驱动开发之select与中断下半部分

驱动开发之select与中断下半部分：

中断上半部、下半部：

1、已知中断应该尽快完成，但是很多时候不能保证中断代码一定会及时完成。
2、当不能保证中断尽快完成时，内核将曾经的一个中断处理函数分成了两个中断处理函数
其中中断上半部用来处理紧急事件
中断下半部用来处理非紧急事件
3、什么是紧急事件：
直接操作硬件时
对时间要求非常敏感
不能被其他中断打断
4，中断下半部分类：
软中断(需要使用软中断号，但是所有的软中断号都被占用了)
小任务
工作队列

小任务:对软中断的二次封装：

初始化小任务：

```
1 void tasklet_init(struct tasklet_struct *t,void (*func)(unsignec
3 参数1：
4 参数2:中断下半部处理函数
5 参数3:给中断下半部处理函数传参的
```

调度小任务：

```
1 tasklet_schedule(struct tasklet_struct *t);
```

什么时候调用下半部?

在中断上半部处理函数返回前，或者返回后调用。

```
1 #include <linux/module.h>
2 #include <linux/fs.h>
3 #include <linux/platform_device.h>
4 #include <linux/device.h>
5 #include <asm/uaccess.h>
6 #include <linux/irqreturn.h>
7 #include <linux/interrupt.h>
8 #include <linux/sched.h>
9
10 int major;
11
12 struct class *cls;
13 struct device *devs;
```

公告

昵称： 向往的生活ing
园龄： 5年10个月
粉丝： 7
关注： 2
+加关注

< 2024年2月 >

日	一	二	三	四	五	六
28	29	30	31	1	2	3
4	5	6	7	8	9	10
11	12	13	14	15	16	17
18	19	20	21	22	23	24
25	26	27	28	29	1	2
3	4	5	6	7	8	9

搜索

找找看

常用链接

我的随笔
我的评论
我的参与
最新评论
我的标签

最新随笔

1.驱动开发之输入子系统
2.驱动开发之I2C总线
3.驱动开发之ADC驱动与通用设备树
4.驱动开发之select与中断下半部分
5.驱动开发之阻塞与按键驱动
6.驱动开发之platform总线与设备树
7.驱动开发之read/write
8.驱动开发之字符设备框架
9.驱动开发之模块与外部编译
10.系统移植

随笔分类

嵌入式—底层学习笔记(10)
嵌入式—基础学习笔记(2)
嵌入式—应用层学习笔记(5)

```
14 struct resource *res_key2;
15 struct resource *res_key3;
16
17 int key;
18 wait_queue_head_t keyq;
19 int flag = 0;
20 struct timer_list t;
21 int glo_irqno;
22
23 struct tasklet_struct ts;
24
25 irqreturn_t fs4412_key_handler(int irqno,void *id)
26 {
27     glo_irqno = irqno;
28     mod_timer(&t,jiffies + 30);
29     return IRQ_HANDLED;
30 }
31
32 void fs4412_key_timer_handler(unsigned long data)
33 {
34     if(glo_irqno == res_key2->start)
35         key = 2;
36     if(glo_irqno == res_key3->start)
37         key = 3;
38 //    wake_up(&keyq);
39     wake_up_interruptible(&keyq);
40     flag = 1;
41     tasklet_schedule(&ts);
42 }
43
44 void fs4412_key_task(unsigned long data)
45 {
46     printk("task schedule\n");
47 }
48
49 int fs4412_key_open(struct inode *inode ,struct file *filp)
50 {
51     return 0;
52 }
53
54 ssize_t fs4412_key_read(struct file *filp,char __user *ubuf,size_t size,loff_t *off
55 {
56     int ret;
57
58 //    wait_event(keyq,flag != 0);
59     wait_event_interruptible(keyq,flag != 0);
60     ret = copy_to_user(ubuf,&key,sizeof(key));
61
62     flag = 0;
63     return sizeof(key);
64 }
65
66 struct file_operations fops = {
67     .owner = THIS_MODULE,
68     .open = fs4412_key_open,
69     .read = fs4412_key_read,
70 };
71
72 int fs4412_key_probe(struct platform_device *pdev)
73 {
74     int ret;
75     printk("match ok\n");
76
77     major = register_chrdev(0,"key",&fops);
78     cls = class_create(THIS_MODULE,"key");
79     devs = device_create(cls,NULL,MKDEV(major,0),NULL,"key");
80
81     res_key2 = platform_get_resource(pdev,IORESOURCE_IRQ,0);//索引interrupts = <>,<>
82     res_key3 = platform_get_resource(pdev,IORESOURCE_IRQ,1);
83     ret = request_irq(res_key2->start,fs4412_key_handler,IRQF_TRIGGER_FALLING,"key2"
84     ret = request_irq(res_key3->start,fs4412_key_handler,IRQF_TRIGGER_FALLING,"key3"
85
86     init_waitqueue_head(&keyq);
87
88     init_timer(&t);
89     t.function = fs4412_key_timer_handler;//定时器中断处理函数
```

