4. Interrupt

NXKR_YoungSikYang

Exported on 03/12/2024

Table of Contents

1. Polling

Each case is checked by one by one in an infinite loop. This busy-waiting causes overheads. This is why the interrupt is used instead.

Polling

```
while True:
  if request==0:
    response0()
  elif request==1:
    response1()
  else:
    response2()
```

2. About interrupt

An **interrupt** is a request for the processor to temporarily halt the current task and switch to another task with a higher priority.

2.1 Types of the interrupt

- **external interrupt(hardware interrupt)** happens as a result of outside interference such as from the user or from the peripherals. It functions as a notifier. (Considered a type of exception)
- **internal interrupt**, also called Trap, happens when wrong instructions or data are used. (exceptions like divided by zero, overflow)
- **software interrupt(system call)**: is a type of interrupt that is triggered by a specific instruction in a program, rather than by an external event or hardware malfunction. It's a mechanism for a program to interrupt the current process flow and request a service from the operating system.
  A program running in user mode needs to send a system call to the operating system to access system resources and perform tasks in kernel mode.

2.2 Steps of an interrupt

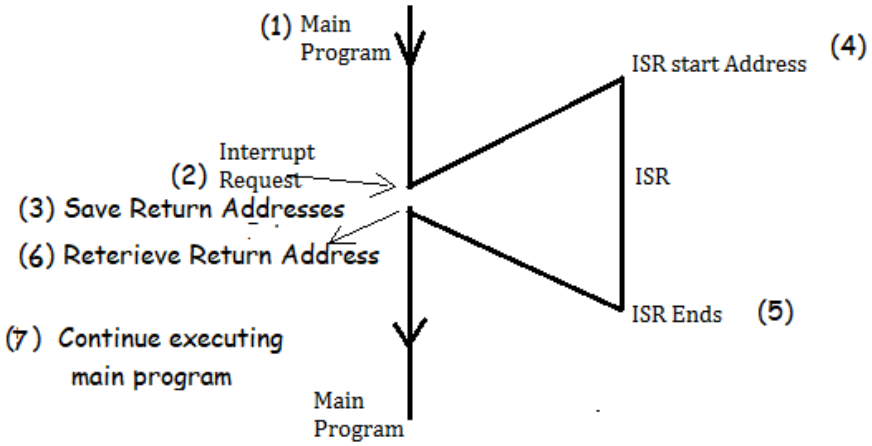


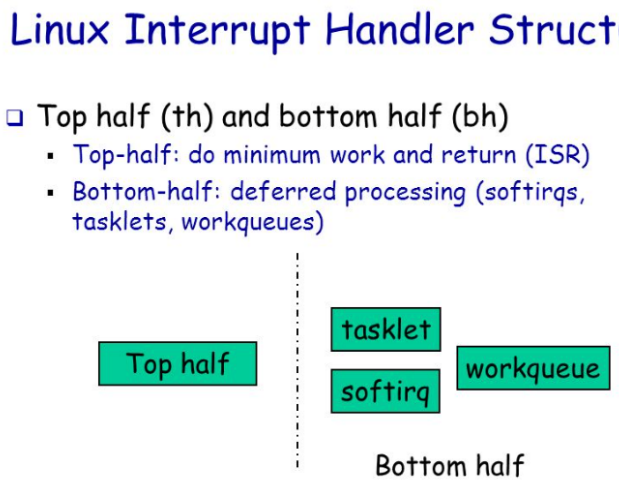
Figure 3.2 Interrupt Cycle

1. An interrupt occurs
2. Current program status is saved onto a stack
3. Jump to the interrupt vector
4. The ISR is executed
5. Jump to the previous program

2.3. ISR(Interrupt Service Routine)

ISR refers to a function that executes in response to an interrupt signal.

ISR handling in Linux is divided into top-half and bottom-half to balance the need for immediate response to interrupts with the need to perform more complex processing without compromising system responsiveness.



