



Yashwantrao  
Chavan  
Maharashtra  
Open University

**CMP507**  
**Operating**  
**System**

# Operating System

---

**Yashwantrao Chavan Maharashtra Open University**  
Dnyangangotri, Near Gangapur Dam  
Nashik-422222

---

**Yashwantrao Chavan Maharashtra Open University**

---

Vice-Chancellor: Prof. E. Vayunandan

---

**SCHOOL OF COMPUTER SCIENCE**

<b>Dr. Pramod Khandare</b> Director School of Computer Science Y.C.M.Open University Nashik	<b>Shri. Madhav Palshikar</b> Associate Professor School of Computer Science Y.C.M.Open University Nashik	<b>Dr. P.V. Suresh</b> Director School of Computer and Information Sciences I.G.N.O.U. New Delhi
<b>Dr. Pundlik Ghodke</b> General Manager R&D, Force Motors Ltd. Pune.	<b>Dr. Sahebrao Bagal</b> Principal, Sapkal Engineering College Nashik	<b>Dr. Madhavi Dharankar</b> Associate Professor Department of Educational Technology S.N.D.T. Women's University, Mumbai
<b>Dr. Urmila Shrawankar</b> Associate Professor, Department of Computer Science and Engineering G.H. Rasoni College of Engineering Hingana Road, Nagpur	<b>Dr. Hemant Rajguru</b> Associate Professor, Academic Service Division Y.C.M.Open University Nashik	<b>Shri. Ram Thakar</b> Assistant Professor School Of Continuing Education, Y.C.M.Open University Nashik
<b>Mrs. Chetna Kamalskar</b> Assistant Professor School of Science and Technology Y.C.M.Open University, Nashik	<b>Smt. Shubhangi Desle</b> Assistant Professor Student Service Division Y.C.M.Open University Nashik	

<b>Writer</b>	<b>Editor</b>	<b>Co-ordinator</b>	<b>Director</b>
<b>1. Prof. Tushar Kute</b> Assistant Professor, Researcher, Computer Science, MITU, Skillologics, Pune <b>2. Ms. Monali R. Borade</b> Academic Co-ordinator School of Computer Science Y.C.M. Open University, Nashik <b>3. Prof. Asmita Hendre</b> Assistant Professor, visiting faculty Sinhgad Institute of Management, Pune YCMOU, Pune	Mr. N.L Bhale HOD,Assistant Professor, Matoshri College of Engineering & Research Centre, Eklahare, Nashik	Ms. Monali R. Borade Academic Co-ordinator School of Computer Science, Y.C.M. Open University, Nashik	<b>Dr. Pramod Khandare</b> <b>Director</b> School of Computer Science, Y.C.M. Open University, Nashik

---

**Production**

---

Note: This Study material is still under development and editing process. This draft is being made available for the sole purpose of reference. The contents are subjected to modification/updation. Final edited copies will be made available once ready.

Unit No. & Name	Details	Counselling Sessions	Weightage
Unit 1 History of The Operating Systems	<ul style="list-style-type: none"> <li>• Introduction(What is OS, Important of OS, Features, Uses, Applications)</li> <li>• Evolution of OS (proprietary, CP/M, DOS, UNIX, Windows and other, Command line to GUI, Portability, Client Server)</li> <li>• Types of Operating System(multiprogramming systems, batch systems , time sharing systems; operating systems for personal computers &amp; workstations, process control &amp; real time systems.)</li> <li>• User's View of the Operating System</li> </ul>		10
Unit 2 Operating System –Functions And Structure	<ul style="list-style-type: none"> <li>• Different Services of the Operating Systems <ul style="list-style-type: none"> <li>○ Information Management</li> <li>○ Process Management</li> <li>○ Memory Management</li> </ul> </li> <li>• Uses of System Calls</li> <li>• Operating System Structure (Monolithic (Simple) Operating System, Layered Operating System, Microkernel Operating System, Exokernel Operating system),</li> <li>• Virtual Machine</li> <li>• Booting</li> </ul>		10
Unit 3 Information Management	<ul style="list-style-type: none"> <li>• Disk Basics</li> <li>• Direct Memory Access (DMA)</li> <li>• File System ( Block and Block numbering Scheme, File Support Levels, Writing/Reading a Record, Relationship between the Operating System and DMS, File Directory Entry, Open/Close Operations, Disk Space Allocation Methods, Directory Structure: User's View, Implementation of a Directory System)</li> <li>• Device Driver (DD) (Basics, Path Management, Submodules of DD)</li> </ul>		10

Unit No. & Name	Details	Counselling Sessions	Weightage
Unit4 Process Management	<ul style="list-style-type: none"> <li>• Process,</li> <li>• Evolution of Multiprogramming</li> <li>• Context Switching,</li> <li>• Process States, Process State Transitions, Process Control Block (PCB), Process Hierarchy, Operation on a Process, Create/ Kill/ Dispatch a Process, Change the Priority of a Process,</li> <li>• Block / Time Up /Wake Up a Process, Suspend/ Resume Operations,</li> <li>• Process Scheduling (Objectives, Concepts of Priority and Time Slice, Scheduling philosophies, Scheduling Levels, Scheduling Policies (For Short Term scheduling)),</li> <li>• Multithreading (Models, Implementation of Threads)</li> </ul>		10
Unit 5 Inter Process Communication	<ul style="list-style-type: none"> <li>• The Producer-Consumer Problems, Solutions to the Producer-Consumer Problems (Interrupt Disabling/Enabling, Lock-flag,</li> <li>• What are Primitives for Mutual Exclusion?</li> <li>• Classical IPC problems</li> <li>• Semaphores</li> <li>• Alternating Policy</li> <li>• Peterson's Algorithm</li> </ul>		15
Unit 6 I/O Management And Deadlock	<ul style="list-style-type: none"> <li>• I/O Procedure, I/O Scheduler, Device Handler, Interrupt Service Routine (ISR)</li> <li>• Terminal I/O(Terminal Hardware, Terminal Software)</li> <li>• Organizing Data on the CD-ROM, DVD-ROM</li> <li>• Graphical Representation of a Deadlock,</li> <li>• Deadlock Prerequisites</li> <li>• Deadlock Strategies (Ignore a Deadlock, Detect a Deadlock, Recover from a Deadlock, Prevent a Deadlock, Avoid a Deadlock)</li> </ul>		10

Unit 7 Memory Management	<ul style="list-style-type: none"> <li>• Single Contiguous Memory Management</li> <li>• Fixed Partitioned Memory Management</li> <li>• Variable Partitions (Allocation Algorithms, Swapping, Relocation and Address Translation, Protection and Sharing, Evaluation)</li> <li>• Non-Contiguous Allocation – General Concepts, Paging (Allocation Algorithms, Swapping, Relocation and Address Translation), Segmentation (Swapping, Address Translation and Relocation, Sharing and Protection)</li> <li>• Combined Systems</li> <li>• Virtual Memory Management Systems (Relocation and Address Translation, Swapping, Relocation and Address Translation, Protection and Sharing, Evaluation, Design Consideration for Virtual Systems)</li> </ul>		<b>10</b>
Unit 8 Protection and Security	<ul style="list-style-type: none"> <li>• Protection and Security Policy mechanism</li> <li>• Authentication</li> <li>• Internal access Authorization</li> </ul>		<b>5</b>
	Revision	4	<b>0</b>
		30	<b>80</b>

# Unit 1 History of the Operating Systems

## 1.1 Introduction

### 1.1.1 What is OS?

An operating system is a program that works as an intermediary/interface between the user of a computer and the computer hardware. The purpose of an operating system is to provide an environment in which a user can execute programs in a convenient and efficient manner. In simple terms an OS is a program that controls all the computer resources/hardware and provides a platform/environment in which a user can execute/run programs.

An Operating system (OS) is software which acts as an interface between the end user and computer hardware. Every computer must have at least one OS to run other programs. Applications like Chrome, MS Word, Games, etc. needs some environment in which it will run and perform its task. The OS helps you to communicate with the computer without knowing how to speak the computer's language. It is not possible for the user to use any computer or mobile device without having an operating system.

### 1.1.2 Features of Operating System

- 1. Convenience:** An operating system makes computer more simple or convenient to use by hiding the hardware details of the computer. It's easy to use as it has a GUI interface.
- 2. Memory Management:** The operating system manages memory. It has complete knowledge of primary memory; which part of the memory is used by which program. Whenever a program requests, it allocates memory.
- 3. Processor Management and program execution:** It allocates the program to the processor (CPU) and also de-allocates it when a program runs out of the CPU needs. OS loads a program into memory and executes the program. The program must be able to end its execution, either normally or abnormally.
- 4. Device Management/I/O operation:** The operating system keeps the information about all devices. It is also called the I/O controller, and the operating system also decides which devices are used to which program, when, and for how long.
- 5. Communication:** when Data transfer between two processes is required, whether both processes are on the computer or on different computer but connected through computer network operating system provides two ways for communication: shared memory and message passing.
- 6. Error Detection/handling:** errors may occur in CPU, in I/O devices or in the memory hardware. The operating system constantly needs to monitor the possible errors. It should take the appropriate actions to provide correct and consistent computing operations.
- 7. Security and Reliability:** It prevents unauthorized access to any program. It uses passwords and other technologies. It is very reliable because any virus and harmful code can be detected in it with the help of proper security enforcing techniques.

### 1.1.3 Uses of Operating systems

The operating system is essential for the operation of any computer. Even in a single-OS that appears, a computer performs three main functions:

- (1) Manage the computer's resources, such as the central processing unit, memory, disk drives, and printers,
- (2) Establish a user interface, and
- (3) Execute and provide services for applications software.

The operating system is used everywhere today, such as banks, schools, hospitals, companies, mobiles, etc. No device can operate without an operating system because it controls all the user's commands. For example-

- **LINUX/UNIX** operating system is used in the bank because it is a very secure operating system.
- **Symbian OS, Windows Mobile, iOS, and Android OS** are used in mobile phone operating systems as these operating systems are a lightweight operating system.

### 1.1.4 Applications of Operating System

1. Embedded systems (Mostly use in home appliances)
2. Automobile engine controllers system
3. Industrial robots, Medical Robots and Research
4. Spacecraft Control System
5. Industrial Control
6. Large-scale computing systems such as green computing, mobile computing, soft computing etc.
7. Diagnostic mode of a computer operating system (OS).
8. Real time systems are used in: Airlines reservation system, Air traffic control system, Systems that provide immediate updating, Defense application systems like RADAR, Networked Multimedia Systems, and Command Control Systems.

### 1.2 Evolution of Operating System

Operating systems have been evolving through the years. Following table shows a summarized history of operating system.

Generation	Year	Electronic devices used	Types of OS and devices
First	1945-55	Vacuum tubes	Plug board
Second	1955-65	Transistors	Batch systems
Third	1965-1980	Integrated circuits(IC)	Multiprogramming
Fourth	Since 1980	Large scale integration	PC

#### 1.2.1 CPM

**CP/M** stands for “**Central Program for microcomputers**” was almost the first OS developed for microcomputer platform based on Intel 8080 machine in 1974 by Gary Kildall. Initially it was developed only as a file system to support all the file related operations for PL/M (High level

Programming language for 8080 machine) compiler. Later utilities like editors, debuggers etc. were added to it. CP/M became popular because of its user friendliness, time sharing and real-time capabilities however it had slower disk operations and couldn't support networking capabilities. Hence later DOS (Disk Operating System) came in picture.

**Note:** PL/M was developed by Gary Kildall in 1973 as a high-level programming language for integrated microprocessors.

### 1.2.2 Proprietary Operating System

Proprietary term describes something owned by a specific individual or company. Usually proprietary software like operating system is "closed-sourced" i.e. It's not open source, freely available or freely licensed. It is designed, customized, developed and sell by only one company. For example-MAC OS X operating system for apple machines developed by Apple Company which is strictly allowed to run on apple hardware.

Some of the features of proprietary OS are simplified user experience; complex graphics, limited customizability, interoperability, development and resource support from manufacturer etc. These type of OS are usually costly as compared to open source OS.

### 1.2.3 DOS/MS-DOS

A company called "Seattle Computer" developed an OS called QDOS for Intel 8086 m/c. Later Microsoft Corporation acquired rights to it and developed MS-DOS (Microsoft Disk Operating system) OS.

#### **The main features of MS-DOS are:**

It is a character user interface (CUI) based OS.

It is a command line operating system.

It supports single-user, single-tasking.

It is a 16-bit OS.

Ms-DOS became popular because features like user-friendliness, faster disk operations, compiler support for various high-level languages like BASIC, COBOL and C language and networking capabilities like LAN. It became immensely popular for software development. Later it adopted GUI (Graphical User Interface) and Windows (MS Windows) was developed.

### 1.2.4 UNIX, LINUX

**UNIX** was initially developed as a single user operating system. It was developed much before Microsoft developed DOS. The design of DOS is based on the features of UNIX operating system. It is popular because of features like multiple user support simultaneously. It is a CUI based OS that supports multiprogramming, time-sharing and multitasking.

It's written in high level language; the code developed in UNIX can be easily modified and compiled on any system even though the hardware is different hence portability and machine independence are the important features of UNIX.

Later **LINUX** an open sourced OS was developed based on features of UNIX. It doesn't need licence and it follows open development model which allows any programmers to access the source code and modify operating system. Apart from supporting UNIX features it provides high security and cross-platform capabilities. Hence it's extremely powerful, popular and inexpensive operating system.



### 1.2.5 Microsoft Windows and others

Microsoft's windows family consists of various graphical user interface (GUI) based OS like Windows 95, Windows 98, Windows NT, Windows 2000, Windows XP and latest versions. The most popular windows OS currently in use are windows XP and its later versions like Windows 8, windows 10 which supports greater networking capabilities, user friendliness, more stability and security, 32-bit/64-bit operations, multitasking and multiprogramming, plug and play features ,compiler support for several advance languages and software development, faster performance.

Some higher versions of operating systems can be customized for parallel processing and advanced graphics capabilities, portability and interoperability and distributed environment.

### 1.2.6 Command Line to GUI

A **command line interface (CLI)** is a text-based user interface (UI) used to view and manage computer files. Before the mouse, users interacted with an operating system (OS) or application with a keyboard. Users typed commands in the command line interface to run tasks or programs on a computer.

Typically, the command line interface looks like a black box with white text. The user responds to a prompt/text in the command line interface by typing a command. The output or response from the system can include a message, table, list, or some other confirmation of a system or application action. The software that handles the command line interface is called shell, also commonly referred to as a command language interpreter. The CLI Commands were difficult to remember and it lacked user friendliness.

The **graphical user interface (GUI)** is the most popular user interface today. A GUI uses windows, menus and icons to execute commands. The advantage of a GUI is, it provides pictorial/visual view of the available functions and it's simple & easy to use. A mouse is the most common way to navigate through a GUI, although many GUIs allow navigation and execution via a keyboard. One example of a GUI-based application is Microsoft Word. A user can change options for page layouts and styles by selecting the corresponding icon with a mouse or keyboard. The GUI is more user-friendly than CLI.

### 1.2.7 Client-Server

#### 1. Client OS:

It is an operating system that operates within desktop. It is used to obtain services from a server. It run on the client devices like laptop, computer and is very simple operating system.

#### 2. Server OS:

It is an operating system that is designed to be used on server. It is used to provide services to multiple clients. It can serve multiple clients at a time and is very advanced operating system.

Client/Server communication involves two components, namely a client and a server. They are usually multiple clients in communication with a single server. The clients send requests to the server and the server responds to the client requests

### 1.3 Types of OS

An operating system may process its workload serially or concurrently. That is resources of the computer system may be dedicated to a single program until its completion, or they may be dynamically reassigned among a collection of active programs in different stages of execution. Several variations of both serial and multiprogrammed operating systems exist.

### 1.3.1 Batch Operating System

- Some computer systems only did one thing at a time. They had a list of instructions to carry out and these would be carried out one after the other this is called a serial system. The mechanics of development and preparation of programs in such environments are quite slow and numerous manual operations involved in the process.
- In the era of 1970s, the Batch processing was very popular. The Jobs were executed in batches. People were used to have a single computer which was called mainframe. In Batch operating system, access is given to more than one person; they submit their respective jobs to the system for the execution.
- The system put all of the jobs in a queue on the basis of first come first serve and then executes the jobs one by one. The users collect their respective output when all the jobs get executed.
- Batch operating system is one where programs and data are collected together in a batch before processing starts. A job is predefined sequence of commands, programs and data that are combined into a single unit called Job. The memory layout for a simple batch system Memory management in batch system is very simple. Memory is usually divided into two areas: Operating system and user program area.
- Scheduling is also simple in batch system. Jobs are processed in the order of submission i.e. first come first served fashion.
- When a job completes execution, its memory is released and the output for the job gets copied into an output spool for later printing. Spooling is similar to buffer queue where all the printing operations/job are put in order.

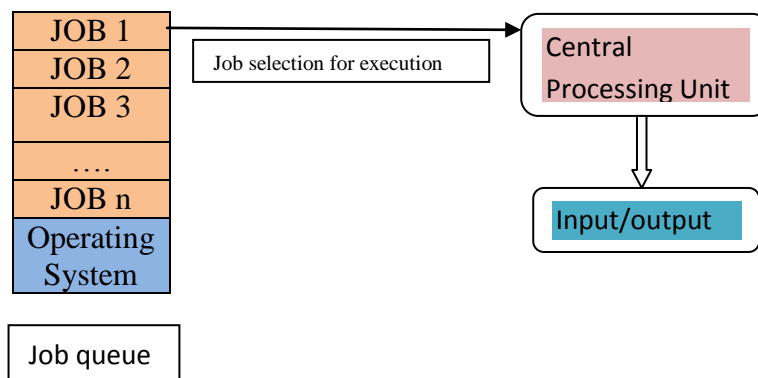


Figure 1.1 Batch OS Job execution

- **Advantages of batch system**

1. Move much of the work of the operator to the computer.
2. Increased performance since it was possible for job to start as soon as the previous job is finished.

- **Disadvantages of Batch OS**

#### 1. Starvation

Batch processing can suffer from starvation. If there are five jobs J1, J2, J3, J4, J5 present in the batch. If the execution time of J1 is very high then other four jobs will never be going to get executed or they will have to wait for a very high time. Hence the other processes get starved.

#### 2. Not Interactive

Batch Processing is not suitable for the jobs which are dependent on the user's input. If a job requires the input of two numbers from the console then it will never be going to get it in the batch processing scenario since the user is not present at the time of execution.

3. Turnaround time can be large from user standpoint.
4. Difficult to debug program
5. A job could enter an infinite loop
6. A job could corrupt the monitor, thus affecting pending jobs
7. Due to lack of protection scheme, one batch job can affect pending

### 1.3.2 Multiprogramming Operating System

- Multiprogramming is an extension to the batch processing where the CPU is kept always busy. Each process needs two types of system time: CPU time and IO time. In multiprogramming environment, for the time a process does its I/O, The CPU can start the execution of other processes. Therefore, multiprogramming improves the efficiency of the system.
- When two or more programs are in memory at the same time, sharing the processor is referred to the multiprogramming operating system. Multiprogramming assumes a single processor that is being shared. It increases CPU utilization by organizing jobs so that the CPU always has one to execute. Fig shows the memory layout for a multiprogramming system.
- The operating system keeps several jobs in memory at a time. This set of jobs is a subset of the jobs kept in the job pool. The operating system picks and begins to execute one of the jobs in the memory.

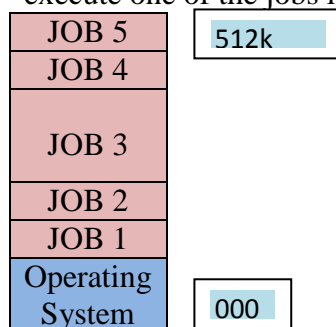


Figure 1.2 Multiprogramming OS job execution

- Multiprogrammed systems provide an environment in which the various system resources are utilized effectively, but they do not provide for user interaction with the computer system.
- Jobs entering into the system are kept into the memory. Operating system picks the job and begins to execute one of the jobs in the memory. Having several programs in memory at the same time requires some form of memory management.
- Multiprogramming operating system monitors the state of all active programs and system resources. This ensures that the CPU is never idle unless there are no jobs.

#### Advantages:

1. High CPU utilization.
2. It appears that many programs are allotted to CPU almost simultaneously.

#### Disadvantages:

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

### 1.3.3 Time sharing systems

- Time sharing system supports interactive users. It is also called multitasking. It is logical extension of multiprogramming. Time sharing system uses CPU scheduling and multiprogramming to provide an economical interactive system of two or more users.
- A time shared operating system uses CPU scheduling and multi-programming to provide each with a small portion of a shared computer at once. Each user has at least one separate program in memory. A program loaded into memory and executes, it performs a short period of time either before completion or to complete I/O. This short period of time during which user gets attention of CPU is known as time slice, time slot or quantum. It is typically of the order of 10 to 100 milliseconds.
- Time shared operating systems are more complex than multiprogramming operating systems. Memory management in time sharing system provides for isolation and protection of co-resident programs. In both, multiple jobs must be kept in memory simultaneously, so the system must have memory management and security. To achieve a good response time, jobs may have to swap in and out of disk from main memory which now serves as a backing store for main memory. A common method to achieve this goal is virtual memory, a technique that allows the execution of a job that may not be completely in memory.
- Time sharing system can run several programs at the same time, so it is also a multiprogramming system. But multiprogramming operating system is not a time sharing system. Difference between both the systems is that time sharing system allows more frequent context switches. This gives each user the impression that the entire computer is dedicated to his use. In multiprogramming system a context switch occurs only when the currently executing process stalls for some reason.

#### **Advantages:**

1. Each task gets an equal opportunity.
2. Less chances of duplication of software.
3. CPU idle time can be reduced.

#### **Disadvantages:**

1. Reliability problem.
2. One must have to take of security and integrity of user programs and data.
3. Data communication problem.

### 1.3.4 Process Control and Real Time Operating System

- RTOS is an operating system intended to serve real time application that process data as it comes in, mostly without buffer delay. Real time systems which were originally used to control autonomous systems such as satellites, robots and hydroelectric dams. A real time operating system is one that must react to inputs and responds to them quickly. A real time system cannot afford to be late with a response to an event or else the output will be useless or failure.
- Real time systems are divided into two groups: Hard real time system and soft real time system.

**A hard real time system:** Hard RTOS, the deadline is handled very strictly which means that given task must start executing on specified scheduled time, and must be completed within the assigned time duration. Example: Medical critical care system, Aircraft systems, etc.

**Soft real time system:** is a less restrictive type. Soft Real time RTOS, accepts some delays by the Operating system. In this type of RTOS, there is a deadline assigned for a specific job, but a delay for a small amount of time is acceptable. So, deadlines are handled softly by this type of RTOS.

Example: Online Transaction system and Livestock price quotation System.

- General real time applications with some examples are listed below

#### **Applications: Example**

Detection: Radar system, Burglar alarm

Process monitoring and control: Petroleum, Paper mill

Communication: Telephone switching system

Flight simulation and control: Auto pilot shuttle mission simulator

Transportation: Traffic light system, Air traffic control

Military applications: for example missiles

Weather forecasting

### **1.3.5 OS for Personal Computer and Workstations**

- During the late 1970, computers had faster CPU, thus creating an even greater disparity between their rapid processing speed and slower I/O access time.
- Multiprogramming schemes to increase CPU use were limited by the physical capacity of the main memory, which was a limited resource and very expensive. This system includes PC running MS window and the Apple Macintosh. The Apple Macintosh OS support new advance hardware i.e. virtual memory and multitasking with virtual memory, the entire program did not need to reside in memory before execution could begin.
- Linux, a UNIX like OS available for PC, has also become popular recently. The microcomputer was developed for single users in the late 1970. Physical size was smaller than the minicomputers of that time, though larger than the microcomputers of today.
- Microcomputer grew to accommodate software with large capacity and greater speeds. The distinguishing characteristics of a microcomputer are its single user status. MS-DOS is an example of a microcomputer operating system.
- The most powerful microcomputers are used by commercial, educational, government enterprises. Hardware cost for microcomputers are sufficiently low that a single user (individuals) have sole use of a computer. Networking capability has been integrated into almost every system.
- **A workstation** is a computer intended for individual use that is faster and more capable than a personal computer. It's intended for business or professional use (rather than home or recreational use). Workstations and applications designed for them are used by small engineering companies, architects, graphic designers, and any organization, department, or individual that requires a faster microprocessor, a large amount of random access memory (RAM), and special features such as high-speed graphics adapters. UNIX operating system, which is often used as the workstation operating system. Workstation OS are also used as client operating systems.

### **1.4 User View of Operating System**

The Operating System is an interface which hides the details which must be performed and presents a virtual machine to the user that makes easier to use. An **operating system** allows the user application programs to interact with the **system** hardware. **Operating system** by itself does not provide any function but it provides an atmosphere in which different applications and programs can do useful work. Operating System provides the following services to the user.

- Execution of a program
- Access to I/O devices
- Controlled access to files
- Error detection (Hardware failures, and software errors)

The user view depends on the system interface that is used by the users. The different types of user view experiences can be explained as follows –

- If the user is using a personal computer, the operating system is largely designed to make the interaction easy. Some attention is also paid to the performance of the system, but there is no need for the operating system to worry about resource utilization. This is because the personal computer uses all the resources available and there is no sharing.
- If the user is using a system connected to a mainframe or a minicomputer, the operating system is largely concerned with resource utilization. This is because there may be multiple terminals connected to the mainframe and the operating system makes sure that all the resources such as CPU, memory, I/O devices etc. are divided uniformly between them.
- If the user is sitting on a workstation connected to other workstations through networks, then the operating system needs to focus on both individual usage of resources and sharing through the network. This happens because the workstation exclusively uses its own resources but it also needs to share files etc. with other workstations across the network.
- If the user is using a handheld computer such as a mobile, then the operating system handles the usability of the device including a few remote operations. The battery level of the device is also taken into account.

There are some devices that contain very less or no user views because there is no interaction with the users. Examples are embedded computers in home devices, automobiles etc.

## **Unit 2 Operating System –Functions and Structure**

### **2.1 Different Services of the Operating Systems**

An Operating System supplies different kinds of services to both the users and to the programs as well. It also provides application programs (that run within an Operating system) an environment to execute it freely. It provides users the services run various programs in a convenient manner. The services provided by one operating system can be different than other operating system. Operating system makes the programming task easier. The common services provided by the operating system are explained below.

#### **2.1.1 Process Management**

- Process refers to a program in execution. The process management 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.
- A process needs certain resources, such as CPU time, memory, files and I/O devices. These resources are either given to the process when it is created or allocated to it while it is running.
- When the process terminates, the operating system will reclaim any reusable resources. All these processes should be managed by process management; the execution of a process must be sequential so, at least one instruction should be executed on behalf of the process.
- The term process refers to an executing set of machine instructions. Program by itself is not a process. A program is a passive entity.
- The operating system is responsible for the following activities of the process management:
  1. Creating and destroying the user and system processes.
  2. Allocating hardware resources among the processes.
  3. Controlling the progress of processes.
  4. Providing mechanisms for process communications.
  5. Also provides mechanisms for deadlock handling

#### **2.1.2 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. For a program to be executed, it must be in the main Memory.
- The memory management modules of an operating system are concerned with the management of the primary (main memory) memory. Memory management is concerned with following functions:
  1. Keeping track of the status of each location of main memory. I.e. each memory location is either free or allocated.
  2. Determining allocation policy for memory
  3. Allocates the memory when a process requests. Allocation techniques are used to choose specific location. It is also responsible for allocation information updating.
  4. It also de-allocates the Memory when a process no longer requires or has been terminated. After de-allocation, status information must be updated
- Memory management is primarily concerned with allocation of physical memory of finite capacity to requesting processes. The overall resource utilization and other performance criteria of a computer system are affected by performance of the memory management

module. Many memory management schemes are available and the effectiveness of the different algorithms depends on the particular situation & OS.

- **Secondary Storage Management:** A storage device is a mechanism by which the computer may store information in such a way that this information may be retrieved at a later time. Secondary storage device is used for storing all the data and programs. These programs and data access by computer system must be kept in main memory. Size of main memory is small to accommodate all data and programs. It also loses the data when power is lost. For this reason, a secondary storage device is used. Therefore, the proper management of disk storage is of central importance to a computer system.

### 2.1.3 File Management

- Logically related data items on the secondary storage are usually organized into named collections called files. In short, a file is a logical collection of information. A computer uses physical media for storing the different information.
- A file may contain a report, an executable program or a set of commands to the operating system. A file consists of a sequence of bits, bytes, lines or records whose meanings are defined by their creators. For storing the files, physical media (secondary storage device) is used.
- Physical media are of different types. These are magnetic disk, magnetic tape and optical disk. All the media has its own characteristics and physical organization. Each medium is controlled by a device.
- The operating system is responsible for the following in connection with file management:
  1. Creating and deleting of files.
  2. Mapping files onto secondary storage.
  3. Creating and deleting directories
  4. Backing up files on stable storage media.
  5. Supporting primitives for manipulating files and directories.
  6. Transmission of file elements between main and secondary storage.The file management subsystem can be implemented as one or more layers of the operating system

### 2.2 Uses of 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. If a file system requires the creation or deletion of files. Reading and writing from files also require a system call. 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.

Modern processors provide instructions that can be used as system calls; System calls provide the interface between a process and the operating system. A system call instruction is an instruction that generates an interrupt that causes the operating system to gain control of the processor.

- **Services Provided by System Calls:**
  1. Process creation and management
  2. Main memory management
  3. File Access, Directory and File system management
  4. Device handling(I/O)



5. Protection
6. Networking, etc.

- **Types of System Calls:** There are 5 different categories of system calls –

**1. Process control/creation and management:**

- a. Create process
- b. Terminate process: Terminate the process making the system call
- c. Wait: wait for another process to exit
- d. Fork: Create a duplicate of the process making a system
- e. End: Halt the process execution normally
- f. Abort: Halt the process execution abnormally
- g. Load: Load the process into memory: : Execute the loaded process
- h. Execute: Execute the loaded process.
- i. Get process attributes and set process attributes
- j. Allocate and free memory

**2. File Access, Directory and File system management :** create file, open file to read or write, close file, delete a file, stat: get information about a file, unlink: remove file from a directory, get attribute and set file attribute: attributes including file names, file type, protection codes and accounting information etc.

**3. I/O Device management:**

- a. Request device: to ensure exclusive use of device
- b. Release device: Release the device after execution is finished
- c. Read/Write: Same as file system call
- d. Stat: Get information about I/O devices.

**4. Information maintenance:**

- a. Get time and date
- b. Set time and date
- c. Get process, file or device attributes
- d. Set process, file or device attributes
- e. Get system data
- f. set system data

**5. Inter-process communication:**

- a. Create message queue: Create a queue to hold message Send a message to a message queue
- b. Send message: send message to message queue
- c. Receive message: Receive a message from a message queue
- d. Close connection: Terminates the communication.

## 2.3 Operating System Structure

The design of operating system architecture traditionally follows the separation of concerns principle. This principle suggests structuring the operating system into relatively independent parts that provide simple individual features, thus keeping the complexity of the design manageable.

The operating system possesses various privileges that allow it to access otherwise protected resources such as physical devices or application memory. When these privileges are granted to the individual parts of the operating system that require them, rather than to the operating system as a whole, the potential for both accidental and malicious privileges misuse is reduced.

- **A kernel:** A kernel is the core component of an operating system. A kernel is an important part of an OS that manages system resources. It also acts as a bridge between the software and hardware of the computer. It is one of the first programs which are loaded on start-up after the boot loader. The Kernel is also responsible for offering secure access to the machine's hardware for various programs. It also decides when and how long

Using inter-process communication and system calls, it acts as a bridge between applications and the data processing performed at the hardware level. The kernel is responsible for low-level tasks such as disk management, task management and memory management. Following figure shows the **kernel of operating system**.

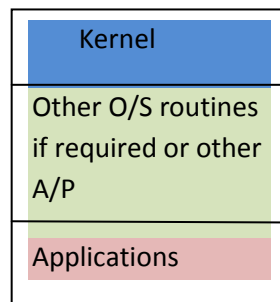


Figure 2.1 Kernel

There are various types of kernel namely monolithic kernel, Microkernel, Hybrid kernel, Exo-Kernel, and Nano-Kernel. Most widely used kernel is Monolithic kernel and Microkernel. Kernel space is where the kernel (i.e., the core of the operating system) executes (i.e., runs) and provides its services. User space is that set of memory locations in which user processes (i.e., everything other than the kernel) run. A process is an executing instance of a program.

### 2.3.1 Monolithic (Simple) Operating System

Monolithic Kernel manages system resources between application and hardware, but user services and kernel services are implemented under same address space. It increases the size of the kernel, thus increases size of operating system as well. This kernel provides CPU scheduling, memory management, file management and other operating system functions through system calls. As both services are implemented under same address space, this makes operating system execution faster. If any service fails the entire system crashes, and it is one of the drawbacks of this kernel. The entire operating system needs modification if user adds a new service.

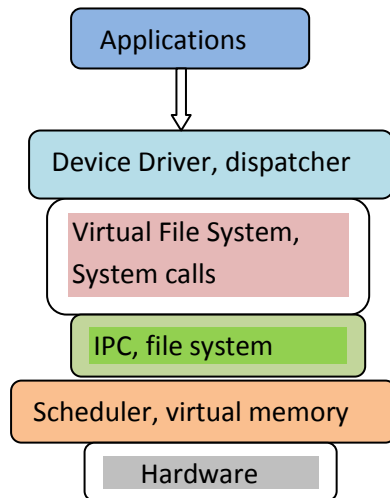


Figure 2.2 Monolithic Kernel

#### **Advantages of Monolithic Kernel –**

- One of the major advantages of having monolithic kernel is that it provides CPU scheduling, memory management, file management and other operating system functions through system calls.
- The other one is that it is a single large process running entirely in a single address space.
- Simple to design and implement.
- Can be expanded using module systems.

#### **Disadvantages of Monolithic Kernel**

- One of the major disadvantages of monolithic kernel is that, if anyone service fails it leads to entire system failure.
- If user has to add any new service. User needs to modify entire operating system.
- Module system may not provide runtime loading and unloading.
- It becomes harder to maintain as code base size increases.
- Lower fault tolerance, if core module or section of the kernel fails, the whole thing fails.

### **2.3.2 Layered Operating System**

- A layered design of the operating system architecture attempts to achieve robustness by structuring the architecture into layers with different privileges. The most privileged layer would contain code dealing with interrupt handling and context switching, the layers above that would follow with device drivers, memory management, file systems, user interface, and finally the least privileged layer would contain the applications.
- The bottom layer is the hardware layer. Bottom layer number is 0. The topmost layer (layer N) is the user interface. An operating system could be structured to contain many layers. Each layer uses the interface provided by the layer below it and provides a more intelligent interface to the layer above it. Following figure shows layered Kernel.

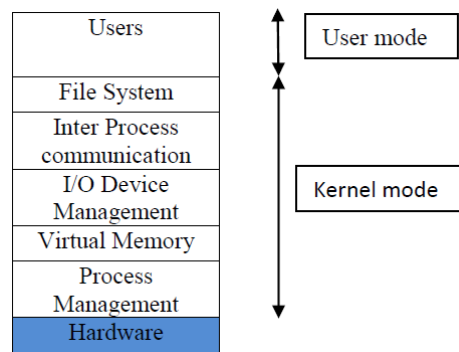


Figure 2.3 Layered OS

- Layered structure provides good modularity. Each layer of the operating system forms a module with a clearly defined functionality and interface with the rest of the operating system. This approach simplifies debugging and system verification. It also provides facility of information hiding.
- All the data and program are hidden from other modules. Debugging is done at first layer. While debugging, if any error is found, the error must be on that layer, because the layers below it are already debugged. Each layer hides the existence of certain data structures, operations and hardware from higher-level layers.
- Layered approach required careful definition of the layers, because a layer can use only those layers below it. Problem with layered implementations is that they tend to be less efficient than other types.
- **Advantages:**
  1. **Modularity:** This design promotes modularity as each layer performs only the tasks it is scheduled to perform.
  2. **Easy debugging:** As the layers are discrete so it is very easy to debug. Suppose an error occurs in the CPU scheduling layer, so the developer can only search that particular layer to debug, unlike the Monolithic system in which all the services are present together.
  3. **Easy update:** A modification made in a particular layer will not affect the other layers.
  4. **No direct access to hardware:** The hardware layer is the innermost layer present in the design. So a user can use the services of hardware but cannot directly modify or access it, unlike the Simple system in which the user had direct access to the hardware.
  5. **Abstraction:** Every layer is concerned with its own functions. So the functions and implementations of the other layers are abstract to it.
    - **Disadvantages:**
      1. **Complex and careful implementation:**  
As a layer can access the services of the layers below it, so the arrangement of the layers must be done carefully. For example, the backing storage layer uses the services of the memory management layer. So it must be kept below the memory management layer. Thus with great modularity comes complex implementation.
      2. **Slower in execution:** If a layer wants to interact with another layer, it sends a request that has to travel through all the layers present in between the two interacting layers. Thus it increases response time, unlike the Monolithic system which is faster.

### 2.3.3 Microkernel Operating System

- Microkernel is a software or code which contains the required minimum amount of functions, data, and features to implement an operating system. It provides a minimal number of mechanisms, which is good enough to run the most basic functions of an

operating system. It allows other parts of the operating system to be implemented as it does not impose a lot of policies.

- Microkernel's and their user environments are usually implemented in the C++ or C programming languages with a little bit of assembly. However, other implementation languages are possible with some high-level coding.
- Microkernel fulfils basic operations like memory, process scheduling mechanisms, and inter-process communication.
- Microkernel is the only software executing at the privileged level. The other important functionalities of the OS are removed from the kernel-mode and run in the user mode. These functionalities may be device drivers, application, file servers, inter-process communication, etc.
- The structure of microkernel is shown in following figure

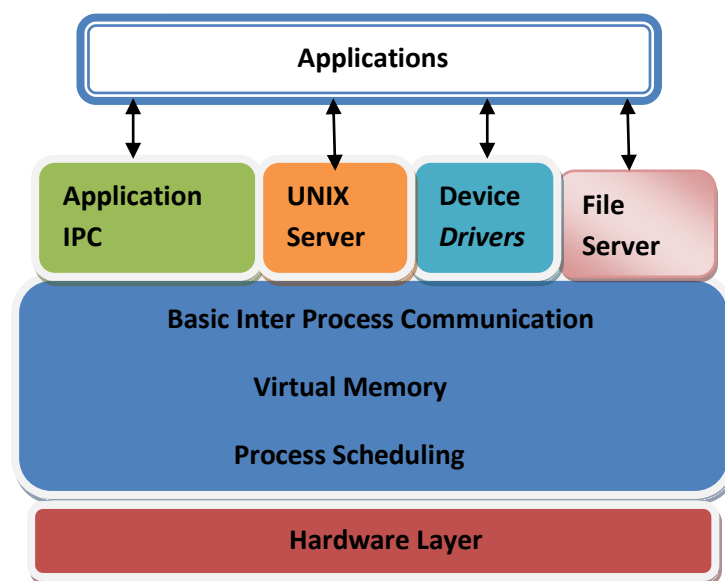


Figure 2.4 Microkernel structure

- **Components of Microkernel**

A microkernel comprises only the core functionalities of the system. A component is included in the Microkernel only if putting it outside would interrupt the functionality of the system. All other non-essential components should be put in the user mode.

- **The minimum functionalities required in the Microkernel are:**

1. Memory management mechanisms like address spaces should be included in the Microkernel. It also contains memory protection features.
2. Processor scheduling mechanisms should contain process and thread schedulers.
3. Inter-process communication manages the servers that run their own address spaces.

- **Advantages of Microkernel**

1. Microkernel architecture is small and isolated therefore it can function better.
2. Microkernels are secure because only those components are included that disrupt the functionality of the system otherwise.
3. The expansion of the system is more accessible, so it can be added to the system application without disturbing the Kernel.
4. Microkernels are modular, and the different modules can be replaced, reloaded, modified without even touching the Kernel.
5. Fewer system crashes when compared with monolithic systems.
6. Server malfunction is also isolated as any other user program's malfunction.
7. Increased security and stability will result in a decreased amount of code which runs on kernel mode

- **Disadvantage of Microkernel**

1. Providing services in a microkernel system are expensive compared to the normal monolithic system.
2. Context switch or a function call needed when the drivers are implemented as procedures or processes, respectively.
3. The performance of a microkernel system can be indifferent and may lead to some problems.

### **2.3.5Exo-kernel Operating system**

- Exo-kernel is an Operating System developed by the Massachusetts Institute of Technology (MIT) with the concept of putting the application in control. Exo-kernel operating systems seek to provide application level management of hardware resources. The architecture given in this operating system is designed to separate the resource protection from management, to facilitate application-specific customization.
- They differ from the other types of Kernels in that their functionality is limited to the protection and multiplexing of the raw hardware, and they provide no hardware abstractions on top of which applications can be constructed. This separation of hardware protection from hardware management enables application developers to determine how to make the most efficient use of the available hardware for each specific program.
- The user mode processes running in this type of Kernel has the ability to access Kernel resources like process tables, etc. directly.
- Exo-kernels in themselves they are extremely small. However, they are accompanied by library operating systems, which provide application developers with the conventional functionalities of a complete operating system.
- A major advantage of Exo-kernel-based systems is that they can incorporate multiple library operating systems, each exporting a different API, such as one for Linux and one for Microsoft Windows, thus making it possible to simultaneously run both Linux and Windows applications,

When compared to traditional kernels like Micro-kernels and Monolithic Kernels, Exo-kernels are very small, and they provide direct access to hardware by removing unnecessary abstractions.

## 2.4 Virtual Machine

- **Virtual Machine** abstracts/separates the hardware of our personal computer such as CPU, disk drives, memory, NIC (Network Interface Card) etc. into many different execution environments as per our requirements, hence giving us a feel that each execution environment is a single computer. For example, VirtualBox.
- When we run different processes on an operating system, it creates an illusion that each process is running on a different processor having its own virtual memory, with the help of CPU scheduling and virtual-memory techniques. There are additional features of a process that cannot be provided by the hardware alone like system calls and a file system. The virtual machine approach does not provide these additional functionalities but it only provides an interface that is same as basic hardware. Each process is provided with a virtual copy of the underlying computer system.
- We can create a virtual machine for several reasons, all of which are fundamentally related to the ability to share the same basic hardware yet can also support different execution environments, i.e., different operating systems simultaneously.
- The main drawback with the virtual-machine approach involves disk systems. Let us suppose that the physical machine has only three disk drives but wants to support seven virtual machines. Obviously, it cannot allocate a disk drive to each virtual machine, because virtual-machine software itself will need substantial disk space to provide virtual memory and spooling. The solution is to provide virtual disks.
- Users are thus given their own virtual machines. After which they can run any of the operating systems or software packages that are available on the underlying machine. The virtual-machine software is concerned with multi-programming multiple virtual machines onto a physical machine, but it does not need to consider any user-support software. This arrangement can provide a useful way to divide the problem of designing a multi-user interactive system, into two smaller pieces.

