ARM驱动 驱动学习链接

Linux中断处理驱动程序编写

2013-03-05 21:43:08 | 分类： Linux 订阅 | 字号 | 举报

本章节我们一起来探讨一下Linux中的中断

中断与定时器:

中断的概念:指CPU在执行过程中，出现某些突发事件急待处理，CPU暂停执行当前程序，转去处理突发事件，处理完后CPU又返回原程序被中断的位置继续执行

中断的分类:内部中断和外部中断
内部中断:中断源来自CPU内部(软件中断指令、溢出、触发错误等)
外部中断:中断源来自CPU外部，由外设提出请求

屏蔽中断和不可屏蔽中断:
可屏蔽中断:可以通过屏蔽字被屏蔽，屏蔽后，该中断不再得到响应
不可屏蔽中断:不能被屏蔽

向量中断和非向量中断:

向量中断： CPU通常为不同的中断分配不同的中断号，当检测到某中断号的中断到来后，就自动跳转到与该中断号对应的地址执行

非向量中断:多个中断共享一个入口地址。进入该入口地址后再通过软件判断中断标志来识别具体哪个是中断
也就是说向量中断由软件提供中断服务程序入口地址，非向量中断由软件提供中断入口地址

```
/*典型的非向量中断首先会判断中断源，然后调用不同中断源的中断处理程序*/
irq_handler()
{
...
int int_src = read_int_status();/*读硬件的中断相关寄存器*/
switch(int_src){//判断中断标志
```

```
case DEV_A:
dev_a_handler();
break;
case DEV_B:
dev_b_handler();
break;
...
default:
break;
}
...
}
```

定时器中断原理:

定时器在硬件上也以来中断，PIT(可编程间隔定时器)接收一个时钟输入，
当时钟脉冲到来时，将目前计数值增1并与已经设置的计数值比较，若相等，证明计数周期满，产生定时器中断，并复位计数值。
如下图所示:

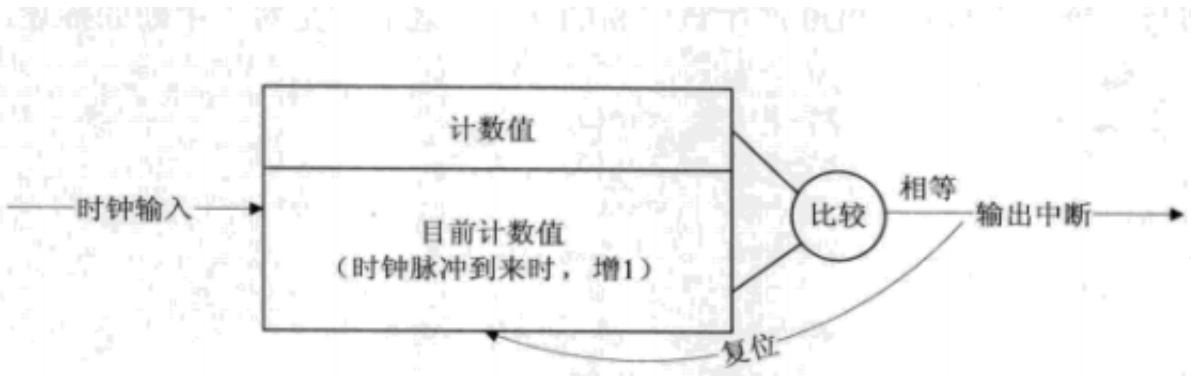

图 10.1　定时器工作原理

Linux中断处理程序架构:

Linux将中断分为:顶半部(top half)和底半部(bottom half)

顶半部:　　完成尽可能少的比较紧急的功能，它往往只是简单的读取寄存器中的中断状态并清除中断标志后就进行“登记中断”(也就是将底半部处理程序挂在到设备的底半部执行队列中)的工作
特点：响应速度快

底半部:　　中断处理的大部分工作都在底半部，它几乎做了中断处理程序的所有事情。
特点：处理相对来说不是非常紧急的事件

小知识:Linux中查看/proc/interrupts文件可以获得系统中断的统计信息。
如下图所示:

```
           Cpuo
0:        135253    XT-PIC    timer
1:            22    XT-PIC    i8042
2:             0    XT-PIC    cascade
8:             1    XT-PIC    rtc
10:          108    XT-PIC    eth0
11:         3707    XT-PIC    BusLogic BT-958
12:          313    XT-PIC    i8042
15:            4    XT-PIC    idel
NMI:           0
ERR:           0
```

