褚逸豪 2017211123　17

Day 12

为我们的操作系统设计定时器。

定时器对于操作系统来说十分重要，在各种地方上我们都能看到定时器的应用，QQ登陆失败定时重试，PPPoE连接失败定时自动重新拨号，crontab定时执行任务。但这些定时器和我们要为操作系统造的定时器不太一样，我们造的更底层，对于操作系统来说也是不可或缺的。CPU可以利用我们的定时器来计时，有了时间的概念，我们才能造出其他的计时功能。

我们首先想到利用指令的周期数来计算到底过去多少时间，这样会有如下几个问题：

1. 每个CPU的主频各不相同，在不同的机器上运行，操作系统对时间的快慢的感受也不一样。
2. 无法使用HLT，HLT会打断计时

幸好，计算机上安装了PIC(Programmable Interval Timer)，他的主要功能是根据你的设定，每隔一段时间产生一个中断送到PIC上。既然我们能得到有规律的中断，那么我们就可以更方便的计时了。

首先我们要设定PIT，需要执行三次out指令

```
#define PIT_CTRL 0x0043
#define PIT_CNT0 0x0040
void init_pit(void)
{
    io_out8(PIT_CTRL, 0x34);
    io_out8(PIT_CNT0, 0x9c);    // 送中断周期的低8位
    io_out8(PIT_CNT0, 0x2e);    // 送中断周期的高8位
    return;
}
```

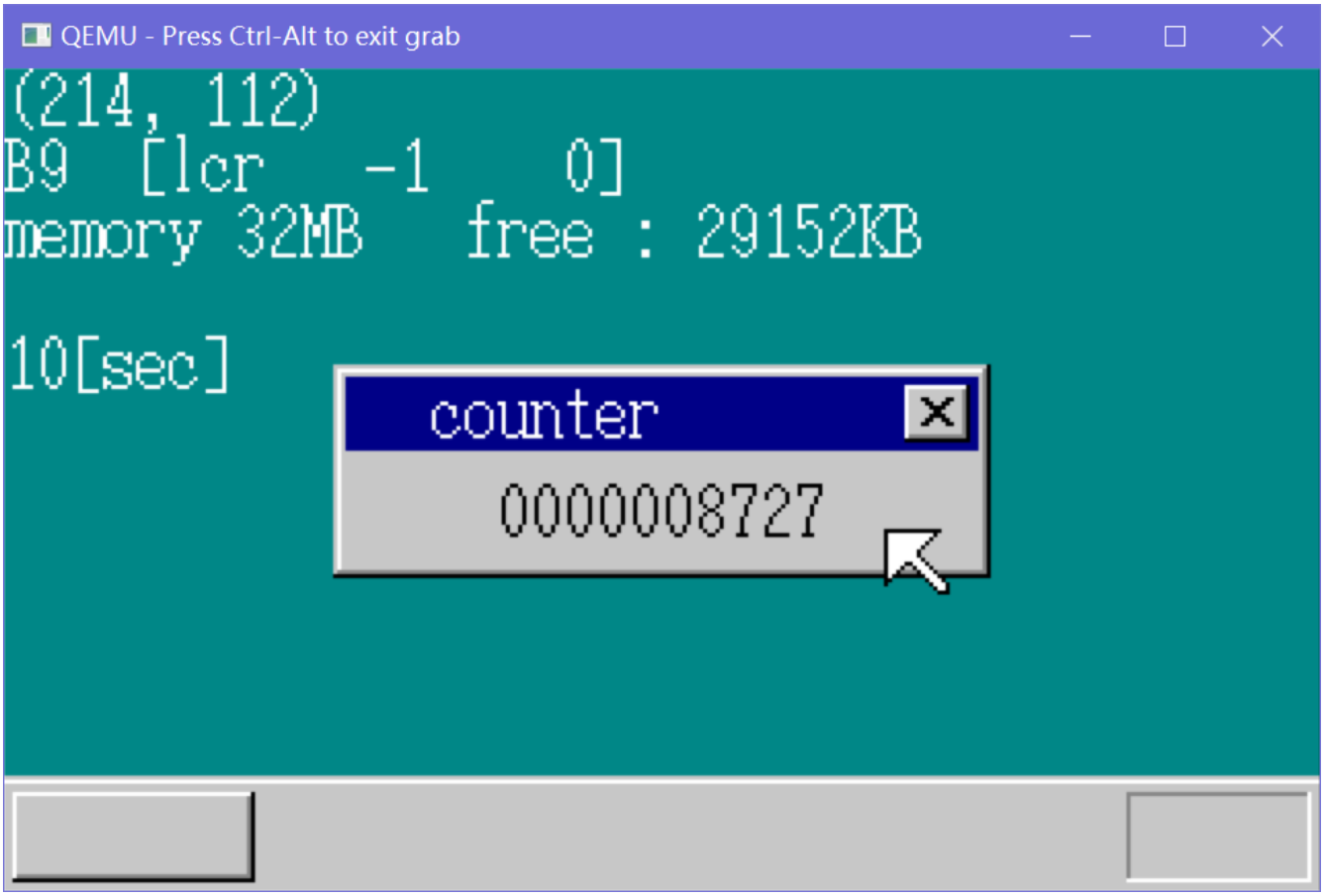
注意如果指定中断周期为0，会被看作是指定为65536。实际的中断频率是主频/设定的数值（看起来这还是与主频有关嘛，不知道作者打算怎么处理）

作者先设置了11932，为100Hz的频率，看起来作者的开发机是1193200Hz的主频呢。

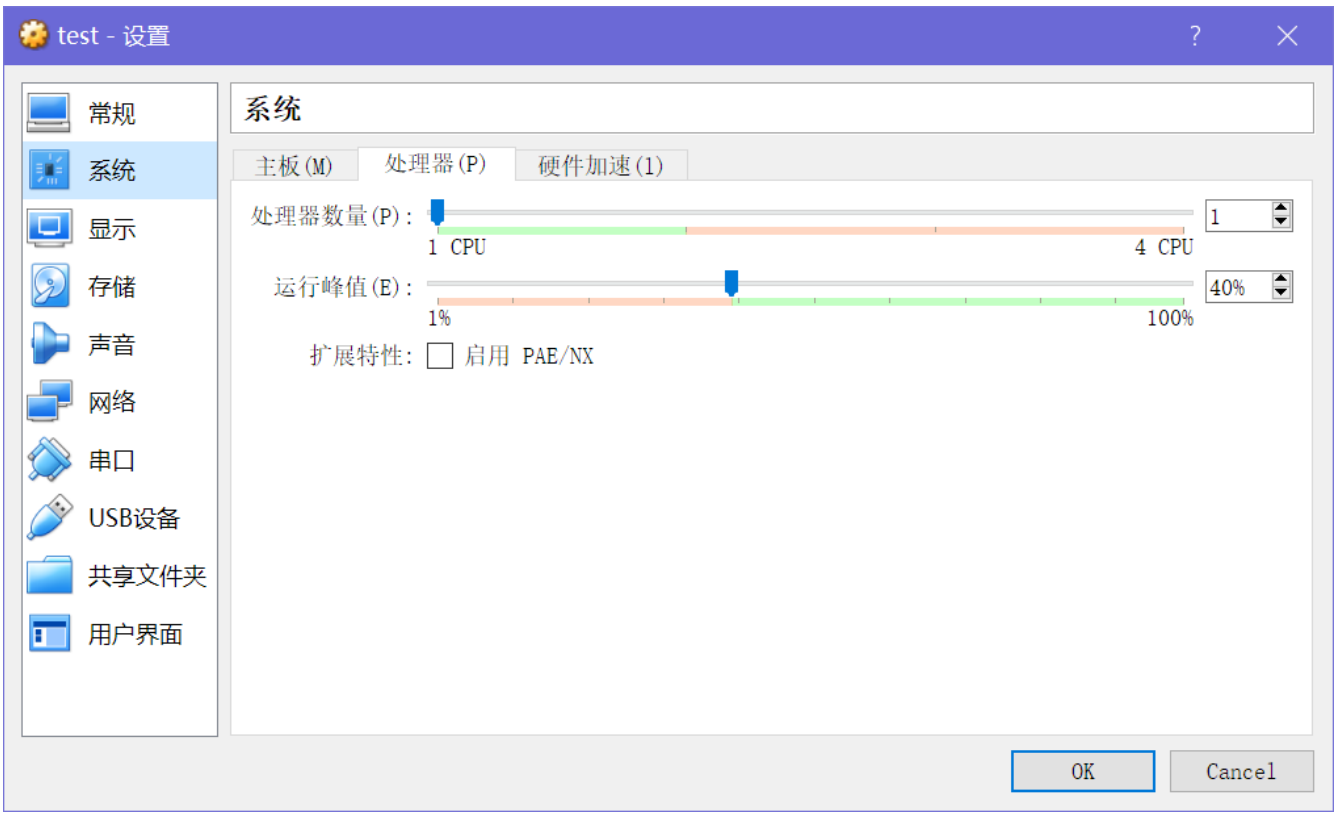
然后我们使用和之前相同的方式来编写中断程序（PIT的中断号是0，从数字上来看，这个PIT真的很重要呢）