### Advantages:

1. There are no protection problems because each virtual machine is completely isolated from all other virtual machines.
2. Virtual machine can provide an instruction set architecture that differs from real computers.
3. Easy maintenance, availability and convenient recovery.

### Disadvantages:

1. When multiple virtual machines are simultaneously running on a host computer, one virtual machine can be affected by other running virtual machines, depending on the workload.
2. Virtual machines are not as efficient as a real one when accessing the hardware.

## 2.5 Booting

**Booting** is a start-up sequence that starts the **operating system** of a computer when it is turned on. A boot sequence is the initial set of operations that the computer performs when it is switched on. Every computer has a boot sequence. The BIOS, operating system and hardware components of a computer system should all be working correctly for it to boot. If any of these elements fail, it leads to a failed boot sequence.

### System Boot Process

The following diagram demonstrates the steps involved in a system boot process –

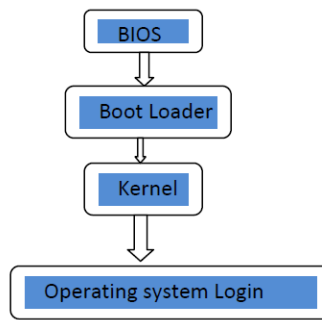


Figure2.5 : System boot process

Here are the steps –

- The CPU initializes itself after the power in the computer is first turned on. This is done by triggering a series of clock ticks that are generated by the system clock.
- After this, the CPU looks for the system's ROM BIOS to obtain the first instruction in the start-up program. This first instruction is stored in the ROM BIOS and it instructs the system to run POST (Power onSelf-Test) in a memory address that is predetermined.
- POST first checks the BIOS chip and then the CMOS RAM. If there is no battery failure detected by POST, then it continues to initialize the CPU.
- POST also checks the hardware devices, secondary storage devices such as hard drives, ports etc. And other hardware devices such as the mouse and keyboard. This is done to make sure they are working properly.
- After POST makes sure that all the components are working properly, then the BIOS finds an operating system to load.
- In most computer systems, the operating system loads from the C drive onto the hard drive. The CMOS chip typically tells the BIOS where the operating system is found.
- The order of the different drives that CMOS looks at while finding the operating system is known as the boot sequence. This sequence can be changed by changing the CMOS setup.
- After finding the appropriate boot drive, the BIOS first finds the boot record which tells it to find the beginning of the operating system.
- After the initialization of the operating system, the BIOS copies the files into the memory. Then the operating system controls the boot process.
- In the end, the operating system does a final inventory of the system memory and loads the device drivers needed to control the peripheral devices.
- The users can access the system applications to perform various tasks.

Without the system boot process, the computer users would have to download all the software components, including the ones not frequently required. With the system boot, only those software components need to be downloaded that are legitimately required and all extraneous components are not required. This process frees up a lot of space in the memory and consequently saves a lot of time.



## Unit 3 Information Management

### 3.1 Introduction

Different types of services of operating system which provides for reading and writing records are to be there. Operating system can be measured to be a collection of various such callable programs or services. These services are categorized by three heads.

- Information Management (IM)
- Process Management(PM)\*
- Memory Management(MM)

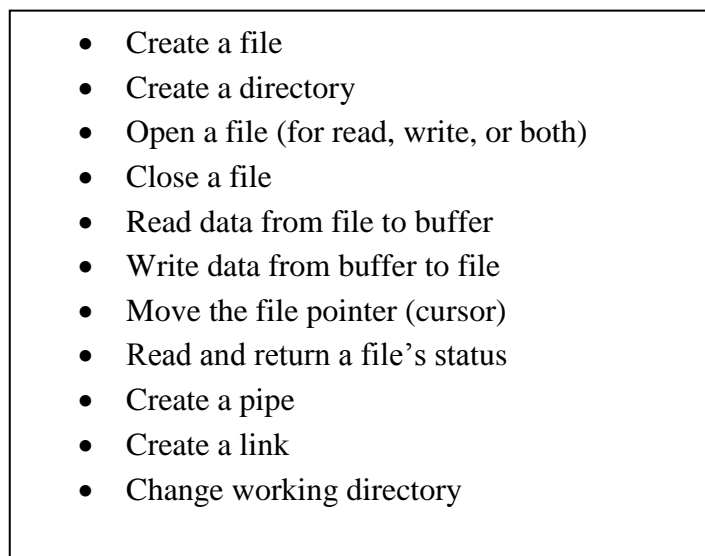
So let's focus more on Information Management in this chapter.

Information management (IM) is the collection and management of information from one or more sources and the allocation of that information to one or more addressees. This sometimes involves those who have a chance in or a right to that information.

Operating system manages the information or data, or managing the information from which it should be sent or received. File system in operating system keeps the track of information, its location and everything.

It also decides on which user should get resources first and put into effect the protection requirements and also provides accessing routines. Also necessary information like opening a file, reading a file is taken care by the file system and finally it retrieve the resources like closing a file.

The organization of the information in terms of directories and files, allocating and deallocating the sectors of different types of files, maintaining and enforcing the access controls to ensure that only the right people can have access to the information, and driving various devices are done by these types of system services. These are normally provided by a single user operating system (i.e. Windows 95) and also multi user operating system (i.e. Windows 2000, UNIX, Virtual Memory system/VMS, multiple virtual storage /MVs). So some of the system calls in this category are listed in following Figure.

- 
- Create a file
  - Create a directory
  - Open a file (for read, write, or both)
  - Close a file
  - Read data from file to buffer
  - Write data from buffer to file
  - Move the file pointer (cursor)
  - Read and return a file's status
  - Create a pipe
  - Create a link
  - Change working directory

**Figure 3.1 System calls related to Information Management**

### 3.1.1 Information Maintenance

Many system calls exist simply for the purpose of transferring information between the user program and the operating system. For example, most systems have a system call to return the current time and date. Other system calls may return information about the system, such as the number of current users, the version number of the operating system, the amount of free memory or disk space, and so on. In addition, the operating system keeps information about all its processes, and system calls are used to access this information. Generally, calls are also used to reset the process information (get process attributes and set process attributes).

### 3.2 Disk Basics

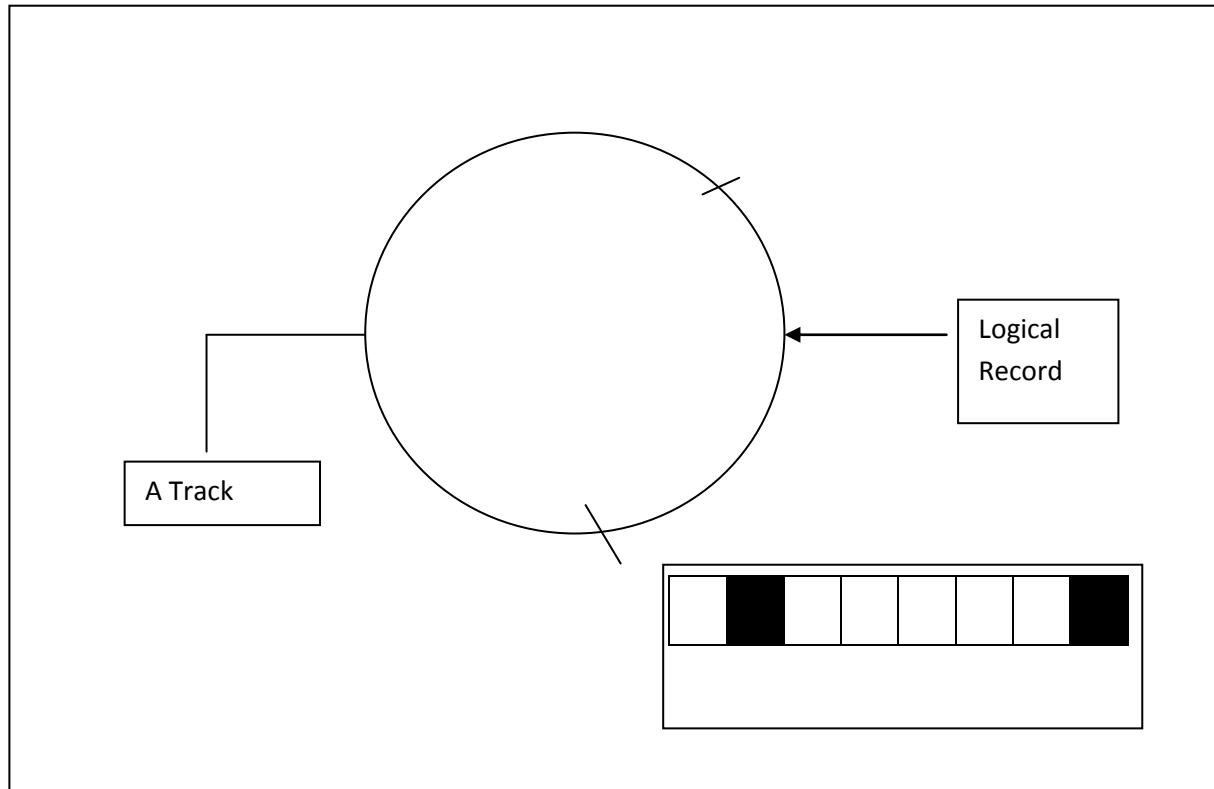
Alternatively known as a floppy disk, a disk is a hard or floppy round, flat, and magnetic platter capable of having information read from and written to it. The most frequently found disks with a computer are the hard disks and floppy disks.

Disk is a very important I/O intermediate used by operating system frequently so it is necessary to understand the functions of it. The operating principle of a floppy disk is similar to that of hard disk. And a hard disk can be considered as being made of multiple floppy disks put one above the other.



**Figure 3.2 Disk** (Source: Google/ computerhope.com)

**Figure 3.3 Data Recording on the Disk**



Disk is nothing but long play music records except that the recording is done in concentric circles and not by spirally. A Floppy disk is made up of a round piece of plastic material, coated with a magnetized recording material. The surface is made up of concentric circles which are called as tracks. Data is recorded on these tracks in bit and it contains magnetized particles of metal is having north and South Pole. It is having only two directions so each particle can acts as a binary values of 0 or 1 and 8 switches can record a character with the coding methods (ASCII). A logical record which consists of different fields (data items), each consisting of several characters which is stored in floppy disk.

Generally, a disk is a round plate on which data can be encoded. There are two basic types of disks: *magnetic disks* and *optical disks*.

### 3.2.1 Magnetic disks

On top of magnetic disks, data is encoded as microscopic magnetized needles on the disk's surface. One can record and erase data on a magnetic disk any number of times, just as with a cassette tape. Magnetic disks come in a number of different forms:

- **Floppy disk:** A typical 5 inch floppy disk can hold 360K or 1.2MB (megabytes). 3 1/2-inch floppies normally store 720K, 1.2MB or 1.44MB of data. Floppy disks are outdated today, and are found on older computer systems.
- **Hard disk:** Hard disks can store anywhere from 20MB to more than 1-TB (terabyte). Hard disks are also from 10 to 100 times faster than floppy disks.
- **Removable cartridge:** Removable cartridges are hard disks covered in a metal or plastic cartridge, so you can remove them just like a floppy disk. Removable cartridges are very fast, though usually not as fast as fixed hard disks.

### 3.2.2 Optical disks

Optical disks record data by burning microscopic holes in the surface of the disk with a laser. To read the disk, another laser beam shines on the disk and detects the holes by changes in the reflection pattern.

Optical disks come in three basic forms:

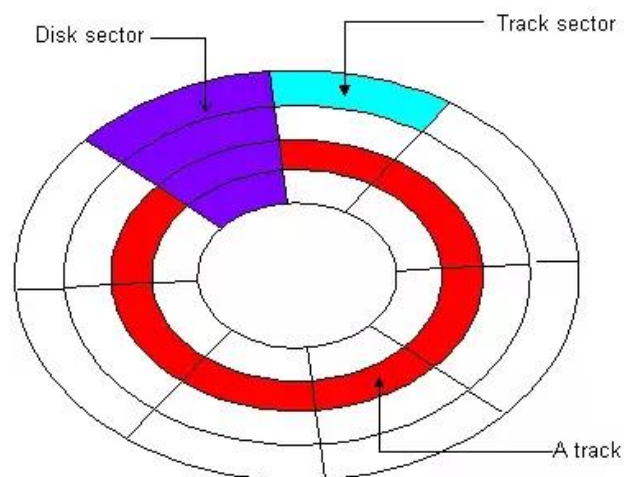
- **CD-ROM:** Most optical disks are read-only. When you purchase them, they are already filled with data. You can read the data from a CD-ROM, but you cannot modify, delete, or write new data.
- **WORM:** Stands for write-once, read-many. WORM disks can be written on once and then read any number of times; however, you need a special WORM drive to write data onto a WORM disk.

- **Erasable optical (EO):** EO disks can be read to, written to, and erased just like magnetic disks.

The machine that turns a disk is called a disk drive. Within each disk drive is one or more heads (often called read/write heads) that actually read and write the data.

Accessing data from a disk is not as fast as accessing data from main memory, but disks are much cheaper. And unlike RAM, disks hold on to data even when the computer is turned off. Accordingly, disks are the storage medium of choice for most types of data. Another storage medium is magnetic tape. But tapes are used only for backup and archiving because they are sequential-access devices (to access data in the middle of a tape, the tape drive must pass through all the previous data).

A disk can be considered several surfaces, each of which consisting number of tracks which is shown in figure 3.4. The tracks are normally numbered from 0 as outermost track, with the number increasing inwards. Each track is divided into a number of sectors which are having equal size. And a sector can store up to 512 bytes. Double sided floppies have two sides or surfaces on which data can be recorded. So a given sector is specified by three components of an address made up of surface number, track number and sector number.



**Figure 3.4 Tracks and Sectors (Source: [installsetupconfig.com](http://installsetupconfig.com))**

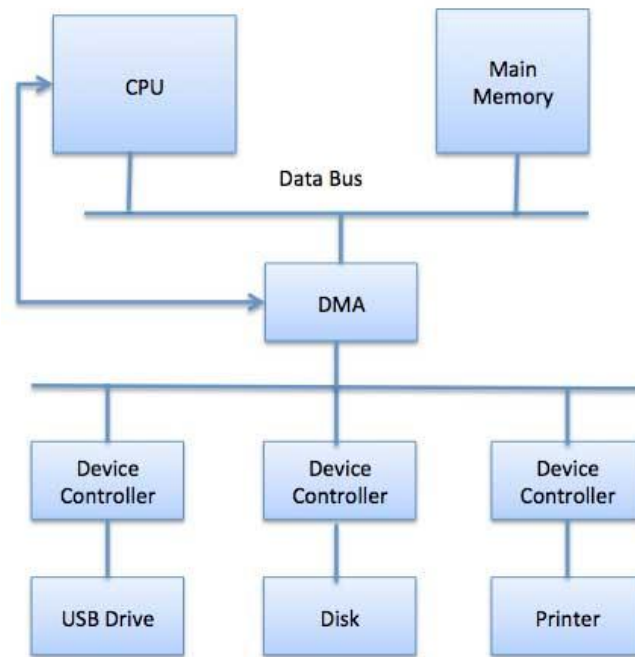
### **3.3 Direct Memory Access (DMA)**

Slow devices like keyboards will generate an interrupt to the main CPU after each byte is transferred. If a fast device such as a disk generated an interrupt for each byte, the operating system would spend most of its time handling these interrupts. So a typical computer uses direct memory access (DMA) hardware to reduce this overhead.

Direct Memory Access (DMA) means CPU grants I/O module authority to read from or write to memory without involvement. DMA module itself controls exchange of data between main memory and the I/O device. CPU is only involved at the beginning and end of the transfer and interrupted only after entire block has been transferred.

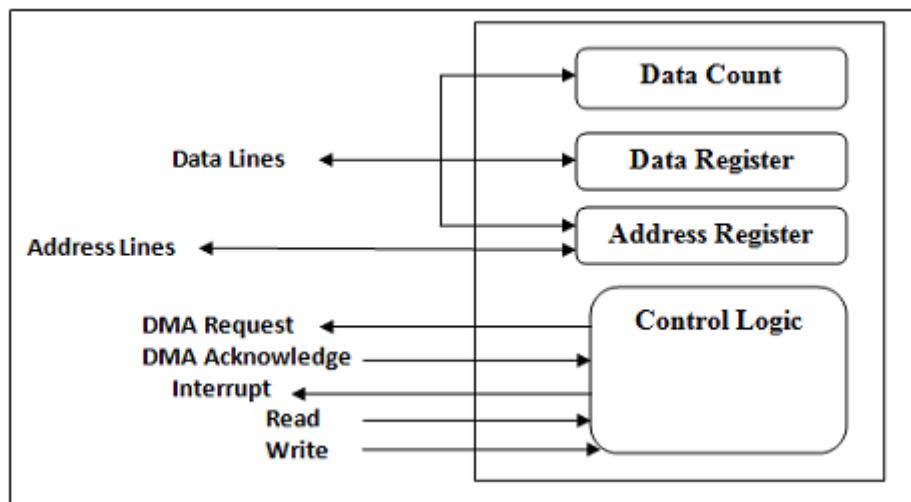
For example, a sound card may need to access data stored in the computer's RAM, but since it can process the data itself, it may use DMA to bypass the CPU. Video cards that support DMA can also access the system memory and process graphics without needing the CPU.

Direct Memory Access needs a special hardware called DMA controller (DMAC) that manages the data transfers and arbitrates access to the system bus. The controllers are programmed with source and destination pointers (where to read/write the data), counters to track the number of transferred bytes, and settings, which includes I/O and memory types, interrupts and states for the CPU cycles.

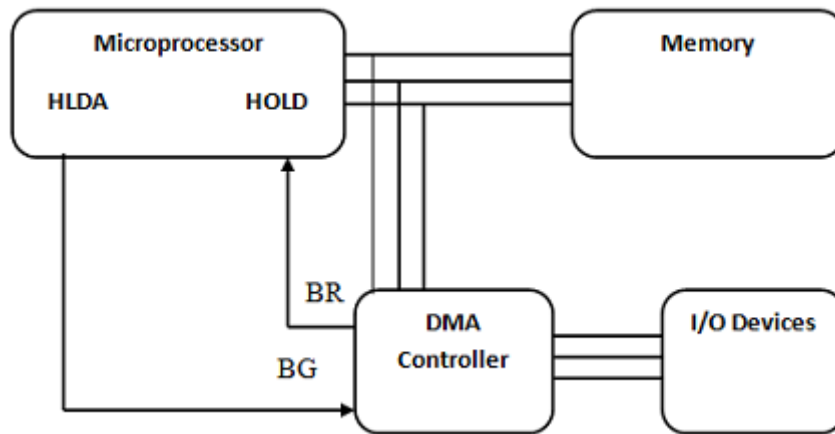


**Figure 3.5 Direct Memory Access**

### 3.3.1 Block diagram of DMA controller and DMA Operations



**Figure 3.6 Block Diagram of DMA Controller**



**Figure 3.7 DMA Control Operations**

### **3.3.2 DMA Operation:**

Direct Memory Access involves transfer of data between I/O devices and memory by an external circuitry system called DMA controller without involving the microprocessor. However, microprocessor itself initiates the DMA control process by providing starting address, size of data block and direction of data flow. DMA contains a control unit to deal with the control functions during DMA operations such as read, write and interrupt. The address register of DMA controller is used to generate address and select I/O device to transfer the data block. The Count registers counts and hold no. of data block transferred. It also specifies direction of data transfer.

### **Steps of DMA Operation:**

- a) For DMA operation to occur, the DMA controller first make a bus request (BR) by sending a control signal HOLD to the control line.
- b) On receiving the BR through HOLD pin high, the microprocessor completes the current instruction execution and afterward it generates HLDA control signal and sends it to the DMA-Controller. This event switches over the control from microprocessor to DMA Controller. The microprocessor gets idle.



c) As soon as DMA controller receives HLDA (Hold Acknowledged) through Bus Grant (BG) line, it takes the control of system bus and start transferring the data blocks between memory and Input / Output devices, without involving the microprocessor.

d) On completion of data transfer, the DMA controller sends a low signal to the HOLD pin and hence microprocessor makes the HLDA pin low and takes the control over system bus.

### **DMA Operation Modes:**

The DMA Controller operates under three modes:

A. **Burst Mode:** Here DMA controller switch over the control to the microprocessor only on completion of entire data transfer, irrespective of microprocessor requiring the bus. Microprocessor has to be idle during the data transfer.

B. **Cycle-Stealing Mode:** DMA controller hand over the control to microprocessor on transfer of every byte, thereby microprocessor gets the control and become able to process highly prioritized instruction. DMA need to make the BG request for each byte.

C. **Transparent Mode:** In this mode, DMA controller can transfer data blocks only when microprocessor are executing such instruction that does not requires system bus utilization.

### **3.3.3 Direct Memory Access Advantages and Disadvantages**

#### **Advantages:**

1. Transferring the data without the involvement of the processor will **speed up** the read-write task.
2. DMA **reduces the clock cycle** requires to read or write a block of data.
3. Implementing DMA also **reduces the overhead** of the processor.

## **Disadvantages**

1. As it is a hardware unit, it would **cost** to implement a DMA controller in the system.
2. Cache **consistency** problem can occur while using DMA controller.

## **3.4 File System**

In our daily lives we use files. Normally a file contains records of similar types of information. E.g. Employee file or Sales file or Electricity bill file. If anyone wants to automate various manual functions then the computer must support a facility for a user to define and manipulate the files. And the operating system does these things.

### **3.4.1 Block and Block numbering Scheme**

In some databases, a block is the smallest amount of data that a program can request. It is a multiple of an operating system block, which is the smallest amount of data that can be retrieved from storage or memory. Multiple blocks in a database comprise an extent.

The operating system looks at a hard disk as a series of sectors, and numbers them serially starting from 0 to 1. One of the possible ways of doing this is shown in following figure which shows a hard disk.

If we consider all the tracks on different surfaces, which are of the same size, we can think of them as a cylinder due to the obvious similarity in shape. In such a case, a sector address can be thought of us having three components such as: Cylinder number. Surface number, Sector number. In this case, Cylinder number is same as track number used in the earlier scheme where the address consisted of Surface number. Track number and Sector number. Therefore, both these schemes are equivalent. The figure shows four platters and therefore, eight surfaces, with 10 sectors per track. The numbering starts with 0 (may be aligned with index

hole in case of a floppy), at the outermost cylinder and topmost surface. We will assume that each sector is numbered anticlockwise so that if the disk rotates clockwise, it will encounter sectors 0, 1, 2 etc, in that sequence.

When all the sectors on that surface on that cylinder are numbered, we go to the next surface below on the same platter and on the same cylinder. This surface is akin to the other side of a coin. After both the surfaces of one platter are over, we continue with other platters for the same cylinder in the same fashion. After the full cylinder is over, we go to the inner cylinder, and continue from the top surface again. By this scheme, Sectors 0 to 9 will be on the topmost surface (i.e. surface number = 0) of the outermost cylinder (i.e. cylinder number = 0). Sector 10 to 19 will be on the next surface (at the back) below (i.e. surface number = 1), but on the same platter and the same cylinder (i.e. cylinder number = 0). Continuing this, with 8 surfaces (i.e. 8 tracks/cylinder), we will have Sectors 0-79 on the outermost cylinder (i.e. Cylinder 0). When the full cylinder is over, we start with the inner cylinder, but from the top surface and repeat procedure. Therefore the next cylinder (Cylinder = 1) will have sector 80 to 159, and so on.

Likewise, we can view the entire disk as a series of sectors starting from 0 to n as Sector Numbers (SN) as shown.

0	1	2	3.....	N
---	---	---	--------	---

### 3.4.2 File support Levels

The Operating System is responsible for the translation from logical to physical level for an Application program. In those days, an application programmer had to specify Operating System the actual disk address (cylinder, surface, and sector) to access a file. If he wanted to access a specific customer record, he had to write routines to keep track of where that record

resided and then he had to specify this address. Each application programmer had a lot of work to do, and the scheme had a lot of problems in terms of security, privacy and complexity.

Ultimately, somebody had to translate from the logical to the physical level. The point was who should do it? Should the Application Program do it or should the Operating System do it? The existing Operating Systems have a great deal of differences in answering this question. Some (like UNIX) treat a file as a Sequence of bytes. This is one extreme of the spectrum where the Operating System provides the minimum support. In this case, the Operating System does not recognize the concept of a record.

Therefore, the record length is not maintained as a file attribute in the file system of such an Operating System like UNIX or Windows 2000. Such an Operating System does not understand an instruction such as `fread` in C or "Read... record" in COBOL. It only understands an instruction "Read byte numbers X to Y". Therefore, something like the application program or the DBMS uses has to do the necessary conversion. At a little higher level of support, some others treat files as consisting of records of fixed or variable length, like AOS/VS). In this case, the Operating System keeps the information about record lengths etc. along with the other information about the file.

### **3.4.3 Writing/Reading a Record**

#### **3.4.3.1 Writing a Record**

File system is responsible for translating the address of logical records into physical address. So how this is achieved by taking simple example of a sequential file i.e. customer records. Let us see step by steps.

1. **Arrangement of logical records:** Consider customer record consists of 700 bytes. The application program is responsible for creating these records in HLL such as “WRITE CUST-REC” to achieve this. At the time of execution, this results in a system call to the operating system to write a record. These records may or may not be in any specific sequence (as customer number). The OS assigns a Relative Record Number (RRN) to each of the record which is starting with 0, as these records are written.

Imagine all the customer records put one after another like carpet shown in following figure 3.8.

PRN=0	PRN = 1	.....	PRN = 9
BYTES 0-699	BYTES 700-1399		BYTES 6300-6999

**Figure 3.8 : The PRN and RBN**

If 10 customer records are there the  $700 \times 10 = 7000$  bytes will be written onto the customer for PRN = 0 to 9. The operating system can calculate a Relative byte number (RBN) for every record. This is the starting byte number for each record. RBN is calculated with respect to 0 as the starting byte number and put after other as logical records. Following figure shows the relationship between RRN and RBN for the records shown in above figure.

RRN	RBN
0	0
1	700
2	1400
.	...

9	6300
---	------

**Figure 3.9: The relation between RRN and RBN**

**2. Arrangement of Blocks:** The second step is how o actually write these logical records into different types of blocks. Let us consider a disk with 8 surfaces (0 to 7). Each surface is having 80 tracks (0 to 79) and each track is having 10 sectors (0 to 9). So the disk has  $8*80*10 = 6400$  sectors of 512 bytes each. 1 block = 1 sector = 512 bytes. So the operating system looks at the disk as consisting of 6400 logical blocks (0 to 6399) each of 512 bytes.

Block = 0	Block = 1	Block = 6399
Bytes 0 to 511	Bytes 512 to 1023	Bytes 3276288 to 3276800

**Figure 3.10: The operating system view on Disk**

**3. The allocation of blocks to a file:** Operating system is acting kike arbitrator and is responsible for allocating/deallocating blocks to different files. Three ways are there, i.e. Contiguous, Indexed and Chained. If a user knows that he has 500 customers, but will never have more than 730, he must ask for  $730*700/512$  blocks = 998.05 or approximately 1000 blocks. Let us consider that the operating system has already allocated block numbers 0 to 99 for some other file and that blocks 100 and thereafter are free. The OS has to maintain the list of free blocks and it allocates block numbers 100 to 1099 for the customer file.

### 3.4.3.2 Reading a Record

How the records are to be read, let us consider by the Operating system on the request of the AP. An unstructured AP for processing all the records sequentially from a file would be

shown in following figures 3.11 and 3.12 for C and COBOL. At the time of processing, it executes the “fread” or “READ” instruction respectively; the AP gives a call to the operating system which then reads a logical record on behalf of the AP. The operating system blocks the AP during this time, after which it is woken up.

```
ABC.  
  
    Count = fread (&custrec, sizeof (custrec), 1 FP);  
  
    if (count == 0) goto EOJ;  
  
    ...  
  
    /* calculate the balance, interest, etc. */  
  
    Process_record ( );  
  
    Goto ABC;  
  
EOJ.  
  
    Exit ( );
```

**Figure 3.11: A 3GL program to read and process records (C version)**

```
ABC.  
  
    READ CUST-REC..... AT END GO TO EOJ.  
  
    ...  
  
    PERFORM PROCESS-REC.  
  
        (calculate the balance, interest etc.)  
  
    GO TO ABC.  
  
EOJ.  
  
    STOP RUN.
```

**Figure 3.12: A 3GL program to read and process records (COBOL version)**

### **3.4.4 Relationship between the Operating System and DMS**

Operating system can present to the Application Program (AP), records in the same sequence that they have been written. If other AP wanted to process the records in a sequence, say by customer number, it is the work of AP to ensure that they are presented to the operating system in that fashion so that they also retrieved.

If some records are to be selectively processed in a sequence then it is required to maintain some data structure like an index for indicate that where a specific record is written. So there is another piece of software to maintain and access these types of indexes and it is known as Data Management Systems (DMS).

The data management functions such as maintenance of an index, etc. were parts of operating system, but as these functions started getting more complex, separate DMS were written. The DMS can be either File Management System (FMS) or Database Management System (DBMS). ORACLE, DB2, INFORMIX are some of the examples of DBMS. DMS is in between the AP and OS and which is responsible for maintaining all the index tables based on keys.

### **3.4.5 File Directory Entry**

Group of files combined is known as directory. A directory contains all information about file, its attributes. The directory can be viewed as a symbol table that translates file names into their directory entries. Directory itself can be organized in many ways.

The organization must allow us to insert entries, to delete entries, to search for a named entry, and to list all the entries in the directory. File directory entry describes files and directories. It



is a primary critical directory entry and must be immediately followed by 1 Stream Extension directory entry and from 1 to 17 File Name directory entries.

For each file created by the operating system, the OS maintains a directory entry which is also known as Volume Table of Contents (VTOC) in IBM. Let us consider following figure shows the possible information that the operating system keeps at a logical level for each file. Physically, it could be kept as one single record or multiple records, where access control information constitutes one small record. The logical records of VTOC or file directory entries are stored on the disk using some hashing algorithm on the file name. Alternatively, an index on the file names can be maintained to allow faster access of this particular information once the file is supplied. But in UNIX, the hashing technique is not used. All entries are created sequentially or in first available empty slot. So consistently the operating system goes through the entire directory starting from the first entry to access the directory entry given a file name. The algorithm is very simple and time consuming for execution.

File Name
File Extension
File size – Current and maximum
File Usage Count
Number of processes having this file opens
File Type (binary, ASCII, etc)
File Record Length (If records are recognized)
File Key length, positions (If ISAM is a part of operating system)
File organization (Sequential, Indexed, Random)
File Owner, Creator (Usually the same)
File Access Control Information
File Dates (Creation, last usage etc.)
Other information
File Address (Block Number) of the first block

**Figure 3.13: File Directory Entry or VTOC**

The significance for address translation is the file address field for every file. This signifies the address (i.e. block number) of the first block allocated to the file. If the allocation is

adjacent, finding out the address of the subsequent blocks is very easy. And if the allocation is chained or indexed, the operating system has to pass through the data structure to access the subsequent blocks of the same file.

### **3.4.6 Open/Close Operations**

File operation are simply those things which user can perform on a file. For example user can create file, save a file, open a file, read a file and modify a file. OS can provide system call for performing various operations on the file are: create, write, read, delete, re positioning and translating.

Following are same list of operations.

1. Create.
2. Write.
3. Close.
4. Open.
5. Read.
6. Delete.

All these operation require that the directory structure be first search for the target file.

#### **Open ( )**

- A file can be opened in one of the two modes, read mode or write mode.
- In read mode, the OS does not allow anyone to alter data; file opened in read mode can be stored among several entities.
- Write mode allows data modifications, file opened in write mode can be read but cannot be shared.

#### **Close ( )**

- This is the most important operation from OS point of view.
- When a request to close a file is generated the OS.

- OS removes all the block (in shared mode)
- Saves the data if altered to the secondary storage media.
- Releases all the buffer and file handlers associated with a file.

### 3.4.7 Disk Space Allocation Methods

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

- Contiguous Allocation
- Linked Allocation
- Indexed Allocation

The main idea behind these methods is to provide:

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

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

#### 1. Contiguous Allocation

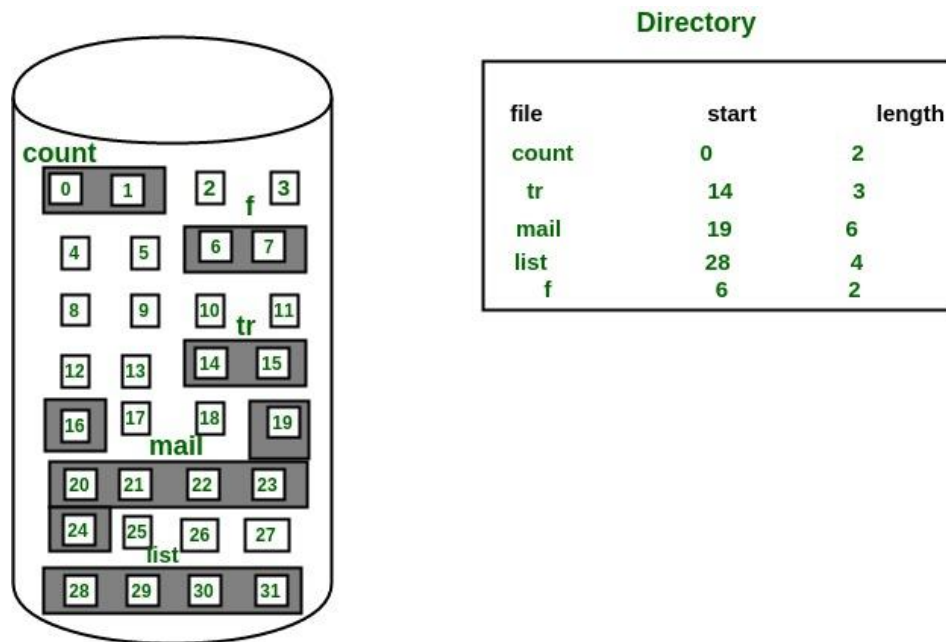
In this scheme, each file occupies a contiguous set of blocks on the disk. For example, if a file requires  $n$  blocks and is given a block  $b$  as the starting location, then the blocks assigned to the file will be:  $b, b+1, b+2, \dots, b+n-1$ . This means that given the starting block address and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

The directory entry for a file with contiguous allocation contains

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

The file 'mail' in the following figure starts from the block 19 with length = 6 blocks.

Therefore, it occupies 19, 20, 21, 22, 23, 24 blocks.



#### Advantages:

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

#### Disadvantages:

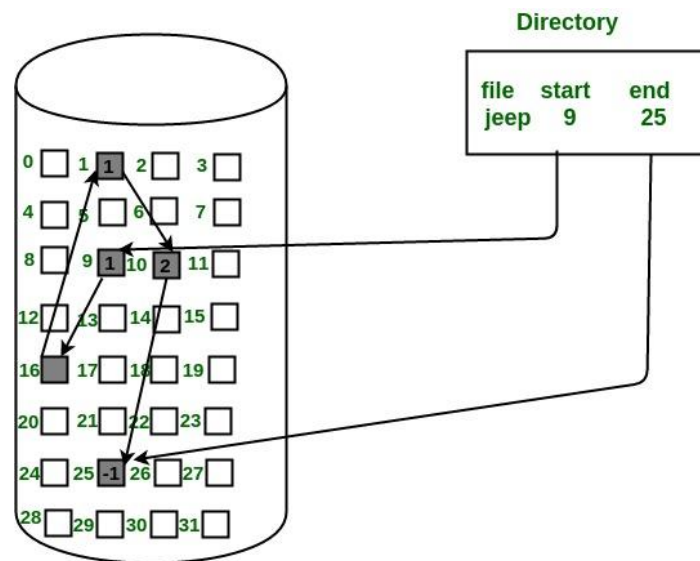
- This method suffers from both internal and external fragmentation. This makes it inefficient in terms of memory utilization.
- Increasing file size is difficult because it depends on the availability of contiguous memory at a particular instance.

## 2. Linked List Allocation

In this scheme, each file is a linked list of disk blocks which **need not be** contiguous. The disk blocks can be scattered anywhere on the disk.

The directory entry contains a pointer to the starting and the ending file block. Each block contains a pointer to the next block occupied by the file.

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



#### Advantages:

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

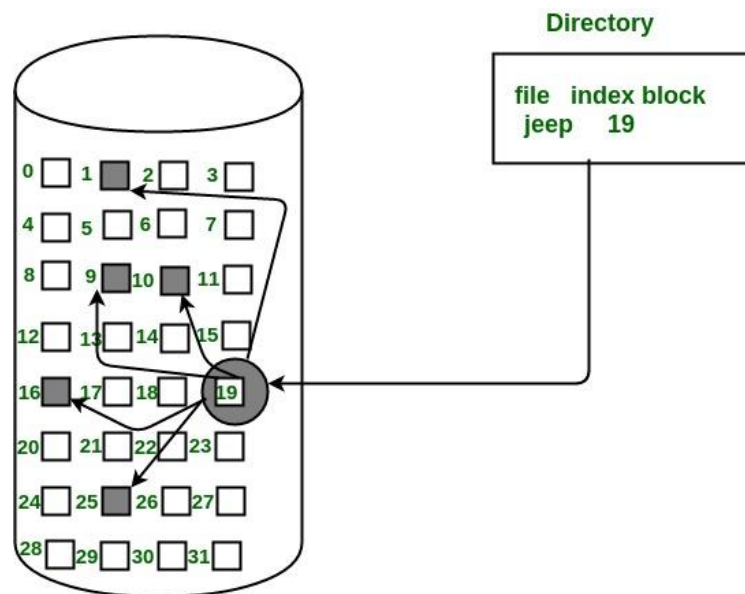
#### Disadvantages:

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

### 3. Indexed Allocation

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

contains the disk address of the  $i$ th file block. The directory entry contains the address of the index block as shown in the image:



#### Advantages:

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

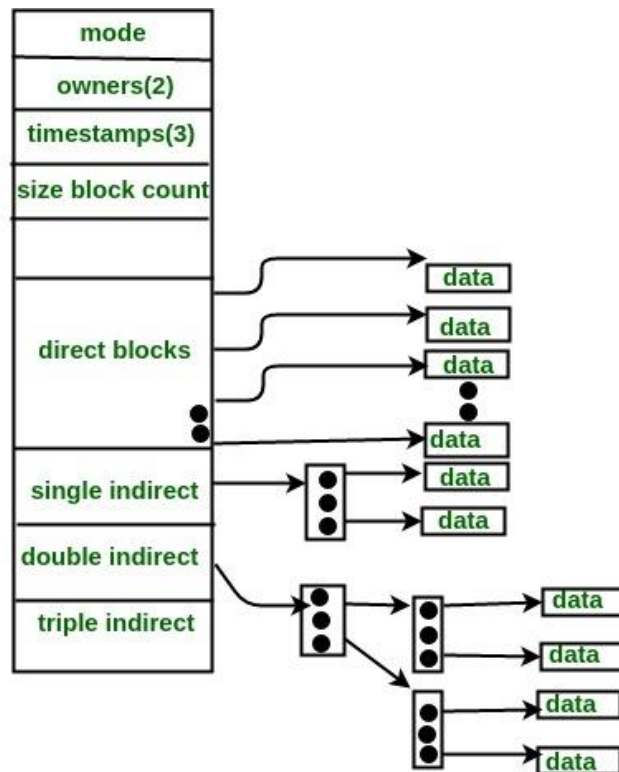
#### Disadvantages:

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

For files that are very large, single index block may not be able to hold all the pointers.

Following mechanisms can be used to resolve this:

1.     **Linked scheme:** This scheme links two or more index blocks together for holding the pointers. Every index block would then contain a pointer or the address to the next index block.
2.     **Multilevel index:** In this policy, a first level index block is used to point to the second level index blocks which in turn points to the disk blocks occupied by the file. This can be extended to 3 or more levels depending on the maximum file size.
3.     **Combined Scheme:** In this scheme, a special block called the I node (information Node) contains all the information about the file such as the name, size, authority, etc and the remaining space of I node is used to store the Disk Block addresses which contain the actual file as shown in the image below. The first few of these pointers in I node point to the direct blocks i.e. the pointers contain the addresses of the disk blocks that contain data of the file. The next few pointers point to indirect blocks. Indirect blocks may be single indirect, double indirect or triple indirect. Single Indirect block is the disk block that does not contain the file data but the disk address of the blocks that contain the file data. Similarly, double indirect blocks do not contain the file data but the disk address of the blocks that contain the address of the blocks containing the file data.

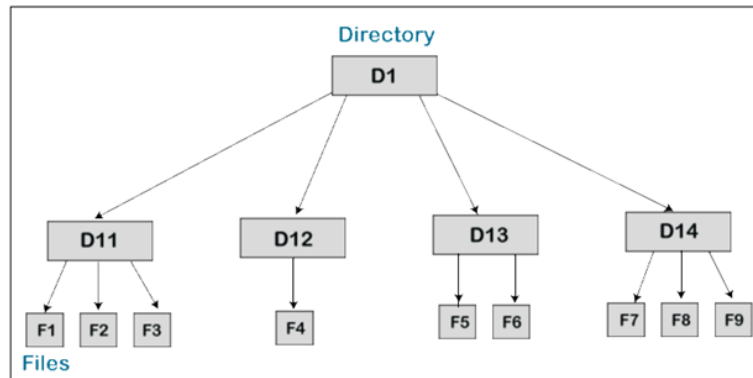


### 3.4.8 Directory Structure: User's View

By considering the user's viewpoint, a directory is a file of files, i.e. a single file containing details about other files belonging to that directory.

A Directory is the collection of the correlated files on the disk. In simple words, a directory is like a container which contains file and folder. In a directory, we can store the complete file attributes or some attributes of the file. A directory can be comprised of various files. With the help of the directory, we can maintain the information related to the files.





To take the advantages of various file systems on the different operating systems, we can divide the hard disk into multiple partitions, which are of different sizes. Partitions are known as minidisks or volumes.

There should be at least one directory that must be present in each partition. Through it, we can list all the files of the partition. In the directory for each file, there is a directory entry, which is maintained, and in that directory entry, all the information related to the file is stored.

