Day 25

增加蜂鸣器发声API

蜂鸣器是BIOS提供的一个功能，是用PIT来进行控制的

蜂鸣器的文档如书中所示

- ❑ 蜂鸣器发声的控制
  - ■ 音高操作
    - ◆ AL = 0xb6; OUT(0x43, AL);
    - ◆ AL = 设定值的低位 8bit; OUT(0x42, AL);
    - ◆ AL = 设定值的高位 8bit; OUT(0x42, AL);
    - ◆ 设定值为 0 时当作 65536 来处理。
    - ◆ 发声的音高为时钟除以设定值，也就是说设定值为 1000 时相当于发出 1.19318KHz 的声音；设定值为 10000 时相当于 119.318Hz。因此设定 2712 即可发出约 440Hz 的声音[①]。
- ❑ 蜂鸣器ON/OFF
  - ◆ 使用 I/O 端口 0x61 控制。
  - ◆ ON：IN(AL, 0x61); AL |= 0x03; AL &= 0x0f; OUT(0x61, AL);
  - ◆ OFF：IN(AL, 0x61); AL &= 0xd; OUT(0x61, AL);

设计API，分配功能号20，一个参数作为声音频率（mHz）（0表示停止）

蜂鸣器发声

EDX	20
EAX	声音频率（单位是mHz，即毫赫兹）

hrb_api部分

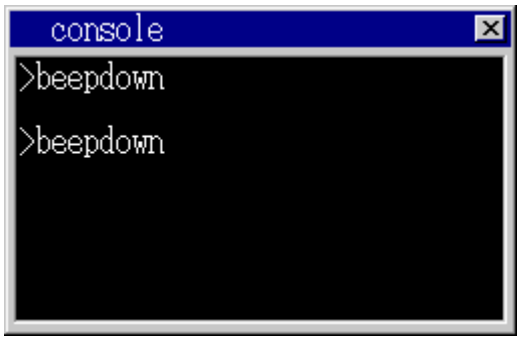
```
else if (edx == 20) {
    if (eax == 0) {
        i = io_in8(0x61);
        io_out8(0x61, i & 0x0d);
    } else {
        i = 1193180000 / eax;
        io_out8(0x43, 0xb6);
        io_out8(0x42, i & 0xff);
        io_out8(0x42, i >> 8);
        i = io_in8(0x61);
        io_out8(0x61, (i | 0x03) & 0x0f);
    }
}
```

api定义部分

```
_api_beep: ; void api_beep(int tone);
        MOV EDX,20
        MOV EAX,[ESP+4] ; tone
        INT 0x40
        RET
```

测试程序

```
void HariMain(void)
{
    int i, timer;
    timer = api_alloctimer();
    api_inittimer(timer, 128);
    for (i = 20000000; i >= 20000; i -= i / 100) {
        /* 20KHz～20Hz，即人类可以听到的声音范围*/
        /* i以1%的速度递减*/
        api_beep(i);
        api_settimer(timer, 1); /* 0.01秒*/
        if (api_getkey(1) != 128) {
            break;
        }
    }
    api_beep(0);
    api_end();
}
```



很遗憾无论是再QEMU还是再VM Virtual Box上，都没有蜂鸣器的模拟，不能在真机上安装，我无法观测出这个现象

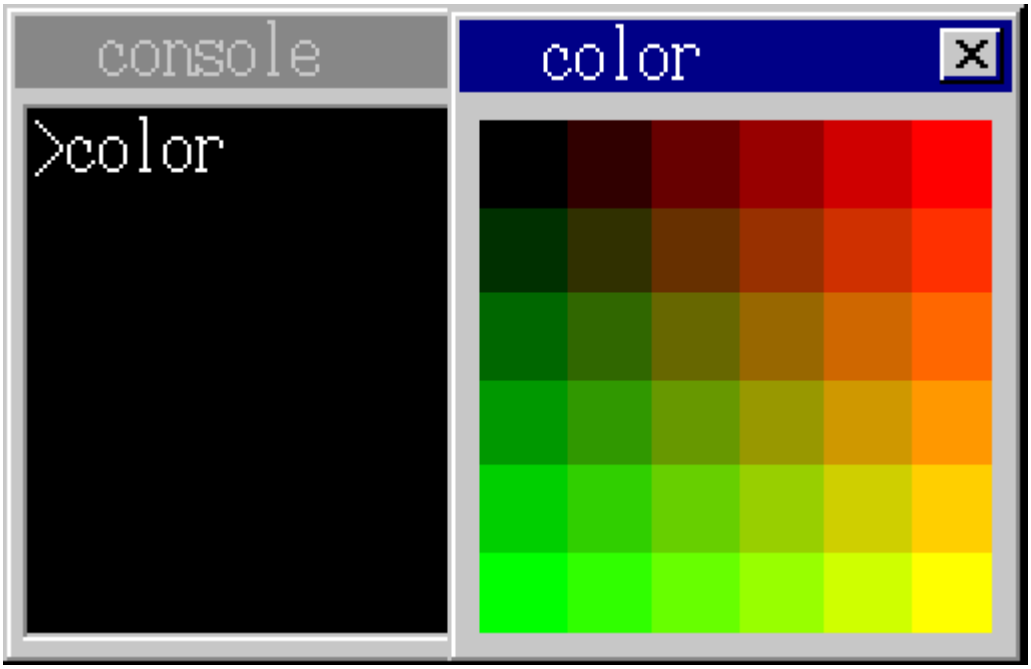
增加更多的颜色。进入到了256色模式，我们理应由更多的颜色可以使用。重新设置色板。为RGB自发光三原色各分配6个色阶，我们总共可以定义出$6^3 = 216$种颜色

```
unsigned char table2[216 * 3];
int r, g, b;
set_palette(0, 15, table_rgb);
for (b = 0; b < 6; b++) {
    for (g = 0; g < 6; g++) {
        for (r = 0; r < 6; r++) {
            table2[(r + g * 6 + b * 36) * 3 + 0] = r * 51;
            table2[(r + g * 6 + b * 36) * 3 + 1] = g * 51;
            table2[(r + g * 6 + b * 36) * 3 + 2] = b * 51;
        }
    }
}
set_palette(16, 231, table2);
```

