INTRODUCTION

What is linux?

Linux is a free and open source operating system(OS) that is based on the Unix-like system(it follows the unix philosophy). It powers everything from personal computers and servers to embedded systems and iot devices.

Unix philosophy:

- Do one thing and do it well: Each program and tool should focus on doing one task efficiently and effectively.
- Work together: One's output should be designed to interoperate. Example if we have a one file it should be used as input to other work.
- Data should be plain text: Data should be stored and manipulated in a text because plain text is easy to read, debug and process.
- Small and modular tools: Instead of building a large program for performing task build the small tools that can combine to perform complex task.example ls | grep “error” | wc -l
- Avoid cluster: keep the things simple and avoid unnecessary features or complexity.tools should work without excessive setup or learning curves.example rm command for deleting a files doesn’t have flashy features -its just remove files.
- Fail Early and Clearly: If something goes wrong, the program should fail quickly and provide clear error messages.

Key Characteristics:

1.The linux kernel is the core of the OS managing hardware resources and enabling software to interact with hardware.

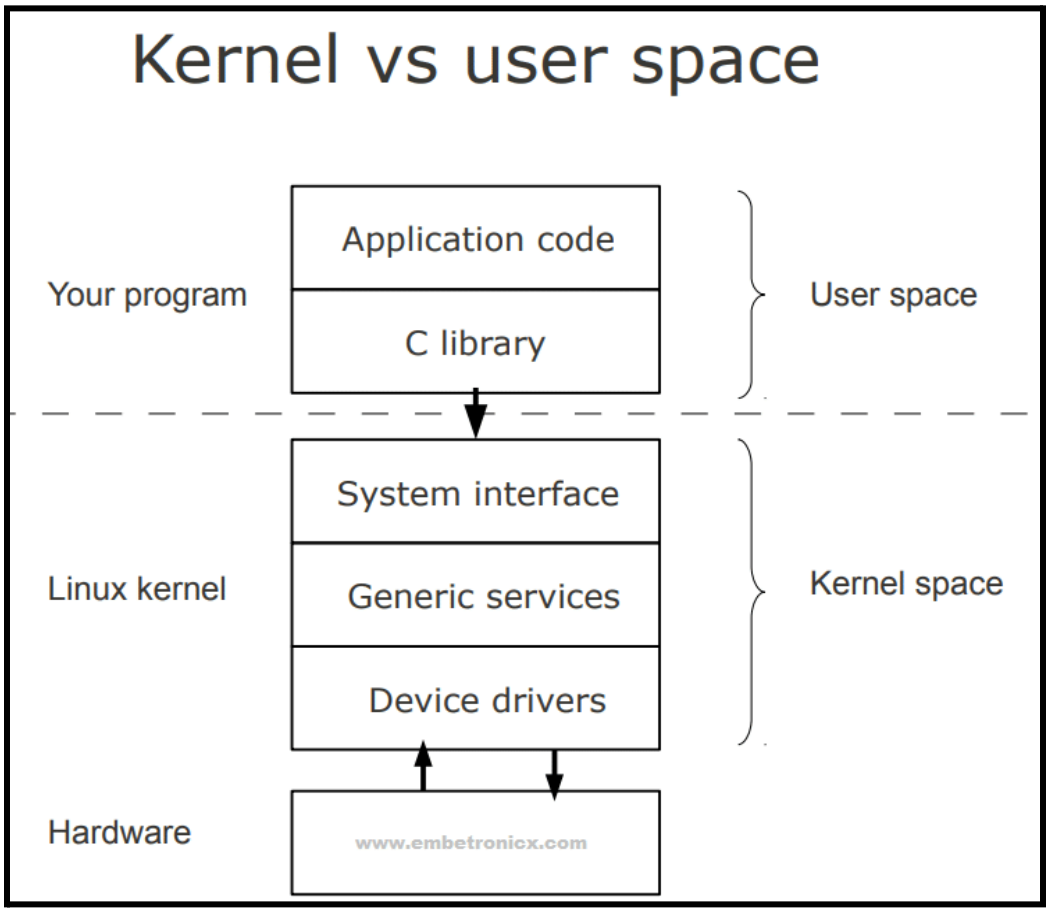
2.The linux is open source and customizable. We can modify and recompile the linux kernel to suit our needs.

3.So it has cross compilation support that runs on various hardware architectures like x86, ARM, RISC - V, etc. Example is that beagle bone black which is a compatible development board to run the linux.

So the users can modify the kernel and create a variation of the source code, known as distributions that are used in computers and other devices.

Linux Architecture

Linux is primarily divided into user space and kernel space. These two components interact through the system call which acts as a gateway interface - which are predefined and Used to interact with Linux kernel from userspace to application.



Kernel Space:

This is the privileged space where the linux kernel operates. It directly interacts with hardware and provides OS COMPONENT services(Process Management, file Management, Memory Management, I/O management, etc…).

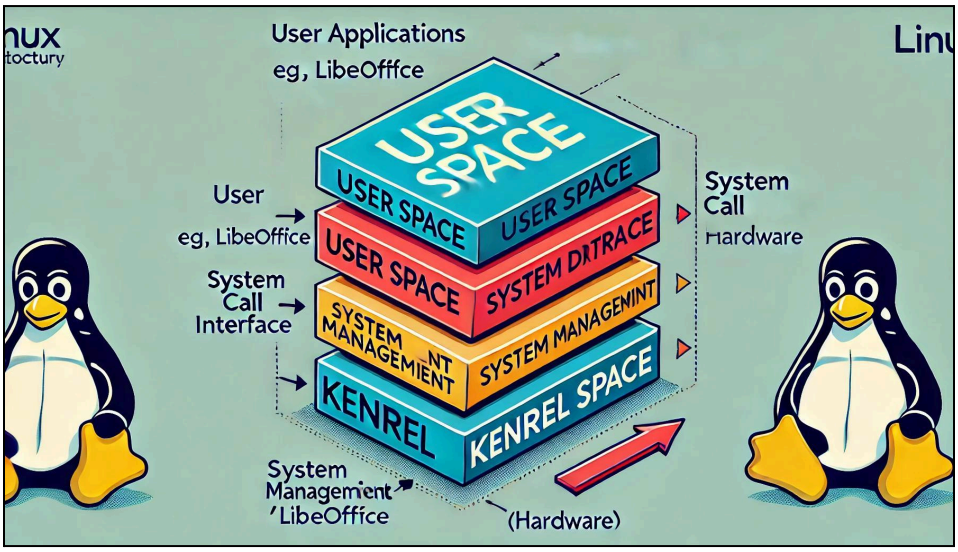
So kernel space is where the kernel (the core of the operating system) execute/runs and provide services.

User Space:

This is where the user applications and services run.It runs in an unprivileged mode and it can not directly access hardware.

The communication happens from user space to kernel space using APIs.

Example based Scenario:



When a user wants to print a document, the process starts in the user space, which includes the desktop environment and the applications the user interacts with. For example, the user opens a document in LibreOffice, a printing application, and sends a print command. However, LibreOffice itself doesn't know how to communicate with the printer directly. It relies on the kernel for help. At this point, the kernel space comes into play. The kernel, which is the core of the operating system, bridges the gap between user applications and hardware. The kernel uses a specialized software called the device driver to manage communication with the hardware. This driver translates the user's print request into precise instructions the printer hardware can understand.

To initiate this process, LibreOffice makes a system call (e.g., write) to send the formatted data to the printer. The system call acts as an interface between user space and kernel space, transitioning the process securely into kernel space. This ensures the communication follows predefined rules, preventing errors or harm to the system. Inside the kernel space, the kernel invokes the printer driver, which is a device driver specific to the printer hardware. The printer driver translates the generic system call into hardware-specific commands the printer understands, such as moving the print head, feeding paper, and applying ink.

Finally, the printer hardware processes these instructions and begins printing the document. During this time, the kernel ensures that other processes in user space do not interfere with the ongoing print job. It also manages system resources like memory and CPU to ensure smooth operation.

Once the job is complete, the printer driver sends a status (e.g., "Print complete") back to the kernel. The kernel then relays this information to the user space application, LibreOffice, which notifies the user with a message like "Print successful!"

Linux Kernel modules:

The Linux kernel is the core of the linux operating system. It interacts with hardware and provides services to user-space applications and a Linux Kernel Module (LKM) is a piece of code that can be dynamically loaded into the kernel to extend its functionality without rebooting the system.

Example: Adding device drivers for new hardware and implementing a new filesystem or extending the system calls.

What is Loading and Unloading ?

Loding: When we insert an LKM into the kernel during runtime and the kernel integreates it makes it functional.

Unloading: when we remove the LKM from the kernel it frees the resources and the kernel returns to its original state.

It is important because without booting we can add and remove the functionality and for the save the system resource we only can load the required parts. When we are testing a new feature or driver we can quickly load/unload it to debug without rebuilding the entire kernel.

Possess of loading and unloading:

Loding:

We use the **insmod** command to load an LKM and this module runs the **init function** performing necessary initializations.

Unloading:

We use **rmmod** command to unload the LKM and this module runs the **exit function** to clean up the resources.

we are loading these codes at runtime and they also not part of the official Linux kernel these are called loadable kernel modules

Example:

```
sudo insmod mymodule.ko  # Load the module
dmesg | tail             # Check logs for initialization
sudo rmmod mymodule      # Unload the module
dmesg | tail             # Check logs for cleanup
```

Methods to add LKM to the kernel:
There are two methods to add LKM to the kernel

Method 1: The basic way is to add the code to the kernel source tree and recompile the kernel. We found that in the driver directory.

Method 2: So it is the efficient way to add the code to the kernel while it is running by using a loading and unloading module where the modules are represented or refers to the code we want to add to the kernel.

Since we are loading these codes at runtime and they are also not part of the official Linux kernel, these are called loadable kernel modules. Which is different from the **Base kernel** is located in the /boot directory and the base kernel is loaded when we boot our machine. Where the LKMs are loaded after the base kernel is loaded.

These LKMs are very much part of our kernel and they communicate with the base kernel to complete their functions.

LKM serve variety of purposes:
1. Device Driver Support:
LKMs allow for the dynamic loading of device drivers, enabling the kernel to recognize and interact with new hardware devices without recompiling the entire kernel.
This flexibility ensures that the kernel can adapt to a wide range of hardware configurations and support new devices as they become available.

2.System Call Extensions:
Adding New System Calls: LKMs can extend the system call interface, providing new ways for user-space applications to interact with the kernel.
Custom System Calls: LKMs can be used to create custom system calls to perform specific tasks.

3. Security Enhancements:
Security Modules: LKMs can be used to implement security modules that provide additional security features, such as intrusion detection or encryption.

4.Debugging and Testing:
Debugging Tools: LKMs can be used to implement debugging tools that help identify and fix kernel-level issues.
Testing New Features: LKMs can be used to test new features without affecting the stability of the running kernel.

Applications Of Lodable kernel Module(LKMs):

1. LINUX DEVICE DRIVER:

The device driver is designed for a specific piece of hardware. The kernel uses it to communicate with that piece of hardware without having to know the details of How the hardware works.
Example:
The hardware components like the keyboard mouse and monitor are like computer body parts. The kernel is the core of the operating system like a brain. A device drive is like a translator; a specified software component enables the kernel(brain) to communicate with hardware devices.

Key points:
Device Drivers are tailored to specific hardware devices a driver for usb keyboard wont work for a usb mouse.
Many Devices are loaded kernel modules, meaning they can be added or removed without rebooting the system providing flexibility.

2.FILE SYSTEM AND FILE SYSTEM DRIVERS:

In the hard drive Each file is stored on a specific partition. But **how does the computer know which type of hard drive it is? Is it a traditional hard drive with files on Partition, or a digital hard drive with files stored electronically?**

These is where the file system recognition comes into play:

FILE SYSTEM:
A file system is a way of organizing and storing data on a storage device. It defines how data is structured, accessed, and stored. Different file systems have different formats.

DIFFERENT FILE SYSTEM FORMAT:

For Windows:

- NTFS (New Technology File System): This is the primary file system used in modern Windows operating systems. It offers features like file compression, encryption, and access control lists.
- FAT32 (File Allocation Table 32): An older file system, still used on older Windows systems and many USB drives. It is simple and compatible with a wide range of devices, but has limitations on file size and partition size.
- exFAT (Extended File Allocation Table): A more modern file system that overcomes the limitations of FAT32. It is compatible with Windows, macOS, and Linux.

For Linux:

- EXT4 (Fourth Extended Filesystem): A widely used file system in Linux systems. It offers good performance, reliability, and features like journaling and online file system checking.
- XFS (X Filesystem): A high-performance journaling file system, often used on servers and high-performance workstations.
- Btrfs (B-tree File System): A modern file system that supports features like snapshots, data deduplication, and self-healing.

Other File Systems:

- HFS+ (Hierarchical File System Plus): Used in older macOS systems.
- APFS (Apple File System): The modern file system used in macOS.
- ReFS (Resilient File System): A Microsoft file system designed for large-scale storage systems.

File System Recognition:

When you connect a new storage device to your computer, the operating system needs to identify the type of file system it uses. This is done through a process called file system recognition.

Here's how it works:

- Superblock: Every file system has a special block of data called a superblock. This superblock contains information about the file system, such as its type, size, and block size.
- Reading the Superblock: When a storage device is connected, the operating system reads the first few sectors of the device to locate the superblock.
- Identifying the File System Type: By examining the contents of the superblock, the operating system can identify the file system type (e.g., ext4, NTFS, FAT32).
- Loading the Appropriate Driver: Once the file system type is identified, the operating system loads the corresponding file system driver.

Why is this important?

- Correct File Access: The correct file system driver is essential for accessing and manipulating files and directories on the storage device.
- Data Integrity: The driver ensures that data is read and written correctly, preventing data corruption.
- Performance Optimization: Different file systems have different performance characteristics. The driver can optimize file operations based on the file system type.

FILE SYSTEM DRIVERS :

A filesystem driver is a component of an operating system that enables it to interact with various storage devices and file systems. It acts as an intermediary between the kernel and the underlying storage hardware.

Key Roles of a Filesystem Drivers:

1.Storage Device Detection and initialization:

The filesystem drivers detect and identify the connected storage devices and initializes that storage hardware to the computer and prepares it for data transfer.

2.File System Recognition:

Recognizes the file system type (e.g., ext4, NTFS, FAT32) based on the file system on-disk format.

3.File System Mounting:

Mounts the file system, making it accessible to the operating system.
Allocates necessary resources (e.g., memory, buffers) for file system operations.

4.File System Operations:

Handles file and directory operations, such as:

- Creating, deleting, and renaming files and directories.
- Reading and writing data to files.
- Seeking specific positions within files.
- Handling file permissions and ownership.

5.Data Transfer:

Manages the transfer of data between the storage device and the system's memory.
Optimizes data transfer performance using techniques like buffering and caching.

6.Error Handling:

Detects and handles errors, such as disk failures, read/write errors, and file system corruption.
Implements error recovery mechanisms to minimize data loss.

3. SYSTEM CALLS

System calls are the interface between the user space and applications and the kernel.This allows a user program to request services from the kernel such as reading /write files (read()/write()) allocating memory, or creating a process fork().

A user program invokes a system call using a wrapper function (eg. printf internally uses write ()) the request is sent to the kernel through a special instruction (like syscall or int 0x80 on x86.)
The kernel identifies the system call and executes the corresponding kernel function and it returns the result (success or failure) back to the user program.

ADVANTAGE OF LOADABLE KERNEL MODULES

- We don't need to keep rebuilding the kernel every time we add a new device or if we upgrade an old device. This saves time and also helps in keeping our base kernel error free.

- LKMs are flexible and they can load and unload with a single line command; this helps in saving memory as we load LKM only when we need it.

DIFFERENCE BETWEEN KERNEL MODULES AND USER PROGRAMS:

KERNEL MODULES HAVE SEPARATE ADDRESS SPACE:

The kernel modules run in kernel space sharing kernels memory address space. And the user programs are run in user space with isolated memory from the kernel.

Why ?
Security: Separating the address spaces prevents user programs from accessing or modifying sensitive kernel data, protecting the system from bugs or malicious actions.
Stability: a faulty user prom can only crash itself, not the entire system. Kernel space isolation ensures the core of the OS remains intact.
Efficiency: The kernel operates with full privileges and direct hardware access while user programs interact with the kernel only via controlled system calls, maintaining efficient resource usage.

KERNEL MODULE HAVE EXECUTION PRIVILEGES:

The kernel module operates with full access to hardware and kernel resources and it interacts directly with hardware, bypassing protections.

Ex: reading and writing to an i/o port.

And other side user programs operate with the restricted privileges via system calls. Must use system calls to access hardware resources ensuring safety.

Ex: reading file.

KERNEL MODULES DO NOT EXECUTE SEQUENTIALLY:

The kernel modules are event driven because they execute as needed (e.g. interrupt handling).

And user programs are executed sequentially from the start to finish in one flow.

HEADER FILES:

The Kernel modules was use kernel headers(eg..<linux/module.h>)

And the User space headers (eg. <stdio.h>)

—----------

DIFFERENCE BETWEEN KERNEL MODULES AND KERNEL DRIVERS:

Kernel Modules:

A kernel module is a piece of code that can be dynamically loaded or unloaded into the linux kernel at runtime and its main purpose is that without needing to reboot or recompile the kernel to reboot or recompile the kernel.

Kernel modules are the tools you can add to the kernel as needed.

Example:

- A filesystem module to support a new filesystem(eg.ext4).
- A module implementing a custom system call.

Kernel Drivers:

A kernel driver is a type of kernel module specifically designed to interact with and control hardware devices.

Kernel drives are a specific kind of tool that is only useful for a particular job - interacting with hardware specific.

Purpose: Enable communication between the kernel and hardware components like network cards, USB devices, GPUs, etc.

Examples:

- A network interface card (NIC) driver for Ethernet hardware.
- A sound card driver to enable audio output.

Key Characteristics:

- "Drives" a particular hardware device.
- Provides the hardware-specific implementation for abstract kernel interfaces.

What is a Device Driver?

A device driver is like a translator that helps the operating system (OS) and user applications communicate with hardware devices.

In a linux system everything is a file and this means linux treats everything as a file even hardware.

Why Do We Need Device Drivers?

- Hardware (like a printer, keyboard, or hard drive) cannot directly communicate with the OS or applications.
- The device driver bridges the gap, allowing software to "talk" to the hardware in a language it understands.

Key Characteristics:

- OS-specific: A driver written for Linux won't work in Windows.
- Hardware-dependent: A driver for an HP printer won't work for a Canon printer.

DIFFERENT TYPES OF DEVICE DRIVER:

There are the three types of the Device drivers these are categorised based on the **how hardware interacts** with the and the **kind of data they deal with**. Each type is specialized to handle specific kinds of devices efficiently.

1.Character Devices:

A character device interacts with hardware by transferring data character by character. It's best suited for devices that need data in small chunks.

What They Are: Devices that handle data one character at a time.

Examples:

- Keyboard: Sends data (like keys pressed) one by one.
- Mouse: Sends movement and click data step-by-step.

How They Work:

- Think of it as writing a letter with a typewriter—one character at a time.
- You interact with these devices sequentially.

Character devices are represented as files in **/dev,** such as **/dev/ttyS0 (serial port)**.

Key Features:

- Data is transferred in real time.
- Can't be used to store large files.

2. Block Device

A block device transfers data block by block (fixed-size chunks), making it ideal for storage devices like hard drives or USB drives.

Examples:

- Hard Disks (HDD/SSD): Reads and writes large amounts of data.
- CD-ROMs: Transfers data in blocks during file access.
- USB Drives: Operates on block sizes for efficient data storage.

Linux Representation:

Block devices are also represented as files in **/dev,** such as **/dev/sda (the first hard disk).**

3. Network Device

A network device handles data transmission as packets over a network. It's used for communication between computers or devices.

Devices that handle packets of data for communication between computers.

Examples:

- Ethernet Card: Sends and receives packets over a wired connection.
- Wi-Fi Adapter: Handles wireless data transmission.
- Loopback Device: A virtual device used to send data to your own system (for testing).

Linux Representation:

Network devices are not visible in /dev. Instead, they can be listed using commands like:
ip link

How They Work:

Think of sending letters in envelopes. Each packet is like an envelope with a destination address and contents.

Why Do We Classify Like This?

It helps developers write drivers tailored for specific data handling needs.

Each type is optimized for certain hardware and tasks, like storage, communication, or interaction.

"In Linux, Everything is a File"

Linux treats all hardware as files. This simplifies interaction with devices, as reading or writing to a device is just like reading or writing a file.

- Character devices: /dev/tty (terminals).
- Block devices: /dev/sda (disks).
- Network devices: Accessed through special interfaces, not /dev.

treating hardware as files, Linux creates a consistent and user-friendly way to manage and interact with various types of hardware.

How to Identify Devices in `/dev`

Use ls -l to list the files and their types:

```
bash
ls -l /dev
```

Output:

```
bash
brw-rw---- 1 root disk 8, 0 Dec 12 10:00 /dev/sda  # Block device
crw-rw---- 1 root tty 4, 0 Dec 12 10:00 /dev/tty0  # Character device
```

The first character in the permission string indicates the type:

- b: Block device
- c: Character device

How /dev is Managed

- The device files in /dev are not created manually.
- They are managed by /dev, a device manager in Linux that dynamically creates/removes device files as hardware is connected/disconnected.

Why /dev is Important

- Provides a consistent way to access and manage hardware devices.
- Abstracts hardware details from the user and applications.
- Makes Linux flexible and efficient for handling various hardware.

IMPORTANT QUESION:

Q: Are Device Drivers Operating System-specific?

Yes, device drivers are operating system-specific. Each operating system requires its own set of device drivers to interact with hardware effectively. Therefore, device drivers designed for Linux may not work on other operating systems like Windows or macOS.

MODULE INFORMATION :

This information helps the kernel and users understand details about the module,such as its purpose, authorship, and versioning.

The module information is by following factors and These pieces of information are present in the Linux/module.h as macros.

- License
- Author
- Module Description
- Module Version

License:

A license in a Linux kernel module tells the kernel and users how the module can be used, shared, or modified. It's like a legal label that describes the "rules" of the module's code.

A software license defines the terms under which a developer code can be used, shared or modified.

The license specifies:

- Whether others can use the code freely.
- If modification must be shared with the community.
- Whether the module integrates well with the linux kernel.

The license we choose determines how the module interacts with linux and how it can be distributed.

Type of licenses:

1.GPL (General Public License)

- Core Rule: If you modify and redistribute the code, you must share your changes under the same license (GPL).
- Think of it like a group project in school.
- If someone uses your work, they must share their changes with everyone.
- You say: “You can use my work for free, but if you improve it, you must share the improved version too!”
- Purpose: Ensures freedom for users to modify and share the code while maintaining transparency.

Use Case:

Most Linux kernel modules use GPL because the Linux kernel itself is under GPL.
Example: A network card driver under GPL. If a company modifies it, they must share their improvements with the community.

Declaration in Code:

```
c
MODULE_LICENSE("GPL");
```

Scenario 1: Writing a GPL Device Driver

You create a driver for a custom sensor.
You want it to be compatible with the Linux kernel.

You use the GPL license:

```
c
#include <linux/module.h>
MODULE_LICENSE("GPL");
MODULE_AUTHOR("Your Name");
MODULE_DESCRIPTION("A GPL-based custom sensor driver");
```

2. BSD License

- Core Rule: Allows users to use, modify, and redistribute the code without requiring them to share modifications.
- Think of it like lending a recipe to a friend.
- They can use it, modify it, and even sell cookies made with it—but they don't have to tell you about their changes.
- You say: "Use my work however you want, but you don't have to share changes."
- Purpose: Provides more flexibility for companies or developers who don't want to disclose their changes.

Use Case:

Suitable for companies that want to use open-source code in proprietary products without revealing their proprietary changes.

Example: A storage driver under BSD used in embedded systems without sharing modifications.

Declaration in Code:

```
c
MODULE_LICENSE("Dual BSD/GPL");
```

3. MIT License

- Core Rule: Completely permissive; users can do anything with the code, including using it in proprietary software, without crediting the original author.
- Think of it as giving your recipe to everyone without any conditions.
- They can use it, change it, or sell products made from it. They don't even have to credit you!
- You say: "Do whatever you want with my work."
- Purpose: Encourages widespread use and adoption without any restrictions.

Use Case:

Commonly used for libraries or utilities that are meant to be reused in various projects.

Example: A library for handling file I/O under MIT license can be freely integrated into a Linux driver.

4. Proprietary License

- Core Rule: Users cannot see, modify, or redistribute the code. Access is controlled by the creator.
- Think of it as a secret recipe.
- Only you know it, and others can use it only if they pay or follow strict rules.
- You say: "You can use my work, but you can't see or change the details."
- Purpose: Protects intellectual property by preventing reverse engineering or unauthorized use.

Use Case:

Closed-source modules like NVIDIA's graphics driver. Users can use it but can't modify or understand the internal implementation.

Declaration in Code:

```
c
MODULE_LICENSE("Proprietary");
```

Scenario 2: Proprietary Driver

A company writes a graphics driver but doesn't want to share its source code.
The driver is declared proprietary:

```
c
#include <linux/module.h>
MODULE_LICENSE("Proprietary");
MODULE_AUTHOR("Company XYZ");
MODULE_DESCRIPTION("Proprietary graphics driver");
```

Consequence: It's not supported by the Linux community for debugging or modifications.

Why Choose a Specific License for Linux Modules?

GPL is the preferred license for Linux kernel modules because:

- It ensures compatibility with the Linux kernel, which is itself GPL-licensed.
- Modinfo (module information) can verify the license type and check compliance.

- Encourages community collaboration and bug fixes.

Non-GPL Licenses (e.g., Proprietary):

- The community and the kernel maintainers may reject bug reports related to proprietary modules because they lack transparency.
- Some companies use non-GPL licenses to protect their business models.

How Licenses Affect Device Drivers

1. Kernel Compatibility: GPL modules integrate seamlessly with the kernel.
2. Community Acceptance: GPL and similar open licenses are preferred by the Linux community.
3. Distribution: A proprietary module cannot be included in Linux distributions that prioritize open-source software.

Summary

- The license you choose determines how your code integrates with the kernel and how it can be used by others.
- GPL is preferred for open collaboration and kernel compatibility.
- Proprietary licenses restrict sharing but protect intellectual property.

2.Author

The MODULE_AUTHOR macro is a way to associate the author's information with the Linux kernel module.

MODULE_INFO Macro:

It stores the key-value pair (author, "Author Name") in the .modinfo section of the compiled module.

The kernel's module loader and modinfo tool use this metadata to fetch and display the author's information.

Purpose of MODULE_AUTHOR

1.Identification:
This macro allows you to specify the author of the module, helping others understand who developed it.

2.Documentation:
The modinfo command displays this information, aiding users or developers in identifying the contributor(s) of a kernel module.

3.Transparency and Collaboration:
Including author details makes it easier to reach out for questions, updates, or bug reports.

How to Use MODULE_AUTHOR
Add the Linux/module.h header file to your module:

```
c
#include <linux/module.h>
```

Use the macro in your module's source code:

```
c
MODULE_AUTHOR("Your Name <your_email@example.com>");
```

For multiple authors, use multiple MODULE_AUTHOR() lines:

```
c
MODULE_AUTHOR("Author1 <author1@example.com>");
MODULE_AUTHOR("Author2 <author2@example.com>");
```

Let’s dive into Module Description and Module Version in a detailed but beginner-friendly manner. These macros are part of the module.h header file and provide metadata about the module. This metadata is primarily used for identifying and managing kernel modules.

3.Module Description

The MODULE_DESCRIPTION macro provides a brief explanation of what the module does.

Purpose:

When you run the modinfo command on a module, it displays the description. This helps users and developers understand the purpose of the module without diving into its code.

How to Use:

```
c
MODULE_DESCRIPTION("A sample driver for learning purposes.");
```

Example : writing a driver for keyboard

```
c

MODULE_DESCRIPTION("Keyboard driver to manage input devices.");
```

Use Case:

Helps system administrators or developers quickly understand what a module does without inspecting the source code.

4.Module Version

The MODULE_VERSION macro specifies the version of your module. This can be used to track updates, fixes, or changes.

Purpose:

Versions are crucial when maintaining or debugging modules. It helps identify if the correct version of a module is loaded in the kernel.

