

MAR GREGORIOS COLLEGE OF ARTS & SCIENCE

Block No.8, College Road, Mogappair West, Chennai – 37

Affiliated to the University of Madras
Approved by the Government of Tamil Nadu
An ISO 9001:2015 Certified Institution



DEPARTMENT OF COMPUTER APPLICATION

SUBJECT NAME: OPERATING SYSTEMS

SUBJECT CODE: SAZ4C

SEMESTER: IV

PREPARED BY: PROF. T.INDUMATHI

Core Paper - X : OPERATING SYSTEMS

Objective of the course :This course introduces the functions of operating systems.

Unit 1:Introduction: Views –Goals –Types of system – OS Structure –Components – Services - System Structures – Layered Approach -Virtual Machines - System Design and Implementation. Process Management: Process - Process Scheduling – Cooperating Process – Threads - Interprocess Communication. CPU Scheduling : CPU Schedulers – Scheduling criteria – Scheduling Algorithms

Unit-2:– Process Synchronization: Critical-Section problem - Synchronization Hardware – Semaphores – Classic Problems of Synchronization – Critical Region – Monitors. Deadlock : Characterization – Methods for handling Deadlocks – Prevention, Avoidance, and Detection of Deadlock - Recovery from deadlock.

Unit 3: Memory Management : Address Binding – Dynamic Loading and Linking – Overlays – Logical and Physical Address Space - Contiguous Allocation – Internal & External Fragmentation . Non Contiguous Allocation: Paging and Segmentation schemes – Implementation – Hardware Protection – Sharing - Fragmentation.

Unit-4:VirtualMemory :: Demand Paging – Page Replacement - Page Replacement Algorithms – Thrashing. – File System: Concepts – Access methods – Directory Structure – Protection Consistency Semantics – File System Structures – Allocation methods – Free Space Management.

Unit-5 : I/O Systems: Overview - I/O Hardware – Application I/O Interface – Kernel I/O subsystem – Transforming I/O Requests to Hardware Operations – Performance. Secondary Storage Structures : Protection – Goals- Domain Access matrix – The security problem – Authentication – Threats – Threat Monitoring – Encryption..

1. **Recommended Texts:** i) Silberschatz A., Galvin P.B., Gange,,2002 , Operating System Principles ,Sixth Edition, John Wiley & Sons.

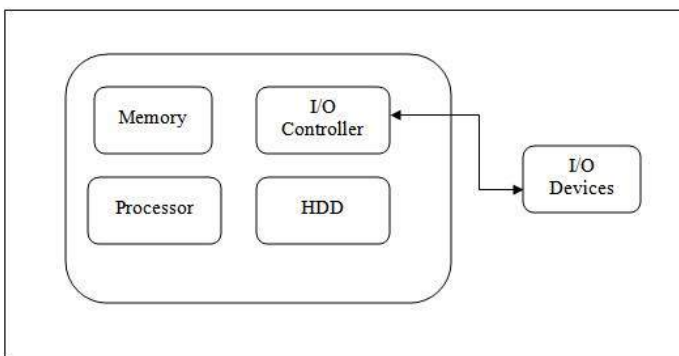
2.**Reference Books:** i) H.M. Deitel ,1990, An Introduction to Operating System,- Second Edition, Addison Wesley.

UNIT I

Operating System:

- An operating system is a program which manages all the computer hardwares.
- It provides the base for application program and acts as an intermediary between a user and the computer hardware.
- The operating system has two objectives such as:
 - Firstly, an operating system controls the computer's hardware.
 - The second objective is to provide an interactive interface to the user and interpret commands so that it can communicate with the hardware.
- The operating system is very important part of almost every computer system.

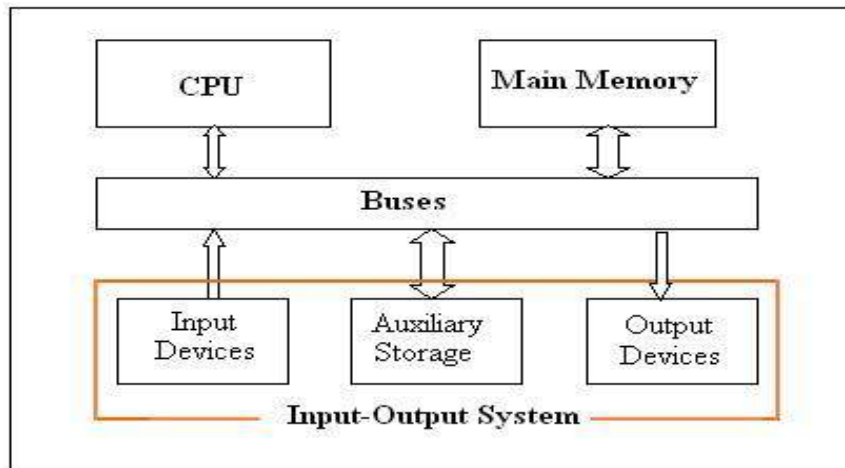
Managing Hardware



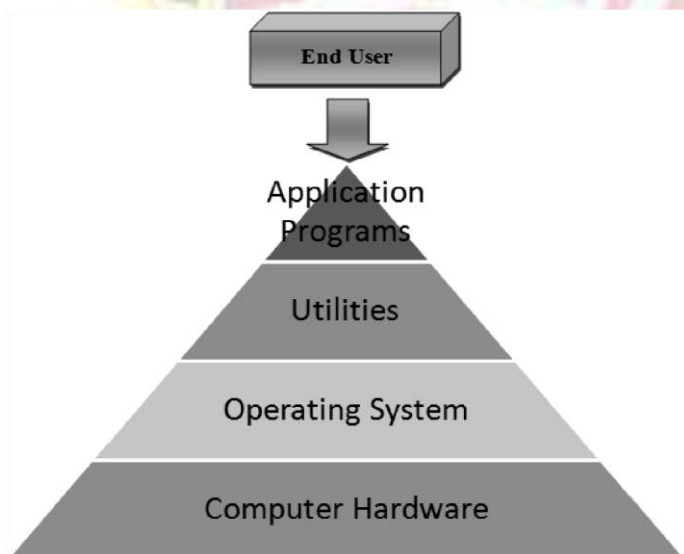
- The prime objective of operating system is to manage & control the various hardware resources of a computer system.
- These hardware resources include processor, memory, and disk space and so on.
- The output result was display in monitor. In addition to communicating with the hardware the operating system provides on error handling procedure and display an error notification.
- If a device not functioning properly, the operating system cannot be communicate with the device.

Providing an Interface

- The operating system organizes application so that users can easily access, use and store them.



- It provides a stable and consistent way for applications to deal with the hardware without the user having known details of the hardware.
- If the program is not functioning properly, the operating system again takes control, stops the application and displays the appropriate error message.
- Computer system components are divided into 5 parts
 - Computer hardware
 - operating system utilities
 - Application programs
 - End user



- The operating system controls and coordinate a user of hardware and various application programs for various users.
- It is a program that directly interacts with the hardware.
- The operating system is the first encoded with the Computer and it remains on the memory all time thereafter.

System goals

- The purpose of an operating system is to be provided an environment in which an user can execute programs.
- Its primary goals are to make the computer system convenience for the user.
- Its secondary goals are to use the computer hardware in efficient manner.

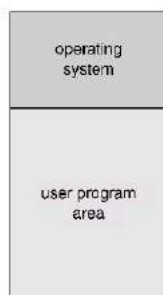
View of operating system

- **User view:** The user view of the computer varies by the interface being used. The examples are -windows XP, vista, windows 7 etc. Most computer user sit in the front of personal computer (pc) in this case the operating system is designed mostly for easy use with some attention paid to resource utilization. Some user sit at a terminal connected to a mainframe/minicomputer. In this case other users are accessing the same computer through the other terminals. There user are share resources and may exchange the information. The operating system in this case is designed to maximize resources utilization to assume that all available CPU time, memory and I/O are used efficiently and no individual user takes more than his/her fair and share. The other users sit at workstations connected to network of other workstations and servers. These users have dedicated resources but they share resources such as networking and servers like file, compute and print server. Here the operating system is designed to compromise between individual usability and resource utilization.
- **System view:** From the computer point of view the operating system is the program which is most intermediate with the hardware. An operating system has resources as hardware and software which may be required to solve a problem like CPU time, memory space, file storage space and I/O devices and so on. That's why the operating system acts as manager of these resources. Another view of the operating system is it is a control program. A control program manages the execution of user programs to present the errors in proper use of the computer. It is especially concerned of the user the operation and controls the I/O devices.

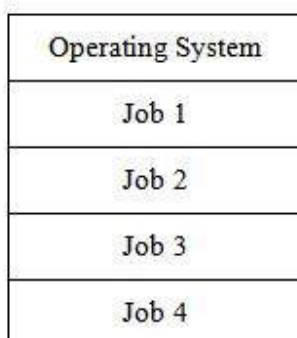
TYPES OF OPERATING SYSTEM

1. **Mainframe System:** It is the system where the first computer used to handle many commercial scientific applications. The growth of mainframe systems traced from simple batch system where the computer runs one and only one application to time shared systems which allowed for user interaction with the computer system
 - a. **Batch /Early System:** Early computers were physically large machine. The common input devices were card readers, tape drivers. The common output devices were line printers, tape drivers and card punches. In these systems the user did not interact directly with the computer system. Instead the user preparing a job which consists of programming data and some control information and then submitted it to the computer operator after some time the output is appeared. The output in these early computer was fairly simple is main task was to transfer control automatically from one job to next. The operating system always resides in the memory. To speed up processing operators batched the jobs with similar needs and ran then together as a group. The disadvantages of batch system are that in this execution environment the CPU is often idle because the speed up of I/O devices is much slower than the CPU.

Memory Layout for a Simple Batch System



- b. **Multiprogrammed System:** Multiprogramming concept increases CPU utilization by organization jobs so that the CPU always has one job to execute the idea behind multiprogramming concept. The operating system keeps several jobs in memory simultaneously as shown in below figure.



This set of job is subset of the jobs kept in the job pool. The operating system picks and beginning to execute one of the jobs in the memory. In this environment the operating system simply switches and executes another job. When a job needs to wait the CPU is simply switched to another job and so on. The multiprogramming operating system is sophisticated because the operating system makes decisions for the user. This is known as scheduling. If several jobs are ready to run at the same time the system choose one among them. This is known as CPU scheduling. The disadvantages of the multiprogrammed system are

- It does not provide user interaction with the computer system during the program execution.
- The introduction of disk technology solved these problems rather than reading the cards from card reader into disk. This form of processing is known as spooling.

SPOOL stands for simultaneous peripheral operations online. It uses the disk as a huge buffer for reading from input devices and for storing output data until the output devices accept them. It is also use for processing data at remote sides. The remote processing is done and its own speed with no CPU intervention. Spooling overlaps the input, output one job with computation of other jobs. Spooling has a beneficial effect on the performance of the systems by keeping both CPU and I/O devices working at much higher time.

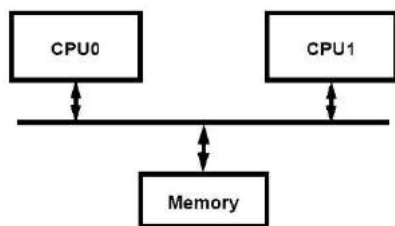
- c. **Time Sharing System:** The time-sharing system is also known as multi user systems. The CPU executes multiple jobs by switching among them but the switches occurs so frequently that the user can interact with each program while it is running. An interactive computer system provides direct communication between a user and system. The user gives instruction to the operating systems or

to a program directly using keyboard or mouse and wait for immediate results. So, the response time will be short. The time-sharing system allows many users to share the computer simultaneously. Since each action in this system is short, only a little CPU time is needed for each user. The system switches rapidly from one user to the next so each user feels as if the entire computer system is dedicated to his use, even though it is being shared by many users. The disadvantages of time-sharing system are:

- It is more complex than multiprogrammed operating system
 - The system must have memory management & protection, since several jobs are kept in memory at the same time.
 - Time sharing system must also provide a file system, so disk management is required.
 - It provides mechanism for concurrent execution which requires complex CPU scheduling schemes.
2. **Personal Computer System/Desktop System:** Personal computer appeared in 1970's. They are microcomputers that are smaller & less expensive than mainframe systems. Instead of maximizing CPU & peripheral utilization, the systems opt for maximizing user convenience & responsiveness. At first file protection was not necessary on a personal machine. But when other computers 2nd other users can access the files on a pc file protection becomes necessary. The lack of protection made it easy for malicious programs to destroy data on such systems. These programs maybe self-replicating & they spread via worm or virus mechanisms. They can disrupt entire companies or even world-wide networks. E.g : windows 98, windows 2000, Linux.
 3. **Microprocessor Systems/ Parallel Systems/ Tightly coupled Systems:** These Systems have more than one processor in close communications which share the computer bus, clock, memory & peripheral devices. Ex: UNIX, LINUX. Multiprocessor Systems have 3 main advantages.
 - a. **Increased throughput:** No. of processes computed per unit time. By increasing the no. of processors more work can be done in less time. The speed up ratio with N processors is not N, but it is less than N. Because a certain amount of overhead is incurred in keeping all the parts working correctly.
 - b. **Increased Reliability:** If functions can be properly distributed among several processors, then the failure of one processor will not halt the system, but slow it down. This ability to continue to operate in spite of failure makes the system fault tolerant.
 - c. **Economic scale:** Multiprocessor systems can save money as they can share peripherals, storage & power supplies.

The various types of multiprocessing systems are:

- **Symmetric Multiprocessing (SMP):** Each processor runs an identical copy of the operating system & these copies communicate with one another as required. Ex: Encore's version of UNIX for multi max computer. Virtually, all modern operating system including Windows NT, Solaris, Digital UNIX, OS/2 & LINUX now provide support for SMP.



- **Asymmetric Multiprocessing (Master – Slave Processors):** Each processor is designed for a specific task. A master processor controls the system & schedules & allocates the work to the slave processors. Ex- Sun's Operating system SUNOS version 4 provides asymmetric multiprocessing.
4. **Distributed System/Loosely Coupled Systems:** In contrast to tightly coupled systems, the processors do not share memory or a clock. Instead, each processor has its own local memory. The processors communicate with each other by various communication lines such as high speed buses or telephone lines. Distributed systems depend on networking for their functionalities. By being able to communicate distributed systems are able to share computational tasks and provide a rich set of features to the users. Networks vary by the protocols used, the distances between the nodes and transport media. TCP/IP is the most common network protocol. The processor in a distributed system varies in size and function. It may be microprocessors, work stations, minicomputer, and large general purpose computers. Network types are based on the distance between the nodes such as LAN (within a room, floor or building) and WAN (between buildings, cities or countries). The advantages of distributed system are resource sharing, computation speed up, reliability, communication.
 5. **Real time Systems:** Real time system is used when there are rigid time requirements on the operation of a processor or flow of data. Sensors bring data to the computers. The computer analyzes data and adjusts controls to modify the sensors inputs. Systems that control scientific experiments, medical imaging systems and some display systems are real time systems. The disadvantages of real time system are:
 - a. A real time system is considered to function correctly only if it returns the correct result within the time constraints.
 - b. Secondary storage is limited or missing instead data is usually stored in short term memory or ROM.
 - c. Advanced OS features are absent.

Real time system is of two types such as:

- **Hard real time systems:** It guarantees that the critical task has been completed on time. The sudden task takes place at a sudden instant of time.
- **Soft real time systems:** It is a less restrictive type of real time system where a critical task gets priority over other tasks and retains that priority until it computes. These have more limited utility than hard real time systems. Missing an occasional deadline is acceptable e.g. QNX, VX works. Digital audio or multimedia is included in this category.

It is a special purpose OS in which there are rigid time requirements on the operation of a processor. A real time OS has well defined fixed time constraints. Processing must be done within the time constraint or the system will fail. A real time system is said to function correctly only if it returns the correct result within the time constraint. These systems are characterized by having time as a key parameter.

BASIC FUNCTIONS OF OPERATION SYSTEM:

The various functions of operating system are as follows:

1. Process Management:

- A program does nothing unless their instructions are executed by a CPU. A process is a program in execution. A time shared user program such as a compiler is a process. A word processing program being run by an individual user on a pc is a process.
- A system task such as sending output to a printer is also a process. A process needs certain resources including CPU time, memory files & I/O devices to accomplish its task.
- These resources are either given to the process when it is created or allocated to it while it is running. The OS is responsible for the following activities of process management.
- Creating & deleting both user & system processes.
- Suspending & resuming processes.
- Providing mechanism for process synchronization.
- Providing mechanism for process communication.
- Providing mechanism for deadlock handling.

2. Main Memory Management:

The main memory is central to the operation of a modern computer system. Main memory is a large array of words or bytes ranging in size from hundreds of thousand to billions. Main memory stores the quickly accessible data shared by the CPU & I/O device. The central processor reads instruction from main memory during instruction fetch cycle & it both reads & writes data from main memory during the data fetch cycle. The main memory is generally the only large storage device that the CPU is able to address & access directly. For example, for the CPU to process data from disk. Those data must first be transferred to main memory by CPU generated E/O calls. Instruction must be in memory for the CPU to execute them. The OS is responsible for the following activities in connection with memory management.

- Keeping track of which parts of memory are currently being used & by whom.
- Deciding which processes are to be loaded into memory when memory space becomes available.
- Allocating & deallocating memory space as needed.

3. File Management:

File management is one of the most important components of an OS computer can store information on several different types of physical media magnetic tape, magnetic disk & optical disk are the most common media. Each medium is controlled by a device such as disk drive or tape drive those has unique characteristics. These characteristics include access speed, capacity, data transfer rate & access method (sequential or random). For convenient use of computer system the OS provides a uniform logical view of information storage. The OS abstracts from the physical properties of its storage devices to define a logical storage unit the file. A file is collection of related information defined by its creator. The OS is responsible for the following activities of file management.

- Creating & deleting files.
- Creating & deleting directories.
- Supporting primitives for manipulating files & directories.

- Mapping files into secondary storage.
- Backing up files on non-volatile media.

4. I/O System Management:

One of the purposes of an OS is to hide the peculiarities of specific hardware devices from the user. For example, in UNIX the peculiarities of I/O devices are hidden from the bulk of the OS itself by the I/O subsystem. The I/O subsystem consists of:

- A memory management component that includes buffering, catching & spooling.
- A general device- driver interfaces drivers for specific hardware devices. Only the device driver knows the peculiarities of the specific device to which it is assigned.

5. Secondary Storage Management:

The main purpose of computer system is to execute programs. These programs with the data they access must be in main memory during execution. As the main memory is too small to accommodate all data & programs & because the data that it holds are lost when power is lost. The computer system must provide secondary storage to back-up main memory. Most modern computer systems are disks as the storage medium to store data & program. The operating system is responsible for the following activities of disk management.

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

Because secondary storage is used frequently it must be used efficiently.

NETWORKING:

A distributed system is a collection of processors that don't share memory peripheral devices or a clock. Each processor has its own local memory & clock and the processor communicate with one another through various communication lines such as high speed buses or networks. The processors in the system are connected through communication networks which are configured in a number of different ways. The communication network design must consider message routing & connection strategies are the problems of connection & security.

Protection or security:

If a computer system has multi users & allow the concurrent execution of multiple processes then the various processes must be protected from one another's activities. For that purpose, mechanisms ensure that files, memory segments, CPU & other resources can be operated on by only those processes that have gained proper authorization from the OS.

Command interpretation:

One of the most important functions of the OS is connected interpretation where it acts as the interface between the user & the OS.

SYSTEM CALLS:

System calls provide the interface between a process & the OS. These are usually available in the form of assembly language instruction. Some systems allow system calls to be made directly from a high level language program like C, BCPL and PERL etc. systems calls

occur in different ways depending on the computer in use. System calls can be roughly grouped into 5 major categories.

1. **Process Control:**

- **End, abort:** A running program needs to be able to has its execution either normally (end) or abnormally (abort).
- **Load, execute:**A process or job executing one program may want to load and executes another program.
- **Create Process, terminate process:** There is a system call specifying for the purpose of creating a new process or job (create process or submit job). We may want to terminate a job or process that we created (terminates process, if we find that it is incorrect or no longer needed).
- **Get process attributes, set process attributes:** If we create a new job or process we should able to control its execution. This control requires the ability to determine & reset the attributes of a job or processes (get process attributes, set process attributes).
- **Wait time:** After creating new jobs or processes, we may need to wait for them to finish their execution (wait time).
- **Wait event, signal event:** We may wait for a specific event to occur (wait event). The jobs or processes then signal when that event has occurred (signal event).

2. **File Manipulation:**

- **Create file, delete file:** We first need to be able to create & delete files. Both the system calls require the name of the file & some of its attributes.
- **Open file, close file:** Once the file is created, we need to open it & use it. We close the file when we are no longer using it.
- **Read, write, reposition file:** After opening, we may also read, write or reposition the file (rewind or skip to the end of the file).
- **Get file attributes, set file attributes:** For either files or directories, we need to be able to determine the values of various attributes & reset them if necessary. Two system calls get file attribute & set file attributes are required for their purpose.

3. **Device Management:**

- **Request device, release device:** If there are multiple users of the system, we first request the device. After we finished with the device, we must release it.
- **Read, write, reposition:** Once the device has been requested & allocated to us, we can read, write & reposition the device.

4. **Information maintenance:**

- **Get time or date, set time or date:**Most systems have a system call to return the current date & time or set the current date & time.
- **Get system data, set system data:** Other system calls may return information about the system like number of current users, version number of OS, amount of free memory etc.
- **Get process attributes, set process attributes:** The OS keeps information about all its processes & there are system calls to access this information.

5. **Communication:** There are two modes of communication such as:

- **Message passing model:** Information is exchanged through an inter process communication facility provided by operating system. Each computer in a network

has a name by which it is known. Similarly, each process has a process name which is translated to an equivalent identifier by which the OS can refer to it. The get hostid and get processed systems calls to do this translation. These identifiers are then passed to the general purpose open & close calls provided by the file system or to specific open connection system call. The recipient process must give its permission for communication to take place with an accept connection call. The source of the communication known as client & receiver known as server exchange messages by read message & write message system calls. The close connection call terminates the connection.

- **Shared memory model:** processes use map memory system calls to access regions of memory owned by other processes. They exchange information by reading & writing data in the shared areas. The processes ensure that they are not writing to the same location simultaneously.

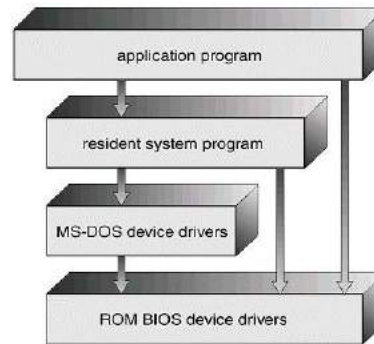
SYSTEM PROGRAMS:

System programs provide a convenient environment for program development & execution. They are divided into the following categories.

- **File manipulation:** These programs create, delete, copy, rename, print & manipulate files and directories.
- **Status information:** Some programs ask the system for date, time & amount of available memory or disk space, no. of users or similar status information.
- **File modification:** Several text editors are available to create and modify the contents of file stored on disk.
- **Programming language support:** compilers, assemblers & interpreters are provided to the user with the OS.
- **Programming loading and execution:** Once a program is assembled or compiled, it must be loaded into memory to be executed.
- **Communications:** These programs provide the mechanism for creating virtual connections among processes users 2nd different computer systems.
- **Application programs:** Most OS are supplied with programs that are useful to solve common problems or perform common operations. Ex: web browsers, word processors & text formatters etc.

SYSTEM STRUCTURE:

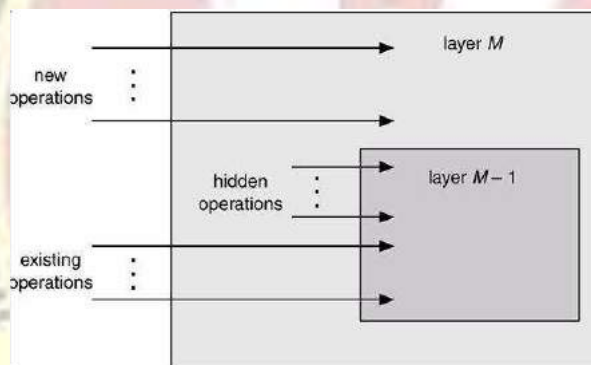
1. **Simple structure:** There are several commercial system that don't have a well-defined structure such operating systems begins as small, simple & limited systems and then grow beyond their original scope. MS-DOS is an example of such system. It was not divided into modules carefully. Another example of limited structuring is the UNIX operating system.



(MS DOS Structure)

2. Layered approach: In the layered approach, the OS is broken into a number of layers (levels) each built on top of lower layers. The bottom layer (layer 0) is the hardware & top most layer (layer N) is the user interface. The main advantage of the layered approach is modularity.

- The layers are selected such that each users functions (or operations) & services of only lower layer.



- This approach simplifies debugging & system verification, i.e. the first layer can be debugged without concerning the rest of the system. Once the first layer is debugged, its correct functioning is assumed while the 2nd layer is debugged & so on.
- If an error is found during the debugging of a particular layer, the error must be on that layer because the layers below it are already debugged. Thus the design & implementation of the system are simplified when the system is broken down into layers.
- Each layer is implemented using only operations provided by lower layers. A layer doesn't need to know how these operations are implemented; it only needs to know what these operations do.
- The layer approach was first used in the operating system. It was defined in six layers.

Layers	Functions
5	User Program
4	I/O Management

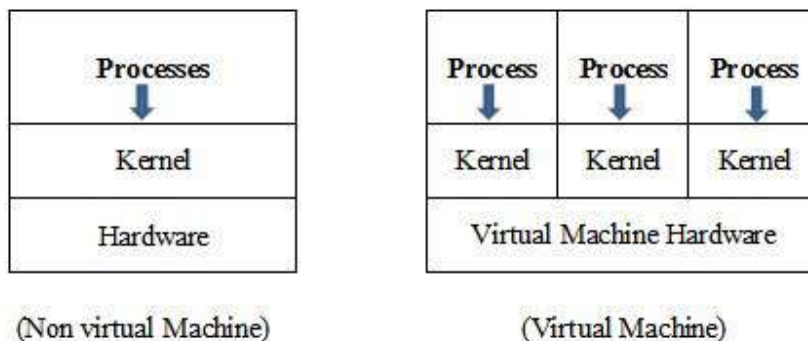
3	Operator Process Communication
2	Memory Management
1	CPU Scheduling
0	Hardware

The main disadvantage of the layered approach is:

- The main difficulty with this approach involves the careful definition of the layers, because a layer can use only those layers below it. For example, the device driver for the disk space used by virtual memory algorithm must be at a level lower than that of the memory management routines, because memory management requires the ability to use the disk space.
- It is less efficient than a non layered system (Each layer adds overhead to the system call & the net result is a system call that take longer time than on a non layered system).

VIRTUAL MACHINES:

By using CPU scheduling & virtual memory techniques an operating system can create the illusion of multiple processes, each executing on its own processors & own virtual memory. Each processor is provided a virtual copy of the underlying computer. The resources of the computer are shared to create the virtual machines. CPU scheduling can be used to create the appearance that users have their own processor.



