# 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;
}
```

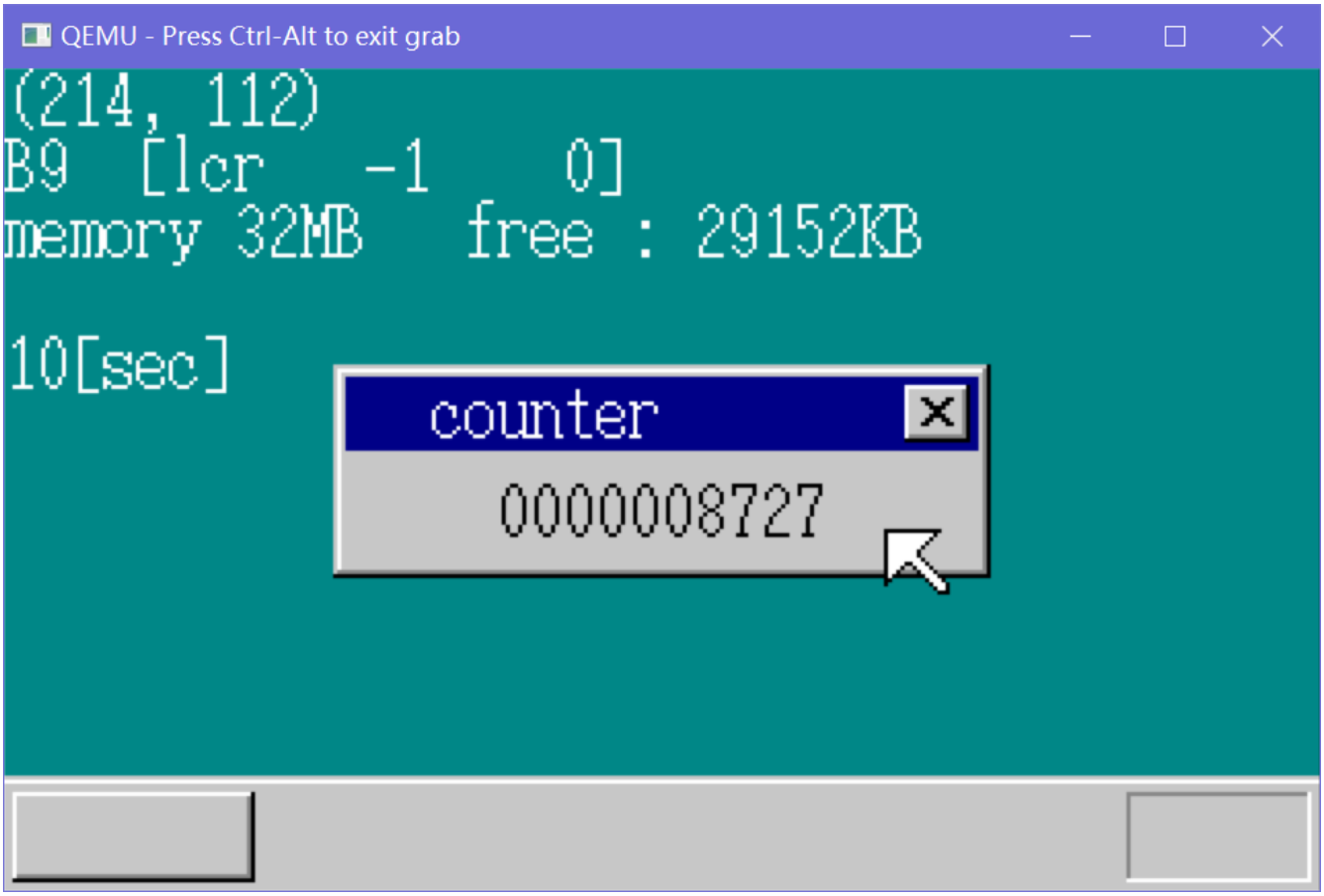
注意如果指定中断周期为0，会被看作是指定为65536。