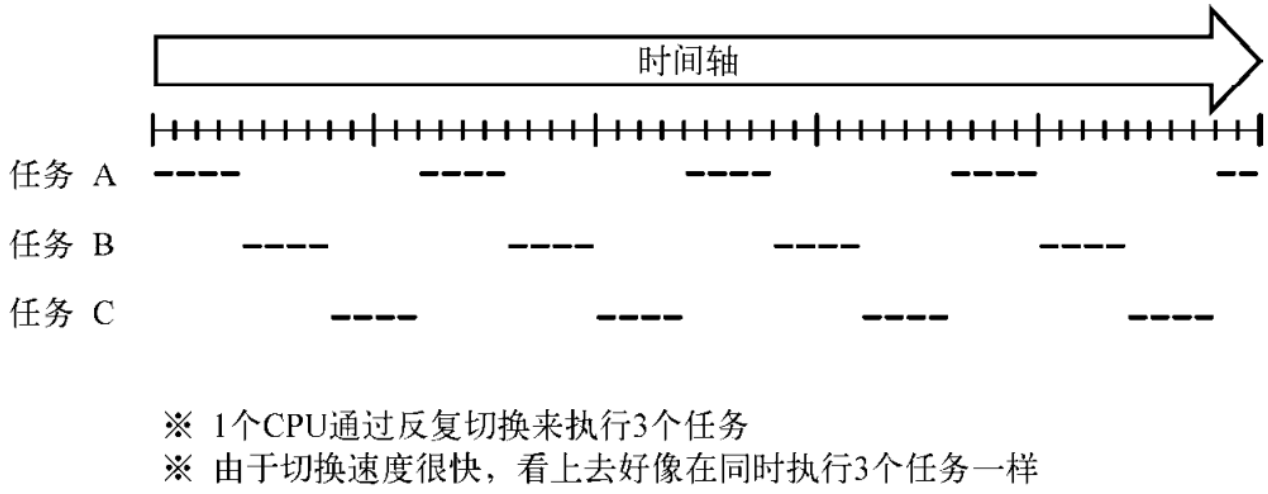
褚逸豪 2017211123 17

Day 15

多任务

结合我们在理论课程中的指示，我们知道多任务是如何实现的。通过不断地把CPU在不同任务之间调度，看起来就像在同时执行一样。



从某一个具体的瞬间来看，只有一个任务正在执行，但从一段时间来看，多个任务在同时运行。

为了使人无法觉察，我们不能给一个任务分配连续过长的时间片，而让别的任务忍饥挨饿。

在任务之间切换的时候会进行上下文的切换，保存PC以及寄存器值等等，这也是需要时间的，如果我们切换的过于频繁，那么大量时间将会被浪费在切换上，从而使程序的效率大幅度降低。

那我们如何进行任务的切换呢？实际机器上不需要我们自己写这些逻辑，

我们需要先设置TSS

```
struct TSS32 {
    int backlink, esp0, ss0, esp1, ss1, esp2, ss2, cr3;
    int eip, eflags, eax, ecx, edx, ebx, esp, ebp, esi, edi;
    int es, cs, ss, ds, fs, gs;
    int ldtr, iomap;
};
```

从开头的backlink起，到cr3为止的几个成员，保存的不是寄存器的数据，而是与任务设置相关的信息，在执行任务切换的时候这些成员不会被写入(有时backlink会被写入)

eip相当于每个任务的PC

ldtr和iomap也是有关于任务设置的，但是也不能随便赋值，在这里我们先将ldtr置为0，将iomap置0x40000000就好了。

> 我觉得这个多任务管理应该就算是进程管理了，这个TSS就相当于我们课上学的PCB。
>
> PCB中包含进程的描述信息，进程的控制信息，资源占用信息和处理器的现场保护。
>
> 不过这个TSS中似乎又没有描述信息和资源占用信息。

然后我们研究一下如何具体进行任务的切换。回忆一下far模式的jmp，

格式为`jmp dword 段:段内地址`

如果我们jmp到的目标地址段不是可执行的代码，而是tss的话，cpu就会把他理解成任务切换

> CPU每次执行带有段地址的指令时，都会去确认一下 GDT中的设置，以便判断接下来要执行的 JMP指令到底是普通的 far-JMP，还是任务切换。也就是说，从汇编程序翻译出来的机器语言来看，普通的 far-JMP和任务切换的 far-JMP，指令本身是没有任何区别的。

在跳转之前我们还需要修改tr寄存器的值，我们需要写个相应的naskfunc

```
_load_tr: ; void load_tr(int tr);
    LTR [ESP+4] ; tr
    RET
```

然后我们要进行far跳转

```
_taskswitch4: ; void taskswitch4(void);
    JMP 4*8:0
    RET
```

我们在10s计时器完成之后进行taskswitch。

不过我们现在还没有准备好taskb呢，先准备下tss_b

```
tss_b.eip = (int) &task_b_main;
tss_b.eflags = 0x00000202; /* IF = 1; */
tss_b.eax = 0;
tss_b.ecx = 0;
tss_b.edx = 0;
tss_b.ebx = 0;
tss_b.esp = task_b_esp;
tss_b.ebp = 0;
tss_b.esi = 0;
tss_b.edi = 0;
tss_b.es = 1 * 8;
tss_b.cs = 2 * 8;
tss_b.ss = 1 * 8;
tss_b.ds = 1 * 8;
tss_b.fs = 1 * 8;
tss_b.gs = 1 * 8;
```

我们给cs置为GDT的2号，其他寄存器都置为GDT的1号，asmhead.nas的时候也是一样的。

我们写一个什么都不做只hlt的函数来进行测试

```
void task_b_main(void)
{
    for (;;) { io_hlt(); }
}
```

然后为任务b分配内存空间作为栈使用

最后再测试一下。没啥特别的现象，就先不贴图了，我们一鼓作气接着往下整吧。

之前我们实现了从task a main函数切到taskb，我们接下来实现从taskb切回taska，方法很简单，我们只要再写个naskfuncjmp回3*8:0就可以了

```
_taskswitch3: ; void taskswitch3(void);
    JMP 3*8:0
    RET
```

我们把这个任务切换写成一个方便使用的函数吧！

```
_farjmp: ; void farjmp(int eip, int cs);
    JMP FAR [ESP+4] ; eip, cs
    RET
```

有了这个farmp函数，我们就能够自由跳转了。

任务之间往往需要共享一些数据，由于跳转之后我们无法访问另一个任务内部的数据，所以我们决定把东西存到一个固定的内存位置。

```
*((int *) 0x0fec) = (int) sht_back;            // 存入
sht_back = (struct SHEET *) *((int *) 0x0fec)  // 读出
```

