Operating System Overview

Concept and Definition

- A modern computer consists of one or more processors, some main memory, disks, printers, a keyboard, a mouse, a display, network interfaces, and various other input/output devices.
- If every application programmer had to understand how all these things work in detail, no code would ever get written. Furthermore, managing all these components and using them optimally is an exceedingly challenging job.
- For this reason, computers are equipped with a layer of software called **the operating system,** whose job is to provide user programs with a better, simpler, cleaner, model of the computer and to handle managing all the resources just mentioned.
- An **operating system** *is system software that manages a computer's hardware*. It also provides a basis for application programs and acts as an *intermediary between the computer user and the computer hardware.*
- Operating system is a program that acts as an intermediary between a user of a computer and the computer hardware. The goals of operating system are :
    - Execute user programs and make solving user problems easier
    - Make the computer system convenient to use
    - Use the computer hardware in an efficient manner

Computer System Structure

- Computer system can be divided into four components:
    - Hardware – provides basic computing resources
        - CPU, memory, I/O devices
    - Operating system
        - Controls and coordinates use of hardware among various applications and users
    - Application programs – define the ways in which the system resources are used to solve the computing problems of the users
        - Word processors, compilers, web browsers, database systems, video games
    - Users
        - People, machines, other computers

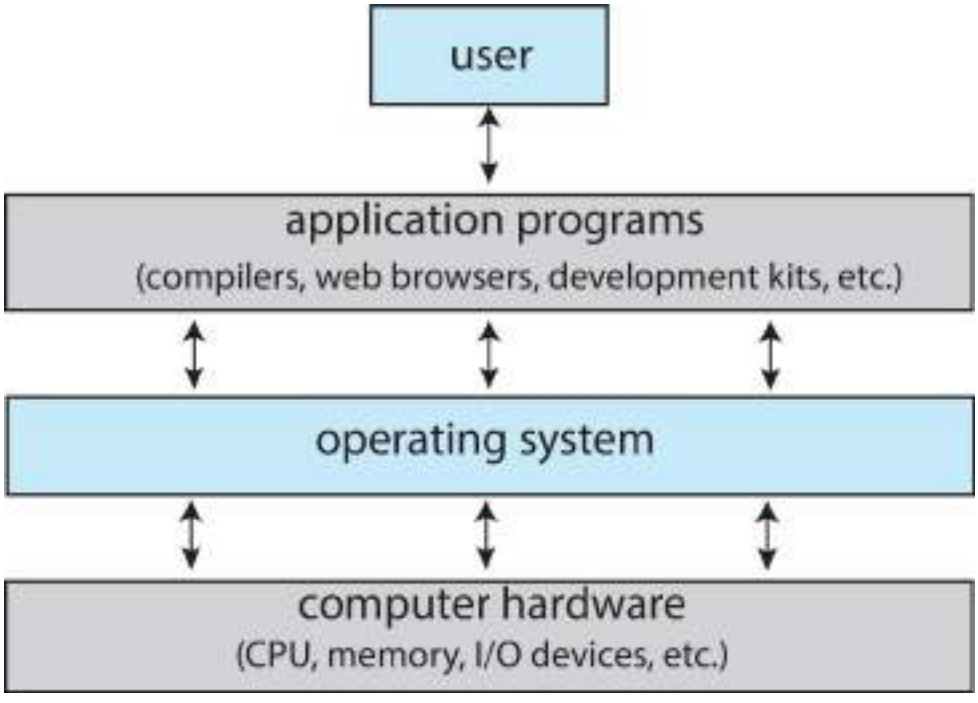

Figure:: Abstract view of the components of a computer system.

The place of Operating system

- A simple overview of the main components under discussion here is given in the figure below.

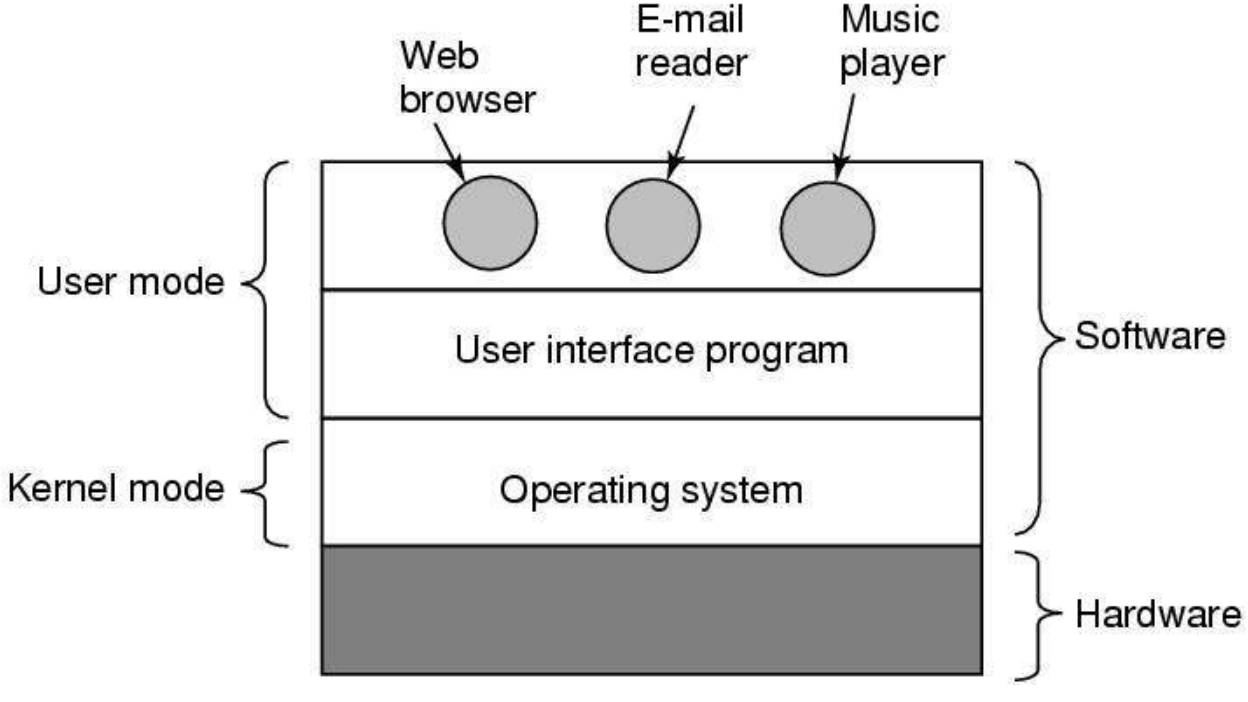

Figure: Where the operating system fits in.

- Here we see the hardware at the bottom. On top of the hardware is the software.
- Most computers have two modes of operation: **kernel** mode and **user mode**.

  - The operating system, the most fundamental piece of software, runs in kernel mode (also called **supervisor** mode).
  - In this mode it has complete access to all the hardware and can execute any instruction the machine is capable of executing.
- The rest of the software runs in user mode, in which only a subset of the machine instructions is available.
- In particular, those instructions that affect control of the machine or do I/O) Input/Output" are forbidden to user-mode programs.
- The user interface program, shell or GUI, is the lowest level of user-mode software, and allows the user to start other programs, such as a Web browser, email reader, or music player. These programs, too, make heavy use of the operating system.
- The placement of the operating system is shown in above figures. **It runs on the bare hardware and provides the base for all the other software.**

An Operating System (OS) is a system software which acts as an interface between a computer user and computer hardware. It performs all the basic tasks like memory management, file management, process management, handling input and output, and controlling peripheral devices such as disk drives and printers.

