(一) tasklet\_interrupt

当我们调用 **tasklet\_init(&my\_tasklet, my\_tasklet\_handler, 0)** 时，我们实际上初始化了名为 my\_tasklet 的 Tasklet 结构体，并指定了它的处理函数my\_tasklet\_handler。

1. &my\_tasklet：这是指向我们已经定义的 Tasklet 结构体的指针。我们通过这个指针来告诉 tasklet\_init 函数要初始化哪个 Tasklet。
2. my\_tasklet\_handler：这是我们定义的 Tasklet 处理函数。当 Tasklet 被调度执行时，内核会调用这个函数来处理实际的工作。
3. 0：这个参数是用来传递给 Tasklet 处理函数的额外数据。在这个例子中，我们没有使用这个参数，所以传递了一个零值。

**req\_ret = request\_irq(irq\_num, my\_interrupt\_handler, IRQF\_SHARED, devname, (void \*)&irq\_count);**

我们调用了 request\_irq 函数来请求一个中断，并将其与我们定义的中断处理函数 my\_interrupt\_handler 相关联。

1. irq\_num：这是我们之前定义的中断号，它是一个整数，用于标识中断的编号。
2. my\_interrupt\_handler：这是我们自定义的中断处理函数。当发生与中断号 irq\_num 相关的中断时，内核将调用这个函数来处理中断。
3. IRQF\_SHARED：这是一个标志，指定了中断的共享属性。当设备共享中断时，需要设置这个标志。在我们的例子中，我们使用了共享中断，因此传递了这个标志。
4. devname：这是设备的名称，用于标识请求中断的设备。
5. (void \*)&irq\_count：这个参数通常用于传递指向设备结构的指针。
6. request\_irq 函数的返回值被赋给了变量 req\_ret。如果 request\_irq 函数成功注册了中断处理函数，它会返回 0；如果注册失败，它会返回一个负数，代表错误代码。所以通过检查 req\_ret 的值，我们可以确定中断请求是否成功。

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

1. static irqreturn\_t: 这是函数的返回类型，表示它返回的是一个 IRQ 状态。在 Linux 内核中，irqreturn\_t 是一个枚举类型，它可以有三个可能的值：IRQ\_NONE、IRQ\_HANDLED 和 IRQ\_WAKE\_THREAD。在这个函数中，我们返回的是 IRQ\_HANDLED，表示我们已经处理了中断。
2. my\_interrupt\_handler(int irq, void \*dev\_id): 这是函数的名称和参数列表。它接受两个参数：irq 表示中断号，dev\_id 是一个指向设备标识符的指针。在这个例子中，我们没有使用 dev\_id。
3. tasklet\_schedule(&my\_tasklet): 这一行代码调度了一个 Tasklet，即 my\_tasklet。在中断发生时，我们调度了这个 Tasklet，以便稍后在 Tasklet 上下文中执行相关的工作。这是为了避免在中断上下文中执行过多的工作，从而降低系统的响应性。
4. return IRQ\_HANDLED;: 最后，我们返回了 IRQ\_HANDLED，表示我们已经处理了中断。

总的来说，my\_interrupt\_handler 函数负责处理中断，它调度了一个 Tasklet 来处理中断事件，并返回了相应的中断状态。

**static void my\_tasklet\_handler(unsigned long data)**

函数签名中的 unsigned long data 参数在这个例子中并没有被使用，因为 Tasklet 函数不需要额外的数据参数。

函数内部首先通过 spin\_lock 获取了自旋锁 irq\_count\_lock，这是为了确保对共享资源的互斥访问。然后，通过 irq\_count++ 将中断计数 irq\_count 增加了一次，表示发生了一次中断。接着，使用 printk 函数打印了一条消息，指示中断发生，并显示了当前的中断计数值。最后，通过 spin\_unlock 释放了自旋锁，允许其他任务继续访问共享资源。

