

# PA4: 虚实交错的魔法：分时多任务

---

匡亚明学院 张祎扬 181840326

实验进度：我完成了全部内容

## PA4: 虚实交错的魔法：分时多任务

必答题

多道程序

实现上下文切换

实现多道程序系统

虚实交错的魔法

超越容量的界限

理解分页机制

在NEMU中实现分页机制

在分页机制上运行用户进程

在分页机制上运行仙剑奇侠传

支持虚存管理的多道程序

来自外部的声音

实现抢占多任务

编写不朽的传奇

展示你的计算机系统

写在最后

## 必答题

分时多任务的具体过程

分页机制

当开启分页机制之后，函数 `_vme_init()` 就会对vme进行相关的初始化，并且设置cr0和cr3寄存器。

```

int _vme_init(void* (*pgalloc_f)(size_t), void (*pgfree_f)(void*)) {
    pgalloc_usr = pgalloc_f;
    pgfree_usr = pgfree_f;

    int i;

    // make all PDEs invalid
    for (i = 0; i < NR_PDE; i++) {
        kpdirs[i] = 0;
    }

    PTE *ptab = kptabs;
    for (i = 0; i < NR_KSEG_MAP; i++) {
        uint32_t pdir_idx = (uintptr_t)segments[i].start / (PGSIZE * NR_PTE);
        uint32_t pdir_idx_end = (uintptr_t)segments[i].end / (PGSIZE * NR_PTE);
        for (; pdir_idx < pdir_idx_end; pdir_idx++) {
            // fill PDE
            kpdirs[pdir_idx] = (uintptr_t)ptab | PTE_P;

            // fill PTE
            PTE pte = PGADDR(pdir_idx, 0, 0) | PTE_P;
            PTE pte_end = PGADDR(pdir_idx + 1, 0, 0) | PTE_P;
            for (; pte < pte_end; pte += PGSIZE) {
                *ptab = pte;
                ptab++;
            }
        }
    }

    set_cr3(kpdirs);
    //printf("%x\n",cpu.cr3);
    set_cr0(get_cr0() | CR0_PG);
    vme_enable = 1;

    return 0;
}

```

开启分页机制后，就在函数 `page_translate()` 中进行虚拟地址到物理地址的转换。

```

3 static paddr_t page_translate(vaddr_t addr){
4     //printf("%x\n",addr);
5     CR3 cr3;
6     PDE pde;
7     PTE pte;
8     cr3.val = cpu.cr3;
9     //printf("%x\n",cr3.page_directory_base);
10    //printf("%x\n",addr);
11    pde.val = paddr_read((cr3.page_directory_base<<12) + PDE_INDEX(addr) * sizeof(PDE), sizeof(PDE));
12    //printf("%x\n",pde.val);
13    //printf("%lx\n",sizeof(PDE));
14    assert(pde.present == 1);
15    pte.val = paddr_read((pde.page_frame<<12) + PTE_INDEX(addr) * sizeof(PTE), sizeof(PTE));
16    assert(pte.present == 1);
17    paddr_t paddr = (pte.page_frame<<12)|(addr & PAGE_MASK);
18    //printf("the translated paddr is %x\n",paddr);
19    return paddr;
20 }

```

这里面的步骤是，首先通过cr3寄存器获得页目录表的基地址，然后通过虚拟地址中的offset从页目录表中获取对应的页目录项。页目录项的高20位指示页表的基地址，并且要判断最低位present位是否有效。接下来，根据基地址和虚拟地址中间的10位，在页表中获取对应的页表项，并判断present位是不是1.最后，页表项的高20位和虚拟地址低12位的offset结合成为真正的物理地址。

开启了分页机制之后，loader函数也要做相应的修改，ELF中的地址全部成为了虚拟地址，所以在加载程序的时候，首先要申请新的物理页面，并且获得它的地址，再调用map函数建立物理地址到虚拟地址到映射，最后用fs\_read()把相应的内容读到物理地址。这样在访问程序的时候，就会根据虚拟地址找到对应的物理地址，并且读取该物理地址处的数据了。

## 硬件中断

每10ms在timer.c中触发一次时钟中断。`timer_intr()`调用`dev_raise_intr()`来请求中断。

```
8 void timer_intr() {
9     if (nemu_state.state == NEMU_RUNNING) {
10         extern void dev_raise_intr(void);
11         dev_raise_intr();
12     }
13 }
```

nemu中的cpu在执行完每一条指令后会检查当前状态是否需要响应中断。响应中断的条件是：cpu处于开中断状态(cpu.eflags.IF==1),并且接收到了硬件中断信息 (cpuINTR == 1).在这里cpu响应中断的方式是自陷。所以当需要响应中断的时候，调用`raise_intr()`保存现场，随后跳转到处理中断的代码。具体的处理中断代码由中断号NO决定，在这里就是时钟中断的序号。

```
5 void raise_intr(uint32_t NO, vaddr_t ret_addr) {
6     /* TODO: Trigger an interrupt/exception with ``NO''.
7      * That is, use ``NO'' to index the IDT.
8      */
9     rtl_push(&cpu.eflags_value);
10    rtl_push(&cpu.cs);
11    rtl_push(&ret_addr);
12    cpu.eflags.IF = 0;
13    vaddr_t idt_addr = cpu.idtr.base + sizeof(GateDesc)*NO;
14    vaddr_t low_addr = vaddr_read(idt_addr,2) & 0xffff;
15    vaddr_t high_addr = vaddr_read(idt_addr+4,4) & 0xffff0000;
16    decinfo.jmp_pc = high_addr | low_addr;
17    decinfo.is_jmp = 1;
18    //printf("%x",decinfo.jmp_pc);
19    rtl_j(decinfo.jmp_pc);
20 }
21 }
22
23 #define IRQ_TIMER 32
24
25 bool isa_query_intr(void) {
26     if(cpuINTR & cpu.eflags.IF){
27         cpuINTR = false;
28         raise_intr(IRQ_TIMER,cpu.pc);
29         return true;
30     }
31
32     return false;
33 }
34 }
```

