**Operating Systems I**

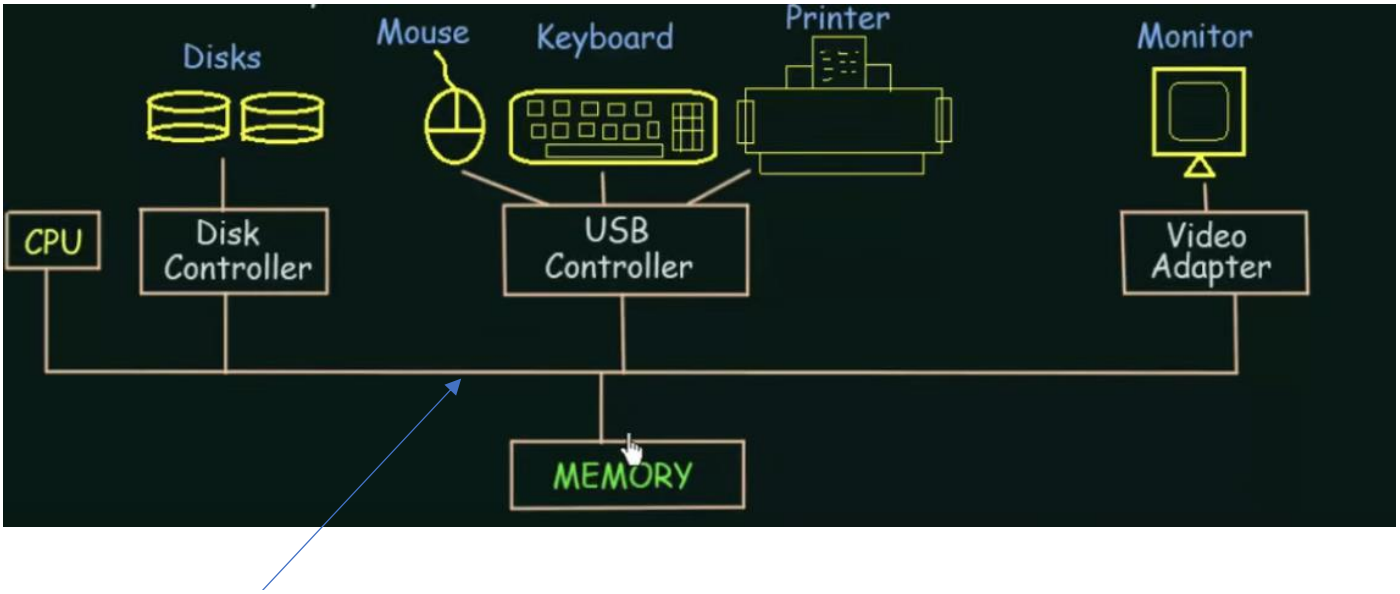
1. Introduction and basics

A modern general-purpose computer system consists of one or more CPUs and a number of device controllers connected through a common bus that provide access to shared memory.

Each device controller is in charge of a specific type of device.

The CPU and the device controllers can execute concurrently, competing for memory cycles.

To ensure orderly access to the shared memory, a memory controller is provided whose function is to synchronize access to the memory.




**Common system bus**

**Bootstrap Program** – The initial program that runs when a computer is powered up or rebooted
It is stored in the ROM
It must know how to load the OS and start executing the system
It must locate and load the OS Kernel in to memory

**Interrupt** – The occurrence of an event is usually signaled by and Interrupt from hardware or software.
Hardware may trigger an interrupt at any time by sending a signal to the CPU, usually by the way of the system bus.

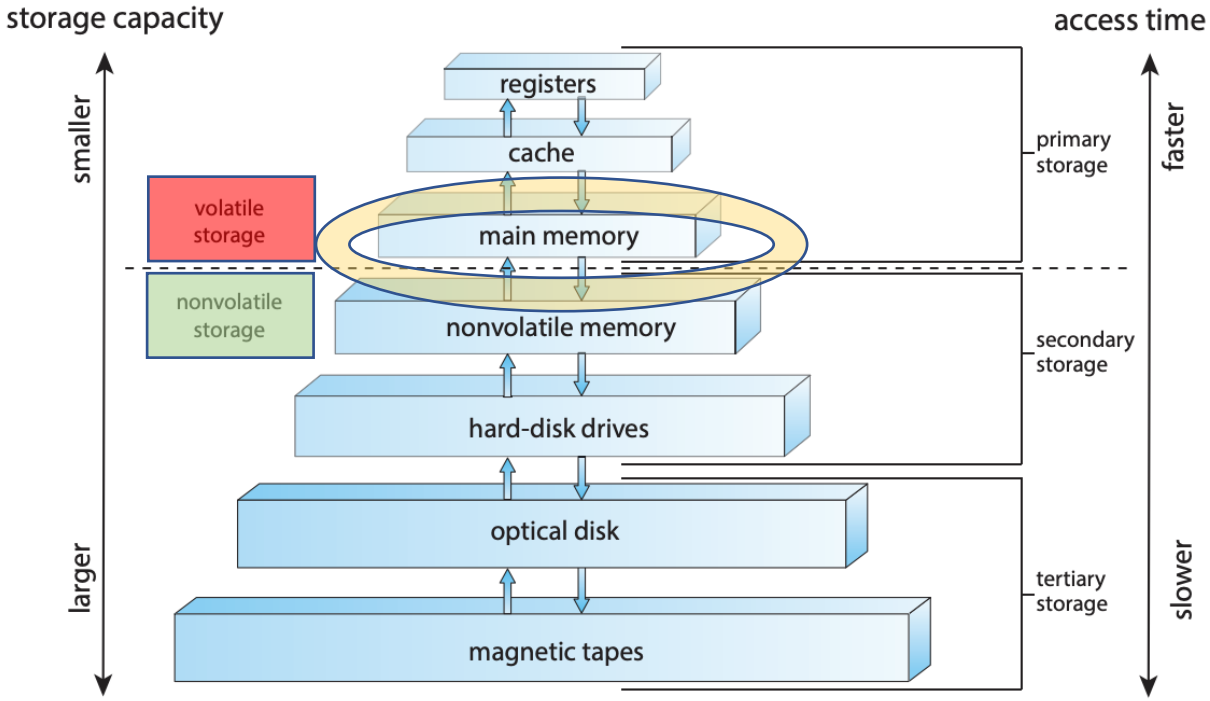
**System Call (Monitor call)** – software may trigger an interrupt by executing a special operation called System
Call

When the CPU is interrupted, it stops what it is doing and immediately transfers execution to a fixed location
The fixed location usually contains the starting address where the service routine of the interrupt is located

The interrupt Service Routine Executes

On completion, the CPU resumes the interrupted computation

## Storage structure




**Cost per bit increases** as we go up the structure
**Access time decreases** as we go up the structure
**Physical size deceases** as we go up the structure

**Electronic disk -** can be designed to be either volatile or nonvolatile…with a hidden magnetic hard disk and
battery backup (ex : Flash Memory)

**Volatile** – Loses its content when power is removed
**Non-volatile** – Retains its content even when power is removed

## Input/Output structure

Storage is one of many I/O devices

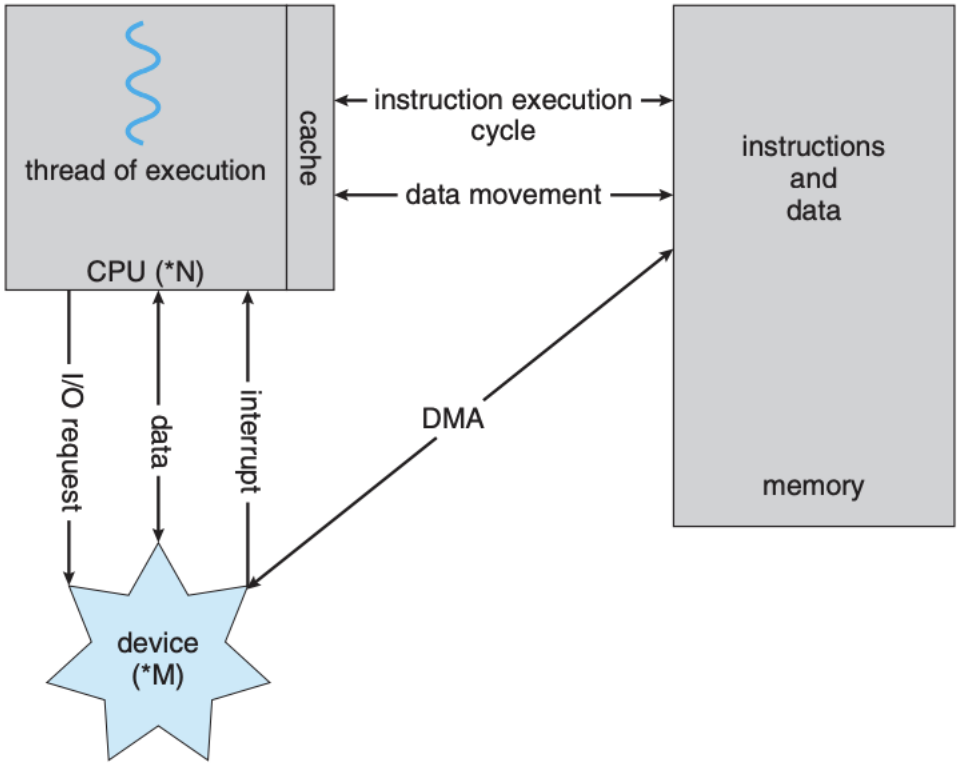
A large portion of operating system code is dedicated to managing I/O, both because of its importance to the reliability and performance of a system and because of the varying nature of the devices.

