ARM+LINUX

yuyunbo.blog.chinaunix.net

To follow the path: look to the master, follow the master, walk with the master, see through the master, become the master.

个人资料

leon_yu

微博 论坛

发纸条 打招呼 加关注 加好友

- 博客访问：45627
- 博文数量：142
- 博客积分：5038
- 博客等级：上校
- 关注人气： 20
- 注册时间：2010-10-05 15:13:30

文章分类

driver(45)
tools(18)
Linux(35)
ARM(2)
未分类博文(2)
订阅我的博客

- 订阅
- 订阅到 鲜果
- 订阅到 抓虾
- 订阅到 Google

字体大小：大 中 小博文
ARM+Linux中断系统详细分析 (2012-08-30 16:54)
标签: 异常向量表 中断优先级 中断号 中断嵌套 共享中断 分类： 内核研究

ULK第四章里明确讲到“Linux实现了一种没有优先级的中断模型”，并且“Linux中断和异常都支持嵌套”。这个我不太理解了，这两种说法都与我以前的理解刚好相反，核对了原书，翻译没有错。
Linux中断系统到底是否支持优先级，可否嵌套，中断号又是怎么来确定的，中断产生时又是如何一步步执行到中断处理函数的。为了彻底搞懂Linux中断系统，我决定从最原始材料出发，一探究竟。（s3c2440+linux2.6.21）

先来看看ARM的硬件执行流程

异常是ARM处理器模式分类，ARM有七种运行模式USR,SYS,SVC,IRQ,FIQ,UND,ABT

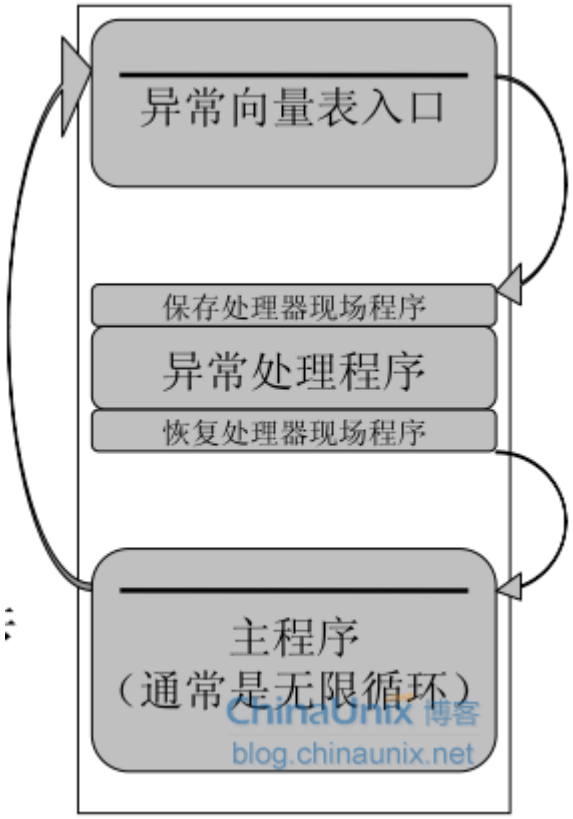
非特权模式	特权模式					
非异常模式		异常模式				
用户模式 **USR**	系统模式 **SYS**	中断模式 **IRQ**	快中断模式 **FIQ**	管理模式 **SVC**	终止模式 **ABT**	未定义模式 **UND**

五种异常模式：SVC,IRQ,FIQ,UND,ABT
中断模式是ARM异常模式之一（IRQ模式，FIQ模式），是一种异步事件，如外部按键产生中断，内部定时器产生中断，通信数据口数据收发产生中断等。
1.当一个异常产生时，以FIQ为例，CPU切入FIQ模式时，
①将原来执行程序的下一条指令地址保存到LR中，就是将R14保存到R14_fiq里面。
②拷贝CPSR到SPSR_fiq。
③改变CPSR模式位的值，改到FIQ模式。
④改变PC值，将其指向相应的异常处理向量表。
离开异常处理的时候，
①将LR（R14_fiq）赋给PC。
②将SPSR（SPSR_fiq）拷贝到CPSR。
③清除中断禁止标志（如果开始时置位了）。

