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）