```
1  /*
2   *  linux/kernel/tty_io.c
3   *
4   *  (C) 1991  Linus Torvalds
5   */
6
7  /*
8   * 'tty_io.c' gives an orthogonal feeling to tty's, be they consoles
9   * or rs-channels. It also implements echoing, cooked mode etc.
10  *
11  * Kill-line thanks to John T Kohl, who also corrected VMIN = VTIME = 0.
12  */
    /*
    * 'tty_io.c'给 tty 终端一种非相关的感觉，不管它们是控制台还是串行终端。
    * 该程序同样实现了回显、规范(熟)模式等。
    *
    * Kill-line 问题，要谢谢 John T Kohl。他同时还纠正了当 VMIN = VTIME = 0 时的问题。
    */
13
14 #include <ctype.h>          // 字符类型头文件。定义了一些有关字符类型判断和转换的宏。
15 #include <errno.h>          // 错误号头文件。包含系统中各种出错号。
16 #include <signal.h>         // 信号头文件。定义信号符号常量，信号结构及其操作函数原型。
17 #include <unistd.h>         // unistd.h 是标准符号常数与类型文件，并声明了各种函数。
18
    // 给出定时警告（alarm）信号在信号位图中对应的比特屏蔽位。
19 #define ALRMMASK (1<<(SIGALRM-1))
20
21 #include <linux/sched.h>  // 调度程序头文件，定义了任务结构 task_struct、任务 0 数据等。
22 #include <linux/tty.h>    // tty 头文件，定义了有关 tty_io，串行通信方面的参数、常数。
23 #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。
24 #include <asm/system.h>   // 系统头文件。定义设置或修改描述符/中断门等嵌入式汇编宏。
25
    // 终止进程组（向进程组发送信号）。参数 pgrp 指定进程组号；sig 指定信号；priv 权限。
    // 即向指定进程组 pgrp 中的每个进程发送指定信号 sig。只要向一个进程发送成功最后就会
    // 返回 0，否则如果没有找到指定进程组号 pgrp 的任何一个进程，则返回出错号-ESRCH，若
    // 找到进程组号是 pgrp 的进程，但是发送信号失败，则返回发送失败的错误码。
26 int kill_pg(int pgrp, int sig, int priv);              // kernel/exit.c，171 行。
    // 判断一个进程组是否是孤儿进程。如果不是则返回 0；如果是则返回 1。
27 int is_orphaned_pgrp(int pgrp);                       // kernel/exit.c，232 行。
28
    // 获取 termios 结构中三个模式标志集之一，或者用于判断一个标志集是否有置位标志。
29 #define _L_FLAG(tty,f)  ((tty)->termios.c_lflag & f)    // 本地模式标志。
30 #define _I_FLAG(tty,f)  ((tty)->termios.c_iflag & f)    // 输入模式标志。
31 #define _O_FLAG(tty,f)  ((tty)->termios.c_oflag & f)    // 输出模式标志。
32
    // 取 termios 结构终端特殊（本地）模式标志集中的一个标志。
33 #define L_CANON(tty)    _L_FLAG((tty),ICANON)    // 取规范模式标志。
34 #define L_ISIG(tty)     _L_FLAG((tty),ISIG)      // 取信号标志。
35 #define L_ECHO(tty)     _L_FLAG((tty),ECHO)      // 取回显字符标志。
36 #define L_ECHOE(tty)    _L_FLAG((tty),ECHOE)     // 规范模式时取回显擦出标志。
37 #define L_ECHOK(tty)    _L_FLAG((tty),ECHOK)     // 规范模式时取 KILL 擦除当前行标志。
38 #define L_ECHOCTL(tty)  _L_FLAG((tty),ECHOCTL)   // 取回显控制字符标志。
```

```
39 #define L_ECHOKE(tty)    _L_FLAG((tty),ECHOKE)    // 规范模式时取 KILL 擦除行并回显标志。
40 #define L_TOSTOP(tty)    _L_FLAG((tty),TOSTOP)    // 对于后台输出发送 SIGTTOU 信号。
41
   // 取 termios 结构输入模式标志集中的一个标志。
42 #define I_UCLC(tty)      _I_FLAG((tty),IUCLC)     // 取大写到小写转换标志。
43 #define I_NLCR(tty)      _I_FLAG((tty),INLCR)     // 取换行符 NL 转回车符 CR 标志。
44 #define I_CRNL(tty)      _I_FLAG((tty),ICRNL)     // 取回车符 CR 转换行符 NL 标志。
45 #define I_NOCR(tty)      _I_FLAG((tty),IGNCR)     // 取忽略回车符 CR 标志。
46 #define I_IXON(tty)      _I_FLAG((tty),IXON)      // 取输入控制流标志 XON。
47
   // 取 termios 结构输出模式标志集中的一个标志。
48 #define O_POST(tty)      _O_FLAG((tty),OPOST)     // 取执行输出处理标志。
49 #define O_NLCR(tty)      _O_FLAG((tty),ONLCR)     // 取换行符 NL 转回车换行符 CR-NL 标志。
50 #define O_CRNL(tty)      _O_FLAG((tty),OCRNL)     // 取回车符 CR 转换行符 NL 标志。
51 #define O_NLRET(tty)     _O_FLAG((tty),ONLRET)    // 取换行符 NL 执行回车功能的标志。
52 #define O_LCUC(tty)      _O_FLAG((tty),OLCUC)     // 取小写转大写字符标志。
53
   // 取 termios 结构控制标志集中波特率。CBAUD 是波特率屏蔽码（0000017）。
54 #define C_SPEED(tty)     ((tty)->termios.c_cflag & CBAUD)
   // 判断 tty 终端是否已挂线（hang up），即其传输波特率是否为 B0（0）。
55 #define C_HUP(tty)       (C_SPEED((tty)) == B0)
56
   // 取最小值宏。
57 #ifndef MIN
58 #define MIN(a,b) ((a) < (b) ? (a) : (b))
59 #endif
60
   // 下面定义 tty 终端使用的缓冲队列结构数组 tty_queues 和 tty 终端表结构数组 tty_table。
   // QUEUES 是 tty 终端使用的缓冲队列最大数量。伪终端分主从两种（master 和 slave）。每个
   // tty 终端使用 3 个 tty 缓冲队列，它们分别是用于缓冲键盘或串行输入的读队列 read_queue、
   // 用于缓冲屏幕或串行输出的写队列 write_queue，以及用于保存规范模式字符的辅助缓冲队列
   // secondary。
61 #define QUEUES   (3*(MAX_CONSOLES+NR_SERIALS+2*NR_PTYS))  // 共 54 项。
62 static struct tty_queue tty_queues[QUEUES];               // tty 缓冲队列数组。
63 struct tty_struct tty_table[256];                         // tty 表结构数组。
64
   // 下面设定各种类型的 tty 终端所使用缓冲队列结构在 tty_queues[]数组中的起始项位置。
   // 8 个虚拟控制台终端占用 tty_queues[]数组开头 24 项（3 X MAX_CONSOLES）（0 -- 23）；
   // 两个串行终端占用随后的 6 项（3 X NR_SERIALS）（24 -- 29）。
   // 4 个主伪终端占用随后的 12 项（3 X NR_PTYS）（30 -- 41）。
