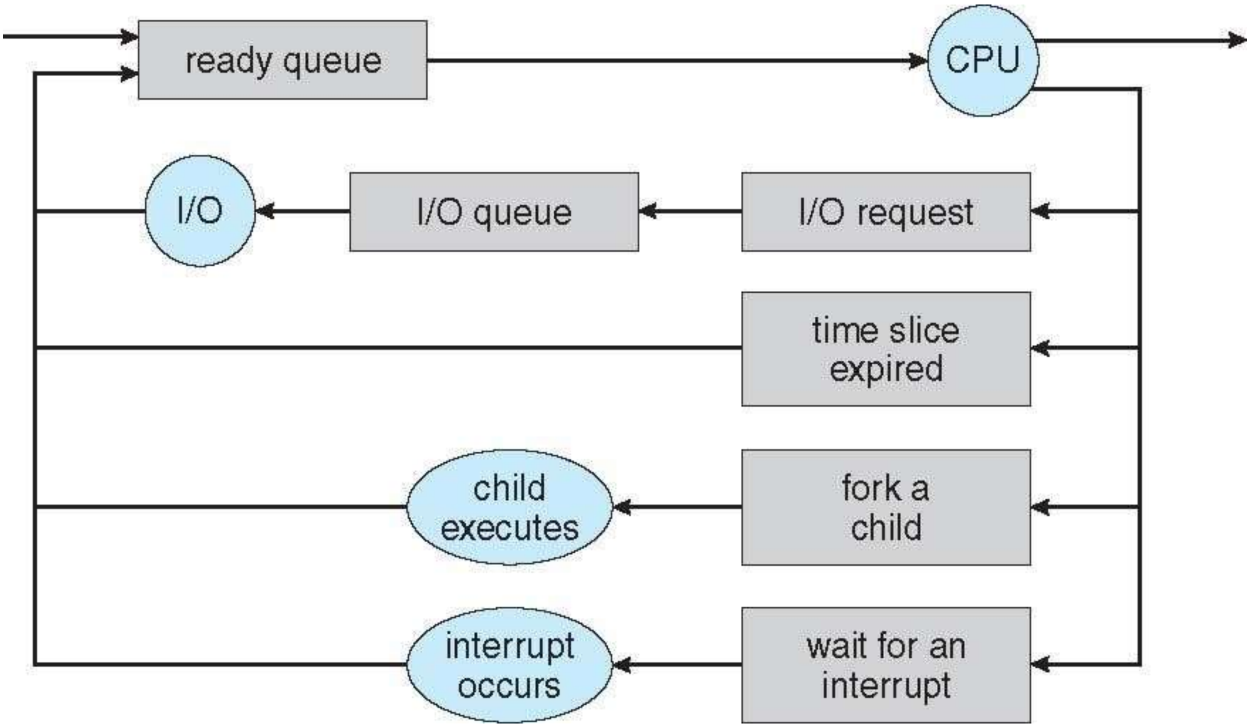


Figure 25: Representation of process scheduling

Queuing diagram

- a common representation of process scheduling
- represents queues, resources, flows

SCHEDULERS

- **Scheduler –** component that decides how processes are selected from these queues for scheduling purposes

Long-term scheduler (or job scheduler)

- On this slide - ‒LTS‖ (LTS is not a common notation)
- In a **batch system**, more processes are submitted then can be executed in memory
- They are spooled to disk
- LTS selects which processes should be brought into the ready queue
- LTS is invoked infrequently
- (seconds, minutes) (may be slow, hence can use advanced algorithms)
- LTS controls the **degree of multiprogramming**
- The number of processes in memory

- Processes can be described as either:

I/O-bound process

- Spends more time doing I/O than computations, many short CPU bursts

CPU-bound process

- Spends more time doing computations; few very long CPU bursts
- LTS strives for good ***process mix***

Short-term scheduler (or CPU scheduler)

- Selects 1 process to be executed next
    - Among ready-to-execute processes
        - From the ready queue
    - Allocates CPU to this process
- Sometimes the only scheduler in a system
- Short-term scheduler is invoked frequently
    - (milliseconds) □ (must be fast)
    - Hence cannot use costly selection logic

- **Medium-term scheduler** can be added if degree of multiple programming needs to decrease
    - Used by time-sharing OSes, etc
        - Too many programs □poor performance □users quit
- Key idea:
    - Reduce the degree of multiprogramming by **swapping**
    - **Swapping** removes a process from memory, stores on disk, brings back in from disk to continue execution

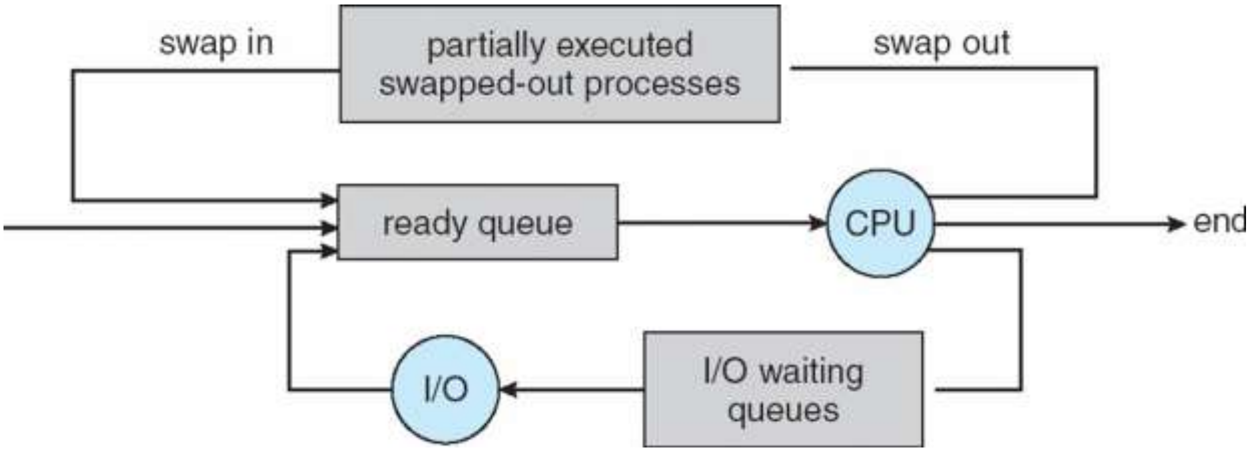


Figure 26:Addition of Medium term scheduling to queueing diagram

CONTEXT SWITCH

- **Context** of a process represented in its PCB

Context switch

  - When CPU switches to another process, the system must:
    - **save the state** of the old process, and
    - load the **saved state** for the new process
- Context-switch time is overhead
  - The system does no useful work while switching
  - The more complex the OS and the PCB
    - the longer the context switch
- This overhead time is dependent on hardware support
  - Some hardware provides multiple sets of registers per CPU
    - multiple contexts are loaded at once
    - switch requires only changing pointer to

the right set OPERATIONS ON PROCESSES

- System must provide mechanisms for:
    - process creation,
    - process termination,
    -

PROCESS CREATION

- A **(parent)** process can create several (**children)** processes
- Children can, in turn, create other processes
- Hence, a **tree** of processes forms
- Generally, process identified and managed via a **process identifier** (**pid**)
- Resource sharing options (of process creation)
- Parent and children share all resources
- Children share subset of parent’s resources
- One usage is to prevent system overload by too many child processes
- Parent and child share no resources
- Execution options
- Parent and children execute concurrently
- Parent waits until children terminate

- Address space options
  - Child duplicate of parent
  - Child has a program loaded into it
- UNIX examples
  - **fork()** system call creates new process
    - Child is a copy of parent's address space
      - except fork() returns 0 to child and nonzero to parent
  - **exec()** system call used after a **fork()** to replace the process' memory space with a new program

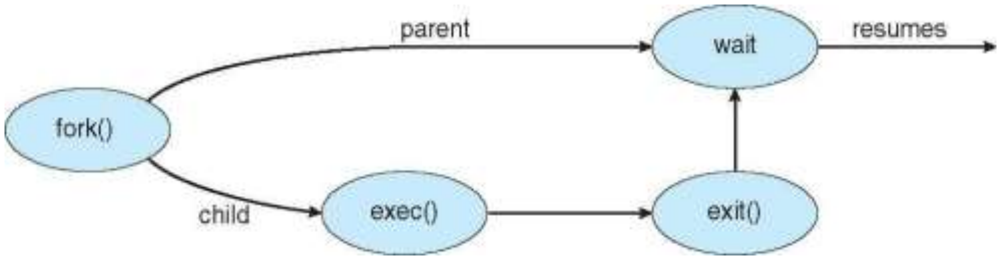


Figure 27: Process Execution – Unix example

PROCESS TERMINATION

- Process executes last statement and then asks the operating system to delete it using the **exit()** system call.
  - Returns status data from child to parent (via **wait()**)
  - Process' resources are deallocated by operating system
- Parent may terminate the execution of children processes using the **abort()** system call. Some reasons for doing so:
  - Child has exceeded allocated resources
  - Task assigned to child is no longer required
  - The parent is exiting and the operating systems does not allow a child to continue if its parent terminates
  - Some OSes don't allow child to exists if its parent has terminated
  - **cascading termination -** if a process terminates, then all its children, grandchildren, etc must also be terminated.
  - The termination is initiated by the operating system
- The parent process may wait for termination of a child process by using the

wait()system call**.**

  - The call returns status information and the pid of the terminated process

- *pid = wait(&status);*
- If no parent waiting (did not invoke **wait()**) process is a **zombie**
  - All its resources are deallocated, but exit status is kept
- If parent terminated without invoking **wait** , process is an **orphan**
  - UNIX: assigns **init** process as the parent
  - Init calls wait periodically

CO OPERATING PROCESSES

- Processes within a system may be ***independent*** or ***cooperating***
  - When processes execute they produce some **computational results**
  - ***Independent*** process cannot affect (or be affected) by such results of another process
  - ***Cooperating*** process can affect (or be affected) by such results of another process
- Advantages of process cooperation
  - Information sharing
  - Computation speed-up
  - Modularity
  - Convenience

INTERPROCESS COMMUNICATION

- For fast exchange of information, cooperating processes need some **interprocess communication** (**IPC**) mechanisms
- Two models of IPC

Shared memory

  - **Message passing**

- An area of memory is shared among the processes that wish to communicate
- The communication is under the control of the users processes, not the OS.
- Major issue is to provide mechanism that will allow the user processes to synchronize their actions when they access shared memory.
- **Producer-consumer problem** – a common paradigm for cooperating processes
  - Used to exemplify one common generic way/scenario of cooperation among processes
  - We will use it to exemplify IPC

  - Very important!
- ***Producer*** process
  - produces some information
  - incrementally
- ***Consumer*** process
  - consumes this information
  - as it becomes available
- Challenge:
  - Producer and consumer should run concurrently and efficiently
  - Producer and consumer must be synchronized
    - Consumer cannot consume an item before it is produced

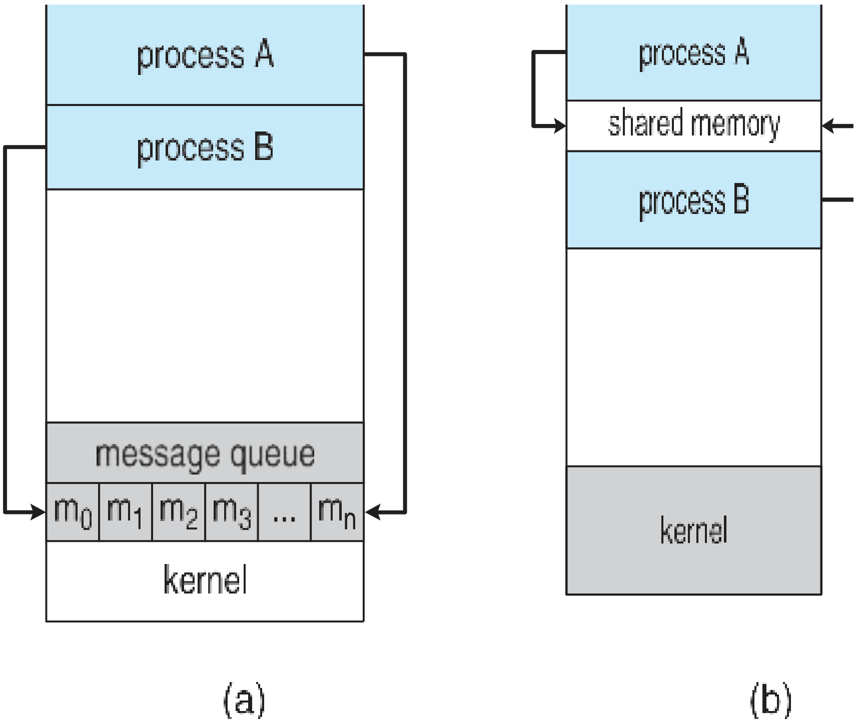


Figure 28: **Interprocess Communication**

- Shared-memory solution to producer-consumer
  - Uses a buffer in shared memory to exchange information
    - **unbounded-buffer:** assumes no practical limit on the buffer size
    - **bounded-buffer** assumes a fixed buffer size

- Shared data #define

```
    BUFFER_SIZE 10 typedef struct
    {
              . . .

    } item;
    item
    buffer[BUFFER_SIZ
    E]; int in = 0;
int out = 0;

    item
    next_produced
    ; while (true) {
              next_produced = ProduceItem();
              while (((in + 1) % BUFFER_SIZE) == out)

                        ; /* do nothing, no space in buffer */
                      //wait for consumer to get items and            //free up some space
          /* enough space in buffer */
              buffer[in] = next_produced; //put item
              into buffer in = (in + 1) %
              BUFFER_SIZE;
    }
```

MESSAGE PASSING

- Mechanism for processes to communicate and to synchronize their actions
- Message system – processes communicate with each other without resorting to shared variables
- IPC facility provides two operations:
    - **send**(*message*)
    - **receive**(*message*)
- The *message* size is either fixed or variable

- If processes *P* and *Q* wish to communicate, they need to:
- Establish a ***communication link*** between them
- Exchange messages via send/receive
- Implementation issues:
- How are links established?
- Can a link be associated with more than two processes?
- How many links can there be between every pair of communicating processes?
- What is the capacity (buffer size) of a link?
- Is the size of a message that the link can accommodate fixed or variable?
- Is a link unidirectional or bi-directional?

Logical implementation of communication link

- Direct or indirect
- Synchronous or asynchronous
- Automatic or explicit

buffering DIRECT

COMMUNICATION

- Processes must name each other explicitly:
  - **send** (*P, message*) – send a message to process P
  - **receive**(*Q*, *message*) – receive a message from process Q
  - Properties of a direct communication link
- Links are established automatically
  - A link is associated with exactly one pair of communicating processes
  - Between each pair there exists exactly one link
  - The link may be unidirectional, but is usually bi-

directional INDIRECT COMMUNICATION

- Messages are directed and received from mailboxes (also referred to as ports)
- Each mailbox has a unique id
- Processes can communicate only if they share a mailbox
- Properties of an indirect communication link
- Link established only if processes share a common mailbox
- A link may be associated with many processes
- Each pair of processes may share several communication links
- Link may be unidirectional or bi-directional

- Operations
    - create a new mailbox (port)
    - send and receive messages through mailbox
    - destroy a mailbox
    - Primitives are defined as:

  send(*A*, *message*) – send a message to mailbox A

receive(*A*, *message*) – receive a message from mailbox A

Question Format :

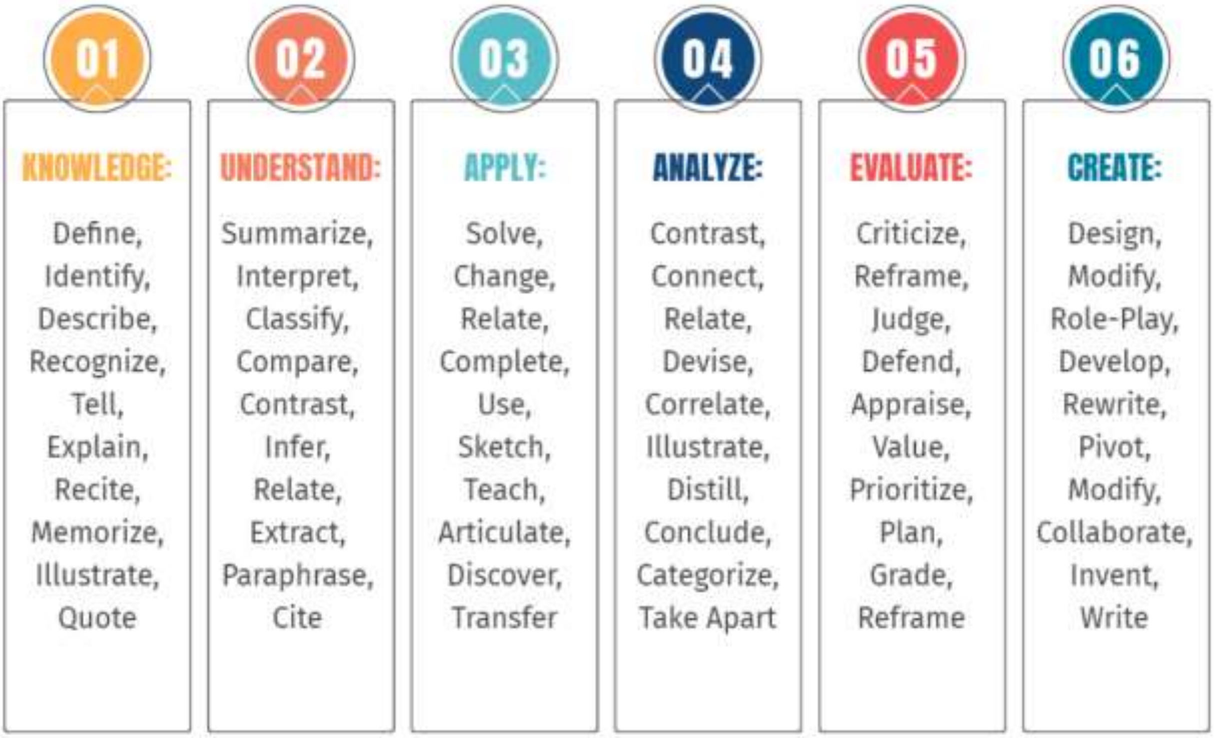
Part-A

Q.No	**Questions**	**Competence**	**BT Level**
1.	What is an operating system?	Knowledge	BT L1
2.	What are operating system services?	Knowledge	BT L1
3.	Describe distributed operating system?	Knowledge	BT L1
4.	Differentiate between process and threads	Understand	BT L2
5.	Define schedulers?	Knowledge	BT L1
6.	What are the types of scheduler?	Knowledge	BT L1
7.	Define process?	Knowledge	BT L1
8.	What does PCB contain?	Knowledge	BT L1
9.	Describe the operating system operations?	Knowledge	BT L1
10.	Explain time sharing operating system?	Knowledge	BT L1