2.3.1 Top-half

- The top half refers to the initial response(interrupt handler) to an interrupt.
- It is supposed to perform minimal work such as acknowledging the interrupt, reading or writing the minimal necessary data to ensure the system can resume its operation as quickly as possible.
- It defers the rest of the time-consuming work to the bottom half.
- Executed in the interrupt context, thus cannot sleep.

2.3.2 Bottom-half

This is where the deferred work from the interrupt is handled and can be done in various ways as listed below:

Feature	SoftIRQ	Tasklet	Workqueue	Threaded IRQ
Deferred work	Deferred work is executed in a SoftIRQ statically defined at compile time	Deferred work is put in a queue	Deferred work is put in a queue	Deferred work is executed in the context of a kernel thread where the kernel synchronization mechanisms are available
Context	Interrupt(atomic) context; cannot sleep	Interrupt(atomic) context; cannot sleep	Process context; can sleep	Process context; can sleep
Use Case	Suitable for high-speed and low-latency tasks e.g. Network packet processing, timer updates	Non-blocking operations without the need for waiting e.g. Device driver deferred processing	Long-running jobs that might need to wait e.g. Filesystem operations, scheduled maintenance tasks	Tasks that need the standard synchronization APIs of the kernel like the mutex
Note	Rarely used since there is already the Tasklet			Simple code, easy to use

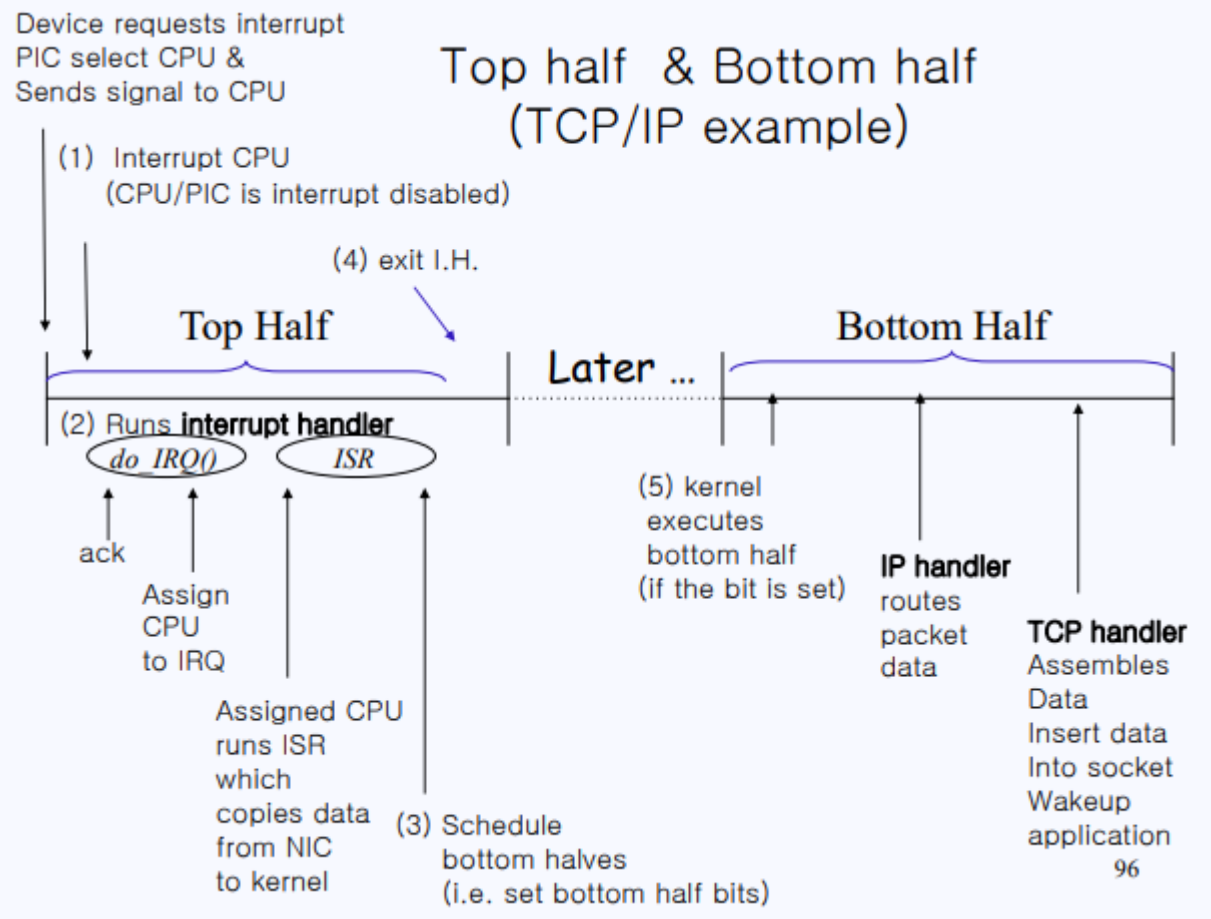
2.3.2.1 Difference between delay() and sleep()