   // 4 个从伪终端占用随后的 12 项（3 X NR_PTYS）（42 -- 53）。
65 #define con_queues tty_queues
66 #define rs_queues ((3*MAX_CONSOLES) + tty_queues)
67 #define mpty_queues ((3*(MAX_CONSOLES+NR_SERIALS)) + tty_queues)
68 #define spty_queues ((3*(MAX_CONSOLES+NR_SERIALS+NR_PTYS)) + tty_queues)
69
   // 下面设定各种类型 tty 终端所使用的 tty 结构在 tty_table[]数组中的起始项位置。
   // 8 个虚拟控制台终端可用 tty_table[]数组开头 64 项（0 -- 63）；
   // 两个串行终端使用随后的 2 项（64 -- 65）。
   // 4 个主伪终端使用从 128 开始的项，最多 64 项（128 -- 191）。
   // 4 个从伪终端使用从 192 开始的项，最多 64 项（192 -- 255）。
70 #define con_table tty_table                 // 定义控制台终端 tty 表符号常数。
71 #define rs_table (64+tty_table)             // 串行终端 tty 表。
```

```
72 #define mpty_table (128+tty_table)          // 主伪终端 tty 表。
73 #define spty_table (192+tty_table)          // 从伪终端 tty 表。
74
75 int fg_console = 0;                 // 当前前台控制台号（范围 0--7）。
76
77 /*
78  * these are the tables used by the machine code handlers.
79  * you can implement virtual consoles.
80  */
   /*
    * 下面是汇编程序中使用的缓冲队列结构地址表。通过修改这个表，
    * 你可以实现虚拟控制台。
    */
   // tty 读写缓冲队列结构地址表。供 rs_io.s 程序使用，用于取得读写缓冲队列结构的地址。
81 struct tty_queue * table_list[]={
82         con_queues + 0, con_queues + 1,        // 前台控制台读、写队列结构地址。
83         rs_queues + 0, rs_queues + 1,          // 串行终端 1 读、写队列结构地址。
84         rs_queues + 3, rs_queues + 4           // 串行终端 2 读、写队列结构地址。
85         };
86
   //// 改变前台控制台。
   // 将前台控制台设定为指定的虚拟控制台。
   // 参数：new_console - 指定的新控制台号。
87 void change_console(unsigned int new_console)
88 {
   // 如果参数指定的控制台已经在前台或者参数无效，则退出。否则设置当前前台控制台号，同
   // 时更新 table_list[]中的前台控制台读、写队列结构地址。最后更新当前前台控制台屏幕。
89         if (new_console == fg_console || new_console >= NR_CONSOLES)
90                 return;
91         fg_console = new_console;
92         table_list[0] = con_queues + 0 + fg_console*3;
93         table_list[1] = con_queues + 1 + fg_console*3;
94         update_screen();                    // kernel/chr_drv/console.c，936 行。
95 }
96
   //// 如果队列缓冲区空则让进程进入可中断睡眠状态。
   // 参数：queue - 指定队列的指针。
   // 进程在取队列缓冲区中字符之前需要调用此函数加以验证。如果当前进程没有信号要处理，
   // 并且指定的队列缓冲区空，则让进程进入可中断睡眠状态，并让队列的进程等待指针指向
   // 该进程。
97 static void sleep_if_empty(struct tty_queue * queue)
98 {
99         cli();
100        while (!(current->signal & ~current->blocked) && EMPTY(queue))
101                interruptible_sleep_on(&queue->proc_list);
102        sti();
103 }
104
   //// 若队列缓冲区满则让进程进入可中断的睡眠状态。
   // 参数：queue - 指定队列的指针。
   // 进程在往队列缓冲区中写入字符之前需要调用此函数判断队列情况。
105 static void sleep_if_full(struct tty_queue * queue)
106 {
```

```
    // 如果队列缓冲区不满则返回退出。否则若进程没有信号需要处理，并且队列缓冲区中空闲剩
    // 余区长度 < 128，则让进程进入可中断睡眠状态，并让该队列的进程等待指针指向该进程。
107         if (!FULL(queue))
108                 return;
109         cli();
110         while (!(current->signal & ~current->blocked) && LEFT(queue)<128)
111                 interruptible_sleep_on(&queue->proc_list);
112         sti();
113 }
114
    //// 等待按键。
    // 如果前台控制台读队列缓冲区空，则让进程进入可中断睡眠状态。
115 void wait_for_keypress(void)
116 {
117         sleep_if_empty(tty_table[fg_console].secondary);
118 }
119
    //// 复制成规范模式字符序列。
    // 根据终端 termios 结构中设置的各种标志，将指定 tty 终端读队列缓冲区中的字符复制转换
    // 成规范模式（熟模式）字符并存放在辅助队列（规范模式队列）中。
    // 参数：tty - 指定终端的 tty 结构指针。
120 void copy_to_cooked(struct tty_struct * tty)
121 {
122         signed char c;
123
    // 首先检查当前终端 tty 结构中缓冲队列指针是否有效。如果三个队列指针都是 NULL，则说明
    // 内核 tty 初始化函数有问题。
124         if (!(tty->read_q || tty->write_q || tty->secondary)) {
125                 printk("copy_to_cooked: missing queues\n\r");
126                 return;
127         }
    // 否则我们根据终端 termios 结构中的输入和本地标志，对从 tty 读队列缓冲区中取出的每个
    // 字符进行适当的处理，然后放入辅助队列 secondary 中。在下面循环体中，如果此时读队列
    // 缓冲区已经取空或者辅助队列缓冲区已经放满字符，就退出循环体。否则程序就从读队列缓
    // 冲区尾指针处取一字符，并把尾指针前移一个字符位置。然后根据该字符代码值进行处理。
    // 另外，如果定义了_POSIX_VDISABLE（\0），那么在对字符处理过程忠，若字符代码值等于
    // _POSIX_VDISABLE 的值时，表示禁止使用相应特殊控制字符的功能。
128         while (1) {
129                 if (EMPTY(tty->read_q))
130                         break;
131                 if (FULL(tty->secondary))
132                         break;
133                 GETCH(tty->read_q,c);           // 取一字符到 c，并前移尾指针。
    // 如果该字符是回车符 CR（13），那么若回车转换行标志 CRNL 置位，则将字符转换为换行符
    // NL（10）。否则如果忽略回车标志 NOCR 置位，则忽略该字符，继续处理其他字符。如果字