Part-B

Q.No	**Questions**	**Competence**	**BT Level**
1.	Explain OS structure	Knowledge	BT L1
2.	Describe the layered approach	Knowledge	BT L1
3.	Explain User Operating-System Interface in detail	Knowledge	BT L1
4.	Explain operating system functions and services in detail.	Knowledge	BT L1
5.	Explain the Inter Process Communication	Knowledge	BT L1
6.	Describe the Operating System Components	Knowledge	BT L1
7.	Describe the process scheduling	Knowledge	BT L1

Bloom's Taxonomy



01 KNOWLEDGE:
Define, Identify, Describe, Recognize, Tell, Explain, Recite, Memorize, Illustrate, Quote

02 UNDERSTAND:
Summarize, Interpret, Classify, Compare, Contrast, Infer, Relate, Extract, Paraphrase, Cite

03 APPLY:
Solve, Change, Relate, Complete, Use, Sketch, Teach, Articulate, Discover, Transfer

04 ANALYZE:
Contrast, Connect, Relate, Devise, Correlate, Illustrate, Distill, Conclude, Categorize, Take Apart

05 EVALUATE:
Criticize, Reframe, Judge, Defend, Appraise, Value, Prioritize, Plan, Grade, Reframe

06 CREATE:
Design, Modify, Role-Play, Develop, Rewrite, Pivot, Modify, Collaborate, Invent, Write

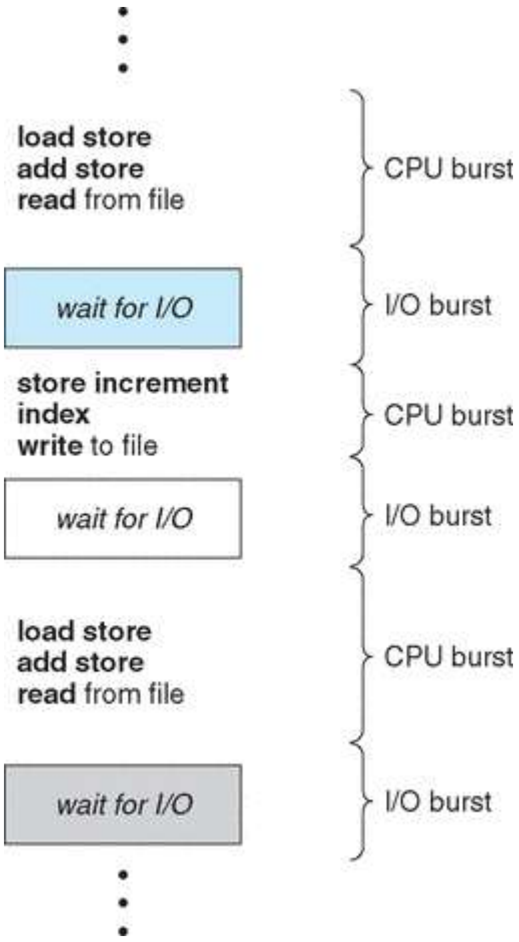
UNIT – II Operating System – SBSA1301

Unit- 2

1. CPU Scheduling -basic concepts

In a single-processor system, only one process can run at a time; any others 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. With multiprogramming, we try to use this time productively. Several processes are kept in memory at one time. When one process has to wait, the operating system takes the CPU away from that process and gives the CPU to another process. This pattern continues. Every time one process has to wait, another process can take over use of the CPU.

Alternating sequence of CPU and I/O bursts.



The success of CPU scheduling depends on an observed property of processes: process execution consists of a cycle of CPU execution and I/O wait. Processes alternate between these two states.

Process execution begins with a CPU burst. That is followed by an I/O burst, which is followed by another CPU burst, then another I/O burst, and so on. Eventually, the final CPU burst ends with a system request to terminate execution.

Whenever the CPU becomes idle, the operating system must select one of the processes in the ready queue to be executed. The selection process is carried out by the short-term scheduler (or CPU scheduler). The scheduler selects a process from the processes in memory that are ready to execute and allocates the CPU to that process.

1.1 Preemptive Scheduling

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

1. When a process switches from the running state to the waiting state (for example, as the result of an I/O request or an invocation of wait for the termination of one of the child processes)

2. When a process switches from the running state to the ready state (for example, when an interrupt occurs)

3. When a process switches from the waiting state to the ready state (for example, at completion of I/O)

4. When a process terminates

For situations 1 and 4, there is no choice in terms of scheduling. A new process (if one exists in the ready queue) must be selected for execution. There is a choice, however, for situations 2 and 3.

When scheduling takes place only under circumstances 1 and 4, we say that the scheduling scheme is nonpreemptive or cooperative; otherwise, it is preemptive. Under nonpreemptive scheduling, once the CPU has been allocated to a process, the process keeps the CPU until it releases the CPU either by terminating or by switching to the waiting state.

1.2 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 the following:

- Switching context
- Switching to user mode
- Jumping to the proper location in the user program to restart that program

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.

2. Scheduling Criteria

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**. If the CPU is busy executing processes, then work is being done. One measure of work is the number of processes that are completed per time unit, called throughput. For long processes, this rate may be one process per hour; for short transactions, it may be ten processes per second.
- **Turnaround time**. From the point of view of a particular process, the important criterion is how long it takes to execute that process. The interval from the time of submission of a process to the time of completion is the turnaround time. Turnaround time is the sum of the periods spent waiting to get into memory, waiting in the ready queue, executing on the CPU, and doing I/O.
- **Waiting time**. The CPU-scheduling algorithm does not affect the amount of time during which a process executes or does I/O; it affects only the amount of time that a process spends waiting in the ready queue. Waiting time is the sum of the periods spent waiting in the ready queue.
- **Response time**. In an interactive system, turnaround time may not be the best criterion. Often, a process can produce some output fairly early and can continue computing new results while previous results are being output to the user. Thus, another measure is the time from the submission of a request until the first response is produced. This measure, called response time, is the time it takes to start responding, not the time it takes to output the response. The turnaround time is generally limited by the speed of the output device.

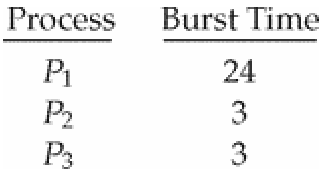
3. Scheduling Algorithms

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.

3.1 First-Come, First-Served Scheduling

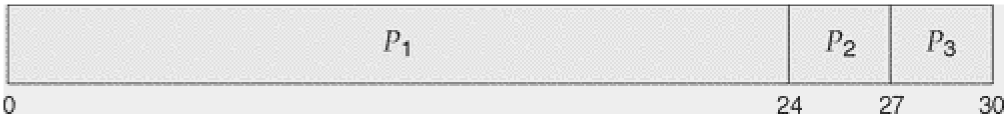
The simplest CPU-scheduling algorithm is the first-come, first-served (FCFS) scheduling algorithm. With this scheme, the process that requests the CPU first is allocated the CPU first. The implementation of the FCFS policy is easily managed with a FIFO queue. When a process enters the ready queue, its PCB is linked onto the tail of the queue. When the CPU is free, it is allocated to the process at the head of the queue. The running process is then removed from the queue. The code for FCFS scheduling is simple to write and understand.

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

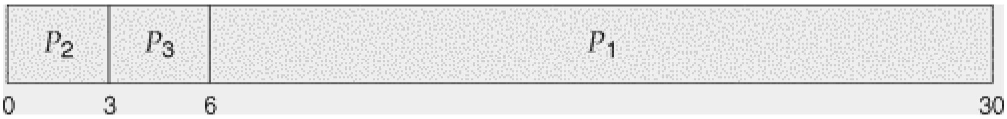


Process	Burst Time
P_1	24
P_2	3
P_3	3

If the processes arrive in the order P1, P2, P3, and are served in FCFS order, we get the result shown in the following Gantt chart, which is a bar chart that illustrates a particular schedule, including the start and finish times of each of the participating processes:



The waiting time is 0 milliseconds for process P1, 24 milliseconds for process P2, and 27 milliseconds for process P3. Thus, the average waiting time is (0 + 24 + 27)/3 = 17 milliseconds. If the processes arrive in the order P2, P3, P1, however, the results will be as shown in the following Gantt chart:

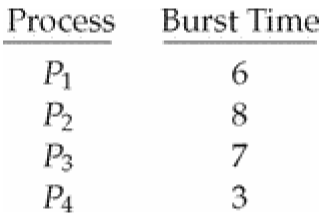


The average waiting time is now (6 + 0 + 3)/3 = 3 milliseconds. This reduction is substantial. Thus, the average waiting time under an FCFS policy is generally not minimal and may vary substantially if the processes CPU burst times vary greatly.

3.2 Shortest-Job-First Scheduling

A different approach to CPU scheduling is the shortest-job-first (SJF) scheduling algorithm. This algorithm associates with each process the length of the process's next CPU burst. When the CPU is available, it is assigned to the process that has the smallest next CPU burst. If the next CPU bursts of two processes are the same, FCFS scheduling is used to break the tie. Note that a more appropriate term for this scheduling method would be the shortest-next-CPU-burst algorithm, because scheduling depends on the length of the next CPU burst of a process, rather than its total length. We use the term SJF because most people and textbooks use this term to refer to this type of scheduling.

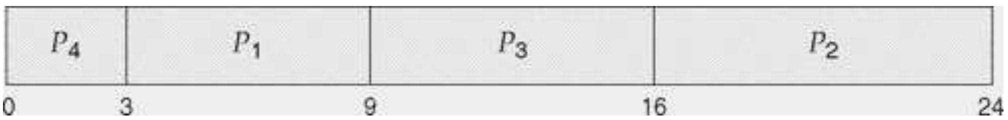
As an example of SJF scheduling, consider the following set of processes, with the length of the CPU burst given in milliseconds:



Process	Burst Time
P_1	6
P_2	8
P_3	7
P_4	3

4.

Using SJF scheduling, we would schedule these processes according to the following Gantt chart:



The waiting time is 3 milliseconds for process P1, 16 milliseconds for process P2, 9 milliseconds for process P3, and 0 milliseconds for process P4. Thus, the average waiting time is (3 + 16 + 9 + 0)/4 = 7 milliseconds. By comparison, if we were using the FCFS scheduling scheme, the average waiting time would be 10.25 milliseconds.

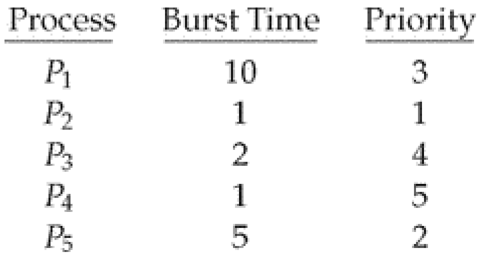
The SJF scheduling algorithm is provably optimal, in that it gives the minimum average waiting time for a given set of processes. Moving a short process before long one decrease the waiting time of the short process more than it increases the waiting time of the long process. Consequently, the average waiting time decreases.

3.3 Priority Scheduling

The SJF algorithm is a special case of the general priority scheduling algorithm. A priority is associated with each process, and the CPU is allocated to the process with the highest priority. Equal-priority processes are scheduled in FCFS order. An SJF algorithm is simply a priority algorithm where the priority (p) is the inverse of the (predicted) next CPU burst. The larger the CPU burst, the lower the priority, and vice versa.

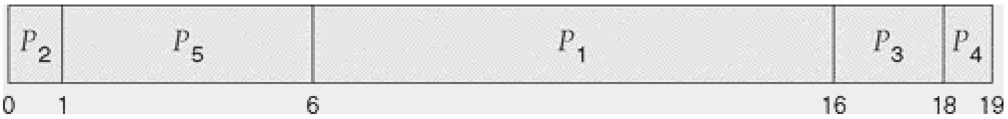
Priorities are generally indicated by some fixed range of numbers, such as 0 to 7 or 0 to 4,095. However, there is no general agreement on whether 0 is the highest or lowest priority. Some systems use low numbers to represent low priority; others use low numbers for high priority. This difference can lead to confusion. In this text, we assume that low numbers represent high priority.

As an example, consider the following set of processes, assumed to have arrived at time 0 in the order P1, P2, ···, P5, with the length of the CPU burst given in milliseconds:



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

Using priority scheduling, we would schedule these processes according to the following Gantt chart:



The average waiting time is 8.2 milliseconds.

Priorities can be defined either internally or externally. Internally defined priorities use some measurable quantity or quantities to compute the priority of a process. For example, time limits, memory requirements, the number of open files, and the ratio of average I/O burst to average CPU burst have been used in computing priorities. External priorities are set by criteria outside the operating system, such as the importance of the process, the type and amount of funds being paid for computer use, the department sponsoring the work, and other, often political, factors.

Priority scheduling can be either preemptive or nonpreemptive. When a process arrives at the ready queue, its priority is compared with the priority of the currently running process. A preemptive priority scheduling algorithm will preempt the CPU if the priority of the newly arrived process is higher than the priority of the currently running process. A nonpreemptive priority scheduling algorithm will simply put the new process at the head of the ready queue.

A major problem with priority scheduling algorithms is indefinite blocking , or starvation. A process that is ready to run but waiting for the CPU can be considered blocked. A priority scheduling algorithm can leave some low-priority processes waiting indefinitely. In a heavily loaded computer system, a steady stream of higher-priority processes can prevent a low-priority process from ever getting the CPU.

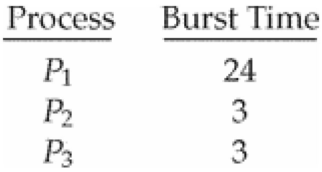
3.4 Round-Robin Scheduling

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

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

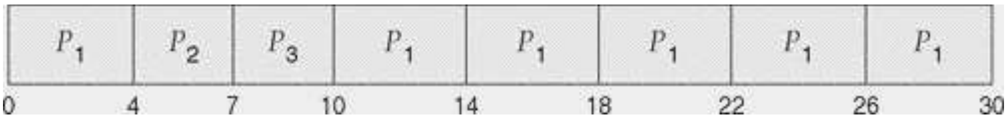
One of two things will then happen. The process may have a CPU burst of less than 1 time quantum. In this case, the process itself will release the CPU voluntarily. The scheduler will then proceed to the next process in the ready queue. Otherwise, if the CPU burst of the currently running process is longer than 1 time quantum, the timer will go off and will cause an interrupt to the operating system. A context switch will be executed, and the process will be put at the tail of the ready queue. The CPU scheduler will then select the next process in the ready queue.

The average waiting time under the RR policy is often long. Consider the following set of processes that arrive at time 0, with the length of the CPU burst given in milliseconds:



Process	Burst Time
P_1	24
P_2	3
P_3	3

If we use a time quantum of 4 milliseconds, then process P1 gets the first 4 milliseconds. Since it requires another 20 milliseconds, it is preempted after the first time quantum, and the CPU is given to the next process in the queue, process P2. Process P2 does not need 4 milliseconds, so it quits before its time quantum expires. The CPU is then given to the next process, process P3. Once each process has received 1 time quantum, the CPU is returned to process P1 for an additional time quantum. The resulting RR schedule is as follows:



Let’s calculate the average waiting time for the above schedule. P1 waits for 6 millisconds (10 - 4), P2 waits for 4 millisconds, and P3 waits for 7 millisconds. Thus, the average waiting time is 17/3 = 5.66 milliseconds.

In the RR scheduling algorithm, no process is allocated the CPU for more than 1 time quantum in a row (unless it is the only runnable process). If a process’s CPU burst exceeds 1 time quantum, that process is preempted and is put back in the ready queue. The RR scheduling algorithm is thus preemptive.

3.5 Multilevel Queue Scheduling

Another class of scheduling algorithms has been created for situations in which processes are easily classified into different groups. For example, a common division is made between foreground (interactive) processes and background (batch) processes. These two types of processes have different response-time requirements and so may have different scheduling needs. In addition, foreground processes may have priority (externally defined) over background processes.

