1st Internals Solution (C & D sections)

1. a) What is O.S.? What are the common tasks performed by O.S. and when they are performed? (7M)

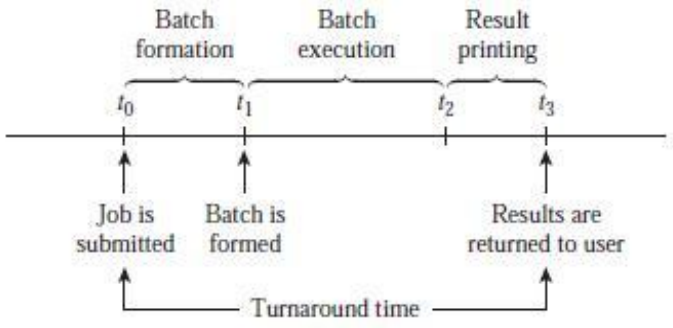
An operating system (OS) is different things to different users. Each user's view is called an abstract view because it emphasizes features that are important from the viewer‘s perspective, ignoring all other features. An operating system implements an abstract view by acting as an intermediary between the user and the computer system. This arrangement not only permits an operating system to provide several functionalities at the same time, but also to change and evolve with time.

Common Tasks Performed by Operating Systems

Task	When performed
Construct a list of resources	during booting
Maintain information for security	while registering new users
Verify identity of a user	at login time
Initiate execution of programs	at user commands
Maintain authorization information	when a user specifies which Collaborators can access what programs or data.
Perform resource allocation	when requested by users or programs
Maintain current status of resources	during resource allocation/deallocation
Maintain current status of programs and perform scheduling	continually during OS operation

b) Explain turnaround time in batch processing system (3M)

A batch of jobs was first recorded on a magnetic tape, using a less powerful and cheap computer. The batch processing system processed these jobs from the tape, which was faster than processing them from cards, and wrote their results on another magnetic tape. These were later printed and released to users. Figure 1.2 shows the factors that make up the turnaround time of a job.

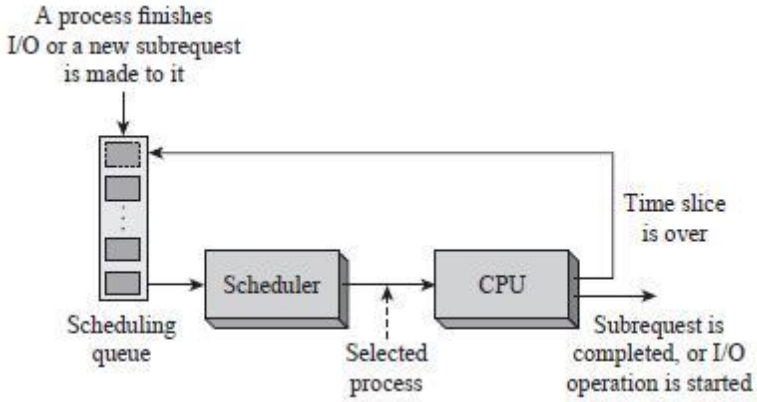


2. a) Explain the resource preemption, resource allocation strategies of an O.S. (5M)

The Resource Allocation function implements resources sharing by the users of a computer system. Basically it performs binding of a set of resources with the requesting program-that is it associates resources with a program. The related functions implement protection of users sharing a set of resources against mutual interference.

Resource Preemption: The scheduling technique used by a time-sharing kernel is called round-robin scheduling with time-slicing. It works as follows The kernel maintains a scheduling queue of processes that wish to use the CPU; it always schedules the process at the head of the queue. The kernel uses the notion of a time slice to avoid this situation. We use the notation δ for the time slice.

Time Slice: The largest amount of CPU time any time-shared process can consume when scheduled to execute on the CPU. If the time slice elapses before the process completes servicing of a subrequest, the kernel preempts the process, moves it to the end of the scheduling queue, and schedules another process. The preempted process would be rescheduled when it reaches the head of the queue once again.



