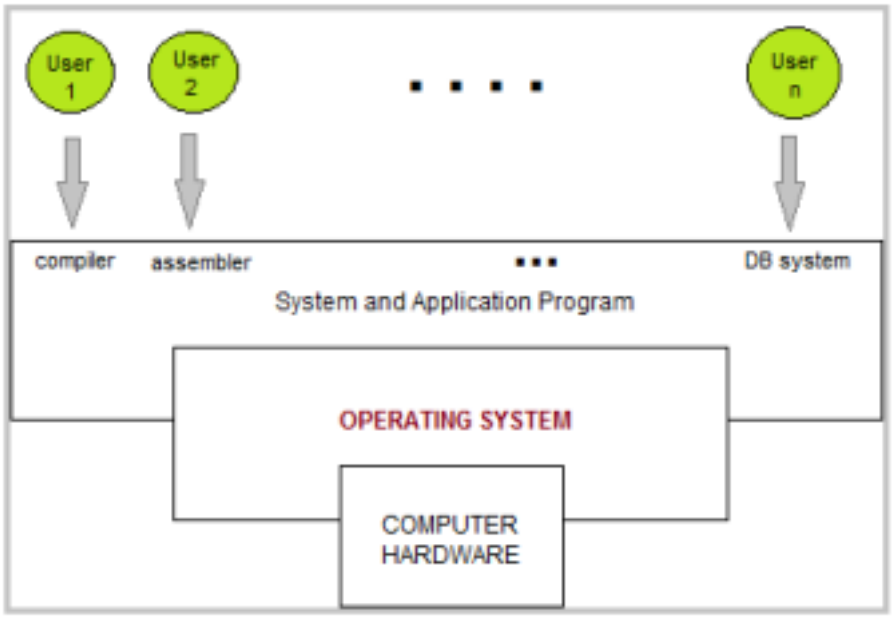
UNIT-1 Introduction to Operating System

OPERATING SYSTEM

An **operating system** is a program that manages the computer hardware. It also provides a basis for application programs and acts as an intermediary between the computer user and the computer hardware. An amazing aspect of operating systems is how varied they are in accomplishing these tasks. Mainframe operating systems are designed primarily to optimize utilization of hardware. Personal computer (PC) operating systems support complex games, business applications, and everything in between. Operating systems for handheld computers are designed to provide an environment in which a user can easily interface with the computer to execute programs.
Thus, some operating systems are designed to be ***convenient*****,** others to be ***efficient***, and others some combination of the two.
The **hardware**—the **central processing unit (CPU)**, the **memory**, and the **input/output (I/O) devices**—provides the basic computing resources for the system. The **application programs**—such as word processors, spreadsheets, compilers, and web browsers—define the ways in which these resources are used to solve users' computing problems. The operating system controls and coordinates the use of the hardware among the various application programs for the various users. The operating system acts as a manager of the above resources and allocates them to specific programs and users, whenever necessary to perform a particular task. Therefore operating system is the resource manager i.e. it can manage the resource of a computer system internally. The resources are processor, memory, files, and I/O devices. In simple terms, **an operating system is the interface between the user and the machine.**

Four Components of a Computer System



Two Views of Operating System

1. User's View
2. System View

User View:

The user view of the computer refers to the interface being used. Such systems are designed for one user to monopolize its resources, to maximize the work that the user is performing. In these

cases, the operating system is designed mostly for ease of use, with some attention paid to performance, and none paid to resource utilization.
In some cases, a user sits at a terminal connected to a **mainframe** or **minicomputer**. Other users are accessing the same computer through other terminals. These users share resources and may exchange information.
The operating system in such cases is designed to maximize resource utilization—to assure that all available CPU time, memory, and I/O are used efficiently and that no individual user takes more than her fair share.

System View:
Operating system can be viewed as a resource allocator also. A computer system consists of many resources like (hardware and software) resources that may be required to solve a problem: CPU time, memory space, file-storage space, I/O devices, and so on- that must be managed efficiently. The operating system acts as the manager of the resources, decides between conflicting requests, controls execution of programs etc.

Operating System Management Tasks

1. **Processor management** which involves putting the tasks into order and pairing them into manageable size before they go to the CPU.
2. **Memory management** which coordinates data to and from RAM (random-access memory) and determines the necessity for virtual memory.
3. **Device management** which provides interface between connected devices.
4. **Storage management** which directs permanent data storage.
5. **Application** which allows standard communication between software and your computer.
6. **User interface** which allows you to communicate with your computer.

Functions of Operating System

1. **Resource Management:** The resource management function of an OS allocates computer resources such as CPU time, main memory, secondary storage, and input and output devices for use.

a. **Process Management:** The operating system is responsible for the following activities in connection with process management:
i. Creating and deleting both user and system processes.
ii. Suspending and resuming processes.
iii. Providing mechanisms for process synchronization.
iv. Providing mechanisms for process communication
v. Providing mechanisms for deadlock handling.

b. **Memory Management:** The operating system is responsible for the following activities in connection with memory management:
i. Keeping track of which parts of memory are currently being used and by whom.
ii. Deciding which processes and data to move into and out of memory.
iii. Allocating and deallocating memory space as needed.

c. **Storage Management:**

i. **File – System Management:** The operating system is responsible for the following activities in connection with the file management:

UNIT-1 Introduction to Operating System

• Creating and deleting files
• Creating and deleting directories to organize files.
• Supporting primitives for manipulating files and directories.
• Mapping files onto secondary storage.
• Backing up files on stable (non-volatile) storage media.

ii. **Mass – Storage Management:** The operating system is responsible for the following activities in connection with disk management:

• Free-space Management
• Storage Allocation
• Disk Scheduling

d**. Device Management:** One of the purposes of operating system is to hide the peculiarities of specific hardware devices from the user.

2. **Data Management:** The data management functions of an OS govern the input and output of the data and their location, storage, and retrieval .

3. **Job Management:** The job management function of an OS prepares, schedules, controls, and monitors jobs submitted for execution to ensure the most efficient processing. A job is a collection of one or more related programs and their data

4. **Standard means of communication between user and computer:** The OS establishes a standard means of communication between users and their computer systems. It does this by providing a user interface and a standard set of commands that control the hardware .

5. In a multitasking operating system where multiple programs can be running at the same time, the operating system determines which applications should run in what order and how much time should be allowed for each application before giving another application a turn.

6. It manages the sharing of internal memory among multiple applications.

7. It handles input and output to and from attached hardware devices, such as hard disks, printers, and dial-up ports.

8. It sends messages to each application or interactive user (or to a system operator) about the status of operation and any errors that may have occurred.

9. It can offload the management of what are called batch jobs (for example, printing) so that the initiating application is freed from this work.

10. On computers that can provide parallel processing, an operating system can manage how to divide the program so that it runs on more than one processor at a time.

11. Operating systems perform basic tasks, such as recognizing input from the keyboard, sending output to the display screen, keeping track of files and directories on the disk, and controlling peripheral devices such as disk drives and printers.

12. The operating system is also responsible for security, ensuring that unauthorized users do not access the system.

UNIT-1 Introduction to Operating System

EVOLUTION OF OPERATING SYSTEMS

The evolution of operating systems is directly dependent to the development of computer systems and how users use them. Here is a quick tour of computing systems through the past fifty years in the timeline.

Early Evolution

· 1945: ENIAC, Moore School of Engineering, University of Pennsylvania. ·
1949: EDSAC and EDVAC
· 1949 BINAC - a successor to the ENIAC
· 1951: UNIVAC by Remington
· 1952: IBM 701
· 1956: The interrupt
· 1954-1957: FORTRAN was developed

Operating Systems by the late 1950s

By the late 1950s Operating systems were well improved and started supporting following usages:

· It was able to Single stream batch processing
· It could use Common, standardized, input/output routines for device access · Program transition capabilities to reduce the overhead of starting a new job was added · Error recovery to clean up after a job terminated abnormally was added. · Job control languages that allowed users to specify the job definition and resource requirements were made possible.

Operating Systems In 1960s

· 1961: The dawn of minicomputers
· 1962 Compatible Time-Sharing System (CTSS) from MIT
· 1963 Burroughs Master Control Program (MCP) for the B5000 system ·
1964: IBM System/360
· 1960s: Disks become mainstream
· 1966: Minicomputers get cheaper, more powerful, and really useful
· 1967-1968: The mouse
· 1964 and onward: Multics
· 1969: The UNIX Time-Sharing System from Bell Telephone Laboratories

Supported OS Features by 1970s

· Multi User and Multi tasking was introduced.
· Dynamic address translation hardware and Virtual machines came into picture. ·
Modular architectures came into existence.
· Personal, interactive systems came into existence.

Accomplishments after 1970

· 1971: Intel announces the microprocessor

• 1972: IBM comes out with VM: the Virtual Machine Operating System •
1973: UNIX 4th Edition is published
• 1973: Ethernet

• 1974 The Personal Computer Age begins
• 1974: Gates and Allen wrote BASIC for the Altair
• 1976: Apple II
• August 12, 1981: IBM introduces the IBM PC
• 1983 Microsoft begins work on MS-Windows
• 1984 Apple Macintosh comes out
• 1990 Microsoft Windows 3.0 comes out
• 1991 GNU/Linux
• 1992 The first Windows virus comes out
• 1993 Windows NT
• 2007: iOS
• 2008: Android OS

And as the research and development work still goes on, with new operating systems being developed and existing ones getting improved and modified to enhance the overall user experience, making operating systems fast and efficient like never before.

