

## Day 24

我们现在可以多窗口了，不过我们仍然无法对窗口进行精确的操作。我们甚至无法更改他们的顺序。我们按部就班，制定以下计划

- 窗口切换
- 移动窗口
- 鼠标点击关闭窗口
- 切换输入到窗口
- 鼠标切换输入窗口

设定F11为将窗口切换到最上层的快捷键

```
if (i == 256 + 0x57 && shtctl->top > 2) { /* F11 */  
    sheet_updown(shtctl->sheets[1], shtctl->top - 1); // 将最下面的窗口放到鼠标下面的那一层  
}
```



测试通过，工作如预期

为了能够通过鼠标点击来切换窗口，我们首先要屏蔽掉鼠标点击移动task\_a的代码。当鼠标点击了一个位置，我们需要从上到下进行判断，我们点击的是那个图层（注意要忽略透明色）

```
if (mouse_decode(&mdec, i - 512) != 0) {  
    if ((mdec.btn & 0x01) != 0) {  
        for (j = shtctl->top - 1; j > 0; j--) {
```

```

    sht = shtctl->sheets[j];
    x = mx - sht->vx0;
    y = my - sht->vy0;
    if (0 <= x && x < sht->bysize && 0 <= y && y < sht->bysize) {
        if (sht->buf[y * sht->bysize + x] != sht->col_inv) {
            sheet_updown(sht, shtctl->top - 1);
            break;
        }
    }
}
}
}
}
}

```

make run—发



鼠标点击成功的把console调到了顶层

实现移动窗口

如何实现我们通常所使用的那种窗口移动呢？将左键按下视作进入窗口移动状态，此时窗口跟随鼠标的移动。鼠标左键弹起时退出窗口移动模式，窗口不再跟随鼠标移动

```

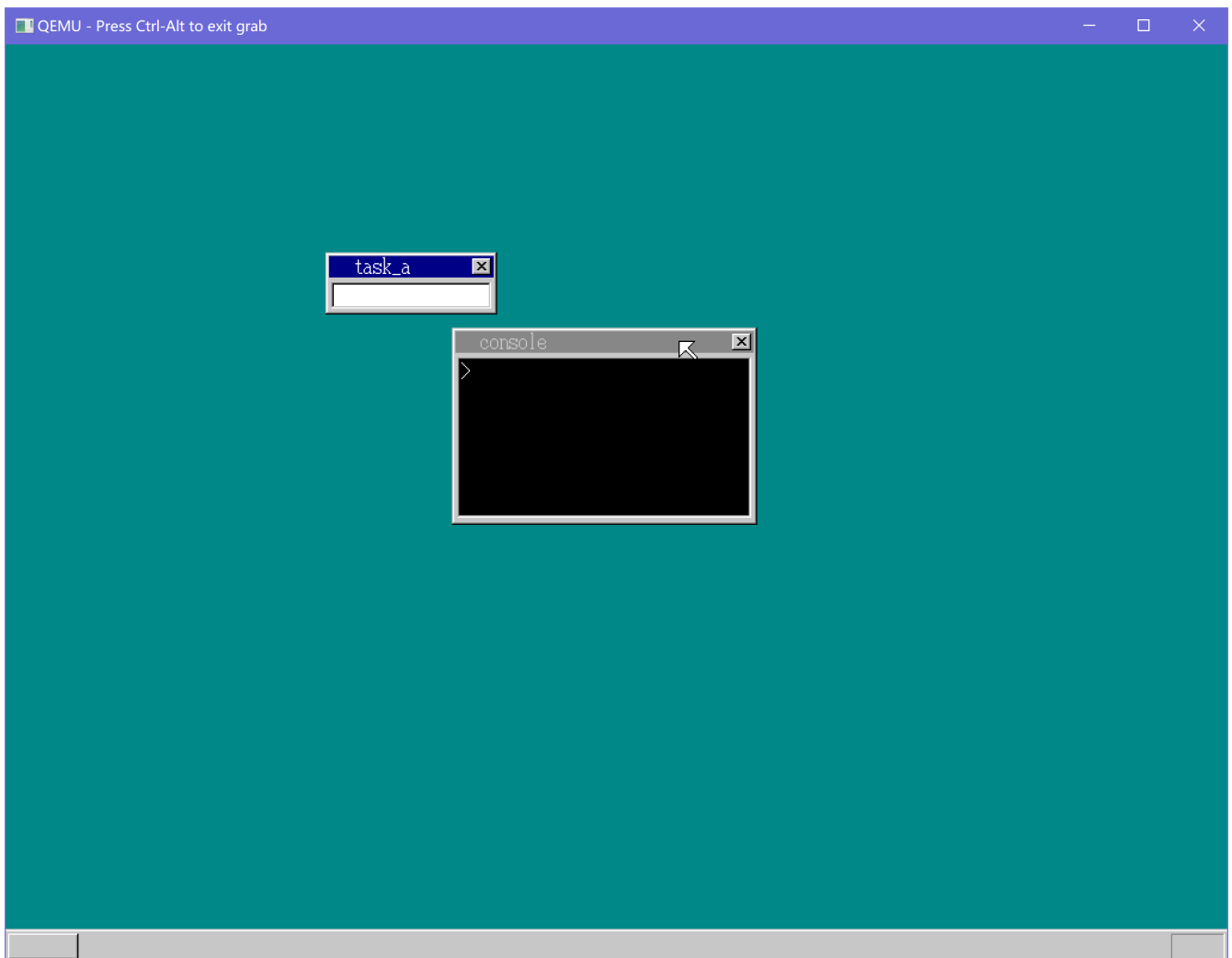
if (mouse_decode(&mdec, i - 512) != 0) {
    /*鼠标指针移动*/
    if ((mdec.btn & 0x01) != 0) {
        /*按下左键*/
        if (mmx < 0) {
            /* 处于非窗口移动模式 */
            for (j = shtctl->top - 1; j > 0; j--) {
                sht = shtctl->sheets[j];
                x = mx - sht->vx0;
                y = my - sht->vy0;
                if (0 <= x && x < sht->bysize && 0 <= y && y < sht->bysize) {

```

```

        if (sht->buf[y * sht->bysize + x] != sht->col_inv) {
            sheet_updown(sht, shtctl->top - 1);
            if (3 <= x && x < sht->bysize - 3 && 3 <= y && y < 21) {
                mmx = mx; /*进入窗口移动模式*/
                mmy = my;
            }
            break;
        }
    }
} else {
    x = mx - mmx; /*计算鼠标的移动距离*/
    y = my - mmy;
    sheet_slide(sht, sht->vx0 + x, sht->vy0 + y); /*移动窗体*/
    mmx = mx;
    mmy = my;
}
} else {
    mmx = -1;
}
}
}

```

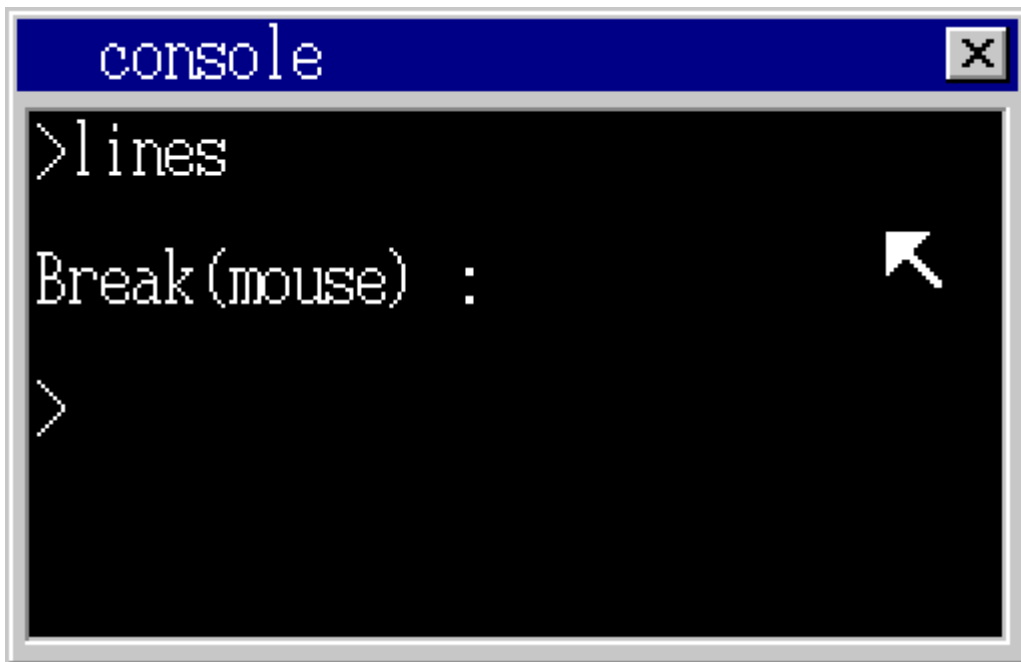


棒极了

## 鼠标关闭窗口

实现和鼠标切换窗口大同小异，判断点击位置是否在以及在那个窗体的X位置上，然后强制结束任务即可。

```
if (sht->bysize - 21 <= x && x < sht->bysize - 5 && 5 <=
    y && y < 19) {
    if (sht->task != 0) {
        cons = (struct CONSOLE *) *((int *) 0x0fec);
        cons_putstr0(cons, "\nBreak(mouse) :\n");
        io_cli(); /*强制结束处理中禁止切换任务*/
        task_cons->tss.eax = (int)
            &(task_cons->tss.esp0);
        task_cons->tss.eip = (int) asm_end_app;
        io_sti();
    }
}
```



尽管我们已经能够让应用程序接受键盘输入，但其实拥有焦点的时命令行窗口，而不是我们的应用程序，这个逻辑应该重新理一下了。

重新制定tab键切换窗口的逻辑：切换到下一层窗口，如果当前窗口已经在最底层了，那么切换到最上层。

之前使用的时key\_to变量，使用类似的方法，不过名字修改为key\_win即当前处于输入模式的窗口地址。

对了，如果处于输入模式的窗口被关闭了怎么办？可以让系统自动选择剩余窗口中最上层的窗口获得焦点。

```

int keywin_off(struct SHEET *key_win, struct SHEET *sht_win, int cur_c, int cur_x)
{
    change_wtitle8(key_win, 0);
    if (key_win == sht_win) {
        cur_c = -1; /*删除光标*/
        boxfill8(sht_win->buf, sht_win->bysize, COL8_FFFFFFFF, cur_x, 28, cur_x + 7, 43);
    } else {
        if ((key_win->flags & 0x20) != 0) {
            fifo32_put(&key_win->task->fifo, 3); /*命令行窗口光标OFF */
        }
    }
    return cur_c;
}

int keywin_on(struct SHEET *key_win, struct SHEET *sht_win, int cur_c)
{
    change_wtitle8(key_win, 1);
    if (key_win == sht_win) {
        cur_c = COL8_000000; /*显示光标*/
    } else {
        if ((key_win->flags & 0x20) != 0) {
            fifo32_put(&key_win->task->fifo, 2); /*命令行窗口光标ON */
        }
    }
    return cur_c;
}

