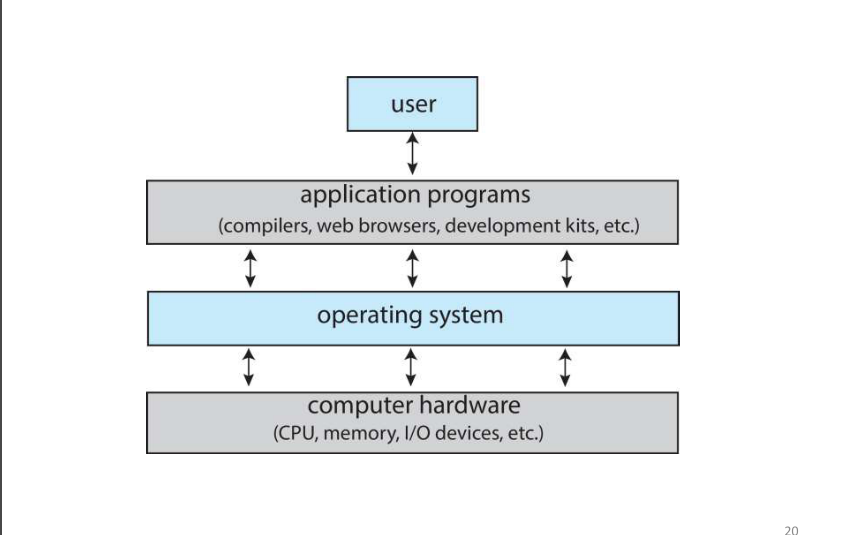


* An Application Programming Interface (commonly abbreviated as API) is like the secret handshake between different computer programs or components. It’s the bridge that allows them to chat, exchange information, and collaborate behind the scenes. Think of it as the backstage pass for software.
* An **API** defines a set of standardized requests and rules that enable different programs to communicate with each other. It’s like a well-organized menu of services that one program offers to another.
* **Operating System APIs:** These let applications interact with the underlying OS. Want to create a file or open a network socket? OS APIs have your back.

**What is the difference between multiprocessor and multicore architecture?**

Multicore and multiprocessor architectures are both designed to increase processing power and efficiency, but they differ in their design and implementation:

**Multicore Architecture:** In a multicore architecture, multiple processing cores are integrated onto a single physical processor or die. Each core is a separate processing unit that executes instructions independently, but shares resources such as memory and I/O interfaces.

**Characteristics:**

1. Multiple cores on a single chip

2. Shared memory and resources

3. Cores communicate through on-chip interconnects

4. Typically, each core has its own cache hierarchy

5. Can be homogeneous (identical cores) or heterogeneous (different cores)

Examples: Intel Core i7, AMD Ryzen 9

**Multiprocessor Architecture:** In a multiprocessor architecture, multiple processors are connected together to form a single system. Each processor can be a separate chip or module, and they communicate with each other through an interconnect or network.

**Characteristics:**

1. Multiple processors, each on a separate chip or module

2. Each processor has its own memory and resources

3. Processors communicate through an interconnect or network

4. Can be symmetric (SMP) or asymmetric (NUMA)

5. Typically used in high-performance computing, servers, and mainframes

Examples: Cluster computing, distributed systems, and some high-end servers

**Key differences:**

1. Integration: Multicore architectures integrate multiple cores on a single chip, while multiprocessor architectures connect separate processors.

2. Scalability: Multiprocessor architectures can scale to hundreds or thousands of processors, while multicore architectures are typically limited to 2-32 cores.

3. Communication: Multicore architectures use on-chip interconnects, while multiprocessor architectures use external interconnects or networks.

4. Memory: Multicore architectures share memory, while multiprocessor architectures typically have separate memory for each processor.

5. Power consumption: Multicore architectures generally consume less power than multiprocessor architectures.

When choosing between multicore and multiprocessor architectures, consider factors such as:

- Performance requirements

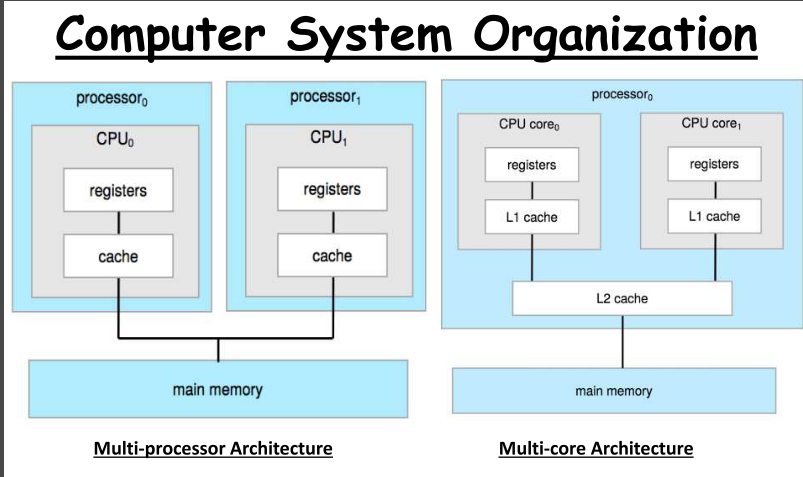
- Power consumption constraints

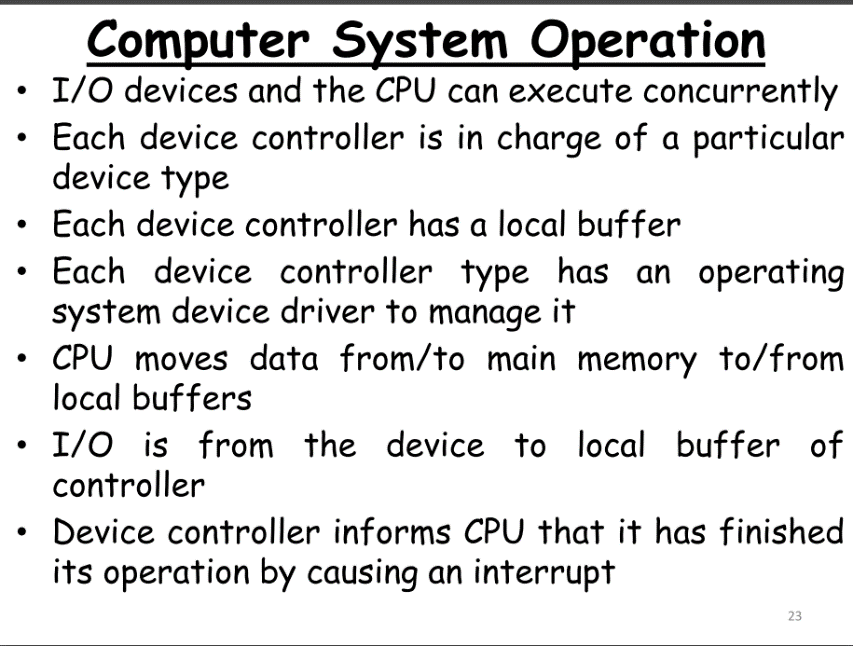
- Scalability needs

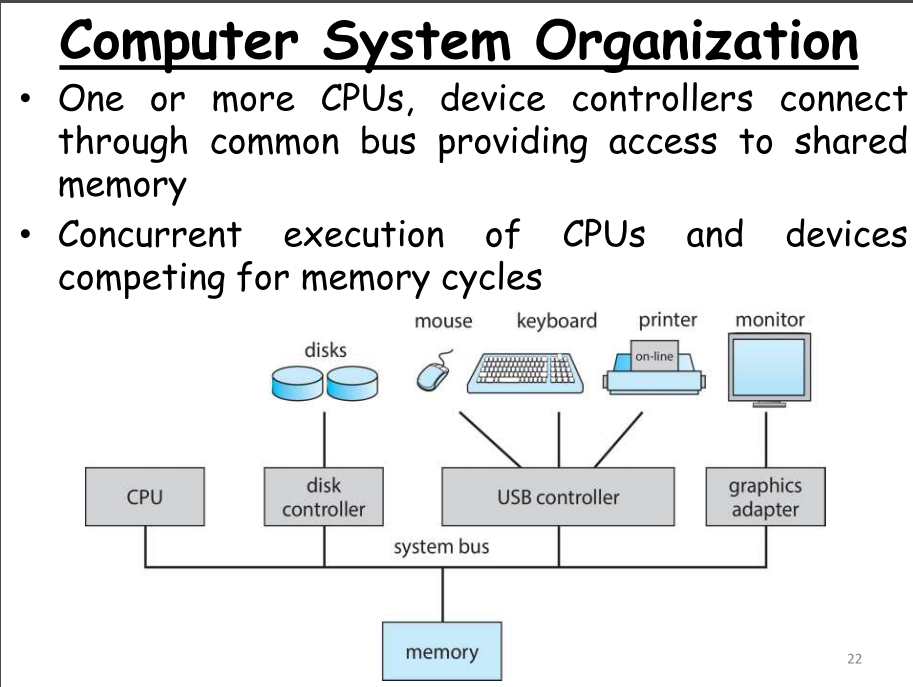
- Application characteristics (e.g., parallelism, memory intensity)