ARM一般在某个固定地址中有一个异常向量表，比如0x0

中断向量地址	异常中断类型	异常中断模式	优先级（6 最低）
0x0	复位	特权模式（SVC）	1
0x4	未定义中断	未定义指令中止模式（Undef)	6
0x8	软件中断（SWI）	特权模式（SVC）	6
0x 0c	指令预取中止	中止模式	5
0x10	数据访问中止	中止模式	2
0x14	保留	未使用	未使用
0x18	外部中断请求（IRQ）	外部中断（IRQ）模式	4
0x 1c	快速中断请求（FIQ）	快速中断（FIQ）模式	3

当一个外部IRQ中断产生时
①处理器切换到IRQ模式
②PC跳到0x18处运行，因为这是IRQ的中断入口。
③通过0x18：LDR PC, IRQ_ADDR，跳转到相应的中断服务程序。这个中断服务程序就要确定中断源，每个中断源会有自己独立的中断服务程序。
④得到中断源，然后执行相应中断服务程序
⑤清除中断标志，返回



这就是一个外部中断完整的执行流程了，下面以具体寄存器来更具体的了解ARM的中断机制。
假设ARM核有两个中断引脚，一根是irq pin,一根是fiq pin，正常情况下，ARM核只是机械地随着PC指示去执行，当CPSR中的I位和F位都为1时，IRQ和FIQ都处于禁止状态，这时候无论发什么信号，ARM都不会理睬。
当I位或F位为0时，irq pin有中断信号过来时，ARM当前工作就会被打断，切换到IRQ模式，并且跳转到异常向量表的中断入口0x18，SRCPND中相应位置1，经过检查中断优先级寄存器以及屏蔽寄存器，确定中断源，INTPND相应位置1(经过仲裁，只有一位置1)，这过程由ARM自动完成。0x18存放的是总的中断处理函数，在这个函数里，可以建立一个二级中断向量表,先清除SRCPND相应位，然后根据中断源执行相应中断服务程序，清除INTPND，返回。
及时清除中断 Pending 寄存器的标志位是为了避免两个问题：①发生中断返回后，立即又被中断，不断的重复响应②丢失中断处理过程中发生的中断，返回后不响应。
在某个IRQ中断程序执行过程中，有另外一个外部IRQ中断产生，会将SRCPND相应位置1，等该中断服务执行完，经过仲裁决定下一个要响应的中断。但是假如当产生的是FIQ,则保存当前IRQ的现场，嵌套响应FIQ，FIQ服务程序执行完，再继续执行IRQ服务。那么当一个FIQ正在服务，产生另外一个FIQ，会怎样呢，答案是不会被打断，跟IRQ一样等当前中断服务完成，再仲裁剩余需要相应的中断。
所以得出这样的结论：
①关于中断嵌套：IRQ模式只能被FIQ模式打断，FIQ模式下谁也打不断。

②关于优先级：ARM核对中断优先级，有明确的可编程管理。

下面再来看看Linux对ARM是怎么处理的，但记住一个前提：Linux对ARM的硬件特性可以取舍，但不可更改。

1.建立异常向量表：

系统从arch/arm/kernel/head.S的ENTRY(stext)开始执行，__lookup_processor_type检查处理器ID，__lookup_machine_type检查机器ID，__create_page_tables创建页表，启动MMU，然后由arch/arm/kernel/head_common.S 跳到start_kernel()->trap_init()

点击(此处)折叠或打开

```
1.  void __init trap_init(void)
2.  {
3.      unsigned long vectors = CONFIG_VECTORS_BASE;
4.      …
5.      memcpy((void *)vectors, __vectors_start, __vectors_end - __vectors_start);
6.      memcpy((void *)vectors + 0x200, __stubs_start, __stubs_end - __stubs_start);
7.      memcpy((void *)vectors + 0x1000 - kuser_sz, __kuser_helper_start, kuser_sz);
8.      …
9.  }
10. #define CONFIG_VECTORS_BASE 0xffff0000
```

CONFIG_VECTORS_BASE在autoconf.h定义，在ARM V4及V4T以后的大部分处理器中，中断向量表的位置可以有两个位置：一个是0，另一个是0xffff0000。可以通过CP15协处理器c1寄存器中V位(bit[13])控制。V和中断向量表的对应关系如下：