TYPES OF OPERATING SYSTEMS

Following are some of the most widely used types of Operating system.

1. Simple Batch System
2. Multiprogramming Batch System
3. Multiprocessor System
4. Desktop System
5. Distributed Operating System
6. Clustered System
7. Real time Operating System
8. Handheld System

SIMPLE BATCH SYSTEMS

The common input devices were card readers and tape drives. The common output devices were line printers, tape drives, and card punches. The user did not interact directly with the computer system. Rather, the user prepared a job which consisted of the program, the data, and some control information about the nature of the job (control cards)—and submitted it to the computer operator. The job was usually in the form of punch cards. At some later time (after minutes, hours, or days), the output appeared. The output consisted of the result of the program, as well as a dump of the final memory and register contents for debugging.

• In this type of system, there is no direct interaction between user and the computer. •
The user has to submit a job (written on cards or tape) to a computer operator. • Then computer operator places a batch of several jobs on an input device. • Jobs are batched together by type of languages and requirement.
• Then a special program, the monitor, manages the execution of each program in the batch.

• The monitor is always in the main memory and available for execution.

Following are some disadvantages of this type of system:

1. No interaction between user and computer.

2. No mechanism to prioritise the processes.

MULTIPROGRAMMING BATCH SYSTEMS

Multiprogramming increases CPU utilization by organizing jobs so that the CPU always has one to execute. The operating system keeps several jobs in memory simultaneously. This set of jobs is a subset of the jobs kept in the job pool which contains all jobs that enter the system

• In this, the operating system picks up and begins to execute one of the jobs from memory. • Once this job needs an I/O operation operating system switches to another job (CPU and OS always busy).
• Jobs in the memory are always less than the number of jobs on disk(Job Pool). • If several jobs are ready to run at the same time, then the system chooses which one to run through the process of CPU Scheduling.
• In Non-multiprogrammed system, there are moments when CPU sits idle and does not do any work.
• In Multiprogramming system, CPU will never be idle and keeps on processing.

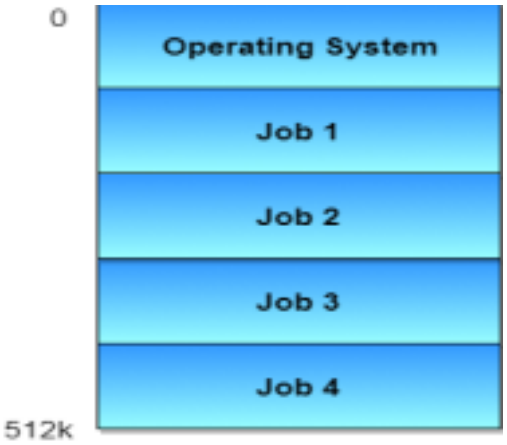
TIME-SHARING SYSTEMS

Time-Sharing Systems are very similar to Multiprogramming batch systems, but multiprogramming systems did not provide user interaction with the computer system. In fact time sharing systems are an extension of multiprogramming systems. In time sharing systems the prime focus is on minimizing the response time, while in multiprogramming the prime focus is to maximize the CPU usage.

In time-sharing systems, the CPU executes multiple jobs by switching among them, but the switches occur so frequently that the users can interact with each program while it is running. A time-shared operating provides each user with a small portion of a time-shared computer. Each user has at least one separate program in memory. A program loaded into memory and executing is commonly referred to as a **process**.

When a process executes, it typically executes for only a short time before it either finishes or needs to perform I/O. I/O may be interactive; that is, output goes to a display for the user and input comes from a user keyboard, mouse, or other device. Since interactive I/O typically runs at "people speeds," it may take a long time to complete. Input, for example, may be bounded by the user's typing speed; seven characters per second is fast for

people, but incredibly slow for computers. Rather than let the CPU sit idle as this interactive input takes place, the operating system will rapidly switch the CPU to the program of some other user.



MULTIPROCESSOR SYSTEMS

A multiprocessor system(also known as **parallel systems** or **tightly coupled systems)** consists of several processors that share a common physical memory. Multiprocessor system provides higher computing power and speed. In multiprocessor system all processors operate under single operating system. Multiplicity of the processors and how they do act together are transparent to the others.

Following are some advantages:

1. **Enhanced performance**
2. **Increased throughput,** Execution of several tasks by different processors concurrently, increases the system's throughput without speeding up the execution of a single task. 3. If possible, system divides task into many subtasks and then these subtasks can be executed in parallel in different processors. Thereby **speeding up the execution** of single tasks.
4. **Economy of scale**, Multiprocessor systems can cost less than equivalent multiple single processor systems, because they can share peripherals, mass storage, and power supplies. If several programs operate on the same set of data, it is cheaper to store those data on one disk and to have all the processors share them
5. **Increased reliability**. If functions can be distributed properly among several processors, then the failure of one processor will not halt the system, only slow it down.

DESKTOP SYSTEMS

Earlier, CPUs and PCs lacked the features needed to protect an operating system from user programs. PC operating systems therefore were neither **multiuser** nor **multitasking**. However, the goals of these operating systems have changed with time; instead of maximizing CPU and peripheral utilization, the systems opt for maximizing user convenience and responsiveness.

These systems are called **Desktop Systems** and include PCs running Microsoft Windows and the Apple Macintosh. Operating systems for these computers have benefited in several ways from the development of operating systems for **mainframes**.

Microcomputers were immediately able to adopt some of the technology developed for larger operating systems. On the other hand, the hardware costs for microcomputers are sufficiently **low**

that individuals have sole use of the computer, and CPU utilization is no longer a prime concern. Thus, some of the design decisions made in operating systems for mainframes may not be appropriate for smaller systems.

DISTRIBUTED OPERATING SYSTEMS

The growth of computer networks—especially the Internet and World Wide Web (WWW)—has had a profound influence on the recent development of operating systems. The motivation behind developing distributed operating systems is the availability of powerful and inexpensive microprocessors and advances in communication technology.

Distributed systems comprising of many computers that are inter connected by communication networks. The main benefit of distributed systems is its low price/performance ratio.

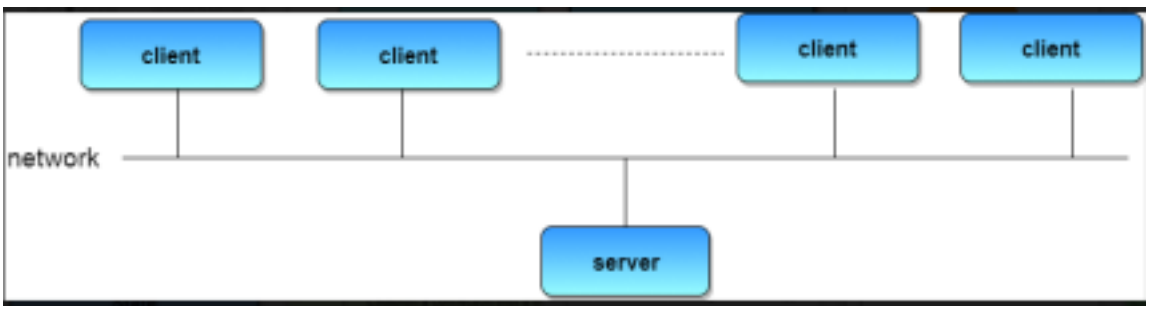
Following are some advantages

1. As there are multiple systems involved, user at one site can utilize the resources of systems at another sites for intensive tasks.
2. Fast processing.
3. Less load on the Host Machine.

The two types of Distributed Operating Systems are: ***Client-Server Systems*** and ***Peer-to-Peer Systems***.

Client-Server Systems

Centralized systems today act as **server systems** to satisfy requests generated by **client systems**. The general structure of a client-server system is depicted in the figure below:



Server Systems can be broadly categorized as **compute servers** and **file servers**.

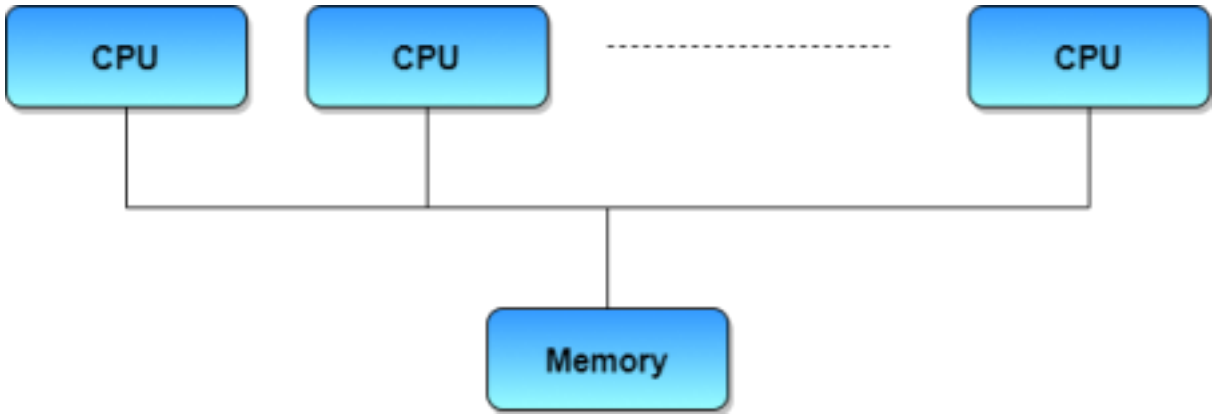
- **Compute-server systems** provide an interface to which clients can send requests to perform an action, in response to which they execute the action and send back results to the client.
- **File-server systems** provide a file-system interface where clients can create, update, read, and delete files.

