**User Mode & Kernel Mode**

**User mode**

In User mode, the executing code has no ability to directly access hardware or reference memory. Code running in user mode must use system APIs to access hardware or memory. Due to the protection by this sort of isolation, crashes in user mode are always recoverable. Most of the code running on your computer will execute in user mode.

* User mode is also known as the unprivileged mode, restricted mode, or slave mode
* Applications run in user mode
* User mode has a lower level of privileges than kernel mode
* Applications run with limited privileges to prevent direct access to hardware
* User mode prevents user processes from interfering with each other

**Kernel mode**

* Kernel mode is also known as the master mode, privileged mode, or system mode
* Core operating system components run in kernel mode
* Kernel mode has full privileges
* Kernel mode allows the kernel to manage hardware and system resources directly
* In kernel mode, the operating system has unrestricted access to all hardware resources
* In kernel mode, the applications have more privileges as compared to user mode

**Switching between modes**

* The transition between modes is called **mode switching**
* When a user mode process needs to perform a privileged operation like input/output operations, data writing in disk, or memory management, it makes a system call
* The operating system switches the processor from user mode to kernel mode to execute the system call
* After finally completing the execution of the process, the CPU again switches back to the user mode

A diagram of a trap

Description automatically generated

**Hardware Support Needed for Dual Mode**

1. **Mode Bit in CPU**
   * A special hardware **mode bit (0 for kernel mode, 1 for user mode)** helps the CPU determine whether the current process is running in user or kernel mode.
2. **Privileged Instructions**
   * Certain instructions (e.g., managing memory, I/O, and process scheduling) are **only executable in kernel mode**.
3. **Timer Interrupt**
   * Ensures that a user process **cannot control the CPU indefinitely**.
   * Helps the OS regain control by switching tasks periodically.

**Trap & Interrupt**

**Trap:** It is generated by the running program itself. It is synchronous with program execution. It allows a program to request services from the operating system safely (system calls) and to handle exceptions or errors that occur during program execution. It causes a switch to kernel mode. The instruction itself can be executed from user mode to request services from the OS. It is designed to be invoked by a user program.

**Traps** are **synchronous events (occur as a direct result of executing an instruction)** initiated by the program (either by design through system calls or by error conditions) and are handled immediately as part of the program’s execution flow. They provide a controlled way for programs to interact with the OS.

**Examples:**

* System calls (when a program intentionally invokes an OS service).
* Exceptions such as division by zero, invalid memory access, or other runtime errors.

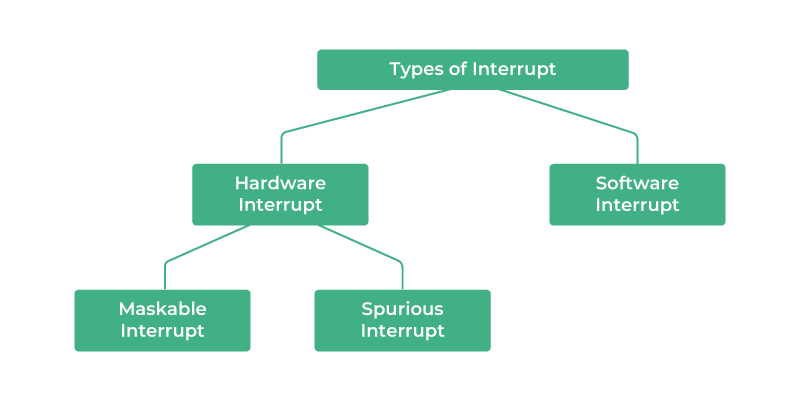
**Interrupts:** It is generated by hardware devices. It is asynchronous; they can occur at any time, regardless of the current instruction stream. It notifies the CPU of external events that require immediate attention and handles asynchronous events like I/O operations and hardware signals.

**Interrupts** are **asynchronous events (occur independently of the current process execution)** generated by hardware or external sources, signalling that some device or timer needs attention. They ensure that the CPU can react quickly to external events without constantly polling devices.