<linux/delay.h> has delay() and sleep() that allow pausing

- **delay()**: busy-waiting(spin-lock) where the processor remains active but does nothing productive while waiting for the specified duration to elapse.
  - Bottom-half mechanisms that cannot sleep can be paused by **delay()**
- **sleep()**: causes the thread to yield the CPU and enter a non-running state for the specified duration. The scheduler will not allocate CPU to the sleeping process, allowing other processes to run.

2.3.3 Example

Example of the top-half & bottom-half in TCP/IPS


Device requests interrupt
PIC select CPU &
Sends signal to CPU
Top half & Bottom half
(TCP/IP example)
(1) Interrupt CPU
(CPU/PIC is interrupt disabled)
(4) exit I.H.
Top Half
Later ...
Bottom Half
(2) Runs interrupt handler
do_IRQ()
ISR
ack
Assign
CPU
to IRQ
Assigned CPU
runs ISR
which
copies data
from NIC
to kernel
(3) Schedule
bottom halves
(i.e. set bottom half bits)
(5) kernel
executes
bottom half
(if the bit is set)
IP handler
routes
packet
data
TCP handler
Assembles
Data
Insert data
Into socket
Wakeup
application
96

3. Interrupt practice

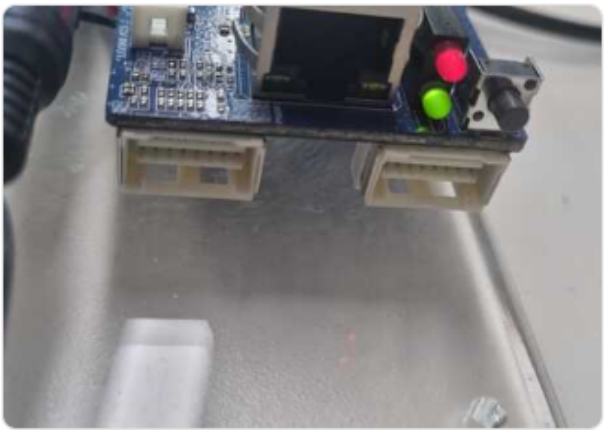
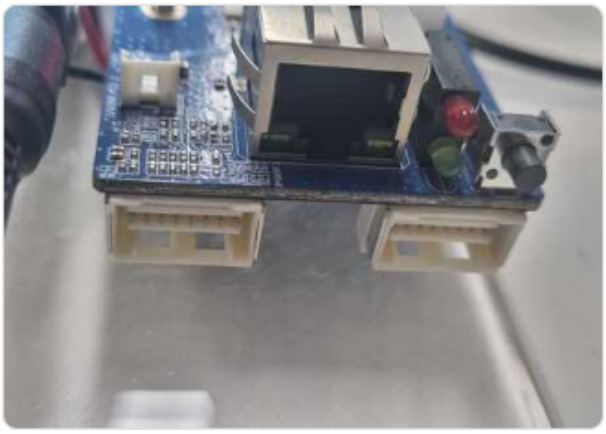
This section describes several ways to register an interrupt that prints a log and toggles LEDs on button press.

