

Tekkaman Ninja

tekkamanninja.blog.chinaunix.net

Linux我的梦想，我的未来！ 本博客的原创文章的内容会不定期更新或修正错误！转载文章都会注明出处，若有侵权，请即时同我联系，我一定马上删除！！原创文章版权所有！如需转载，请注明出处： tekkamanninja.blog.chinaunix.net ，谢谢合作！！ 拒绝一切广告性质的评论，一经发现立即举报并删除！

首页 | 博文目录 | 关于我



tekkamanninj

博客访问： 75924

博文数量： 263

博客积分： 15936

博客等级： 上将

技术积分： 13951

用户组： 普通用户

注册时间： 2007-03-27 11:22

加关注 短消息

论坛 加好友

个人简介

Fedora-ARM

文章分类

- 全部博文（263）
- Red Hat（2）

代码管理（6）

感悟（3）

Linux调试技术（2）

MaxWit（1）

Linux设备驱动程（41）

Android（20）

neo freerunner（2）

计算机硬件技术（9）

网络（WLAN or LA（8）

励志（7）

ARM汇编语言（1）

Linux操作系统的（15）

Linux内核研究（38）

ARM-Linux应用程（19）

建立根文件系统（4）

Linux内核移植（14）

Bootloader（45）

建立ARM-Linux交（7）

未分配的博文（19）

文章存档

2014年（1）

Linux设备驱动程序学习（11）-中断处理

2007-11-29 12:00:04

分类： LINUX

Linux设备驱动程序学习（11）-中断处理

可以让设备在产生某个事件时通知处理器的方法就是中断。一个“中断”仅是一个信号，当硬件需要获得处理器对它的关注时，就可以发送这个信号。Linux 处理中断的方式非常类似在用户空间处理信号的方式。大多数情况下，一个驱动只需要为它的设备的中断注册一个处理例程，并当中断到来时进行正确的处理。本质上来讲，中断处理例程和其他的代码并行运行。因此，它们不可避免地引起并发问题，并竞争数据结构和硬件。透彻地理解并发控制技术对中断来讲非常重要。

安装中断处理例程

内核维护了一个中断信号线的注册表，类似于 I/O 端口的注册表。模块在使用中断前要先请求一个中断通道(或者 IRQ中断请求)，并在使用后释放它。所用的函数声明在 <linux/interrupt.h> (在此文件中并未真正包含，是通过它include的文件间接包含的，函数在/kernel/irq/Manage.h中)，中断注册和释放的函数接口如下：

```
int request_irq(unsigned int irq,
               irqreturn_t (*handler)(int, void *, struct pt_regs *),
               unsigned long flags,
               const char *dev_name,
               void *dev_id);

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

request_irq 的返回值： 0 指示成功，或返回一个负的错误码，如 -EBUSY 表示另一个驱动已经占用了你所请求的中断线。

函数的参数如下：

- unsigned int irq：请求的中断号
- irqreturn_t (*handler)：安装的处理函数指针。
- unsigned long flags：一个与中断管理相关的位掩码选项。
- const char *dev_name：传递给 request_irq 的字符串，用来在 /proc/interrupts 来显示中断的拥有者。
- void *dev_id：用于共享中断信号线的指针。它是唯一的标识，在中断线空闲时可以使用它，驱动程序也可以用它来指向自己的私有数据区(来标识哪个设备产生中断)。若中断没有被共享，dev_id 可以设置为 NULL，但推荐用它指向设备的数据结构。

- flags 中可以设置的位如下：
- SA_INTERRUPT：快速中断标志。快速中断处理例程运行在当前处理器禁止中断的状态下。
- SA_SHIRQ：在设备间共享中断标志。
- SA_SAMPLE_RANDOM：该位表示产生的中断能对 /dev/random 和 /dev/urandom 使用的熵池（entropy pool）有贡献。读取这些设备会返回真正的随机数，从而有助于应用程序软件选择用于加密的安全密钥。若设备以真正随机的周期产生中断，就应当设置这个标志。若设备中断是可预测的，这个标志不值得设置。可能被攻击者影响的设备不应当设置这个标志。更多信息看 drivers/char/random.c 的注释。

中断处理例程可在驱动初始化时或在设备第一次打开时安装。推荐在设备第一次打开、硬件被告知产生中断前时申请中断，因为可以共享有限的中断资源。这样调用 free_irq 的位置是设备最后一次被关闭、硬件被告知不用再中断处理器之后。但这种方式的缺点是必须为每个设备维护一个打开计数。

以下是中断申请的示例（并口）：

2013年（3）
2012年（61）
2011年（66）
2010年（27）
2009年（30）
2008年（23）
2007年（52）

我的朋友



小蜗牛快



cfm5538



jikaishi



shizhenc



pxy05215



李怀远



yan1990



wkm81018



xiousi

最近访客



apang199



appcount



zaichu



lhxzui



小蜗牛快



小尾巴鱼



erain_30



hushup



wilfred_

订阅

推荐博文

- linux 3.x的 通用时钟架构 ...
- SCN的相关解析
- Flash驱动学习
- 浅谈nagios之state type和 no...
- DB2 (Linux 64位) 安装教程...
- insert语句造成latch:library...
- 2014.06.13 网络公开课《让我...
- MySQL Slave异常关机的处理 (...)
- 巧用shell脚本分析数据库用户...
- 查询linux, HP-UX的cpu信息...

热词专题

- linux系统权限修复——学生误...
- Modbus协议使用
- linux
- busybox原理
- php环境搭建教程