A general-purpose computer system consists of CPUs and multiple device controllers that are connected through a common bus.


Typically, operating systems have a device for each device controller

Direct Memory Access

Direct memory access (DMA) is a method that allows an input/output (I/O) device to send or receive data directly to or from the main memory, bypassing the CPU to speed up memory operations.
The process is managed by a chip known as a DMA controller (DMAC).




Computer System Architecture

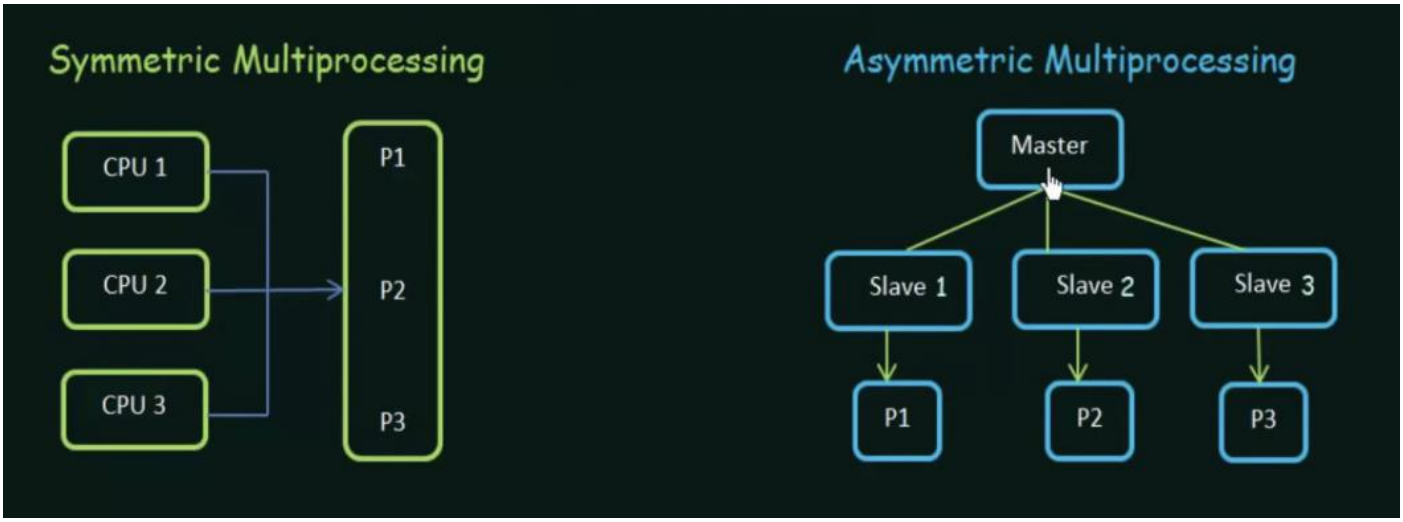
**Single processor systems**
One main CPU capable of executing general purpose instruction set
Other special purpose processors are also present which perform device specific tasks

**Multiprocessor systems**
Aka parallel systems or tightly coupled systems.
Has two or more processors in close communication, sharing the computer bus and sometimes the clock, memory and peripheral devices.

**Advantages**

- Increased throughput – large amount of data is output
- Economy of scale – all processors can share resources
- Increased reliability – if one processor fails the system can keep running without a total failure

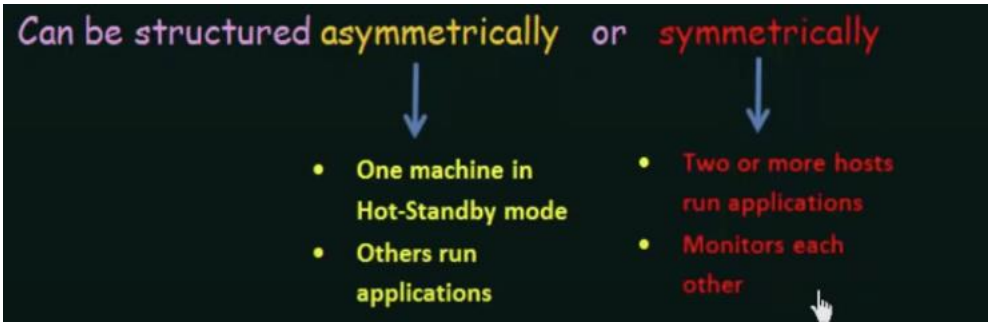
Types of Multiprocessor systems




• CPU—The hardware that executes instructions.
• Processor—A physical chip that contains one or more CPUs.
• Core—The basic computation unit of the CPU.
• Multicore— Including multiple computing cores on the same CPU.
• Multiprocessor— Including multiple processors.

Although virtually all systems are now multicore, we use the general term CPU when referring to a single computational unit of a computer system and core as well as multicore when specifically referring to one or more cores on a CPU.

**Clustered Systems**
Clustered Systems gather together multiple CPUs to accomplish computational work
Has two or more individual systems coupled together.
High availability provides increased reliability




In 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.

In symmetric clustering, two or more hosts are running applications and are monitoring each other. This structure is obviously more efficient, as it uses all of the available hardware. However, it does require that more than one application be available to run

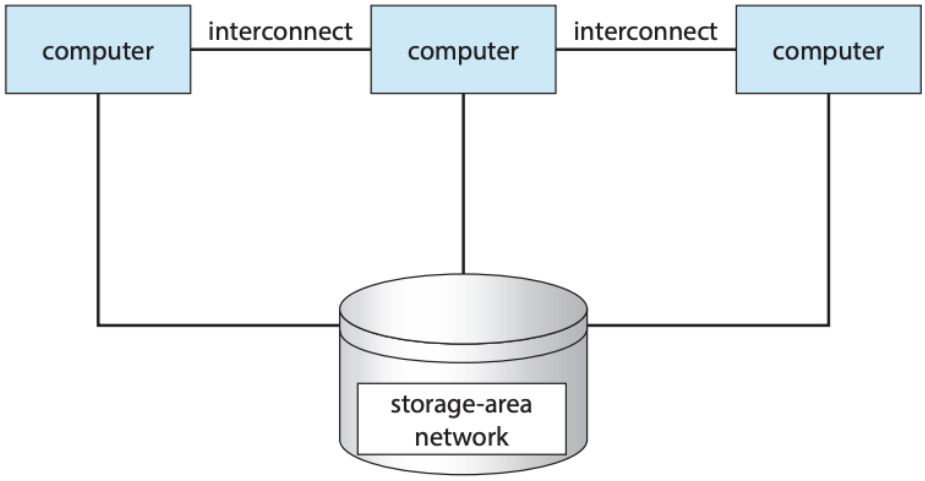



**Figure 1.11** General structure of a clustered system.

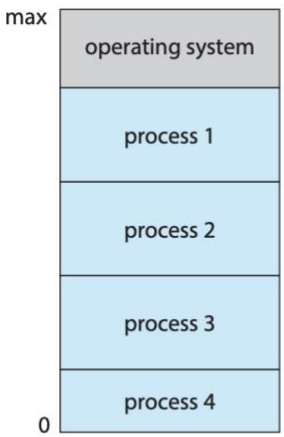
## 2. Operating System Structure

Operating systems vary greatly in their makeup internally

COMMONALITIES

i. Multiprogramming
ii. Time Sharing (Multitasking)

**Multiprogramming**
Multiprogramming increases CPU utilization, as well as keeping users satisfied, by organizing programs so that the CPU always has one to execute. In a multiprogrammed system, a program in execution is termed a process.
When *that* process needs to wait, the CPU switches to *another* process, and so on. Eventually, the first process finishes waiting and gets the CPU back. As long as at least one process needs to execute, the CPU is never idle.




Memory layout for a multiprogramming system.