我们试验一下我们的新色板好不好用。

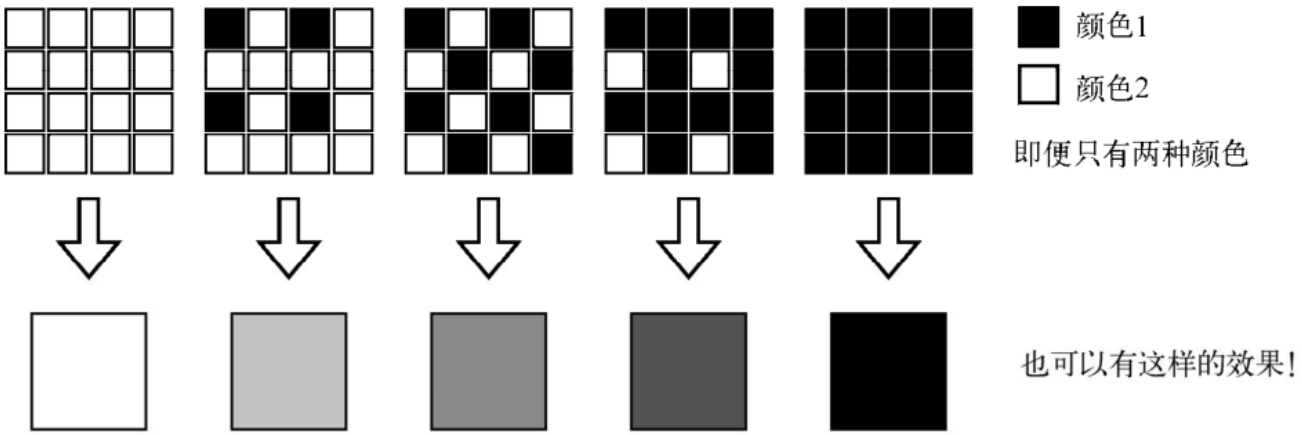
编写应用程序color.c

```
int api_openwin(char *buf, int xsiz, int ysiz, int col_inv, char *title);
void api_initmalloc(void);
char *api_malloc(int size);
void api_refreshwin(int win, int x0, int y0, int x1, int y1);
void api_linewin(int win, int x0, int y0, int x1, int y1, int col);
int api_getkey(int mode);
void api_end(void);
void HariMain(void)
{
    char *buf;
    int win, x, y, r, g, b;
    api_initmalloc();
    buf = api_malloc(144 * 164);
    win = api_openwin(buf, 144, 164, -1, "color");
    for (y = 0; y < 128; y++) {
        for (x = 0; x < 128; x++) {
            r = x * 2;
            g = y * 2;
            b = 0;
            buf[(x + 8) + (y + 28) * 144] = 16 + (r / 43) + (g / 43) * 6 + (b / 43) * 36;
        }
    }
    api_refreshwin(win, 8, 28, 136, 156);
    api_getkey(1);
    api_end();
}
```



更多的颜色!

