在前一章也提到过，之所以中断会分成上下两部分，是由于中断对时限的要求非常高，需要尽快的响应硬件。

**主要内容：**

* 中断下半部处理
* 实现中断下半部的机制
* 总结中断下半部的实现
* 中断实现示例

### **1. 中断下半部处理**

那么对于一个中断，如何划分上下两部分呢？哪些处理放在上半部，哪些处理放在下半部？

这里有一些经验可供借鉴：

1. 如果一个任务对时间十分敏感，将其放在上半部
2. 如果一个任务和硬件有关，将其放在上半部
3. 如果一个任务要保证不被其他中断打断，将其放在上半部
4. 其他所有任务，考虑放在下半部

### **2. 实现中断下半部的机制**

实现下半部的方法很多，随着内核的发展，产生了一些新的方法，也淘汰了一些旧方法。

目前使用最多的是以下3中方法

2.1 软中断

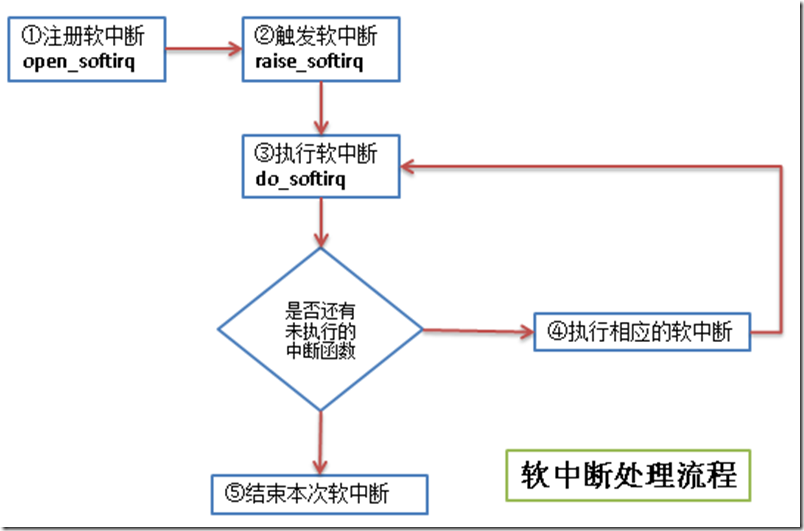
2.2 tasklet

2.3 工作队列

#### **2.1 软中断**

软中断的代码在：kernel/softirq.c

软中断的流程如下：

[](https://images0.cnblogs.com/blog/83005/201304/23111716-dc0f4c90540c48569dd18f835b2d8af9.png)

流程图中几个步骤的说明：

① 注册软中断的函数 open\_softirq参见 kernel/softirq.c文件)

/\*

 \* 将软中断类型和软中断处理函数加入到软中断序列中

 \* @nr                             - 软中断类型

 \* @(\*action)(struct softirq\_action \*) - 软中断处理的函数指针

 \*/

void open\_softirq(int nr, void (\*action)(struct softirq\_action \*))

**软中断类型：**目前有10个，其定义在 include/linux/interrupt.h 文件中：

enum

{

HI\_SOFTIRQ=0,

TIMER\_SOFTIRQ,

NET\_TX\_SOFTIRQ,

NET\_RX\_SOFTIRQ,

BLOCK\_SOFTIRQ,

BLOCK\_IOPOLL\_SOFTIRQ,

TASKLET\_SOFTIRQ,

SCHED\_SOFTIRQ,

HRTIMER\_SOFTIRQ,

RCU\_SOFTIRQ, /\* Preferable RCU should always be the last softirq \*/

NR\_SOFTIRQS

};

**struct softirq\_action：** 定义也在 include/linux/interrupt.h 文件中

/\*

 \* 这个结构体的字段是个函数指针，字段名称是action

 \* 函数指针的返回指是void型

 \* 函数指针的参数是 struct softirq\_action 的地址，其实就是指向 softirq\_vec 中的某一项

 \*     如果 open\_softirq 是这样调用的： open\_softirq(NET\_TX\_SOFTIRQ, my\_tx\_action);

 \*     那么 my\_tx\_action 的参数就是 softirq\_vec[NET\_TX\_SOFTIRQ]的地址

 \*/

struct softirq\_action

{

    void (\*action)(struct softirq\_action \*);