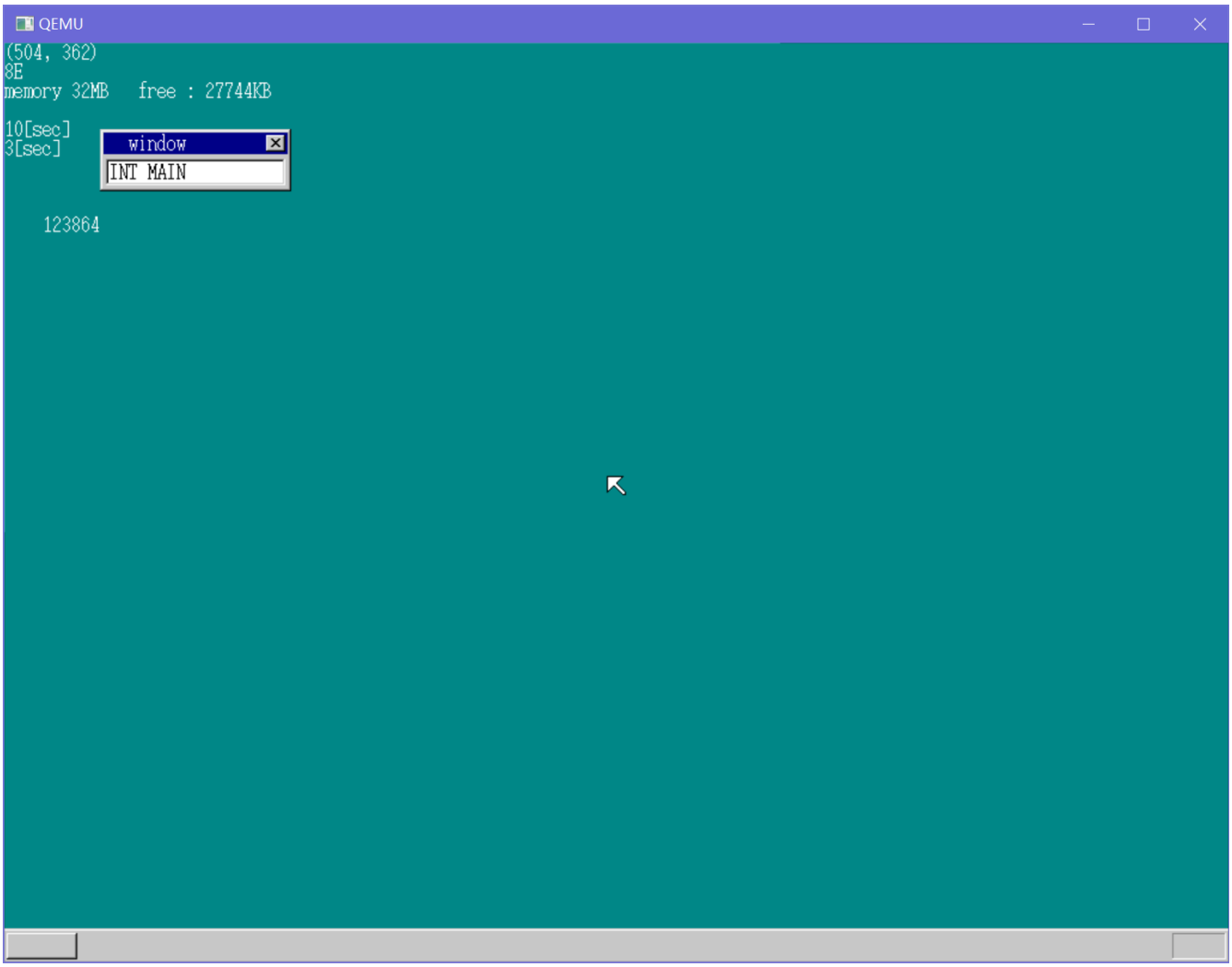
就像这样。

共享完数据之后，我们让b来打印些东西吧！

```
void task_b_main(void) {
    struct FIFO32 fifo;
    struct TIMER *timer_ts;
    int i, fifobuf[128], count = 0;
    char s[11];
    struct SHEET *sht_back;
    for (;;) {
        count++;
        sprintf(s, "%10d", count);
```

```
            putfonts8_asc_sht(sht_back, 0, 144, COL8_FFFFFF, COL8_008484, s, 10);
            io_cli();
            if (fifo32_status(&fifo) == 0) {
                io_sti();
            } else {
                //
            }
        }
    }
```

跑一下



计数器和工作框都在同时工作，真是棒极了。

然后我们进行一些性能上的优化。我们的计数器每改变一次值就要更新一次屏幕，但实际上大多数显示器的刷新率都是60Hz的，我们根本不需要那么频繁的更新屏幕内容，我们可以设定一个定时器，每0.01s绘图count一次。（没法设成60Hz的是因为我们的定时器事时间粒度是0.01s，如果定时器设成0.02s的话，那么就是一秒刷新50次了，效果不好）

```
timer_init(timer_put, &fifo, 1);
timer_settime(timer_put, 1);
for (;;) {
    count++;
    io_cli();
    if (fifo32_status(&fifo) == 0) {
        io_sti();
    } else {
        i = fifo32_get(&fifo);
        io_sti();
        if (i == 1) {
            sprintf(s, "%11d", count);                    // timer打印
            putfonts8_asc_sht(sht_back, 0, 144, COL8_FFFFFF, COL8_008484, s, 11);
            timer_settime(timer_put, 1);
        } else if (i == 2) {
            farjmp(0, 3 * 8);
            timer_settime(timer_ts, 2);
        }
    }
}
```

之前我们获取sht_back的方法有点麻烦，如果我们利用c语言参数的内存地址是esp+4这个特性的话，我们或许可以用参数的方式来获取sht_back。

我们设置taskb的esp需要稍微做一下改动

```
task_b_esp = memman_alloc_4k(memman, 64 * 1024) + 64 * 1024 - 8;
*((int *) (task_b_esp + 4)) = (int) sht_back;
```

为什么减8不是减4呢，因为esp是写入的低地址端，如果减4的话，如果像栈中写入内容，就会覆盖掉我们的参数。测试一下，工作正常，不贴图了。

作者提到了return，我们此处又了解了c语言的一些特性，C语言会再return的时候return到[esp]所指定的位置，我们以后或许会用到这个特性。

在此之前，我们的多任务是两个任务分散控制，他们有各自的交出cpu的逻辑，我们不可能以后每有一个新任务就新写一个逻辑，我们希望任务切换对于任务来说是透明的。

我们先采用round robin的调度方法，每隔一段固定的时间进行一次调度

```
void mt_init(void)
{
    mt_timer = timer_alloc();
    /* timer_initは必要ないのでやらない */
    timer_settime(mt_timer, 2);
    mt_tr = 3 * 8;
```

```
    return;
}

void mt_taskswitch(void)
{
    if (mt_tr == 3 * 8) {
        mt_tr = 4 * 8;
    } else {
        mt_tr = 3 * 8;
    }
    timer_settime(mt_timer, 2); // 不使用timer_init是因为在发生超时的时候不需要向FIFO缓冲区写入数据
    farjmp(0, mt_tr);
    return;
}
```

然后修改一下定时器中断处理程序

```
void inthandler20(int *esp)
{
    struct TIMER *timer;
    char ts = 0;
    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;
        if (timer != mt_timer) {
            fifo32_put(timer->fifo, timer->data);
        } else {
            ts = 1;
        }
        timer = timer->next;
    }
    timerctl.t0 = timer;
    timerctl.next = timer->timeout;
    if (ts != 0) {
        mt_taskswitch();
    }
    return;
}
```

为什么我们在最后进行任务切换而不是立即切换呢?

