UNIT-1

What is an Operating System?

Operating System: Operating System is an interface or mediater between user and computer system(Hardware).

- An operating system is a program that manages the computer hardware. An operating system is an important part of almost every computer system.
- A computer system can be divided roughly into four components: the *hardware,* the *operating sys tem,* the *application programs,* and the *users.*

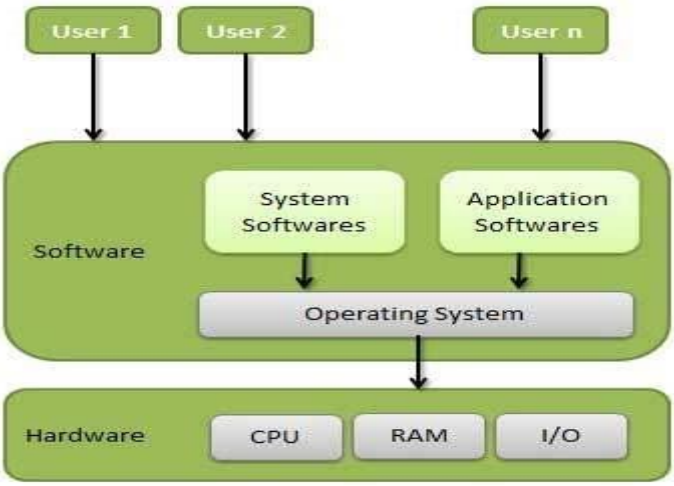


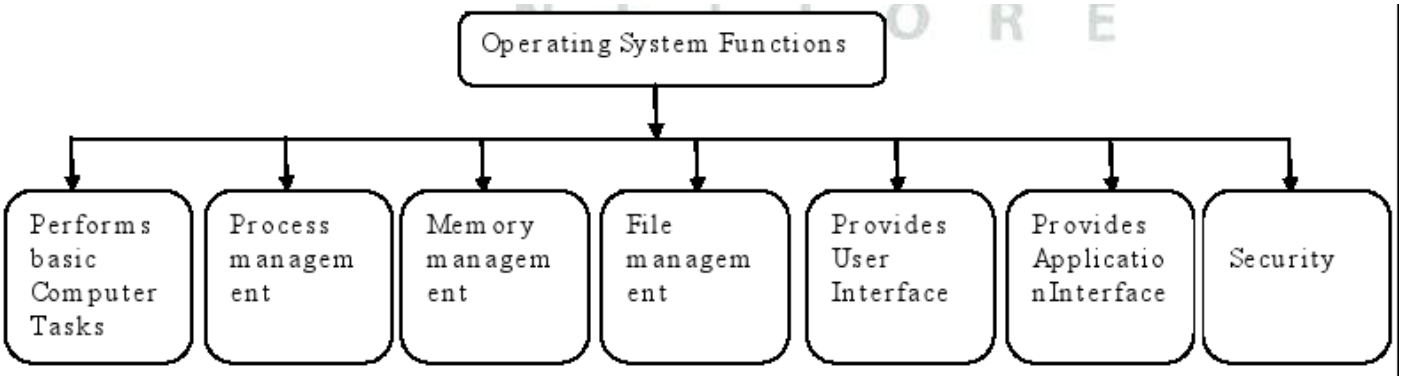
Fig: Computer System Architecture

- The **Hardware** includes the central processing unit **(CPU),** the memory, and the input output devices.
- The **Application programs** includes word processors, spreadsheets and web browsers. these provides solutions to user problems.
- The **Operating system** controls and coordinates the use of the hardware among the various application programs for the various users.

Objectives of OS

- **Convenience**: An OS makes a computer more convenient to use.
- **Efficiency:** An OS allows the computer system resources to be used in an efficient manner.
- **Ability to evolve:** An OS should be constructed in such a way as to permit the effective development, testing, and introduction of new system functions without interfering with service.

Functions of OS/Components:



Performs basic computer Tasks:

The operating System performs a computer s basic tasks such as managing peripheral devices: Keyboard, Mouse, Printer, and so on. The Operating system available today are based on the plug and play concept. i. e when a new device connected it will be automatically detected and configured without any user s interaction.

Process management:

A program in execution is called a process. Assignment of processors to different tasks being performed by the computer system.

Memory management:

A program to be executed must be loaded in the main memory along with its data. Allocation of main memory & other storage areas to the system programs as well as user programs and data.

File management:

All users store, retrieve, and work on the information stored in files. The operating System, therefore enables the users to create, copy, delete, move, and rename a file. The storage of files on various storage devices and the transfer of these files from one storage device to another.

Provides user interface:

The operating system enables users to easily interact with the computer hardware. For Example the windows operating system displays icons through which the users can interact with the system . In general, a user can interact with the computer system in two ways

i. **Command Line Interface(CLI).**
ii. **Graphical User Interface(GUI).**

Provides application Programming Interface.

The operating system provides a consistent application interface that ensure programmers that the application interface that ensures programmers that the applications developed on one computer will also run on other computers with the same or different hardware configuration.

Security:

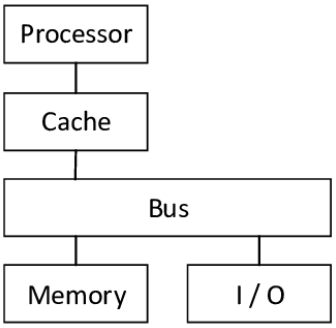
It keeps different programs and data in such a manner that they do not interfere with each other.

Computer-System Architecture:

A computer system may be organized in a number of different ways, which we can categorize roughly according to the number of general-purpose processors used.

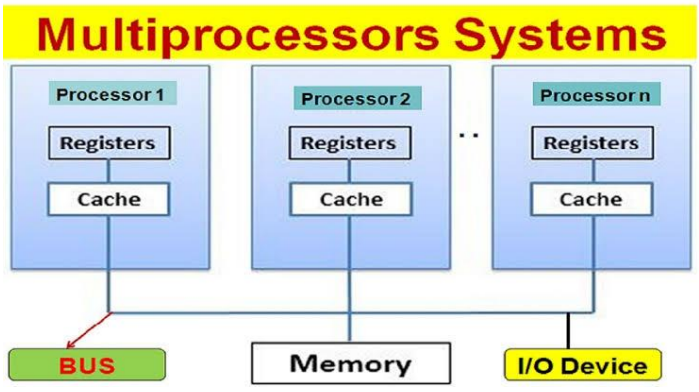
1.Single-Processor Systems:

On a single-processor system, there is one main CPU capable of executing a general-purpose instruction set, including instructions from user processes. Almost all systems have other special-purpose processors as well.



2.Multiprocessor Systems:

multiprocessor systems (also known as parallel systems or tightly coupled systems) are growing in importance. Such systems have two or more processors in close communication, sharing the computer bus and sometimes the clock, memory, and peripheral devices.



Multiprocessor systems have three main advantages:

a.Increased throughput: By increasing the number of processors, we expect to get more work done in less time. The speed-up ratio with N processors is not N,however; rather, it is less than N. When multiple processors cooperate on a task, a certain amount of overhead is incurred in keeping all the parts working correctly. This overhead, plus contention for shared resources, lowers the expected gain from additional processors.

b.Economy of scale: Multiprocessor systems can cost less than equivalent multiple single-processor systems, because they can share peripherals, mass storage, and power supplies.

c.Increased reliability: If functions can be distributed properly among several processors, then the failure of one processor will not halt the system, only slow it down.

Operating System Structure

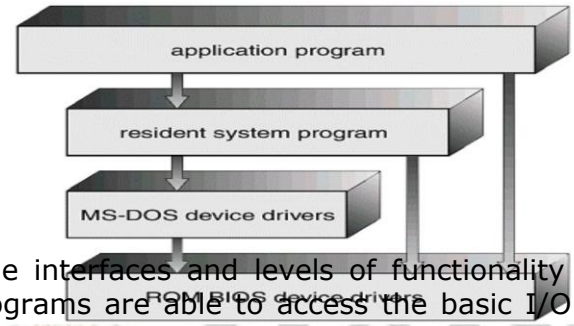
A system as large and complex as a modern operating system must be engineered carefully if it is to function properly and be modified easily.

Simple Structure:

Many operating systems do not have well-defined structures. Frequently, such systems started as small, simple, and limited systems and then grew beyond their original scope.

MS-DOS Layer Structure:

MS-DOS is an example of such a system. The following diagram shows the layer Structure of MS-DOS.



In MS-DOS, the interfaces and levels of functionality are not well separated. For instance, application programs are able to access the basic I/O routines to write directly to the display and disk drives. Designed to get maximal functionality with scarce memory.

MS-DOS was also limited by the hardware of its era. Because the Intel 8088 for which it was written provides no dual mode and no hardware protection, the designers of MS-DOS had no choice but to leave the base hardware accessible.

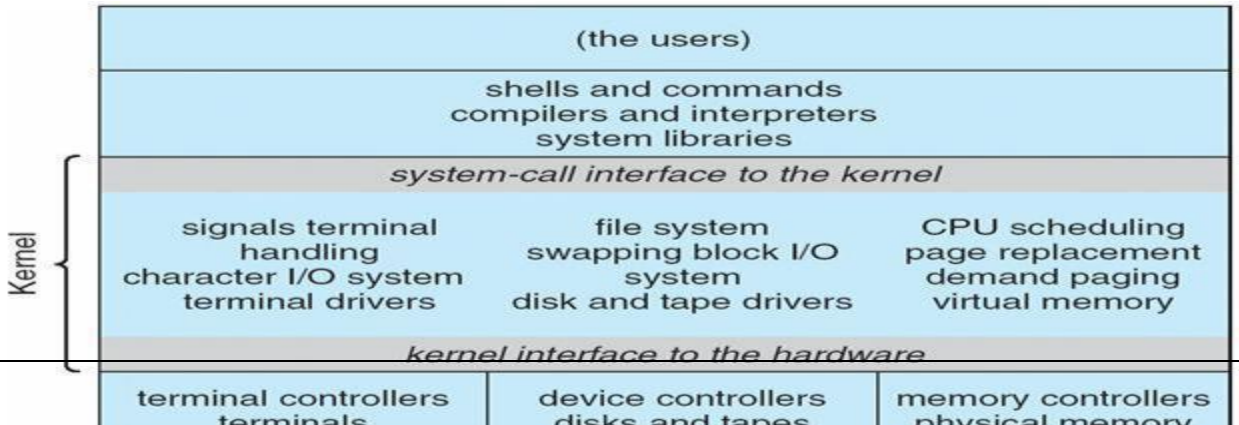
UNIX System Structure:

Another example of limited structuring is the original UNIX operating system. Like MS-DOS, UNIX initially was limited by hardware functionality.

It consists of two separable parts:

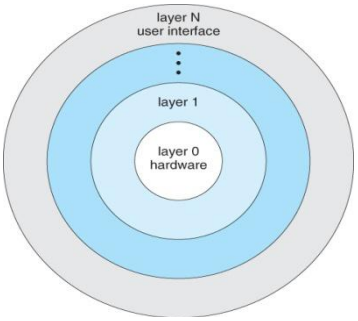
1. Kernel and
2. The system programs

Kernel: The kernel is further separated into a series of interfaces and device drivers, which have been added and expanded over the years as UNIX has evolved. The kernel provides the file system, CPU scheduling, memory management, and other operating-system functions through system calls. The following diagram shows the traditional UNIX operating system as being layered.



Layered Approach:

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.



Main advantage of layered approach:

- First layer can be debugged without any concern, because it uses only basic hardware.
- Once the first layer is debugged, its correct functioning while second layer is worked on and so on.
- If an error occur we know in which layer.
- Each layer is implemented using those operations provided by lower-level layers.
- A layer does not need to know how the low-level operations are implemented, it needs to know what these operations do.
- Each layer hides the existence of data structures, operations, and hardware from higher-level layer.

System Programs

Another aspect of a modern system is the collection of system programs. They canbe divided into these categories:

● **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. Such programs include web browsers, word processors and text formatters, spreadsheets, database systems, compilers, plotting and

statistical-analysis packages, and games. These programs are known as **system utilities** or **application programs.**

Operating System Services:

An Operating System provides services to both the users and to the programs.

- It provides programs, an environment to execute.
- It provides users, services to execute the programs in a convenient manner.

Following are few common services provided by operating systems.

- Program execution
- I/O operations
- File System manipulation
- Communication
- Error Detection
- Resource Allocation
- Protection

Program execution

- Operating system handles many kinds of activities from user programs to system programs like printer spooler, name servers, file server etc. Each of these activities is
- encapsulated as a process.
- A process includes the complete execution context (code to execute, data to manipulate, registers, OS resources in use). Following are the major activities of an operating system with respect to program management.

  - Loads a program into memory.
  - Executes the program.
  - Handles program's execution.
  - Provides a mechanism for process synchronization.
  - Provides a mechanism for process communication.
  - Provides a mechanism for deadlock handling.

I/O Operation:

I/O subsystem comprised of I/O devices and their corresponding driver software. Drivers hides the peculiarities of specific hardware devices from the user as the device driver knows the peculiarities of the specific device. Operating System manages the communication between user and device drivers. Following are the major activities of an operating system with respect to I/O Operation.

- I/O operation means read or write operation with any file or any specific I/O device.
- Program may require any I/O device while running.
- Operating system provides the access to the required I/O device when required.

File system manipulation

A file represents a collection of related information. Computer can store files on the disk (secondary storage), for long term storage purpose. A file system is normally organized into directories for easy navigation and usage. These directories may contain files and other directions. Following are the major activities of an operating system with respect to file management.

- Program needs to read a file or write a file.
- The operating system gives the permission to the program for operation on file.
- Permission varies from read-only, read-write, denied and so on.
- Operating System provides an interface to the user to create/delete files.
- Operating System provides an interface to the user to create/delete directories.
- Operating System provides an interface to create the backup of file system.

Communication:

In case of distributed systems which are a collection of processors that do not share memory, peripheral devices, or a clock, operating system manages communications between processes. Multiple processes with one another through communication lines in the network.

OS handles routing and connection strategies, and the problems of contention and security. Following are the major activities of an operating system with respect to communication.

- Two processes often require data to be transferred between them.
- The both processes can be on the one computer or on different computer but are connected through computer network.
- Communication may be implemented by two methods either by Shared Memory or by Message Passing.

Error handling

Error can occur anytime and anywhere. Error may occur in CPU, in I/O devices or in the memory hardware. Following are the major activities of an operating system with respect to error handling.

- OS constantly remains aware of possible errors.
- OS takes the appropriate action to ensure correct and consistent computing.

Resource Management

In case of multi-user or multi-tasking environment, resources such as main memory, CPU cycles and files storage are to be allocated to each user or job. Following are the major activities of an operating system with respect to resource management.

- OS manages all kind of resources using schedulers.
- CPU scheduling algorithms are used for better utilization of CPU.

Protection

Considering a computer systems having multiple users the concurrent execution of multiple processes, then the various processes must be protected from each another's activities. Protection refers to mechanism or a way to control the access of programs, processes, or users to the resources defined by a computer systems. Following are the major activities of an operating system with respect to protection.

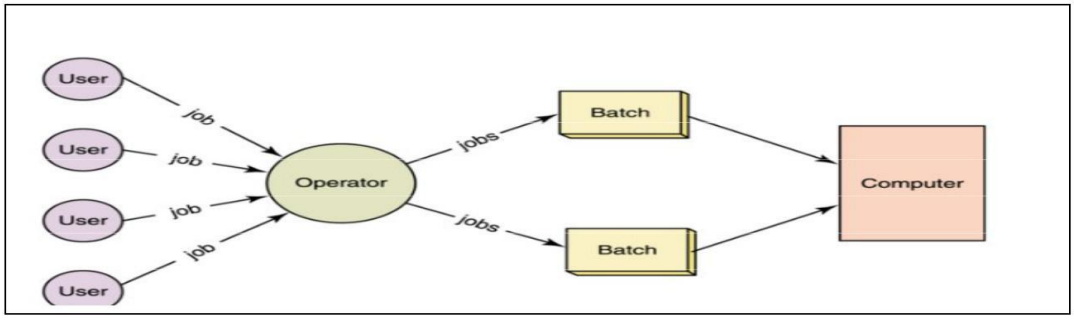
- OS ensures that all access to system resources is controlled.
- OS ensures that external I/O devices are protected from invalid access attempts.
- OS provides authentication feature for each user by means of a password.

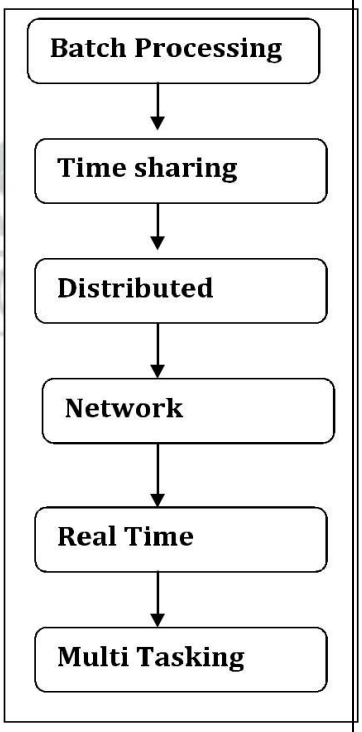
Evolution of Operating System (or) Different types of Operating System:

Operating systems are there from the very first computer generation. Operating systems keep evolving over the period of time. Following are few of the important types of operating system which are most commonly used.

Batch Processing operating system:

The users of batch operating system do not interact with the computer directly. Each user prepares his job on an off-line device like punch cards and submits it to the computer operator. The operator then sorts programs into batches with similar requirements.





The problems with Batch Systems are following.

- Lack of interaction between the user and job.
- CPU is often idle, because the speeds of the mechanical I/O devices is slower than CPU.
- Difficult to provide the desired priority**.**

Time Sharing Operating System:

Time sharing is a technique which enables many people, located at various terminals, to use a particular computer system at the same time. Time-sharing or multitasking is a logical extension of multiprogramming. Processor's time which is shared among multiple users simultaneously is termed as time-sharing.

The main difference between Multiprogrammed, Batch Systems and Time-Sharing Systems is that in case of Multiprogrammed and batch systems, objective is to maximize processor use, whereas in Time-Sharing Systems objective is to minimize response time**.**

Advantages:

- Provide advantages of quick response.
- Avoids duplication of Software.
- Reduces CPU idle time.

Disadvantages:

- It is very difficult and expensive to develop.
- It is hard to implement.

Distributed operating System

Distributed systems use multiple central processors to serve multiple real time application and multiple users. Data processing jobs are distributed among the processors accordingly to which one can perform each job most efficiently.

The processors communicate with one another through various communication lines (such as high -speed buses or telephone lines). These are referred as loosely coupled systems or distributed systems.

The advantages of distributed systems are following.

- With resource sharing facility user at one site may be able to use the resources available at another.
- Speedup the exchange of data with one another via electronic mail.
- If one site fails in a distributed system, the remaining sites can potentially continue operating.
- Better service to the customers.
- Reduction of the load on the host computer.
- Reduction of delays in data processing.

