

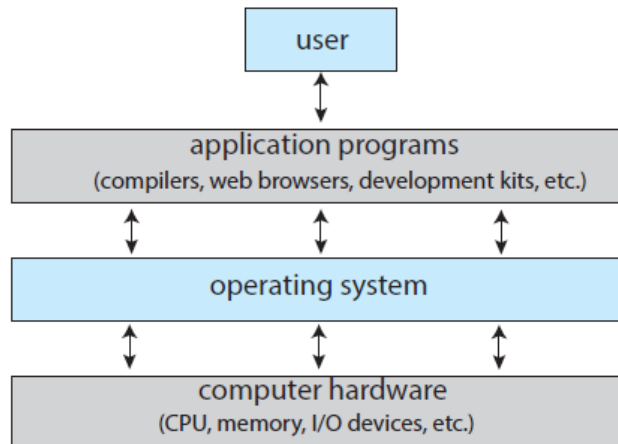
## Unit – I

### Operating Systems Overview

**Introduction – Computer System Organization – Computer System Architecture – Operations – Resource Management – Security and Protection – Virtualization – Computing Environments. Operating Systems Structures: Services – User and OS Interface – System Calls.**

#### Introduction

A computer system can be divided roughly into four components: the *hardware*, the *operating system*, the *application programs*, and a *user*.



- **The hardware** - the central processing unit (CPU), the memory, and the input/output (I/O) devices - basic computing resources for the system.
- **The application programs** - such as word processors, spreadsheets, compilers, and web browsers - through which these resources are used to solve users' computing problems.
- **The operating system controls the hardware and coordinates among the various application programs.**

#### User View

**The user's view of the computer varies according to the interface being used.**

Users typically interact with computers through interfaces like laptops, PCs with monitors, keyboards, and mice, or mobile devices with touchscreens and voice recognition.

In this case, the operating system is designed mostly for ease of use.

Many users interact with mobile devices such as smartphones and tablets.

These devices are connected to networks through cellular or other wireless technologies.

The user interface for mobile computers are: a touch screen, where the user interacts with the system by pressing and swiping fingers across the screen rather than using a physical keyboard and mouse.

Many mobile devices also allow users to interact through a voice recognition interface, such as Apple's Siri.

Some computers have little or no user view.

Example: Embedded computers in home devices and automobiles may have numeric keypads and may turn indicator lights on or off to show status. These operating systems and applications are designed to run without user intervention.

### System View

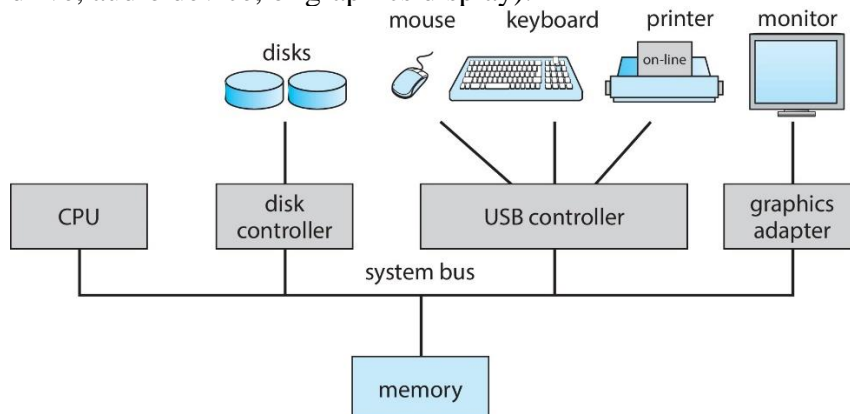
- From a computer's perspective, the operating system (OS) manages resources like CPU, memory, and storage.
- It acts as a resource allocator to handle competing demands efficiently.
- It also works as a control program, managing user programs and I/O devices to prevent errors.
- An operating system is a control program.
- A control program manages the execution of user programs to prevent errors and improper use of the computer. It is concerned with the operation and control of I/O devices.
- An OS makes computers usable by providing common functions needed by applications (like controlling I/O).
- The OS includes the **kernel**, which runs, **system programs** (supporting the OS) and **application programs** (unrelated to the OS).
- Early computers evolved from single-purpose machines to general-purpose systems requiring OS management.
- Moore's Law predicted increasing transistor counts, enabling more powerful, smaller systems with diverse OS types.
- Mobile OS like iOS and Android include kernels and **middleware** for added features like multimedia and graphics.

An OS typically consists of:

- The kernel, running continuously.
- Middleware for extra developer tools.
- System programs to manage the system.

### Computer System Organization

- A general-purpose computer system consists of one or more CPUs and a number of device controllers connected through a common **bus** that provides access between components and shared memory.
- Each device controller is in charge of a specific type of device (for example, a disk drive, audio device, or graphics display).



**A typical PC computer system**

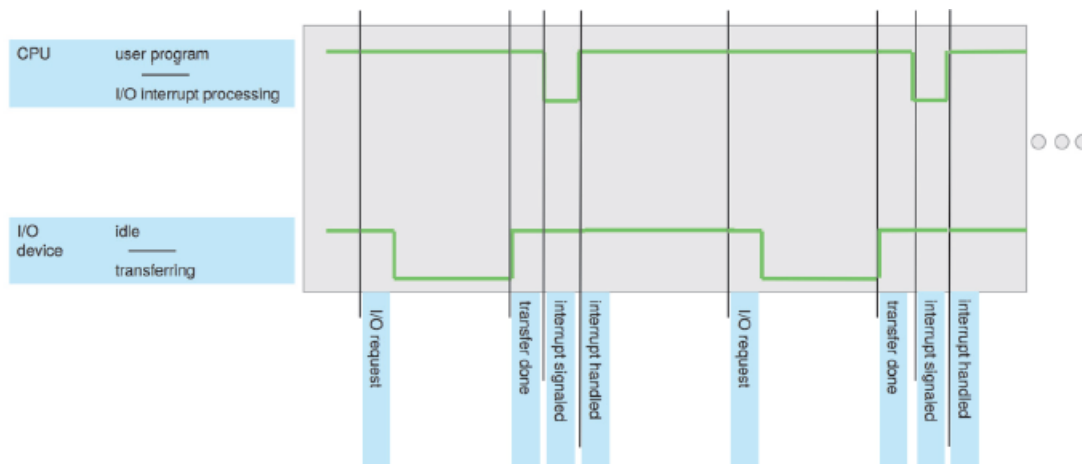
- A device controller maintains some local buffer storage and a set of special-purpose registers.
- The device controller is responsible for moving the data between the peripheral devices that it controls and its local buffer storage.
- Operating systems have a device driver for each device controller.
- Device driver provides the rest of the operating system with a uniform interface to the device.
- The CPU and the device controllers can execute in parallel, competing for memory cycles.
- To ensure orderly access to the shared memory, a memory controller synchronizes access to the memory.
- Device controller informs CPU that it has finished its operation by causing an **interrupt**.

### **Interrupt**

- Consider a computer operation: a program performing I/O.
- To start an I/O operation, the device driver loads the appropriate registers in the device controller.
- The device controller examines the contents of these registers to determine what action to take (such as “read a character from the keyboard”).
- 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 that it has finished its operation.
- Interrupt transfers control to the interrupt service routine, 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**.
- The device driver then gives control to other parts of the operating system.

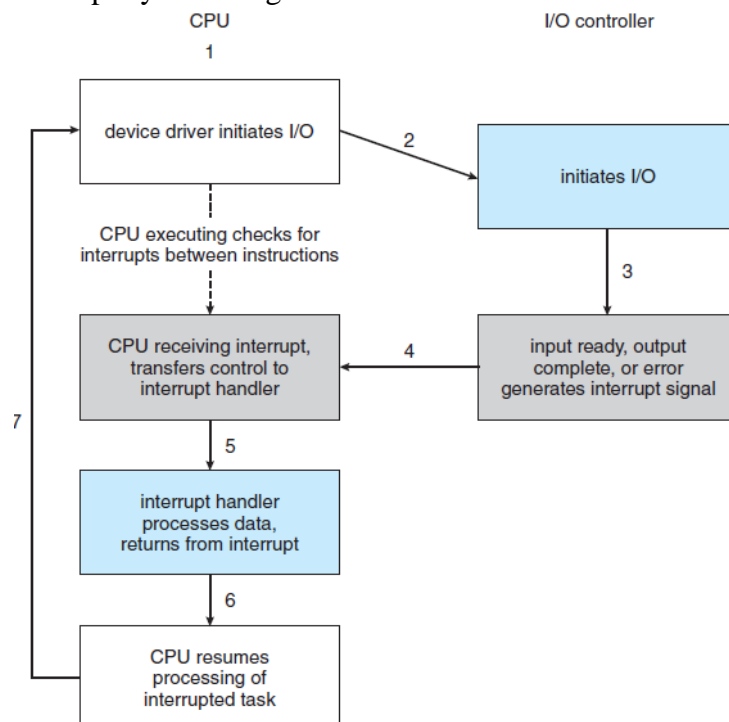
### Overview

- Hardware may trigger an interrupt at any time by sending a signal to the CPU, usually by the system bus. (There may be many buses within a computer system, but the system bus is the main communications path between the major components.)
- Interrupts are used for many other purposes and are a key part of how operating systems and hardware interact.
- 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 for the interrupt is located.
- The interrupt service routine executes; on completion, the CPU resumes the interrupted computation.
- Interrupts are an important part of a computer architecture. Each computer design has its own interrupt mechanism.



### 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**.
- It then starts execution at the address associated with that index.
- CPU *catches* the interrupt and *dispatches* it to the interrupt handler, and the handler *clears* the interrupt by servicing the device.



In a modern operating system, we need more sophisticated interrupt handling features.

1. We need the ability to defer interrupt handling during critical processing.
2. We need an efficient way to dispatch to the proper interrupt handler for a device.
3. We need multilevel interrupts, so that the operating system can distinguish between high- and low-priority interrupts and can respond with the appropriate degree of urgency.