```

以下函数用于修改窗体标题，不同于make\_wtitle8的地方在于，我们不知道窗口的名称也可以修改标题栏的颜色。

```

void change_wtitle8(struct SHEET *sht, char act)
{
    int x, y, xsize = sht->bysize;
    char c, tc_new, tbc_new, tc_old, tbc_old, *buf = sht->buf;
    if (act != 0) {
        tc_new = COL8_FFFFFFFF;
        tbc_new = COL8_000084;
        tc_old = COL8_C6C6C6;
        tbc_old = COL8_848484;
    } else {
        tc_new = COL8_C6C6C6;
        tbc_new = COL8_848484;
        tc_old = COL8_FFFFFFFF;
        tbc_old = COL8_000084;
    }
    for (y = 3; y <= 20; y++) {
        for (x = 3; x <= xsize - 4; x++) {
            c = buf[y * xsize + x];
            if (c == tc_old && x <= xsize - 22) {
                c = tc_new;
            } else if (c == tbc_old) {
                c = tbc_new;
            }
        }
    }
}

```

```

        buf[y * xsize + x] = c;
    }
}
sheet_refresh(sht, 3, 3, xsize, 21);
return;
}

```

cmd\_app也需要修改。由于没有运行应用程序的命令行窗口，所以它的task也不为0，需要通过flags&0x11来进行判断是否自动关闭。

```

for (i = 0; i < MAX_SHEETS; i++) {
    sht = &(shtctl->sheets0[i]);
    if ((sht->flags & 0x11) == 0x11 && sht->task == task) {
        sheet_free(sht);
    }
}

```

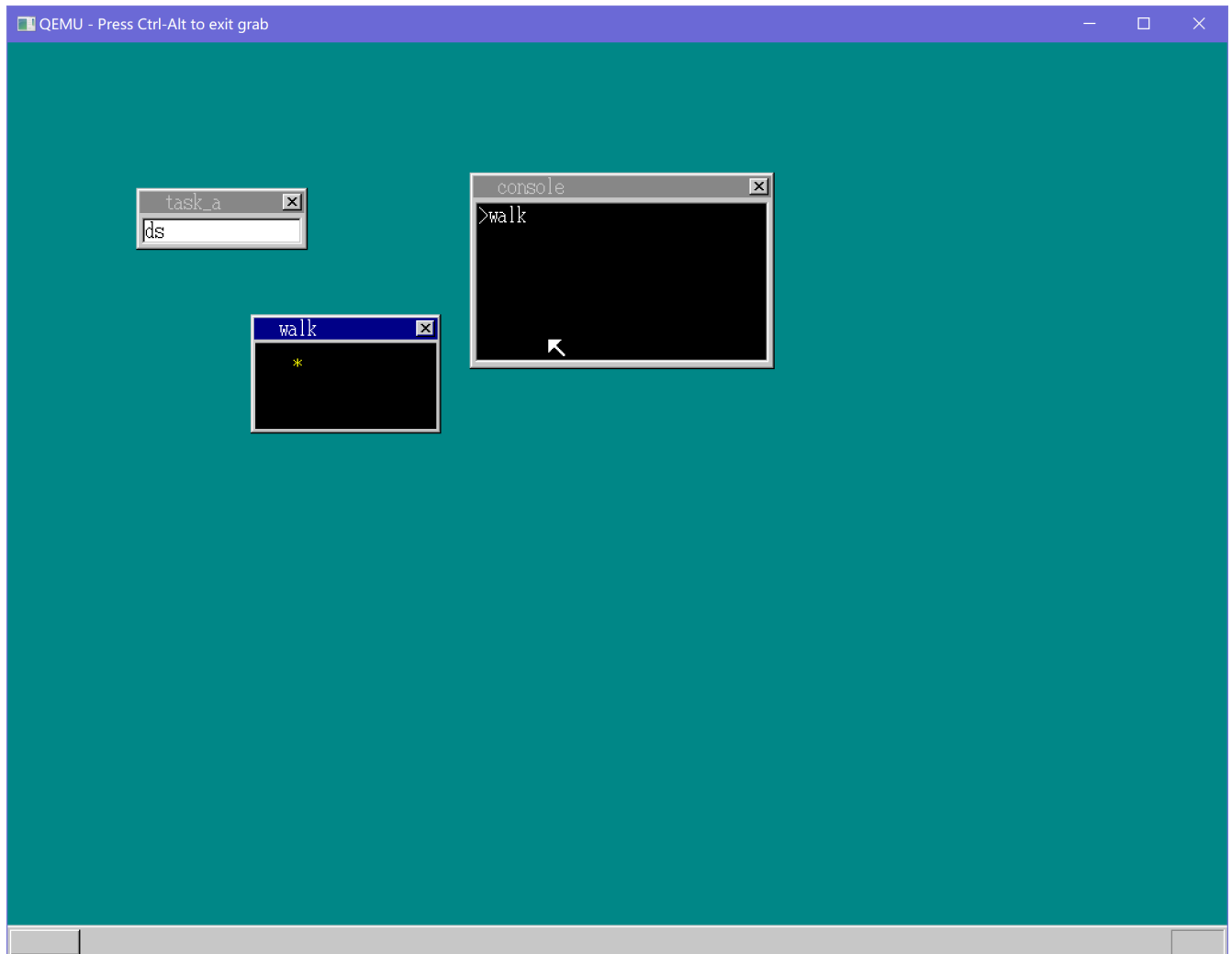
于是hrb\_api也需要进行修改

```

} else if (edx == 5) {
    sht = sheet_alloc(shtctl);
    sht->task = task;
    sht->flags |= 0x10;
    sheet_setbuf(sht, (char *) ebx + ds_base, esi, edi, eax);
    make_window8((char *) ebx + ds_base, esi, edi, (char *) ecx + ds_base, 0);
    sheet_slide(sht, 100, 50);
    sheet_updown(sht, 3);
    reg[7] = (int) sht;
}

```

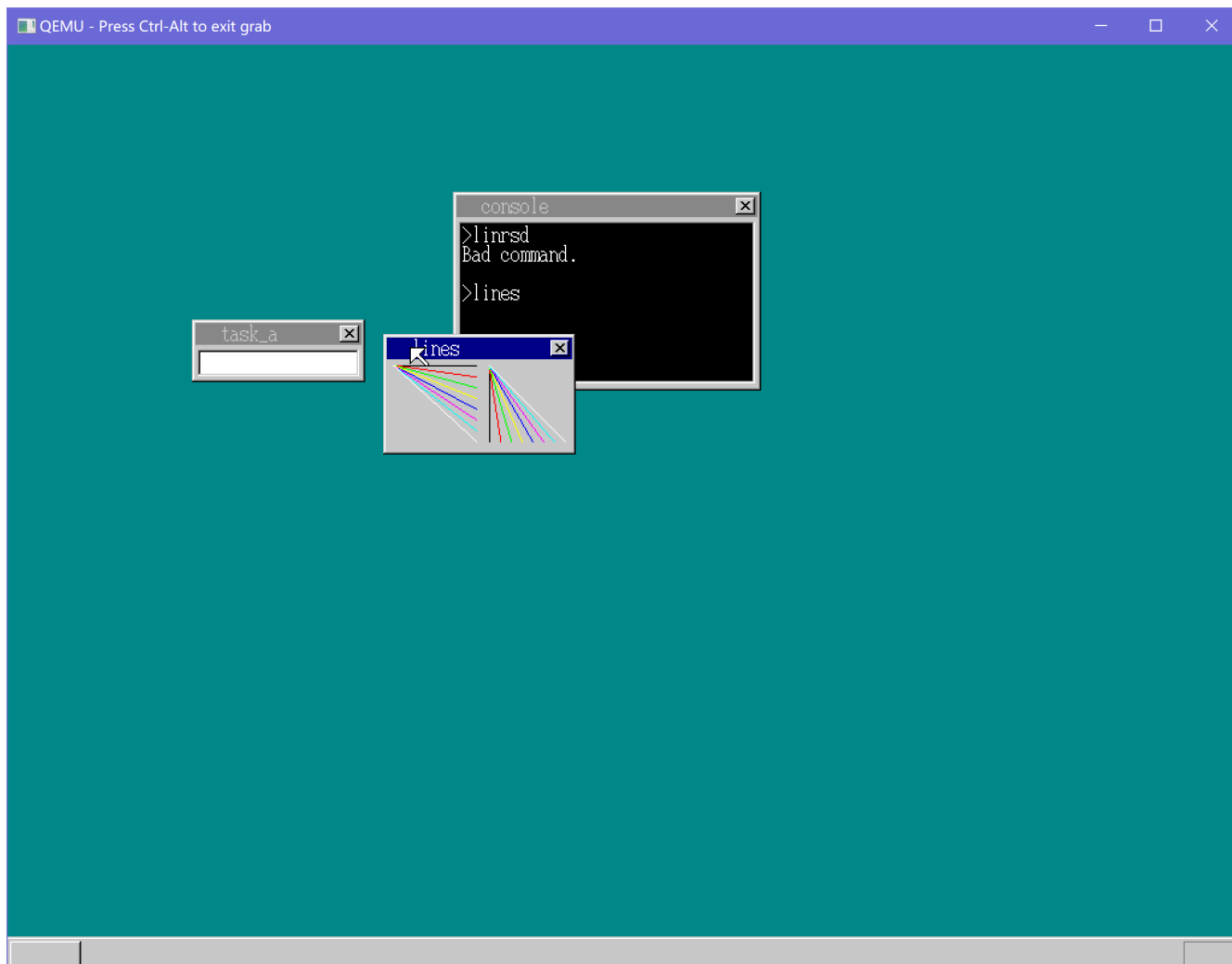
make run成功了



接下来实现用鼠标切换输入窗口。之前实现用tab切换已经修改了大量的代码，这次只要进行少许的修改就可以了。

```
if (sht != key_win) {  
    cursor_c = keywin_off(key_win, sht_win, cursor_c, cursor_x);  
    key_win = sht;  
    cursor_c = keywin_on(key_win, sht_win, cursor_c);  
}
```

测试一下



定时器。不光是操作系统需要使用定时器，应用程序当中也需要使用定时器。我们设计定时器API，让应用程序也能够使用计时器。

获取定时器（alloc）

EDX	16
EAX	定时器句柄（由操作系统返回）

设置定时器的发送数据（init）

EDX	17
EBX	定时器句柄
EAX	数据

定时器时间设定（set）



EDX	18
EBX	定时器句柄
EAX	时间

释放定时器 (free)

EDX	19
EBX	定时器句柄

hrb\_api设计

```

} else if (edx == 16) { /*从此开始*/
    reg[7] = (int) timer_alloc();
} else if (edx == 17) {
    timer_init((struct TIMER *) ebx, &task->fifo, eax + 256);
} else if (edx == 18) {
    timer_settime((struct TIMER *) ebx, eax);
} else if (edx == 19) {
    timer_free((struct TIMER *) ebx); /*到此结束*/
}

```

功能号15也要改一下，因为我们的应用程序不仅需要接收键盘的数据，还需要接受定时器超时所发出的数据

```

if (i >= 256) { /*键盘数据（通过任务A）等*/
    reg[7] = i - 256;
    return 0;
}

```

api定义和声明

```

_api_alloctimer: ; int api_alloctimer(void);
    MOV EDX,16
    INT 0x40
    RET
_api_inittimer: ; void api_inittimer(int timer, int data);
    PUSH EBX
    MOV EDX,17
    MOV EBX,[ESP+ 8] ; timer
    MOV EAX,[ESP+12] ; data
    INT 0x40
    POP EBX
    RET
_api_settimer: ; void api_settimer(int timer, int time);
    PUSH EBX
    MOV EDX,18
    MOV EBX,[ESP+ 8] ; timer
    MOV EAX,[ESP+12] ; time
    INT 0x40
    POP EBX

