MAHAVEER INSTITUTE OF SCIENCE AND TECHNOLOGY

(AN UGC AUTONOMOUS INSTITUTION)

Approved by AICTE, Affiliated to JNTUH, Accredited by NAAC with 'A' Grade
Recognized Under Section 2(f) of UGC Act 1956, ISO 9001:2015 Certified
Vyasapuri, Bandlaguda, Post: Keshavgiri, Hyderabad- 500 005, Telangana, India.
https://www.mist.ac.in E-mail:principal.mahaveer@gmail.com, Mobile: 8978380692

ESTD : 2001

Department of Computer Science and Engineering (AIML)

(R22) OPERATING SYSTEM

Lecture Notes

B. Tech II YEAR – I SEM

Prepared by

SANGYAM SOUNDARYA
(Assistant Professor)
Dept.CSE(AIML)

OPERATING SYSTEMS

B.Tech. II Year I Sem.	**L**	**T**	**P**	**C**
	3	**0**	**0**	**3**

Prerequisites:

1. A course on "Computer Programming and Data Structures".
2. A course on "Computer Organization and Architecture".

Course Objectives:

- Introduce operating system concepts (i.e., processes, threads, scheduling, synchronization, deadlocks, memory management, file and I/O subsystems and protection)
- Introduce the issues to be considered in the design and development of operating system
- Introduce basic Unix commands, system call interface for process management, interprocess communication and I/O in Unix

Course Outcomes:

- Will be able to control access to a computer and the files that may be shared
- Demonstrate the knowledge of the components of computers and their respective roles in computing.
- Ability to recognize and resolve user problems with standard operating environments.
- Gain practical knowledge of how programming languages, operating systems, and architectures interact and how to use each effectively.

UNIT - I

Operating System - Introduction, Structures - Simple Batch, Multiprogrammed, Time-shared, Personal Computer, Parallel, Distributed Systems, Real-Time Systems, System components, Operating System services, System Calls

Process - Process concepts and scheduling, Operations on processes, Cooperating Processes, Threads

UNIT - II

CPU Scheduling - Scheduling Criteria, Scheduling Algorithms, Multiple -Processor Scheduling. System call interface for process management-fork, exit, wait, waitpid, exec

Deadlocks - System Model, Deadlocks Characterization, Methods for Handling Deadlocks, Deadlock Prevention, Deadlock Avoidance, Deadlock Detection, and Recovery from Deadlock

UNIT - III

Process Management and Synchronization - The Critical Section Problem, Synchronization Hardware, Semaphores, and Classical Problems of Synchronization, Critical Regions, Monitors

Interprocess Communication Mechanisms: IPC between processes on a single computer system, IPC between processes on different systems, using pipes, FIFOs, message queues, shared memory.

UNIT - IV

Memory Management and Virtual Memory - Logical versus Physical Address Space, Swapping, Contiguous Allocation, Paging, Segmentation, Segmentation with Paging, Demand Paging, Page Replacement, Page Replacement Algorithms.

UNIT - V

File System Interface and Operations -Access methods, Directory Structure, Protection, File System Structure, Allocation methods, Free-space Management. Usage of open, create, read, write, close, lseek, stat, ioctl system calls.

TEXT BOOKS:

1. Operating System Principles- Abraham Silberchatz, Peter B. Galvin, Greg Gagne 7th Edition, John Wiley.
2. Advanced programming in the UNIX environment, W.R. Stevens, Pearson education.

REFERENCE BOOKS:

1. Operating Systems- Internals and Design Principles, William Stallings, Fifth Edition–2005, Pearson Education/PHI
2. Operating System A Design Approach- Crowley, TMH.
3. Modern Operating Systems, Andrew S. Tanenbaum 2nd edition, Pearson/PHI
4. UNIX programming environment, Kernighan and Pike, PHI/ Pearson Education
5. UNIX Internals -The New Frontiers, U. Vahalia, Pearson Education.

UNIT I

INTRODUCTION

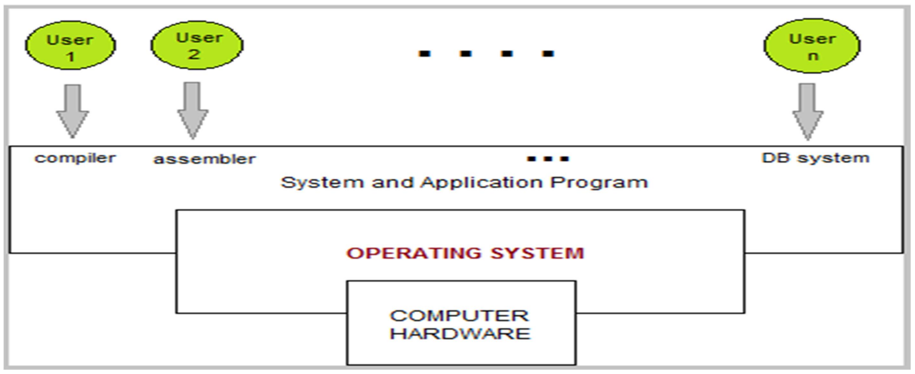
OPERATING SYSTEM

A computer system has many resources (hardware and software), which may be required to complete a task. The commonly required resources are input/output devices, memory, file storage space, CPU, etc. The operating system acts as a manager of the above resources and allocates them to specific programs and users, whenever necessary to perform a particular task. Therefore the operating system is the resource manager i.e. it can manage the resource of a computer system internally. The resources are processor, memory, files, and I/O devices.

In simple terms, an operating system is an interface between the computer user and the machine.

It is very important for you that every computer must have an operating system in order to run other programs. The operating system mainly coordinates the use of the hardware among the various system programs and application programs for various users.
An operating system acts similarly like government means an operating system performs no useful function by itself; though it provides an environment within which other programs can do useful work. Below we have an abstract view of the components of the computer system.

Four Components of a Computer System



- The Computer Hardware contains a central processing unit(CPU), the memory, and the input/output (I/O) devices and it provides the basic computing resources for the system.
- The Application programs like spreadsheets, Web browsers, word processors, etc. are usedto define the ways in which these resources are used to solve the computing problems of the users. And the System program mainly consists of compilers, loaders, editors, OS, etc.

- The Operating System is mainly used to control the hardware and coordinate its use among the various application programs for the different users.
- Basically, Computer System mainly consists of hardware, software, and data. OS is mainly designed in order to serve two basic purposes:
- The operating system mainly controls the allocation and use of the computing System's resourcesamong the various user and tasks.
- It mainly provides an interface between the computer hardware and the programmer thatsimplifies and makes feasible for coding, creation of application programs and debugging

Two Views of Operating System
User's View System
View

Operating System: User View
The user view of the computer refers to the interface being used. Such systems are designed for one user to monopolize its resources, to maximize the work that the user is performing. In these cases,the operating system is designed mostly for ease of use, with some attention paid to performance, and none paid to resource utilization.

Operating System: System View
The operating system can be viewed as a resource allocator also. A computer system consists of many resources like - hardware and software - that must be managed efficiently. The operating system acts as the manager of the resources, decides between conflicting requests, controls the execution of programs, etc.

Types of Operating System
Given below are different types of Operating System:

- Simple Batch System
- Multi programmed
- Time-Shared
- Personal Computer
- Parallel
- Distributed Systems
- Real Time Systems

STRUCTURES

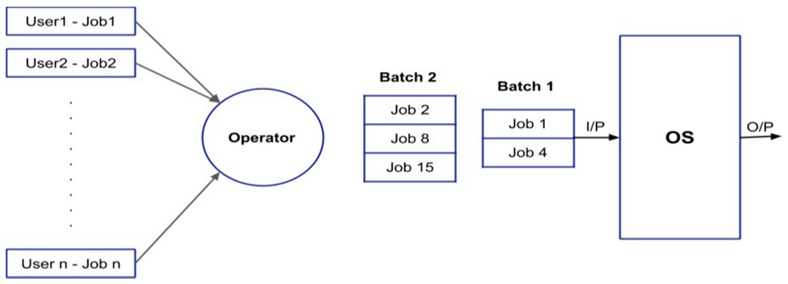
SIMPLE BATCH

In a Batch Operating System, the similar jobs are grouped together into batches with the help of some operator and these batches are executed one by one. For example, let us assume that we have 10 programs that need to be executed. Some programs are written in C++, some in C and rest in Java. Now, every time when we run these programs individually then we will have to load the compiler of that particular language and then execute the code. But what if we make a batch of these 10 programs. The benefit with this approach is that, for the C++ batch, you need to load the compiler only once. Similarly, for Java and C, the compiler needs to be loaded only once andthe whole batch gets executed.

The following batch processing actions are performed by a computer system running this operating system:

- A job is a single unit made up of a pre-programmed set of data, commands, and programs.
- The orders are processed in the order in which these are received, meaning first come, first served.
- These jobs are saved in memory and run without the need for any manual input.
- The OS releases memory after a job is completed successfully.

The following image describes the working of a Batch Operating System.



The OS keeps track of the number of jobs and executes them one by one. Jobs are processed in the order in which they are received. A batch is defined for each work set. When a job is completed, the memory associated with it is released, and the work's output is sent to an output spool for printing or processing later. In a batch operating system, user engagement is limited. When the system takes over the task from the user, the person is free to do other things. You can also make use of the batch processing system to make changes to data in any transactions or records.

Examples

- Payroll System
- Bank Invoice System
- Transactions Process
- Daily Report
- Research Segment
- Billing System

Advantages:

- The overall time taken by the system to execute all the programmes will be reduced.
- The Batch Operating System can be shared between multiple users.

Disadvantages:

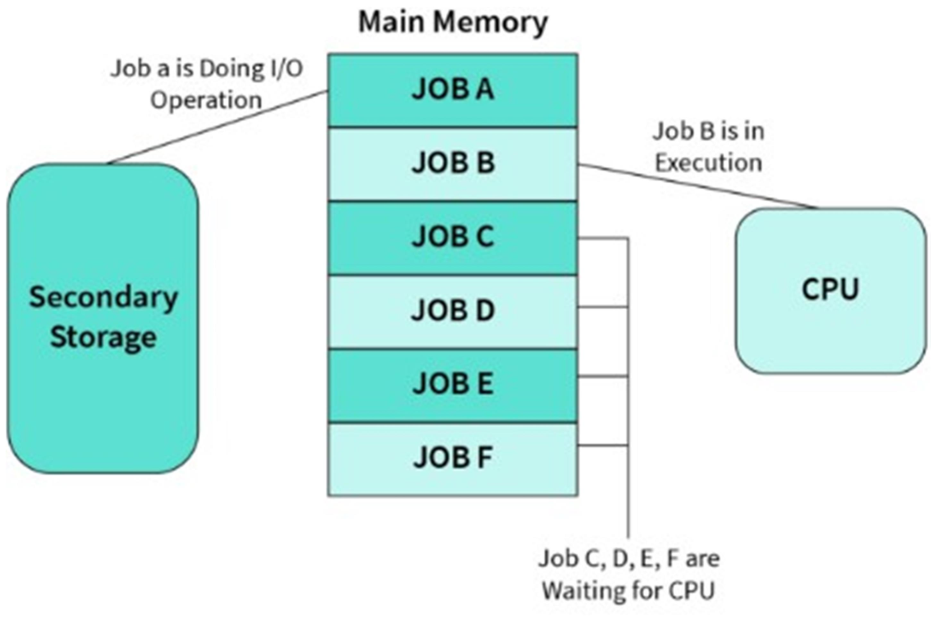
- Manual interventions are required between two batches.
- The CPU utilization is low because the time taken in loading and unloading of batches is veryhigh as compared to execution time.

MULTI-PROGRAMMING

On a single processor computer, a multiprogramming OS can run many programs. In a multiprogramming OS, if one program must wait for an input/output transfer, the other programmes are ready to use the CPU. As a result, different jobs may have to split CPU time. However, their jobs are not scheduled to be completed at the same time.
Multiprogramming's main purpose is to manage all of the system's resources. The file system, transient area, command processor, and I/O control system are the main components of a multiprogramming system.

Working of multi-programming

In the multiprogramming system, multiple users can complete their tasks at the same time, and they can be saved in the main memory. While a programme is performing I/O operations, the CPU may distribute time to other applications while in idle mode.



Multiprogramming Operating System

While one application waits for an I/O transfer, another is always ready to use the processor, and multiple programmes may share CPU time. Although not all tasks are executed at the same time, there may be multiple jobs operating on the processor at the same time, with parts of other processes first being executed, followed by another segment, and so on. As a result, a multiprogramming system's overall purpose is to keep the CPU busy unless and until some tasks in the job pool become available. As a result, a single processor computer may run multiple programmes, and the CPU is never idle

Examples

- Apps like office, chrome, etc.
- Microcomputers like MP/M, XENIX, and ESQview.
- Windows O/S
- UNIX O/S

Advantages

- High and efficient CPU utilization.
- User feels that many programs are allotted CPU almost simultaneously.

Disadvantages

- CPU scheduling is required.
- To accommodate many jobs in memory, memory management is required

TIME-SHARING OPERATING SYSTEM

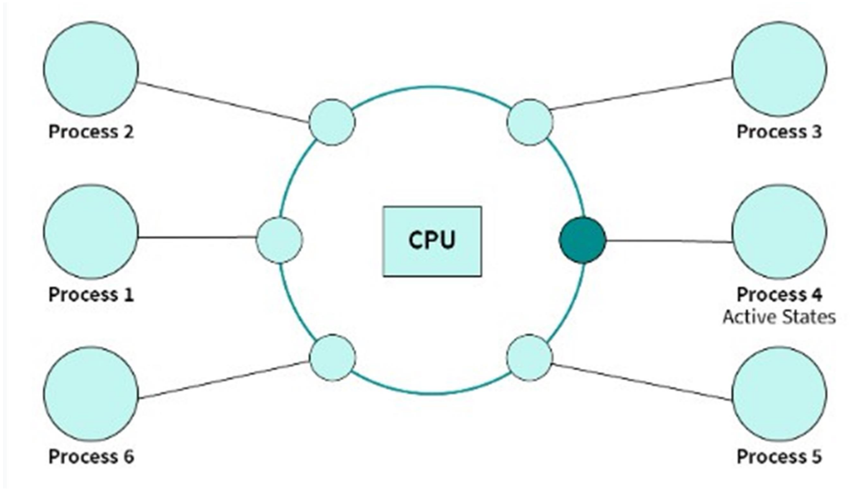
Time-sharing is a logical extension of multiprogramming. The CPU executes multiple jobs by switching, among them, but the switches occur so frequently that the users can interact with each program while it is running. An interactive computer provides direct communication between the user and the system. The user gives instructions to the OS or a program directly, using hardware, and waits for results.

A time-shared operating system uses CPU scheduling and multiprogramming to provide each user with a small portion of a time-shared computer. Each user has at least one separate program in memory. When a process executes, it executes for only a short time before it either finishes or needs to perform input/output. In time-sharing operating systems several jobs must be kept simultaneously in memory, so the system must have memory management and protection.

Working of timesharing operating system

In a Multi-tasking Operating System, more than one processes are being executed at a particular time with the help of the time-sharing concept. So, in the time-sharing environment, we decide a time that is called time quantum and when the process starts its execution then the execution continues for only that amount of time and after that, other processes will be given chance for that amount of time only. In the next cycle, the first process will again come for its execution and itwill be executed for that time quantum only and again next process will come. This process will continue.

The following image describes the working of a Time-Sharing Operating System.



Advantages:
Since equal time quantum is given to each process, so each process gets equal opportunity toexecute.
The CPU will be busy in most of the cases and this is good to have case.
Disadvantages:
1. Process having higher priority will not get the chance to be executed first because the equalopportunity is given to each process.

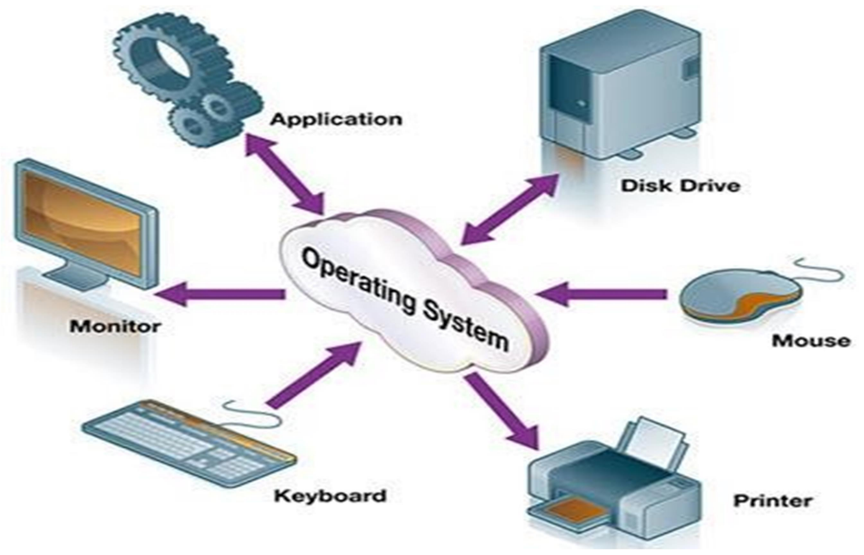
Examples

- Windows 2000 server
- Windows NT server
- Unix
- Linux

PERSONAL COMPUTER OPERATING SYSTEM

Personal computer operating system provides a good interface to a single user.

- Personal computer operating systems are widely used for word processing, spreadsheets and Internetaccess.
- Personal computer operating system is made only for personal. You can say that your laptops, computer systems, tablets etc. are your personal computers and the operating system such as windows 7; windows 10, android, etc. are your personal computer operating system.
- you can use your personal computer operating system for your personal purposes, for example, to chatting with your friends using some social media sites, reading some articles from internet, making some projects through Microsoft PowerPoint or any other, designing your website, programming something, watching some videos and movies, listening to some songs and many more.



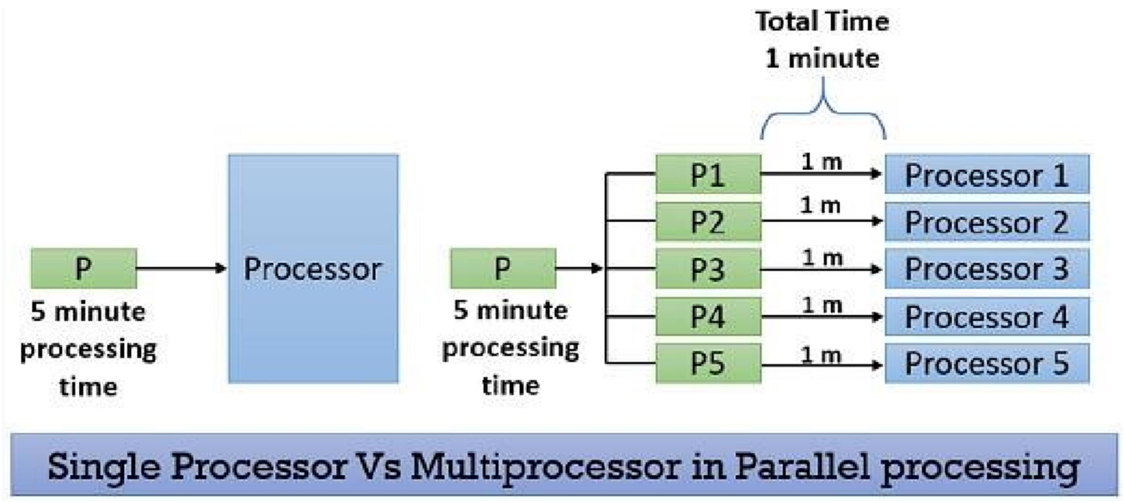
A Personal Operating System is incredibly useful. By habituating high-value practices, you make decisions about how you want to respond to life in advance. This makes life simpler. For instance, having a morning routine means not burning precious creative energy in the first few hours of your day by trying to decide how to organize your time. There are countless micro-decisions that you make every day that can be removed by adopting a broader philosophy for how you want to live your life.

PARALLEL PROCESSING (MULTIPROCESSOR OPERATING SYSTEM)

Multiprocessor operating system utilizes multiple processors, which are connected with physical memory, computer buses, clocks, and peripheral devices (touchpad, joystick, etc). The main objective of using a multiprocessor OS is to consume high computing power and increase the execution speed of the system.

Following are four major components, used in the Multiprocessor Operating System:

1. **CPU** – capable to access memories as well as controlling the entire I/O tasks.
2. **Input Output Processor** – I/P processor can access direct memories, and every I/O processors have to be responsible for controlling all input and output tasks.
3. **Input/output Devices** – These devices are used for inserting the input commands, and producing output after processing.
4. **Memory Unit** – Multiprocessor system uses the two types of memory modules - shared memory and distributed shared memory.



Advantages

- It saves time and money as many resources working together will reduce the time and cutpotential costs.
- It can be impractical to solve larger problems on Serial Computing.
- It can take advantage of non-local resources when the local resources are finite.
- Serial Computing 'wastes' the potential computing power, thus Parallel Computing makesbetter work of the hardware.

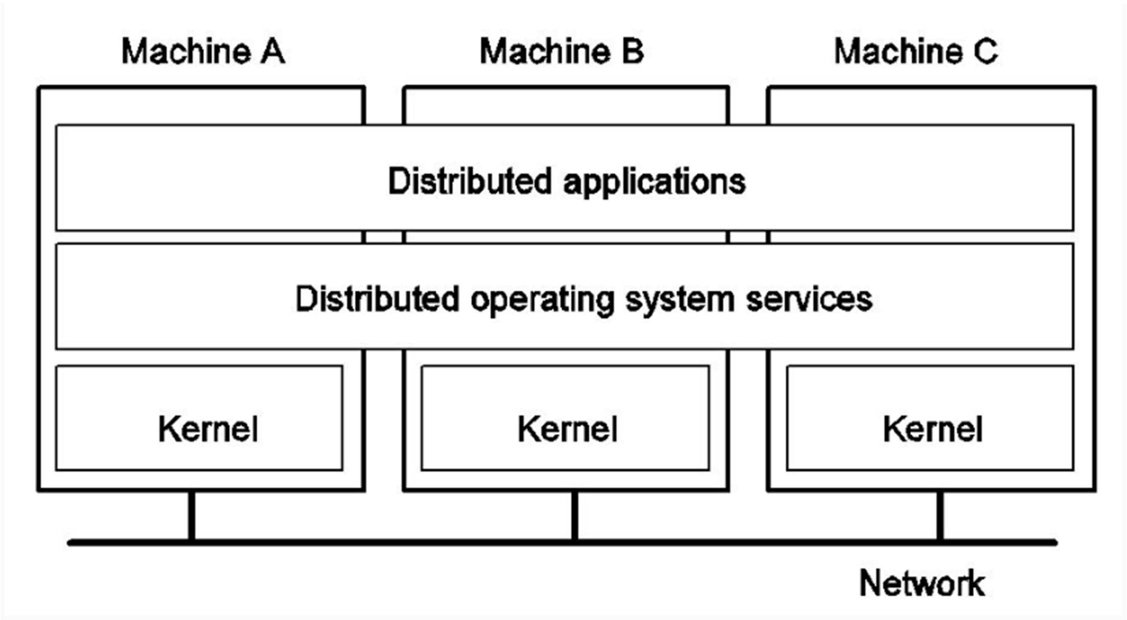
Disadvantages

- It addresses such as communication and synchronization between multiple sub- tasks andprocesses which is difficult to achieve.
- The algorithms must be managed in such a way that they can be handled in a parallelmechanism.
- The algorithms or programs must have low coupling and high cohesion. But it's difficult tocreate such programs.
- More technically skilled and expert programmers can code a parallelism- based program well.

DISTRIBUTED OPERATING SYSTEM

A distributed operating system allows the distribution of entire systems on the couples of center processors, and it serves on multiple real-time products as well as multiple users. All processors are connected by valid communication mediums such as high-speed buses and telephone lines, and in which every processor contains its local memory along with other local processors.

The Distributed operating systems are also known as loosely coupled systems. They involve multiple computers, nodes, and sites. These components are linked to each other with LAN/WAN lines. A distributed OS is capable of sharing its computational capacity and I/O files while allowing virtual machine abstraction to users.



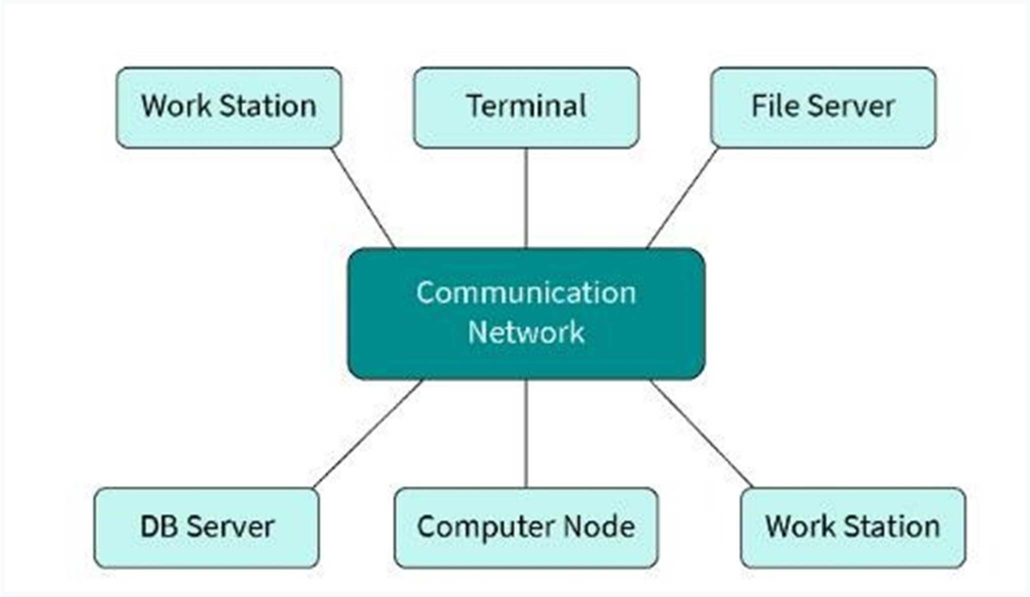


Fig: Network in Distributed Operating system

Examples

- Solaris
- OSF/1
- Micros
- DYNIX
- Locus
- Mach

Advantages

- A distributed operating system may share all resources from one site to another, increasing data availability across the entire system.
- It reduces the probability of data corruption because all data is replicated across all sites.
- The entire system operates independently of one another, and as a result, if one site crashes, the entire system does not halt.
- A distributed operating system is an open system since it may be accessed from both local and remote locations.
- It helps in the reduction of data processing time.
- Most distributed systems are made up of several nodes that interact to make them fault-tolerant. If a single machine fails, the system remains operational.

Disadvantages

- The system must decide which jobs must be executed when they must be executed, and where they must be executed. A scheduler has limitations, which can lead to underutilized hardware and unpredictable runtimes.
- It is hard to implement adequate security in a distributed operating system since the nodes and connections must be secured.
- The database connected to a DOS is relatively complicated and hard to manage in contrast to a single-user system.
- The underlying software is extremely complex and is not understood very well compared to other systems.
- The more widely distributed a system is, the more communication latency can be expected. As a result, teams and developers must choose between availability, consistency, and latency.
- Gathering, processing, presenting, and monitoring hardware use metrics for big clusters can be a real issue.

Examples

- Solaris
- OSF/1
- Micros
- DYNIX
- Locus
- Mach

REAL TIME OPERATING SYSTEM

It is developed for real-time applications where data should be processed in a fixed, small duration of time. It is used in an environment where multiple processes are supposed to be accepted and processed in a short time. RTOS requires quick input and immediate response, e.g., in a petroleum refinery, if the temperate gets too high and crosses the threshold value, there should be an immediateresponse to this situation to avoid the explosion. Similarly, this system is used to control scientific instruments, missile launch systems, traffic lights control systems, air traffic control systems, etc.

This system is further divided into two types based on the time constraints:

Hard Real-Time Systems:
These are used for the applications where timing is critical or response time is a major factor; even a delay of a fraction of the second can result in a disaster. For example, airbags and automatic parachutes that open instantly in case of an accident. Besides this, these systems lack virtual memory.

Soft Real-Time Systems:
These are used for application where timing or response time is less critical. Here, the failure to meetthe deadline may result in a degraded performance instead of a disaster. For example, video surveillance (cctv), video player, virtual reality, etc. Here, the deadlines are not critical for every taskevery time.

Advantages

- The output is more and quick owing to the maximum utilization of devices and system
- Task shifting is very quick, e.g., 3 microseconds, due to which it seems that several tasks areexecuted simultaneously
- Gives more importance to the currently running applications than the queued application
- It can be used in embedded systems like in transport and others.
- It is free of errors.
- Memory is allocated appropriately.

Disadvantages

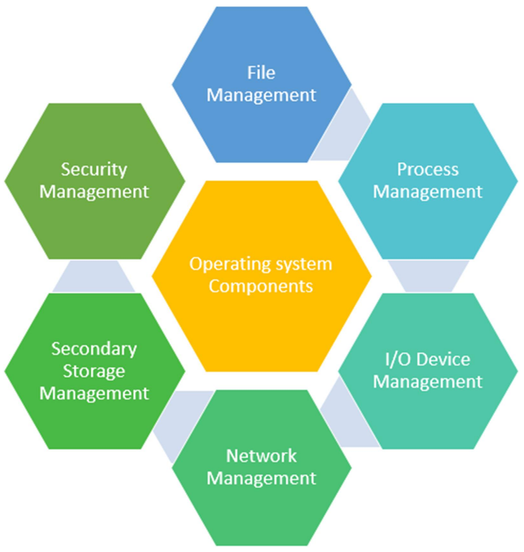
- A fewer number of tasks can run simultaneously to avoid errors.
- It is not easy for a designer to write complex and difficult algorithms or proficient programsrequired to get the desired output.
- Specific drivers and interrupt signals are required to respond to interrupts quickly.
- It may be very expensive due to the involvement of the resources required to work.

SYSTEM COMPONENTS

An operating system is a large and complex system that can only be created by partitioning into small pieces. These pieces should be a well-defined portion of the system, which carefully defined inputs, outputs, and functions.

Although Mac, Unix, Linux, Windows, and other OS do not have the same structure, most of the operating systems share similar OS system components like File, Process, Memory, I/O device management.

Let's see each of these components in detail.



File Management

A file is a set of related information which is should define by its creator. It commonly represents programs; both source and object forms, and data. Data files can be numeric, alphabetic, or alphanumeric.

Function of file management in OS:

The operating system has the following important given activities in connections with file management:

- File and directory creation and deletion.
- For manipulating files and directories.
- Mapping files onto secondary storage.
- Backup files on stable storage media

Process Management

- The process management component is a procedure for managing the many processes that are running simultaneously on the operating system. Every software application program has one or more processes associated with them when they are running.

- For example, when you use a browser like Google Chrome, there is a process running for that browser program. The OS also has many processes running, which performing various functions.
- All these processes should be managed by process management, which keeps processes for running efficiently. It also uses memory allocated to them and shutting them down when needed.
- The execution of a process must be sequential so, at least one instruction should be executed on behalf of the process.

Functions of process management in OS:

The following are functions of process management.

- Process creation and deletion.
- Suspension and resumption.
- Synchronization process
- Communication process

I/O Device Management

One of the important use of an operating system that helps you to hide the variations of specific hardware devices from the user.

Functions of I/O management in OS:

- It offers buffer caching system
- It provides general device driver code
- It provides drivers for particular hardware devices.
- I/O helps you to know the individualities of a specific device.

Network management

Network management is the process of administering and managing computer networks. It includes performance management, fault analysis, provisioning of networks, and maintaining the quality of service.

A distributed system is a collection of computers/processors that never share their own memory or a clock. In this type of system, all the processors have their local Memory, and the processors communicate with each other using different communication lines, like fiber optics or telephone lines.

The computers in the network are connected through a communication network, which can be configured in a number of different ways. With the help of network management, the network can be fully or partially connected, which helps users to design routing and connection strategies that overcome connection and security issues.

Functions of Network management:

- Distributed systems help you to various computing resources in size and function. They may involve microprocessors, minicomputers, and many general-purpose computer systems.
- A distributed system also offers the user access to the various resources the network shares.
- It helps to access shared resources that help computation to speed-up or offers data availability and reliability.

Main Memory management

Main Memory is a large array of storage or bytes, which has an address. The memory management process is conducted by using a sequence of reads or writes of specific memory addresses.

In order to execute a program , it should be mapped to absolute addresses and loaded inside the Memory. The selection of a memory management method depends on several factors.

However, it is mainly based on the hardware design of the system. Each algorithm requires corresponding hardware support. Main Memory offers fast storage that can be accessed directly by the CPU. It is costly and hence has a lower storage capacity.
However, for a program to be executed, it must be in the main Memory.

Functions of Memory management in OS:

An Operating System performs the following functions for Memory Management:

- It helps you to keep track of primary memory.
- Determine what part of it are in use by whom, what part is not in use.
- In a multiprogramming system, the OS takes a decision about which process will get Memory and how much.
- Allocates the memory when a process requests
- It also de-allocates the Memory when a process no longer requires or has been terminated.

Secondary-Storage Management

The most important task of a computer system is to execute programs. These programs, along with the data, help you to access, which is in the main memory during execution.

This Memory of the computer is very small to store all data and programs permanently. The computer system offers secondary storage to back up the main Memory. Today modern computers use hard drives/SSD as the primary storage of both programs and data. However, the secondary storage management also works with storage devices, like a USB flash drive, and CD/DVD drives.

Programs like assemblers, compilers, stored on the disk until it is loaded into memory, and then use the disk as a source and destination for processing.

Functions of Secondary storage management in OS:

Here, are major functions of secondary storage management in OS:

- Storage allocation
- Free space management
- Disk scheduling

Security Management

The various processes in an operating system need to be secured from each other's activities. For that purpose, various mechanisms can be used to ensure that those processes which want to operate files, memory CPU, and other hardware resources should have proper authorization from the operating system.

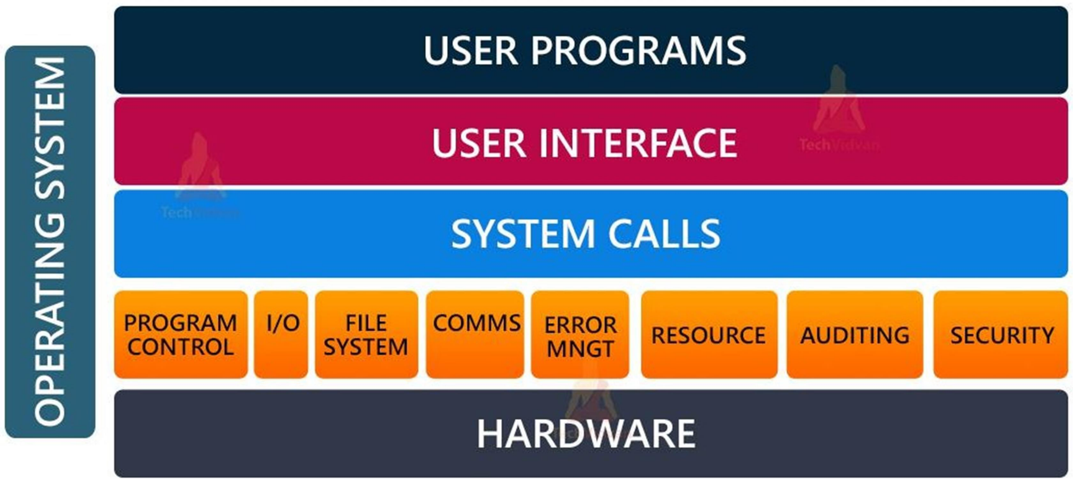
For example, Memory addressing hardware helps you to confirm that a process can be executed within its own address space. The time ensures that no process has control of the CPU without renouncing it.

OPERATING SYSTEM SERVICES

An operating system executes programs and makes the process of solving them easier. It also makes the computer system easier to use and helps the user use the computer hardware efficiently. Apart from these, it also provides an array of services both to the users and the programs.

Services of Operating System

The OS is the resource manager of a system. Thus, there are multiple services it provides in order to have an efficient system that can utilize these resources to the fullest. Following are the services provided by operating systems:



1. User Interface

- An interface is required to communicate with the user. Then it can either be a Command Line Interface or a Graphical User Interface.

- There is also a third kind that is Batch Based Interface which is usually overlooked. It uses commands to manage the commands entered into files and then these files get executed.
- As for the first two, *Command Line Interface* commonly uses text commands input by the users to interact with the system. These commands can also be given using a terminal emulator, or remote shell client.
- A *Graphical User Interface (GUI)* allows users to interact with the computer system or any other computer-controlled device.
- A GUI usually consists of all the graphical icons displayed on a computer screen, visual indicators like widgets, texts, labels, and text navigation. Thus, a user can directly perform actions with a click of the mouse or keyboard.

2. Program Execution

The OS loads a program into memory and then executes that program. It also makes sure that once started that program can end its execution, either normally or forcefully. The major steps during program management are:

- Loading a program into memory.
- Executing the program.
- Making sure the program completes its execution.
- Providing a mechanism for:
  1. process synchronization.
  2. Process communication.
  3. Deadlock handling.

3. File System Manipulation

A program is read and then written in the form of directories and files. These files can be stored on the storage disk for the long term. The OS allows the users to create and delete files, duplicate these files, and search files and their information or properties.

It also does permission management for these files i.e., allowing or denying access to these files or directories based on the file ownership.

4. I/O Operations

I/O operations are required during the execution of a program. To maintain efficiency and protection of the program, users cannot directly govern the I/O devices instead the OS allows to read or write operations with any file using the I/O devices and also allows access to any required I/O device when required.

5. Communication systems

Processes need to swap information among themselves. These processes can be from the same computer system or different computer systems as long as they are connected through communication lines in a network.

This can be done with the help of OS support using shared memory or message passing. The OS also manages routing, connection strategies, and the problem of contention and security.

6. Resource Allocation

When multiple users or multiple jobs run on a system concurrently, the resources need to be allocated equally to all of them.

CPU scheduling is used to allocate resources fairly and for the better utilization of the CPU. These resources may include CPU cycles, main memory storage, file storage, and I/O devices.

7. Error Detection

Errors may occur in any of the resources like CPU, I/O devices, or memory hardware. The OS keeps a lookout for such errors, corrects errors when they occur, and makes sure that the system works uninterruptedly.

8. Accounting

This keeps a check of which resource is being used by a user and for how long it is being used. This is usually done for statistical purposes.

9. Protection and Security

This is to ensure the safety of the system. Thus, user authentication is required to access a system. It is also necessary to protect a process from another when multiple processes are running on a system at the same time.

The OS controls the access to the resources, protects the I/O devices from invalid access, and provides authentication through passwords.

10. Command Interpretation

The OS understands and interprets commands that are input by the user and displays the input accordingly. Multiple command interpreters can be supported by an OS and they do not necessarily need to run in kernel mode.

If the interpreter is separate from the kernel then you can modify the interpreter and prevent any unauthorized access into the system.

11. Resource Manager

The OS manages resources such as processor, memory, I/O devices etc efficiently. It allocates resources to processes and administers running programs to ensure user satisfaction.

It also decides the time at which a program should run, the amount of memory allocated for execution, where to save a file, and much more.

Apart from these basics, there are some more services the OS provides that are a part of resource management. These are:

a. Process Management
when multiple processes run simultaneously on a system, the OS help manage them in order to enhance system performance. Apart from this, the OS also manages printer spooling, virtual memory, swapping, etc. and CPU scheduling is used to allocate resource

Major activities regarding process management are:
1. Creates and deletes processes.
2. Suspends and re-activates processes.
3. A mechanism for

- process synchronization
- process communication
- deadlock handling

b. Main-Memory Management
This deals with the primary, secondary, and virtual memory and increases the amount of memory available for each process. In order to perform program execution, it is necessary to load the program into the main memory.

The OS ensures that there is enough memory for a process to execute and different locations of memory are being used properly for effective execution of processes.

During execution, the memory manager tracks available memory locations, where processes can be allocated or unallocated.

c. Secondary-Storage Management
Primary and cache storage are volatile memories thus, the data is lost once power is turned off.
Moreover, main memory cannot accommodate all data and programs so secondary storage is needed as a backup like tape drives, disk drives, and other media. This provides easy access to the files and folders in the secondary storage using disk scheduling algorithms.

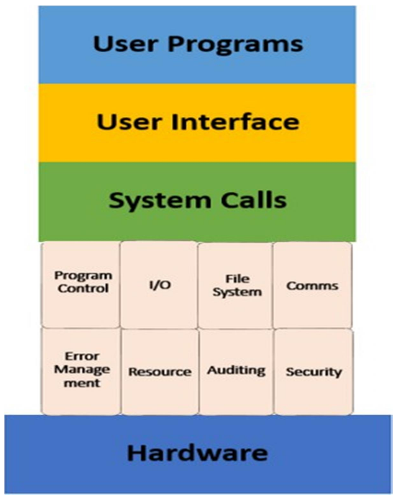
OS manages free space on the secondary storage devices, allocates storage space to new files, schedules memory access requests, and creates and deletes files.

d. Network Management
The OS works as a network resource manager when multiple systems form a network or in a distributed system. The processors communicate through network lines called networks. Today's networks are usually based on client-server configuration where a client is the program running on the local machine requesting a service and a server is the program running on the remote machine providing a service.

SYSTEM CALLS

A **system call** is a mechanism that provides the interface between a process and the operating system. It is a programmatic method in which a computer program requests a service from the kernel of the OS.

System call offers the services of the operating system to the user programs via API (Application Programming Interface). System calls are the only entry points for the kernel system.



Example of System Call

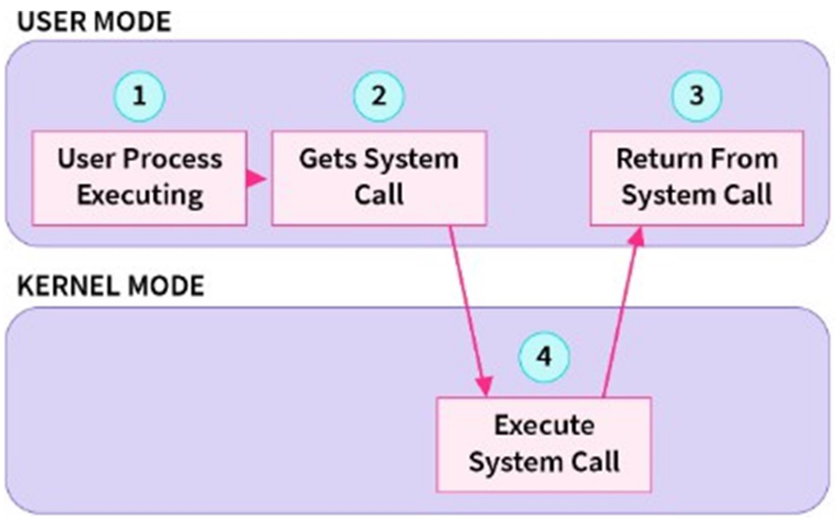
For example if we need to write a program code to read data from one file, copy that data into another file. The first information that the program requires is the name of the two files, the input and out0put files.

In an interactive system, this type of program execution requires some system calls by OS.

- First call is to write a prompting message on the screen
- Second, to read from the keyboard, the characters which define the two files.

How System Call Works?

Here are the steps for System Call in OS:



As you can see in the above-given System Call example diagram.

Step 1) The processes executed in the user mode till the time a system call interrupts it.
Step 2) After that, the system call is executed in the kernel-mode on a priority basis.
Step 3) Once system call execution is over, control returns to the user mode.,
Step 4) The execution of user processes resumed in Kernel mode.

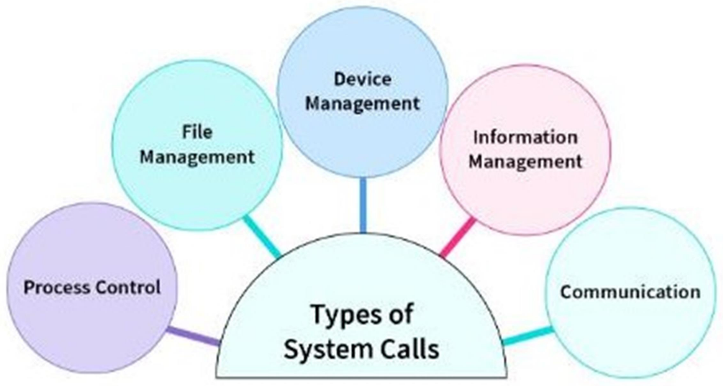
Why do you need System Calls in OS?

Following are situations which need system calls in OS:

- Reading and writing from files demand system calls.
- If a file system wants to create or delete files, system calls are required.
- System calls are used for the creation and management of new processes.

- Network connections need system calls for sending and receiving packets.
- Access to hardware devices like scanner, printer, need a system call.

Types of system calls



Process Control

This system calls perform the task of process creation, process termination, etc.

Functions:

- End and Abort
- Load and Execute
- Create Process and Terminate Process
- Wait and Signal Event
- Allocate and free memory

File Management

File management system calls handle file manipulation jobs like creating a file, reading, and writing, etc.

Functions:

- Create a file
- Delete file
- Open and close file
- Read, write, and reposition
- Get and set file attributes

Device Management

Device management does the job of device manipulation like reading from device buffers, writing into device buffers, etc.

Functions:

- Request and release device
- Logically attach/ detach devices
- Get and Set device attributes

Information Maintenance

It handles information and its transfer between the OS and the user program.

Functions:

- Get or set time and date
- Get process and device attributes

Communication:

These types of system calls are specially used for interprocess communications.

Functions:

- Create, delete communications connections
- Send, receive message
- Help OS to transfer status information
- Attach or detach remote devices

Example of System Calls in Windows and Unix

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

Rules for Passing Parameters in System Call

1. The floating-point parameters can’t be passed as a parameter in the system call.
2. Only a limited number of arguments can be passed in the system call.
3. If there are more arguments, then they should be stored in the block of memory and the address of that memory block is stored in the register.
4. Parameters can be pushed and popped from the stack only by the operating system.

Important System Calls Used in OS

wait()

In some systems, a process needs to wait for another process to complete its execution. This type of situation occurs when a parent process creates a child process, and the execution of the parent process remains suspended until its child process executes.

The suspension of the parent process automatically occurs with a wait() system call. When the child process ends execution, the control moves back to the parent process.

fork()

Processes use this system call to create processes that are a copy of themselves. With the help of this system Call parent process creates a child process, and the execution of the parent process will be suspended till the child process executes.

exec()

This system call runs when an executable file in the context of an already running process that replaces the older executable file. However, the original process identifier remains as a new process is not built, but stack, data, head, data, etc. are replaced by the new process.

kill():

The kill() system call is used by OS to send a termination signal to a process that urges the process to exit. However, a kill system call does not necessarily mean killing the process and can have various meanings.

exit():

The exit() system call is used to terminate program execution. Specially in the multi- threaded environment, this call defines that the thread execution is complete. The OS reclaims resources that were used by the process after the use of exit() system call.

Short note on system call

- System call is the interface through which the process communicates with the system call.
- Computer system operates in two modes: **User Mode** and **Kernel Mode**
- Process executes in user mode, and when a system call is made, the mode is switched to kernel mode. Once the system call execution is completed, the control is passed back to the process in user mode.
- System calls in OS are made by sending a trap signal to the kernel, which reads the system call code from the register and executes the system call.
- Major type of system calls are **Process Control**, **File Management**, **Device Management**, **Information maintenance** and **Communication**.
- Rules for parameter passing while making a system call is it should not be a floating number, a limited number of arguments should be passed and if arguments are more, they should be stored in memory block and address of that memory block should be passed, and the push, pop operations from the stack will be made only by the operating system.
- **wait()**, **fork()**, **exec()**, **kill()** and **exit()** are few important system calls of our computer system.

UNIT I

Process

Process

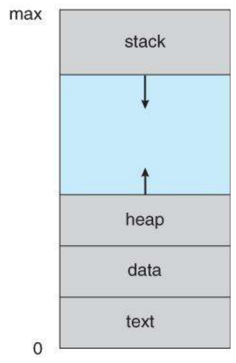
- Process Concepts And Scheduling
- Operations On Processes
- Cooperating Processes
- Threads

PROCESS CONCEPTS AND CPU SCHEDULING

Process: *A process is a program at the time of execution.*

A process is a program in execution which then forms the basis of all computation. The process is not as same as program code but a lot more than it. A process is an 'active' entity as opposed to the program which is considered to be a 'passive' entity. Attributes held by the process include hardware state, memory, CPU, etc.

Process memory is divided into four sections for efficient working:

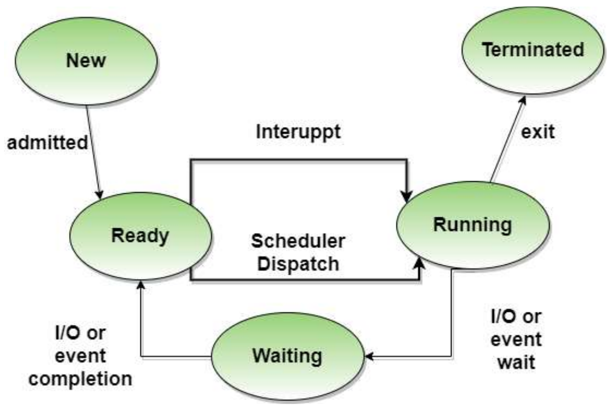


- The **Text section** is made up of the compiled program code, read in from non-volatile storage when the program is launched.
- The **Data section** is made up of the global and static variables, allocated and initialized prior to executing the main.
- The **Heap** is used for the dynamic memory allocation and is managed via calls to new, delete, malloc, free, etc.
- The **Stack** is used for local variables. Space on the stack is reserved for local variables **when they are declared.**

PROCESS STATES

Processes in the operating system can be in any of the following states:

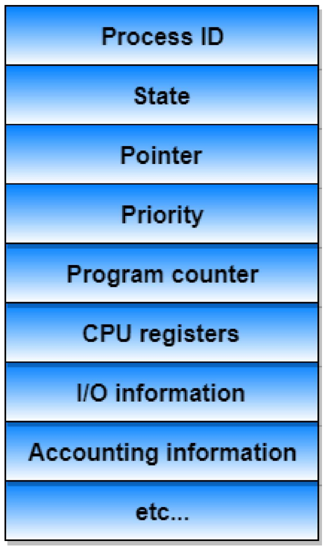
- NEW- The process is being created.
- READY- The process is waiting to be assigned to a processor.
- RUNNING- Instructions are being executed.
- WAITING- The process is waiting for some event to occur (such as an I/O completion or reception of a signal).
- TERMINATED- The process has finished execution.



Process Control Block

There is a Process Control Block for each process, enclosing all the information about the process. It is also known as the task control block. It is a data structure, which contains the following:

- **Process State**: It can be running, waiting, etc.
- **Process ID** and the **parent process ID**.
- CPU registers and Program Counter. **Program Counter** holds the address of the next instruction to be executed for that process.
- **CPU Scheduling** information: Such as priority information and pointers to scheduling queues.
- **Memory Management information**: For example, page tables or segment tables.
- **Accounting information**: The User and kernel CPU time consumed, account numbers, limits, etc.
- **I/O Status information**: Devices allocated, open file tables, etc.



