Unit- I Introduction

1.1 Operating System:

An **operating system** is a program that manages the computer hardware. It also provides a basis for application programs and acts as an intermediary between the computer user and the computer hardware.

Components of Computer System:

- The **hardware—**the **central processing unit (CPU),** the **memory,** and the **input/output (I/O) devices**—provides the basic computing resources for the system.
- The **application program** define the ways in which these resources are used to solve users' computing problems.
- The **operating system** controls and coordinates the use of the hardware among the various application programs for the various users.

Operating system can be explored from two view points:

A. User View B. System view

A. User View:

- Designed mostly for ease of **use.**
- some attention paid to performance.
- None paid to **resource utilization**

B. System View:

- Operating system as a **resource allocator (**CPU time, memory space, file-storage space, I/O devices, and so on.)
- **Control program (**manages the execution of user programs to prevent errors and improper use of the computer.)

1.2 Classification of Operating System:

A. Batch Systems:

- **Executing a series of no interactive similar types of jobs all at one time**. The term originated in the days when users entered programs on punch cards. They would give a batch of these programmed cards to the system operator, who would feed them into the computer.
- **Batch jobs can be stored up during working hours and then executed during the evening or whenever the computer is idle**. Batch processing is particularly useful for operations that require the computer or a peripheral device for an extended period of time. Once a batch job begins, it continues until it is done or until an error occurs. Note that batch processing implies that there is no interaction with the user while the program is being executed.

B. <u>Multi Programmed System:</u>

- In Multiprogramming, several **programs are run at the same time on a uniprocessor in an interleaved manner**. Since there is only one processor , **there can be no true simultaneous execution of different programs**. To the user it appears that all programs are executing at the same time.
- **If the machine has the capability of causing an interrupt after a specified time interval**, then the operating system will **execute each program for a given length of time**, regain control, and then execute another program for a given length of time, and so on. **In the absence of this mechanism**, the **operating system has no choice but to begin to execute a program with the expectation**, but not the certainty, that the program will eventually return control to the operating system.

Memory layout for a multiprogramming system.

C. **Time Sharing System:**

Refers to the concurrent use of a computer by more than one user -- users *share* the computer's time. *Time sharing* is synonymous with *multi-user*. It is also called interactive system.

A **time-sharing system** is one that allows multiple users to share time on a single computer.

- Each user is given a **time slice** of CPU time (e.g., each user is served every 0.1 s by the computer).
- The computer works so fast that each user seems to be the sole user of the computer.
- One example of a time-sharing system is the bank's bankcard system, which allows hundreds of people to access the same program on the mainframe at the same time.
- A time-sharing system that allows different users to independently run different programs at the same time is also called a multi-user system.
-

D. Multitasking Operating System:

- Multitasking operating system is a time sharing system that also supports multiple process per user.
- In multitasking, only one CPU is involved, but **it switches from one program to another so quickly that it gives the appearance of executing all of the programs at the same tim**e.

E. Multi Processor System (Parallel System or Tightly Coupled Systems):

- Such systems have two or more processors in close communication within the same computer system, sharing the computer bus and sometimes the clock, memory, and peripheral devices.

  Advantages:

  i. Increased throughput: By increasing the number of processors, we expect to get more work done in less time.

  ii. Economy of scale: Save more money than multiple single processor system.

iii. Increased reliability: The failure of one processor will not halt the system, only slow it down.

- The multiple-processor systems in use today are of two types.

  i. symmetric multiprocessing (SMP): each processor runs an identical copy of the O.S. and these copies communicate with one another as needed (no master slave relationship exists between processors).

  ii. Asymmetric multiprocessing (ASMP): each processor is assigned a specific task. A master processor controls the system; the other processors either look to the master for instruction or have predefined tasks.

F. Distributed System:

- In a distributed system, software and data maybe distributed around the system, programs and files maybe stored on different storage devices which are located in different geographical locations and maybe accessed from different computer terminals. These systems have their own local memory. They do not have shared memory.
- **Client Server System:**
  - i. Centralized systems act **as server systems** to satisfy request generated by client system.
  - ii. Server systems may be **computer server systems** (only request) or **file server systems** (file read, write, create etc.)

Client Server system

- **Peer to Peer System:**
  - i. Processor communicates with one another through various communication lines such as high speed buses or telephone lines.
  - ii. Peer to peer system may be tightly coupled (do not share memory or clock, have local memory to communicate) or loosely coupled (sharable).

G. Clustered system:

- Like multiprocessor systems, clustered systems gather together multiple CPUs to accomplish computational work. Clustered systems differ from multiprocessor systems; in multiprocessor system, two or more CPUs are tied together within a single system through bus, sharing memory, I/O and clock, while in distributed system two or more individual system are tied together sharing the memory space through LAN.
- Clustered computers share storage and are closely link via LAN networking.
- Clustering is usually used to provide **high-availability.** A layer of cluster software runs on the cluster nodes. Each node can monitor one or more of the others (over the LAN). If the monitored machine fails, the monitoring machine can take ownership of its storage and restart the applications that were running on the failed machine. The users and clients of the applications see only a brief interruption of service.
- **Distributed Lock manager (DLM)** provides access control and locking to the files to ensure no conflicting operations occurs.
- Clustering can be structured asymmetrically or symmetrically.

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

  ii. **In symmetric mode,** two or more hosts are running applications, and are monitoring each other.

H. Real Time System:

A system is said to be real-time if the total correctness of an operation depends not only upon its logical correctness, but also upon the time in which it is performed.

OR

A real-time system is a software system where the correct functioning of the system depends on the results produced by the system and the time at which these results are produced.

A real-time operating system (RTOS) is an operating system (OS) intended to serve real-time application requests

Types of Real Time Systems

1. **hard real-time system**

   A hard real-time system is a system whose operation is incorrect if results are not produced according to the timing specification.

   Hard real-time systems are used when it is imperative that an event is reacted to within a strict deadline. Such strong guarantees are required of systems for which not reacting in a certain interval of time would cause great loss in some manner, especially damaging the surroundings physically or threatening human lives (although the strict definition is simply that missing the deadline constitutes failure of the system). For example, a car engine control system is a hard real-time system because a delayed signal may cause engine failure or damage. . Hard real-time systems are typically found interacting at a low level with physical hardware. In these systems secondary storage are either limited or absent.OS is stored on ROM (Read Only Memory).

- **soft real-time system**

A soft real time system is a system whose operation is degraded if results are not produced according to the specified timing requirements

- Late completion of jobs is undesirable but not fatal.
- System performance degrades as more & more jobs miss deadlines.
- Live audio-video systems are usually soft real-time; violation of constraints results in degraded quality, but the system can continue to operate

Examples of RTOS: QNX, RTLINUX and VxWorks

I. **Hand Held System**:

- It includes PDA (Personal Digital Assistants) such as Palm-Pilots or cellular telephones with connectivity to a network such as internet.
- The developers of hand held systems have some limitations like **less memory, less processor, small screen display and limited power backup.**

J. Embedded system: an embedded system is one in which OS is embeds on the system itself.

- They are very fast.
- They are application specific.
- They are used in washing machine, cellphones,control systems.

K. Network operating system

A networking operating system (NOS), also referred to as the Dialoguer, is the software that runs on a server and enables the server to manage data, users, groups, security, applications, and other networking functions. The network operating system is designed to allow shared file and printer access among multiple computers in a network, typically a local area network (LAN), a private network or to other networks. The most popular network operating systems are Microsoft Windows Server 2003, Microsoft Windows Server 2008, UNIX, Linux, Mac OS X, and Novell NetWare.

Network Operating Systems are based on a client/server architecture in which a server enables multiple clients to share resources.

The Network Operating System can also do the following:

Centrally manage network resources, such as programs, data and devices.

Secure access to a network.

Allow remote users to connect to a network.

Allow users to connect to other networks like the Internet.

Back up data and make sure it's always available.

Allow for simple additions of clients and resources.

Monitor the status and functionality of network elements.

Distribute programs and software updates to clients.

Ensure efficient use of a server's capabilities**.**

1.3 Functions of Operating System:

- **Management of the processor**: the operating system is responsible for managing allocation of the processor between the different programmes using a **scheduling algorithm**. The type of scheduler is totally dependent on the operating system, according to the desired objective.
- **Management of the random access memory**: the operating system is responsible for managing the memory space allocated to each application and, where relevant, to each user. If there is insufficient physical memory, the operating system can create a memory zone on the hard drive, known as "**virtual memory**". The virtual memory lets you run applications requiring more memory than there is available RAM on the system. However, this memory is a great deal slower.
- **Management of input/output**: the operating system allows unification and control of access of programmes to material resources via drivers (also known as peripheral administrators or input/output administrators).
- **Management of execution of applications**: the operating system is responsible for smooth execution of applications by allocating the resources required for them to operate. This means an application that is not responding correctly can be "killed".

- **Management of authorizations**: the operating system is responsible for security relating to execution of programmes by guaranteeing that the resources are used only by programmes and users with the relevant authorizations.
- **File management**: the operating system manages reading and writing in the file system and the user and application file access authorizations.
- **Information management**: the operating system provides a certain number of indicators that can be used to diagnose the correct operation of the machine.

1.4 Operating System Structure:

A. Simple Structure :

In this, the most functionalities are provided in least space so it was not divided into modules carefully.

Example: MS-DOS System Structure

- MS-DOS – written to provide the most functionality in the least space
  - Not divided into modules

– Although MS-DOS has some structure, its interfaces and levels of functionality are not well separate.

–

B. Layered Structure:

- The operating system is divided into a number of layers (levels), each built on top of lower layers. The **bottom layer (layer 0), is the hardware**; **the highest (layer N) is the user interface**.
- An OS layer is an implementation of an abstract object that is the encapsulation of data and operations that can manipulate those data. These operations (routines) can be invoked by higher-level layers. The layer itself can invoke operations on lower-level layers.
- Layered approach provides modularity. With modularity, layers are selected such that each layer uses functions (operations) and services of only lower-level layers.
- Each layer is implemented by using only those operations that are provided lower level layers.
- The major difficulty is appropriate definition of various layers.

▪

Layered Operating System.

Example: UNIX System Structure

UNIX – limited by hardware functionality, the original UNIX operating system had limited structuring. The UNIX OS consists of two separable parts.

- Systems programs – use kernel supported system calls to provide useful functions such as compilation and file manipulation.
- The kernel
  - Consists of everything below the system-call interface and above the physical hardware
  - Provides the file system, CPU scheduling, memory management, and other operating-system functions; a large number of functions for one level.

Kernel:

Kernel is the core part of the operating system. This kernel is in the inner layer of OS layers and directly interacts with the hardware. It is responsible for performing various functions related to the hardware such as main memory management, process management, I/O management and secondary storage management. When a computer boots up, it goes through some initialization functions, such as checking memory. It then loads the kernel and switches control to it. The kernel then starts up all the processes needed to communicate with the user and the rest of the environment (e.g. the LAN)

The kernel is always loaded into memory, and kernel functions always run, handling processes, memory, and devices.

The traditional structure of a kernel is a *layered* system, such as Unix. In this, all layers are part of the kernel, and each layer can talk to only a few other layers. Application programs and utilities live above the kernel.

Types of Kernel:

1. **Monolithic kernel:** In a monolithic kernel, all OS services such as process management, memory management, file management, storage management & I/O management run along with the main kernel thread, thus also residing in the same memory area.

   Advantage:

   1. Easier to design
   2. More efficient due to the use of shared kernel memory.

Disadvantage:

1. Large kernel size
2. Lack of extensibility
3. Unreliable: as a bug anywhere in the kernel can bring down the whole system.

2. **Micro kernel structure**

- Kernel using **microkernel approach**.
- Remove all **non-essential components from the kernel** & implementing them as system & user level programs (smaller kernels).
- Main function of microkernel is to provide a communication facility **between a client program & the various services** that are also running in user space.
- Benefits:
  - Easier to extend a microkernel
  - Easier to port the operating system to new architectures
  - More reliable (less code is running in kernel mode)

- More secure

- Disadvantages:
  - Performance (System calls can require a lot of protection mode changes)
  - Expensive to re implement everything with a new model

1.5 Operating System Components:

- Due to the complex nature of the modern operating systems, it is partitioned into smaller component. Each component performs a well-defined function with well-defined inputs and outputs.
- Many modern operating systems have the following components.
  - Process Management
  - Main Memory Management
  - File Management
  - I/O System Management
  - Secondary Management
  - Networking
  - Protection System
  - Command-Interpreter System

A. Process Management

- A process is 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.
- Remember a program itself is not a process rather it is a passive entity.

