



湖南大学

HUNAN UNIVERSITY

操作系统实验报告

课 程 名 称： 操作系统

实验项目名称： 操作系统内核编程实验

专 业 班 级： 软件 2203

姓 名： 白旭

学 号： 202226010306

指 导 教 师： 周军海

完 成 时 间： 2024 年 5 月 16 日

信息科学与工程学院

实验题目：实验四 中断和异常

- 实验目的：
- 使用 tasklet 完成一个中断程序：
编写内核模块，对 31 号中断注册一个中断处理函数，打印出其调用的次数
把加载、卸载内核模块以 install/uninstall 写入 Makefile 文件中
 - 工作队列：
编写内核模块，分别发送一个实时任务（立即执行）和一个延迟任务(延后十秒)，观察它们的执行顺序
编写对应 Makefile 文件，并使用 make 编译上述内核模块
 - 思考题：
响应中断的时机在哪里？

- 实验环境：
- 华为 ESC 弹性云服务器
 - WinScp

- 实验内容及操作步骤：
1. 打开 ESC 弹性云服务器：



2. 使用 powershell 连接服务器与主机：

```

Welcome to Huawei Cloud Service

Last login: Wed May  8 22:35:58 2024 from 222.244.139.20

Welcome to 4.19.90-2110.8.0.0119.oe1.aarch64

System information as of time:  Thu May 16 20:33:35 CST 2024

System load:      0.62
Processes:        162
Memory used:      5.1%
Swap used:        0.0%
Usage On:         38%
IP address:       192.168.1.122
Users online:     1

[root@openeuler ~]# |
```

3. 编写代码

1) tasklet_interrupt.c

头文件、变量声明、模块参数与模块许可证

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

static int irq;                // Interrupt Request, 中断请求
static char *devname;          // 注册中断的设备名
static struct tasklet_struct mytasklet; // 声明tasklet结构体
static unsigned int req_ret = 0; // 用于计数中断调用次数

module_param(irq, int, 0644); // 用户向内核传递irq参数
module_param(devname, charp, 0644); // 用户向内核传递devname参数

MODULE_LICENSE("GPL");
```

首先定义了一个存储设备相关信息的结构体 `struct myirq`，并初始化了一个实例 `mydev`。然后定义了一个 `mytasklet_handler` 函数作为任务队列处理函数。接着是中断处理函数 `myirq_handler`，其中对中断计数进行递增并调用了任务队列 `mytasklet`

```
// 用于存储设备相关的信息
// 在注册中断处理程序时被传递给 request_irq 函数，用于标识设备并传递给中断处理程序
struct myirq
{
    int devid;
};

struct myirq mydev = {1900};

/*tasklet结构体的处理函数*/
// 每调用一次tasklet_schedule，则本函数会执行一次
// 内核会在适当的上下文中执行 mytasklet_handler 函数
// 不会立即执行，所以日志信息中没有
static void mytasklet_handler(unsigned long data)
{
    printk(KERN_INFO "=== tasklet is working...\n");
}

/*中断处理函数*/
// 绝大多数情况下，tasklet都是在中断处理函数中激活就绪的
static irqreturn_t myirq_handler(int irq, void *dev)
{
    req_ret++;
    printk(KERN_INFO "=== req_ret is %u\n", req_ret);
    // 调度API，把tasklet加入到就绪的列表中
    // 所有就绪的tasklet会被逐一从链表中取出运行（执行tasklet_struct中的callback或func）
    tasklet_schedule(&mytasklet);

    // IRQ_NONE：表示中断处理程序没有处理这个中断。
    // 通常用于共享中断的情况下，表明当前处理程序不是这个中断的处理者。
    // IRQ_HANDLED：表示中断处理程序成功处理了这个中断。
    return IRQ_HANDLED;
}
```

这段代码是内核模块的初始化和退出函数。在模块初始化函数 `myirq_init` 中，首先打印模块开始运行的信息和当前的中断处理次数。然后使用 `tasklet_init` 函数初始化了任务队列 `mytasklet`，并调用 `request_irq` 函数注册了中断处理函数 `myirq_handler`。注册成功后，打印注册成功的信息。在模块退出函数 `myirq_exit` 中，打印模块退出的信息，清除了任务队列 `mytasklet`，并使用 `free_irq` 函数注销了中断处理函数。最后打印注销成功的信息。