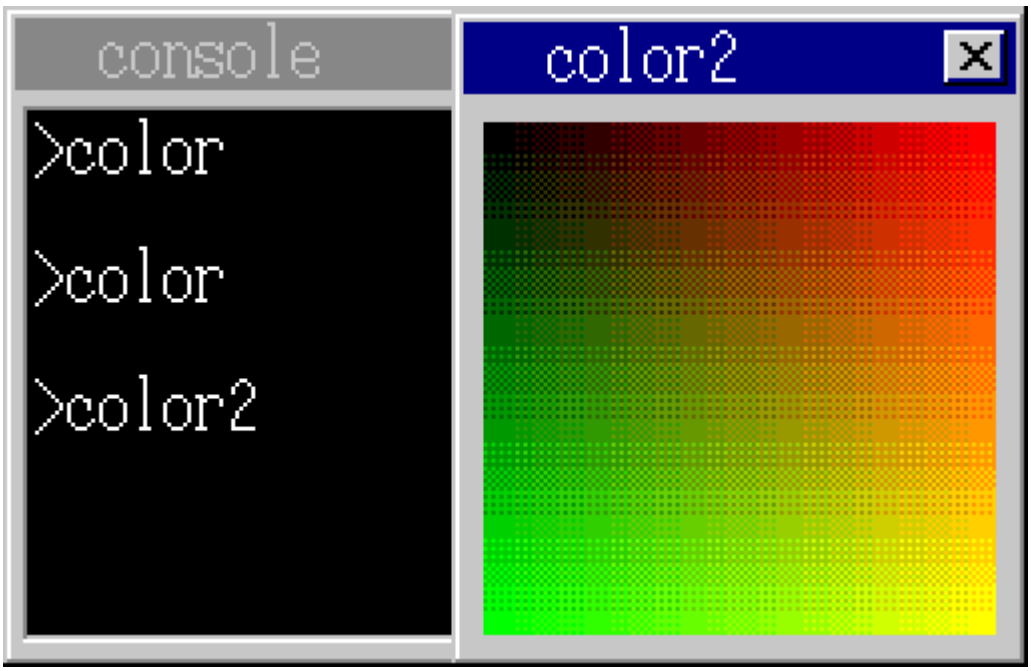
在无法使用全彩色的条件下，作者提供了（伪）实现更多色阶的一种方法。将像素点划分为4x4的单位，两种颜色按照以下方式进行混合排列，就可以产生额外的三种中间色



需要修改应用程序

```
/*调色部分*/
unsigned char rgb2pal(int r, int g, int b, int x, int y)
{
    static int table[4] = { 3, 1, 0, 2 };
    int i;
    x &= 1; /*判断是偶数还是奇数*/
    y &= 1;
    i = table[x + y * 2]; /*用来生成中间色的常量*/
    r = (r * 21) / 256; /* r为0～20*/
    g = (g * 21) / 256;
    b = (b * 21) / 256;
    r = (r + i) / 4; /* r为0～5*/
```

```
    g = (g + i) / 4;
    b = (b + i) / 4;
    return 16 + r + g * 6 + b * 36;
}
/*然后之前那一长串表达式改成这个*/
buf[(x + 8) + (y + 28) * 144] = rgb2pal(x * 2, y * 2, 0, x, y);
```



从远处看好像还不错

设置窗口初始位置

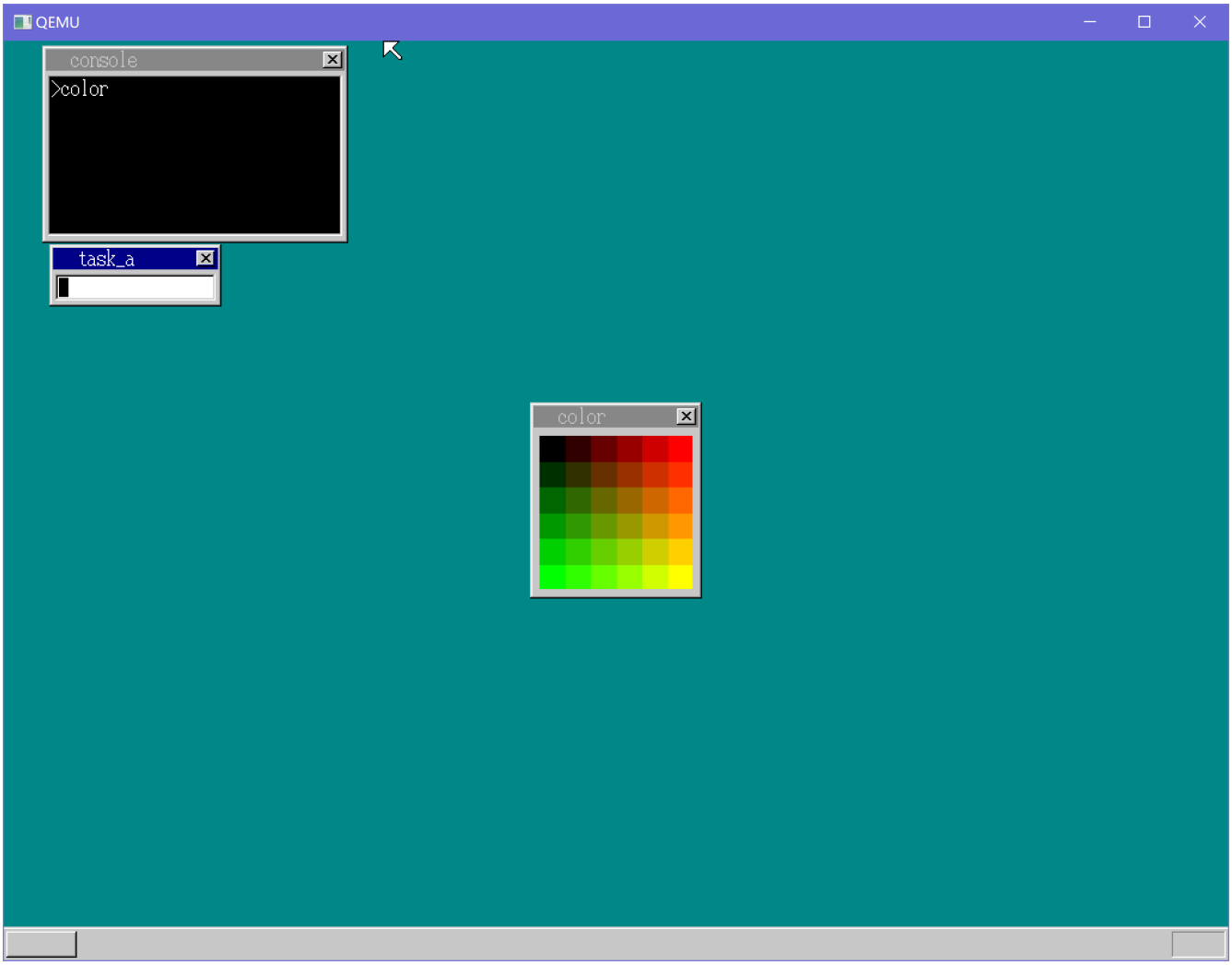
之前我们新建窗口的默认高度是3，当以后窗口多了之后他就不在最上方了，而且我们的窗口显示位置很随意。我们让窗体能居中，并且一定处于最上层。