```

```

    RET
_api_freetimer: ; void api_freetimer(int timer);
    PUSH EBX
    MOV EDX,19
    MOV EBX,[ESP+ 8] ; timer
    INT 0x40
    POP EBX
    RET

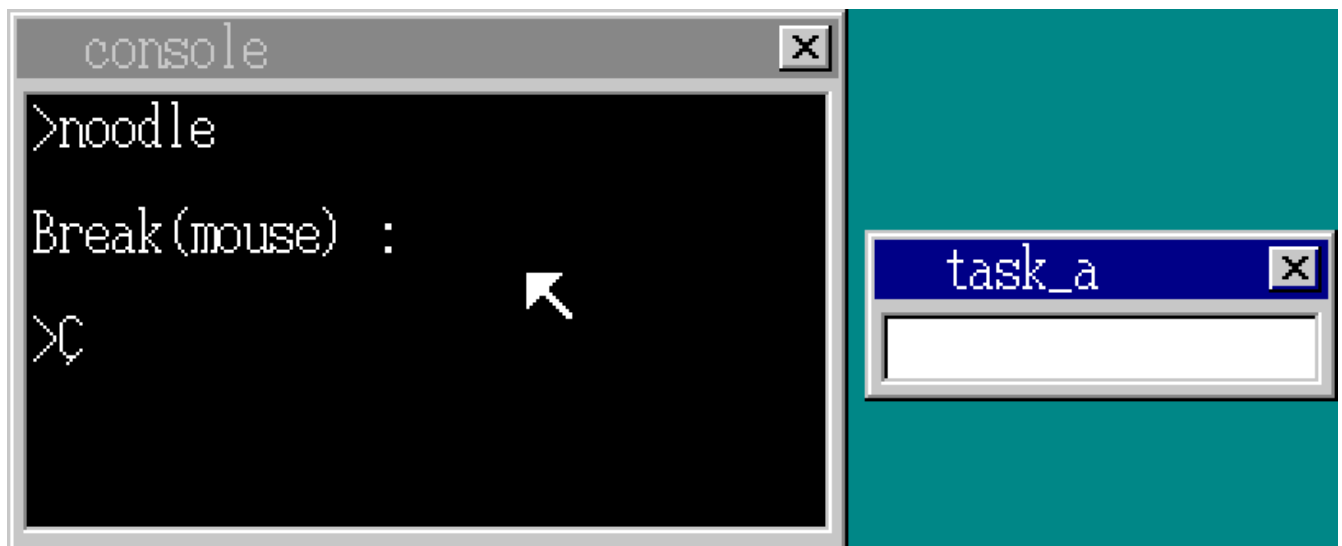
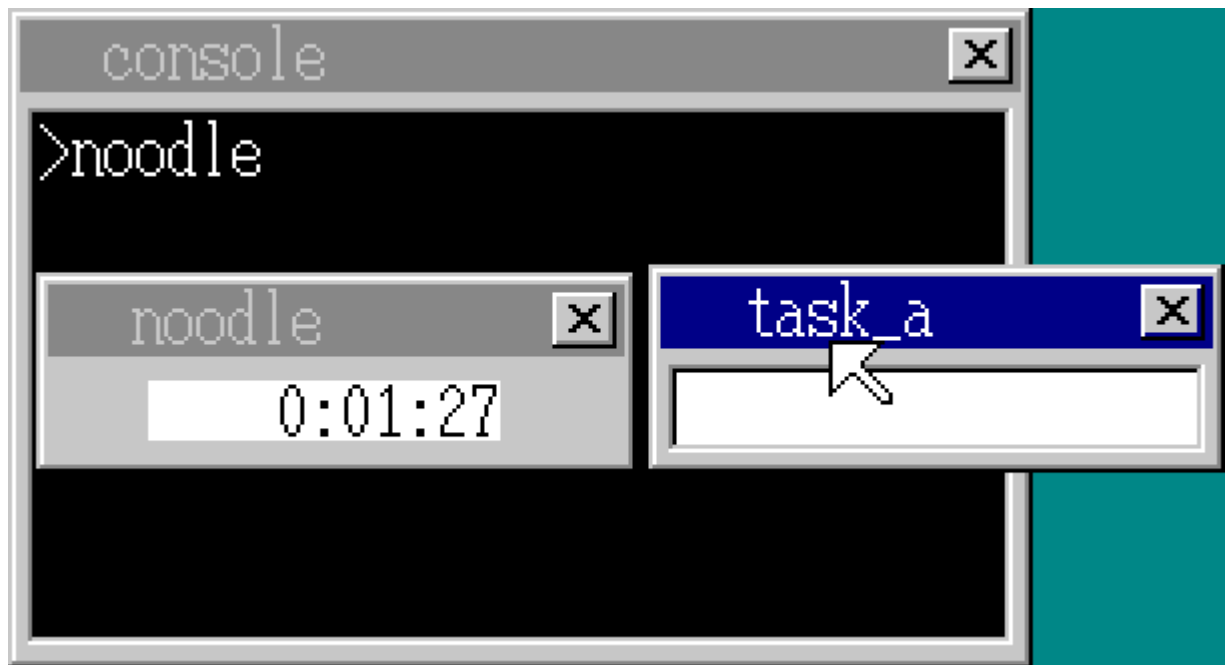
```

我们测试一个时钟功能

```

void HariMain(void)
{
    char *buf, s[12];
    int win, timer, sec = 0, min = 0, hou = 0;
    api_initmalloc();
    buf = api_malloc(150 * 50);
    win = api_openwin(buf, 150, 50, -1, "noodle");
    timer = api_alloctimer();
    api_inittimer(timer, 128);
    for (;;) {
        sprintf(s, "%5d:%02d:%02d", hou, min, sec);
        api_boxfilwin(win, 28, 27, 115, 41, 7);
        api_putstrwin(win, 28, 27, 0, 11, s);
        api_settimer(timer, 100);
        if (api_getkey(1) != 128) {
            break;
        }
        sec++;
        if (sec == 60) {
            sec = 0;
            min++;
            if (min == 60) {
                min = 0;
                hou++;
            }
        }
    }
    api_end();
}

```



我们发现结束程序之后控制台出现了奇怪的字符。

这是因为我们的定时器的数据依然被送到了命令行窗口。我们需要取消待机中的计时器

```
int timer_cancel(struct TIMER *timer)
{
    int e;
    struct TIMER *t;
    e = io_load_eflags();
    io_cli();
    if (timer->flags == TIMER_FLAGS_USING) { /*是否需要取消? */
        if (timer == timerctl.t0) {
            /*第一个定时器的取消处理*/
            t = timer->next;
            timerctl.t0 = t;
            timerctl.next = t->timeout;
        } else {
```

```

        /*非第一个定时器的取消处理*/
        t = timerctl.t0;
        for (;;) {
            if (t->next == timer) {
                break;
            }
            t = t->next;
        }
        t->next = timer->next; /*链表删除*/
    }
    timer->flags = TIMER_FLAGS_ALLOC;
    io_store_eflags(e);
    return 1; /*取消处理成功*/
}
io_store_eflags(e);
return 0; /*不需要取消处理*/
}

```

为了实现定时器的自动取消，我们在timer中添加一个flag

```

struct TIMER {
    struct TIMER *next;
    unsigned int timeout;
    char flags, flags2; //<-----
    struct FIFO32 *fifo;
    int data;
};

struct TIMER *timer_alloc(void)
{
    int i;
    for (i = 0; i < MAX_TIMER; i++) {
        if (timerctl.timers0[i].flags == 0) {
            timerctl.timers0[i].flags = TIMER_FLAGS_ALLOC;
            timerctl.timers0[i].flags2 = 0; //<-----
            return &timerctl.timers0[i];
        }
    }
    return 0;
}

```

hrb\_api

```

} else if (edx == 16) {
    reg[7] = (int) timer_alloc();
    ((struct TIMER *) reg[7])->flags2 = 1; /*允许自动取消*/ //<-----
}

```

```

void timer_cancelall(struct FIFO32 *fifo)
{
    int e, i;
    struct TIMER *t;

```

```
e = io_load_eflags();
io_cli();
for (i = 0; i < MAX_TIMER; i++) {
    t = &timerctl.timers0[i];
    if (t->flags != 0 && t->flags2 != 0 && t->fifo == fifo) {
        timer_cancel(t);
        timer_free(t);
    }
}
io_store_eflags(e);
return;
}
```

然后再cmd\_app的sheet\_free(sht);后面添加timer\_calcelall(&task->info);即可

# 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<sup>①</sup>。

### □ 蜂鸣器ON/OFF

- ◆ 使用 I/O 端口 0x61 控制。
- ◆ ON:  $IN(AL, 0x61)$ ;  $AL \mid= 0x03$ ;  $AL \&= 0x0f$ ;  $OUT(0x61, AL)$ ;
- ◆ 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

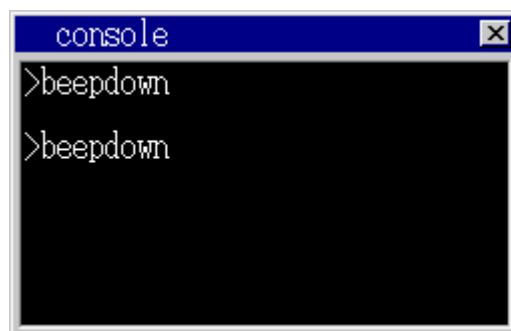
```

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

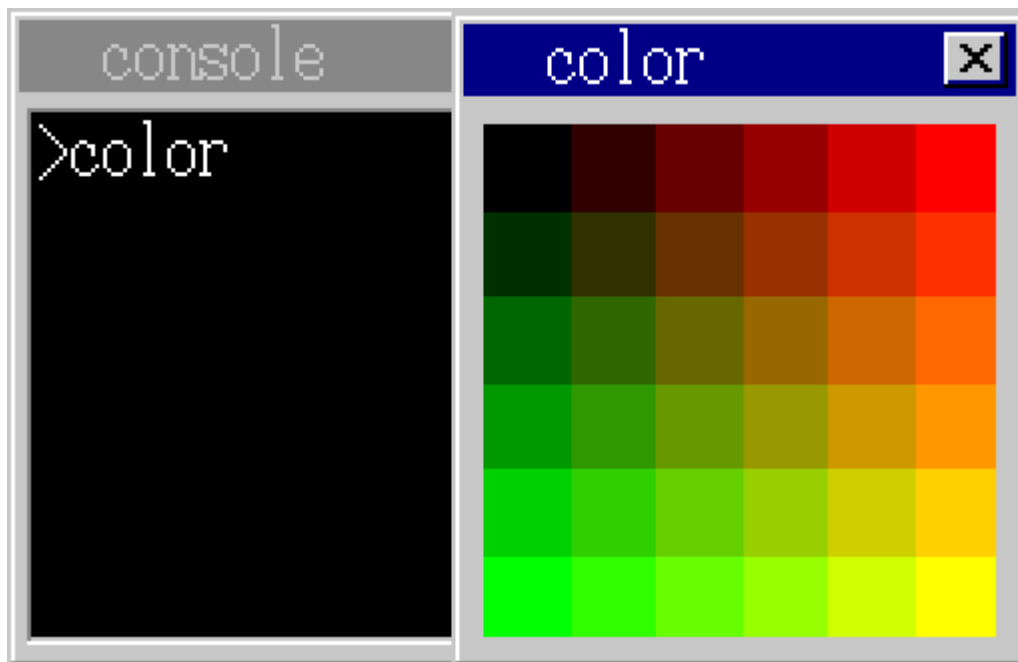
```
unsigned char table2[216 * 3];
int r, g, b;
set_palette(0, 15, table_rgb);
for (b = 0; b < 6; b++) {
    for (g = 0; g < 6; g++) {
        for (r = 0; r < 6; r++) {
            table2[(r + g * 6 + b * 36) * 3 + 0] = r * 51;
            table2[(r + g * 6 + b * 36) * 3 + 1] = g * 51;
            table2[(r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