- A process needs certain resources, including CPU time, memory, files, and I/O devices, to accomplish its task. These resources are either given to the process when it is created or when it is running. When the process completes, the OS reclaims all the resources.
- The operating system is responsible for the following activities in connection with process management.
  - Process creation and deletion.
  - Process suspension and resumption.
  - Provision of mechanisms for:
    - Process synchronization
    - Process communication

B. Main Memory Management

- Memory is a large array of words or bytes, each with its own address. It is a repository of quickly accessible data shared by the CPU and I/O devices.
- Main memory is a volatile storage device. It loses its contents in the case of system failure.
- The operating system is responsible for the following activities in connections with memory management:
  - Keep track of which parts of memory are currently being used and by whom.
  - Decide which processes to load when memory space becomes available.
  - Allocate and deallocate memory space as needed.

C. File Management:

- Most visible component of OS. Computers can store information on several different types of physical media (e.g. magnetic tape, magnetic disk, CD etc).
- For convenient use of the computer system, the OS provides a uniform logical view of information storage.

- A file a logical storage unit, which abstract 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 connections with file management:
  - File creation and deletion.
  - Directory creation and deletion.
  - Support of primitives for manipulating files and directories.
  - Mapping files onto secondary storage.
  - File backup on stable (nonvolatile) storage media.

D. I/O System Management

Used to manage I/O devices and I/O operations.

- The I/O system consists of:
  - A buffer-caching system
  - A general device-driver interface
  - Drivers for specific hardware devices

E. Secondary Storage management

- 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:
  - Free space management
  - Storage allocation
  - Disk scheduling

F. Networking (Distributed Systems)

- A *distributed* system is a collection processors that do not share memory or a clock. Each processor has its own local memory.
- The processors in the system are connected through a communication network.
- Communication takes place using a *protocol.*
- A distributed system provides user access to various system resources.
- Access to a shared resource allows:
  - Computation speed-up
  - Increased data availability
  - Enhanced reliability

G. Protection System

- *Protection* refers to a mechanism for controlling access by programs, processes, or users to both system and user resources.
- The protection mechanism must:
  - Distinguish between authorized and unauthorized usage.
  - Specify the controls to be imposed.
  - Provide a means of enforcement.

H. Command-Interpreter System

- Many commands are given to the operating system by control statements which deal with:
  - Process creation and management
  - I/O handling
  - Secondary-storage management
  - Main-memory management
  - File-system access
  - Protection
  - Networking

- The program that reads and interprets control statements is called variously:
  - Command-line interpreter
  - Shell (in UNIX)
- Its function is to get and execute the next command statement.

1.6 Operating System Services:

- **Program execution** – system capability to load a program into memory and to run it.
- **I/O operations** – since user programs cannot execute I/O operations directly, the operating system must provide some means to perform I/O.
- **File-system manipulation** – program capability to read, write, create, and delete files.
- **Communications** – exchange of information between processes executing either on the same computer or on different systems tied together by a network. Implemented via *shared memory* or *message passing*.
- **Error detection** – ensure correct computing by detecting errors in the CPU and memory hardware, in I/O devices, or in user programs.

➢ **Additional functions** exist not for helping the user, but rather for ensuring efficient system operations.

- **Resource allocation** – allocating resources to multiple users or multiple jobs running at the same time.
- **Accounting** – keep track of and record which users use how much and what kinds of computer resources for account billing or for accumulating usage statistics.
- **Protection** – ensuring that all access to system resources is controlled

1.7 System Calls

- System calls provide the interface between a process and the operating system. These calls are generally available as assembly language instructions. Some systems also allow to make system calls from a high level language, such as C.They are used to allow user level processes to request services of the OS.User programs communicate with an OS &

requests services from it by making system calls.each system call has a library procedure that the user program calls.

Types of System Calls :

The various types of system calls are as given :

- Process control
  - end, abort
  - load, execute
  - create process, terminate process
  - get process attributes, set process attribute
  - wait for time
  - wait event, signal event
  - allocate and free memory
- File management
  - create file, delete file
  - open, close
  - read, write, reposition
  - get file attributes, set file attributes
- Device management
  - request device, release device
  - read, write, reposition
  - get device attributes, set device attributes
  - logically attach or detach devices
- Information maintenance
  - get time or date, set time or date
  - get system data, set system data
  - get process, file, or device attributes
  - set process, file, or device attributes
- Communications
  - create, delete communication connection
  - send, receive messages

- Communications
  - create, delete communication connection
  - send, receive messages
  - transfer status information
  - attach or detach remote devices

1.8 System Programs:

- System programs provide a convenient environment for program development and execution. Some of them are simply user interfaces to system calls.
- ***Types of system programs :***

  - **File management.** These programs create, delete, copy, rename, print, dump, list, and generally manipulate files and directories.
  - **Status information.** Some programs simply ask the system for the date, time, amount of available memory or disk space, number of users, or similar status information. Others are more complex, providing detailed performance, logging, and debugging information. Typically, these programs format and print the output to the terminal or other output devices or files or display it in a window of the GUI. Some systems also support a registry, which is used to store and retrieve configuration information.
  - **File modification.** Several text editors may be available to create and modify the content of files stored on disk or other storage devices. There may also be special commands to search contents of files or perform transformations of the text.
  - **Programming-language support.** Compilers, assemblers, debuggers and interpreters for common programming languages (such as C, C++, Java, Visual Basic, and PERL) are often provided to the user with the operating system.
  - **Program loading and execution.** Once a program is assembled or compiled, it must be loaded into memory to be executed. The system may provide absolute loaders, relocatable loaders, linkage editors, and overlay loaders. Debugging systems for either higher-level languages or machine language are needed as well.
  - **Communications.** These programs provide the mechanism for creating virtual connections among processes, users, and computer systems. They allow users to send messages to one another's screens, to browse web pages, to send electronic-mail messages, to log in remotely, or to transfer files from one machine to another.

Q 1:

What is the distinction between kernel mode and user mode in operating system.

Answer:. Certain instructions could be executed only when the CPU is in kernel mode. Similarly, hardware devices could be accessed only when the program is executing in kernel mode. Control over when interrupts could be enabled or disabled is also possible only when the CPU is in kernel mode. Consequently, the CPU has very limited capability when executing in user mode,

What is the purpose of the command interpreter? Why is it usually separate from the kernel?

Answer: It reads commands from the user or from a file of commands and executes them, usually by turning them into one or more system calls. It is usually not part of the kernel since the command interpreter is subject to changes.

What is the purpose of system programs?

Answer: System programs can be thought of as bundles of useful system calls. They provide basic functionality to users so that users do not need to write their own programs to solve common problems.

What is the main advantage of the layered approach to system design? What are the disadvantages of using the layered approach?

Answer: As in all cases of modular design, designing an operating system in a modular way has several advantages. The system is easier to debug and modify because changes affect only limited sections of the system rather than touching all sections of the operating system. Information is kept only where it is needed and is accessible only within a defined and restricted area, so any bugs affecting that data must be limited to a specific module or layer.

UNIT-2 PART -1

Types of processes :

- **Independent Processes** operating concurrently on a systems are those that can neither affect other processes or be affected by other processes.
- **Cooperating Processes** are those that can affect or be affected by other processes. There are several reasons why cooperating processes are allowed:
    - **Information Sharing** - There may be several processes which need access to the same file for example. (e.g. pipelines.)
    - **Computation speedup** - Often a solution to a problem can be solved faster if the problem can be broken down into sub-tasks to be solved simultaneously (particularly when multiple processors are involved.)
    - **Modularity** - The most efficient architecture may be to break a system down into cooperating modules. (E.g. databases with a client-server architecture.)
    - **Convenience** - Even a single user may be multi-tasking, such as editing, compiling, printing, and running the same code in different windows.

Producer-Consumer Problem

- Producer-consumer problem is a common paradigm for cooperating processes

- Producer process produces information that is consumed by a consumer process .

- One solution is to use shared memory for the two processes to communicate - Useful to have a buffer that can be filled by the producer and emptied by the consumer if they are to run concurrently

- Unbounded-buffer places no practical limit on the size of the buffer

- Bounded-buffer assumes that there is a fixed buffer size

Interprocess communication (IPC):

Operating system provides the means for cooperating process to communicate with each other through an interprocess communication.

IPC provides a mechanism to allow processes to communicate and to synchronize their without sharing the same address space. IPC is best provided by a message passing system.

Message-Passing Systems : The function of a message system is to allow processes to communicate with one another without the need to resort to shared data.

- An IPC facility provides at least the two operation : "send (message)" and "receive(message)".
- Message sent by a process can be of fixed or variable size.

- A communication link must be established between the cooperating processes before messages can be sent.
- There are three key issues to be resolved in message passing systems as further explored in the next three subsections:
  - **Direct or indirect communication**
  - **Synchronous or asynchronous communication**
  - **Automatic or explicit buffering.**

Direct communication the sender must know the name of the receiver to which it wishes to send a message.

- There is a one-to-one link between every sender-receiver pair.
- For **symmetric** communication, the receiver must also know the specific name of the sender from which it wishes to receive messages.
- The send and receive primitives are defined as

  - **Send(P,Message)-** send a message to process P
  - **Receive(Q,Message)-**receive a message from process Q

For **asymmetric** communications, this is not necessary.

- In asymmetric communication, only the sender names the recipient, the recipient is not required to name the sender.
- The send and receive primitives are defined as

  - **Send(P,Message)-** send a message to process P

- **Receive(id,Message)-**receive a message from any process. The variable id is set to the name of the process with which communication has taken place.

- **Indirect communication** uses shared mailboxes, or ports.
  - Multiple processes can share the same mailbox or boxes.
  - Only one process can read any given message in a mailbox. Initially the process that creates the mailbox is the owner, and is the only one allowed to read mail in the mailbox, although this privilege may be transferred.
  - **Properties of indirect communication link**
  - - Link established only if processes share a common mailbox – A
  - link may be associated with more than two processes –
  - Each pair of processes may share several communication links
  - - Link may be unidirectional or bi-directional
  - Primitives are defined as:
  - **send(A , message)** – send a message to mailbox A –
  - **receive(A , message)** – receive a message from mailbox A
  - The OS must provide system calls to create and delete mailboxes, and to send and receive messages to/from mailboxes.

Synchronization

Message passing may be either blocking or non-blocking

• Blocking is considered synchronous

Blocking send has the sender block until the message is received –

Blocking receive has the receiver block until a message is available •

Non-blocking is considered asynchronous –

Non-blocking send has the sender send the message and continue –

Non-blocking receive has the receiver receive a valid message or null

Buffering

- Messages are passed via queues, which may have one of three capacity configurations:

1. **Zero capacity** - Messages cannot be stored in the queue, so senders must block until receivers accept the messages.
2. **Bounded capacity**- There is a certain pre-determined finite capacity in the queue. Senders must block if the queue is full, until space becomes available in the queue, but may be either blocking or non-blocking otherwise.
3. **Unbounded capacity** - The queue has a theoretical infinite capacity, so senders are never forced to block.

Unit -2 part 2

Threads & Their Management

Thread : Thread is an execution unit which consists of its own program counter, a stack, and a set of registers. Threads are also known as Lightweight processes. Threads are popular way to improve application through parallelism. The CPU switches rapidly back and forth among the threads giving illusion that the threads are running in parallel.

As each thread has its own independent resource for process execution, multpile processes can be executed parallely by increasing number of threads.

single-threaded process

multithreaded process

Processes Vs Threads

As we mentioned earlier that in many respect threads operate in the same way as that of processes. Some of the similarities and differences are:

Similarities

- Like processes threads share CPU and only one thread active (running) at a time.
- Like processes, threads within processes, threads within a processes execute sequentially.
- Like processes, thread can create children.
- And like process, if one thread is blocked, another thread can run.

Differences

- Unlike processes, threads are not independent of one another.
- Unlike processes, all threads can access every address in the task .
- Unlike processes, thread are design to assist one other. Note that processes might or might not assist one another because processes may originate from different users.

Types of Thread

There are two types of threads :

- **User Threads**
- **Kernel Threads**

User-Level Threads

User-level threads implement in user-level libraries, rather than via systems calls, so thread switching does not need to call operating system and to cause interrupt to the kernel. In fact, the kernel knows nothing about user-level threads and manages them as if they were single-threaded processes.

Advantages:

The most obvious advantage of this technique is that a user-level threads package can be implemented on an Operating System that does not support threads. Some other advantages are

- A user-level thread does not require modification to operating systems.
- **Simple Representation:**
  Each thread is represented simply by a PC, registers, stack and a small control block, all stored in the user process address space.
- **Simple Management:**
  This simply means that creating a thread, switching between threads and synchronization between threads can all be done without intervention of the kernel.
- **Fast and Efficient:**
  Thread switching is not much more expensive than a procedure call.

