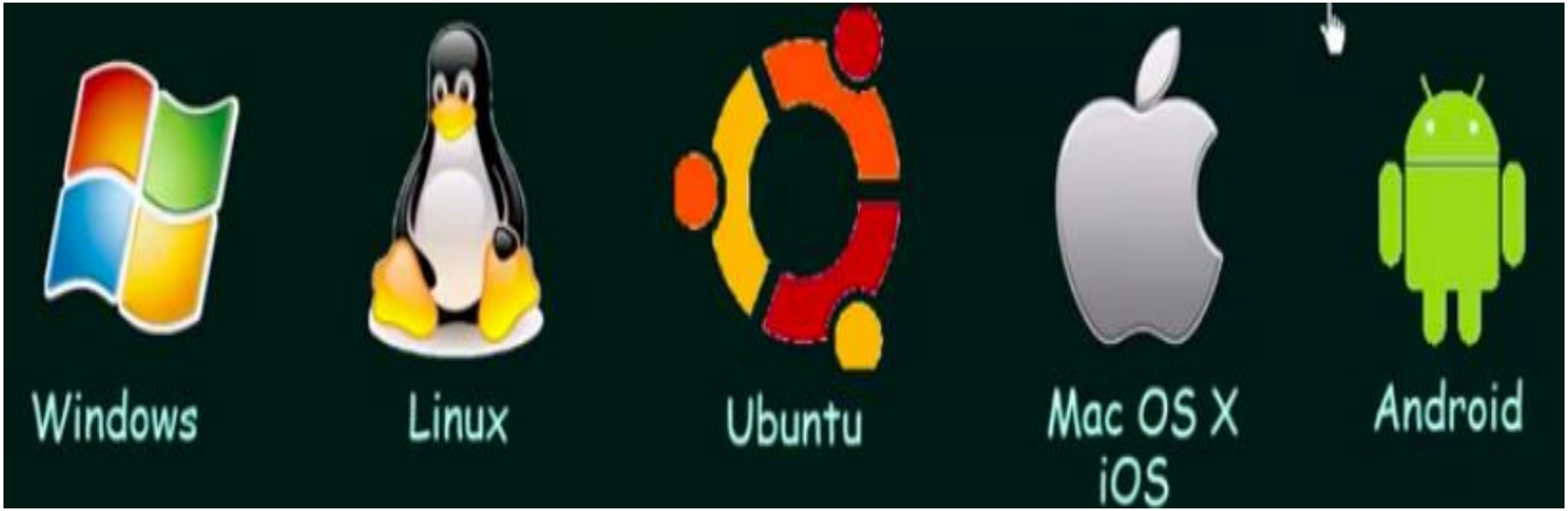
UNIT 1- Operating Systems Overview

J.Premalatha
Professor/IT
Kongu Engineering College
Perundurai

Introduction –Chapter1

- What Operating Systems Do
- Computer-System Organization
- Computer-System Architecture
- Operating-System Operations
- Resource Management
- Security and Protection
- Virtualization
- Computing Environments

Various Operating systems



→ An Operating System (OS) is a program that manages the computer hardware.

→ It also provides a basis for Application Programs and acts as an intermediary between computer User and compter Hardware.

What is an Operating System?

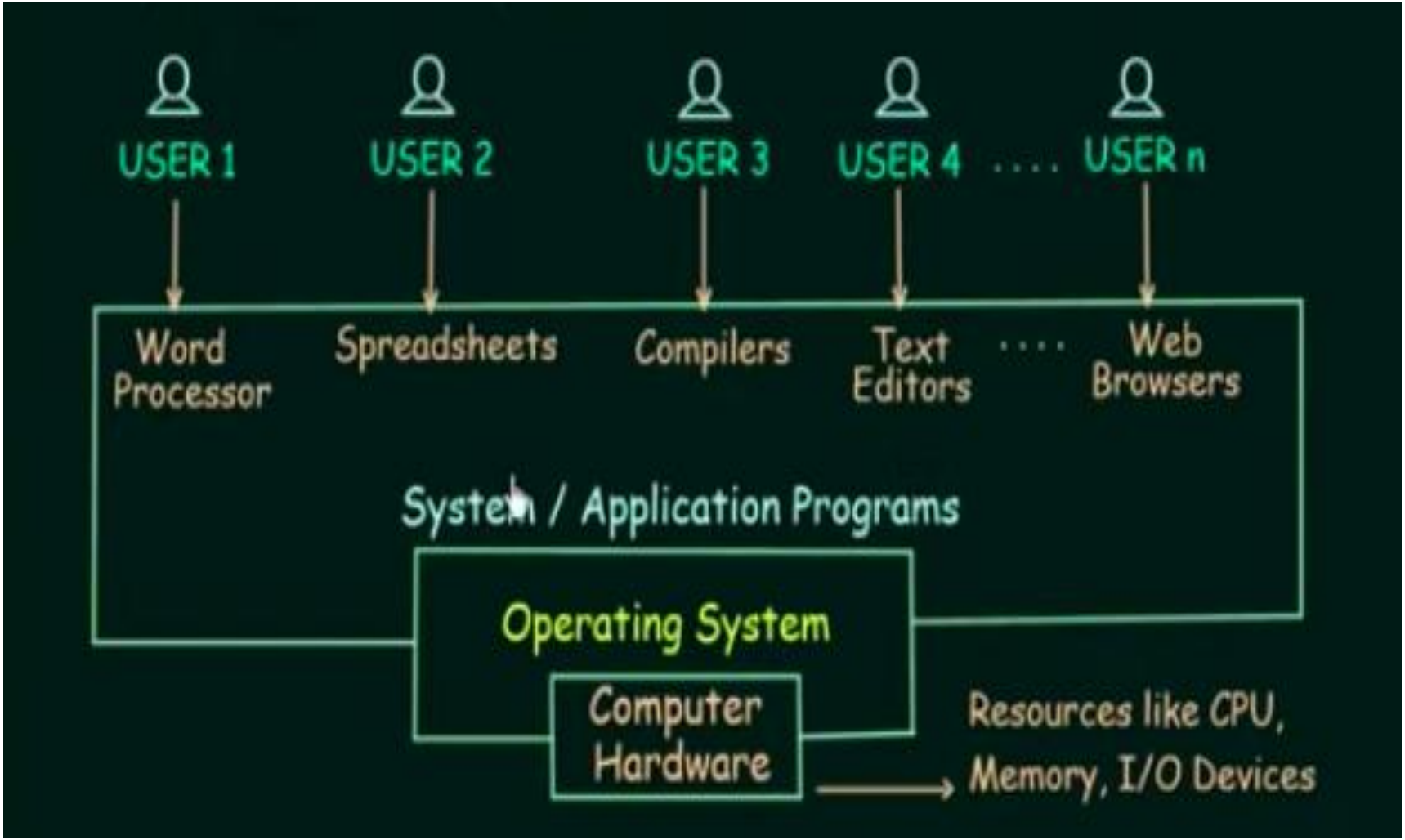
- A program that acts as an intermediary between a computer user and the computer hardware
- **Operating system goals:**
  - Execute user programs and make solving user problems easier
  - Make the computer system convenient to use
  - Use the computer hardware in an efficient manner
  - Some of OS are designed to **Convenient** and others to be **efficient**
  - Some OS provides **Both**

Convenient – It is the interface between user and hardware

Efficient - Allocation of resources

Both – Management of resources and security etc

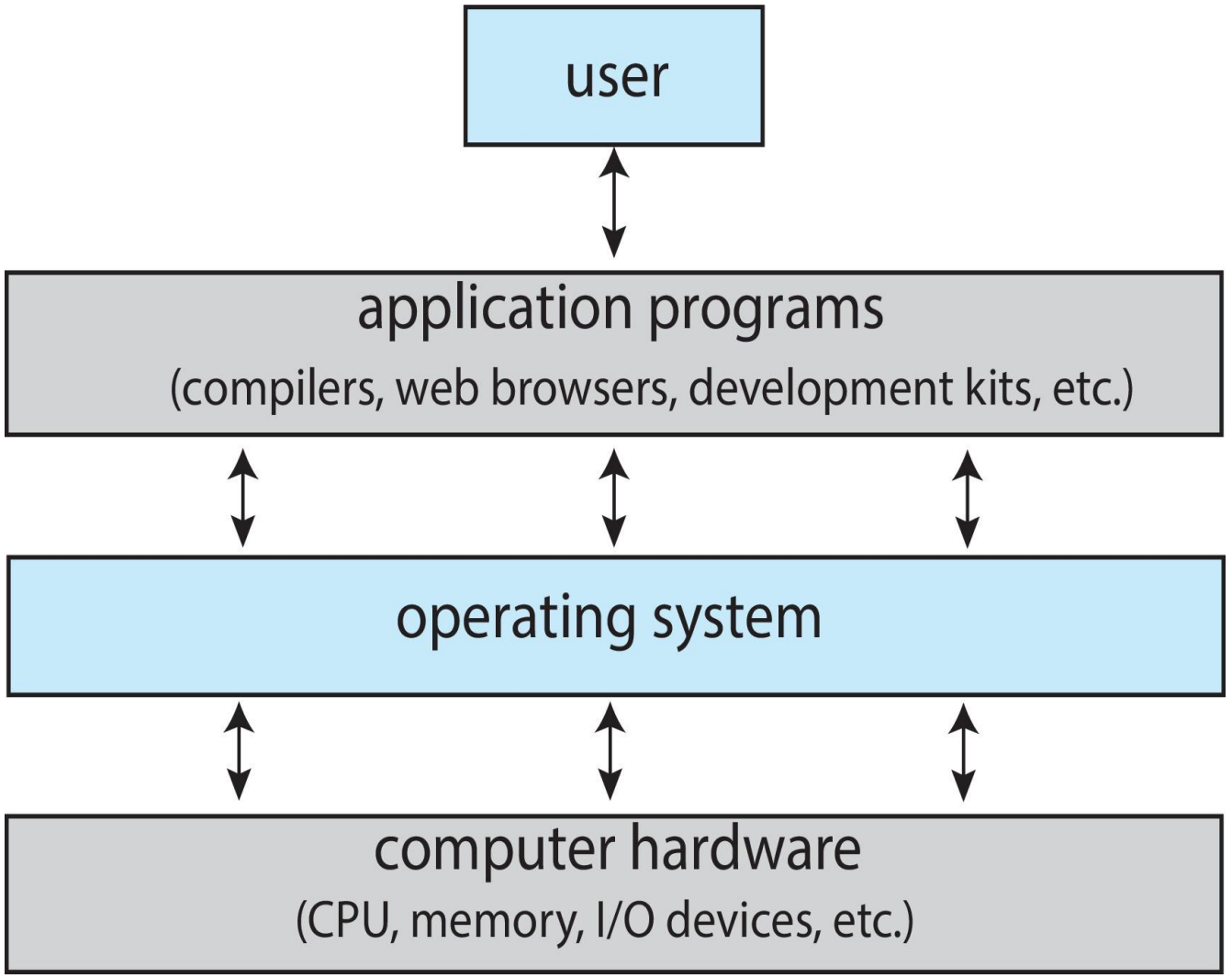
What is an Operating System?



Computer System Structure

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

Abstract View of Components of Computer



What Operating Systems Do

- Depends on the point of view 1. User view 2. System view
- Users want convenience, **ease of use** and **good performance-** Don't care about **resource utilization [User view]**
- Operating system is a **resource allocator** and **control program** making efficient use of Hardware and managing execution of user programs **[System view]**
- But shared computer such as **mainframe** or **minicomputer** must keep all users happy
- Users of dedicate systems such as **workstations** have dedicated resources but frequently use shared resources from **servers**
- Mobile devices like smart phones and tablets are resource poor, optimized for usability and battery life
  - Mobile user interfaces such as touch screens, voice recognition
- Some computers have little or no user interface, such as embedded computers in devices and automobiles -Run primarily without user intervention

- Term OS covers many roles
  - Because of myriad designs and uses of OSes
  - Present in toasters through ships, spacecraft, game machines, TVs and industrial control systems
  - Born when fixed use computers for military became more general purpose and needed resource management and program control

Different types of OS

- **Batch OS**
- **Time sharing OS**
- **Network OS**
- **Distributed OS**
- **Real time OS**
- **Multiprogramming/ processing / tasking**

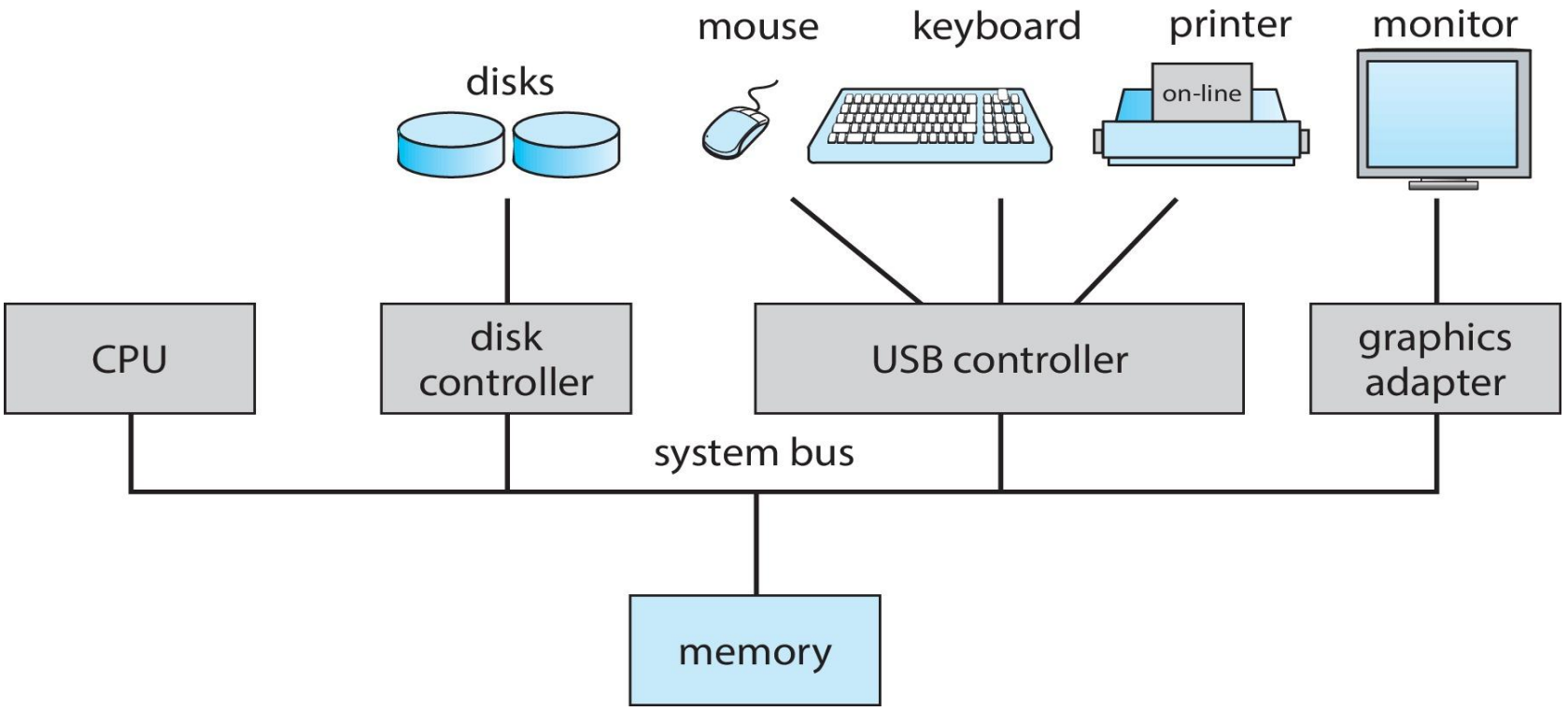
Operating System Definition

- No universally accepted definition
- "Everything a vendor ships when you order an operating system" is a good approximation - But varies wildly
- A more common definition is , that the operating system is **the one program running at all times on the computer—usually called the kernel. Along with the kernel, there are two other types of programs: system programs, which are associated with the operating system but are not necessarily part of the kernel, and application programs, which include all programs not associated with the operation of the system.**
- Today's OSes for general purpose and mobile computing also include **middleware** – **a set of software frameworks that provide additional services to application developers such as databases, multimedia, graphics**

Overview of Computer System Structure

Computer System Organization

- Computer-system operation
  - One or more CPUs, device controllers connect through common **bus** providing access to shared memory
  - Concurrent execution of CPUs and devices competing for memory cycles



Computer-System Operation

- I/O devices and the CPU can execute concurrently
- Each device controller is in charge of a particular device type
- Each device controller has a local buffer
- Each device controller type has an operating system **device driver** to manage it
- CPU moves data from/to main memory to/from local buffers
- I/O is from the device to local buffer of controller
- Device controller informs CPU that it has finished its operation by causing an **interrupt**

Computer Startup - Bootstrap

1) Bootstrap Program:→The initial program that runs when a computer is powered up or rebooted.

→It is stored in the ROM. **or EPROM, known as firmware**

→It must know how to load the OS and start executing that system.

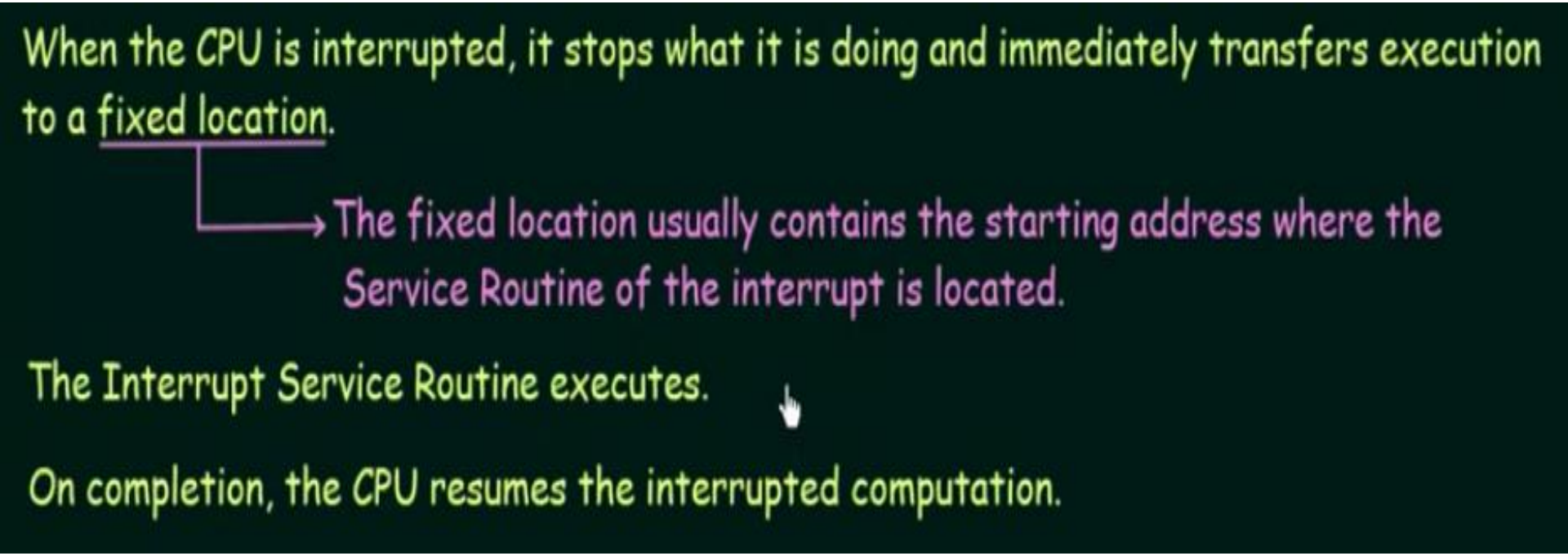
→It must locate and load into memory the OS Kernel.

2) Interrupt: → The occurence of an event is usually signalled by an 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.

3) System Call (Monitor call): → Software may trigger an interrupt by executing a special operation called System Call.