- These three features are provided by the CPU and the **interrupt-controller hardware**.

- Most CPUs have two interrupt request lines.
  1. **Nonmaskable interrupt** - reserved for events such as unrecoverable memory errors.
  2. **Maskable** - it can be turned off by the CPU before the execution of critical instruction sequences that must not be interrupted. It is used by device controllers to request service.
- Vectored interrupts reduce the need for a single handler to search all possible interrupt sources.
- However, since there are often more devices than interrupt vector entries, **interrupt chaining** is used. In this method, each vector entry points to a list of handlers.
- When an interrupt occurs, the handlers are checked one by one until the correct one is found.
- The interrupt mechanism implements a system of **interrupt priority levels**. These levels enable the CPU to defer the handling of low-priority interrupts without masking all interrupts and makes it possible for a high-priority interrupt to preempt the execution of a low-priority interrupt.

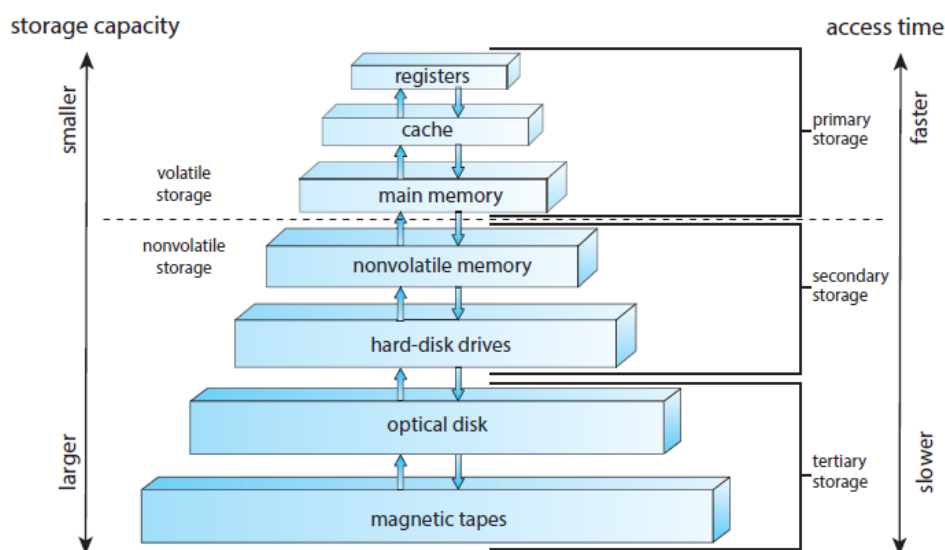
### Storage Structure

- The CPU loads instructions only from memory, so programs must be placed in memory to run.
- Most computers use **main memory** (RAM), typically dynamic random-access memory (DRAM), for general-purpose operations.
- However, since RAM is **volatile** and loses data when powered off, a **bootstrap program** stored in nonvolatile memory like EEPROM initializes the system at startup.
- Memory interacts with the CPU through **load** and **store** instructions, moving data between memory and CPU registers.
- The CPU continuously fetches, decodes, and executes instructions from memory, managed by a **program counter**.
- Memory itself handles addresses without distinguishing between data and instructions.
- Main memory has limitations:
  1. **It's too small to hold all programs and data.**
  2. **It's volatile, requiring secondary storage for long-term data storage.**
- Common secondary storage devices include **hard-disk drives (HDDs)** and **nonvolatile memory (NVM)**, like flash storage, which is slower than RAM but offers higher capacity.
- Large storage systems may also use **tertiary storage**, like magnetic tapes and optical disks, for backup and archival purposes.
- Storage systems differ in speed, size, and volatility, forming a **hierarchy**.
- **Memory** refers to volatile storage like RAM.
- **Nonvolatile Storage (NVS)** includes secondary storage, classified as:
  - ✓ **Mechanical** (e.g., HDDs, optical disks).
  - ✓ **Electrical (NVM)** (e.g., flash memory, SSDs).
- Mechanical storage is larger and cheaper per byte, while electrical storage is faster and more expensive.

The basic unit of computer storage is the **bit**. A bit can contain one of two values, 0 and 1. All other storage in a computer is based on collections of bits.

Given enough bits, it is amazing how many things a computer can represent: numbers, letters, images, movies, sounds, documents, and programs, to name a few. A **byte** is 8 bits, and on most computers it is the smallest convenient chunk of storage. For example, most computers don't have an instruction to move a bit but do have one to move a byte. A less common term is **word**, which is a given computer architecture's native unit of data. A word is made up of one or more bytes. For example, a computer that has 64-bit registers and 64-bit memory addressing typically has 64-bit (8-byte) words. A computer executes many operations in its native word size rather than a byte at a time.

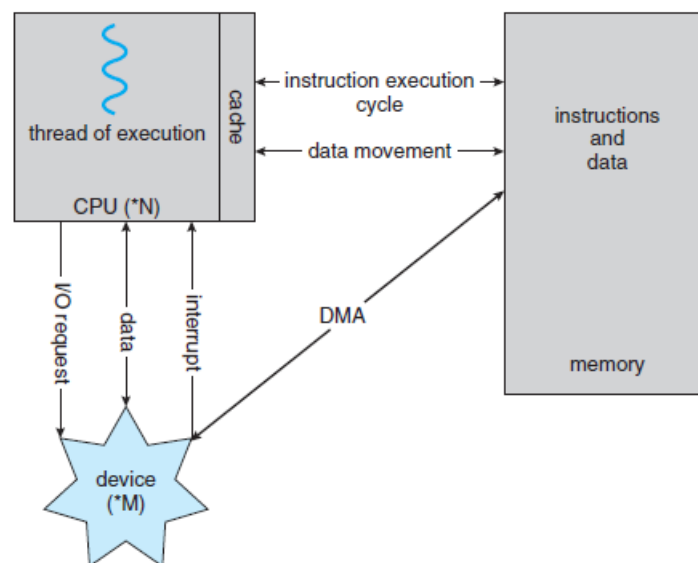
Computer storage, along with most computer throughput, is generally measured and manipulated in bytes and collections of bytes. A **kilobyte**, or **KB**, is 1,024 bytes; a **megabyte**, or **MB**, is  $1,024^2$  bytes; a **gigabyte**, or **GB**, is  $1,024^3$  bytes; a **terabyte**, or **TB**, is  $1,024^4$  bytes; and a **petabyte**, or **PB**, is  $1,024^5$  bytes. Computer manufacturers often round off these numbers and say that a megabyte is 1 million bytes and a gigabyte is 1 billion bytes. Networking measurements are an exception to this general rule; they are given in bits (because networks move data a bit at a time).



**Storage-device hierarchy**

## I/O Structure

- Operating system manages **input/output (I/O)** to ensure system performance and reliability.
- In general-purpose computers, devices share a common **bus** for data exchange. For small data transfers, **interrupt-driven I/O** works well, but it creates high overhead for large data transfers.
- To handle bulk data efficiently, **direct memory access (DMA)** is used. With DMA, the device controller moves entire data blocks directly between main memory and the device without CPU involvement, generating only **one interrupt per block** instead of one per byte. This frees the CPU for other tasks while the transfer occurs.
- Some advanced systems use a **switch architecture** instead of a shared bus, allowing simultaneous communication between components. In such systems, DMA is even more efficient.



### How a modern computer system works

#### Computer System Architecture

A computer system can be organized in a number of ways, which can be categorized according to the number of processors used.

#### Single-Processor Systems

A **single-processor system** has one CPU with a single core that executes instructions and uses registers for local data storage. This core runs general-purpose instruction sets, including process instructions.

These systems also include **special-purpose processors** like disk, keyboard, and graphics controllers. These processors handle specific tasks with a limited instruction set and do not run processes.

The operating system may manage some of these processors by assigning tasks and monitoring their status. For instance, a disk controller uses its own scheduling to reduce CPU overhead, and a keyboard microprocessor converts keystrokes into codes for the CPU.

Other special-purpose processors operate autonomously, without direct OS control. Despite using these specialized microprocessors, having only one general-purpose CPU makes the system a **single-processor system**, though true single-processor systems are rare in modern computing.

## Multiprocessor Systems

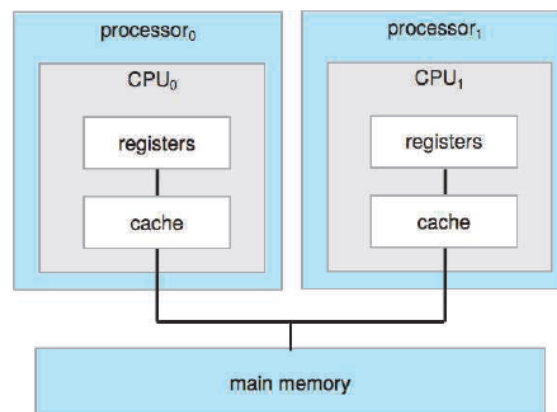
Modern computers, from mobile devices to servers, commonly use **multiprocessor systems**. These systems typically have two or more processors, each with a single-core CPU, sharing memory, the system bus, and sometimes the clock and peripherals.

**Key benefits** include increased throughput—more work completed in less time. However, adding processors does not provide a linear speed increase due to overhead from coordination and resource contention.

The most common model is **symmetric multiprocessing (SMP)**, where all processors are peers, handling both operating system and user tasks. Each CPU has its own registers and cache but shares physical memory. SMP allows multiple processes to run simultaneously, but inefficiencies can occur if load balancing isn't optimized.

The term **multiprocessor** now includes **multicore systems**, where multiple cores exist on a single chip. Multicore systems are more efficient because communication between cores on the same chip is faster than between separate chips.

One chip with multiple cores uses significantly less power than multiple single-core chips.