There are various types of information which are stored in a directory:

1. Name
2. Type
3. Location
4. Size
5. Position
6. Protection
7. Usage
8. Mounting

1. **Name:** – Name is the name of the directory, which is visible to the user.
2. **Type:** – Type of a directory means what type of directory is present such as single-level directory, two-level directory, tree-structured directory, and acyclic graph directory.

3.       **Location:** – Location is the location of the device where the header of a file is located.
4.       **Size:** – Size means number of words/blocks/bytes in the file.
5.       **Position:** – Position means the position of the next-read pointer and the next-write pointer.
6.       **Protection:** – Protection means access control on the read/write/delete/execute.
7.       **Usage:** – Usage means the time of creation, modification, and access, etc.
8.       **Mounting:** – Mounting means if the root of a file system is grafted into the existing tree of other file systems.

### Operations on Directory

The various types of operations on the directory are:

1.       Creating
  2.       Deleting
  3.       Searching
  4.       List a directory
  5.       Renaming
  6.       Link
  7.       Unlink
1.       **Creating:** – In this operation, a directory is created. The name of the directory should be unique.
  2.       **Deleting:** – If there is a file that we don't need, then we can delete that file from the directory. We can also remove the whole directory if the directory is not required. An empty directory can also be deleted. An empty directory is a directory that only consists of dot and dot-dot.

3. **Searching:** – Searching operation means, for a specific file or another directory, we can search a directory.

4. **List a directory:** – In this operation, we can retrieve the entire files list in the directory. And we can also retrieve the content of the directory entry for every file present in the list.

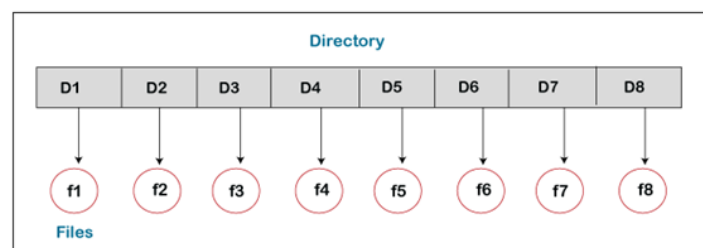
If in the directory, we want to read the list of all files, then first, it should be opened, and afterwards we read the directory, it is a must to close the directory so that the internal table space can be free up.

## Types of Directory Structure

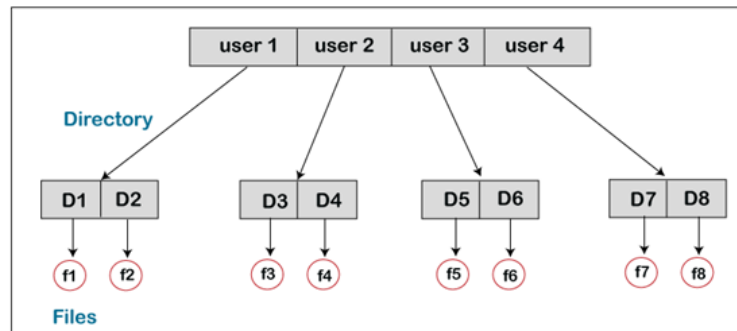
There are various types of directory structure:

1. Single-Level Directory
2. Two-Level Directory
3. Tree-Structured Directory
4. Acyclic Graph Directory
5. General-Graph Directory

**1. Single-Level Directory:** – Single-Level Directory is the easiest directory structure. There is only one directory in a single-level directory, and that directory is called a root directory. In a single-level directory, all the files are present in one directory that makes it easy to understand. In this, under the root directory, the user cannot create the subdirectories.

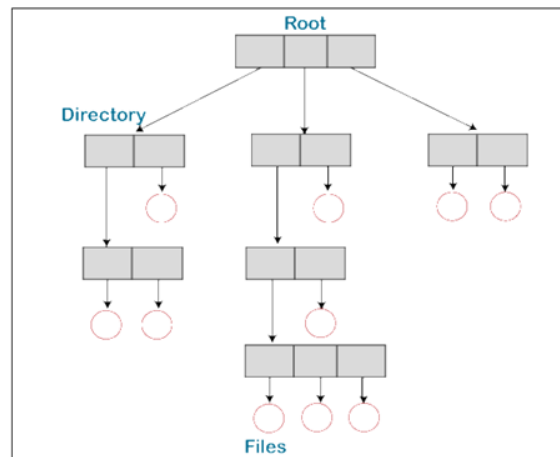


2. **Two-Level Directory:** Two-Level Directory is another type of directory structure. In this, it is possible to create an individual directory for each of the users. There is one master node in the two-level directory that includes an individual directory for every user. At the second level of the directory, there is a different directory present for each of the users. Without permission, no user can enter into the other user's directory.



3. **Tree-Structured Directory:** A Tree-structured directory is another type of directory structure in which the directory entry may be a sub-directory or a file. The tree-structured directory reduces the limitations of the two-level directory. We can group the same type of files into one directory.

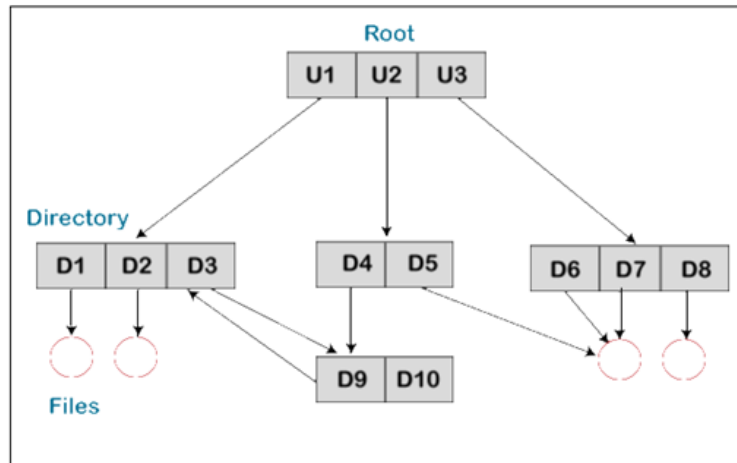
In a tree-structured directory, there is an own directory of each user, and any user is not allowed to enter into the directory of another user. Although the user can read the data of root, the user cannot modify or write it. The system administrator only has full access to the root directory. In this, searching is quite effective and we use the current working concept. We can access the file by using two kinds of paths, either absolute or relative.



**4. Acyclic-Graph Directory:** In the tree-structure directory, the same files cannot exist in the multiple directories, so sharing the files is the main problem in the tree-structure directory. With the help of the acyclic-graph directory, we can provide the sharing of files. In the acyclic-graph directory, more than one directory can point to a similar file or subdirectory. We can share those files among the two directory entries.

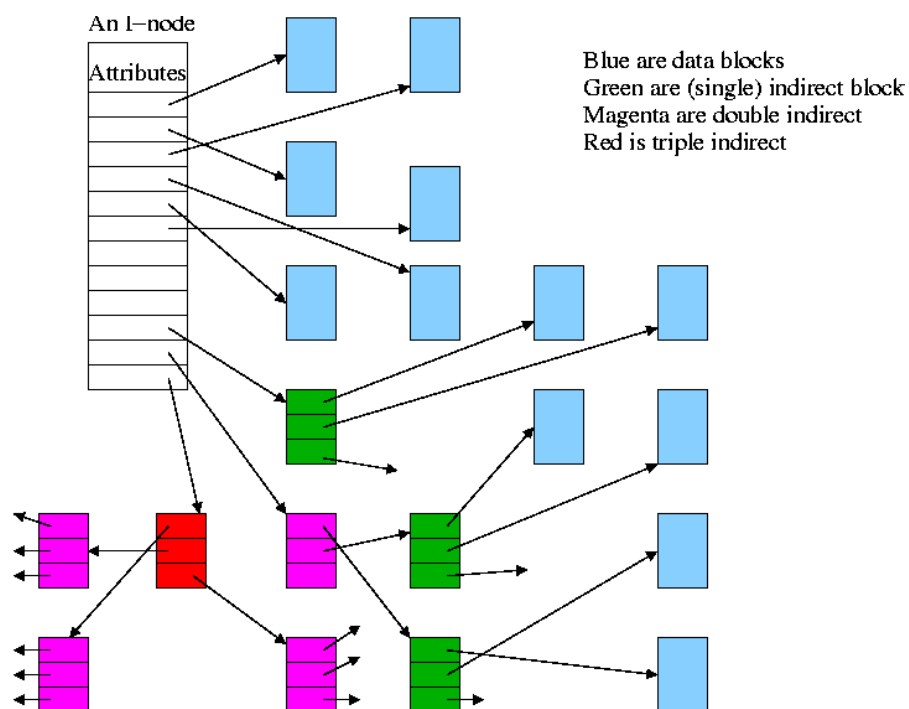
With the help of aliases, and links, we can create this type of directory graph. We may also have a different path for the same file. Links may be of two kinds, which are hard link (physical) and symbolic (logical).

**5. General-Graph Directory:** The General-Graph directory is another vital type of directory structure. In this type of directory, within a directory we can create cycle of the directory where we can derive the various directories with the help of more than one parent directory. The main issue in the general-graph directory is to calculate the total space or size, taken by the directories and the files.



### 3.4.9 Implementation of a Directory System

The selection of directory-allocation and directory-management algorithms significantly affects the efficiency, performance, and reliability of the file system.



## **Linear List**

The simplest method of implementing a directory is to use a linear list of file names with pointers to the data blocks. This method is simple to program but time-consuming to execute. 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, then release the space allocated to it.

To reuse the directory entry, we can do one of several things. We can mark the entry as unused (by assigning it a special name, such as an all-blank name, or with a used-unused, bit in each entry), or we can attach it to a list of free directory entries. A third alternative is to copy the last entry in the directory into the freed location and to decrease the length of the directory. A linked list can also be used to decrease the time required to delete a file.

The real disadvantage of a linear list of directory entries is that finding a file requires a linear search. Directory information is used frequently, and users will notice if access to it is slow. In fact, many operating systems implement a software cache to store the most recently used directory information. A cache hit avoids the need to constantly reread the information from disk. A sorted list allows a binary search and decreases the average search time.

However, the requirement that the list be kept sorted may complicate creating and deleting files, since we may have to move substantial amounts of directory information to maintain a sorted directory. A more sophisticated tree data structure, such as a B-tree, might help here. An advantage of the sorted list is that a sorted directory listing can be produced without a separate sort step.

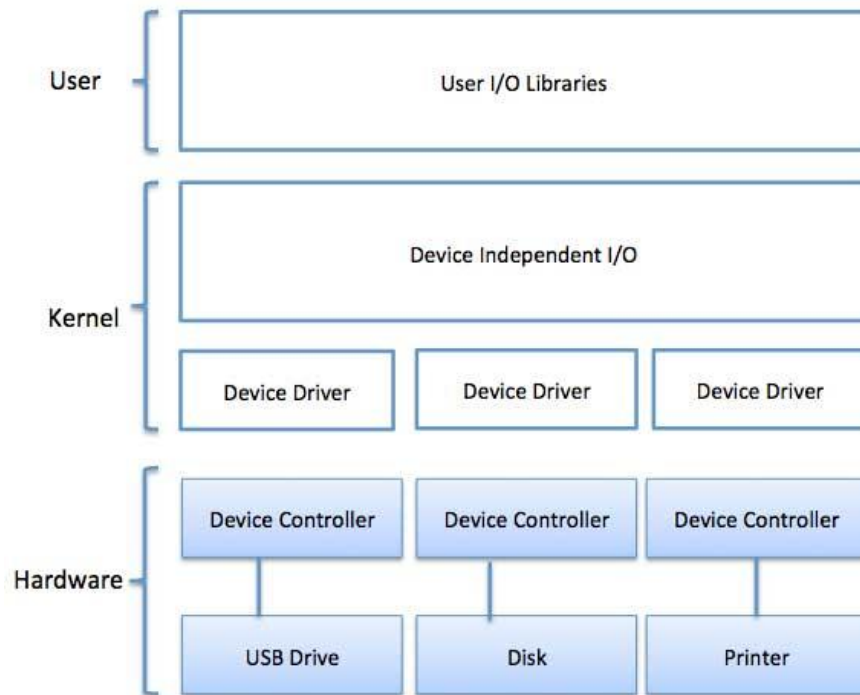
## **Hash Table**

Another data structure used for a file directory is a hash table. With this method, a linear list stores the directory entries, but a hash data structure is also used. The hash table takes a value computed from the file name and returns a pointer to the file name in the linear list. Therefore, it can greatly decrease the directory search time. Insertion and deletion are also fairly straightforward, although some provision must be made for collisions—situations in which two file names hash to the same location.

The major difficulties with a hash table are its generally fixed size and the dependence of the hash function on that size. For 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, for instance, by using the remainder of a division by 64. If we later try to create a 65th file, we must enlarge the directory hash table—say, to 128 entries. As a result, we need a new hash function that must map file names to the range 0 to 127, and we must reorganize the existing directory entries to reflect their new hash-function values. Alternatively, a chained-overflow hash table can be used. Each hash entry can be a linked list instead of an individual value, and we can resolve collisions by adding the new entry to the linked list. Lookups may be somewhat slowed, because searching for a name might require stepping through a linked list of colliding table entries. Still, this method is likely to be much faster than a linear search through the entire directory.



### 3.5 Device Driver



**Figure 3.14 : Device Drivers**

**Device driver** (commonly referred to as a *driver*) is a computer program that operates or controls a particular type of device that is attached to a computer. A driver provides a software interface to hardware devices, enabling operating systems and other computer programs to access hardware functions without needing to know precise details of the hardware being used. A driver typically communicates with the device through the computer bus or communications subsystem to which the hardware connects. When a calling program invokes a routine in the driver, the driver issues commands to the device. Once the device sends data back to the driver, the driver may invoke routines in the original calling program. Drivers are hardware-dependent and operating-system-specific. They usually provide the interrupt handling required for any necessary asynchronous time-dependent hardware interface.

Device drivers are software modules that can be plugged into an OS to handle a particular device. Operating System takes help from device drivers to handle all I/O devices. Device drivers encapsulate device-dependent code and implement a standard interface in such a way

that code contains device-specific register reads/writes. Device driver is generally written by the device's manufacturer and delivered along with the device on a CD-ROM.

### **3.5.1 Functions of Device Driver**

Device drivers simplify programming by acting as translator between a hardware device and the applications or operating systems that use it. Programmers can write the higher-level application code independently of whatever specific hardware the end-user is using.

- It accept abstract read and write requests from the device independent software above to it.
- It controls the power requirement and log event.
- It checks the status of the device. If device is in idle state then request is processed and if devices is in busy state then it queue the request which will process latter.
- Interact with the device controller to take and give I/O and perform required error handling
- Making sure that the request is executed successfully

### **3.5.2 Path Management**

Different controllers require large memory buffers and they have complicated electronics so are expensive. There is an idea for reducing the cost is to have different types of devices attached to only one controller. At a single time, a controller can control one and only one device and that's why, only one device can be active even if some amount of parallelism is possible due to overlapped seeks. If there are I/O requests from both the devices attached to the controller, one of them will have to wait. And all pending requests for such any device are queued in a device request queue by the device drivers. And the device driver creates a data structure and has the algorithm to handle the issue of queue. If the response time is very

important, these I/O waits have to be reduced, and if one is ready to spend more money, one can have separate controller for each device. The connection will exist between each controller/device pair.

In some of the mainframe computers, the functions of a controller are very complex, and they are split into two units. One is Channel and other is known as Control Unit (CU). Channel sounds like a wire or a bus, but it is actually a very small computer with the capacity of executing only some specific I/O instructions. A controller normally controls devices of the same types, but a channel can handle controllers of different types and there exists multiple paths between the memory and the devices and these paths are symmetrical or asymmetrical.

Because of multiple paths, the complexity of the device driver routine increases, but the response time increases. One controller controls two devices, the speed will be slower, if there was only one controller per device. And this is because, there is no issue of path management and waiting period until a controller gets free.

### **3.5.3 The Sub modules of DD**

As the information management divided into two parts, i.e. file system and Device Drivers. Conceptually it is divided in four sub modules:

1. **I/O Procedure:** In order to perform the path management and to create the pending requests, the I/O procedure maintains the data structures by using Channel Control Block (CCB), Control Unit Control Block (CUCB), Device Control Block (DCB), and Input/output Request Block (IORB). The data structure is maintained in the memory and it is uploaded as new request arrives and it is serviced. One can consider a certain area into the memory dedicated to contain the IORB records. This area could have number of slots

for the IORB records for all the devices, CUs and channels. The management of free slots, allocation and unlinking these IORBs to proper DCBs, CUCBs and CCBs will require complex algorithms which are main part of I/O procedure.

2. **I/O Scheduler:** Request for I/O operations pending for a device normally originates from different processes with different priorities. It seems that rational solution to the problem of scheduling is by the priorities of requesting processes. But this is generally not followed. The I/O operation is electromechanical and therefore, it is very slow. Hence. The whole emphasis of I/O scheduling is on reducing the disk arm movement, and consequently the seek time. This improves throughput and response time even if it might mean a little extra wait for a process with higher science, let us imagine that there are two processes waiting for a device to get free and there are Two IORBs for that device). Let us also assume that the process With the lower priority wants to read from the same track where the R/W arms currently are, therefore, requiring no seek time
3. **Device Handler:** The device handler basically is a piece of software which prepares an I/O program for the channel or a controller, loads it for them and instructs the hardware to execute the actual I/O. After this happens, the device handler goes to sleep. The hardware performs the actual I/O and on completion, it generates an interrupt. The ISR (Interrupt Service Routines) for this interrupt wakes up the device handler again. It checks for any errors. If there is no error, the device handler instructs the DMA transfer the data to the memory.

4. **Interrupt Service Routines (ISR):** An interrupt service routine (ISR) is a software routine that hardware invokes in response to an interrupt. ISR examines an interrupt and determines how to handle it executes the handling, and then returns a logical interrupt value. If no further handling is required the ISR notifies the kernel with a return value. An ISR must perform very quickly to avoid slowing down the operation of the device and the operation of all lower-priority ISRs.

## Unit 5

### Inter Process Communication

#### The Producer-Consumer Problems

The Producer-Consumer problem is a classic problem this is used for multi-process synchronization i.e. synchronization between more than one processes In the producer-consumer problem, there is one Producer that is producing something and there is one Consumer that is consuming the products produced by the Producer. The producers and consumers share the same memory buffer that is of fixed-size.

The job of the Producer is to generate the data, put it into the buffer, and again start generating data. While the job of the Consumer is to consume the data from the buffer.

#### *What's the problem here?*

The following are the problems that might occur in the Producer-Consumer:

- The producer should produce data only when the buffer is not full. If the buffer is full, then the producer shouldn't be allowed to put any data into the buffer.
- The consumer should consume data only when the buffer is not empty. If the buffer is empty, then the consumer shouldn't be allowed to take any data from the buffer.
- The producer and consumer should not access the buffer at the same time.

#### **What's the solution?**

The above three problems can be solved with the help of semaphores. In the producer-consumer problem, we use three semaphore variables:

1. Semaphore S: This semaphore variable is used to achieve mutual exclusion between processes. By using this variable, either Producer or Consumer will be allowed to use or access the shared buffer at a particular time. This variable is set to 1 initially.
2. Semaphore E: This semaphore variable is used to define the empty space in the buffer. Initially, it is set to the whole space of the buffer i.e. "n" because the buffer is initially empty.
3. Semaphore F: This semaphore variable is used to define the space that is filled by the producer. Initially, it is set to "0" because there is no space filled by the producer initially.

By using the above three semaphore variables and by using the *wait()* and *signal()* function, we can solve our problem(the *wait()* function decreases the semaphore variable by 1 and the *signal()* function increases the semaphore variable by 1). So. let's see how.

The following is the pseudo-code for the producer:

```
void producer() {  
  while(T) {  
    produce()  
    wait(E)  
    wait(S)  
    append()  
    signal(S)  
    signal(F)  
  }  
}
```

The above code can be summarized as:

- *while()* is used to produce data, again and again, if it wishes to produce, again and again.
- *produce()* function is called to produce data by the producer.
- *wait(E)* will reduce the value of the semaphore variable "E" by one i.e. when the producer produces something then there is a decrease in the value of the empty space in the buffer. If the buffer is full i.e. the value of the semaphore variable "E" is "0", then the program will stop its execution and no production will be done.
- *wait(S)* is used to set the semaphore variable "S" to "0" so that no other process can enter into the critical section.
- *append()* function is used to append the newly produced data in the buffer.
- *signal(s)* is used to set the semaphore variable "S" to "1" so that other processes can come into the critical section now because the production is done and the append operation is also done.
- *signal(F)* is used to increase the semaphore variable "F" by one because after adding the data into the buffer, one space is filled in the buffer and the variable "F" must be updated.

This is how we solve the produce part of the producer-consumer problem. Now, let's see the consumer solution. The following is the code for the consumer:

```
void consumer() {
```

```
while(T) {  
wait(F)  
wait(S)  
take()  
signal(S)  
signal(E)  
use()  
}  
}
```

The above code can be summarized as:

- *while()* is used to consume data, again and again, if it wishes to consume, again and again.
- *wait(F)* is used to decrease the semaphore variable "F" by one because if some data is consumed by the consumer then the variable "F" must be decreased by one.
- *wait(S)* is used to set the semaphore variable "S" to "0" so that no other process can enter into the critical section.
- *take()* function is used to take the data from the buffer by the consumer.
- *signal(S)* is used to set the semaphore variable "S" to "1" so that other processes can come into the critical section now because the consumption is done and the take operation is also done.
- *signal(E)* is used to increase the semaphore variable "E" by one because after taking the data from the buffer, one space is freed from the buffer and the variable "E" must be increased.
- *use()* is a function that is used to use the data taken from the buffer by the process to do some operation.

## Race condition

In General operating systems different processes may share a common storage area, such as memory or files on a hard disk, where they are allowed to read and write. Operating systems have some responsibilities to coordinate the desired operations in the right way between processes that use common areas. These processes need to communicate, need to communicate with each other, have a lot of business, in order to ensure that the action of one process will not affect the normal action of another process, and then lead to the process can not get the desired results after running. In the concept of operating system, the term IPC (Inter Process Communication) is usually used to represent the communication between multiple processes.



To explain what race condition is, we introduce a simple example to illustrate: A file contains a number  $n$ , process A and process B want to read the number of this file, and add this number to 1, save back to the file. Assuming that the initial value of  $n$  is 0, in our ideal case, after process A and process B run, the value of  $N$  in the file should be 2, but in fact, it may occur that the value of  $n$  is 1. We can consider what steps each process needs to take to do this:

- Read the value of  $N$  in the file
- Let  $n = n + 1$
- Save the new  $n$  value back to the file.

Before further explaining race conditions, it is necessary to review several knowledge points in the concept of operating system:

- Processes can run concurrently, even when there is only one CPU)
- Clock interruption in the operating system can cause a rescheduling of the process.
- In addition to clock interrupts, interrupts from other devices can also cause a rescheduling of process runs.

Assuming that a clock interrupt occurs when process A runs steps 1 and 2, but it has not started to run step 3, the operating system makes process B start to run by scheduling. When process B runs step 1, it finds that the value of  $n$  is 0, so it runs steps 2 and 3, and finally saves  $n = 1$  to a file. After that, process A continues to run, because it does not know that process B has changed the value in the file before it runs step 3, so process A also writes  $n = 1$  back to the file. That's the problem. In the process of running process A, there are other processes to manipulate the data it operates on.

The only way to make  $n = 2$  is to expect process A and process B In order Complete all steps separately.

We can give a definition of race conditions.

Two or more processes read and write some shared data, and the final result depends on the exact timing of the process, called race conditions.

In the above text, we use processes as objects to discuss race conditions, which is also applicable to threads, where threads include but are not limited to kernel threads and user threads. Because in an operating system, processes actually rely on threads to run programs. Even more, the concept of race conditions applies to thread security in the Java language. (Refer to Chapter 2 of Java Concurrent Programming Practice)

## **Mutual Exclusion and Critical Zone**

To avoid race condition, we need to use some means to ensure that when a process uses a shared variable or file, other processes cannot do the same thing. In other words, we need to “mutually exclusive”.

As an example, we can define the program fragment in step 1-3 as a critical area, which means that the area is sensitive, because once the process runs to this area, it means that the common data area or file will be operated on, and it means that other processes may also be running to the critical area. If the two processes can not be in the critical zone at the same time in an appropriate way, then the race conditions can be avoided.

That is to say, we need to think about how to “mutually exclusive”.

### **Mutual Exclusive Scheme**

The essence of mutual exclusion is to prevent multiple processes from entering the critical region at the same time.

### **Shielding interruption**

In the previous example, process B was able to enter the critical zone because process A was interrupted in the critical zone. If we allow process A to shield all interrupts immediately after entering the critical area and respond to interrupts only after leaving the critical area, then even if interrupts occur, the CPU will not switch to other processes, so process A can safely modify the file content at this time, without worrying that other processes will interfere with its work.

However, the idea is good, but in fact it is not feasible. First, if there are more than one CPU, then process A can not shield interruption to other CPUs, it can only shield the CPU that is dispatching it, so the process dispatched by other CPUs can still enter the critical area; second, on the issue of power, can the power of shielding interruption be given to user processes? If the process masks the interrupt and no longer responds to the interrupt, then a process may hang up the entire operating system.

### **Lock variables**

Perhaps by setting a lock flag bit and setting its initial value to 0, when a process wants to enter the critical zone, it first checks whether the lock value is 0, if 0, then it is set to 1, then it enters the critical zone, and after exiting the critical zone, it changes the lock value to 0; if the lock value is already 1 when checking, it means that other processes are already in the critical zone, so the process follows. The loop waits and constantly detects the value of the lock until it becomes zero.

However, there are race conditions in this way, because when a process reads out a lock value of 0, before it sets its value to 1, another process is scheduled, and it reads the lock value of 0, so that both processes are in the critical zone at the same time.

### **Strict Alternation**

The problem with lock variables is that the action of changing the lock variable from 0 to 1 is performed by the process that wants to acquire the lock. If we change this action to a process that has acquired a lock to execute, then there is no race condition.

First, set a variable turn to represent who is currently allowed to get the lock. Assuming there are two processes, the logic of process A is as follows:

```
While (turn!= 0) { // If it's not your turn to acquire the lock, enter the loop and wait
}
Do_critical_region();//Procedures for executing critical zones
Turn = 1; // The lock variable is modified to other values by the party that acquires the lock,
allowing other processes to acquire the lock
Do_non_critical_region();//Procedures to execute non-critical zones
```

The logic of process B is as follows:

```
When (turn!= 1) { // If it's not your turn to acquire the lock, enter the loop and wait
}
Do_critical_region();//Procedures for executing critical zones
Turn = 0; // / The lock variable is modified to other values by the party that acquires the lock,
allowing other processes to acquire the lock
Do_non_critical_region();//Procedures to execute non-critical zones
```

But here we need to consider one thing. Suppose that the do\_non\_critical\_region() of process A needs to be executed for a long time, that is, the logic of the non-critical zone of process A needs to be executed for a long time, and the logic of the non-critical zone of process B is executed quickly. Obviously, the frequency of process A entering the critical zone will be a little less than that of process B. Ideally, process B should be more approaching. Boundary area several times. But because process B sets turn to 0 before executing the logic of non-critical region, when it quickly finishes executing the logic of non-critical region and checks the value of turn, it is found that the value of turn is not always 1. The value of turn needs process A to set it to 1, while process A is carrying on a long logic code of non-critical region at this time, so process B can not enter. Enter the critical zone.

This shows that strict rotation is not a good method when one process is much slower than another.

### **Peterson algorithm**

The problem of strict rotation method lies in two words, that is, multiple processes enter the critical region in turn. The fundamental reason is that the process of obtaining locks depends on the modification of lock variables by other processes, while other processes need to be executed by non-critical region logic before they can modify lock variables.

The turn variable in the strict rotation method is not only used to indicate who is currently in the turn to acquire the lock, but also means that it still prevents other processes from entering the critical zone before its value is changed. It is precisely that a process always changes the turn value after passing through the logic of the non-critical zone.

Therefore, we can use two variables to express, one to indicate who should get the lock at present, and the other to indicate that the current process has left the critical zone. This method is actually called Peterson's algorithm, which was proposed by T. Dekker, a Dutch mathematician.

```
static final int N = 2;
int turn = 0;
boolean[] interested = new boolean[N];

void enter_region(int process) {
    int other = 1 - process;
    interested[process] = true;
    turn = process;
    while(turn == process && !interested[other]) {
    }
}

void exit_region(int process) {
    interested[process] = false;
}
```

When a process needs to enter the critical region, it calls enter\_region first, and after leaving the critical region, it calls exit\_region. Peterson algorithm makes the process eliminate its “interest” in the critical area immediately after leaving the critical area, so other processes can judge whether they can legally enter the critical area according to the value of turn.

### **TSL Directive**

Looking back at the “lock variable” method we mentioned earlier, one of its fatal drawbacks is that when state variables are changed, such as from 0 to 1 or from 1 to 0, they can be interrupted, so there are race conditions.

Then, on the basis of locking variables, we propose that if the modification of locking variables is not made by the process that wants to get into the critical region, but by the process that wants to leave the critical region after entering the critical region, the race condition can be avoided, and then the strict rotation method and the improved Peterson algorithm based on the strict rotation method are introduced. These methods are all considered in the way of software. In fact, with the support of hardware CPU, the change of lock variables can be guaranteed without interruption, which makes lock variables a good solution to process mutual exclusion.

At present, most computer CPUs support TSL instructions, which are called Test and Set Lock. It reads a variable of memory (word) into the register RX, and then stores a non-zero value on the memory address. Reading and writing operations are guaranteed to be non-interruptible from the hardware level, that is to say, atomic. It uses the method of locking the memory bus when executing TSL instructions and prohibiting other CPUs from accessing memory before the end of TSL instructions. This is what we often call CAS (Compare and Set).

When it is necessary to change the lock variable from 0 to 1, first copy the value of memory to the register, and set the value of memory to 1, then check whether the value of the register is 0, if it is 0, then the operation is successful, if not 0, then repeat detection, knowing that the value of the register has changed to 0, if the value of the register has changed to 0, it means that another process has left the critical area. When a process leaves the critical region, it is necessary to set the value of the variable in memory to 0.

### **Be busy waiting**

In fact, the Peterson algorithm and TSL method mentioned above all have one characteristic: they are busy waiting when they are waiting to enter the critical region, which we often call spin. Its disadvantage is that it wastes CPU time slices and can lead to priority inversion Questions.

Consider that a computer has two processes, H priority is higher and L priority is lower. Scheduling rules stipulate that H can run as long as H is in a ready state. At some point, L is in the critical region, and H is in the ready state, ready to run. But H needs to be busy waiting, while L can not be scheduled because of its low priority, so it can not leave the critical area, so H will always be busy waiting, while L can not leave the critical area. This situation is called priority inversion problem( priority inversion problem)

### **Blocking and awakening of process**

The operating system provides some primitives, sleep and wakeup.

The process or function provided by the kernel to an out-of-core call becomes a primitive, and the primitive is not allowed to interrupt during execution.

Sleep is a system call that calls a blocked process until another process calls the wakeup method, which takes the blocked process as a parameter and wakes it up. The biggest difference between blocking and busy waiting is that when a process is blocked, the CPU will not allocate time slices to it, while busy waiting is always idle, consuming time slices of the CPU.

### **Semaphores**

First of all, we need to understand what the semaphore is to solve, because the blockage and wake-up of a process are caused by different processes, such as process A calling sleep () will enter the blockage, and process B calling wakeup (A) will wake up process A. Because it is carried out in different processes, there is also the problem of interruption. Procedure A needs to call sleep () into blocking state according to logic. However, just before it calls sleep method, process B starts running because of clock interruption. According to logic, it calls wakeup () method to wake up process A, but because process A has not entered blocking state, the wakeup signal is lost, waiting for process A to interrupt before. When the location starts to continue running and gets blocked, there may be no process to wake it up.

Therefore, the blocking and waking of a process should be recorded by introducing an additional variable, which records the number of wakes-up times, and the value of the variable is added to 1 for each wake-up. With this variable, even if the wakeup operation precedes the sleep operation, the wakeup operation is recorded in the variable. When the process sleeps, because other processes have been awakened, it is considered that the process does not need to be blocked.

This variable, in the operating system concept, is called semaphore, a method proposed by Dijkstra in 1965. There are two operations for semaphores, down and up.

Down operation corresponds to sleep, which first detects whether the semaphore value is greater than 0, and if it is greater than 0, it decreases by 1. The process does not need to block at this time, which is equivalent to consuming a wakeup; if the semaphore is 0, the process will enter a blocking state.

The up operation corresponds to wakeup. If a process is blocked on the semaphore, the system will choose one of the processes to wake it up. At this time, the value of the semaphore does not need to change, but the blocked process is one less. If no process is blocked on the semaphore during up operation, it will add the value of the semaphore to 1.

In some places, down and up operations are called PV operations because in Dijkstra's paper, P and V are used to represent down and up, respectively.

The down and up operations of semaphores are primitives supported by the operating system. They are atomic operations without interruption.

Mutex is actually a special case of semaphores. Its value is only 0 and 1. When we do not need the counting ability of semaphores, we can use mutex. In fact, it means that the critical value allows only one process to enter at the same time, while semaphores allow multiple processes to enter the critical area at the same time.

A semaphore is a variable that indicates the number of resources that are available in a system at a particular time and this semaphore variable is generally used to achieve the process synchronization. It is generally denoted by "S". You can use any other variable name of your choice.

A semaphore uses two functions i.e. *wait()* and *signal()*. Both these functions are used to change the value of the semaphore but the value can be changed by only one process at a particular time and no other process should change the value simultaneously.

The *wait()* function is used to decrement the value of the semaphore variable "S" by one if the value of the semaphore variable is positive. If the value of the semaphore variable is 0, then no operation will be performed.

```
wait(S) {  
while (S == 0); //there is ";" sign here  
S--;  
}
```

The *signal()* function is used to increment the value of the semaphore variable by one.

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

There are two types of semaphores:

- **Binary Semaphores:** In Binary semaphores, the value of the semaphore variable will be 0 or 1. Initially, the value of semaphore variable is set to 1 and if some process wants to use some resource then the *wait()* function is called and the value of the semaphore is changed to 0 from 1. The process then uses the resource and when it releases the resource then the *signal()* function is called and the value of the semaphore variable is increased to 1. If at a particular instant of time, the value of the semaphore variable is 0 and some other process wants to use the same resource then it has to wait for the release of the resource by the previous process. In this way, process synchronization can be achieved.
- **Counting Semaphores:** In Counting semaphores, firstly, the semaphore variable is initialized with the number of resources available. After that, whenever a process needs some resource, then the *wait()* function is called and the value of the semaphore variable is decreased by one. The process then uses the resource and after using the resource, the *signal()* function is called and the value of the semaphore variable is increased by one. So, when the value of the semaphore variable goes to 0 i.e all the resources are taken by the process and there is no resource left to be used, then if some other process wants to use resources then that process has to wait for its turn. In this way, we achieve the process synchronization.

### ***Advantages of semaphore***

- The mutual exclusion principle is followed when you use semaphores because semaphores allow only one process to enter into the critical section.

- Here, you need not verify that a process should be allowed to enter into the critical section or not. So, processor time is not wasted here.

#### ***Disadvantages of semaphore***

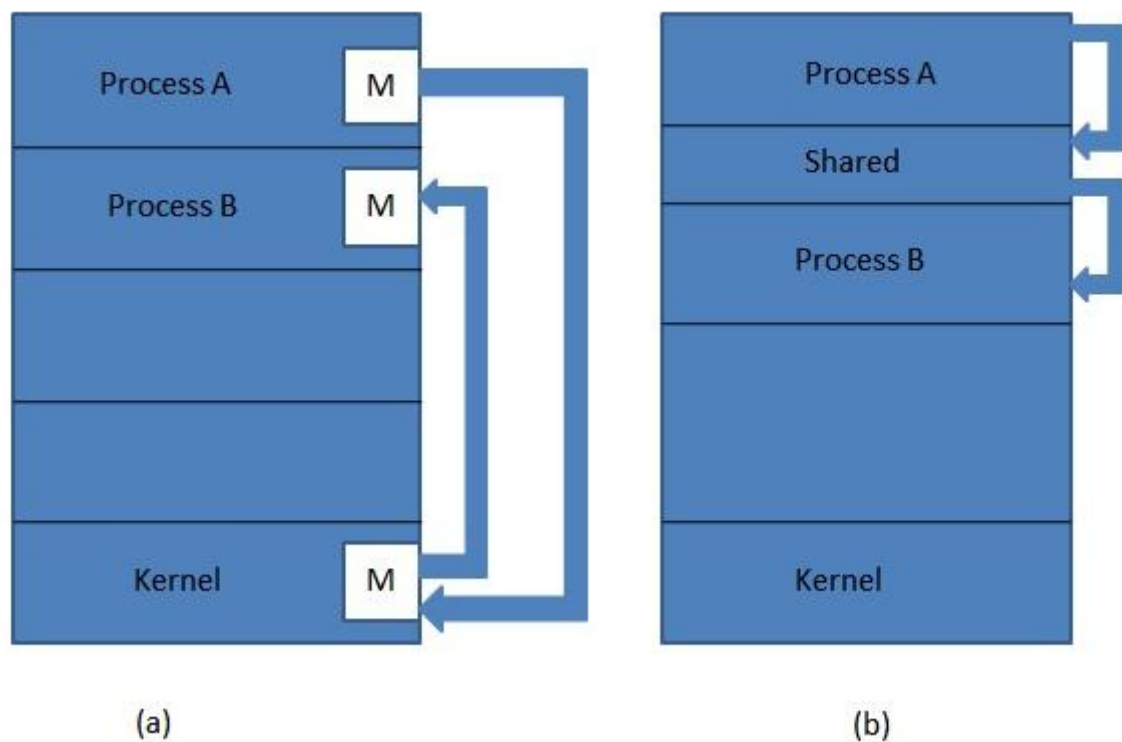
- While using semaphore, if a low priority process is in the critical section, then no other higher priority process can get into the critical section. So, the higher priority process has to wait for the complete execution of the lower priority process.
- The *wait()* and *signal()* functions need to be implemented in the correct order. So, the implementation of a semaphore is quite difficult.

#### **Interprocess Communication**

- Processes within a system may be independent or cooperating
- Cooperating process can affect or be affected by other processes, including sharing data
- Reasons for cooperating processes:
  - Information sharing
  - Computation speedup
  - Modularity
  - Convenience
- Cooperating processes need interprocess communication (IPC)
- Two models of IPC
  - Shared memory
  - Message passing



## Communications Models



## Cooperating Processes

- Independent process cannot affect or be affected by the execution of another process
- Cooperating process can affect or be affected by the execution of another process
- Advantages of process cooperation
  - Advantages of process cooperation
  - Computation speed-up
  - Modularity
  - Convenience

The operating systems literature is full of interesting problems that have been widely discussed and

analyzed using a variety of synchronization methods. In the following sections we will examine four of the better-known problems.

### i) Shared Memory Method

Ex: Producer-Consumer problem

There are two processes: Producer and Consumer. Producer produces some item and Consumer consumes that item. The two processes share a common space or memory location known as a buffer where the item produced by Producer is stored and from which the Consumer consumes the item, if needed. There are two versions of this problem: the first one is known as unbounded buffer problem in which Producer can keep on producing items and there is no limit on the size of the buffer, the second one is known as the bounded buffer problem in which Producer can produce up to a certain number of items before it starts waiting for Consumer to consume it. We will discuss the bounded buffer problem. First, the Producer and the Consumer will share some common memory, then producer will start producing items. If the total produced item is equal to the size of buffer, producer will wait to get it consumed by the Consumer. Similarly, the consumer will first check for the availability of the item. If no item is available, Consumer will wait for Producer to produce it. If there are items available, Consumer will consume it. The pseudo code to demonstrate is provided below:

#### Shared Data between the two Processes

```
#define buff_max 25
#define mod %

    struct item{

        // different member of the produced data
        // or consumed data
        -----
    }

    // An array is needed for holding the items.
    // This is the shared place which will be
    // access by both process
    // item shared_buff [ buff_max ];

    // Two variables which will keep track of
    // the indexes of the items produced by producer
    // and consumer The free index points to
    // the next free index. The full index points to
    // the first full index.
    int free_index = 0;
    int full_index = 0;
```

### Producer Process Code

```
item nextProduced;

while(1){

    // check if there is no space
    // for production.
    // if so keep waiting.
    while((free_index+1) mod buff_max == full_index);

    shared_buff[free_index] = nextProduced;
    free_index = (free_index + 1) mod buff_max;
}
```

### Consumer Process Code

```
item nextConsumed;

while(1){

    // check if there is an available
    // item for consumption.
    // if not keep on waiting for
    // get them produced.
    while((free_index == full_index);

    nextConsumed = shared_buff[full_index];
    full_index = (full_index + 1) mod buff_max;
}
```

In the above code, the Producer will start producing again when the  $(\text{free\_index}+1) \bmod \text{buff\_max}$  will be free because if it is not free, this implies that there are still items that can be consumed by the Consumer so there is no need to produce more. Similarly, if free index and full index point to the same index, this implies that there are no items to consume.

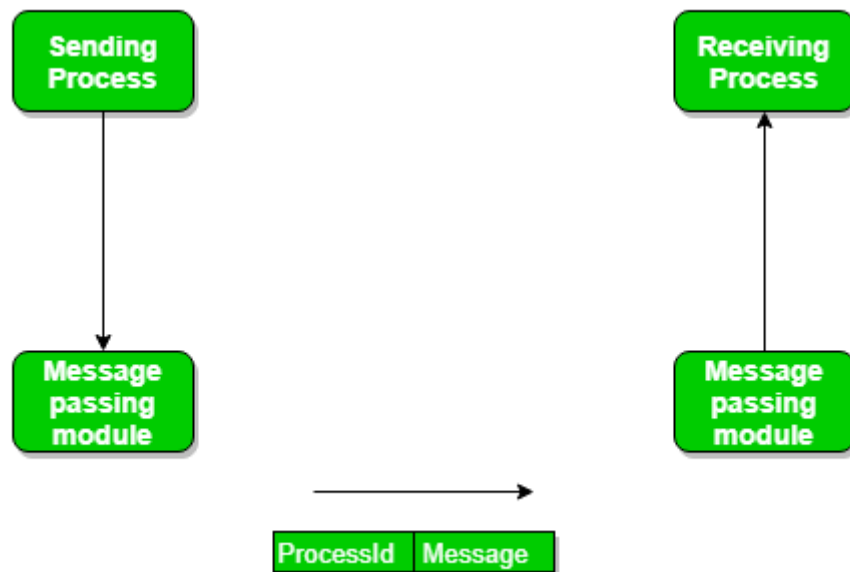
### ii) Messaging Passing Method

Now, We will start our discussion of the communication between processes via message passing. In this method, processes communicate with each other without using any kind of shared memory. If two processes p1 and p2 want to communicate with each other, they proceed as follows:

- Establish a communication link (if a link already exists, no need to establish it again.)
- Start exchanging messages using basic primitives.