Common Functions of Interrupts

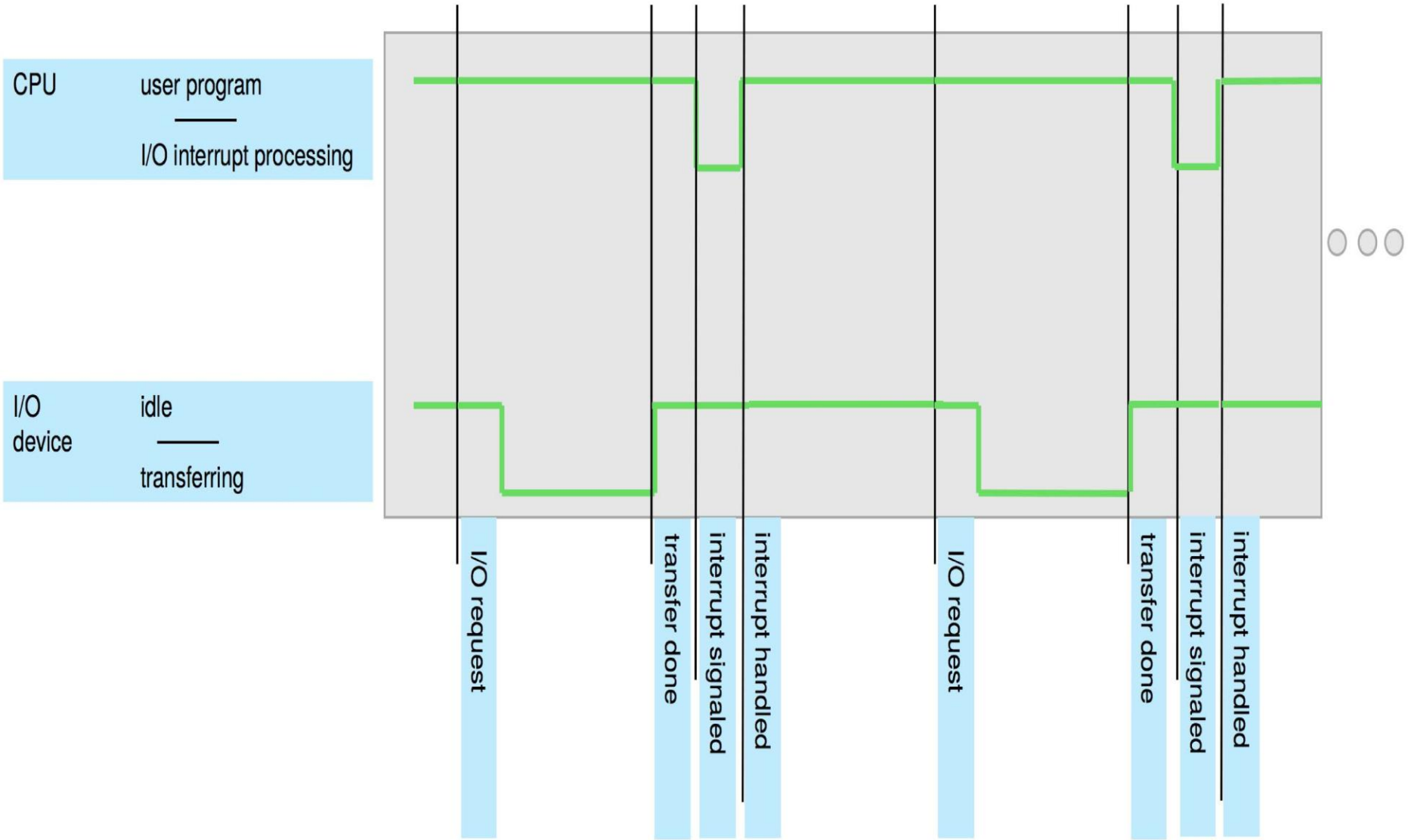


- Interrupt transfers control to the interrupt service routine generally, through the **interrupt vector**, which contains the addresses of all the service routines
- Interrupt architecture must save the address of the interrupted instruction
- A **trap** or **exception** is a software-generated interrupt caused either by an error or a user request
- An operating system is **interrupt driven**

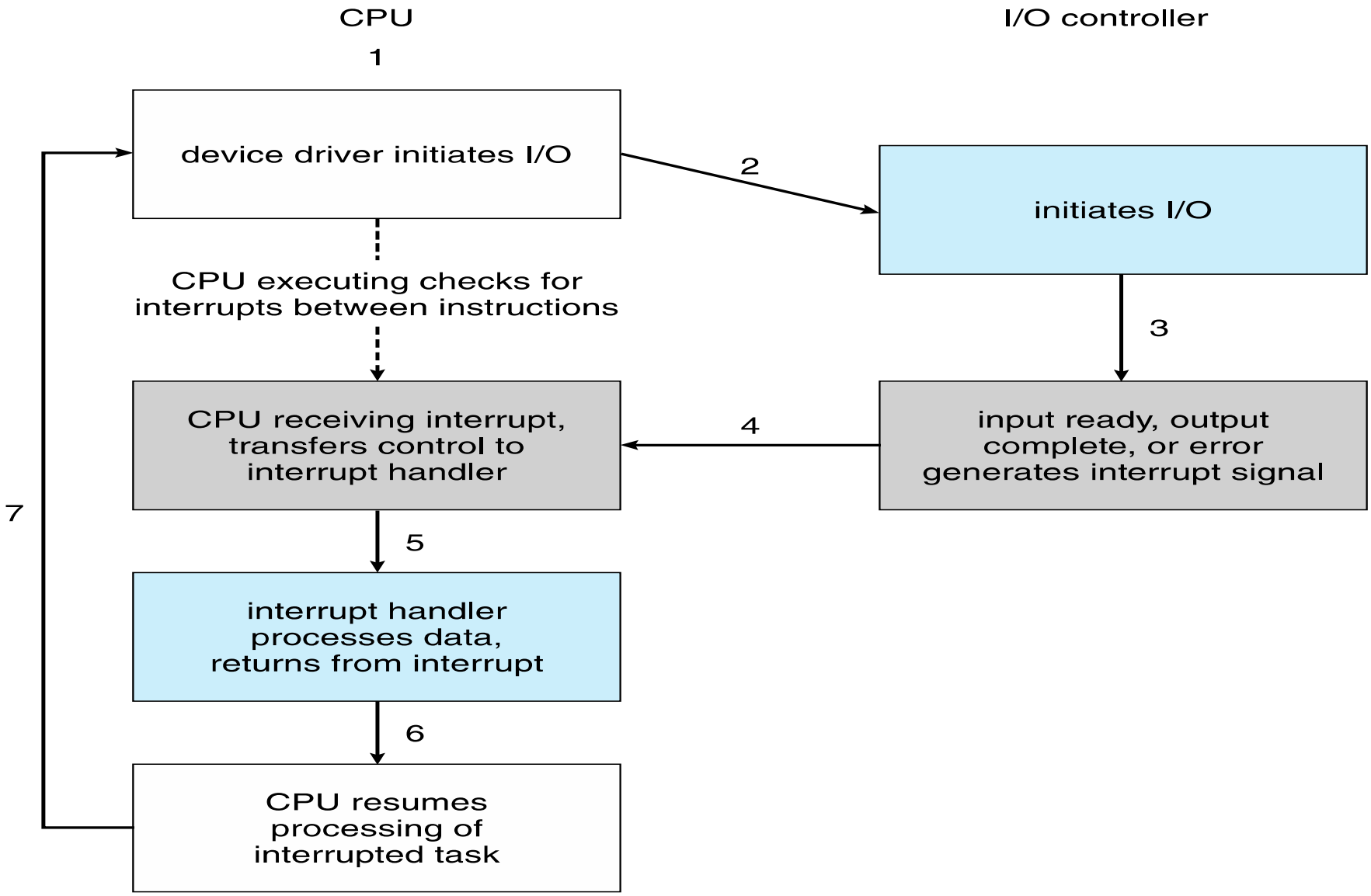
1. Interrupt Handling

- The operating system preserves the state of the CPU by storing the registers and the program counter
- Determines which type of interrupt has occurred:
- Separate segments of code determine what action should be taken for each type of interrupt

Interrupt Timeline



Interrupt-drive I/O Cycle



I/O Structure (Cont.)

- After I/O starts, control returns to user program only upon I/O completion
  - Wait instruction idles the CPU until the next interrupt
  - Wait loop (contention for memory access)
  - At most one I/O request is outstanding at a time, no simultaneous I/O processing
- After I/O starts, control returns to user program without waiting for I/O completion
  - **System call** – request to the OS to allow user to wait for I/O completion
  - **Device-status table** contains entry for each I/O device indicating its type, address, and state
  - OS indexes into I/O device table to determine device status and to modify table entry to include interrupt

Interrupt Implementation

- The CPU hardware has a wire called the **interrupt-request line that the CPU senses after executing every** instruction.
- When the CPU detects that a controller has asserted a signal on the interrupt-request line, it reads the **interrupt number and jumps to the interrupt-handler routine by using that interrupt number as an index into the interrupt vector.**
- It then starts execution at the address associated with that index.
- The interrupt handler saves any state it will be changing during its operation, determines the cause of the interrupt, performs the necessary processing, performs a state restore, and executes a return from interrupt instruction to return the CPU to the execution state prior to the interrupt.
- Most **CPUs have two interrupt request lines**. One is the **nonmaskable interrupt, which is reserved for events**. The second interrupt line is **maskable: it can be turned off by the CPU before the execution of critical instruction sequences that must not be interrupted.**

Nonmaskable & maskable interrupt

- Non-Maskable Interrupt (NMI) is a type of hardware interrupt (or signal to the processor) that prioritizes process. Unlike other types of interrupts, the non-maskable interrupt cannot be ignored through the use of interrupt masking techniques.

[Ex] user presses **control, alt, delete** to create an immediate signal to the system when the computer is not responding.

- Maskable Interrupt : An Interrupt that can be disabled or ignored by the instructions of CPU are called as Maskable Interrupt. When an interrupt of this type occurs, the system can handle it after it executes the current instructions.

Design of the interrupt vector for Intel processors

- The events from 0 to 31, which are nonmaskable, are used to signal various error conditions.
- The events from 32 to 255, which are maskable, are used for purposes such as device-generated interrupts.

vector number	description
0	divide error
1	debug exception
2	null interrupt
3	breakpoint
4	INTO-detected overflow
5	bound range exception
6	invalid opcode
7	device not available
8	double fault
9	coprocessor segment overrun (reserved)
10	invalid task state segment
11	segment not present
12	stack fault
13	general protection
14	page fault
15	(Intel reserved, do not use)
16	floating-point error
17	alignment check
18	machine check
19–31	(Intel reserved, do not use)
32–255	maskable interrupts

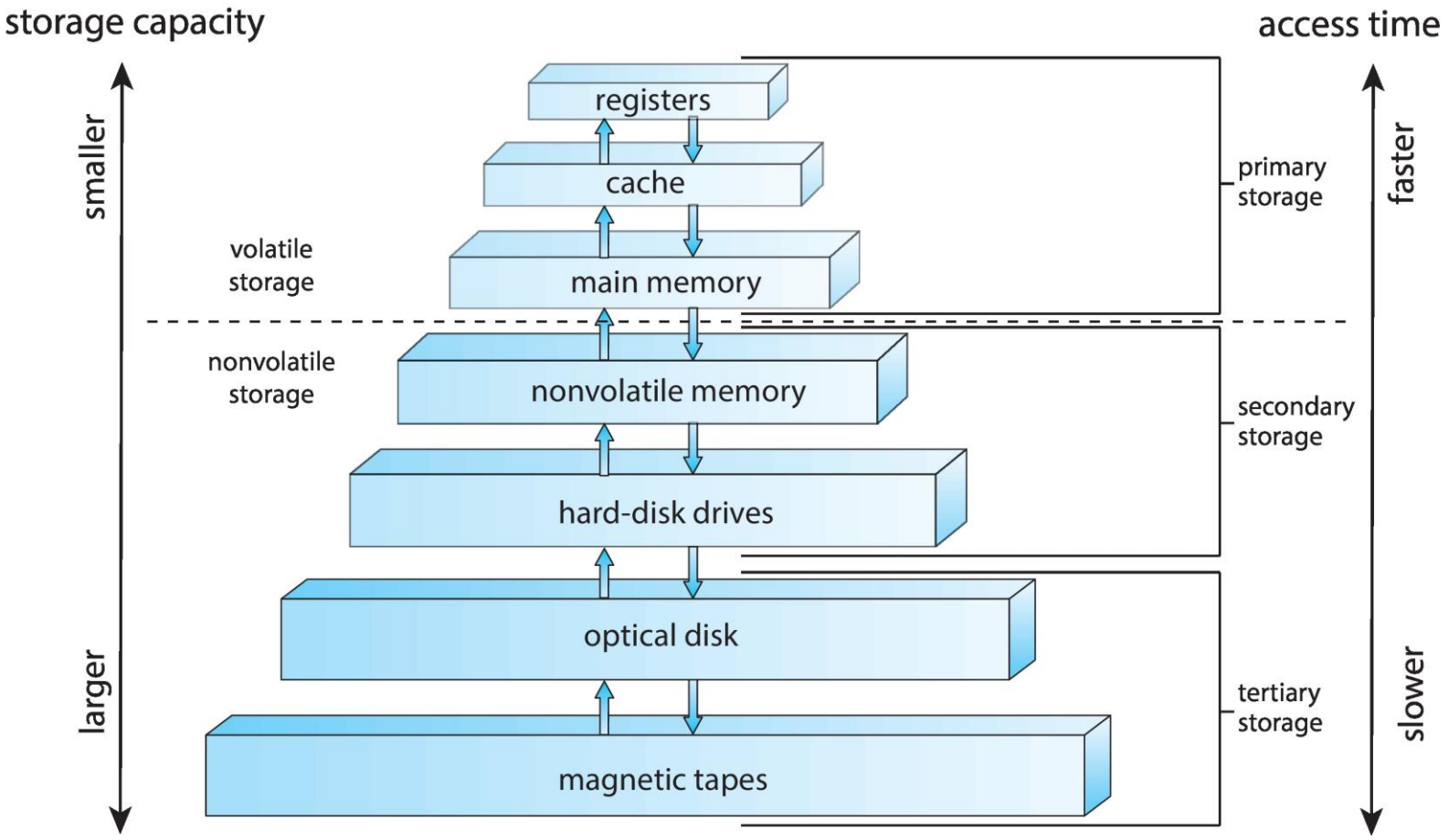
Storage Structure

- Main memory – only large storage media that the CPU can access directly
  - **Random access** Typically **volatile**
  - Typically **random-access memory** in the form of **Dynamic Random-access Memory (DRAM)**
- Secondary storage – extension of main memory that provides large **nonvolatile** storage capacity
- **Hard Disk Drives** (**HDD**) – rigid metal or glass platters covered with magnetic recording material
  - Disk surface is logically divided into **tracks**, which are subdivided into **sectors**
  - The **disk controller** determines the logical interaction between the device and the computer
- **Non-volatile memory (NVM)** devices– faster than hard disks, nonvolatile

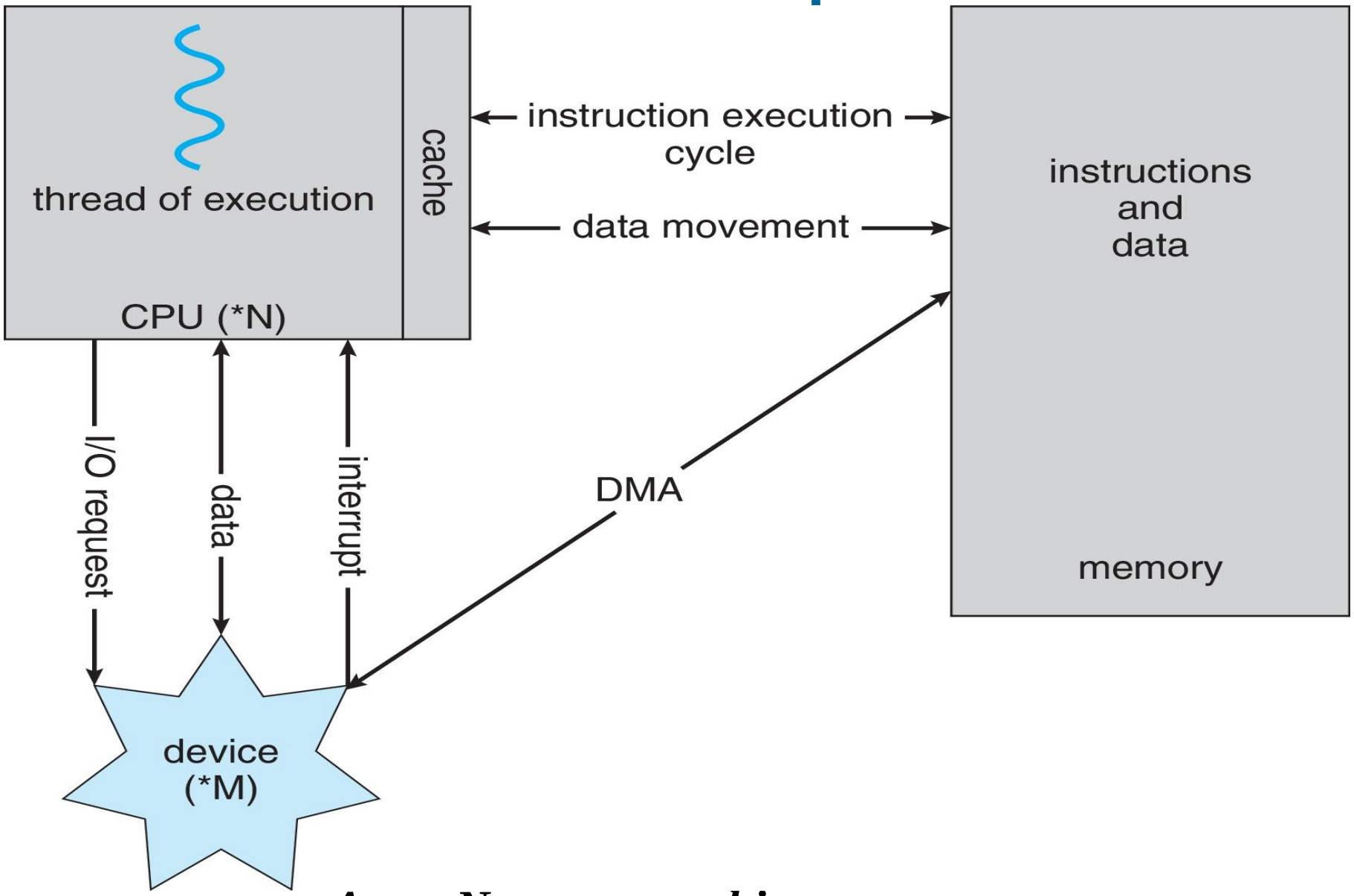
Storage Hierarchy

- Storage systems organized in hierarchy
  - Speed
  - Cost
  - Volatility
- **Caching** – copying information into faster storage system; main memory can be viewed as a cache for secondary storage
- **Device Driver** for each device controller to manage I/O
  - Provides uniform interface between controller and kernel

Storage-Device Hierarchy



How a Modern Computer Works



A von Neumann architecture

-> To start an I/O operation, the device driver loads the appropriate registers within the device controller

-> The device controller, in turn, examines the contents of these registers to determine what action to take

-> The controller starts the transfer of data from the device to its local buffer

-> Once the transfer of data is complete, the device controller informs the device driver via an interrupt that it has finished its operation

-> The device driver then returns control to the operating system

This form of interrupt-driven I/O is fine for moving small amounts of data but can produce high overhead when used for bulk data movement

To solve this problem Direct Memory Access (DMA) is used

- Used for high-speed I/O devices able to transmit information at close to memory speeds
- Device controller transfers blocks of data from buffer storage directly to main memory without CPU intervention
- Only one interrupt is generated per block, rather than the one interrupt per byte

Computer System Architecture

Computer-System Architecture

- Single Processor System
- Multi Processor System
- Clustered Systems

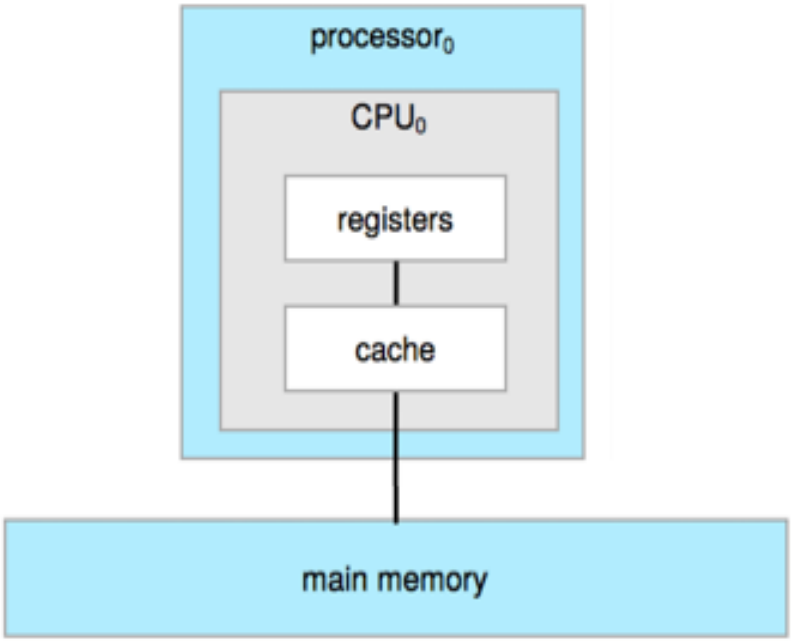
DEFINITIONS OF COMPUTER SYSTEM COMPONENTS

- 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.

Computer-System Architecture

Single Processor System

- Many years ago, most computer systems used a single processor containing one CPU with a single processing core. The **core is the component that executes instructions and registers for storing data locally.**
- The one main CPU with its core is capable of executing a general-purpose instruction set, including instructions from processes.



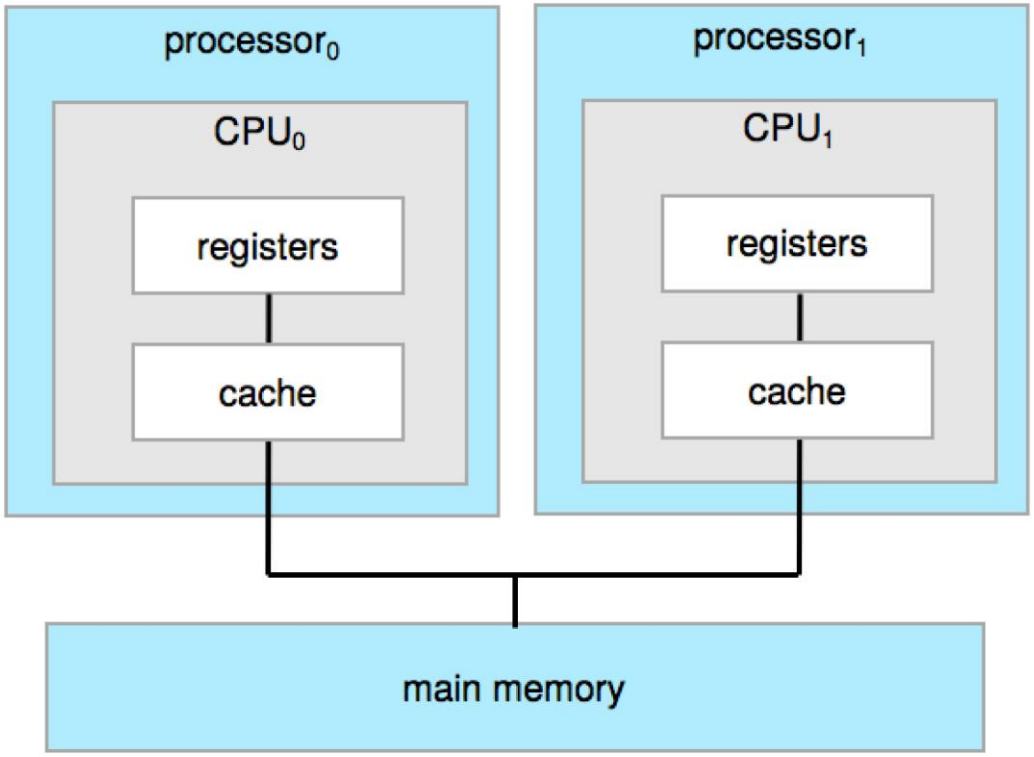
- These systems have other special-purpose processors. They may come in the form of device-specific processors, such as disk, keyboard, and graphics controllers.

