Module – 1
Introduction to Operating Systems

Module 1 Question bank
[2018]

1 a. Define Operating System. with a neat diagram, explain the dual-mode operation of operating system. (06 Marks)
b. Explain the services of operating system that are helpful for user and the system. (06 Marks)
c. Define the following terms :
i) Virtual Machine
ii) CPU scheduler
iii) System call
iv) Context switch. (04 Marks)

OR

2 a. With a neat diagram, explain the different states of a process. (05 Marks)
b. Explain the layered approach of operating system structure, with supporting diagram. (05 Marks)
c. What is interprocess communication? Explain direct and indirect communication with respect to message passing system. (06 Marks)

1 a. What is an operating system? Explain abstract view of component of a computer system. (07 Marks)
b. List the different services that an operating system provides. Explain. (06 Marks)
c. Explain the concept of virtual machines. Bring out its advantages. (07 Marks)

2 a. What is a process? With a state diagram, explain states of a process. (06 Marks)
b. Describe the implementation of IPC using shared memory and message passing. (07 Marks)

[2017]

1 a. Distinguish between the following pairs of terms :
i) Symmetric and asymmetric multiprocessor systems
ii) Cpu burst and I/O burst jobs
iii) User's view and systems view of OS
iv) Batch systems and time sharing systems
v). User mode and kernel mode operations. (10 Marks)
b. List the three main advantages of multiprocessor systems. Also bring out the difference between graceful degradation and fault tolerance in this context. (05 Marks)
c. What are virtual machines? How are they implemented? (05 Marks)

2 a. What is a process? What are the states a process can be in? Give the process state diagram clearly indicating the conditions for a process to shift from one state to another. (08 Marks)
b. What are the merits of inter process communication? Name the two major models of inter process communication. (06 Marks)

1 a. What is operating system? Explain multiprogramming and time sharing systems. **(06 Marks)**
b. Explain dual mode operation in OS with a neat block diagram. **(04 Marks)**
c. What are system calls? Briefly point out its types. **(04 Marks)**
d. What are virtual machines? Explain with block diagram. Point out its benefits. **(06 Marks)**

2 a. Why is it important for the scheduler to distinguish I/O bound programs from CPU bound programs? **(02 Marks)**
b. What is interprocess communication? Explain its types. **(06 Marks)**

[2016]

1 a. List and explain the functions and services of an operating system and OS operations. **(08 Marks)**
b. What are virtual machines? Explain VM-WARE architecture with a neat diagram. **(08 Marks)**
c. Differentiate between multiprogramming, multiprocessing and multitasking systems. **(04 Marks)**

2 a. Explain process states with state transition diagram. Also explain PCB with a neat diagram. **(08 Marks)**
b. What is IPC? Explain Direct and Indirect communication with respect to message passing systems. **(05 Marks)**

[2015]

1. Differentiate between multiprogramming and multiprocessing [5M]
2. Explain various functions of OS wrt process and memory management [5M]
3. List the different services that an OS provides. Explain any 2 [6M]
4. What are the different categories of a system program. Explain [6M]
5. With a neat diagram, explain the components of PCB [5M]
6. Explain the direct and indirect communication wrt message passing system.

What Operating Systems Do

An operating system is a program that manages the computer hardware. It provides a basis for application program & acts as an intermediary between user of computer & computer hardware. The purpose of an OS is to provide an environment in which the user can execute the program in a convenient & efficient manner. OS is an important part of almost every computer systems.

An OS has 3 major goals

- Execute user programs and make solving user problems easier
- Make the computer system convenient to use
- Use the computer hardware in an efficient manner

A computer system can be roughly divided into four components

1. The Hardware
2. The OS
3. The application Program
4. The user

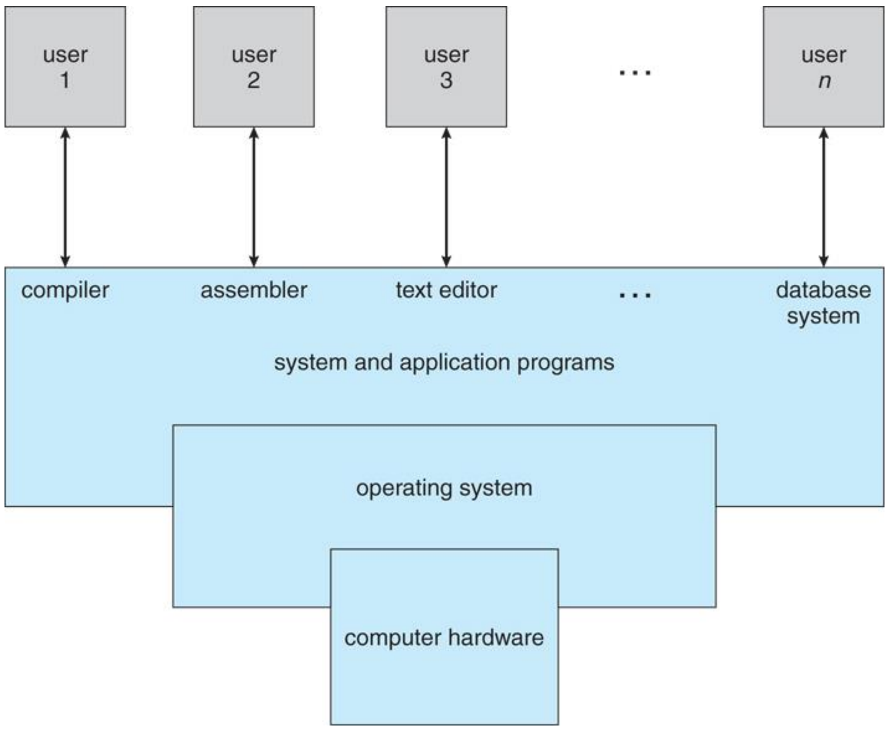


Figure 1: Abstract view of the components of a computer system

The Hardware consists of memory, CPU, I/O devices, peripherals devices & storage devices- which provides the basic computing resources for the system. The application programs such as word processors, spread sheets, compilers & web browsers etc define the ways in which these resources are used to solve users computing problems.

The OS controls the hardware & co-ordinates its use among various application programs for various users. *To understand the operating system's role completely, we explore it* ***from two different viewpoints****: that of the user and that of the system.*

User View: The user view of the computer depends on the interface used.

1. Most users use a PC: Such systems are designed for one user to monopolize its resources. Here the operating system is designed mostly for **ease of use**, with some attention paid to performance and none paid to resource utilization.

2. Some users may use a terminal connected to a mainframe or minicomputers: The operating system in such cases is designed to **maximize resource utilization**.

3. Users of workstations connected to networks of other workstations and servers: In this case the OS is designed to compromise between **individual usability** and **resource utilization**.

System Views:

4. We can view the operating system as a **resource allocator.** A computer system has many resources (like CPU Time, memory space, file storage space, I/O devices) that may be required to solve a problem. The OS acts as a manager of these resources. The OS must decide how to allocate these resources to programs and the users so that it can operate the computer system efficiently and fairly.

5. A different view of an OS is that it need to control various I/O devices & user programs i.e. an OS is a control program used to manage the execution of user program to prevent errors and improper use of the computer.

Computer-System Organization

Refer Text Book

Key terms- Bootstrap program, event driven or interrupt driven execution, interrupt driven I/O and direct memory access (DMA).

Computer-System Architecture

The organization of a computer system can be categorized according to the number of general-purpose processors used.

Single-Processor Systems

- A single-processor system, has one main CPU capable of executing a general-purpose instruction set, including instructions from user processes.
- Most systems also have special-purpose processors (device-specific processors, graphics controllers, etc) for performing device specific tasks.
- These special-purpose processors run a limited instruction set and do not run user processes. Sometimes they are managed by the operating system, in that the operating system sends them information about their next task and monitors their status.
- The use of special-purpose microprocessors does not turn a single-processor system into a multiprocessor. If there is only one general-purpose CPU, then the system is a single-processor system.

Multiprocessor Systems

- **Multiprocessor systems** (also known as **parallel systems** or **tightly coupled systems)** have two or more processors in close communication, sharing the computer bus and sometimes the clock, memory, and peripheral devices.

  Multi processor systems are of two types.

- Asymmetric Multi processors- 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*.

- Symmetric Multi processors (SMP)- Each processor performs all tasks within the operating system. SMP means that all processors are *peers*; no master-slave relationship exists between processors.
  Figure 2 illustrates a typical SMP architecture.

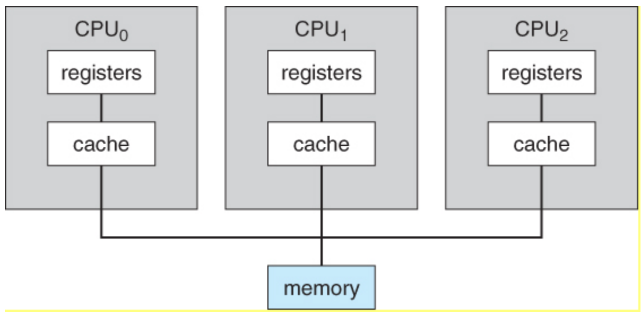


Figure 2: Symmetric multiprocessing architecture

Multiprocessor systems have three main advantages:

1. **Increased throughput:**
   - By increasing the number of processors, we expect to get *more work done in less time*.
   - The speed-up ratio with *N* processors is not *N*, however; rather, it is less than N. When multiple processors cooperate on a task, a certain amount of overhead is incurred in keeping all the parts working correctly. This *overhead, plus contention for shared resources, lowers the expected gain from additional processors*.
2. **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 than to have many computers with local disks and many copies of the data.
6. **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*. If we have ten processors and one fails, then each of the *remaining nine processors can pick up a share of the work of the failed processor*.

Increased reliability of a computer system is crucial in many applications. *The ability to continue providing service proportional to the level of surviving hardware is called* ***graceful degradation*****.**

Some systems go beyond graceful degradation and are called ***fault tolerant,*** *because they can suffer a failure of any single component and still continue operation*. Fault tolerance requires a mechanism to allow the failure to be detected, diagnosed, and, if possible, corrected.

Clustered Systems

- Clustered systems gather together multiple CPUs to accomplish computational work. Clustered systems are composed of *two or more individual systems coupled together*.
- Clustered computers share storage and are closely linked via a **local-area network (LAN)** or a faster interconnect such as InfiniBand.
- Clustering is usually used to provide **high-availability** service; that is, service will continue even if one or more systems in the cluster fail.
- *Each node can monitor one or more of the others* (over the LAN). If the monitored machine fails, the monitoring machine can take ownership of its storage and restart the applications that were running on the failed machine. The users and clients of the applications see only a brief interruption of service.

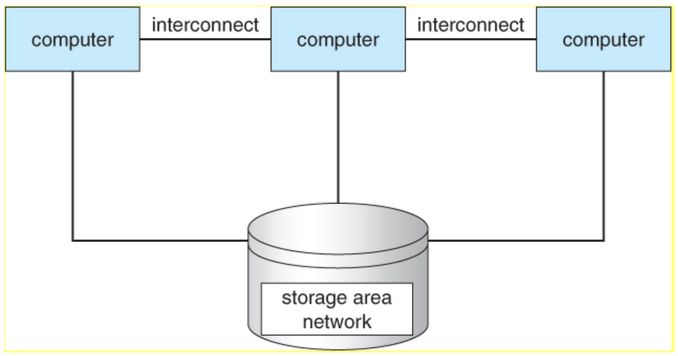


Figure 3: General structure of a clustered system

Clustered systems can be asymmetric or symmetric.

- **Asymmetric clustering** -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-** Two or more hosts are running applications, and are monitoring each other. This mode is more efficient, as it uses all of the available hardware.

OPERATING SYSTEM ARCHITECTURE

- An operating system provides an environment within which programs are executed. One of the most important aspects of operating systems is the ability to multi-program.
- **Multiprogramming** increases CPU utilization by organizing jobs (code and data) so that the CPU always has one to execute.
- The operating system keeps several jobs in memory simultaneously (as shown in Figure 4). This set of jobs can be a subset of jobs kept in the job pool which contains all jobs that enter the system.
- The operating system picks and begins to execute one of the jobs in memory. Eventually, the job may have to wait for some task, such as an I/O operation, to complete.