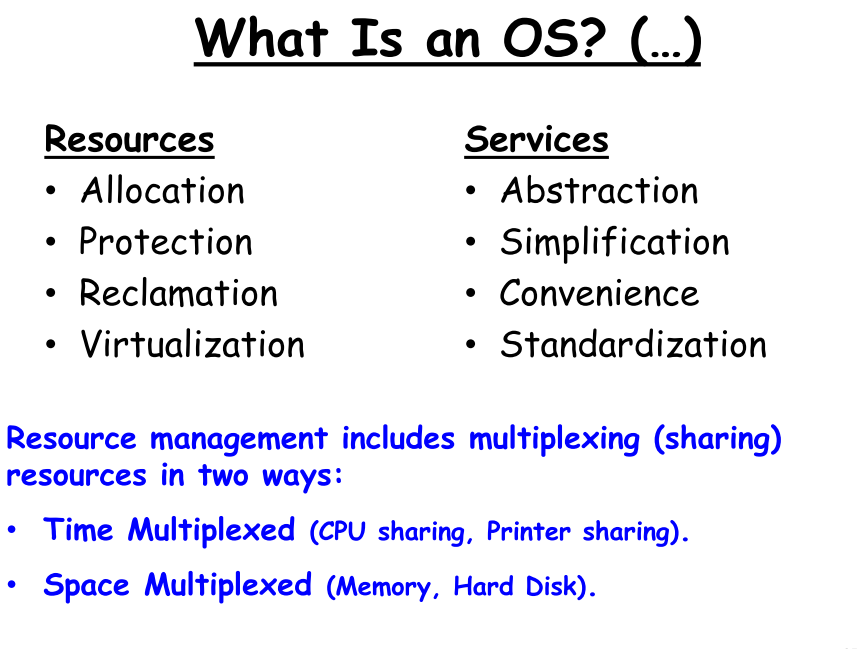
- Cost and complexity considerations

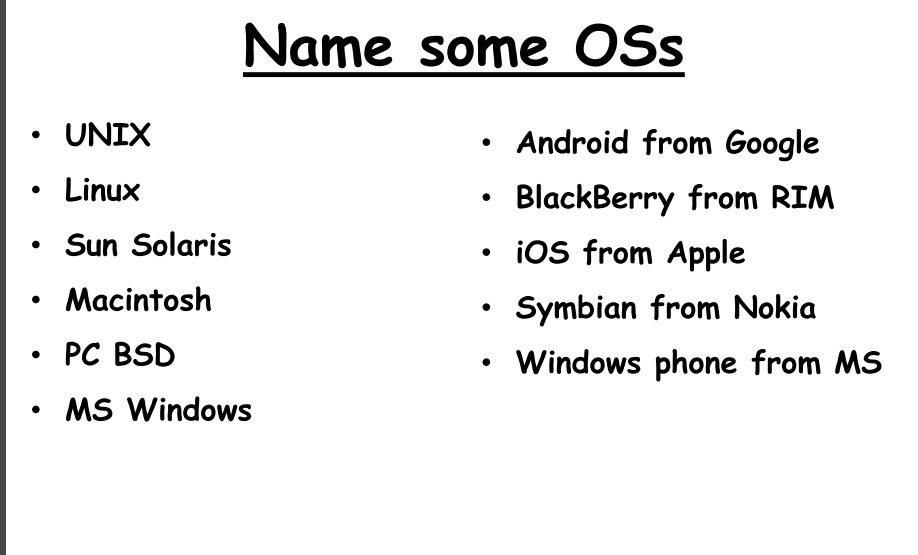
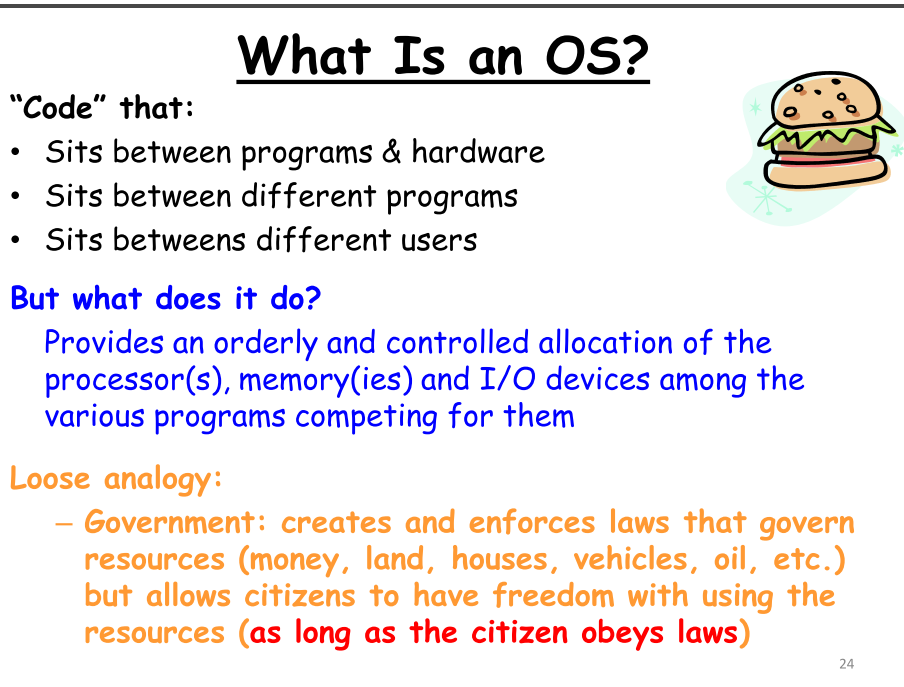
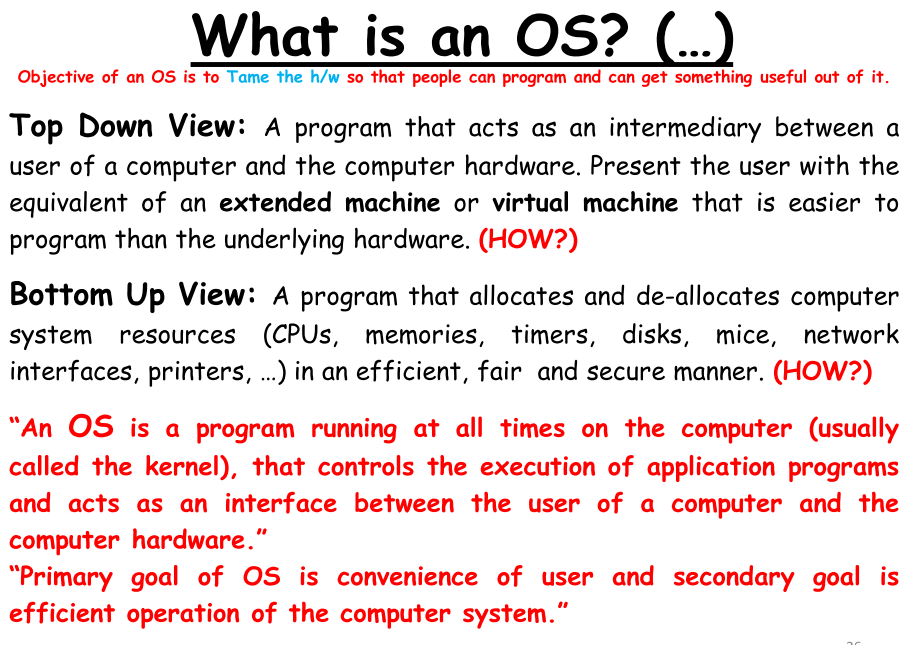
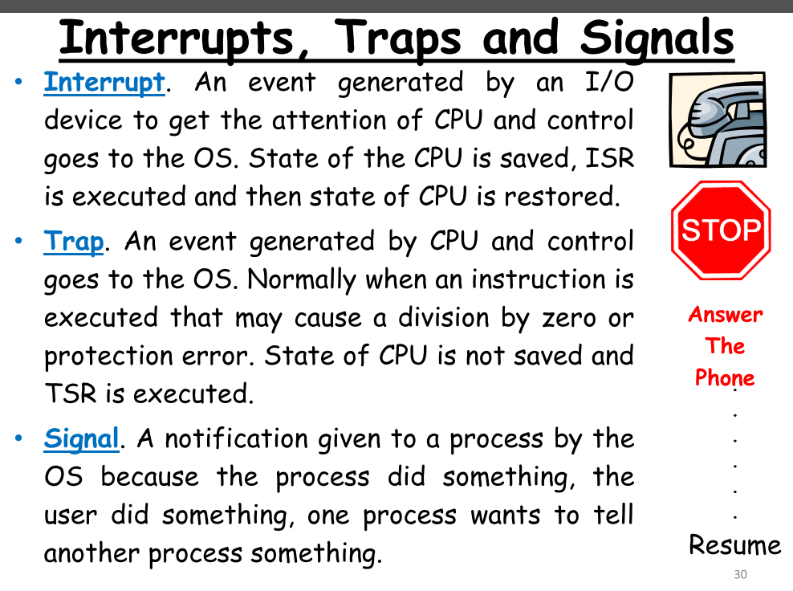
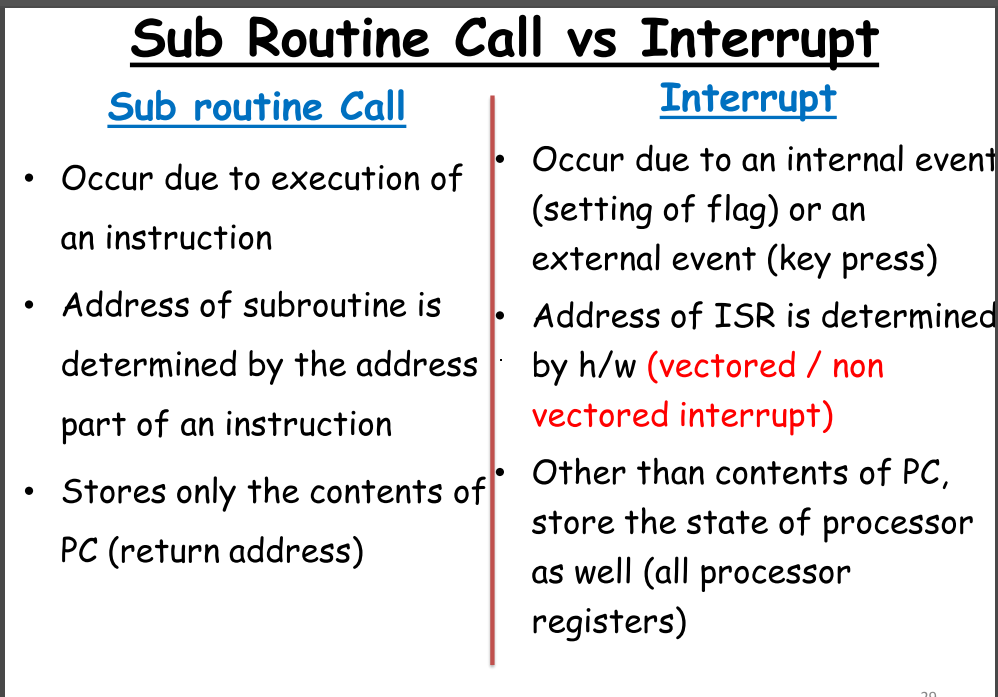
Keep in mind that modern systems often combine elements of both architectures, such as multi-socket servers with multicore processors.