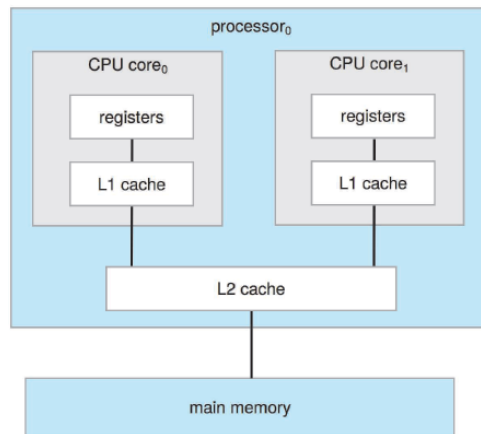


**Symmetric multiprocessing architecture**

The above figure is a SMP architecture with two processors, each with its own CPU. Each CPU processor has its own set of registers, as well as a private or local cache. However, all processors share physical memory over the system bus.

The figure below shows a dual-core design with two cores on the same processor chip. In this design, each core has its own register set, as well as its own local cache, often known as a level 1, or L1, cache. A 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.





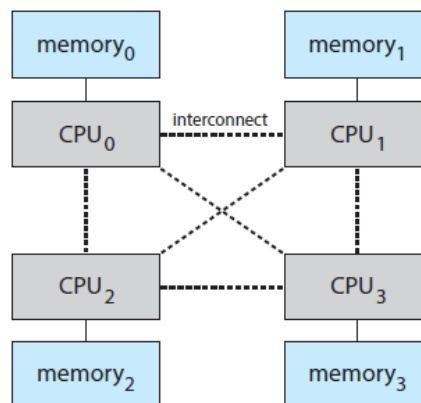
### A dual-core design with two cores on the same chip

A multicore processor with  $N$  cores appears to the operating system as  $N$  standard CPUs. Modern operating systems like Windows, macOS, Linux, Android, and iOS support **multicore symmetric multiprocessing (SMP)**.

Adding more CPUs to a multiprocessor system increases power but faces scaling limits due to **bus contention**, which slows performance.

A better solution is **non-uniform memory access (NUMA)**, where each CPU (or group of CPUs) has its own **local memory** connected by a shared interconnect.

Accessing local memory is faster, reducing contention, but accessing remote memory introduces **latency**. Operating systems minimize this using smart CPU scheduling and memory management. NUMA scales better, making it popular for servers and high-performance systems.

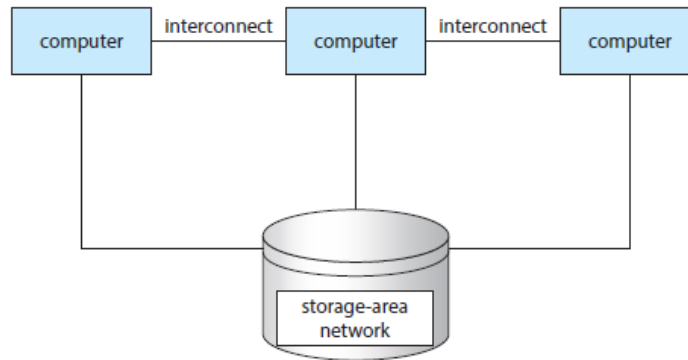


### NUMA multiprocessing architecture

**Blade servers** combine multiple processors, I/O, and networking boards in one chassis. Each blade can boot independently with its own operating system. Some blades are multiprocessor systems, blurring distinctions between server types.

### Clustered Systems

A **clustered system** connects multiple individual systems (or nodes), typically multicore, to work together.



**General structure of a clustered system**

Unlike tightly-coupled multiprocessor systems, clustered systems are **loosely coupled** and linked via a **local-area network (LAN)** or high-speed interconnects like **InfiniBand**.

Clustering offers **high availability**, where redundancy ensures continued service even if some nodes fail.

Cluster software monitors the nodes, transferring tasks to working nodes when failures occur, providing **graceful degradation** or even **fault tolerance** for uninterrupted operation.

Clusters can be **asymmetric** (a standby machine monitors an active one) or **symmetric** (all nodes run tasks and monitor each other). Symmetric clustering uses resources more efficiently but requires multiple applications.

Clusters also support **high-performance computing** by parallelizing applications to run across nodes. Parallel clusters enable simultaneous data access by multiple nodes, requiring specialized software (e.g., Oracle Real Application Cluster) with mechanisms like a **distributed lock manager (DLM)** to prevent conflicts.

**Storage-area networks (SANs)** allow flexible data access, enhancing performance and reliability by enabling task reassignment to any node if one fails.

### **Operating-System Operations**

An **operating system (OS)** creates an environment where programs run on a computer.

For 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, or bootstrap program, tends to be simple. This program is stored in **firmware** and initializes the system (CPU, memory, and devices). Its main job is to load the **operating system kernel** into memory and start it.

Once the kernel is loaded and executing, it can start providing services to the system and its users. The **kernel** is the core of the OS.

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.

If there are no processes to execute, no I/O devices to service, and no users to whom to respond, an operating system will sit quietly, waiting for something to happen. Events are almost always signaled by the occurrence of an interrupt.

**Hardware interrupts** occur from devices needing attention.

**Software interrupts (traps)** happen due to errors (e.g., division by zero or invalid memory access) or requests for OS services (using **system calls**).

## Multiprogramming and Multitasking

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.

Users *want* to run more than one program at a time. **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**.

In a **multiprogrammed system (Batch System)**, multiple processes (programs in execution) are kept in memory at once.

The operating system chooses one process to run. If that process has to wait (e.g., for I/O), the OS switches to another process.

This keeps the CPU busy, as it always has a process to run.

A real-life example is a lawyer managing multiple cases at once — working on one case while waiting for progress on another.

**Multitasking (Timesharing)** is an advanced form of multiprogramming where the CPU switches between processes very frequently.

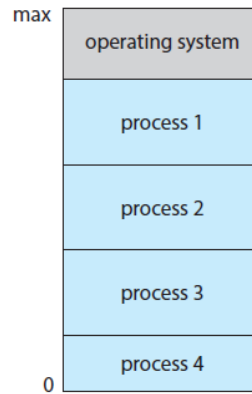
This frequent switching gives the user fast responses, even for interactive tasks like typing or using a mouse, which are slow by computer standards.

Managing several processes at once requires **memory management**. When multiple processes are ready to run, the OS must decide which one to run next, called **CPU scheduling**.

The OS must also prevent processes from interfering with one another, ensuring safety and fairness. A common method for doing so is **virtual memory**.

**Virtual memory** allows the OS to run processes larger than the available physical memory.

It abstracts physical memory into a large, unified space, separating how users view memory from how it's physically managed.



**Memory layout for a multiprogramming system**

### Dual-Mode and Multimode Operation

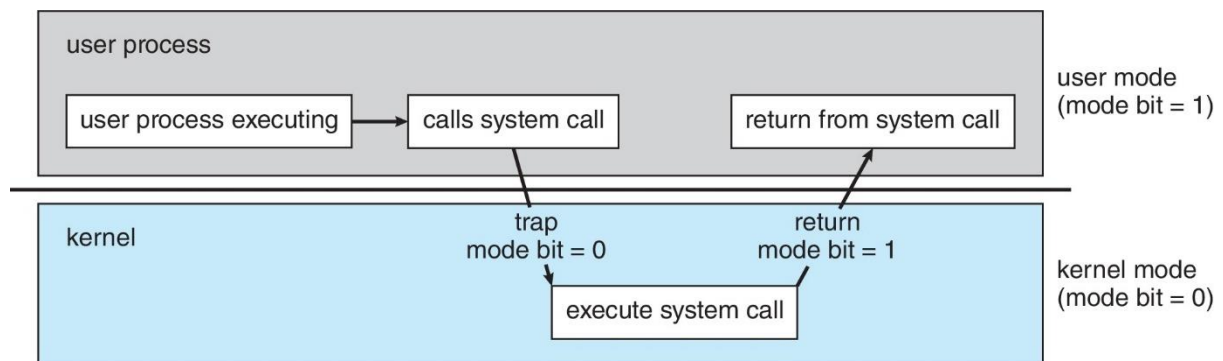
The operating system and its users share the hardware and software resources of the computer system, a properly designed operating system must ensure that an incorrect (or malicious) program cannot cause other programs—or the operating system itself—to execute incorrectly.

In order to ensure the proper execution of the system, we must be able to distinguish between the execution of operating-system code and user-defined code.

we need two separate *modes* of operation: **user mode** and **kernel mode** (also called **supervisor mode**, **system mode**, or **privileged mode**).

A bit, called the **mode bit**, is added to the hardware of the computer to indicate the current mode: kernel (0) or user (1). With the mode bit, we can distinguish between a task that is executed on behalf of the operating system or on behalf of the user.

When a user application requests a service from the operating system (via a system call), the system must transition from user to kernel mode to fulfil the request.



At system boot time, the hardware starts in kernel mode. The operating system is then loaded and starts user applications in user mode.

Whenever a trap or interrupt occurs, the hardware switches from user mode to kernel mode.

Some instructions (like I/O control and timer management) are **privileged** and can only run in kernel mode. If a user program tries to run a privileged instruction, it triggers a trap (error) that the OS handles.

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. (Although 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.

A **system call** allows user programs to request OS services. It works like a **software interrupt** that passes control to the OS. The OS verifies the request, performs the action, and returns control to the user program.

If a user program tries to do something illegal (like accessing memory outside its limits), the hardware traps the error and passes control to the OS. The OS terminates the program, shows an error message, and may create a **memory dump** for debugging.

### Timer

We must ensure that the operating system maintains control over the CPU. We cannot allow a user program to get stuck in an infinite loop or to fail to call system services and never return control to the operating system. To accomplish this goal, we can use a **timer**.

A timer can be set to interrupt the computer after a specified period. The period may be fixed (for example, 1/60 second) or variable (for example, from 1 millisecond to 1 second). A **variable timer** is generally implemented by a fixed-rate clock and a counter.

A **counter** linked to a clock decrement with each tick. When the counter reaches zero, an interrupt signals the OS.

Example: A timer with a 1-millisecond clock and a 10-bit counter can trigger interrupts between **1 ms and 1,024 ms**.

Before giving control to a user program, the OS sets the timer. If the timer interrupts, control returns to the OS, which can stop or extend the program's execution. Modifying the timer is a privileged instruction, restricted to the OS for security.