PROCESS VS PROGRAM

Process	**Program**
The process is basically an instance of the computer program that is being executed.	A Program is basically a collection of instructions that mainly performs a specific task when executed by the computer.

Process	Program
A process has a shorter lifetime.	A Program has a longer lifetime.
A Process requires resources such as memory, CPU, Input-Output devices.	A Program is stored by hard-disk and does not require any resources.
A process has a dynamic instance of code and data	A Program has static code and static data.
Basically, a process is the running instance of the code.	On the other hand, the program is the executable code.

WHAT IS PROCESS SCHEDULING?

The act of determining which process is in the **ready** state, and should be moved to the **running** state is known as **Process Scheduling**.

The prime aim of the process scheduling system is to keep the CPU busy all the time and to deliver minimum response time for all programs. For achieving this, the scheduler must apply appropriate rules for swapping processes IN and OUT of CPU.

Scheduling fell into one of the two general categories:

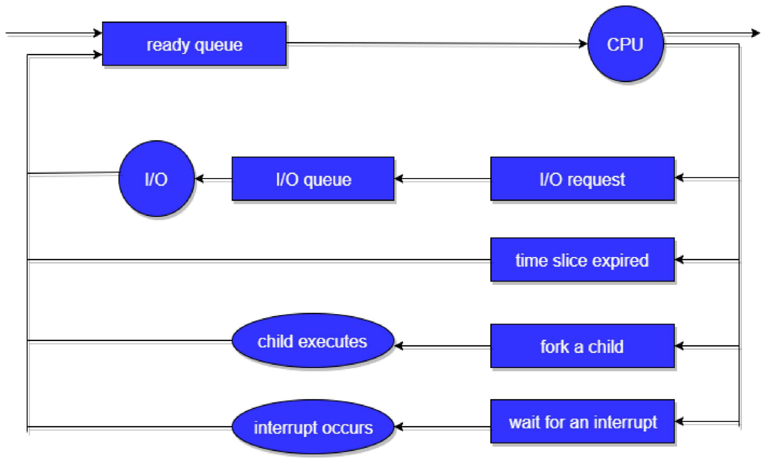
- **Non Pre-emptive Scheduling:** When the currently executing process gives up the CPU voluntarily.
- **Pre-emptive Scheduling:** When the operating system decides to favour another process, pre-empting the currently executing process.

SCHEDULING QUEUES:

- **Job Queue**: All processes, upon entering into the system, are stored in the **Job Queue**.
- **Ready Queue**: Processes in the Ready state are placed in the **Ready Queue**.
- **Device Queues**: Processes waiting for a device to become available are placed in **Device Queues**. There are unique device queues available for each I/O device.

A new process is initially put in the **Ready queue**. It waits in the ready queue until it is selected for execution (or dispatched). Once the process is assigned to the CPU and is executing, one of the following several events can occur:

- The process could issue an I/O request, and then be placed in the **I/O queue**.
- The process could create a new sub process and wait for its termination.
- The process could be removed forcibly from the CPU, as a result of an interrupt, and be put back in the ready queue.



In the first two cases, the process eventually switches from the waiting state to the ready state, and is then put back in the ready queue. A process continues this cycle until it terminates, at which time it is removed from all queues and has its PCB and resources deallocated.

TYPES OF SCHEDULERS

There are three types of schedulers available:

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

Long Term Scheduler

Long term scheduler runs less frequently. Long Term Schedulers decide which program must get into the job queue. From the job queue, the Job Processor, selects processes and loads them into the memory for execution. Primary aim of the Job Scheduler is to maintain a good degree of Multiprogramming. An optimal degree of Multiprogramming means the average rate of process creation is equal to the average departure rate of processes from the execution memory.

Short Term Scheduler

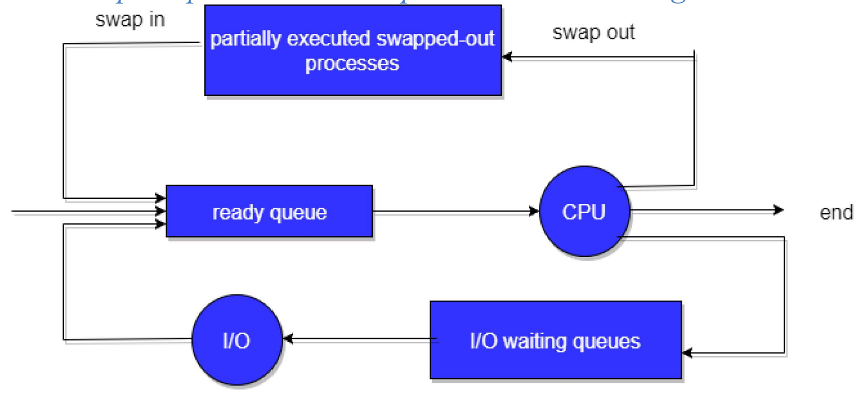
This is also known as CPU Scheduler and runs very frequently. The primary aim of this scheduler is to enhance CPU performance and increase process execution rate.

Medium Term Scheduler

This scheduler removes the processes from memory (and from active contention for the CPU), and thus reduces the degree of multiprogramming. At some later time, the process can be

reintroduced into memory and its execution van be continued where it left off. This scheme is called swapping. The process is swapped out, and is later swapped in, by the medium term scheduler.
Swapping may be necessary to improve the process mix, or because a change in memory requirements has overcommitted available memory, requiring memory to be freed up.

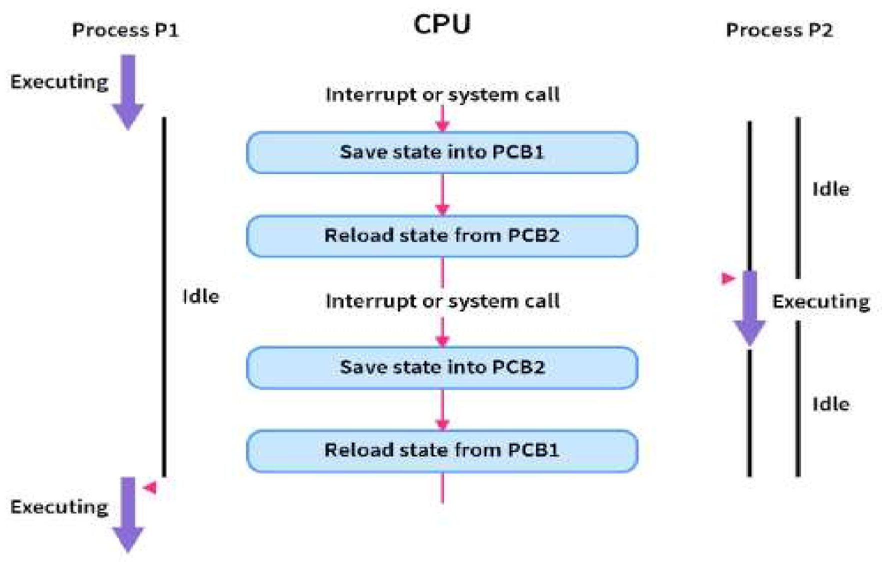
This complete process is descripted in the below diagram:



CONTEXT SWITCHING

Context Switching stores the state of the current process, and the state includes the information about the data that was stored in registers, the value of the program counter, and the stack pointer. Context Switching is necessary because if we directly pass the control of CPU to the new process without saving the state of the old process and later if we want to resume the old process from where it was stopped, we won't be able to do that as we don't know what the last instruction the old process executed was. Context Switching overcomes this problem by storing the state of the process.

Steps Involved in Context Switching



PCB stands for Process Control Block. PCB is a special data structure that is used by the operating system to store all the process-relevant data in it. PCB also stores the state of the process, which depicts whether the process is ready, running, or waiting.

As we can see in the image, there are two processes, namely P1 and P2. P1 is being executed by the CPU currently. If process P1 requires performing some I/O(Input-Output) task or if any interrupt takes place, the state of the process needs to change, but before changing the state, the Context Switching takes place.

Following are the steps which takes place while switching process P1 and process P2:

1. The data present in the register and program counter will be stored in PCB of process P1, let's say PCB1, and change the state in PCB1 .
2. Process P1 will be moved to the appropriate queue, which can be either ready queue, I/O queue, or waiting queue.
3. The next process will be picked from the ready queue, let's say P2.
4. The state of the process P2 will be changed to running state, and if P2 was the process that was previously executed by CPU, it would resume the execution from where it was stopped.
5. If we need to execute the process P1, we need to carry out the same tasks mentioned in step 1 to step 4.

Context Switching Triggers

Following are the three main triggers for Context Switching:

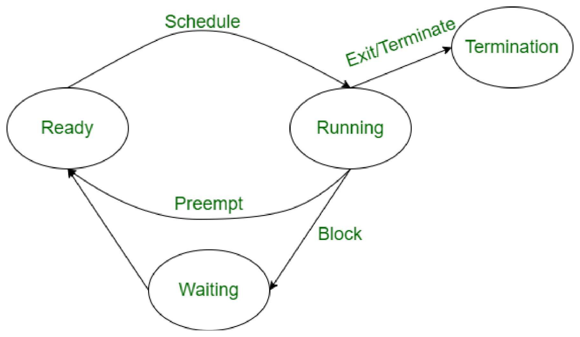
1. **Multitasking:** In the multitasking environment, when one process is utilizing CPU, and there is a need for CPU by another process, Context Switching triggers. Context Switching saves the state of the old process and passes the control of the CPU to the new process.

2. **Interrupt Handling:** When an interrupt takes place, the CPU needs to handle the interrupt. So before handling the interrupt, the Context Switching gets triggered, which saves the state of the process before handling the interrupt.
3. **User and Kernel Mode Switching:** The user mode is the normal mode in which the user application can execute with limited access, whereas kernel mode is the mode in which the process can carry out the system-level operations that are not available in the user mode. So, whenever the switching takes from user mode to kernel mode, mode switching triggers the Context Switching, which stores the state on an ongoing process.

OPERATION ON A PROCESS:

The execution of a process is a complex activity. It involves various operations.

Following are the operations that are performed while execution of a process:



1. Creation: This is the initial step of process execution activity. Process creation means the construction of a new process for the execution. This might be performed by system, user or old process itself. There are several events that leads to the process creation. Some of the such events are following:

- When we start the computer, system creates several background processes.
- A user may request to create a new process.
- A process can create a new process itself while executing.
- Batch system takes initiation of a batch job.

2. Scheduling/Dispatching: The event or activity in which the state of the process is changed from ready to running. It means the operating system puts the process from ready state into the running state. Dispatching is done by operating system when the resources are free or the process has higher priority than the ongoing process. There are various other cases in which the process in running state is preempted and process in ready state is dispatched by the operating system.

3. Blocking: When a process invokes an input-output system call that blocks the process and operating system put in block mode. Block mode is basically a mode where process waits for input-output. Hence on the demand of process itself, operating system blocks the process and dispatches another process to the processor. Hence, in process blocking operation, the operating system puts the process in 'waiting' state.

4. Preemption: When a timeout occurs that means the process hadn't been terminated in the allotted time interval and next process is ready to execute, then the operating system preempts the process. This operation is only valid where CPU scheduling supports preemption. Basically this happens in priority scheduling where on the incoming of high priority process the ongoing process is preempted. Hence, in process preemption operation, the operating system puts the process in 'ready' state.

5. Termination: Process termination is the activity of ending the process. In other words, process termination is the relaxation of computer resources taken by the process for the execution. Like creation, in termination also there may be several events that may lead to the process termination. Some of them are:

- Process completes its execution fully and it indicates to the OS that it has finished.
- Operating system itself terminates the process due to service errors.
- There may be problem in hardware that terminates the process.
- One process can be terminated by another process.

COOPERATING PROCESS

Before termination, there may be multiple processes being executed in a system. There are two modes in which the processes can be executed. *These two modes are:*

- Serial mode
- Parallel mode

In **serial mode**, the process will be executed one after the other means the next process cannot be executed until the previous process gets terminated.

On the contrary in **parallel mode**, there may be several processes being executed at the same time quantum. In this way, there will be two types of processes which can be either cooperating processes or independent processes.

Cooperating Process in the operating system is a process that gets affected by other processes under execution or can affect any other process under execution. It shares data with other processes in the system by directly sharing a logical space which is both code and data or by sharing data through files or messages.

Whereas, an independent process in an operating system is one that does not affect or impact any other process of the system. It does not share any data with other processes.

Methods of Cooperating Process in OS
Cooperating processes in OS requires a communication method that will allow the processes to exchange data and information.

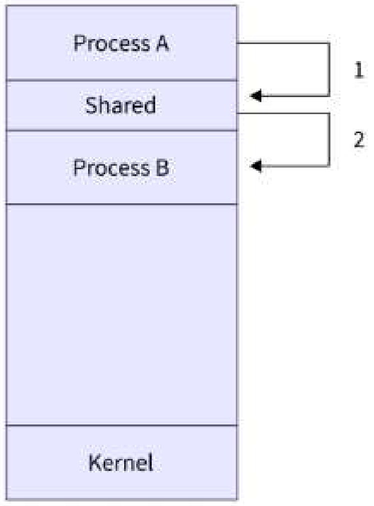
There are two methods by which cooperating process in OS can communicate:

- Cooperation by Sharing
- Cooperation by Message Passing

Cooperation by Sharing
The cooperation processes in OS can communicate with each other using the shared resource which includes data, memory, variables, files, etc.

Processes can then exchange the information by reading or writing data to the shared region. We can use a critical section that provides data integrity and avoids data inconsistency.

Let's see a diagram to understand more clearly the communication by shared region:



In the above diagram, We have two processes A and B which are communicating with each other through a shared region of memory. Process A will write the information in the shared region and then Process B will read the information from the shared memory and that's how the process of communication takes place between the cooperating processes by sharing.

Cooperation by Message Passing

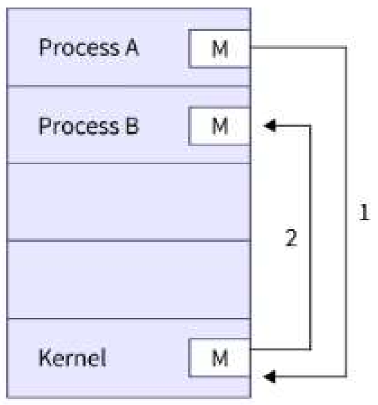
The cooperating processes in OS can communicate with each other with the help of message passing. The production process will send the message and the consumer process will receive the same message.

There is no concept of shared memory instead the producer process will first send the message to the kernel and then the kernel sends that message to the consumer process.

*A **kernel** is known as the heart and core of an operating system. The kernel interacts with the hardware to execute the processes given by the user space. It works as a bridge between the user space and hardware. Functions of the kernel include process management, file management, memory management, and I/O management.*

If a consumer process waits for a message from another process to execute a particular task then this may cause a problem of deadlock and if the consumer process does not receive the message then this may cause a problem of process starvation.

Let's see a diagram to understand more clearly the cooperation by communication:



In the above diagram, the process A and B are communicating with each other. Process A first sends the message to the kernel and then the kernel will interpret that this message is meant for Process B.

The kernel then sends the message to the process P2 and that's how the process of communication takes place between the cooperation processes by communication.

Need of Cooperating Processes in OS
One process will write to the file and the other process reads the file. Therefore, every process in the system could be affected by the other process.

The need for cooperating processes in OS can be divided into four types:

- Information Sharing
- Computation Speed
- Convenience
- Modularity

Information Sharing
As we know the cooperating process in OS shares data and information between other processes. There may be a possibility that different processes are accessing the same file. Processes can access the files concurrently which makes the execution of the process more efficient and faster.

Computation Speed
When a task is divided into several subtasks and starts executing them parallelly, this improves the computation speed of the execution and makes it faster. Computation speed can be achieved if a system has multiple CPUs and input/output devices.

When the tasks are assigned into several subtasks they become several different processes that need to communicate with each other. That's why we need cooperating processes in the operating system.

Convenience
A user may be performing several tasks at the same time which leads to the running of different processes concurrently. These processes need to cooperate so that every process can run smoothly without interrupting each other.

Modularity
We want to divide a system of complex tasks into several different modules and later they will be established together to achieve a goal. This will help in completing tasks with more efficiency as well as speed.

Advantages of Cooperating Process in Operating System

- With help of data and information sharing, the processes can be executed with much faster speed and efficiency as processes can access the same files concurrently.
- Modularity gives the advantage of breaking a complex task into several modules which are later put together to achieve the goal of faster execution of processes.
- Cooperating processes provide convenience as different processes running at the same time can cooperate without any interruption among them.
- The computation speed of the processes increases by dividing processes into different subprocesses and executing them parallelly at the same time.

Disadvantages of Cooperating Process in Operating System

- During the communication method of the cooperation processes, there may be a problem of deadlock if a consumer process waits for the message from the production process and the message does not receive at the consuming end.
- There may be a condition of process starvation where the next process will have to wait until the message is received by the previous consuming process.
- The cooperating processes in the operating system can damage the data which may occur due to modularity.
- During information sharing, it may also share any sensitive data of the user with the other process that the user might not want to share.

Example: producer-consumer problem which is also known as a bounded buffer problem to understand **cooperating processes.**

THREADS

Thread is a sequential flow of tasks within a process. Threads in OS can be of the same or different types. Threads are used to increase the performance of the applications.

Each thread has its own program counter, stack, and set of registers. But the threads of a single process might share the same code and data/file. Threads are also termed as lightweight processes as they share common resources.

Example: While playing a movie on a device the audio and video are controlled by different threads in the background.

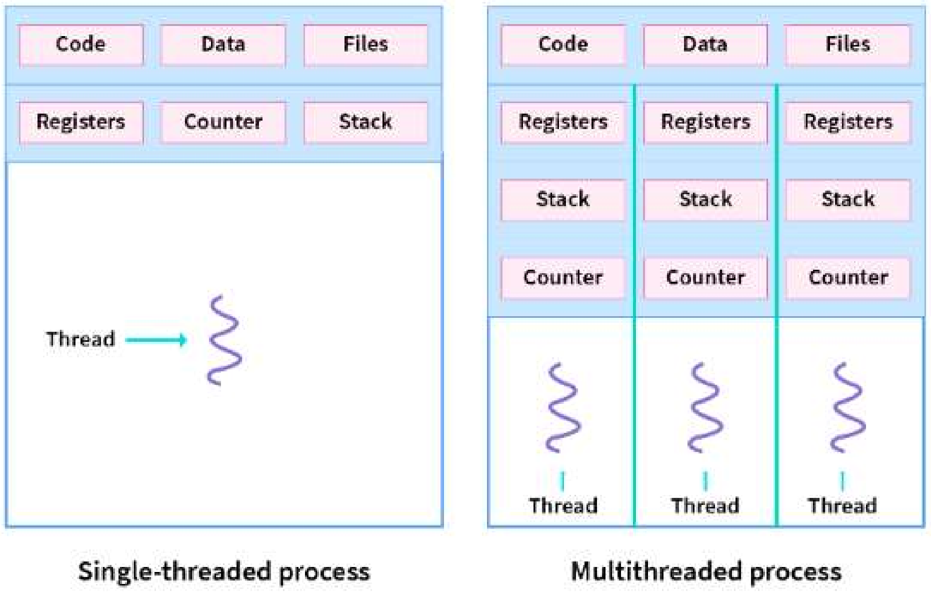
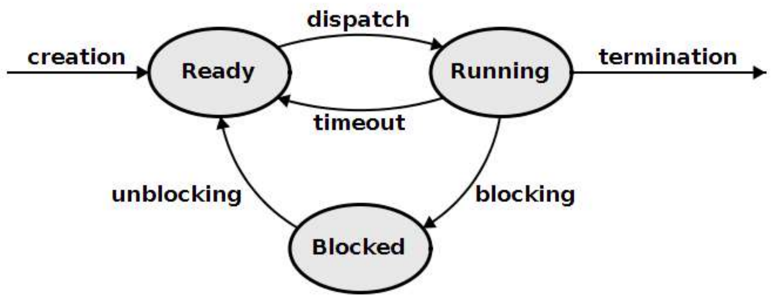


Figure: Single-threaded process and a multithreaded process and the resources that are shared among threads

Life cycle of Thread(Thread States)

1. Born: A thread is just created.
2. Ready: The thread is waiting for CPU.
3. Running : System assigns the processor to the thread
4. Sleep/blocked: A sleeping thread becomes ready after the designated sleep time expires.
5. Dead/terminate: The Execution of the thread finished.

Example: Word processor. Typing, Formatting, Spell check, saving are thread



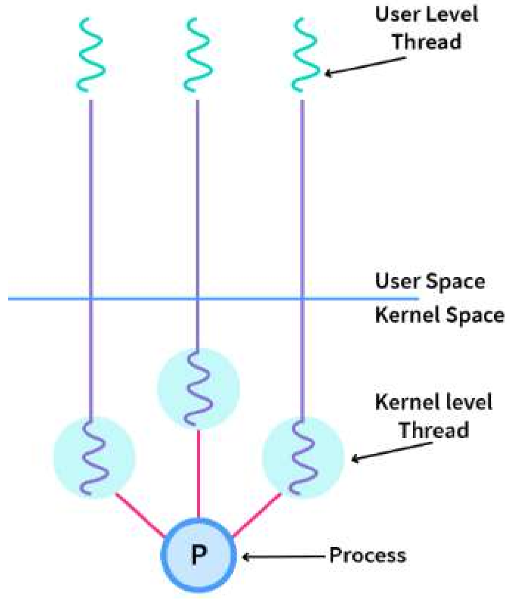
Types of Thread

1. User Level Thread:
User-level threads are implemented and managed by the user and the kernel is not aware of it.

- User-level threads are implemented using user-level libraries and the OS does not recognize these threads.
- User-level thread is faster to create and manage compared to kernel-level thread.
- Context switching in user-level threads is faster.
- If one user-level thread performs a blocking operation then the entire process gets blocked. **Example:** POSIX threads, Java threads, etc.
-

2. Kernel level Thread:
Kernel level threads are implemented and managed by the OS.

- Kernel level threads are implemented using system calls and Kernel level threads are recognized by the OS.
- Kernel-level threads are slower to create and manage compared to user-level threads.
- Context switching in a kernel-level thread is slower.
- Even if one kernel-level thread performs a blocking operation, it does not affect other threads. **Example:** Window Solaris.



The above diagram shows the functioning of user-level threads in user space and kernel-level threads in kernel space.

A thread has the following three components:

- Program Counter
- Register Set
- Stack space

Why do we need Threads?
Threads in the operating system provide multiple benefits and improve the overall performance of the system. Some of the reasons threads are needed in the operating system are:

Since threads use the same data and code, the operational cost between threads is low.
Creating and terminating a thread is faster compared to creating or terminating a process.
Context switching is faster in threads compared to processes.

Advantages of Threading

- Threads improve the overall performance of a program.
- Threads increases the responsiveness of the program
- Context Switching time in threads is faster.
- Threads share the same memory and resources within a process.
- Communication is faster in threads.
- Threads provide concurrency within a process.
- Enhanced throughput of the system.
- Since different threads can run parallelly, threading enables the utilization of the multiprocessor architecture to a greater extent and increases efficiency.

Differences between Process and Thread

PROCESS	THREAD
A Process simply means any program in execution.	Thread simply means a segment of a process.
The process consumes more resources	Thread consumes fewer resources.
The process requires more time for creation.	Thread requires comparatively less time for creation than process.
The process is a heavyweight process	Thread is known as a lightweight process
The process takes more time to terminate	The thread takes less time to terminate.

PROCESS	THREAD
Processes have independent data and code segments	A thread mainly shares the data segment, code segment, files, etc. with its peer threads.
The process takes more time for context switching.	The thread takes less time for context switching.
Communication between processes needs more time as compared to thread.	Communication between threads needs less time as compared to processes.
For some reason, if a process gets blocked then the remaining processes can continue their execution	In case if a user-level thread gets blocked, all of its peer threads also get blocked.

MULTITHREADING
In Multithreading, the idea is to divide a single process into multiple threads instead of creating a whole new process. Multithreading is done to achieve parallelism and to improve the performance of the applications as it is faster in many ways which were discussed above.

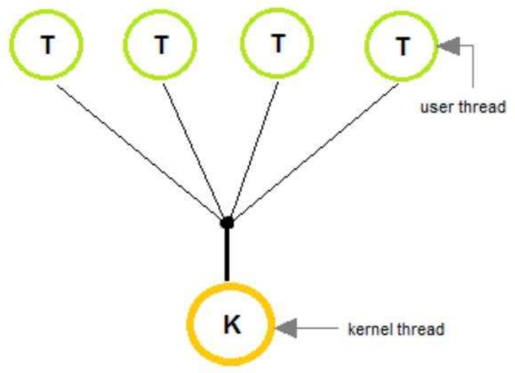
The other advantages of multithreading are mentioned below.

Resource Sharing: Threads of a single process share the same resources such as code, data/file.
Responsiveness: Program responsiveness enables a program to run even if part of the program is blocked or executing a lengthy operation. Thus, increasing the responsiveness to the user.
Economy: It is more economical to use threads as they share the resources of a single process. On the other hand, creating processes is expensive.

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

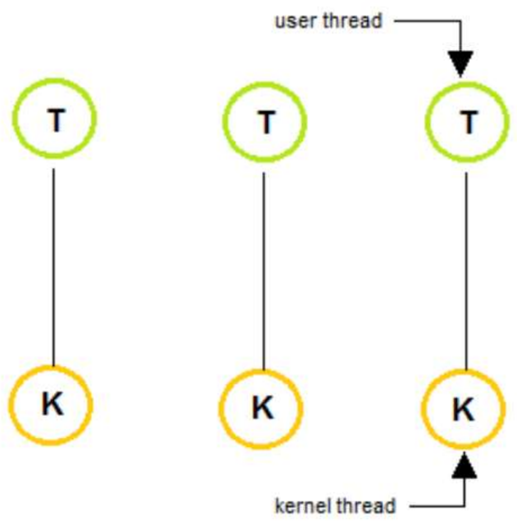
- Many to One Model
- One to One Model
- Many to Many Model

Many to One Model
- In the **many to one** model, many user-level threads are all mapped onto a single kernel thread.
- Thread management is handled by the thread library in user space, which is efficient in nature.
- In this case, if user-level thread libraries are implemented in the operating system in some way that the system does not support them, then the Kernel threads use this many-to-one relationship model.



One to One Model

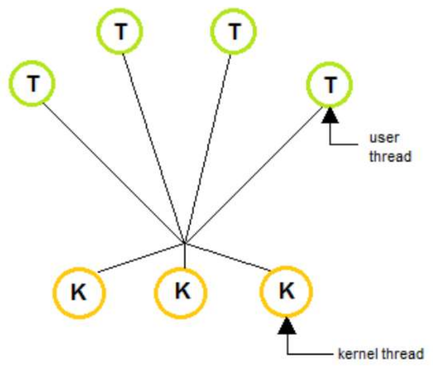
- The **one to one** model creates a separate kernel thread to handle each and every user thread.
- Most implementations of this model place a limit on how many threads can be created.
- Linux and Windows from 95 to XP implement the one-to-one model for threads.
- This model provides more concurrency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



Many to Many Model

- The many to many model multiplexes any number of user threads onto an equal or smaller number of kernel threads, combining the best features of the one-to-one and many-to-one models.

- Users can create any number of threads.
- Blocking the kernel system calls does not block the entire process.
- Processes can be split across multiple processors.



Multithreading Issues
Below we have mentioned a few issues related to multithreading. Well, it's an old saying, *All good things, come at a price.*

Thread Cancellation
Thread cancellation means terminating a thread before it has finished working. There can be two approaches for this, one is **Asynchronous cancellation**, which terminates the target thread immediately. The other is **Deferred cancellation** allows the target thread to periodically check if it should be canceled.

Signal Handling
Signals are used in UNIX systems to notify a process that a particular event has occurred. Now in when a multithreaded process receives a signal, to which thread it must be delivered? It can be delivered to all or a single thread.

fork() System Call
fork() is a system call executed in the kernel through which a process creates a copy of itself. Now the problem in the Multithreaded process is, if one thread forks, will the entire process be copied or not?

Security Issues
Yes, there can be security issues because of the extensive sharing of resources between multiple threads.

There are many other issues that you might face in a multithreaded process, but there are appropriate solutions available for them. Pointing out some issues here was just to study both sides of the coin.

MAHAVEER INSTITUTE OF SCIENCE AND TECHNOLOGY

(AN UGC AUTONOMOUS INSTITUTION)

Approved by AICTE, Affiliated to JNTUH, Accredited by NAAC with 'A' Grade
Recognized Under Section 2(f) of UGC Act 1956, ISO 9001:2015 Certified
Vyasapuri, Bandlaguda, Post: Keshavgiri, Hyderabad- 500 005, Telangana, India.
https://www.mist.ac.in E-mail:principal.mahaveer@gmail.com, Mobile: 8978380692

ESTD : 2001

Department of Computer Science and Engineering (AIML)

(R22)

OPERATING SYSTEM

Lecture Notes

B. Tech II YEAR – I SEM

Prepared by

SANGYAM SOUNDARYA
(Assistant Professor)
Dept.CSE(AIML)

OPERATING SYSTEMS

B.Tech. II Year I Sem.

L	T	P	C
3	**0**	**0**	**3**

Prerequisites:

1. A course on “Computer Programming and Data Structures”.
2. A course on “Computer Organization and Architecture”.

Course Objectives:

- Introduce operating system concepts (i.e., processes, threads, scheduling, synchronization, deadlocks, memory management, file and I/O subsystems and protection)
- Introduce the issues to be considered in the design and development of operating system
- Introduce basic Unix commands, system call interface for process management, interprocess communication and I/O in Unix

Course Outcomes:

- Will be able to control access to a computer and the files that may be shared
- Demonstrate the knowledge of the components of computers and their respective roles in computing.
- Ability to recognize and resolve user problems with standard operating environments.
- Gain practical knowledge of how programming languages, operating systems, and architectures interact and how to use each effectively.

UNIT - I

Operating System - Introduction, Structures - Simple Batch, Multiprogrammed, Time-shared, Personal Computer, Parallel, Distributed Systems, Real-Time Systems, System components, Operating System services, System Calls

Process - Process concepts and scheduling, Operations on processes, Cooperating Processes, Threads

UNIT - II

CPU Scheduling - Scheduling Criteria, Scheduling Algorithms, Multiple -Processor Scheduling. System call interface for process management-fork, exit, wait, waitpid, exec

Deadlocks - System Model, Deadlocks Characterization, Methods for Handling Deadlocks, Deadlock Prevention, Deadlock Avoidance, Deadlock Detection, and Recovery from Deadlock

UNIT - III

Process Management and Synchronization - The Critical Section Problem, Synchronization Hardware, Semaphores, and Classical Problems of Synchronization, Critical Regions, Monitors

Interprocess Communication Mechanisms: IPC between processes on a single computer system, IPC between processes on different systems, using pipes, FIFOs, message queues, shared memory.

UNIT - IV

Memory Management and Virtual Memory - Logical versus Physical Address Space, Swapping, Contiguous Allocation, Paging, Segmentation, Segmentation with Paging, Demand Paging, Page Replacement, Page Replacement Algorithms.

UNIT - V

File System Interface and Operations -Access methods, Directory Structure, Protection, File System Structure, Allocation methods, Free-space Management. Usage of open, create, read, write, close, lseek, stat, ioctl system calls.

TEXT BOOKS:

1. Operating System Principles- Abraham Silberchatz, Peter B. Galvin, Greg Gagne 7th Edition, John Wiley.
2. Advanced programming in the UNIX environment, W.R. Stevens, Pearson education.

REFERENCE BOOKS:

1. Operating Systems- Internals and Design Principles, William Stallings, Fifth Edition–2005, Pearson Education/PHI
2. Operating System A Design Approach- Crowley, TMH.
3. Modern Operating Systems, Andrew S. Tanenbaum 2nd edition, Pearson/PHI
4. UNIX programming environment, Kernighan and Pike, PHI/ Pearson Education
5. UNIX Internals -The New Frontiers, U. Vahalia, Pearson Education.

UNIT II
CPU SCHEDULING

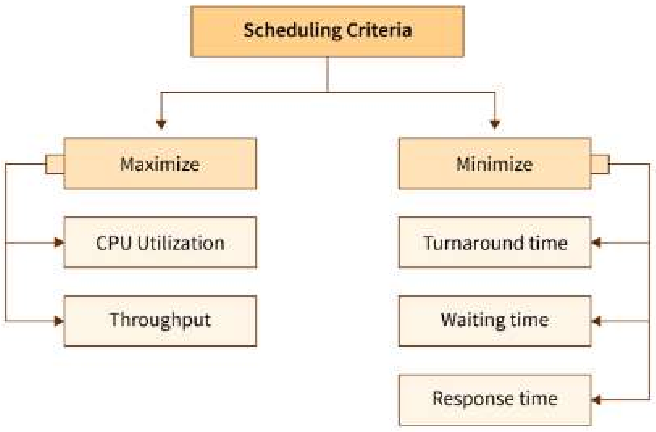
SCHEDULING CRITERIA

There are different CPU scheduling algorithms with different properties. The choice of algorithm is dependent on various different factors such as waiting for time, efficiency, CPU utilization, etc. In this blog, we will learn about Scheduling criteria in OS and its different criteria in detail.

Scheduling is a process of allowing one process to use the CPU resources, keeping on hold the execution of another process due to the unavailability of resources CPU.

Example:

- First-Come First-Serve Scheduling, FCFS.
- Shortest-Job-First Scheduling, SJF.
- Priority Scheduling.
- Round Robin Scheduling.
- Multilevel Queue Scheduling.
- Multilevel Feedback-Queue Scheduling.



Types of Scheduling Criteria in an Operating System

There are different CPU scheduling algorithms with different properties. The choice of algorithm is dependent on various different factors. There are many criteria suggested for comparing CPU schedule algorithms, some of which are:

- CPU utilization
- Throughput
- Turnaround time
- Waiting time

- Response time

CPU utilization- The object of any CPU scheduling algorithm is to keep the CPU busy if possible and to maximize its usage. In theory, the range of CPU utilization is in the range of 0 to 100 but in real-time, it is actually 50 to 90% which relies on the system's load.

Throughput- It is a measure of the work that is done by the CPU which is directly proportional to the number of processes being executed and completed per unit of time. It keeps on varying which relies on the duration or length of processes.

Turnaround time- An important Scheduling criterion in OS for any process is how long it takes to execute a process. A turnaround time is the elapsed from the time of submission to that of completion. It is the summation of time spent waiting to get into the memory, waiting for a queue to be ready, for the I/O process, and for the execution of the CPU.

The formula for calculating: *TurnAroundTime=Compilationtime−Arrivaltime*.

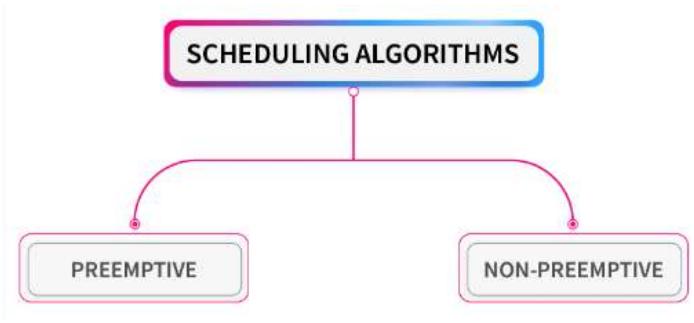
Waiting time- Once the execution starts, the scheduling process does not hinder the time that is required for the completion of the process. The only thing that is affected is the waiting time of the process, i.e the time that is spent by a process waiting in a queue. **The formula for calculating KaTeX parse error: Expected 'EOF', got '–' at position 31: …Turnaround Time –_Burst Time.**

Response time- Turnaround time is not considered as the best criterion for comparing scheduling algorithms in an interactive system. Some outputs of the process might produce early while computing other results simultaneously. Another criterion is the time that is taken from process submission to generate the first response. This is called response time and **the formula for calculating it is, KaTeX parse error: Expected 'EOF', got '–' at position 79: …for the first) –_Arrival Time.**

SCHEDULING ALGORITHMS

A CPU scheduling algorithm is used to determine which process will use CPU for execution and which processes to hold or remove from execution. The main goal or objective of CPU scheduling algorithms in OS is to make sure that the CPU is never in an idle state, meaning that the OS has at least one of the processes ready for execution among the available processes in the ready queue.

There are two types of scheduling algorithms in OS:



Preemptive Scheduling Algorithms
In these algorithms, processes are assigned with priority. Whenever a high-priority process comes in, the lower-priority process which has occupied the CPU is preempted. That is, it releases the CPU, and the high-priority process takes the CPU for its execution.

Non-Preemptive Scheduling Algorithms
In these algorithms, we cannot preempt the process. That is, once a process is running on CPU, it will release it either by context switching or terminating. Often, these are the types of algorithms that can be used because of the limitation of the hardware.

Differences between Preemptive vs. Non-Preemptive Scheduling

Basis For Comparison	Preemptive Scheduling	Non Preemptive Scheduling
Basic	The resources are allocated to a process for a limited time.	Once resources are allocated to a process, the process holds it till it completes its burst time or switches to waiting state.
Interrupt	Process can be interrupted in between.	Process cannot be interrupted till it terminates or switches to waiting state.
Starvation	If a high priority process frequently arrives in the ready queue, low priority process may starve.	If a process with long burst time is running CPU, then another process with less CPU burst time may starve.
Overhead	Preemptive scheduling has overheads of scheduling the processes.	Non-preemptive scheduling does not have overheads.
Flexibility	Preemptive scheduling is flexible.	Non-preemptive scheduling is rigid.
Cost	Preemptive scheduling is cost associated.	Non-preemptive scheduling is not cost associative.

First Come First Serve (FCFS) Scheduling Algorithm

First Come First Serve is the easiest and simplest CPU scheduling algorithm to implement. In this type of scheduling algorithm, the CPU is first allocated to the process which requests the CPU first. That means the process with minimal arrival time will be executed first by the CPU. It is a non-preemptive scheduling algorithm as the priority of processes does not matter, and they are executed in the manner they arrive in front of the CPU. This scheduling algorithm is implemented with a FIFO(First In First Out) queue. As the process is ready to be executed, its Process Control Block (PCB) is linked with the tail of this FIFO queue. Now when the CPU becomes free, it is assigned to the process at the beginning of the queue.

Advantages

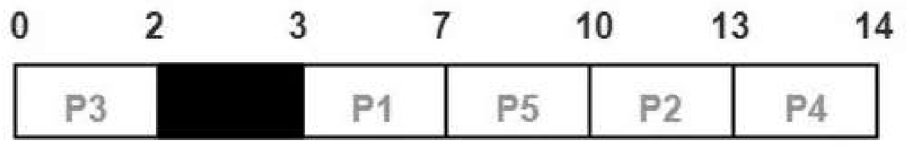
- Involves no complex logic and just picks processes from the ready queue one by one.
- Easy to implement and understand.
- Every process will eventually get a chance to run so no starvation occurs.

Disadvantages

- Waiting time for processes with less execution time is often very long.
- It favors CPU-bound processes then I/O processes.
- Leads to convoy effect.
- Causes lower device and CPU utilization.
- Poor performance as the average wait time is high.

Solve: Consider the set of 5 processes whose arrival time and burst time are given below.If the CPU scheduling policy is FCFS, calculate the average waiting time and average turnaround time.

Process Id	Arrival time	Burst time
P1	3	4
P2	5	3
P3	0	2
P4	5	1
P5	4	3



Gantt Chart

Now, we know-

- ***Turn Around time = Exit time – Arrival time***
- ***Waiting time = Turn Around time – Burst time***

Process Id	Exit time	Turn Around time	Waiting time
P1	7	7 – 3 = 4	4 – 4 = 0
P2	13	13 – 5 = 8	8 – 3 = 5
P3	2	2 – 0 = 2	2 – 2 = 0
P4	14	14 – 5 = 9	9 – 1 = 8
P5	10	10 – 4 = 6	6 – 3 = 3

Now,

- Average Turn Around time = (4 + 8 + 2 + 9 + 6) / 5 = 29 / 5 = 5.8 unit
- Average waiting time = (0 + 5 + 0 + 8 + 3) / 5 = 16 / 5 = 3.2 unit

Shortest Job First (SJF) Scheduling Algorithm

Shortest Job First is a non-preemptive scheduling algorithm in which the process with the shortest burst or completion time is executed first by the CPU. That means the lesser the execution time, the sooner the process will get the CPU. In this scheduling algorithm, the arrival time of the processes must be the same, and the processor must be aware of the burst time of all the processes in advance. If two processes have the same burst time, then First Come First Serve (FCFS) scheduling is used to break the tie.

The preemptive mode of SJF scheduling is known as the Shortest Remaining Time First scheduling algorithm.

Advantages

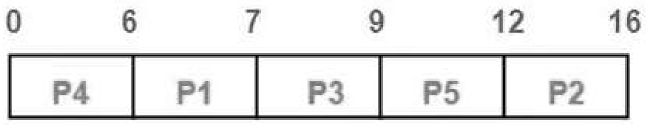
- Results in increased Throughput by executing shorter jobs first, which mostly have a shorter turnaround time.
- Gives the minimum average waiting time for a given set of processes.
- Best approach to minimize waiting time for other processes awaiting execution.
- Useful for batch-type processing where CPU time is known in advance and waiting for jobs to complete is not critical.

Disadvantages

- May lead to starvation as if shorter processes keep on coming, then longer processes will never get a chance to run.
- Time taken by a process must be known to the CPU beforehand, which is not always possible.

Solve: Consider the set of 5 processes whose arrival time and burst time are given below- If the CPU scheduling policy is SJF non-preemptive, calculate the average waiting time and average turnaround time.

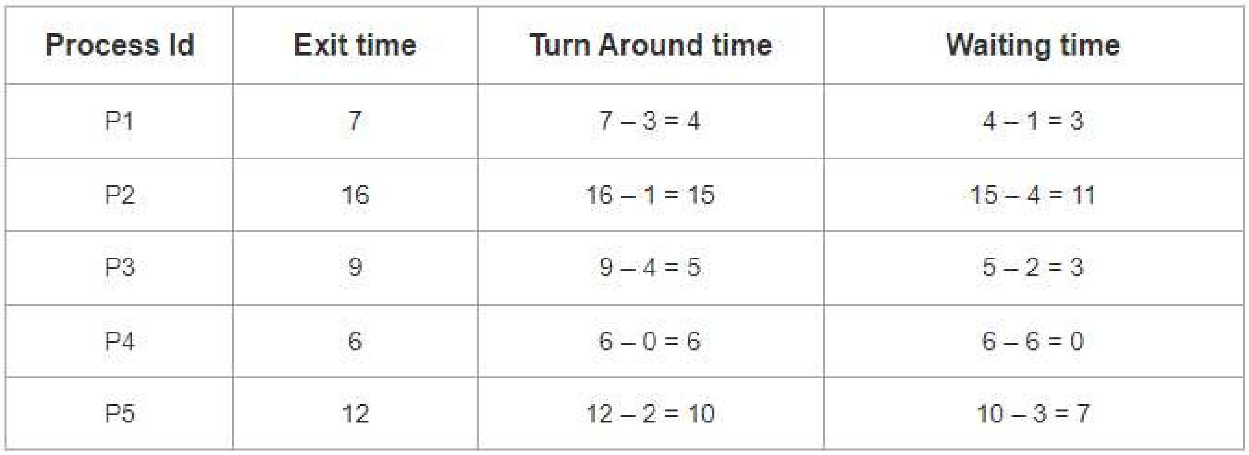
Process Id	Arrival time	Burst time
P1	3	1
P2	1	4
P3	4	2
P4	0	6
P5	2	3



Gantt Chart

Now, we know-

- **Turn Around time = Exit time – Arrival time**
- **Waiting time = Turn Around time – Burst time**



Process Id	Exit time	Turn Around time	Waiting time
P1	7	7 – 3 = 4	4 – 1 = 3
P2	16	16 – 1 = 15	15 – 4 = 11
P3	9	9 – 4 = 5	5 – 2 = 3
P4	6	6 – 0 = 6	6 – 6 = 0
P5	12	12 – 2 = 10	10 – 3 = 7

Now,

- Average Turn Around time = (4 + 15 + 5 + 6 + 10) / 5 = 40 / 5 = 8 unit
- Average waiting time = (3 + 11 + 3 + 0 + 7) / 5 = 24 / 5 = 4.8 unit

Round robin Scheduling Algorithm

The Round Robin algorithm is related to the First Come First Serve (FCFS) technique but implemented using a preemptive policy. In this scheduling algorithm, processes are executed cyclically, and each process is allocated a small amount of time called time slice or time quantum. The ready queue of the processes is implemented using the circular queue technique in which the CPU is allocated to each process for the given time quantum and then added back to the ready queue to wait for its next turn. If the process completes its execution within the given quantum of time, then it will be preempted, and other processes will execute for the given period of time. But if the process is not completely executed within the given time quantum, then it will again be added to the ready queue and will wait for its turn to complete its execution.

The round-robin scheduling is the oldest and simplest scheduling algorithm that derives its name from the round-robin principle. In this principle, each person will take an equal share of something in turn.

This algorithm is mostly used for multitasking in time-sharing systems and operating systems having multiple clients so that they can make efficient use of resources.

Advantages

- All processes are given the same priority; hence all processes get an equal share of the CPU.
- Since it is cyclic in nature, no process is left behind, and starvation doesn't exist.

Disadvantages

- The performance of Throughput depends on the length of the time quantum. Setting it too short increases the overhead and lowers the CPU efficiency, but if we set it too long, it gives a poor response to short processes and tends to exhibit the same behavior as FCFS.
- Average waiting time of the Round Robin algorithm is often long.
- Context switching is done a lot more times and adds to the overhead time.

Important note-01: With decreasing value of time quantum,

- Number of context switch increases
- Response time decreases
- Chances of starvation decreases

Thus, smaller value of time quantum is better in terms of response time.

Important note-02: With increasing value of time quantum,

- Number of context switch decreases
- Response time increases
- Chances of starvation increases

Thus, higher value of time quantum is better in terms of number of context switch.

Important note-03:

- With increasing value of time quantum, Round Robin Scheduling tends to become FCFS Scheduling.
- When time quantum tends to infinity, Round Robin Scheduling becomes FCFS Scheduling.

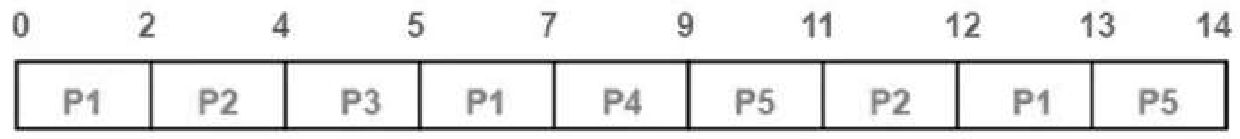
Important note-03:

- The performance of Round Robin scheduling heavily depends on the value of time quantum.
- The value of time quantum should be such that it is neither too big nor too small.

Solve: Consider the set of 5 processes whose arrival time and burst time are given below If the CPU scheduling policy is Round Robin with time quantum = 2 unit, calculate the average waiting time and average turnaround time.

Process Id	Arrival time	Burst time
P1	0	5
P2	1	3
P3	2	1
P4	3	2
P5	4	3

P5, P1, P2, P5, P4, P1, P3, P2, P1



Gantt Chart

- **Turn Around time = Exit time – Arrival time**
- **Waiting time = Turn Around time – Burst time**

Process Id	Exit time	Turn Around time	Waiting time
P1	13	13 – 0 = 13	13 – 5 = 8
P2	12	12 – 1 = 11	11 – 3 = 8
P3	5	5 – 2 = 3	3 – 1 = 2
P4	9	9 – 3 = 6	6 – 2 = 4
P5	14	14 – 4 = 10	10 – 3 = 7

- Average Turn Around time = (13 + 11 + 3 + 6 + 10) / 5 = 43 / 5 = 8.6 unit
- Average waiting time = (8 + 8 + 2 + 4 + 7) / 5 = 29 / 5 = 5.8 unit

Priority Scheduling Algorithm

The priority of a process is generally the inverse of the CPU burst time, i.e. the larger the burst time the lower is the priority of that process.
In case of priority scheduling the priority is not always set as the inverse of the CPU burst time, rather it can be internally or externally set, but yes the scheduling is done on the basis of priority of the process where the process which is most urgent is processed first, followed by the ones with lesser priority in order.

Processes with same priority are executed in FCFS manner.

The priority of process, when internally defined, can be decided based on **memory requirements**, **time limits**, number **of open files**, **ratio of I/O burst to CPU burst** etc.

Whereas, external priorities are set based on criteria outside the operating system, like the importance of the process, funds paid for the computer resource use, market factor etc.

Priority scheduling can be of two types:

1. **Preemptive Priority Scheduling**: If the new process arrived at the ready queue has a higher priority than the currently running process, the CPU is preempted, which means the processing of the current process is stoped and the incoming new process with higher priority gets the CPU for its execution.

2. **Non-Preemptive Priority Scheduling**: In case of non-preemptive priority scheduling algorithm if a new process arrives with a higher priority than the current running process, the incoming process is put at the head of the ready queue, which means after the execution of the current process it will be processed.

Advantages of Priority Scheduling Algorithm

- High priority processes do not have to wait for their chance to be executed due to the current running process.
- We are able to define the relative importance / priority of processes.
- The applications in which the requirements of time and resources fluctuate are useful.

Disadvantages of Priority Scheduling Algorithm

- Since we only execute high priority processes, this can lead to starvation of the processes that have a low priority. Starvation is the phenomenon in which a process gets infinitely postponed because the resources that are required by the process are never allocated to it, since other processes are executed before it. You can research more about starvation on Google.
- If the system eventually crashes, all of the processes that have low priority will get lost since they are stored in the RAM.

Static and Dynamic Priority

When we were learning about priorities being given to proceses, did you ponder about how the priorities were assigned to the process in the first place? Or rather, when was the priority assigned to the process?

In priority based scheduling, we saw that it could be implemented in two ways - preemptive and non-preemptive with the most common implementation being preemptive priority based scheduling. The same way, it can be categorized again on the basis of the method by which the priorities to processes are assigned.

The two classifications are:

- Static priority
- Dynamic priority

The Static Priority algorithm, assigns priorities to processes at design time, and these assigned priorities do not change, they remain constant for the lifetime of that process.However in the dynamic priority algorithm, the priorities are assigned to the processes at run time and this assignment is based on the execution parameters of the processes such as upcoming deadlines.

Of course, the static priority algorithms are simpler than the dynamic priority algorithms. There are other scheduling algorithms, like multilevel queues and multilevel feedback queues scheduling which are important for a system that provides better performance. Let's take a quick look into them.

Important note-01:

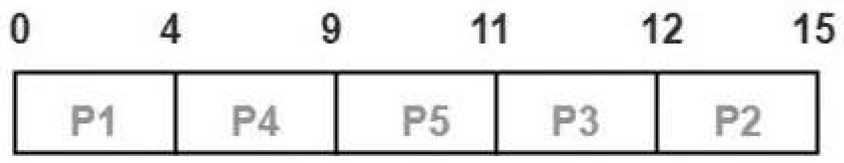
- The waiting time for the process having the highest priority will always be zero in preemptive mode.
- The waiting time for the process having the highest priority may not be zero in non-preemptive mode.

