目 录

1 键盘

对于处理键盘输入，我们本能是想用中断去处理。在此之前，我们先学习计算机响应键盘的过程。

1.1 键盘敲击的过程

在键盘中存在一枚叫做键盘编码器的芯片 Intel 8048，在计算机中有一个键盘控制器，是 intel 8042 芯片。

过程如下：

1. 8048 芯片检测到一个键的动作，将相应的扫描码发送给 8042。
2. 8042 讲扫描码转换成相应的 Scan code set1 扫描码，然后放置在输入缓冲区中。
3. 此后 8042 告诉 8259A 产生中断，如果此时键盘又有新的键被按下，8042 将不再接收，一直到缓冲区被清空，8042 才能收到更多的扫描码。

1.2 读取键盘输入

8042 的输出缓冲区的端口是 0x60，我们可以直接去通过 in_byte() 函数去读取。

但为了真正处理键盘输入，我们还需要了解键盘编码：

```
    敲击键盘产生的编码被称为扫描码：
        当一个键被按下时，将产生Make Code，
        当一个键弹起时，产生Break Code。
    除了Pause键之外，每一个按键都对应一个Make Code和一个Break Code。
```

在代码中，为了处理 Break Code 和 Make Code，可以创建一个数组：

```
u32 keymap[NR_SCAN_CODES * MAP_COLS] = {

/* scan-code                !Shift      Shift       E0 XX    */
/* ============================================================================ */
/* 0x00 - none          */  0,          0,          0,
/* 0x01 - ESC           */  ESC,        ESC,        0,
/* 0x02 - '1'           */  '1',        '!',        0,
/* 0x03 - '2'           */  '2',        '@',        0,
/* 0x04 - '3'           */  '3',        '#',        0,
/* 0x05 - '4'           */  '4',        '$',        0,
/* 0x06 - '5'           */  '5',        '%',        0,
/* 0x07 - '6'           */  '6',        '^',        0,
/* 0x08 - '7'           */  '7',        '&',        0,
/* 0x09 - '8'           */  '8',        '*',        0,
/* 0x0A - '9'           */  '9',        '(',        0,
/* 0x0B - '0'           */  '0',        ')',        0,
/* 0x0C - '-'           */  '-',        '_',        0,
```

```
18  /* 0x0D — '='          */  '=',          '+',          0,
19  /* 0x0E — BS           */  BACKSPACE,    BACKSPACE,    0,
20  /* 0x0F — TAB          */  TAB,          TAB,          0,
21  /* 0x10 — 'q'          */  'q',          'Q',          0,
22  /* 0x11 — 'w'          */  'w',          'W',          0,
23  /* 0x12 — 'e'          */  'e',          'E',          0,
24  /* 0x13 — 'r'          */  'r',          'R',          0,
25  /* 0x14 — 't'          */  't',          'T',          0,
26  /* 0x15 — 'y'          */  'y',          'Y',          0,
27  /* 0x16 — 'u'          */  'u',          'U',          0,
28  /* 0x17 — 'i'          */  'i',          'I',          0,
29  /* 0x18 — 'o'          */  'o',          'O',          0,
30  /* 0x19 — 'p'          */  'p',          'P',          0,
31  /* 0x1A — '['          */  '[',          '{',          0,
32  /* 0x1B — ']'          */  ']',          '}',          0,
33  /* 0x1C — CR/LF        */  ENTER,        ENTER,        PAD_ENTER,
34  /* 0x1D — l. Ctrl      */  CTRL_L,       CTRL_L,       CTRL_R,
35  /* 0x1E — 'a'          */  'a',          'A',          0,
36  /* 0x1F — 's'          */  's',          'S',          0,
37  /* 0x20 — 'd'          */  'd',          'D',          0,
38  /* 0x21 — 'f'          */  'f',          'F',          0,
39  /* 0x22 — 'g'          */  'g',          'G',          0,
40  /* 0x23 — 'h'          */  'h',          'H',          0,
41  /* 0x24 — 'j'          */  'j',          'J',          0,
42  /* 0x25 — 'k'          */  'k',          'K',          0,
43  /* 0x26 — 'l'          */  'l',          'L',          0,
44  /* 0x27 — ';'          */  ';',          ':',          0,
45  /* 0x28 — '\''         */  '\'',         '"',          0,
46  /* 0x29 — '`'          */  '`',          '~',          0,
47  /* 0x2A — l. SHIFT     */  SHIFT_L,      SHIFT_L,      0,
48  /* 0x2B — '\'          */  '\\',         '|',          0,
49  /* 0x2C — 'z'          */  'z',          'Z',          0,
50  /* 0x2D — 'x'          */  'x',          'X',          0,
51  /* 0x2E — 'c'          */  'c',          'C',          0,
52  /* 0x2F — 'v'          */  'v',          'V',          0,
53  /* 0x30 — 'b'          */  'b',          'B',          0,
54  /* 0x31 — 'n'          */  'n',          'N',          0,
55  /* 0x32 — 'm'          */  'm',          'M',          0,
56  /* 0x33 — ','          */  ',',          '<',          0,
57  /* 0x34 — '.'          */  '.',          '>',          0,
58  /* 0x35 — '/'          */  '/',          '?',          PAD_SLASH,
59  /* 0x36 — r. SHIFT     */  SHIFT_R,      SHIFT_R,      0,