A multilevel queue scheduling algorithm partitions the ready queue into several separate queues. The processes are permanently assigned to one queue, generally based on some property of the process, such as memory size, process priority, or process type. Each queue has its own scheduling algorithm. For example, separate queues might be used for foreground and background processes. The foreground queue might be scheduled by an RR algorithm, while the background queue is scheduled by an FCFS algorithm.

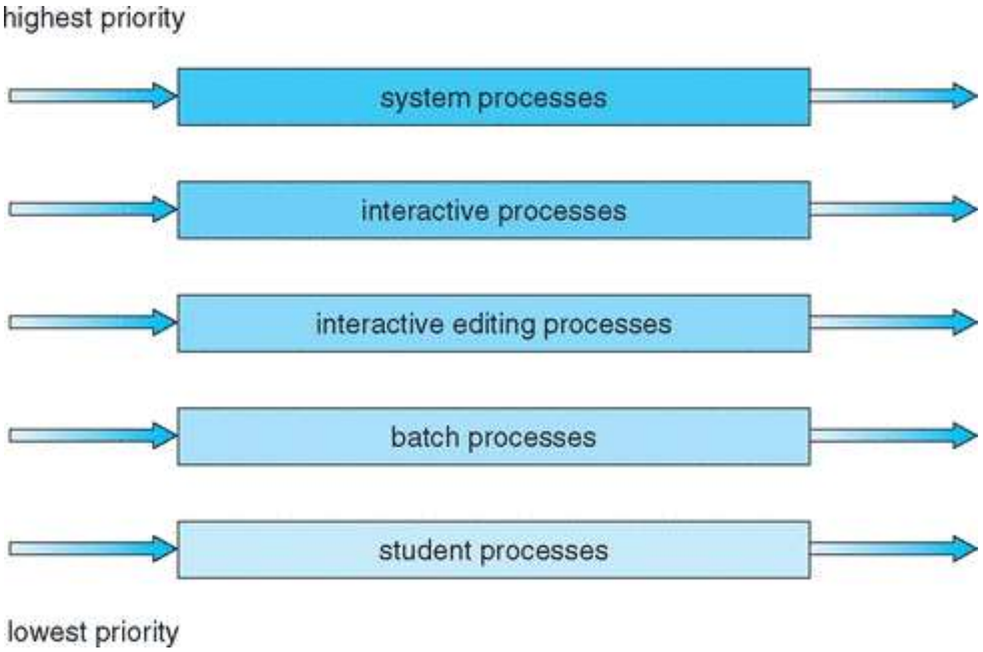


Figure 29 : Multilevel queue scheduling.

In addition, there must be scheduling among the queues, which is commonly implemented as fixed-priority preemptive scheduling. For example, the foreground queue may have absolute priority over the background queue.

Let’s look at an example of a multilevel queue scheduling algorithm with five queues, listed below in order of priority:

1. System processes
2. Interactive processes
3. Interactive editing processes
4. Batch processes
5. Student processes

Each queue has absolute priority over lower-priority queues. No process in the batch queue, for example, could run unless the queues for system processes, interactive processes, and interactive editing processes were all empty. If an interactive editing process entered the ready queue while a batch process was running, the batch process would be preempted.

Another possibility is to time-slice among the queues. Here, each queue gets a certain portion of the CPU time, which it can then schedule among its various processes. For instance, in the foreground-background queue example, the foreground queue can be given 80 percent of the CPU time for RR scheduling among its processes, whereas the background queue receives 20 percent of the CPU to give to its processes on an FCFS basis.

3.6 Multilevel Feedback Queue Scheduling

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. In addition, 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.

For example, consider a multilevel feedback queue scheduler with three queues, numbered from 0 to 2 (Figure 5.7). The scheduler first executes all processes in queue 0. Only when queue 0 is empty will it execute processes in queue 1. Similarly, processes in queue 2 will only be executed if queues 0 and 1 are empty. A process that arrives for queue 1 will preempt a process in queue 2. A process in queue 1 will in turn be preempted by a process arriving for queue 0.

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

This scheduling algorithm gives highest priority to any process with a CPU burst of 8 milliseconds or less. Such a process will quickly get the CPU, finish its CPU burst, and go off to its next I/O burst. Processes that need more than 8 but less than 24 milliseconds are also served quickly, although with lower priority than shorter processes. Long processes automatically sink to queue 2 and are served in FCFS order with any CPU cycles left over from queues 0 and 1.

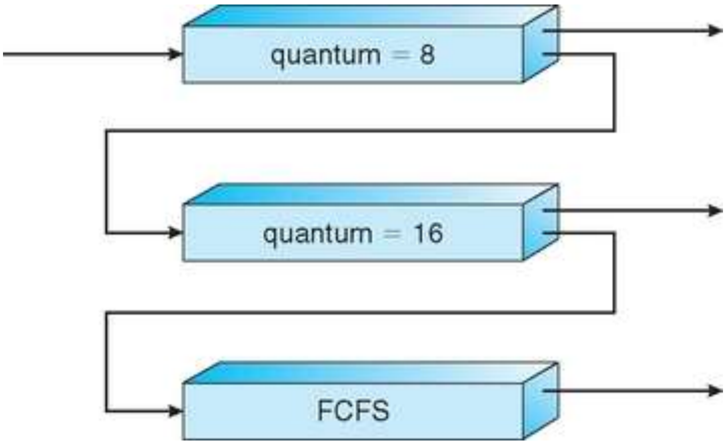


Figure 30: Multilevel feedback queues.

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

- The number of queues
- The scheduling algorithm for each queue
- The method used to determine when to upgrade a process to a higher-priority queue
- The method used to determine when to demote a process to a lower-priority queue
- The method used to determine which queue a process will enter when that process needs service

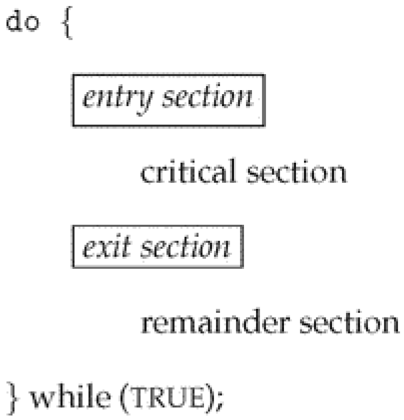
The definition of a multilevel feedback queue scheduler makes it the most general CPU-scheduling algorithm. It can be configured to match a specific system under design. Unfortunately, it is also the most complex algorithm, since defining the best scheduler requires some means by which to select values for all the parameters.

4. Process synchronization

A cooperating process is one that can affect or be affected by other processes executing in the system. Cooperating processes can either directly share a logical address space (that is, both code and data) or be allowed to share data only through files or messages. Concurrent access to shared data may result in data inconsistency, however. In this chapter, we discuss various mechanisms to ensure the orderly execution of cooperating processes that share a logical address space, so that data consistency is maintained.

4.1 The Critical-Section Problem

Consider a system consisting of n processes {P0, P1, ..., Pn-1}. Each process has a segment of code, called a critical section, in which the process may be changing common variables, updating a table, writing a file, and so on. The important feature of the system is that, when one process is executing in its critical section, no other process is to be allowed to execute in its critical section. That is, no two processes are executing in their critical sections at the same time. The critical-section problem is to design a protocol that the processes can use to cooperate. Each process must request permission to enter its critical section. The section of code implementing this request is the entry section. The critical section may be followed by an exit section. The remaining code is the remainder section. The general structure of a typical process Pi is shown below. The entry section and exit section are enclosed in boxes to highlight these important segments of code.



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

Figure 31: General Structure of a typical process Pi.

A solution to the critical-section problem must satisfy the following three requirements:

1. **Mutual exclusion**. If process Pi is executing in its critical section, then no other processes can be executing in their critical sections.

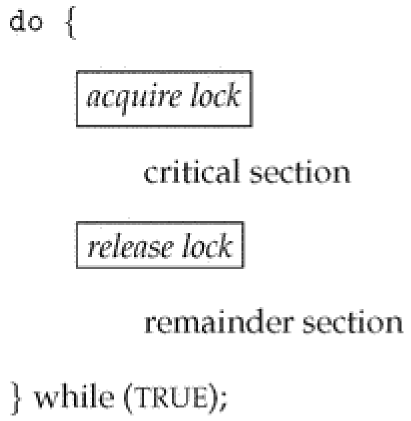
2. **Progress**. If no process is executing in its critical section and some processes wish to enter their critical sections, then only those processes that are not executing in their remainder sections can participate in deciding which will enter its critical section next, and this selection cannot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

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

We assume that each process is executing at a nonzero speed. However, we can make no assumption concerning the relative speed of the n processes.

4.2 Synchronization hardware

Instead, we can generally state that any solution to the critical-section problem requires a simple tool—a lock. Race conditions are prevented by requiring that critical regions be protected by locks. That is, a process must acquire a lock before entering a critical section; it releases the lock when it exits the critical section. This is illustrated below



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

Figure 32 : Solution to the critical-section problem using locks.

In the following discussions, we explore several more solutions to the critical-section problem using techniques ranging from hardware to software-based APIs available to application programmers. All these solutions are based on the premise of locking; however, as we shall see, the designs of such locks can be quite sophisticated.

We start by presenting some simple hardware instructions that are available on many systems and showing how they can be used effectively in solving the critical-section problem. Hardware features can make any programming task easier and improve system efficiency.

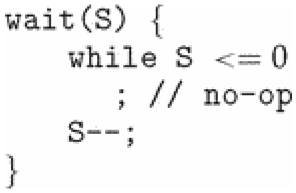
The critical-section problem could be solved simply in a uniprocessor environment if we could prevent interrupts from occurring while a shared variable was being modified. In this manner, we could be sure that the current sequence of instructions would be allowed to execute in order without preemption. No other instructions would be run, so no unexpected modifications could be made to the shared variable. This is often the approach taken by nonpreemptive kernels.

Unfortunately, this solution is not as feasible in a multiprocessor environment. Disabling interrupts on a multiprocessor can be time consuming, as the message is passed to all the processors. This message passing delays entry into each critical section, and system efficiency decreases. Also consider the effect on a system’s clock if the clock is kept updated by interrupts.

5. Semaphores

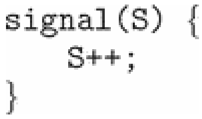
The hardware-based solutions to the critical-section problem presented in Section 6.4 are complicated for application programmers to use. To overcome this difficulty, we can use a synchronization tool called a semaphore.

A semaphore S is an integer variable that, apart from initialization, is accessed only through two standard atomic operations: wait() and signal(). The wait() operation was originally termed P (from the Dutch proberen, “to test”); signal() was originally called V (from verhogen, “to increment”). The definition of wait() is as follows:



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

The definition of signal() is as follows:

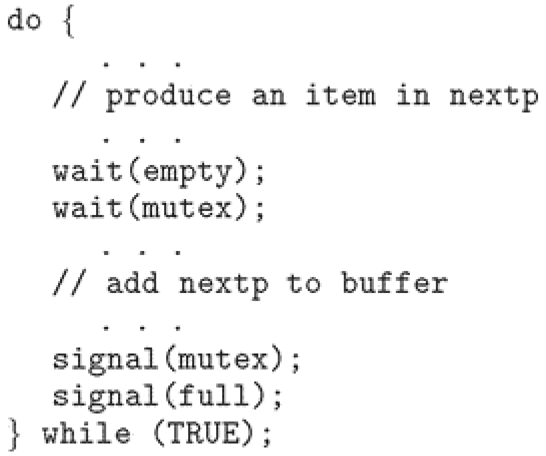


```
signal(S) {
   S++;
}
```

All modifications to the integer value of the semaphore in the wait() and signal() operations must be executed indivisibly. That is, when one process modifies the semaphore value, no other process can simultaneously modify that same semaphore value. In addition, in the case of wait(S), the testing of the integer value of S (S ≤ 0), as well as its possible modification (S--), must be executed without interruption.

6. Classic Problems of Synchronization

These problems are used for testing nearly every newly proposed synchronization scheme. In our solutions to the problems, we use semaphores for synchronization.



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

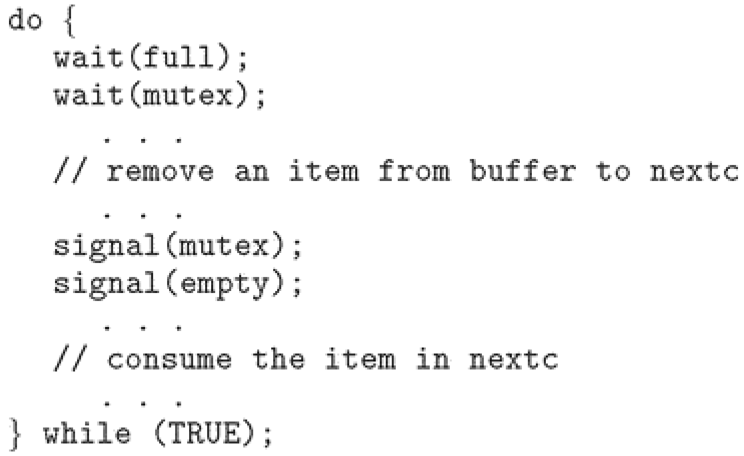
Figure 33 The structure of the producer process.

The Bounded-Buffer Problem

The bounded-buffer problem was introduced in Section 6.1; it is commonly used to illustrate the power of synchronization primitives. Here, we present a general structure of this scheme without committing ourselves to any particular implementation; we provide a related programming project in the exercises at the end of the chapter.

We assume that the pool consists of n buffers, each capable of holding one item. The mutex semaphore provides mutual exclusion for accesses to the buffer pool and is initialized to the value 1. The empty and full semaphores count the number of empty and full buffers. The semaphore empty is initialized to the value n; the semaphore full is initialized to the value 0.

The code for the producer process is shown in Figure 6.10; the code for the consumer process is shown in Figure 6.11. Note the symmetry between the producer and the consumer. We can interpret this code as the producer producing full buffers for the consumer or as the consumer producing empty buffers for the producer.



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

Figure 34 The structure of the consumer process.

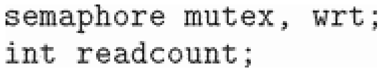
The Readers-Writers Problem

Suppose that a database is to be shared among several concurrent processes. Some of these processes may want only to read the database, whereas others may want to update (that is, to read and write) the database. We distinguish between these two types of processes by referring to the former as readers and to the latter as writers. Obviously, if two readers access the shared data simultaneously, no adverse effects will result. However, if a writer and some other process (either a reader or a writer) access the database simultaneously, chaos may ensue.

To ensure that these difficulties do not arise, we require that the writers have exclusive access to the shared database while writing to the database. This synchronization problem is referred to as the readers-writers problem. Since it was originally stated, it has been used to test nearly every new synchronization primitive. The readers-writers problem has several variations, all involving priorities. The simplest one, referred to as the first readers-writers problem, requires that no reader be kept waiting unless a writer has already obtained permission to use the shared object. In other words, no reader should wait for other readers to finish simply because a writer is waiting. The second readers- writers problem requires that, once a writer is ready, that writer performs its write as soon as possible. In other words, if a writer is waiting to access the object, no new readers may start reading.

A solution to either problem may result in starvation. In the first case, writers may starve; in the second case, readers may starve. For this reason, other variants of the problem have been proposed. Next, we present a solution to the first readers-writers problem. Refer to the bibliographical notes at the end of the chapter for references describing starvation-free solutions to the second readers-writers problem.

In the solution to the first readers-writers problem, the reader processes share the following data structures:

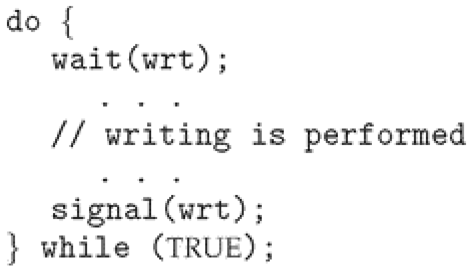


```
semaphore mutex, wrt;
int readcount;
```

The semaphores mutex and wrt are initialized to 1; readcount is initialized to 0. The semaphore wrt is common to both reader and writer processes. The mutex semaphore is used to ensure mutual exclusion when the variable readcount is updated. The readcount variable keeps track of how many processes are currently reading the object. The semaphore wrt functions as a mutual-exclusion semaphore for the writers. It is also used by the first or last reader that enters or exits the critical section. It is not used by readers who enter or exit while other readers are in their critical sections.

Note that, if a writer is in the critical section and n readers are waiting, then one reader is queued on wrt, and n - 1 readers are queued on mutex. Also observe that, when a writer executes signal(wrt), we may resume the execution of either the waiting readers or a single waiting writer. The selection is made by the scheduler.

The readers-writers problem and its solutions have been generalized to provide reader-writer locks on some systems. Acquiring a reader-writer lock requires specifying the mode of the lock: either read or write access. When a process wishes only to read shared data, it requests the reader-writer lock in read mode; a process wishing to modify the shared data must request the lock in write mode. Multiple processes are permitted to concurrently acquire a reader-writer lock in read mode, but only one process may acquire the lock for writing, as exclusive access is required for writers.