Roles of OS

- Implementing the user interface
- Sharing hardware among users
- Allowing users to share data among themselves
- Preventing users from interfering with one another
- Scheduling resources among users
- Facilitating input/output
- Recovering from errors
- Accounting for resource usage
- Facilitating parallel operations
- Organizing data for secure and rapid access
- Handling network communications

It is hard to pin down what an operating system is other than saying it is the software that runs in kernel mode—and even that is not always true.

- No universally accepted definition of OS
- Operating systems perform two essentially unrelated functions:
    1. Providing application programmers (and application programs, naturally) a clean Abstract set of resources instead of the messy hardware ones (functioning as an Extended Machine)
    2. Managing the hardware resources (functioning as a Resource Manager)

The Operating System as an Extended Machine

- At the Machine level the structure of a computer's system is complicated to program, mainly for input or output. Programmers do not deal with hardware. They will always mainly focus on implementing software. Therefore, a level of abstraction is supposed to be maintained.
- Operating systems provide a layer of abstraction for using disk such as files.
- OS creates a high level of abstraction for the programmer
- Example—(Disk I/O Operation)
    - Disk contains a list of named files
    - Each file can be opened for read write
    - After read write is complete, close that file
    - No any detail to deal
- The abstraction hides the truth about the hardware from the programmer and presents a simple view of the named files that can be read and written.
- This abstraction is the key to managing all this complexity. Good abstractions turn a nearly impossible task into two manageable ones. The first is defining and implementing the abstractions. The second is using these abstractions to solve the problem at hand. One abstraction that almost every computer user understands is the file, as mentioned above.
- OS prevents the user with the equivalent of an extended machine or virtual machine that is easier to program than the underlying hardware.
- It should be noted that the **operating system's real customers are the application programs** (via the application programmers, of course). They are the ones who deal directly with the operating system and its abstractions.

- In contrast, **end users deal with the abstractions provided by the user interface, either a command- line shell or a graphical interface**. While the abstractions at the user interface may be similar to the ones provided by the operating system, this is not always the case. To make this point clearer, consider the normal Windows desktop and the line-oriented command prompt. Both are programs running on the Windows operating system and use the abstractions Windows provides, but they offer very different user interfaces.

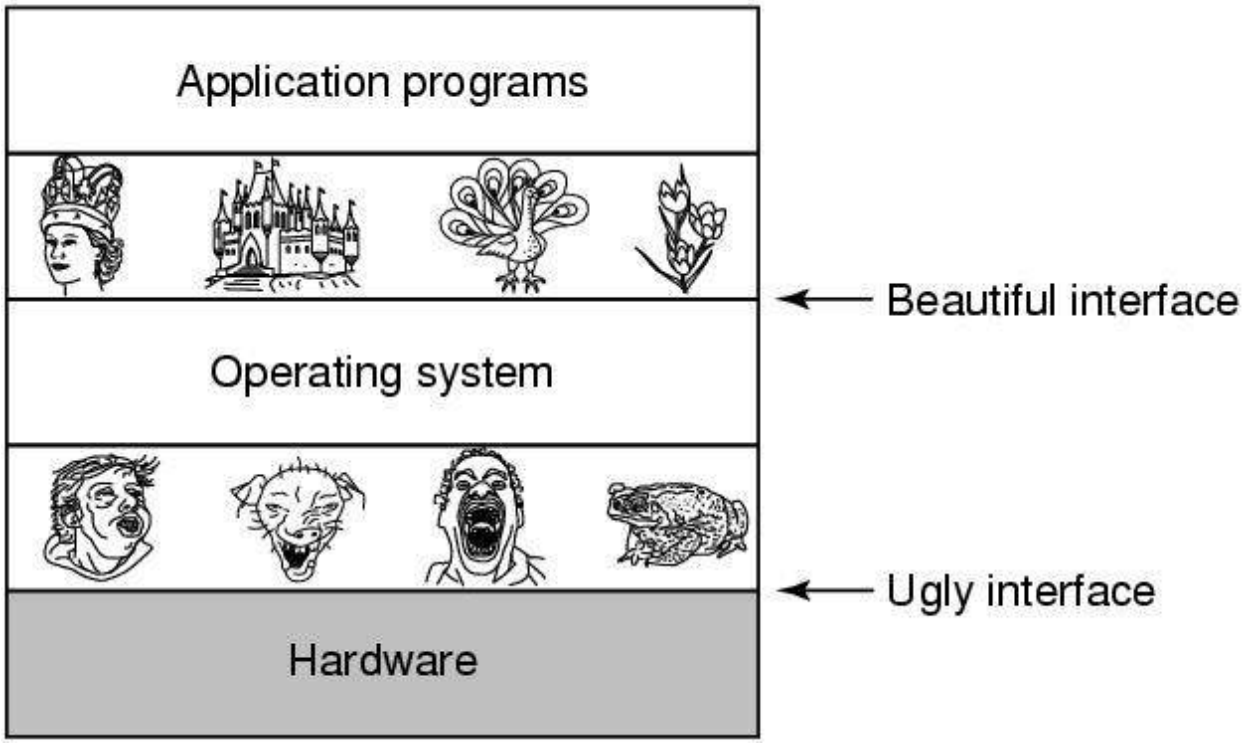

Figure: Operating systems turn ugly hardware into beautiful abstractions.

The Operating System as a Resource Manager

- Now-a-days all modern computers consist of processors, memories, timers, network interfaces, printers, and so many other devices.
- The operating system provides for an orderly and controlled allocation of the processors, memories, and I/O devices among the various programs in the bottom-up view.
- Operating system allows multiple programs to be in memory and run at the same time.
- Resource management includes multiplexing or sharing resources in two different ways: in **time** and in **space.**
- When a resource is **time multiplexed**, different programs or users take turns using it. First one of them gets to use the resource, then another, and so on.

- For example, with only one CPU and multiple programs that want to run on it, the operating system first allocates the CPU to one program, then, after it has run long enough, another program gets to use the CPU, then another, and then eventually the first one again.
- Determining how the resource is time multiplexed—who goes next and for how long—is the task of the operating system.
- Another example of time multiplexing is sharing the printer.
    - When multiple print jobs are queued up for printing on a single printer, a decision has to be made about which one is to be printed next.

- In **space multiplexing**, instead of the customers taking a chance, each one gets part of the resource.
    - For example − Main memory is divided into several running programs, so each one can be resident at the same time.
    - Another resource that is space multiplexed is the disk. In many systems a single disk can hold files from many users at the same time. Allocating disk space and keeping track of who is using which disk blocks is a typical operating system task.

Operating system is a software that manages resources and provides abstractions.

Evolution of operating system

Operating systems have been evolving through the years. Since operating systems have historically been closely tied to the architecture of the computers on which they run, we will look at successive generations of computers to see what their operating systems were like.

The First Generation (1945–55): Vacuum Tubes

- The earliest electronic digital computers had no operating systems.
- Machines of the time were so primitive that programs were often entered one bit at time on rows of mechanical switches (plug boards).
- Programming languages were unknown (not even assembly languages).
- Operating systems were unheard of.

The Second Generation (1955–65): Transistors and Batch Systems

- By the early 1950's, the routine had improved somewhat with the introduction of punch cards.
- The General Motors Research Laboratories implemented the first operating systems in early 1950's for their IBM 701.
- The system of the 50's generally ran one job at a time. These were called single-stream batch processing systems because programs and data were submitted in groups or batches.