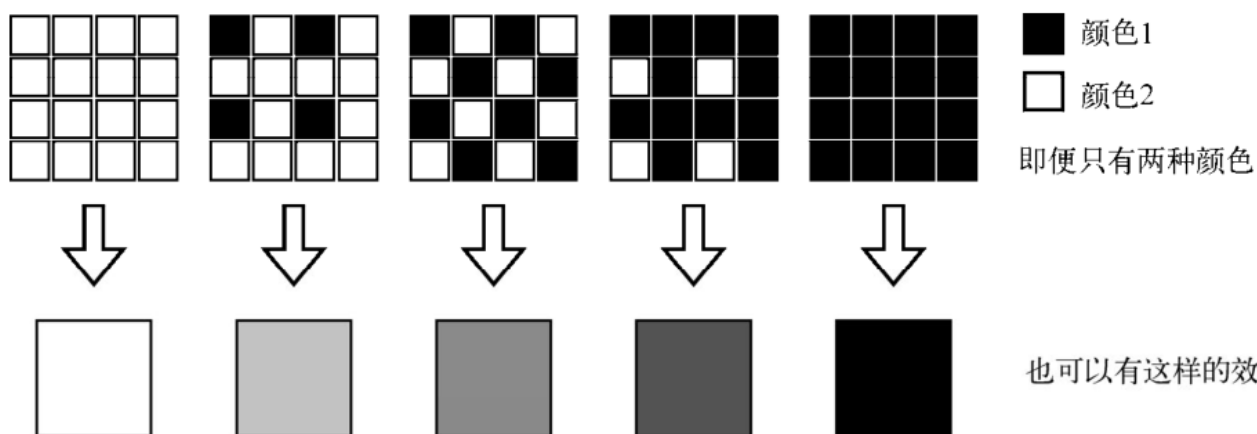
```
int api_openwin(char *buf, int xsiz, int ysiz, int col_inv, char *title);
void api_initmalloc(void);
char *api_malloc(int size);
void api_refreshwin(int win, int x0, int y0, int x1, int y1);
void api_linewin(int win, int x0, int y0, int x1, int y1, int col);
int api_getkey(int mode);
void api_end(void);
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 ([x/4][y/4]) 的方式进行混合排列，就可以产生额外的三种中间色



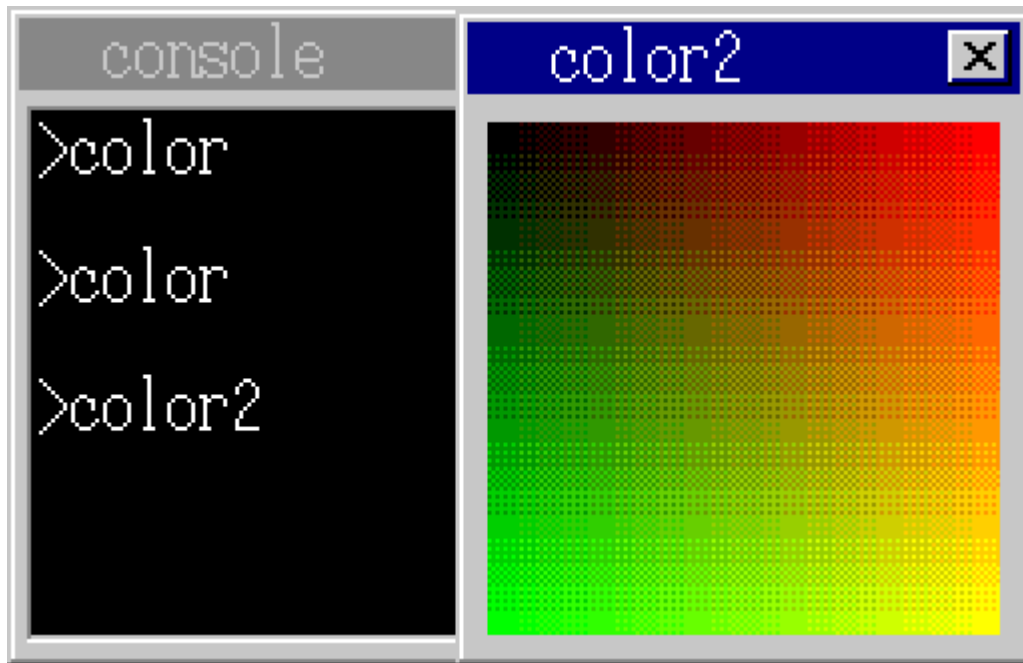
需要修改应用程序