    // 符是换行符 NL（10），并且换行转回车标志 NLCR 置位，则将其转换为回车符 CR（13）。
134                 if (c==13) {
135                         if (I_CRNL(tty))
136                                 c=10;
137                         else if (I_NOCR(tty))
138                                 continue;
139                 } else if (c==10 && I_NLCR(tty))
140                         c=13;
```

```
    // 如果大写转小写输入标志 UCLC 置位，则将该字符转换为小写字符。
141                 if (I_UCLC(tty))
142                         c=tolower(c);
    // 如果本地模式标志集中规范模式标志 CANON 已置位，则对读取的字符进行以下处理。 首先，
    // 如果该字符是键盘终止控制字符 KILL(^U)，则对已输入的当前行执行删除处理。删除一行字
    // 符的循环过程如是：如果 tty 辅助队列不空，并且取出的辅助队列中最后一个字符不是换行
    // 符 NL（10），并且该字符不是文件结束字符（^D），则循环执行下列代码：
    // 如果本地回显标志 ECHO 置位，那么：若字符是控制字符（值 < 32），则往 tty 写队列中放
    // 入擦除控制字符 ERASE（^H）。然后再放入一个擦除字符 ERASE，并且调用该 tty 写函数，把
    // 写队列中的所有字符输出到终端屏幕上。 另外，因为控制字符在放入写队列时需要用 2 个字
    // 节表示（例如^V），因此要求特别对控制字符多放入一个 ERASE。最后将 tty 辅助队列头指针
    // 后退 1 字节。另外，如果定义了_POSIX_VDISABLE（\0），那么在对字符处理过程忠，若字符
    // 代码值等于 _POSIX_VDISABLE 的值时，表示禁止使用相应特殊控制字符的功能。
143                 if (L_CANON(tty)) {
144                         if ((KILL_CHAR(tty) != _POSIX_VDISABLE) &&
145                             (c==KILL_CHAR(tty))) {
146                                 /* deal with killing the input line */
147                                 while(!(EMPTY(tty->secondary) ||
148                                         (c=LAST(tty->secondary))==10 ||
149                                         ((EOF_CHAR(tty) != _POSIX_VDISABLE) &&
150                                          (c==EOF_CHAR(tty))))) {
151                                         if (L_ECHO(tty)) {    // 若本地回显标志置位。
152                                                 if (c<32)     // 控制字符要删 2 字节。
153                                                         PUTCH(127,tty->write_q);
154                                                 PUTCH(127,tty->write_q);
155                                                 tty->write(tty);
156                                         }
157                                         DEC(tty->secondary->head);
158                                 }
159                                 continue;          // 继续读取读队列中字符进行处理。
160                         }
    // 如果该字符是删除控制字符 ERASE（^H），那么：如果 tty 的辅助队列为空，或者其最后一个
    // 字符是换行符 NL(10)，或者是文件结束符，则继续处理其他字符。如果本地回显标志 ECHO 置
    // 位，那么：若字符是控制字符（值< 32），则往 tty 的写队列中放入擦除字符 ERASE。再放入
    // 一个擦除字符 ERASE，并且调用该 tty 的写函数。最后将 tty 辅助队列头指针后退 1 字节，继
    // 续处理其他字符。同样地，如果定义了_POSIX_VDISABLE（\0），那么在对字符处理过程忠，
    // 若字符代码值等于 _POSIX_VDISABLE 的值时，表示禁止使用相应特殊控制字符的功能。
161                         if ((ERASE_CHAR(tty) != _POSIX_VDISABLE) &&
162                             (c==ERASE_CHAR(tty))) {
163                                 if (EMPTY(tty->secondary) ||
164                                    (c=LAST(tty->secondary))==10 ||
165                                    ((EOF_CHAR(tty) != _POSIX_VDISABLE) &&
166                                     (c==EOF_CHAR(tty))))
167                                         continue;
168                                 if (L_ECHO(tty)) {          // 若本地回显标志置位。
169                                         if (c<32)
170                                                 PUTCH(127,tty->write_q);
171                                         PUTCH(127,tty->write_q);
172                                         tty->write(tty);
173                                 }
174                                 DEC(tty->secondary->head);
175                                 continue;
176                         }
```

```
177                 }
    // 如果设置了 IXON 标志，则使终端停止/开始输出控制字符起作用。如果没有设置此标志，那
    // 么停止和开始字符将被作为一般字符供进程读取。在这段代码中，如果读取的字符是停止字
    // 符 STOP（^S），则置 tty 停止标志，让 tty 暂停输出。同时丢弃该特殊控制字符（不放入
    // 辅助队列中），并继续处理其他字符。如果字符是开始字符 START（^Q），则复位 tty 停止
    // 标志，恢复 tty 输出。同时丢弃该控制字符，并继续处理其他字符。
    // 对于控制台来说，这里的 tty->write()是 console.c 中的 con_write()函数。因此控制台将
    // 由于发现 stopped=1 而会立刻暂停在屏幕上显示新字符（chr_drv/console.c，第 586 行）。
    // 对于伪终端也是由于设置了终端 stopped 标志而会暂停写操作（chr_drv/pty.c，第 24 行）。
    // 对于串行终端，也应该在发送终端过程中根据终端 stopped 标志暂停发送，但本版未实现。
178                 if (I_IXON(tty)) {
179                         if ((STOP_CHAR(tty) != _POSIX_VDISABLE) &&