All of these special-purpose processors run a limited instruction set and do not run processes. Sometimes, they are managed by the operating system

Computer-System Architecture

- **Multiprocessor systems:** On modern computers, from mobile devices to servers is available. Traditionally, such systems have two (or more) processors, each with a single-core CPU. The processors share the computer bus and sometimes the clock, memory, and peripheral devices. Also known as **parallel systems**, **tightly-coupled systems**
  - Advantages include:
    1. **Increased throughput** - Speedup ratio: N processor is not N, However less than N
    2. **Economy of scale** – cost is less than equivalent multiple single processor systems (Because they can share their peripherals)
    3. **Increased reliability** – graceful degradation or fault tolerance
  - Two types:
    1. **Asymmetric Multiprocessing** – each processor is assigned a specific task.
    2. **Symmetric Multiprocessing** – each processor performs all tasks

Symmetric Multiprocessing Architecture(SMP)

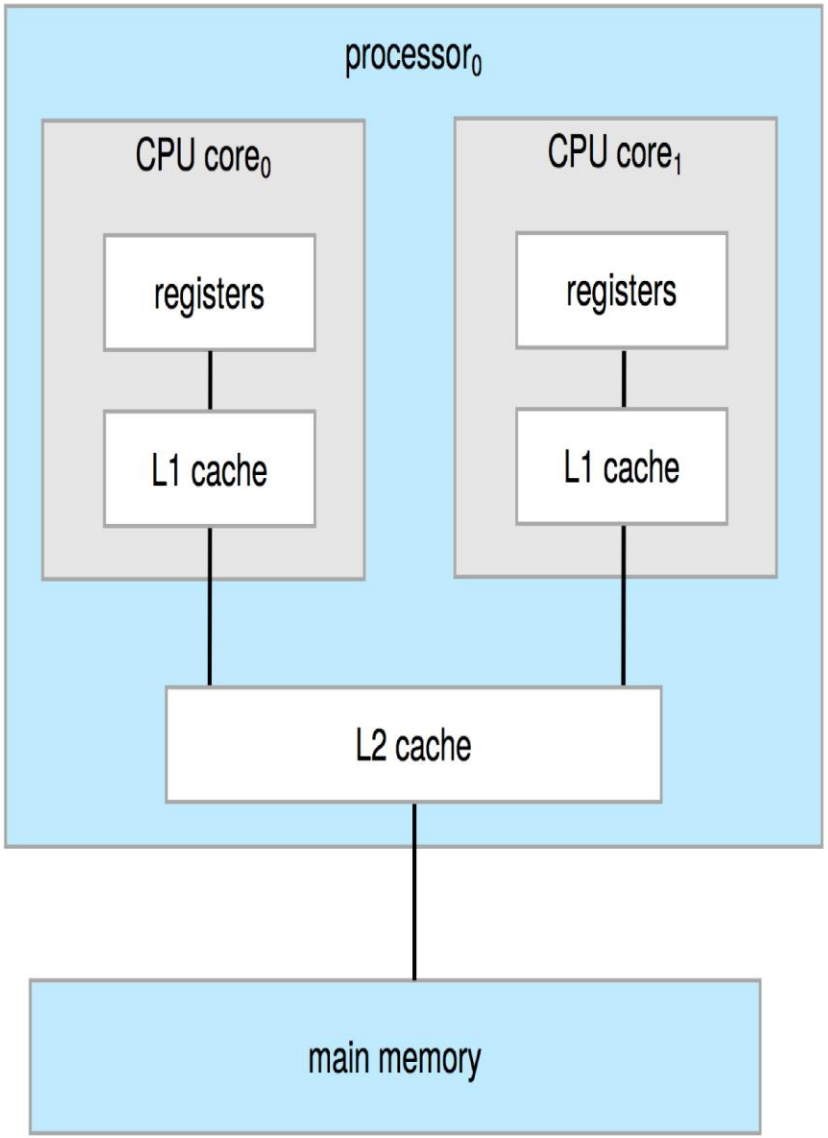


•Each CPU processor performs all tasks, including operating-system functions and user processes. Each CPU processor has its own set of registers and local cache.

•However, all processors share physical memory over the system bus.

•The benefit of this model is that many processes can run simultaneously ***N processes can run if there are N CPUs.***

Dual-Core Design

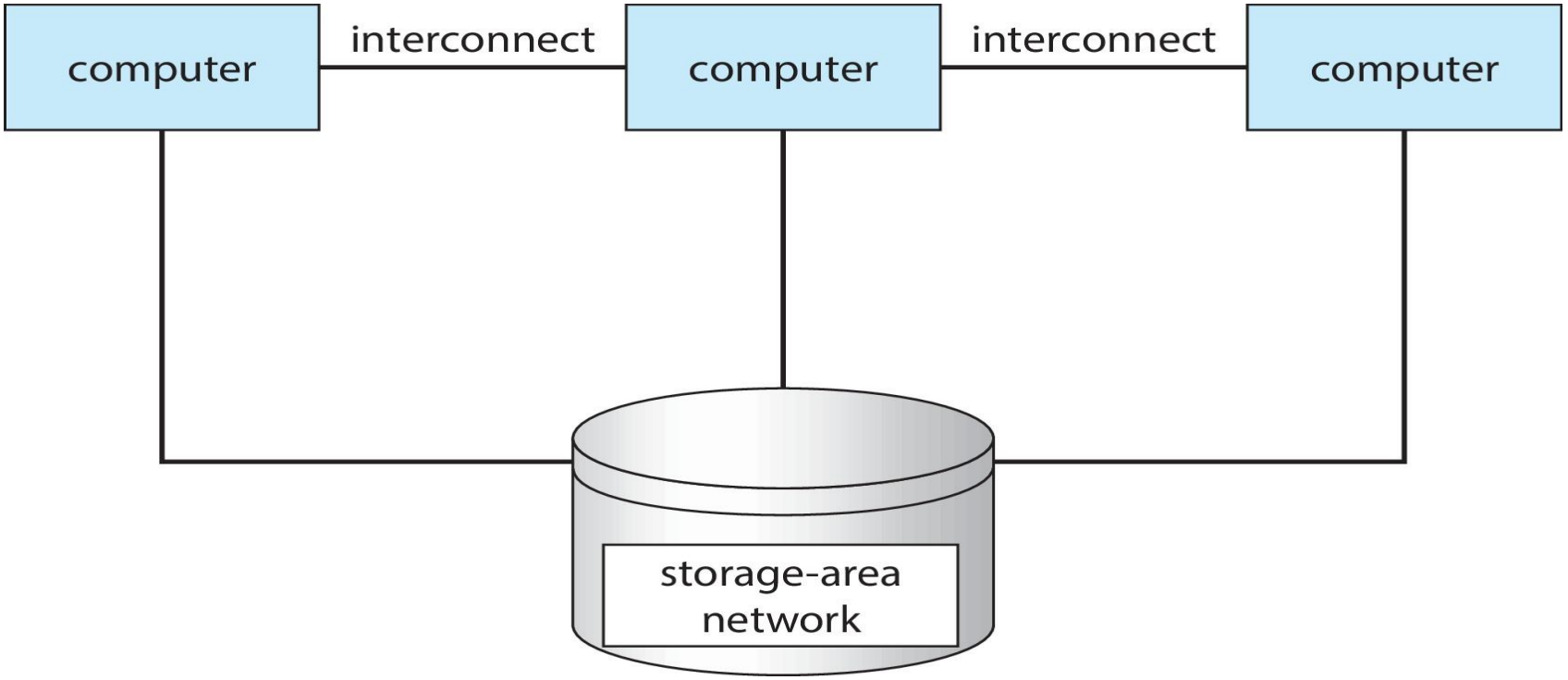


- Each core has its own register set, as well as its own local cache, often known as a level 1, or L1, cache.
- Level 2 (L2) cache is local to the chip but is shared by the two processing cores.
- Most architectures adopt this approach, combining local and shared caches, where local, lower-level caches are generally smaller and faster than higher-level shared caches.

Clustered Systems

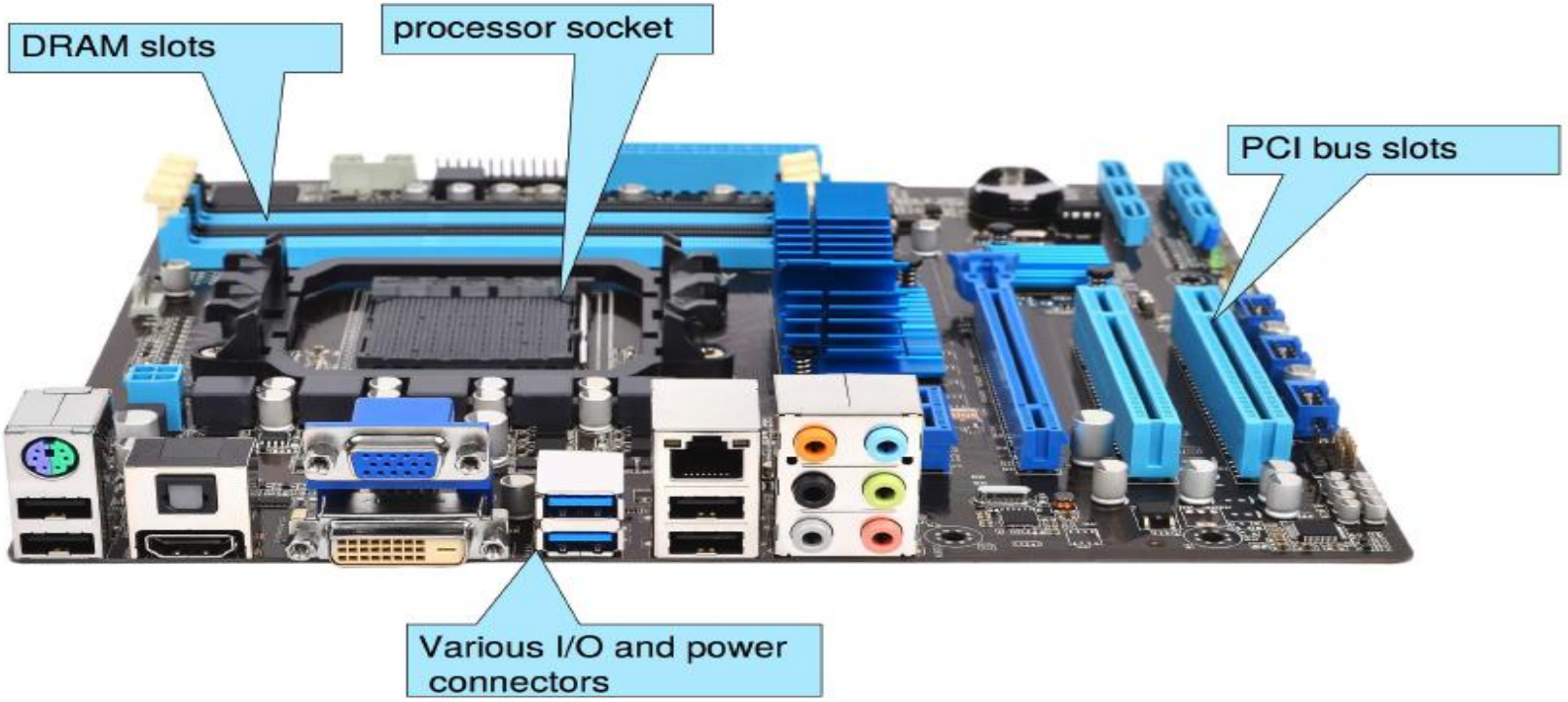
- Like multiprocessor systems, but multiple systems working together
  - Usually sharing storage via a **storage-area network (SAN)**
  - Provides a **high-availability** service which survives failures
    - **Asymmetric clustering** has one machine in **hot-standby** mode [Monitoring other machines]
    - **Symmetric clustering** has multiple nodes running applications, monitoring each other
  - Some clusters are for **high-performance computing (HPC)**
    - Applications must be written to use **parallelization**
  - Some have **distributed lock manager** (**DLM**) to avoid conflicting operations

Clustered Systems



PC Motherboard

Consider the desktop PC motherboard with a processor socket shown below:



This board is a fully-functioning computer, once its slots are populated. It consists of a processor socket containing a CPU, DRAM sockets, PCIe bus slots, and I/O connectors of various types. Even the lowest-cost general-purpose CPU contains multiple cores. Some motherboards contain multiple processor sockets. More advanced computers allow more than one system board, creating NUMA systems.

1.Batch OS : Sequence of jobs in a program on a computer **without manual** interventions. There is an operator which takes similar jobs having the same requirement and group them into batches. It is the responsibility of the operator to sort jobs with similar needs. **There is no interaction between user and CPU**

Examples of Batch based Operating System: Payroll System, Bank Statements, etc

2.Time-Sharing Operating Systems – Each task is given some time to execute so that all the tasks work smoothly. Each user gets the time of CPU as they use a single system. **These systems are also known as Multitasking Systems.** The task can be from a single user or different users also. The time that each task gets to execute is called **quantum.** After this time interval is over OS switches over to the next task.

Examples of Time-Sharing OSs are: Multics, Unix, etc.

3.Distributed Operating System – These types of the operating system is a recent advancement in the world of computer technology and are being widely accepted all over the world. Various autonomous interconnected computers communicate with each other using a shared communication network. Independent systems possess their own memory unit and CPU.

Examples of Distributed Operating System are- MS-DOS, LOCUS

4. Network Operating System – These systems run on a server and provide the capability to manage data, users, groups, security, applications, and other networking functions. These types of operating systems allow shared access of files, printers, security, applications, and other networking functions over a small private network.

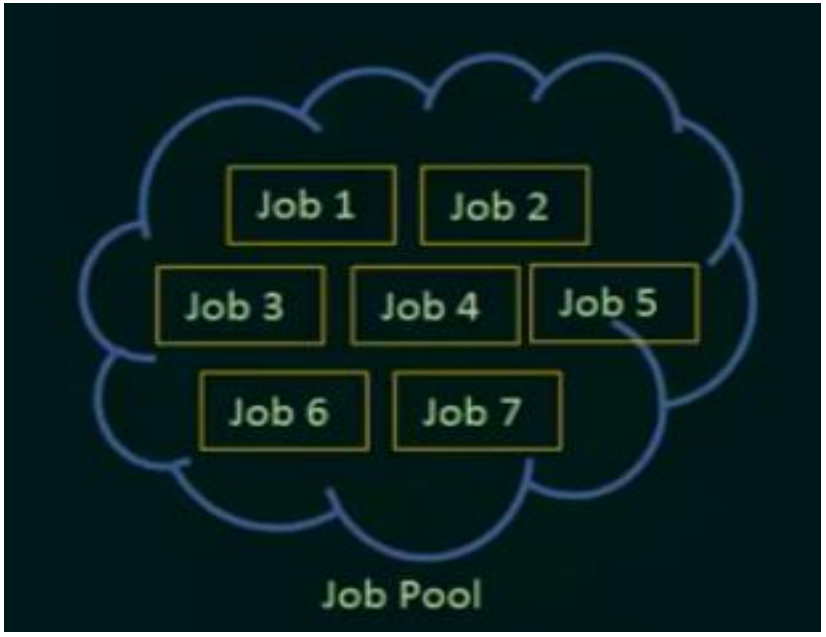
Examples of Network Operating System are: Microsoft Windows Server 2003, Microsoft Windows Server 2008, UNIX, Linux, Mac OS X, Novell NetWare, and BSD, etc.

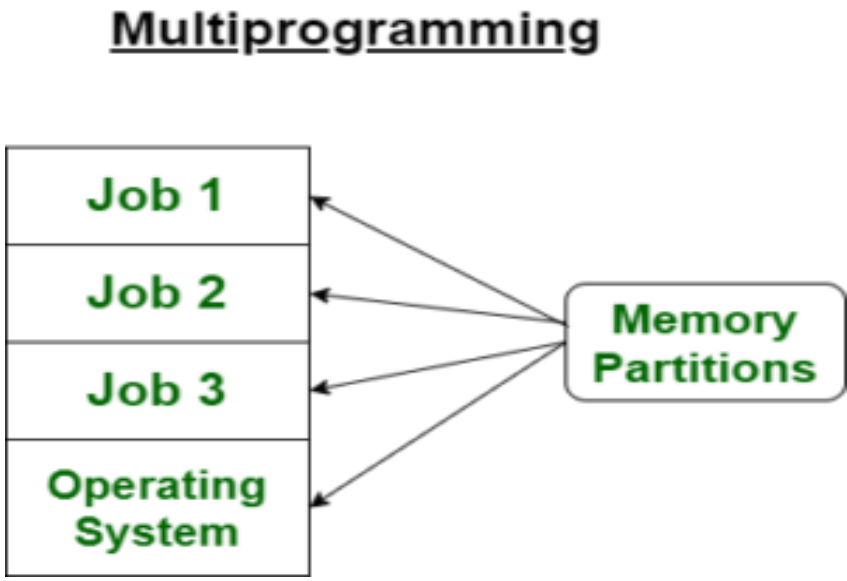
5. Real-Time Operating System – These types of OSs serve real-time systems. **The time interval required to process and respond to inputs is very small.** This time interval is called **response time**. **Real-time systems** are used when there are time requirements that are very strict like **missile systems, air traffic control systems, robots,**

Examples of Real-Time Operating Systems are: Scientific experiments, medical imaging systems, industrial control systems, weapon systems, robots, air traffic control systems, etc.

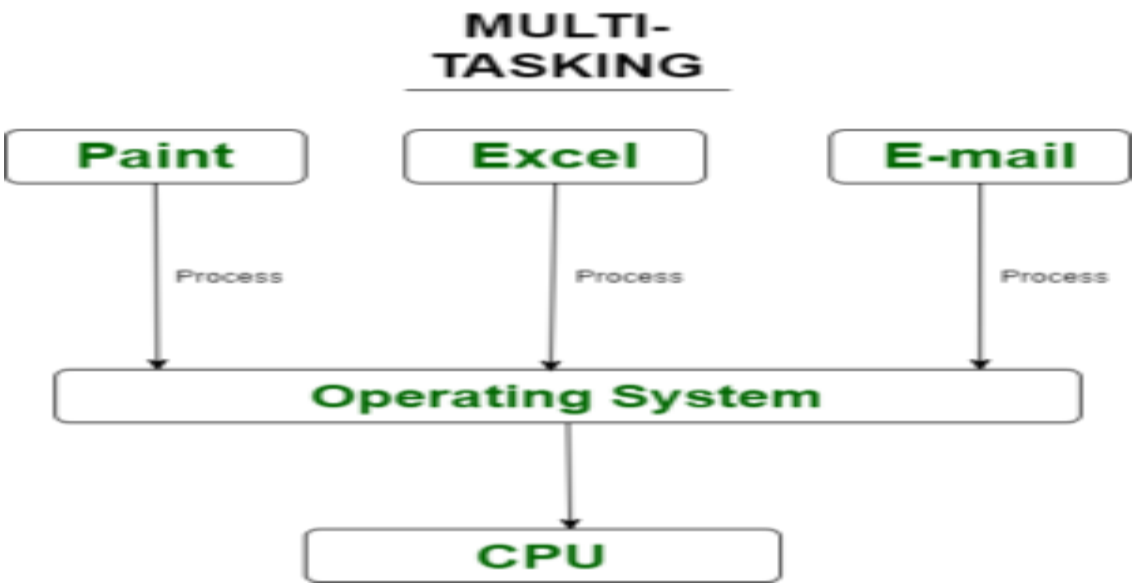
6. Multiprogramming/ tasking / processing : CPU is a super fast device and keeping it occupied for a single task is never a good idea. Considering the huge differences between CPU speed and IO speed, many concepts like multiprogramming, multitasking, multithreading, etc have been introduced to make better CPU utilization.

Multi programming:- Multi-programming increases CPU utilisation by organising jobs (code and data) so that the CPU always has one to execute. The idea is to keep multiple jobs in main memory. If one job gets occupied with IO, CPU can be assigned to other job.





Multi-tasking:- It is a logical extension of multiprogramming. Multitasking is the ability of an OS to execute more than one **task** simultaneously on *a CPU machine*. These multiple tasks share common resources (like CPU and memory). In multi-tasking systems, the CPU executes multiple jobs by switching among them typically using a small time **quantum**, and the switches occur so quickly that the users feel like interact with each executing task at the same time.



Difference between Multiprogramming and Multi-tasking

S.no	Multiprogramming	Multi-tasking
1.	Both of these concepts are for single CPU.	Both of these concepts are for single CPU.
2.	Concept of Context Switching is used.	Concept of Context Switching and Time Sharing is used.
3.	In multiprogrammed system, the operating system simply switches to, and executes, another job when current job needs to wait.	The processor is typically used in time sharing mode. Switching happens when either allowed time expires or where there other reason for current process needs to wait (example process needs to do IO).
4.	Multi-programming increases CPU utilization by organising jobs .	In multi-tasking also increases CPU utilization, it also increases responsiveness.
5.	The idea is to reduce the CPU idle time for as long as possible.	The idea is to further extend the CPU Utilization concept by increasing responsiveness Time Sharing.