```
/*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

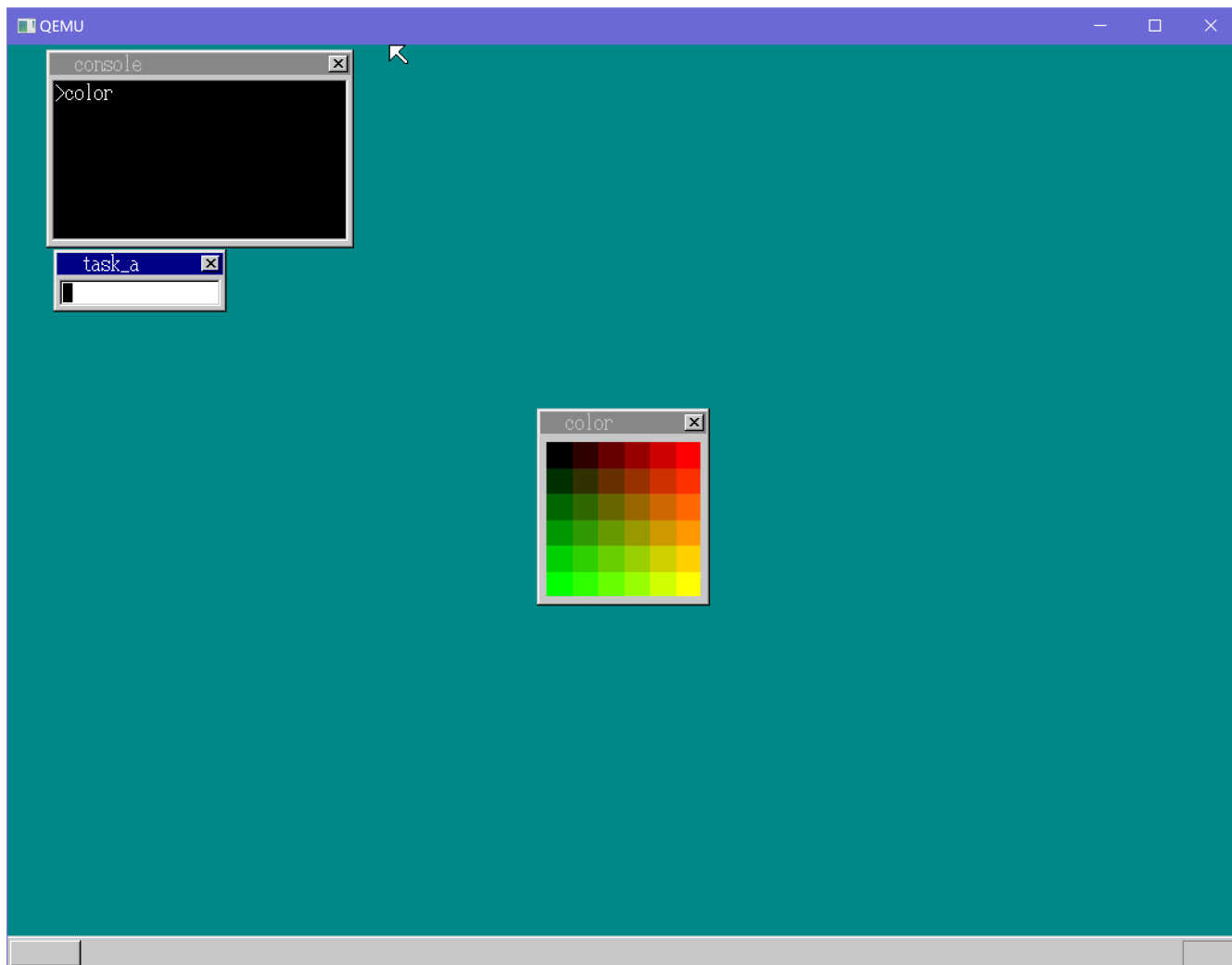
在功能号5的api中断那里加两句

```

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。

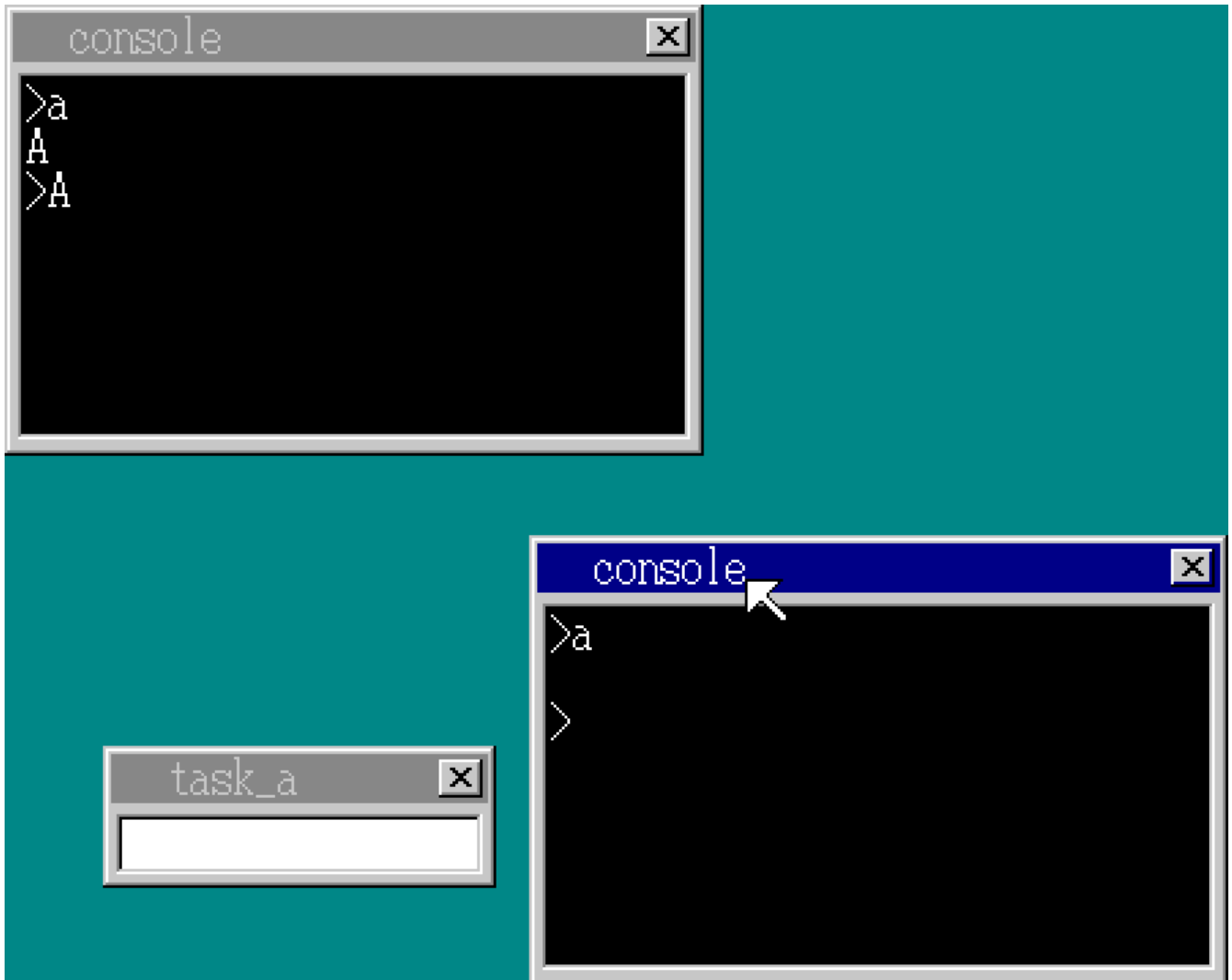
把命令行窗口相关的变量

- buf\_cons
- sht\_cons
- task\_cons
- cons

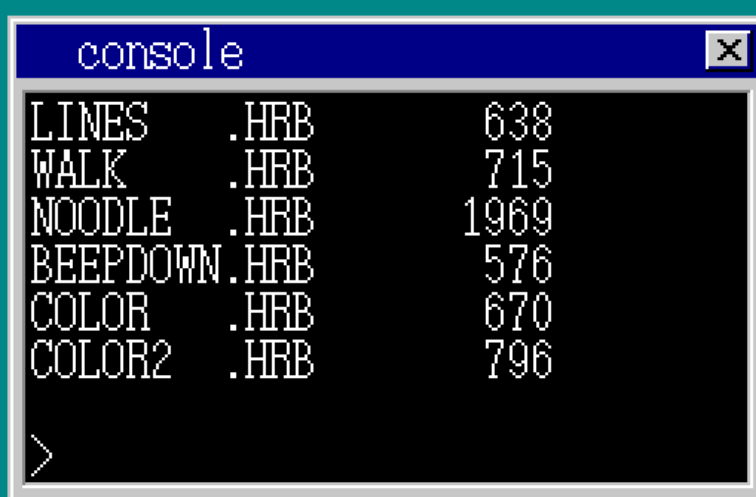
各准备2个，使用数组

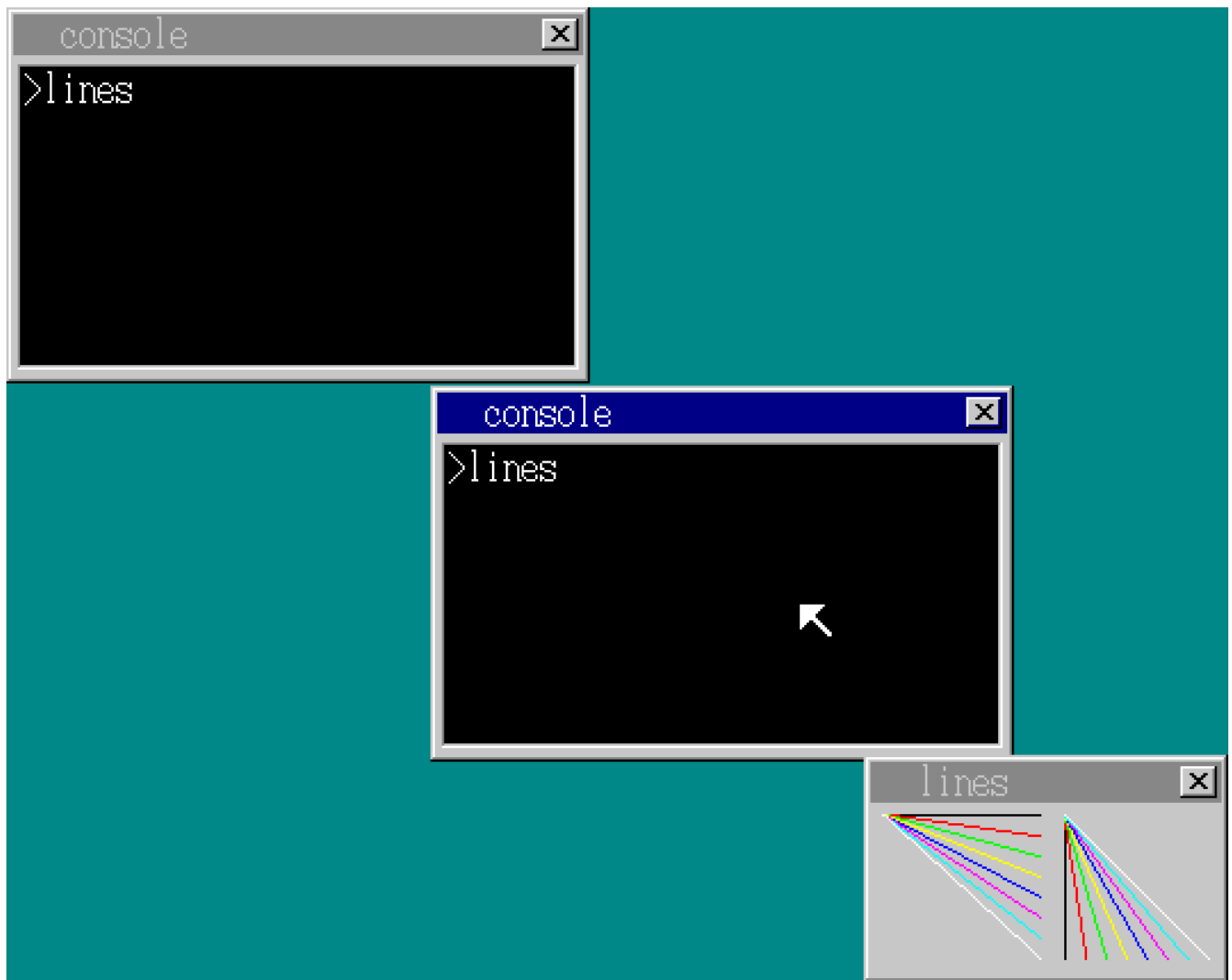
```
unsigned char *buf_back, buf_mouse[256], *buf_win, *buf_cons[2]; /*从此开始*/  
struct SHEET *sht_back, *sht_mouse, *sht_win, *sht_cons[2];  
struct TASK *task_a, *task_cons[2];
```

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）





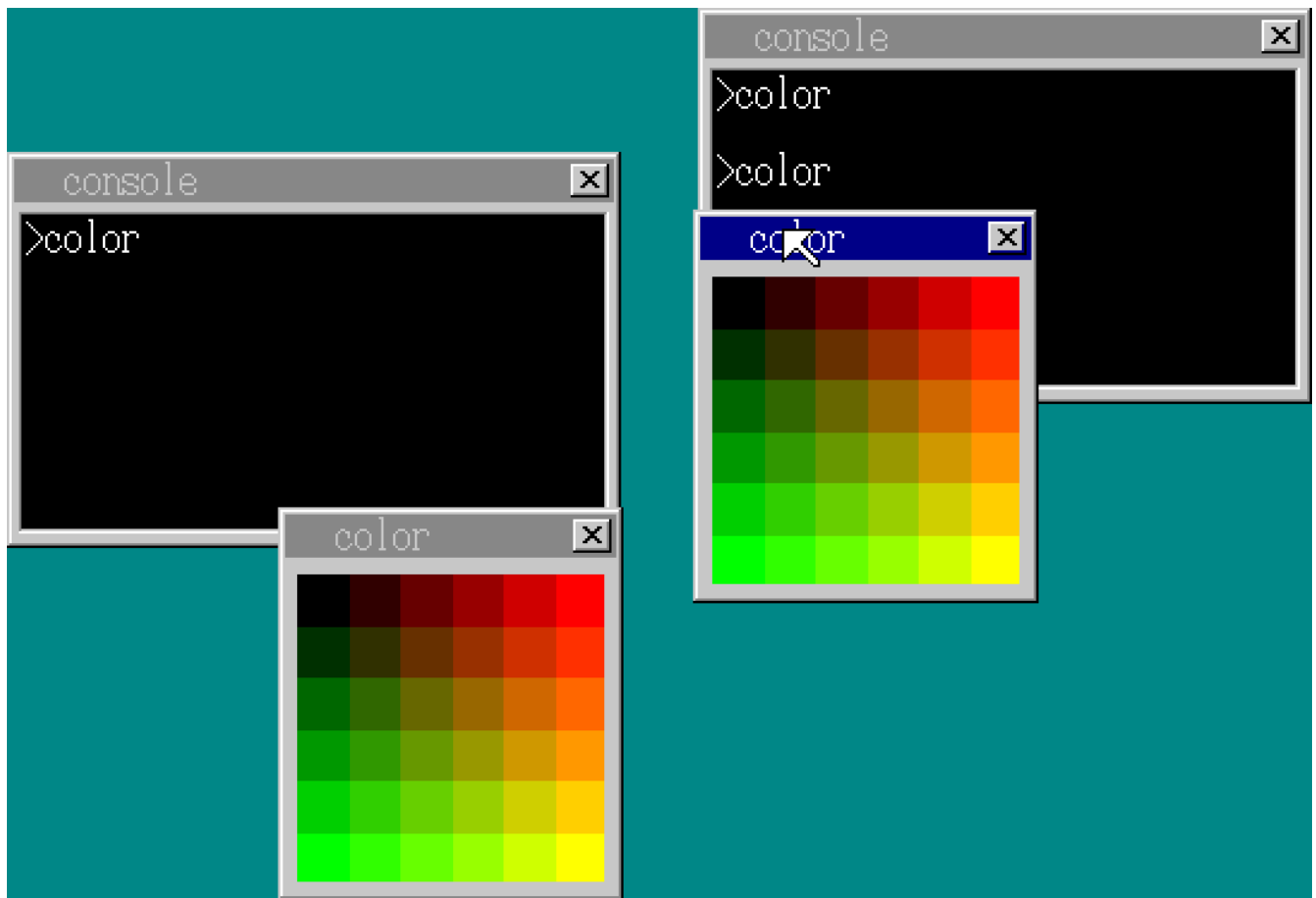
运行两个图形程序，系统直接卡住了

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

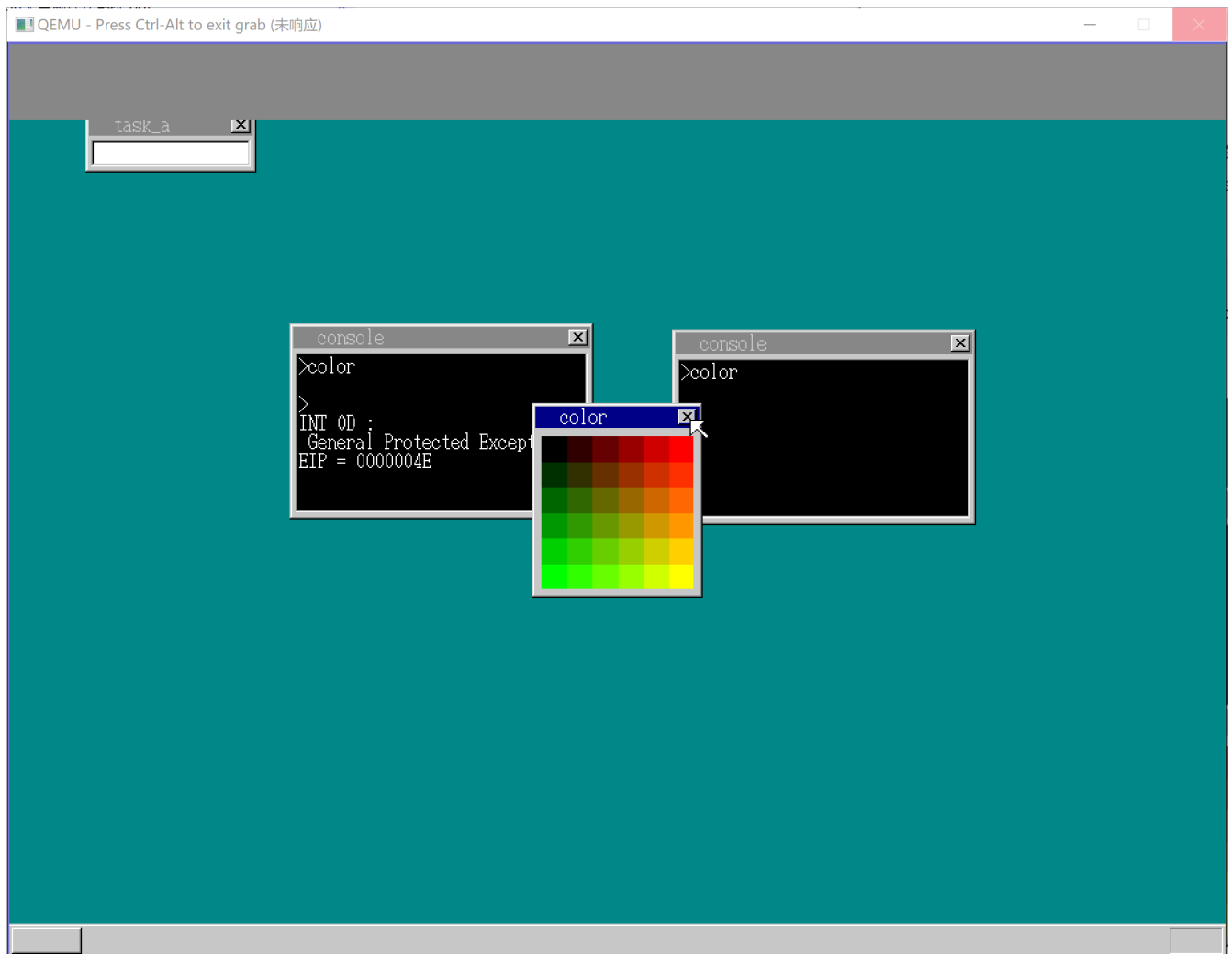
```
int ds_base = task->ds_base;  
struct CONSOLE *cons = task->cons;
```

测试一下



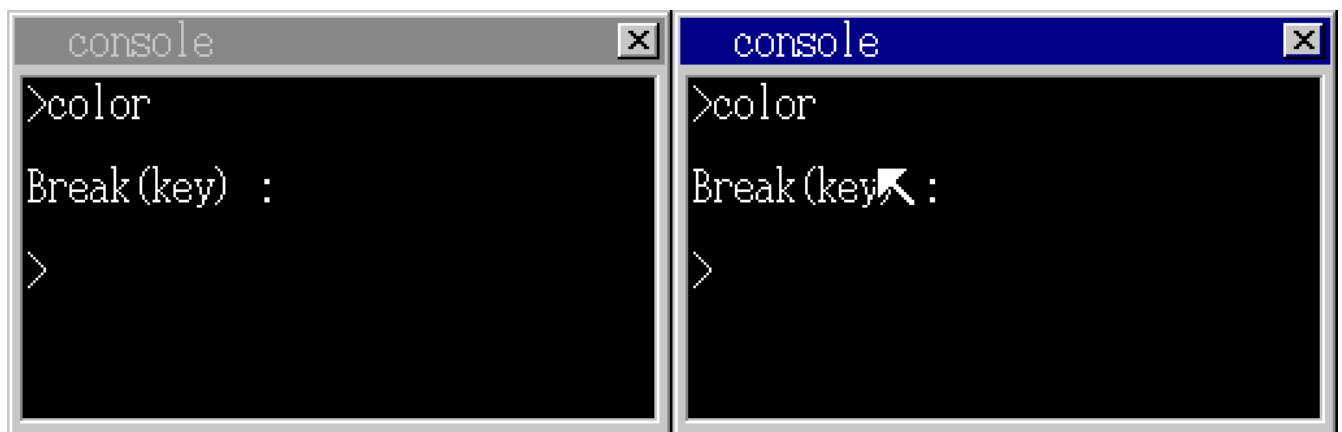
成功的运行了！

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。

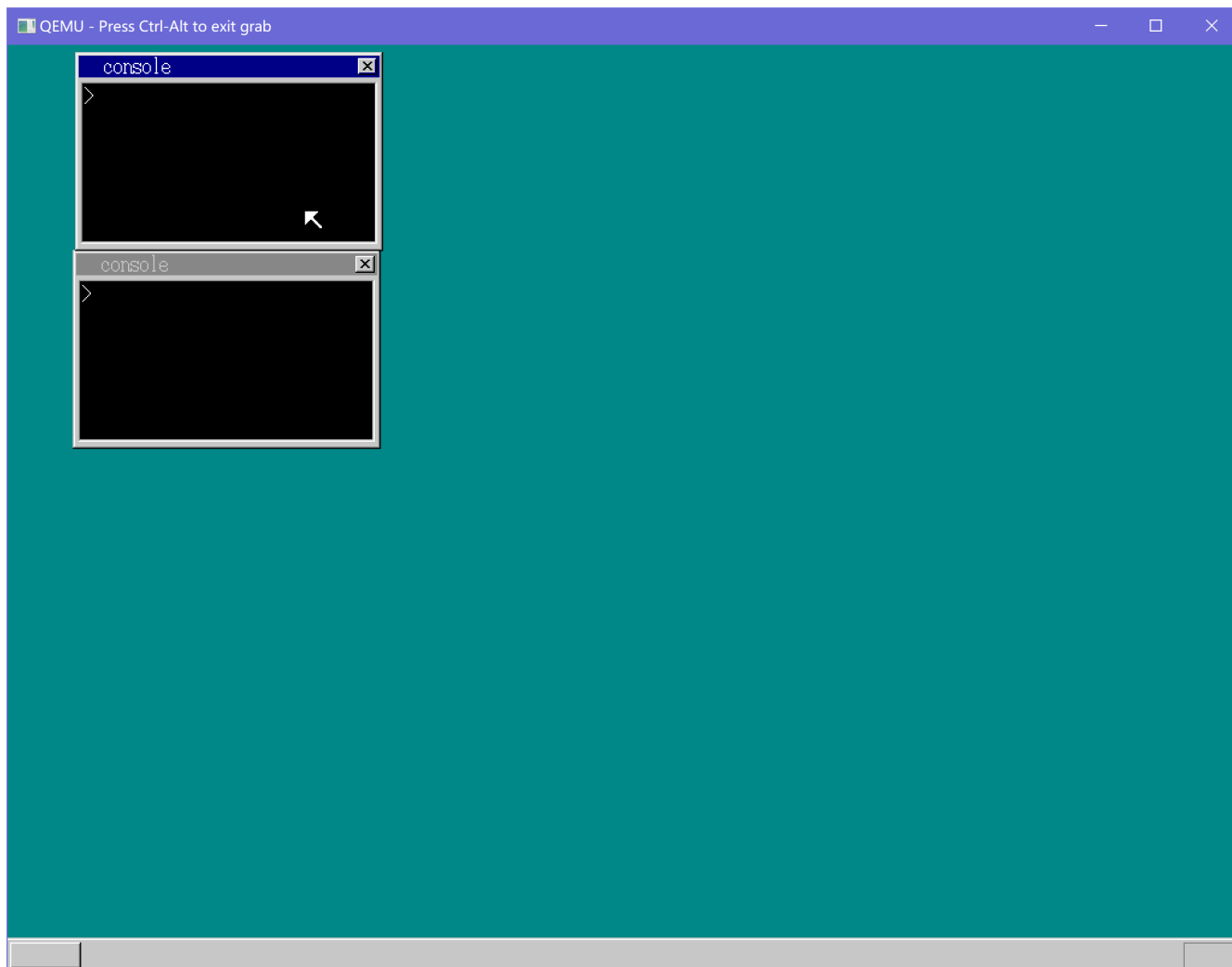
task\_a的逻辑在bootpack当中，先删除掉，然后将console\_task的FIFO初始化放到harimain当中。



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

# Day 26

今天的任务真不少，我们要做的事情有

- 提高窗口移动速度
- 启动时只打开一个命令行窗口
- 增加更多的命令行窗口
- 实现命令行窗口的关闭
- 实现start命令
- 实现ncst命令

提高窗口的移动速度在于提高窗体的绘图速度并减少不必要的绘图。

通过分析sheet\_refreshmap我们发现每次改变像素点颜色之前都会有个if判断该点是否是透明的。在某些图层当中，例如窗体，我们并没有透明部分，所以这个检查是多余的，我们可以在最外层添加一个判断，判断正在绘图的这个窗口是否含有透明部分，如果没有透明的部分就进入没有判断的循环，反之进入有判断的循环。