s

A **function call** and an **interrupt** are both mechanisms that cause the execution of a new block of code, but they differ in how and when they occur, as well as in their purpose and control flow. Here’s a breakdown of the differences:

**1. Triggering Mechanism:**

* **Function Call**:
  + **Explicitly invoked by the program**. The programmer or the code itself decides when and where to call the function. It occurs as a normal part of program execution.
  + Example: sum(3, 4); is a function call that is deliberately placed in the code to execute the sum function.
* **Interrupt**:
  + **Triggered by external or internal events**. An interrupt is generated by hardware devices (hardware interrupt) or by software events such as exceptions (software interrupt). It is typically asynchronous and occurs outside the normal flow of the program.
  + Example: A keyboard press generating a hardware interrupt to notify the CPU.

**2. Control Flow:**

* **Function Call**:
  + **Synchronous**: The function call happens at a defined point in the program and the function returns control to the point where it was called after execution. The flow is predictable.
  + The caller waits until the function completes execution before resuming.
* **Interrupt**:
  + **Asynchronous**: Interrupts can occur at any time, interrupting the current execution of the program. Once the interrupt handler finishes, control is returned to the interrupted program at the point it was paused.
  + The interrupt handler runs outside the normal program flow and is managed by the operating system or hardware.

**3. Purpose:**

* **Function Call**:
  + To execute a **specific block of code** (function) when explicitly requested by the program.
  + Typically used to organize code, avoid redundancy, and improve modularity and maintainability.
* **Interrupt**:
  + To respond to **urgent events or conditions** that need immediate attention, such as hardware requests (I/O devices) or system errors (e.g., divide by zero).
  + Ensures that time-sensitive tasks (e.g., input handling, real-time clock) are handled without waiting for the current process to finish.

**4. Control:**

* **Function Call**:
  + **Under programmer control**: The programmer explicitly defines when and where the function is called. The program's execution is predictable and follows the logic set by the developer.
  + Control is passed from the calling function to the called function, and once the called function is done, control returns to the caller.
* **Interrupt**:
  + **Out of programmer control**: Interrupts can occur at any moment based on hardware signals or system events, and the CPU has to handle them immediately. The control of the program is temporarily handed over to an interrupt handler or service routine, which is generally managed by the operating system.

**5. Return Mechanism:**

* **Function Call**:
  + After the function executes, it **returns control** to the place in the code where the function was called. The execution continues from that point.
* **Interrupt**:
  + After the interrupt is handled by an **interrupt service routine (ISR)**, control is returned to the instruction that was executing at the moment the interrupt occurred, resuming normal program flow.

**6. Overhead:**

* **Function Call**:
  + Generally has **low overhead**. It involves saving the current execution state, jumping to the function, and then returning.
* **Interrupt**:
  + **Higher overhead** than a function call because it involves saving the entire CPU state (registers, program counter, etc.), handling the interrupt, and restoring the state before resuming the original program.

**Summary of Differences:**

| **Aspect** | **Function Call** | **Interrupt** |
| --- | --- | --- |
| **Triggering** | Explicitly called by program | Triggered by external or internal events |
| **Control Flow** | Synchronous, predictable | Asynchronous, unpredictable |
| **Purpose** | Code modularity, specific tasks | Handle urgent events, time-sensitive tasks |
| **Control** | Controlled by programmer | Controlled by hardware or operating system |
| **Return Mechanism** | Returns to caller after execution | Returns to interrupted point after handling |
| **Overhead** | Low | Higher due to context switching |

In summary, **function calls** are intentional and used within the flow of a program, while **interrupts** are external or internal events that disrupt the normal flow of the program to handle urgent tasks, often without the programmer’s direct involvement.