代码改动非常小

在功能号5的api中断那里加两句

```
sheet_slide(sht, (shtctl->xsize - esi) / 2, (shtctl->ysize - edi) / 2);
sheet_updown(sht, shtctl->top);
```

就OK了



自动出现在中央

增加命令行窗口

目前位置我们只能同时运行一个应用程序，这是因为当一个命令行窗口启动一个应用程序的时就会进入阻塞状态。同时运行多个程序可以有多个解决方案

- 启动应用程序不阻塞命令行窗口（这个目前修改代码量较大）
- 启动多个命令行窗口

我们选择第二个方案。

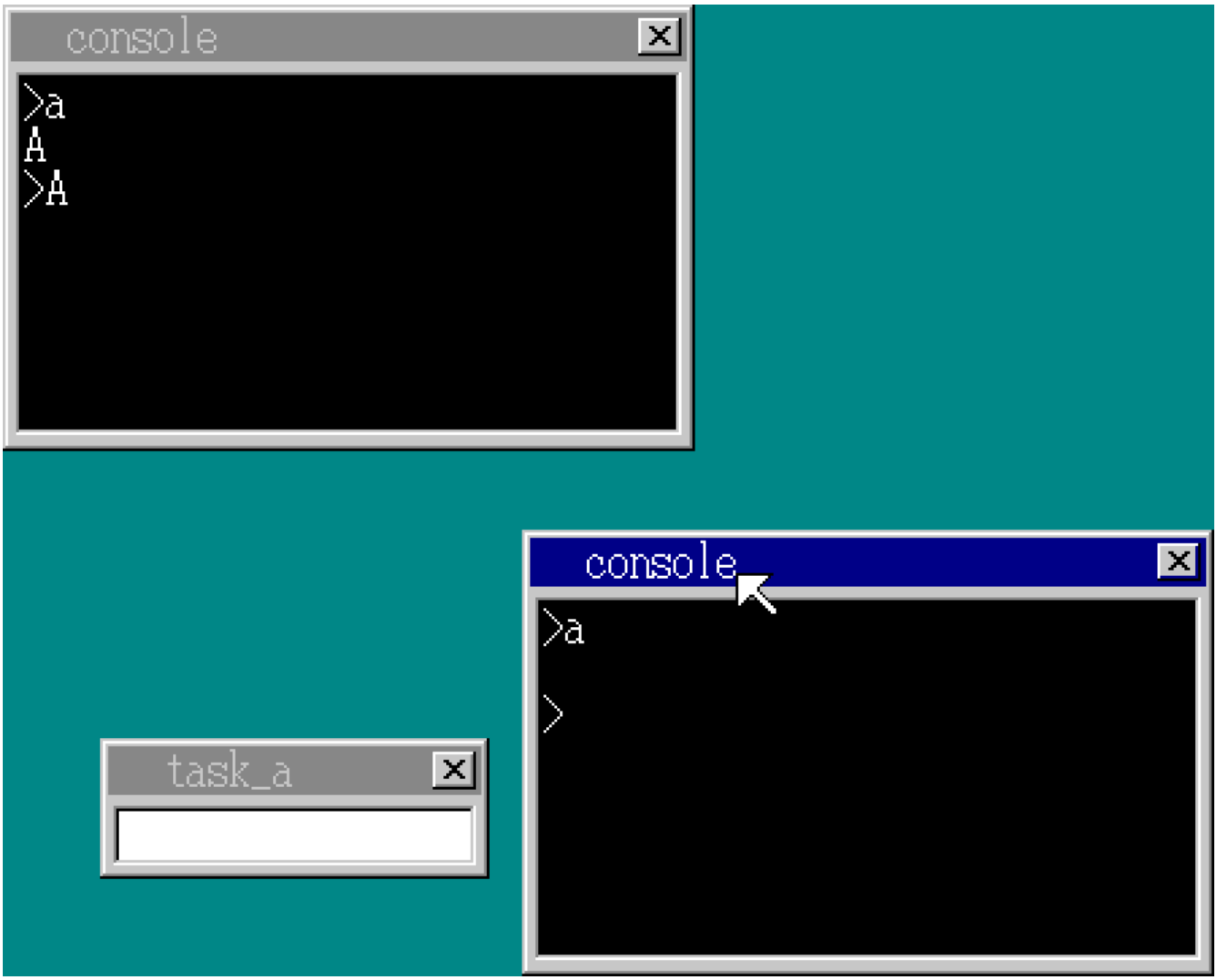