60  /* 0x37 — '*'          */  '*',          '*',          0,
61  /* 0x38 — ALT          */  ALT_L,        ALT_L,        ALT_R,
62  /* 0x39 — ' '          */  ' ',          ' ',          0,
63  /* 0x3A — CapsLock     */  CAPS_LOCK,    CAPS_LOCK,    0,
64  /* 0x3B — F1           */  F1,       F1,       0,
65  /* 0x3C — F2           */  F2,       F2,       0,
66  /* 0x3D — F3           */  F3,       F3,       0,
67  /* 0x3E — F4           */  F4,       F4,       0,
68  /* 0x3F — F5           */  F5,       F5,       0,
69  /* 0x40 — F6           */  F6,       F6,       0,
70  /* 0x41 — F7           */  F7,       F7,       0,
71  /* 0x42 — F8           */  F8,       F8,       0,
72  /* 0x43 — F9           */  F9,       F9,       0,
73  /* 0x44 — F10          */  F10,      F10,      0,
```

```
74  /* 0x45 — NumLock    */  NUM_LOCK,     NUM_LOCK,     0,
75  /* 0x46 — ScrLock    */  SCROLL_LOCK,      SCROLL_LOCK,      0,
76  /* 0x47 — Home       */  PAD_HOME,     '7',          HOME,
77  /* 0x48 — CurUp      */  PAD_UP,       '8',          UP,
78  /* 0x49 — PgUp       */  PAD_PAGEUP,   '9',          PAGEUP,
79  /* 0x4A — '-'        */  PAD_MINUS,    '-',          0,
80  /* 0x4B — Left       */  PAD_LEFT,     '4',          LEFT,
81  /* 0x4C — MID        */  PAD_MID,      '5',          0,
82  /* 0x4D — Right      */  PAD_RIGHT,    '6',          RIGHT,
83  /* 0x4E — '+'        */  PAD_PLUS,     '+',          0,
84  /* 0x4F — End        */  PAD_END,      '1',          END,
85  /* 0x50 — Down       */  PAD_DOWN,     '2',          DOWN,
86  /* 0x51 — PgDown     */  PAD_PAGEDOWN,     '3',          PAGEDOWN,
87  /* 0x52 — Insert     */  PAD_INS,      '0',          INSERT,
88  /* 0x53 — Delete     */  PAD_DOT,      '.',          DELETE,
89  /* 0x54 — Enter      */  0,        0,        0,
90  /* 0x55 — ???        */  0,        0,        0,
91  /* 0x56 — ???        */  0,        0,        0,
92  /* 0x57 — F11        */  F11,          F11,          0,
93  /* 0x58 — F12        */  F12,          F12,          0,
94  /* 0x59 — ???        */  0,        0,        0,
95  /* 0x5A — ???        */  0,        0,        0,
96  /* 0x5B — ???        */  0,        0,        GUI_L,
97  /* 0x5C — ???        */  0,        0,        GUI_R,
98  /* 0x5D — ???        */  0,        0,        APPS,
99  /* 0x5E — ???        */  0,        0,        0,
100 /* 0x5F — ???        */  0,        0,        0,
101 /* 0x60 — ???        */  0,        0,        0,
102 /* 0x61 — ???        */  0,        0,        0,
103 /* 0x62 — ???        */  0,        0,        0,
104 /* 0x63 — ???        */  0,        0,        0,
105 /* 0x64 — ???        */  0,        0,        0,
106 /* 0x65 — ???        */  0,        0,        0,
107 /* 0x66 — ???        */  0,        0,        0,
108 /* 0x67 — ???        */  0,        0,        0,
109 /* 0x68 — ???        */  0,        0,        0,
110 /* 0x69 — ???        */  0,        0,        0,
111 /* 0x6A — ???        */  0,        0,        0,
112 /* 0x6B — ???        */  0,        0,        0,
113 /* 0x6C — ???        */  0,        0,        0,
114 /* 0x6D — ???        */  0,        0,        0,
115 /* 0x6E — ???        */  0,        0,        0,
116 /* 0x6F — ???        */  0,        0,        0,
117 /* 0x70 — ???        */  0,        0,        0,
118 /* 0x71 — ???        */  0,        0,        0,
119 /* 0x72 — ???        */  0,        0,        0,
120 /* 0x73 — ???        */  0,        0,        0,
121 /* 0x74 — ???        */  0,        0,        0,
122 /* 0x75 — ???        */  0,        0,        0,
123 /* 0x76 — ???        */  0,        0,        0,
124 /* 0x77 — ???        */  0,        0,        0,
125 /* 0x78 — ???        */  0,        0,        0,
126 /* 0x78 — ???        */  0,        0,        0,
127 /* 0x7A — ???        */  0,        0,        0,
128 /* 0x7B — ???        */  0,        0,        0,
129 /* 0x7C — ???        */  0,        0,        0,
```

```
/* 0x7D — ???        */  0,      0,      0,
/* 0x7E — ???        */  0,      0,      0,
/* 0x7F — ???        */  0,      0,      0
};
```