};

② 触发软中断的函数 raise\_softirq 参见 kernel/softirq.c文件

/\*

 \* 触发某个中断类型的软中断

 \* @nr - 被触发的中断类型

 \* 从函数中可以看出，在处理软中断前后有保存和恢复寄存器的操作

 \*/

void raise\_softirq(unsigned int nr)

③ 执行软中断 do\_softirq 参见 kernel/softirq.c文件

asmlinkage void do\_softirq(void)

④ 执行相应的软中断 - 执行自己写的中断处理

linux中，执行软中断有专门的内核线程，每个处理器对应一个线程，名称ksoftirqd/n (n对应处理器号)

通过top命令查看我的单核虚拟机，CentOS系统中的ksoftirqd线程如下：

[root@vbox ~]# top | grep ksoftirq

4 root 20 0 0 0 0 S 0.0 0.0 0:00.02 ksoftirqd/0

#### **2.2 tasklet**

除了对性能要求特别高的情况，一般建议使用tasklet来实现自己的中断。

tasklet对应的结构体在 <linux/interrupt.h> 中

struct tasklet\_struct

{

    struct tasklet\_struct \*next; /\* 链表中的下一个tasklet \*/

    unsigned long state;         /\* tasklet状态 \*/

    atomic\_t count;              /\* 引用计数器 \*/

    void (\*func)(unsigned long); /\* tasklet处理函数 \*/

    unsigned long data;          /\* tasklet处理函数的参数 \*/

};

tasklet状态只有3种值：

值 0 表示该tasklet没有被调度

值 TASKLET\_STATE\_SCHED 表示该tasklet已经被调度

值 TASKLET\_STATE\_RUN 表示该tasklet已经运行

引用计数器count 的值不为0，表示该tasklet被禁止。

tasklet使用流程如下：

1. 声明tasklet (参见<linux/interrupt.h>)

/\* 静态声明一个tasklet \*/

#define DECLARE\_TASKLET(name, func, data) \

    struct tasklet\_struct name = { NULL, 0, ATOMIC\_INIT(0), func, data }

#define DECLARE\_TASKLET\_DISABLED(name, func, data) \

struct tasklet\_struct name = { NULL, 0, ATOMIC\_INIT(1), func, data }

/\* 动态声明一个tasklet 传递一个tasklet\_struct指针给初始化函数 \*/

extern void tasklet\_init(struct tasklet\_struct \*t,

             void (\*func)(unsigned long), unsigned long data);

2. 编写处理程序

参照tasklet处理函数的原型来写自己的处理逻辑

void tasklet\_handler(unsigned long date)

3. 调度tasklet

中断的上半部处理完后调度tasklet，在适当时候进行下半部的处理

tasklet\_schedule(&my\_tasklet) /\* my\_tasklet就是之前声明的tasklet\_struct \*/

#### **2.3 工作队列**

工作队列子系统是一个用于创建内核线程的接口，通过它可以创建一个工作者线程来专门处理中断的下半部工作。

工作队列和tasklet不一样，不是基于软中断来实现的。

缺省的工作者线程名称是 events/n (n对应处理器号)。

通过top命令查看我的单核虚拟机，CentOS系统中的events线程如下：

[root@vbox ~]# top | grep event

7 root 20 0 0 0 0 S 0.0 0.0 0:03.71 events/0

工作队列主要用到下面3个结构体，弄懂了这3个结构体的关系，也就知道工作队列的处理流程了。

/\* 在 include/linux/workqueue.h 文件中定义 \*/

struct work\_struct

{

    atomic\_long\_t data;       /\* 这个并不是处理函数的参数，而是表示此work是否pending等状态的flag \*/

#define WORK\_STRUCT\_PENDING 0 /\* T if work item pending execution \*/

#define WORK\_STRUCT\_FLAG\_MASK(3UL)

#define WORK\_STRUCT\_WQ\_DATA\_MASK(~WORK\_STRUCT\_FLAG\_MASK) struct list\_head entry; /\* 中断下半部处理函数的链表 \*/

    work\_func\_t func;                                                            /\* 处理中断下半部工作的函数 \*/

#ifdef CONFIG\_LOCKDEP

    struct lockdep\_map lockdep\_map;

#endif

};

