实验六：使用时钟中断实现进程切换

实验目的

1. 使用时钟中断实现进程切换
2. 重写操作系统，进入保护模式，并尽可能吸收现代linux系统的架构特点，实现一个易读性强，耦合性低的操作系统。

本次实验说明

在期中的时候，我打算进入保护模式，但是那时候刚好逢实验六的DDL快到了，想到我前一个实验已经做完了进程切换，实验六再做也就没意思了，于是就打算进入保护模式试一下，没想到一试就试了这么久。拖了这么久的DDL，实在是抱歉！！

在实现本次实验的过程中，由于对之前自己实验一-实验四写的代码不太满意，又看了一些关于linux内核源码的书籍，萌生了重写一遍操作系统的想法。这一次的操作系统，尽可能做到注释清晰，层次模块分明，以便于以后自己往里面增加功能。对于一些文件的组织，代码的编写思路，参考了一些linux内核源码的设计，同时经过了自己的消化整理，在自己的操作系统中实现了类似的功能。

启动流程说明

进入内核前

由于保护模式下，所有BIOS中断都不能使用，因此在进入保护前，必须将完整的内核都事先从软盘读进内核中。理解这一点，对理解下面的说明至关重要。

编写boot的代码的思路，主要来源于linux-0.11的源码与orange的源码，以下做进一步细节的说明。

1. BOOT阶段（加载所需的扇区）
    1. 将内存0x7C00-0x7DFF复制到0x90000至0x901FF处
    2. 将第2-5个扇区（Setup程序）加载到0x92000处
    3. 将一整个系统内核（0x30000字节）加载到0x10000处
    4. 跳转到Setup程序中
2. Setup程序（进入保护模式）
    1. 将原本加载到0x10000的整个系统内核复制到0x00000处。
    2. 加载段描述符，打开A20地址线，初始化8259A中断控制器
    3. 写cr0寄存器，进入保护模式
    4. 跳入地址0x00000处（系统内核中head.asm）
3. Head.asm
    1. 本文件位于系统内核前0xB000字节
    2. 初始化全局段描述符，中断描述符
    3. 开启分页机制
    4. 进入内核main()函数。

> 关于开机流程，我想最值得讨论的就是开机时代码的各种移动了。我在以前的操作系统中，受限于原来加载在0x7c00的引导程序，没能好好的利用这片空间，对于内核代码的组织，也没有向linux-0.11的组织这么清晰。linus的做法，虽然在大量的复制中耗费了比较多的时间，不过花费的时间在之后换来的是开发过程的便捷，同时还有内存空间的充分利用，我想这样的行为是值得的。

关于这一部分的代码，由于比较多汇编代码较难说明，同时这也不是在学习操作系统中的核心内容，因此不做过多说明。

进入内核后

在进入内核后，程序将会跳转到 `init/main.c` 中进行各模块的初始化的工作，然后进入 `main_test` 函数进行各模块的测试。

```
#include <test/test.h>
#include <chr_drv/tty_drv.h>
#include <intr/interrupt.h>
#include <proc/process.h>

int main(){
    tty_init();
    interrupt_init();
    process_init();

    main_test();
    return 0;
}
```

`main_test` 函数定义在 `/test/test.c` 中，在编写此操作系统的时候，我编写了五个测试，便于自己在修改了代码之后对各模块进行测试。具体的测试代码我就不放在报告上了，这并不是报告的重点。

操作系统整体设计

整体架构说明

1. 目前 `wyf-os` 内核中主要由三个模块构成：
    1. `tty` 模块，
    2. `interrupt` 模块，
    3. `process模块` 。
2. 除去以上模块， `wyf-os` 内核中还有以下两类公共函数。
    1. `common` 文件夹下的公共例程，主要为字符串操作，内存复制等常用操作
    2. `protect.h` 定义的公共例程，主要为保护模式下特有的设立中断门，调用门等操作

全局变量说明

在 `global.h` 中，描述了在本系统中用到了所有全局变量的名称

```
/**
 * @brief 终端设备表，定义在tty_drv.c中
 *
 */
extern struct tty_struct g_tty_table[1];
/**
 * @brief 页目录表，定义在head.asm中
 *
 */
extern page_dir_entry_t g_page_dir[1024];
/**
 * @brief 中断描述符表
 *
 * 中断描述符表，总共256项，定义在head.asm中
 */
extern desc_table_t g_idt_table;
/**
 * @brief 全局描述符表
 *
 * GDT，全局描述符表，定义在head.asm中
 */
extern desc_table_t g_gdt_table;
/**
 * @brief 进程控制块表
 *
 * 定义在process.h中
 */
extern proc_task_struct_t g_pcb_table[MAX_PROCESS_NUM];

/**
 * @brief 当前进程指针
 *
 *
 * 该为指向当前进程的进程控制块的指针
 */
extern proc_task_struct_t * g_cur_proc;

/**
```

```
 * @brief 当前进程内核栈地址
 *
 *
 * 用于恢复上下文
 */
extern proc_regs_t * g_cur_proc_context_stack;
```