把命令行窗口相关的变量

- buf_cons
- sht_cons
- task_cons
- cons

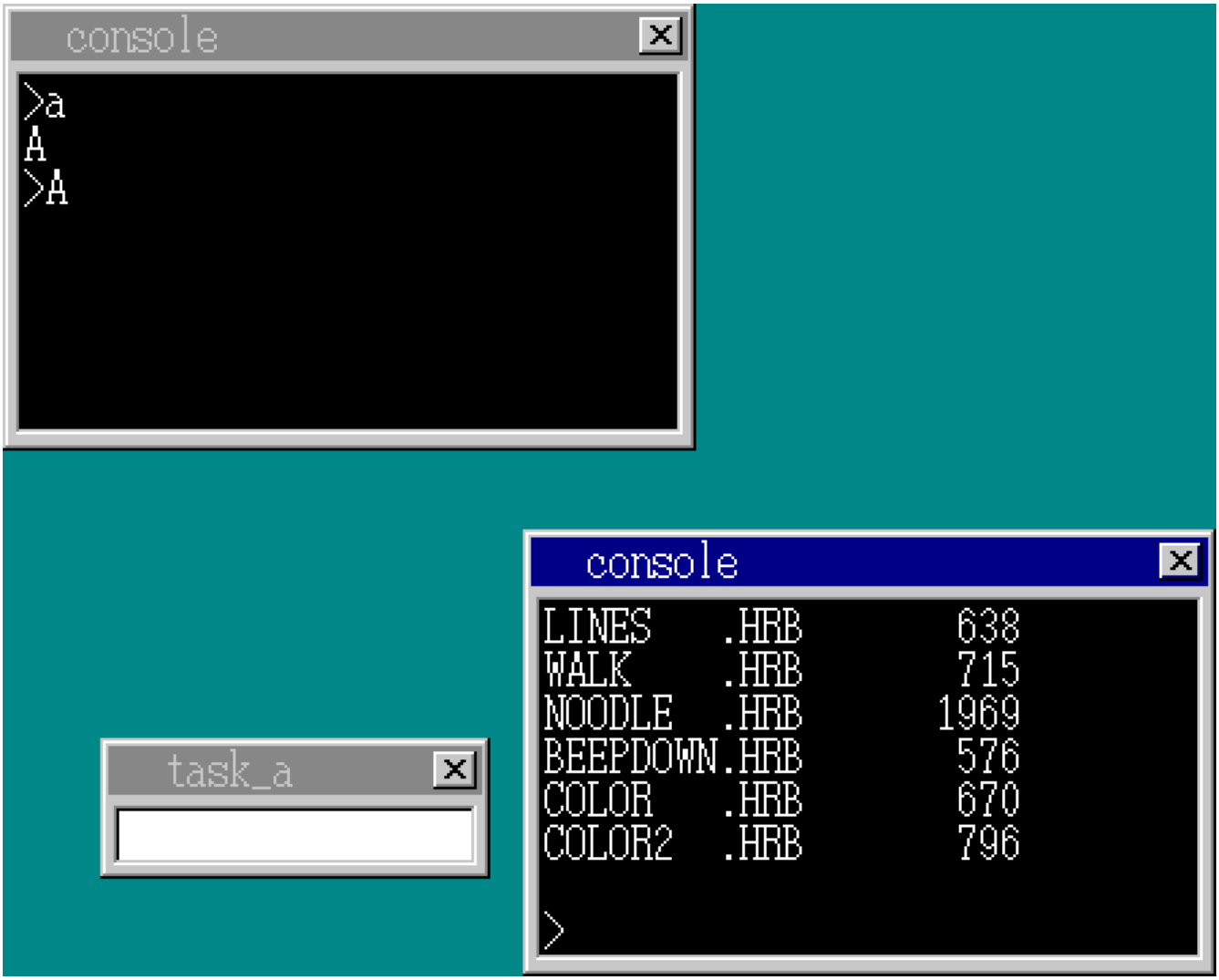
各准备2个，使用数组

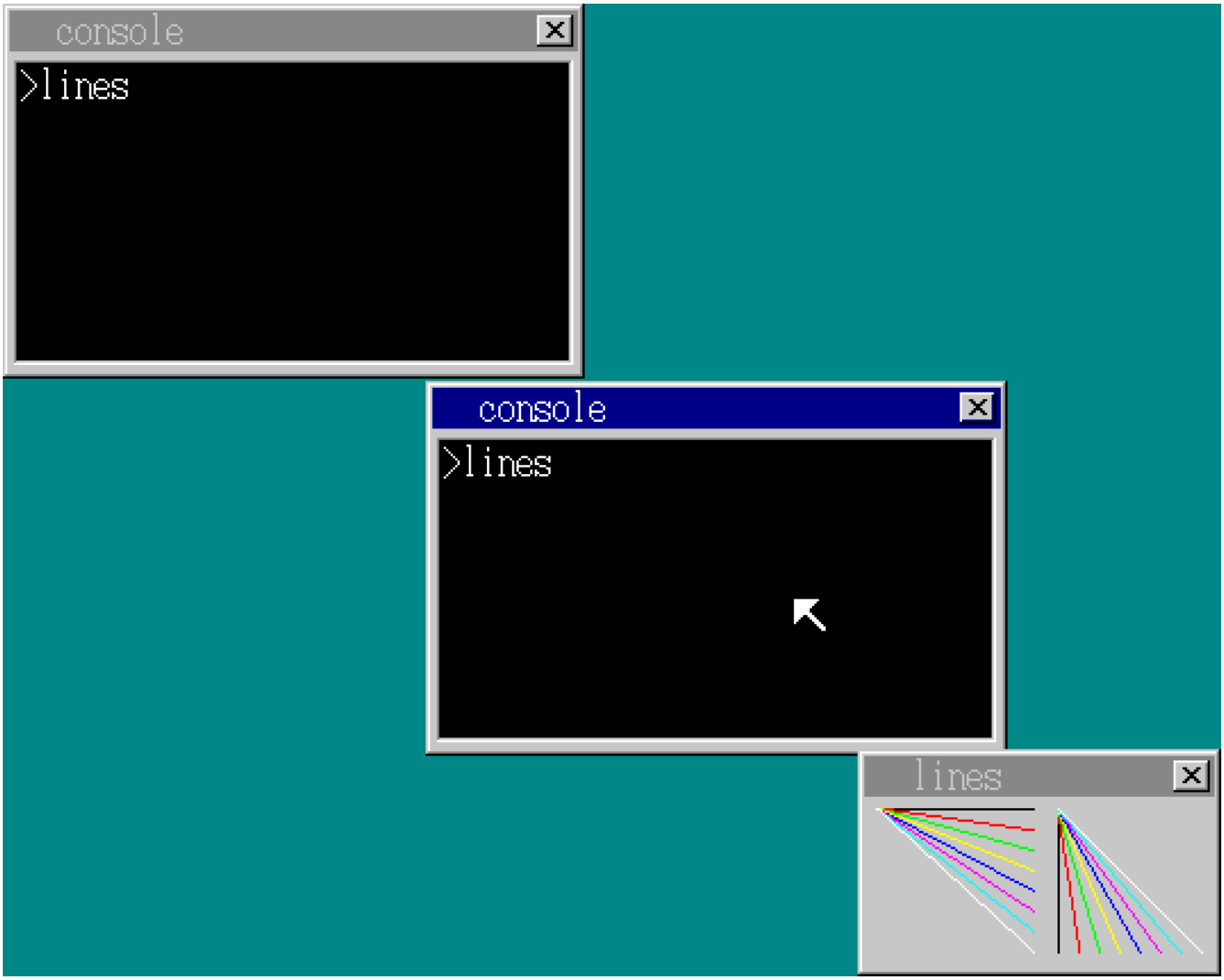
```
unsigned char *buf_back, buf_mouse[256], *buf_win, *buf_cons[2]; /*从此开始*/
struct SHEET *sht_back, *sht_mouse, *sht_win, *sht_cons[2];
struct TASK *task_a, *task_cons[2];
```

然后我们把原来的变量操作用循环抱起来，并修改为数组。



我们发现两个命令行窗口运行应用程序的输出都只在一个窗口当中（内嵌命令是正常的）


console
>a
A
>A
task_a
console
LINES .HRB 638
WALK .HRB 715
NOODLE .HRB 1969
BEEPDOWN.HRB 576
COLOR .HRB 670
COLOR2 .HRB 796
>



运行两个图形程序，系统直接卡住了

字符输出的问题出在哪呢？我们把cons句柄放在0x0fec位置，hrb_api总是从这个地方读取。我们把它存在task结构当中（以及ds_base），这样就可以避免从内存中读取了