Peer-to-Peer Systems

The growth of computer networks - especially the Internet and World Wide Web (WWW) – has had a profound influence on the recent development of operating systems. When PCs were introduced in the 1970s, they were designed for **personal** use and were generally considered

standalone computers. With the beginning of widespread public use of the Internet in the 1980s for electronic mail and ftp many PCs became connected to computer networks.

In contrast to the **tightly coupled** systems, the computer networks used in these applications consist of a collection of processors that do not share memory or a clock. Instead, each processor has its own local memory. The processors communicate with one another through various communication lines, such as high-speed buses or telephone lines. These systems are usually referred to as loosely coupled systems (or distributed systems).



CLUSTERED SYSTEMS

- Like parallel systems, clustered systems gather together multiple CPUs to accomplish computational work.
- Clustered systems differ from parallel systems, however, in that they are composed of two or more individual systems coupled together.
- The definition of the term clustered is **not concrete;** the general accepted definition is that clustered computers share storage and are closely linked via LAN networking. • Clustering is usually performed to provide **high availability**.
- A layer of cluster software runs on the cluster nodes. Each node can monitor one or more of the others. If the monitored machine fails, the monitoring machine can take ownership of its storage, and restart the application(s) that were running on the failed machine. The failed machine can remain down, but the users and clients of the application would only see a brief interruption of service.
- **Asymmetric Clustering -** In this, one machine is in hot standby mode while the other is running the applications. The hot standby host (machine) does nothing but monitor the active server. If that server fails, the hot standby host becomes the active server.
- **Symmetric Clustering -** In this, two or more hosts are running applications, and they are monitoring each other. This mode is obviously more efficient, as it uses all of the available hardware.
- **Parallel Clustering -** Parallel clusters allow multiple hosts to access the same data on the shared storage. Because most operating systems lack support for this simultaneous data access by multiple hosts, parallel clusters are usually accomplished by special versions of software and special releases of applications.

Clustered technology is rapidly changing. Clustered system use and features should expand greatly as **Storage Area Networks(SANs)**. SANs allow easy attachment of multiple hosts to multiple storage units. Current clusters are usually limited to two or four hosts due to the complexity of connecting the hosts to shared storage.

REAL-TIME OPERATING SYSTEM

UNIT-1 Introduction to Operating System

It is defined as an operating system known to give maximum time for each of the critical operations that it performs, like OS calls and interrupt handling.

The Real-Time Operating system which guarantees the maximum time for critical operations and complete them on time are referred to as **Hard Real-Time Operating Systems.** This goal requires that all delays in the system be bounded, from the retrieval of stored data to the time it takes the operating system to perform any request made of it.

While the real-time operating systems that can only guarantee a maximum of the time, i.e. the critical task will get priority over other tasks, but no assurity of completing it in a defined time. These systems are referred to as **Soft Real-Time Operating Systems**.

HANDHELD SYSTEMS

Handheld systems include **Personal Digital Assistants(PDAs)**, such as `Palm-Pilots` or `Cellular Telephones` with connectivity to a network such as the Internet. They are usually of limited size due to which most handheld devices have a small amount of memory, include slow processors, and feature small display screens.

- Many handheld devices have between **512 KB** and **8 MB** of memory. As a result, the operating system and applications must manage memory efficiently. This includes returning all allocated memory back to the memory manager once the memory is no longer being used.
- Currently, many handheld devices do **not use virtual memory** techniques, thus forcing program developers to work within the confines of limited physical memory.
- Processors for most handheld devices often run at a fraction of the speed of a processor in a PC. Faster processors require **more power**. To include a faster processor in a handheld device would require a **larger battery** that would have to be replaced more frequently.
- The last issue confronting program designers for handheld devices is the small display screens typically available. One approach for displaying the content in web pages is **web clipping**, where only a small subset of a web page is delivered and displayed on the handheld device.

Some handheld devices may use wireless technology such as **BlueTooth**, allowing remote access to e-mail and web browsing. **Cellular telephones** with connectivity to the Internet fall into this category.

Types of Operating System

1. **Batch Operating System:** Batch operating system is the operating system which analyzes your input and groups them into batches i.e. data in each batch is of similar characteristics. And then it performs operation on each individual batch.

2. **Real-time:** A real-time operating system is a multitasking operating system that aims at executing real-time applications. Real-time operating systems often use specialized scheduling algorithms so that they can achieve a deterministic nature of behavior. The main object of real time operating systems is their quick and predictable response to events. They either have an event-driven or a time-sharing design. An event-driven system switches between tasks based on their priorities while time-sharing operating systems switch tasks based on clock interrupts.

UNIT-1 Introduction to Operating System

a. **Hard real-time system:** It has the most stringent requirements, guaranteeing that real-time tasks be completed within their deadlines. Safety-critical systems are typically hard real-time systems.

b. **Soft real-time system:** It is less restrictive, simply providing that a critical real-time task will receive priority over other tasks and that it will retain that priority until it completes. Many commercial operating systems – as well as Linux – provide soft real-time support.

3. **Multi-user vs. Single-user: A multi-user operating system** allows multiple users to access a computer system concurrently. Time-sharing system can be classified as multi-user systems as they enable a multiple user access to a computer through the sharing of time. **Single-user operating systems**, as opposed to a multi-user operating system, are usable by a single user at a time. Being able to have multiple accounts on a Windows operating system does not make it a multi-user system. Rather, only the network administrator is the real user. But for a Unix-like operating system, it is possible for two users to login at a time and this capability of the OS makes it a multi-user operating system.

4. **Multi-tasking vs. Single-tasking:** When a single program is allowed to run at a time, the system is grouped under a single-tasking system, while in case the operating system allows the execution of multiple tasks at one time, it is classified as a multi-tasking operating system. Multi tasking can be of two types namely, pre-emptive or co-operative. In **pre-emptive multitasking**, the operating system slices the CPU time and dedicates one slot to each of the programs. Unix like operating systems such as Solaris and Linux support pre-emptive multitasking. **Cooperative multitasking** is achieved by relying on each process to give time to the other processes in a defined manner. MS Windows prior to Windows 95 used to support cooperative multitasking.

5. **Single-processor Systems:** On a single-processor system, there is one main CPU capable of executing a general-purpose instruction set, including instructions from user processes. 6. **Multi-processor Systems:** A multiprocessing operating system allows a program to run on more than one central processing unit (CPU) at a time. This can come in very handy in some work environments, at schools, and even for some home-computing situations. a. **Asymmetric multiprocessing:** In this each processor is assigned a specific task. A master processor controls the system; the other processors either look to the master for instruction or have predefined tasks. This scheme defines a master-slave relationship. The master processor schedules and allocates work to the slave processors.

b. **Symmetric multiprocessing (SMP):** In this each processor performs all tasks within the operating system. SMP means that all processors are peers; no master-slave relationship exists between processors.

7. **Distributed:** A distributed operating system manages a group of independent computers and makes them appear to be a single computer. The development of networked computers that could be linked and communicate with each other, gave rise to distributed computing. Distributed computations are carried out on more than one machine. When computers in a group work in cooperation, they make a distributed system.

8. **Embedded:** Embedded operating systems are designed to be used in embedded computer systems. They are designed to operate on small machines like PDAs with less autonomy. They are able to operate with a limited number of resources. They are very compact and extremely efficient by design. Windows CE and Minix 3 are some examples of embedded operating systems.

UNIT-1 Introduction to Operating System

PROTECTION

Early computer systems were single-user programmer-operated systems. When programmers operated the computer from the console, they had complete control over the system. As operating systems developed, however, this control shifted. Early operating systems were called resident monitors; and starting with resident monitors, operating systems began to perform many of the functions, especially I/O, for which the programmer had previously been responsible.

In addition, to improve system utilization, the operating system began to share system resources among several programs simultaneously. With spooling, for example, one program might have been executing while I/O occurred for other processes; the disk simultaneously held data for many processes. With multiprogramming, several programs might be in memory at the same time.

This sharing both improved utilization and increased problems. When the system was run without sharing, an error in a program could cause problems only for the one program that was running. With sharing, many processes could be adversely affected by a bug in one program.

Dual-Mode and Multimode Operation

- User mode when executing harmless code in user applications
- Kernel mode (a.k.a. system mode, supervisor mode, privileged mode) when executing potentially dangerous code in the system kernel.
- Certain machine instructions (privileged instructions) can only be executed in kernel mode.
- Kernel mode can only be entered by making system calls. User code cannot flip the mode switch.
- Modern computers support dual-mode operation in hardware, and therefore most modern OS support dual-mode operation.