180                             (c==STOP_CHAR(tty))) {
181                                 tty->stopped=1;
182                                 tty->write(tty);
183                                 continue;
184                         }
185                         if ((START_CHAR(tty) != _POSIX_VDISABLE) &&
186                             (c==START_CHAR(tty))) {
187                                 tty->stopped=0;
188                                 tty->write(tty);
189                                 continue;
190                         }
191                 }
    // 若输入模式标志集中 ISIG 标志置位，表示终端键盘可以产生信号，则在收到控制字符 INTR、
    // QUIT、SUSP 或 DSUSP 时，需要为进程产生相应的信号。 如果该字符是键盘中断符（^C），
    // 则向当前进程之进程组中所有进程发送键盘中断信号 SIGINT，并继续处理下一字符。 如果该
    // 字符是退出符（^\），则向当前进程之进程组中所有进程发送键盘退出信号 SIGQUIT，并继续
    // 处理下一字符。如果字符是暂停符（^Z），则向当前进程发送暂停信号 SIGTSTP。同样，若定
    // 义了_POSIX_VDISABLE（\0），那么在对字符处理过程忠，若字符代码值等于_POSIX_VDISABLE
    // 的值时，表示禁止使用相应特殊控制字符的功能。以下不再啰嗦了 :-)
192                 if (L_ISIG(tty)) {
193                         if ((INTR_CHAR(tty) != _POSIX_VDISABLE) &&
194                             (c==INTR_CHAR(tty))) {
195                                 kill_pg(tty->pgrp, SIGINT, 1);
196                                 continue;
197                         }
198                         if ((QUIT_CHAR(tty) != _POSIX_VDISABLE) &&
199                             (c==QUIT_CHAR(tty))) {
200                                 kill_pg(tty->pgrp, SIGQUIT, 1);
201                                 continue;
202                         }
203                         if ((SUSPEND_CHAR(tty) != _POSIX_VDISABLE) &&
204                             (c==SUSPEND_CHAR(tty))) {
205                                 if (!is_orphaned_pgrp(tty->pgrp))
206                                         kill_pg(tty->pgrp, SIGTSTP, 1);
207                                 continue;
208                         }
209                 }
    // 如果该字符是换行符 NL（10），或者是文件结束符 EOF（4，^D），表示一行字符已处理完，
    // 则把辅助缓冲队列中当前含有字符行数值 secondary.data 增 1。如果在函数 tty_read()中取
    // 走一行字符，该值即会被减 1，参见 315 行。
210                 if (c==10 || (EOF_CHAR(tty) != _POSIX_VDISABLE &&
```

```
211                             c==EOF_CHAR(tty)))
212                         tty->secondary->data++;
    // 如果本地模式标志集中回显标志 ECHO 在置位状态，那么，如果字符是换行符 NL（10），则将
    // 换行符 NL（10）和回车符 CR（13）放入 tty 写队列缓冲区中；如果字符是控制字符（值<32）
    // 并且回显控制字符标志 ECHOCTL 置位，则将字符'^'和字符 c+64 放入 tty 写队列中（也即会
    // 显示^C、^H 等）；否则将该字符直接放入 tty 写缓冲队列中。最后调用该 tty 写操作函数。
213                 if (L_ECHO(tty)) {
214                         if (c==10) {
215                                 PUTCH(10,tty->write_q);
216                                 PUTCH(13,tty->write_q);
217                         } else if (c<32) {
218                                 if (L_ECHOCTL(tty)) {
219                                         PUTCH('^',tty->write_q);
220                                         PUTCH(c+64,tty->write_q);
221                                 }
222                         } else
223                                 PUTCH(c,tty->write_q);
224                         tty->write(tty);
225                 }
    // 每一次循环末将处理过的字符放入辅助队列中。
226                 PUTCH(c,tty->secondary);
227         }
    // 最后在退出循环体后唤醒等待该辅助缓冲队列的进程（如果有的话）。
228         wake_up(&tty->secondary->proc_list);
229 }
230
231 /*
232  * Called when we need to send a SIGTTIN or SIGTTOU to our process
233  * group
234  *
235  * We only request that a system call be restarted if there was if the
236  * default signal handler is being used.  The reason for this is that if
237  * a job is catching SIGTTIN or SIGTTOU, the signal handler may not want
238  * the system call to be restarted blindly.  If there is no way to reset the
239  * terminal pgrp back to the current pgrp (perhaps because the controlling
240  * tty has been released on logout), we don't want to be in an infinite loop
241  * while restarting the system call, and have it always generate a SIGTTIN
242  * or SIGTTOU.  The default signal handler will cause the process to stop
243  * thus avoiding the infinite loop problem.  Presumably the job-control
244  * cognizant parent will fix things up before continuging its child process.