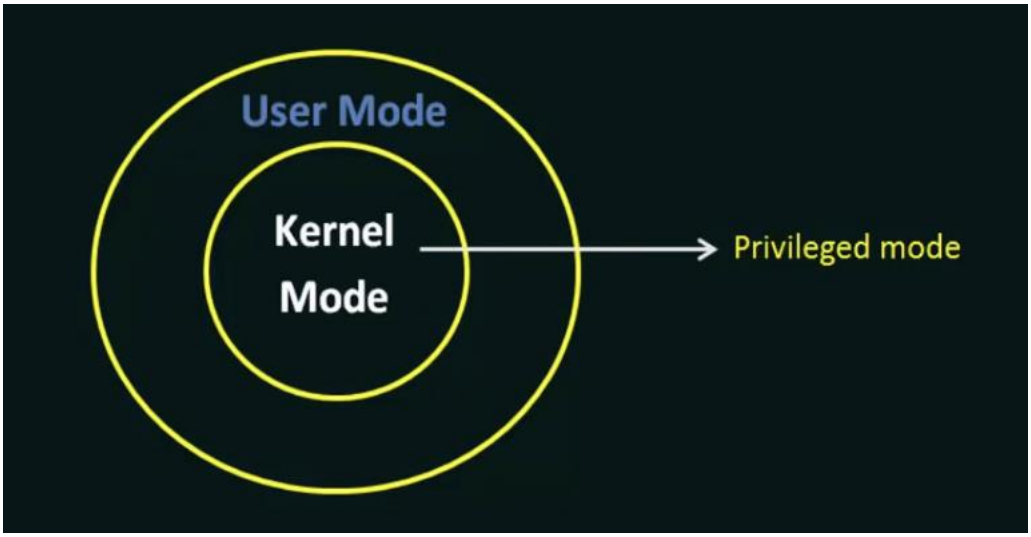
Multiprogrammed systems provide an environment in which the various system resources (for example CPU, memory and peripheral devices) are utilised effectively, but they do not provide for user interaction with the computer system. (The user can only see the output of the running programs and not interact with it directly.)

## Operating System Services

1. User Interface
    a. Command Line Interface (CLI)
    b. Graphical User Interface (GUI)
2. Program execution
3. Resource allocation
4. File-System manipulation
5. Mass-Storage Management
6. Cache Management
7. I/O operations
8. Security and protection
9. Accounting statistics of usage (ex: Activity Monitor in MacOS)
10. Error detection
11. Communications
    one process needs to exchange information with another process. Such communication may occur between processes that are executing on the same computer or between processes that are executing on different computer systems tied together by a network. Communications may be implemented via shared memory

**Command Interpreters** – aka *SHELLs.* A command interpreter allows the user to interact with a program using commands in the form of text lines. It was frequently used until the 1970's. However, in modern times many command interpreters are replaced by graphical user interfaces and menu-driven interfaces. s. For example, on UNIX and Linux systems, a user may choose among several different shells, including the C shell, Bourne-Again shell, Korn shell, and others

## System Calls

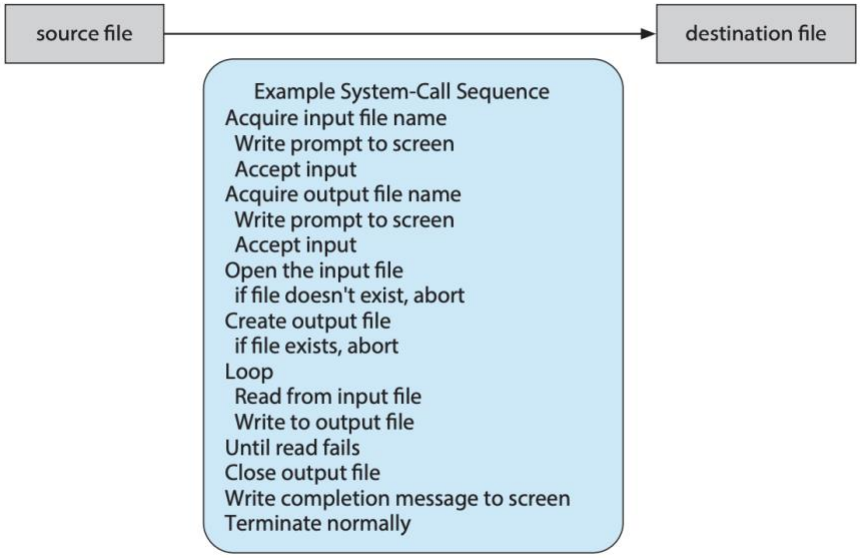



If a program is running in User mode and if it happens to crash the whole system would not crash
If a program is running in Kernel mode and if it happens to crash the whole system would come to a halt. Because of that User mode is safer.

User mode doesn't have privileges to access resources like I/O and memory. Kernel mode is the privileged mode. If a program is running on User mode and wants access to a system resource, the program makes a **call** to the OS and for an instance it switches from User mode to Kernel mode to access the resources. That call is called a **System call**.

System call is the programmatic way in which a computer program requests a service from the kernel of the operating system
These calls are generally available as routines written in C and C++

Example of a sequence of system calls when a content of a file is copied to another file




## Types of System Calls

There are major 6 categories of system calls

1. **Process control** system calls
   - create process, terminate process
   - load, execute
   - get process attributes, set process attributes
   - wait event, signal event
   - allocate and free memory

2. **File management** system calls
   - create file, delete file
   - open, close
   - read, write, reposition
   - get file attributes, set file attributes

3. **Device management** system calls
   - request device, release device
   - read, write, reposition
   - get device attributes, set device attributes
   - logically attach or detach devices

4. **Information maintenance** system calls
   - get time or date, set time or date
   - get system data, set system data
   - get process, file, or device attributes
   - set process, file, or device attributes

5. **Communications** system calls
   - create, delete communication connection
   - send, receive messages
   - transfer status information
   - attach or detach remote devices

6. **Protection** system calls
   - get file permissions
   - set file permissions

## System services

- File management
  These programs create, delete, copy, rename, print, list, and generally access and manipulate files and directories.

- Status Information
  Some programs simply ask the system for the date, time, amount of available memory or disk space, number of users, or similar status information

- Programming language support
  Compilers, assemblers, debuggers, and interpreters for common programming languages (such as C, C++, Java, and Python) are often provided with the operating system or available as a separate download.

- Program loading and execution
  Once a program is assembled or compiled, it must be loaded into memory to be executed

- Communications
  These programs provide the mechanism for creating virtual connections among processes, users, and computer systems

- File modification
  Several text editors may be available to create and modify the content of files stored on disk or other storage devices. There may also be special commands to search contents of files or perform transformations of the text.

## Operating-System Design and Implementation

Requirements

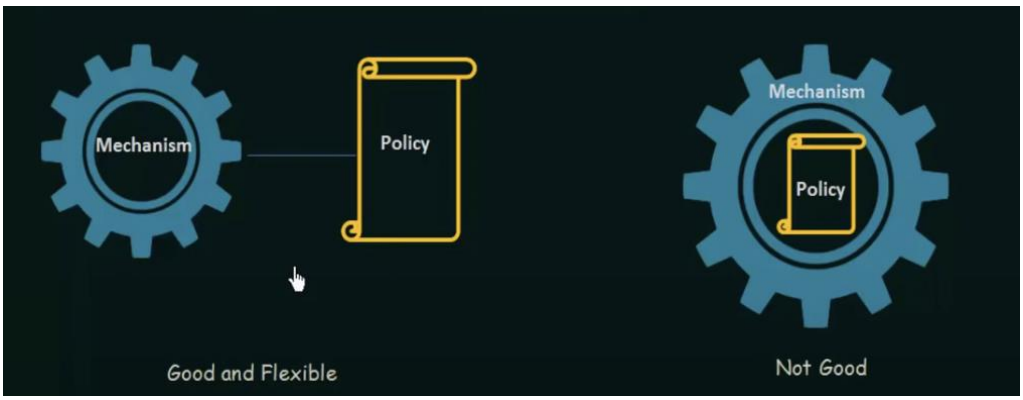
1. User goals
   the system should be convenient to use, easy to learn and use, reliable, safe and fast

2. System goals
   The system should be easy to design, implement, maintain, operate. It should be flexible, reliable, error free and efficient.