随笔档案

2018年9月(10)
2018年8月(7)

阅读排行榜

1. 驱动开发之platform总线与设备树(2807)
2. 驱动开发之ADC驱动与通用设备树(2200)
3. 驱动开发之read/write(1592)
4. 驱动开发之I2C总线(1390)
5. 驱动开发之模块与外部编译(1343)

评论排行榜

1. 驱动开发之ADC驱动与通用设备树(1)

推荐排行榜

1. 驱动开发之输入子系统(1)
2. 驱动开发之ADC驱动与通用设备树(1)
3. 驱动开发之模块与外部编译(1)

最新评论

1. Re:驱动开发之ADC驱动与通用设备树
感觉好熟悉的文章，是不是李相山的杰作?
--小韦爵爷

```
90     add_timer(&t);//让内核认识定时器中断处理函数
91
92     tasklet_init(&ts,fs4412_key_task,0);
93     return 0;
94 }
95
96 int fs4412_key_remove(struct platform_device *pdev)
97 {
98     del_timer(&t);
99     free_irq(res_key3->start,NULL);
100    free_irq(res_key2->start,NULL);
101    device_destroy(cls,MKDEV(major,0));
102    class_destroy(cls);
103    unregister_chrdev(major,"key");
104    return 0;
105 }
106
107 struct of_device_id fs4412_dt_tbl[] = {
108    {
109        .compatible = "fs4412,key",
110    },
111    {},
112 };
113
114
115 struct platform_driver pdrv = {
116    .driver = {
117        .name = "fs4412-key",
118        .of_match_table = fs4412_dt_tbl,
119    },
120    .probe = fs4412_key_probe,
121    .remove = fs4412_key_remove,
122 };
123
124 module_platform_driver(pdrv);
125 MODULE_LICENSE("GPL");
```

工作队列：

```
1 INIT_WORK(struct work_struct *,void (*)(struct work_struct *work
2
3 static inline bool schedule_work(struct work_struct *work)
```

如何选择下半部的使用方法?