```
static int __init myirq_init(void)
{
    int ret;

    printk(KERN_INFO "=== Module starts...\n");
    printk(KERN_INFO "=== req_ret is %u\n", req_ret);

    // 初始化tasklet,指定处理函数:mytasklet_handler
    tasklet_init(&mytasklet, mytasklet_handler, 0);

    // request_irq 函数向内核注册一个中断处理程序,并指定 IRQ 号和处理程序
    // 每当指定的中断发生时,内核会调用注册的中断处理程序 myirq_handler
    // IRQF_SHARED,表示多个设备共享中断
    ret = request_irq(irq, myirq_handler, IRQF_SHARED, devname, &mydev);
    if (ret)
    {
        printk(KERN_ERR "=== %s request IRQ:%d failed with %d\n", devname, irq, ret);
        return ret;
    }
    printk(KERN_INFO "=== %s request IRQ:%d success...\n", devname, irq);

    // 模拟中断事件发生
    // printk(KERN_INFO "=== Simulating interrupt event...\n");

    return 0;
}

static void __exit myirq_exit(void)
{
    printk(KERN_INFO "=== Module exits...\n");
    // 清除一个处于就绪状态(已schedule,正在等待运行)的tasklet
    tasklet_kill(&mytasklet);
    // 注销中断处理函数时
    free_irq(irq, &mydev);
    printk(KERN_INFO "=== %s request IRQ:%d leaving success...\n", devname, irq);
}

module_init(myirq_init);
module_exit(myirq_exit);
```

其中 `tasklet_struct` 的具体实现：

```
struct tasklet_struct
{
    struct tasklet_struct *next; // 用于tasklet单链表
    unsigned long state;         // 状态,用于保存“就绪标志”等
    atomic_t count;              // 引用计数,为0时表示可调度,非0则不可调度
    bool use_callback;           // 选择任务处理函数的类型
    union
    {
        void (*func)(unsigned long data); // 旧格式的任务处理函数
        void (*callback)(struct tasklet_struct *t); // 新格式的处理函数
    };
    unsigned long data; // 旧格式函数的回传参数
};
```

函数 tasklet_init() 的具体实现:

```
void tasklet_init(struct tasklet_struct *t, void (*func)(unsigned long),
unsigned long data)
{
    t->next = NULL;
    t->state = 0;
    atomicset(&t->count, 0);
    t->func = func;
    t->use_callback = false;
    t->data = data;
}
```

Makefile 文件:

按题目要求, 把加载、卸载内核模块以 install/uninstall 写入 Makefile 文件中

执行时, 只需要输入 make install/uninstall 即可加载或卸载模块

```
ifneq ($(KERNELRELEASE),)
    obj-m := tasklet_interrupt.o
else
    KERNELDIR ?= /usr/lib/modules/$(shell uname -r)/build
    PWD := $(shell pwd)

default:
    $(MAKE) -C $(KERNELDIR) M=$(PWD) modules

# 定义内核模块参数
irq ?= 31
devname ?= "tasklet_dev"

# 安装内核模块, 传入参数
install: default
    sudo insmod tasklet_interrupt.ko irq=$(irq) devname=$(devname)

# 卸载内核模块
uninstall:
    sudo rmmod tasklet_interrupt

# 清理编译生成的文件
clean:
    -rm *.mod.c *.o *.order *.symvers *.ko

endif
# 使用 .PHONY 声明 clean、install 和 uninstall 为伪目标
.PHONY: clean install uninstall
```

2) workqueue_test.c

模块许可证、作者和描述信息，以及一个模块参数 `times` 用于指定打印时间的次数。还定义了工作队列指针和两个延迟工作结构体变量

```
#include <linux/module.h>
#include <linux/kernel.h>
#include <linux/workqueue.h>
#include <linux/timer.h>
#include <linux/timex.h>
#include <linux/rtc.h>
#include <linux/delay.h>

MODULE_LICENSE("GPL");
MODULE_AUTHOR("BX");
MODULE_DESCRIPTION(
    "Use workqueue to print the time periodically");

int times = 0;
module_param(times, int, 0644);
// 用户向内核传递times参数

static struct workqueue_struct *queue = NULL;
static struct delayed_work work1;
static struct delayed_work work2;
static int i = 0;
```

这段代码是一个立即执行任务的处理函数。它包含了获取当前时间的功能，使用了 `do_gettimeofday` 函数获取当前时间并存储在 `txc.time` 中，然后将秒数转换为人类可读时间并存储在 `rtc_time` 结构体 `tm` 中。接着，打印了当前时间，并递增了一个计数变量 `i`。最后，如果 `i` 小于指定的次数 `times`，则调度了一个延迟工作 `work1`，延迟时间为 5 秒