标识符命名风格

在本次操作系统的开发过程中，我发现一个良好的命名风格能够很大程度的提高操作系统的开发效率。因此，我对自己的代码做出了约定。

标识符类型	标识符规定	样例
类型名	modulename_name_t,以模块名为首，_t结尾	proc_tack_struct_t
全局变量	以g_modulename_前缀开头,取global前的字符g为标志	g_pcb_table
模块对外接口	modulename_feature,以模块名做前缀	tty_write()
模块内部函数	_modulename_feature,注意下划线开头表示私有	_tty_queue_put()
公共例程	以com_开头	com_print

注释风格

整体注释风格采用 `doxygen` 生成文档的格式，由于该注释的标准定下来较晚，因此有部分没有这样的注释，会在后期进一步补充。

各模块说明

公共例程

保护模式下的描述符操作

在 `include/protect.h` 中写好了对全局中断向量表进行操作的工具函数，于是在内核就可以通过简单的调用 `set_intr_gate` 来对中断向量表进行设置。

```
void set_intr_gate(uint32_t n, void * addr){
    _set_gate(n, DESCTYPE_INT, addr, __KERNEL_CS);
}

void _set_gate(uint32_t gate, uint32_t type, void * addr, uint8_t seg){
    uint32_t a,b;
    _pack_gate(&a, &b, (uint32_t)addr, seg, type, 0);
    _write_idt_entry(g_idt_table, gate, a, b);
}
```

常用操作

在 `common` 文件夹下，我定义了一些常用的操作，如 `com_strlen` , `com_strcpy` 等操作。这些操作的实现并不是重点，因此不加详细描述。

字符设备的设计

在实现该字符设备的过程中，《Linux-0.11 源代码完全注释》一书给予了我很多的帮助，其中关于字符设备的思想，也是从这本书中对终端的设计而来。

linux操作系统的设计者将所有设备，文本等都抽象成一个文件的形式，我们在外界调用其 `write()` 函数，就能够使用该设备的功能，或者向文件写入内容。在这个的基础上，linux实现了一个虚拟文件系统。虽然我还没有实现该虚拟文件系统，不过我也尽可能仿照了这种设计的哲学，将我的终端抽象成一个字符设备，外界的 `com_printk` 函数在往内核终端输出信息时，只需要调用终端提供的 `write()` 函数即可。

字符设备相关的数据结构

在前面提到，内核提供的字符设备tty，以 `tty_struct_t` 形式的数组作为全局变量被访问。 `tty_struct_t` 的结构如下代码所示。在 `tty_struct_t` 还具有一个写队列，写队列中的结构也在这里给了出来。该写队列的作用是：临时缓存需要写到控制台上的字符，当中断内部调用 `write` 的时候,就会将内容写到终端并且将该队列中已输出的部分清空。

> 注意：tty_struct_t中的write成员为函数指针，需要在初始化tty过程将该函数初始化。由于不同方法实现的 `console` write函数的实现可能不一致，这样的设计给开发者开发功能更加强大的 `console` 带来了更多的方便之处。

```
typedef struct tty_struct{
    void (*write)(struct tty_struct * tty);
    // tty_queue_t read_q;
    tty_queue_t write_q;
    // tty_queue_t secondary;
}tty_struct_t;

typedef struct tty_queue {
    uint32_t data;
    uint32_t head;
    uint32_t tail;
    char buf[TTY_BUF_SIZE];
}tty_queue_t;
```