Implementation: Although the virtual machine concept is useful, it is difficult to implement since much effort is required to provide an exact duplicate of the underlying machine. The CPU is being multiprogrammed among several virtual machines, which slows down the virtual machines in various ways.

Difficulty: A major difficulty with this approach is regarding the disk system. The solution is to provide virtual disks, which are identical in all respects except size. These are known as mini disks in IBM's VM OS. The sum of sizes of all mini disks should be less than the actual amount of physical disk space available.

I/O Structure

A general purpose computer system consists of a CPU and multiple device controller which is connected through a common bus. Each device controller is in charge of a specific type of device. A device controller maintains some buffer storage and a set of special purpose register. The device controller is responsible for moving the data between peripheral devices and buffer storage.

I/O Interrupt: To start an I/O operation the CPU loads the appropriate register within the device controller. In turn the device controller examines the content of the register to determine the actions which will be taken. For example, suppose the device controller finds the read request then, the controller will start the transfer of data from the device to the buffer. Once the transfer of data is complete the device controller informs the CPU that the operation has been finished. Once the I/O is started, two actions are possible such as

- In the simplest case the I/O is started then at I/O completion control is return to the user process. This is known as synchronous I/O.

The other possibility is asynchronous I/O in which the control is return to the user program without waiting for the I/O completion. The I/O then continues with other operations.

When an interrupt occurs first determine which I/O device is responsible for interrupting. After searching the I/O device table the signal goes to the each I/O request. If there are additional request waiting in the queue for one device the operating system starts processing the next request. Finally, control is return from the I/O interrupt.

DMA controller: DMA is used for high speed I/O devices. In DMA access the device controller transfers on entire block of data to of from its own buffer storage to memory. In this access the interrupt is generated per block rather than one interrupt per byte. The operating system finds a buffer from the pool of buffers for the transfer. Then a portion of the operating system called a device driver sets the DMA controller registers to use appropriate source and destination addresses and transfer length. The DMA controller is then instructed to start the I/O operation. While the DMA controller is performing the data transfer, the CPU is free to perform other tasks. Since the memory generally can transfer only one word at a time, the DMA controller steals memory cycles from the CPU. This cycle stealing can slow down the CPU execution while a DMA transfer is in progress. The DMA controller interrupts the CPU when the transfer has been completed. **Storage Structure**

The storage structure of a computer system consists of two types of memory such as

- Main memory
- Secondary memory

Basically the programs & data are resided in main memory during the execution. The programs and data are not stored permanently due to following two reasons.

- Main memory is too small to store all needed programs and data permanently.
- Main memory is a volatile storage device which lost its contents when power is turned off. **Main Memory:** The main memory and the registers are the only storage area that the CPU can access the data directly without any help of other device. The machine instruction which take memory address as arguments do not take disk address. Therefore in execution any instructions and any data must be resided in any one of direct

access storage device. If the data are not in memory they must be moved before the CPU can operate on them. There are two types of main memory such as:

RAM (Random Access Memory): The RAM is implemented in a semiconductor technology is called D-RAM (Dynamic RAM) which forms an array of memory words/cells. Each & every word should have its own address/locator. Instruction is performed through a sequence of load and store instruction to specific memory address. Each I/O controller includes register to hold commands of the data being transferred. To allow more convenient access to I/O device many computer architecture provide memory mapped I/O. In the case of memory mapped I/O ranges of memory address are mapped to the device register. Read and write to this memory address because the data to be transferred to and from the device register.

Secondary Storage: The most common secondary storage devices are magnetic disk and magnetic tape which provide permanent storage of programs and data.

Magnetic Disk: It provides the bulk of secondary storage for modern computer systems. Each disk platter has flat circular shape like a CD. The diameter of a platter range starts from 1.8 to 5.25 inches. The two surfaces of a platter are covered with a magnetic material which records the information/data is given by the user. The read, write head are attached to a disk arm, which moves all the heads as a unit. The surface of a platter is logically divided into circular tracks which are sub divided into sectors. The set of tracks which are at one arm position forms a cylinder. There are may be thousands of cylinders in a disk drive & each track contains 100 of sectors. The storage capacity of a common disk drive is measured in GB. When the disk is in use a drive motor spins it at high speed. Most drives rotated 62 to 200 time/sec. The disk speed has two parts such as transfer rate & positioning time. The transfer rate is the rate at which data flow between the drive & the computer. The positioning time otherwise called as random access time. It consists of two parts such as seek time & rotational latency. The seek time is the time taken to move the disk arc to the desired cylinder. The rotational latency is the time taken to rotate the disk head.

Magnetic Tape: It was used as early secondary storage medium. It is also permanent and can hold large quantity of data. Its access time is slower, comparison to main memory devices. Magnetic tapes are sequential in nature. That's why random access to magnetic tape is thousand times slower than the random access to magnetic disk. The magnetic tapes are used mainly for backup the data. The magnetic tape must be kept in a non dusty environment and temperature controlled area. But the main advantage of the secondary storage device is that it can hold 2 to 3 times more data than a large disk drive. There are 4 types of magnetic tapes such as:

- ½ Inch
- ¼ Inch
- 4 mm
- 8 mm

OPERATING SYSTEM SERVICES

An operating system provides an environment for the execution of the program. It provides some services to the programs. The various services provided by an operating system are as follows:

- **Program Execution:** The system must be able to load a program into memory and to run that program. The program must be able to terminate this execution either normally or abnormally.
- **I/O Operation:** A running program may require I/O. This I/O may involve a file or a I/O device for specific device. Some special function can be desired. Therefore the operating system must provide a means to do I/O.
- **File System Manipulation:** The programs need to create and delete files by name and read and write files. Therefore the operating system must maintain each and every files correctly.
- **Communication:** The communication is implemented via shared memory or by the technique of message passing in which packets of information are moved between the processes by the operating system.
- **Error detection:** The operating system should take the appropriate actions for the occurrences of any type like arithmetic overflow, access to the illegal memory location and too large user CPU time.
- **Resource Allocation:** When multiple users are logged on to the system the resources must be allocated to each of them. For current distribution of the resource among the various processes the operating system uses the CPU scheduling run times which determine which process will be allocated with the resource.
- **Accounting:** The operating system keep track of which users use how many and which kind of computer resources.
- **Protection:** The operating system is responsible for both hardware as well as software protection. The operating system protects the information stored in a multiuser computer system.

PROCESS MANAGEMENT:

Process: A process or task is an instance of a program in execution. The execution of a process must programs in a sequential manner. At any time at most one instruction is executed. The process includes the current activity as represented by the value of the program counter and the content of the processors registers. Also it includes the process stack which contain temporary data (such as method parameters return address and local variables) & a data section which contain global variables.

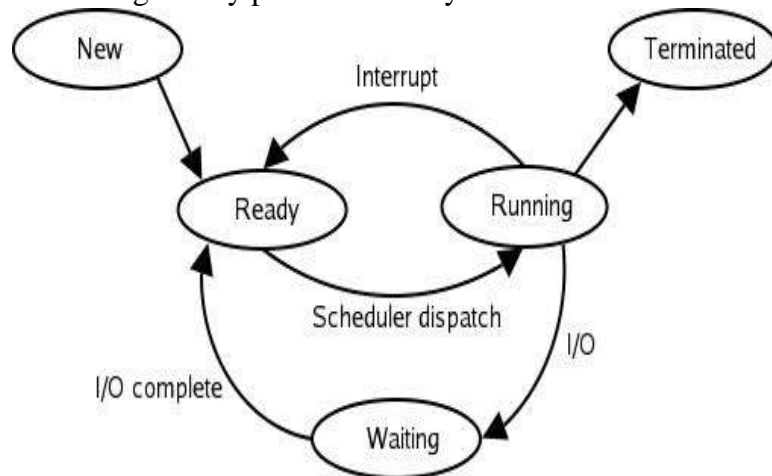
Difference between process & program:

A program by itself is not a process. A program in execution is known as a process. A program is a passive entity, such as the contents of a file stored on disk where as process is an active entity with a program counter specifying the next instruction to execute and a set of associated resources may be shared among several process with some scheduling algorithm being used to determinate when the stop work on one process and service a different one.

Process state: As a process executes, it changes state. The state of a process is defined by the correct activity of that process. Each process may be in one of the following states.

- **New:** The process is being created.
- **Ready:** The process is waiting to be assigned to a processor.
- **Running:** Instructions are being executed.
- **Waiting:** The process is waiting for some event to occur.
- **Terminated:** The process has finished execution.

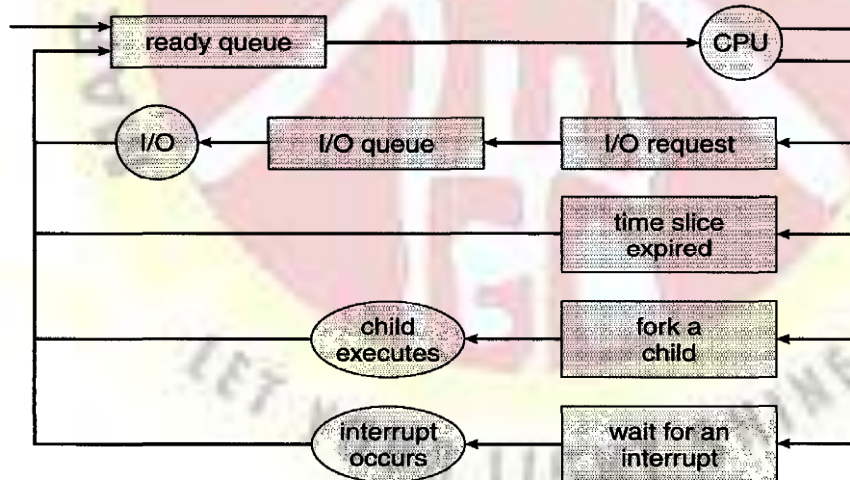
Many processes may be in ready and waiting state at the same time. But only one process can be running on any processor at any instant.



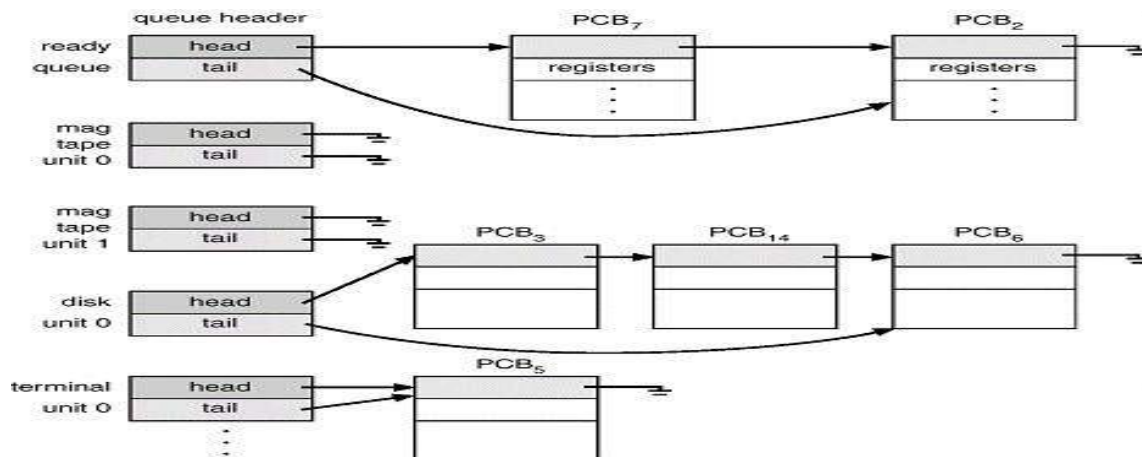
Process scheduling:

Scheduling is a fundamental function of OS. When a computer is multiprogrammed, it has multiple processes competing for the CPU at the same time. If only one CPU is available, then a choice has to be made regarding which process to execute next. This decision making process is known as scheduling and the part of the OS that makes this choice is called a scheduler. The algorithm it uses in making this choice is called scheduling algorithm.

Scheduling queues: As processes enter the system, they are put into a job queue. This queue consists of all process in the system. The process that are residing in main memory and are ready & waiting to execute or kept on a list called ready queue.



This queue is generally stored as a linked list. A ready queue header contains pointers to the first & final PCB in the list. The PCB includes a pointer field that points to the next PCB in the ready queue. The lists of processes waiting for a particular I/O device are kept on a list called device queue. Each device has its own device queue. A new process is initially put in the ready queue. It waits in the ready queue until it is selected for execution & is given the CPU

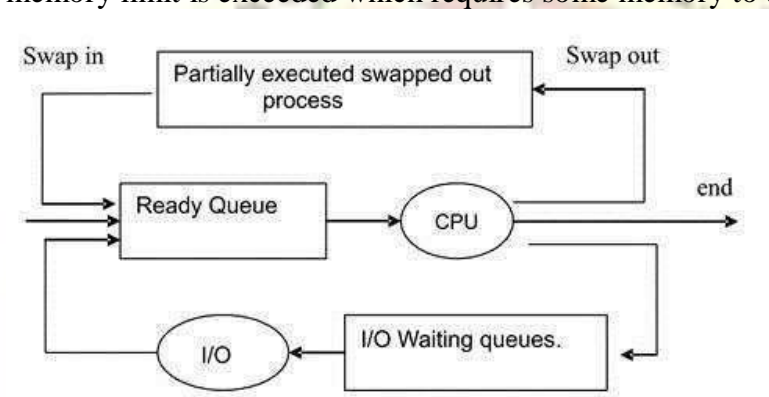


SCHEDULERS: A process migrates between the various scheduling queues throughout its life-time purposes. The OS must select for scheduling processes from these queues in some fashion. This selection process is carried out by the appropriate scheduler. In a batch system, more processes are submitted and then executed immediately. So, these processes are spooled to a mass storage device like disk, where they are kept for later execution.

Types of schedulers: There are 3 types of schedulers mainly used:

1. **Long term scheduler:** Long term scheduler selects process from the disk & loads them into memory for execution. It controls the degree of multi-programming i.e. no. of processes in memory. It executes less frequently than other schedulers. If the degree of multiprogramming is stable than the average rate of process creation is equal to the average departure rate of processes leaving the system. So, the long term scheduler is needed to be invoked only when a process leaves the system. Due to longer intervals between executions it can afford to take more time to decide which process should be selected for execution. Most processes in the CPU are either I/O bound or CPU bound. An I/O bound process (an interactive 'C' program is one that spends most of its time in I/O operation than it spends in doing I/O operation. A CPU bound process is one that spends more of its time in doing computations than I/O operations (complex sorting program). It is important that the long term scheduler should select a good mix of I/O bound & CPU bound processes.

2. **Short - term scheduler:** The short term scheduler selects among the process that are ready to execute & allocates the CPU to one of them. The primary distinction between these two schedulers is the frequency of their execution. The short-term scheduler must select a new process for the CPU quite frequently. It must execute at least one in 100ms. Due to the short duration of time between executions, it must be very fast.
3. **Medium - term scheduler:** some operating systems introduce an additional intermediate level of scheduling known as medium - term scheduler. The main idea behind this scheduler is that sometimes it is advantageous to remove processes from memory & thus reduce the degree of multiprogramming. At some later time, the process can be reintroduced into memory & its execution can be continued from where it had left off. This is called as swapping. The process is swapped out & swapped in later by medium term scheduler. Swapping is necessary to improve the process miss or due to some change in memory requirements, the available memory limit is exceeded which requires some memory to be freed up.



Process control block:

Each process is represented in the OS by a process control block. It is also by a process control block. It is also known as task control block.

pointer	process state
process number	
program counter	
registers	
memory limits	
list of open files	
⋮	

A process control block contains many pieces of information associated with a specific process.

It includes the following information's.

- **Process state:** The state may be new, ready, running, waiting or terminated state.
- **Program counter:** it indicates the address of the next instruction to be executed for this purpose.

- **CPU registers:** The registers vary in number & type depending on the computer architecture. It includes accumulators, index registers, stack pointer & general purpose registers, plus any condition- code information must be saved when an interrupt occurs to allow the process to be continued correctly after- ward.
- **CPU scheduling information:** This information includes process priority pointers to scheduling queues & any other scheduling parameters.
- **Memory management information:** This information may include such information as the value of the base & limit registers, the page tables or the segment tables, depending upon the memory system used by the operating system.
- **Accounting information:** This information includes the amount of CPU and real time used, time limits, account number, job or process numbers and so on.
- **I/O Status Information:** This information includes the list of I/O devices allocated to this process, a list of open files and so on. The PCB simply serves as the repository for any information that may vary from process to process.

CPU Scheduling Algorithm:

CPU Scheduling deals with the problem of deciding which of the processes in the ready queue is to be allocated first to the CPU. There are four types of CPU scheduling that exist.

1. **First Come, First Served Scheduling (FCFS) Algorithm:** This is the simplest CPU scheduling algorithm. In this scheme, the process which requests the CPU first, that is allocated to the CPU first. The implementation of the FCFS algorithm is easily managed with a FIFO queue. When a process enters the ready queue its PCB is linked onto the rear of the queue. The average waiting time under FCFS policy is quite long. Consider the following example:

Process	CPU time
P ₁	3
P ₂	5
P ₃	2
P ₄	4

Using FCFS algorithm find the average waiting time and average turnaround time if the order is

P₁, P₂, P₃, P₄.

Solution: If the process arrived in the order P₁, P₂, P₃, P₄ then according to the FCFS the Gantt chart will be:

P1	P2	P3	P4
0	3	8	10
			14

The waiting time for process P₁ = 0, P₂ = 3, P₃ = 8, P₄ = 10 then the turnaround time for process P₁ = 0 + 3 = 3, P₂ = 3 + 5 = 8, P₃ = 8 + 2 = 10, P₄ = 10 + 4 = 14.

Then average waiting time = $(0 + 3 + 8 + 10)/4 = 21/4 = 5.25$

Average turnaround time = $(3 + 8 + 10 + 14)/4 = 35/4 = 8.75$

The FCFS algorithm is non preemptive means once the CPU has been allocated to a process then the process keeps the CPU until the release the CPU either by terminating or requesting I/O.

2. **Shortest Job First Scheduling (SJF) Algorithm:** This algorithm associates with each process if the CPU is available. This scheduling is also known as shortest next CPU burst, because the scheduling is done by examining the length of the next CPU burst of the process rather than its total length. Consider the following example:

Process	CPU time
P ₁	3
P ₂	5
P ₃	2
P ₄	4

Solution:According to the SJF the Gantt chart will be

P3	P1	P2	P4
0	2	5	9
			14

The waiting time for process P₁ = 0, P₂ = 2, P₃ = 5, P₄ = 9 then the turnaround time for process P₃ = 0 + 2 = 2, P₁ = 2 + 3 = 5, P₄ = 5 + 4 = 9, P₂ = 9 + 5 = 14.

Then average waiting time = $(0 + 2 + 5 + 9)/4 = 16/4 = 4$

Average turnaround time = $(2 + 5 + 9 + 14)/4 = 30/4 = 7.5$

The SJF algorithm may be either preemptive or non preemptive algorithm. The preemptive SJF is also known as shortest remaining time first.

Consider the following example.

Process	Arrival Time	CPU time
P ₁	0	8
P ₂	1	4
P ₃	2	9
P ₄	3	5

In this case the Gantt chart will be

P1	P2	P4	P1	P3
0	1	5	10	17
				26

The waiting time for process

$$P_1 = 10 - 1 = 9$$

$$P_2 = 1 - 1 = 0$$

$$P_3 = 17 - 2 = 15$$

$$P_4 = 5 - 3 = 2$$

The average waiting time = $(9 + 0 + 15 + 2)/4 = 26/4 = 6.5$

3. **Priority Scheduling Algorithm:** In this scheduling a priority is associated with each process and the CPU is allocated to the process with the highest priority. Equal priority processes are scheduled in FCFS manner. Consider the following example:

Process	Arrival Time	CPU time
P ₁	10	3
P ₂	1	1
P ₃	2	3
P ₄	1	4
P ₅	5	2

According to the priority scheduling the Gantt chart will be

P2	P5	P1	P3	P4
----	----	----	----	----

0 1 6 16 18 19

The waiting time for process

$$P_1 = 6$$

$$P_2 = 0$$

$$P_3 = 16$$

$$P_4 = 18$$

$$P_5 = 1$$

$$\text{The average waiting time} = (0 + 1 + 6 + 16 + 18)/5 = 41/5 = 8.2$$

4. **Round Robin Scheduling Algorithm:** This type of algorithm is designed only for the time-sharing system. It is similar to FCFS scheduling with preemption condition to switch between processes. A small unit of time called quantum time or time slice is used to switch between the processes. The average waiting time under the round robin policy is quite long. Consider the following example:

Process	CPU time
P ₁	3
P ₂	5
P ₃	2
P ₄	4

Time Slice = 1 millisecond.

P1	P2	P3	P4	P1	P2	P3	P4	P1	P2	P4	P2	P4	P2	
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14

The waiting time for process

$$P_1 = 0 + (4 - 1) + (8 - 5) = 0 + 3 + 3 = 6$$

$$P_2 = 1 + (5 - 2) + (9 - 6) + (11 - 10) + (12 - 11) + (13 - 12) = 1 + 3 + 3 + 1 + 1 + 1 = 10$$

$$P_3 = 2 + (6 - 3) = 2 + 3 = 5$$

$$P_4 = 3 + (7 - 4) + (10 - 8) + (12 - 11) = 3 + 3 + 2 + 1 = 9$$

$$\text{The average waiting time} = (6 + 10 + 5 + 9)/4 = 7.5$$

UNIT 2

PROCESS SYNCHRONIZATION:

A co-operation process is one that can affect or be affected by other processes executing in the system. Co-operating process may either directly share a logical address space or be allotted to the shared data only through files. This concurrent access is known as Process synchronization.

CRITICAL SECTION PROBLEM:

Consider a system consisting of n processes (P_0, P_1, \dots, P_{n-1}) each process has a segment of code which is known as critical section in which the process may be changing common variable, updating a table, writing a file and so on. The important feature of the system is that when the process is executing in its critical section no other process is to be allowed to execute in its critical section. The execution of critical sections by the processes is a mutually exclusive. The critical section problem is to design a protocol that the process can use to cooperate each process must request permission to enter its critical section. The section of code implementing this request is the entry section. The critical section is followed on exit section. The remaining code is the remainder section.

Example:

```

While (1)
{
  Entry Section;
    Critical Section;
  Exit Section;
    Remainder Section;
}

```

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

1. **Mutual Exclusion:** If process P_i is executing in its critical section then no any other process can be executing in their critical section.
2. **Progress:** If no process is executing in its critical section and some process wish to enter their critical sections then only those process that are not executing in their remainder section can enter its critical section next.
3. **Bounded waiting:** There exists a bound on the number of times that other processes are allowed to enter their critical sections after a process has made a request.

SEMAPHORES:

For the solution to the critical section problem one synchronization tool is used which is known as semaphores. A semaphore 'S' is an integer variable which is accessed through two standard operations such as wait and signal. These operations were originally termed 'P' (for wait means to test) and 'V' (for single means to increment). The classical definition of wait is


```

Wait (S)
{
    While (S <= 0)
    {
        Test;
    }
    S--;
}

```

The classical definition of the signal is

```

Signal (S)
{
    S++;
}

```

In case of wait the test condition is executed with interruption and the decrement is executed without interruption.

Binary Semaphore:

A binary semaphore is a semaphore with an integer value which can range between 0 and 1. Let 'S' be a counting semaphore. To implement the binary semaphore we need following the structure of data. Binary Semaphores S_1, S_2 ; int C; Initially $S_1 = 1, S_2 = 0$ and the value of C is set to the initial value of the counting semaphore 'S'.

Then the wait operation of the binary semaphore can be implemented as follows.

```

Wait
(S1) C--
; if (C <
0)
{
    Signal (S1);
    Wait (S2);
}
Signal (S1);

```

The signal operation of the binary semaphore can be implemented as follows:

```

Wait (S1);
C++;
if (C <= 0)
Signal (S2);
Else

```

Signal (S₁);

CLASSICAL PROBLEM ON SYNCHRONIZATION:

There are various types of problem which are proposed for synchronization scheme such as

- **Bounded Buffer Problem:** This problem was commonly used to illustrate the power of synchronization primitives. In this scheme we assumed that the pool consists of 'N' buffer and each capable of holding one item. The 'mutex' semaphore provides mutual exclusion for access to the buffer pool and is initialized to the value one. The empty and full semaphores count the number of empty and full buffer respectively. The semaphore empty is initialized to 'N' and the semaphore full is initialized to zero. This problem is known as producer and consumer problem. The code of the producer is producing full buffer and the code of consumer is producing empty buffer. The structure of producer process is as follows:

```
do {
  produce an item in nextp
```

```
.....
Wait (empty);
```

```
Wait (mutex);
```

```
.....
```

```
add nextp to buffer
```

```
.....
```

```
Signal (mutex);
```

```
Signal (full);
```

```
} While (1);
```

The structure of consumer process is as follows:

```
do {
```

```
Wait (full);
```

```
Wait
(mutex);
```

```
.....
```

```
Remove an item from buffer to nextc
```

```
.....
```

```
Signal (mutex);
```

```
Signal (empty);
```

```
.....
```

```
Consume the item in nextc;
```

```
.....
```

```
} While (1);
```

- **READER WRITER PROBLEM:** In this type of problem there are two types of process are used such as Reader process and Writer process. The reader process is

responsible for only reading and the writer process is responsible for writing. This is an important problem of synchronization which has several variations like o The simplest one is referred as first reader writer problem which requires that no reader will be kept waiting unless a writer has obtained permission to use the shared object. In other words no reader should wait for other reader to finish because a writer is waiting.

- o The second reader writer problem requires that once a writer is ready then the writer performs its write operation as soon as possible.

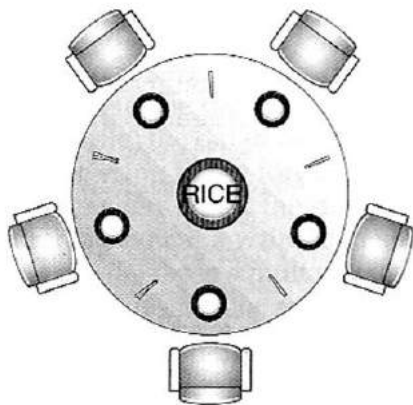
The structure of a reader process is as follows:

```
Wait (mutex);
Read count++; if
(read count == 1)
Wait (wrt);
Signal
(mutex);
.....
Reading is performed
.....
Wait (mutex);
Read count --; if
(read count == 0)
Signal (wrt);
Signal (mutex);
```

The structure of the writer process is as follows:

```
Wait (wrt);
Writing is performed;
Signal (wrt);
```

- **DINING PHILOSOPHER PROBLEM:** Consider 5 philosophers to spend their lives in thinking & eating. A philosopher shares common circular table surrounded by 5 chairs each occupies by one philosopher. In the center of the table there is a bowl of rice and the table is laid with 6 chopsticks as shown in below figure.