1. #include <linux/init.h>
2. #include <linux/module.h>
3. #include <linux/kernel.h>
4. #include <linux/interrupt.h>
6. **static** **int** irq\_num = 0; //中断号
7. **static** **char** \*devname = NULL;  //设备名称
8. **static** **int** irq\_count = 0;   //中断计数
9. **int** req\_ret = 0;  // 中断请求返回值
10. **static** **struct** tasklet\_struct my\_tasklet;
11. DEFINE\_SPINLOCK(irq\_count\_lock);
13. module\_param(irq\_num, **int**, S\_IRUGO);
14. module\_param(devname, charp, S\_IRUGO);
16. // Tasklet 处理函数
17. **static** **void** my\_tasklet\_handler(unsigned **long** data)
18. {
19. spin\_lock(&irq\_count\_lock); // 获取自旋锁
20. irq\_count++;
21. printk(KERN\_INFO "Tasklet: Interrupt occurred. Count: %d\n", irq\_count);
22. spin\_unlock(&irq\_count\_lock); // 释放自旋锁
23. }
25. // 中断处理函数
26. **static** irqreturn\_t my\_interrupt\_handler(**int** irq, **void** \*dev\_id)
27. {
28. // 调度 tasklet
29. tasklet\_schedule(&my\_tasklet);
30. **return** IRQ\_HANDLED;
31. }
33. **static** **int** \_\_init interrupt\_module\_init(**void**)
34. {
35. // 初始化自旋锁
36. spin\_lock\_init(&irq\_count\_lock);
38. // 检查参数
39. **if** (!irq\_num || !devname) {
40. printk(KERN\_ERR "Missing parameters\n");
41. **return** -EINVAL;
42. }
44. printk(KERN\_INFO "Module starts...\n");
46. // 初始化 tasklet
47. tasklet\_init(&my\_tasklet, my\_tasklet\_handler, 0);
49. // 注册中断处理函数
50. req\_ret = request\_irq(irq\_num, my\_interrupt\_handler, IRQF\_SHARED, devname, (**void** \*)&irq\_count);
51. printk(KERN\_INFO "req\_ret is %d...\n", req\_ret);
52. **if** (req\_ret)
53. {
54. printk(KERN\_ERR "Failed to request IRQ:%d for device %s\n", irq\_num, devname);
55. **return** -EFAULT;
56. }
58. printk(KERN\_INFO "Request IRQ:%d success...\n", irq\_num);
59. **return** 0;
60. }
62. **static** **void** \_\_exit interrupt\_module\_exit(**void**)
63. {
64. // 卸载中断处理函数
65. free\_irq(irq\_num, (**void** \*)&irq\_num);
67. // 清理 tasklet
68. tasklet\_kill(&my\_tasklet);
70. printk(KERN\_INFO "Module exits...\n");
71. }
73. module\_init(interrupt\_module\_init);
74. module\_exit(interrupt\_module\_exit);
76. MODULE\_LICENSE("GPL");
77. MODULE\_AUTHOR("Your Name");
78. workqueue\_test.c
79. 首先定义了一些常量，如 DELAY\_SECONDS 和 TIMER\_INTERVAL，它们用于指定延迟工作的延迟时间和定时器的间隔时间。
80. 然后，声明了几个结构体和变量，用于管理工作队列、工作项、延迟工作和定时器。其中，my\_workqueue 是一个指向工作队列的指针，my\_work 和 my\_delayed\_work 分别是立即执行和延迟执行的工作项，my\_timer 是一个定时器。
81. 接下来，定义了三个处理函数：my\_work\_handler 用于处理立即执行的工作项，my\_delayed\_work\_handler 用于处理延迟执行的工作项，my\_timer\_handler 用于定时器的处理。
82. 在模块初始化函数 workqueue\_test\_init 中，代码完成了以下任务：初始化定时器，并设置定时器的处理函数为 my\_timer\_handler，然后启动定时器以定时触发处理函数。创建了一个名为 "my\_workqueue" 的工作队列。初始化并排队了立即执行的工作项 my\_work。初始化并排队了延迟执行的工作项 my\_delayed\_work。
83. 最后，在模块退出函数 workqueue\_test\_exit 中，代码执行以下操作：取消延迟执行的工作项。删除定时器。清空并销毁工作队列。