```
if (sht->col_inv == -1) {
    /*无透明色*/
    for (by = by0; by < by1; by++) {
        vy = sht->vy0 + by;
        for (bx = bx0; bx < bx1; bx++) {
            vx = sht->vx0 + bx;
            map[vy * ctl->xsize + vx] = sid;
        }
    }
} else {
    /*有透明色*/
    for (by = by0; by < by1; by++) {
        vy = sht->vy0 + by;
        for (bx = bx0; bx < bx1; bx++) {
            vx = sht->vx0 + bx;
            if (buf[by * sht->bxsize + bx] != sht->col_inv) {
                map[vy * ctl->xsize + vx] = sid;
            }
        }
    }
}
```

注意到我们是将内存中连续区域复制到显存的多组连续区域当中，我们目前每次复制一字节，如果我们一次能赋值更多字节就好了。

作者通过指针转换的方式实现了一次复制4字节。但这弄得有点小麻烦，其实memcpy函数已经是能够按照机器字长进行拷贝的，我们直接用memcpy应该也可以提高速度的。（作者这种写法反而使得窗体只能移动到4的倍数的位置上了）

还有一些其他的trick可以使用，但作者并没有使用。例如**循环展开**：循环展开可以充分利用CPU的多个功能单元，使得指令能够并行执行从而得到常数级别的指令运行速度提升。

我们把这些trick也应用到sheet\_refreshsub上

```
for (i = 0; i < i1; i++) { /* 4的倍数部分*/
    if (p[i] == sid4) {
        q[i] = r[i];
    } else {
        bx2 = bx + i * 4;
        vx = sht->vx0 + bx2;
        if (map[vy * ctl->xsize + vx + 0] == sid) {
            vram[vy * ctl->xsize + vx + 0] = buf[by * sht->bysize + bx2 + 0];
        }
        if (map[vy * ctl->xsize + vx + 1] == sid) {
            vram[vy * ctl->xsize + vx + 1] = buf[by * sht->bysize + bx2 + 1];
        }
        if (map[vy * ctl->xsize + vx + 2] == sid) {
            vram[vy * ctl->xsize + vx + 2] = buf[by * sht->bysize + bx2 + 2];
        }
        if (map[vy * ctl->xsize + vx + 3] == sid) {
            vram[vy * ctl->xsize + vx + 3] = buf[by * sht->bysize + bx2 + 3];
        }
    }
}
for (bx += i1 * 4; bx < bx1; bx++) { /*剩余部分逐字节写入*/
    vx = sht->vx0 + bx;
    if (map[vy * ctl->xsize + vx] == sid) {
        vram[vy * ctl->xsize + vx] = buf[by * sht->bysize + bx];
    }
}
```

得益于计算机性能的提升，即使实在模拟器当中运行，目前为止的窗体移动速度已经足够快了

但是还是有优化的空间的。

当FIFO中有多个数据时进行绘图是不明智的，因为我们绘制完的窗体在FIFO中的数据得到处理之后马上又会变更了，这个FIFO处理的很快，我们绘制完的窗体用户还没怎么看见马上就又需要重绘了，所以我们决定等待FIFO为空之后再进行绘图。