When a philosopher thinks she does not interact with her colleagues. From time to time a philosopher gets hungry and tries to pickup two chopsticks that are closest to her. A philosopher may pickup one chopstick or two chopsticks at a time but she

cannot pick up a chopstick that is already in hand of the neighbor. When a hungry philosopher has both her chopsticks at the same time, she eats without releasing her chopsticks. When she finished eating, she puts down both of her chopsticks and starts thinking again. This problem is considered as classic synchronization problem. According to this problem each chopstick is represented by a semaphore. A philosopher grabs the chopsticks by executing the wait operation on that semaphore. She releases the chopsticks by executing the signal operation on the appropriate semaphore. The structure of dining philosopher is as follows:

```
do{
  Wait ( chopstick [i]);
  Wait      (chopstick
  [(i+1)%5]);
  .....
  Eat
  .....
  Signal (chopstick [i]);
  Signal      (chopstick
  [(i+1)%5]);
  .....
  Think
  .....
} While (1);
```

CRITICAL REGION:

According to the critical section problem using semaphore all processes must share a semaphore variable `mutex` which is initialized to one. Each process must execute `wait (mutex)` before entering the critical section and execute the `signal (mutex)` after completing the execution but there are various difficulties may arise with this approach like:

Case 1: Suppose that a process interchanges the order in which the wait and signal operations on the semaphore `mutex` are executed, resulting in the following execution:

```
Signal (mutex);
```

```
.....
```

```
Critical Section
```

```
.....
```

```
Wait (mutex);
```

In this situation several processes may be executing in their critical sections simultaneously, which is violating mutual exclusion requirement.

Case 2: Suppose that a process replaces the `signal (mutex)` with `wait (mutex)`. The execution is as follows: `Wait (mutex);`

```
.....
```

```
Critical Section
```

```
.....
```

```
Wait (mutex);
```

In this situation a deadlock will occur

Case 3: Suppose that a process omits the wait (mutex) and the signal (mutex). In this case the mutual exclusion is violated or a deadlock will occur.

To illustrate the various types of error generated by using semaphore there are some high level language constructs have been introduced such as critical region and monitor.

Critical region is also known as conditional critical regions. It constructs guards against certain simple errors associated with semaphore. This high level language synchronization construct requires a variable V of type T which is to be shared among many processes. It is declared as V: shared T;

The variable V can be accessed only inside a region statement as like below:

```

Wait (mutex);
While (! B) {
First_count++; if
(second_count> 0)
    Signal (second_delay);
Else
    Signal (mutex);
Wait (first_delay);
First_count--;
Second_count++;
if (first_count> 0)
    Signal (first_delay);
Else
    Signal (second_delay);
Wait (second_delay);
Second_count --;
}
S;
if (first_count> 0)
    Signal (first_delay);
Else if (second_count> 0)
    Signal (second_delay);
Else
    Signal (mutex);

```

(Implementation of the conditional region constructs)

Where B is a Boolean variable which governs the access to the critical regions which is initialized to false. Mutex, First_delay and Second_delay are the semaphores which are initialized to 1, 0, and 0 respectively. First_count and Second_count are the integer variables which are initialized to zero.

MONITOR:

It is characterized as a set of programmer defined operators. Its representation consists of declaring of variables, whose value defines the state of an instance. The syntax of monitor is as follows. Monitor monitor_name

```
{
  Shared variable declarations  Procedure
  body P1 (.....) {
    .....
  }
  Procedure body P2 (.....)
{
  .....
}
.
.
.
  Procedure body Pn (.....)
{
  .....
}
  {
  Initialization Code
}
}
```

ATOMIC TRANSACTION:

This section is related to the field of database system. Atomic transaction describes the various techniques of database and how they are can be used by the operating system. It ensures that the critical sections are executed automatically. To determine how the system should ensure atomicity we need first to identify the properties of the devices used to for storing the data accessed by the transactions. The various types storing devices are as follows:

- **Volatile Storage:** Information residing in volatile storage does not survive in case of system crash. Example of volatile storage is main memory and cache memory.
- **Non volatile Storage:** Information residing in this type of storage usually survives in case of system crash. Examples are Magnetic Disk, Magnetic Tape and Hard Disk.
- **Stable Storage:** Information residing in stable storage is never lost. Example is non volatile cache memory.

The various techniques used for ensuring the atomicity are as follows:

1. **Log based Recovery:** This technique is used for achieving the atomicity by using data structure called log. A log has the following fields:
 - a. **Transaction Name:** This is the unique name of the transaction that performed the write operation.
 - b. **Data Item Name:** This is the unique name given to the data.
 - c. **Old Value:** This is the value of the data before to the write operation.
 - d. **New value:** This is the value of the data after the write operation.

This recovery technique uses two processes such as Undo and Redo. Undo restores the value of old data updated by a transaction to the old values. Redo sets the value of the data updated by a transaction to the new values.

2. **Checkpoint:** In this principle system maintains the log. The checkpoint requires the following sequences of action.
 - a. Output all the log records from volatile storage into stable storage.
 - b. Output all modified data residing in volatile to the stable storage.
 - c. Output checkpoint onto the stable storage.

T0	Write (B)	T1
Read (A)		
Write (A)		
Read (B)		

3. **Serializability:** In this technique the transaction Read (A) executed serially in some arbitrary order. Consider a system Write (A) consisting two data items A and B which are both read and Read (B) written by two transactions T_0 and T_1 . Suppose that their Write (B) transactions are executed automatically in the order

T_0 followed by T_1 . This execution sequence is known as schedule which is represented as below.

If transactions are overlapped then their execution resulting schedule is known as non-serial scheduling or concurrent schedule as like below:

T0	T1
Read (A)	
Write (A)	
	Read (A)
	Write (A)
Read (B)	
Write (B)	
	Read (B)
	Write (B)

4. **Locking:** This technique governs how the locks are acquired and released. There are two types of lock such as shared lock and exclusive lock. If a transaction T has obtained a shared lock (S) on data item Q then T can read this item but cannot write. If a transaction

T has obtained an exclusive lock (S) on data item Q then T can both read and write in the data item Q.

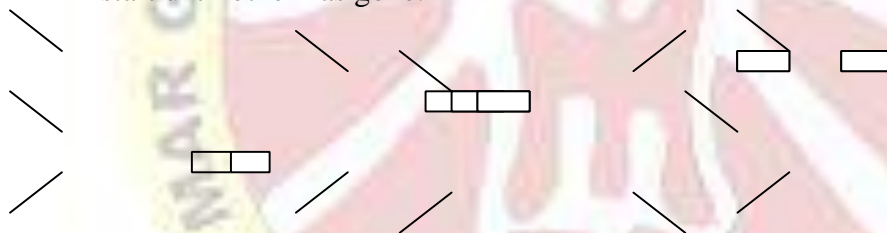
5. **Timestamp:** In this technique each transaction in the system is associated with unique fixed timestamp denoted by TS. This timestamp is assigned by the system before the transaction starts. If a transaction T_i has been assigned with a timestamp $TS(T_i)$ and later a new transaction T_j enters the system then $TS(T_i) < TS(T_j)$. There are two types of timestamp such as W-timestamp and R-timestamp. W-timestamp denotes the largest timestamp of any transaction that performed write operation successfully. R-timestamp denotes the largest timestamp of any transaction that executed read operation successfully.

DEADLOCK:

In a multiprogramming environment several processes may compete for a finite number of resources. A process request resources; if the resource is available at that time a process enters the wait state. Waiting process may never change its state because the resources requested are held by other waiting process. This situation is known as deadlock.

Example

- System has 2 disk drives.
- P1 and P2 each hold one disk drive and each needs another one.
- 2 train approaches each other at crossing, both will come to full stop and neither shall start until other has gone.



- Traffic only in one direction.
- Each section of a bridge can be viewed as a resource.
- If a deadlock occurs, it can be resolved if one car backs up (preempt resources and rollback).
- Several cars may have to be backed up if a deadlock occurs.
- Starvation is possible **System Model:**

A system consists of a finite number of resources to be distributed among a number of competing processes. The resources are partitioned into several types each of which consists of a number of identical instances. A process may utilize a resource in the following sequence

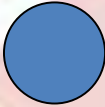
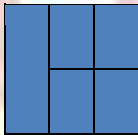
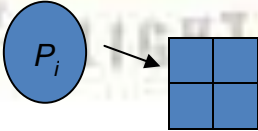
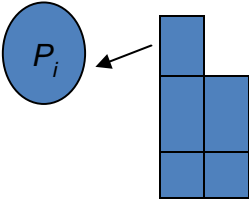
- **Request:** In this state one can request a resource.
- **Use:** In this state the process operates on the resource.
- **Release:** In this state the process releases the resources.

DEADLOCK CHARACTERISTICS: In a deadlock process never finish executing and system resources are tied up. A deadlock situation can arise if the following four conditions hold simultaneously in a system.

- **Mutual Exclusion:** At a time only one process can use the resources. If another process requests that resource, requesting process must wait until the resource has been released.
- **Hold and wait:** A process must be holding at least one resource and waiting to additional resource that is currently held by other processes.
- **No Preemption:** Resources allocated to a process can't be forcibly taken out from it unless it releases that resource after completing the task.
- **Circular Wait:** A set $\{P_0, P_1, \dots, P_n\}$ of waiting state/ process must exists such that P_0 is waiting for a resource that is held by P_1 , P_1 is waiting for the resource that is held by P_2 $P_{(n-1)}$ is waiting for the resource that is held by P_n and P_n is waiting for the resources that is held by P_0 .

RESOURCE ALLOCATION GRAPH:

Deadlock can be described more clearly by directed graph which is called system resource allocation graph. The graph consists of a set of vertices 'V' and a set of edges 'E'. The set of vertices 'V' is partitioned into two different types of nodes such as $P = \{P_1, P_2, \dots, P_n\}$, the set of all the active processes in the system and $R = \{R_1, R_2, \dots, R_m\}$, the set of all the resource type in the system. A directed edge from process P_i to resource type R_j is denoted by $P_i \rightarrow R_j$. It signifies that process P_i is an instance of resource type R_j and waits for that resource. A directed edge from resource type R_j to the process P_i which signifies that an instance of resource type R_j has been allocated to process P_i . A directed edge $P_i \rightarrow R_j$ is called as request edge and $R_j \rightarrow P_i$ is called as assigned edge.

- Process 
- Resource Type with 4 instances 
- P_i requests instance of R_j 
- P_i is holding an instance of R_j 

When a process P_i requests an instance of resource type R_j then a request edge is inserted as resource allocation graph. When this request can be fulfilled, the request edge is transformed to an assignment edge. When the process no longer needs access to the resource it releases

the resource and as a result the assignment edge is deleted. The resource allocation graph shown in below figure has the following situation.

- The sets P, R, E

$$P = \{P_1, P_2, P_3\}$$

$$R = \{R_1, R_2, R_3, R_4\}$$

$$E = \{P_1 \rightarrow R_1, P_2 \rightarrow R_3, R_1 \rightarrow P_2, R_2 \rightarrow P_2, R_2 \rightarrow P_1, R_3 \rightarrow P_3\}$$

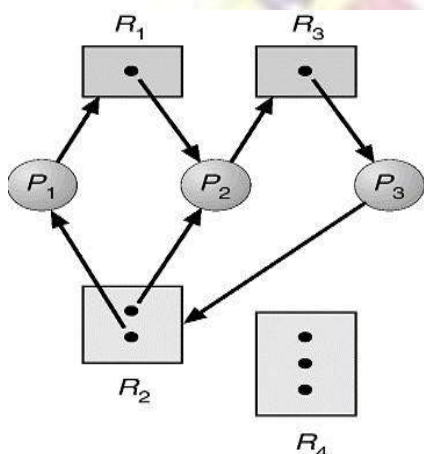
The resource instances are

Resource R_1 has one instance

Resource R_2 has two instances.

Resource R_3 has one instance

Resource R_4 has three instances.



The process states are:

Process P_1 is holding an instance of R_2 and waiting for an instance of R_1 .

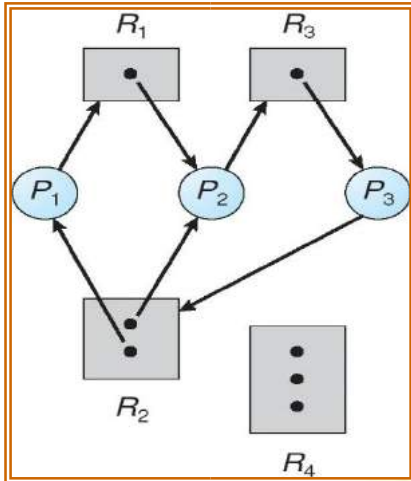
Process P_2 is holding an instance of R_1 and R_2 and waiting for an instance R_3 .

Process P_3 is holding an instance of R_3 .

The following example shows the resource allocation graph with a deadlock.

$P_1 \rightarrow R_1 \rightarrow P_2 \rightarrow R_3 \rightarrow P_3 \rightarrow R_2 \rightarrow P_1$

$P_2 \rightarrow R_3 \rightarrow P_3 \rightarrow R_2 \rightarrow P_1$



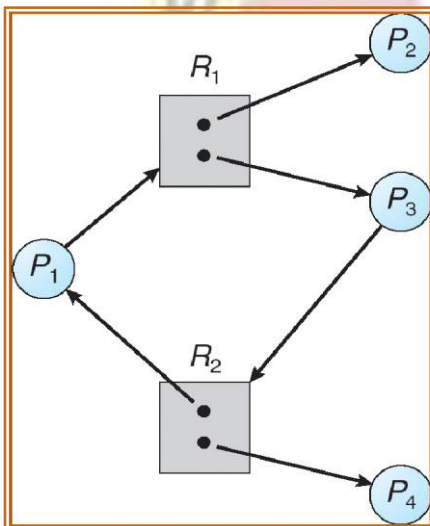
The following example shows the resource allocation graph with a cycle but no deadlock.

$P_1 \rightarrow R_1 \rightarrow P_3 \rightarrow R_2 \rightarrow P_1$

No deadlock

P_4 may release its instance of resource R_2

- Then it can be allocated to P_3



Methods for Handling Deadlocks

The problem of deadlock can deal with the following 3 ways.

We can use a protocol to prevent or avoid deadlock ensuring that the system will never enter to a deadlock state.

We can allow the system to enter a deadlock state, detect it and recover.

We can ignore the problem all together.

To ensure that deadlock never occur the system can use either a deadlock prevention or deadlock avoidance scheme.

DEADLOCK PREVENTION:

Deadlock prevention is a set of methods for ensuring that at least one of these necessary conditions cannot hold.

Mutual Exclusion: The mutual exclusion condition holds for non sharable. The example is a printer cannot be simultaneously shared by several processes. Sharable resources do not require mutual exclusive access and thus cannot be involved in a dead lock. The example is read only files which are in sharing condition. If several processes attempt to open the read only file at the same time they can be guaranteed simultaneous access.

Hold and wait: To ensure that the hold and wait condition never occurs in the system, we must guaranty that whenever a process requests a resource it does not hold any other resources. There are two protocols to handle these problems such as one protocol that can be used requires each process to request and be allocated all its resources before it begins execution. The other protocol allows a process to request resources only when the process has no resource. These protocols have two main disadvantages. First, resource utilization may be low, since many of the resources may be allocated but unused for a long period. Second, starvation is possible. A process that needs several popular resources may have to wait indefinitely, because at least one of the resources that it needs is always allocated to some other process.

No Preemption: To ensure that this condition does not hold, a protocol is used. If a process is holding some resources and request another resource that cannot be immediately allocated to it. The preempted one added to a list of resources for which the process is waiting. The process will restart only when it can regain its old resources, as well as the new ones that it is requesting. Alternatively if a process requests some resources, we first check whether they are available. If they are, we allocate them. If they are not available, we check whether they are allocated to some other process that is waiting for additional resources. If so, we preempt the desired resources from the waiting process and allocate them to the requesting process. If the resources are not either available or held by a waiting process, the requesting process must wait.

Circular Wait: We can ensure that this condition never holds by ordering of all resource type and to require that each process requests resource in an increasing order of enumeration. Let R

$= \{R_1, R_2, \dots, R_n\}$ be the set of resource types. We assign to each resource type a unique integer number, which allows us to compare two resources and to determine whether one precedes another in our ordering. Formally, we define a one to one function $F: R \rightarrow N$, where N is the set of natural numbers. For example, if the set of resource types R includes tape drives, disk drives and printers, then the function F might be defined as follows:

$F(\text{Tape Drive}) = 1,$

$F(\text{Disk Drive}) = 5,$

$F(\text{Printer}) = 12.$

We can now consider the following protocol to prevent deadlocks: Each process can request resources only in an increasing order of enumeration. That is, a process can initially request any number of instances of a resource type, say R_i . After that, the process can request instances of resource type R_j if and only if $F(R_j) > F(R_i)$. If several instances of the same resource type are needed, defined previously, a process that wants to use the tape drive and printer at the same time must first request the tape drive and then request the printer.

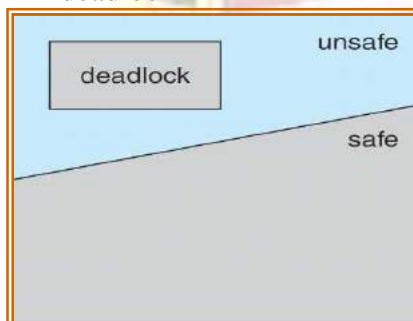
DEADLOCK AVOIDANCE

Requires additional information about how resources are to be used. Simplest and most useful model requires that each process declare the maximum number of resources of each type that

it may need. The deadlock-avoidance algorithm dynamically examines the resource-allocation state to ensure that there can never be a circular-wait condition. Resource-allocation state is defined by the number of available and allocated resources, and the maximum demands of the processes. **Safe State**

When a process requests an available resource, system must decide if immediate allocation leaves the system in a safe state. Systems are in safe state if there exists a safe sequence of all process. A sequence $\langle P_1, P_2, \dots, P_n \rangle$ of ALL the processes is the system such that for each P_i , the resources that P_i can still request can be satisfied by currently available resources + resources held by all the P_j , with $j < i$. That is:

- If P_i resource needs are not immediately available, then P_i can wait until all P_j have finished.
- When P_j is finished, P_i can obtain needed resources, execute, return allocated resources, and terminate.
- When P_i terminates, P_{i+1} can obtain its needed resources, and so on.
- If system is in safe state \Rightarrow No deadlock
- If system is not in safe state \Rightarrow possibility of deadlock
- OS cannot prevent processes from requesting resources in a sequence that leads to deadlock
- Avoidance \Rightarrow ensure that system will never enter an unsafe state, prevent getting into deadlock



Example:

	Maximum Needs	Current Needs
P_0	10	5
P_1	4	2
P_2	9	2

- Suppose processes P_0 , P_1 , and P_2 share 12 magnetic tape drives • Currently 9 drives are held among the processes and 3 are available
- Question: Is this system currently in a safe state?
- Answer: Yes!
 - o Safe Sequence: $\langle P_1, P_0, P_2 \rangle$

	Maximum Needs	Current Needs
P_0	10	5
P_1	4	2
P_2	9	2

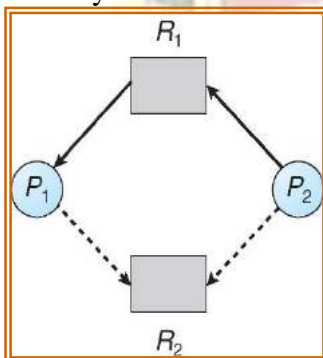
- Suppose process P_2 requests and is allocated 1 more tape drive.
- Question: Is the resulting state still safe?
- Answer: No! Because there does not exist a safe sequence anymore.

Only P_1 can be allocated its maximum needs.

IF P_0 and P_2 request 5 more drives and 6 more drives, respectively, then the resulting state will be deadlocked.

RESOURCE ALLOCATION GRAPH ALGORITHM

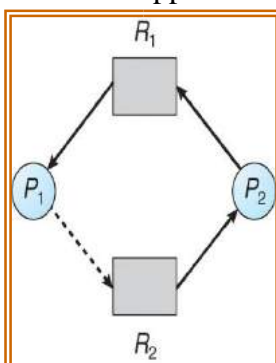
In this graph a new type of edge has been introduced is known as claim edge. Claim edge $P_i \rightarrow R_j$ indicates that process P_j may request resource R_j ; represented by a dashed line. Claim edge converts to request edge when a process requests a resource. Request edge converted to an assignment edge when the resource is allocated to the process. When a resource is released by a process, assignment edge reconverts to a claim edge. Resources must be claimed a priori in the system.



P_2 requesting R_1 , but R_1 is already allocated to P_1 .

Both processes have a claim on resource R_2

What happens if P_2 now requests resource R_2 ?



Cannot allocate resource R_2 to process P_2

Why? Because resulting state is unsafe

- P1 could request R2, thereby creating deadlock!

Use only when there is a single instance of each resource type

- Suppose that process P_i requests a resource R_j
- The request can be granted only if converting the request edge to an assignment edge does not result in the formation of a cycle in the resource allocation graph.
- Here we check for safety by using cycle-detection algorithm. **Banker's Algorithm**

This algorithm can be used in banking system to ensure that the bank never allocates all its available cash such that it can no longer satisfy the needs of all its customer. This algorithm is applicable to a system with multiple instances of each resource type. When a new process enter in to the system it must declare the maximum number of instances of each resource type that it may need. This number may not exceed the total number of resources in the system. Several data structure must be maintained to implement the banker's algorithm.

Let,

- n = number of processes
- m = number of resources types

Available: Vector of length m . If $Available[j] = k$, there are k instances of resource type R_j available.

Max: $n \times m$ matrix. If $Max [i,j] = k$, then process P_i may request at most k instances of resource type R_j .

Allocation: $n \times m$ matrix. If $Allocation[i,j] = k$ then P_i is currently allocated k instances of R_j .

Need: $n \times m$ matrix. If $Need[i,j] = k$, then P_i may need k more instances of R_j to complete its task.

$Need [i,j] = Max[i,j] - Allocation [i,j]$.

SAFETY ALGORITHM

1. Let Work and Finish be vectors of length m and n , respectively. Initialize: Work = Available
Finish $[i] = false$ for $i = 0, 1, \dots, n- 1$.
2. Find and i such that both:
 - (a) Finish $[i] = false$
 - (b) $Need_i \leq Work$
 If no such i exists, go to step 4.
3. Work = Work + Allocation $_i$ Finish $[i] = true$ go to step 2.
4. If Finish $[i] == true$ for all i , then the system is in a safe state.

RESOURCE ALLOCATION ALGORITHM

Request = request vector for process P_i . If Request $_i[j] = k$ then process P_i wants k instances of resource type R_j .

1. If $Request_i \leq Need_i$ go to step 2. Otherwise, raise error condition, since process has exceeded its maximum claim.
2. If $Request_i \leq Available$, go to step 3. Otherwise P_i must wait, since resources are not available.
3. Pretend to allocate requested resources to P_i by modifying the state as follows:

$$Available = Available - Request;$$

$$Allocation_i = Allocation_i + Request_i;$$

$$Need_i = Need_i - Request_i;$$

- If safe \Rightarrow the resources are allocated to P_i .
- If unsafe $\Rightarrow P_i$ must wait, and the old resource-allocation state is restored

Example

- 5 processes P_0 through P_4 ;
- 3 resource types:
A (10 instances), B (5 instances), and C (7 instances).

- Snapshot at time T_0 :

	<u>Allocation</u>			<u>Max</u>			<u>Available</u>		
A B C	A B C	A B C	A B C	A B C	A B C	A B C	A B C	A B C	
P_0	0	1	0	7	5	3	3	3	2
P_1	2	0	0	3	2	2			
P_2	3	0	2	9	0	2			
P_3	2	1	1	2	2	2			
P_4	0	0	2	4	3	3			

- The content of the matrix Need is defined to be $Max - Allocation$.

	<u>Need</u>		
A B C	A B C	A B C	A B C
P_0	7	4	3
P_1	1	2	2
P_2	6	0	0
P_3	0	1	1
P_4	4	3	1

- The system is in a safe state since the sequence $\langle P_1, P_3, P_4, P_2, P_0 \rangle$ satisfies safety criteria.

P_1 requests (1, 0, 2)

- Check that $Request \leq Available$ (that is, $(1,0,2) \leq (3,3,2) \Rightarrow true$).

<u>Allocation</u>	<u>Need</u>	<u>Available</u>
A B C	A B C	A B C A B C

P ₀	0 1 0	7 4 3 2 3 0
P ₁	3 0 2	0 2 0
P ₂	3 0 1	6 0 0
P ₃	2 1 1	0 1 1
P ₄	0 0 2	4 3 1

- Executing safety algorithm shows that sequence $\langle P_1, P_3, P_4, P_0, P_2 \rangle$ satisfies safety requirement.
- Can request for (3,3,0) by P₄ be granted? –NO
- Can request for (0,2,0) by P₀ be granted? –NO (Results Unsafe)

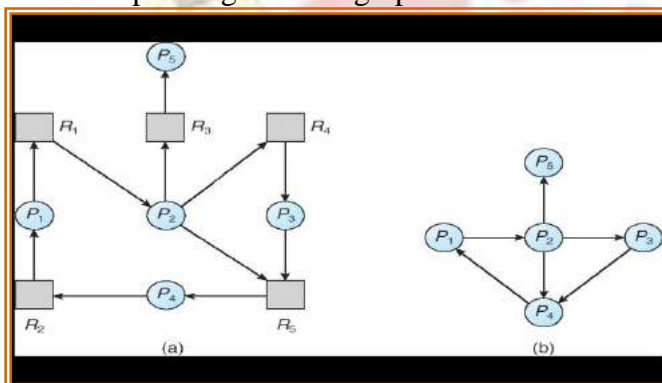
DEADLOCK DETECTION

If a system doesn't employ either a deadlock prevention or deadlock avoidance, then deadlock situation may occur. In this environment the system must provide

- An algorithm to recover from the deadlock.
- An algorithm to remove the deadlock is applied either to a system which pertains single in instance each resource type or a system which pertains several instances of a resource type.

Single Instance of each Resource type

If all resources only a single instance then we can define a deadlock detection algorithm which uses a new form of resource allocation graph called "Wait for graph". We obtain this graph from the resource allocation graph by removing the nodes of type resource and collapsing the appropriate edges. The below figure describes the resource allocation graph and corresponding wait for graph.



Resource-Allocation
Graph

Correspondin
wait-for graph

- For single instance
- P_i ->P_j(P_i is waiting for P_j to release a resource that P_i needs)

- $P_i \rightarrow P_j$ exist if and only if RAG contains 2 edges $P_i \rightarrow R_q$ and $R_q \rightarrow P_j$ for some resource R_q **Several Instances of a Resource type**

The wait for graph scheme is not applicable to a resource allocation system with multiple instances of each resource type. For this case the algorithm employs several data structures which are similar to those used in the banker's algorithm like available, allocation and request.

- **Available:** A vector of length m indicates the number of available resources of each type.
- **Allocation:** An $n \times m$ matrix defines the number of resources of each type currently allocated to each process.
- **Request:** An $n \times m$ matrix indicates the current request of each process. If Request $[i] = k$, then process P_i is requesting k more instances of resource type. R_j .

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

(a) Work = Available

(b) For $i = 1, 2, \dots, n$, if $\text{Allocation}_i \neq 0$, then $\text{Finish}[i] = \text{false}$; otherwise, $\text{Finish}[i] = \text{true}$.

2. Find an index i such that both:

(a) $\text{Finish}[i] == \text{false}$

(b) $\text{Request}_i \leq \text{Work}$

If no such i exists, go to step 4.