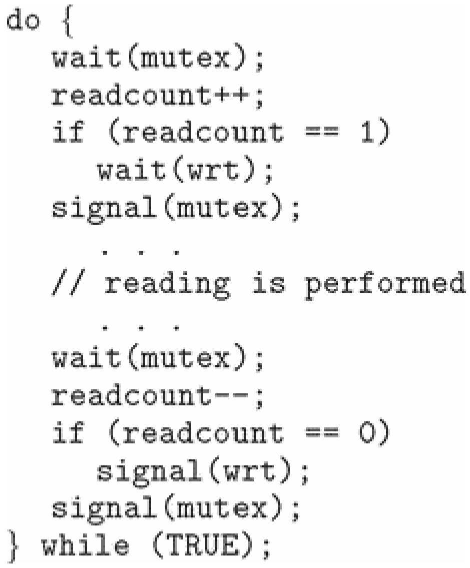
```
do {
   wait(wrt);
      . . .
   // writing is performed
      . . .
   signal(wrt);
} while (TRUE);
```

Figure 35 The structure of a writer process.

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

- In applications where it is easy to identify which processes only read shared data and which processes only write shared data.
- In applications that have more readers than writers. This is because reader- writer locks generally require more overhead to establish than semaphores or mutual-exclusion locks. The increased concurrency of allowing multiple readers compensates for the overhead involved in setting up the reader- writer lock.

The Dining-Philosophers Problem



```
do {
   wait(mutex);
   readcount++;
   if (readcount == 1)
      wait(wrt);
   signal(mutex);
      . . .
   // reading is performed
      . . .
   wait(mutex);
   readcount--;
   if (readcount == 0)
      signal(wrt);
   signal(mutex);
} while (TRUE);
```

Figure 36 The structure of a reader process.

Consider five philosophers who spend their lives thinking and eating. The philosophers share a circular table surrounded by five chairs, each belonging to one philosopher. In the center of the table is a bowl of rice, and the table is laid with five single chopsticks. When a philosopher thinks, she does not interact with her colleagues. From time to time, a philosopher gets hungry and tries to pick up the two chopsticks that are closest to her (the chopsticks that are between her and her left and right neighbors). A philosopher may pick up only one chopstick at a time. Obviously, she cannot pick up a chopstick that is already in the hand of a neighbor. When a hungry philosopher has both her chopsticks at the same time, she eats without releasing her chopsticks. When she is finished eating, she puts down both of her chopsticks and starts thinking again

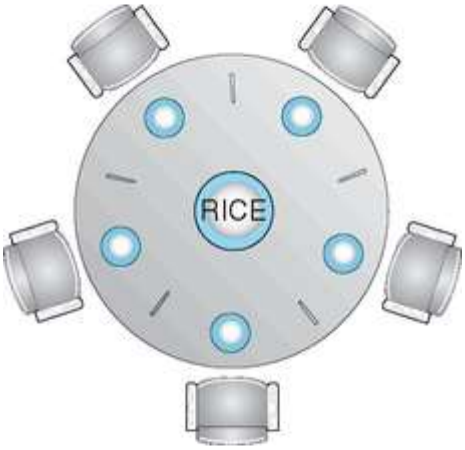
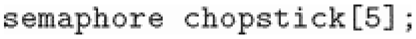


Figure 37 The situation of the dining philosophers.

The dining-philosophers problem is considered a classic synchronization problem neither because of its practical importance nor because computer scientists dislike philosophers but because it is an example of a large class of concurrency-control problems. It is a simple representation of the need to allocate several resources among several processes in a deadlock-free and starvation-free manner.

One simple solution is to represent each chopstick with a semaphore. A philosopher tries to grab a chopstick by executing a wait() operation on that semaphore; she releases her chopsticks by executing the signal() operation on the appropriate semaphores. Thus, the shared data are



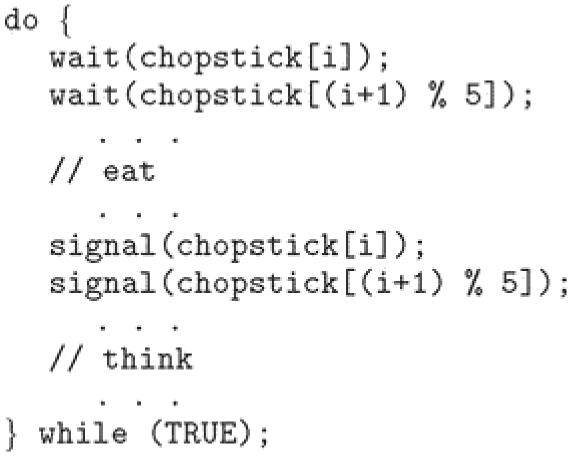
```
semaphore chopstick[5];
```

where all the elements of chopstick are initialized to 1.

Although this solution guarantees that no two neighbors are eating simultaneously, it nevertheless must be rejected because it could create a deadlock. Suppose that all five philosophers become hungry simultaneously and each grabs her left chopstick. All the elements of chopstick will now be equal to 0. When each philosopher tries to grab her right chopstick, she will be delayed forever.

Several possible remedies to the deadlock problem are listed next.

- Allow at most four philosophers to be sitting simultaneously at the table.
- Allow a philosopher to pick up her chopsticks only if both chopsticks are available (to do this, she must pick them up in a critical section).
- Use an asymmetric solution; that is, an odd philosopher picks up first her left chopstick and then her right chopstick, whereas an even philosopher picks up her right chopstick and then her left chopstick.



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

Figure 38 The structure of philosopher i.

we present a solution to the dining-philosophers problem that ensures freedom from deadlocks. Note, however, that any satisfactory solution to the dining-philosophers problem must guard against the possibility that one of the philosophers will starve to death. A deadlock-free solution does not necessarily eliminate the possibility of starvation.

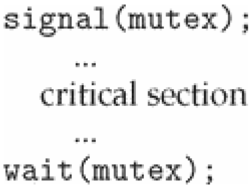
7. Monitors

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.

We have seen an example of such errors in the use of counters in our solution to the producer-consumer problem (Section 6.1). In that example, the timing problem happened only rarely, and even then the counter value appeared to be reasonable—off by only 1. Nevertheless, the solution is obviously not an acceptable one. It is for this reason that semaphores were introduced in the first place.

Unfortunately, such timing errors can still occur when semaphores are used. To illustrate how, we review the semaphore solution to the critical-section problem. All processes share a semaphore variable mutex, which is initialized to 1. Each process must execute wait(mutex) before entering the critical section and signal(mutex) afterward. If this sequence is not observed, two processes may be in their critical sections simultaneously. Next, we examine the various difficulties that may result. Note that these difficulties will arise even if a single process is not well behaved. This situation may be caused by an honest programming error or an uncooperative programmer.

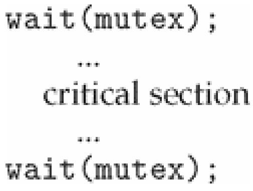
- Suppose that a process interchanges the order in which the wait() and signal() operations on the semaphore mutex are executed, resulting in the following execution:



```
signal(mutex);
    ...
   critical section
    ...
wait(mutex);
```

In this situation, several processes may be executing in their critical sections simultaneously, violating the mutual-exclusion requirement. This error may be discovered only if several processes are simultaneously active in their critical sections. Note that this situation may not always be reproducible.

- Suppose that a process replaces signal(mutex) with wait(mutex). That is, it executes



```
wait(mutex);
    ...
  critical section
    ...
wait(mutex);
```

In this case, a deadlock will occur.

• Suppose that a process omits the wait(mutex), or the signal(mutex), or both. In this case, either mutual exclusion is violated or a deadlock will occur.

These examples illustrate that various types of errors can be generated easily when programmers use semaphores incorrectly to solve the critical-section problem. Similar problems may arise in the other synchronization models discussed in Section 6.6.

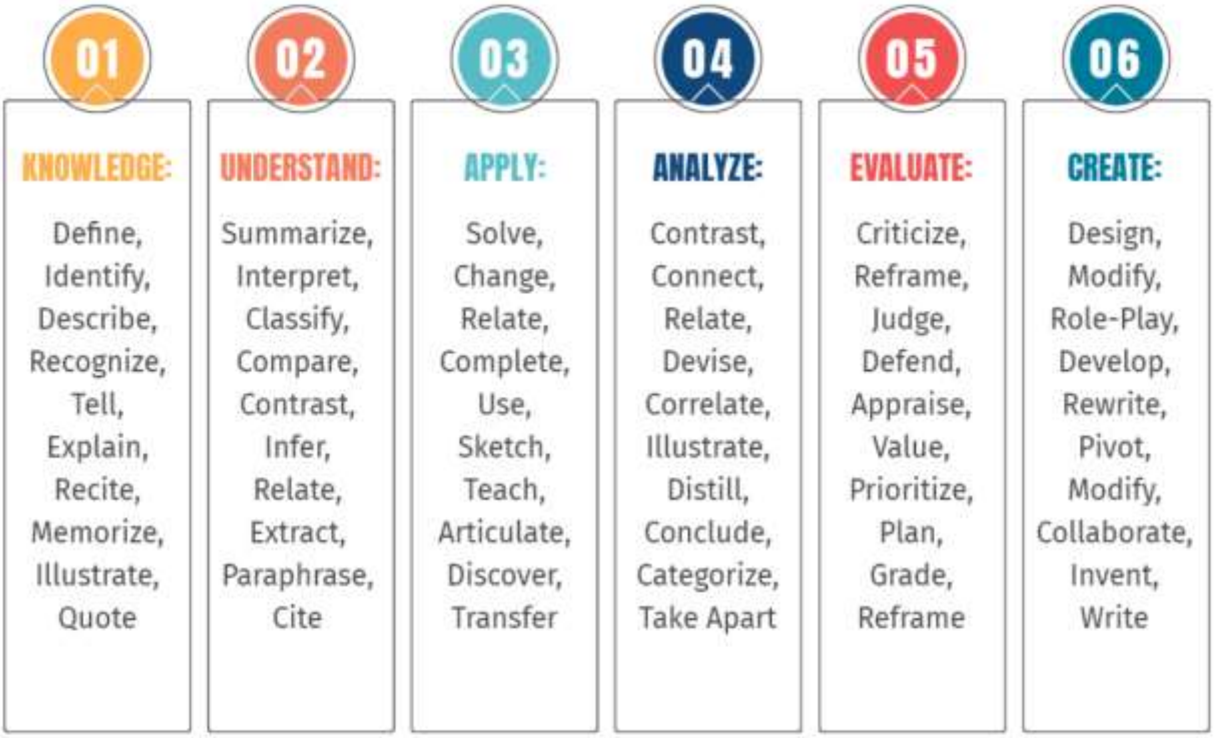
To deal with such errors, researchers have developed high-level language constructs. In this section, we describe one fundamental high-level synchronization construct—the monitor type.

Question Format :

Part-A			
Q.No	**Questions**	**Competence**	**BT Level**
1.	What is starvation? Explain with example.	Knowledge	BT L1
2.	What are the 3 different types of scheduling queues?	Knowledge	BT L1
3.	Define schedulers?	Knowledge	BT L1
4.	What are the types of scheduler?	Knowledge	BT L1
5.	Define critical section?	Knowledge	BT L1
6.	Define semaphores.	Knowledge	BT L1
7.	Name dome classic problem of synchronization?	Understanding	BT L2
8.	What is the use of cooperating processes?	Knowledge	BT L1
9.	Define race condition.	Knowledge	BT L1
10.	What are the requirements that a solution to the critical section problem must satisfy?	Knowledge	BT L1
Part-B			
Q.No	**Questions**	**Competence**	**BT Level**
1.	Explain the scheduling criteria	Knowledge	BT L1
2.	Explain FCFS scheduling algorithm with example.	Knowledge	BT L1
3.	Explain SJF scheduling algorithm with example	Knowledge	BT L1
4.	Explain Priority scheduling algorithm with example	Knowledge	BT L1
5.	Explain Round Robin scheduling algorithm with example.	Understand	BT L2
6.	Write about the various CPU scheduling algorithms	Create	BT L6
7.	Consider the following five processes, with the length of the CPU burst time given in milliseconds. Process Burst time P1 10 P2 29 P3 3 P4 7 P5 12 Consider the First come First serve (FCFS),	Analyse	BT L4

	Non Preemptive Shortest Job First(SJF), Round Robin(RR) (quantum=10ms) scheduling algorithms. Illustrate the scheduling using Gantt chart. Which		
8.	What is the important feature of critical section? State the Readers Writers problem and give solution using semaphore	Knowledge	BT L1

Bloom's Taxonomy



01 KNOWLEDGE: Define, Identify, Describe, Recognize, Tell, Explain, Recite, Memorize, Illustrate, Quote

02 UNDERSTAND: Summarize, Interpret, Classify, Compare, Contrast, Infer, Relate, Extract, Paraphrase, Cite

03 APPLY: Solve, Change, Relate, Complete, Use, Sketch, Teach, Articulate, Discover, Transfer

04 ANALYZE: Contrast, Connect, Relate, Devise, Correlate, Illustrate, Distill, Conclude, Categorize, Take Apart

05 EVALUATE: Criticize, Reframe, Judge, Defend, Appraise, Value, Prioritize, Plan, Grade, Reframe

06 CREATE: Design, Modify, Role-Play, Develop, Rewrite, Pivot, Modify, Collaborate, Invent, Write

UNIT – III - Operating Systems – SBSA 1301

UNIT 3

Deadlocks: Characterization- Methods for Handling Deadlocks - Deadlock Prevention - Avoidance - Detection - Recovery.

DEADLOCKS

System model:

A system consists of a finite number of resources to be distributed among a number of competing processes. The resources are partitioned into several types, each consisting of some number of identical instances. Memory space, CPU cycles, files, I/O devices are examples of resource types. If a system has 2 CPUs, then the resource type CPU has 2 instances.

A process must request a resource before using it and must release the resource after using it. A process may request as many resources as it requires to carry out its task. The number of resources as it requires to carry out its task. The number of resources requested may not exceed the total number of resources available in the system. A process cannot request 3 printers if the system has only two.

A process may utilize a resource in the following sequence:

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

(II)USE: The process can operate on the resource .if the resource is a printer, the process can print on the printer.

(III)RELEASE: The process release the resource.

For each use of a kernel managed by a process the operating system checks that the process has requested and has been allocated the resource. A system table records whether each resource is free (or) allocated. For each resource that is allocated, the table also records the process to which it is allocated. If a process requests a resource that is currently allocated to another process, it can be added to a queue of processes waiting for this resource.

To illustrate a deadlocked state, consider a system with 3 CDRW drives. Each of 3 processes holds one of these CDRW drives. If each process now requests another drive, the 3 processes will be in a deadlocked state. Each is waiting for the event “CDRW is released” which can be caused only by one of the other waiting processes. This example illustrates a deadlock involving the same resource type.

Deadlocks may also involve different resource types. Consider a system with one printer and one DVD drive. The process Pi is holding the DVD and process Pj is holding the printer. If Pi requests the printer and Pj requests the DVD drive, a deadlock occurs.

DEADLOCK CHARACTERIZATION:

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

NECESSARY CONDITIONS:

A deadlock situation can arise if the following 4 conditions hold simultaneously in a system:

1.MUTUAL EXCLUSION: Only one process at a time can use the resource. If another process requests that resource, the requesting process must be delayed until the resource has been released.

2.HOLD AND WAIT: A process must be holding at least one resource and waiting to acquire additional resources that are currently being held by other processes.

3.NO PREEMPTION: Resources cannot be preempted. A resource can be released only voluntarily by the process holding it, after that process has completed its task.

4.CIRCULAR WAIT: A set {P0,P1,…..Pn} of waiting processes must exist such that P0 is waiting for resource held by P1, P1 is waiting for a resource held by P2,……,Pn-1 is waiting for a resource held by Pn and Pn is waiting for a resource held byP0.

RESOURCE ALLOCATION GRAPH

Deadlocks can be described more precisely in terms of a directed graph called a system resource allocation graph. This graph consists of a set of vertices V and a set of edges E. the set of vertices V is partitioned into 2 different types of nodes:

P = {P1, P2….Pn}, the set consisting of all the active processes in the system. R= {R1, R2….Rm}, the set consisting of all resource types in the system.

A directed edge from process Pi to resource type Rj is denoted by Pi ->Rj. It signifies that process

Pi has requested an instance of resource type Rj and is currently waiting for that resource.

A directed edge from resource type Rj to process Pi is denoted by Rj ->Pi, it signifies that an instance of resource type Rj has been allocated to process Pi.

A directed edge Pi ->Rj is called a requested edge. A directed edge Rj->Pi is called an assignment edge.

We represent each process Pi as a circle, each resource type Rj as a rectangle. Since resource type Rj may have more than one instance. We represent each such instance as a dot within the rectangle. A request edge points to only the rectangle Rj. An assignment edge must also designate one of the dots in the rectangle.

When process Pi requests an instance of resource type Rj, a request edge is inserted in the resource allocation graph. When this request can be fulfilled, the request edge is instantaneously transformed to an assignment edge. When the process no longer needs access to the resource, it releases the resource, as a result, the assignment edge is deleted.

The sets P, R, E:

P= {P1, P2, P3}

R= {R1, R2, R3, R4}

E= {P1 ->R1, P2 ->R3, R1 ->P2, R2 ->P2, R2 ->P1, R3 ->P3}

One instance of resource type R1

Two instances of resource type R2 One instance of resource type R3 Three instances of resource type R4 PROCESS STATES:

Process P1 is holding an instance of resource type R2 and is waiting for an instance of resource type R1.

