实验报告：设备驱动程序

—— 黄浩睿 **2018202143**

在 Xinu 中实现键盘与显示器的驱动程序。

设备定义

在 Configuration 文件中添加两个设备类型：vga 和 kbd：

```
67  /* type of screen device */
68  vga:
69      on top
70          -i vgainit  -o ionull   -c ionull
71          -r ioerr    -g ioerr    -p vgaputc
72          -w vgawrite -s ioerr    -n ioerr
73          -intr ionull
74
75  /* type of keyboard device */
76  kbd:
77      on top
78          -i kbdinit  -o ionull   -c ionull
79          -r kbdread  -g kbdgetc  -p ioerr
80          -w ioerr    -s ioerr    -n ioerr
81          -intr kbddisp
```

其中 vga 支持初始化和输出，kbd 支持初始化和输入。为键盘设备 kbd 定义的中断处理函数为 kbddisp。

根据两个设备类型，定义两个设备：SCREEN 和 KEYBOARD：

```
112     /* Define a screen device */
113
114     SCREEN is vga on top
115
116     /* Define a keyboard device */
117
118     KEYBOARD is kbd on top -irq 0041
```

此处定义了键盘的中断编号为八进制的 0041，也就是十进制的 33。

显示器

对于显示器设备，需要维护当前光标位置。由于只有一个显示器，这里直接使用了全局变量来存储，没有使用 device 数据结构。

通过以下函数设置当前光标位置。该函数不仅会修改保存的光标位置，还会通过 I/O 端口更新显示器上显示的光标位置。

```
void setCursorPosition(uint8 row, uint8 column) {
    uint16 index = getScreenPositionIndex(row, column);

    outb(0x3D4, 0x0F);
    outb(0x3D5, index & 0xFF);
    outb(0x3D4, 0x0E);
    outb(0x3D5, (index >> 8) & 0xFF);

    curr.row = row;
    curr.column = column;
}
```

当输出字符时，判断如果要输出的是特殊字符（换行、制表符、退格）则进行特殊操作，否则写入字符并移动光标到下一个位置：

```
void outputAsciiCharacter(char ascii, uint8 mode) {
    switch (ascii) {
    case '\n':
        // Newline
        moveToNewLine();
        break;
    case '\t':
        // Tab
        for (int32 i = 0; i < 8 - (curr.column % 8); i++)
            outputAsciiCharacter(' ', mode);
        break;
    case '\b':
        // Backspace
        backspace();
        break;
    default:
        writeCharacterToScreenMemory(curr.row, curr.column, ascii, mode);
        if (curr.column == SCREEN_WIDTH - 1)
            moveToNewLine();
        else
            setCursorPosition(curr.row, curr.column + 1);
    }
}
```

写入字符的方式是直接写入文本模式缓冲区：

```
const uint8 SCREEN_WIDTH = 80;
const uint8 SCREEN_HEIGHT = 25;
uint16 *const TEXT_MODE_BUFFER = (uint16 *)0xB8000;
const uint8 BLACK_WHITE = 0x07;

uint16 getScreenPositionIndex(uint8 row, uint8 column) {
    return (uint16)row * SCREEN_WIDTH + column;
}

uint16 combineCharacterWithMode(char ascii, uint8 mode) {
    return ((uint16)mode << 8) | ascii;
}

void writeCharacterToScreenMemory(uint8 row, uint8 column, char ascii, uint8 mode) {
    uint16 index = getScreenPositionIndex(row, column);
    TEXT_MODE_BUFFER[index] = combineCharacterWithMode(ascii, mode);
}
```

移动到下一行（因为换行符或行满）时，判断是否需要滚屏：

```
78   void moveToNewLine() {
79       if (curr.row == SCREEN_HEIGHT - 1) {
80           scrollUpScreen();
81           setCursorPosition(SCREEN_HEIGHT - 1, 0);
82       } else {
83           setCursorPosition(curr.row + 1, 0);
84       }
85   }
```

滚屏的实现方式为，将从第 1 行开始，直到最后一行的数据，写到从第 0 行开始的缓冲区中，并清空原最后一行的缓冲区数据：

```
62   void scrollUpScreen() {
63       // Move everything in the buffer except the last line
64       memmove(
65           TEXT_MODE_BUFFER,                // Buffer address from 0-th line
66           TEXT_MODE_BUFFER + SCREEN_WIDTH, // Buffer address from 1-st line
67           (int32)SCREEN_WIDTH * (SCREEN_HEIGHT - 1) * sizeof(uint16)
68       );
69
70       // Fill the last line
71       memset16(
72           TEXT_MODE_BUFFER + (int32)SCREEN_WIDTH * (SCREEN_HEIGHT - 1),
73           SCREEN_WIDTH,
74           combineCharacterWithMode(' ', BLACK_WHITE)
75       );
76   }
```

