程序 10-4 linux/kernel/chr_drv/rs_io.s

```
1  /*
2   *  linux/kernel/rs_io.s
3   *
4   *  (C) 1991  Linus Torvalds
5   */
6
7  /*
8   *      rs_io.s
9   *
10  * This module implements the rs232 io interrupts.
11  */
    /*
     * 该模块实现 rs232 输入输出中断处理程序。
     */
12
13 .text
14 .globl _rs1_interrupt,_rs2_interrupt
15
    // size 是读写队列缓冲区的字节长度。该值必须是 2 的次方，并且必须与 tty_io.c 中的匹配。
16 size    = 1024                          /* must be power of two !
17                                             and must match the value
18                                             in tty_io.c!!! */
19
20 /* these are the offsets into the read/write buffer structures */
    /* 以下这些是读写缓冲队列结构中的偏移量 */
    // 对应 include/linux/tty.h 文件中 tty_queue 结构中各字段的字节偏移量。其中 rs_addr
    // 对应 tty_queue 结构的 data 字段。对于串行终端缓冲队列，该字段存放着串行端口基地址。
21 rs_addr = 0                       // 串行端口号字段偏移（端口是 0x3f8 或 0x2f8）。
22 head = 4                          // 缓冲区中头指针字段偏移。
23 tail = 8                          // 缓冲区中尾指针字段偏移。
24 proc_list = 12                    // 等待该缓冲的进程字段偏移。
25 buf = 16                          // 缓冲区字段偏移。
26
    // 当一个写缓冲队列满后，内核就会把要往写队列填字符的进程设置为等待状态。当写缓冲队列
    // 中还剩余最多 256 个字符时，中断处理程序就可以唤醒这些等待进程继续往写队列中放字符。
27 startup = 256                     /* chars left in write queue when we restart it */
                                     /* 当我们重新开始写时，队列里最多还剩余字符个数。*/
28
29 /*
30  * These are the actual interrupt routines. They look where
31  * the interrupt is coming from, and take appropriate action.
32  */
    /*
     * 这些是实际的中断处理程序。程序首先检查中断的来源，然后执行
     * 相应的处理。
     */
    //// 串行端口 1 中断处理程序入口点。
    // 初始化时 rs1_interrupt 地址被放入中断描述符 0x24 中，对应 8259A 的中断请求 IRQ4 引脚。
    // 这里首先把 tty 表中串行终端 1（串口 1）读写缓冲队列指针的地址入栈（tty_io.c，81），
    // 然后跳转到 rs_int 继续处理。这样做可以让串口 1 和串口 2 的处理代码公用。字符缓冲队列
    // 结构 tty_queue 格式请参见 include/linux/tty.h，第 22 行。
33 .align 2
```

```
34 _rs1_interrupt:
35         pushl $_table_list+8    // tty 表中串口 1 读写缓冲队列指针地址入栈。
36         jmp rs_int
37 .align 2
   //// 串行端口 2 中断处理程序入口点。
38 _rs2_interrupt:
39         pushl $_table_list+16   // tty 表中串口 2 读写缓冲队列指针地址入栈。

   // 这段代码首先让段寄存器 ds、es 指向内核数据段，然后从对应读写缓冲队列 data 字段取出
   // 串行端口基地址。该地址加 2 即是中断标识寄存器 IIR 的端口地址。若位 0 = 0，表示有需
   // 要处理的中断。于是根据位 2、位 1 使用指针跳转表调用相应中断源类型处理子程序。在每
   // 个子程序中会在处理完后复位 UART 的相应中断源。在子程序返回后这段代码会循环判断是
   // 否还有其他中断源（位 0 = 0？）。如果本次中断还有其他中断源，则 IIR 的位 0 仍然是 0。
   // 于是中断处理程序会再调用相应中断源子程序继续处理。直到引起本次中断的所有中断源都
   // 被处理并复位，此时 UART 会自动地设置 IIR 的位 0 =1，表示已无待处理的中断，于是中断
   // 处理程序即可退出。
40 rs_int:
41         pushl %edx
42         pushl %ecx
43         pushl %ebx
44         pushl %eax
45         push %es
46         push %ds                /* as this is an interrupt, we cannot */
47         pushl $0x10             /* know that bs is ok. Load it */
48         pop %ds                 /* 由于这是一个中断程序，我们不知道 ds 是否正确，*/
49         pushl $0x10             /* 所以加载它们（让 ds、es 指向内核数据段） */
50         pop %es
51         movl 24(%esp),%edx      // 取上面 35 或 39 行入栈的相应串口缓冲队列指针地址。
52         movl (%edx),%edx        // 取读缓冲队列结构指针（地址）➔edx。
53         movl rs_addr(%edx),%edx // 取串口 1（或串口 2）端口基地址➔edx。