```
// 立即执行任务的处理函数
static void work_handler(struct work_struct *work)
{
    // timex 是一个包含时间相关信息的结构体
    struct timex txc;
    // rtc_time 是一个用来表示人类可读时间的结构体，包含年、月、日、时、分、秒等信息
    struct rtc_time tm;
    // 获取当前时间，并将其存储在 txc.time 中
    do_gettimeofday(&(txc.time));
    // 将 txc.time.tv_sec (即当前时间的秒数) 转换为 rtc_time 结构体 tm
    rtc_time_to_tm(txc.time.tv_sec, &tm);

    printk(KERN_INFO "%d:\n", i);
    printk(KERN_INFO "Immediate Work: Current time: %d-%02d-%02d %02d:%02d:%02d\n",
        tm.tm_year + 1900, tm.tm_mon + 1, tm.tm_mday,
        tm.tm_hour, tm.tm_min, tm.tm_sec);

    i++;
    if (i < times)
    {
        queue_delayed_work(queue, &work1, 5 * HZ);
    }
}
```

这段代码是一个延迟任务的处理函数。它与前面的立即执行任务的处理函数类似，同样包含了获取当前时间的功能，使用了 `do_gettimeofday` 函数获取当前时间并存储在 `txc.time` 中，然后将秒数转换为可读时间并存储在 `rtc_time` 结构体 `tm` 中。接着，打印了当前时间。这个处理函数被用作延迟工作队列中的任务，在一段时间后被执行。

```
// 延迟任务的处理函数
static void work_handler_delay(struct work_struct *work)
{
    struct timex txc;
    struct rtc_time tm;
    do_gettimeofday(&(txc.time));
    rtc_time_to_tm(txc.time.tv_sec, &tm);

    printk(KERN_INFO "Delayed Work: Current time: %d-%02d-%02d %02d:%02d:%02d\n",
           tm.tm_year + 1900, tm.tm_mon + 1, tm.tm_mday,
           tm.tm_hour, tm.tm_min, tm.tm_sec);
}
```

模块初始化与退出函数

```
static int __init init_workqueue(void)
{
    printk(KERN_INFO "Initializing Workqueue Module\n");

    // 创建工作队列
    queue = create_workqueue("my_workqueue");
    if (!queue)
    {
        printk(KERN_ERR "Failed to create workqueue\n");
        return -ENOMEM;
    }

    // 初始化工作
    INIT_DELAYED_WORK(&work1, work_handler);
    INIT_DELAYED_WORK(&work2, work_handler_delay);

    // 调度立即执行的工作
    queue_delayed_work(queue, &work1, 0);

    // 调度延迟执行的工作，延迟10秒
    queue_delayed_work(queue, &work2, 10 * HZ);

    return 0;
}

static void __exit exit_workqueue(void)
{
    printk(KERN_INFO "Exiting Workqueue Module\n");

    // 取消工作
    cancel_delayed_work_sync(&work1);
    cancel_delayed_work_sync(&work2);

    // 销毁工作队列
    if (queue)
    {
        destroy_workqueue(queue);
    }
}

module_init(init_workqueue);
module_exit(exit_workqueue);
```

4. 模块功能验证

1) tasklet_interrupt

```
[root@openeuler task1]# make
make -C /usr/lib/modules/4.19.90-2110.8.0.0119.oe1.aarch64/build M=/root/interrupt/exp1/task1 modules
make[1]: Entering directory '/usr/src/kernels/4.19.90-2110.8.0.0119.oe1.aarch64'
CC [M] /root/interrupt/exp1/task1/tasklet_interrupt.o
Building modules, stage 2.
MODPOST 1 modules
CC /root/interrupt/exp1/task1/tasklet_interrupt.mod.o
LD [M] /root/interrupt/exp1/task1/tasklet_interrupt.ko
make[1]: Leaving directory '/usr/src/kernels/4.19.90-2110.8.0.0119.oe1.aarch64'
[root@openeuler task1]# make install
make -C /usr/lib/modules/4.19.90-2110.8.0.0119.oe1.aarch64/build M=/root/interrupt/exp1/task1 modules
make[1]: Entering directory '/usr/src/kernels/4.19.90-2110.8.0.0119.oe1.aarch64'
Building modules, stage 2.
MODPOST 1 modules
make[1]: Leaving directory '/usr/src/kernels/4.19.90-2110.8.0.0119.oe1.aarch64'
sudo insmod tasklet_interrupt.ko irq=31 devname="tasklet_dev"
[root@openeuler task1]# make uninstall
sudo rmmod tasklet_interrupt
[root@openeuler task1]# dmesg | tail -n5
[ 5214.200030] === Module starts...
[ 5214.200262] === req_ret is 0
[ 5214.200459] === tasklet_dev request IRQ:31 success...
[ 5217.124413] === Module exits...
[ 5217.124660] === tasklet_dev request IRQ:31 leaving success...
```

2) workqueue_test