/\* 在 kernel/workqueue.c文件中定义

 \* 每个工作者线程对应一个 cpu\_workqueue\_struct ，其中包含要处理的工作的链表

 \* (即 work\_struct 的链表，当此链表不空时，唤醒工作者线程来进行处理)

 \*/

struct cpu\_workqueue\_struct

{

spinlock\_t lock; /\* 锁保护这种结构 \*/

    struct list\_head worklist; /\* 工作队列头节点 \*/

    wait\_queue\_head\_t more\_work;

struct work\_struct \*current\_work;

    struct workqueue\_struct \*wq; /\* 关联工作队列结构 \*/

    struct task\_struct \*thread;  /\* 关联线程 \*/

} \_\_\_\_cacheline\_aligned;

/\* 也是在 kernel/workqueue.c 文件中定义的

 \* 每个 workqueue\_struct 表示一种工作者类型，系统默认的就是 events 工作者类型

 \* 每个工作者类型一般对应n个工作者线程，n就是处理器的个数

 \*/

struct workqueue\_struct

{

    struct cpu\_workqueue\_struct \*cpu\_wq; /\* 工作者线程 \*/

    struct list\_head list;

    const char \*name;

    int singlethread;

    int freezeable; /\* Freeze threads during suspend \*/

    int rt;

#ifdef CONFIG\_LOCKDEP

    struct lockdep\_map lockdep\_map;

#endif

};

使用工作者队列的方法见下图：

[](https://images0.cnblogs.com/blog/83005/201304/23111723-ae64659dc99a4c708c62357e5c403b51.png)

① 创建推后执行的工作 - 有静态创建和动态创建2种方法

/\* 静态创建一个work\_struct

\* @n - work\_struct结构体，不用事先定义

\* @f - 下半部处理函数

\*/

#define DECLARE\_WORK(n, f) \

struct work\_struct n = \_\_WORK\_INITIALIZER(n, f)

/\* 动态创建一个 work\_struct

\* @\_work - 已经定义好的一个 work\_struct

\* @\_func - 下半部处理函数

\*/

#ifdef CONFIG\_LOCKDEP

#define INIT\_WORK(\_work, \_func) \

do { \

static struct lock\_class\_key \_\_key; \

\

(\_work)->data = (atomic\_long\_t) WORK\_DATA\_INIT(); \

lockdep\_init\_map(&(\_work)->lockdep\_map, #\_work, &\_\_key, 0);\

INIT\_LIST\_HEAD(&(\_work)->entry); \

PREPARE\_WORK((\_work), (\_func)); \

} while (0)#else#define INIT\_WORK(\_work, \_func) \

do { \

(\_work)->data = (atomic\_long\_t) WORK\_DATA\_INIT(); \

INIT\_LIST\_HEAD(&(\_work)->entry); \

PREPARE\_WORK((\_work), (\_func)); \

} while (0)#endif

工作队列处理函数的原型：

typedef void (\*work\_func\_t)(struct work\_struct \*work);

② 刷新现有的工作，这个步骤不是必须的，可以直接从第①步直接进入第③步

   刷新现有工作的意思就是在追加新的工作之前，保证队列中的已有工作已经执行完了。

/\* 刷新系统默认的队列，即 events 队列 \*/

void flush\_scheduled\_work(void);

/\* 刷新用户自定义的队列

\* @wq - 用户自定义的队列

\*/

void flush\_workqueue(struct workqueue\_struct \*wq);

③ 调度工作 - 调度新定义的工作，使之处于等待处理器执行的状态

/\* 调度第一步中新定义的工作，在系统默认的工作者线程中执行此工作

\* @work - 第一步中定义的工作

\*/

schedule\_work(struct work\_struct \*work);

/\* 调度第一步中新定义的工作，在系统默认的工作者线程中执行此工作

\* @work - 第一步中定义的工作

\* @delay - 延迟的时钟节拍

\*/

int schedule\_delayed\_work(struct delayed\_work \*work, unsigned long delay);

/\* 调度第一步中新定义的工作，在用户自定义的工作者线程中执行此工作

\* @wq - 用户自定义的工作队列类型

\* @work - 第一步中定义的工作

\*/

int queue\_work(struct workqueue\_struct \*wq, struct work\_struct \*work);