小任务运行在中断上下文中(内核空间)，对时间要求敏感
工作队列运行在进程上下文中(用户空间和内核空间)，对时间不敏感

在工作队列中可以使用等待队列，互斥锁，信号量。小任务不可以。

```
1 应用层：
2 int select(int nfds, fd_set *readfds, fd_set *writefds,fd_set*ex
3 参数1:文件描述符最大值+1
4 参数2:读表
5 参数3:写表
6 参数4:异常表
7 参数5:超时检测
8 返回值:出错返回-1，超时返回0，成功返回>0
9 功能:如果没有就绪的文件描述符则阻塞，如果有就绪的文件描述符则唤醒，唤醒的同时
```

```
1 FD_SET();将指定文件描述符加入表中
2 FD_ZERO();清空文件描述符表
```

```
FD_CLR();删除指定的文件描述符
FD_ISSET();判断fd是否在表中
```

```
内核源码分析：
struct file_operations
{
unsigned int (*poll) (struct file *, struct poll_table_struct *
}

vi fs/select.c
找到do_select函数
mask = (*f_op->poll)(f.file, wait);

poll_initwait(&table);

--->void poll_initwait(struct poll_wqueues *pwq)
{
    init_poll_funcptr(&pwq->pt, __pollwait);
    pwq->polling_task = current;//当前进程
}

--->static inline void init_poll_funcptr(poll_table *pt, poll_c
{
     pt->_qproc = qproc;//将__pollwait赋值给了_qproc
 }

如果想要调用__pollwait,需要使用
static inline void poll_wait(struct file * filp, wait_queue_hea
{
     if (p && p->_qproc && wait_address)
    p->_qproc(filp, wait_address, p);//调用__pollwait
}

--->static void __pollwait(struct file *filp, wait_queue_head_t
{
     struct poll_wqueues *pwq = container_of(p, struct poll_wqu
    struct poll_table_entry *entry = poll_get_entry(pwq);
     if (!entry)
    return;
    entry->filp = get_file(filp);
    entry->wait_address = wait_address;
    entry->key = p->_key;
     init_waitqueue_func_entry(&entry->wait, pollwake);//唤醒接口
    entry->wait.private = pwq;
    add_wait_queue(wait_address, &entry->wait);//将等待队列项添加
}
```

```
#include <linux/module.h>
#include <linux/fs.h>
#include <linux/platform_device.h>
#include <linux/device.h>
#include <asm/uaccess.h>
#include <linux/irqreturn.h>
#include <linux/interrupt.h>
```

```c
#include <linux/sched.h>

int major;

struct class *cls;
struct device *devs;
struct resource *res_key2;
struct resource *res_key3;

int key;
wait_queue_head_t keyq;
int flag = 0;
struct timer_list t;
int glo_irqno;

struct work_struct ws;

irqreturn_t fs4412_key_handler(int irqno,void *id)
{
    glo_irqno = irqno;
    mod_timer(&t,jiffies + 30);
    return IRQ_HANDLED;
}

void fs4412_key_timer_handler(unsigned long data)
{
    if(glo_irqno == res_key2->start)
        key = 2;
    if(glo_irqno == res_key3->start)
        key = 3;
//    wake_up(&keyq);
    wake_up_interruptible(&keyq);
    flag = 1;
    schedule_work(&ws);
}

void fs4412_key_work(struct work_struct *work)
{
    printk("work schedule\n");
}

int fs4412_key_open(struct inode *inode ,struct file *filp)
{
    return 0;
}

ssize_t fs4412_key_read(struct file *filp,char __user *ubuf,size_t size,loff_t *off
{
    int ret;

//    wait_event(keyq,flag != 0);
    wait_event_interruptible(keyq,flag != 0);
    ret = copy_to_user(ubuf,&key,sizeof(key));

    flag = 0;
    return sizeof(key);
}

struct file_operations fops = {
    .owner = THIS_MODULE,
    .open = fs4412_key_open,
    .read = fs4412_key_read,
};

int fs4412_key_probe(struct platform_device *pdev)
{
    int ret;
    printk("match ok\n");

    major = register_chrdev(0,"key",&fops);
    cls = class_create(THIS_MODULE,"key");
    devs = device_create(cls,NULL,MKDEV(major,0),NULL,"key");

    res_key2 = platform_get_resource(pdev,IORESOURCE_IRQ,0);//索引interrupts = <>,<>
    res_key3 = platform_get_resource(pdev,IORESOURCE_IRQ,1);
    ret = request_irq(res_key2->start,fs4412_key_handler,IRQF_TRIGGER_FALLING,"key2"
```

```
    ret = request_irq(res_key3->start,fs4412_key_handler,IRQF_TRIGGER_FALLING,"key3"

    init_waitqueue_head(&keyq);

    init_timer(&t);
    t.function = fs4412_key_timer_handler;//定时器中断处理函数
    add_timer(&t);//让内核认识定时器中断处理函数

    INIT_WORK(&ws,fs4412_key_work);
    return 0;
}

int fs4412_key_remove(struct platform_device *pdev)
{
    del_timer(&t);
    free_irq(res_key3->start,NULL);
    free_irq(res_key2->start,NULL);
    device_destroy(cls,MKDEV(major,0));
    class_destroy(cls);
    unregister_chrdev(major,"key");
    return 0;
}

struct of_device_id fs4412_dt_tbl[] = {
    {
        .compatible = "fs4412,key",
    },
    {},
};


struct platform_driver pdrv = {
    .driver = {
        .name = "fs4412-key",
        .of_match_table = fs4412_dt_tbl,
    },
    .probe = fs4412_key_probe,
    .remove = fs4412_key_remove,
};

module_platform_driver(pdrv);
MODULE_LICENSE("GPL");
```

```
#include <stdio.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>

int main(int argc, const char *argv[])
{
    int fd;

    fd = open("/dev/key",O_RDWR);
    if(fd == -1)
    {
        perror("open");
        return -1;
    }

    int key;
    while(1)
    {
        read(fd,&key,sizeof(key));
        printf("key = %d\n",key);
    }
    return 0;
}
```