Network operating System:

Network Operating System runs on a server and provides server the capability to manage data, users, groups, security, applications, and other networking functions. The primary purpose of the network operating system is to allow shared file and printer access among multiple computers in a network, typically a local area network (LAN).

Examples of network operating systems are Microsoft Windows Server 2003, Microsoft Windows Server 2008, UNIX, Linux,

The advantages of network operating systems are following.

- ∑ Centralized servers are highly stable.
- ∑ Security is server managed.

The disadvantages of network operating systems are following.

- Dependency on a central location for most operations.
- High cost of buying and running a server
- Regular maintenance and updates are required

Real time Operating System:

Real-time operating system(RTOS) - Real-time operating systems are used to control machinery, scientific instruments and industrial systems. An RTOS typically has very little user-

interface capability, and no end-user utilities, since the system will be a "sealed box" when delivered for use.

Advantages:

- It is easy to design and.
- It offers maximum consumption of the system.
- It relatively requires less memory space.
- Upgrades to new technologies and hardwares can be easily integrated into the system.
- Remote access to servers is possible from different locations and types of systems.

There are two types of real-time operating systems.

Hard real-time systems

Hard real-time systems guarantee that critical tasks complete on time. In hard real-time systems secondary storage is limited or missing with data stored in ROM. In these systems virtual memory is almost never found.

Soft real-time systems

Soft real time systems are less restrictive. Critical real-time task gets priority over other tasks and retains the priority until it completes. Soft real-time systems have limited utility than hard real-time systems.

For example, Multimedia, virtual reality, and Advanced Scientific Projects like undersea exploration

Multi Tasking:

A multi-user operating system allows many different users to take advantage of the computer's resources simultaneously. Unix, VMS and mainframe operating systems, such as *MVS*, are examples of multi-user operating systems.

Advantages:

- It makes better use of resources.
- It reduces response time.

Disadvantages:

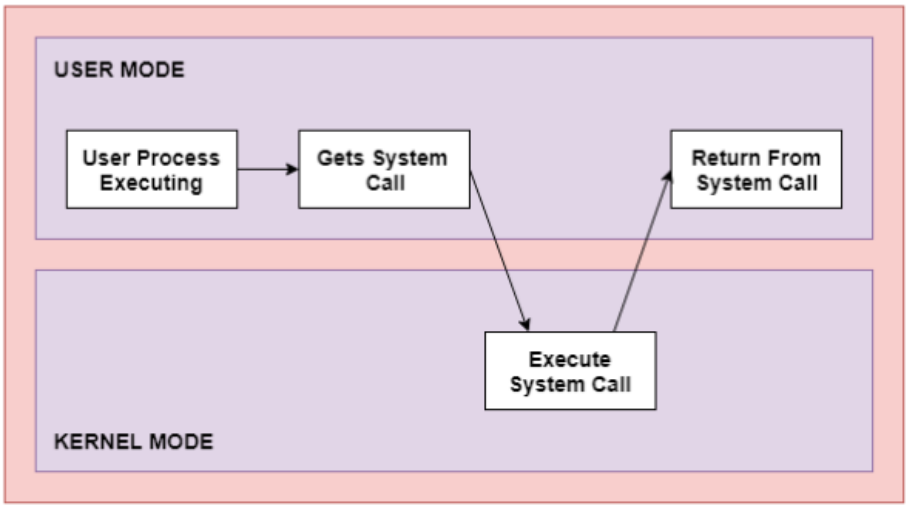
- It has complex configuration.
- It is difficult to handle and maintain.
- It requires a lot of memory to process.

System calls

The interface between a process and an operating system is provided by system calls.

In general, system calls are available as assembly language instructions. They are also included in the manuals used by the assembly level programmers. System calls are usually made when a process in user mode requires access to a resource. Then it requests the kernel to provide the resource via a system call.

A figure representing the execution of the system call is given as follows –



As can be seen from this diagram, the processes execute normally in the user mode until a system call interrupts this. Then the system call is executed on a priority basis in the kernel

mode. After the execution of the system call, the control returns to the user mode and execution of user processes can be resumed.

In general, system calls are required in the following situations −

- If a file system requires the creation or deletion of files. Reading and writing from files also require a system call.
- Creation and management of new processes.
- Network connections also require system calls. This includes sending and receiving packets.
- Access to a hardware devices such as a printer, scanner etc. requires a system call.

Types of System Calls

There are mainly five types of system calls. These are explained in detail as follows −

Process Control

These system calls deal with processes such as process creation, process termination etc.

File Management

These system calls are responsible for file manipulation such as creating a file, reading a file, writing into a file etc.

Device Management

These system calls are responsible for device manipulation such as reading from device buffers, writing into device buffers etc.

Information Maintenance

These system calls handle information and its transfer between the operating system and the user program.

Communication

These system calls are useful for interprocess communication. They also deal with creating and deleting a communication connection.

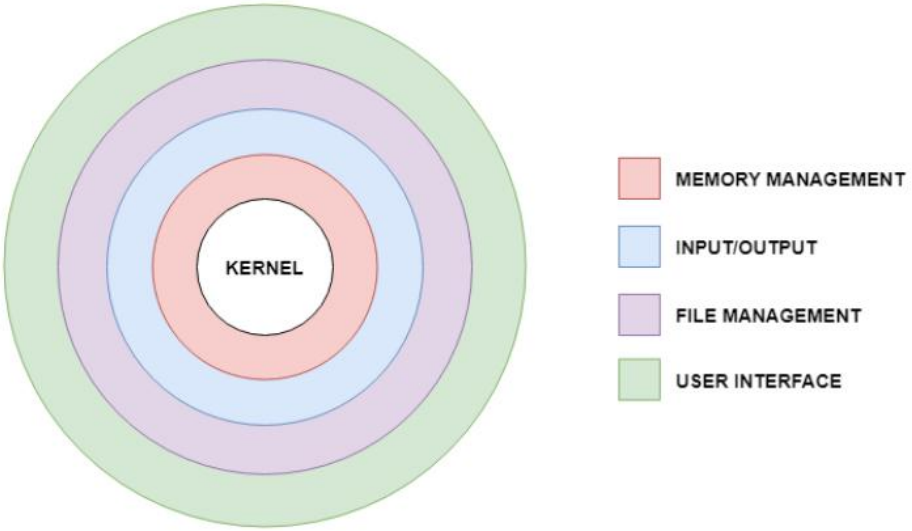
Some of the examples of all the above types of system calls in Windows and Unix are given as follows −

Types of System Calls	Windows	Linux
Process Control	CreateProcess() ExitProcess() WaitForSingleObject()	fork() exit() wait()
File Management	CreateFile() ReadFile() WriteFile() CloseHandle()	open() read() write() close()
Device Management	SetConsoleMode() ReadConsole() WriteConsole()	ioctl() read() write()
Information Maintenance	GetCurrentProcessID() SetTimer() Sleep()	getpid() alarm() sleep()
Communication	CreatePipe() CreateFileMapping() MapViewOfFile()	pipe() shmget() mmap()

Design and Implementation of OS:

An operating system is a construct that allows the user application programs to interact with the system hardware. Operating system by itself does not provide any function but it provides an atmosphere in which different applications and programs can do useful work.

There are many problems that can occur while designing and implementing an operating system. These are covered in operating system design and implementation.



Layered Operating System Design

Operating System Design Goals

It is quite complicated to define all the goals and specifications of the operating system while designing it.The design changes depending on the type of the operating system i.e if it is batch system, time shared system, single user system, multi user system, distributed system etc.

There are basically two types of goals while designing an operating system. These are −

User Goals

The operating system should be convenient, easy to use, reliable, safe and fast according to the users. However, these specifications are not very useful as there is no set method to achieve these goals.

System Goals

The operating system should be easy to design, implement and maintain. These are specifications required by those who create, maintain and operate the operating system. But there is not specific method to achieve these goals as well.

Operating System Mechanisms and Policies

There is no specific way to design an operating system as it is a highly creative task. However, there are general software principles that are applicable to all operating systems.

A subtle difference between mechanism and policy is that mechanism shows how to do something and policy shows what to do. Policies may change over time and this would lead to changes in mechanism. So, it is better to have a general mechanism that would require few changes even when a policy change occurs.

For example - If the mechanism and policy are independent, then few changes are required in mechanism if policy changes. If a policy favours I/O intensive processes over CPU intensive processes, then a policy change to preference of CPU intensive processes will not change the mechanism.

Operating System Implementation

The operating system needs to be implemented after it is designed. Earlier they were written in assembly language but now higher level languages are used. The first system not written in assembly language was the Master Control Program (MCP) for Burroughs Computers.

Advantages of Higher Level Language

There are multiple advantages to implementing an operating system using a higher level language such as: the code is written more fast, it is compact and also easier to debug and understand. Also, the operating system can be easily moved from one hardware to another if it is written in a high level language.

Disadvantages of Higher Level Language

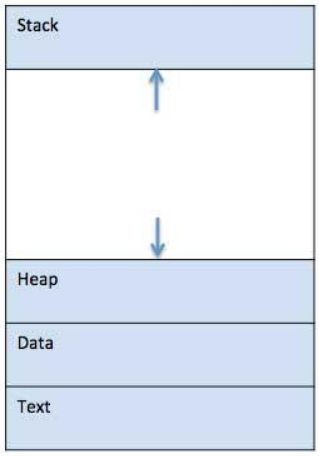
Using high level language for implementing an operating system leads to a loss in speed and increase in storage requirements. However in modern systems only a small amount of code is needed for high performance, such as the CPU scheduler and memory manager. Also, the bottleneck routines in the system can be replaced by assembly language equivalents if required.

Process Concept:

The Process:

A process is a program in execution. The execution of a process must progress in a sequential fashion. A process is defined as an entity which represents the basic unit of work to be implemented in the system. When a program is loaded into the memory and it becomes a process, it can be divided into four sections — stack, heap, text and data.

The following image shows a simplified layout of a process inside main memory –



Stack:

The process Stack contains the temporary data such as method/function parameters, return address and local variables.

Heap:

This is dynamically allocated memory to a process during its run time

Text: This includes the current activity represented by the value of Program Counter and the contents of the processor's **registers**.

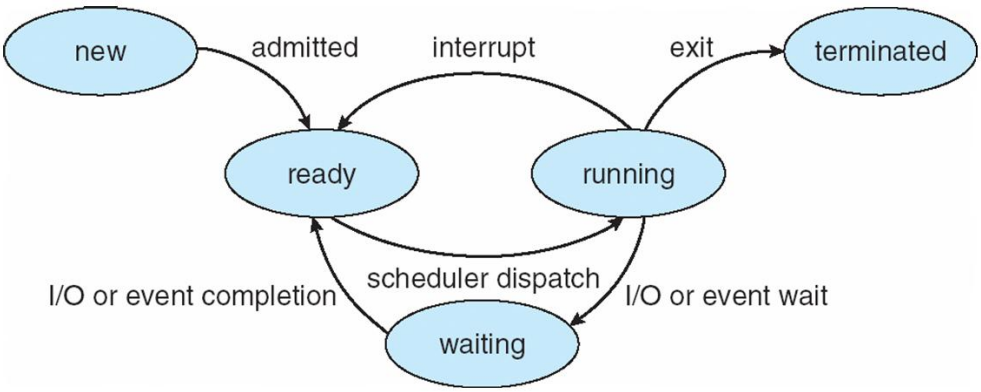
Data:

This section contains the global and static variables.

Process State(or) Process Lifecycle:

When a process executes, it passes through different states. These stages may differ in different operating systems, and the names of these states are also not standardized.

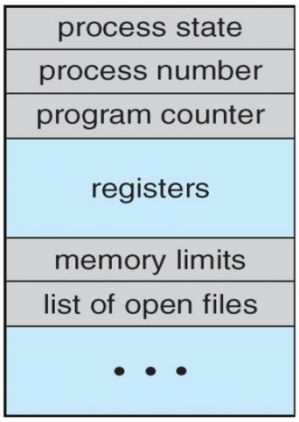
In general, a process can have one of the following five states at a time.



Start(New): This is the initial state when a process is first started/created.
Running. Instructions are being executed.
Waiting. The process is waiting for some event to occur (such as an I/O completion or reception of a signal).
Ready. The process is waiting to be assigned to a processor.
Terminated. The process has finished execution.

Process Control Block:

Each process is represented in the operating system by a **process control block (PCB)**—also called a *task control block.* It contains many pieces of information associated with a specific process, including these:



Process State:
The current state of the process i.e., whether it is ready, running, waiting, or whatever.
Process ID/number:
Unique identification for each of the process in the operating system.
Program Counter:
Program Counter is a pointer to the address of the next instruction to be executed for this process.
CPU registers:
Various CPU registers where process need to be stored for execution for running state.
CPU Scheduling Information:

Process priority and other scheduling information which is required to schedule the process

Memory management information:

This includes the information of page table, memory limits, Segment table depending on memory used by the operating system.

Accounting information:

This includes the amount of CPU used for process execution, time limits, execution ID etc.

IO status information:

This includes a list of I/O devices allocated to the process.

Interprocess communication

Interprocess communication is the mechanism provided by the operating system that allows processes to communicate with each other. This communication could involve a process letting another process know that some event has occurred or the transferring of data from one process to another.

A diagram that illustrates interprocess communication is as follows −

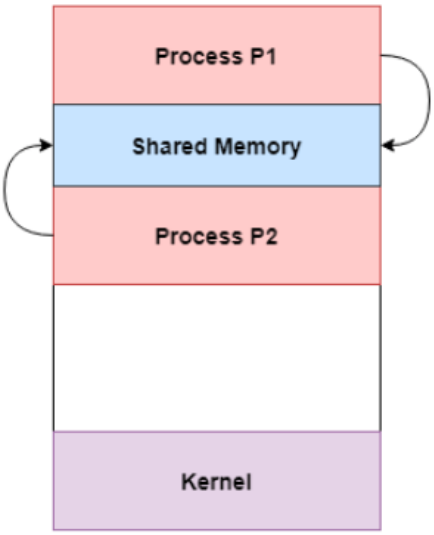
Approaches to Interprocess Communication

The different approaches to implement interprocess communication are given as follows −

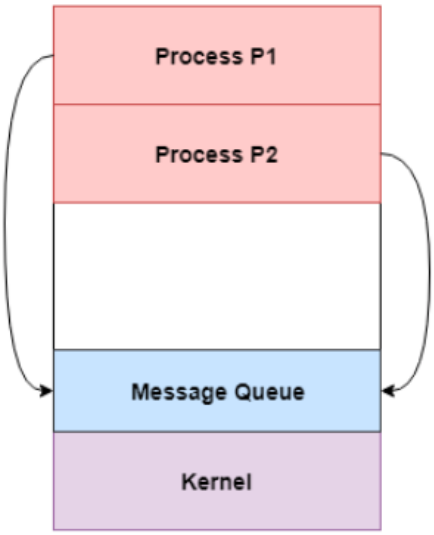
- **Pipe**
  A pipe is a data channel that is unidirectional. Two pipes can be used to create a two-way data channel between two processes. This uses standard input and output methods. Pipes are used in all POSIX systems as well as Windows operating systems.
- **Socket**
  The socket is the endpoint for sending or receiving data in a network. This is true for data sent between processes on the same computer or data sent between different computers on the same network. Most of the operating systems use sockets for interprocess communication.
- **File**
  A file is a data record that may be stored on a disk or acquired on demand by a file server. Multiple processes can access a file as required. All operating systems use files for data storage.
- **Signal**
  Signals are useful in interprocess communication in a limited way. They are system messages that are sent from one process to another. Normally, signals are not used to transfer data but are used for remote commands between processes.
- **Shared Memory**
  Shared memory is the memory that can be simultaneously accessed by multiple processes. This is done so that the processes can communicate with each other. All POSIX systems, as well as Windows operating systems use shared memory.
- **Message Queue**
  Multiple processes can read and write data to the message queue without being connected to each other. Messages are stored in the queue until their recipient retrieves them. Message queues are quite useful for interprocess communication and are used by most operating systems.

A diagram that demonstrates message queue and shared memory methods of interprocess communication is as follows

Approaches to Interprocess Communication



Shared Memory



Message Queue

UNIT-2
CPU SCHEDULING

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

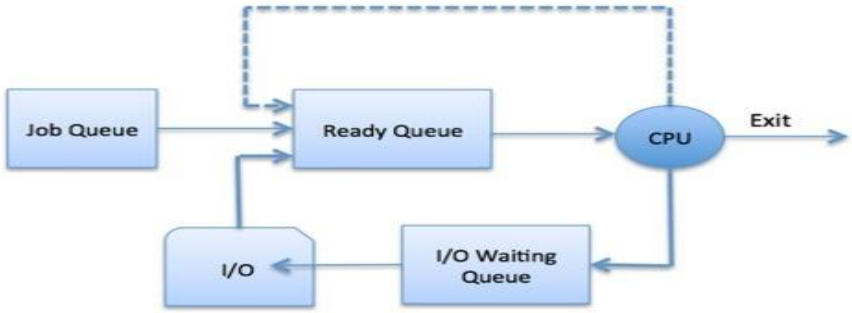
Process scheduling is an essential part of a Multiprogramming operating systems. Such operating systems allow more than one process to be loaded into the executable memory at a time and the loaded process shares the CPU using time multiplexing.

Process Scheduling Queues

The OS maintains all PCBs in Process Scheduling Queues. The OS maintains a separate queue for each of the process states and PCBs of all processes in the same execution state are placed in the same queue.

The Operating System maintains the following important process scheduling queues –

- **Job queue** – This queue keeps all the processes in the system.
- **Ready queue** – This queue keeps a set of all processes residing in main memory, ready and waiting to execute. A new process is always put in this queue.
- **Device queues** – The processes which are blocked due to unavailability of an I/O device constitute this queue.



The OS scheduler determines how to move processes between the ready and run queues which can only have one entry per processor core on the system.

Two-State Process Model:

Running: When a new process is created, it enters into the system as in the running state.

Not Running: Processes that are not running are kept in queue, waiting for their turn to execute. Each entry in the queue is a pointer to a particular process. When a process is interrupted, that process is transferred in the waiting queue. If the process has completed or aborted, the process is discarded.

5.Schedulers:

Schedulers are special system software which handle process scheduling in various ways. Their main task is to select the jobs to be submitted into the system and to decide which process to run. Schedulers are of three types –

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

Long Term Scheduler:

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

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

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

Short Term Scheduler:

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

Short-term schedulers, also known as dispatchers, make the decision of which process to execute next. Short-term schedulers are faster than long-term schedulers.