Resource Allocation & Related Functions:

The resource allocation function allocates resources for use by a user's computation. Resources can be divided into two types:

1. System Provided Resources – like CPU, memory and IO devices
2. User created Resources – like files etc.

Resource allocation depends on whether a resource is a system resource or a user created resource.

There are two popular strategies for resource allocation

Partitioning of resources:. Using resource partition approach, OS decides priori what resources should be allocated to a user computation. This is known as static allocation as the allocation is made before the execution of the program starts.

Allocation from a pool: Using pool allocation approach, OS maintains a common pool & allocates resources from this pool on a need basis. This is called dynamic allocation because it takes place during the execution of program. It can lead to better utilization of resources because the allocation is made when a program request a resource. An OS can use a resource table as a central data structure for resource allocation. The table contains an entry for each resource unit in the system. The entry contains the name or address of the resource unit and its present system i.e whether it is free or allocated to some program. When a program raises a request for a resource, the resource should be allocated to it if it is presently free.

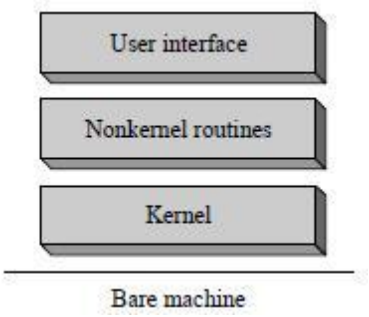
In the partition resource allocation approach, the OS decides on the resources to be allocated to a program based on the number of the program in the system. For Example, an OS may decide that a program can be allocated 1 MB of memory, 200 disk blocks and a monitor. Such a collection of resources is referred to as a partition. The resource table can have an entry for each resource partition. When a new program is to be started, an available partition is allocated to it.

b) Explain the benefits/features of distributed operating systems. (5M)

Benefit	Description
Resource sharing	Resources can be utilized across boundaries of individual computer systems.
Reliability	The OS continues to function even when computer systems or resources in it fail.
Computation speedup	Processes of an application can be executed in different computer systems to speed up its completion.
Communication	Users can communicate among themselves irrespective of their locations in the system.

3. a) Explain the kernel based operating system with a structure of time sharing system. (6M)

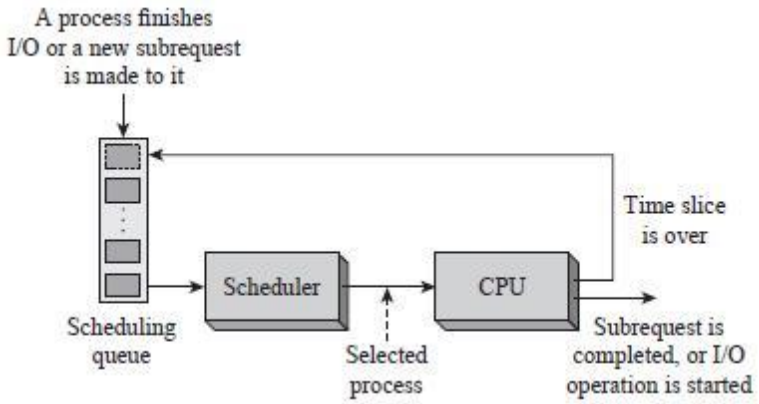
Figure below shows an abstract view of a kernel-based OS. The kernel is the core of the OS; it provides a set of functions and services to support various OS functionalities. The rest of the OS is organized as a set of nonkernel routines, which implement operations on processes and resources that are of interest to users, and a user interface.



A system call may be made by the user interface to implement a user command, by a process to invoke a service in the kernel, or by a nonkernel routine to invoke a function of the kernel. For example, when a user issues a command to execute the program stored in some file, say file alpha, the user interface makes a system call, and the interrupt servicing routine invokes a nonkernel routine to set up execution of the program. The nonkernel routine would make system calls to allocate memory for the program‘s execution, open file alpha, and load its contents into the allocated memory area, followed by another system call to initiate operation of the process that represents execution of the program. If a process wishes to create a child process to execute the program in file alpha, it, too, would make a system call and identical actions would follow. The historical motivations for the kernel-based OS structure were portability of the OS and convenience in the design and coding of nonkernel routines.