关于队列，这是一个数组环状队列，定义在文件 `tty_drv.c` 中，由于队列的使用较普遍，并且大家都清楚队列的功能与实现，节省篇幅起见，这里并不细说。

`com_printk` 函数执行流程

为了能够更加清晰的说明字符设备的设计，我通过一个输出的实例来说明。以下主要描述在调用 `com_printk` 函数的时候是如何使用字符设备输出的。

下面是 `com_printk` 函数的实现，可以看见，先使用 `com_vsprintk` 函数获取格式化后的字符串，然后将字符串以及该字符串的长度作为参数传递进 `tty_write` 函数中，在传参的时候才传入了0，告诉该 `tty` 应使用0号tty来进行输出。.

```
int com_printk(char * fmt, ...){
    va_list args;
    int i;
    va_start(args, fmt);
    // buf是定义来printk.c内的一个内部使用的缓冲区
    // 用于暂存字符串
    i = com_vsprintk(buf, fmt, args);
    // 将长度为i的buf字符串输出到0号终端中。
    tty_write(0, buf, i);
    return i;
}
```

下面进一步解释 `tty_write` 函数的实现，在前面先根据中断号，从全局的 `g_tty_table` 中获得活动tty结构体，然后尝试将 `buf` 内的字符放入队列中。如果队列已满，或者 `buf` 已经完全放入队列中，就执行 `tty->write(tty)` 函数将队列中的内容显示到终端上，并且清空输出队列，再次查看 `buf` 是否已经完全输出。可以发现，这个函数的关键，在于对 `tty->write_q` 输入队列的操作以及函数 `tty->write` 。

> 在这里，整个tty的数据和功能被分割成一个输入队列 `tty->write_q` 与一个可变的函数指针 `tty->write` ,相当于将原本为一个整体的tty模块进一步解耦合，缓冲区实现后，再保证 `tty->write` 的语义为将队列中的元素写入中断即可，不需要关注内部的实现细节，内部的实现细节也可以很容易通过更换模块来实现。

```
int tty_write(uint32_t channel, char * buf, uint32_t nr)
{
    tty_struct_t * tty = channel + g_tty_table;
    uint32_t cur_char_index = 0;
    // TODO:终端号的范围
    if (channel>2 || nr<0) return -1;

    while (cur_char_index < nr) {
        // TODO:将缓冲区内的字符放入tty的缓冲区中，并将tty内没输出的字符给输出了。
        while(!_tty_queue_is_full(&tty->write_q) && cur_char_index < nr){
            _tty_queue_put(&tty->write_q, buf[cur_char_index]);
            cur_char_index++;
        }
        tty->write(tty);
    }
    return cur_char_index;
}
```

关于 `tty->write(tty)` 在这里到底调用了什么函数，我们可以在 `g_tty_table` 结构体数组的初始化中，找到我对该tty初始化的函数指针。

```
tty_struct_t g_tty_table[1] = {
    {
    _console_write,
    {0,0,0,""},
    }
};
```

这里的 `_console_write` 函数，定义在文件 `console.c` 中。该函数的关键实现如下代码所示。其中 `display_ptr` 为当前显存地址，我可以通过向该显存写字符来实现输出。同时，该代码中的 `x` , `y` 可控制光标的位置。最终，我成功的将队列中的字符写到了显存中，实现了字符设备的功能。

```
    while(!_tty_queue_is_empty(&tty->write_q)){
        char c = _tty_queue_get(&tty->write_q);
        display_ptr = (char *)(video_mem_start + x*video_size_row + y*2);
        *display_ptr = c;
        // 颜色
        *(display_ptr+1) = 0x0007;
        y++;
        if (y >= video_num_columns){
            y = 0;
            x++;
        }
        // TODO:x行数超出限制需要滚屏
    }
```

在本字符设备的设计中，本着开发效率之上，保证结构完整性的原则，实现了一个鲁棒性不高的字符设备，在此后还可以为该字符设备增加很多功能性的升级，如对滚屏的支持，或者增加输入队列，结合键盘驱动实现一个对用户友好的 `console` 。

中断系统的设计

在设计中断系统的过程中，李云华编著的《独辟蹊径评内核：Linux内核源代码导读》中的中断部分给了我很大的启发。以下设计中的许多我认为的亮点，很多都是来自于这本书上对Linux-2.6版本的内核的中断系统的介绍。

通用中断处理例程的设计

在文件 `kernel/intr/interrupt_table.asm` 中，我对256个中断处理成进行了如下的初始化。在源代码中，我使用了nasm宏指令帮助生成代码，并在下面放了理论上预处理后的代码帮助理解。注意，为了保证每一个中断处理例程刚好4个字节，在后面还使用了nop指令用作对齐凑整之用。

```
interrupt_table:
%assign i 0
%rep 256
push i
jmp common_handler
nop
%assign i i+1
%endrep

;/////////////预处理后/////////////////////
....
;interrupt_30:
    push 30
    jmp <common_handler>
```

```
    nop
;interrupt_31:
    push 31
    jmp <common_handler>
    nop
....
```