## Mechanisms and Policies

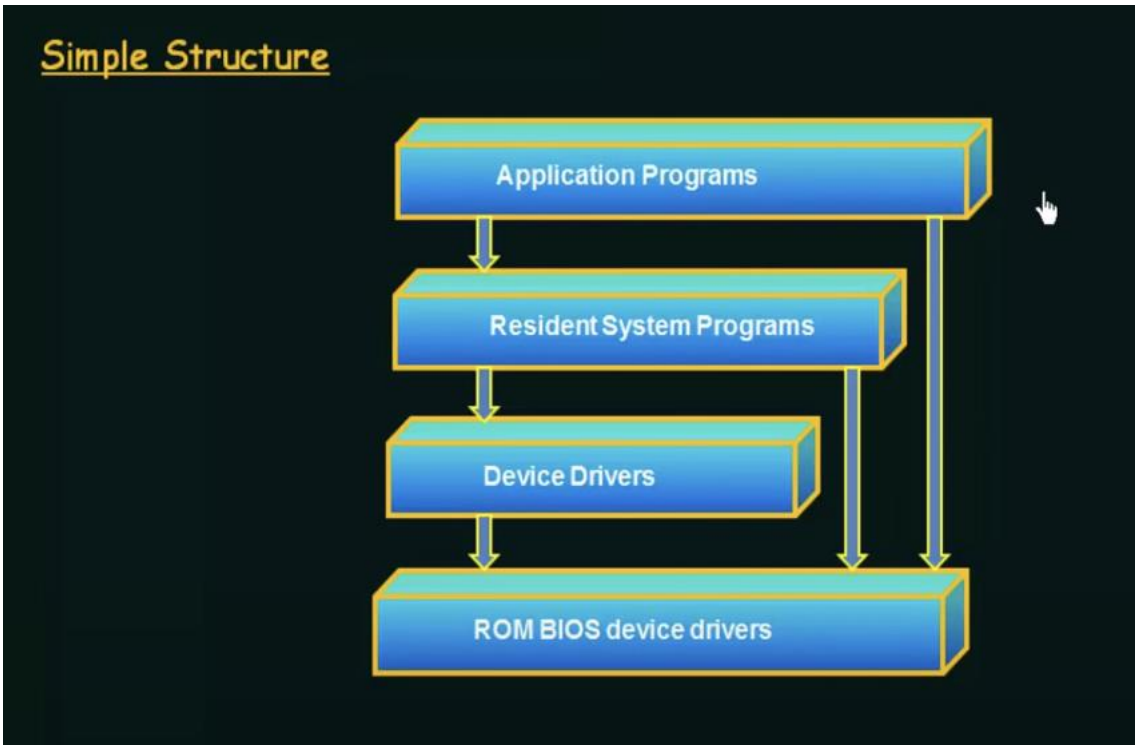
Mechanisms determine **how** to do something
Policies determine **what** will be done

An important principle is to separate policy from mechanism

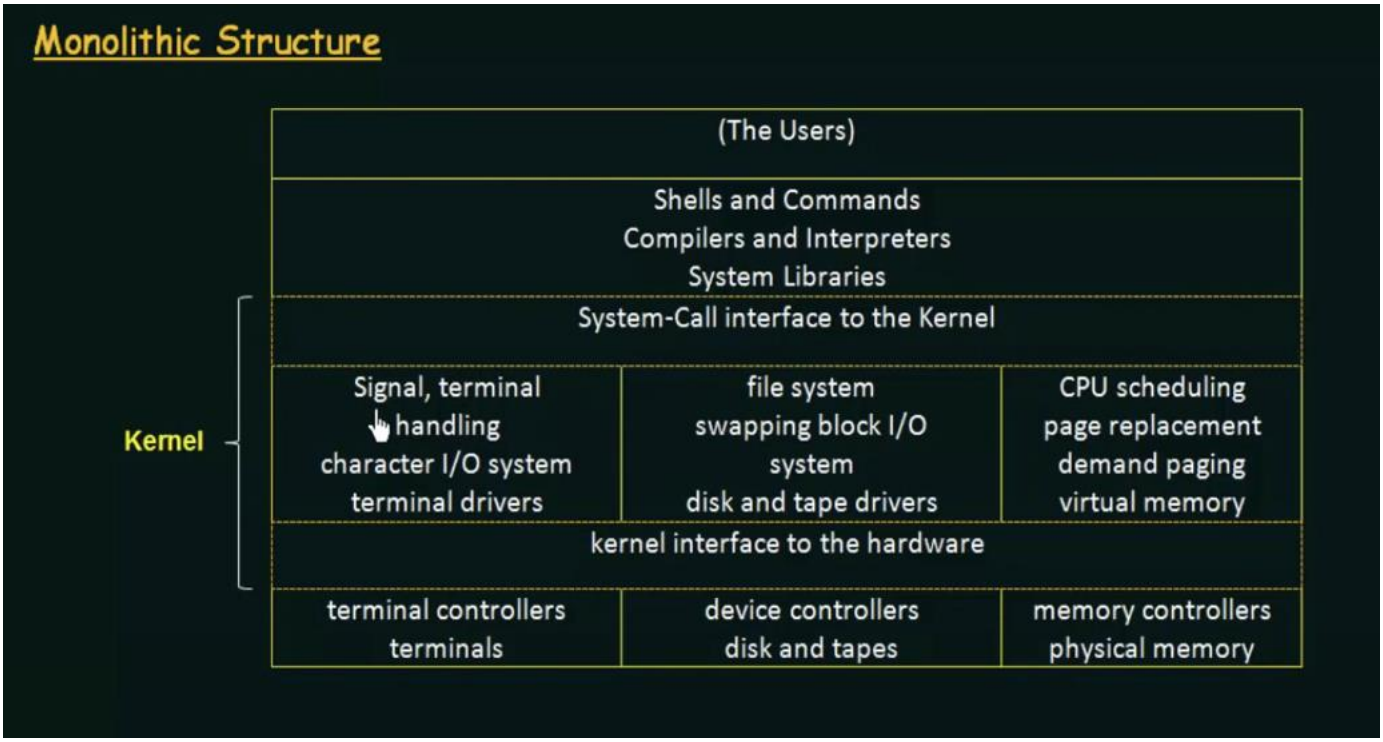



A change in policy should not affect the mechanism of the OS. The separation of policy and mechanism is important for flexibility. Policies are likely to change across places or over time. In the worst case, each change in policy would require a change in the underlying mechanism. A general mechanism flexible enough to work across a range of policies is preferable.

Operating-System Structure




Even malicious programs have access to hardware. This structure is not well protected, structured or well defined. Not a good structure.
Ex: Microsoft Disk Operating System



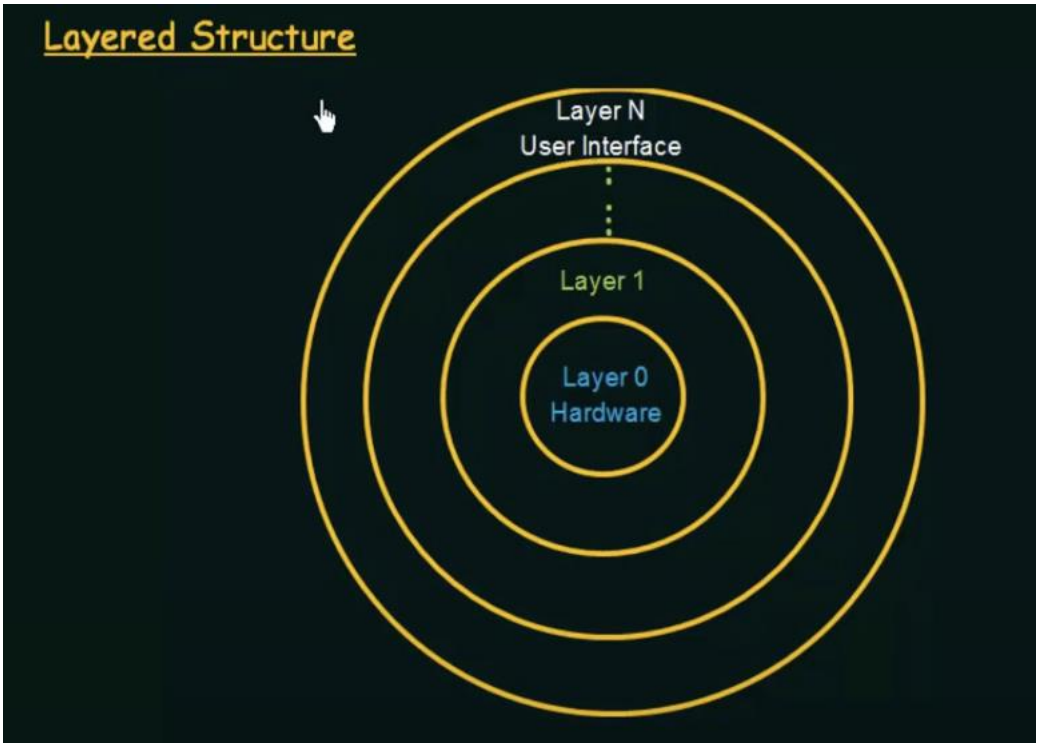

The simplest structure for organizing an operating system is no structure at all. That is, place all of the functionality of the kernel into a single, static binary file that runs in a single address space. This approach—known as a monolithic structure

Ex: original UNIX Operating System

consists of two separable parts: the kernel and the system programs. The kernel is further separated into a series of interfaces and device drivers

The monolithic approach is often known as a tightly coupled system because changes to one part of the system can have wide-ranging effects on other parts.

Despite the apparent simplicity of monolithic kernels, they are difficult to implement and extend. Monolithic kernels do have a distinct performance advantage, however: there is very little overhead in the system-call interface, and communication within the kernel is fast. Despite the drawbacks of monolithic kernels, their speed and efficiency explain why we still see evidence of this structure in the UNIX, Linux, and Windows.