The scheduling technique used by a time-sharing kernel is called round-robin scheduling with time-slicing. It works as follows: The kernel maintains a scheduling queue of processes that wish to use the CPU; it always schedules the process at the head of the queue. When a scheduled process completes servicing of a subrequest, or starts an I/O operation, the kernel removes it from the queue and schedules another process. Such a process would be added at the end of the queue when it receives a new subrequest, or when its I/O operation completes. This arrangement ensures that all processes would suffer comparable delays before getting to use the CPU. However, response times of processes would degrade if a process consumes too much CPU time in servicing its subrequest. The kernel uses the notion of a time slice to avoid this situation. We use the notation δ for the time slice.

Time Slice The largest amount of CPU time any time-shared process can consume when scheduled to execute on the CPU. If the time slice elapses before the process completes servicing of a subrequest, the kernel preempts the process, moves it to the end of the scheduling queue, and schedules another process. The preempted process would be rescheduled when it reaches the head of the queue once again.



b) Explain the functions of an O.S. (4M)

Function	Description
Process management	Initiation and termination of processes, scheduling
Memory management	Allocation and deallocation of memory, swapping, virtual memory management
I/O management	I/O interrupt servicing, initiation of I/O operations, optimization of I/O device performance
File management	Creation, storage and access of files
Security and protection	Preventing interference with processes and resources
Network management	Sending and receiving of data over the network

4. a) Explain the following: (6M)

i) System generation

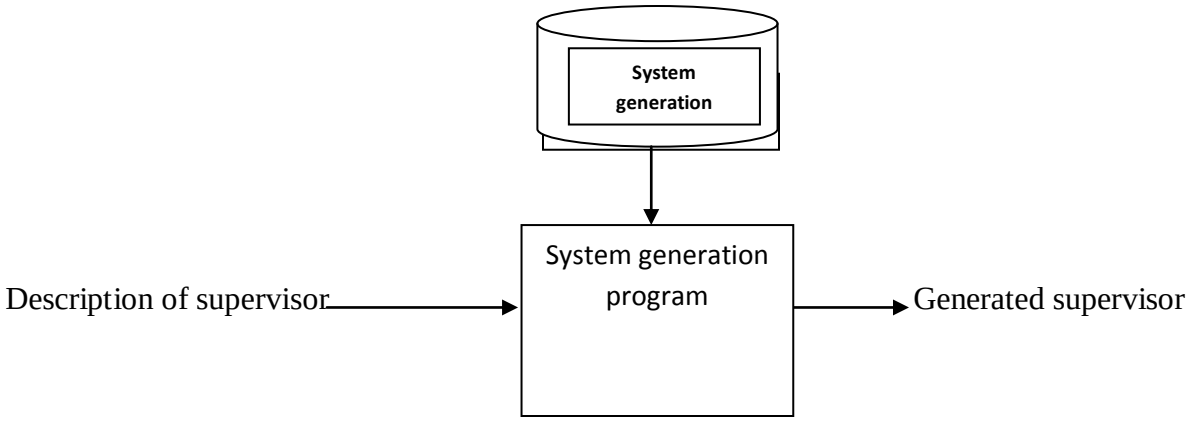
System generation was widely used during 1960's and 1970's. A supervisor was generated using software called the system generation program. The advantage of this approach was that the supervisors were not

individually designed, implemented and maintained. Only the supervisor generation program had to be maintained.

Steps to generate the supervisor:

The description of the supervisor is given as input to the system generation program. This program constructs a supervisor which matches the description.

System generation does not construct the supervisor; instead it selects some code modules from the ready library. All these modules are compiled and linked together to form the supervisor.



ii) Configuration tools

A configuration tool takes the input from system administrator about the configuration of hardware during installing O.S. and then prepares an appropriate version of the supervisor. This version is written on the disk and the minimal version of supervisor copied from the distribution tape is deleted from the disk. After this is done, the system admin is prompted to reboot the system so that the generated supervisor assumes control of the system.