在nexus-am中，对32号中断处理的入口是 \_\_am\_irq0.

```
38 int _cte_init(_Context*(*handler)(_Event, _Context*)) {
39     static GateDesc idt[NR_IRQ]; //256
40
41     // initialize IDT
42     for (unsigned int i = 0; i < NR_IRQ; i++) {
43         idt[i] = GATE(STS_TG32, KSEL(SEG_KCODE), __am_vecnull, DPL_KERN);
44     }
45     //STS_TG32=0xF
46     //KSEL(desc) ((desc) << 3) | DPL_KERN
47     //SEG_KCODE 1
48
49     // ----- interrupts -----
50     idt[32] = GATE(STS_IG32, KSEL(SEG_KCODE), __am_irq0, DPL_KERN);
51     // ----- system call -----
52     idt[0x80] = GATE(STS_TG32, KSEL(SEG_KCODE), __am_vecsys, DPL_USER);
53     idt[0x81] = GATE(STS_TG32, KSEL(SEG_KCODE), __am_vectrap, DPL_KERN);
54
55     set_idt(idt, sizeof(idt));

```

```
4 .globl __am_irq0;      __am_irq0: pushl $32; jmp __am_asm_trap
5 .globl __am_vecnull; __am_vecnull: pushl $-1; jmp __am_asm_trap
6
7 __am_asm_trap:
8 pushal
9
10 pushl $0
11
12 pushl %esp
13 call __am_irq_handle
14
15 movl %eax,%esp
16
17
18 addl $4, %esp
19 popal
20 addl $4, %esp
21
22 iret
```

irq0会将中断号压栈，保存现场，然后调用 \_\_am\_irq\_handle 的处理函数是nanos-lite中的 do\_event()。在函数 do\_event() 中添加对时钟中断响应的情况，并且调用 yield()，强迫当前用户产生自陷，从而切换到另一个用户进程。

在nanos-lite的调度函数 schedule() 中，对各个用户进程进行时间分配，完成了分时多任务的进行。

## 多道程序

### 实现上下文切换

首先实现CTE的 \_\_kcontext() 函数。它的作用是在 stack 的底部创建一个以 entry 为返回地址的上下文结构，然后返回这一结构的指针。可以暂时先忽略参数 arg。查看头文件，我发现 \_\_Area 结构中有两个 void \*，分别标志着 stack 的起始和终点。因为要把上下文创建在 stack 底部，所以要把指针类型转化为指向 \_\_context 的指针，再将指针减一，则实际地址减小了一个 \_\_context 的大小。

然后实现 `schedule()` 函数。目前按照讲义内容完善即可。

接着在收到 `_EVENT_TIELD` 之后，把上下文 `c` 作为参数，调用 `schedule()` 函数并返回新的上下文（不要忘记声明函数）。

然后修改 `_am_asm_trap()` 的实现，在恢复上下文之前先切换到新进程的上下文。因为在调用完 `_am_irq_handle()` 之后，返回值在寄存器 `eax` 中，要切换栈顶指针 `esp`，只需要加一条汇编指令，把寄存器 `eax` 送寄存器 `esp` 即可。

完成后我尝试用 `hello_fun` 进行测试，却发现它在初始化中死循环了。

```
[/home/katherine/ics2019/nanos-lite/src/main.c,15,main] Hello World! from Nanos-lite
[/home/katherine/ics2019/nanos-lite/src/main.c,16,main] Build time: 16:24:24, Dec 8 2019
[/home/katherine/ics2019/nanos-lite/src/ramdisk.c,28,init_ramdisk] ramdisk info: start = 0x1024e0, end = 0x1bd8571, size = 28139665 bytes
[/home/katherine/ics2019/nanos-lite/src/device.c,88,init_device] Initializing devices...
[/home/katherine/ics2019/nanos-lite/src/irq.c,17,init_irq] Initializing interrupt/exception handler...
[/home/katherine/ics2019/nanos-lite/src/main.c,30,main] Finish initialization
[/home/katherine/ics2019/nanos-lite/src/main.c,15,main] 'Hello World!' from Nanos-lite
[/home/katherine/ics2019/nanos-lite/src/main.c,16,main] Build time: 16:24:24, Dec 8 2019
[/home/katherine/ics2019/nanos-lite/src/ramdisk.c,28,init_ramdisk] ramdisk info: start = 0x1024e0, end = 0x1bd8571, size = 28139665 bytes
[/home/katherine/ics2019/nanos-lite/src/device.c,88,init_device] Initializing devices...
[/home/katherine/ics2019/nanos-lite/src/irq.c,17,init_irq] Initializing interrupt/exception handler...
[/home/katherine/ics2019/nanos-lite/src/main.c,30,main] Finish initialization
[/home/katherine/ics2019/nanos-lite/src/main.c,15,main] 'Hello World!' from Nanos-lite
[/home/katherine/ics2019/nanos-lite/src/main.c,16,main] Build time: 16:24:24, Dec 8 2019
[/home/katherine/ics2019/nanos-lite/src/ramdisk.c,28,init_ramdisk] ramdisk info: start = 0x1024e0, end = 0x1bd8571, size = 28139665 bytes
[/home/katherine/ics2019/nanos-lite/src/device.c,88,init_device] Initializing devices...
[/home/katherine/ics2019/nanos-lite/src/irq.c,17,init_irq] Initializing interrupt/exception handler...
[/home/katherine/ics2019/nanos-lite/src/main.c,30,main] Finish initialization
[/home/katherine/ics2019/nanos-lite/src/main.c,15,main] 'Hello World!' from Nanos-lite
[/home/katherine/ics2019/nanos-lite/src/main.c,16,main] Build time: 16:24:24, Dec 8 2019
[/home/katherine/ics2019/nanos-lite/src/ramdisk.c,28,init_ramdisk] ramdisk info: start = 0x1024e0, end = 0x1bd8571, size = 28139665 bytes
[/home/katherine/ics2019/nanos-lite/src/device.c,88,init_device] Initializing devices...
[/home/katherine/ics2019/nanos-lite/src/irq.c,17,init_irq] Initializing interrupt/exception handler...
[/home/katherine/ics2019/nanos-lite/src/main.c,30,main] Finish initialization
```