Operating-System Operations

- A computer to start running—for instance, when it is powered up or rebooted—it needs to have an initial program to run. This initial program is **bootstrap program** and it is stored within the computer hardware(ROM) or EPROM as firmware.
- Bootstrap program – simple code to initialize the system, load the kernel
- Once the kernel is loaded and executing, it can start providing services to the system and its users. **Some services are provided outside of the kernel by system programs that are loaded into memory at boot time to become system daemons, which run the entire time the kernel is running.**
- On Linux, **the first system program is "systemd,"** and it starts many other daemons. Once this phase is complete, the system is fully booted, and the system waits for some event to occur.
- Events are always signaled by the occurrence of an interrupt.

Operating-System Operations

- The **interrupts are from both** hardware and software
  - Hardware interrupt by one of the devices
  - Software interrupt (**exception** or **trap):**
    - Software error (e.g., division by zero)
    - a specific request from a user program that an operating-system service be performed by executing a special operation called a **system call.**
    - Other process problems include infinite loop, processes modifying each other

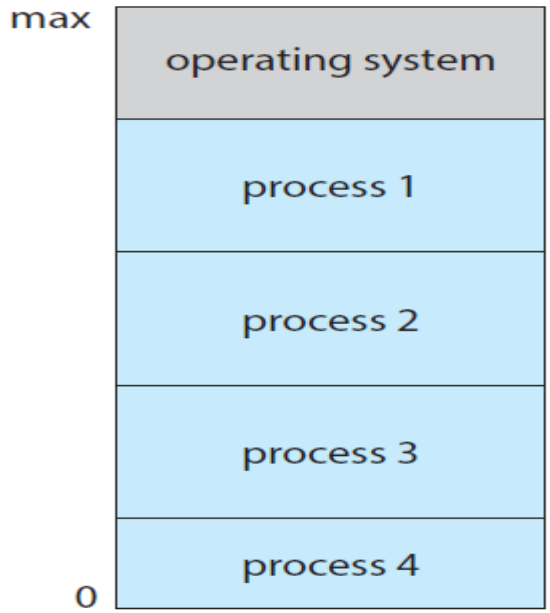
The OS operations are ,

1. Multiprogramming and Multi tasking
2. Dual mode and Multimode operation
3. Timer

i) Multiprogramming and Multitasking (Timesharing)

Multiprogramming

- One of the most important aspects of operating systems is the ability to run multiple programs, as a single program cannot keep either the CPU or the I/O devices busy at all times.
- Furthermore, users *want to run* more than one program at a time as well. **Multiprogramming increases CPU** utilization,
- In a multiprogrammed system, a program in execution is termed a **process.**

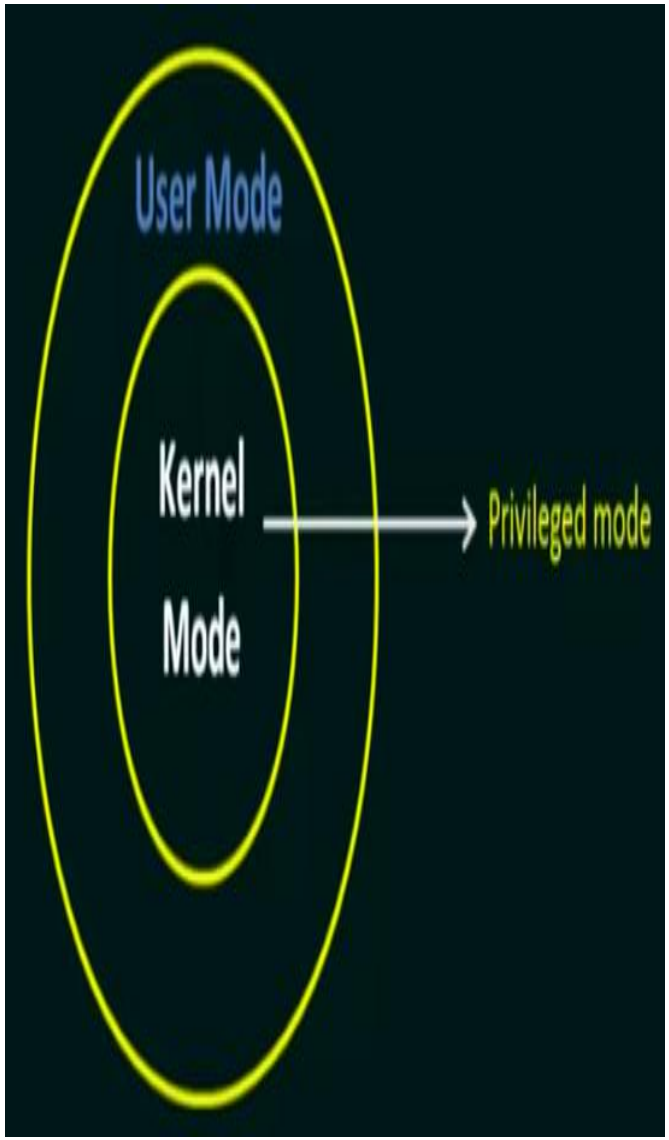


Multitasking (Timesharing)

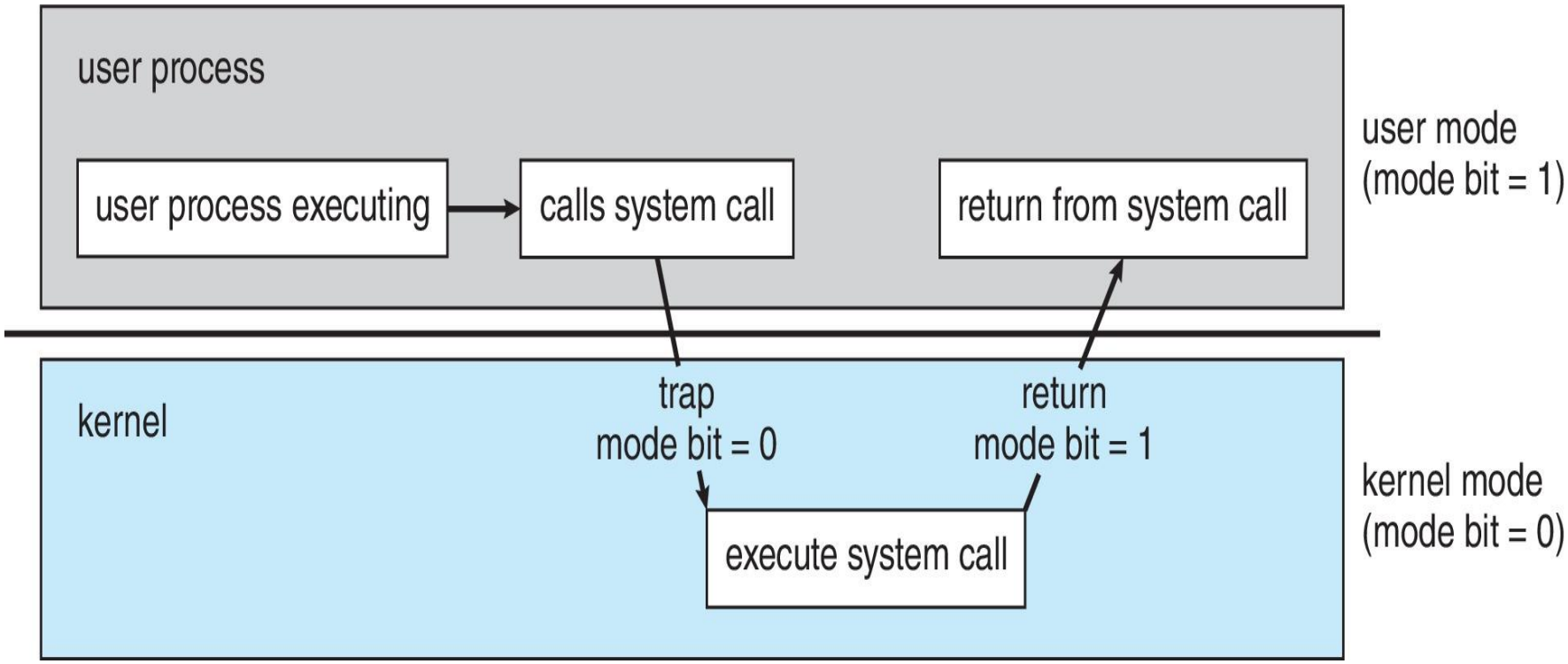
- The CPU switches jobs so frequently that users can interact with each job while it is running, creating **interactive** computing
- **Response time** should be < 1 second
- Each user has at least one program executing in memory ⇨**process**
- If several jobs ready to run at the same time ⇨ **CPU scheduling**
- If processes don't fit in memory, **swapping** moves them in and out to run
- **Virtual memory** allows execution of processes not completely in memory

2. Dual-mode and Multi mode Operation

- **Dual-mode operation allows OS to protect itself and other system components**
- **User mode and kernel mode[**(also called as **supervisor mode, system mode, or privileged mode**).
- **Mode bit provided by hardware**
- **Provides ability to distinguish when system is running user code or kernel code.**
- **When a user is running ➔ mode bit is "user" [No direct access to memory, h/w]**
- **When kernel code is executing ➔ mode bit is "kernel"**
- **System call changes mode to kernel mode, return from call resets it to user mode**



Transition from User to Kernel Mode



Some instructions designated as **privileged [direct access to memory, h/w]** , only executable in kernel mode

Need for user mode : When a program is executing in kernel mode, and **if that program happens to crash during its execution then the entire system would to crash. This is the problem of kernel mode**. But **if the program is executing in user mode, and if it crashes the entire system does not crash. So, user mode is safe mode.**

Multimode

- The concept of modes can be extended beyond two modes. For example, **Intel processors** have four separate **protection rings, where ring 0 is kernel mode and ring 3 is user mode.** Rings 1 and 2 could be used for various operating-system services, in practice they are rarely used.
- **ARMv8 systems have seven modes**. CPUs that support virtualization frequently have a separate mode to indicate when the **virtual machine manager (VMM) is** in control of the system. In this mode, the **VMM has more privileges than user mode but fewer than the kernel mode**.

3. Timer

- Timer to prevent infinite loop [or process hogging resources] (hogging - to take or use more than necessary of something)
  - Timer is set to interrupt the computer after some time period
  - Keep a counter that is decremented by the physical clock
  - Operating system set the counter (privileged instruction)
  - When counter is zero, it generate an interrupt
  - Set up before scheduling process to regain control or terminate program that exceeds allotted time

Resource Management

1. Process Management
2. Memory Management
3. File-system Management
4. Mass-Storage Management
5. Cache Management
6. I/O Subsystem

1. Process Management

- **A process is a program in execution**. It is a unit of work within the system. Program is a ***passive entity;*** process is an ***active entity***.
- Process needs resources to accomplish its task
  - CPU, memory, I/O, files, Initialization data
- Process termination requires reclaim of any reusable resources
- **Single-threaded process has one program counter** specifying location of next instruction to execute
  - Process executes instructions sequentially, one at a time, until completion
- **Multi-threaded process has one program counter per thread**
- A **system consists of a collection of processes**, some of which are **operating-system processes** (**those that execute system code**) and the **user processes** (**those that execute user code**). All these processes can potentially execute concurrently—by multiplexing on a single CPU core—or in parallel across multiple CPU cores.

Process Management Activities

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

- Creating and deleting both user and system processes
- Suspending and resuming processes
- Providing mechanisms for process synchronization
- Providing mechanisms for process communication
- Providing mechanisms for deadlock handling

2. Memory Management

- To execute a program, all (or part) of the instructions must be in memory
- Memory management determines what is in memory
  - Optimizing CPU utilization and response time to users

Memory management activities

- Keeping track of which parts of memory are currently being used and which process is using them
- Allocating and deallocating memory space as needed
- Deciding which processes and data to move into and out of memory

3. File-system Management

- OS provides uniform, logical view of information storage
  - Abstracts physical properties of a file to logical storage unit
  - Each medium is controlled by device (i.e., disk drive, tape drive)
    - Varying properties include access speed, capacity, data-transfer rate, access method (sequential or random)
- File-System management
  - Files usually organized into directories
  - Access control on most systems to determine who can access what
  - **OS activities include**
    - Creating and deleting files and directories
    - Primitives to manipulate files and directories
    - Mapping files onto secondary storage
    - Backup files onto stable (non-volatile) storage media

4. Mass-Storage Management

- Usually disks used to store data that does not fit in main memory or data that must be kept for a “long” period of time
- Proper management is of central importance
- Entire speed of computer operation hinges on disk subsystem and its algorithms
- **OS activities**
  - Mounting and unmounting
  - Free-space management
  - Storage allocation
  - Disk scheduling
  - Partitioning
  - Protection

5. Cache Management

- Important principle, performed at many levels in a computer (in hardware, operating system, software)
- Information in use copied from slower to faster storage temporarily
- Faster storage (cache) checked first to determine if information is there
  - If it is, information used directly from the cache (fast)
  - If not, data copied to cache and used there
- Cache smaller than storage being cached
  - Cache management important design problem
  - Cache size and replacement policy

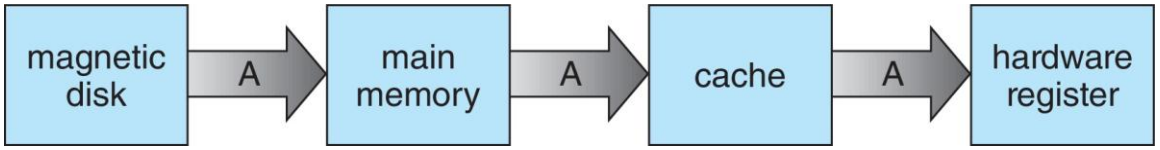
Characteristics of Various Types of Storage

Level	1	2	3	4	5
Name	registers	cache	main memory	solid-state disk	magnetic disk
Typical size	< 1 KB	< 16MB	< 64GB	< 1 TB	< 10 TB
Implementation technology	custom memory with multiple ports CMOS	on-chip or off-chip CMOS DRAM	CMOS DRAM	flash memory	magnetic disk
Access time (ns)	0.25-0.5	0.5-25	80-250	25,000-50,000	5,000,000
Bandwidth (MB/sec)	20,000-100,000	5,000-10,000	1,000-5,000	500	20-150
Managed by	compiler	hardware	operating system	operating system	operating system
Backed by	cache	main memory	disk	disk	disk or tape

Movement between levels of storage hierarchy can be explicit or implicit

Migration of data “A” from Disk to Register

- Multitasking environments must be careful to use most recent value, no matter where it is stored in the storage hierarchy



- **Multiprocessor environment** must provide **cache coherency** in hardware such that **all CPUs have the most recent value in their cache**
- In a distributed environment, the situation becomes even more complex . In this environment, several copies (or replicas) of the same file can be kept on different computers. 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.

6. I/O Subsystem

- One purpose of OS is to hide peculiarities of hardware devices from the user
- I/O subsystem responsible for
  - Memory management of I/O including buffering (storing data temporarily while it is being transferred), caching (storing parts of data in faster storage for performance), spooling (the overlapping of output of one job with input of other jobs)
  - General device-driver interface
  - Drivers for specific hardware devices

Protection and Security

- **Protection** – any mechanism for controlling access of processes or users to resources defined by the OS
- **Security** – defense of the system against internal and external attacks
  - Huge range, including denial-of-service, worms, viruses, identity theft, theft of service
- Systems generally first distinguish among users, to determine who can do what
  - User identities (**user IDs, security IDs**) include name and associated number, one per user
  - User ID then associated with all files, processes of that user to determine access control
  - Group identifier (**group ID**) allows set of users to be defined and controls managed, then also associated with each process, file
  - **Privilege escalation** allows user to change to effective ID with more rights

Difference between compiler and Interpreter

Compiler	Interpreter
A compiler translates the entire source code in a single run.	An interpreter translates the entire source code line by line.
It consumes less time i.e., it is faster than an interpreter.	It consumes much more time than the compiler i.e., it is slower than the compiler.
It is more efficient.	It is less efficient.
CPU utilization is more.	CPU utilization is less as compared to the compiler.
Both syntactic and semantic errors can be checked simultaneously.	Only syntactic errors are checked.
The compiler is larger.	Interpreters are often smaller than compilers.
The localization of errors is difficult.	The localization of error is easier than the compiler.
A presence of an error can cause the whole program to be re-organized.	A presence of an error causes only a part of the program to be re-organized.
The compiler is used by the language such as C, C++.	An interpreter is used by languages such as Java.

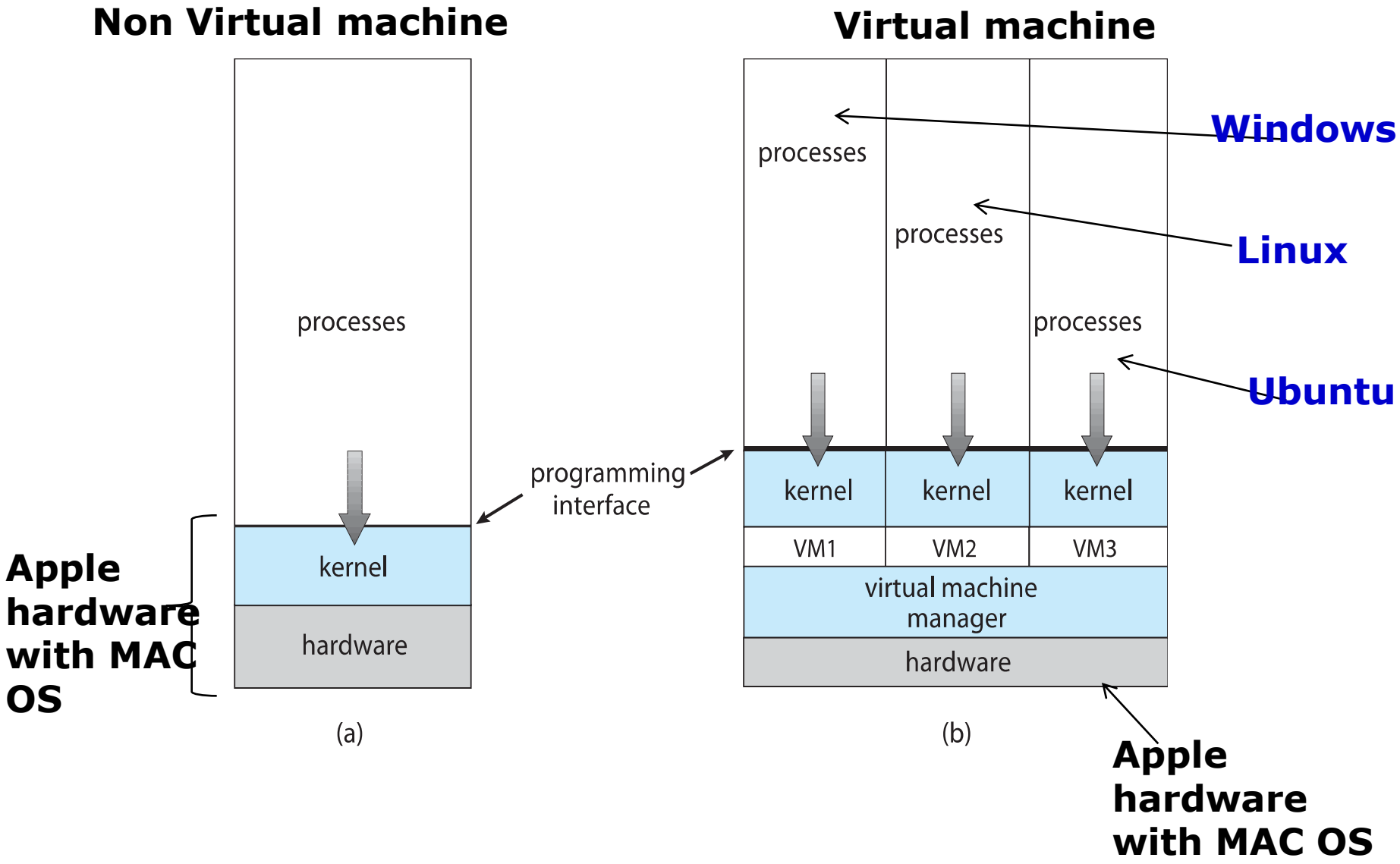
Virtualization

- Allows operating systems to run applications within other OSes
  - Vast and growing industry
- **Emulation, which involves simulating computer hardware in software,** is typically used **when the source CPU type is different from the target CPU type.** For example, when Apple switched from the IBM Power CPU to the Intel x86 CPU for its desktop and laptop computers, it included an emulation facility called "**Rosetta,"** which allowed applications compiled for the IBM CPU to run on the Intel CPU. **Emulation comes at a heavy price**.
- **Virtualization** is used the hardware of a single computer(CPU, Memory, disk, network interface card) into several execution environments, thereby **creating illusion that each separate execution environment is running its own private computer.** OS natively compiled for CPU, running **guest** OSes also natively compiled
- [Ex]:VMware running WinXP guests, each running applications, all on native WinXP **host** OS.**VMM** (virtual machine Manager) provides virtualization services

Virtualization (cont.)