Medium Term Scheduler

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

CPU Scheduling Algorithms

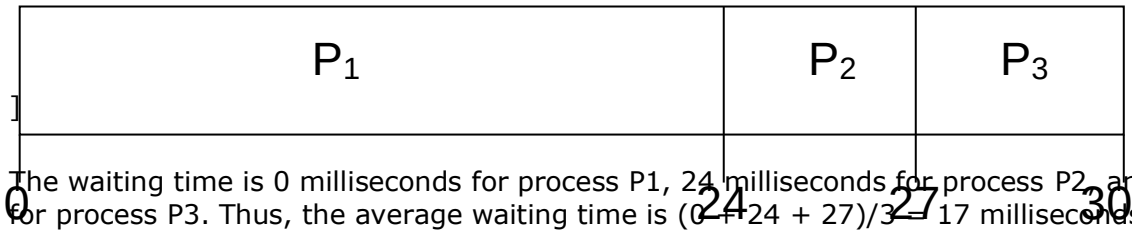
CPU scheduling deals with the problem of deciding which of the processes in the ready queue is to be allocated the CPU. There are many different CPU scheduling algorithms. In this section, we describe several of them.

First-Come, First-Served Scheduling(FCFS)

By far the simplest CPU-scheduling algorithm is the **first-come, first-served (FCFS) scheduling algorithm.** With this scheme, the process that requests the CPU first is allocated the CPU first.

Process	Burst Time
P_1	24
P_2	3
P_3	3

Suppose that the processes arrive in the order: P_1 , P_2 , P_3The Gantt Chart for the schedule is:



The waiting time is 0 milliseconds for process P1, 24 milliseconds for process P2, and 27 milliseconds for process P3. Thus, the average waiting time is (0 + 24 + 27)/3 = 17 milliseconds.

The average waiting time under an FCFS policy is generally not minimal and may vary substantially if the process's CPU burst times vary greatly.

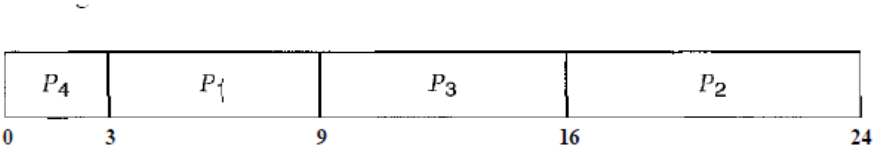
The FCFS scheduling algorithm is non preemptive. Once the CPU has been allocated to a process, that process keeps the CPU until it releases the CPU, either by terminating or by requesting I/O.

Shortest-Job-First Scheduling(SJF)

A different approach to CPU scheduling is the **shortest-job-first (SJF) scheduling algorithm. In** This algorithm the cpu assigns to the process which is having smallest burst time .If CPU bursts of two processes are the same, FCFS scheduling is used to break the tie. As an example of SJF scheduling, consider the following set of processes, with the length of the CPU burst given in milliseconds:

Process	Burst Time
P1	**6**
P2	**8**
P3	**7**
P4	**3**

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



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

Round-Robin Scheduling(RR)

The **round-robin (RR) scheduling algorithm** is designed especially for timesharing systems. It is similar to FCFS scheduling, but preemption is added to switch between processes. A small unit of time, called a **time quantum** or time slice, is defined. A time quantum is generally from 10 to 100 milliseconds. Consider the following set of processes that arrive at time 0, with the length of the CPU burst given in milliseconds:

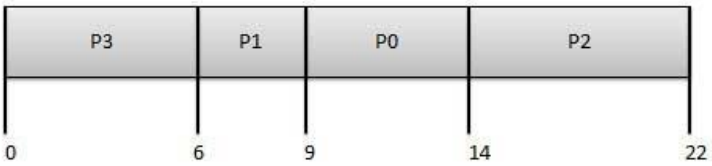
Process	Burst Time
P1	**24**
P2	3
P3	3

If we use a time quantum of 4 milliseconds.

Priority Based Scheduling

- Priority scheduling is a non-preemptive algorithm and one of the most common scheduling algorithms in batch systems.
- Each process is assigned a priority. Process with highest priority is to be executed first and so on.
- Processes with same priority are executed on first come first served basis.
- Priority can be decided based on memory requirements, time requirements or any other resource requirement.

Process	Arrival Time	Execute Time	Priority	Service Time
P0	0	5	1	9
P1	1	3	2	6
P2	2	8	1	14
P3	3	6	3	0



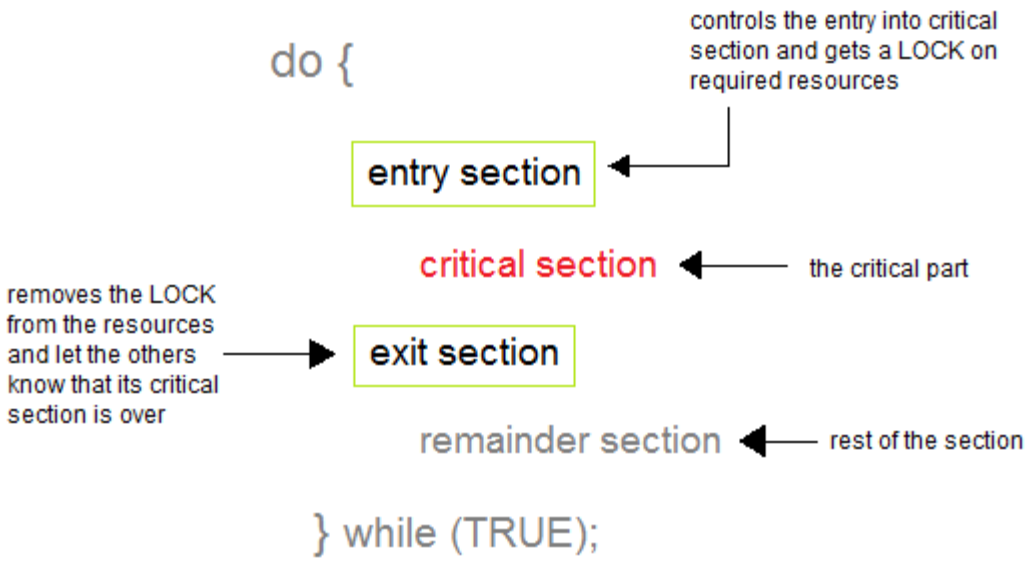
PROCESS SYNCHRONIZATION

Process Synchronization means sharing system resources by processes in a such a way that, Concurrent access to shared data is handled thereby minimizing the chance of inconsistent data. Maintaining data consistency demands mechanisms to ensure synchronized execution of cooperating processes.

Process Synchronization was introduced to handle problems that arose while multiple process executions. Some of the problems are discussed below.

Critical Section Problem

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



Solution to Critical Section Problem

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

1. **Mutual Exclusion**

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

2. **Progress**

   If no process is in its critical section, and if one or more threads want to execute their critical section then any one of these threads must be allowed to get into its critical section.

3. **Bounded Waiting**

   After a process makes a request for getting into its critical section, there is a limit for how many other processes can get into their critical section, before this process's request is granted. So after the limit is reached, system must grant the process permission to get into its critical section.

Synchronization Hardware

Many systems provide hardware support for critical section code. The critical section problem could be solved easily in a single-processor environment if we could disallow interrupts to occur while a shared variable or resource is being modified.

In this manner, we could be sure that the current sequence of instructions would be allowed to execute in order without pre-emption. Unfortunately, this solution is not feasible in a multiprocessor environment.

Disabling interrupt on a multiprocessor environment can be time consuming as the message is passed to all the processors.

This message transmission lag, delays entry of threads into critical section and the system efficiency decreases.

Mutex Locks

As the synchronization hardware solution is not easy to implement for everyone, a strict software approach called Mutex Locks was introduced. In this approach, in the entry section of code, a LOCK is acquired over the critical resources modified and used inside critical section, and in the exit section that LOCK is released.

As the resource is locked while a process executes its critical section hence no other process can access it.

Semaphores

In 1965, Dijkstra proposed a new and very significant technique for managing concurrent processes by using the value of a simple integer variable to synchronize the progress of interacting processes. This integer variable is called **semaphore**. So it is basically a synchronizing tool and is accessed only through two low standard atomic operations, wait and signal designated by P() and V() respectively.

The classical definition of wait and signal are :

- Wait : decrement the value of its argument S as soon as it would become non-negative.
- Signal : increment the value of its argument, S as an individual operation.

Properties of Semaphores

1. Simple
2. Works with many processes
3. Can have many different critical sections with different semaphores
4. Each critical section has unique access semaphores
5. Can permit multiple processes into the critical section at once, if desirable.

Types of Semaphores

Semaphores are mainly of two types:

1. **Binary Semaphore**

   It is a special form of semaphore used for implementing mutual exclusion, hence it is often called Mutex. A binary semaphore is initialized to 1 and only takes the value 0 and 1 during execution of a program.

2. **Counting Semaphores**

   These are used to implement bounded concurrency.

Limitations of Semaphores

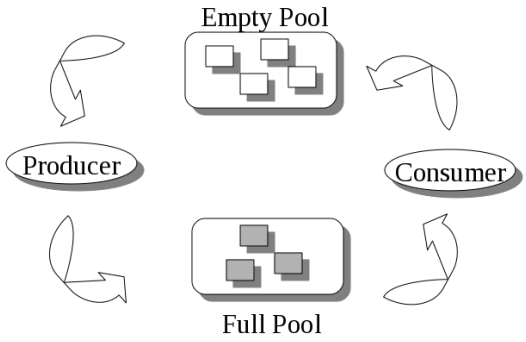
1. Priority Inversion is a big limitation of semaphores.
2. Their use is not enforced, but is by convention only.
3. With improper use, a process may block indefinitely. Such a situation is called Deadlock. We will be studying deadlocks in details in coming lessons.

Classical Problem of Synchronization

Following are some of the classical problem faced while process synchronaization in systems where cooperating processes are present.

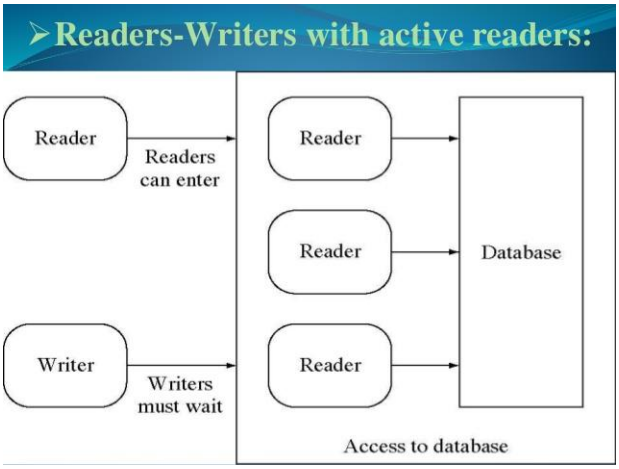
Bounded Buffer Problem

- This problem is generalised in terms of the Producer-Consumer problem.
- Solution to this problem is, creating two counting semaphores "full" and "empty" to keep track of the current number of full and empty buffers respectively.



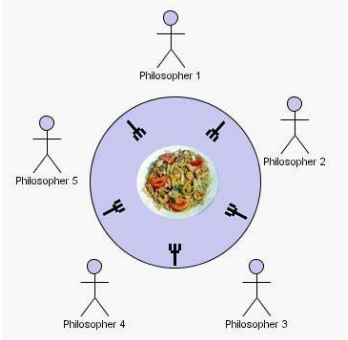
The Readers Writers Problem

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



Dining Philosophers Problem

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



Monitors in Operating System

Monitors are used for process synchronization. With the help of programming languages, we can use a monitor to achieve mutual exclusion among the processes. **Example of monitors:** ***Java***

Synchronized methods such as Java offers notify() and wait() constructs.
In other words, monitors are defined as the construct of programming language, which helps in controlling shared data access.

The Monitor is a module or package which encapsulates shared data structure, procedures, and the synchronization between the concurrent procedure invocations.

Characteristics of Monitors.

1. Inside the monitors, we can only execute one process at a time.
2.Monitors are the group of procedures, and condition variables that are merged together in a special type of module.
3. If the process is running outside the monitor, then it cannot access the monitor's internal variable. But a process can call the procedures of the monitor.
4. Monitors offer high-level of synchronization
5. Monitors were derived to simplify the complexity of synchronization problems.
6. There is only one process that can be active at a time inside the monitor.

Components of Monitor

There are four main components of the monitor:

1. Initialization
2. Private data
3. Monitor procedure
4. Monitor entry queue

Initialization: – Initialization comprises the code, and when the monitors are created, we use this code exactly once.
Private Data: – Private data is another component of the monitor. It comprises all the private data, and the private data contains private procedures that can only be used within the monitor. So, outside the monitor, private data is not visible.
Monitor Procedure: – Monitors Procedures are those procedures that can be called from outside the monitor.
Monitor Entry Queue: – Monitor entry queue is another essential component of the monitor that includes all the threads, which are called procedures.

Syntax of monitor

```
Monitor Demo // Name of the Monitor
{
variables;
condition variables;
procedure p1 {.....}
procedure p2 {.....}
}
```

Condition Variables

There are two types of operations that we can perform on the condition variables of the monitor:

1. Wait
2. Signal

Difference between Monitors and Semaphore

Monitors	Semaphore
1.We can use condition variables only in the monitors.	1.In semaphore, we can use condition variables anywhere in the program, but we cannot use conditions variables in a semaphore.
2.In monitors, wait always block the caller.	2.In semaphore, wait does not always block the caller.

3.The monitors are comprised of the shared variables and the procedures which operate the shared variable.

3.The semaphore S value means the number of shared resources that are present in the system.

Deadlocks

Deadlock: In a multiprogramming environment, several processes may compete for a finite number of resources. **A** process requests resources; if the resources are not available at that time, the process enters a wait state. Waiting processes may never again change state, because the resources they have requested are held by other waiting processes. This situation is called a **deadlock**

Deadlock Characterization

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

1. **Mutual exclusion**: At least one resource must be held in a nonsharable mode; that is, only one process at a time can use the resource. If another process requests that resource, the requesting process must be delayed until the resource has been released.

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

3. No preemption: Resources cannot be preempted; that is, a resource can be released only voluntarily by the process holding it, after that process has completed its task.

4**. Circular wait**: A set {Po, PI, ..., P,) of waiting processes must exist such that Po is waiting for a resource that is held by P1, P1 is waiting for a resource that is held by P2, ..., Pn-1 is waiting for a resource that is held by Pn and Pn is waiting for a resource that is held by Po.

Resource-Allocation Graph

Deadlocks can be described more precisely in terms of a directed graph called a system resource-allocation graph. This graph consists of a set of vertices V and a set of edges E. The set of vertices V is partitioned into two different types of nodes P = {PI, P2, ..., Pn}, the set consisting of all the active processes in the system, and R = {R1, R2, ..., Rm), the set consisting of all resource types in the system.

A directed edge from process Pi to resource type *Rj* is denoted by Pi →Rj;

it signifies that process ***Pi*** requested an instance of resource type Ri and is currently waiting for that resource. **A** directed edge from resource type Ri to process Pi is denoted by Rj → Pi; it signifies that an instance of resource type Ri has been allocated to process Pi. **A** directed edge Pi → Rj is called a request edge; a directed edge Rj → Pi is called an assignment edge.

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

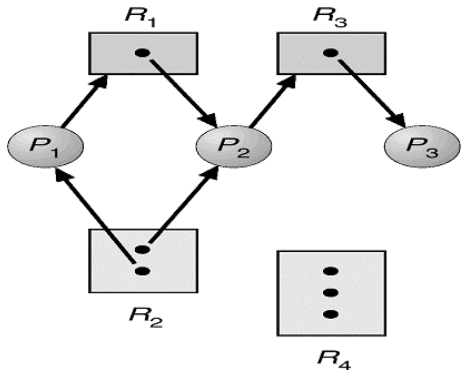


Figure 8.1 Resource-allocation graph.

Methods for Handling Deadlocks:

Principally, we can deal with the deadlock problem in one of three ways:

- We can use a protocol to prevent or avoid deadlocks, ensuring that the system will never enter a deadlock state.
- We can allow the system to enter a deadlock state, detect it, and recover
- We can ignore the problem altogether, and pretend that deadlocks never occur in the system. This solution is used by most operating systems, including UNIX.

To ensure that deadlocks never occur, the system can use either a deadlockprevention or a deadlock-avoidance scheme. **Deadlock prevention** is a set of methods for ensuring that at least one of the necessary conditions cannot hold.

Deadlock avoidance, on the other hand, requires that the operating system be given in advance additional information concerning which resources a process will request and use during its lifetime. With this additional knowledge, we can decide for each request whether or not the process should wait.

Deadlock Prevention:

for a deadlock to occur, each of the four necessary conditions must hold. By ensuring that at least one of these conditions cannot hold, we can prevent the occurrence of a deadlock. Let us elaborate on this approach by examining each of the four necessary conditions separately.

Mutual Exclusion

The mutual exclusion condition must hold for non sharable resources (printer). Sharable resources do not require mutually exclusive access and thus cannot be involved in a deadlock (read only file). We cannot prevent deadlocks by denying the mutual exclusion condition because some resources are intrinsically non sharable.

Hold and Wait

To ensure that the hold and wait condition never occurs in the system, we must guarantee that whenever a process requests a resource, it does not hold any other resources.

One protocol that can be used requires each process to request and be allocated all its resources before it begins execution.

No Preemption

The third necessary condition for deadlocks is that there be no preemption of resources that have already been allocated. To ensure that this condition does not hold, use the following protocol.

If a process is holding some resources and requests another resource that cannot be immediately allocated to it, then all resources currently being held are preempted i.e. these resources are implicitly released. The preempted resources are added to the list of resources for which the process is waiting. The process will be restarted only when it can regain its old resources as well as the new ones that it is requesting.

Circular Wait

The fourth and final condition for deadlocks is the circular wait condition. One way to ensure that this condition never holds is to impose a total ordering of all resource types and to require that each process requests resources in an increasing order of enumeration

Deadlock Avoidance

Deadlock prevention algorithms prevent deadlocks by restraining how requests can be made. The restraints ensure that at least one of the necessary conditions for deadlock cannot occur and hence that deadlocks cannot hold. Possible side effects of preventing deadlocks by this method are low device utilization and reduced system through put.

Banker's Algorithm in Operating System

It is a banker algorithm used to **avoid deadlock** and **allocate resources** safely to each process in the computer system. The '**S-State'** examines all possible tests or activities before deciding whether the allocation should be allowed to each process.

Suppose the number of account holders in a particular bank is 'n', and the total money in a bank is 'T'. If an account holder applies for a loan; first, the bank subtracts the loan amount from full cash and then estimates the cash difference is greater than T to approve the loan amount.

Similarly, it works in an **operating system**. When a new process is created in a computer system, the process must provide all types of information to the operating system like upcoming processes, requests for their resources, counting them, and delays. Based on these criteria, the operating system decides which process sequence should be executed or waited so that no deadlock occurs in a system. Therefore, it is also known as **deadlock avoidance algorithm** or **deadlock detection** in the operating system.

When working with a banker's algorithm, it requests to know about three things:

1. How much each process can request for each resource in the system. It is denoted by the [**MAX**] request.
2. How much each process is currently holding each resource in a system. It is denoted by the [**ALLOCATED**] resource.
3. It represents the number of each resource currently available in the system. It is denoted by the [**AVAILABLE**] resource.