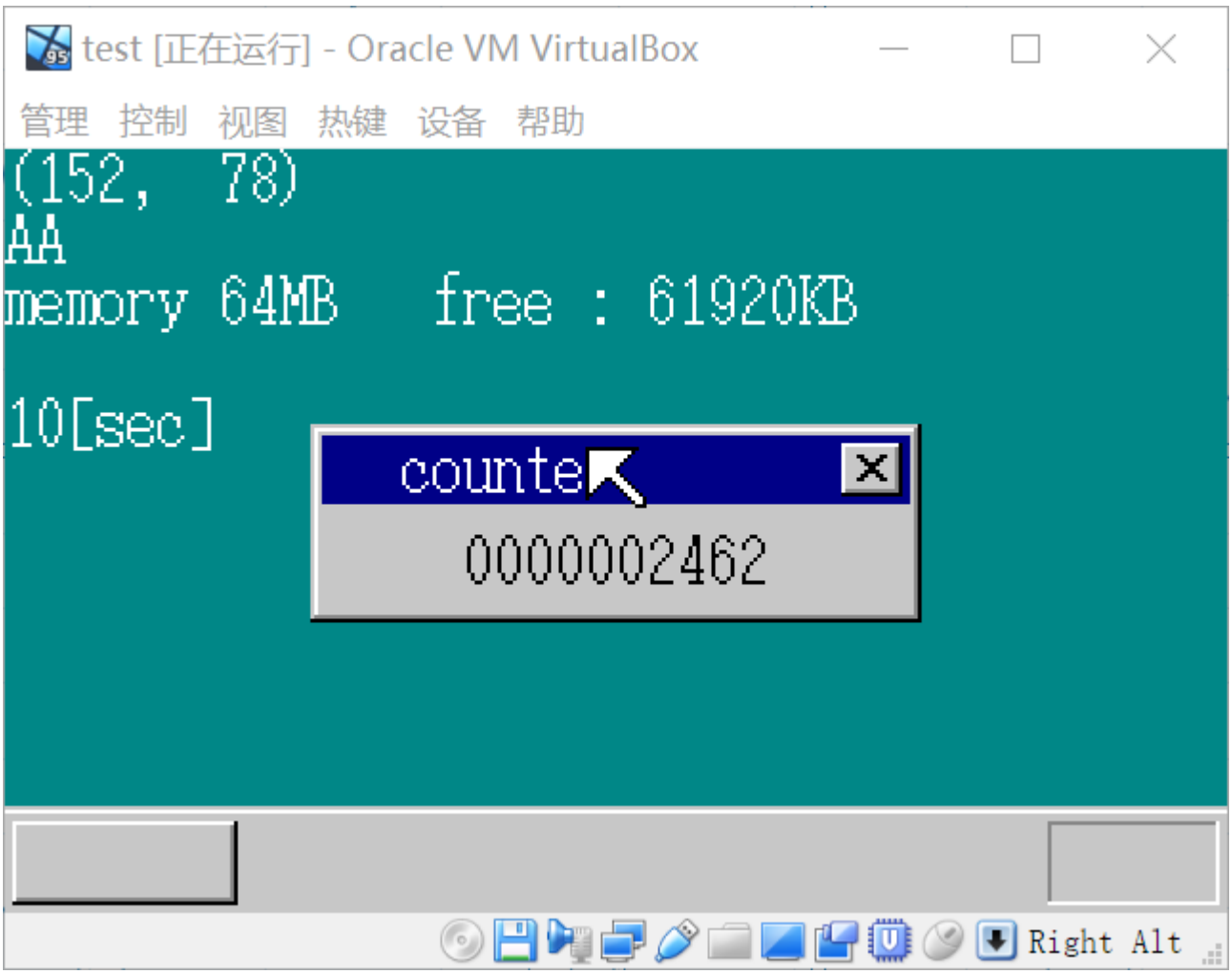
设置完中断之后，我们得让他做点什么，不然无法观察现象。

我们弄个计数器，在中断处理函数中为他添加自加一的命令，然后我们在桌布上打印计时器的数值。

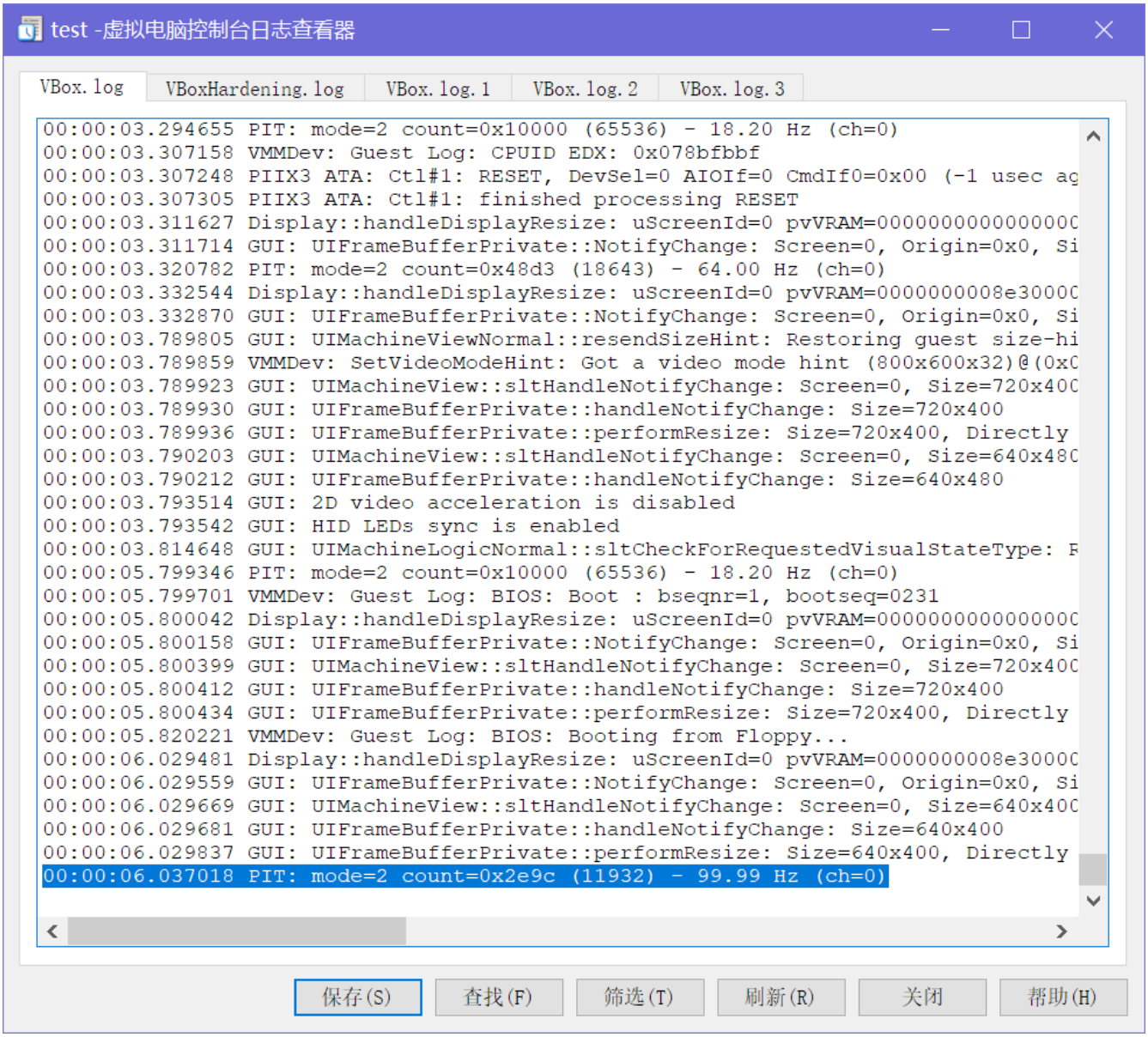


我有点小问题，如果我用virtualbox跑，修改主频，那计时器的速度会不会因此而改变呢?





我利用vbox将虚拟机占用主频限制在了40%，可以看到计时器的速度明显减慢。但是打开vbox的日志



我们找到了设置PIT所产生的日志，看来设置这个数值应该就是会产生100Hz的中断。

查阅了一些资料https://blog.csdn.net/axx1611/article/details/1786599

发现原来这个主频指的是PIT的主频，emmm，看来是我之前理解错了。不过作者说的不是很明白，锅有一半算他的!

然后我们来制作超时功能，主要思路是设定一个timeout值，每次来一个中断就自减1，如果到0了就将一个超时事件送入队列。

```
void inthandler20(int *esp)
{
    io_out8(PIC0_OCW2, 0x60);
    timerctl.count++;
    if (timerctl.timeout > 0) {
        timerctl.timeout--;
```

```
        if (timerctl.timeout == 0) {
            fifo8_put(timerctl.fifo, timerctl.data);
        }
    }
    return;
}

void settimer(unsigned int timeout, struct FIFO8 *fifo, unsigned char data)
{
    int eflags;
    eflags = io_load_eflags();
    io_cli();
    timerctl.timeout = timeout;
    timerctl.fifo = fifo;
    timerctl.data = data;
    io_store_eflags(eflags);
    return;
}
```

在settimer函数里，如果设定还没有完全结束IRQ0的中断就进来的话，会引起混乱，所以我们先禁止中断，然后完成设定，最后再把中断状态复原。有点像进程同步里的类原子操作

不过作者在这里为啥不`io_sti()`呢？不是说好要把中断状态复原嘛？

我们趁热打铁，设定多个计时器。计时器的作用很多，有些外设的状态并不是通过中断来告诉CPU去查询设备状态的，而是要通过定时器每隔一段时间去询问一下。

我们把timer搞成一个数组，再增添一些设定每个timer的函数就OK了，没啥特别的难度

我们应当允许设定每个timer的周期，timer的额外数据，存储timer产生事件的fifo队列。

