程序 8-2 linux/kernel/traps.c

```
1  /*
2   *  linux/kernel/traps.c
3   *
4   *  (C) 1991  Linus Torvalds
5   */
6
7  /*
8   * 'Traps.c' handles hardware traps and faults after we have saved some
9   * state in 'asm.s'. Currently mostly a debugging-aid, will be extended
10  * to mainly kill the offending process (probably by giving it a signal,
11  * but possibly by killing it outright if necessary).
12  */
    /*
     * 在程序 asm.s 中保存了一些状态后，本程序用来处理硬件陷阱和故障。目前主要用于调试目的，
     * 以后将扩展用来杀死遭损坏的进程（主要是通过发送一个信号，但如果必要也会直接杀死）。
     */
13 #include <string.h>        // 字符串头文件。主要定义了一些有关内存或字符串操作的嵌入函数。
14
15 #include <linux/head.h>    // head 头文件，定义了段描述符的简单结构，和几个选择符常量。
16 #include <linux/sched.h>   // 调度程序头文件，定义了任务结构 task_struct、初始任务 0 的数据，
                              // 还有一些有关描述符参数设置和获取的嵌入式汇编函数宏语句。
17 #include <linux/kernel.h> // 内核头文件。含有一些内核常用函数的原形定义。
18 #include <asm/system.h>    // 系统头文件。定义了设置或修改描述符/中断门等的嵌入式汇编宏。
19 #include <asm/segment.h>   // 段操作头文件。定义了有关段寄存器操作的嵌入式汇编函数。
20 #include <asm/io.h>        // 输入/输出头文件。定义硬件端口输入/输出宏汇编语句。
21
    // 以下语句定义了三个嵌入式汇编宏语句函数。有关嵌入式汇编的基本语法见本程序列表后的说明。
    // 用圆括号括住的组合语句（花括号中的语句）可以作为表达式使用，其中最后的__res 是其输出值。
    // 第 23 行定义了一个寄存器变量__res。该变量将被保存在一个寄存器中，以便于快速访问和操作。
    // 如果想指定寄存器（例如 eax），那么我们可以把该句写成“register char __res asm("ax");”。
    // 取段 seg 中地址 addr 处的一个字节。
    // 参数：seg - 段选择符；addr - 段内指定地址。
    // 输出：%0 - eax (__res)；输入：%1 - eax (seg)；%2 - 内存地址 (*(addr))。
22 #define get_seg_byte(seg,addr) ({ \
23 register char __res; \
24 __asm__("push %%fs;mov %%ax,%%fs;movb %%fs:%2,%%al;pop %%fs" \
25         :"=a" (__res):"0" (seg),"m" (*(addr))); \
26 __res;})
27
    // 取段 seg 中地址 addr 处的一个长字（4 字节）。
    // 参数：seg - 段选择符；addr - 段内指定地址。
    // 输出：%0 - eax (__res)；输入：%1 - eax (seg)；%2 - 内存地址 (*(addr))。
28 #define get_seg_long(seg,addr) ({ \
29 register unsigned long __res; \
30 __asm__("push %%fs;mov %%ax,%%fs;movl %%fs:%2,%%eax;pop %%fs" \
31         :"=a" (__res):"0" (seg),"m" (*(addr))); \
32 __res;})
33
    // 取 fs 段寄存器的值（选择符）。
    // 输出：%0 - eax (__res)。
34 #define _fs() ({ \
35 register unsigned short __res; \
```

```
__asm__("mov %%fs,%%ax":"=a" (__res):); \
__res;})

// 以下定义了一些函数原型。
void page_exception(void);                  // 页异常。实际是 page_fault（mm/page.s，14）。

void divide_error(void);                    // int0（kernel/asm.s，20）。
void debug(void);                           // int1（kernel/asm.s，54）。
void nmi(void);                             // int2（kernel/asm.s，58）。
void int3(void);                            // int3（kernel/asm.s，62）。
void overflow(void);                        // int4（kernel/asm.s，66）。
void bounds(void);                          // int5（kernel/asm.s，70）。
void invalid_op(void);                      // int6（kernel/asm.s，74）。
void device_not_available(void);            // int7（kernel/sys_call.s，158）。
void double_fault(void);                    // int8（kernel/asm.s，98）。
void coprocessor_segment_overrun(void);     // int9（kernel/asm.s，78）。
void invalid_TSS(void);                     // int10（kernel/asm.s，132）。
void segment_not_present(void);             // int11（kernel/asm.s，136）。
void stack_segment(void);                   // int12（kernel/asm.s，140）。