```
[root@openeuler task1]# dmesg | tail -n13
[ 4321.223910] Initializing Workqueue Module
[ 4321.224369] 0:
[ 4321.224483] Immediate Work: Current time: 2024-05-24 09:13:41
[ 4326.336540] 1:
[ 4326.336676] Immediate Work: Current time: 2024-05-24 09:13:46
[ 4331.456516] 2:
[ 4331.456649] Immediate Work: Current time: 2024-05-24 09:13:51
[ 4331.457005] Delayed Work: Current time: 2024-05-24 09:13:51
[ 4336.576501] 3:
[ 4336.576714] Immediate Work: Current time: 2024-05-24 09:13:56
[ 4341.696471] 4:
[ 4341.696610] Immediate Work: Current time: 2024-05-24 09:14:02
[ 4364.855374] Exiting Workqueue Module
```

实验结果及分析:

1. tasklet_interrupt 执行流程总结:

1) 加载模块:

当内核加载该模块时, myirq_init 函数被调用。

在 myirq_init 函数中:

初始化了一个 tasklet_struct 结构体 mytasklet, 并指定了处理函数为 mytasklet_handler。

使用 request_irq 函数注册了一个中断处理函数 myirq_handler, 并指定了中断号、设备名、以及一个结构体 mydev 作为参数。

如果注册成功, 打印注册成功的信息; 如果失败, 则打印注册失败的信息。

2) 中断处理:

当系统中断事件发生时, 内核会调用注册的中断处理函数 myirq_handler。

在 myirq_handler 函数中:

递增了一个用于计数的变量 req_ret, 并打印了其值。

调用 tasklet_schedule 函数, 将 mytasklet 添加到就绪列表中, 以便稍后执行。

3) 任务队列执行:

任务队列 `mytasklet` 被调度后, 内核会在适当的上下文中执行 `mytasklet_handler` 函数。

在 `mytasklet_handler` 函数中, 打印了一条消息, 表示任务队列正在工作。

4) 卸载模块:

当内核卸载该模块时, `myirq_exit` 函数被调用。

在 `myirq_exit` 函数中:

调用 `tasklet_kill` 函数, 清除处于就绪状态的任务队列 `mytasklet`。

使用 `free_irq` 函数注销了中断处理函数, 并释放了相应的资源。

打印模块退出的信息。

通过以上步骤, 该模块实现了中断处理和任务队列的功能, 用于处理系统中断事件, 并在中断上下文之外执行延迟的工作。

2. **workqueue_test 执行流程总结:**

1) 加载模块:

创建工作队列。

初始化并调度 `work1` 立即执行。

初始化并调度 `work2` 延迟 10 秒执行。

2) 执行 `work1`:

获取并打印当前时间。

递增计数器 `i`。

如果 `i` 小于 `times`, 则调度 `work1` 延迟 5 秒再次执行。

3) 执行 `work2`:

获取并打印当前时间。

4) 卸载模块:

取消所有延迟工作。

销毁工作队列。

通过这个过程, `work1` 将每 5 秒执行一次, 总共执行 5 次, 而 `work2` 将在模块加载后 10 秒执行一次。

收获与体会:

1. 掌握了如何在 `makefile` 文件中写入加载与卸载模块
2. 掌握了 linux 内核的 `tasklet` 中断程序和工作队列实现延迟
3. 了解了中断的工作机制

思考题：

1. 响应中断的时机在哪里？