我们应当编写申请timer和释放timer的函数

```
#define TIMER_FLAGS_ALLOC       1   // timer已经被分配出去了
#define TIMER_FLAGS_USING       2   // timer已经被激活了

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

void timer_free(struct TIMER *timer)
{
```

```
    timer->flags = 0;  // flag为0表示当前timer未被使用
    return;
}

void timer_init(struct TIMER *timer, struct FIFO8 *fifo, unsigned char data)
{
    timer->fifo = fifo;
    timer->data = data;
    return;
}

void timer_settime(struct TIMER *timer, unsigned int timeout)
{
    timer->timeout = timeout;
    timer->flags = TIMER_FLAGS_USING;
    return;
}
```

现在我们我们为不同的timer编写不同的事件，我们只要分别检查不同timer对应的fifo队列里面有没有东西就行了。哦对了，我们不要忘记先设置timer

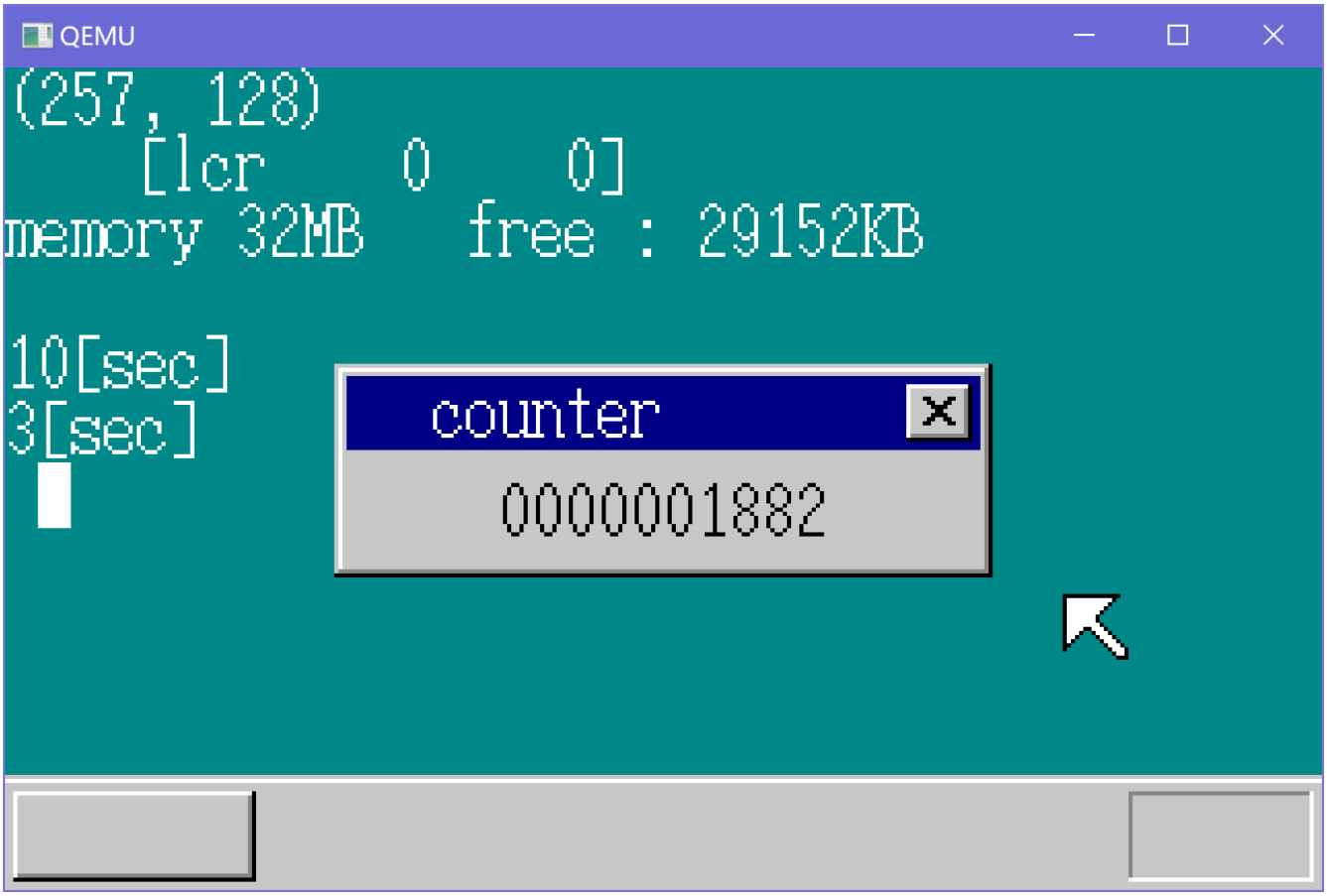
```
fifo8_init(&timerfifo, 8, timerbuf);
timer = timer_alloc();
timer_init(timer, &timerfifo, 1);
timer_settime(timer, 1000);
fifo8_init(&timerfifo2, 8, timerbuf2);
timer2 = timer_alloc();
timer_init(timer2, &timerfifo2, 1);
timer_settime(timer2, 300);
fifo8_init(&timerfifo3, 8, timerbuf3);
timer3 = timer_alloc();
timer_init(timer3, &timerfifo3, 1);
timer_settime(timer3, 50);
```

timer是个十秒的timer，

timer2是个三秒的timer

timer3是一个零点五秒的timer，用于光标闪烁。

跑一下！



看起来不错

之前作者实现的检查是否到时间有点小蠢，每次检查一个计数器都要自减以下，这样效率蛮低的，所以我们的第一个优化是将timeout改为目标时刻，这样我们就可以避免每次都做自减了，但是这个优化肯定不会很厉害，因为我们还是要经常检查每个timer。