A hardware interrupt causes the processor to save its state of execution via a context switch and begin execution of an **interrupt handler**. Software interrupts are usually implemented as **instructions in the instruction set**, which cause a context switch to an interrupt handler similar to a hardware interrupt.

**Examples:**

* **A keyboard press.**
* **Completion of a disk I/O operation.**
* **Timer events (e.g., the system clock signalling a time slice expiration).**



There are multiple types of interrupts in computing, including hardware, software, external, internal, maskable, and non-maskable.

**Hardware interrupts**

* Triggered by external devices like keyboards, printers, and communication cards
* An electronic signal that indicates the device needs attention from the operating system
* For example, pressing a key or moving the mouse

**Software interrupts**

* Initiated by program instructions
* Allow different processes to interact with each other
* For example, system calls and exceptions
* Exceptions such as division by zero, invalid memory access, or other runtime errors.

**External interrupts**

* Generated by peripheral devices
* The CPU will execute the instructions from the external interrupt first

**Internal interrupts**

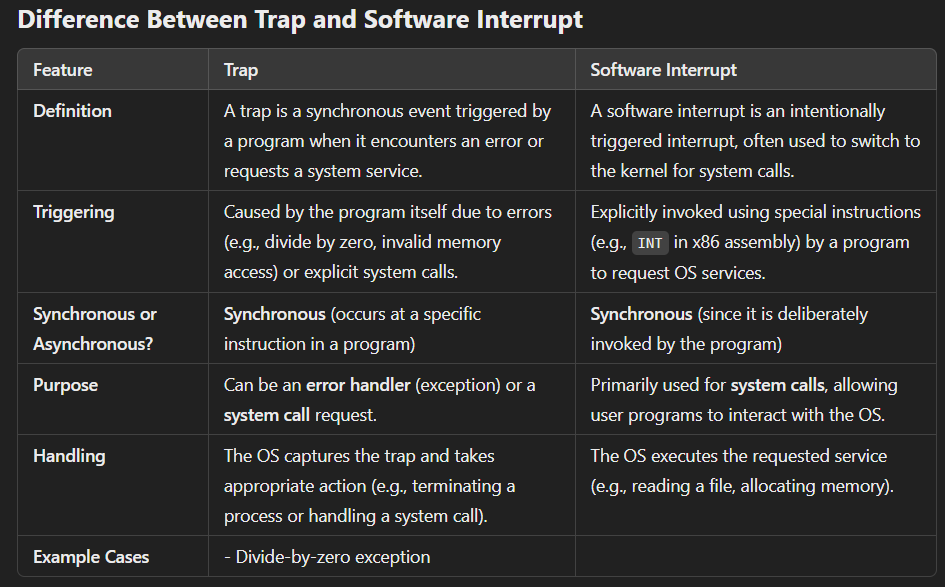
* Synchronous with the program
* If the program returns, the internal interrupts will occur in the same place each time

**Maskable interrupts**

* Can be enabled or disabled by the programmer

**Non-maskable interrupts**

* Have a higher priority and cannot be disabled
* Typically used for critical events like power loss



**-> Determine which Instructions are privileged or non-privileged.**

**Generate any trap instruction:**

* **Classification:** **Non-privileged**
* **Explanation:** A trap instruction (often used for system calls) is designed to be invoked by a user program. Although it causes a switch to kernel mode, the instruction itself can be executed from user mode to request services from the OS.

**Clear the Memory or Remove a Process from the Memory:**

* **Classification:** **Privileged**
* **Explanation:** These operations directly modify system memory or process management data. Allowing user programs to clear memory or remove processes could destabilize the entire system, so these instructions are restricted to kernel mode.

**Set Timer:**

* **Classification:** **Privileged**
* **Explanation:** Setting a hardware timer involves direct interaction with system hardware. Since this can affect the system’s scheduling and overall operation, only the operating system (kernel mode) is allowed to execute such instructions.

**Sending Final Printout:**

* **Classification:** **Non-privileged**
* **Explanation:** Typically, sending a final printout is a high-level operation (such as printing a document) that a user program requests. The request is handled via system calls and device drivers. Although the underlying I/O operations are privileged, the user-level command to "print" is non-privileged.