3. Work = Work + Allocation

Finish $[i] = \text{true}$

Go to step 2

4. If $\text{Finish}[i] = \text{false}$, for some i , $1 \leq i \leq n$, then the system is in a deadlock state. Moreover, if $\text{Finish}[i] = \text{false}$, then process P_i is deadlocked.

RECOVERY FROM DEADLOCK

When a detection algorithm determines that a deadlock exists, several alternatives exist. One possibility is to inform the operator that a deadlock has occurred, and to let the operator deal with the deadlock manually. The other possibility is to let the system recover from the deadlock automatically. There are two options for breaking a deadlock. One solution is simply to abort one or more processes to break the circular wait. The second option is to preempt some resources from one or more of the deadlocked processes.

Process Termination:

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

- **Abort all deadlocked processes:** This method clearly will break the deadlock cycle, but at a great expense; these processes may have computed for a long time, and the results of these partial computations must be discarded and probably recomputed later.
- **Abort one process at a time until the deadlock cycle is eliminated:** This method incurs considerable overhead, since after each process is aborted, a deadlock detection algorithm must be invoked to determine whether any processes are still deadlocked.

Resource Preemption:

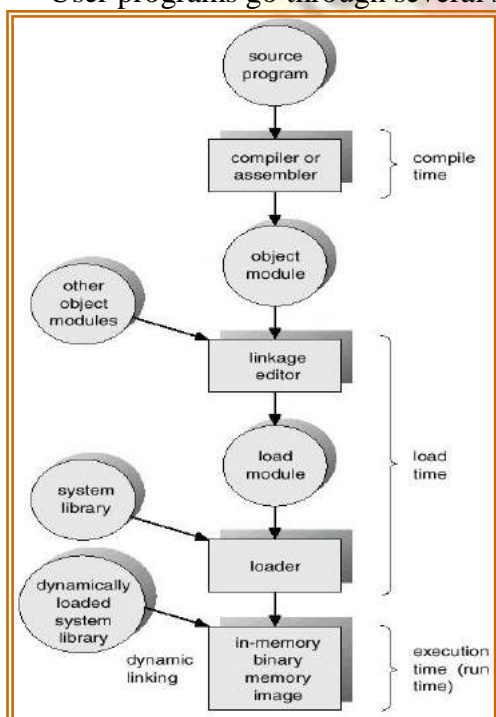
To eliminate deadlocks using resource preemption, we successively preempt some resources from processes and give these resources to other processes until the deadlock cycle is broken. If preemption is required to deal with deadlocks, then three issues need to be addressed.

- **Selecting a victim:** Which resources and which processes are to be preempted? As in process termination, we must determine the order of preemption to minimize cost. Cost factors may include such parameters as the numbers of resources a deadlocked process is holding, and the amount of time a deadlocked process has thus far consumed during its execution.
- **Rollback:** If we preempt a resource from a process, what should be done with that process? Clearly, it cannot continue with its normal execution; it is missing some needed resource. We must rollback the process to some safe state, and restart it from that state.
- **Starvation:** In a system where victim selection is based primarily on cost factors, it may happen that the same process is always picked as a victim. As a result, this process never completes its designated task, a starvation situation that needs to be dealt with in any practical system. Clearly, we must ensure that a process can be picked as a victim only a small finite number of times. The most common solution is to include the number of rollbacks in the cost factor.

UNIT III

MEMORY MANAGEMENT

- Memory consists of a large array of words or bytes, each with its own address. The CPU fetches instructions from memory according to the value of the program counter. These instructions may cause additional loading from and storing to specific memory addresses.
- Memory unit sees only a stream of memory addresses. It does not know how they are generated.
- Program must be brought into memory and placed within a process for it to be run.
- Input queue – collection of processes on the disk that are waiting to be brought into memory for execution.
- User programs go through several steps before being run.

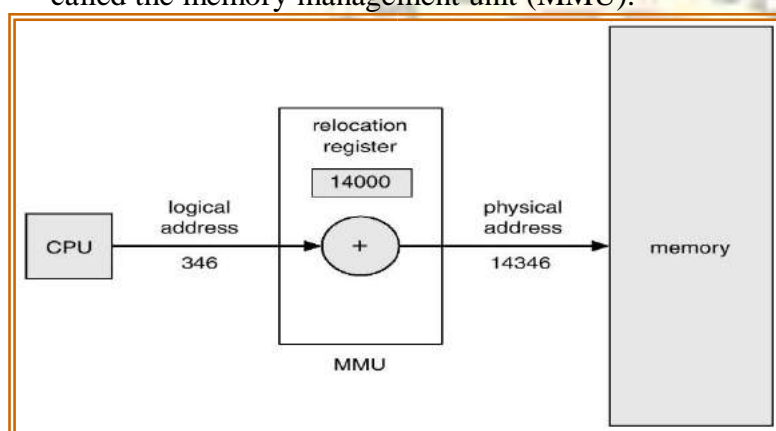


Address binding of instructions and data to memory addresses can happen at three different stages.

- **Compile time:** If memory location known a priori, absolute code can be generated; must recompile code if starting location changes.
Example: .COM-format programs in MS-DOS.
- **Load time:** Must generate relocatable code if memory location is not known at compile time.
- **Execution time:** Binding delayed until run time if the process can be moved during its execution from one memory segment to another. Need hardware support for address maps (e.g., relocation registers).

LOGICAL VERSUS PHYSICAL ADDRESS SPACE

- The concept of a logical address space that is bound to a separate physical address space is central to proper memory management.
 - Logical address – address generated by the CPU; also referred to as virtual address.
 - Physical address – address seen by the memory unit.
- The set of all logical addresses generated by a program is a logical address space; the set of all physical addresses corresponding to these logical addresses are a physical address space.
- Logical and physical addresses are the same in compile-time and load-time address-binding schemes; logical (virtual) and physical addresses differ in execution-time address-binding scheme.
- The run-time mapping from virtual to physical addresses is done by a hardware device called the memory management unit (MMU).



- This method requires hardware support slightly different from the hardware configuration. The base register is now called a relocation register. The value in the relocation register is added to every address generated by a user process at the time it is sent to memory.
- The user program never sees the real physical addresses. The program can create a pointer to location 346, store it in memory, manipulate it and compare it to other addresses. The user program deals with logical addresses. The memory mapping hardware converts logical addresses into physical addresses. The final location of a referenced memory address is not determined until the reference is made.

DYNAMIC LOADING

- Routine is not loaded until it is called.
- All routines are kept on disk in a relocatable load format.
- The main program is loaded into memory and is executed. When a routine needs to call another routine, the calling routine first checks to see whether the other the desired routine into memory and to update the program's address tables to reflect this change. Then control is passed to the newly loaded routine.
- Better memory-space utilization; unused routine is never loaded.
- Useful when large amounts of code are needed to handle infrequently occurring cases.
- No special support from the operating system is required.

- Implemented through program design.

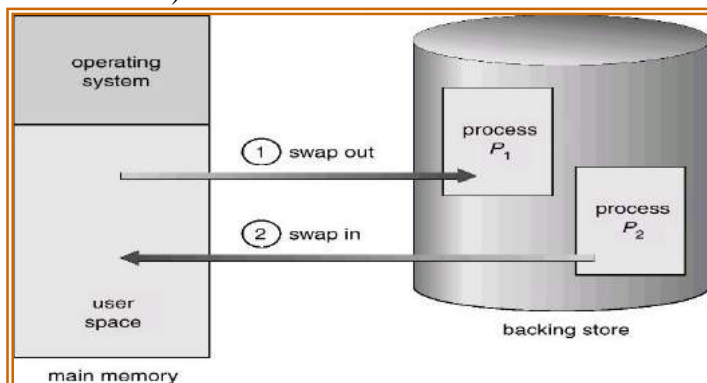
DYNAMIC LINKING

- Linking is postponed until execution time.
- Small piece of code, stub, is used to locate the appropriate memory-resident library routine, or to load the library if the routine is not already present.
- When this stub is executed, it checks to see whether the needed routine is already in memory. If not, the program loads the routine into memory.
- Stub replaces itself with the address of the routine, and executes the routine.
- Thus the next time that code segment is reached, the library routine is executed directly, incurring no cost for dynamic linking.
- Operating system is needed to check if routine is in processes' memory address.
- Dynamic linking is particularly useful for libraries.

SWAPPING

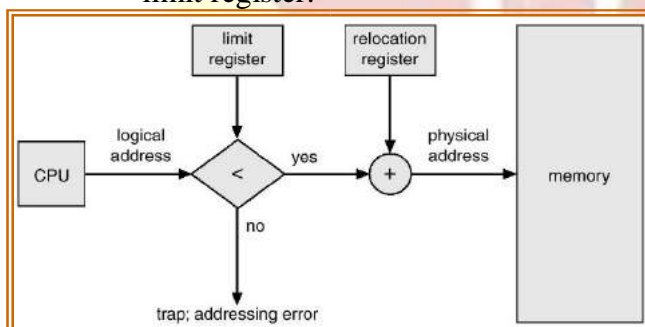
- A process can be swapped temporarily out of memory to a backing store, and then brought back into memory for continued execution. For example, assume a multiprogramming environment with a round robin CPU scheduling algorithm. When a quantum expires, the memory manager will start to swap out the process that just finished, and to swap in another process to the memory space that has been freed. In the mean time, the CPU scheduler will allocate a time slice to some other process in memory. When each process finished its quantum, it will be swapped with another process. Ideally, the memory manager can swap processes fast enough that some processes will be in memory, ready to execute, when the CPU scheduler wants to reschedule the CPU. The quantum must also be sufficiently large that reasonable amounts of computing are done between swaps.
- Roll out, roll in – swapping variant used for priority-based scheduling algorithms. If a higher priority process arrives and wants service, the memory manager can swap out the lower priority process so that it can load and execute lower priority process can be swapped back in and continued. This variant is some times called roll out, roll in. Normally a process that is swapped out will be swapped back into the same memory space that it occupied previously. This restriction is dictated by the process cannot be moved to different locations. If execution time binding is being used, then a process can be swapped into a different memory space, because the physical addresses are computed during execution time.
- Backing store – fast disk large enough to accommodate copies of all memory images for all users; must provide direct access to these memory images. It must be large enough to accommodate copies of all memory images for all users, and it must provide direct access to these memory images. The system maintains a ready queue consisting of all processes whose memory images are scheduler decides to execute a process it calls the dispatcher. The dispatcher checks to see whether the next process in the queue is in memory. If not, and there is no free memory region, the dispatcher swaps out a process currently in memory and swaps in the desired process. It then reloads registers as normal and transfers control to the selected process.

- Major part of swap time is transfer time; total transfer time is directly proportional to the amount of memory swapped.
- Modified versions of swapping are found on many systems (i.e., UNIX, Linux, and Windows).



CONTIGUOUS MEMORY ALLOCATION

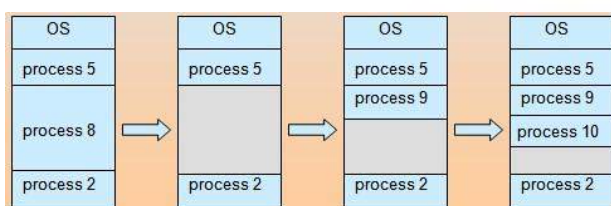
- Main memory is usually divided into two partitions:
 - Resident operating system, usually held in low memory with interrupt vector.
 - User processes, held in high memory.
- In contiguous memory allocation, each process is contained in a single contiguous section of memory.
- Single-partition allocation
 - Relocation-register scheme used to protect user processes from each other, and from changing operating-system code and data.
 - Relocation register contains value of smallest physical address; limit register contains range of logical addresses – each logical address must be less than the limit register.



- Multiple-partition allocation
 - Hole – block of available memory; holes of various size are scattered throughout memory.
 - When a process arrives, it is allocated memory from a hole large enough to accommodate it.
 - Operating system maintains information about:
 - a) allocated partitions
 - b) free partitions (hole)
 - A set of holes of various sizes, is scattered throughout memory at any given time. When a process arrives and needs memory, the system searches this set for a hole

that is large enough for this process. If the hole is too large, it is split into two: one part is allocated to the arriving process; the other is returned to the set of holes. When a process terminates, it releases its block of memory, which is then placed back in the set of holes. If the new hold is adjacent to other holes, these adjacent holes are merged to form one larger hole.

- This procedure is a particular instance of the general dynamic storage allocation problem, which is how to satisfy a request of size n from a list of free holes. There are many solutions to this problem. The set of holes is searched to determine which hole is best to allocate. The first-fit, best-fit and worst-fit strategies are the most common ones used to select a free hole from the set of available holes.



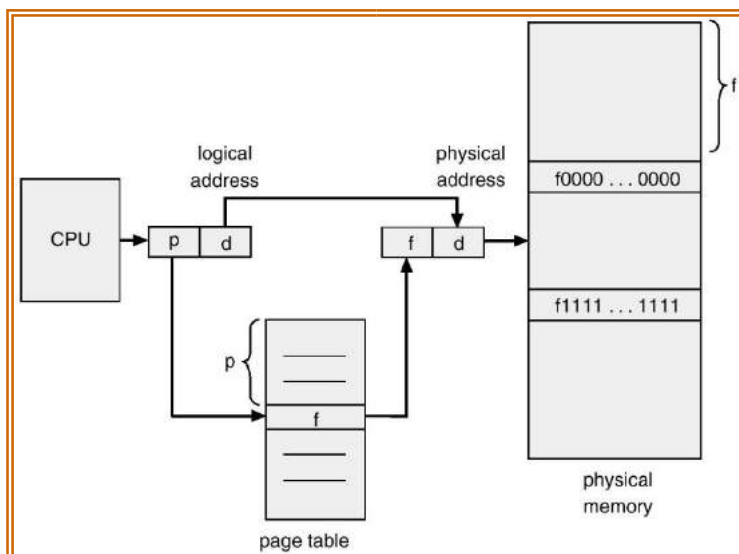
- **First-fit:** Allocate the first hole that is big enough.
- **Best-fit:** Allocate the smallest hole that is big enough; must search entire list, unless ordered by size.
- **Worst-fit:** Allocate the largest hole; must also search entire list.

FRAGMENTATION

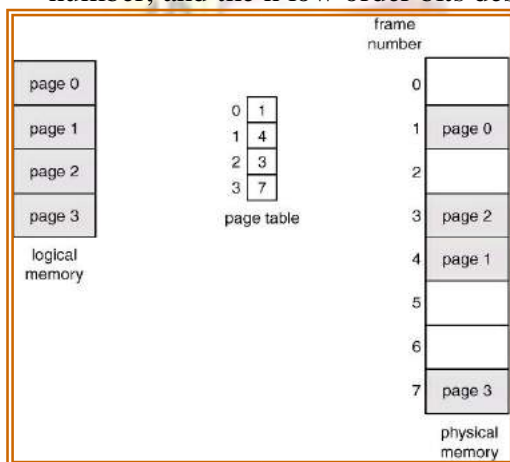
- **External Fragmentation** – total memory space exists to satisfy a request, but it is not contiguous.
- **Internal Fragmentation** – allocated memory may be slightly larger than requested memory; this size difference is memory internal to a partition, but not being used.
- Reduce external fragmentation by compaction
 - Shuffle memory contents to place all free memory together in one large block.
 - Compaction is possible only if relocation is dynamic, and is done at execution time.

PAGING

- Paging is a memory management scheme that permits the physical address space of a process to be non contiguous.
- Divide physical memory into fixed-sized blocks called **frames** (size is power of 2, for example 512 bytes).
- Divide logical memory into blocks of same size called **pages**. When a process is to be executed, its pages are loaded into any available memory frames from the backing store. The backing store is divided into fixed sized blocks that are of the same size as the memory frames.
- The hardware support for paging is illustrated in below figure.
- Every address generated by the CPU is divided into two parts: a page number (p) and a page offset (d). The page number is used as an index into a page table. The page table contains the base address of each page in physical memory. This base address is combined with the page offset to define the physical memory address that is sent to the memory unit.



- The paging model of memory is shown in below figure. The page size is defined by the hardware. The size of a page is typically of a power of 2, varying between 512 bytes and 16 MB per page, depending on the computer architecture. The selection of a power of 2 as a page size makes the translation of a logical address into a page number and page offset particularly easy. If the size of logical address is 2^m , and a page size is 2^n addressing units, then the high order $m-n$ bits of a logical address designate the page number, and the n low order bits designate the page offset.



- Keep track of all free frames.
- To run a program of size n pages, need to find n free frames and load program.
- Set up a page table to translate logical to physical addresses.
- Internal fragmentation may occur.

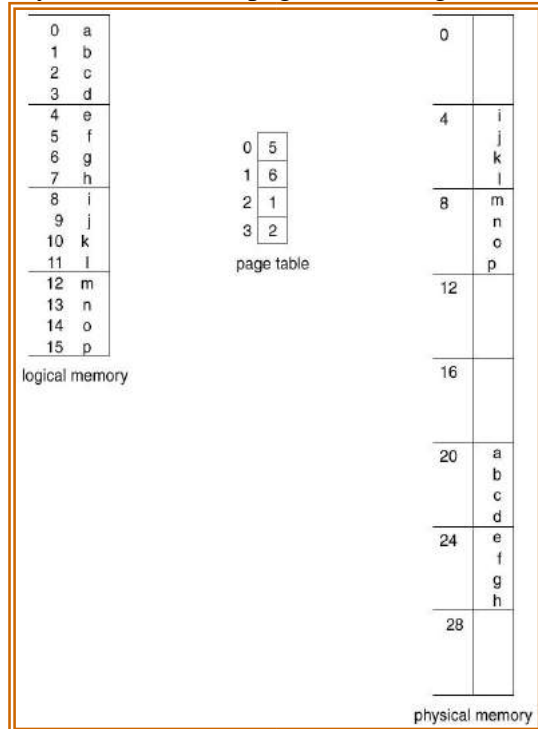
Let us take an example. Suppose a program needs 32 KB memory for allocation. The whole program is divided into smaller units assuming 4 KB and is assigned some address. The address consists of two parts such as:

- A large number in higher order positions and
- Displacement or offset in the lower order bits.

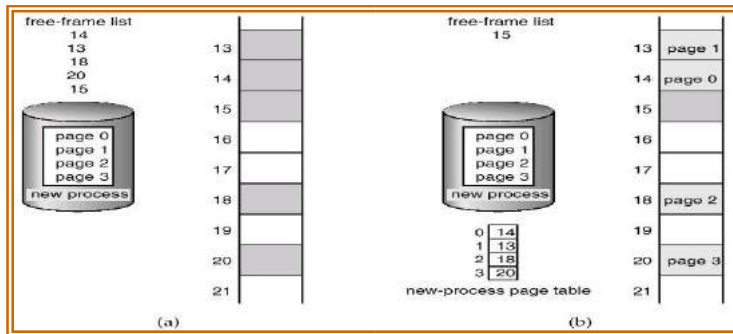
The numbers allocated to pages are typically in power of 2 to simplify extraction of page numbers and offsets. To access a piece of data at a given address, the system first extracts the

page number and the offset. Then it translates the page number to physical page frame and access data at offset in physical page frame. At this moment, the translation of the address by the OS is done using a page table. Page table is a linear array indexed by virtual page number which provides the physical page frame that contains the particular page. It employs a lookup process that extracts the page number and the offset. The system in addition checks that the page number is within the address space of process and retrieves the page number in the page table. Physical address will be calculated by using the formula.

Physical address = page size of logical memory X frame number + offset

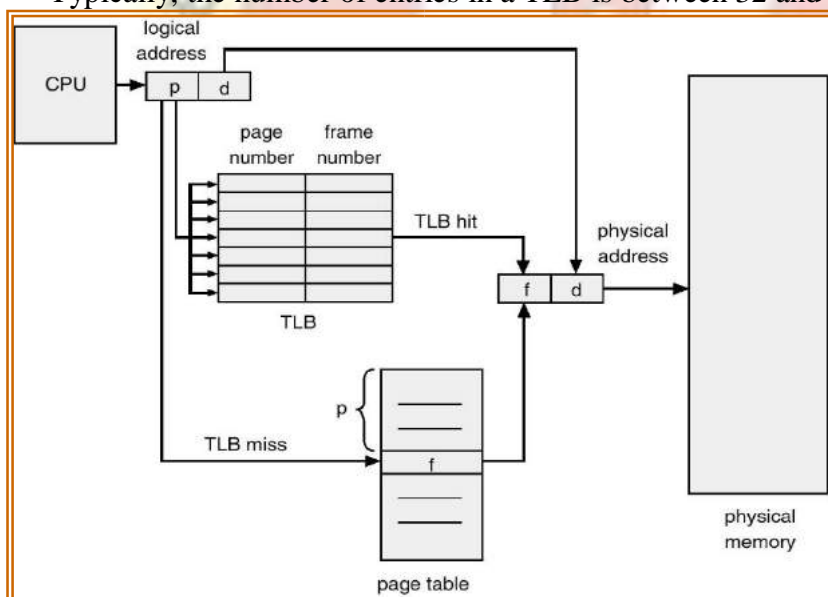


When a process arrives in the system to be executed, its size expressed in pages is examined. Each page of the process needs one frame. Thus if the process requires n pages, at least n frames must be available in memory. If n frames are available, they are allocated to this arriving process. The first page of the process is loaded into one of the allocated frames, and the frame number is put in the page table for this process. The next page is loaded into another frame, and its frame number is put into the page table and so on as in below figure. An important aspect of paging is the clear separation between the user's view of memory and the actual physical memory. The user program views that memory as one single contiguous space, containing only this one program. In fact, the user program is scattered throughout physical memory, which also holds other programs. The difference between the user's view of memory and the actual physical memory is reconciled by the address-translation hardware. The logical addresses are translated into physical addresses. This mapping is hidden from the user and is controlled by the operating system.



Implementation of Page Table

- Page table is kept in main memory.
- Page-tablebase register (PTBR) points to the page table.
- In this scheme every data/instruction-byte access requires two memory accesses. One for the page-table entry and one for the byte.
- The two memory access problem can be solved by the use of a special fast-lookup hardware cache called associative registers or associative memory or translation look-aside buffers (TLBs).
- Typically, the number of entries in a TLB is between 32 and 1024.



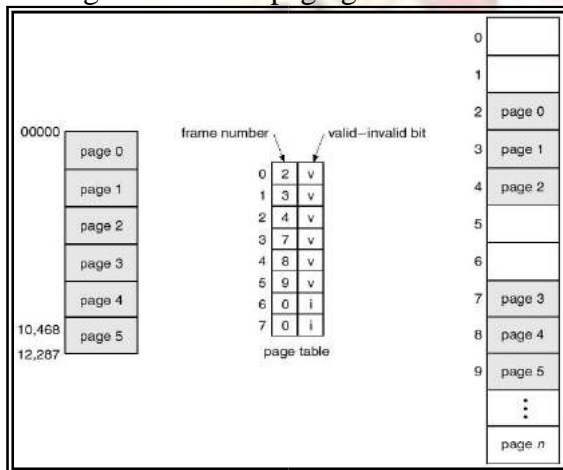
- The TLB contains only a few of the page table entries. When a logical address is generated by the CPU, its page number is presented to the TLB. If the page number is found, its frame number is immediately available and is used to access memory. The whole task may take less than 10 percent longer than it would if an unmapped memory reference were used.
- If the page number is not in the TLB (known as a TLB miss), a memory reference to the page table must be made. When the frame number is obtained, we can use it to access memory. **Hit Ratio**
- Hit Ratio: the percentage of times that a page number is found in the associative registers.
- For example, if it takes 20 nanoseconds to search the associative memory and 100 nanoseconds to access memory; for a 98-percent hit ratio, we have

$$\text{Effective memory-access time} = 0.98 \times 120 + 0.02 \times 220 = 122 \text{ nanoseconds.}$$

- The Intel 80486 CPU has 32 associative registers, and claims a 98-percent hit ratio.

Valid or invalid bit in a page table

- Memory protection implemented by associating protection bit with each frame.
- Valid-invalid bit attached to each entry in the page table:
 - “Valid” indicates that the associated page is in the process’ logical address space, and is thus a legal page.
 - “Invalid” indicates that the page is not in the process’ logical address space.
- Pay attention to the following figure. The program extends to only address 10,468, any reference beyond that address is illegal. However, references to page 5 are classified as valid, so accesses to addresses up to 12,287 are valid. This reflects the internal fragmentation of paging.



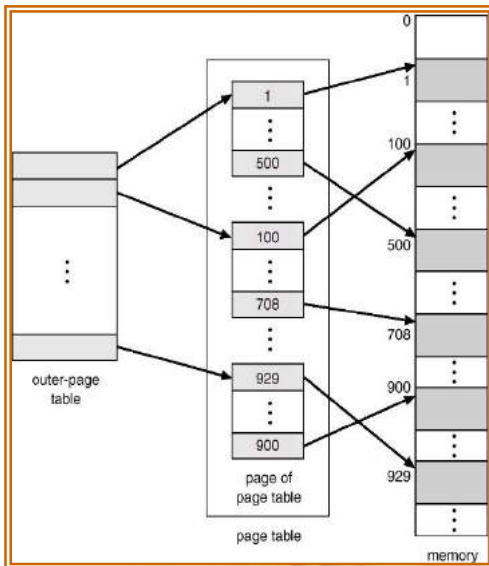
Structure of the Page Table

Hierarchical Paging:

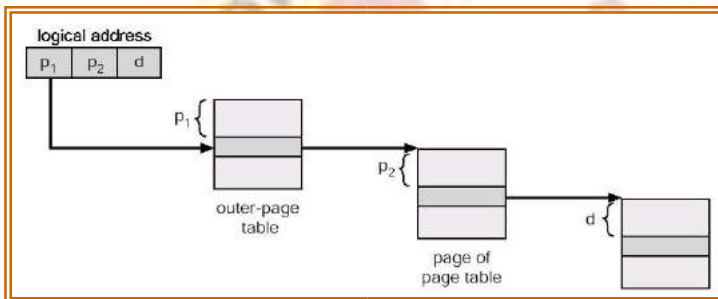
- A logical address (on 32-bit machine with 4K page size) is divided into:
 - A page number consisting of 20 bits.
 - A page offset consisting of 12 bits.
- Since the page table is paged, the page number is further divided into:
 - A 10-bit page number.
 - A 10-bit page offset.
- Thus, a logical address is as follows:

page number		page offset
p_1	p_2	d
10	10	12

Where p_1 is an index into the outer page table, and p_2 is the displacement within the page of the outer page table. The below figure shows a two level page table scheme.

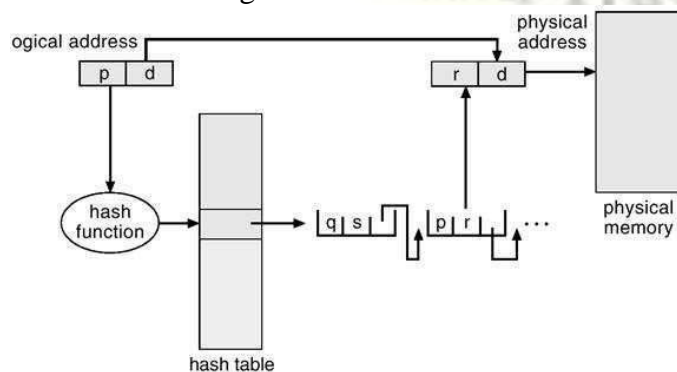


Address-translation scheme for a two-level 32-bit paging architecture is shown in below figure.