最后实现 Xinu 的驱动接口：

```
125  devcall vgainit(struct dentry *devptr) {
126      clearScreen();
127      setCursorPosition(0, 0);
128      return OK;
129  }
130
131  devcall vgaputc(struct dentry *devptr, char ch) {
132      kprintf("vgaputc: char = [%d] %c\n", (int)ch, ch);
133      outputAsciiCharacter(ch, BLACK_WHITE);
134      return OK;
135  }
136
137  devcall vgawrite(struct dentry *devptr, char *buff, int32 count) {
138      for (int32 i = 0; i < count; i++)
139          outputAsciiCharacter(buff[i], BLACK_WHITE);
140
141      return OK;
142  }
```

键盘

键盘输入的实现采用了生产者/消费者的模式，用一个环形缓冲区存储到来的字符，用一个信号量来维护剩余的字符数量 —— 如果键盘中断得到了字符，但缓冲区已满，该字符会被丢弃；如果应用程序请求字符，但缓冲区为空，则应用程序会通过等待的方式阻塞，直到有字符到来。

为了方便测试，这里采用了较小的缓冲区。在真实的操作系统中，该缓冲区可以足够大，以避免 CPU 负载较高时丢失按键。

初始化：创建信号量并设置中断向量表中键盘中断的处理函数。

```
3   #define KBD_BUFFER_SIZE 4
4
5   sid32 kbdSem; // kbdSem = buffer length - 1
6   char kbdBuffer[KBD_BUFFER_SIZE];
7   int32 kbdBufferHead;
8
9   devcall kbdinit(struct dentry *devptr) {
10      kbdSem = semcreate(0);
11      if (kbdSem == SYSERR)
12          return SYSERR;
13
14      kbdBufferHead = 0;
15
16      set_evec(devptr->dvirq, (uint32)devptr->dvintr);
17
18      return OK;
19  }
```

键盘中断处理函数会调用该函数，该函数首先禁用调度（因为处于中断处理中），然后处理尽量多的新按键，（通过 readKeyboardBuffer 函数获取按键对应的字符，该函数为实验指导中所提供的 kbdgetc）根据缓冲区是否已满决定丢弃或加入缓冲区。

```
21  void kbdhandler(void) {
22      reschedule_cntl(DEFER_START);
23
24      while (1) {
25          int32 bufferLength = semcount(kbdSem);
26          if (bufferLength < 0) bufferLength = 0;
27
28          // Get key
29          char newChar = readKeyboardBuffer();
30          if (newChar == -1 || newChar == 0) break;
31
32          if (bufferLength >= KBD_BUFFER_SIZE) {
33              // Buffer full
34              kprintf("kbdhandler: buffer full, buffer head = %d, buffer length = %d, char = [%d] '%c'\n", kbdBufferHead, bufferLength, (int)newChar, newChar);
35              break;
36          }
37
38          // Append to buffer
39          kprintf("kbdhandler: got key, buffer head = %d, buffer length = %d, char = [%d] '%c'\n", kbdBufferHead, bufferLength, (int)newChar, newChar);
40
41          int32 pos = (kbdBufferHead + bufferLength) % KBD_BUFFER_SIZE;
42          kbdBuffer[pos] = newChar;
43          signal(kbdSem);
44      }
45
46      reschedule_cntl(DEFER_STOP);
47  }
```

在 kbdgetc 函数中，只需要等待该信号量，并从队列中获取新的字符即可。

```
49  devcall kbdgetc(struct dentry *devptr) {
50      wait(kbdSem);
51      int i = kbdBufferHead++;
52      char gotChar = kbdBuffer[i];
53      kbdBufferHead %= KBD_BUFFER_SIZE;
54      kprintf("kbdgetc: got char, buffer head = %d, char = [%d] '%c'\n", i, (int)gotChar, gotChar);
55      return gotChar;
56  }
57
58  devcall kbdread(struct dentry *devptr, char *buff, int32 count) {
59      char ch;
60      int32 i = 0;
61      while (i < count && (ch = kbdgetc(devptr)) != '\n')
62          buff[i++] = ch;
63      return i;
64  }
65
```

测试

这里新增了一个 Shell 命令：kbdtest。执行该命令会进入键盘/显示器驱动程序测试，该程序非常简单，即将从键盘上读取到的内容输出到显示器上：

```
5   shellcmd xsh_kbdtest(int nargs, char *args[]) {
6       while (1) {
7           char ch = fgetc(KEYBOARD);
8           fputc(ch, SCREEN);
9       }
10      return 0;
11  }
```

测试发现，键盘驱动能正确地将即时输入的字符返回给应用程序，也能将缓冲区中（在应用程序启动前之前）输入的字符返回给应用程序。显示器驱动程序能正确处理换行、折行、退格等操作。