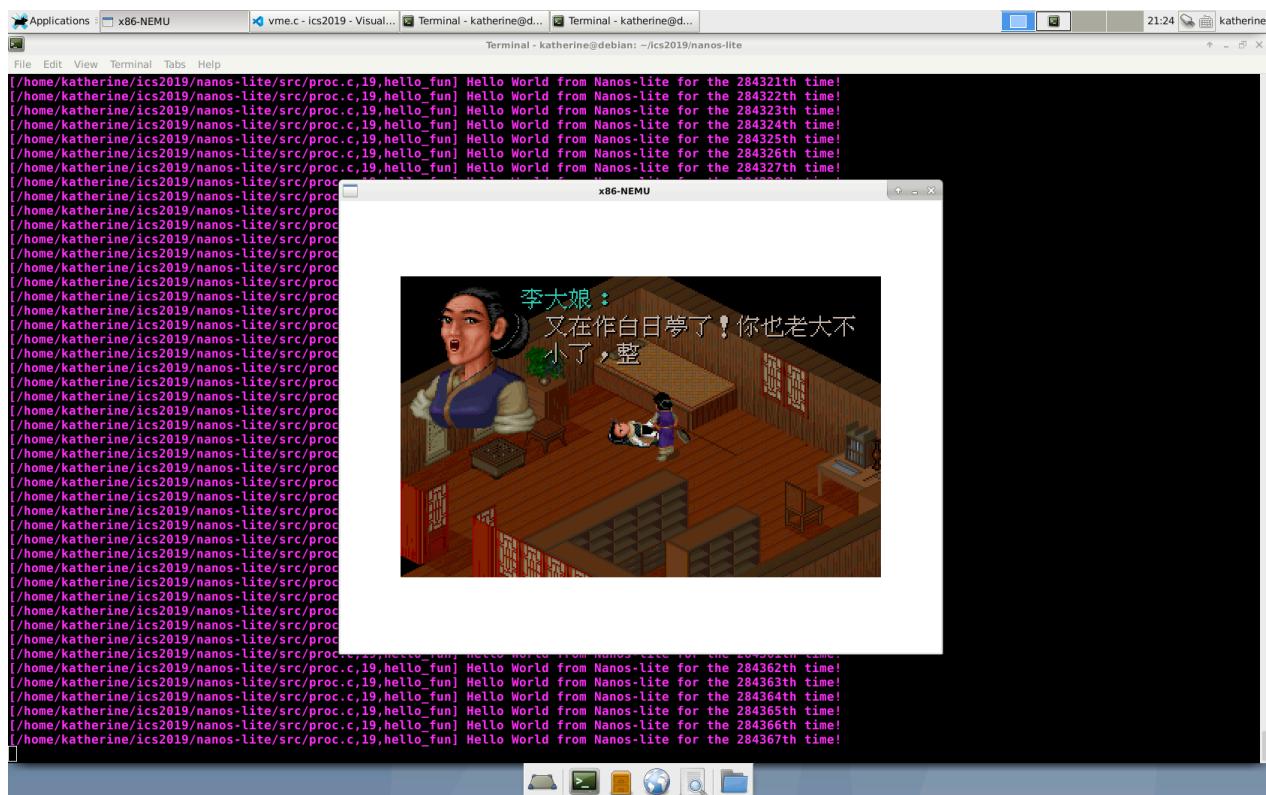
于是我尝试用 native 跑，发现结果是正常的，也就是说问题应该不是出在 nanos-lite，于是我就去排查 am。最后发现在 trap.S 中，加了一行汇编代码将 `eax` 送 `esp` 之后，应该删掉一个把 `esp` 加 4 的汇编语句。改完之后结果就正常了。

```
[/home/katherine/ics2019/nanos-lite/src/proc.c,18,hello_fun] Hello World from Nanos-lite for the 32939th time!
[/home/katherine/ics2019/nanos-lite/src/proc.c,18,hello_fun] Hello World from Nanos-lite for the 32940th time!
[/home/katherine/ics2019/nanos-lite/src/proc.c,18,hello_fun] Hello World from Nanos-lite for the 32941th time!
[/home/katherine/ics2019/nanos-lite/src/proc.c,18,hello_fun] Hello World from Nanos-lite for the 32942th time!
[/home/katherine/ics2019/nanos-lite/src/proc.c,18,hello_fun] Hello World from Nanos-lite for the 32943th time!
[/home/katherine/ics2019/nanos-lite/src/proc.c,18,hello_fun] Hello World from Nanos-lite for the 32944th time!
[/home/katherine/ics2019/nanos-lite/src/proc.c,18,hello_fun] Hello World from Nanos-lite for the 32945th time!
[/home/katherine/ics2019/nanos-lite/src/proc.c,18,hello_fun] Hello World from Nanos-lite for the 32946th time!
[/home/katherine/ics2019/nanos-lite/src/proc.c,18,hello_fun] Hello World from Nanos-lite for the 32947th time!
[/home/katherine/ics2019/nanos-lite/src/proc.c,18,hello_fun] Hello World from Nanos-lite for the 32948th time!
[/home/katherine/ics2019/nanos-lite/src/proc.c,18,hello_fun] Hello World from Nanos-lite for the 32949th time!
[/home/katherine/ics2019/nanos-lite/src/proc.c,18,hello_fun] Hello World from Nanos-lite for the 32950th time!
[/home/katherine/ics2019/nanos-lite/src/proc.c,18,hello_fun] Hello World from Nanos-lite for the 32951th time!
[/home/katherine/ics2019/nanos-lite/src/proc.c,18,hello_fun] Hello World from Nanos-lite for the 32952th time!
[/home/katherine/ics2019/nanos-lite/src/proc.c,18,hello_fun] Hello World from Nanos-lite for the 32953th time!
[/home/katherine/ics2019/nanos-lite/src/proc.c,18,hello_fun] Hello World from Nanos-lite for the 32954th time!
[/home/katherine/ics2019/nanos-lite/src/proc.c,18,hello_fun] Hello World from Nanos-lite for the 32955th time!
[/home/katherine/ics2019/nanos-lite/src/proc.c,18,hello_fun] Hello World from Nanos-lite for the 32956th time!
[/home/katherine/ics2019/nanos-lite/src/proc.c,18,hello_fun] Hello World from Nanos-lite for the 32957th time!
[/home/katherine/ics2019/nanos-lite/src/proc.c,18,hello_fun] Hello World from Nanos-lite for the 32958th time!
[/home/katherine/ics2019/nanos-lite/src/proc.c,18,hello_fun] Hello World from Nanos-lite for the 32959th time!
[/home/katherine/ics2019/nanos-lite/src/proc.c,18,hello_fun] Hello World from Nanos-lite for the 32960th time!
```