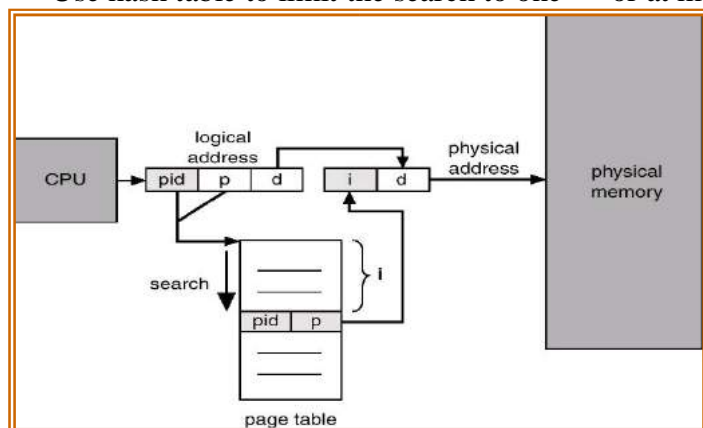
Hashed Page Table:

A common approach for handling address spaces larger than 32 bits is to use a hashed page table, with the hash value being the virtual page number. Each entry in the hash table contains a linked list of elements that has to the same location. Each element consists of three fields: (a) the virtual page number, (b) the value of the mapped page frame, and (c) a pointer to the next element in the linked list. The algorithm works as follows: The virtual page number in the virtual address is hashed into the hash table. The virtual page number is compared to field (a) in the first element in the linked list. If there is a match, the corresponding page frame (field (b)) is used to form the desired physical address. If there is no match, subsequent entries in the linked list are searched for a matching virtual page number. The scheme is shown in below figure.



Inverted Page Table:

- One entry for each real page (frame) of memory.
- Entry consists of the virtual address of the page stored in that real memory location, with information about the process that owns that page.
- There is only one page table in the system. Not per process.
- Decreases memory needed to store each page table, but increases time needed to search the table when a page reference occurs.
- Use hash table to limit the search to one — or at most a few — page-table entries.

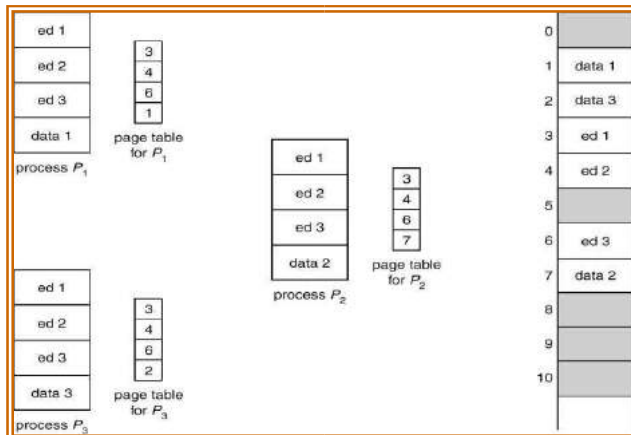


Each virtual address in the system consists of a triple $\langle \text{process-id, page-number, offset} \rangle$. Each inverted page table entry is a pair $\langle \text{process-id, page-number} \rangle$ where the process-id assumes the role of the address space identifier. When a memory reference occurs, part of the virtual address, consisting of $\langle \text{process-id, page-number} \rangle$, is presented to the memory subsystem. The inverted page table is then searched for a match. If a match is found say at entry i , then the physical address $\langle i, \text{offset} \rangle$ is generated. If no match is found, then an illegal address access has been attempted.

SHARED PAGE:

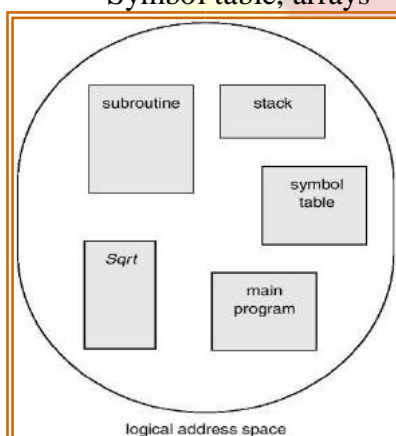
- Shared code
 - One copy of read-only (reentrant) code shared among processes (i.e., text editors, compilers, window systems).
 - Shared code must appear in same location in the logical address space of all processes.
- Private code and data
 - Each process keeps a separate copy of the code and data.
 - The pages for the private code and data can appear anywhere in the logical address space.

Reentrant code or pure code is non self modifying code. If the code is reentrant, then it never changes during execution. Thus, two or more processes can execute the same code at the same time. Each process has its own copy of registers and data storage to hold the data for the process' execution. The data for two different processes will of course vary for each process.



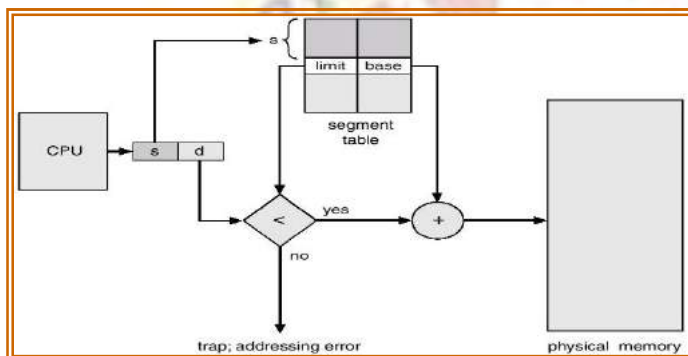
Segmentation

- Memory-management scheme that supports user view of memory.
- A program is a collection of segments. A segment is a logical unit such as:
 - Main program,
 - Procedure,
 - Function,
 - Method,
 - Object,
 - Local variables, global variables,
 - Common block,
 - Stack,
 - Symbol table, arrays

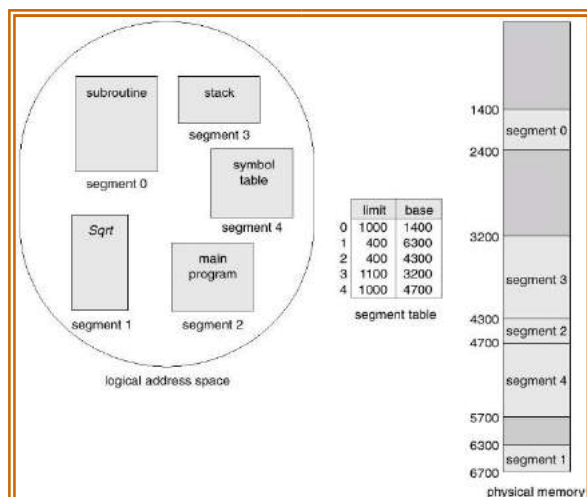


- Segmentation is a memory management scheme that supports this user view of memory.
- A logical address space is a collection of segments. Each segment has a name and a length.
- The addresses specify both the segment name and the offset within the segment.

- The user therefore specifies each address by two quantities such as segment name and an offset. For simplicity of implementation, segments are numbered and are referred to by a segment number, rather than by a segment name.
- Logical address consists of a two tuples:
 $\langle \text{segment-number, offset} \rangle$
- Segment table – maps two-dimensional physical addresses; each table entry has:
 - Base – contains the starting physical address where the segments reside in memory.
 - Limit – specifies the length of the segment.
- Segment-table base register (STBR) points to the segment table's location in memory.
- Segment-table length register (STLR) indicates number of segments used by a program; Segment number s is legal if $s < \text{STLR}$.



- When the user program is compiled by the compiler it constructs the segments.
- The loader takes all the segments and assigned the segment numbers.
- The mapping between the logical and physical address using the segmentation technique is shown in above figure.
- Each entry in the segment table as limit and base address.
- The base address contains the starting physical address of a segment where the limit address specifies the length of the segment.
- The logical address consists of 2 parts such as segment number and offset.
- The segment number is used as an index into the segment table. Consider the below example is given below.



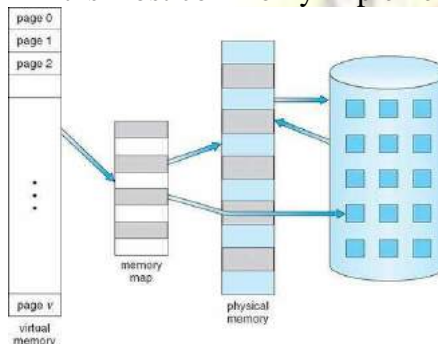
Segmentation with Paging

- Both paging and segmentation have advantages and disadvantages, that's why we can combine these two methods to improve this technique for memory allocation.
- These combinations are best illustrated by architecture of Intel-386.
- The IBM OS/2 is an operating system of the Intel-386 architecture. In this technique both segment table and page table is required.
- The program consists of various segments given by the segment table where the segment table contains different entries one for each segment.
- Then each segment is divided into a number of pages of equal size whose information is maintained in a separate page table.
- If a process has four segments that is 0 to 3 then there will be 4 page tables for that process, one for each segment.
- The size fixed in segmentation table (SMT) gives the total number of pages and therefore maximum page number in that segment with starting from 0.
- If the page table or page map table for a segment has entries for page 0 to 5.
- The address of the entry in the PMT for the desired page p in a given segment s can be obtained by $B + P$ where B can be obtained from the entry in the segmentation table.
- Using the address $(B + P)$ as an index in page map table (page table), the page frame (f) can be obtained and physical address can be obtained by adding offset to page frame.

UNIT IV

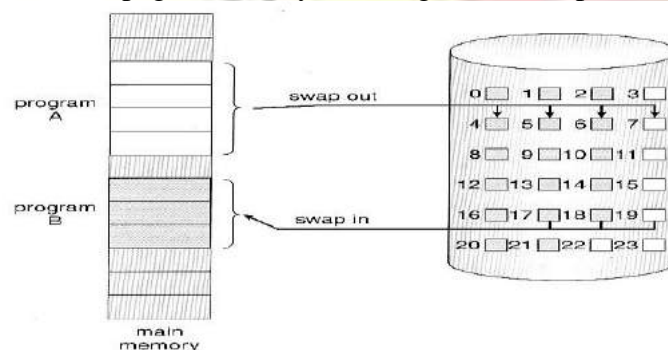
VIRTUAL MEMORY

- It is a technique which allows execution of process that may not be compiled within the primary memory.
- It separates the user logical memory from the physical memory. This separation allows an extremely large memory to be provided for program when only a small physical memory is available.
- Virtual memory makes the task of programming much easier because the programmer no longer needs to working about the amount of the physical memory is available or not.
- The virtual memory allows files and memory to be shared by different processes by page sharing.
- It is most commonly implemented by demand paging.



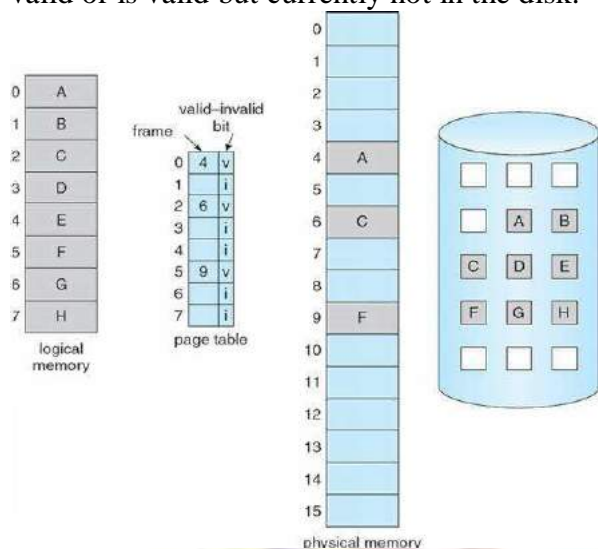
DEMAND PAGING

A demand paging system is similar to the paging system with swapping feature. When we want to execute a process we swap it into the memory. A swapper manipulates entire process where as a pager is concerned with the individual pages of a process. The demand paging concept is using pager rather than swapper. When a process is to be swapped in, the pager guesses which pages will be used before the process is swapped out again. Instead of swapping in a whole process, the pager brings only those necessary pages into memory. The transfer of a paged memory to contiguous disk space is shown in below figure.



Thus it avoids reading into memory pages that will not be used in any way, decreasing the swap time and the amount of physical memory needed. In this technique, we need some hardware support to distinguish between the pages that are in memory and those that are on the disk. A valid and invalid bit is used for this purpose. When this bit is set to valid, it indicates that the

associate page is in memory. If the bit is set to invalid it indicates that the page is either not valid or is valid but currently not in the disk.



Marking a page invalid will have no effect if the process never attempts to access that page. So while a process executes and access pages that are memory resident, execution proceeds normally. Access to a page marked invalid causes a page fault trap. It is the result of the OS's failure to bring the desired page into memory.

PROCEDURE TO HANDLE PAGE FAULT

If a process refers to a page that is not in physical memory then

- We check an internal table (page table) for this process to determine whether the reference was valid or invalid.
- If the reference was invalid, we terminate the process, if it was valid but not yet brought in, we have to bring that from main memory.
- Now we find a free frame in memory.
- Then we read the desired page into the newly allocated frame.
- When the disk read is complete, we modify the internal table to indicate that the page is now in memory.
- We restart the instruction that was interrupted by the illegal address trap. Now the process can access the page as if it had always been in memory.

PAGE REPLACEMENT

- Each process is allocated frames (memory) which hold the process's pages (data)
- Frames are filled with pages as needed – this is called demand paging
- Over-allocation of memory is prevented by modifying the page-fault service routine to replace pages
- The job of the page replacement algorithm is to decide which page gets victimized to make room for a new page
- Page replacement completes separation of logical and physical memory

PAGE REPLACEMENT ALGORITHM

Optimal algorithm

- Ideally we want to select an algorithm with the lowest page-fault rate
- Such an algorithm exists, and is called (unsurprisingly) the optimal algorithm:
- Procedure: replace the page that will not be used for the longest time (or at all) – i.e. replace the page with the greatest forward distance in the reference string
- Example using 4 frames:

Reference #	1	2	3	4	5	6	7	8	9	10	11	12
Page referenced	1	2	3	4	1	2	5	1	2	3	4	5
Frames	<u>1</u>	1	1	1	1	1	1	1	1	1	<u>4</u>	4
_ = faulting page		<u>2</u>	2	2	2	2	2	2	2	2	2	2
			<u>3</u>	3	3	3	3	3	3	3	3	3
				<u>4</u>	4	4	<u>5</u>	5	5	5	5	5

- Analysis: 12 page references, 6 page faults, 2 page replacements. Page faults per number of frames = $6/4 = 1.5$
- Unfortunately, the optimal algorithm requires special hardware (crystal ball, magic mirror, etc.) not typically found on today's computers
- Optimal algorithm is still used as a metric for judging other page replacement algorithms

FIFO algorithm

- Replaces pages based on their order of arrival: oldest page is replaced
- Example using 4 frames:

Reference #	1	2	3	4	5	6	7	8	9	10	11	12
Page referenced	1	2	3	4	1	2	5	1	2	3	4	5
Frames	<u>1</u>	1	1	1	1	1	<u>5</u>	5	5	5	<u>4</u>	4
_ = faulting page		<u>2</u>	2	2	2	2	2	<u>1</u>	1	1	1	<u>5</u>
			<u>3</u>	3	3	3	3	3	<u>2</u>	2	2	2
				<u>4</u>	4	4	4	4	4	<u>3</u>	3	3

- Analysis: 12 page references, 10 page faults, 6 page replacements. Page faults per number of frames = $10/4 = 2.5$

n = reference count		<u>1</u>	1	1	1	2	2	2	3	3	3	3
		2	2	2	2	2	2	2	2	2	2	2
			<u>1</u> <u>3</u>	1 3	1 3	1 3	1 <u>5</u>	1 5	1 5	1 <u>3</u>	1 3	1 <u>5</u>
			<u>1</u> <u>4</u>	1 4	1 4	1 4	1 4	1 4	1 4	2 4	2 4	

- At the 7th reference, we victimize the page in the frame which has been referenced least often -- in this case, pages 3 and 4 (contained within frames 3 and 4) are candidates, each with a reference count of 1. Let's pick the page in frame 3. Page 5 is paged in and frame 3's reference count is reset to 1.
- At the 10th reference, we again have a page fault. Pages 5 and 4 (contained within frames 3 and 4) are candidates, each with a count of 1. Let's pick page 4. Page 3 is paged into frame 3, and frame 3's reference count is reset to 1.
- Analysis: 12 page references, 7 page faults, 3 page replacements. Page faults per number of frames = $7/4 = 1.75$

LRU algorithm

- Replaces pages based on their most recent reference – replace the page with the greatest backward distance in the reference string
- Example using 4 frames:

Reference #	1	2	3	4	5	6	7	8	9	10	11	12
Page referenced	1	2	3	4	1	2	5	1	2	3	4	5
Frames	<u>1</u>	1	1	1	1	1	1	1	1	1	1	<u>5</u>
_ = faulting page		<u>2</u>	2	2	2	2	2	2	2	2	2	2
			<u>3</u>	3	3	3	<u>5</u>	5	5	5	<u>4</u>	4
				<u>4</u>	4	4	4	4	4	<u>3</u>	3	3

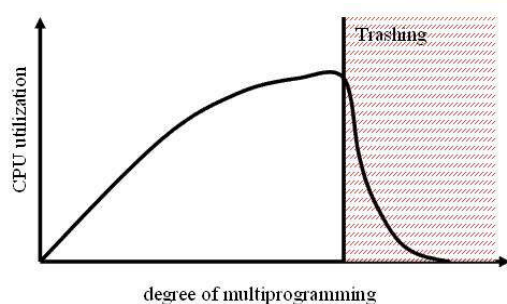
Analysis: 12 page references, 8 page faults, 4 page replacements. Page faults per number of frames = $8/4 = 2$

- One possible implementation (not necessarily the best):
 - Every frame has a time field; every time a page is referenced, copy the current time into its frame's time field
 - When a page needs to be replaced, look at the time stamps to find the oldest

THRASHING

- If a process does not have “enough” pages, the page-fault rate is very high
 - low CPU utilization

- OS thinks it needs increased multiprogramming
 - adds another process to system
- Thrashing is when a process is busy swapping pages in and out
 - Thrashing results in severe performance problems. Consider the following scenario, which is based on the actual behaviour of early paging systems. The operating system monitors CPU utilization. If CPU utilization is too low, we increase the degree of multiprogramming by introducing a new process to the system. A global page replacement algorithm is used; it replaces pages with no regard to the process to which they belong. Now suppose that a process enters a new phase in its execution and needs more frames.



FILE SYSTEM

File concept:

A file is a collection of related information that is stored on secondary storage. Information stored in files must be persistent i.e. not affected by power failures & system reboots. Files may be of free form such as text files or may be formatted rigidly. Files represent both programs as well as data.

Part of the OS dealing with the files is known as file system. The important file concepts include:

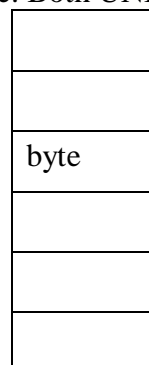
1. **File attributes:** A file has certain attributes which vary from one operating system to another.
 - **Name:** Every file has a name by which it is referred.
 - **Identifier:** It is unique number that identifies the file within the file system.
 - **Type:** This information is needed for those systems that support different types of files.
 - **Location:** It is a pointer to a device & to the location of the file on that device
 - **Size:** It is the current size of a file in bytes, words or blocks.
 - **Protection:** It is the access control information that determines who can read, write & execute a file.
 - **Time, date & user identification:** It gives information about time of creation or last modification & last use.
2. **File operations:** The operating system can provide system calls to create, read, write, reposition, delete and truncate files.

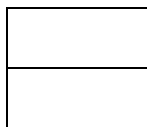
- **Creating files:** Two steps are necessary to create a file. First, space must be found for the file in the file system. Secondly, an entry must be made in the directory for the new file.
 - **Reading a file:** Data & read from the file at the current position. The system must keep a read pointer to know the location in the file from where the next read is to take place. Once the read has been taken place, the read pointer is updated.
 - **Writing a file:** Data are written to the file at the current position. The system must keep a write pointer to know the location in the file where the next write is to take place. The write pointer must be updated whenever a write occurs.
 - **Repositioning within a file (seek):** The directory is searched for the appropriate entry & the current file position is set to a given value. After repositioning data can be read from or written into that position.
 - **Deleting a file:** To delete a file, we search the directory for the required file. After deletion, the space is released so that it can be reused by other files.
 - **Truncating a file:** The user may erase the contents of a file but allows all attributes to remain unchanged except the file length which is reset to '0' & the space is released.
3. **File types:** The file name is split into 2 parts, Name & extension. Usually these two parts are separated by a period. The user & the OS can know the type of the file from the extension itself.

Listed below are some file types along with their extension:

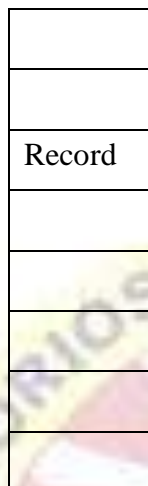
File Type	Extension
Executable File	exe, bin, com
Object File	obj, o (compiled)
Source Code file	C, C++, Java, pas
Batch File	bat, sh (commands to command the interpreter)
Text File	txt, doc (textual data documents)
Archive File	arc, zip, tar (related files grouped together into file compressed for storage)
Multimedia File	mpeg (Binary file containing audio or A/V information)

4. **File structure:** Files can be structured in several ways. Three common possible are:
- **Byte sequence:** The figure shows an unstructured sequence of bytes. The OS doesn't care about the content of file. It only sees the bytes. This structure provides maximum flexibility. Users can write anything into their files & name them according to their convenience. Both UNIX & windows use this approach.





- **Record sequence:** In this structure, a file is a sequence of fixed length records. Here the read operation returns one records & the write operation overwrites or append or record.



- **Tree:** In this organization, a file consists of a tree of records of varying lengths. Each record consists of a key field. The tree is stored on the key field to allow first searching for a particular key.

ACCESS METHODS: Basically, access method is divided into 2 types:

- **Sequential access:** It is the simplest access method. Information in the file is processed in order i.e. one record after another. A process can read all the data in a file in order starting from beginning but can't skip & read arbitrarily from any location. Sequential files can be rewind. It is convenient when storage medium was magnetic tape rather than disk.
- **Direct access:** A file is made up of fixed length-logical records that allow programs to read & write records rapidly in no particular O order. This method can be used when disk are used for storing files. This method is used in many applications e.g. database systems. If an airline customer wants to reserve a seat on a particular flight, the reservation program must be able to access the record for that flight directly without reading the records before it. In a direct access file, there is no restriction in the order of reading or writing. For example, we can read block 14, then read block 50 & then write block 7 etc. Direct access files are very useful for immediate access to large amount of information.

Directory structure: The file system of computers can be extensive. Some systems store thousands of file on disk. To manage all these data, we need to organize them. The organization is done in 2 steps. The file system is broken into partitions. Each partition contains information about file within it.

Operation on a directory:

- **Search for a file:** We need to be able to search a directory for a particular file.
- **Create a file:** New files are created & added to the directory.
- **Delete a file:** When a file is no longer needed, we may remove it from the directory.

- **List a directory:** We should be able to list the files of the directory.
- **Rename a file:** The name of a file is changed when the contents of the file changes.
- **Traverse the file system:** It is useful to be able to access every directory & every file within a directory.

Structure of a directory: The most common schemes for defining the structure of the directory are:

1. **Single level directory:** It is the simplest directory structure. All files are present in the same directory. So it is easy to manage & understand.

Limitation: A single level directory is difficult to manage when the no. of files increases or when there is more than one user. Since all files are in same directory, they must have unique names. So, there is confusion of file names between different users.

2. **Two level directories:** The solution to the name collision problem in single level directory is to create a separate directory for each user. In a two level directory structure, each user has its own user file directory. When a user logs in, then master file directory is searched. It is indexed by user name & each entry points to the UFD of that user.

Limitation: It solves name collision problem. But it isolates one user from another. It is an advantage when users are completely independent. But it is a disadvantage when the users need to access each other's files & co-operate among themselves on a particular task.

3. **Tree structured directories:** It is the most common directory structure. A two level directory is a two level tree. So, the generalization is to extend the directory structure to a tree of arbitrary height. It allows users to create their own subdirectories & organize their files. Every file in the system has a unique path name. It is the path from the root through all the sub-directories to a specified file. A directory is simply another file but it is treated in a special way. One bit in each directory entry defines the entry as a file (O) or as sub-directories. Each user has a current directory. It contains most of the files that are of current interest to the user. Path names can be of two types: An absolute path name begins from the root directory & follows the path down to the specified files. A relative path name defines the path from the current directory. E.g. If the current directory is root/spell/mail, then the relative path name is prt/first & the absolute path name is root/spell/mail/prt/first. Here users can access the files of other users also by specifying their path names.

4. **A cyclic graph directory:** It is a generalization of tree structured directory scheme. An a cyclic graph allows directories to have shared sub-directories & files. A shared directory or file is not the same as two copies of a file. Here a programmer can view the copy but the changes made in the file by one programmer are not reflected in the other's copy. But in a shared file, there is only one actual file. So many changes made by a person would be immediately visible to others. This scheme is useful in a situation where several people are working as a team. So, here all the files that are to be shared are put together in one directory. Shared files and sub-directories can be implemented in several ways. A common way used in UNIX systems is to create a new directory entry called link. It is a pointer to another file or sub-directory. The other approach is to duplicate all information in both sharing directories. A cyclic graph structure is more flexible than a tree structure but it is also more complex.

Limitation: Now a file may have multiple absolute path names. So, distinct file names may refer to the same file. Another problem occurs during deletion of a shared file. When a file is removed by any one user. It may leave dangling pointer to the non existing file. One serious problem in a cyclic graph structure is ensuring that there are no cycles. To avoid these problems, some systems do not allow shared directories or files. E.g. MS-

DOS uses a tree structure rather than a cyclic to avoid the problems associated with deletion. One approach for deletion is to preserve the file until all references to it are deleted. To implement this approach, we must have some mechanism for determining the last reference to the file. For this we have to keep a list of reference to a file. But due to the large size of the no. of references. When the count is zero, the file can be deleted.

5. **General graph directory:** When links are added to an existing tree structured directory, the tree structure is destroyed, resulting in a simple graph structure. Linking is a technique that allows a file to appear in more than one directory. The advantage is the simplicity of algorithm to transverse the graph & determines when there are no more references to a file. But a similar

problem exists when we are trying to determine when a file can be deleted. Here also a value zero in the reference count means that there are no more references to the file or directory & the file can be deleted. But when cycle exists, the reference count may be non-zero even when there are no references to the directory or file. This occurs due to the possibility of self referencing (cycle) in the structure. So, here we have to use garbage collection scheme to determine when the last references to a file has been deleted & the space can be reallocated. It involves two steps:

- Transverse the entire file system & mark everything that can be accessed.
- Everything that isn't marked is added to the list of free space.

But this process is extremely time consuming. It is only necessary due to presence of cycles in the graph. So, a cyclic graph structure is easier to work than this.

