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->bxsize && 0 <= y && y < sht->bysize) {
                if (sht->buf[y * sht->bxsize + x] != sht->col_inv) {
                    sheet_updown(sht, shtctl->top - 1);
                    break;
                }
            }
        }
    }
}
```

make run一发

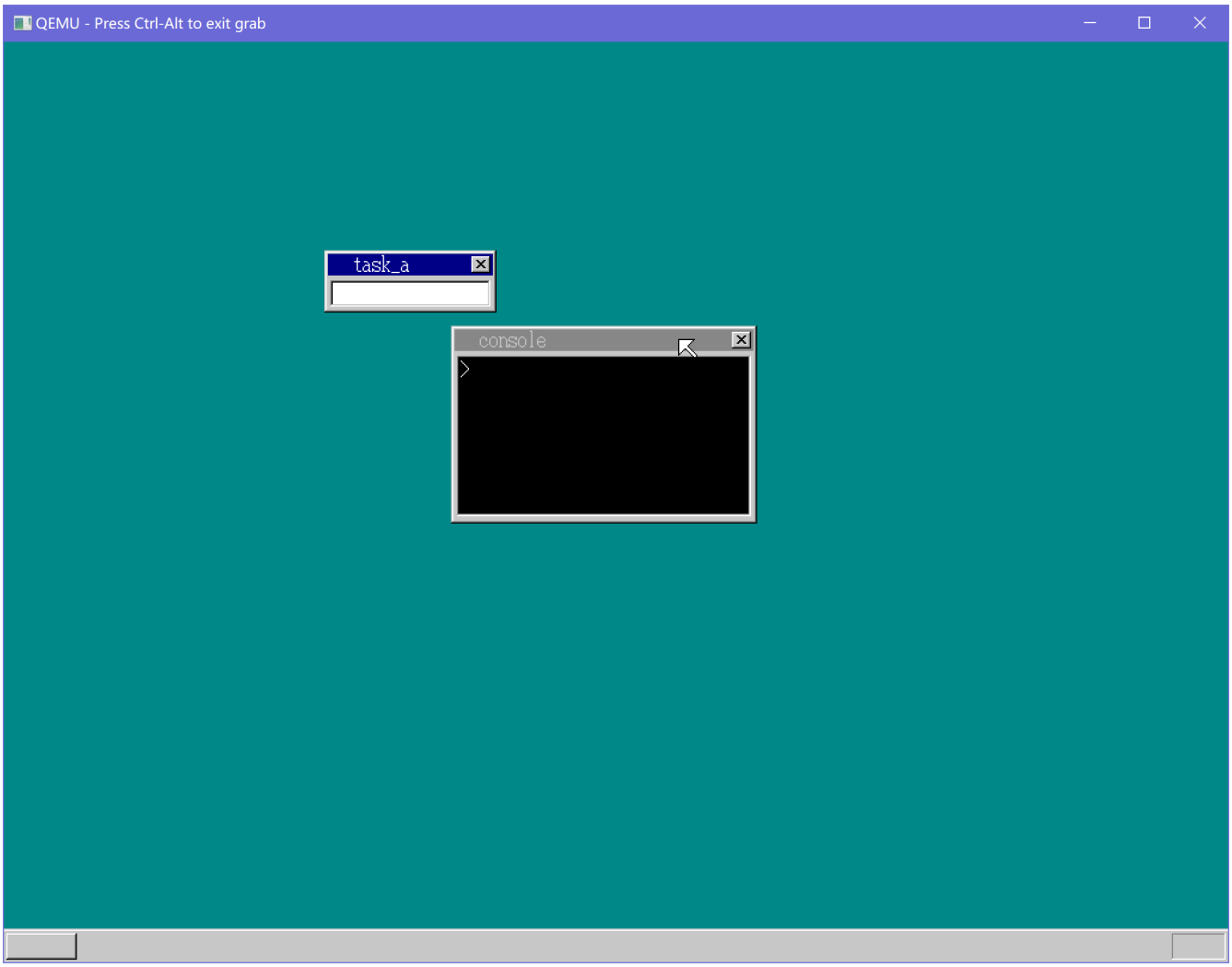
鼠标点击成功的把console调到了顶层

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->bxsize && 0 <= y && y < sht->bysize) {