Disadvantages:

- There is a lack of coordination between threads and operating system kernel. Therefore, process as whole gets one time slice irrespective of whether process has one thread or 1000 threads within. It is up to each thread to relinquish control to other threads.

- User-level threads requires non-blocking systems call i.e., a multithreaded kernel. Otherwise, entire process will blocked in the kernel, even if there are runable threads left in the processes. For example, if one thread causes a page fault, the process blocks.

Kernel-Level Threads

In this method, the kernel knows about and manages the threads. No runtime system is needed in this case. Instead of thread table in each process, the kernel has a thread table that keeps track of all threads in the system. In addition, the kernel also maintains the traditional process table to keep track of processes. Operating Systems kernel provides system call to create and manage threads.

Advantages:

- Because kernel has full knowledge of all threads, Scheduler may decide to give more time to a process having large number of threads than process having small number of threads.
- Kernel-level threads are especially good for applications that frequently block.

Disadvantages:

- The kernel-level threads are slow and inefficient. For instance, threads operations are hundreds of times slower than that of user-level threads.
- Since kernel must manage and schedule threads as well as processes. It require a full thread control block (TCB) for each thread to maintain information about threads. As a result there is significant overhead and increased in kernel complexity.

Multithreading Models

The user threads must be mapped to kernel threads, by one of the following strategies.

- Many-To-One Model
- One-To-One Model
- Many-To-Many Model

Many-To-One Model

- In the many-to-one model, many user-level threads are all mapped onto a single kernel thread.
- Thread management is handled by the thread library in user space, which is efficient in nature.

One-To-One Model

- The one-to-one model creates a separate kernel thread to handle each and every user thread.
- Most implementations of this model place a limit on how many threads can be created.
- Linux and Windows from 95 to XP implement the one-to-one model for threads.

Many-To-Many Model

- The many-to-many model multiplexes any number of user threads onto an equal or smaller number of kernel threads, combining the best features of the one-to-one and many-to-one models.
- Users can create any number of the threads.
- Blocking the kernel system calls does not block the entire process.
- Processes can be split across multiple processors.

What are the benefits and detriments of each of the following? Consider both the systems and the programmers' levels.

a. Symmetric and asymmetric communication

b. Automatic and explicit buffering

c. Send by copy and send by reference

d. Fixed-sized and variable-sized messages

Answer:

a. **Symmetric and asymmetric communication** - A benefit of symmetric communication is that it allows a rendezvous between the sender and receiver. A disadvantage of a blocking send is that a rendezvous may not be required and the message could be delivered asynchronously; received at a point of no interest to the sender. As a result, message-passing systems often provide both

forms of synchronization.

b. **Automatic and explicit buffering** -Automatic buffering provides a queuewith indefinite length; thus ensuring the sender will never have to block while waiting to copy a message. There are no specifications how automatic buffering will be provided; one scheme may reserve sufficiently large memory where much of the memory is wasted. Explicit buffering specifies how large the buffer is. In this situation, the sender may be blocked while waiting for available space in the queue. However, it is less likely memory will be wasted with explicit buffering.

c. **Send by copy and send by reference** - Send by copy does not allow the receiver to alter the state of the parameter; send by reference does allow it.Abenefit of send by reference is that it allows the programmer towrite a distributed version of a centralized application.

d. **Fixed-sized and variable-sized messages** - The implications of this are mostly related to buffering issues; with fixed-size messages, a buffer with a specific size can hold a known number of messages. The number of variable-sized messages that can be held by such a buffer is unknown. Consider how Windows 2000 handles this situation: with fixed-sizedmessages (anything < 256 bytes), the messages are copied from the address space of the sender to the address space of the receiving process. Larger messages (i.e. variable-sized messages) use shared memory to pass the message.

Process Synchronization

Producer Consumer Problem

Producer Process : a producer process produces information which is stored in a buffer . the buffer has a fixed buffer size.

Algorithm for producer process

```
while (true) {
        /* produce an item in next produced */


        while (counter == BUFFER_SIZE) ;

                /* do nothing */

        buffer[in] = next_produced;

        in = (in + 1) % BUFFER_SIZE;

        counter++; }
```

Consumer Process : a Consumer process consumes information which is produced by producer and which is stored in a buffer .

Algorithm for Consumer process

```
while (true) {

        while (counter == 0)

                ; /* do nothing */

        next_consumed = buffer[out];

        out = (out + 1) % BUFFER_SIZE;      counter--;

        /* consume the item in next consumed */

}
```

Race Condition :

Race condition is a condition where several processes access and manipulate the same data items concurrently and the outcome of the execution depends on the order in which the access take place .

In order to avoid such a race condition, we need to ensure that only one process at a time is allowed to access the shared variable. For that purpose, we need to synchronize the processes called process synchronization.

Critical section (CS) problem:

Each process has a segment of code called '**Critical Section**', in which the process may be changing common variables, updating a table, writing a file and so on.

- ***n*** processes competing to use some shared data.
- Problem – ensure that when one process is executing in its CS, no other process is allowed to execute in its CS.
- When a process executes code that manipulates shared data (or resource), we say that the process is in it's Critical Section (for that shared data).
- The execution of critical sections must be mutually exclusive: at any time, only one process is allowed to execute in its critical section (even with multiple processors).
- So each process must first request permission to enter its critical section.
- The section of code implementing this request is called the Entry Section (ES).
- The critical section (CS) might be followed by a Leave/Exit Section (LS).
- The remaining code is the Remainder Section (RS).
- The critical section problem is to design a protocol that the processes can use so that their action will not depend on the order in which their execution is interleaved (possibly on many processors).

do {

entry section

critical section

exit section

remainder section

} **while (TRUE);**

General structure of process P_i

- Processes may share some common variables to synchronize their actions.

Solution to Critical-Section Problem

- There are 3 requirements that must stand for a correct solution:We can check on all three requirements in each proposed solution, even though the non-existence of each one of them is enough for an incorrect solution.
- **Mutual Exclusion** – If process P_i is executing in its critical section, then no other processes can be executing in their critical sections.
- **Progress –** If no process is executing in its critical section and there exist some processes that wish to enter their critical section, then the selection of the process that will enter the critical section next cannot be postponed indefinitely:
- **Bounded Waiting** – A bound must exist 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.

1. Two Process Solution :

(a) Algorithm 1:

int turn; initially turn = 0

turn = i $\Rightarrow$ P_i can enter its critical section

Process P_i

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

Satisfies mutual exclusion and bounded waiting, but not progress

(b) Algorithm 2 :

Process P_i

```
do {
        flag[i] = TRUE;
        while (flag[j]);
        critical section
        flag [i] = FALSE;
        remainder section
} while (TRUE);
```

Mutual Exclusion is satisfied but Progress requirement is not satisfied

(c) Peterson algorithm

Process P_i

```
do {
        flag [i] = TRUE;
turn = j;
while (flag [j] && turn == j);
                critical section
        flag [i] = FALSE;
                remainder section
```

} while (TRUE);

To prove that the solution is correct, we must examine the three conditions listed above:

1. **Mutual exclusion -** If one process is executing their critical section when the other wishes to do so, the second process will become blocked by the flag of the first process. If both processes attempt to enter at the same time, the last process to execute "turn = j" will be blocked.
2. **Progress -** Each process can only be blocked at the while if the other process wants to use the critical section (flag[j] = = true), AND it is the other process's turn to use the critical section (turn = = j). If both of those conditions are true, then the other process (j) will be allowed to enter the critical section, and upon exiting the critical section, will set flag[j] to false, releasing process i. The shared variable turn assures that only one process at a time can be blocked, and the flag variable allows one process to release the other when exiting their critical section.
3. **Bounded Waiting -** As each process enters their entry section, they set the turn variable to be the other processes turn. Since no process ever sets it back to their own turn, this ensures that each process will have to let the other process go first at most one time before it becomes their turn again.

- Meets all three requirements; solves the critical-section problem for two processes.

(d) Dekker's algorithm

- Process P_i

```
do {
        flag[i] = TRUE;
while (flag[j]) {
If (turn==j)
{ flag[i]= false;
While (turn==j);
Flag[i]=true;} }
                critical section
Turn=j;
        flag [i] = FALSE;
                remainder section
} while (TRUE);
```

(2) **n Process solution :**

Bakery algorithm

– Before entering its critical section, a process receives a number (like in a bakery). Holder of the smallest number enters the critical section.

- The numbering scheme here always generates numbers in increasing order of enumeration; i.e., 1,2,3,3,3,3,4,5...
- If processes P_i and P_j receive the same number, if $i < j$, then P_i is served first; else P_j is served first (PID assumed unique).

```
do {
        choosing[i] = TRUE;
        number[i] = max(number[0], …, number[n – 1]) +1;
        choosing[i] = FALSE;
        for (j = 0; j < n; j++) {
                while (choosing[j]) ;
                while ((number[j] != 0) &&
          ((number[j],j) < (number[i],i))) ;
        }
                critical section
        number[i] = 0;
                remainder section
} while (TRUE);
```

Semaphores:

- A semaphore S is an integer variable that, apart from initialization, can only be accessed through 2 atomic exclusive operations:
  - wait(S) P(S))
  - signal(S) V(S)
- Less complicated.
- Access is via two atomic operations:
- ***wait* (*S*):**
- while (*S* <= 0);
  S--;
- ***signal* (*S*):**
- *S*++;

Semaphores used for n process critical section problem

```
Shared data:

  semaphore mutex; // initiallized to 1

Process Pi:

do {
   wait(mutex);
      critical section

   signal(mutex);
   remainder section
} while (TRUE);
```

Busy Waiting :

When a process is in its critical section, any other process that tries to enter its critical section must loop continuously in the entry code. This condition is called **busy waiting**.

- Busy waiting wastes CPU cycles.
- To overcome busy waiting problem, we modify the definitions of Semaphores 'Wait' and 'Signal'
- When a process must wait for a semaphore S, it is blocked and put on the semaphore's queue.
- Signal operation removes (assume a fair policy like FIFO) first process from the queue and puts it on list of ready processes.
- **Block:** place the process invoking the operation on the appropriate waiting queue.
- **Wakeup:** remove one of processes in the waiting queue and place it in the ready queue.

- **Modified definition of Wait and Signal for busy waiting**

```
wait (semaphore *S) {
        S->value--;
        if (S->value < 0) {
                add this process to S->list;
                block();
        }
}
```

```
signal (semaphore *S) {
        S->value++;
        if (S->value <= 0) {
                remove a process P from S->list;
                wakeup(P);
        }
}
```

Types of Semapores :

1. ***Counting* semaphore** – integer value can range over an unrestricted domain.

Counting semaphores can take on any integer value, and are usually used to count the number remaining of some limited resource. The counter is initialized to the number of such resources available in the system, and whenever the counting semaphore is greater than zero, then a process can enter a critical section and use one of the resources. When the counter gets to zero (or negative in some implementations), then the process blocks until another process frees up a resource and increments the counting semaphore with a signal call. (The binary semaphore can be seen as just a special case where the number of resources initially available is just one.)

2. ***Binary* semaphore –** integer value can range only between 0 and 1 (really a Boolean); can be simpler to implement (use waitB and signalB operations); same as Mutex lock.

Wait B(S):

```
 if (S.count = 1) {
   S.count := 0;
 } else {
  block this process
  place this process in S.queue
 }
```

Signal B(S):

```
 if (S.queue is empty) {
  S.count := 1;
```

```
} else {
  remove a process P from S.queue
  place this process P on ready list
}
```

Deadlock and starvation due to semaphores :

- The implementation of a semaphores with a waiting queue may result in a situation where two or more process are waiting indefinitely for an event that can be caused only by one of the waiting process.such a state is called **Deadlock state.**

P_0	P_1
wait(S);	wait(Q);
wait(Q);	wait(S);
.	.
.	.
.	.
signal(S);	signal(Q);
signal(Q);	signal(S);

- **Starvation** – indefinite blocking. A process may never be removed from the semaphore queue (say, if LIFO) in which it is suspended.

Classic Problems of Synchronization

The following classic problems are used to test virtually every new proposed synchronization algorithm.

5.7.1 The Bounded-Buffer Problem

- This is a generalization of the producer-consumer problem wherein access is controlled to a shared group of buffers of a limited size.
- In this solution, the two counting semaphores "full" and "empty" keep track of the current number of full and empty buffers respectively (and initialized to 0 and N respectively.) The binary semaphore mutex controls access to the critical section. The producer and consumer processes are nearly identical - One can think of the producer as producing full buffers, and the consumer producing empty buffers.