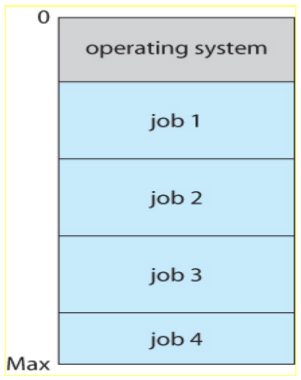


Figure 4: Memory layout for a multiprogramming system.

- In a non-multi-programmed system, the CPU would sit idle. In a multi-programmed system, the operating system simply switches to, and executes, another job. When *that* job needs to wait, the CPU is switched to *another* job, and so on. Eventually, the first job finishes waiting and gets the CPU back. As long as at least one job needs to execute, the CPU is never idle.
- Multi-programmed systems provide an environment in which the various system resources are utilized effectively, but they do not provide for user interaction with the computer system.

Time sharing (or multitasking) is a logical extension of multiprogramming. 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.

- Time sharing requires an **interactive** (or **hands-on) computer system,** which provides direct communication between the user and the system. The user gives instructions to the operating system or to a program directly, using an input device such as a keyboard or a mouse, and waits for immediate results on an output device. Accordingly, the **response time** should be short—typically less than one second.
- A time-shared operating system *allows many users to share the computer simultaneously*. Since each action or command in a time-shared system tends to be short, only a little CPU time is needed for each user. As the system switches rapidly from one user to the next, each user is given the impression that the entire computer system is dedicated to his use, even though it is being shared among many users.

- Time-sharing and multiprogramming require several jobs to be kept simultaneously in memory. *Since in general main memory is too small to accommodate all jobs, the jobs are kept initially on the disk in the* ***job pool*****.** This pool consists of all processes residing on disk awaiting allocation of main memory.

If *several jobs are ready* to be brought into memory, and if there is not enough room for all of them, then the *system must choose among them*. Making this decision is **job scheduling**. Having several programs in memory at the same time requires some form of memory management. If several jobs are ready to run at the same time, the system must choose among them. Making this decision is **CPU scheduling**.

In a *time-sharing system, the operating system must ensure reasonable response time*, which is sometimes accomplished through ***swapping,*** *where processes are swapped in and out of main memory to the disk*.

OPERATING SYSTEM OPERATIONS

- Modern operating systems are **interrupt driven or event driven.** Events are almost always signalled by the occurrence of an interrupt or a trap. *A **trap (or** an **exception)** is a software-generated interrupt* caused either by an error (for example, division by zero or invalid memory access) or by a specific request from a user program that an operating-system service be performed.

- *For each type of interrupt*, separate segments of code in the operating system determine what action should be taken. An *interrupt service routine* is provided that is responsible for dealing with the interrupt.

- Since the operating system and the users share the hardware and software resources of the computer system, we need to *make sure that an error in a user program does not cause problems to other running programs*. For example, if a process gets stuck in an infinite loop, this loop could prevent the correct operation of many other processes.

- Without protection against these sorts of errors, either the computer must execute only one process at a time or all output must be suspect. *A properly designed operating system must ensure that an incorrect (or malicious) program cannot cause other programs to execute incorrectly*.

Dual-Mode Operation

- In order to ensure the proper execution of the operating system, we must be able to *distinguish between the execution of operating-system code and user-defined code.* At the very least, we need two separate **modes** of operation: **user mode** and **kernel mode** (also called **supervisor mode, system mode, or privileged mode).** A bit, called the **mode bit,** is added to the hardware of the computer to indicate the current mode: kernel (0) or user (1). With the mode bit, we are able to distinguish between a task that is executed on behalf of the operating system and one that is executed on behalf of the user.

- When the computer system is executing on behalf of a user application, the system is in user mode. However, *when a user application requests a service from the operating system (via a system call), it must transition from user to kernel mode to fulfil the request*. This is shown in Figure.

- At system boot time, the hardware starts in kernel mode. The operating system is then loaded and starts user applications in user mode. *Whenever a trap or interrupt occurs, the hardware switches from user mode to kernel mode* (that is, changes the state of the mode bit to 0). Thus, whenever the operating system gains control of the computer, it is in kernel mode. The system always switches to user mode (by setting the mode bit to 1) before passing control to a user program.

- The dual mode of operation provides us with the means for protecting the operating system from errant users—and errant users from one another. We accomplish this protection by designating some of the machine instructions that may cause harm as **privileged instructions.**

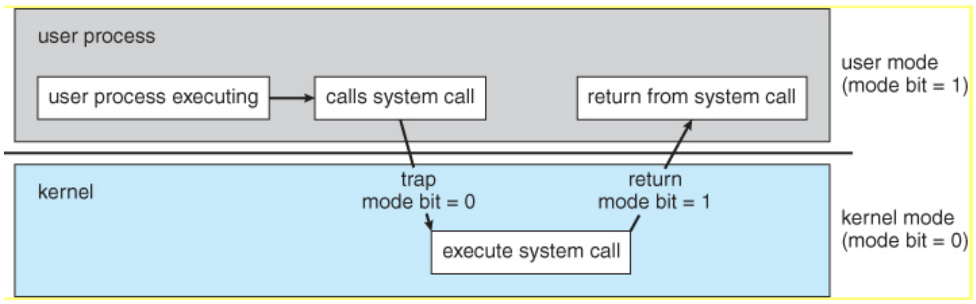


Figure 5: Transition from user to kernel mode

- *The hardware allows privileged instructions to be executed only in kernel mode. If an attempt is made to execute a privileged instruction in user mode, the hardware does not execute the instruction but rather treats it as illegal and traps it to the operating system.*

Timer

- We must *ensure that the operating system maintains control over the CPU*. We must prevent a user program from getting stuck in an infinite loop or not calling system services 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.

- *Before turning over control to the user, the operating system ensures that the timer is set to interrupt*. If the timer interrupts, control transfers automatically to the operating system, which may treat the interrupt as a fatal error or may give the program more time. Thus, we can use the timer to prevent a user program from running too long.

Process Management

- A program is a passive entity. *A program in execution is a process* (active entity).
- A *process requires certain resources* like CPU time, memory, files, and I/O devices—*to accomplish its task*. These resources are allocated to the process when it is created or while it is running. When the process terminates, the operating system will reclaim any reusable resources.
- The operating system is responsible for the following activities in connection with process management:

  → Scheduling processes and threads on the CPUs
  → Creating and deleting both user and system processes
  → Suspending and resuming processes
  → Providing mechanisms for process synchronization
  → Providing mechanisms for process communication

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.
- 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 (on a Von Neumann architecture). The main memory is generally the only large storage device that the CPU is able to address and access directly.
- *To improve both the utilization of the CPU and the speed of the computer's response to its users*, *general-purpose computers must keep several programs in memory, creating a need for memory management.*

The operating system is responsible for the following activities in connection with memory management:

→ Keeping track of which parts of memory are currently being used and by whom
→ Deciding which processes (or parts thereof) and data to move into and out of memory
→ Allocating and deallocating memory space as needed

Storage Management

To make the computer system convenient for users, the operating system provides a uniform, logical view of information storage. The operating system abstracts from the physical properties of its storage devices to define a logical storage unit, the file. The operating system maps files onto physical media and accesses these files via the storage devices.

1) File-System Management

- Computers can store information on several different types of physical media (Magnetic disk, optical disk, magnetic tape) with each having 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: access speed, capacity', data-transfer rate, and access method (sequential or random).
- The operating system implements the abstract concept of a file from the data stored in mass storage media, such as tapes and disks, and the devices that control them. Also, files are normally organized into directories to make them easier to use.
- A file is a collection of related information defined by its creator.

The operating system is responsible for the following activities in connection with file management:

→ 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 (nonvolatile) storage media

2) Mass-Storage Management

- As main memory is volatile and is too small to accommodate all data and programs, the computer system must provide secondary storage to back up main memory.

Because secondary storage is used frequently, it must be used efficiently. The entire speed of operation of a computer may hinge on the speeds of the disk subsystem and of the algorithms that manipulate that subsystem.

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

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

3) Caching

- *Caching is the process of moving data to a faster storage temporarily to speed up data access*. When CPU needs a piece of information, it first checks the cache. If it is, it uses the information directly from the cache; else, the information is fetched from the source, putting a copy in the cache under the assumption that we will need it again soon.

- *Because caches have limited size, cache management is an important design problem*. Careful selection of the cache size and of a replacement policy can result in greatly increased performance.

- Main memory can be viewed as a fast cache for secondary storage, since data in secondary storage must be copied into main memory for use, and data must be in main memory before being moved to secondary storage for safekeeping.

- *In a hierarchical storage structure, the same data may appear in different levels of the storage system.* For example, suppose that an integer A that is to be incremented by 1 is located in file B, and file B resides on magnetic disk. The increment operation proceeds by first issuing an I/O operation to copy the disk block on which A resides to main memory. This operation is followed by copying A to the cache and to an internal register. Thus, the copy of A appears in several places: on the magnetic disk, in main memory, in the cache, and in an internal register (see Figure 6). Once the increment takes place in the internal register, the value of A differs in the various storage systems. The value of A becomes the same only after the new value of A is written from the internal register back to the magnetic disk.

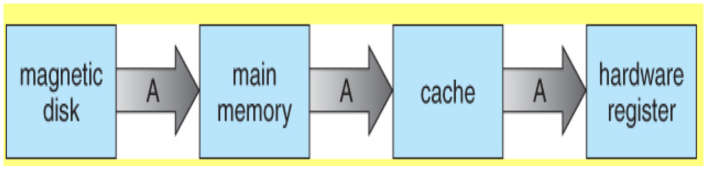


Figure 6: Migration of A from disk to register

- → In a **uniprocess system**, this arrangement poses no difficulties, since an access to integer A will always be to the copy at the highest level of the hierarchy.
- → However, in a **multitasking environment**, where the CPU is switched back and forth among various processes, extreme care must be taken to ensure that, if several processes wish to access A, then each of these processes will obtain the most recently updated value of A.
- → In a **multiprocessor environment** where, in addition to maintaining internal registers, each of the CPUs also contains a local cache. In such an environment, a copy of A may exist simultaneously in several caches. Since the various CPUs can all execute concurrently, we must make sure that an update to the value of A in one cache is immediately reflected in all other caches where A resides. This situation is called **cache coherency,** and it is usually a hardware problem (handled below the operating-system level).
- → In a **distributed environment**, several copies (or replicas) of the same file can be kept on different computers that are distributed in space. Since the various replicas may be accessed and updated concurrently, some distributed systems ensure that, when a replica is updated in one place, all other replicas are brought up to date as soon as possible.

4) I/O Systems

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 several components:

- 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.

Protection and Security

- *If a computer system has multiple users and allows the concurrent execution of multiple processes, then access to data must be regulated*. 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** is any mechanism for controlling the access of processes or users to the resources defined by a computer system.* A protection-oriented system provides a means to distinguish between authorized and unauthorized usage. A system can have adequate protection but still be prone to failure and allow inappropriate access. Consider a user whose authentication information is stolen. Her data could be copied or deleted, even though file and memory protection are working. It is the job of **security** to defend a system from external and internal attacks.
    - Most operating systems maintain a list of user names and associated **user identifiers (user IDs).** When a user logs in to the system, the authentication stage determines the appropriate user ID for the user. That user ID is associated with all of the user's processes and threads. When an ID needs to be user readable, it is translated back to the user name via the user name list.
    - In some circumstances, we wish to distinguish among sets of users rather than individual users. For example, the owner of a file on a UNIX system may be allowed to issue all operations on that file, whereas a selected set of users may only be allowed to read the file. To accomplish this, we need to define a group name and the set of users belonging to that group. Group functionality can be implemented as a system-wide list of group names and **group identifiers.**

Distributed Systems

- A distributed system is a *collection of physically separate, possibly heterogeneous computer systems that are networked to provide the users with access to the various resources* that the system maintains. **Access to a shared resource increases computation speed, functionality, data availability, and reliability**.
- A **network,** in the simplest terms, is a communication path between two or more systems. Distributed systems *depend on networking for their functionality*. Networks vary by the protocols used, the distances between nodes, and the transport media

- Networks are characterized based on the distances between their nodes. A **local-area network (LAN)** connects computers within a room, a floor, or a building. A **wide-area network (WAN)** usually links buildings, cities, or countries.
- *A distributed operating system provides a less autonomous environment: The different operating systems communicate closely enough to provide the illusion that only a single operating system controls the network.*

Special-Purpose Systems

There are different classes of computer systems whose functions are more limited and whose objective is to deal with limited computation domains.