**I/O Instructions:**

* **Classification:** **Privileged**
* **Explanation:** Direct I/O instructions (which access hardware devices like disks, keyboards, printers, etc.) are restricted to kernel mode to protect hardware integrity and ensure system security. User processes must use system calls to request I/O operations rather than executing these instructions directly.

**Halt Instructions:**

* **Classification:** **Privileged**
* **Explanation:** The halt instruction stops the CPU. If a user-level process were allowed to execute a halt instruction, it could bring down the entire system. Therefore, such instructions are reserved for use in kernel mode only.

**Bootstrap**

A "bootstrap program" in an operating system is the very first piece of code that executes when a computer is turned on, essentially responsible for initializing the hardware and loading the necessary components to start the operating system itself; it's essentially the initial loader that sets the stage for the full OS to take over. It Runs only during system startup.

Key points about a bootstrap program:

* **Location:** Typically stored in the computer's **Read Only Memory (ROM)** or **Firmware**.
* **Function:**
  + Bootstrap performs basic hardware checks, initializes system components, and sets up the memory.
  + It then locates the operating system's boot loader on the storage device (like a hard drive) and loads it into memory.
  + Transfers control to the bootloader which then fully loads the operating system kernel into memory.

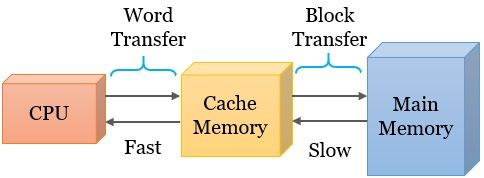
**Firmware**

* Firmware manages basic hardware operations like initialization, communication protocols, and device startup, providing a foundation for the OS to run applications on top. It runs continuously or when needed by the hardware.
* Unlike regular software on a hard drive, firmware is stored directly on the hardware chip (often called ROM).
* While some firmware is permanently written and can't be changed, many modern devices allow firmware updates to add new features or fix bugs.
* **BIOS (Basic Input/Output System):** This is a common example of firmware found in computers, responsible for initializing the hardware when the computer is turned on and loading the operating system.
* When a computer is powered on, the firmware runs the bootstrap program, which initializes the hardware and loads the OS into RAM. So, firmware is a broader term, while the bootstrap program is a specific function within firmware that helps boot the system.