Important note-02:
Priority scheduling in preemptive and non-preemptive mode behaves exactly same under following conditions-

- The arrival time of all the processes is same
- All the processes become available

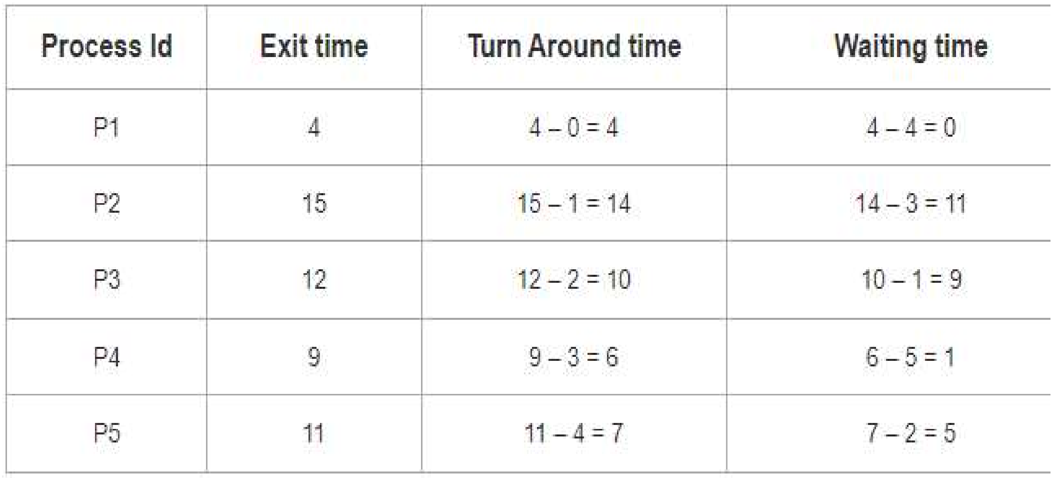
Solve: Consider the set of 5 processes whose arrival time and burst time are given below If the CPU scheduling policy is priority non-preemptive, calculate the average waiting time and average turnaround time. (Higher number represents higher priority)

Process Id	Arrival time	Burst time	Priority
P1	0	4	2
P2	1	3	3
P3	2	1	4
P4	3	5	5
P5	4	2	5



Gantt Chart

- Turn Around time = Exit time – Arrival time
- Waiting time = Turn Around time – Burst time



Process Id	Exit time	Turn Around time	Waiting time
P1	4	4 – 0 = 4	4 – 4 = 0
P2	15	15 – 1 = 14	14 – 3 = 11
P3	12	12 – 2 = 10	10 – 1 = 9
P4	9	9 – 3 = 6	6 – 5 = 1
P5	11	11 – 4 = 7	7 – 2 = 5

- Average Turn Around time = (4 + 14 + 10 + 6 + 7) / 5 = 41 / 5 = 8.2 unit
- Average waiting time = (0 + 11 + 9 + 1 + 5) / 5 = 26 / 5 = 5.2 unit

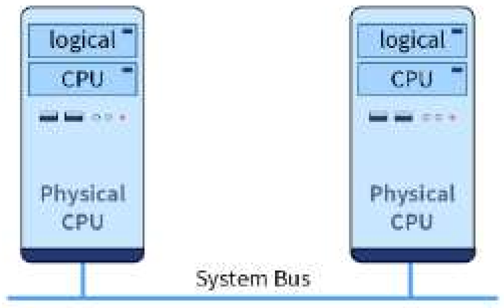
MULTIPLE-PROCESSOR SCHEDULING

A multi-processor is a system that has more than one processor but shares the same memory, bus, and input/output devices. The bus connects the processor to the RAM, to the I/O devices, and to all the other components of the computer.

The system is a tightly coupled system. This type of system works even if a processor goes down. The rest of the system keeps working. In multi-processor scheduling, more than one processors(CPUs) share the load to handle the execution of processes smoothly.

The scheduling process of a multi-processor is more complex than that of a single processor system because of the following reasons.

- Load balancing is a problem since more than one processors are present.
- Processes executing simultaneously may require access to shared data.
- Cache affinity should be considered in scheduling.



A Typical SMT architecture

Approaches to Multiple Processor Scheduling

Symmetric Multiprocessing: In symmetric multi-processor scheduling, the processors are self-scheduling. The scheduler for each processor checks the ready queue and selects a process to execute. Each of the processors works on the same copy of the operating system and communicates with each other. If one of the processors goes down, the rest of the system keeps working.

Symmetrical Scheduling with global queues: If the processes to be executed are in a common queue or a global queue, the scheduler for each processor checks this global-ready queue and selects a process to execute.

Symmetrical Scheduling with per queues: If the processors in the system have their own private ready queues, the scheduler for each processor checks their own private queue to select a process.

Asymmetric Multiprocessing: In asymmetric multi-processor scheduling, there is a master server, and the rest of them are slave servers. The master server handles all the scheduling processes and I/O processes, and the slave servers handle the users' processes. If the master server goes down, the whole system comes to a halt. However, if one of the slave servers goes down, the rest of the system keeps working.

Processor Affinity

A process has an affinity for a processor on which it runs. This is called processor affinity.

- When a process runs on a processor, the data accessed by the process most recently is populated in the cache memory of this processor. The following data access calls by the process are often satisfied by the cache memory.
- However, if this process is migrated to another processor due to some reason, the content of cache memory of the first processor is invalidated, and the second processor's cache memory has to be repopulated.
- To avoid the cost of invalidating and repopulating the cache memory, the Migration of processes from one processor to another is avoided.

There are two types of processor affinity.

Soft Affinity: The system has a rule of trying to keep running a process on the same processor but does not guarantee it. This is called soft affinity.

Hard Affinity: The system allows the process to specify the subset of processors on which it may run, i.e., each process can run only some of the processors. Systems such as Linux implement soft affinity, but they also provide system calls such as sched_setaffinity() to support hard affinity.

Load Balancing

In a multi-processor system, all processors may not have the same workload. Some may have a long ready queue, while others may be sitting idle. To solve this problem, load balancing comes into the picture. Load Balancing is the phenomenon of distributing workload so that the processors have an even workload in a symmetric multi-processor system.

In symmetric multiprocessing systems which have a global queue, load balancing is not required. In such a system, a processor examines the global ready queue and selects a process as soon as it becomes ideal.

However, in asymmetric multi-processor with private queues, some processors may end up idle while others have a high workload.

There are two ways to solve this.

Push Migration: In push migration, a task routinely checks the load on each processor. Some processors may have long queues while some are idle. If the workload is unevenly distributed, it will extract the load from the overloaded processor and assign the load to an idle or a less busy processor.

Pull Migration: In pull migration, an idle processor will extract the load from an overloaded processor itself.

Multi-Core Processors

A multi-core processor is a single computing component comprised of two or more CPUs called cores. Each core has a register set to maintain its architectural state and thus appears to the operating system as a separate physical processor. A processor register can hold an instruction, address, etc. Since each core has a register set, the system behaves as a multi-processor with each core as a processor.

Symmetric multiprocessing systems which use multi-core processors allow higher performance at low energy.

Symmetric multiprocessor

In Symmetric multi-processors, the memory has only one operating system, which can be run by any central processing unit. When a system call is made, the CPU on which the system call was made traps the kernel and processed that system call. The model works to balance processes and memory dynamically. As the name suggests, it uses symmetric multiprocessing to schedule processes, and every processor is self-scheduling. Each processor checks the global or private ready queue and selects a process to execute it.

Note: The kernel is the central component of the operating system. It connects the system hardware to the application software.

There are three ways of conflict that may arise in a symmetric multi-processor system. These are as follows.

Locking system: The resources in a multi-processor are shared among the processors. To make the access safe of these resources to the processors, a locking system is required. This is done to serialize the access of the resources by the processors.

Shared data: Since multiple processors are accessing the same data at any given time, the data may not be consistent across all of these processors. To avoid this, we must use some kind of strategy or locking scheme.

Cache Coherence: When the resource data is stored in multiple local caches and shared by many clients, it may be rendered invalid if one of the clients changes the memory block. This can be resolved by maintaining a consistent view of the data.

Master-Slave Multiprocessor

In a master-slave multi-processor, one CPU works as a master while all others works as slave processors. This means the master processor handles all the scheduling processes and the I/O processes while the slave processors handle the user's processes. The memory and input-output devices are shared among all the processors, and all the processors are connected to a common bus. It uses asymmetric multiprocessing to schedule processes.

Virtualization and Threading

Virtualization is the process of running multiple operating systems on a computer system. So a single CPU can also act as a multi-processor. This can be achieved by having a host operating system and other guest operating systems.

- Different applications run on different operating systems without interfering with one another.
- A virtual machine is a virtual environment that functions as a virtual computer with its CPU, memory, network interface, and storage, created on a physical hardware system.
- In a time-sharing OS, 100ms (millisecond) is allocated to each time slice to give users a reasonable response time. But it takes more than 100ms, maybe 1 second or more. This results in a poor response time for users logged into the virtual machine.
- Since the virtual operating systems receive a fraction of the available CPU cycles, the clocks in virtual machines may be incorrect. This is because their timers do not take any longer to trigger than they do on dedicated CPUs.

SYSTEM CALL INTERFACE FOR PROCESS MANAGEMENT

A system is used to create a new process or a duplicate process called a fork.

The duplicate process consists of all data in the file description and registers common. The original process is also called the parent process and the duplicate is called the child process.

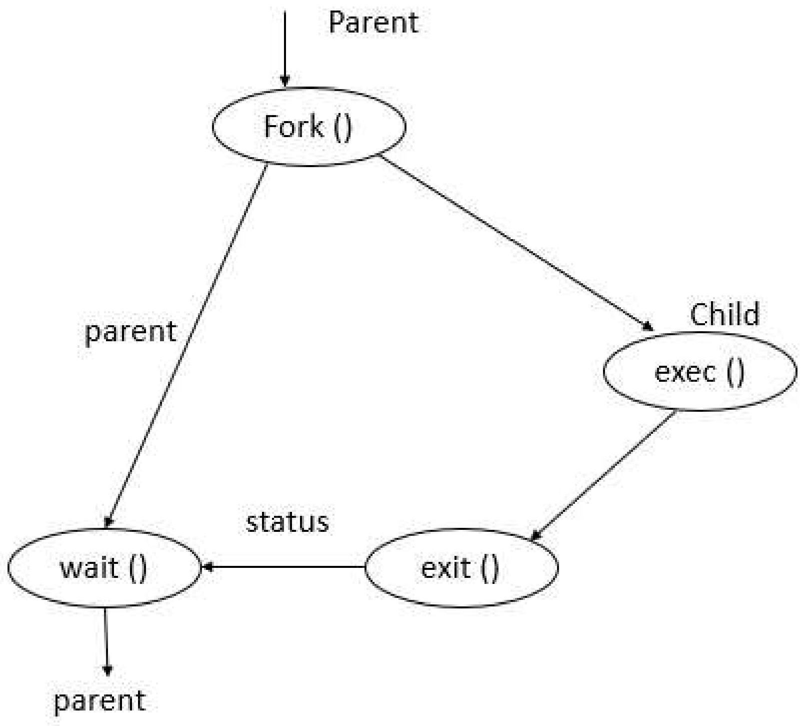
The fork call returns a value, which is zero in the child and equal to the child's PID (Process Identifier) in the parent. The system calls like exit would request the services for terminating a process.

Loading of programs or changing of the original image with duplicate needs execution of exec. Pid would help to distinguish between child and parent processes.

Process management system calls in Linux.

- **fork** − For creating a duplicate process from the parent process.
- **wait** − Processes are supposed to wait for other processes to complete their work.
- **exec** − Loads the selected program into the memory.
- **exit** − Terminates the process.

The pictorial representation of process management system calls is as follows −



fork() − A parent process always uses a fork for creating a new child process. The child process is generally called a copy of the parent. After execution of fork, both parent and child execute the same program in separate processes.

exec() − This function is used to replace the program executed by a process. The child sometimes may use exec after a fork for replacing the process memory space with a new program executable making the child execute a different program than the parent.

exit() − This function is used to terminate the process.

wait() − The parent uses a wait function to suspend execution till a child terminates. Using wait the parent can obtain the exit status of a terminated child.

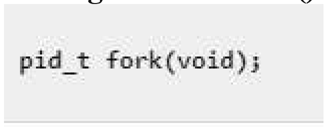
FUNCTIONS

1. The fork() Function

The **fork** function is used to create a process from within a process.
The resultant new process created by fork() is known as child process while the original process (from which fork() was called) becomes the parent process.

The function fork() is called once (in the parent process) but it returns twice. Once it returns in the parent process while the second time it returns in the child process. Note that the order of execution of the parent and the child may vary depending upon the process scheduling algorithm. So we see that fork function is used in **process creation**.

The signature of fork() is :



```
pid_t fork(void);
```

2. The exit() Funcion

The exit() is such a function or one of the system calls that is used to terminate the process. This system call defines that the thread execution is completed especially in the case of a multi-threaded environment. For future reference, the status of the process is captured.

After the use of exit() system call, all the resources used in the process are retrieved by the operating system and then terminate the process. The system call Exit() is equivalent to exit().

Its signature is :

```
#include <unistd.h>
void _exit(int status);
#include <stdlib.h>
void _Exit(int status);
```

3. **The exec Function**

Another set of functions that are generally used for creating a process is the **exec** family of functions. These functions are mainly used where there is a requirement of running an existing binary from withing a process.
For example, suppose we want to run the 'whoami' command from within a process, then in these kind of scenarios the exec() function or other members of this family is used. A point worth noting here is that with a call to any of the exec family of functions, the current process image is replaced by a new process image.A common member of this family is the execv() function.

Its signature is :

```
int execv(const char *path, char *const argv[]);
```

4. **The wait() and waitpid() Functions**

There are certain situations where when a child process terminates or changes state then the parent process should come to know about the change of the state or termination status of the child process. In that case functions like **wait()** are used by the parent process where the parent can query the change in state of the child process using these functions.

The signature of wait() is :

```
pid_t wait(int *status);
```

For the cases where a parent process has more than one child processes, there is a
function **waitpid()** that can be used by the parent process to query the change state of a particular child.

The signature of waitpid() is :

```
pid_t waitpid(pid_t pid, int *status, int options);
```

By default, waitpid() waits only for terminated children, but this behavior is modifiable via the options argument, as described below.

The value of pid can be:

- < -1 : Wait for any child process whose process group ID is equal to the absolute value of pid.
- -1 : Wait for any child process.
- 0 : Wait for any child process whose process group ID is equal to that of the calling process.
- > 0 : Wait for the child whose process ID is equal to the value of pid.

The value of options is an OR of zero or more of the following constants:

- WNOHANG : Return immediately if no child has exited.
- WUNTRACED : Also return if a child has stopped. Status for traced children which have stopped is provided even if this option is not specified.
- WCONTINUED : Also return if a stopped child has been resumed by delivery of SIGCONT.

UNIT-III
DEADLOCKS

DEADLOCKS

A set of processes is in a **Deadlock** state when every process in the set is waiting for an event that can be caused only by another process in the set. The events with which we are mainly concerned here are resource acquisition and resource release.

SYSTEM MODEL

A system consists of a finite number of resources to be distributed among a number of competing processes.

Resources are categorized into two types: Physical resources and Logical resources

- **Physical resources**: Printers, Tape drives, DVD drives, memory space and CPU cycles
- **Logical resources:** Semaphores, Mutex locks and files.

Each resource type consists of some number of identical instances. (i.e.) If a system has two CPU's then the resource type CPU has two instances.

A process may utilize a resource in the following sequence under normal mode of operation:

1. **Request**: The process requests the resource. If the resource is being used by another process then the request cannot be granted immediately then the requesting process must wait until it can acquire the resource.
2. **Use**: The process can operate on the resource.
   Example: If the resource is a printer, the process can print on the printer.
3. **Release**: The process releases the resource.

System calls for requesting and releasing resources:

- Device System calls: request( ) and release( )
- Semaphore System calls: wait( ), signal( )
- Mutex locks: acquire( ), release( ).
- Memory System Calls: allocate( ) and free( )
- File System calls: open( ), close( ).

A **System Table** maintains the status of each resource whether the resource is free or allocated. For each resource that is allocated, the table also records the process to which it is allocated. If a process requests a resource that is currently allocated to another process, it can be added to a queue of processes waiting for this resource.

FOUR NECESSARY CONDITIONS OF DEADLOCK

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

1. **Mutual exclusion**. Only one process at a time can use the resource. If other 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**. If a process holding a resource and the resource cannot be preempted until the process has completed its task.

4. **Circular wait**. A set *{P*0, *P*1, ..., *Pn}* of waiting processes must exist such that *P*0 is waiting for a resource held by *P*1, *P*1 is waiting for a resource held by *P*2, ..., *Pn*−1 is waiting for a resource held by *Pn* and *Pn* is waiting for a resource held by *P*0.

RESOURCE-ALLOCATION GRAPH

The resource allocation graph is used for identification of deadlocks in the system.
A **System Resource-Allocation Graph G={V,E}** is a directed graph that consists of a set of vertices *V* and a set of edges *E*.

The set of vertices *V* is partitioned into two types of nodes: Processes and Resources.

1. **Process set** P= *{P*1, *P*2, ..., *Pn}* consisting of all the active processes in the system.
2. **Resource set** R= *{R*1, *R*2, ..., *Rm}* consisting of all resource types in the system.

The set of Edges E is divided into types: Request Edge and Assignment Edge.

1. **Request Edge (Pi → Rj):** It signifies that process *Pi* has requested an instance of resource type *Rj* and **Pi** is currently waiting for the resource **Rj**.
2. **Assignment edge (Rj → Pi):** It signifies that an instance of resource type *Rj* has been allocated to process *Pi*.

Processes can be represented in **Circles** and **Resources** can be represented in **Rectangles**. **Instance** of resource can be represented by a **Dot**.

Note:

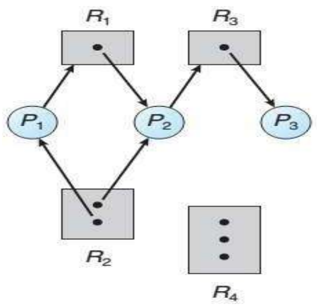
1. When process *Pi* requests an instance of resource type *Rj*, a request edge is inserted in the Resource-allocation graph.
2. When this request can be fulfilled, the request edge is transformed to an assignment edge.
3. When the process no longer needs access to the resource, it releases the resource and the assignment edge is deleted.

Resource allocation graph shows three situations:

1. Graph with No deadlock
2. Graph with a cycle and deadlock
3. Graph with a cycle and no deadlock

Resource Allocation Graph without Deadlock

The below graph consists of three sets: Process **P**, Resources **R** and Edges **E**.



- Process set P= {P1, P2, P3}.
- Resources set R= {R1, R2, R3, R4}.
- Edge set E= E = {P1 → R1, P2 → R3, R1 → P2, R2 → P2, R2 → P1, R3 → P3}.

Resource type R1 and R3 has only one instance and R2 has two instances and R4 has three instances.

The Resource Allocation Graph depicts that:

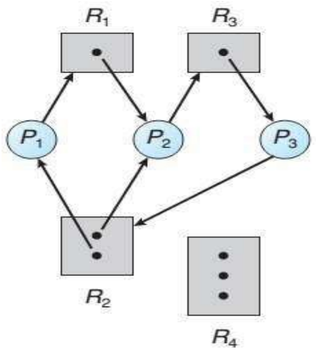
- **R2 → P1, P1 → R1:** P1 is holding an instance of resource type R2 and is waiting for an instance of R1.
- **R1 → P2, R2 → P2, P2 → R3:** Process P2 is holding an instance of R1 and an instance of R2 and is waiting for an instance of R3.
- **R3 → P3:** Process P3 is holding an instance of R3.

The above Resource allocation graph does not contain any cycle then there is no process in the system is deadlocked.

Note:

1. If each resource type has exactly one instance then a cycle implies a deadlock.
2. If each resource type has several instances then a cycle does not necessarily imply that a deadlock has occurred.

Resource Allocation Graph with a Cycle and Deadlock



Consider the above graph, with processes and Resources and have some edges:

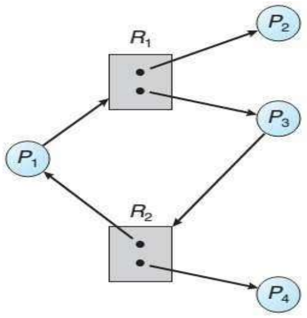
P1 → R1 → P2 → R3 → P3 → R2 → P1

P2 → R3 → P3 → R2 → P2

- Process P2 is waiting for the resource R3, which is held by process P3.
- Process P3 is waiting for either process P1 or process P2 to release resource R2.
- Process P1 is waiting for process P2 to release resource R1.

Hence the Processes *P*1, *P*2 and *P*3 are deadlocked.

Resource Allocation Graph with a Cycle and No Deadlock



The graph has a cycle: P1 → R1 → P3 → R2 → P1.

- This cycle does not lead to deadlock, because the process P4 and P2 is not waiting for any resource.
- Process *P*4 may release its instance of resource type *R*2. That resource can then be allocated to *P*3, breaking the cycle.

METHODS FOR HANDLING DEADLOCKS

The Deadlock can be handled by 3 methods:

1. Deadlock Prevention
2. Deadlock Avoidance
3. Deadlock Detection and Recovery

DEADLOCK PREVENTION

Deadlock prevention provides a set of methods to ensure that at least one of the necessary conditions cannot hold. (i.e.) Deadlock can be prevented if any of mutual exclusion, Hold and wait, No preemption and Circular wait condition cannot hold.

Mutual Exclusion

The mutual exclusion condition must hold when at least one resource must be non-sharable.

- We cannot prevent deadlocks by denying the mutual-exclusion condition, because some resources by default are nonsharable.
  Example 1: A mutex lock cannot be simultaneously shared by several processes.
  Example 2: Printer is a resource where only one process can use it.
- Sharable resources do not require mutually exclusive access and thus cannot be involved in a deadlock. Example: Read-only files.
- If several processes attempt to open a read-only file at the same time, they can be granted simultaneous access to the file. A process never needs to wait for a sharable resource.

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.

- **Protocol 1:** Each process can request the resources and be allocated all its resources before it begins execution. We can implement this provision by requiring that system calls requesting resources for a process precede all other system calls.
  Example: Consider a process that copies data from a **DVD drive** to a file on **Hard disk**, sorts the file and then prints the results to a **Printer**.
  If all resources must be requested at the beginning of the process, then the process must initially request the **DVD drive**, **disk file** and **Printer**. It will hold the printer for its entire execution, even though it needs the printer only at the end.
- **Protocol 2:** A process can be allowed to request resources only when it has none. A process may request some resources and use them. Before it can request any additional resources, it must release all the resources that it is currently allocated.
  Example: Consider a process that copies data from a **DVD drive** to a file on **Hard disk**, sorts the file and then prints the results to a **Printer**.
  The process to request initially **only** the **DVD drive** and **Hard disk** file. It copies from the **DVD drive** to the **Hard disk** and then releases both the DVD drive and the disk file. The process must then request the **Hard disk** file and the **Printer**. After copying the disk file to the printer, it releases these two resources and terminates.

Problem: Starvation and Low Resource utilization

- Resource utilization is low, since resources may be allocated but unused for a long period.
- A process that needs several resources may have to wait indefinitely leads to starvation.

No Preemption

To ensure that No preemption condition does not hold, we can 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 the process is currently holding are preempted (i.e.) 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 resources that it is requesting.

Note: This protocol is often applied to resources whose state can be easily saved and restored later such as CPU registers and memory space. It cannot be applied to resources such as mutex locks and semaphores.

Circular Wait

One way to ensure that circular wait 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.

Consider the set of resource types R={R1, R2, ..., Rm} and **N** be the set of natural numbers.
we define a one-to-one function **F: R → N**.

- The function assigns each resource type to a unique integer number, which allows us to compare two resources and to determine whether one resource precedes another resource in our ordering.

Example: If the set of resource types R includes tape drives, disk drives and printers, then the function **F: R → N** might be defined as follows:

F (Tape drive) = 1 (F: Tape drive → 1)
F (Disk drive) = 5 (F: Disk drive → 1)
F (Printer) = 12 (F: Printer → 1)

We can now consider the following protocol to prevent deadlocks:

- Each process can request resources only in an increasing order of enumeration. That is, a process can initially request any number of instances of a resource type Ri.
- After that, the process can request instances of resource type Rj iff **F(Rj) > F(Ri)**
- Example: A process that wants to use the tape drive and printer at the same time must first request the tape drive and then request the printer.
- Alternatively, we can require that a process requesting an instance of resource type Rj must have released any resources Ri such that **F(Ri) ≥ F(Rj)**.

Note: If several instances of the same resource type are needed, a **single** request for all of them must be issued.

Disadvantage of Deadlock Prevention

Deadlock-prevention algorithms leads to low resource utilization and the system throughput will be reduced.

DEADLOCK AVOIDANCE

In Deadlock avoidance the processes first informs the operating system about their maximum allocation of resources to be requested and used during its life time.

- With this information, the operating system can decide for each request whether the resource will be granted immediately or the process should wait for resources.
- To take this decision about decision about the resource allocation, the operating system must consider the resources currently available, the resources currently allocated to each process and the future requests and releases of each process.

Algorithms for Deadlock Avoidance

A deadlock-avoidance algorithm dynamically examines the **Resource-Allocation State** to ensure that a circular-wait condition can never exist.

The Resource Allocation **State** is defined by the number of available resources and allocated resources and the maximum demands of the processes.

There are three algorithms are designed for deadlock avoidance:

1. Safe State
2. Resource Allocation Graph Algorithm
3. Bankers Algorithm

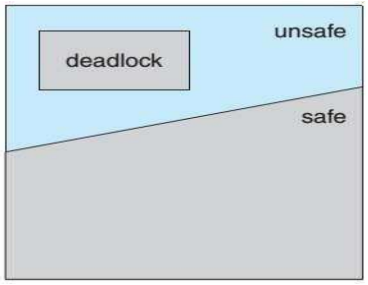
Safe State Algorithm

If the system can allocate resources to each process up to its maximum in some order and still avoid a deadlock then the state is called Safe state.

- A system is in a safe state only if there exists a **Safe sequence**.
- A sequence of processes <P1, P2, ..., Pn> is a safe sequence for the current allocation state, if for each process **Pi**, the resource requests that Pi can still make can be satisfied by the currently available resources plus the resources held by all Pj, with j < i.
- In this situation, if the resources that Pi needs are not immediately available, then Pi can wait until all Pj have finished.
- When Pj have finished its task, Pi can obtain all of its needed resources and after completing its designated task Pi can return its allocated resources and terminate.
- When P_i terminates, P_{i+1} can obtain its needed resources and so on.
- If no such sequence exists, then the system state is said to be unsafe.

Note:

1. A safe state is not a deadlocked state and a deadlocked state is an unsafe state.
2. An unsafe state ***may*** lead to a deadlock but **not all** unsafe states are deadlocks.
3. As long as the state is safe, the operating system can avoid unsafe and deadlocked states.
4. In an unsafe state, operating system cannot prevent processes from requesting resources in such a way that a deadlock occurs. Behavior of the processes controls unsafe states.



Example: Consider a system with 12 magnetic tape drives and 3 processes: *P*0, *P*1 and *P*2.

Process	Maximum Needs	Current Needs
P0	10	5
P1	4	2
P2	9	2

The above table describes as follows:

- Process **P0** requires 10 tape drives, **P1** needs 4 tape drives and **P2** need 9 tape drives.
- At time t0, process P0 is holding 5 tape drives, P1 and P2 is holding 2 tape drives each.
- Now there are 3 free tape drives.

At time **t0**, the system is in a safe state. <*P*1, *P*0, *P*2> sequence satisfies the safety condition.

- Process *P*1 can immediately be allocated all its tape drives and then return all 4 resources. (i.e.) P1 currently holding 2 tape drives and out of 3 free tape drives 2 tape drives will be given to P1. Now P1 is having all 4 resources. Hence P1 will use all of its resources and after completing its task P1 releases all 4 resources and then returns to the system. Now the system is having **5 available** tape drives.
- Now process P0 needs 5 tape drives and the system has 5 available tape drives. Hence **P0** can get all its tape drives and it reaches its maximum 10 tape drives. After completing its task P0 returns the resources to the system. Now system has 10 available tape drives.
- Now the process P2 needs 7 additional resources and system have 10 resources available. Hence process *P*2 can get all its tape drives and return them. Now the system will have all 12 tape drives available.

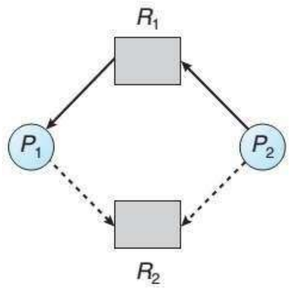
Problem: Low Resource utilization

If a process requests a resource that is currently available, it may still have to wait. Hence there exist a low resource utilization is possible.

<u>Resource-Allocation-Graph Algorithm</u>

In this algorithm we use three edges: request edge, assignment edge and a **claim edge**.

- Claim edge **Pi** → **Rj** indicates that process **Pi** may request resource **Rj** at some time in the future.
- Claim edge resembles a request edge in direction but is represented by dashed line.
- When process **Pi** requests resource **Rj**, the claim edge **Pi** → **Rj** is converted to a request edge.
- When a resource **Rj** is released by **Pi**, the assignment edge **Rj** → **Pi** is reconverted to a claim edge **Pi** → **Rj**.
- The resources must be claimed a priori in the system. That is, before process *Pi* starts executing, all its claim edges must already appear in the resource-allocation graph.



Now suppose that process Pi requests resource Rj.

- The request can be granted only if converting the request edge **Pi → Rj** to an assignment edge **Rj → Pi** does not result in the formation of a cycle in the resource-allocation graph. We check for safety by using a cycle-detection algorithm.
- If no cycle exists, then the allocation of the resource will leave the system in a safe state.
- If a cycle is found, then the allocation will put the system in an unsafe state. In that case, process Pi will have to wait for its requests to be satisfied.

Example: consider the above resource-allocation graph. Suppose that P2 requests R2.

- R2 is currently free still we cannot allocate it to P2, since this will create a cycle in graph.
- A cycle indicates that the system is in an unsafe state.
- If P1 requests R2 and P2 requests R1, then a deadlock will occur.

Problem: The resource-allocation-graph algorithm is not applicable to a resource allocation system with multiple instances of each resource type.

BANKER's ALGORITHM

Banker's algorithm is used in a system with multiple instance of each resource type.
The name was chosen because the algorithm could be used in a banking system to ensure that the bank never allocated its available cash in such a way that it could no longer satisfy the needs of all its customers.
Banker's algorithm uses two algorithms:

1. Safety algorithm
2. Resource-Request algorithm

Process of Banker's algorithm:

- When a new process enters the system, the process must declare the **Maximum** number of instances of each resource type that it may need.
- The **Maximum** number may not exceed the **Total** number of resources in the system.
- When a user requests a set of resources, the system must determine whether the allocation of these resources will leave the system in a safe state.
- If the system is in safe state then the resources are allocated.
- If the system is in unsafe state then the process must wait until some other process releases enough resources.

Data structures used to implement the Banker's algorithm

Consider the system with **n** number of processes and m number of resource types:

- **Available_m:** A vector of length m indicates the number of available resources of each type.
- **$\text{Max}_{n \times m}$:** An n × m matrix defines the maximum demand of each process.
- **$\text{Allocation}_{n \times m}$:** An n × m matrix defines the number of resources of each type currently allocated to each process.
- **$\text{Need}_{n \times m}$:** An n × m matrix indicates the remaining resource need of each process.

$$\textbf{Need[i][j] = Max[i][j]} - \textbf{Allocation[i][j].}$$

- **Available[j] = k** means then k instances of resource type Rj are available.
- **Max[i][j] = k** means process Pi may request at most k instances of resource type Rj.

- **Allocation[i][j]=k** means process Pi is currently allocated k instances of resource type Rj.
- **Need[i][j]=k** means process Pi may need k more instances of resource type Rj to complete its task.

Each row in the matrices **Allocation** $_{n \times m}$ and **Need** $_{n \times m}$ are considered as vectors and refer to them as $\mathbf{Allocation_i}$ and $\mathbf{Need_i}$.

- The vector $\mathbf{Allocation_i}$ specifies the resources currently allocated to process Pi.
- The vector $\mathbf{Need_i}$ specifies the additional resources that process Pi may still request to complete its task.

Safety algorithm

Safety algorithm finds out whether the system is in safe state or not. The algorithm can be described as follows:

1. Let **Work** and **Finish** be vectors of length m and n, respectively. We initialize

   Work = **Available**

   Finish[**i**] = **false** for i = 0, 1, ..., n − 1.
2. Find an index i such that both

   Finish[i] == **false**

   $\mathbf{Need_i} \leq \mathbf{Work}$

   If no such i exists, go to step 4.
3. $\mathbf{Work} = \mathbf{Work} + \mathbf{Allocation_i}$

   Finish[i] = **true**

   Go to step 2.
4. If **Finish**[i] == **true** for all i, then the system is in a safe state.

Note: To determine a safe state, this algorithm requires an order of $\mathbf{m \times n^2}$ operations.

Resource-Request Algorithm

This algorithm determines whether requests can be safely granted.

Let $\mathbf{Request_i}$ be the request vector for process Pi. If $\mathbf{Request_i\ [\ j] == k}$, then process Pi wants k instances of resource type Rj.

When a request for resources is made by process Pi, the following actions are taken:

1. If $\mathbf{Request_i} \leq \mathbf{Need_i}$, go to step 2.

   Otherwise, raise an error condition, since the process has exceeded its maximum claim.
2. If $\mathbf{Request_i} \leq$ **Available,** go to step 3.

   Otherwise, Pi must wait, since the resources are not available.
3. Have the system pretend to have allocated the requested resources to process Pi by modifying the state as follows:

   $\mathbf{Available = Available - Request_i}$;

   $\mathbf{Allocation_i = Allocation_i + Request_i}$;

   $\mathbf{Need_i = Need_i - Request_i}$;
4. If the resulting resource-allocation state is safe, the transaction is completed and process Pi is allocated its resources.

   If the new state is unsafe, then Pi must wait for $\mathbf{Request_i}$ and the old resource-allocation state is restored.

Example for Banker's Algorithm

Consider a system with 5 processes: **P0, P1, P2, P3, P4** and 3 resource types **A, B** and **C** with **10, 5, 7** instances respectively. (i.e.) . Resource type **A=10, B= 5** and **C=7** instances.

Suppose that, at time T0, the following snapshot of the system has been taken:

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

The Available vector can be calculated by subtractring total no of resources from the sum of resources allocated to each process.

Available resources of A= Total resources of A – Sum of resources allocated to Process P1 to P4

The Need matrix can be obtained by using **Need[i][j] = Max[i][j]−Allocation[i][j]**

	Max			Allocation			**Need**		
	A	B	C	A	B	C	**A**	**B**	**C**
P0	7	5	3	0	1	0	**7**	**4**	**3**
P1	3	2	2	2	0	0	**1**	**2**	**2**
P2	9	0	2	3	0	2	**6**	**0**	**0**
P3	2	2	2	2	1	1	**0**	**1**	**1**
P4	4	3	3	0	0	2	**4**	**3**	**1**

By using the banker's algorithm we can decide whether the state is safe or not.

After solving the above problem by using bankers algorithm we will get to a safe state with safe sequence <P1,P3,P4,P0,P2>.

Now we get a safe state, the resources will be granted immediately for requested process P1.

DEADLOCK DETECTION ALGORITHM

If a system does not employ either a Deadlock-Prevention or a Deadlock-Avoidance algorithm then a deadlock situation may occur. In this environment, the system may provide:

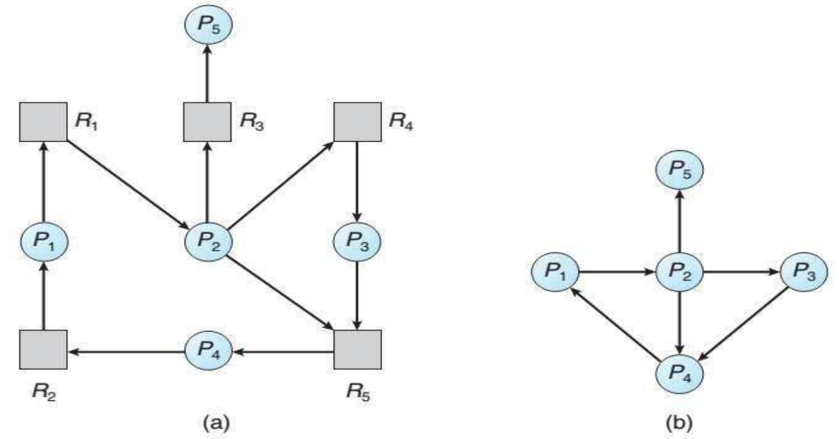
- An algorithm that examines the state of the system to determine whether a deadlock has occurred
- An algorithm to recover from the deadlock.

Deadlock Detection in Single Instance of Each Resource Type

If all resources have only a single instance then we can define a Deadlock-Detection algorithm that uses a variant of the resource-allocation graph called a **wait-for** graph.

We obtain wait-for graph from the resource-allocation graph by removing the resource nodes and collapsing the appropriate edges.

- An edge from **Pi** to **Pj** in a wait-for graph implies that process **Pi** is waiting for process **Pj** to release a resource that **Pi** needs.
- An edge **Pi → Pj** exists in a wait-for graph if and only if the corresponding resource allocation graph contains two edges **Pi → Rq** and **Rq → Pj** for some resource **Rq** .



- In above figure we present a resource-allocation graph and the corresponding wait-for graph. 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 **invoke an algorithm** that searches for a cycle in the graph.
- An algorithm to detect a cycle in a graph requires an order of $\mathbf{n^2}$ 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.

We will implement a Deadlock Detection algorithm that is similar to the Banker's algorithm. The data structures used in Deadlock Detection algorithm is:

- **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 each process.
- **Request:** An **n** × **m** matrix indicates the current request of each process.
  If **Request[i][j]==k**, then process Pi is requesting **k** more instances of resource type **Rj**.

The detection algorithm described here simply investigates every possible allocation sequence for the processes that remain to be completed.

```
1. Let Work and Finish be vectors of length m and n, respectively. We Initialize
            Work = Available. For i = 0, 1, ..., n–1.
            if Allocation_i != 0, then Finish[i] = false.
            Otherwise, Finish[i] = true.
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, then process Pi is deadlocked.
```

This algorithm requires an order of **m** × **n**^{2} operations to detect whether the system is in a deadlocked state.

Example:

Consider a system with 5 processes: **P0, P1, P2, P3, P4** and 3 resource types **A, B** and **C** with **10, 5, 7** instances respectively. (i.e.) . Resource type **A=7, B= 2** and **C=6** instances.

Suppose that, at time T0, we have the following resource-allocation state:

	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			

Initially the system is not in Deadlock State. If we apply the Deadlock Detection algorithm we will find the sequence < P0, P2, P3, P1, P4 > results in **Finish**[i] == true for all i.

The system is in safe state hence there is no deadlock.

RECOVERY FROM DEADLOCK

There are two options for breaking a deadlock.

1. Process termination
2. Resource Preemption

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 great expense. The deadlocked processes may have computed for a long time and the results of these partial computations must be discarded and probably will have to be 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.

Many factors may affect which process is chosen for preempting includes:

1. Priority of the process.
2. How long the process has computed and how much longer the process will compute before completing its designated task.
3. How many and what types of resources the process has used.
4. How many more resources the process needs in order to complete.
5. How many processes will need to be terminated?
6. Whether the process is Interactive or Batch.

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. There are 3 issues related to Resource Preemption:

1. **Selecting a victim**. As in process termination, we must determine the order of preemption to minimize cost. Cost factors may include the number of resources a deadlocked process is holding and the amount of time the process has thus far consumed.
2. **Rollback**. If we preempt a resource from a process then the process cannot continue with its normal execution. It is missing some needed resource. We must do total roll back of the process and restart it from that state: abort the process and then restart it.
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?
   In a system where victim selection is based on cost factors, it may happen that the same process is always picked as a victim. As a result, this process never completes its task which leads to starvation. Hence we must ensure that a process can be picked as a victim only a finite number of times. The solution is to include the number of rollbacks in the cost factor.

MAHAVEER INSTITUTE OF SCIENCE AND TECHNOLOGY
(AN UGC AUTONOMOUS INSTITUTION)
Approved by AICTE, Affiliated to JNTUH, Accredited by NAAC with 'A' Grade
Recognized Under Section 2(f) of UGC Act 1956, ISO 9001:2015 Certified
Vyasapuri, Bandlaguda, Post: Keshavgiri, Hyderabad- 500 005, Telangana, India.
https://www.mist.ac.in E-mail:principal.mahaveer@gmail.com, Mobile: 8978380692

ESTD : 2001

Department of Computer Science and Engineering (AIML)

(R22) OPERATING SYSTEM

Lecture Notes

B. Tech II YEAR – I SEM

Prepared by

SANGYAM SOUNDARYA
(Assistant Professor)
Dept.CSE(AIML)

OPERATING SYSTEMS

B.Tech. II Year I Sem.

L	T	P	C
3	0	0	3

Prerequisites:
1. A course on "Computer Programming and Data Structures".
2. A course on "Computer Organization and Architecture".

Course Objectives:
- Introduce operating system concepts (i.e., processes, threads, scheduling, synchronization, deadlocks, memory management, file and I/O subsystems and protection)
- Introduce the issues to be considered in the design and development of operating system
- Introduce basic Unix commands, system call interface for process management, interprocess communication and I/O in Unix

Course Outcomes:
- Will be able to control access to a computer and the files that may be shared
- Demonstrate the knowledge of the components of computers and their respective roles in computing.
- Ability to recognize and resolve user problems with standard operating environments.
- Gain practical knowledge of how programming languages, operating systems, and architectures interact and how to use each effectively.

UNIT - I
Operating System - Introduction, Structures - Simple Batch, Multiprogrammed, Time-shared, Personal Computer, Parallel, Distributed Systems, Real-Time Systems, System components, Operating System services, System Calls
Process - Process concepts and scheduling, Operations on processes, Cooperating Processes, Threads

UNIT - II
CPU Scheduling - Scheduling Criteria, Scheduling Algorithms, Multiple -Processor Scheduling. System call interface for process management-fork, exit, wait, waitpid, exec
Deadlocks - System Model, Deadlocks Characterization, Methods for Handling Deadlocks, Deadlock Prevention, Deadlock Avoidance, Deadlock Detection, and Recovery from Deadlock

UNIT - III
Process Management and Synchronization - The Critical Section Problem, Synchronization Hardware, Semaphores, and Classical Problems of Synchronization, Critical Regions, Monitors
Interprocess Communication Mechanisms: IPC between processes on a single computer system, IPC between processes on different systems, using pipes, FIFOs, message queues, shared memory.

UNIT - IV
Memory Management and Virtual Memory - Logical versus Physical Address Space, Swapping, Contiguous Allocation, Paging, Segmentation, Segmentation with Paging, Demand Paging, Page Replacement, Page Replacement Algorithms.

UNIT - V
File System Interface and Operations -Access methods, Directory Structure, Protection, File System Structure, Allocation methods, Free-space Management. Usage of open, create, read, write, close, lseek, stat, ioctl system calls.

TEXT BOOKS:

1. Operating System Principles- Abraham Silberchatz, Peter B. Galvin, Greg Gagne 7th Edition, John Wiley.
2. Advanced programming in the UNIX environment, W.R. Stevens, Pearson education.

REFERENCE BOOKS:

1. Operating Systems- Internals and Design Principles, William Stallings, Fifth Edition–2005, Pearson Education/PHI
2. Operating System A Design Approach- Crowley, TMH.
3. Modern Operating Systems, Andrew S. Tanenbaum 2nd edition, Pearson/PHI
4. UNIX programming environment, Kernighan and Pike, PHI/ Pearson Education
5. UNIX Internals -The New Frontiers, U. Vahalia, Pearson Education.

UNIT III

UNIT-III
PROCESS MANAGEMENT AND SYNCHRONIZATION

Processes executing concurrently in the operating system may be either independent processes or cooperating processes.

1. **Independent Process:** Any process that does not share data with any other process. An Independent process does not affect or be affected by the other processes executing in the system.
2. **Cooperating Process:** Any process that shares data with other processes. A cooperating process can affect or be affected by the other processes executing in the system. Cooperating processes require an **Inter-Process Communication (IPC)** mechanism that will allow them to exchange data and information.

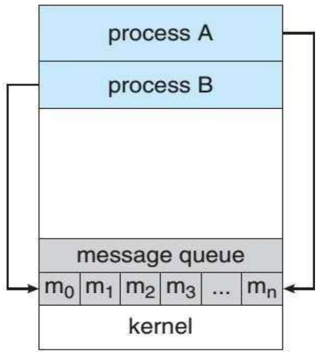
Reasons for providing cooperative process environment:

- **Information Sharing**: Several users may be interested in the same piece of information (i.e. a shared file), we must provide an environment to allow concurrent access to such information.
- **Computation Speedup:** If we want a particular task to run faster, we must break it into subtasks. Each task will be executing in parallel with the other tasks.
- **Modularity**: Dividing the system functions into separate processes or threads.
- **Convenience**: Even an individual user may work on many tasks at the same time. For example a user may be editing, listening to music and compiling in parallel.

There are two models of IPC: **Message passing** and **Shared memory**.

Message-Passing Systems

In the message-passing model, communication takes place by means of messages exchanged between the cooperating processes.



- Message passing provides a mechanism to allow processes to communicate and to synchronize their actions without sharing the same address space.
- It is particularly useful in a distributed environment, where the communicating processes may reside on different computers connected by a network.
- A message-passing facility provides two operations: send, receive.
- Messages sent by a process can be either fixed or variable in size.

Example: An Internet chat program could be designed so that chat participants communicate with one another by exchanging messages.

P and Q are two processes wants to communicate with each other then they send and receive messages to each other through a communication link such as Hardware bus or Network.
Methods for implementing a logical communication links are:
1. Naming
2. Synchronization
3. Buffering

Naming
Processes that want to communicate use either **Direct** or **Indirect** communication.
In **Direct communication**, each process that wants to communicate must explicitly name the recipient or sender of the communication.
- send(P, message) — Send a message to process P.
- receive(Q, message) — Receive a message from process Q.

A communication link in direct communication scheme has the following properties:
- A link is established automatically between every pair of processes that want to communicate. The processes need to know only each other's identity to communicate.
- A link is associated with exactly two processes. (i.e.) between each pair of processes, there exists exactly one link.

In **Indirect communication**, the messages are sent to and received from **mailboxes** or **ports**.
- A mailbox can be viewed abstractly as an object into which messages can be placed by processes and from which messages can be removed.
- Each mailbox has a unique integer identification value.

A process can communicate with another process via a number of different mailboxes, but two processes can communicate only if they have a shared mailbox.
- send (A, message) — Send a message to mailbox A.
- receive (A, message) — Receive a message from mailbox A.

In this scheme, a communication link has the following properties:
- A link is established between a pair of processes only if both members of the pair have a shared mailbox.
- A link may be associated with more than two processes.
- Between each pair of communicating processes, a number of different links may exist, with each link corresponding to one mailbox.

Synchronization
Message passing done in two ways:
1. Synchronous or Blocking
2. Asynchronous or Non-Blocking

- **Blocking send:** The sending process is blocked until the message is received by the receiving process or by the mailbox.
- **Non-blocking send:** The sending process sends the message and resumes operation.
- **Blocking receive:** The receiver blocks until a message is available.
- **Non-blocking receive:** The receiver retrieves either a valid message or a null.

Buffering

Messages exchanged by communicating processes reside in a temporary queue. Those queues can be implemented in three ways:

1. **Zero Capacity:** Zero-capacity is called as a message system with no buffering. The sender must block until the recipient receives the message.
2. **Bounded Capacity:** The queue has finite length n. Hence at most n messages can reside in it. If the queue is not full when a new message is sent, the message is placed in the queue and the sender can continue execution without waiting. If the link is full, the sender must block until space is available in the queue.
3. **Unbounded Capacity:** The queue's length is potentially infinite. Hence any number of messages can wait in it. The sender never blocks.

IPC - Pipes in UNIX

Pipes were one of the first IPC mechanisms in early UNIX systems.

The pipes in UNIX are categorized into two types: **Ordinary Pipes** and **Named Pipes**

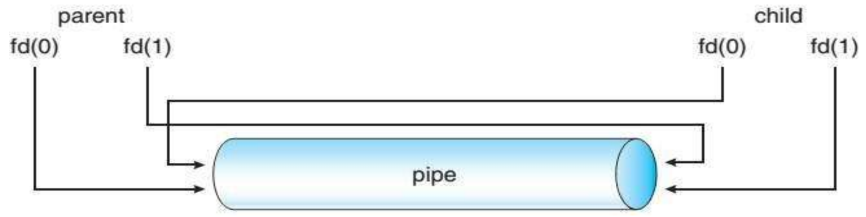
Ordinary Pipes

- Ordinary pipes allow two processes to communicate in standard producer–consumer fashion.
- The producer writes to one end of the pipe (**write-end**) and the consumer reads from the other end (**read-end**).
- Ordinary pipes are unidirectional which allows only one-way communication. If two-way communication is required, two pipes must be used. Each pipe transfer data in a different direction.

On UNIX systems ordinary pipes are constructed using the function:

pipe(int fd[ ])

- This function creates a pipe that is accessed through the int fd[ ] file descriptors: fd[0] is the read-end of the pipe and fd[1] is the write-end.
- UNIX treats a pipe as a special type of file. Thus, pipes can be accessed using ordinary read( ) and write( ) system calls.



An ordinary pipe cannot be accessed from outside the process that created it.

- A parent process creates a pipe and uses it to communicate with a child process.
- A child process inherits open files from its parent. Since a pipe is a special type of file, the child inherits the pipe from its parent process.
- If a parent writes to the pipe then the child reads from pipe.

Named Pipes

Named pipe provides the bidirectional communication and no parent–child relationship is required.

- Once a named pipe is established, several processes can use it for communication. A named pipe has several writers.
- Named pipes continue to exist after communicating processes have finished.
- Both UNIX and Windows systems support named pipes.

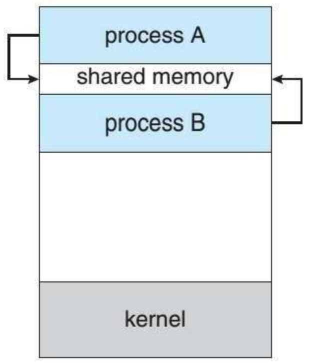
Named pipes are referred to as FIFOs in UNIX systems.

- Once Named pipes are created, they appear as typical files in the file system.
- FIFO is created with the mkfifo( ) system call and manipulated with the ordinary open( ), read( ), write( ) and close( ) system calls.
- It will continue to exist until it is explicitly deleted from the file system.
- Although FIFOs allow bidirectional communication, only one-way transmission is permitted. If data must travel in both directions, two FIFOs are used. The communicating processes must reside on the same machine.
- If inter-machine communication is required, sockets must be used.