We need at least two primitives:

- send(message, destination) or send(message)
- receive(message, host) or receive(message)



The message size can be of fixed size or of variable size. If it is of fixed size, it is easy for an OS designer but complicated for a programmer and if it is of variable size then it is easy for a programmer but complicated for the OS designer. A standard message can have two parts: header and body.

The header part is used for storing message type, destination id, source id, message length, and control information. The control information contains information like what to do if runs out of buffer space, sequence number, priority. Generally, message is sent using FIFO style.

### **Message Passing through Communication Link.**

Direct and Indirect Communication link

Now, We will start our discussion about the methods of implementing communication link. While implementing the link, there are some questions which need to be kept in mind like :

1. How are links established?
2. Can a link be associated with more than two processes?
3. How many links can there be between every pair of communicating processes?
4. What is the capacity of a link? Is the size of a message that the link can accommodate fixed or variable?
5. Is a link unidirectional or bi-directional?

A link has some capacity that determines the number of messages that can reside in it temporarily for which every link has a queue associated with it which can be of zero capacity, bounded capacity, or unbounded capacity. In zero capacity, the sender waits until the receiver

informs the sender that it has received the message. In non-zero capacity cases, a process does not know whether a message has been received or not after the send operation. For this, the sender must communicate with the receiver explicitly. Implementation of the link depends on the situation, it can be either a direct communication link or an in-directed communication link.

Direct Communication links are implemented when the processes use a specific process identifier for the communication, but it is hard to identify the sender ahead of time.

For example: the print server.

In-direct Communication is done via a shared mailbox (port), which consists of a queue of messages. The sender keeps the message in mailbox and the receiver picks them up.

### **Message Passing through Exchanging the Messages.**

Synchronous and Asynchronous Message Passing:

A process that is blocked is one that is waiting for some event, such as a resource becoming available or the completion of an I/O operation. IPC is possible between the processes on same computer as well as on the processes running on different computer i.e. in networked/distributed system. In both cases, the process may or may not be blocked while sending a message or attempting to receive a message so message passing may be blocking or non-blocking. Blocking is considered synchronous and blocking send means the sender will be blocked until the message is received by receiver. Similarly, blocking receive has the receiver block until a message is available. Non-blocking is considered asynchronous and Non-blocking send has the sender send the message and continue. Similarly, Non-blocking receive has the receiver receive a valid message or null. After a careful analysis, we can come to a conclusion that for a sender it is more natural to be non-blocking after message passing as there may be a need to send the message to different processes. However, the sender expects acknowledgement from the receiver in case the send fails. Similarly, it is more natural for a receiver to be blocking after issuing the receive as the information from the received message may be used for further execution. At the same time, if the message send keep on failing, the receiver will have to wait indefinitely. That is why we also consider the other possibility of message passing. There are basically three preferred combinations:

- Blocking send and blocking receive
- Non-blocking send and Non-blocking receive
- Non-blocking send and Blocking receive (Mostly used)

In Direct message passing, The process which want to communicate must explicitly name the recipient or sender of communication.

e.g. `send(p1, message)` means send the message to p1.

similarly, `receive(p2, message)` means receive the message from p2.

In this method of communication, the communication link gets established automatically, which can be either unidirectional or bidirectional, but one link can be used between one pair of the sender and receiver and one pair of sender and receiver should not possess more than one pair of links. Symmetry and asymmetry between sending and receiving can also be implemented i.e. either both process will name each other for sending and receiving the messages or only the sender will name receiver for sending the message and there is no need for receiver for naming the sender for receiving the message. The problem with this method of communication is that if the name of one process changes, this method will not work.

In Indirect message passing, processes use mailboxes (also referred to as ports) for sending and receiving messages. Each mailbox has a unique id and processes can communicate only if they share a mailbox. Link established only if processes share a common mailbox and a single link can be associated with many processes. Each pair of processes can share several communication links and these links may be unidirectional or bi-directional. Suppose two process want to communicate though Indirect message passing, the required operations are: create a mail box, use this mail box for sending and receiving messages, then destroy the mail box. The standard primitives used are: send(A, message) which means send the message to mailbox A. The primitive for the receiving the message also works in the same way e.g. received (A, message). There is a problem in this mailbox implementation. Suppose there are more than two processes sharing the same mailbox and suppose the process p1 sends a message to the mailbox, which process will be the receiver? This can be solved by either enforcing that only two processes can share a single mailbox or enforcing that only one process is allowed to execute the receive at a given time or select any process randomly and notify the sender about the receiver. A mailbox can be made private to a single sender/receiver pair and can also be shared between multiple sender/receiver pairs. Port is an implementation of such mailbox which can have multiple sender and single receiver. It is used in client/server applications (in this case the server is the receiver). The port is owned by the receiving process and created by OS on the request of the receiver process and can be destroyed either on request of the same receiver process or when the receiver terminates itself. Enforcing that only one process is allowed to execute the receive can be done using the concept of mutual exclusion. Mutex mailbox is create which is shared by n process. Sender is non-blocking and sends the message. The first process which executes the receive will enter in the critical section and all other processes will be blocking and will wait.

Now, lets discuss the Producer-Consumer problem using message passing concept. The producer places items (inside messages) in the mailbox and the consumer can consume an item when at least one message present in the mailbox. The code is given below:

Producer Code

```
void Producer(void){  
  
    int item;  
    Message m;  
  
    while(1){
```

```

        receive(Consumer, &m);
        item = produce();
        build_message(&m , item ) ;
        send(Consumer, &m);
    }
}

```

## Consumer Code

```

void Consumer(void){
    int item;
    Message m;

    while(1){

        receive(Producer, &m);
        item = extracted_item();
        send(Producer, &m);
        consume_item(item);
    }
}

```

## Examples of IPC systems

1. Posix : uses shared memory method.
2. Mach : uses message passing
3. Windows XP : uses message passing using local procedural calls

## Communication in client/server Architecture:

There are various mechanism:

- Pipe
- Socket
- Remote Procedural calls (RPCs)

Below are some of the classical problem depicting flaws of process synchronaization in systems where cooperating processes are present.

We will discuss the following three problems:

1. Bounded Buffer (Producer-Consumer) Problem
2. Dining Philosophers Problem.
3. The Readers Writers Problem

## Bounded Buffer Problem

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

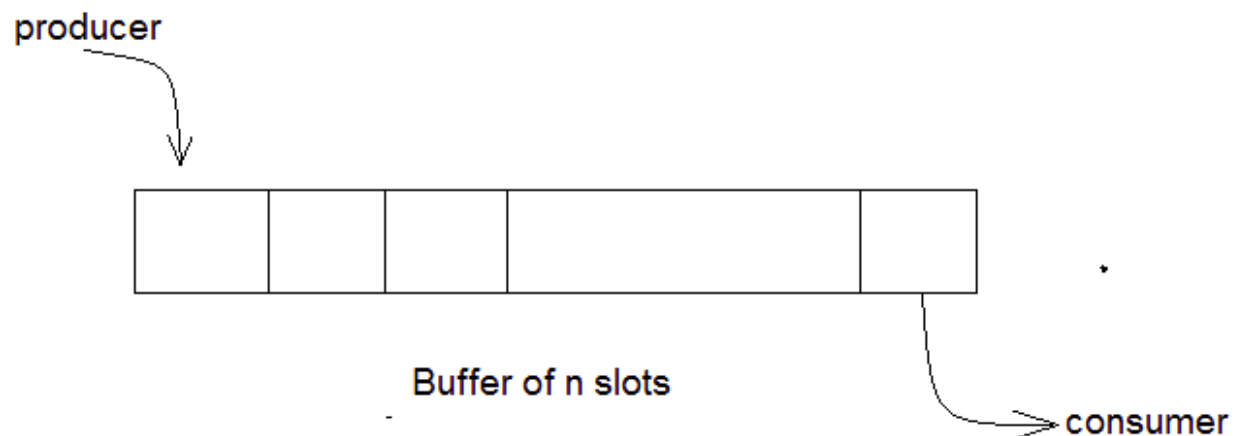
Because the buffer pool has a maximum size, this problem is often called the Bounded buffer problem.

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

Bounded buffer problem, which is also called producer consumer problem, is one of the classic problems of synchronization. Let's start by understanding the problem here, before moving on to the solution and program code.

### What is the Problem Statement?

There is a buffer of  $n$  slots and each slot is capable of storing one unit of data. There are two processes running, namely, producer and consumer, which are operating on the buffer.



### Bounded Buffer Problem

A producer tries to insert data into an empty slot of the buffer. A consumer tries to remove data from a filled slot in the buffer. As you might have guessed by now, those two processes won't produce the expected output if they are being executed concurrently.

There needs to be a way to make the producer and consumer work in an independent manner.

Here's a Solution

One solution of this problem is to use semaphores. The semaphores which will be used here are:



- m, a binary semaphore which is used to acquire and release the lock.
- **empty**, a counting semaphore whose initial value is the number of slots in the buffer, since, initially all slots are empty.
- **full**, a counting semaphore whose initial value is **0**.

At any instant, the current value of empty represents the number of empty slots in the buffer and full represents the number of occupied slots in the buffer.

### The Producer Operation

The pseudocode of the producer function looks like this:

```
do
{
// wait until empty > 0 and then decrement 'empty'
wait(empty);
// acquire lock
wait(mutex);

/* perform the insert operation in a slot */

// release lock
signal(mutex);
// increment 'full'
signal(full);
}
while(TRUE)
```

- Looking at the above code for a producer, we can see that a producer first waits until there is atleast one empty slot.
- Then it decrements the empty semaphore because, there will now be one less empty slot, since the producer is going to insert data in one of those slots.
- Then, it acquires lock on the buffer, so that the consumer cannot access the buffer until producer completes its operation.
- After performing the insert operation, the lock is released and the value of full is incremented because the producer has just filled a slot in the buffer.

### The Consumer Operation

The pseudocode for the consumer function looks like this:

```
do
{
// wait until full > 0 and then decrement 'full'
wait(full);
```

```
// acquire the lock
wait(mutex);

/* perform the remove operation in a slot */

// release the lock
signal(mutex);
// increment 'empty'
signal(empty);
}
while(TRUE);
```

- The consumer waits until there is atleast one full slot in the buffer.
- Then it decrements the full semaphore because the number of occupied slots will be decreased by one, after the consumer completes its operation.
- After that, the consumer acquires lock on the buffer.
- Following that, the consumer completes the removal operation so that the data from one of the full slots is removed.
- Then, the consumer releases the lock.
- Finally, the empty semaphore is incremented by 1, because the consumer has just removed data from an occupied slot, thus making it empty.

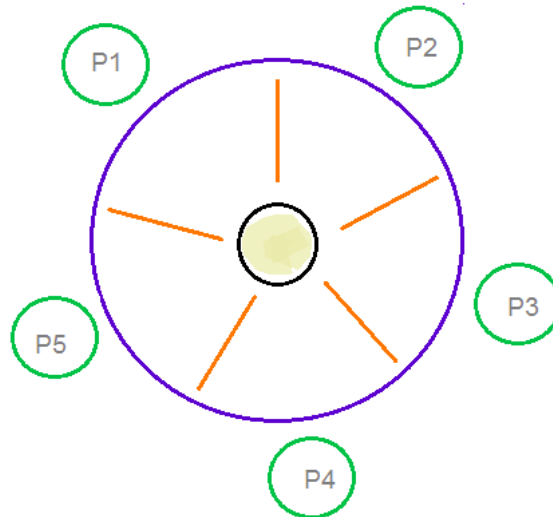
## Dining Philosophers Problem

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

The dining philosophers problem is another classic synchronization problem which is used to evaluate situations where there is a need of allocating multiple resources to multiple processes.

What is the Problem Statement?

Consider there are five philosophers sitting around a circular dining table. The dining table has five chopsticks and a bowl of rice in the middle as shown in the below figure.



At any instant, a philosopher is either eating or thinking. When a philosopher wants to eat, he uses two chopsticks - one from their left and one from their right. When a philosopher wants to think, he keeps down both chopsticks at their original place.

Here's the Solution

From the problem statement, it is clear that a philosopher can think for an indefinite amount of time. But when a philosopher starts eating, he has to stop at some point of time. The philosopher is in an endless cycle of thinking and eating.

An array of five semaphores, `stick[5]`, for each of the five chopsticks.

The code for each philosopher looks like:

```
while(TRUE)
{
wait(stick[i]);
/*
mod is used because if i=5, next
chopstick is 1 (dining table is circular)
*/
wait(stick[(i+1) % 5]);

/* eat */
signal(stick[i]);

signal(stick[(i+1) % 5]);
/* think */
}
```

When a philosopher wants to eat the rice, he will wait for the chopstick at his left and picks

up that chopstick. Then he waits for the right chopstick to be available, and then picks it too. After eating, he puts both the chopsticks down.

But if all five philosophers are hungry simultaneously, and each of them picks up one chopstick, then a deadlock situation occurs because they will be waiting for another chopstick forever. The possible solutions for this are:

- A philosopher must be allowed to pick up the chopsticks only if both the left and right chopsticks are available.
- Allow only four philosophers to sit at the table. That way, if all the four philosophers pick up four chopsticks, there will be one chopstick left on the table. So, one philosopher can start eating and eventually, two chopsticks will be available. In this way, deadlocks can be avoided.

## The Readers Writers Problem

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

Readers writer problem is another example of a classic synchronization problem. There are many variants of this problem, one of which is examined below.

### The Problem Statement

There is a shared resource which should be accessed by multiple processes. There are two types of processes in this context. They are reader and writer. Any number of readers can read from the shared resource simultaneously, but only one writer can write to the shared resource. When a writer is writing data to the resource, no other process can access the resource. A writer cannot write to the resource if there are non zero number of readers accessing the resource at that time.

### The Solution

From the above problem statement, it is evident that readers have higher priority than writer. If a writer wants to write to the resource, it must wait until there are no readers currently accessing that resource. Here, we use one mutex **m** and a semaphore **w**. An integer variable **read\_count** is used to maintain the number of readers currently accessing the resource. The variable **read\_count** is initialized to **0**. A value of **1** is given initially to **m** and **w**. Instead of

having the process to acquire lock on the shared resource, we use the mutex **m** to make the process to acquire and release lock whenever it is updating the **read\_count** variable.

The code for the writer process looks like this:

```
while(TRUE)
{
wait(w);

/* perform the write operation */

signal(w);
}
```

And, the code for the reader process looks like this:

```
while(TRUE)
{
//acquire lock
wait(m);
read_count++;
if(read_count == 1)
wait(w);

//release lock
signal(m);

/* perform the reading operation */

// acquire lock
wait(m);
read_count--;
if(read_count == 0)
signal(w);

// release lock
signal(m);
}
```

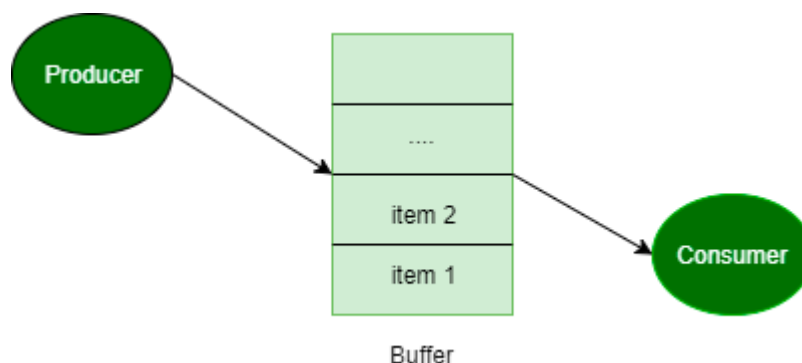
Here is the Code uncoded:

- As seen above in the code for the writer, the writer just waits on the w semaphore until it gets a chance to write to the resource.

- After performing the write operation, it increments  $w$  so that the next writer can access the resource.
- On the other hand, in the code for the reader, the lock is acquired whenever the `read_count` is updated by a process.
- When a reader wants to access the resource, first it increments the `read_count` value, then accesses the resource and then decrements the `read_count` value.
- The semaphore  $w$  is used by the first reader which enters the critical section and the last reader which exits the critical section.
- The reason for this is, when the first reader enters the critical section, the writer is blocked from the resource. Only new readers can access the resource now.
- Similarly, when the last reader exits the critical section, it signals the writer using the  $w$  semaphore because there are zero readers now and a writer can have the chance to access the resource.

### Peterson's Algorithm in Process Synchronization

Problem: The producer consumer problem (or bounded buffer problem) describes two processes, the producer and the consumer, which share a common, fixed-size buffer used as a queue. Producer produces an item and put it into buffer. If buffer is already full then producer will have to wait for an empty block in buffer. Consumer consumes an item from buffer. If buffer is already empty then consumer will have to wait for an item in buffer. Implement Peterson's Algorithm for the two processes using shared memory such that there is mutual exclusion between them. The solution should have free from synchronization problems.



### Peterson's algorithm –

```
// code for producer (j)

// producer j is ready
// to produce an item
flag[j] = true;

// but consumer (i) can consume an item
turn = i;

// if consumer is ready to consume an item
// and if its consumer's turn
while (flag[i] == true && turn == i)

    { // then producer will wait }

    // otherwise producer will produce
    // an item and put it into buffer (critical Section)

    // Now, producer is out of critical section
    flag[j] = false;
    // end of code for producer

//-----
// code for consumer i

// consumer i is ready
// to consume an item
flag[i] = true;

// but producer (j) can produce an item
turn = j;

// if producer is ready to produce an item
// and if its producer's turn
while (flag[j] == true && turn == j)

    { // then consumer will wait }

    // otherwise consumer will consume
    // an item from buffer (critical Section)

    // Now, consumer is out of critical section
    flag[i] = false;
// end of code for consumer
```

## Explanation of Peterson's algorithm

Peterson's Algorithm is used to synchronize two processes. It uses two variables, a bool array flag of size 2 and an int variable turn to accomplish it.

In the solution i represents the Consumer and j represents the Producer. Initially the flags are false. When a process wants to execute its critical section, it sets its flag to true and turn as the index of the other process. This means that the process wants to execute but it will allow the other process to run first. The process performs busy waiting until the other process has finished its own critical section. After this the current process enters its critical section and adds or removes a random number from the shared buffer. After completing the critical section, it sets its own flag to false, indicating it does not wish to execute anymore.

The program runs for a fixed amount of time before exiting. This time can be changed by changing value of the macro RT.

```
// C program to implement Peterson's Algorithm
// for producer-consumer problem.
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <time.h>
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/shm.h>
#include <stdbool.h>
#define _BSD_SOURCE
#include <sys/time.h>
#include <stdio.h>

#define BSIZE 8 // Buffer size
#define PWT 2 // Producer wait time limit
#define CWT 10 // Consumer wait time limit
#define RT 10 // Program run-time in seconds

int shmid1, shmid2, shmid3, shmid4;
key_t k1 = 5491, k2 = 5812, k3 = 4327, k4 = 3213;
bool* SHM1;
int* SHM2;
int* SHM3;

int myrand(int n) // Returns a random number between 1 and n
{
    time_t t;
    srand((unsigned)time(&t));
    return (rand() % n + 1);
}
```



```

}

int main()
{
    shmId1 = shmget(k1, sizeof(bool) * 2, IPC_CREAT | 0660); // flag
    shmId2 = shmget(k2, sizeof(int) * 1, IPC_CREAT | 0660); // turn
    shmId3 = shmget(k3, sizeof(int) * BSIZE, IPC_CREAT | 0660); // buffer
    shmId4 = shmget(k4, sizeof(int) * 1, IPC_CREAT | 0660); // time stamp

    if (shmId1 < 0 || shmId2 < 0 || shmId3 < 0 || shmId4 < 0) {
        perror("Main shmget error: ");
        exit(1);
    }
    SHM3 = (int*)shmat(shmId3, NULL, 0);
    int ix = 0;
    while (ix < BSIZE) // Initializing buffer
        SHM3[ix++] = 0;

    struct timeval t;
    time_t t1, t2;
    gettimeofday(&t, NULL);
    t1 = t.tv_sec;

    int* state = (int*)shmat(shmId4, NULL, 0);
    *state = 1;
    int wait_time;

    int i = 0; // Consumer
    int j = 1; // Producer

    if (fork() == 0) // Producer code
    {
        SHM1 = (bool*)shmat(shmId1, NULL, 0);
        SHM2 = (int*)shmat(shmId2, NULL, 0);
        SHM3 = (int*)shmat(shmId3, NULL, 0);
        if (SHM1 == (bool*)-1 || SHM2 == (int*)-1 || SHM3 == (int*)-1) {
            perror("Producer shmat error: ");
            exit(1);
        }

        bool* flag = SHM1;
        int* turn = SHM2;
        int* buf = SHM3;
        int index = 0;

        while (*state == 1) {
            flag[j] = true;
            printf("Producer is ready now.\n\n");
            *turn = i;
            while (flag[i] == true && *turn == i)

```

```

        ;

// Critical Section Begin
index = 0;
while (index < BSIZE) {
    if (buf[index] == 0) {
        int tempo = myrand(BSIZE * 3);
        printf("Job %d has been produced\n", tempo);
        buf[index] = tempo;
        break;
    }
    index++;
}
if (index == BSIZE)
    printf("Buffer is full, nothing can be produced!!!\n");
printf("Buffer: ");
index = 0;
while (index < BSIZE)
    printf("%d ", buf[index++]);
printf("\n");
// Critical Section End

flag[j] = false;
if (*state == 0)
    break;
wait_time = myrand(PWT);
printf("Producer will wait for %d seconds\n\n", wait_time);
sleep(wait_time);
}
exit(0);
}

if (fork() == 0) // Consumer code
{
    SHM1 = (bool*)shmat(shmid1, NULL, 0);
    SHM2 = (int*)shmat(shmid2, NULL, 0);
    SHM3 = (int*)shmat(shmid3, NULL, 0);
    if (SHM1 == (bool*)-1 || SHM2 == (int*)-1 || SHM3 == (int*)-1) {
        perror("Consumer shmat error:");
        exit(1);
    }

    bool* flag = SHM1;
    int* turn = SHM2;
    int* buf = SHM3;
    int index = 0;
    flag[i] = false;
    sleep(5);
    while (*state == 1) {
        flag[i] = true;

```

```

        printf("Consumer is ready now.\n\n");
        *turn = j;
        while (flag[j] == true && *turn == j)
            ;

        // Critical Section Begin
        if (buf[0] != 0) {
            printf("Job %d has been consumed\n", buf[0]);
            buf[0] = 0;
            index = 1;
            while (index < BSIZE) // Shifting remaining jobs forward
            {
                buf[index - 1] = buf[index];
                index++;
            }
            buf[index - 1] = 0;
        } else
            printf("Buffer is empty, nothing can be consumed!!!\n");
        printf("Buffer: ");
        index = 0;
        while (index < BSIZE)
            printf("%d ", buf[index++]);
        printf("\n");
        // Critical Section End

        flag[i] = false;
        if (*state == 0)
            break;
        wait_time = myrand(CWT);
        printf("Consumer will sleep for %d seconds\n\n", wait_time);
        sleep(wait_time);
    }
    exit(0);
}

// Parent process will now for RT seconds before causing child to terminate
while (1) {
    gettimeofday(&t, NULL);
    t2 = t.tv_sec;
    if (t2 - t1 > RT) // Program will exit after RT seconds
    {
        *state = 0;
        break;
    }
}

// Waiting for both processes to exit
wait();
wait();
printf("The clock ran out.\n");
return 0;
}

```

Output:

Producer is ready now.

Job 9 has been produced

Buffer: 9 0 0 0 0 0 0

Producer will wait for 1 seconds

Producer is ready now.

Job 8 has been produced

Buffer: 9 8 0 0 0 0 0

Producer will wait for 2 seconds

Producer is ready now.

Job 13 has been produced

Buffer: 9 8 13 0 0 0 0

Producer will wait for 1 seconds

Producer is ready now.

Job 23 has been produced

Buffer: 9 8 13 23 0 0 0

Producer will wait for 1 seconds

Consumer is ready now.

Job 9 has been consumed

Buffer: 8 13 23 0 0 0 0

Consumer will sleep for 9 seconds

Producer is ready now.

Job 15 has been produced

Buffer: 8 13 23 15 0 0 0

Producer will wait for 1 seconds

Producer is ready now.

Job 13 has been produced

Buffer: 8 13 23 15 13 0 0

Producer will wait for 1 seconds

Producer is ready now.

Job 11 has been produced

Buffer: 8 13 23 15 13 11 0

Producer will wait for 1 seconds

Producer is ready now.

Job 22 has been produced

Buffer: 8 13 23 15 13 11 22 0

Producer will wait for 2 seconds

Producer is ready now.

Job 23 has been produced

Buffer: 8 13 23 15 13 11 22 23

Producer will wait for 1 seconds

The clock ran out.

### **References:**

1. Operating System Concepts, 8<sup>th</sup> Edition, by Galvin et al, 2008, Wiley Publications.
2. Lecture notes and ppt of Ariel J. Frank, Bar-Ilan University.
3. Operating Systems | Internals and Design Principles | by William Stallings, Ninth Edition | By Pearson Publications
4. Web Portal: <https://www.geeksforgeeks.org>

## Unit 6

### I/O Management and Deadlock

#### Introduction

I/O software is often organized in the following layers –

- **User Level Libraries** – this provides simple interface to the user program to perform input and output. For example, **stdio** is a library provided by C and C++ programming languages.
- **Kernel Level Modules** – this provides device driver to interact with the device controller and device independent I/O modules used by the device drivers.
- **Hardware** – this layer includes actual hardware and hardware controller which interact with the device drivers and makes hardware alive.

A key concept in the design of I/O software is that it should be device independent where it should be possible to write programs that can access any I/O device without having to specify the device in advance. For example, a program that reads a file as input should be able to read a file on a floppy disk, on a hard disk, or on a CD-ROM, without having to modify the program for each different device.

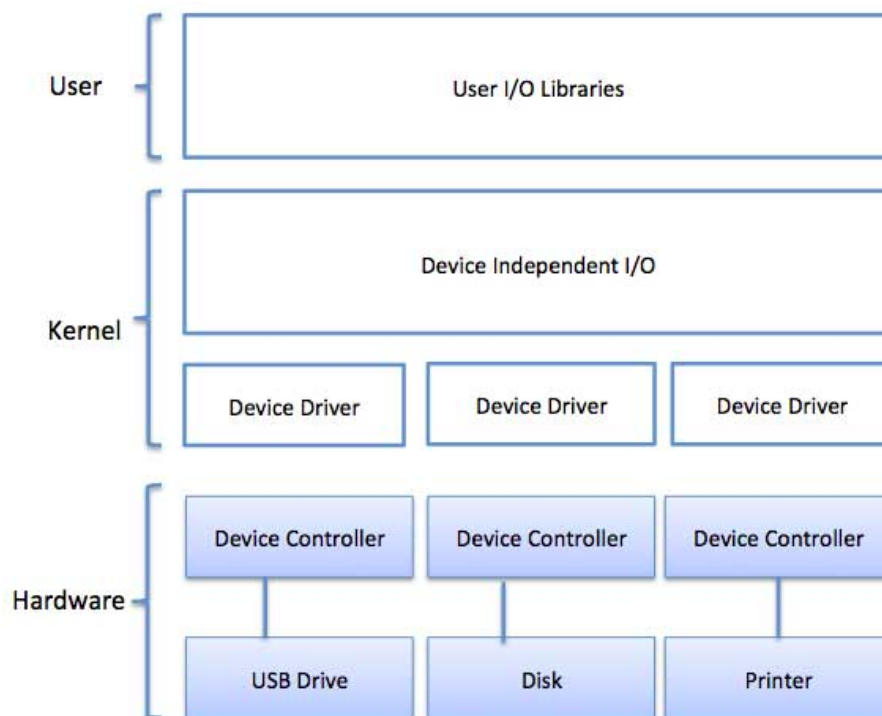


Fig. Operating System interface

## **Device Drivers**

Device drivers are software modules that can be plugged into an OS to handle a particular device. Operating System takes help from device drivers to handle all I/O devices. Device drivers encapsulate device-dependent code and implement a standard interface in such a way that code contains device-specific register reads/writes. Device driver, is generally written by the device's manufacturer and delivered along with the device on a CD-ROM.

A device driver performs the following jobs –

- To accept request from the device independent software above to it.
- Interact with the device controller to take and give I/O and perform required error handling
- Making sure that the request is executed successfully

How a device driver handles a request is as follows: Suppose a request comes to read a block N. If the driver is idle at the time a request arrives, it starts carrying out the request immediately. Otherwise, if the driver is already busy with some other request, it places the new request in the queue of pending requests.

### **Interrupt handlers:**

An interrupt handler, also known as an interrupt service routine or ISR, is a piece of software or more specifically a callback function in an operating system or more specifically in a device driver, whose execution is triggered by the reception of an interrupt.

When the interrupt happens, the interrupt procedure does whatever it has to in order to handle the interrupt, updates data structures and wakes up process that was waiting for an interrupt to happen.

The interrupt mechanism accepts an address — a number that selects a specific interrupt handling routine/function from a small set. In most architectures, this address is an offset stored in a table called the interrupt vector table. This vector contains the memory addresses of specialized interrupt handlers.

### **Device-Independent I/O Software**

The basic function of the device-independent software is to perform the I/O functions that are common to all devices and to provide a uniform interface to the user-level software. Though it is difficult to write completely device independent software but we can write some modules which are common among all the devices. Following is a list of functions of device-independent I/O Software –

1. Uniform interfacing for device drivers
2. Device naming - Mnemonic names mapped to Major and Minor device numbers
3. Device protection
4. Providing a device-independent block size
5. Buffering because data coming off a device cannot be stored in final destination.
6. Storage allocation on block devices
7. Allocation and releasing dedicated devices
8. Error Reporting

### User-Space I/O Software

These are the libraries which provide richer and simplified interface to access the functionality of the kernel or ultimately interactive with the device drivers. Most of the user-level I/O software consists of library procedures with some exception like spooling system which is a way of dealing with dedicated I/O devices in a multiprogramming system.

I/O Libraries (e.g., stdio) are in user-space to provide an interface to the OS resident device-independent I/O SW. For example putchar(), getchar(), printf() and scanf() are example of user level I/O library stdio available in C programming.

### Kernel I/O Subsystem

Kernel I/O Subsystem is responsible to provide many services related to I/O. Following are some of the services provided.

- **Scheduling** – Kernel schedules a set of I/O requests to determine a good order in which to execute them. When an application issues a blocking I/O system call, the request is placed on the queue for that device. The Kernel I/O scheduler rearranges the order of the queue to improve the overall system efficiency and the average response time experienced by the applications.
- **Buffering** – Kernel I/O Subsystem maintains a memory area known as **buffer** that stores data while they are transferred between two devices or between a device with an application operation. Buffering is done to cope with a speed mismatch between the producer and consumer of a data stream or to adapt between devices that have different data transfer sizes.
- **Caching** – Kernel maintains cache memory which is region of fast memory that holds copies of data. Access to the cached copy is more efficient than access to the original.
- **Spooling and Device Reservation** – A spool is a buffer that holds output for a device, such as a printer, that cannot accept interleaved data streams. The spooling system



copies the queued spool files to the printer one at a time. In some operating systems, spooling is managed by a system daemon process. In other operating systems, it is handled by an in kernel thread.

- **Error Handling** – An operating system that uses protected memory can guard against many kinds of hardware and application errors.

### **Terminal I/O(Terminal Hardware, Terminal):**

One of the important jobs of an Operating System is to manage various I/O devices including mouse, keyboards, touch pad, disk drives, display adapters, USB devices, Bit-mapped screen, LED, Analog-to-digital converter, On/off switch, network connections, audio I/O, printers etc.

An I/O system is required to take an application I/O request and send it to the physical device, then take whatever response comes back from the device and send it to the application. I/O devices can be divided into two categories –

- **Block devices** – A block device is one with which the driver communicates by sending entire blocks of data. For example, Hard disks, USB cameras, Disk-On-Key etc.
- **Character devices** – A character device is one with which the driver communicates by sending and receiving single characters (bytes, octets). For example, serial ports, parallel ports, sounds cards etc

### **Device Controllers**

Device drivers are software modules that can be plugged into an OS to handle a particular device. Operating System takes help from device drivers to handle all I/O devices. The Device Controller works like an interface between a device and a device driver. I/O units (Keyboard, mouse, printer, etc.) typically consist of a mechanical component and an electronic component where electronic component is called the device controller.

There is always a device controller and a device driver for each device to communicate with the Operating Systems. A device controller may be able to handle multiple devices. As an interface its main task is to convert serial bit stream to block of bytes, perform error correction as necessary.

Any device connected to the computer is connected by a plug and socket, and the socket is connected to a device controller. Following is a model for connecting the CPU, memory, controllers, and I/O devices where CPU and device controllers all use a common bus for communication.

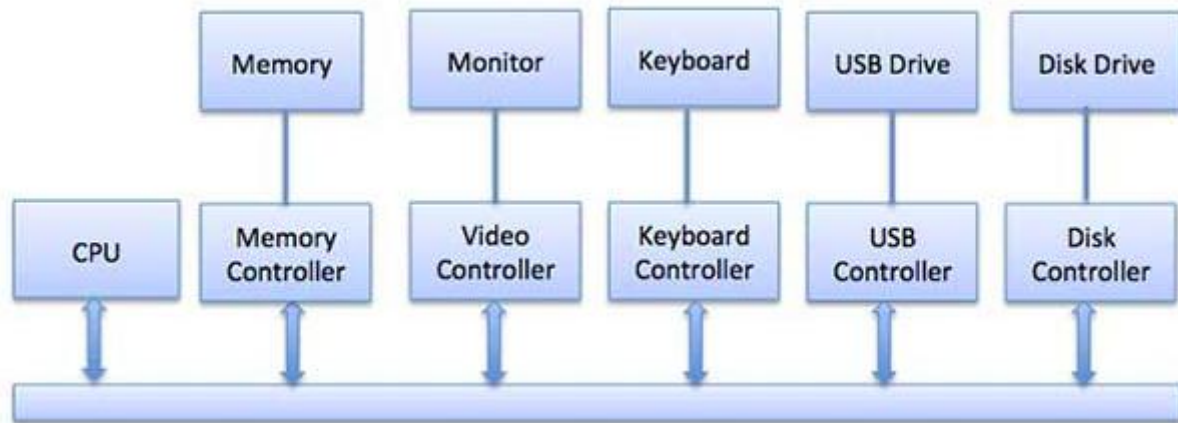


Fig. Device Controllers

### Synchronous vs asynchronous I/O

- **Synchronous I/O** – In this scheme CPU execution waits while I/O proceeds
- **Asynchronous I/O** – I/O proceeds concurrently with CPU execution

### Communication to I/O Devices

The CPU must have a way to pass information to and from an I/O device. There are three approaches available to communicate with the CPU and Device.

- Special Instruction I/O
- Memory-mapped I/O
- Direct memory access (DMA)

### Special Instruction I/O

This uses CPU instructions that are specifically made for controlling I/O devices. These instructions typically allow data to be sent to an I/O device or read from an I/O device.

### Memory-mapped I/O

When using memory-mapped I/O, the same address space is shared by memory and I/O devices. The device is connected directly to certain main memory locations so that I/O device can transfer block of data to/from memory without going through CPU.

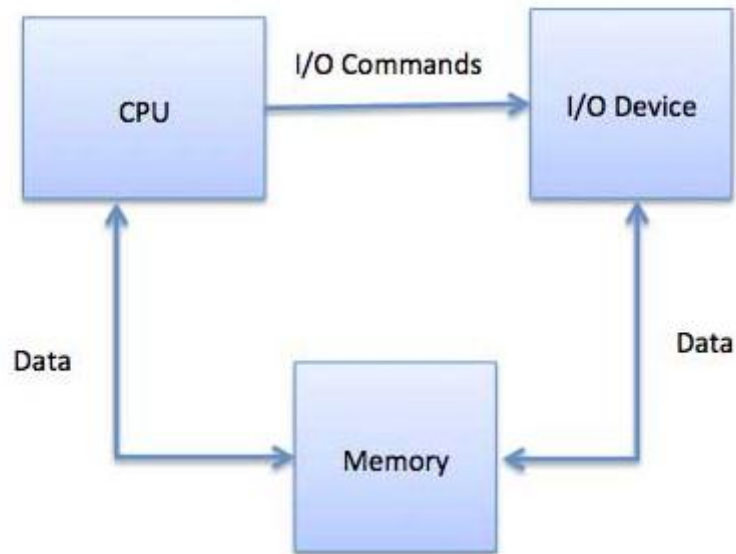


Fig. Memory Mapped IO

While using memory mapped IO, OS allocates buffer in memory and informs I/O device to use that buffer to send data to the CPU. I/O device operates asynchronously with CPU, interrupts CPU when finished.

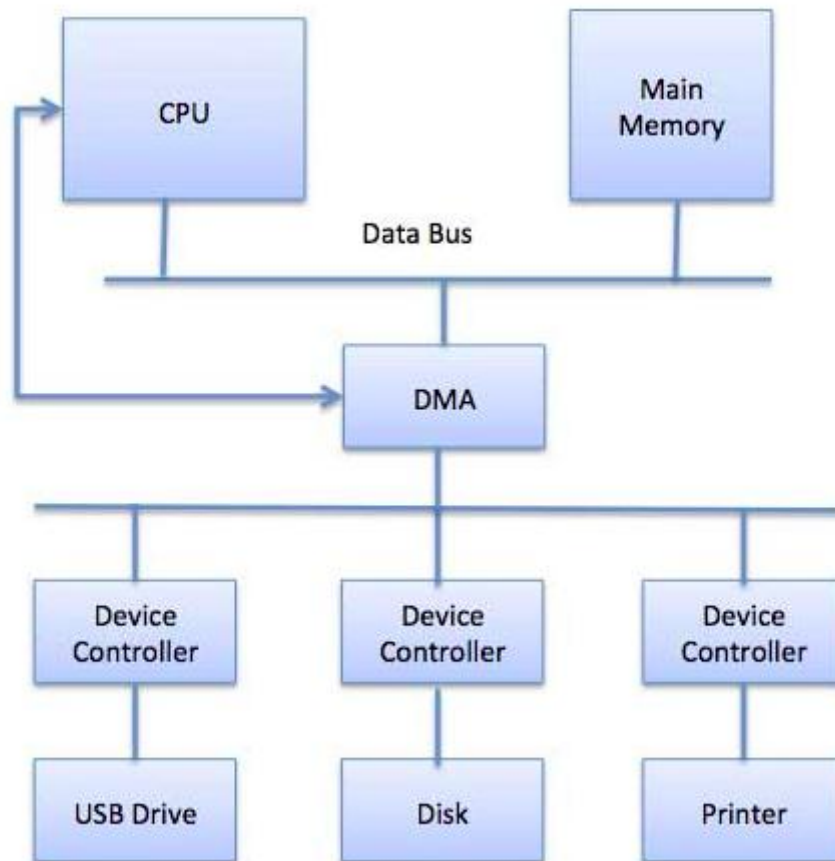
The advantage to this method is that every instruction which can access memory can be used to manipulate an I/O device. Memory mapped IO is used for most high-speed I/O devices like disks, communication interfaces.

### **Direct Memory Access (DMA)**

Slow devices like keyboards will generate an interrupt to the main CPU after each byte is transferred. If a fast device such as a disk generated an interrupt for each byte, the operating system would spend most of its time handling these interrupts. So a typical computer uses direct memory access (DMA) hardware to reduce this overhead.

Direct Memory Access (DMA) means CPU grants I/O module authority to read from or write to memory without involvement. DMA module itself controls exchange of data between main memory and the I/O device. CPU is only involved at the beginning and end of the transfer and interrupted only after entire block has been transferred.

Direct Memory Access needs a special hardware called DMA controller (DMAC) that manages the data transfers and arbitrates access to the system bus. The controllers are programmed with source and destination pointers (where to read/write the data), counters to track the number of transferred bytes, and settings, which includes I/O and memory types, interrupts and states for the CPU cycles.



The operating system uses the DMA hardware as follows –

Step	Description
1	Device driver is instructed to transfer disk data to a buffer address X.
2	Device driver then instruct disk controller to transfer data to buffer.
3	Disk controller starts DMA transfer.
4	Disk controller sends each byte to DMA controller.
5	DMA controller transfers bytes to buffer, increases the memory address, decreases the counter C until C becomes zero.
6	When C becomes zero, DMA interrupts CPU to signal transfer completion.

## **Polling vs Interrupts I/O**

A computer must have a way of detecting the arrival of any type of input. There are two ways that this can happen, known as **polling** and **interrupts**. Both of these techniques allow the processor to deal with events that can happen at any time and that are not related to the process it is currently running.

### **Polling I/O**

Polling is the simplest way for an I/O device to communicate with the processor. The process of periodically checking status of the device to see if it is time for the next I/O operation, is called polling. The I/O device simply puts the information in a Status register, and the processor must come and get the information.

Most of the time, devices will not require attention and when one does it will have to wait until it is next interrogated by the polling program. This is an inefficient method and much of the processors time is wasted on unnecessary polls.

Compare this method to a teacher continually asking every student in a class, one after another, if they need help. Obviously the more efficient method would be for a student to inform the teacher whenever they require assistance.

### **Interrupts I/O**

An alternative scheme for dealing with I/O is the interrupt-driven method. An interrupt is a signal to the microprocessor from a device that requires attention.

A device controller puts an interrupt signal on the bus when it needs CPU's attention when CPU receives an interrupt, It saves its current state and invokes the appropriate interrupt handler using the interrupt vector (addresses of OS routines to handle various events). When the interrupting device has been dealt with, the CPU continues with its original task as if it had never been interrupted.

## **Physical Organization of CD-ROM**

- Compact Disk – read only memory (write once)
- Data is encoded and read optically with a laser
- Can store around 600MB data

Digital data is represented as a series of Pits and Lands:

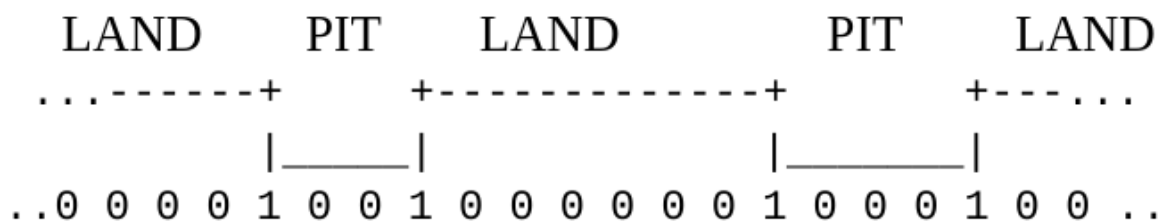
- Pit= a little depression, forming a lower level in the track
- Land= the flat part between pits, or the upper level in the track

Organization of data

Reading a CD is done by shining a laser at the disc and detecting changing reflection patterns.