这是因为进行任务切换的时候中断中断允许标志可能会被重设为1，这样就有可能在中断还未处理完成的时候又来一个中断，造成中断丢失。

完成中断服务程序的修改之后，我们只需要把taska和taskb里面关于任务切换的代码删除就好了。

今天先到这里吧。

Day 16

回忆之前我们设置taskb，好麻烦啊，一长串赋值代码，一点也不优雅。我们仿照sheetctl写一个taskctl来简化我们的多任务创建吧。

```
#define MAX_TASKS 1000 /*最大任务数量*/
#define TASK_GDT0 3 /*定义从GDT的几号开始分配给TSS */
struct TASK {
    int sel, flags; /* sel用来存放GDT的编号*/
    struct TSS32 tss;
};
struct TASKCTL {
    int running; /*正在运行的任务数量*/
    int now; /*这个变量用来记录当前正在运行的是哪个任务*/
    struct TASK *tasks[MAX_TASKS];
    struct TASK tasks0[MAX_TASKS];
};
```

task_alloc函数负责从taskctl的tasklist里找到一个空闲的task，然后对他的tss进行初始化。

task_init函数负责初始化taskctl，让正在运行的程序成为一个task，方便管理。

```
void task_run(struct TASK *task)
{
    task->flags = 2; /*活动中标志*/
    taskctl->tasks[taskctl->running] = task;
    taskctl->running++;
    return;
}
```

task_run将task添加到tasks的末尾，然后让running加1（大概就是送到ready）

然后我们的任务切换函数也需要进行一下修改。显然我们如果只有一个任务的话是不需要进行任务切换的

```
void task_switch(void)
{
    timer_settime(task_timer, 2);
    if (taskctl->running >= 2) {
        taskctl->now++;
        if (taskctl->now == taskctl->running) {
            taskctl->now = 0;
        }
        farjmp(0, taskctl->tasks[taskctl->now]->sel);
    }
    return;
}
```

```
struct TASK *task_b;

task_init(memman);
task_b = task_alloc();
task_b->tss.esp = memman_alloc_4k(memman, 64 * 1024) + 64 * 1024 - 8;
task_b->tss.eip = (int) &task_b_main;
task_b->tss.es = 1 * 8;
task_b->tss.cs = 2 * 8;
task_b->tss.ss = 1 * 8;
task_b->tss.ds = 1 * 8;
task_b->tss.fs = 1 * 8;
task_b->tss.gs = 1 * 8;
*((int *) (task_b->tss.esp + 4)) = (int) sht_back;
task_run(task_b);
```

然后我们harimain里就可以做如上修改

任务列表中的任务不是每时每刻都需要运行的，不需要运行的时候我们可以让他们休眠，不给他们分配时间片，来提高CPU的利用率。

```
void task_sleep(struct TASK *task)
{
    int i;
    char ts = 0;
    if (task->flags == 2) {         // 要休眠了
        if (task == taskctl->tasks[taskctl->now]) {
            ts = 1;                 // 如果是自己让自己休眠的话，我们一会还要做些其他的事情
        }
        // 找task
        for (i = 0; i < taskctl->running; i++) {
            if (taskctl->tasks[i] == task) {
                // 搁这里呢
                break;
            }
        }
        taskctl->running--;
        if (i < taskctl->now) {
            taskctl->now--;
        }
        for (; i < taskctl->running; i++) {
            taskctl->tasks[i] = taskctl->tasks[i + 1];
        }
        task->flags = 1; // 不工作的状态
        if (ts != 0) {
            // 切换
            if (taskctl->now >= taskctl->running) {
                // 修正now
                taskctl->now = 0;
            }
            farjmp(0, taskctl->tasks[taskctl->now]->sel); // 自己让自己休眠之后要立即切换任务
```

```
        }
    }
    return;
}
```

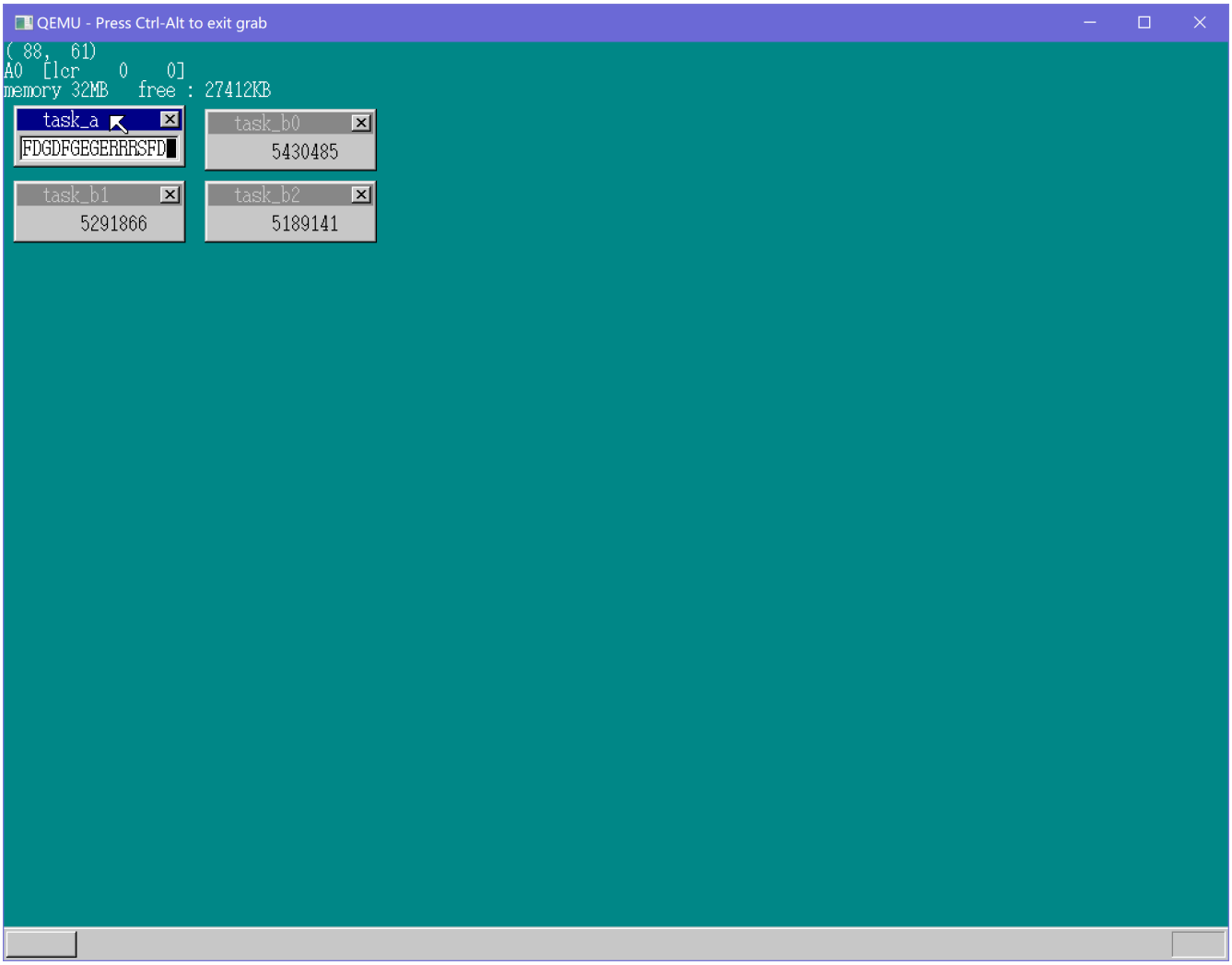
自动唤醒，我们可以在FIFO中附加一个task，当有事件来的时候自动唤醒相应的任务进行数据处理，这样处理程序就可以及时的去休息，不用一直检查，做无用功了。

```
void fifo32_init(struct FIFO32 *fifo, int size, int *buf, struct TASK *task)
/* FIFO缓冲区初始化*/
{
    fifo->size = size;
    fifo->buf = buf;
    fifo->free = size;
    fifo->flags = 0;
    fifo->p = 0;
    fifo->q = 0;
    fifo->task = task; // 有数据写入时需要唤醒的任务
    return;
}
```

