Interrupts

Polling Analogy: Waiting at the Door

- **Scenario**: You're expecting guests to arrive at your door, and you're continuously going to check if they're there.
- **Explanation**: This is like polling in computing. In the context of computing, polling is a technique where the system checks the status of a device or process at regular intervals. It's like repeatedly going to the door to see if the guests have arrived. However, this method can be inefficient because you might spend a lot of time checking when nothing has changed.

Interrupt Analogy: Doorbell Rings for Your Crush

- **Scenario**: You're engaged in an activity, but suddenly you hear your crush's voice, and it makes you stop what you're doing and go outside.
- **Explanation**: This is similar to interrupts in computing. Interrupts are signals that "interrupt" the normal flow of a program to handle a specific event. In the analogy, the crush's voice is the interrupt. When you hear it, you stop your current activity (interrupting it) and go outside (handle the interrupt). In computing, this is an efficient way to respond to external events without constantly checking.

Linux Interrupt Handling Analogy: Responding to the Doorbell

- **Scenario**: Guests are arriving, and you have tasks to do inside. You only go to the door when the doorbell rings, which serves as a signal that guests are there.
- **Explanation**: In Linux, interrupt signals are like the doorbell ringing. The processor is working on its tasks (similar to your preparations or unrelated work), and when an interrupt occurs (doorbell rings), it stops its current work to handle the interrupt. The new activity it engages in (opening the door) is called the interrupt handler or interrupt service routine (ISR). This ensures that Linux can respond quickly and efficiently to external events without constantly checking for them.

These analogies make the concepts of polling and interrupts more relatable for those who may not be familiar with computer systems.

What will happen when the interrupt comes?

Generation of Interrupt:

- **Source**: Interrupts are generated by electronic signals from hardware devices. These signals are sent to input pins on an interrupt controller.
- **Interrupt Controller**: The interrupt controller is a chip that consolidates multiple interrupt lines into a single line to the processor. It helps manage and prioritize various interrupt signals.

Initiation of Interrupt Handling:

- **Signal to Processor**: Upon receiving an interrupt from the interrupt controller, a signal is sent to the processor.
- **Processor Response**: The processor detects this interrupt signal and interrupts its current execution to address the interrupt. This is akin to the processor momentarily pausing its ongoing tasks to deal with the incoming interrupt.

Notification to Operating System:

- **Interrupt Acknowledgment**: The processor acknowledges the interrupt and may notify the operating system that an interrupt has occurred.
- **Operating System Involvement**: The operating system becomes involved in handling the interrupt. It needs to identify the type of interrupt and determine the appropriate action to take.

Differentiation and Handling:

- **Unique Values for Interrupts**: Each interrupt is associated with a unique value. This allows the operating system to distinguish between different types of interrupts and identify the hardware device that triggered the interrupt.
- **Interrupt Handlers**: The operating system has specific routines called interrupt handlers or interrupt service routines (ISRs) for each type of interrupt. These handlers are responsible for managing the specific actions required when a particular interrupt occurs.

Considerations in Interrupt Handling:

- **Asynchronous Nature**: Interrupts can occur asynchronously, meaning they can happen at any time, independent of the processor clock. The system must be prepared to handle interrupts whenever they occur.
- **Concurrent Interrupts**: Given the asynchronous nature of interrupts, the kernel might be in the process of handling one interrupt when another interrupt of a different type occurs. The system must manage these concurrent interrupts effectively.
- **Critical Regions**: There are critical regions within the kernel code where interrupts must be disabled temporarily. This is to prevent potential conflicts or issues when executing certain sensitive operations. However, these critical regions should be minimized to avoid hindering system responsiveness.

In summary, interrupt handling is a crucial aspect of the kernel's operation, ensuring that the system can respond promptly and appropriately to external events triggered by hardware devices.

It involves the coordination of the interrupt controller, processor, and the operating system to manage interrupts and execute the necessary actions through specialized interrupt handlers.

Critical Regions:

Definition:

- Critical regions refer to sections of the kernel code where specific operations are being performed, and it is crucial that these operations proceed without interruption or interference.