In all the examples below, on button presses:

- "Button pressed!" is printed

```
root@s5p6818:~# [   28.368000] Button pressed!
[   29.132000] Button pressed!
[   29.772000] Button pressed!
[   30.116000] Button pressed!
[   31.948000] Button pressed!
```

- And LEDs are toggled

3.1 Directly in the top-half

In this practice, an interrupt is registered that processes its work in the top-half without passing it to the bottom-half.

In **btn_irq_init()**

1. GPIOs are initialized
2. **gpio_to_irq()** gets the IRQ number that corresponds to the button GPIO.
3. **request_irq()** registers an interrupt that executes **btn_irq_handler()** in reaction to button press.

top_half.c

```
#include <linux/module.h>
#include <linux/gpio.h>
#include <linux/interrupt.h>
#include <linux/kernel.h>

#define button 3 // Example GPIO number
#define LED1 4
#define LED2 5
#define ON 0
#define OFF 1

static unsigned int irq_number;
unsigned int gpio_value = OFF;

// Interrupt handler function(top-half)
static irqreturn_t btn_irq_handler(int irq, void *dev_id)
{
    gpio_value = gpio_get_value(LED1);
    gpio_set_value(LED1, !gpio_value);
    gpio_set_value(LED2, !gpio_value);
    // gpio_value != gpio_value;
    pr_info("Button pressed!\n");
    return IRQ_HANDLED;
}

static int __init btn_irq_init(void)
{
    // Request GPIOs
    int result_btn, result1, result2, result_irq;
    result_btn = gpio_request(button, "sysfs");
    result1 = gpio_request(LED1, "sysfs");
    result2 = gpio_request(LED2, "sysfs");
    if (result_btn || result1 || result2) {
        pr_info("Cannot request the GPIO\n");
        return 0;
    }
    // Set the direction of button GPIO
    gpio_direction_input(button);
    // Get the IRQ number for our GPIO
    irq_number = gpio_to_irq(button);
    if (irq_number < 0) {
        printk(KERN_INFO "GPIO to IRQ mapping failed\n");
        return irq_number;
    }
    // Request the IRQ line
    result_irq = request_irq(irq_number, btn_irq_handler, IRQF_TRIGGER_FALLING,
"btn_irq", NULL);
    if (result_irq) {
```

```
        printk(KERN_INFO "IRQ request failed\n");
        return result_irq;
    }
    return 0;
}

static void __exit btn_irq_exit(void) {
    free_irq(irq_number, NULL);
    gpio_free(button);
    gpio_free(LED1);
    gpio_free(LED2);
}

module_init(btn_irq_init);
module_exit(btn_irq_exit);

MODULE_LICENSE("GPL");
MODULE_AUTHOR("Your Name");
MODULE_DESCRIPTION("A simple Linux driver for a GPIO interrupt");
MODULE_VERSION("0.1");
```

3.2 Using softIRQ

This method registers an interrupt that passes its work to the bottom-half using softIRQ.

1. Add a softIRQ(**YANG_SOFTIRQ**) in interrupt.h

```
enum
{
    HI_SOFTIRQ=0,
    TIMER_SOFTIRQ,
    NET_TX_SOFTIRQ,
    NET_RX_SOFTIRQ,
    BLOCK_SOFTIRQ,
    BLOCK_IOPOLL_SOFTIRQ,
    TASKLET_SOFTIRQ,
    SCHED_SOFTIRQ,
    HRTIMER_SOFTIRQ, /* U
                numbering
    RCU_SOFTIRQ,    /* Pr

    NR_SOFTIRQS,
    YANG_SOFTIRQ
};
```