1) Real-Time Embedded Systems

- Embedded computers are found everywhere, from car engines and manufacturing robots to VCRs and microwave ovens. They *tend to have very specific tasks.*
- The *systems they run on are usually primitive, and so the operating systems provide limited features.* Usually, they have *little or no user interface*, preferring to spend their time *monitoring and managing hardware* devices, such as automobile engines and robotic arms.
- Embedded systems almost always run **real-time operating systems.** *A real-time system is used when rigid time requirements have been placed on the operation of a processor or the flow of data;* thus, it is often used as a control device in a dedicated application.
- A real-time system has well-defined, fixed time constraints. Processing must be done within the defined constraints, or the system will fail. A real-time system functions correctly only if it returns the correct result within its time constraints.

2) Multimedia Systems

- Most operating systems are designed to handle conventional data such as text files, programs, word-processing documents, and spreadsheets.
- However, a recent trend in technology is the incorporation of **multimedia data.** These data differ from conventional data in that multimedia data—such as frames of video—*must be delivered (streamed) according to certain time restrictions* (for example, 30 frames per second).
- Multimedia describes a wide range of applications that are in popular use today. These include audio files such as MP3 DVD movies, video conferencing, and short video clips of movie previews or news stories downloaded over the Internet.

3) Handheld Systems

- **Handheld systems** include personal digital assistants (PDAs), such as Palm and Pocket-PCs, and cellular telephones, many of which *use special-purpose embedded operating systems*.
- Developers of handheld systems and applications face many **challenges**, most of which are due to the *limited size* of such devices. Because of their size, most handheld devices have a *small amount of memory, slow processors*, and *small display screens*.
- The amount of physical memory in a handheld depends upon the device, but typically is is somewhere between 512 KB and 128 MB. (Contrast this with a typical PC or workstation, which may have several gigabytes 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 when the memory is not being used

- A second issue of concern to developers of handheld devices is the **speed of the processor** used in the devices. Processors for most handheld devices 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, which would take up more space* and would have to be replaced (or recharged) more frequently. Most handheld devices use smaller, slower processors that consume less power. Therefore, *the operating system and applications must be designed not to drain the processor*.

- A lack of physical space limits input methods to small keyboards, handwriting recognition, or small screen-based keyboards. The small display screens limit output options. *Familiar tasks, such as reading e-mail and browsing web pages, must be condensed into smaller displays*. 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.

Computing Environments

1) Traditional Computing

- Just a few years ago, the computing environment consisted of PCs connected to a network, with servers providing file and print services. *Remote access was awkward, and portability was achieved by use of laptop computers*.
- The current trend is toward providing more ways to access these computing environments. Web technologies are stretching the boundaries of traditional computing. Companies establish **portals,** which provide web accessibility to their internal servers. **Network computers** are essentially terminals that understand web-based computing.

2 Client-Server Computing

- As PCs have become faster, more powerful, and cheaper, designers have shifted away from centralized system architecture. As a result, many of todays systems act as **server systems to** satisfy requests generated by **client systems.** This form of specialized distributed system, called **client-server** system, has the general structure depicted in Figure.
- Server systems can be broadly categorized as compute servers and file servers:
- The **compute-server system** provides an interface to which a client can send a request to perform an action (for example, read data); in response, the server executes the action and sends back results to the client

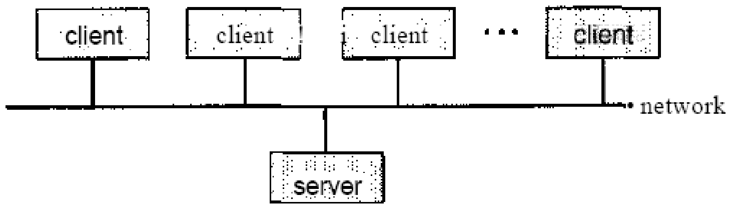


Figure 1.11 General structure of a client-server system.

- The **file-server system** provides a file-system interface where clients can create, update, read, and delete files.

3) Peer-to-Peer Computing

- In this model, *clients and servers are not distinguished from one another*; instead, all nodes within the system are considered peers, and each may act as either a client or a server, depending on whether it is requesting or providing a service. In a peer-to-peer system, services can be provided by several nodes distributed throughout the network.
- To participate in a peer-to-peer system, *a node must first join the network of peers*. Once a node has joined the network, *it can begin providing services to—and requesting services from*—other nodes in the network. Determining what services are available is accomplished in one of two general ways:
- When a node joins a network, it registers its service with a *centralized lookup service* on the network.
- *Two options are available for any node desiring a specific service*
  - It can contact the centralized lookup service to determine which node provides the service. The remainder of the communication takes place between the client and the service provider.
  - A peer can broadcast a request for the service to all other nodes in the network. The node (or nodes) providing that service responds to the peer making the request.

4) Web-Based Computing

- Web computing has increased the emphasis on networking. Devices that were not previously networked now include wired or wireless access. Devices that were networked now have faster network connectivity, provided by either improved networking technology, optimized network implementation code, or both.
- The implementation of web-based computing has given rise to new categories of devices, such as **load balancers,** which distribute network connections among a pool of similar servers.

Ch 2: System Structures

OPERATING SYSTEM SERVICES:

OS provides an environment for execution of programs. It provides certain services to programs and to the users of those programs. OS services are provided for the convenience of the programmer, to make the programming task easier.

One set of OS services provides functions that are helpful to the user –

1. **User interface:** User interfaces are the means by which a user can communicate with the operating system. Interfaces are of three types-
   - → *Command Line Interface*: uses text commands and a method for entering them
   - → *Batch interface*: commands and directives to control those commands are entered into files and those files are executed.
   - → *Graphical user interface*: A window system with a pointing device to direct I/O, choose from menus and make selections and a keyboard to enter text.
2. **Program execution:** The system must *be able to load a program* into memory and *run* that program. The program must be able to *end its execution either normally or abnormally* (indicating error).
3. **I/O operations:** A running program may require I/O which may involve a file or an I/O device. For efficiency and protection, *users cannot control I/O devices directly*. Therefore, *the operating system must provide a means to do I/O*.

4. **File system manipulation:** 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*. Many operating systems provide a variety of file systems, sometimes to allow personal choice, and sometimes to provide specific features or performance characteristics.

5. **Communications:** One process might need 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*.

6. **Error detection:** Errors may occur in
   - → CPU & memory-hardware (ex: power failure)
   - → I/O devices (ex: lack of paper in the printer) and
   - → user program (ex: arithmetic overflow)

   For each type of error, OS *should take appropriate action* to ensure correct & consistent computing.

Another set of OS functions exist **for ensuring efficient operation** of the system. They are-

1. **Resource allocation:** When there are multiple users or multiple jobs running at the same time, resources must be *allocated to each of them*. Different types of resources such as CPU cycles, main memory and file storage are managed by the operating system.

2. **Accounting:** We want to keep track of
   - i. which users use how many resources and
   - ii. what kinds of resources.

   b. This record keeping may be used for
   - i. accounting (so that users can be billed) or
   - ii. gathering usage-statistics.

3. **Protection and security:** Controlling the use of information stored in a multiuser or networked computer system. Protection involves ensuring that all access to system resources is controlled. Security starts with requiring each user to authenticate himself or herself to the system by means of password and to gain access to system resources.

USER OPERATING SYSTEM INTERFACE

As mentioned earlier, the OS provides 2 types of service.

- i. Command Line Interface or Command Interpreter.
- ii. Graphical User Interface.

Command Interpreter

→ Provides a command line interface. User has to type in appropriate commands to get the job done. Its function is to get and execute the next user-specified

→ Two general ways to implement:

a. **Command interpreter itself contains code** to execute command (ie the logic of the command is a part of the command interpreter). As the number of commands increase the size of the command interpreter also increases.

b. **Commands are implemented through system programs.** This is used by UNIX. In this case, the command interpreter does not understand the command, it merely uses the command to identify a file to be loaded into memory and executed (ie the logic for the commands are implemented as files).

For example, the UNIX command to delete file 'rm file.txt' would search for a file called rm, load the file into memory, and execute it with the parameter file.txt. The function associated with the rm command would be defined completely by the code in the file rm. In this way new commands can be added to the system easily by creating new files with the proper names. The command interpreter program, which can be small, does not have to be changed for new commands to be added.

Graphical User Interfaces

In this rather than entering commands directly via a command-line interface, *users employ a mouse-based window-and-menu system*. The user moves the mouse to position its pointer on images or icons on the screen (the desktop) that represent programs, files, directories, and system functions. Depending on the mouse pointer's location, clocking a button on the mouse can invoke a program, select a file or directory or pull down a menu that contains commands.

SYSTEM CALLS

User processes cannot perform privileged operations themselves. They must request OS to do so on their behalf by issuing system calls

System calls is the programming interface to the services provided by the OS.

- These calls are generally *available as routines written in C, and C++* and sometimes using *assembly language instructions*.

An example to illustrate how system calls are used: Writing a simple program to read data from one file and copy them to another file- [refer text book]

➢ The disadvantages of a system call:
  - They are **operating system dependent**. *To get similar service on different OS different system calls must be used.*
    - Example: To create a process in windows the system call is NTCreateProcess(), whereas to create a process in UNIX the system call is fork().
  - The *application* or the program which consists of system calls *will not be portable*.

Solution: Application Programming Interface [API]

- API *specifies set of functions* that are available to an application programmer to access the services of OS. Behind the scenes the functions that make up an API *typically invokes the actual system call on behalf of the programmer*.
- The runtime support system of most programming language provides a system call interface that serves as a link to system calls made available by the OS (as shown in the figure).
- The system call interface intercepts function calls in the API and invokes the necessary system call within the OS.

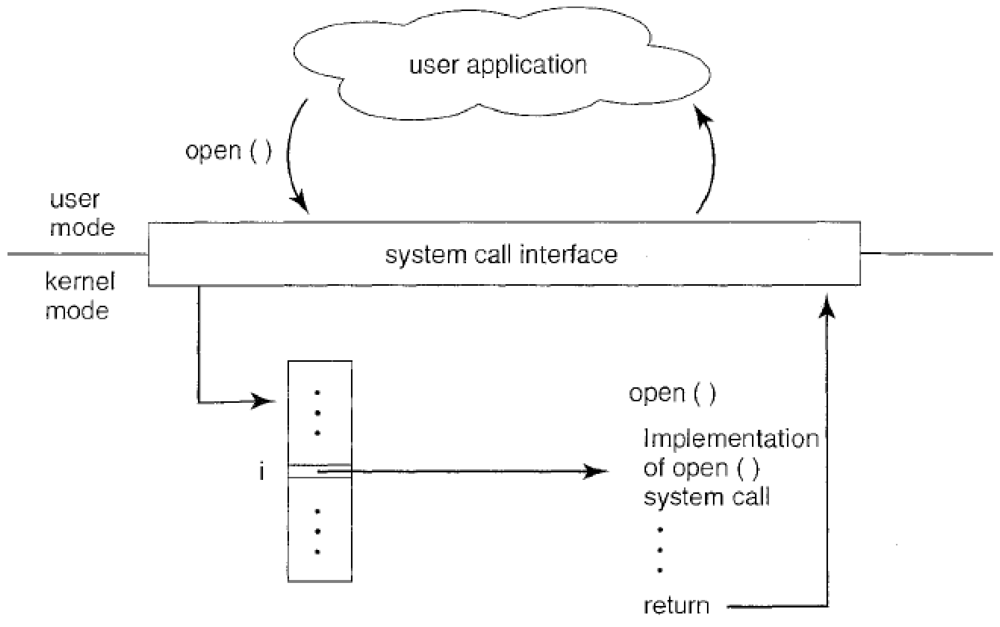


Figure 2.6 The handling of a user application invoking the open() system call.