The Third Generation (1965–1980): ICs and Multiprogramming

- The systems of the 1960's were also batch processing systems, but they were able to take better advantage of the computer's resources by running several jobs at once.
- So operating systems designers developed the concept of **multiprogramming** in which several jobs are in main memory at once; a processor is switched from job to job as needed to keep several jobs advancing while keeping the peripheral devices in use.
    - For example, on the system with no multiprogramming, when the current job paused to wait for other I/O operation to complete, the CPU simply sat idle until the I/O finished.
    - The solution for this problem that evolved was to partition memory into several pieces, with a different job in each partition. While one job was waiting for I/O to complete, another job could be using the CPU.
- Another major feature in third-generation operating system was the technique called **spooling** (simultaneous peripheral operations on line).
    - In spooling, a high-speed device like a disk interposed between a running program and a low-speed device involved with the program in input/output.
    - Instead of writing directly to a printer, for example, outputs are written to the disk.
    - Programs can run to completion faster, and other programs can be initiated sooner when the printer becomes available, the outputs may be printed.
- Another feature present in this generation was **time-sharing** technique, a variant of multiprogramming technique, in which each user has an on-line (i.e., directly connected) terminal. Because the user is present and interacting with the computer, the computer system must respond quickly to user requests, otherwise user productivity could suffer. Timesharing systems were developed to multiprogram large number of simultaneous interactive users.

- The first general-purpose timesharing system, CTSS (Compatible Time Sharing System), was developed at M.I.T.
- MULTICS (MULTiplexed Information and Computing Service)

The Fourth Generation (1980–Present): Personal Computers

- With the development of LSI (Large Scale Integration) circuits, chips, operating system entered in the system entered in the personal computer and the workstation age.
- Microprocessor technology evolved to the point that it becomes possible to build desktop computers as powerful as the mainframes of the 1970s.
- Two operating systems have dominated the personal computer scene: MS-DOS, written by Microsoft, Inc. for the IBM PC and other machines using the Intel 8088 CPU and its successors, and UNIX, which is dominant on the large personal computers using the Motorola 6899 CPU family.
- MAC OSX , Windows 95, Windows NT, Windows 98, Windows 200, Windows Me Windows XP, Windows 7, Windows 8, Windows 10, Windows11

The Fifth Generation (1990–Present): Mobile Computers

— The first real mobile phone appeared in 1946 and weighed some 40 kilos. You could take it wherever you went as long as you had a car in which to carry it.

— The first true handheld phone appeared in the 1970s and, at roughly one kilogram, was positively featherweight. It was affectionately known as ‘‘the brick.’’

— Pretty soon everybody wanted one. Today, mobile phone penetration is close to 90% of the global population. We can make calls not just with our portable phones and wrist watches, but soon with eyeglasses and other wearable items. Moreover, the phone part is no longer that interesting. We receive email, surf the Web, text our friends, play games, navigate around heavy traffic—and do not even think twice about it.

— While the idea of combining telephony and computing in a phone-like device has been around since the 1970s also, the first real smartphone did not appear until the mid-1990s when Nokia released the N9000, which literally combined two, mostly separate devices: a phone and a PDA (Personal Digital Assistant). In 1997, Ericsson coined the term smartphone for its GS88 ‘‘Penelope.’’

— Mobile phones, Smartphones

— Symbian OS, Blackberry OS, Android OS, iOS

Types of operating system

1. **Mainframe Operating Systems**

— Operating systems for mainframe computers (big size, thousands of disks)

— The operating systems for mainframes are heavily oriented toward processing many jobs at once, most of which need lots of I/O

— They typically offer three kinds of services: **batch**, **transaction processing**, and **timesharing**.

— A **batch system** is one that processes routine jobs without any interactive user present.

- Claims processing in an insurance company or sales reporting for a chain of stores is typically done in batch mode.

— **Transaction-processing systems** handle large numbers of small requests, for example, check processing at a bank or airline reservations.

- Each unit of work is small, but the system must handle hundreds or thousands per second.

— **Timesharing systems** allow multiple remote users to run jobs on the computer at once, such as querying a big database.

— These functions are closely related; mainframe operating systems often perform all of them.

— An example mainframe operating system is OS/390, a descendant of OS/360. However, mainframe operating systems are gradually being replaced by UNIX variants such as Linux.

2. **Server Operating Systems**

— They run on servers, which are either very large personal computers, workstations, or even mainframes.

— They serve multiple users at once over a network and allow the users to share hardware and software resources. Servers can provide print service, file service, or Web service.

— Typical server operating systems are Solaris, FreeBSD, Linux and Windows Server 201x.

3. **Multiprocessor Operating Systems**

— An increasingly common way to get major-league computing power is to connect multiple CPUs into a single system. Depending on precisely how they are connected and what is

shared, these systems are called **parallel computers, multicomputers, or multiprocessors.**

— They need special operating systems, but often these are variations on the server operating systems, with special features for communication, connectivity, and consistency.

— With the recent advent of multicore chips for personal computers, even conventional desktop and notebook operating systems are starting to deal with at least small-scale multiprocessors and the number of cores is likely to grow over time.

— Many popular operating systems, including Windows and Linux, run on multiprocessors.

4. Personal Computer Operating Systems

— All modern PC operating systems support multiprogramming, often with dozens of programs started up at boot time.

— Their job is to provide good support to a single user.

— They are widely used for word processing, spreadsheets, games, and Internet access.

— Common examples are Linux, FreeBSD, Windows 8, Windows 10, and Apple's OS X.

5. Handheld Computer Operating Systems

— Continuing on down to smaller and smaller systems, we come to tablets, smartphones and other handheld computers. A handheld computer, originally known as a PDA (Personal Digital Assistant), is a small computer that can be held in your hand during operation.

— Smartphones and tablets are the best-known examples.

— Market is currently dominated by Google's Android and Apple's iOS, but they hav e many competitors.

— Most of these devices boast multicore CPUs, GPS, cameras and other sensors, copious amounts of memory, and sophisticated operating systems.

6. Embedded Operating Systems

— Embedded systems run on the computers that control devices that are not generally thought of as computers and which do not accept user-installed software

— Typical examples are microwave ovens, TV sets, cars, DVD recorders, traditional phones, and MP3 players.

— The main property which distinguishes embedded systems from handhelds is the certainty that no untrusted software will ever run on it.

— You cannot download new applications to your microwave oven—all the software is in ROM. This means that there is no need for protection between applications, leading to design simplification.

— Systems such as Embedded Linux, QNX and VxWorks are popular in this domain.

7. Sensor-Node Operating Systems

— Networks of tiny sensor nodes are being deployed for numerous purposes. These nodes are tiny computers that communicate with each other and with a base station using wireless communication.

— Sensor networks are used to protect the perimeters of buildings, guard national borders, detect fires in forests, measure temperature and precipitation for weather forecasting, glean information about enemy movements on battlefields, and much more.

— Each **sensor node is a real computer**, with a CPU, RAM, ROM, and one or more environmental sensors. It runs a small, but real operating system, usually one that is event driven, responding to external events or making measurements periodically based on an internal clock.

— *The operating system has to be small and simple because the nodes have little RAM and battery lifetime is a major issue.*

— Also, as with embedded systems, all the programs are loaded in advance; users do not suddenly start programs they downloaded from the Internet, which makes the design much simpler.

— **TinyOS** is a well-known operating system for a sensor node.