### Resource Management

An operating system is a **resource manager**. The system's CPU, memory space, file-storage space, and I/O devices are among the resources that the operating system must manage.

### Process Management

A process is a program in execution. A process needs certain resources—including **CPU time, memory, files, and I/O devices**—to accomplish its task.

In addition to the various physical and logical resources, various initialization data (input) may be passed along.

A program is a **passive** entity, like the contents of a file stored on disk, whereas a process is an **active** entity. A single-threaded process has one **program counter** specifying the next instruction to execute.

The execution of such a process must be sequential. The CPU executes one instruction of the process after another, until the process completes.

A multithreaded process has multiple program counters, each pointing to the next instruction to execute for a given thread.

A process is the **unit of work in a system**. A system consists of a collection of processes, some of which are operating-system processes (execute system code) and the rest of which are user processes (execute user code). All these processes can potentially execute concurrently—by multiplexing on a single CPU core—or in parallel across multiple CPU cores.

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

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

## Memory Management

The main memory is central to the operation of a modern computer system. Main memory is a large array of bytes, ranging in size from hundreds of thousands to billions. Each byte has its own address.

Main memory is a repository of quickly accessible data shared by the CPU and I/O devices. The CPU reads instructions from main memory during the instruction-fetch cycle and reads and writes data from main memory during the data-fetch cycle.

Example: for the CPU to process data from disk, those data must first be transferred to main memory. Instructions must be in memory for the CPU to execute them.

For a program to be executed, it must be mapped to absolute addresses and loaded into memory. As the program executes, it accesses program instructions and data from memory by generating these absolute addresses.

Eventually, the program terminates, its memory space is declared available, and the next program can be loaded and executed.

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

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

## File-System Management

To make the computer system convenient for users, the operating system provides a uniform, logical view of information storage.

The operating system abstracts from the physical properties of its storage devices to define a logical storage unit, the **file**.

The operating system maps files onto physical media and accesses these files via the storage devices.

Computers can store information on different types of physical media. Secondary storage is the most common, but tertiary storage is also possible.

Each medium is controlled by device (i.e., disk drive, tape drive) that has its own unique characteristics. Varying properties include access speed, capacity, data-transfer rate, access method (sequential or random).

A file is a collection of related information defined by its creator. Commonly, files represent programs (both source and object forms) and data. Data files may be numeric, alphabetic, alphanumeric, or binary.

The operating system implements the abstract concept of a file by managing mass storage media and the devices that control them.

Files are normally organized into directories to make them easier to use.

When multiple users have access to files, it may be desirable to control which user may access a file and how that user may access it (for example, read, write, append).

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

- Creating and deleting files
- Creating and deleting directories to organize files
- Supporting primitives for manipulating files and directories
- Mapping files onto mass storage
- Backing up files on stable (nonvolatile) storage media

## 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.

Hence, the proper management of secondary storage is important to a computer system. The entire speed of operation of a computer may hinge on the speeds of the secondary storage subsystem and the algorithms that manipulate that subsystem.

Tertiary storage is not crucial to system performance, but it still must be managed. Some operating systems take on this task, while others leave tertiary-storage management to application programs.

Some of the functions that operating systems can provide include mounting and unmounting media in devices, allocating and freeing the devices for exclusive use by processes, and migrating data from secondary to tertiary storage.

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

- Mounting and unmounting
- Free-space management
- Storage allocation
- Disk scheduling
- Partitioning
- Protection

### Cache Management

**Caching** is an important principle of computer systems.

Information is normally kept in some storage system (such as main memory). As it is used, it is copied into a faster storage system—the cache—on a temporary basis.

When we need a particular piece of information, we first check whether it is in the cache. If it is, we use the information directly from the cache. If not, data copied to cache and used there.

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 SRAM	CMOS SRAM	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

### Characteristics of various types of storage

In addition, internal programmable registers provide a high-speed cache for main memory. The programmer (or compiler) implements the register allocation and register-replacement algorithms to decide which information to keep in registers and which to keep in main memory.

Other caches are implemented totally in hardware. For instance, most systems have an instruction cache to hold the instructions expected to be executed next. Without this cache, the CPU would have to wait several cycles.



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

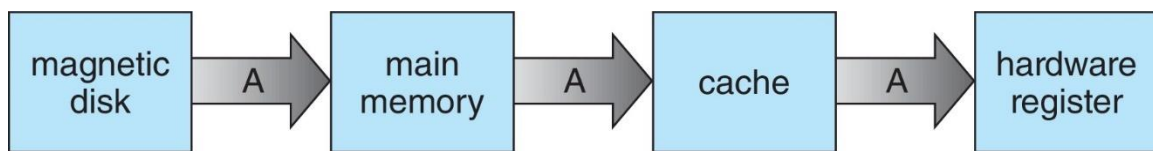
In a **storage hierarchy**, data can exist at multiple levels (e.g., disk, memory, cache, registers). The system must ensure data consistency when it's modified at one level but not yet updated in others.

### Example

To increase a number **A** stored on a disk:

- **A** is copied from the disk to memory, cache, and a CPU register.
- **A** is incremented in the register.
- **A** must be written back to the disk to make all copies consistent.

In **multiprocessor systems**, each CPU has its own cache. When one CPU updates **A**, other CPUs must see the change immediately. Hardware ensures this **cache coherency**.



**Migration of integer A from disk to register**

In **distributed systems**, data might exist on multiple computers. Synchronizing updates to all copies is a bigger challenge, managed by the system to ensure data consistency across devices.

### I/O System Management

One of the purposes of an operating system is to hide the peculiarities of specific hardware devices from the user.

Example: in UNIX, the peculiarities of I/O devices are hidden from the bulk of the operating system itself by the I/O subsystem.

The I/O subsystem consists of several components:

- A memory-management component that includes buffering, caching, and spooling
- A general device-driver interface
- Drivers for specific hardware devices

Only the device driver knows the peculiarities of the specific device to which it is assigned.

### Security and Protection

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

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

**Protection**, is any mechanism for controlling the access of processes or users to the resources defined by a computer system. Protection can improve reliability by detecting latent errors at the interfaces between component subsystems. Early detection of interface errors can prevent contamination of a healthy subsystem by another subsystem that is malfunctioning.

An unprotected resource cannot defend against use (or misuse) by an unauthorized or incompetent user.

**Security** protects against attacks like viruses, denial-of-service (DoS), identity theft, and unauthorized access. Security features involve policies, operating-system functions, and software tools to defend against these threats.

Prevention of these attacks is considered an operating system function on some systems, while other systems leave it to policy or additional software.

Protection and security are required to distinguish among all its users. Most operating systems maintain a list of user names and associated **user identifier (user IDs)**.

In Windows, this is a **security ID (SID)**. These numerical IDs are unique, one per user. When a user logs in to the system, the authentication stage determines the appropriate user ID for the user. That user ID is associated with all of the user's processes and threads.

In some circumstances, we wish to distinguish among sets of users rather than individual users. For example, the owner of a file on a UNIX system may be allowed to issue all operations on that file, whereas a selected set of users may be allowed only to read the file. To accomplish this, we need to define a group name and the set of users belonging to that group.

**Group IDs** allow managing permissions for sets of users. A user can belong to multiple groups.

In normal system use, the user ID and group ID for a user are sufficient. However, a user sometimes needs to **escalate privileges** to gain extra permissions for an activity. The user may need access to a device that is restricted.

Example: Operating systems provide various methods to allow privilege escalation. On UNIX, for instance, the *setuid* attribute on a program causes that program to run with the user ID of the owner of the file, rather than the current user's ID. The process runs with this **effective UID** until it turns off the extra privileges or terminates.

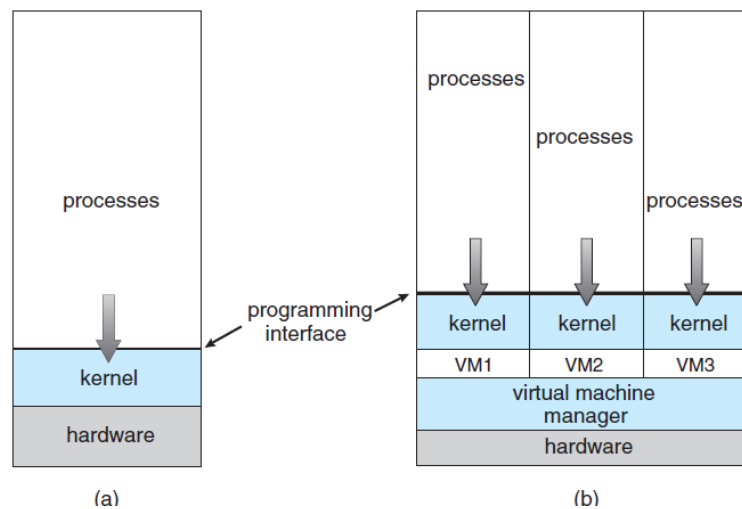
### Virtualization

**Virtualization** is a technology that allows us to abstract the hardware of a single computer (the CPU, memory, disk drives, network interface cards, and so forth) into several different execution environments, by creating the illusion that each separate environment is running on its own private computer.

These environments can be viewed as different individual operating systems (for example, Windows and UNIX) that may be running at the same time and may interact with each other. A user of a **virtual machine** can switch among the various operating systems in the same way a user can switch among the various processes running concurrently in a single operating system.

Virtualization allows operating systems to run as applications within other operating systems.

**Emulation** is another related technology. Emulation simulates hardware, to run software designed for one type of CPU on a different type (e.g., running IBM CPU apps on Intel CPUs). However, emulation can be slow since it translates instructions one by one, whereas virtualization allows an operating system to run natively on the same type of CPU.



### A computer running (a) a single operating system and (b) three virtual machines

Virtualization first emerged on IBM mainframes to allow multiple users run tasks on a single system.

Running multiple virtual machines allowed many users to run tasks on a system designed for a single user.