Why Interrupts are Disabled:

Data Consistency:

- In some operations, especially those involving shared data structures or variables, it's crucial to maintain data consistency.
- Disabling interrupts ensures that the execution of the critical section remains uninterrupted, preventing potential conflicts when multiple parts of the system attempt to access or modify shared data simultaneously.

Atomicity:

- Certain operations need to be atomic, meaning they should be executed as a single, indivisible unit.
- Disabling interrupts helps achieve atomicity by preventing interruptions between the different steps of an operation.

Avoiding Race Conditions:

- Race conditions occur when the behavior of a program depends on the relative timing of events.
- By disabling interrupts, the kernel aims to prevent race conditions that could arise if an interrupt were to occur at an inopportune moment during the execution of a critical section.

Hardware Interaction:

- Some operations involve direct interaction with hardware registers or devices.
- Disabling interrupts ensures that the processor has exclusive control over the hardware during these operations.

Importance of Minimizing Critical Regions:

System Responsiveness:

- The more time interrupts are disabled, the longer the system is unable to respond to external events.
- Minimizing critical regions is essential to maintain system responsiveness, allowing the kernel to promptly handle interrupts and respond to time-sensitive tasks.

Concurrency and Multitasking:

- In a multi-tasking environment, where multiple processes or threads are running concurrently, lengthy critical regions can lead to delays in handling other tasks.

- Minimizing critical regions supports effective concurrency, allowing the system to switch between tasks more frequently.

Reducing Deadlocks:

- Prolonged disabling of interrupts increases the risk of deadlocks, where different parts of the system are waiting for resources held by each other.
- Minimizing critical regions reduces the likelihood of deadlocks and enhances overall system stability.

Enhancing Scalability:

- Systems with minimized critical regions are often more scalable. They can efficiently handle increased loads and adapt to varying workloads without sacrificing responsiveness.

In summary, while critical regions with disabled interrupts are necessary for certain sensitive operations, it is crucial to strike a balance. Minimizing the duration of critical regions helps ensure that the system remains responsive, can handle multiple tasks concurrently, and is less prone to issues such as deadlocks. This balance is especially important in modern computing environments where responsiveness and scalability are key considerations.

Interrupts and Exceptions

Interrupts are events that occur asynchronously, independent of the processor clock. They are typically generated by external hardware devices to request attention from the processor. Examples include signals from I/O devices, timer expiration, or hardware errors.

Characteristics:

- **Asynchronous Nature:**
- Interrupts can happen at any time, and the processor must be prepared to handle them whenever they occur.
- They are initiated by external hardware, and their timing is not synchronized with the processor's clock.
- **External Hardware Source:**
- Interrupts are generated by external hardware devices such as keyboards, mice, disk drives, etc.
- The purpose is to notify the processor about an event that requires its attention.
- **Interrupt Controller:**
- An interrupt controller is used to manage and prioritize multiple interrupt signals from various hardware devices.

Exceptions:

Definition:

Exceptions are events that occur synchronously with respect to the processor clock. They are often referred to as synchronous interrupts because they are produced by the processor itself during the execution of instructions.

Characteristics:

- **Synchronous Nature:**
- Exceptions occur in response to specific conditions that arise during the execution of instructions, and they are synchronous with the processor clock.
- **Processor-Generated:**
- Exceptions are produced by the processor itself. They can be triggered by various events, including programming errors (e.g., divide by zero) or abnormal conditions that require intervention (e.g., a page fault).
- **Similar Handling to Interrupts:**
- Many processor architectures handle exceptions in a manner similar to interrupts. The kernel infrastructure for handling exceptions often shares similarities with interrupt handling.

Relationship between Interrupts and Exceptions:

- **Similar Handling by the Kernel:**
- While interrupts are generated by external hardware, and exceptions are produced by the processor, the kernel often handles both in a similar way.
- Both interrupts and exceptions involve interrupt handlers or interrupt service routines (ISRs) that the kernel executes to respond appropriately to the event.
- **Examples of Exceptions:**
- **System Calls:** System calls are a type of exception. On x86 architecture, a software interrupt is issued to trap into the kernel, leading to the execution of a special system call handler.
- **Abnormal Conditions:** Exceptions also handle abnormal conditions like division by zero or page faults.
- **Classification:**
- Both interrupts and exceptions can be further classified based on their source or nature.