8. Real-Time Operating Systems

— These systems are characterized by having time as a key parameter. For example, in industrial process- control systems, real-time computers have to collect data about the production process and use it to control machines in the factory.

— Often there are **hard deadlines** that must be met. For example, if a car is moving down an assembly line, certain actions must take place at certain instants of time.

- If, for example, a welding robot welds too early or too late, the car will be ruined.
- If the action absolutely must occur at a certain moment (or within a certain range), we have a **hard real-time system**.
- These systems must provide absolute guarantees that a certain action will occur by a certain time.

— A **soft real-time system**, is one where missing an occasional deadline, while not desirable, is acceptable and does not cause any permanent damage.
— Digital audio or multimedia systems fall in this category. Smartphones are also soft real-time systems.

NOTE: The embedded and real-time systems run only software put in by the system designers; users cannot add their own software, which makes protection easier. The handhelds and embedded systems are intended for consumers, whereas real-time systems are more for industrial usage.

9. Smart Card Operating Systems

— The smallest operating systems run on smart cards, which are credit-card-sized devices containing a CPU chip.
— They have very severe processing power and memory constraints.
— Some are powered by contacts in the reader into which they are inserted, but contactless smart cards are inductively powered, which greatly limits what they can do.
— Some of them can handle only a single function, such as electronic payments, but others can handle multiple functions.
— Often these are proprietary systems.

System Call

— A system call is a mechanism that provides the interface between a process and the operating system. It is a programmatic method in which a computer program requests a service from the kernel of the OS.
— The interface between a process and an operating system is provided by system call
— System calls are usually made when a process in user mode requires access to a resource. Then it requests the kernel to provide the resource via a system call.

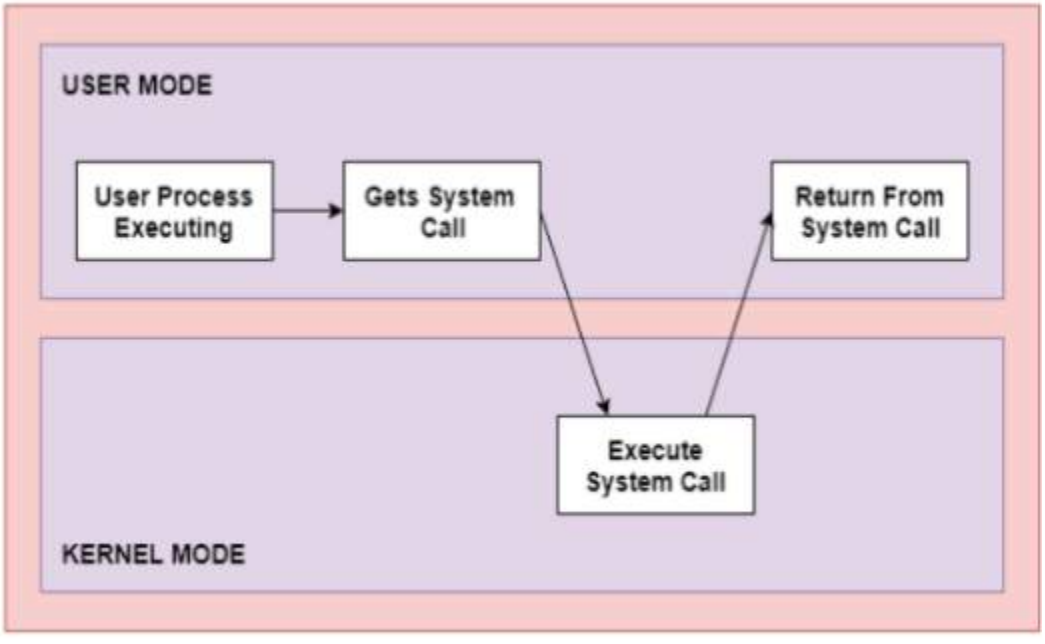

Figure: Execution of system call

As can be seen from above diagram, the processes execute normally in the user mode until a system call interrupts this. Then the system call is executed on a priority basis in the kernel mode. After the execution of the system call, the control returns to the user mode and execution of user processes can be resumed.

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

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

Understanding system call mechanism in Read System Call (in UNIX)

Like nearly all system calls, it is invoked from C programs by calling a library procedure with the same name as the system call: read. A call from a C program might look like this:

```
count = read(fd, buffer, nbytes);
```

It has three parameters: the first one specifying the file, the second one pointing to the buffer, and the third one giving the number of bytes to read.

System calls are performed in a series of steps. The steps in **read** system call are as follows :

- In preparation for calling the read library procedure, which actually makes the read system call, the calling program first pushes the parameters onto the stack, as shown in **steps 1–3** in following figure.
- The first and third parameters are called by value, but the second parameter is passed by reference, meaning that the address of the buffer (indicated by &) is passed, not the contents of the buffer. Then comes the actual call to the library procedure **(step 4).** This instruction is the normal procedure- call instruction used to call all procedures.
- The library procedure, possibly written in assembly language, typically puts the system-call number in a place where the operating system expects it, such as a register **(step 5).**
- Then it executes a TRAP instruction to switch from user mode to kernel mode and start execution at a fixed address within the kernel **(step 6).**
    - The TRAP instruction is actually fairly similar to the procedure-call instruction in the sense that the instruction following it is taken from a distant location and the return address is saved on the stack for use later.
    - Nevertheless, the TRAP instruction also differs from the procedure-call instruction in two fundamental ways.
        - First, as a side effect, it switches into kernel mode. The procedure call instruction does not change the mode.
        - Second, rather than giving a relative or absolute address where the procedure is located, the TRAP instruction cannot jump to an arbitrary address.
- The kernel code that starts following the TRAP examines the system-call number and then dispatches to the correct system-call handler, usually via a table of pointers to system-call handlers indexed on system-call number **(step 7).**
- At that point the system-call handler runs **(step 8).**
- Once it has completed its work, control may be returned to the user-space library procedure at the instruction following the TRAP instruction **(step 9).**
- This procedure then returns to the user program in the usual way procedure calls return **(step 10).**
- To finish the job, the user program has to clean up the stack, as it does after any procedure call **(step 11).**

- Push parameter into the stack (1-3)
- Calls library procedure (4)
- Pass parameters in registers (5)
- Switch from user mode to kernel mode and start to execute (6)
- Examine the system call number and then dispatch to the correct system call handler via a table of pointer (7)
- Run System call Handlers (8)
- Once the system call handler completed its work, control return to the library procedure (9)
- This procedure then return to the user program in the usual way (10)
- Increment SP before call to finish the job. (11)