Process P2 is holding an instance of R1 and an instance of R2 and is waiting for instance of R3. Process P3 is holding an instance of R3.

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

Suppose that process P3 requests an instance of resource type R2. Since no resource instance is currently available, a request edge P3 ->R2 is added to the graph.

2cycles:

P1 ->R1 ->P2 ->R3 ->P3 ->R2 ->P1 P2 ->R3 ->P3 ->R2 ->P2

Processes P1, P2, P3 are deadlocked. Process P2 is waiting for the resource R3, which is held by process P3.process P3 is waiting for either process P1 (or) P2 to release resource R2. In addition, process P1 is waiting for process P2 to release resource R1.

We also have a cycle: P1 ->R1 ->P3 ->R2 ->P1

However there is no deadlock. Process P4 may release its instance of resource type R2. That resource can then be allocated to P3, breaking the cycle.

Methods for handling deadlock

There are three ways to handle deadlock

1) Deadlock prevention or avoidance: The idea is to not let the system into a deadlock state.

One can zoom into each category individually, Prevention is done by negating one of above mentioned necessary conditions for deadlock.

Avoidance is kind of futuristic in nature. By using strategy of “Avoidance”, we have to make an assumption. We need to ensure that all information about resources which process will need are known to us prior to execution of the process. We use Banker’s algorithm (Which is in-turn a gift from Dijkstra) in order to avoid deadlock.

2) Deadlock detection and recovery: Let deadlock occur, then do preemption to handle it once occurred.

3) Ignore the problem altogether: If deadlock is very rare, then let it happen and reboot the system. This is the approach that both Windows and UNIX take.

DEADLOCK PREVENTION

For a deadlock to occur, each of the 4 necessary conditions must held. By ensuring that at least one of these conditions cannot hold, we can prevent the occurrence of a deadlock.

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

oRequire process to request and be allocated all its resources before it begins execution, or allow process to request resources only when the process has none

o Low resource utilization; starvation possible

No Preemption –

oIf 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

oPreempted resources are added to the list of resources for which the process is waiting

oProcess 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 .

Safe State

•When a process requests an available resource, system must decide if immediate allocation leaves the system in a safe state

System is in safe state if there exists a sequence <P1, P2, …, Pn> of ALL the processes in the systems such that for each Pi, the resources that Pi can still request can be satisfied by currently available resources + resources held by all the Pj, with j <I

That is:

- If Pi resource needs are not immediately available, then Pi can wait until all
- Pj have finished
- When Pj is finished, Pi can obtain needed resources, execute, return allocated resources, and terminate
- When Pi terminates, Pi +1 can obtain its needed resources, and so on If a system is in safe state no deadlocks

If a system is in unsafe statepossibility of deadlock Avoidance ensure that a system will never enter an unsafe state Avoidance algorithms

Single instance of a resource type

- Use a resource-allocation graph Multiple instances of a resource type
- Use the banker's algorithm

Resource-Allocation Graph Scheme

Claim edge Pi Æ 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 Request edge converted to an assignment edge when the resource is allocated to the process When a resource is released by a process, assignment edge reconverts to a claim edge Resources must be claimed a priori in the system

Unsafe State In Resource-Allocation Graph

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 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

Rj available

Max: n x m matrix. If Max [i,j] = k, then process Pi may request at most k

instances of resource type Rj

Allocation: n x m matrix. If Allocation[i,j] = k then Pi is currently allocated k instances of Rj

Need: n x m matrix. If Need[i,j] = k, then Pi may need k more instances of

Rj 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 = 0, 1, …,n- 1

2.Find an 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 Pi

Request = request vector for process Pi. If Request i[j] = k then process Pi wants

k instances of resource type Rj

1.If Request is Need i go to step

2. Otherwise, raise error condition, since process has exceeded its maximum claim

2.If Request is Available, go to step 3. Otherwise Pi must wait, since resources are not available

3.Pretend to allocate requested resources to Pi by modifying the state as follows:

Available = Available – Request; Allocation i=Allocation i + Request i; Need i=Need i – Request i;

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

Example of Banker's Algorithm

consider 5 processes P0 through P4; 3 resource types:

A (10 instances), B (5instances), and C (7 instances)

Snapshot at time T0:

Allocation	Max	Available
A B C	A B C	A B C
P0 0 1 0	7 5 3	3 3 2
P1 2 0 0	3 2 2	
P2 3 0 2	9 0 2	
P3 2 1 1	2 2 2	
P4 0 0 2	4 3 3	

Σ The content of the matrix Need is defined to be Max

– Allocation Need A B C

The system is in a safe state since the sequence <P1, P3, P4, P2, P0>

satisfies safety criteria

P1 Request (1,0,2)

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
P0 0 1 0	7 4 3	2 3 0
P1 3 0 2	0 2 0	
P2 3 0 2	6 0 0	
P3 2 1 1	0 1 1	
P4 0 0 2	4 3 1	

Executing safety algorithm shows that sequence <P1, P3, P4, P0, P2> satisfies safety requirement

Deadlock Detection

Allow system to enter deadlock state Detection algorithm

Recovery scheme

Single Instance of Each Resource Type

Maintain wait-for graph Nodes are processes Pi Æ Pj if Pi is waiting for Pj

Periodically invoke an algorithm that searches for a cycle in the graph. If there is a cycle, there exists a deadlock

An algorithm to detect a cycle in a graph requires an order of n2 operations, where n is the number of vertices in the graph

Resource-Allocation Graph and Wait-for Graph

Resource-Allocation GraphCorresponding 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 Pi is requesting k more instances of resource type. Rj.

Detection Algorithm

Let Work and Finish be vectors of length m and n, respectively Initialize:

(a)Work = Available

(b)For i = 1,2, …, n, if Allocation iπ 0, then Finish[i] = false; otherwise, Finish[i] = true

2.Find an index i such that both:

(a)Finish[i] == false

(b)Request is 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 if n, then the system is in deadlock state. Moreover, if

Finish[i] == false, then Pi is deadlocked

Deadlock avoidance

In deadlock avoidance, the request for any resource will be granted if the resulting state of the system doesn't cause deadlock in the system. The state of the system will continuously be checked for safe and unsafe states.

In order to avoid deadlocks, the process must tell OS, the maximum number of resources a process can request to complete its execution.

The simplest and most useful approach states that the process should declare the maximum number of resources of each type it may ever need. The Deadlock avoidance algorithm examines the resource allocations so that there can never be a circular wait condition.

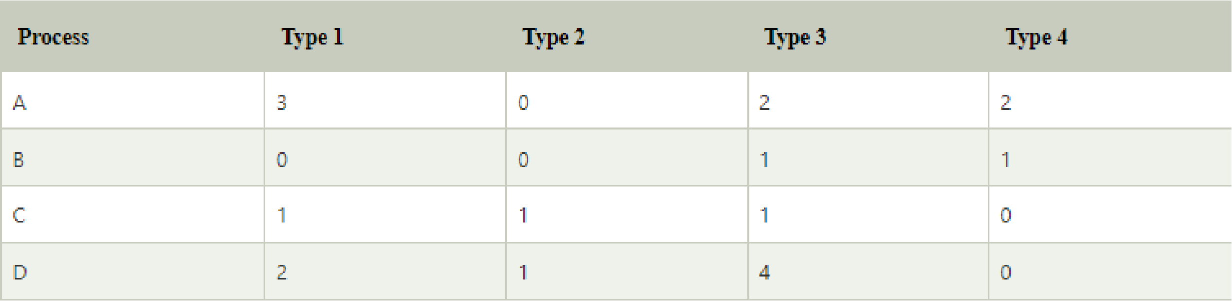
Safe and Unsafe States

The resource allocation state of a system can be defined by the instances of available and allocated resources, and the maximum instance of the resources demanded by the processes.

A state of a system recorded at some random time is shown below.

Resources Assigned

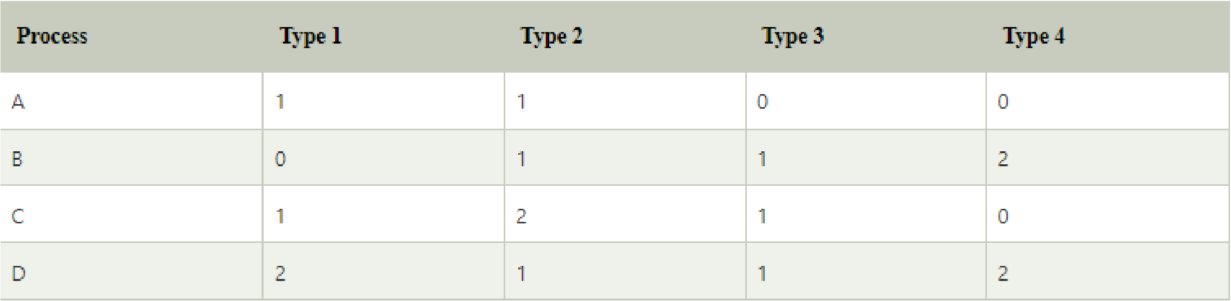
Table :



Process	Type 1	Type 2	Type 3	Type 4
A	3	0	2	2
B	0	0	1	1
C	1	1	1	0
D	2	1	4	0

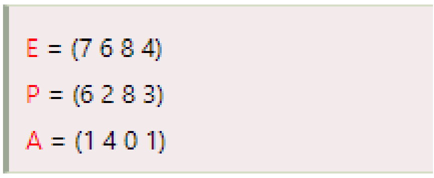
Resources still needed

Table:



Process	Type 1	Type 2	Type 3	Type 4
A	1	1	0	0
B	0	1	1	2
C	1	2	1	0
D	2	1	1	2

Table:



E = (7 6 8 4)
P = (6 2 8 3)
A = (1 4 0 1)

Above tables and vector E, P and A describes the resource allocation state of a system. There are 4 processes and 4 types of the resources in a system. Table 1 shows the instances of each resource assigned to each process.

Table shows the instances of the resources, each process still needs. Vector E is the representation of total instances of each resource in the system.

Vector P represents the instances of resources that have been assigned to processes. Vector A represents the number of resources that are not in use.

A state of the system is called safe if the system can allocate all the resources requested by all the processes without entering into deadlock.

If the system cannot fulfill the request of all processes then the state of the system is called unsafe.

The key of Deadlock avoidance approach is when the request is made for resources then the request must only be approved in the case if the resulting state is also a safe state.

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?

Resource Preemption

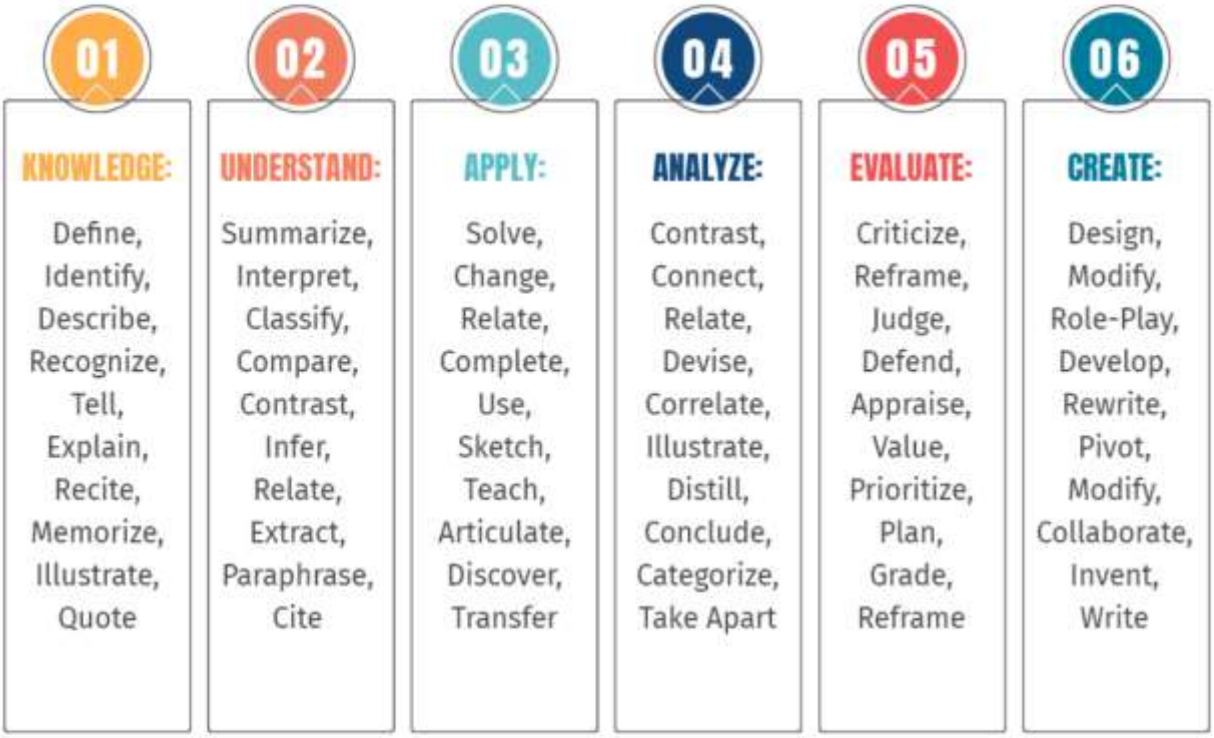
Selecting a victim – minimize cost

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

Question Format :

Part-A			
Q.No	**Questions**	**Competence**	**BT Level**
1.	Write the resource allocation algorithm for dead lock?	Create	BT L6
2.	What is deadlock?	Knowledge	BT L1
3.	What are necessary conditions for deadlocks?	Knowledge	BT L1
4.	State the characters of deadlock.	Knowledge	BT L1
5.	State the methods of handling deadlocks.	Knowledge	BT L1
6.	State the necessary conditions for deadlock avoidance.	Knowledge	BT L1
7.	State the necessary conditions for deadlock prevention.	Knowledge	BT L1
8.	What is mutual exclusion?	Knowledge	BT L1
9.	What causes deadock?	Knowledge	BT L1
10.	What is hold and wait in deadlock?	Knowledge	BT L1
Part-B			
Q.No	**Questions**	**Competence**	**BT Level**
1.	Explain about Deadlock Prevention	Knowledge	BT L1
2.	Explain about Deadlock Avoidance.	Knowledge	BT L1
3.	Explain about necessary conditions of deadlock	Knowledge	BT L1
4.	Explain about resource allocation graph(RAG)?	Knowledge	BT L1
5.	Explain about recovery from deadlock?	Knowledge	BT L1
6.	What is deadlock? Explain deadlock recovery in detail.	Knowledge	BT L1
7.	Explain deadlock detection and recovery.	Knowledge	BT L1

Bloom's Taxonomy



01	02	03	04	05	06
KNOWLEDGE:	UNDERSTAND:	APPLY:	ANALYZE:	EVALUATE:	CREATE:
Define,	Summarize,	Solve,	Contrast,	Criticize,	Design,
Identify,	Interpret,	Change,	Connect,	Reframe,	Modify,
Describe,	Classify,	Relate,	Relate,	Judge,	Role-Play,
Recognize,	Compare,	Complete,	Devise,	Defend,	Develop,
Tell,	Contrast,	Use,	Correlate,	Appraise,	Rewrite,
Explain,	Infer,	Sketch,	Illustrate,	Value,	Pivot,
Recite,	Relate,	Teach,	Distill,	Prioritize,	Modify,
Memorize,	Extract,	Articulate,	Conclude,	Plan,	Collaborate,
Illustrate,	Paraphrase,	Discover,	Categorize,	Grade,	Invent,
Quote	Cite	Transfer	Take Apart	Reframe	Write

UNIT – IV - Operating Systems – SBSA 1301

UNIT-IV

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

1. Memory Management

As a result of CPU scheduling, we can improve both the utilization of the CPU and the speed of the computer's response to its users. To realize this increase in performance, however, we must keep several processes in memory; that is, we must share memory.

The memory management algorithms vary from a primitive bare-machine approach to paging and segmentation strategies. Each approach has its own advantages and disadvantages. Selection of a memory-management method for a specific system depends on many factors, especially on the hardware design of the system.

Memory is central to the operation of a modern computer system. Memory consists of a large array of words or bytes, each with its own address. The CPU fetches instructions from memory according to the value of the program counter. These instructions may cause additional loading from and storing to specific memory addresses. A typical instruction-execution cycle, for example, first fetches an instruction from memory. The instruction is then decoded and may cause operands to be fetched from memory. After the instruction has been executed on the operands, results may be stored back in memory. The mernory unit sees only a stream of memory addresses; it does not know how they are generated (by the instruction counter, indexing, indirection, literal addresses, and so on) or what they are for (instructions or data). Accordingly, we can ignore hozu a program generates a memory address. We are interested only in the sequence of memory addresses generated by the running program.

Main memory and the registers built into the processor itself are the only storage that the CPU can access directly. There are machine instructions that take memory addresses as arguments, but none that take disk addresses. Therefore, any instructions in execution, and any data being used by the instructions, must be in one of these direct-access storage devices. If the data are not in memory, they must be moved there before the CPU can operate on them. Registers that are built into the CPU are generally accessible within one cycle of the CPU clock. Most CPUs can decode instructions and perform simple operations on register contents at the rate of one or more operations per clock tick. The same cannot be said of main memory, which is accessed via a transaction on the memory bus. Completing a memory access may take many cycles of the CPU clock. In such cases, the processor normally needs to stall, since it does not have the data required to complete the instruction that it is executing. This situation is intolerable because of the frequency of memory accesses. The remedy is to add fast memory between the CPU and main memory. A memory buffer used to accommodate a speed differential, called a cache.