然后再fifo入队函数中添加如下代码

```
if (fifo->task != 0) {
    if (fifo->task->flags != 2) { // 如果任务处于休眠状态（避免重复注册）
        task_run(fifo->task); // 将任务唤醒
    }
}
```

我们的多任务管理差不多就成型了，我们接下来加两个窗口玩一玩吧。看看各个窗口的计时能否同时进行。



虽然刷新的很慢，但是增长的速度都差不多。

哦对了，因为另外三个窗口不是焦点窗口，所以我们把他们刷新的速度给调慢了。

我们接着引入非抢占式优先级调度，taskctl内部用round robin，分的时间片根据priority的大小而有不同。另外关于优先级的设计，对于与用户产生直接交互的部分，我们要让他的优先级尽可能高（iOS优先处理动画所以看起来比安卓流畅很多）

开设多个taskctl，当任意时刻，只在level最高的taskctl中进行任务调度。这样就能保证优先的任务被及时处理了。

实际上，我们不需要创建多个TASKCTL，只要在TASKCTL中创建多个tasks[]即可

```
#define MAX_TASKS_LV 100
#define MAX_TASKLEVELS 10
struct TASK {
    int sel, flags; /* se1用来存放GDT的编号*/
    int level, priority;
    struct TSS32 tss;
};
struct TASKLEVEL {
    int running; /*正在运行的任务数量*/
    int now; /*这个变量用来记录当前正在运行的是哪个任务*/
    struct TASK *tasks[MAX_TASKS_LV];
};
struct TASKCTL {
    int now_lv; /*现在活动中的LEVEL */
    char lv_change; /*在下次任务切换时是否需要改变LEVEL，这个字段感觉可以用来搞多级反馈任务调度 */
    struct TASKLEVEL level[MAX_TASKLEVELS];
    struct TASK tasks0[MAX_TASKS];
};
```

每个level中我们最多允许创建100个task，共10个level，我们总共最多可以有1000个任务

task_now函数，用来返回现在活动中的struct TASK的内存地址

task_add函数，用来向struct TASKLEVEL中添加一个任务

task_remove函数，用来从struct TASKLEVEL中删除一个任务（代码结构和task_sleep差不多）

task_switchsub函数，用来在任务切换时决定接下来切换到哪个LEVEL

然后我们需要改写一些其他的函数，首先是task_init，我们先让最开始的任务呆在level 0（最高级），之后再考虑用task_run重新设置

```
void task_run(struct TASK *task, int level, int priority)
{
    if (level < 0) {
        level = task->level; /*不改变LEVEL */
    }
    if (priority > 0) {
        task->priority = priority;
    }
    if (task->flags == 2 && task->level != level) { /*改变活动中的LEVEL */
        task_remove(task); /*这里执行之后flag的值会变为1，于是下面的if语句块也会被执行*/
    }
    if (task->flags != 2) {
        /*从休眠状态唤醒的情形*/
        task->level = level;
        task_add(task);
    }
    taskctl->lv_change = 1; /*下次任务切换时检查LEVEL */
    return;
}
```

task_sleep也要进行相应的改写

```
void task_sleep(struct TASK *task)
{
    struct TASK *now_task;
    if (task->flags == 2) {
        /*如果处于活动状态*/
        now_task = task_now();
        task_remove(task); /*执行此语句的话flags将变为1 */
        if (task == now_task) {
            /*如果是让自己休眠，则需要进行任务切换*/
            task_switchsub();
            now_task = task_now(); /*在设定后获取当前任务的值*/
            farjmp(0, now_task->sel);
        }
    }
    return;
}
```

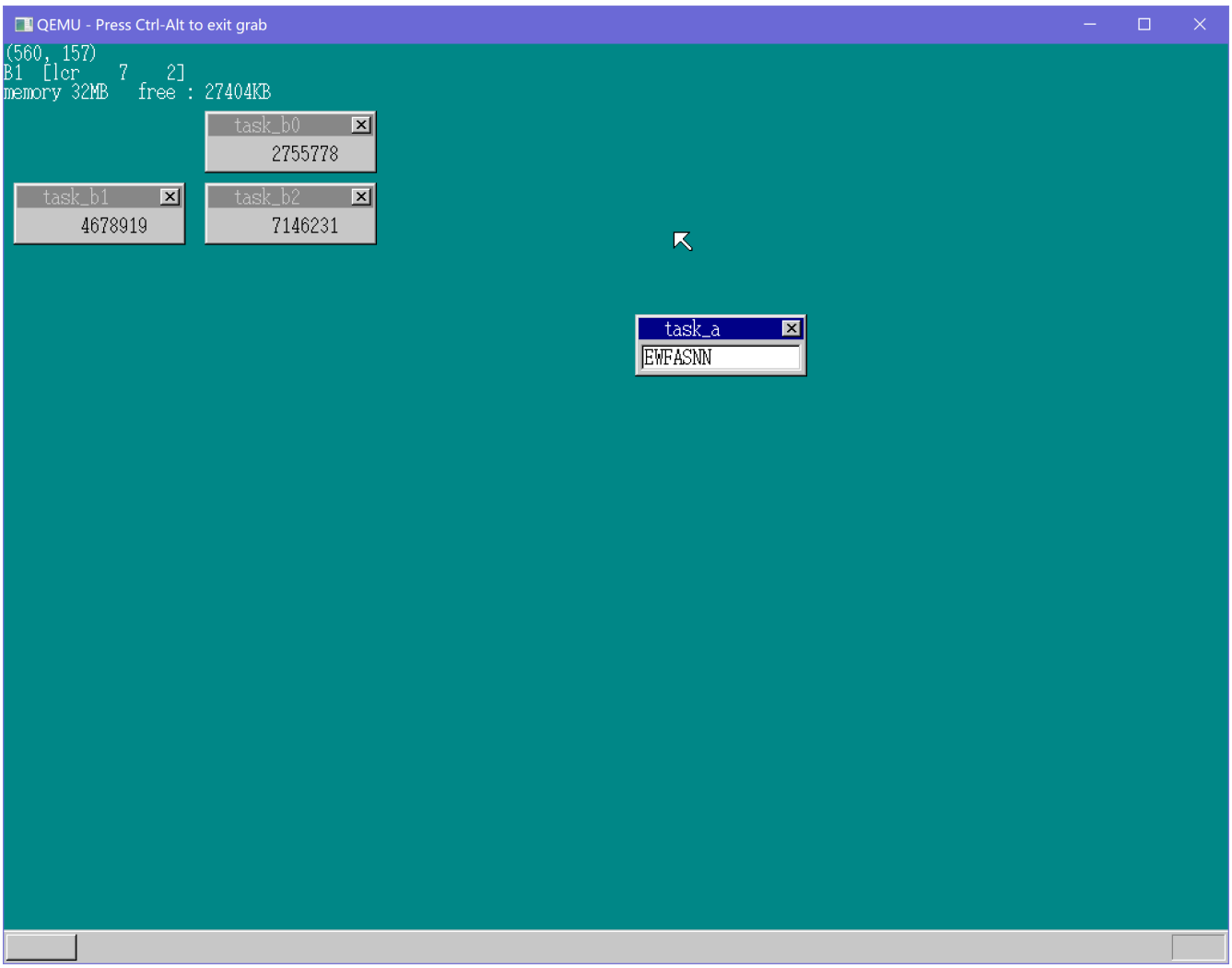
最后是task_switch，我们要对lv_change不为0是做出相应的处理

```
void task_switch(void)
```

```
{
    struct TASKLEVEL *tl = &taskctl->level[taskctl->now_lv];
    struct TASK *new_task, *now_task = tl->tasks[tl->now];
    tl->now++;
    if (tl->now == tl->running) {
        tl->now = 0;
    }
    if (taskctl->lv_change != 0) {
        task_switchsub();
        tl = &taskctl->level[taskctl->now_lv];
    }
    new_task = tl->tasks[tl->now];
    timer_settime(task_timer, new_task->priority);
    if (new_task != now_task) {
        farjmp(0, new_task->sel);
    }
    return;
}
```