- Each **system call** will be *associated with a number* and the **system call interface** *maintains a table indexed according* to these numbers.
- The system call interface then *invokes the intended system call* in the OS kernel and returns the status of the system call and any return values.

Benefits of programming with API over system calls:

- ➢ **Program portability** – An application programmer designing a program using an API can *expect program to compile and run on any system that supports the same API.*
- ➢ **Actual system calls** can be more *detailed and difficult* to work with than the API available to an application programmer.

To use system calls, we need to know the *identity* (name) of the system call and the *essential parameters* required to invoke the system call. Three general methods are used to **pass parameters to system calls:**

1. Registers
2. Store parameters in blocks or tables in memory and the *address of the block* is passed as a parameter in a register
3. Stack **-** Program can push parameters onto a stack and OS can pop the values from it.

Some OS prefer method 2 & 3 because *they do not limit the length or number of parameters being passed*

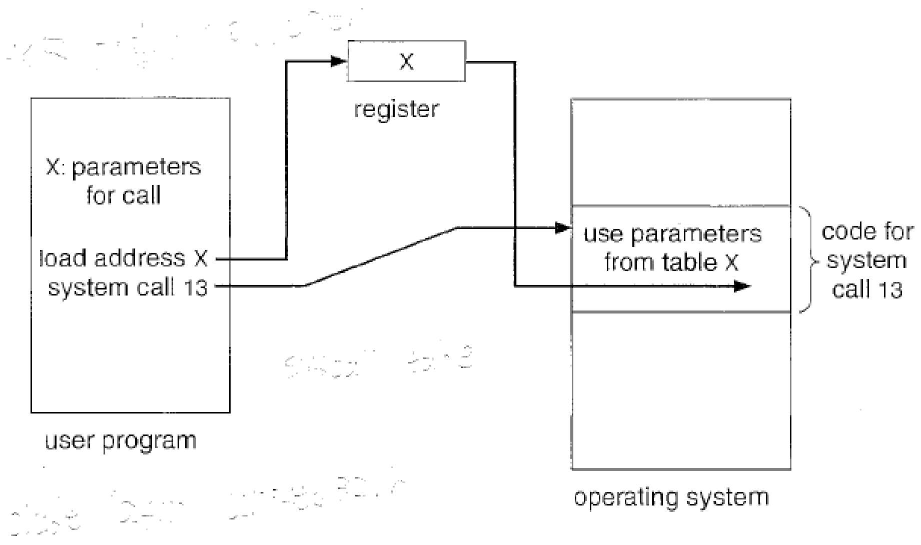


Figure 2.7 Passing of parameters as a table.

Types of system calls

The system calls can be categorized into 6 major categories.

1. Process Control
2. File Manipulation
3. Device Manipulation
4. Information Maintenance
5. Communications
6. Protection

1. Process control

a. **end, abort -** A running program needs to be able to halt its execution either normally (end) or abnormally(abort).
b. **load, execute -** A process executing a job my want to load and execute another program.
c. **create process, terminate process**
d. **get process attributes, set process attributes -** We should be able to create a new process, set its properties (attributes) such as process priority, maximum execution time, etc
e. **wait for time ,wait event, signal event -** Having created new jobs we may want to wait for them to complete execution, or wait for some time to pass, or may want to wait for the completion of an event. The jobs should be able to signal that an event has occurred
f. **allocate and free memory -** Quite often, two or more processes may share data. To ensure the integrity of data being shared, OS often allows system calls allowing a process to lock shared data and release when data is no longer used by a process.

2) File Management

I. **create file, delete file**
II. **open, close**
III. **read, write, reposition**
IV. **get file attributes, set file attributes**

<u>Working procedure:</u>

1. We need to create and delete files.
2. Once the file is **created**,
   → we need to **open** it and to use it.
   → we may also **read** or **write**.
3. Finally, we need to **close** the file.
   • We need to be able to
      → determine the values of file-attributes and
      → reset the file-attributes if necessary.
   • **File attributes** include file name, file type, protection codes and accounting information.

3) Device management

A process may need several resources to execute. If resources are available they will be granted, else the process has to wait until sufficient resources are available. The various resources controlled by OS are referred to as devices.

I. **request device, release device -** The process that needs the device must first **request** for the resource. After the process has finished using the device, it has to release the device

II. **read, write, reposition -** Once the device is aloted to a process it can read, write, and reposition the read and write pointers on the device

III. **get device attributes, set device attributes**

IV. **logically attach or detach devices -** attach/detach the device to the file system

4) Information maintenance

Some system calls exist *<u>purely to transfer information between user programs and OS</u>* (Ex: number of current users, the version number of the operating system, the amount of free memory or disk space, and so on).

I. get time or date, set time or date

II. get system data, set system data

III. get process, file or device attributes

IV. set process, file or device attributes

OS keeps track of all its processes and provides system calls to query this info(Ex: time profile of program to tell the time it took for execution of a particular instruction).

5) Communications

The system calls supported for interprocess communication includes:

I. create, delete communication connection

II. send, receive messages

III. transfer status information

IV. attach or detach remote devices

There are 2 models for interprocess communication

- **Message Passing model -** Uses a common mailbox to pass messages between the processes
- **Shared Memory model -** Here 2 proceses exchange data by reading and writing to the same shared data

In message passing model

- Before communication takes place, connection must be **opened.** The name (identifier) of the other communication party must be known which are passed to **open** connection and **close** connection system calls
- Processes can give permission for communication with **accept connection** system call. They execute a **waitForConnection** system call and are invoked when a connection is established.
- Sender and reciever exchange messages using **read message** and **write message** system call.
- **Close connection** system call terminates the communication

In shared memory model

- Process use **shared memory create** to create and gain access to regions of memory owned by other processes.
- Normally OS prevents a process from accessing another process memory. This model require process to agree to remove the restriction.

6) Protection

Protection provides a mechanism for *controlling access to the resources* provided by a computer system. System calls providing protection include *set permission* and *get permission*, which manipulate the permission settings of resources such as files and disks. The *allow user and deny user* system calls specify whether particular users can or cannot be allowed access to certain resources.

SYSTEM PROGRAMS

- System programs are also refered to as '*System Utilities*'.
- They are bundles of useful system calls.
- System programs provide a convenient environment for program development and execution.

Categories of System Programs:

- **File management**. These programs manipulate files i.e. *create, delete, copy*, and *rename* 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*. Others are more complex, providing detailed *performance, logging*, and *debugging* information.
- **File modification**. *text editors* to create and modify the content of files, special commands to *search contents* of files or *perform transformations* of the text.
- **Programming-language support**. *Compilers, assemblers, debuggers* and *interpreters* for common programming languages (such as C, C++, Java, Visual Basic, and PERL)

- **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 as well.
- **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.

OPERATING SYSTEM DESIGN AND IMPLEMENTATION

Design Goals

- The first problem in designing a system is to
  - → **define goals** - different users have different expectations from the Operating system, hence it becomes difficult to generalize a design goal for all users of OS

  - The design of the system will be affected by
    - choice of hardware and type of system (batch or time shared, single user or multiuser)
  - Two basic groups of requirements:User goals and 2. System goals
    - → **User Goals**

      The system should be
      - convenient to use
      - easy to learn and to use
      - reliable, safe, and fast.
    - → **System Goals**

      The system should be
      - easy to design
      - implement, and maintain
      - flexible, reliable, error free, and efficient.

→ Mechanisms & Policies

- → Mechanisms determine how to do something.
- → Policies determine what will be done.
- → Separating policy and mechanism is important for flexibility.
- → Policies change over time; mechanisms should be general.

→ Implementation

- Once the design goals are finalized and mechanisms and policies are created, OS can be developed using some high level language such as C/C++

OPERATING SYSTEM STRUCTURES

The OS, *being large and complex software*, *need to be designed carefully* if it has to function correctly and be modified easily. Over the years several types structures have been used for designing operating systems. Some of the popular structures are:

- Simple structure
- Layered approach
- Microkernel
- Modules.

Simple Structure

Earlier operating systems such as MS-DOS and traditional UNIX did not have well-defined structures. These operating systems started as small, simple, and limited systems and then grew beyond their original scope.

MS-DOS was initially designed to provide the most functionality in the least space, hence it was not divided into modules carefully.

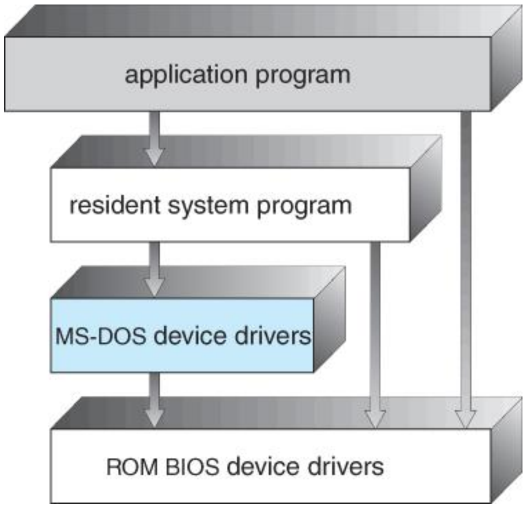


fig: MS-DOS structure

The MS-DOS structure *does not break the system into subsystems*, and has **no** distinction between **user and kernel modes** (because of hardware limitations), allowing all programs direct access to the underlying hardware therefore was vulnerable and could be attacked easily by malicious programs.

The original UNIX OS used a simple layered approach, but **almost all the OS was in one big layer**. It consists of two separable parts: kernel and the system programs. All the important functions - file management, I/O access, memory management, paging, scheduling, etc were provided by the kernel through system calls (therefore referred to as "**Monolithic kernel**"). A monolithic kernel becomes difficult to implement and maintain.

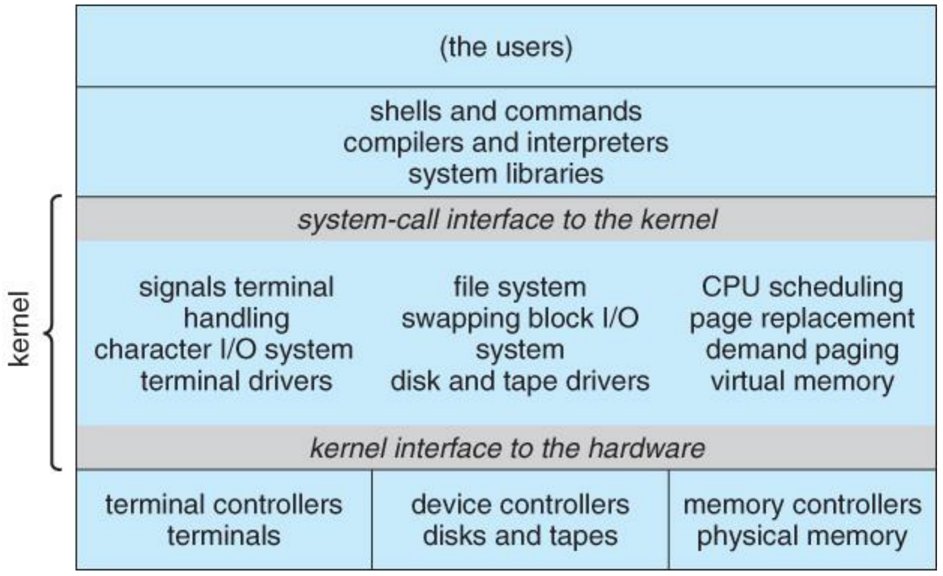
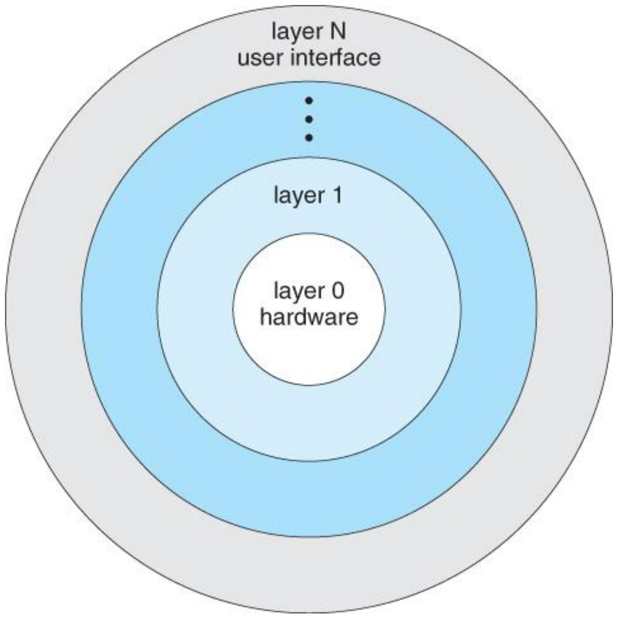


Fig: traditional UNIX system structure

Layered approach

This approach *breaks up the operating system into different layers*. Each layer rests on the layer below it, and *relies solely on the services provided by the next lower layer*.

