Coherent (or consistent) memory is **memory for which a write by either the device or the processor can immediately be read by the processor or device without having to worry about caching effects**.

The function call dma\_set\_coherent\_mask(&pdev->dev, DMA\_BIT\_MASK(32)); is used in Linux kernel device drivers to set the **DMA coherent mask** for a device. This defines the range of physical addresses that the device can use for Direct Memory Access (DMA) operations.

## **What is Spinlock**

[synchronization](https://www.geeksforgeeks.org/introduction-of-process-synchronization/) mechanism usedSpinlock is a in operating systems to protect shared resources from single access by multiple threads or processes. Unlike other synchronization methods such as [semaphores](https://www.geeksforgeeks.org/semaphores-in-process-synchronization/) or [mutexes](https://www.geeksforgeeks.org/mutex-vs-semaphore/), spinlocks use a busy-wait method, where a thread continuously selects a lock until it becomes available.

**SPINLOCK**

### **Spinlocks vs. Other Synchronization Mechanisms**

|  |  |  |
| --- | --- | --- |
| **Mechanism** | **Use Case** | **Behavior** |
| **Spinlocks** | Short critical sections in non-sleepable contexts | Busy-waits until the lock is free. |
| **Mutexes** | Longer critical sections in sleepable contexts | Puts the thread to sleep if locked. |
| **Read-Write Locks** | Separate read and write locks for shared data access | Allows multiple readers but only one writer. |

A **spinlock** is applied to **shared resources or critical sections of code** that need to be accessed safely by multiple threads (or execution contexts) concurrently. It is not applied to threads themselves but is used to synchronize access to data or code that threads (or interrupt handlers) might access simultaneously.

### **Why Use a Mutex?**

* A **mutex** is a synchronization primitive used to enforce mutual exclusion.
* It ensures that only one thread can access a shared resource or execute a critical section at any given time.
* Unlike spinlocks, mutexes **can sleep** when the lock is not available, making them more suitable for **blocking contexts**.

**SYNTAX** void mutex\_init(struct mutex \*lock);

**EXAMPLE FOR SPIN LOCK**

// Shared variable

int counter = 0;

spinlock\_t lock;

// Initialize spinlock

spin\_lock\_init(&lock);

// Thread 1

spin\_lock(&lock); // Acquire the spinlock

counter = counter + 1; // Critical section: increment counter

spin\_unlock(&lock); // Release the spinlock

// Thread 2

spin\_lock(&lock); // Acquire the spinlock

counter = counter + 1; // Critical section: increment counter

spin\_unlock(&lock); // Release the spinlock

**INTERRUPT LINE DISABLING**

The combination of test\_and\_set\_bit() and disable\_irq\_nosync() is likely used to prevent re-entrancy issues or ensure that an interrupt is disabled under specific conditions:

1. **test\_and\_set\_bit(0, &priv->flags)**:
   * This checks and sets a flag (bit 0) in priv->flags atomically.
   * If the flag was **not set** before (test\_and\_set\_bit returns 0), it means this is the first time the interrupt handler or the critical section is being executed.
   * If the flag was already **set** (test\_and\_set\_bit returns 1), it indicates that the interrupt has already been processed or a previous handler is already running, and we should not re-enter or handle it again.
2. **disable\_irq\_nosync(irq)**:
   * If the bit was **not set** before (test\_and\_set\_bit returned 0), the code proceeds to **disable the interrupt** (irq).
   * This prevents further interrupts from being serviced, effectively **disabling the IRQ line** while the current interrupt handler is running.

**RELATED TO OF\_NODE (UIO DRIVER)**

So we have a **platform device** and inside it is a structure **struct device.** Using these we can get device configuration data and use it in our platform driver to configure device.

In Simple sense this is what happens,

When a match occurs,the probe function of the driver is called, with a **struct platform\_device** structure as the parameter, which contains a **struct device dev** field, in which there is a field struct device\_node \*of\_node that corresponds that corresponds to the node associated to our device, so that one can use it to extract the device settings

**struct\_resource{}**

struct [resource](https://elixir.bootlin.com/linux/v6.11/C/ident/resource) {

[resource\_size\_t](https://elixir.bootlin.com/linux/v6.11/C/ident/resource_size_t) start;

[resource\_size\_t](https://elixir.bootlin.com/linux/v6.11/C/ident/resource_size_t) end;

const char \*name;

unsigned long flags;

unsigned long desc;

struct [resource](https://elixir.bootlin.com/linux/v6.11/C/ident/resource) \*[parent](https://elixir.bootlin.com/linux/v6.11/C/ident/parent), \*[sibling](https://elixir.bootlin.com/linux/v6.11/C/ident/sibling), \*[child](https://elixir.bootlin.com/linux/v6.11/C/ident/child);

};