245  */
    /* 当需要发送信号 SIGTTIN 或 SIGTTOU 到我们进程组中所有进程时就会调用该函数。
     *
     * 在进程使用默认信号处理句柄情况下，我们仅要求一个系统调用被重新启动，如果
     * 有系统调用因本信号而被中断。这样做的原因是，如果一个作业正在捕获 SIGTTIN
     * 或 SIGTTOU 信号，那么相应信号句柄并不会希望系统调用被盲目地重新启动。如果
     * 没有其他方法把终端的 pgrp 复位到当前 pgrp（例如可能由于在 logout 时控制终端
     * 已被释放），那么我们并不希望在重新启动系统调用时掉入一个无限循环中，并且
     * 总是产生 SIGTTIN 或 SIGTTOU 信号。默认的信号句柄会使得进程停止，因而可以
     * 避免无限循环问题。这里假设可识别作业控制的父进程会在继续执行其子进程之前
     * 把问题搞定。
     */
    //// 向使用终端的进程组中所有进程发送信号。
```

```
    // 在后台进程组中的一个进程访问控制终端时，该函数用于向后台进程组中的所有进程发送
    // SIGTTIN 或 SIGTTOU 信号。无论后台进程组中的进程是否已经阻塞或忽略掉了这两个信号，
    // 当前进程都将立刻退出读写操作而返回。
246 int tty_signal(int sig, struct tty_struct *tty)
247 {
    // 我们不希望停止一个孤儿进程组中的进程（参见文件 kernel/exit.c 中第 232 行上的说明）。
    // 因此如果当前进程组是孤儿进程组，就出错返回。否则就向当前进程组所有进程发送指定信
    // 号 sig。
248         if (is_orphaned_pgrp(current->pgrp))
249                 return -EIO;            /* don't stop an orphaned pgrp */
250         (void) kill_pg(current->pgrp,sig,1);    // 发送信号 sig。
    // 如果这个信号被当前进程阻塞（屏蔽），或者被当前进程忽略掉，则出错返回。否则，如果
    // 当前进程对信号 sig 设置了新的处理句柄，那么就返回我们可被中断的信息。否则就返回在
    // 系统调用重新启动后可以继续执行的信息。
251         if ((current->blocked & (1<<(sig-1))) ||
252             ((int) current->sigaction[sig-1].sa_handler == 1))
253                 return -EIO;            /* Our signal will be ignored */
254         else if (current->sigaction[sig-1].sa_handler)
255                 return -EINTR;          /* We _will_ be interrupted :-) */
256         else
257                 return -ERESTARTSYS;    /* We _will_ be interrupted :-) */
258                                         /* (but restart after we continue) */
259 }
260
    //// tty 读函数。
    // 从终端辅助缓冲队列中读取指定数量的字符，放到用户指定的缓冲区中。
    // 参数：channel - 子设备号；buf - 用户缓冲区指针；nr - 欲读字节数。
    // 返回已读字节数。
261 int tty_read(unsigned channel, char * buf, int nr)
262 {
263         struct tty_struct * tty;
264         struct tty_struct * other_tty = NULL;
265         char c, * b=buf;
266         int minimum,time;
267
    // 首先判断参数有效性并取终端的 tty 结构指针。如果 tty 终端的三个缓冲队列指针都是 NULL，
    // 则返回 EIO 出错信息。如果 tty 终端是一个伪终端，则再取得另一个对应伪终端的 tty 结构
    // other_tty。
268         if (channel > 255)
269                 return -EIO;
270         tty = TTY_TABLE(channel);
271         if (!(tty->write_q || tty->read_q || tty->secondary))
272                 return -EIO;
    // 如果当前进程使用的是这里正在处理的 tty 终端，但该终端的进程组号却与当前进程组号不
    // 同，表示当前进程是后台进程组中的一个进程，即进程不在前台。于是我们要停止当前进程
    // 组的所有进程。因此这里就需要向当前进程组发送 SIGTTIN 信号，并返回等待成为前台进程
    // 组后再执行读操作。
273         if ((current->tty == channel) && (tty->pgrp != current->pgrp))
274                 return(tty_signal(SIGTTIN, tty));
    // 如果当前终端是伪终端，那么对应的另一个伪终端就是 other_tty。若这里 tty 是主伪终端，
    // 那么 other_tty 就是对应的从伪终端，反之也然。
275         if (channel & 0x80)
276                 other_tty = tty_table + (channel ^ 0x40);
```

```
    // 然后根据 VTIME 和 VMIN 对应的控制字符数组值设置读字符操作超时定时值 time 和最少需
    // 要读取的字符个数 minimum。在非规范模式下，这两个是超时定时值。VMIN 表示为了满足读
    // 操作而需要读取的最少字符个数。VTIME 是一个 1/10 秒计数计时值。
277         time = 10L*tty->termios.c_cc[VTIME];        // 设置读操作超时定时值。
278         minimum = tty->termios.c_cc[VMIN];          // 最少需要读取的字符个数。
    // 如果 tty 终端处于规范模式，则设置最小要读取字符数 minimum 等于进程欲读字符数 nr。同
    // 时把进程读取 nr 字符的超时时间值设置为极大值（不会超时）。否则说明终端处于非规范模
    // 式下，若此时设置了最少读取字符数 minimum，则先临时设置进城读超时定时值为无限大，以
    // 让进程先读取辅助队列中已有字符。如果读到的字符数不足 minimum 的话，后面代码会根据
    // 指定的超时值 time 来设置进程的读超时值 timeout，并会等待读取其余字符。参见 328 行。
    // 若此时没有设置最少读取字符数 minimum（为 0），则将其设置为进程欲读字符数 nr，并且如
    // 果设置了超时定时值 time 的话，就把进程读字符超时定时值 timeout 设置为系统当前时间值
    //  + 指定的超时值 time，同时复位 time。 另外，如果以上设置的最少读取字符数 minimum 大
    // 于进程欲读取的字符数 nr，则让 minimum=nr。即对于规范模式下的读取操作，它不受 VTIME
    // 和 VMIN 对应控制字符值的约束和控制，它们仅在非规范模式（生模式）操作中起作用。
279         if (L_CANON(tty)) {
280                 minimum = nr;
281                 current->timeout = 0xffffffff;
282                 time = 0;
283         } else if (minimum)
284                 current->timeout = 0xffffffff;
285         else {
286                 minimum = nr;
287                 if (time)
288                         current->timeout = time + jiffies;
289                 time = 0;
290         }
291         if (minimum>nr)
292                 minimum = nr;                   // 最多读取要求的字符数。

    // 现在我们开始从辅助队列中循环取出字符并放到用户缓冲区 buf 中。当欲读的字节数大于 0，
    // 则执行以下循环操作。在循环过程中，如果当前终端是伪终端，那么我们就执行其对应的另
    // 一个伪终端的写操作函数，让另一个伪终端把字符写入当前伪终端辅助队列缓冲区中。即让
    // 另一终端把写队列缓冲区中字符复制到当前伪终端读队列缓冲区中，并经行规程函数转换后
    // 放入当前伪终端辅助队列中。
293         while (nr>0) {
294                 if (other_tty)