```
if (fifo32_status(&fifo) == 0) {
    /* FIFO为空，当存在搁置的绘图操作时立即执行*/ /*从此开始*/
    if (new_mx >= 0) {
        io_sti();
        sheet_slide(sht_mouse, new_mx, new_my);
        new_mx = -1;
    } else if (new_wx != 0x7fffffff) {
        io_sti();
        sheet_slide(sht, new_wx, new_wy);
        new_wx = 0x7fffffff;
    } else {
        task_sleep(task_a);
        io_sti();
    } /*到此结束*/
} else {
    // .....
}
```

```

} else {
    /*没有按下鼠标左键*/
    mmx = -1; /*切换到一般模式*/
    if (new_wx != 0x7fffffff) { /*从此开始*/
        sheet_slide(sht, new_wx, new_wy); /*固定图层位置*/
        new_wx = 0x7fffffff;
    } /*到此结束*/
}

```

我们把鼠标移动的位置暂时的保存下来，而不真的移动鼠标的位置（new\_mx new\_my）

窗体同理，不过我们用来标记暂时不需要绘图的值用的是0x7fffffff而不是-1，这是因为窗体坐标是有可能为-1的。

当鼠标左键弹起推出窗口移动模式的时候，我们也要立即更新窗口的位置，这是因为用户可能接着回去操作移动别的窗口，sht会发生改变，导致我们之后移动的是错误的窗体。

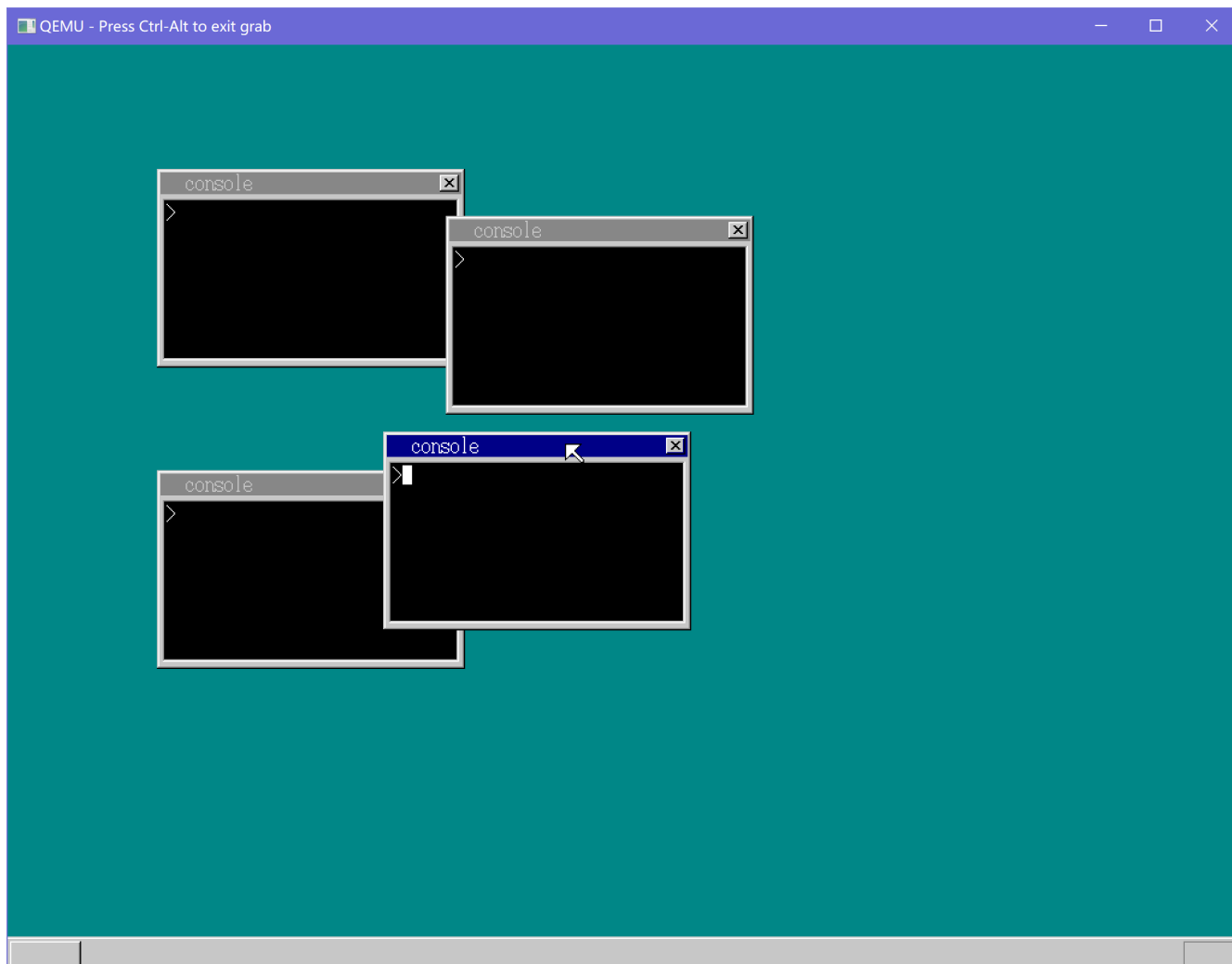
（其实做了这个优化之后直观上感觉速度和之前没有什么提升）

系统启动的时候自动打开两个命令行窗口很奇怪，一般都是先开一个，然后用户根据需求打开更多的窗口。我们先把命令行窗口的数量减少到一个，然后设定一个打开命令行窗口的快捷键 Shift + F2

```

if (i == 256 + 0x3c && key_shift != 0) { /* Shift+F2 */
    keywin_off(key_win);
    key_win = open_console(shtctl, memtotal);
    sheet_slide(key_win, 32, 4);
    sheet_updown(key_win, shtctl->top);
    keywin_on(key_win);
}

```



既然已经实现了打开多个命令行窗口，我们也应该实现命令行窗口的关闭功能。具体思路是回退打开命令行窗口所造成的影响。

- 释放内存空间
- 回收图层
- 回收task
- 回收栈

为了能够回收栈，我们把栈地址记录在task当中

把必要信息在open\_console中记录在task中

```
task->cons_stack = memman_alloc_4k(memman, 64 * 1024);
task->tss.esp = task->cons_stack + 64 * 1024 - 12;
```

然后我们的思路就比较清晰了。

```
void close_constack(struct TASK *task)
{
    struct MEMMAN *memman = (struct MEMMAN *) MEMMAN_ADDR;
    task_sleep(task);
    memman_free_4k(memman, task->cons_stack, 64 * 1024);
    memman_free_4k(memman, (int) task->fifo.buf, 128 * 4);
}
```

```

    task->flags = 0;
    return;
}
void close_console(struct SHEET *sht)
{
    struct MEMMAN *memman = (struct MEMMAN *) MEMMAN_ADDR;
    struct TASK *task = sht->task;
    memman_free_4k(memman, (int) sht->buf, 256 * 165);
    sheet_free(sht);
    close_constask(task);
    return;
}

```

先释放内存，然后释放图层，最后关闭任务。关闭任务的部分比较有趣：我们先让任务休眠，这是为了将任务从ready队列中安全地一处出来，这样就不会再调度到这个任务了。然后我们才可以安全地释放栈和FIFO缓冲区。当内存释放完毕之后，我们给task的flags标记为0，以便下次分配使用。

然后我们要做的就是将exit命令和我们上面写的函数进行绑定。

有一个比较头疼的问题，就是如果我们的console自己调用了close\_console的话，那么它sleep了自己之后，接下来的代码都无法执行了，所以我们让另一个task来进行这个操作。我们之前只是取消了task\_a的窗口，但是task\_a本身还在，所以我们让他来负责close\_console就好了。

通过命令行窗口任务向task\_a的FIFO发送数据，然后task\_a读到数据并关闭对应的console。

console发送完数据之后就可以休眠了。

```

void cmd_exit(struct CONSOLE *cons, int *fat)
{
    struct MEMMAN *memman = (struct MEMMAN *) MEMMAN_ADDR;
    struct TASK *task = task_now();
    struct SHTCTL *shtctl = (struct SHTCTL *) *((int *) 0x0fe4);
    struct FIFO32 *fifo = (struct FIFO32 *) *((int *) 0x0fec);
    timer_cancel(cons->timer);
    memman_free_4k(memman, (int) fat, 4 * 2880);
    io_cli();
    fifo32_put(fifo, cons->sht - shtctl->sheets0 + 768);
    io_sti();
    for (;;) {
        task_sleep(task);
    }
}

```

```

/*-----HariMain-----*/
else if (768 <= i && i <= 1023) {
    close_console(shtctl->sheets0 + (i - 768));
}


```

如果我们的所有窗口都被关闭了，是会出问题的，这里我们需要处理一下

```

if (key_win != 0 && key_win->flags == 0) {
    if (shtctl->top == 1) { /*只剩鼠标和背景*/
        key_win = 0;
    } else {
        key_win = shtctl->sheets[shtctl->top - 1];
        keywin_on(key_win);
    }
}

```

然后鼠标点击  的部分我们也进行修改，以便我们能够通过鼠标来关闭命令行窗口

```

if ((sht->flags & 0x10) != 0) { /*是否为应用程序
窗口? */
    //.....
} else { /*命令行窗口*/
    task = sht->task;
    io_cli();
    fifo32_put(&task->fifo, 4);
    io_sti();
}

```

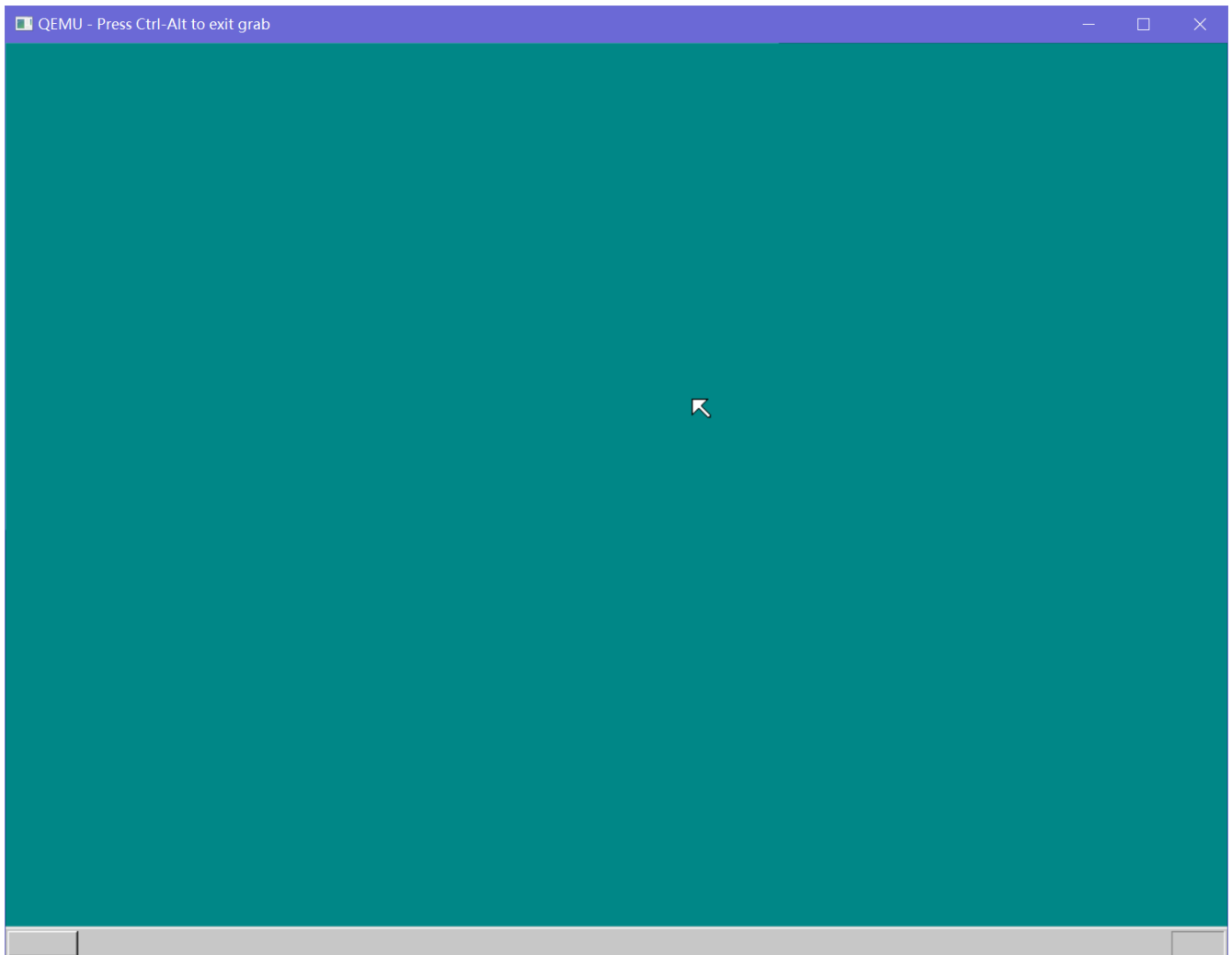
consoletask中对队列数据处理部分要稍微改一下