Shared Memory Systems

In the shared-memory model, a region of memory will be shared by cooperating processes.

- Processes can exchange information by reading and writing data to the shared region.
- A shared-memory region (segment) resides in the address space of the process.
- Other processes that wish to communicate using this shared-memory segment must attach it to their address space.



- Normally, the operating system tries to prevent one process from accessing another process's memory.
- Shared memory requires that two or more processes agree to remove this restriction. They can then exchange information by reading and writing data in the shared areas.
- The form of the data and the location are determined by these processes and are not under the operating system's control.
- The processes are also responsible for ensuring that they are not writing to the same location simultaneously.

Producer-Consumer Problem in Cooperative process

A producer process produces information that is consumed by a consumer process.

Example: A compiler may produce assembly code that is consumed by an assembler. The assembler may produce object modules that are consumed by the loader.

One solution to the producer–consumer problem uses shared memory.

- To allow producer and consumer processes to run concurrently, we must have available a buffer of items that can be filled by the producer and emptied by the consumer.
- This buffer will reside in a region of memory that is shared by the producer and consumer processes.
- A producer can produce one item while the consumer is consuming another item.
- The producer and consumer must be synchronized, so that the consumer does not try to consume an item that has not yet been produced.

Two types of buffers can be used. Unbounded Buffer and Bounded Buffer

- **Unbounded Buffer:** The size of the buffer is not limited. The consumer may have to wait for new items, but the producer can always produce new items.
- **Bounded Buffer:** The buffer size is limited. The consumer must wait if the buffer is empty and the producer must wait if the buffer is full.

Code for Producer and Consumer Process in Bounded Buffer IPC

Following variables reside in shared memory region by the producer and consumer processes:

```
#define BUFFER_SIZE 10
typedef struct {
. . . . . . . . . . . .
}item;
item buffer[BUFFER_SIZE];
int in = 0;
int out = 0;
```

Producer process code is as follows:

```
while (true)
{
        /* produce an item in next_produced */
while (counter == BUFFER_SIZE) ; /*do nothing Buffer
full */
buffer[in] = next_produced;
in = (in + 1) % BUFFER_SIZE;
counter++;
}
```

Consumer process code is as follows:

```
while (true)
{
  while (counter == 0); /* do nothing Buffer Empty */
  next_consumed = buffer[out];
out = (out + 1) % BUFFER SIZE;
counter--;
  /* consume the item in next_consumed */
}
```

The shared buffer is implemented as a circular array with two logical pointers: in and out.

- The variable –**in**‖ points to the next free position in the buffer and –**out**‖ points to the first full position in the buffer.
- An integer variable counter is initialized to 0. Counter is incremented every time we add a new item to the buffer and is decremented every time we remove one item from the buffer.
- The buffer is empty when counter== 0 and the buffer is full when counter== Buffer_size.
- The producer process has a local variable next_produced in which the new item to be produced is stored.
- The consumer process has a local variable next_consumed in which the item to be consumed is stored.

Note: Although the producer and consumer routines shown above are correct separately, they may not function correctly when executed concurrently.

Example: Let us consider the **counter =5**. Producer and consumer processes concurrently execute the statements –**counter++**‖ and –**counter--**‖.

Let R1 and R2 are two registers. the statement –counter++‖ may be implemented in machine language as follows:

T0: R1 = counter
T1: R1 = R1 + 1
T2: counter = R1

where register1 is one of the local CPU registers. Similarly, the statement –counter--‖ is implemented as follows:

T3: R2 = counter
T4: R2 = R2 – 1
T5: counter = R2

We execute the statements in the order **T0, T1, T2, T3, T4, T5,T6** then we get the accurate counter value=5.

Now if these statements are executed concurrently by interleaving as follows:

T0: R1 = counter	{R1 = 5, counter=5}
T1: R1 = R1 + 1	{R1 = 6, counter=5}
T2: R2 = counter	{R2 = 5, counter=5}
T3: R2 = R2 − 1	{R2 = 4, counter=5}
T4: counter = R1	{counter = 6, R1=6}
T5: counter = R2	{counter = 4, R2=4}

Note that we have arrived at the incorrect state –counter == 4‖, indicating that four buffer locations are full but actually five buffer locations are full.

If we reversed the order of the statements at T4 and T5, we would arrive at the incorrect state –counter == 6‖.

Race condition

- Several processes access and manipulate the same data concurrently and the outcome of the execution depends on the particular order in which the access takes place is called a Race Condition.
- To guard against the race condition, we need to ensure that only one process at a time can be manipulating the variable counter. To make such a guarantee, we require that the processes be synchronized.

THE CRITICAL-SECTION PROBLEM

Consider a system consisting of *n* processes {*P*0, *P*1, ..., *Pn*−1}.

- Each process has a segment of code, called a **Critical Section**, in which the process may be changing common variables, updating a table, writing a file and so on.
- When one process is executing in its critical section, no other process is allowed to execute in its critical section.

```
do {
        Entry section

          Critical Section

        Exit section

          Remainder section
   } while (true);
```

- Each process must request permission to enter its critical section. The section of code implementing entering request is the **Entry section**.
- The critical section may be followed by an **Exit section**.
- The remaining code is the **Remainder section**.
- The entry section and exit section are enclosed in boxes to highlight these important segments of code.

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

1. **Mutual exclusion**. If process *Pi* is executing in its critical section, then no other processes can be executing in their critical sections.
2. **Progress**. If no process is executing in its critical section and some processes wish to enter their critical sections, then only those processes that are not executing in their remainder sections can participate in deciding which will enter its critical section next and this selection cannot be postponed indefinitely.
3. **Bounded waiting**. After a process has made a request to enter its critical section and before that request is granted, there exists a limit on the number of times that other processes are allowed to enter their critical sections..

Two general approaches are used to handle critical sections in operating systems:

1. **Preemptive Kernel** allows a process to be preempted while it is running in kernel mode.
2. **Non-preemptive Kernel** does not allow a process running in kernel mode to be preempted.

PETERSON’S SOLUTION

Peterson’s solution is a classic **Software-Based Solution** to the critical-section problem. Peterson’s solution is restricted to two processes that alternate execution between their critical sections and remainder sections.

The processes are numbered ***P0*** and ***P1***. Let ***Pi*** represents one process and ***Pj*** represents other processes (i.e. j = i-1)

```
do
{
    flag[i] = true;
    turn = j;
    while (flag[j] && turn == j);

        Critical Section

    flag[i] = false;

    Remainder Section
} while (true);
```

Peterson’s solution requires the two processes to share two data items:

```
int  turn;
boolean flag[2];
```

The variable turn indicates whose turn it is to enter its critical section. At any point of time the turn value will be either 0 or 1 but not both.

- if turn == i, then process *Pi* is allowed to execute in its critical section.
- if turn == j, then process *Pj* is allowed to execute in its critical section.
- The flag array is used to indicate if a process is ready to enter its critical section.

Example: if flag[i] is true, this value indicates that *Pi* is ready to enter its critical section.

- To enter the critical section, process *Pi* first sets **flag[i]=true** and then sets **turn=j**, thereby **Pi** checks if the other process wishes to enter the critical section, it can do so.
- If both processes try to enter at the same time, turn will be set to both i and j at the same time. Only one of these assignments will be taken. The other will occur but will be overwritten immediately.
- The eventual value of turn determines which of the two processes is allowed to enter its critical section first.

The above code must satisfy the following requirements:

1. Mutual exclusion
2. The progress
3. The bounded-waiting

Check for Mutual Exclusion

- Each ***Pi*** enters its critical section only if either **flag[j] == false** or **turn == i**.
- If both processes can be executing in their critical sections at the same time, then **flag[0] == flag[1] == true**. But the value of turn can be either **0** or **1** but cannot be both.

- Hence ***P*0** and ***P*1** could not have successfully executed their **while** statements at about the same time.
- If ***Pi*** executed **–turn == j‖** and the process ***Pj*** executed **flag[j]=true** then Pj will have successfully executed the while statement. Now Pj will enter into its **Critical section**.
- At this time, **flag[j] == true** and **turn == j** and this condition will persist as long as ***Pj*** is in its critical section. As a result, mutual exclusion is preserved.

Check for Progress and Bounded-waiting

- The while loop is the only possible way that a process ***Pi*** can be prevented from entering the critical section only if it is stuck in the while loop with the condition **flag[j] == true** and **turn == j**.
- If ***Pj*** is not ready to enter the critical section, then **flag[j] == false** and ***Pi*** can enter its critical section.
- If ***Pj*** has set **flag[j] == true** and is also executing in its while statement, then either **turn == i** or **turn == j**.
- If **turn == i**, then ***Pi*** will enter the critical section. If **turn == j**, then ***Pj*** will enter the critical section.
- Once ***Pj*** exits its critical section, it will reset **flag[j]** to false, allowing ***Pi*** to enter its critical section.
- If ***Pj*** resets **flag[j]** to true, it must also set **turn == i**. Thus, since ***Pi*** does not change the value of the variable **turn** while executing the while statement, ***Pi*** will enter the critical section (**Progress**) after at most one entry by ***Pj*** (**Bounded Waiting**).

Problem with Peterson Solution

There are no guarantees that Peterson's solution will work correctly on modern computer architectures perform basic machine-language instructions such as load and store.

- The critical-section problem could be solved simply in a single-processor environment if we could prevent interrupts from occurring while a shared variable was being modified. This is an Non-preemptive kernel approach.
- We could be sure that the current sequence of instructions would be allowed to execute in order without preemption.
- No other instructions would be run, so no unexpected modifications could be made to the shared variable.

This solution is not as feasible in a multiprocessor environment.

- Disabling interrupts on a multiprocessor can be time consuming, since the message is passed to all the processors.
- This message passing delays entry into each critical section and system efficiency decreases and if the clock is kept updated by interrupts there will be an effect on a system's clock.

SYNCHRONIZATION HARDWARE

Modern computer systems provide special hardware instructions that allow us either to test and modify the content of a word or to swap the contents of two words **atomically (i.e.)** as one uninterruptible unit.

There are two approaches in hardware synchronization:

1. test_and_set function
2. compare_and_swap function

test_and_set function

The test_and_set instruction is executed atomically (i.e.) if two test_and_set( ) instructions are executed simultaneously each on a different CPU, they will be executed sequentially in some arbitrary order.

If the machine supports the test_and_set( ) instruction, then we can implement mutual exclusion by declaring a boolean variable **lock.** The lock is initialized to false.

The definition of test_and_set instruction for process **P_i** is given as:

```
boolean test_and_set(boolean *target)
{
boolean rv = *target;
*target = true;
return rv;
}
```

Below algorithm satisfies all the requirements of Critical Section problem for the process **P_i** that uses two Boolean data structures: waiting[ ], lock.

```
boolean waiting[i] = false;
boolean lock = false;
do
{
        waiting[i] = true;
        key = true;
        while (waiting[i] && key)
        key = test and set(&lock);
        waiting[i] = false;

                /* critical section */

        j = (i + 1) % n;
        while ((j != i) && !waiting[j])
        j = (j + 1) % n;
        if (j == i)
          lock = false;
        else
          waiting[j] = false;

        /* remainder section */
} while (true);
```

Mutual Exclusion

- Process Pi can enter its critical section only if either **waiting[i] == false** or **key == false.**
- The value of key can become false only if the **test_and_set( )** is executed.
- The first process to execute the **test_and_set( )** will find **key == false** and all other processes must wait.

- The variable **waiting[i]** can become false only if another process leaves its critical section. Only one **waiting[i]** is set to **false**, maintaining the mutual-exclusion requirement.

Progress

- A process exiting the critical section either sets **lock==false** or sets **waiting[j]==false**.
- Both allow a process that is waiting to enter its critical section to proceed.
- This requirement ensures progress property.

Bounded Waiting

- When a process leaves its critical section, it scans the array waiting in the cyclic ordering (**i+1, i+2, ..., n−1, 0, ..., i−1**).
- It designates the first process in this ordering that is in the entry section (**waiting[j] == true**) as the next one to enter the critical section.
- Any process waiting to enter its critical section will thus do so within **n−1** turns.
- This requirement ensures the bounded waiting property.

<u>compare_and_swap function</u>

compare_and_swap( ) is also executed atomically. The compare_and_swap( ) instruction operates on three operands. The definition code as given below:

```
int compare_and_swap(int *value, int expected, int new_value)
{
        int temp = *value;
        if (*value == expected)
        *value = new_value;
        return temp;
}
```

Mutual-exclusion implementation with the compare and swap( ) instruction:

```
do
{
   while (compare_and_swap(&lock, 0, 1) != 0);  /* do nothing */
   /* critical section */
      lock = 0;
  /* remainder section */
} while (true);
```

The operand value is set to new value only if the expression (*value == exected) is true. Regardless, compare_and_swap( ) always returns the original value of the variable value.

Mutual Exclusion with compare_and_swap( )

- A global variable lock is declared and is initialized to 0 **(i.e. lock=0**).
- The first process that invokes compare_and_swap( ) will set **lock=1**. It will then enter its critical section, because the original value of lock was equal to the expected value of 0.
- Subsequent calls to compare_and_swap( ) will not succeed, because lock now is not equal to the expected value of 0. (**lock==1**).
- When a process exits its critical section, it sets lock back to 0 (**lock ==0**), which allows another process to enter its critical section.

Problem with Hardware Solution:
The hardware-based solutions to the critical-section problem are complicated and they are inaccessible to application programmers.

MUTEX LOCKS
Mutex Locks are short for Mutual Exclusive Locks.

- Mutex locks are software solution to the critical section problem.
- Mutex lock are used to protect critical regions and thus prevent race conditions.
- In Mutex locking, a process must acquire the lock before entering a critical section and it releases the lock when it exits the critical section.
- The acquire( )function acquires the lock and the release( ) function releases the lock.

Critical section solution through Mutex locks:

```
do
{
  acquire( )
  {
     while (!available); /* busy wait */
      available = false;
   }

        Critical Section

  release( )
   {
        available = true;
    }

        remainder section
} while (true);
```

Explanation of Algorithm:

- A mutex lock has a boolean variable **available** whose value indicates if the lock is available or not.
- If the lock is available, a call to acquire( ) succeeds and the lock is then set **unavailable** to other processes.
- A process that attempts to acquire an unavailable lock is blocked until the lock is released. Calls to either acquire( ) or release( ) must be performed atomically.

Disadvantage: Mutex Locks implementation leads to Busy Waiting

- While a process is in its critical section, any other process that tries to enter its critical section must loop continuously in the call to acquire( ).
- This type of mutex lock is also called a **Spinlock** because the process ―spins‖ while waiting for the lock to become available.
- This continual looping is clearly a problem in a real multiprogramming system, where a single CPU is shared among many processes. Busy waiting wastes a lot of CPU cycles.

SEMAPHORES

Semaphores provides solution to the critical section problem.

A **semaphore** S is an integer variable that is accessed only through two standard atomic operations: wait( ) and signal( ).

The definition of wait( ) and signal( ) are as follows:

```
wait(S)
{
        while (S <= 0); // busy wait
        S--;
  }
signal(S)
{
        S++;
  }
```

All modifications to the integer value of the semaphore in the wait( ) and signal( ) operations must be executed all at once.

- When one process modifies the semaphore value, no other process can simultaneously modify that same semaphore value.
- In wait(S) function, the statements ($S \leq 0$) and its possible modification (S--) must be executed without interruption.

Semaphore Usage

Operating system provides two types of semaphores: Binary and Counting Semaphore.

Binary Semaphore

- The value of a **binary semaphore** can range only between 0 and 1.
- Binary semaphores behave similarly to mutex locks.

Counting Semaphore

- The value of a **counting semaphore** can range over an unrestricted domain.
- Counting semaphores can be used to control access to a given resource consisting of a finite number of instances.
- The semaphore is initialized to the number of resources available.
- Each process that wishes to use a resource performs a wait( ) operation on the semaphore and thereby decrementing the count value **(S--)**.
- When a process releases a resource, it performs a signal( ) operation and incrementing the count value **(S++)**.
- When the count for the semaphore goes to 0 **(S<=0)**, all resources are being used.
- After that, processes that wish to use a resource will block until the count becomes greater than 0.

Semaphores provides solution for Synchronization problem

Consider two concurrently running processes: $P1$ with a statement $S1$ and $P2$ with a statement $S2$. Suppose we require that $S2$ be executed only after $S1$ has completed.

We can implement this scheme by letting $P1$ and $P2$ share a common semaphore synch, initialized to 0 **(i.e. synch==0).**

In process P1, we insert the statements.

```
S1;
signal(synch);
```

In process P2, we insert the statements

```
wait(synch);
S2 ;
```

Because synch is initialized to 0 **(i.e. synch==0)**, P2 will execute S2 only after P1 has invoked signal(synch), which is after statement S1 has been executed.

Semaphore Implementation

Definitions of the wait( ) and signal( ) semaphore operations leads to busy waiting problem.
To overcome the problem of busy waiting, the definitions of wait( ) and signal( ) must have to modified as follows:

- When a process executes wait( ) operation and finds that the semaphore value is not positive, it must wait. (i.e.) Rather than engaging in busy waiting, the process can block itself.
- The block operation places a process into a waiting queue associated with the semaphore and the state of the process is switched to the waiting state.
- Now control is transferred to **CPU scheduler**, which selects another process to execute.
- A process that is blocked, waiting on a semaphore S, should be restarted when some other process executes a signal( ) operation.
- The process is restarted by a wakeup( ) operation, which changes the process from the waiting state to the ready state. The process is then placed in the ready queue.

The semaphore definition has been changed to single integer variable to a Structure type with two variables (i.e) Each semaphore has an integer value and list of processes.
When a process must wait on a semaphore, it is added to the list of processes. A signal( ) operation removes one process from the list of waiting processes and awakens that process.

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

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

- It is critical that semaphore operations wait( ) and signal( ) be executed atomically. This ensures **mutual exclusion** property of Critical Section problem.
- The block( ) operation suspends the process that invokes it.
- The wakeup(P) operation resumes the execution of a blocked process P.
- The list of waiting processes can be easily implemented by a link field in each process control block (PCB).
- Each semaphore contains an integer value and a pointer to a list of PCBs.
- A FIFO queue is used to add and remove a process from the list. It ensures the Bounded Waiting property.
- The semaphore contains both head and tail pointers to the FIFO queue.

Note: The above algorithm does not eliminate the **busy waiting** problem completely instead it limits the **busy_waiting** problem to very short amount of time.

Deadlocks and Starvation

A set of processes is in a deadlocked state when every process in the set is waiting for an event that can be caused only by another process in the set. The events may be either resource acquisition or resource release.

Note: 1. Deadlocks leads to a problem of indefinite blocking and starvation.
2. The implementation of a semaphore with a waiting queue may result in Deadlock.

Example: Consider a system consisting of two processes P0 and P1. Both are accessing two semaphores S and Q. and **S==Q==1**.

```
P0             P1
wait(S);       wait(Q);
wait(Q);        wait(S);
.                     .
.                     .
.                     .
signal(S);     signal(Q);
signal(Q);     signal(S);
```

- Suppose that *P*0 executes wait(S) and then *P*1 executes wait(Q).
- When *P*0 executes wait(Q), it must wait until *P*1 executes signal(Q).
- Similarly, when *P*1 executes wait(S), it must wait until *P*0 executes signal(S).
- Since these signal( ) operations cannot be executed, *P*0 and *P*1 are deadlocked.

Priority Inversion

Priority inversion is a scheduling problem. It occurs only in systems with more than two priorities.

Consider there are three processes L, M, H whose order of priorities are given as **L< M < H** and all are wanted to access the resource R.

- When a higher-priority process (H) needs to read or modify kernel data resource R that are currently being accessed by a lower-priority process (L).
- Since kernel data are protected with a lock, the higher-priority process (H) will have to wait for a lower-priority (L) to finish with the resource.
- The situation becomes more complicated if the lower-priority process (L) is preempted in favor of another process (M) with a higher priority.
- Now the higher priority process (H) must have to wait until the process M has to release the lock and then L has to release the lock then only the process H can access the kernel data resource **R**.
- This problem is called **Priority Inversion**.

One solution for priority inversion is that the system will have only two priorities.

- These systems solve the priority inversion problem by implementing a **Priority Inheritance Protocol**.
- In this protocol, all processes that are accessing resources needed by a higher-priority process inherit the higher priority until they are finished with the resources in question.
- When they are finished, their priorities revert to their original values.

Example:

- A priority-inheritance protocol would allow process **L** to temporarily inherit the priority of process H, thereby preventing process M from preempting process **L's** execution.
- When process L had finished using resource R, it would relinquish its inherited priority from H and assume its original priority.
- Because resource R would now be available, process H will get the resource instead of process M.

Note: This solution is inefficient for systems with more than one priority given for processes.

CLASSIC PROBLEMS OF SYNCHRONIZATION

There are three classic problem that are related synchronization when processes are executing concurrently.

1. The Bounded-Buffer Problem
2. The Readers–Writers Problem
3. The Dining-Philosophers Problem

The Bounded-Buffer Problem

The major problem in Producer-consumer process is The Bounded-Buffer problem.

Let the Buffer pool consists of **n** locations, each location stores one item.

The Solution for Producer-Consumer Process can be given as:

```
int n;
semaphore empty = n; semaphore full = 0
semaphore mutex = 1;
```

```
Producer process:   do {
                        /* produce an item in next produced */
                        wait(empty);   /* If any one location on buffer is empty */
                        wait(mutex);

                                /* add next produced item to the buffer */

                        signal(mutex);
                        signal(full);
                    } while (true);
Consumer process:   do {
                        wait(full);        /* If any one location in buffer is full */
                        wait(mutex);

                                /* remove an item from buffer to next consumed */

                        signal(mutex);
                        signal(empty);

                                /* consume the item in next consumed */

                     } while (true);
```

Explanation of above algorithm

There is one integer variable and three semaphore variables are declared in the definition.

- Integer variable n represents the size of the buffer.
- The mutex semaphore provides mutual exclusion for accesses to the buffer pool and is initialized to the value 1. Mutex variable allows only either producer process or consumer process allows to access the buffer at a time.
- The **empty** semaphore counts the no. of free location in buffer, it is initialized to **n**.
- The **full** semaphore counts the number of full locations buffers, is initialized to **0**.
- **empty=n** and **full=0** represents that initially all locations in the buffer are empty.

Producer Process

- **wait(empty):** Any location in the buffer is free, then wait(empty) operation is successful and producer process will put an item into buffer. If wait(empty) is false then the producer process will be blocked.
- **wait(mutex):** It allows the producer to use the buffer and produce an item into buffer.
- **signal(mutex):** Buffer will be released by producer process.
- **signal(full):** It indicates one or more item added to the buffer if signal(full) is successful.

Consumer process

- **wait(full):** If wait(full) < **0** then the buffer is empty and there are no item in the buffer to consume. Hence the consumer process will be blocked. Otherwise the buffer is having some items the consumer process can consume an item.
- **wait(mutex):** It allows consumer to use buffer.
- **signal(mutex):** Buffer is released by consumer.
- **signal(empty):** Consumer consumed an item. Hence one more location in buffer is free.

The Readers–Writers Problem

The Reader-Writer problem occurs in the following situation:

- Consider there is a shared file that is shared by two processes called Reader process and Writer process.
- The reader process will read the data from the file and the writer process will modifies the contents of the file.
- A reader process does not modify the shared file contents. Hence if two reader processes access the shared data simultaneously then there will be no problem of inconsistency.
- If a writer process is writing the data and any other process wants to read or write the shared file simultaneously that may result the inconsistency of data. Hence we can't allow more than one writer process to access the shared file.
- If one writer process is modifying the contents of the shared file, no other reader or writer process will read or write the data of shared file.
- Hence the writer process has exclusive access to the shared file while performing write operation.
- This synchronization problem is referred to as the **Readers–Writers problem**.

Solution to Reader-Writer Problem

```
semaphore rw_mutex = 1;
semaphore mutex = 1;
int read_count = 0;
```

Code for Reading the data:

```
  do
  {
  wait(mutex);
  read_count++;
  if (read_count == 1)
  wait(rw_mutex);
  signal(mutex);
  . . .
  /* reading is performed */
  . . .
  wait(mutex);
  read_count--;
  if (read_count == 0)
  signal(rw_mutex);
  signal(mutex);
} while (true);
```

Code for Writing data:

```
do
 {
    wait(rw_mutex);
     . . .
```

```
            /* writing is performed */
             . . .
            signal(rw_mutex);
           } while (true);
```

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

Reader-Writer process code Explanation

- Let us consider the process P1 that is executing **wait(mutex)** operation that increments the **read_count** to **1** (i.e. now **read_count=1**).
- The **if condition** has been satisfied by P1 and it can invoke **wait(rw_mutex)** and P1 will enter into the critical section and modify (write) the contents of the shared file.
- Later P1 executes **signal(rw_mutex)** operation and the control pointer returns to readering process and executes **signal(mutex)**.
- The **signal(mutex)** operation allows another process P2 to enter into reader process and executes **wait(mutex)** and increments **read_count** value by **1** (i.e. now **read_count=2**).
- Now P2 fails to satify the **if condition** because the **read_count** value.
- Hence P2 can only reads the file but not writes on the file because P1 is still in critical section and reading the updated data.
- If P1 coming out of the critical section and executes **wait(mutex)** and decrements the **read_count** value by **1**. (now **read_count =1**) and executes **signal(mutex)** to allow P2.
- If P2 now executes **if condition**, then the condition is satisfied and P2 will enter into writing process and writes the contents of the file.
- This is how the semaphore variable solves the synchronization problem of critical section.

The Dining-Philosophers Problem

Consider five philosophers who spend their lives thinking and eating.

- The philosophers share a circular table surrounded by five chairs, each belonging to one philosopher.
- In the center of the table is a bowl of rice and the table is laid with five single chopsticks.
- When a philosopher thinks, she does not interact with her colleagues.
- From time to time, a philosopher gets hungry and tries to pick up the two chopsticks that are closest to her. The chopsticks are placed between her and her left and right neighbors.
- A philosopher may pick up only one chopstick at a time. Obviously, she cannot pick up a chopstick that is already in the hand of a neighbor.

- When a hungry philosopher has both her chopsticks at the same time, she eats without releasing the chopsticks.
- When she is finished eating, she puts down both chopsticks and starts thinking again.

Solution with semaphores:

```
semaphore chopstick[5];
do {
        wait(chopstick[i]);
        wait(chopstick[(i+1) % 5]);

        /* eat for a while */

        signal(chopstick[i]);
        signal(chopstick[(i+1) % 5]);

        /* think for a while */

} while (true);
```

- Each chopstick is represented with a semaphore and all the elements of chopstick are initialized to 1.
- A philosopher tries to grab a chopstick by executing a wait( ) operation on that semaphore.
- A philosopher releases chopsticks by executing the signal( ) operation on the appropriate semaphores.

Note: Although this solution guarantees that no two neighbors are eating simultaneously, it nevertheless must be rejected because it could create a deadlock.

1. Suppose that all five philosophers become hungry at the same time and each grabs her left chopstick.
2. All the elements of chopstick will now be equal to 0.
3. When each philosopher tries to grab her right chopstick, she will be delayed forever.

Possible remedies to the Deadlock problem:

- Allow at most four philosophers to be sitting simultaneously at the table.
- Allow a philosopher to pick up her chopsticks only if both chopsticks are available. To do this, she must pick them up in a critical section.
- An Asymmetric solution will be used, an odd-numbered philosopher picks up first her left chopstick and then her right chopstick, whereas an even numbered philosopher picks up her right chopstick and then her left chopstick.

PROBLEM WITH SEMAPHORES

Semaphores provide a convenient and effective mechanism for process synchronization but using semaphores incorrectly can result in timing errors that are difficult to detect.

These errors happen only if particular execution sequences take place and these sequences do not always occur.

Examples: 1

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

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

In this situation, several processes may be executing in their critical sections simultaneously, violating the mutual-exclusion requirement. This error may be discovered only if several processes are simultaneously active in their critical sections.

Example:2

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

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

In this case, a deadlock will occur.

Example:3

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

MONITORS

Monitors are developed to deal with semaphore errors. Monitors are high-level language synchronization constructs.

- A ***monitor type*** is an ADT that includes a set of programmer defined operations that are provided with mutual exclusion within the monitor.
- The monitor type also declares the variables whose values define the state of an instance of that type, along with the bodies of functions that operate on those variables.

Monitor Syntax:

```
monitor monitor_ name
{
  /* shared variable declarations */
        function P1 ( . . . )
        {
        . . .
        }
        function P2 ( . . . )
        {
        . . .
        }
        function Pn ( . . . )
        {
          . . .
          }
```

```
                    initialization code ( . . . )
                    {
                    . . .
                    }
                }
```

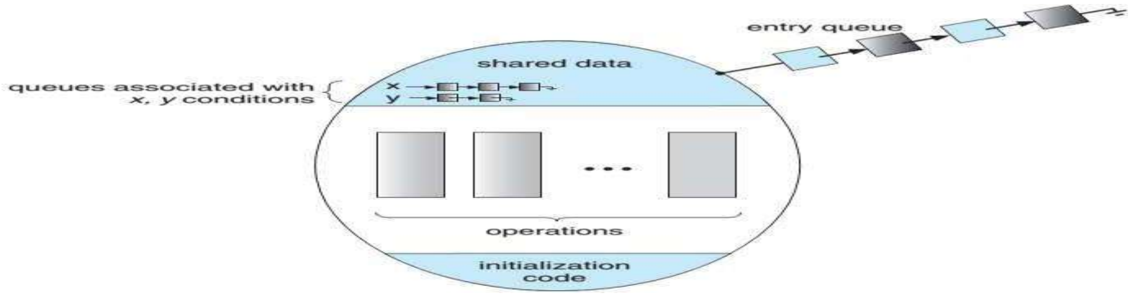
The representation of a monitor type cannot be used directly by the various processes.

- A function defined within a monitor can access only those variables declared locally within the monitor and its formal parameters.
- Similarly, the local variables of a monitor can be accessed by only the local functions.
- The monitor construct ensures that only one process at a time is active within the monitor.
- the programmer does not need to code this synchronization constraint explicitly

To solve the problem of synchronization problem along with monitors a construct called **Condition** has been implemented. Condition construct defines one or more variables of type **"condition".** The syntax of condition is:

condition x,y;

The only operations that can be invoked on a condition variable are wait( ) and signal( ).
x.wait( ): Process invokes x.wait( ) is suspended until another process invokes **x.signal( )**.
x.signal( ): It resumes exactly one suspended process.



Dining-Philosophers Solution Using Monitors

By using monitor we can have a deadlock free solution for dining philosopher's problem.
This solution imposes the restriction that a philosopher may pick up her chopsticks only if both of them are available.
There are 3 states for each philosopher: Thinking, Hungry, Eating.

enum { THINKING, HUNGRY, EATING } state[5];

Philosopher *i* can set the variable state[i] = EATING only if her two neighbors are not eating:

(state[(i+4) % 5] != EATING) and (state[(i+1) % 5] != EATING)

We also need to declare: **condition self[5];**
This allows philosopher *i* to delay herself when she is hungry but is unable to obtain the chopsticks she needs.

Solution through monitor:

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

The distribution of the chopsticks is controlled by the monitor **Dining_Philosophers**.

- Each philosopher before starting to eat, must invoke the operation pickup( ). This act may result in the suspension of the philosopher process.
- After the successful completion of the pickup( ) operation, the philosopher may eat.
- After pickup( ) operation the philosopher invokes the putdown( ) operation.

Thus, philosopher i must invoke pickup( ) & putdown( ) operations in following sequence:

```
DiningPhilosophers.pickup(i);
            Eat
DiningPhilosophers.putdown(i);
```

This solution ensures that no two neighbors are eating simultaneously and that no deadlocks will occur.

Implementing a Monitor Using Semaphores

Monitor uses three variables: semaphore mutex=1; semaphore next=0; int next_count;

- For each monitor, a semaphore mutex is provided. A process must execute wait(mutex) before entering the monitor and must execute signal(mutex) after leaving the monitor.
- The signaling processes can use **next** variable to suspend themselves, because signaling process must wait until the resumed process exit or leave.
- **next count** is an integer variable used to count the number of processes suspended on next.

Each external function F is replaced by:

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

The above code ensures the mutual exclusion property.

Implementing condition variables

For each condition x, we introduce a semaphore x_sem and an integer variable x_count.

```
semaphore x_sem=0;
int x_count=0;
```

The operation x.wait( ) can now be implemented as

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

The operation x.signal( ) can be implemented as

```
if (x_count > 0)
{
        next_count++;
        signal(x_sem);
        wait(next);
        next_count--;
}
```

Resuming Processes within a Monitor

Consider a situation, if several processes are suspended on condition x, and an x.signal( ) operation is executed by some process, then how do we determine which of the suspended processes should be resumed next?

We have two solutions: FCFS and Priority mechanism.

We use first-come, first-served (FCFS) ordering, so that the process that has been waiting the longest is resumed first.
In priority mechanism the conditional-wait construct can be used.

x.wait(c);

where c is an integer priority numbers that is evaluated when the wait( ) operation is executed. The **c** value is then stored with the name of the process that is suspended.

When x.signal( ) is executed, the process with the smallest priority number is resumed next.

Consider the below code the **Resource_Allocator monitor** that controls the allocation of a single resource among competing processes.

```
monitor Resource_Allocator
{
        boolean busy;
        condition x;
        void acquire(int time)
        {
                if (busy)
                x.wait(time);
                busy = true;
        }
        void release( )
        {
                busy = false;
                x.signal( );
        }
        initialization_code( )
        {
            busy = false;
        }
}
```

- Each process, when requesting an allocation of this resource, specifies the maximum time it plans to use the resource.
- Monitor allocates the resource to the process that has the shortest time-allocation request.

The process that needs to access the resource must observe the following sequence:

```
R.acquire(t);  /* R is an instance of type ResourceAllocator */
        access the resource;
R.release( );
```

Problems with monitor

1. A process might access a resource without first gaining access permission to the resource.
2. A process might never release a resource once it has been granted access to the resource.
3. A process might attempt to release a resource that it never requested.
4. A process might request the same resource twice (without first releasing the resource).

SYNCHRONIZATION EXAMPLES

Synchronization in Windows

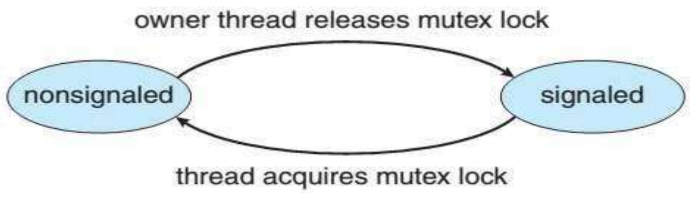
The Windows operating system is a multithreaded kernel that provides support for real-time applications and multiple processors.

- On a single-processor system, when the Windows kernel accesses a global resource, it temporarily masks interrupts for all interrupt handlers that may also access the global resource.
- On a multiprocessor system, Windows protects access to global resources using spinlocks.
- For reasons of efficiency, the kernel ensures that a thread will never be preempted while holding a spinlock.

For thread synchronization outside the kernel, Windows provides **Dispatcher objects**.

- Using a dispatcher object, threads synchronize according to several different mechanisms, including mutex locks, semaphores, events, and timers.
- The system protects shared data by requiring a thread to gain ownership of a mutex to access the data and to release ownership when it is finished.
- **Events** are similar to condition variables (i.e.) they may notify a waiting thread when a desired condition occurs.
- Finally, timers are used to notify one or more threads that a specified amount of time has expired.

Dispatcher objects may be in either a **signaled state** or a **nonsignaled state**.



- An object in a **signaled state** is available and a thread will not block when acquiring the object.
- An object in a **non-signaled state** is not available and a thread will block when attempting to acquire the object.

Relationship between State of a Dispatcher object and State of a Thread:

- When a thread blocks on a nonsignaled dispatcher object, its state changes from ready to waiting state and the thread is placed in a waiting queue for that object.
- When the state for the dispatcher object moves to signaled, the kernel checks whether any threads are waiting on the object.
- If there are waiting threads, the kernel moves one or more threads from the waiting state to the ready state and they can resume executing.
- The number of threads the kernel selects from the waiting queue depends on the type of dispatcher object for which it is waiting.
- The kernel will select only one thread from the waiting queue for a mutex, since a mutex object may be ―owned‖ by only a single thread.
- For an event object, the kernel will select all threads that are waiting for the event.

Mutex Lock on Dispatcher Objects and Thread States:

- If a thread tries to acquire a mutex dispatcher object that is in a nonsignaled state, that thread will be suspended and placed in a waiting queue for the mutex object.
- When the mutex moves to the signaled state, the thread waiting at the front of the queue will be moved from the waiting state to the ready state and will acquire the mutex lock.

Critical Section Object

- A critical-section object is a user-mode mutex that can often be acquired and released without kernel intervention.
- On a multiprocessor system, a critical-section object first uses a spinlock while waiting for the other thread to release the object.
- If it spins too long, the acquiring thread will then allocate a kernel mutex and yield its CPU.
- Critical-section objects are particularly efficient because the kernel mutex is allocated only when there is contention for the object.

Synchronization in Linux

From Linux Version 2.6, it is using Preemptive Kernel. Prior to Version 2.6, Linux was a non-preemptive kernel.

- Linux provides several different mechanisms for synchronization in the kernel.
- Linux uses the Synchronization technique within the kernel is an atomic integer, which is represented using the opaque data type atomic_t.
- All math operations using atomic integers are performed without interruption.

The following code illustrates declaring an atomic integer counter and then performing various atomic operations:

```
atomic_t counter;
int value;
atomic_set(&counter,5);        /* counter = 5 */
atomic_add(10, &counter);      /* counter = counter + 10 */
atomic_sub(4, &counter);       /* counter = counter - 4 */
atomic_inc(&counter);          /* counter = counter + 1 */
value = atomic_read(&counter); /* value = 12 */
```

Atomic integers are particularly efficient in situations where an integer variable such as a counter needs to be updated, since atomic operations do not require the overhead of locking mechanisms.

Mutex Locks in Linux

Mutex locks are available in Linux for protecting critical sections within the kernel.

- A task must invoke the mutex_lock( ) function prior to entering a critical section and the mutex_unlock( ) function after exiting the critical section.
- If the mutex lock is unavailable, a task calling mutex_lock( ) is put into a sleep state and is awakened when the lock's owner invokes mutex_unlock( ).

Semaphores in Linux

- Linux provides spinlocks and semaphores for locking in the kernel.
- On SMP machines, the fundamental locking mechanism is a spinlock, and the kernel is designed so that the spinlock is held only for short durations.
- on single-processor systems, rather than holding a spinlock, the kernel disables kernel preemption and rather than releasing the spinlock, it enables kernel preemption.
- Linux provides two system calls: **preempt_disable( )** used to disabling Kernel pre-emption and **preempt_enable( )** used to enabling kernel preemption.

Process of preemption

- Each task in the system has a thread-info structure containing a counter called **preempt_count,** to indicate the number of locks being held by the task.
- When a lock is acquired, **preempt_count** is incremented. It is decremented when a lock is released.
- If the value of preempt_count for the task currently running in the kernel is greater than 0, it is not safe to preempt the kernel because this task currently holds a lock.
- If the count is 0, the kernel can safely be interrupted, there are no outstanding calls to preempt_disable( ).

Note:

1. Spinlocks along with enabling and disabling kernel preemption are used in the kernel only when a lock is held for a short duration.
2. When a lock must be held for a longer period, semaphores or mutex locks are appropriate for use.

UNIT III
INTER PROCESS COMMUNICATION (IPC) MECHANISMS

Inter process communication (IPC) is a process that allows different processes of a computer system to share information. IPC lets different programs run in parallel, share data, and communicate with each other. It's important for two reasons: First, it speeds up the execution of tasks, and secondly, it ensures that the tasks run correctly and in the order that they were executed.

interprocess communication is necessary

IPC lets different programs run in parallel, share data, and communicate with each other.

- It speeds up the execution of tasks.
- It ensures that the tasks run correctly and in the order that they were executed.
- IPC is essential for the efficient operation of an operating system.
- Operating systems use IPC to exchange data with tools and components that the system uses to interact with the user, such as the keyboard, the mouse, and the graphical user interface (GUI).
- IPC also lets the system run multiple programs at the same time. For example, the system might use IPC to provide information to the windowing system about the status of a window on the screen.

How does IPC work in Computer Systems?

- IPC occurs when an application sends a message to an operating system process. The operating system sends the message to a designated IPC mechanism, which handles the message and sends a response back to the application. IPC mechanisms can be found in the kernel or the user space of an operating system.
- IPC is an essential process in the operation of computer systems. It enables different programs to run in parallel, share data, and communicate with each other. IPC is important for the efficient operation of an operating system and ensures that the tasks run correctly and in the order that they were executed.

A system can have two types of processes

- **Independent Process**

1. There may be several processes running in the system at the same time which can be either cooperating processes or independent processes.
2. An independent process cannot be impacted or affected by other processes.
3. Cooperating Process in OS is a process that can affect or get affected by any other process under execution.

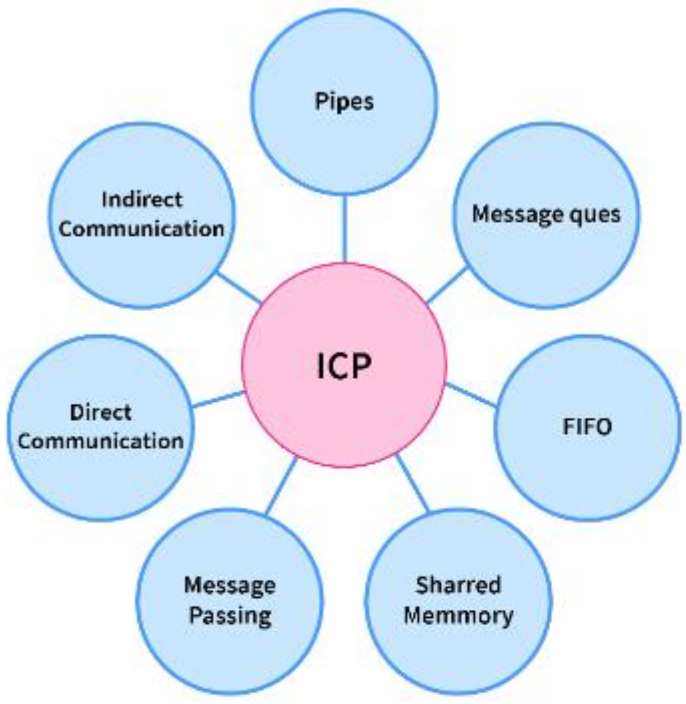
- **Cooperating Process**

Cooperating processes affect each other and may share data and information among themselves. Interprocess Communication or IPC provides a mechanism to exchange data and information across multiple processes, which might be on single or multiple computers connected by a network.

IPC helps achieve these things:

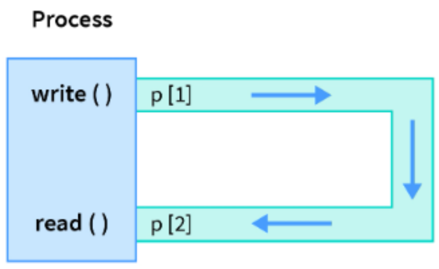
- Computational Speedup
- Modularity
- Information and data sharing
- Privilege separation
- Processes can communicate with each other and synchronize their action.

Different Ways to Implement Inter Process Communication (IPC)



Pipes

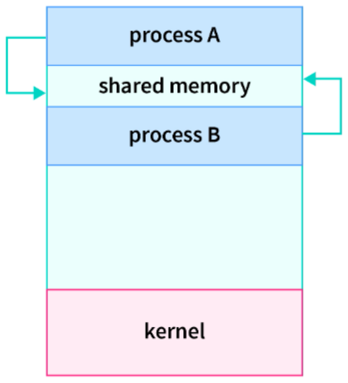
- It is a half-duplex method (or one-way communication) used for IPC between two related processes.
- It is like a scenario like filling the water with a tap into a bucket. The filling process is writing into the pipe and the reading process is retrieved from the pipe.



Shared Memory

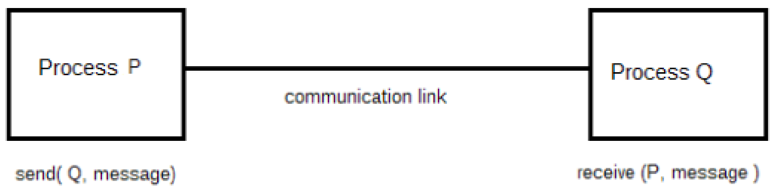
shared memory is a memory shared between all processes by two or more processes established using shared memory. This type of memory should protect each other by synchronizing access between all processes. Both processes, like A and B, can set up a shared memory segment and exchange data through this shared memory area. Shared memory is important for these reasons-

It is a way of passing data between processes.
Shared memory is much faster and more reliable than these methods.
Shared memory allows two or more processes to share the same copy of the data.
Suppose process A wants to communicate with process B and needs to attach its address space to this shared memory segment. Process A will write a message to the shared memory, and Process B will read that message from the shared memory. So, processes are responsible for ensuring synchronization so that both processes do not write to the same location at the same time.



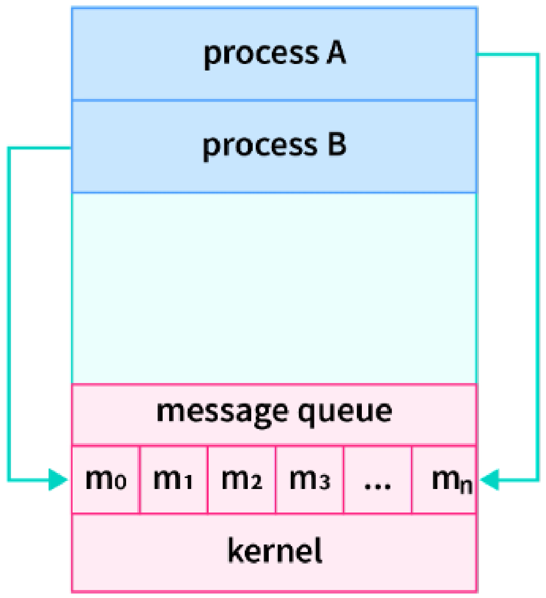
Message Passing

- In IPC, this is used by a process for communication and synchronization.
- Processes can communicate without any shared variables, therefore it can be used in a distributed environment on a network.
- It is slower than the shared memory technique.
- It has two actions sending (fixed size message) and receiving messages.



Message Queues

We have a linked list to store messages in a kernel of OS and a message queue is identified using "message queue identifier".

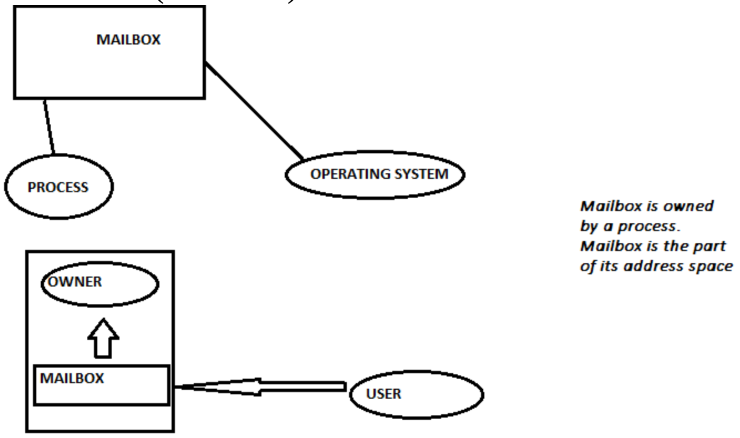


Direct Communication

- In this, processes that wanna communicate must name the sender or receiver.
- A pair of communicating processes must have one link between them.
- A link (generally bi-directional) establishes between every pair of communicating processes.

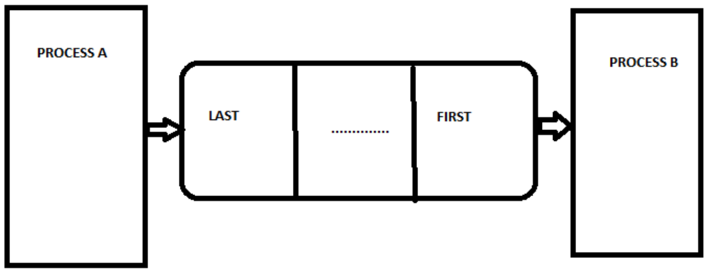
Indirect Communication

- Pairs of communicating processes have shared mailboxes.
- Link (uni-directional or bi-directional) is established between pairs of processes.
- Sender process puts the message in the port or mailbox of a receiver process and the receiver process takes out (or deletes) the data from the mailbox.


MAILBOX
PROCESS
OPERATING SYSTEM
Mailbox is owned by a process. Mailbox is the part of its address space
OWNER
MAILBOX
USER

FIFO

- Used to communicate between two processes that are not related.
- Full-duplex method - Process P1 is able to communicate with Process P2, and vice versa.



Advantages of Inter-Process Communication (IPC)

Inter-Process Communication (IPC) allows different processes running on the same or different systems to communicate with each other. There are several advantages of using IPC, which are:

Data Sharing: IPC allows processes to share data with each other. This can be useful in situations where one process needs to access data that is held by another process.
Resource Sharing: IPC allows processes to share resources such as memory, files, and devices. This can help reduce the amount of memory or disk space that is required by a system.
Synchronization: IPC allows processes to synchronize their activities. For example, one process may need to wait for another process to complete its task before it can continue.
Modularity: IPC allows processes to be designed in a modular way, with each process performing a specific task. This can make it easier to develop and maintain complex systems.
Scalability: IPC allows processes to be distributed across multiple systems, which can help improve performance and scalability.

Disadvantages of Inter-Process Communication (IPC)

Complexity: IPC can add complexity to the design and implementation of software systems, as it requires careful coordination and synchronization between processes. This can lead to increased development time and maintenance costs.
Overhead: IPC can introduce additional overhead, such as the need to serialize and deserialize data, and the need to synchronize access to shared resources. This can impact the performance of the system.
Scalability: IPC can also limit the scalability of a system, as it may be difficult to manage and coordinate large numbers of processes communicating with each other.
Security: IPC can introduce security vulnerabilities, as it creates additional attack surfaces for malicious actors to exploit. For example, a malicious process could attempt to gain unauthorized access to shared resources or data.
Compatibility: IPC can also create compatibility issues between different systems, as different operating systems and programming languages may have different IPC mechanisms and APIs. This can make it difficult to develop cross-platform applications that work seamlessly across different environments.

Examples of Inter Process Communication: Pipes, Shared Memory, Message Queues, Sockets

MAHAVEER INSTITUTE OF SCIENCE AND TECHNOLOGY

(AN UGC AUTONOMOUS INSTITUTION)

Approved by AICTE, Affiliated to JNTUH, Accredited by NAAC with 'A' Grade
Recognized Under Section 2(f) of UGC Act 1956, ISO 9001:2015 Certified
Vyasapuri, Bandlaguda, Post: Keshavgiri, Hyderabad- 500 005, Telangana, India.
https://www.mist.ac.in E-mail:principal.mahaveer@gmail.com, Mobile: 8978380692

ESTD : 2001

Department of Computer Science and Engineering (AIML)

(R22) OPERATING SYSTEM

Lecture Notes

B. Tech II YEAR – I SEM

Prepared by

SANGYAM SOUNDARYA
(Assistant Professor)
Dept.CSE(AIML)

OPERATING SYSTEMS

B.Tech. II Year I Sem.	**L**	**T**	**P**	**C**
	3	**0**	**0**	**3**

Prerequisites:
1. A course on “Computer Programming and Data Structures”.
2. A course on “Computer Organization and Architecture”.