PROTECTION

When information is kept in a computer system, a major concern is its protection from physical damage (reliability) as well as improper access.

Types of access: In case of systems that don't permit access to the files of other users. Protection is not needed. So, one extreme is to provide protection by prohibiting access. The other extreme is to provide free access with no protection. Both these approaches are too extreme for general use. So, we need controlled access. It is provided by limiting the types of file access. Access is permitted depending on several factors. One major factor is type of access requested. The different type of operations that can be controlled are:

- **Read**
- **Write**
- **Execute**
- **Append**
- **Delete**
- **List**

Access lists and groups:

Various users may need different types of access to a file or directory. So, we can associate an access lists with each file and directory to implement identity dependent access. When a user access requests access to a particular file, the OS checks the access list associated with that file. If that user is granted the requested access, then the access is allowed. Otherwise, a protection violation occurs & the user is denied access to the file. But the main problem with access lists is their length. It is very tedious to construct such a list. So, we use a condensed version of the access list by classifying the users into 3 categories:

- **Owners:** The user who created the file.

- **Group:** A set of users who are sharing the files.
- **Others:** All other users in the system.

Here only 3 fields are required to define protection. Each field is a collection of bits each of which either allows or prevents the access. E.g. The UNIX file system defines 3 fields of 3 bits each: rwx • r(read access)

- w(write access)
- x(execute access)

Separate fields are kept for file owners, group & other users. So, a bit is needed to record protection information for each file.

Allocation methods

There are 3 methods of allocating disk space widely used.

1. Contiguous allocation:

- It requires each file to occupy a set of contiguous blocks on the disk.
- Number of disk seeks required for accessing contiguously allocated file is minimum.
- The IBM VM/CMS OS uses contiguous allocation. Contiguous allocation of a file is defined by the disk address and length (in terms of block units).
- If the file is 'n' blocks long and starts at location 'b', then it occupies blocks b, b+1, b+2,----- -b+ n-1.
- The directory for each file indicates the address of the starting block and the length of the area allocated for each file.
- Contiguous allocation supports both sequential and direct access. For sequential access, the file system remembers the disk address of the last block referenced and reads the next block when necessary.
- For direct access to block i of a file that starts at block b we can immediately access block b + i.
- Problems:** One difficulty with contiguous allocation is finding space for a new file. It also suffers from the problem of external fragmentation. As files are deleted and allocated, the free disk space is broken into small pieces. A major problem in contiguous allocation is how much space is needed for a file. When a file is created, the total amount of space it will need must be found and allocated. Even if the total amount of space needed for a file is known in advance, pre-allocation is inefficient. Because a file that grows very slowly must be allocated enough space for its final size even though most of that space is left unused for a long period time. Therefore, the file has a large amount of internal fragmentation.

2. Linked Allocation:

- Linked allocation solves all problems of contiguous allocation.
- In linked allocation, each file is linked list of disk blocks, which are scattered throughout the disk.
- The directory contains a pointer to the first and last blocks of the file.
- Each block contains a pointer to the next block.
- These pointers are not accessible to the user. To create a new file, we simply create a new entry in the directory.
- For writing to the file, a free block is found by the free space management system and this new block is written to & linked to the end of the file.
- To read a file, we read blocks by following the pointers from block to block.

- h. There is no external fragmentation with linked allocation & any free block can be used to satisfy a request.
 - i. Also there is no need to declare the size of a file when that file is created. A file can continue to grow as long as there are free blocks.
 - j. **Limitations:** It can be used effectively only for sequential access files. To find the 'i' th block of the file, we must start at the beginning of that file and follow the pointers until we get the ith block. So it is inefficient to support direct access files. Due to the presence of pointers each file requires slightly more space than before. Another problem is reliability. Since the files are linked together by pointers scattered throughout the disk. What would happen if a pointer were lost or damaged.
3. **Indexed Allocation:**
- a. Indexed allocation solves the problem of linked allocation by bringing all the pointers together to one location known as the index block.
 - b. Each file has its own index block which is an array of disk block addresses. The ith entry in the index block points to the ith block of the file.
 - c. The directory contains the address of the index block. To read the ith block, we use the pointer in the ith index block entry and read the desired block.
 - d. To write into the ith block, a free block is obtained from the free space manager and its address is put in the ith index block entry.
 - e. Indexed allocation supports direct access without suffering external fragmentation.
 - f. **Limitations:** The pointer overhead of index block is greater than the pointer overhead of linked allocation. So here more space is wasted than linked allocation. In indexed allocation, an entire index block must be allocated, even if most of the pointers are nil.

FREE SPACE MANAGEMENT

Since there is only a limited amount of disk space, it is necessary to reuse the space from the deleted files. To keep track of free disk space, the system maintains a free space list. It records all the disk blocks that are free i.e. not allocated to some file or dictionary. To create a file, we search the free space list for the required amount of space and allocate it to the new file. This space is then removed from the free space list. When a file is deleted, its disk space is added to the free space list.

Implementation:

There are 4 ways to implement the free space list such as:

- **Bit Vector:** The free space list is implemented as a bit map or bit vector. Each block is represented as 1 bit. If the block is free, the bit is 1 and if it is allocated then the bit is 0. For example, consider a disk where blocks 2, 3, 4, 5, 8, 9, 10, 11, 12, 13, 17, 18, 25, 26 & 27 are free and rest of the blocks are allocated. The free space bit map would be 0011110011111100011000000111.....

The main advantage of this approach is that it is simple and efficient to find the first free block or n consecutive free blocks on the disk. But bit vectors are inefficient unless the entire vector is kept in main memory. It is possible for smaller disks but not for larger ones.

- **Linked List:** Another approach is to link together all the free disk blocks and keep a pointer to the first free block. The first free block contains a pointer to the next free block and so on. For example, we keep a pointer to block 2 as the free block. Block 2 contains a

pointer to block which points to block 4 which then points to block 5 and so on. But this scheme is not efficient.

To traverse the list, we must read each block which require a lot of I/O time.

- **Grouping:** In this approach, we store the address of n free blocks in the first free block. The first $n-1$ of these blocks is actually free. The last block contains the address of another n free blocks and so on. Here the addresses of a large number of free blocks can be found out quickly.
- **Counting:** Rather than keeping a list of n free disk block addresses, we can keep the address of the first free block and the number of free contiguous blocks. So here each entry in the free space list consists of a disk address and a count.



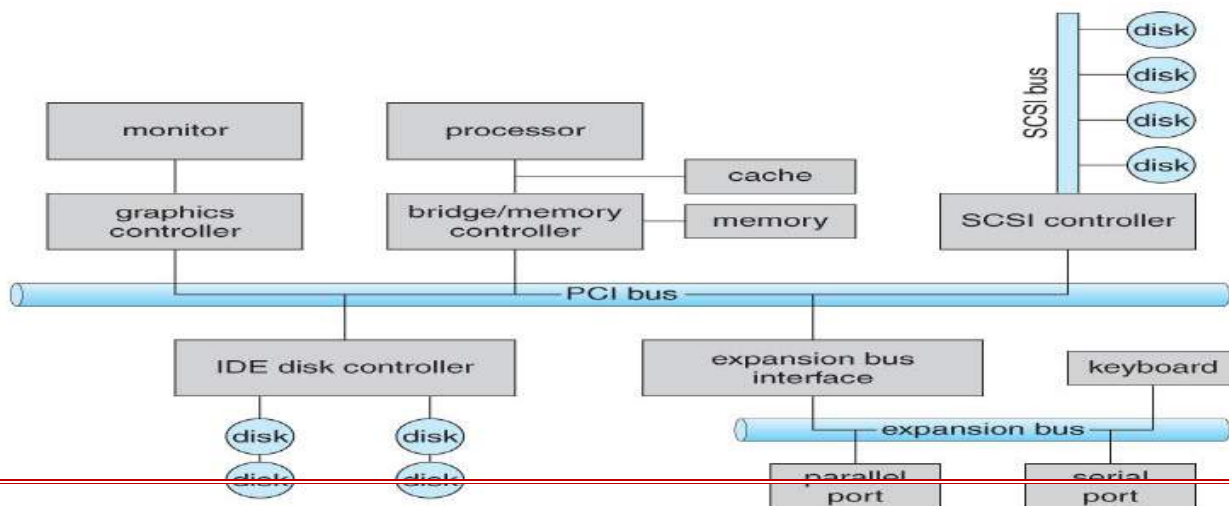
UNIT V

I/O SYSTEMS OVERVIEW

- Management of I/O devices is a very important part of the operating system - so important and so varied that entire I/O subsystems are devoted to its operation. (Consider the range of devices on a modern computer, from mice, keyboards, disk drives, display adapters, USB devices, network connections, audio I/O, printers, special devices for the handicapped, and many special-purpose peripherals.)
- I/O Subsystems must contend with two (conflicting?) trends: (1) The gravitation towards standard interfaces for a wide range of devices, making it easier to add newly developed devices to existing systems, and (2) the development of entirely new types of devices, for which the existing standard interfaces are not always easy to apply.
- *Device drivers* are modules that can be plugged into an OS to handle a particular device or category of similar devices.

I/O HARDWARE

- I/O devices can be roughly categorized as storage, communications, user-interface, and other
- Devices communicate with the computer via signals sent over wires or through the air.
- Devices connect with the computer via *ports*, e.g. a serial or parallel port.
- A common set of wires connecting multiple devices is termed a *bus*.
 - Buses include rigid protocols for the types of messages that can be sent across the bus and the procedures for resolving contention issues.
 - Figure 13.1 below illustrates three of the four bus types commonly found in a modern PC:
 1. The *PCI bus* connects high-speed high-bandwidth devices to the memory subsystem (and the CPU.)
 2. The *expansion bus* connects slower low-bandwidth devices, which typically deliver data one character at a time (with buffering.)
 3. The *SCSI bus* connects a number of SCSI devices to a common SCSI controller.
 4. A *daisy-chain bus*, (not shown) is when a string of devices is connected to each other like beads on a chain, and only one of the devices is directly connected to the host.



A typical PC bus structure.

- One way of communicating with devices is through *registers* associated with each port. Registers may be one to four bytes in size, and may typically include (a subset of) the following four:
 1. The *data-in register* is read by the host to get input from the device.
 2. The *data-out register* is written by the host to send output.
 3. The *status register* has bits read by the host to ascertain the status of the device, such as idle, ready for input, busy, error, transaction complete, etc.
 4. The *control register* has bits written by the host to issue commands or to change settings of the device such as parity checking, word length, or full- versus half-duplex operation.
- Figure 13.2 shows some of the most common I/O port address ranges.

I/O address range (hexadecimal)	device
000–00F	DMA controller
020–021	interrupt controller
040–043	timer
200–20F	game controller
2F8–2FF	serial port (secondary)
320–32F	hard-disk controller
378–37F	parallel port
3D0–3DF	graphics controller
3F0–3F7	diskette-drive controller
3F8–3FF	serial port (primary)

Device I/O port locations on PCs (partial).

- Another technique for communicating with devices is *memory-mapped I/O*.
 - In this case a certain portion of the processor's address space is mapped to the device, and communications occur by reading and writing directly to/from those memory areas.
 - Memory-mapped I/O is suitable for devices which must move large quantities of data quickly, such as graphics cards.
 - Memory-mapped I/O can be used either instead of or more often in combination with traditional registers. For example, graphics cards still use registers for control information such as setting the video mode.
 - A potential problem exists with memory-mapped I/O, if a process is allowed to write directly to the address space used by a memory-mapped I/O device.
 - (Note: Memory-mapped I/O is not the same thing as direct memory access, DMA. See section 13.2.3 below.)

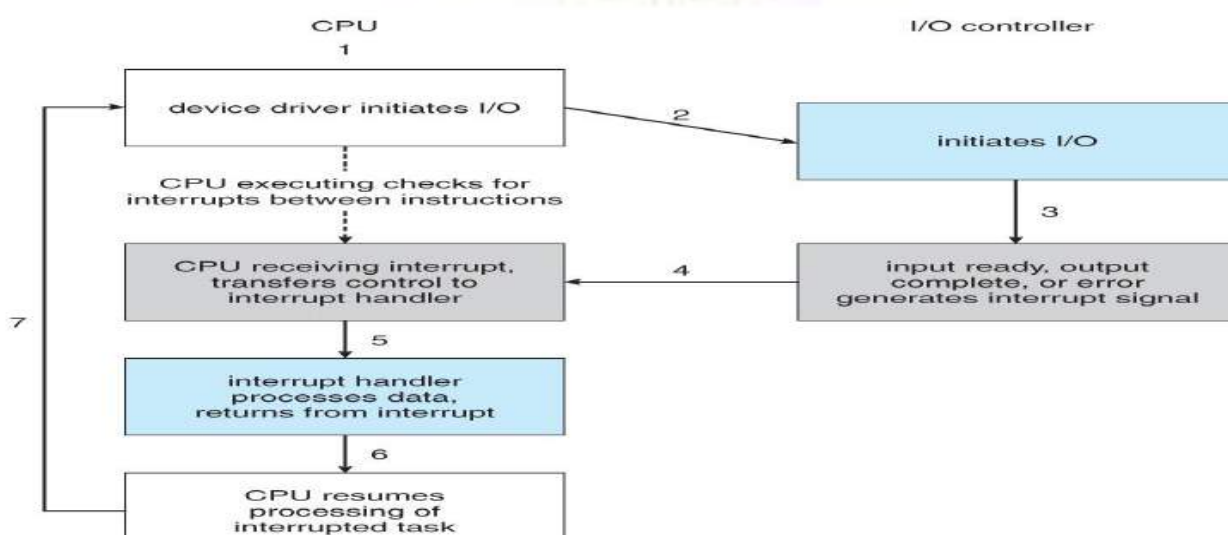
POLLING

- One simple means of device *handshaking* involves polling:

1. The host repeatedly checks the **busy bit** on the device until it becomes clear.
 2. The host writes a byte of data into the data-out register, and sets the **write bit** in the command register (in either order.)
 3. The host sets the **command ready bit** in the command register to notify the device of the pending command.
 4. When the device controller sees the command-ready bit set, it first sets the busy bit.
 5. Then the device controller reads the command register, sees the write bit set, reads the byte of data from the data-out register, and outputs the byte of data.
 6. The device controller then clears the **error bit** in the status register, the command-ready bit, and finally clears the busy bit, signaling the completion of the operation.
- Polling can be very fast and efficient, if both the device and the controller are fast and if there is significant data to transfer. It becomes inefficient, however, if the host must wait a long time in the busy loop waiting for the device, or if frequent checks need to be made for data that is infrequently there.

INTERRUPTS

- Interrupts allow devices to notify the CPU when they have data to transfer or when an operation is complete, allowing the CPU to perform other duties when no I/O transfers need its immediate attention.
- The CPU has an **interrupt-request line** that is sensed after every instruction.
 - A device's controller **raises** an interrupt by asserting a signal on the interrupt request line.
 - The CPU then performs a state save, and transfers control to the **interrupt handler** routine at a fixed address in memory. (The CPU **catches** the interrupt and **dispatches** the interrupt handler.)
 - The interrupt handler determines the cause of the interrupt, performs the necessary processing, performs a state restore, and executes a **return from interrupt** instruction to return control to the CPU. (The interrupt handler **clears** the interrupt by servicing the device.)
 - (Note that the state restored does not need to be the same state as the one that was saved when the interrupt went off. See below for an example involving time-slicing.)
- Figure illustrates the interrupt-driven I/O procedure:



INTERRUPT-DRIVEN I/O CYCLE.

- The above description is adequate for simple interrupt-driven I/O, but there are three needs in modern computing which complicate the picture:
 1. The need to defer interrupt handling during critical processing,
 2. The need to determine *which* interrupt handler to invoke, without having to poll all devices to see which one needs attention, and
 3. The need for multi-level interrupts, so the system can differentiate between high- and low-priority interrupts for proper response.
- These issues are handled in modern computer architectures with *interrupt-controller* hardware.
 - Most CPUs now have two interrupt-request lines: One that is *non-maskable* for critical error conditions and one that is *maskable*, that the CPU can temporarily ignore during critical processing.
 - The interrupt mechanism accepts an *address*, which is usually one of a small set of numbers for an offset into a table called the *interrupt vector*. This table (usually located at physical address zero ?) holds the addresses of routines prepared to process specific interrupts.
 - The number of possible interrupt handlers still exceeds the range of defined interrupt numbers, so multiple handlers can be *interrupt chained*. Effectively the addresses held in the interrupt vectors are the head pointers for linked-lists of interrupt handlers.
 - Figure 13.4 shows the Intel Pentium interrupt vector. Interrupts 0 to 31 are non-maskable and reserved for serious hardware and other errors. Maskable interrupts, including normal device I/O interrupts begin at interrupt 32.
 - Modern interrupt hardware also supports *interrupt priority levels*, allowing systems to mask off only lower-priority interrupts while servicing a high-priority interrupt, or conversely to allow a high-priority signal to interrupt the processing of a low-priority one.

vector number	description
0	divide error
1	debug exception
2	null interrupt
3	breakpoint
4	INTO-detected overflow
5	bound range exception
6	invalid opcode
7	device not available
8	double fault
9	coprocessor segment overrun (reserved)
10	invalid task state segment
11	segment not present
12	stack fault
13	general protection
14	page fault
15	(Intel reserved, do not use)
16	floating-point error
17	alignment check
18	machine check
19–31	(Intel reserved, do not use)
32–255	maskable interrupts

Intel Pentium processor event-vector table.

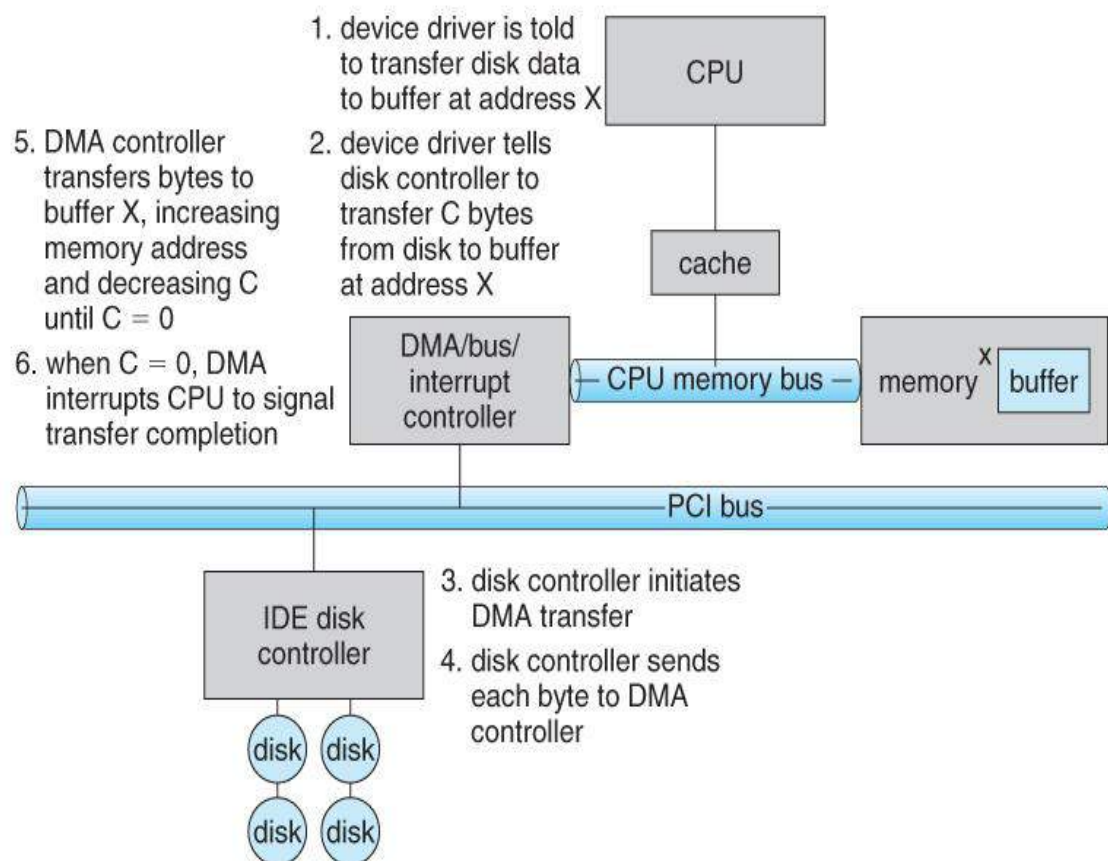
- At boot time the system determines which devices are present, and loads the appropriate handler addresses into the interrupt table.

- During operation, devices signal errors or the completion of commands via interrupts.
- Exceptions, such as dividing by zero, invalid memory accesses, or attempts to access kernel mode instructions can be signaled via interrupts.
- Time slicing and context switches can also be implemented using the interrupt mechanism.
 - The scheduler sets a hardware timer before transferring control over to a user process.
 - When the timer raises the interrupt request line, the CPU performs a state-save, and transfers control over to the proper interrupt handler, which in turn runs the scheduler.
 - The scheduler does a state-restore of a *different* process before resetting the timer and issuing the return-from-interrupt instruction.
- A similar example involves the paging system for virtual memory - A page fault causes an interrupt, which in turn issues an I/O request and a context switch as described above, moving the interrupted process into the wait queue and selecting a different process to run. When the I/O request has completed (i.e. when the requested page has been loaded up into physical memory), then the device interrupts, and the interrupt handler moves the process from the wait queue into the ready queue, (or depending on scheduling algorithms and policies, may go ahead and context switch it back onto the CPU.)
- System calls are implemented via *software interrupts*, a.k.a. *traps*. When a (library) program needs work performed in kernel mode, it sets command information and possibly data addresses in certain registers, and then raises a software interrupt. (E.g. 21 hex in DOS.) The system does a state save and then calls on the proper interrupt handler to process the request in kernel mode. Software interrupts generally have low priority, as they are not as urgent as devices with limited buffering space.
- Interrupts are also used to control kernel operations, and to schedule activities for optimal performance. For example, the completion of a disk read operation involves **two** interrupts:
 - A high-priority interrupt acknowledges the device completion, and issues the next disk request so that the hardware does not sit idle.
 - A lower-priority interrupt transfers the data from the kernel memory space to the user space, and then transfers the process from the waiting queue to the ready queue.
- The Solaris OS uses a multi-threaded kernel and priority threads to assign different threads to different interrupt handlers. This allows for the "simultaneous" handling of multiple interrupts, and the assurance that high-priority interrupts will take precedence over low-priority ones and over user processes.

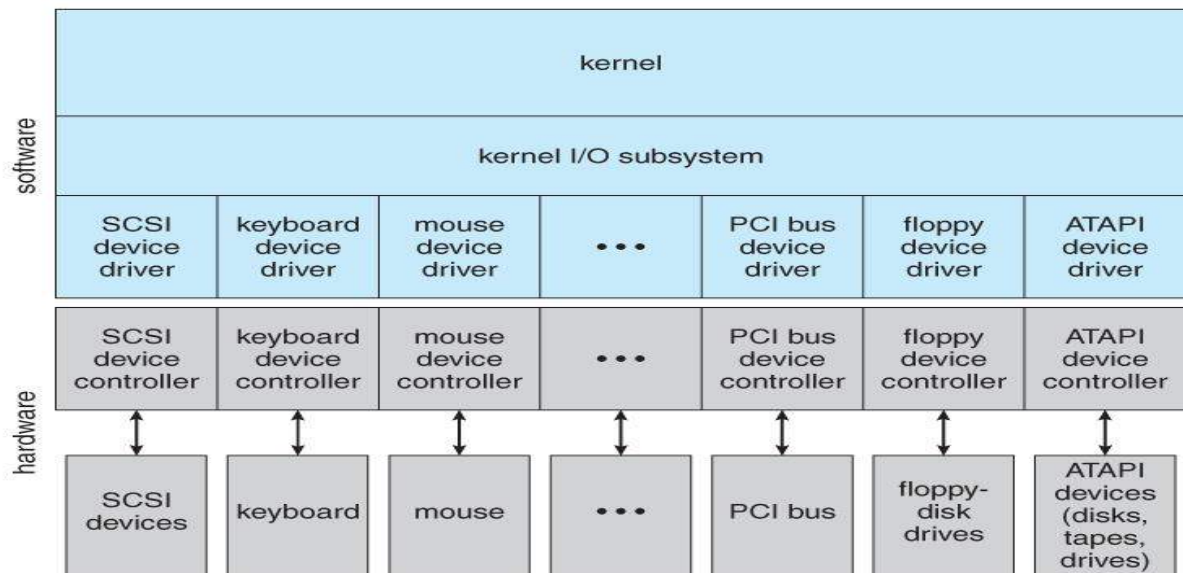
DIRECT MEMORY ACCESS

- For devices that transfer large quantities of data (such as disk controllers), it is wasteful to tie up the CPU transferring data in and out of registers one byte at a time.
- Instead this work can be off-loaded to a special processor, known as the *Direct Memory Access, DMA, Controller*.

- The host issues a command to the DMA controller, indicating the location where the data is located, the location where the data is to be transferred to, and the number of bytes of data to transfer. The DMA controller handles the data transfer, and then interrupts the CPU when the transfer is complete.
- A simple DMA controller is a standard component in modern PCs, and many *bus-mastering* I/O cards contain their own DMA hardware.
- Handshaking between DMA controllers and their devices is accomplished through two wires called the DMA-request and DMA-acknowledge wires.
- While the DMA transfer is going on the CPU does not have access to the PCI bus (including main memory), but it does have access to its internal registers and primary and secondary caches.
- DMA can be done in terms of either physical addresses or virtual addresses that are mapped to physical addresses. The latter approach is known as *Direct Virtual Memory Access, DVMA*, and allows direct data transfer from one memory-mapped device to another without using the main memory chips.
- Direct DMA access by user processes can speed up operations, but is generally forbidden by modern systems for security and protection reasons. (I.e. DMA is a kernel-mode operation.)
- Figure 13.5 below illustrates the DMA process.



APPLICATION I/O INTERFACE User application access to a wide variety of different devices is accomplished through layering, and through encapsulating all of the device-specific code into *device drivers*, while application layers are presented with a common interface for all (or at least large general categories of) devices.



A kernel I/O structure.

- Devices differ on many different dimensions, as outlined in Figure 13.7:

aspect	variation	example
data-transfer mode	character block	terminal disk
access method	sequential random	modem CD-ROM
transfer schedule	synchronous asynchronous	tape keyboard
sharing	dedicated sharable	tape keyboard
device speed	latency seek time transfer rate delay between operations	
I/O direction	read only write only read-write	CD-ROM graphics controller disk

Characteristics of I/O devices.

- Most devices can be characterized as either block I/O, character I/O, memory mapped file access, or network sockets. A few devices are special, such as time-of-day clock and the system timer.
- Most OSes also have an *escape*, or *back door*, which allows applications to send commands directly to device drivers if needed. In UNIX this is the *ioctl()* system call (I/O Control). *ioctl()* takes three arguments - The file descriptor for the device driver being accessed, an integer indicating the desired function to be performed, and an address used for communicating or transferring additional information.