How to Use:

```
c
MODULE_VERSION("1.0");
```

Example:

For a mouse driver:

```
c
Copy code
MODULE_VERSION("2.0.1");
```

Version Format Explained:

1.`<epoch> (optional)`: Used to reset the versioning. Defaults to 0 if not specified.
2.`<version> (mandatory)`: Main version number, e.g., 1.0.
3.`<extra-version> (optional):` Custom tags for distribution or customization,

e.g., debian1 or custom1.

Example with all parts:

```
c
MODULE_VERSION("1:3.4.5-custom1");
```

- 1: → Epoch
- 3.4.5 → Version
- custom1 → Extra version

How This Metadata Is Useful

modinfo Command: When you run the modinfo command, this metadata is displayed.

```
bash
modinfo my_module.ko
```

Example Output:

```
makefile

description:    A sample driver for learning purposes.
author:         Your Name <your.email@example.com>
version:        1.0
license:        GPL
```

——----------xox——----------

Simple kernel Module Programming:

Introduction

In Linux kernel module programming, instead of the standard main function used in user-space programs, Init and Exit functions act as the entry and exit points of a module. These functions manage the loading and unloading of the module into/from the kernel. Let's break it down step by step.

1. Init Function

What It Does:

- This function is called when the module is inserted into the kernel (e.g., using insmod).
- It acts as the "constructor" of the kernel module, where you can initialize resources, register device drivers, or set up data structures.

Syntax:

```
static int __init hello_world_init(void)
{
}
module_init(hello_world_init);
```

key Points:

- __init: A compiler attribute that marks this function as initialization code. After initialization, this code is freed to save memory.
- module_init: A macro that registers the init function with the kernel.

2. Exit Function

What It Does:

This function is called when the module is removed from the kernel (e.g., using rmmod).
It acts as the "destructor" of the kernel module, cleaning up any resources allocated in the init function.

Syntax:

```
void __exit hello_world_exit(void)
{
}
module_exit(hello_world_exit);
```

Key Points:

- __exit: A compiler attribute that marks this function as cleanup code.
- module_exit: A macro that registers the exit function with the kernel.

What is a Log?
A log is simply a record of events, messages, or data generated by a program, operating system, or device to give information about its operation. Logs help developers, administrators, or users to:

- Monitor what's happening.
- Diagnose problems.
- Debug code.

In the Linux kernel, the logs record what is happening inside the kernel. These logs are created using the printk() function.

What is a Log Level?
A log level is a priority or severity assigned to a log message. It indicates how important the message is. By using log levels, you can categorize logs, making it easier to focus on critical messages while ignoring less important ones.

In the Linux kernel, log levels are represented by macros such as KERN_INFO, KERN_ERR, etc.

Why Use Log Levels?
To classify messages based on their importance.
To easily filter messages (e.g., focus on errors while ignoring debug messages).
To help developers identify and prioritize issues quickly.

3.Printk Function
The printk() function is used for logging messages in the kernel. It behaves like printf() in user-space programs but works in kernel space.

How it Works:
Messages logged via printk() are stored in the kernel log buffer.
Use the dmesg command to view these messages.
printk() Function in the Linux Kernel
In user-space programs, you often use the printf() function to print messages to the terminal. However, in the Linux kernel, you use the printk() function to log messages, which is similar to printf() but is specifically designed for the kernel.

Syntax of printk()

```
c
printk(log_level, "Your message here");
```

Where:

log_level: This specifies the priority of the message (e.g., KERN_INFO, KERN_ERR).
Message: The actual text message that you want to log.

Log Levels in printk()
Linux kernel supports different log levels that help categorize messages based on their severity. These log levels allow you to prioritize messages and focus on the most critical ones when needed.

Log Level	Macro	Explanation	Example	Alias Function
Emergency	KERN_EMERG	System in critical state, imminent crash	printk(KERN_EMERG "System crash imminent!");	pr_emerg()
Alert	KERN_ALERT	Requires immediate attention	printk(KERN_ALERT "Critical disk failure!");	pr_alert()
Critical	KERN_CRIT	Serious hardware/software conditions	printk(KERN_CRIT "Memory corruption detected!");	pr_crit()
Error	KERN_ERR	Error conditions needing fixing	printk(KERN_ERR "Device not found!");	pr_err()
Warning	KERN_WARNING	Potential problem, not immediate threat	printk(KERN_WARNING "Disk space low!");	pr_warn()
Notice	KERN_NOTICE	Normal but noteworthy events	printk(KERN_NOTICE "New USB device detected.");	pr_notice()
Information	KERN_INFO	General information, regular operations	printk(KERN_INFO "Module loaded successfully.");	pr_info()
Debug	KERN_DEBUG	Debugging messages, development use	printk(KERN_DEBUG "Entering function XYZ.");	pr_debug()
Default	KERN_DEFAULT	Default log level	printk(KERN_DEFAULT "Some default message");	N/A
Continue	KERN_CONT	Continues previous log message	printk(KERN_CONT "Continuing from previous mes	pr_cont()

The difference between printf and printk

- Printk() is a kernel-level function, which has the ability to print out to different log levels as defined in. We can see the prints using the dmesg command.
- printf() will always print to a file descriptor – STD_OUT. We can see the prints in the STD_OUT console.

Passing Arguments to Linux Device Drivers:

In Linux Kernel Modules (LKMs), just like passing arguments to a user-space program (via argc and argv in main), we can also pass parameters to a kernel module. These parameters are typically passed during module loading using insmod or modprobe.

To accomplish this, Linux provides module parameter macros that allow you to define parameters your kernel module can accept. These parameters are then accessible from within the module and can be used to alter its behavior.

Permissions:

Before discussing the module parameter macros we can see the permissions.

What Are Permissions?

Permissions control who (user, group, or others) can read, write, or execute a parameter (or file) in the Linux system. In the context of module parameters, permissions decide who can access and modify the parameter values.

Permissions are applied when a module parameter is exposed in the `/sys/module/<module_name>/parameters/` directory. This is where Linux creates an interface for accessing module parameters.

Understanding Permission Macros

The permission macros are defined in Linux headers **(<linux/stat.h>**), and they follow a standard convention:

```
Structure of a Macro

S_<Action><Scope>
```

Action:

- R: Read — Permission to read the value.
- W: Write — Permission to modify the value.
- X: Execute — Permission to execute (not typically used for module parameters).

Scope:

- USR: User — The owner of the module (usually root or the one who loaded it).
- GRP: Group — Users belonging to the same group as the module's owner.
- OTH: Others — All other users on the system.

Permission	Description
MacroPermissionValueS_IRUSROwner	Owner can read the file/parameter
S_IWUSR	Owner can write to the file/parameter
S_IXUSR	Owner can execute the file/parameter
S_IRGRP	Group members can read the file/parameter
S_IWGRP	Group members can write to the file/parameter
S_IXGRP	Group members can execute the file/parameter
S_IROTH	Others can read the file/parameter
S_IWOTH	Others can write to the file/parameter
S_IXOTH	Others can execute the file/parameter

How Permissions Are Combined

You can combine multiple permissions using the bitwise OR (|) operator.

Examples:

1.Read + Write for User Only

```
c
S_IRUSR | S_IWUSR
```

- The module parameter can be read and written by the user (owner).

2.Read for Everyone

```
c
S_IRUSR | S_IRGRP | S_IROTH
```

- The module parameter is readable by the owner, group, and others.

3.Read for User, Write for Group

```
c
S_IRUSR | S_IWGRP
```

- The owner can **read**, and group members can **write**.

Why Are Permissions Important?

- Security: Prevent unauthorized users from changing critical parameters.
- Control: Ensure only the right people (e.g., root) can modify sensitive parameters.
- Debugging: Allow broader read access for debugging while restricting write access.

Module Parameters Macros for Passing Arguments to Linux Device Driver:

1.module_param()

The module_param() macro is used in Linux kernel modules to allow the user to pass arguments (parameters) to the module at runtime. These parameters can be configured when the module is loaded using the insmod command.

Syntax of module_param()

```
c
module_param(name, type, perm);
```

- name: The variable to store the value of the parameter.
- type: The type of the variable (e.g., int, bool, charp, etc.).
- perm: Permissions for accessing this parameter from user space via /sys/module/<module_name>/parameters/<parameter_name>.

How it works

1. When a kernel module uses module_param(), a parameter is created in the `/sys/module/<module_name>/parameters/ directory.`
    - Example: If the parameter name is my_value, and the module is named hello_module, the parameter file will be located at `/sys/module/hello_module/parameters/my_value.`
2. Users can view or modify these parameters dynamically by accessing this sysfs entry.

Example:

```
c
module_param(my_value, int, S_IWUSR | S_IRUSR);
```

- This creates a parameter named my_value.
- It can store an integer value.
- The parameter can be read (S_IRUSR) and written (S_IWUSR) by the owner (user).

Supported Data Types

1. bool

- Stores a boolean (true/false) value.
- Variable type: int (0 = false, 1 = true).

Example:

```
c
module_param(my_flag, bool, S_IRUSR);
```

- If the user sets my_flag=1, it means "true".
- If my_flag=0, it means "false".

2. in bool

- Inverted boolean. If the user provides 1 (true), the parameter is treated as false internally, and vice versa.

Example:

```
C
module_param(my_inv_flag, in bool, S_IRUSR);
```

- If the user sets my_inv_flag=1, the module treats it as "false".

3. charp

- Stores a string value.
- Variable type: char*.

Example:

```
c
module_param(my_string, charp, S_IRUSR);
```

- If the user passes my_string="Hello", this value is stored in the module.

4. int

- Stores a signed integer.

Example:

```
c
module_param(my_number, int, S_IRUSR | S_IWUSR);
```

- The user can set my_number=42.

5. uint

- Stores an unsigned integer.

Example:

```
c
module_param(my_unsigned, uint, S_IRUSR);
```

- The user can set my_unsigned=100.

6. long

- Stores a signed long integer.

Example:

```
C
module_param(my_long, long, S_IRUSR);
```

7. ulong

- Stores an unsigned long integer.

Example:

```
C
module_param(my_ulong, ulong, S_IRUSR);
```

8. short

- Stores a signed short integer.

Example:

```
c
module_param(my_short, short, S_IRUSR);
```

9. ushort

- Stores an unsigned short integer.

Example:

```
c
module_param(my_ushort, ushort, S_IRUSR);
```

IMPORTANT : PRACTICAL EXAMPLE OF THESE IN FOLDER.

2.module_param_arry()

What is module_param_array()?

The module_param_array() macro allows you to pass an array of values to a Linux kernel module as a parameter during module loading. These values are provided as a comma-separated list from the command line.

Syntax of module_param_array()

```
c
module_param_array(name, type, num, perm);
```

Parameters:

- name: The name of the array (and the parameter name passed to the module).
- type: The type of the array elements (int, charp, etc.).
- num: A pointer to an integer variable where the count of array elements will be stored. Pass NULL if you don't need this.
- perm: The file permissions for the parameter (e.g., 0644 or 0444).

Example: Passing an Array to a Kernel Module

Here's a basic example:

How to Use It:

1.Compile the Module:

Use your Makefile to build the .ko file.

2.Insert the Module with Parameters:

Pass an array during module insertion using the insmod command:

```
bash
sudo insmod my_module.ko my_array=10,20,30
```

3.Check Kernel Logs:

Use dmesg to verify the array elements passed:

```
bash
dmesg | tail
```

4.Remove the Module:

Use:

```
bash
sudo rmmod my_module
```

```c
c
#include <linux/module.h>
#include <linux/init.h>

#define MAX_ARRAY_SIZE 5

static int my_array[MAX_ARRAY_SIZE];  // Array to hold values
static int array_size;               // To store the number of elements passed

module_param_array(my_array, int, &array_size, 0444);
MODULE_PARM_DESC(my_array, "An array of integers");

static int __init my_module_init(void)
{
    int i;

    pr_info("Module loaded with parameters:\n");
    for (i = 0; i < array_size; i++) {
        pr_info("my_array[%d] = %d\n", i, my_array[i]);
    }
    return 0;
}

static void __exit my_module_exit(void)
{
    pr_info("Module unloaded.\n");
}

module_init(my_module_init);
module_exit(my_module_exit);

MODULE_LICENSE("GPL");
MODULE_AUTHOR("Vicky Deokar");
MODULE_DESCRIPTION("Example module with array parameter.");
```

Expected Output:

If you pass my_array=10,20,30, the output in dmesg will look like this:

```
less

Module loaded with parameters:
my_array[0] = 10
my_array[1] = 20
my_array[2] = 30
Module unloaded.
```

3.module_param_cb():

The module_param_cb() macro allows you to register a callback function to handle changes to module parameters. This is useful when you need to perform actions or handle events dynamically when a module parameter is updated.

Syntax of module_param_cb()

```
c
module_param_cb(name, ops, arg, perm);
```

Parameters:

1. **name:** The name of the parameter (as seen in `/sys/module/<module_name>/parameters/<name>`).
2. **ops:** A pointer to a structure of type `struct kernel_param_ops.` This structure defines the get and set callbacks for the parameter.
3. **arg:** A pointer to the variable associated with the parameter.
4. **perm:** The permissions for the sysfs entry (e.g., 0644, 0444).

The `struct kernel_param_ops`
This structure defines the callback functions:

```
c
Copy code
struct kernel_param_ops
{
    int (*set)(const char *val, const struct kernel_param *kp);
    int (*get)(char *buffer, const struct kernel_param *kp);
};
```

- set(): Called when the parameter value is updated.
- get(): Called when the parameter value is read.

Example: Using module_param_cb()

```c
#include <linux/module.h>
#include <linux/kernel.h>
#include <linux/init.h>
#include <linux/moduleparam.h>

/* Variable to store the parameter value */
static int valueETX = 0;

/* Callback functions */
static int param_set_callback(const char *val, const struct kernel_param *kp)
{
    int res = kstrtoint(val, 10, &valueETX); // Convert string to integer
    if (res < 0)
        return res;

    pr_info("Parameter 'valueETX' updated to: %d\n", valueETX);
    return 0;
}

static int param_get_callback(char *buffer, const struct kernel_param *kp)
{
    return sprintf(buffer, "%d\n", valueETX);
}

/* Define the kernel_param_ops structure */
static const struct kernel_param_ops param_ops = {
    .set = param_set_callback,
    .get = param_get_callback,
};

/* Register the parameter with a callback */
module_param_cb(valueETX, &param_ops, &valueETX, 0644);
MODULE_PARM_DESC(valueETX, "An integer parameter with a callback");

static int __init my_module_init(void)
{
    pr_info("Module with callback parameter loaded.\n");
    pr_info("Initial valueETX = %d\n", valueETX);
    return 0;
}

static void __exit my_module_exit(void)
{
    pr_info("Module with callback parameter unloaded.\n");
}

module_init(my_module_init);
module_exit(my_module_exit);

MODULE_LICENSE("GPL");
MODULE_AUTHOR("Vicky Deokar");
MODULE_DESCRIPTION("Example module using module_param_cb()");
```

How It Works:

1.Set Callback (param_set_callback):

- Triggered when the value of valueETX is changed (e.g., using echo in sysfs).
- Converts the input string into an integer and updates valueETX.

2.Get Callback (param_get_callback):

- Triggered when the parameter is read (e.g., cat /sys/module/<module_name>/parameters/valueETX).
- Formats valueETX as a string for output.

Testing the Module:

1. Insert the Module:

```
bash
sudo insmod my_module.ko
```

2. Read the Parameter:

```
bash
cat /sys/module/my_module/parameters/valueETX
```

3. Change the Parameter:

```
bash
echo 42 > /sys/module/my_module/parameters/valueETX
```

4. Check Kernel Logs:

```
bash
dmesg | tail
```

You'll see a message like:

```
sql

Parameter 'valueETX' updated to: 42
```

5.Remove the Module:

```
bash
sudo rmmod my_module
```

What Is MODULE_PARM_DESC()?

Purpose:

- Provides a human-readable description of a module parameter.
- Helps users understand what each parameter does when using commands like modinfo.

Visibility:

- This description appears in the output of the modinfo command, making it easier for others to use your module correctly.

Format:

```
c
MODULE_PARM_DESC(parameter_name, "Description of the
parameter");
```

When Would You Need This Notification?

A notification is crucial when a change in a parameter value requires an immediate action or reaction in the system. Let's break this down with a practical scenario:

Scenario: Writing to a Hardware Register

What's happening?

- You're managing hardware through a driver, and a hardware register needs to be updated when a specific parameter (e.g., valueETX) is set to 1.
- Without a notification mechanism, the driver has no way to automatically detect that the parameter was changed and, therefore, cannot take any immediate action.

How Does module_param_cb() Help?

- Using the callback mechanism, the set function gets triggered whenever the parameter value changes.
- In this function, you can check the value and write to the hardware register (or perform any other necessary action).

Why Is This Important?

- **Dynamic Control**: You can dynamically control the behavior of your driver without recompiling the module.
- **Real-Time Responses**: Immediate responses to parameter changes allow the driver to interact efficiently with hardware or the kernel.

Introduction to Character Drivers

Character drivers are a specific type of device driver that manage devices operating with **byte-oriented input/output (I/O).** These are essential for interacting with devices where data flows sequentially, such as serial ports or audio devices.

Byte-oriented I/O refers to a mode of communication where data is transferred or processed one byte at a time. This approach is commonly used for devices that handle data in small, sequential units, typically in the form of 8-bit bytes.

Characteristics of Byte-Oriented I/O

1.Sequential Data Flow:

- Data is read or written in a continuous stream, one byte after another.
- Examples:
- Reading characters from a keyboard.
- Sending/receiving data over a serial port.

2.Immediate Processing:

- Each byte is processed as it arrives or is sent.
- Unlike block-oriented I/O (used in storage), there is no concept of buffering large chunks of data for batch processing.

3.Device Communication:

- Ideal for devices like:
- Serial ports (e.g., UART, RS-232).
- Character-based terminals.
- Sensors sending single readings.

Why Use Byte-Oriented I/O?

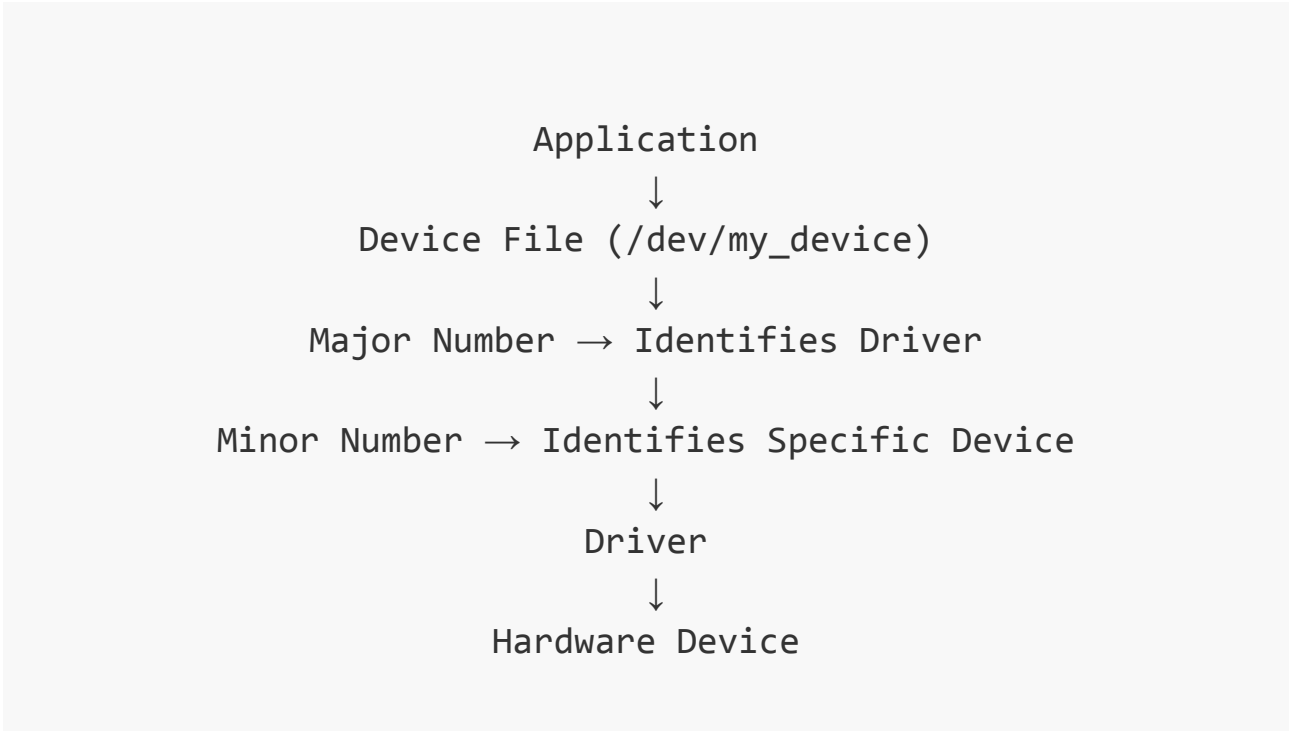
- Simplicity: Devices that don't require high throughput or buffering work well with byte-oriented I/O.
- Real-Time Communication: Enables immediate processing of data, which is critical for interactive devices like keyboards or sensors.
- Low Resource Requirement: Requires minimal memory and computational resources compared to block-oriented systems.

What Makes Character Drivers Special?

- They handle **sequential data** (byte-by-byte operations).
- Commonly used for a wide range of devices:
  - **Serial ports**
  - **Audio devices**
  - **Video and camera devices**
  - **Basic I/O devices**
- Any driver that doesn't involve block storage (like hard disks) or networking usually falls under the category of **character device drivers**.

How Applications Communicate with Hardware
The communication path between an application and a hardware device involves several layers. Here's an overview:

6. Visual Representation of the Process



Application Layer:

- The application initiates communication by opening a device file (e.g., /dev/ttyS0 for a serial port).
- This device file acts as an abstraction of the physical hardware device.

Device File Layer:

- Each hardware device has a corresponding device file in the /dev directory.

Device files are identified by:

- Major Number: Identifies the driver associated with the device.
- Minor Number: Identifies the specific instance of the device handled by the driver.
- These numbers allow the kernel to associate the device file with the correct driver.

Device Driver Layer:

- The character driver implements functionality for the hardware device.
- It processes system calls like read, write, and ioctl to interact with the device.
- It translates high-level operations (e.g., reading a file) into low-level hardware operations.

Hardware Layer:

- The driver communicates directly with the hardware using:
- I/O ports
- Memory-mapped registers
- Interrupts

Major and Minor Numbers

In Linux, device files (also called special files) are used to interact with hardware devices. These files are located in the /dev/ directory. Although applications interact with devices using the name of the device file, the Linux kernel uses major numbers and minor numbers to establish the connection between the device file and the corresponding device driver.

Major Number:

- The major number identifies the device type (e.g., IDE disk, SCSI disk, serial port, etc.).
- It acts as the driver identifier: Each device driver in the Linux system is assigned a unique major number.
- When the kernel receives a request for a device, it uses the major number to determine which driver is responsible for handling that device.

Minor Number

- The minor number identifies the specific device instance handled by the driver (e.g., first disk, second serial port, etc.).
- It acts as a device specifier: It distinguishes between multiple devices managed by the same driver.

Example:

Imagine a driver controlling 4 UART serial ports:

Major number = 240 (driver for serial ports).
Minor numbers = 0, 1, 2, 3 (for /dev/ttyS0, /dev/ttyS1, /dev/ttyS2, /dev/ttyS3).

Most of the time the major identifies the driver while the minor number identifies each physical device served by the driver.

Where Are Major and Minor Numbers Used?

Device files in /dev/ directory use Major and Minor numbers.

Major Number: Used by the kernel to identify the driver that handles the device.
Minor Number: Passed to the driver to identify a specific device.
To see the Major and Minor numbers:

```
bash
ls -l /dev/ttyS0
crw-rw---- 1 root dialout 4, 64 Dec 15 10:00 /dev/ttyS0
```

Here:

- c: Character device.
- 4: Major number.
- 64: Minor number.

This tells the kernel:

- Major 4 → Serial driver.
- Minor 64 → Specific port (/dev/ttyS0).

Where Are Major and Minor Numbers Located?

You can view the assigned Major and Minor numbers in two places:

1. At /proc/devices:

Lists all major numbers and their associated drivers.

```
bash
Copy code
cat /proc/devices
```

Example Output:

```
Character devices:
1 mem
4 tty
240 my_device

Block devices:
1 ramdisk
/dev/ Directory:
```

2.Special device files with Major and Minor numbers.

```
bash

ls -l /dev/my_device
```

Output:

<

```
bash

crw------- 1 root root 240, 0 Dec 15 10:00 /dev/my_device
```

Here:

- 240 → Major Number.
- 0 → Minor Number

—---

Allocating Major and Minor Number

There are two ways to allocate a major and minor number.

1. Statically allocating
2. Dynamically Allocating.

1.Static Allocation of Major/Minor Numbers

- Static allocation is used when we want to set a particular major number for a driver like manually set..
- If the major number is already taken to another device or that number is allocated then it fails to create a device file.

Function:

```
int register_chrdev_region(dev_t first, unsigned int count, char *name);
```

- dev_t first: Start of the device number range (both major and minor).
- unsigned int count: How many device numbers you need.
- **char *name:** The name of your device (visible in /proc/devices).

Steps for Static Allocation:

1.Use MKDEV to create a dev_t structure with a specific major and minor number.

```
c
dev_t dev = MKDEV(235, 0);  // 235 is the major number, 0 is minor
```

2.Register the device number using register_chrdev_region().

```
c
register_chrdev_region(dev, 1, "my_device");
```

3.If successful, your major and minor numbers are ready.

To Retrieve Major/Minor Numbers:

- MAJOR(dev) gives the major number.
- MINOR(dev) gives the minor number.

2.Dynamic Allocation of Major/Minor Numbers

IF we don't want the fixed major and minor number then we use dynamic allocation.
This allocates the major number dynamically to the driver.

Function:

```
c
int alloc_chrdev_region(dev_t *dev, unsigned int firstminor, unsigned int count, char *name);
```

- dev: Stores the first allocated device number.
- firstminor: The starting minor number (usually 0).
- count: How many numbers you need.
- name: Device name.

Steps for Dynamic Allocation:

1.Call alloc_chrdev_region().

```
c

dev_t dev;
alloc_chrdev_region(&dev, 0, 1, "my_device");
```

2.The kernel allocates a major number, and you can retrieve it using:

```
c
printk("Allocated Major = %d\n", MAJOR(dev));
```

3.Freeing the Device Numbers

```
C
void unregister_chrdev_region(dev_t first, unsigned int count);
```

Why Do We Free Device Numbers?
When you allocate major and minor numbers using:

- register_chrdev_region() (static allocation), or
- alloc_chrdev_region() (dynamic allocation),

The kernel assigns those numbers to your driver.
If you don't free these numbers when your driver is no longer in use:

1. Resource Leak: The numbers remain "reserved" in the kernel. Other drivers will not be able to use them.
2. Conflicts: If you reload your driver, the kernel may try to assign the same numbers again. This can lead to errors or unexpected behavior.
3. Kernel Cleanliness: Properly freeing resources ensures the system remains stable and clean.

The kernel maintains a list of allocated device numbers.
unregister_chrdev_region() tells the kernel: "I'm done with these numbers; you can reuse them."

What Happens If You Don't Free Them?

- When you unload the driver (using rmmod), the numbers still appear allocated in the kernel's internal tables.
- If you try to load the driver again, the kernel might fail or warn you that the device numbers are already in use.
- Over time, this can consume the available major/minor number space, causing system instability.

How It Works:

When you call:

```
c

void unregister_chrdev_region(dev_t first, unsigned int count);
```

- first: The starting device number (major + minor).
- count: How many contiguous numbers you want to release.

The kernel removes these numbers from its list of allocated device numbers, making them available for reuse.

Difference between static vs dynamic allocation:

Static	Dynamic
You manually set the major number.	**Kernel assigns the major number.**
Prone to conflicts with other drivers.	**No conflicts—always safe.**
Useful if you need a fixed number.	**Preferred method—avoids conflicts.**
Device nodes must match major/minor.	**Device nodes are created at runtime.**

Device Node in Linux

1. What is a Device Node?

A device node is a special file located in the /dev directory in Linux. It serves as an interface that allows user-space applications to interact with device drivers in the kernel.