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

Figure 5.9 The structure of the producer process.

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

Figure 5.10 The structure of the consumer process.

Readers-Writers Problem

A data set is shared among a number of concurrent processes.

- **Readers** – only read the data set; they do not perform any updates
- **Writers** – can both read and write

Problem – allow multiple readers to read at the same time. Only one single writer can access the shared data at the same time.

Shared Data

- Data set
- Semaphore mutex initialized to 1 (controls access to readcount)
- Semaphore wrt initialized to 1 (writer access)
- Integer readcount initialized to 0 (how many processes are reading object)

Structure of a writer process

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

The structure of a reader process

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

The Dining-Philosophers Problem

The dining philosophers problem is a classic synchronization problem involving the allocation of limited resources amongst a group of processes in a deadlock-free and starvation-free manner:

- Consider five philosophers sitting around a table, in which there are five chopsticks evenly distributed and an endless bowl of rice in the center, as shown in the diagram below. (There is exactly one chopstick between each pair of dining philosophers.)
- These philosophers spend their lives alternating between two activities: eating and thinking.
- When it is time for a philosopher to eat, it must first acquire two chopsticks - one from their left and one from their right.

- When a philosopher thinks, it puts down both chopsticks in their original locations.

Figure 5.13 - The situation of the dining philosophers

One possible solution, as shown in the following code section, is to use a set of five semaphores (chopsticks[5]), and to have each hungry philosopher first wait on their left chopstick (chopsticks[i]), and then wait on their right chopstick (chopsticks[(i + 1) % 5])
But suppose that all five philosophers get hungry at the same time, and each starts by picking up their left chopstick. They then look for their right chopstick, but because it is unavailable, they wait for it, forever, and eventually all the philosophers starve due to the resulting deadlock.

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

Figure 5.14 - The structure of philosopher i.

Some potential solutions to the problem include:

- Only allow four philosophers to dine at the same time. (Limited simultaneous processes.)
- Allow philosophers to pick up chopsticks only when both are available, in a critical section. (All or nothing allocation of critical resources.)
- Use an asymmetric solution, in which odd philosophers pick up their left chopstick first and even philosophers pick up their right chopstick first. (Will this solution always work? What if there are an even number of philosophers?)

Unit- III CPU Scheduling

1.1 Process Concept:

1.1.1 Process:

- A **process** is an executing program, including the current values of the **program counter, registers, and variables**.
- Difference between a process and a program is that the program is a **group of instructions** whereas the **process is the activity**.
- Process can be described:
  - **I/O Bound Process-** spends more time doing I/O then computation.
  - **CPU Bound Process-** spends more time doing computation.

1.1.2 Process States:

Start : The process has just arrived.

Ready : The process is waiting to grab the processor.

Running : The process has been allocated by the processor.

Waiting : The process is doing I/O work or blocked.

Halted : The process has finished and is about to leave the system.

1.1.3 Process/Task Control Block (PCB):

In the OS, each process is represented by its PCB (Process Control Block). The PCB, generally contains the following information:

• **Process State:** The state may be new, ready, running, waiting, halted, and so on.
• **Process ID :** each process has a unique ID.
• **Program Counter (PC) value:** The counter indicates the address of the next instruction to be executed or this process.
• **Register values:** The registers vary in number and type, depending on the computer architecture. They include accumulators, index registers, stack pointers, and general-purpose registers, plus any condition-code information. Along with the program counter, this state information must be saved when an interrupt occurs, to allow the process to be continued correctly afterward.

• **Memory Management Information** (page tables, base/bound registers etc.):
• **Processor Scheduling Information** (priority, last processor burst time etc.)
• **I/O Status Info** (outstanding I/O requests, I/O devices held, etc.)
• **List of Open Files**
• **Accounting Info:** This information includes the amount of CPU and real time used, time limits, account numbers, job or process numbers, and so on.

1.1.4 Operations on process:

A. Process Creation

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

- **Resource sharing**
  - Parent and children share all resources
  - Children share subset of parent's resources
  - Parent and child share no resources
- **Execution**
  - Parent and children execute concurrently
  - Parent waits until children terminate

B. Process Termination

- **Process executes last statement and asks the operating system to delete it (exit)**
  - Output data from child to parent (via wait)
  - Process' resources are deallocated by operating system
- **Parent may terminate execution of children processes (abort)**
  - Child has exceeded allocated resources
  - Task assigned to child is no longer required
  - If parent is exiting
    - Some operating system do not allow child to continue if its parent terminates

1.1.5 Process Address Space:

- The process address space consists of the linear address range presented to each process. Each process is given a **flat 32- or 64-bit** address space, with the size depending on the architecture. The term "flat" describes the fact that the address space exists in a single range. (As an example, a 32-bit address space extends from the address 0 to 429496729.)
- Some operating systems provide a **segmented address space**, with addresses existing not in a single linear range, but instead in multiple segments. Modern virtual memory operating systems generally have a flat memory model and not a segmented one.
- A memory address is a given value within the address space, such as 4021f000. The process can access a memory address only in a valid memory area. **Memory areas have associated permissions, such as readable, writable, and executable,** that the associated process must respect. If a process accesses a memory address not in a valid memory area, or if it accesses a valid area in an invalid manner, the kernel kills the process with the dreaded "Segmentation Fault" message.

- Memory areas can contain all sorts of goodies, such as
    - A memory map of the executable file's code, called the **text section.**
    - A memory map of the executable file's initialized global variables, called the **data section.**
    - A memory map of the zero page (a page consisting of all zeros, used for purposes such as this) containing uninitialized global variables, called the **bss section**
    - A memory map of the zero page used for the process's user-space stack (do not confuse this with the process's kernel stack, which is separate and maintained and used by the kernel)
    - An additional text, data, and bss section for each shared library, such as the C library and dynamic linker, loaded into the process's address space.
    - Any memory mapped files
    - Any shared memory segments
    - Any anonymous memory mappings, such as those associated with malloc().

1.1.6 Process Identification Information

- **Process Identifier** (**process ID** or **PID**) is a number used by most operating system kernels (such as that of UNIX, Mac OS X or Microsoft Windows) to temporarily uniquely identify a process.
- This number may be used as a parameter in various function calls allowing processes to be manipulated, such as adjusting the process's priority or killing it altogether.
- In Unix-like operating systems, new processes are created by the **fork()** system call. The PID is returned to the parent enabling it to refer to the child in further function calls. The parent may, for example, wait for the child to terminate with the **waitpid()** function, or terminate the process with **kill()**.

1.2 Threads:

1.2.1 Introduction to Thread:

- A thread is a basic unit of CPU utilization; it comprises a **thread ID, a program counter, a register set, and a stack.** It shares with other threads belonging to the same process its code section, data section, and other operating-system resources, such as open files and signals.
- A traditional process has a single thread of control. If a process has multiple threads of control, it can perform more than one task at a time.

1.2.2 Benefits of Threads:

A. Responsiveness. Multithreading an interactive application may allow a program to continue running even if part of it is blocked or is performing a lengthy operation.

B. Resource sharing. Threads share the memory and the resources of the process to which they belong. The benefit of sharing code and data is that it allows an application to have several different threads of activity within the same address space.

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

D. Utilization of multiprocessor architectures. The benefits of multithreading can be greatly increased in a multiprocessor architecture, where threads may be running in parallel on different processors. A single threaded process can only run on one CPU, no matter how many are available. Multithreading on a multi-CPU machine increases concurrency.

1.2.3 Multithreading Models (Management of Threads):

Threads may be provided either at the user level, for **user threads,** or by the kernel, for **kernel threads. User threads** are supported above the **kerne**l and are managed without kernel support, whereas **kernel threads** are supported and managed directly by the **operating system**. There must exist a relationship between user threads and kernel threads. There are three common ways of establishing this relationship.

A. Many-to-One Model:

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

B. One-to-One Model:

The one-to-one model maps **each user thread to a kernel thread**. It provides **more concurrency than the many-to-one model** by allowing another thread to run when a thread makes a blocking system call; it also allows **multiple threads to run in parallel** on multiprocessors**. The only drawback to this model is that creating a user thread requires creating the corresponding kernel thread.**

C. Many-to-Many Model:

The many-to-many model multiplexes many user-level threads to a smaller or equal number of kernel threads. **The one-to-one model allows for greater concurrency**.

The many-to-many model suffers from neither of these shortcomings:

Developers can **create as many user threads as necessary, and the corresponding kernel threads can run in parallel on a multiprocessor**. Also, when a thread performs a blocking system call, the kernel can schedule another thread for execution.

1.3 CPU Scheduling Concept:

The main objective of CPU Scheduling is to maximize CPU utilization. Basically we use **process scheduling** to maximize CPU utilization. Process Scheduling is done by following ways:

1.3.1 CPU-I/O Burst Cycle:

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.

1.3.2 Scheduling Queue: queue is generally stored as a linked list. A ready-queue header contains pointers to the first and final PCBs in the list. Each PCB includes a pointer field that points to the next PCB in the ready queue.

- **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

1.3.3 Schedulers: A process migrates among the various scheduling queues throughout its lifetime. The operating system must select, for scheduling purposes, processes from these queues in some fashion. The selection process is carried out by the appropriate **scheduler.** Schedulers have two types:

1. Long Term Scheduler	2. Short Term Scheduler
A. selects which process should be brought into the **ready queue.** **B.** L.T.S. is invoked very infrequently (seconds, minutes). **C.** L.T.S. controls the degree of multiprogramming (number of processes in memory)	**A.** selects which process should be executed next & allocates CPU. **B.** STS is invoked very frequently (milliseconds).

Medium-Term Scheduler:

Some operating systems, such as time-sharing systems, may introduce an additional, intermediate level of scheduling. **medium-term scheduler** is that sometimes it can be advantageous to remove processes from memory (and from active contention for the CPU) and thus reduce the degree of multiprogramming. Later, the process can be reintroduced into memory, and its execution can be continued where it left off. This scheme is called **swapping**.

1.3.4 Dispatcher:

Dispatcher is the module that gives control of the CPU to the process selected by the short-term scheduler. This function involves:

- Switching Context
  - 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.
  - Time dependent on hardware support.
- 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, given that 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 **dispatch latency.**

Q. Write short note on Preemptive Scheduling & Non-Preemptive Scheduling:

Ans. Preemptive scheduling: The preemptive scheduling is prioritized. The highest priority process should always be the process that is currently utilized.

Non-Preemptive scheduling: When a process enters the state of running, the state of that process is not deleted from the scheduler until it finishes its service time.

Non-Preemptive Scheduling may be of switching from **running to waiting state, running to ready state, waiting to ready states, process terminates**; while **others are preemptive**.

1.3.5 Scheduling Performance Criteria:

- **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).

Processor Utilization = (Processor Busy Time / (Processor Busy Time + Processor Idle time))*100

- **Throughput:** the number of processes that are completed per time unit, called throughput.

Throughput = No. of Process Completed / Time Unit

- **Turnaround Time:** The amount of time to execute a particular process is called turnaround time.

Turnaround Time = T(Process Completed) – T(Process Submitted)

- **Waiting Time:** the amount of time that a process spends waiting in the ready queue.

Waiting Time = Turnaround Time – Processing Time

- **Response Time:** time from the submission of a request until the first response is produced. This measure, called response time.

Response Time = T(First Response) – T(submission of request)

➢ **Optimization Criteria:**

- ✓ Max CPU utilization
- ✓ Max throughput
- ✓ Min turnaround time
- ✓ Min waiting time
- ✓ Min response time

1.3.6 Scheduling Algorithms:

A. First-Come, First-Served Scheduling

B. Shortest-Job-First Scheduling

C. Priority Scheduling

D. Round-Robin Scheduling

E. Multilevel Queue Scheduling

F. Multilevel Feedback Queue Scheduling

A. First-Come, First-Served Scheduling (FCFS):

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.