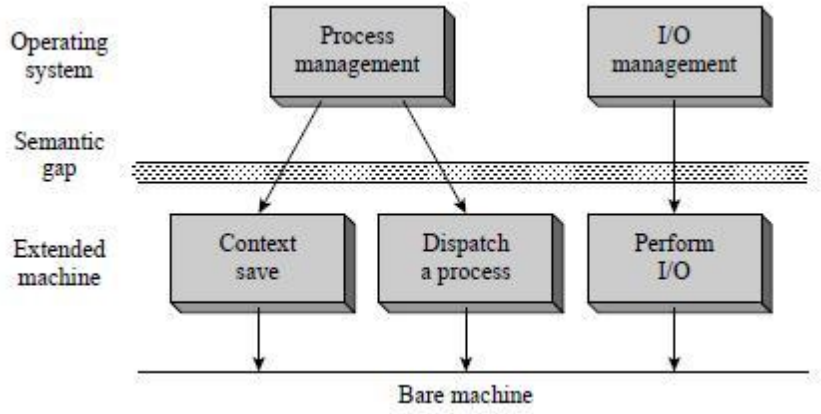
iii) Dynamic configuration of supervisor

Supervisor configuration approaches (both System generation and Configuration tools) both needs predefined and corresponding options must be specified during supervisor generation. The main difficulty with both the option is, it may be impossible to use a new kind of I/O device simply because the supervisor does not know how to perform I/O on it.

This difficulty is solved by using device drivers, where it can be added to it at any time during system operation. To enable this, the device handler and the supervisor must integrate together dynamical linking. After linking the device handler is stored in a library and is loaded from this library when needed.

b) Explain with a figure the working of a two layered O.S. Structure. (4M)

The simulator, which is a program, executes on the bare machine and mimics a more powerful machine that has many features desired by the OS. This new ―machine‖ is called an extended machine, and its simulator is called the extended machine software. Figure below illustrates a two-layered OS. The extended machine provides operations like context save, dispatching, swapping, and I/O initiation. The operating system layer is located on top of the extended machine layer. This arrangement considerably simplifies the coding and testing of OS modules by separating the algorithm of a function from the implementation of its primitive operations. It is now easier to test, debug, and modify an OS module than in a monolithic OS. We say that the lower layer provides an abstraction that is the extended machine. We call the operating system layer the top layer of the OS. The layered structures of operating systems have been evolved in various ways—using different abstractions and a different number of layers.



5.a) Explain: i) Monolithic OS and ii) Microkernel OS, specifying respective advantages and disadvantages.

MONOLITHIC STRUCTURE

An OS is a complex software that has a large number of functionalities and may contain millions of instructions. It is designed to consist of a set of software modules, where each module has a well-defined *interface* that must be used to access any of its functions or data. Such a design has the property that a module cannot ―see‖ inner details of functioning of other modules. This property simplifies design, coding and testing of an OS.

Early operating systems had a *monolithic* structure, whereby the OS formed a single software layer between the user and the bare machine, i.e., the computer system‘s hardware (see Figure). The user interface was provided by a command interpreter. The command interpreter organized creation of user processes. Both the command interpreter and user processes invoked OS functionalities and services through system calls.

Two kinds of problems with the monolithic structure were realized over a period of time. The sole OS layer had an interface with the bare machine. Hence architecture-dependent code was spread throughout the OS, and so there was poor portability. It also made testing and debugging difficult, leading to high costs of maintenance and enhancement. These problems led to the search for alternative ways to structure an OS.

• *Layered structure:* The layered structure attacks the complexity and cost of developing and maintaining an OS by structuring it into a number of layers.
The multiprogramming system of the 1960s is a well known example of a layered OS.

• *Kernel-based structure:* The kernel-based structure confines architecture dependence to a small section of the OS code that constitutes the kernel, so that portability is increased. The Unix OS has a kernel-based structure.

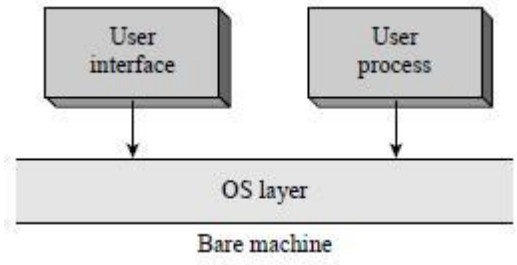


Figure : Monolithic OS.

