Linux 0.01 中断部分简要及注释

Willweb 于 4.10

中断是 linux 系统中的一个重要并且复杂的组成部分，它提供给外设一种运行 cpu 的方式，一个完善的支持外设扩充的系统中，由于有着多种多样的外部设备，这些设备通过中断方式来平衡速度和获得资源。

中断可以分为 cpu 的内部中断和外设的外部中断两种，cpu 的内部中断又可以叫做异常，异常的主要作用是报告一些程序运行过程中的错误和处理缺页中断（page_fault），这些异常都是通过 set_trap_gate 来设置的，intel 的 cpu 为 trap 保留了 32 个，具体在 idt 中的号是从 00 到 20。外部设备中断是通过 set_intr_gate 来设置的，它从 21 开始到 ff。另外还有一种软件中断，也就是前一次讲到的系统调用，它是在用户端可以调用的。

Linux2。4 的中断处理比较复杂，因为涉及的设备比较多，同时顾及到许多处理的问题，但是 0。01 由于比较简单，没有处理很复杂的操作，它里面主要做了几个部分的工作：

1. Trap.s 将各个 cpu 的 trap 信息填入到 idt 中
2. Asm.s 处理当出现 trap 的时候后续的一系列处理
3. 另外，0。01 中的处理的外部中断有 time（0x20），串口（0x23 和 0x24），硬盘（0x2e），键盘（0x21），具体的处理函数在 rs_io.s，hd.c 和 keyboard.s 中

linux 下都是采用两片 8259 作为接收外部的中断控制器，其初始化是在 boot.s 中初始化的，具体的连接是 8259 的从片接主片的第二个中断口，其他的中断线就可以通过其他设备共享，到外设产生一个中断之后，8259 自己将 irq 号转换为中断向量（一般是加上 20），然后发给 cpu 并进行等待，当 cpu 应该之后再清 intr 线，cpu 接收到中断之后在内核堆栈中保留 irq 值和寄存器值，同时发送一个 pic 应答并执行 isr 中断服务程序。

如果多个设备共享一个中断，那么每当一个设备产生一个中断的时候 cpu 会执行所有的 isr 中断服务程序。具体的一个完整的中断产生和处理流程是：

产生中断——>cpu 应答——>查找 idt 中的对应向量——>在 gdt 中查找 idt 项的代码段——>对比当前的 cpl 和描述符的 dpl 看是否产生越级保护——>检查是否发生特权级的变化，如果是就保存 ss 和 esp，否则不保存——>保存 eflags、cs、eip 和错误码——>将 idt 对应描述符地址装入 cs 和 eip 中以便执行——>执行 irq_interrupt——>执行 do_irq——>循环执行 isr——>返回