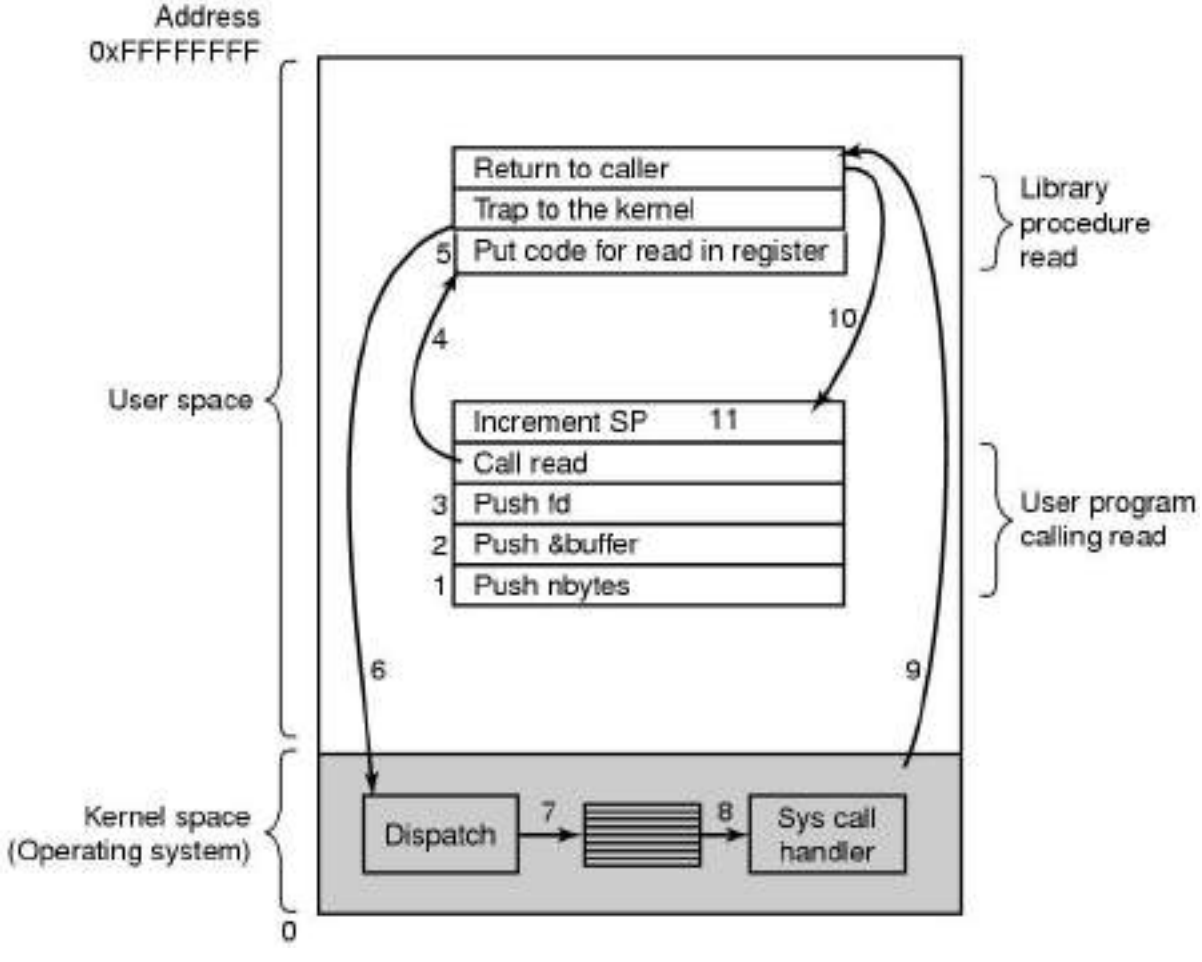

Figure : The 11 steps in making the system call read(fd, buffer, nbytes).

System Calls

- ☞ Interface between user programs and OS
- ☞ Varies from OS to OS
- ☞ System call issued by user program
- ☞ Call uses a library call of the same name
- ☞ Library routine puts machine into kernel modes (by issuing a special instruction)
- ☞ Finds actual routine for system call in a table
- ☞ Does the work involved in the call
- ☞ Returns to user program

Types of System Calls

The system calls can be classified in following types:

Process Management	☞ These system calls deal with processes such as process creation, process termination etc.
File Management	☞ These system calls are responsible for file manipulation such as creating a file, reading a file, writing into a file etc.
Directory Management	☞ These system calls are responsible for system calls that are related to directories or the file system like creating new directory, mounting filesystem etc.
Miscellaneous	☞ Other different system calls for different tasks

Some system calls in POSIX system

Some of the major POSIX system calls are listed below. The return code *s* is −1 if an error has occurred. The return codes are as follows: *pid* is a process id, *fd* is a file descriptor, *n* is a byte count, *position* is an offset within the file, and *seconds* is the elapsed time.

Process management

Call	Description
pid = fork( )	Create a child process identical to the parent
pid = waitpid(pid, &statloc, options)	Wait for a child to terminate
s = execve(name, argv, environp)	Replace a process' core image
exit(status)	Terminate process execution and return status

File management

Call	Description
fd = open(file, how, ...)	Open a file for reading, writing, or both
s = close(fd)	Close an open file
n = read(fd, buffer, nbytes)	Read data from a file into a buffer
n = write(fd, buffer, nbytes)	Write data from a buffer into a file
position = lseek(fd, offset, whence)	Move the file pointer
s = stat(name, &buf)	Get a file's status information

Directory- and file-system management

Call	Description
s = mkdir(name, mode)	Create a new directory
s = rmdir(name)	Remove an empty directory
s = link(name1, name2)	Create a new entry, name2, pointing to name1
s = unlink(name)	Remove a directory entry
s = mount(special, name, flag)	Mount a file system
s = umount(special)	Unmount a file system

Miscellaneous

Call	Description
s = chdir(dirname)	Change the working directory
s = chmod(name, mode)	Change a file's protection bits
s = kill(pid, signal)	Send a signal to a process
seconds = time(&seconds)	Get the elapsed time since Jan. 1, 1970

System programs

- ☞ System programs provide an environment where programs can be developed and executed. Also called system software.
- ☞ In the simplest sense, system programs also provide a bridge between the user interface and system calls. In reality, they are much more complex. For example, a compiler is a complex system program.
- ☞ The system program serves as a part of the operating system. It traditionally lies between the user interface and the system calls.
- ☞ System programs interacts with operating system through system calls. Sometimes it is also called as collection of system calls.
- ☞ The user view of the system is actually defined by system programs and not system calls because that is what they interact with and system programs are closer to the user interface.
- ☞ System programs as well as application programs form a bridge between the user interface and the system calls. So, from the user view the operating system observed is actually the system programs and not the system calls.