Microkernel-based OS structure: The microkernel provides a minimal set of facilities and services for implementing an OS. Its use provides portability. It also provides extensibility because changes can be made to the OS without requiring changes in the microkernel.

MICROKERNEL-BASED OPERATING SYSTEMS

Putting all architecture-dependent code of the OS into the kernel provides good portability. However, in practice, kernels also include some architecture independent code. This feature leads to several problems. It leads to a large kernel size, which detracts from the goal of portability. It may also necessitate kernel modification to incorporate new features, which causes low extensibility.

A large kernel supports a large number of system calls. Some of these calls may be used rarely, and so their implementations across different versions of the kernel may not be tested thoroughly. This compromises reliability of the OS.

The *microkernel* was developed in the early 1990s to overcome the problems concerning portability, extensibility, and reliability of kernels. A microkernel is an essential core of OS code, thus it contains only a subset of the mechanisms typically included in a kernel and supports only a small number of system calls, which are heavily tested and used.

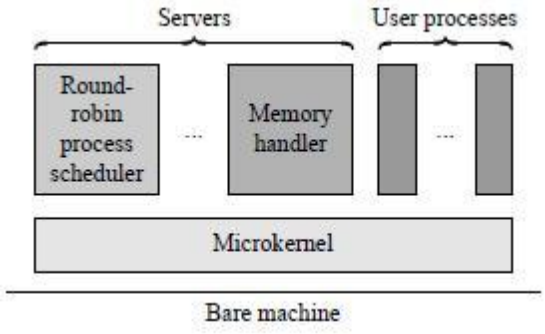


Figure :Structure of microkernel-based operating systems.

This feature enhances portability and reliability of the microkernel. Less essential parts of OS code are outside the microkernel and use its services, hence these parts could be modified without affecting the kernel; in principle, these modifications could be made without having to reboot the OS! The services provided in a microkernel are not biased toward any specific features or policies in an OS, so new functionalities and features could be added to the OS to suit specific operating environments.

Figure illustrates the structure of a microkernel-based OS. The microkernel includes mechanisms for process scheduling and memory management, etc., but does not include a scheduler or memory handler. These functions are implemented as *servers*, which are simply processes that never terminate. The servers and user processes operate on top of the microkernel, which merely performs interrupt handling and provides communication between the servers and user processes.

The small size and extensibility of microkernels are valuable properties for the embedded systems environment, because operating systems need to be both small and fine-tuned to the requirements of an embedded application. Extensibility of microkernels also conjures the vision of using the same microkernel for a wide spectrum of computer systems, from palm-held systems to large parallel and distributed systems. This vision has been realized to some extent.

b) Define the following with respect to an OS: i) Policies and mechanisms. ii) Portability and Extensibility.

Policies and Mechanisms

In determining how an operating system is to perform one of its functions, the OS designer needs to think at two distinct levels:

• *Policy:* A policy is the guiding principle under which the operating system will perform the function.

• *Mechanism:* A mechanism is a specific action needed to implement a policy.

A policy decides *what* should be done, while a mechanism determines *how* something should be done and actually does it. A policy is implemented as a decision-making module that decides which mechanism modules to call under what conditions. A mechanism is implemented as a module that performs a specific action. The following example identifies policies and mechanisms in round-robin scheduling.

Example 2.1 Policies and Mechanisms in Round-Robin Scheduling

In scheduling, we would consider the round-robin technique to be a *policy*. The following mechanisms would be needed to implement the round-robin scheduling policy:

Maintain a queue of ready processes
Switch the CPU to execution of the selected process (this action is called *dispatching*).

Portability and Extensibility of Operating Systems

The design and implementation of operating systems involves huge financial investments. To protect these investments, an operating system design should have a lifetime of more than a decade. Since several changes will take place in computer architecture, I/O device technology, and application environments during this time, it should be possible to adapt an OS to these changes. Two features are important in this context—portability and extensibility.

Porting is the act of adapting software for use in a new computer system.