## 实现多道程序系统

对于 `_ucontext()` 的实现，结合讲义提供的栈帧图，我的理解是相当于在 `_kcontext()` 的基础上把原来的 `end` 减小了一点，多出来的空间就是 `_start()` 的栈帧。但是我还不是特别理解怎么规划 `_start()` 的栈帧的大小，这里好像也暂时没有用到的样子，于是就大概仿照 native 先写了一个 `_ucontext()`。写完之后我发现其实就算完全仿照 `_kcontext()`，PA4.1 的任务也还是可以跑的。以后再来继续打磨吧。

然后就是按照讲义所讲的添加开机画面进程，修改调度代码。最后在三个函数的开头加入 `_yield()` 调用。运行起来发现可以成功同时运行两个进程。

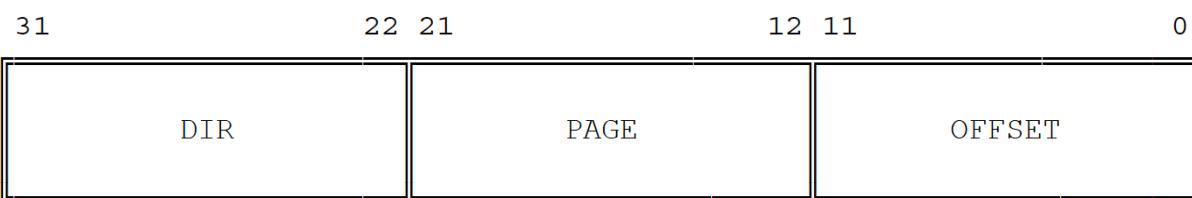


## 虚实交错的魔法

### 超越容量的界限

#### 理解分页机制

我决定在这里做一点笔记，记录我阅读手册理解页表机制的过程。



线性空间地址的格式如图。DIR域（高10位）是页目录的index，PAGE域（中间10位）是根据DIR索引到的页表的index，而OFFSET（低12位）是页表内的偏移量。

i386采用两层页表机制。第一层的是页目录。第二层是页中间表。当前页目录的物理地址存在控制寄存器CR3中。在这种情况下，线性地址的转换分两步完成。第一步是基于两级转换页表最终找到地址所在的页面，第二步是基于偏移量(offset)在所在页中找到对应偏移量的物理地址。

