**Interrupt Handling**

**Understanding Interrupts and IRQs**

**Interrupts** are signals sent to the processor by hardware or software indicating an event that needs immediate attention. The processor responds by pausing its current tasks, saving its state, and executing a function called an interrupt handler to address the event.

**IRQs (Interrupt Requests)** are hardware lines over which devices can send interrupt signals to the processor. Each IRQ line is assigned a specific priority level, and when an interrupt occurs, the processor uses this priority to decide which interrupt to service first if multiple interrupts occur simultaneously.

**Real-world example**:

* **Keyboard Input**: When you press a key, the keyboard sends an interrupt signal to the processor, causing it to temporarily halt its current tasks to process the keystroke.
* **Network Packet Arrival**: When a network interface card receives a packet, it sends an interrupt to the CPU to process the incoming data.

**IRQ Numbers and IRQ Descriptors**

Each IRQ is assigned a unique number known as an IRQ number, which identifies the interrupt source. **IRQ descriptors** are data structures used by the operating system to manage and handle these interrupts. They contain information about the interrupt handler, device context, and other metadata.

**Real-world example**:

* **Mouse Movement**: The mouse might be assigned IRQ number 12. The IRQ descriptor for this IRQ would contain details about the mouse driver and the function to execute when a mouse interrupt occurs.

**Top Half and Bottom Half Handlers**

Interrupt handling is typically divided into two parts:

* **Top Half**: The initial interrupt handler that runs immediately when an interrupt is received. It performs minimal, time-critical tasks and schedules the bottom half.
* **Bottom Half**: Deferred processing that handles less urgent tasks, allowing the system to quickly return to handling other interrupts.

**Real-world example**:

* **Disk I/O**: When a disk read operation completes, the top half handler might acknowledge the interrupt and schedule the bottom half. The bottom half then processes the data read from the disk and updates the filesystem structures.

**Writing an Interrupt Handler**

An interrupt handler is a special function written to handle a specific interrupt. It should be designed to be efficient and execute quickly to avoid delaying other interrupts.

**Real-world example**:

* **GPIO Pin Change**: For an embedded system, you might write an interrupt handler that executes whenever a GPIO pin state changes. This could be used to detect button presses or sensor signals.

**request\_irq(), free\_irq()**

In many operating systems, especially Linux, the request\_irq() function is used to register an interrupt handler for a specific IRQ number. The free\_irq() function is used to deregister the handler when it is no longer needed.

**Real-world example**:

* **Driver Development**: When writing a device driver for a new hardware component, you would use request\_irq() to register your handler when the device is initialized and free\_irq() when the device is shut down.

**Shared Interrupt Lines**

Some systems allow multiple devices to share a single IRQ line. The interrupt handler must determine which device triggered the interrupt and handle it appropriately.

**Real-world example**:

* **PCI Devices**: In systems with limited IRQ lines, multiple PCI devices might share an IRQ. The interrupt handler would need to check each device to see which one raised the interrupt.

**Application in Real-World Scenarios**

* **Embedded Systems**: Handling sensor inputs, button presses, or communication events often relies on efficient interrupt handling to ensure timely responses.
* **Operating Systems**: Managing hardware resources, responding to user inputs, and maintaining system stability require sophisticated interrupt handling mechanisms.
* **Device Drivers**: Writing drivers for new hardware involves setting up and managing interrupts to ensure the hardware operates correctly and efficiently.

Top of Form

**Assignment: Implementing an Interrupt Handler in a Linux Kernel Module**

**Objective**

The objective of this assignment is to understand and implement basic interrupt handling in a Linux kernel module. You will write a kernel module that handles interrupts from a simulated hardware device.

**Steps to Complete the Assignment**

1. **Setup Environment**: Ensure you have a Linux development environment set up with kernel headers installed.
2. **Understand the IRQ Number**: Identify an IRQ number to use for simulation (for real hardware, you would get this from the device datasheet or system configuration).
3. **Write the Kernel Module**: Implement the kernel module to handle the interrupt.
4. **Test the Module**: Insert and remove the module, simulating an interrupt to verify your handler works correctly.

**Instructions**

1. **Setup Environment**:
   * Install necessary packages: build-essential, linux-headers-$(uname -r)
   * Ensure you have root privileges for module operations: sudo su
2. **Write the Kernel Module**:

//my\_interrupt\_module.c:

#include <linux/module.h>

#include <linux/kernel.h>

#include <linux/interrupt.h>

#include <linux/gpio.h>

#define IRQ\_NUMBER 1 // This is an example; we can use a different IRQ number

#define GPIO\_PIN 17 // Example GPIO pin for simulation

static irqreturn\_t my\_interrupt\_handler(int irq, void \*dev\_id) {

printk(KERN\_INFO "Interrupt handled: IRQ %d\n", irq);

return IRQ\_HANDLED;

}

static int \_\_init my\_module\_init(void) {

int result;

result = request\_irq(IRQ\_NUMBER, my\_interrupt\_handler, IRQF\_SHARED, "my\_interrupt", (void \*)(my\_interrupt\_handler));

if (result) {

printk(KERN\_ERR "Failed to request IRQ %d\n", IRQ\_NUMBER);

return result;

}

if (!gpio\_is\_valid(GPIO\_PIN)) {

printk(KERN\_ERR "Invalid GPIO pin %d\n", GPIO\_PIN);

return -ENODEV;

}

gpio\_request(GPIO\_PIN, "sysfs");

gpio\_direction\_input(GPIO\_PIN);

gpio\_export(GPIO\_PIN, false);

printk(KERN\_INFO "Module loaded and interrupt handler registered.\n");

return 0;

}

static void \_\_exit my\_module\_exit(void) {

free\_irq(IRQ\_NUMBER, (void \*)(my\_interrupt\_handler));

gpio\_unexport(GPIO\_PIN);

gpio\_free(GPIO\_PIN);

printk(KERN\_INFO "Module unloaded and interrupt handler unregistered.\n");

}

MODULE\_LICENSE("GPL");

MODULE\_AUTHOR("Your Name");

MODULE\_DESCRIPTION("A simple Linux kernel module to handle interrupts.");

MODULE\_VERSION("1.0");

module\_init(my\_module\_init);

module\_exit(my\_module\_exit);

Bottom of Form