异步通知(信号操作):

```c
#include<stdio.h>
#include<sys/types.h>
#include<sys/stat.h>
#include<fcntl.h>
#include<unistd.h>
#include<errno.h>
#include<stdlib.h>
#include<signal.h>
int fd;
char buf[64] = {0};

void func(int sig)
{
    read(fd,buf,sizeof(buf));
    printf("%s\n",buf);
}

int main(int argc, const char *argv[])
{

    char buf1[64] = {0};
    if(mkfifo("./fifo",0664) == -1)
    {
        if(errno == EEXIST)
        {
            fd = open("./fifo",O_RDONLY);
        }
        else
        {
            perror("mkfifo");
            exit(1);
        }
    }
    else
    {
        fd = open("./fifo",O_RDONLY);
    }

    signal(SIGIO,func);

    int flag;
    flag = fcntl(fd,F_GETFL);
    flag = flag | O_ASYNC;
    fcntl(fd,F_SETFL,flag);
    fcntl(fd,F_SETOWN,getpid());
    while(1)
    {
        printf("hello\n");
        sleep(1);
    }
    return 0;
}
```

```c
#include<stdio.h>
#include<sys/types.h>
#include<sys/stat.h>
#include<fcntl.h>
#include<unistd.h>
#include<errno.h>
#include<stdlib.h>
#include<string.h>

int main(int argc, const char *argv[])
{
    int fd;

