Interrupt

What are Interrupts?

In computing, an interrupt is a signal sent by a hardware device or a software process to the processor (CPU) of a computer, indicating that it requires attention or needs to be serviced. Interrupts are used to handle time-sensitive events or to respond to external events that require immediate action.

Possible causes of interrupt

Interrupts can be generated by various sources or events in a computer system. Here are some common causes of interrupts:

* Hardware Interrupts: These interrupts are generated by external hardware devices to signal events that require attention. Some examples include:
  + Keyboard or mouse input: When a user presses a key on the keyboard or moves the mouse, an interrupt is generated to handle the input.
  + Disk or network I/O completion: When a data transfer operation to or from a disk or network interface completes, an interrupt is generated to notify the CPU.
  + Timer or clock: A system timer generates interrupts at regular intervals to keep track of time, schedule tasks, or trigger time-sensitive operations.
  + Hardware Failures: Certain hardware failures, like memory errors or disk failures, can generate interrupts to alert the system about the issue, enabling error handling or system shutdown procedures to prevent data loss or damage
  + Power Management: Events related to power management, such as low battery alerts on laptops, can trigger interrupts to notify the operating system to save work or enter a lower power state.
* Software Interrupts: These interrupts are generated by software processes or the operating system to request specific services or actions. Some examples include:
  + System calls: Software programs invoke system calls to request services from the operating system, such as file operations, memory allocation, or process management.
  + Exception handling: Exceptions occur when the CPU encounters errors or exceptional conditions during program execution, such as divide-by-zero errors or invalid memory access and overflow errors
  + Inter-process Communication (IPC): Software interrupts can be used to signal events or communicate between different processes running on the system, facilitating synchronization and data exchange
* Internal Interrupts: These interrupts are generated by the CPU itself to handle specific events or conditions. Some examples include:
  + Instruction fetch: The CPU may generate interrupts to fetch instructions from memory when the instruction cache is empty or when executing branch or jump instructions.
  + Floating-point exceptions: When performing floating-point arithmetic, the CPU may generate interrupts to handle exceptional conditions, such as overflow or invalid operations.
* CPU Usage: Some systems have mechanisms to generate interrupts when certain performance thresholds are reached, such as CPU usage limits, enabling monitoring and management of system resources.
* Security Alerts: Hardware or software may generate interrupts in response to detected security breaches or suspicious activities by malicious agents attempting unauthorized intrusion, exploiting vulnerabilities, injecting viruses, launching denial-of-service attacks, tampering firmware images, extracting sensitive data, altering configurations, installing rootkits, escalating privileges, evading detection measures, or disrupting services.

Understanding the sources and types of interrupts is essential for system design, troubleshooting, and ensuring efficient and responsive computing environments. Handling interrupts properly allows operating systems and applications to react promptly to external events, manage resources efficiently, and maintain system stability and security.

Application of interrupts

* Device Input/Output (I/O):
* When a user interacts with input devices like keyboards or mice, interrupts are generated to handle the input events. This allows for real-time responsiveness and enables immediate processing of user commands.
* Disk drives use interrupts to signal the completion of data read/write operations, enabling the CPU to perform other tasks while waiting for I/O operations to complete.
* Real-Time Data Processing:

In systems like embedded systems or real-time computing applications (e.g., robotics, automotive control systems), interrupts ensure timely processing of real-time data, allowing the system to respond to external events within strict timing constraints.

* System Timing
* Clock Updates: Timer interrupts are used to keep the system clock updated, trigger scheduled tasks, or manage time slices in multitasking operating systems.
* Timeout Management: Timers can generate interrupts to monitor timeouts for operations, ensuring that a system can recover from stalls or hang-ups in software execution.
* Multitasking and Process Management
* In multitasking operating systems, interrupts are used to switch between processes, allocating CPU time efficiently among multiple tasks.
* Interrupts facilitate synchronization mechanisms between processes, allowing for efficient communication and data exchange among concurrent processes.
* Power Management   
  In mobile and portable computing devices, interrupts signal when to enter or exit power-saving modes based on user actions (e.g., opening a laptop lid) or system conditions (e.g., battery levels).
* Networking   
  Network interfaces use interrupts to notify the CPU of incoming data packets, enabling efficient processing of network communications without requiring the CPU to constantly poll the network interface.
* Hardware Malfunction Detection   
  Hardware components can generate interrupts to signal malfunctions or errors (e.g., memory errors, disk failures), allowing the system to take corrective actions, such as error logging or system shutdown to prevent data loss.
* Security   
  Some security hardware may generate interrupts upon detecting unauthorized access attempts or other security threats, triggering immediate responses from security software.
* Performance Monitoring   
  Systems can be configured to generate interrupts when certain performance thresholds are reached (e.g., CPU or memory usage), facilitating dynamic resource management and optimization.
* Debugging and Profiling:   
  Interrupts can be utilized for debugging and profiling purposes. They can be used to halt program execution at specific points, allowing developers to inspect program state and debug issues. Profiling tools can also leverage interrupts to periodically collect performance data.