/KERNEL/TRAPS.C 简要注释

```
/*
 * 'Traps.c' handles hardware traps and faults after we have saved some
 * state in 'asm.s'. Currently mostly a debugging-aid, will be extended
 * to mainly kill the offending process (probably by giving it a signal,
 * but possibly by killing it outright if necessary).
 */
#include <string.h>

#include <linux/head.h>
#include <linux/sched.h>
#include <linux/kernel.h>
#include <asm/system.h>
#include <asm/segment.h>

//得到段寄存器中 addr 地址的数据,返回一个字节
#define get_seg_byte(seg,addr) ({ \
register char __res; \
__asm__("push %%fs;mov %%ax,%%fs;movb %%fs:%2,%%al;pop %%fs" \
    :"=a" (__res):"0" (seg),"m" (*(addr))); \
__res;})

//返回四个字节的 addr 地址的段寄存器内容
#define get_seg_long(seg,addr) ({ \
register unsigned long __res; \
__asm__("push %%fs;mov %%ax,%%fs;movl %%fs:%2,%%eax;pop %%fs" \
    :"=a" (__res):"0" (seg),"m" (*(addr))); \
__res;})

//取得 fs 段寄存器的内容
#define _fs() ({ \
register unsigned short __res; \
__asm__("mov %%fs,%%ax":"=a" (__res):); \
__res;})

//当出现程序内部错误的时候(例如除 0 等异常),打印出 cpu 中各个寄存器的值然后退出程
序
static void die(char * str,long esp_ptr,long nr)
{
    long * esp = (long *) esp_ptr;
    int i;

//显示出错信息并打印出每个寄存器的值
```

```
    printk("%s: %04x\n\r",str,nr&0xffff);
    printk("EIP:\t%04x:%p\nEFLAGS:\t%p\nESP:\t%04x:%p\n",
        esp[1],esp[0],esp[2],esp[4],esp[3]);
    printk("fs: %04x\n",_fs());
    printk("base: %p, limit: %p\n",get_base(current->ldt[1]),get_limit(0x17));
    if (esp[4] == 0x17) {
        printk("Stack: ");
        for (i=0;i<4;i++)
            printk("%p ",get_seg_long(0x17,i+(long *)esp[3]));
        printk("\n");
    }
    str(i);
    printk("Pid: %d, process nr: %d\n\r",current->pid,0xffff & i);
    for(i=0;i<10;i++)
        printk("%02x ",0xff & get_seg_byte(esp[1],(i+(char *)esp[0])));
    printk("\n\r");
//退出程序
    do_exit(11);        /* play segment exception */
}

//异常处理程序
void do_double_fault(long esp, long error_code)
{
    die("double fault",esp,error_code);
}

void do_general_protection(long esp, long error_code)
{
    die("general protection",esp,error_code);
}

void do_divide_error(long esp, long error_code)
{
    die("divide error",esp,error_code);
}
//do_int3 处理断点的 trap,具体是将所有的寄存器值打印出来
void do_int3(long * esp, long error_code,
        long fs,long es,long ds,
        long ebp,long esi,long edi,
        long edx,long ecx,long ebx,long eax)
{
    int tr;

    __asm__("str %%ax":"=a" (tr):"0" (0));
```

```
    printk("eax\t\tebx\t\tecx\t\tedx\n\r%8x\t%8x\t%8x\t%8x\n\r",
        eax,ebx,ecx,edx);
    printk("esi\t\tedi\t\tebp\t\tesp\n\r%8x\t%8x\t%8x\t%8x\n\r",
        esi,edi,ebp,(long) esp);
    printk("\n\rds\tes\tfs\ttr\n\r%4x\t%4x\t%4x\t%4x\n\r",
        ds,es,fs,tr);
    printk("EIP: %8x   CS: %4x  EFLAGS: %8x\n\r",esp[0],esp[1],esp[2]);
}

//以下的 trap 操作转化为 die 函数操作
void do_nmi(long esp, long error_code)
{
    die("nmi",esp,error_code);
}

void do_debug(long esp, long error_code)
{
    die("debug",esp,error_code);
}

void do_overflow(long esp, long error_code)
{
    die("overflow",esp,error_code);
}

void do_bounds(long esp, long error_code)
{
    die("bounds",esp,error_code);
}

void do_invalid_op(long esp, long error_code)
{
    die("invalid operand",esp,error_code);
}

void do_device_not_available(long esp, long error_code)
{
    die("device not available",esp,error_code);
}

void do_coprocessor_segment_overrun(long esp, long error_code)
{
    die("coprocessor segment overrun",esp,error_code);
}
```

```
void do_invalid_TSS(long esp,long error_code)
{
    die("invalid TSS",esp,error_code);
}

void do_segment_not_present(long esp,long error_code)
{
    die("segment not present",esp,error_code);
}

void do_stack_segment(long esp,long error_code)
{
    die("stack segment",esp,error_code);
}

void do_coprocessor_error(long esp, long error_code)
{
    die("coprocessor error",esp,error_code);
}

void do_reserved(long esp, long error_code)
{
    die("reserved (15,17-31) error",esp,error_code);
}

//初始化 trap 信息,将前面的 32 个 idt 填入异常处理函数的地址
void trap_init(void)
{
    int i;

//系统门和陷阱门的类型都是 1111(15),而中断门类型是 1110(14),但是中断门和陷阱门的
dpl 都是 0,而系统门的 dpl 是 3
    set_trap_gate(0,&divide_error);
    set_trap_gate(1,&debug);
    set_trap_gate(2,&nmi);
    set_system_gate(3,&int3);   /* int3-5 can be called from all */
    set_system_gate(4,&overflow);
    set_system_gate(5,&bounds);
    set_trap_gate(6,&invalid_op);
    set_trap_gate(7,&device_not_available);
    set_trap_gate(8,&double_fault);
    set_trap_gate(9,&coprocessor_segment_overrun);
    set_trap_gate(10,&invalid_TSS);
```

```
    set_trap_gate(11,&segment_not_present);
    set_trap_gate(12,&stack_segment);
    set_trap_gate(13,&general_protection);
    set_trap_gate(14,&page_fault);
    set_trap_gate(15,&reserved);
    set_trap_gate(16,&coprocessor_error);
    for (i=17;i<32;i++)
        set_trap_gate(i,&reserved);
/*  __asm__("movl $0x3ff000,%%eax\n\t"
        "movl %%eax,%%db0\n\t"
        "movl $0x000d0303,%%eax\n\t"
        "movl %%eax,%%db7"
        :::"ax");*/
}
```