V=0 ~ 0x00000000~0x0000001C

V=1 ~ 0xffff0000~0xffff001C

__vectors_end 至 __vectors_start之间为异常向量表，位于arch/arm/kernel/entry-armv.S

点击(此处)折叠或打开

```
1.  .globl __vectors_start
2.  __vectors_start:
3.  swi SYS_ERROR0
4.  b vector_und + stubs_offset//复位异常
5.  ldr pc, .LCvswi + stubs_offset //未定义异常
6.  b vector_pabt + stubs_offset//软件中断异常
7.  b vector_dabt + stubs_offset//数据异常
8.  b vector_addrexcptn + stubs_offset//保留
9.  b vector_irq + stubs_offset //普通中断异常
10. b vector_fiq + stubs_offset//快速中断异常
11.
12. .globl __vectors_end
```

```
13.  __vectors_end:
```

stubs_offset值如下：
.equ stubs_offset, __vectors_start + 0x200 - __stubs_start
stubs_offset是如何确定的呢？（引用网络上的一段比较详细的解释）
当汇编器看到B指令后会把要跳转的标签转化为相对于当前PC的偏移量（±32M）写入指令码。从上面的代码可以看到中断向量表和stubs都发生了代码搬移，所以如果中断向量表中仍然写成b vector_irq，那么实际执行的时候就无法跳转到搬移后的vector_irq处，因为指令码里写的是原来的偏移量，所以需要把指令码中的偏移量写成搬移后的。我们把搬移前的中断向量表中的irq入口地址记irq_PC,它在中断向量表的偏移量就是irq_PC-vectors_start, vector_irq在stubs中的偏移量是vector_irq-stubs_start，这两个偏移量在搬移前后是不变的。搬移后 vectors_start在0xffff0000处，而stubs_start在0xffff0200处，所以搬移后的vector_irq相对于中断向量中的中断入口地址的偏移量就是，200+vector_irq在stubs中的偏移量再减去中断入口在向量表中的偏移量，即200+ vector_irq-stubs_start-irq_PC+vectors_start = (vector_irq-irq_PC) + vectors_start+200-stubs_start,对于括号内的值实际上就是中断向量表中写的vector_irq，减去irq_PC是由汇编器完成的，而后面的 vectors_start+200-stubs_start就应该是stubs_offset，实际上在entry-armv.S中也是这样定义的。

2.中断响应

当有外部中断产生时，跳转到异常向量表的“b vector_irq + stubs_offset //普通中断异常”
进入异常处理函数，跳转的入口位置 arch\arm\kernel\entry-armv.S 代码简略如下

点击(此处)折叠或打开

```
 1.  .globl __stubs_start
 2.  __stubs_start:
 3.  /*
 4.  * Interrupt dispatcher
 5.  */
 6.  vector_stub irq, IRQ_MODE, 4
 7.  .long __irq_usr @ 0 (USR_26 / USR_32)
 8.  .long __irq_invalid @ 1 (FIQ_26 / FIQ_32)
 9.  .long __irq_invalid @ 2 (IRQ_26 / IRQ_32)
10.  .long __irq_svc @ 3 (SVC_26 / SVC_32)
11.
12.  vector_stub dabt, ABT_MODE, 8
13.  vector_stub pabt, ABT_MODE, 4
14.  vector_stub und, UND_MODE
15.  /*
16.  * Undefined FIQs
17.  */
18.  vector_fiq:
19.  disable_fiq
20.  subs pc, lr, #4
21.  vector_addrexcptn:
22.  b vector_addrexcptn
```

vector_stub是个函数调用宏，根据中断前的工作模式决定进入__irq_usr,__irq_svc。这里入__irq_svc，同时看到这里FIQ产生时，系统未做任何处理，直接返回，即Linux没有提供对FIQ的支持，继续跟进代码

点击(此处)折叠或打开

```
1.  __irq_svc:
2.  svc_entry
3.  …
4.  irq_handler
```

svc_entry是一个宏，主要实现了将SVC模式下的寄存器、中断返回地址保存到堆栈中。然后进入最核心的中断响应函数irq_handler，irq_handler实现过程arch\arm\kernel\entry-armv.S