第二个优化是记一下下次timeout会在多久之后产生，这样只要没到这个时刻，我们就不需要检查所有的timer，只在到达这个时刻之后进行检查并更新这个字段就好了。

第三个优化就是建立一个线性表，把所有活动的timer都加入进去，实际上这个优化对于学过数据结构的大学生来说很容易想到了。不过作者这里暂时还是在使用数组，其实改成链表会更好（我们之前在内存管理那里已经完成过一次改写，方法大同小异，不再赘述）

今天已经很困了，明天继续吧。

Day 13

在正式开始这一天之前，我们先归纳一下之前写的一些函数功能。

1. 字符串显示：我们可以将打印背景、打印文字、刷新图层合并成一个函数，这样可以提高我们的编码效率

```
    void putfonts8_asc_sht(struct SHEET *sht, int x, int y, int c, int b, char *s, int l)
    {
     boxfill8(sht->buf, sht->bxsize, b, x, y, x + l * 8 - 1, y + 15);
     putfonts8_asc(sht->buf, sht->bxsize, x, y, c, s);
     sheet_refresh(sht, x, y, x + l * 8, y + 16);
     return;
    }
/*
    x，y表示位置的坐标
    c表示字符的颜色
    b表示背景颜色
    s表示字符串
    l表示字符串长度
*/
```

2. 之前我们为每个timer都分配了一个队列，但实际上这样有点混乱，我们把他归纳到一个队列吧，这样申请新的定时器也比较方便。为了实现这样的改变，我们可以在fifo队列元素里附加一个字段，来指示是哪一个timer到时间了

搞完这些之后，我们尝试用timer来搞一些事情，我们来测定一下性能：先对HariMain略加修改，恢复变量count，然后完全不显示计数，全力执行`count++;`语句。当到了10秒后超时的时候，再显示这个count值。

如果最后显示的这个count越多，那么就说明在单位时间内我们执行了更多的主循环，我们的效率就越高。

为什么要在三秒的时候`count = 0;`呢？这是因为系统启动所耗费的时间并不稳定，每次可能相差很大，如果把这些时间计算在内，测试结果就不够准确，所以我们在第三秒清空一下count，这个时候启动相关的操作应该都执行完毕了，从第三秒开始计时应该不会过多影响我们的结果。

我们之前归纳了不同timer的fifo队列，我们把鼠标和键盘的fifo队列也归纳一下吧，连同定时器一起归纳到一个队列里面，这样我们进一步减少了每个主循环的代价，能在单位时间内循环更多次，count更加佳佳，性能更好。

我们用数据进行区分。

0~1 表示 光标闪烁
3 表示 3秒定时器
10 表示 10秒定时器
256~511 表示 键盘输入（键盘控制器读入的值再加上256）

512~767 表示 鼠标输入（键盘控制器读入的值再加上512）

这样的话，我们要用32位fifo队列。**所有的中断事件最后都要被送进一个32位队列里**

```
/* HariMain() */

void HariMain(void)
{
    struct FIFO32 fifo;
    int fifobuf[128];
    struct TIMER *timer, *timer2, *timer3;

    fifo32_init(&fifo, 128, fifobuf);

    init_keyboard(&fifo, 256);          // 第二个参数是送到fifo里时数据要加上的偏移量
    enable_mouse(&fifo, 512, &mdec);    // 同上
    timer = timer_alloc();
    timer_init(timer, &fifo, 10);
    timer_settime(timer, 1000);
    timer2 = timer_alloc();
    timer_init(timer2, &fifo, 3);
    timer_settime(timer2, 300);
    timer3 = timer_alloc();
    timer_init(timer3, &fifo, 1);
    timer_settime(timer3, 50);
    for (;;) {
        io_cli();
        if (fifo32_status(&fifo) == 0) {
            io_sti();
        } else {
            i = fifo32_get(&fifo);
            io_sti();

            if (256 <= i && i <= 511) { /* 键盘数据 */
            } else if (512 <= i && i <= 767) { /* 鼠标数据 */
            } else if (i == 10) { /* 10秒定时器 */
            } else if (i == 3) { /* 3秒定时器 */
            } else if (i == 1) { /* 光标定时器 */
            } else if (i == 0) { /* 光标定时器 */
            }
        }
    }
}
```

然后作者终于决定上链表了，感动！

还上了哨兵！牛逼！

链表的原理是我先找个老大，然后老大告诉我老二在哪，我去找老二，老二再告诉我老三在哪，我再去找老三，以此类推，如果我们想弄个老2.5排在老二老三之间，我们就告诉老2.5，你记住老三住在哪，然后在告诉老二，你别记老三了，记老2.5吧。开出一个人也是类似的，我们告诉老大别管老二了，管老三，老二就从链表中被删除出去了。

哨兵这个东西就像是一个永远不会被删除的人，不论一个链表是否为空，他都存在，并且永远是第一个元素，这样我们可以降低编码复杂度，减少很多判断。

```
void init_pit(void)
{
    int i;
    struct TIMER *t;
    io_out8(PIT_CTRL, 0x34);
    io_out8(PIT_CNT0, 0x9c);
    io_out8(PIT_CNT0, 0x2e);
    timerctl.count = 0;
    for (i = 0; i < MAX_TIMER; i++) {
        timerctl.timers0[i].flags = 0;
    }
    t = timer_alloc();
    t->timeout = 0xffffffff;
    t->flags = TIMER_FLAGS_USING;
    t->next = 0;
    timerctl.t0 = t;
    timerctl.next = 0xffffffff;
    return;
}
void timer_settime(struct TIMER *timer, unsigned int timeout)
{
    int e;
    struct TIMER *t, *s;
    timer->timeout = timeout + timerctl.count;
    timer->flags = TIMER_FLAGS_USING;
    e = io_load_eflags();
    io_cli();
    t = timerctl.t0;
    if (timer->timeout <= t->timeout) {
        timerctl.t0 = timer;
        timer->next = t;
        timerctl.next = timer->timeout;
        io_store_eflags(e);
        return;
    }
    for (;;) {
        s = t;
        t = t->next;
        if (timer->timeout <= t->timeout) {
            s->next = timer;
            timer->next = t;
            io_store_eflags(e);
            return;
        }
```