1 = change in height (land to pit or pit to land)

0 = a "fixed" amount of time between 1's



Note : we cannot have two 1's in a row! => uses Eight to Fourteen Modulation (EFM) encoding table.

The basic unit of data stored on the CD-ROM is the *frame*. Each frame contains 24 bytes of input data. One byte of the source data occupies 17 bits on the disk: 14 bits - the code of the byte of the source data and 3 bits of the merge. For error correction, 180 bits are used. Thus, one frame on the disk occupies  $17 \cdot 24 + 180 = 588$  bits. The frames are combined into *sectors*. The sector contains 2352 data bytes (98 frames) and 882 bytes for error correction and control. The use of error correction algorithms allows providing a qualitative reading of information with an error probability of 10<sup>-10</sup> bits.

There are over 17,000 tracks on the working surface of the CD. Each track contains 32 *sectors* with ordinal numbers (addresses) from 0 to 31. Each sector stores user data and service information part with a total of 9996 bits. The sector is divided into 49 *segments* by 204 bits each. Segments are numbered from 0 to 48, with:

- in the zeroth segment is the sector index, which determines its beginning;
- The second segment contains the data providing the clock adjustment for playback (reading the disc);

- The third segment stores the service information: disk type and side, sector number on the track, etc.
- Segments 4 through 48 are used for data.

### General principles of writing/reading

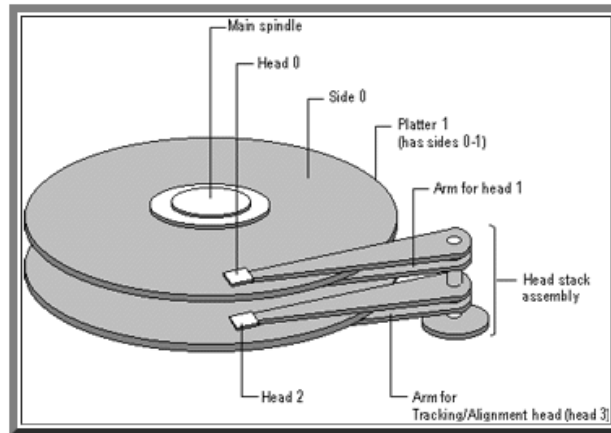
The processes of recording information on a CD (and reading from it) are based on geometric phenomena (reflection, absorption, transmission and refraction of light) and wave (interference, diffraction and polarization of light) optics. The CD contains a recording (working) layer, on which a signalogram is plotted with the help of a laser beam in the form of certain alternations of its states corresponding to a logical zero and a logical unit. In this case, the properties and optical characteristics of the recording layer material change. During the reading process, the laser beam captures these changes and converts them into a digital signal. The most widely used method of recording, leading to a change in the reflection coefficient of the material.

Optical recording achieves a high density of data on the disk. Its limit is due to diffraction of light and is determined by the minimum size of the label. The smaller the wavelength of the light flux emitted by the laser, the higher the recording density. Optical recording (and reading) of data is carried out through a transparent substrate, which is also used to protect the disk from damage. The following recording methods are known:

- ablation (ablation), a method of recording once, at which by heating the material of the recording layer by a laser beam its individual regions are removed;
- bubble a write-once-recording method in which the recording layer is expanded. When reading, the ray reflected from the expanded section (bubble) is scattered and therefore has a lower intensity compared to the ray reflected from the flat surface of the layer;
- The method of multiple recording, in which the phase state or the color of the recording layer portion for the data bit changes with the help of a laser beam. The refractive index or the absorption coefficient depends on the phase state of the recording layer material.

The data on the CD are located along the track in the form of a *waveform*, which can be represented as a set of depressions and areas. The shape of the signalogram, or the configuration of the labels, depends on the way the logical levels are coded. As an example, Fig. 6.11 shows the signalograms for two ways of encoding the logical levels with the same source data code 110101:

## Structure of a hard disk

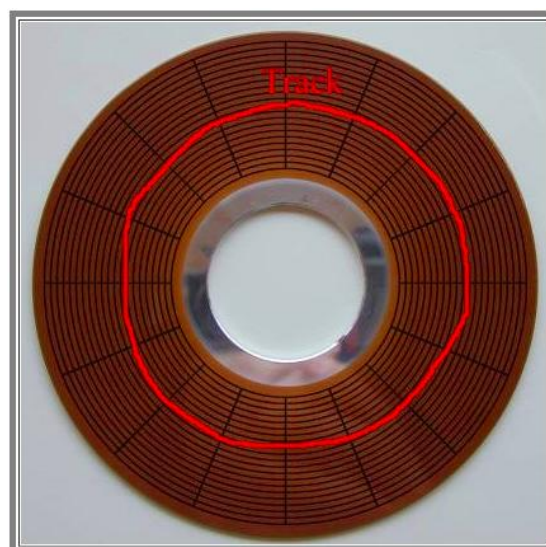


- Hard disk consists of a number of platters
- The platters rotate at a very high speed (5400 RPM to 10,000 RPM)
- Disk (read/write) heads move over the platter surface to read and write (magnetize) data bits
- The disk head can read or write data only when the desired disk surface area is under the disk head.

Data access time of data on disk consists of:

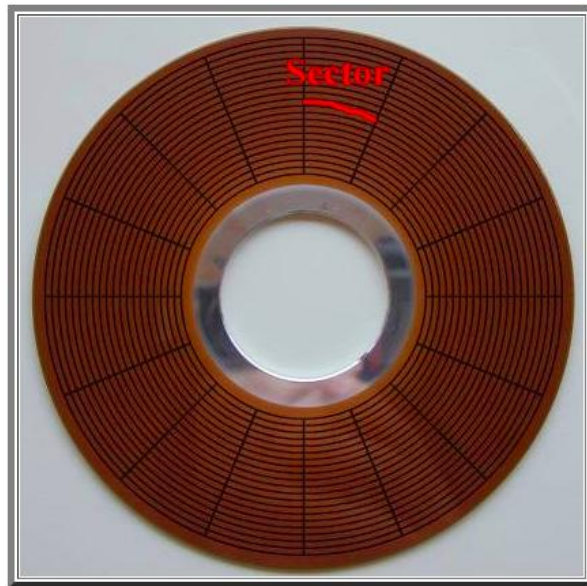
1. Seek time (get to the right track)
2. Latency (wait for the right sector to rotate under the disk head)
3. Transfer time (actual reading of the data)

Each platter is logically divided into a number of tracks





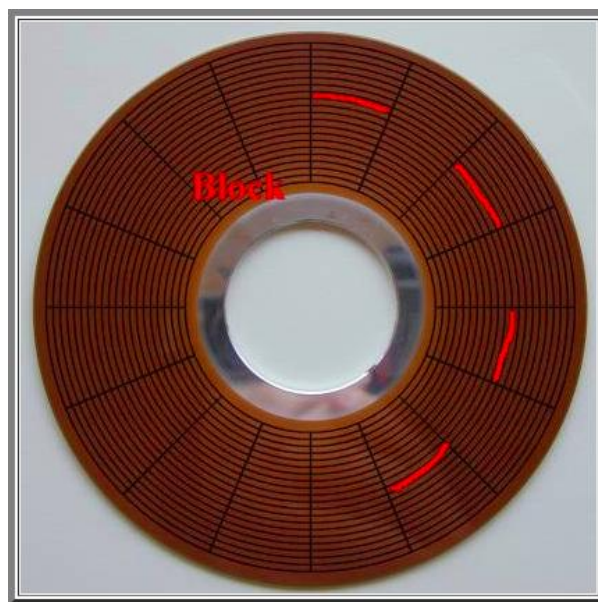
Each track is logically divided into a number of sectors:



Each sector is uniquely identified by its sector number

Disk sectors are usually 512 bytes in size.

To facilitate data access (larger chunks) a number of sectors are **logically** groups into a **disk block**



- Each disk block contains a number of sections
- Each disk block is uniquely identified by its block number.

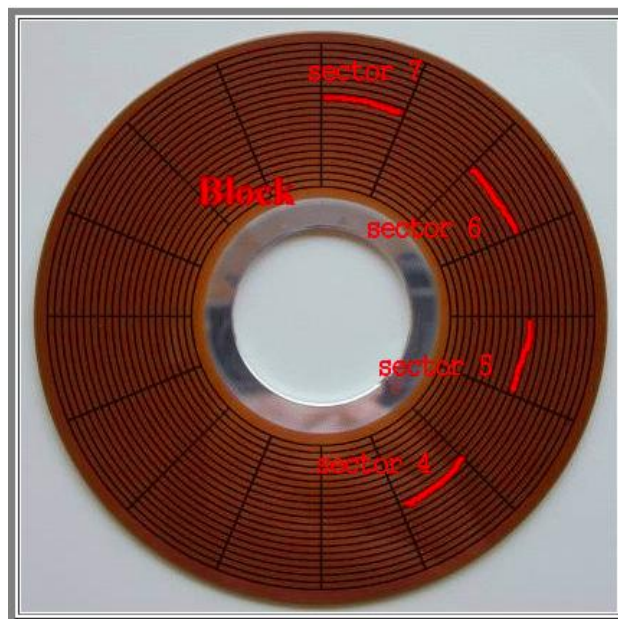
There is a mapping defined between disk block numbers and sector numbers

If 1 disk block contains N sectors, then

disk block k = sector number  $k*N$ ,  $k*N+1$ , ...,  $k*N + (N-1)$

Disk blocks are usually 4K or 8 K bytes in size.

Sectors of a disk block need not be located "contiguously" on the platter (for speedier access, sectors are usually interspersed)  
E.g., the sectors (4, 5, 6, 7) of this block is **interspersed**:



The reason to do so is:

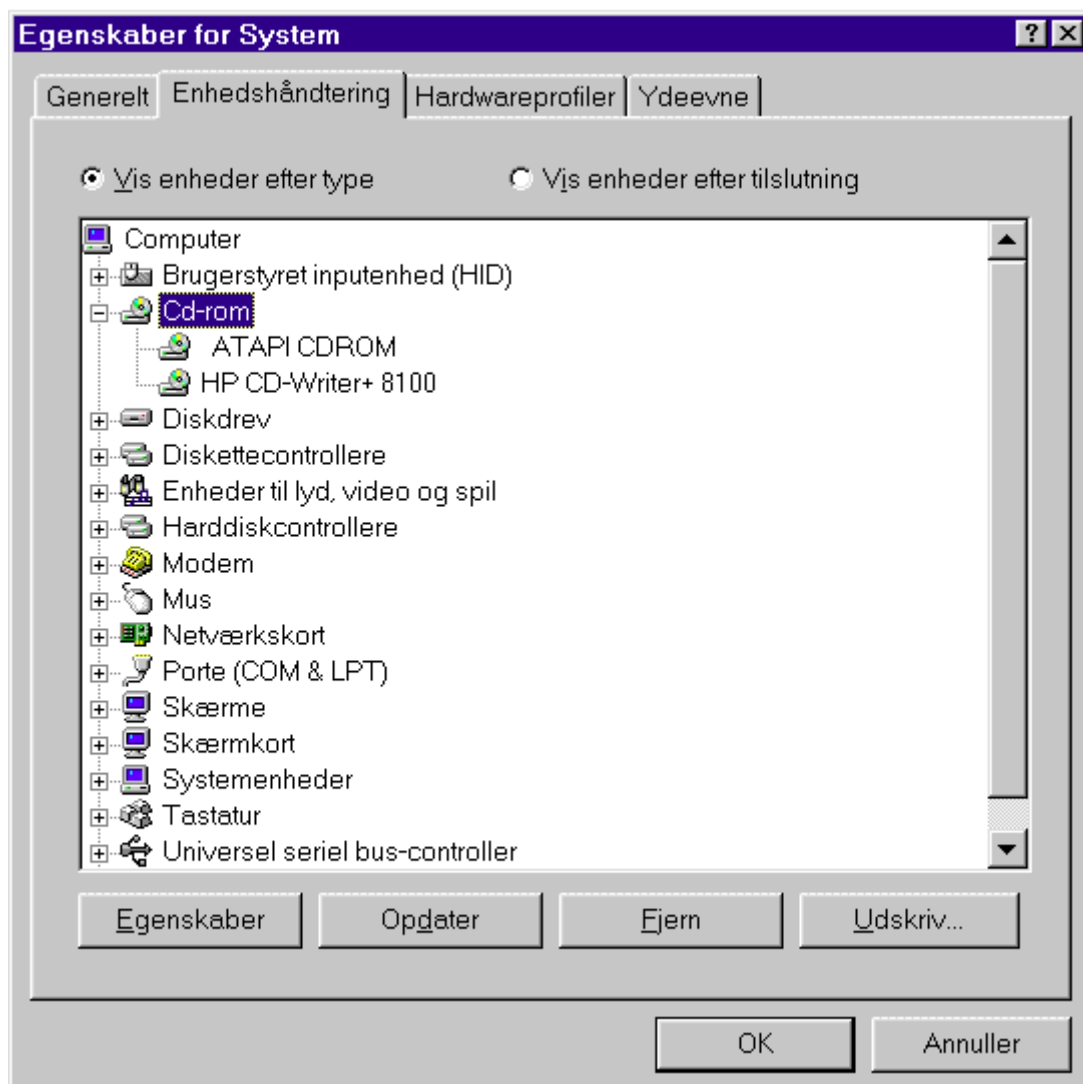
- Sectors of a block will be read consecutively (one after another)
- Furthermore, the disk drive will perform integrity checks on the data (This takes up some time; in the mean time, the disk drive will keep rotating !!!).

- If the sectors are located contiguously, then the disk head may have *passed* the start of the next sector when the drive finishes checking.... In that case, the drive will have to wait one whole revolution of the platter to read the next sector.
- By interspersing the sectors, the disk drive can achieve better access time for the consecutive sectors read/write

## Drives and operating system

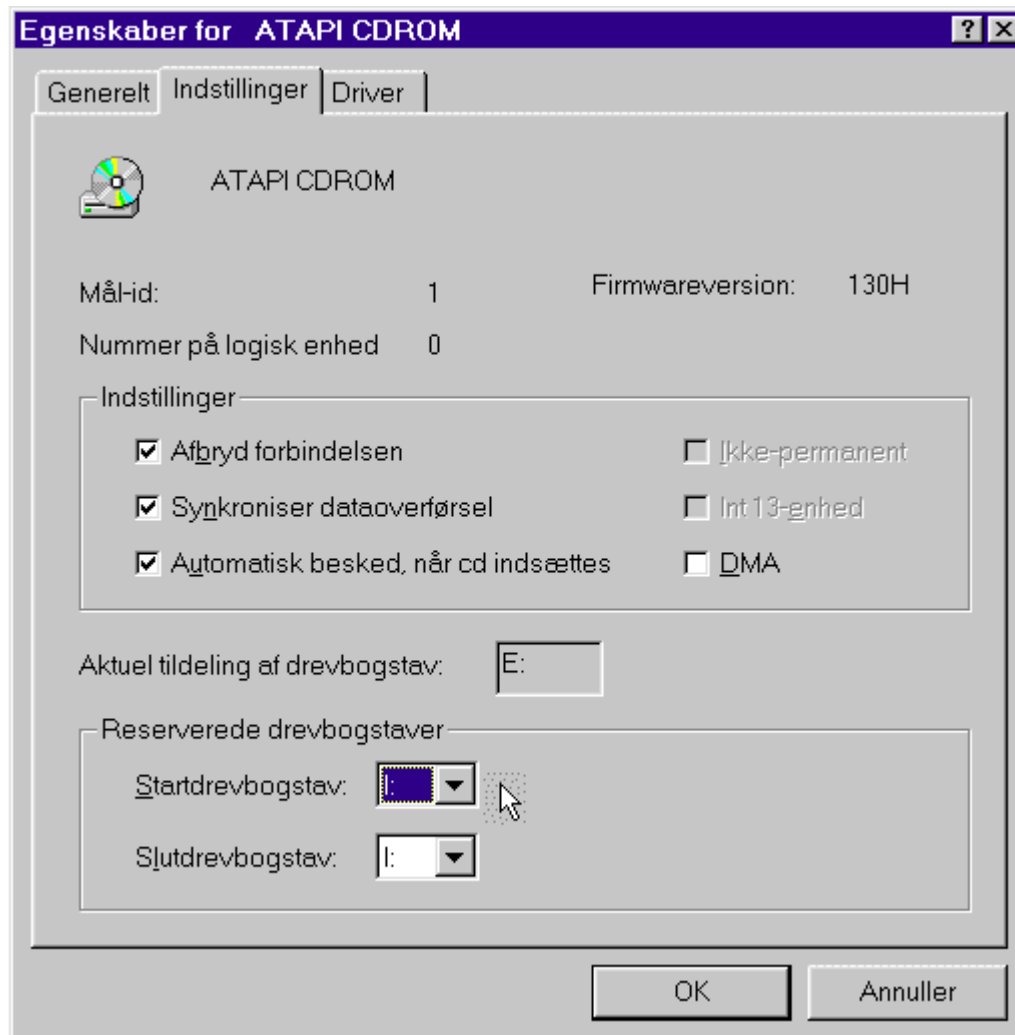
The drive must be assigned a *drive* letter. That is a task for the operating system, which must be able to recognize the CD-ROM drive. That is usually no problem in Windows 95/98. However, the alphabet can be quite messy, if there are many different drives attached.

Each drive must have its own letter. They are assigned on a first come first-serve-basis. The CD-ROM drive usually gets the first vacant letter after other existing drives, typically D, E, or F. But the letter can be changed in Windows. If you hit Win+Pause, the System box opens. Find your CD-ROM drives like here (The box is Danish, but you'll find it) :



Highlight the drive and choose Properties. Then you can arrange the drive letters:

I like to reserve the letters I: and J: for CD drives. Once the CD-ROM spins and the operating system (DOS or Windows ) has "found" the CD-ROM drive, data can be read for processing. Now the CD-ROM works like any other drive. Only, it is Read Only Memory! The CD-ROM holds its own file system called ISO 9660. It is not using FAT!



### **An introduction to the DVD:**

The DVD is a high-capacity optic media. The DVD standard was developed in the mid 1990s by leading companies like Philips and Sony. DVD stands for Digital Versatile Disk.

The DVD is an all-round disk, which probably will replace CD-ROM and laser disks. Over a few years DVD should replace VHS tapes for videos as well.

Some DVD drives can both read and write the disks. The drives are sold in many versions and with many incompatible sub-standards.

## A CD-like disk

The DVD is a flat disk of the same size as a CD. It holds a diameter of 4.7 inches (12 cm) and is .05 inches (1.2 mm) thick. Data are stored in a small indentation in a spiral track, just like in the CD, only the tracks are more narrow.

DVD disks are read by a laser beam of shorter wave-length than used by the CD-ROM drives. This allows for smaller indentations and increased storage capacity.

The data layer is only half as thick as in the CD-ROM. This opens the possibility to write data in two layers. The outer gold layer is semi transparent, to allow reading of the underlying silver layer. The laser beam is set to two different intensities, strongest for reading the underlying silver layer. Here you see a common type DVD ROM drive:



The DVD drives come in EIDE and SCSI editions and in 5X, etc. versions, like do the CD-ROMs.



The DVD drives are often bundled with a MPEG-2 decoder. This is required if you want to replay DVD video disks at optimal quality. Some graphics cards like Matrox-G400 MAX

come with a Cinemaster-based software decoder. This works together with the graphics accelerator chip and gives reasonable DVD replay quality.

The DVD drives will not replace the magnetic hard disks. The hard disks are being improved as rapidly as DVD, and they definitely offer the fastest seek time and transmission rate (currently 20-30 MB/second). No optic media can keep up with this nor with the speedy seeks we get from the harddisks.

But the DVD will undoubtedly gain a place as the successor to the CD-ROM. New drives will read both CD-ROMs and DVDs.

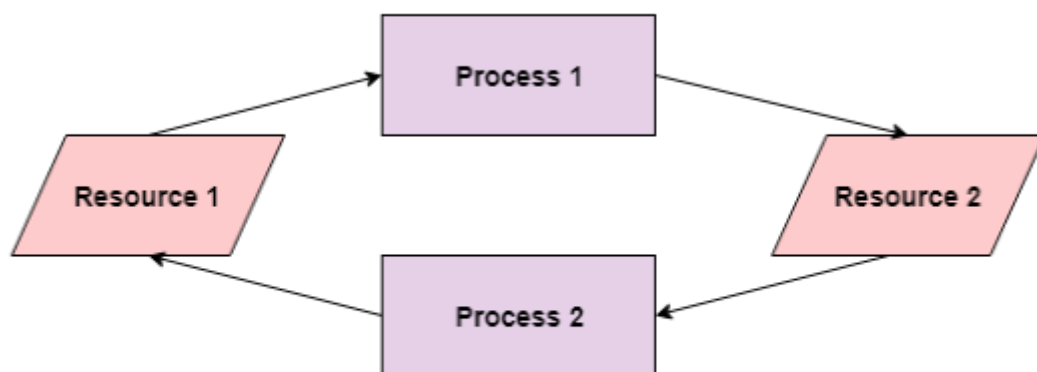
## DVD RAM

Three writable technologies are present at the market: Pioneer has a DVD-Recordable technology placing 3.95 GB per disk. DVD-RAM is a RW-disk from Hitachi and Matsushita. The 1. generation disks hold 3.6 GB, while the 2. generation hold 4.7 GB. The disks are held in a special cartridge. The so-called DVD+RW, supported by HP, Sony, Philips, Yamaha, Ricoh and Mitsubishi holds up to 4.7 GB per disk.

None of the three products are compatible. However, the companies behind DVD+RW control 75% of the market, so I think this will become the new standard. It appears that the DVD-RAM disks are extremely sensitive to greasy fingers and other contaminants. Therefore they must be handled in special cassettes, which do not fit into ordinary DVD players.

## Deadlocks

A deadlock happens in operating system when two or more processes need some resource to complete their execution that is held by the other process.



Deadlock in Operating System

In the above diagram, the process 1 has resource 1 and needs to acquire resource 2. Similarly process 2 has resource 2 and needs to acquire resource 1. Process 1 and process 2 are in deadlock as each of them needs the other's resource to complete their execution but neither of them is willing to relinquish their resources.

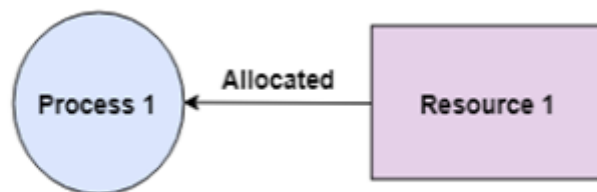
### Coffman Conditions

A deadlock occurs if the four Coffman conditions hold true. But these conditions are not mutually exclusive.

The Coffman conditions are given as follows –

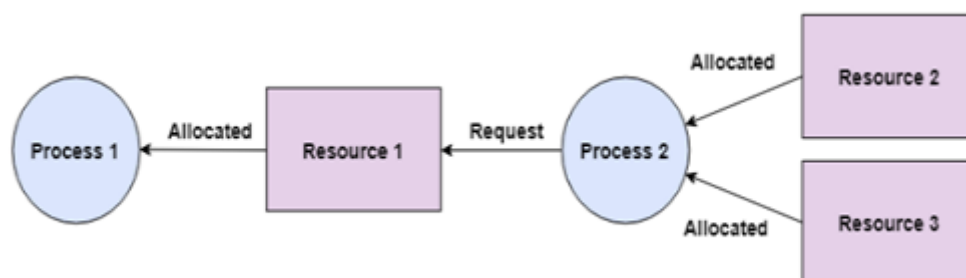
#### Mutual Exclusion

There should be a resource that can only be held by one process at a time. In the diagram below, there is a single instance of Resource 1 and it is held by Process 1 only.



#### Hold and Wait

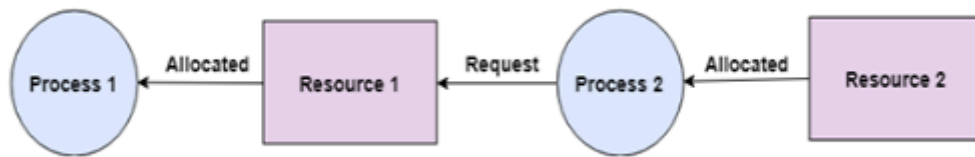
A process can hold multiple resources and still request more resources from other processes which are holding them. In the diagram given below, Process 2 holds Resource 2 and Resource 3 and is requesting the Resource 1 which is held by Process 1.



#### No Preemption

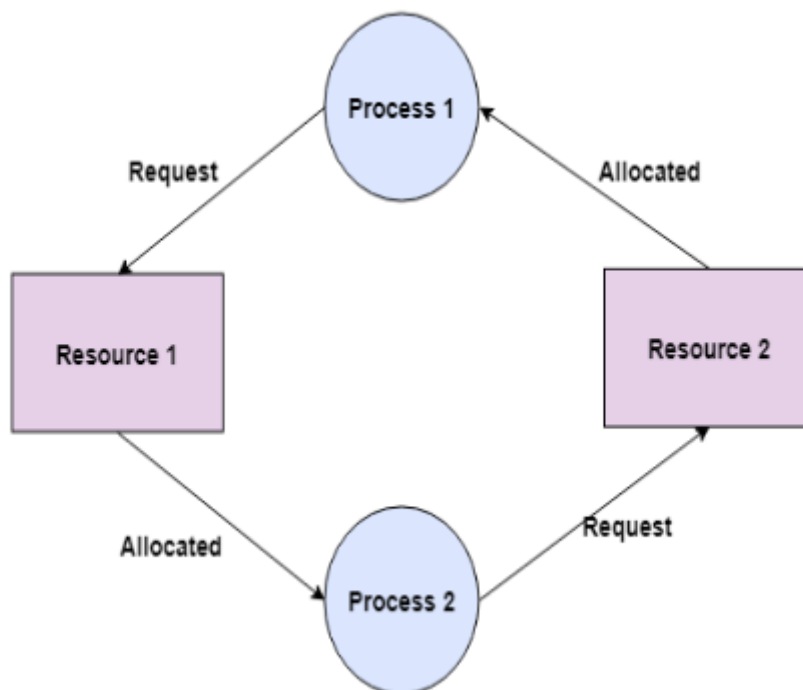
A resource cannot be preempted from a process by force. A process can only release a resource voluntarily. In the diagram below, Process 2 cannot preempt Resource 1 from Process 1. It will only be released when Process 1 relinquishes it voluntarily after its execution is complete.





### Circular Wait

A process is waiting for the resource held by the second process, which is waiting for the resource held by the third process and so on, till the last process is waiting for a resource held by the first process. This forms a circular chain. For example: Process 1 is allocated Resource2 and it is requesting Resource 1. Similarly, Process 2 is allocated Resource 1 and it is requesting Resource 2. This forms a circular wait loop.



### Deadlock Detection

A deadlock can be detected by a resource scheduler as it keeps track of all the resources that are allocated to different processes. After a deadlock is detected, it can be resolved using the following methods –

- All the processes that are involved in the deadlock are terminated. This is not a good approach as all the progress made by the processes is destroyed.
- Resources can be preempted from some processes and given to others till the deadlock is resolved.



## Deadlock Prevention

It is very important to prevent a deadlock before it can occur. So, the system checks each transaction before it is executed to make sure it does not lead to deadlock. If there is even a slight chance that a transaction may lead to deadlock in the future, it is never allowed to execute.

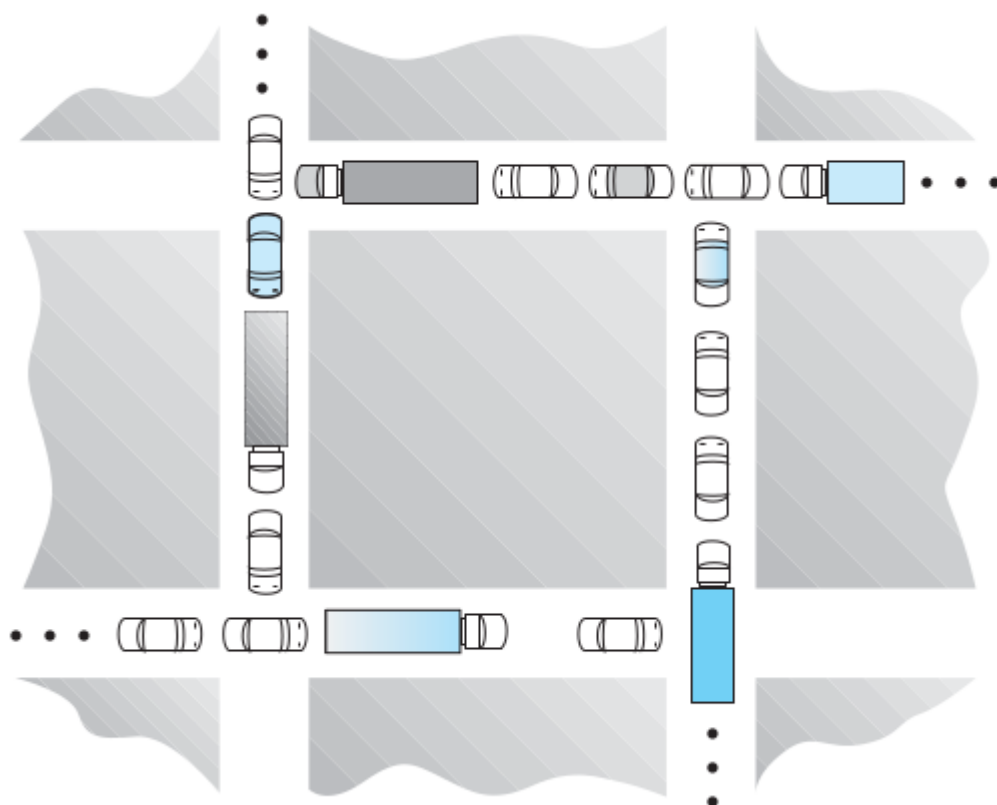
## Deadlock Avoidance

It is better to avoid a deadlock rather than take measures after the deadlock has occurred. The wait for graph can be used for deadlock avoidance. This is however only useful for smaller databases as it can get quite complex in larger databases.

Problem statements on deadlock:

Deadlock is a problem that can only arise in a system with multiple active asynchronous processes. It is important that the students learn the three basic approaches to deadlock: prevention, avoidance, and detection (although the terms prevention and avoidance are easy to confuse).

It can be useful to pose a deadlock problem in human terms and ask why human systems never deadlock. Can the students transfer this understanding of human systems to computer systems?



Projects can involve simulation: create a list of jobs consisting of requests and releases of resources (single type or multiple types). Ask the students to allocate the resources to prevent deadlock. This basically involves programming the Banker's Algorithm. The survey paper by Coffman, Elphick, and Shoshani.

1) Consider the traffic deadlock depicted in Figure.

- a. Show that the four necessary conditions for deadlock indeed hold in this example.
- b. State a simple rule for avoiding deadlocks in this system.

**Answer:**

- a. The four necessary conditions for a deadlock are (1) mutual exclusion; (2) hold-and-wait; (3) no preemption; and (4) circular wait.

The mutual exclusion condition holds as only one car can occupy a space in the roadway. Hold-and-wait occurs where a car holds onto their place in the roadway while they wait to advance in the roadway. A car cannot be removed (i.e. preempted) from its position in the roadway. Lastly, there is indeed a circular wait as each car is waiting for a subsequent car to advance. The circular wait condition is also easily observed from the graphic.

- b. A simple rule that would avoid this traffic deadlock is that a car may not advance into an intersection if it is clear they will not be able to immediately clear the intersection.

2) Consider the deadlock situation that could occur in the dining-philosophers problem when the philosophers obtain the chopsticks one at a time. Discuss how the four necessary conditions for deadlock indeed hold in this setting. Discuss how deadlocks could be avoided by eliminating any one of the four conditions.

**Answer:**

Deadlock is possible because the four necessary conditions hold in the following manner:

- 1) mutual exclusion is required for chopsticks,
- 2) the philosophers tend to hold onto the chopstick in hand while they wait for the other chopstick, 3) there is no preemption of chopsticks in the sense that a chopstick allocated to a philosopher cannot be forcibly taken away, and
- 4) there is a possibility of circular wait.

Deadlocks could be avoided by overcoming the conditions in the following manner:

- 1) allow simultaneous sharing of chopsticks,

2) have the philosophers relinquish the first chopstick if they are unable to obtain the other chopstick,

3) allow for chopsticks to be forcibly taken away if a philosopher has had a chopstick for a long period of time, and

4) enforce a numbering of the chopsticks and always obtain the lower numbered chopstick before obtaining the higher numbered one.

3) A possible solution for preventing deadlocks is to have a single, higher-order resource that must be requested before any other resource. For example, if multiple threads attempt to access the synchronization objects  $A \cdot \cdot \cdot E$ , deadlock is possible. (Such synchronization objects may include mutexes, semaphores, condition variables, etc.) We can prevent the deadlock by adding a sixth object  $F$ . Whenever a thread wants to acquire the synchronization lock for any object  $A \cdot \cdot \cdot E$ , it must first acquire the lock for object  $F$ . This solution is known as containment: The locks for objects  $A \cdot \cdot \cdot E$  are contained within the lock for object  $F$ . Compare this scheme with the circular-wait scheme of Section.

**Answer:**

This is probably not a good solution because it yields too large a scope. It is better to define a locking policy with as narrow a scope as possible.

4) Compare the circular-wait scheme with the deadlock-avoidance schemes (like the banker's algorithm) with respect to the following issues

a. Runtime overheads

b. System throughput

**Answer:**

A deadlock-avoidance scheme tends to increase the runtime overheads due to the cost of keep track of the current resource allocation. However, a deadlock-avoidance scheme allows for more concurrent use of resources than schemes that statically prevent the formation of deadlock. In that sense, a deadlock-avoidance scheme could increase system throughput.

5) In a real computer system, neither the resources available nor the demands of processes for resources are consistent over long periods (months). Resources break or are replaced, new processes come and go, new resources are bought and added to the system. If deadlock is controlled by the banker's algorithm, which of the following changes can be made safely (without introducing the possibility of deadlock), and under what circumstances?

a. Increase Available (new resources added).

b. Decrease Available (resource permanently removed from system)

c. Increase Max for one process (the process needs more resources than allowed, it may want more)

d. Decrease Max for one process (the process decides it does not need that many resources)

- e. Increase the number of processes.
- f. Decrease the number of processes.

**Answer:**

- a. Increase Available (new resources added) - This could safely be changed without any problems.
- b. Decrease Available (resource permanently removed from system)- This could have an effect on the system and introduce the possibility of deadlock as the safety of the system assumed there were a certain number of available resources.
- c. Increase Max for one process (the process needs more resources than allowed, it may want more) - This could have an effect on the system and introduce the possibility of deadlock.
- d. Decrease Max for one process (the process decides it does not need that many resources) - This could safely be changed without any problems.
- e. Increase the number of processes - This could be allowed assuming that resources were allocated to the new process(es) such that the system does not enter an unsafe state.
- f. Decrease the number of processes - This could safely be changed without any problems.

6) Consider a system consisting of four resources of the same type that are shared by three processes, each of which needs at most two resources. Show that the system is deadlock-free.

**Answer:**

Suppose the system is deadlocked. This implies that each process is holding one resource and is waiting for one more. Since there are three processes and four resources, one process must be able to obtain two resources. This process requires no more resources and, therefore it will return its resources when done.

7) Consider a system consisting of  $m$  resources of the same type, being shared by  $n$  processes. Resources can be requested and released by processes only one at a time. Show that the system is deadlock free if the following two conditions hold:

- a. The maximum need of each process is between 1 and  $m$  resources
- b. The sum of all maximum needs is less than  $m + n$

**Answer:**

Using the terminology of Section 6 we have:

$$a. \sum_{i=1}^n Max_i < m + n$$

$$b. Max_i \geq 1 \text{ for all } i$$

$$\text{Proof: } Need_i = Max_i - Allocation_i$$

If there exists a deadlock state then:

$$c. \sum_{i=1}^n Allocation_i = m$$

$$\text{Use a. to get: } \sum Need_i + \sum Allocation_i = \sum Max_i < m + n$$

$$\text{Use c. to get: } \sum Need_i + m < m + n$$

$$\text{Rewrite to get: } \sum_{i=1}^n Need_i < n$$

This implies that there exists a process  $P_i$  such that  $Need_i = 0$ . Since  $Max_i \geq 1$  it follows that  $P_i$  has at least one resource that it can release.

Hence the system cannot be in a deadlock state.

8) Consider the dining-philosophers problem where the chopsticks are placed at the center of the table and any two of them could be used by a philosopher. Assume that requests for chopsticks are made one at a time. Describe a simple rule for determining whether a particular request could be satisfied without causing deadlock given the current allocation of chopsticks to philosophers.

**Answer:**

The following rule prevents deadlock: when a philosopher makes a request for the first chopstick, do not satisfy the request only if there is no other philosopher with two chopsticks and if there is only one chopstick remaining.

9) Consider the same setting as the previous problem. Assume now that each philosopher requires three chopsticks to eat and that resource requests are still issued separately. Describe some simple rules for determining whether a particular request could be satisfied without causing deadlock given the current allocation of chopsticks to philosophers.

**Answer:**

When a philosopher makes a request for a chopstick, allocate the request if: 1) the philosopher has two chopsticks and there is at least one chopstick remaining, 2) the philosopher has one chopstick and there is at least two chopsticks remaining, 3) there is at least one chopstick remaining, and there is at least one philosopher with three chopsticks, 4) the philosopher has no chopsticks, there are two chopsticks remaining, and there is at least one other philosopher with two chopsticks assigned.

**10)** We can obtain the banker's algorithm for a single resource type from the general banker's algorithm simply by reducing the dimensionality of the various arrays by 1. Show through an example that the multiple- resource-type banker's scheme cannot be implemented by individual application of the single-resource-type scheme to each resource type.

**Answer:**

Consider a system with resources A, B, and C and processes P 0 , P 1 , P 2 , P 3 , and P 4 with the following values of Allocation:

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

And the following value of Need:

Need			
	A	B	C
P0	7	4	3
P1	0	2	0
P2	6	0	0
P3	0	1	1
P4	4	3	1

If the value of Available is (2 3 0), we can see that a request from process P 0 for (0 2 0) cannot be satisfied as this lowers Available to (2 1 0) and no process could safely finish.

However, if we were to treat the three resources as three single-resource types of the banker's algorithm, we get the following: For resource A (which we have 2 available),

	Allocated	Need
P0	0	7
P1	3	0
P2	3	6
P3	2	0
P4	0	4

Processes could safely finish in the order of P 1 , P 3 , P 4 , P 2 , P 0 . For resource B (which we now have 1 available as 2 were assumed assigned to process P 0 ),

	Allocated	Need
P0	3	2
P1	0	2
P2	0	0
P3	1	1
P4	0	3

Processes could safely finish in the order of P 2 , P 3 , P 1 , P 0 , P 4 . And finally, for For resource C (which we have 0 available),

	Allocated	Need
P0	0	3
P1	2	0
P2	2	0
P3	1	1
P4	2	1

Processes could safely finish in the order of P 1 , P 2 , P 0 , P 3 , P 4 .

As we can see, if we use the banker's algorithm for multiple resource types, the request for resources (0 2 0) from process P 0 is denied as it leaves the system in an unsafe state. However, if we consider the banker's algorithm for the three separate resources where we use a single resource type, the request is granted. Therefore, if we have multiple resource types, we must use the banker's algorithm for multiple resource types.

**Reference:**

- Operating System Concepts, 8<sup>th</sup> Edition, by Galvin et al, 2008, Wiley Publications.
- Lecture notes and ppt of Ariel J. Frank, Bar-Ilan University.
- **Operating Systems | Internals and Design Principles | by William Stallings, Ninth Edition | By Pearson Publications**
- <http://www.mathcs.emory.edu>
- <https://www.geeksforgeeks.org>
- <https://www.tutorialspoint.com>



## Unit 7

### Memory Management

In a uniprogramming system, main memory is divided into two parts: one part for the operating system (resident monitor, kernel) and one part for the program currently being executed. In a multiprogramming system, the “user” part of memory must be further subdivided to accommodate multiple processes. The task of subdivision is carried out dynamically by the operating system and is known as memory management.

Effective memory management is vital in a multiprogramming system. If only a few processes are in memory, then for much of the time all of the processes will be waiting for I/O and the processor will be idle. Thus memory needs to be allocated to ensure a reasonable supply of ready processes to consume available processor time.

#### Contiguous Memory Allocation

Contiguous memory allocation is a memory allocation method that allocates a single contiguous section of memory to a process or a file. This method takes into account the size of the file or a process and also estimates the maximum size, up to what the file or process can grow?

Taking into account the future growth of the file and its request for memory, the operating system allocates sufficient contiguous memory blocks to that file. Considering this future expansion and the file’s request for memory, the operating system will allocate those many contiguous blocks of memory to that file.