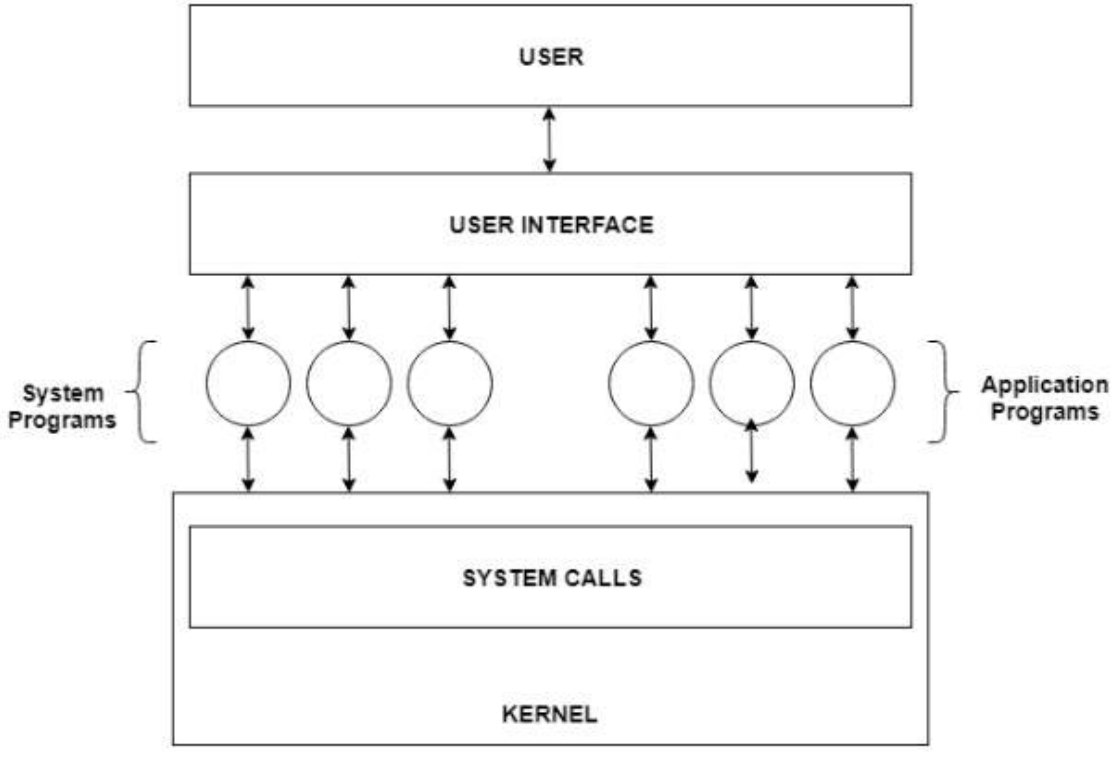

Figure: System Programs in OS Hierarchy

System programs can be divided into the following categories:

File Management

- These system programs are used to manipulate files in a system. This can be done using various commands like create, delete, copy, rename, print etc. These commands can create files, delete files, copy the contents of one file into another, rename files, print them etc.
- They are used to list, and generally manipulate files and directories.

Status information

- The status information system programs provide required data on the current or past status of the system.
- This may include the system date, system time, available space in memory or disk space, logged in users, detailed performance, logging, and debugging information etc.

File Modification

- System programs that are used for file modification basically change the data in the file or modify it in some other way.
- For Files stored on disks or other storage devices, we used different types of editors. For searching contents of files or perform transformations of files we use special commands
- Example: Text Editors

Programming-Language support

- These system programs provide additional support features for different programming languages. Some examples of these are compilers, assemblers, interpreters, debuggers etc. These compile a program and make sure it is error free respectively.

Communication

- These programs provide connection among processes, users, and computer systems. Users can send messages to another user on their screen, user can send e-mail, browsing on web pages, remote login, the transformation of files from one user to another
- Provides system communications.

Program Loading and Execution

— The system programs that deal with program loading and execution make sure that programs can be loaded into memory and executed correctly.

— Loaders and Linkers are a prime example of this type of system programs.

Operating System Structures

— There are different structures of operating system. The six designs we will discuss here are **monolithic systems**, **layered systems, microkernels, client-server systems, virtual machines,** and **exokernels**.

Monolithic Systems

— By far the most common organization, in the monolithic approach the entire operating system runs as a single program in kernel mode.

— The operating system is written as a collection of procedures, linked together into a single large executable binary program.

— The OS in it is written as a collection of procedures, each of which can call any of the other ones whenever it needs to.

— In this technique each procedure is free to call any other one, if the latter provides some useful computation than the former needs.

— To construct the actual object of the operating system when this approach is used, one first compiles all the individuals' procedures, or files containing the procedures and then binds them all together into a single object file using the system linker.

— In terms of information hiding, there is essentially none – every procedure is visible to every other procedure.

— Even in monolithic systems, however, it is possible to have some structure.

- The services (system calls) provided by the operating system are requested by putting the parameters in a well-defined place (e.g., on the stack) and then executing a trap instruction.
- This instruction switches the machine from user mode to kernel mode and transfers control to the operating system
- The operating system then fetches the parameters and determines which system call is to be carried out. After that, it indexes into a table that contains in slot k a pointer to the procedure that carries out system call k

This organization suggests a basic structure for the operating system:

1. A main program that invokes the requested service procedure.
2. A set of service procedures that carry out the system calls.
3. A set of utility procedures that help the service procedures.

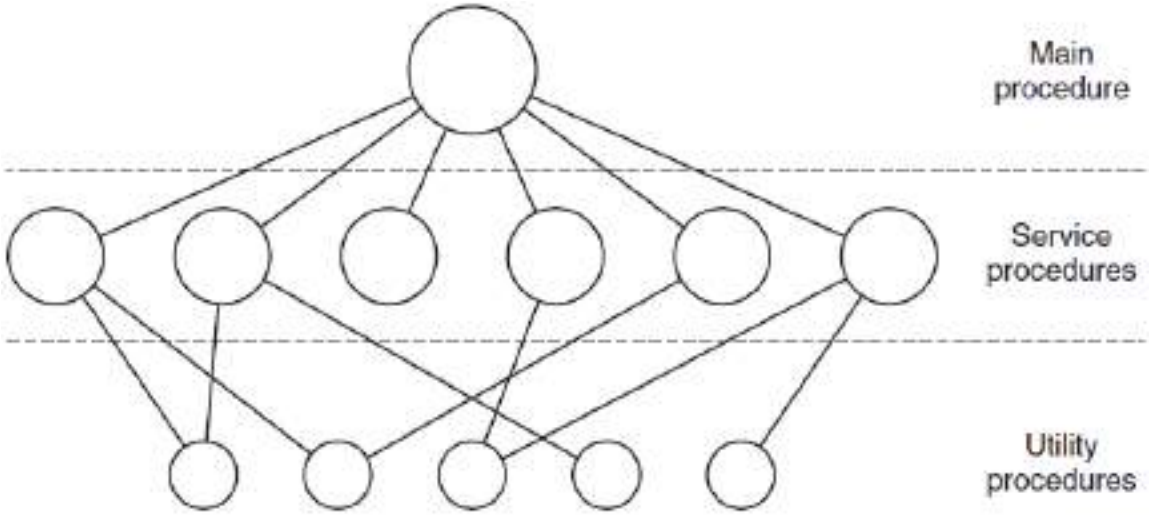

Figure : A simple structuring model for a monolithic system

Layered Systems

— To generalize the monolithic system into the hierarchy of layers, the first system called THE was built by E.W. Dijkstra in 1986.

— This system had 6 layers.

Layer	Function
5	The operator
4	User programs
3	Input/output management
2	Operator-process communication
1	Memory and drum management
0	Processor allocation and multiprogramming

Figure: Structure of THE operating system

— An OS can be broken into pieces and retain much more control on system. In this structure the OS is broken into number of layers (levels)

Microkernels

- With the layered approach, the designers have a choice where to draw the kernel- user boundary. Traditionally, all the layers went in the kernel, but that is not necessary.
    - In fact, a strong case can be made for putting as little as possible in kernel mode because bugs in the kernel can bring down the system instantly.
    - In contrast, user processes can be set up to have less power so that a bug there may not be fatal.
- The basic idea behind the microkernel design is to achieve high reliability by splitting the operating system up into small, well-defined modules, only one of which—the microkernel—runs in kernel mode and the rest run as relatively powerless ordinary user processes.
    - In particular, by running each device driver and file system as a separate user process, a bug in one of these can crash that component, but cannot crash the entire system.
    - Thus a bug in the audio driver will cause the sound to be garbled or stop, but will not crash the computer.
    - In contrast, in a monolithic system with all the drivers in the kernel, a buggy audio driver can easily reference an invalid memory address and bring the system to a grinding halt instantly

Example: MINIX system

- The process structure of MINIX 3 is shown in the figure, with the kernel call handlers labeled Sys. The device driver for the clock is also in the kernel because the scheduler interacts closely with it. The other device drivers run as separate user processes.
- Outside the kernel, the system is structured as three layers of processes all running in user mode.
    - The lowest layer contains the **device drivers**. Since they run in user mode, they do not have physical access to the I/O port space and cannot issue I/O commands directly. Instead, to program an I/O device, the driver builds a structure telling which values to write to which I/O ports and makes a kernel call telling the kernel

to do the write. This approach means that the kernel can check to see that the driver is writing (or reading) from I/O it is authorized to use. Consequently (and unlike a monolithic design), a buggy audio driver cannot accidentally write on the disk.