第一列是中断号　　第二列是向CPU产生该中断的次数

介绍完相关基础概念后，在我们一起来探讨一个Linux中断编程

Linux中断编程:

内核维护了一个中断信号线的注册表，该注册表类似于I/O端口的注册表。驱动模块在使用中断前要先请求一个中断通道（或者中断请求**IRQ**），然后在使用后释放该通道。

在头文件<linux/interrupt.h>中申明了申请和释放IRQ的接口。

1.申请和释放中断

申请中断:

int request_irq(unsigned int irq,irq_handler_t handler,

unsigned long irqflags,const char *devname,void *dev_id)

参数介绍:

irq　　是要申请的硬件中断号

handler　是向系统登记的中断处理程序(顶半部)，是一个回调函数，

　　中断发生时，系统调用它，将dev_id参数传递给它。

irqflags:　　是中断处理的属性,可以指定中断的触发方式和处理方式:

触发方式:　　IRQF_TRIGGER_RISING、IRQF_TRIGGER_FALLING、IRQF_TRIGGER_HIGH、IRQF_TRIGGER_LOW

处理方式:　　IRQF_DISABLE表明中断处理程序是快速处理程序，快速处理程序被调用时屏蔽所有中断

IRQF_SHARED　　表示多个设备共享中断

dev_id　　在中断共享时会用到，一般设置为NULL

返回值:为0表示成功，返回-EINVAL表示中断号无效，返回-EBUSY表示中断已经被占用，且不能共享

顶半部的handler的类型irq_handler_t定义为

typedef irqreturn_t (*irq_handler_t)(int,void*);

typedef int irqreturn_t;

2.释放IRQ

有请求当然就有释放了

void free_irq(unsigned int irq,void *dev_id);

参数定义与request_irq类似

3.使能和屏蔽中断

void disable_irq(int irq);//等待目前中断处理完成(最好别在顶板部使用，你懂得)

void disable_irq_nosync(int irq);//立即返回

void enable_irq(int irq);//

4.屏蔽本CPU内所有中断:

#define local_irq_save(flags).../禁止中断并保存状态

void local_irq_disable(void);//禁止中断，不保存状态

下面来分别介绍一下顶半部和底半部的实现机制

底半部机制:

简介:底半部机制主要有tasklet、工作队列、软中断

1.底半部实现方法之一tasklet

(1)我们需要定义tasklet机器处理器并将两者关联

例如:

void my_tasklet_func(unsigned long);/*定义一个处理函数*/

DECLARE_TASKLET(my_tasklet,my_tasklet_func,data);

/*上述代码定义了名为my_tasklet的tasklet并将其与my_tasklet_func()函数绑定，传入的参数为data*/

(2)调度

tasklet_schedule(&my_tasklet);

//使用此函数就能在适当的时候进行调度运行

tasklet使用模板

```
    /*定义tasklet和底半部函数并关联*/
void xxx_do_tasklet(unsigned long);

DECLARE_TASKLET(xxx_tasklet,xxx_do_tasklet,0);
    //将xxx_tasklet与xxx_do_tasklet绑定，传入参数0

/*中断处理底半部*/
void xxx_do_tasklet(unsigned long)
{
...
}

/*中断处理顶半部*/
```

```
irqreturn_t xxx_interrupt(int irq,void *dev_id)
{
    ...
    tasklet_schedule(&xxx_tasklet);//调度底半部
    ...
}

/*设备驱动模块加载函数*/
int __init xxx_init(void)
{
...
/*申请中断*/
    /**/
result = request_irq(xxx_irq,xxx_interrupt,IRQF_DISABLED,"xxx",NULL);
...
return IRQ_HANDLED;
}

/*设备驱动模块卸载函数*/
void __exit xxx_exit(void)
{
...
/*释放中断*/
free_irq(xxx_irq,xxx_interrupt);
...
}
```

2.底半部实现方法之二---工作队列

使用方法和tasklet类似

相关操作:

struct work_struct my_wq; /*定义一个工作队列*/

void my_wq_func(unsigned long);/*定义一个处理函数*/

通过INIT_WORK()可以初始化这个工作队列并将工作队列与处理函数绑定

INIT_WORK(&my_wq,(void (*)(void *))my_wq_func,NULL);

