程序 8-1 linux/kernel/asm.s

```
/*
 *  linux/kernel/asm.s
 *
 *  (C) 1991  Linus Torvalds
 */

/*
 * 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.
 */
/*
 * asm.s 程序中包括大部分的硬件故障（或出错）处理的底层次代码。页异常由内存管理程序
 * mm 处理，所以不在这里。此程序还处理（希望是这样）由于 TS-位而造成的 fpu 异常，因为
 * fpu 必须正确地进行保存/恢复处理，这些还没有测试过。
 */

# 本代码文件主要涉及对 Intel 保留中断 int0--int16 的处理（int17-int31 留作今后使用）。
# 以下是一些全局函数名的声明，其原形在 traps.c 中说明。
.globl _divide_error,_debug,_nmi,_int3,_overflow,_bounds,_invalid_op
.globl _double_fault,_coprocessor_segment_overrun
.globl _invalid_TSS,_segment_not_present,_stack_segment
.globl _general_protection,_coprocessor_error,_irq13,_reserved
.globl _alignment_check

# 下面这段程序处理无出错号的情况。
# int0 -- 处理被零除出错的情况。  类型：错误；出错号：无。
# 在执行 DIV 或 IDIV 指令时，若除数是 0，CPU 就会产生这个异常。当 EAX（或 AX、AL）容纳
# 不了一个合法除操作的结果时，也会产生这个异常。21 行标号'_do_divide_error'实际上是
# C 语言函数 do_divide_error()编译后所生成模块中对应的名称。函数'do_divide_error'在
# traps.c 中实现（第 101 行开始）。
_divide_error:
        pushl $_do_divide_error     # 首先把将要调用的函数地址入栈。
no_error_code:                      # 这里是无出错号处理的入口处，见下面第 56 行等。
        xchgl %eax,(%esp)           # _do_divide_error 的地址 ➔ eax，eax 被交换入栈。
        pushl %ebx
        pushl %ecx
        pushl %edx
        pushl %edi
        pushl %esi
        pushl %ebp
        push %ds                    # ！！16 位的段寄存器入栈后也要占用 4 个字节。
        push %es
        push %fs
        pushl $0                    # "error code"   # 将数值 0 作为出错码入栈。
        lea 44(%esp),%edx           # 取有效地址，即栈中原调用返回地址处的栈指针位置，
        pushl %edx                  # 并压入堆栈。
        movl $0x10,%edx             # 初始化段寄存器 ds、es 和 fs，加载内核数据段选择符。
        mov %dx,%ds
        mov %dx,%es
        mov %dx,%fs
```

```
    # 下行上的'*'号表示调用操作数指定地址处的函数，称为间接调用。这句的含义是调用引起本次
    # 异常的 C 处理函数，例如 do_divide_error()等。第 41 行是将堆栈指针加 8 相当于执行两次 pop
    # 操作，弹出（丢弃）最后入堆栈的两个 C 函数参数（33 行和 35 行入栈的值），让堆栈指针重新
    # 指向寄存器 fs 入栈处。
40          call *%eax
41          addl $8,%esp
42          pop %fs
43          pop %es
44          pop %ds
45          popl %ebp
46          popl %esi
47          popl %edi
48          popl %edx
49          popl %ecx
50          popl %ebx
51          popl %eax                   # 弹出原来 eax 中的内容。
52          iret
53
    # int1 -- debug 调试中断入口点。处理过程同上。类型：错误/陷阱（Fault/Trap）；无错误号。
    # 当 eflags 中 TF 标志置位时而引发的中断。当发现硬件断点（数据：陷阱，代码：错误）；或者
    # 开启了指令跟踪陷阱或任务交换陷阱，或者调试寄存器访问无效（错误），CPU 就会产生该异常。
54  _debug:
55          pushl $_do_int3         # _do_debug  # C 函数指针入栈。以下同。
56          jmp no_error_code
57
    # int2 -- 非屏蔽中断调用入口点。   类型：陷阱；无错误号。
    # 这是仅有的被赋予固定中断向量的硬件中断。每当接收到一个 NMI 信号，CPU 内部就会产生中断
    # 向量 2，并执行标准中断应答周期，因此很节省时间。NMI 通常保留为极为重要的硬件事件使用。
    # 当 CPU 收到一个 NMI 信号并且开始执行其中断处理过程时，随后所有的硬件中断都将被忽略。
58  _nmi:
59          pushl $_do_nmi
60          jmp no_error_code
61
    # int3 -- 断点指令引起中断的入口点。   类型：陷阱；无错误号。
    # 由 int 3 指令引发的中断，与硬件中断无关。该指令通常由调式器插入被调式程序的代码中。
    # 处理过程同_debug。
62  _int3:
63          pushl $_do_int3
64          jmp no_error_code
65
    # int4 -- 溢出出错处理中断入口点。   类型：陷阱；无错误号。
    # EFLAGS 中 OF 标志置位时 CPU 执行 INTO 指令就会引发该中断。通常用于编译器跟踪算术计算溢出。
66  _overflow:
67          pushl $_do_overflow
68          jmp no_error_code
69
    # int5 -- 边界检查出错中断入口点。   类型：错误；无错误号。
    # 当操作数在有效范围以外时引发的中断。当 BOUND 指令测试失败就会产生该中断。BOUND 指令有
    # 3 个操作数，如果第 1 个不在另外两个之间，就产生异常 5。
70  _bounds:
71          pushl $_do_bounds
72          jmp no_error_code
73
```

```
    # int6 -- 无效操作指令出错中断入口点。  类型：错误；无错误号。
    # CPU 执行机构检测到一个无效的操作码而引起的中断。
74  _invalid_op:
75          pushl $_do_invalid_op
76          jmp no_error_code
77
    # int9 -- 协处理器段超出出错中断入口点。  类型：放弃；无错误号。
    # 该异常基本上等同于协处理器出错保护。因为在浮点指令操作数太大时，我们就有这个机会来
    # 加载或保存超出数据段的浮点值。