/\* 调度第一步中新定义的工作，在用户自定义的工作者线程中执行此工作

\* @wq - 用户自定义的工作队列类型

\* @work - 第一步中定义的工作

\* @delay - 延迟的时钟节拍

\*/

int queue\_delayed\_work(struct workqueue\_struct \*wq,

struct delayed\_work \*work, unsigned long delay);

### **3. 总结中断下半部的实现**

下面对实现中断下半部工作的3种机制进行总结，便于在实际使用中决定使用哪种机制

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **下半部机制** | **上下文** | **复杂度** | **执行性能** | **顺序执行保障** |
| 软中断 | 中断 | 高 (需要自己确保软中断的执行顺序及锁机制) | 好 | 没有 |
| tasklet | 中断 | 中 (提供了简单的接口来使用软中断) | 中 | 同类型不能同时执行 |
| 工作队列 | 进程 | 低 (在进程上下文中运行，与写用户程序差不多) | 差 | 没有 |

### **4. 中断实现示例**

#### **4.1 软中断的实现**

本来想用内核模块的方法来测试一下软中断的流程，但是编译时发现软中断注册函数(open\_softirq)和触发函数(raise\_softirq)

并没有用EXPORT\_SYMBOL导出，所以自定义的内核模块中无法使用。

测试的代码如下：

#include <linux/init.h>

#include <linux/slab.h>

#include <linux/module.h>

#include <linux/kernel.h>

#include <linux/time.h>

#include <linux/string.h>

#include <linux/interrupt.h>

#include "kn\_common.h"

MODULE\_LICENSE("Dual BSD/GPL");

static void my\_softirq\_func(struct softirq\_action \*);

static int testsoftirq\_init(void)

{

    // 注册softirq，这里注册的是定时器的下半部

open\_softirq(TIMER\_SOFTIRQ, my\_softirq\_func);

    // 触发softirq

raise\_softirq(TIMER\_SOFTIRQ);

    return 0;

}

static void testsoftirq\_exit(void)

{

    printk(KERN\_ALERT "\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\n");

    printk(KERN\_ALERT "testrbtree is exited!\n");

    printk(KERN\_ALERT "\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\n");

}

static void my\_softirq\_func(struct softirq\_action \*act)

{

    printk(KERN\_ALERT "=========================\n");

    printk(KERN\_ALERT "my softirq function is been called!....\n");

    printk(KERN\_ALERT "=========================\n");

}

module\_init(testsoftirq\_init);

module\_exit(testsoftirq\_exit);

由于内核没有用**EXPORT\_SYMBOL**导出**open\_softirq**和**raise\_softirq**函数，所以编译时有如下警告：

WARNING: "open\_softirq" [/root/chap08/mysoftirq.ko] undefined!

WARNING: "raise\_softirq" [/root/chap08/mysoftirq.ko] undefined!

**注：**编译用的系统时centos6.3 (uname -r结果 - 2.6.32-279.el6.x86\_64)

没办法，只能尝试修改内核代码(将open\_softirq和raise\_softirq用EXPORT\_SYMBOL导出)，再重新编译内核，然后再尝试能否测试软中断。

主要修改2个文件，(既然要修改代码，干脆加了一种软中断类型)：

/\* 修改 kernel/softirq.c \*/// ... 略 ...

char \*softirq\_to\_name[NR\_SOFTIRQS] = {

"HI", "TIMER", "NET\_TX", "NET\_RX", "BLOCK", "BLOCK\_IOPOLL",

"TASKLET", "SCHED", "HRTIMER", "RCU", "WYB"

}; /\* 追加了一种新的softirq，即 "WYB"，我名字的缩写 ^\_^ \*/

// ... 略 ...

void raise\_softirq(unsigned int nr)

{

unsigned long flags;

local\_irq\_save(flags);

raise\_softirq\_irqoff(nr);

local\_irq\_restore(flags);

}

EXPORT\_SYMBOL(raise\_softirq); /\* 追加的代码 \*/

void open\_softirq(int nr, void (\*action)(struct softirq\_action \*))

{

softirq\_vec[nr].action = action;

}

EXPORT\_SYMBOL(open\_softirq); /\* 追加的代码 \*/

// ... 略 ...