然后fifo唤醒task的时候我们也需要修改一下，以相同优先级重新run task就行了，代码就不贴了。

最后我们改写harimain，我们将任务A设为level 0，其他任务弄成level 2吧，然后进行测试。



Day 17

如果所有任务都去休眠了怎么办？系统无事可做，我们只好hlt。为了降低编码复杂度，我们可以设置一个idle process，他的功能就是循环hlt

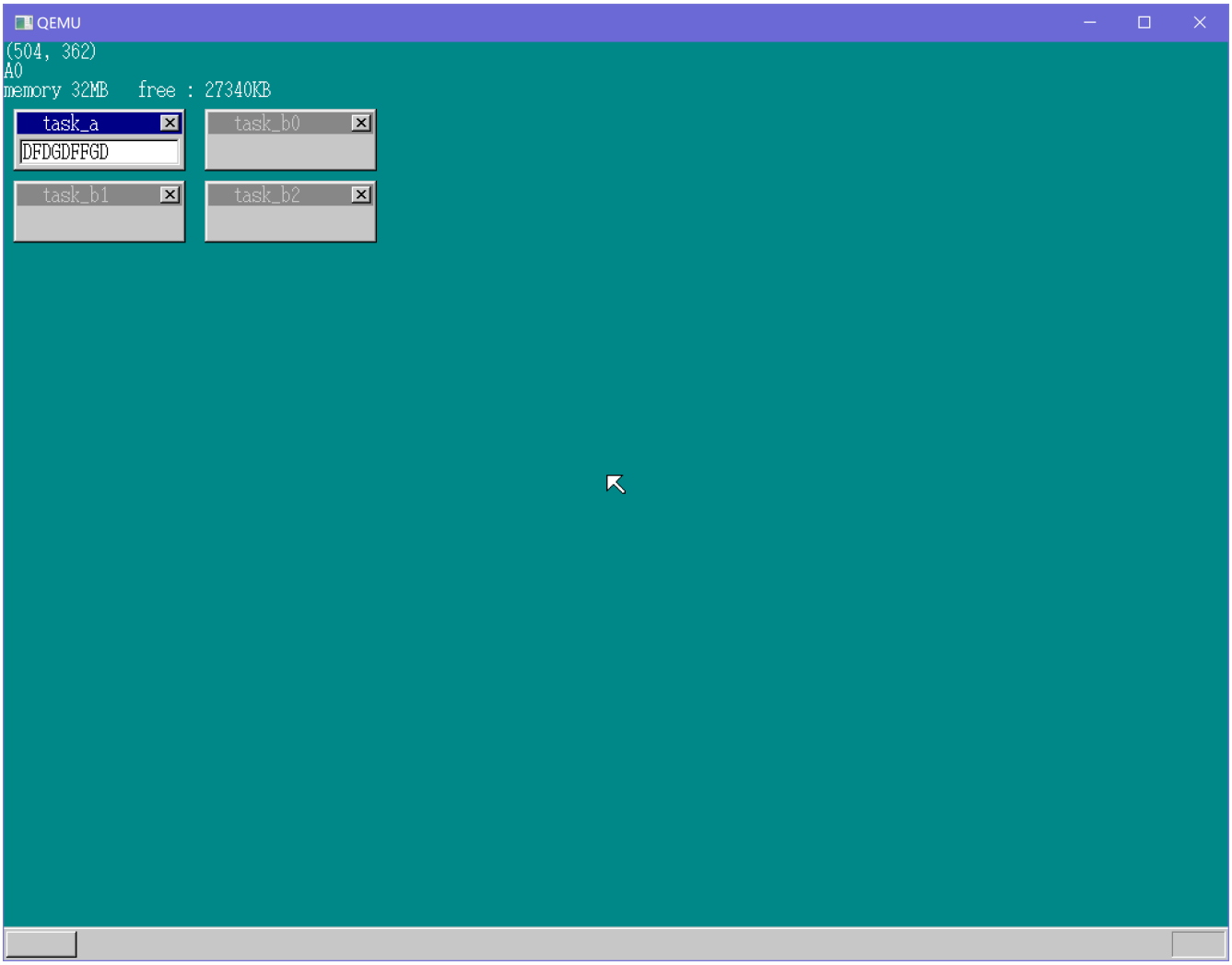
```
for (;;) io_hlt();
```

这个idle process具有最低的优先级，他就像链表中的哨兵一样，是为了降低我们编码复杂度而存在。

我们所要接着做的，就是再task_init的时候，把这个task放入最下层的taskctl中。

```
struct TASK *task_init(struct MEMMAN *memman)
{
    struct TASK *task, *idle;
    idle = task_alloc();
    idle->tss.esp = memman_alloc_4k(memman, 64 * 1024) + 64 * 1024;
    idle->tss.eip = (int) &task_idle;
    idle->tss.es = 1 * 8;
    idle->tss.cs = 2 * 8;
    idle->tss.ss = 1 * 8;
    idle->tss.ds = 1 * 8;
    idle->tss.fs = 1 * 8;
    idle->tss.gs = 1 * 8;
    task_run(idle, MAX_TASKLEVELS - 1, 1);
    return task;
}
```