Portability refers to the ease with which a software program can be ported—it is inversely proportional to the porting effort. *Extensibility* refers to the ease with which new functionalities can be added to a software system.

Porting of an OS implies changing parts of its code that are architecture dependent so that the OS can work with new hardware. Some examples of architecture-dependent data and instructions in an OS are:

• An interrupt vector contains information that should be loaded in various fields of the PSW to switch the CPU to an interrupt servicing routine. This information is architecture-specific.

• Information concerning memory protection and information to be provided to the memory management unit (MMU) is architecture-specific.

• I/O instructions used to perform an I/O operation are architecture-specific.

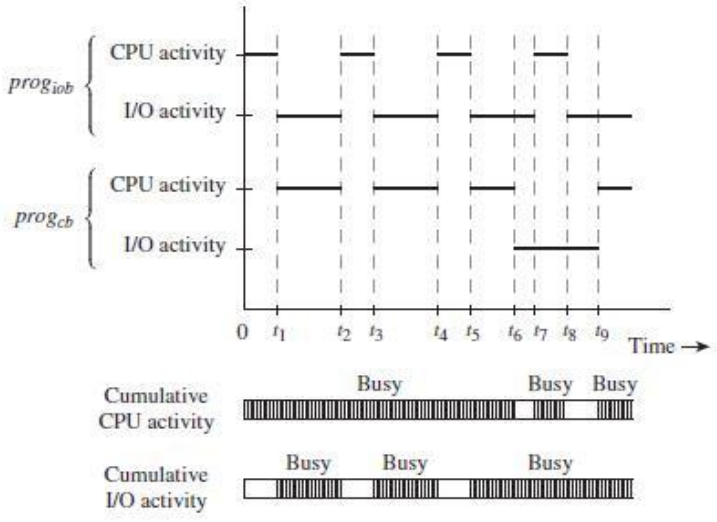
The architecture-dependent part of an operating system‘s code is typically associated with mechanisms rather than with policies. An OS would have high portability if its architecture-dependent code is small in

size, and its complete code is structured such that the porting effort is determined by the size of the architecture dependent code, rather than by the size of its complete code. Hence the issue of OS portability is addressed by separating the architecture-dependent and architecture-independent parts of an OS and providing well-defined interfaces between the two parts.

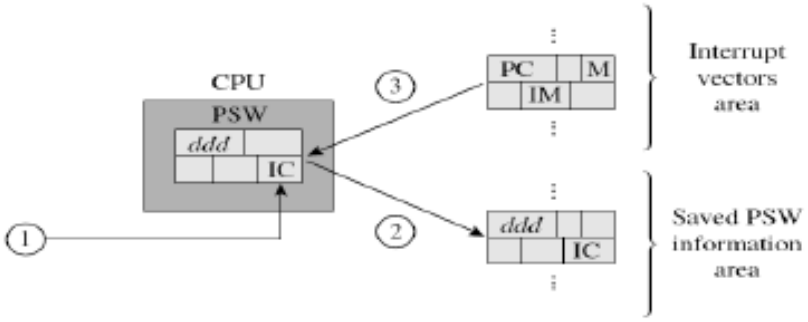
Extensibility of an OS is needed for two purposes: for incorporating new hardware in a computer system—typically newI/O devices or network adapters— and for providing new functionalities in response to new user expectations. Early operating systems did not provide either kind of extensibility. Hence even addition of a newI/O device required modifications to theOS. Later operating systems solved this problem by adding a functionality to the boot procedure. It would check for hardware that was not present when the OS was last booted, and either prompt the user to select appropriate software to handle the new hardware, typically a set of routines called a *device driver* that handled the new device, or itself select such software. The new software was then loaded and integrated with the kernel so that it would be invoked and used appropriately.

6. a) Why I/O bound programs should be given higher priorities in a multiprogramming environment? Illustrate with a timing diagram, assuming 2 processes where P1 requires 40ms of CPU & 50ms I/O time and P2 requires 30ms of CPU & 30ms of I/O time to complete they execution.