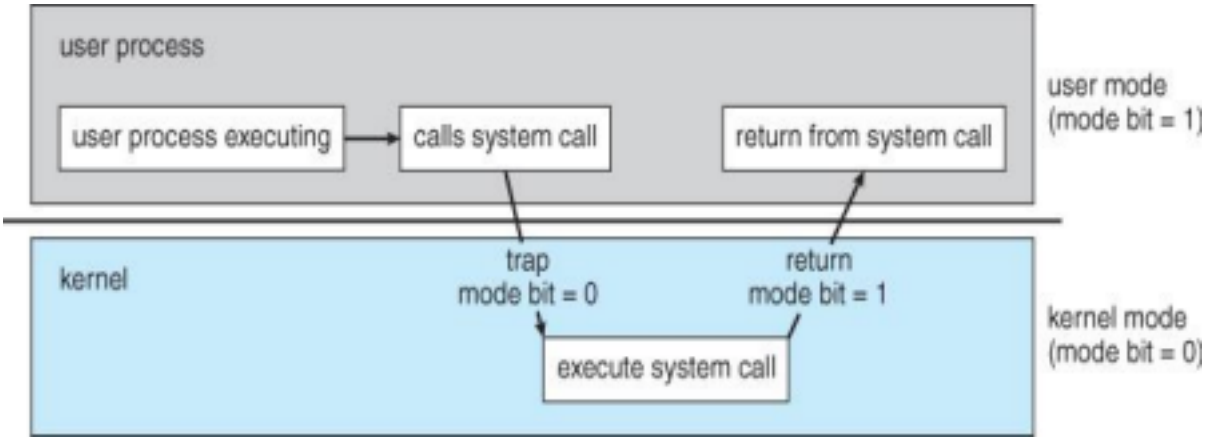


Figure - Transition from user to kernel mode

- The concept of modes can be extended beyond two, requiring more than a single mode bit

- CPUs that support virtualization use one of these extra bits to indicate when the virtual machine manager, VMM, is in control of the system. The VMM has more privileges than ordinary user programs, but not so many as the full kernel.
- System calls are typically implemented in the form of software interrupts, which causes the hardware's interrupt handler to transfer control over to an appropriate interrupt handler, which is part of the operating system, switching the mode bit to kernel mode in the process. The interrupt handler checks exactly which interrupt was generated, checks additional parameters (generally passed through registers) if appropriate, and then calls the appropriate kernel service routine to handle the service requested by the system call.
- User programs' attempts to execute illegal instructions (privileged or non-existent instructions), or to access forbidden memory areas, also generate software interrupts, which are trapped by the interrupt handler and control is transferred to the OS, which issues an appropriate error message, possibly dumps data to a log (core) file for later analysis, and then terminates the offending program.

I/O Protection

A user program may disrupt the normal operation of the system by issuing illegal I/O instructions, by accessing memory locations within the operating system itself, or by refusing to relinquish the CPU. We can use various mechanisms to ensure that such disruptions cannot take place in the system.

To prevent users from performing illegal I/O, we define all I/O instructions to be privileged instructions. Thus, users cannot issue I/O instructions directly; they must do it through the operating system by means of a system call. The operating system, executing in monitor mode, checks to make sure that the request is valid and (if it is valid) performs the I/O requested. The operating system then returns to the user. For I/O protection to be complete, we must be sure that a user program can never gain control of the computer in monitor mode.

UNIT-1 Introduction to Operating System

Memory Protection

we can provide memory protection by protecting the operating system from access by user programs and, in addition, to protect user programs from one another. This protection must be provided by the hardware. It can be implemented in several ways, we can separate each program's memory space from the others', we need the ability to determine the range of legal addresses that the program may access and to protect the memory outside that space. We can provide this protection by using two registers, usually a base and a limit, The base register holds the smallest legal physical memory address; the limit register contains the size of the range. For example, if the base register holds 300040 and the limit register is 120900, then the program can legally access all addresses from 300040 through 420940 inclusive.

The base and limit registers can be loaded only by the operating system, which uses a special privileged instruction. Since privileged instructions can be executed only in monitor mode, and since only the operating system executes in monitor mode, only the operating system can load the base and limit registers. This scheme allows the monitor to change the value of the registers but prevents user programs from changing the registers' contents

CPU Protection

In addition to protecting I/O and memory, we must ensure that the operating system 'maintains control. We must prevent a user program from,getting stuck in an infinite loop and never returning control to the operating system.

To accomplish this goal, we can use a **timer**. A timer can be set to interrupt the computer after a specified period. The period may be fixed (for example, 1/60 second) or variable (for example, from 1 millisecond to 1 second). A **variable timer** is generally implemented by a fixed-rate clock and a counter. The operating system sets the counter. Every time the clock ticks, the counter is decremented. When the counter reaches 0, an interrupt occurs.

Figure: Hardware address protection with base and limit registers.

OPERATING SYSTEM COMPONENTS

We can create a system as large and complex as an operating system only by partitioning it into smaller pieces. Each of these pieces should be a well-delineated portion of the system, with carefully defined inputs, outputs, and functions.

Process Management

A program does nothing unless its instructions are executed by a CPU. A program in execution can be thought of as a **process**. 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, also is a process.

A process needs certain resources—including CPU time, memory, files, and 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.

A program by itself is not a process; a program is a passive entity, such as the contents of a file stored on disk, whereas a process is an active entity.

Some processes are operating-system processes (those that execute system code) and the rest of which are user processes (those that execute user code). All these processes can potentially execute concurrently—by multiplexing the CPU among them on a single CPU

An OS is responsible for the following tasks with regards to process management:

- Creating and deleting both user and system processes
- Ensuring that each process receives its necessary resources, without interfering with other processes.
- Suspending and resuming processes
- Process synchronization and communication
- Deadlock handling

Main-Memory Management

Main memory is a large array of words or bytes, ranging in size from hundreds of thousands to billions. Each word or byte has its own address. Main memory is a repository of quickly accessible data shared by the CPU and I/O devices. The central processor reads instructions from main memory during the instruction-fetch cycle and both reads and writes data from main memory during the data-fetch cycle (at least on a Von Neumann architecture).

UNIT-1 Introduction to Operating System

To improve both the utilization of the CPU and the speed of the computer's response to its users, we must keep several programs in memory, creating a need for memory management. Many different memory-management schemes are used. These schemes reflect various approaches, and the effectiveness of the different algorithms depends on the particular situation. Selection of a memory-management
An OS is responsible for the following tasks with regards to memory management:

• Keeping track of which blocks of memory are currently in use, and by which processes. • Determining which blocks of code and data to move into and out of memory, and when. • Allocating and deallocating memory as needed. (E.g. new, malloc)

File Management

File management is one of the most visible components of an operating system. Computers can store information on several different types of physical media. Magnetic disk, optical disk, and magnetic tape are the most common. Each of these media has its own characteristics and physical organization. Each medium is controlled by a device, such as a disk drive or tape drive, that also has its own unique characteristics. These properties include access speed, capacity, data-transfer rate, and access method (sequential or random).
A **file** is a collection of related information defined by its creator. Commonly, files represent programs (both source and object forms) and data. Data files may be numeric, alphabetic, alphanumeric, or binary. Files may be freeform (for example, text files), or they may be formatted rigidly (for example, fixed fields). Clearly, the concept of a file is an extremely general one.

An OS is responsible for the following tasks with regards to filesystem management:

- Creating and deleting files and directories
- Supporting primitives for manipulating files and directories. (open, flush, etc.) • Mapping files onto secondary storage.
- Backing up files onto stable permanent storage media.

I/O system Management

One of the purposes of an operating system 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 operating system itself by the **I/O subsystem**. The I/O subsystem consists of • A memory-management component that includes buffering, caching, and spooling • A general device-driver interface
• Drivers for specific hardware devices Only the device driver knows the peculiarities of the specific device to which it is assigned.

Secondary-Storage Management

The main purpose of a computer system is to execute programs. These programs, with the data they access, must be in main memory, or **primary storage**, during execution. Because main memory is too small to accommodate all data and programs, and 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 use disks as the principal on-line storage medium for both programs and data. Most programs—including compilers, assemblers, word processors, editors, and formatters—are stored on a disk until loaded into memory and then use the disk as both the source and destination of their processing. Hence, the proper management of disk storage is of central importance to a computer system.

The operating system is responsible for the following activities in connection with disk

management:

- Free-space management
- Storage allocation
- Disk scheduling

Networking

A **distributed system** is a collection of processors that do not share memory, peripheral devices, or a clock. Instead, each processor has its own local memory and clock, and the processors communicate with one another through various communication lines, such as high-speed buses or networks. The processors in a distributed system vary in size and function. They may include small microprocessors, workstations, minicomputers, and large, general-purpose computer systems.

The processors in the system are connected through a **communication network**, which can be configured in a number of different ways. The network may be fully or partially connected. The communication-network design must consider message routing and connection strategies, as well as the problems of contention and security.

A distributed system collects physically separate, possibly heterogeneous, systems into a single coherent system, providing the user with access to the various resources that the system maintains.