建立这个数组以后，我们就可以通过扫描码直接对应到真正的字母了。

这个数组还考虑到了按住 shift 键时的字母对应关系。

1.3 建立键盘输入缓冲区

因为我们输入键盘时可以会同时按住几个键，所以我们要在中断处理程序中处理这种情况。

考虑到因为扫描码的值和长度不一样，如果直接处理扫描码，可能会搞得代码很冗长。在这里建立一个键盘输入的缓冲区，让中断处理程序将每次收到的扫描码放入这个缓冲区，然后建立一个新的任务专门用来解析这些扫描码并做相应的处理。

键盘缓冲区如下：

```
typedef struct s_kb
{
    char* p_head;
    char* p_tail;
    int count;
    char buf[KB_IN_BYTES];
}KB_INPUT;
```

创建对应的键盘中断处理程序：

```
PRIVATE KB_INPUT kb_in;

PUBLIC void keyboard_handler(int irq)
{
    u8 scan_code = in_byte(0x60);
    if(kb_in.count < KB_IN_BYTES)
    {
        *(kv_in.p_head) = scan_code;
        kb_in.p_head++;
        if(kv_in.p_head == kb_in.buf + KB_IN_BYTES)
            kb_in.p_head = kb_in.buf;
        kb_in.count++;
    }
}

// 初始化键盘中断处理程序
PUBLIC void init_keyboard()
{
    kb_in.count = 0;
    kb_in.p_head = kb_in.p_tail = kb_in.buf;

    put_irq_handler(KEYBOARD_IRQ, keyboard_handler);
    enable_irq(KEYBOARD_IRQ);
```

```
24      }
25
26      // 打开键盘中断
27      PUBLIC int kernel_main()
28      {
29          ...
30          init_keyboard();
31          ...
32      }
```

1.4 添加新的进程处理缓冲区

1.4.1 读取缓冲区

函数体如下：

```
1       PUBLIC void task_tty()
2       {
3           while(1)
4           {
5               keyboard_read();
6           }
7       }
8
9       PUBLIC void keyboard_read()
10      {
11          u8 scan_code;
12          if(kb_in.count > 0)
13          {
14              disable_int();
15              scan_code = *(kb_in.p_tail);
16              kb_in.p_tail++;
17              if(kb_in.p_tail == kb_in.buf + KB_IN_BYTES)
18                  kb_in.p_tail = kb_in.buf;
19              kb_in.count--;
20              enable_int();
21              disp_int(scan_code);
22          }
23      }
```