/\* 还修改了 include/linux/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,

RCU\_SOFTIRQ, /\* Preferable RCU should always be the last softirq \*/

WYB\_SOFTIRQS, /\* 追加的一种中断类型 \*/

NR\_SOFTIRQS

};

重新编译内核后，在新的内核上再次实验软中断代码：

(编译内核方法参见：[《Linux内核设计与实现》读书笔记（五）- 系统调用](http://www.cnblogs.com/wang_yb/archive/2012/09/17/2688263.html" \t "https://www.cnblogs.com/wang_yb/archive/2013/04/23/_blank) 3.3节)

测试软中断的代码：testsoftirq.c

#include <linux/interrupt.h>

#include "kn\_common.h"

MODULE\_LICENSE("Dual BSD/GPL");

static void my\_softirq\_func(struct softirq\_action\*);

static int testsoftirq\_init(void)

{

    printk(KERN\_ALERT "interrupt's top half!\n");

    // 注册softirq，这里注册的是自定义的软中断类型

    open\_softirq(WYB\_SOFTIRQS, my\_softirq\_func);

    // 触发softirq

raise\_softirq(WYB\_SOFTIRQS);

    return 0;

}

static void testsoftirq\_exit(void)

{

    printk(KERN\_ALERT "\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\n");

    print\_current\_time(0);

    printk(KERN\_ALERT "testsoftirq is exited!\n");

    printk(KERN\_ALERT "\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\n");

}

static void my\_softirq\_func(struct softirq\_action\* act)

{

    printk(KERN\_ALERT "=========================\n");

    print\_current\_time(0);

    printk(KERN\_ALERT "my softirq function is been called!....\n");

    printk(KERN\_ALERT "=========================\n");

}

module\_init(testsoftirq\_init);

module\_exit(testsoftirq\_exit);

Makefile：

obj-m += mysoftirq.o

mysoftirq-objs := testsoftirq.o kn\_common.o

#generate the path

CURRENT\_PATH:=$(shell pwd)

#the current kernel version number

LINUX\_KERNEL:=$(shell uname -r)

#the absolute path

LINUX\_KERNEL\_PATH:=/usr/src/kernels/$(LINUX\_KERNEL)

#complie object

all:

make -C $(LINUX\_KERNEL\_PATH) M=$(CURRENT\_PATH) modules

rm -rf modules.order Module.symvers .\*.cmd \*.o \*.mod.c .tmp\_versions \*.unsigned

#clean

clean:

rm -rf modules.order Module.symvers .\*.cmd \*.o \*.mod.c \*.ko .tmp\_versions \*.unsigned

测试软中断的方法如下：

makeinsmod mysoftirq.ko

rmmod mysoftirq

dmesg | tail -9

# 运行结果

interrupt's top half!

=========================2013-4-22 14:4:57

my softirq function is been called!....=========================

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*2013-4-22 14:5:2

testsoftirq is exited!

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

#### **4.2 tasklet的实现**

tasklet的实验用默认的内核即可，我们切换到centos6.3的默认内核(uname -r: 2.6.32-279.el6.x86\_64)

从中我们也可以看出，内核之所以没有导出open\_softirq和raise\_softirq函数，可能还是因为提倡我们尽量用tasklet来实现中断的下半部工作。

tasklet测试代码：testtasklet.c

#include <linux/init.h>

#include <linux/slab.h>

#include <linux/module.h>

#include <linux/kernel.h>

#include <linux/rbtree.h>

#include <linux/time.h>

#include <linux/string.h>

#include <linux/interrupt.h>

MODULE\_LICENSE("Dual BSD/GPL");

static void my\_tasklet\_func(unsigned long);

// 定义 tasklet 结构

/\* mytasklet 必须定义在testtasklet\_init函数的外面，否则会出错 \*/

DECLARE\_TASKLET(mytasklet, my\_tasklet\_func, 1000);

static int testtasklet\_init(void)

{

printk(KERN\_ALERT "interrupt's top half!\n");

    // 如果在这里定义的话，那么 mytasklet是函数的局部变量，

    // 后面调度的时候会找不到 mytasklet

// DECLARE\_TASKLET(mytasklet, my\_tasklet\_func, 1000);

    // 调度tasklet， 处理器会在适当时候执行这个tasklet

    tasklet\_schedule(&mytasklet);

    return 0;

}

static void testtasklet\_exit(void)