The bottom layer (layer 0) is the hardware; the highest (layer N) is the user interface. This layering structure is depicted in Figure below.



Advantages:

- **Simplifies Debugging and system verification**
    - This approach *allows each layer to be developed and debugged independently*, with the assumption that all lower layers have already been debugged and are trusted to deliver proper services.
    - The first layer can be debugged without any concern for the rest of the system, because, it uses only the basic hardware (which is assumed correct). Once the first layer is debugged, it can be used to verify the proper functioning of the next layer. 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

- **Simplifies design and implementation of the system**
    - Each layer is implemented with only those operations provided by lowerlevel layers. A layer does not need to know how these operations are implemented; it needs to know only what these operations do. Hence, each layer hides the existence of certain data structures, operations, and hardware from higher-level layers.

Challenges:

- The problem is ***deciding the order*** in which to place the layers, as no layer can call upon the services of any higher layer, and so many chicken-and-egg situations may arise.

- Layered approaches can also be ***less efficient***, as a request for service from a higher layer has to filter through all lower layers before it reaches the HW, possibly with significant processing at each step.

Microkernels

- The basic idea behind micro kernels is to *remove all non-essential services* from the kernel, and implement them as system and user level programs, thereby *making the kernel as small and efficient* as possible.
- Most microkernels provide minimal ***process and memory management***, and a ***communications*** facility. Communication between components of the OS is provided by message passing.
- Examples of OS that used Microkernel structure - Tru64 UNIX, QNX, Windows NT.

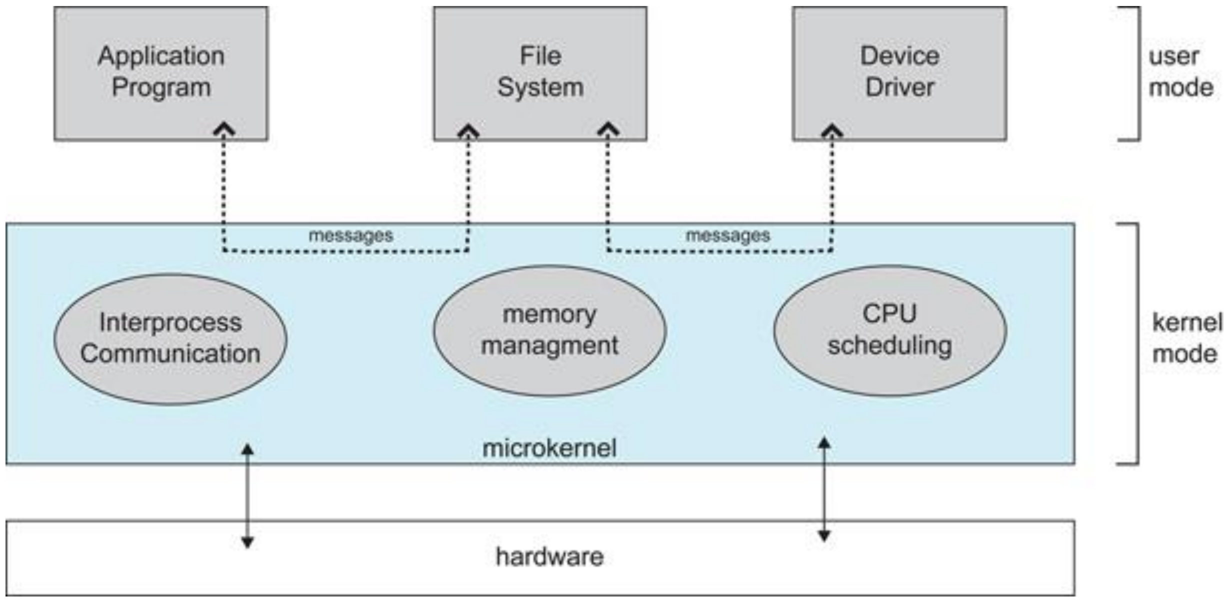


Fig: Architecture of a typical microkernel

Benefits

Reduced Kernel helps in

- ➔ **Ease of extending the OS -** All new services are added to user space and consequently do not require modification of the kernel.
- ➔ **Fewer Modifications to OS -** Since kernel is smaller, any changes to kernel tend to be fewer
- ➔ **Enhanced Protection -** The number of components which have direct access to hardware is few.
- ➔ **Increased Reliability -** since most services are running as user—rather than kernel—processes, If a service fails, the rest of the operating system remains untouched.
- ➔ **Increased portability of OS -** Reduced size of the kernel ⇒ less code to be modified for porting the OS.

Disadvantage

- ➔ **Reduced performance** due to increased system overhead from message passing (need to switch from user mode to kernel mode very often).

Modules

Modern OS development is object-oriented, with a relatively small core kernel and a set of ***modules*** which can be linked in dynamically during boot time or runtime.

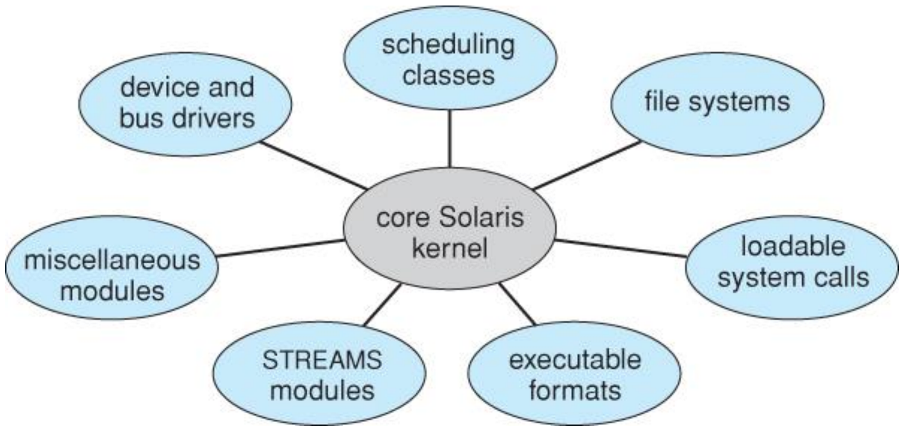


Fig: Solaris Loadable Modules

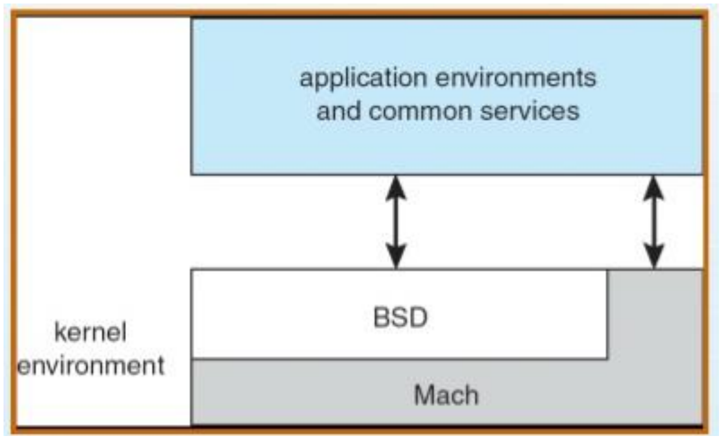
This approach is more like the microkernel approach in that the primary module has only core functions and knowledge of how to load and communicate with other modules; but it is more efficient, because modules can interact with each other directly and do not need to invoke message passing in order to communicate.

Advantages:

- easy to extend the system by adding new modules or new hardware.

Hybrid Structure

The Apple Macintosh Mac OS X operating system uses a **hybrid structure**. Mac OS X structures the operating system using a layered technique where one layer consists of the Mach microkernel (as shown below).



The top layers include application environments and a set of services providing a graphical interface to applications. Below these layers is the kernel environment, which consists primarily of the Mach microkernel and the BSD kernel. Mach provides memory management; support for remote procedure calls (RPCs) and inter-process communication (IPC) facilities, including message passing; and thread scheduling.

The BSD component provides a BSD command line interface, support for networking and file systems, and an implementation of POSIX APIs, including Pthreads. The kernel environment also provides an I/O kit for development of device drivers and dynamically loadable modules (kernel extensions).

VIRTUAL MACHINES

The fundamental idea behind a virtual machine is to *abstract the hardware of a single computer* (the CPU, memory, disk drives, network interface cards, and so forth*) into several different execution environments*, thereby creating the illusion that each separate execution environment is running its own private computer.

An OS creates the illusion that a process has

→ own processor &

→ own (virtual) memory.

• The virtual-machine provides

→ an interface that is identical to the underlying hardware

→ a (virtual) copy of the underlying computer to each process.

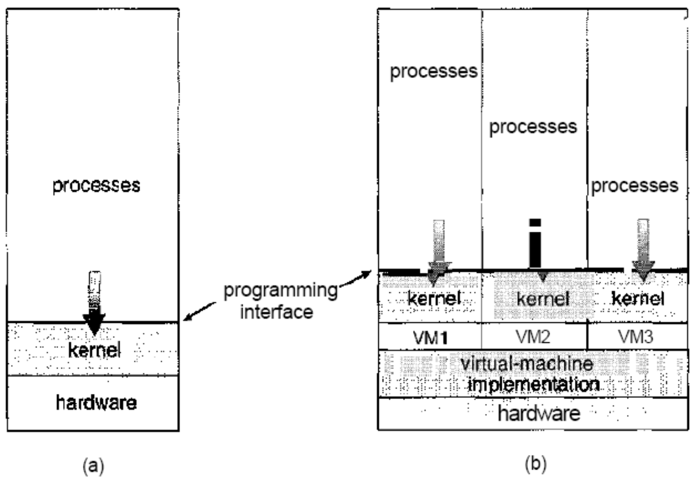


Figure 2.15 System models. (a) Nonvirtual machine. (b) Virtual machine.

Implementation

The difficulty in implementing virtual machine lies in providing the exact duplicate of the underlying machine.

The underlying machine has two modes. User mode and kernel mode. The virtual machine software can run in kernel mode since it is the OS. The virtual machine itself execute only in user mode. As the physical machine virtual machine also has 2 modes- virtual user mode and virtual kernel mode.

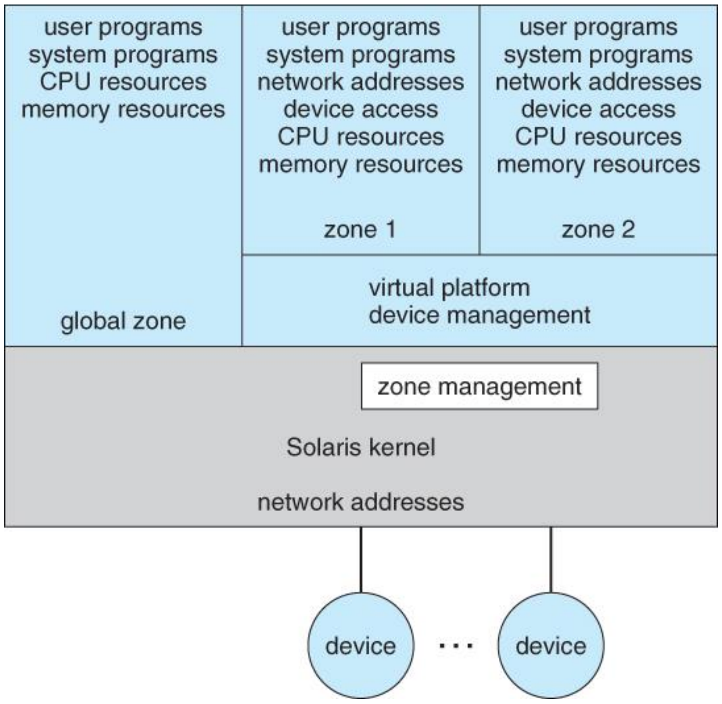
Those actions that cause a transfer from user mode to kernel mode on a real machine must also cause a transfer from virtual user mode to virtual kernel mode on a virtual machine.