Following are the important data structures terms applied in the banker's algorithm as follows:

1. **Available**: It is an array of length 'm' that defines each type of resource available in the system. When Available[j] = K, means that 'K' instances of Resources type R[j] are available in the system.
2. **Max:** It is a [n x m] matrix that indicates each process P[i] can store the maximum number of resources R[j] (each type) in a system.
3. **Allocation:** It is a matrix of m x n orders that indicates the type of resources currently allocated to each process in the system. When Allocation [i, j] = K, it means that process P[i] is currently allocated K instances of Resources type R[j] in the system.
4. **Need:** It is an M x N matrix sequence representing the number of remaining resources for each process. When the Need[i] [j] = k, then process P[i] may require K more instances of resources type Rj to complete the assigned work.
   Nedd[i][j] = Max[i][j] - Allocation[i][j].
5. **Finish**: It is the vector of the order **m**. It includes a Boolean value (true/false) indicating whether the process has been allocated to the requested resources, and all resources have been released after finishing its task.

Safety Algorithm

It is a safety algorithm used to check whether or not a system is in a safe state or follows the safe sequence in a banker's algorithm:
1. There are two vectors **Wok** and **Finish** of length m and n in a safety algorithm.
Initialize: Work = Available
Finish[i] = false; for I = 0, 1, 2, 3, 4… n - 1.
2. Check the availability status for each type of resources [i], such as:
Need[i] <= Work
Finish[i] == false
If the i does not exist, go to step 4.
3. Work = Work +Allocation(i) // to get new resource allocation
Finish[i] = true
Go to step 2 to check the status of resource availability for the next process.
4. If Finish[i] == true; it means that the system is safe for all processes.

Resource Request Algorithm

A resource request algorithm checks how a system will behave when a process makes each type of resource request in a system as a request matrix.
Let create a resource request array R[i] for each process P[i]. If the Resource Request$_i$ [j] equal to 'K', which means the process P[i] requires 'k' instances of Resources type R[j] in the system.

1. When the number of **requested resources** of each type is less than the **Need** resources, go to step 2 and if the condition fails, which means that the process P[i] exceeds its maximum claim for the resource. As the expression suggests:
If Request(i) <= Need
Go to step 2;
2. And when the number of requested resources of each type is less than the available resource for each process, go to step (3). As the expression suggests:
If Request(i) <= Available
Else Process P[i] must wait for the resource since it is not available for use.
3. When the requested resource is allocated to the process by changing state:
Available = Available - Request
Allocation(i) = Allocation(i) + Request (i)
$Need_i = Need_i - Request_i$
When the resource allocation state is safe, its resources are allocated to the process P(i). And if the new state is unsafe, the Process P (i) has to wait for each type of Request R(i) and restore the old resource-allocation state.

Example: Consider a system that contains five processes P1, P2, P3, P4, P5 and the three resource types A, B and C. Following are the resources types: A has 10, B has 5 and the resource type C has 7 instances.

Process	Allocation			Max			Available		
	A	**B**	**C**	**A**	**B**	**C**	**A**	**B**	**C**
P1	0	1	0	7	5	3	3	3	2
P2	2	0	0	3	2	2			
P3	3	0	2	9	0	2			
P4	2	1	1	2	2	2			
P5	0	0	2	4	3	3			

Answer the following questions using the banker's algorithm:

1. What is the reference of the need matrix?
2. Determine if the system is safe or not.

Ans. 1: Context of the need matrix is as follows:
Need [i] = Max [i] - Allocation [i]
Need for P1: (7, 5, 3) - (0, 1, 0) = 7, 4, 3
Need for P2: (3, 2, 2) - (2, 0, 0) = 1, 2, 2
Need for P3: (9, 0, 2) - (3, 0, 2) = 6, 0, 0
Need for P4: (2, 2, 2) - (2, 1, 1) = 0, 1, 1
Need for P5: (4, 3, 3) - (0, 0, 2) = 4, 3, 1

Process	Need		
	A	**B**	**C**
P1	7	4	3
P2	1	2	2
P3	6	0	0
P4	0	1	1
P5	4	3	1

Hence, we created the context of need matrix.

Ans. 2: Apply the Banker's Algorithm:
Available Resources of A, B and C are 3, 3, and 2.
Now we check if each type of resource request is available for each process.
Step 1: For Process P1:
Need <= Available
7, 4, 3 <= 3, 3, 2 condition is **false**.
So, we examine another process, P2.
Step 2: For Process P2:
Need <= Available
1, 2, 2 <= 3, 3, 2 condition **true**
New available = available + Allocation
(3, 3, 2) + (2, 0, 0) => 5, 3, 2

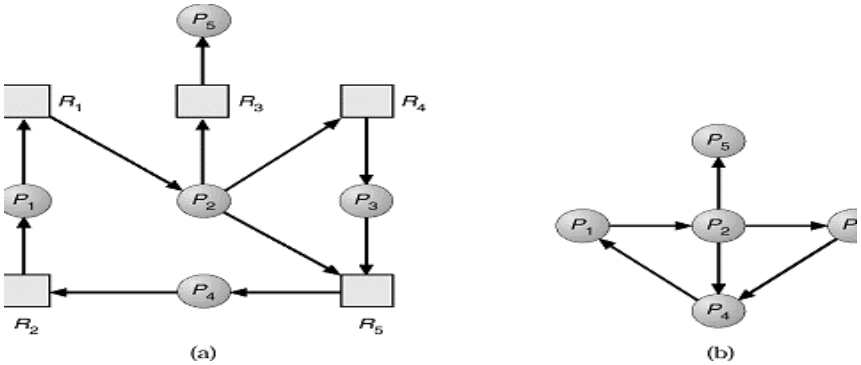
Similarly, we examine another process P3.
Step 3: For Process P3:
P3 Need <= Available
6, 0, 0 < = 5, 3, 2 condition is **false**.
Similarly, we examine another process, P4.
Step 4: For Process P4:
P4 Need <= Available
0, 1, 1 <= 5, 3, 2 condition is **true**
New Available resource = Available + Allocation
5, 3, 2 + 2, 1, 1 => 7, 4, 3
Similarly, we examine another process P5.
Step 5: For Process P5:
P5 Need <= Available
4, 3, 1 <= 7, 4, 3 condition is **true**
New available resource = Available + Allocation
7, 4, 3 + 0, 0, 2 => 7, 4, 5
Now, we again examine each type of resource request for processes P1 and P3.
Step 6: For Process P1:
P1 Need <= Available
7, 4, 3 <= 7, 4, 5 condition is **true**
New Available Resource = Available + Allocation
7, 4, 5 + 0, 1, 0 => 7, 5, 5
So, we examine another process P2.
Step 7: For Process P3:
P3 Need <= Available
6, 0, 0 <= 7, 5, 5 condition is true
New Available Resource = Available + Allocation
7, 5, 5 + 3, 0, 2 => 10, 5, 7
Hence, we execute the banker's algorithm to find the safe state and the safe sequence like P2, P4, P5, P1 and P3.

Deadlock Detection:

Single Instance of Each Resource Type

If all resources have only a single instance, then we can define a deadlockdetection algorithm that uses a variant of the resource-allocation graph, called a *wait-for* graph. We obtain this graph from the resource-allocation graph by removing the nodes of type resource and collapsing the appropriate edges. More precisely, an edge from Pi to ***Pj*** in a wait-for graph implies that process Pi is waiting for process Pjto release a resource that Pi needs.

An edge ***Pi*** → *Pj* exists in a wait-for graph if and only if the corresponding resourceallocation graph contains two edges ***Pi*** → *Rq* and *Rq* → *Pj* for some resource *Rq.* For example, in Figure 8.7, we present a resource-allocation graph and the corresponding wait-for graph. As before, a deadlock exists in the system if and only if the wait-for graph contains a cycle. To detect deadlocks, the system needs to *maintain* the wait-for graph and periodically to *invoke an algorithm* that searches for a cycle in the graph. **An** algorithm to detect a cycle in a graph requires an order of n2 operations, where *n* is the number of vertices in the graph.



Several Instances of a Resource Type

The wait-for graph scheme is not applicable to a resource-allocation system with multiple instances of each resource type. The deadlock-detection algorithm that we describe next is applicable to such a system. The algorithm employs several time-varying data structures that are similar to those used in the banker's algorithm :

The algorithm employs several times varying data structures:

- **Available –**
  A vector of length m indicates the number of available resources of each type.
- **Allocation –**
  An n*m matrix defines the number of resources of each type currently allocated to a process. Column represents resource and resource represent process.
- **Request –**
  An n*m matrix indicates the current request of each process. If request[i][j] equals k then process P_i is requesting k more instances of resource type R_j.

This algorithm has already been discussed here

Now, Bankers algorithm includes a **Safety Algorithm / Deadlock Detection Algorithm**

The algorithm for finding out whether a system is in a safe state can be described as follows:

Steps of Algorithm:

1. Let *Work* and *Finish* be vectors of length m and n respectively. Initialize *Work= Available*. For *i=0, 1, …., n-1*, if $Request_i$ = 0, then *Finish[i]* = true; otherwise, *Finish[i]*= false.
2. Find an index i such that both
   a) *Finish[i] == false*
   b) $Request_i$ <= *Work*
   If no such *i* exists go to step 4.
3. *Work= Work+* $Allocation_i$
   Finish[i]= true
   Go to Step 2.
4. If *Finish[i]== false* for some i, 0<=i<n, then the system is in a deadlocked state. Moreover, if *Finish[i]==false* the process P_i is deadlocked.

For example,

	Allocation			Request			Available		
	A	B	C	A	B	C	A	B	C
P0	0	1	0	0	0	0	0	0	0
P1	2	0	0	2	0	2			
P2	3	0	3	0	0	0			
P3	2	1	1	1	0	0			
P4	0	0	2	0	0	2			

1. In this, Work = [0, 0, 0] &
   Finish = [false, false, false, false, false]

2. *i=0* is selected as both Finish[0] = false and [0, 0, 0]<=[0, 0, 0].

3. Work =[0, 0, 0]+[0, 1, 0] =>[0, 1, 0] &
   Finish = [true, false, false, false, false].

4. *i=2* is selected as both Finish[2] = false and [0, 0, 0]<=[0, 1, 0].
5. Work =[0, 1, 0]+[3, 0, 3] =>[3, 1, 3] &
   Finish = [true, false, true, false, false].

6. *i=1* is selected as both Finish[1] = false and [2, 0, 2]<=[3, 1, 3].
7. Work =[3, 1, 3]+[2, 0, 0] =>[5, 1, 3] &
   Finish = [true, true, true, false, false].

8. *i=3* is selected as both Finish[3] = false and [1, 0, 0]<=[5, 1, 3].
9. Work =[5, 1, 3]+[2, 1, 1] =>[7, 2, 4] &
   Finish = [true, true, true, true, false].

10. *i=4* is selected as both Finish[4] = false and [0, 0, 2]<=[7, 2, 4].

11. Work =[7, 2, 4]+[0, 0, 2] =>[7, 2, 6] &
    Finish = [true, true, true, true, true].

12. Since Finish is a vector of all true it means **there is no deadlock** in this example.

Recovery from Deadlock:

There are two options for breaking a deadlock. One solution is simply to abort one or more processes to break the circular wait. The second option is to preempt some resources from one or more of the deadlocked processes.

Process Termination

To eliminate deadlocks by aborting a process, we use one of two methods. In both methods, the system reclaims all resources allocated to the terminated processes.

Abort all deadlocked processes: This method clearly will break the deadlock cycle, but at a great expense; these processes may have computed for along time, and the results of these partial computations must be discarded and probably recomputed later.

Abort one process at a time until the deadlock cycle is eliminated: This method incurs considerable overhead, since, after each process is aborted, a deadlock-detection algorithm must be invoked to determine whether any processes are still deadlocked. Aborting a process may not be easy. If the process was in the midst of updating a file, terminating it will leave that file in an incorrect state. Similarly, if the process was in the midst of printing data on the printer, the system must reset the printer to a correct state before printing the next job. If the partial termination method is used, then, given a set of deadlocked processes, we must determine which process (or processes) should be terminated in an attempt to break the deadlock.

Resource Preemption

To eliminate deadlocks using resource preemption, we successively preempt some resources from processes and give these resources to other processes until the deadlock cycle is broken. If preemption is required to deal with deadlocks, then three issues need to be addressed:

1. Selecting a victim: Which resources and which processes are to be preempted? As in process termination, we must determine the order of preemption to minimize cost. Cost factors may include such parameters as the

number of resources a deadlock process is holding, and the amount of time deadlocked process has thus far consumed during its execution.

2. Rollback: If we preempt a resource from a process, what should be done with that process? Clearly, it cannot continue with its normal execution; it is missing some needed resource. We must roll back the process to some safe state, and restart it from that state .Since, in general, it is difficult to determine what a safe state is, the simplest solution is a total rollback: Abort the process and then restart it. However, it is more effective to roll back the process only as far as necessary to break the deadlock. On the other hand, this method requires the system to keep more information about the state of all the running processes.

3. Starvation: How do we ensure that starvation will not occur? That is, how can we guarantee that resources will not always be preempted from the same process?

UNIT-3

Memory management

Memory consists of a large array of words or bytes, each with its own address. The CPU fetches instructions from memory according to the value of the program counter. These instructions may cause additional loading from and storing to specific memory addresses. A typical instruction-execution cycle, for example, first fetches an instruction from memory The instruction is then decoded and may use operands to be fetched from memory. After the instruction has been executed on the operands, results may be stored back in memory.

Address Binding:

Usually, a program resides on a disk as a binary executable file. To be executed, the program must be brought into memory and placed within a process. Depending on the memory management in use, the process may be moved between disk and memory during its execution. The processes on the disk that are waiting to be brought into memory for execution form the input queue.

A compiler will typically bind these symbolic addresses to relocatable addresses (such as "14 bytes from the beginning of this module''). The linkage editor or loader will in turn bind the relocatable addresses to absolute addresses (such as 74014). Each binding is a mapping from one address space to another. Classically, the binding of instructions and data to memory addresses can be done at any step along the way:

- **Compile time**. If you know at compile time where the process will reside in memory, then absolute code can be generated
- **Load time**. If it is not known at compile time where the process will reside in memory, then the compiler must generate relocatable code
- **Execution time**. If the process can be moved during its execution from one memory segment to another, then binding must be delayed until run time.

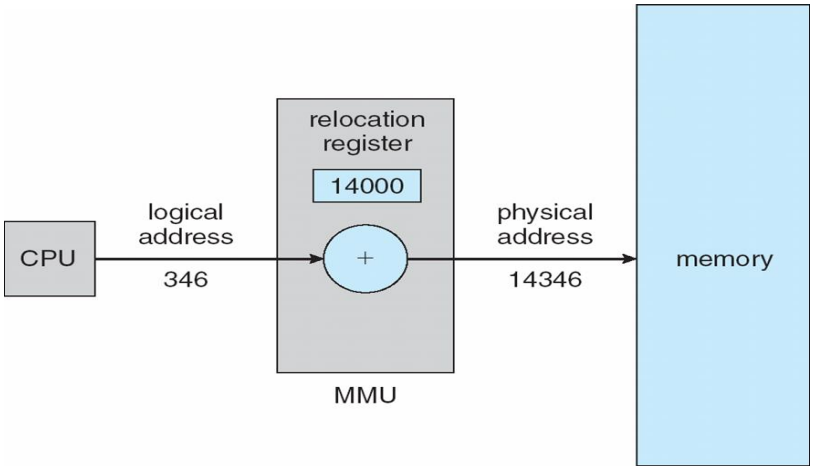
Logical Versus Physical Address Space:

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

The set of all logical addresses generated by a program is a **logical** address **space;** the set of all physical addresses corresponding to these logical addresses is a **physical** address **space.** Thus, in the execution-time address-binding scheme, the logical and physical address spaces differ. The run-time mapping from virtual to physical addresses is done by a hardware device called the **memory-management unit (MMU).**

For the time being, we illustrate this mapping with a simple MMU scheme, which is a generalization of the base-register . The base register is now called a **relocation register.**

The value in the relocation register is added to every address generated by a user process at the time it is sent to memory. For example, if the base is at 14000, then an attempt by the user to address location 0 is dynamically relocated to location 14000; an access to location 346 is mapped to location 14346.

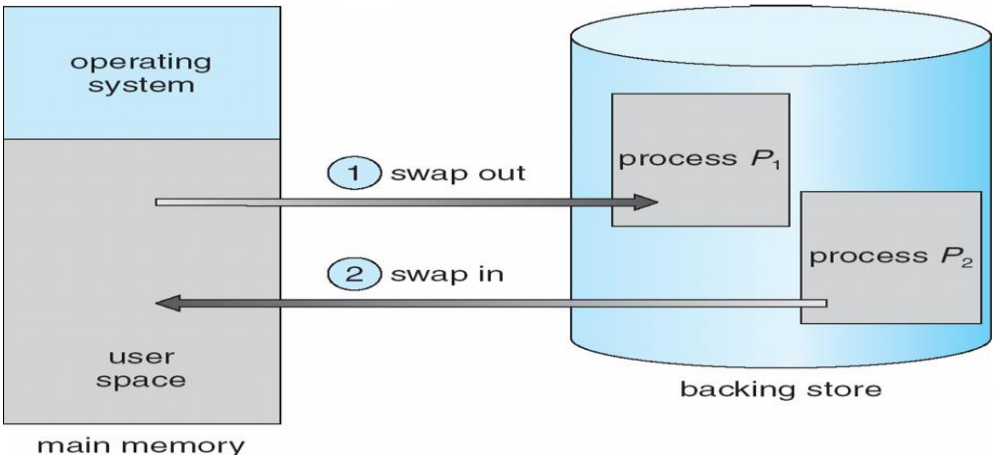


Swapping:

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

- For example, assume a multiprogramming environment with a round-robin CPU-scheduling algorithm. When a quantum expires, the memory manager will start to swap out the process that just finished and to swap another process into the memory space that has been freed. In the meantime, the CPU scheduler will allocate a time slice to some other process in memory. When each process finishes its quantum, it will be swapped with another process. Ideally, the memory manager can swap processes fast enough that some processes will be in memory, ready to execute, when the CPU scheduler wants to reschedule the CPU. In addition, the quantum must be large enough to allow reasonable amounts of computing to be done between swaps.

- A variant of this swapping policy is used for priority-based scheduling algorithms. If a higher-priority process arrives and wants service, the memory manager can swap out the lower-priority process and then load and execute the higher-priority process. When the higher-priority process finishes, the lower-priority process can be swapped back in and continued. This variant of swapping is sometimes called **roll out, roll in.**



Swapping requires a backing store. The backing store is commonly a fast disk. It must be large enough to accommodate copies of all memory images for all users, and it must provide direct access to these memory images.

Contiguous Memory Allocation:

Memory Mapping and Protection

Before discussing memory allocation further, we must discuss the issue of memory mapping and protection. We can provide these features by using a relocation register, with a limit register, The relocation register contains the value of the smallest physical address; the limit register contains the range of logical addresses (for example, relocation = 100040 and limit = 74600). With relocation and limit registers, each logical address must be less than the limit register; the MMU maps the logical address dynamically by adding the value in the relocation register. This mapped address is sent to memory .