54         addl $2,%edx            /* interrupt ident. reg */  /* 指向中断标识寄存器 */
                                   // 中断标识寄存器端口地址是 0x3fa（0x2fa）。
55 rep_int:
56         xorl %eax,%eax
57         inb %dx,%al             // 取中断标识字节，以判断中断来源（有 4 种中断情况）。
58         testb $1,%al            // 首先判断有无待处理中断（位 0 = 0 有中断）。
59         jne end                 // 若无待处理中断，则跳转至退出处理处 end。
60         cmpb $6,%al             /* this shouldn't happen, but ... */ /*这不会发生，但…*/
61         ja end                  // al 值大于 6，则跳转至 end（没有这种状态）。
62         movl 24(%esp),%ecx      // 调用子程序之前把缓冲队列指针地址放入 ecx。
63         pushl %edx              // 临时保存中断标识寄存器端口地址。
64         subl $2,%edx            // edx 中恢复串口基地址值 0x3f8（0x2f8）。
65         call jmp_table(,%eax,2) /* NOTE! not *4, bit0 is 0 already */
   // 上面语句是指，当有待处理中断时，al 中位 0=0，位 2、位 1 是中断类型，因此相当于已经将
   // 中断类型乘了 2，这里再乘 2，获得跳转表（第 79 行）对应各中断类型地址，并跳转到那里去
   // 作相应处理。中断来源有 4 种：modem 状态发生变化；要写（发送）字符；要读（接收）字符；
   // 线路状态发生变化。允许发送字符中断通过设置发送保持寄存器标志实现。在 serial.c 程序
   // 中，当写缓冲队列中有数据时，rs_write()函数就会修改中断允许寄存器内容，添加上发送保
   // 持寄存器中断允许标志，从而在系统需要发送字符时引起串行中断发生。
66         popl %edx               // 恢复中断标识寄存器端口地址 0x3fa（或 0x2fa）。
67         jmp rep_int             // 跳转，继续判断有无待处理中断并作相应处理。

68 end:    movb $0x20,%al          // 中断退出处理。向中断控制器发送结束中断指令 EOI。
```

```
69          outb %al,$0x20          /* EOI */
70          pop %ds
71          pop %es
72          popl %eax
73          popl %ebx
74          popl %ecx
75          popl %edx
76          addl $4,%esp            # jump over _table_list entry    # 丢弃队列指针地址。
77          iret
78
    // 各中断类型处理子程序地址跳转表，共有 4 种中断来源：
    // modem 状态变化中断，写字符中断，读字符中断，线路状态有问题中断。
79  jmp_table:
80          .long modem_status,write_char,read_char,line_status
81
    // 由于 modem 状态发生变化而引发此次中断。通过读 modem 状态寄存器 MSR 对其进行复位操作。
82  .align 2
83  modem_status:
84          addl $6,%edx            /* clear intr by reading modem status reg */
85          inb %dx,%al             /* 通过读 modem 状态寄存器进行复位（0x3fe） */
86          ret
87
    // 由于线路状态发生变化而引起这次串行中断。通过读线路状态寄存器 LSR 对其进行复位操作。
88  .align 2
89  line_status:
90          addl $5,%edx            /* clear intr by reading line status reg. */
91          inb %dx,%al             /* 通过读线路状态寄存器进行复位（0x3fd） */
92          ret
93
    // 由于 UART 芯片接收到字符而引起这次中断。对接收缓冲寄存器执行读操作可复位该中断源。
    // 这个子程序将接收到的字符放到读缓冲队列 read_q 头指针（head）处，并且让该指针前移一
    // 个字符位置。若 head 指针已经到达缓冲区末端，则让其折返到缓冲区开始处。最后调用 C 函
    // 数 do_tty_interrupt()（也即 copy_to_cooked()），把读入的字符经过处理放入规范模式缓
    // 冲队列（辅助缓冲队列 secondary）中。
94  .align 2
95  read_char:
96          inb %dx,%al             // 读取接收缓冲寄存器 RBR 中字符➔al。
97          movl %ecx,%edx          // 当前串口缓冲队列指针地址➔edx。
98          subl $_table_list,%edx  // 当前串口队列指针地址 - 缓冲队列指针表首址 ➔edx，
99          shrl $3,%edx            // 差值/8，得串口号。对于串口 1 是 1，对于串口 2 是 2。
100         movl (%ecx),%ecx         # read-queue     // 取读缓冲队列结构地址➔ecx。
101         movl head(%ecx),%ebx    // 取读队列中缓冲头指针➔ebx。
102         movb %al,buf(%ecx,%ebx) // 将字符放在缓冲区中头指针所指位置处。
103         incl %ebx               // 将头指针前移（右移）一字节。
104         andl $size-1,%ebx       // 用缓冲区长度对头指针取模操作。