Course Objectives:
- Introduce operating system concepts (i.e., processes, threads, scheduling, synchronization, deadlocks, memory management, file and I/O subsystems and protection)
- Introduce the issues to be considered in the design and development of operating system
- Introduce basic Unix commands, system call interface for process management, interprocess communication and I/O in Unix

Course Outcomes:
- Will be able to control access to a computer and the files that may be shared
- Demonstrate the knowledge of the components of computers and their respective roles in computing.
- Ability to recognize and resolve user problems with standard operating environments.
- Gain practical knowledge of how programming languages, operating systems, and architectures interact and how to use each effectively.

UNIT - I
Operating System - Introduction, Structures - Simple Batch, Multiprogrammed, Time-shared, Personal Computer, Parallel, Distributed Systems, Real-Time Systems, System components, Operating System services, System Calls
Process - Process concepts and scheduling, Operations on processes, Cooperating Processes, Threads

UNIT - II
CPU Scheduling - Scheduling Criteria, Scheduling Algorithms, Multiple -Processor Scheduling. System call interface for process management-fork, exit, wait, waitpid, exec
Deadlocks - System Model, Deadlocks Characterization, Methods for Handling Deadlocks, Deadlock Prevention, Deadlock Avoidance, Deadlock Detection, and Recovery from Deadlock

UNIT - III
Process Management and Synchronization - The Critical Section Problem, Synchronization Hardware, Semaphores, and Classical Problems of Synchronization, Critical Regions, Monitors
Interprocess Communication Mechanisms: IPC between processes on a single computer system, IPC between processes on different systems, using pipes, FIFOs, message queues, shared memory.

UNIT - IV
Memory Management and Virtual Memory - Logical versus Physical Address Space, Swapping, Contiguous Allocation, Paging, Segmentation, Segmentation with Paging, Demand Paging, Page Replacement, Page Replacement Algorithms.

UNIT - V
File System Interface and Operations -Access methods, Directory Structure, Protection, File System Structure, Allocation methods, Free-space Management. Usage of open, create, read, write, close, lseek, stat, ioctl system calls.

TEXT BOOKS:

1. Operating System Principles- Abraham Silberchatz, Peter B. Galvin, Greg Gagne 7th Edition, John Wiley.
2. Advanced programming in the UNIX environment, W.R. Stevens, Pearson education.

REFERENCE BOOKS:

1. Operating Systems- Internals and Design Principles, William Stallings, Fifth Edition–2005, Pearson Education/PHI
2. Operating System A Design Approach- Crowley, TMH.
3. Modern Operating Systems, Andrew S. Tanenbaum 2nd edition, Pearson/PHI
4. UNIX programming environment, Kernighan and Pike, PHI/ Pearson Education
5. UNIX Internals -The New Frontiers, U. Vahalia, Pearson Education.

UNIT-4

UNIT-IV

MEMORY MANAGEMENT

MAIN MEMORY

The main purpose of a computer system is to execute programs.

- During the execution of these programs together with the data they access must be stored in main memory.
- Memory consists of a large array of bytes. Each Byte has its own address.
- CPU fetches instructions from memory according to the value of the program counter.

BASIC HARDWARE

CPU can access data directly only from Main memory and processor registers.

- Main memory and the Processor registers are called **Direct Access Storage Devices**.
- Any instructions in execution and any data being used by the instructions must be in one of these direct-access storage devices.
- If the data are not in memory then the data must be moved to main memory before the CPU can operate on them.
- Registers that are built into the CPU are accessible within one CPU clock cycle.
- Completing a memory access from main memory may take many CPU clock cycles. Memory access from main memory is done through memory bus.
- In such cases, the processor needs to **stall**, since it does not have the required data to complete the instruction that it is executing.
- To avoid **memory stall**, we need to implement Cache memory in between Main memory and CPU.

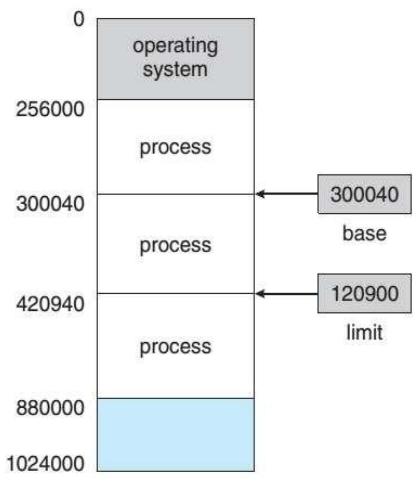
BASE REGISTER & LIMIT REGISTER

Each process has a separate memory space that protects the processes from each other. It is fundamental to having multiple processes loaded in memory for concurrent execution.

There are two register that provides protection: Base register and Limit register

- **Base Register** holds the smallest legal physical memory address.
- **Limit register** specifies the size of the range (i.e. process size).

Example: if the base register holds 300040 and the limit register is 120900, then the program can legally access all addresses from 300040 through 420939 (inclusive).



The base and limit registers can be loaded only by the operating system by using a special privileged instruction that can be executed only in kernel mode.

- Protection of memory space is accomplished by having the CPU hardware compare every address generated in user mode with the registers.
- Any attempt by a program executing in user mode to access operating-system memory or other users' memory results in a trap to the operating system, which treats the attempt as a **Fatal Error**.
- This scheme prevents a user program from either accidentally or deliberately modifying the code or data structures of other users and the operating system.

Operating system executing in kernel mode is given unrestricted access to both operating-system memory and users' memory. This provision allows the operating system to do certain tasks such as:

- Load users' programs into users' memory
- To dump out those programs in case of errors
- To access and modify parameters of system calls
- To perform I/O to and from user memory etc.

Example: A Multiprocessing Operating system must execute context switches, storing the state of one process from the registers into main memory before loading the next process's context from main memory into the registers.

Address Binding

- A program resides on a disk as a binary executable file. The program must be brought into memory and placed within a process for execution.
- 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 are put into the **Input Queue**.

Addresses may be represented in different ways during these steps.

- Addresses in the source program are generally symbolic, such as the variable **count**.
- A compiler typically **binds** these symbolic addresses to **Relocatable addresses** such as "14 bytes from the beginning of this module".
- The **Linkage** editor or **Loader** in turn binds **Relocatable addresses** to **Absolute addresses** such as **74014** (i.e. 74000+14=74014).
- Each binding is a mapping from one address space to another address space.

Binding of instructions and data to memory addresses can be done at any of following steps:

Compile time.

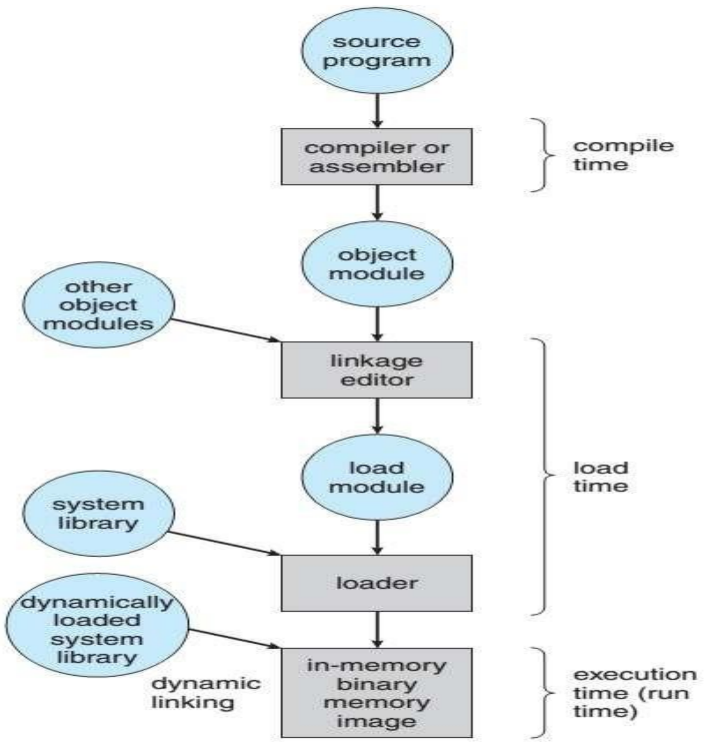
- If you know at compile time where the process will reside in memory then **Absolute code** can be generated.
- Example: If you know that a user process will reside starting at location **R**, then the generated compiler code will start at that location and extend up from there.
- After some time, if the starting location has been changed then it will be necessary to recompile this code.
- The MS-DOS .COM-format programs are bound at compile time.

Load time

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

Execution time

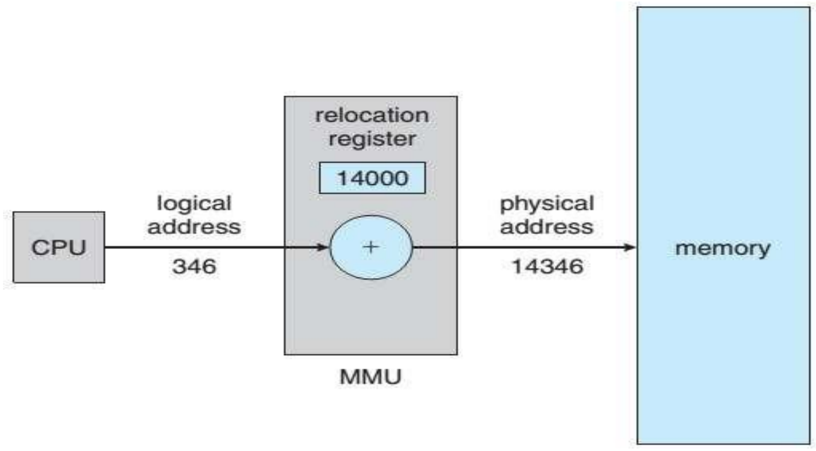
- If the process can be moved during its execution from one memory segment to another, then binding must be delayed until run time.
- Most general-purpose operating systems use this method.



<u>Logical Versus Physical Address Space</u>

- Logical address is the address generated by the CPU.
- Physical address is the address that is loaded into the **Memory-Address Register** of the memory.
- 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** A**ddress Space**.
- The Compile-time and Load-time address-binding methods generate identical logical and physical addresses.
- The execution-time address binding scheme results in different logical and physical addresses. At this time we call logical address as **Virtual address**.
- The run-time mapping from virtual address to physical addresses is done by a hardware device called the **Memory-Management Unit (MMU)**.
- Base register is now called a **Relocation Register**. Value in the relocation register is added to every address generated by a user process at the time the address is sent to memory.

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.



- The user program never sees the real Physical addresses. The program can create a pointer to location 346, store it in memory, manipulate it and compare it with other addresses all as the number 346.
- Only when it is used as a memory address, it is relocated relative to the base register.
- The user program deals with logical addresses. The Memory-mapping hardware converts logical addresses into physical addresses.
- Final location of a referenced memory address is not determined until the reference is made.

Example: Logical addresses in the range **0 to max** and Physical addresses in the range **(R+0)** to (**R + max**) for a base value **R**.

- The user program generates only logical addresses and thinks that the process runs in locations 0 to max.
- These logical addresses must be mapped to physical addresses before they are used.

Dynamic Loading

With dynamic loading, a routine is not loaded until it is called.

- All routines are kept on disk in a relocatable load format. The main program is loaded into memory and it is executed.
- When a routine needs to call another routine, the calling routine first checks to see whether the other routine has been loaded.
- If it has not loaded, the relocatable linking loader is called to load the desired routine into memory and to update the program's address tables to reflect this change.
- Then control is passed to the newly loaded routine.

Advantage: It is useful when large amounts of code are needed to handle infrequently occurring cases, such as error routines. In this case, although the total program size may be large, the portion that is used may be much smaller.

Note: It is the responsibility of the users to design their programs to support Dynamic linking. Operating systems may help the programmer by providing library routines to implement dynamic loading.

Dynamic Linking

Dynamically linked libraries are system libraries that are linked to user programs when the programs are running.

- In static linking system libraries are treated like any other object module and they are combined by the loader into the binary program image.
- In Dynamic linking, the linking is postponed until execution time.
- This feature is usually used with system libraries, such as language subroutine libraries.
- Without dynamic linking, each program on a system must include a copy of its language library in the executable image. This will waste both disk space and main memory.

With dynamic linking, a **stub** is included in the image for each library routine reference.

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

Shared Libraries

- A library may be replaced by a new version and all programs that reference the library will automatically use the new version.
- Without dynamic linking, all such programs would need to be relinked to gain access to the new library.
- So that programs will not accidentally execute new or incompatible versions of libraries.

Version information is included in both the program and the library.

- More than one version of a library may be loaded into memory and each program uses its version information to decide which copy of the library to use.
- Versions with minor changes retain the same version number, whereas versions with major changes increment the number.
- Thus, only programs that are compiled with the new library version are affected by any incompatible changes incorporated in it.
- Other programs linked before the new library was installed will continue using the older library.
- This system is also known as **shared libraries**.

Note: Dynamic linking and shared libraries require help from the operating system.

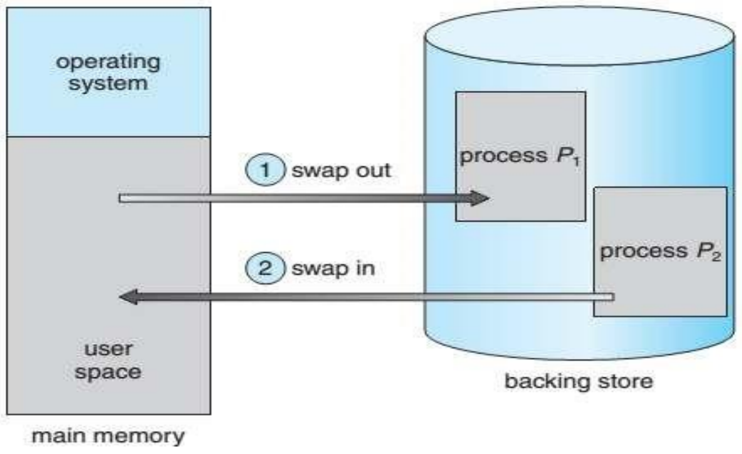
SWAPPING

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

Swapping makes it possible for the total physical address space of all processes to exceed the real physical memory of the system, thus increasing the degree of multiprogramming in a system.

Standard Swapping

- Standard swapping involves moving processes between main memory and a backing store.
- The backing store is commonly a fast disk (i.e. Hard 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.
- The system maintains a **Ready Queue** consisting of all processes whose memory images are on the backing store or in memory and the processes are ready to run.
- Whenever the CPU scheduler decides to execute a process, it calls the dispatcher.
- The dispatcher checks to see whether the next process in the queue is in main memory.
- If it is not in main memory and if there is no free memory region, the dispatcher **swaps out** a process currently in main memory and **swaps in** the desired process. It then reloads registers and transfers control to the selected process.
- The context-switch time in such a swapping system is fairly high. The major part of the swap time is transfer time. The total transfer time is directly proportional to the amount of memory swapped.
- If we want to swap a process, we must be sure that the process is completely idle such as waiting for I/O.



Standard swapping is not used in modern operating systems. It requires too much swapping time and provides too little execution time to be a reasonable memory-management solution.

UNIX, Linux and Windows use modified versions of swapping as below:

- Swapping is enabled only when the amount of free memory falls below a threshold amount.
- Swapping is disabled when the amount of free memory increases.
- Operating system swaps portions of processes rather than the entire process to decrease the swap time.

Note: This type of swapping works in conjunction with Virtualization.

Swapping on Mobile systems
Mobile systems such as iOS and Android do not support swapping.

- Mobile devices generally use flash memory rather than more spacious Hard disks as their persistent storage.
- Mobile operating-system designers avoid swapping because of the less space constraint.
- Flash memory can tolerate only the limited number of writes before it becomes unreliable and the poor throughput between main memory and flash memory in these devices.

Alternative methods used in Mobile systems instead of swapping:

- Apple's iOS asks applications to voluntarily relinquish allocated memory when free memory falls below a certain threshold.
- Read-only data (i.e. code) are removed from the system and later reloaded from flash memory if necessary.
- Data that have been modified such as the **stack** are never removed.
- Any applications that fail to free up sufficient memory may be terminated by the operating system.
- Android may terminate a process if insufficient free memory is available. Before terminating a process android writes its **Application state** to flash memory so that it can be quickly restarted.

CONTIGUOUS MEMORY ALLOCATION

Memory allocation can be done in two ways:

1. Fixed Partition Scheme (Multi-programming with Fixed Number of Tasks)
2. Variable partition scheme (Multi-programming with Variable Number of Tasks)

Fixed Partition Scheme (MFT)

The memory can be divided into several **Fixed-Sized** partitions.

- Each partition may contain exactly one process. Thus, the degree of multiprogramming is bound by the number of partitions.
- In this **Multiple-Partition** 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.

Note: This method was originally used by the IBM OS/360 operating system (called MFT) but is no longer in use.

Variable partition scheme (MVT)

In the variable-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 it is considered one large block of available memory called as **Hole**.
- Eventually the memory contains a set of holes of various sizes.
- 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 CPU time.
- When a process terminates, it releases its memory. The operating system may use this free fill with another process from the input queue.

Memory is allocated to processes until the memory requirements of the next process cannot be satisfied (i.e.) there is no available block of memory is large enough to hold that process.

Then operating system can wait until a large block is available for the process or it can skip the process and moves down to the input queue to see whether the smaller memory requirements of some other process can be met.

- The memory blocks available comprise a **set** of holes of various sizes scattered throughout main 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 and the other part 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.

Dynamic Storage Allocation Problem:
The above procedure leads to Dynamic storage allocation problem which concerns how to satisfy a request of size **n** from a list of free holes.
There are 3-solutions for this problem: First fit, Best fit, worst fit.

- **First fit:** It allocates the first hole that is big enough. Searching can start either at the beginning of the set of holes or at the location where the previous first-fit search ended. We can stop searching as soon as we find a free hole that is large enough.
- **Best fit**. It allocates the smallest hole that is big enough. We must search the entire list, unless the list is ordered by size. This strategy produces the smallest leftover hole.
- **Worst fit**. It allocates the largest hole. Again, we must search the entire list, unless it is sorted by size. This strategy produces the largest leftover hole, which may be more useful than the smaller leftover hole from a best-fit approach.

FRAGMENTATION

There are 2-problems with Memory allocation

1. Internal Fragmentation
2. External Fragmentation

Internal Fragmentation
Consider a multiple-partition allocation scheme with a hole of 18,464 bytes.

- Suppose that the next process requests 18,462 bytes. If we allocate exactly the requested block, we are left with a hole of 2 bytes.
- The overhead to keep track of this hole will be substantially larger than the hole itself.
- The general approach to avoiding this problem is to break the physical memory into fixed-sized blocks and allocate memory in units based on block size.
- With this approach, the memory allocated to a process may be slightly larger than the requested memory.
- The difference between these two numbers is **Internal Fragmentation**. It is unused memory that is internal to a partition.

External Fragmentation

- Both the first-fit and best-fit strategies for memory allocation suffer from **External Fragmentation**.
- As processes are loaded and removed from main memory, the free memory space is broken into small pieces.
- External fragmentation exists when there is enough total memory space to satisfy a request but the available spaces are not contiguous, the storage is fragmented into a large number of small holes.
- External fragmentation problem can be severe. In the worst case, we could have a block of free memory between every two processes that is wasted.
- If all these small pieces of memory were in one big free block instead, we might be able to run several more processes.

Solution to External fragmentation

One solution to the problem of external fragmentation is **Compaction**.

- The goal is to shuffle the memory contents so as to place all free memory together in one large block.
- Compaction is possible only if relocation is dynamic and is done at execution time.
- If addresses are relocated dynamically, relocation requires only moving the program and data and then changing the base register to reflect the new base address.
- If relocation is static and is done at assembly or load time, compaction cannot be done.

Note: Compaction can be expensive, because it moves all processes toward one end of memory. All holes move in the other direction and produces one large hole of available memory.

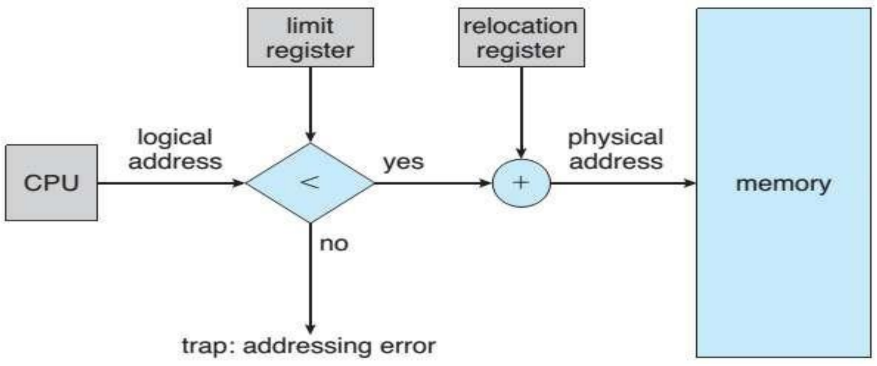
Other solutions to External fragmentation are **Segmentation** and **Paging.** They allow a process to be allocated physical memory wherever such memory is available. These are Non-contiguous memory allocation techniques.

Memory Protection

OS can prevent a process from accessing other process memory. We use two registers for this purpose: Relocation register and Limit register.

- Relocation register contains the value of the smallest physical address such as 100040.
- Limit register contains the range of logical addresses such as 74600.
- Each logical address must be within the range specified by the limit register.

- The relocation-register scheme provides an effective way to allow the operating system's size to change dynamically.
- Memory Management Unit maps the logical address dynamically by adding the value in the relocation register. This mapped address is sent to memory
- When the CPU scheduler selects a process for execution, the dispatcher loads the relocation register and limit registers with the correct values as part of the context switch.
- Because every address generated by a CPU is checked against these registers, we can protect both the operating system and the other users' programs and data from being modified by this running process.

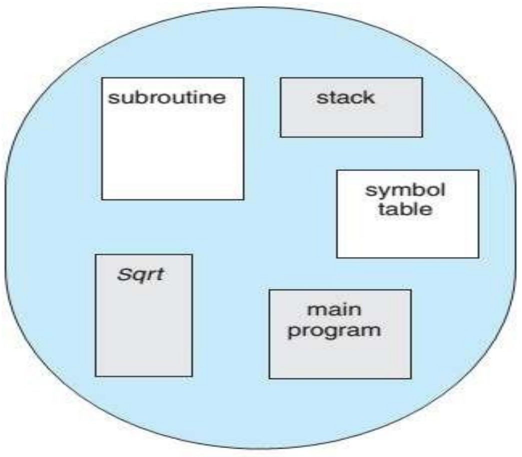


SEGMENTATION

Segmentation is a memory-management scheme that permits the physical address space of a process to be noncontiguous.

A logical address space is a collection of segments. Each segment has a name and a length.

- Logical addresses specify both the segment name and the offset within the segment.
- The programmer specifies each address by two quantities: a segment name and an offset.
- The segments are referred to by Segment Number.
- A logical address consisting of two tuples: **<Segment Number, offset>**



logical address

A C compiler might create separate segments for the following:

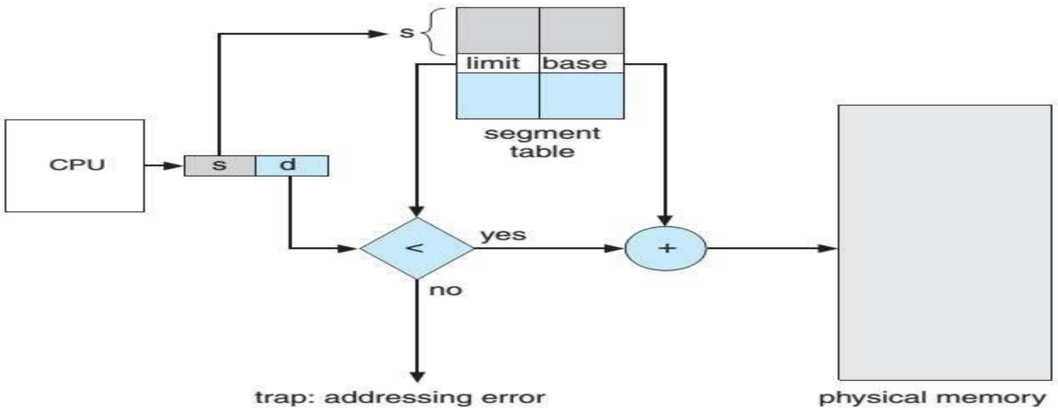
- The code
- Global variables
- The heap, from which memory is allocated
- The stacks used by each thread
- The standard C library

Note: Libraries that are linked in during compile time might be assigned separate segments. The loader would take all these segments and assign them segment numbers.

Segmentation Hardware

Logical address can be viewed by a programmer as a two dimensional address and where as actual Physical address is a one dimensional address.

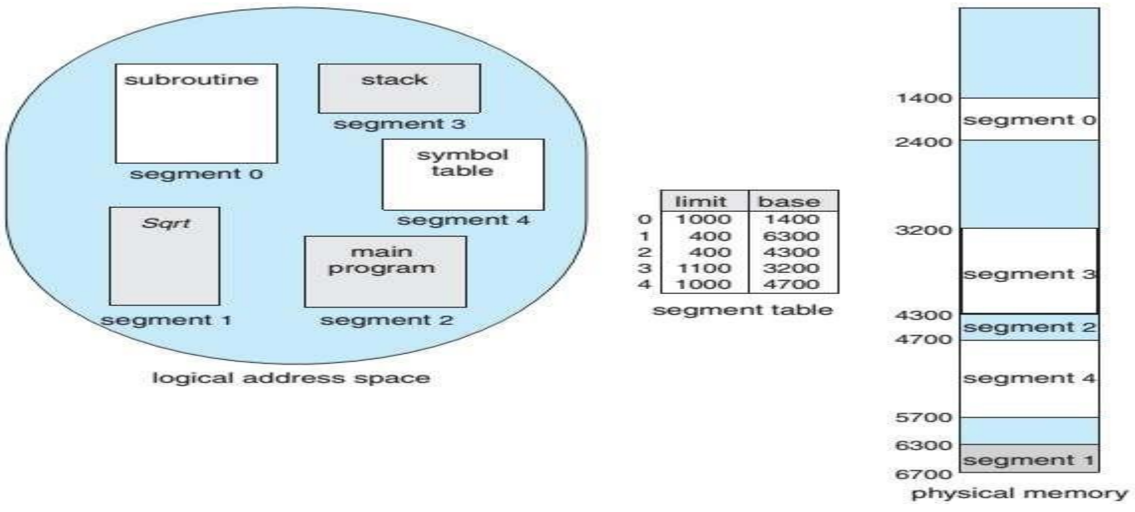
- The Memory Management Unit (MMU) maps two-dimensional user-defined addresses into one-dimensional physical address.
- This mapping is effected by a **Segment table**.
- Each entry in the segment table has a **segment base** and a **segment limit**.
- The segment base contains the starting physical address where the segment resides in memory and the segment limit specifies the length of the segment.



A logical address consists of two parts: segment number **s** and an offset into that segment d.

- The segment number is used as an index to the segment table.
- The offset d of the logical address must be between 0 and the segment limit.
- When an offset is legal, it is added to the segment base to produce the address in physical memory of the desired byte.
- If d>=segment limit, it is illegal then an addressing error trap will be generated that indicates logical addressing attempt beyond end of segment.
- The segment table is essentially an array of base–limit register pairs.

Example: Consider the below diagram that have five segments numbered from 0 to 4. The segments are stored in physical memory.



	limit	base
0	1000	1400
1	400	6300
2	400	4300
3	1100	3200
4	1000	4700

segment table

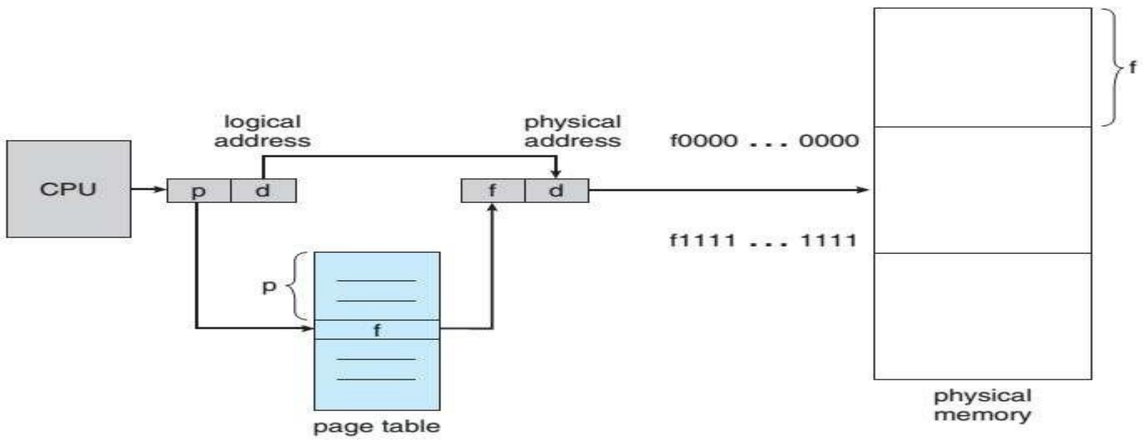
The segment table has a separate entry for each segment, giving the beginning address of the segment in physical memory (i.e. base) and the length of that segment (i.e. limit).

1. 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.
2. A reference to segment 3, byte 852 is mapped to 3200 (base of segment 3) + 852 = 4052.
3. 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.

PAGING

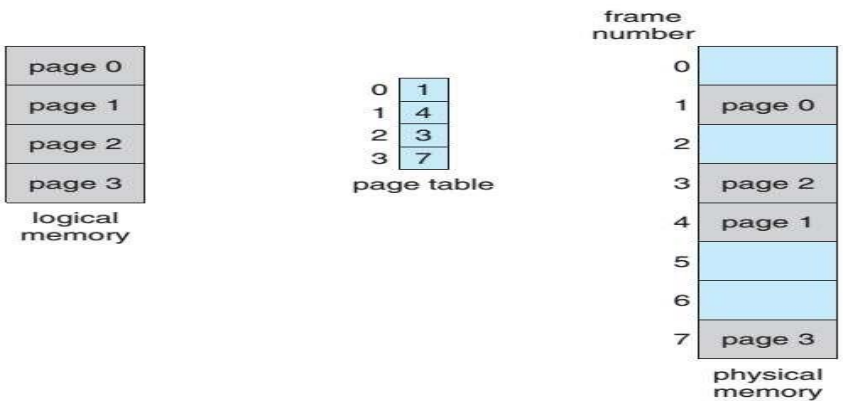
Paging also permits the physical address space of a process to be noncontiguous.
Paging avoids External fragmentation and need for compaction.
Paging is implemented through cooperation between the operating system and the computer hardware.

- Physical memory is divided into fixed-sized blocks called **Frames.**
- Logical memory is divided into blocks of the same size called **Pages**.
- Frame size is equal to the Page size.
- When a process is to be executed, its pages are loaded into any available memory frames from their source such as a file system or backing store.
- The backing store is divided into fixed-sized blocks that are the same size as the memory frames or clusters of multiple frames.
- Frame table maintains list of frame and the allocation details of the frames (i.e.) A frame is free or allocated to some page.
- Each process has its own page table. When a page is loaded into main memory the corresponding page table is active in the system and all other page tables are inactive.
- Page tables and Frame tables are kept in main memory. A **Page-Table Base Register (PTBR)** points to the page table.



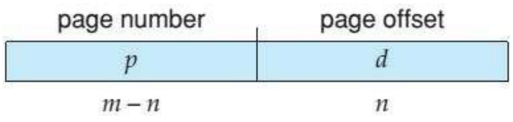
- Every address generated by the CPU is divided into two parts: a **page number (p)** and a **page-offset (d)**.
- The page number is used as an index into a **Page table**. The page table contains the base address of each page in physical memory.
- This base address is combined with the page offset to define the physical memory address that is sent to the memory unit.

The below shows the paging of Logical and Physical memory:

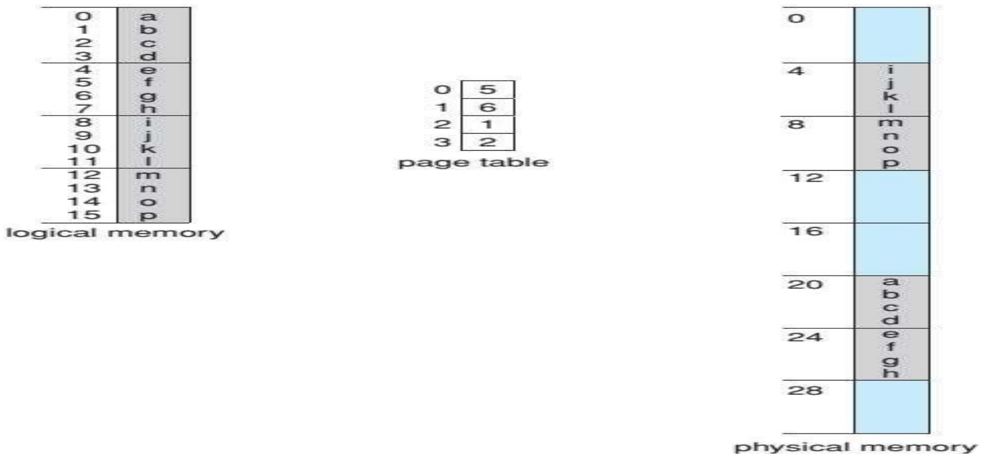


- The page size is defined by the hardware. The size of a page is a power of 2.
- Depending on the computer architecture the page size varies between 512 bytes and 1 GB per page.
- The selection of a power of 2 as a page size makes the translation of a logical address into a page number and page offset particularly easy.
- If the size of the logical address space is 2^m and a page size is 2^n bytes, 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.

The logical address contains: **p** is an index into the page table and **d** is the displacement within the page.



Example: consider the memory in the below figure where *n*= 2 and *m* = 4.
Using a page size of 4 bytes and a physical memory of 32 bytes (8 pages), we show how the programmer's view of memory can be mapped into physical memory.



- Logical address 0 is page 0, offset 0. Indexing into the page table, we find that page 0 is in frame 5. Thus, logical address 0 maps to physical address 20 [= (5 × 4) + 0].
- Logical address 3 (page 0, offset 3) maps to physical address 23 [= (5 × 4) + 3].
- Logical address 4 is page 1, offset 0; according to the page table, page 1 is mapped to frame 6. Thus, logical address 4 maps to physical address 24 [= (6 × 4) + 0].
- Logical address 13 maps to physical address 9.

Paging scheme avoids external fragmentation but it creates internal fragmentation because of fixed size pages.

- If page size is **2048** bytes, a process of **20489** bytes will need **10** pages plus **9** bytes.
- It will be allocated **11** frames, resulting in internal fragmentation of **2048−9 = 2037** bytes.
- In the worst case, a process would need ***n*** pages plus **1** byte. It would be allocated ***n*** **+ 1** frames resulting in internal fragmentation of almost an entire frame.
- If the page size is small then the number of entries in page table is more this will leads to huge number of context switches.
- If the page size is large then the number of entries in page table is less and the number of context switches is less.

Paging separates the programmer's view of memory and the actual physical memory

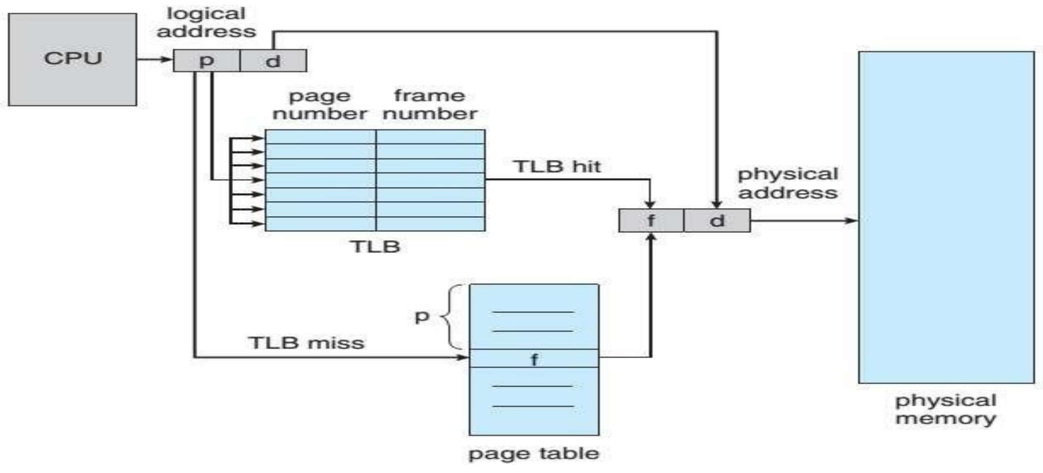
- The programmer views memory as one single space, containing only this one program.
- In fact, the user program is scattered throughout physical memory, which also holds other programs.
- The difference between the programmer's view of memory and the actual physical memory is reconciled by the **Address-Translation Hardware**. The logical addresses are translated into physical addresses.
- This mapping is hidden from the programmer and is controlled by the operating system.
- User process is unable to access memory that it does not own (i.e. other process memory).
- It has no way of addressing memory outside of its page table and the table includes only those pages that the process owns.

Problem: Slow access of a user memory location

- If we want to access location ***i***, we must first index into the page table using the value in the PTBR offset by the page number for ***i***. This task requires a memory access.
- It provides us with the frame number, which is combined with the page offset to produce the actual address. We can then access the desired place in memory.
- With this scheme, ***two*** memory accesses are needed to access a byte (i.e.) one for the page-table entry, one for the byte.
- Thus, memory access is slowed by a factor of 2. This delay is intolerable.

Solution: Translation Look-aside Buffer (TLB)

TLB is a special, small, fast lookup hardware cache. It is associative, high-speed memory.



- Each entry in the TLB consists of two parts: **a key (or tag)** and **a value**.
- The size of TLB is between 32 and 1024 entries.
- When the associative memory is presented with an item, the item is compared with all keys simultaneously.
- If the item is found, the corresponding value field is returned.
- Multiple levels of TLBs are maintained if the system is having multiple levels of Cache.

The TLB contains only a few of the page-table entries.

- When a logical address is generated by the CPU, its page number is presented to the TLB.
- If the page number is found, its frame number is immediately available and is used to access memory. This is called **TLB Hit.**
- These TLB lookup steps are executed as part of the instruction pipeline within the CPU, which does not add any performance penalty compared with a system that does not implement paging.
- If the page number is not in the TLB is known as a **TLB miss**. At the time of TLB miss, a memory reference to the page table must be made.
- Depending on the CPU, this may be done automatically in hardware or via an interrupt to the **OS**. When the frame number is obtained, we can use it to access memory.
- Then we add the page number and frame number to the TLB, so that they will be found quickly on the next reference.
- If the TLB is already full of entries, an existing entry must be selected for replacement by using any of page replacement algorithms.
- **Wired Down entries:** These are the entries that cannot be removed from the TLB. Examples for these are **Key Kernel Code entries.**

Address Space Identifiers (ASID's) in TLB

TLBs store Address-Space Identifiers in each TLB entry.

- An ASID uniquely identifies each process and is used to provide address-space protection for that process.
- When the TLB attempts to resolve virtual page numbers, it ensures that the ASID for the currently running process matches the ASID associated with the virtual page.
- If the ASIDs do not match, the attempt is treated as a TLB miss.
- In addition to providing address-space protection, an ASID allows the TLB to contain entries for several different processes simultaneously.
- If the TLB does not support separate ASIDs, then every time a new page table is selected, the TLB must be **flushed** or erased to ensure that the next executing process does not use the wrong translation information.
- Otherwise, the TLB could include old entries that contain valid virtual addresses but have incorrect or invalid physical addresses left over from the previous process.

TLB Hit Ratio/ Miss Ratio

Percentage of times that the page number is found in the TLB is called the **Hit ratio**. Percentage of times that the page number is not found in the TLB is called the **Miss ratio**.

TLB Miss ratio=1-Hit ratio

Effective Memory Access Time

It is the sum of time taken for a page to access for TLB hit ratio and TLB miss ratio.

Example: An 80-percent hit ratio means that we find the desired page number in the TLB 80 percent of the time. If it takes 100 nanoseconds to access memory, then a mapped-memory access takes 100 nanoseconds when the page number is in the TLB.

If we fail to find the page number in the TLB then we must first access memory for the page table and frame number for 100 nanoseconds and then access the desired byte in memory for 100 nanoseconds with a total of 200 nanoseconds.

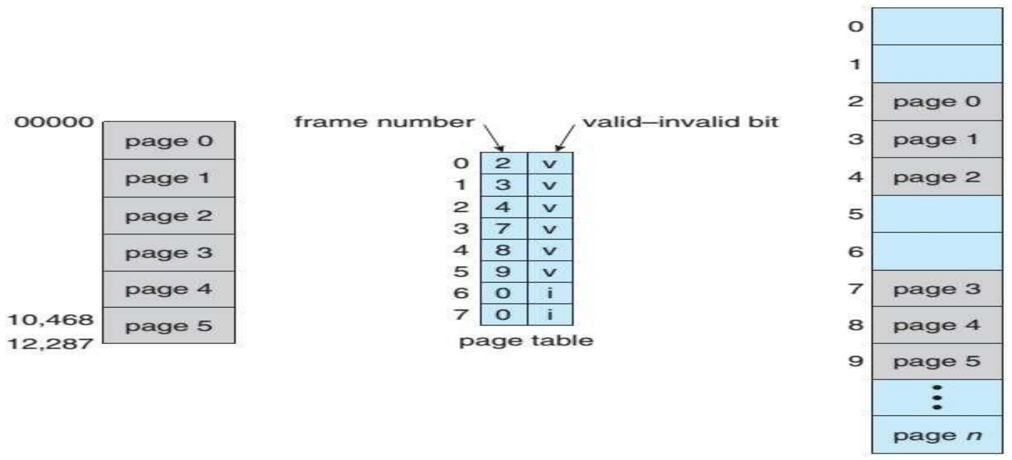
Effective Memory Access Time = (0.80 × 100 ns) + (0.20 × 200 ns)

= 120 nanoseconds

Memory Protection in Paging Environment

Memory protection in a paged environment is accomplished by protection bits associated with each frame. These bits are kept in the page table.

- A one bit **valid–invalid** bit is attached to each entry in the page table.
- When this bit is set to *valid*, the associated page is in the process's logical address space and it is a legal or valid page.
- When the bit is set to *invalid*, the page is not in the process's logical address space. Illegal addresses are trapped by use of the valid–invalid bit.
- The operating system sets this bit for each page to allow or disallow access to the page.



Consider the above figure: A system with a 14-bit address space (0 to 16383), we have a program that should use only addresses 0 to 10468. Each page size is of 2 KB.

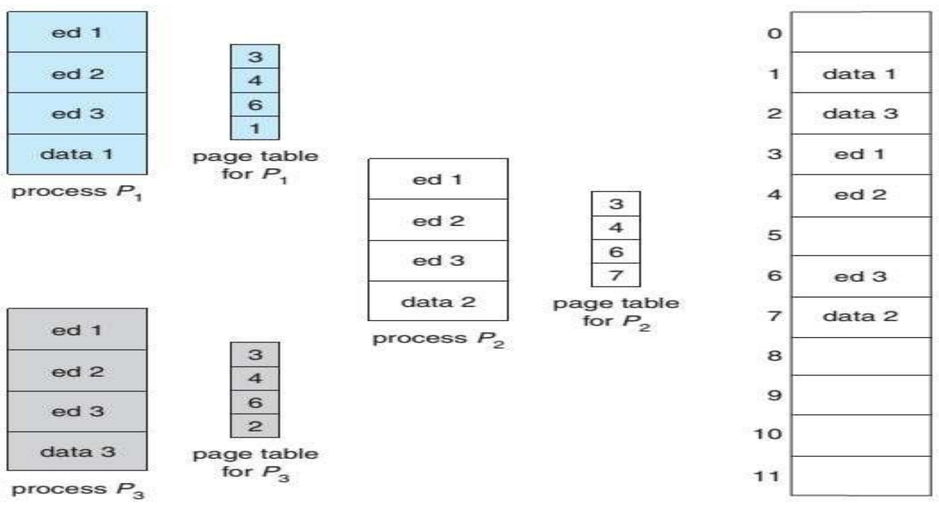
- Addresses in pages 0, 1, 2, 3, 4 and 5 are mapped normally through the page table.
- Any attempt to generate an address in pages 6 or 7 will find that the valid–invalid bit is set to invalid and the computer will trap to the operating system indicating that **Invalid** page reference.

Problem: The program extends only to address 10468, any reference beyond that address is illegal. But references to page 5 are classified as valid, so accesses to addresses up to 12287 are valid. Only the addresses from 12288 to 16383 are invalid.

Solution: To avoid this problem we use a register **Page-Table Length Register (PTLR)** to indicate the size of the page table. This PLTR value is checked against every logical address to verify that the address is in the valid range for the process.

Shared Pages

Paging has an advantage of **Sharing Common Code** This is important in Time sharing Environment.



Consider the above figure that shows a system that supports 40 users, each of whom executes a text editor.

- If the text editor consists of 150 KB of code and 50 KB of data space, we need 8,000 KB to support the 40 users.
- If the code is **Reentrant Code** or **Pure Code** or **Reusable code** it can be shared.
- Each process has its own data page. All three processes sharing a three-page editor each page 50 KB in size.
- Reentrant code is non-self-modifying code (i.e.) it never changes during execution. Thus, two or more processes can execute the same code at the same time.
- Each process has its own copy of registers and data storage to hold the data for the process's execution. The data for two different processes will be different.

Only one copy of the editor need be kept in physical memory.

- Each user's page table maps onto the same physical copy of the editor, but data pages are mapped onto different frames.
- Thus, to support 40 users, we need only one copy of the editor (150 KB), plus 40 copies of the 50 KB of data space per user.
- The total space required is now 2,150 KB instead of 8,000 KB by saving 5850 KB.

Other heavily used programs can also be shared such as compilers, window systems, run-time libraries, database systems and so on.

STRUCTURE OF THE PAGE TABLE

Page tables can be structured in 3 ways:

1. Hierarchical Paging
2. Hashed Page Tables
3. Inverted Page Tables

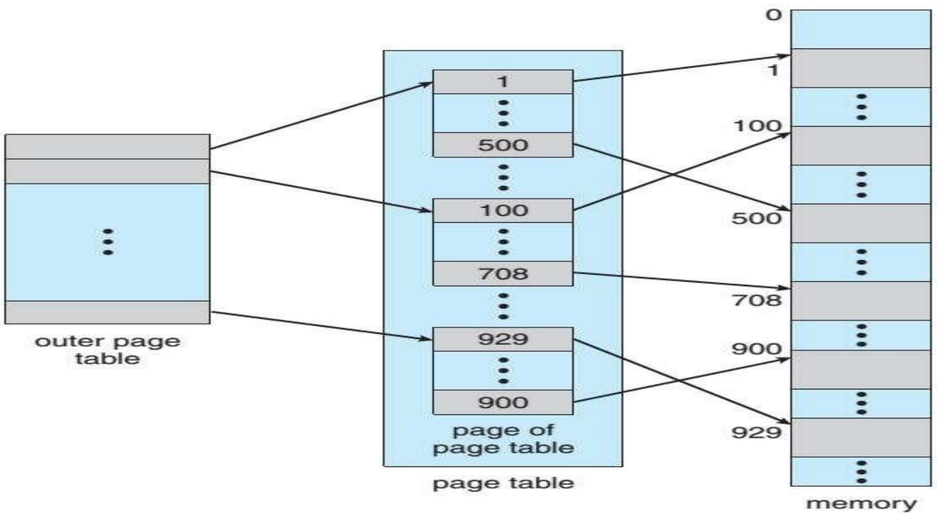
Hierarchical Paging

Most modern computer systems support a large logical address space (2^{32} to 2^{64} Bytes) that leads to excessively larger page tables.

Consider a system with a 32-bit (4GB) logical address space.

- If the page size in such a system is 4 KB (**2^{12}**), then a page table may consist of up to 1 million entries ($2^{32}/2^{12}$).
- Assuming that each entry consists of 4 bytes, each process may need up to 4 MB of physical address space for the page table alone.
- These page tables are not allocated in main memory contiguously and we will divide this page tables into smaller pieces.

Hierarchical paging uses Two Level Paging Algorithm for structuring of page tables. In Two level paging algorithm Page tables are itself paged.

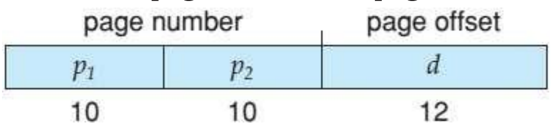


Consider the system with a 32-bit logical address space and a page size of 4 KB.

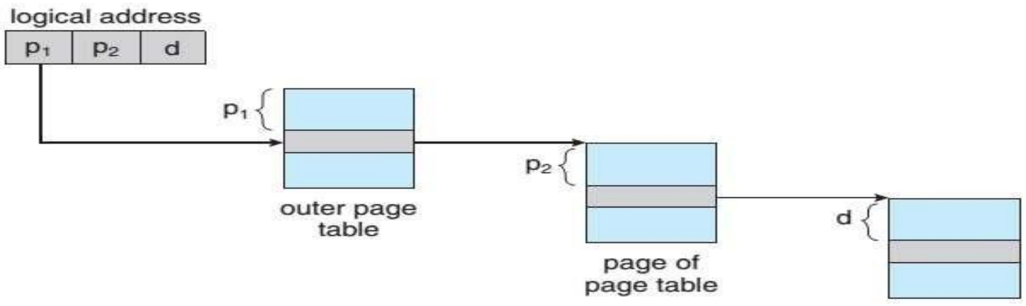
- A logical address is divided into a page number consisting of 20 bits and a page offset consisting of 12 bits.
- Because we page the page table, the page number is further divided into a 10-bit page number and a 10-bit page offset.

A logical address has two indexes: p1 and p2.

- *p*1 is an index into the outer page table
- *p*2 is the displacement within the page of the inner page table.



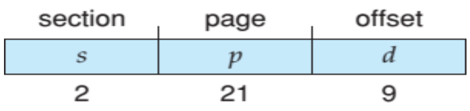
The below figure shows the Address translation for a **Two-level** 32-bit paging architecture. Address translation works from the outer page table inward, this scheme is also known as a **Forward-Mapped Page Table**.



Example of Two level paging: VAX Mini Computer

- The VAX was the most popular minicomputer from 1977 through 2000.
- The VAX architecture supported a variation of **Two-Level paging**.
- The VAX is a 32- bit machine with a page size of 512 bytes.
- The logical address space of a process is divided into **4-equal** sections, each of which consists of $\mathbf{2^{30}}$ bytes.
- Each section represents a different part of the logical address space of a process.
- The first 2 high-order bits of the logical address designate the appropriate section.
- The next 21 bits represent the logical page number of that section and the final 9 bits represent an offset in the desired page.
- By partitioning the page table in this manner, the operating system can leave partitions unused until a process needs them.
- Entire sections of virtual address space are frequently unused and multilevel page tables have no entries for these spaces, greatly decreasing the amount of memory needed to store virtual memory data structures.