Benefits

- **Protection of resources**: Any virus infecting one guest OS will not be able to harm the host system or any other guest OSes running on the system
- **Simplifies OS research and Development,** that otherwise requires careful efforts or else the entire system might break down
- Since multiple OSes can run on the same machine, **rapid porting and testing** of programs in various environments can be done.
- Virtualization supports **system consolidation**. Many lightly used systems can be combined to create one or more heavily used system. Such conversions result in resource optimization.

Para-Virtualization

In para-virtualization the guest OS is fully aware that it is a guest and that it does not have direct control over the hardware. Accordingly the guest OSes drivers are modified to run on the para-virtualized hardware.



[Refer text for more details]

Examples

1) VMware

VMware is a popular commercial application that abstracts Intel 80X86 hardware into isolated virtual machines. VMware runs as an application on a host operating system such as Windows or Linux and allows this host system to concurrently run several different guest operating systems as independent virtual machines.

Consider the following scenario: A developer has designed an application and would like to test it on Linux, FreeBSD, Windows NT, and Windows XP. Such testing could be accomplished concurrently on the same physical computer using VMware. In this case, the programmer could test the application on a host operating system and on three guest operating systems with each system running as a separate virtual machine.

The architecture of such a system is shown below.

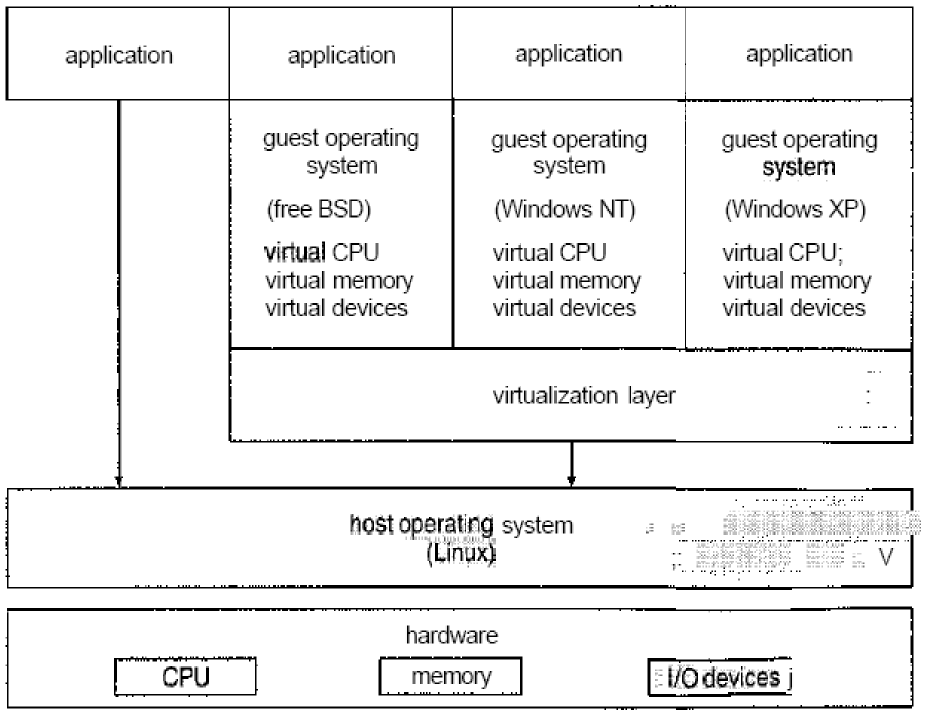


Figure 2.16 VMware architecture.

In this scenario, Linux is running as the host operating system; FreeBSD, Windows NT, and Windows XP are running as guest operating systems. The virtualization layer is the heart of VMware, as it abstracts the physical hardware into isolated virtual machines running as guest operating systems. Each virtual machine has its own virtual CPU, memory, disk drives, network interfaces, and so forth.

2) The Java Virtual Machine

Java is a popular object-oriented programming language introduced by Sun Microsystems. In addition to a language specification and a large API library, Java also provides a specification for a Java virtual machine—or JVM.

A Java program consists of one or more classes. For each Java class, the compiler produces an architecture-neutral bytecode output (.class) file that will run on any implementation of the JVM.

The JVM consists of a class loader and a Java interpreter that executes the architecture-neutral bytecodes, as shown in Figure. The class loader loads the compiled . class files from both the Java program and the Java API for execution by the Java interpreter. After a class is loaded, the verifier checks that the . class file is valid Java bytecode and does not overflow or underflow the stack. It also ensures that the bytecode does not perform pointer arithmetic, which could provide illegal memory access. If the class passes verification, it is run by the Java interpreter. The JVM automatically manages memory by performing garbage collection.

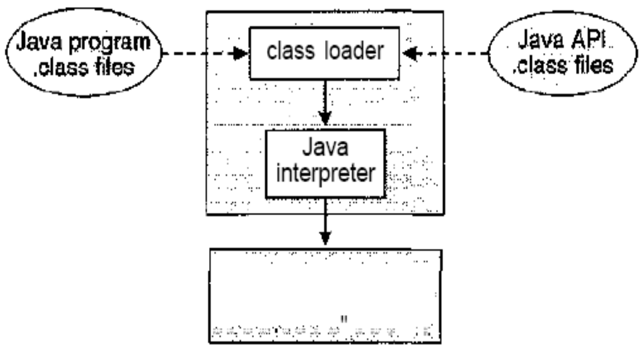


Figure 2.17 The Java virtual machine.

Operating System Generation

→ OS is designed to run on any of a class of machines.

o However, the system must be configured for each specific computer site

- **SYSGEN** is used for *configuring a system for each specific computer site*
- **SYSGEN** program must determine:
  - What CPU will be used?
  - How will boot disk be formatted?
  - How much memory is available?
  - What devices are available?
  - What OS options are desired?

→ A system-administrator can use the above information to modify a copy of the source code of the OS

System Boot

→ Booting means starting a computer by loading the kernel.

→ Bootstrap program is a code stored in ROM.

→ The bootstrap program

- o locates the kernel
- o loads the kernel into main memory and
- o starts execution of kernel.
  - ▪ OS must be made available to hardware so hardware can start it.

PROCESS MANAGEMENT

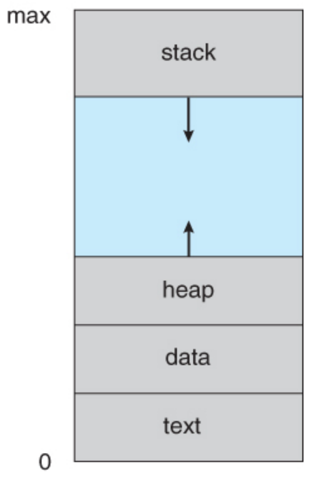
Process Concept

A process is an ***instance of a computer program under execution***. When we ask the computer to execute a program, the code for the program is loaded from the disk into memory and executed as a process. Processes are sometimes referred to as *jobs* or *tasks*.

1. **What is a process? Draw and explain the state diagram of a process. Give a note on Context switch.** [8M]
2. What do you understand by PCB? Where is PCB used? Explain its contents [8M]

The Process

When a program is loaded into the memory and it becomes a process, it can be divided into four sections ➔ stack, heap, text and data. The following image shows a simplified layout of a process inside main memory



The memory containing the process is divided into the following segments:

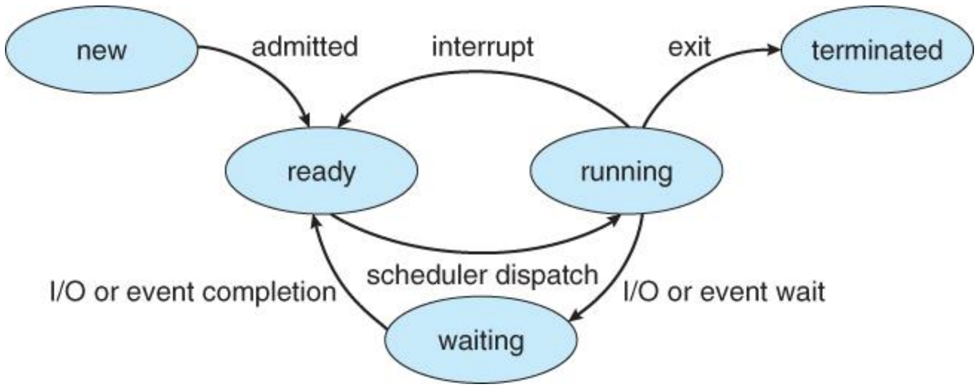
- **Text** - Program code and other read only data are placed into the text segment
- **Data** - Global variables used in the program, that can be read and written to, are stored in the data segment.
- **Stack** - Temporary data such as automatic variables, local variables, function parameters that are allocated at compile time are placed on the stack segment
- **Heap** - Data structures explicitly allocated at runtime are placed on the heap. As memory is used by a process, the stack and the heap grow toward each other.

Process State

During the lifetime of a process, the process moves between several states. Each process may be in one of the following states

- **New** - the initial state when a process is first started/created.
- **Ready** - When process is ready to execute with all the necessary resources at hand, but is waiting to be assigned to the CPU
- **Running** - When the process is currently being executed by a processor
- **Waiting** - The process is waiting for a specific event to occur (ex: keyboard input, disk access request, inter-process messages, a timer to go off, or a child process to finish) before it can proceed
- **Terminated** - Once the process finishes its execution, or it is terminated by the operating system, it is moved to the terminated state where it waits to be removed from main memory.

The figure below shows the state transition diagram for the processes in the system.



The lines connecting the states represent possible transitions from one state to another. At any instant, a process will exist in one of these three states.

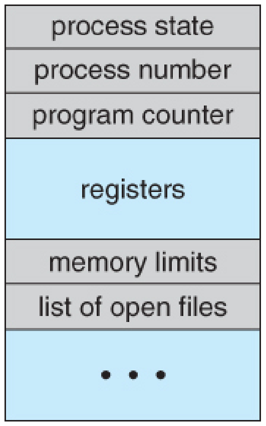
Note that certain rules apply here. Processes entering the system must initially go into the ready state. A process can only enter the running state from the ready state. A process can normally only leave the system from the running state, although a process in the ready or blocked state may be aborted by the system (in the event of an error, for example), or by the user.

Process Control Block

A Process Control Block is a data structure maintained by the Operating System for every process. In order to track processes correctly and allow multiple processes to share the same system, the OS maintains some information associated with each process. Each process is represented by the OS by a Process Control Block (PCB) or task control block.

A PCB contains many pieces of information associated with a process including:

- **Process State** - new, running, ready, waiting or terminated
- **Program counter** - Contains the address of the next instruction to be executed.
- **CPU registers** - The PCB also stores the contents of various processor registers (accumulator, index register, stack pointers, general purpose registers, etc), which are saved when a process leaves the running state and which are restored to the processor when the process returns to the running state.
- **CPU scheduling information** - process priority, pointer to scheduling queues, etc
- **Memory management** - information about the logical address space of the process in memory (value in base & limit registers and page tables & segment tables)
- **Accounting Information -** OS needs to keep scheduling info and statistics about the process such as process ids, process owner, statistics about process execution - amount of user and kernel CPU time used, time limits, etc.
- **I/O status information** - list of I/O devices allocated to the process, list of open files, network connections used by the process, etc.



The PCB serves as a repository of all necessary information about a process.

Threads

On a modern system, processes consist of one or more threads of execution i.e. it can execute one instruction at a time or can execute several instructions at the same time, thereby allowing execution of more than one task at a time. Ex: type and spell check in word processors. On a system that supports threads, the PCB is extended to include information about each thread.