```
if (short_irq >= 0)
{
    result = request_irq(short_irq, short_interrupt,
                        SA_INTERRUPT, "short", NULL);

    if (result) {
        printk(KERN_INFO "short: can't get assigned irq %i\n",
                short_irq);

        short_irq = -1;
    } else { /*打开中断硬件的中断能力*/
        outb(0x10, short_base+2);
    }
}
```

i386 和 x86_64 体系定义了一个函数来查询一个中断线是否可用：

```
int can_request_irq(unsigned int irq, unsigned long flags); /*当能够成功分配给定中断，
则返回非零值。但注意，在 can_request_irq 和 request_irq 的调用之间给定中断可能被占用*/
```

快速和慢速处理例程

快速中断是那些能够很快处理的中断，而处理慢速中断会花费更长的时间。在处理慢速中断时处理器重新使能中断，避免快速中断被延时过长。在现代内核中，快速和慢速中断的区别已经消失，剩下的只有一个：快速中断（使用 SA_INTERRUPT）执行时禁止所有在当前处理器上的其他中断。注意：其他的处理器仍然能够处理中断。

除非你充足的理由在禁止其他中断情况下来运行中断处理例程，否则不应当使用SA_INTERRUPT。

x86中断处理内幕

这个描述是从 2.6 内核 arch/i386/kernel/irq.c, arch/i386/kernel/apic.c, arch/i386/kernel/entry.S, arch/i386/kernel/i8259.c, 和 include/asm-i386/hw_irq.h 中得出，尽管基本概念相同，硬件细节与其他平台上不同。

底层中断处理代码在汇编语言文件 entry.S。在所有情况下，这个代码将中断号压栈并且跳转到一个公共段，公共段会调用 do_IRQ（在 irq.c 中定义）。do_IRQ 做的第一件事是应答中断以便中断控制器能够继续其他事情。它接着获取给定 IRQ 号的一个自旋锁，阻止其他 CPU 处理这个 IRQ，然后清除几个状态位（包括IRQ_WAITING）然后查找这个 IRQ 的处理例程。若没有找到，什么也不做；释放自旋锁，处理任何待处理的软件中断，最后 do_IRQ 返回。从中断中返回的最后一件事可能是一次处理器的重新调度。

IRQ的探测是通过为每个缺乏处理例程的IRQ设置 IRQ_WAITING 状态位来完成。当中断发生，因为没有注册处理例程，do_IRQ 清除这个位并且接着返回。当probe_irq_off被一个函数调用，只需搜索没有设置 IRQ_WAITING 的 IRQ。

/proc 接口

当硬件中断到达处理器时，内核提供的一个内部计数器会递增，产生的中断报告显示在文件 /proc/interrupts中。这一方法可以用来检查设备是否按预期地工作。此文件只显示当前已安装处理例程的中断的计数。若以前request_irq的一个中断，现在已经free_irq了，那么就不会显示在这个文件中，但是它可以显示终端共享的情况。

/proc/stat记录了几个关于系统活动的底层统计信息，包括（但不仅限于）自系统启动以来收到的中断数。stat 的每一行以一个字符串开始，是该行的关键词：intr 标志是中断计数。第一个数是所有中断的总数，而其他每一个代表一个单独的中断线的计数，从中断 0 开始（包括当前没有安装处理例程的中断），无法显示终端共享的情况。

以上两个文件的一个不同是：/proc/interrupts几乎不依赖体系，而/proc/stat的字段数依赖内核下的硬件中断，其定义在<asm/irq.h>中。ARM的定义为：