- Use cases involve laptops and desktops running multiple OSes for exploration or compatibility
  - Apple laptop running Mac OS X host, Windows as a guest
  - **Developing apps for multiple OSes without having multiple systems**
  - Quality assurance testing applications without having multiple systems
  - Executing and managing compute environments within data centers
- VMM can run natively, in which case they are also the host
  - There is no general-purpose host then (VMware ESX and Citrix XenServer)

Computing Environments - Virtualization



Distributed Systems

- Distributed computing
  - Collection of separate, possibly heterogeneous, systems networked together
    - **Network** is a communications path, **TCP/IP** most common
      - **Local Area Network** (**LAN**)
      - **Wide Area Network** (**WAN**)
      - **Metropolitan Area Network** (**MAN**)
      - **Personal Area Network** (**PAN**)
  - **Network Operating System** provides features between systems across network
    - Communication scheme allows systems to exchange messages
    - Illusion of a single system

Computer System Environments

Computing Environments

- Traditional
- Mobile
- Client Server
- Pear-to-Pear
- Cloud computing
- Real-time Embedded

Traditional

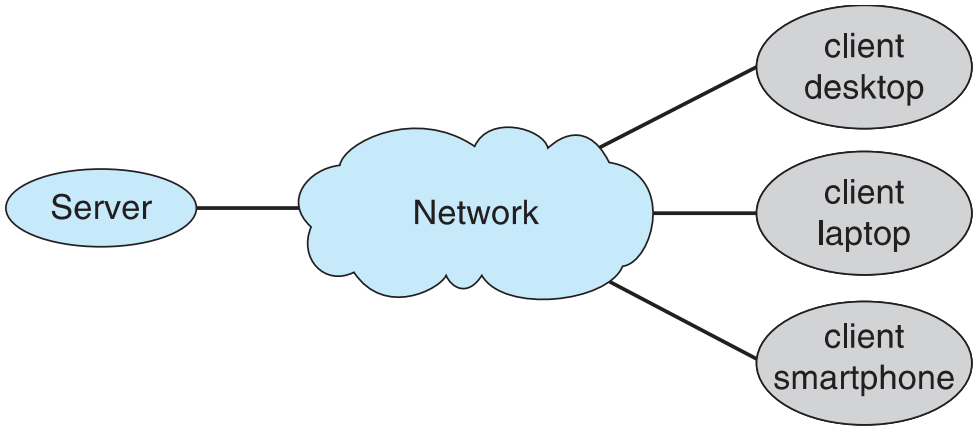
- Stand-alone general-purpose machines
- But most systems interconnect with others (i.e., the Internet)
- **Portals** provide web access to internal server systems
- **Network computers** (**thin clients**) which are essentially terminals that understand web-based computing— are used in place of traditional workstations where more security or easier maintenance is desired.
- Mobile computers interconnect via **wireless networks**
- Networking becoming ubiquitous – even home systems use **firewalls** to protect home computers from Internet attacks

Mobile

- Handheld smartphones, tablets, etc.
- The functional difference between mobile and a “traditional” laptop is – more OS features (GPS, gyroscope)
- Allows new types of apps like ***augmented reality***
- Use IEEE 802.11 wireless, or cellular data networks for connectivity
- Leaders are **Apple iOS** and **Google Android**

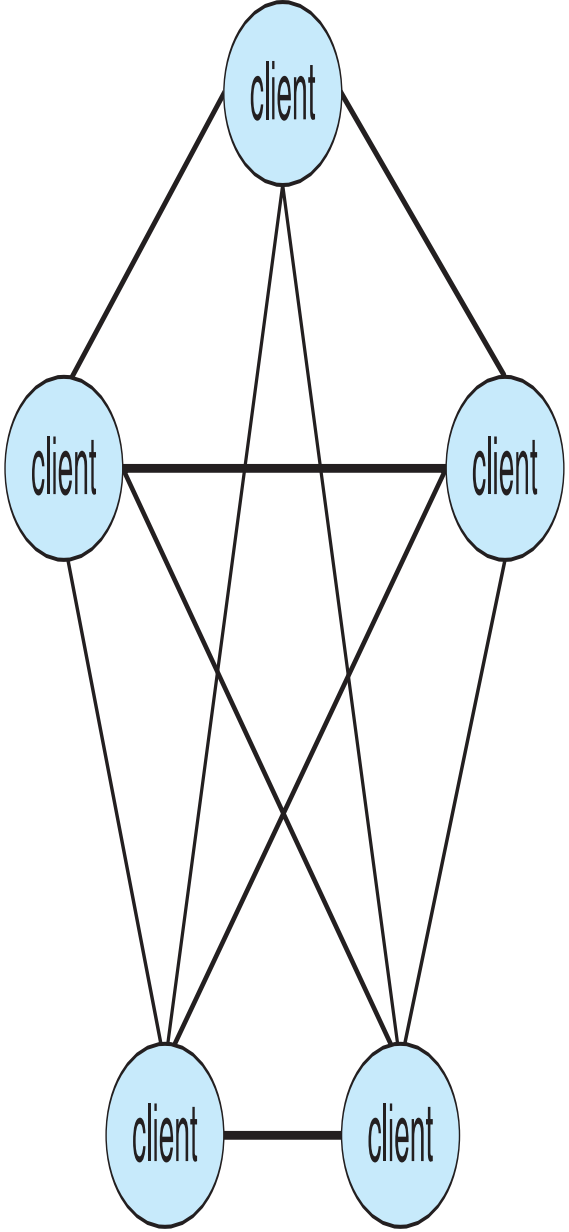
Client Server

- Client-Server Computing
  - Dumb terminals supplanted by smart PCs
  - Many systems now **servers**, responding to requests generated by **clients**
    - **Compute-server system** provides an interface to client to request services (i.e., database)
    - **File-server system** provides interface for clients to store and retrieve files



Peer-to-Peer

- Another model of distributed system with no centralized service
- P2P does not distinguish clients and servers
  - Instead all nodes are considered peers
  - May each act as client, server or both
  - Node must join P2P network
    - Registers its service with central lookup service on network, or
    - Broadcast request for service and respond to requests for service via ***discovery protocol***
  - Examples include Napster and Gnutella, **Voice over IP** (**VoIP**) such as Skype



Cloud Computing

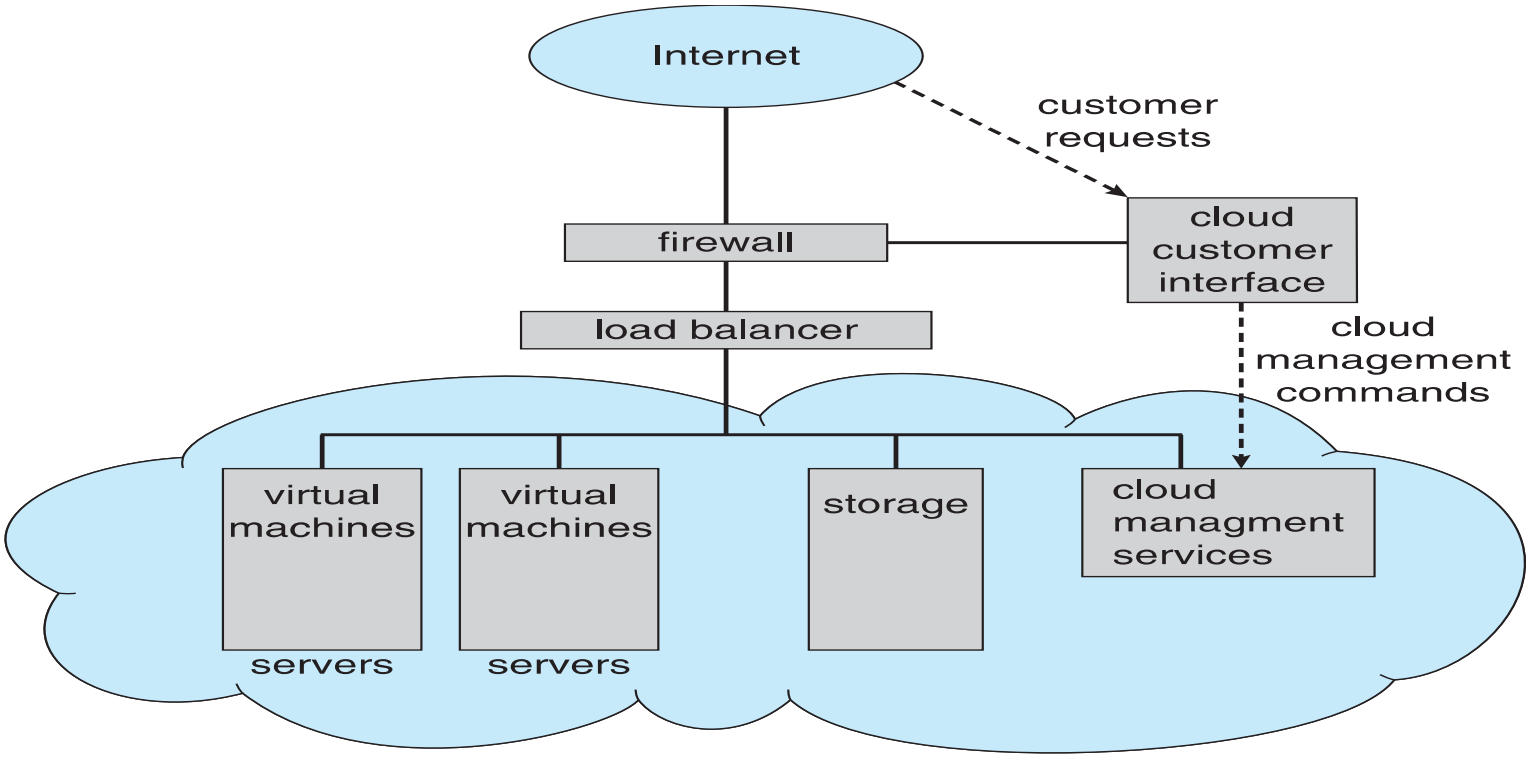
- Delivers computing, storage, even apps as a service across a network
- Logical extension of virtualization because it uses virtualization as the base for it functionality.
  - Amazon **EC2** has thousands of servers, millions of virtual machines, petabytes of storage available across the Internet, pay based on usage

Cloud Computing (Cont.)

- Many types
  - **Public cloud** – available via Internet to anyone willing to pay
  - **Private cloud** – run by a company for the company's own use
  - **Hybrid cloud** – includes both public and private cloud components
  - Software as a Service (**SaaS**) – one or more applications available via the Internet (i.e., word processor)
  - Platform as a Service (**PaaS**) – software stack ready for application use via the Internet (i.e., a database server)
  - Infrastructure as a Service (**IaaS**) – servers or storage available over Internet (i.e., storage available for backup use)

Cloud Computing (cont.)

- Cloud computing environments composed of traditional OSes, plus VMMs, plus cloud management tools
  - Internet connectivity requires security like firewalls
  - Load balancers spread traffic across multiple applications



Real-Time Embedded Systems

- Embedded computers are the most prevalent form of computers in existence. These devices are found everywhere, from car engines, robots 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.**
- Many other special computing environments as well
  - **Some have OSes, some perform tasks without an OS**
- **Real-time OS has well-defined fixed time constraints**
  - Processing ***must*** be done within constraint
  - Correct operation execute , only if constraints met

Free and Open-Source Operating Systems

- Operating systems made available in source-code format rather than just binary **closed-source** and **proprietary[Ex: Windows]**
- Counter to the **copy protection** and **Digital Rights Management (DRM)** movement
- Started by **Free Software Foundation (FSF)**, which has "copyleft" **GNU Public License (GPL)**
  - Free software and open-source software are two different ideas championed by different groups of people
    - http://gnu.org/philosophy/open-source-misses-the-point.html/
- Examples include **GNU/Linux** and **BSD UNIX**, and many more
- Can use VMM like VMware Player (Free on Windows), Virtualbox (open source and free on many platforms - http://www.virtualbox.com)
  - Use to run guest operating systems for exploration

Operating-System Services – chapter2

In this chapter, to study about

- Operating system services
- User and Operating System-Interface
- System Calls
- System Services
- Linkers and Loaders
- Why Applications are Operating System Specific
- Design and Implementation
- Operating System Structure
- Building and Booting an Operating System
- Operating System Debugging

Operating System Services

Operating systems provide an environment for execution of programs and services to programs and users

1. **One set of operating-system services provides functions that are helpful to the user:**
   - **User interface** - Almost all operating systems have a user interface (**UI**).
     - Varies between **Command-Line Interface (CLI), Graphics User Interface (GUI), touch-screen**
   - **Program execution** - The system must be able to load a program into memory and to run that program, end execution, either normally or abnormally (indicating error)
   - **I/O operations** - A running program may require I/O, which may involve a file or an I/O device
   - **File-system manipulation** - Programs need to read and write files and directories, create and delete them, search them, list file Information, permission management.

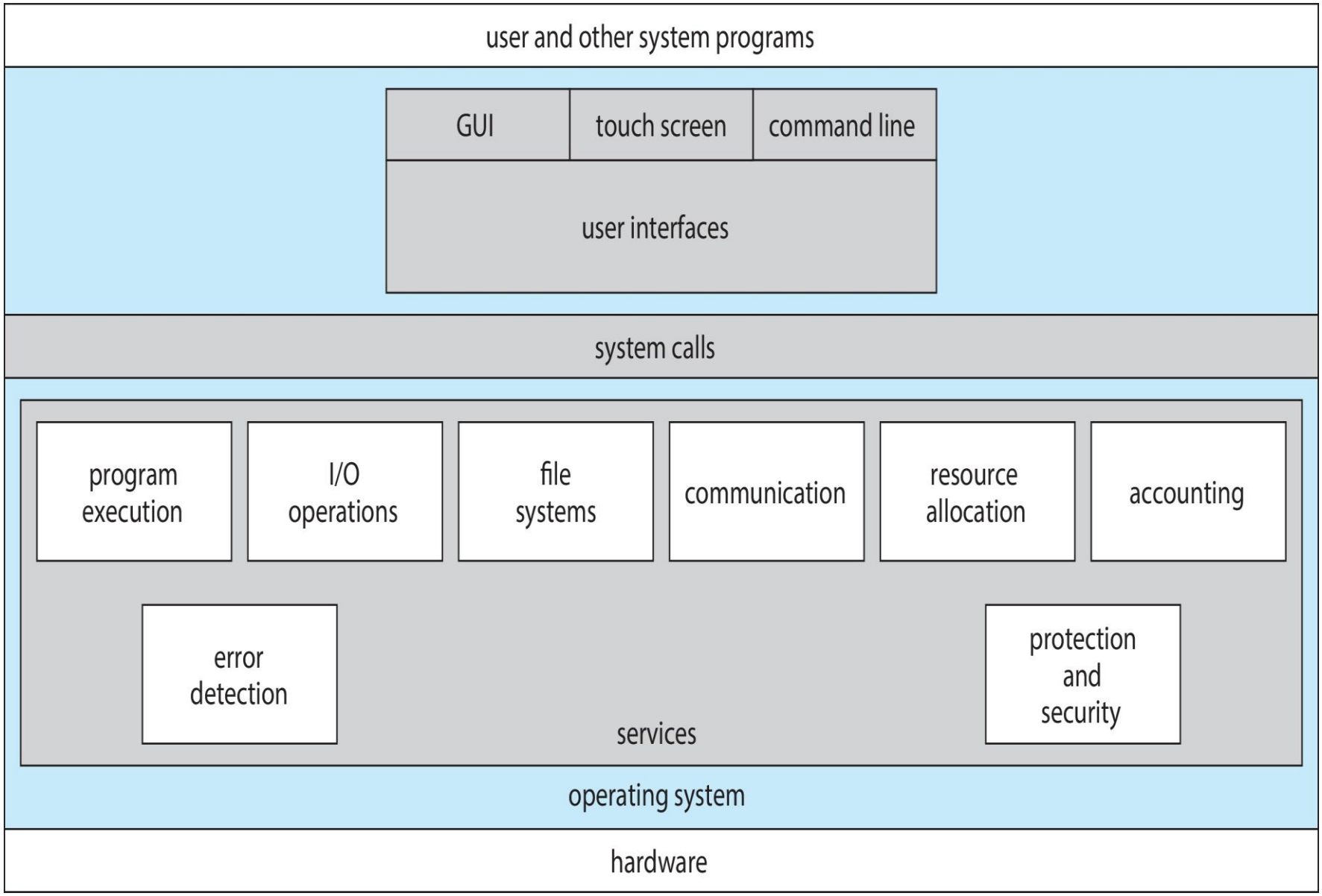
Operating System Services (Cont.)

- **Communications** – Processes may exchange information, on the same computer or between computers over a network
  - Communications may be via shared memory or through message passing (packets moved by the OS)
- **Error detection** – OS needs to be constantly aware of possible errors
  - May occur in the CPU and memory hardware, in I/O devices, in user program
  - For each type of error, OS should take the appropriate action to ensure correct and consistent computing **[ex- paper out in printer]**
  - Debugging facilities can greatly enhance the user's and programmer's abilities to efficiently use the system

Operating System Services (Cont.)

2. **Another set of OS functions exists for ensuring the efficient operation of the system itself via resource sharing**
   - **Resource allocation** - When multiple users or multiple jobs running concurrently, resources must be allocated to each of them
     - Many types of resources - CPU cycles, main memory, file storage, I/O devices.
   - **Logging** - To keep track of which users use how much and what kinds of computer resources
   - **Protection and security** - The owners of information stored in a multiuser or networked computer system may want to control use of that information, concurrent processes should not interfere with each other
     - **Protection** involves ensuring that all access to system resources is controlled
     - **Security** of the system from outsiders requires user authentication, extends to defending external I/O devices from invalid access attempts

A View of Operating System Services



Command Line Interface or Command Interpreter

- CLI allows direct command entry – text based commands [eg- command prompt in windows, terminal in Linux] – user has to remember all the commands to perform tasks
- Some OS implemented in kernel, some OS treat CLI as systems program like windowsXP
- Sometimes multiple command interpreters are implemented in Linux – called as **shells**

✓ **eg – Bourne shell**

✓ **C shell**

✓ **Bourne -Again shell(BASH)**

✓ **Korn shell**

- Primarily fetches a command from user and executes it
- Sometimes commands built-in, sometimes just names of programs

Command Interpreter in Windows – create and delete directory

1. Type cmd in search
2. Type “ cd desktop”
3. Type “mkdir jp”
4. Check the desktop , the directory is there or not
5. Or type “dir” [list the directories is in desktop]
6. Type “rmdir jp”

Some network commands

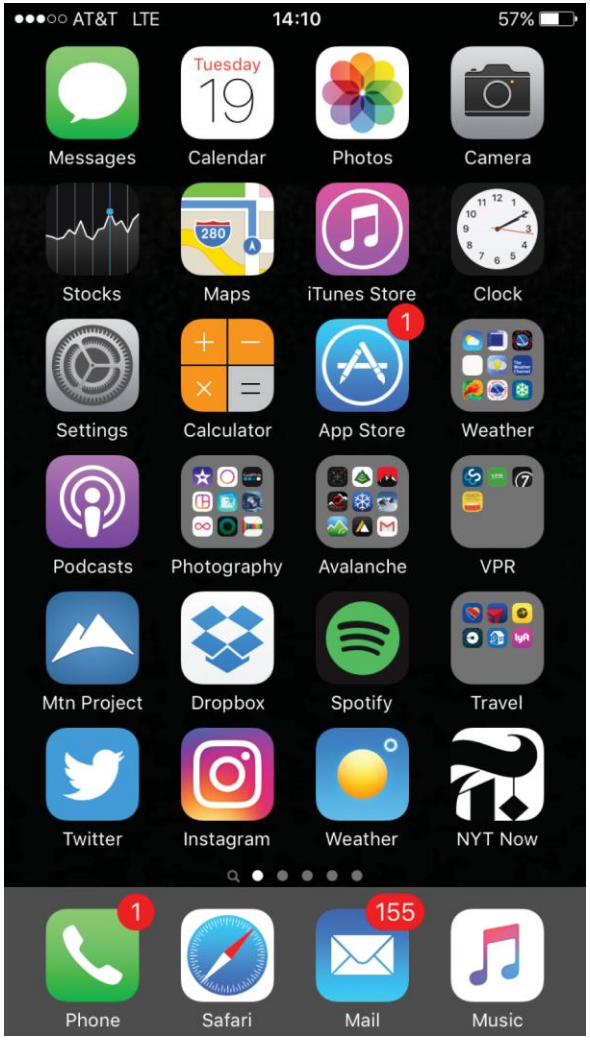
1. Ipconfig
2. Ping
3. Hostname
4. Getmac
5. arp

User Operating System Interface - GUI

- User-friendly **desktop** interface
  - Usually mouse, keyboard, and monitor
  - **Icons** represent files, programs, actions, etc
  - Various mouse buttons over objects in the interface cause various actions (provide information, options, execute function, open directory (known as a **folder**)
  - Invented at Xerox PARC
- Many systems now include both CLI and GUI interfaces
  - Microsoft Windows is GUI with CLI "command" shell
  - Apple Mac OS X is "Aqua" GUI interface with UNIX kernel underneath and shells available
  - Unix and Linux have CLI with optional GUI interfaces (CDE, KDE, GNOME)

Touchscreen Interfaces

- Touchscreen devices require new interfaces
  - Mouse not possible or not desired
  - Actions and selection based on gestures
  - Virtual keyboard for text entry
- Voice commands



System Calls

•System calls provide an interface to the services made available by an operating system.