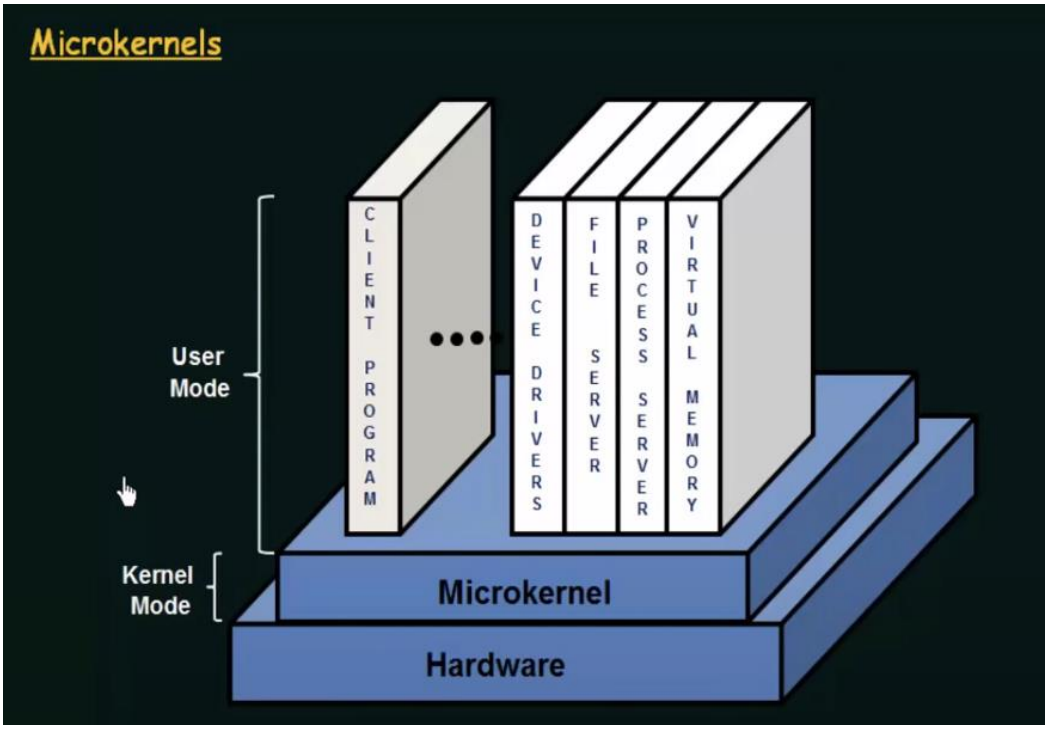



A system can be made modular in many ways. One method is the layered approach, in which the operating system is broken into a number of layers (levels). The bottom layer (layer 0) is the hardware; the highest (layer N) is the user interface

The advantage of this modular approach is that changes in one component affect only that component, and no others, allowing system implementers more freedom in creating and changing the inner workings of the system.

The main advantage of the layered approach is simplicity of construction and debugging. . The layers are selected so that each uses functions (operations) and services of only lower-level layers.

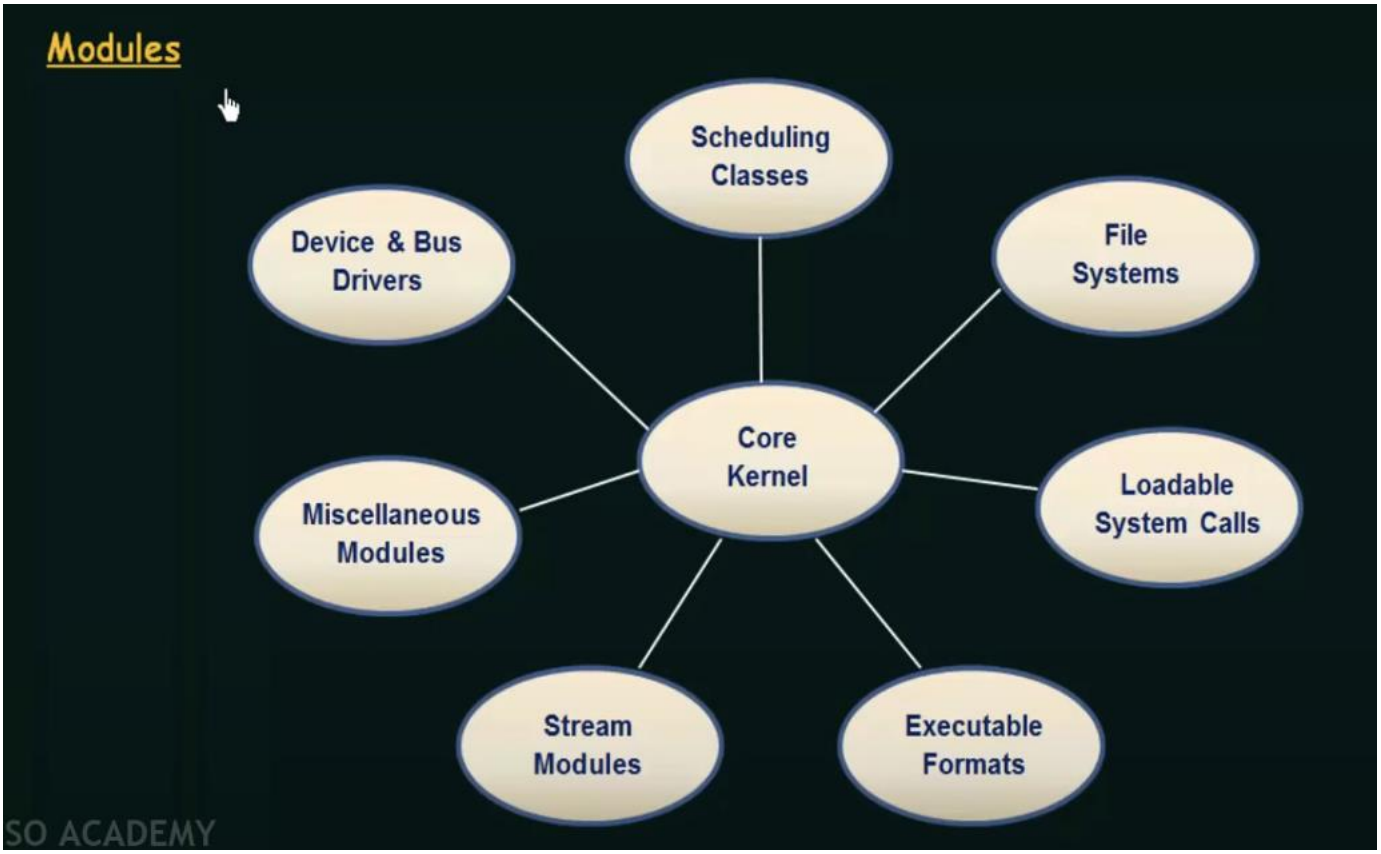
the overall performance of such systems is poor due to the overhead of requiring a user program to traverse through multiple layers to obtain an operating-system service.




This method structures the operating system by removing all nonessential components from the kernel and implementing them as userlevel programs that reside in separate address spaces. The result is a smaller kernel.

The main function of the microkernel is to provide communication between the client program and the various services that are also running in user space.

it makes extending the operating system easier, easier to port from one hardware design to another, more security and reliability, since most services are running as user—rather than kernel—processes. If a service fails, the rest of the operating system remains untouched.

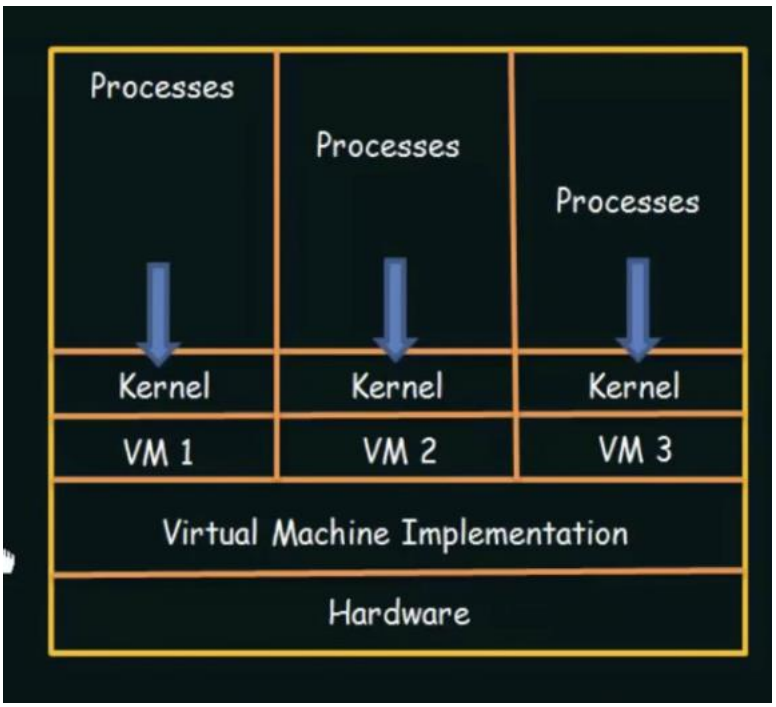



The best current methodology for operating-system design involves using loadable kernel modules (LKMs).