**Multiprogramming vs multitasking vs timesharing**

**Multiprogramming**, **multitasking**, and **timesharing** are all techniques used by operating systems to manage multiple tasks and make efficient use of system resources, particularly the CPU. Although they have similar goals of improving system utilization and responsiveness, they differ in their approaches and applications. Here's a comparison of the three:

### 1. ****Multiprogramming****:

* **Definition**: Multiprogramming is a technique where multiple programs reside in memory at the same time, and the CPU switches between them to maximize resource utilization. While one program is waiting for I/O operations (like reading from a disk), the CPU can execute another program.
* **Key Features**:
  + **CPU Utilization**: Designed to keep the CPU busy by always having a program ready to execute, reducing idle time.
  + **Non-preemptive**: Typically, a program runs until it needs to perform an I/O operation or voluntarily relinquishes the CPU. There is no forced interruption by the operating system.
  + **Context Switching**: When one program is waiting (e.g., for I/O), the OS switches context to another program.
* **Example**: In older batch systems, multiple jobs (programs) are kept in memory and processed one by one. If a job needs to wait for input from a peripheral, the CPU will switch to another job that is ready to run.

### 2. ****Multitasking****:

* **Definition**: Multitasking refers to the ability of an operating system to execute multiple tasks (processes) seemingly simultaneously. In reality, the CPU switches between tasks so quickly that users perceive all tasks as running concurrently.
* **Key Features**:
  + **Time-sharing**: Modern multitasking systems rely on time-sharing, where each task is given a small amount of CPU time (a time slice or quantum) before switching to another task.
  + **Preemptive**: In most modern systems, the OS can interrupt (preempt) a running task to ensure that other tasks get CPU time. This is important for responsiveness, especially in interactive systems.
  + **Concurrency**: Multitasking can give the illusion of parallel execution, even if there's only one CPU core.
* **Example**: In personal computers, you can open a web browser, listen to music, and edit a document at the same time. The OS ensures that each task gets enough CPU time to maintain responsiveness.

### 3. ****Time-sharing****:

* **Definition**: Time-sharing is a specific type of multitasking designed to support multiple users interacting with a system simultaneously. In time-sharing systems, the CPU's time is divided among multiple users, and each user gets a time slice to execute their tasks.
* **Key Features**:
  + **Multiple Users**: Primarily designed for environments where multiple users are interacting with the system concurrently (e.g., via terminals or remote access).
  + **Fair CPU Distribution**: The system switches between user tasks so quickly that each user feels like they have exclusive access to the CPU.
  + **Preemptive**: Like multitasking, time-sharing systems use preemptive multitasking to switch between user processes regularly, ensuring fairness and responsiveness.
  + **Interactive**: The goal is to maintain short response times so users can interact with the system in real-time, unlike batch processing in multiprogramming.
* **Example**: In mainframe or server environments, multiple users connect to a single machine via terminals, and the operating system allocates CPU time to each user’s tasks in quick succession.

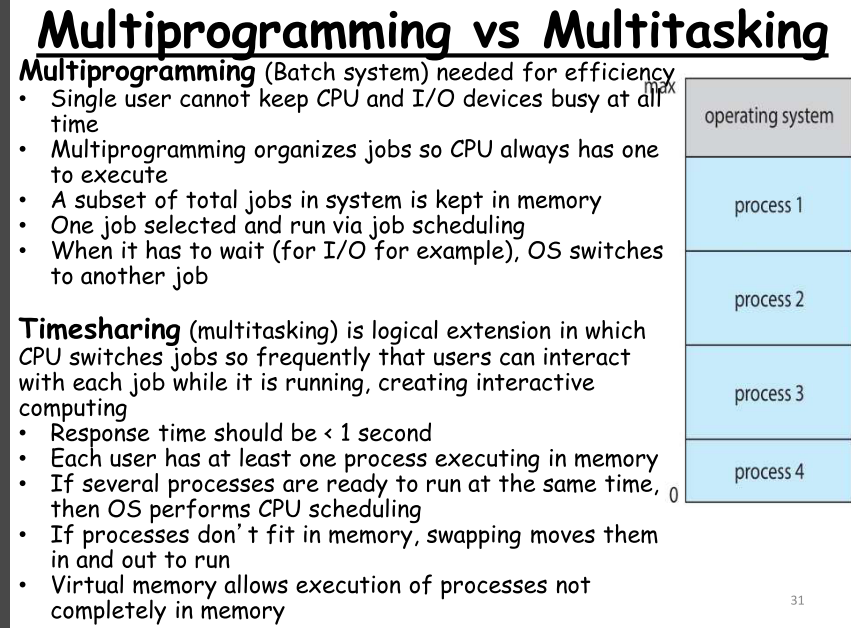
### ****Comparison Table****:

| **Aspect** | **Multiprogramming** | **Multitasking** | **Time-sharing** |
| --- | --- | --- | --- |
| **Purpose** | Maximize CPU utilization by switching between programs during I/O waits | Allow multiple tasks (processes) to run concurrently | Provide simultaneous access to multiple users |
| **Preemptive/Non-preemptive** | Non-preemptive (typically waits for I/O or voluntary relinquishing of CPU) | Preemptive (OS interrupts tasks for time-sharing) | Preemptive (fair CPU sharing among users) |
| **Focus** | CPU efficiency and utilization | User-level multitasking and system responsiveness | Fair access and responsiveness for multiple users |
| **Users** | Typically single-user systems | Single-user systems (modern computers) | Multi-user systems (e.g., servers, mainframes) |
| **Response Time** | Not a primary concern | Short response time for tasks | Short response time for multiple users |
| **Concurrency** | Programs run concurrently, but only one is active at a time | Tasks run concurrently, fast switching gives illusion of parallelism | Users run tasks concurrently in real-time |