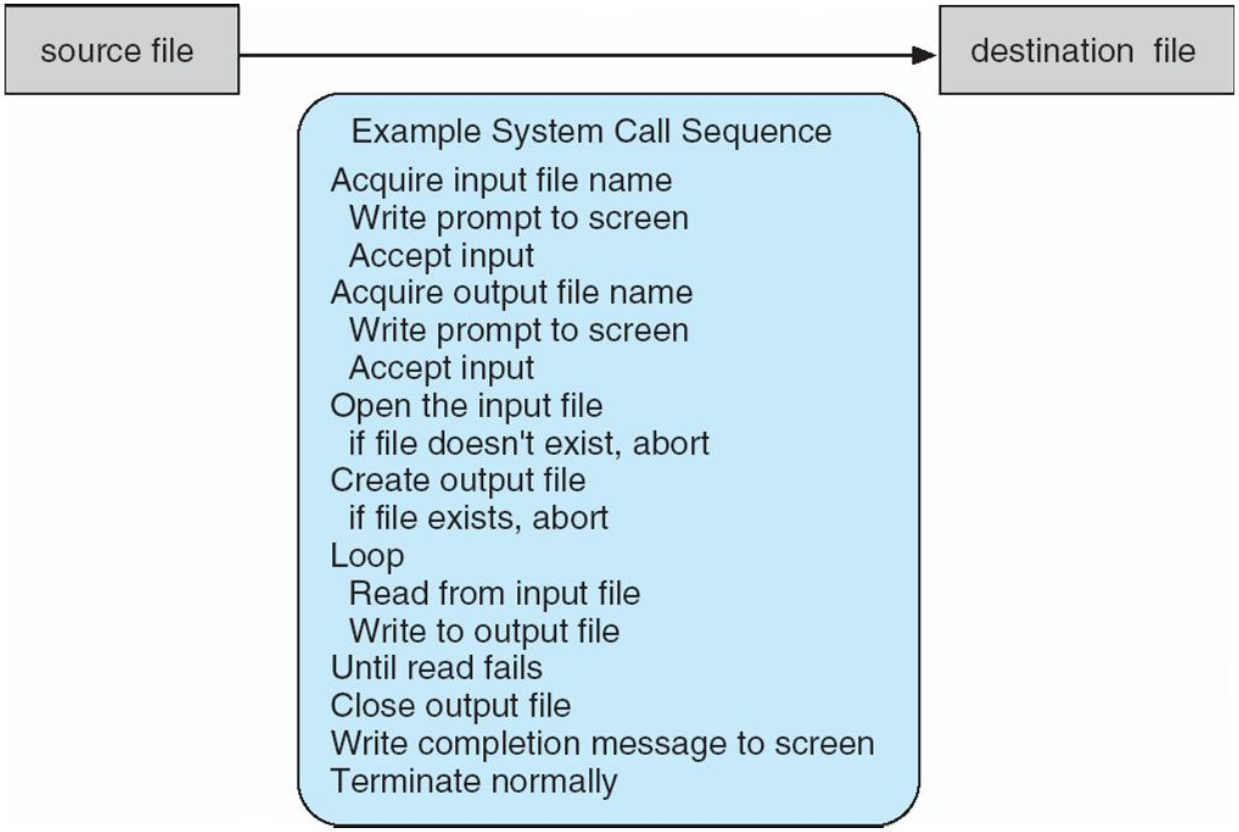
•These calls are generally available as functions written in C and C++.

- Mostly accessed by programs via a high-level **Application Programming Interface (API) rather than direct system call**
- Three most common APIs are Win32 API for Windows, POSIX API for POSIX-based systems (including virtually all versions of UNIX, Linux, and Mac OS X), and Java API for the Java virtual machine (JVM)
- System call to copy the contents of one file to another file in Unix

cp in.txt out.txt

Example of System Call sequence in GUI to make copy the contents

- System call sequence to copy the contents of one file to another file



Example of Standard API

EXAMPLE OF STANDARD API

As an example of a standard API, consider the `read()` function that is available in UNIX and Linux systems. The API for this function is obtained from the `man` page by invoking the command

```
man read
```

on the command line. A description of this API appears below:

```
#include <unistd.h>

ssize_t     read(int fd, void *buf, size_t count)
```

return value — function name — parameters

A program that uses the `read()` function must include the `unistd.h` header file, as this file defines the `ssize_t` and `size_t` data types (among other things). The parameters passed to `read()` are as follows:

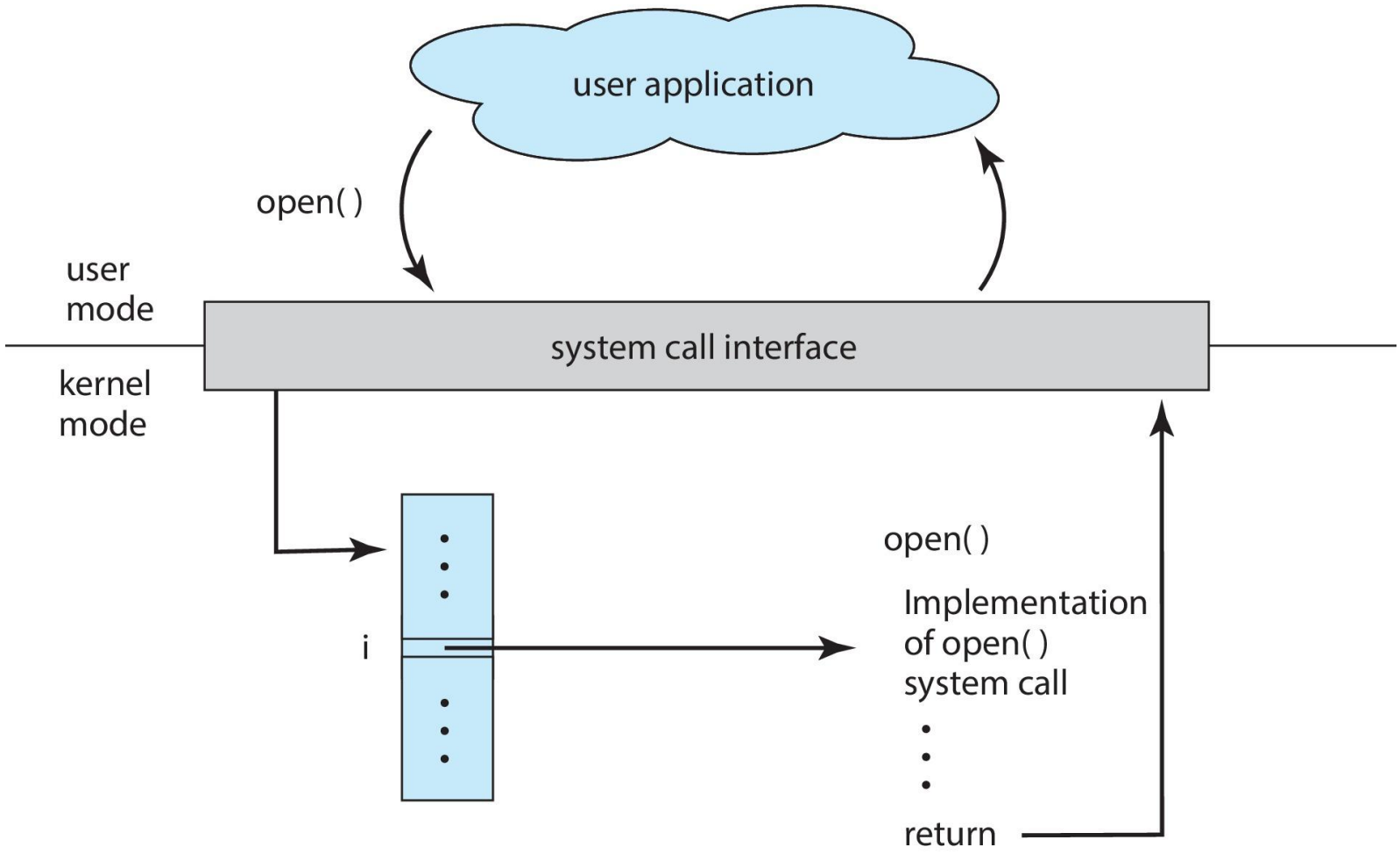
- `int fd`—the file descriptor to be read
- `void *buf`—a buffer into which the data will be read
- `size_t count`—the maximum number of bytes to be read into the buffer

On a successful read, the number of bytes read is returned. A return value of 0 indicates end of file. If an error occurs, `read()` returns −1.

System Call Implementation

- Typically, a number is associated with each system call
  - **System-call interface** maintains a table indexed according to these numbers
- The system call interface invokes the intended system call in OS kernel and returns status of the system call and any return values
- The caller need know nothing about how the system call is implemented
  - Just needs to obey API and understand what OS will do as a result call
  - Most details of OS interface hidden from programmer by API
    - Managed by run-time support library (set of functions built into libraries included with compiler)

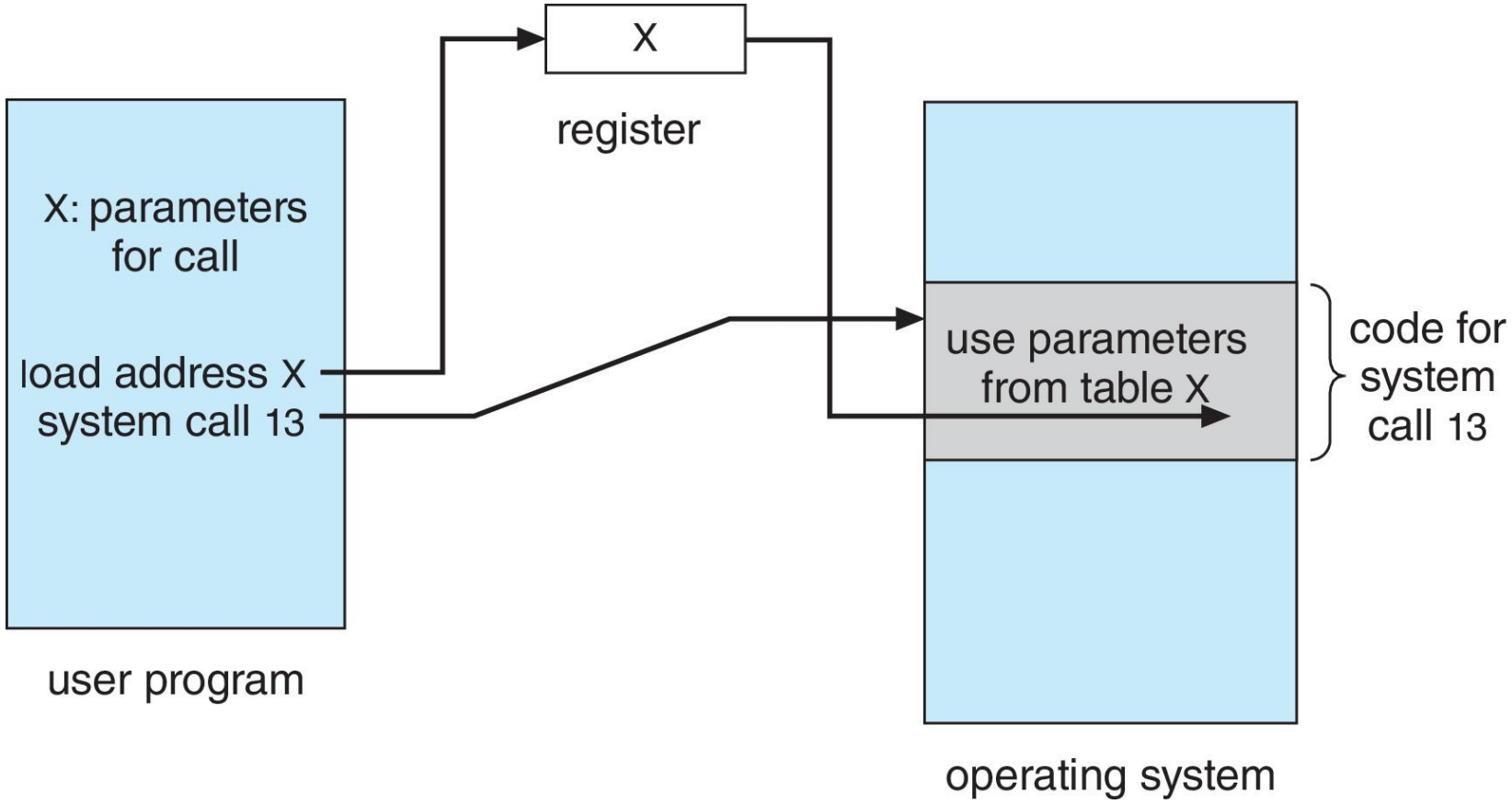
API – System Call – OS Relationship



System Call Parameter Passing

- Often, more information is required than simply identity of desired system call
  - Exact type and amount of information vary according to OS and call
- Three general methods used to pass parameters to the OS
  - Simplest: pass the parameters in registers
    - In some cases, may be more parameters than registers
  - Parameters stored in a block, or table, in memory, and address of block passed as a parameter in a register
    - This approach taken by Linux and Solaris
  - Parameters placed, or **pushed**, onto the **stack** by the program and **popped** off the stack by the operating system
  - Block and stack methods do not limit the number or length of parameters being passed

Parameter Passing via Table



Types of System Calls

- **Process control**
  - create process, terminate process
  - end, abort
  - load, execute
  - get process attributes, set process attributes
  - wait for time
  - wait event, signal event
  - allocate and free memory
  - Dump memory if error
  - **Debugger** for determining **bugs, single step** execution
  - **Locks** for managing access to shared data between processes

Types of System Calls (Cont.)

- **File management**
  - create file, delete file
  - open, close file
  - read, write, reposition
  - get and set file attributes
- **Device management**
  - request device, release device
  - read, write, reposition
  - get device attributes, set device attributes
  - logically attach or detach devices

Types of System Calls (Cont.)

- **Information maintenance**
  - get time or date, set time or date
  - get system data, set system data
  - get and set process, file, or device attributes
- **Communications**
  - create, delete communication connection
  - send, receive messages
    - From **client** to **server**
  - **Shared-memory model** create and gain access to memory regions
  - transfer status information
  - attach and detach remote devices
- **Protection**

  Control access to resources

  Get and set permissions

  Allow and deny user access

Examples of Windows and Unix System Calls

EXAMPLES OF WINDOWS AND UNIX SYSTEM CALLS

The following illustrates various equivalent system calls for Windows and UNIX operating systems.

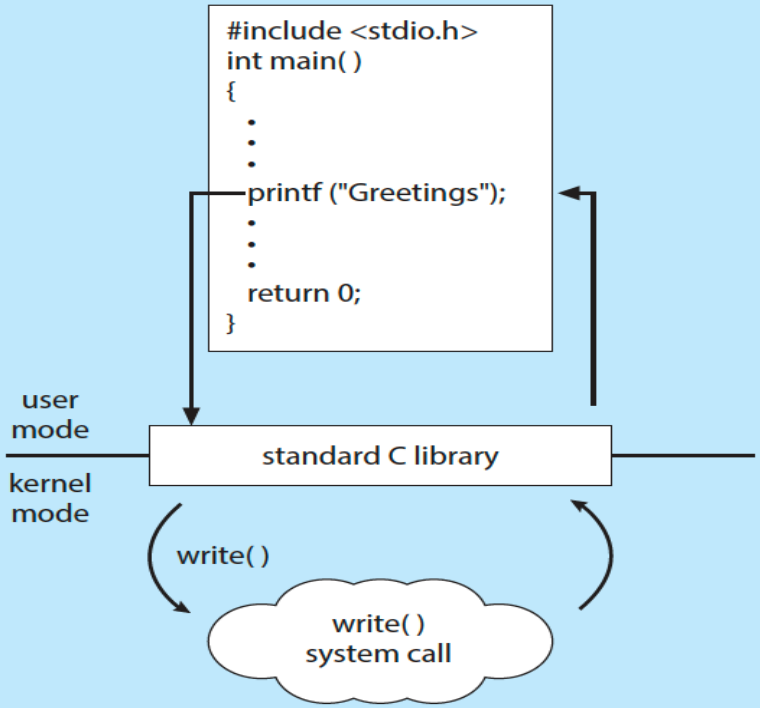
	Windows	Unix
Process control	`CreateProcess()`	`fork()`
	`ExitProcess()`	`exit()`
	`WaitForSingleObject()`	`wait()`
File management	`CreateFile()`	`open()`
	`ReadFile()`	`read()`
	`WriteFile()`	`write()`
	`CloseHandle()`	`close()`
Device management	`SetConsoleMode()`	`ioctl()`
	`ReadConsole()`	`read()`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

Standard C Library Example

- C program invoking printf() library call, which calls write() system call

THE STANDARD C LIBRARY

The standard C library provides a portion of the system-call interface for many versions of UNIX and Linux. As an example, let's assume a C program invokes the `printf()` statement. The C library intercepts this call and invokes the necessary system call (or calls) in the operating system—in this instance, the `write()` system call. The C library takes the value returned by `write()` and passes it back to the user program:



System call for process control

➢Two or more processes may share data. To ensure the integrity of the data being shared, **operating systems often provide system call**s allowing a process to **lock** shared data. Then, no other process can access the data until the lock is released. **Typically, such system calls include acquire lock() and release lock().**

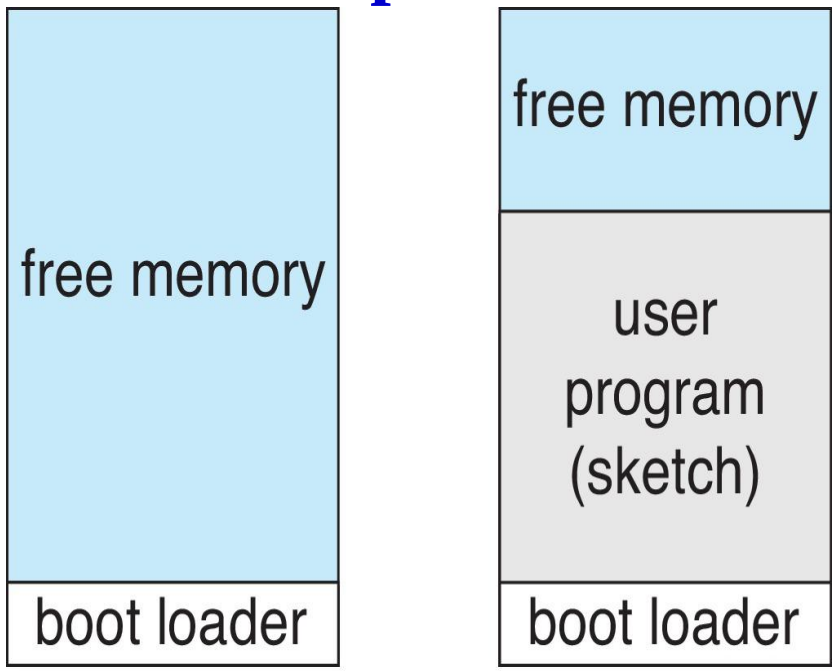
➢There are so many variations in process control. **Two examples— one involving a single-tasking system and the other a multitasking system**

1. Single tasking – [ex] - Arduino

Arduino is an open-source platform consisting of a microcontroller along with input sensors that respond to a variety of events based on easy-to-use hardware and software. Arduino are able to **read inputs - light on a sensor, a finger on a button - and turn it into an output - activating a motor, turning on an LED**

Example: Arduino

- To write a program for the Arduino, **first write the program on a PC and then upload the compiled program (known as a sketch) from the PC to the Arduino's flash memory via a USB connection.**
- The standard Arduino platform does not provide an operating system; instead, a **small piece of software known as a boot loader loads the sketch into a specific region in the Arduino's memory .**
- Once the sketch has been loaded, it begins running, waiting for the events that it is programmed to respond to.



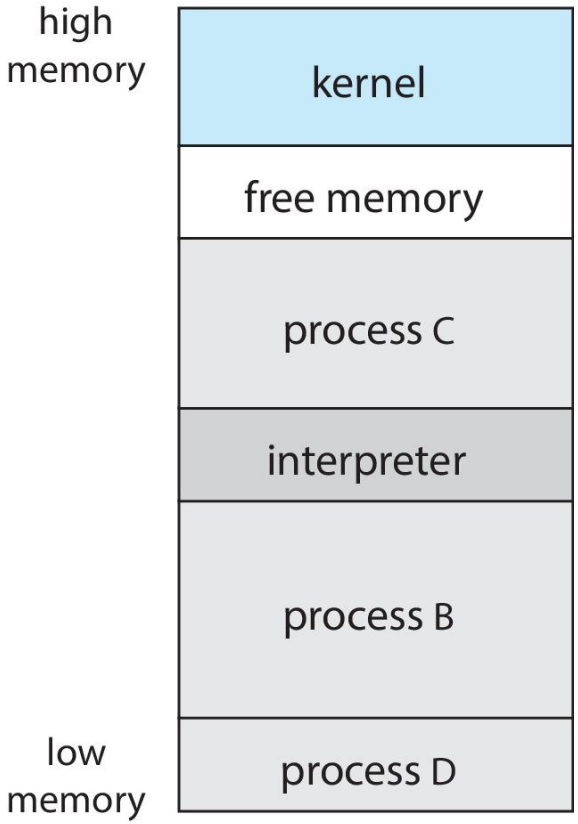
(a) At system startup

(b) running a program

- An Arduino is considered a single-tasking system, **as only one sketch can be present in memory at a time; if another sketch is loaded, it replaces the existing sketch.**

Example: FreeBSD

- Unix variant
- Multitasking
- User login -> invoke user's choice of shell [4 shells in Linux]
- Shell executes fork() system call to create process
  - Executes exec() to load program into process
  - Shell waits for process to terminate or continues with user commands
- Process exits with:
  - code = 0 – no error
  - code > 0 – error code



System Services

- System programs **provide a convenient environment for program development and execution**. They can be divided into:
  - File manipulation
  - Status information sometimes stored in a file
  - Programming language support
  - Program loading and execution
  - Communications
  - Background services
  - Application programs

System Services (Cont.)

- **File management** - Create, delete, copy, rename, print, dump, list, and generally manipulate files and directories

- **Status information**
  - Some ask the system for info - date, time, amount of available memory, disk space, number of users
  - Others provide detailed performance, logging, and debugging information
  - Typically, these programs format and print the output to the terminal or other output devices
  - Some systems implement a **registry** - used to store and retrieve configuration information

System Services (Cont.)