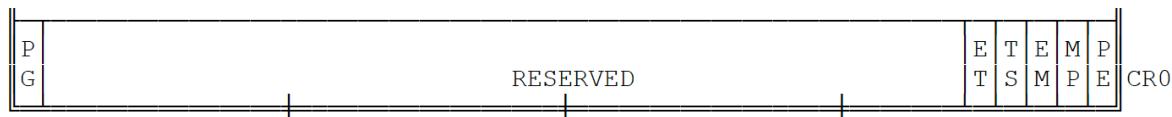
其余具体就不赘述了，计算机系统基础那本书上讲的很清楚。

#### 在NEMU中实现分页机制

首先，我按照讲义要求定义了宏 `HAS_VME`，并且尝试编译运行，碰到了未实现的指令，`0x0f 0x22`。通过查阅手册，我了解到该指令的作用是把32位操作数从通用寄存器移到控制寄存器CR0。于是转而实现这个指令。执行函数调用 `mov_r2cr`（在 `system.c` 中），发现这个指令没有实现，所以要先实现它，也就要先在cpu中添加寄存器cr0和cr3。实现完这个指令之后，继续运行，发现下一个没有实现的指令

是 `0x0f 0x20`，它的执行函数是 `mov_cr2r`，也需要我自己去实现。

阅读讲义得知CR0只需要设置PG位即可，阅读手册我知道了PG位是cr0寄存器中的最后一位。



为了实现分页机制，我已经实现了cr3和cr0寄存器，并且添加了操作它们的指令。下一步我要修改 `isa_vaddr_read()` 函数和 `isa_vaddr_write()` 函数。按照讲义添加 `isa_vaddr_read()` 函数的框架，我要做的是在if条件中添加对是否超出边界的判断。我在memory.h中找到了对宏 `PAGE_MASK` 和宏 `PAGE_SIZE` 的定义，就可以很轻松地实现条件的判断了。接着仿照read实现 `isa_addr_write()`，同样判断是否超过边界。在没超过边界时调用函数 `page_translate()` 进行转换，在超出边界时先 `assert(0)` 强行终止，之后要用到的时候再来实现跨页读写的功能吧。

接下来就需要实现 `page_translate()` 函数，这也是实现分页机制的关键所在。通过阅读头文件 mmu.h中的宏定义，我发现了CR3,PDE,PTE等非常有用的结构定义。所以我首先声明一个CR3结构，并且将寄存器cr3的值读入该结构，这样就可以直接利用结构的定义来获得我需要的部分，比如 `page_directory_base`。

根据手册，我可以知道线性地址的高10位是DIR，也就是页目录的索引，所以我要找的页目录的地址就是 `page_directory_base` 加上索引乘以页目录的大小。为了方便，我在mmu.h中定义了一个宏 `PDE_INDEX`，用来表示线性地址的高10位，这样就可以调用 `paddr_read()` 函数，并且用以上提到的几个变量算出需要的页目录的物理地址，并且将值读出 `sizeof(PDE)` 的长度，并且赋给pte.val.在这一过程完成后，根据讲义提示，要及时用assertion检查present位。

线性地址的中间10位是页表的索引，所以我依旧定义一个index宏来方便表示它。用同样的方法，调用 `paddr_read()` 来读取结构pte的值。并且用assert检查present位。

最后的物理地址就是pte的page\_frame加上线性地址中的offset。将它们组合之后作为函数的返回地址即可。

之后我尝试run，发现我之前插入的 `assert(present==1)` 报错了。我尝试打印了寄存器的值，发现 cr3永远是0，这让我很纳闷。而且run之后并没有跑nanos-lite，而是直接在跑nemu。我发现它根本就没有跑nanos-lite的main函数。这让我纳闷了很久。因为对present位的检查在函数 `page_translate()` 里，而只有 `isa_vaddr_read()` 和 `isa_vaddr_write()` 这两个函数会调用它。在我思考了很久很久很久以后.....我突然意识到！！！ `isa_vaddr_read()` 和 `isa_vaddr_write()` 函数在被我改过以后，是默认已经实现了分页机制的.....可是那个时候还没有调用到 `init_mm()` 函数，也就是说，分页机制还不存在.....我当时可是原原本本按照讲义上的改的.....这真的是一个大坑！所以我重新修改了这两个函数，用定义的宏CR0来读入寄存器cr0，判断结构中的 paging是不是1，如果是1，则说明实现了分页机制。当还没有执行 `init_mm()` 的时候，cr0寄存器的pg位还没有被设置为1，所以仍然采用原来的read和write函数的形式。

改完之后，果然两处判断present位的地方不再assert了。但是又出现了新的assert，是讲义中提到的数据跨越虚拟页边界的情况。因为之前是同时进行 `hello_fun` 和 `init` 两个进程，我注释掉 `init` 进程，只保留 `hello_fun`，可以成功运行了！

但我还是决定实现一下数据跨越虚拟页边界的情况。实现方法很简单，算出原地址和页边界的距离，将长度len分为两段，分别读取它们的地址，并且将其拼接起来。需要注意的是len的数据类型是int，左移的位数应该是位数而不是字节数。

全部实现之后，可以同时跑hello\_fun和init了！还可以跑仙剑！

## 在分页机制上运行用户进程

根据讲义提示，首先我需要在am中实现`_map()`函数。通过阅读源代码，我在x86.h中找到了一些可以使用的宏，比如`PDX()`, `PTX()`, `OFF()`。我会在map的实现中用到它们。大体思路就是如果有效位为0的话就申请一个新页面，然后通过索引index建立物理地址和虚拟地址的关系，在这里不作赘述。实现完`_map()`之后，我按照讲义修改或者调用了其他函数。最后一步是修改loader的实现。