- Above the drivers is another user-mode layer containing the **servers**, which do most of the work of the operating system. One or more file servers manage the file system(s), the process manager creates, destroys, and manages processes, and so on.
- **User programs** obtain operating system services by sending short messages to the servers asking for the POSIX system calls..

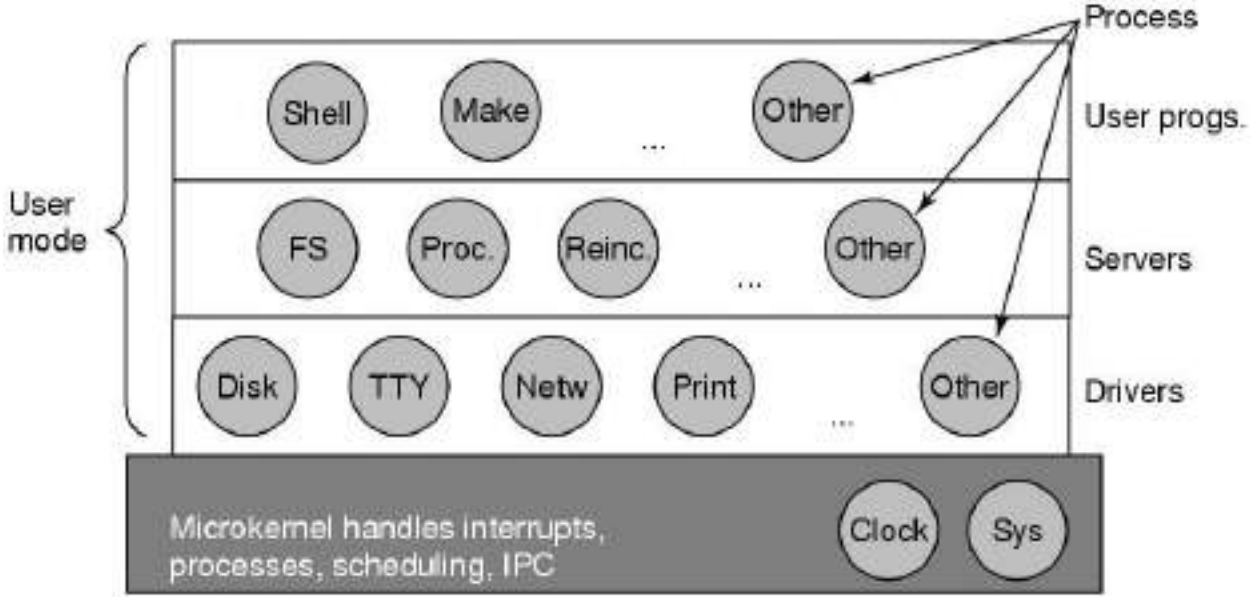

Figure: Simplified structure of the MINIX system

- An idea somewhat related to having a **minimal kernel** is to put the **mechanism** for doing something in the kernel but not the **policy**.
  - To make this point better, consider the scheduling of processes. A relatively simple scheduling algorithm is to assign a numerical priority to every process and then have the kernel run the highest-priority process that is runnable.
  - The **mechanism**—in the kernel—is to look for the highest-priority process and run it.
  - The **policy**—assigning priorities to processes—can be done by user-mode processes.

- In this way, policy and mechanism can be decoupled and the kernel can be made smaller

Microkernels

- Small number of processes are allowed in the kernel
- Minimizes effects of bugs
  - Don't want bug in driver to crash system
- Put mechanism in kernel and policy outside the kernel
  - Mechanism- schedule processes by priority scheduling algorithm
  - Policy-assign process priorities in user space
- Policy and mechanism can be decoupled and the kernel can be made smaller

Client-Server Model

- A slight variation of the microkernel idea is to distinguish two classes of processes, the **servers,** each of *which provides some service*, and the **clients**, *which use these services*. This model is known as the **client-server model**.
- Since clients communicate with servers by sending messages, the clients need not know whether the messages are handled locally on their own machines, or whether they are sent across a network to servers on a remote machine.
  - As far as the client is concerned, the same thing happens in both cases: requests are sent and replies come back.
  - Thus the client-server model is an abstraction that can be used for a single machine or for a network of machines.
- .Increasingly many systems involve users at their home PCs as clients and large machines elsewhere running as servers. In fact, much of the Web operates this way. A PC sends a request for a Web page to the server and the Web page comes back. This is a typical use of the client server model in a network.

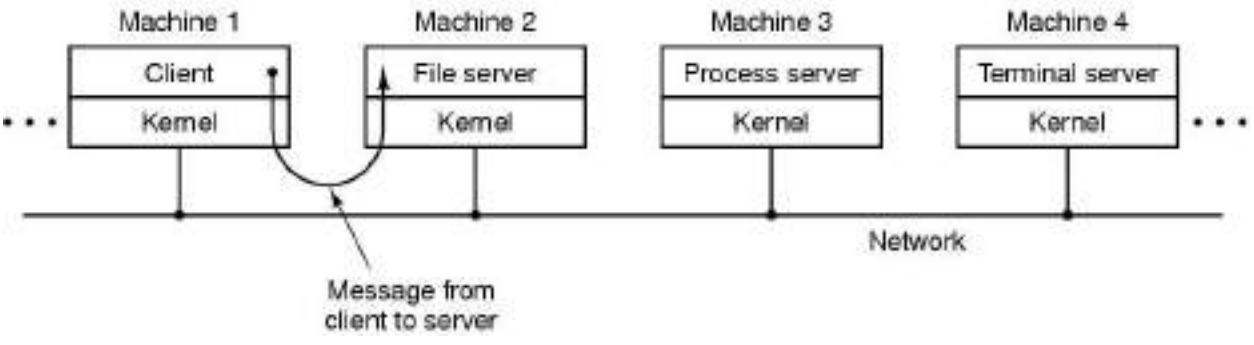

Figure: The client-server model over a network.

Virtual Machines

- The concept of a virtual machine is to provide an interface that looks like independent hardware, to multiple different OSes running simultaneously on the same physical hardware. Each OS believes that it has access to and control over its own CPU, RAM, I/O devices, hard drives, etc.
- One obvious use for this system is for the development and testing of software that must run on multiple platforms and/or OSes.
- One obvious difficulty involves the sharing of hard drives, which are generally partitioned into separate smaller virtual disks for each operating OS.