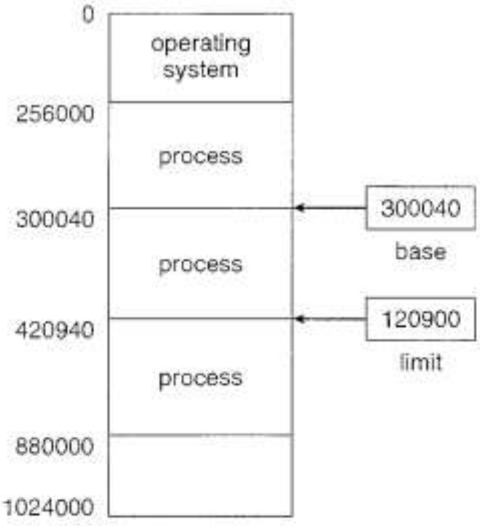


Figure A base and a limit register define a logical address space.

Figure 38 :A base and a limit register define a logical address space

Protection of memory space is accomplished by having the CPU hardware compare every address generated in user mode with the registers. Any attempt by a program executing in user mode to access operating-system memory or other users' memory results in a trap to the operating system, which treats the attempt as a fatal error. This scheme prevents a user program from (accidentally or deliberately) modifying the code or data structures of either the operating system or other users. The base and limit registers can be loaded only by the operating system, which uses a special privileged instruction. Since privileged instructions can be executed only in kernel mode, and since only the operating system executes in kernel mode, only the operating system can load the base and limit registers. This scheme allows the operating system to change the value of the registers but prevents user programs from changing the registers' contents. The operating system, executing in kernel mode, is given unrestricted access to both operating system memory and users' memory. This provision allows the operating system to load users' programs into users' memory, to dump out those programs in case of errors, to access and modify parameters of system calls, and so on.

2. Address Binding

Usually, a program resides on a disk as a binary executable file. To be executed, the program must be brought into memory and placed within a process. Depending on the memory management in use, the process may be moved between disk and memory during its execution. The processes on the disk that are waiting to be brought into memory for execution form the normal procedure is to select one of the processes in the input queue and to load that process into memory. As the process is executed, it accesses instructions and data from memory. Eventually, the process terminates, and its memory space is declared available. Most systems allow a user process to reside in any part of the physical memory. Thus, although the address space of the computer starts at 00000, the first address of the user process need not be 00000. This approach affects the addresses that the user program can use. In most cases, a user program will go through several steps-some of which may be optional-before being executed. Addresses may be represented in different ways during these steps. Addresses in the source program are generally symbolic (such as count). A compiler will typically bind these symbolic addresses to relocatable addresses (such as "14 bytes from the beginning of this module"). The linkage editor or loader will in turn bind the relocatable addresses to absolute addresses (such as 74014). Each binding is a mapping from one address space to another. Classically, the binding of instructions and data to memory addresses can be done at any step along the way:

Compile time: If you know at compile time where the process will reside in memory, then absolute code can be generated. For example, if you krww that a user process will reside starting at location R, then the generated compiler code will start at that location and extend up from there. If, at some later time, the starting location changes, then it will be necessary to recompile this code. The MS-DOS .COM-format programs are bound at compile time.

Load time: If it is not known at compile time where the process will reside in memory, then the compiler must generate relocatable code. In this case, final binding is delayed until load time. If the starting addresses changes, we need only reload the user code to incorporate this changed value.

Execution time: If the process can be moved during its execution from one memory segment to another, then binding must be delayed until run time. Special hardware must be available for this scheme to work. Most general-purpose operating systems use this method.

3. Dynamic Loading

It has been necessary for the entire program and all data of a process to be in physical memory for the process to execute. The size of a process has thus been limited to the size of physical memory. To obtain better memory-space utilization, we can use dynamic loading. With dynamic loading, a routine is not loaded until it is called. All routines are kept on disk in a relocatable load format. The main program is loaded into memory and is executed. When a routine needs to call another routine, the calling routine first checks to see whether the other routine has been loaded. If it has not, the relocatable linking loader is called to load the desired routine into menwry and to update the program's address tables to reflect this change. Then control is passed to the newly loaded routine.

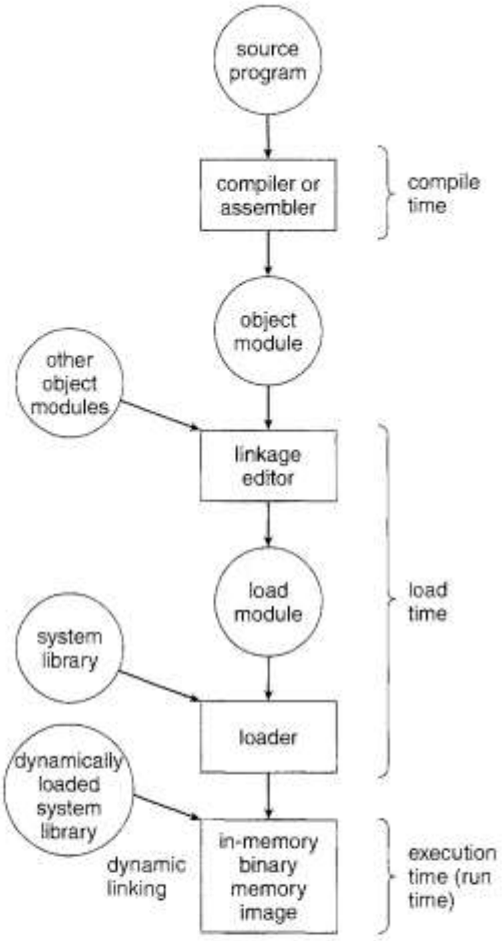


Figure Multistep processing of a user program.

Figure 39:Multiprocessing of a User Program

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

4. Dynamic Linking and Shared Libraries

Some operating systems support only static linking, in which system language libraries are treated like any other object module and are combined by the loader into the binary program image. Dynamic linking, in contrast, is similar to dynamic loading. Here, though, linking, rather than loading, is postponed until execution time. This feature is usually used with system libraries, such as language subroutine libraries. Without this facility, each program on a system must include a copy of its language library (or at least the routines referenced by the program) in the executable image. This requirement wastes both disk space and main memory.

With dynamic linking, a stub is included in the image for each library routine reference. The stub is a small piece of code that indicates how to locate the appropriate memory-resident library routine or how to load the library if the routine is not already present. When the stub is executed, it checks to see whether the needed routine is already in memory. If it is not, the program loads the routine into memory. Either way, the stub replaces itself with the address of the routine and executes the routine. Thus, the next time that particular code segment is reached, the library routine is executed directly, incurring no cost for dynamic linking. Under this scheme, all processes that use a language library execute only one copy of the library code.

Unlike dynamic loading, dynamic linking generally requires help from the operating system. If the processes in memory are protected from one another, then the operating system is the only entity that can check to see whether the needed routine is in another process's memory space or that can allow multiple processes to access the same memory addresses.

5. Overlays

The main problem in fixed partitioning is the size of a process has to be limited by the maximum size of the partition, which means a process can never be span over another. In order to solve this problem, earlier people have used some solution which is called as Overlays.

The concept of overlays is that whenever a process is running it will not use the complete program at the same time, it will use only some part of it. Then overlays concept says that whatever part you required, you load it an once the part is done, then you just unload it, means just pull it back and get the new part you required and run it.

Formally, the process of transferring a block of program code or other data into internal memory, replacing what is already stored. Sometimes it happens that compare to the size of the biggest partition, the size of the program will be even more, then, in that case, you should go with overlays.

So overlay is a technique to run a program that is bigger than the size of the physical memory by keeping only those instructions and data that are needed at any given time. Divide the program into modules in such a way that not all modules need to be in the memory at the same time.

Advantages

- Reduce memory requirement
- Reduce time requirement

Disadvantages

- Overlap map must be specified by programmer
- Programmer must know memory requirement
- Overlapped module must be completely disjoint
- Programming design of overlays structure is complex and not possible in all cases

Example –

The best example of overlays is assembler.Consider the assembler has 2 passes, 2 pass means at any time it will be doing only one thing, either the 1st pass or the 2nd pass.Which means it will finish 1st pass first and then 2nd pass.Let assume that available main memory size is 150KB and total code size is 200KB.

Pass 1.......................70KB

Pass 2.......................80KB

Symbol table.................30KB

Common routine...............20KB

As the total code size is 200KB and main memory size is 150KB, it is not possible to use 2 passes together.So, in this case, we should go with the overlays technique.According to the overlays concept at any time only one pass will be used and both the passes always need symbol table and common routine.Now the question is if overlays-driver* is 10KB, then what is the minimum partition size required?For pass 1 total memory needed is = (70KB + 30KB + 20KB + 10KB) = 130KB and for pass 2 total memory needed is = (80KB + 30KB + 20KB + 10KB) = 140KB.So if we have minimum 140KB size partition then we can run this code very easily.

Overlays driver:-It is the user responsibility to take care of overlaying; the operating system will not provide anything. Which means the user should write even what part is required in the 1st pass and once the 1st pass is over, the user should write the code to pull out the pass 1 and load the pass 2.That is what is the responsibility of the user, that is known as the Overlays driver. Overlays driver will just help us to move out and move in the various part of the code.

6. Logical and Physical Address

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

The hardware device called Memory-Management Unit is used for mapping logical address to its corresponding physical address.

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

Differences between Logical and Physical Address in Operating System

1. The basic difference between Logical and physical address is that Logical address is generated by CPU in perspective of a program whereas the physical address is a location that exists in the memory unit.
2. Logical Address Space is the set of all logical addresses generated by CPU for a program whereas the set of all physical address mapped to corresponding logical addresses is called Physical Address Space.

3. The logical address does not exist physically in the memory whereas physical address is a location in the memory that can be accessed physically.
4. Identical logical addresses are generated by Compile-time and Load time address binding methods whereas they differs from each other in run-time address binding method. Please refer this for details.
5. The logical address is generated by the CPU while the program is running whereas the physical address is computed by the Memory Management Unit (MMU).

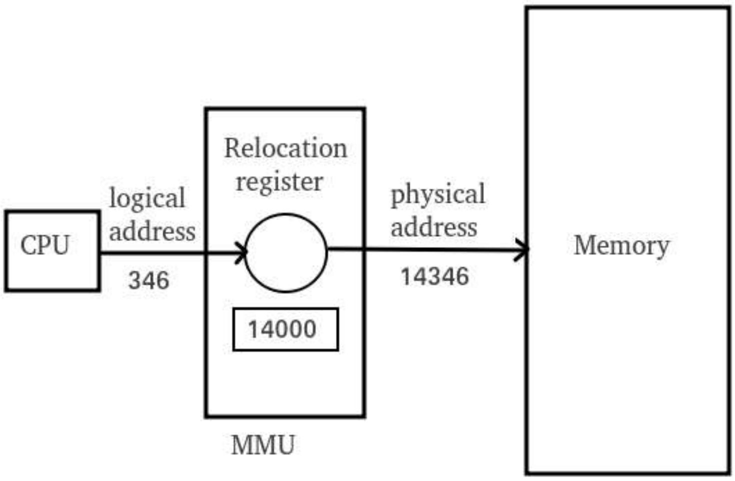


Figure 40:Memory Management Unit

7. **Contiguous Memory Allocation**

Contiguous memory allocation is basically a method in which a single contiguous section/part of memory is allocated to a process or file needing it. Because of this all the available memory space resides at the same place together, which means that the freely/unused available memory partitions are not distributed in a random fashion here and there across the whole memory space.

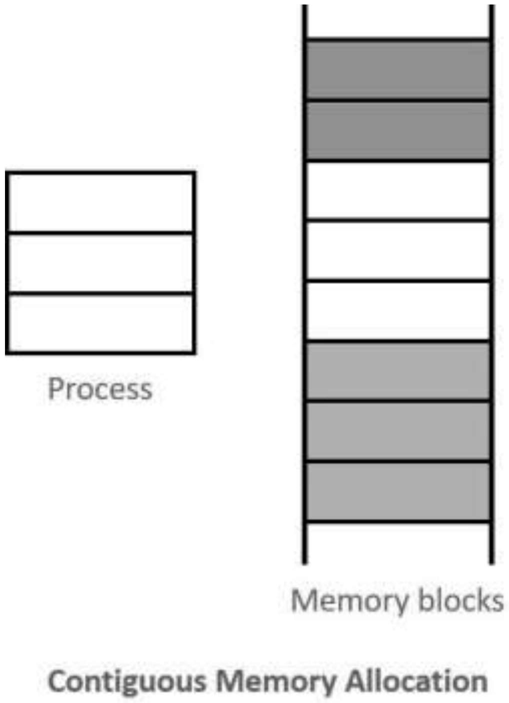
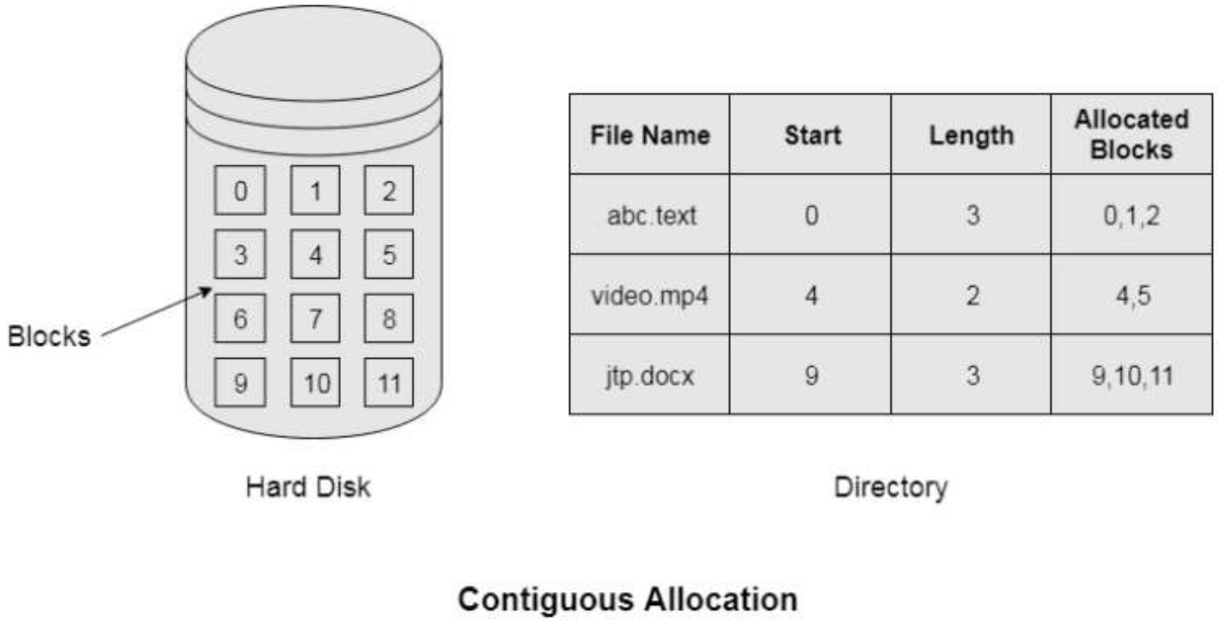


Figure 41:Contiguous Memory Allocation

The main memory is a combination of two main portions- one for the operating system and other for the user program. We can implement/achieve contiguous memory allocation by dividing the memory partitions into fixed size partitions.



File Name	Start	Length	Allocated Blocks
abc.text	0	3	0,1,2
video.mp4	4	2	4,5
jtp.docx	9	3	9,10,11

Figure 42: Contiguous Allocation

In the image shown below, there are three files in the directory. The starting block and the length of each file are mentioned in the table. We can check in the table that the contiguous blocks are assigned to each file as per its need.

Advantages

1. It is simple to implement
2. We will get Excellent read performance
3. Supports Random Access into files

Disadvantages

1. The disk will become fragmented
2. It may be difficult to have a file grow

8. **Non-Contiguous Memory Allocation**

Non-Contiguous memory allocation is basically a method on the contrary to contiguous allocation method, allocates the memory space present in different locations to the process as per its requirements. As all the available memory space is in a distributed pattern so the freely available memory space is also scattered here and there.
This technique of memory allocation helps to reduce the wastage of memory, which eventually gives rise to Internal and external fragmentation.

In non-contiguous allocation, Operating system needs to maintain the table which is called Page Table for each process which contains the base address of the each block which is acquired by the process in memory space. In non-contiguous memory allocation, different parts of a process are allocated different places in Main Memory. Spanning is allowed which is not possible in other techniques like Dynamic or Static Contiguous memory allocation. That's why paging is needed to ensure effective memory allocation. Paging is done to remove External Fragmentation.

Here a process can be spanned across different spaces in main memory in non-consecutive manner. Suppose process P of size 4KB. Consider main memory have two empty slots each of size 2KB. Hence total free space is, 2*2= 4 KB. In contiguous memory allocation, process P cannot be accommodated as spanning is not allowed.