点击(此处)折叠或打开

```
1.  .macro irq_handler
2.  get_irqnr_preamble r5, lr
3.  1: get_irqnr_and_base r0, r6, r5, lr @判断中断号，通过R0返回，3.5节有实现过程
4.  movne r1, sp
5.  @
6.  @ routine called with r0 = irq number, r1 = struct pt_regs *
7.  @
8.  adrne lr, 1b
9.  bne asm_do_IRQ @进入中断处理。
10.  ……
11.  .endm
```

get_irqnr_and_base中断号判断过程，include/asm/arch-s3c2410/entry-macro.s

点击(此处)折叠或打开

```
1.  .macro get_irqnr_and_base, irqnr, irqstat, base, tmp
2.  mov \base, #S3C24XX_VA_IRQ
3.  @@ try the interrupt offset register, since it is there
4.  ldr \irqstat, [ \base, #INTPND ]
5.  teq \irqstat, #0
6.  beq 1002f
7.  ldr \irqnr, [ \base, #INTOFFSET ] @通过判断INTOFFSET寄存器得到中断位置
8.  …
9.  @@ we have the value
10.  1001:
11.  adds \irqnr, \irqnr, #IRQ_EINT0 @加上中断号的基准数值，得到最终的中断号，注意：此时没有考虑
```

```
    子中断的具体情况。IRQ_EINT0在include/asm/arch-s3c2410/irqs.h中定义.从这里可以看出，中断号
    的具体值是有平台相关的代码决定的，和硬件中断挂起寄存器中的中断号是不等的。
12.  1002:
13.  @@ exit here, Z flag unset if IRQ
14.  .endm
```

asm_do_IRQ实现过程，arch/arm/kernel/irq.c

点击(此处)折叠或打开

```
 1.  asmlinkage void asm_do_IRQ(unsigned int irq, struct pt_regs *regs)
 2.  {
 3.      struct pt_regs *old_regs = set_irq_regs(regs);
 4.      struct irq_desc *desc = irq_desc + irq;//根据中断号，找到响应的irq_desc
 5.
 6.      /*
 7.       * Some hardware gives randomly wrong interrupts. Rather
 8.       * than crashing, do something sensible.
 9.       */
10.      if (irq >= NR_IRQS)
11.          desc = &bad_irq_desc;
12.
13.      irq_enter();
14.
15.      desc_handle_irq(irq, desc);//根据irq和desc进入中断处理
16.
17.      /* AT91 specific workaround */
18.      irq_finish(irq);
19.
20.      irq_exit();
21.      set_irq_regs(old_regs);
22.  }
23.  static inline void desc_handle_irq(unsigned int irq, struct irq_desc *desc)
24.  {
25.      desc->handle_irq(irq, desc);//中断处理
26.  }
```

上述asmlinkage void __exception asm_do_IRQ(unsigned int irq, struct pt_regs *regs)使用了asmlinkage标识。那么这个标识的含义如何理解呢？
该符号定义在kernel/include/linux/linkage.h中，如下所示：

#include <asm/linkage.h>//各个具体处理器在此文件中定义asmlinkage

点击(此处)折叠或打开

```
#ifdef __cplusplus
        #define CPP_ASMLINKAGE extern "C"
        #else
        #define CPP_ASMLINKAGE
        #endif
#ifndef asmlinkage//如果以前没有定义asmlinkage
        #define asmlinkage CPP_ASMLINKAGE
        #endif
```

对于ARM处理器的<asm/linkage.h>，没有定义asmlinkage，所以没有意义（不要以为参数是从堆栈传递的，对于ARM平台来说还是符合ATPCS过程调用标准，通过寄存器传递的）。
但对于X86处理器的<asm/linkage.h>中是这样定义的：
#define asmlinkage CPP_ASMLINKAGE __attribute__((regparm(0)))
表示函数的参数传递是通过堆栈完成的。
中断处理过程代码就跟到这了，那么最后一个问题desc->handle_irq(irq, desc);是怎么跟我们注册的中断函数相关联的呢？再从中断模型注册入手：
中断相关的数据结构：在include/asm/arch/irq.h中定义。
irq_desc [] 是一个指向irq_desc_t结构的数组，irq_desc_t结构是各个设备中断服务例程的描述符。Irq_desc_t结构体中的成员action指向该中断号对应的irqaction结构体链表。Irqaction结构体定义在include/linux/interrupt.h中，如下：

点击(此处)折叠或打开

```
truct irqaction {
irq_handler_t handler; //中断处理函数，注册时提供
unsigned long flags; //中断标志，注册时提供
cpumask_t mask; //中断掩码
const char *name; //中断名称
void *dev_id; //设备id，本文后面部分介绍中断共享时会详细说明这个参数的作用
struct irqaction *next; //如果有中断共享，则继续执行，
int irq; //中断号，注册时提供
struct proc_dir_entry *dir; //指向IRQn相关的/proc/irq/n目录的描述符
};
```