Process Scheduling

1. Explain the different types of schedulers
2. **Differentiate between Short term, long term and medium term scheduler [6M]**
3. List out the circumstances under which CPU scheduling takes place [5M]
4. **Write a note on Context switch [5M]**
5. Describe the actions an Operating system takes to context switch between the processes [4M]

The objective of multiprogramming is to have *some process running at all times*, to maximize CPU utilization. The objective of time sharing is to *switch the CPU among processes so frequently* that users can interact with each program while it is running.

The process scheduler must meet these objectives by implementing suitable policies for swapping processes in and out of the CPU. (Note that these objectives can be conflicting. In particular, every time the system steps in to swap processes it takes up time on the CPU to do so, which is thereby "lost" from doing any useful productive work.)

Scheduling Queues

The OS maintains all PCBs in Process Scheduling Queues. The OS *maintains a separate queue (usually implemented as a linked list) for each of the process states* and PCBs of all processes in the same execution state are placed in the same queue. When the *state of a process is changed*, *its PCB is unlinked* from its current queue and moved to its new state queue.

The Operating System maintains the following important process scheduling queues −

- **Job queue** − This queue keeps all the processes in the system.
- **Ready queue** − This queue keeps a set of all processes residing in main memory, ready and waiting to execute. A new process is always put in this queue.
- **Device queues** − The processes which are blocked due to unavailability of an I/O device constitute this queue.

The OS can use different policies to manage each queue (FIFO, Round Robin, Priority, etc.). The OS *scheduler determines how to move processes between the ready and run queues* (run queue can only have one entry per processor core on the system).

As processes enter the system, they are put into a **job queue**, which consists of all processes in the system. The processes that are residing in main memory and are ready and waiting to execute are kept on a list called the **ready queue**.

When a process is allocated the CPU, it executes for a while and eventually quits, is interrupted, or waits for the occurrence of a particular event (Ex: I/O access to a shared disk). Since there are many processes in the system, the disk may be busy with the I/0 request of some other process. The process therefore may have to wait for the disk. The list of processes waiting for a particular I/0 device is called a **device queue**. Each device has its own device queue.

A common representation of process scheduling is shown using a queuing diagram:

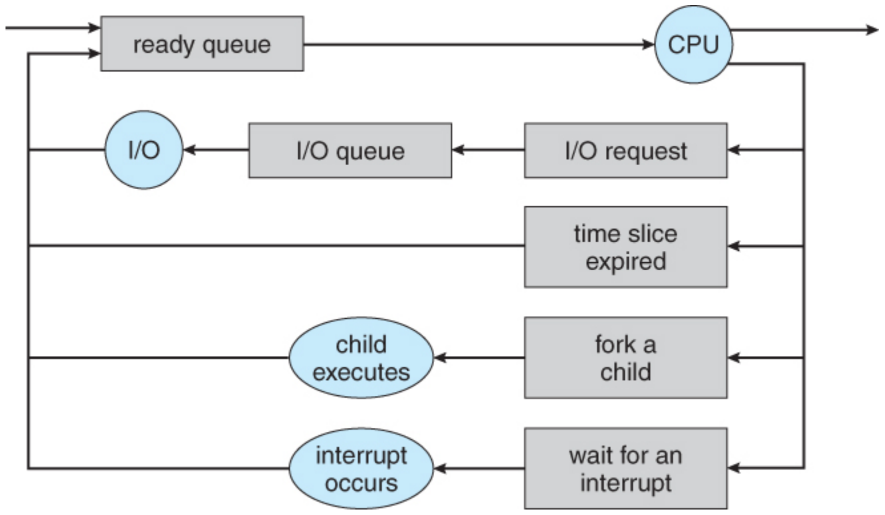


Fig: Queuing diagram representation of process scheduling

A new process is initially put in the **ready queue**. It waits there until it is selected for execution, or is dispatched. Once the process is allocated the CPU and is executing, one of several events could occur:

- The process could issue an I/0 request and then be placed in an I/0 queue.
- The process could create a new subprocess and wait for the sub-process's termination.
- The process could be removed forcibly from the CPU, as a result of an interrupt, and be put back in the ready queue.

Schedulers

A scheduler is an operating system program (module) that selects the next job to be admitted for execution. The main objective of scheduling is to increase CPU utilization and higher throughput.

There are 3 types of schedulers categorized by their *frequency of execution*:

1. Long Term Schedulers
2. Short Term Schedulers
3. Medium Term Schedulers

Long Term Scheduler

A **long-term scheduler** is typical of a batch system or a very heavily loaded system (where number of processes admitted to the system is more than the number of processes that can be executed immediately). These processes are kept in mass storage devices like disk for later processing. The long term scheduler selects processes from this pool and loads them into memory into the **ready queue**.

➔ The **long term scheduler** (also called **job scheduler**) selects processes from this pool and loads them into memory for execution.

➔ It controls the **degree of multiprogramming** (number of processes loaded into memory). The long term scheduler may need to be invoked only when a process leaves the system, thus giving it more time to carefully select a ***good mix*** of *CPU bound* (processes which are computation intensive) and *I/O bound* process (processes that spend more time doing I/O). [If all processes are I/O bound, the ready queue will almost always be empty. If all processes are CPU bound, waiting queue will always be empty, devices go unused and again the system will be unbalanced]

Short Term Scheduler

The **Short term Scheduler** (also called **CPU Scheduler** or **dispatcher**) selects from among the processes in memory which are ready to execute and assigns the CPU to one of them.

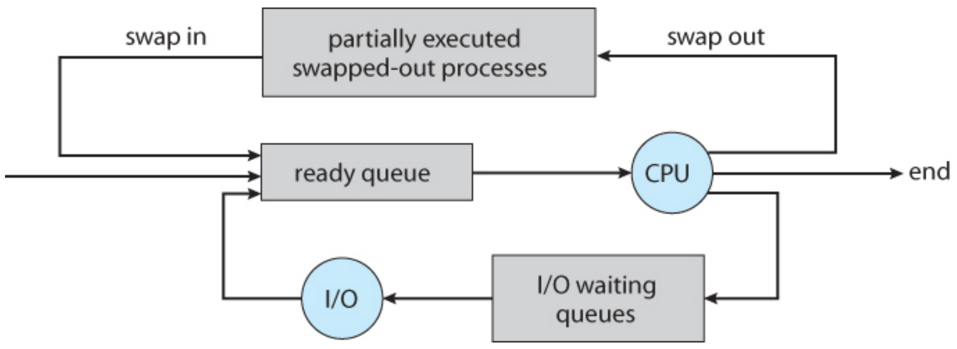
A short term scheduler is invoked whenever an event occurs that may lead to interruption of the currently running process

- A process under execution may have to halt execution because it requests an I/O operation, or because it times out, or because a hardware interrupt has occurred. The objectives of short-term scheduling is to ensure efficient utilization of the processor and to provide an acceptable response time to users.
- Short term scheduler **runs very frequently**, in the order of 100 milliseconds, and must very quickly swap one process out of the CPU and swap in another one.

Medium Term Scheduler

Some operating systems may include an additional intermediate level of scheduling - medium term scheduling. At times it may be necessary to swap out a processes which have not been active for a long time or has lower priority or takes up lot of memory space in order to free the memory for new processes or for one of previously suspended processes.

The mid term scheduler takes the responsibility of "***swapping***" out such processes when needed and also is responsible for swapping the process back into memory when sufficient memory is available or when the process is unblocked or the process is no longer waiting for some resource.



When a part of the memory becomes free as a result of termination of a certain process, the Medium term scheduler looks at the list of suspended/ ready processes and decides which one to swap into the memory for execution.

Context Switch

- Whenever CPU is interrupted it has to stop execution of the current task and move on to handle the interrupt. C**ontext switch** is the process of storing and restoring the state (PC, register contents) of a process or thread so that execution can be resumed from the same point at a later time
- Similarly switching the CPU to another process also requires performing a state save of the current process and state restore of another process. This is referred to as "Context Switch"

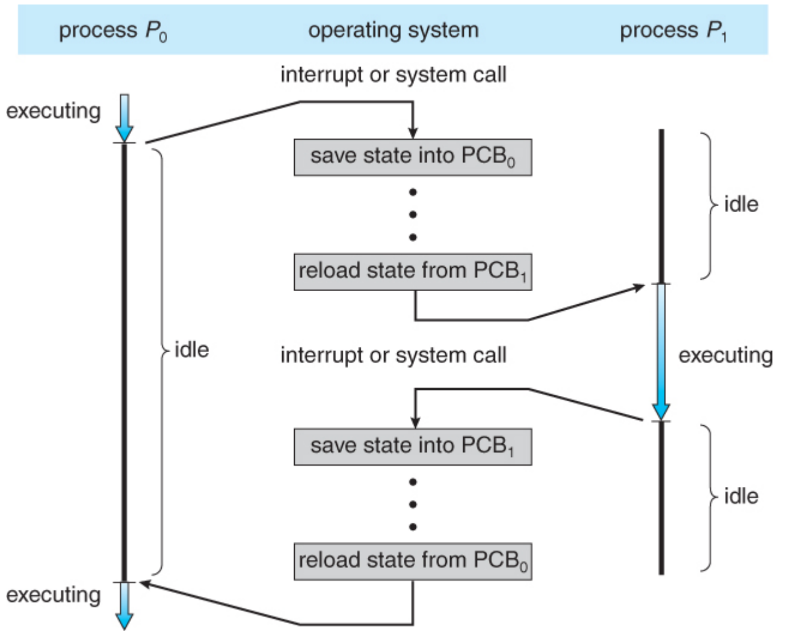


Figure: Diagram showing CPU switch from one process to another

- When a context switch occurs, the kernel *saves the context of the old process in its PCB and loads the saved context of the new process scheduled to run.*
- Context-switch time is ***pure overhead***, because the system does no useful work while switching. Some hardware has special provisions for speeding this up, such as a single machine instruction for saving or restoring all registers at once.

Operations on Process

1. With illustrations explain how process are created and executed in OS [4M]
2. Discuss the operations of process creation and process termination in UNIX [7M]
3. Write a C program forking a separate process [4M]
4. **Write a C program to create a child process and synchronize with the main program [6M]**
5. Describe the use of fork() and exec() system calls [4M]
6. Explain how process are created and terminated

The processes on most systems run concurrently and they may be created and deleted dynamically. Therefore systems must provide mechanisms for process creation and termination.

Process creation

Through appropriate system calls, such as fork or spawn, processes may create other processes. The process which creates another process, is termed the **parent** of the other process, while the created sub-process is termed its **child**. Each of these new process may in turn create other processes, forming a tree of processes.

Each process is given an integer identifier, termed as process identifier, or PID. The parent PID (PPID) is also stored for each process.

- In general a process will need certain resources to execute (CPU time, memory, files, I/O devices) to accomplish its task. When a process creates a subprocess, **the subprocess can be allotted the required resources in different ways:**
    - Parent and children share all resources
    - Children share subset of parent's resources
    - Parent and child share no resources [a separate copy of the resources is provided to child]
    - Restricting a child process to a subset of parent resources prevents any process from overloading the system by creating too many subprocesses.
    - Along with the necessary resources, the parent process may also pass any required initialization data (input) to the child process
- When a process creates a new process, **two possibilities exists in terms of execution**:
    - Parent and children execute concurrently
    - Parent waits until children terminate
- There are also **2 possibilities in terms of address space of the new process**
    - Child duplicate of parent
    - Child has a program loaded into it

Example: Process creation on UNIX:

In UNIX A new process is created by the fork() system call. The new process consists of a copy of the address space of the original process. This mechanism allows the parent process to communicate easily with its child process. Both processes (the parent and the child) continue execution at the instruction after the fork () , with one difference: the return code for the fork() is zero for the new (child) process, whereas the (nonzero) process identifier of the child is returned to the parent.

We now have *two different processes running copies of the same program*. The only difference is that the value of pid (the process identifier) for the child process is zero, while that for the parent is an integer value greater than zero (in fact, it is the actual pid of the child process). The child process inherits privileges and scheduling attributes from the parent, as well certain resources, such as open files.