我们修改一下我们的程序，不启动taskb，只保留输入框的taska，然后我们不给外部输入。



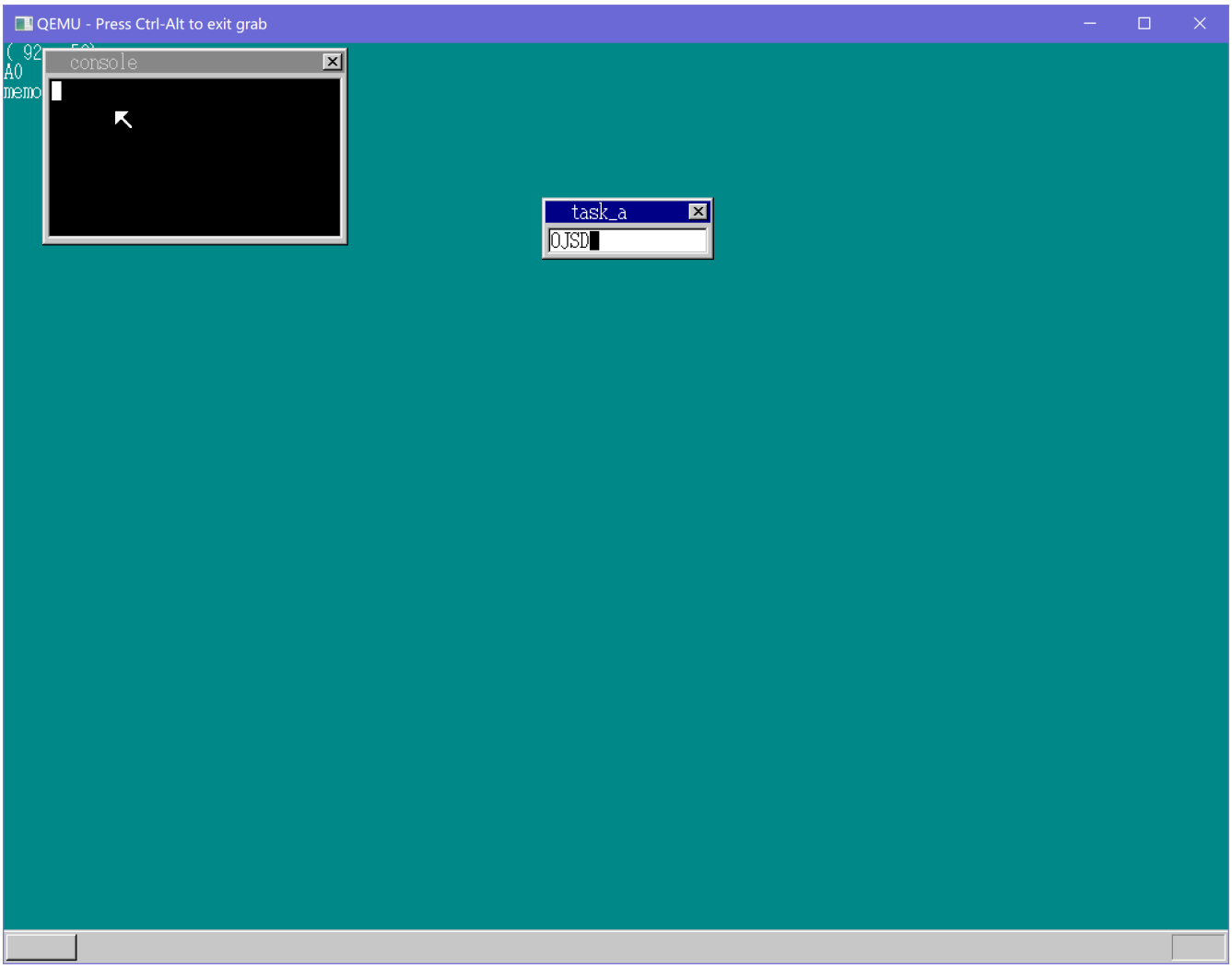
工作没有出现异常

然后我们来创建命令行窗口，方法是和创建输入框大同小异的，不同的是我们再让自己休眠的时候可以用task_now()函数来获取自己task的地址

```
void console_task(struct SHEET *sheet)
{
    struct FIFO32 fifo;
    struct TIMER *timer;
    struct TASK *task = task_now();

    int i, fifobuf[128], cursor_x = 8, cursor_c = COL8_000000;
    fifo32_init(&fifo, 128, fifobuf, task);

    timer = timer_alloc();
    timer_init(timer, &fifo, 1);
    timer_settime(timer, 50);

    for (;;) {
        io_cli();
```

```
        if (fifo32_status(&fifo) == 0) {
            task_sleep(task);
            io_sti();
        } else {
            i = fifo32_get(&fifo);
            io_sti();
            if (i <= 1) {
                if (i != 0) {
                    timer_init(timer, &fifo, 0);
                    cursor_c = COL8_FFFFFF;
                } else {
                    timer_init(timer, &fifo, 1);
                    cursor_c = COL8_000000;
                }
                timer_settime(timer, 50);
                boxfill8(sheet->buf, sheet->bxsize, cursor_c, cursor_x, 28, cursor_x + 7,
43);
                sheet_refresh(sheet, cursor_x, 28, cursor_x + 8, 44);
            }
        }
    }
}
```

make run嘻嘻

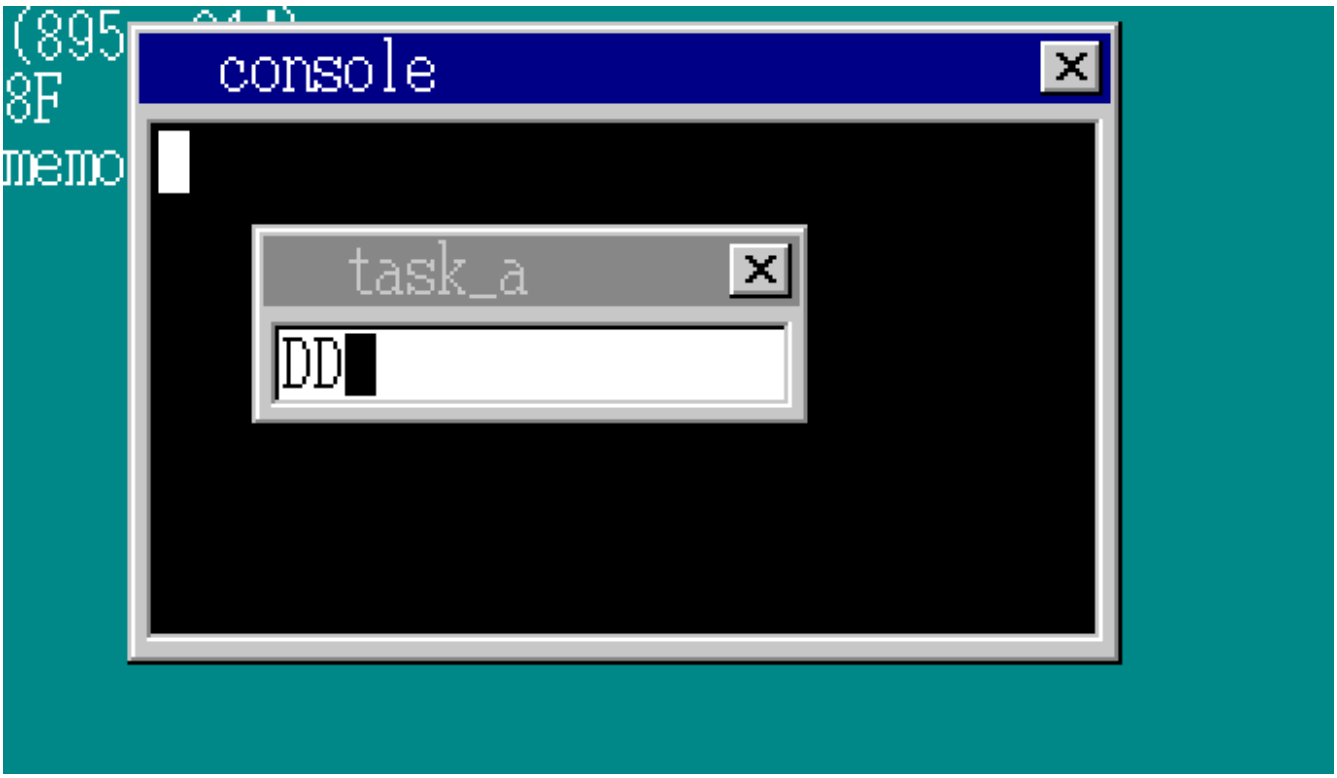


两个光标都在闪烁，但是相位稍有差别，看来是启动时间不一致所导致的。

之前我们只能再taska的输入框中进行输入，我们要想再console中输入，我们必须要实现切换窗口焦点的功能，我们决定使用tab键作为切换窗口焦点的快捷键。窗口获得焦点之后，窗口标题栏应当变为蓝色，而非焦点窗口标题栏应当变为灰色。

我们再设置一个keyto变量，用于指定焦点在哪一个窗体，之后我们处理按键事件的时候就可以根据keyto变量做出相应的响应

```
if (i == 256 + 0x0f) { /* 按键事件处理中对tab的处理 */
    if (key_to == 0) {
        key_to = 1;
        make_wtitle8(buf_win,  sht_win->bxsize,  "task_a",  0);
        make_wtitle8(buf_cons, sht_cons->bxsize, "console", 1);
    } else {
        key_to = 0;
        make_wtitle8(buf_win,  sht_win->bxsize,  "task_a",  1);
        make_wtitle8(buf_cons, sht_cons->bxsize, "console", 0);
    }
    sheet_refresh(sht_win,  0, 0, sht_win->bxsize,  21);
    sheet_refresh(sht_cons, 0, 0, sht_cons->bxsize, 21);
}
```