1.5 解析扫描码

1.5.1 显示普通的字符

查看扫描码，会知道以 0xE0 和 0xE1 开头的扫描码对应的按键一般比较特殊，我们先跳过处理。

下面先进行最基本的字符显示：

```
PUBLIC void keyboard_read()
{
    char output[2];
    int make;

    memset(output, 0, 2);

    if(kb_in.count > 0)
    {
        ...
        // 扫描码已经读入scan_code
        if(scan_code == 0xE1)
        {

        }
        else if(scan_code == 0xE0)
        {

        }
        else
        {
            // 如果make是FALSE，代表这个码是break code，否则就是make code
            make = (scan_code & FLAG_BREAK ? FALSE : TRUE);
            output[0] = keymap[(scan_code&0x7F)*MAP_COLS];

            if(make)
            {
                disp_str(output);
            }
        }
    }
}
```

1.5.2 处理 shift、alt 和 ctrl

处理这些组合按键的算法如下：

1. 首先判断是否是 shift、alt 或者 ctrl。这里对这些按键都声明一个标识符，如果按下就为 1，没按下就为 0，可以直接用 make 这个标识符来对它们赋值。此外，还另外标记一个 key 标识符，如果是这些特殊字符或者 break code，key 就为 0，否则为按键实际值。
2. 还记得我们定义 keymap 数组时，定义了三列数据。第二列数据就是 shift 按下时的字符显示情况。如果在处理中，遇到上述 shift 标识符为 1 的情况，就对列进行相应的操作。如果是其他标识符，就进行其他的操作。
3. 检查 key 标识符，如果是 1 就显示字符。

代码如下：

```
PRIVATE int shift_l;
PRIVATE int shift_r;
PRIVATE int alt_l;
PRIVATE int alt_r;
PRIVATE int ctrl_l;
PRIVATE int ctrl_r;
PRIVATE int caps_lock;
PRIVATE int num_lock;
PRIVATE int scroll_lock;
PRIVATE int column;

PUBLIC void keyboard_read()
{
    u32 key = 0;
    u32* keyrow;

    if(kb_in.count > 0)
    {
        ...
        if(scan_code == 0xE1)
        {

        }
        else if(scan_code == 0xE0)
        {

        }
        else
        {
            // 得到扫描码对应的那一行数据
            keyrow = &keymap[(scan_code & 0x7F) * MAP_COLS];
            column = 0;

            if(shift_l || shift_r)
                column = 1;

            key = keyrow[column];

            switch(key)
            {
                case SHIFT_L:
                    shift_l = make;
                    key = 0;
                    break;
                case SHIFT_R:
                    shift_r = make;
                    key = 0;
                    break;
                case CTRL_L:
                    ctrl_l = make;
                    key = 0;
                    break;
                case CTRL_R:
                    ctrl_r = make;
                    key = 0;
```

```
56                          break;
57                      case ALT_L:
58                          alt_l = make;
59                          key = 0;
60                          break;
61                      case ALT_R:
62                          alt_r = make;
63                          key = 0;
64                          break;
65                      default:
66                          if(!make)
67                              key = 0;
68                          break;
69                  }
70
71                  if(key)
72                  {
73                      output[0] = key;
74                      disp_str(output);
75                  }
76              }
77          }
78      }
```

1.5.3 处理不可打印的字符

这里我们将 keyboard_read() 函数再做一个处理，将它变成单纯读取键盘缓冲区的函数，然后增加一个 in_process() 函数，用于专门处理读出的字符。

注意，我们还处理了不可打印的字符，算法如下：

1. 为每一个不可打印的按键定义一个宏，键值 key 在传入 in_process() 函数之前先根据实际情况选择是否与这些宏相或。
2. 在 in_process() 函数中，再与一个宏相与，从而决定是否打印该字符。