Example:

*Process **p1,p2,p3,p4,p5** having arrival time of **0,2,3,5,8** microseconds and processing time **3,3,1,4,2** microseconds, Draw Gantt Chart & Calculate Average Turn Around Time, Average Waiting Time, CPU Utilization & Throughput using **FCFS**.*

Processes	Arrival Time	Processing Time	T.A.T. **T(P.C.)-T(P.S.)**	W.T. **TAT- T(Proc.)**
P1	0	3	3-0=3	3-3=0
P2	2	3	6-2=4	4-3=1
P3	3	1	7-3=4	4-1=3
P4	5	4	11-5=6	6-4=2
P5	8	2	13-8=5	5-2=3

GANTT CHART:

P1	**P2**	**P3**	**P4**	**P5**

0 3 6 7 11 13

Average T.A.T. =(3+4+4+6+5)/5 = 22/5 = 4.4 Microsecond

Average W.T. = (0+1+3+2+3)/5 =9/5 = 1.8 Microsecond

CPU Utilization = (13/13)*100 = 100%

Throughput = 5/13 = 6.38

B. Shortest-Job-First Scheduling (SJF):

- Associate with each process the length of its next CPU burst. Use these lengths to schedule the process with the shortest time
- **Two schemes:**
  i. **nonpreemptive** – once CPU given to the process it cannot be preempted until completes its CPU burst
  ii. **preemptive** – if a new process arrives with CPU burst length less than remaining time of current executing process, preempt. **This scheme is known as the Shortest-Remaining-Time-First (SRTF)**
- **SJF is optimal** – gives minimum average waiting time for a given set of processes

Example:

*Process **p1,p2,p3,p4** having burst time of **6,8,7,3** microseconds. Draw Gantt Chart & Calculate Average Turn Around Time, Average Waiting Time, CPU Utilization & Throughput using **SJF.***

Processes	Burst Time	T.A.T. **T(P.C.)-T(P.S.)**	W.T. **TAT- T(Proc.)**
P4	3	3-0=3	3-3=0
P1	6	9-0=9	9-6=3
P3	7	16-0=16	16-7=9
P2	8	24-0=24	24-8=16

GANTT CHART

P4	P1	P3	P2

0 3 9 16 24

Average T.A.T. =(3+9+16+24)/4 = 13 microsecond

Average W.T. = (0+3+9+16)/4 =28/4 = 7 microsecond

CPU Utilization = (24/24)*100 = 100%

Throughput = 4/24

C. Priority Scheduling:

- A priority number (integer) is associated with each process
- The CPU is allocated to the process with the highest **priority (smallest integer ≡ highest priority)**
- **Problem** ≡ Starvation – low priority processes may never execute
- **Solution ≡ Aging** – as time progresses increase the priority of the process

Example: *Process **p1,p2,p3,p4,p5** having burst time of **10,1,2,1,5** microseconds and priorities are **3,1,4,5,2.** Draw Gantt Chart & Calculate Average Turn Around Time, Average Waiting Time, CPU Utilization & Throughput using **Priority Scheduling**.*

Processes	Priority	Processing Time	T.A.T. **T(P.C.)-T(P.S.)**	W.T. **TAT- T(Proc.)**
P2	1	1	1-0=1	1-1=0
P5	2	5	6-0=6	6-5=1
P1	3	10	16-0=16	16-10=6
P3	4	2	18-0=18	18-2=16
P4	5	1	19-0=19	19-1=18

GANTT CHART:

P2	**P5**	**P1**	**P3**	**P4**

0 1 6 16 18 19

Average T.A.T. =(1+6+16+18+19)/5 = 12 microsecond

Average W.T. = (0+1+6+16+18)/5 =41/5 = 8.2 microsecond

CPU Utilization = (19/19)*100 = 100%

Throughput = 5/19

D. Round-Robin Scheduling:

- Each process gets a small unit of CPU time (*time quantum*), usually 10-100 milliseconds. After this time has elapsed, the process is preempted and added to the end of the ready queue.
- If there are *n* processes in the ready queue and the time quantum is *q*, then each process gets 1/*n* of the CPU time in chunks of at most *q* time units at once. No process waits more than (*n*-1)*q* time units.
- Used for time sharing & multiuser O.S.

Example:

*Process **p1,p2,p3** having processing time of **24,3,3** milliseconds. Draw Gantt Chart & Calculate Average Turn Around Time, Average Waiting Time, CPU Utilization & Throughput using **Round Robin** with time slice of 4milliseconds.*

Processes	Processing Time	T.A.T. **T(P.C.)-T(P.S.)**	W.T. **TAT- T(Proc.)**
P1	24	30-0=30	30-24=6
P2	3	7-0=7	7-3=4
P3	3	10-0=10	10-3=7

GANTT CHART

P1	P2	P3	P1	P1	P1	P1	P1

0 4 7 10 14 18 22 26 30

Average T.A.T. =(30+7+7)/3 = 44/3 = 14.67 millisecond

Average W.T. = (6+4+7)/3 =17/3 = 5.67 millisecond

CPU Utilization = (30/30)*100 = 100%

Throughput = 3/30=0.1

E. Multilevel Feedback-Queue Scheduling

1. Ready queue is partitioned into separate queues:
2. The processes are permanently assigned to one queue based on some property of the process. such as memory size, process priority or process type.
3. Each queue has its own scheduling algorithm.
4. Scheduling must be done between the queues.
   - F Fixed priority scheduling;. Possibility of starvation.
   - F Time slice(RR)Scheduling – each queue gets a certain amount of CPU time which it can schedule amongst its processes

Multilevel Feedback Queue Scheduling

1. A process can move between the various queues
2. Process using too much CPU time will be moved to a lower priority queue.
3. Process waiting too long in a lower priority queue may be moved to a higher priority queue
4. This form of aging prevents starvation.

5. **Example**: in a MFQ scheduler with 3 queues, the scheduler first executes all the processes in queue 0. only when queue 0 is empty, it will execute processes in queue 1. Similarly processes in queue 2 will be executed only if queues 0 and 1 are empty.
6. Three queues:

   Q_0 – time quantum 8 milliseconds(RR)

   Q_1 – time quantum 16 milliseconds(RR)

   Q_2 – FCFS
7. Scheduling
   1. A new job enters queue Q_0. When it gains CPU, job receives 8 milliseconds. If it does not finish in 8 milliseconds, job is moved to tail of queue Q_1.
   2. If queue Q_0 *is* empty, the process at the head of Q_1 is given a quantum of 16 additional milliseconds. If it still does not complete, it is preempted and moved to queue Q_2.Processes in queue 2 are run on an FCFS basis only when queue Q_0 and Q_1 are empty

   Multilevel-feedback-queue scheduler defined by the following parameters:

   1. number of queues
   2. scheduling algorithms for each queue
   3. method used to determine when to upgrade a process
   4. method used to determine when to demote a process
   5. method used to determine which queue a process will enter when that process needs service

Multiple-Processor Scheduling:

CPU scheduling more complex when multiple CPUs are available.

1. **Homogeneous multiprocessor system :** Processors are identical in terms of functionality; any available processor can be used to run any process in the queue. Load sharing can be done. In order to avoid unbalancing, we can allow all processes to go to one queue and are scheduled onto available processor.

Scheduling approach in Homogeneous MP

Two scheduling approach are used.

a) Self scheduling:

Each processor is self scheduling. Each process examines the ready queue and selects a process to execute.

Problem arises when more than one processor access the same process from ready queue. Different techniques are used to resolve the problem.

b) Master - Slave structure :

In this approach, one processor can be appointed as scheduler for the other processors. The other processor only executes user code.

2. Heterogeneous multiprocessor system

Processors are different in these systems. Only program compiled for a given processor's instruction set could be run on that processor.

- Simpler than homogeneous multiprocessor system.
- No need of data sharing.

Unit 4 part 1 & 2

Address Binding

- User programs typically refer to memory addresses with symbolic names These symbolic names must be mapped or ***bound*** to physical memory addresses, which typically occurs in several stages:
  - **Compile Time -** If it is known at compile time where a program will reside in physical memory, then ***absolute code*** can be generated by the compiler, containing actual physical addresses. However if the load address changes at some later time, then the program will have to be recompiled. DOS .COM programs use compile time binding.
  - **Load Time -** If the location at which a program will be loaded is not known at compile time, then the compiler must generate ***relocatable code***, which references addresses relative to the start of the program. If that starting address changes, then the program must be reloaded but not recompiled.
  - **Execution Time -** If a program can be moved around in memory during the course of its execution, then binding must be delayed until execution time. This requires special hardware, and is the method implemented by most modern OSes.

Logical Versus Physical Address Space

- The address generated by the CPU is a ***logical address***, whereas the address actually seen by the memory hardware is a ***physical address***.
- Addresses bound at compile time or load time have identical logical and physical addresses.
- Addresses created at execution time, however, have different logical and physical addresses.
  - In this case the logical address is also known as a ***virtual address***, and the two terms are used interchangeably by our text.
  - The set of all logical addresses used by a program composes the ***logical address space***, and the set of all corresponding physical addresses composes the ***physical address space.***
- The run time mapping of logical to physical addresses is handled by the ***memory-management unit, MMU***.
  - The MMU can take on many forms. One of the simplest is a modification of the base-register scheme described earlier.
  - The base register is now termed a ***relocation register***, whose value is added to every memory request at the hardware level.

- Note that user programs never see physical addresses. User programs work entirely in logical address space, and any memory references or manipulations are done using purely logical addresses. Only when the address gets sent to the physical memory chips is the physical memory address generated.

Swapping

- A process must be loaded into memory in order to execute.
- If there is not enough memory available to keep all running processes in memory at the same time, then some processes who are not currently using the CPU may have their memory swapped out to a fast local disk called

Swapping of two processes using a disk as a backing store

Contiguous Memory Allocation

- One approach to memory management is to load each process into a contiguous space. The operating system is allocated space first, usually at either low or high memory locations, and then the remaining available memory is allocated to processes as needed. (The OS is usually loaded low, because that is where the interrupt vectors are located, but on older systems part of the OS was loaded high to make more room in low memory (within the 640K barrier) for user processes.)

Memory Protection (was Memory Mapping and Protection)

- The system shown in Figure 8.6 below allows protection against user programs accessing areas that they should not, allows programs to be relocated to different memory starting addresses as needed, and allows the memory space devoted to the OS to grow or shrink dynamically as needs change.

- One method of allocating contiguous memory is to divide all available memory into equal sized partitions, and to assign each process to their own partition. This restricts both the number of simultaneous processes and the maximum size of each process, and is no longer used.
- An alternate approach is to keep a list of unused (free) memory blocks (holes), and to find a hole of a suitable size whenever a process needs to be loaded into memory. There are many different strategies for finding the "best" allocation of memory to processes, including the three most commonly discussed:

1. **First fit** - Search the list of holes until one is found that is big enough to satisfy the request, and assign a portion of that hole to that process. Whatever fraction of the hole not needed by the request is left on the free list as a smaller hole. Subsequent requests may start looking either from the beginning of the list or from the point at which this search ended.
2. **Best fit** - Allocate the *smallest* hole that is big enough to satisfy the request. This saves large holes for other process requests that may need them later, but the resulting unused portions of holes may be too small to be of any use, and will therefore be wasted. Keeping the free list sorted can speed up the process of finding the right hole.
3. **Worst fit** - Allocate the largest hole available, thereby increasing the likelihood that the remaining portion will be usable for satisfying future requests.

- Simulations show that either first or best fit are better than worst fit in terms of both time and storage utilization. First and best fits are about equal in terms of storage utilization, but first fit is faster.

Fragmentation

- All the memory allocation strategies suffer from ***external fragmentation***, though first and best fits experience the problems more so than worst fit. External fragmentation means that the available memory is broken up into lots of little pieces, none of which is big enough to satisfy the next memory requirement, although the sum total could.
- The amount of memory lost to fragmentation may vary with algorithm, usage patterns, and some design decisions such as which end of a hole to allocate and which end to save on the free list.
- Statistical analysis of first fit, for example, shows that for N blocks of allocated memory, another 0.5 N will be lost to fragmentation.
- ***Internal fragmentation*** also occurs, with all memory allocation strategies. This is caused by the fact that memory is allocated in blocks of a fixed size, whereas the actual memory needed will rarely be that exact size
- If the programs in memory are relocatable, (using execution-time address binding), then the external fragmentation problem can be reduced via ***compaction***, i.e. moving all processes down to one end of physical memory. This only involves updating the relocation register for each process, as all internal work is done using logical addresses.

Segmentation :

Basic Method

- Most users (programmers) do not think of their programs as existing in one continuous linear address space.
- Rather they tend to think of their memory in multiple ***segments***, each dedicated to a particular use, such as code, data, the stack, the heap, etc.
- Memory ***segmentation*** supports this view by providing addresses with a segment number (mapped to a segment base address) and an offset from the beginning of that segment.

Programmer's view of a program.

Segmentation Hardware

- A ***segment table*** maps segment-offset addresses to physical addresses, and simultaneously checks for invalid addresses, using a system similar to the page tables and relocation base registers discussed previously.