/*初始化工作队列并将其与处理函数绑定*/

schedule_work(&my_wq);/*调度工作队列执行*/

/*工作队列使用模板*/

```
/*定义工作队列和关联函数*/
struct work_struct xxx_wq(unsigned long);

void xxx_do_work(unsigned long);

/*中断处理底半部*/
void xxx_do_work(unsigned long)
{
...
}
```

```c
/*中断处理顶半部*/
irqreturn_t xxx_interrupt(int irq,void *dev_id)
{
    ...
    schedule_work(&my_wq);//调度底半部
    ...
    return IRQ_HANDLED;
}


/*设备驱动模块加载函数*/
int xxx_init(void)
{
    ...
    /*申请中断*/
    result = request_irq(xxx_wq,xxx_interrupt,IRQF_DISABLED,"xxx",NULL);
    ...
    /*初始化工作队列*/
    INIT_WORK(&my_wq,(void (*)(void *))xxx_do_work,NULL);
}


/*设备驱动模块卸载函数*/
void xxx_exit(void)
{
    ...
    /*释放中断*/
    free_irq(xxx_irq,xxx_interrupt);
    ...
}
```

中断共享

中断共享是指多个设备共享一根中断线的情况

中断共享的使用方法:

(1).在申请中断时，使用IRQF_SHARED标识

(2).在中断到来时，会遍历共享此中断的所有中断处理程序，直到某一个函数返回IRQ_HANDLED，在中断处理程序顶半部中，应迅速根据硬件寄存器中的信息参照dev_id参数判断是否为本设备的中断，若不是立即返回IR1_NONE

```c
/*共享中断编程模板*/
irqreturn_t xxx_interrupt(int irq,void *dev_id,struct pt_regs *regs)
{
...
int status = read_int_status();/*获知中断源*/
if(!is_myint(dev_id,status))/*判断是否为本设备中断*/
```

```
return IRQ_NONE;/*不是本设备中断，立即返回*/

/*是本设备中断,进行处理*/
...
return IRQ_HANDLED;/*返回IRQ_HANDLER表明中断已经被处理*/
}

/*设备模块加载函数*/
int xxx_init(void)
{
...
/*申请共享中断*/
result = request_irq(sh_irq,xxx_interrupt,IRQF_SHARE,"xxx",xxx_dev);
...
}

/*设备驱动模块卸载函数*/
void xxx_exit()
{
...
/*释放中断*/
free_irq(xxx_irq,xxx_interrupt);
...
}
```

内核定时器

内核定时器编程:

简介:软件意义上的定时器最终是依赖于硬件定时器实现的，内核在时钟中断发生后检测各定时器是否到期，到期后定时器处理函数作为软中断在底半部执行。

Linux内核定时器操作：

1.timer_list结构体

每一个timer_list对应一个定时器

```
struct timer_list{
struct list_head entry;/*定时器列表*/
unsigned long expires;/*定时器到期时间*/
void (*function)(unsigned long);/*定时器处理函数*/
unsigned long data;/*作为参数被传递给定时器处理函数*/
struct timer_base_s *base;
...
};
```

当定时器满的时候，定时器处理函数将被执行

2.初始化定时器

void init_timer(struct timer_list * timer);

//初始化timer_list的entry的next为NULL，并给base指针赋值。

TIMER_INITIALIZER(_function,_expires,_data);//此宏用来

//赋值定时器结构体的function、expires

、data和base成员

#define TIMER_INITIALIZER(function,_expires,_data){

.entry = {.prev = TIMER_ENTRY_STATIC},\

.function= (_function), \

.expires = (_expire), \

.data = (_data), \

.base = &boot_tvec_bases,\

}

DEFINE_TIMER(_name,_function,_expires,_data)//定义一个定时器结构体变量

//并为此变量取名_name

//还有一个setup_timer()函数也可以用于定时器结构体的初始化，此函数大家自己去网上查吧

3.增加定时器

void add_timer(struct timer_list * timer);

//注册内核定时器，也就是将定时器加入到内核动态定时器链表当中

4.删除定时器

del_timer(struct timer_list *timer);

del_timer_sync()//在删除一个定时器时等待删除操作被处理完(不能用于中断上下文中)

5.修改定时器expires

int mod_timer(struct timer_list * timer,unsigned long expires);

//修改定时器的到期时间

/*内核定时器使用模板*/

/*xxx设备结构体*/

struct xxx_dev{

struct cdev cdev;

...