中断的响应时机是在 `myirq_handler` 函数中。当系统中断事件发生时，内核会调用注册的中断处理函数 `myirq_handler`。在这个函数中，中断被处理，执行了一些特定的操作（例如增加计数器 `req_ret`，然后调度任务队列 `mytasklet`）。因此，中断的响应和处理发生在 `myirq_handler` 函数内部

2. 描述使用 `tasklet` 完成中断程序的基本步骤，并解释 `tasklet` 在中断处理中的作用。

使用 `tasklet` 完成中断程序的基本步骤如下：

- 1) 定义中断处理函数：首先，需要定义一个中断处理函数，用于处理中断事件。这个函数会被注册到特定的中断号上，以便在中断事件发生时被调用快速分配：由于分配器从链表的开头开始查找空闲块，因此可以快速找到满足要求的内存块进行分配
- 2) 初始化 `tasklet`：在模块初始化阶段，需要初始化一个或多个 `tasklet`。这通常通过调用 `tasklet_init` 函数来完成
- 3) 定义 `tasklet` 处理函数：定义一个函数，作为 `tasklet` 的处理函数。这个函数会在任务队列中执行，并负责处理由中断事件触发的工作
- 4) 在中断处理函数中调度 `tasklet`：在中断处理函数中，通过调用 `tasklet_schedule` 函数来调度 `tasklet`。这样，当中断事件发生时，`tasklet` 就会被添加到内核的任务队列中，等待被执行
- 5) 在模块退出阶段清理 `tasklet`：在模块退出时，需要确保清理已初始化的 `tasklet`，以释放资源并避免内存泄漏。这通常通过调用 `tasklet_kill` 函数来完成

`tasklet` 在中断处理中的作用是提供一种轻量级的下半部处理机制。在中断处理函数中，通常只做一些必要的工作，然后立即返回，以尽快释放中断服务例程的资源，以便内核能够响应其他中断。然而，有时候我们可能需要执行一些更加复杂或耗时的操作，这时就可以使用 `tasklet`。通过将这些操作放入 `tasklet` 的处理函数中，在中断上下文之外执行，避免了在中断处理函数中执行耗时操作可能导致的延迟和竞争条件。这样可以提高系统的响应性能和稳定性

3. 解释工作队列（`workqueue`）的概念，并举例说明其在内核中的一个应用场景。

工作队列（`workqueue`）是 Linux 内核中的一种机制，用于在内核上下文之外执行延迟的工作。它允许在系统中的后台线程中执行长时间运行的任务，而不会阻塞当前正在执行的进程或中断处理程序。工作队列是一种异步执行任务的方式，使得内核可以并行地处理多个任务，提高系

统的响应性能和并发处理能力。

工作队列的一个常见应用场景是执行后台任务：在文件系统中，当需要将缓存中的数据写入磁盘时，可以将文件系统刷新任务放入工作队列中延迟执行。这样可以避免在写入大量数据时阻塞当前进程，提高系统的响应速度

4. 解释内核如何处理中断，以及中断处理程序中应该注意哪些问题？

在 Linux 内核中，中断是一种异步事件，用于处理来自硬件设备的通知或请求。当硬件设备触发中断时，CPU 会立即中断当前正在执行的任务，并跳转到相应的中断处理程序，执行与该中断相关的处理逻辑。中断处理程序负责处理中断事件，并可能执行一些特定的操作，如读取数据、写入数据、更新状态等。在中断处理程序完成后，CPU 会返回到原来的上下文中，继续执行之前的任务。

在编写中断处理程序时，需要注意以下几个问题：

- 1) 响应时间和效率：中断处理程序应尽可能地快速执行，以确保及时响应中断事件。长时间的中断处理可能会影响系统的响应性能和实时性
- 2) 共享资源访问：中断处理程序可能会访问共享资源，如全局变量、内核数据结构等。因此，需要考虑对共享资源的访问同步，以避免数据竞争和并发访问的问题
- 3) 禁用和启用中断：在中断处理程序中，可能需要禁用或启用其他中断。在禁用中断期间，需要确保不会产生死锁或其他不良影响，否则可能导致系统无响应或异常
- 4) 中断嵌套：内核支持中断嵌套，即在一个中断处理程序中可以触发另一个中断。在处理中断嵌套时，需要注意避免死锁和递归调用等问题，以确保系统的稳定性和可靠性
- 5) 异常处理：中断处理程序可能会发生异常或错误，如空指针访问、越界访问等。因此，需要在中断处理程序中进行适当的错误处理和异常处理，以防止系统崩溃或数据损坏

实验
成绩