void general_protection(void);              // int13（kernel/asm.s，144）。
void page_fault(void);                      // int14（mm/page.s，14）。
void coprocessor_error(void);               // int16（kernel/sys_call.s，140）。
void reserved(void);                        // int15（kernel/asm.s，82）。
void parallel_interrupt(void);              // int39（kernel/sys_call.s，295）。
void irq13(void);                           // int45 协处理器中断处理（kernel/asm.s，86）。
void alignment_check(void);                 // int46（kernel/asm.s，148）。

// 该子程序用来打印出错中断的名称、出错号、调用程序的 EIP、EFLAGS、ESP、fs 段寄存器值、
// 段的基址、段的长度、进程号 pid、任务号、10 字节指令码。如果堆栈在用户数据段，则还
// 打印 16 字节的堆栈内容。这些信息可用于程序调试。
static void die(char * str,long esp_ptr,long nr)
{
        long * esp = (long *) esp_ptr;
        int i;

        printk("%s: %04x\n\r",str,nr&0xffff);
// 下行打印语句显示当前调用进程的 CS:EIP、EFLAGS 和 SS:ESP 的值。参照错误!未找到引用源。可知，这里 esp[0]
// 即为图中的 esp0 位置。因此我们把这句拆分开来看为：
// (1) EIP:\t%04x:%p\n  -- esp[1]是段选择符（cs），esp[0]是 eip
// (2) EFLAGS:\t%p      -- esp[2]是 eflags
// (3) ESP:\t%04x:%p\n  -- esp[4]是原 ss，esp[3]是原 esp
        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) {               // 若原 ss 值为 0x17（用户栈），则还打印出
                printk("Stack: ");          // 用户栈中的 4 个长字值（16 字节）。
                for (i=0;i<4;i++)
                        printk("%p ",get_seg_long(0x17,i+(long *)esp[3]));
                printk("\n");
        }
        str(i);                     // 取当前运行任务的任务号（include/linux/sched.h，210 行）。
```

```
79          printk("Pid: %d, process nr: %d\n\r",current->pid,0xffff & i); // 进程号，任务号。
80          for(i=0;i<10;i++)
81                  printk("%02x ",0xff & get_seg_byte(esp[1],(i+(char *)esp[0])));
82          printk("\n\r");
83          do_exit(11);            /* play segment exception */
84  }
85
    // 以下这些以 do_开头的函数是 asm.s 中对应中断处理程序调用的 C 函数。
86  void do_double_fault(long esp, long error_code)
87  {
88          die("double fault",esp,error_code);
89  }
90
91  void do_general_protection(long esp, long error_code)
92  {
93          die("general protection",esp,error_code);
94  }
95
96  void do_alignment_check(long esp, long error_code)
97  {
98      die("alignment check",esp,error_code);
99  }
100
101 void do_divide_error(long esp, long error_code)
102 {
103         die("divide error",esp,error_code);
104 }
105
    // 参数是进入中断后被顺序压入堆栈的寄存器值。参见 asm.s 程序第 24--35 行。
106 void do_int3(long * esp, long error_code,
107                 long fs,long es,long ds,
108                 long ebp,long esi,long edi,
109                 long edx,long ecx,long ebx,long eax)
110 {
111         int tr;
112
113         __asm__("str %%ax":"=a" (tr):"" (0));                   // 取任务寄存器值➔tr。
114         printk("eax\t\tebx\t\tecx\t\tedx\n\r%8x\t%8x\t%8x\t%8x\n\r",
115                 eax,ebx,ecx,edx);
116         printk("esi\t\tedi\t\tebp\t\tesp\n\r%8x\t%8x\t%8x\t%8x\n\r",
117                 esi,edi,ebp,(long) esp);
118         printk("\n\rds\tes\tfs\ttr\n\r%4x\t%4x\t%4x\t%4x\n\r",
119                 ds,es,fs,tr);
120         printk("EIP: %8x   CS: %4x  EFLAGS: %8x\n\r",esp[0],esp[1],esp[2]);
121 }
122
123 void do_nmi(long esp, long error_code)
124 {
125         die("nmi",esp,error_code);