Memory Allocation

- Now we are ready to turn to memory allocation. One of the simplest methods for allocating memory is to divide memory into several fixed-sized partitions. Each partition may contain exactly one process.

- In this multiplepartition method, when a partition is free, a process is selected from the input queue and is loaded into the free partition. When the process terminates, the partition becomes available for another process.
- In the fixed-partition scheme, the operating system keeps a table indicating which parts of memory are available and which are occupied. Initially, all memory is available for user processes and is considered one large block of available memory, a **hole.**
- When a process arrives and needs memory, we search for a hole large enough for this process. If we find one, we allocate only as much memory as is needed, keeping the rest available to satisfy future requests.
- As processes enter the system, they are put into an input queue. The operating system takes into account the memory requirements of each process and the amount of available memory space in determining which processes are allocated memory. When a process is allocated space, it is loaded into memory, and it can then compete for the CPU. When a process terminates, it releases its memory, which the operating system may then fill with another process from the input queue.
- In general, at any given time we have a set of holes of various sizes scattered throughout memory. When a process arrives and needs memory, the system searches the set for a hole that is large enough for this process.

- If the hole is too large, it is split into two parts. One part is allocated to the arriving process; the other is returned to the set of holes. When a process terminates, it releases its block of memory, which is then placed back in the set of holes. If the new hole is adjacent to other holes, these adjacent holes are merged to form one larger hole. At this point, the system may need to check whether there are processes waiting for memory and whether this newly freed and recombined memory could satisfy the demands of any of these waiting processes.
- This procedure is a particular instance of the general **dynamic storageallocation problem,** which concerns how to satisfy a request of size n from a list of free holes. There are many solutions to this problem. The **first-fit, best-fit,** and **worst-fit** strategies are the ones most commonly used to select a free hole from the set of available holes.

1 First fit. Allocate the first hole that is big enough.
2 Best fit. Allocate the smallest hole that is big enough.
3 Worst fit. Allocate the largest hole.

Fragmentation

- **External Fragmentation** – total memory space exists to satisfy a request, but it is not contiguous
- **Internal Fragmentation** – allocated memory may be slightly larger than requested memory; this size difference is memory internal to a partition, but not being used.

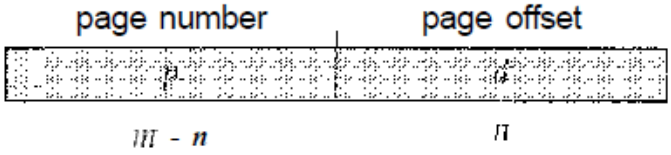
Paging:

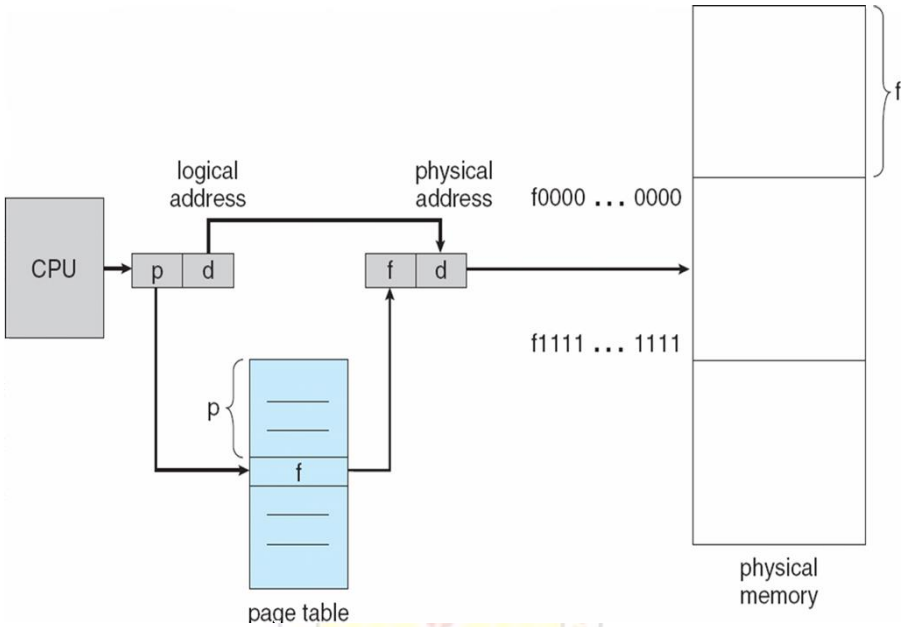
- Paging is a memory-management scheme that permits the physical address space of a process to be noncontiguous.
- Paging avoids the considerable problem of fitting memory chunks of varying sizes onto the backing store; most memory-management schemes used before the introduction of paging suffered from this problem.

Basic Method

- The basic method for implementing paging involves breaking physical memory into fixed-sized blocks called frames and breaking logical memory into blocks of the same size called pages.
- When a process is to be executed, its pages are loaded into any available memory frames from the backing store.
- The backing store is divided into fixed-sized blocks that are of the same size as the memory frames.
- Every address generated by the CPU is divided into two parts: a page number (p) and a page offset (d). The page number is used as an index into a page table.
- The page table contains the base address of each page in physical memory. This base address is combined with the page offset to define the physical memory address that is sent to the memory unit

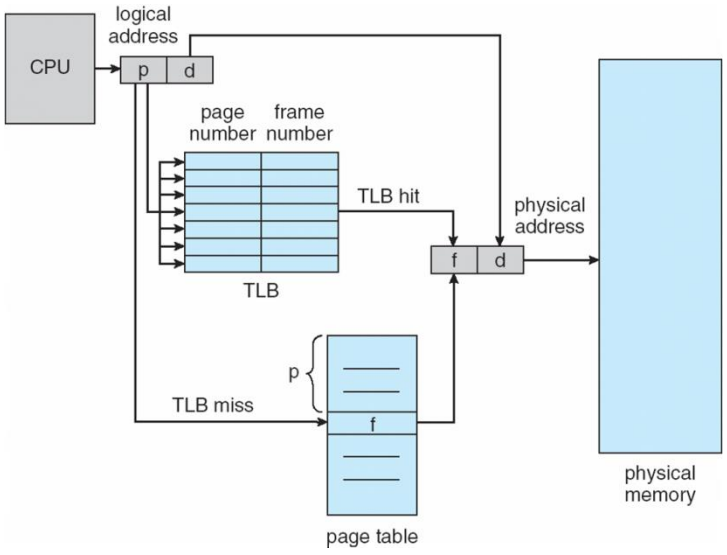
- The page size (like the frame size) is defined by the hardware. The size of a page is typically a power of 2, varying between 512 bytes and 16 MB per page.
- If the size of logical address space is 2^m and a page size is 2^n addressing units (bytes or words), then the high-order m – n bits of a logical address designate the page number, and the n low-order bits designate the page offset. Thus, the logical address is as follows:





Hardware Support

- The hardware implementation of the page table can be done in several ways. In the simplest case, the page table is implemented as a set of dedicated registers.The use of registers for the page table is satisfactory if the page table is reasonably small (for example, 256 entries).
- Most contemporary computers, however, allow the page table to be very large (for example, 1 million entries). For these machines, the use of fast registers to implement the page table is not feasible. Rather, the page table is kept in main memory, and a page-table base register (PTBR) points to the page table. Changing page tables requires changing only this one register.



Structure of Page Table:

A **page table** is the data structure used by a virtual memory system in a computer operating system to store the mapping between virtual addresses and physical addresses. Virtual addresses are used by the accessing process, while physical addresses are used by the hardware, or more specifically, by the RAM subsystem.

Role of the page table:

In operating systems that use virtual memory, every process is given the impression that it is working with large, contiguous sections of memory. Physically, the memory of each process may be dispersed across different areas of physical memory, or may have been moved (paged out) to another storage, typically to a hard disk drive.

Page table data:

The page table holds the mapping between a virtual address of a page and the address of a physical frame. There is also auxiliary information about the page such as a present bit, a dirty or modified bit, address space or process ID information, amongst others.

Page table types:

There are several types of page tables, that are best suited for different requirements.

- **Inverted page table**
- **Multilevel page table.**
- **Virtualized page table.**
- **Nested page tables**

Inverted page table:

The inverted page table (IPT) is best thought of as an off-chip extension of the TLB which uses normal system RAM. Unlike a true page table, it is not necessarily able to hold all current mappings. The OS must be prepared to handle misses, just as it would with a MIPS-style software-filled TLB.

The IPT combines a page table and a frame table into one data structure. At its core is a fixed-size table with the number of rows equal to the number of frames in memory. If there are 4000 frames, the inverted page table has 4000 rows. For each row there is an entry for the virtual page number (VPN), the physical page number (not the physical address), some other data and a means for creating a collision chain, as we will see later.

Multilevel page table

The multilevel page table may keep a few of the smaller page tables to cover just the top and bottom parts of memory and create new ones only when strictly necessary. Now, each of these smaller page tables are linked together by a master page table, effectively creating a tree data structure. There need not be only two levels, but possibly multiple ones.

A virtual address in this schema could be split into three parts: the index in the root page table, the index in the sub-page table, and the offset in that page. Multilevel page tables are also referred to as hierarchical page tables.

Virtualized page table

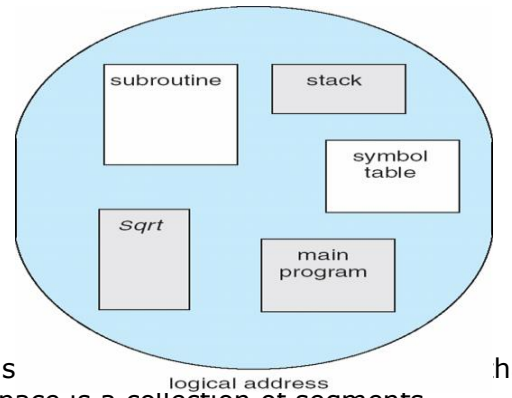
It was mentioned that creating a page table structure that contained mappings for every virtual page in the virtual address space could end up being wasteful. But, we can get around the excessive space concerns by putting the page table in virtual memory, and letting the virtual memory system manage the memory for the page table.

Nested page tables

Nested page tables can be implemented to increase the performance of hardware virtualization. By providing hardware support for page-table virtualization, the need to emulate is greatly reduced.

Segmentation:

- Users prefer to view memory as a collection of variable-sized segments, with no necessary ordering among segments .
- Consider how you think of a program when you are writing it. You think of it as a main program with a set of subroutines, procedures, functions, or modules. There may also be various data structures: tables, arrays, stacks, variables, and so on. Each of these modules or data elements is referred to by name. Each of these segments is of variable length; the length is intrinsically defined by the purpose of the segment in the program. Elements within a segment are identified by their offset from the beginning of the segment.



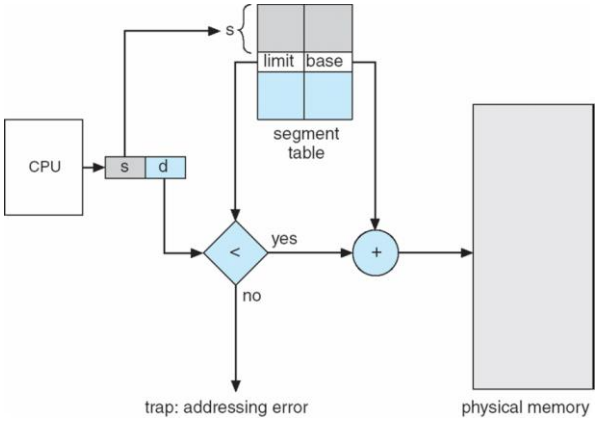
logical address

- **Segmentation** is hat supports this user view of memory. A logical-address space is a collection of segments.
- Each segment has a name and a length. The addresses specify both the segment name and the offset within the segment.
- The user therefore specifies each address by two quantities: a segment name and an offset. For simplicity of implementation, segments are numbered and are referred to by a segment number, rather than by a segment name. Thus, a logical address consists of a **two** tuple:

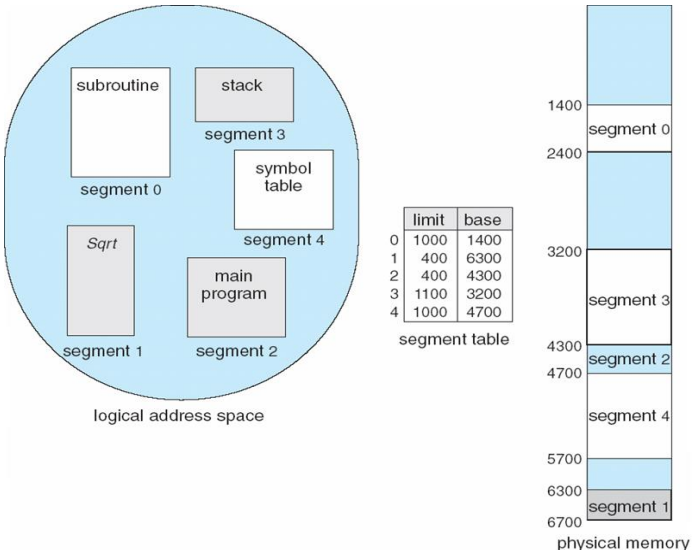
<segment-number, offset>,

Hardware

- Each entry of the segment table has a segment base and a segment **limit.** The segment base contains the starting physical address where the segment resides in memory, whereas the segment limit specifies the length of the segment.
- A logical address consists of two parts: a segment number, s, and an offset into that segment, d. The segment number is used as an index into the segment table.
- The offset d of the logical address must be between 0 and the segment limit. If it is not, we trap to the operating system (logical addressing attempt beyond end of segment). If this offset is legal, it is added to the segment base to produce the address in physical memory of the desired byte..



- As an example. We have five segments numbered from 0 through 4. The segments are stored in physical memory as shown. The segment table has a separate entry for each segment, giving the beginning address of the segment in physical memory (or base) and the length of that segment (or limit).
- For example, segment 2 is 400 bytes long and begins at location 4300. Thus, a reference to



byte 53 of segment 2 is mapped onto location 4300 + 53 = 4353. A reference to segment 3, byte 852, is mapped to 3200 (the base of segment 3) + 852 = 4052. A reference to byte 1222 of segment 0 would result in a trap to the operating system, as this segment is only 1,000 bytes long.

Protection and Sharing

The memorymapping hardware will check the protection bits associated with each segmenttable entry to prevent illegal accesses to memory, such as attempts to write into a read-only segment, or to use an execute-only segment as data. Another advantage of segmentation involves the sharing of code or data

Virtual Memory

Virtual memory is a technique that allows the execution of processes that may not be completely in memory. One major advantage of this scheme is that programs can be larger than physical memory. Further, virtual memory abstracts main memory into an extremely large, uniform array of storage, separating logical memory as viewed by the user from physical memory.

Demand Paging:

- A demand-paging system is similar to a paging system with swapping . Processes reside on secondary memory (which is usually a disk). When we want to execute a process, we swap it into memory. Rather than swapping the entire process into memory, however, we use a **lazy swapper.**
- A lazy swapper never swaps a page into memory unless that page will be needed. Since we are now viewing a process as a sequence of pages, rather than as one large contiguous address space, use of swap is technically incorrect. A swapper manipulates entire processes, whereas a **pager** is concerned with the individual pages of a process. We thus use pager, rather than swapper, in connection with demand paging.
- When a process is to be swapped in, the pager guesses which pages will be used before the process is swapped out again. Instead of swapping in a whole process, the pager brings only those necessary pages into memory. Thus, it avoids reading into memory pages that will not be used anyway, decreasing the swap time and the amount of physical memory needed.
- With this scheme, we need some form of hardware support to distinguish between those pages that are in memory and those pages that are on the disk. This time, however, when this bit is set to "valid," this value indicates that the associated page is both legal and in memory. If the bit is set to "invalid," this value indicates that the page either is not valid (that is, not in the logical address space of the process), or is valid but is currently on the disk.
- While the process executes and accesses pages that are memory resident, execution proceeds normally. But what happens if the process tries to access a page that was not brought into memory? Access to a page marked invalid causes a page-fault trap. The paging hardware, in

translating the address through the page table, will notice that the invalid bit is set, causing a trap to the operating system.

- This trap is the result of the operating system's failure to bring the desired page into memory. The procedure for handling this page fault is straightforward :

1. We check an internal table (usually kept with the process control block) for this process, to determine whether the reference was a valid or invalid memory access.

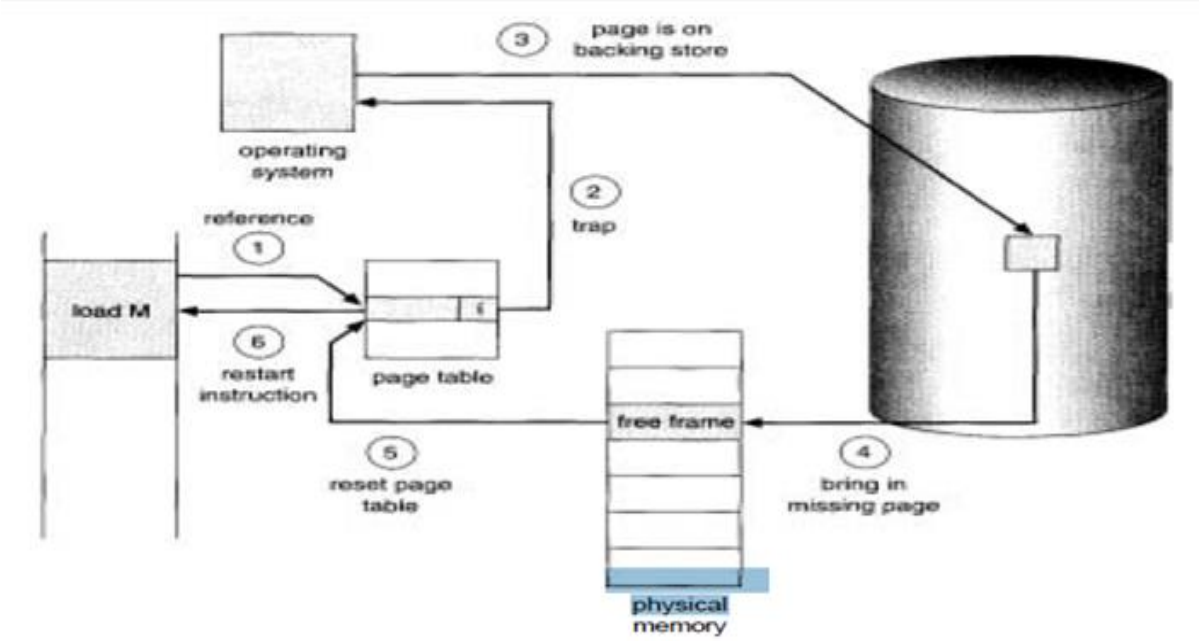


Figure 10.4 Steps in handling a page fault.