```
#define NR_IRQS 128
```

自动检测 IRQ 号

驱动初始化时最迫切的问题之一是决定设备要使用的IRQ 线，驱动需要信息来正确安装处理例程。自动检测中断号对驱动的可用性来说是一个基本需求。有时自动探测依赖一些设备具有的默认特性，以下是典型的并口中断探测程序：

```
if (short_irq < 0) /* 依靠使并口的端口号，确定中断*/
switch(short_base) {
```

```
case 0x378: short_irq = 7; break;
case 0x278: short_irq = 2; break;
case 0x3bc: short_irq = 5; break;
}
```

有的驱动允许用户在加载时覆盖默认值：

```
insmod xxxxx.ko irq=x
```

当目标设备有能力告知驱动它要使用的中断号时，自动探测中断号只是意味着探测设备，无需做额外的工作探测中断。

但不是每个设备都对程序员友好，对于他们还是需要一些探测工作。这个工作技术上非常简单：驱动告知设备产生中断并且观察发生了什么。如果一切顺利，则只有一个中断信号线被激活。尽管探测在理论上简单，但实现可能不简单。有 2 种方法来进行探测中断：[调用内核定义的辅助函数](#)和[DIY探测](#)。

（1）调用内核定义的辅助函数

Linux 内核提供了一个底层设施来探测中断号，且只能在非共享中断模式下工作，它包括 2 个函数，在 `<linux/interrupt.h>` 中声明（也描述了探测机制）：

```
unsigned long probe_irq_on(void);
/*这个函数返回一个未分配中断的位掩码。驱动必须保留返回的位掩码，并在后面传递给
probe_irq_off。在调用probe_irq_on之后，驱动应当安排它的设备产生至少一次中断*/

int probe_irq_off(unsigned long);
/*在请求设备产生一个中断后，驱动调用这个函数，并将 probe_irq_on 返回的位掩码作为参数传
递给probe_irq_off。probe_irq_off 返回在“probe_on”之后发生的中断号。如果没有中断发生，
返回 0；如果产生了多次中断，probe_irq_off 返回一个负值*/
```

程序员应当注意在调用 `probe_irq_on` 之后启用设备上的中断，并在调用 `probe_irq_off` 前禁用。此外还必须记住在 `probe_irq_off` 之后服务设备中待处理的中断。

以下是LDD3中的并口示例代码，（并口的管脚 9 和 10 连接在一起，探测五次失败后放弃）：

```
int count = 0;
do
{
    unsigned long mask;
    mask = probe_irq_on();
    outb_p(0x10, short_base+2); /* enable reporting */
    outb_p(0x00, short_base); /* clear the bit */
    outb_p(0xFF, short_base); /* set the bit: interrupt! */
    outb_p(0x00, short_base+2); /* disable reporting */
    udelay(5); /* give it some time */
    short_irq = probe_irq_off(mask);

    if (short_irq == 0) { /* none of them? */
        printk(KERN_INFO "short: no irq reported by probe\n");
        short_irq = -1;
    }
} while (short_irq < 0 && count++ < 5);
if (short_irq < 0)
    printk("short: probe failed %i times, giving up\n", count);
```

最好只在模块初始化时探测中断线一次。

大部分体系定义了这两个函数（即便是空的）来简化设备驱动的移植。

（2）DIY探测

DIY探测与前面原理相同：使能所有未使用的中断，接着等待并观察发生什么。我们对设备的了解：通常一个设备能够使用3或4个IRQ 号中的一个来进行配置，只探测这些 IRQ 号使我们能不必测试所有可能的中断就探测到正确的IRQ 号。

下面的LDD3中的代码通过测试所有“可能的”中断并且察看发生的事情来探测中断。 `trials` 数组列出要尝试的中断，以 0 作为结尾标志；`tried` 数组用来跟踪哪个中断号已经被这个驱动注册。