The idea of the design is for the kernel to provide core services, while other services are implemented dynamically, as the kernel is running

Module can communicate with each other through the Kernel easily with any overhead.

Virtual Machines



Virtual machine software – runs in Kernel mode
Virtual machine itself – runs in User mode

Protection of system resources is an advantage of using VMs.

## Operating System Generation and System Boot

There are two approaches to design and implement operating systems,

1. Design, code and implement an operating system specifically for one machine at one site
2. Operating systems are designed to run on any of a class of machines at a variety of sites with a variety of peripheral devices.

The system must then be configured or generated for each specific computer site; a process sometimes known as a system generation (SYSGEN) is used for this

The following kinds of information must be determined by the SYSGEN program

- What CPU is to be used
- How much is available
- What devices are available
- What operating-system options are desired

**System Boot**

The procedure of starting a computer by loading the Kernel is known as booting the system

On most computer systems, a small piece of code known as the bootstrap program (bootstrap loader) locates the Kernel

This program is in the form of read-only-memory (ROM), because the RAM is in an unknown state at the system startup (because its volatile). ROM is convenient because it needs no initialization and cannot be infected by a computer virus.

The bootstrap program traverses the file system and located the Kernel, loads it in to memory and starts its execution. Now the system is said to be RUNNING.

## 3. Processes

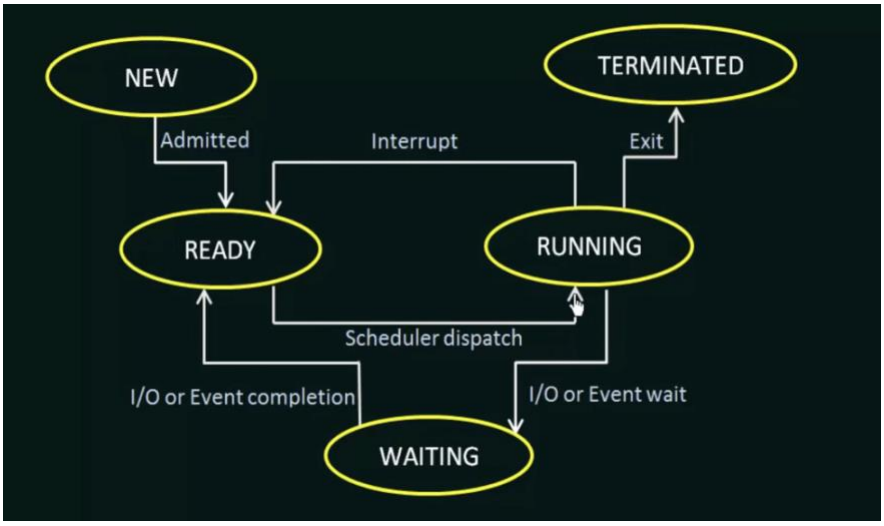
### Process management

**Process** – is a program in execution
**Thread** – unit of execution within a process. A process can have anywhere from just one thread to many threads.

### Process state

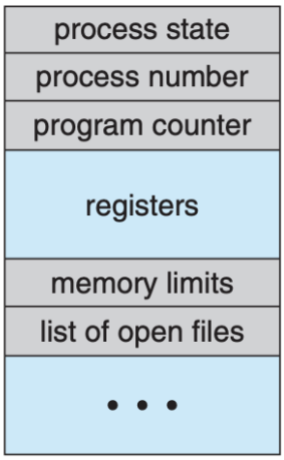
A process changes its state while executing.
State of a process means the current activity of the process

- NEW – the process is being created
- RUNNING – instructions are being executed
- WAITING – the process is waiting for something (Ex an I/O completion)
- READY – the process is waiting to be assign to a processor
- TERMINATED - process has finished execution




### Process Control Block

Each process is represented in the operating system by a process control block (PCB)—also called a task control block.




It contains many pieces of information associated with a specific process, including these:

• **Process state**. The state may be new, ready, running, waiting, halted, and so on.

• **Program counter**. The counter indicates the address of the next instruction to be executed for this process.

• **CPU registers**. They include accumulators, index registers, stack pointers, and general-purpose registers, plus any condition-code information. Along with the program counter, this state information must be saved when an interrupt occurs, to allow the process to be continued correctly afterward when it is rescheduled to run.

• **CPU-scheduling information**. This information includes a process priority, pointers to scheduling queues, and any other scheduling parameters.

• **Memory-management information**. This information may include such items as the value of the base and limit registers and the page tables, or the segment tables, depending on the memory system used by the operating system.

• **Accounting information.** This information includes the amount of CPU and real time used, time limits, account numbers, job or process numbers, and so on.

• **I/O status information.** includes the list of I/O devices allocated to the process, a list of open files, and so on.

## Process Scheduling

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.

To meet these objectives, the process scheduler selects an available process (possibly from a set of available processes) for execution on the CPU.

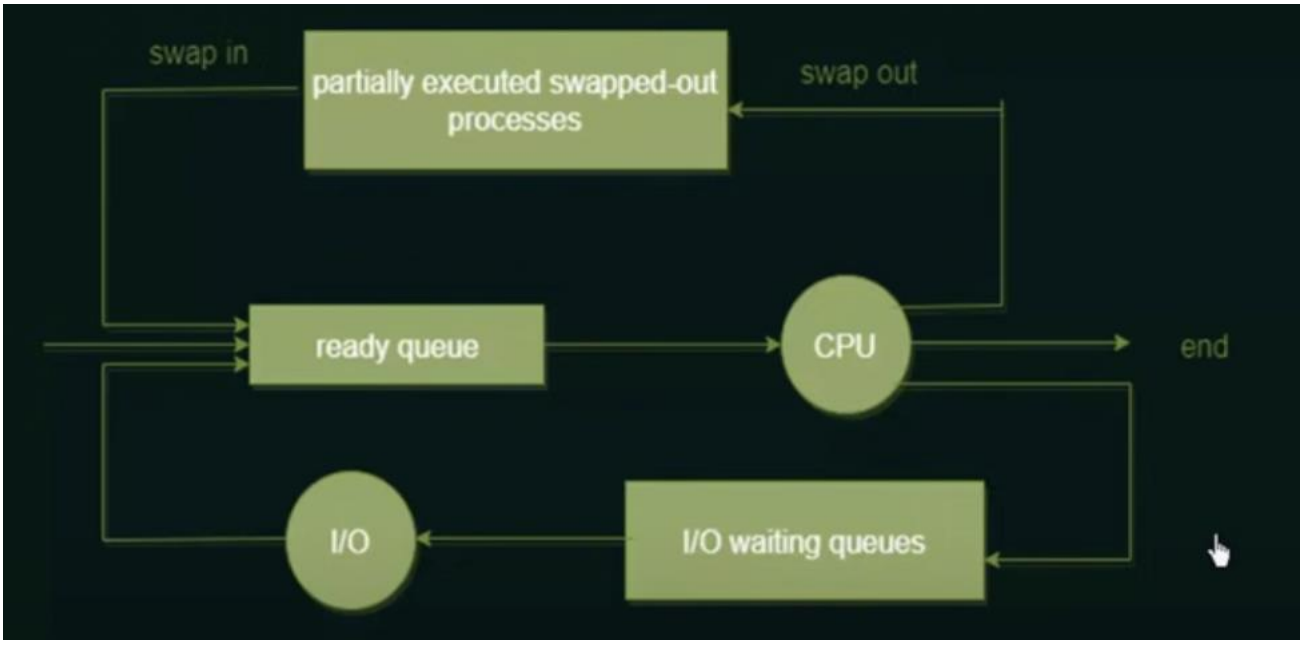
- For a single-processor system, **there will never be more than one running process** in a given time
- If there are more processes, the rest will have to wait until the CPU is free and can be rescheduled.

In order to help with processes scheduling we have **Scheduling Queues.** There are two kinds of queues.

**Job Queue –** as processes enter the system, they are put in to job queue, which consists of all processes in the system.

**Ready Queue** – the processes that are residing in the main memory and are ready and waiting to execute are kept on a list called the ready queue.

Job Queue




Context Switch

Interrupts cause the operating system to change a CPU from its current task and to run a Kernel routine.

Such operation happens frequently on a general-purpose system

When an interrupt occurs, the system needs to save the current **CONTEXT** of the process currently running on the CPU so that it can restore that context when its processing is done, basically suspending the process and then resuming it.

The context is represented in the PCB of the process

Operations on Processes (Process Creation)

A process may create several new processes via a create-process system call, during the course of execution.

The **creating process** is called the **parent process** and the **newly created prosses** are called the **children processes**.

Each of these new processes may in turn create other processes forming a tree of processes.

Each is process is identified by their unique PID (process id)

When a process creates a new process, two possibilities exist in terms of execution:

1. The parent continues to execute concurrently with its children
2. The parent waits until some or all of its children have terminated.