Figure - Segmentation hardware

Figure - Example of segmentation

Paging

- Paging is a memory management scheme that allows processes physical memory to be discontinuous, and which eliminates problems with fragmentation by allocating memory in equal sized blocks known as ***pages***.

Basic Method

- The basic idea behind paging is to divide physical memory into a number of equal sized blocks called ***frames***, and to divide a programs logical memory space into blocks of the same size called ***pages.***
- Any page (from any process) can be placed into any available frame.
- The ***page table*** is used to look up what frame a particular page is stored in at the moment. In the following example, for instance, page 2 of the program's logical memory is currently stored in frame 3 of physical memory:

Figure - Paging hardware

Figure 8.11 - Paging model of logical and physical memory

- A logical address consists of two parts: A page number in which the address resides, and an offset from the beginning of that page. (The number of bits in

the page number limits how many pages a single process can address. The number of bits in the offset determines the maximum size of each page, and should correspond to the system frame size.)

- The page table maps the page number to a frame number, to yield a physical address which also has two parts: The frame number and the offset within that frame. The number of bits in the frame number determines how many frames the system can address, and the number of bits in the offset determines the size of each frame.

- Consider the following micro example, in which a process has 16 bytes of logical memory, mapped in 4 byte pages into 32 bytes of physical memory. (Presumably some other processes would be consuming the remaining 16 bytes of physical memory.)

Figure 8.12 - Paging example for a 32-byte memory with 4-byte pages

- There is no external fragmentation with paging. All blocks of physical memory are used, and there are no gaps in between and no problems with finding the right sized hole for a particular chunk of memory.
- There is, however, internal fragmentation. Memory is allocated in chunks the size of a page, and on the average, the last page will only be half full, wasting on the average half a page of memory per process.
- When a process requests memory (e.g. when its code is loaded in from disk), free frames are allocated from a free-frame list, and inserted into that process's page table.

Figure 8.13 - Free frames (a) before allocation and (b) after allocation

Hardware Support

- to store the page table in main memory, and to use a single register (called the ***page-table base register, PTBR***) to record where in memory the page table is located.
- use a very special high-speed memory device called the ***translation look-aside buffer, TLB.***The benefit of the TLB is that it can search an entire table for a key value in parallel, and if it is found anywhere in the table, then the corresponding lookup value is returned.

Figure - Paging hardware with TLB

.

- The percentage of time that the desired information is found in the TLB is termed the ***hit ratio***.

For example, suppose that it takes 100 nanoseconds to access main memory, and only 20 nanoseconds to search the TLB. So a TLB hit takes 120 nanoseconds total (20 to find the frame number and then another 100 to go get the data), and a TLB miss takes 220 (20 to search the TLB, 100 to go get the frame number, and then another 100 to go get the data.) So with an 80% TLB hit ratio, the average memory access time would be:

0.80 * 120 + 0.20 * 220 = 140 nanoseconds

Protection

- The page table can also help to protect processes from accessing memory that they shouldn't, or their own memory in ways that they shouldn't.
- A bit or bits can be added to the page table to classify a page as read-write, read-only, read-write-execute, or some combination of these sorts of things. Then each memory reference can be checked to ensure it is accessing the memory in the appropriate mode.
    - Valid / invalid bits can be added to "mask off" entries in the page table that are not in use by the current process, as shown by example in Figure 8.12 below.

Figure 8.15 - Valid (v) or invalid (i) bit in page table

Shared Pages

- Paging systems can make it very easy to share blocks of memory, by simply duplicating page numbers in multiple page frames. This may be done with either code or data.
- In the example given below, three different users are running the editor simultaneously, but the code is only loaded into memory (in the page frames) one time.

Figure - Sharing of code in a paging environment

Structure of the Page Table

Hierarchical Paging

- Most modern computer systems support logical address spaces of 2^32 to 2^64.
- One option is to use a two-tier paging system, i.e. to page the page table.

  For example, the 20 bits described above could be broken down into two 10-bit page numbers. The first identifies an entry in the outer page table, which identifies where in memory to find one page of an inner page table. The second 10 bits finds a specific entry in that inner page table, which in turn identifies a particular frame in physical memory. (The remaining 12 bits of the 32 bit logical address are the offset within the 4K frame.)

Figure A two-level page-table scheme

Figure - Address translation for a two-level 32-bit paging architecture

Hashed Page Tables

- One common data structure for accessing data that is sparsely distributed over a broad range of possible values is with ***hash tables***.

- **Figure - Hashed page table**

8.6.3 Inverted Page Tables

- Another approach is to use an ***inverted page table***. Instead of a table listing all of the pages for a particular process, an inverted page table lists all of the pages currently loaded in memory, for all processes. (I.e. there is one entry per ***frame*** instead of one entry per ***page***.)
- Access to an inverted page table can be slow, as it may be necessary to search the entire table in order to find the desired page (or to discover that it is not there.)

Inverted page table

Unit 4 part 3

Virtual Memory

- Virtual memory is the separation of user logical memory from physical memory.
- It allows the execution of processes that may not be completely in memory.
- Virtual memory is commonl;y implemented by demand paging
- Demand segmentation can also be used to provide virtual memory.

Demand Paging

- The basic idea behind ***demand paging*** is that when a process is swapped in, its pages are not swapped in all at once. Rather they are swapped in only when the process needs them. (on demand.) This is termed a ***lazy swapper***, although a ***pager*** is a more accurate term.

Transfer of a paged memory to contiguous disk space

Basic Concepts

- The basic idea behind paging is that when a process is swapped in, the pager only loads into memory those pages that it expects the process to need (right away.)
- Pages that are not loaded into memory are marked as invalid in the page table, using the invalid bit. (The rest of the page table entry may either be blank or contain information about where to find the swapped-out page on the hard drive.)
- If the process only ever accesses pages that are loaded in memory (***memory resident*** pages), then the process runs exactly as if all the pages were loaded in to memory.

- In an extreme case, NO pages are swapped in for a process until they are requested by page faults. This is known as ***pure demand paging.***
- In theory each instruction could generate multiple page faults. In practice this is very rare, due to ***locality of reference***,

Page table when some pages are not in main memory

Page fault:

- Page fault occurs when the process tries to access a page that was not brought into memory .
- The procedure for handling page fault is as given :
    1. The memory address requested is first checked, to make sure it was a valid memory request.

2. If the reference was invalid, the process is terminated. Otherwise, the page must be paged in.
3. A free frame is located, possibly from a free-frame list.
4. A disk operation is scheduled to bring in the necessary page from disk. (This will usually block the process on an I/O wait, allowing some other process to use the CPU in the meantime.)
5. When the I/O operation is complete, the process's page table is updated with the new frame number, and the invalid bit is changed to indicate that this is now a valid page reference.
6. The instruction that caused the page fault must now be restarted from the beginning, (as soon as this process gets another turn on the CPU.)

Steps in handling a page fault

Page Replacement

Find some page in memory that isn't being used right now, and swap that page only out to disk, freeing up a frame that can be allocated to the process requesting it. This is known as ***page replacement***, and is the most common solution. There are many different algorithms for page replacement.

Basic Page Replacement

- The previously discussed page-fault processing assumed that there would be free frames available on the free-frame list. Now the page-fault handling must be modified to free up a frame if necessary, as follows:
    1. Find the location of the desired page on the disk, either in swap space or in the file system.
    2. Find a free frame:
        a. If there is a free frame, use it.
        b. If there is no free frame, use a page-replacement algorithm to select an existing frame to be replaced, known as the ***victim frame***.
        c. Write the victim frame to disk. Change all related page tables to indicate that this page is no longer in memory.
    3. Read in the desired page and store it in the frame. Adjust all related page and frame tables to indicate the change.
    4. Restart the process that was waiting for this page.

Steps in handling a page fault

Page replacement algorithms:

FIFO Page Replacement

- A simple and obvious page replacement strategy is ***FIFO***, i.e. first-in-first-out.
- As new pages are brought in, they are added to the tail of a queue, and the page at the head of the queue is the next victim. In the following example, 20 page requests result in 15 page faults:

Page-fault curve for FIFO replacement on a reference string

- Although FIFO is simple and easy, it is not always optimal, or even efficient.

- An interesting effect that can occur with FIFO is ***Belady's anomaly***, in which increasing the number of frames available can actually ***increase*** the number of page faults that occur

Optimal Page Replacement

- The discovery of Belady's anomaly lead to the search for an ***optimal page-replacement algorithm***, which is simply that which yields the lowest of all possible page-faults, and which does not suffer from Belady's anomaly.
- Such an algorithm does exist, and is called ***OPT or MIN.*** This algorithm is simply "Replace the page that will not be used for the longest time in the future."
- For example, Figure 9.14 shows that by applying OPT to the same reference string used for the FIFO example, the minimum number of possible page faults is 9.
- Unfortunately OPT cannot be implemented in practice,

LRU Page Replacement

- The prediction behind ***LRU,*** the ***Least Recently Used,*** algorithm is that the page that has not been used in the longest time is the one that will not be used again in the near future. (Note the distinction between FIFO and LRU: The former looks at the oldest ***load*** time, and the latter looks at the oldest ***use*** time.).
- LRU for our sample string, yielding 12 page faults, (as compared to 15 for FIFO and 9 for OPT.)

Allocation of Frames

We said earlier that there were two important tasks in virtual memory management: a page-replacement strategy and a frame-allocation strategy. This section covers the second part of that pair.

Minimum Number of Frames

- The absolute minimum number of frames that a process must be allocated is dependent on system architecture

Allocation Algorithms

- **Equal Allocation -** If there are m frames available and n processes to share them, each process gets m / n frames, and the leftovers are kept in a free-frame buffer pool.
- **Proportional Allocation -** Allocate the frames proportionally to the size of the process, relative to the total size of all processes. So if the size of process i is S_i, and S is the sum of all S_i, then the allocation for process P_i is a_i = m * S_i / S.
- Variations on proportional allocation could consider priority of process rather than just their size.

Global versus Local Allocation

- whether frame allocation (page replacement) occurs on a local or global level.
- With local replacement, the number of pages allocated to a process is fixed, and page replacement occurs only amongst the pages allocated to this process.
- With global replacement, any page may be a potential victim, whether it currently belongs to the process seeking a free frame or not.
- Local page replacement allows processes to better control their own page fault rates, and leads to more consistent performance of a given process over different system load levels.
- Global page replacement is overall more efficient, and is the more commonly used approach.

Thrashing

- If a process cannot maintain its minimum required number of frames, then it must be swapped out, freeing up frames for other processes. This is an intermediate level of CPU scheduling.
- But what about a process that can keep its minimum, but cannot keep all of the frames that it is currently using on a regular basis? In this case it is forced to page out pages that it will need again in the very near future, leading to large numbers of page faults.
- A process that is spending more time paging than executing is said to be ***thrashing.***

Cause of Thrashing

- Early process scheduling schemes would control the level of multiprogramming allowed based on CPU utilization, adding in more processes when CPU utilization was low.
- The problem is that when memory filled up and processes started spending lots of time waiting for their pages to page in, then CPU utilization would lower, causing the schedule to add in even more processes and exacerbating the problem! Eventually the system would essentially grind to a halt.
- Local page replacement policies can prevent one thrashing process from taking pages away from other processes, but it still tends to clog up the I/O queue, thereby slowing down any other process that needs to do even a little bit of paging (or any other I/O for that matter.)
-

Solution of thrashing:

1. Working-Set Model

- The ***working set model*** is based on the concept of locality, and defines a ***working set window***, of length ***delta.*** Whatever pages are included in the most recent delta page references are said to be in the processes working set window, and comprise its current working set, as illustrated:

- The selection of delta is critical to the success of the working set model - If it is too small then it does not encompass all of the pages of the current locality, and if it is too large, then it encompasses pages that are no longer being frequently accessed.
- The total demand, D, is the sum of the sizes of the working sets for all processes. If D exceeds the total number of available frames, then at least one process is thrashing, because there are not enough frames available to satisfy its minimum working set. If D is significantly less than the currently available frames, then additional processes can be launched.
- The hard part of the working-set model is keeping track of what pages are in the current working set, since every reference adds one to the set and removes one older page. An approximation can be made using reference bits and a timer that goes off after a set interval of memory references:

2. Page fault frequency:

- A more direct approach is to recognize that what we really want to control is the page-fault rate, and to allocate frames based on this directly measurable

value. If the page-fault rate exceeds a certain upper bound then that process needs more frames, and if it is below a given lower bound, then it can afford to give up some of its frames to other processes.

- (I suppose a page-replacement strategy could be devised that would select victim frames based on the process with the lowest current page-fault frequency.)