实际的中断频率是主频/设定的数值（看起来这还是与主频有关嘛，不知道作者打算怎么处理）

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。

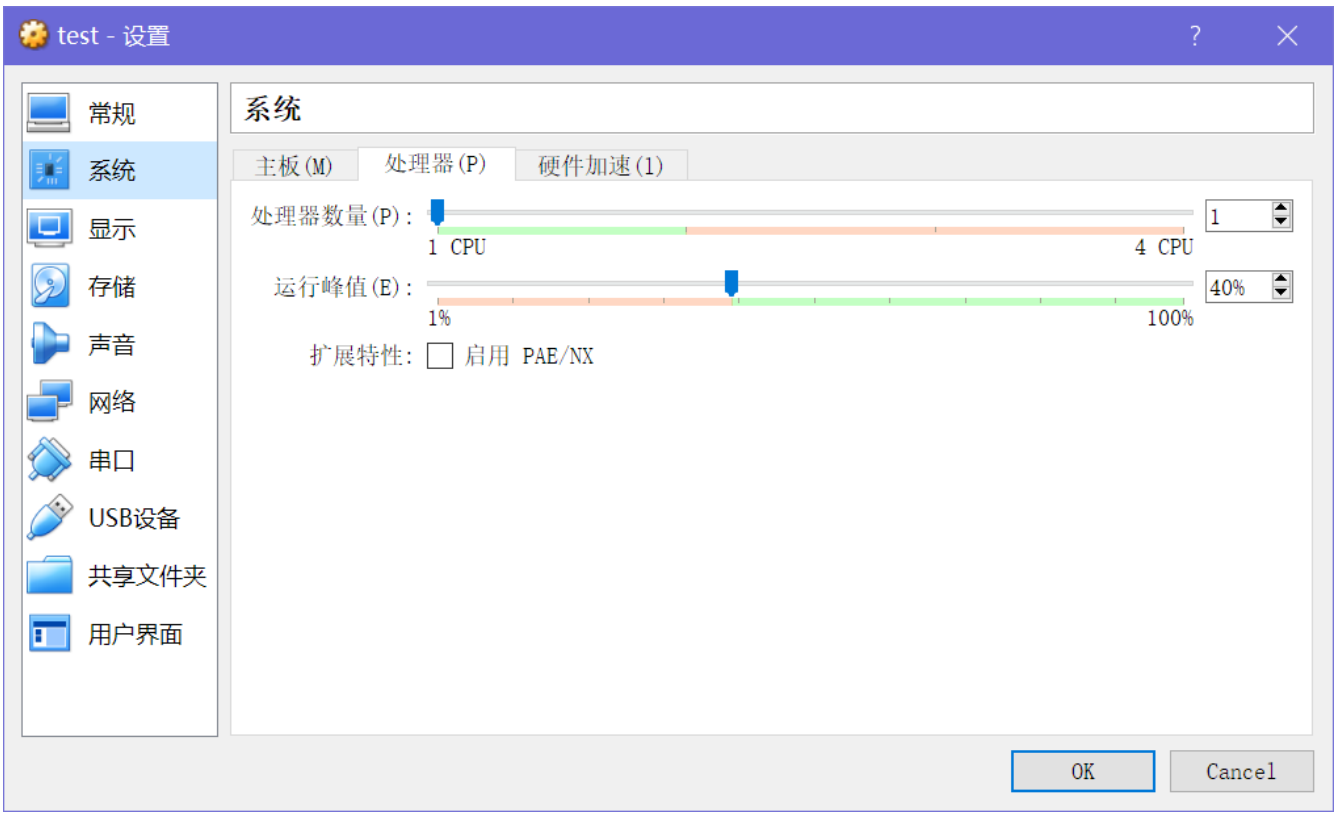