295                         other_tty->write(other_tty);
    // 如果 tty 辅助缓冲队列为空，或者设置了规范模式标志并且 tty 读队列缓冲区未满，并且辅
    // 助队列中字符行数为 0，那么，如果没有设置过进程读字符超时值（为 0），或者当前进程
    // 目前收到信号，就先退出循环体。否则如果本终端是一个从伪终端，并且其对应的主伪终端
    // 已经挂断，那么我们也退出循环体。如果不是以上这两种情况，我们就让当前进程进入可中
    // 断睡眠状态，返回后继续处理。由于规范模式时内核以行为单位为用户提供数据，因此在该
    // 模式下辅助队列中必须起码有一行字符可供取用，即 secondary.data 起码是 1 才行。
296                 cli();
297                 if (EMPTY(tty->secondary) || (L_CANON(tty) &&
298                     !FULL(tty->read_q) && !tty->secondary->data)) {
299                         if (!current->timeout ||
300                           (current->signal & ~current->blocked)) {
301                                 sti();
302                                 break;
303                         }
```

```
304                         if (IS_A_PTY_SLAVE(channel) && C_HUP(other_tty))
305                                 break;
306                         interruptible_sleep_on(&tty->secondary->proc_list);
307                         sti();
308                         continue;
309                 }
310                 sti();
    // 下面开始正式执行取字符操作。需读字符数 nr 依次递减，直到 nr=0 或者辅助缓冲队列为空。
    // 在这个循环过程中，首先取辅助缓冲队列字符 c，并且把缓冲队列尾指针 tail 向右移动一个
    // 字符位置。如果所取字符是文件结束符（^D）或者是换行符 NL（10），则把辅助缓冲队列中
    // 含有字符行数值减 1。 如果该字符是文件结束符（^D）并且规范模式标志成置位状态，则中
    // 断本循环，否则说明现在还没有遇到文件结束符或者正处于原始（非规范）模式。在这种模
    // 式中用户以字符流作为读取对象，也不识别其中的控制字符（如文件结束符）。于是将字符
    // 直接放入用户数据缓冲区 buf 中，并把欲读字符数减 1。此时如果欲读字符数已为 0 则中断
    // 循环。另外，如果终端处于规范模式并且读取的字符是换行符 NL（10），则也退出循环。
    // 除此之外，只要还没有取完欲读字符数 nr 并且辅助队列不空，就继续取队列中的字符。
311                 do {
312                         GETCH(tty->secondary,c);
313                         if ((EOF_CHAR(tty) != _POSIX_VDISABLE &&
314                              c==EOF_CHAR(tty)) || c==10)
315                                 tty->secondary->data--;
316                         if ((EOF_CHAR(tty) != _POSIX_VDISABLE &&
317                              c==EOF_CHAR(tty)) && L_CANON(tty))
318                                 break;
319                         else {
320                                 put_fs_byte(c,b++);
321                                 if (!--nr)
322                                         break;
323                         }
324                         if (c==10 && L_CANON(tty))
325                                 break;
326                 } while (nr>0 && !EMPTY(tty->secondary));
    // 执行到此，那么如果 tty 终端处于规范模式下，说明我们可能读到了换行符或者遇到了文件
    // 结束符。如果是处于非规范模式下，那么说明我们已经读取了 nr 个字符，或者辅助队列已经
    // 被取空了。于是我们首先唤醒等待读队列的进程，然后看看是否设置过超时定时值 time。如
    // 果超时定时值 time 不为 0，我们就要求等待一定的时间让其他进程可以把字符写入读队列中。
    // 于是设置进程读超时定时值为系统当前时间 jiffies + 读超时值 time。当然，如果终端处于
    // 规范模式，或者已经读取了 nr 个字符，我们就可以直接退出这个大循环了。
327                 wake_up(&tty->read_q->proc_list);
328                 if (time)
329                         current->timeout = time+jiffies;
330                 if (L_CANON(tty) || b-buf >= minimum)
331                         break;
332         }
    // 此时读取 tty 字符循环操作结束，因此复位进程的读取超时定时值 timeout。如果此时当前进
    // 程已收到信号并且还没有读取到任何字符，则以重新启动系统调用号返回。否则就返回已读取
    // 的字符数(b-buf)。
333         current->timeout = 0;
334         if ((current->signal & ~current->blocked) && !(b-buf))
335                 return -ERESTARTSYS;
336         return (b-buf);
337 }
338
```

```
    //// tty 写函数。
    // 把用户缓冲区中的字符放入 tty 写队列缓冲区中。
    // 参数：channel - 子设备号；buf - 缓冲区指针；nr - 写字节数。
    // 返回已写字节数。
339 int tty_write(unsigned channel, char * buf, int nr)
340 {
341         static cr_flag=0;
342         struct tty_struct * tty;
343         char c, *b=buf;
344
    // 首先判断参数有效性并取终端的 tty 结构指针。如果 tty 终端的三个缓冲队列指针都是 NULL，
    // 则返回 EIO 出错信息。
345         if (channel > 255)
346                 return -EIO;
347         tty = TTY_TABLE(channel);
348         if (!(tty->write_q || tty->read_q || tty->secondary))
349                 return -EIO;
    // 如果若终端本地模式标志集中设置了 TOSTOP，表示后台进程输出时需要发送信号 SIGTTOU。
    // 如果当前进程使用的是这里正在处理的 tty 终端，但该终端的进程组号却与当前进程组号不
    // 同，即表示当前进程是后台进程组中的一个进程，即进程不在前台。于是我们要停止当前进
    // 程组的所有进程。因此这里就需要向当前进程组发送 SIGTTOU 信号，并返回等待成为前台进
    // 程组后再执行写操作。
350         if (L_TOSTOP(tty) &&
351             (current->tty == channel) && (tty->pgrp != current->pgrp))
352                 return(tty_signal(SIGTTOU, tty));
    // 现在我们开始从用户缓冲区 buf 中循环取出字符并放到写队列缓冲区中。当欲写字节数大于 0，
    // 则执行以下循环操作。在循环过程中，如果此时 tty 写队列已满，则当前进程进入可中断的睡
    // 眠状态。如果当前进程有信号要处理，则退出循环体。
353         while (nr>0) {
354                 sleep_if_full(tty->write_q);
355                 if (current->signal & ~current->blocked)