```

int trials[] = {3, 5, 7, 9, 0};
int tried[] = {0, 0, 0, 0, 0};
int i, count = 0;

for (i = 0; trials[i]; i++)
    tried[i] = request_irq(trials[i], short_probing,
                           SA_INTERRUPT, "short probe", NULL);

do
{
    short_irq = 0; /* none got, yet */
    outb_p(0x10, short_base+2); /* enable */
    outb_p(0x00, short_base);
    outb_p(0xFF, short_base); /* toggle the bit */
    outb_p(0x00, short_base+2); /* disable */
    udelay(5); /* give it some time */
    /* 等待中断, 若在这段时间有中断产生, handler会改变 short_irq */
    /* the value has been set by the handler */
    if (short_irq == 0) { /* none of them? */
        printk(KERN_INFO "short: no irq reported by probe\n");
    }
} while (short_irq <= 0 && count++ < 5);

/* end of loop, uninstall the handler */
for (i = 0; trials[i]; i++)
    if (tried[i] == 0)
        free_irq(trials[i], NULL);

if (short_irq < 0)
    printk("short: probe failed %i times, giving up\n", count);

```

以下是handler的源码:

```

irqreturn_t short_probing(int irq, void *dev_id, struct pt_regs *regs)
{
    if (short_irq == 0) short_irq = irq; /* found */
    if (short_irq != irq) short_irq = -irq; /* ambiguous */
    return IRQ_HANDLED;
}

```

若事先不知道“可能的” IRQ , 就需要探测所有空闲的中断, 所以不得不从 IRQ 0 探测到 IRQ NR_IRQS-1。

处理例程的参数及返回值

传递给一个中断处理例程的参数有: int irq、void *dev_id和 struct pt_regs *regs。

int irq (中断号): 若要打印 log 消息时, 是很有用。

void *dev_id: 一种用户数据类型 (驱动程序可用的私有数据), 传递给 request_irq的 void* 参数, 会在中断发生时作为参数传给处理例程。我们通常传递一个指向设备数据结构的指针到 dev_id 中, 这样一个管理若干相同设备的驱动在中断处理例程中不需要任何额外的代码, 就可以找出哪个设备产生了当前的中断事件。

struct pt_regs *regs很少用到。

中断处理例程的典型使用如下:

```

static irqreturn_t sample_interrupt(int irq, void *dev_id, struct pt_regs *regs)
{
    struct sample_dev *dev = dev_id;
    /* now `dev' points to the right hardware item */
    /* .... */
}

```

和这个处理例程关联的打开代码如下:

```

static void sample_open(struct inode *inode, struct file *filp)

```

```

{
    struct sample_dev *dev = hwinfo + MINOR(inode->i_rdev);
    request_irq(dev->irq, sample_interrupt, 0 /* flags */, "sample", dev /* dev_id
*/);

    /*...*/
    return 0;
}

```

中断处理例程应当返回一个值指示是否真正处理了一个中断。如果处理例程发现设备确实需要处理，应当返回 `IRQ_HANDLED`；否则返回值 `IRQ_NONE`。以下宏可产生返回值：

```
IRQ_RETVAL(handled) /*若要处理中断，handled应是非零*/
```

有位网友在处理返回值是按惯例 `return 0;`，导致了oops。吸取经验教训，我们应特别注意这种返回值，以下是有关中断处理例程的返回值的内核定义（`#include <linux/irqreturn.h>`），看了就知道导致oops的原因了，以后应多多注意：