Later VMware created a new virtualization technology in the form of an application that ran on Windows. That application ran one or more **guest** copies of Windows or other native x86 operating systems, each running its own applications.

Windows was the **host operating system**, and the VMware application was the **virtual machine manager (VMM)**. The VMM runs the guest operating systems, manages their resource use, and protects each guest from the others.

Virtualization is increasingly popular, allowing users to run different operating systems on laptops and desktops (e.g., running Windows on a macOS laptop).

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

### Computing Environments

- ❖ **Traditional Computing**
- ❖ **Mobile Computing**
- ❖ **Client Server Computing**
- ❖ **Peer-to-Peer Computing**
- ❖ **Cloud computing**
- ❖ **Real-time Embedded Systems**

### **Traditional Computing**

As computing has matured, the lines separating many of the traditional computing environments have blurred.

In the past, a typical office setup had PCs connected to servers for file and print services, with limited remote access and portability provided by laptops. Today, web technologies and better network speeds allow companies to use portals for accessing internal servers, thin clients for simpler maintenance and security, and mobile devices that sync with PCs or access networks wirelessly for portability and web access.

At home, most users once had a single computer with a slow modem connection to the office, the Internet, or both. Today, network-connection speeds are available at great cost are relatively inexpensive in many places, giving home users more access to more data. These fast data connections are allowing home computers to serve up web pages and to run networks that include printers, client PCs, and servers.

Many homes use **firewall** to protect their networks from security breaches. Firewalls limit the communications between devices on a network.

In earlier computing, resources were scarce. Batch systems processed jobs in bulk without user interaction, while interactive systems waited for user input. Time-sharing systems let multiple users share computing resources, cycling processes through the CPU using scheduling algorithms.

Modern computing still uses time-sharing for multitasking, but most processes belong to one user. For example, a desktop PC manages multiple tasks at once, and even a web browser runs separate processes for each open tab or website, sharing CPU time between them.

### **Mobile Computing**

**Mobile computing** refers to computing on handheld smartphones and tablet computers. These devices share the distinguishing physical features of being portable and lightweight.

These devices were once limited in screen size, memory, and functionality compared to desktops and laptops. However, modern mobile devices now rival or even surpass traditional computers in many areas.

Today, mobile devices are used for tasks like email, web browsing, playing media, reading digital books, taking photos, and recording videos.

Augmented reality (AR) applications use mobile device features, like GPS and accelerometers, to overlay digital information on real-world views.

They also support unique features, including:

- **GPS chips** for precise location tracking and navigation.
- **Accelerometers** to detect tilting, shaking, and orientation, enabling intuitive controls in games and other apps.
- **Gyroscopes** for enhanced motion detection, useful in augmented reality applications.

Mobile devices access online services through **Wi-Fi (802.11 standard)** or **cellular networks**, though their processing power and storage are typically smaller than those of desktops. For instance, a smartphone might have 256 GB of storage, while desktops can have several terabytes.

Two operating systems currently dominate mobile computing: **Apple iOS** and **Google Android**. iOS was designed to run on Apple iPhone and iPad mobile devices.

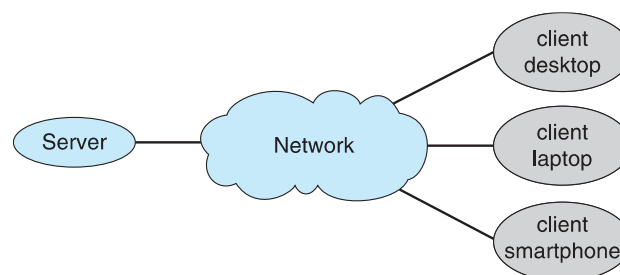
### Client Server Computing

In a client-server system, servers handle requests from clients over a network.

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

The **compute-server system** provides an interface to which a client can send a request to perform an action (for example, read data). In response, the server executes the action and sends the results to the client. A server running a database that responds to client requests for data is an example of such a system.

**File servers** provide access to files, allowing clients to create, read, update, or delete them. An example is a web server that delivers web pages or multimedia content to browsers.



**General structure of a client–server system**

### Peer-to-Peer Computing

In this model, clients and servers are not distinguished from one another.

Instead, all nodes within the system are considered peers, and each may act as either a client or a server, depending on whether it is requesting or providing a service. Peer-to-peer systems offer an advantage over traditional client–server systems.

In a client–server system, the server is a bottleneck; but in a peer-to-peer system, services can be provided by several nodes distributed throughout the network.

To participate in a peer-to-peer system, a node must first join the network of peers. Once a node has joined the network, it can begin providing services to—and requesting services from—other nodes in the network.

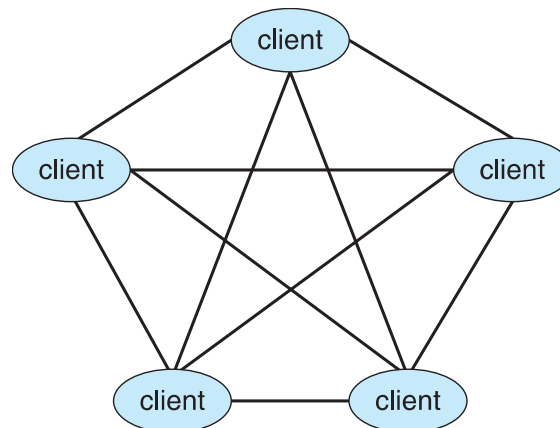
Determining what services are available is accomplished in one of two general ways:

- When a node joins a network, it registers its service with a centralized lookup service on the network. Any node desiring a specific service first contacts this centralized lookup service to determine which node provides the service.
- An alternative scheme uses no centralized lookup service. A node broadcasts a service request to all peers. Peers offering the service respond directly, using a discovery protocol for communication.

Examples of P2P networks:

- **Napster:** Used a central server to index files, but file exchanges occurred between peers. It was shut down in 2001 due to copyright issues.
- **Gnutella:** Used decentralized lookup by broadcasting requests to peers.

Skype is another example of peer-to-peer computing. It allows clients to make voice calls and video calls and to send text messages over the Internet using a technology known as **voice over IP (VoIP)**. Skype uses a hybrid peer-to-peer approach. It includes a centralized login server, but it also incorporates decentralized peers and allows two peers to communicate.



**Peer-to-peer system with no centralized service**

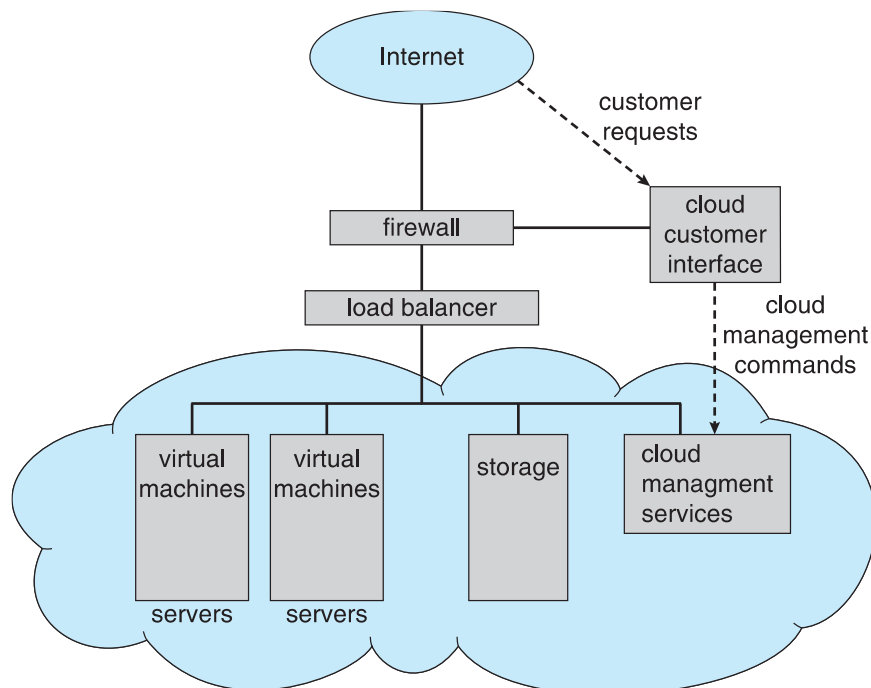
## Cloud computing

Cloud computing delivers computing, storage, and even applications as a service across a network. It is a logical extension of virtualization, because it uses virtualization as a base for its functionality.

Example: the Amazon Elastic Compute Cloud (**ec2**) facility has thousands of servers, millions of virtual machines, and petabytes of storage available for use by anyone on the Internet.

## Types of cloud computing

- **Public cloud**—a cloud available via the Internet to anyone willing to pay for the services
- **Private cloud**—a cloud run by a company for that company's own use
- **Hybrid cloud**—a cloud that includes both public and private cloud components
- **Software as a service (SaaS)**—one or more applications (such as word processors or spreadsheets) available via the Internet
- **Platform as a service (PaaS)**—a software stack ready for application use via the Internet (for example, a database server)
- **Infrastructure as a service (IaaS)**—servers or storage available over the Internet (for example, storage available for making backup copies of production data)



## Cloud computing

Cloud services often mix these types. Traditional operating systems are part of cloud infrastructure, while **virtual machine managers (VMMs)** control virtual machines.

Cloud management tools, like VMware vCloud Director and Eucalyptus, manage resources and serve as a new layer of operating systems for cloud environments.

## Real-time Embedded Systems

Embedded computers are the most common type of computers, found in devices like car engines, robots, and microwaves. They perform specific tasks, often with simple operating systems.

Usually, they have little or no user interface, preferring to spend their time monitoring and managing hardware devices, such as automobile engines and robotic arms.

Some embedded systems are general-purpose computers, running standard operating systems—such as Linux while others rely on specialized hardware with application-specific integrated circuits (**ASICs**) that perform their tasks without an operating system.

The use of embedded systems is growing, powering smart homes where central systems control lights, heating, and appliances. Web access allows remote control, and future appliances like refrigerators may even order groceries automatically.