126 }
127
128 void do_debug(long esp, long error_code)
129 {
```

```
130         die("debug",esp,error_code);
131 }
132
133 void do_overflow(long esp, long error_code)
134 {
135         die("overflow",esp,error_code);
136 }
137
138 void do_bounds(long esp, long error_code)
139 {
140         die("bounds",esp,error_code);
141 }
142
143 void do_invalid_op(long esp, long error_code)
144 {
145         die("invalid operand",esp,error_code);
146 }
147
148 void do_device_not_available(long esp, long error_code)
149 {
150         die("device not available",esp,error_code);
151 }
152
153 void do_coprocessor_segment_overrun(long esp, long error_code)
154 {
155         die("coprocessor segment overrun",esp,error_code);
156 }
157
158 void do_invalid_TSS(long esp,long error_code)
159 {
160         die("invalid TSS",esp,error_code);
161 }
162
163 void do_segment_not_present(long esp,long error_code)
164 {
165         die("segment not present",esp,error_code);
166 }
167
168 void do_stack_segment(long esp,long error_code)
169 {
170         die("stack segment",esp,error_code);
171 }
172
173 void do_coprocessor_error(long esp, long error_code)
174 {
175         if (last_task_used_math != current)
176                 return;
177         die("coprocessor error",esp,error_code);
178 }
179
180 void do_reserved(long esp, long error_code)
181 {
182         die("reserved (15,17-47) error",esp,error_code);
```

```
183 }
184
    // 下面是异常（陷阱）中断程序初始化子程序。设置它们的中断调用门（中断向量）。
    // set_trap_gate()与 set_system_gate()都使用了中断描述符表 IDT 中的陷阱门（Trap Gate），
    // 它们之间的主要区别在于前者设置的特权级为 0，后者是 3。因此断点陷阱中断 int3、溢出中断
    // overflow 和边界出错中断 bounds 可以由任何程序调用。 这两个函数均是嵌入式汇编宏程序，
    // 参见 include/asm/system.h，第 36 行、39 行。
185 void trap_init(void)
186 {
187         int i;
188
189         set_trap_gate(0,&divide_error);      // 设置除操作出错的中断向量值。以下雷同。
190         set_trap_gate(1,&debug);
191         set_trap_gate(2,&nmi);
192         set_system_gate(3,&int3);            /* int3-5 can be called from all */
193         set_system_gate(4,&overflow);        /* int3-5 可以被所有程序执行 */
194         set_system_gate(5,&bounds);
195         set_trap_gate(6,&invalid_op);
196         set_trap_gate(7,&device_not_available);
197         set_trap_gate(8,&double_fault);
198         set_trap_gate(9,&coprocessor_segment_overrun);
199         set_trap_gate(10,&invalid_TSS);
200         set_trap_gate(11,&segment_not_present);
201         set_trap_gate(12,&stack_segment);
202         set_trap_gate(13,&general_protection);
203         set_trap_gate(14,&page_fault);
204         set_trap_gate(15,&reserved);
205         set_trap_gate(16,&coprocessor_error);
206         set_trap_gate(17,&alignment_check);

    // 下面把 int17-47 的陷阱门先均设置为 reserved，以后各硬件初始化时会重新设置自己的陷阱门。
207         for (i=18;i<48;i++)
208                 set_trap_gate(i,&reserved);

    // 设置协处理器中断 0x2d（45）陷阱门描述符，并允许其产生中断请求。设置并行口中断描述符。
209         set_trap_gate(45,&irq13);
210         outb_p(inb_p(0x21)&0xfb,0x21);          // 允许 8259A 主芯片的 IRQ2 中断请求。
211         outb(inb_p(0xA1)&0xdf,0xA1);            // 允许 8259A 从芯片的 IRQ13 中断请求。
212         set_trap_gate(39,&parallel_interrupt);  // 设置并行口 1 的中断 0x27 陷阱门描述符。
213 }
214
```