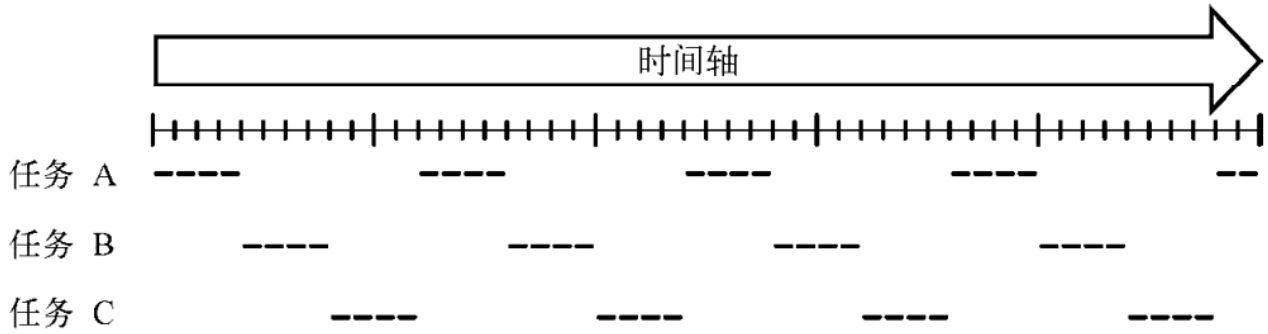
Day 15

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。



※ 1个CPU通过反复切换来执行3个任务
※ 由于切换速度很快，看上去好像在同时执行3个任务一样

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
```

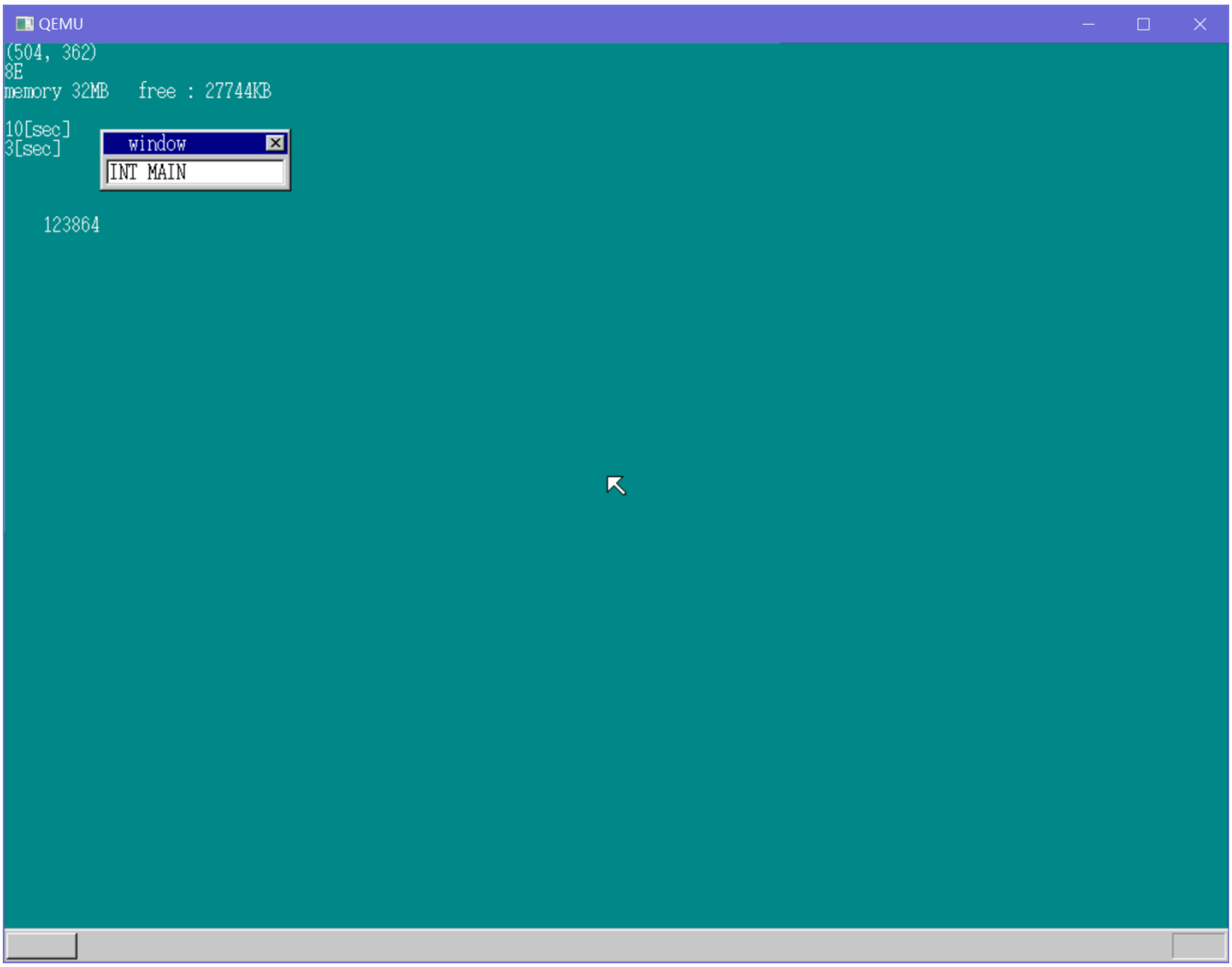
就像这样。

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。