/KERNEL/ASM.S 具体的异常处理程序

```
/*
 * asm.s contains the low-level code for most hardware faults.
 * page_exception is handled by the mm, so that isn't here. This
 * file also handles (hopefully) fpu-exceptions due to TS-bit, as
 * the fpu must be properly saved/resored. This hasn't been tested.
 */

.globl _divide_error,_debug,_nmi,_int3,_overflow,_bounds,_invalid_op
.globl _device_not_available,_double_fault,_coprocessor_segment_overrun
.globl _invalid_TSS,_segment_not_present,_stack_segment
.globl _general_protection,_coprocessor_error,_reserved

//这里的 asm.s 的处理流程是在每个处理函数中先把处理函数的地址压栈,然后调用 no_error_code 或 error_code,在 no_error_code 或 error_code 中调用压入的地址再执行具体的函数
_divide_error:
    pushl $_do_divide_error
no_error_code:
//eax 得到压入的函数地址
    xchgl %eax,(%esp)
//保存各个寄存器变量
    pushl %ebx
    pushl %ecx
    pushl %edx
    pushl %edi
    pushl %esi
    pushl %ebp
    push %ds
    push %es
//压入一个错误码,由于这里是 no_error_code 那么错误码是 0
    push %fs
    pushl $0        # "error code"
//得到栈顶地址,以方便检查哪里出了错误
    lea 44(%esp),%edx
    pushl %edx
//转换为内核代码段
    movl $0x10,%edx
    mov %dx,%ds
    mov %dx,%es
    mov %dx,%fs
//执行具体的函数
    call *%eax
```

```
    addl $8,%esp
    pop %fs
    pop %es
    pop %ds
    popl %ebp
    popl %esi
    popl %edi
    popl %edx
    popl %ecx
    popl %ebx
    popl %eax
    iret

//每一个处理函数的流程都一样,先压入一个处理函数地址,然后在转跳到 no_error_code 执行
_debug:
    pushl $_do_int3     # _do_debug
    jmp no_error_code

_nmi:
    pushl $_do_nmi
    jmp no_error_code

_int3:
    pushl $_do_int3
    jmp no_error_code

_overflow:
    pushl $_do_overflow
    jmp no_error_code

_bounds:
    pushl $_do_bounds
    jmp no_error_code

_invalid_op:
    pushl $_do_invalid_op
    jmp no_error_code

math_emulate:
    popl %eax   //弹出 eax,因为在调用 math_emulate 的时候 push 了 eax
    pushl $_do_device_not_available
    jmp no_error_code
_device_not_available:
```

```
    pushl %eax
    movl %cr0,%eax
//测试是否需要模拟协处理器
    bt $2,%eax          # EM (math emulation bit)
    jc math_emulate
//如果不模拟的话,就用协处理器
    clts                # clear TS so that we can use math
    movl _current,%eax
    cmpl _last_task_used_math,%eax
    je 1f               # shouldn't happen really ...
    pushl %ecx
    pushl %edx
    push %ds
    movl $0x10,%eax
    mov %ax,%ds
    call _math_state_restore
    pop %ds
    popl %edx
    popl %ecx
1:  popl %eax
    iret

_coprocessor_segment_overrun:
    pushl $_do_coprocessor_segment_overrun
    jmp no_error_code

_reserved:
    pushl $_do_reserved
    jmp no_error_code

_coprocessor_error:
    pushl $_do_coprocessor_error
    jmp no_error_code
//如果有错误码的话,那么在执行对应的函数之前自动会把错误码压入的
_double_fault:
    pushl $_do_double_fault
error_code:
    xchgl %eax,4(%esp)      # error code <-> %eax
    xchgl %ebx,(%esp)       # &function <-> %ebx
//保存寄存器变量
    pushl %ecx
    pushl %edx
    pushl %edi
    pushl %esi
```

```
    pushl %ebp
    push %ds
    push %es
    push %fs
//这里压入的是错误码而不是 0 了
    pushl %eax          # error code
    lea 44(%esp),%eax       # offset
    pushl %eax
    movl $0x10,%eax
    mov %ax,%ds
    mov %ax,%es
    mov %ax,%fs
//调用处理函数
    call *%ebx
    addl $8,%esp
    pop %fs
    pop %es
    pop %ds
    popl %ebp
    popl %esi
    popl %edi
    popl %edx
    popl %ecx
    popl %ebx
    popl %eax
    iret

_invalid_TSS:
    pushl $_do_invalid_TSS
    jmp error_code

_segment_not_present:
    pushl $_do_segment_not_present
    jmp error_code

_stack_segment:
    pushl $_do_stack_segment
    jmp error_code

_general_protection:
    pushl $_do_general_protection
    jmp error_code
```