Interrupts: Hardware-generated, asynchronous.
Exceptions: Processor-generated, synchronous.

In summary, interrupts and exceptions are both mechanisms by which a computer system can respond to external events or abnormal conditions. While interrupts are asynchronous and initiated by external hardware, exceptions are synchronous and generated by the processor itself during the execution of instructions. Despite their differences, the kernel often employs similar infrastructure to handle both interrupts and exceptions.

Maskable Interrupts:

- **Definition:**

- **Maskable interrupts** are generated by I/O devices, and they can be in one of two states: masked or unmasked.
- If an interrupt is masked, the control unit ignores it, allowing the system to focus on other tasks. If unmasked, the control unit recognizes and processes the interrupt.
- **Usage:**
- Maskable interrupts are typically used for events that the system can choose to temporarily ignore, giving priority to other tasks. For example, an I/O device may generate an interrupt, but the system might choose to mask it if it is currently handling a more critical task.

Non-maskable Interrupts:

- **Definition:**
- **Non-maskable interrupts (NMI)** are reserved for critical events that the system cannot afford to ignore. These events often include hardware failures or other serious issues.
- Unlike maskable interrupts, NMIs are always recognized by the CPU, and their processing cannot be delayed.
- **Usage:**
- NMIs are employed for events that demand immediate attention, such as hardware malfunctions. Since they cannot be masked, the CPU must respond promptly.

Exceptions:

1. **Faults:**
   - **Definition:**
   - **Faults** are exceptions that occur due to errors in the execution of instructions. Examples include:
   - **Divide by Zero:** Occurs when the CPU encounters a division by zero in a program.
   - **Page Fault:** Happens when a program tries to access a page of memory that is not currently in physical RAM.
   - **Segmentation Fault:** Arises when a program attempts to access a restricted area of memory.
   - **Handling:**
   - Faults usually require intervention by the operating system to resolve the error condition and allow the program to continue executing.
2. **Traps:**
   - **Definition:**
   - **Traps** are exceptions that are reported immediately after the execution of the trapping instruction.
   - Examples include breakpoints or system calls.
   - **Usage:**
   - Traps are often used for specific conditions that require special handling. For instance, a breakpoint trap might be used for debugging purposes, allowing the system to pause and inspect the program at a particular point.

3. **Aborts:**

- **Definition:**
- **Aborts** are used to report severe errors, such as hardware failures or invalid values in system tables.
- **Usage:**
- Aborts are typically employed for conditions that indicate a critical failure or inconsistency in the system. The handling of aborts often involves shutting down the system or taking corrective actions.

Interrupt Handlers in Device Drivers:

- **Role:**
- For each interrupt that a device can generate, its corresponding device driver registers an interrupt handler.
- The interrupt handler is a piece of code that specifies how the system should respond when a specific interrupt occurs.
- **Functionality:**
- When an interrupt is triggered by a device, the associated interrupt handler in the device driver is called.
- The interrupt handler manages the immediate response to the interrupt, ensuring that the necessary actions are taken.

In summary, interrupts are events that disrupt the normal execution of a program, and exceptions are mechanisms to handle errors or specific conditions during program execution. Maskable interrupts can be ignored or delayed, while non-maskable interrupts demand immediate attention. Faults, traps, and aborts are different types of exceptions, each serving a specific purpose in handling errors or exceptional conditions. Interrupt handlers in device drivers play a crucial role in managing the response to interrupts generated by I/O devices.

Interrupt handler

Interrupt Handler Basics:

- **Definition:**
- An **interrupt handler** or **interrupt service routine (ISR)** is a function that the kernel executes in response to a specific interrupt generated by a hardware device.
- **Association with Devices:**
- Each device that is capable of generating interrupts has an associated interrupt handler.
- The interrupt handler for a particular device is part of the device's driver, which is the kernel code responsible for managing that device.
- **Implementation in Linux:**
- In Linux, interrupt handlers are implemented as normal C functions.

- They must adhere to a specific prototype, allowing the kernel to pass information to the handler in a standard way.
- **Context of Execution:**
- Interrupt handlers run in a special context known as **interrupt context**.
- This context is sometimes referred to as **atomic context** because code executing in this context is unable to block. This means that it cannot be preempted or put to sleep.