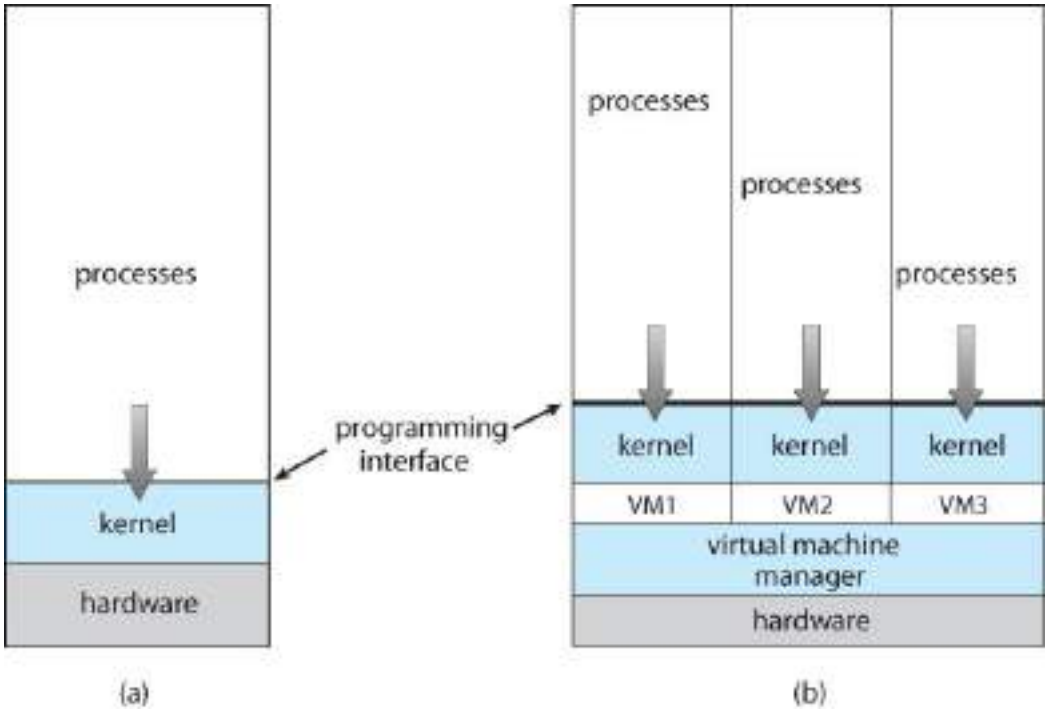

Figure: System models. (a) Nonvirtual machine. (b)Virtual machine

- Examples of Virtual Machines: VMWare, JVM, CLR ,etc.

VM/370 (First Virtual Machine)

- Initial release of OS/360 were strictly batch systems
- Many 360 users wanted to have time sharing and IBM developed a machine originally called CP/CMS and later named VM/370
- The heart of the system, known as the virtual machine monitor, runs on the bare hardware and does the multiprogramming, providing not only one, but several virtual machines to the next layer up.
- The virtual machines are not the extended machines but are the exact copies of the bare hardware including kernel/user mode, I/O, interrupts and everything else the real machine has.
- The CMS (Conversational Monitor System) is for interactive sharing users.
- Because each virtual machine is identical to the true hardware, each one can run any operating system that will run directly on the bare hardware. Different virtual machines can, and frequently do, run different operating systems.

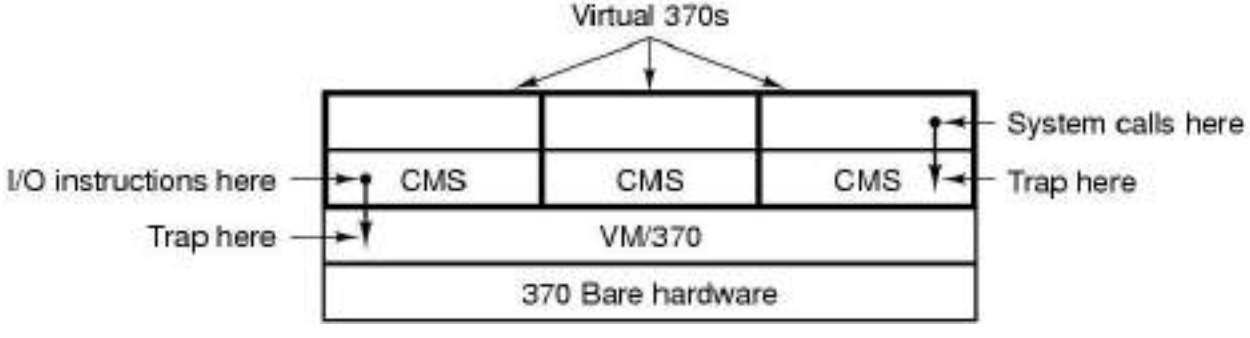

Figure: The structure of VM/370 with CMS.

NOTES:

- Virtual machines are made possible through virtualization technology. Virtualization uses software to simulate virtual hardware that allows multiple VMs to run on a single machine. The physical machine is known as the host while the VMs running on it are called guests.
- This process is managed by software known as a hypervisor. The hypervisor is responsible for managing and provisioning resources—like memory and storage—from the host to guests. It also schedules operations in VMs so they don't overrun each other when using resources. VMs only work if there is a hypervisor to virtualize and distribute host resources

Assignment: Discuss about type1 and type2 hypervisors

Exokernels

- Rather than cloning the actual machine, as is done with virtual machines, another strategy is partitioning it, in other words, giving each user a subset of the resources. Thus one virtual machine might get disk blocks 0 to 1023, the next one might get blocks 1024 to 2047, and so on. At the bottom layer, running in kernel mode, is a program called the **exokernel.** Its job is to allocate resources to virtual machines and then check attempts to use them to make sure no machine is trying to use somebody else's resources.
- The advantage of the exokernel scheme is that it saves a layer of mapping. In the other designs, each virtual machine thinks it has its own disk, with blocks running from 0 to some maximum, so the virtual machine monitor must maintain tables to remap disk addresses (and all other resources). With the exokernel, this remapping is not needed. The exokernel need only keep track of which virtual machine has been assigned which resource. This method still has the advantage of separating the multiprogramming (in the exokernel) from the user operating system code (in user space), but with less overhead, since all the exokernel has to do is keep the virtual machines out of each other's hair.

Research task: The structures of OSes (latest stable versions) like iOS, Android, Ubuntu, Windows 10 /11 , Mac OS

The Shell

- A shell is software that provides an interface between users and operating system of a computer to access the services of a kernel.
- It is named a shell because it is the outermost layer around the operating system
- The shell is the outermost layer of the operating system which provide a way to communicate with the operating system. This communication is carried out either interactively (input from the keyboard is acted upon immediately) or as a *shell script*. A shell script is a sequence of shell and operating system commands that is stored in a file.
- Operating system shells generally fall into one of two categories: command-line and graphical.
- Command-line shells provide a command-line interface (CLI) to the operating system, while graphical shells provide a graphical user interface (GUI)
- Command-line shells require the user to be familiar with commands and their calling syntax, and to understand concepts about the shell-specific scripting language (for

example, bash). Graphical shells place a low burden on beginning computer users, and are characterized as being easy to use. Since they also come with certain disadvantages, most GUI-enabled operating systems also provide CLI shells.

- In either category the primary purpose of the shell is to invoke or "launch" another program; however, shells frequently have additional capabilities such as viewing the contents of directories.
- Examples :
  - Windows Shell (GUI)
  - CommandPromt , PowerShell (***Windows CLI***:)

  Linux CLI

  - Bourne Shell (sh)
  - C Shell (csh)
  - Korn Shell (ksh)
  - Bourne-Again Shell(bash)

Open Source Operating System

- The term **open source** refers to something people can modify and share because its design is publicly accessible.
- **Open source software** is software with source code that anyone can inspect, modify, and enhance.
- The operating system whose source code is publicly visible and editable is called of an **open-source operating system.**
- Most of the open-source operating systems are Linux based. Examples: Linux Kernel, Ubuntu,Fedora, Red Hat Enterprise Linux, Linux Mint, Chromium OS, Elementary OS, etc.