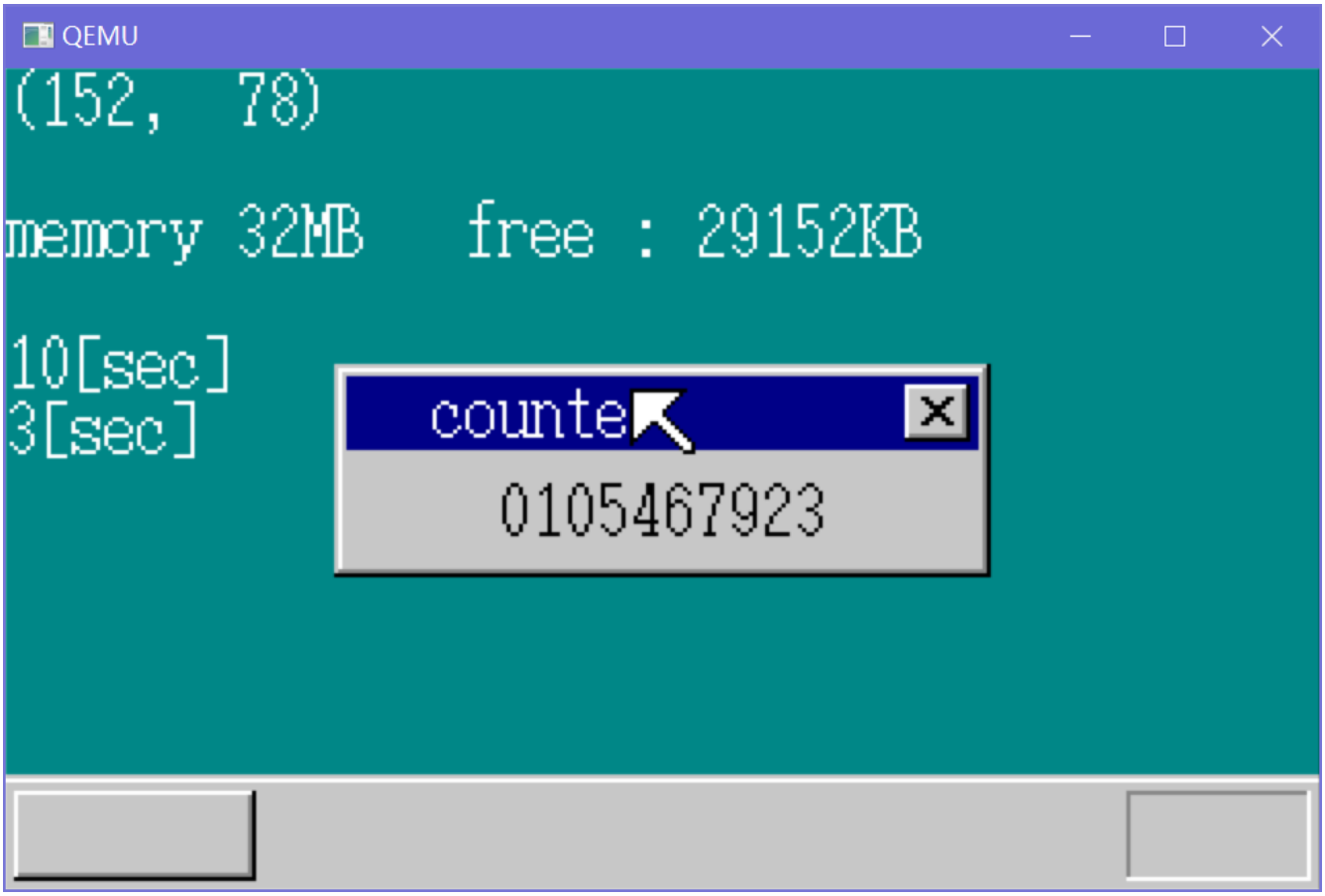
```
    }
}
void inthandler20(int *esp)
{
    struct TIMER *timer;
    io_out8(PIC0_OCW2, 0x60);
    timerctl.count++;
    if (timerctl.next > timerctl.count) {
        return;
    }
    timer = timerctl.t0;
    for (;;) {
        if (timer->timeout > timerctl.count) {
            break;
        }
        timer->flags = TIMER_FLAGS_ALLOC;
        fifo32_put(timer->fifo, timer->data);
        timer = timer->next;
    }

    timerctl.t0 = timer;
    timerctl.next = timer->timeout;

    return;
}
```