在注册中断号为irq的中断服务程序时，系统会根据注册参数封装相应的irqaction结构体。并把中断号为irq的irqaction结构体写入irq_desc [irq]->action。这样就把设备的中断请求号与该设备的中断服务例程irqaction联系在一起了。当CPU接收到中断请求后，就可以根据中断号通

过irq_desc []找到该设备的中断服务程序。

3. 中断共享的处理模型

共享中断的不同设备的iqraction结构体都会添加进该中断号对应的irq_desc结构体的action成员所指向的irqaction链表内。当内核发生中断时，它会依次调用该链表内所有的handler函数。因此，若驱动程序需要使用共享中断机制，其中断处理函数必须有能力识别是否是自己的硬件产生了中断。通常是通过读取该硬件设备提供的中断flag标志位进行判断。也就是说不是任何设备都可以做为中断共享源的，它必须能够通过的它的中断flag判断出是否发生了中断。

中断共享的注册方法是：

int request_irq(unsigned int irq, irq_handler_t handler,IRQF_SHARED, const char *devname, void *dev_id)

很多权威资料中都提到，中断共享注册时的注册函数中的dev_id参数是必不可少的，并且dev_id的值必须唯一。那么这里提供唯一的dev_id值的究竟是做什么用的？

根据我们前面中断模型的知识，可以看出发生中断时，内核并不判断究竟是共享中断线上的哪个设备产生了中断，它会循环执行所有该中断线上注册的中断处理函数（即irqaction->handler函数）。因此irqaction->handler函数有责任识别出是否是自己的硬件设备产生了中断，然后再执行该中断处理函数。通常是通过读取该硬件设备提供的中断flag标志位进行判断。那既然kernel循环执行该中断线上注册的所有irqaction->handler函数，把识别究竟是哪个硬件设备产生了中断这件事交给中断处理函数本身去做，那request_irq的dev_id参数究竟是做什么用的？

很多资料中都建议将设备结构指针作为dev_id参数。在中断到来时，迅速地根据硬件寄存器中的信息比照传入的dev_id参数判断是否是本设备的中断，若不是，应迅速返回。这样的说法没有问题，也是我们编程时都遵循的方法。但事实上并不能够说明为什么中断共享必须要设置dev_id。

下面解释一下dev_id参数为什么必须的，而且是必须唯一的。

当调用free_irq注销中断处理函数时（通常卸载驱动时其中断处理函数也会被注销掉），因为dev_id是唯一的，所以可以通过它来判断从共享中断线上的多个中断处理程序中删除指定的一个。如果没有这个参数，那么kernel不可能知道给定的中断线上到底要删除哪一个处理程序。

注销函数定义在Kernel/irq/manage.c中定义：

void free_irq(unsigned int irq, void *dev_id)

4.S3C2410子中断的注册的实现

前面判断中断号的方法，可以看到只是通过S3C2410中断控制器中的INTOFFSET寄存器来判断的。对于INTPND中的EINT4_7、EINT8_23、INT_UART0、INT_ADC 等带有子中断的向量，INTOFFSET无法判断出具体的中断号。平台留给我们的注册方法如下：

在include/asm/arch/irqs.h中有类似如下定义：

点击(此处)折叠或打开

```
1.  /* interrupts generated from the external interrupts sources */
2.  #define IRQ_EINT4 S3C2410_IRQ(32) /* 48 */
3.  #define IRQ_EINT5 S3C2410_IRQ(33)
4.  #define IRQ_EINT6 S3C2410_IRQ(34)
5.  #define IRQ_EINT7 S3C2410_IRQ(35)
6.  #define IRQ_EINT8 S3C2410_IRQ(36)
7.  #define IRQ_EINT9 S3C2410_IRQ(37)
8.  #define IRQ_EINT10 S3C2410_IRQ(38)
```

```
 9.  #define IRQ_EINT11 S3C2410_IRQ(39)
10.  #define IRQ_EINT12 S3C2410_IRQ(40)
11.  #define IRQ_EINT13 S3C2410_IRQ(41)
12.  #define IRQ_EINT14 S3C2410_IRQ(42)
13.  #define IRQ_EINT15 S3C2410_IRQ(43)
14.  #define IRQ_EINT16 S3C2410_IRQ(44)
15.  #define IRQ_EINT17 S3C2410_IRQ(45)
16.  #define IRQ_EINT18 S3C2410_IRQ(46)
17.  #define IRQ_EINT19 S3C2410_IRQ(47)
18.  #define IRQ_EINT20 S3C2410_IRQ(48) /* 64 */
19.  #define IRQ_EINT21 S3C2410_IRQ(49)
20.  #define IRQ_EINT22 S3C2410_IRQ(50)
21.  #define IRQ_EINT23 S3C2410_IRQ(51)
```