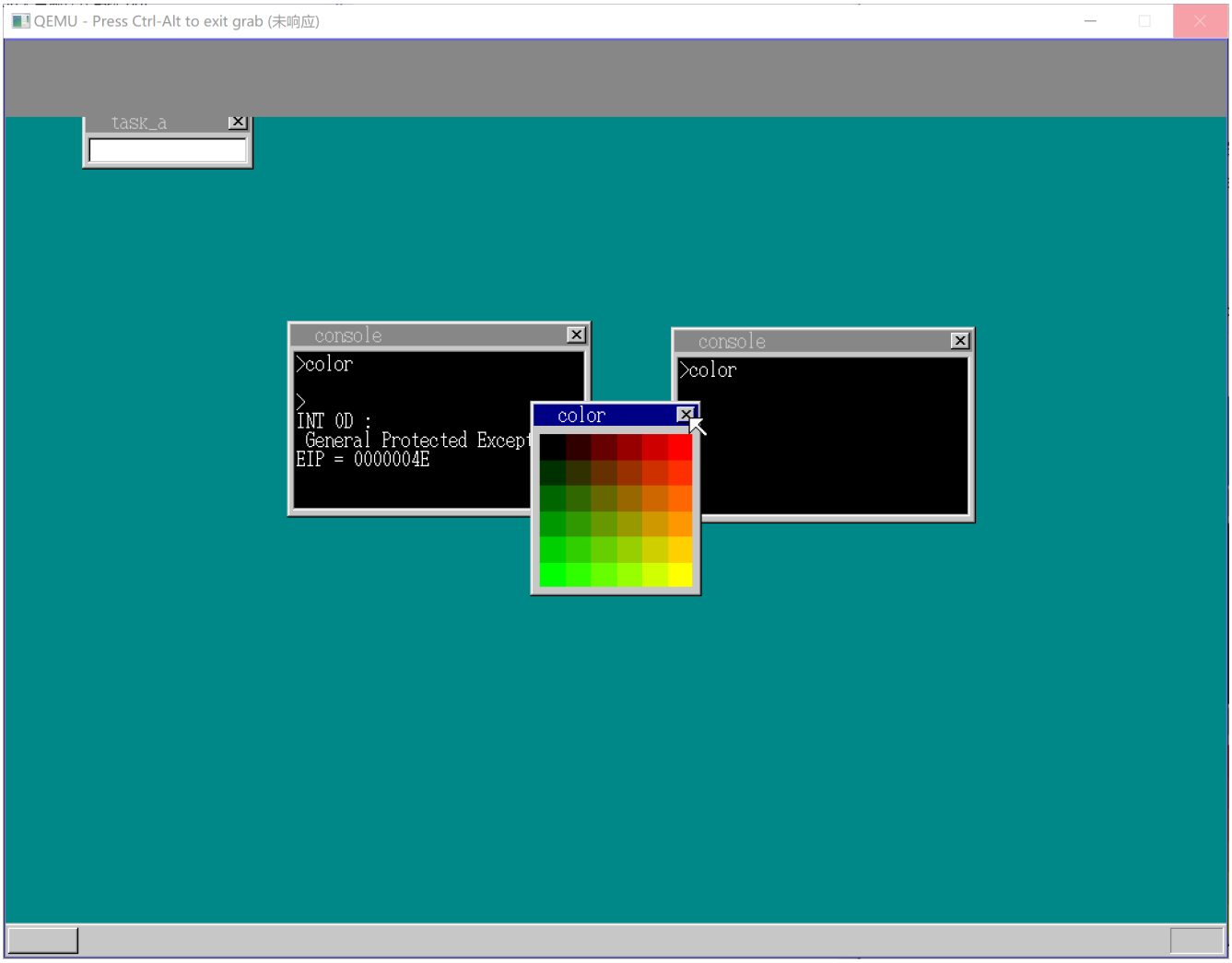
然后hrb_api中的就可以改成

```
int ds_base = task->ds_base;
struct CONSOLE *cons = task->cons;
```