Protection System

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

Protection, then, is any mechanism for controlling the access of programs, processes, or users to the resources defined by a computer system. This mechanism must provide means for specification of the controls to be imposed and means for enforcement.

For example, memory-addressing hardware ensures that a process can execute only within its own address space. The timer ensures that no process can gain control of the CPU without eventually relinquishing control. Device-control registers are not accessible to users, so the integrity of the various peripheral devices is protected.

Protection can improve reliability by detecting latent errors at the interfaces between component subsystems. Early detection of interface errors can often prevent contamination of a healthy subsystem by another subsystem that is malfunctioning.

Command-Interpreter System

One of the most important systems programs for an operating system is the **command Interpreter**, which is the interface between the user and the operating system. Some operating systems include the command interpreter in the kernel.

Others, such as MS-DOS and UNIX, treat the command interpreter as a special program that is running when a job is initiated or when a user first logs on (on timesharing systems). Many commands are given to the operating system by **control statements**. When a new job is started in a batch system, or when a user logs onto a time-shared system, a program that reads and interprets control statements is executed automatically.

This program is sometimes called the **control-card interpreter**, or **shell**. Its function is simple: to get the next command statement and execute it.

Operating System Services

UNIT-1 Introduction to Operating System

An operating system provides an environment for the execution of programs. It provides certain services to programs and to the users of those programs. The specific services provided, of course, differ from one operating system to another, but we can identify common classes. These operating-system services are provided for the convenience of the programmer, to make the programming task easier.

One set of operating-system services provides functions that are helpful to the user. • ***Program execution:*** The system must be able to load a program into memory and to run that program. The program must be able to end its execution, either normally or abnormally (indicating error).

• ***I/O operations:*** A running program may require I/O, which may involve a file or an I/O device. For specific devices, special functions may be desired (such as rewinding a tape drive or blanking a CRT screen). For efficiency and protection, users usually cannot control I/O devices directly. Therefore, the operating system must provide a means to do I/O.

• ***File-system manipulation:*** The file system is of particular interest. Obviously, programs need to read and write files and directories. They also need to create and delete them by name, search for a given file, and list file information.

• ***Communications:*** There are many circumstances in which one process needs to exchange information with another process. Such communication may occur between processes that are executing on the same computer or between processes that are executing on different computer systems tied together by a computer network. Communications may be implemented via *shared memory* or through *message passing*, in which packets of information are moved between processes by the operating system.

• ***Error detection:*** The operating system constantly needs to be aware of possible errors. Errors may occur in the CPU and memory hardware (such as a memory error or a power failure), in I/O devices (such as a parity error on tape, a connection failure on a network, or lack of paper in the printer), and in the user program (such as an arithmetic overflow, an attempt to access an illegal memory location, or a too great use of CPU time). For each type of error, the operating system should take the appropriate action to ensure correct and consistent computing.

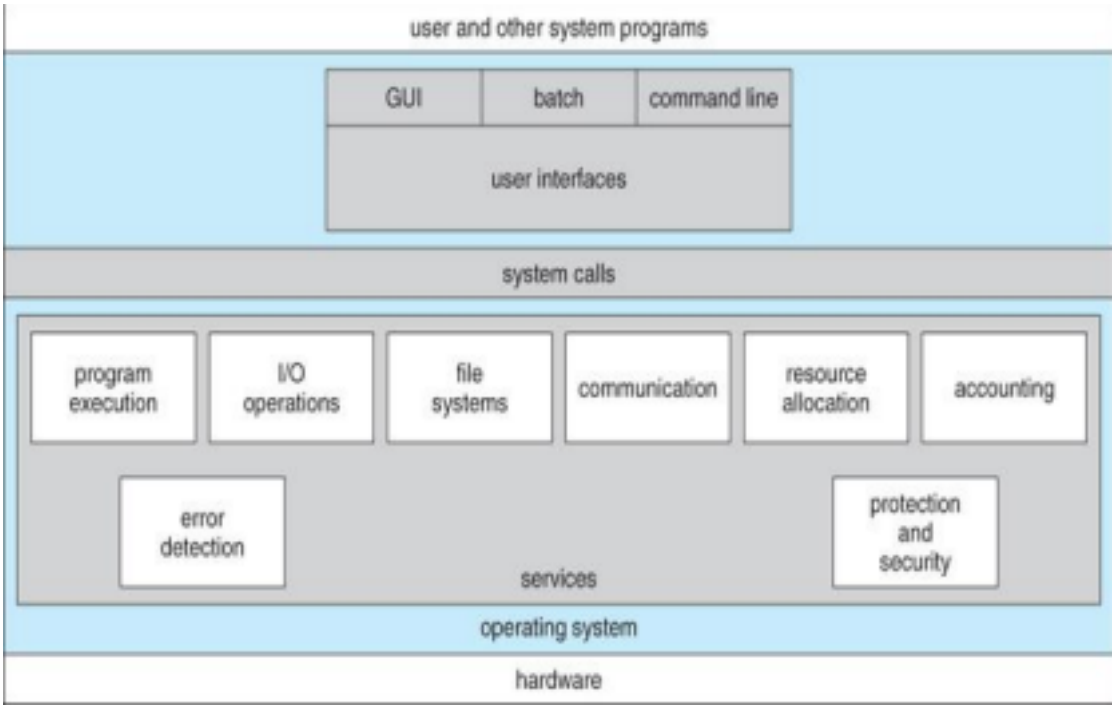


Figure - A view of operating system services

UNIT-1 Introduction to Operating System

Another set of operating-system functions exists not for helping the user but rather for ensuring the efficient operation of the system itself. Systems with multiple users can gain efficiency by sharing the computer resources among the users.

• ***Resource allocation:*** When there are multiple users or multiple jobs running at the same time, resources must be allocated to each of them. Many different types of resources are managed by the operating system. Some (such as CPU cycles, main memory, and file storage) may have special allocation code, whereas others (such as I/O devices) may have much more general request and release code. For instance, in determining how best to use the CPU, operating systems have CPU-scheduling routines that take into account the speed of the CPU, the jobs that must be executed,
the number of registers available, and other factors.

• ***Accounting:*** We want to keep track of which users use how much and what kinds of computer resources. This record keeping may be used for accounting (so that users can be billed) or simply for accumulating usage statistics. Usage statistics may be a valuable tool for researchers who wish to reconfigure the system to improve computing services.

• ***Protection and security:*** The owners of information stored in a multiuser computer system may want to control use of that information. When several disjoint processes execute concurrently, it should not be possible for one process to interfere with the others or with the operating system itself. Protection involves ensuring that all access to system resources is controlled. *Security* of the system from outsiders is also important. Such security starts with requiring each user to authenticate himself or herself to the system, usually by means of a password, to gain access to system resources. It extends to defending external I/O devices, including modems and network adapters, from invalid access attempts and to recording all such connections for detection of break-ins.

SYSTEM CALLS

System calls provide the interface between a process and the operating system. These calls are generally available as assembly-language instructions and are usually listed in the manuals used by assembly-language programmers. Several languages—notably C and C++—have been defined to replace assembly language for systems programming.

Certain systems allow system calls to be made directly from a higher-level language program, in which case the calls normally resemble predefined function or subroutine calls. They may generate a call to a special run-time routine that makes the system call, or the system call may be generated directly in-line.

Six major categories, as outlined in Figure

- Process control
  - end, abort
  - load, execute
  - create process, terminate process
  - get process attributes, set process attributes
  - wait for time
  - wait event, signal event
  - allocate and free memory
- File management
  - create file, delete file
  - open, close
  - read, write, reposition
  - get file attributes, set file attributes
- Device management
  - request device, release device
  - read, write, reposition
  - get device attributes, set device attributes
  - logically attach or detach devices
- Information maintenance
  - get time or date, set time or date
  - get system data, set system data
  - get process, file, or device attributes
  - set process, file, or device attributes
- Communications
  - create, delete communication connection
  - send, receive messages
  - transfer status information

EXAMPLES OF WINDOWS AND UNIX SYSTEM CALLS

	Windows	Unix
Process Control	CreateProcess()	fork()
	ExitProcess()	exit()
	WaitForSingleObject()	wait()
File Manipulation	CreateFile()	open()
	ReadFile()	read()
	WriteFile()	write()
	CloseHandle()	close()
Device Manipulation	SetConsoleMode()	ioctl()
	ReadConsole()	read()
	WriteConsole()	write()
Information Maintenance	GetCurrentProcessID()	getpid()
	SetTimer()	alarm()
	Sleep()	sleep()
Communication	CreatePipe()	pipe()
	CreateFileMapping()	shmget()
	MapViewOfFile()	mmap()
Protection	SetFileSecurity()	chmod()
	InitlializeSecurityDescriptor()	umask()
	SetSecurityDescriptorGroup()	chown()

As an example of how system calls are used, consider writing ***sequence of system call to read data from one file and copy them to another file***.

The first input that the program will need is the names of the two files: the input file and the output file. These names can be specified in many ways, depending on the operating-system design. One approach is for the program to ask the user for the names of the two files. In an interactive system, this approach will require a sequence of system calls, first to write a prompting message on the screen and then to read from the keyboard the characters that define the two files. On mouse-based and icon-based systems, a menu of file names is usually displayed in a window. The user can then use the mouse to select the source name, and a window can be

opened for the destination name to be specified. This sequence would require many I/O system calls.