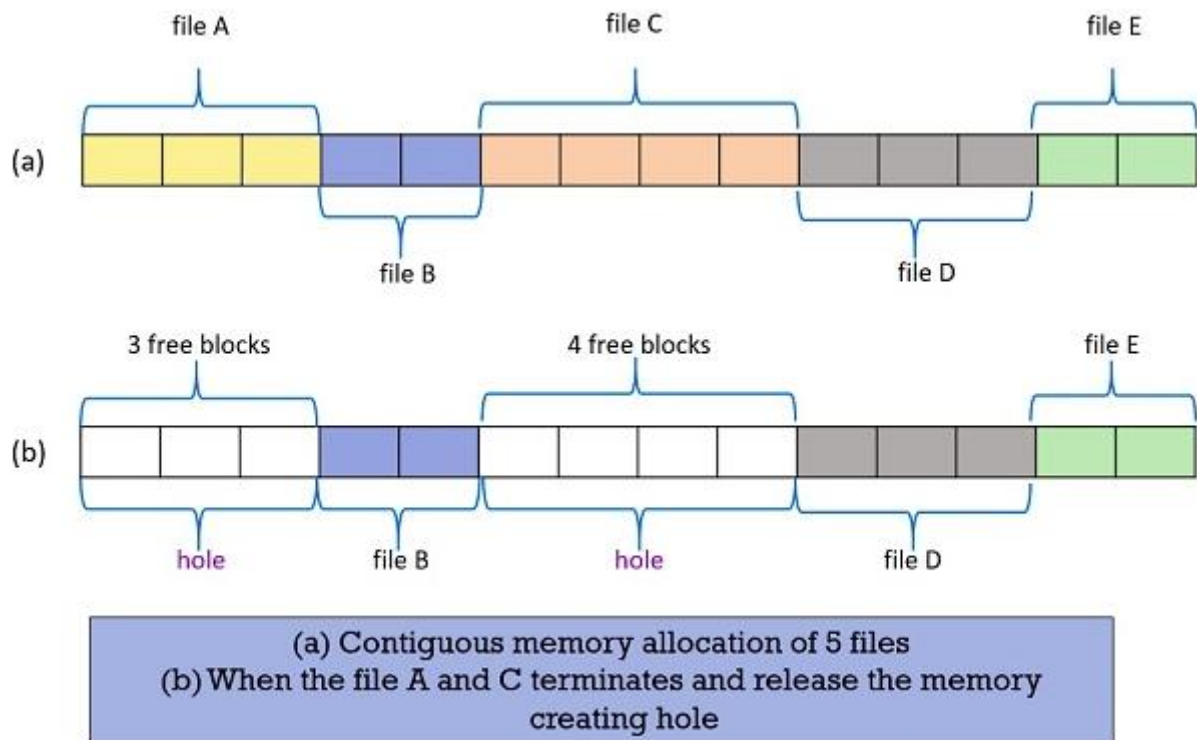
In this section, we will discuss contiguous memory allocation in detail along with its advantages and disadvantages. So let us start

#### Memory Allocation

The main memory has to accommodate both the **operating system** and **user space**. Now, here the user space has to accommodate various user processes. We also want these several user processes must reside in the main memory at the same time.

Now, the question arises how to allocate the available memory space to the user processes that are waiting in a ready queue?

In Contiguous memory allocation, when the process arrives from the ready queue to the main memory for execution, the contiguous memory blocks are allocated to the process according to its requirement. Now, to allocate the **contiguous space** to user processes, the memory can be dividing either in the fixed-sized partition or in the variable-sized partition.



**Fixed-Sized Partition:** In the fixed-sized partition, the memory is divided into fixed-sized blocks and each block contains exactly one process. But, the fixed-sized partition will limit the degree of multiprogramming as the number of the partition will decide the number of processes.

**Variable-Size Partition:** In the variable size partition method, the operating system maintains a table that contains the information about all memory parts that are **occupied by** the processes and all memory parts that are still **available for** the processes.

### How holes are created in the memory?

Initially, the whole memory space is available for the user processes as a large block, a hole. Eventually, when the processes arrive in the memory, executes, terminates and leaves the memory you will see the set of holes of variable sizes.

In the figure above, you can see that when file A and file C release the memory allocated to them, creates the holes in the memory of variable size.

In the **variable size partition method**, the operating system analyses the memory requirement of the process and see whether it has a memory block of the required size. If it finds the match, then it allocates that memory block to the process. If not, then it searches the ready queue for the process that has a smaller memory requirement.

The operating system allocates the memory to the process until it cannot satisfy the memory requirement of the next process in the ready queue. It stops allocating memory to the process if it does not have a memory block (**hole**) that is large enough to hold that process.

If the memory block (**hole**) is too **large** for the process it gets **split into** two parts. One part of the memory block is allocated to the arrived process and the other part is returned to the set of holes. When a process terminates and releases the memory allocated to it, the released memory is then placed back to the set of holes. The two holes that are adjacent to each other, in the set of holes, are merged to form one large hole.

Now, at this point, the operating system checks whether this newly formed free large hole is able to satisfy the memory requirement of any process waiting in the ready queue. And the process goes on.

## Memory Management

In the above section, we have seen how the operating system allocates the contiguous memory to the processes. Here, we will see how the operating system selects a free hole from the set of holes?

The operating system uses either the block allocation list or the bit map to select the hole from the set of holes.

### Block Allocation List

Block allocation list maintains two tables. One table contains the entries of the blocks that are allocated to the various files. The other table contains the entries of the holes that are free and can be allocated to the process in the waiting queue.

Now, as we have entries of free blocks which one must be chosen can be decided using either of these strategies: first-fit, best-fit, worst-fit strategies.

#### 1. First-fit

Here, the searching starts either at the beginning of the table or from where the previous first-fit search has ended. While searching, the first hole that is found to be large enough for a process to accommodate is selected.

#### 2. Best-fit

This method needs the list of free holes to be sorted according to their size. Then the smallest hole that is large enough for the process to accommodate is selected from the list of free holes. This strategy reduces the wastage of memory as it does not allocate a hole of larger size which leaves some amount of memory even after the process accommodates the space.

#### 3. Worst-fit

This method requires the entire list of free holes to be sorted. Here, again the largest hole among the free holes is selected. This strategy leaves the largest leftover hole which may be useful for the other process.

## Bit Map

The bit map method only keeps track of the free or allocated block. One block is represented by one bit, bit 0 resembles the free block and bit 1 resembles that the block is allocated to a file or a process.

1110000111100000001100000111000000111110

## Bit Map

It does not have entries of the files or processes to which the specific blocks are allocated. Normally, implementing the first fit will search the number of consecutive zeros/free blocks required by a file or process. Having found that much of consecutive zeros it allocates a file or process to those blocks.

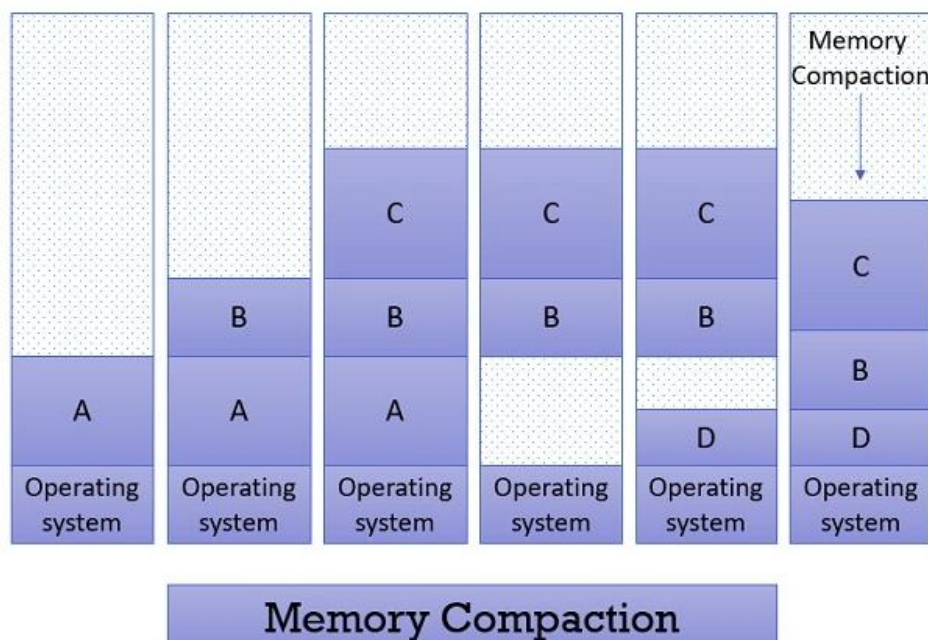
But implementing best-fit or worse-fit will be expensive, as the table of free blocks sorted according to the hole size has to be maintained. But the bit map method is easy to implement.

## Fragmentation

Fragmentation can either be external fragmentation or internal fragmentation. **External fragmentation** is when the free memory blocks available in memory are too small and even non-contiguous. Whereas, the **internal fragmentation** occurs when the process does not fully utilize the memory allocated to it.

The solution to the problem of external fragmentation is **memory compaction**. Here, all the memory contents are shuffled to the one area of memory thereby creating a large block of memory which can be allocated to one or more new processes.

As in the figure below, you can see at the last process C, B, D are moved downwards to make large hole.



## **Disadvantages**

The main **disadvantage** of contiguous memory allocation is **memory wastage and inflexibility**. As the memory is allocated to a file or a process keeping in mind that it will grow during the run. But until a process or a file grows many blocks allocated to it remains unutilized. And they even they cannot be allocated to the other process leading to wastage of memory.

In case, the process or the file grows beyond the expectation i.e. beyond the allocated memory block, then it will abort with the message “No disk space” leading to inflexibility.

## **Advantages**

The **advantage** of contiguous memory allocation is it increases the processing speed. As the operating system uses the buffered I/O and reads the process memory blocks consecutively it reduces the head movements. This speed ups the processing.

## **Key Takeaways**

- Memory allocation can be done either by a fixed-sized partition scheme or by variable-sized partition scheme.
- Block allocation list has three methods to select a hole from the list of free holes first-fit, best-fit and worse-fit.
- Bit map keeps track of free blocks in memory, it has one bit for one memory block, bit 0 shows that the block is free and bit 1 shows the block is allocated to some file or a process.
- Contiguous memory allocation leads to fragmentation. Further fragmentation can either be external or internal.
- Contiguous memory allocation leads to memory wastage and inflexibility. If the operating system uses buffered I/O during processing, then contiguous memory allocation can enhance processing speed.

## **Fixed Partitioned Memory Management**

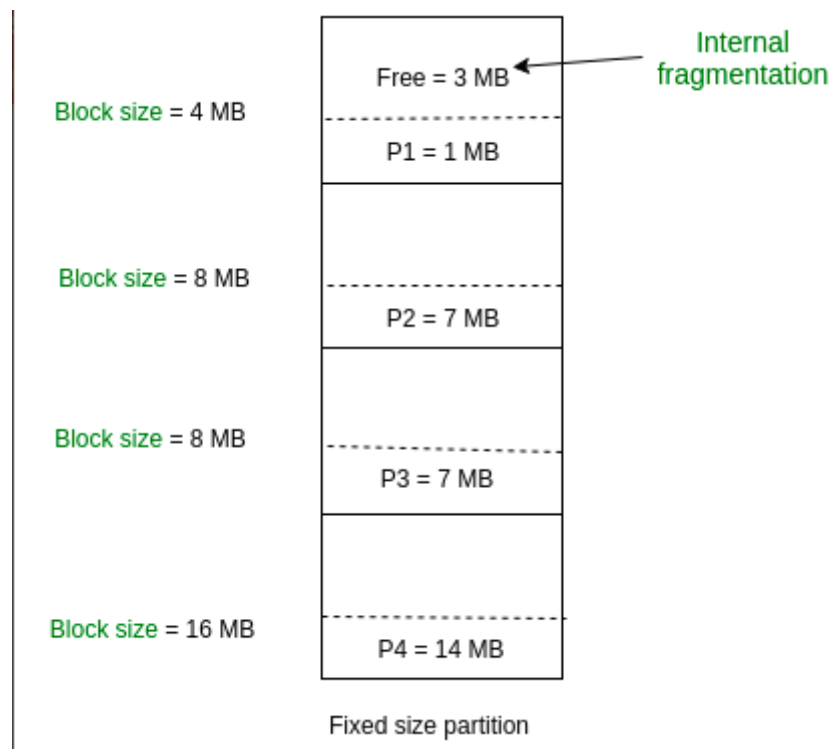
In operating systems, Memory Management is the function responsible for allocating and managing computer's main memory. Memory Management function keeps track of the status of each memory location, either allocated or free to ensure effective and efficient use of Primary Memory.

There are two Memory Management Techniques: Contiguous, and Non-Contiguous. In Contiguous Technique, executing process must be loaded entirely in main-memory. Contiguous Technique can be divided into:

1. Fixed (or static) partitioning
2. Variable (or dynamic) partitioning

Fixed Partitioning:

This is the oldest and simplest technique used to put more than one processes in the main memory. In this partitioning, number of partitions (non-overlapping) in RAM are fixed but size of each partition may or may not be same. As it is contiguous allocation, hence no spanning is allowed. Here partition are made before execution or during system configure.



As illustrated in above figure, first process is only consuming 1MB out of 4MB in the main memory.

Hence, Internal Fragmentation in first block is  $(4-1) = 3\text{MB}$ . Sum of Internal Fragmentation in every block =  $(4-1)+(8-7)+(8-7)+(16-14) = 3+1+1+2 = 7\text{MB}$ .

Suppose process P5 of size 7MB comes. But this process cannot be accommodated inspite of available free space because of contiguous allocation (as spanning is not allowed). Hence, 7MB becomes part of External Fragmentation.

There are some advantages and disadvantages of fixed partitioning.

## Advantages of Fixed Partitioning –

1. **Easy to implement:**

Algorithms needed to implement Fixed Partitioning are easy to implement. It simply requires putting a process into certain partition without focussing on the emergence of Internal and External Fragmentation.

2. **Little OS overhead:**

Processing of Fixed Partitioning require lesser excess and indirect computational power.

## Disadvantages of Fixed Partitioning

1. **Internal Fragmentation:**

**Main memory use is inefficient. Any program, no matter how small, occupies an entire partition. This can cause internal fragmentation.**

2. **External Fragmentation:**

The total unused space (as stated above) of various partitions cannot be used to load the processes even though there is space available but not in the contiguous form (as spanning is not allowed).

3. **Limit process size:**

Process of size greater than size of partition in Main Memory cannot be accommodated. Partition size cannot be varied according to the size of incoming process's size. Hence, process size of 32MB in above stated example is invalid.

4. **Limitation on Degree of Multiprogramming:**

Partition in Main Memory are made before execution or during system configure. Main Memory is divided into fixed number of partition. Suppose if there are  $n_1$  partitions in RAM and  $n_2$  are the number of processes, then  $n_2 \leq n_1$  condition must be fulfilled. Number of processes greater than number of partitions in RAM is invalid in Fixed Partitioning.

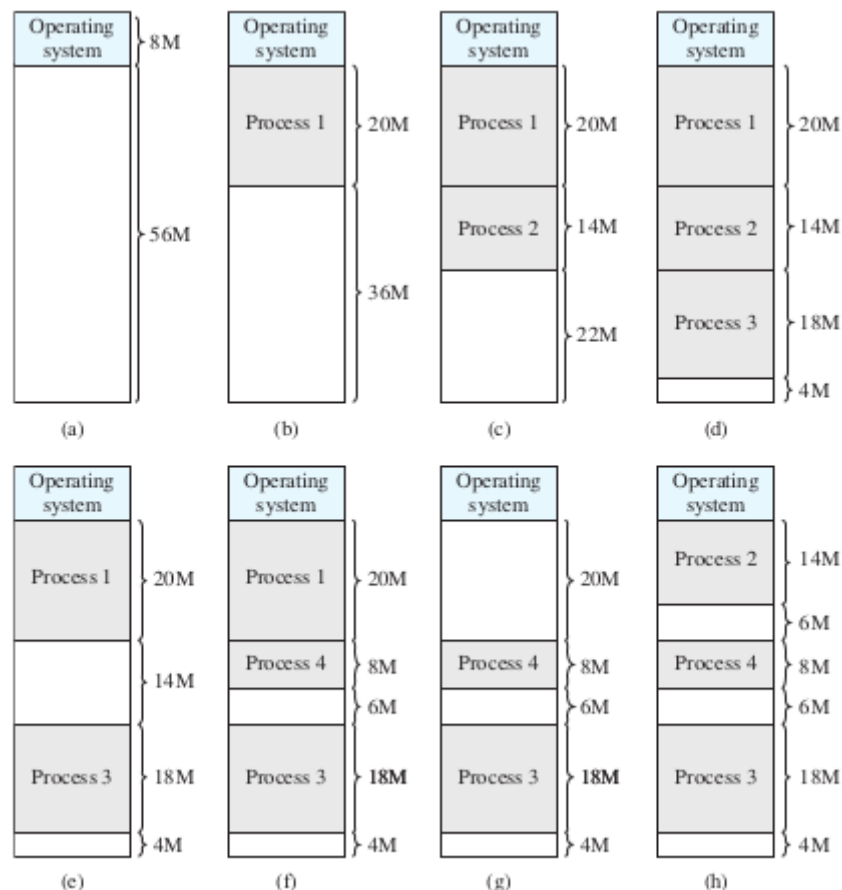
## Dynamic Partitioning

To overcome some of the difficulties with fixed partitioning, an approach known as dynamic partitioning was developed. Again, this approach has been supplanted by more sophisticated memory management techniques. An important operating system that used this technique was IBM's mainframe operating system, OS/MVT (Multiprogramming with a Variable Number of Tasks). With dynamic partitioning, the partitions are of variable length and number.

When a process is brought into main memory, it is allocated exactly as much memory as it requires and no more. An example, using 64 Mbytes of main memory, is shown in Figure Initially, main memory is empty, except for the operating system (a). The first three processes are loaded in, starting where the operating system ends and occupying just enough space for each process (b, c, d). This leaves a "hole" at the end of memory that is too small for a fourth process. At some point, none of the processes in memory is ready. The operating system swaps out process 2 (e), which leaves sufficient room to load a new process, process 4 (f). Because process 4 is smaller than process 2, another small hole is created. Later, a point is reached at which none of the processes in main memory is ready, but process 2, in the Ready-

Suspend state, is available. Because there is insufficient room in memory for process 2, the operating system swaps process 1 out (g) and swaps process 2 back in (h).

As this example shows, this method starts out well, but eventually it leads to a situation in which there are a lot of small holes in memory. As time goes on, memory becomes more and more fragmented, and memory utilization declines. This phenomenon is referred to as external fragmentation, indicating that the memory that is external to all partitions becomes increasingly fragmented. This is in contrast to internal fragmentation, referred to earlier. One technique for overcoming external fragmentation is compaction: From time to time, the operating system shifts the processes so that they are contiguous and so that all of the free memory is together in one block. For example, in Figure compaction will result in a block of free memory of length 16M. This may well be sufficient to load in an additional process. The difficulty with compaction is that it is a time consuming procedure and wasteful of processor time. Note that compaction implies the need for a dynamic relocation capability. That is, it must be possible to move a program from one region to another in main memory without invalidating the memory references in the program

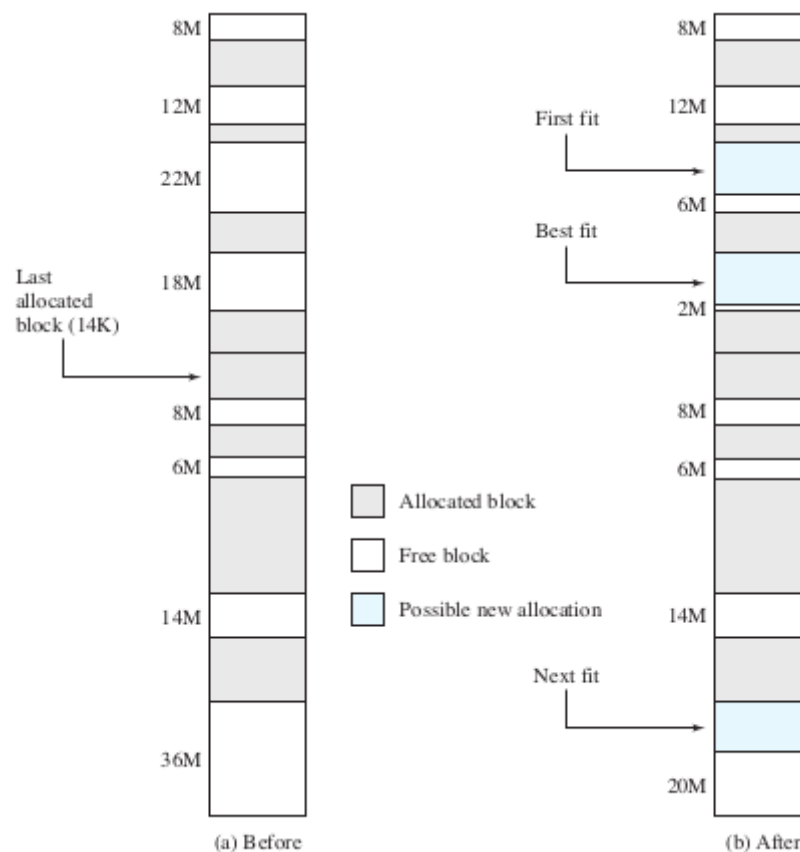


Placement Algorithm because memory compaction is time consuming, the operating system designer must be clever in deciding how to assign processes to memory (how to plug the holes). When it is time to load or swap a process into main memory, and if there is more than one free block of memory of sufficient size, then the operating system must decide which free block to allocate. Three placement algorithms that might be considered are best-fit, first-fit, and next-fit. All, of course, are limited to choosing among free blocks of main memory that



are equal to or larger than the process to be brought in. Best-fit chooses the block that is closest in size to the request. First-fit begins to scan memory from the beginning and chooses the first available block that is large enough. Next-fit begins to scan memory from the location of the last placement, and chooses the next available block that is large enough. Figure shows an example memory configuration after a number of placement and swapping-out operations. The last block that was used was a 22-Mbyte block from which a 14-Mbyte partition was created. Figure. shows the difference between the best-, first-, and next-fit placement algorithms in satisfying a 16-Mbyte allocation request. Best-fit will search the entire list of available blocks and make use of the 18-Mbyte block, leaving a 2-Mbyte fragment. First-fit results in a 6-Mbyte fragment, and next-fit results in a 20-Mbyte fragment.

Which of these approaches is best will depend on the exact sequence of process swappings that occurs and the size of those processes. The first-fit algorithm is not only the simplest but usually the best and fastest as well. The next-fit algorithm tends to produce slightly worse results than the first-fit. The next-fit algorithm will more frequently lead to an allocation from a free block at the end of memory. The result is that the largest block of free memory, which usually appears at the end of the memory space, is quickly broken up into small fragments. Thus, compaction may be required more frequently with next-fit. On the other hand, the first-fit algorithm may litter the front end with small free partitions that need to be searched over on each subsequent first-fit pass. The best-fit algorithm, despite its name, is usually the worst performer. Because this algorithm looks for the smallest block that will satisfy the requirement, it guarantees that the fragment left behind is as small as possible. Although each memory request always wastes the smallest amount of memory, the result is that main memory is quickly littered by blocks too small to satisfy memory allocation requests. Thus, memory compaction must be done more frequently than with the other algorithms.



**Replacement Algorithm** In a multiprogramming system using dynamic partitioning, there will come a time when all of the processes in main memory are in a blocked state and there is insufficient memory, even after compaction, for an additional process. To avoid wasting processor time waiting for an active process to become unblocked, the operating system will swap one of the processes out of main memory to make room for a new process or for a process in a Ready-Suspend state. Therefore, the operating system must choose which process to replace. Because the topic of replacement algorithms will be covered in some detail with respect to various virtual memory schemes, we defer a discussion of replacement algorithms until then.

### Buddy System

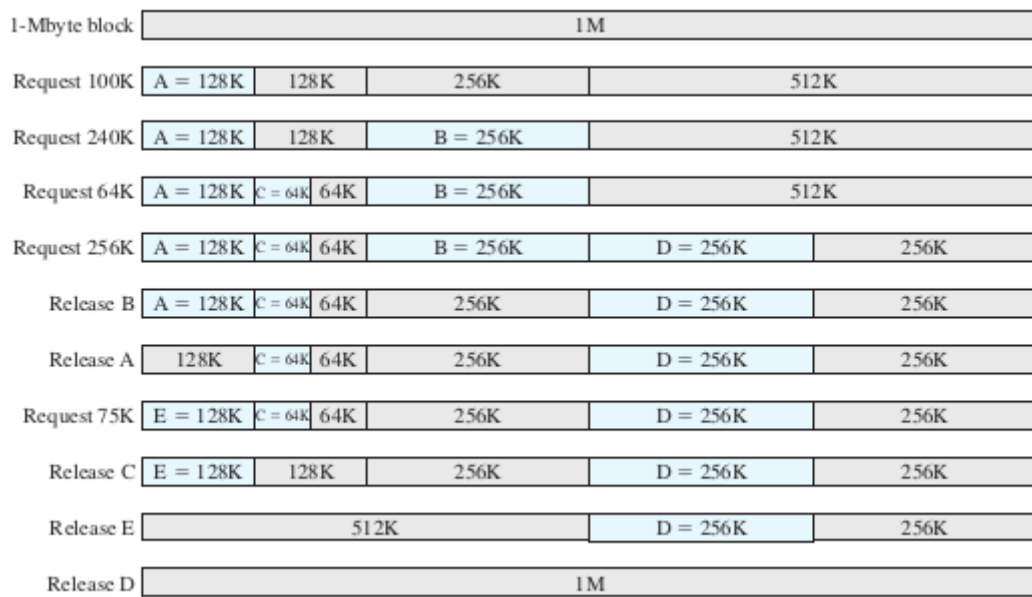
Both fixed and dynamic partitioning schemes have drawbacks. A fixed partitioning scheme limits the number of active processes and may use space inefficiently if there is a poor match between available partition sizes and process sizes. A dynamic partitioning scheme is more complex to maintain and includes the overhead of compaction.

In a buddy system, memory blocks are available of size  $2^k$  words,  $L \leq k \leq U$ ,

where  $2^L$  = smallest size block that is allocated  $2^U$  = largest size block that is allocated; generally  $2^U$  is the size of the entire memory available for allocation. To begin, the entire space available for allocation is treated as a single block of size  $2^U$ . If a request of size  $s$  such that  $2^{U-1} < s \leq 2^U$  is made, then the entire block is allocated. Otherwise, the block is split into two equal buddies of size  $2^{U-1}$ . If  $2^{U-2} < s \leq 2^{U-1}$ , then the request is allocated to one of the two buddies. Otherwise, one of the buddies is split in half again. This process continues until the smallest block greater than or equal to  $s$  is generated and allocated to the request. At any time, the buddy system maintains a list of holes (unallocated blocks) of each size  $2^i$ . A hole may be removed from the  $(i \neq 1)$  list by splitting it in half to create two buddies of size  $2^i$  in the  $i$  list. Whenever a pair of buddies on the  $i$  list both become unallocated, they are removed from that list and coalesced into a single block on the  $(i \neq 1)$  list. Presented with a request for an allocation of size  $k$  such that  $2^{i-1} < k \leq 2^i$ , the following recursive algorithm is used to find a hole of size  $2^i$ :

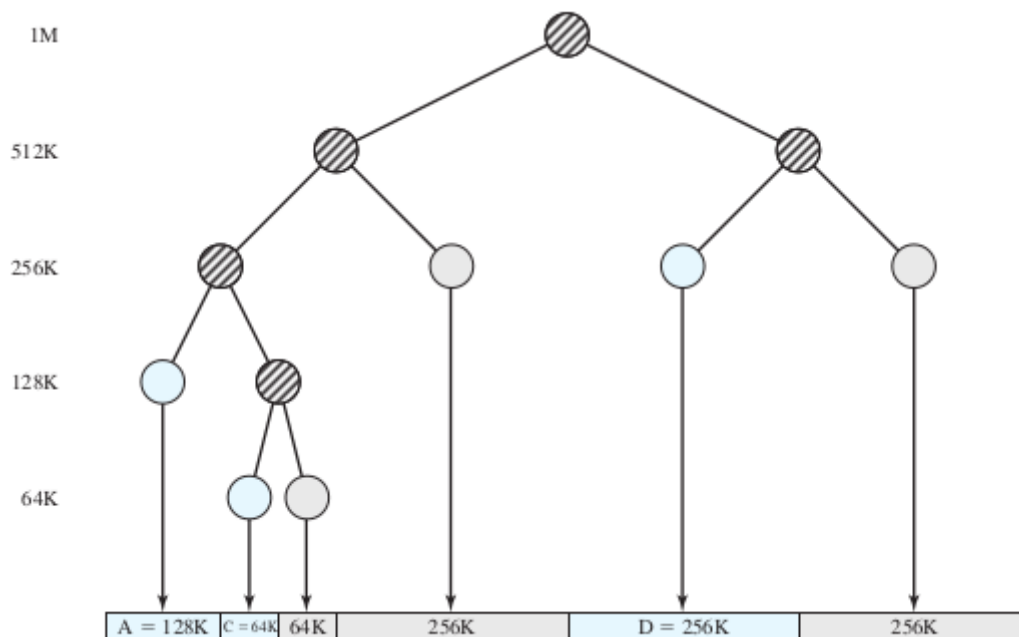
```
void get_hole(int i)
{
    if (i == (U + 1)) <failure>;
    if (<i_list empty>) {
        get_hole(i + 1);
        <split hole into buddies>;
        <put buddies on i_list>;
    }
    <take first hole on i_list>;
}
```

Below figure gives an example using a 1-Mbyte initial block. The first request, A, is for 100 Kbytes, for which a 128K block is needed. The initial block is divided into two 512K buddies. The first of these is divided into two 256K buddies, and the first of these is divided into two 128K buddies, one of which is allocated to A. The next request, B, requires a 256K block. Such a block is already available and is allocated. The process continues with splitting and coalescing occurring as needed. Note that when E is released, two 128K buddies are coalesced into a 256K block, which is immediately coalesced with its buddy.



Below  
with  
Figure  
e  
shows  
a  
binary  
tree  
representat  
ion of  
the  
buddy  
system  
allocat  
ion

immediately after the Release B request. The leaf nodes represent the current partitioning of the memory. If two buddies are leaf nodes, then at least one must be allocated; otherwise they would be coalesced into a larger block. The buddy system is a reasonable compromise to overcome the disadvantages of both the fixed and variable partitioning schemes, but in contemporary operating systems, virtual memory based on paging and segmentation is superior. However, the buddy system has found application in parallel systems as an efficient means of allocation and release for parallel programs (e.g., see [JOHN92]).



## Relocation

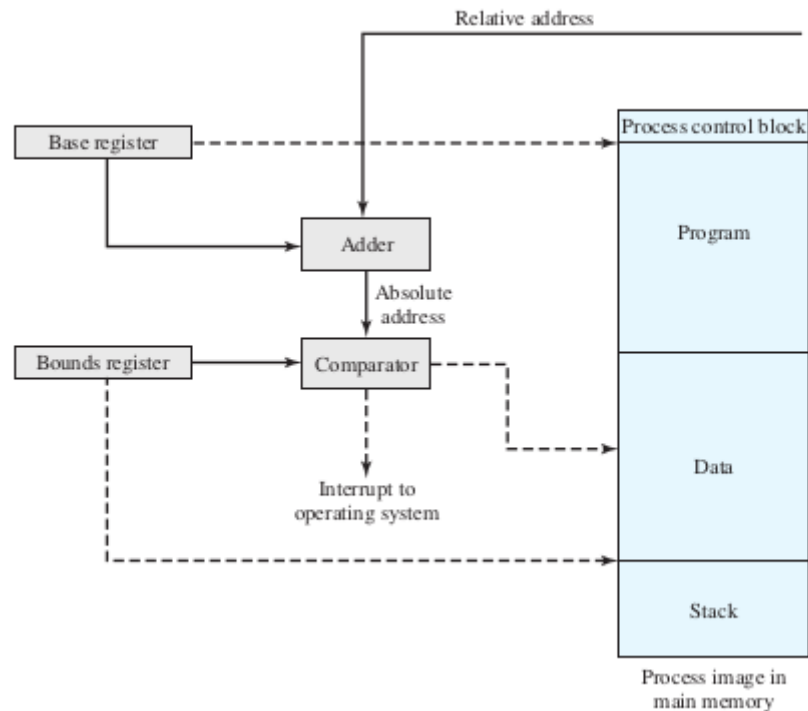
Before we consider ways of dealing with the shortcomings of partitioning, we must clear up one loose end, which relates to the placement of processes in memory. When the fixed partition scheme of Figure is used, we can expect that a process will always be assigned to the same partition. That is, whichever partition is selected when a new process is loaded will

always be used to swap that process back into memory after it has been swapped out. In that case, a simple relocating loader, , can be used: When the process is first loaded, all relative memory references in the code are replaced by absolute main memory addresses, determined by the base address of the loaded process. In the case of equal-size partitions Figure, and in the case of a single process queue for unequal-size partitions Figure, a process may occupy different partitions during the course of its life.

When a process image is first created, it is loaded into some partition in main memory. Later, the process may be swapped out; when it is subsequently swapped back in, it may be assigned to a different partition than the last time. The same is true for dynamic partitioning. Observe in Figures and h that process 2 occupies two different regions of memory on the two occasions when it is brought in. Furthermore, when compaction is used, processes are shifted while they are in main memory. Thus, the locations (of instructions and data) referenced by a process are not fixed. They will change each time a process is swapped in or shifted. To solve this problem, a distinction is made among several types of addresses. A logical address is a reference to a memory location independent of the current assignment of data to memory; a translation must be made to a physical address before the memory access can be achieved.

A relative address is a particular example of logical address, in which the address is expressed as a location relative to some known point, usually a value in a processor register. A physical address, or absolute address, is an actual location in main memory. Programs that employ relative addresses in memory are loaded using dynamic run-time loading. Typically, all of the memory references in the loaded process are relative to the origin of the program. Thus a hardware mechanism is needed for translating relative addresses to physical main memory addresses at the time of execution of the instruction that contains the reference. Below Figure shows the way in which this address translation is typically accomplished. When a process is assigned to the Running state, a special processor register, sometimes called the base register, is loaded with the starting address in main memory of the program. There is also a “bounds” register that indicates the ending location of the program; these values must be set when the program is loaded into memory or when the process image is swapped in. During the course of execution of the process, relative addresses are encountered. These include the contents of the instruction register, instruction addresses that occur in branch and call instructions, and data addresses that occur in load and store instructions. Each such relative address goes through two steps of manipulation by the processor.

First, the value in the base register is added to the relative address to produce an absolute address. Second, the resulting address is compared to the value in the bounds register. If the address is within bounds, then the instruction execution may proceed. Otherwise, an interrupt is generated to the operating system, which must respond to the error in some fashion. The scheme of Figure allows programs to be swapped in and out of memory during the course of execution. It also provides a measure of protection: Each process image is isolated by the contents of the base and bounds registers and safe from unwanted accesses by other processes.



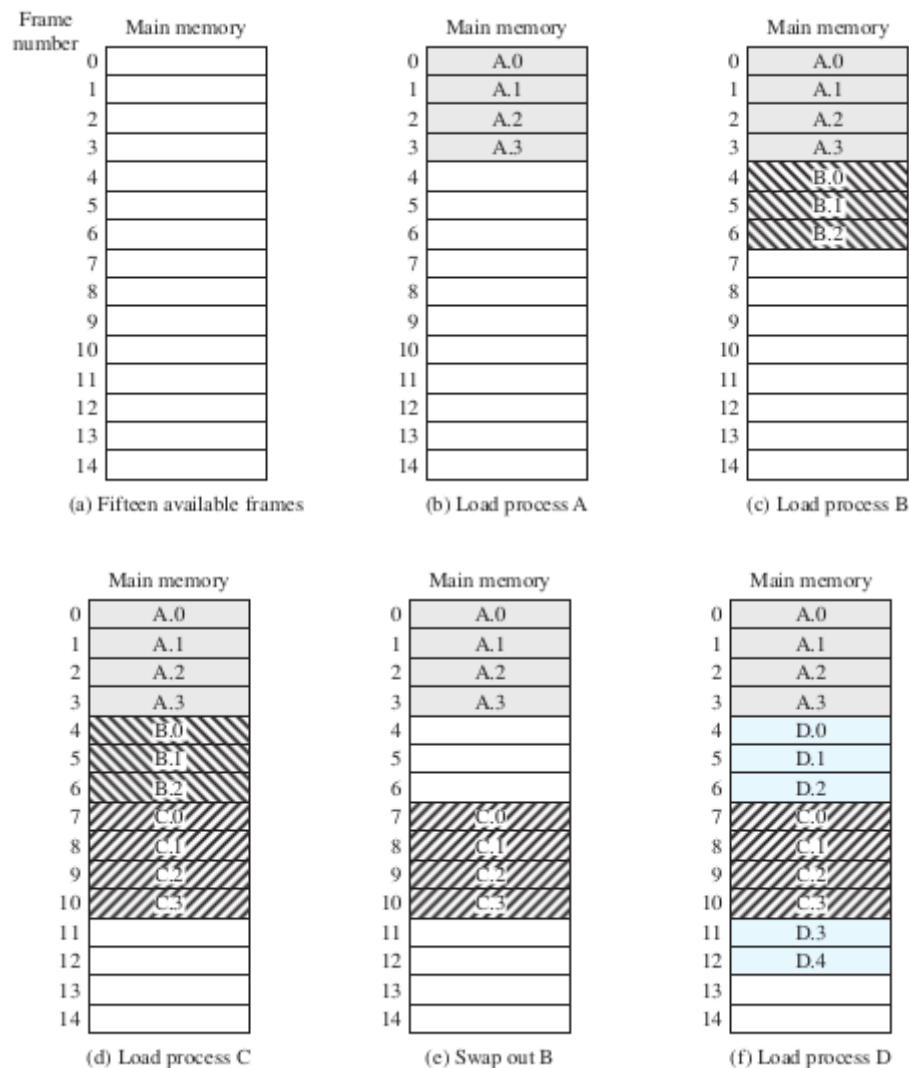
## Paging

Both unequal fixed-size and variable-size partitions are inefficient in the use of memory; the former results in internal fragmentation, the latter in external fragmentation. Suppose, however, that main memory is partitioned into equal fixed-size chunks that are relatively small, and that each process is also divided into small fixed-size chunks of the same size. Then the chunks of a process, known as pages, could be assigned to available chunks of memory, known as frames, or page frames. We show in this section that the wasted space in memory for each process is due to internal fragmentation consisting of only a fraction of the last page of a process. There is no external fragmentation.

Below Figure illustrates the use of pages and frames. At a given point in time, some of the frames in memory are in use and some are free. A list of free frames is maintained by the operating system. Process A, stored on disk, consists of four pages. When it comes time to load this process, the operating system finds four free frames and loads the four pages of process A into the four frames (Figure next) Process B, consisting of three pages, and process C, consisting of four pages, are subsequently loaded.

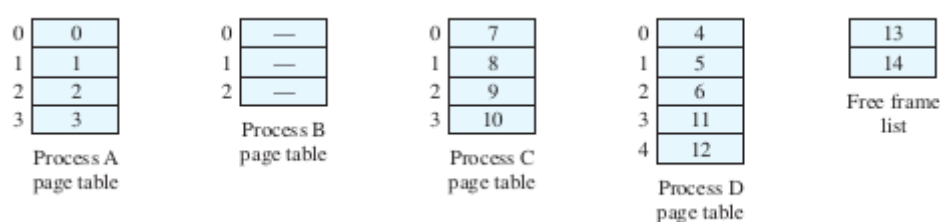
Then process B is suspended and is swapped out of main memory. Later, all of the processes in main memory are blocked, and the operating system needs to bring in a new process, process D, which consists of five pages. Now suppose, as in this example, that there are not sufficient unused contiguous frames to hold the process. Does this prevent the operating system from loading D? The answer is no, because we can once again use the concept of logical address. A simple base address register will no longer suffice. Rather, the operating system maintains a page table for each process. The page table shows the frame location for each page of the process. Within the program, each logical address consists of a page number and an offset within the page. Recall that in the case of simple partition, a logical address is the location of a word relative to the beginning of the program; the processor translates that into a physical address. With paging, the logical-to-physical address translation is still done by processor hardware. Now the processor must know how to access

the page table of the current process. Presented with a logical address (page number, offset), the processor uses the page table to produce a physical address (frame number, offset).

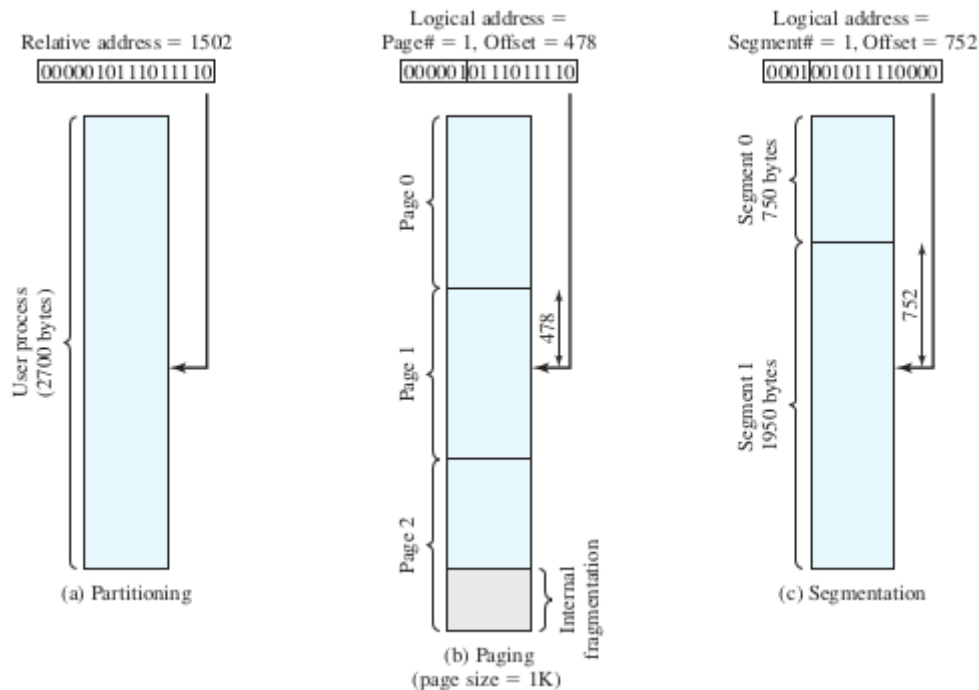


Continuing our example, the five pages of process D are loaded into frames 4, 5, 6, 11, and 12. Figure shows the various page tables at this time. A page table contains one entry for each page of the process, so that the table is easily indexed by the page number (starting at page 0). Each page table entry contains the number of the frame in main memory, if any, that holds the corresponding page. In addition, the operating system maintains a single free-frame list of all frames in main memory that are currently unoccupied and available for pages.

Thus we see that simple paging, as described here, is similar to fixed partitioning. The differences are that, with paging, the partitions are rather small; a program may occupy more than one partition; and these partitions need not be contiguous.