The kernel assigns numeric priorities to programs. We assume that priorities are positive integers and a large value implies a high priority. When many programs need the CPU at the same time, the kernel gives the CPU to the program with the highest priority. It uses priority in a preemptive manner; i.e., it pre-empts a low-priority program executing on the CPU if a high-priority program needs the CPU. This way, the CPU is always executing the highest-priority program that needs it. To understand implications of priority-based preemptive scheduling, consider what would happen if a high-priority program is performing an I/O operation, a low-priority program is executing on the CPU, and the I/O operation of the high-priority program completes—the kernel would immediately switch the CPU to the high-priority program. Assignment of priorities to programs is a crucial decision that can influence system throughput. Multiprogramming systems use the following priority assignment rule:
An I/O-bound program should have a higher priority than a CPU-bound program.



b) What are the operations performed by kernel when an interrupt occurs?



Step	Description
1. Set interrupt code	The interrupt hardware forms a code describing the cause of the interrupt. This code is stored in the *interrupt code* (IC) field of the PSW.
2. Save the PSW	The PSW is copied into the *saved PSW information* area. In some computers, this action also saves the general-purpose registers.
3. Load interrupt vector	The interrupt vector corresponding to the interrupt class is accessed. Information from the interrupt vector is loaded into the corresponding fields of the PSW. This action switches the CPU to the appropriate interrupt servicing routine of the kernel.

7. a) Explain virtual machine operating system(VMOS). What are the advantages of using virtual machines? (6M)

The VM OS creates several virtual machines. Each virtual machine is allocated to one user, who can use any OS of his own choice on the virtual machine and run his programs under this OS. This way user of the computer system can use different operating systems at the same time.

Each of these operating systems a guest OS and call the virtual machine OS the host OS. The computer used by the VM OS is called the host machine. A virtual machine is a virtual resource. Let us consider a virtual machine that has the same architecture as the host machine; i.e., it has a virtual CPU capable of executing the same instructions, and similar memory and I/O devices. It may, however, differ from the host machine in terms of some elements of its configuration like memory size and I/O devices. Because of the identical architectures of the virtual and host machines, no semantic gap exists between them, so operation of a virtual machine does not introduce any performance, software intervention is also not needed to run a guest OS on a virtual machine.

Virtual machines are employed for diverse purposes:

- To use an existing server for a new application that requires use of a different operating system. This is called workload consolidation; it reduces the hardware and operational cost of computing by reducing the number of servers needed in an organization.
- To provide security and reliability for applications that use the same host and the same OS. This benefit arises from the fact that virtual machines of different applications cannot access each other‘s resources.
- To test a modified OS (or a new version of application code) on a server concurrently with production runs of that OS.
- To provide disaster management capabilities by transferring a virtual machine from a server that has to shut down because of an emergency to another server available on the network.

b) Explain the goals of an O.S. (4M)

- Efficient use: Ensure efficient use of a computer‘s resources.
- User convenience: Provide convenient methods of using a computer system.
- Non-interference: Prevent interference in the activities of its users.

1 Efficient Use

An operating system must ensure efficient use of the fundamental computer system resources of memory, CPU, and I/O devices such as disks and printers. Poor efficiency can result if a program does not use a resource allocated to it, e.g., if memory or I/O devices allocated to a program remain idle. Such a situation may have a snowballing effect: Since the resource is allocated to a program, it is denied to other programs that need it. These programs cannot execute, hence resources allocated to them also remain idle. In addition, the OS itself consumes some CPU and memory resources during its own operation, and this consumption of resources constitutes an overhead that also reduces the resources available to user programs. To achieve good efficiency, the OS must minimize the waste of resources by programs and also minimize its own overhead. Efficient use of resources can be obtained by monitoring use of resources and performing corrective actions when necessary.

2 User Convenience

In the early days of computing, user convenience was synonymous with bare necessity—the mere ability to execute a program written in a higher level language was considered adequate. Experience with early operating systems led to demands for better service, which in those days meant only fast response to a user request. Other facets of user convenience evolved with the use of computers in new fields. Early operating systems had command-line interfaces, which required a user to type in a command and specify values of its parameters. Users needed substantial training to learn use of the commands, which was acceptable because most users were scientists or computer professionals. However, simpler interfaces were needed to facilitate use of computers by new classes of users. Hence graphical user interfaces (GUIs) were evolved. These interfaces used icons on a screen to represent programs and files and interpreted mouse clicks on the icons and associated menus as commands concerning them.