整个代码的目的是在加载模块时创建一个工作队列，并演示如何使用工作队列、延迟工作和定时器来执行异步任务。

1. #include <linux/init.h>
2. #include <linux/module.h>
3. #include <linux/kernel.h>
4. #include <linux/workqueue.h>
5. #include <linux/time.h>
7. #define DELAY\_SECONDS 10
8. #define TIMER\_INTERVAL (5 \* HZ)
10. **static** **struct** workqueue\_struct \*my\_workqueue;
11. **static** **struct** work\_struct my\_work;
12. **static** **struct** delayed\_work my\_delayed\_work;
13. **static** **struct** timer\_list my\_timer;
15. **struct** timeval tv;
16. **struct** **tm** **tm**;

19. // 立即执行的工作项处理函数
20. **static** **void** my\_work\_handler(**struct** work\_struct \*work)
21. {
22. do\_gettimeofday(&tv);
23. time\_to\_tm(tv.tv\_sec, 0, &**tm**);
24. printk(KERN\_INFO "正在执行【立即执行】的任务 当前时间： %ld-%d-%d %d:%d:%d\n" ,
25. (**long**)**tm**.tm\_year + 1900, **tm**.tm\_mon + 1, **tm**.tm\_mday,
26. **tm**.tm\_hour, **tm**.tm\_min, **tm**.tm\_sec);
27. }
29. // 延迟执行的工作项处理函数
30. **static** **void** my\_delayed\_work\_handler(**struct** work\_struct \*work)
31. {
32. do\_gettimeofday(&tv);
33. time\_to\_tm(tv.tv\_sec, 0, &**tm**);
34. printk(KERN\_INFO "正在执行【延迟执行】的任务 当前时间： %ld-%d-%d %d:%d:%d\n"  ,
35. (**long**)**tm**.tm\_year + 1900, **tm**.tm\_mon + 1, **tm**.tm\_mday,
36. **tm**.tm\_hour, **tm**.tm\_min, **tm**.tm\_sec);
37. }
39. // 定时器处理函数
40. **static** **void** my\_timer\_handler(**struct** timer\_list \*timer)
41. {

44. do\_gettimeofday(&tv);
45. time\_to\_tm(tv.tv\_sec, 0, &**tm**);
46. printk(KERN\_INFO "当前时间： %ld-%d-%d %d:%d:%d\n",
47. (**long**)**tm**.tm\_year + 1900, **tm**.tm\_mon + 1, **tm**.tm\_mday,
48. **tm**.tm\_hour, **tm**.tm\_min, **tm**.tm\_sec);
49. mod\_timer(&my\_timer, jiffies + TIMER\_INTERVAL);
50. }
52. // 模块加载函数
53. **static** **int** \_\_init workqueue\_test\_init(**void**)
54. {

57. // 初始化定时器
58. timer\_setup(&my\_timer, my\_timer\_handler, 0);
59. mod\_timer(&my\_timer, jiffies + TIMER\_INTERVAL);
61. // 创建工作队列
62. my\_workqueue = create\_workqueue("my\_workqueue");
63. **if** (!my\_workqueue) {
64. printk(KERN\_ERR "无法创建工作队列\n");
65. **return** -ENOMEM;
66. }
68. // 初始化并排队立即执行的工作项
69. INIT\_WORK(&my\_work, my\_work\_handler);
70. queue\_work(my\_workqueue, &my\_work);
72. // 初始化并排队延迟执行的工作项
73. INIT\_DELAYED\_WORK(&my\_delayed\_work, my\_delayed\_work\_handler);
74. queue\_delayed\_work(my\_workqueue, &my\_delayed\_work, DELAY\_SECONDS \* HZ);


78. **return** 0;
79. }
81. // 模块卸载函数
82. **static** **void** \_\_exit workqueue\_test\_exit(**void**)
83. {
84. // 停止延迟执行的工作项
85. cancel\_delayed\_work(&my\_delayed\_work);
87. // 停止定时器
88. del\_timer(&my\_timer);
90. // 刷新和销毁工作队列
91. flush\_workqueue(my\_workqueue);
92. destroy\_workqueue(my\_workqueue);
94. printk(KERN\_INFO "卸载工作队列测试模块\n");
95. }
97. module\_init(workqueue\_test\_init);
98. module\_exit(workqueue\_test\_exit);
100. MODULE\_LICENSE("GPL");