- **Programming-language support** - Compilers, assemblers, debuggers and interpreters sometimes provided
- **Program loading and execution-** Absolute loaders, relocatable loaders, linkage editors, and overlay-loaders, debugging systems for higher-level and machine language
- **Communications** - Provide the mechanism for creating virtual connections among processes, users, and computer systems
  - Allow users to send messages to one another's screens, browse web pages, send electronic-mail messages, log in remotely, transfer files from one machine to another
- **File modification**
  - Text editors to create and modify files
  - Special commands to search contents of files or perform transformations of the text

System Services (Cont.)

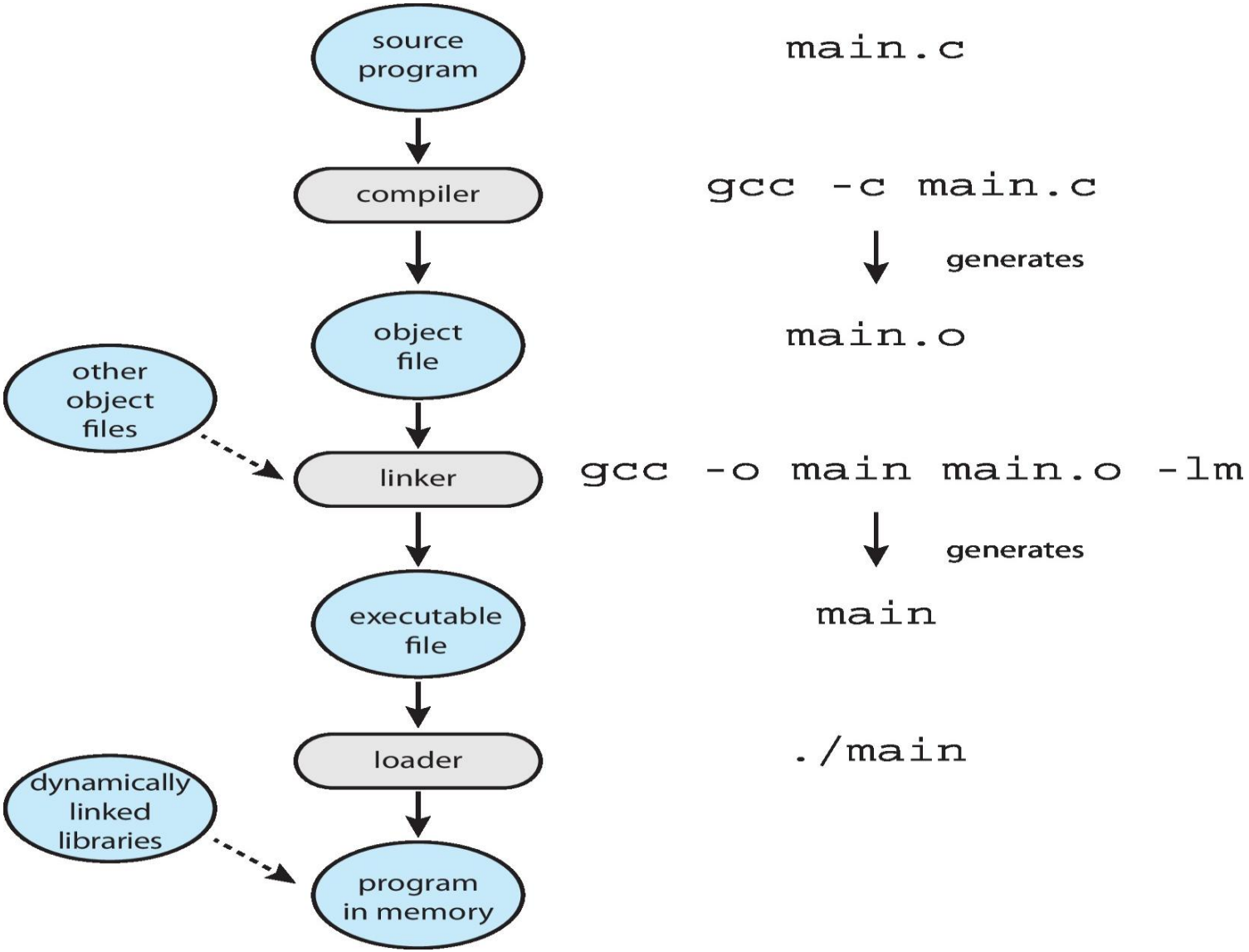
- **Background Services**
  - Launch at boot time
    - Some for system startup, then terminate
    - Some from system boot to shutdown
  - Provide facilities like disk checking, process scheduling, error logging, printing
  - Run in user context not kernel context
  - Known as **services**, **subsystems**, **daemons**

- **Application programs**
  - Don’t pertain to system
  - Run by users
  - Not typically considered part of OS
  - Launched by command line, mouse click, finger poke

Linkers and Loaders

- **Compiler :** Source code compiled into object files designed to be loaded into any physical memory location – **relocatable object file**
- **Linker** combines these into single binary **executable** file
  - Also brings in libraries
- Program resides on secondary storage as binary executable
- Must be brought into memory by **loader** to be executed
  - **Relocation** assigns final addresses to program parts and adjusts code and data in program to match those addresses
- Modern general purpose systems don't link libraries into executables
  - Rather, **Dynamically Linked Libraries(DLLs) [Ex-in Windows],** are loaded as needed, shared by all that use the same version of that same library (loaded once)
- Object **executable files have standard formats, so operating system knows how to load and start them**

The Role of the Linker and Loader



Why Applications are Operating System Specific

- **Apps compiled on one system usually not executable on other operating systems**
- Each operating system **provides its own unique system calls**
  - **Own file formats, etc.**
- **If Apps can be multi-operating system, then it**
  - Written in interpreted language like Python, Ruby and interpreter available on multiple operating systems
  - written in language(like Java) that includes a VM containing the running app
  - Use standard language (like C), compile separately on each operating system to run on each
- **Application Binary Interface (ABI)** is architecture equivalent of API, defines how different components of binary code can interface for a given operating system on a given architecture, CPU, etc.

Design and Implementation of Operating System

- Design and Implementation of OS is not “solvable”, but some approaches have proven successful
- **Internal structure of different Operating Systems can vary widely**
- Start the design by defining goals and specifications
- Affected by choice of hardware, type of system
- **User** goals and **System** goals
  - User goals – operating system should be convenient to use, easy to learn, reliable, safe, and fast
  - System goals – operating system should be easy to design, implement, and maintain, as well as flexible, reliable, error-free, and efficient
- **Specifying and designing an OS is highly creative task of software engineering**

Policy and Mechanism

- **Policy:** **What** needs to be done?
  - Example: Interrupt after every 100 seconds
- **Mechanism:** **How** to do something?
  - Example: timer
- Important principle: separate policy from mechanism
- The separation of policy from mechanism is a very important principle, it allows maximum flexibility if policy decisions are to be changed later.
  - Example: change 100 to 200

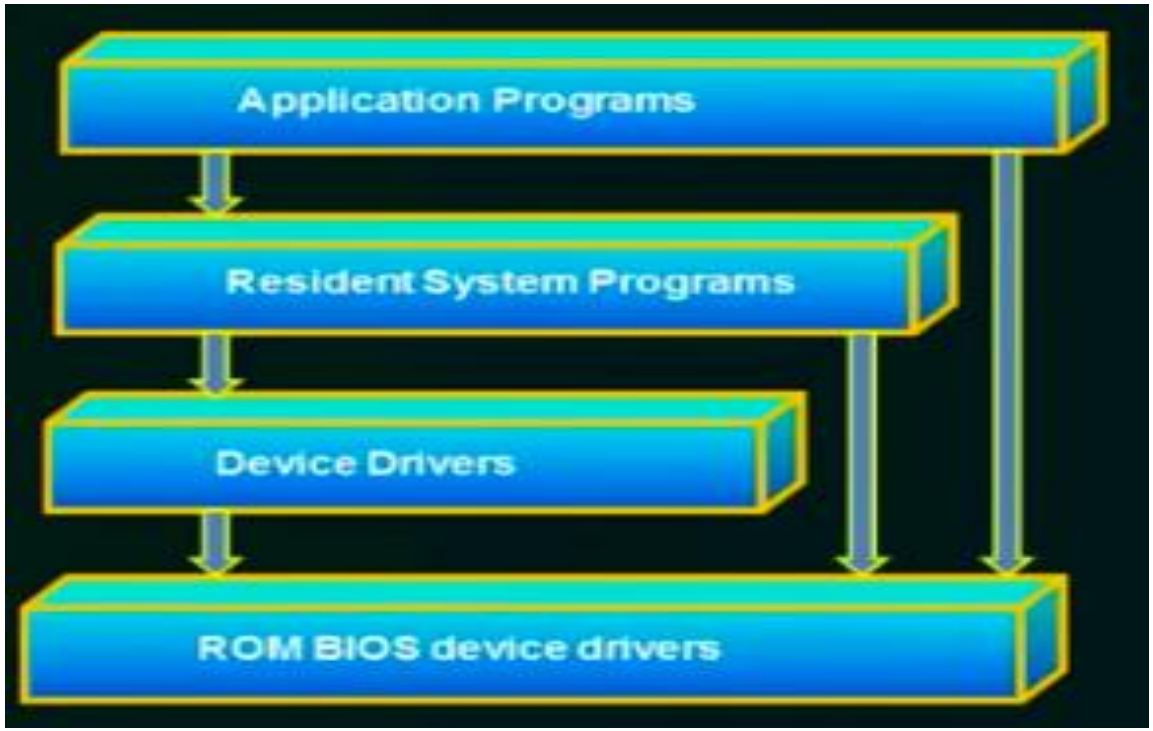
Implementation

- Much variation
  - Early OSes in assembly language
  - Then system programming languages like Algol, PL/1
  - Now C, C++
- Actually usually a mix of languages
  - Lowest levels in assembly
  - Main body in C
  - Systems programs in C, C++, scripting languages like PERL, Python, shell scripts
- More high-level language easier to **port** to other hardware
  - But slower
- **Emulation** can allow an OS to run on non-native hardware

Operating System Structure

- General-purpose OS is very large program
- **Various ways to structure OS are**
  - Simple structure – MS-DOS
  - Monolithic Structure – UNIX
  - Monolithic plus modular design – Linux
  - Layered – an abstraction
  - Microkernel – Mach

Operating System Structure – Simple structure-MS DOS



Advantages:

Base hardware can be accessed by all the things above it

Dis Advantages :

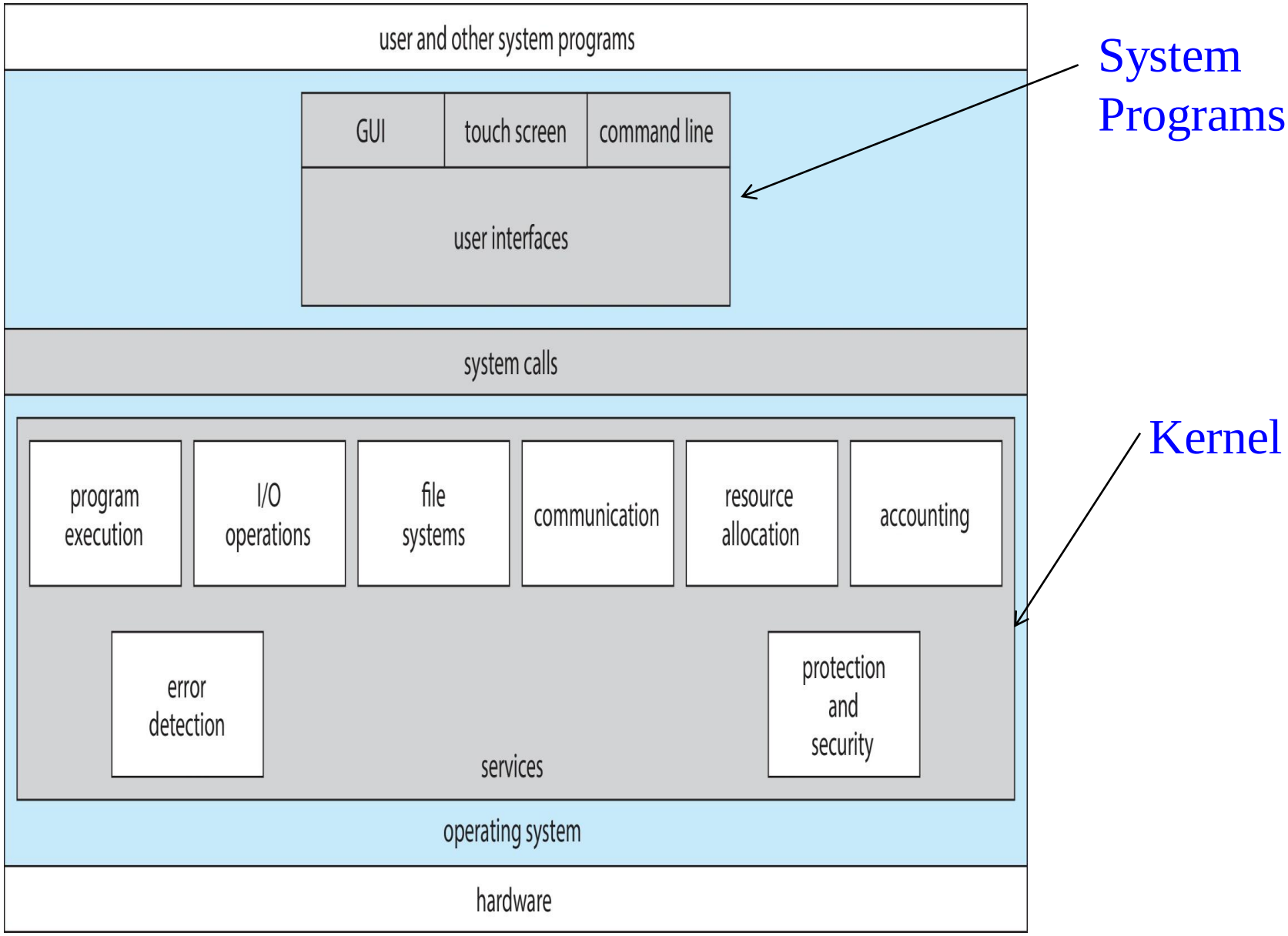
1. Vulnerable to error and malicious user programs causing the entire system to crash when the user program fails and it affects kernel of OS and hardware.
2. Look like layered structure but it not.
3. So, it is not protected, not well structured and not well defined

MS DOS was developed using Intel 8088, this does not provide dual mode or any hardware protection. So, the developers had no other option than leave the base hardware accessible by all the layers above

Operating System Structure –Monolithic Structure –UNIX

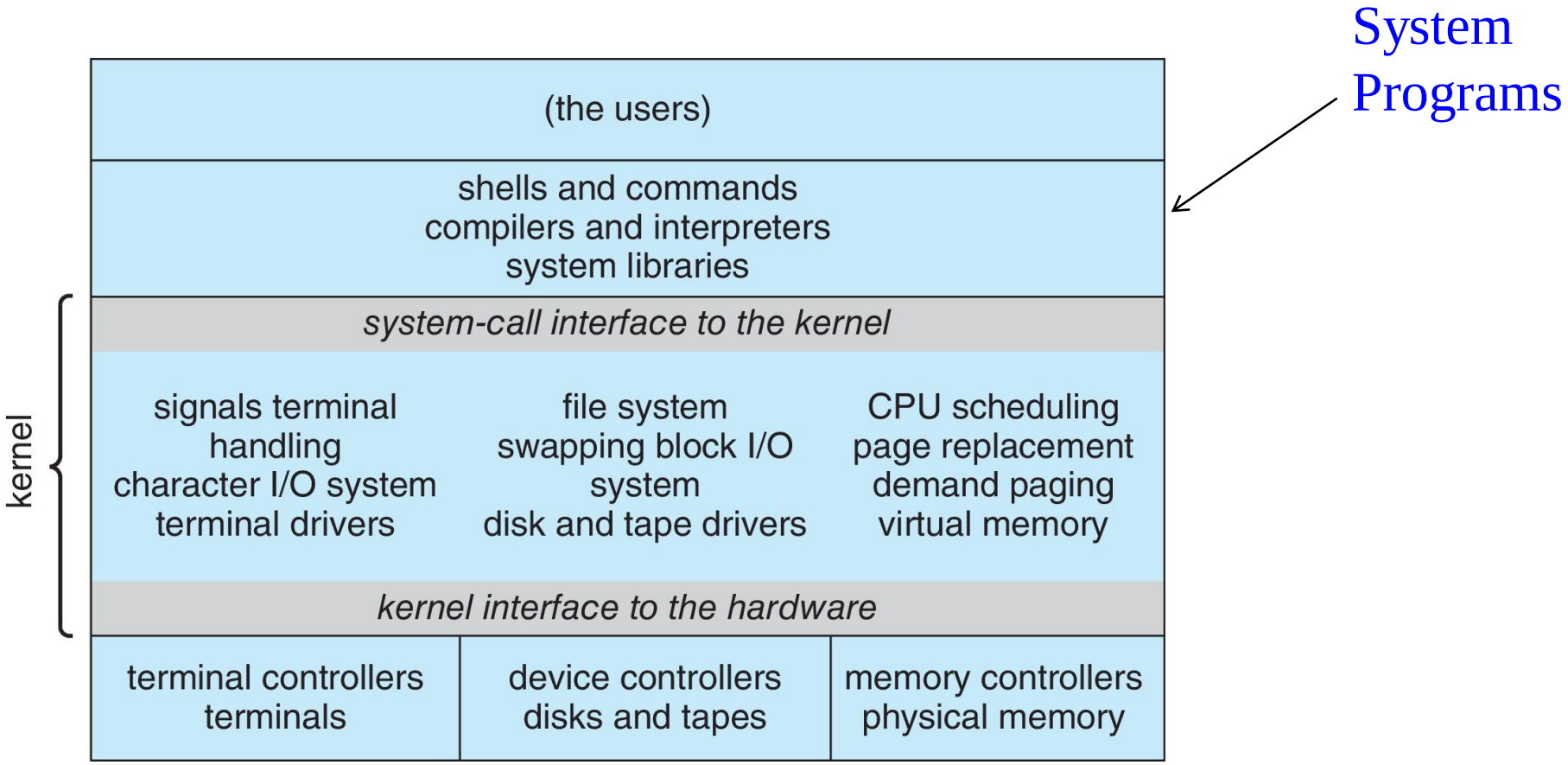
- UNIX – limited by hardware functionality, the original UNIX operating system had limited structuring.
- **Place all the functionality of the kernel into a single file** that runs in a single address space. This approach—known as a **monolithic** structure—is a common technique for designing operating systems.
- The UNIX OS consists of two separable parts
  - System programs
  - The kernel
    - Consists of everything below the system-call interface and above the physical hardware
    - Provides the file system, CPU scheduling, memory management, and other operating-system functions; a large number of functions for one level
    - Taken in sum, that is an enormous amount of functionality to be combined into one single address space.

A View of Operating System Services



Traditional UNIX System Structure

Beyond simple but not fully layered



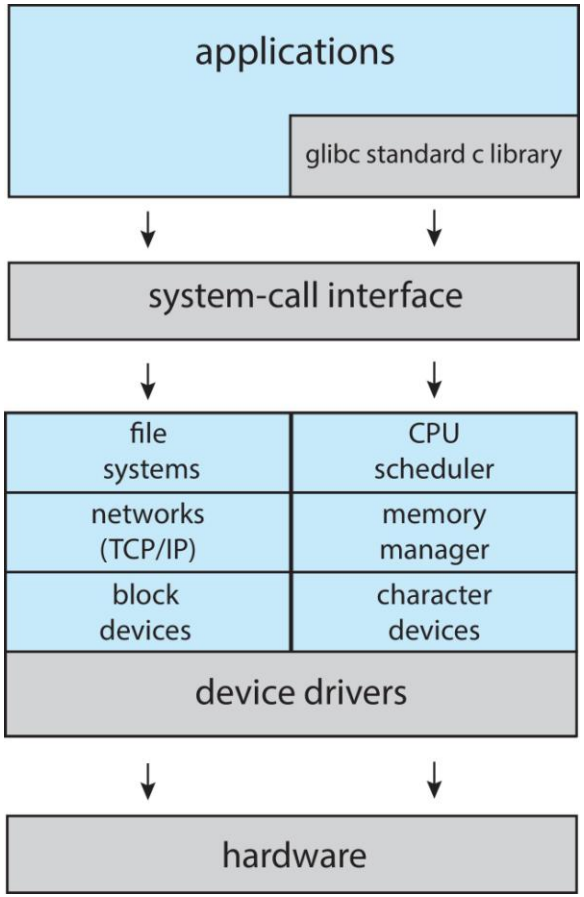
Operating System Structure –Monolithic Structure –UNIX

Advantages : Simple structure- one layered

Dis Advantages : All the functionalities are packed into one level called as kernel. Too many functions packed in one level and this makes implementation and maintenance very difficult. To want to add something, entire kernel have to change or modify. [ex]: A problem in CPU scheduling , in order to fix the problem the entire kernel is touched. So, the implementation and maintenance is very difficult

Linux System Structure

Monolithic plus modular design



- Applications typically use the glibc standard C library when communicating with the system call interface to the kernel.
- The Linux kernel is monolithic in that it runs entirely in kernel mode in a single address space, it does have a modular design that allows the kernel to be modified during run time.
- Despite the apparent simplicity of monolithic kernels, they are difficult to implement and extend.
- **Monolithic kernels do have a distinct performance advantage, there is very little overhead in the system-call interface, and communication within the kernel is fast.**
- Therefore, despite the drawbacks of monolithic kernels, their speed and efficiency explains the evidence of using this structure in the UNIX, Linux, and Windows operating systems.