Once the two file names are obtained, the program must open the input file and create the output file. Each of these operations requires another system call. There are also possible error conditions for each operation. When the program tries to open the input file, it may find that there is no file of that name or that the file is protected against access. In these cases, the program should print a message on the console (another sequence of system calls) and then terminate abnormally (another system call). If the input file exists, then we must create a new output file. We may find that there is already an output file with the same name. This situation may cause the program to abort (a system call), or we may delete the existing file (another system call) and create a new one (another system call). Another option, in an interactive system, is to ask the user (via a sequence of system calls to output the prompting message and to read the response from the terminal) whether to replace the existing file or to abort the program. Now that both files are set up, we enter a loop that reads from the input file (a system call) and writes to the output file (another system call). Each read and write must return status information regarding various possible error conditions. On input, the program may find that the end of the file has been reached or that there was a hardware failure in the read (such as a parity error). The write operation may Encounter various errors, depending on the output device (no more disk space, physical end of tape, printer out of paper, and so on).

Finally, after the entire file is copied, the program may close both files (another system call), write a message to the console or window (more system calls), and finally terminate normally (the final system call). As we can see, programs may make heavy use of the operating system. Frequently, systems execute thousands of system calls per second.

System calls can be grouped roughly into five major categories: **process control, file manipulation, device manipulation, information maintenance**, and **communications**.

Process Control

A running program needs to be able to halt its execution either normally (`end`) or abnormally (`abort`). If a system call is made to terminate the currently running program abnormally, or if the program runs into a problem and causes an error trap, a dump of memory is sometimes taken and an error message generated. The dump is written to disk and may be examined by a *debugger*. A process or job executing one program may want to `load` and `execute` another program. we have created a new job or process to be multiprogrammed. Often, there is a system call specifically for this purpose (`create process` or `submit job`). If we create a new job or process, or perhaps even a set of jobs or processes, we should be able to control its execution. This control requires the ability to determine and reset the attributes of a job or process, including the job's priority, its maximum allowable execution time, and so on (`get process attributes` and `set process attributes`). We may also want to terminatea job or process that we created (`terminate process`) if we find that it is incorrect or is no longer needed We may want to wait for a certain amount of time (`wait time`); more probably, we will want to wait for a specific event to occur (`wait event`). The jobs or processes should then signal when that event has occurred (`signal event`).

File Management

We can identify several common system calls dealing with files, however,We first need to be able to `create` and `delete` files. Either system call requires the name of the file and perhaps some of the file's attributes. Once the file is created, we need to `open` it and to use it. We may also `read`, `write`, or `reposition` (rewinding or skipping to the end of the file, for example). Finally, we need to `close` the file, indicating that we are no longer using it.

We may need these same sets of operations for directories if we have a directory structure for organizing files in the file system. In addition, for either files or directories, we need to be able to

determine the values of various attributes and perhaps to reset them if necessary. File attributes include the file name, a file type, protection codes, accounting information, and so on. At least two system calls, `get file attribute` and `set file attribute`, are required for this function.

Device Management

A program, as it is running, may need additional resources to proceed—more memory, tape drives, access to files, and so on. If the resources are available, they can be granted, and control can be returned to the user program. Otherwise, the program will have to wait until sufficient resources are available.

If there are multiple users of the system, however, we must first `request` the device, to ensure exclusive use of it. After we are finished with the device, we must `release` it. These functions are similar to the `open` and `close` system calls for files. Once the device has been requested (and allocated to us), we can `read`, `write`, and (possibly) `reposition` the device, just as we can with ordinary files

Information Maintenance

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

Communication

There are two common models of communication: the message-passing model and the shared memory model. In the **message-passing model**, information is exchanged through an interprocess-communication facility provided by the operating system. Before communication can take place, a connection must be opened. The name of the other communicator must be known, be it another process on the same system or a process on another computer connected by a communications network. Using *host name*, such as an IP name, by which it is commonly known or a *process name*, which is translated into an identifier by which the operating system can refer to it. The `get hostid` and `get processid` system calls do this translation.

These identifiers are then passed to the general-purpose `open` and `close` calls provided by the file system or to specific `open connection` and `close connection` system calls, depending on the system's model of communications.

In the **shared-memory model**, processes use `map memory` system calls to gain access to regions of memory owned by other processes. Recall that, normally, the operating system tries to prevent one process from accessing another process's memory. Shared memory requires that two or more processes agree to remove this restriction. They may then exchange information by reading and writing data in the shared areas.

Communications models. (a) Message passing (b) Shared memory.

System Programs

Another aspect of a modern system is the collection of system programs. The logical computer hierarchy depicts the lowest level as hardware. Next is the operating system, then the system programs, and finally the application programs. System programs provide a convenient environment for program development and execution. Some of them are simply user interfaces to system calls; others are considerably more complex. They can be divided into these categories:

• *File management:* These programs create, delete, copy, rename, print, dump, list, and generally manipulate files and directories.
• *Status information:* Some programs simply ask the system for the date, time, amount of available memory or disk space, number of users, or similar status information. That information is then formatted and printed to the terminal or other output device or file.
• *File modification:* Several text editors may be available to create and modify the content of files stored on disk or tape.
• *Programming-language support:* Compilers, assemblers, and interpreters for common programming languages (such as C, C++, Java, Visual Basic, and PERL) are often provided to the user with the operating system, although some of these programs are now priced and provided separately.
• *Program loading and execution:* Once a program is assembled or compiled, it must be loaded into memory to be executed. The system may provide absolute loaders, relocatable loaders, linkage editors, and overlay loaders. Debugging systems for either higher-level languages or

machine language are needed also.
• *Communications:* These programs provide the mechanism for creating virtual connections among processes, users, and computer systems. They allow users to send messages to one another's screens, to browse web pages, to send electronic mail messages, to log in remotely, or to transfer files from one machine to another.

In addition to systems programs, most operating systems are supplied with programs that are useful in solving common problems or performing common operations. Such programs include web browsers, word processors and text formatters, spreadsheets, database systems, compilers, plotting and statistical-analysis packages, and games. These programs are known as **system utilities** or **application programs**.

UNIT-1 Introduction to Operating System

Operating System Structure

A system as large and complex as a modern operating system must be engineered carefully if it is to function properly and be modified easily. A common approach is to partition the task into small components rather than have one monolithic system.
Each of these modules should be a well-defined portion of the system, with carefully defined inputs, outputs, and functions.

Simple Structure

Many commercial systems do not have well-defined structures. Frequently, such operating systems started as small, simple, and limited systems and then grew beyond their original scope. MS-DOS is an example of such a system. It was originally designed and implemented by a few people who had no idea that it would become so popular. It was written to provide the most functionality in the least space (because of the limited hardware on which it ran), so it was not divided into modules carefully.

MS-DOS layer structure.

In MS-DOS, the interfaces and levels of functionality are not well separated. For instance, application programs are able to access the basic I/O routines to write directly to the display and disk drives. Such freedom leaves MS-DOS vulnerable to errant (or malicious) programs, causing entire system crashes when user programs fail.

Another example of limited structuring is the original UNIX operating system. UNIX is another system that initially was limited by hardware functionality. It consists of two separable parts: the

kernel and the system programs. The kernel is further separated into a series of interfaces and device drivers, which have been added and expanded over the years as UNIX has evolved. We can view the traditional UNIX operating system as being layered, as shown in Figure. Everything below the system-call interface and above the physical hardware is the kernel. The kernel provides the file system, CPU scheduling, memory management, and other operating-system functions through system calls. System calls define the **application programmer interface (API)** to UNIX; the set of system programs commonly available defines the **user interface**. The programmer and user interfaces define the context that the kernel must support.

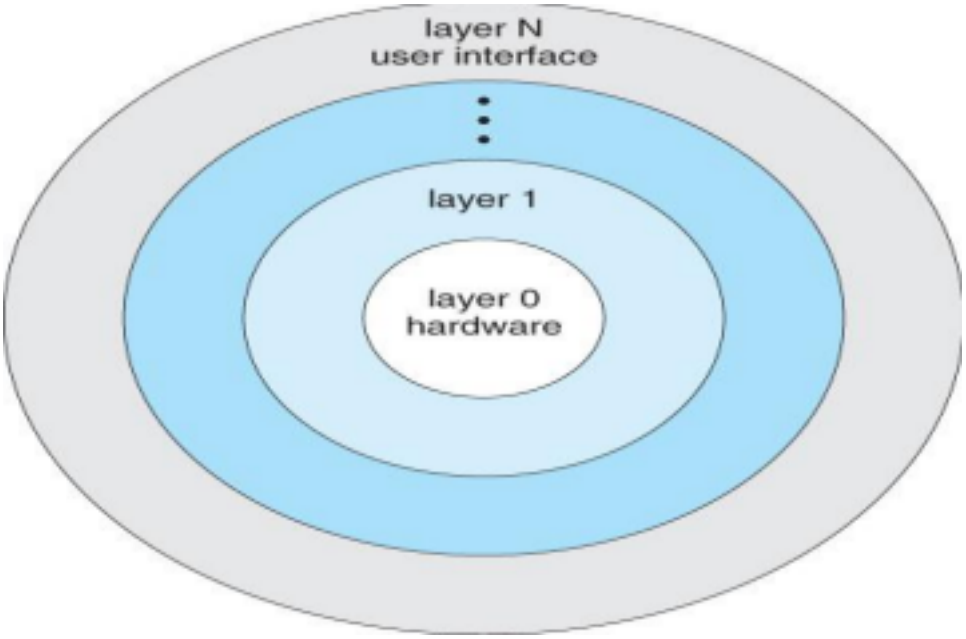
UNIX system structure

Layered Approach

A system can be made modular in many ways. One method is the **layered approach**, in which the operating system is broken up into a number of layers (levels). The bottom layer (layer 0) is the hardware; the highest (layer *N*) is the user interface.
A typical operating-system layer—say, layer *M*—is depicted in Figure. It consists of data structures and a set of routines that can be invoked by higher-level layers. Layer *M*, in turn, can invoke operations on lower-level layers. The main advantage of the layered approach is **modularity**. The layers are selected so that each uses functions (operations) and services of only lower-level layers.