然后我们使用和之前相同的方式来编写中断程序（PIT的中断号是0，从数字上来看，这个PIT真的很重要呢）

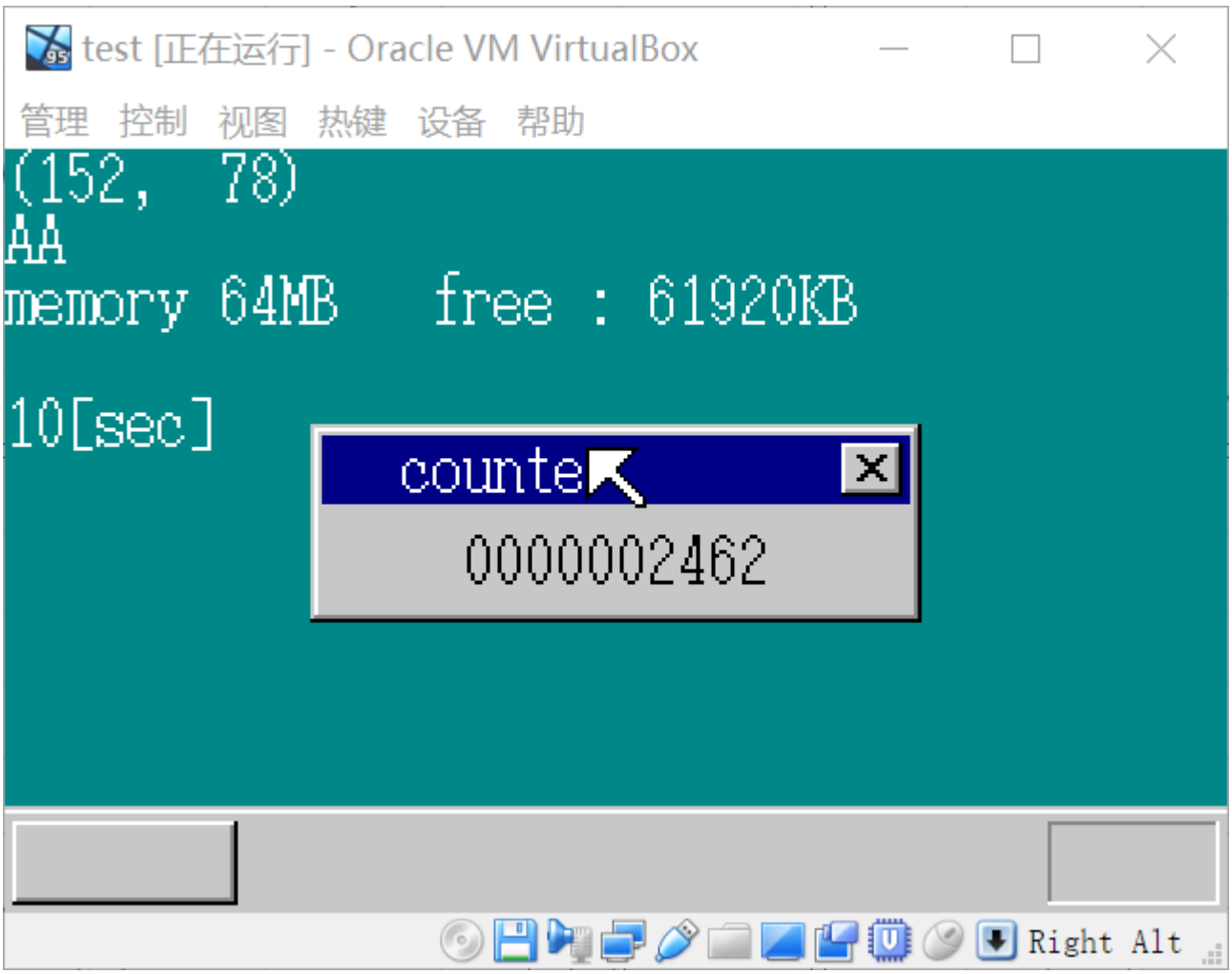
设置完中断之后，我们得让他做点什么，不然无法观察现象。