Unit 5 (part-1) Mass-Storage Structure

Magnetic Disks

One or more ***platters*** in the form of disks covered with magnetic media. ***Hard disk*** platters are made of rigid metal,

Each platter has two working ***surfaces.*** Each working surface is divided into a number of concentric rings called ***tracks.***

The collection of all tracks that are the same distance from the edge of the platter, (i.e. all tracks immediately above one another in the following diagram) is called a ***cylinder***.

Each track is further divided into ***sectors,*** traditionally containing 512 bytes of data each, although some modern disks occasionally use larger sector sizes

The data on a hard drive is read by read-write ***heads.*** The standard configuration (shown below) uses one head per surface, each on a separate ***arm***, and controlled by a common ***arm assembly*** which moves all heads simultaneously from one cylinder to another. the number of sectors per track, times the number of bytes per sector.

o The storage capacity of a traditional disk drive is equal to the number of heads (i.e. the number of working surfaces), times the number of tracks per surface, imes

The rate at which data can be transferred from the disk to the computer is composed of several steps:

- The ***positioning time***, a.k.a. the ***seek time*** or ***random access time*** is the time required to move the heads from one cylinder to another, and for the heads to settle down after the move.
- The ***rotational latency*** is the amount of time required for the desired sector to rotate around and come under the read-write head.
- The ***transfer rate***, which is the time required to move the data electronically from the disk to the computer

Disk Scheduling

FCFS Scheduling: Illustration shows total head movement of 640 cylinders

SSTF Scheduling

- ***Shortest Seek Time First*** scheduling is more efficient, but may lead to starvation if a constant stream of requests arrives for the same general area of the disk.
- SSTF reduces the total head movement to 236 cylinders, down from 640 required for the same set of requests under FCFS.

SCAN Scheduling

- The ***SCAN*** algorithm, a.k.a. the ***elevator*** algorithm moves back and forth from one end of the disk to the other, similarly to an elevator processing requests in a tall building.

Illustration shows total head movement of 208 cylinders

Under the SCAN algorithm, If a request arrives just ahead of the moving head then it will be processed right away, but if it arrives just after the head has passed, then it will have to wait for the head to pass going the other way on the return trip.

C-SCAN Scheduling

- The ***Circular-SCAN*** algorithm improves upon SCAN by treating all requests in a circular queue fashion - Once the head reaches the end of the disk, it returns to the other end without processing any requests, and then starts again from the beginning of the disk:

C-SCAN disk scheduling.

LOOK Scheduling

- ***LOOK*** scheduling improves upon SCAN by looking ahead at the queue of pending requests, and not moving the heads any farther towards the end of the disk than is necessary.

- The following diagram illustrates the C- LOOK:

C-LOOK disk scheduling.

Selection of a Disk-Scheduling Algorithm

- With very low loads all algorithms are equal, since there will normally only be one request to process at a time.
- For slightly larger loads, SSTF offers better performance than FCFS, but may lead to starvation when loads become heavy enough.
- For busier systems, SCAN and LOOK algorithms eliminate starvation problems.

RAID Levels

Mirroring provides reliability but is expensive;

Striping improves performance, but does not improve reliability.

Accordingly there are a number of different schemes that combine the principals of mirroring and striping in different ways, in order to balance reliability versus performance versus cost.

These are described by different ***RAID levels***, as follows: (In the diagram that follows, "C" indicates a copy, and "P" indicates parity, i.e. checksum bits.)

1. ***Raid Level 0 -*** This level includes striping only, with no mirroring.
2. ***Raid Level 1 -*** This level includes mirroring only, no striping.
3. ***Raid Level 2 -*** This level stores error-correcting codes on additional disks, allowing for any damaged data to be reconstructed by subtraction from the remaining undamaged data. Note that this scheme requires only three extra disks to protect 4 disks worth of data, as opposed to full mirroring. (The number of disks required is a function of the error-correcting algorithms, and the means by which the particular bad bit(s) is(are) identified.)

4. ***Raid Level 3 -*** This level is similar to level 2, except that it takes advantage of the fact that each disk is still doing its own error-detection, so that when an error occurs, there is no question about which disk in the array has the bad data. As a result a single parity bit is all that is needed to recover the lost data from an array of disks. Level 3 also includes striping, which improves performance. The downside with the parity approach is that every disk must take part in every disk access, and the parity bits must be constantly calculated and checked, reducing performance. Hardware-level parity calculations and NVRAM cache can help with both of those issues. In practice level 3 is greatly preferred over level 2.

5. ***Raid Level 4 -*** This level is similar to level 3, employing block-level striping instead of bit-level striping. The benefits are that multiple blocks can be read independently, and changes to a block only require writing two blocks (data and parity) rather than involving all disks. Note that new disks can be added seamlessly to the system provided they are initialized to all zeros, as this does not affect the parity results.

6. ***Raid Level 5 -*** This level is similar to level 4, except the parity blocks are distributed over all disks, thereby more evenly balancing the load on the system. For any given block on the disk(s), one of the disks will hold the parity information for that block and the other N-1 disks will hold the data. Note that the same disk cannot hold both data and parity for the same block, as both would be lost in the event of a disk crash.

7. ***Raid Level 6 -*** This level extends raid level 5 by storing multiple bits of error-recovery codes, each bit position of data, rather than a single parity bit. In the example shown below 2 bits of ECC are stored for every 4 bits of data, allowing data recovery in the face of up to two simultaneous disk failures. Note that this still involves only 50% increase in storage needs, as opposed to 100% for simple mirroring which could only tolerate a single disk failure

(a) RAID 0: non-redundant striping.

(b) RAID 1: mirrored disks.

(c) RAID 2: memory-style error-correcting codes.

(d) RAID 3: bit-interleaved parity.

(e) RAID 4: block-interleaved parity.

(f) RAID 5: block-interleaved distributed parity.

(g) RAID 6: P + Q redundancy.

There are also two RAID levels which combine RAID levels 0 and 1 (striping and mirroring) in different combinations, designed to provide both performance and reliability at the expense of increased cost.

- **RAID level 0 + 1** disks are first striped, and then the striped disks mirrored to another set. This level generally provides better performance than RAID level 5.
- **RAID level 1 + 0** mirrors disks in pairs, and then stripes the mirrored pairs. The storage capacity, performance, etc. are all the same, but there is an advantage to this approach in the event of multiple disk failures, as illustrated below:.

- In diagram (a) below, the 8 disks have been divided into two sets of four, each of which is striped, and then one stripe set is used to mirror the other set.
    - If a single disk fails, it wipes out the entire stripe set, but the system can keep on functioning using the remaining set.
    - However if a second disk from the other stripe set now fails, then the entire system is lost, as a result of two disk failures.
- In diagram (b), the same 8 disks are divided into four sets of two, each of which is mirrored, and then the file system is striped across the four sets of mirrored disks.
    - If a single disk fails, then that mirror set is reduced to a single disk, but the system rolls on, and the other three mirror sets continue mirroring.
    - Now if a second disk fails, (that is not the mirror of the already failed disk), then another one of the mirror sets is reduced to a single disk, but the system can continue without data loss.
    - In fact the second arrangement could handle as many as four simultaneously failed disks, as long as no two of them were from the same mirror pair.

Unit-5(part-2) file system

File Concept

11.1.1 File Attributes

- Different OSes keep track of different file attributes, including:
    - **Name -** Some systems give special significance to names, and particularly extensions (.exe, .txt, etc.), and some do not. Some extensions may be of significance to the OS (.exe), and others only to certain applications (.jpg)
    - **Identifier** (e.g. inode number)
    - **Type -** Text, executable, other binary, etc.
    - **Location -** on the hard drive.
    - **Size-** size of the file in bytes
    - **Protection-** read access, write access, execute etc
    - **Time & Date :** file creation time, last modification time
    - **User ID :** id of the file creater

File Operations

- The file supports many common operations:
    - Creating a file
    - Writing a file
    - Reading a file
    - Repositioning within a file
    - Deleting a file
    - Truncating a file.

File types:

file type	usual extension	function
executable	exe, com, bin or none	ready-to-run machine-language program
object	obj, o	compiled, machine language, not linked
source code	c, cc, java, perl, asm	source code in various languages
batch	bat, sh	commands to the command interpreter
markup	xml, html, tex	textual data, documents
word processor	xml, rtf, docx	various word-processor formats
library	lib, a, so, dll	libraries of routines for programmers
print or view	gif, pdf, jpg	ASCII or binary file in a format for printing or viewing
archive	rar, zip, tar	related files grouped into one file, sometimes compressed, for archiving or storage
multimedia	mpeg, mov, mp3, mp4, avi	binary file containing audio or A/V information

Access Methods

11.2.1 Sequential Access

- A sequential access file emulates magnetic tape operation, and generally supports a few operations:
  - read next - read a record and advance the tape to the next position.
  - write next - write a record and advance the tape to the next position.
  - rewind

- skip n records - May or may not be supported. N may be limited to positive numbers, or may be limited to +/- 1.

Figure - Sequential-access file.

Direct Access

- Jump to any record and read that record. Operations supported include:
  - read n - read record number n. (Note an argument is now required.)
  - write n - write record number n. (Note an argument is now required.)
  - jump to record n - could be 0 or the end of file.
  - Query current record - used to return back to this record later.
  - Sequential access can be easily emulated using direct access. The inverse is complicated and inefficient

Index access

- An indexed access scheme can be easily built on top of a direct access system. Very large files may require a multi-tiered indexing scheme, i.e. indexes of indexes.

Figure - Example of index and relative files.

Directory Structure

Storage Structure

- A disk can be used in its entirety for a file system.
- Alternatively a physical disk can be broken up into multiple ***partitions, slices, or mini-disks***, each of which becomes a virtual disk and can have its own filesystem. (or be used for raw storage, swap space, etc.)
- Or, multiple physical disks can be combined into one ***volume***, i.e. a larger virtual disk, with its own filesystem spanning the physical disks.

Figure - A typical file-system organization.

File Sharing

Multiple Users

- On a multi-user system, more information needs to be stored for each file:
  - The owner (user) who owns the file, and who can control its access.
  - The group of other user IDs that may have some special access to the file.
  - What access rights are afforded to the owner (User), the Group, and to the rest of the world (the universe, a.k.a. Others.)
  - Some systems have more complicated access control, allowing or denying specific accesses to specifically named users or groups.

Remote File Systems

- The advent of the Internet introduces issues for accessing files stored on remote computers

- The original method was ftp, allowing individual files to be transported across systems as needed. Ftp can be either account and password controlled, or *anonymous*, not requiring any user name or password.
- Various forms of *distributed file systems* allow remote file systems to be mounted onto a local directory structure, and accessed using normal file access commands. (The actual files are still transported across the network as needed, possibly using ftp as the underlying transport mechanism.)
- The WWW has made it easy once again to access files on remote systems without mounting their filesystems, generally using (anonymous) ftp as the underlying file transport mechanism.

Client-Server Model

- When one computer system remotely mounts a filesystem that is physically located on another system, the system which physically owns the files acts as a *server*, and the system which mounts them is the *client.*
- User IDs and group IDs must be consistent across both systems for the system to work properly. (I.e. this is most applicable across multiple computers managed by the same organization, shared by a common group of users.)
- The same computer can be both a client and a server. (E.g. cross-linked file systems.)

Distributed Information Systems

- The *Domain Name System, DNS,* provides for a unique naming system across all of the Internet.
- Domain names are maintained by the *Network Information System, NIS*, which unfortunately has several security issues. NIS+ is a more secure version, but has not yet gained the same widespread acceptance as NIS.

Protection

- Files must be kept safe for reliability (against accidental damage), and protection (against deliberate malicious access.) The former is usually managed with backup copies. This section discusses the latter.
- One simple protection scheme is to remove all access to a file. However this makes the file unusable, so some sort of controlled access must be arranged.

Types of Access

- The following low-level operations are often controlled:
  - Read - View the contents of the file
  - Write - Change the contents of the file.
  - Execute - Load the file onto the CPU and follow the instructions contained therein.
  - Append - Add to the end of an existing file.
  - Delete - Remove a file from the system.
  - List -View the name and other attributes of files on the system.
- Higher-level operations, such as copy, can generally be performed through combinations of the above.

Access Control

- One approach is to have complicated ***Access Control Lists, ACL,*** which specify exactly what access is allowed or denied for specific users or groups.
  - The AFS uses this system for distributed access.
  - Control is very finely adjustable, but may be complicated, particularly when the specific users involved are unknown. (AFS allows some wild cards, so for example all users on a certain remote system may be trusted, or a given username may be trusted when accessing from any remote system.)
- UNIX uses a set of 9 access control bits, in three groups of three. These correspond to R, W, and X permissions for each of the Owner, Group, and Others. (See "man chmod" for full details.) The RWX bits control the following privileges for ordinary files and directories:

bit	Files	Directories
R	Read (view) file contents.	Read directory contents. Required to get a listing of the directory.
W	Write	Change directory contents. Required to create or delete files.