The first layer can be debugged without any concern for the rest of the system, because, by definition, it uses only the basic hardware (which is assumed correct) to implement its functions. Once the first layer is debugged, its correct functioning can be assumed while the second layer is debugged, and 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 and implementation of the system is simplified when the system is broken down into layers. Each layer is implemented with only those operations provided by lower-level layers. A layer does not need to know how these operations are implemented; it needs to know only what these operations do.
The major difficulty with the layered approach involves appropriately defining the various layers.

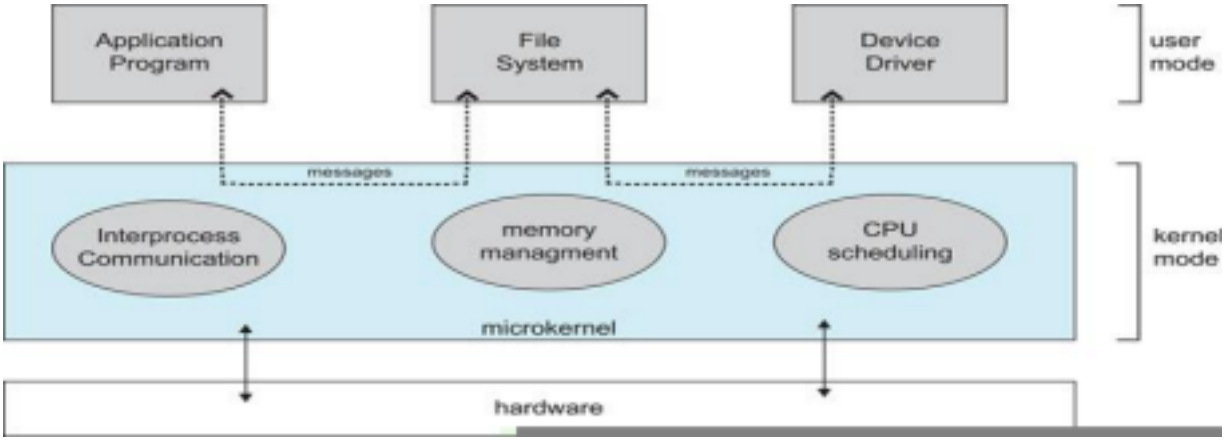
Because a layer can use only lower-level layers, careful planning is necessary. For example, the device driver for the backing store (disk space used by virtual-memory algorithms) must be at a level lower than that of the memory management routines, because memory management requires the ability to use the backing store. A final problem with layered implementations is that they tend to be less efficient than other types. For instance, when a user program executes an I/O operation, it executes a system call that is trapped to the I/O layer, which calls the memory management layer, which in turn calls the CPU-scheduling layer, which is then passed to the hardware.



Microkernels

As UNIX expanded, the kernel became large and difficult to manage. In the mid- 1980s, researchers at Carnegie Mellon University developed an operating system called **Mach** that modularized the kernel using the **microkernel** approach. This method structures the operating system by removing all nonessential components from the kernel and implementing them as system and user-level programs. The result is a smaller kernel. The main function of the microkernel is to provide a communication facility between the client program and the various services that are also running in user space. Communication is provided by *message passing*, For example, if the client program wishes to access a file, it must interact with the file server. The client program and service never interact directly. Rather, they communicate indirectly by exchanging messages with the microkernel.

The benefits of the microkernel approach include ease of extending the operating system. All new services are added to user space and consequently do not require modification of the kernel. When the kernel does have to be modified, the changes tend to be fewer, because the microkernel is a smaller kernel. The resulting operating system is easier to port from one hardware design to another. The microkernel also provides more security and reliability, since most services are running as user—rather than kernel—processes. If a service fails, the rest of the operating system remains untouched.



Architecture of a typical microkernel

Modules

Perhaps the best current operating-system-design methodology involves the use of object oriented programming techniques to create a modular kernel. Here, the kernel has a set of core components and dynamically links in additional services either during boot time or during run time. Such a strategy uses dynamically loadable modules and is common in modern implementations of UNIX such as Solaris, Linux, and Mac OS X. For example, the Solaris operating system structure, shown in Figure, is organized around a core kernel with seven types of loadable kernel modules:

1. Scheduling classes
2. File systems
3. Loadable system calls
4. Executable formats
5. STREAMS modules
6. Device and bus drivers
7. Miscellaneous

Such a design allows the kernel to provide core services yet also allows certain features to be implemented dynamically. For example, device and bus drivers for specific hardware can be added to the kernel, and support for different file systems can be added as loadable modules.

Solaris loadable modules.

The overall result resembles a layered system in that each kernel section has defined, protected interfaces; but it is more flexible than a layered system in that any module can call any other module. Furthermore, the approach is like the microkernel approach in that the primary module has only core functions and knowledge of how to load and communicate with other modules; but it is more efficient, because modules do not need to invoke message passing in order to

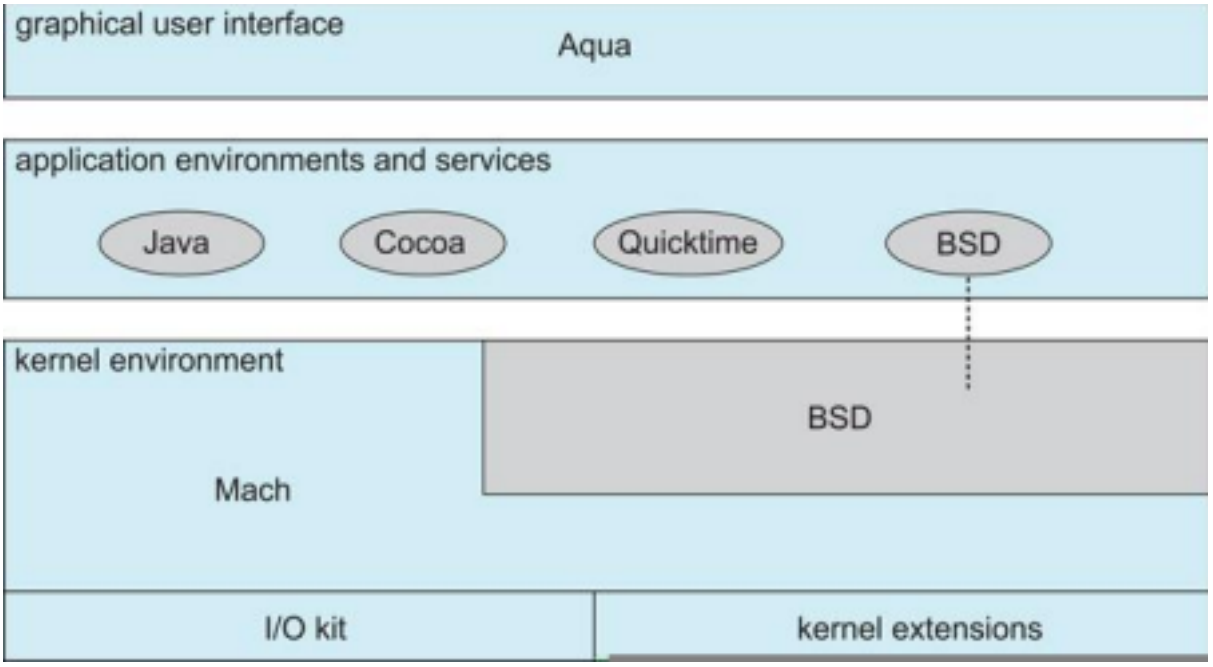
communicate.

Hybrid Systems

Most OSes today do not strictly adhere to one architecture, but are hybrids of several.