**Cache Memory**

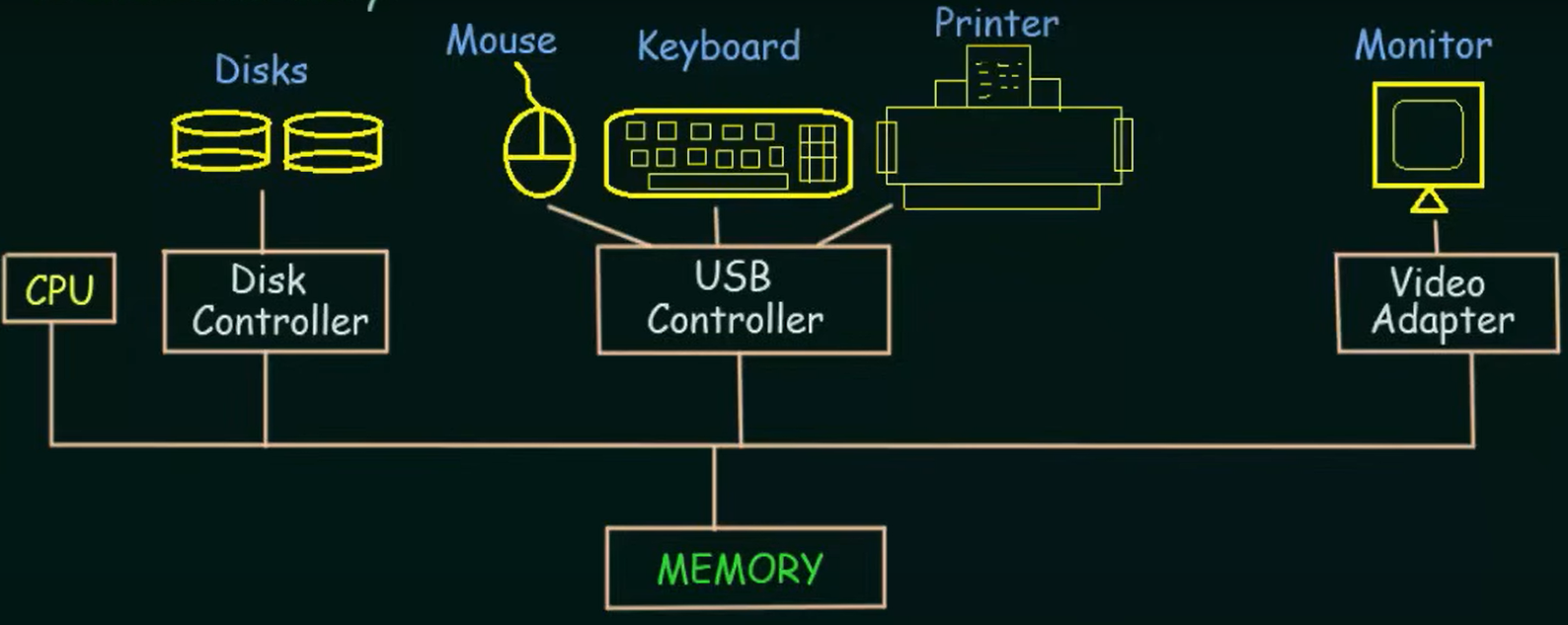
Cache memory is a temporary storage space for frequently accessed data in a computer. It's a type of random-access memory (RAM) that's located between the CPU and the main RAM. It stores copies of the data from frequently used main memory locations.

* This allows the CPU to access data faster than if it had to access it directly from the main memory.
* Reduces the frequency of accessing main memory, which can lead to faster data transfers.
* Cache memory is typically divided into three levels: L1, L2, and L3. Each level has varying sizes and speeds.
* Extremely fast memory type that acts as a buffer between RAM and the CPU.
* Holds frequently requested data and instructions, ensuring that they are immediately available to the CPU when needed.
* If the cache is full, it can slow down the device, cause apps to crash or not load properly, and lead to other performance issues.
* Cache Coherence: In systems with multiple caches (like multi-core processors), it's crucial to ensure that all caches have a consistent view of the data. This requires complex cache coherence protocols. When multiple caches store the same data, there's a risk of inconsistency. If one processor modifies the data in its cache, the other caches might have an outdated copy. This can lead to errors and unpredictable behavior.
* Cost: Cache memory is significantly more expensive per bit than the memory used in storage devices. Building a cache as large as a disk would be prohibitively expensive.
* Speed: As a cache gets larger, it can become slower to search. This can negate the benefits of the larger size, especially if the search time becomes comparable to the access time of the device being cached.
* **Power Consumption:** Larger caches consume more power, which can be a concern, especially in battery-powered devices.