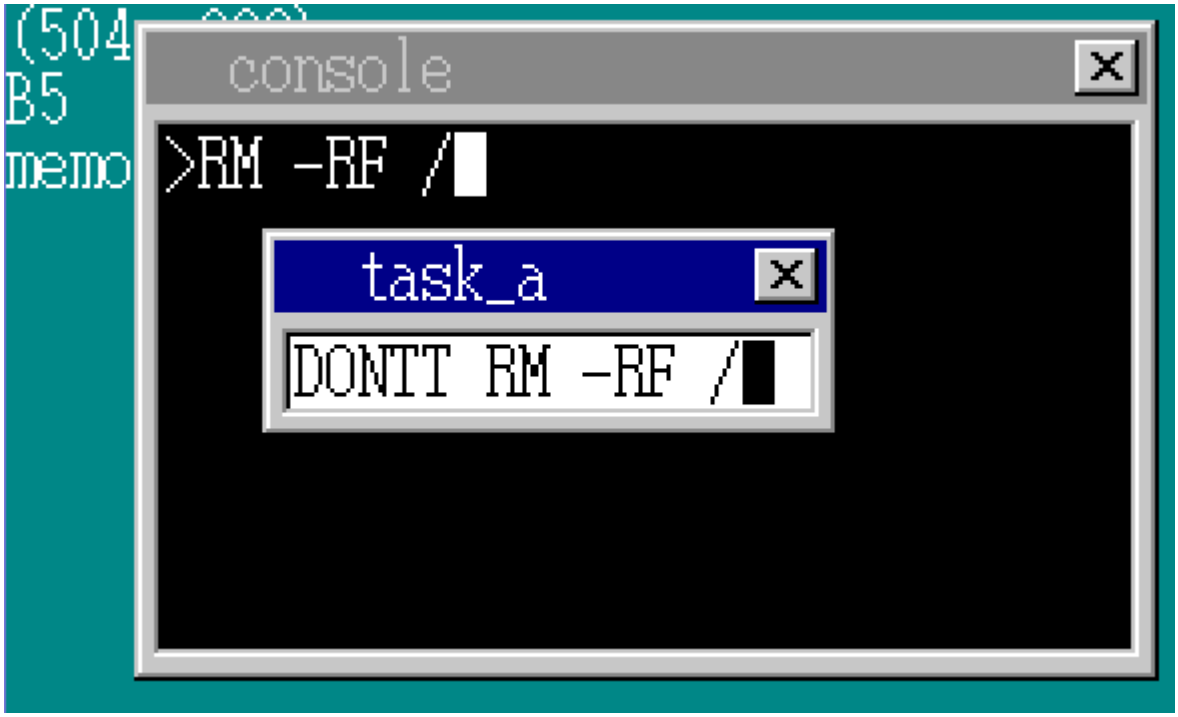


为了让输入也能够在窗体之间切换，我们还需要做些其他的事情

为了在console中输入，我们要知道console的fifo，然后只要把数据送到哪个fifo里就可以了。在task中添加一个fifo，让他作为按键事件的fifo。注意我们送进去ascii而不是原始十六进制数，这样可以不用单独的在consoletask中写转换逻辑了。

然后我们在console task中写上相应的按键事件处理逻辑。

```
void console_task(struct SHEET *sheet)
{
    struct TIMER *timer;
    struct TASK *task = task_now();
    int i, fifobuf[128], cursor_x = 16, cursor_c = COL8_000000;
    char s[2];

    fifo32_init(&task->fifo, 128, fifobuf, task);
    timer = timer_alloc();
    timer_init(timer, &task->fifo, 1);
    timer_settime(timer, 50);
    putfonts8_asc_sht(sheet, 8, 28, COL8_FFFFFF, COL8_000000, ">", 1); // 提示符
    for (;;) {
        io_cli();
        if (fifo32_status(&task->fifo) == 0) {
            task_sleep(task);
            io_sti();
        } else {
            i = fifo32_get(&task->fifo);
            io_sti();
            if (i <= 1) { // 光标处理
                if (i != 0) {
                    timer_init(timer, &task->fifo, 0);
                    cursor_c = COL8_FFFFFF;
                } else {
                    timer_init(timer, &task->fifo, 1);
                    cursor_c = COL8_000000;
                }
                timer_settime(timer, 50);
            }
            if (256 <= i && i <= 511) { // 来自task a的键盘数据
                if (i == 8 + 256) { // 退格键
                    if (cursor_x > 16) {
                        putfonts8_asc_sht(sheet, cursor_x, 28, COL8_FFFFFF, COL8_000000, "
", 1);
                        cursor_x -= 8; // 擦除前移
                    }
                } else { // 一般字符
                    if (cursor_x < 240) {
                        s[0] = i - 256; // 打印字符并后移
                        s[1] = 0;
                        putfonts8_asc_sht(sheet, cursor_x, 28, COL8_FFFFFF, COL8_000000, s,
1);
                        cursor_x += 8;
                    }
                }
            }
            boxfill8(sheet->buf, sheet->bxsize, cursor_c, cursor_x, 28, cursor_x + 7, 43);
            sheet_refresh(sheet, cursor_x, 28, cursor_x + 8, 44);
        }
    }
}
```



工作正常

接下来我们处理shift键，当shift键按下的时候，应当改变默认的大小写输入状态，若是符号或者数字键，则应当输入上方的符号。

我们准备一个key_shift变量，当左Shift按下时置为1，右Shift按下时置为2，两个都不 按时置为0，两个都按下的时候就置为3。

```
if (i == 256 + 0x2a) {  // lshift on
    key_shift |= 1;
}
if (i == 256 + 0x36) {  // rshift on
    key_shift |= 2;
}
if (i == 256 + 0xaa) {  // lshift off
    key_shift &= ~1;
}
if (i == 256 + 0xb6) {  // rshift off
    key_shift &= ~2;
}
```