78  _coprocessor_segment_overrun:
79          pushl $_do_coprocessor_segment_overrun
80          jmp no_error_code
81
    # int15 - 其他 Intel 保留中断的入口点。
82  _reserved:
83          pushl $_do_reserved
84          jmp no_error_code
85
    # int45 -- (0x20 + 13) Linux 设置的数学协处理器硬件中断。
    # 当协处理器执行完一个操作时就会发出 IRQ13 中断信号，以通知 CPU 操作完成。80387 在执行
    # 计算时，CPU 会等待其操作完成。下面 89 行上 0xF0 是协处理端口，用于清忙锁存器。通过写
    # 该端口，本中断将消除 CPU 的 BUSY 延续信号，并重新激活 80387 的处理器扩展请求引脚 PEREQ。
    # 该操作主要是为了确保在继续执行 80387 的任何指令之前，CPU 响应本中断。
86  _irq13:
87          pushl %eax
88          xorb %al,%al
89          outb %al,$0xF0
90          movb $0x20,%al
91          outb %al,$0x20              # 向 8259 主中断控制芯片发送 EOI（中断结束）信号。
92          jmp 1f                      # 这两个跳转指令起延时作用。
93  1:      jmp 1f
94  1:      outb %al,$0xA0              # 再向 8259 从中断控制芯片发送 EOI（中断结束）信号。
95          popl %eax
96          jmp _coprocessor_error      # 该函数原在本程序中，现已放到 system_call.s 中。
97
    # 以下中断在调用时 CPU 会在中断返回地址之后将出错号压入堆栈，因此返回时也需要将出错号
    # 弹出（参见图 5.3(b)）。

    # int8 -- 双出错故障。  类型：放弃；有错误码。
    # 通常当 CPU 在调用前一个异常的处理程序而又检测到一个新的异常时，这两个异常会被串行地进行
    # 处理，但也会碰到很少的情况，CPU 不能进行这样的串行处理操作，此时就会引发该中断。
98  _double_fault:
99          pushl $_do_double_fault     # C 函数地址入栈。
100 error_code:
101         xchgl %eax,4(%esp)          # error code <-> %eax，eax 原来的值被保存在堆栈上。
102         xchgl %ebx,(%esp)           # &function <-> %ebx，ebx 原来的值被保存在堆栈上。
103         pushl %ecx
104         pushl %edx
105         pushl %edi
106         pushl %esi
107         pushl %ebp
108         push %ds
109         push %es
```

```
110         push %fs
111         pushl %eax                  # error code    # 出错号入栈。
112         lea 44(%esp),%eax           # offset        # 程序返回地址处堆栈指针位置值入栈。
113         pushl %eax
114         movl $0x10,%eax             # 置内核数据段选择符。
115         mov %ax,%ds
116         mov %ax,%es
117         mov %ax,%fs
118         call *%ebx                  # 间接调用，调用相应的 C 函数，其参数已入栈。
119         addl $8,%esp                # 丢弃入栈的 2 个用作 C 函数的参数。
120         pop %fs
121         pop %es
122         pop %ds
123         popl %ebp
124         popl %esi
125         popl %edi
126         popl %edx
127         popl %ecx
128         popl %ebx
129         popl %eax
130         iret
131
    # int10 -- 无效的任务状态段(TSS)。   类型：错误；有出错码。
    # CPU 企图切换到一个进程，而该进程的 TSS 无效。根据 TSS 中哪一部分引起了异常，当由于 TSS
    # 长度超过 104 字节时，这个异常在当前任务中产生，因而切换被终止。其他问题则会导致在切换
    # 后的新任务中产生本异常。
132 _invalid_TSS:
133         pushl $_do_invalid_TSS
134         jmp error_code
135
    # int11 -- 段不存在。   类型：错误；有出错码。
    # 被引用的段不在内存中。段描述符中标志指明段不在内存中。
136 _segment_not_present:
137         pushl $_do_segment_not_present
138         jmp error_code
139
    # int12 -- 堆栈段错误。   类型：错误；有出错码。
    # 指令操作试图超出堆栈段范围，或者堆栈段不在内存中。这是异常 11 和 13 的特例。有些操作
    # 系统可以利用这个异常来确定什么时候应该为程序分配更多的栈空间。
140 _stack_segment:
141         pushl $_do_stack_segment
142         jmp error_code
143
    # int13 -- 一般保护性出错。   类型：错误；有出错码。
    # 表明是不属于任何其他类的错误。若一个异常产生时没有对应的处理向量（0--16），通常就
    # 会归到此类。
144 _general_protection:
145         pushl $_do_general_protection
146         jmp error_code
147
    # int17 -- 边界对齐检查出错。
    # 在启用了内存边界检查时，若特权级 3（用户级）数据非边界对齐时会产生该异常。
148 _alignment_check:
```

```
149         pushl $_do_alignment_check
150         jmp error_code
151
    # int7 -- 设备不存在（_device_not_available）在 kernel/sys_call.s，158 行。
    # int14 -- 页错误（_page_fault）在 mm/page.s，14 行。
    # int16 -- 协处理器错误（_coprocessor_error）在 kernel/sys_call.s，140 行。
    # 时钟中断 int 0x20（_timer_interrupt）在 kernel/sys_call.s，189 行。
    # 系统调用 int 0x80（_system_call）在 kernel/sys_call.s，84 行。
```