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。

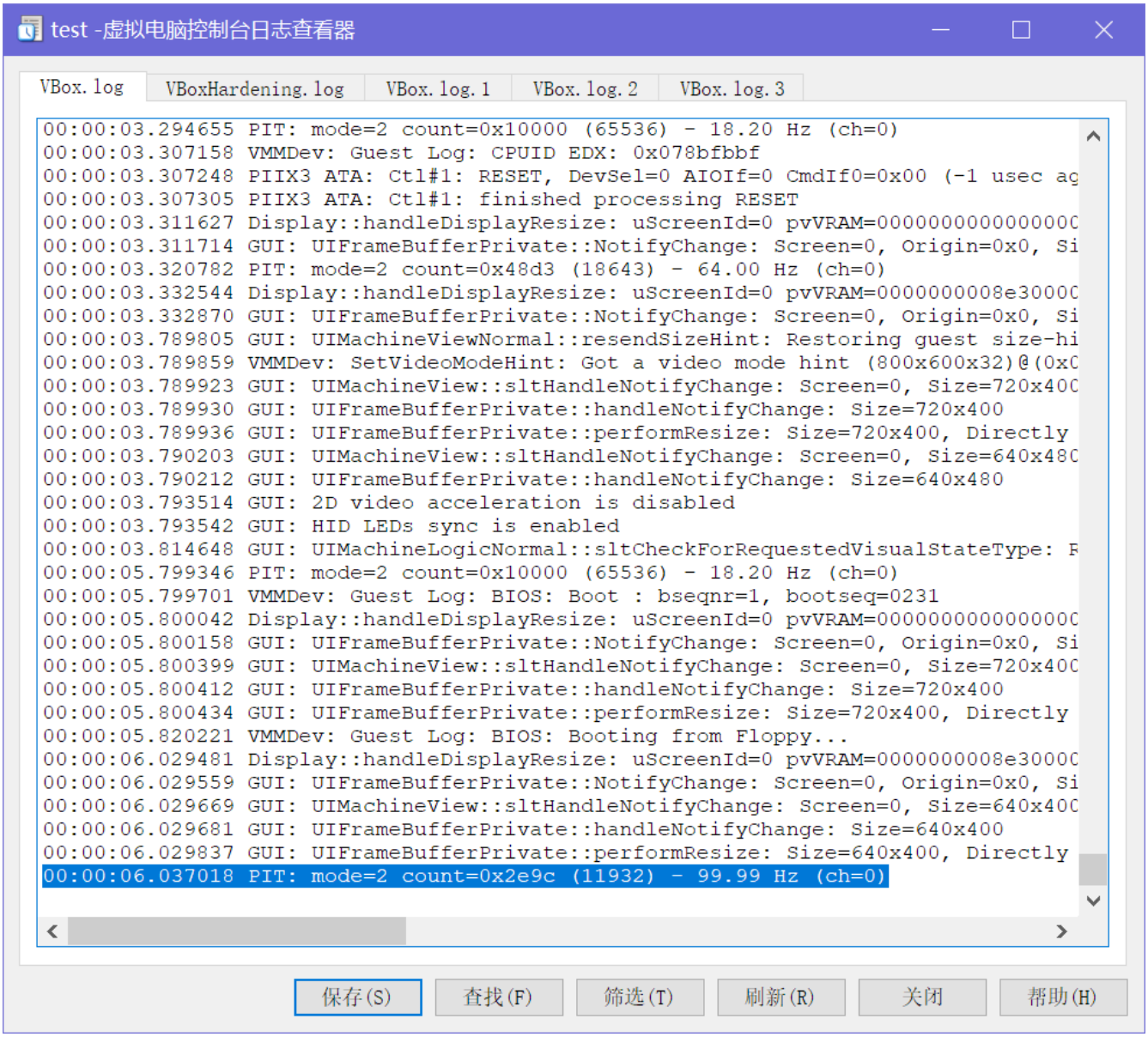



我有点小问题，如果我用virtualbox跑，修改主频，那计时器的速度会不会因此而改变呢?