2. If the reference was invalid, we terminate the process. If it was valid, but we have not yet brought in that page, we now page it in.

3. We find a free frame (by taking one from the free-frame list, for example).

4. We schedule a disk operation to read the desired page into the newly allocated frame.

5. When the disk read is complete, we modify the internal table kept with the process and the page table to indicate that the page is now in memory.

6. We restart the instruction that was interrupted by the illegal address trap.

In this way, we are able to execute a process, even though portions of it are not (yet) in memory. When the process tries to access locations that are not in memory, the hardware traps to the operating system (page fault). The operating system reads the desired page into memory and restarts the process as though the page had always been in memory.

This scheme is **pure demand paging:** Never bring **a** page into memory until it is required. The hardware to support demand paging is the same as the hardware for paging and swapping:

Page table: This table has the ability to mark an entry invalid through a valid-invalid bit or special value of protection bits.

Secondary memory: This memory holds those pages that are not present in main memory. The secondary memory is usually a high-speed disk. It is known as the swap device, and the section of disk used for this purpose is known as **swap space.** .

Page Replacement Algorithm:

Page replacement algorithms are the techniques using which an Operating System decides which memory pages to swap out, write to disk when a page of memory needs to be allocated. Paging happens whenever a page fault occurs and a free page cannot be used for allocation purpose accounting to reason that pages are not available or the number of free pages is lower than required pages.

Reference String

The string of memory references is called reference string. Reference strings are generated artificially or by tracing a given system and recording the address of each memory reference. The latter choice produces a large number of data, where we note two things.

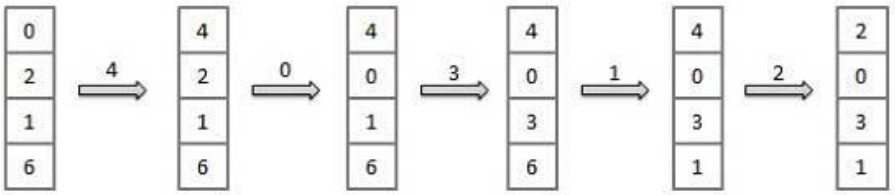
- For a given page size, we need to consider only the page number, not the entire address.
- If we have a reference to a page **p**, then any immediately following references to page **p** will never cause a page fault. Page p will be in memory after the first reference; the immediately following references will not fault.
- For example, consider the following sequence of addresses – 123,215,600,1234,76,96
- If page size is 100, then the reference string is 1,2,6,12,0,0

First In First Out (FIFO) algorithm

- Oldest page in main memory is the one which will be selected for replacement.
- Easy to implement, keep a list, replace pages from the tail and add new pages at the head.

Reference String : 0, 2, 1, 6, 4, 0, 1, 0, 3, 1, 2, 1

Misses : x x x x x x x x x



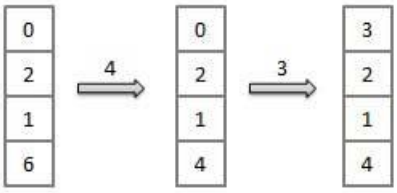
Fault Rate = 9 / 12 = 0.75

Optimal Page algorithm

- An optimal page-replacement algorithm has the lowest page-fault rate of all algorithms. An optimal page-replacement algorithm exists, and has been called OPT or MIN.
- Replace the page that will not be used for the longest period of time. Use the time when a page is to be used.

Reference String : 0, 2, 1, 6, 4, 0, 1, 0, 3, 1, 2, 1

Misses : x x x x x x



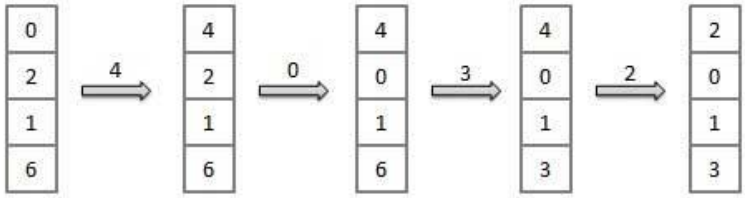
Fault Rate = 6 / 12 = 0.50

Least Recently Used (LRU) algorithm

- Page which has not been used for the longest time in main memory is the one which will be selected for replacement.
- Easy to implement, keep a list, replace pages by looking back into time.

Reference String : 0, 2, 1, 6, 4, 0, 1, 0, 3, 1, 2, 1

Misses : x x x x x x x x



Fault Rate = 8 / 12 = 0.67

Page Buffering algorithm

- To get a process start quickly, keep a pool of free frames.
- On page fault, select a page to be replaced.
- Write the new page in the frame of free pool, mark the page table and restart the process.
- Now write the dirty page out of disk and place the frame holding replaced page in free pool.

Least frequently Used(LFU) algorithm

- The page with the smallest count is the one which will be selected for replacement.
- This algorithm suffers from the situation in which a page is used heavily during the initial phase of a process, but then is never used again.

Most frequently Used(MFU) algorithm

- This algorithm is based on the argument that the page with the smallest count was probably just brought in and has yet to be used.

Allocation of Frames:

An important aspect of operating systems, virtual memory is implemented using demand paging. Demand paging necessitates the development of a page-replacement algorithm and a **frame allocation algorithm**. Frame allocation algorithms are used if you have multiple processes; it helps decide how many frames to allocate to each process.

There are various constraints to the strategies for the allocation of frames:

- You cannot allocate more than the total number of available frames.
- At least a minimum number of frames should be allocated to each process. This constraint is supported by two reasons. The first reason is, as less number of frames are allocated, there is an increase in the page fault ratio, decreasing the performance of the execution of the process. Secondly, there should be enough frames to hold all the different pages that any single instruction can reference.

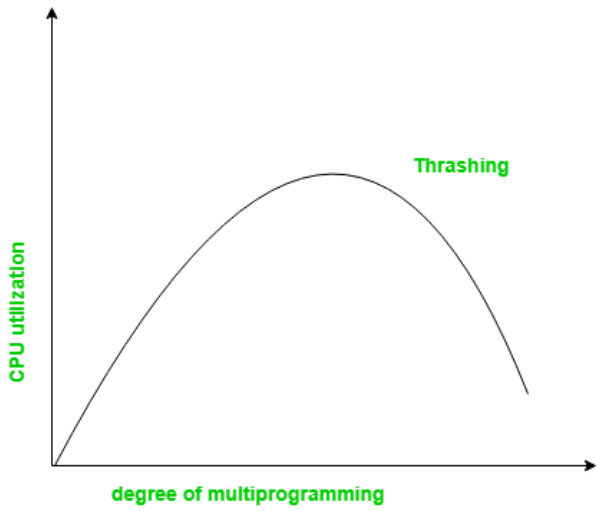
Frame allocation algorithms –

The two algorithms commonly used to allocate frames to a process are:

1. **Equal allocation:** In a system with x frames and y processes, each process gets equal number of frames, i.e. x/y. For instance, if the system has 48 frames and 9 processes, each process will get 5 frames. The three frames which are not allocated to any process can be used as a free-frame buffer pool.
    - **Disadvantage:** In systems with processes of varying sizes, it does not make much sense to give each process equal frames. Allocation of a large number of frames to a small process will eventually lead to the wastage of a large number of allocated unused frames.
2. **Proportional allocation:** Frames are allocated to each process according to the process size. For a process p_i of size s_i, the number of allocated frames is $\mathbf{a_i = (s_i/S)*m}$, where S is the sum of the sizes of all the processes and m is the number of frames in the system. For instance, in a system with 62 frames, if there is a process of 10KB and another process of 127KB, then the first process will be allocated (10/137)*62 = 4 frames and the other process will get (127/137)*62 = 57 frames.
    - **Advantage:** All the processes share the available frames according to their needs, rather than equally.

Thrashing:

If this page fault and then swapping happening very frequently at higher rate, then operating system has to spend more time to swap these pages. This state is called thrashing. Because of this, CPU utilization is going to be reduced.



Effect of Thrashing

Whenever thrashing starts, operating system tries to apply either **Global page replacement** Algorithm or **Local page replacement** algorithm.

Global Page Replacement

Since global page replacement can access to bring any page, it tries to bring more pages whenever thrashing found. But what actually will happen is, due to this, no process gets enough frames and by result thrashing will be increase more and more. So global page replacement algorithm is not suitable when thrashing happens.

Local Page Replacement

Unlike global page replacement algorithm, local page replacement will select pages which only belongs to that process. So there is a chance to reduce the thrashing. But it is proven that there are many disadvantages if we use local page replacement. So local page replacement is just alternative than global page replacement in thrashing scenario.

FILE-SYSTEM INTERFACE

File Concept:

File Attributes

- A file is a named collection of related information that is recorded on secondary storage.
- Commonly, files represent programs (both source and object forms) and data. Data files may be numeric, alphabetic, alphanumeric, or binary. Files may be free form, such as text files, or may be formatted rigidly. In general, a file is a sequence of bits, bytes, lines, or records, the meaning of which is defined by the file's creator and user. The concept of a file is thus extremely general.
- The information in a file is defined by its creator. Many different types of information may be stored in a file-source programs, object programs, executable programs, numeric data, text, payroll records, graphic images, sound recordings, and so on.
- A file is named, for the convenience of its human users, and is referred to by its name. A name is usually a string of characters, such as examp1e.c
- A file has certain other attributes, which vary from one operating system to another, consist of these:
- **Name**: The symbolic file name is the only information kept in humanreadable form.
- **Identifier**: This unique tag, usually a number, identifies the file within the file system; it is the non-human-readable name for the file.
- **Type**: This information is needed for those systems that support different types.
- **Location:** This information is a pointer to a device and to the location of the file on that device.
- **Size:** The current size of the file (in bytes, words, or blocks), and possibly the maximum allowed size are included in this attribute.
- **Protection:** Access-control information determines who can do reading, writing, executing, and so on.
- **Time, date, and user identification:** This information may be kept for creation, last modification, and last use. These data can be useful for protection, security, and usage monitoring.

File Operations

A file is an **abstract data type.** To define a file properly, we need to consider the operations that can be performed on files. The operating system can provide system calls to create, write, read, reposition, delete, and truncate files.

- **Creating a file:** Two steps are necessary to create a file. First, space in the file system must be found for the file.
- **Writing a file:** To write a file, we make a system call specifying both the name of the file and the information to be written to the file.
- **Reading a file:** To read from a file, we use a system call that specifies the name of the file and where (in memory) the next block of the file should be put.
- **Repositioning within a file:** The directory is searched for the appropriate entry, and the current-file-position is set to a given value. Repositioning within a file does not need to involve any actual I/O. This file operation is also known as a file seek.
- **a Deleting a file:** To delete a file, we search the directory for the named file.
- **a Truncating a file:** The user may want to erase the contents of a file but keep its attributes.
- These six basic operations certainly comprise the minimal set of required file operations. Other common operations include appending new information to the end of an existing file and renaming an existing file.

several pieces of information are associated with an open file.

File pointer: On systems that do not include a file offset as part of the read and write system calls, the system must track the last read-write location as a current-file-position pointer.

File open count: As files are closed, the operating system must reuse its open-file table entries, or it could run out of space in the table. Because multiple processes may open a file, the system must wait for the last file
to close before removing the open-file table entry.

Disk location of the file: Most file operations require the system to modify data within the file. The information needed to locate the file on disk is kept in memory to avoid having to read it from disk for each operation.

Access rights: Each process opens a file in an access mode. This information is stored on the per-process table so the operating system can allow or deny subsequent I/O requests.

File Types

- If an operating system recognizes the type of a file, it can then operate on the file in reasonable ways.
- A common technique for implementing file types is to include the type as part of the file name. The name is split into two parts-a name and an extension, usually separated by a period character.
- For example, in MS-DOS, a name can consist of up to eight characters followed by a period and terminated by an extension of up to three characters. The system uses the extension to indicate the type of the file and the type of operations that can be done on that file.
- For instance, only a file with a .com, .exe, or .bat extension can be executed. The .corn and .exe files are two forms of binary executable files, whereas a .bat file is a **batch file** containing, in ASCII format, commands to the operating system. MS-DOS recognizes only a few extensions, but application programs also use extensions to indicate file types in which they are interested. For example, assemblers expect source files to have an .asm extension, and the Wordperfect word processor expects its file to end with a .wp extension

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

Access Methods:

Files store information. When it is used, this information must be accessed and read into computer memory. The information in the file can be accessed in several ways.

Sequential Access

The simplest access method is **sequential access.** Information in the file is processed in order, one record after the other. This mode of access is by far the most common; for example, editors and compilers usually access files in this fashion.

The bulk of the operations on a file is reads and writes. A read operation reads the next portion of the file and automatically advances a file pointer, which tracks the I/O location. Similarly, a write appends to the end of the file and advances to the end of the newly written material (the new end of file . Sequential access is based on a tape model of a file

Direct Access

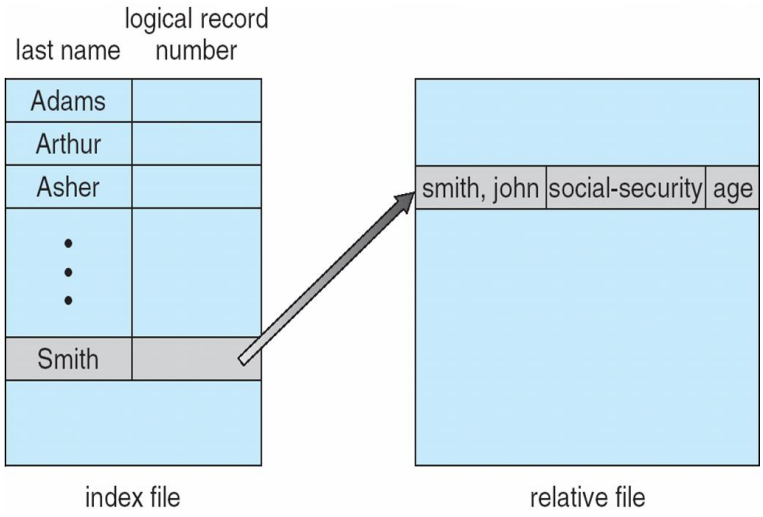
Another method is **direct access** (or **relative access).** A file is made up of fixed length **logical records** that allow programs to read and write records rapidly in no particular order. The direct-access method is based on a disk model of a file, since disks allow random access to any file block. For direct access, the file is viewed as a numbered sequence of blocks or records. A direct-access file allows arbitrary blocks to be rea d or written. Thus, we may read block 14, then read block 53, and then write block 7. There are no restrictions on the order of reading or writing for a direct-access file. Direct-access files are of great use for immediate access to large amounts of information. Databases are often of this type.

As a simple example, on an airline-reservation system, we might store all the information about a particular flight (for example, flight 713) in the block identified by the flight number. Thus, the number of available seats for flight 713 is stored in block 713 of the reservation file. To store information about a larger set, such as people, we might compute a hash function on the people's names, or search a small in-memory index to determine a block to read and search.

For the direct-access method, the file operations must be modified to include the block number as a parameter. Thus, we have read n, where n is the block number, rather than read next, and write n rather than write next. An alternative approach is to retain read next and write next, as with sequential access, and to add an operation position file to n, where n is the block number. Then, to effect a read n, we would position to n and then read next.

Other Access Methods

Other access methods can be built on top of a direct-access method. These methods generally involve the construction of an index for the file. The index, like an index in the back of a book, contains pointers to the various blocks. TO find a record in the file, we first search the index, and then use the pointer to access the file directly and to find the desired record



Directory Structure:

The file systems of computers can be extensive. Some systems store millions of files on terabytes of disk. To manage all these data, we need to organize them. This organization is usually done in two parts. First, disks are split into one or more partitions, also known as minidisks in the IBM world or volumes in the PC and Macintosh arenas. Typically, each disk on a system contains at least one partition, which is a low-level structure in which files and directories reside.

Second, each partition contains information about files within it. This information is kept in entries in a device directory or volume table of contents. The device directory (more commonly known simply as a directory) records information-such as name, location, size, and type-for all files on that partition.

the operations that are to be performed on a directory:

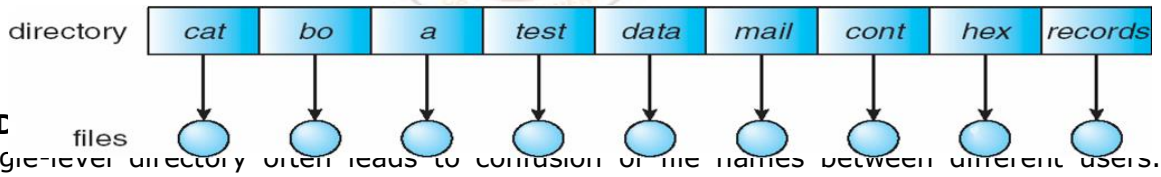
- **Search for a file:** We need to be able to search a directory structure to find the entry for a particular file.
- **Create a file:** New files need to be created and added to the directory.
- **Delete a file:** When a file is no longer needed, we want to remove it from the directory.
- **List a directory:** We need to be able to list the files in a directory, and the contents of the directory entry for each file in the list.
- **Rename a file:** Because the name of a file represents its contents to its users, the name must be changeable when the contents or use of the file changes.

Traverse the file system: We may wish to access every directory, and every file within a directory structure.

Single-Level Directory

The simplest directory structure is the single-level directory. All files are contained in the same directory, which is easy to support and understand . A single-level directory has significant limitations, however, when the number of files increases or when the system has more than one user.

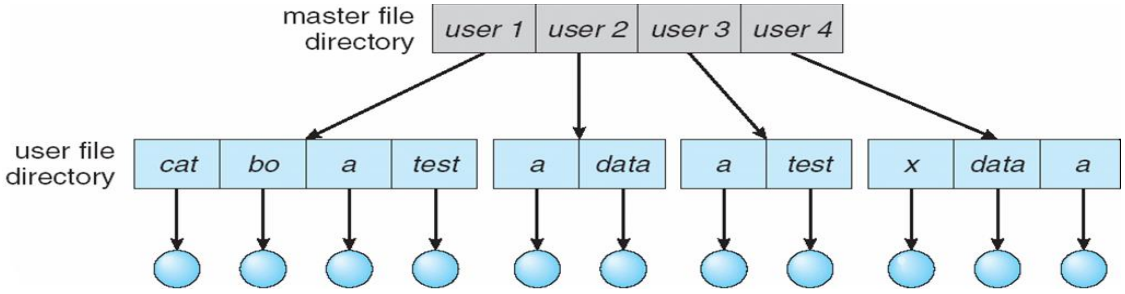
Even a single user on a single-level directory may find it difficult to remember thenames of all the files, as the number of files increases. It is not uncommon for a user to have hundreds of files on one computer system and an equal number of additional files on another system. In such an environment, keeping track of so many files is a daunting task.