可以看到平台为每种子中断都定义了中断号，如果你想实现EINT10的中断注册，直接按照IRQ_EINT10这个中断号注册都可以了。那么平台代码是如何实现这部分中断注册的呢?

5.S3C2410子中断注册问题的解决

点击(此处)折叠或打开

```
 1.  /*arch/arm/plat-s3c24xx/irq.c*/
 2.  void __init s3c24xx_init_irq(void)
 3.  {……
 4.  set_irq_chained_handler(IRQ_EINT4t7, s3c_irq_demux_extint4t7);
 5.  set_irq_chained_handler(IRQ_EINT8t23, s3c_irq_demux_extint8);
 6.  set_irq_chained_handler(IRQ_UART0, s3c_irq_demux_uart0);
 7.  set_irq_chained_handler(IRQ_UART1, s3c_irq_demux_uart1);
 8.  set_irq_chained_handler(IRQ_UART2, s3c_irq_demux_uart2);
 9.  set_irq_chained_handler(IRQ_ADCPARENT, s3c_irq_demux_adc);
10.  ……
11.  }
```

平台在初始化时会调用到s3c24xx_init_irq，在此函数中实现了对EINT4_7、EINT8_23、INT_UART0、INT_ADC等中断的注册。下面看看这些带有子中断的中断号对应的处理函数的内容。以IRQ_EINT4t7为例，其它情况类似。

点击(此处)折叠或打开

```
 1.  /*arch/arm/plat-s3c24xx/irq.c*/
 2.  s3c_irq_demux_extint4t7(unsigned int irq,
 3.  struct irq_desc *desc)
 4.  {
 5.  unsigned long eintpnd = __raw_readl(S3C24XX_EINTPEND);
```

```
 6.   unsigned long eintmsk = __raw_readl(S3C24XX_EINTMASK);
 7.   eintpnd &= ~eintmsk;
 8.   eintpnd &= 0xff; /* only lower irqs */
 9.   /* eintpnd中可以有多个位同时置1，这一点和intpnd的只能有1个位置1是不一样的 */
10.   while (eintpnd) { //循环执行所有置位的子中断
11.   irq = __ffs(eintpnd); //算出第一个不为0的位，类似arm v5后的clz前导0的作用
12.   eintpnd &= ~(1<<irq);//清除相应的位
13.   irq += (IRQ_EINT4 - 4);//算出对应的中断号
14.   desc_handle_irq(irq, irq_desc + irq);//执行对应子中断的注册函数
15.   }
16.   }
```

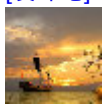
从上面的函数可以看出子中断是如何注册及被调用到的。有人可能会问为何不在include/asm/arch-s3c2410/entry-macro.s 文件中get_irqnr_and_base函数判断中断号时，直接算出对应的子中断号，就可以直接找到子中断处理了呢?
原因是: get_irqnr_and_base是平台给系统提供的函数，对于多个子中断同时置位的情况无法通过一个值返回（因为子中断中，如eintpnd是可以多个位同时置位的））。而intpnd则没有这个问题。

至此,对于s3c2440/10+linux2.6得出以下结论:
①不支持中断嵌套(因为FIQ不支持)
②有明确中断优先级(可编程)
③中断号是根据硬件特性固定的，riq号通过某种转换得到与寄存器相应位，一般在irqs.h文件定义

中断的用法见Ldd3的笔记：http://blog.chinaunix.net/uid-24708340-id-3035617.html

前一篇：深入理解Linux内核(4)---中断和异常(x86平台)

[发评论] 评论 重要提示：警惕虚假中奖信息!

- 

zhongli_i 13小时前

每次都写这么多，你好有耐心。学习了。

亲，您还没有登录,请[登录]或[注册]后再进行评论