1. **open_softirq()** in **btn_irq_init()** assigns **bottom_half()** to the softIRQ "**YANG_SOFTIRQ**".
2. **raise_softirq()** in **btn_irq_handler()** triggers the target softIRQ.

softIRQ.c

```
#include <linux/module.h>
#include <linux/gpio.h>
#include <linux/interrupt.h>
#include <linux/kernel.h>
#include <linux/slab.h> // kmalloc()

#define button 3 // Example GPIO number
#define LED1 4
#define LED2 5
#define ON 0
#define OFF 1

static unsigned int irq_number;
unsigned int gpio_value = OFF;

static void bottom_half(struct softirq_action *action){
    gpio_value = gpio_get_value(LED1);
    gpio_set_value(LED1, !gpio_value);
    gpio_set_value(LED2, !gpio_value);
    // gpio_value != gpio_value;
    pr_info("Button pressed!\n");
}


// Interrupt handler function(top-half)
static irqreturn_t btn_irq_handler(int irq, void *dev_id)
{
    raise_softirq(YANG_SOFTIRQ);
    return IRQ_HANDLED;
}

static int __init btn_irq_init(void)
{
    // Request GPIOs
    int result_btn, result1, result2, result_irq;
    result_btn = gpio_request(button, "sysfs");
    result1 = gpio_request(LED1, "sysfs");
    result2 = gpio_request(LED2, "sysfs");
    if (result_btn || result1 || result2) {
        pr_info("Cannot request the GPIO\n");
        return 0;
    }
    // Set the direction of button GPIO
    gpio_direction_input(button);
    // Get the IRQ number for our GPIO
    irq_number = gpio_to_irq(button);
    if (irq_number < 0) {
        printk(KERN_INFO "GPIO to IRQ mapping failed\n");
```

```
        return irq_number;
    }
    // Assign bottom half() to the softIRQ "YANG_SOFTIRQ"
    open_softirq(YANG_SOFTIRQ, bottom_half);
    // Request the IRQ line
    result_irq = request_irq(irq_number, btn_irq_handler, IRQF_TRIGGER_FALLING,
"btn_irq", NULL);
    if (result_irq) {
        printk(KERN_INFO "IRQ request failed\n");
        return result_irq;
    }
    return 0;
}

static void __exit btn_irq_exit(void) {
    free_irq(irq_number, NULL);
    gpio_free(button);
    gpio_free(LED1);
    gpio_free(LED2);
}

module_init(btn_irq_init);
module_exit(btn_irq_exit);

MODULE_LICENSE("GPL");
MODULE_AUTHOR("Your Name");
MODULE_DESCRIPTION("A simple Linux driver for a GPIO interrupt");
MODULE_VERSION("0.1");
```

3.3 Using tasklet

This method registers an interrupt that passes its work to the bottom-half using tasklet.

1. **DECLARE_TASKLET()** initializes a **tasklet_struct** named tasklet using **bottom_half()** statically at compile time.
    - `interrupt.h`

      ```
      #define DECLARE_TASKLET(name, func, data) \
      struct tasklet_struct name = { NULL, 0, ATOMIC_INIT(0), func, data }
      ```

1. A tasklet is placed into one queue out of two, depending on the priority (1. high priority 2. normal priority).
    - **tasklet_schedule()** puts the **tasklet_struct** in the normal priority queue.
    - **tasklet_hi_schedule()** will put the **tasklet_struct** in the high priority queue.
2. **tasklet_kill()** in **btn_irq_exit()** kills the tasklet.