Two-Level D

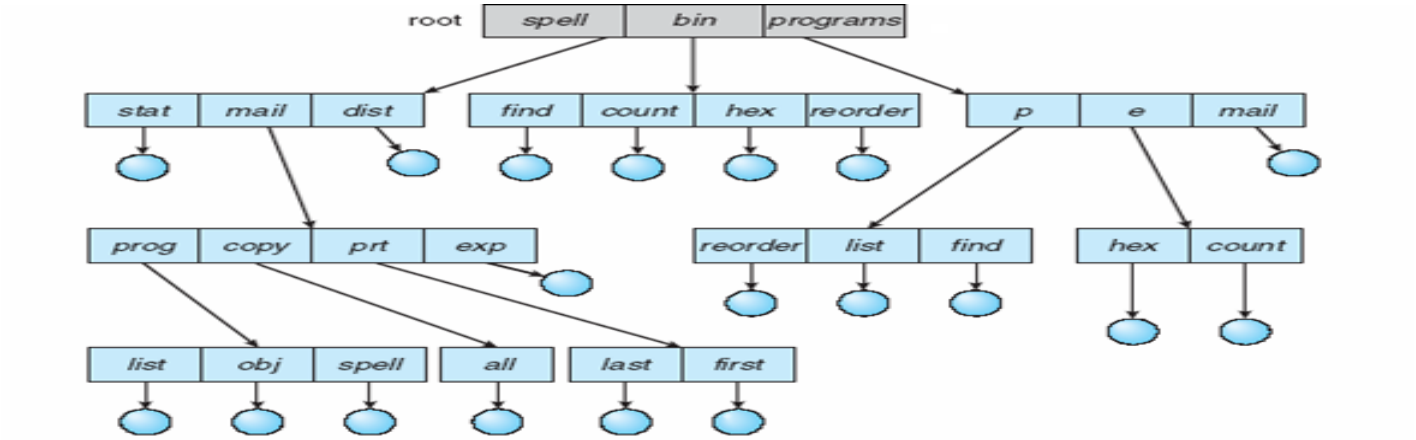
A single-level directory often leads to confusion of file names between different users. The standard solution is to create a separate directory for each user. In the two-level directory structure, each user has her own user file directory (UFD). Each UFD has a similar structure, but lists only the files of a single user. When a user job starts or a user logs in, the system's master file directory (MFD) is searched.

Although the two-level directory structure solves the name-collision problem, it still has disadvantages. This structure effectively isolates one user from another. This isolation is an advantage when the users are completely indep ndent but is a disadvantage when the users want to cooperate on some task and to access one another's files. Some systems simply do not allow local user files to be accessed by other users.



Tree-Structured Directories

Once we have seen how to view a two-level directory as a two-level tree, the natural generalization is to extend the directory structure to a tree of arbitrary height . This generalization allows users to create their own subdirectories and to organize their files accordingly. A tree is the most common directory structure. The tree has a root directory, and every file in the system has a unique path name.



Protection:

When information is kept in a computer system, we want to keep it safe from physical damage (reliability) and improper access (protection).

Reliability is generally provided by duplicate copies of files. Many computers have systems programs that automatically (or through computer-operator intervention) copy disk files to tape at regular intervals (once per day or week or month) to maintain a copy should a file system be accidentally destroyed.

File systems can be damaged by hardware problems (such as errors in reading or writing), power surges or failures, head crashes, dirt, temperature extremes. Files may be deleted accidentally. Bugs in the file-system software can also cause file contents to be lost. Protection can be provided in many ways. For a small single-user system, we might provide protection by physically removing the floppy disks and locking them in a desk drawer or file cabinet. In a multiuser system, however, other mechanisms are needed.

Types of Access

The need to protect files is a direct result of the ability to access files. Systems that do not permit access to the files of other users do not need protection. Thus, we could provide complete protection by prohibiting access. Alternatively, we could provide free access with no protection. Both approaches are too extreme for general use. What is needed is **controlled access.**

. Several different types of operations may be controlled:

- **Read:** Read from the file.
- **Write:** Write or rewrite the file.
- **Execute:** Load the file into memory and execute it.
- **Append:** Write new information at the end of the file.
- **Delete:** Delete the file and free its space for possible reuse.
- **List:** List the name and attributes of the file.

Other operations, such as renaming, copying, or editing the file, may also be controlled

Access Control

The most common approach to the protection problem is to make access dependent on the identity of the user. Various users may need different types ofaccess to a file or directory. The most general scheme to implement identity-dependent access is to associate with each file and directory an access-control list (ACL) specifying the user name and the types of access allowed for each user. When a user requests access to a particular file, the operating system checks the access list associated with that file. If that user is listed for the requested access, the access is allowed. Otherwise, a protection violation occurs, and the user job is denied access to the file.

three classifications of users in connection with each file:

- Owner: The user who created the file is the owner.
- Group: A set of users who are sharing the file and need similar access is a group, or work group.
- Universe: All other users in the system constitute the universe.

Other Protection Approaches

Another approach to the protection problem is to associate a password with each file. Just as access to the computer system is often controlled by a password, access to each file can be controlled by a password. If the passwords are chosen randomly and changed often, this scheme may be effective in limiting access to a file to only those users who know the password. This scheme, however, has several disadvantages. First, the number of passwords that a user needs to remember may become large, making the scheme impractical. Secondly, if only one password is used for all the files, then, once it is discovered, all files are accessible.

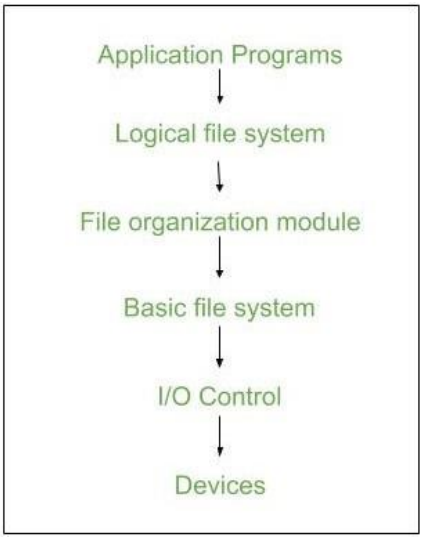
UNIT- 4

File Systems Implementation

A**file** is a collection of related information. The file system resides on secondary storage and provides efficient and convenient access to the disk by allowing data to be stored, located, and retrieved.

File system Structure:

Disks provide the bulk of secondary storage on which a file system is maintained. the file system itself is generally composed of many different levels. Each level in the design uses the features of lower levels to create new features for use by higher levels. The lowest level, the I/O control, consists of **device drivers** and interrupt handlers to transfer information between the main memory and the disk system.



- **I/O Control level –**
  Device drivers acts as interface between devices and Os, they help to transfer data between disk and main memory. It takes block number a input and as output it gives low level hardware specific instruction.
  Basic file system –
  It Issues general commands to device driver to read and write physical blocks on disk.It manages the memory buffers and caches. A block in buffer can hold the contents of the disk block and cache stores frequently used file system metadata.
- **File organization Module –**
  It has information about files, location of files and their logical and physical blocks.Physical blocks do not match with logical numbers of logical blo-ck numbered from 0 to N. It also has a free space which tracks unallocated blocks.
- **Logical file system –**
  It manages metadata information about a file i.e includes all details about a file except the actual contents of file. It also maintains via file control blocks. File control block (FCB) has information about a file – owner, size, permissions, location of file contents.

Advantages :

1. Duplication of code is minimized.
2. Each file system can have its own logical file system.

Disadvantages :
If we access many files at same time then it results in low performance.

We can **implement** file system by using two types data structures :

1. On-disk Structures –
Generally they contain information about total number of disk blocks, free disk blocks, location of them and etc. Below given are different on-disk structures :

1. **Boot Control Block –**
   It is usually the first block of volume and it contains information needed to boot an operating system.In UNIX it is called boot block and in NTFS it is called as partition boot sector.
2. **Volume Control Block –**
   It has information about a particular partition ex:- free block count, block size and block pointers etc.In UNIX it is called super block and in NTFS it is stored in master file table.
3. **Directory Structure –**
   They store file names and associated inodenumbers.In UNIX, includes file names and associated file names and in NTFS, it is stored in master file table.
4. **Per-File FCB –**
   It contains details about files and it has a unique identifier number to allow association with directory entry. In NTFS it is stored in master file table.

2. In-Memory Structure :
They are maintained in main-memory and these are helpful for file system management for caching. Several in-memory structures given below :

1. **Mount Table –**
   It contains information about each mounted volume.
2. **Directory-Structure cache –**
   This cache holds the directory information of recently accessed directories.
3. **System wide open-file table –**
   It contains the copy of FCB of each open file.
4. **Per-process open-file table –**
   It contains information opened by that particular process and it maps with appropriate system wide open-file.

Directory Implementation :

1. **Linear List –**
   It maintains a linear list of filenames with pointers to the data blocks.It is time-consuming also.To create a new file, we must first search the directory to be sure that no existing file has the same name then we add a file at end of the directory.To delete a file, we search the directory for the named file and release the space.To reuse the directory entry either we can mark the entry as unused or we can attach it to a list of free directories.
2. **Hash Table –**
   The hash table takes a value computed from the file name and returns a pointer to the file. It decreases the directory search time. The insertion and deletion process of files is easy. The

major difficulty is hash tables are its generally fixed size and hash tables are dependent on hash function on that size.

File Allocation Methods:

The allocation methods define how the files are stored in the disk blocks. There are three main disk space or file allocation methods.

- Contiguous Allocation
- Linked Allocation
- Indexed Allocation

The main idea behind these methods is to provide:

- Efficient disk space utilization.
- Fast access to the file blocks.

All the three methods have their own advantages and disadvantages as discussed below:

1. Contiguous Allocation

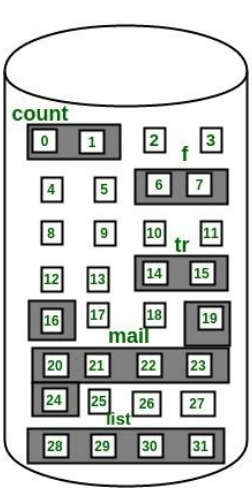
In this scheme, each file occupies a contiguous set of blocks on the disk. For example, if a file requires n blocks and is given a block b as the starting location, then the blocks assigned to the file will be: *b, b+1, b+2,……b+n-1.* This means that given the starting block address and the length of the file (in terms of blocks required), we can determine the blocks occupied by the file.
The directory entry for a file with contiguous allocation contains

- Address of starting block
- Length of the allocated portion.

The *file 'mail'* in the following figure starts from the block 19 with length = 6 blocks. Therefore, it occupies *19, 20, 21, 22, 23, 24* blocks.

Advantages:

- Both the Sequential and Direct Accesses are supported by this. For direct access, the address of the kth block of the file which starts at block b can easily be obtained as (b+k).
- This is extremely fast since the number of seeks are minimal because of contiguous allocation of file blocks.



Directory

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

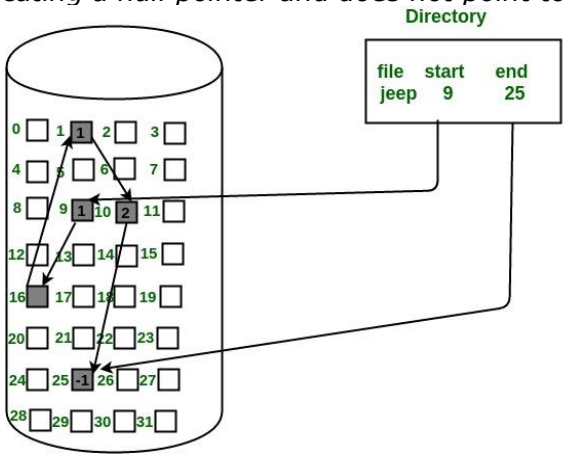
Disadvantages:

- This method suffers from both internal and external fragmentation. This makes it inefficient in terms of memory utilization.
- Increasing file size is difficult because it depends on the availability of contiguous memory at a particular instance.

2. Linked List Allocation

In this scheme, each file is a linked list of disk blocks which **need not be** contiguous. The disk blocks can be scattered anywhere on the disk.
The directory entry contains a pointer to the starting and the ending file block. Each block contains a pointer to the next block occupied by the file.

The file 'jeep' in following image shows how the blocks are randomly distributed. The last block (25) contains -1 indicating a null pointer and does not point to any other block.



Advantages:

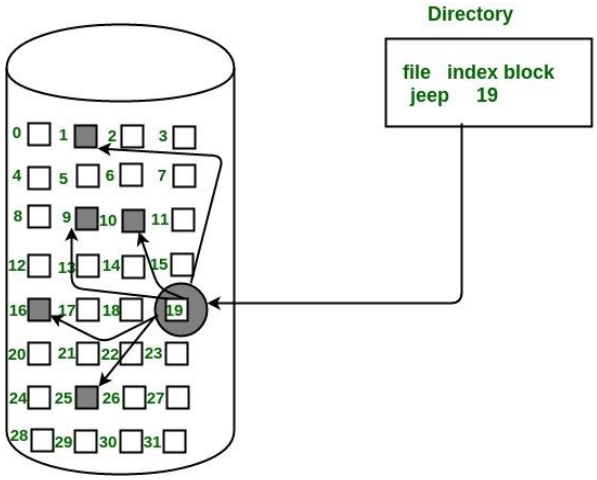
- This is very flexible in terms of file size. File size can be increased easily since the system does not have to look for a contiguous chunk of memory.
- This method does not suffer from external fragmentation. This makes it relatively better in terms of memory utilization.

Disadvantages:

- Because the file blocks are distributed randomly on the disk, a large number of seeks are needed to access every block individually. This makes linked allocation slower.
- It does not support random or direct access. We can not directly access the blocks of a file. A block k of a file can be accessed by traversing k blocks sequentially (sequential access) from the starting block of the file via block pointers.
- Pointers required in the linked allocation incur some extra overhead.

3. Indexed Allocation

In this scheme, a special block known as the **Index block** contains the pointers to all the blocks occupied by a file. Each file has its own index block. The ith entry in the index block contains the disk address of the ith file block. The directory entry contains the address of the index block as shown in the image:



Advantages:

- This supports direct access to the blocks occupied by the file and therefore provides fast access to the file blocks.
- It overcomes the problem of external fragmentation.

Disadvantages:

- The pointer overhead for indexed allocation is greater than linked allocation.
- For very small files, say files that expand only 2-3 blocks, the indexed allocation would keep one entire block (index block) for the pointers which is inefficient in terms of memory utilization. However, in linked allocation we lose the space of only 1 pointer per block.

Free space management:

The system keeps tracks of the free disk blocks for allocating space to files when they are created. Also, to reuse the space released from deleting the files, free space management becomes crucial. The system maintains a free space list which keeps track of the disk blocks that are not allocated to some file or directory. The free space list can be implemented mainly as:

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

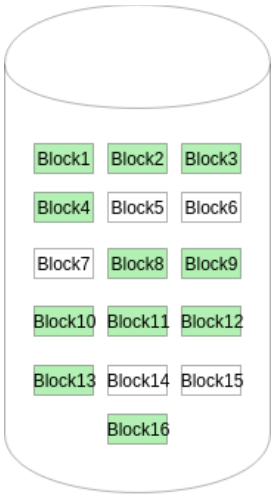


Figure - 1

Advantages –

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

☐ For the *Figure-1*, we scan the bitmap sequentially for the first non-zero word.
The first group of 8 bits (00001110) constitute a non-zero word since all bits are not 0. After the non-0 word is found, we look for the first 1 bit. This is the 5th bit of the non-zero word. So, offset = 5.
Therefore, the first free block number = 8*0+5 = 5.

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

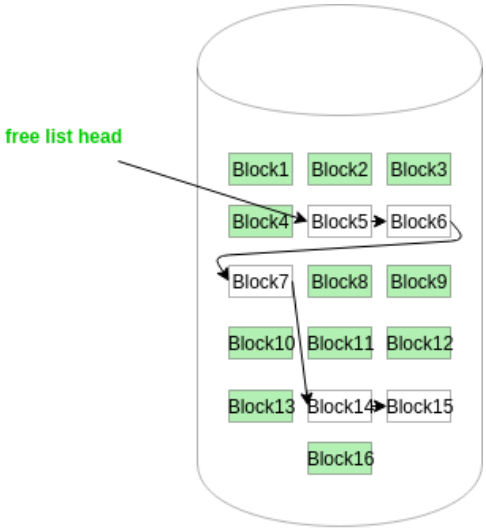


Figure - 2

In *Figure-2*, the free space list head points to Block 5 which points to Block 6, the next free block and so on. The last free block would contain a null pointer indicating the end of free list.
A drawback of this method is the I/O required for free space list traversal.

☐ **Grouping –**
This approach stores the address of the free blocks in the first free block. The first free block stores the address of some, say n free blocks. Out of these n blocks, the first n-1 blocks are actually free and the last block contains the address of next free n blocks.
An **advantage** of this approach is that the addresses of a group of free disk blocks can be found easily.
☐ **Counting –**
This approach stores the address of the first free disk block and a number n of free contiguous disk blocks that follow the first block.
Every entry in the list would contain:

1. Address of first free disk block
2. A number n

For example, *in Figure-1*, the first entry of the free space list would be: ([Address of Block 5], 2), because 2 contiguous free blocks follow block 5.

The block number can be calculated as:
*(number of bits per word) *(number of 0-values words) + offset of bit first bit 1 in the non-zero word* .

Mass-Storage Structure

Overview of Mass-Storage Structure

1 Magnetic Disks

- Traditional magnetic disks have the following basic structure:
  - One or more ***platters*** in the form of disks covered with magnetic media. ***Hard disk*** platters are made of rigid metal, while "***floppy***" disks are made of more flexible plastic.
  - 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. (Sectors also include a header and a trailer, including checksum information among other things. Larger sector sizes reduce the fraction of the disk consumed by headers and trailers, but increase internal fragmentation and the amount of disk that must be marked bad in the case of errors.)
  - 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. (Other configurations, including independent read-write heads, may speed up disk access, but involve serious technical difficulties.)
  - The storage capacity of a traditional disk drive is equal to the number of heads (i.e. the number of working surfaces), times the number of tracks per surface, times the number of sectors per track, times the number of bytes per sector. A particular physical block of data is specified by providing the head-sector-cylinder number at which it is located.

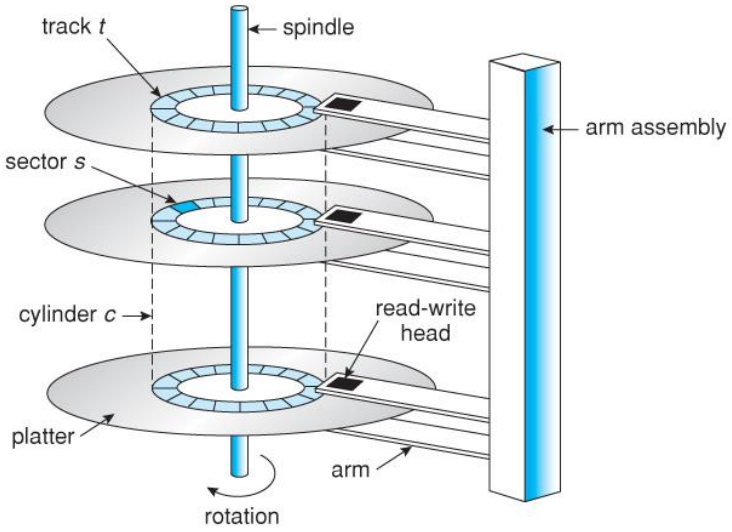


Figure 10.1 - Moving-head disk mechanism.