{

    printk(KERN\_ALERT "\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\n");

    printk(KERN\_ALERT "testtasklet is exited!\n");

    printk(KERN\_ALERT "\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\n");

}

static void my\_tasklet\_func(unsigned long data)

{

    printk(KERN\_ALERT "=========================\n");

    printk(KERN\_ALERT "my tasklet function is been called!....\n");

    printk(KERN\_ALERT "parameter data is %ld\n", data);

    printk(KERN\_ALERT "=========================\n");

}

module\_init(testtasklet\_init);

module\_exit(testtasklet\_exit);

#### **4.3 工作队列的实现**

workqueue的例子的中静态定义了一个工作，动态定义了一个工作。

静态定义的工作由系统工作队列（events/n）调度，

动态定义的工作由自定义的工作队列（myworkqueue）调度。

测试工作队列的代码：testworkqueue.c

#include <linux/workqueue.h>

#include "kn\_common.h"

MODULE\_LICENSE("Dual BSD/GPL");

static void my\_work\_func(struct work\_struct \*);

static void my\_custom\_workqueue\_func(struct work\_struct \*);

/\* 静态创建一个工作，使用系统默认的工作者线程，即 events/n \*/

DECLARE\_WORK(mywork, my\_work\_func);

static int testworkqueue\_init(void)

{

    /\*自定义的workqueue \*/

struct workqueue\_struct \*myworkqueue = create\_workqueue("myworkqueue");

    /\* 动态创建一个工作 \*/

    struct work\_struct \*mywork2;

    mywork2 = kmalloc(sizeof(struct work\_struct), GFP\_KERNEL);

INIT\_WORK(mywork2, my\_custom\_workqueue\_func);

printk(KERN\_ALERT "interrupt's top half!\n");

    /\* 刷新系统默认的队列 \*/

    flush\_scheduled\_work();

    /\* 调度工作 \*/

schedule\_work(&mywork);

    /\* 刷新自定义的工作队列 \*/

    flush\_workqueue(myworkqueue);

    /\* 调度自定义工作队列上的工作 \*/

queue\_work(myworkqueue, mywork2);

    return 0;

}

static void testworkqueue\_exit(void)

{

    printk(KERN\_ALERT "\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\n");

    print\_current\_time(0);

    printk(KERN\_ALERT "my workqueue test is exited!\n");

    printk(KERN\_ALERT "\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\n");

}

static void my\_work\_func(struct work\_struct \*work)

{

    printk(KERN\_ALERT "=========================\n");

    print\_current\_time(0);

    printk(KERN\_ALERT "my workqueue function is been called!....\n");

    printk(KERN\_ALERT "=========================\n");

}

static void my\_custom\_workqueue\_func(struct work\_struct \*work)

{

    printk(KERN\_ALERT "=========================\n");

    print\_current\_time(0);

    printk(KERN\_ALERT "my cutomize workqueue function is been called!....\n");

    printk(KERN\_ALERT "=========================\n");

    kfree(work);

}

module\_init(testworkqueue\_init);

module\_exit(testworkqueue\_exit);

Makefile:

obj-m += myworkqueue.o

myworkqueue-objs := testworkqueue.o kn\_common.o

#generate the path

CURRENT\_PATH:=$(shell pwd)

#the current kernel version number

LINUX\_KERNEL:=$(shell uname -r)

#the absolute path

LINUX\_KERNEL\_PATH:=/usr/src/kernels/$(LINUX\_KERNEL)

#complie object

all:

make -C $(LINUX\_KERNEL\_PATH) M=$(CURRENT\_PATH) modules

rm -rf modules.order Module.symvers .\*.cmd \*.o \*.mod.c .tmp\_versions \*.unsigned

#clean

clean:

rm -rf modules.order Module.symvers .\*.cmd \*.o \*.mod.c \*.ko .tmp\_versions \*.unsigned

测试workqueue的方法如下：

makeinsmod myworkqueue.ko

rmmod myworkqueue

dmesg | tail -13

# 运行结果

interrupt's top half!

=========================2013-4-23 9:55:29

my workqueue function is been called!....=========================

=========================2013-4-23 9:55:29

my cutomize workqueue function is been called!....=========================

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*2013-4-23 9:55:29

my workqueue is exited!

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*