	(change) file contents.	
X	Execute file contents as a program.	Access detailed directory information. Required to get a long listing, or to access any specific file in the directory. Note that if a user has X but not R permissions on a directory, they can still access specific files, but only if they already know the name of the file they are trying to access.

File-System Structure

- Hard disks have two important properties that make them suitable for secondary storage of files in file systems: (1) Blocks of data can be rewritten in place, and (2) they are direct access, allowing any block of data to be accessed with only (relatively) minor movements of the disk heads and rotational latency. (See Chapter 12)

- Disks are usually accessed in physical blocks, rather than a byte at a time. Block sizes may range from 512 bytes to 4K or larger.
- File systems organize storage on disk drives, and can be viewed as a layered design:
    - At the lowest layer are the physical devices, consisting of the magnetic media, motors & controls, and the electronics connected to them and controlling them. Modern disk put more and more of the electronic controls directly on the disk drive itself, leaving relatively little work for the disk controller card to perform.
    - ***I/O Control*** consists of ***device drivers***, special software programs (often written in assembly) which communicate with the devices by reading and writing special codes directly to and from memory addresses corresponding to the controller card's registers. Each controller card (device) on a system has a different set of addresses (registers, a.k.a. ***ports***) that it listens to, and a unique set of command codes and results codes that it understands.
    - The ***basic file system*** level works directly with the device drivers in terms of retrieving and storing raw blocks of data, without any consideration for what is in each block. Depending on the system, blocks may be referred to with a single block number, (e.g. block # 234234), or with head-sector-cylinder combinations.
    - The ***file organization module*** knows about files and their logical blocks, and how they map to physical blocks on the disk. In addition to translating from logical to physical blocks, the file organization module also maintains the list of free blocks, and allocates free blocks to files as needed.
    - The ***logical file system*** deals with all of the meta data associated with a file (UID, GID, mode, dates, etc), i.e. everything about the file except the data itself. This level manages the directory structure and the mapping of file names to ***file control blocks, FCBs***, which contain all of the meta data as well as block number information for finding the data on the disk.

Figure - Layered file system.

Directory Implementation

- Directories need to be fast to search, insert, and delete, with a minimum of wasted disk space.

Linear List

- A linear list is the simplest and easiest directory structure to set up, but it does have some drawbacks.
- Finding a file (or verifying one does not already exist upon creation) requires a linear search.
- Deletions can be done by moving all entries, flagging an entry as deleted, or by moving the last entry into the newly vacant position.
- Sorting the list makes searches faster, at the expense of more complex insertions and deletions.
- A linked list makes insertions and deletions into a sorted list easier, with overhead for the links.
- More complex data structures, such as B-trees, could also be considered.

Hash Table

- A hash table can also be used to speed up searches.
- Hash tables are generally implemented ***in addition to*** a linear or other structure

Allocation Methods

- There are three major methods of storing files on disks: contiguous, linked, and indexed.

Contiguous Allocation

- ***Contiguous Allocation*** requires that all blocks of a file be kept together contiguously.
- Performance is very fast, because reading successive blocks of the same file generally requires no movement of the disk heads, or at most one small step to the next adjacent cylinder.

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

Figure - Contiguous allocation of disk space.

Linked Allocation

- Disk files can be stored as linked lists, with the expense of the storage space consumed by each link. (E.g. a block may be 508 bytes instead of 512.)
- Linked allocation involves no external fragmentation, does not require pre-known file sizes, and allows files to grow dynamically at any time.
- Another big problem with linked allocation is reliability if a pointer is lost or damaged. Doubly linked lists provide some protection, at the cost of additional overhead and wasted space.

Figure - Linked allocation of disk space.

Indexed Allocation

- ***Indexed Allocation*** combines all of the indexes for accessing each file into a common block (for that file), as opposed to spreading them all over the disk or storing them in a FAT table.

Figure - Indexed allocation of disk space.

Free-Space Management

- Another important aspect of disk management is keeping track of and allocating free space.

1. Bit Vector

- One simple approach is to use a ***bit vector***, in which each bit represents a disk block, set to 1 if free or 0 if allocated.
- Fast algorithms exist for quickly finding contiguous blocks of a given size

2. Linked List

- A linked list can also be used to keep track of all free blocks.

- Traversing the list and/or finding a contiguous block of a given size are not easy, but fortunately are not frequently needed operations. Generally the system just adds and removes single blocks from the beginning of the list.
- The FAT table keeps track of the free list as just one more linked list on the table.

Figure - Linked free-space list on disk.

Grouping

- A variation on linked list free lists is to use links of blocks of indices of free blocks. If a block holds up to N addresses, then the first block in the linked-list contains up to N-1 addresses of free blocks and a pointer to the next block of free addresses.

Counting

- When there are multiple contiguous blocks of free space then the system can keep track of the starting address of the group and the number of contiguous free blocks. As long as the average length of a contiguous group of free blocks is greater than two this offers a savings in space needed for the free list.

Unit 5 –part-3(I/O Systems)

Fig. 6.1. Structure of Input/Output System.

6.2 COMPONENTS OF INPUT/OUTPUT SYSTEM

(1) Input/Output Devices—

- **Input Device:** An input device is an electromechanical device that generates data for a Computer to read Such as Mouse, Keyboard, trackball etc.
- **Output Device:** An output device is also an electro-mechanical device that accepts data from a Computer Such as Printer, graphic display Screen etc.
- Many devices are both input and output devices are called **Input/Output devices,** such as Disk, tape, Writable CD etc.

Fig. 6.2. Device Controllers and I/O Devices.

(2) Devices Controller: A device Controller is a hardware Unit which is attached with the input/output bus of the computer and provides a hardware interface between the computer and the input/output devices. Some device controllers do not know how to execute programs of more than one Instruction. Then, they are called Input/Output processors (IOP) or sometimes channels as shown in Fig. 6.2

(3) Device Driver: A device driver is a software module, that controls various input/output devices through the device Controller.

The module device drivers knows the detail working of device controller such as address of the control register, the format of message, error codes etc. as given in Fig. 6.3.

(4) Input/Output Control System (IOCS): IOCS is used to deals with input/output related system calls *i.e.*, IOCS is the subset of rest of the OS. The two main components of this IOCS are:

(1) Input/Output traffic Controller.

(2) Input/Output Scheduling.

6.3 INPUT/OUTPUT FUNCTIONS

There are three techniques for performing input/output.

(1) Programmed Input/Output: When the processor is executing a program and encounters an instruction relating to input/output, it executes that instruction by issuing a command to the appropriate input/output module.

With programmed input/output, the input/output module will perform the required action and then set the appropriate bits in the input/output status register as shown in Fig. 6.4

Fig. 6.4. Programmed Input/Output.

Scanned by CamScanner

Fig. 6.6. DMA.

4. INPUT/OUTPUT BUFFERING

a Process issues input/output Command, is suspended awaiting the result, and then is swapped out prior to the beginning of the operation. The process is blocked waiting for the input/output event and the input/output operation is blocked waiting for the process to be swapped in.

The same situation may arise on output operation. If a block is being transferred from a User process area directly to an input/output module, then the process is blocked during the transfer and the process may not be swapped out.

To avoid these Overheads and inefficiencies, it is Convenient to perform input transfers in advance of request being made and to perform output transfer sometimes later, after the request has been made. This technique is known as **Buffering**.

6.4.1 Types of Input/Output Devices

1. **Block-Oriented Devcies:** The input/output devices which store information in blocks, that are Usually of fixed size, and transfers are made one block at a time. Generally, the data is referenced by their block number. Disks and tapes are examples of block-Oriented devices.
2. **Stream-Oriented Devices:** These devices transfer data in and out as a stream of bytes. They have no block structure. Printer, mouse and other devices etc are stream-Oriented.

6.4.2 Types of Buffering

1. **Single Buffering:** The simplest type of buffering supported by an operating system is called single buffering. When a User process issues an input/output request, the operating system assigns a buffer in the system that is a portion of main memory to the operation as given in Fig. 6.7.

Fig. 6.7. Single Buffering.

2. **Double Buffering:** An improvement over Single buffering Can be made by assigning two system buffers to the input/output operations. A process new transfers data to or from one buffer while the operating system empties or fills the other. This technique is known as double buffering or buffer swapping as given in Fig. 6.8.

Fig. 6.8. Double Buffering.

3. **Circular Buffering:** Double buffering may be inadequate if the process performs the rapid bursts of input/output. In this case the problem can be solved by using more than two buffers. When more than two buffers are used, the collection of buffers is itself referred as a circular buffer, with each individual buffer being one unit in the circular buffer as given in Fig. 6.9.

Fig. 6.9. Circular Buffering.

6.5 DISK STORAGE AND DISK SCHEDULING-MAGNETIC DISKS

6.5.1 Disk Storage/Input-Output Disk

Magnetic Disks are circular metal plate coated with magnetized material on both sides. Several disks are stacked to a spindle, one below the other called the platter with read/write head to make a disk Pack as shown in Fig. 6.10 and 6.11.

The Disk drive consists of a motor and all disks are rotate together at very high speed along cylinder.

Information are stored on the surface of a disk along Concentric Set of rings called tracks. These tracks are divided into sections called sectors. Let S bytes are stored per sector, P sector are there per track, t tracks per surface and m surfaces.

$$\text{Capacity} = m \times t \times P \times S$$

If d is diameter

6.15 FILE SYSTEM PROTECTION AND SECURITY

6.15.1 Protection

When information is kept in a Computer system, a major concern is its protection from both physical damage and improper access.

- **Protection from Physical Damage:** Reliability is generally provided by duplicate copies of file. many computers have system programs that automatically copy disk files to tape at regular intervals. File systems can be damaged by hardware problems, power failures, head crashes etc.
- **Protection From Improper Access:** On the systems that do not permit access to the files to other Users, protection is not needed. One extreme would be to provide complete protection by prohibiting access. Other extreme would be to provide free access with no protection. Protection mechanism provide controlled access by limiting the types of file access.

6.15.2 Security Issues

- File system contains information that is highly valuable to their users, therefore protecting this information is a major concern.
- The causes of data loss be any one of the following.
  1. **Uncontrolled Causes:** Floods, fire, earth Quakes etc.
  2. **Hardware and Software Errors:** Program bugs, CPU malfunctions etc.
  3. **Human Errors:** Wrong tape, wrong program runs etc.
- Another Problem is of intruders.
  1. **Active Intruder:** Who make Unauthorized Changes to data.
  2. **Passive Intruder:** They read files which are not authorized to read.
- To protect the system, we must take security measures at four levels.
  1. **Physical:** Must be physically secured so that the intruders can be prohibited to enter the site.
  2. **Human:** Even the permitted Users should be carefully screened, if a security system fails, the protection provided by OS will become almost Useless.
  3. **Network:** The interruption of the communication network could be a remote denial-of-service attack and reduces the trust of the system.
  4. **Operating System:** The system must be protected itself from accidental or purposeful security breaches.

6.15.3 Authentication

User authentication of a system is a major issues associated with an operating system. It authenticate a user who tries to log-on the system. Authentication is based on—

1. **Passwords:** Most common authentication mechanism which is based on Sharing of a Secret.
2. **Artifact Based Authentication:** The artifacts commonly used for user authentication. Card readers are installed in the computers and [illegible] supply the artifacts for authentication.

3. **Biometrics:** This authentication mechanism is based on Unique characteristics of User. These are–

- Physiological Characteristics such as fingerprints, facial characterstics, hand geometry etc.
- Behavioural characterstics such as Signature, voice Pattern etc.

6.15.4 Program Threats

When a program written by one user, may be used by another user, misuse and unexpected behaviour may occurs are Trojan horse, trap doors, stack and buffes overflow etc.

6.15.5 System Threats

OS resources and User files are misused. The two known methods for the cause of misuse are worms and viruses.

- **Worms:** Worms are basically the programs that replicate themselves from system to system with the Use of host file.
- **Viruses:** A computer Virus is a small program written to alter the way a computer operates and it executes without the permission or knowledge of the User. A program is a virus if it execute and replicate by itself.

The existence of virus encompasses four stages

1. Dormant
2. Propagate
3. Triggering
4. Damage

6.15.6 Threat Monitoring

It is a technique through which security of the system can be improved. **Firewall** is a technique that can be used for threat monitoring.