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

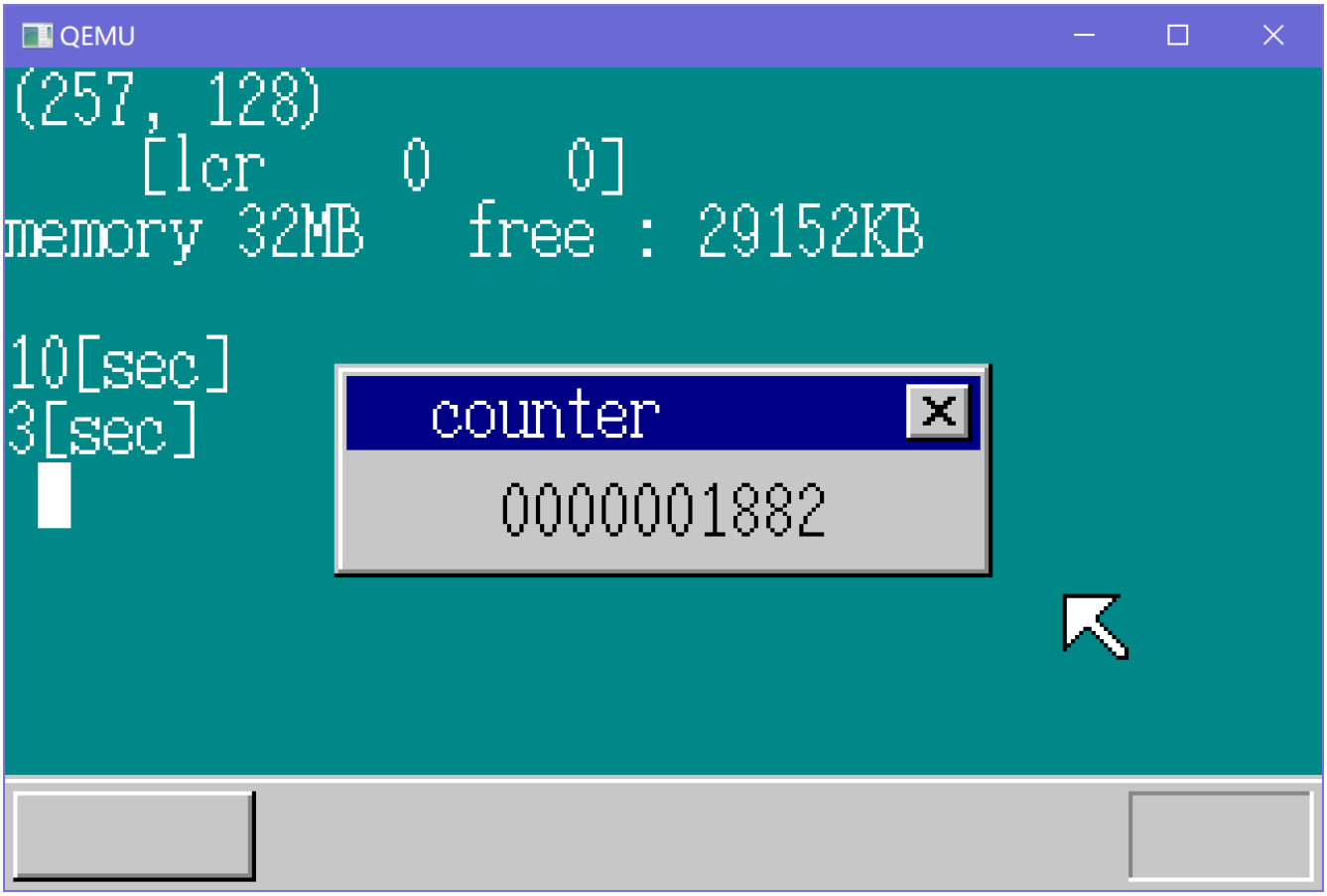
```
fifo8_init(&timerfifo, 8, timerbuf);
timer = timer_alloc();
timer_init(timer, &timerfifo, 1);
timer_settime(timer, 1000);
fifo8_init(&timerfifo2, 8, timerbuf2);
timer2 = timer_alloc();
timer_init(timer2, &timerfifo2, 1);
timer_settime(timer2, 300);
fifo8_init(&timerfifo3, 8, timerbuf3);
timer3 =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