3. Non-interference

A computer user can face different kinds of interference in his computational activities. Execution of his program can be disrupted by actions of other persons, or the OS services which he wishes to use can be disrupted in a similar manner. The OS prevents such interference by allocating resources for exclusive use of programs and OS services, and preventing illegal accesses to resources. Another form of interference concerns programs and data stored in user files. A computer user may collaborate with some other users in the development or use of a computer application, so he may wish to share some of his files with them. Attempts by any other person to access his files are illegal and constitute interference. To prevent this form of interference, an OS has to know which files of a user can be accessed by which persons. It is achieved through the act of authorization, whereby a user specifies which collaborators can access what files. The OS uses this information to prevent illegal accesses to files.

8. a) Define process. List the different fields of a process control block. (5M)

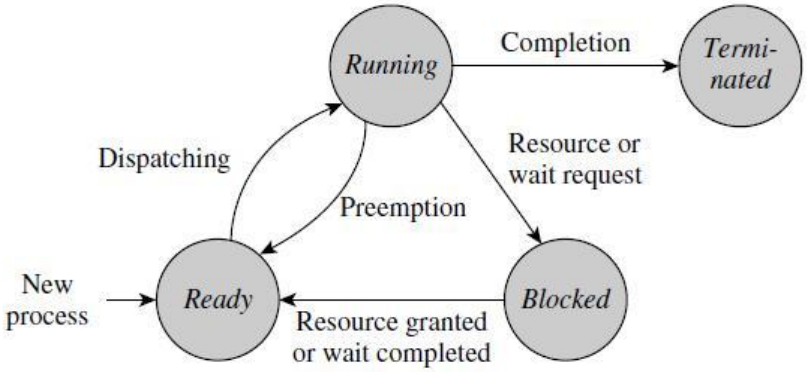
A program is a passive entity that does not perform any actions by itself; it has to be executed if the actions it calls for are to take place. A process is an execution of a program. It actually performs the actions specified in a program. An operating system shares the CPU among processes. This is how it gets user programs to execute.

Process Control Block (PCB) The process control block (PCB) of a process contains three kinds of information concerning the process—identification information such as the process id, id of its parent process, and id of the

user who created it; process state information such as its state, and the contents of the PSW and the general-purpose registers (GPRs); and information that is useful in controlling its operation, such as its priority, and its interaction with other processes. It also contains a pointer field that is used by the kernel to form PCB lists for scheduling, e.g., a list of ready processes. Table below describes the fields of the PCB data structure.

PCB field	Contents
Process id	The unique id assigned to the process at its creation.
Parent, child ids	These ids are used for process synchronization, typically for a process to check if a child process has terminated.
Priority	The priority is typically a numeric value. A process is assigned a priority at its creation. The kernel may change the priority dynamically depending on the nature of the process (whether CPU-bound or I/O-bound), its age, and the resources consumed by it (typically CPU time).
Process state	The current state of the process.
PSW	This is a snapshot, i.e., an image, of the PSW when the process last got blocked or was preempted. Loading this snapshot back into the PSW would resume operation of the process.
GPRs	Contents of the general-purpose registers when the process last got blocked or was preempted.
Event information	For a process in the *blocked* state, this field contains information concerning the event for which the process is waiting.
Signal information	Information concerning locations of signal handlers
PCB pointer	This field is used to form a list of PCBs for scheduling purposes.

b) Explain the four fundamental states of a process with state transition diagram. (5M)



State	Description
Running	A CPU is currently executing instructions in the process code.
Blocked	The process has to wait until a resource request made by it is granted, or it wishes to wait until a specific event occurs.
Ready	The process wishes to use the CPU to continue its operation; however, it has not been dispatched.
Terminated	The operation of the process, i.e., the execution of the program represented by it, has completed normally, or the OS has aborted it.