356                         break;
    // 当要写的字符数 nr 还大于 0 并且 tty 写队列缓冲区不满，则循环执行以下操作。首先从用户
    // 缓冲区中取 1 字节。如果终端输出模式标志集中的执行输出处理标志 OPOST 置位，则执行对
    // 字符的后处理操作。
357                 while (nr>0 && !FULL(tty->write_q)) {
358                         c=get_fs_byte(b);
359                         if (O_POST(tty)) {
    // 如果该字符是回车符'\r'（CR，13）并且回车符转换行符标志 OCRNL 置位，则将该字符换成
    // 换行符'\n'（NL，10）；否则如果该字符是换行符'\n'（NL，10）并且换行转回车功能标志
    // ONLRET 置位的话，则将该字符换成回车符'\r'（CR，13）。
360                                 if (c=='\r' && O_CRNL(tty))
361                                         c='\n';
362                                 else if (c=='\n' && O_NLRET(tty))
363                                         c='\r';
    // 如果该字符是换行符'\n' 并且回车标志 cr_flag 没有置位，但换行转回车-换行标志 ONLCR
    // 置位的话，则将 cr_flag 标志置位，并将一回车符放入写队列中。然后继续处理下一个字符。
    // 如果小写转大写标志 OLCUC 置位的话，就将该字符转成大写字符。
364                                 if (c=='\n' && !cr_flag && O_NLCR(tty)) {
365                                         cr_flag = 1;
366                                         PUTCH(13,tty->write_q);
367                                         continue;
368                                 }
```

```
369                             if (O_LCUC(tty))                // 小写转成大写字符。
370                                     c=toupper(c);
371                     }
    // 接着把用户数据缓冲指针 b 前移 1 字节；欲写字节数减 1 字节；复位 cr_flag 标志，并将该
    // 字节放入 tty 写队列中。
372                     b++; nr--;
373                     cr_flag = 0;
374                     PUTCH(c,tty->write_q);
375             }
    // 若要求的字符全部写完，或者写队列已满，则程序退出循环。此时会调用对应 tty 写函数，
    // 把写队列缓冲区中的字符显示在控制台屏幕上，或者通过串行端口发送出去。如果当前处
    // 理的 tty 是控制台终端，那么 tty->write()调用的是 con_write()；如果 tty 是串行终端，
    // 则 tty->write()调用的是 rs_write()函数。若还有字节要写，则等待写队列中字符取走。
    // 所以这里调用调度程序，先去执行其他任务。
376             tty->write(tty);
377             if (nr>0)
378                     schedule();
379         }
380         return (b-buf);                 // 最后返回写入的字节数。
381 }
382
383 /*
384  * Jeh, sometimes I really like the 386.
385  * This routine is called from an interrupt,
386  * and there should be absolutely no problem
387  * with sleeping even in an interrupt (I hope).
388  * Of course, if somebody proves me wrong, I'll
389  * hate intel for all time :-). We'll have to
390  * be careful and see to reinstating the interrupt
391  * chips before calling this, though.
392  *
393  * I don't think we sleep here under normal circumstances
394  * anyway, which is good, as the task sleeping might be
395  * totally innocent.
396  */
    /*
     * 呵，有时我真得很喜欢 386。该子程序被从一个中断处理程序中
     * 调用，并且即使在中断处理程序中睡眠也应该绝对没有问题（我
     * 希望如此）。当然，如果有人证明我是错的，那么我将憎恨 intel
     * 一辈子☺。但是我们必须小心，在调用该子程序之前需要恢复中断。
     *
     * 我不认为在通常环境下会处在这里睡眠，这样很好，因为任务睡眠
     * 是完全任意的。
     */
    //// tty 中断处理调用函数 - 字符规范模式处理。
    // 参数：tty - 指定的 tty 终端号。
    // 将指定 tty 终端队列缓冲区中的字符复制或转换成规范(熟)模式字符并存放在辅助队列中。
    // 该函数会在串口读字符中断（rs_io.s，109）和键盘中断（kerboard.S，69）中被调用。
397 void do_tty_interrupt(int tty)
398 {
399         copy_to_cooked(TTY_TABLE(tty));
400 }
401
```

```
    //// 字符设备初始化函数。空，为以后扩展做准备。
402 void chr_dev_init(void)
403 {
404 }
405
    //// tty 终端初始化函数。
    // 初始化所有终端缓冲队列，初始化串口终端和控制台终端。
406 void tty_init(void)
407 {
408         int i;
409
    // 首先初始化所有终端的缓冲队列结构，设置初值。对于串行终端的读/写缓冲队列，将它们的
    // data 字段设置为串行端口基地址值。串口 1 是 0x3f8，串口 2 是 0x2f8。然后先初步设置所有
    // 终端的 tty 结构。其中特殊字符数组 c_cc[]设置的初值定义在 include/linux/tty.h 文件中。
410         for (i=0 ; i < QUEUES ; i++)
411                 tty_queues[i] = (struct tty_queue) {0,0,0,0,""};
412         rs_queues[0] = (struct tty_queue) {0x3f8,0,0,0,""};
413         rs_queues[1] = (struct tty_queue) {0x3f8,0,0,0,""};
414         rs_queues[3] = (struct tty_queue) {0x2f8,0,0,0,""};
415         rs_queues[4] = (struct tty_queue) {0x2f8,0,0,0,""};
416         for (i=0 ; i<256 ; i++) {
417                 tty_table[i] =  (struct tty_struct) {
418                         {0, 0, 0, 0, 0, INIT_C_CC},
419                         0, 0, 0, NULL, NULL, NULL, NULL
420                 };
421         }
    // 接着初始化控制台终端（console.c，834 行）。把 con_init()放在这里，是因为我们需要根
    // 据显示卡类型和显示内存容量来确定系统中虚拟控制台的数量 NR_CONSOLES。该值被用于随后
    // 的控制台 tty 结构初始化循环中。对于控制台的 tty 结构，425--430 行是 tty 结构中包含的
    // termios 结构字段。其中输入模式标志集被初始化为 ICRNL 标志；输出模式标志被初始化为含
    // 有后处理标志 OPOST 和把 NL 转换成 CRNL 的标志 ONLCR；本地模式标志集被初始化含有 IXON、
    // ICANON、ECHO、ECHOCTL 和 ECHOKE 标志；控制字符数组 c_cc[]被设置含有初始值 INIT_C_CC。
    // 435 行上初始化控制台终端 tty 结构中的读缓冲、写缓冲和辅助缓冲队列结构，它们分别指向