代码如下：

```
1   #define SHIFT_L (0x08 + FLAG_EXT)
2   #define SHIFT_R (0x09 + FLAG_EXT)
3   #define CTRL_L (0x0A + FLAG_EXT)
4   #define CTRL_R (0x0B + FLAG_EXT)
5   #define ALT_L (0x0C + FLAG_EXT)
6   #define ALT_R (0x0D + FLAG_EXT)
7
8   PUBLIC void keyboard_read()
9   {
10      ...
11      // 键值key在传入in_process()函数之前先根据实际情况选择是否与这些宏相或
12      if(make)
13      {
14          key |= (shift_l ? FLAG_SHIFT_L : 0);
```

```
15              key |= (shift_r ? FLAG_SHIFT_R : 0);
16              key |= (ctrl_l ? FLAG_CTRL_L : 0);
17              key |= (ctrl_r ? FLAG_CTRL_R : 0);
18              key |= (alt_l ? FLAG_ALT_L : 0);
19              key |= (alt_r ? FLAG_ALT_R : 0);
20
21              in_process(key);
22          }
23      }
24
25      PUBLIC void in_process(u32 key)
26      {
27          char output[2];
28          memset(output, 0, 2);
29
30          // 在in_process()函数中，再与一个宏相与，从而决定是否打印该字符
31          if(!(key & FLAG_EXT))
32          {
33              output[0] = key & 0xFF;
34              disp_str(output);
35          }
36      }
```

2 TTY 任务

我们接下来要实现 TTY，首先需要知道一些基本框架：

1. 每一个 TTY 都有自己读和写的动作，所以在 keyboard_read() 函数中需要知道当前是在哪个 TTY 下，所以我们需要为这个函数传入一个参数。
2. TTY 任务应该自己负责自己的显示工作，所以 in_process() 函数不再负责显示任务，而是交给 TTY 来完成。
3. 每个 TTY 回显字符时操作的 console 是不同的，所以需要有一个成员来记载其对应的 console 信息。

2.1 TTY 的一些数据结构

为了让 keyboard_read() 函数知道是哪个 TTY 调用它，需要定义一个 TTY 结构。为了将 TTY 结构与特定的 console 关联起来，需要定义一个 console 结构，并作为 TTY 结构的一个成员：

```
1       struct s_console;
2
3       typedef struct s_tty
4       {
5           u32 in_buf[TTY_IN_BYTES];
```

```
6       u32* p_inbuf_head;
7       u32* p_inbuf_tail;
8       int inbuf_count;
9
10      struct s_console* p_console;
11  }TTY;
12
13  typedef struct s_console
14  {
15      unsigned int current_start_addr;
16      unsigned int original_addr;
17      unsigned int v_mem_limit;
18      unsigned int cursor;
19  }CONSOLE;
```

2.2 构建 TTY 框架

首先文字描述一下这个框架：

1. 定义三个 TTY 结构，在 tty 任务中对这三个 TTY 进行循环，一次进行读操作和写操作。
2. 如果控制台是当前控制台，TTY 就可以读取键盘。
3. 在 keyboard_read() 函数中，首先处理键盘扫描码，然后将键值 key 传入 in_process() 函数。
4. 在 in_process() 函数中，将键值 key 写入 TTY 缓冲区。
5. tty 任务执行 tty_do_write() 函数，然后直接向显存写入要显示的字符，还通过操作 VGA 的寄存器来设置光标。

首先改写 tty 任务的结构体，加入对 TTY 结构体的操作：

```
1   #define NR_CONSOLES 3
2
3   PUBLIC TTY tty_table[NR_CONSOLES];
4   PUBLIC CONSOLE console_table[NR_CONSOLES];
5
6   #define TTY_FIRST (tty_table)
7   #define TTY_END (tty_table + NR_CONSOLES)
8
9   PUBLIC void init_tty(TTY* p_tty);
10
11  PUBLIC void task_tty()
12  {
13      TTY* p_tty;
14      init_keyboard();
15      for(p_tty = TTY_FIRST; p_tty < TTY_END; ++p_tty)
16      {
```

```
            init_tty(p_tty);
        }

        nr_current_console = 0;
        while(1)
        {
            for(p_tty=TTY_FIRST; p_tty<TTY_END; ++p_tty)
            {
                tty_do_read(p_tty);
                tty_do_write(p_tty);
            }
        }
    }

    PRIVATE void init_tty(TTY* p_tty)
    {
        p_tty->inbuf_count = 0;
        p_tty->p_inbuf_head = p_tty->p_inbuf_tail = p_tty->in_buf;

        int nr_tty = p_tty - tty_table;
        p_tty->p_console = console_table + nr_tty;
    }
```