timer_list xxx_timer;/*设备要使用的定时器*/

};

```
/*xxx驱动中的某函数*/
xxx_funcl(...)
{
struct xxx_dev *dev = filp->private_data;
...
/*初始化定时器*/
init_timer(&dev->xxx_timer);
dev->xxx_timer.function = &xxx_do_timer;
dev->xxx_timer.data = (unsigned long)dev;
/*设备结构体指针作为定时器处理函数参数*/

dev->xxx_timer.expires = jiffes + delays;
/*添加(注册)定时器*/
add_timer(&dev->xxx_timer);
...
}

/*xxx驱动中的某函数*/
xxx_func2(...)
{
...
/*删除定时器*/
del_timer(&dev->xxx_timer);
...
}


/*定时器处理函数*/
static void xxx_do_timer(unsigned long arg)
{
struct xxx_device *dev = (struct xxx_device *)(arg);
...
/*调度定时器再执行*/
dev->xxx_timer.expires = jiffes + delay;
add_timer(&dev -> xxx_timer);
...
}

//定时器到期时间往往是在jiffies的基础上添加一个时延，若为HZ
则表示延迟一秒
```

内核中的延迟工作:
简介:对于这种周期性的工作，Linux提供了一套封装好的快捷机制，
本质上利用工作队列和定时器实现

这其中用到两个结构体:
(1)struct delayed_work{
struct work_struct work;
struct timer_list timer;
};

(2)struct work_struct{
atomic_long_t data;
...
}

相关操作:
int schedule_delay_work(struct delayed_work *work,unsigned long delay);
//当指定的delay到来时delay_work中的work成员的work_func_t类型成员func()会被执行
work_func_t类型定义如下:
typedef void (*work_func_t)(struct work_struct *work);
//delay参数的单位是jiffes

mescs_to_jiffies(unsigned long mesc);//将毫秒转化成jiffes单位

int cancel_delayed_work(struct delayed_work *work);
int cancel_delayed_work_sync(struct delayed_work *work);//等待直到删除(不能用于中断上下文)

内核延迟：
短延迟:
Linux内核提供了如下三个函数分别进行纳秒、微妙和毫秒延迟:
void ndelay(unsigned long nsecs);
void udelay(unsigned long usecs);
void mdelay(unsigned long msecs);

机制:根据CPU频率进行一定次数的循环(忙等待)

注意:在Linux内核中最好不要使用毫秒级的延时，因为这样会无谓消耗CPU的资源
对于毫秒以上的延时，Linux提供如下函数
void msleep(unsigned int millisecs);
unsigned long msleep_interruptible(unsigned int millisecs);//可以被打断
void ssleep(unsigned int seconds);
//上述函数使得调用它的进程睡眠指定的时间

长延迟:
机制:设置当前jiffies加上时间间隔的jiffies，直到未来的jiffies达到目标jiffires

/*实例:先延迟100个jiffies再延迟2s*/
unsigned long delay = jiffies + 100;
while(time_before(jiffies,delay));

/*再延迟2s*/
unsigned long delay = jiffies + 2*Hz;
while(time_before(jiffies,delay));//循环直到到达指定的时间

与timer_before()相对应的还有一个time_after

睡着延迟:
睡着延迟是比忙等待更好的一种方法
机制:在等待的时间到来之前进程处于睡眠状态，CPU资源被其他进程使用
实现函数有:
schedule_timeout()
schedule_timeout_uninterruptible()
其实在短延迟中的msleep() msleep_interruptible()
本质上都是依赖于此函数实现的

下面两个函数可以让当前进程加入到等待队列中，从而在等待队列上睡眠，当超时发生时，进程被唤醒
sleep_on_timeout(wait_queue_head_t *q,unsigned long timeout);
interruptible_sleep_on_timeout(wait_queue_head_t *q,unsigned long timeout);

最近读者


纯蓝忧郁

hanha…

gjianw21

some…

lianbin7

nbhh3…

darcyliu

leigeng1

在LOFTER的更多文章

Microsoft 编写无错

两个版本的故事 让我们直接进入memcpy，看看怎

C语言中 有符号数、无符号

C语言中存在两种整数算术运算，有符号型和无符号型。

关于类成员函数中访问同类

原址：http://wenku.baidu（注释： 为了尊重

Linux软连接和硬链接

1.Linux链接概念 Linux链接分两种，一种被称为硬链接

C++ 虚函数表解析 [转]

目录(?)[-] 前言 虚函数表 一般继承无虚函数覆盖 一般继

评论


纯蓝忧郁

发表