- Example: /dev/cdac_edd is a device node.
- Purpose: It connects user-space programs to the kernel driver.

2. Why Do We Need a Device Node?

The Linux kernel uses Major Numbers and Minor Numbers to identify the device driver and the specific device it controls. A device node acts as a link to these numbers.

- Major Number: Identifies which device driver will handle the request.
- Minor Number: Specifies which device (if multiple) the driver should manage.

3. How Does It Work?

Creating the Device Node

To create the device node, use the mknod command with the following syntax:

```
bash
sudo mknod /dev/cdac_edd c 202 0
```

- c: Indicates that the device is a character device.
- 202: Major Number (the link to the driver).
- 0: Minor Number (identifies the specific device if there are multiple).

After creating the device node, the kernel associates it with the specified major and minor numbers.

Interacting with the Device

When a user-space program interacts with /dev/cdac_edd, the kernel will check the major number (202) and send the request to the driver that registered this major number.

For example:

```
bash
Copy code
echo "hello" > /dev/cdac_edd
```

- The kernel sees that /dev/cdac_edd has major number 202, so it directs the request to the driver that has registered major 202.
- This makes the driver process the request (e.g., reading or writing data).

4. Analogy

- Major Number = Phone Number for the driver (the link to the driver).
- Device Node = Phone that you pick up to make a call.
- User Program (like echo, cat) = Caller that wants to talk to the driver.

Without the device node, user-space programs have no way to talk to the driver.

5. Key Takeaways

- The device node is essential to allow user-space programs to communicate with kernel-space drivers.
- Major Number links the device node to the driver.
- Minor Number helps the driver identify specific devices.
- Device nodes are created using mknod and provide a way for programs to read, write, and interact with the kernel driver.

In Short:

- **Device node** = Link between user-space and your driver.
- It allows you to access your driver using standard file commands (like cat, echo, or open() in C).**PRACTICAL EXAMPLE:**

STATIC EXAMPLE:

```
/* Major:Minor static allotment */

#define pr_fmt(fmt)   KBUILD_MODNAME ": " fmt

#include <linux/module.h>
#include <linux/init.h>
#include <linux/fs.h>

#define MY_MAJOR_NUM  (202)
#define MY_DEV_NAME   "cdac_edd"
dev_t dev = MKDEV(MY_MAJOR_NUM, 0);

static int __init my_mod_init(void)
{
    int ans;

    pr_info("Hello world from mod31!\n");
    ans = register_chrdev_region(dev, 1, MY_DEV_NAME);
    if (ans < 0)
    {
        pr_info("Error in major:minor allotment!\n");
        return -1;
    }
    pr_info("major:minor %d:%d allotted!\n", MAJOR(dev), MINOR(dev));
    return 0;
}

static void __exit my_mod_exit(void)
{
    pr_info("Goodbye world from mod31!\n");
    unregister_chrdev_region(dev, 1);
    pr_info("major:minor numbers freed up...\n");
    return;
}

module_init(my_mod_init);
module_exit(my_mod_exit);

MODULE_LICENSE("GPL");
MODULE_AUTHOR("EDD <edd@cdac.gov.in>");
MODULE_DESCRIPTION("major:minor static allotment module!");
```

COMPILE AND TEST.

1. Compile the Module

cmd for cross compilation:

```
make ARCH=arm CROSS_COMPILE=arm-linux-gnueabihf-
```

2. Insert a module
Run:

```
bash

sudo insmod static_allocation.ko
```

3. Verify the Module is Inserted
Check kernel logs:

```
bash

dmesg | tail
```

You should see messages like:

```
vb net
mod31: Hello world from mod31!
mod31: major:minor 202:0 allotted!
```

Verify Module in /proc/devices
Check if your driver appears in the devices list:

```
bash

cat /proc/devices | grep cdac_edd
```

You should see:

```
202 cdac_edd
```

4. Create the Device Node
The kernel has allocated major number 202 and minor number 0 for this driver.

Create a device node using the mknod command:

```
bash

sudo mknod /dev/cdac_edd c 202 0
```

Verify the device node:

```
bash

ls -l /dev/cdac_edd
```

Output:

```
bash

crw-r--r-- 1 root root 202, 0 <date> /dev/cdac_edd
```

5. Testing the Driver
You can interact with the device node by reading or writing to it. For example:

```
bash

echo "test message" > /dev/cdac_edd
cat /dev/cdac_edd
```

Since this driver is simple and doesn't implement actual read/write callbacks, these commands will likely result in errors, but it shows interaction.

6. Testing the Driver
You can interact with the device node by reading or writing to it. For example:

```
bash
echo "test message" > /dev/cdac_edd
cat /dev/cdac_edd
```

Since this driver is simple and doesn't implement actual read/write callbacks, these commands will likely result in errors, but it shows interaction.

7. Unload the Module
7.1 Remove the Module
Run:

```
bash

sudo rmmod static_major_driver
```

7.2 Verify the Module is Removed
Check kernel logs:

```
bash
dmesg | tail
```

YOU SHOULD SEEN :

```
vb net

mod31: Goodbye world from mod31!
mod31: major:minor numbers freed up...
```

7.3 Remove the Device Node

```
To clean up the device node:

bash
sudo rm /dev/cdac_edd
```

Same process for dynamic also i will not provide a process.

DYNAMIC ALLOCATION EXAMPLE:

```c
/* Major:Minor dynamic allotment */

#define pr_fmt(fmt) KBUILD_MODNAME ": " fmt

#include <linux/module.h>
#include <linux/init.h>
#include <linux/fs.h>

#define MY_DEV_NAME "cdac_edd"  // Device name

dev_t dev = 0;  // Initialize with 0, to allow dynamic major number allocation

/*
** Module Init function
*/
static int __init hello_world_init(void)
{
    int ans;

    pr_info("Hello world from mod32!\n");

    // Dynamically allocate major and minor numbers
    ans = alloc_chrdev_region(&dev, 0, 1, MY_DEV_NAME);  // Allocates one device number
    if (ans < 0)
    {
        pr_info("Error in major:minor allocation!\n");
        return -1;
    }
    pr_info("major:minor %d:%d allotted!\n", MAJOR(dev), MINOR(dev));
    return 0;
}

/*
** Module exit function
*/
static void __exit hello_world_exit(void)
{
    pr_info("Goodbye world from mod32!\n");
    unregister_chrdev_region(dev, 1);  // Free the allocated major:minor numbers
    pr_info("major:minor numbers freed up...\n");
}

module_init(hello_world_init);
module_exit(hello_world_exit);

MODULE_LICENSE("GPL");
MODULE_AUTHOR("EDD <edd@cdac.gov.in>");
MODULE_DESCRIPTION("Major:Minor dynamic allocation module!");
```

Compile and check all stages .

Introduction to Device Files

Device Files in Linux
Device files allow transparent communication between user-space applications and hardware through the kernel's device drivers. They provide a simple and standard way to interact with devices.

Key Characteristics of Device Files

1.Not Regular Files:

- Although they look like normal files, they are not normal files, they are special files.
- Applications can perform operations like read, write, and mmap on device files, just like regular files.
- The kernel recognizes these operations and forwards them to the device driver, which communicates with the hardware.

2.Abstraction:

- Device files hide the complexity of the hardware from applications.
- Programmers don’t need to know the technical details of the hardware to interact with it.

3.Kernel Involvement:

- Device drivers that handle the operations are part of the Linux kernel.

Location of Device Files

All device files are stored in the `/dev` directory.
To view these files, use the command:

```
bash
ls -l /dev/
```

Examples of Device Files

- **/dev/ttyS0:** Represents the first serial port (COM1 in MS-DOS).
- **/dev/hda2:** Represents the second partition on the first IDE drive.

Device File Permissions

When you use ls -l to list files in /dev, you'll see entries like this:

```
bash
crw--w---- 1 root tty 4, 0 Aug 15 10:40 tty0
brw-rw---- 1 root disk 1, 0 Aug 15 10:40 ram0
```

Explanation:

1.The first letter of the permission field indicates the type of device:

- **c:** Character device (e.g., serial ports, keyboards).
- **b:** Block device (e.g., hard disks, SSDs).

2.The size field is replaced by two numbers:

- **Major Number:** Identifies the device driver handling the device.
- **Minor Number:** Specifies the device instance managed by the driver.

How Device Files Work

- When an application accesses a device file, the kernel looks at the major number to identify the appropriate driver.
- The minor number helps the driver determine which specific device is being accessed.
- The driver then performs the required operation (e.g., reading from or writing to the hardware).

Creating Device Files

Device files can be created in two ways:

1.Manually
2.Automatically

In this section, we focus on manually creating device files.

1.Manually Creating a Device File

We can manually create a device file using the **mknod** command. This method is useful when you want to set up the device file before the driver loads or need flexibility in creating it.

Command Syntax:

```
bash

mknod -m <permissions> <name> <device type> <major> <minor>
```

Here:

- **<name>:** The name of the device file, including the full path (e.g., /dev/my_device).

- **<device type>:** Type of device file:
  - **c:** For character devices.
  - **b:** For block devices.

- **<major>:** The major number assigned to your driver (identifies the driver).

- **<minor>: T**he minor number assigned to the device (identifies a specific device instance).

- **-m <permissions>:** (Optional) Set permissions during file creation. You can also set permissions later using chmod.

Example Command

To create a character device file named /dev/etx_device with major number 246 and minor number 0, use:

```
bash
Copy code
sudo mknod -m 666 /dev/etx_device c 246 0
```

666: Grants read and write permissions to all users.

If permissions are not specified during creation, you can use the chmod command to modify them:

```
bash
Copy code
sudo chmod 666 /dev/etx_device
```

Advantages of Manual Creation

1. You can create the device file even before loading the driver.
2. It provides flexibility—anyone with the required permissions can create the device file.

Rules for Manually Creating Device Files

1.Match Major and Minor Numbers

- Ensure the major number matches the driver registered in the kernel.
- The minor number should correspond to a specific device instance handled by the driver.

2.Use Correct Device Type

- Use c for character devices and b for block devices when specifying the device type in the mknod command.

3.File Name and Path

- Always create the device file inside the /dev/ directory (e.g., /dev/my_device). This is the standard location for all device files.

4.Set Appropriate Permissions

- Permissions must be set carefully to control access to the device. For example:
  - 666: Allows all users to read and write.
  - 660: Limits read and write to the owner and group.

5.Driver Must Be Loaded

- Ensure that the corresponding driver is loaded in the kernel before using the device file. Otherwise, user-space applications cannot communicate with the hardware.

6.Avoid Conflicts

- Check that the major and minor numbers do not conflict with existing device files. Use ls -l /dev/ to confirm.

7.Test the Device File

- After creating the file, test it by performing read or write operations using simple user-space programs.

Programming Example

```c
#include <linux/kernel.h>
#include <linux/init.h>
#include <linux/module.h>
#include <linux/kdev_t.h>
#include <linux/fs.h>

dev_t dev = 0;

/* Module initialization function */
static int __init hello_world_init(void)
{
    /* Allocating a major number dynamically */
    if ((alloc_chrdev_region(&dev, 0, 1, "Embetronicx_Dev")) < 0) {
        pr_err("Cannot allocate major number for device\n");
        return -1;
    }
    pr_info("Kernel Module Inserted Successfully...\n");
    return 0;
}

/* Module cleanup function */
static void __exit hello_world_exit(void)
{
    unregister_chrdev_region(dev, 1);
    pr_info("Kernel Module Removed Successfully...\n");
}

module_init(hello_world_init);
module_exit(hello_world_exit);

MODULE_LICENSE("GPL");
MODULE_AUTHOR("AmbeTronicS <embetronicx@gmail.com>");
MODULE_DESCRIPTION("Simple Linux driver (Manually Creating a Device file)");
MODULE_VERSION("1.1");
```

Steps to Compile, Load, and Create the Device File

1.Compile the Driver

Use the Makefile to build the driver:

```
bash
Copy code
sudo make
```

2.Insert the Driver Module
Load the compiled module into the kernel:

```
bash
sudo insmod driver.ko
```

3.Verify the Device File
Check the /dev directory:

```
bash
ls -l /dev/
```

At this point, the device file is not yet created.

4.Manually Create the Device File
Use the mknod command to create the device file:

```
bash
sudo mknod -m 666 /dev/etx_device c 246 0
```

5.Verify Creation
List the /dev directory again to confirm the file was created:

```
bash
ls -l /dev/ | grep "etx_device"
```

You should see an output similar to:

```
bash
crw-rw-rw- 1 root root 246, 0 Aug 15 13:53 etx_device
```

6.Unload the Driver
Remove the module when you're done:

```
bash
sudo rmmod driver
```

2.Automatically Creating Device File

In Linux, you can automate the creation of device files using udev, a device manager that dynamically handles device nodes in the /dev directory. This method is simpler and reduces manual work. Below are the detailed steps and concepts related to automatically creating device files.

Steps to Automatically Create a Device File

1.Include Required Headers

- Include the necessary kernel headers for device creation:

```
#include <linux/device.h>
#include <linux/kdev_t.h>
```

2.Allocate Major and Minor Numbers

- Use alloc_chrdev_region() to dynamically allocate a major number for your device.

```
if (alloc_chrdev_region(&dev, 0, 1, "etx_Dev") < 0) {
    pr_err("Cannot allocate major number for device\n");
    return -1;
}
```

3.Create a Struct Class

- Use class_create() to create a device class, which organizes device entries in /sys/class/.

```
dev_class = class_create(THIS_MODULE, "etx_class");
if (IS_ERR(dev_class)) {
    pr_err("Cannot create the struct class for device\n");
    goto r_class;}
```

4.Create the Device File

- Use device_create() to register the device with the class. This automatically creates a device file in /dev/.

```
if (IS_ERR(device_create(dev_class, NULL, dev, NULL, "etx_device"))) {
    pr_err("Cannot create the Device\n");
    goto r_device;
}
```

5.Clean Up on Exit

- Use device_destroy() and class_destroy() to clean up resources during module removal.

```
device_destroy(dev_class, dev);
class_destroy(dev_class);
unregister_chrdev_region(dev, 1);
```

Program: Automatically Creating a Device File

Below is a simple Linux kernel module for automatic device file creation.

```
#include <linux/kernel.h>
#include <linux/init.h>
#include <linux/module.h>
#include <linux/kdev_t.h>
#include <linux/fs.h>
#include <linux/device.h>

dev_t dev = 0;
static struct class *dev_class;

/* Module init function */
static int __init hello_world_init(void)
{
    /* Allocate Major Number */
    if ((alloc_chrdev_region(&dev, 0, 1, "etx_Dev")) < 0) {
        pr_err("Cannot allocate major number for device\n");
        return -1;
    }
    pr_info("Major = %d Minor = %d\n", MAJOR(dev), MINOR(dev));

    /* Create Struct Class */
    dev_class = class_create(THIS_MODULE, "etx_class");
    if (IS_ERR(dev_class)) {
        pr_err("Cannot create the struct class for device\n");
        goto r_class;
    }

    /* Create Device */
    if (IS_ERR(device_create(dev_class, NULL, dev, NULL, "etx_device"))) {
        pr_err("Cannot create the Device\n");
        goto r_device;
    }
    pr_info("Kernel Module Inserted Successfully...\n");
    return 0;

r_device:
    class_destroy(dev_class);
r_class:
    unregister_chrdev_region(dev, 1);
    return -1;
}

/* Module exit function */
static void __exit hello_world_exit(void)
{
    device_destroy(dev_class, dev);
    class_destroy(dev_class);
    unregister_chrdev_region(dev, 1);
    pr_info("Kernel Module Removed Successfully...\n");
}

module_init(hello_world_init);
module_exit(hello_world_exit);

MODULE_LICENSE("GPL");
MODULE_AUTHOR("EmbeTronicX <embetronicx@gmail.com>");
MODULE_DESCRIPTION("Simple linux driver (Automatically Creating a Device file)");
MODULE_VERSION("1.2");
```

Key Functions Explained

1.class_create()

- Creates a device class that appears in /sys/class/.

```
c
Copy code
struct class *class_create(struct module *owner, const char
*name);
```

- **Parameters:**
    - owner: Points to the module creating the class.
    - name: The class name.

Example:

```
c
dev_class = class_create(THIS_MODULE, "etx_class");
```

2.device_create()

- Registers the device with the class and creates an entry in /dev/.

```
c
Copy code
struct device *device_create(struct class *class, struct device *parent, dev_t dev, void
*drvdata, const char *fmt, ...);
```

- Parameters:
    - class: Pointer to the class created earlier.
    - parent: Pointer to a parent device (if any).
    - dev: Device number (major and minor).
    - drvdata: Driver-specific data (optional).
    - fmt: Device file name format.

Example:

```
c
device_create(dev_class, NULL, dev, NULL, "etx_device");
```

3.Cleanup Functions

- class_destroy(): Frees the class.

```c
c

void class_destroy(struct class *class);
```

- device_destroy(): Removes the device.

```c
c

void device_destroy(struct class *class, dev_t dev);
```

Steps to Test the Program
1.Build the Driver

- **Use make to compile the driver:**

```bash
bash

sudo make
```

2.Insert the Module

Load the driver into the kernel:

```bash
bash

sudo insmod driver.ko
```

3.Verify the Device File

Check the created device file:

```bash
bash

ls -l /dev/ | grep "etx_device"
```

Example Output:

```
lua

crw------- 1 root root 246, 0 Aug 15 13:36 etx_device
```

4.Remove the Module

Unload the driver:

```
bash

sudo rmmod driver
```

Advantages of Automatic Device File Creation

1. No manual intervention is needed for creating the device file.
2. Simplifies driver deployment by automating repetitive tasks.
3. Ensures the device file is always created with the correct major/minor numbers and permissions.
4. Works seamlessly with udev for dynamic device management.

—---

Cdev structure and File Operations

Purpose: Character device driver provides a way for user-space applications (like programs we write) to interact with hardware devices. These devices are typically associated with a single character stream, like serial port or keyboard.

Components for communication:
To interaction between user-application to kernel drivers we have to use two structures as follow:

1. Cdev structure : This structure represents a character device in the kernel. It contains information about the device, such as minor and major numbers. So the struct cdev is used to represent a character device.

- It acts as a bridge between the kernel and the character device driver we write and the struct cdev is like a "registration book" where the kernel notes down all the information it needed to handle our character device.
- This structure is linked with the inode of the device.**(An inode is a data structure in the kernel uses to manage files.)**

Fields in struct cdev:

```
c
struct cdev {
    struct kobject kobj;                // Kernel object, used for sysfs interaction.
    struct module *owner;               // Module that owns this `cdev` (usually
`THIS_MODULE`).
    const struct file_operations *ops;  // Pointer to the file_operations structure.
    struct list_head list;              // Links multiple cdevs (not often used directly).
    dev_t dev;                          // Device number (Major + Minor).
    unsigned int count;                 // Number of device numbers associated with this
cdev.
};
```

Key Fields in Simple Terms:

- **owner:** Prevents the driver module from being unloaded while the device is in use. Always set this to THIS_MODULE.
- **ops:** Points to your file_operations structure, which defines the functions your driver will handle (like open, read, write).
- **dev:** Contains the device number (Major and Minor).
- **count:** Specifies how many minor numbers this device covers.

Cdev Allocation and Initialization.

The structure struct cdev is essential in linux character device driver as it represents the character device in the kernel at /dev directory.
For declaring a cdev There are a two way allocate the struct cdev as follow:

1.Dynamic allocation

We use cdev_alloc() to allocate memory for the struct cdev dynamically at runtime.

Code example:

```
struct cdev *my_cdev = cdev_alloc();  // Allocate memory for struct cdev
my_cdev->ops = &my_fops;              // Link cdev to file_operations
```

Explanation:
cdev_alloc():

- Allocates memory for a struct cdev from the kernel heap.
- Returns a pointer to the allocated structure.

`my_cdev->ops = &my_fops;`:

- The ops field in struct cdev is assigned the address of a file_operations structure.
- This tells the kernel which functions to call when user-space interacts with the device (e.g., open, read, write).

Pros of Dynamic Allocation:

- Useful when your driver needs to create multiple devices at runtime or when the number of devices isn't fixed.
- Reduces kernel memory usage if the cdev is needed only under specific conditions.

Cons of Dynamic Allocation:

- We must ensure that the dynamically allocated cdev is freed during cleanup (kfree() or implicitly by cdev_del()).

2.Static Allocation:

In this method, we declare the struct cdev as a static/global variable, so its memory is allocated at compile time and remains fixed.

Code Example:

```
static struct cdev my_cdev;           // Statically allocated struct cdev
cdev_init(&my_cdev, &my_fops);        // Initialize cdev with file_operations
```

Explanation:

Static Allocation

- The struct cdev is part of the driver's global/static data.
- It is automatically allocated by the compiler and linked to the driver's lifetime.

cdev_init() function for initialization:

- Initializes the statically allocated struct cdev by associating it with a file_operations structure.
- Does not register the cdev with the kernel yet. This step just sets up the structure.

```
void cdev_init(struct cdev *cdev, const struct file_operations *fops);
```

- cdev: Pointer to the struct cdev you want to initialize.
- fops: Pointer to the file_operations structure that defines the device's behavior.

Pros of Static Allocation:

1. Simpler to use in cases where the number of devices is fixed.
2. Memory management is easier since there's no need to explicitly free the memory.

Cons of Static Allocation:

1. May consume unnecessary memory if the cdev is not always used (since it exists for the driver's entire lifetime).

Registration of cdev and Remove the unregister character device.

1. cdev_add()

This function registers a character device with the kernel, making it accessible to user space through the device file (e.g., /dev/mydevice). It's a critical step in the lifecycle of a struct cdev.

```
int cdev_add(struct cdev *cdev, dev_t dev, unsigned int count);
```

Parameters:

1.struct cdev *cdev

Pointer to the struct cdev object you want to add. This must already be initialized using cdev_init().

2.dev_t dev

The device number (major and minor) for the character device. You typically allocate this using alloc_chrdev_region() or set it manually with MKDEV().

3.unsigned int count

Number of contiguous device numbers. Usually 1, unless you're registering multiple devices at once.

Return Value:

- 0 on success.
- Negative error code (e.g., -ENOMEM, -EINVAL) on failure.

What It Does:

- Registers the device with the kernel so it knows about the device and associates it with the provided dev_t number.
- Links the device number to the file operations defined in the struct cdev object.
- Once added, the device can be accessed via user-space tools (e.g., open(), read(), write()).

Example:

```
// Assume 'my_cdev' is initialized and 'my_fops' is set
struct cdev my_cdev;
dev_t dev;

// Allocate device numbers
alloc_chrdev_region(&dev, 0, 1, "mydevice");

// Initialize cdev
cdev_init(&my_cdev, &my_fops);

// Add the cdev to the kernel
if (cdev_add(&my_cdev, dev, 1) < 0) {
    pr_err("Failed to add cdev\n");
    unregister_chrdev_region(dev, 1);
}
```

2. cdev_del()

This function removes a previously registered character device from the kernel. It is the counterpart to cdev_add().

What It Does:

- Unregisters the device from the kernel, making it inaccessible to user space.
- Frees any internal resources allocated during cdev_add().
- After calling cdev_del(), the associated dev_t is no longer linked to the device, and operations like open() will fail.

```
void cdev_del(struct cdev *cdev);
```

Parameters:

struct cdev *cdev

- Pointer to the struct cdev object to remove. This must be a device that was successfully registered using cdev_add().

Return Value:

- None. This is a void function.

Summary of cdev_add() and cdev_del():

- cdev_add(): Registers your device with the kernel, linking it to a dev_t and making it accessible from user space.
- cdev_del(): Unregisters the device, freeing resources and making the dev_t invalid.

Without cdev_add(), the kernel doesn't know about your device. Without cdev_del(), the kernel might still reference your device, leading to potential errors when unloading the driver.

Linux device driver example that demonstrates how to create, register, and manage a character device.

```c
/***************************************************************************//**
*  \file       driver.c
*
*  \details    Simple Linux device driver (File Operations)
*
*  \author     EmbeTronicX
*
*  \Tested with Linux raspberrypi 5.10.27-v7l-embetronicx-custom+
*******************************************************************************/

#include <linux/kernel.h>       // Kernel log functions
#include <linux/init.h>         // __init and __exit macros
#include <linux/module.h>       // Essential module macros
#include <linux/kdev_t.h>       // Major and minor number macros
#include <linux/fs.h>           // File operations structure
#include <linux/err.h>          // Error handling functions
#include <linux/cdev.h>         // Character device functions
#include <linux/device.h>       // Device creation functions

/* Global Variables */
dev_t dev = 0;                  // Device major and minor numbers
static struct class *dev_class; // Device class
static struct cdev etx_cdev;    // Character device structure

/*
** Function Prototypes
*/
static int __init etx_driver_init(void);
static void __exit etx_driver_exit(void);
static int etx_open(struct inode *inode, struct file *file);
static int etx_release(struct inode *inode, struct file *file);
static ssize_t etx_read(struct file *filp, char __user *buf, size_t len, loff_t *off);
static ssize_t etx_write(struct file *filp, const char __user *buf, size_t len, loff_t *off);

/* File Operations Structure */
static struct file_operations fops = {
    .owner = THIS_MODULE,       // Owner of the module
    .read = etx_read,           // Read operation
    .write = etx_write,         // Write operation
    .open = etx_open,           // Open operation
    .release = etx_release,     // Close operation
};

/* File Operation Functions */

/*
** Open function: Called when the device is opened
*/
static int etx_open(struct inode *inode, struct file *file) {
    pr_info("Driver Open Function Called...!!!\n");
    return 0; // Always succeeds
}

/*
** Release function: Called when the device is closed
*/
static int etx_release(struct inode *inode, struct file *file) {
    pr_info("Driver Release Function Called...!!!\n");
    return 0; // Always succeeds
}

/*
** Read function: Called when data is read from the device
*/
static ssize_t etx_read(struct file *filp, char __user *buf, size_t len, loff_t *off) {
    pr_info("Driver Read Function Called...!!!\n");
    return 0; // Indicates end-of-file
}

/*
** Write function: Called when data is written to the device
*/
static ssize_t etx_write(struct file *filp, const char __user *buf, size_t len, loff_t *off) {
    pr_info("Driver Write Function Called...!!!\n");
    return len; // Acknowledges the data length written
}
```

```c
/* Module Initialization Function */

/*
** Module init: Sets up the device and registers it
*/
static int __init etx_driver_init(void) {
    pr_info("Initializing the Device Driver...\n");

    /* Allocate Major and Minor Numbers */
    if ((alloc_chrdev_region(&dev, 0, 1, "etx_Dev")) < 0) {
        pr_err("Cannot allocate major number\n");
        return -1;
    }
    pr_info("Major = %d, Minor = %d\n", MAJOR(dev), MINOR(dev));

    /* Initialize the cdev Structure */
    cdev_init(&etx_cdev, &fops);

    /* Add the cdev to the Kernel */
    if (cdev_add(&etx_cdev, dev, 1) < 0) {
        pr_err("Cannot add the device to the system\n");
        goto r_class;
    }

    /* Create a Class */
    if (IS_ERR(dev_class = class_create(THIS_MODULE, "etx_class"))) {
        pr_err("Cannot create the struct class\n");
        goto r_class;
    }

    /* Create a Device Node in /dev */
    if (IS_ERR(device_create(dev_class, NULL, dev, NULL, "etx_device"))) {
        pr_err("Cannot create the Device\n");
        goto r_device;
    }

    pr_info("Device Driver Inserted Successfully...!!!\n");
    return 0;

r_device:
    class_destroy(dev_class);
r_class:
    unregister_chrdev_region(dev, 1);
    return -1;
}

/* Module Exit Function */

/*
** Module exit: Cleans up the device and unregisters it
*/
static void __exit etx_driver_exit(void) {
    /* Destroy the Device Node */
    device_destroy(dev_class, dev);

    /* Destroy the Class */
    class_destroy(dev_class);

    /* Remove the cdev from the Kernel */
    cdev_del(&etx_cdev);

    /* Unregister Major and Minor Numbers */
    unregister_chrdev_region(dev, 1);

    pr_info("Device Driver Removed Successfully...!!!\n");
}

/* Module Metadata */
module_init(etx_driver_init);
module_exit(etx_driver_exit);

MODULE_LICENSE("GPL");
MODULE_AUTHOR("EmbeTronicX <embetronicx@gmail.com>");
MODULE_DESCRIPTION("Simple Linux device driver (File Operations)");
MODULE_VERSION("1.3");
```

1. Include Necessary Headers

```
#include <linux/kernel.h>       // Kernel log functions
#include <linux/init.h>         // __init and __exit macros
#include <linux/module.h>       // Essential module macros
#include <linux/kdev_t.h>       // Major and minor number macros
#include <linux/fs.h>           // File operations structure
#include <linux/err.h>          // Error handling functions
#include <linux/cdev.h>         // Character device functions
#include <linux/device.h>       // Device creation functions
```

These headers provide access to essential kernel APIs, such as device registration, logging, and module initialization.