```

```
                        if (sht->buf[y * sht->bxsize + x] != sht->col_inv) {
                            sheet_updown(sht, shtctl->top - 1);
                            if (3 <= x && x < sht->bxsize - 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;
        }
}
```

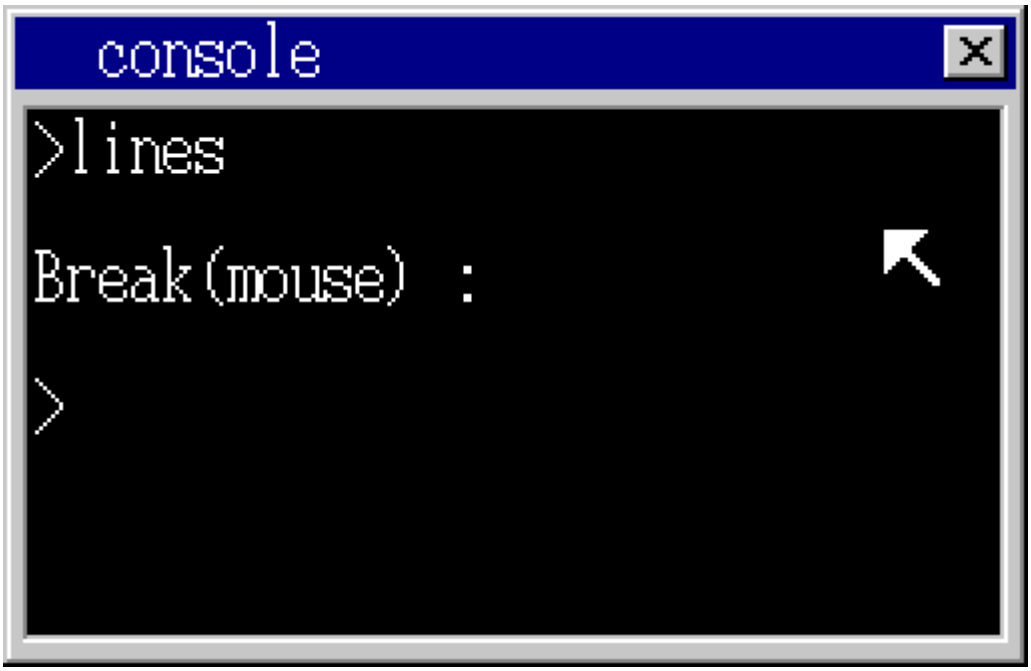


棒极了

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->bxsize - 21 <= x && x < sht->bxsize - 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->bxsize, COL8_FFFFFF, 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->bxsize;
    char c, tc_new, tbc_new, tc_old, tbc_old, *buf = sht->buf;
    if (act != 0) {
        tc_new = COL8_FFFFFF;
        tbc_new = COL8_000084;
        tc_old = COL8_C6C6C6;
        tbc_old = COL8_848484;
    } else {
        tc_new = COL8_C6C6C6;
        tbc_new = COL8_848484;
        tc_old = COL8_FFFFFF;
        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;
}
```

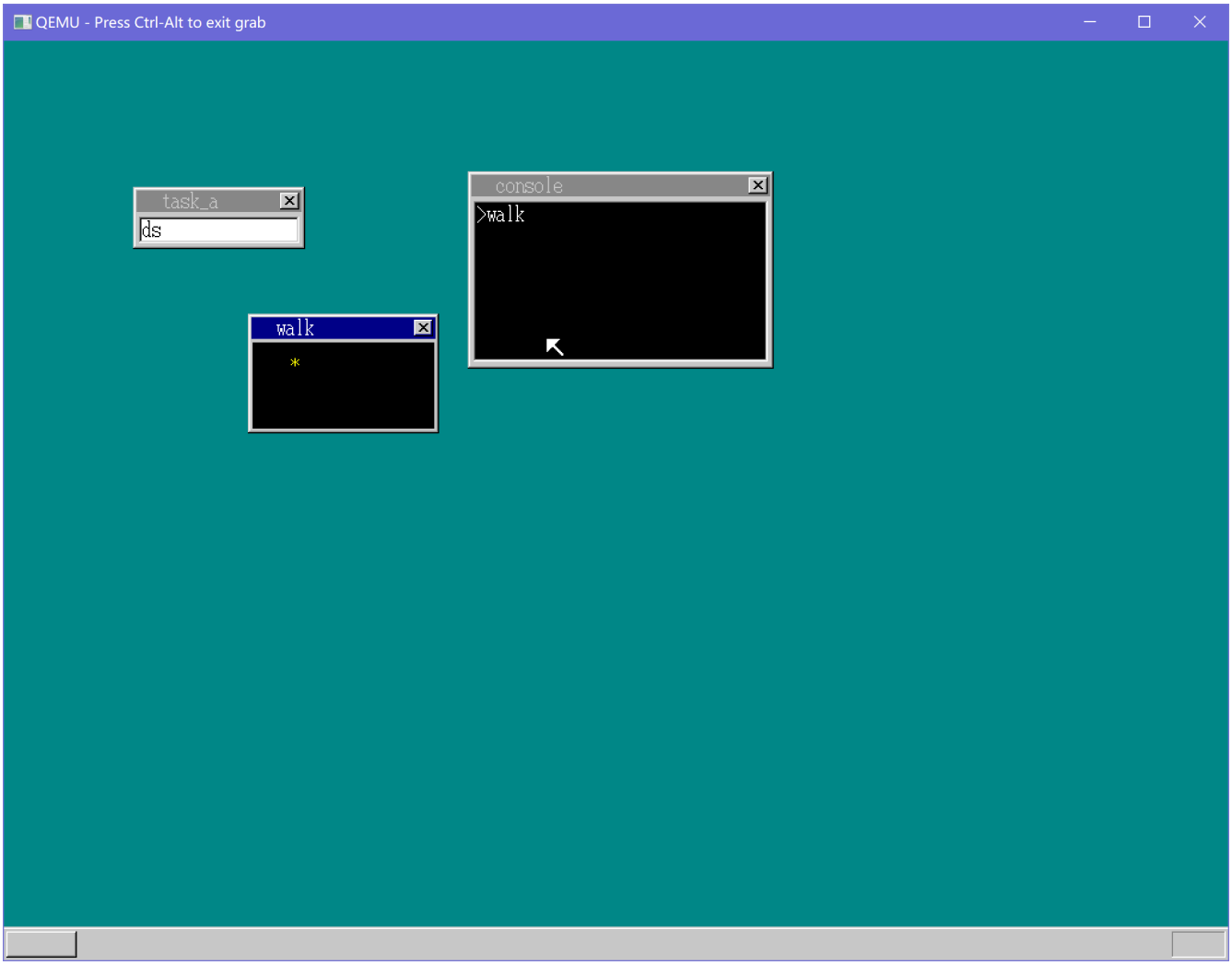
cmd_app也需要修改。由于没有运行应用程序的命令行窗口，所以它的task也不为0，需要通过flags&0x11来进行判断是否自动关闭。

```
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;
}
```

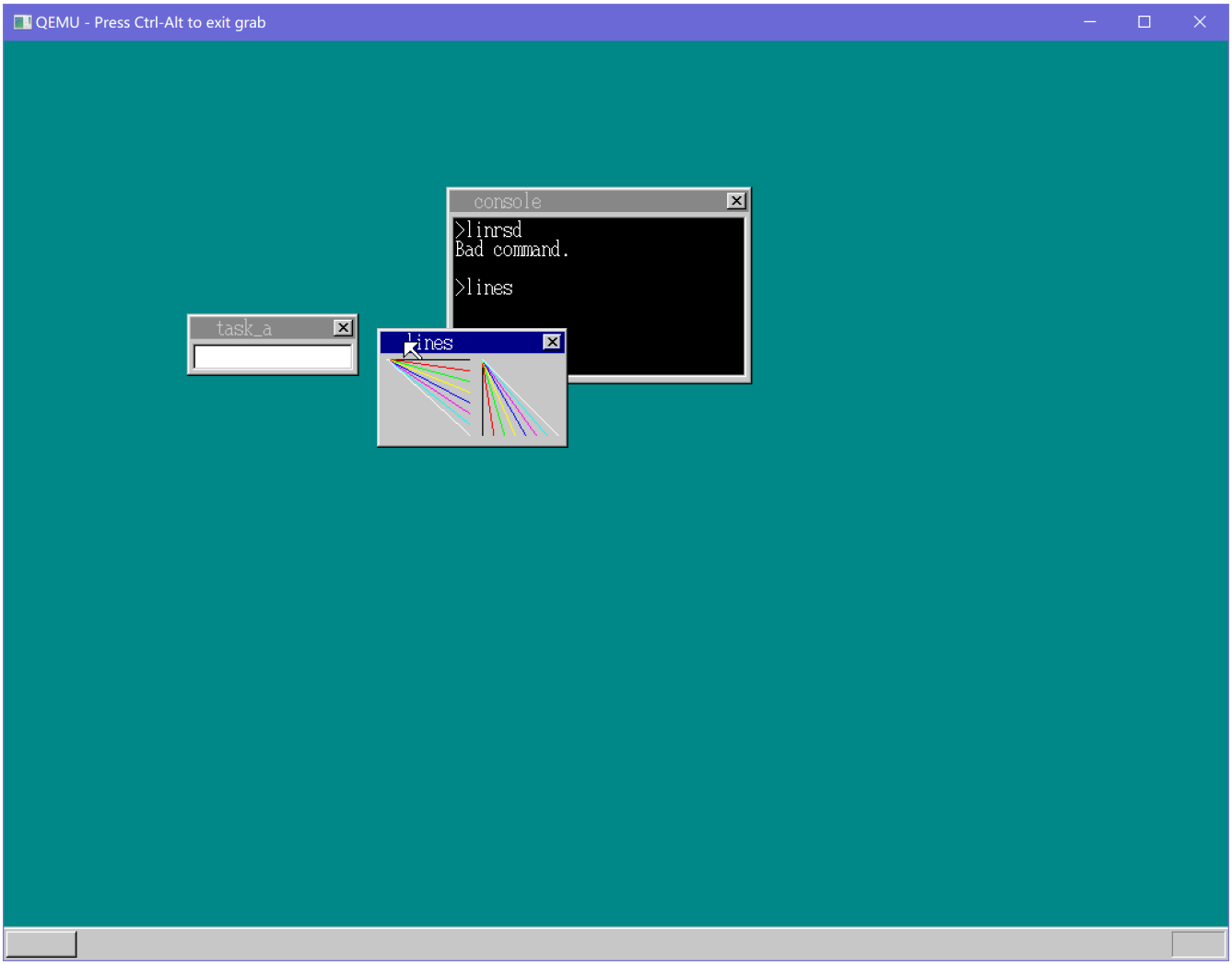
make run成功了



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;
}
```

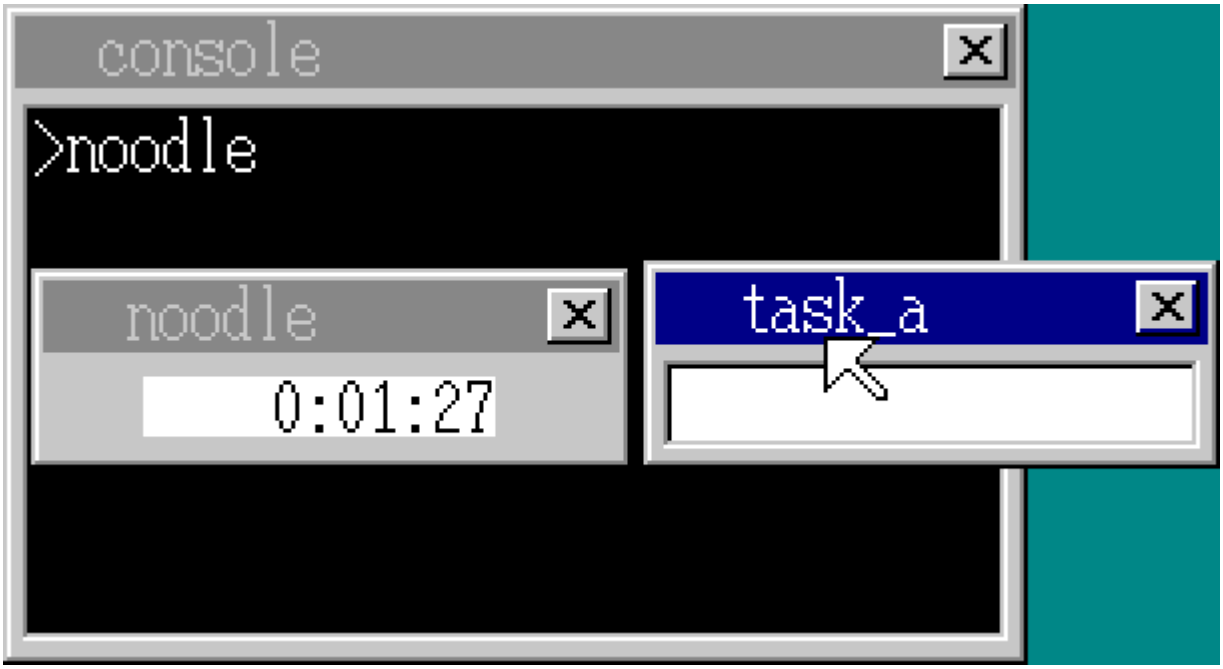
api定义和声明