Block and Character Devices

- **Block devices** are accessed a block at a time, and are indicated by a "b" as the first character in a long listing on UNIX systems. Operations supported include *read()*, *write()*, and *seek()*.
 - Accessing blocks on a hard drive directly (without going through the filesystem structure) is called *raw I/O*, and can speed up certain operations by bypassing the buffering and locking normally conducted by the OS. (It then becomes the application's responsibility to manage those issues.)
 - A new alternative is *direct I/O*, which uses the normal filesystem access, but which disables buffering and locking operations.
- Memory-mapped file I/O can be layered on top of block-device drivers.
 - Rather than reading in the entire file, it is mapped to a range of memory addresses, and then paged into memory as needed using the virtual memory system.
 - Access to the file is then accomplished through normal memory accesses, rather than through *read()* and *write()* system calls. This approach is commonly used for executable program code.
- **Character devices** are accessed one byte at a time, and are indicated by a "c" in UNIX long listings. Supported operations include *get()* and *put()*, with more advanced functionality such as reading an entire line supported by higher-level library routines.

Network Devices

- Because network access is inherently different from local disk access, most systems provide a separate interface for network devices.
- One common and popular interface is the *socket* interface, which acts like a cable or pipeline connecting two networked entities. Data can be put into the socket at one end, and read out sequentially at the other end. Sockets are normally full-duplex, allowing for bi-directional data transfer.
- The *select()* system call allows servers (or other applications) to identify sockets which have data waiting, without having to poll all available sockets.

Clocks and Timers

- Three types of time services are commonly needed in modern systems:

- Get the current time of day.
- Get the elapsed time (system or wall clock) since a previous event.
- Set a timer to trigger event X at time T.
- Unfortunately time operations are not standard across all systems.
- A **programmable interrupt timer, PIT** can be used to trigger operations and to measure elapsed time. It can be set to trigger an interrupt at a specific future time, or to trigger interrupts periodically on a regular basis.
 - The scheduler uses a PIT to trigger interrupts for ending time slices.
 - The disk system may use a PIT to schedule periodic maintenance cleanup, such as flushing buffers to disk.
 - Networks use PIT to abort or repeat operations that are taking too long to complete. I.e. resending packets if an acknowledgement is not received before the timer goes off.
 - More timers than actually exist can be simulated by maintaining an ordered list of timer events, and setting the physical timer to go off when the next scheduled event should occur.
- On most systems the system clock is implemented by counting interrupts generated by the PIT. Unfortunately this is limited in its resolution to the interrupt frequency of the PIT, and may be subject to some drift over time. An alternate approach is to provide direct access to a high frequency hardware counter, which provides much higher resolution and accuracy, but which does not support interrupts.

Blocking and Non-blocking I/O

- With **blocking I/O** a process is moved to the wait queue when an I/O request is made, and moved back to the ready queue when the request completes, allowing other processes to run in the meantime.
- With **non-blocking I/O** the I/O request returns immediately, whether the requested I/O operation has (completely) occurred or not. This allows the process to check for available data without getting hung completely if it is not there.
- One approach for programmers to implement non-blocking I/O is to have a multi-threaded application, in which one thread makes blocking I/O calls (say to read a keyboard or mouse), while other threads continue to update the screen or perform other tasks.
- A subtle variation of the non-blocking I/O is the **asynchronous I/O**, in which the I/O request returns immediately allowing the process to continue on with other tasks, and then the process is notified (via changing a process variable, or a software interrupt, or a callback function) when the I/O operation has completed and the data is available for use. (The regular non-blocking I/O returns immediately with whatever results are available, but does not complete the operation and notify the process later.)

Kernel I/O Subsystem

I/O Scheduling

- Scheduling I/O requests can greatly improve overall efficiency. Priorities can also play a part in request scheduling.
- The classic example is the scheduling of disk accesses, as discussed in detail in chapter 12.
- Buffering and caching can also help, and can allow for more flexible scheduling options.
- On systems with many devices, separate request queues are often kept for each device:

Buffering

- Buffering of I/O is performed for (at least) 3 major reasons:
 1. Speed differences between two devices. (See Figure 13.10 below.) A slow device may write data into a buffer, and when the buffer is full, the entire buffer is sent to the fast device all at once. So that the slow device still has somewhere to write while this is going on, a second buffer is used, and the two buffers alternate as each becomes full. This is known as **double buffering**. (Double buffering is often used in (animated) graphics, so that one screen image can be generated in a buffer while the other (completed) buffer is displayed on the screen. This prevents the user from ever seeing any half-finished screen images.)
 2. Data transfer size differences. Buffers are used in particular in networking systems to break messages up into smaller packets for transfer, and then for re-assembly at the receiving side.
 3. To support **copy semantics**. For example, when an application makes a request for a disk write, the data is copied from the user's memory area into a kernel buffer. Now the application can change their copy of the data, but the data which eventually gets written out to disk is the version of the data at the time the write request was made.

Caching

- Caching involves keeping a **copy** of data in a faster-access location than where the data is normally stored.
- Buffering and caching are very similar, except that a buffer may hold the only copy of a given data item, whereas a cache is just a duplicate copy of some other data stored elsewhere.
- Buffering and caching go hand-in-hand, and often the same storage space may be used for both purposes. For example, after a buffer is written to disk, then the copy in memory can be used as a cached copy, (until that buffer is needed for other purposes.)

Spooling and Device Reservation

- A **spool** (*Simultaneous Peripheral Operations On-Line*) buffers data for (peripheral) devices such as printers that cannot support interleaved data streams.

- If multiple processes want to print at the same time, they each send their print data to files stored in the spool directory. When each file is closed, then the application sees that print job as complete, and the print scheduler sends each file to the appropriate printer one at a time.
- Support is provided for viewing the spool queues, removing jobs from the queues, moving jobs from one queue to another queue, and in some cases changing the priorities of jobs in the queues.
- Spool queues can be general (any laser printer) or specific (printer number 42.)
- Oses can also provide support for processes to request / get exclusive access to a particular device, and/or to wait until a device becomes available.

Error Handling

- I/O requests can fail for many reasons, either transient (buffers overflow) or permanent (disk crash).
- I/O requests usually return an error bit (or more) indicating the problem. UNIX systems also set the global variable *errno* to one of a hundred or so well-defined values to indicate the specific error that has occurred. (See *errno.h* for a complete listing, or *man errno*.)
- Some devices, such as SCSI devices, are capable of providing much more detailed information about errors, and even keep an on-board error log that can be requested by the host.

I/O Protection

- The I/O system must protect against either accidental or deliberate erroneous I/O.
- User applications are not allowed to perform I/O in user mode - All I/O requests are handled through system calls that must be performed in kernel mode.
- Memory mapped areas and I/O ports must be protected by the memory management system, **but** access to these areas cannot be totally denied to user programs. (Video games and some other applications need to be able to write directly to video memory for optimal performance for example.) Instead the memory protection system restricts access so that only one process at a time can access particular parts of memory, such as the portion of the screen memory corresponding to a particular window.

Kernel Data Structures

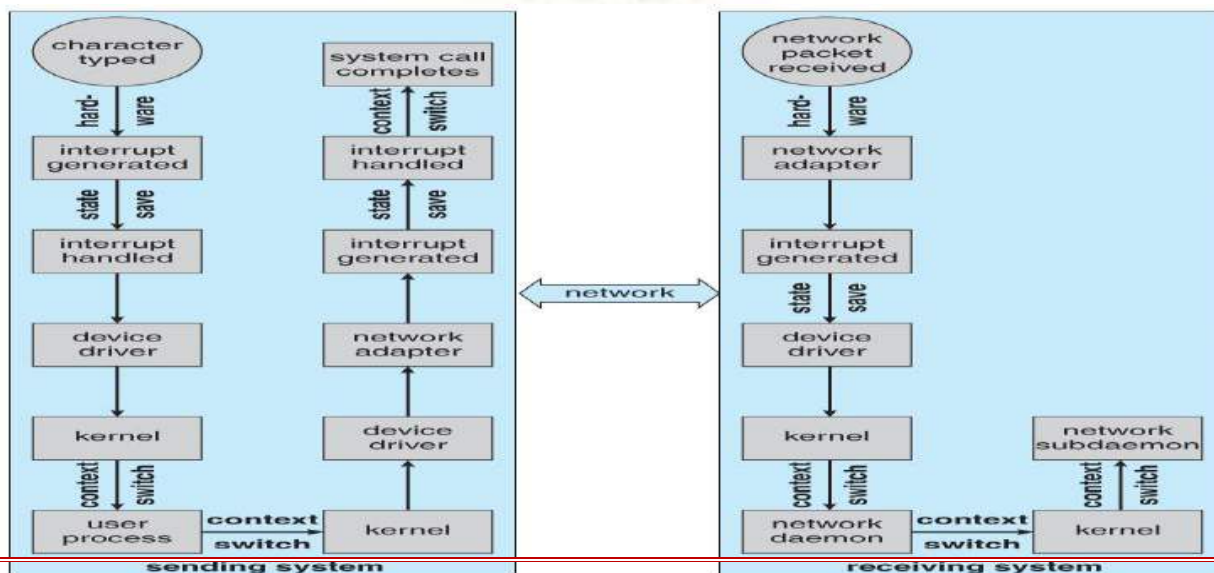
- The kernel maintains a number of important data structures pertaining to the I/O system, such as the open file table.
- These structures are object-oriented, and flexible to allow access to a wide variety of I/O devices through a common interface. (See Figure 13.12 below.)
- Windows NT carries the object-orientation one step further, implementing I/O as a message-passing system from the source through various intermediaries to the device.

Transforming I/O Requests to Hardware Operations

- Users request data using file names, which must ultimately be mapped to specific blocks of data from a specific device managed by a specific device driver.
- DOS uses the colon separator to specify a particular device (e.g. C:, LPT:, etc.)
- UNIX uses a *mount table* to map filename prefixes (e.g. /usr) to specific mounted devices. Where multiple entries in the mount table match different prefixes of the filename the one that matches the longest prefix is chosen. (e.g. /usr/home instead of /usr where both exist in the mount table and both match the desired file.)
- UNIX uses special *device files*, usually located in /dev, to represent and access physical devices directly.
 - Each device file has a major and minor number associated with it, stored and displayed where the file size would normally go.
 - The major number is an index into a table of device drivers, and indicates which device driver handles this device. (E.g. the disk drive handler.)
 - The minor number is a parameter passed to the device driver, and indicates which specific device is to be accessed, out of the many which may be handled by a particular device driver. (e.g. a particular disk drive or partition.)
- A series of lookup tables and mappings makes the access of different devices flexible, and somewhat transparent to users.
- Figure 13.13 illustrates the steps taken to process a (blocking) read request:

Performance

- The I/O system is a major factor in overall system performance, and can place heavy loads on other major components of the system (interrupt handling, process switching, memory access, bus contention, and CPU load for device drivers just to name a few.)
- Interrupt handling can be relatively expensive (slow), which causes programmed I/O to be faster than interrupt-driven I/O when the time spent busy waiting is not excessive.



- Network traffic can also put a heavy load on the system. Consider for example the sequence of events that occur when a single character is typed in a telnet session, as shown in figure 13.15. (And the fact that a similar set of events must happen in reverse to echo back the character that was typed.) Sun uses in-kernel threads for the telnet daemon, increasing the supportable number of simultaneous telnet sessions from the hundreds to the thousands.
- Other systems use *front-end processors* to off-load some of the work of I/O processing from the CPU. For example a *terminal concentrator* can multiplex with hundreds of terminals on a single port on a large computer.
- Several principles can be employed to increase the overall efficiency of I/O processing:
 1. Reduce the number of context switches.
 2. Reduce the number of times data must be copied.
 3. Reduce interrupt frequency, using large transfers, buffering, and polling where appropriate.
 4. Increase concurrency using DMA.
 5. Move processing primitives into hardware, allowing their operation to be concurrent with CPU and bus operations.
 6. Balance CPU, memory, bus, and I/O operations, so a bottleneck in one does not idle all the others.
- The development of new I/O algorithms often follows a progression from application level code to on-board hardware implementation, as shown in Figure 13.16. Lower-level implementations are faster and more efficient, but higher-level ones are more flexible and easier to modify. Hardware-level functionality may also be harder for higher-level authorities (e.g. the kernel) to control.

Protection

Goals of Protection

- Obviously to prevent malicious misuse of the system by users or programs. See chapter 15 for a more thorough coverage of this goal.
- To ensure that each shared resource is used only in accordance with system *policies*, which may be set either by system designers or by system administrators.
- To ensure that errant programs cause the minimal amount of damage possible.
- Note that protection systems only provide the *mechanisms* for enforcing policies and ensuring reliable systems. It is up to administrators and users to implement those mechanisms effectively.

Principles of Protection

- The *principle of least privilege* dictates that programs, users, and systems be given just enough privileges to perform their tasks.
- This ensures that failures do the least amount of harm and allow the least of harm to be done.

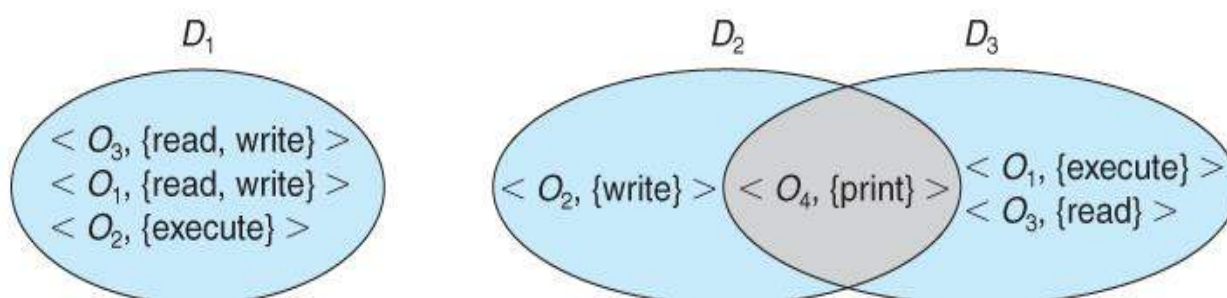
- For example, if a program needs special privileges to perform a task, it is better to make it a SGID program with group ownership of "network" or "backup" or some other pseudo group, rather than SUID with root ownership. This limits the amount of damage that can occur if something goes wrong.
- Typically each user is given their own account, and has only enough privilege to modify their own files.
- The root account should not be used for normal day to day activities - The System Administrator should also have an ordinary account, and reserve use of the root account for only those tasks which need the root privileges

Domain of Protection

- A computer can be viewed as a collection of *processes* and *objects* (both HW & SW).
- The ***need to know principle*** states that a process should only have access to those objects it needs to accomplish its task, and furthermore only in the modes for which it needs access and only during the time frame when it needs access.
- The modes available for a particular object may depend upon its type.

Domain Structure

- A ***protection domain*** specifies the resources that a process may access.
- Each domain defines a set of objects and the types of operations that may be invoked on each object.
- An ***access right*** is the ability to execute an operation on an object.
- A domain is defined as a set of $\langle \text{object}, \{ \text{access right set} \} \rangle$ pairs, as shown below. Note that some domains may be disjoint while others overlap.



System with three protection domains.

- The association between a process and a domain may be *static* or *dynamic*.
 - If the association is static, then the need-to-know principle requires a way of changing the contents of the domain dynamically.
 - If the association is dynamic, then there needs to be a mechanism for ***domain switching***.
- Domains may be realized in different fashions - as users, or as processes, or as procedures. E.g. if each user corresponds to a domain, then that domain

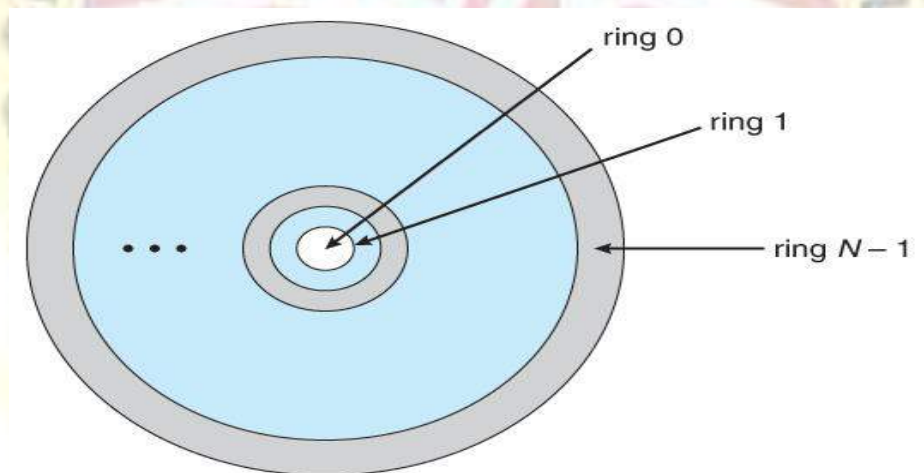
defines the access of that user, and changing domains involves changing user ID.

An Example: UNIX

- UNIX associates domains with users.
- Certain programs operate with the SUID bit set, which effectively changes the user ID, and therefore the access domain, while the program is running. (and similarly for the SGID bit.) Unfortunately this has some potential for abuse.
- An alternative used on some systems is to place privileged programs in special directories, so that they attain the identity of the directory owner when they run. This prevents crackers from placing SUID programs in random directories around the system.
- Yet another alternative is to not allow the changing of ID at all. Instead, special privileged daemons are launched at boot time, and user processes send messages to these daemons when they need special tasks performed.

An Example: MULTICS

- The MULTICS system uses a complex system of rings, each corresponding to a different protection domain, as shown below:



MULTICS ring structure.

- Rings are numbered from 0 to 7, with outer rings having a subset of the privileges of the inner rings.
- Each file is a memory segment, and each segment description includes an entry that indicates the ring number associated with that segment, as well as read, write, and execute privileges.
- Each process runs in a ring, according to the *current-ring-number*, a counter associated with each process.

- A process operating in one ring can only access segments associated with higher (farther out) rings, and then only according to the access bits. Processes cannot access segments associated with lower rings.
- Domain switching is achieved by a process in one ring calling upon a process operating in a lower ring, which is controlled by several factors stored with each segment descriptor:
 - An *access bracket*, defined by integers $b1 \leq b2$.
 - A *limit* $b3 > b2$
 - A *list of gates*, identifying the entry points at which the segments may be called.
- If a process operating in ring i calls a segment whose bracket is such that $b1 \leq i \leq b2$, then the call succeeds and the process remains in ring i .
- Otherwise a trap to the OS occurs, and is handled as follows:
 - If $i < b1$, then the call is allowed, because we are transferring to a procedure with fewer privileges. However if any of the parameters being passed are of segments below $b1$, then they must be copied to an area accessible by the called procedure.
 - If $i > b2$, then the call is allowed only if $i \leq b3$ and the call is directed to one of the entries on the list of gates.
- Overall this approach is more complex and less efficient than other protection schemes.

Access Matrix

- The model of protection that we have been discussing can be viewed as an *access matrix*, in which columns represent different system resources and rows represent different protection domains. Entries within the matrix indicate what access that domain has to that resource.

domain \ object	F_1	F_2	F_3	printer
D_1	read		read	
D_2				print
D_3		read	execute	
D_4	read write		read write	

Access matrix.

- Domain switching can be easily supported under this model, simply by providing "switch" access to other domains:

object \ domain	F_1	F_2	F_3	laser printer	D_1	D_2	D_3	D_4
D_1	read		read			switch		
D_2				print			switch	switch
D_3		read	execute					
D_4	read write		read write		switch			

Access matrix of with domains as objects.

- The ability to *copy* rights is denoted by an asterisk, indicating that processes in that domain have the right to copy that access within the same column, i.e. for the same object. There are two important variations:
 - If the asterisk is removed from the original access right, then the right is *transferred*, rather than being copied. This may be termed a *transfer* right as opposed to a *copy* right.
 - If only the right and not the asterisk is copied, then the access right is added to the new domain, but it may not be propagated further. That is the new domain does not also receive the right to copy the access. This may be termed a *limited copy* right, as shown in Figure 14.5 below:

object \ domain	F_1	F_2	F_3
D_1	execute		write*
D_2	execute	read*	execute
D_3	execute		

(a)

object \ domain	F_1	F_2	F_3
D_1	execute		write*
D_2	execute	read*	execute
D_3	execute	read	

(b)

Access matrix with *copy* rights.

- The *owner* right adds the privilege of adding new rights or removing

existing ones: Copy and owner rights only allow the modification of rights within a column. The addition of **control rights**, which only apply to domain objects, allow a process operating in one domain to affect the rights available in other domains.

domain \ object	F_1	F_2	F_3
D_1	owner execute		write
D_2		read* owner	read* owner write
D_3	execute		

(a)

domain \ object	F_1	F_2	F_3
D_1	owner execute		write
D_2		owner read* write*	read* owner write
D_3		write	write

(b)

Access matrix with *owner* rights.

For example in the table below, a process operating in domain D2 has the right to control any of the rights in domain D4.

domain \ object	F_1	F_2	F_3	laser printer	D_1	D_2	D_3	D_4
D_1	read		read			switch		
D_2				print			switch	switch control
D_3		read	execute					
D_4	write		write		switch			

Modified access matrix

The Security Problem

- Protection dealt with protecting files and other resources from accidental misuse by cooperating users sharing a system, generally using the computer for normal purposes.
- This Security deals with protecting systems from deliberate attacks, either internal or external, from individuals intentionally attempting to steal information, damage information, or otherwise deliberately wreak havoc in some manner.
- Some of the most common types of **violations** include:
 - **Breach of Confidentiality** - Theft of private or confidential information, such as credit-card numbers, trade secrets, patents, secret

formulas, manufacturing procedures, medical information, financial information, etc.

- **Breach of Integrity** - Unauthorized *modification* of data, which may have serious indirect consequences. For example a popular game or other program's source code could be modified to open up security holes on users systems before being released to the public.
- **Breach of Availability** - Unauthorized *destruction* of data, often just for the "fun" of causing havoc and for bragging rites. Vandalism of web sites is a common form of this violation.
- **Theft of Service** - Unauthorized use of resources, such as theft of CPU cycles, installation of daemons running an unauthorized file server, or tapping into the target's telephone or networking services.
- **Denial of Service, DOS** - Preventing legitimate users from using the system, often by overloading and overwhelming the system with an excess of requests for service.
- One common attack is *masquerading*, in which the attacker pretends to be a trusted third party. A variation of this is the *man-in-the-middle*, in which the attacker masquerades as both ends of the conversation to two targets.
- A *replay attack* involves repeating a valid transmission. Sometimes this can be the entire attack, (such as repeating a request for a money transfer), or other times the content of the original message is replaced with malicious content.

There are four levels at which a system must be protected:

1. **Physical** - The easiest way to steal data is to pocket the backup tapes. Also, access to the root console will often give the user special privileges, such as rebooting the system as root from removable media. Even general access to terminals in a computer room offers some opportunities for an attacker, although today's modern high-speed networking environment provides more and more opportunities for remote attacks.
2. **Human** - There is some concern that the humans who are allowed access to a system be trustworthy, and that they cannot be coerced into breaching security. However more and more attacks today are made via *social engineering*, which basically means fooling trustworthy people into accidentally breaching security.
 - **Phishing** involves sending an innocent-looking e-mail or web site designed to fool people into revealing confidential information. E.g. spam e-mails pretending to be from e-Bay, PayPal, or any of a number of banks or credit-card companies.
 - **Dumpster Diving** involves searching the trash or other locations for passwords that are written down. (Note: Passwords that are too hard to remember, or which must be changed frequently are more likely to be written down somewhere close to the user's station.)
 - **Password Cracking** involves divining users passwords, either by watching them type in their passwords, knowing something about them like their pet's names, or simply trying all words in common dictionaries. (Note: "Good" passwords should involve a minimum number of characters, include non-alphabetical

characters, and not appear in any dictionary (in any language), and should be changed frequently. Note also that it is proper etiquette to look away from the keyboard while someone else is entering their password.)

3. **Operating System** - The OS must protect itself from security breaches, such as runaway processes (denial of service), memory-access violations, stack overflow violations, the launching of programs with excessive privileges, and many others.
4. **Network** - As network communications become ever more important and pervasive in modern computing environments, it becomes ever more important to protect this area of the system. (Both protecting the network itself from attack, and protecting the local system from attacks coming in through the network.) This is a growing area of concern as wireless communications and portable devices become more and more prevalent.

Program Threats

- There are many common threats to modern systems. Only a few are discussed here.

Trojan Horse

- A *Trojan Horse* is a program that secretly performs some maliciousness in addition to its visible actions.
- Some Trojan horses are deliberately written as such, and others are the result of legitimate programs that have become infected with *viruses*, (see below.)
- One dangerous opening for Trojan horses is long search paths, and in particular paths which include the current directory (".") as part of the path. If a dangerous program having the same name as a legitimate program (or a common mis-spelling, such as "sl" instead of "ls") is placed anywhere on the path, then an unsuspecting user may be fooled into running the wrong program by mistake.
- Another classic Trojan Horse is a login emulator, which records a users account name and password, issues a "password incorrect" message, and then logs off the system. The user then tries again (with a proper login prompt), logs in successfully, and doesn't realize that their information has been stolen.
- Two solutions to Trojan Horses are to have the system print usage statistics on logouts, and to require the typing of non-trappable key sequences such as Control-Alt-Delete in order to log in. (This is why modern Windows systems require the Control-Alt-Delete sequence to commence logging in, which cannot be emulated or caught by ordinary programs. I.e. that key sequence always transfers control over to the operating system.)
- *Spyware* is a version of a Trojan Horse that is often included in "free" software downloaded off the Internet. Spyware programs generate pop-up browser windows, and may also accumulate information about the user and deliver it to some central site. (This is an example of *covert channels*, in which surreptitious communications occur.) Another common task of

spyware is to send out spam e-mail messages, which then purportedly come from the infected user.

Trap Door

- A *Trap Door* is when a designer or a programmer (or hacker) deliberately inserts a security hole that they can use later to access the system.
- Because of the possibility of trap doors, once a system has been in an untrustworthy state, that system can never be trusted again. Even the backup tapes may contain a copy of some cleverly hidden back door.
- A clever trap door could be inserted into a compiler, so that any programs compiled with that compiler would contain a security hole. This is especially dangerous, because inspection of the code being compiled would not reveal any problems.

Logic Bomb

- A *Logic Bomb* is code that is not designed to cause havoc all the time, but only when a certain set of circumstances occurs, such as when a particular date or time is reached or some other noticeable event.
- A classic example is the *Dead-Man Switch*, which is designed to check whether a certain person (e.g. the author) is logging in every day, and if they don't log in for a long time (presumably because they've been fired), then the logic bomb goes off and either opens up security holes or causes other problems.