2. Declare Global Variables

```
/* Global Variables */
dev_t dev = 0;                  // Device major and minor numbers
static struct class *dev_class; // Device class
static struct cdev etx_cdev;    // Character device structure
```

- dev_t dev: Stores the major and minor numbers assigned to the device.
- struct class *dev_class: Represents a device class for grouping related devices under /sys/class.
- struct cdev etx_cdev: Represents the character device.

3. File Operations

```
/* File Operations Structure */
static struct file_operations fops = {
    .owner = THIS_MODULE,       // Owner of the module
    .read = etx_read,           // Read operation
    .write = etx_write,         // Write operation
    .open = etx_open,           // Open operation
    .release = etx_release,     // Close operation
};
```

- fops: Links the file operations (e.g., open, read, write, release) to this device driver.
- Each function defined here will be invoked when the corresponding operation is performed on the device file.

File Operation Functions:

```c
/* File Operation Functions */

/*
** Open function: Called when the device is opened
*/
static int etx_open(struct inode *inode, struct file *file) {
    pr_info("Driver Open Function Called...!!!\n");
    return 0; // Always succeeds
}

/*
** Release function: Called when the device is closed
*/
static int etx_release(struct inode *inode, struct file *file) {
    pr_info("Driver Release Function Called...!!!\n");
    return 0; // Always succeeds
}

/*
** Read function: Called when data is read from the device
*/
static ssize_t etx_read(struct file *filp, char __user *buf, size_t len, loff_t *off) {
    pr_info("Driver Read Function Called...!!!\n");
    return 0; // Indicates end-of-file
}

/*
** Write function: Called when data is written to the device
*/
static ssize_t etx_write(struct file *filp, const char __user *buf, size_t len, loff_t *off) {
    pr_info("Driver Write Function Called...!!!\n");
    return len; // Acknowledges the data length written
}
```

File Operation Functions:

1.etx_open(): Called when the device file is opened.

```c
pr_info("Driver Open Function Called...!!!\n");
```

Logs that the device was opened.

2.etx_release(): Called when the device file is closed.

```c
pr_info("Driver Release Function Called...!!!\n");
```

Logs that the device was closed.

3.etx_read(): Called when data is read from the device.

```c
pr_info("Driver Read Function Called...!!!\n");
return 0;
```

Logs the read request and returns 0 (no data is returned in this example).

4.etx_write(): Called when data is written to the device.

```c
pr_info("Driver Write Function Called...!!!\n");
return len;
```

Logs the write request and returns the length of the data written.

4. Module Initialization (etx_driver_init)

```
/* Module Initialization Function */

/*
** Module init: Sets up the device and registers it
*/
static int __init etx_driver_init(void) {
    pr_info("Initializing the Device Driver...\n");

    /* Allocate Major and Minor Numbers */
    if ((alloc_chrdev_region(&dev, 0, 1, "etx_Dev")) < 0) {
        pr_err("Cannot allocate major number\n");
        return -1;
    }
    pr_info("Major = %d, Minor = %d\n", MAJOR(dev), MINOR(dev));

    /* Initialize the cdev Structure */
    cdev_init(&etx_cdev, &fops);

    /* Add the cdev to the Kernel */
    if (cdev_add(&etx_cdev, dev, 1) < 0) {
        pr_err("Cannot add the device to the system\n");
        goto r_class;
    }

    /* Create a Class */
    if (IS_ERR(dev_class = class_create(THIS_MODULE, "etx_class"))) {
        pr_err("Cannot create the struct class\n");
        goto r_class;
    }

    /* Create a Device Node in /dev */
    if (IS_ERR(device_create(dev_class, NULL, dev, NULL, "etx_device"))) {
        pr_err("Cannot create the Device\n");
        goto r_device;
    }

    pr_info("Device Driver Inserted Successfully...!!!\n");
    return 0;

r_device:
    class_destroy(dev_class);
r_class:
    unregister_chrdev_region(dev, 1);
    return -1;
}
```

This function runs when the module is loaded using insmod (`__init etx_driver_init(void)`)

Step-by-step Flow:

1.Allocate Major and Minor Numbers:

```
c

if((alloc_chrdev_region(&dev, 0, 1, "etx_Dev")) < 0){
    pr_err("Cannot allocate major number\n");
    return -1;
}
pr_info("Major = %d Minor = %d \n", MAJOR(dev), MINOR(dev));
```

- Allocates a major number dynamically.
- Minor numbers start from 0.
- Logs the assigned major and minor numbers.

2.Initialize the cdev Structure:

```
c
cdev_init(&etx_cdev, &fops);
```

Initializes the cdev structure and links it with the fops.

3.Add the Device to the System:

```
c
if((cdev_add(&etx_cdev, dev, 1)) < 0){
    pr_err("Cannot add the device to the system\n");
    goto r_class;
}
```

- Registers the device with the kernel.
- Links the device number to the etx_cdev.

4.Create a Device Class:

```
c
if(IS_ERR(dev_class = class_create(THIS_MODULE, "etx_class"))){
    pr_err("Cannot create the struct class\n");
    goto r_class;
}
```

- Creates a class named etx_class in /sys/class for grouping related devices.
- The class is used to simplify device creation.

5.Create the Device File:

```
c
if(IS_ERR(device_create(dev_class, NULL, dev, NULL, "etx_device"))){
    pr_err("Cannot create the Device 1\n");
    goto r_device;
}
```

- Creates a device file /dev/etx_device.
- Links the file with the etx_cdev.

6.Log Successful Initialization:

```
c
pr_info("Device Driver Insert...Done!!!\n");
```

Logs that the driver has been successfully inserted.

Error Handling:

- If any step fails, the code releases allocated resources using goto.

5.Module Exit

```
/* Module Exit Function */

/*
** Module exit: Cleans up the device and unregisters it
*/
static void __exit etx_driver_exit(void) {
    /* Destroy the Device Node */
    device_destroy(dev_class, dev);

    /* Destroy the Class */
    class_destroy(dev_class);

    /* Remove the cdev from the Kernel */
    cdev_del(&etx_cdev);

    /* Unregister Major and Minor Numbers */
    unregister_chrdev_region(dev, 1);

    pr_info("Device Driver Removed Successfully...!!!\n");
}
```

5. Module Exit (etx_driver_exit)

This function runs when the module is removed using rmmod.

Step-by-step Flow:
1.Destroy the Device File:

```
c
device_destroy(dev_class, dev);
```

Removes the /dev/etx_device file.

2.Destroy the Device Class:

```
c
class_destroy(dev_class);
```

Removes the etx_class from /sys/class.

3.Remove the cdev from the Kernel:

```
c
cdev_del(&etx_cdev);
```

Unregisters the cdev.

4.Free Major and Minor Numbers:

```
c
unregister_chrdev_region(dev, 1);
```

Free the device numbers.

5.Log Successful Cleanup:

```
c
pr_info("Device Driver Remove...Done!!!\n");
```

Logs that the driver has been successfully removed.

6. Module Macros

```
c
Copy code
module_init(etx_driver_init);
module_exit(etx_driver_exit);

MODULE_LICENSE("GPL");
MODULE_AUTHOR("EmbeTronicX <embetronicx@gmail.com>");
MODULE_DESCRIPTION("Simple Linux device driver (File Operations)");
MODULE_VERSION("1.3");
```

- module_init: Specifies the initialization function to run when the module is loaded.
- module_exit: Specifies the cleanup function to run when the module is removed.
- Metadata: Includes the license, author, description, and version.

Program Flow Overview

1.Load the Driver (Initialization):

- Allocate device numbers.
- Initialize and add the cdev.
- Create a device class and device file.
- The driver is ready to handle user-space operations.

2.Perform File Operations:

- User-space applications can open, read, write, and close /dev/etx_device.
- Corresponding functions (etx_open, etx_read, etc.) are invoked.

3.Unload the Driver (Cleanup):

- Remove the device file.
- Destroy the class and cdev.
- Free the device numbers

Practical Steps for Testing:

1.Compile the Driver:

```
bash
make
```

2.Insert the Module:

```
bash
sudo insmod driver.ko
```

3.Check Device Creation:

```
bash
ls /dev/etx_device
```

4.Perform File Operations:

```
bash
sudo cat /dev/etx_device    # Read
sudo echo "Hello" > /dev/etx_device  # Write
```

5.Remove the Module:

```
Bash
sudo rmmod driver
```

6.Check Logs:

```
bash
dmesg
```

Creating A Real Device

So we learn about major and minor numbers, device files and file operations of device drivers using dummy drivers. But today we are going to write a real driver without hardware.

We know that in linux everything is a file and we are going to develop two applications as follow:
1. User space application(User program).
2.kernel space program(driver.)

The user program will communicate with the kernel space program using the device file.

Kernel space program(Device Driver)

We are going to file operations in the device driver. Basically there are four functions in the device driver.
1.Open driver
2.write driver
3.read driver
4.close driver

Problem statement:

In this driver we can send string or data to the kernel device driver using the write function. It will store the string in the kernel space. Then when we read the device file it will send the data which is written by write by function to the user space.

Functions used in this driver:

1.Kmalloc():

Kmalloc is a function used to allocate the memory in kernel space. This is like the malloc() function in userspace. Its kmalloc function is used to dynamically allocate the memory in kernel space.

```
#include <linux/slab.h>
void *kmalloc(size_t size, gfp_t flags);
```

Arguments:

1.#include<linux/slab.h>

This directive in the linux kernel module includes the necessary definitions and functions for memory allocation and deallocation in kernel space.

2.size

The number of bytes I want to allocate.

3.flags

Determines the behaviour of memory allocation. Common flag include:

Common gfp_t Flags:

The flags parameter controls how kmalloc() behaves during allocation.

Flags means:

1.basic flags:

1.GFP_KERNEL:

- Used for normal kernel memory allocation.
- May sleep if memory is not immediately available.
- Typically used in non-critical contexts, such as process context.

2.GFP_ATOMIC:

- Used in critical contexts where sleeping is not allowed (e.g., inside interrupt handlers).
- Allocates memory from emergency pools if necessary.

3.GFP_USER:

- Used when memory is allocated on behalf of a user process.
- May sleep.

4.GFP_NOWAIT:

- Allocation does not sleep and returns immediately if memory is unavailable.

5.GFP_DMA:

- Allocates memory suitable for DMA (Direct Memory Access) operations.

Required for devices needing specific physical memory regions.

6.GFP_NOFS:

- Prevents filesystem calls during memory allocation.

It also has some flags that are advanced.

Key characteristics of kmalloc()

a) **Allocated in kernel space:** kernel have its own way to manage memory and kmalloc() provides a way to reserve memory that the kernel modules can use and memory allocated by kmalloc() is not directly accessible by user space application.

b) Fast(usually): Kmalloc() is designed to be efficient in certain cases like when memory is low it may block the execution while waiting for memory to be available.

c) when the kmalloc allocates memory it does not clear or reset the memory it provides.

- The memory might still contain data left over from its previous use.
- We need to clear it if required using **memset()** or similar methods.

Example:

```
char *buffer = kmalloc(100, GFP_KERNEL);
if (buffer) {
    memset(buffer, 0, 100); // Clear the memory to set it to zero.
}
```

memset() funcion: sets a block of memory to a specific value (e.g., filling with zeros or any other byte) and It only changes the contents of the memory, but the memory remains allocated.we can still use the memory after calling memset().

2.kfree function :

The kfree() function is used to release memory that was previously allocated using kmalloc().

```
#include <linux/slab.h>
void kfree(const void *objp);
```

objp: A pointer to the memory block that was returned by kmalloc()

What it does:

- Releases the allocated memory back to the system so that it can be reused.
- After calling kfree(), you can no longer use that memory (the pointer becomes invalid).

Example:

```
c
char *buffer = kmalloc(100, GFP_KERNEL);
kfree(buffer); // Frees the memory
```

If you try to access buffer after kfree(), it can cause a crash or undefined behavior.

3.copy_from_user():

- This function is used in linux programming to copy data from user space(application level to memory) to kernel space(kernel - level memory).
- Its simply transfer data from a user application to the kernel and used when a user program communicates with a kernel module or driver pasing data like commands or config.

Function systex:

```
unsigned long copy_from_user(void *to, const void __user *from, unsigned long n);
```

1.to:

The destination buffer in the kernel space where the data will be copied to.

2.from:

The source buffer in user space that contains the data you want to copy.

3.n:

The number of bytes to copy.

Return Value

- 0: All bytes were successfully copied.
- Non-zero: The number of bytes that could not be copied.

4.copy_to_user()

This function is used to Copy a block of data into userspace (Copy data from kernel space to user space).

```
unsigned long copy_to_user(const void __user *to, const void *from, unsigned long  n);
```

Arguments

- to – Destination address, in the user space
- from – The source address in the kernel space
- n – Number of bytes to copy

Returns a number of bytes that could not be copied. On success, this will be zero.

Kernel space code:

```c
/***************************************************************************//**
*  \file       driver.c
*
*  \details    Simple Linux device driver (Real Linux Device Driver)
*
*  \author     Vicky
*
*  \Tested with Linux Beaglebone black*
*******************************************************************************/
#include <linux/kernel.h>
#include <linux/init.h>
#include <linux/module.h>
#include <linux/kdev_t.h>
#include <linux/fs.h>
#include <linux/cdev.h>
#include <linux/device.h>
#include<linux/slab.h>                 // kmalloc() for memory allocation
#include<linux/uaccess.h>              // copy_to/from_user() for data transfer
#include <linux/err.h>

#define mem_size        1024           // Memory Size for the kernel buffer

dev_t dev = 0;                         // Declare the device number
static struct class *dev_class;        // Declare a pointer to class
static struct cdev etx_cdev;           // Declare the character device structure
uint8_t *kernel_buffer;                // Declare a pointer for the kernel buffer

/*
** Function Prototypes for the operations in the driver
*/
static int      __init etx_driver_init(void);
static void     __exit etx_driver_exit(void);
static int      etx_open(struct inode *inode, struct file *file);
static int      etx_release(struct inode *inode, struct file *file);
static ssize_t  etx_read(struct file *filp, char __user *buf, size_t len, loff_t *off);
static ssize_t  etx_write(struct file *filp, const char *buf, size_t len, loff_t *off);


/*
** File Operations structure that defines how the driver interacts with the device
*/
static struct file_operations fops =
{
        .owner          = THIS_MODULE,      // Defines module ownership
        .read           = etx_read,         // Read operation
        .write          = etx_write,        // Write operation
        .open           = etx_open,         // Open operation
        .release        = etx_release,      // Release operation
};

/*
** This function will be called when we open the Device file
*/
static int etx_open(struct inode *inode, struct file *file)
{
        pr_info("Device File Opened...!!!\n");
        return 0;   // Return 0 if open is successful
}

/*
** This function will be called when we close the Device file
*/
```

```
static int etx_release(struct inode *inode, struct file *file)
{
        pr_info("Device File Closed...!!!\n");
        return 0;   // Return 0 if close is successful
}

/*
** This function will be called when we read the Device file
*/
static ssize_t etx_read(struct file *filp, char __user *buf, size_t len, loff_t *off)
{
        // Copy the data from kernel space to user space
        if( copy_to_user(buf, kernel_buffer, mem_size) )
        {
                pr_err("Data Read: Error in copying data to user space!\n");
        }
        pr_info("Data Read: Done!\n");
        return mem_size;  // Return the number of bytes read
}

/*
** This function will be called when we write to the Device file
*/
static ssize_t etx_write(struct file *filp, const char __user *buf, size_t len, loff_t *off)
{
        // Copy the data from user space to kernel space
        if( copy_from_user(kernel_buffer, buf, len) )
        {
                pr_err("Data Write: Error in copying data from user space!\n");
        }
        pr_info("Data Write: Done!\n");
        return len;  // Return the number of bytes written
}

/*
** Module Init function - Called when the module is loaded into the kernel
*/
static int __init etx_driver_init(void)
{
        /* Allocating a Major number for the device */
        if((alloc_chrdev_region(&dev, 0, 1, "etx_Dev")) < 0){
                pr_info("Cannot allocate major number\n");
                return -1;
        }
        pr_info("Major = %d Minor = %d \n", MAJOR(dev), MINOR(dev));

        /* Creating cdev structure */
        cdev_init(&etx_cdev, &fops);

        /* Adding the character device to the system */
        if((cdev_add(&etx_cdev, dev, 1)) < 0){
            pr_info("Cannot add the device to the system\n");
            goto r_class;
        }

        /* Creating struct class for the device */
        if(IS_ERR(dev_class = class_create(THIS_MODULE, "etx_class"))){
            pr_info("Cannot create the struct class\n");
            goto r_class;
        }

        /* Creating the device */
        if(IS_ERR(device_create(dev_class, NULL, dev, NULL, "etx_device"))){
```

```
            pr_info("Cannot create the device\n");
            goto r_device;
        }

        /* Allocating physical memory for the kernel buffer */
        if((kernel_buffer = kmalloc(mem_size , GFP_KERNEL)) == 0){
            pr_info("Cannot allocate memory in kernel\n");
            goto r_device;
        }

        strcpy(kernel_buffer, "Hello_World");  // Initialize the buffer with a string

        pr_info("Device Driver Insert: Done!!!\n");
        return 0;  // Return success

r_device:
        class_destroy(dev_class);  // Cleanup if device creation failed
r_class:
        unregister_chrdev_region(dev, 1);  // Unregister the device number
        return -1;  // Return failure
}

/*
** Module Exit function - Called when the module is unloaded from the kernel
*/
static void __exit etx_driver_exit(void)
{
        kfree(kernel_buffer);      // Free the allocated memory
        device_destroy(dev_class, dev);  // Destroy the device
        class_destroy(dev_class);  // Destroy the class
        cdev_del(&etx_cdev);       // Delete the character device
        unregister_chrdev_region(dev, 1);  // Unregister the device number
        pr_info("Device Driver Remove: Done!!!\n");
}

module_init(etx_driver_init);  // Register the module init function
module_exit(etx_driver_exit);  // Register the module exit function

MODULE_LICENSE("GPL");
MODULE_AUTHOR("Vicky");  // Updated author name
MODULE_DESCRIPTION("Simple Linux device driver (Real Linux Device Driver)");
MODULE_VERSION("1.4");
```

Instruction:

1.load the driver first then run the user-space application.

2.to see the device loaded enter ls /dev/etx_driver.

User-space application.

```c
/***************************************************************************//**
*  \file       test_app.c
*
*  \details    Userspace application to test the Device driver
*
*  \author     Vicky  // Updated author name
*
*  \Tested with Linux raspberrypi 5.10.27-v7l-embetronicx-custom+
*
*******************************************************************************/
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <unistd.h>

int8_t write_buf[1024];  // Buffer for writing data to the device
int8_t read_buf[1024];   // Buffer for reading data from the device

int main()
{
        int fd;
        char option;

        // Display initial information
        printf("*********************************\n");
        printf("*******WWW.EmbeTronicX.com*******\n");

        // Open the device file for reading and writing
        fd = open("/dev/etx_device", O_RDWR);
        if(fd < 0) {
                // If the device file can't be opened, display an error message
                printf("Cannot open device file...\n");
                return 0;
        }

        // Main menu loop
        while(1) {
                // Display options for the user
                printf("****Please Enter the Option******\n");
                printf("        1. Write                \n");
                printf("        2. Read                 \n");
                printf("        3. Exit                 \n");
                printf("*********************************\n");

                // Get the user's option
                scanf(" %c", &option);
                printf("Your Option = %c\n", option);

                switch(option) {
                        case '1':
                                // Write data to the driver
                                printf("Enter the string to write into driver :");
                                scanf("  %[^\t\n]s", write_buf);  // Read a string with spaces
                                printf("Data Writing ...");
                                write(fd, write_buf, strlen(write_buf)+1);  // Write to the device
                                printf("Done!\n");
                                break;
                        case '2':
```

```
                                // Read data from the driver
                                printf("Data Reading ...");
                                read(fd, read_buf, 1024);  // Read from the device
                                printf("Done!\n\n");
                                printf("Data = %s\n\n", read_buf);  // Display the read data
                                break;
                        case '3':
                                // Close the file descriptor and exit the program
                                close(fd);
                                exit(1);
                                break;
                        default:
                                // If the user enters an invalid option
                                printf("Enter Valid option = %c\n", option);
                                break;
                }
        }

        // Close the file descriptor (this line will never be reached due to exit in option 3)
        close(fd);
}
```

Just run and seen the drivers and user space application.

IOCTL in linux (I/O control)

The operating system divides memory into two main areas: **kernel space** and **user space**.

- Kernel Space: This is where the core parts of the operating system, such as the kernel and device drivers, run. It is protected to ensure system stability and security, meaning user applications cannot directly access this area.
- User Space: This is where user applications and programs run, such as text editors, browsers, and games. The OS can swap data from this area to disk when more memory is needed.

Communication between these two spaces happens through various mechanisms, allowing user applications to interact with the kernel or device drivers. These methods include:

- IOCTL (Input/Output Control): A system call used by user applications to communicate directly with the kernel or device drivers to send commands or control hardware.
- Procfs (Process Filesystem): A virtual filesystem in Linux that provides information about processes and kernel parameters. Applications can read or write files in this filesystem to interact with the kernel.
- Sysfs: A virtual filesystem that exposes information and configuration parameters of kernel objects, like devices and modules, allowing users to query or modify system hardware settings.
- Configfs: A virtual filesystem for configuring kernel parameters, often used for complex configurations and device management.
- Debugfs: A virtual filesystem used for debugging and development, providing kernel logs and debugging information.
- Sysctl: A mechanism to query or modify kernel parameters at runtime, commonly used for configuring kernel behaviors like network settings.
- UDP Sockets: A communication protocol used for sending data over a network, which can also be used for kernel-to-user communication.
- Netlink Sockets: A communication protocol for networking tasks, enabling user-space applications to manage network interfaces, routing tables, and other network-related operations.

These methods enable user applications to control, query, or communicate with the kernel, hardware, or the underlying OS, providing flexibility in managing device interactions, configurations, and network tasks.

Introduction to IOCTL

IOCTL (Input/Output Control) is a system call used to communicate with device drivers in Linux. It is widely used when specific device operations cannot be handled by standard system calls like read() or write(). IOCTL allows user applications to send commands to devices and perform operations that require kernel-level interaction.

Real-time Applications of IOCTL include:

- Ejecting the media from a CD drive.
- Changing the baud rate of a serial port.
- Adjusting the volume of sound devices.
- Reading or writing device-specific registers.
- Changing screen resolution.
- Controlling fan speed.
- Configuring network interface(enableing / Disabling)
- Modifying printer settings
- Toggling led lights
- Accessing power management features
- Managing device ports(USB, Ports)
- Controlling Cameras.
- Adjusting power states
- Enabling / Disabling hardware Features. (ble or wifi)

While the read() and write() functions are used in many cases to interact with devices, they are not sufficient for more specialized device operations. IOCTL provides a mechanism for handling these operations.

Steps Involved in IOCTL:

There are some steps involved to implementing IOCTL in linux Device Driver follow the following steps.

1.Create IOCTL command in the driver.

First, we need to define the IOCTL commands that the driver will handle. These commands are essentially instructions for the driver to perform certain actions.

```
#define IOCTL_NAME __IOX('magic', 'command', 'data type')
```

Parameters:

- 'magic': A unique identifier for your IOCTL commands (often a character or number).
- 'command': A number to distinguish different commands.
- 'data type': The type of data that the IOCTL command will use.

There are four main types for IOCTL commands:

- IO: No parameters.
- IOW: Command with data to write to the driver (copy_from_user).
- IOR: Command that reads data from the driver (copy_to_user).
- IOWR: Command that both reads and writes data.

Example:

```
c
#define WR_VALUE _IOW('a', 'a', int32_t*)
#define RD_VALUE _IOR('a', 'b', int32_t*)
```

The necessary header file:

```
c
#include <linux/ioctl.h>
```

2. Write the IOCTL Function in the Driver

Next, implement the IOCTL function in our driver. This function handles the commands defined in step 1.

The function prototype looks like this:

```
c
int ioctl(struct inode *inode, struct file *file, unsigned int cmd, unsigned long
arg)
```

Parameters:

- inode: Refers to the file being worked on.
- file: The file pointer passed by the user application.
- cmd: The command (e.g., WR_VALUE, RD_VALUE) sent from the user application.
- arg: The arguments sent with the command.

Inside this function, you handle different commands with a switch statement:

```c
c

static long etx_ioctl(struct file *file, unsigned int cmd, unsigned long arg)
{
    switch (cmd) {
        case WR_VALUE:
            // Write value from user to driver
            copy_from_user(&value, (int32_t*)arg, sizeof(value));
            break;
        case RD_VALUE:
            // Read value from driver to user
            copy_to_user((int32_t*)arg, &value, sizeof(value));
            break;
        default:
            pr_info("Invalid command\n");
            break;
    }
    return 0;
}
```

In the file operations structure, you associate the IOCTL function with the driver's unlocked_ioctl field:

```c
c

static struct file_operations fops = {
    .unlocked_ioctl = etx_ioctl,
    // Other file operations like read, write, etc.
};
```

3. Create IOCTL Command in Userspace Application

In the userspace application, define the same IOCTL commands as in the driver:

```c
C

#define WR_VALUE _IOW('a', 'a', int32_t*)
#define RD_VALUE _IOR('a', 'b', int32_t*)
```

4. Use IOCTL System Call in Userspace

Finally, in the userspace application, we call the IOCTL system call to interact with the device. This is how we send commands to the driver from userspace.

The ioctl system call syntax is:

```
c
long ioctl(int fd, unsigned int cmd, unsigned long arg);
```

Where:

- fd: The file descriptor of the device (opened with open()).
- cmd: The IOCTL command to be executed (e.g., WR_VALUE, RD_VALUE).
- arg: The arguments passed to the IOCTL command.

Example:

```
c
ioctl(fd, WR_VALUE, (int32_t*)&number);  // Write data to driver
ioctl(fd, RD_VALUE, (int32_t*)&value);   // Read data from driver
```

Summary

- Define IOCTL commands in the driver.
- Write the IOCTL function to handle those commands.
- Define the same IOCTL commands in the userspace application.
- Use the ioctl system call in the userspace application to interact with the driver.

Key Points to Follow During IOCTL Implementation:

- Unique IOCTL Commands: Use unique magic numbers and command numbers to avoid conflicts.
- Validate Inputs: Always validate input arguments using copy_from_user() and copy_to_user().
- Minimal Kernel Work: Perform only essential operations in kernel space; offload complex tasks to userspace.
- Backward Compatibility: Design commands to support older versions of the driver.
- Error Handling: Return meaningful error codes for failed operations.
- Restrict Access: Use permission checks to secure sensitive commands.
- Non-blocking Operations: Avoid long-running or blocking IOCTL calls; use asynchronous mechanisms if needed.
- Document Commands: Clearly document all IOCTL commands, arguments, and return values.
- Thread Safety: Use synchronization mechanisms to handle concurrent access safely.
- Resource Management: Clean up resources properly to avoid memory leaks or inconsistencies.

Thorough Testing: Test extensively, including edge cases and concurrent calls.
Follow Standards: Adhere to Linux kernel coding guidelines for clarity and maintainability.