在这里我大概停顿了一整天，因为一开始有点想不通虚拟地址物理地址之类的关系，也不知道怎么调用。后来printf了一下ELF表中的程序入口地址，发现修改了Makefile文件以后，它已经自动变成了虚拟地址0x40000000开始的地址。这给了我关于实现loader的很大的启发。一开始阅读讲义我有点云里雾里，以为loader要大改特改，后来想通之后发现其实很简单。只要添加几行代码，在分段读取程序体的时候先申请新的物理页面，再调用map进行映射，最后调用`fs_read()`时只需读到物理地址即可。为了防止程序大小超出页面大小，所以还要加一个判断，如果超出的话则继续申请继续读取。

全部实现完之后我修改了调度代码，按照讲义要求在exit中调用`_halt()`，但是出现了assertion。是present位为0的情况。

一开始我以为是我的loader实现了问题，于是我每一次读取之后都printf，发现读到的东西都是正确的，入口地址也是正确的。如下图(中间比较短的是读取的大小，可以看到最后不足一页的时候读取的大小小于页面大小)

```
[/home/katherine/ics2019/nanos-lite/src
enter loader
before read
0x4000105c
0x40001000
0x1000
0x1c6f000
0x1000
0x1c71000
0x1000
0x1c72000
0xac
0x1c73000
0x40005000
0x1000
0x1c74000
0x970
0x1c75000
0x40008000
0x868
0x1c76000
going out loader
[/home/katherine/ics2019/nanos-lite/src
```

一番寻找之后无果，我开启了DEBUG模式，并且打开log文件。如下图：

```

1296381 10116a: 89 c4          movl %eax,%esp
1296382 10116c: 83 c4 04      addl $0x4,%esp
1296383 10116f: 61             popa
1296384 101170: 83 c4 04      addl $0x4,%esp
1296385 101173: cf             addb %al,(%eax)

1296386 40000105c: 00 00       addb %al,(%eax)
1296387 40000105e: 00 00       addb %al,(%eax)
1296388 400001060: 00 00       addb %al,(%eax)
1296389 400001062: 00 00       addb %al,(%eax)
1296390 400001064: 00 00       addb %al,(%eax)
1296391 400001066: 00 00       addb %al,(%eax)
1296392 400001068: 00 00       addb %al,(%eax)
1296393 40000106a: 00 00       addb %al,(%eax)
1296394 40000106c: 00 00       addb %al,(%eax)
1296395 40000106e: 00 00       addb %al,(%eax)
1296396 400001070: 00 00       addb %al,(%eax)
1296397 400001072: 00 00       addb %al,(%eax)
1296398 400001074: 00 00       addb %al,(%eax)
1296399 400001076: 00 00       addb %al,(%eax)

```

这是一个很奇怪的现象，程序跳转到了正确的入口地址，但是这个地址却没有数据（全是0）。在这里我疑惑了很久.....因为经过之前那么多次的检验，`fs_read()`的行为不太可能出错.....并且我也通过输出调试法确定了在`loader`中虚拟地址和物理地址都是正确的，所以，是真的很迷惑。徒劳了很久之后我意识到，说不定是把虚拟地址转换为物理地址的函数错了，返回了错误的物理地址，而那个地址并没有加载有用的程序数据！！！于是我仔细检查了`page_translate`函数，但是并没有找到什么问题，（而且之前成功运行了一点分页机制也可以证明问题不大），于是我又查了`map`函数，最后发现是`map`写错了.....我鬼使神差的修改了`pa`的值。改完之后充满期待地再次编译运行，**hit! good ! Trap!**

```

[src/monitor/monitor.c,25,welcome] Debug: OFF
[src/monitor/monitor.c,28,welcome] Build time: 21:36:27, Dec 14 2019
Welcome to x86-NEMU!
For help, type "help"
[/home/katherine/ics2019/nanos-lite/src/mm.c,25,init_mm] free physical pages starting from 0x1c6d000
[/home/katherine/ics2019/nanos-lite/src/main.c,15,main] 'Hello World!' from Nanos-lite
[/home/katherine/ics2019/nanos-lite/src/main.c,16,main] Build time: 21:41:33, Dec 14 2019
[/home/katherine/ics2019/nanos-lite/src/ramdisk.c,28,init_ramdisk] ramdisk info: start = 0x1028e0, end = 0x1
[/home/katherine/ics2019/nanos-lite/src/device.c,91,init_device] Initializing devices...
[/home/katherine/ics2019/nanos-lite/src/irq.c,18,init_irq] Initializing interrupt/exception handler...
[/home/katherine/ics2019/nanos-lite/src/main.c,30,main] Finish initialization
nemu: HIT GOOD TRAP at pc = 0x00100c0e

```

这个小小的bug差不多耗费了我将近两个小时的时间，看到hit good trap的绿色以后，我开心地去洗澡了！

## 在分页机制上运行仙剑奇侠传

按照讲义的介绍，这一块内容其实并不难。无非就是实现一个新的`mm_brk()`然后在系统调用中使用它而已。但在这里我不知道第二个参数`increment`的作用。按照之前的讲义，`brk()`其实只有一个参数，就是新的`brk`。所以在这里我按照自己的理解实现，忽略了`increment`。实现完`mm_brk()`之后果然引发了`present`位缺失的assert。