Responsibilities of Interrupt Handlers:

- **Quick Execution:**
- Interrupt handlers must execute quickly to minimize the delay in servicing the interrupt.
- Since an interrupt can occur at any time, it is crucial for the handler to run efficiently and resume the execution of the interrupted code as soon as possible.
- **Acknowledging Interrupts:**
- At the very least, an interrupt handler's primary responsibility is to acknowledge the receipt of the interrupt to the hardware.
- This acknowledgment informs the hardware that the interrupt has been recognized and is being processed by the operating system.
- **Timely Service:**
- Interrupt handlers should service the interrupt promptly to meet two critical requirements:
- **To the Hardware:** The operating system must service the interrupt without delay to maintain proper functioning of the hardware.
- **To the Rest of the System:** The interrupt handler should execute in as short a period as possible to minimize the impact on the overall system's responsiveness.
- **Varied Workloads:**
- While acknowledging the interrupt is a minimal requirement, interrupt handlers may have additional tasks to perform.
- Depending on the nature of the interrupt and the associated device, interrupt handlers can have a varying amount of work to complete.

Significance:

- **Real-Time Responsiveness:**
- The efficiency of interrupt handlers is crucial for maintaining real-time responsiveness in the system.
- Quick execution ensures that the system can promptly respond to external events signaled by interrupts.
- **Hardware Interaction:**
- Interrupt handlers play a vital role in managing the interaction between the operating system and hardware devices.
- They facilitate the seamless flow of information and actions between the software and hardware layers of the system.
- **Interrupt Context Constraints:**

- Understanding the constraints of interrupt context, where blocking is not allowed, is essential for designing efficient interrupt handlers.

In summary, interrupt handlers are essential components in the interaction between the operating system and hardware devices. They execute in a special context, responding quickly to acknowledge interrupts, and play a critical role in maintaining the real-time responsiveness and proper functioning of the system.

What is difference between the atomic context and process context?

Atomic Context:

- In an atomic context, the execution of code is guaranteed to be uninterrupted, i.e., it cannot be preempted by another interrupt. This means that once the code starts executing, it will run to completion without being interrupted by other interrupts.
- During an atomic context, the system typically disables interrupts temporarily to ensure that the current code can execute without being preempted.
- Code executed in an atomic context is often referred to as a "critical section." It is important to keep critical sections short to minimize the impact on system responsiveness.

Process Context:

- In contrast, process context refers to the normal execution context of a process or task within the operating system. This context includes the execution of user-level code and system-level code associated with a specific process.
- When an interrupt occurs, it may cause a switch from process context to interrupt context. The interrupt handler, which runs in interrupt context, deals with the interrupt and may perform actions that are not suitable for the process context.
- Process context is characterized by the execution of user-level code and system-level code within the context of a specific process or task.

During the handling of an interrupt, the system often transitions between these two contexts. The interrupt handler typically runs in atomic context, ensuring that it completes its execution without being preempted by other interrupts. After the interrupt is handled, the system returns to the process context, allowing the interrupted process to continue its execution.

It's worth noting that the exact details may vary depending on the specific operating system and architecture. The concepts of atomic context and process context are essential for understanding how interrupt handling works in an operating system.

Limitations of Interrupt Context:

- **No Sleep or Relinquish:**
- Code running in interrupt context cannot go to sleep or relinquish the processor. It must complete its execution before allowing other tasks.

- **No Mutex Acquisition:**
- Interrupt context code cannot acquire a mutex, as it may lead to deadlocks or other synchronization issues.
- **No Time-Consuming Tasks:**
- Time-consuming tasks are discouraged in interrupt context to ensure that interrupts are handled promptly.
- **No Access to User Space Memory:**
- Code in interrupt context cannot directly access user space virtual memory, as this could lead to security and stability issues.

Problem with Lengthy Interrupt Handling:

- **Issue:**
- If an interrupt handler takes a significant amount of time to execute (a long duration), it can lead to problems:
- Other interrupts of the same type might be missed.
- While the highest-priority interrupt is running, it prevents other interrupts from being processed.

Solution: Splitting Interrupt Processing: Top Halves and Bottom Halves

Interrupts: https://linux-kernel-labs.github.io/refs/heads/master/lectures/interrupts.html