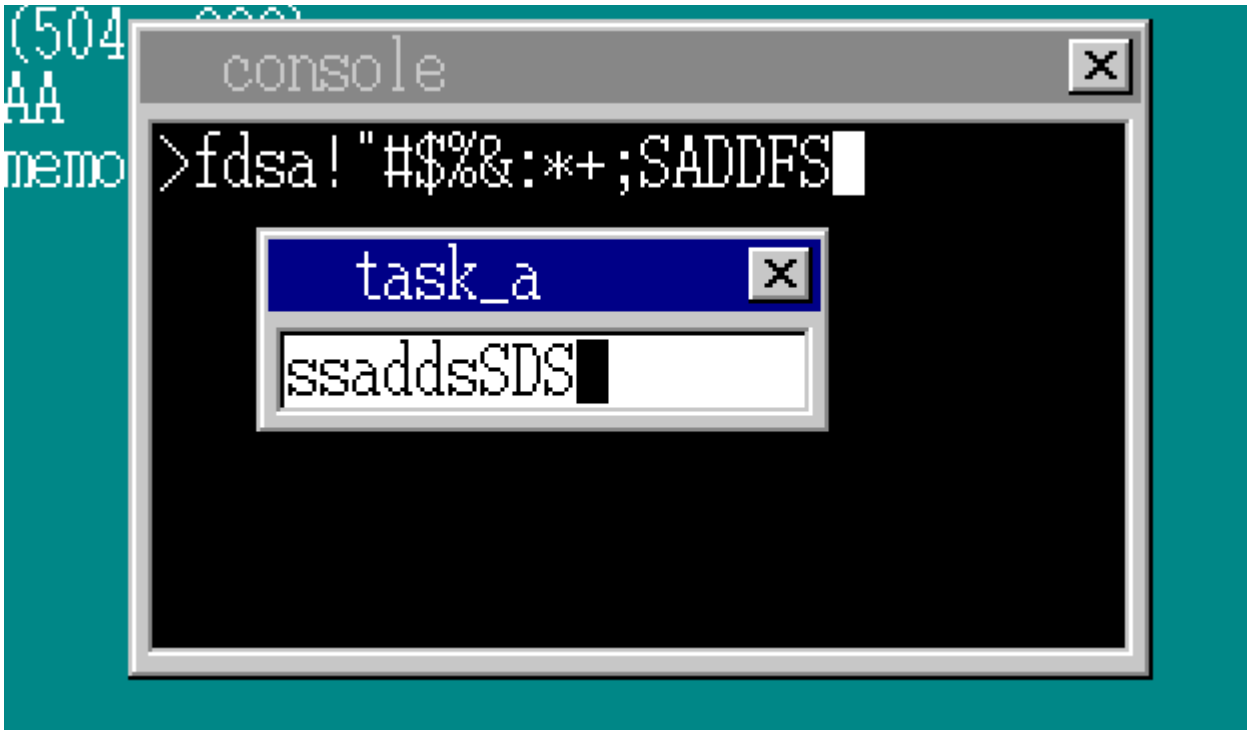
除此之外我们还需要处理一下caps lock，从key_leds中取出指定的bit就可以了。

binfo->leds 的第 4 位 → ScrollLock 状态 binfo->leds 的第 5 位 → NumLock 状态 binfo->leds 的第 6 位 → CapsLock 状态

然后输入大写的条件是key_shift^caps_lock

```
if ('A' <= s[0] && s[0] <= 'Z') {
    if (((key_leds & 4) == 0 && key_shift == 0) ||
    ((key_leds & 4) != 0 && key_shift != 0)) {
        s[0] += 0x20;
    }
}
```

make run



咦？模拟器中caps lock怎么不好用呢？

原来亮灯灭灯逻辑需要我们自己来写

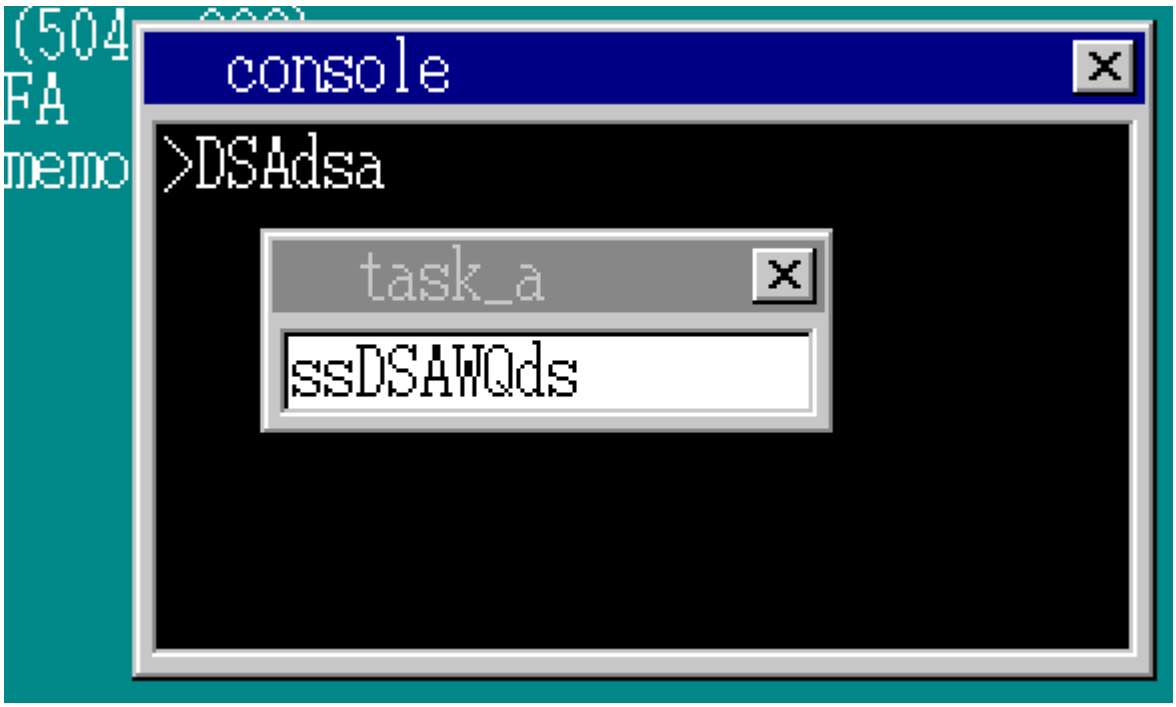
- 读取状态寄存器，等待 bit 1的值变为 0。
- 向数据输出（0060）写入要发送的 1个字节数据。
- 等待键盘返回 1个字节的信息，这和等待键盘输入所采用的方法相同（用 IRQ等待 或者用轮询状态寄存器 bit 1 的值直到其变为 0都可以）。
- 返回的信息如果为 0xfa，表明 1个字节的数据已成功发送给键盘。如为 0xfe则表明 发送失败，需要返回第 1步重新发送。

而要控制LED的状态，需要按上述方法执行两次，向键盘发送EDxx数据。其中，xx的bit0代表ScrollLock，bit 1代表NumLock，bit 2代表CapsLock（0表示熄灭，1表示点亮）。bit 3～7为保留位，置0即可

```
for (;;) {
    if (fifo32_status(&keycmd) > 0 && keycmd_wait < 0) { /*从此开始*/
        /*如果存在向键盘控制器发送的数据，则发送它 */
        keycmd_wait = fifo32_get(&keycmd);
        wait_KBC_sendready();
        io_out8(PORT_KEYDAT, keycmd_wait);
    } /*到此结束*/
```

```
    io_cli();
    if (fifo32_status(&fifo) == 0) {
        task_sleep(task_a);
        io_sti();
    } else {
        i = fifo32_get(&fifo);
        io_sti();
        if (256 <= i && i <= 511) { /* 键盘数据 */
            /*从此开始*/ if (i == 256 + 0x3a) { /* CapsLock */
                key_leds ^= 4;
                fifo32_put(&keycmd, KEYCMD_LED);
                fifo32_put(&keycmd, key_leds);
            }
            if (i == 256 + 0x45) { /* NumLock */
                key_leds ^= 2;
                fifo32_put(&keycmd, KEYCMD_LED);
                fifo32_put(&keycmd, key_leds);
            }
            if (i == 256 + 0x46) { /* ScrollLock */
                key_leds ^= 1;
                fifo32_put(&keycmd, KEYCMD_LED);
                fifo32_put(&keycmd, key_leds);
            }
            if (i == 256 + 0xfa) { /*键盘成功接收到数据*/
                keycmd_wait = -1;
            }
            if (i == 256 + 0xfe) { /*键盘没有成功接收到数据*/
                wait_KBC_sendready();
                io_out8(PORT_KEYDAT, keycmd_wait);
                /*到此结束*/ }
        } else if (512 <= i && i <= 767) { /*鼠标数据*/

        } else if (i <= 1) { /*光标用定时器*/

        }
    }
}
```

make run一下



完美