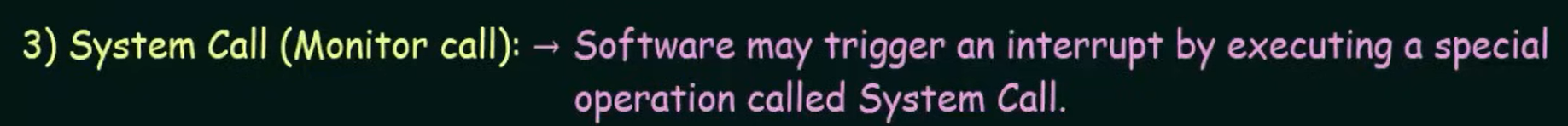
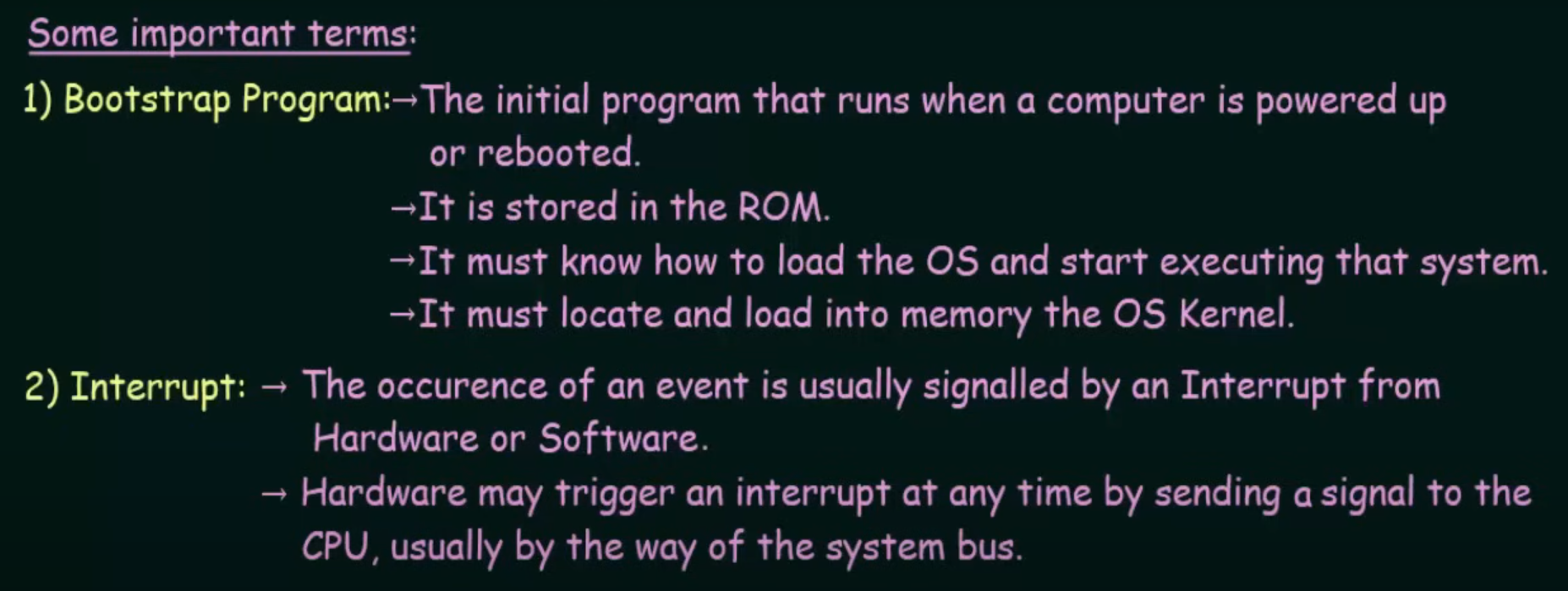


**Computer System Operations**

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



* Each device controller is in charge of a specific type of device
* The CPU and the device controllers can execute concurrently, competing for memory cycles
* To ensure orderly access to the shared memory, a memory controller is provided whose function is to synchronize access to the memory



**System Call**

A system call is made in user mode but is actually executed in kernel mode; when a program makes a system call, the CPU switches from user mode to kernel mode to allow the operating system to perform the requested operation with privileged access to system resources.

A system call triggers a special instruction called a **"trap"** or **"software interrupt"** which causes the CPU to switch modes from user mode to kernel mode, allowing the operating system to handle the requested system call operation; essentially, this is how a user program can request services from the OS.

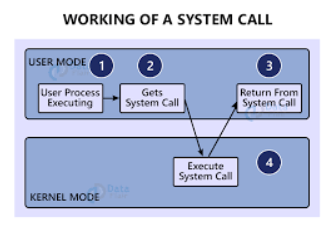
Once in kernel mode, the operating system kernel takes over, executes the requested system call operation (like opening/reading a file, opening a network connection), and then switches back to user mode.

System calls are used for **hardware-level programming** as they are faster.