The code below demonstrates the static allocation of a tasklet. Dynamic allocation is also possible following the same method described in section **3.4.2**.

tasklet.c

```c
#include <linux/module.h>
#include <linux/gpio.h>
#include <linux/interrupt.h>
#include <linux/kernel.h>
#include <linux/workqueue.h>
#include <linux/slab.h> // kmalloc()

#define button 3 // Example GPIO number
#define LED1 4
#define LED2 5
#define ON 0
#define OFF 1

static unsigned int irq_number;
unsigned int gpio_value = OFF;

static void bottom_half(const char *data){
    gpio_value = gpio_get_value(LED1);
    gpio_set_value(LED1, !gpio_value);
    gpio_set_value(LED2, !gpio_value);
    // gpio_value != gpio_value;
    pr_info("Button pressed! %s\n", data);
}

DECLARE_TASKLET(tasklet, bottom_half, "hello");

// Interrupt handler function(top-half)
static irqreturn_t btn_irq_handler(int irq, void *dev_id)
{
    tasklet_schedule(&tasklet);
    return IRQ_HANDLED;
}

static int __init btn_irq_init(void)
{
    // Request GPIOs
    int result_btn, result1, result2, result_irq;
    result_btn = gpio_request(button, "sysfs");
    result1 = gpio_request(LED1, "sysfs");
    result2 = gpio_request(LED2, "sysfs");
    if (result_btn || result1 || result2) {
        pr_info("Cannot request the GPIO\n");
        return 0;
    }
    // Set the direction of button GPIO
    gpio_direction_input(button);
    // Get the IRQ number for our GPIO
    irq_number = gpio_to_irq(button);
```

```
    if (irq_number < 0) {
        printk(KERN_INFO "GPIO to IRQ mapping failed\n");
        return irq_number;
    }
    // Request the IRQ line
    result_irq = request_irq(irq_number, btn_irq_handler, IRQF_TRIGGER_FALLING,
"btn_irq", NULL);
    if (result_irq) {
        printk(KERN_INFO "IRQ request failed\n");
        return result_irq;
    }
    return 0;
}

static void __exit btn_irq_exit(void) {
    free_irq(irq_number, NULL);
    gpio_free(button);
    gpio_free(LED1);
    gpio_free(LED2);
    tasklet_kill(&tasklet);
}

module_init(btn_irq_init);
module_exit(btn_irq_exit);

MODULE_LICENSE("GPL");
MODULE_AUTHOR("Your Name");
MODULE_DESCRIPTION("A simple Linux driver for a GPIO interrupt");
MODULE_VERSION("0.1");
```

3.4 Using workqueue

This method registers an interrupt that passes its work to the bottom-half using workqueue.

3.4.1 Static allocation of work_struct and using the kernel-global workqueue

1. **DECLARE_WORK()** initializes a **work_struct** named work using **bottom_half()** statically at compile time.
    - `workqueue.h`

```
#define DECLARE_WORK(n, f)                                      \
    struct work_struct n = __WORK_INITIALIZER(n, f)
```

1. **schedule_work()** puts the **work_struct** in the kernel-global workqueue.

workqueue.c

```
#include <linux/module.h>
#include <linux/gpio.h>
#include <linux/interrupt.h>
#include <linux/kernel.h>
#include <linux/workqueue.h>
#include <linux/slab.h> // kmalloc()

#define button 3 // Example GPIO number
#define LED1 4
#define LED2 5
#define ON 0
#define OFF 1

static unsigned int irq_number;
unsigned int gpio_value = OFF;

static void bottom_half(struct work_struct *work) {
    gpio_value = gpio_get_value(LED1);
    gpio_set_value(LED1, !gpio_value);
    gpio_set_value(LED2, !gpio_value);
    // gpio_value != gpio_value;
    pr_info("Button pressed!\n");
}

// Initialize a work_struct named work using bottom_half() statically at compile
time.
DECLARE_WORK(work, bottom_half);

// Interrupt handler function(top-half)
static irqreturn_t btn_irq_handler(int irq, void *dev_id)
{
    // Put the work in the kernel-global workqueue
    schedule_work(&work);
    return IRQ_HANDLED;
}

static int __init btn_irq_init(void)
{
    // Request GPIOs
    int result_btn, result1, result2, result_irq;
    result_btn = gpio_request(button, "sysfs");
    result1 = gpio_request(LED1, "sysfs");
    result2 = gpio_request(LED2, "sysfs");
    if (result_btn || result1 || result2) {
        pr_info("Cannot request the GPIO\n");
        return 0;
    }
    // Set the direction of button GPIO
```

```
    gpio_direction_input(button);
    // Get the IRQ number for our GPIO
    irq_number = gpio_to_irq(button);
    if (irq_number < 0) {
        printk(KERN_INFO "GPIO to IRQ mapping failed\n");
        return irq_number;
    }
    // Request the IRQ line
    result_irq = request_irq(irq_number, btn_irq_handler, IRQF_TRIGGER_FALLING,
"btn_irq", NULL);
    if (result_irq) {
        printk(KERN_INFO "IRQ request failed\n");
        return result_irq;
    }
    return 0;
}

static void __exit btn_irq_exit(void) {
    free_irq(irq_number, NULL);
    gpio_free(button);
    gpio_free(LED1);
    gpio_free(LED2);
}

module_init(btn_irq_init);
module_exit(btn_irq_exit);

MODULE_LICENSE("GPL");
MODULE_AUTHOR("Your Name");
MODULE_DESCRIPTION("A simple Linux driver for a GPIO interrupt");
MODULE_VERSION("0.1");
```

