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;
}
```

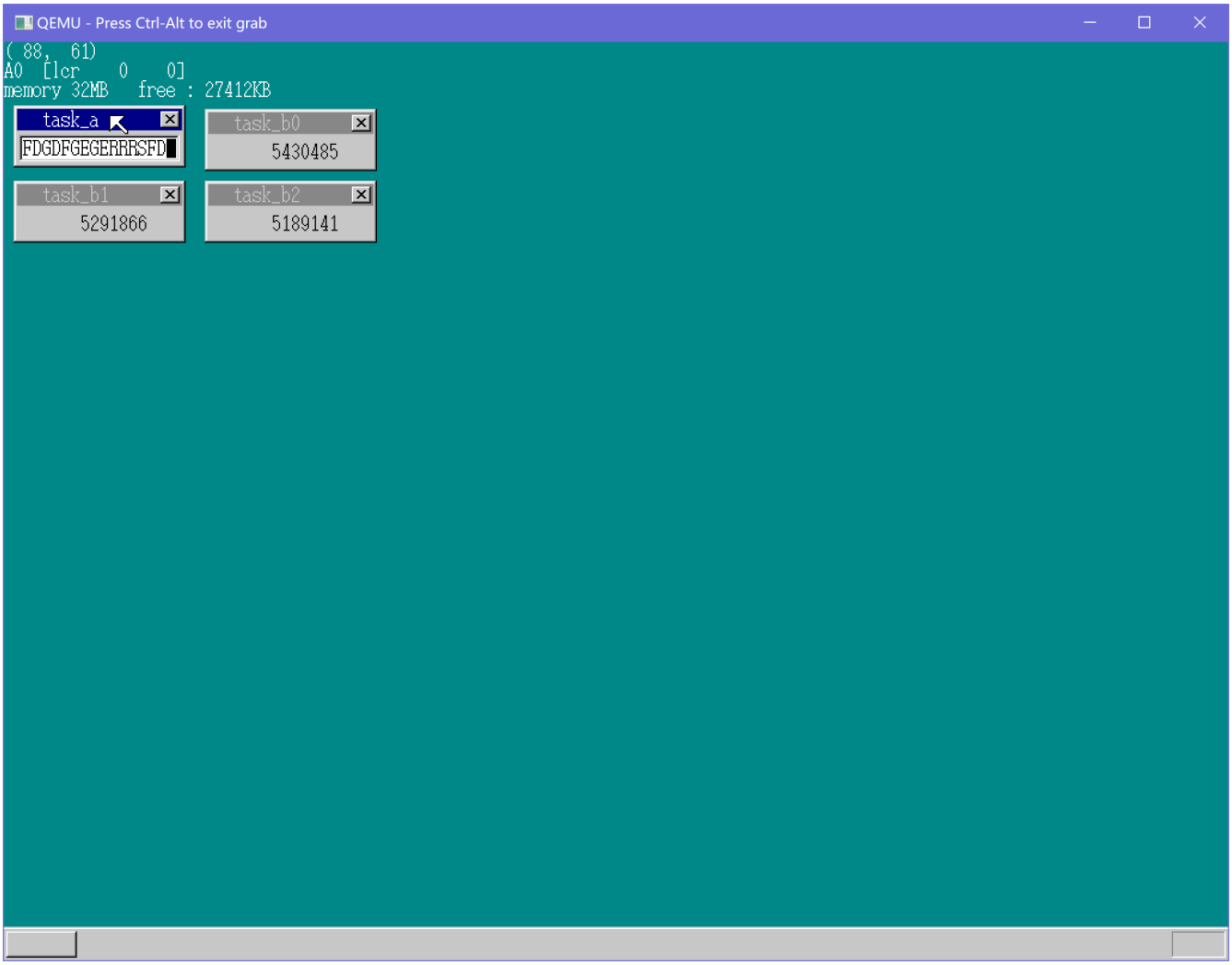
自动唤醒，我们可以在FIFO中附加一个task，当有事件来的时候自动唤醒相应的任务进行数据处理，这样处理程序就可以及时的去休息，不用一直检查，做无用功了。

```
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。