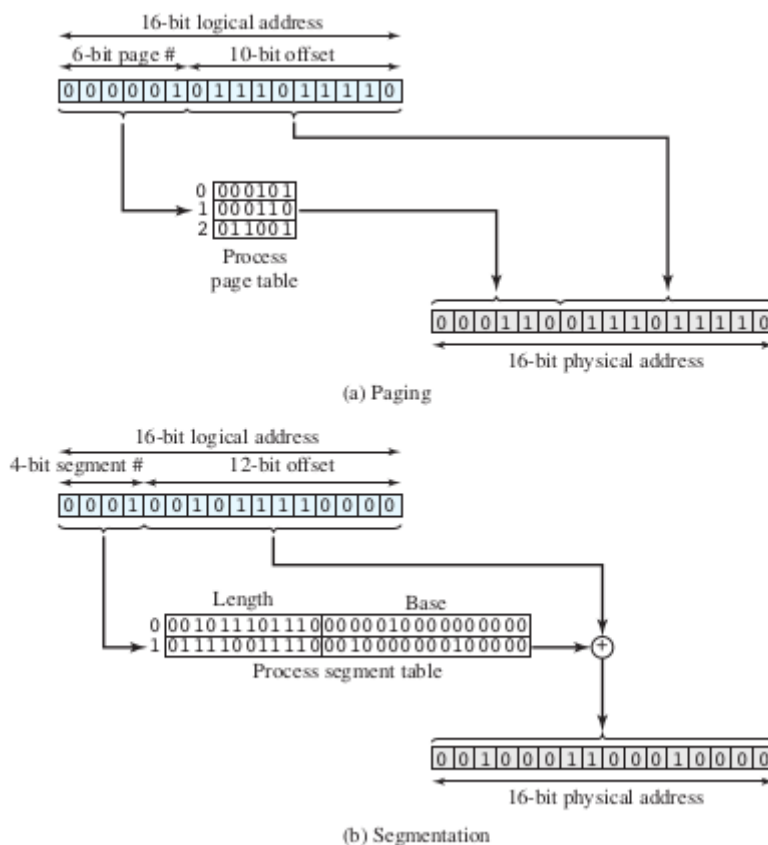
To make this paging scheme convenient, let us dictate that the page size, hence the frame size, must be a power of 2. With the use of a page size that is a power of 2, it is easy to demonstrate that the relative address, which is defined with reference to the origin of the program, and the logical address, expressed as a page number and offset, are the same. An example is shown in Above Figure In this example, 16-bit addresses are used, and the page size is 1K # 1024 bytes. The relative address 1502, in binary form, is 0000010111011110. With a page size of 1K, an offset field of 10 bits is needed, leaving 6 bits for the page number. Thus a program can consist of a maximum of  $2^6$  # 64 pages of 1K bytes each. As Figure shows, relative address 1502 corresponds to an offset of 478 (0111011110) on page 1 (000001), which yields the same 16-bit number, 0000010111011110



The consequences of using a page size that is a power of 2 are twofold. First, the logical addressing scheme is transparent to the programmer, the assembler, and the linker. Each logical address (page number, offset) of a program is identical to its relative address. Second, it is a relatively easy matter to implement a function in hardware to perform dynamic address translation at run time. Consider an address of  $n + m$  bits, where the leftmost  $n$  bits are the page number and the rightmost  $m$  bits are the offset. In our example (Figure),  $n = 6$  and  $m = 10$ . The following steps are needed for address translation:

- Extract the page number as the leftmost  $n$  bits of the logical address.
- Use the page number as an index into the process page table to find the frame number,  $k$ .
- The starting physical address of the frame is  $k \times 2^m$ , and the physical address of the referenced byte is that number plus the offset. This physical address need not be calculated; it is easily constructed by appending the frame number to the offset.

In our example, we have the logical address 0000010111011110, which is page number 1, offset 478. Suppose that this page is residing in main memory frame 6 = binary 000110. Then the physical address is frame number 6, offset 478 = 0001100111011110 (below Figure).



To summarize, with simple paging, main memory is divided into many small equal-size frames. Each process is divided into frame-size pages; smaller processes require fewer pages, larger processes require more. When a process is brought in, all of its pages are loaded into available frames, and a page table is set up. This approach solves many of the problems inherent in partitioning.

## SEGMENTATION

A user program can be subdivided using segmentation, in which the program and its associated data are divided into a number of segments. It is not required that all segments of all programs be of the same length, although there is a maximum segment length. As with paging, a logical address using segmentation consists of two parts, in this case a segment number and an offset.

Because of the use of unequal-size segments, segmentation is similar to dynamic partitioning. In the absence of an overlay scheme or the use of virtual memory, it would be required that all of a program's segments be loaded into memory for execution. The difference, compared to dynamic partitioning, is that with segmentation a program may occupy more than one partition, and these partitions need not be contiguous. Segmentation eliminates internal fragmentation but, like dynamic partitioning, it suffers from external fragmentation. However, because a process is broken up into a number of smaller pieces, the external fragmentation should be less.

Whereas paging is invisible to the programmer, segmentation is usually visible and is provided as a convenience for organizing programs and data. Typically, the programmer or compiler will assign programs and data to different segments. For purposes of modular programming, the program or data may be further broken down into multiple segments. The



principal inconvenience of this service is that the programmer must be aware of the maximum segment size limitation.

Another consequence of unequal-size segments is that there is no simple relationship between logical addresses and physical addresses. Analogous to paging, a simple segmentation scheme would make use of a segment table for each process and a list of free blocks of main memory. Each segment table entry would have to give the starting address in main memory of the corresponding segment. The entry should also provide the length of the segment, to assure that invalid addresses are not used. When a process enters the Running state, the address of its segment table is loaded into a special register used by the memory management hardware. Consider an address of  $n + m$  bits, where the leftmost  $n$  bits are the segment number and the rightmost  $m$  bits

are the offset. In our example (Figure ),  $n = 4$  and  $m = 12$ . Thus the maximum segment size is  $2^{12} = 4096$ . The following steps are needed for address translation:

- Extract the segment number as the leftmost  $n$  bits of the logical address.
- Use the segment number as an index into the process segment table to find the starting physical address of the segment.
- Compare the offset, expressed in the rightmost  $m$  bits, to the length of the segment. If the offset is greater than or equal to the length, the address is invalid.
- The desired physical address is the sum of the starting physical address of the segment plus the offset.

In our example, we have the logical address 0001001011110000, which is segment number 1, offset 752. Suppose that this segment is residing in main memory starting at physical address 0010000000100000. Then the physical address is  $0010000000100000 + 001011110000 = 0010001100010000$  (Figure).

To summarize, with simple segmentation, a process is divided into a number of segments that need not be of equal size. When a process is brought in, all of its segments are loaded into available regions of memory, and a segment table is set up.

### **Combined Systems:**

#### **Segmented Paging**

Pure segmentation is not very popular and not being used in many of the operating systems. However, Segmentation can be combined with Paging to get the best features out of both the techniques.

In Segmented Paging, the main memory is divided into variable size segments which are further divided into fixed size pages.

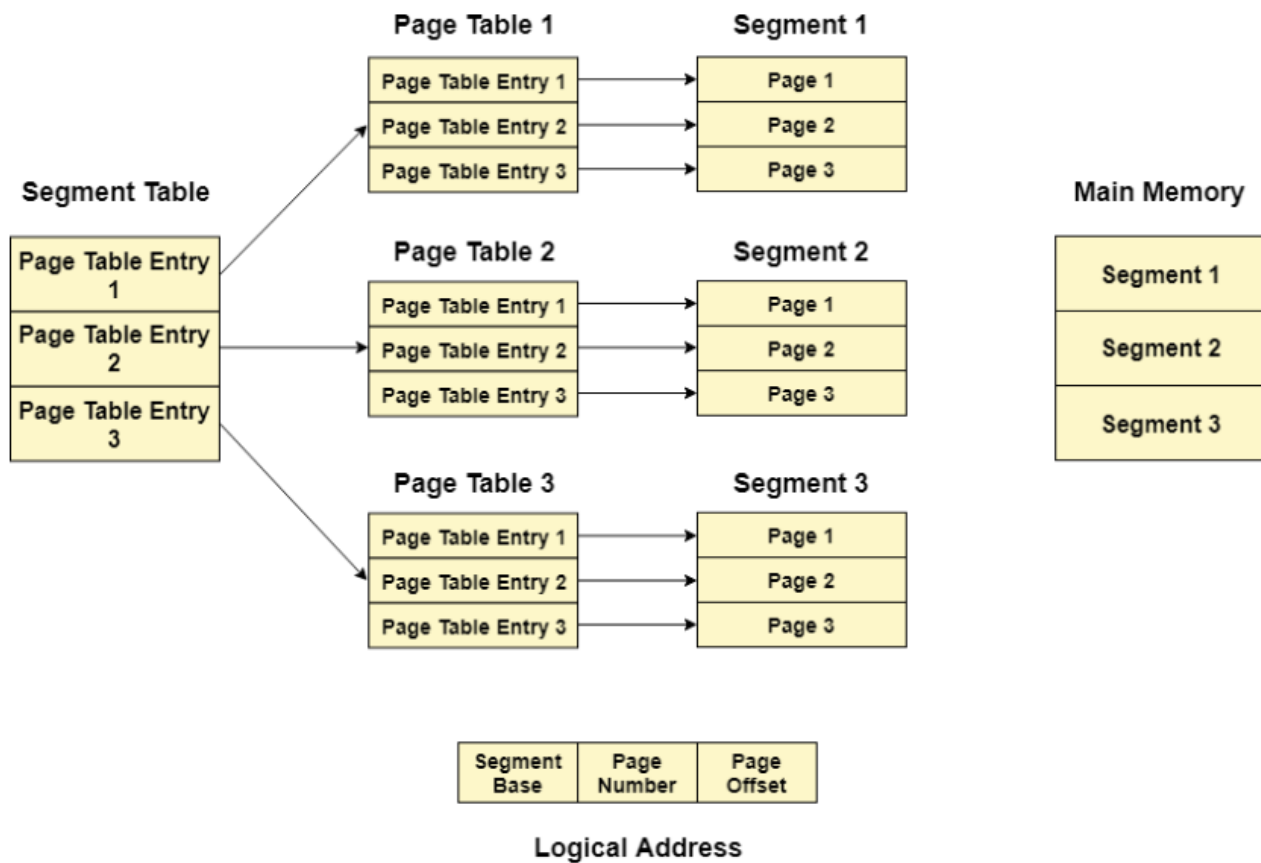
1. Pages are smaller than segments.
2. Each Segment has a page table which means every program has multiple page tables.
3. The logical address is represented as Segment Number (base address), Page number and page offset.

**Segment Number** → It points to the appropriate Segment Number.

**Page Number** → It Points to the exact page within the segment

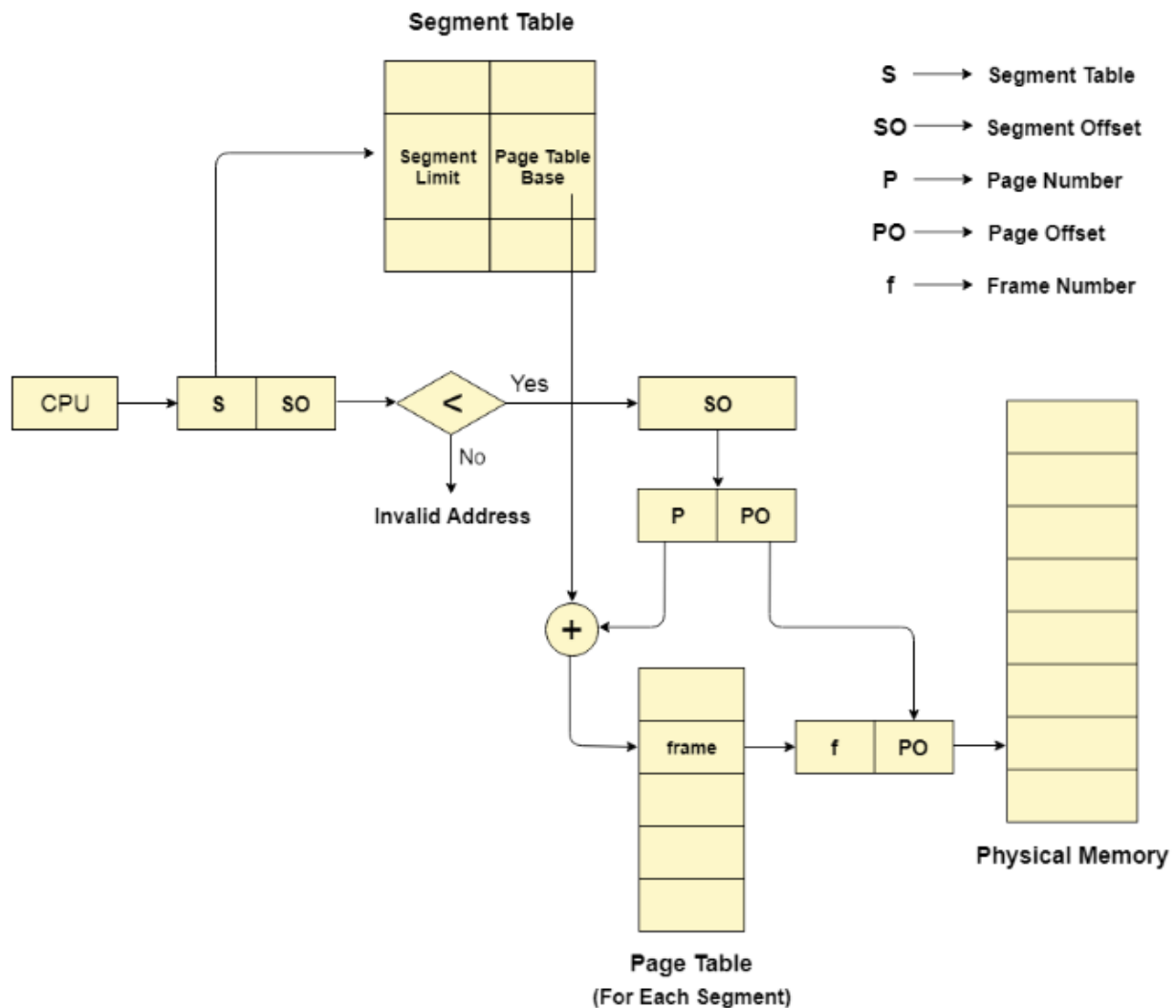
**Page Offset** → Used as an offset within the page frame

Each Page table contains the various information about every page of the segment. The Segment Table contains the information about every segment. Each segment table entry points to a page table entry and every page table entry is mapped to one of the page within a segment.



### Translation of logical address to physical address

The CPU generates a logical address which is divided into two parts: Segment Number and Segment Offset. The Segment Offset must be less than the segment limit. Offset is further divided into Page number and Page Offset. To map the exact page number in the page table, the page number is added into the page table base.



The actual frame number with the page offset is mapped to the main memory to get the desired word in the page of the certain segment of the process.

#### Advantages of Segmented Paging

1. It reduces memory usage.
2. Page table size is limited by the segment size.
3. Segment table has only one entry corresponding to one actual segment.
4. External Fragmentation is not there.
5. It simplifies memory allocation.

#### Disadvantages of Segmented Paging

1. Internal Fragmentation will be there.
2. The complexity level will be much higher as compare to paging.
3. Page Tables need to be contiguously stored in the memory.

**Virtual memory management system:**

**Virtual Memory** is a storage mechanism which offers user an illusion of having a very big main memory. It is done by treating a part of secondary memory as the main memory. In Virtual memory, the user can store processes with a bigger size than the available main memory.

Therefore, instead of loading one long process in the main memory, the OS loads the various parts of more than one process in the main memory. Virtual memory is mostly implemented with demand paging and demand segmentation.

**Why Need Virtual Memory?**

- Whenever your computer doesn't have space in the physical memory it writes what it needs to remember to the hard disk in a swap file as virtual memory.
- If a computer running Windows needs more memory/RAM, then installed in the system, it uses a small portion of the hard drive for this purpose.

**How Virtual Memory Works?**

In the modern world, virtual memory has become quite common these days. It is used whenever some pages require to be loaded in the main memory for the execution, and the memory is not available for those many pages.

So, in that case, instead of preventing pages from entering in the main memory, the OS searches for the RAM space that are minimum used in the recent times or that are not referenced into the secondary memory to make the space for the new pages in the main memory.

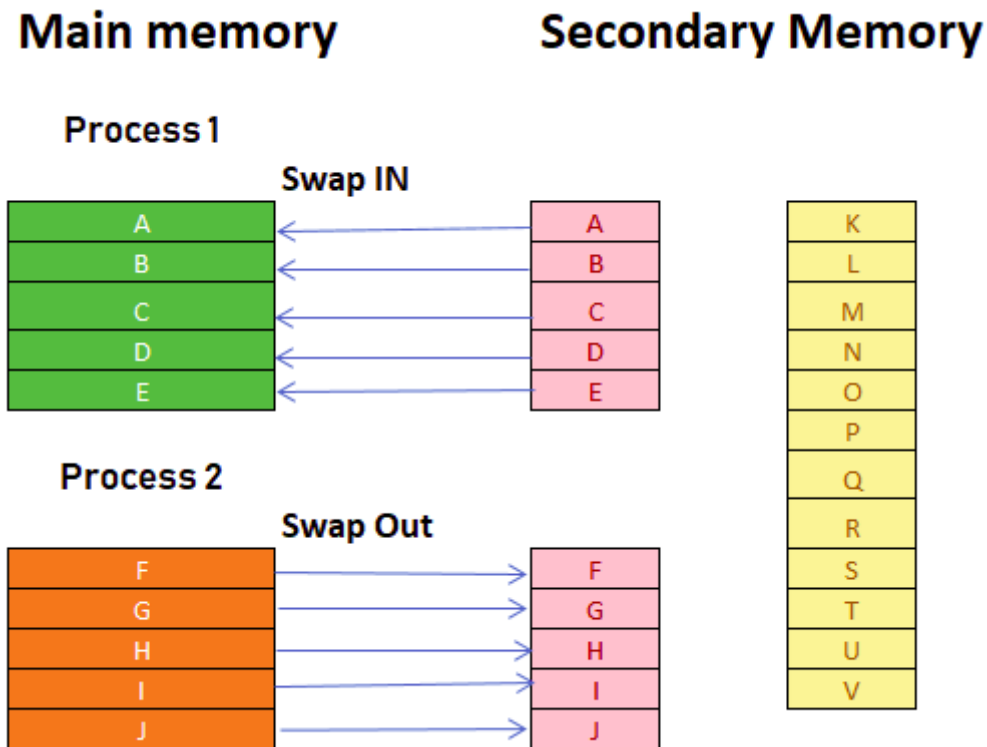
Let's understand virtual memory management with the help of one example.

**For example:**

Let's assume that an OS requires 300 MB of memory to store all the running programs. However, there's currently only 50 MB of available physical memory stored on the RAM.

- The OS will then set up 250 MB of virtual memory and use a program called the Virtual Memory Manager(VMM) to manage that 250 MB.
- So, in this case, the VMM will create a file on the hard disk that is 250 MB in size to store extra memory that is required.
- The OS will now proceed to address memory as it considers 300 MB of real memory stored in the RAM, even if only 50 MB space is available.
- It is the job of the VMM to manage 300 MB memory even if just 50 MB of real memory space is available.

What is Demand Paging?



A demand paging mechanism is very much similar to a paging system with swapping where processes stored in the secondary memory and pages are loaded only on demand, not in advance.

So, when a context switch occurs, the OS never copy any of the old program's pages from the disk or any of the new program's pages into the main memory. Instead, it will start executing the new program after loading the first page and fetches the program's pages, which are referenced.

During the program execution, if the program references a page that may not be available in the main memory because it was swapped, then the processor considers it as an invalid memory reference. That's because the page fault and transfers send control back from the program to the OS, which demands to store page back into the memory.

### Types of Page Replacement Methods

Here, are some important Page replacement methods

- FIFO
- Optimal Algorithm
- LRU Page Replacement

## FIFO Page Replacement

FIFO (First-in-first-out) is a simple implementation method. In this method, memory selects the page for a replacement that has been in the virtual address of the memory for the longest time.

On a page fault, the frame that has been in memory the longest is replaced.

Frame	0	1	2	3	0	1	4	0	1	2	3	4
0	<u>0</u>	0	0	<u>3</u>	<u>3</u>	<u>3</u>	<u>4</u>	4	4	4	4	4
1		<u>1</u>	1	1	<u>0</u>	0	<u>0</u>	0	0	<u>2</u>	2	2
2			<u>2</u>	2	2	<u>1</u>	<u>1</u>	1	1	1	<u>3</u>	3
Frame	0	1	2	3	0	1	4	0	1	2	3	4
0	<u>0</u>	0	0	0	0	0	<u>4</u>	4	4	4	<u>3</u>	3
1		<u>1</u>	1	1	1	1	<u>1</u>	<u>0</u>	0	0	0	<u>4</u>
2			<u>2</u>	2	2	2	<u>2</u>	2	<u>1</u>	1	1	1
3				<u>3</u>	3	3	<u>3</u>	3	3	<u>2</u>	2	2

FIFO is not a stack algorithm. In certain cases, the number of page faults can actually increase when more frames are allocated to the process. In the example below, there are 9 page faults for 3 frames and 10 page faults for 4 frames.

Features:

- Whenever a new page loaded, the page recently comes in the memory is removed. So, it is easy to decide which page requires to be removed as its identification number is always at the FIFO stack.
- The oldest page in the main memory is one that should be selected for replacement first.

## Optimal Algorithm

The optimal page replacement method selects that page for a replacement for which the time to the next reference is the longest.

The Belady's optimal algorithm cheats. It looks forward in time to see which frame to replace on a page fault. Thus it is not a real replacement algorithm. It gives us a frame of reference for a given static frame access sequence.

Page reference stream:

1 2 3 2 1 5 2 1 6 2 5 6 3 1 3 6 1 2 4 3

---

1	1	1	1	1	1	1	1	6	6	6	6	6	6	6	6	6	2	2	2
	2	2	2	2	2	2	2	2	2	2	2	2	1	1	1	1	1	4	4
		3	3	3	5	5	5	5	5	5	5	3	3	3	3	3	3	3	3
*	*	*			*			*				*	*				*	*	

Optimal

Total 9 page faults

## Features:

- Optimal algorithm results in the fewest number of page faults. This algorithm is difficult to implement.
- An optimal page-replacement algorithm method has the lowest page-fault rate of all algorithms. This algorithm exists and which should be called MIN or OPT
- Replace the page which unlikely to use for a longer period of time. It only uses the time when a page needs to be used.

## LRU Page Replacement

The full form of LRU is the Least Recently Used page. This method helps OS to find page usage over a short period of time. This algorithm should be implemented by associating a counter with an even- page.

## How does it work?

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

On a page fault, the frame that was least recently used in replace

Page reference stream:

1 2 3 2 1 5 2 1 6 2 5 6 3 1 3 6 1 2 4 3

---

1 1 1 1 3 2 1 5 2 1 6 2 5 6 6 1 3 6 1 2

2 2 3 2 1 5 2 1 6 2 5 6 3 1 3 6 1 2 4

3 2 1 5 2 1 6 2 5 6 3 1 3 6 1 2 4 3

\* \* \* \* \* \* \* \* \*

LRU

Total 11 page faults

On a page fault, the frame that was least recently used in replaced.

Page reference stream:

1 2 3 2 1 5 2 1 6 2 5 6 3 1 3 6 1 2 4 3

---

1 1 1 1 3 2 1 5 2 1 6 2 5 6 6 1 3 6 1 2

2 2 3 2 1 5 2 1 6 2 5 6 3 1 3 6 1 2 4

3 2 1 5 2 1 6 2 5 6 3 1 3 6 1 2 4 3

\* \* \* \* \* \* \* \* \*

LRU

Total 11 page faults



## LRU Approximation

Pages with a current copy on disk are first choice for pages to be removed when more memory is needed. To facilitate Page Replacement Algorithms, a table of valid or invalid bits (also called *dirty bits*) is maintained.

Page #	valid-invalid bit
	1
	1
	1
	1
	0
⋮	
	0
	0

page table

With each page table entry a valid-invalid bit is associated:

- 1 (in-memory)
- 0 (not-in-memory)

Initially valid-invalid bit is set to 0 on all entries.

In idle times, dirty frame are copied to disk.

Additional **Reference Bit**: whether the frame was recently referenced.

### Features:

- The LRU replacement method has the highest count. This counter is also called aging registers, which specify their age and how much their associated pages should also be referenced.
- The page which hasn't been used for the longest time in the main memory is the one that should be selected for replacement.
- It also keeps a list and replaces pages by looking back into time.

### Fault rate

Fault rate is a frequency with which a designed system or component fails. It is expressed in failures per unit of time. It is denoted by the Greek letter  $\lambda$  (lambda).

## **Advantages of Virtual Memory**

Here, are pros/benefits of using Virtual Memory:

- Virtual memory helps to gain speed when only a particular segment of the program is required for the execution of the program.
- It is very helpful in implementing a multiprogramming environment.
- It allows you to run more applications at once.
- It helps you to fit many large programs into smaller programs.
- Common data or code may be shared between memory.
- Process may become even larger than all of the physical memory.
- Data / code should be read from disk whenever required.
- The code can be placed anywhere in physical memory without requiring relocation.
- More processes should be maintained in the main memory, which increases the effective use of CPU.
- Each page is stored on a disk until it is required after that, it will be removed.
- It allows more applications to be run at the same time.
- There is no specific limit on the degree of multiprogramming.
- Large programs should be written, as virtual address space available is more compared to physical memory.

## **Disadvantages of Virtual Memory**

Here, are drawbacks/cons of using virtual memory:

- Applications may run slower if the system is using virtual memory.
- Likely takes more time to switch between applications.
- Offers lesser hard drive space for your use.
- It reduces system stability.
- It allows larger applications to run in systems that don't offer enough physical RAM alone to run them.
- It doesn't offer the same performance as RAM.
- It negatively affects the overall performance of a system.
- Occupy the storage space, which may be used otherwise for long term data storage.

References:

1. Operating System Concepts, 8<sup>th</sup> Edition, by Galvin et al, 2008, Wiley Publications.
2. Lecture notes and ppt of Ariel J. Frank, Bar-Ilan University.
3. **Operating Systems | Internals and Design Principles | by William Stallings, Ninth Edition | By Pearson Publications**
4. **Web Portal:** <https://www.geeksforgeeks.org>

# Unit 8

## Protection and Security

In this age of universal electronic connectivity, viruses and hackers, electronic eavesdropping, and electronic fraud, security has become a central issue. Two trends have come together to make the topic of this part of vital interest. First, the explosive growth in computer systems and their interconnections via networks has increased the dependence of both organizations and individuals on the information stored and communicated using these systems. This, in turn, has led to a heightened awareness of the need to protect data and resources from disclosure, to guarantee the authenticity of data and messages, and to protect systems from network-based attacks. Second, the disciplines of cryptography and computer security have matured, leading to the development of practical, readily available applications to enforce security.

### COMPUTER SECURITY CONCEPTS

**Computer Security:** The protection afforded to an automated information system in order to attain the applicable objectives of preserving the integrity, availability and confidentiality of information system resources (includes hardware, software, firmware, information/data, and telecommunications)

This definition introduces three key objectives that are at the heart of computer security:

- **Confidentiality:** This term covers two related concepts:
  - **Data confidentiality:** Assures that private or confidential information is not made available or disclosed to unauthorized individuals
  - **Privacy:** Assures that individuals control or influence what information related to them may be collected and stored and by whom and to whom that information may be disclosed
- **Integrity:** This term covers two related concepts:
  - **Data integrity:** Assures that information and programs are changed only in a specified and authorized manner
  - **System integrity:** Assures that a system performs its intended function in an unimpaired manner, free from deliberate or inadvertent unauthorized manipulation of the system
- **Availability:** Assures that systems work promptly and service is not denied to authorized users

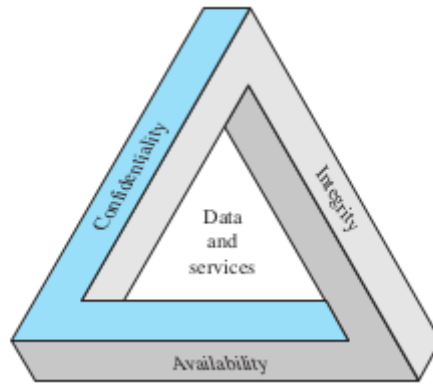


Figure 1 The Security Requirements Triad

These three concepts form what is often referred to as the CIA triad (Figure 1). The three concepts embody the fundamental security objectives for both data and for information and computing services. For example, the NIST standard FIPS 199 (Standards for Security Categorization of Federal Information and Information Systems) lists confidentiality, integrity, and availability as the three security objectives for information and for information systems. FIPS PUB 199 provides a useful characterization of these three objectives in terms of requirements and the definition of a loss of security in each category:

- Confidentiality: Preserving authorized restrictions on information access and disclosure, including means for protecting personal privacy and proprietary information. A loss of confidentiality is the unauthorized disclosure of information
- Integrity: Guarding against improper information modification or destruction, including ensuring information non-repudiation and authenticity. A loss of integrity is the unauthorized modification or destruction of information.
- Availability: Ensuring timely and reliable access to and use of information. A loss of availability is the disruption of access to or use of information or an information system.

Although the use of the CIA triad to define security objectives is well established, some in the security field feel that additional concepts are needed to present a complete picture. Two of the most commonly mentioned are as follows:

- Authenticity: The property of being genuine and being able to be verified and trusted; confidence in the validity of a transmission, a message, or message originator. This means verifying that users are who they say they are and that each input arriving at the system came from a trusted source.
- Accountability: The security goal that generates the requirement for actions of an entity to be traced uniquely to that entity. This supports nonrepudiation, deterrence, fault isolation, intrusion detection and prevention, and after-action recovery and legal action. Because truly secure systems aren't yet an achievable goal, we must be able to trace a security breach to a responsible party. Systems must keep records of their

activities to permit later forensic analysis to trace security breaches or to aid in transaction disputes.

## **THREATS, ATTACKS, AND ASSETS**

### **Threats and Attacks**

Based on RFC 2828, describes four kinds of threat consequences and lists the kinds of attacks that result in each consequence. Unauthorized disclosure is a threat to confidentiality. The following types of attacks can result in this threat consequence:

- **Exposure:** This can be deliberate, as when an insider intentionally releases sensitive information, such as credit card numbers, to an outsider. It can also be the result of a human, hardware, or software error, which results in an entity gaining unauthorized knowledge of sensitive data. There have been numerous instances of this, such as universities accidentally posting student confidential information on the Web.
- **Interception:** Interception is a common attack in the context of communications. On a shared local area network (LAN), such as a wireless LAN or a broadcast Ethernet, any device attached to the LAN can receive a copy of packets intended for another device. On the Internet, a determined hacker can gain access to e-mail traffic and other data transfers. All of these situations create the potential for unauthorized access to data.
- **Inference:** An example of inference is known as traffic analysis, in which an adversary is able to gain information from observing the pattern of traffic on a network, such as the amount of traffic between particular pairs of hosts on the network. Another example is the inference of detailed information from a database by a user who has only limited access; this is accomplished by repeated queries whose combined results enable inference.
- **Intrusion:** An example of intrusion is an adversary gaining unauthorized access to sensitive data by overcoming the system's access control protections

Deception is a threat to either system integrity or data integrity. The following types of attacks can result in this threat consequence:

- **Masquerade:** One example of masquerade is an attempt by an unauthorized user to gain access to a system by posing as an authorized user; this could happen if the unauthorized user has learned another user's logon ID and password. Another example is malicious logic, such as a Trojan horse, that appears to perform a useful or desirable function but actually gains unauthorized access to system resources or tricks a user into executing other malicious logic.

## Threats and Assets

The assets of a computer system can be categorized as hardware, software, data, and communication lines and networks. In this subsection, we briefly describe these four categories and relate these to the concepts of integrity, confidentiality, and availability introduced in above Section

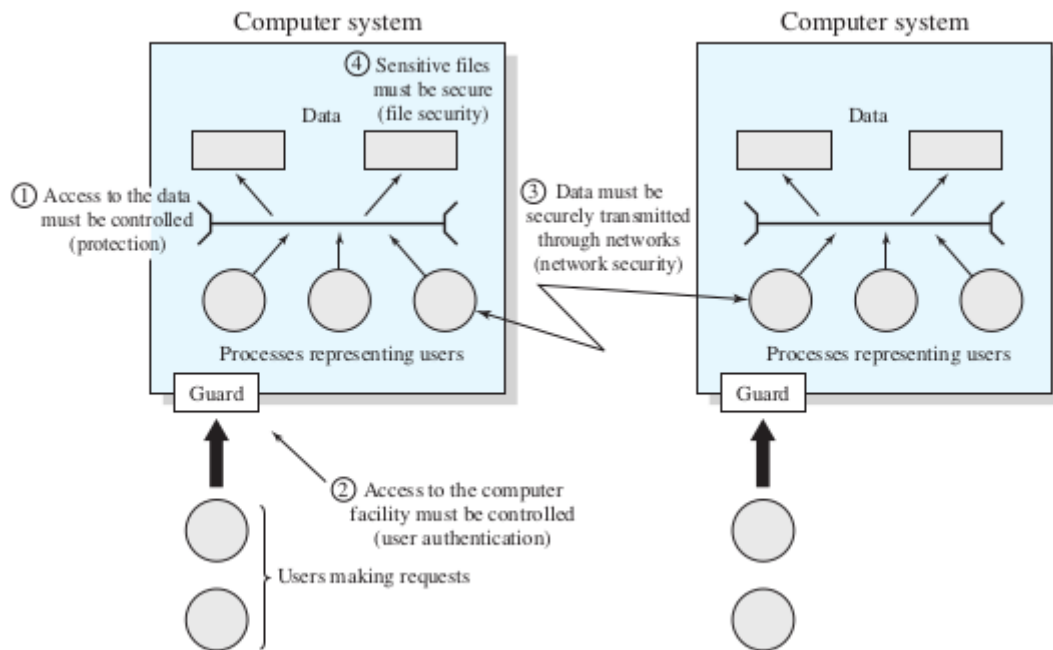


Figure 2 Scope of System Security

Table 1 Computer and Network Assets, with Examples of Threats

	Availability	Confidentiality	Integrity
Hardware	Equipment is stolen or disabled, thus denying service.		
Software	Programs are deleted, denying access to users.	An unauthorized copy of software is made.	A working program is modified, either to cause it to fail during execution or to cause it to do some unintended task.
Data	Files are deleted, denying access to users.	An unauthorized read of data is performed. An analysis of statistical data reveals underlying data.	Existing files are modified or new files are fabricated.
Communication Lines	Messages are destroyed or deleted. Communication lines or networks are rendered unavailable	Messages are read. The traffic pattern of messages is observed.	Messages are modified, delayed, reordered, or duplicated. False messages are fabricated.

**Hardware** A major threat to computer system hardware is the threat to availability. Hardware is the most vulnerable to attack and the least susceptible to automated controls. Threats include accidental and deliberate damage to equipment as well as theft. The proliferation of personal computers and workstations and the widespread use of LANs increase the potential for losses in this area. Theft of CD-ROMs and DVDs can lead to loss of confidentiality. Physical and administrative security measures are needed to deal with these threats.

**Software** Software includes the operating system, utilities, and application programs. A key threat to software is an attack on availability. Software, especially application software, is often easy to delete. Software can also be altered or damaged to render it useless. Careful software configuration management, which includes making backups of the most recent version of software, can maintain high availability. A more difficult problem to deal with is software modification that results in a program that still functions but that behaves differently than before, which is a threat to integrity/authenticity. Computer viruses and related attacks



fall into this category. A final problem is protection against software piracy. Although certain countermeasures are available, by and large the problem of unauthorized copying of software has not been solved.

Data Hardware and software security are typically concerns of computing center professionals or individual concerns of personal computer users. A much more widespread problem is data security, which involves files and other forms of data controlled by individuals, groups, and business organizations.

Security concerns with respect to data are broad, encompassing availability, secrecy, and integrity. In the case of availability, the concern is with the destruction of data files, which can occur either accidentally or maliciously.

The obvious concern with secrecy is the unauthorized reading of data files or databases, and this area has been the subject of perhaps more research and effort than any other area of computer security. A less obvious threat to secrecy involves the analysis of data and manifests itself in the use of so-called statistical databases, which provide summary or aggregate information. Presumably, the existence of aggregate information does not threaten the privacy of the individuals involved. However, as the use of statistical databases grows, there is an increasing potential for disclosure of personal information. In essence, characteristics of constituent individuals may be identified through careful analysis. For example, if one table records the aggregate of the incomes of respondents A, B, C, and D and another records the aggregate of the incomes of A, B, C, D, and E, the difference between the two aggregates would be the income of E. This problem is exacerbated by the increasing desire to combine data sets. In many cases, matching several sets of data for consistency at different levels of aggregation requires access to individual units. Thus, the individual units, which are the subject of privacy concerns, are available at various stages in the processing of data sets.

Finally, data integrity is a major concern in most installations. Modifications to data files can have consequences ranging from minor to disastrous.

**Communication Lines and Networks** Network security attacks can be classified as passive attacks and active attacks. A passive attack attempts to learn or make use of information from the system but does not affect system resources. An active attack attempts to alter system resources or affect their operation.

**Passive attacks** are in the nature of eavesdropping on, or monitoring of, transmissions. The goal of the attacker is to obtain information that is being transmitted. Two types of passive attacks are release of message contents and traffic analysis.

The **release of message contents** is easily understood. A telephone conversation, an electronic mail message, and a transferred file may contain sensitive or confidential information. We would like to prevent an opponent from learning the contents of these transmissions.

A second type of passive attack, **traffic analysis**, is subtler. Suppose that we had a way of masking the contents of messages or other information traffic so that opponents, even if they captured the message, could not extract the information from the message. The common technique for masking contents is encryption. If we had encryption protection in place, an opponent might still be able to observe the pattern of these messages. The opponent could determine the location and identity of communicating hosts and could observe the frequency and length of messages being exchanged. This information might be useful in guessing the nature of the communication that was taking place.

Passive attacks are very difficult to detect because they do not involve any alteration of the data. Typically, the message traffic is sent and received in an apparently normal fashion and neither the sender nor receiver is aware that a third party has read the messages or observed the traffic pattern. However, it is feasible to prevent the success of these attacks, usually by means of encryption. Thus, the emphasis in dealing with passive attacks is on prevention rather than detection.

**Active attacks** involve some modification of the data stream or the creation of a false stream and can be subdivided into four categories: replay, masquerade, modification of messages, and denial of service.

**Replay** involves the passive capture of a data unit and its subsequent retransmission to produce an unauthorized effect.

A **masquerade** takes place when one entity pretends to be a different entity. A masquerade attack usually includes one of the other forms of active attack. For example, authentication sequences can be captured and replayed after a valid authentication sequence has taken place, thus enabling an authorized entity with few privileges to obtain extra privileges by impersonating an entity that has those privileges.

**Modification of messages** simply means that some portion of a legitimate message is altered, or that messages are delayed or reordered, to produce an unauthorized effect. For example, a message stating "Allow John Smith to read confidential file accounts" is modified to say "Allow Fred Brown to read confidential file accounts."

The **denial of service** prevents or inhibits the normal use or management of communications facilities. This attack may have a specific target; for example, an entity may suppress all messages directed to a particular destination (e.g., the security audit service). Another form of service denial is the disruption of an entire network, either by disabling the network or by overloading it with messages so as to degrade performance.

Active attacks present the opposite characteristics of passive attacks. Whereas passive attacks are difficult to detect, measures are available to prevent their success. On the other hand, it is quite difficult to prevent active attacks absolutely, because to do so would require physical protection of all communications facilities and paths at all times. Instead, the goal is to detect them and to recover from any disruption or delays caused by them. Because the detection has a deterrent effect, it may also contribute to prevention.

## INTRUDERS

One of the two most publicized threats to security is the intruder (the other is viruses), often referred to as a hacker or cracker. In an important early study of intrusion, Anderson [ANDE80] identified three classes of intruders:

- **Masquerader:** An individual who is not authorized to use the computer and who penetrates a system's access controls to exploit a legitimate user's account
- **Misfeasor:** A legitimate user who accesses data, programs, or resources for which such access is not authorized, or who is authorized for such access but misuses his or her privileges
- **Clandestine user:** An individual who seizes supervisory control of the system and uses this control to evade auditing and access controls or to suppress audit collection

The masquerader is likely to be an outsider; the misfeasance generally is an insider; and the clandestine user can be either an outsider or an insider.

Intruder attacks range from the benign to the serious. At the benign end of the scale, there are many people who simply wish to explore internets and see what is out there. At the serious end are individuals who are attempting to read privileged data, perform unauthorized modifications to data, or disrupt the system.

lists the following examples of intrusion:

- Performing a remote root compromise of an e-mail server
- Defacing a Web server
- Guessing and cracking passwords
- Copying a database containing credit card numbers
- Viewing sensitive data, including payroll records and medical information, without authorization
- Running a packet sniffer on a workstation to capture usernames and passwords Using a permission error on an anonymous FTP server to distribute pirated software and music files
- Dialing into an unsecured modem and gaining internal network access Posing as an executive, calling the help desk, resetting the executive's e-mail password, and learning the new password

- Using an unattended, logged-in workstation without permission

## **Intruder Behaviour Patterns**

The techniques and behavior patterns of intruders are constantly shifting, to exploit newly discovered weaknesses and to evade detection and countermeasures. Even so, intruders typically follow one of a number of recognizable behavior patterns, and these patterns typically differ from those of ordinary users. In the following, we look at three broad examples of intruder behavior patterns to give the reader some feel for the challenge facing the security administrator. Table 2, based on, summarizes the behavior.

**Hackers** Traditionally, those who hack into computers do so for the thrill of it or for status. The hacking community is a strong meritocracy in which status is determined by level of competence. Thus, attackers often look for targets of opportunity and then share the information with others. A typical example is a break-in at a large financial institution reported in [RADC04]. The intruder took advantage of the fact that the corporate network was running unprotected services, some of which were not even needed. In this case, the key to the break-in was the pcAnywhere application. The manufacturer, Symantec, advertises this program as a remote control solution that enables secure connection to remote devices. But the attacker had an easy time gaining access to pcAnywhere; the administrator used the same three-letter username and password for the program. In this case, there was no intrusion detection system on the 700-node corporate network. The intruder was only discovered when a vice president walked into her office and saw the cursor moving files around on her Windows workstation.

Benign intruders might be tolerable, although they do consume resources and may slow performance for legitimate users. However, there is no way in advance to know whether an intruder will be benign or malign. Consequently, even for systems with no particularly sensitive resources, there is a motivation to control this problem.

Intrusion detection systems (IDSs) and intrusion prevention systems (IPSs), of the type, are designed to counter this type of hacker threat. In addition to using such systems, organizations can consider restricting remote logons to specific IP addresses and/or use virtual private network technology.

Table 2. Some Examples of Intruder Patterns of Behaviour

(a) Hacker

1. Select the target using IP lookup tools such as NSLookup, Dig, and others.
2. Map network for accessible services using tools such as NMAP.
3. Identify potentially vulnerable services (in this case, pcAnywhere).
4. Brute force (guess) pcAnywhere password.
5. Install remote administration tool called DameWare.
6. Wait for administrator to log on and capture his password.
7. Use that password to access remainder of network.