3.4.2 Dynamic allocation of work_struct at runtime

1. In **btn_irq_init()**
    a. **create_workqueue()** creates a workqueue
    b. **kmalloc()** allocates **work_struct, INIT_WORK()** initializes it. (The use of **GFP_ATOMIC** in **kmalloc()** means that sleep is not allowed. **GFP_KERNEL** means sleep is allowed.)
2. **queue_work()** puts the **work_struct** in the custom workqueue.
3. **work_struct** is freed using **kfree()** in **btn_irq_exit().**

workqueue.c

```
#include <linux/module.h>
#include <linux/gpio.h>
#include <linux/interrupt.h>
```

```c
#include <linux/kernel.h>
#include <linux/workqueue.h>
#include <linux/slab.h> // kmalloc()

#define button 3 // Example GPIO number
#define LED1 4
#define LED2 5
#define ON 0
#define OFF 1

static unsigned int irq_number;
unsigned int gpio_value = OFF;
static struct work_struct *work;
static struct workqueue_struct *workqueue;

static void bottom_half(struct work_struct *work) {
    gpio_value = gpio_get_value(LED1);
    gpio_set_value(LED1, !gpio_value);
    gpio_set_value(LED2, !gpio_value);
    // gpio_value != gpio_value;
    pr_info("Button pressed!\n");
}

// Interrupt handler function(top-half)
static irqreturn_t btn_irq_handler(int irq, void *dev_id)
{
    // Put the work in the custom workqueue
    queue_work(workqueue, work);
    return IRQ_HANDLED;
}

static int __init btn_irq_init(void)
{
    // Request GPIOs
    int result_btn, result1, result2, result_irq;
    result_btn = gpio_request(button, "sysfs");
    result1 = gpio_request(LED1, "sysfs");
    result2 = gpio_request(LED2, "sysfs");
    if (result_btn || result1 || result2) {
        pr_info("Cannot request the GPIO\n");
        return 0;
    }
    // Set the direction of button GPIO
    gpio_direction_input(button);
    // Create a workqueue
    workqueue = create_workqueue("own_wq");
    // Allocate work_struct
    work = (struct work_struct *)kmalloc(sizeof(struct work_struct), GFP_ATOMIC);
    // Initialize the work_struct using bottom_half()
    INIT_WORK(work, bottom_half);
    // Get the IRQ number for our GPIO
    irq_number = gpio_to_irq(button);
```

```
    if (irq_number < 0) {
        printk(KERN_INFO "GPIO to IRQ mapping failed\n");
        return irq_number;
    }
    // Request the IRQ line
    result_irq = request_irq(irq_number, btn_irq_handler, IRQF_TRIGGER_FALLING,
"btn_irq", NULL);
    if (result_irq) {
        printk(KERN_INFO "IRQ request failed\n");
        return result_irq;
    }
    return 0;
}

static void __exit btn_irq_exit(void) {
    free_irq(irq_number, NULL);
    gpio_free(button);
    gpio_free(LED1);
    gpio_free(LED2);
    destroy_workqueue(workqueue);
    // Deallocate the finished work
    kfree(work);
}

module_init(btn_irq_init);
module_exit(btn_irq_exit);

MODULE_LICENSE("GPL");
MODULE_AUTHOR("Your Name");
MODULE_DESCRIPTION("A simple Linux driver for a GPIO interrupt");
MODULE_VERSION("0.1");
```