```

typedef int irqreturn_t;
#define IRQ_NONE      (0)
#define IRQ_HANDLED   (1)
#define IRQ_RETVAL(x) ((x) != 0)

```

实现中断处理例程

中断处理例程唯一的特别之处在中断时运行，它能做的事情受到了一些限制。这些限制与我们在内核定时器上看到的相同：

- （1）中断处理例程不能与用户空间传递数据，因为它不在进程上下文执行；
- （2）中断处理例程也不能做任何可能休眠的事情，例如调用 `wait_event`，使用除 `GFP_ATOMIC` 之外任何东西来分配内存，或者锁住一个信号量；
- （3）处理者不能调用 `schedule()`。

中断处理例程的作用是将关于中断接收的信息反馈给设备并根据被服务的中断的含义读、写数据。中断处理例程第一步常常包括清除设备的一个中断标志位，大部分硬件设备在清除“中断挂起”位前不会再产生中断。这也要根据硬件的工作原理决定，这一步也可能需要在最后做而不是开始；这里没有通用的规则。一些设备不需要这步，因为它们没有一个“中断挂起”位；这样的设备是少数。

一个中断处理的典型任务是：如果中断通知它所等待的事件已经发生（例如新数据到达），就会唤醒休眠在设备上的进程。

不管是快速或慢速处理例程，程序员应编写执行时间尽可能短的处理例程。 如果需要进行长时间计算，最好的方法是使用 `tasklet` 或者 `workqueue` 在一个更安全的时间来调度计算任务。

启用和禁止中断

有时设备驱动必须在一段时间（希望较短）内阻塞中断发生。并必须在持有一个自旋锁时阻塞中断，以避免死锁系统。注意：应尽量少禁止中断，即使是在设备驱动中，且这个技术不应当用于驱动中的互斥机制。

禁止单个中断

有时（但是很少！）一个驱动需要禁止一个特定中断。但不推荐这样做，特别是不能禁止共享中断（在现代系统中，共享的中断是很常见的）。内核提供了 3 个函数，是内核 API 的一部分，声明在 `<asm/irq.h>`：

```

void disable_irq(int irq); /*禁止给定的中断，并等待当前的中断处理例程结束。如果调用
disable_irq 的线程持有任何中断处理例程需要的资源（例如自旋锁），系统可能死锁*/
void disable_irq_nosync(int irq); /*禁止给定的中断后立即返回（可能引入竞态）*/
void enable_irq(int irq);

```

调用任一函数可能更新在可编程控制器(PIC)中的特定 `irq` 的掩码，从而禁止或使能所有处理器特定的IRQ。这些函数的调用能够嵌套，即如果 `disable_irq` 被连续调用 2 次，则需要 2 个 `enable_irq` 重新使能 IRQ。可以在中断处理例程中调用这些函数，但在处理某个IRQ时再打开它是不好的做法。

禁止所有中断

在 2.6 内核，可使用下面 2 个函数中的任一个（定义在 `<asm/system.h>`）关闭当前处理器上所有中断：

```

void local_irq_save(unsigned long flags); /*在保存当前中断状态到 flags 之后禁止中断*/
void local_irq_disable(void); /* 关闭中断而不保存状态*/

```