There are also two possibilities in terms of the address of the new process:

1. The child process is a duplicate of the parent process (it has the same program and data as the parent)
2. The child process has a new program loaded into it.

A process terminates when it finishes executing its final statement and asks the OS to delete it by using the exit() system call

At that point, the process may return a status value (typically an integer) to its parent process (via wait() system call)

All the resources of the process – including physical and virtual memory, open files, and I/O buffers – are deallocated by the OS.

Termination can occur in other circumstances as well:

A process can cause the termination of another process via an appropriate system call.

Usually, such system call be invoked only by the parent of the process that is to be terminated.

Otherwise, users could arbitrarily kill each other's' jobs.

A parent may terminate the execution of one of its children for a variety of reasons, such as these,

- The child has exceeded its usage of some of the resources that it has been allocated. (to determine whether it has occurred, the parent must have a mechanism to inspect the state of its children)
- The task assigned to the child is no longer required.
- The parent is exiting, and the OS does not allow a child to continue if its parent terminated.

## Inter-process Communication

Process executing concurrently in the operating system may be either independent processes or cooperating processes

Independent processes – They cannot affect or be affected by the other processes executing in the system

Cooperating processes – They can affect or be affected by the other processes executing in the system

Any process that shares data with other processes is a cooperating process.

Reasons for providing an environment that allows process cooperation,

- Information sharing
- Computation speedup
- Modularity
- Convenience

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

Two fundamental models of inter-process communication,

1. **Shared Memory**
2. **Message passing**

In shared-memory model, a region of memory that s shared by the cooperating processes is established.

In the message passing model, communication takes place by means of message exchanged between the cooperating processes.

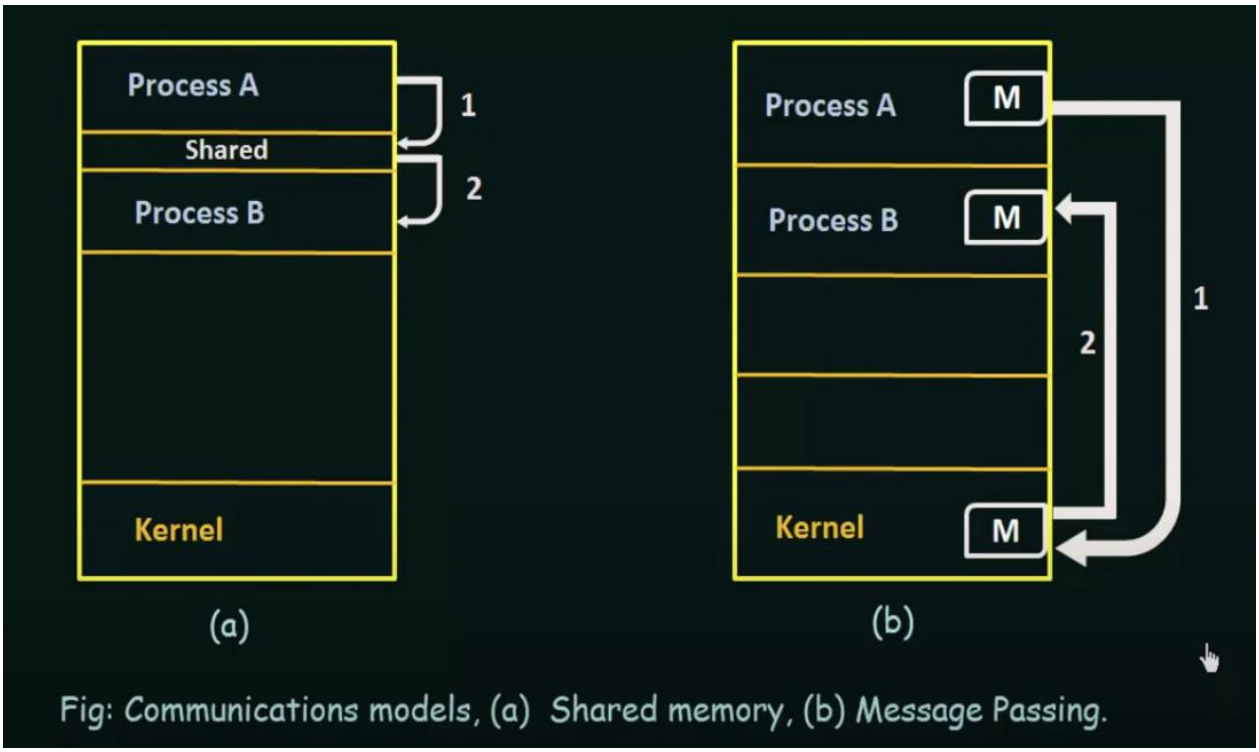


Fig: Communications models, (a) Shared memory, (b) Message Passing.

## Shared Memory Systems

Inter-process communication using shared memory requires communicating process to establish a region of shared memory.

Typically, a shared-memory region resides in the address space of the process creating the shared-memory segment.

Other processes that wish to communicate using this shared-memory must attach it to their address space.

Normally, the operating system tries to prevent on process from accessing another process's memory.

Shared memory requires that two or more processes agree to remove this restriction.

### Producer Consumer problem

A producer process produces information that is consumed by a consumer process

Ex; a compiler may produce assembly code, which is consumed by an assembler.

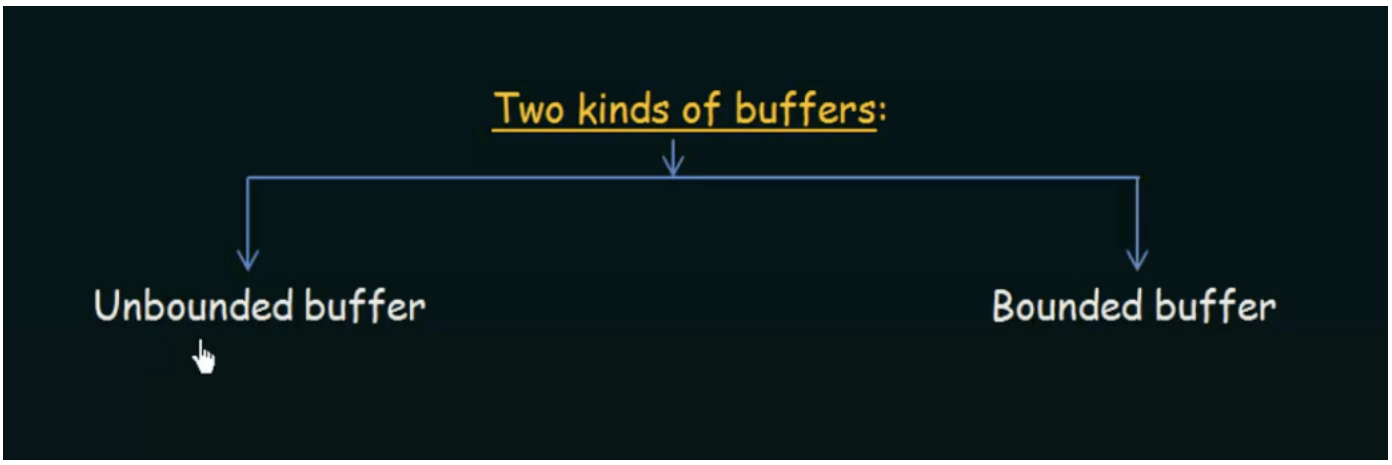
One solution to the producer-consumer problem is, **shared memory**

To allow producer and consumer process to run concurrently, we must have available **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 the 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.

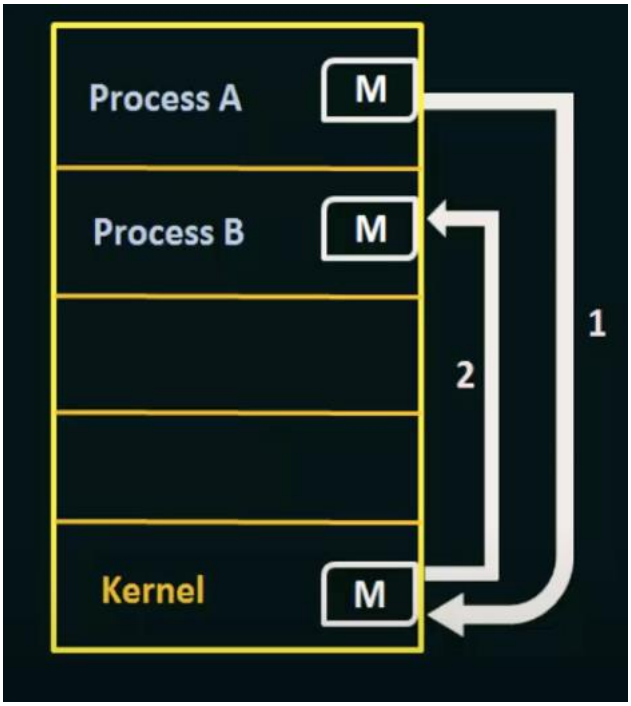


**Unbounded buffer** – places no practical limit on the size of the buffer. The consumer may have to wait for new items, but the producer can always produce new items.