An address on the VAX architecture consists of **3-parts**: Segment number (**s**), index to page table (**p**), Displacement with in the page (**d**).

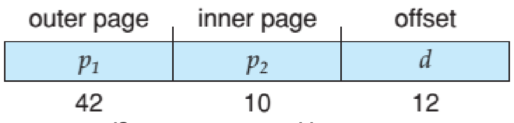


- After this scheme is used, the size of a one-level page table for a VAX process using one section is 2^{21} bits ∗ 4 bytes per entry = 8 MB.
- To further reduce main-memory use, the VAX pages the user-process page tables.

Problems with Two level paging

For a system with a 64-bit logical address space, a two-level paging scheme is no longer appropriate.

Let's take the page size in such a system is 4 KB ($\mathbf{2^{12}}$). In this case, the page table consists of up to $\mathbf{2^{52}}$ entries. If we use a two-level paging scheme, then the inner page tables can conveniently be one page long or contain $\mathbf{2^{10}}$ 4-byte entries.

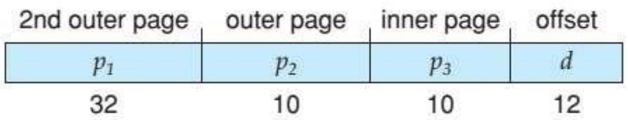


The outer page table consists of $\mathbf{2^{42}}$ entries or $\mathbf{2^{44}}$ bytes. The obvious way to avoid such a large table is to divide the outer page table into smaller pieces.

To avoid this problem we can divide the outer page again called Three level paging.

Three level paging

Suppose that the outer page table is made up of standard-size pages ($\mathbf{2^{10}}$ entries or $\mathbf{2^{12}}$ bytes). In this case, a 64-bit address space is still daunting:



Outer page table is still 2^{34} bytes (16 GB) in size. We can still divide this into **4-Level** paging.

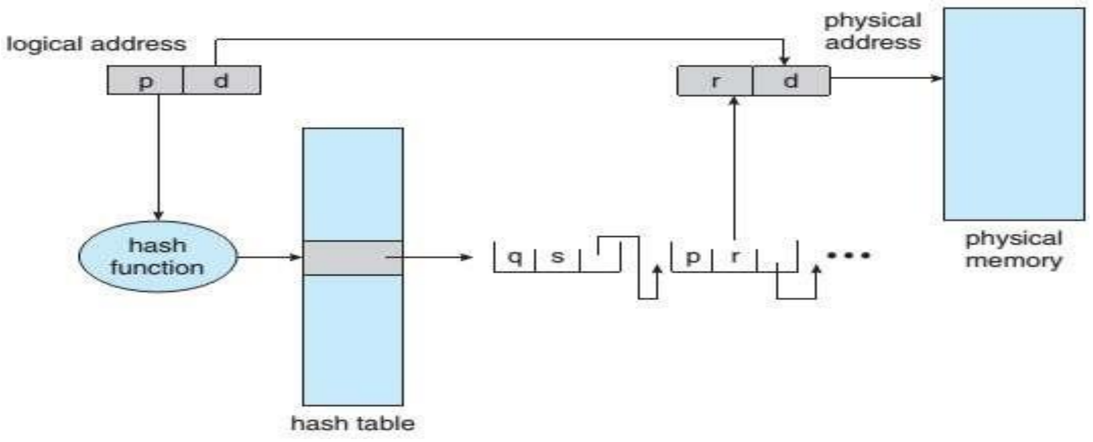
Hashed Page Tables

Hashed page table is used to handling address spaces larger than 32 bits.

Each entry in the hash table contains a linked list of elements that hash to the same location to handle collisions.

Each element consists of three fields:

1. Virtual page number
2. Value of the mapped page frame
3. A pointer to the next element in the linked list.



- The virtual page number in the virtual address is hashed into the hash table.
- The virtual page number is compared with field 1 in the first element in the linked list.
- If there is a match, the corresponding page frame (field 2) is used to form the desired physical address.
- If there is no match, subsequent entries in the linked list are searched for a matching virtual page number.

For 64 bit address space Clustered page table has been proposed.

- Clustered page tables are similar to hashed page tables except that each entry in the hash table refers to several pages (such as 16) rather than a single page.
- Therefore, a single page-table entry can store the mappings for multiple physical-page frames.
- Clustered page tables are particularly useful for **sparse** address spaces, where memory references are noncontiguous and scattered throughout the address space.

Inverted Page Tables

Problem wih page tables:

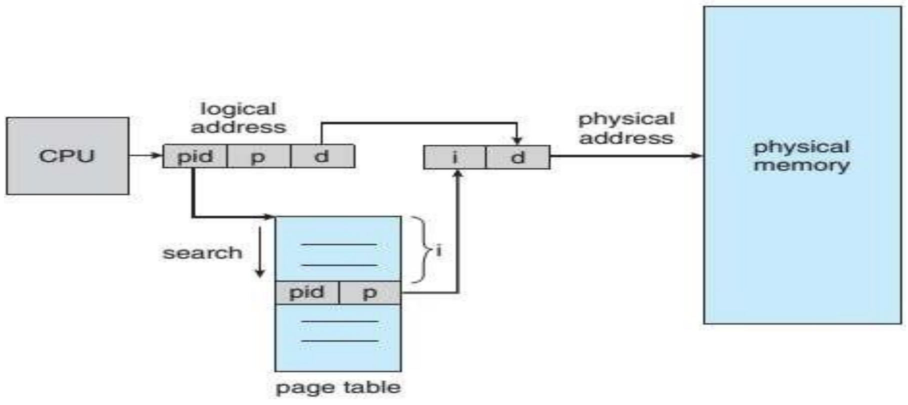
- Each process has an associated page table. The page table has one entry for each page that the process is using.
- Processes reference pages through the pages' virtual addresses.
- The operating system must then translate this reference into a physical memory address.
- Since the table is sorted by virtual address, the operating system is able to calculate where in the table the associated physical address entry is located and to use that value directly.
- The drawback of this method is, each page table may consist of **Millions** of entries.
- These tables may consume large amounts of physical memory just to keep track of how other physical memory is being used.

Solution: Inverted page tables will solve the above problem

An inverted page table has one entry for each real page (i.e. frame) of memory.

- Each entry consists of the virtual address of the page stored in that real memory location with information about the process that owns the page.
- Thus, only one page table is in the system and it has only one entry for each page of physical memory.
- Inverted page tables often require that an address-space identifier be stored in each entry of the page table, since the table usually contains several different address spaces mapping physical memory. Storing the address-space identifier ensures that a logical page for a particular process is mapped to the corresponding physical page frame.

Examples: Inverted page tables are used in the 64-bit UltraSPARC and PowerPC systems.



The above figure shows the inverted page tables used in IBM RT:

- Each virtual address in the system consists of: **<process-id, page-number, offset>**
- Each inverted page-table entry is a pair <process-id, page-number> where the process-id is the address-space identifier.
- When a memory reference occurs, part of the virtual address, consisting of <process-id, page-number> is presented to the memory subsystem. The inverted page table is then searched for a match.
- If a match is found at entry ***i*** then the physical address **<*i*, offset>** is generated.
- If no match is found, then an illegal address access has been attempted.

Problem with inverted page table

- Although this scheme decreases the amount of memory needed to store each page table, it increases the amount of time needed to search the table when a page reference occurs.
- Systems that use inverted page tables have difficulty implementing shared memory.

<u>Oracle SPARC Solaris</u>

Oracle SPARC Solaris is a modern 64-bit CPU and operating system that are tightly integrated to provide low-overhead virtual memory.

- **Solaris** running on the **SPARC** CPU is a fully 64-bit operating system and solves the problem of virtual memory without using up all of its physical memory by keeping multiple levels of page tables.
- It uses **2-Hash Tables**: one for the kernel and one for all user processes. Each maps memory addresses from virtual to physical memory.

- Each hash-table entry represents a contiguous area of mapped virtual memory, which is more efficient than having a separate hash-table entry for each page.
- Each entry has a base address and a span indicating the number of pages the entry represents.

Virtual-to-physical translation would take too long if each address required searching through a hash table, so the CPU implements a TLB that holds translation table entries (TTEs) for fast hardware lookups.

- A cache of these TTEs reside in a **Translation Storage Buffer** (TSB), which includes an entry per recently accessed page.
- When a virtual address reference occurs, the hardware searches the TLB for a translation. If none is found, the hardware walks through the in-memory TSB looking for the TTE that corresponds to the virtual address that caused the lookup. This **TLB walk** functionality is found on many modern CPUs.
- If a match is found in the TSB, the CPU copies the TSB entry into the TLB and the memory translation completes.
- If no match is found in the TSB, the kernel is interrupted to search the hash table. The kernel then creates a TTE from the appropriate hash table and stores it in the TSB for automatic loading into the TLB by the CPU memory-management unit.
- Finally, the interrupt handler returns control to the MMU, which completes the address translation and retrieves the requested byte or word from main memory.

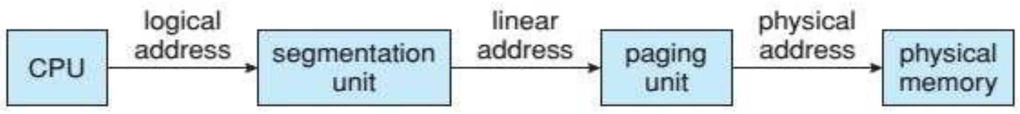
INTEL ARCHITECTURE (IA-32)

The 16-bit Intel 8086 appeared in the late 1970s. Intel 8088 16-bit chip was used in the original IBM PC.

- Both Intel 8086 chip and the 8088 chip were based on a segmented architecture.
- Intel later produced a series of 32-bit chips (IA-32), which included the family of 32-bit Pentium processors.
- The IA-32 architecture supported both paging and segmentation.
- Recently Intel has produced a series of 64-bit chips based on the x86-64 architecture.
- Windows, Mac OS X and Linux operating systems run on Intel chips.

Memory management in IA-32 systems is divided into 2 components:

1. Segmentation
2. Paging.



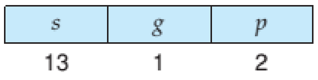
- The CPU generates logical addresses, which are given to the segmentation unit.
- The segmentation unit produces a linear address for each logical address.
- The linear address is then given to the paging unit, which in turn generates the physical address in main memory.
- Thus, the segmentation and paging units form the equivalent of the memory-management unit (MMU).

IA-32 Segmentation

The IA-32 architecture allows a segment to be as large as 4 GB and the maximum number of segments per process is 16K.

The logical address space of a process is divided into two partitions.

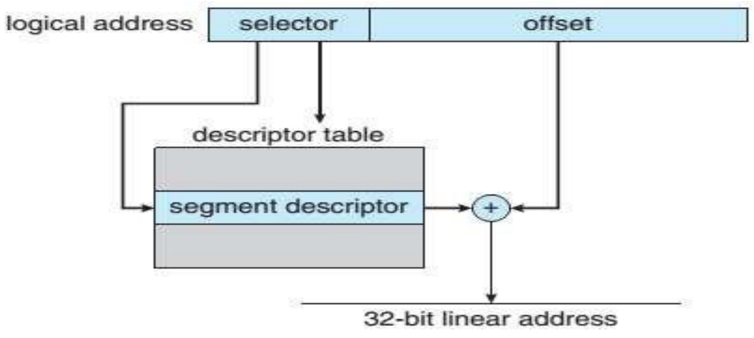
- The first partition consists of up to 8K segments that are private to that process.
- The second partition consists of up to 8 K segments that are shared among all the processes.
- Information about the first partition is kept in the **Local Descriptor Table** (**LDT**).
- Information about the second partition is kept in the **Global Descriptor Table** (**GDT**).
- Each entry in the LDT and GDT consists of an 8-byte segment descriptor with detailed information about a particular segment, including the base location and limit of that segment.



The logical address is a pair **<selector, offset>,** where the selector is a 16-bit number, it consists of:

- 13-bit Segment number (s)
- 1-bit Location of segment in GDT or LDT (g)
- 2-bit protection
- The offset is a 32-bit number specifying the location of the byte within the segment.

The machine has six segment registers, allowing six segments to be addressed at any one time by a process. It also has six 8-byte micro-program registers to hold the corresponding descriptors from either the LDT or GDT. This cache lets the Pentium avoid having to read the descriptor from memory for every memory reference.

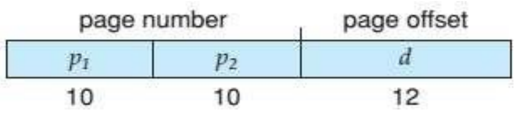


The linear address on the IA-32 is 32 bits long and is formed as follows.

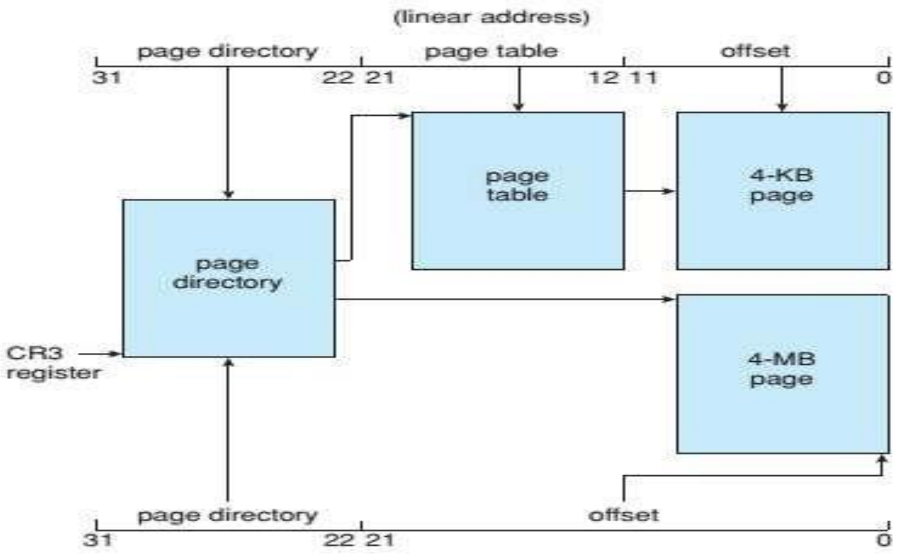
- Segment register points to the appropriate entry in the LDT or GDT.
- The base and limit information about the segment is used to generate a **Linear Address**.
- First, the limit is used to check for address validity.
- If the address is not valid, a memory fault is generated, resulting in a trap to the operating system.
- If it is valid, then the value of the offset is added to the value of the base, resulting in a32-bit linear address.

IA-32 Paging

The IA-32 architecture allows a page size of either 4 KB or 4 MB. For 4-KB pages, IA-32 uses a two-level paging scheme in which the division of the 32-bit linear address is as follows:



- The 10 high-order bits reference an entry in the outermost page table, which IA-32 terms the **Page directory**. The CR3 register points to the page directory for the current process.
- The page directory entry points to an inner page table that is indexed by the contents of the innermost 10 bits in the linear address.
- The low-order bits 0–11 refer to the offset in the 4-KB page pointed to in the page table.



- One entry in the page directory is the Page Size flag. If Page Size flag is set that indicates the size of the page frame is 4 MB and not the standard 4 KB.
- If this flag is set, the page directory points directly to the 4-MB page frame bypassing the inner page table and the 22 low-order bits in the linear address refer to the offset in the 4-MB page frame.

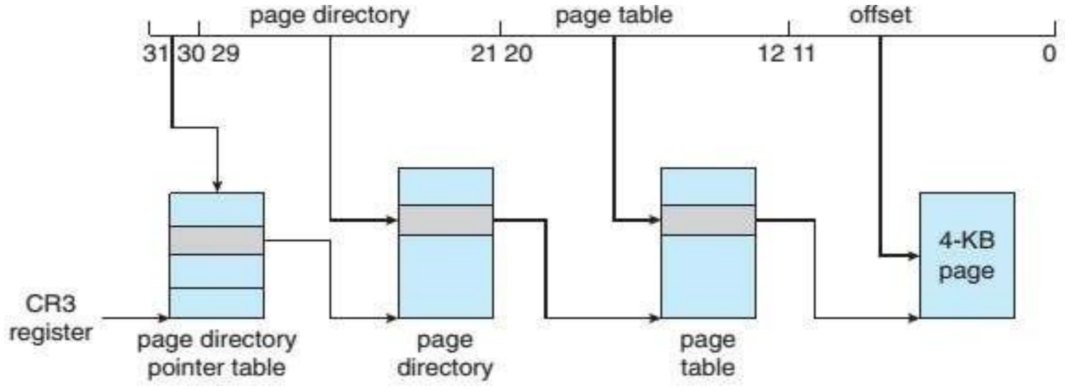
To improve the efficiency of physical memory use, IA-32 page tables can be swapped to disk.

- In this case, an invalid bit is used in the page directory entry to indicate whether the table to which the entry is pointing is in memory or on disk.
- If the table is on disk, the operating system can use the other 31-bits to specify the disk location of the table. The table can then be brought into memory on demand.

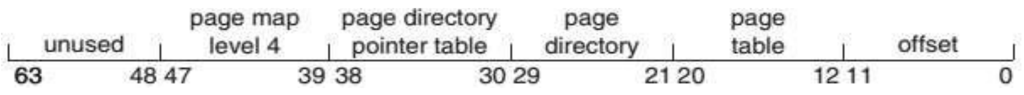
As software developers began to discover the 4-GB memory limitations of 32-bit architectures, Intel adopted a **Page Address Extension (PAE)**, which allows 32-bit processors to access a physical address space larger than 4 GB.

The fundamental difference introduced by PAE support was that paging went from a two-level scheme to a three-level scheme, where the top two bits refer to a **Page Directory Pointer Table**.

The below figure illustrates a PAE system with 4-KB pages.



PAE also increased the page-directory and page-table entries from 32 to 64 bits in size, which allowed the base address of page tables and page frames to extend from 20 to 24 bits.

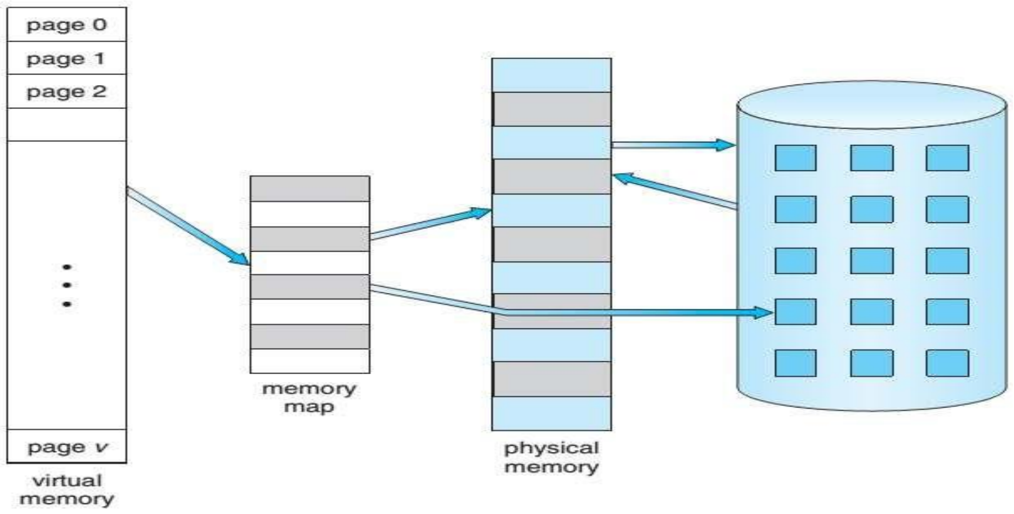


- Combined with the 12-bit offset, adding PAE support to IA-32 increased the address space to 36 bits, which supports up to 64 GB of physical memory.
- It is important to note that operating system support is required to use PAE. Both Linux and Intel Mac OS X support PAE.
- 32-bit versions of Windows desktop operating systems still provide support for only 4 GB of physical memory, even if PAE is enabled.

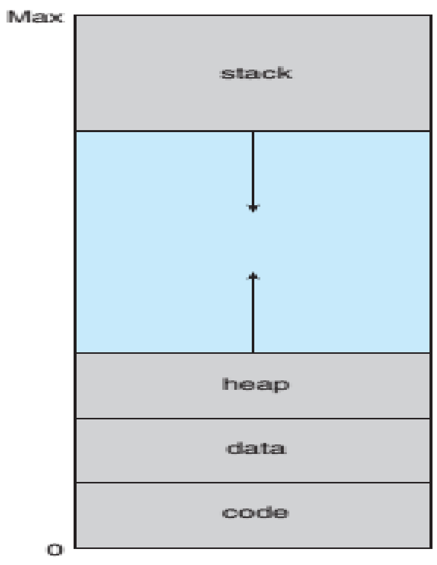
VIRTUAL MEMORY

INTRODUCTION

- Virtual Memory is a technique that allows the execution of processes that are not completely in Main-memory.
- Virtual memory involves the separation of Logical Memory as perceived by users from Physical Memory.
- This separation allows an extremely large virtual memory to be provided for programmers when only a smaller physical memory is available.
- A program would no longer be constrained by the amount of physical memory that is available. Users would be able to write programs for a large ***virtual*** address space.
- Because each user program could take less physical memory, more programs could be run at the same time with a corresponding increase in CPU utilization and throughput but it will not increase the Response time or Turnaround time.
- Less I/O would be needed to load or swap user programs into memory, so each user program would run faster. This process will benefit both system and user.



Virtual Address Space of a process refers to the logical (or virtual) view of how a process is stored in memory. The below figure shows a virtual address space:

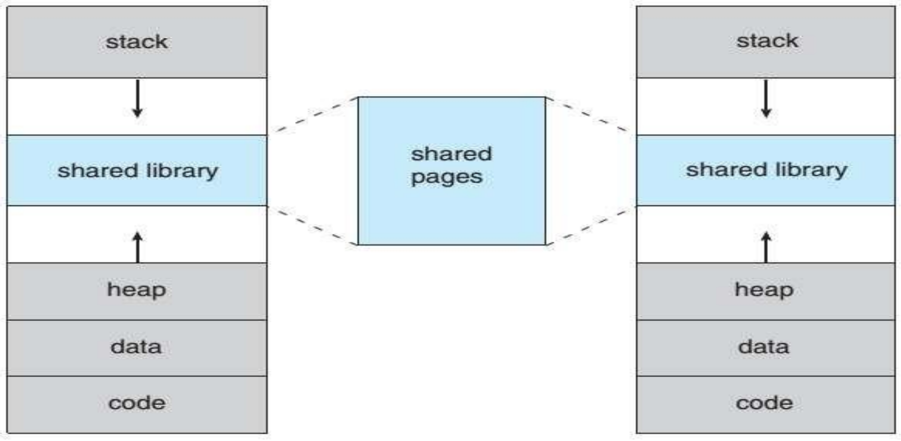


- In the above figure, a process begins at a certain logical address (say address 0) and exists in contiguous memory.
- Physical memory organized in page frames and the physical page frames assigned to a process may not be contiguous.
- **Memory Management Unit** (**MMU**) maps logical pages to physical page frames in Main memory.
- In the above figure, we allow the heap to grow upward in memory as it is used for dynamic memory allocation.
- We allow for the stack to grow downward in memory through successive function calls.
- The large blank space (i.e. hole) between the heap and the stack is part of the Virtual address space but will require actual physical pages only if the heap or stack grows. Virtual address spaces that include holes are known as **Sparse Address Spaces**.
- Using a Sparse Address Space is beneficial because the holes can be filled as the stack or heap segments grow or if we wish to dynamically link libraries or possibly other shared objects during program execution.

Shared Library using Virtual Memory

System libraries can be shared by several processes through mapping of the shared object into a virtual address space.

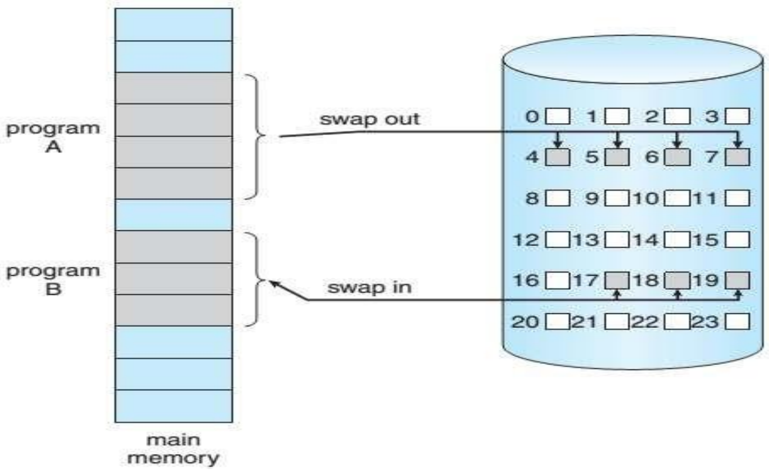
- Each process considers the libraries to be part of its virtual address space, the actual pages where the libraries reside in physical memory are shared by all the processes.
- A library is mapped read-only into the space of each process that is linked with it.



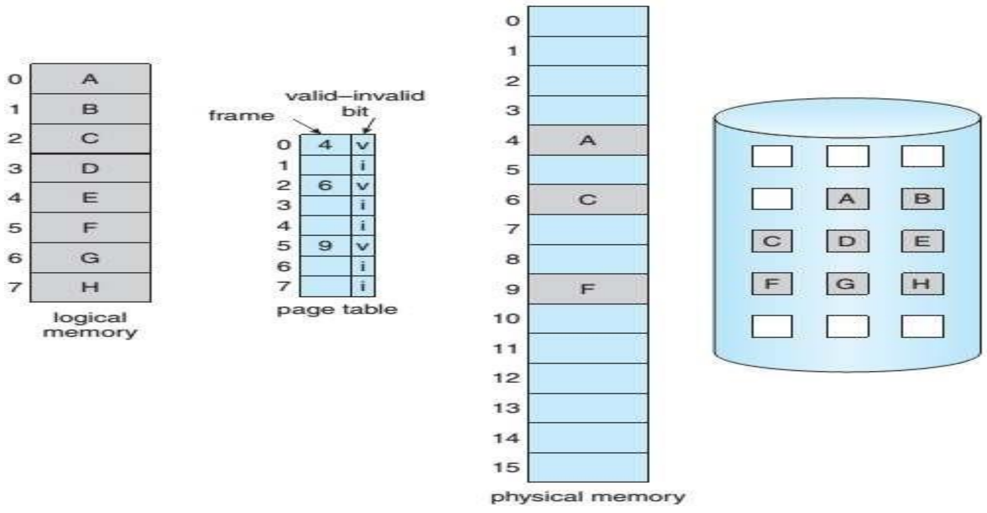
- Two or more processes can communicate through the use of shared memory.
- Virtual memory allows one process to create a region of memory that it can share with another process.
- Processes sharing this region consider it is part of their virtual address space, yet the actual physical pages of memory are shared.
- Pages can be shared during process creation with the fork( ) system call, thus speeding up process creation.

DEMAND PAGING

With Demand-paged virtual memory, pages are loaded only when they are demanded during program execution. Pages that are never accessed are never loaded into physical memory.



- A demand-paging system is similar to a paging system with swapping, where processes reside in secondary memory (i.e. disk).
- Demand paging uses the concept of **Lazy Swapper** or **Lazy Pager**. A lazy swapper or pager never swaps a page into memory unless that page will be needed.
- **Valid–Invalid** bit is used to distinguish between the pages that are in memory and the pages that are on the disk.
- When this bit is set to "valid," the associated page is both legal and is in main memory.
- If the bit is set to "invalid," the page either is not valid (i.e. not in the logical address space of the process) or is valid but is currently on the disk.
- The page-table entry for a page that is brought into main memory is set to valid, but the page-table entry for a page that is not currently in main memory is either marked as invalid or contains the address of the page on disk.

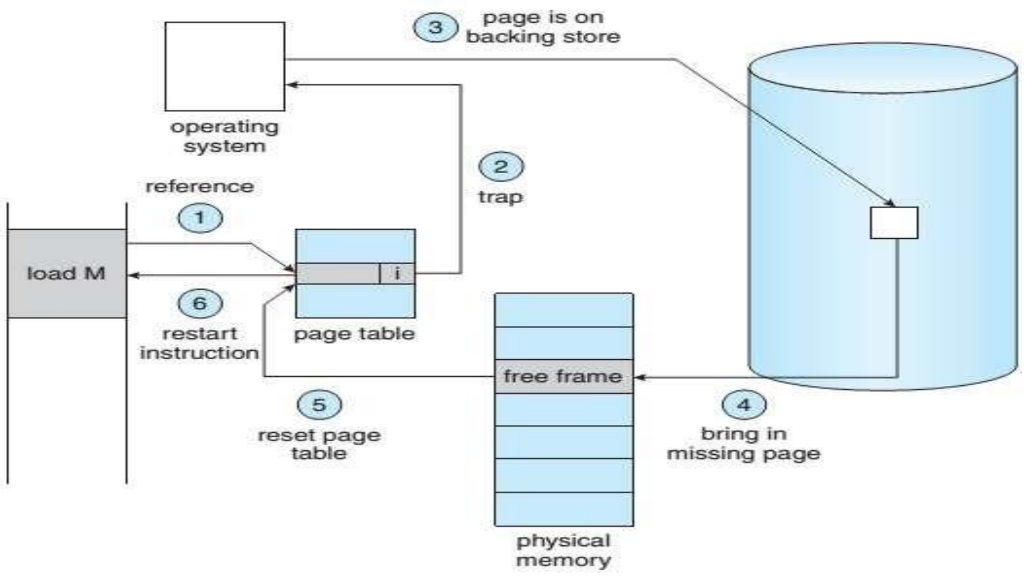


Note: The process executes and accesses pages that are **Main Memory Resident** then the execution proceeds normally.

PAGE FAULT

Access to a page marked invalid causes a **Page Fault**. The paging hardware, in translating the address through the page table will notice that the invalid bit is set that causes a trap to the operating system. This trap is the result of the operating system's failure to bring the desired page into memory.

Procedure for Handling Page Fault



1. We check an internal page table kept with the **Process Control Block** for this process to determine whether the reference was a valid or an invalid memory access.
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 (Example: by taking one from the free-frame list).
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 also the page table to indicate that the page is now in memory.
6. We restart the instruction that was interrupted by the trap. The process can now access the page in main memory.

Pure Demand Paging

- System can start executing a process with ***no*** pages in memory.
- When the operating system sets the instruction pointer to the first instruction of the process and that process is not resides in main memory then the process immediately faults for the page.
- After this page is brought into memory, the process continues to execute and faulting as necessary until every page that it needs is in memory.
- At that point, it can execute with no more faults. This scheme is called **Pure Demand Paging**.
- Pure Demand Paging never brings a page into memory until it is required.

Hardware support for Demand Paging

The hardware to support demand paging is the same as the hardware for paging and swapping:

- **Page table** has the ability to mark an entry as invalid through a valid–invalid bit or a special value of protection bits.
- **Secondary 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**.

Performance of Demand Paging

Demand paging can significantly affect the performance of a computer system.

- As long as we have no page faults, the effective access time is equal to the memory access time.
- If a page fault occurs, we must first read the relevant page from disk and then access the desired word.

$$\boxed{\textbf{Effective Access Time} = (1 - p) \times ma + p \times \textbf{page fault time}}$$

- Where ma denotes Memory Access Time
- p be probability of a page fault ($0 \leq p \leq 1$). If p is closer to zero then there are few page faults.

Sequence of Steps followed by Page Fault

1. Trap to the operating system.
2. Save the user registers and process state.
3. Determine that the interrupt was a page fault.
4. Check the page reference was legal and determines the location of the page on the disk.
5. Issue a read from the disk to a free frame:
    a. Wait in a queue for this device until the read request is serviced.
    b. Wait for the device seek and/or latency time.
    c. Begin the transfer of the page to a free frame.
6. While waiting, allocate the CPU to some other user by using CPU scheduling.
7. Receive an interrupt from the disk I/O subsystem (I/O completed).
8. Save the registers and process state for the other user.
9. Determine that the interrupt was from the disk.
10. Correct the page table and other tables to show that the desired page is now in memory.
11. Wait for the CPU to be allocated to this process again.
12. Restore the user registers, process state and new page table and then resume the interrupted instruction.

Example: With an average page-fault service time of 8 milliseconds and a memory access time of 200 nanoseconds, the effective access time in nanoseconds is:

$$\begin{aligned}\text{Effective Access Time} &= (1 - p) \times (200) + p\ (8 \text{ milliseconds}) \\ &= (1 - p) \times 200 + p \times 8{,}000{,}000 \\ &= 200 + 7{,}999{,}800 \times p.\end{aligned}$$

Note: The effective access time is directly proportional to the **Page-Fault rate.**

COPY-ON-WRITE

- The fork( ) system call creates a child process that is a duplicate of its parent.
- fork( ) worked by creating a copy of the parent's address space for the child, duplicating the pages belonging to the parent.
- Many child processes invoke the exec( ) system call immediately after creation and the copying of the parent's address space may be unnecessary.
- **Copy-on-Write** is a technique which allows the parent and child processes initially to share the same pages.
- These shared pages are marked as Copy-on-Write pages, meaning that if either process writes to a shared page, a copy of the shared page is created.

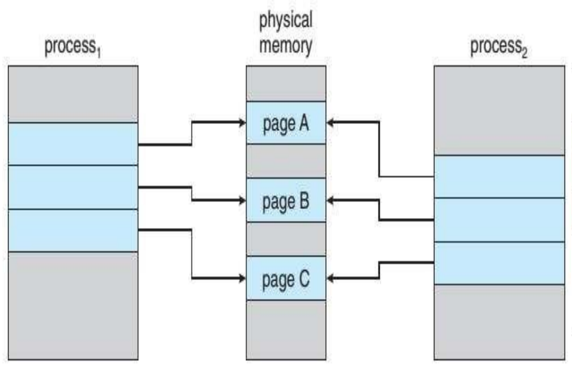


Figure 9.7 Before process 1 modifies page C.

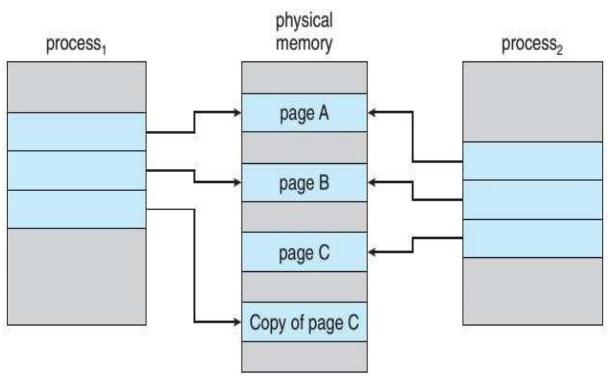


Figure 9.8 After process 1 modifies page C.

Example: Assume that the child process attempts to modify a page containing portions of the stack, with the pages set to be Copy-on-Write.

- Operating system will create a copy of this page, mapping it to the address space of the child process.
- The child process will then modify its own copied page but not the page belonging to the parent process.
- When the Copy-on-Write technique is used, only the pages that are modified by either process are copied.
- Only pages that can be modified need be marked as Copy-on-Write. Pages that cannot be modified can be shared by the parent and child processes.
- Windows XP, Linux and Solaris operating systems uses Copy-on-Write technique.

Zero-fill-on-demand

- When it is determined that a page is going to be duplicated using Copy-on-Write, it is important to note the location from which the free page will be allocated.
- Many operating systems provide a **pool** of free pages for such requests. These free pages are typically allocated when the stack or heap for a process must expand or when there are Copy-on-Write pages to be managed.
- OS typically allocate these pages using a technique known as **Zero-fill-on-demand**. In Zero-fill-on-demand the previous contents of the pages are erased before being allocated.

vfork( ): Virtual Memory fork

vfork( ) does not support Copy-on-Write technique used by Solaris and Linux.
vfork( ) is modified version of fork( ) system call.

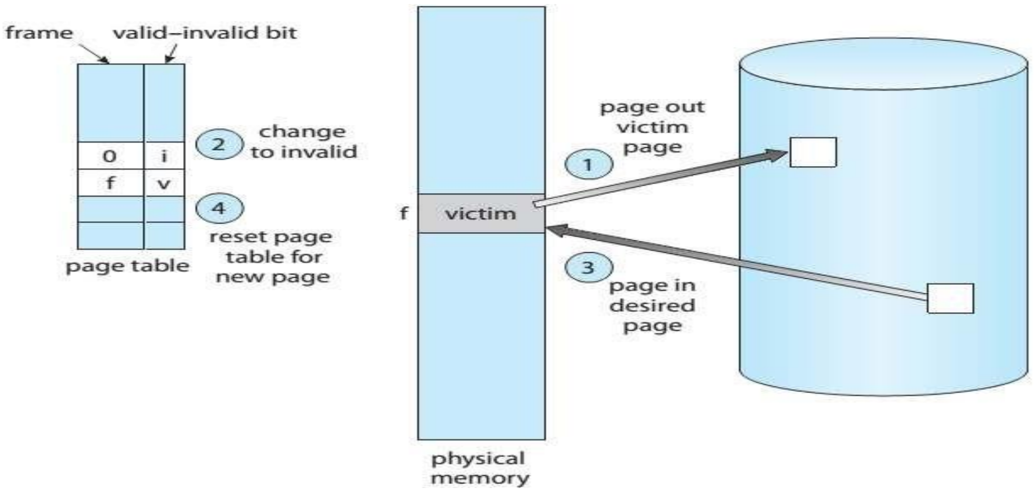
- With vfork( ), the parent process is suspended and the child process uses the address space of the parent.
- Because vfork( ) does not use Copy-on-Write, if the child process changes any pages of the parent's address space, the altered pages will be visible to the parent once it resumes.
- Therefore, vfork( ) must be used with caution to ensure that the child process does not modify the address space of the parent.
- vfork( ) is intended to be used when the child process calls exec( ) immediately after creation.
- vfork( ) is an extremely efficient because it does not copy any pages at the time of process creation. Vfork( ) sometimes used to implement UNIX command-line shell interfaces.

PAGE REPLACEMENT ALGORITHMS

- If no frame is free, we find one that is not currently being used and free it.
- We can free a frame by writing its contents to swap space and changing the page table to indicate that the page is no longer in main memory.
- We can now use the freed frame to hold the page for which the process faulted.

We modify the page-fault service routine to include Page Replacement:

1. Find the location of the desired page on the disk.
2. Find a free frame:
    a. If there is a free frame, use it.
    b. If there is no free frame, use a Page-Replacement algorithm to select a **Victim frame**.
    c. Write the victim frame to the disk and change the page table and frame table.
3. Read the desired page into newly freed frame and change the page table and frame table.
4. Continue the user process from where the page fault occurred.



Modify bit or Dirty bit

Each page or frame has a modify bit associated with it in the hardware.

- The modify bit for a page is set by the hardware whenever any byte in the page has been modified.
- When we select a page for replacement, we examine its modify bit.
- If the bit is set, the page has been modified since it was **read in** from the disk. Hence we must write the page to the disk.
- If the modify bit is not set, the page has ***not*** been modified since it was read into memory. Hence there is no need for write the memory page to the disk, because it is already there.

Note: To determine the number of page faults for a particular reference string and page-replacement algorithm, we also need to know the number of page frames available. As the number of frames available increases, the number of page faults decreases.

There is several Page Replacement Algorithms are in use:

1. FIFO Page Replacement Algorithm
2. Optimal Page Replacement Algorithm
3. LRU Page Replacement Algorithm
4. Counting Based Page Replacement Algorithm

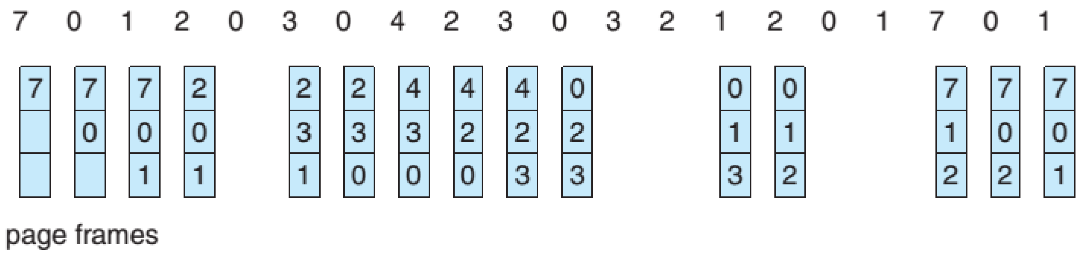
First-In-First-Out Page Replacement Algorithm

FIFO algorithm associates with time of each page when it was brought into main memory.

- When a page must be replaced, the oldest page is chosen.
- We can create a FIFO queue to hold all pages in memory.
- We replace the page at the **Head** of the queue.
- When a page is brought into memory, we insert it at the tail of the queue.

Example: Consider the below reference string and memory with three frames

7, 0, 1, 2, 0, 3, 0, 4, 2, 3, 0, 3, 2, 1, 2, 0, 1, 7, 0, 1



- First **3-references** (7, 0, 1) cause **3-Page faults** and are brought into these empty frames.
- The next reference (2) replaces page 7, because page 7 was brought in first.
- Since 0 is the next reference and 0 is already in memory, we have no fault for this reference.
- The first reference to 3 results in replacement of page 0, since it is now first in line. Because of this replacement, the next reference to 0, will fault. Page 1 is then replaced by page 0.
- By the end, there are **Fifteen** page faults altogether.

Problem: Belady's Anomaly

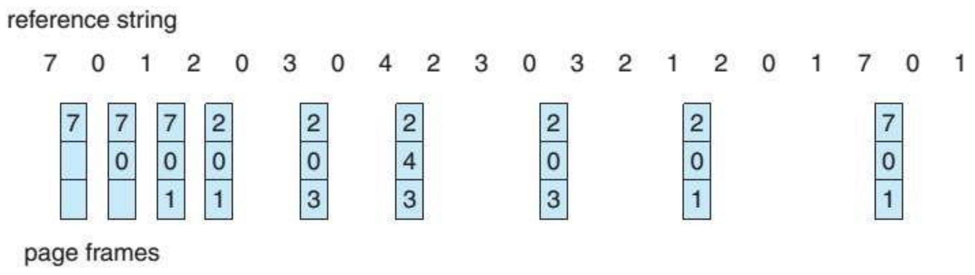
Belady's Anomaly states that: the page-fault rate may ***increase*** as the number of allocated frames increases. Researchers identifies that Belady's anomaly is solved by using Optimal Replacement algorithm.

Optimal Page Replacement Algorithm (OPT Algorithm)

- It will never suffer from Belady's anomaly.
- OPT states that: Replace the page that will not be used for the longest period of time.
- OPT guarantees the lowest possible page fault rate for a fixed number of frames.

Example: Consider the below reference string and memory with three frames

7, 0, 1, 2, 0, 3, 0, 4, 2, 3, 0, 3, 2, 1, 2, 0, 1, 7, 0, 1



- The first **3-references** cause faults that fill the **3-empty** frames.
- The reference to page 2 replaces page 7, because page 7 will not be used until reference number 18, whereas page 0 will be used at 5 and page 1 at 14.

- The reference to page 3 replaces page 1 because page 1 will be the last of the three pages in memory to be referenced again.
- At the end there are only **9-page faults** by using optimal replacement algorithm which is much better than a FIFO algorithm with **15-page faults**.

Note: No replacement algorithm can process this reference string in **3-frames** with fewer than **9-faults**.

Problem with Optimal Page Replacement algorithm

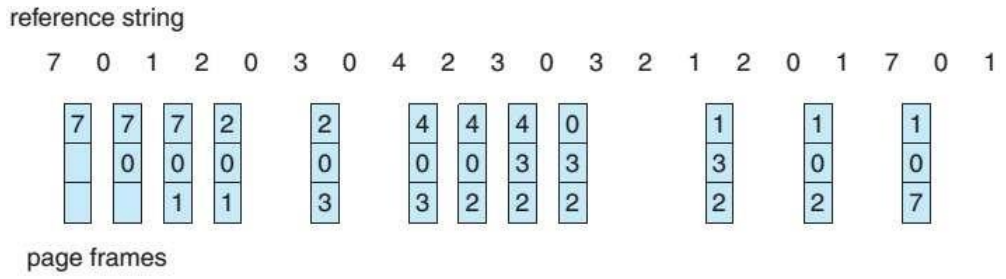
The optimal page-replacement algorithm is difficult to implement, because it requires future knowledge of the reference string. The optimal algorithm is used mainly for comparison studies (i.e. performance studies).

LRU Page Replacement Algorithm

In LRU algorithm, the page that has **not** been used for the **longest** period of time will be replaced (i.e.) we are using the recent past as an approximation of the near future.
LRU replacement associates with each page the time of that page's last use.

Example: Consider the below reference string and memory with three frames

7, 0, 1, 2, 0, 3, 0, 4, 2, 3, 0, 3, 2, 1, 2, 0, 1, 7, 0, 1



- The first five faults are the same as those for optimal replacement.
- When the reference to page 4 occurs LRU replacement sees that out of the three frames in memory, page 2 was used least recently.
- Thus, the LRU algorithm replaces page 2, not knowing that page 2 is about to be used.
- When it then faults for page 2, the LRU algorithm replaces page 3, since it is now the least recently used of the three pages in memory.
- The total number of page faults with LRU is 12 which is less as compared to FIFO.

LRU can be implemented in two ways: Counters and Stack

Counters

- Each page-table entry is associated with a time-of-use field and add to the CPU a logical clock or counter. The clock is incremented for every memory reference.
- Whenever a reference to a page is made, the contents of the clock register are copied to the time-of-use field in the page-table entry for that page.
- We replace the page with the smallest time value. This scheme requires a search of the page table to find the LRU page and a write to memory to the time-of-use field in the page table for each memory access.
- The times must also be maintained when page tables are changed due to CPU scheduling.

Stack

LRU replacement is implemented by keeping a stack of page numbers.

- Whenever a page is referenced, it is removed from the stack and put on the top.
- In this way, the most recently used page is always at the top of the stack and the least recently used page is always at the bottom.
- Because entries must be removed from the middle of the stack, it is best to implement this approach by using a doubly linked list with a Head pointer and a Tail pointer.
- Removing a page and putting it on the top of the stack then requires changing six pointers at worst.
- Each update is a little more expensive, but there is no search for a replacement.
- The tail pointer points to the bottom of the stack, which is the LRU page.
- This approach is particularly appropriate for software or microcode implementations of LRU replacement.

Counting-Based Page Replacement Algorithm

There are two approaches in this scheme: LFU and MFU

Least Frequently Used (LFU)

- In LFU page-replacement algorithm, the page with the smallest count will be replaced.
- Reason for this selection is that an actively used page should have a large reference count.
- A problem arises when a page is used heavily during the initial phase of a process but then is never used again.
- Since it was used heavily, it has a large count and remains in memory even though it is no longer needed.
- One solution is to shift the counts right by 1 bit at regular intervals, forming an exponentially decaying average usage count.

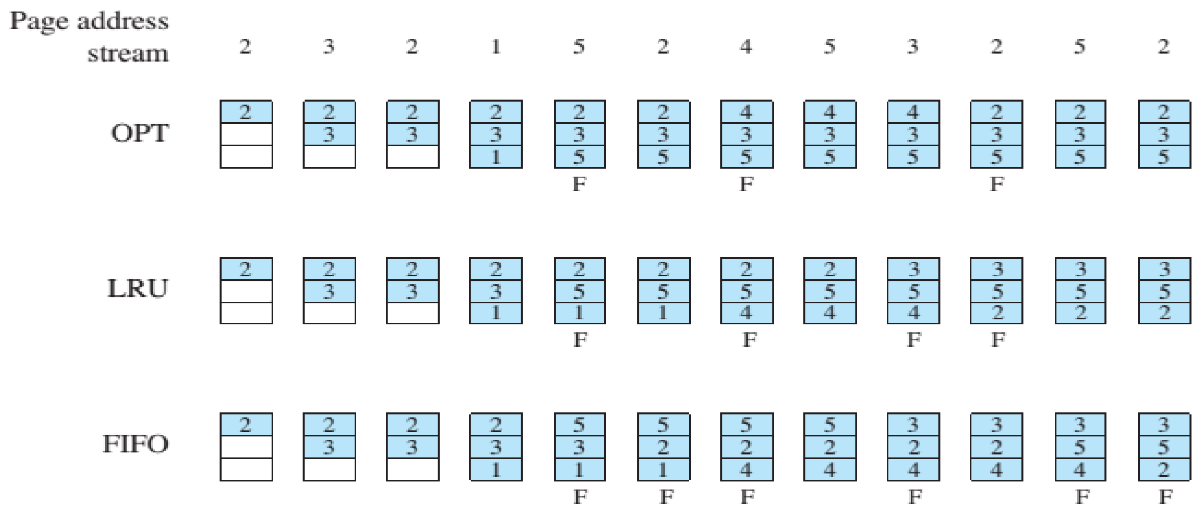
Most Frequently Used (MFU)

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

Note: Neither MFU nor LFU replacement is common. The implementation of these algorithms is expensive and they do not approximate OPT replacement well.

Example:2 Consider the below reference string and the frame size is 3.

2,3,2,1,5,2,4,5,3,2,5,2


Page address stream
2 3 2 1 5 2 4 5 3 2 5 2
OPT
2 | 2 3 | 2 3 | 2 3 1 | 2 3 5 F | 2 3 5 | 4 3 5 F | 4 3 5 | 4 3 5 | 2 3 5 F | 2 3 5 | 2 3 5
LRU
2 | 2 3 | 2 3 | 2 3 1 | 2 5 1 F | 2 5 1 | 2 5 4 F | 2 5 4 | 3 5 4 F | 3 5 2 F | 3 5 2 | 3 5 2
FIFO
2 | 2 3 | 2 3 | 2 3 1 | 5 3 1 F | 5 2 1 F | 5 2 4 F | 5 2 4 | 3 2 4 F | 3 2 4 | 3 5 4 F | 3 5 2 F

ALLOCATION OF FRAMES

There are different approaches used for allocation of frames:

1. Minimum Number of Frames
2. Allocation Algorithms
3. Global versus Local Allocation
4. Non-Uniform Memory Access

Minimum Number of Frames

Allocation is based on minimum number of frames per process is defined by the computer architecture. The maximum number is defined by the amount of available physical memory. We must also allocate at least a minimum number of frames.

- One reason for allocating at least a minimum number of frames involves performance.
- When a page fault occurs before an executing instruction is complete, the instruction must be restarted.
- As the number of frames allocated to each process decreases, the page-fault rate increases and slows the process execution.
- Hence the process must have enough frames to hold all the different pages that any single instruction can reference.

Allocation Algorithms

There are two algorithm are used: Equal allocation and Proportional allocation.

Equal Allocation

- This algorithm splits *m* frames among *n* processes is to give everyone an equal share, *m*/*n* frames.
- Example: If there are 93 frames and five processes, each process will get 18 frames (93/5=18). The **3-leftover** frames can be used as a free-frame buffer pool.