今天就到这吧（说起来今天没啥十分特别的东西，主要是归纳了一个函数几个队列，然后进行了我们大家一直都很想做的链表优化而已）

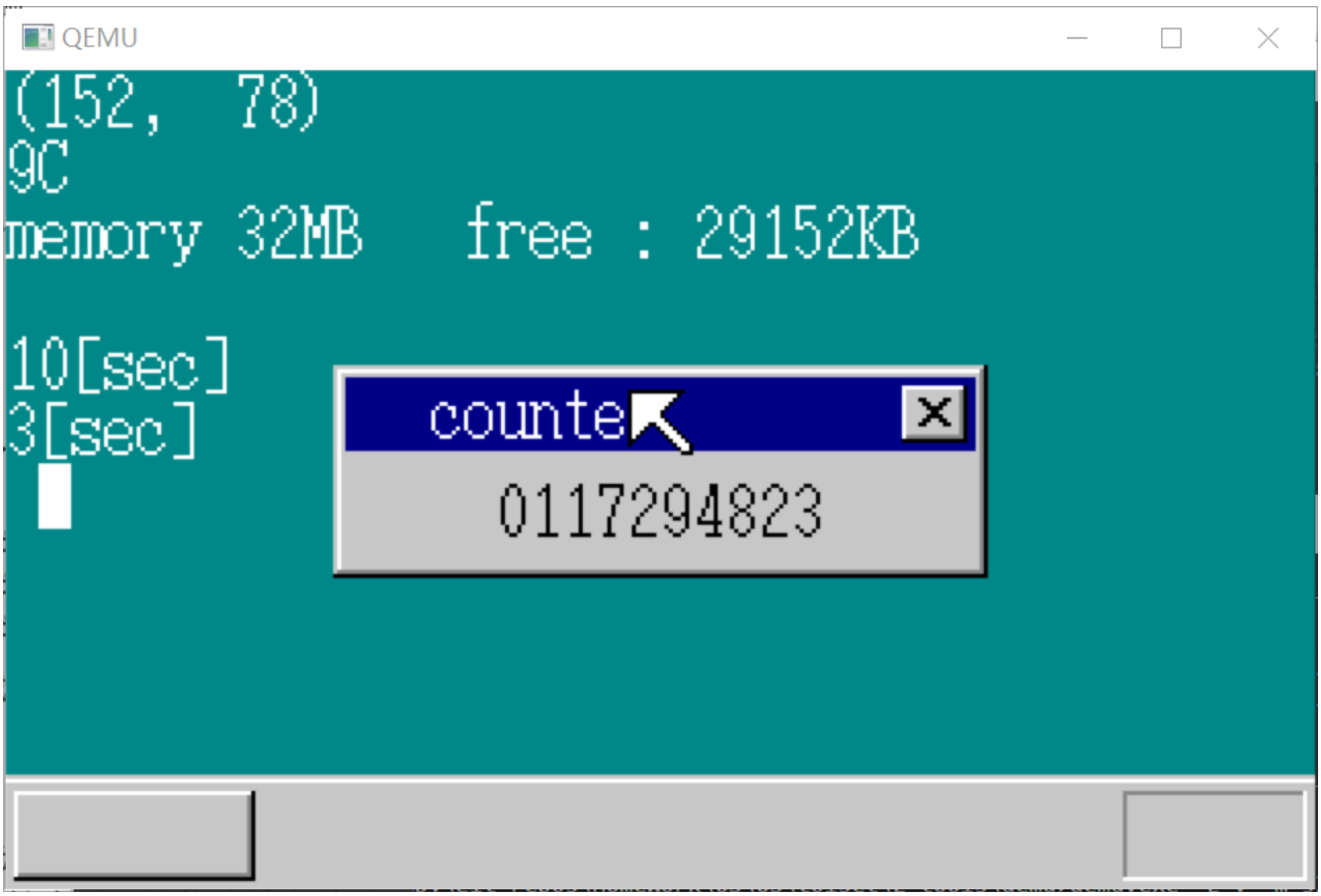
Day 14

今天我们做的第一件事情是测试性能

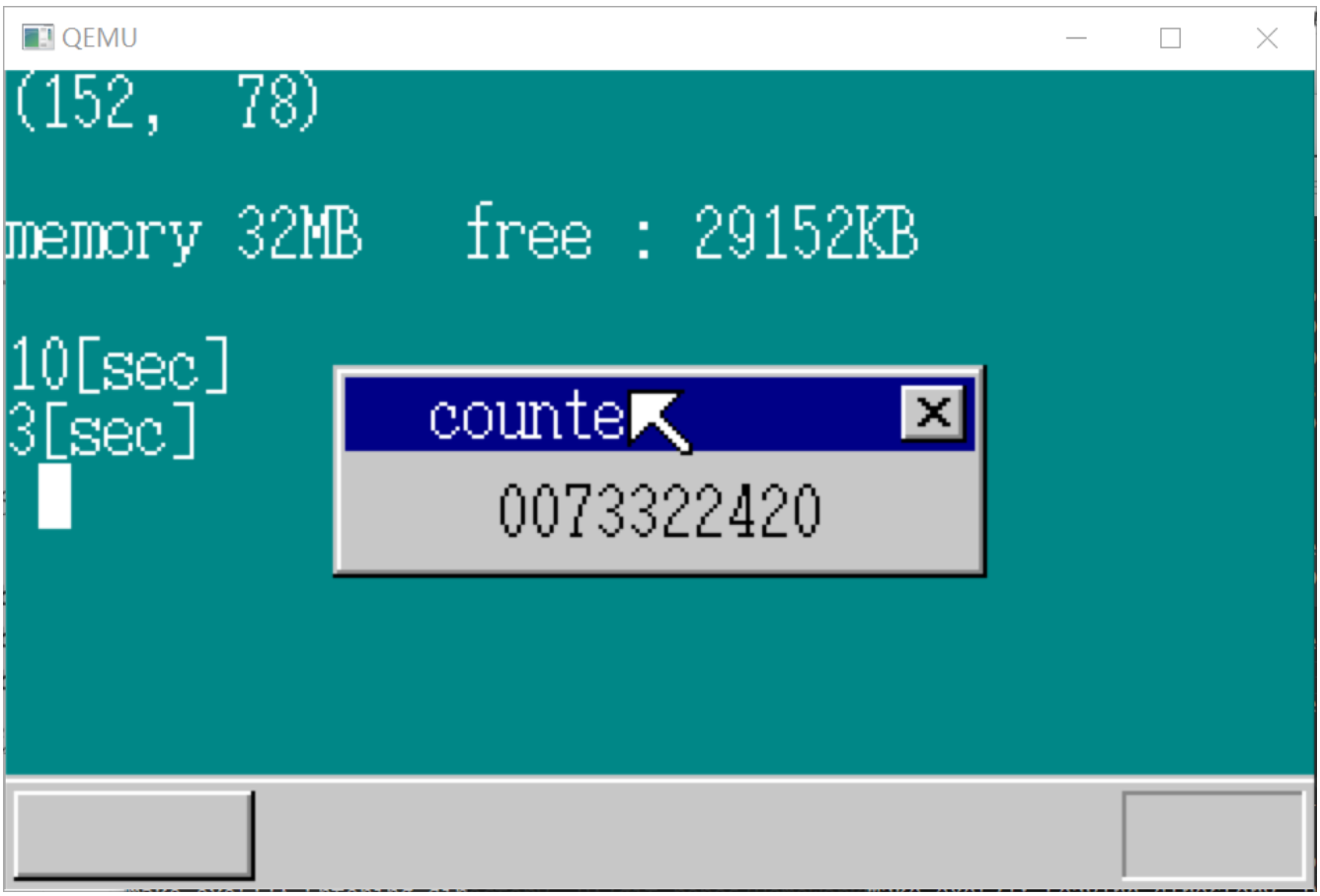
A 追加490个定时器，有移位



B 追加490个定时器，链表，没有哨兵



C 追加490个定时器，链表，没有哨兵



emmm，这跟作者的实机测试结果来比，哪怕三个数据的大小关系都不一样，看来系统上正在运行的其他应用对运行结果的影响蛮大的。

没法跑真机，我们还是暂时忘了他吧。

然后还有一个玄学jmp问题，前面加set490以后各个指令的地址也都会相应地错开几个字节，结果造成JMP指令的地址也略有变化，因此执行时间也稍稍延迟。

然后我们来提高我们操作系统的分辨率吧，在2k屏、3k屏上看320x200的画面真是有点别扭。不同的显卡，高分辨率利用方法也不太一样，我们先只考虑如何在QEMU中使用高分辨率

```
MOV BX,0x4101 ; VBE的640x480x8bit彩色
MOV AX,0x4f02
INT 0x10
MOV BYTE [VMODE],8 ; 记下画面模式（参考C语言）
MOV WORD [SCRNX],640
MOV WORD [SCRNY],480
MOV DWORD [VRAM],0xe000000
```

由于曾经的显卡种类众多，各家标准也不太相同，设定起来不一样，混乱而复杂，为了解决这样的问题，显卡公司们成立了VESA协会，制定了基础通用的设定方法。我们按照VESA指定的格式就可以开启高分辨率模式了。

但是有的公司并没有加入VESA，这种设定方法也不一定能够使用，我们最好还是先检查一下VBE是否存在以及他的版本。

```
; 确认VBE是否存在
MOV AX,0x9000
MOV ES,AX
MOV DI,0
MOV AX,0x4f00
INT 0x10
CMP AX,0x004f
JNE scrn320
```

```
; 检查VBE的版本
MOV AX,[ES:DI+4]
CMP AX,0x0200
JB scrn320 ; if (AX < 0x0200) goto scrn320
```

```
; 取得画面模式信息
MOV CX,VBEMODE
MOV AX,0x4f01
INT 0x10
CMP AX,0x004f
JNE scrn320
```

类型	位置	备注
WORD	[ES : DI+0x00]	模式属性.....bit7 不是 1 就不好办(能加上 0x4000)
WORD	[ES : DI+0x12]	X 的分辨率
WORD	[ES : DI+0x14]	Y 的分辨率
BYTE	[ES : DI+0x19]	颜色数.....必须为 8
BYTE	[ES : DI+0x1b]	颜色的指定方法.....必须为 4 (4 是调色板模式)
DWORD	[ES : DI+0x28]	VRAM 的地址

获取完画面模式信息之后，我们要检查一下他是否支持我们要选用的高分辨率的模式

```
; 画面模式信息的确认
CMP BYTE [ES:DI+0x19],8
JNE scrn320
CMP BYTE [ES:DI+0x1b],4
JNE scrn320
MOV AX,[ES:DI+0x00]
AND AX,0x0080
JZ scrn320 ; 模式属性的bit7是0，所以放
```