Kernel Space Code:

```c
/***************************************************************************//**
*  \file       new_driver.c
*
*  \details    Enhanced Linux device driver (IOCTL)
*
*  \author     YourName
*
*  \Tested with Linux kernel 5.15.0-embedded-custom
*
*******************************************************************************/
#include <linux/kernel.h>
#include <linux/init.h>
#include <linux/module.h>
#include <linux/kdev_t.h>
#include <linux/fs.h>
#include <linux/cdev.h>
#include <linux/device.h>
#include <linux/slab.h>                 // kmalloc()
#include <linux/uaccess.h>              // copy_to/from_user()
#include <linux/ioctl.h>

#define WR_VALUE _IOW('b','x',int32_t*)
#define RD_VALUE _IOR('b','y',int32_t*)

int32_t value = 0;
dev_t dev = 0;
static struct class *dev_class;
static struct cdev my_cdev;

/* Function Prototypes */
static int      __init my_driver_init(void);
static void     __exit my_driver_exit(void);
static int      my_open(struct inode *inode, struct file *file);
static int      my_release(struct inode *inode, struct file *file);
static ssize_t  my_read(struct file *filp, char __user *buf, size_t len,loff_t * off);
static ssize_t  my_write(struct file *filp, const char __user *buf, size_t len, loff_t * off);
static long     my_ioctl(struct file *file, unsigned int cmd, unsigned long arg);

/* File operations structure */
static struct file_operations fops =
{
    .owner          = THIS_MODULE,
    .read           = my_read,
    .write          = my_write,
    .open           = my_open,
    .unlocked_ioctl = my_ioctl,
    .release        = my_release,
};

/* Device open function */
static int my_open(struct inode *inode, struct file *file)
{
    pr_info("My Device File Opened\n");
```

```
    return 0;
}

/* Device release function */
static int my_release(struct inode *inode, struct file *file)
{
    pr_info("My Device File Closed\n");
    return 0;
}

/* Device read function */
static ssize_t my_read(struct file *filp, char __user *buf, size_t len, loff_t *off)
{
    pr_info("My Read Function\n");
    return 0;
}

/* Device write function */
static ssize_t my_write(struct file *filp, const char __user *buf, size_t len, loff_t *off)
{
    pr_info("My Write Function\n");
    return len;
}

/* IOCTL function */
static long my_ioctl(struct file *file, unsigned int cmd, unsigned long arg)
{
    switch(cmd) {
        case WR_VALUE:
            if (copy_from_user(&value, (int32_t*)arg, sizeof(value))) {
                pr_err("Error writing data\n");
            }
            pr_info("Value Written: %d\n", value);
            break;
        case RD_VALUE:
            if (copy_to_user((int32_t*)arg, &value, sizeof(value))) {
                pr_err("Error reading data\n");
            }
            break;
        default:
            pr_info("Invalid Command\n");
            break;
    }
    return 0;
}

/* Module initialization */
static int __init my_driver_init(void)
{
    /* Allocate Major and Minor numbers */
    if ((alloc_chrdev_region(&dev, 0, 1, "my_device")) < 0) {
        pr_err("Failed to allocate major number\n");
        return -1;
    }
    pr_info("Major: %d, Minor: %d\n", MAJOR(dev), MINOR(dev));
```

```
    /* Create cdev structure */
    cdev_init(&my_cdev, &fops);

    /* Add cdev to the system */
    if ((cdev_add(&my_cdev, dev, 1)) < 0) {
        pr_err("Failed to add cdev\n");
        goto r_class;
    }

    /* Create struct class */
    if (IS_ERR(dev_class = class_create(THIS_MODULE, "my_class"))) {
        pr_err("Failed to create class\n");
        goto r_class;
    }

    /* Create device */
    if (IS_ERR(device_create(dev_class, NULL, dev, NULL, "my_device"))) {
        pr_err("Failed to create device\n");
        goto r_device;
    }

    pr_info("Device Driver Inserted Successfully\n");
    return 0;

r_device:
    class_destroy(dev_class);
r_class:
    unregister_chrdev_region(dev, 1);
    return -1;
}

/* Module cleanup */
static void __exit my_driver_exit(void)
{
    device_destroy(dev_class, dev);
    class_destroy(dev_class);
    cdev_del(&my_cdev);
    unregister_chrdev_region(dev, 1);
    pr_info("Device Driver Removed Successfully\n");
}

module_init(my_driver_init);
module_exit(my_driver_exit);

MODULE_LICENSE("GPL");
MODULE_AUTHOR("Vicky");
MODULE_DESCRIPTION("Enhanced Linux Device Driver with IOCTL");
MODULE_VERSION("2.0");
```

User Space Code:

```c
#include <stdio.h>
#include <fcntl.h>
#include <unistd.h>
#include <sys/ioctl.h>
#include <stdint.h>

#define WR_VALUE _IOW('a', 'a', int32_t*)
#define RD_VALUE _IOR('a', 'b', int32_t*)

int main()
{
    int fd;
    int32_t value, number;

    printf("Opening Device...\n");
    fd = open("/dev/my_device", O_RDWR);
    if (fd < 0) {
        perror("Cannot open device");
        return -1;
    }

    printf("Enter the value to send: ");
    scanf("%d", &number);
    printf("Writing value to the device...\n");
    if (ioctl(fd, WR_VALUE, &number) < 0) {
        perror("IOCTL Write Error");
        close(fd);
        return -1;
    }

    printf("Reading value from the device...\n");
    if (ioctl(fd, RD_VALUE, &value) < 0) {
        perror("IOCTL Read Error");
        close(fd);
        return -1;
    }
    printf("Value received from device: %d\n", value);

    printf("Closing Device...\n");
    close(fd);

    return 0;
}
```

Steps to Create and Run the User Space Code
Load the Kernel Module:

1.Insert the kernel module (my_driver.ko) into the kernel:

```
sudo insmod my_driver.ko
```

Check for successful insertion using:

```
dmesg | tail
```

2.Create the Device File:

If not automatically created, manually create the device file:

```
sudo mknod /dev/my_device c <major_number> 0
```

Replace <major_number> with the major number displayed in dmesg or logs during module insertion.

3.Run the Program:
Execute the user-space program:

```
./user_ioctl
```

4.Interact with the Device:

Follow the prompts to send and receive data via the ioctl interface.

5.Verify Operation:

Check the kernel logs for messages using:

```
dmesg | tail
```

6.Clean Up:

After testing, remove the device file and unload the module:

```
sudo rm /dev/my_device
sudo rmmod my_driver
```

Procfs In Linux

Process File System ----it is a runtime interface.
The procfs (Process File System) is a virtual filesystem in Linux that provides an interface to kernel data structures. It does not correspond to physical files on disk but is created dynamically in memory when the system boots.
The procfs is typically mounted at /proc and allows users and applications to query and sometimes modify system and process-specific information.

Key Points:
procfs are a virtual filesystem in linux that shows information about sysstem and running processes.
it is not stored on a disk insted it is created in memory while the system is running.
we can find it mounted at /proc in linux.

1.Provides System Information:
runtime system information, including CPU, memory, devices, and kernel parameters.

Examples:

- /proc/cpuinfo: Information about the CPU.
- /proc/meminfo: Memory usage statistics.
- /proc/version: Kernel version.

2.Exposes Process Information:
Contains a directory for each running process, named by its Process ID (PID).

3.Allows Runtime Configuration:

Some files allow writing to adjust kernel parameters dynamically without requiring a reboot.

4.Tools and location in linux to seen the virtual created process:
Tools: top, ps and htop.
location: /proc/

It act as a bridge between userspace and kernel space read info form kernel. Every file in a proc is provides information about kernel.

Some Example: /proc/ file is provided information such as we have one file called "meminfo" That gives the details of memory used in system just type following command :
cat /proc/meminfo

more examples:
1.cat /proc/module - give info about all modules that are part of kernel.

most important commad is lsmod :- its show the status of modules that are running on kernel modules. all the module that are loaded .

Some of other files inside /proc/ provide a info that most are read-only as given follow:

- /proc/devices — registered character and block major numbers
- /proc/iomem — on-system physical RAM and bus device addresses
- /proc/ioports — on-system I/O port addresses (especially for x86 systems)
- /proc/interrupts — registered interrupt request numbers
- /proc/softirqs — registered soft IRQs
- /proc/swaps — currently active swaps
- /proc/kallsyms — running kernel symbols, including from loaded modules
- /proc/partitions — currently connected block devices and their partitions
- /proc/filesystems — currently active filesystem drivers
- /proc/cpuinfo — information about the CPU(s) on the system.

In some case we can write the proc files.

Main features:
The proc file system is also very useful when we want to debug a kernel module. While debugging we might want to know the values of various variables in the module or maybe the data that the module is handling. In such situations, we can create a proc entry for ourselves and dump whatever data we want to look into in the entry.

whatever data in user space to kernel space we write changes are temporary onces the system reboot the data is change or lost.So depending on that we have two kinds of proc entries.
1.An entry that only reads data from the kernel space.
2.An entry that reads as well as writes data into and from kernel space.

Creating Procfs Entries

Creating a Directory

This means adding a custom folder (subdirectory) under /proc for your kernel module or driver. This is done using the proc_mkdir() function, which helps organize multiple related entries.

```
c
struct proc_dir_entry *proc_mkdir(const char *name, struct proc_dir_entry *parent);
```

Parameters:

- name: Name of the directory.
- parent: Parent directory under /proc. If NULL, the directory is created at the root of /proc.

Creating Proc Files

This involves adding specific files under /proc (or its subdirectories) to expose or interact with your kernel module's information.

```
c
Copy code
struct proc_dir_entry *proc_create(const char *name, umode_t mode, struct proc_dir_entry
*parent, const struct file_operations *proc_fops);
```

Parameters:

- name: Name of the proc entry.
- mode: Access permissions.
- parent: Parent directory pointer. NULL implies root /proc.
- proc_fops: File operations (e.g., read, write, open).

From kernel version 5.6 onwards, proc_fops is replaced with proc_ops.

Header file for create file and directory

```
#include <linux/proc_fs.h>  // For proc file system functions
```

File Operations

For proc entries, file operations are defined in:

```
static struct file_operations proc_fops = {
    .open = open_proc,
    .read = read_proc,
    .write = write_proc,
    .release = release_proc
};
```

Functions for Procfs File Operations

1.Open and Release

The open and release functions are optional:

```
//for opening
static int open_proc(struct inode *inode, struct file *file) {
    printk(KERN_INFO "Proc file opened");
    return 0;
}
For release a proc
static int release_proc(struct inode *inode, struct file *file) {
    printk(KERN_INFO "Proc file released");
    return 0;
}
```

2.Write Operation

Data can be written to the kernel using copy_from_user:

```
static ssize_t write_proc(struct file *filp, const char *buff, size_t len, loff_t *off) {
    printk(KERN_INFO "Proc file write");
    copy_from_user(etx_array, buff, len);
    return len;
}
```

3.Read Operation

Data can be read from the kernel using copy_to_user:

```
static ssize_t read_proc(struct file *filp, char __user *buffer, size_t length, loff_t *offset)
{
    if (copy_to_user(buffer, etx_array, 20)) {
        pr_err("Data read error");
    }
    return length;
}
```

Removing Proc Entries

```
void remove_proc_entry(const char *name, struct proc_dir_entry *parent);
```

For example:

```
remove_proc_entry("etx_proc", NULL);
```

To remove entire directories:

proc_remove(parent);

Wait Queues in Linux

Wait Queues are a kernel mechanism used to put a process to sleep until a certain condition becomes true. This allows the CPU to perform other tasks while a process waits for an event to occur. Once the event happens, the process is woken up and resumes its operation.

Key Concepts:

1.Sleeping and Waking:

- Sleeping: A process suspends its execution when it is waiting for an event (e.g., I/O completion, data availability).
- Waking up: When the awaited event occurs, the process is moved from the sleep state to a runnable state.

2.Why Wait Queues?:

- To efficiently handle situations where a process cannot proceed until an event occurs.
- To prevent busy waiting, where the CPU is unnecessarily consumed by repeatedly checking for an event.

3.Structure of a Wait Queue:

A wait queue is essentially a list (queue) of processes that are waiting for a specific condition to become true. When the condition is met, one or more processes on the queue are woken up.

Example:

1.Inter-Process Communication (IPC)

- What happens? One process wants to send data to another process but must wait for the other process to read the current data first.
- How does it work? The sender sleeps while waiting. Once the receiver processes the data, the sender wakes up to send more data.

2. Multithreading in Kernel

- What happens? Multiple threads are sharing a resource (like a memory buffer). A thread may need to wait if the resource is busy.
- How does it work? The thread sleeps until the resource is free. Once space becomes available, the thread is woken up to use the resource.

These examples, wait queues help processes avoid wasting CPU time by sleeping until they get a signal that the event they’re waiting for has occurred. This makes the system more efficient and faster.

There are three key steps involved in using waitqueues:

1.Initializing a Waitqueue
2.Queuing (Putting Tasks to Sleep)
3.Waking Up Queued Tasks

1. Initializing a Waitqueue

Before using a waitqueue in a Linux kernel, you need to create and initialize it. This sets up the waitqueue structure so that processes can be added to it or woken up later.

Waitqueues are used in the Linux kernel to manage processes that need to wait for certain events to occur. Proper initialization is crucial for their use. There are two types of waitqueue initialization: static initialization and dynamic initialization.
To use a waitqueue, it must first be initialized. Include the header file:

```c
#include <linux/wait.h>
```

Waitqueues are used in the Linux kernel to manage processes that need to wait for certain events to occur. Proper initialization is crucial for their use. There are two types of waitqueue initialization: static initialization and dynamic initialization.

1. Static Initialization

Static initialization sets up the waitqueue at the time of declaration. It is simple and the waitqueue is ready for use immediately after declaration.

Macro for Static Initialization:

```c
DECLARE_WAIT_QUEUE_HEAD(wq);
DECLARE_WAIT_QUEUE_HEAD(wq):
```

- Declares a waitqueue named wq.
- Initializes wq immediately and makes it ready for use.

Use Case

- Suitable for cases where the waitqueue will always be needed and its name is fixed.
- Commonly used for static and global waitqueues in device drivers.

Advantages

- Simpler and requires less code.
- Initialization happens automatically during declaration.

Dynamic initialization

- Declaration of the waitqueue.
- Manual initialization before use.

Steps for Dynamic Initialization

```
c
wait_queue_head_t wq;        // Step 1: Declare
init_waitqueue_head(&wq);    // Step 2: Initialize
```

1.wait_queue_head_t wq;:
- Declares a waitqueue named wq without initializing it.

2.init_waitqueue_head(&wq);:
- Initializes the declared waitqueue wq.
- Makes it ready for use.

Use Case
- Used when the waitqueue is needed only conditionally or later in the code.
- Preferred for dynamically created or allocated waitqueues.

Advantages

- Provides flexibility to control when and where the waitqueue is initialized.
- Useful for creating waitqueues dynamically at runtime.

Visual Analogy

- Static Initialization: Like a pre-assembled, ready-to-use tool that doesn't require setup.
- Dynamic Initialization: Like a modular tool you need to assemble before using.

Queuing (Putting Tasks to Sleep) in Linux Kernel

Queuing involves making a process sleep on a waitqueue until a specific condition becomes true.
This mechanism allows efficient CPU usage by avoiding busy waiting.
Linux provides several macros to implement this functionality based on the requirements. Each macro has specific behavior and return values.

a) wait_event

Purpose

- The process get sleep untile an condition not get change.
- Puts the process to sleep in a TASK_UNINTERRUPTIBLE state until a specified condition evaluates to true.

Syntax

```
wait_event(wq, condition);
```

where:

- wq: The waitqueue the process will sleep on.
- condition: A boolean expression. The process sleeps until this condition evaluates to true.

Key Features

- The condition is checked every time the wait queue is woken up.
- The process cannot be interrupted by signals while sleeping.

Use Case

When the process should only wake up after the event occurs and cannot be interrupted while running the other process in the processor.

Example

```c
DECLARE_WAIT_QUEUE_HEAD(wq);

void example_function(void) {
    int condition_met = 0;

    // Wait until the condition becomes true
    wait_event(wq, condition_met == 1);

    // Continue execution when condition_met becomes true
}
```

b) wait_event_timeout

Purpose

- Puts the process to sleep in a TASK_UNINTERRUPTIBLE state until the condition becomes true or a timeout occurs.

Syntax

```
wait_event_timeout(wq, condition, timeout);
```

- wq: The waitqueue the process will sleep on.
- condition: A boolean expression. The process sleeps until this condition evaluates to true.
- timeout: Timeout duration, specified in jiffies.

Return Values

- 0: The condition was false, and the timeout occurred.
- 1: The condition became true after the timeout elapsed.
- Remaining jiffies: The condition became true before the timeout.

Use Case

When the process should wake up either on event occurrence or after a specific timeout.

Example:

```
DECLARE_WAIT_QUEUE_HEAD(wq);

void example_function(void) {
    int condition_met = 0;
    long timeout = 100; // Timeout in jiffies

    // Wait until condition_met becomes true or timeout occurs
    long remaining = wait_event_timeout(wq, condition_met == 1,
timeout);

    if (remaining > 0)
        printk("Condition met before timeout.\n");
    else
        printk("Timeout occurred.\n");
}
```

c) wait_event_cmd

Purpose

Puts the process to sleep until a condition is true, executing specified commands before and after sleeping.

Syntax

```
wait_event_cmd(wq, condition, cmd1, cmd2);
```

cmd1: Command executed before putting the process to sleep.
cmd2: Command executed after waking up.

Use Case

When additional setup or cleanup operations are needed around the sleep.

Example

```
DECLARE_WAIT_QUEUE_HEAD(wq);

void example_function(void) {
    int condition_met = 0;

    wait_event_cmd(wq, condition_met == 1,
                   printk("Preparing to sleep...\n"),
                   printk("Woke up!\n"));
}
```

d) wait_event_interruptible

Purpose

Puts the process to sleep in a TASK_INTERRUPTIBLE state, allowing it to be interrupted by signals.

Syntax

```
wait_event_interruptible(wq, condition);
```

Return Values

0: The condition became true.
-ERESTARTSYS: The process was interrupted by a signal.

Use Case

When the process must remain responsive to user signals while waiting for an event.

Example:

```
DECLARE_WAIT_QUEUE_HEAD(wq);

void example_function(void) {
    int condition_met = 0;

    int ret = wait_event_interruptible(wq, condition_met == 1);

    if (ret == -ERESTARTSYS)
        printk("Interrupted by signal.\n");
    else
        printk("Condition met.\n");
}
```

e) wait_event_interruptible_timeout

Purpose

- Puts the process to sleep in a TASK_INTERRUPTIBLE state until the condition becomes true, a timeout occurs, or the process is interrupted.

Syntax

- wait_event_interruptible_timeout(wq, condition, timeout);
- timeout: Timeout duration, specified in jiffies.

Return Values

- 0: The condition was false, and the timeout occurred.
- 1: The condition became true after the timeout elapsed.
- Remaining jiffies: The condition became true before the timeout.
- -ERESTARTSYS: The process was interrupted by a signal.

Use Case

When both timeout and interruptibility are required.

Example:

```
DECLARE_WAIT_QUEUE_HEAD(wq);

void example_function(void) {
    int condition_met = 0;
    long timeout = 100;

    int ret = wait_event_interruptible_timeout(wq, condition_met == 1, timeout);

    if (ret == -ERESTARTSYS)
        printk("Interrupted by signal.\n");
    else if (ret == 0)
        printk("Timeout occurred.\n");
    else
        printk("Condition met before timeout.\n");
}
```

f) wait_event_killable

Purpose

Puts the process to sleep in a TASK_KILLABLE state, allowing it to be killed by certain signals.

Syntax

wait_event_killable(wq, condition);

Return Values

0: The condition became true.
-ERESTARTSYS: The process was interrupted by a kill signal.

Use Case

When the process should be terminated only by specific signals (e.g., SIGKILL).

Example

```
DECLARE_WAIT_QUEUE_HEAD(wq);

void example_function(void) {
    int condition_met = 0;

    int ret = wait_event_killable(wq, condition_met == 1);

    if (ret == -ERESTARTSYS)
        printk("Killed by signal.\n");
    else
        printk("Condition met.\n");
}
```

3. Waking Up Queued Tasks

When a task is waiting for an event to happen (sleeping), it can be woken up using specific functions. Here's how each function works:

a) wake_up

- What it does: Wakes up one task that is sleeping in the TASK_UNINTERRUPTIBLE state.
- When to use: When you want to wake up one task that can't be interrupted by signals while sleeping.

Example:

```
wake_up(&wq);
```

If you have a task waiting for an event using wait_event(wq, condition); and the condition becomes true, this will wake up the task.

b) wake_up_all

- What it does: Wakes up all tasks that are sleeping in the TASK_UNINTERRUPTIBLE state.
- When to use: When you want to wake up all tasks waiting on the same event.

Example:

```
wake_up_all(&wq);
```

This will wake up every task waiting on the wq waitqueue, allowing all of them to proceed once the event happens.

c) wake_up_interruptible

- What it does: Wakes up one task that is sleeping in the TASK_INTERRUPTIBLE state, meaning it can be interrupted by signals.
- When to use: When you want to wake up a task that may have been waiting with the possibility of being interrupted by a signal.

Example:

```
wake_up_interruptible(&wq);
```

If a task was waiting using wait_event_interruptible(wq, condition);, this function will wake it up once the condition is true.

d) wake_up_sync and wake_up_interruptible_sync

What they do: These functions wake up tasks similar to wake_up and wake_up_interruptible, but with a twist. They try to avoid immediate rescheduling of the CPU, which can improve performance in some situations.

- wake_up_sync wakes up tasks in TASK_UNINTERRUPTIBLE.
- wake_up_interruptible_sync wakes up tasks in TASK_INTERRUPTIBLE.

When to use: When you want to wake up tasks but avoid the CPU immediately switching to those tasks. This can be useful if you have more work to do before the tasks should start running.

Example:

```
wake_up_sync(&wq);
wake_up_interruptible_sync(&wq);
```

These will wake up the tasks, but the CPU won't immediately reschedule them, allowing the current task to finish some additional work first.

Simple Scenario Example
Imagine a driver that waits for data from a device. It puts tasks to sleep while waiting for the data. Once the data arrives:

- Use wake_up(&wq); to wake up one waiting task.
- Use wake_up_all(&wq); to wake up all waiting tasks if multiple tasks are waiting for the same data.
- Use wake_up_interruptible(&wq); if the waiting tasks could be interrupted by signals.
- Use wake_up_sync(&wq); if you want to wake up tasks but delay their execution slightly, maybe to finish some last-minute work.

—---

1. What is sysfs?

sysfs is a special filesystem in Linux that the kernel uses to communicate information about devices, drivers, and kernel objects to user space. Think of it as a bridge between the kernel and user programs, allowing you to access information and control devices through files in the /sys directory.

2. What are Kernel Objects?

- Kernel Object (kobject): The foundation of sysfs is the kernel object. A kernel object is anything the kernel manages that needs to be represented in the sysfs filesystem (such as a device, driver, or subsystem).

```
#define KOBJ_NAME_LEN 20

struct kobject {
 char *k_name;
 char name[KOBJ_NAME_LEN];
 struct kref kref;
 struct list_head entry;
 struct kobject *parent;
 struct kset *kset;
 struct kobj_type *ktype;
 struct dentry *dentry;
};
```

- struct kobject: In the kernel, a kobject is represented by the structure struct kobject. It includes important information about the object, such as:

    - name: The name of the object.
    - parent: The parent object (if any) of the current object.
    - ktype: The type associated with the object.
    - kref: A reference counter to manage memory.
    - entry: A list head linking it to other objects in the system.

3. Creating Directories in sysfs

When you want to create a directory in sysfs to represent a device or any kernel object, you use the function:

```
struct kobject * kobject_create_and_add(const char *name, struct kobject *parent);
```

- name: The name of the directory to be created (this will appear under /sys/).
- parent: The parent directory for the new directory (could be kernel_kobj for /sys/kernel/).

Example:

```
struct kobject *kobj_ref;
kobj_ref = kobject_create_and_add("etx_sysfs", kernel_kobj);  // Creates
/sys/kernel/etx_sysfs
```

- After the task is done, you can free the kobject memory using kobject_put(kobj_ref).

4. Creating sysfs Files (Attributes)

Once a directory is created for the kernel object, you need to add files (also called attributes) to interact with user space. These files are regular files but are special because they interact with kernel space.

How to create an attribute:

An attribute is a file in sysfs that holds a value. You define an attribute using the kobj_attribute structure:

```
struct kobj_attribute {
    struct attribute attr;  // Basic file information
    ssize_t (*show)(struct kobject *kobj, struct kobj_attribute
*attr, char *buf);
    ssize_t (*store)(struct kobject *kobj, struct kobj_attribute
*attr, const char *buf, size_t count);
};
```

- attr: Represents the file that will be created.
- show: This function is called when you read the file (e.g., using cat).
- store: This function is called when you write to the file (e.g., using echo).

Create a sysfs file using __ATTR macro:

The macro __ATTR is used to define a new sysfs attribute:

__ATTR(name, permission, show_ptr, store_ptr);

- name: Name of the sysfs attribute.
- permission: Permissions for the file (e.g., 0660).
- show_ptr: Pointer to the show function.
- store_ptr: Pointer to the store function.

Example:
We'll create a file named etx_value with read and write functionality.

```
struct kobj_attribute etx_attr = __ATTR(etx_value, 0660,
sysfs_show, sysfs_store);
```

- sysfs_show: This function is called when the file is read.
- sysfs_store: This function is called when data is written to the file.

5. Writing show and store Functions

- show function: This function is used to show the data when reading from the sysfs file.

```
static ssize_t sysfs_show(struct kobject *kobj, struct kobj_attribute
*attr, char *buf) {
    return sprintf(buf, "%d", etx_value);  // Displays the value of
etx_value
}
```

- store function: This function is used to store data when writing to the sysfs file.

```
static ssize_t sysfs_store(struct kobject *kobj, struct
kobj_attribute *attr, const char *buf, size_t count) {
    sscanf(buf, "%d", &etx_value);  // Updates etx_value with the
new input value
    return count;  // Returns the number of bytes written
}
```

6. Creating the Sysfs File
Once the attribute and the show and store functions are ready, you can create the sysfs file using:

```
int sysfs_create_file(struct kobject *kobj, const struct
attribute *attr);
```

- kobj: The kernel object (directory) to associate the attribute with.
- attr: The attribute (file) to be created.

Example:

```
if (sysfs_create_file(kobj_ref, &etx_attr.attr)) {
    printk(KERN_INFO "Cannot create sysfs file...\n");
    goto r_sysfs;
}
```

7. Removing the Sysfs File

When you no longer need the sysfs file, remove it using:

```
sysfs_remove_file(kobj_ref, &etx_attr.attr);
```

8. Complete Example

Let's put all of it together in an example.

Driver Code:

```
// Define the sysfs attribute
struct kobj_attribute etx_attr = __ATTR(etx_value, 0660, sysfs_show, sysfs_store);

// The show function
static ssize_t sysfs_show(struct kobject *kobj, struct kobj_attribute *attr, char *buf) {
    return sprintf(buf, "%d", etx_value);  // Display the value
}

// The store function
static ssize_t sysfs_store(struct kobject *kobj, struct kobj_attribute *attr, const char *buf,
size_t count) {
    sscanf(buf, "%d", &etx_value);  // Store the new value
    return count;
}

// The init function to create sysfs file
static int __init sysfs_driver_init(void) {
    struct kobject *kobj_ref;

    // Create a directory under /sys/kernel/
    kobj_ref = kobject_create_and_add("etx_sysfs", kernel_kobj);

    // Create sysfs file for etx_value
    if (sysfs_create_file(kobj_ref, &etx_attr.attr)) {
        printk(KERN_INFO "Cannot create sysfs file...\n");
        return -ENOMEM;
    }

    return 0;
}
```

```c
// The exit function to remove sysfs file
static void __exit sysfs_driver_exit(void) {
    struct kobject *kobj_ref;

    // Remove the sysfs file
    sysfs_remove_file(kernel_kobj, &etx_attr.attr);
    kobject_put(kobj_ref);  // Free memory
}

module_init(sysfs_driver_init);
module_exit(sysfs_driver_exit);
```

Summary:

- sysfs provides a way for user space to interact with the kernel and devices via files.
- Kernel objects (kobjects) represent various kernel-managed entities and are the basis of sysfs.
- Attributes in sysfs are represented as files and allow you to read/write kernel data from user space.
- Functions like kobject_create_and_add and sysfs_create_file are used to create directories and files in sysfs, while show and store functions handle read/write operations.