In contiguous allocation, space in memory should be allocated to whole process. If not, then that space remains unallocated. But in Non-Contiguous allocation, process can be divided into different parts and hence filling the space in main memory. In this example, process P can be divided into two parts of equal size – 2KB. Hence one part of process P can be allocated to first 2KB space of main memory and other part of processP can be allocated to second 2KB space of main memory.

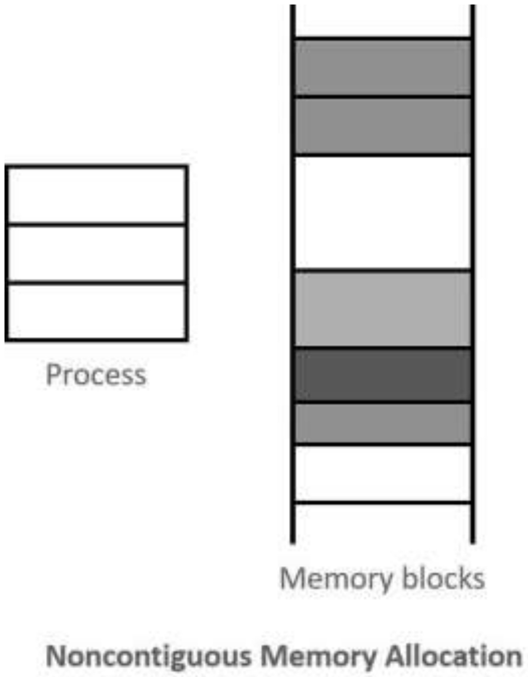


Figure 43 :Non Contiguous Memory Allocation

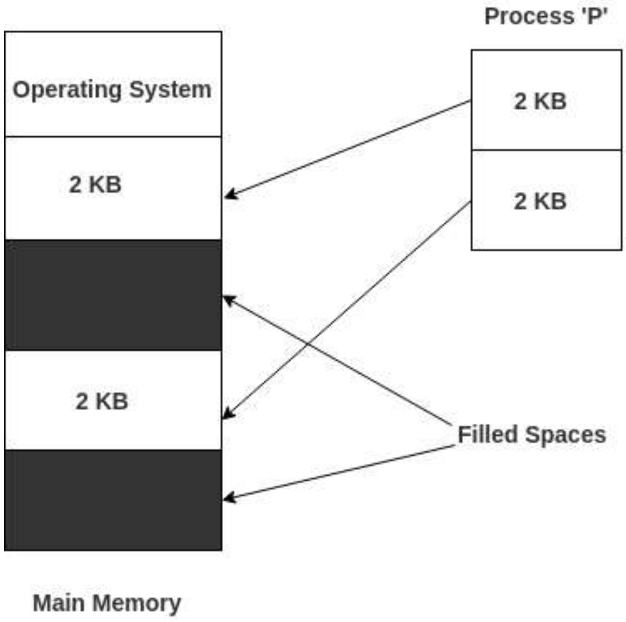


Figure:44 Non Contiguous Memory Allocation

But, in what manner we divide a process to allocate them into main memory is very important to understand. Process is divided after analysing the number of empty spaces and their size in main memory. Then only we divide our process. It is very time consuming process. Their number as well as their sizes changing every time due to execution of already present processes in main memory.

In order to avoid this time consuming process, we divide our process in secondary memory in advance before reaching the main memory for its execution. Every process is divided into various parts of equal size called Pages. We also divide our main memory into different parts of equal size called Frames.

Size of page in process = Size of frame in memory

Although their numbers can be different. Below diagram will make you understand in better way: consider empty main memory having size of each frame is 2 KB, and two processes P1 and P2 are 2 KB each

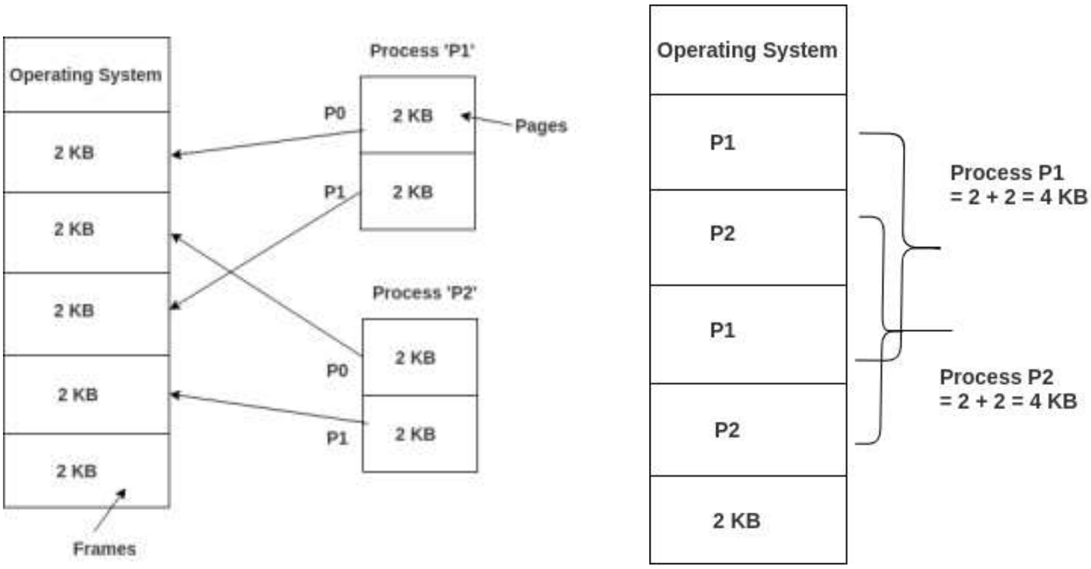


Figure 45 :Paging Memory Allocation

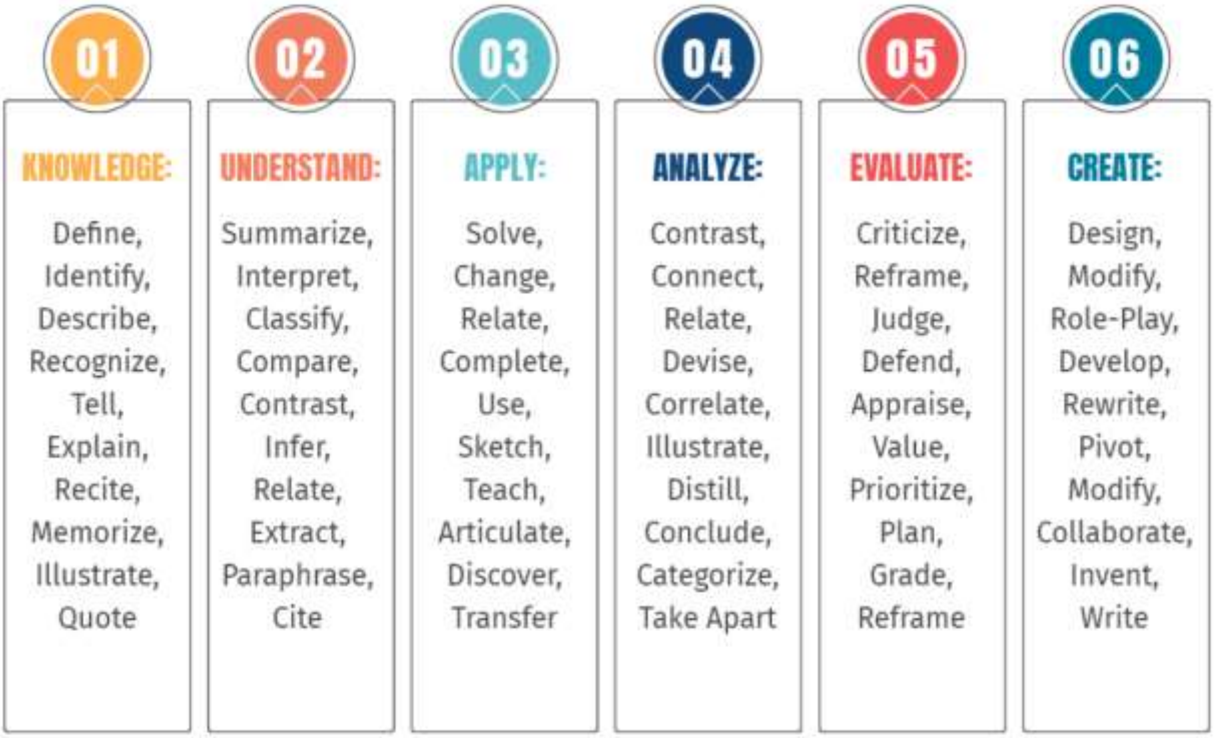
In conclusion we can say that, Paging allows memory address space of a process to be non-contiguous. Paging is more flexible as only pages of a process are moved. It allows more processes to reside in main memory than Contiguous memory allocation.

Question Format :

Part-A			
Q.No	**Questions**	**Competence**	**BT Level**
1.	Define logical address and physical address.	Knowledge	BT L1
2.	What is the main function of the memory-management unit?	Knowledge	BT L1
3.	What is logical address space and physical address space?	Knowledge	BT L1
4.	Define dynamic loading.	Knowledge	BT L1
5.	Define dynamic linking	Knowledge	BT L1
6.	What are overlays?	Knowledge	BT L1
7.	Define swapping.	Knowledge	BT L1
8.	Define Demand paging and write advantages.	Knowledge	BT L1
9.	What is address binding?	Knowledge	BT L1
10.	Define Overlays	Knowledge	BT L1

Part-B			
Q.No	**Questions**	**Competence**	**BT Level**
1.	Write in detail about the contiguous memory storage	Understand	BT L2
2.	Explain the various page table structures in detail.	Knowledge	BT L1
3.	Write short notes on non-contiguous memory management	Create	BT L6
4.	Explain the concept of demand paging in detail with neat diagram	Knowledge	BT L1
5.	Why are translation look-aside buffers important> Explain the details stored in a TLB table entry?	Knowledge	BT L1
6.	Brief the memory management system in Operating System	Create	BT L6
7.	Write short notes on physical address space in OS	Create	BT L6

Bloom’s Taxonomy



01	02	03	04	05	06
KNOWLEDGE:	UNDERSTAND:	APPLY:	ANALYZE:	EVALUATE:	CREATE:
Define,	Summarize,	Solve,	Contrast,	Criticize,	Design,
Identify,	Interpret,	Change,	Connect,	Reframe,	Modify,
Describe,	Classify,	Relate,	Relate,	Judge,	Role-Play,
Recognize,	Compare,	Complete,	Devise,	Defend,	Develop,
Tell,	Contrast,	Use,	Correlate,	Appraise,	Rewrite,
Explain,	Infer,	Sketch,	Illustrate,	Value,	Pivot,
Recite,	Relate,	Teach,	Distill,	Prioritize,	Modify,
Memorize,	Extract,	Articulate,	Conclude,	Plan,	Collaborate,
Illustrate,	Paraphrase,	Discover,	Categorize,	Grade,	Invent,
Quote	Cite	Transfer	Take Apart	Reframe	Write

UNIT – V - Operating Systems – SBSA 1301

UNIT 5

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

File System

File Concept:

Computers can store information on various storage media such as, magnetic disks, magnetic tapes, optical disks. The physical storage is converted into a logical storage unit by operating system. The logical storage unit is called FILE. A file is a collection of similar records. A record is a collection of related fields that can be treated as a unit by some application program. A field is some basic element of data. Any individual field contains a single value. A data base is collection of related data.

Student	Marks	Marks	Fail/Pas
KUMA	85	86	P
LAKSH	93	92	P

DATA FILE

Student name, Marks in sub1, sub2, Fail/Pass is fields. The collection of fields is called a RECORD.
RECORD:

LAKSH	93	92	P

Collection of these records is called a data file.

FILE ATTRIBUTES :

1. Name : A file is named for the convenience of the user and is referred by its name. A name is usually a string of characters.

2. Identifier : This unique tag, usually a number ,identifies the file within the file system.

3. Type : Files are of so many types. The type depends on the extension of the file.

Example:

.exe Executable file

.obj Object file

.src Source file

4. Location : This information is a pointer to a device and to the location of the file on that device.

5. Size : The current size of the file (in bytes, words,blocks).

6. Protection : Access control information determines who can do reading, writing, executing and so on.

7. Time, Date, User identification : This information may be kept for creation, last modification,last use.

FILE OPERATIONS

1. Creating a file : Two steps are needed to create a file. They are:

- Check whether the space is available ornot.

- If the space is available then made an entry for the new file in the directory. The entry includes name of the file, path of the file,etc…

2. Writing a file : To write a file, we have to know 2 things. One is name of the file and second is the information or data to be written on the file, the system searches the entired given location for the file. If the file is found, the system must keep a write pointer to the location in the file where the next write is to take place.

3. Reading a file : To read a file, first of all we search the directories for the file, if the file is found, the system needs to keep a read pointer to the location in the file where the next read is to take place. Once the read has taken place, the read pointer is updated.

4. Repositioning within a file : The directory is searched for the appropriate entry and the current file position pointer is repositioned to a given value. This operation is also called file seek.

5. Deleting a file : To delete a file, first of all search the directory for named file, then released the file space and erase the directoryentry.

6. Truncating a file : To truncate a file, remove the file contents only but, the attributes are as itis.

FILE TYPES:The name of the file split into 2 parts. One is name and second is Extension. The file type is depending on extension of the file.

File Type Extension Purpose

Executable .exe

.com

.bin Ready to run (or) ready

to run

machine

Source code .c

.cpp

.asm Source code in

various languages.

Object .obj

.o Compiled,

machine

Batch .bat .sh Commands to the command

Text .txt

.doc Textual data, documents

Word processor .doc

.wp

.rtf Various word processor

form ats

Library.lib

.dll Libraries of

routines for

Print or View .pdf

.jpg Binary file in a

format for

Archive .arc

.zip Related files

grouped into a

Multimedia .mpeg

.mp3

.avi Binary file containing audio

or audio/video

FILE STRUCTURE

File types also can be used to indicate the internal structure of the file. The operating system requires that an executable file have a specific structure so that it can determine where in memory to load the file and what the location of the first instruction is. If OS supports multiple file structures, the resulting size of OS is large. If the OS defines 5 different file structures, it needs to contain the code to support these file structures. All OS must support at least one structure that of an executable file so that the system is able to load and run programs.

INTERNAL FILE STRUCTURE

In UNIX OS, defines all files to be simply stream of bytes. Each byte is individually addressable by its offset from the beginning or end of the file. In this case, the logical record size is 1 byte. The file system automatically packs and unpacks bytes into physical disk blocks, say 512 bytes per block.

The logical record size, physical block size, packing determines how many logical records are in each physical block. The packing can be done by the user’s application program or OS. A file may be considered a sequence of blocks. If each block were 512 bytes, a file of 1949 bytes would be allocated 4 blocks (2048 bytes). The last 99 bytes would be wasted. It is called internal fragmentation all file systems suffer from internal fragmentation, the larger the block size, the greater the internal fragmentation.

FILE ACCESS METHODS

Files stores information, this information must be accessed and read into computer memory. There are so many ways that the information in the file can be accessed.

1. Sequential file access:

Information in the file is processed in order i.e. one record after the other. Magnetic tapes are supporting this type of file accessing.

Eg : A file consisting of 100 records, the current position of read/write head is 45th record, suppose we want to read the 75th record then, it access sequentially from 45, 46, 47

…….. 74, 75. So the read/write head traverse all the records between 45 to 75.

2. Direct access:

Direct access is also called relative access. Here records can read/write randomly without any order. The direct access method is based on a disk model of a file, because disks allow random access to any file block.

Eg : A disk containing of 256 blocks, the position of read/write head is at 95th block. The block is to be read or write is 250th block. Then we can access the 250th block directly without any restrictions.

Eg : CD consists of 10 songs, at present we are listening song 3, If we want to listen song 10, we can shift to 10.

3. Indexed Sequential File access

The main disadvantage in the sequential file is, it takes more time to access a Record

.Records are organized in sequence based on a key field. Eg :

A file consisting of 60000 records,the master index divide the total records into 6 blocks, each block consisiting of a pointer to secondary index.The secondary index divide the 10,000 records into 10 indexes.Each index consisting of a pointer to its orginal location.Each record in the index file consisting of 2 field, A key field and a pointer field.

DIRECTORY STRUCTURE

Sometimes the file system consisting of millions of files,at that situation it is very hard to manage the files. To manage these files grouped these files and load one group into one partition.

Each partition is called a directory .a directory structure provides a mechanism for organizing many files in the file system.

OPERATION ON THE DIRECTORIES :

1. Search for a file : Search a directory structure for requiredfile.

2. createafile : New files need to be created, added to thedirectory.

3. Deleteafile : When a file is no longer needed,we want to remove it from the directory.

4. List adirectory : We can know the list of files in thedirectory.

5. Renameafile : When ever we need to change the name of the file,we can change the name.

6. Traverse the file system : We need to access every directory and every file with in a directory structure we can traverse the file system

The various directory structures

1. Single level directory:

The directory system having only one directory,it consisting of all files some times it is said to be root directory.

E.g :- Here directory containing 4 files (A,B.C,D).the advantage of the scheme is its simplicity and the ability to locate files quickly.The problem is different users may accidentally use the same names for their files.

E.g :- If user 1 creates a files caled sample and then later user 2 to creates a file called sample,then user2’s file will overwrite user 1 file.Thats why it is not used in the multi user system.