Most embedded systems use **real-time operating systems** (RTOS) to meet strict timing requirements. These systems process sensor data and adjust controls in applications like medical imaging, industrial automation, and automotive fuel injection.

Real-time systems must complete tasks within fixed time limits to avoid failure. For example, a robot arm must stop before hitting a car, while a standard computer system can tolerate slower responses.

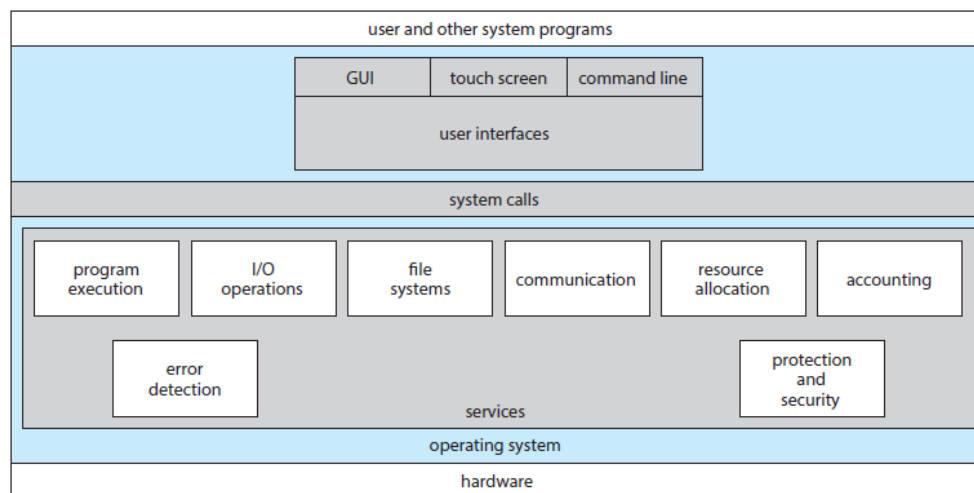
## Operating Systems Structures

### Services

An operating system provides an environment for the execution of programs.

It makes certain services available to programs and to the users.

The specific services differ from one operating system to another.



### **A view of operating system services**

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). This interface can take several forms. Most commonly, a **graphical user interface** (GUI) is used. Here, the interface is a window system with a mouse that serves as a pointing device to direct I/O, choose from menus, and make selections and a keyboard to enter text.

Mobile systems such as phones and tablets provide a **touch-screen interface**, enabling users to slide their fingers across the screen or press buttons on the screen to select choices.



**Command-line interface (CLI)**, uses text commands and a method for entering them (say, a keyboard for typing in commands in a specific format with specific options).

Some systems provide two or all three of these variations.

- **Program execution** - The system must be able to load a program into memory and to run that program. The program must be able to end its execution, either normally or abnormally (indicating error).

- **I/O operations** - A running program may require I/O, which may involve a file or an I/O device. For specific devices, special functions may be desired (such as reading from a network interface or writing to a file system). For efficiency and protection, users usually cannot control I/O devices directly. Therefore, the operating system must do I/O.

- **File-system manipulation** - Programs need to read and write files and directories. They also need to create and delete them by name, search for a given file, and list file information.

- **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 of a network.

Communications may be implemented via **shared memory**, in which two or more processes read and write to a shared section of memory, or **message passing**, in which packets of information in predefined formats are moved between processes by the operating system.

- **Error detection** - The operating systems monitors and handles errors from hardware, devices, or programs, ensuring system stability by taking actions like halting, terminating faulty processes, or returning error codes.

Another set of operating-system functions exists for ensuring the efficient operation of the system.

- **Resource allocation** - When multiple processes run simultaneously, the operating system allocates resources like CPU time, memory, and storage. Special routines manage CPU scheduling and peripheral device usage.

- **Logging** - We want to keep track of which programs use how much and what kinds of computer resources. This record keeping may be used for accounting or for accumulating usage statistics. Usage statistics may be a valuable tool for system administrators to reconfigure the system to improve computing services.

- **Protection and security**. The OS controls access to resources, preventing interference between processes. It ensures user authentication (e.g., passwords) and protects devices from unauthorized access, recording attempts to detect security breaches. Security depends on strong precautions across the entire system.

When several separate processes execute concurrently, it should not be possible for one process to interfere with the others or with the operating system itself. Protection involves ensuring that all access to system resources is controlled. Security of the system from outsiders is also important.

### User and OS Interface

- ❖ **Command Interpreters**
- ❖ **Graphical User Interface**
- ❖ **Touch-Screen Interface**
- ❖ **Choice of Interface**

#### **Command Interpreters**

Most operating systems, like Linux, UNIX, and Windows, use a **command interpreter** (or **shell**) to execute user commands. The shell can be one of several types, such as the C shell or Bourne-Again shell (bash). Users choose a shell based on personal preference.

```

1. root@r6181-d5-us01:~ (ssh)
Last login: Thu Jul 14 08:47:01 on ttys002
iMacPro:~ pbg$ ssh root@r6181-d5-us01
root@r6181-d5-us01's password:
Last login: Thu Jul 14 06:01:11 2016 from 172.16.16.162
[root@r6181-d5-us01 ~]# uptime
 06:57:48 up 16 days, 10:52,  3 users,  load average: 129.52, 80.33, 56.55
[root@r6181-d5-us01 ~]# df -kh
Filesystem      Size  Used Avail Use% Mounted on
/dev/mapper/vg_ks-lv_root
                  50G   19G   28G   41% /
tmpfs            127G  520K  127G    1% /dev/shm
/dev/sda1        477M   71M  381M   16% /boot
/dev/dssd0000    1.0T  480G  545G   47% /dssd_xfs
tcp://192.168.150.1:3334/orangefs
                  12T   5.7T   6.4T   47% /mnt/orangefs
/dev/gpfs-test   23T   1.1T   22T    5% /mnt/gpfs
[root@r6181-d5-us01 ~]#
[root@r6181-d5-us01 ~]# ps aux | sort -nrk 3,3 | head -n 5
root    97653  11.2  6.6 42665344 17520636 ?    S<Ll  Jul13 166:23 /usr/lpp/mmfs/bin/mmfstd
root    69849   6.6  0.0      0      0 ?        S    Jul12 181:54 [vpthread-1-1]
root    69850   6.4  0.0      0      0 ?        S    Jul12 177:42 [vpthread-1-2]
root    3829    3.0  0.0      0      0 ?        S    Jun27 730:04 [rp_thread 7:0]
root    3826    3.0  0.0      0      0 ?        S    Jun27 728:08 [rp_thread 6:0]
[root@r6181-d5-us01 ~]# ls -l /usr/lpp/mmfs/bin/mmfstd
-r-x----- 1 root root 20667161 Jun  3  2015 /usr/lpp/mmfs/bin/mmfstd
[root@r6181-d5-us01 ~]#

```

#### **The bash shell command interpreter in macOS**

The main function of the command interpreter is to get and execute the next user-specified command. Many of the commands given at this level manipulate files: create, delete, list, print, copy, execute, and so on.

These commands can be implemented in two general ways.

1. **Built-in Commands:** the command interpreter itself contains the code to execute the command. For example, a command to delete a file may cause the command interpreter to jump to a code that sets up the parameters and makes the appropriate system call.
2. **System Programs:** In this case, the command interpreter does not understand the command in any way; it uses the command to identify a file to be loaded into memory and executed.

Thus, the UNIX command to delete a file

**rm file.txt**

would search for a file called rm, load the file into memory, and execute it with the parameter file.txt.

The logic associated with the `rm` command would be defined completely by the code in the file `rm`. In this way, programmers can add new commands to the system easily by creating new files with the proper program logic. The command-interpreter program, which can be small, does not have to be changed for new commands to be added.

### Graphical User Interface

A **graphical user interface (GUI)** is a user-friendly way to interact with an operating system. Instead of entering commands directly via a command-line interface, users employ a mouse-based window and-menu system.

The user moves the mouse to position its pointer on images, or **icons**, on the screen (the desktop) that represent programs, files, directories, and system functions.

Depending on the mouse pointer's location, clicking a button on the mouse can invoke a program, select a file or directory—known as a **folder**—or pull down a menu that contains commands.

GUIs were first developed in the early 1970s at Xerox PARC and appeared on the Xerox Alto computer in 1973. GUIs became more popular with the release of Apple's Macintosh in the 1980s. Microsoft introduced a GUI with Windows 1.0, adding it to their MS-DOS system, and improved it in later versions.

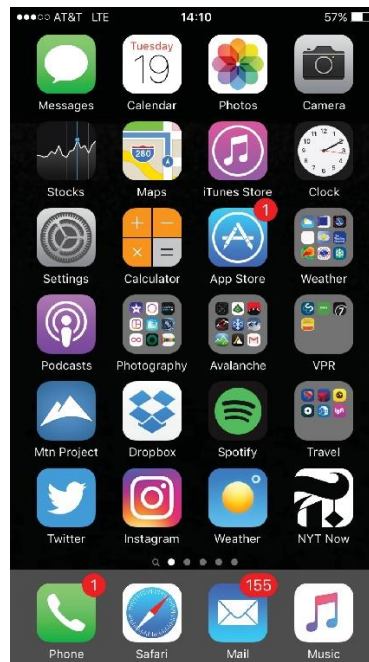
UNIX systems traditionally used command-line interfaces, but GUIs like **KDE** and **GNOME** have been developed for UNIX and Linux systems. These are open-source, meaning users can view and modify their source code under certain licenses.

### Touch-Screen Interface

Command-line interface or mouse-and-keyboard system is impractical for most mobile systems, smartphones and handheld tablet computers.

Mobile devices like smartphones and tablets use a **touch-screen interface** instead of a command-line or mouse-and-keyboard system. users interact by making **gestures** on the touch screen—for example, pressing and swiping fingers across the screen.

Although earlier smartphones included a physical keyboard, most smartphones and tablets now simulate a keyboard on the touch screen. For example, both the iPhone and iPad use the Springboard touch-screen interface.