**Interrrupt Service**

ISR stands for Interrupt Service Routine. It is a piece of code or a function that is executed in response to an interrupt. The ISR is responsible for handling the interrupt event and performing the necessary actions associated with it.

Here is a general overview of how the ISR is used in interrupt handling:

* An interrupt event is triggered by a hardware device, software process, or the CPU itself. The interrupt request is sent to the processor, indicating that attention is needed.
* The processor receives the interrupt request and identifies the corresponding interrupt source. The interrupt source can be identified through interrupt vectors, interrupt priority, or interrupt lines.
* The processor saves the current state of the program being executed, including the program counter, registers, and other relevant information. It then transfers control to the appropriate ISR associated with the specific interrupt.
* The ISR code is executed to handle the interrupt. The ISR performs the necessary actions based on the interrupt source and the specific requirements of the interrupt event. This can include tasks such as reading data from a device, updating system status, servicing I/O operations, or responding to exceptional conditions.
* After the ISR completes its execution, the processor restores the saved state of the interrupted program. This includes restoring the program counter, registers, and other relevant information. Control is then returned to the interrupted program, resuming its execution from the point it was interrupted.

When are interrupt detected during the fetch-execute cycle

Interrupts can theoretically be detected during any stage of the fetch-execute cycle, although they usually get caught right after fetching an instruction or just before starting its execution.

The precise timing of interrupt detection can vary depending on the architecture and design of the CPU.

Fetch Stage:

* + In the fetch stage, the processor fetches the next instruction from memory.
  + Interrupt detection can occur during this stage if the interrupt controller or interrupt-handling logic detects an incoming interrupt request. The interrupt request may come from an external device or from within the processor itself.

1. Decode Stage:
   * In the decode stage, the fetched instruction is decoded to determine the operation to be performed.
   * Interrupt detection can occur during this stage if the hardware identifies an interrupt instruction or a specific bit pattern that indicates an interrupt request.
2. Execute Stage:
   * In the execute stage, the decoded instruction is executed, and the corresponding operation is performed.
   * Interrupt detection can occur during this stage if the processor is designed to check for interrupt requests while executing instructions.
3. Completion of Instruction:
   * Interrupt detection can also occur after the completion of the current instruction.
   * The processor may check for interrupt requests at the end of each instruction or at specific points during instruction execution.

When an interrupt is detected, the processor will initiate the interrupt handling process, which involves saving the current context, transferring control to the Interrupt Service Routine (ISR), executing the ISR code, and eventually returning to the interrupted program after the ISR completes

How are interrupt handled

The process of handling interrupts involves several key steps, from the initial detection of an interrupt request (IRQ) to the execution of a specific routine designed to address the interrupt, and finally, the return to the original process. Here's an overview of how interrupts are handled:

* When an interrupt occurs, it is detected by the interrupt controller or interrupt-handling logic. The interrupt can be triggered by an external hardware device, a software-generated interrupt, or an internal event.
* Before handling the interrupt, the processor saves the current context of the interrupted program. This includes the program counter, register values, and other relevant information. The context is typically saved in a designated area of memory or in processor registers.
* The processor acknowledges the interrupt source, indicating that it has recognized the interrupt request. This acknowledgment may involve sending a signal or response to the interrupt source to confirm that the interrupt has been received.
* Each type of interrupt is associated with a specific Interrupt Service Routine (ISR), a function designed to handle the interrupt. The CPU uses an **Interrupt Vector Table (IVT)** or a similar mechanism to map each interrupt to its corresponding ISR. The IVT contains addresses of ISRs for all the recognized interrupts.
* The processor transfers control to the ISR by jumping to the address indicated by the interrupt vector. The ISR performs the necessary actions associated with the interrupt, such as reading data from a device, updating system state, or responding to exceptional conditions.
* The appropriate actions are taken in the ISR to handle the interrupt. This can include tasks such as servicing I/O requests, processing data, updating system status, or performing specific operations required by the interrupt event.
* After the ISR completes execution, the interrupt is cleared or reset. This involves performing any necessary actions to acknowledge the completion of the interrupt handling process. For example, clearing interrupt flags or resetting interrupt requests.
* Once the interrupt is handled, the processor restores the saved context of the interrupted program. This includes restoring the program counter, register values, and other relevant information.
* With the context restored, the processor resumes execution of the interrupted program from the point where it was interrupted. The program continues its normal flow as if the interrupt had not occurred.