```

```
    char buf[64] = {0};
    if(mkfifo("./fifo",0664) == -1)
    {
        if(errno == EEXIST)
        {
            fd = open("./fifo",O_WRONLY);
        }
        else
        {
            perror("mkfifo");
            exit(1);
        }
    }
    else
    {
        fd = open("./fifo",O_WRONLY);
    }

    while(1)
    {
        fgets(buf,sizeof(buf),stdin);
        write(fd,buf,strlen(buf) + 1);
    }
    return 0;
}
```

```
应用程序：
int flag;
fd = open("1.txt",O_RDONLY);//一旦打开文件，内核会创建file结构体,其中
flag = fcntl(fd,F_GETFL);//从file结构体中获取O_RDONLY保存到flag变量
flag |= O_ASYNC;//将O_RDONLY | O_ASYNC保存到flag变量中（flag是自定义
fctnl(fd,F_SETFL,flag);//将O_RDONLY | O_ASYNC保存到file结构体的f_f

fcntl(fd,F_SETOWN,getpid());//需要注册信号，但是这个函数只指定了给哪个
```

```
内核源码分析：
vi arch/arm/include/uapi/asm/unistd.h

#define __NR_fcntl (__NR_SYSCALL_BASE+ 55)

--->__SYSCALL(__NR_fcntl, sys_fcntl)

--->err = do_fcntl(fd, cmd, arg, f.file);

--->case F_GETFL:
err = filp->f_flags;对应了flag = fcntl(fd,F_GETFL);
    break;
    case F_SETFL:
    err = setfl(fd, filp, arg);fctnl(fd,F_SETFL,flag)

---> if (((arg ^ filp->f_flags) & FASYNC) && filp->f_op->fasync
error = filp->f_op->fasync(fd, filp, (arg & FASYNC) != 0);
}
上层如果执行fctnl(fd,F_SETFL,flag)，驱动层可能执行fasync接口,前提是fl
```

```c
#include <linux/module.h>
#include <linux/fs.h>
#include <linux/platform_device.h>
#include <linux/device.h>
#include <asm/uaccess.h>
#include <linux/irqreturn.h>
#include <linux/interrupt.h>
#include <linux/sched.h>
#include <linux/poll.h>

int major;

struct class *cls;
struct device *devs;
struct resource *res_key2;
struct resource *res_key3;

int key;
wait_queue_head_t keyq;
int flag = 0;
struct timer_list t;
int glo_irqno;

struct fasync_struct *fa;

irqreturn_t fs4412_key_handler(int irqno,void *id)
{
    glo_irqno = irqno;
    mod_timer(&t,jiffies + 30);
    return IRQ_HANDLED;
}

void fs4412_key_timer_handler(unsigned long data)
{
    if(glo_irqno == res_key2->start)
        key = 2;
    if(glo_irqno == res_key3->start)
        key = 3;
//    wake_up(&keyq);
    wake_up_interruptible(&keyq);
    flag = 1;

    //注册SIGIO信号
    kill_fasync(&fa,SIGIO,POLLIN);
}

int fs4412_key_open(struct inode *inode ,struct file *filp)
{
    return 0;
}

ssize_t fs4412_key_read(struct file *filp,char __user *ubuf,size_t size,loff_t *off
{
    int ret;

//    wait_event(keyq,flag != 0);
    wait_event_interruptible(keyq,flag != 0);
    ret = copy_to_user(ubuf,&key,sizeof(key));

    flag = 0;
    return sizeof(key);
}

int fs4412_key_fasync(int fd,struct file *filp,int on)
{
    fasync_helper(fd,filp,on,&fa);//为了kill_fasync函数做一些成员赋值
    return 0;
}

struct file_operations fops = {
    .owner = THIS_MODULE,
    .open = fs4412_key_open,
    .read = fs4412_key_read,
```

```
    .fasync = fs4412_key_fasync,
};

int fs4412_key_probe(struct platform_device *pdev)
{
    int ret;
    printk("match ok\n");

    major = register_chrdev(0,"key",&fops);
    cls = class_create(THIS_MODULE,"key");
    devs = device_create(cls,NULL,MKDEV(major,0),NULL,"key");

    res_key2 = platform_get_resource(pdev,IORESOURCE_IRQ,0);//索引interrupts = <>,<>
    res_key3 = platform_get_resource(pdev,IORESOURCE_IRQ,1);
    ret = request_irq(res_key2->start,fs4412_key_handler,IRQF_TRIGGER_FALLING,"key2"
    ret = request_irq(res_key3->start,fs4412_key_handler,IRQF_TRIGGER_FALLING,"key3"

    init_waitqueue_head(&keyq);

    init_timer(&t);
    t.function = fs4412_key_timer_handler;//定时器中断处理函数
    add_timer(&t);//让内核认识定时器中断处理函数
    return 0;
}

int fs4412_key_remove(struct platform_device *pdev)
{
    del_timer(&t);
    free_irq(res_key3->start,NULL);
    free_irq(res_key2->start,NULL);
    device_destroy(cls,MKDEV(major,0));
    class_destroy(cls);
    unregister_chrdev(major,"key");
    return 0;
}

struct of_device_id fs4412_dt_tbl[] = {
    {
        .compatible = "fs4412,key",
    },
    {

       },
};


struct platform_driver pdrv = {
    .driver = {
        .name = "fs4412-key",
        .of_match_table = fs4412_dt_tbl,
    },
    .probe = fs4412_key_probe,
    .remove = fs4412_key_remove,
};

module_platform_driver(pdrv);
MODULE_LICENSE("GPL");
```

```
#include <stdio.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <signal.h>

int fd;
int key;
void func(int signo)
{
    if(signo == SIGIO)
    {
```

```
13         read(fd,&key,sizeof(key));
14         printf("key = %d\n",key);
15     }
16 }
17
18 int main(int argc, const char *argv[])
19 {
20     signal(SIGIO,func);
21
22     fd = open("/dev/key",O_RDWR);
23     if(fd == -1)
24     {
25         perror("open");
26         return -1;
27     }
28
29     int flag;
30     flag = fcntl(fd,F_GETFL);
31     flag |= FASYNC;
32     fcntl(fd,F_SETFL,flag);//主要功能为了调用驱动中的fasync接口
33     fcntl(fd,F_SETOWN,getpid());
34
35     while(1)
36     {
37         printf("hello\n");
38         sleep(1);
39     }
40     return 0;
41 }
```