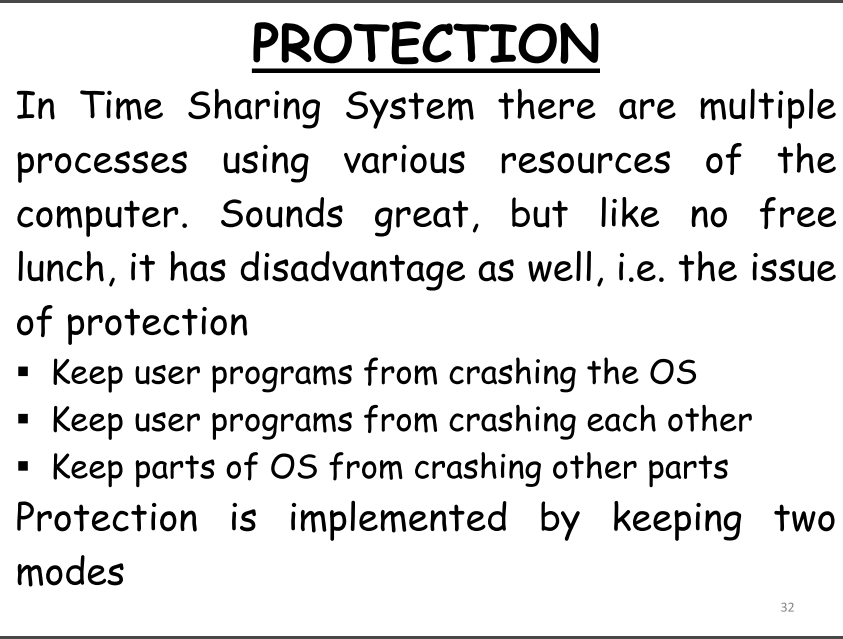
### Summary:

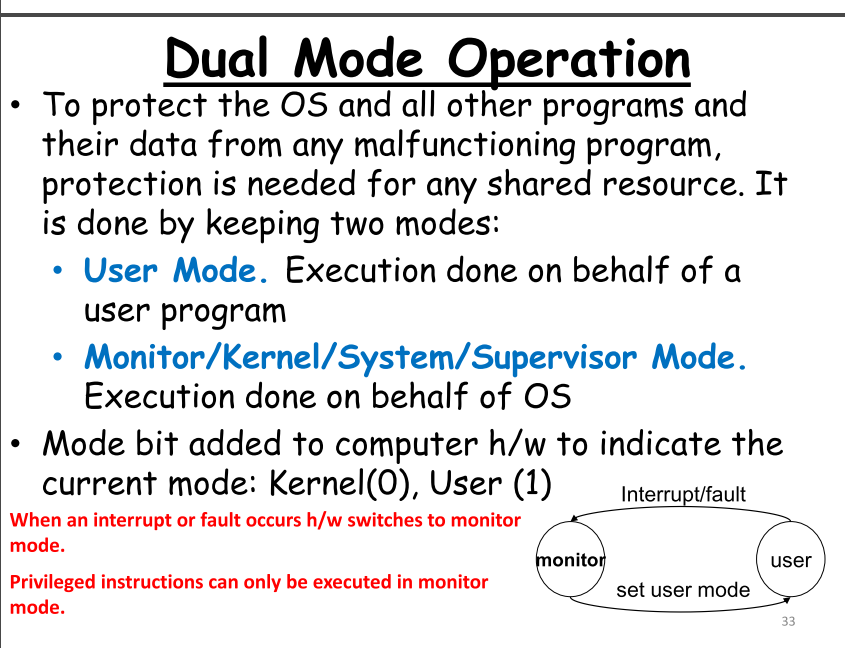
* **Multiprogramming**: Focuses on keeping the CPU busy by switching between programs when one is waiting for I/O, leading to better resource utilization but without real-time responsiveness.
* **Multitasking**: Allows multiple tasks to run seemingly simultaneously by rapidly switching between them, ensuring responsiveness for interactive users.
* **Time-sharing**: A specialized form of multitasking aimed at providing multiple users simultaneous access to a system, ensuring fair distribution of CPU resources among them.