Feature	/sys (Sysfs)	/proc (Procfs)	/dev (Devfs)	Column 1
Purpose	Kernel objects and device info	Process and system information	Device interface	
Contents	Kernel objects, device configs	Process directories, system files	Device files	
Interaction	Read/write for config and info	Read (mostly), some writeable files	Read/write for device I/O	
Example	/sys/class/net/ (network devices)	/proc/cpuinfo (CPU details)	/dev/sda (hard drive access)	

INTERRUPTS IN A LINUX KERNEL

Definition:
An interrupt is a signal sent to the processor that temporarily halts the current execution of code, allowing the processor to handle a specific event. Once the event is handled, the processor resumes its previous activity.

Purpose:
Interrupts are used to handle events that require immediate attention, such as hardware signals (keyboard presses, mouse clicks) or software requests (system calls). They help in managing asynchronous events efficiently.

2. Analogy to Real Life

Polling vs Interrupts (Doorbell Analogy):

Polling: Similar to repeatedly checking the door to see if guests have arrived, polling involves continuously checking hardware to see if any event has occurred. This method is inefficient because it wastes CPU time.

Interrupts: Like responding to a doorbell, the CPU can continue other tasks and only check the hardware when an interrupt signal (similar to the doorbell) is received. This makes the process more efficient, as the CPU can do other tasks in the meantime.

3. Interrupt Mechanism
- When an interrupt occurs, the processor stops its current execution.
- It saves the state of the current execution so that it can resume later.
- It then transfers control to a specific function known as an Interrupt Service Routine (ISR) or Interrupt Handler, which handles the event.
- After the ISR completes its task, control is returned to the interrupted process.

4. Interrupt Service Routine (ISR)
- The ISR is a special function in the kernel designed to handle specific interrupt events.
- Each hardware device that can generate an interrupt has a corresponding ISR.
- The ISR must be fast and efficient because it holds up other processes.

5. Polling vs Interrupts
Polling:
The CPU continuously checks each device to see if it needs service.
It is resource-intensive as it consumes CPU cycles even when no event occurs.
Example: A salesperson knocking on every door to check if someone needs something.
Interrupts:
The CPU responds only when a device signals that it needs attention, freeing up CPU time for other tasks.
Example: A shopkeeper waiting for customers to approach when they need something.

Key Takeaways:
- Interrupts improve efficiency by allowing the CPU to handle events as they occur rather than continuously checking for them.
- The use of ISRs ensures that each interrupt is handled quickly and appropriately.
- Polling can be wasteful, whereas interrupts provide a more efficient way to handle asynchronous events.

Interrupts and Exceptions

Interrupts:

- Asynchronous: They occur independently of the processor's current instruction cycle.
- Generated by hardware: Devices like keyboards, mice, or network cards send interrupt signals to the processor to indicate they need attention.

Exceptions:

- Synchronous: They occur in sync with the processor's instruction cycle, meaning they happen as a direct result of executing an instruction.
- Generated by the processor: They are triggered by certain events during instruction execution, such as errors or special conditions.

Comparison Between Interrupts and Exceptions

Timing:

Interrupts happen asynchronously, meaning they can occur at any time, regardless of what the CPU is doing.

Exceptions are synchronous, meaning they happen precisely when a specific instruction is being executed.

Source:

Interrupts come from external hardware.

Exceptions are caused by the processor itself while executing instructions.

Examples of Exceptions

Abnormal Conditions:

Example: A page fault, which occurs when a program accesses a portion of memory that is not currently mapped to physical memory. The kernel needs to handle this by loading the required page into memory.

Handling Mechanism

Both interrupts and exceptions are handled similarly in the kernel:

- When an interrupt or exception occurs, the processor stops its current task and jumps to a specific routine in the kernel to handle it.
- This routine could be an Interrupt Service Routine (ISR) for interrupts or an exception handler for exceptions.

System Calls

- System Calls are a specific type of exception.
- On the x86 architecture, system calls are implemented using software interrupts.
- A software interrupt is issued when a program requests a service from the kernel, such as file operations or process control.
- The software interrupt triggers a trap into the kernel, leading to the execution of a system call handler.

Further Classification
Interrupts and exceptions can be further classified based on their types:

Maskable vs Non-maskable interrupts:

- Maskable: These can be ignored or delayed by the processor.
- Non-maskable: These cannot be ignored and must be handled immediately.

Faults, Traps, Aborts (types of exceptions):

- Faults: Can be corrected, and the instruction can be retried (e.g., page faults).
- Traps: Used to intentionally transfer control to the kernel (e.g., system calls).
- Aborts: Severe errors that usually do not allow the continuation of the process (e.g., hardware failure).

Interrupt Handler (also known as an Interrupt Service Routine or ISR)

Definition

An Interrupt Handler is a function that the Linux kernel executes in response to an interrupt. Each device capable of generating interrupts has a corresponding interrupt handler. This handler is part of the device's driver, which is the kernel code managing that specific device.

Function in Linux

- They follow a specific prototype, which ensures the kernel can pass necessary information to the handler in a standard way.
- What sets interrupt handlers apart from other kernel functions is:
- They are invoked in response to interrupts.
- They run in a special context called interrupt context (or atomic context), where blocking operations are not allowed.

Responsibilities of an Interrupt Handler

At the very least, an interrupt handler must acknowledge the interrupt to the hardware, signaling that the interrupt has been received and will be handled.

The handler might perform various tasks:

- Reading data from or writing data to the device.
- Clearing status registers.
- Initiating other actions required to service the interrupt.

Efficiency Considerations

- For hardware: The operating system must service interrupts promptly to ensure hardware can continue its operations without bottlenecks.
- For the system: The interrupt handler should execute as quickly as possible to minimize disruption to the interrupted code and maintain system performance.

Process Context and Interrupt Context

context refers to the state or environment in which a program, process, or part of the operating system (such as the kernel) operates.

Process Context
Definition: Kernel code that services system calls issued by user applications runs in the process context.
Preemptibility: Kernel code in this context is preemptible, meaning it can be interrupted to run other code, including interrupt handlers or higher-priority tasks.
Capabilities:

- Can access user-space memory.
- Can go to sleep or block, waiting for resources.
- Can acquire locks like mutexes.

Interrupt Context
Definition: Interrupt handlers execute in the interrupt context, triggered asynchronously by hardware events.
Non-preemptible: Code in interrupt context is not preemptible and must run to completion before the CPU can handle other tasks.
Restrictions:

- Cannot go to sleep or block.
- Cannot acquire mutexes (which could lead to deadlocks).
- Should not perform time-consuming tasks.
- Cannot directly access user-space virtual memory.

Why Interrupt Handlers Should be Quick

- Interrupt handlers must run quickly to prevent blocking other interrupts and to maintain system performance.

Issues with Long-running ISRs:

- Blocking other interrupts: While a high-priority ISR runs, other interrupts are blocked.
- Missed interrupts: **If the ISR for a particular type takes too long time**, subsequent interrupts of the type might be missed.

Top Halves and Bottom Halves

We use top halves and bottom halves to ensure quick response to interrupts by handling urgent tasks immediately in the top half, while deferring non-urgent processing to the bottom half to maintain system efficiency and responsiveness.

Top Halves

Definition: The top half is **the part of the interrupt handler that runs immediately when an interrupt is received.**

Purpose: It handles **time-critical tasks that must be done right away**, like acknowledging the interrupt or resetting the hardware.

Example: Imagine a network card that receives packets. When it gets a packet, it triggers an interrupt. The top half would quickly acknowledge this interrupt and prepare the card for more packets.

Reason: The top half must run quickly because it responds to hardware signals, and delaying could cause the hardware (like the network card) to miss new data or events.

Bottom Halves

Definition: The bottom half processes **less urgent tasks that can be deferred to a later time** when the system is less busy.

Purpose: It allows the **top half to handle new interrupts without delay**, **focusing only on non-time-critical tasks.**

Example: Continuing with the network card example, after the top half acknowledges the packet, the bottom half processes the packet data (like checking its destination or handling errors).
Reason: By splitting the work into top and bottom halves, the system remains responsive to new interrupts and can handle multiple tasks efficiently.

Bottom Half Mechanisms in Linux

The bottom halves mechanism helps in **“non-urgent tasks”** to be processed later, allowing the system to handle new interrupts efficiently and promptly.

Workqueue:
Description: Allows deferred work to run in the context of a kernel thread, enabling complex tasks that might involve sleeping or blocking operations.

Why Use It: It can perform more extensive processing that the top half cannot handle.

Example:A file system might use a workqueue to handle disk I/O operations initiated by an interrupt.

Threaded IRQs:
Description: Allows interrupt handlers to run as kernel threads, making them **preemptible and capable of blocking.**
Why Use It: Provides more flexibility, as **threaded IRQs can be prioritized or scheduled like regular threads.**
Example: A device driver that needs to perform heavy computation or access user-space data may use a threaded IRQ.

Softirq:
Description: A mechanism for handling high-priority tasks that don't need to run immediately but should run soon.
Why Use It: Balances between quick response and deferred execution, handling tasks like networking or block device processing.
Example: The networking stack uses softirqs to process incoming packets after they are copied into main memory by the top half.

Tasklets:

Description: Similar to **softirqs but designed for simpler tasks that don't require complex processing**.
Why Use It: Provides a lightweight mechanism for handling quick deferred tasks.
Example: A mouse driver might use a tasklet to update the cursor position on the screen after processing input data from an interrupt.

Key Takeaways:

- Top Half: Handles urgent, minimal tasks that cannot wait, ensuring the system remains responsive.
- Bottom Half: Defers non-urgent tasks to a more convenient time, preventing the system from missing new interrupts.
- Mechanisms: Different mechanisms (Workqueue, Threaded IRQs, Softirq, Tasklets) provide flexibility in handling deferred tasks based on their complexity and urgency.

Functions Related to Interrupt Handling

1. request_irq

Description: This function is **used to register an interrupt request (IRQ) line and associate it with an interrupt handler function.**

Syntax:

```
int request_irq(unsigned int irq, irq_handler_t handler, unsigned long flags, const
char *name, void *dev_id);
```

Parameters:

- **irq: IRQ number that needs to be allocated.**
- **handler:** The interrupt handler function **(irq_handler_t) to be invoked whenever the interrupt occurs.** It should return **IRQ_HANDLED when it successfully processes the interrupt**, and **IRQ_NONE if it fails.**
- **flags:** Flags that can modify the behavior of the interrupt. Important flags include:
  - **IRQF_DISABLED:** Disables all interrupts when the handler runs.
  - **IRQF_SAMPLE_RANDOM:** Uses the interrupt as a source of entropy for random number generation.
  - **IRQF_SHARED:** Allows multiple interrupt handlers to share the same IRQ line.
  - **IRQF_TIMER:** Specifies that the handler deals with system timer interrupts.
- **name:** The device name that uses this IRQ, visible in /proc/interrupts.
- **dev_id:** A unique identifier (device structure pointer) for the interrupt handler. Used for shared interrupt lines to differentiate between different handlers.

- **Return value:**
  - Returns 0 on success.
  - Returns non-zero if there's an error.

Important note: request_irq() cannot be called from an interrupt context (i.e., within another interrupt handler), as it may block, causing system issues.

2. free_irq

Description: This function **releases an interrupt handler that was previously registered with request_irq().The free_irq function removes the interrupt handler from the system and disables the IRQ line.**

Syntax:

```
void free_irq(unsigned int irq, void *dev_id);
```

Parameters:

- irq: The IRQ number to release.
- dev_id: The device identifier (same as used in request_irq()).

Behavior:

- If the interrupt line is not shared, the function removes the handler and disables the IRQ line.
- If the interrupt line is shared, the handler identified by dev_id is removed, but the IRQ line is disabled only when the last

handler is removed.

- Important note: free_irq() must bIf the interrupt line is not shared, the function removes the handler and disables the IRQ line.

3. enable_irq

Description: The enable_irq function ensures that the interrupt is enabled and can be serviced by the interrupt handler.

Syntax:

```
void enable_irq(unsigned int irq);
```

4. disable_irq

Description: The disable_irq function ensures that no more interrupts are handled for the specified IRQ, which helps in cleaning up and avoiding any issues when the handler is no longer needed.

Syntax:

```
void disable_irq(unsigned int irq);
```

5. disable_irq_nosync

Description: Disables an IRQ, but it ensures that the interrupt handler (if already running) is allowed to complete before the IRQ line is fully disabled.

Syntax:

```
void disable_irq_nosync(unsigned int irq);
```

6. in_irq

Description: Returns true if the current execution is inside an interrupt handler.

Syntax:

```
bool in_irq(void);
```

Interrupt Flags:

1.IRQF_DISABLED

Description: When set, all interrupts are disabled while the interrupt handler is executing.

Note: This flag is generally avoided for most interrupt handlers because disabling all interrupts can negatively impact the system's performance by increasing interrupt latency.

2. IRQF_SAMPLE_RANDOM

Description: When set, the timing of the interrupts generated by the device is added to the kernel entropy pool for random number generation.This flag should be used with devices that generate interrupts at non-deterministic times (like hardware random number generators or sensors).

3. IRQF_TIMER

Description: This flag specifies that the interrupt handler is responsible for handling interrupts from the system timer.

4. IRQF_SHARED

Description: Allows multiple interrupt handlers to share the same IRQ line.

Interrupt Handler Execution:
When an interrupt occurs, the interrupt handler is responsible for handling the interrupt and making sure the system responds quickly. The handler should not perform time-consuming tasks, as it must return as soon as possible to avoid blocking other interrupts.

Top Half:

What is the Top Half?

- The top half is the actual interrupt handler function that is executed immediately when an interrupt occurs.
- It handles the time-critical tasks that must be done quickly.
- The primary responsibility of the top half is to acknowledge the interrupt and prepare the hardware for the next interrupt (e.g., by clearing flags, resetting devices, etc.).

When to Use Top Half?

- The top half is sufficient when the interrupt handler only needs to do a small amount of work quickly.
- It is typically used when the tasks are hardware-specific and need to be handled immediately to ensure smooth operation.

Bottom Half:

What is the Bottom Half?

- The bottom half is used to **defer time-consuming tasks that don't need to be executed immediately after the interrupt.**
- **It allows the system to continue handling interrupts efficiently without blocking for too long.** The bottom half runs later in a more convenient time, usually after the interrupt handler has completed its critical work.

Why Do We Need the Bottom Half?

- If the interrupt handler performs a lot of work or needs to interact with complex data structures, performing these tasks in the top half would delay the system's response to new interrupts.
- So, we split the work between the top half and bottom half, allowing the top half to handle urgent tasks and the bottom half to do the heavier lifting later.

When to Use the Bottom Half?

- When the interrupt handler needs to perform time-consuming work (such as processing data, updating buffers, etc.).
- The bottom half is useful when interrupts can be serviced later without affecting the system's real-time responsiveness

How is the Bottom Half Executed?

- The **bottom half can be executed using mechanisms like softirqs, tasklets, or workqueues.**
- These **mechanisms defer the work and allow the kernel to process other interrupts in the meantime.**

Workqueue in Linux Kernel

A workqueue is a mechanism in the Linux kernel used to **defer work (i.e., tasks) that need to be done after handling an interrupt or event.Instead of processing everything right away in the interrupt handler** (which can block other interrupts), we can **defer the work to a kernel thread that will run in process context.**

Deferred work in the context of the Linux kernel refers **to tasks or operations that are delayed or postponed and not executed immediately when an interrupt or event occurs.**

Process Context: This means the deferred work runs like a regular task.

When we say that **workqueue** tasks run in **process context**, it means that the deferred work scheduled through workqueues is executed in a kernel thread that behaves similarly to a user-space process.

Why Use Workqueue?

Deferred Work: When an interrupt occurs, you may want to do some heavy work, but doing it immediately in the interrupt handler (top half) can delay the system's responsiveness. So, you "defer" the work to be done later by a kernel thread.

Process Context: The work that is deferred to a workqueue runs in process context. This is important because it allows the **work to perform longer operations like waiting (sleeping) or allocating resources,** which **cannot be done in an interrupt context.**

There are two main methods to use work queues in the kernel:

Global Workqueue (Static/Dynamic)

Global Workqueues are pre-existing work queues that the kernel already provides.We can simply submit work to these queues without having to create a custom one.

Dynamic Workqueue: This is another global workqueue, but it allows the kernel to create or modify work queues dynamically as needed.

Advantages of Workqueues:

- **Sleepable:** Can perform longer operations (e.g., sleeping).
- **Schedulable:** Work can be executed at a later time, allowing the kernel to handle other tasks in the meantime.
- **Flexibility:** You can create custom workqueues for more control over how work is deferred.

Static Workqueue

This is a pre-defined global workqueue that is always available, and you can submit work to it whenever needed.

1.Initializing Work Using the Static Method

To initialize work, we use the **DECLARE_WORK macro**, which creates a work_struct and associates it with a function to be executed.

Syntax:

```
DECLARE_WORK(name, void (*func)(struct work_struct *work));
```

Parameters

- name: The name of the work item.
- func: The function to be executed when the work is scheduled.

Example:

```c
#include <linux/workqueue.h>
#include <linux/module.h>
#include <linux/init.h>

void my_work_function(struct work_struct *work) {
    printk(KERN_INFO "Workqueue function executed\n");
}

// Declare and initialize a workqueue item named my_work
DECLARE_WORK(my_work, my_work_function);

static int __init my_module_init(void) {
    printk(KERN_INFO "Module loaded\n");
    // Schedule the work to be executed
    schedule_work(&my_work): //schedule the work it adds my_work to kernel
    return 0;
}

static void __exit my_module_exit(void) {
    printk(KERN_INFO "Module unloaded\n");
}

module_init(my_module_init);
module_exit(my_module_exit);
MODULE_LICENSE("GPL");
```

2. Scheduling Work to the Workqueue

Once you have initialized the work item we can schedule it using different functions depending on our needs.

2.1 schedule_work

Schedules a work item to the global workqueue for immediate execution.

Syntax:

```
int schedule_work(struct work_struct *work);
```

Example:

```
schedule_work(&my_workqueue);
```

- This adds my_workqueue to the global workqueue for execution as soon as possible.

2.2 schedule_delayed_work

Schedules a work item to be executed after a specified delay.

Syntax:

```
int schedule_delayed_work(struct delayed_work *dwork, unsigned long delay);
```

- dwork: The delayed work item.
- delay: The delay in jiffies before the work is executed.

Example:

```
DECLARE_DELAYED_WORK(my_delayed_workqueue, workqueue_fn);
schedule_delayed_work(&my_delayed_workqueue, 10); // Delay in jiffies
```

2.3 schedule_work_on

Schedules a work item to run on a specific CPU.

Syntax:

```
int schedule_work_on(int cpu, struct work_struct *work);
```

Example:

```
schedule_work_on(1, &my_workqueue); // Schedule on CPU 1
```

2.4 schedule_delayed_work_on

Schedules delayed work on a specific CPU.

Syntax:

```
int schedule_delayed_work_on(int cpu, struct delayed_work *dwork, unsigned long delay);
```

Example:

```
schedule_delayed_work_on(1, &my_delayed_workqueue, 10); // Delay on CPU 1
```

3 Deleting Work from Workqueue

We can remove or flush work from the workqueue using specific functions.

3.1 flush_work

Waits for a specific work item to complete.

Syntax:

```
int flush_work(struct work_struct *work);
```

Example:

```
flush_work(&my_workqueue);
```

3.2 flush_scheduled_work

Waits for all scheduled work items in the global workqueue to finish.

Syntax:

```
void flush_scheduled_work(void);
```

Example:

```
flush_scheduled_work();
```

4. Canceling Work from Workqueue

we may need to cancel work if it's no longer needed or if the module is being unloaded.

4.1 cancel_work_sync

Cancels a work item if it hasn't started. If it has, it waits for the work to complete.

Syntax:

```
int cancel_work_sync(struct work_struct *work);
```

Example:

```
cancel_work_sync(&my_workqueue);
```

4.2 cancel_delayed_work_sync

Cancels delayed work if it hasn't started, or waits for its completion.

Syntax:

```
int cancel_delayed_work_sync(struct delayed_work *dwork);
```

Example:

```
cancel_delayed_work_sync(&my_delayed_workqueue);
```

5. Checking the Workqueue

We can check whether work is pending in the queue using the following functions.

5.1 work_pending

Checks if a work item is pending.

Syntax:

```
bool work_pending(struct work_struct *work);
```

Example:

```
if (work_pending(&my_workqueue)) {
    printk(KERN_INFO "Work is pending\n");
}
```

6.2. delayed_work_pending

Checks if delayed work is pending.

Syntax:

```
bool delayed_work_pending(struct delayed_work *work);
```

Example:

```
if (delayed_work_pending(&my_delayed_workqueue)) {
    printk(KERN_INFO "Delayed work is pending\n");
}
```

Static Code:

In this example, we define a simple kernel module that uses a workqueue to defer the execution of a task. The workqueue task is defined by a function, workqueue_fn, which prints a message when executed. This function is associated with a work item, my_work, which is declared using the struct work_struct structure.

Upon loading the module, the my_module_init function is called. Inside this function, we initialize the work item with INIT_WORK and associate it with our workqueue function, workqueue_fn. We then schedule this work to be executed by the global workqueue using schedule_work. Immediately after scheduling, we check if the work is pending using the work_pending function, which returns a status indicating whether the work is queued but not yet executed.

The my_module_exit function handles the module cleanup when it is unloaded. Before the module exits, we attempt to cancel the work using cancel_work_sync. This function cancels the work if it hasn't been executed yet and waits for its completion if it is already running. We then check again if the work is still pending. Finally, a message is printed to indicate that the module is being unloaded.

This flow illustrates how to initialize, schedule, cancel, and check the status of a work item using workqueues in the Linux kernel. It provides a clear lifecycle of a workqueue task, from creation to cleanup, within a kernel module context.

```c
#include <linux/module.h>
#include <linux/init.h>
#include <linux/workqueue.h>
#include <linux/delay.h>

static struct work_struct my_work; // Declare a work_struct

// Workqueue function to be executed
void workqueue_fn(struct work_struct *work) {
    printk(KERN_INFO "Workqueue function executed\n");
}

// Initialize the module
static int __init my_module_init(void) {
    printk(KERN_INFO "Module loaded\n");

    // Initialize the workqueue item with the workqueue function
    INIT_WORK(&my_work, workqueue_fn);

    // Schedule the work to be executed by the global workqueue
    schedule_work(&my_work);

    // Check if the work is pending (should return false since we just scheduled it)
    if (work_pending(&my_work)) {
        printk(KERN_INFO "Work is pending\n");
    } else {
        printk(KERN_INFO "Work is not pending\n");
    }

    return 0;
}

// Exit function to cleanup the module
static void __exit my_module_exit(void) {
    printk(KERN_INFO "Module unloading\n");

    // Cancel the work if it hasn't been executed yet
    if (cancel_work_sync(&my_work)) {
        printk(KERN_INFO "Work was pending and now cancelled\n");
    } else {
        printk(KERN_INFO "Work was already completed or not pending\n");
    }

    // Check again if the work is still pending (should return false now)
    if (work_pending(&my_work)) {
        printk(KERN_INFO "Work is still pending\n");
    } else {
        printk(KERN_INFO "Work is not pending\n");
    }

    printk(KERN_INFO "Module unloaded\n");
}

module_init(my_module_init);
module_exit(my_module_exit);
MODULE_LICENSE("GPL");
```

Workqueue in Linux - Dynamic Method

In the dynamic method, workqueues are initialized and managed dynamically at runtime. This provides flexibility in creating and scheduling tasks in workqueues. The dynamic method primarily uses the INIT_WORK macro and other related functions.

1.Initialization with Dynamic Method

INIT_WORK

The INIT_WORK macro is used to initialize a work item dynamically. This macro sets up a work_struct with a specific function that will be executed when the work item is processed.

Syntax:

```
INIT_WORK(struct work_struct *work, void (*work_fn)(struct work_struct *));
```

- work: The work item to be initialized, typically a work_struct structure.
- work_fn: The function to be executed when the work item is scheduled.

Example:

```
struct work_struct my_work;

void work_fn(struct work_struct *work) {
    printk(KERN_INFO "Work function executed\n");
}

INIT_WORK(&my_work, work_fn);
```

2. Scheduling Work

Once initialized, work items can be scheduled using various functions, depending on when and where you want the work to be executed.

2.1 schedule_work

This function schedules a work item to be executed in the kernel-global workqueue.

Syntax:

```
int schedule_work(struct work_struct *work);
```

- work: The work item to be scheduled.

Example:

```
schedule_work(&my_work);
```

2.2 schedule_delayed_work

This function schedules a work item to be executed after a specified delay.

Syntax:

```
int schedule_delayed_work(struct delayed_work *dwork, unsigned long delay);
```

- dwork: The delayed work item to be scheduled.
- delay: The number of jiffies to wait before executing the work.

Example:

```
struct delayed_work my_delayed_work;
INIT_DELAYED_WORK(&my_delayed_work, work_fn);
schedule_delayed_work(&my_delayed_work, 100); // Delays for 100 jiffies
```

2.2 schedule_work_on

This function schedules a work item to be executed on a specific CPU.

Syntax:

```
int schedule_work_on(int cpu, struct work_struct *work);
```

- cpu: The CPU on which to run the work.
- work: The work item to be scheduled.

Example:

```
schedule_work_on(1, &my_work); // Schedule on CPU 1
```

2.3 schedule_delayed_work_on

Similar to schedule_delayed_work, but allows specifying the CPU on which the work should be executed after a delay.

Syntax:

```
int schedule_delayed_work_on(int cpu, struct delayed_work *dwork, unsigned long delay);
```

Example:

```
schedule_delayed_work_on(1, &my_delayed_work, 100); // Delays for 100 jiffies on CPU 1
```

3 Deleting and Canceling Work

3.1 flush_work

This function blocks until the specified work item has finished executing.

Syntax:

```
int flush_work(struct work_struct *work);
```

Example:

```
flush_work(&my_work);
```

3.2 flush_scheduled_work

Flushes all work items in the global workqueue.

Syntax:

```
void flush_scheduled_work(void);
```

3.3 cancel_work_sync

This function cancels a work item if it is not currently executing, or waits for it to finish if it is already running.

Syntax:

```
int cancel_work_sync(struct work_struct *work);
```

Example:

```
cancel_work_sync(&my_work);
```

3.4 cancel_delayed_work_sync

Cancels a delayed work item in a similar fashion.

Syntax:

```
int cancel_delayed_work_sync(struct delayed_work *dwork);
```

Example:

```
cancel_delayed_work_sync(&my_delayed_work);
```

4. Checking Work Status

4.1 work_pending

Checks if a work item is pending.

Syntax:

```
int work_pending(const struct work_struct *work);
```

Example:

```
if (work_pending(&my_work)) {
    printk(KERN_INFO "Work is pending\n");
}
```

4.2 delayed_work_pending

Checks if a delayed work item is pending.

Syntax:

```
int delayed_work_pending(const struct delayed_work *dwork);
```

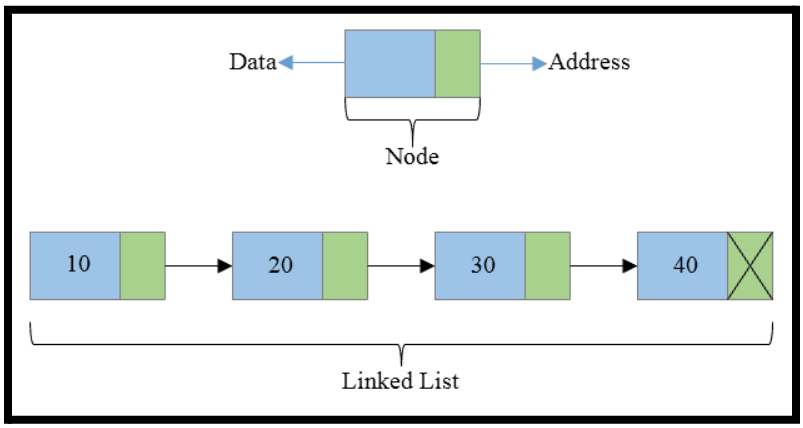
Example:

```
if (delayed_work_pending(&my_delayed_work)) {
    printk(KERN_INFO "Delayed work is pending\n");
}
```

Linked List in Linux Kernel

Introduction to Linked List

A linked list is a fundamental data structure comprising a sequence of nodes, where each node consists of two main components: the data field (which stores the actual data) and the reference field (a pointer pointing to the next node in the sequence).