2. Two level directory:

The problem in single level directory is different user may be accidentally use

the same name for their files. To avoid this problem each user need a private directory,

Names chosen by one user don't interfere with names chosen by a different user.

Root directory is the first level directory.user 1,user2,user3 are user level of directory A,B,C are files.

3. Tree structured directory:

Two level directory eliminates name conflicts among users but it is not satisfactory for users with a large number of files.To avoid this create the sub- directory and load the same type of files into the sub-directory.so, here each can have as many directories are needed.

There are 2 types of path

1. Absoulte path
2. Relative path

Absoulte path : Begging with root and follows a path down to specified files giving directory, directory name on the path.

Relative path : A path from current directory.

4. Acyclic graphdirectory

Multiple users are working on a project, the project files can be stored in a comman sub-directory of the multiple users. This type of directory is called acyclic graph directory . The common directory will be declared a shared directory. The graph contain no cycles with shared files, changes made by one user are made visible to other users. A file may now have multiple absolute paths. when shared directory/file is deleted, all pointers to the directory/ files also to be removed.

5. General graph directory:

When we add links to an existing tree structured directory, the tree structure is destroyed, resulting is a simple graph structure.

Advantages :- Traversing is easy. Easy sharing is possible.

Protection Consistency in File System

In computer systems, alot of user's information is stored, the objective of the operating system is to keep safe the data of the user from the improper access to the system. Protection can be provided in number of ways. For a single laptop system, we might provide protection by locking the computer in a desk drawer or file cabinet. For multi-user systems, different mechanisms are used for the protection.

Types of Access :

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

Read –

Reading from a file.

Write –

Writing or rewriting the file.

Execute –

Loading the file and after loading the execution process starts.

Append –

Writing the new information to the already existing file, editing must be end at the end of the existing file.

Delete –

Deleting the file which is of no use and using its space for the another data.

List –

List the name and attributes of the file.

Operations like renaming, editing the existing file, copying; these can also be controlled. There are many protection mechanism. each of them mechanism have different advantages and disadvantages and must be appropriate for the intended application.

Access Control :

There are different methods used by different users to access any file. The general way of protection is to associate identity-dependent access with all the files and directories an list called access-control list (ACL) which specify the names of the users and the types of access associate with each of the user. The main problem with the access list is their length. If we want to allow everyone to read a file, we must list all the users with the read access. This technique has two undesirable consequences:

Constructing such a list may be tedious and unrewarding task, especially if we do not know in advance the list of the users in the system.

Previously, the entry of the any directory is of the fixed size but now it changes to the variable size which results in the complicates space management. These problems can be resolved by use of a condensed version of the access list. To condense the length of the access-control list, many systems recognize three classification of users in connection with each file:

Owner –

Owner is the user who has created the file.

Group –

A group is a set of members who has similar needs and they are sharing the same file.

Universe –

In the system, all other users are under the category called universe.

The most common recent approach is to combine access-control lists with the normal general owner, group, and universe access control scheme. For example: Solaris uses the three categories of access by default but allows access-control lists to be added to specific files and directories when more fine-grained access control is desired.

Other Protection Approaches:

The access to any system is also controlled by the password. If the use of password are is random and it is changed often, this may be result in limit the effective access to a file.

The use of passwords has a few disadvantages:

The number of passwords are very large so it is difficult to remember the large passwords.

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

Consistency Semantics for file sharing

Consistency Semantics is concept which is used by users to check file systems which are supporting file sharing in their systems. Basically, it is specification to check that how in a single system multiple users are getting access to same file and at same time. They are used to check various things in files, like when will modification by some user in some file is noticeable to others.

Consistency Semantics is concept which is in a direct relation with concept named process synchronization algorithms. But process synchronization algorithms are not used in case of file I/O because of several issues like great latency, slower rate of transfer of disk and network.

Example :

When an atomic transaction to remote disk is performed by user, it involves network communications, disks read and write or both. System which is completing their task with full set of functionalities, had a poor performance. In Andrew file system, successful implementation of sharing semantics is found.

To access same file by user process is always enclosed between open() and close() operations. When there are series of access take place for same file, then it makes up a file session.

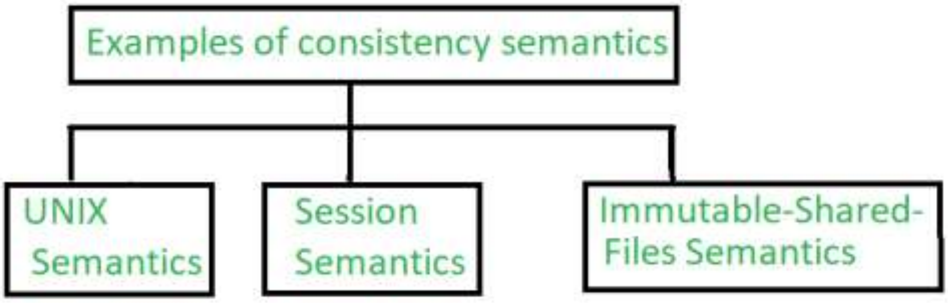

Examples of consistency semantics
UNIX Semantics
Session Semantics
Immutable-Shared-Files Semantics

Figure 46: Example of Consistency Semantics

1. UNIX Semantics :

The file systems in UNIX uses following consistency semantics –

The file which user is going to be write will be visible to all users who are sharing that file at that time.

Their is one mode in UNIX semantics to share file is via sharing pointer of current location. But it will affect all other sharing users.

In this, a file which is shared is associated with a single physical image that is accessed as an exclusive resources. This single image causes delays in user processes.

2. Session Semantics :

The file system in Andrew uses following consistency semantics.

The file which user is going to be write will not visible to all users who are sharing that file at that time.

After closing file, changes done to that file by user are only visible only in sessions starting later. If changes file is already open by other user, then changes will not be visible to that user.

In this, a file which is shared is associated with a several images and there is no delay in this because it allows multiple users to perform both read and write accesses concurrently on their images.

3. Immutable-Shared-Files Semantics :

There seems a unique approach immutable shared files. In this, user are not allowed to modify file, which is declared as shared by its creator.

An Immutable file has two properties which are as follows –

Its name may not be reused.

Its content may not be altered.

In this file system, content of file are fixed. The implementation of semantics in a distributed system is simple, because sharing is disciplined.

File system structure:

Disk provides the bulk of secondary storage on which a file system is maintained. They have 2 characteristics that make them a convenient medium for storing multiple files.

1. A disk can be rewritten in place. It is possible to read a block from the disk, modify the block, and write it back into same place.

2. A disk can access directly any block of information it contains.

I/O Control: consists of device drivers and interrupt handlers to transfer information between the main memory and the disk system. The device driver writes specific bit patterns to special locations in the I/O controller's memory to tell the controller which device location to act on and what actions to take.

The Basic File System needs only to issue commands to the appropriate device driver to read and write physical blocks on the disk. Each physical block is identified by its numeric disk address (Eg. Drive 1, cylinder 73, track2, sector 10).

The File Organization Module knows about files and their logical blocks and physical blocks. By knowing the type of file allocation used and the location of the file, file organization module can translate logical block address to physical addresses for the basic file system to transfer. Each file's logical blocks are numbered from 0 to n. so, physical blocks containing the data usually do not match the logical numbers. A translation is needed to locate each block.

The Logical File System manages all file system structure except the actual data (contents of file). It maintains file structure via file control blocks. A file control block (inode in Unix file systems) contains information about the file, ownership, permissions, location of the file contents.

File System Implementation:

Overview:

A Boot Control Block (per volume) can contain information needed by the system to boot an OS from that volume. If the disk does not contain an OS, this block can be empty.

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

A Typical File Control Block

A Directory Structure (per file system) is used to organize the files. A PER-FILE FCB contains many details about the file.

A file has been created; it can be used for I/O. First, it must be opened. The open( ) call passes a file name to the logical file system. The open( ) system call First searches the system wide open file table to see if the file is already in use by another process. If it is ,a per process open file table entry is created pointing to the existing system wide open file table. If the file is not already open, the directory structure is searched for the given file name. Once the file is found, FCB is copied into a system wide open file table in memory. This table not only stores the FCB but also tracks the number of processes that have the file open.

Next, an entry is made in the per – process open file table, with the pointer to the entry in the system wide open file table and some other fields. These are the fields include a pointer to the current location in the file (for the next read/write operation) and the access mode in which the file is open. The open () call returns a pointer to the appropriate entry in the per-process file system table. All file operations are preformed via this pointer. When a process closes the file the per- process table entry is removed. And the system wide entry open count is decremented. When all users that have opened the file close it, any updated metadata is copied back to the disk base directory structure. System wide open file table entry is removed.

System wide open file table contains a copy of the FCB of each open file, other information. Per process open file table, contains a pointer to the appropriate entry in the system wide open file table, other information.

Allocation Methods – Contiguous

An allocation method refers to how disk blocks are allocated for files:

Contiguous allocation – each file occupies set of contiguous blocks o Best performance in most cases

o Simple – only starting location (block #) and length (number ofblocks) are required

o Problems includefindingspace for file, knowing file size, external fragmentation, need for compaction off-line (downtime) or on-line

Linked

Linked allocation – each file a linked list of blocks o File ends at nil pointer

o No external fragmentation

o Each block contains pointer to next block

o No compaction, external fragmentation

o Free space management system called when new block needed o Improve efficiency by clustering blocks into groups but increases internal fragmentation

o Reliability can be a problem

o Locating a block can take many I/Os and disk seeks FAT (File Allocation Table) variation

o Beginning of volume has table, indexed by block number

o Much like a linked list, but faster on disk and cacheable

File-Allocation Table

Indexed allocation

o Each file has its own index block(s) of pointers to its data blocks

Free-Space Management

File system maintains free-space list to track available blocks/clusters Linked list (free list)

o Cannot get contiguous space easily

o No waste of space

o No need to traverse the entire list

1. Bitmap or Bit vector – A Bitmap or Bit Vector is series or collection of bits where each bit corresponds to a disk block. The bit can take two values: 0 and 1: 0 indicates that the block is allocated and 1 indicates a free block. The given instance of disk blocks on the disk in Figure 1 (where green blocks are allocated) can be represented by a bitmap of 16 bits as: 0000111000000110.

Advantages –

- Simple to understand.
- Finding the first free block is efficient. It requires scanning the words (a group of 8 bits) in a bitmap for a non-zero word. (A 0-valued word has all bits 0). The first free block is then found by scanning for the first 1 bit in the non-zero word.

Linked Free Space List on Disk

In this approach, the free disk blocks are linked together i.e. a free block contains a pointer to the next free block. The block number of the very first disk block is stored at a separate location on disk and is also cached in memory.

Grouping

Modify linked list to store address of next n-1 free blocks in first free block, plus a pointer to next block that contains free-block-pointers (like this one).

An advantage of this approach is that the addresses of a group of free disk blocks can be found easily

Counting

Because space is frequently contiguously used and freed, with contiguous- allocation allocation, extents, or clustering.

Keep address of first free block and count of following free blocks. Free space list then has entries containing addresses and counts.

Directory Implementation

1. Linear List

In this algorithm, all the files in a directory are maintained as singly lined list. Each file contains the pointers to the data blocks which are assigned to it and the next file in the directory.

Characteristics

1. When a new file is created, then the entire list is checked whether the new file name is matching to a existing file name or not. In case, it doesn't exist, the file can be created at the beginning or at the end. Therefore, searching for a unique name is a big concern because traversing the whole list takes time.

2. The list needs to be traversed in case of every operation (creation, deletion, updating, etc) on the files therefore the systems become inefficient.

2. Hash Table

To overcome the drawbacks of singly linked list implementation of directories, there is an alternative approach that is hash table. This approach suggests to use hash table along with the linked lists.

A key-value pair for each file in the directory gets generated and stored in the hash table. The key can be determined by applying the hash function on the file name while the key points to the corresponding file stored in the directory.

Now, searching becomes efficient due to the fact that now, entire list will not be searched on every operating. Only hash table entries are checked using the key and if an entry found then the corresponding file will be fetched using the value.

Efficiency and Performance

Efficiency dependent on:

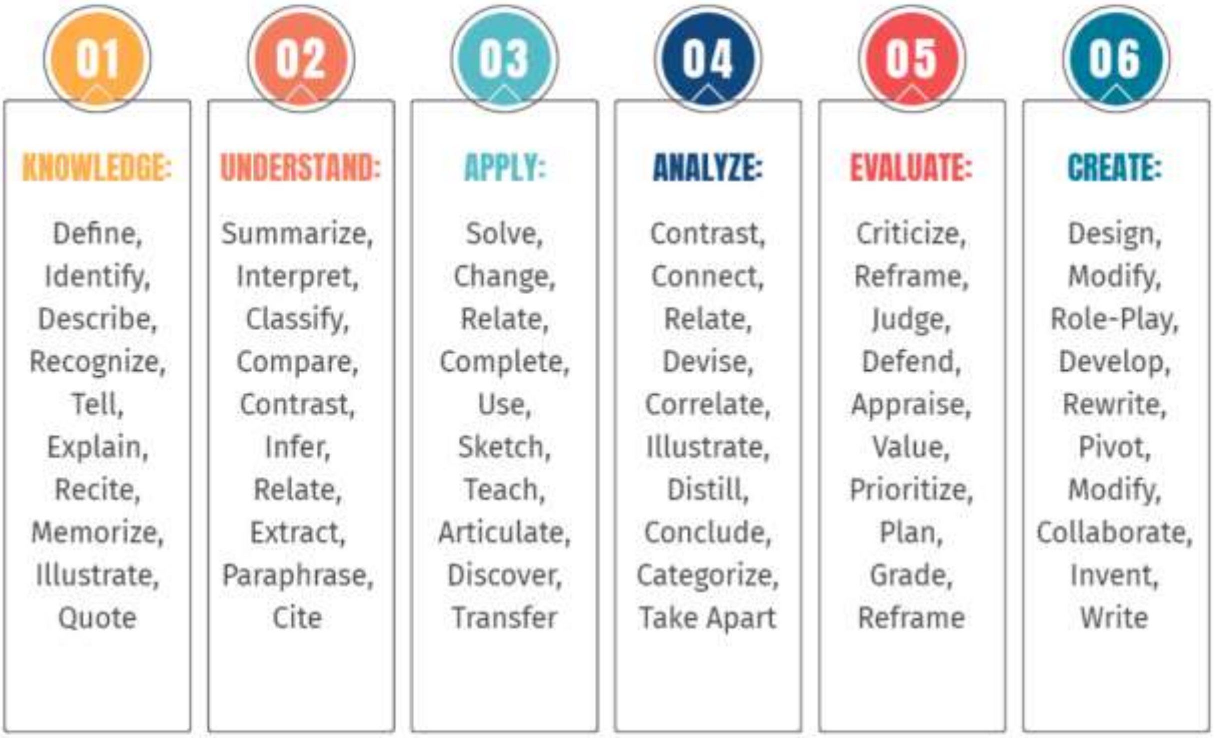
- Disk allocation and directory algorithms
- Types of data kept in file’s directory entry Performance
- Disk cache – separate section of main memory for frequently used blocks
- free-behind and read-ahead – techniques to optimize sequential access
- improve PC performance by dedicating section of memory as virtual disk, or RAM disk

Question Format :

Part-A			
Q.No	**Questions**	**Competence**	**BT Level**
11.	What is a File?	Knowledge	BT L1
12.	List the various File Attributes	Knowledge	BT L1
13.	What are the various File Operations?	Knowledge	BT L1
14.	What is the information associated with an Open File?	Knowledge	BT L1
15.	What are the different Accessing Methods of a File?	Knowledge	BT L1
16.	What are the operations that can be performed on a Directory?	Knowledge	BT L1
17.	What is a Path Name?	Knowledge	BT L1
18.	What are the Allocation Methods of a Disk Space?	Knowledge	BT L1
19.	What are the advantages of Contiguous Allocation?	Knowledge	BT L1
20.	What are the various Disk-Scheduling Algorithms?	Knowledge	BT L1

Part-B			
Q.No	**Questions**	**Competence**	**BT Level**
8.	Discuss the criteria for choosing a file organization	Understand	BT L2
9.	Explain about file attributes, file operations, and file types?	Knowledge	BT L1
10.	Explain about single-level, two-level directory structure?	Knowledge	BT L1
11.	Explain about file system mounting, file sharing?	Knowledge	BT L1
12.	Discuss the objectives for file management systems	Understand	BT L2
13.	Describe indexed file, indexed sequential file organization	Knowledge	BT L1
14.	Explain hash files organization	Knowledge	BT L1

Bloom's Taxonomy



01 KNOWLEDGE:	02 UNDERSTAND:	03 APPLY:	04 ANALYZE:	05 EVALUATE:	06 CREATE:
Define, Identify, Describe, Recognize, Tell, Explain, Recite, Memorize, Illustrate, Quote	Summarize, Interpret, Classify, Compare, Contrast, Infer, Relate, Extract, Paraphrase, Cite	Solve, Change, Relate, Complete, Use, Sketch, Teach, Articulate, Discover, Transfer	Contrast, Connect, Relate, Devise, Correlate, Illustrate, Distill, Conclude, Categorize, Take Apart	Criticize, Reframe, Judge, Defend, Appraise, Value, Prioritize, Plan, Grade, Reframe	Design, Modify, Role-Play, Develop, Rewrite, Pivot, Modify, Collaborate, Invent, Write