**The iPhone touch screen**

### Choice of Interface

The choice of whether to use a command-line or GUI interface is mostly one of personal preference.

**System administrators** who manage computers and **power users** who have deep knowledge of a system frequently use the command-line interface. Because it's more efficient and allows faster access to system tasks.

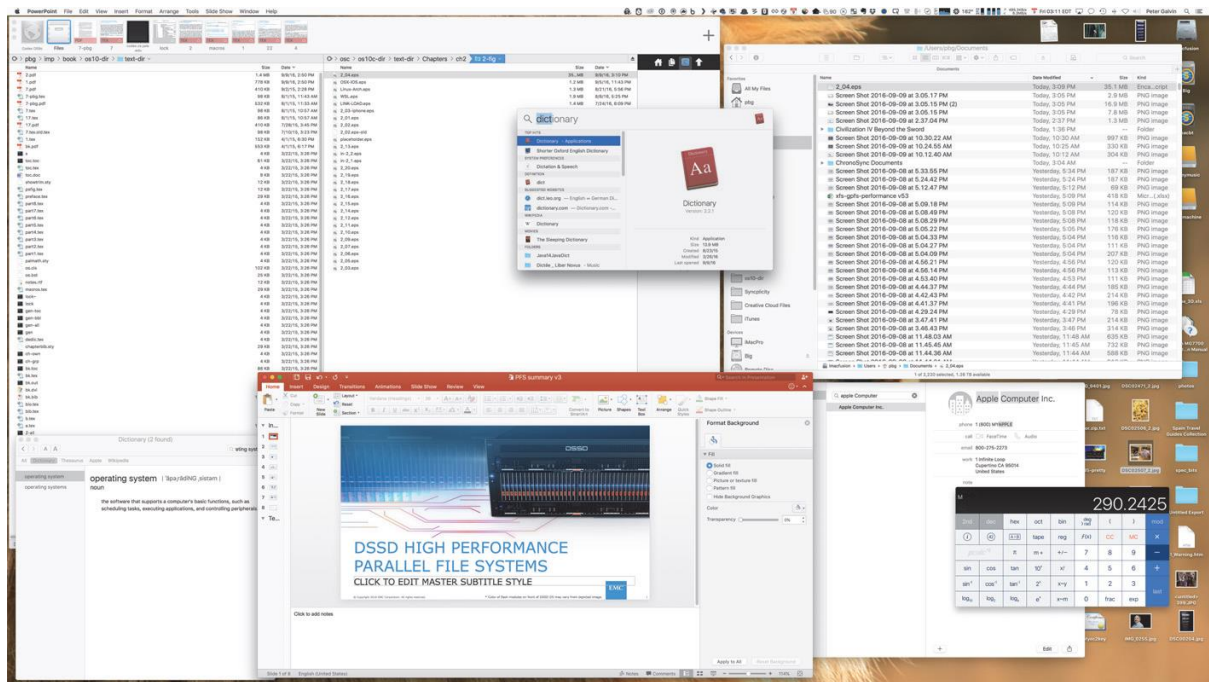
Command-line interfaces also make repetitive tasks easier by using shell scripts, which can automate commands.

Example: if a frequent task requires a set of command line steps, those steps can be recorded into a file, and that file can be run just like a program. The program is not compiled into executable code but interpreted by the command-line interface. These **shell scripts** are common on UNIX and Linux systems.

On the other hand, Windows users prefer the GUI and rarely use the shell interface. Recent versions of Windows offer both a GUI for desktops and touch-screen support for tablets. The macOS system, historically GUI-based, provides both an Aqua GUI and a command-line interface.

Although there are apps that provide a command-line interface for iOS and Android mobile systems, they are rarely used. Instead, almost all users of mobile systems interact with their devices using the touch-screen interface.

The user interface can vary from system to system and even from user to user within a system.



## The macOS GUI

### System Calls

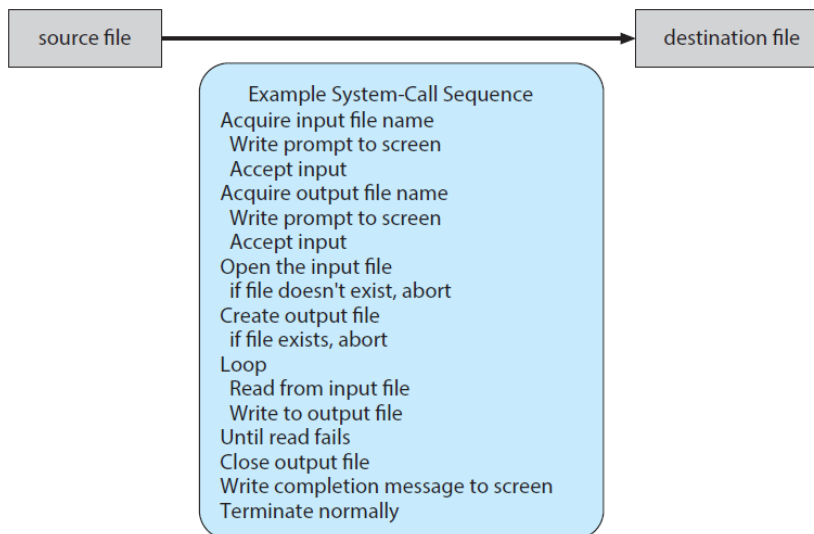
**System calls** provide an interface to the services made available by an operating system. These calls are generally available as functions written in C and C++.

Example: writing a simple program to read data from one file and copy them to another file. The first input that the program will need is the names of the two files: the input file and the output file.

**cp in.txt out.txt**

This command copies the input file in.txt to the output file out.txt.

Once the two file names have been obtained, the program must open the input file and create and open the output file. Each of these operations requires another system call.



**Example of how system calls are used**

When both files are set up, we enter a loop that reads from the input file (a system call) and writes to the output file (another system call). Each read and write must return status information regarding various possible error conditions. On input, the program may find that the end of the file has been reached or that there was a hardware failure in the read.

The write operation may encounter various errors, depending on the output device (for example, no more available disk space).

Finally, after the entire file is copied, the program may close both files (two system calls), write a message to the console or window (more system calls), and finally terminate normally (the final system call).

## Application Programming Interface

Systems execute thousands of system calls per second to communicate with the operating system, but developers usually use **Application Programming Interfaces (APIs)** instead of directly working with system calls.




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).

### 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:

<code>#include &lt;unistd.h&gt;</code>		
<code>ssize_t</code>	<code>read</code>	<code>(int fd, void *buf, size_t count)</code>
		
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`.

A programmer accesses an API via a library of code provided by the operating system. In the case of UNIX and Linux for programs written in the C language, the library is called **libc**.

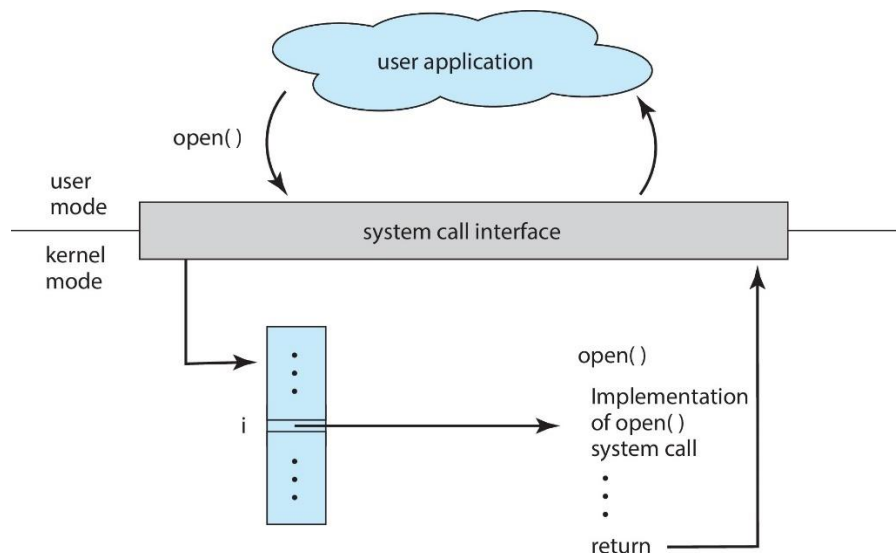
APIs are simpler and more user-friendly than system calls. Behind the scenes, the API functions call the actual system calls.

#### Example:

The Windows function `CreateProcess()` uses the system call `NTCreateProcess()` in the Windows kernel.

Another important factor in handling system calls is the **run-time environment (RTE)**—Manages the software tools needed to run a program, like compilers, libraries, and loaders.

The RTE provides a **system-call interface** - Links API functions to system calls. It assigns numbers to system calls and uses a table to match calls and return status information to the API function.



#### **The handling of a user application invoking the `open()` system call**

Three general methods are used to pass parameters to the operating system.

The simplest approach is to **pass the parameters in registers**.

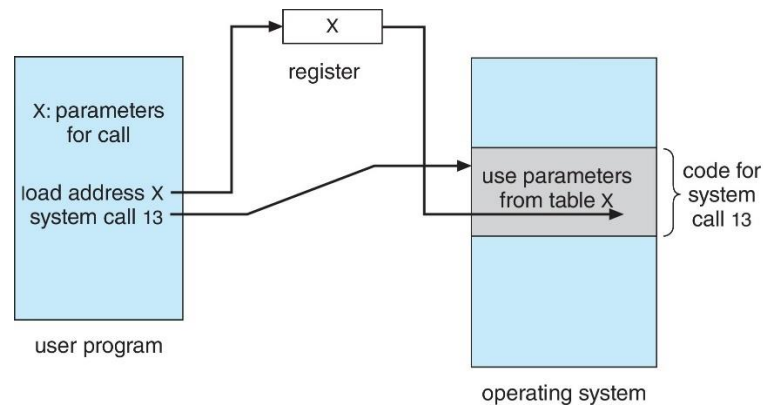
In some cases, however, there may be more parameters than registers. In these cases, the parameters are generally stored in a block, or table, in memory, and the **address of the block is passed as a parameter** in a register. Linux uses a combination of these approaches.