Each node in a linked list is termed an element and the list is tracked using a head pointer that always points to the first element. the elements in a linked list do not need to occupy contiguous memory locations, and each node connects to the next through pointers, forming a chain-like structure

Advantages of Linked Lists

- Dynamic Memory Allocation: Linked lists can grow or shrink in size as needed, unlike arrays that have a fixed size. This means you can add or remove items from a linked list without worrying about running out of space or wasting memory.
- Efficient Insertions and Deletions: Adding or removing items from a linked list is easier compared to arrays. In arrays, if you want to insert or delete an item in the middle, you have to shift all the other items around, which can be slow. In linked lists, you simply change a few pointers, which is much quicker.
- Building Other Structures: Linked lists are like building blocks for more complicated data structures such as stacks and queues.
- Faster for Some Tasks: In some cases, linked lists can be faster than arrays because you can directly add or remove items without shifting others, which saves time.

Disadvantages of Linked Lists

- **Extra Memory Usage**: Each node requires additional memory for pointers, leading to potential memory overhead.
- **Sequential Access**: Elements must be accessed sequentially, unlike arrays that allow random access.
- **Difficult Reverse Traversal**: Traversing backward through the list is complicated, especially in singly linked lists.

Applications of Linked Lists

- Used in the implementation of stacks, queues, and graph representations.
- No need to define the size in advance, making them more flexible compared to arrays.

Types of Linked Lists

While there are multiple types, the primary categories are:

- **Singly Linked List:** Each node points to the next node in the sequence.
- **Doubly Linked List:** Each node has two pointers, one pointing to the next node and another pointing to the previous node, allowing bidirectional traversal.
- **Circular Linked List:** The last node points back to the first node, forming a circle.

Introduction to Linked Lists in the Linux Kernel

In the linux kernel. A linked list is a crucial data structure that allows efficient storage and manipulation of data. The kernel provides a built-in doubly linked list implementation, meaning each node has pointers to both the previous and next nodes.

Important note:This built-in linked list eliminates the need for custom implementations or third-party libraries.

1 Declaring a Linked List

In the Linux kernel, we use the struct list_head for the linked list pointers:

```
struct my_list {
    struct list_head list;
    int data;
};
```

Here, struct list_head contains two pointers (next and prev), and it's declared in the header file list.h.

2. Initializing the Linked List Head

Before adding any nodes, the list's head (the starting point) must be created. The macro LIST_HEAD(name) initializes the head of the list:

```
LIST_HEAD(etx_linked_list);
```

This macro creates a struct list_head and sets its next and prev pointers to point to itself, indicating an empty list

3. Creating and Initializing a Node

initialize this list node using the INIT_LIST_HEAD() function:
struct my_list new_node;

```
INIT_LIST_HEAD(&new_node.list);
new_node.data = 10;
```

4. Adding Nodes to the Linked List

4.1 Add After the Head: Use list_add() to add a node after the head, useful for stack-like behavior (LIFO).
Syntax:

```
list_add(&new_node.list, &etx_linked_list);
```

4.2 Add Before the Head: Use list_add_tail() to add a node before the head, useful for queue-like behavior (FIFO).
Syntax:

```
list_add_tail(&new_node.list, &etx_linked_list);
```

5 Deleting Nodes from the Linked List
5.1 Delete a Node: Use list_del() to remove a node from the list. This function disconnects the node but doesn't free its memory.

```
list_del(&new_node.list);
```

5.2 Delete and Reinitialize a Node: Use list_del_init() to delete a node and reset its pointers.

```
list_del_init(&new_node.list);
```

6. Replacing Nodes
6.1 Replace a Node: Use list_replace() to swap an old node with a new one.

```
list_replace(&old_node.list, &new_node.list);
```

6.2 Replace and Reinitialize: Use list_replace_init() to replace a node and reinitialize the old one.

```
list_replace_init(&old_node.list, &new_node.list);
```

7. Rotating the Linked List
Use list_rotate_left() to rotate the list, moving the first node to the last position.

```
list_rotate_left(&etx_linked_list);
```

8. Checking Linked List Properties
8.1 Is the List Empty?: Use list_empty() to check if the list has no nodes.

```
if (list_empty(&etx_linked_list)) {
    // List is empty
}
```

8.2 Is a Node the Last in the List?: Use list_is_last() to check if a node is the last one.

```
if (list_is_last(&node.list, &etx_linked_list)) {
    // Node is the last one
}
```

9. Splitting and Joining Lists

9.1 Splitting: Use list_cut_position() to divide a list into two parts.

```
list_cut_position(&new_list, &etx_linked_list, &node.list);
```

9.2 Joining: Use list_splice() to combine two lists.

```
list_splice(&new_list, &etx_linked_list);
```

10 Traversing the Linked List

10.1 Basic Traversal: Use list_for_each() to iterate through the list.

```
struct list_head *pos;
list_for_each(pos, &etx_linked_list) {
    // Use list_entry() to get the containing struct
}
```

10.2 Safe Traversal: Use list_for_each_entry_safe() for safe traversal, especially when removing nodes.

```
struct my_list *pos, *n;
list_for_each_entry_safe(pos, n, &etx_linked_list, list) {
    // Process each node
}
```

10.3 Reverse Traversal: Use list_for_each_prev() to iterate in reverse order.

```
list_for_each_prev(pos, &etx_linked_list) {
    // Process each node
}
```

Kernel Thread – Linux Device Driver

Process vs. Thread in Linux Kernel

Process

- Definition: An executing instance of a program.
- Terminology: Some operating systems use the term ‘task’ to refer to a program that is being executed.

Characteristics:

- Heavyweight Process: Consumes more resources.
- Context Switch: Switching between processes is time-consuming due to the need to save and load context (CPU registers, program counter, etc.).

Threads

- Definition: An independent flow of control within the same address space as other threads in the same process.

Characteristics:

- Lightweight: Requires fewer system resources compared to processes.
- Shared Address Space: Threads share data easily and synchronize more efficiently than processes.
- Concurrency Issues: Shared resources can lead to concurrency problems, requiring synchronization mechanisms.

Kernel Threads and Processes

Multiple Threads in a Process: One process can have multiple threads running concurrently, each executing different parts of the code.

Advantages:

- Easier communication between threads due to shared address space.
- Faster creation and context switching compared to processes.

Disadvantages:

- Requires synchronization to handle concurrency issues.

- **User-Level Threads**:
  - Managed by the user-level thread library.
  - The kernel is unaware of these threads.
  - **Library Functions**: **Create, destroy, schedule threads, and handle** context switches in user space.
- **Kernel-Level Threads**:
  - Managed by the OS kernel.
  - Thread operations are implemented in the kernel code.
  - No thread management code in the user application.

Thread Management

Thread Management: Managing **threads involves creating, destroying, scheduling, and handling** their execution state (program counter, CPU registers).

Execution State: Consists of the program counter (next instruction to execute) and CPU registers (hold execution arguments).

Create Kernel Thread

1.Function: kthread_create

Purpose: Creates a new kernel thread.

syntax:

```
struct task_struct * kthread_create(int (*threadfn)(void *data), void *data, const char namefmt[], ...);
```

Parameters:

- **threadfn:** Pointer to the function that the thread will execute.
- **data:** Argument passed to threadfn.
- **namefmt:** Format string for naming the thread.

Returns: A pointer to the task_struct of the new thread or an error pointer on failure.

Start Kernel Thread

1.Function: wake_up_process( )

Wakes up a process or thread, moving it to the runnable state.

Prototype:

```
int wake_up_process(struct task_struct * p);
```

Parameters:

- p: Process to be woken up.

Returns: 1 if the process was woken up, 0 if it was already running.

Stop Kernel Thread

1.1Function: kthread_stop

Purpose: Stops a kernel thread previously created by kthread_create.

Prototype:

```
int kthread_stop(struct task_struct *k);
```

Parameters:

- k: Thread created by kthread_create

Returns: Result of threadfn or -EINTR if wake_up_process was never called.

1.2 Function: kthread_should_stop

Purpose: Checks if the thread should stop execution.

Prototype:

```
int kthread_should_stop(void);
```

Returns: 1 if the thread should stop, otherwise 0.

1.3 Function: kthread_bind

Purpose: Binds a kernel thread to a specific CPU.

Prototype:

```
void kthread_bind(struct task_struct *k, unsigned int cpu);
```

Parameters:

- k: The task_struct of the thread to bind.
- cpu: The CPU to bind the thread to.

Example:

```
kthread_bind(my_thread, 0);
```

Creating and Starting Kernel Thread

```
Creating:
static struct task_struct *etx_thread;
etx_thread = kthread_create(thread_function, NULL, "eTx Thread");
if (etx_thread)
{
    wake_up_process(etx_thread);
}
else
{
    printk(KERN_ERR "Cannot create kthread\n");
}
```

Tasklets in the Linux Kernel

Tasklets in the Linux kernel are part of the bottom-half mechanism, designed to handle deferred work from interrupts. When an interrupt occurs, the immediate work is done in the top-half, and the less urgent work is deferred to be handled later by the bottom-half, where tasklets come into play.

What Are Tasklets?

- Tasklets are a way to schedule small tasks that can be executed later.
- They are like mini-threads without their own stack or full context.
- Tasklets are used to handle work that doesn't need to be done immediately but should happen soon.

Key Characteristics

Single CPU Execution:

- A tasklet **runs only on the CPU that scheduled it.** This **helps improve cache performance because the tasklet doesn't have to load its data from another CPU's cache.**
- A tasklet **cannot be executed on multiple CPUs at the same time.**

Parallel Execution:

- **Different tasklets can run in parallel on different CPUs**.
- However, the same tasklet will not run more than once at the same time on any CPU.

Non-preemptive Scheduling:

- Tasklets are **executed one after another in the order they are scheduled.**
- There are **two priorities for scheduling tasklets: normal and high.**

Atomic Nature:

- **Tasklets run in an atomic context,** meaning **they cannot be interrupted by other tasklets.** Because of this, tasklets **can't use functions that sleep or block**, like sleep(), or use **synchronization primitives like mutexes or semaphores.**
- We can use spinlocks if you need to protect data that might be accessed by other parts of the kernel.

Points to Remember

1. **Atomic Context: No sleeping or waiting.** Use spinlocks if necessary.
2. **Single CPU:** A **one tasklet runs on the CPU that scheduled it, not on multiple CPUs.**
3. Concurrency Control: **While different tasklets can run on different CPUs,** a single tasklet won’t run concurrently on multiple CPUs.
4. **Priorities:** Tasklets can be scheduled with normal or high priority.

Creation of Tasklets

Tasklets can be created in two ways:

- 1. Static Method: Static tasklets are created at compile-time using predefined macros, such as DECLARE_TASKLET or DECLARE_TASKLET_DISABLED
- 2. Dynamic Method **Definition**: Dynamic tasklets are created at runtime using functions like **tasklet_init.**

Static (Tasklet Method)

The **static method** of creating a tasklet in the Linux kernel involves defining the tasklet at compile time using macros like **DECLARE_TASKLET or DECLARE_TASKLET_DISABLED**. This approach initializes the tasklet structure with specific function pointers and data, setting it up as either enabled or disabled.

Structure of tasklet:

The tasklet_struct is the core data structure used to define a tasklet.

```
struct tasklet_struct {
    struct tasklet_struct *next;  // Next tasklet in line for scheduling
    unsigned long state;          // Tasklet state: TASKLET_STATE_SCHED or
TASKLET_STATE_RUN
    atomic_t count;               // Nonzero if disabled, 0 if enabled
    void (*func)(unsigned long); // Pointer to the function to execute
    unsigned long data;           // Data to pass to the function
};
```

Parameters:

- next: Points to the next tasklet in the queue.
- state: Indicates the tasklet's state, either scheduled or running.
- count: Holds the value indicating if the tasklet is enabled (0) or disabled (nonzero).
- func: The main function that the tasklet will execute.
- data: Data passed to the function func.

1 Creating Tasklets

1.1 DECLARE_TASKLET

This macro is used to create a tasklet and initialize its parameters. The tasklet is in the enabled state by default.

```
Function Prototype:
DECLARE_TASKLET(name, func, data);
```

Parameters:

- name: The name of the tasklet structure.

- func: Pointer to the function that will be executed.
- data: Data to pass to the function.

Example

```
DECLARE_TASKLET(tasklet, tasklet_fn, 1);
```

This creates a tasklet structure with the name tasklet and assigns the parameters. The structure will look like:

```
struct tasklet_struct tasklet = { NULL, 0, 0, tasklet_fn, 1 };
```

1.2 DECLARE_TASKLET_DISABLED

This macro creates a tasklet in a disabled state. It must be enabled using tasklet_enable before it can run.

```
DECLARE_TASKLET_DISABLED(name, func, data);
```

Parameters:

- name: The name of the tasklet structure.
- func: Pointer to the function that will be executed.
- data: Data to pass to the function.

2. Enabling and Disabling Tasklets

2.1 tasklet_enable

- Enabling a tasklet means making it eligible to be scheduled and executed. When a tasklet is enabled, it can be placed in the queue and run by the CPU.
- This function enables a previously disabled tasklet.

```
void tasklet_enable(struct tasklet_struct *t);
```

2.2 tasklet_disable

- Disabling a tasklet means preventing it from being scheduled and executed. A disabled tasklet won't run even if it is placed in the queue until it is explicitly enabled again.
- This function disables a tasklet and waits for its current operation to complete.

```
void tasklet_disable(struct tasklet_struct *t);
```

2.2 tasklet_disable_nosync

This function disables a tasklet immediately without waiting for its current operation to complete.

```
void tasklet_disable_nosync(struct tasklet_struct *t);
```

Note: If a tasklet is disabled, it can still be added to the queue but will not run until enabled. The count field tracks the number of times a tasklet is disabled and must be enabled the same number of times.

3 Scheduling Tasklets

When a tasklet is scheduled, it is placed in one of two queues based on its priority. Each CPU has its own queue.

3.1 tasklet_schedule

This function schedules a tasklet with normal priority.

```
void tasklet_schedule(struct tasklet_struct *t);
```

Example

```
/* Scheduling Task to Tasklet */
tasklet_schedule(&tasklet);
```

3.2 tasklet_hi_schedule

This function schedules a tasklet with high priority.

```
void tasklet_hi_schedule(struct tasklet_struct *t);
```

3.3 tasklet_hi_schedule_first

This function schedules a tasklet with high priority without affecting other tasklets.

```
void tasklet_hi_schedule_first(struct tasklet_struct *t);
```

4 Killing Tasklets

4.1 tasklet_kill

This function waits for a tasklet to complete and then deletes it.

```
void tasklet_kill(struct tasklet_struct *t);
```

Example

```
/* Kill the Tasklet */
tasklet_kill(&tasklet);
```

4.2 tasklet_kill_immediate

This function is used to delete a tasklet immediately when a CPU is in a dead state.

```
void tasklet_kill_immediate(struct tasklet_struct *t, unsigned int
cpu);
```

- t: Pointer to the tasklet structure.
- cpu: CPU number.

Dynamically Creating a Tasklet

1. tasklet_init Function

The tasklet_init function is used to initialize a tasklet dynamically.

```
void tasklet_init(struct tasklet_struct *t, void (*func)(unsigned
long), unsigned long data);
```

Parameters:

- t: Pointer to the tasklet_struct that will be initialized.
- func: Pointer to the function that will be executed by the tasklet.
- data: Data to pass to the function func.

Example

```
/* Tasklet by Dynamic Method */
struct tasklet_struct *tasklet
/* Init the tasklet by Dynamic Method */
tasklet = kmalloc(sizeof(struct tasklet_struct), GFP_KERNEL);
if (tasklet == NULL) {
    printk(KERN_INFO "etx_device: cannot allocate Memory");
}
tasklet_init(tasklet, tasklet_fn, 0);
```

In this example:

- Memory is allocated dynamically for the tasklet_struct using kmalloc.
- The tasklet_init function initializes the tasklet with the specified function tasklet_fn and the data value 0.

When tasklet_init is called, the function and data are assigned to the tasklet structure, and the tasklet's state is set to scheduled (TASKLET_STATE_SCHED), and its count is initialized to 0, indicating it is enabled.

```
tasklet->func = tasklet_fn;             // Function to execute
tasklet->data = 0;                      // Data argument
tasklet->state = TASKLET_STATE_SCHED;   // Tasklet state is scheduled
atomic_set(&tasklet->count, 0);         // Tasklet enabled
```

General Difference Between Static and Dynamic Tasklets

- Static Tasklets: **Declared and initialized at compile-time using macros like DECLARE_TASKLET. They are always available throughout the module's lifetime.**
- Dynamic Tasklets: **Created at runtime using tasklet_init, allowing for flexible creation and destruction as needed.** They are suitable for **situations where tasklets are not required for the entire lifetime of the module or need to be created conditionally.**

MUTEX

Introduction
Mutex (short for mutual exclusion) is a synchronization primitive used to prevent race conditions by ensuring that only one thread can access a shared resource at a time.

Analogy
- A car designed for one person can cause an explosion if multiple people enter simultaneously.
- The key to the car allows only one person to enter at a time, similar to how a mutex allows only one thread to access a resource at a time.

Example Problems (Race Conditions)
- SPI Connection: Multiple threads trying to read/write simultaneously.
- LED Display: Multiple threads writing different data at different positions simultaneously.
- Linked List: One thread inserting while another is deleting.

race conditions occur when multiple threads access shared resources concurrently, leading to unpredictable behavior or crashes.

Mutex
A mutex ensures mutual exclusion, allowing only one thread to access a resource at a time.
The thread that locks a mutex must also unlock it.

Mutex in Linux Kernel
Multitasking can lead to concurrency issues like race conditions.
struct mutex is used in the kernel to implement mutexes, ensuring safe access to shared resources.

Initializing Mutex
We can initialize Mutex in two ways:

Static Method
- A static mutex is a mutex that is declared and initialized at compile time. It is part of the static or global memory of the kernel module.
- It exists for the entire lifetime of the module, meaning it is always present and does not need to be allocated or deallocated explicitly.

Dynamic Method
- A dynamic mutex is a mutex that is allocated and initialized at runtime. It is typically a part of a dynamically allocated structure or used in cases where the mutex's lifetime is tied to the resource it protects.
- Dynamic mutexes are useful when you have multiple instances of a resource that require synchronization, or when the resource's lifetime is limited and determined at runtime.

1. Static Method

Static Method:
DEFINE_MUTEX(name): Used for global mutexes.
Example:
DEFINE_MUTEX(my_mutex);

2. Dynamic Method:

mutex_init(struct mutex *lock): Used for per-object mutexes.
Example:
struct mutex my_mutex;
mutex_init(&my_mutex);

Mutex Locking

Prototype:1. mutex_lock

Purpose: This function locks the mutex for the current thread.

Behavior:

- If the mutex is already locked by another thread, the calling thread will block (sleep) until the mutex becomes available.
- Once the mutex is available, it gets locked by the current thread.
- Usage: This is used in situations where the thread needs exclusive access to a resource and is willing to wait until it gets the lock.

Prototype:

```
void mutex_lock(struct mutex *lock);
```

Example:

```
struct mutex my_mutex;
mutex_lock(&my_mutex);
// Critical section code here
mutex_unlock(&my_mutex);
```

2. mutex_lock_interruptible

Purpose: Similar to mutex_lock, but the function can be interrupted by signals.
Behavior:

- If a signal is received while the thread is waiting for the mutex, the function returns with an error (-EINTR).
- This is useful in cases where the thread should not block indefinitely and needs to handle signals.

Return Value:

- 0 if the mutex was successfully locked.
- -EINTR if interrupted by a signal.

Prototype:

```
int mutex_lock_interruptible(struct mutex *lock);
```

Example:

```
if (mutex_lock_interruptible(&my_mutex)) {
    // Handle signal interruption
} else {
    // Critical section code here
    mutex_unlock(&my_mutex);
}
```

3. mutex_trylock

Purpose: Attempts to lock the mutex without waiting.
Behavior:

- If the mutex is already locked, this function returns immediately with a failure.
- If the mutex is not locked, it locks the mutex and returns success.
- This is useful for non-blocking scenarios where you want to try to acquire the lock but don't want to wait if it's not available.

Return Value:

- 1 if the mutex was successfully locked.
- 0 if the mutex was already locked by another thread.

Prototype:

```
int mutex_trylock(struct mutex *lock);
```

```
Example:
if (mutex_trylock(&my_mutex)) {
    // Critical section code here
    mutex_unlock(&my_mutex);
} else {   // Mutex was already locked, handle this case
```

Mutex Unlocking Function

1. mutex_unlock

Purpose: Unlocks the mutex that was previously locked by the current thread.

Behavior:

- This allows other threads waiting for the mutex to acquire it.
- It is an error to unlock a mutex that is not locked or to unlock a mutex locked by another thread.

Prototype:

void mutex_unlock(struct mutex *lock);

Example:

```
mutex_lock(&my_mutex);
// Critical section code here
mutex_unlock(&my_mutex);
```

Checking Mutex Status Function

1 mutex_is_locked

Purpose: Checks whether the mutex is currently locked or not.

Behavior:

- Returns 1 if the mutex is locked.
- Returns 0 if the mutex is not locked.

This function is typically used for debugging or logging purposes rather than for controlling program flow.

Prototype:

int mutex_is_locked(struct mutex *lock);

Example:

```
if (mutex_is_locked(&my_mutex)) {
    printk(KERN_INFO "Mutex is locked");
} else {
    printk(KERN_INFO "Mutex is not locked");
}
```

What is Spinlock?

- Spinlock is a type of lock used to protect shared data in a multi-threaded environment., if a thread cannot acquire the spinlock, it will "spin" (keep trying in a loop) until the lock is available.
- **Two States: Locked or Unlocked.**
- Used where the waiting time is expected to be short, avoiding the overhead of sleep and wake-up mechanisms.

Spinlock in Linux Kernel Device Driver

- Uniprocessor systems: Spinlocks are usually optimized away since there's no contention.
- Multiprocessor systems (CONFIG_SMP): Spinlocks are fully functional to manage concurrency.
- Preemption Enabled (CONFIG_PREEMPT): Spinlocks disable preemption to avoid race conditions.

Initialization Methods

1.Static Method

Use DEFINE_SPINLOCK(name); to create and initialize.
Example:
DEFINE_SPINLOCK(etx_spinlock); // Spinlock named etx_spinlock initialized to UNLOCKED.

2.Dynamic Method

Use spin_lock_init(spinlock_t *lock); to initialize.
Example:
spinlock_t etx_spinlock;
spin_lock_init(&etx_spinlock); // Dynamically initializing etx_spinlock.

Usage Approaches
1. Locking in User Context (Kernel Threads)

```
Lock: spin_lock(spinlock_t *lock);
Try Lock: spin_trylock(spinlock_t *lock); (Non-blocking).
Unlock: spin_unlock(spinlock_t *lock);
Check Lock: spin_is_locked(spinlock_t *lock);
```

Example:

```
spin_lock(&etx_spinlock);
// Critical section
spin_unlock(&etx_spinlock);
```

2. Locking Between Bottom Halves

Use the **same methods as in user context.**

3. **Locking Between User Context and Bottom Halves**

```
Lock: spin_lock_bh(spinlock_t *lock); (Disables soft interrupts).

Unlock: spin_unlock_bh(spinlock_t *lock);
```

Example:

```
spin_lock_bh(&etx_spinlock);
// Critical section
spin_unlock_bh(&etx_spinlock);
```

4. Locking Between Hard IRQ and Bottom Halves

```
Lock: spin_lock_irq(spinlock_t *lock); (Disables interrupts).
Unlock: spin_unlock_irq(spinlock_t *lock);
```

Example:

```
spin_lock_irq(&etx_spinlock);
// Critical section
spin_unlock_irq(&etx_spinlock);
```

5. Alternative to Approach 4

```
Lock: spin_lock_irqsave(spinlock_t *lock, unsigned long flags); (Saves
interrupt state).
Unlock: spin_unlock_irqrestore(spinlock_t *lock, unsigned long flags);
(Restores interrupt state).
```

Example:

```
unsigned long flags;
spin_lock_irqsave(&etx_spinlock, flags);
// Critical section
spin_unlock_irqrestore(&etx_spinlock, flags);
```

6. Locking Between Hard IRQs

Use the spin_lock_irqsave and spin_unlock_irqrestore methods as in Approach 5 for safe handling across different IRQs.

Difference between Mutex and Spinlock:

1. Definition

- Mutex: A mutex (mutual exclusion) is a synchronization primitive used to protect shared data by allowing only one thread to access the critical section at a time. If the mutex is already locked, other threads will sleep (or be blocked) until the lock is available.
- Spinlock: A spinlock is a lock mechanism that causes a thread trying to acquire it to spin (loop continuously) while waiting for the lock to become available, without putting the thread to sleep.

2. Behavior When Locked

- Mutex: When a thread tries to lock a mutex that is already locked, it is put to sleep (blocked) until the mutex is available.
- Spinlock: When a thread tries to lock a spinlock that is already locked, it spins (keeps checking repeatedly) until the lock becomes available.

3. Use Cases

- Mutex: Suitable for scenarios where the lock might be held for a longer period, as it avoids CPU busy-waiting by putting threads to sleep.
- Spinlock: Suitable for short critical sections where the lock will be held for a very short time and the overhead of sleeping and waking threads is higher than the cost of spinning.

4. CPU Usage

- Mutex: More CPU-efficient because threads are put to sleep while waiting.
- Spinlock: Consumes CPU cycles as the waiting thread keeps spinning until the lock is released.

5. Preemption and Interrupts

- Mutex: Can be used in contexts where preemption or blocking is allowed (such as in user-space or kernel-space threads).
- Spinlock: Generally used in kernel-space where blocking is not an option, such as in interrupt handlers or critical kernel sections. However, spinlocks must not be held in preemptible contexts without disabling preemption.

6. Overhead

- Mutex: Has higher context switch overhead because putting a thread to sleep and waking it up involves switching the context to another thread or process.
- Spinlock: Has minimal overhead but can be inefficient if the lock is held for a long time due to the busy-waiting.

7. Nested Locks

- Mutex: Supports recursive locking (same thread can lock the mutex multiple times).
- Spinlock: Does not support recursive locking; attempting to acquire a spinlock that the current thread already holds can lead to deadlock.

8. Interrupt Context

- Mutex: Cannot be used in interrupt context because it may sleep, which is not allowed in interrupt handlers.
- Spinlock: Can be used in interrupt context, but care must be taken to disable interrupts on the local CPU to avoid deadlocks (using spin_lock_irq or spin_lock_irqsave).

9. Example Scenarios

- Mutex: Used in user-space applications or kernel threads that can afford to sleep while waiting for the resource.
- Spinlock: Used in kernel drivers, interrupt handlers, or short critical sections in kernel code where sleeping is not possible.

Read-Write Spinlocks in Linux Kernel

Scenario:

- We have five threads sharing access to a single global variable.
- Thread 1 is responsible for writing data to the variable.
- Threads 2-5 are responsible for reading data from the variable.

Problem with Using a Spinlock:

If we use a simple spinlock:

- When multiple reader threads (Threads 2-5) attempt to read the variable concurrently, they will compete for the single lock.
- Even though the data is not being modified (no writer thread is active), only one reader thread can acquire the lock at a time.
- This leads to unnecessary waiting and wasted CPU cycles for the other reader threads, significantly impacting performance.

Benefits of Using a Read-Write Spinlock:

A read-write spinlock allows multiple reader threads to acquire the lock concurrently.

When only reader threads are active:

- Each reader thread can acquire a read lock independently.
- This enables simultaneous reading operations, significantly improving performance compared to a single-threaded spinlock.
- Writer threads still require exclusive access to the variable.

Feature	Spinlock	Read-Write Spinlock
Concurrency	Only one thread can hold the lock at a time.	Multiple readers can hold the lock concurrently.
Reader/Writer Priority	No distinction between read and write access.	Prioritizes readers in most implementations.
Performance	Can be inefficient for read-heavy workloads due to	Generally more efficient for read-heavy workloads due to increased concurrency.