**Bounded Buffer -** assumes a fixed buffer size. In this case, the consumer must wait if the buffer is empty, and producer must wait if the buffer is full.

## Message Passing Systems

Message passing provides a mechanism to allow processes to communicate and to synchronize their actions without sharing the same address space and is particularly useful in a distributed environment, where the communication processes may reside on different computers connected by a network.




A message-passing facility provides at least two operations,

- send(message) and
- receive(message)

message sent by a process can be of either fixed or variable size.

**Fixed size**: The system-level implementation is straightforward…but makes the task of programming more difficult.

Message passing provides a mechanism

**Variable size:** Requires a more complex system-level implementation…but programming task becomes simpler.

If processes **P** and **Q** want to communicate, they must send messages to and receive messages from each other.

A **communication link** must exist between them.

This link can be implemented in a variety of ways. There several methods for logically implementing a link and the send() / receive() operations, like.

- Direct or indirect communication (issue - naming)
- Synchronous or asynchronous communication (issue - synchronization)
- Automatic or explicit buffering (issue - buffering)

There are several issues related with features like,

- Naming
- Synchronization
- Buffering

**Naming**

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

Under direct communication – each process that wants to communicate must explicitly name the recipient or the sender of the communication.

- send (P, message) – send a message to process P
- receive (Q, message) –receive a message from process 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.

This scheme exhibits **symmetry in addressing;** that is, both the sender process and receiver process must name the other to communicate.

Another variant of Direct Communication, - here, only the sender names the recipient, the recipient is not required to name the sender.

- send (P, message) - send a message to process P
- receive (id, message) – receive a message from any process; *the variable id is set to the name of the process with which communication has taken place.*

This scheme employs **asymmetry in addressing.**

The **disadvantage** in both these schemes (**symmetric and asymmetric**) is the **limited modularity** of the resulting process definitions. Changing the identifier of a process may necessitate examining all other process definitions.

With indirect communication,

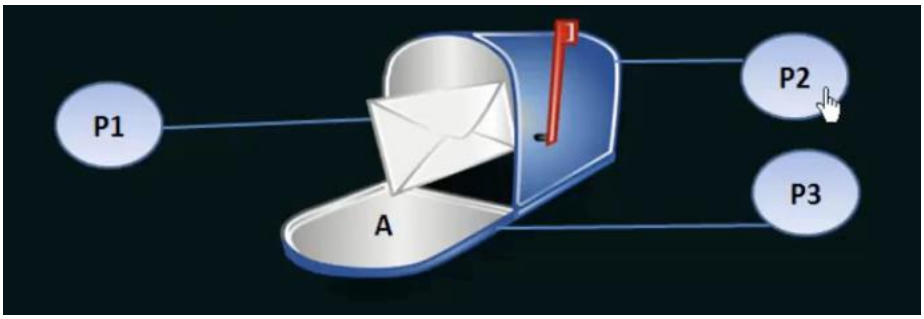
Messages are sent to and received from **mailboxes**, or ports.

- A mailbox can be viewed abstractly as an object into which messages can be placed by processes and from which messages can be removed.
- Each mailbox has a unique identification.
- Two processes can communicate only if the processes have a shared mailbox
  - send (A, message) – send a message to mailbox A
  - receive (A, message) – receive a message from mailbox A.

A communication link in this scheme has the following properties,

- A link is established between a pair of processes only if both members of the pair have a shared mailbox.
- 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 mailbox.

Now suppose that processes P1, P2 and P3 all share mailbox A




Processes P1 sends a message to A, while both P2 and P3 execute a receive() from A…which **process will receive the message sent by P1** ?

The answer depends on which of the following methods we choose,

- Allow a link to be associated with two processes at most
- Allow at most one process at a time to execute a receive () operation.
- Allow the system to select arbitrarily which process will receive the message (that either P2 or P3, but not both, will receive the message). The system also may define an algorithm for selecting which process will receive the message (that is, round robin where processes take turns receiving messages). The system may identify the receiver to the sender.

A **mailbox** may be **owned** either by a **process** or by the **operating system.**

**Synchronization**

Communication between processes takes place through calls to send() and receive() primitives. There are different design options for implementing each primitive.

Message passing may be either **blocking** or **nonblocking** – also known as **synchronous** and **asynchronous** respectively.

Blocking send: The sending process is blocked until the message is received by the receiving process or by the mailbox.

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 receive retrieves either a valid message or a null

**Buffering**

Whether communication is direct or indirect, messages exchanged by communicating processes reside in a temporary queue. Basically, such queues can be implemented in three ways,

**Zero capacity** – The queue has a 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 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 queue's length is potentially infinite; thus, any number of messages can wait in it. The sender never blocks.

## Sockets

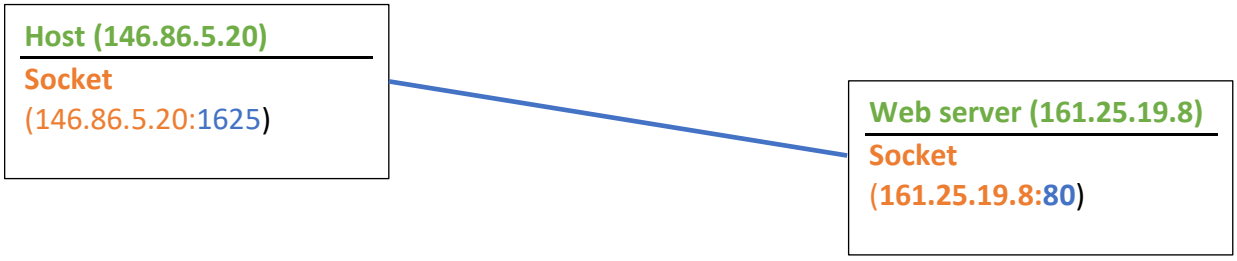
Used for communication in client-server systems

- A socket is defined as an endpoint for communication.
- A pair of processes communicating over a network employ a pair of sockets – one for each process.

- A socket is defined by an IP address concatenated with a port number.
- The server waits for incoming client requests by listening to a specified port. Once a request is received, the server accepts a connection from the client socket to complete the connection.

- Servers implementing specific serveces (such as telnet, ftp and http) listen to well-known ports…(a telnet server listens to port 23, an ftp server listens to port 21, and a web, or http, server listens to port 80).
- All ports below 1024 are considered well-known; we can use them to implement standard services.

Communicating using sockets




- When a client process initiates a request for a connection, it is assigned a port by the host computer.
- This port is some arbitrary number greater than 1024.
- The packets traveling between the host are delivered to the appropriate process based on the destination port number.

## Remote Procedure Calls (RPC)

Remote procedure call (RPC) is a protocol that one program can use to request a service from a program located in another computer on a network without having to understand the network's details.

- It is similar in many ways respect to the IPC mechanism.
- However, because we are dealing with an environment in which the processors are executing on separate systems, we must use a message-based communication scheme to provide remote service
- In contrasts to the IPC facility, the messages exchanged in RPC communication are well structured and are thus no longer just packets of data.
- Each message is addressed to an RPC daemon listening to a port on the remote system, and each contains an identifier on=f the function to execute and the parameters to pass to that function.
- The function is then executed as requested, and any output is sent back to the requester in a separate message.

Daemon – a program always listens / waits for an input

The sematic of RPCs allow a client to invoke a procedure on a remote host as it would invoke a procedure locally,

- The RPC system hides the details that allow communication to take place by providing a stub on the client side.
- Typically, a separate stub exists for each separate remote procedure.
- When the client invokes a remote procedure, the RPC system calls the appropriate stub, passing it the parameters provided to the remote procedure, this stub locates the port on the server and marshals (set) the parameters.
- Parameter marshalling invokes packaging the parameters into a form that can be transmitted over a network.
- The stub then transmits a message to the server using message passing.
- A similar stub on the server side receives this message and invokes the procedure on the server.
- If necessary, return values are passed back to the client using the same technique.

Issues in RPC and how they are resolved

| Issues | How they are resolves |
|---|---|
| Difference in data representation on the client and server machines.<br><br>E.g., Representation of 32-bit integers<br><br>Some systems (known as big-endian) use the high memory address to store the most significant byte, while other systems (known as little-endian) store the least significant byte at the high memory address. | RPC systems define a machine-independent representation of the data. One such representation is known as external data representation (XDR).<br><br>On the client side, parameter marshaling involves converting the machine dependent data into XDR before they are sent to the server.<br><br>On the server side, the XDR data are unmarshalled and converted to the machine-dependent representation for the server. |

| | |
|---|---|
| Whereas the local procedure calls fail only under extreme circumstances,<br><br>RPCs can fail, or be duplicated and executed more than once, as a result of common network errors. | The operating system must ensure that messages are acted on exactly once, rather than at most once. Most local procedure calls have the “exactly once” functionality, but it is more difficult to implement |