Stack and Buffer Overflow

- This is a classic method of attack, which exploits bugs in system code that allows buffers to overflow. Consider what happens in the following code, for example, if argv[1] exceeds 256 characters:
 - The strcpy command will overflow the buffer, overwriting adjacent areas of memory.
 - (The problem could be avoided using strncpy, with a limit of 255 characters copied plus room for the null byte.)

```
#include
#define BUFFER_SIZE 256

int main( int argc, char * argv[ ] )
{
    char buffer[ BUFFER_SIZE ];

    if( argc < 2 )
        return -1;
    else {
        strcpy( buffer, argv[ 1 ] );
        return 0;
    }
}
```

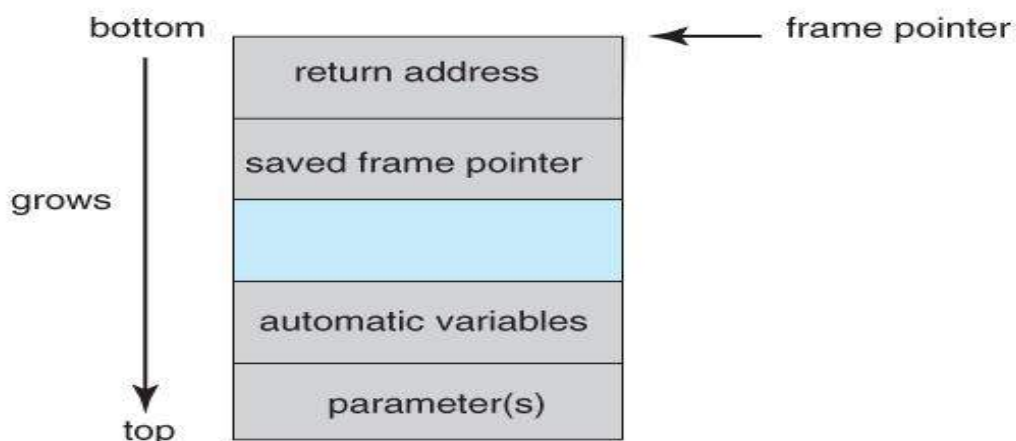
```

}
}

```

C program with buffer-overflow condition.

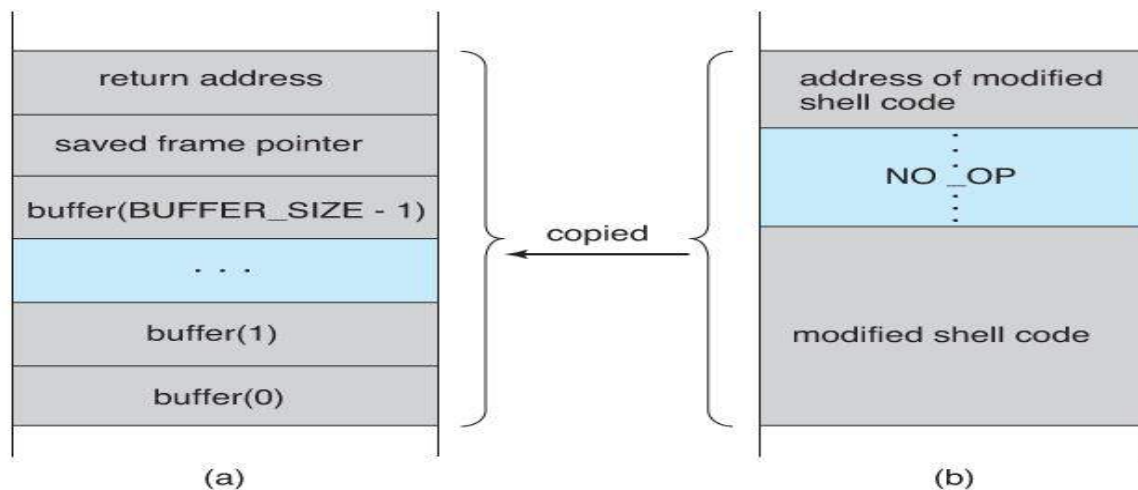
- So how does overflowing the buffer cause a security breach? Well the first step is to understand the structure of the stack in memory:
 - The "bottom" of the stack is actually at a high memory address, and the stack grows towards lower addresses.
 - However the address of an array is the lowest address of the array, and higher array elements extend to higher addresses. (I.e. an array "grows" towards the bottom of the stack.
 - In particular, writing past the top of an array, as occurs when a buffer overflows with too much input data, can eventually overwrite the return address, effectively changing where the program jumps to when it returns.



The layout for a typical stack frame.

- Now that we know how to change where the program returns to by overflowing the buffer, the second step is to insert some nefarious code, and then get the program to jump to our inserted code.
- Our only opportunity to enter code is via the input into the buffer, which means there isn't room for very much. One of the simplest and most obvious approaches is to insert the code for "exec(/bin/sh)". To do this requires compiling a program that contains this instruction, and then using an assembler or debugging tool to extract the minimum extent that includes the necessary instructions.
- The bad code is then padded with as many extra bytes as are needed to overflow the buffer to the correct extent, and the address of the buffer inserted into the return address location. (Note, however, that neither the bad code or the padding can contain null bytes, which would terminate the strcpy.)

- The resulting block of information is provided as "input", copied into the buffer by the original program, and then the return statement causes control to jump to the location of the buffer and start executing the code to launch a shell.



Hypothetical stack frame for (a) before and (b) after.

- Unfortunately famous hacks such as the buffer overflow attack are well published and well known, and it doesn't take a lot of skill to follow the instructions and start attacking lots of systems until the law of averages eventually works out. (*Script Kiddies* are those hackers with only rudimentary skills of their own but the ability to copy the efforts of others.)
- Fortunately modern hardware now includes a bit in the page tables to mark certain pages as non-executable. In this case the buffer-overflow attack would work up to a point, but as soon as it "returns" to an address in the data space and tries executing statements there, an exception would be thrown crashing the program.

Viruses

- A virus is a fragment of code embedded in an otherwise legitimate program, designed to replicate itself (by infecting other programs), and (eventually) wreaking havoc.
- Viruses are more likely to infect PCs than UNIX or other multi-user systems, because programs in the latter systems have limited authority to modify other programs or to access critical system structures (such as the boot block.)
- Viruses are delivered to systems in a *virus dropper*, usually some form of a Trojan Horse, and usually via e-mail or unsafe downloads.
- Viruses take many forms (see below.) Figure 15.5 shows typical operation of a boot sector virus:

Some of the forms of viruses include:

- **File** - A file virus attaches itself to an executable file, causing it to run the virus code first and then jump to the start of the original program. These viruses are termed *parasitic*, because they do not leave any new files on the system, and the original program is still fully functional.
- **Boot** - A boot virus occupies the boot sector, and runs before the OS is loaded. These are also known as *memory viruses*, because in operation they reside in memory, and do not appear in the file system.
- **Macro** - These viruses exist as a macro (script) that are run automatically by certain macro-capable programs such as MS Word or Excel. These viruses can exist in word processing documents or spreadsheet files.
- **Source code** viruses look for source code and infect it in order to spread.
- **Polymorphic** viruses change every time they spread - Not their underlying functionality, but just their *signature*, by which virus checkers recognize them.
- **Encrypted** viruses travel in encrypted form to escape detection. In practice they are self-decrypting, which then allows them to infect other files.
- **Stealth** viruses try to avoid detection by modifying parts of the system that could be used to detect it. For example the read() system call could be modified so that if an infected file is read the infected part gets skipped and the reader would see the original unadulterated file.
- **Tunneling** viruses attempt to avoid detection by inserting themselves into the interrupt handler chain, or into device drivers.
- **Multipartite** viruses attack multiple parts of the system, such as files, boot sector, and memory.
- **Armored** viruses are coded to make them hard for anti-virus researchers to decode and understand. In addition many files associated with viruses are hidden, protected, or given innocuous looking names such as "...".
- In 2004 a virus exploited three bugs in Microsoft products to infect hundreds of Windows servers (including many trusted sites) running Microsoft Internet Information Server, which in turn infected any Microsoft Internet Explorer web browser that visited any of the infected server sites. One of the back-door programs it installed was a *keystroke logger*, which records users keystrokes, including passwords and other sensitive information.
- There is some debate in the computing community as to whether a *monoculture*, in which nearly all systems run the same hardware, operating system, and applications, increases the threat of viruses and the potential for harm caused by them.

System and Network Threats

- Most of the threats described above are termed *program threats*, because they attack specific programs or are carried and distributed in programs. The threats in this section attack the operating system or the network itself, or leverage those systems to launch their attacks.

Worms

- A **worm** is a process that uses the fork / spawn process to make copies of itself in order to wreak havoc on a system. Worms consume system resources, often blocking out other, legitimate processes. Worms that propagate over networks can be especially problematic, as they can tie up vast amounts of network resources and bring down large-scale systems.
- One of the most well-known worms was launched by Robert Morris, a graduate student at Cornell, in November 1988. Targeting Sun and VAX computers running BSD UNIX version 4, the worm spanned the Internet in a matter of a few hours, and consumed enough resources to bring down many systems.
- This worm consisted of two parts:
 1. A small program called a **grappling hook**, which was deposited on the target system through one of three vulnerabilities, and
 2. The main worm program, which was transferred onto the target system and launched by the grappling hook program.

The three vulnerabilities exploited by the Morris Internet worm were as follows:

1. **rsh (remote shell)** is a utility that was in common use at that time for accessing remote systems without having to provide a password. If a user had an account on two different computers (with the same account name on both systems), then the system could be configured to allow that user to remotely connect from one system to the other without having to provide a password. Many systems were configured so that **any** user (except root) on system A could access the same account on system B without providing a password.
 2. **finger** is a utility that allows one to remotely query a user database, to find the true name and other information for a given account name on a given system. For example "finger joeUser@somemachine.edu" would access the finger daemon at somemachine.edu and return information regarding joeUser. Unfortunately the finger daemon (which ran with system privileges) had the buffer overflow problem, so by sending a special 536-character user name the worm was able to fork a shell on the remote system running with root privileges.
 3. **sendmail** is a routine for sending and forwarding mail that also included a debugging option for verifying and testing the system. The debug feature was convenient for administrators, and was often left turned on. The Morris worm exploited the debugger to mail and execute a copy of the grappling hook program on the remote system.
- Once in place, the worm undertook systematic attacks to discover user passwords:
 1. First it would check for accounts for which the account name and the password were the same, such as "guest", "guest".
 2. Then it would try an internal dictionary of 432 favorite password choices. (I'm sure "password", "pass", and blank passwords were all on the list.)
 3. Finally it would try every word in the standard UNIX on-line dictionary to try and break into user accounts.

- Once it had gotten access to one or more user accounts, then it would attempt to use those accounts to rsh to other systems, and continue the process.
- With each new access the worm would check for already running copies of itself, and 6 out of 7 times if it found one it would stop. (The seventh was to prevent the worm from being stopped by fake copies.)
- Fortunately the same rapid network connectivity that allowed the worm to propagate so quickly also quickly led to its demise - Within 24 hours remedies for stopping the worm propagated through the Internet from administrator to administrator, and the worm was quickly shut down.
- There is some debate about whether Mr. Morris's actions were a harmless prank or research project that got out of hand or a deliberate and malicious attack on the Internet. However the court system convicted him, and penalized him heavy fines and court costs.
- There have since been many other worm attacks, including the W32.Sobig.F@mm attack which infected hundreds of thousands of computers and an estimated 1 in 17 e-mails in August 2003. This worm made detection difficult by varying the subject line of the infection-carrying mail message, including "Thank You!", "Your details", and "Re: Approved".

Port Scanning

- **Port Scanning** is technically not an attack, but rather a search for vulnerabilities to attack. The basic idea is to systematically attempt to connect to every known (or common or possible) network port on some remote machine, and to attempt to make contact. Once it is determined that a particular computer is listening to a particular port, then the next step is to determine what daemon is listening, and whether or not it is a version containing a known security flaw that can be exploited.
- Because port scanning is easily detected and traced, it is usually launched from *zombie systems*, i.e. previously hacked systems that are being used without the knowledge or permission of their rightful owner. For this reason it is important to protect "innocuous" systems and accounts as well as those that contain sensitive information or special privileges.

Denial of Service

- **Denial of Service (DOS)** attacks do not attempt to actually access or damage systems, but merely to clog them up so badly that they cannot be used for any useful work. Tight loops that repeatedly request system services are an obvious form of this attack.
- DOS attacks can also involve social engineering, such as the Internet chain letters that say "send this immediately to 10 of your friends, and then go to a certain URL", which clogs up not only the Internet mail system but also the web server to which everyone is directed. (Note: Sending a "reply all" to such a message notifying everyone that it was just a hoax also clogs up the Internet mail service, just as effectively as if you had forwarded the thing.)
- Security systems that lock accounts after a certain number of failed login attempts are subject to DOS attacks which repeatedly attempt logins to all accounts with invalid passwords strictly in order to lock up all accounts.

- Sometimes DOS is not the result of deliberate maliciousness. Consider for example:
 - A web site that sees a huge volume of hits as a result of a successful advertising campaign.
 - CNN.com occasionally gets overwhelmed on big news days, such as Sept 11, 2001.
 - CS students given their first programming assignment involving `fork()` often quickly fill up process tables or otherwise completely consume system resources. :-)
 - (Please use `ipcs` and `ipcrm` when working on the inter-process communications assignment !)

Cryptography as a Security Tool

- Within a given computer the transmittal of messages is safe, reliable and secure, because the OS knows exactly where each one is coming from and where it is going.
- On a network, however, things aren't so straightforward - A rogue computer (or e-mail sender) may spoof their identity, and outgoing packets are delivered to a lot of other computers besides their (intended) final destination, which brings up two big questions of security:
 - **Trust** - How can the system be sure that the messages received are really from the source that they say they are, and can that source be trusted?
 - **Confidentiality** - How can one ensure that the messages one is sending are received only by the intended recipient?
- Cryptography can help with both of these problems, through a system of **secrets** and **keys**. In the former case, the key is held by the sender, so that the recipient knows that only the authentic author could have sent the message; In the latter, the key is held by the recipient, so that only the intended recipient can receive the message accurately.
- Keys are designed so that they cannot be divined from any public information, and must be guarded carefully. (*Asymmetric encryption* involve both a public and a private key.)

Encryption

- The basic idea of encryption is to encode a message so that only the desired recipient can decode and read it. Encryption has been around since before the days of Caesar, and is an entire field of study in itself. Only some of the more significant computer encryption schemes will be covered here.
- The basic process of encryption is shown in Figure 15.7, and will form the basis of most of our discussion on encryption. The steps in the procedure and some of the key terminology are as follows:
 1. The **sender** first creates a **message, m** in plaintext.
 2. The message is then entered into an **encryption algorithm, E**, along with the **encryption key, Ke**.
 3. The encryption algorithm generates the **ciphertext, c**, = **E(Ke)(m)**. For any key k, E(k) is an algorithm for generating ciphertext from a

message, and both E and $E(k)$ should be efficiently computable functions.

4. The ciphertext can then be sent over an unsecure network, where it may be received by **attackers**.
5. The **recipient** enters the ciphertext into a **decryption algorithm, D** , along with the **decryption key, K_d** .
6. The decryption algorithm re-generates the plaintext message, $m = D(K_d)(c)$. For any key k , $D(k)$ is an algorithm for generating a clear text message from a ciphertext, and both D and $D(k)$ should be efficiently computable functions.
7. The algorithms described here must have this important property: Given a ciphertext c , a computer can only compute a message m such that $c = E(k)(m)$ if it possesses $D(k)$. (In other words, the messages can't be decoded unless you have the decryption algorithm and the decryption key.)

Symmetric Encryption

- With *symmetric encryption* the same key is used for both encryption and decryption, and must be safely guarded. There are a number of well-known symmetric encryption algorithms that have been used for computer security:
 - The *Data-Encryption Standard, DES*, developed by the National Institute of Standards, NIST, has been a standard civilian encryption standard for over 20 years. Messages are broken down into 64-bit chunks, each of which are encrypted using a 56-bit key through a series of substitutions and transformations. Some of the transformations are hidden (black boxes), and are classified by the U.S. government.
 - DES is known as a *block cipher*, because it works on blocks of data at a time. Unfortunately this is a vulnerability if the same key is used for an extended amount of data. Therefore an enhancement is to not only encrypt each block, but also to XOR it with the previous block, in a technique known as *cipher-block chaining*.
 - As modern computers become faster and faster, the security of DES has decreased, to where it is now considered insecure because its keys can be exhaustively searched within a reasonable amount of computer time. An enhancement called *triple DES* encrypts the data three times using three separate keys (actually two encryptions and one decryption) for an effective key length of 168 bits. Triple DES is in widespread use today.
 - The *Advanced Encryption Standard, AES*, developed by NIST in 2001 to replace DES uses key lengths of 128, 192, or 256 bits, and encrypts in blocks of 128 bits using 10 to 14 rounds of transformations on a matrix formed from the block.
 - The *twofish algorithm*, uses variable key lengths up to 256 bits and works on 128 bit blocks.

- **RC5** can vary in key length, block size, and the number of transformations, and runs on a wide variety of CPUs using only basic computations.
- **RC4** is a *stream cipher*, meaning it acts on a stream of data rather than blocks. The key is used to seed a pseudo-random number generator, which generates a *keystream* of keys. RC4 is used in **WEP**, but has been found to be breakable in a reasonable amount of computer time.

Asymmetric Encryption

- With *asymmetric encryption*, the decryption key, K_d , is not the same as the encryption key, K_e , and more importantly cannot be derived from it, which means the encryption key can be made publicly available, and only the decryption key needs to be kept secret. (or vice-versa, depending on the application.)
- One of the most widely used asymmetric encryption algorithms is **RSA**, named after its developers - Rivest, Shamir, and Adleman.
- RSA is based on two large prime numbers, p and q , (on the order of 512 bits each), and their product N .
 - K_e and K_d must satisfy the relationship:

$$(K_e * K_d) \% [(p - 1) * (q - 1)] = 1$$
 - The encryption algorithm is:

$$c = E(K_e)(m) = m^{K_e} \% N$$
 - The decryption algorithm is:

$$m = D(K_d)(c) = c^{K_d} \% N$$
- An example using small numbers:
 - $p = 7$
 - $q = 13$
 - $N = 7 * 13 = 91$
 - $(p - 1) * (q - 1) = 6 * 12 = 72$
 - Select $K_e < 72$ and relatively prime to 72, say 5
 - Now select K_d , such that $(K_e * K_d) \% 72 = 1$, say 29
 - The public key is now $(5, 91)$ and the private key is $(29, 91)$
 - Let the message, $m = 42$
 - Encrypt: $c = 42^5 \% 91 = 35$
 - Decrypt: $m = 35^{29} \% 91 = 42$

Note that asymmetric encryption is much more computationally expensive than symmetric encryption, and as such it is not normally used for large transmissions. Asymmetric encryption is suitable for small messages, authentication, and key distribution, as covered in the following sections.

Authentication

- Authentication involves verifying the identity of the entity who transmitted a message.
- For example, if $D(K_d)(c)$ produces a valid message, then we know the sender was in possession of $E(K_e)$.

- This form of authentication can also be used to verify that a message has not been modified
- Authentication revolves around two functions, used for *signatures* (or *signing*), and *verification*:
 - A signing function, $S(K_s)$ that produces an *authenticator*, A , from any given message m .
 - A Verification function, $V(K_v, m, A)$ that produces a value of "true" if A was created from m , and "false" otherwise.
 - Obviously S and V must both be computationally efficient.
 - More importantly, it must not be possible to generate a valid authenticator, A , without having possession of $S(K_s)$.
 - Furthermore, it must not be possible to divine $S(K_s)$ from the combination of (m and A), since both are sent visibly across networks.
- Understanding authenticators begins with an understanding of hash functions, which is the first step:
 - **Hash functions, $H(m)$** generate a small fixed-size block of data known as a *message digest*, or *hash value* from any given input data.
 - For authentication purposes, the hash function must be **collision resistant on m** . That is it should not be reasonably possible to find an alternate message m' such that $H(m') = H(m)$.
 - Popular hash functions are **MD5**, which generates a 128-bit message digest, and **SHA-1**, which generates a 160-bit digest.
- Message digests are useful for detecting (accidentally) changed messages, but are not useful as authenticators, because if the hash function is known, then someone could easily change the message and then generate a new hash value for the modified message. Therefore authenticators take things one step further by encrypting the message digest.
- A **message-authentication code, MAC**, uses symmetric encryption and decryption of the message digest, which means that anyone capable of verifying an incoming message could also generate a new message.
- An asymmetric approach is the **digital-signature algorithm**, which produces authenticators called **digital signatures**. In this case K_s and K_v are separate, K_v is the public key, and it is not practical to determine $S(K_s)$ from public information. In practice the sender of a message signs it (produces a digital signature using $S(K_s)$), and the receiver uses $V(K_v)$ to verify that it did indeed come from a trusted source, and that it has not been modified.
- There are three good reasons for having separate algorithms for encryption of messages and authentication of messages:
 1. Authentication algorithms typically require fewer calculations, making verification a faster operation than encryption.
 2. Authenticators are almost always smaller than the messages, improving space efficiency. (?)
 3. Sometimes we want authentication only, and not confidentiality, such as when a vendor issues a new software patch.

- Another use of authentication is *non-repudiation*, in which a person filling out an electronic form cannot deny that they were the ones who did so.

Key Distribution

- Key distribution with symmetric cryptography is a major problem, because all keys must be kept secret, and they obviously can't be transmitted over unsecure channels. One option is to send them *out-of-band*, say via paper or a confidential conversation.
- Another problem with symmetric keys, is that a separate key must be maintained and used for each correspondent with whom one wishes to exchange confidential information.
- Asymmetric encryption solves some of these problems, because the public key can be freely transmitted through any channel, and the private key doesn't need to be transmitted anywhere. Recipients only need to maintain one private key for all incoming messages, though senders must maintain a separate public key for each recipient to which they might wish to send a message. Fortunately the public keys are not confidential, so this *key-ring* can be easily stored and managed.
- Unfortunately there are still some security concerns regarding the public keys used in asymmetric encryption. Consider for example the following man-in-the-middle attack involving phony public keys:

One solution to the above problem involves *digital certificates*, which are public keys that have been digitally signed by a trusted third party. But wait a minute - How do we trust that third party, and how do we know *they* are really who they say they are? Certain *certificate authorities* have their public keys included within web browsers and other certificate consumers before they are distributed. These certificate authorities can then vouch for other trusted entities and so on in a web of trust, as explained more fully in section 15.4.3.

Implementation of Cryptography

- Network communications are implemented in multiple layers - Physical, Data Link, Network, Transport, and Application being the most common breakdown.
- Encryption and security can be implemented at any layer in the stack, with pros and cons to each choice:
 - Because packets at lower levels contain the contents of higher layers, encryption at lower layers automatically encrypts higher layer information at the same time.
 - However security and authorization may be important to higher levels independent of the underlying transport mechanism or route taken.
- At the network layer the most common standard is **IPSec**, a secure form of the IP layer, which is used to set up **Virtual Private Networks, VPNs**.
- At the transport layer the most common implementation is SSL, described below.

User Authentication

- A lot of chapter 14, Protection, dealt with making sure that only certain users were allowed to perform certain tasks, i.e. that a users privileges were dependent on his or her identity. But how does one verify that identity to begin with?

Passwords

- Passwords are the most common form of user authentication. If the user is in possession of the correct password, then they are considered to have identified themselves.
- In theory separate passwords could be implemented for separate activities, such as reading this file, writing that file, etc. In practice most systems use one password to confirm user identity, and then authorization is based upon that identification. This is a result of the classic trade-off between security and convenience.

Password Vulnerabilities

- Passwords can be guessed.
 - Intelligent guessing requires knowing something about the intended target in specific, or about people and commonly used passwords in general.
 - Brute-force guessing involves trying every word in the dictionary, or every valid combination of characters. For this reason good passwords should not be in any dictionary (in any language), should be reasonably lengthy, and should use the full range of allowable characters by including upper and lower case characters, numbers, and special symbols.
- "Shoulder surfing" involves looking over people's shoulders while they are typing in their password.
 - Even if the lurker does not get the entire password, they may get enough clues to narrow it down, especially if they watch on repeated occasions.
 - Common courtesy dictates that you look away from the keyboard while someone is typing their password.
 - Passwords echoed as stars or dots still give clues, because an observer can determine how many characters are in the password. :-)
- "Packet sniffing" involves putting a monitor on a network connection and reading data contained in those packets.
 - SSH encrypts all packets, reducing the effectiveness of packet sniffing.
 - However you should still never e-mail a password, particularly not with the word "password" in the same message or worse yet the subject header.
 - Beware of any system that transmits passwords in clear text. ("Thank you for signing up for XYZ. Your new account and password information are shown below".) You probably want to have a spare throw-away password to give these entities, instead of using the same high-security password that you use for banking or other confidential uses.

- Long hard to remember passwords are often written down, particularly if they are used seldomly or must be changed frequently. Hence a security trade-off of passwords that are easily divined versus those that get written down. :- (
- Passwords can be given away to friends or co-workers, destroying the integrity of the entire user-identification system.
- Most systems have configurable parameters controlling password generation and what constitutes acceptable passwords.
 - They may be user chosen or machine generated.
 - They may have minimum and/or maximum length requirements.
 - They may need to be changed with a given frequency. (In extreme cases for every session.)
 - A variable length history can prevent repeating passwords.
 - More or less stringent checks can be made against password dictionaries.

Encrypted Passwords

- Modern systems do not store passwords in clear-text form, and hence there is no mechanism to look up an existing password.
- Rather they are encrypted and stored in that form. When a user enters their password, that too is encrypted, and if the encrypted version match, then user authentication passes.
- The encryption scheme was once considered safe enough that the encrypted versions were stored in the publicly readable file "/etc/passwd".
 - They always encrypted to a 13 character string, so an account could be disabled by putting a string of any other length into the password field.
 - Modern computers can try every possible password combination in a reasonably short time, so now the encrypted passwords are stored in files that are only readable by the super user. Any password-related programs run as setuid root to get access to these files. (/etc/shadow)
 - A random seed is included as part of the password generation process, and stored as part of the encrypted password. This ensures that if two accounts have the same plain-text password that they will not have the same encrypted password. However cutting and pasting encrypted passwords from one account to another will give them the same plain-text passwords.

One-Time Passwords

- One-time passwords resist shoulder surfing and other attacks where an observer is able to capture a password typed in by a user.
 - These are often based on a **challenge** and a **response**. Because the challenge is different each time, the old response will not be valid for future challenges.
 - For example, The user may be in possession of a secret function $f(x)$. The system challenges with some given value for x , and the user responds with $f(x)$, which the system can then verify. Since the challenger gives a different (random) x each time, the answer is constantly changing.

- A variation uses a map (e.g. a road map) as the key. Today's question might be "On what corner is SEO located?", and tomorrow's question might be "How far is it from Navy Pier to Wrigley Field?" Obviously "Taylor and Morgan" would not be accepted as a valid answer for the second question!
- Another option is to have some sort of electronic card with a series of constantly changing numbers, based on the current time. The user enters the current number on the card, which will only be valid for a few seconds. A *two-factor authorization* also requires a traditional password in addition to the number on the card, so others may not use it if it were ever lost or stolen.
- A third variation is a *code book*, or *one-time pad*. In this scheme a long list of passwords is generated, and each one is crossed off and cancelled as it is used. Obviously it is important to keep the pad secure.

BIOMETRICS

- Biometrics involve a physical characteristic of the user that is not easily forged or duplicated and not likely to be identical between multiple users.
 - Fingerprint scanners are getting faster, more accurate, and more economical.
 - Palm readers can check thermal properties, finger length, etc.
 - Retinal scanners examine the back of the users' eyes.
 - Voiceprint analyzers distinguish particular voices.
 - Difficulties may arise in the event of colds, injuries, or other physiological changes.