Core working principles of a Read-Write Spinlock

Initial State: When **no thread is accessing the shared resource** (within the "critical section"), **both reader and writer threads can acquire the respective locks** (read lock or write lock) and **enter the critical section.** However, **only one thread can be within the critical section at any given time, ensuring data consistency.**

Reader Threads:

- **If one reader thread is already in the critical section**, other **reader threads can enter .**
- However, **a writer thread must wait until all existing reader threads** have exited the critical section before it can acquire the write lock and enter.

Reader Priority: The described behavior generally prioritizes reader threads. Once a reader thread has entered the critical section, **subsequent reader threads can enter without being blocked by the writer thread.**

Writer Thread:

- **If a writer thread is in the critical section, neither reader nor writer threads can enter.** The writer has exclusive access.

Writer Priority (Seqlock): In Linux, the **seqlock mechanism is designed to prioritize writer threads over reader threads.**

Initialization Methods

1. Static Method:

- Uses the DEFINE_RWLOCK(etx_rwlock); macro.
- This macro directly defines and initializes an rwlock_t variable named etx_rwlock during compilation.
- Suitable when the spinlock's lifetime is the same as the module itself.

2. Dynamic Method:

- Declares an rwlock_t variable: rwlock_t etx_rwlock;
- Initializes the variable using rwlock_init(&etx_rwlock); at runtime.
- Offers more flexibility as initialization can be delayed or conditional.

Choosing a Method:

- Use the static method for simplicity if the spinlock needs to exist throughout the module's lifetime.
- Use the dynamic method if you need more control over when the spinlock is initialized (e.g., during module initialization or based on specific conditions).

Key Points:

- rwlock_t: This is the data structure used to represent a Read-Write Spinlock in the Linux kernel.
- DEFINE_RWLOCK(): This macro simplifies the static initialization process.
- rwlock_init(): This function is used for dynamic initialization of a Read-Write Spinlock.

Example:

```c
#include <linux/rwlock.h>

// Static initialization
DEFINE_RWLOCK(my_rwlock);

// Dynamic initialization
rwlock_t my_dynamic_rwlock;

static int __init my_module_init(void)
{
    rwlock_init(&my_dynamic_rwlock);
    // ... rest of your module initialization
}

static void __exit my_module_exit(void)
{
    // ... module cleanup
}

module_init(my_module_init);
module_exit(my_module_exit);
```

1. Locking Between User Contexts

Scenario: This is the most basic scenario where you're dealing with kernel threads or processes running in user space.

Mechanism:

- **read_lock():** Acquires a read lock. If another thread already holds the write lock, it will spin (busy-wait) until the write lock is released.
- **read_unlock():** Releases the read lock.
- **write_lock():** Acquires a write lock. If any thread (reader or writer) holds the lock, it will spin.
- **write_unlock():** Releases the write lock.

Usage: Suitable when you need to protect shared data accessed by multiple kernel threads or processes within the user context. No special interrupt handling is required.

Example:

```c
#include <linux/kernel.h>
#include <linux/module.h>
#include <linux/rwlock.h>
#include <linux/kthread.h>
#include <linux/delay.h>

static rwlock_t my_rwlock;
static int etx_global_variable = 0;

static int thread_function1(void *pv)
{
    while (!kthread_should_stop()) {
        write_lock(&my_rwlock);
        etx_global_variable++;
        write_unlock(&my_rwlock);
        msleep(1000);
    }
    return 0;
}

static int thread_function2(void *pv)
{
    while (!kthread_should_stop()) {
        read_lock(&my_rwlock);
        printk(KERN_INFO "Read value: %lu\n", etx_global_variable);
        read_unlock(&my_rwlock);
        msleep(1000);
    }
    return 0;
}

// ... (rest of your module code)
```

2. Locking Between Bottom Halves

Scenario: Deals with situations where you have bottom halves (like tasklets or softirqs) that need to access shared data.

Mechanism: Uses the same read_lock(), read_unlock(), write_lock(), and write_unlock() functions as Approach 1.

Ensures synchronization between different bottom halves or within the same bottom half when accessing shared data.

Example:

```c
#include <linux/kernel.h>
#include <linux/module.h>
#include <linux/rwlock.h>
#include <linux/interrupt.h>

static rwlock_t my_rwlock;
static int etx_global_variable = 0;
static DECLARE_TASKLET(my_tasklet, tasklet_fn, 0);

static void tasklet_fn(unsigned long arg)
{
    read_lock(&my_rwlock);
    printk(KERN_INFO "Tasklet Function: %lu\n", etx_global_variable);
    read_unlock(&my_rwlock);
}

// ... (rest of your module code)
```

3. Locking Between User Context and Bottom Halves

Scenario: When **we need to synchronize access to shared data between kernel threads/processes in user space and bottom halves.**

Mechanism:

- read_lock_bh(): Acquires a read lock and disables soft interrupts before entering the critical section. This prevents soft interrupts from preempting the code while holding the lock.
- read_unlock_bh(): Releases the read lock and re-enables soft interrupts.
- write_lock_bh(): Same as read_lock_bh(), but acquires a write lock.
- write_unlock_bh(): Same as read_unlock_bh().

Usage: Crucial when you want to prevent soft interrupts from interfering with the critical section, ensuring data consistency.

Example:

```c
#include <linux/kernel.h>
#include <linux/module.h>
#include <linux/rwlock.h>
#include <linux/interrupt.h>

static rwlock_t my_rwlock;
static int etx_global_variable = 0;
static DECLARE_TASKLET(my_tasklet, tasklet_fn, 0);

static int thread_function(void *pv)
{
    while (!kthread_should_stop()) {
        write_lock_bh(&my_rwlock);
        etx_global_variable++;
        write_unlock_bh(&my_rwlock);
        msleep(1000);
    }
    return 0;
}

static void tasklet_fn(unsigned long arg)
{
    read_lock_bh(&my_rwlock);
    printk(KERN_INFO "Tasklet Function: %lu\n", etx_global_variable);
    read_unlock_bh(&my_rwlock);
}

// ... (rest of your module code)
```

4. Locking Between Hard IRQ and Bottom Halves

Scenario: When we need to synchronize access between hardware interrupt service routines (ISRs) and bottom halves.

Mechanism:

- read_lock_irq(): Disables all interrupts on the CPU before acquiring the read lock. This is the most stringent form of locking.
- read_unlock_irq(): Releases the read lock and re-enables all interrupts.
- write_lock_irq(): Same as read_lock_irq(), but acquires a write lock.
- write_unlock_irq(): Same as read_unlock_irq().

Usage: Essential for scenarios where you need to prevent any interrupt from interrupting the critical section, such as when dealing with hardware interrupts.

Example:

```c
#include <linux/kernel.h>
#include <linux/module.h>
#include <linux/rwlock.h>
#include <linux/interrupt.h>

static rwlock_t my_rwlock;
static int etx_global_variable = 0;
static DECLARE_TASKLET(my_tasklet, tasklet_fn, 0);

static irqreturn_t irq_handler(int irq, void *dev_id)
{
    read_lock_irq(&my_rwlock);
    printk(KERN_INFO "ISR Function: %lu\n", etx_global_variable);
    read_unlock_irq(&my_rwlock);
    tasklet_schedule(&my_tasklet);
    return IRQ_HANDLED;
}

static void tasklet_fn(unsigned long arg)
{
    write_lock_irq(&my_rwlock);
    etx_global_variable++;
    write_unlock_irq(&my_rwlock);
}

// ... (rest of your module code)
```

5. Locking Between Hard IRQ and Bottom Halves (IRQ Save/Restore)

Scenario: Similar to Approach 4, but allows you to save the current interrupt state before disabling interrupts and restore it later. This is useful when you need to maintain the previous interrupt state.

Mechanism:

- read_lock_irqsave(): Saves the current interrupt state (enabled/disabled), disables interrupts, and acquires the read lock.
- read_unlock_irqrestore(): Releases the read lock and restores the previously saved interrupt state.
- write_lock_irqsave(): Same as read_lock_irqsave(), but acquires a write lock.
- write_unlock_irqrestore(): Same as read_unlock_irqrestore().

Example:

```c
#include <linux/kernel.h>
#include <linux/module.h>
#include <linux/rwlock.h>
#include <linux/interrupt.h>

static rwlock_t my_rwlock;
static int etx_global_variable = 0;
static DECLARE_TASKLET(my_tasklet, tasklet_fn, 0);

static irqreturn_t irq_handler(int irq, void *dev_id)
{
    unsigned long flags;
    read_lock_irqsave(&my_rwlock, flags);
    printk(KERN_INFO "ISR Function: %lu\n", etx_global_variable);
    read_unlock_irqrestore(&my_rwlock, flags);
    tasklet_schedule(&my_tasklet);
    return IRQ_HANDLED;
}

static void tasklet_fn(unsigned long arg)
{
    unsigned long flags;
    write_lock_irqsave(&my_rwlock, flags);
    etx_global_variable++;
    write_unlock_irqrestore(&my_rwlock, flags);
}

// ... (rest of your module code)
```

6. Locking Between Hard IRQs

Scenario: When multiple hardware ISRs need to access shared data.
Mechanism: Uses the same locking functions as in Approaches 4 and 5 (*_irq() and *_irqsave() variants).
Usage: Ensures synchronization and data consistency between different hardware interrupt handlers.
Key Considerations:

- Interrupt Handling: The choice of locking functions depends heavily on the interrupt handling requirements.
- For simple cases, *_bh() might suffice.
- For scenarios involving hardware interrupts, *_irq() or *_irqsave() are necessary to prevent unexpected interrupts.

Performance: Disabling interrupts has performance implications. Use it only when necessary and for the shortest possible time.

Example:

```c
#include <linux/kernel.h>
#include <linux/module.h>
#include <linux/rwlock.h>
#include <linux/interrupt.h>

static rwlock_t my_rwlock;
static int etx_global_variable = 0;

static irqreturn_t irq_handler1(int irq, void *dev_id)
{
    read_lock_irq(&my_rwlock);
    printk(KERN_INFO "ISR Handler 1: %lu\n", etx_global_variable);
    read_unlock_irq(&my_rwlock);
    return IRQ_HANDLED;
}

static irqreturn_t irq_handler2(int irq, void *dev_id)
{
    write_lock_irq(&my_rwlock);
    etx_global_variable++;
    write_unlock_irq(&my_rwlock);
    return IRQ_HANDLED;
}

// ... (rest of your module code)
```

Signals in Linux

Definition: **In Linux, signals are software interrupts.** They are **asynchronous messages sent to a process to notify it of an event.**

Purpose: Signals are used for various purposes, including:

- Inter-process communication: As you mentioned, they enable communication between processes.
- Error handling: Notify processes of errors (e.g., segmentation faults, illegal instructions).
- Control flow: Interrupt the normal execution flow of a process.
- External events: Signal a process about external events (e.g., user input, system events).

Sending Signals:

- **User Space:** Processes can send signals to other processes using system calls like kill().
- **Kernel Space:**
  - Drivers: Device drivers can send signals to user-space processes using functions like send_signal_info().
  - Kernel Threads: Kernel threads can also send signals to user-space processes.

Receiving Signals:

Processes can handle signals in various ways:

Default action: The default action for a signal can be defined (e.g., terminate the process).

Key Points:

- Asynchronous: Signals are asynchronous, meaning they can interrupt the normal execution flow of a process at any time.
- Interrupts: Signals are similar to hardware interrupts in that they cause a change in the normal execution flow of a process.

Sending Signal from Linux Device Driver to User Space

1. Define the Signal
Choose a unique signal number to be used for communication. This signal number should not conflict with any existing signals.

Example:

```
#define SIGETX    44
```

2. Register the User-Space Application
The driver needs to know which process to send the signal to. This is typically done by obtaining the Process ID (PID) of the user-space application.

Registration Methods:

IOCTL: Use an IOCTL command **(e.g., REG_CURRENT_TASK) to allow the user-space application to register itself with the driver.** The driver can then store the PID of the current process.

Example:

```
static long etx_ioctl(struct file *file, unsigned int cmd, unsigned
long arg) {
    if (cmd == REG_CURRENT_TASK) {
        printk(KERN_INFO "REG_CURRENT_TASK\n");
        task = get_current(); // Get the current process
        signum = SIGETX;
    }
    return 0;
}
```

3. Send the Signal

Once an event occurs (e.g., an interrupt), the driver can send the signal to the registered user-space process.

- **send_signal_info(): This kernel function is used to send a signal to a specific process.**

```
// Interrupt handler for IRQ 11
static irqreturn_t irq_handler(int irq, void *dev_id) {
    struct siginfo info;

    printk(KERN_INFO "Shared IRQ: Interrupt Occurred\n");

    // Prepare the signal information
    memset(&info, 0, sizeof(struct siginfo));
    info.si_signo = SIGETX;
    info.si_code = SI_QUEUE;
    info.si_int = 1; // Optional: Pass data with the signal

    if (task != NULL) {
        printk(KERN_INFO "Sending signal to app\n");
        if(send_sig_info(SIGETX, &info, task) < 0) {
            printk(KERN_INFO "Unable to send signal\n");
        }
    }
    return IRQ_HANDLED;
}
```

4. Unregister the User-Space Application

To prevent unintended signal deliveries, unregister the user-space application when it is no longer interested in receiving signals from the driver.

Unregistration Methods:
Device File Close: Unregister the application when the device file is closed (release() system call).

Example:

```
static int etx_release(struct inode *inode, struct file *file) {
    struct task_struct *ref_task = get_current();
    printk(KERN_INFO "Device File Closed...!!!\n");

    // Delete the task registration if the current process is the
registered one
    if(ref_task == task) {
        task = NULL;
    }
    return 0;
}
```

Overview of Timers

Definition: A timer is a mechanism used to measure or control specific time intervals.

Key Characteristics:

- Measurement: Timers are used to measure the duration of events or the time elapsed between events.
- Control: Timers can be used to trigger events or actions after a specific time interval.
- Flexibility: Timers can be configured to count up (stopwatches) or count down (countdown timers).
- Versatility: Timers are used in a wide range of applications, from everyday household appliances (microwaves, washing machines) to complex industrial systems and scientific experiments.

Types of Timers:

Stopwatches:

- **Measure the elapsed time between the start and stop signals.**
- Used to measure the duration of events like races, sports activities, or cooking times.

Countdown Timers:

- **Count down from a pre-set time interval to zero**.
- Used for setting alarms, scheduling events, and controlling processes with time limits.

Timers in the Linux Kernel

Timer Interrupts:

- The foundation of **timekeeping in the Linux kernel lies in timer interrupts.**
- These interrupts are generated periodically by the system's hardware timer (often a dedicated hardware device).
- The frequency of these interrupts is **typically in the range of milliseconds or microseconds, depending on the system configuration.**
- Each **timer interrupt increments a system-wide counter (often called the jiffies counter).**

Jiffies Counter:

- **The jiffies counter represents the number of timer interrupts (jiffies) that have occurred since the system boot.**

- The **value of jiffies is continuously incremented by the timer interrupt handler.**

Kernel Timers:

- Kernel timers provide a **mechanism to schedule the execution of a function (called a timer function) at a specific time in the future.**
- **They are implemented as a data structure that holds information about the timer, such as:**
    - The **timer function to be executed.**
    - The **time at which the timer should expire.**
    - The **interval between timer expirations (for repeating timers).**
    - **When a timer expires, the kernel executes the associated timer function.**

Uses of Kernel Timers:

- **Scheduling tasks:** Scheduling tasks **to run at specific times or intervals.**
- **Network timeouts:** Implementing timeouts for **network operations (e.g., waiting for responses from other systems).**
- **Device driver operations: Handling time-sensitive operations in device drivers (e.g., timeouts for I/O operations).**
- System maintenance: **Performing periodic maintenance tasks (e.g., checking disk space, cleaning up caches).**

Types of Kernel Timers:

- **One-shot timers:** Expire only once.
- **Repeating timers:** Expire periodically at a specified interval.

Kernel Timer API in Linux Kernel Device Driver

The Linux kernel provides the **Kernel Timer API for creating, registering, and deleting non-periodic timers.**
To use kernel timers, include the header file:

```
#include <linux/timer.h>
```

Timers in the kernel are described by the **timer_list structure:**

```
struct timer_list {
    /* ... */
    unsigned long expires;
    void (*function)(unsigned long);
    unsigned long data;
};
```

- expires: Specifies **the timer's expiration time** (in jiffies).
- function: **The callback function called when the timer expires.**
- data: **Argument passed to the callback function.**

Timer Initialization

The Linux kernel provides the Kernel Timer API for creating, registering, and deleting non-periodic timers.

1. timer_setup: Initializes a kernel timer by setting up its callback function and data. It's used in newer kernel versions.

Function:

```
void timer_setup(struct timer_list *timer, void (*function)(unsigned long), unsigned long data);
```

- timer: Timer to be initialized.
- function: Callback function with struct timer_list * argument.
- data: Data passed to the callback.

2. mod_timer: Modifies the expiration time of an active timer or starts it if it's inactive. It's an efficient way to update the timer's timeout.

Function:

```
int mod_timer(struct timer_list *timer, unsigned long expires);
```

- timer: Timer to modify.
- expires: New expiration time.

Efficiently updates the timer's timeout without removing and re-adding it.

3. del_timer: Deactivates a timer, whether it is active or inactive.

Function:

```
int del_timer(struct timer_list *timer);
```

- timer: Timer to deactivate.
- Stops the timer, works on both active and inactive timers.

1. Callback Function:

Purpose: This function is the core of a kernel timer. It's the code that gets executed when the timer expires.

Program for understanding :

```c
#include <linux/module.h>
#include <linux/timer.h>
#include <linux/kernel.h>
#include <linux/init.h>

#define TIMER_TIMEOUT 5  // Timeout in seconds

static struct timer_list my_timer;

void timer_callback(struct timer_list *timer) {
    pr_info("Timer callback function executed.\n");

    // Re-schedule the timer
    mod_timer(&my_timer, jiffies + msecs_to_jiffies(TIMER_TIMEOUT * 1000));
}

static int __init my_module_init(void) {
    pr_info("Module loaded. Setting up timer.\n");

    // Initialize the timer
    timer_setup(&my_timer, timer_callback, 0);

    // Start the timer with an initial timeout
    mod_timer(&my_timer, jiffies + msecs_to_jiffies(TIMER_TIMEOUT * 1000));

    return 0;
}

static void __exit my_module_exit(void) {
    pr_info("Module unloaded. Deactivating timer.\n");

    // Deactivate the timer
    del_timer(&my_timer);
}

module_init(my_module_init);
module_exit(my_module_exit);

MODULE_LICENSE("GPL");
MODULE_AUTHOR("Vicky");
MODULE_DESCRIPTION("A simple example of using kernel timers.");
```

High Resolution Timers (HRT)

HRTs are a specialized timer mechanism within the Linux kernel designed to address the limitations of traditional kernel timers in terms of resolution and performance.

Need for HRT:

Limitations of Kernel Timers: Kernel timers are bound to jiffies, which represent the number of timer **interrupts. This granularity (resolution) might not be sufficient for applications requiring precise timing, such as:**

- Multimedia applications: Audio/video processing, gaming, etc., where precise timing is crucial for smooth playback and synchronization.
- Networking: High-performance networking applications that require precise timing for packet scheduling and synchronization.
- Real-time systems: Applications with strict timing requirements.

HRT Features:

- Higher Resolution: HRTs provide much finer time resolution than kernel timers, typically in nanoseconds.
- 64-bit Timestamps: HRTs use 64-bit timestamps for greater precision and support for longer time intervals.

Enabling HRT:

- Kernel Configuration: HRTs are enabled by default in most modern Linux kernels. However, you can check the kernel configuration file **(/boot/config) for the CONFIG_HIGH_RES_TIMERS** option.
- **/proc/timer_list:** This file provides information about the timer subsystem. Look for **.resolution with a value in nanoseconds** and event_handler as hrtimer_interrupt to confirm HRT support.
- clock_getres() system call: This system call can be used to obtain the resolution of the system's clock.

High-Resolution Timer API

1 Header Files

```
#include <linux/hrtimer.h>
#include <linux/ktime.h>
```

- #include <linux/hrtimer.h>: Contains definitions and functions for high-resolution timers.
- #include <linux/ktime.h>: Provides support for handling ktime_t, a datatype for time values in nanoseconds.

2 Data Structure: struct hrtimer:

Purpose: Represents a high-resolution timer.

```
struct hrtimer {
        struct rb_node node;          /* Red-black tree node */
        ktime_t expires;              /* Time at which the timer
expires */
        enum hrtimer_mode mode;      /* HRTIMER_MODE_REL or
HRTIMER_MODE_ABS */
        unsigned flags;              /* Flags for the timer */
        int signo;                   /* Signal number (if used) */
        siginfo_t si;                /* Signal information */
        void (*function)(struct hrtimer *); /* Callback function */
        struct task_struct *task;    /* Task to send signal to */
        ktime_t softexpires;          /* Software expiration time
*/
};
```

Fields:

- struct rb_node node: Node for red-black tree insertion based on time order.
- ktime_t expires: Absolute expiry time in the internal representation of HR timers.
- int (*function)(struct hrtimer *): Callback function called when the timer expires.
- struct hrtimer_base *base: Pointer to the timer base (specific to the CPU and clock).

3 ktime_t Datatype

Purpose: Stores time values with nanosecond precision.

Conversion Function:

```
ktime_t ktime_set(long secs, long nanosecs);
```

Converts seconds and nanoseconds to ktime_t.

Parameters:

- secs: Seconds to set.
- nanosecs: Nanoseconds to set.

Returns: ktime_t value representing the time.

4 Timer Initialization

4.1 hrtimer_init

Purpose: Initializes an HR timer.

Function:

```
void hrtimer_init(struct hrtimer *timer, clockid_t clock_id, enum hrtimer_mode
mode);
```

Parameters:

- timer: Pointer to the HR timer to initialize.
- clock_id: Clock to use (e.g., CLOCK_MONOTONIC, CLOCK_REALTIME).
- mode: Timer mode (absolute HRTIMER_MODE_ABS or relative HRTIMER_MODE_REL).

5 Starting and Modifying Timers

5.1 hrtimer_start

Purpose: Starts or restarts an HR timer.

Function:

```
int hrtimer_start(struct hrtimer *timer, ktime_t time, const enum
hrtimer_mode mode);
```

Parameters:

- timer: Timer to start.
- time: Expiry time.
- mode: Expiry mode (HRTIMER_MODE_ABS or HRTIMER_MODE_REL).

Returns:

- 0 on success.
- 1 if the timer was already active.

6 Stopping and Canceling Timers

6.1hrtimer_cancel

Purpose: Cancels an active timer and waits for its handler to finish.

Function:

```
int hrtimer_cancel(struct hrtimer *timer);
```

Parameters:

- timer: Timer to cancel.

Returns:

- 0 if the timer was inactive.
- 1 if the timer was active.

7 Changing Timer Timeout

7.12 hrtimer_forward_now

Purpose: Forwards the timer's expiry from the current time by a specified interval.

Function:

```
hrtimer_forward_now(struct hrtimer *timer, ktime_t interval);
```

Parameters:

- timer: Timer to forward.
- interval: Interval to forward.

Returns: Number of overruns.

Example code:

```c
#include <linux/module.h>
#include <linux/kernel.h>
#include <linux/hrtimer.h>
#include <linux/ktime.h>

static struct hrtimer my_hrtimer;

static enum hrtimer_restart my_hrtimer_callback(struct hrtimer *timer)
{
    printk(KERN_INFO "High Resolution Timer Expired!\n");

    // Reschedule the timer to expire again in 1 second
    hrtimer_forward_now(timer, ktime_set(1, 0));

    return HRTIMER_RESTART;
}

static int __init my_module_init(void)
{
    hrtimer_init(&my_hrtimer, CLOCK_MONOTONIC, HRTIMER_MODE_REL);
    my_hrtimer.function = my_hrtimer_callback;

    hrtimer_start(&my_hrtimer, ktime_set(1, 0), HRTIMER_MODE_REL);

    printk(KERN_INFO "High Resolution Timer Module Loaded\n");
    return 0;
}

static void __exit my_module_exit(void)
{
    hrtimer_cancel(&my_hrtimer);
    printk(KERN_INFO "High Resolution Timer Module Unloaded\n");
}

module_init(my_module_init);
module_exit(my_module_exit);
MODULE_LICENSE("GPL");
```

What is a GPIO?

General Purpose Input/Output (GPIO): An interface on microcontrollers and other embedded systems.

Functionality: **Each GPIO pin can be configured as either an input or output.**

Input: Reads the digital signal (high or low) from an external device.
Output: Drives an external device with a digital signal (high or low).

Linux Kernel GPIO Support:
gpiolib: The Linux kernel provides a framework called **gpiolib for managing GPIO pins**.
Device Drivers: **GPIO drivers** are responsible for interacting with **the specific GPIO controller on the target hardware.**
User Space Access: User-space applications can interact with GPIOs through various mechanisms:

- sysfs interface: Access GPIOs through files in the /sys/class/gpio directory.
- Character devices: Create a character device driver to provide a more structured interface for GPIO access.
- Platform-specific APIs: Some platforms may offer platform-specific APIs for GPIO access.

GPIO APIs in Linux Kernel:

To use these APIs, we need to include the linux/gpio.h header file in kernel module.

```
#include <linux/gpio.h>
```

1. Validating GPIO

GPIO pin, it's crucial to validate whether the GPIO number is valid for the platform.

```
bool gpio_is_valid(int gpio_number);
```

Parameters:

- gpio_number: The GPIO number to validate.

Returns:

- true if the GPIO number is valid, false otherwise.

2. Requesting GPIO

We must request a GPIO before using it to ensure exclusive access.

```
int gpio_request(unsigned gpio, const char *label);
```

Parameters:

- gpio: The GPIO number to request.
- label: A string label for the GPIO, visible in /sys/kernel/debug/gpio.

Returns: 0 on success, a negative number on failure.

2.1 Request one GPIO with flags:

```
int gpio_request_one(unsigned gpio, unsigned long flags, const char
*label);
```

2.2 Request multiple GPIOs:

```
int gpio_request_array(struct gpio *array, size_t num);
```

3. Exporting GPIO

To debug or manipulate GPIOs from user space, you can export a GPIO to sysfs.

```
int gpio_export(unsigned int gpio, bool direction_may_change);
```

Parameters:

- gpio: The GPIO number to export.
- direction_may_change: Allows user space to change the direction if true.

Returns:

0 on success, an error code otherwise.

4. Unexporting GPIO

To remove a GPIO from sysfs after it has been exported:

```
void gpio_unexport(unsigned int gpio);
```

Parameters:

- gpio: The GPIO number to unexport.

5. Setting GPIO Direction

To configure a GPIO as input or output:

5.1 Set as Input:

```
int gpio_direction_input(unsigned gpio);
```

Parameters:

- gpio: The GPIO number to set as input.

Returns:

0 on success, an error code otherwise.

5.2 Set as Output:

```
int gpio_direction_output(unsigned gpio, int value);
```

Parameters:

- gpio: The GPIO number to set as output.
- value: Initial value for the output (0 for low, 1 for high).

Returns:

0 on success, an error code otherwise.

6. Changing GPIO Value

To change the value of a GPIO configured as output:

```
void gpio_set_value(unsigned int gpio, int value);
```

Parameters:

- gpio: The GPIO number.
- value: The value to set (0 for low, 1 for high).

7. Reading GPIO Value

To read the current value of a GPIO:

```
int gpio_get_value(unsigned gpio);
```

Parameters:

- gpio: The GPIO number to read.

Returns:

The value of the GPIO (0 or 1).

8. GPIO Interrupt (IRQ)

Converting a GPIO to an interrupt request line:

```
int gpio_to_irq(unsigned gpio);
```

Parameters:

- gpio: The GPIO number.

Returns:

The IRQ number associated with the GPIO.

9. Releasing GPIO

To release a previously requested GPIO:

```
void gpio_free(unsigned int gpio);
```

Parameters:

gpio: The GPIO number to release.

Variants:

9.1 Release multiple GPIOs:

```
void gpio_free_array(struct gpio *array, size_t num);
```

10. Debouncing GPIO

To set or get debounce time for a GPIO:

```
int get_set_debounce(unsigned gpio, unsigned debounce);
```

Parameters:

- gpio: The GPIO number.
- debounce: The debounce time in milliseconds.

Returns:

- 0 on success, an error code otherwise.