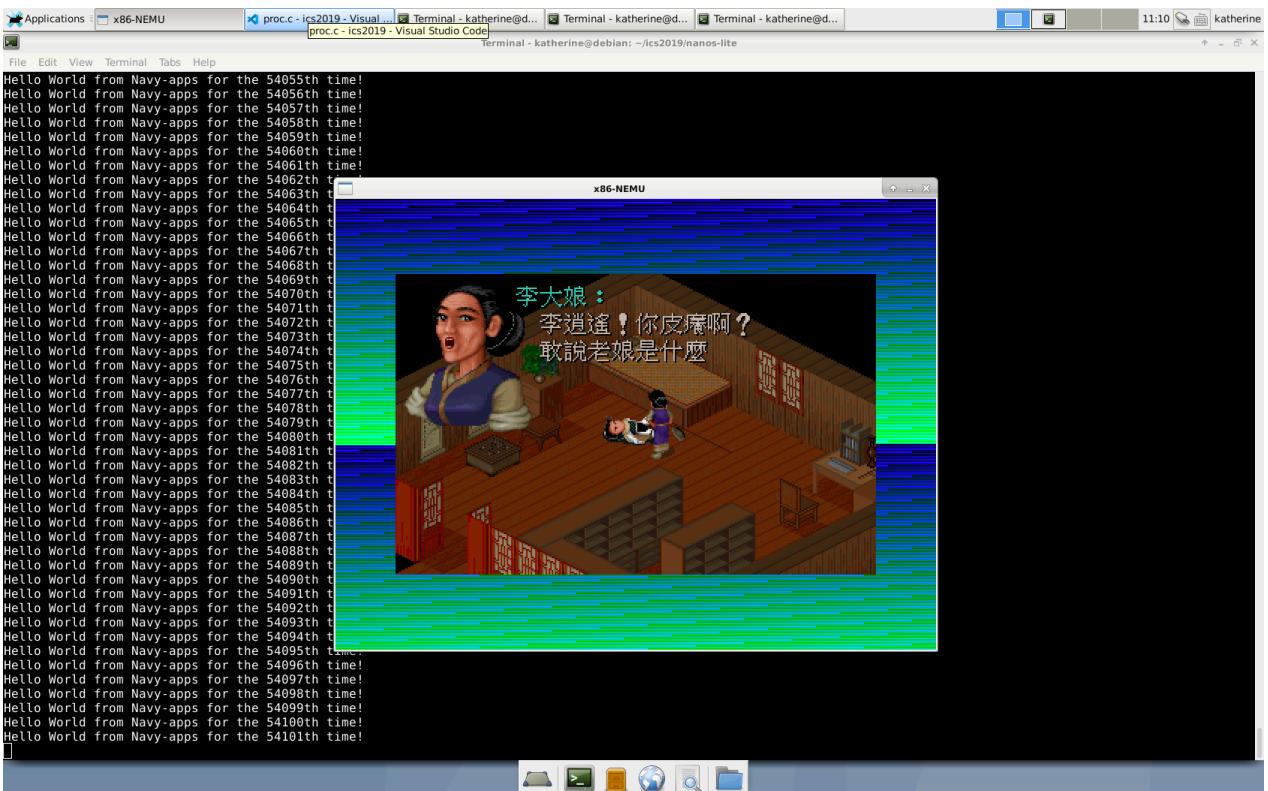
在这个问题上我大概困扰了整整三天，也没有找到原因所在。感谢屈道涵助教对我的帮助和启发，让我意识到最终问题其实出在我的`loader`实现上。一个是加载时加载的大小是`filesize`而不是`memsz`，导致.bss节应该加载到内存的部分没有加载，也就没有映射。第二个是`filesize`到`memsz`中间的清零写的不对（这个还有点难想清楚）。