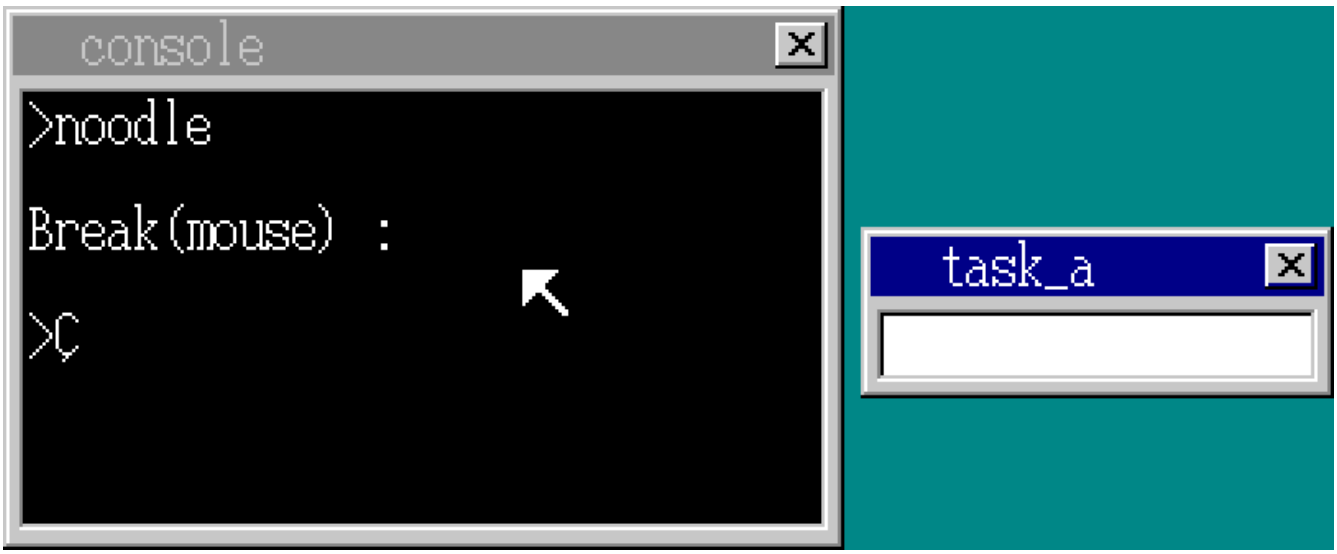
```
_api_alloctimer: ; int api_alloctimer(void);
        MOV EDX,16
        INT 0x40
        RET
_api_inittimer: ; void api_inittimer(int timer, int data);
        PUSH EBX
        MOV EDX,17
        MOV EBX,[ESP+ 8] ; timer
        MOV EAX,[ESP+12] ; data
        INT 0x40
        POP EBX
        RET
_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中出现了奇怪的字符。

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