- In operation the disk rotates at high speed, such as 7200 rpm (120 revolutions per second.) The rate at which data can be transferred from the disk to the computer is composed of several steps:
  - The ***positioning time***, the ***seek time*** or ***random access time*** is the time required to move the heads from one cylinder to another, and for the heads to settle down after the move.
  - 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 Structure:

- The traditional head-sector-cylinder, HSC numbers are mapped to linear block addresses by numbering the first sector on the first head on the outermost track as sector 0. Numbering proceeds with the rest of the sectors on that same track, and then the rest of the tracks on the same cylinder before proceeding through the rest of the cylinders to the center of the disk. In modern practice these linear block addresses are used in place of the HSC numbers for a variety of reasons:
    1. The linear length of tracks near the outer edge of the disk is much longer than for those tracks located near the center, and therefore it is possible to squeeze many more sectors onto outer tracks than onto inner ones.
    2. All disks have some bad sectors, and therefore disks maintain a few spare sectors that can be used in place of the bad ones. The mapping of spare sectors to bad sectors in managed internally to the disk controller.
    3. Modern hard drives can have thousands of cylinders, and hundreds of sectors per track on their outermost tracks. These numbers exceed the range of HSC numbers for many (older) operating systems, and therefore disks can be configured for any convenient combination of HSC values that falls within the total number of sectors physically on the drive.
- There is a limit to how closely packed individual bits can be placed on a physical media, but that limit is growing increasingly more packed as technological advances are made.
- Modern disks pack many more sectors into outer cylinders than inner ones, using one of two approaches:
    - With ***Constant Linear Velocity, CLV,*** the density of bits is uniform from cylinder to cylinder. Because there are more sectors in outer cylinders, the disk spins slower when reading those cylinders, causing the rate of bits passing under the read-write head to remain constant. This is the approach used by modern CDs and DVDs.
    - With ***Constant Angular Velocity, CAV,*** the disk rotates at a constant angular speed, with the bit density decreasing on outer cylinders. (These disks would have a constant number of sectors per track on all cylinders.)

Disk Scheduling:

FCFS Scheduling

- ***First-Come First-Serve*** is simple and intrinsically fair, but not very efficient. Consider in the following sequence the wild swing from cylinder 122 to 14 and then back to 124:

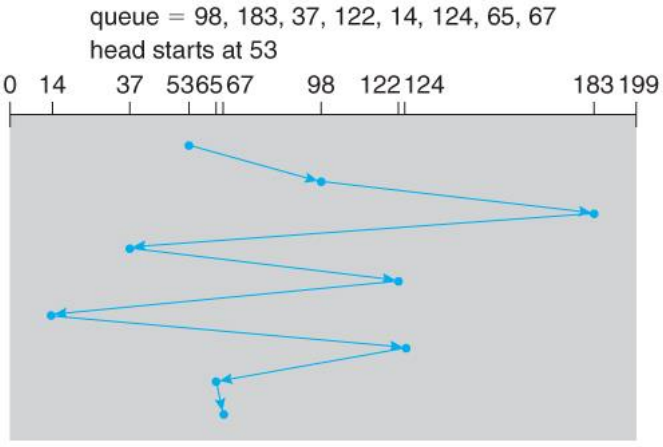


Figure 10.4 - FCFS disk scheduling.

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. Note, however that the distance could be reduced still further to 208 by starting with 37 and then 14 first before processing the rest of the requests.

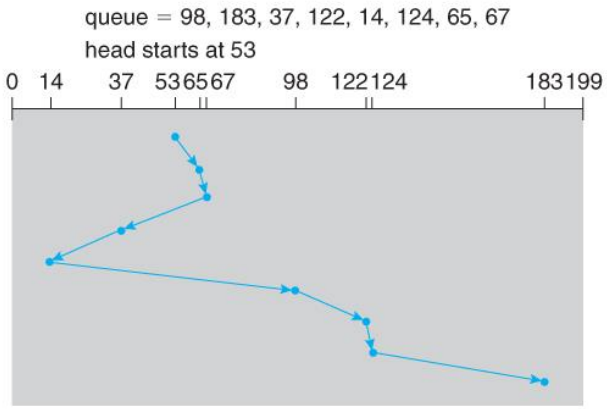


Figure 10.5 - SSTF disk scheduling.

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.

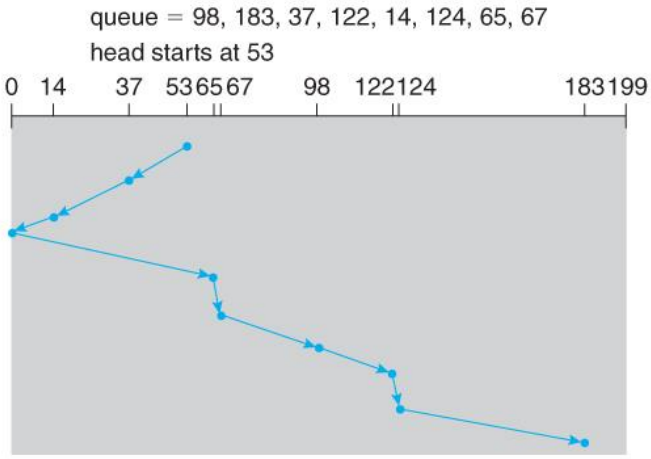


Figure 10.6 - SCAN disk scheduling.

- 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. This leads to a fairly wide variation in access times which can be improved upon.
- Consider, for example, when the head reaches the high end of the disk: Requests with high cylinder numbers just missed the passing head, which means they are all fairly recent requests, whereas requests with low numbers may have been waiting for a much longer time. Making the return scan from high to low then ends up accessing recent requests first and making older requests wait that much longer.

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:

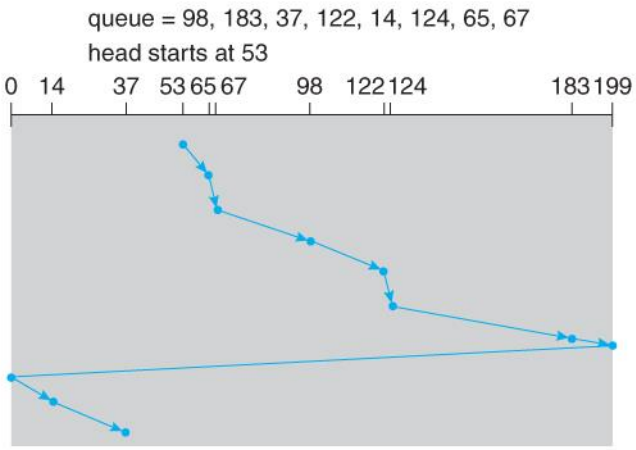


Figure 10.7 - C-SCAN disk scheduling.

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 circular form of LOOK:

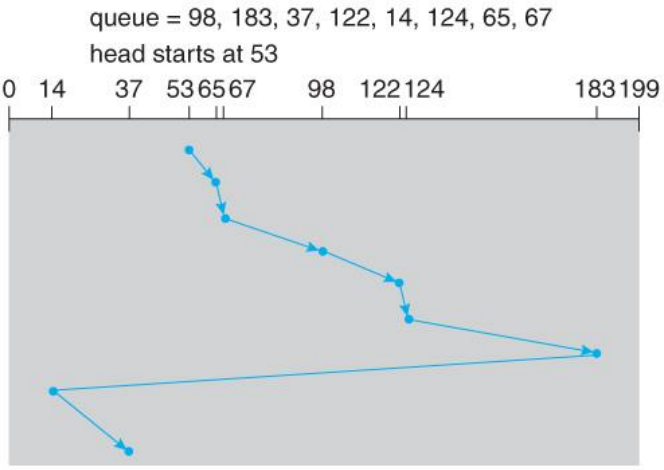


Figure 10.8 - C-LOOK disk scheduling.

Disk Management:

Disk Formatting

- Before a disk can be used, it has to be ***low-level formatted***, which means laying down all of the headers and trailers marking the beginning and ends of each sector. Included in the header and trailer are the linear sector numbers, and ***error-correcting codes, ECC,*** which allow damaged sectors to not only be detected, but in many cases for the damaged data to be recovered (depending on the extent of the damage.) Sector sizes are traditionally 512 bytes, but may be larger, particularly in larger drives.
- ECC calculation is performed with every disk read or write, and if damage is detected but the data is recoverable, then a ***soft error*** has occurred. Soft errors are generally handled by the on-board disk controller, and never seen by the OS. (See below.)
- Once the disk is low-level formatted, the next step is to partition the drive into one or more separate partitions. This step must be completed even if the disk is to be used as a single large partition, so that the partition table can be written to the beginning of the disk.
- After partitioning, then the filesystems must be ***logically formatted,*** which involves laying down the master directory information (FAT table or inode structure), initializing free lists, and creating at least the root directory of the filesystem. (Disk partitions which are to be used as raw devices are not logically formatted. This saves the overhead and disk space of the filesystem structure, but requires that the application program manage its own disk storage requirements.)

Boot Block

- Computer ROM contains a ***bootstrap*** program (OS independent) with just enough code to find the first sector on the first hard drive on the first controller, load that sector into memory, and transfer control over to it. (The ROM bootstrap program may look in floppy and/or CD drives before accessing the hard drive, and is smart enough to recognize whether it has found valid boot code or not.)
- The first sector on the hard drive is known as the ***Master Boot Record, MBR,*** and contains a very small amount of code in addition to the ***partition table.*** The partition table documents how the disk is partitioned into logical disks, and indicates specifically which partition is the ***active*** or ***boot*** partition.
- The boot program then looks to the active partition to find an operating system, possibly loading up a slightly larger / more advanced boot program along the way.
- In a ***dual-boot*** (or larger multi-boot) system, the user may be given a choice of which operating system to boot, with a default action to be taken in the event of no response within some time frame.

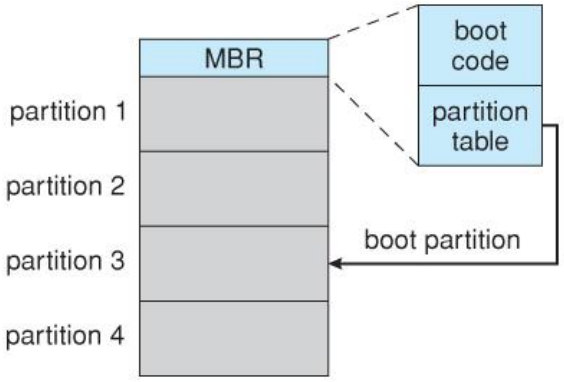


Figure 10.9 - Booting from disk in Windows 2000.

Bad Blocks

- No disk can be manufactured to 100% perfection, and all physical objects wear out over time. For these reasons all disks are shipped with a few bad blocks, and additional blocks can be expected to go bad slowly over time. If a large number of blocks go bad then the entire disk will need to be replaced, but a few here and there can be handled through other means.

Swap-Space Management:

- Modern systems typically swap out pages as needed, rather than swapping out entire processes. Hence the swapping system is part of the virtual memory management system.
- Managing swap space is obviously an important task for modern OSes.

Swap-Space Use

- The amount of swap space needed by an OS varies greatly according to how it is used. Some systems require an amount equal to physical RAM; some want a multiple of that; some want an amount equal to the amount by which virtual memory exceeds physical RAM, and some systems use little or none at all!
- Some systems support multiple swap spaces on separate disks in order to speed up the virtual memory system.

Swap-Space Location

Swap space can be physically located in one of two locations:

- As a large file which is part of the regular filesystem. This is easy to implement, but inefficient. Not only must the swap space be accessed through the directory system, the file is also subject to fragmentation issues. Caching the block location helps in finding the physical blocks, but that is not a complete fix.
- As a raw partition, possibly on a separate or little-used disk. This allows the OS more control over swap space management, which is usually faster and more efficient. Fragmentation of swap space is generally not a big issue, as the space is re-initialized every time the system is rebooted. The downside of keeping swap space on a raw partition is that it can only be grown by repartitioning the hard drive.

Swap-Space Management: An Example

- Historically OSes swapped out entire processes as needed. Modern systems swap out only individual pages, and only as needed. (For example process code blocks and other blocks that have not been changed since they were originally loaded are normally just freed from the virtual memory system rather than copying them to swap space, because it is faster to go find them again in the filesystem and read them back in from there than to write them out to swap space and then read them back.)
- In the mapping system shown below for Linux systems, a map of swap space is kept in memory, where each entry corresponds to a 4K block in the swap space. Zeros indicate free slots and non-zeros refer to how many processes have a mapping to that particular block (>1 for shared pages only.)

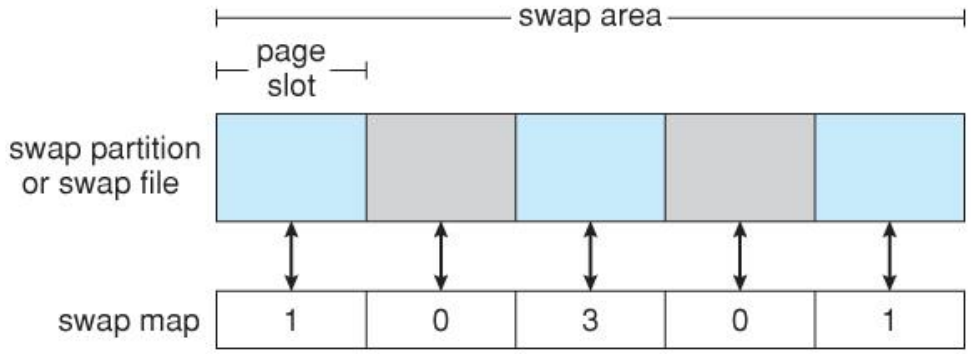


Figure 10.10 - The data structures for swapping on Linux systems.

RAID Structure:

- The general idea behind RAID is to employ a group of hard drives together with some form of duplication, either to increase reliability or to speed up operations, (or sometimes both.)
- ***RAID*** originally stood for ***Redundant Array of Inexpensive Disks,*** and was designed to use a bunch of cheap small disks in place of one or two larger more expensive ones. Today RAID systems employ large possibly expensive disks as their components, switching the definition to ***Independent*** disks.

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, (such as the ***Reed-Solomon codes***), for 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.

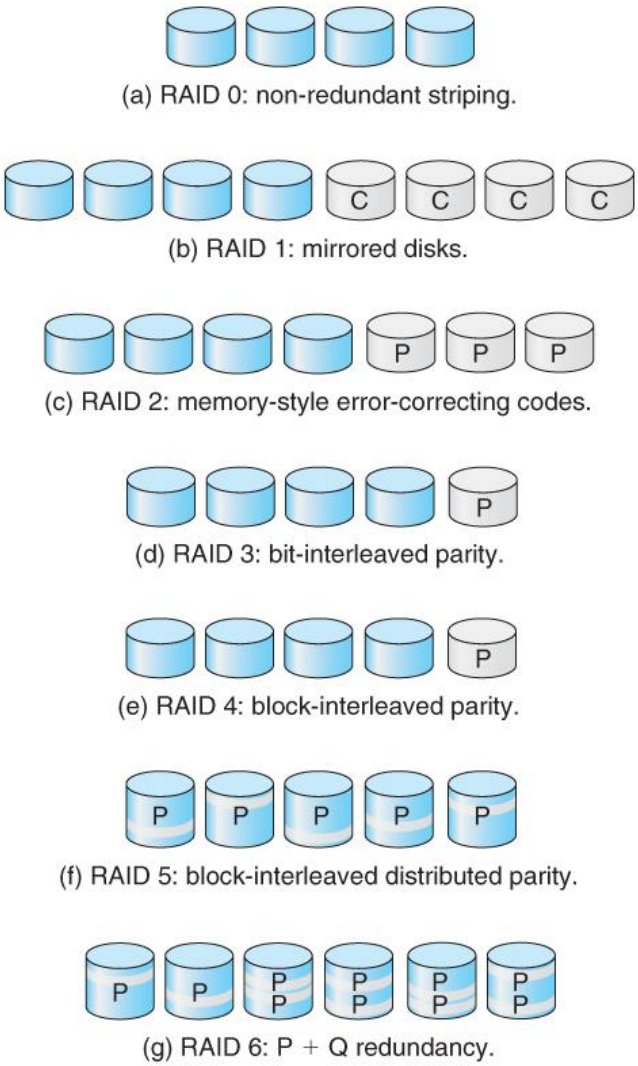


Figure 10.11 - RAID levels.

Stable-Storage Implementation:

- The concept of stable storage (first presented in chapter 6) involves a storage medium in which data is ***never*** lost, even in the face of equipment failure in the middle of a write operation.
- To implement this requires two (or more) copies of the data, with separate failure modes.
- An attempted disk write results in one of three possible outcomes:
    1. The data is successfully and completely written.
    2. The data is partially written, but not completely. The last block written may be garbled.
    3. No writing takes place at all.
- Whenever an equipment failure occurs during a write, the system must detect it, and return the system back to a consistent state. To do this requires two physical blocks for every logical block, and the following procedure:
    1. Write the data to the first physical block.
    2. After step 1 had completed, then write the data to the second physical block.
    3. Declare the operation complete only after both physical writes have completed successfully.
- During recovery the pair of blocks is examined.
    - If both blocks are identical and there is no sign of damage, then no further action is necessary.
    - If one block contains a detectable error but the other does not, then the damaged block is replaced with the good copy. (This will either undo the operation or complete the operation, depending on which block is damaged and which is undamaged.)
    - If neither block shows damage but the data in the blocks differ, then replace the data in the first block with the data in the second block. (Undo the operation.)

Because the sequence of operations described above is slow, stable storage usually includes NVRAM as a cache, and declares a write operation complete once it has been written to the NVRAM.

Tertiary-Storage Structure:

- Primary storage refers to computer memory chips; Secondary storage refers to fixed-disk storage systems (hard drives); And ***Tertiary Storage*** refers to ***removable media,*** such as tape drives, CDs, DVDs, and to a lesser extend floppies, thumb drives, and other detachable devices.

Tertiary-Storage Devices

Removable Disks

- Removable magnetic disks (e.g. floppies) can be nearly as fast as hard drives, but are at greater risk for damage due to scratches. Variations of removable magnetic disks up to a GB or more in capacity have been developed. (Hot-swappable hard drives?)

Tapes

- Tape drives typically cost more than disk drives, but the cost per MB of the tapes themselves is lower.
- Tapes are typically used today for backups, and for enormous volumes of data stored by certain scientific establishments. (E.g. NASA's archive of space probe and satellite imagery, which is currently being downloaded from numerous sources faster than anyone can actually look at it.)
- Robotic tape changers move tapes from drives to archival tape libraries upon demand.
- (Never underestimate the bandwidth of a station wagon full of tapes rolling down the highway!)