Typically, the *exec() system call is used after a fork() system call by one of the two processes to replace the process's memory space with a new program.* The exec() system call loads a binary file into memory (destroying the memory image of the program containing the exec() system call) and starts its execution. In this manner, the two processes are able to communicate and then go their separate ways. The parent can then create more children; or, if it has nothing else to do while the child runs, it can issue await() system call to move itself off the ready queue until the termination of the child.

```
int main()
{
pid_t pid;

    /* fork a child process */
    pid = fork();

    if (pid < 0) { /* error occurred */
      fprintf(stderr, "Fork Failed");
      return 1;
    }
    else if (pid == 0) { /* child process */
      execlp("/bin/ls","ls",NULL);
    }
    else { /* parent process */
      /* parent will wait for the child to complete */
      wait(NULL);
      printf("Child Complete");
    }

    return 0;
}
```

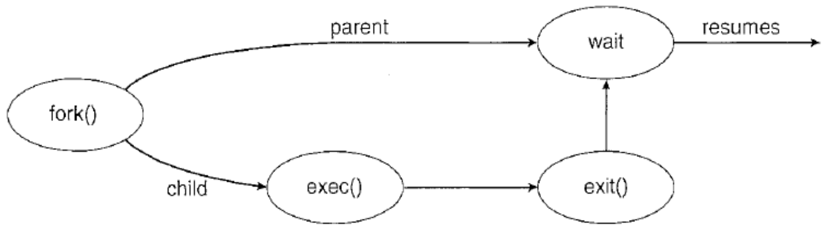


Figure 3.11 Process creation using `fork()` system call.

In the given example, The child process overlays its address space with the UNIX command /bin/ls using the execlp() system call (same as exec). The parent waits for the child process to complete with the wait() system call. When the child process completes (by either implicitly or explicitly invoking exit ()) the parent process resumes from the call to wait (),where it completes using the exit() system call.

Example 2: Process creation on Windows

On Windows, CreateProcess() is used to create a child process. Unlike UNIX where a copy of the parent address space is created for the child, On windows, CreateProcess() requires loading a specified program into child address space at the time of process creation. Ex: Consider a pseudocode for program that creates a child process that loads an mspaint.exe.

```
allocate required memory for new process
if( ! CreateProcess(/*NULL, mspaint.exe,...*/))
      Print failure message and exit
parent will wait for child to complete
close all opened file handles
exit
```

Process Termination

A process terminates when it finishes executing its last statement and asks the operating system to delete it by using exit() system call. When a process ends, all of its *system resources are freed up*, *open files flushed and closed*, etc. The process termination *status and execution times are returned to the parent* if the parent is waiting for the child to terminate, or eventually returned to init if the process already became an orphan.

Alternatively, a process may be terminated by another process (the parent process) via an appropriate system call (Ex: TerminateProcess() in Win32, abort()). **A parent may terminate the execution of one of its children for the following reasons:**

- Child has exceeded allocated resources
- Task assigned to child is no longer required
- The parent is exiting, Some operating systems do not allow child to continue if its parent terminates - "**cascading termination**"

On UNIX systems, a process can be terminated using exit() system call; its parent process may wait for the termination of a child process by using the wait() system call. The wait() system call returns the process identifier of a terminated child so that the parent can tell which of its children has terminated.

If the parent terminates, however, all its children have assigned as their new parent the init process. Thus, the children still have a parent to collect their status and execution statistics.

InterProcess Communication

1. **What are cooperating processes? Give reasons for process cooperation** [5M]
2. What are the benefits offered by cooperating processes? Explain direct and indirect interprocess communication [8M]
3. What are co-operating processes? Describe the mechanism of inter process communication using shared memory in a producer-consumer problem
4. **Distinguish between symmetric and asymmetric communication between processes**.
5. Explain the 2 fundamental approaches to inter process communication [5M]
6. **Explain direct and indirect communications with Message Passing System** [4M]

Processes within a system may be independent or cooperating.

- A process is **independent** if it cannot affect or be affected by the other processes executing in the system. It does not share data with any other process.

- A **cooperating** process on the other hand can be affected or can affect other processes in the system.

There are several **reasons for providing an environment that allows process cooperation**:

→ **Information sharing**: Since several users may be interested in the same piece of information (for instance, a shared file), we must provide an environment to allow concurrent access to these types of resources.

→ **Computation speedup**: If a particular task needs to run faster, it must be broken up into subtasks, each of which will be executing in parallel with the others.

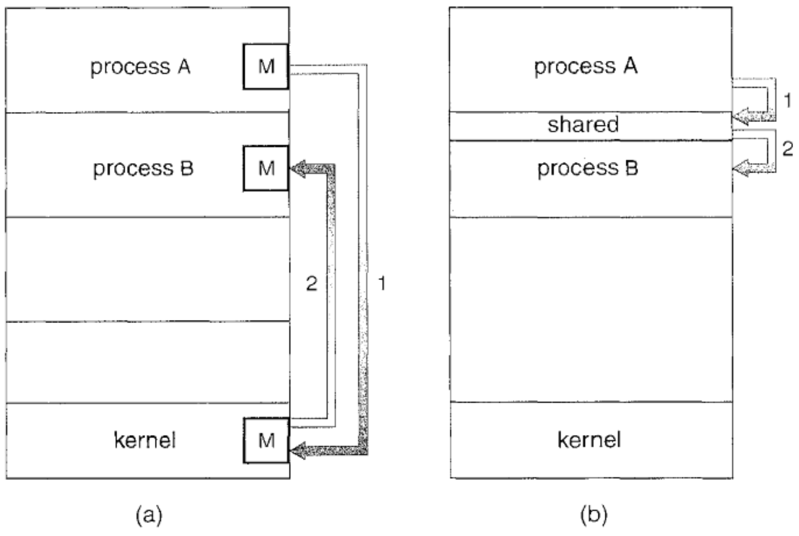
→ **Modularity** – Co-operating environment allows system construction in a modular fashion, dividing the system functions into separate processes or threads thereby simplifying the structure

→ **Convenience**: Even an individual user may have many tasks to work on at one time. For instance, a user may be editing, printing, and compiling in parallel.

Co-operating processes require an inter process communication (IPC) mechanism that allow them to exchange data and information.

There are **two fundamental models of inter process communication** –

1. Shared memory 2. Message Passing



Communication models: a) Message passing b) Shared memory

Shared Memory Model

Shared Memory model *allows a memory region to be simultaneously accessed* by multiple programs with intent to provide communication among them.

→ One process will *create an area in memory which other processes can access*. Other processes that wish to communicate using this shared memory segment *must attach* it to their address space

→ Normally the OS prevents processes from accessing the memory of another process, but the Shared Memory features in the OS can allow data to be shared. Since both processes can access the shared memory area like regular working memory, this is a very *fast way of communication*.

→ This method is preferred if the *communicating processes exist on the same system*

Producer-Consumer Example Using Shared Memory

- A producer produces information that is consumed by a consumer process.
- One solution to the producer consumer problem uses shared memory. To allow producer and consumer processes to run concurrently, we must have available a **buffer** of items that can be filled by the producer and emptied by the consumer. This buffer will reside in a region of memory that is shared by the producer and consumer processes. A producer can produce one item while consumer is consuming another item. The producer and consumer must be synchronized so that the consumer does not try to consume an item that has not yet been produced.

  Two types of buffers can be used –
    - Unbounded buffer : places no limit on the size of the buffer
    - Bounded: assumes a fixed size buffer.

The following variable reside in a region of memory shared by the producer and consumer process.

```
#define BUFFER_SIZE 10

typedef struct {
     . . .
} item;

item buffer[ BUFFER_SIZE ];
int in = 0;
int out = 0;
```

The shared buffer is implemented as a circular array with two logical pointers: in and out. The variable *in* points to the next free position in the buffer; *out* points to the first full position in the buffer. The buffer is empty when in== out; the buffer is full when ((in+ 1)% BUFFER_SIZE) == out.

The code for the producer and the consumer is as below.

```
item nextProduced;
             while( true ) {
                    /* Produce an item in nextProduced */
                    while( ( ( in + 1 ) % BUFFER_SIZE ) ==out)
                          ; /* Do Nothing */
                    buffer[ in ] = nextProduced;
                    in = ( in + 1 ) % BUFFER_SIZE;
             }                   Producer process
item nextConsumed;
while( true ) {
while( in == out )
      ; /* Do nothing */
nextConsumed = buffer[ out ];
out = ( out + 1 ) % BUFFER_SIZE;
/* Consume the item in nextConsumed*/
}
             Consumer process
```

The producer process has a local variable nextProduced in which the new item to be produced is stored.
The consumer process has a local variable nextConsumed in which the new item to be consumed is stored.

Message Passing Model

This method allows processes to communicate and to synchronize their actions without sharing the same address space. It is particularly useful in a distributed environment where the communicating processes may reside on different computers connected by a network.

A message passing facility provides at least two operations: send (message) and receive (message). Messages sent by a process can be of either fixed or variable size.

If two processes wish to communicate, they need to:

- → establish a communication link between them
- → exchange messages via send/receive

Here are several methods for logically implementing a link and the send ()/ receive () operations:

- ➢ **Direct or indirect communication**
- ➢ **Synchronous or asynchronous communication**
- ➢ **Automatic or explicit buffering**

Issues related to the above operations:

1. Naming:
   - → Processes that want to communicate must have a way to refer to each other. They can use either *direct* or *indirect* communication.

Direct Communication

- → Under direct communication, each process that wants to communicate *must explicitly name the recipient or sender of the communication*. The send() and receive() primitives are defined as-

  *send (**P**, message)* – send to process P
  *receive(**Q**, message)* – receive from Q

- → A communication link in this scheme has the following properties –
  - A link is established automatically between every pair of processes that want to communicate. The processes need to know only each other's identity to communicate.
  - A link is associated with exactly two processes.
  - Between each pair of processes, there exists exactly one link.

→ The method of addressing another process may be **symmetric** or **asymmetric**

- **Symmetric** - sender process and receiver process must name the other to communicate

  *send (**P**, message)* – send to process P
  *receive(**Q**, message)* – receive from Q

- **Asymmetric -** sender names the recipient; the recipient is not required to name the sender.

  *send (**P**, message)* – send to process P
  *receive(**id**, message)* – rx from any; system sets id = sender

- **Disadvantage**: a process must know the name or ID of the process(es) it wishes to communicate with.

Indirect Communication:

- → With indirect communication, the *messages are sent to and received from mailboxes, or ports*.

→ A mail box can be viewed abstractly as an object into which messages can be placed by processes and from which messages can be removed.

→ Each mail box has a *unique identification*. A process can communicate with some other process via a number of different mail boxes. Two processes *can communicate only if the processes have a shared mail box*

send(A, message) – send a message to mailbox A
receive(A, message) – receive a message from mailbox A

→ In this scheme, **a communication link has the following properties**

- A link is established only if *both members of the pair have a shared mail box*.
- A link may be associated with more than two processes.
- Between each pair of communicating processes, there may be a number of different links, with each link corresponding to one mail box.

→ A mail box may be owned either by a process or by the OS. If the mail box is owned by a process, then we distinguish between the owner and the user. *When a process that owns the mail box terminates, the mail box disappears*. Any process that subsequently sends a message to this mail box must be notified that the mail box no longer exists. *A mail box owned by the OS is independent and is not attached to any particular process.*

2. Synchronization:

Communication between processes takes place through calls to send() and receive() primitives. Message passing may be either **blocking or non blocking** – also known as **synchronous and asynchronous**.

- **Blocking send**: The *sending process is blocked until the message is received* by the receiving process or the mail box.
- **Non blocking send**: The *sending process sends the message and resumes operation*.
- **Blocking receive**: The *receiver blocks until a message is available*.
- **Non blocking receive**: The *receiver retrieves either a valid message or a null*.

3. Buffering

Messages exchanged by communicating processes *reside in a temporary queue*. Such queues can be implemented in three ways –

- **Zero capacity**: The queue has the maximum length of zero; thus *the link cannot have any messages waiting in it.* In this case, the sender must block until the recipient receives the message.

- **Bounded capacity**: The queue *has finite length n;* thus, at most n messages can reside in it. If the queue is not full when a new message is sent, the message is placed in the queue (either the message is copied or a pointer to the message is kept), and the sender can continue execution without waiting. The link's capacity is finite, however. If the link is full, the *sender must block until space is available* in the queue.
- **Unbounded capacity:** The queues *length is infinite*; thus any number of messages can wait in it. The *sender never blocks.*