3.5 Through threaded IRQ

This method registers an interrupt that passes its work to the bottom-half using threaded IRQ.

request_threaded_irq() in **btn_irq_init()** registers a threaded IRQ where:

1. **IRQ_WAKE_THREAD** in **btn_irq_handler()** calls the **bottom-half()** thread.
2. The thread returns IRQ_HANDLED that indicates that the processing is completed successfully,

threadedIRQ.c

```
#include <linux/module.h>
#include <linux/gpio.h>
#include <linux/interrupt.h>
#include <linux/kernel.h>
#include <linux/workqueue.h>
```

```
#include <linux/slab.h> // kmalloc()

#define button 3 // Example GPIO number
#define LED1 4
#define LED2 5
#define ON 0
#define OFF 1

static unsigned int irq_number;
unsigned int gpio_value = OFF;

static void bottom_half(int irq, void *dev_id){
    gpio_value = gpio_get_value(LED1);
    gpio_set_value(LED1, !gpio_value);
    gpio_set_value(LED2, !gpio_value);
    // gpio_value != gpio_value;
    pr_info("Button pressed!\n");
    return IRQ_HANDLED;
}

// Interrupt handler function(top-half)
static irqreturn_t btn_irq_handler(int irq, void *dev_id)
{
    return IRQ_WAKE_THREAD;
}

static int __init btn_irq_init(void)
{
    // Request GPIOs
    int result_btn, result1, result2, result_irq;
    result_btn = gpio_request(button, "sysfs");
    result1 = gpio_request(LED1, "sysfs");
    result2 = gpio_request(LED2, "sysfs");
    if (result_btn || result1 || result2) {
        pr_info("Cannot request the GPIO\n");
        return 0;
    }
    // Set the direction of button GPIO
    gpio_direction_input(button);
    // Get the IRQ number for our GPIO
    irq_number = gpio_to_irq(button);
    if (irq_number < 0) {
        printk(KERN_INFO "GPIO to IRQ mapping failed\n");
        return irq_number;
    }
    // Request the IRQ line
    result_irq = request_threaded_irq(irq_number, btn_irq_handler, bottom_half,
IRQF_TRIGGER_FALLING, "btn_irq", NULL);
    if (result_irq) {
        printk(KERN_INFO "IRQ request failed\n");
        return result_irq;
    }
```

```
    return 0;
}

static void __exit btn_irq_exit(void) {
    free_irq(irq_number, NULL);
    gpio_free(button);
    gpio_free(LED1);
    gpio_free(LED2);
}

module_init(btn_irq_init);
module_exit(btn_irq_exit);

MODULE_LICENSE("GPL");
MODULE_AUTHOR("Your Name");
MODULE_DESCRIPTION("A simple Linux driver for a GPIO interrupt");
MODULE_VERSION("0.1");
```

Reference

- https://embetronicx.com/tutorials/linux/device-drivers/tasklets-dynamic-method/
- https://embetronicx.com/tutorials/linux/device-drivers/softirq-in-linux-kernel/
- https://embetronicx.com/tutorials/linux/device-drivers/workqueue-in-linux-kernel/
- https://embetronicx.com/tutorials/linux/device-drivers/threaded-irq-in-linux-kernel/