Mac OS X

The Max OSX architecture relies on the Mach microkernel for basic system management services, and the BSD kernel for additional services. Application services and dynamically loadable modules (kernel extensions) provide the rest of the OS functionality:



The Mac OS X structure

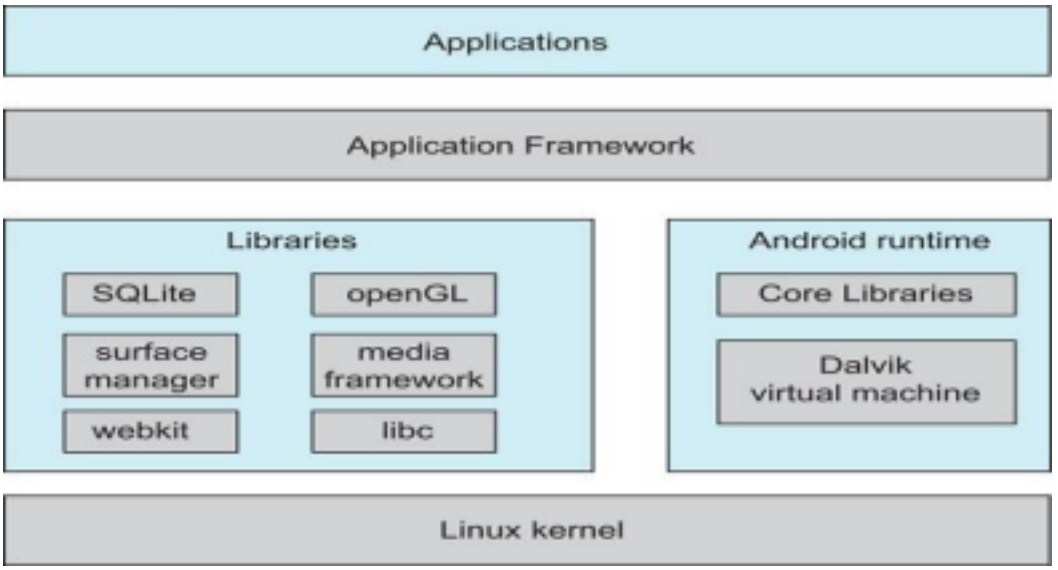
IOS

The **iOS** operating system was developed by Apple for iPhones and iPads. It runs with less memory and computing power needs than Max OS X, and supports touchscreen interface and graphics for small screens:

Architecture of Apple's iOS.

Android

- The Android OS was developed for Android smartphones and tablets by the Open Handset Alliance, primarily Google.
- Android is an open-source OS, as opposed to iOS, which has lead to its popularity. • Android includes versions of Linux and a Java virtual machine both optimized for small platforms.
- Android apps are developed using a special Java-for-Android development environment.



Architecture of Google's Android

Difference Between Spooling and Buffering

Spooling

Spooling is a technique to minimize the problems due to slowness of input /output devices and share the system resources to complete the process efficiently. The name is an acronym for "**S**imultaneous **P**eripheral **O**peration **O**n-**L**ine". Spooling essentially uses the disk as a very large

buffer for reading as far ahead as possible on input device and for storing output files until the output devices are able to accept them.
In a disk system cards are read directly from the card reader on to the disk. The location of the card images is recorded in a table kept by the os. Each job is noted in the table as it is read in. When a job is executed, it request or card reader input and are satisfied reading from the disk. Similarly when the job requests the printer to output a line, that line is copied in to the system buffer and written to the disk. When the job is completed, the output is actually printed. This form of processing is called spooling.

Buffering

Buffering attempts to keep both the CPU and I/O devices busy all the time. The idea is quite simple. After data has been read and the CPU is about to start operating on it, the input device is instructed to begin the next input immediately. The CPU and the input device are both busy. By time the CPU is ready for the next data item, the input device will have finished reading it. The CPU can then begin processing the newly read data, while the input device starts to read the following data. Similarly buffering can be done for output.

UNIT-1 Introduction to Operating System

"Buffering overlaps the input output of a job with its computation. The advantage of spooling over buffering is that spooling overlaps the input/output of one job with the computations of other job. Even in a simple system the spooler may be reading the input of one job while printing output of different job. During this time still another job may be executed reading their cards from disk and printing their output line on to the disk".

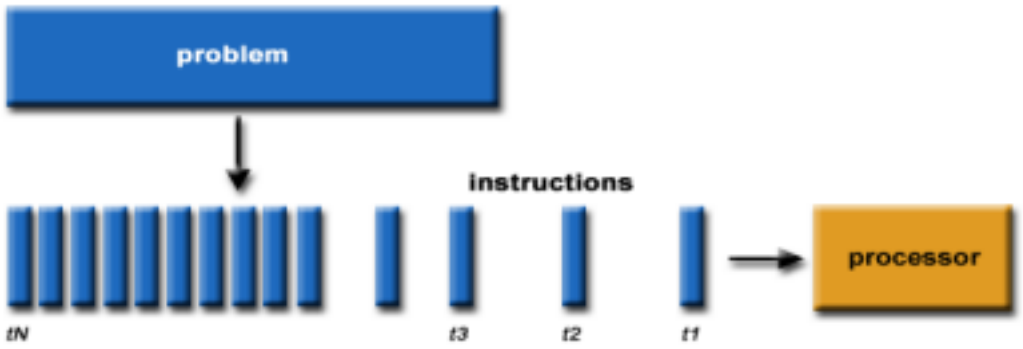
Basis of Distinction	**Spooling in OS**	**Buffering in OS**
Definition	A process where the data temporary becomes available by holding it and then used and executed by either a device, system or a program by the help of request for executions.	A process where the specified region exists that holds all the data on a temporary basis and helps it to move from one location to the other.
Operation	The overlapping of the input and output of one task with the calculation of the other task.	The overlapping of comments and output of one function with the count of the same task.
Name	Simultaneous peripheral operation online	No name.
Memory	A large area in the hard disk exists.	A small area with limited scope.

Introduction to Parallel Computation

Parallel computing is a type of computation in which many calculations or the execution of processes are carried out simultaneously.[1] Large problems can often be divided into smaller ones, which can then be solved at the same time. There are several different forms of parallel computing: bit-level, instruction-level, data, and task parallelism.

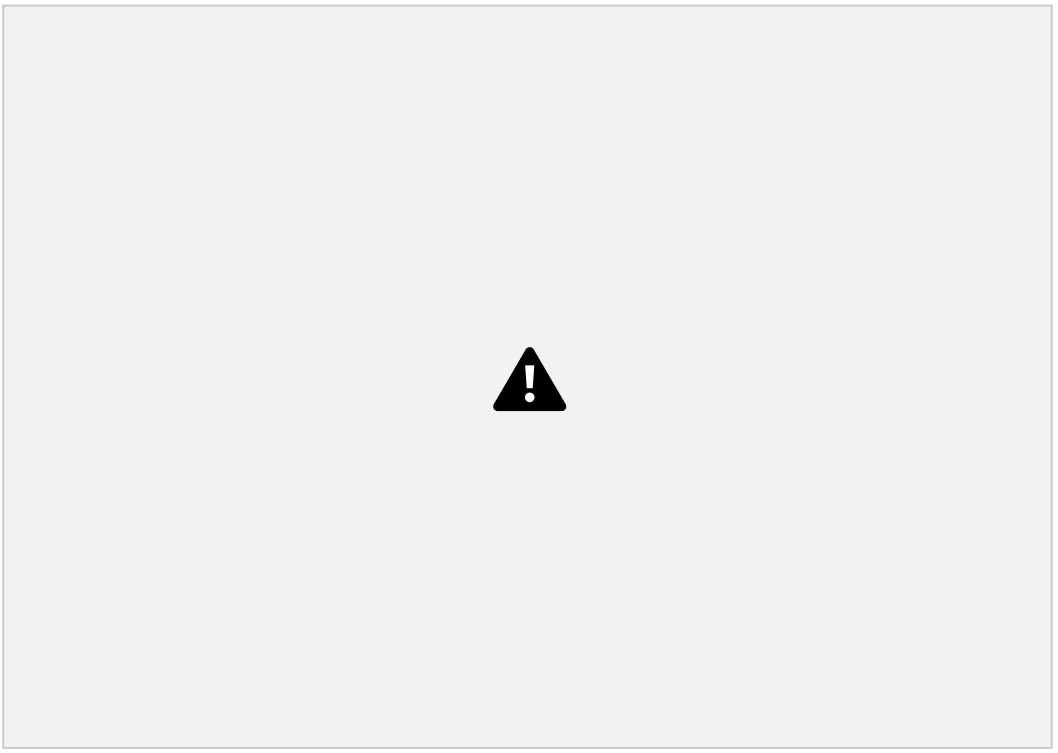
Serial Computing:
Traditionally, software has been written for serial computation:

- A problem is broken into a discrete series of instructions
- Instructions are executed sequentially one after another
- Executed on a single processor
- Only one instruction may execute at any moment in time



Parallel Computing:

- In the simplest sense, ***parallel computing*** is the simultaneous use of multiple compute resources to solve a computational problem:
  - A problem is broken into discrete parts that can be solved concurrently
  - Each part is further broken down to a series of instructions
  - Instructions from each part execute simultaneously on different processors
  - An overall control/coordination mechanism is employed



For example:

- The computational problem should be able to:
  - Be broken apart into discrete pieces of work that can be solved simultaneously;
  - Execute multiple program instructions at any moment in time;
  - Be solved in less time with multiple compute resources than with a single compute resource.
- The compute resources are typically:
  - A single computer with multiple processors/cores
  - An arbitrary number of such computers connected by a network