总结:

```
1 中断上半部和下半部
2 将一个中断处理函数分成了2个函数。
3
4 上半部用来处理紧急事件
5
6 紧急事件:
7 1、直接操作硬件
8 2、对时间敏感
9 3、不能被打断的
10
11 下半部:
12 软中断:
13     open_softirq();
14     raise_softirq();
15 几乎不会使用软中断，内核提供的软中断号都已经被占用了，对应的接口也没有导出
16
17 小任务:属于一种特殊的软中断,运行在中断上下文,中断处理函数中不能有延时，只能
18     tasklet_init();//初始化小任务
19     tasklet_schedule();//调度小任务
20
21 工作队列:运行在进程上下文,可以有延时，可以使用互斥锁，信号量，等待队列。
22     INIT_WORK();
23     schedule_work();
24
25 什么时候调度中断下半部?
26 一般都是在上半部中断处理函数返回前或者返回后。
27
28 多路复用:
```

```
int select(文件描述符最大值+1,读表,写表,异常表,超时);
功能:如果文件描述符没有就绪则阻塞。
      如果文件描述符就绪则唤醒,唤醒后会清除未就绪的文件描述符。

vi arch/arm/include/uapi/asm/unistd.h


异步通知:当一种事件(按键按下)发生时会在应用层产生一个SIGIO信号，进而捕捉S

应用层:

signal(SIGIO, func);
fd = open();

flag = fcntl(fd,F_GETFL);//从file结构体的f_flags成员中获取属性存放给
flag |= FASYNC;
fcntl(fd,F_SETFL,flag);//会和FASYNC进行&,如果成功则会执行驱动中的fas
fcntl(fd,F_SETOWN,getpid());//明确给哪个进程注册SIGIO信号(还没有真正

while(1)
{
    功能;
}

驱动层:
中断处理函数
{
    kill_fasync();
    //给进程注册信号，但是注册信号时需要使用到一下变量内容，需要准备这些内
}

fasync()
{
    fasync_helper();//为kill_fasync的功能实现做准备。
}
```

分类: 嵌入式—底层学习笔记

好文要顶 关注我 收藏该文 微信分享

向往的生活ing
粉丝 - 7 关注 - 2
+加关注

0 推荐 0 反对

升级成为会员

« 上一篇: 驱动开发之阻塞与按键驱动
» 下一篇: 驱动开发之ADC驱动与通用设备树

posted @ 2018-09-18 19:08 向往的生活ing 阅读(644) 评论(0) 编辑 收藏 举报

会员力量，点亮园子希望 刷新页面 返回顶部

登录后才能查看或发表评论，立即 登录 或者 逛逛 博客园首页