```

if (fifo32_status(&task->fifo) == 0) {
    //.....
} else {
    //.....
    if (i == 4) {
        cmd_exit(&cons, fat);
    }
    //.....
}

```

测试一下



工作正常

实现start命令，start命令的功能是打开一个新的命令行窗口并运行指定的程序

```
} else if (strncmp(cmdline, "start ", 6) == 0) {  
    cmd_start(cons, cmdline, memtotal);  
}
```

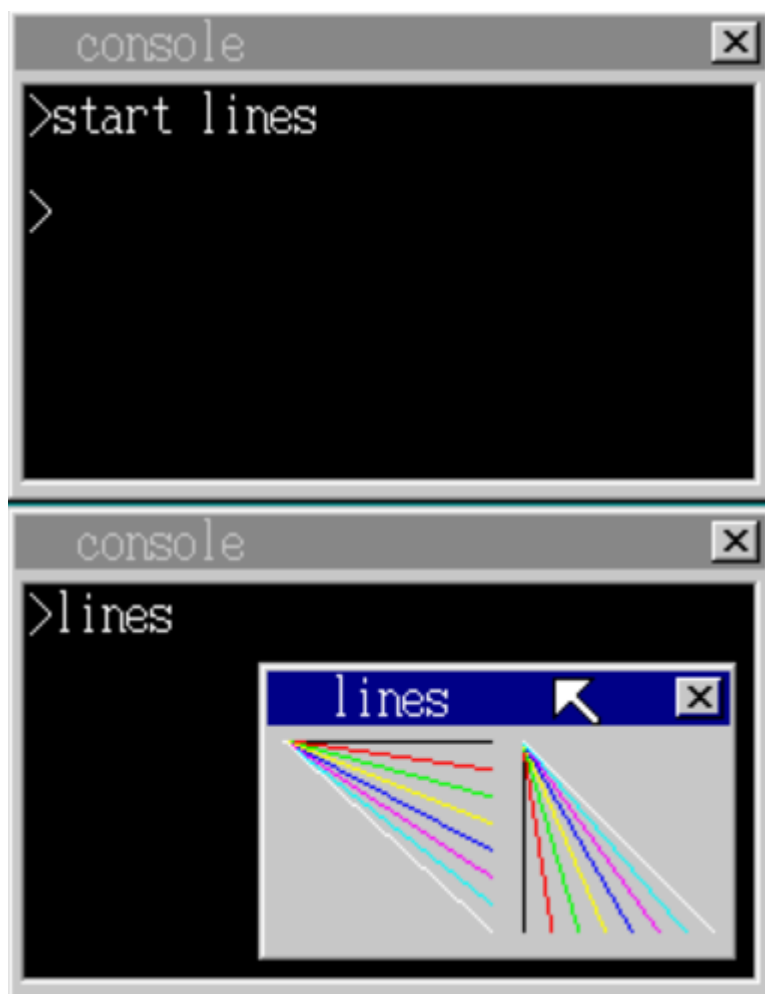
```
void cmd_start(struct CONSOLE *cons, char *cmdline, int memtotal)  
{  
    struct SHTCTL *shtctl = (struct SHTCTL *) *((int *) 0x0fe4);  
    struct SHEET *sht = open_console(shtctl, memtotal);  
    struct FIFO32 *fifo = &sht->task->fifo;  
    int i;  
    sheet_slide(sht, 32, 4);  
    sheet_updown(sht, shtctl->top);  
    for (i = 6; cmdline[i] != 0; i++) {  
        fifo32_put(fifo, cmdline[i] + 256);  
    }  
    fifo32_put(fifo, 10 + 256); /*回车键*/  
    cons_newline(cons);  
}
```



```
    return;  
}
```

思路就是新建一个命令行窗口，然后把命令按键逐个发送过去。

试一下吧



成功

今天最后的任务是ncst命令。ncst是no console start的缩写，意思是不打开命令行窗口的启动。

思路是禁止向命令行窗口显示内容（避免出现错误）

将命令行任务的 `cons->sht` 规定为0。由于没有窗口，所以命令行窗口的内建命令我们可以直接忽略。

```

/*-----cons_runcmd-----*/
if (strcmp(cmdline, "mem") == 0 && cons->sht != 0) {
    cmd_mem(cons, memtotal);
} else if (strcmp(cmdline, "cls") == 0 && cons->sht != 0) {
    cmd_cls(cons);
} else if (strcmp(cmdline, "dir") == 0 && cons->sht != 0) {
    cmd_dir(cons);
} else if (strncmp(cmdline, "type ", 5) == 0 && cons->sht != 0) {
    cmd_type(cons, fat, cmdline);
}

```

```

void cmd_ncst(struct CONSOLE *cons, char *cmdline, int memtotal)
{
    struct TASK *task = open_constask(0, memtotal);
    struct FIFO32 *fifo = &task->fifo;
    int i;
    for (i = 5; cmdline[i] != 0; i++) {
        fifo32_put(fifo, cmdline[i] + 256);
    }
    fifo32_put(fifo, 10 + 256);
    cons_newline(cons);
    return;
}

```

充分利用已经做好的功能，我们仍然使用和start相似的做法

然后我们在cons\_putchar和newline中也进行修改，也加上类似的判断。然后屏蔽输出这部分我们就完成了。

然后就是程序结束后的自动退出

```

if (sheet == 0) {
    cmd_exit(&cons, fat);
} /*这一句应该放在cons_runcmd之后*/

```

cmd\_exit也需要修改，有命令行窗口时，我们可以通过图层的地址告诉task\_a需要结束哪个任务，现在我们需要用FIFO的方法来告诉task\_a

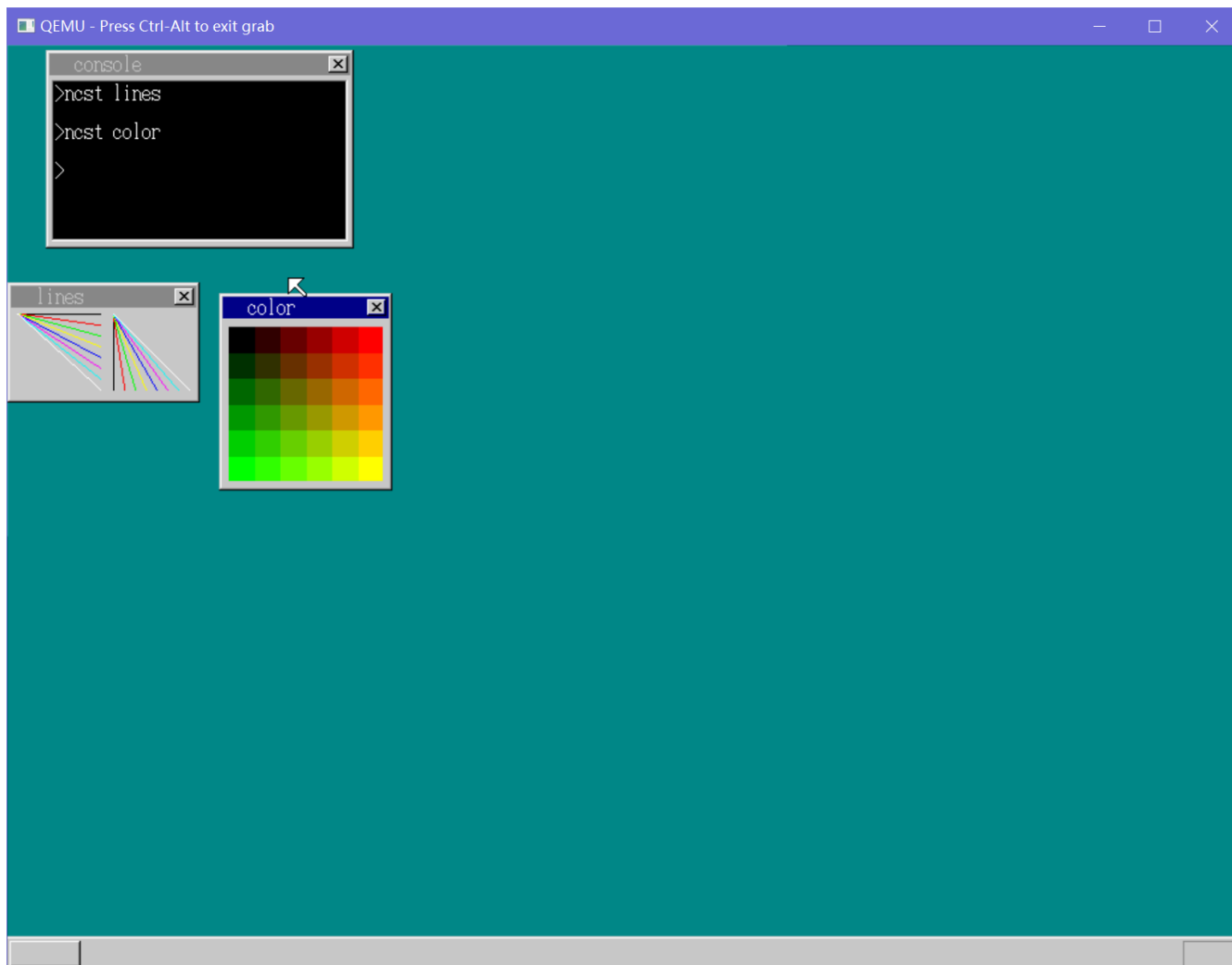
harimain中也需要些小修改

```

} else if (768 <= i && i <= 1023) { /*命令行窗口关闭处理*/
    close_console(shtctl->sheets0 + (i - 768));
} else if (1024 <= i && i <= 2023) {
    close_constask(taskctl->tasks0 + (i - 1024));
}

```

测试一下



点击关闭却出现了问题，应用程序窗口无法被关闭。