    // tty 缓冲队列结构数组 tty_table[]中的相应结构项。参见 61--73 行上的相关说明。
422         con_init();
423         for (i = 0 ; i<NR_CONSOLES ; i++) {
424                 con_table[i] = (struct tty_struct) {
425                         {ICRNL,         /* change incoming CR to NL */   /* CR 转 NL */
426                         OPOST|ONLCR,    /* change outgoing NL to CRNL */ /*NL 转 CRNL*/
427                         0,                                               // 控制模式标志集。
428                         IXON | ISIG | ICANON | ECHO | ECHOCTL | ECHOKE, // 本地标志集。
429                         0,              /* console termio */             // 线路规程，0 -- TTY。
430                         INIT_C_CC},                                      // 控制字符数组 c_cc[]。
431                         0,              /* initial pgrp */               // 所属初始进程组 pgrp。
432                         0,              /* initial session */            // 初始会话组 session。
433                         0,              /* initial stopped */            // 初始停止标志 stopped。
434                         con_write,                                       // 控制台写函数。
435                         con_queues+0+i*3,con_queues+1+i*3,con_queues+2+i*3
436                 };
437         }
    // 然后初始化串行终端的 tty 结构各字段。450 行初始化串行终端 tty 结构中的读/写和辅助缓
    // 冲队列结构，它们分别指向 tty 缓冲队列结构数组 tty_table[]中的相应结构项。参见 61--
    // 73 行上的相关说明。
```

```
438         for (i = 0 ; i<NR_SERIALS ; i++) {
439                 rs_table[i] = (struct tty_struct) {
440                         {0, /* no translation */   // 输入模式标志集。0，无须转换。
441                         0,  /* no translation */   // 输出模式标志集。0，无须转换。
442                         B2400 | CS8,          // 控制模式标志集。2400bps，8 位数据位。
443                         0,                          // 本地模式标志集。
444                         0,                          // 线路规程，0 -- TTY。
445                         INIT_C_CC},                 // 控制字符数组。
446                         0,                          // 所属初始进程组。
447                         0,                          // 初始会话组。
448                         0,                          // 初始停止标志。
449                         rs_write,                   // 串口终端写函数。
450                         rs_queues+0+i*3,rs_queues+1+i*3,rs_queues+2+i*3   // 三个队列。
451                 };
452         }
    // 然后再初始化伪终端使用的 tty 结构。伪终端是配对使用的，即一个主（master）伪终端配
    // 有一个从（slave）伪终端。因此对它们都要进行初始化设置。在循环中，我们首先初始化
    // 每个主伪终端的 tty 结构，然后再初始化其对应的从伪终端的 tty 结构。
453         for (i = 0 ; i<NR_PTYS ; i++) {
454                 mpty_table[i] = (struct tty_struct) {
455                         {0, /* no translation */   // 输入模式标志集。0，无须转换。
456                         0,  /* no translation */   // 输出模式标志集。0，无须转换。
457                         B9600 | CS8,          // 控制模式标志集。9600bps，8 位数据位。
458                         0,                          // 本地模式标志集。
459                         0,                          // 线路规程，0 -- TTY。
460                         INIT_C_CC},                 // 控制字符数组。
461                         0,                          // 所属初始进程组。
462                         0,                          // 所属初始会话组。
463                         0,                          // 初始停止标志。
464                         mpty_write,                 // 主伪终端写函数。
465                         mpty_queues+0+i*3,mpty_queues+1+i*3,mpty_queues+2+i*3
466                 };
467                 spty_table[i] = (struct tty_struct) {
468                         {0, /* no translation */   // 输入模式标志集。0，无须转换。
469                         0,  /* no translation */   // 输出模式标志集。0，无须转换。
470                         B9600 | CS8,          // 控制模式标志集。9600bps，8 位数据位。
471                         IXON | ISIG | ICANON,       // 本地模式标志集。
472                         0,                          // 线路规程，0 -- TTY。
473                         INIT_C_CC},                 // 控制字符数组。
474                         0,                          // 所属初始进程组。
475                         0,                          // 所属初始会话组。
476                         0,                          // 初始停止标志。
477                         spty_write,                 // 从伪终端写函数。
478                         spty_queues+0+i*3,spty_queues+1+i*3,spty_queues+2+i*3
479                 };
480         }
    // 最后初始化串行中断处理程序和串行接口 1 和 2（serial.c，37 行），并显示系统含有的虚拟
    // 控制台数 NR_CONSOLES 和伪终端数 NR_PTYS。
481         rs_init();
482         printk("%d virtual consoles\n\r",NR_CONSOLES);
483         printk("%d pty's\n\r",NR_PTYS);
484 }
485
```