```
/*如果调用链中有多个函数可能需要禁止中断, 应使用 local_irq_save*/  
/*打开中断使用:*/  
void local_irq_restore(unsigned long flags);  
void local_irq_enable(void);  
/*在 2.6 内核, 没有方法全局禁用整个系统上的所有中断*/
```

顶半部和底半部

中断处理需要很快完成并且不使中断阻塞太长, 所以中断处理的一个主要问题是如何在处理例程中完成耗时的任务。

Linux (连同许多其他系统)通过将中断处理分为两部分来解决这个问题:

“**顶半部**”是实际响应中断的例程 (request_irq 注册的那个例程)。

“**底半部**”是被顶半部调度, 并在稍后更安全的时间内执行的函数。

他们最大的不同在底半部处理例程执行时, 所有中断都是打开的 (这就是所谓的在更安全的时间内运行)。典型的情况是: 顶半部保存设备数据到一个设备特定的缓存并调度它的底半部, 最后退出: 这个操作非常快。底半部接着进行任何其他需要的工作。这种方式的好处是在底半部工作期间, 顶半部仍然可以继续为新中断服务。

Linux 内核有 2 个不同的机制可用来实现底半部处理:

- (1) tasklet (首选机制), 它非常快, 但是所有的 tasklet 代码必须是原子的;
- (2) 工作队列, 它可能有更高的延时, 但允许休眠。

tasklet和工作队列在《时间、延迟及延缓操作》已经介绍过, 具体的实现代码请看实验源码!

中断共享

Linux 内核支持在所有总线上中断共享。

安装共享的处理例程

通过 request_irq 来安装共享中断与非共享中断有 2 点不同:

- (1) 当request_irq 时, flags 中必须指定SA_SHIRQ 位;
- (2) dev_id 必须唯一。任何指向模块地址空间的指针都行, 但 dev_id 绝不能设置为 NULL。

内核为每个中断维护一个中断共享处理例程列表, dev_id 就是区别不同处理例程的签名。释放处理例程通过执行free_irq实现。 dev_id 用来从这个中断的共享处理例程列表中选择正确的处理例程来释放, 这就是为什么 dev_id 必须是唯一的。

请求一个共享的中断时, 如果满足下列条件之一, 则request_irq 成功:

- (1) 中断线空闲;
- (2) 所有已经注册该中断信号线的处理例程也标识了IRQ是共享。

一个共享的处理例程必须能够识别自己的中断, 并且在自己的设备没有被中断时快速退出 (返回IRQ_NONE)。

共享处理例程没有探测函数可用, 但使用的中断信号线是空闲时标准的探测机制才有效。

一个使用共享处理例程的驱动需要小心: 不能使用 enable_irq 或 disable_irq, 否则, 对其他共享这条线的设备就无法正常工作了。即便短时间禁止中断, 另一设备也可能产生延时而为设备和其用户带来问题。所以程序员必须记住: 他的驱动并不是独占这个IRQ, 它的行为应当比独占这个中断线更加"社会化"。

中断驱动的 I/O

当与驱动程序管理的硬件间的数据传送可能因为某种原因而延迟, 驱动编写者应当实现缓存。一个好的缓存机制需采用中断驱动的 I/O, 一个输入缓存在中断时被填充, 并由读取设备的进程取走缓冲区的数据, 一个输出缓存由写设备的进程填充, 并在中断时送出数据。

为正确进行中断驱动的数据传送, 硬件应能够按照下列语义产生中断:

输入: 当新数据到达时并处理器准备好接受时, 设备中断处理器。

输出: 当设备准备好接受新数据或确认一个成功的数据传送时, 设备产生中断。

ARM9开发板实验

这次的实验和硬件相关, 利用了友善之臂SBC2440V4提供的四个中断按键。

K1: 中断+tasklet

K2: 中断+工作队列

K3: 中断+共享工作队列

K4: 中断+linux内核缓冲kfifo

具体的实现方法请看源码!

中断模块源码: [IO_irq.tar.gz](#)

测试程序: [IO_irq_test.tar.gz](#)

实验数据:

```
[Tekkaman2440@SBC2440V4]#cd /lib/modules/
[Tekkaman2440@SBC2440V4]#insmod IO_irq.ko
[Tekkaman2440@SBC2440V4]#cat /proc/devices
Character devices:
 1 mem
 2 pty
 3 ttyp
 4 /dev/vc/0
 4 tty
 4 ttyS
 5 /dev/tty
 5 /dev/console
 5 /dev/ptmx
 7 vcs
10 misc
13 input
14 sound
81 video4linux
89 i2c
90 mtd
116 alsa
128 ptm
136 pts
153 spi
180 usb
189 usb_device
204 s3c2410_serial
252 IO_irq
253 usb_endpoint
254 rtc

Block devices:
 1 ramdisk
256 rfd
 7 loop
31 mtblock
93 nftl
96 inftl
179 mmc
[Tekkaman2440@SBC2440V4]#mknod -m 666 IO_irq c 252 0
[Tekkaman2440@SBC2440V4]#/tmp/IO_irq_test
IO_irq: opened !
IO_irq: the module can not lseek!
please input the command :
*****KEY = 1*****
NO prevkey
now jiffies =0x00059ae5
count = 1
```

```
*****KEY = 1 END*****

*****key1_tasklet_start*****
time:00059ae7 delta: 2 inirq:1 pid: 0 cpu:0 command:swapper

*****key1_tasklet_end*****

*****KEY = 2*****
prevkey=1 at 0x00059ae5
NO prekey2
now jiffies =0x00059c54
count = 1
result = 1
*****KEY = 2 END*****

*****key2_workqueue_start*****
time:00059c5f delta: 11 inirq:0 pid:832 cpu:0 command:tekkamanwork/0

*****key2_workqueue_end*****

*****KEY = 3*****
prevkey=2 at 0x00059c54
NO prekey3
now jiffies =0x00059f63
count = 1
result = 1
*****KEY = 3 END*****

*****key3_workqueue_start*****
time:00059f66 delta: 3 inirq:0 pid: 5 cpu:0 command:events/0

*****key3_workqueue_end*****

IO_irq: Invalid input ! only 1、2、3、4、5、6、7、8、q !