Problem with Equal Allocation

Consider a system with a **1-KB** frame size and two processes P1 and P2.

- Process **P1** is of **10 KB** and process **P2** is of **127 KB** are the only two processes running in a system with **62** free frames.
- Now if we apply Equal allocation then both P1 and P2 will get **31** frames.
- It does not make sense to give P1 process to **31** frames where its maximum use is **10** frames and other **21** frames are **wasted**.

Proportional Allocation

- In this algorithm we allocate available memory to each process according to its size.
- Let the $\mathbf{s_i}$ be the size of the virtual memory for process $\mathbf{P_i}$ then $\mathbf{S = \sum s_i}$
- If the total number of available frames is m, we allocate $\mathbf{a_i}$ frames to process **Pi**, where $\mathbf{a_i}$ is approximately: $\mathbf{a_i = s_i/S \times m}$.
- We must adjust each $\mathbf{a_i}$ to be an integer that is greater than the minimum number of frames required by the instruction set, with a sum not exceeding **m**.

Example: With proportional allocation, we would split **62** frames between **2-processes**, one of **10** pages and one of **127** pages, by allocating **4** frames for P1 and **57** frames for P2.

$$\mathbf{P1\text{->}\ 10/137 \times 62 \approx 4}$$
$$\mathbf{P2\text{->}\ 127/137 \times 62 \approx 57}$$

In this way both processes share the available frames according to their “needs,” rather than equally.

Global versus Local Allocation

With multiple processes competing for frames, we can classify page-replacement algorithms into two broad categories: **Global Replacement** and **Local Replacement**.

- Global replacement allows a process to select a replacement frame from the set of all frames, even if that frame is currently allocated to some other process (i.e.) one process can take a frame from another process.
- Local replacement requires that each process select from only its own set of allocated frames.

Example: Consider an allocation scheme wherein we allow High-priority processes to select frames from low-priority processes for replacement.

- A process can select a replacement from its own frames or the frames of any lower-priority process.
- This approach allows a High-priority process to increase its frame allocation at the expense of a low-priority process.
- With a local replacement strategy, the number of frames allocated to a process does not change.
- With global replacement, a process may happen to select only frames allocated to other processes, thus increasing the number of frames allocated to it.

Non-Uniform Memory Access (NUMA)

- Consider a system is made up of several system boards, each containing multiple CPUs and some memory.
- In systems with multiple CPUs, a one CPU can access some sections of main memory faster than it can access others.
- The system boards are interconnected in various ways, ranging from system buses to High-speed network connections.
- The CPUs on a particular board can access the memory on that board with less delay than they can access memory on other boards in the system.
- Systems in which memory access times vary significantly are known collectively as **Non-Uniform Memory Access (NUMA)** systems.
- NUMA systems are slower than systems in which memory and CPUs are located on the same motherboard.

THRASHING

A process is thrashing if it is spending more time for paging than executing.

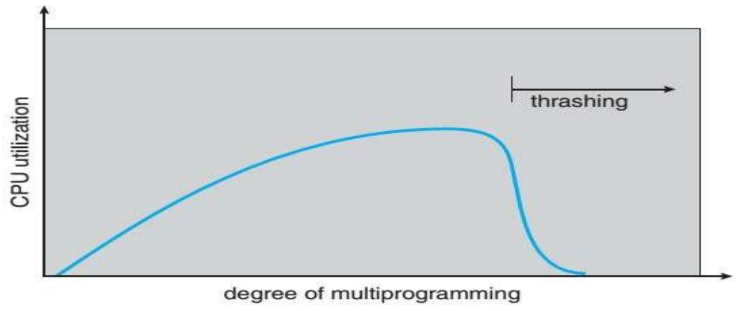
- If the number of frames allocated to a low-priority process falls below the minimum number required by the computer architecture, we must suspend that process’s execution.
- We should then page out its remaining pages, freeing all its allocated frames.
- This provision introduces a swap-in, swap-out level of intermediate CPU scheduling.
- If the process does not have the number of frames it needs to support pages in active use, it will quickly page-fault and the process must replace some page.

- Since all of its pages are in active use, it must replace a page that will be needed again right away. Consequently, it quickly faults again and again by replacing pages that it must bring back in immediately.
- This high paging activity is called **Thrashing**.

Cause of Thrashing

The operating system monitors CPU utilization. If CPU utilization is too low, we increase the degree of multiprogramming by introducing a new process to the system.

- If a global page-replacement algorithm is used then it replaces pages without regard to the process to which the pages are belongs to.
- Now suppose that a process enters a new phase in its execution and needs more frames. It starts faulting and taking frames away from other processes.
- These processes need those pages which have been faulted earlier so they also fault taking frames from other processes.
- These faulting processes must use the paging device to swap pages in and out.
- As they queue up for the paging device, the ready queue empties. As processes wait for the paging device, CPU utilization decreases.
- The CPU scheduler sees the decreasing CPU utilization and ***increases*** the degree of multiprogramming by introducing new process in to the system again.
- The new process tries to get started by taking frames from running processes, causing more page faults and a longer queue for the paging device.
- As a result, CPU utilization drops even further and the CPU scheduler tries to increase the degree of multiprogramming even more.
- Thrashing has occurred and system throughput decreases. The page fault rate increases tremendously. As a result, the effective memory-access time increases.
- No work is getting done, because the processes are spending all their time paging.



Consider the above figure that show how thrashing will occur:

- As the degree of multiprogramming increases, CPU utilization also increases until a maximum is reached.
- If the degree of multiprogramming is increased even further then thrashing occurs and CPU utilization drops sharply.
- At this point, we must stop thrashing and increase the CPU utilization by decreasing the the degree of multiprogramming.

Solutions to Thrashing

1. Local Replacement Algorithm (or) Priority Replacement Algorithm
2. Locality Model

Local Replacement Algorithm

- With local replacement, if one process starts thrashing, it cannot steal frames from another process.
- Local replacement Algorithm limits thrashing but it cannot avoid thrashing entirely.
- If processes are thrashing, they will be paging device queue most of the time.
- The average service time for a page fault will increase because of the longer average queue for the paging device.
- Thus, the effective access time will increase even for a process that is not thrashing.

Locality Model

- The locality model states that, as a process executes, it moves from locality to locality.
- A locality is a set of pages that are actively used together. A program is generally composed of several different localities, which may overlap.

Example: When a function is called, it defines a new locality. In this locality, memory references are made to the instructions of the function call, its local variables and a subset of the global variables. When we exit the function, the process leaves this locality, since the local variables and instructions of the function are no longer in active use. We may return to this locality later.
Note: Localities are defined by the program structure and its data structures.

Suppose we allocate enough frames to a process to accommodate its current locality.

- It will fault for the pages in its locality until all these pages are in memory; then, it will not fault again until it changes localities.
- If we do not allocate enough frames to accommodate the size of the current locality, the process will thrash, since it cannot keep in memory all the pages that it is actively using.

VIRTUAL MEMORY IN WINDOWS

Windows implements virtual memory using demand paging with **Clustering**.

- Clustering handles page faults by bringing in not only the faulting page but also several pages following the faulting page.
- When a process is first created, it is assigned a working-set minimum and maximum.
- The **Working-set Minimum** is the minimum number of pages the process is guaranteed to have in memory.
- If sufficient memory is available, a process may be assigned as many pages as its **Working-set Maximum**.
- The virtual memory manager maintains a list of free page frames. A threshold value is associated with this free frame list that is used to indicate whether sufficient free memory is available.
- If a page fault occurs for a process that is below its working-set maximum, the virtual memory manager allocates a page from this list of free pages.
- If a process that is at its working-set maximum incurs a page fault, it must select a page for replacement using a local LRU page-replacement policy.

- When the amount of free memory falls below the threshold, the virtual memory manager uses a tactic known as **Automatic Working-set Trimming** to restore the value above the threshold.
- Automatic working-set trimming works by evaluating the number of pages allocated to processes.
- If a process has been allocated more pages than its working-set minimum, the virtual memory manager removes pages until the process reaches its working-set minimum.
- A process that is at its working-set minimum may be allocated pages from the free-page-frame list once sufficient free memory is available.
- Windows performs working-set trimming on both user mode and system processes.

MAHAVEER INSTITUTE OF SCIENCE AND TECHNOLOGY

(AN UGC AUTONOMOUS INSTITUTION)

Approved by AICTE, Affiliated to JNTUH, Accredited by NAAC with 'A' Grade
Recognized Under Section 2(f) of UGC Act 1956, ISO 9001:2015 Certified
Vyasapuri, Bandlaguda, Post: Keshavgiri, Hyderabad- 500 005, Telangana, India.
https://www.mist.ac.in E-mail:principal.mahaveer@gmail.com, Mobile: 8978380692

ESTD : 2001

Department of Computer Science and Engineering (AIML)

(R22) OPERATING SYSTEM

Lecture Notes

B. Tech II YEAR – I SEM

Prepared by

SANGYAM SOUNDARYA
(Assistant Professor)
Dept.CSE(AIML)

OPERATING SYSTEMS

B.Tech. II Year I Sem.

L	T	P	C
3	**0**	**0**	**3**

Prerequisites:

1. A course on “Computer Programming and Data Structures”.
2. A course on “Computer Organization and Architecture”.

Course Objectives:

- Introduce operating system concepts (i.e., processes, threads, scheduling, synchronization, deadlocks, memory management, file and I/O subsystems and protection)
- Introduce the issues to be considered in the design and development of operating system
- Introduce basic Unix commands, system call interface for process management, interprocess communication and I/O in Unix

Course Outcomes:

- Will be able to control access to a computer and the files that may be shared
- Demonstrate the knowledge of the components of computers and their respective roles in computing.
- Ability to recognize and resolve user problems with standard operating environments.
- Gain practical knowledge of how programming languages, operating systems, and architectures interact and how to use each effectively.

UNIT - I

Operating System - Introduction, Structures - Simple Batch, Multiprogrammed, Time-shared, Personal Computer, Parallel, Distributed Systems, Real-Time Systems, System components, Operating System services, System Calls

Process - Process concepts and scheduling, Operations on processes, Cooperating Processes, Threads

UNIT - II

CPU Scheduling - Scheduling Criteria, Scheduling Algorithms, Multiple -Processor Scheduling. System call interface for process management-fork, exit, wait, waitpid, exec

Deadlocks - System Model, Deadlocks Characterization, Methods for Handling Deadlocks, Deadlock Prevention, Deadlock Avoidance, Deadlock Detection, and Recovery from Deadlock

UNIT - III

Process Management and Synchronization - The Critical Section Problem, Synchronization Hardware, Semaphores, and Classical Problems of Synchronization, Critical Regions, Monitors

Interprocess Communication Mechanisms: IPC between processes on a single computer system, IPC between processes on different systems, using pipes, FIFOs, message queues, shared memory.

UNIT - IV

Memory Management and Virtual Memory - Logical versus Physical Address Space, Swapping, Contiguous Allocation, Paging, Segmentation, Segmentation with Paging, Demand Paging, Page Replacement, Page Replacement Algorithms.

UNIT - V

File System Interface and Operations -Access methods, Directory Structure, Protection, File System Structure, Allocation methods, Free-space Management. Usage of open, create, read, write, close, lseek, stat, ioctl system calls.

TEXT BOOKS:

1. Operating System Principles- Abraham Silberchatz, Peter B. Galvin, Greg Gagne 7th Edition, John Wiley.
2. Advanced programming in the UNIX environment, W.R. Stevens, Pearson education.

REFERENCE BOOKS:

1. Operating Systems- Internals and Design Principles, William Stallings, Fifth Edition–2005, Pearson Education/PHI
2. Operating System A Design Approach- Crowley, TMH.
3. Modern Operating Systems, Andrew S. Tanenbaum 2nd edition, Pearson/PHI
4. UNIX programming environment, Kernighan and Pike, PHI/ Pearson Education
5. UNIX Internals -The New Frontiers, U. Vahalia, Pearson Education.

UNIT -V

UNIT-5

FILE SYSTEM INTERFACE

FILE CONCEPT

A file is a named collection of related information that is recorded on secondary storage.

- A file is the smallest allotment of logical secondary storage (i.e.) data cannot be written to secondary storage unless they are within a file.
- Files represent programs and data. Data files may be numeric, alphanumeric or binary.
- The information in a file is defined by its creator.
- Different types of information may be stored in a file such as source or executable programs, numeric or text data, photos, music, video and so on.

A file structure depends on its type:

- **Text file** is a sequence of characters organized into lines.
- **Source file** is a sequence of functions, each of which is further organized as declarations followed by executable statements.
- **Executable file** is a series of code sections that the loader can bring into memory and execute.

File Attributes

A file is referred to by its name. The following are the list of file attributes:

- **Name**: The symbolic file name is the only information kept in human-readable form.
- **Identifier**: This is a unique number that identifies the file within the file system. It is the non-human-readable name for the file.
- **Type**: This information is needed for systems that support different types of files.
- **Location**: It is a pointer to a device and to the location of the file on that device.
- **Size**: The current size of the file and the maximum size are included in this attribute.
- **Protection**: It is access-control information determines who can do reading, writing, executing and so on.
- **Time, Date and User Identification**: This information kept for creation, last modification and last use. These data can be useful for protection, security and usage monitoring.

Directory structure keeps information about all files. It resides on secondary storage.

- A directory entry consists of the file's name and its unique identifier.
- The identifier locates the other file attributes.
- It may take more than a kilobyte to record this information for each file.

File Operations

There are 6 basic operations performed on file and corresponding System call are:

1. **Creating a file: create( )** system call is used to create a file. To create a file Operating system checks whether there is enough space in the system. If yes, then a new entry will be made in the directory structure.
2. **Repositioning within a file**. The directory is searched for the appropriate entry and the current-file-position pointer is repositioned to a given value. Repositioning within a file need not involve any actual I/O. This file operation is also known as a **File-Seek**.

3. **Deleting a file: delete( )** system call is used to delete a file. To deete a file, we search the directory for the named file. If we found the associated directory entry, we release all file space and erase the directory entry.
4. **Truncating a file**. The user erases all the contents of a file but keep its attributes. The length of the file will be reset to zero.
5. **Writing a file**. write( ) system call is used to write a file. It specifies both the name of the file and the information to be written to the file. The system searches the filename in the directory to find the file's location.
6. **Reading a file**. read( ) system call is used to read from a file. It specifies the name of the file and where the next block of the file should be put. The directory is searched for the associated entry.

Note: A process is usually either reading from or writing to a file, the current operation location can be kept as a per-process **Current-File-Position Pointer**. Both the read and write operations use this same pointer.

open( ) and close( ) system calls

- System calls open( ) and close( ) are used to open and to close a file respectively.
- **OS** maintains information about all open files in **Open-File Table.**
- When a file operation is requested, the file is indexed into Open File Table.
- When a file is closed by a process then the OS removes its entry from the open-file table.

Operating system uses **2-levels** of Internal tables:

1. **Per-Process Table** The per-process table tracks all files that a process has open. This table stores information regarding the process's use of the file. Each entry in the per-process table in turn points to a system-wide open-file table.
2. **System-Wide Table:** It contains process-independent information, such as the location of the file on disk, access dates and file size. Once a file has been opened by one process, the system-wide table includes an entry for the file. When another process executes an open( ) call, a new entry is added to the process's open-file table pointing to the appropriate entry in the system-wide table.

An open file is associated with following information:

- **File pointer:** This pointer is unique to each process operating on the file. It must be kept separate from the on-disk file attributes. 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:** The open-file table also has an **open count** associated with each file to indicate how many processes have opened that file. Each close( ) decreases the open count. When the open count reaches zero, the file is no longer in use and the file's entry is removed from the open-file table.
- **Disk location of the file**: Most file operations require the system to modify data several times within the file. The information needed to locate the file on disk is kept in main-memory so that the system does not have 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

- File types are generally included as part of file name. The file name is split into two parts: a name and an extension usually separated by a dot.
- The system uses the extension to indicate the type of the file and the type of operations that can be done on that file.

Example: resume.docx, server.c and ReaderThread.cpp.

The below table shows the common file types:

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

Internal File Structure

- Locating an offset within a file can be complicated for the operating system.
- Disk systems have a well-defined block size determined by the size of a sector.
- All disk I/O is performed in units of one block (physical record). All blocks are the same size.

Problem: It is unlikely that the physical record size will exactly match the length of the desired logical record. Logical records may have different lengths.

Solution: Packing a number of logical records into physical blocks solves this problem.

Example: The UNIX operating system defines all files to be streams of bytes.

Each byte is individually addressable by its offset from the beginning of the file.

- Logical record size, physical block size and packing technique determine how many logical records are in each physical block.
- Packing can be done either by the user’s application program or by the operating system.

Note: All file systems suffer from internal fragmentation. The larger the block size, the greater the internal fragmentation.

ACCESS METHODS

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

1. Sequential Access
2. Direct Access
3. Indexed Access

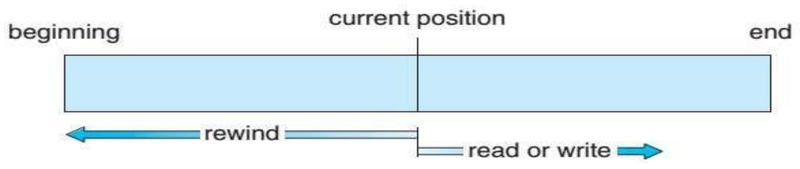
Sequential Access

Information in the file is processed in order, one record after the other record.

Example: editors and compilers usually access files in sequential order.

Reads and writes make up the bulk of operations on a file:

- **read_next( )** operation reads the next portion of the file and automatically advances a file pointer, which tracks the I/O location.
- **write_next( )** operation appends to the end of the file and advances to the end of the newly written material.



Sequential access is based on a tape model of a file and works on sequential-access devices.

Direct Access or Relative Access

- In direct access method, the file is viewed as a numbered sequence of blocks or records.
- There are no restrictions on the order of reading or writing for a direct-access file.
- Thus, we may read block 14, then read block 53 and then write block 7.
- 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.
- Databases are direct access type. When a query concerning a particular **subject** arrives, we compute which block contains the **answer** and then read that block directly to provide the desired information.

Example: Airline-reservation system

- We might store all the information about a particular flight 713 in the block identified by the flight number.
- 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 to determine a block to read and search.

File operation in Direct Access Method

read(n) and write(n) are the read and write operation performed in Direct Access method where **n** represent the block number.

The block number provided by the user to the operating system is a **Relative Block Number**.

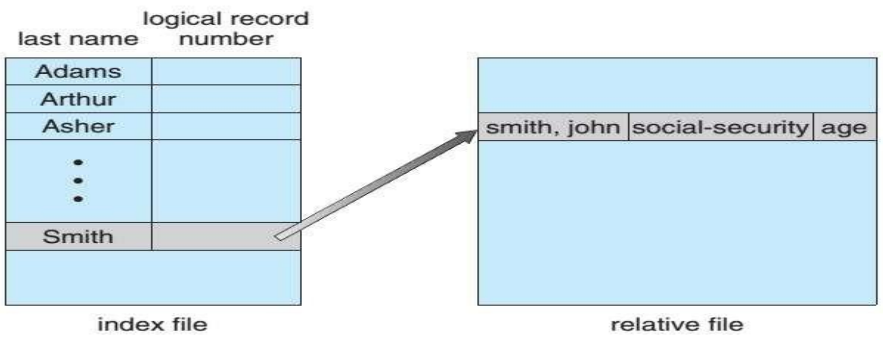
- A relative block number is an index relative to the beginning of the file.
- Thus, the first relative block of the file is 0, the next is 1 and so on.
- Relative block numbers allows the **OS** to decide where the file should be placed and helps to prevent user from accessing portions of the file system that may not be part of its file.

Indexed Access

The **index** is like an index in the back of a book that 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.

- To find a record we can make a binary search of the index. This search helps us to know exactly which block contains the desired record and access that block.
- This structure allows us to search a large file doing little I/O.
- With large files, the index file itself may become too large to be kept in memory.
- One solution is to create an index for the index file. The primary index file contains pointers to secondary index files, which point to the actual data items.

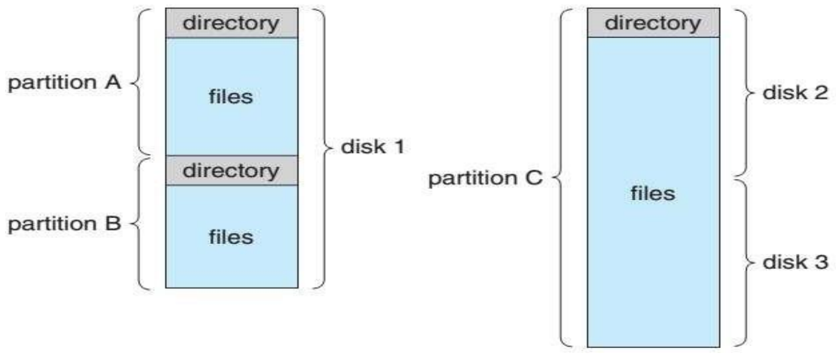


DIRECTORY AND DISK STRUCTURE

Files are stored on random-access storage devices such as Hard-disks, Optical-disks and Solid-state disks.

- A storage device can be used for a file system. It can be subdivided for finer-grained control.
- **Ex:** A disk can be **partitioned** into quarters. Each quarter can hold a separate file system.
- Partitioning is useful for limiting the sizes of individual file systems, putting multiple file-system types on the same device or leaving part of the device available for other uses, such as swap space or unformatted (raw) disk space.
- A file system can be created on each of these disk partitions. Any entity containing a file system is generally known as a **Volume**.
- Volume may be a subset of a device, a whole device. Each volume can be thought of as a virtual disk.
- Volumes can also store multiple operating systems. Volumes allow a system to boot and run more than one operating system.
- Each volume contains a file system maintains information about the files in the system.
- This information is kept in entries in a **Device directory** or **Volume table of contents**.
- The device directory (**directory**) records information such as name, location, size and type for all files on that volume.

The below figure shows the typical file system organization:



Storage Structure in Solaris OS

The file systems of computers can be extensive. Even within a file system, it is useful to segregate files into groups and manage those groups. This organization involves the use of directories.

Consider the types of file systems in the Solaris Operating system:

- **Tmpfs** is a ―temporary‖ file system that is created in volatile main memory and has its contents erased if the system reboots or crashes
- **objfs** is a ―virtual‖ file system that gives debuggers access to kernel symbols
- **ctfs** is a virtual file system that maintains ―contract‖ information to manage which processes start when the system boots and must continue to run during operation
- **lofs** is a ―loop back‖ file system that allows one file system to be accessed in place of another file system.
- **procfs** is a virtual file system that presents information on all processes as a file system
- **ufs, zfs** are general-purpose file systems.

Operations on Directory

Different operations performed on a directory are:

- **Search for a file**. This operation searches a directory structure to find the entry for a particular file. It finds all files whose names match with a particular pattern.
- **Create a file**. When a new file is created its entry is added to the directory.
- **Delete a file**. When a file is no longer needed, we can 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**. A file can be renamed, when the contents or use of the file changes (i.e.) csec.txt to cse.txt or cse.txt to cse.c file etc.
- **Traverse the file system**. We may wish to access every directory and every file within a directory structure.

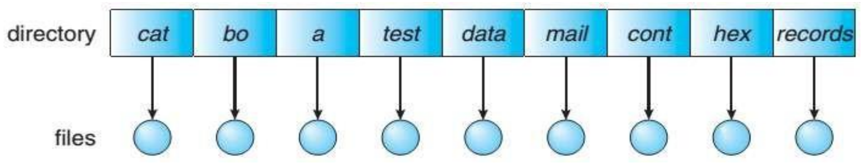
LOGICAL STRUCTURE OF A DIRECTORY

The different directory structures are:

1. Single Level Directory
2. Two-Level Directory
3. Tree Structured Directory
4. Acyclic-Graph Directories
5. General Graph Directory

Single-Level Directory

In Single-Level Directory structure, all files are contained in the same directory.

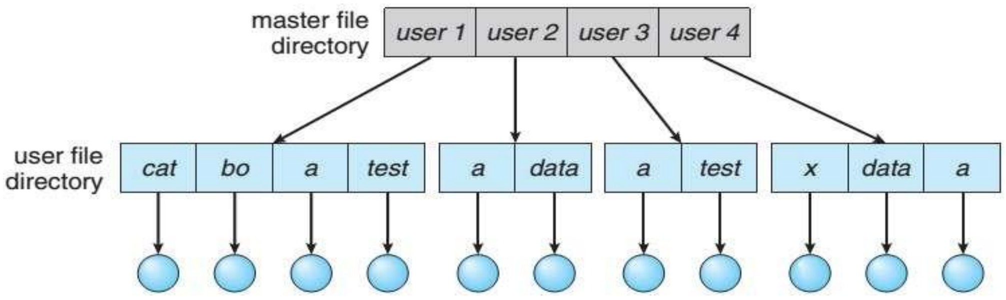


- A single-level directory has significant limitations that when the number of files increases or when the system has more than one users, all files are in the same directory, they must have unique names.

- Even a single user on a single-level directory may find it difficult to remember the names of all the files as the number of files increases.
- It is common for a user to have hundreds of files on one computer system.
- Keeping track of so many files is a difficult task.

Two-Level Directory

- In the two-level directory structure, each user has his own **user file directory (UFD)**.
- Each UFD 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. MFD is indexed by user name or account number and each entry points to the UFD for that user.



- When a user refers to a particular file, only his own UFD is searched. Thus, different users may have files with same name as long as all the file names within each UFD are unique.
- To create a file for a user, the operating system searches only that user's UFD to check whether another file of that name exists.
- To delete a file, the operating system confines its search to the local UFD; thus, it cannot accidentally delete another user's file that has the same name.
- This way the two-level directory structure solves the name-collision problem.
- To name a particular file uniquely in a two-level directory, we must give both the user name and the file name.

A two-level directory can be thought of as a tree or an inverted tree, of height 2.

- The root of the tree is the MFD.
- MFD's direct descendants are the UFDs.
- The descendants of the UFDs are the files.
- The files are the leaves of the tree.

Specifying a user name and a file name defines a path in the tree from the root (MFD) to a leaf (a file).

- A user name and a file name define a **path name**.
- Every file in the system has a path name.
- To name a file uniquely, a user must know the path name of desired file.

The user directories themselves must be created and deleted as necessary.

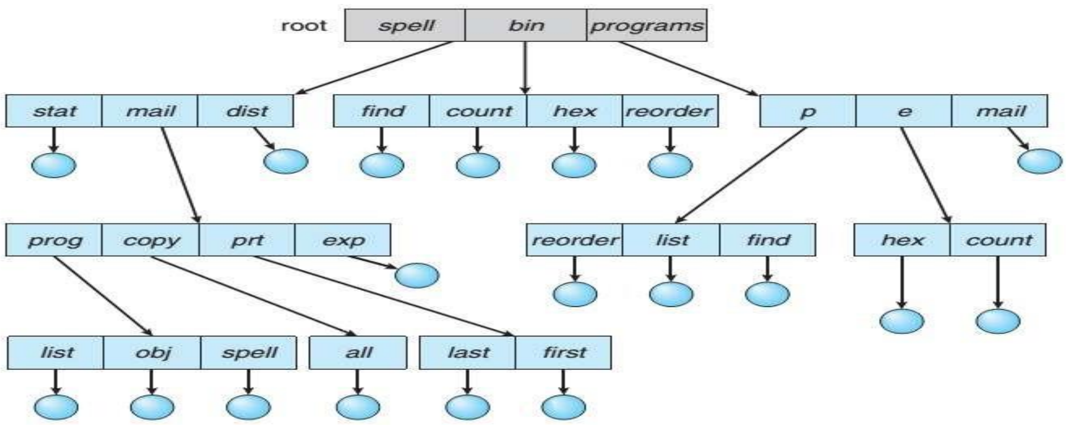
- A special system program is run with the appropriate user name and account information.
- The program creates a new UFD and adds an entry in the MFD.
- The execution of this program might be restricted to system administrators.

Disadvantages:

- This structure effectively isolates one user from another.
- Isolation is an advantage when the users are completely independent but it is a disadvantage when the users want to cooperate on some task and to access others files.
- Some systems simply do not allow one local user files to be accessed by other users.

<u>Tree-Structured Directories</u>

Tree Structure allows users to create their own subdirectories and to organize their files accordingly.



- The tree has a root directory and every file in the system has a unique path name.
- A directory (or) subdirectory contains a set of files or subdirectories.
- A directory is simply another file, but it is treated in a special way. All directories have the same internal format.
- One bit in each directory entry defines the entry as a file (0) or as a subdirectory (1).

Each process has a current directory. The **current directory** should contain most of the files that are of current interest to the process.

- When reference is made to a file, the current directory is searched.
- If a file is needed that is not in the current directory, then the user must specify a path name or change the current directory to the directory holding that file.
- To change directories, a system call is provided that takes a directory name as a parameter and uses it to redefine the current directory.
- Thus, the user can change the current directory whenever the user wants.

Path names can be of **2-types:** Absolute and Relative path name.

1. An **absolute path name** begins at the root and follows a path down to the specified file, giving the directory names on the path.
2. A **relative path name** defines a path from the current directory.

Example: If the current directory is **root/spell/mail** then the relative path name **prt/first** refers to the same file as does the absolute path name **root/spell/mail/prt/first.**

Deletion of Directory

If a directory is empty, its entry will be deleted form corresponding the directory.

If the directory to be deleted is not empty but contains several files or subdirectories then one of the two approaches can be followed:

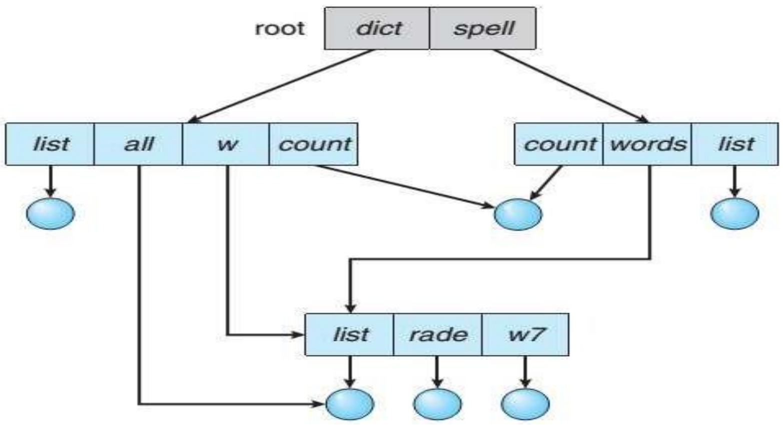
1. Some systems will not delete a directory unless it is empty. Thus, to delete a directory, the user must first delete all the files in that directory.
2. When a request is made to delete a directory, all the directory's files and subdirectories are also to be deleted. Example: UNIX rm command used for this purpose.

Note: With a tree-structured directory system, users can be allowed to access the files of other users. Example: user B can access a file of user A by specifying its path names.

Acyclic-Graph Directories

The acyclic graph is a natural generalization of the tree-structured directory scheme.

- A tree structure prohibits the sharing of files or directories.
- An **acyclic graph** is a graph with no cycles allows directories to share subdirectories and files. The same file or subdirectory may be in two different directories.
- With a shared file, only one actual file exists, so any changes made by one person are immediately visible to the other.
- Sharing is particularly important for subdirectories; a new file created by one person will automatically appear in all the shared subdirectories.
- UNIX implements Shared files and subdirectories by creating a new directory entry called a Link. A **link** is effectively a pointer to another file or subdirectory.



Example: A link may be implemented as an absolute or a relative path name.

- When a reference to a file is made, we search the directory.
- If the directory entry is marked as a link, then the name of the real file is included in the link information.
- We **resolve** the link by using that path name to locate the real file.
- Links are easily identified by their format in the directory entry and Links are effectively indirect pointers.
- The operating system ignores these links when traversing directory trees to preserve the acyclic structure of the system.

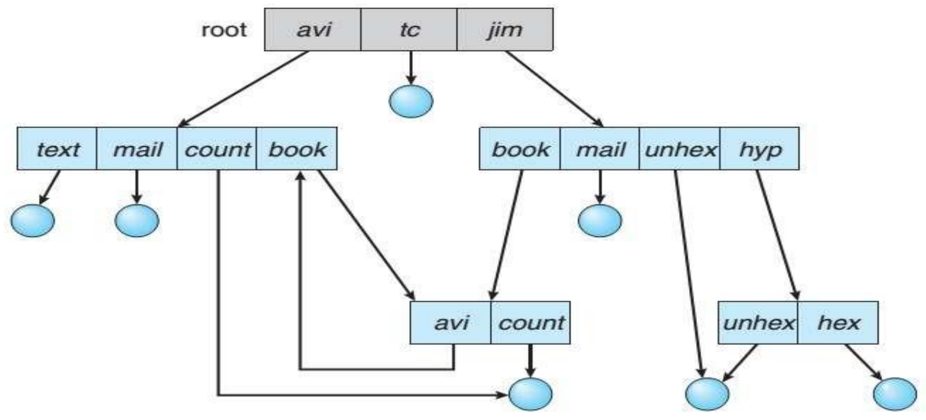
Problems with Acyclic-Graph Directories

1. A file may now have multiple absolute path names. Consequently, distinct file names may refer to the same file.
2. Deletion of shared file is problematic. Because more than one user is using the file if one user deletes a shared file, it may leave dangling pointers to non-existence file for other users.

General Graph Directory

General Graph Directory structure is an acyclic graph with Cycles.

- A problem with using an acyclic-graph structure is ensuring that there are no cycles.
- In tree-structure directory we can add new files and subdirectories to an existing tree-structured directory preserves the tree-structured nature but if we add links, the tree structure is destroyed, resulting in a simple graph structure.



- The primary advantage of an acyclic graph is the relative simplicity of the algorithms to traverse the graph and to determine when there are no more references to a file.
- If cycles are allowed to exist in the directory, we likewise want to avoid searching any component twice, for reasons of correctness as well as performance.
- **Problem:** A poorly designed algorithm might result in an infinite loop continually searching through the cycle and never terminating.
- **Solution:** we can limit arbitrarily the number of directories that will be accessed during a search.

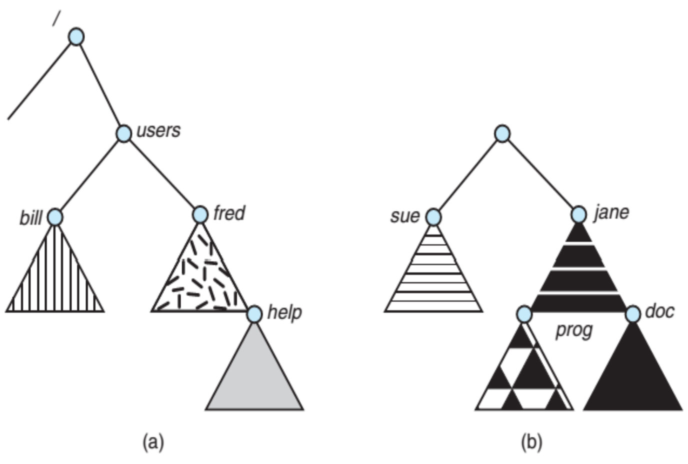
FILE-SYSTEM MOUNTING

A file system must be mounted before it can be available to processes on the system.

- The directory structure may be built out of multiple volumes, which must be mounted to make them available within the file-system name space.
- When a file system is mounting, the operating system is given the name of the device and the **Mount point.**
- The mount point is the location within the file structure where the file system is to be attached. In general a mount point is an empty directory.
- Example: On a UNIX system, a file system containing a user's **home** directories might be mounted as /home, then to access the directory structure within that file system, we could precede the directory names with /home, as in **/home/jane**.
- After mounting, the operating system verifies that the device contains a valid file system.
- Finally, the operating system notes in its directory structure that a file system is mounted at the specified mount point.

Consider the above file system, the triangles represent subtrees of directories.

- Figure (a) shows existing systems and Figure (b) shows Unmounted volume residing on /device/dsk.
- The last figure shows the mounting of the volume residing on **/device/dsk** over **/users**.



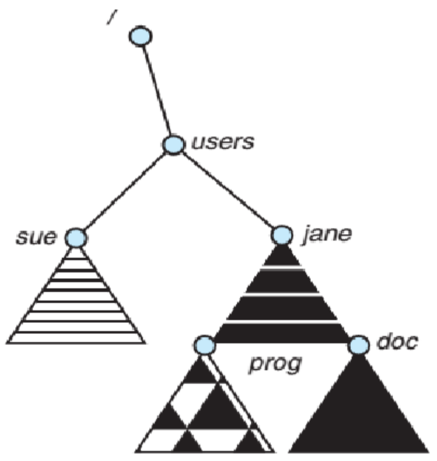


Figure 11.14 File system. (a) Existing system. (b) Unmounted volume. **Figure 11.15** Mount point.

Mounting in Windows Operating System

- The Microsoft Windows family of operating systems maintains an extended two-level directory structure, with devices and volumes assigned drive letters.
- Volumes have a general graph directory structure associated with the drive letter.
- The path to a specific file takes the form of **drive-letter:\path\to\file** (i.e. **F:\dir\f1.txt)**

PROTECTION

The information is stored on the computer system. Protection deals with issue of improper access of information to the illegitimate users.
Protection provides controlled access by limiting the types of file access that can be made.
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.
- Renaming, copying and editing the file, may also be controlled.

Access Control

- Different users may need different types of access to a file or directory.
- Systems uses A**ccess-control list (ACL) Scheme that** specifies user names 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.

Many systems recognize 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 work group.
- **Universe**: All other users in the system constitute the universe.

Protection in UNIX

In the UNIX system, groups can be created and modified only by the manager or super user.

- With the more limited protection classification, only three fields are needed to define protection.
- Each field is a collection of bits and each bit either allows or prevents the access associated with it.
- The UNIX system defines three fields of 3 bits each—rwx, where **r** controls read access, **w** controls write access and **x** controls execution.
- A separate field is kept for the file owner, for the file's group and for all other users.
- In this scheme, 9 bits per file are needed to record protection information.

FILE-SYSTEM STRUCTURE

File systems are maintained on Secondary Storage Disks.

Two reasons for storing file systems on disk are:

1. A disk can be rewritten in place (i.e.) It is possible to read a block from the disk, modify the block and write it back into the same place.
2. A disk can access directly any block of information it contains. Thus, it is simple to access any file either sequentially or randomly and switching from one file to another requires only moving the read–write heads and waiting for the disk to rotate.

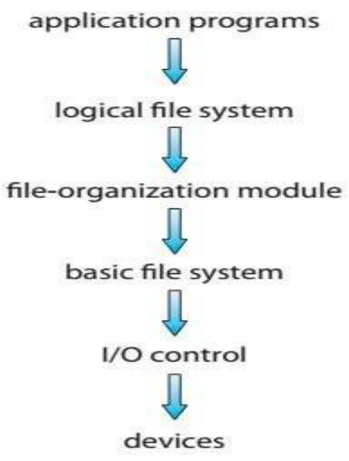
I/O transfers between memory and disk are performed in units of **blocks**.

- Each block has one or more sectors.
- Depending on the disk drive, sector size varies from 32 bytes to 4,096 bytes; the usual size is 512 bytes.
- **File systems** provide efficient and convenient access to the disk by allowing data to be stored, located and retrieved easily.

Design issues of File System

1. Defining how the file system should look to the user. This task involves defining a file and its attributes, the operations allowed on a file and the directory structure for organizing files.
2. Creating algorithms and data structures to map the logical file system onto the physical secondary-storage devices.

Layered Structured File System



- **I/O control** level consists of device drivers and interrupt handlers to transfer information between the main memory and the disk system.
- **Basic File System** needs only to issue generic commands to the appropriate device driver to read and write physical blocks on the disk.
- Each physical block is identified by its numeric disk address
- Example: drive 1, cylinder 73, track 2, sector 10.
- Basic file system layer also manages the memory buffers and caches that hold various file-system, directory and data blocks.
- F**ile-Organization Module** knows about files and their logical blocks, as well as physical blocks.
- The file-organization module includes the free-space manager, which tracks unallocated blocks and provides these blocks to the file-organization module when requested.
- **Logical File System** manages metadata information. Metadata includes all of the file-system structure except the actual data (or) contents of the files.
- Logical File System maintains file structure via File-Control Blocks. A **File-Control Block (FCB)** contains information about the file, including ownership, permissions and location of the file contents. In UNIX FCB is called as an **inode**.
- The logical file system is also responsible for protection.

Advantage: Layered structure minimizes the duplication of code. Code reusability is possible with this structure.

Disadvantage: Layering can introduce more operating-system overhead, which may result in decreased performance.

File systems supported by different Operating systems

1. **UNIX** uses the **UNIX File System (UFS)** is based on Berkeley Fast File System (FFS).
2. **Windows** supports disk file-system formats of **FAT**, **FAT32** and **NTFS** as well as CD-ROM and DVD file-system formats.
3. **Standard Linux file system** is known as the **Extended File System**, with the most common versions being ext3 and ext4.

FILE SYSTEM IMPLEMENTATION

Several on-disk and in-memory structures are used to implement a file system. These structures vary depending on the operating system and the file system.

On-Disk Structure

The file system may contain information about how to boot an operating system stored on disk, the total number of blocks, the number and location of free blocks, the directory structure and individual files.

Several On-Disk structure are given below:

Boot Control Block

- A Boot Control Block (per volume) can contain information needed by the system to boot an operating system from that volume.
- If the disk does not contain an operating system, this block can be empty.
- It is typically the first block of a volume.
- In UFS, it is called the **Boot Block**. In NTFS, it is the **Partition Boot Sector**.

Volume Control Block

- A Volume Control Block (per volume) contains volume (or) partition details such as the number of blocks in the partition, the size of the blocks, a free-block count and free-block pointers and a free-FCB count and FCB pointers.
- In UFS, this is called a **Super-Block**. In NTFS, it is stored in the **Master File Table**.

Directory Structure

- A directory structure (per file system) is used to organize the files.
- In UFS, this includes file names and associated inode numbers.
- In NTFS, it is stored in the master file table.

Per-File FCB

- A per-file FCB contains many details about the file.
- It has a unique identifier number to allow association with a directory entry.
- In NTFS, this information is actually stored within the master file table, which uses a relational database structure, with a row per file.

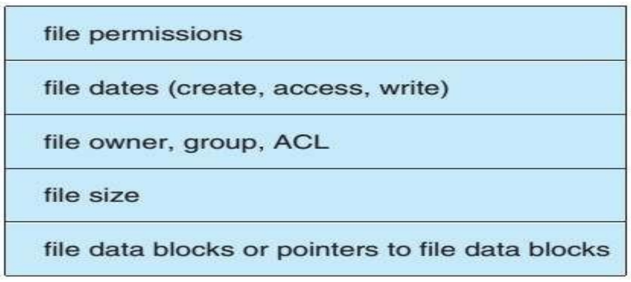
<u>In-Memory Structure</u>

The in-memory information is used for both file-system management and performance improvement via caching. The data are loaded at mount time, updated during file-system operations and discarded at dismount.

Several in-memory structures are given below:

- An in-memory **mount table** contains information about each mounted volume.
- An in-memory directory-structure cache holds the directory information of recently accessed directories. For directories at which volumes are mounted, it can contain a pointer to the volume table.
- The **system-wide open-file table** contains a copy of the FCB of each open file, as well as other information.
- The **per-process open-file table** contains a pointer to the appropriate entry in the system-wide open-file table, as well as other information.
- Buffers hold file-system blocks when they are being read from disk or written to disk.

The below figure shows the FCB

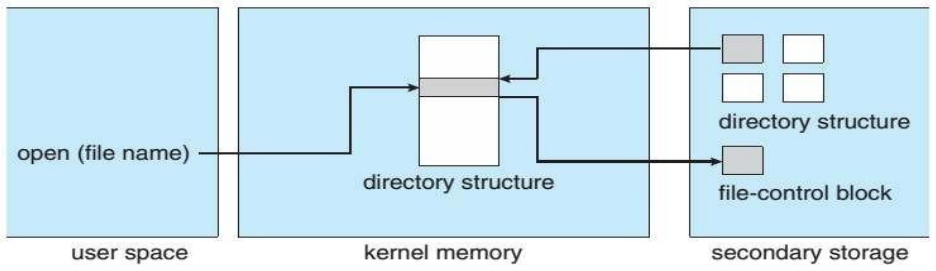


file permissions
file dates (create, access, write)
file owner, group, ACL
file size
file data blocks or pointers to file data blocks

- To create a new file, an application program calls the logical file system.
- The logical file system knows the format of the directory structures.
- To create a new file, it allocates a new FCB.
- The system then reads the appropriate directory into memory, updates it with the new file name and FCB and writes it back to the disk.

Process of Opening a file

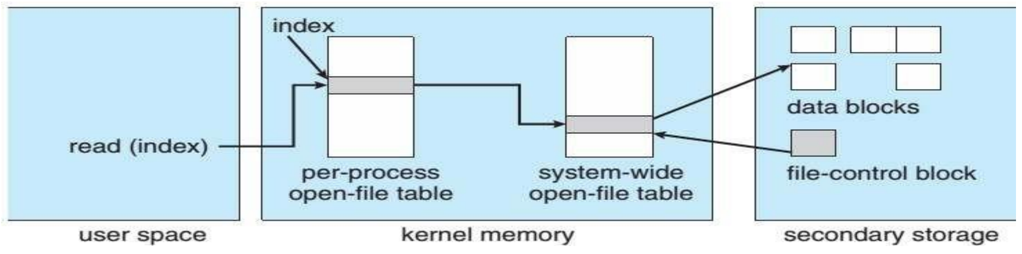
After a file has been created, it can be used for I/O.



- To open a file we use a system call open( ). The open( ) call passes a file name to the logical file system.
- The open( ) system call first searches the system-wide open-file table to see if the file is already in use by another process.
- If the file is open, a per-process open-file table entry is created pointing to the existing system-wide open-file table.
- If the file is not already open, the directory structure is searched for the given file name.
- Once the file is found, the FCB is copied into a system-wide open-file table in memory.
- This table not only stores the FCB but also tracks the number of processes that have the file open.

Process of Reading a File

- After an entry has been made in the per-process open-file table, with a pointer to the entry in the system-wide open-file table and some other fields.
- These other fields may include a pointer to the current location in the file for the next read( ) or write( ) operation and the access mode in which the file is open.



- The open( ) call returns a pointer to the appropriate entry in the per-process file-system table. All file operations are then performed via this pointer.
- The file name may not be part of the open-file table, as the system has no use for it once the appropriate FCB is located on disk.
- FCB could be cached to save time on subsequent opens of the same file. The name given to the entry varies.
- UNIX refers to FCB as a **File Descriptor**. Windows refers to FCB as a **File Handle.**

Process of Closing the File

- When a process closes the file, the per-process table entry is removed and the system-wide entry’s open count is decremented.
- When all users that have opened the file close it, any updated metadata is copied back to the disk-based directory structure and the system-wide open-file table entry is removed.

<u>Partitions and Mounting</u>

- A disk can be sliced into multiple partitions. A partition can be raw or cooked partition.
- A partition which does not contain any file system is called raw partition.

- A partition that contains a file system is called as cooked partition.
- UNIX swap space can use a raw partition, since it uses its own format on disk and does not use a file system.
- Boot information can be stored in a separate partition.
- It has its own format, because at boot time the system does not have the file-system code loaded and therefore cannot interpret the file-system format.
- Boot information is a sequential series of blocks, loaded as an image into memory.
- Execution of the image starts at a predefined location, such as the first byte.
- This **boot loader** knows about the file-system structure to be able to find and load the kernel and start it executing.
- It can contain more than the instructions for how to boot a specific operating system.
- Many systems can be **dual-booted**, allowing us to install multiple operating systems on a single system.
- A boot loader that understands multiple file systems and multiple operating systems can occupy the boot space.
- Once loaded, it can boot one of the operating systems available on the disk.
- The disk can have multiple partitions, each containing a different type of file system and a different operating system.
- **Root partition** contains the operating-system kernel. Sometimes other system files are mounted at boot time.
- Other volumes can be automatically mounted at boot or manually mounted later, depending on the operating system.

Example 1: Microsoft Windows System mounts each volume in a separate name space, denoted by a letter and a colon.

- To record that a file system is mounted at **F:**, the operating system places a pointer to the file system in a field of the device structure corresponding to **F:**.
- When a process specifies the driver letter, the operating system finds the appropriate file-system pointer and traverses the directory structures on that device to find the specified file or directory.
- Later versions of Windows can mount a file system at any point within the existing directory structure.

Example 2: In UNIX based systems, file systems can be mounted at any directory.

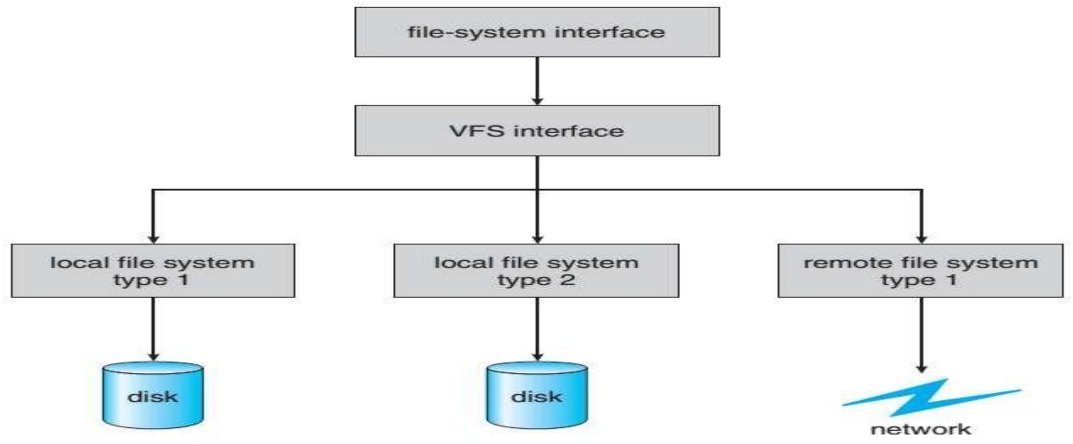
- Mounting is implemented by setting a flag in the in-memory copy of the inode for that directory.
- The flag indicates that the directory is a mount point. A field then points to an entry in the mount table, indicating which device is mounted there.
- Mount table entry contains a pointer to the superblock of the file system on that device.

<u>Virtual File Systems</u>

The first layer is the file-system interface, based on the open( ), read( ), write( ) and close( ) system calls and also based on file descriptors.

The second layer is called the **virtual file system (VFS)** layer. The VFS layer serves two important functions:

1. It separates file-system-generic operations from their implementation by defining a clean VFS interface. Several implementations for the VFS interface may coexist on the same machine, allowing transparent access to different types of file systems mounted locally.
2. It provides a mechanism for uniquely representing a file throughout a network. The VFS is based on a file-representation structure called a **vnode**, that contains a numerical designator for a network-wide unique file. This network-wide uniqueness is required for support of network file systems. The kernel maintains one vnode structure for each active node. A node may be a file or directory.



The VFS distinguishes local files from remote ones and local files are further distinguished according to their file-system types.