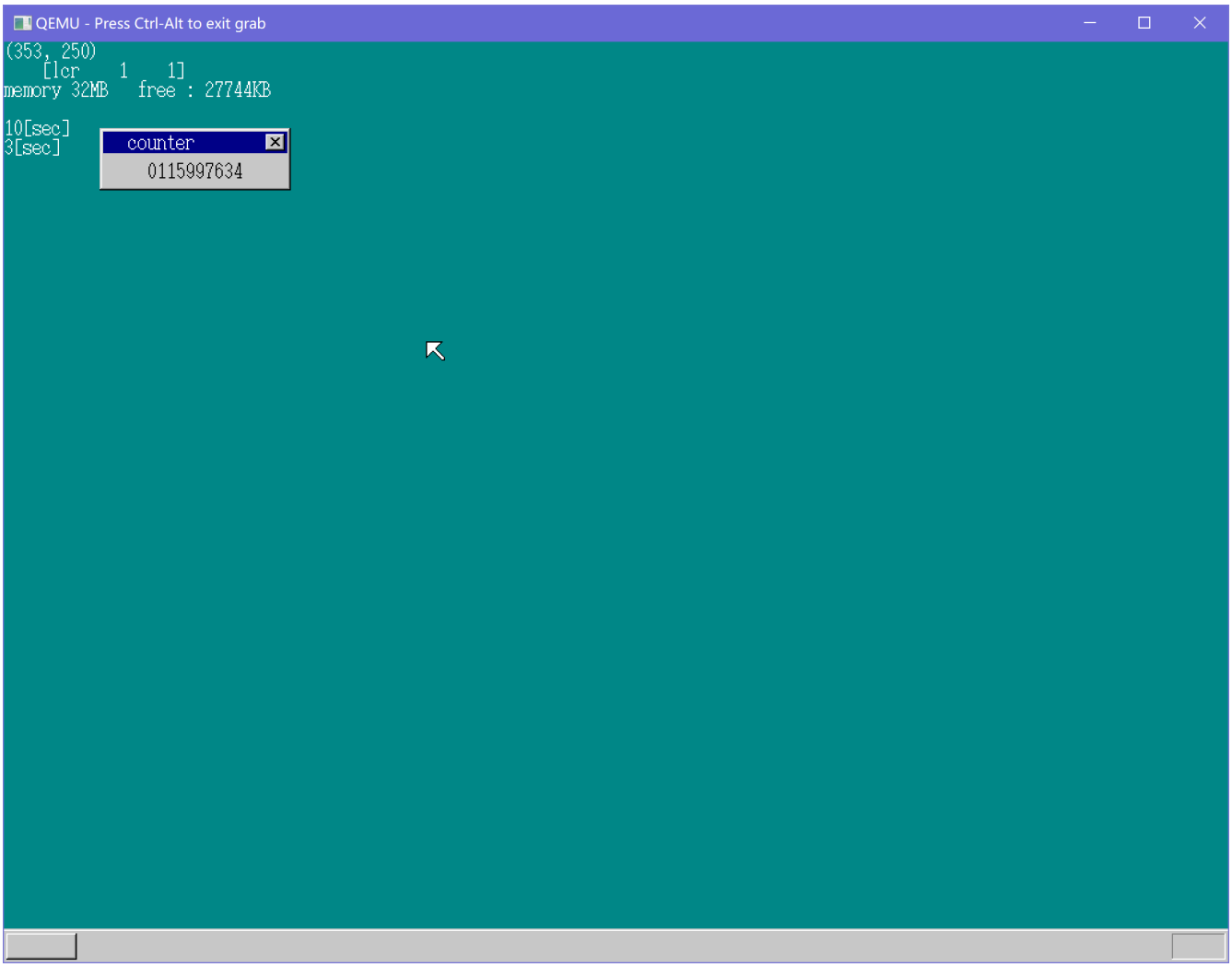
做完这些还不够，记得我们之前进入32位模式的之前我们做了什么嘛，我们把画面的相关信息存入了bootinfo中，我们这次也要这么做

```
; 画面模式的切换
MOV BX,VBEMODE+0x4000
MOV AX,0x4f02
INT 0x10
MOV BYTE [VMODE],8 ; 记下画面模式（参考C语言）
MOV AX,[ES:DI+0x12]
MOV [SCRNX],AX
MOV AX,[ES:DI+0x14]
MOV [SCRNY],AX
MOV EAX,[ES:DI+0x28]
MOV [VRAM],EAX
JMP keystatus
```

如果任意一个地方有问题，我们就要将就使用320x200画面了

```
scrn320:
MOV AL,0x13 ; VGA图、320x200x8bit彩色
MOV AH,0x00
INT 0x10
MOV BYTE [VMODE],8 ; 记下画面模式（参考C语言）
MOV WORD [SCRNX],320
MOV WORD [SCRNY],200
MOV DWORD [VRAM],0x000a0000
```

`make run`一下



键盘输入

有如下的表格

按下键时的数值表

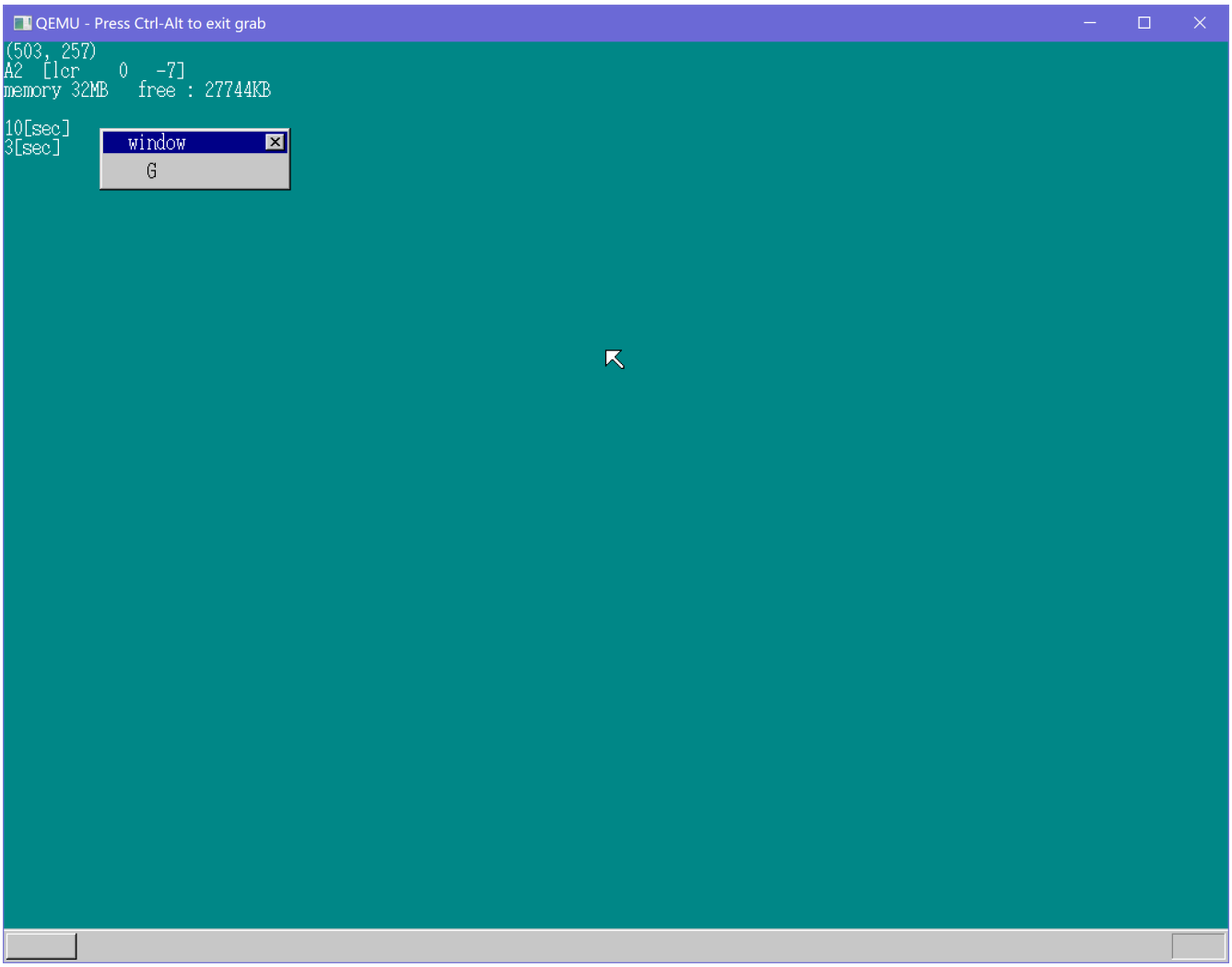
00：没有被分配使用	20：D	40：F6	60：保留
01：ESC	21：F	41：F7	61：保留？
02：主键盘的1	22：G	42：F8	62：保留？
03：主键盘的2	23：H	43：F9	63：保留？
04：主键盘的3	24：J	44：F10	64：保留？
05：主键盘的4	25：K	45：NumLock	65：保留？
06：主键盘的5	26：L	46：ScrollLock	66：保留？
07：主键盘的6	27：;	47：数字键的7	67：保留？
08：主键盘的7	28：:（在英语键盘是'）	48：数字键的8	68：保留？
09：主键盘的8	29：全角・半角（在英语键盘是`）	49：数字键的9	69：保留？
0A：主键盘的9	2A：左Shift	4A：数字键的-	6A：保留？
0B：主键盘的0	2B：]（在英语键盘是backslash（反斜线））	4B：数字键的4	6B：保留？
0C：主键盘的-	2C：Z	4C：数字键的5	6C：保留？
0D：主键盘的^（在英语键盘是=）	2D：X	4D：数字键的6	6D：保留？
0E：退格键	2E：C	4E：数字键的+	6E：保留？
0F：TAB键	2F：V	4F：数字键的1	6F：保留？
10：Q	30：B	50：数字键的2	70：平假名（日文键盘）
11：W	31：N	51：数字键的3	71：保留？
12：E	32：M	52：数字键的0	72：保留？
13：R	33：,	53：数字键的.	73：_
14：T	34：.	54：SysReq	74：保留？
15：Y	35：/	55：保留？	75：保留？
16：U	36：右Shift	56：保留？	76：保留？
17：I	37：数字键的*	57：F11	77：保留？
18：O	38：左Alt	58：F12	78：保留？
19：P	39：Space	59：保留？	79：变换（日文键盘）
1A：@（在英语键盘是[）	3A：CapsLock	5A：保留？	7A：保留？
1B：[（在英语键盘是]）	3B：F1	5B：保留？	7B：无变换（日文键盘）
1C：主键盘的Enter	3C：F2	5C：保留？	7C：保留？
1D：左Ctrl	3D：F3	5D：保留？	7D：\
1E：A	3E：F4	5E：保留？	7E：保留？
1F：S	3F：F5	5F：保留？	7F：保留？