在调用 tty_do_read() 函数时，如果是当前 console，就可以读取键盘缓冲区，然后用 in_process() 函数将读取的键值 key 写入 tty 缓冲区：

```
    PUBLIC int is_current_console(CONSOLE* p_console);

    PRIVATE void tty_do_read(TTY* p_tty)
    {
        if(is_current_console(p_tty->p_console))
            keyboard_read(p_tty);
    }

    PUBLIC int is_current_console(CONSOLE* p_console)
    {
        return (p_console == &console_table[nr_current_console]);
    }

    PUBLIC void in_process(TTY* p_tty, u32 key)
    {
        char output[2];
        memset(output, 0, 2);

        if(!(key & FLAG_EXT))
        {
            if(p_tty->inbuf_count < TTY_IN_BYTES)
            {
                *(p_tty->p_inbuf_head) = key;
                p_tty->p_inbuf_head++;
                if(p_tty->p_inbuf_head == p_tty->in_buf + TTY_IN_BYTES)
                    p_tty->p_inbuf_head = p_tty->in_buf;
                p_tty->inbuf_count++;
            }
        }
```

```
30      }
```

tty 任务执行 tty_do_write() 函数，然后直接向显存写入要显示的字符：

```
1       PUBLIC void out_char(CONSOLE* p_console, char ch);
2
3       PRIVATE void tty_do_write(TTY* p_tty)
4       {
5           if(p_tty->inbuf_count > 0)
6           {
7               char ch = *(p_tty->p_inbuf_tail);
8
9               p_tty->p_inbuf_tail++;
10              if(p_tty->p_inbuf_tail == p_tty->in_buf + TTY_IN_BYTES)
11                  p_tty->p_inbuf_tail = p_tty->in_buf;
12
13              out_char(p_tty->p_console, ch);
14          }
15      }
16
17      PUBLIC void out_char(CONSOLE* p_console, char ch)
18      {
19          u8* p_vmem = (u8*)(V_MEM_BASE + disp_pos);
20
21          *p_vmem++ = ch;
22          *p_vmem++ = DEFAULT_CHAR_COLOR;
23          disp_pos += 2;
24
25          set_cursor(disp_pos/2);
26      }
```

这里的 set_cursor() 函数是通过向寄存器写入值来进行光标的设置：

```
1       PRIVATE void set_cursor(unsigned int position)
2       {
3           disable_int();
4           out_byte(CRTC_ADDR_REG, CURSOR_H);
5           out_byte(CRTC_DATA_REG, (position >> 8) & 0xFF);
6           out_byte(CRTC_ADDR_REG, CURSOR_L);
7           out_byte(CRTC_DATA_REG, position & 0xFF);
8       }
```

2.3 多控制台

为了显示多个控制台，我们应该构建如下框架：

1. 初始化每个 tty 结构中的 console 成员，设置它们初始对应的显存。
2. 将 outchar() 与 console 结构体挂钩，根据结构体中的 cursor 成员进行字符的显示。
3. 在 in_process() 函数中，添加对 Alt+Fn 的处理。当按下 Alt+Fn 时，进行控制台的切换。

初始化每个 tty 结构中的 console 成员，设置它们初始对应的显存：

```
PUBLIC void init_screen(TTY* p_tty)
{
    int nr_tty = p_tty - tty_table;
    p_tty->p_console = console_table + nr_tty;
    int v_mem_size = V_MEM_BASE >> 1;
    int con_v_mem_size = v_mem_size / NR_CONSOLES;
    p_tty->p_console->original_addr = nr_tty * con_v_mem_size;
    p_tty->p_console->v_mem_limit = con_v_mem_size;
    p_tty->p_console->current_start_addr = p_tty->console->original_addr;
    p_tty->p_console->cursor = p_tty->p_console->original_addr;

    if(nr_tty == 0)
    {
        p_tty->p_console->cursor = disp_pos / 2;
        disp_pos = 0;
    }
    else
    {
        out_char(p_tty->p_console, nr_tty + '0');
        out_char(p_tty->p_console, '#');
    }

    set_cursor(p_tty->p_console->cursor);
}
```