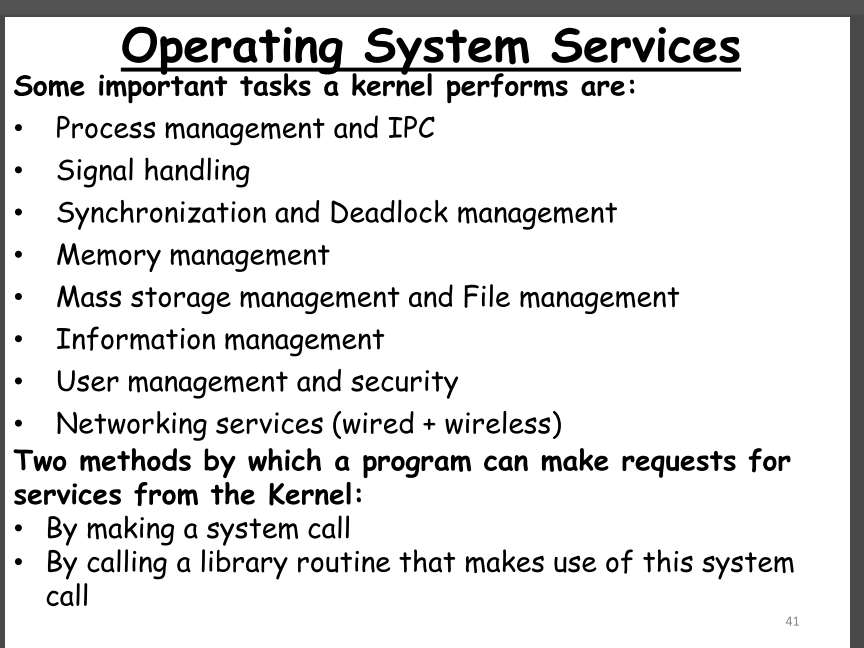
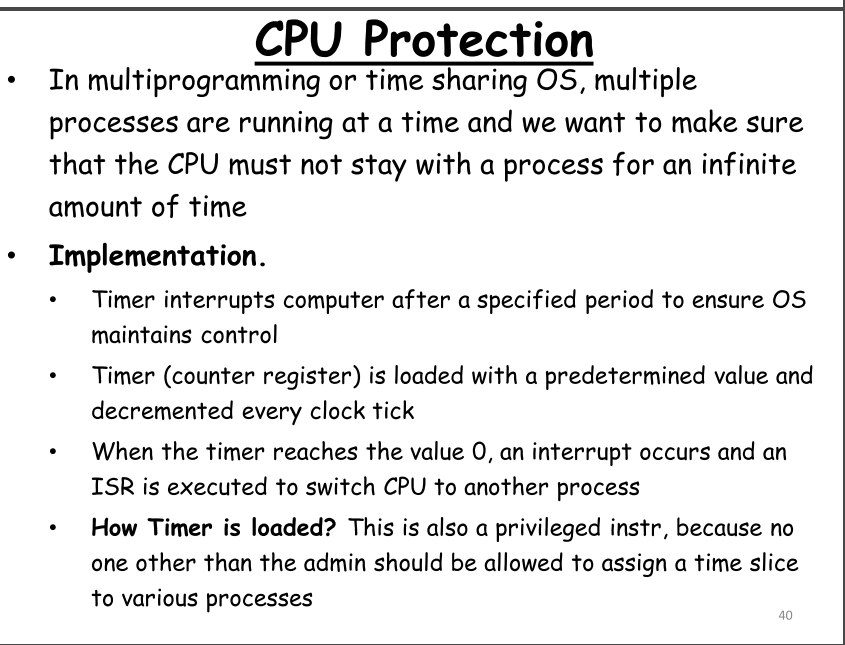
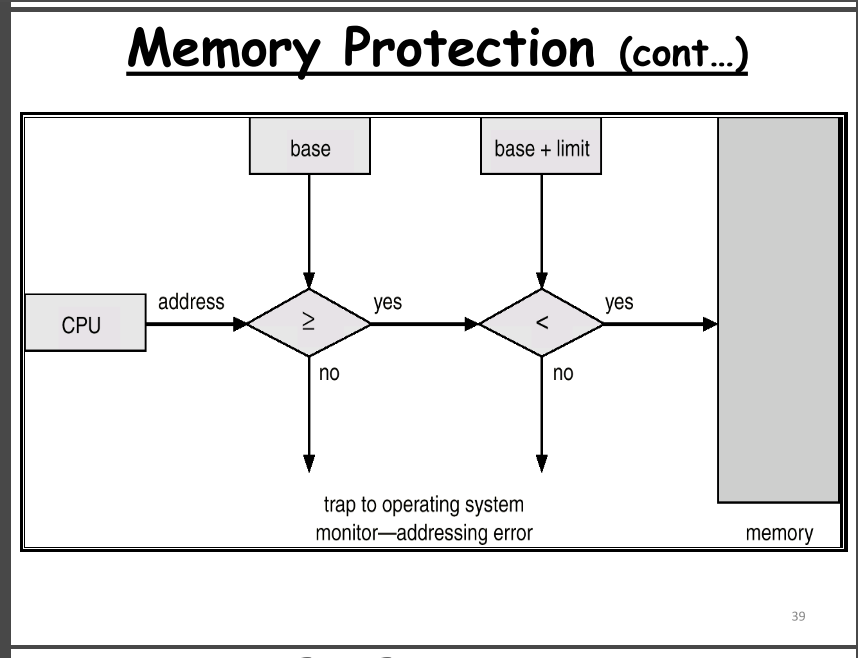
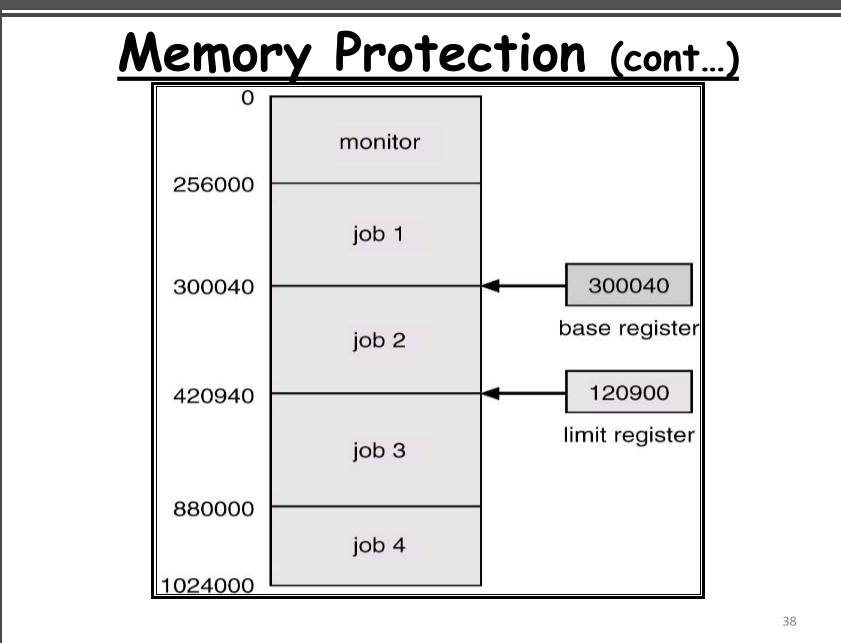
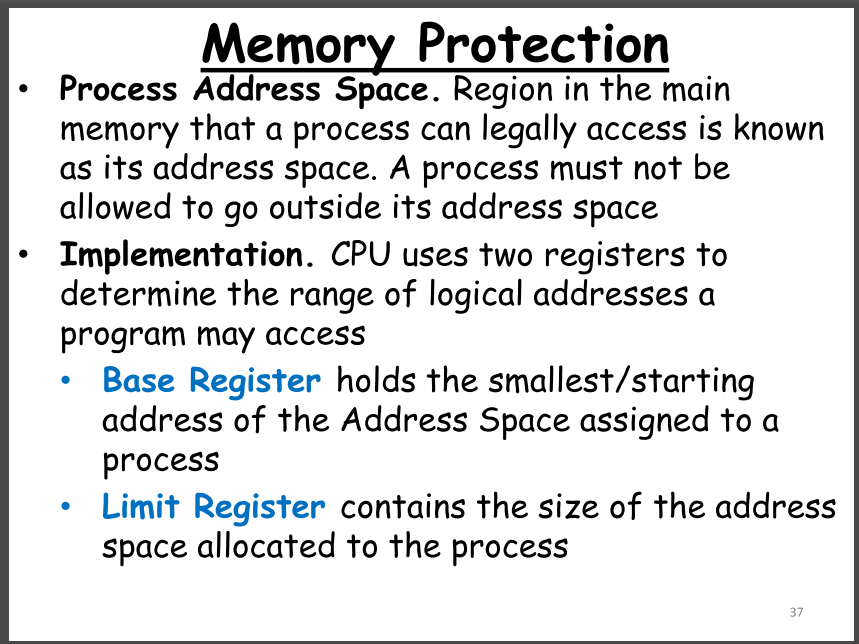
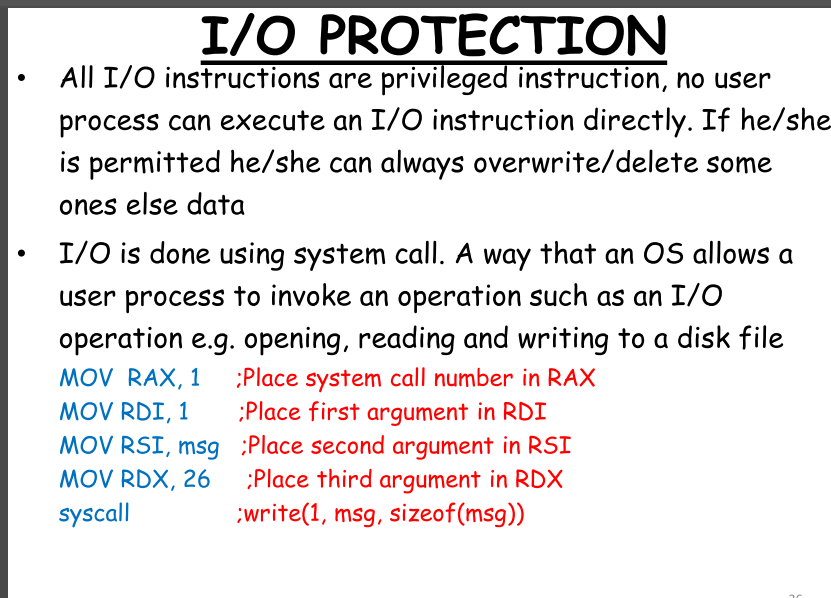
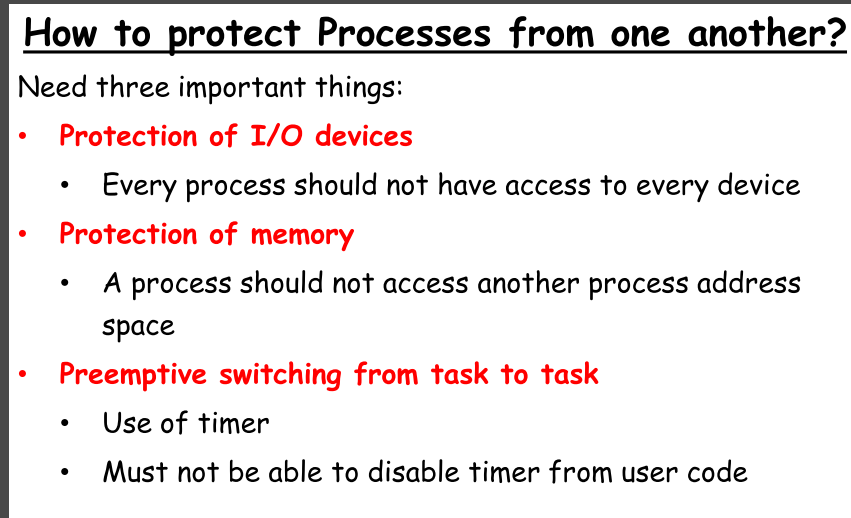
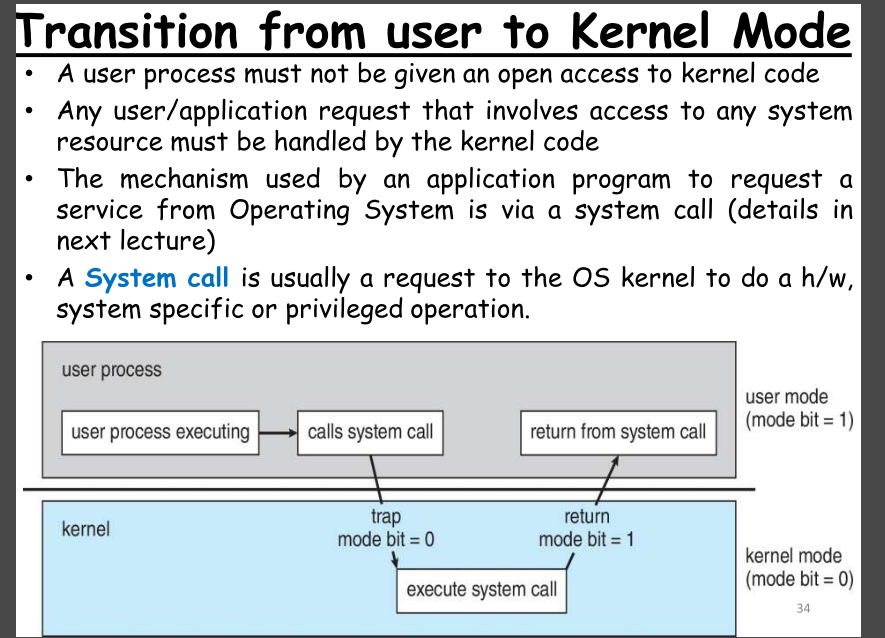
Each technique has its own use case and evolution, with **multiprogramming** being an early method, while **multitasking** and **time-sharing** are more advanced techniques suited to modern systems and multi-user environments.

****

**Issues of Timesharing:**

****

**Protection implemented by 2 modes: **



**TYPES OF OS:**