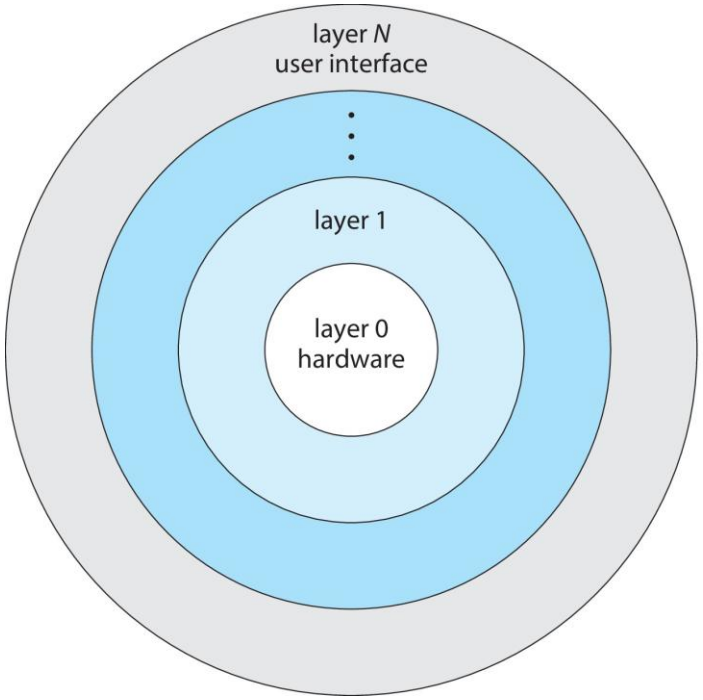
Layered Approach

- 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.

- Alternatively, a **loosely coupled** system is designed. Such a system is divided into separate, smaller components that have specific and limited functionality. All these components together comprise the kernel. 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.

Layered Approach

- The operating system is divided into a number of layers (levels), each built on top of lower layers. The bottom layer (layer 0), is the hardware; the highest (layer N) is the user interface.
- A typical operating-system layer—say, layer *M*—consists of data structures and a set of functions that can be invoked by higher-level layers. Layer *M*, in turn, can invoke operations on lower-level layers.
- With modularity, layers are selected such that each uses functions (operations) and services of only lower-level layers.



Layered Approach

•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. This approach simplifies debugging and system verification.

•The first layer can be debugged without any concern for the rest of the system, because, by definition, it uses only the basic hardware (which is assumed correct) to implement its functions. Once the first layer is debugged, its correct functioning can be assumed while the second layer is debugged, and so on.

•**If an error is found during the debugging of a particular layer, the error must be on that layer**, because the layers below it are already debugged. Thus, the design and implementation of the system are simplified.

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

Layered systems have been successfully used in computer networks (such as TCP/IP) and web applications.

Microkernels

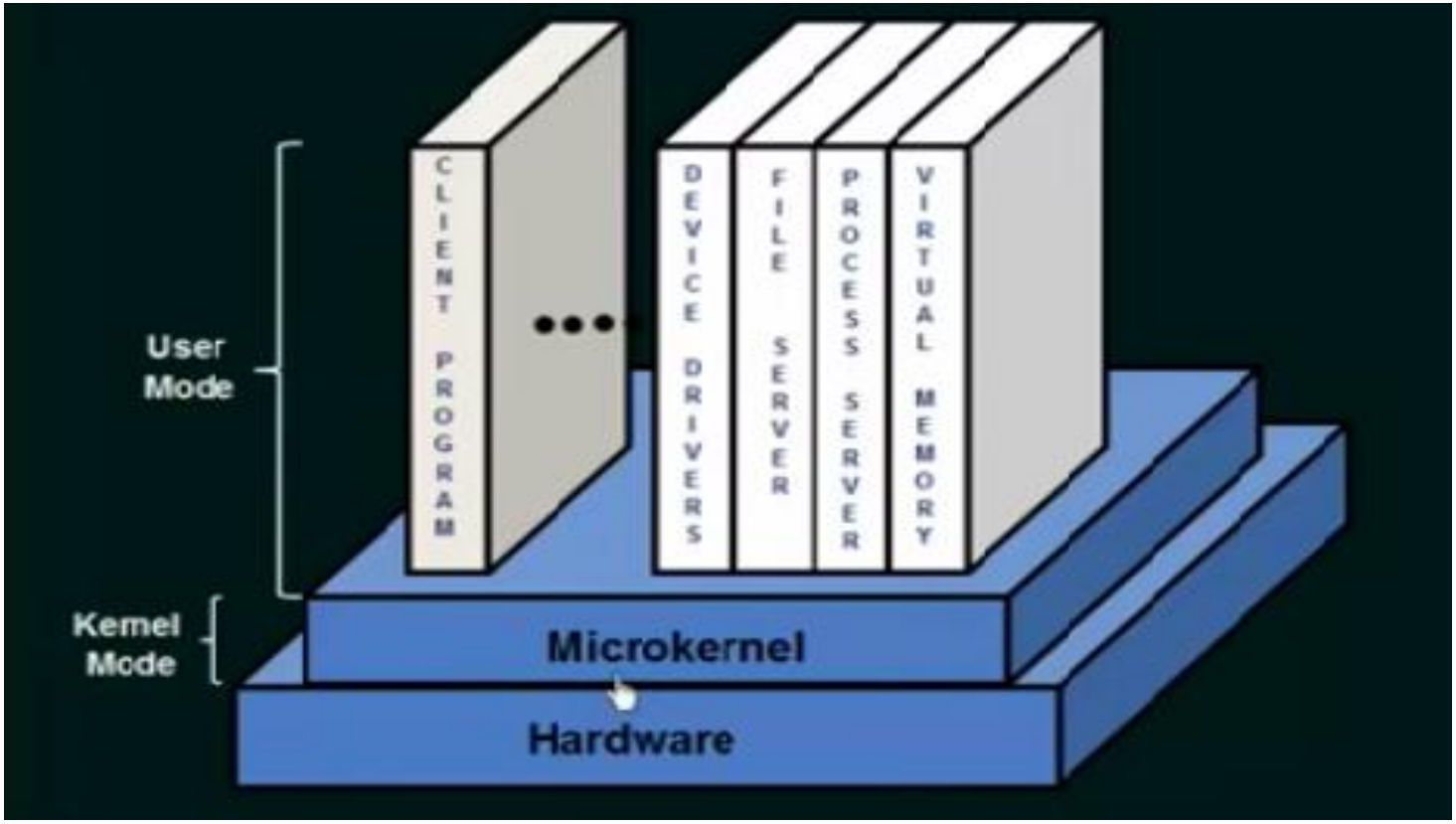
- The original UNIX system had a monolithic structure. As UNIX expanded, the kernel became large and difficult to manage.
- In the mid-1980s, researchers at Carnegie Mellon University developed an operating system called **Mach** that modularized the kernel using the **micro- kernel** approach.
- This method structures the operating system by removing all non essential components from the kernel and implementing them as user- level programs that reside in separate address spaces. The result is a smaller kernel.
- There is little consensus regarding which services should remain in the kernel and which should be implemented in user space.
- Typically, however, microkernels provide minimal process and memory management, in addition to a communication facility.

Microkernels

- 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. **Communication is provided through message passing**.
- For example, if the client program wishes to access a file, it must interact with the file server. The client program and service never interact directly. Rather, they communicate indirectly by exchanging messages with the microkernel.

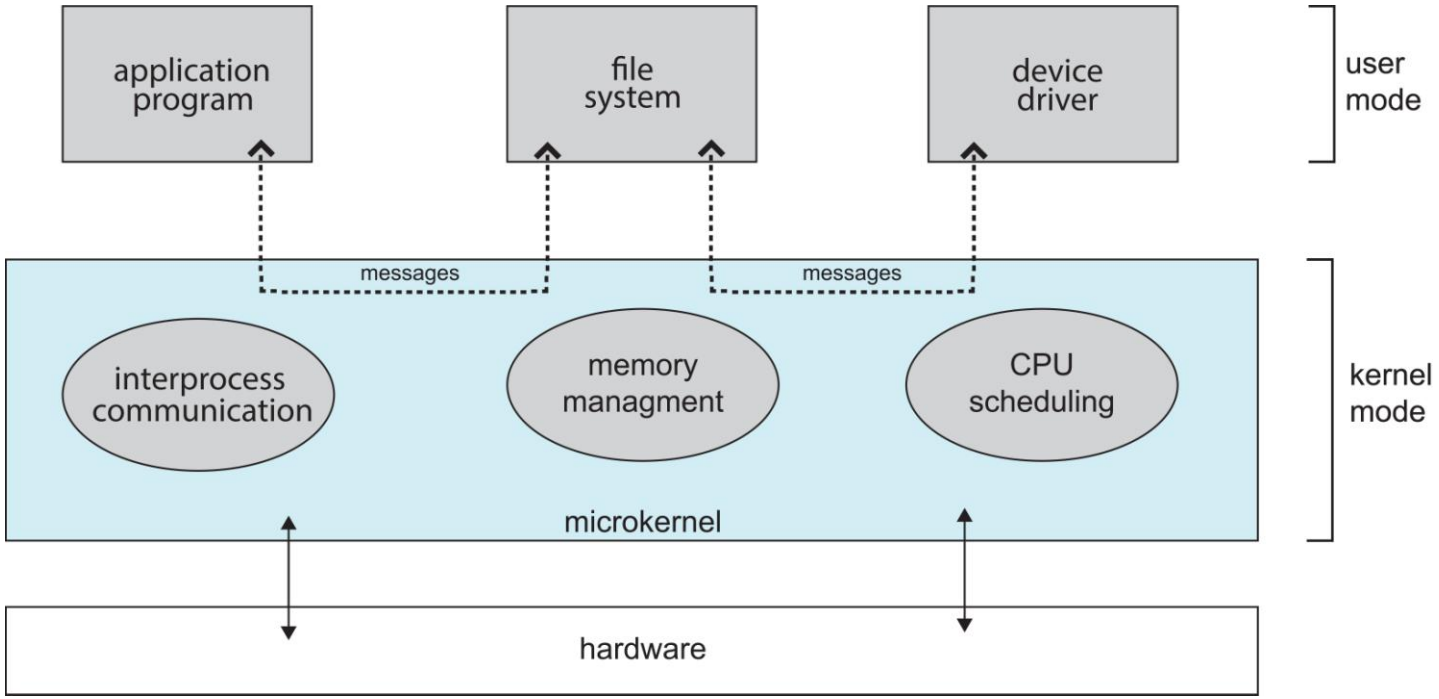
- Benefits:
  - Easier to extend a microkernel
  - Easier to port the operating system to new architectures
  - More reliable (less code is running in kernel mode)
  - More secure
- Detriments:
  - Performance overhead of user space to kernel space communication

Microkernels



The best-known illustration of a micro kernel operating system is ***Darwin***, the kernel component of the macOS and iOS operating systems. Darwin, in fact, consists of two kernels, one of which is the Mach microkernel and BSD UNIX kernel.

Microkernel System Structure



Modules

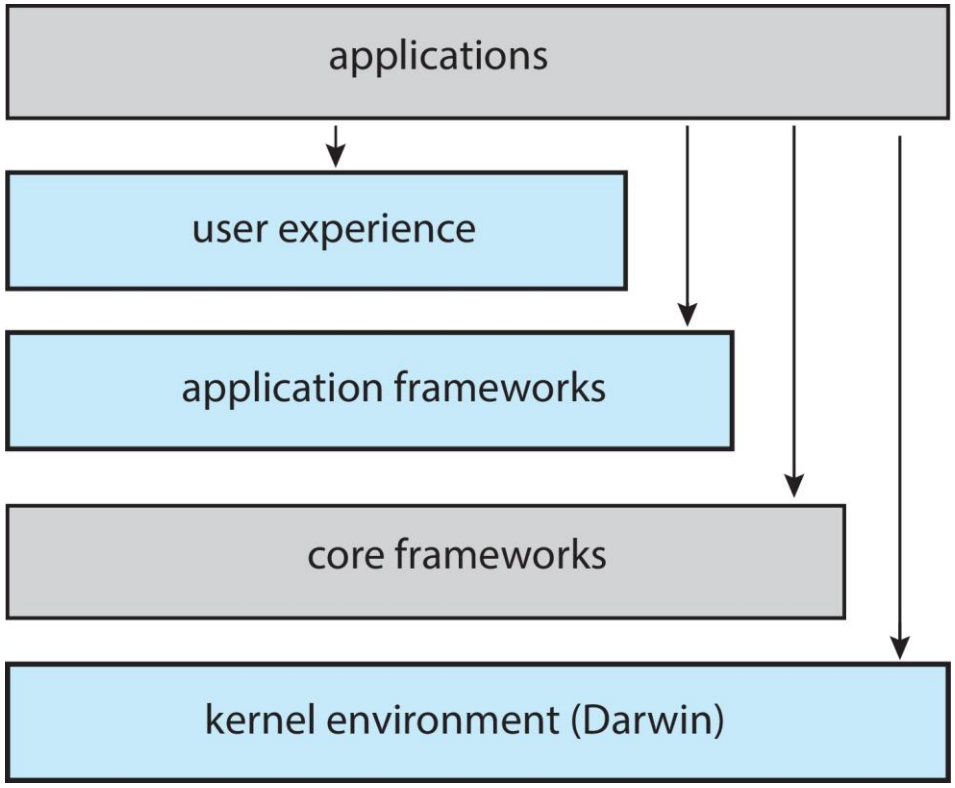
- Many modern operating systems implement **loadable kernel modules (LKMs).** Here, the kernel has a set of core components and can link in additional services via modules, either at boot time or during run time. This type of design is common in modern implementations of UNIX, such as Linux, macOS, and Solaris, as well as Windows.
  - Uses object-oriented approach
  - Each core component(module) is separate
  - Each talks to the others over known interfaces
  - Each is loadable as needed within the kernel

  Linux uses loadable kernel modules, primarily for supporting device drivers and file systems. LKMs can be “inserted” into the kernel as the system is started (or ***booted***) or during run time, such as **when a USB device is plugged into a running machine**.

Hybrid Systems

- Most modern operating systems are not one pure model
  - Hybrid combines multiple approaches to address performance, security, usability needs
  - **Linux and Solaris** kernels are **monolithic plus modular** for dynamic loading of functionality
  - **Windows** mostly **monolithic plus microkernel**. Windows systems also provide support for **dynamically loadable kernel modules.**
- **Apple Mac OS X hybrid**, layered, **Aqua** UI plus **Cocoa** programming environment
  - Below is Darwin kernel consisting of Mach microkernel and BSD Unix parts, plus I/O kit and dynamically loadable modules (called **kernel extensions**)

General architecture of macOS and iOS Structure



- **User experience layer:** This layer defines the software interface that allows users to interact with the computing devices. macOS uses the ***Aqua*** user interface, which is designed for a mouse or trackpad, whereas iOS uses the ***Springboard*** user interface, which is designed for touch devices.

- **Application frameworks layer:** This layer includes the ***Cocoa*** and ***Cocoa Touch*** frameworks, which provide an API for the programming languages. The primary difference between Cocoa and Cocoa Touch is, Cocoa is used for developing macOS applications, and the Cocoa Touch is used in iOS to provide support for hardware features unique to mobile devices, such as touch screens.

- **Core frameworks:** This layer defines frameworks that support graphics and media including, Quicktime and OpenGL.

- **Kernel environment:** This environment, also known as **Darwin**, includes the Mach microkernel and the BSD UNIX kernel.

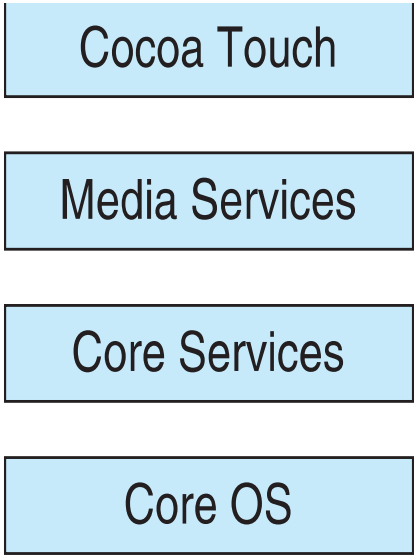
Some significant distinctions between macOS and iOS include the following:

•Because macOS is intended for desktop and laptop computer systems, it is compiled to run on Intel architectures. iOS is designed for mobile devices and thus is compiled for ARM-based architectures. Similarly, the iOS kernel has been modified somewhat to address specific features and needs of mobile systems, such as power management and aggressive memory management. Additionally, iOS has more stringent security settings than macOS.

•The iOS operating system is generally much more restricted to developers than macOS and may even be closed to developers. For example, iOS restricts access to POSIX and BSD APIs on iOS, whereas they are openly available to developers on macOS.

iOS

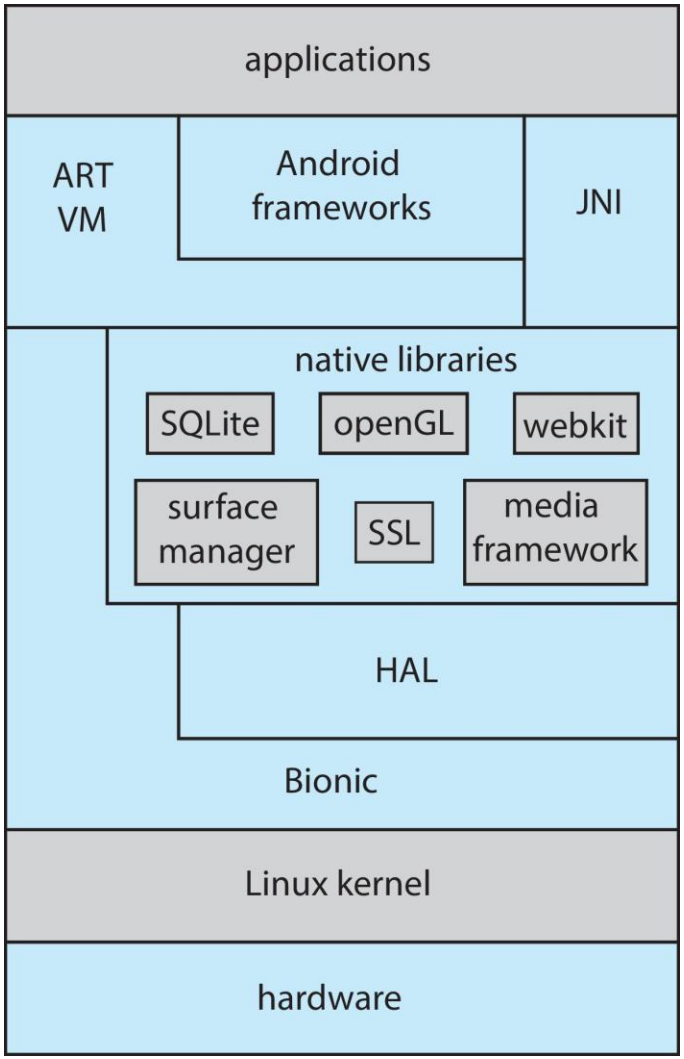
- Apple mobile OS for ***iPhone***, ***iPad***
  - Structured on Mac OS X, added functionality
  - Does not run OS X applications natively
    - Also runs on different CPU architecture (ARM vs. Intel)
  - **Cocoa Touch** Objective-C API for developing apps
  - **Media services** layer for graphics, audio, video
  - **Core services** provides cloud computing, databases
  - Core operating system, based on Mac OS X kernel



Android

- Developed by Open Handset Alliance (mostly Google)
  - Open Source
- Similar stack to iOS
- Based on Linux kernel but modified
  - Provides process, memory, device-driver management
  - Adds power management
- Runtime environment includes core set of libraries and Dalvik virtual machine
  - Apps developed in Java plus Android API
    - Java class files compiled to Java bytecode then translated to executable then runs in Dalvik VM
- Libraries include frameworks for web browser (webkit), database (SQLite), multimedia, smaller libc

Android Architecture



Building and Booting an Operating System

- Operating systems generally designed to run on a class of systems with variety of peripherals
- Commonly, operating system already installed on purchased computer
  - But can build and install some other operating systems
  - If generating an operating system from scratch
    - Write the operating system source code
    - Configure the operating system for the system on which it will run
    - Compile the operating system
    - Install the operating system
    - Boot the computer and its new operating system

Building and Booting Linux

- Download Linux source code (http://www.kernel.org)
- Configure kernel via “make menuconfig”
- Compile the kernel using “make”
  - Produces vmlinuz, the kernel image
  - Compile kernel modules via “make modules”
  - Install kernel modules into vmlinuz via “make modules_install”
  - Install new kernel on the system via “make install”

System Boot

- When power initialized on system, execution starts at a fixed memory location
- Operating system must be made available to hardware so hardware can start it
  - Small piece of code – **bootstrap loader, BIOS, stored in ROM or EEPROM locates the kernel, loads it into memory, and starts it**
  - Sometimes **two-step** process where **boot block at fixed location loaded by ROM code, which loads bootstrap loader from disk**
  - **Modern systems replace BIOS with Unified Extensible Firmware Interface (UEFI)**
- Common **bootstrap loader, GRUB**, allows selection of kernel from multiple disks, versions, kernel options
- **Kernel loads and system is then running**
- Boot loaders frequently allow various boot states, such as single user mode

Operating-System Debugging

- **Debugging** is finding and fixing errors, or **bugs**
- Also **performance tuning**
- OS generate **log files** containing error information
- **Failure of an application** can generate **core dump** file capturing memory of the process
- **Operating system failure** can generate **crash dump** file containing kernel memory
- Beyond crashes**, performance tuning can optimize system performance**
  - Sometimes using ***trace listings* of activities**, recorded for analysis
  - **Profiling** is periodic sampling of instruction pointer to look for statistical trends

Kernighan's Law: "Debugging is twice as hard as writing the code in the first place. Therefore, if you write the code as cleverly as possible, you are, by definition, not smart enough to debug it."