105         cmpl tail(%ecx),%ebx    // 缓冲区头指针与尾指针比较。
106         je 1f                   // 若指针移动后相等，表示缓冲区满，不保存头指针，跳转。
107         movl %ebx,head(%ecx)    // 保存修改过的头指针。
108 1:      addl $63,%edx           // 串口号转换成 tty 号（63 或 64）并作为参数入栈。
109         pushl %edx
110         call _do_tty_interrupt  // 调用 tty 中断处理 C 函数（tty_io.c，342 行）。
111         addl $4,%esp            // 丢弃入栈参数，并返回。
112         ret
```

```
113
    // 由于设置了发送保持寄存器允许中断标志而引起此次中断。说明对应串行终端的写字符缓冲队
    // 列中有字符需要发送。于是计算出写队列中当前所含字符数，若字符数已小于 256 个，则唤醒
    // 等待写操作进程。然后从写缓冲队列尾部取出一个字符发送，并调整和保存尾指针。如果写缓
    // 冲队列已空，则跳转到 write_buffer_empty 处处理写缓冲队列空的情况。
114 .align 2
115 write_char:
116         movl 4(%ecx),%ecx               # write-queue  // 取写缓冲队列结构地址➔ecx。
117         movl head(%ecx),%ebx           // 取写队列头指针➔ebx。
118         subl tail(%ecx),%ebx           // 头指针 - 尾指针 = 队列中字符数。
119         andl $size-1,%ebx               # nr chars in queue
120         je write_buffer_empty          // 若头指针 = 尾指针，说明写队列空，跳转处理。
121         cmpl $startup,%ebx             // 队列中字符数还超过 256 个？
122         ja 1f                          // 超过则跳转处理。
123         movl proc_list(%ecx),%ebx       # wake up sleeping process  # 唤醒等待的进程。
                                           // 取等待该队列的进程指针，并判断是否为空。
124         testl %ebx,%ebx                 # is there any?    # 有等待写的进程吗？
125         je 1f                          // 是空的，则向前跳转到标号 1 处。
126         movl $0,(%ebx)                 // 否则将进程置为可运行状态（唤醒进程）。
127 1:      movl tail(%ecx),%ebx           // 取尾指针。
128         movb buf(%ecx,%ebx),%al        // 从缓冲中尾指针处取一字符➔al。
129         outb %al,%dx                   // 向端口 0x3f8（0x2f8）写到发送保持寄存器中。
130         incl %ebx                      // 尾指针前移。
131         andl $size-1,%ebx              // 尾指针若到缓冲区末端，则折回。
132         movl %ebx,tail(%ecx)           // 保存已修改过的尾指针。
133         cmpl head(%ecx),%ebx           // 尾指针与头指针比较，
134         je write_buffer_empty          // 若相等，表示队列已空，则跳转。
135         ret
    // 处理写缓冲队列 write_q 已空的情况。若有等待写该串行终端的进程则唤醒之，然后屏蔽发
    // 送保持寄存器空中断，不让发送保持寄存器空时产生中断。
    // 如果此时写缓冲队列 write_q 已空，表示当前无字符需要发送。于是我们应该做两件事情。
    // 首先看看有没有进程正等待写队列空出来，如果有就唤醒之。另外，因为现在系统已无字符
    // 需要发送，所以此时我们要暂时禁止发送保持寄存器 THR 空时产生中断。当再有字符被放入
    // 写缓冲队列中时，serial.c 中的 rs_write()函数会再次允许发送保持寄存器空时产生中断，
    // 因此 UART 就又会“自动”地来取写缓冲队列中的字符，并发送出去。
136 .align 2
137 write_buffer_empty:
138         movl proc_list(%ecx),%ebx       # wake up sleeping process # 唤醒等待的进程。
                                           // 取等待该队列的进程的指针，并判断是否为空。
139         testl %ebx,%ebx                 # is there any?    # 有等待的进程吗？
140         je 1f                          // 无，则向前跳转到标号 1 处。
141         movl $0,(%ebx)                 // 否则将进程置为可运行状态（唤醒进程）。
142 1:      incl %edx                      // 指向端口 0x3f9（0x2f9）。
143         inb %dx,%al                    // 读取中断允许寄存器 IER。
144         jmp 1f                         // 稍作延迟。
145 1:      jmp 1f                         /* 屏蔽发送保持寄存器空中断（位 1） */
146 1:      andb $0xd,%al                  /* disable transmit interrupt */
147         outb %al,%dx                   // 写入 0x3f9(0x2f9)。
148         ret
```