- VFS activates file-system-specific operations to handle local requests according to their file-system types and calls the Network File System (NFS) protocol procedures for remote requests.
- File handles are constructed from the relevant vnodes and are passed as arguments to these procedures.

The third layer implements the file-system type or the remote-file-system protocol.

DIRECTORY IMPLEMENTATION

1. Linear List
2. Hash Table

Linear List

- It maintains linear list of file names with pointers to the data blocks. It is time consuming.
- 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 new entry at the end of the directory.
- To delete a file, we search the directory for the named file and then release the space allocated to it.
- To reuse the directory entry either we can mark the entry as unused or we can attach it to a list of free directory entries.

Disadvantage: It uses linear search to find a file. Linear search is very slow.

Hash Table

- The hash table takes a value computed from the file name and returns a pointer to the file name in the linear list. It decreases the directory search time.
- Insertion and deletion are also very easy to implement.

The major difficulty hash tables are its generally fixed size and Hash tables are dependent on hash function on that size.
Example: Assume that we make a linear-probing hash table that holds 64 entries.

- The hash function converts file names into integers from 0 to 63.
- If we try to create a 65th file, we must enlarge the directory hash table to 128 entries.
- Hence we need a new hash function that must map file names to the range 0 to 127 and must reorganize the existing directory entries to reflect their new hash-function values.

ALLOCATION METHODS

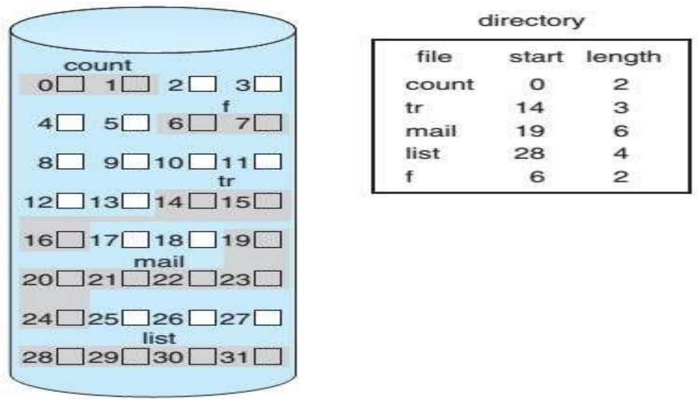
Three major methods of allocating disk space are in wide use:

1. Contiguous Allocation
2. Linked Allocation
3. Indexed Allocation

Contiguous Allocation

Contiguous allocation requires that each file occupy a set of contiguous blocks on the disk.

- Disk addresses define a linear ordering on the disk. With this ordering, assuming that only one job is accessing the disk, accessing block *b* + 1 after block *b* normally requires no head movement.
- When head movement is needed the head need only move from one track to the next.
- Contiguous allocation of a file is defined by the disk address and length (in block units) of the first block.
- If the file is *n* blocks long and starts at location *b*, then it occupies blocks ***b*, *b* + 1, *b* + 2, ..., *b* + *n* − 1**.
- The directory entry for each file indicates the address of the starting block and the length of the area allocated for this file.



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

Accessing a file that has been allocated contiguously is easy.

- For sequential access, the file system remembers the disk address of the last block referenced and when necessary, reads the next block.
- For direct access to block ***i*** of a file that starts at block ***b***, it can immediately access block **b+i**.
- Both sequential and direct access can be supported by contiguous allocation.

Two-Problems with Contiguous Allocation:

1. Finding space for a new file
2. Determining how much space is needed for a file.

Finding space for a new file

- The contiguous-allocation problem occurs in dynamic storage-allocation that involves how to satisfy a request of size *n* from a list of free holes.
- First fit and best fit are the most common strategies used to select a free hole from the set of available holes.
- Both, First fit and Best fit algorithms suffer from the problem of **external fragmentation**.
- As files are allocated and deleted, the free disk space is broken into little pieces.
- External fragmentation exists whenever free space is broken into chunks.
- It becomes a problem when the largest contiguous chunk is insufficient for a request.
- The storage is fragmented into a number of holes, none of which is large enough to store the data.

One solution for this problem is **Compaction:**

- Compaction solves external fragmentation by coping an entire file system onto another disk.
- The original disk is then freed completely, creating one large contiguous free space.
- We then copy the files back onto the original disk by allocating contiguous space from this one large hole.
- The cost of compaction is very high when the size of the hard disk is huge. The time taken for compaction will be high as the size of the hard disk increases.

Determining how much space is needed for a file

- When the file is created, the total amount of space it will need must be found and allocated.
- If we allocate too little space to a file, we may find that the file cannot be extended.
- Especially with a best-fit allocation strategy, the space on both sides of the file may be in use. Hence, we cannot make the file larger in place.

Two possibilities then exist.

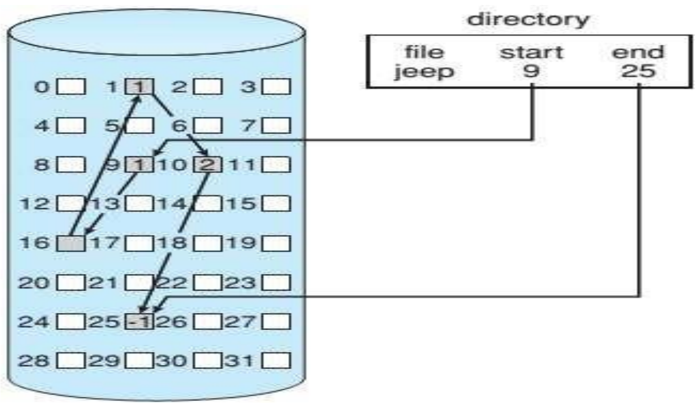
- First, the user program can be terminated, with an appropriate error message.
- The user must then allocate more space and run the program again.
- These repeated runs may be costly.
- To prevent them, the user will normally overestimate the amount of space needed, resulting in considerable wasted space.
- The other possibility is to find a larger hole, copy the contents of the file to the new space and release the previous space.
- All these are time consuming and system performance will be effected.

<u>**Linked Allocation**</u>

Linked allocation solves all problems of contiguous allocation.

- With linked allocation, each file is a linked list of disk blocks.
- Disk blocks are scattered anywhere on the disk.
- The directory contains a pointer to the first and last blocks of the file.

Consider the below figure that shows linked list allocation:



- A file of five blocks might start at block 9 and continue at block 16, then block 1, then block 10 and finally block 25.
- Each block contains a pointer to the next block.
- These pointers are not made available to the user.
- Thus, if each block is 512 bytes in size and a disk address (the pointer) requires 4 bytes, then the user sees blocks of 508 bytes.

Advantage:

Linked List allocation avoids Compaction

- To create a new file, we simply create a new entry in the directory.
- With linked allocation, each directory entry has a pointer to the first disk block of the file.
- This pointer is initialized to null (the end-of-list pointer value) to signify an empty file. The size field is also set to 0.
- A write to the file causes the free-space management system to find a free block and this new block is written to and is linked to the end of the file.
- To read a file, we simply read blocks by following the pointers from block to block.
- There is no external fragmentation with linked allocation and any free block on the free-space list can be used to satisfy a request.
- The size of a file need not be declared when the file is created. A file can continue to grow as long as free blocks are available. Hence by Linked List allocation avoids external fragmentation and it avoid need for compact disk space.

Disadvantages:

1. It is inefficient for Direct Access
2. Space for Pointers
3. Reliability

Linked list allocation can be used effectively only for sequential-access files.

- To find the **i**th block of a file, we must start at the beginning of that file and follow the pointers until we get to the **i**th block.
- Each access to a pointer requires a disk read and some require a disk seek.
- It is inefficient to support a direct-access capability for linked-allocation files.

Another disadvantage is the space required for the pointers.

- If a pointer requires 4 bytes out of a 512-byte block, then 0.78 percent of the disk is being used for pointers, rather than for information.
- Each file requires slightly more space than it would otherwise.

Solutions to above problems: **Clustering**

- Cluster is a collection multiple blocks and we allocate clusters rather than blocks.
- Let the file system define a cluster as four blocks and operate on the disk only in cluster units. Pointers then use a much smaller percentage of the file's disk space.
- This method improves disk throughput and decreases the space needed for block allocation and free-list management.
- Clustering approach leads to the problem of internal fragmentation, because more space is wasted when a cluster is partially full than when a block is partially full.

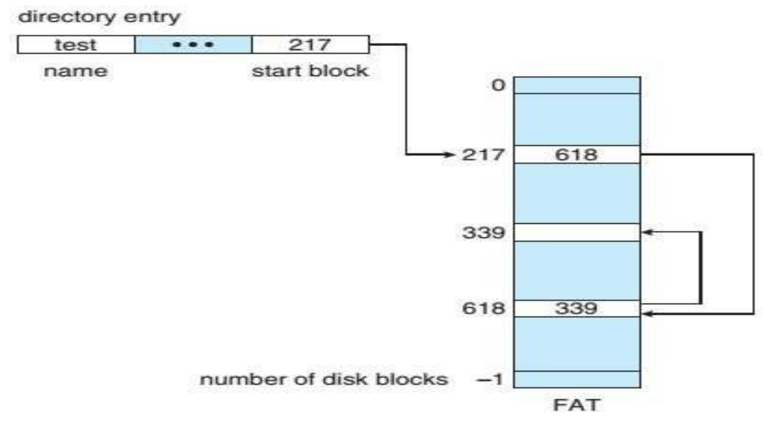
Reliability issues will be arised

- The files are linked together by pointers scattered all over the disk.
- If a pointer were lost or damaged, this might result in picking up the wrong pointer.
- This error could in turn result in linking into the free-space list or into another file.

<u>FILE ALLOCATION TABLE (FAT)</u>

Linked List allocation uses File Allocation Table.

- FAT is very efficient method of disk-space allocation. It was used by the MS-DOS operating system.
- A section of disk at the beginning of each volume contains the table.
- The table has one entry for each disk block and is indexed by block number.
- The directory entry contains the block number of the first block of the file.
- The table entry indexed by that block number contains the block number of the next block in the file.
- This chain continues until it reaches the last block, which has a special end-of-file value as the table entry.
- An unused block is indicated by a table value of 0. Allocating a new block to a file is to find the first 0-valued table entry and replacing the previous end-of-file value with the address of the new block. The 0 is then replaced with the end-of-file value.

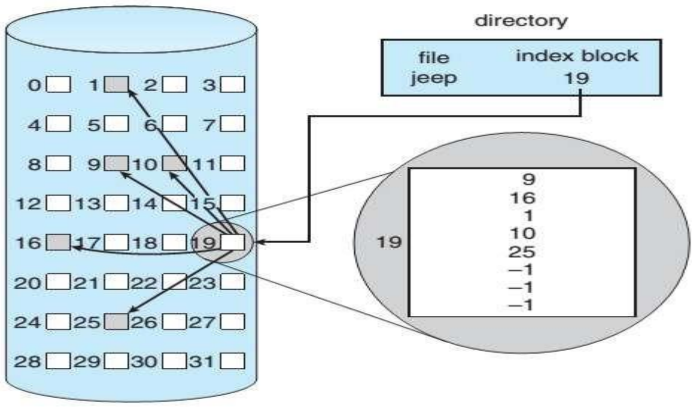


- The FAT allocation scheme can result in a significant number of disk head seeks, unless the FAT is cached.
- The disk head must move to the start of the volume to read the FAT and find the location of the block in question, then move to the location of the block itself.
- A benefit is that random-access time is improved, because the disk head can find the location of any block by reading the information in the FAT.

Indexed Allocation

In Indexed allocation, each file has its own index block. An index block is an array of disk-block addresses.

- The **i**th entry in the index block points to the **i**th block of the file.
- The directory contains the address of the index block.
- To find and read the **i**th block, we use the pointer in the **i**th index-block entry.
- When the file is created, all pointers in the index block are set to null.
- When the **i**th block is first written, a block is obtained from the free-space manager and its address is put in the **i**th index-block entry.
- Indexed allocation supports direct access, without suffering from external fragmentation, because any free block on the disk can satisfy a request for more space.



Indexed allocation suffers from wasted space and Pointer Overhead.

- The pointer overhead of the index block is greater than the pointer overhead of linked allocation.
- Consider we have a file of only one or two blocks.
- With linked allocation, we lose the space of only one pointer per block.
- With indexed allocation, an entire index block must be allocated, even if only one or two pointers will be non-null.

Determining size of the index block is a big issue in Indexed allocation. Several mechanisms are used for this purpose are:

1. Linked Scheme
2. Multilevel Index
3. Combined Scheme

Linked Scheme

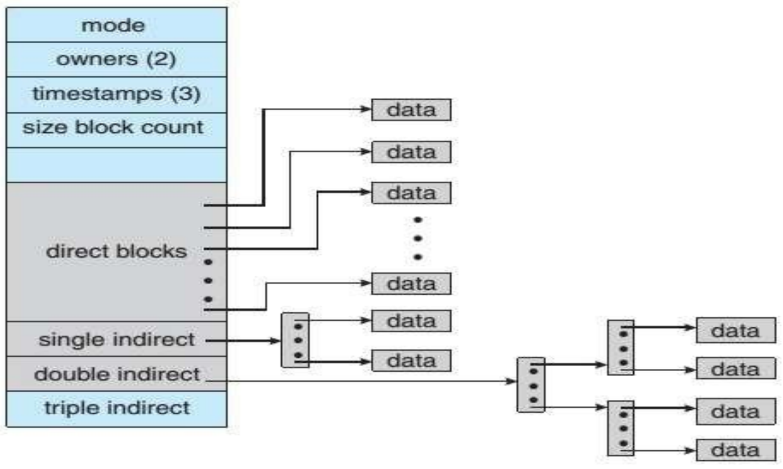
- An index block is normally one disk block. Thus, it can be read and written directly by itself.
- To allow for large files, we can link together several index blocks.
- **Example:** An index block might contain a small header giving the name of the file and a set of the first 100 disk-block addresses.
- The next address (i.e.) the last word in the index block is null (for a small file) or is a pointer to another index block (for a large file).

Multilevel index

- A variant of linked representation uses a first-level index block to point to a set of second-level index blocks, which in turn point to the file blocks.
- To access a block, the operating system uses the first-level index to find a second-level index block and then uses that block to find the desired data block.
- This approach could be continued to a third or fourth level, depending on the desired maximum file size.
- With 4,096-byte blocks, we could store 1,024 four-byte pointers in an index block.
- Two levels of indexes allow 1,048,576 data blocks and a file size of up to 4 GB.

Combined Scheme

- It is used by the UNIX based file system that keeps the first 15 pointers of the index block in the file's inode.
- The first 12 of these pointers point to **direct blocks** (i.e.) they contain addresses of blocks that contain data of the file.
- Thus, the data for small files of no more than 12 blocks do not need a separate index block.



- If the block size is 4 KB, then up to 48 KB of data can be accessed directly.
- The next three pointers point to **Indirect blocks**.
- The first points to a **Single indirect block**, which is an index block containing not data but the addresses of blocks that do contain data.
- The second points to a **Double indirect block**, which contains the address of a block that contains the addresses of blocks that contain pointers to the actual data blocks.
- The last pointer contains the address of a **Triple indirect block**.

FREE-SPACE MANAGEMENT

- Since disk space is limited, we need to reuse the space from deleted files for new files.
- To keep track of free disk space, the system maintains a **Free-Space List**.
- Free-space list records all free disk blocks, those not allocated to some file or directory.
- To create a file, we search the free-space list for the required amount of space and allocate that space to the new file. This space is then removed from the free-space list.
- When a file is deleted, its disk space is added to the free-space list.

The free space can be managed in several ways:

1. Bit Vector
2. Linked List
3. Grouping
4. Counting
5. Space Maps

Bit Vector

The free-space list is implemented as a **Bit map** or **Bit vector**.

Each block is represented by one bit, the bit 1 represents block is free and bit 0 represents block is allocated.

Example: Consider a disk where blocks 2, 3, 4, 5, 8, 9, 10, 11, 12, 13, 17, 18, 25, 26 and 27 are free and the rest of the blocks are allocated. The free-space bit map would be

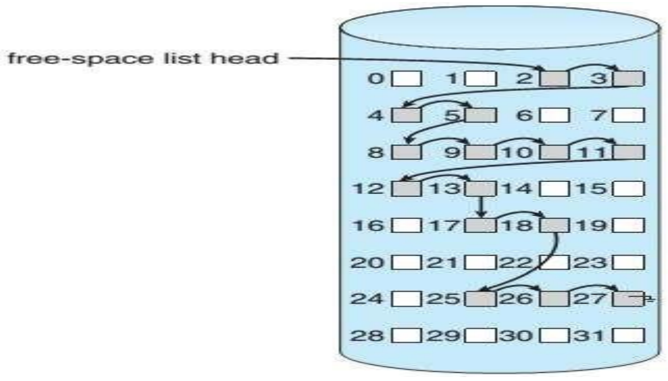
001111001111110001100000011100000

Advantage: Its relative simplicity and its efficiency in finding the first free block or *n* consecutive free blocks on the disk.

Disadvantage: Bit Vectors are kept in main memory is possible for smaller disks. For larger disks it is not efficient to keep it in Main memory because A 1-TB disk with 4-KB blocks requires 256 MB to store its bit map. So, as the disk size increases, the bit vector size is also increases.

Linked List

All the free disk blocks are linked together by keeping a pointer to the first free block in a special location on the disk and caching it in memory. This first block contains a pointer to the next free disk block and so on.



- Consider the above figure, that shows the set of free blocks 2, 3, 4, 5, 8, 9, 10, 11, 12, 13, 17, 18, 25, 26 and 27.
- The system would keep a pointer to block 2 as the first free block. Block 2 would contain a pointer to block 3 and so on.
- This scheme is not efficient; to traverse the list, we must read each block, which requires substantial I/O time.

Grouping

- It stores the addresses of *n* free blocks in the first free block.
- The first *n*−1 of these blocks are free blocks and the last block contains the addresses of another *n* free blocks and so on.
- Addresses of a large number of free blocks can be found quickly than linked-list method.

<u>Counting</u>

- When space is allocated with the contiguous-allocation algorithm or through clustering, several contiguous blocks may be allocated or freed simultaneously.
- Here we keep the address of the first free block and the number (*n*) of free contiguous blocks that follow the first block.
- Each entry in the free-space list then consists of a disk address and a count. Hence the overall disk entries are small.

<u>Space Maps</u>

Oracle's **ZFS** file system was designed to encompass huge numbers of files, directories and even file systems.

- In its management of free space, ZFS uses a combination of techniques to control the size of data structures and minimize the I/O needed to manage those structures.
- ZFS creates **meta-slabs** to divide the space on the device into chunks of manageable size.
- A volume contains hundreds of meta-slabs. Each meta-slab has an associated space map.
- ZFS uses the counting algorithm to store information about free blocks. It uses log-structured file-system techniques to record them.
- The space map is a log of all block activity such as allocating and freeing, in time order and in counting format.
- When ZFS decides to allocate or free space from a meta-slab, it loads the associated space map into memory in a balanced-tree structure (for very efficient operation), indexed by offset and replays the log into that structure.
- The in-memory space map is then an accurate representation of the allocated and free space in the meta-slab.
- ZFS also condenses the map as much as possible by combining contiguous free blocks into a single entry.
- Finally, the free-space list is updated on disk as part of the transaction-oriented operations of ZFS.
- During the collection and sorting phase, block requests can still occur and ZFS satisfies these requests from the log. In essence, the log plus the balanced tree is the free list

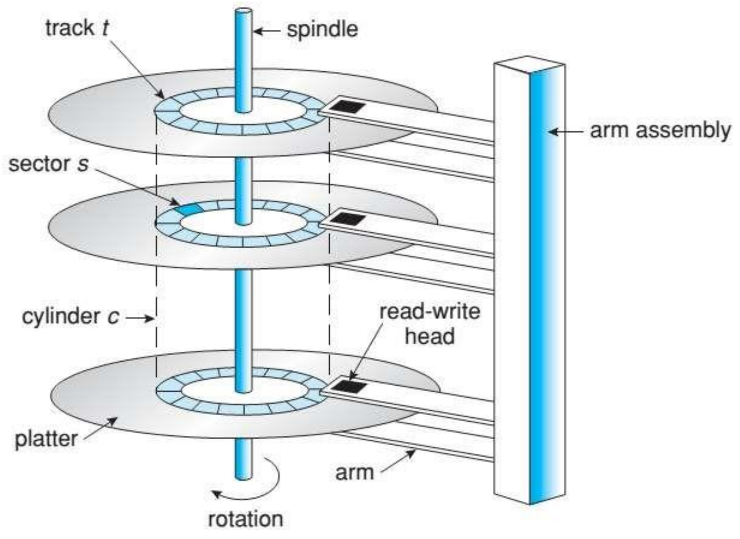
UNIT-5

MASS-STORAGE STRUCTURE

MAGNETIC DISK

Magnetic disks provide the bulk of secondary storage for modern computer systems.

- Each disk **platter** has a flat circular shape, like a CD.
- Common platter diameters range from 1.8 to 3.5 inches.
- The two surfaces of a platter are covered with a magnetic material.
- We store information by recording it magnetically on the platters.
- A read–write head –flies‖ just above each surface of every platter.
- The heads are attached to a **Disk arm** that moves all the heads as a unit.
- The surface of a platter is logically divided into circular **Tracks**.
- The tracks are subdivided into **Sectors**. Each track may contain hundreds of sectors.
- The set of tracks that are at one arm position makes up a **Cylinder**.
- There may be thousands of concentric cylinders in a disk drive.
- The storage capacity of common disk drives is measured in Giga-bytes.
- When the disk is in use, a drive motor spins it at high speed. Common drives spin at 5,400, 7,200, 10,000 and 15,000 RPM.



Transfer rate is the rate at which data flow between the drive and the computer.
The **Positioning time** or **Random-access time** consists of two parts:

- Seek time: It is the time necessary to move the disk arm to the desired cylinder.
- Rotational Latency: It is the time necessary for the desired sector to rotate to the disk head.

Typical disks can transfer several megabytes of data per second and they have seek times and rotational latencies of several milliseconds.

Head crash

The disk head flies on an extremely thin cushion of air, there is a danger that the head will make contact with the disk surface and damage the magnetic surface is called Head Crash.
A head crash cannot be repaired. The entire disk must be replaced.

Removable Disks

- A disk can be **removable**, allowing different disks to be mounted as needed.
- Removable magnetic disks consist of one platter, held in a plastic case to prevent damage while not in the disk drive.
- Other forms of removable disks include CDs, DVDs and Blu-ray discs removable flash-memory devices known as **flash drives.**

A disk drive is attached to a computer by a set of wires called an **I/O bus**. Several kinds of buses are available, including **advanced technology attachment (ATA)**, **serial ATA (SATA)**, **eSATA**, **universal serial bus (USB)** and **fibre channel (FC)**.

SOLID-STATE DISKS (SSD)

- SSD is nonvolatile memory that is used like a hard drive.
- SSDs are more reliable because they have no moving parts.
- SSDs are faster because they have no seek time or latency.
- SSDs can be much faster than magnetic disk drives.
- By comparing Hard disk, SSDs consumes less power but SSDs are more expensive per megabyte, have less capacity and may have shorter life spans than hard disks.

MAGNETIC TAPES

- **Magnetic tape** was used as an early secondary-storage medium.
- Tapes relatively permanent and can hold large quantities of data.
- Magnetic Tape access time is slow compared with that of main memory and magnetic disk. Random access to magnetic tape is about a thousand times slower than random access to magnetic disk, so tapes are not very useful for secondary storage.
- Tapes are used mainly for backup, for storage of infrequently used information and as a medium for transferring information from one system to another system.
- A tape is kept in a spool and it is wound or rewound past a read–write head.
- Moving to the correct spot on a tape can take minutes, but once positioned, tape drives can write data at speeds comparable to disk drives.
- Tape capacity is up to several terabytes.

DISK STRUCTURE

Magnetic disk drives are addressed as large one-dimensional arrays of **logical blocks**.

- The Logical block is the smallest unit of transfer.
- Size of the Logical block is 512 Bytes or 1024 Bytes.
- The one-dimensional array of logical blocks is mapped onto the sectors of the disk sequentially. Sector 0 is the first sector of the first track on the outermost cylinder.
- The mapping proceeds in order through that track, then through the rest of the tracks in that cylinder and then through the rest of the cylinders from outermost to innermost.
- By using this mapping, we can—at least in theory—convert a logical block number into an old-style disk address that consists of a cylinder number, a track number within that cylinder and a sector number within that track.

The number of sectors per track is not a constant on some drives.

- The track which is far from the center of the disk, the length of the track is more and this track can hold more sector than the track that is nearer to the center of disk,
- Tracks in the outermost zone typically hold 40 percent more sectors than do tracks in the innermost zone.
- The drive increases its rotation speed as the head moves from the outer to the inner tracks to keep the same rate of data moving under the head.
- This method is used in CD-ROM and DVD-ROM drives.

DISK SCHEDULING ALGORITHMS

One of the responsibilities of the operating system is to use the hardware efficiently.
The disk **bandwidth** is the total number of bytes transferred, divided by the total time between the first request for service and the completion of the last transfer.
Different Disk Scheduling algorithms are:

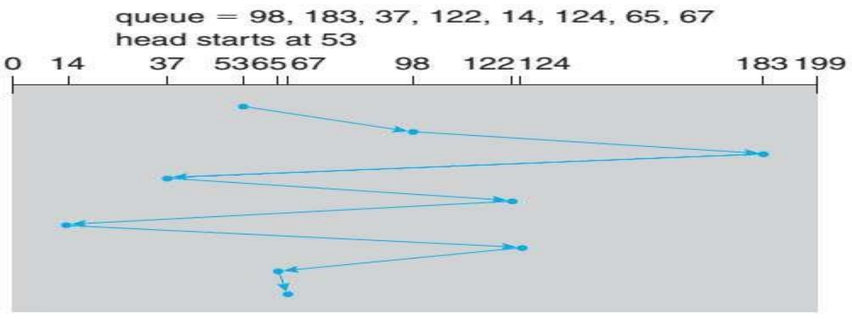
1. FCFS Scheduling
2. SSTF Scheduling
3. SCAN Scheduling
4. C- SCAN Scheduling
5. LOOK Scheduling

First-Come-First-Serve Algorithm

FCFS does not provide the fastest service.
Consider a disk queue with requests for I/O to blocks on cylinders in the order:

98, 183, 37, 122, 14, 124, 65, 67

The disk head is initially at cylinder **53**. By using FCFS algorithm:



- It will first move from 53 to 98 the head movement of 45 cylinders
- Then 98 to 183 the head movement of 85 cylinders
- Then 183 to 37 the head movement of 146 cylinders
- Then 37 to 122 the head movement of 85 cylinders
- Then 122 to 14 the head movement of 108 cylinders
- Then 14 to 124 the head movement of 110 cylinders
- Then 124 to 65 the head movement of 59 cylinders
- Then 65 to 67 the head movement of 2 cylinders
- The total head movement of 640 cylinders.

FCFS algorithm reduces the system performance.

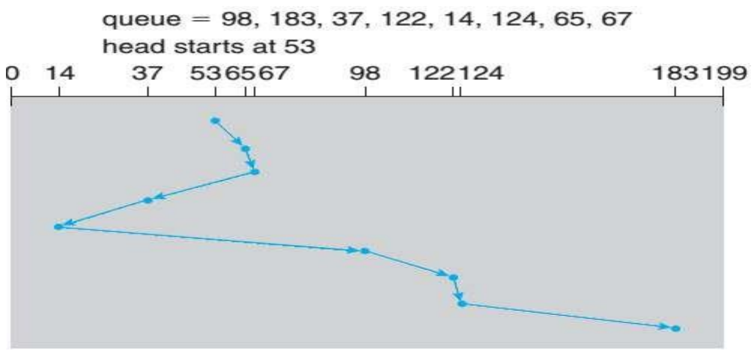
SSTF Scheduling

Shortest-Seek-Time-First (SSTF) algorithm service all the requests close to the current head position before moving the head far away to service other requests.

The SSTF algorithm selects the request with the least seek time from the current head position. (i.e.) SSTF chooses the pending request closest to the current head position.
Consider a disk queue with requests for I/O to blocks on cylinders in the order:

98, 183, 37, 122, 14, 124, 65, 67

The disk head is initially at cylinder **53**.



Closest request to the initial head position 53 is at cylinder 65 takes 12 cylinders movements.

- Once we are at cylinder 65, the next closest request is at cylinder 67 (2 moves).
- From 67, the request at cylinder 37 is closer than the one at 98, so 37 is served next.
- Similarly we service the request at cylinder 14, then 98, 122, 124 and finally 183.
- This scheduling method results in a total head movement of only 236 cylinders.

The performance of SSTF is better than FCFS but SSTF causes starvation of some requests.

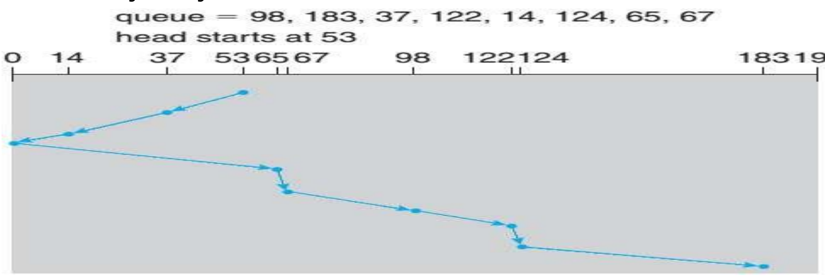
- Suppose that we have two requests in the queue, for cylinders 14 and 186 and while the request from 14 is being serviced, a new request 30 near 14 arrives.
- This new request 30 will be serviced next, making the request at 186 wait.
- While request 30 is being serviced, another request close to 30 could arrive.
- A continual stream of requests near one another could cause the request for cylinder 186 to wait indefinitely.

<u>SCAN algorithm</u>

In the **SCAN algorithm**, the disk arm starts at one end of the disk and moves toward the other end, servicing requests as it reaches each cylinder, until it gets to the other end of the disk. At the other end, the direction of head movement is reversed and servicing continues.
The head continuously scans back and forth across the disk. The SCAN algorithm is also called as the **Elevator algorithm**, since the disk arm behaves just like an elevator in a building, first servicing all the requests going up and then reversing to service requests the other way.
Consider a disk queue with requests for I/O to blocks on cylinders in the order:

98, 183, 37, 122, 14, 124, 65, 67

The disk head is initially at cylinder **53**.



Before applying SCAN algorithm we need to know the the direction of head movement in addition to the head's current position.

- Assuming that the disk arm is moving toward 0 and that the initial head position is again 53 the head will next service 37 and then 14.
- At cylinder 0, the arm will reverse and will move toward the other end of the disk, servicing the requests at 65, 67, 98, 122, 124 and 183.
- A request arrives in the queue in front of the head, it will be serviced almost immediately.
- A request arriving just behind the head will have to wait until the arm moves to the end of the disk, reverses direction and comes back.

C-SCAN Scheduling

Circular SCAN (C-SCAN) scheduling is a variant of SCAN designed to provide a more uniform wait time.

Like SCAN, C-SCAN moves the head from one end of the disk to the other, servicing requests along the way.

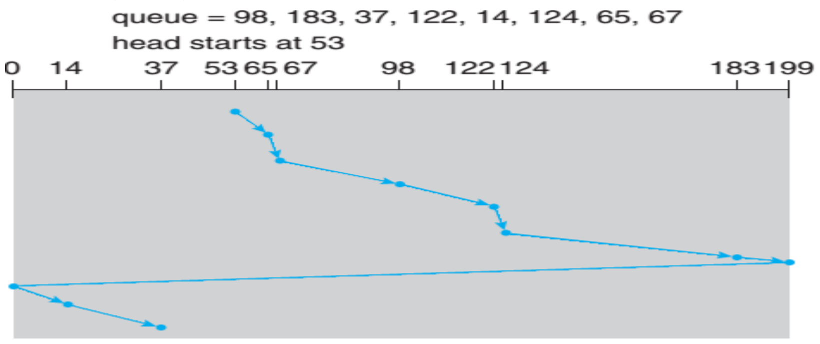
When the head reaches the other end, it immediately returns to the beginning of the disk without servicing any requests on the return trip.

The C-SCAN scheduling algorithm essentially treats the cylinders as a circular list that wraps around from the final cylinder to the first one.

Consider a disk queue with requests for I/O to blocks on cylinders in the order:

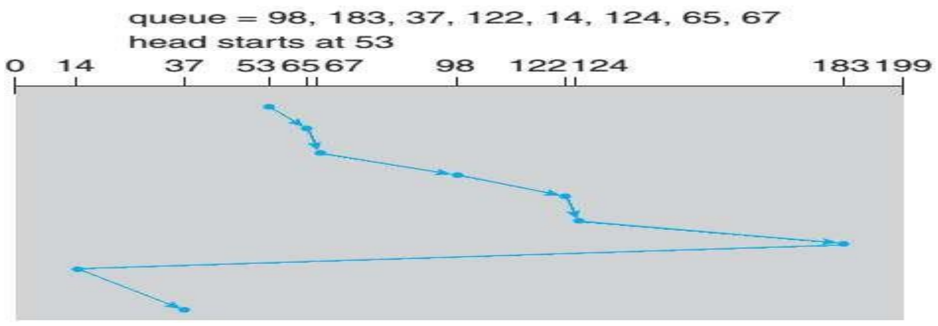
98, 183, 37, 122, 14, 124, 65, 67

The disk head is initially at cylinder **53**.



LOOK Scheduling

- Both SCAN and C-SCAN move the disk arm across the full width of the disk, neither of these algorithms are implemented.
- In LOOK and C-LOOK scheduling, the arm goes only as far as the final request in each direction. Then, it reverses direction immediately, without going all the way to the end of the disk. (i.e.) they ***look*** for a request before continuing to move in a given direction.



DISK MANAGEMENT

The operating system is responsible for several other aspects of disk management such as:

1. Disk Formatting
2. Boot Block
3. Bad Blocks

Disk Formatting

A new magnetic disk is an empty disk. Before a disk can store data, it must be divided into sectors that the disk controller can read and write. This process is called **Low-Level Formatting** or **Physical Formatting**.

- Low-level formatting fills the disk with a special data structure for each sector.
- The data structure for a sector consists of a header, a data area and a trailer. The data are is of 512 bytes.
- The header and trailer contain information used by the disk controller, such as a sector number and an **error-correcting code (ECC)**.
- When the controller writes a sector of data during normal I/O, the ECC is updated with a value calculated from all the bytes in the data area.
- When the sector is read, the ECC is recalculated and compared with the stored value.
- If the stored and calculated numbers are different, this mismatch indicates that the data area of the sector has become corrupted and that the disk sector may be bad.
- The ECC is an error-correcting code because it contains the information of how many bits of data have been corrupted and identifies the corrupted bits and corrects the bits.

Before operating system can use a disk to hold files, the operating system still needs to record its own data structures on the disk. This will be done in two steps:

1. A disk is to be **partition** into one or more groups of cylinders. The operating system can treat each partition as a separate disk. **Example:** One partition can hold a copy of the operating system's executable code, while another partition holds user files.
2. **Logical Formatting:** It means creation of a file system. the operating system stores the initial file-system data structures onto the disk. These data structures may include maps of free and allocated space and an initial empty directory.

Raw Disk is a partition that does not contain any file system. An I/O operation done on this raw disk is termed as Raw I/O.

Note: To increase efficiency, most file systems group blocks together into larger chunks, frequently called **clusters**. Disk I/O is done via blocks, but file system I/O is done via clusters.

Boot Block

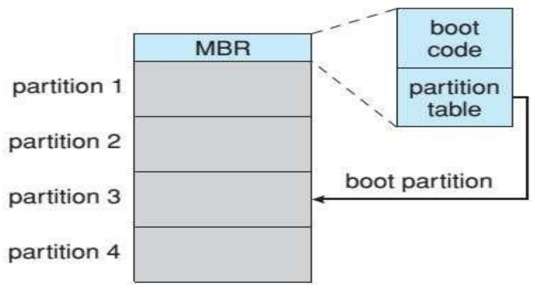
- When a computer is powered up or rebooted an initial program called bootstrap program will run. The bootstrap program initializes all aspects of the system, from CPU registers to device controllers and the contents of main memory and then starts the operating system.
- To start the operating system, the bootstrap program finds the operating-system kernel on disk, loads that kernel into memory and jumps to an initial address to begin the operating-system execution.

- The bootstrap is stored in **read-only memory (ROM)**, because ROM needs no initialization and it is at a fixed location that the processor can start executing when powered up or reset. Since ROM is read only, it cannot be infected by a computer virus.

Problem: Changing this bootstrap code requires changing the ROM hardware chips.
Solution: Most systems store a tiny bootstrap loader program in the boot ROM whose only job is to bring in a full bootstrap program from disk.

- The full bootstrap program can be changed easily: a new version is simply written onto the disk.
- The full bootstrap program is stored in the –boot blocks‖ at a fixed location on the disk. A disk that has a boot partition is called a **boot disk** or **system disk**.
- The code in the boot ROM instructs the disk controller to read the boot blocks into memory (no device drivers are loaded at this point) and then starts executing that code.

Booting in Windows Operating system
Windows allows a hard disk to be divided into partitions and one partition identified as the **boot partition.**



- **The boot partition** contains the operating system and device drivers.
- The Windows system places its boot code in the first sector on the hard disk called the **Master Boot Record** (**MBR)**.
- Booting begins by running code that is resident in the system's ROM memory. This code directs the system to read the boot code from the MBR.
- The MBR also contains a table listing the partitions for the hard disk and a flag indicating which partition the system is to be booted from.
- Once the system identifies the boot partition, it reads the boot sector from that partition and continues with the remainder of the boot process, which includes loading the various subsystems and system services.

<u>**Bad Blocks**</u>
The disks are prone to failure. If the disk is completely failed then the disk is to be replaced and contents are restored from backup media to the new disk.
If the failure is partial (i.e.) one or more sectors become defective, These sectors are called Bad Blocks.
Depending on the disk and controller in use, these blocks are handled in a variety of ways:

- While the disk is being formatted, the disk can be scanned to find the bad blocks.
- Any bad blocks that are discovered are flagged as unusable so that the file system does not allocate them.

- If blocks go bad during normal operation, a special program must be run manually to search for the bad blocks and to lock them away.
- Data that resided on the bad blocks usually are lost.

Bad Block Recovery

- The controller maintains a list of bad blocks on the disk.
- This Bad Block list is initialized during the low-level formatting at the factory and this list is updated over the life of the disk.
- Low-level formatting also sets aside spare sectors not visible to the operating system.
- The controller can be told to replace each bad sector logically with one of the spare sectors. This scheme is known as **Sector Sparing** or **Forwarding**.
- Most disks are formatted to provide a few spare sectors in each cylinder and a spare cylinder.
- When a bad block is remapped, the controller uses a spare sector from the same cylinder.

SWAP-SPACE MANAGEMENT

Swap-Space Management is another low-level task of the operating system.

- Virtual memory uses disk space as an extension of main memory by using swap space.
- Since disk swap space access is much slower than main memory access the system performance decreases.
- The main goal for the design and implementation of swap space is to provide the best throughput for the virtual memory system.

Swap-Space Use

- Swap space is used in various ways by different operating systems, depending on the memory-management algorithms in use.
- Systems that implement swapping may use swap space to hold an entire process image, including the code and data segments.
- Systems that support Paging may simply store pages that have been pushed out of main memory.
- The amount of swap space needed on a system can therefore vary from a few megabytes of disk space to gigabytes, depending on the amount of physical memory, the amount of virtual memory it is using for back-up and the way in which the virtual memory is used.

Example: Older Linux systems has suggested setting swap space to double the amount of physical memory but the modern Linux systems use considerably less swap space.

Note: Some operating systems including Linux allow the use of multiple swap spaces, including both files and dedicated swap partitions.

Swap-Space Location

A swap space can reside in one of two places:

1. It can be carved out of the normal file system.
2. It can be in a separate disk partition.

- Swap-space can be created in a separate **Raw Partition**.
- No file system or directory structure is placed in this space instead a separate swap-space storage manager is used to allocate and deallocate the blocks from the raw partition.

- Swap space manager uses algorithms optimized for speed rather than for storage efficiency, because swap space is accessed much more frequently than file systems when it is used.
- The raw-partition approach creates a fixed amount of swap space during disk partitioning.
- Adding more swap space requires either repartitioning the disk or adding another swap space elsewhere.

REDUNDANT ARRAYS OF INDEPENDENT DISKS (RAID) STRUCTURE

- Disk drives have continued to get smaller and cheaper, so it is now economically feasible to attach many disks to a computer system.
- Having a large number of disks in a system and if they are operated in parallel we can improve the rate at which data can be read or written.
- This setup offers the potential for improving the reliability of data storage, because redundant information can be stored on multiple disks. Thus, failure of one disk does not lead to loss of data.
- This disk-organization techniques is called as **Redundant Arrays Of Independent Disks (RAID)**.
- RAIDs are used for Higher reliability and Higher data-transfer rates.

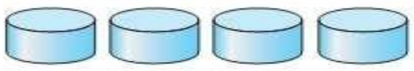
RAID Levels

RAIDs can be implemented in different levels:

1. RAID 0: Non-Redundant Striping
2. RAID 1: Mirrored Disks.
3. RAID 2: Memory-Style Error-Correcting Codes.
4. RAID 3: Bit-Interleaved Parity.
5. RAID 4: Block-Interleaved Parity.
6. RAID 5: Block-Interleaved Distributed Parity
7. RAID 6: P+Q redundancy
8. RAID 0 + 1 and 1 + 0

RAID 0: Non-Redundant Striping

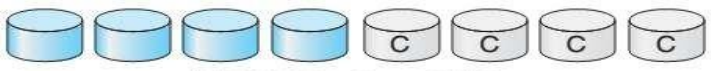
RAID level 0 refers to disk arrays with striping at the level of blocks but without any redundancy. (i.e.) some part of the data is stored in one disk other part of the data is stored in other disks without duplicating the data.



(a) RAID 0: non-redundant striping.

RAID level 1: Disk Mirroring

The entire data in the disk is copied in to other disks. (i.e.) the data that is stored in one disk the same data will be copied in other disk. If one disk fails we can recover the data from its copied disk called as Backup disk.



(b) RAID 1: mirrored disks.

Note: Mirroring provides high reliability, but it is expensive. Striping provides high data-transfer rates, but it does not improve reliability.

RAID level 2. Memory-Style Error-Correctingcode (ECC) Organization

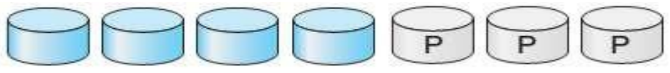
- It uses parity bits to detect errors. Each byte in a memory system may have a parity bit associated with it that records whether the number of bits in the byte set to 1 is even (parity = 0) or odd (parity = 1).
- If one of the bits in the byte is damaged (i.e.) either a 1 becomes a 0 or a 0 becomes a 1, the parity of the byte changes and thus does not match the stored parity.
- Similarly, if the stored parity bit is damaged, it does not match the computed parity.
- Thus, all single-bit errors are detected by the memory system.

ECC can be used directly in disk arrays via striping of bytes across disks.

Example: The first bit of each byte can be stored in disk 1, the second bit in disk 2 and so on until the eighth bit is stored in disk 8; the error-correction bits are stored in further disks.

If one of the disks fails, the remaining bits of the byte and the associated error-correction bits can be read from other disks and used to reconstruct the damaged data.

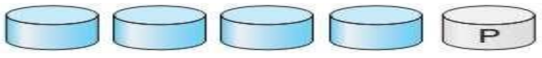
RAID level 2 requires only three disks whereas RAID1 requires four disks.



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

RAID level 3: Bit-Interleaved Parity

- Here, the disk controllers can detect whether a sector has been read correctly, so a single parity bit can be used for error correction as well as for detection.
- If one of the sectors is damaged, we know exactly which sector it is and we can figure out whether any bit in the sector is a 1 or a 0 by computing the parity of the corresponding bits from sectors in the other disks.
- If the parity of the remaining bits is equal to the stored parity, the missing bit is 0; otherwise, it is 1.
- RAID level 3 is less expensive than RAID2, it requires only one extra disk.

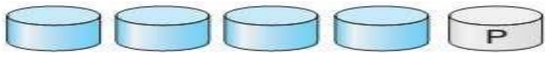


(d) RAID 3: bit-interleaved parity.

RAID level 4: Block-Interleaved Parity Organization

It Uses block-level striping and keeps a parity block on a separate disk for corresponding blocks from *N* other disks.

If one of the disks fails, the parity block can be used with the corresponding blocks from the other disks to restore the blocks of the failed disk.



(e) RAID 4: block-interleaved parity.

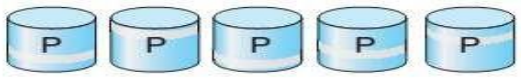
RAID level 5: Block-Interleaved Distributed Parity

It differs from level 4 in that it spreads data and parity among all *N*+ 1 disks, rather than storing data in *N* disks and parity in one disk. For each block, one of the disks stores the parity and the others store data.

Ex: With an array of five disks, the parity for the *n*th block is stored in disk **(*n* mod 5)+1**.

- The *n*th blocks of the other four disks store actual data for that block.

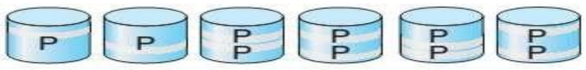
- A parity block cannot store parity for blocks in the same disk, because a disk failure would result in loss of data as well as of parity and the loss would not be recoverable.
- By spreading the parity across all the disks in the set, RAID 5 avoids potential overuse of a single parity disk, which can occur with RAID 4.
- RAID 5 is the most common parity RAID system.



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

RAID level 6: P + Q redundancy scheme

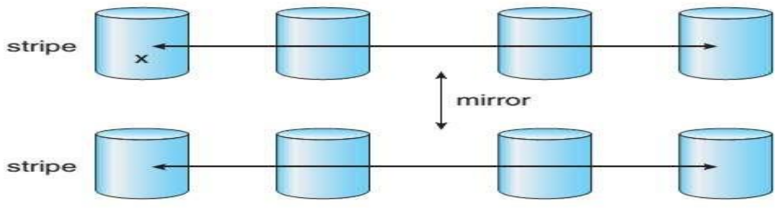
- It is like RAID level 5 but stores extra redundant information to guard against multiple disk failures.
- Instead of parity, error-correcting codes such as the **Reed–Solomon codes** are used.
- 2 bits of redundant data are stored for every 4 bits of data compared with 1 parity bit in level 5 and the system can tolerate two disk failures.



(g) RAID 6: P + Q redundancy.

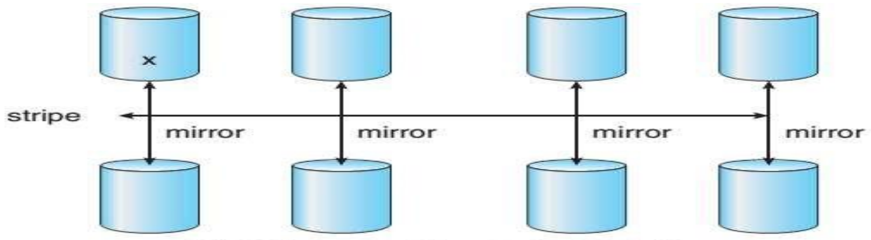
RAID levels 0 + 1 and 1 + 0

- RAID level 0 + 1 refers to a combination of RAID levels 0 and 1.
- RAID 0 provides the performance, while RAID 1 provides the reliability.
- In RAID 0 + 1, a set of disks are striped and then the stripe is mirrored to another, equivalent stripe.



a) RAID 0 + 1 with a single disk failure.

- RAID level 1 + 0, in which disks are mirrored in pairs and then the resulting mirrored pairs are striped.
- If a single disk fails in RAID 0 + 1, an entire stripe is inaccessible, leaving only the other stripe.
- With a failure in RAID 1 + 0, a single disk is unavailable, but the disk that mirrors it is still available, as are all the rest of the disk.



b) RAID 1 + 0 with a single disk failure.

DISK ATTACHMENT

Computers access disk storage in two ways.

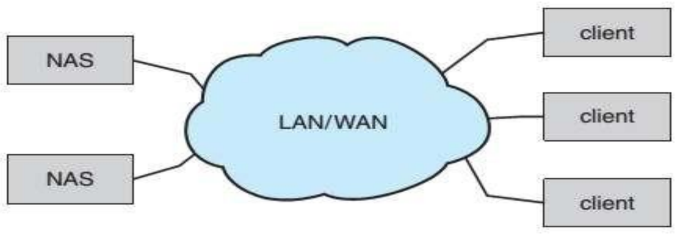
1. Host-Attached Storage (HAS)
2. Network-Attached Storage (NAS)

Host-Attached Storage

- Host-attached storage is storage accessed through local I/O ports.
- The typical desktop PC uses an I/O bus architecture called IDE or ATA or SATA.
- This architecture supports a maximum of two drives per I/O bus.
- Hard disk drives, RAID arrays and CD, DVD and tape drives are storage devices that are suitable for use as Host-Attached Storage.
- The I/O commands that initiate data transfers to a host-attached storage device are reads and writes of logical data blocks directed to specifically identified storage units.

Network-Attached Storage

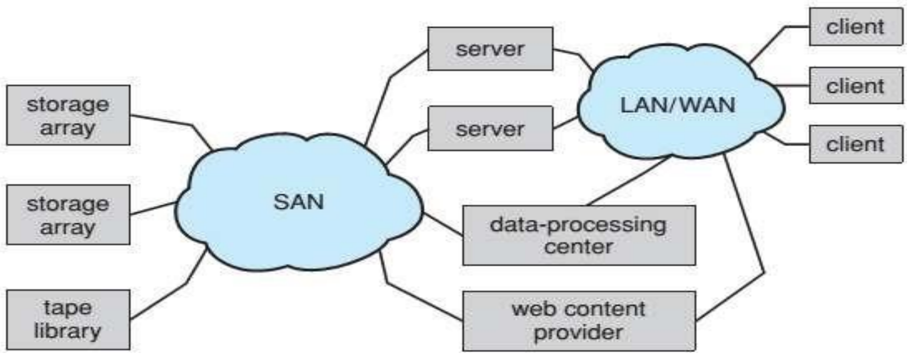
- A network-attached storage (NAS) device is a special-purpose storage system that is accessed remotely over a data network.



- Clients access network-attached storage via a remote-procedure-call interface such as NFS for UNIX systems or CIFS for Windows machines.
- The remote procedure calls (RPCs) are carried via TCP or UDP over an IP network usually the same local-area network (LAN) that carries all data traffic to the clients.
- Network-attached storage provides a convenient way for all the computers on a LAN to share a pool of storage with the same ease of naming and access enjoyed with local host-attached storage.

Storage-Area Network

A storage-area network (SAN) is a private network connecting servers and storage units.



- Multiple hosts and multiple storage arrays can attach to the same SAN and storage can be dynamically allocated to hosts.
- A SAN switch allows or prohibits access between the hosts and the storage.
- Example: If a host is running low on disk space, the SAN can be configured to allocate more storage to that host.
- SANs make it possible for clusters of servers to share the same storage and for storage arrays to include multiple direct host connections.