测试一下

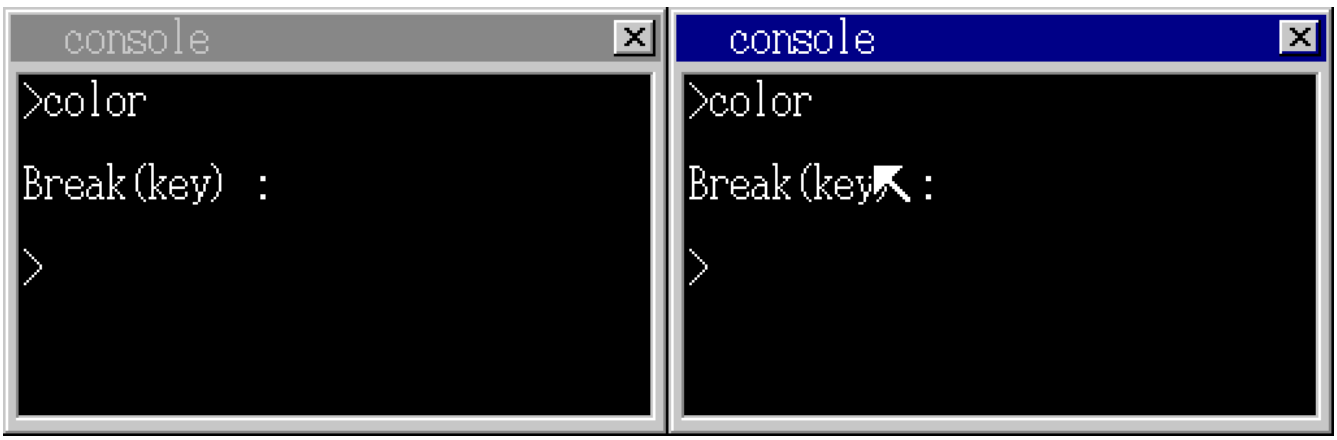
成功的运行了！

当我们点击其中的一个x的时候，咦？为什么另一个窗口却关闭了，再点击剩下的窗口中的x



糟糕

这是为什么呢？（h）是因为bootpack当中仍然是用0x0fec找cons的，给他也改掉就好了



这次都能正常退出了

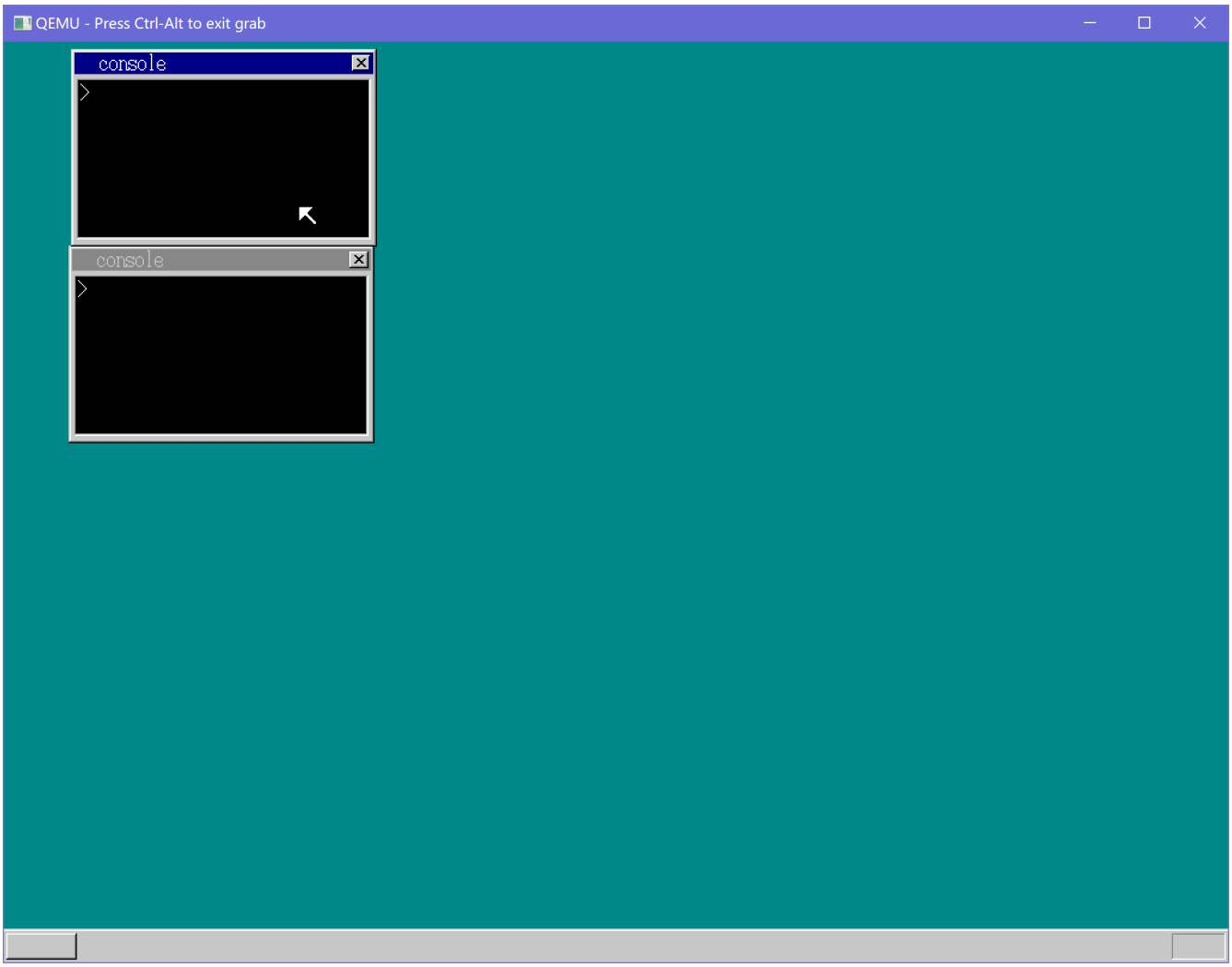
删除task_a窗口，这个窗口实在是没有什么用了。

task_a的逻辑在bootpack当中，先删除掉，然后将console_task的FIFO初始化放到harimain当中。

> 只有当bootpack.c的HariMain休眠之 后才会运行命令行窗口任务，而如果不运行这个任务的话，FIFO缓冲区就不会被初始化，这就相 当于我们在向一个还没初始化的FIFO强行发送数据，于是造成fifo32_put混乱而导致重启。

```
int fifobuf[128];
for (i = 0; i < 2; i++) {
    cons_fifo[i] = (int *) memman_alloc_4k(memman, 128 * 4);
    fifo32_init(&task_cons[i]->fifo, 128, cons_fifo[i], task_cons[i]);
}
```

然后彻底删除console_task当中的fifo初始化代码



OK了