(b) Criminal Enterprise

1. Act quickly and precisely to make their activities harder to detect.
2. Exploit perimeter through vulnerable ports.
3. Use Trojan horses (hidden software) to leave back doors for reentry.
4. Use sniffers to capture passwords.
5. Do not stick around until noticed.
6. Make few or no mistakes.

(c) Internal Threat

1. Create network accounts for themselves and their friends.
2. Access accounts and applications they wouldn't normally use for their daily jobs.
3. E-mail former and prospective employers.
4. Conduct furtive instant-messaging chats.
5. Visit Web sites that cater to disgruntled employees, such as f'dcompany.com.
6. Perform large downloads and file copying.
7. Access the network during off hours.

One of the results of the growing awareness of the intruder problem has been the establishment of a number of computer emergency response teams (CERTs). These cooperative ventures collect information about system vulnerabilities and disseminate it to systems managers. Hackers also routinely CERT reports. Thus, it is important for system administrators to quickly insert all software patches to discovered vulnerabilities. Unfortunately, given the complexity of many IT systems and the rate at which patches are released, this is increasingly difficult to achieve without automated updating. Even then, there are problems caused by incompatibilities resulting from the updated software (hence the need for multiple layers of defence in managing security threats to IT systems).

Criminals Organized groups of hackers have become a widespread and common threat to Internet-based systems. These groups can be in the employ of a corporation or government but often are loosely affiliated gangs of hackers. Typically, these gangs are young, often Eastern European, Russian, or southeast Asian hackers who do business on the Web. They meet in underground forums with names like DarkMarket.org and theftservices.com to trade tips and data and coordinate attacks. A common target is a credit card file at an e-commerce server. Attackers attempt to gain root access. The card numbers are used by organized crime gangs to purchase expensive items and are then posted to carder sites, where others can access and use the account numbers; this obscures usage patterns and complicates investigation.

Whereas traditional hackers look for targets of opportunity, criminal hackers usually have specific targets or at least classes of targets in mind. Once a site is penetrated, the attacker acts quickly, scooping up as much valuable information as possible and exiting.

IDSs and IPSs can also be used for these types of attackers but may be less effective because of the quick in-and-out nature of the attack. For e-commerce sites, database encryption should be used for sensitive customer information, especially credit cards. For hosted e-commerce sites (provided by an outsider service), the e-commerce organization should make use of a dedicated server (not used to support multiple customers) and closely monitor the provider's security services.

**Insider Attacks** Insider attacks are among the most difficult to detect and prevent. Employees already have access to and knowledge of the structure and content of corporate databases. Insider attacks can be motivated by revenge or simply a feeling of entitlement. An example of the former is the case of Kenneth Patterson, fired from his position as data communications manager for American Eagle Outfitters. Patterson disabled the company's ability to process credit card purchases during five days of the holiday season of 2002. As for a sense of entitlement, there have always been many employees who felt entitled to take extra office supplies for home use, but this now extends to corporate data. An example is that of a vice president of sales for a stock analysis firm who quit going to a competitor. Before she left, she copied the customer database to take with her. The offender reported feeling no animus toward her former employee; she simply wanted the data because it would be useful to her.

Although IDS and IPS facilities can be useful in countering insider attacks, other more direct approaches are of higher priority. Examples include the following:

- Enforce least privilege, only allowing access to the resources employees need to do their job.
- Set logs to see what users access and what commands they are entering.
- Protect sensitive resources with strong authentication.
- Upon termination, delete employee's computer and network access.
- Upon termination, make a mirror image of employee's hard drive before reissuing it. That evidence might be needed if your company information turns up at a competitor.

## **Intrusion Techniques**

The objective of the intruder is to gain access to a system or to increase the range of privileges accessible on a system. Most initial attacks use system or software vulnerabilities that allow a user to execute code that opens a back door into the system. Intruders can get access to a system by exploiting attacks such as buffer overflows on a program that runs with certain privileges.

Alternatively, the intruder attempts to acquire information that should have been protected. In some cases, this information is in the form of a user password. With knowledge of some other user's password, an intruder can log in to a system and exercise all the privileges accorded to the legitimate user.

## **VIRUSES, WORMS, AND BOTS**

### **Viruses**

A computer virus is a piece of software that can “infect” other programs by modifying them; the modification includes injecting the original program with a routine to make copies of the virus program, which can then go on to infect other programs.

Biological viruses are tiny scraps of genetic code—DNA or RNA—that can take over the machinery of a living cell and trick it into making thousands of flawless replicas of the original virus. Like its biological counterpart, a computer virus carries in its instructional code the recipe for making perfect copies of itself. The typical virus becomes embedded in a program on a computer. Then, whenever the infected computer comes into contact with an uninfected piece of software, a fresh copy of the virus passes into the new program. Thus, the infection can be spread from computer to computer by unsuspecting users who either swap disks or send programs to one another over a network. In a network environment, the ability to access applications and system services on other computers provides a perfect culture for the spread of a virus.

**The Nature of Viruses** A virus can do anything that other programs do. The only difference is that it attaches itself to another program and executes secretly when the host program is run. Once a virus is executing, it can perform any function that is allowed by the privileges of the current user, such as erasing files and programs.

A computer virus has three parts:

- **Infection mechanism:** The means by which a virus spreads, enabling it to replicate. The mechanism is also referred to as the infection vector.
- **Trigger:** The event or condition that determines when the payload is activated or delivered.
- **Payload:** What the virus does, besides spreading. The payload may involve damage or may involve benign but noticeable activity.

During its lifetime, a typical virus goes through the following four phases:

- Dormant phase: The virus is idle. The virus will eventually be activated by some event, such as a date, the presence of another program or file, or the capacity of the disk exceeding some limit. Not all viruses have this stage.
- Propagation phase: The virus places an identical copy of itself into other programs or into certain system areas on the disk. Each infected program will now contain a clone of the virus, which will itself enter a propagation phase.
- Triggering phase: The virus is activated to perform the function for which it was intended. As with the dormant phase, the triggering phase can be caused by a variety of system events, including a count of the number of times that this copy of the virus has made copies of itself.
- Execution phase: The function is performed. The function may be harmless, such as a message on the screen, or damaging, such as the destruction of programs and data files.

Most viruses carry out their work in a manner that is specific to a particular operating system and, in some cases, specific to a particular hardware platform. Thus, they are designed to take advantage of the details and weaknesses of particular systems.

**Virus Structure** A virus can be prepended or postponed to an executable program, or it can be embedded in some other fashion. The key to its operation is that the infected program, when invoked, will first execute the virus code and then execute the original code of the program.

A very general depiction of virus structure is shown in Figure 3. In this case, the virus code, V, is prepended to infected programs, and it is assumed that the entry point to the program, when invoked, is the first line of the program.

The infected program begins with the virus code and works as follows. The first line of code is a jump to the main virus program. The second line is a special marker that is used by the virus to determine whether or not a potential victim program has already been infected with this virus. When the program is invoked, control is immediately transferred to the main virus program. The virus program may first seek out uninfected executable files and infect them. Next, the virus may perform some action, usually detrimental to the system. This action could be performed every time the program is invoked, or it could be a logic bomb that triggers only under certain conditions. Finally, the virus transfers control to the original program. If the infection phase of the program is reasonably rapid, a user is unlikely to notice any difference between the execution of an infected and an uninfected program.

A virus such as the one just described is easily detected because an infected version of a program is longer than the corresponding uninfected one. A way to thwart such a simple means of detecting a virus is to compress the executable file so that both the infected and uninfected versions are of identical length. Figure 4 shows in general terms the logic required. The important lines in this virus are numbered. We assume that program P 1 is infected with the virus CV. When this program is invoked, control passes to its virus, which performs the following steps:



1. For each uninfected file P 2 that is found, the virus first compresses that file to produce P' 2, which is shorter than the original program by the size of the virus.
2. A copy of the virus is prepended to the compressed program.
3. The compressed version of the original infected program, P 9 1 , is uncompressed.
4. The uncompressed original program is executed.

```
program V :=  
  
{goto main;  
 1234567;  
  
subroutine infect-executable :=  
  {loop:  
   file := get-random-executable-file;  
   if (first-line-of-file = 1234567)  
     then goto loop  
     else prepend V to file; }  
  
subroutine do-damage :=  
  {whatever damage is to be done}  
  
subroutine trigger-pulled :=  
  {return true if some condition holds}  
  
main:  main-program :=  
  {infect-executable;  
   if trigger-pulled then do-damage;  
   goto next;}  
  
next:  
  
}
```

Figure 3. A Simple Virus

```

program CV :=

{goto main;
 01234567;

subroutine infect-executable :=
  {loop:
    file := get-random-executable-file;
    if (first-line-of-file = 01234567) then goto loop;
  (1)  compress file;
  (2)  prepend CV to file;
  }

main:  main-program :=
  {if ask-permission then infect-executable;
  (3)  uncompress rest-of-file;
  (4)  run uncompressed file;}
  }

```

Figure 4. Logic for a Compression Virus

In this example, the virus does nothing other than propagate. As previously mentioned, the virus may include a logic bomb.

**Initial Infection** Once a virus has gained entry to a system by infecting a single program, it is in a position to potentially infect some or all other executable files on that system when the infected program executes. Thus, viral infection can be completely prevented by preventing the virus from gaining entry in the first place. Unfortunately, prevention is extraordinarily difficult because a virus can be part of any program outside a system. Thus, unless one is content to take an absolutely bare piece of iron and write all one's own system and application programs, one is vulnerable. Many forms of infection can also be blocked by denying normal users the right to modify programs on the system.

The lack of access controls on early PCs is a key reason why traditional machine code based viruses spread rapidly on these systems. In contrast, while it is easy enough to write a machine code virus for UNIX systems, they were almost never seen in practice because the existence of access controls on these systems prevented effective propagation of the virus. Traditional machine code based viruses are now less prevalent, because modern PC operating systems have more effective access controls. However, virus creators have found other avenues, such as macro and e-mail viruses, as discussed subsequently.

**Viruses Classification** There has been a continuous arms race between virus writers and writers of antivirus software since viruses first appeared. As effective countermeasures are developed for existing types of viruses, newer types are developed. There is no simple or universally agreed upon classification scheme for viruses. In this section, we follow [AYCO06] and classify viruses along two orthogonal axes: the type of target the virus tries to infect and the method the virus uses to conceal itself from detection by users and antivirus software.

A virus **classification by target** includes the following categories:

- **Boot sector infector:** Infects a master boot record or boot record and spreads when a system is booted from the disk containing the virus
- **File infector:** Infects files that the operating system or shell consider to be executable.
- **Macro virus:** Infects files with macro code that is interpreted by an application

A virus classification by concealment strategy includes the following categories:

- **Encrypted virus:** A typical approach is as follows. A portion of the virus creates a random encryption key and encrypts the remainder of the virus. The key is stored with the virus. When an infected program is invoked, the virus uses the stored random key to decrypt the virus. When the virus replicates, a different random key is selected. Because the bulk of the virus is encrypted with a different key for each instance, there is no constant bit pattern to observe.
- **Stealth virus:** A form of virus explicitly designed to hide itself from detection by antivirus software. Thus, the entire virus, not just a payload, is hidden.
- **Polymorphic virus:** A virus that mutates with every infection, making detection by the “signature” of the virus impossible.
- **Metamorphic virus:** As with a polymorphic virus, a metamorphic virus mutates with every infection. The difference is that a metamorphic virus rewrites itself completely at each iteration, increasing the difficulty of detection. Metamorphic viruses may change their behavior as well as their appearance.

One example of a stealth virus was discussed earlier: a virus that uses compression so that the infected program is exactly the same length as an uninfected version. Far more sophisticated techniques are possible. For example, a virus can place intercept logic in disk I/O routines, so that when there is an attempt to read suspected portions of the disk using these routines, the virus will present back the original, uninfected program. Thus, stealth is not a term that applies to a virus as such but, rather, refers to a technique used by a virus to evade detection.

A polymorphic virus creates copies during replication that are functionally equivalent but have distinctly different bit patterns. As with a stealth virus, the purpose is to defeat programs that scan for viruses. In this case, the “signature” of the virus will vary with each copy. To achieve this variation, the virus may randomly insert superfluous instructions or interchange the order of independent instructions. A more effective approach is to use encryption. The strategy of the encryption virus is followed. The portion of the virus that is responsible for

generating keys and performing encryption/decryption is referred to as the mutation engine. The mutation engine itself is altered with each use.

**Virus Kits** Another weapon in the virus writers' armoury is the virus-creation toolkit. Such a toolkit enables a relative novice to quickly create a number of different viruses. Although viruses created with toolkits tend to be less sophisticated than viruses designed from scratch, the sheer number of new viruses that can be generated using a toolkit creates a problem for antivirus schemes.

**Macro Viruses** In the mid-1990s, macro viruses became by far the most prevalent type of virus. Macro viruses are particularly threatening for a number of reasons:

1. A macro virus is platform independent. Many macro viruses infect Microsoft Word documents or other Microsoft Office documents. Any hardware platform and operating system that supports these applications can be infected.
2. Macro viruses infect documents, not executable portions of code. Most of the information introduced onto a computer system is in the form of a document rather than a program.
3. Macro viruses are easily spread. A very common method is by electronic mail.
4. Because macro viruses infect user documents rather than system programs, traditional file system access controls are of limited use in preventing their spread.

Macro viruses take advantage of a feature found in Word and other office applications such as Microsoft Excel—namely, the macro. In essence, a macro is an executable program embedded in a word processing document or other type of file. Typically, users employ macros to automate repetitive tasks and thereby save keystrokes. The macro language is usually some form of the Basic programming language. A user might define a sequence of keystrokes in a macro and set it up so that the macro is invoked when a function key or special short combination of keys is input.

Successive releases of MS Office products provide increased protection against macro viruses. For example, Microsoft offers an optional Macro Virus Protection tool that detects suspicious Word files and alerts the customer to the potential risk of opening a file with macros. Various antivirus product vendors have also developed tools to detect and correct macro viruses. As in other types of viruses, the arms race continues in the field of macro viruses, but they no longer are the predominant virus threat.

## **E-Mail Viruses**

A more recent development in malicious software is the e-mail virus. The first rapidly spreading e-mail viruses, such as Melissa, made use of a Microsoft Word macro embedded in an attachment. If the recipient opens the email attachment, the Word macro is activated. Then

1. The e-mail virus sends itself to everyone on the mailing list in the user's e-mail package
2. The virus does local damage on the user's system.

In 1999, a more powerful version of the e-mail virus appeared. This newer version can be activated merely by opening an e-mail that contains the virus rather than opening an attachment. The virus uses the Visual Basic scripting language supported by the e-mail package.

Thus we see a new generation of malware that arrives via e-mail and uses e-mail software features to replicate itself across the Internet. The virus propagates itself as soon as it is activated (either by opening an e-mail attachment or by opening the e-mail) to all of the e-mail addresses known to the infected host. As a result, whereas viruses used to take months or years to propagate, they now do so in hours. This makes it very difficult for antivirus software to respond before much damage is

done. Ultimately, a greater degree of security must be built into Internet utility and application software on PCs to counter the growing threat.

## **Worms**

A worm is a program that can replicate itself and send copies from computer to computer across network connections. Upon arrival, the worm may be activated to replicate and propagate again. In addition to propagation, the worm usually performs some unwanted function. An e-mail virus has some of the characteristics of a worm because it propagates itself from system to system. However, we can still classify it as a virus because it uses a document modified to contain viral macro content

and requires human action. A worm actively seeks out more machines to infect and each machine that is infected serves as an automated launching pad for attacks on other machines.

Network worm programs use network connections to spread from system to system. Once active within a system, a network worm can behave as a computervirus or bacteria, or it could implant Trojan horse programs or perform any number of disruptive or destructive actions.

To replicate itself, a network worm uses some sort of network vehicle. Examples include the following:

- **Electronic mail facility:** A worm mails a copy of itself to other systems, so that its code is run when the e-mail or an attachment is received or viewed.
- **Remote execution capability:** A worm executes a copy of itself on another system, either using an explicit remote execution facility or by exploiting a program flaw in a network service to subvert its operations (such as buffer overflow)
- **Remote login capability:** A worm logs onto a remote system as a user and then uses commands to copy itself from one system to the other, where it then executes.

The new copy of the worm program is then run on the remote system where, in addition to any functions that it performs at that system, it continues to spread in the same fashion.

A network worm exhibits the same characteristics as a computer virus: a dormant phase, a propagation phase, a triggering phase, and an execution phase. The propagation phase generally performs the following functions:

1. Search for other systems to infect by examining host tables or similar repositories of remote system addresses.
2. Establish a connection with a remote system.
3. Copy itself to the remote system and cause the copy to be run.

The network worm may also attempt to determine whether a system has previously been infected before copying itself to the system. In a multiprogramming system, it may also disguise its presence by naming itself as a system process or using some other name that may not be noticed by a system operator. As with viruses, network worms are difficult to counter.

**Worm Propagation Model** describes a model for worm propagation based on an analysis of recent worm attacks. The speed of propagation and the total number of hosts infected depend on a number of factors, including the mode of propagation, the vulnerability or vulnerabilities exploited, and the degree of similarity to preceding attacks. For the latter factor, an attack that is a variation of a recent previous attack may be countered more effectively than a more novel attack.

Figure 5 shows the dynamics for one typical set of parameters. Propagation proceeds through three phases. In the initial phase, the number of hosts increases exponentially. To see that this is so, consider a simplified case in which a worm is launched from a single host and infects two nearby hosts. Each of these hosts infects two more hosts, and so on. This results in exponential growth. After a time, infecting hosts waste some time attacking already infected hosts, which reduces the rate of infection. During this middle phase, growth is approximately linear, but the rate of infection is rapid. When most vulnerable computers have been infected, the attack enters a slow finish phase as the worm seeks out those remaining hosts that are difficult to identify.

Clearly, the objective in countering a worm is to catch the worm in its slow start phase, at a time when few hosts have been infected.

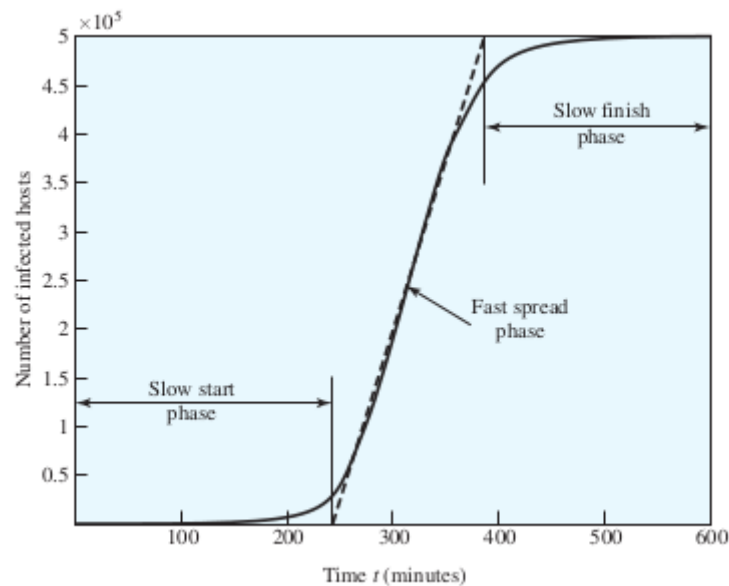


Figure 5. Worm Propagation Model

**State of Worm Technology** The state of the art in worm technology includes the following:

- **Multiplatform:** Newer worms are not limited to Windows machines but can attack a variety of platforms, especially the popular varieties of UNIX.
- **Multiexploit:** New worms penetrate systems in a variety of ways, using exploits against Web servers, browsers, e-mail, file sharing, and other network-based applications.
- **Ultrafast spreading:** One technique to accelerate the spread of a worm is to conduct a prior Internet scan to accumulate Internet addresses of vulnerable machines.
- **Polymorphic:** To evade detection, skip past filters, and foil real-time analysis, worms adopt the virus polymorphic technique. Each copy of the worm has new code generated on the fly using functionally equivalent instructions and encryption techniques.
- **Metamorphic:** In addition to changing their appearance, metamorphic worms have a repertoire of behavior patterns that are unleashed at different stages of propagation.
- **Transport vehicles:** Because worms can rapidly compromise a large number of systems, they are ideal for spreading other distributed attack tools, such as distributed denial of service bots.
- **Zero-day exploit:** To achieve maximum surprise and distribution, a worm should exploit an unknown vulnerability that is only discovered by the general network community when the worm is launched.

## Bots

A bot (robot), also known as a zombie or drone, is a program that secretly takes over another Internet-attached computer and then uses that computer to launch attacks that are difficult to trace to the bot's creator. The bot is typically planted on hundreds or thousands of computers belonging to unsuspecting third parties. The collection of bots often is capable of acting in a coordinated manner; such a collection is referred to as a botnet.

A botnet exhibits three characteristics: the bot functionality, a remote control facility, and a spreading mechanism to propagate the bots and construct the botnet. We examine each of these characteristics in turn.

Uses of Bots lists the following uses of bots:

- **Distributed denial-of-service attacks:** A DDoS attack is an attack on a computer system or network that causes a loss of service to users.
- **Spamming:** With the help of a botnet and thousands of bots, an attacker is able to send massive amounts of bulk e-mail (spam).
- **Sniffing traffic:** Bots can also use a packet sniffer to watch for interesting clear- text data passing by a compromised machine. The sniffers are mostly used to retrieve sensitive information like usernames and passwords.
- **Keylogging:** If the compromised machine uses encrypted communication channels (e.g. HTTPS or POP3S), then just sniffing the network packets on the victim's computer is useless because the appropriate key to decrypt the packets is missing. But by using a keylogger, which captures keystrokes on the infected machine, an attacker can retrieve sensitive information. An implemented filtering mechanism (e.g., "I am only interested in key sequences near the keyword 'paypal.com'") further helps in stealing secret data.
- **Spreading new malware:** Botnets are used to spread new bots. This is very easy since all bots implement mechanisms to download and execute a file via HTTP or FTP. A botnet with 10,000 hosts that acts as the start base for a worm or mail virus allows very fast spreading and thus causes more harm.
- **Installing advertisement add-ons and browser helper objects (BHOs):** Botnets can also be used to gain financial advantages. This works by setting up a fake Web site with some advertisements: The operator of this Web site negotiates a deal with some hosting companies that pay for clicks on ads. With the help of a botnet, these clicks can be "automated" so that instantly a few thousand bots click on the pop-ups. This process can be further enhanced if the bot hijacks the start page of a compromised machine so that the "clicks" are executed each time the victim uses the browser.
- **Attacking IRC chat networks:** Botnets are also used for attacks against Internet relay chat (IRC) networks. Popular among attackers is especially the so called clone attack: In this kind of attack, the controller orders each bot to connect a large number of clones to the victim IRC network. The victim is flooded by service request from thousands of bots or thousands of channel-joins by these cloned bots. In this way, the victim IRC network is brought down, similar to a DDoS attack.



- **Manipulating online polls/games:** Online polls/games are getting more and more attention and it is rather easy to manipulate them with botnets. Since every bot has a distinct IP address, every vote will have the same credibility as a vote cast by a real person. Online games can be manipulated in a similar way.

**Remote Control Facility** The remote control facility is what distinguishes a bot from a worm. A worm propagates itself and activates itself, whereas a bot is controlled from some central facility, at least initially.

A typical means of implementing the remote control facility is on an IRC server. All bots join a specific channel on this server and treat incoming messages as commands. More recent botnets tend to avoid IRC mechanisms and use covert communication channels via protocols such as HTTP. Distributed control mechanisms are also used, to avoid a single point of failure.

Once a communications path is established between a control module and the bots, the control module can activate the bots. In its simplest form, the control module simply issues command to the bot that causes the bot to execute routines that are already implemented in the bot. For greater flexibility, the control module can issue update commands that instruct the bots to download a file from some Internet location and execute it. The bot in this latter case becomes a more general-purpose tool that can be used for multiple attacks.

**Constructing the Attack Network** The first step in a botnet attack is for the attacker to infect a number of machines with bot software that will ultimately be used to carry out the attack. The essential ingredients in this phase of the attack are the following:

1. Software that can carry out the attack. The software must be able to run on a large number of machines, must be able to conceal its existence, must be able to communicate with the attacker or have some sort of time-triggered mechanism, and must be able to launch the intended attack toward the target.
2. A vulnerability in a large number of systems. The attacker must become aware of a vulnerability that many system administrators and individual users have failed to patch and that enables the attacker to install the bot software.
3. A strategy for locating and identifying vulnerable machines, a process known as **scanning** or **fingerprinting**.

In the scanning process, the attacker first seeks out a number of vulnerable machines and infects them. Then, typically, the bot software that is installed in the infected machines repeats the same scanning process, until a large distributed network of infected machines is created. lists the following types of scanning strategies:

- **Random:** Each compromised host probes random addresses in the IP address space, using a different seed. This technique produces a high volume of Internet traffic, which may cause generalized disruption even before the actual attack is launched.

- **Hit list:** The attacker first compiles a long list of potential vulnerable machines. This can be a slow process done over a long period to avoid detection that an attack is underway. Once the list is compiled, the attacker begins infecting machines on the list. Each infected machine is provided with a portion of the list to scan. This strategy results in a very short scanning period, which may make it difficult to detect that infection is taking place.
- **Topological:** This method uses information contained on an infected victim machine to find more hosts to scan.
- **Local subnet:** If a host can be infected behind a firewall, that host then looks for targets in its own local network. The host uses the subnet address structure to find other hosts that would otherwise be protected by the firewall.

## AUTHENTICATION

In most computer security contexts, user authentication is the fundamental building block and the primary line of defense. User authentication is the basis for most types of access control and for user accountability. RFC 2828 defines user authentication as follows:

The process of verifying an identity claimed by or for a system entity. An authentication process consists of two steps:

- Identification step: Presenting an identifier to the security system. (Identifiers should be assigned carefully, because authenticated identities are the basis for other security services, such as access control service.)
- Verification step: Presenting or generating authentication information that corroborates the binding between the entity and the identifier.

For example, user Alice Toklas could have the user identifier ABTOKLAS. This information needs to be stored on any server or computer system that Alice wishes to use and could be known to system administrators and other users. A typical item of authentication information associated with this user ID is a password, which is kept secret (known only to Alice and to the system). If no one is able to obtain or guess Alice's password, then the combination of Alice's user ID and password enables administrators to set up Alice's access permissions and audit her activity. Because Alice's ID is not secret, system users can send her e-mail, but because her password is secret, no one can pretend to be Alice.

In essence, identification is the means by which a user provides a claimed identity to the system; user authentication is the means of establishing the validity of the claim. Note that user authentication is distinct from message authentication. As defined in Chapter 2, message authentication is a procedure that allows communicating parties to verify that the contents of a received message have not been altered and that the source is authentic. This chapter is concerned solely with user authentication.

## Means of Authentication

There are four general means of authenticating a user's identity, which can be used alone or in combination:

- **Something the individual knows:** Examples include a password, a personal identification number (PIN), or answers to a prearranged set of questions
- **Something the individual possesses:** Examples include electronic keycards, smart cards, and physical keys. This type of authenticator is referred to as a token.
- **Something the individual is (static biometrics):** Examples include recognition by fingerprint, retina, and face.
- **Something the individual does (dynamic biometrics):** Examples include recognition by voice pattern, handwriting characteristics, and typing rhythm.

All of these methods, properly implemented and used, can provide secure user authentication. However, each method has problems. An adversary may be able to guess or steal a password. Similarly, an adversary may be able to forge or steal a token. A user may forget a password or lose a token. Further, there is a significant administrative overhead for managing password and token information on systems and securing such information on systems. With respect to biometric authenticators, there are a variety of problems, including dealing with false positives and false negatives, user acceptance, cost, and convenience.

## Password-Based Authentication

A widely used line of defense against intruders is the password system. Virtually all multiuser systems, network-based servers, Web-based e-commerce sites, and other similar services require that a user provide not only a name or identifier (ID) but also a password. The system compares the password to a previously stored password for that user ID, maintained in a system password file. The password serves to authenticate the ID of the individual logging on to the system. In turn, the ID provides security in the following ways:

- The ID determines whether the user is authorized to gain access to a system. In some systems, only those who already have an ID filed on the system are allowed to gain access.
- The ID determines the privileges accorded to the user. A few users may have supervisory or "superuser" status that enables them to read files and perform functions that are especially protected by the operating system. Some systems have guest or anonymous accounts, and users of these accounts have more limited privileges than others.
- The ID is used in what is referred to as discretionary access control. For example, by listing the IDs of the other users, a user may grant permission to them to read files owned by that user.

**The Use of Hashed Passwords** A widely used password security technique is the use of hashed passwords and a salt value. This scheme is found on virtually all UNIX variants as well as on a number of other operating systems. The following procedure is employed (Figure 15.1a). To load a new password into the system, the user selects or is assigned a

password. This password is combined with a fixed-length salt value [MORR79]. In older implementations, this value is related to the time at which the password is assigned to the user. Newer implementations use a pseudo- random or random number. The password and salt serve as inputs to a hashing algorithm to produce a fixed-length hash code. The hash algorithm is designed to be slow to execute to thwart attacks. The hashed password is then stored, together with a plaintext copy of the salt, in the password file for the corresponding user ID. The hashed-password method has been shown to be secure against a variety of cryptanalytic attacks.

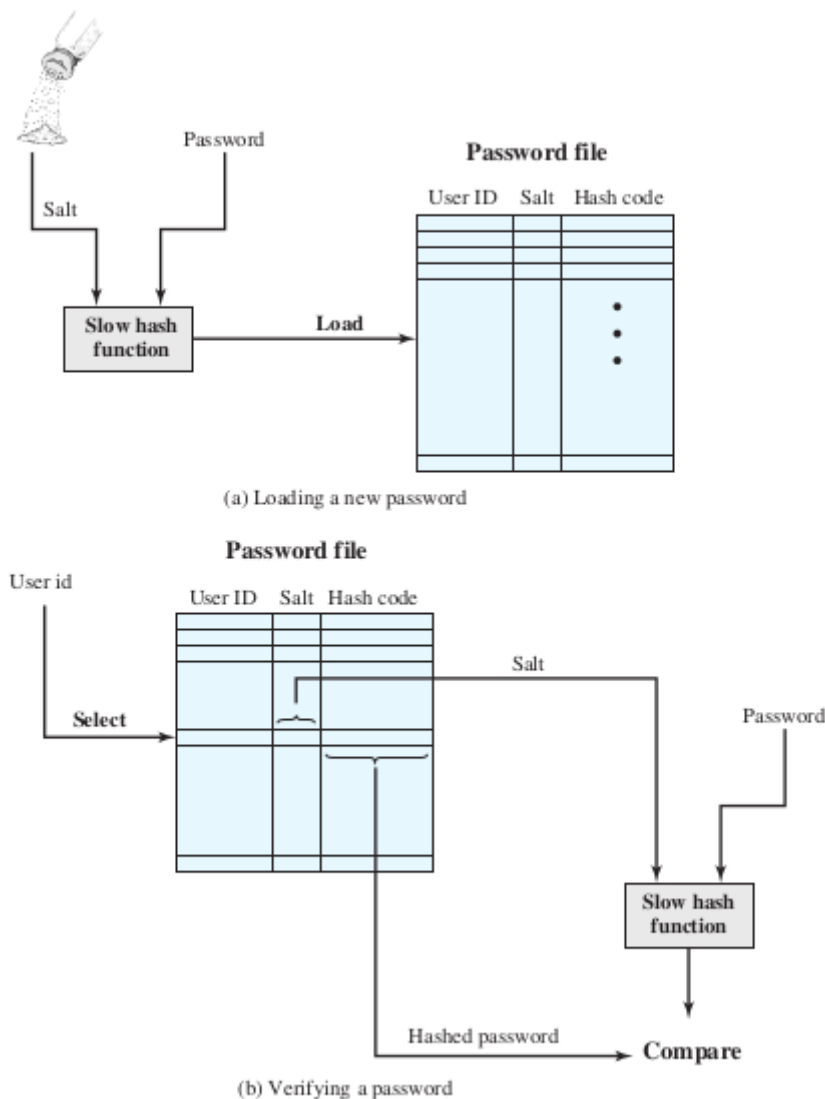


Figure 6. UNIX Password Scheme

When a user attempts to log on to a UNIX system, the user provides an ID and a password (Figure 6b). The operating system uses the ID to index into the password file and retrieve the

plaintext salt and the encrypted password. The salt and user-supplied password are used as input to the encryption routine. If the result matches the stored value, the password is accepted.

The salt serves three purposes:

- It prevents duplicate passwords from being visible in the password file. Even if two users choose the same password, those passwords will be assigned different salt values. Hence, the hashed passwords of the two users will differ.
- It greatly increases the difficulty of offline dictionary attacks. For a salt of length  $b$  bits, the number of possible passwords is increased by a factor of  $2^b$ , increasing the difficulty of guessing a password in a dictionary attack.
- It becomes nearly impossible to find out whether a person with passwords on two or more systems has used the same password on all of them.

To see the second point, consider the way that an offline dictionary attack would work. The attacker obtains a copy of the password file. Suppose first that the salt is not used. The attacker's goal is to guess a single password. To that end, the attacker submits a large number of likely passwords to the hashing function. If any of the guesses matches one of the hashes in the file, then the attacker has found a password that is in the file. But faced with the UNIX scheme, the attacker must take each guess and submit it to the hash function once for each salt value in the dictionary file, multiplying the number of guesses that must be checked.

There are two threats to the UNIX password scheme. First, a user can gain access on a machine using a guest account or by some other means and then run a password guessing program, called a password cracker, on that machine. The attacker should be able to check many thousands of possible passwords with little resource consumption. In addition, if an opponent is able to obtain a copy of the password file, then a cracker program can be run on another machine at leisure. This enables the opponent to run through millions of possible passwords in a reasonable period.

## **UNIX Implementations**

Since the original development of UNIX, most implementations have relied on the following password scheme. Each user selects a password of up to eight printable characters in length. This is converted into a 56-bit value (using 7-bit ASCII) that serves as the key input to an encryption routine. The hash routine, known as `crypt(3)`, is based on DES. A 12-bit salt value is used. The modified DES algorithm is executed with a data input consisting of a 64-bit block of zeros. The output of the algorithm then serves as input for a second encryption. This process is repeated for a total of 25 encryptions. The resulting 64-bit output is then translated into an 11-character sequence. The modification of the DES algorithm converts it into a one-way hash function. The `crypt(3)` routine is designed to discourage guessing attacks. Software

implementations of DES are slow compared to hardware versions, and the use of 25 iterations multiplies the time required by 25.

This particular implementation is now considered woefully inadequate. For example, reports the results of a dictionary attack using a supercomputer. The attack was able to process over 50 million password guesses in about 80 minutes. Further, the results showed that for about \$10,000 anyone should be able to do the same in a few months using one uniprocessor machine. Despite its known weaknesses, this UNIX scheme is still often required for compatibility with existing account management software or in multivendor environments.

There are other, much stronger, hash/salt schemes available for UNIX. The recommended hash function for many UNIX systems, including Linux, Solaris, and FreeBSD, is based on the MD5 secure hash algorithm (which is similar to, but not as secure as SHA-1).<sup>1</sup> The MD5 crypt routine uses a salt of up to 48 bits and effectively has no limitations on password length. It produces a 128-bit hash value. It is also far slower than crypt(3). To achieve the slowdown, MD5 crypt uses an inner loop with 1000 iterations

Probably the most secure version of the UNIX hash/salt scheme was developed for OpenBSD, another widely used open source UNIX. This scheme, reported, uses a hash function based on the Blowfish symmetric block cipher. The hash function, called Bcrypt, is quite slow to execute. Bcrypt allows passwords of up to 55 characters in length and requires a random salt value of 128 bits, to produce a 192-bit hash value. Bcrypt also includes a cost variable; an increase in the cost variable causes a corresponding increase in the time required to perform a Bcrypt hash. The cost assigned to a new password is configurable, so that administrators can assign a higher cost to privileged users.

## **Token-Based Authentication**

Objects that a user possesses for the purpose of user authentication are called tokens. In this subsection, we examine two types of tokens that are widely used; these are cards that have the appearance and size of bank cards.

**Memory Cards** Memory cards can store but not process data. The most common such card is the bank card with a magnetic stripe on the back. A magnetic stripe can store only a simple security code, which can be read (and unfortunately reprogrammed) by an inexpensive card reader. There are also memory cards that include an internal electronic memory.

Memory cards can be used alone for physical access, such as a hotel room. For computer user authentication, such cards are typically used with some form of password or personal identification number (PIN). A typical application is an automatic teller machine (ATM).

The memory card, when combined with a PIN or password, provides significantly greater security than a password alone. An adversary must gain physical possession of the card (or be able to duplicate it) plus must gain knowledge of the PIN. Among the potential drawbacks are the following:

- **Requires special reader:** This increases the cost of using the token and creates the requirement to maintain the security of the reader's hardware and software.
- **Token loss:** A lost token temporarily prevents its owner from gaining system access. Thus there is an administrative cost in replacing the lost token. In addition, if the token is found, stolen, or forged, then an adversary now need only determine the PIN to gain unauthorized access.
- **User dissatisfaction:** Although users may have no difficulty in accepting the use of a memory card for ATM access, its use for computer access may be deemed inconvenient.

**Smart Cards** A wide variety of devices qualify as smart tokens. These can be categorized along three dimensions that are not mutually exclusive:

- **Physical characteristics:** Smart tokens include an embedded microprocessor. A smart token that looks like a bank card is called a smart card. Other smart tokens can look like calculators, keys, or other small portable objects.
- **Interface:** Manual interfaces include a keypad and display for human/token interaction. Smart tokens with an electronic interface communicate with a compatible reader/writer.
- **Authentication protocol:** The purpose of a smart token is to provide a means for user authentication. We can classify the authentication protocols used with smart tokens into three categories:
  - **Static:** With a static protocol, the user authenticates himself or herself to the token and then the token authenticates the user to the computer. The latter half of this protocol is similar to the operation of a memory token.
  - **Dynamic password generator:** In this case, the token generates a unique password periodically (e.g., every minute). This password is then entered into the computer system for authentication, either manually by the user or electronically via the token. The token and the computer system must be initialized and kept synchronized so that the computer knows the password that is current for this token.
  - **Challenge-response:** In this case, the computer system generates a challenge, such as a random string of numbers. The smart token generates a response based on the challenge. For example, public-key cryptography could be used and the token could encrypt the challenge string with the token's private key.

For user authentication to computer, the most important category of smart token is the smart card, which has the appearance of a credit card, has an electronic interface, and may use any of the type of protocols just described. The remainder of this section discusses smart cards.

A smart card contains within it an entire microprocessor, including processor, memory, and I/O ports. Some versions incorporate a special co-processing circuit for cryptographic operation to speed the task of encoding and decoding messages or generating digital signatures to validate the information transferred. In some cards, the I/O ports are directly accessible by a compatible reader by means of exposed electrical contacts. Other cards rely instead on an embedded antenna for wireless communication with the reader.

## **Biometric Authentication**

A biometric authentication system attempts to authenticate an individual based on his or her unique physical characteristics. These include static characteristics, such as fingerprints, hand geometry, facial characteristics, and retinal and iris patterns; and dynamic characteristics, such as voiceprint and signature. In essence, biometrics is based on pattern recognition. Compared to passwords and tokens, biometric authentication is both technically complex and expensive. While it is used in a number of specific applications, biometrics has yet to mature as a standard tool for user authentication to computer systems.

A number of different types of physical characteristics are either in use or under study for user authentication. The most common are the following:

- **Facial characteristics:** Facial characteristics are the most common means of human-to-human identification; thus it is natural to consider them for identification by computer. The most common approach is to define characteristics based on relative location and shape of key facial features, such as eyes, eyebrows, nose, lips, and chin shape. An alternative approach is to use an infrared camera to produce a face thermogram that correlates with the underlying vascular system in the human face.
- **Fingerprints:** Fingerprints have been used as a means of identification for centuries, and the process has been systematized and automated particularly for law enforcement purposes. A fingerprint is the pattern of ridges and furrows on the surface of the fingertip. Fingerprints are believed to be unique across the entire human population. In practice, automated fingerprint recognition and matching system extract a number of features from the fingerprint for storage as a numerical surrogate for the full fingerprint pattern.
- **Hand geometry:** Hand geometry systems identify features of the hand, including shape, and lengths and widths of fingers.
- **Retinal pattern:** The pattern formed by veins beneath the retinal surface is unique and therefore suitable for identification. A retinal biometric system obtains a digital image of the retinal pattern by projecting a low-intensity beam of visual or infrared light into the eye.
- **Iris:** Another unique physical characteristic is the detailed structure of the iris.
- **Signature:** Each individual has a unique style of handwriting, and this is reflected especially in the signature, which is typically a frequently written sequence. However, multiple signature samples from a single individual will not be identical. This



complicates the task of developing a computer representation of the signature that can be matched to future samples.

- **Voice:** Whereas the signature style of an individual reflects not only the unique physical attributes of the writer but also the writing habit that has developed, voice patterns are more closely tied to the physical and anatomical characteristics of the speaker. Nevertheless, there is still a variation from sample to sample over time from the same speaker, complicating the biometric recognition task.

Figure 7. gives a rough indication of the relative cost and accuracy of these biometric measures. The concept of accuracy does not apply to user authentication schemes using smart cards or passwords. For example, if a user enters a password, it either matches exactly the password expected for that user or not. In the case of biometric parameters, the system instead must determine how closely a presented biometric characteristic matches a stored characteristic. Before elaborating on the concept of biometric accuracy, we need to have a general idea of how biometric systems work.

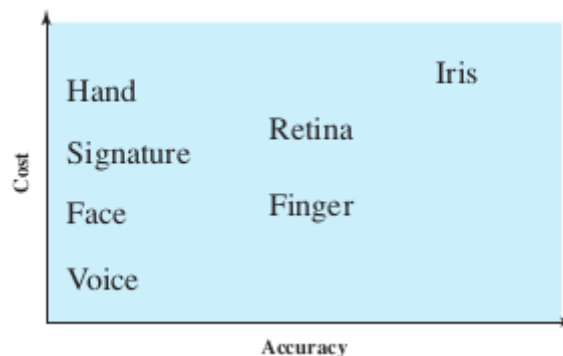


Figure 7. Cost versus Accuracy of Various Biometric Characteristics in User Authentication Schemes

#### References:

1. Operating System Concepts, 8<sup>th</sup> Edition, by Galvin et al, 2008, Wiley Publications.
2. Lecture notes and ppt of Ariel J. Frank, Bar-Ilan University.
3. **Operating Systems | Internals and Design Principles | by William Stallings, Ninth Edition | By Pearson Publications**
4. **Web Portal:** <https://www.geeksforgeeks.org>