将 outchar() 与 console 结构体挂钩，根据结构体中的 cursor 成员进行字符的显示：

```
PUBLIC void out_char(CONSOLE* p_console, char ch)
{
    u8* p_vmem = (u8*)(V_MEM_BASE + p_console->cursor * 2);
    *p_vmem++ = ch;
    *p_vmem++ = DEFAULT_CHAR_COLOR;
    p_console->cursor++;
    set_cursor(p_console->cursor);
}
```

在 in_process() 函数中，添加对 Alt+Fn 的处理。当按下 Alt+Fn 时，进行控制台的切换：

```
PUBLIC void select_console(int nr_console)
{
    if((nr_console < 0) || (nr_console >= NR_CONSOLES))
        return;

    nr_current_console = nr_console;
    set_cursor(console_table[nr_console].cursor);
    set_vedio_start_addr(console_table[nr_console].current_start_addr);
}

PUBLIC void in_process(TTY* p_tty, u32 key)
{
    ...
```

```
        else
        {
            case F1:
            case F2:
            case F3:
            case F4:
            case F5:
            case F6:
            case F7:
            case F8:
            case F9:
            case F10:
            case F11:
            case F12:
                if((key & FLAG_ALT_L) || (key & FLAG_ALT_R))
                    select_console(raw_code - F1);
                break;
        }
    }
```

3 实现 printf 函数

这里的 printf 函数就是我们熟悉的 printf 函数，当某个进程调用 printf() 函数时，操作系统就会向控制台输出字符。

实现 printf 函数的步骤如下：

1. 向进程表添加成员，为每个进程指定特定的 TTY。
2. 实现系统调用 write，这里的 write() 函数借助我们之前实现的 TTY 结构和 outchar() 进行实现。
3. 借助系统调用 write() 实现 printf 函数。

向进程表添加成员，为每个进程指定特定的 TTY：

```
    typedef struct s_proc
    {
        STACK_FRAME regs;
        u16 ldt_sel;

        int ticks;
        int priority;

        u32 pid;
        char p_name[16];

        int nr_tty;
    }
```

实现系统调用 write()：

```
// 添加关于write()函数的声明
PUBLIC void sys_write(char* buf, int len, PROCESS* p_proc);
PUBLIC void write(char* buf, int len);

// 编写write()的函数体
write:
    mov eax, _NR_write
    mov ebx, [esp + 4]
    mov ecx, [esp + 8]
    int INT_VECTOR_SYS_CALL
    ret

// 编写sys_write()的函数体
PUBLIC void tty_write(TTY* p_tty, char* buf, int len)
{
    char* p = buf;
    int i = len;

    while(i)
    {
        out_char(p_tty->p_console, *p++);
        i--;
    }
}

PUBLIC int sys_write(char* buf, int len, PROCESS* p_proc)
{
    tty_write(&tty_table[p_proc->nr_tty], buf, len);
    return 0;
}
```

借助系统调用 write() 实现 printf 函数：

```
int printf(const char* fmt, ...)
{
    int i;
    char buf[256];

    va_list arg = (va_list)((char*)(&fmt) + 4);
    i = vsprintf(buf, fmt, arg);
    write(buf, i);
    return i;
}

int vsprintf(char* buf, const char* fmt, va_list args)
{
    char* p;
    char tmp[256];
    va_list p_next_arg = args;

    for(p=buf; *fmt; fmt++)
    {
        if(*fmt != '%')
```

```
21              {
22                  *p++ = *fmt;
23                  continue;
24              }
25
26              fmt++;
27
28              switch(*fmt)
29              {
30                  case 'x':
31                      itoa(tmp, *((int*)p_next_arg));
32                      strcpy(p, tmp);
33                      p_next_arg += 4;
34                      p += strlen(tmp);
35                      break;
36                  case 's':
37                      while(((char*)p_next_arg) != '\0')
38                      {
39                          *p = (char*)p_next_arg;
40                          p++;
41                          p_next_arg++;
42                      }
43                      break;
44                  default:
45                      break;
46              }
47          }
48      }
```