The **main types** of system calls are **process control, file management, device management, information maintenance, and communication.**

System call examples

* **Process control**: Examples include fork(), exit(), and wait()
* **File management**: Examples include open(), read(), and write()
* **Device management**: Examples include ioctl(), read(), and write()
* **Information maintenance**: Examples include getpid(), alarm(), and sleep()
* **Communication**: Examples include system calls that send or receive messages between processes



**Differentiate between kernel and shell**

**-Kernel:** The core of the operating system that manages hardware resources (CPU, memory, devices) and enables communication between hardware and software.

**-Shell:** A user interface (command-line or graphical) that allows users to directly interact with the kernel i.e. using bash scripts in linux and run programs or commands.

**-Difference:** The kernel works directly with hardware, while the shell acts as a bridge between the user and the kernel.

**Interrupt Handling**

Interrupt handling is the process by which the CPU responds to and processes an interrupt signal, which can be generated by hardware (I/O devices, timers) or software (system calls, exceptions).

**1. Interrupt Occurs**

* An interrupt signal is generated by hardware (e.g., keyboard input, disk I/O completion) or software (e.g., system call, exception).
* The CPU detects the interrupt and pauses the currently executing process.

**2. CPU Saves Context (Registers & Program Counter)**

* The CPU **saves the current execution state** (registers, program counter, stack pointer) of the running process.
* This ensures that the process can resume execution after the interrupt is handled.

**3. Identify the Interrupt Source**

* The CPU checks the **Interrupt Vector Table (IVT)**, which contains the addresses of **Interrupt Service Routines (ISRs)** for different interrupt sources.

**4. Transfer Control to the ISR (Interrupt Service Routine)**

* The CPU jumps to the ISR address and begins executing the corresponding interrupt handler.
* The ISR processes the interrupt (e.g., handling keyboard input, reading data from a disk).

**5. Execute the ISR and Perform Required Task**

* The ISR handles the interrupt request (e.g., copies data from I/O buffer to memory, acknowledges hardware signals).
* If the ISR needs to send additional instructions, it may schedule another process or signal completion to the OS.

**6. Restore Context and Resume Execution**

* After the ISR completes, the saved **CPU context is restored**.
* The CPU resumes execution of the interrupted process or schedules another process if necessary.

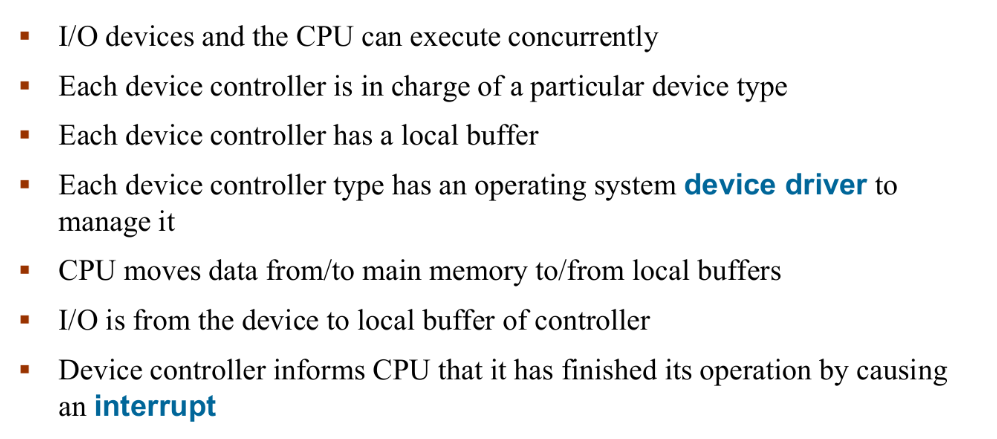
**Polling vs Vector Interrupt**

**Vector** **interrupts** allow the CPU to **only react when needed**, reducing CPU usage.

**Polling** requires the CPU to **continuously check** for events, wasting resources.

**Basic Operations**

• Device Controller



Device Driver for each device controller to manage I/O. Provides uniform interface between controller and kernel.

• Concurrent Execution