按键弹起的数值是按下的数值加上0x80

根据这张表，我们可以将显示的内容从之前的一个字节十六进制数变为可读的。

看起来这张表和我们用的标准美式键盘不太一样，所以我在作者的基础上稍微修改了几个字符，不过修改的还不完全。

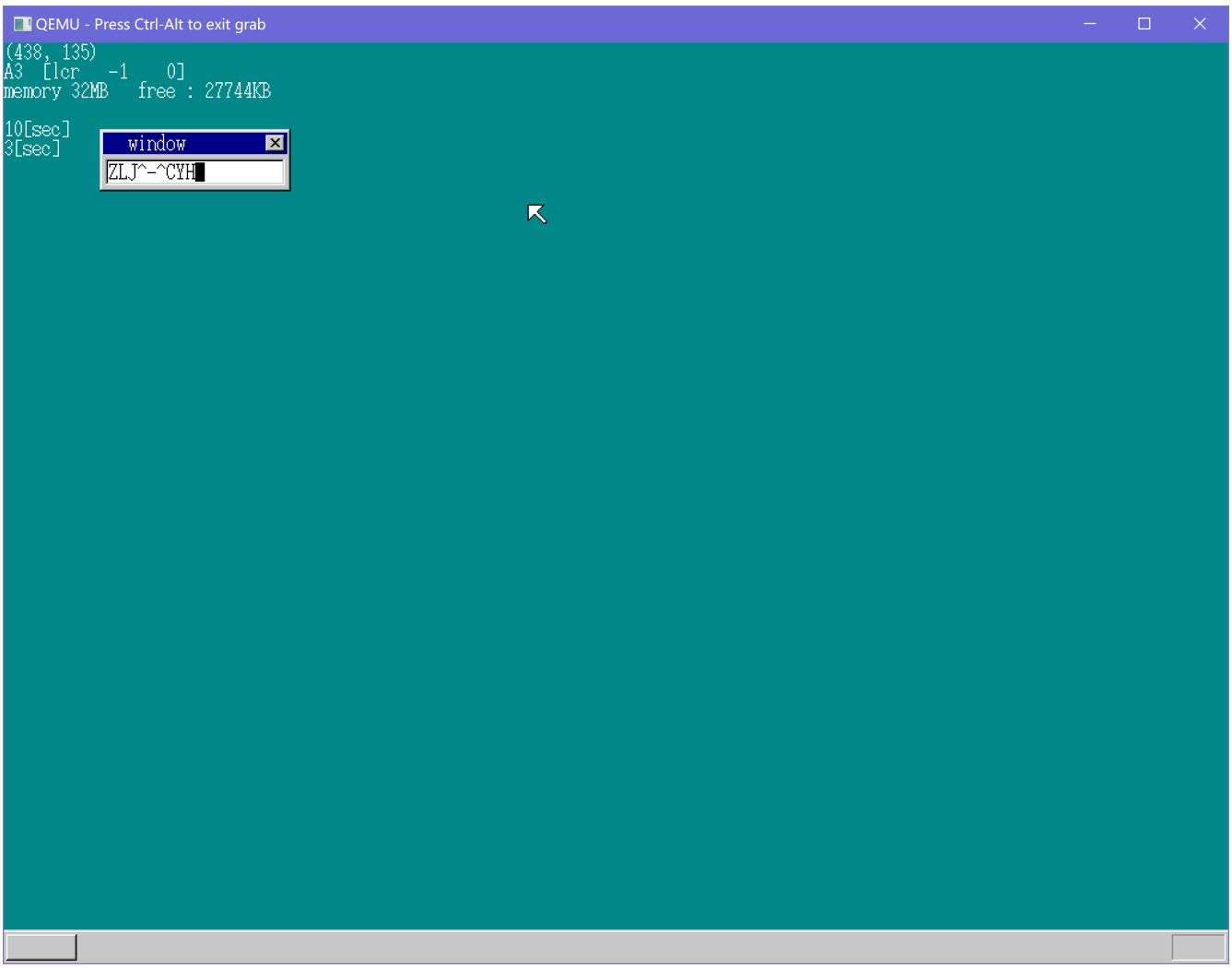
```
static char keytable[0x54] = {
        0,   0,   '1', '2', '3', '4', '5', '6', '7', '8', '9', '0', '-', '^', 0,   0,
        'Q', 'W', 'E', 'R', 'T', 'Y', 'U', 'I', 'O', 'P', '@', '[', 0,   0,   'A', 'S',
        'D', 'F', 'G', 'H', 'J', 'K', 'L', ';', ':', 0,   0,   ']', 'Z', 'X', 'C', 'V',
        'B', 'N', 'M', ',', '.', '/', 0,   '*', 0,   ' ', 0,   0,   0,   0,   0,   0,
        0,   0,   0,   0,   0,   0,   0,   '7', '8', '9', '-', '4', '5', '6', '+', '1',
        '2', '3', '0', '.'
    };
```



我们还可以以此来模仿一个简单的输入框，它的主要原理是遇到按键光标和打印位置都向右移动8个像素，按退格的时候则各像左移动8个像素

```
if (256 <= i && i <= 511) {
                sprintf(s, "%02X", i - 256);
                putfonts8_asc_sht(sht_back, 0, 16, COL8_FFFFFF, COL8_008484, s, 2);
                if (i < 0x54 + 256) {
                    if (keytable[i - 256] != 0 && cursor_x < 144) {
                        s[0] = keytable[i - 256];
                        s[1] = 0;
```

```
                        putfonts8_asc_sht(sht_win, cursor_x, 28, COL8_000000, COL8_FFFFFF,
s, 1);
                        cursor_x += 8; // 输入处理
                    }
                }
                if (i == 256 + 0x0e && cursor_x > 8) {
                    putfonts8_asc_sht(sht_win, cursor_x, 28, COL8_000000, COL8_FFFFFF, " ",
1);
                    cursor_x -= 8;
                    // 退格处理
                }
                boxfill8(sht_win->buf, sht_win->bxsize, cursor_c, cursor_x, 28, cursor_x +
7, 43);
                sheet_refresh(sht_win, cursor_x, 28, cursor_x + 8, 44);
            }
```



然后我们制作窗体移动的功能，其实比较简单，我们只要修改sheet左上角的坐标就好了，让他为鼠标点击位置减去一个固定的向量。

```
if ((mdec.btn & 0x01) != 0) {
    sheet_slide(sht_win, mx - 80, my - 8);
}
```

ok!