If there are five or fewer parameters, registers are used.

If there are more than five parameters, the block method is used. Parameters also can be placed, or **pushed**, onto a **stack** by the program and **popped** off the stack by the operating system.

Some operating systems prefer the block or stack method because those approaches do not limit the number or length of parameters being passed.





### Passing of parameters as a table

### Types of System Calls

System calls can be grouped roughly into six major categories:

- ❖ **Process control**
- ❖ **File management**
- ❖ **Device management**
- ❖ **Information maintenance**
- ❖ **Communications and**
- ❖ **Protection**

### Process control

A running program needs to halt its execution either normally (`end()`) or abnormally (`abort()`).

If a system call is made to terminate the currently running program abnormally, or if the program runs into a problem and causes an error trap, a dump of memory is sometimes taken and an error message generated.

The dump is written to a special log file on disk and may be examined by a **debugger**—a system program designed to aid the programmer in finding and correcting errors, or **bugs**—to determine the cause of the problem.

After termination, control returns to the **command interpreter** to await further commands. In graphical user interfaces (GUIs), a pop-up may notify the user of errors.

Programs can load and execute other programs using system calls like `load()` or `create process()`. The system must decide whether to:

1. **Return to the original program** (saving its state).
2. **Allow both programs to run concurrently** (creating a new process).

Processes can be managed with system calls for:

- **Setting or getting attributes** (e.g., priority).
- **Terminating a process** (`terminate process()`).
- **Waiting for processes or events** (`wait event()` and `signal event()`).



- **Locking shared data** (acquire lock() and release lock()).

### Single-tasking vs. Multitasking

#### 1. Single-tasking system (Arduino):

- Runs one program at a time (called a **sketch**).
- Uses sensors for input and has no operating system or advanced user interface.
- A **boot loader** uploads the sketch from a PC into memory.

#### 2. Multitasking system (FreeBSD):

- Allows multiple programs to run simultaneously.
- Uses the fork() system call to create a new process and exec() to load a program.
- The shell can wait for the process to finish or let it run in the background.
- The exit() call terminates the process and returns a status code.

Multitasking systems provide flexibility, allowing users to run multiple processes, change priorities, and manage I/O through files or GUIs.

#### Types of System Calls

- Process control
  - create process, terminate process
  - load, execute
  - get process attributes, set process attributes
  - wait event, signal event
  - allocate and free memory
- File management
  - create file, delete file
  - open, close
  - read, write, reposition
  - get file attributes, set file attributes
- Device management
  - request device, release device
  - read, write, reposition
  - get device attributes, set device attributes
  - logically attach or detach devices
- Information maintenance

- get time or date, set time or date
- get system data, set system data
- get process, file, or device attributes
- set process, file, or device attributes
- Communications
  - create, delete communication connection
  - send, receive messages
  - transfer status information
  - attach or detach remote devices
- Protection
  - get file permissions
  - set file permissions

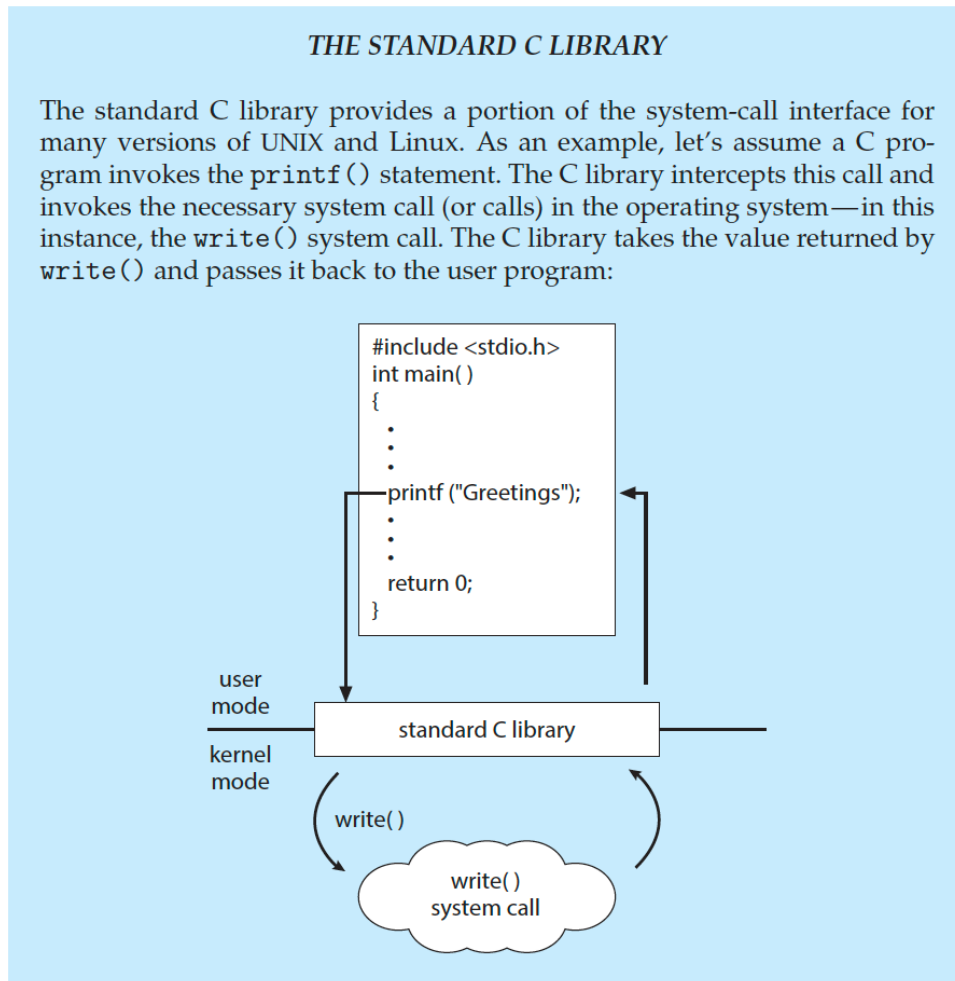
#### *EXAMPLES OF WINDOWS AND UNIX SYSTEM CALLS*

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

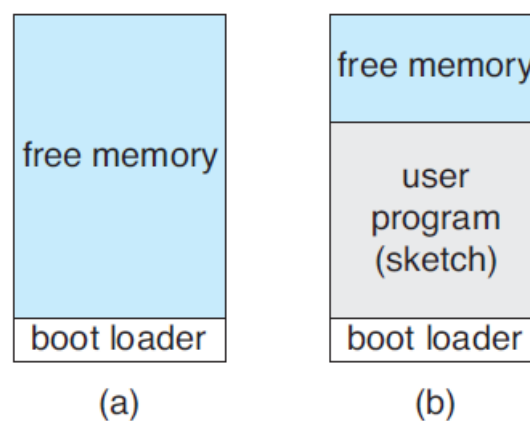
	Windows	Unix
<b>Process control</b>	CreateProcess() ExitProcess() WaitForSingleObject()	fork() exit() wait()
<b>File management</b>	CreateFile() ReadFile() WriteFile() CloseHandle()	open() read() write() close()
<b>Device management</b>	SetConsoleMode() ReadConsole() WriteConsole()	ioctl() read() write()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

### Standard C Library Example

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



#### Example: Arduino

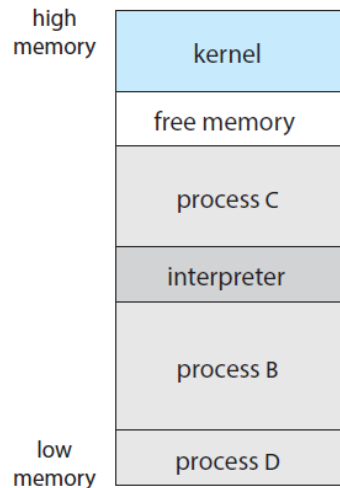


**Arduino execution. (a) At system startup. (b) Running a sketch.**

- Single-tasking
- No operating system
- Programs (sketch) loaded via USB into flash memory

- Single memory space
- Boot loader loads program
- Program exit -> shell reloaded

### Example: FreeBSD



### **FreeBSD running multiple programs**

- Unix variant
- Multitasking
- User login -> invoke user's choice of shell
- 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

### **File management**

Common system calls for working with files include:

- **`create()`** and **`delete()`**: For creating or removing files, requiring the file name and attributes.
- **`open()`**, **`read()`**, **`write()`**, and **`reposition()`**: To use and manage file contents.
- **`close()`**: To stop using a file.

Directories use similar operations for organization. Additionally, system calls like **`get file attributes()`** and **`set file attributes()`** manage file properties such as name, type, and permissions.

Some systems offer more functions, like **move()** and **copy()**, either as system calls or through APIs and system programs. System programs callable by other programs effectively act as APIs.

### Device management

Processes require resources like memory and disk drives to execute. System calls for device management include:

- **request()** and **release()**: To allocate and free devices, similar to **open()** and **close()** for files.
- **read()**, **write()**, and **reposition()**: For data operations on devices.

### Information maintenance

System calls transfer data between a program and the operating system:

- **time()** and **date()**: Retrieve the system time and date.
- **get process attributes()** and **set process attributes()**: Manage process information.
- **dump()** and debugging tools: Help diagnose errors, while **strace** traces system calls on Linux. Profiling tools use timer interrupts to analyze program execution times.
- Even microprocessors provide a CPU mode, known as **single step**, in which a trap is executed by the CPU after every instruction. The trap is usually caught by a debugger.

### Communications

Two models of interprocess communication:

1. **Message-Passing**: Processes exchange messages directly or via mailboxes. Common calls include **open connection()**, **send message()**, and **close connection()**.
2. **Shared Memory**: Processes use **shared memory create()** and **attach()** to exchange data by reading and writing to a shared region. Synchronization is required to prevent conflicts.

### Protection

Protection ensures secure access to system resources. Typical calls:

- **set permission()** and **get permission()**: Manage access rights.
- **allow user()** and **deny user()**: Control user access to resources.