please input the command :6
IO_irq: ioctl IO_KFIFO_SIZE len=414
please input the command :8

*****KEY = 4*****
prevkey=3 at 0x00059f63
NO prekey4
now jiffies =0x0005a2fd
count = 1
*****KEY = 4 END*****

*****KEY = 4*****
prevkey=4 at 0x0005a2fd
prekey4 at 0x0005a2fd
now jiffies =0x0005a304
count = 2
*****KEY = 4 END*****

*****KEY = 4*****
prevkey=4 at 0x0005a304
prekey4 at 0x0005a304
now jiffies =0x0005a304
count = 3
*****KEY = 4 END*****

please input the command :6
```



```
IO_irq: ioctl IO_KFIFO_SIZE len=142
please input the command :7
IO_status= 1f !
IO_irq: ioctl IO_KFIFO_RESET
please input the command :6
IO_irq: ioctl IO_KFIFO_SIZE len=0
please input the command :8
please input the command :q
IO_irq: release !
[Tekkaman2440@SBC2440V4]#
```

阅读 (7700) | 评论 (3) | 转发 (36) |

上一篇: Linux设备驱动程序学习（10）-时间、延迟及延缓操作


下一篇: Linux设备驱动程序学习（3-补）-Linux中的循环缓冲区

0

相关热门文章

- | | | |
|-------------------------|----------------------------|-----------------------|
| 图像处理JPEGCodec类已经从Jdk... | linux 常见服务端口 | 移植 ushare 到开发板 |
| 客户管理系统的可行性分析... | 【ROOTFS搭建】busybox的httpd... | 系统提供的库函数存在内存泄漏... |
| 强类型语言、弱类型语言、静态... | xmanager 2.0 for linux配置 | linux虚拟机 求教 |
| 车险理赔节后未现高峰 鞭炮炸... | 什么是shell | 初学UNIX环境高级编程的，关于... |
| MySQL Slave异常关机的处理 ... | linux socket的bug?? | chinaunix博客什么时候可以设... |


给主人留下些什么吧！~~

- 

ylgryq

2011-11-18 18:08:58

tekkamanninja: 不好意思~~那个代码是我刚开始学Linux的时候写的，4年前的事了，写得实在是烂，我自己都看不下去了。
我随便看了下，可能是当时觉得只有一个设备，就弄成全局变....
能看到您的回复真的很高兴。呵呵 您过谦了，您的代码是非常好的参考的。真的给初学者帮了大忙。比如我.....希望能继续拜读您的文章。祝您工作顺利了


回复 | 举报
- 

tekkamanninja

2011-11-17 21:55:42

ylgryq: 您好！一直在拜读您的文章。确实是受益匪浅。慢慢一步步跟着做，感觉学到了很多。每次有不会写的地方都来您这里找参考。但是这个中断处理程序中，您用了一个全局....
不好意思~~那个代码是我刚开始学Linux的时候写的，4年前的事了，写得实在是烂，我自己都看不下去了。
我随便看了下，可能是当时觉得只有一个设备，就弄成全局变量了，但是如果是用来锁设备的，应该在设备的结构体中，不知道我说的对不对。

那个代码只是随便做实验用的，但是结构上很差，以后有空我再换了它，喜欢不会教坏你才是，呵呵😊

回复 | 举报
- 

ylgryq

2011-11-17 21:13:29

您好！一直在拜读您的文章。确实是受益匪浅。慢慢一步步跟着做，感觉学到了很多。每次有不会写的地方都来您这里找参考。但是这个中断处理程序中，您用了一个全局的spinlock 和 atomic_t来避免竞争，觉得这里似乎所不上设备吧。是不是应该放在struct IO_irq_dev里呢。。。稍有疑惑，就来问问，望您不介意。。

回复 | 举报

评论热议

请登录后再评论。

[登录](#) [注册](#)



[关于我们](#) | [关于IT168](#) | [联系方式](#) | [广告合作](#) | [法律声明](#) | [免费注册](#)

Copyright 2001-2010 ChinaUnix.net All Rights Reserved 北京皓辰网域网络信息技术有限公司. 版权所有

感谢所有关心和支持过ChinaUnix的朋友们

京ICP证041476号 京ICP证060528号