找到问题在哪里以后，仙剑终于可以在分页机制上跑了！！！



## 支持虛存管理的多道程序

修改调度函数，可以成功运行仙剑奇侠传和hello这两个用户进程！

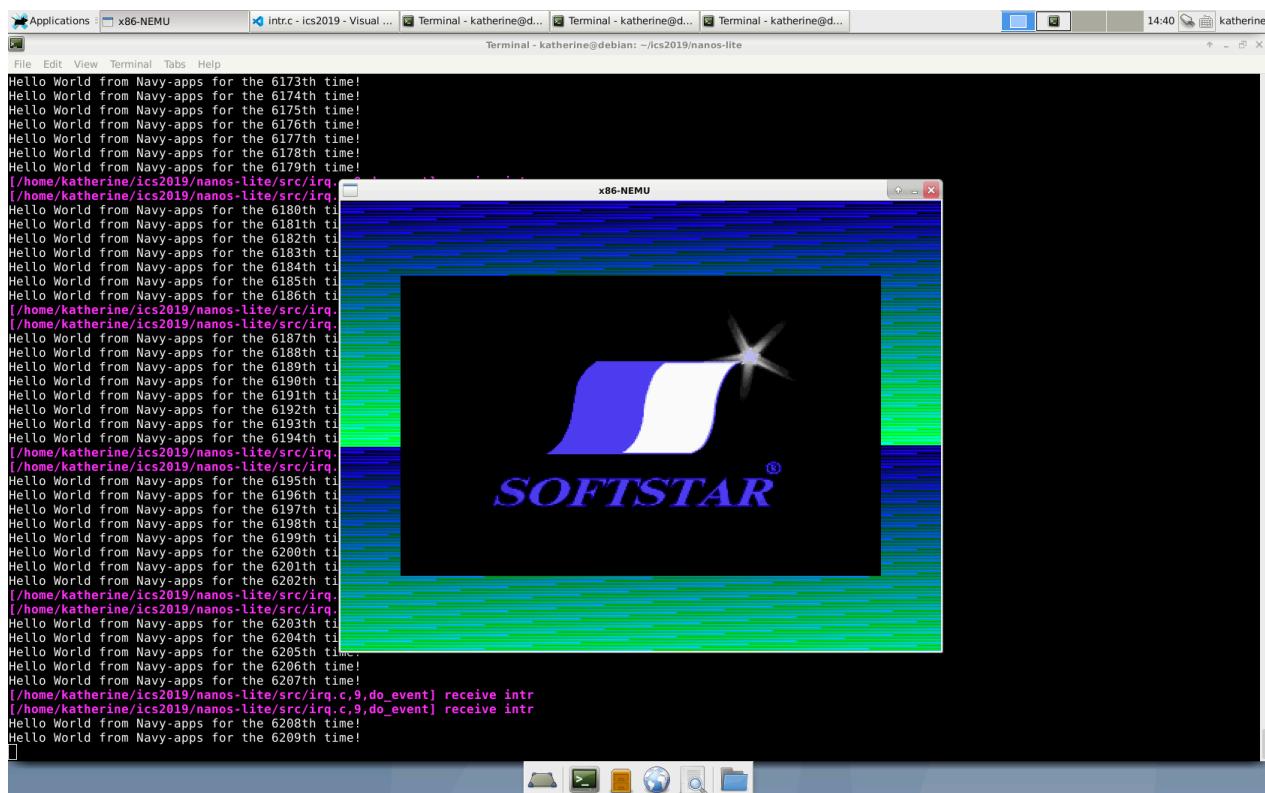


## 来自外部的声音

实现抢占多任务

按照讲义描述，首先在cpu中添加bool变量INTR，接着在 `dev_raise_intr()` 中将INTR变量初始化为1，接着在 `exec_once()` 的末尾添加轮询引脚的代码。然后是实现 `isa_query_intr()` 函数。if条件我写的是cpu的INTR和cpu.eflags.IF同时为1，`raise_intr()` 的另一个参数我写的是cpu.pc。然后继续按照讲义内容修改代码，这也没什么难度。

一开始我的if条件写的是INTR为1，然后在跑的时候有报错，后来我想了一下才改对的。我用log输出 `receive intr`，可以看到确实收到了 `_EVENT_IRQ_TIMER` 事件。



## 编写不朽的传奇

### 展示你的计算机系统

emm我觉得我的仙剑有点点卡，所以我添加了一个静态变量count，用来统计次数，并且在count达到0xff时清零，并将进程切换为hello，这样可以保证大部分进程都分配给除了hello的其他用户进程。为了添加前台程序，我设置了一个静态变量fg\_pcb，用它的编号来表示前台程序。

我在schedule函数所在的文件中编写了一个函数 `set_pcb_id()`，用来设置fg\_pcb的值。然后转到events\_read()函数中添加判断。在这里有一个题外话，因为我用的是带**touch bar**的**MacBook Pro**，所以它没有**F1, F2, F3**键.....所以我分别把三个程序关联到了数字键1, 2, 3。在函数events\_read()中，在获取键码以后，我添加了对键的判断，如果是关联的键，就调用函数 `set_pcb_id()`，设置fg\_pcb的值。同时修改之前的调度代码，把pcb数组的下标改成fg\_pcb即可。

最终实现的效果就是可以同时运行四个进程，默认显示第一个仙剑奇侠传，并且可以按数字键1, 2, 3来切换前台程序的值，所以我可以同时玩三个不同进度的仙剑奇侠传并且在之间任意切换。并且由于我修改了一点调度代码，把大部分进程分给了仙剑奇侠传，所以它的速度稍微变快了一点。

## 写在最后

不知不觉，整个PA到这里就结束了。回顾整整一个学期，我已经不记得有多少次因为找不到bug而濒临崩溃，又有多少次因为终于找到了bug而在半夜激动地睡不着。讲义上说完成的平均时间一共是140小时，但我觉得我真正花在上面的时间远远不止这些。

开始PA之间，可以说我对计算机系统几乎一无所知，但现在的我已经对整个计算机系统的运作有了一个非常全面的认识和了解了，这是几个月之前的我所远不能想象的。

随着PA的推进，我越来越清晰地感觉到，找到一个bug所需要的代价加大了。为了完成一个任务，或者添加一个新功能，我需要修改的文件越来越多，也需要AM，NEMU，nanos-lite之间的协同配合，真的是“牵一发而动全身”。同时，在有了抽象以后，一切又是那么简单和有规律。

因为时间原因和水平有限，我并没有完成太多选做和思考题。我会利用寒假继续往下思考，并且尽量完成PA5。如果有机会二周目的话，我相信一定会有更多更为深刻的收获。

最后，非常感谢屈道涵助教、胡俊豪同学和赵逸凡同学在我困难时候给予我的帮助，感谢周围的小伙伴在我绝望的时候给予我的鼓励，让我能坚持到这里。同时，感谢我自己因为对于PA发自内心的兴趣而一学期的坚守。

那么，PA，再见啦。