这样处理的好处，是可以使用一个公共的中断处理例程，使用函数参数来得到该中断的号码，并且做出相应的处理。注意，下面还有部分保存环境的细节需要知晓，此部分放在进程切换中描述，这里均需要知道，此时push进栈里的所有数值，在 `_interrupt_handle` 中都能通过参数中的结构体来获取。

```
common_handler:
    SAVE_ALL
    mov eax, esp
    push eax
    call _interrupt_handler
    pop eax
    jmp ret_from_intr
```

在文件 `kernel/intr/intrrupt.c` 中我定义了 `_interrupt_handler` ,该函数接受一个 `proc_regs_t` 结构体类型的指针作为参数。该参数，来自于 `common_handler` 中的第四行 `push eax` 。该 `eax` 为当时栈的栈顶，将这个指针传进去，也就是说，在保存环境过程中存放在栈中的数据，与 `proc_regs_t` 结构体的格式应该是一致的。因此，在下面代码中，我就直接使用了 `regs->orig_eax` 来获得之前push进栈中的中断号并根据该中断号，调用相应的中断处理例程。

```
void _interrupt_handler(proc_regs_t * regs){
    // 将上下文保存到当前进程控制块中。
    _update_current_process_context(regs);
    uint32_t v = regs->orig_eax;

    switch (v){
        /* 时钟中断 */
        case 0x20:{
            irq0_clock_handler();
            break;
        }
        case 0x66:{
            irq0_clock_handler();
            break;
        }
        default:{
            com_printk("in the %d interrupt!", v);
            break;
        }
    }
}
```

irq线对应的中断处理例程

关于这里的时钟中断处理例程，我是如下实现的(可见文件 `kernel/intr/clock.c`)。其中 `proc_schedule` 为进程调度函数，该函数的实现在下一部分会详细说明。

```
void irq0_clock_handler(){
    proc_schedule();
    _basic_outb(INT_M_CTL, EOI); // 向主8259A发送EOI
}
```

进程模块的设计

关于进程模块的设计，最重要的是思考清楚如何保存现场与恢复现场。关于这点，在不同的书上的实现差异都很大，并且很多都为了效率，不惜一切代码魔改代码（毕竟进程切换可能是操作系统中最频繁的操作了）。这一部分的实现，我从以前看的《orange's 一个操作系统的实现》中得到些启发，结合我现有的对中断的实现，写出了一个在我的操作系统上可以正常运行双内核进程切换的版本。以下是详细说明。

两个全局指针

在进程的设计中，有两个全局的指针是得以成功实现的关键，该说明在前面有出现过，这里仅放关键的两个，可见 `include/global.h` 。

```
/**
 * @brief 当前进程指针
 *
 *
 * 该为指向当前进程的进程控制块的指针
 */
extern proc_task_struct_t * g_cur_proc;

/**
 * @brief 当前进程内核栈地址
 *
 *
 * 用于恢复上下文
 */
extern proc_regs_t * g_cur_proc_context_stack;
```

1. `g_cur_proc` 指针，始终指向当前活动进程的进程控制块
2. `g_cur_proc_context_stack` 指针，在恢复进程上下文事，该指针一定会指向所需恢复进程的内核栈，内核栈中存有了该进程之前push的上下文信心。

在有了 `g_cur_proc_context_stack` 指针后，之后的保护现场和恢复现场就很容易完成了。

保护现场与恢复现场

在保护现场阶段，会将所有寄存器都push到该进程的内核栈中。其中 `SAVE_ALL` 宏指令展开后就是所有寄存器push到内核栈中。

```
common_handler:
    SAVE_ALL
    mov eax, esp
    push eax
    call _interrupt_handler
    pop eax
    jmp ret_from_intr
```

在恢复现场阶段，从全局变量 `g_cur_proc_context_stack` 处获得所需恢复进程的内核栈地址，然后恢复。注意下面的一行 `add esp, 4` 主要是用来清掉曾经push 的中断号（这里曾经导致我de了半天，后来才找到我原来在这里漏了）

```
ret_from_intr:
_proc_restart:
    ; TODO:获得新进程的内核栈指针
    mov esp, [g_cur_proc_context_stack]
    RESTORE_ALL
    add esp, 4
    iret
```

进程调度

有关进程调度的代码可见 `kernel/proc/schedule.c` 。进程的调度通过修改两个全局指针即可完成，同时这样的实现保证了未来的拓展性。

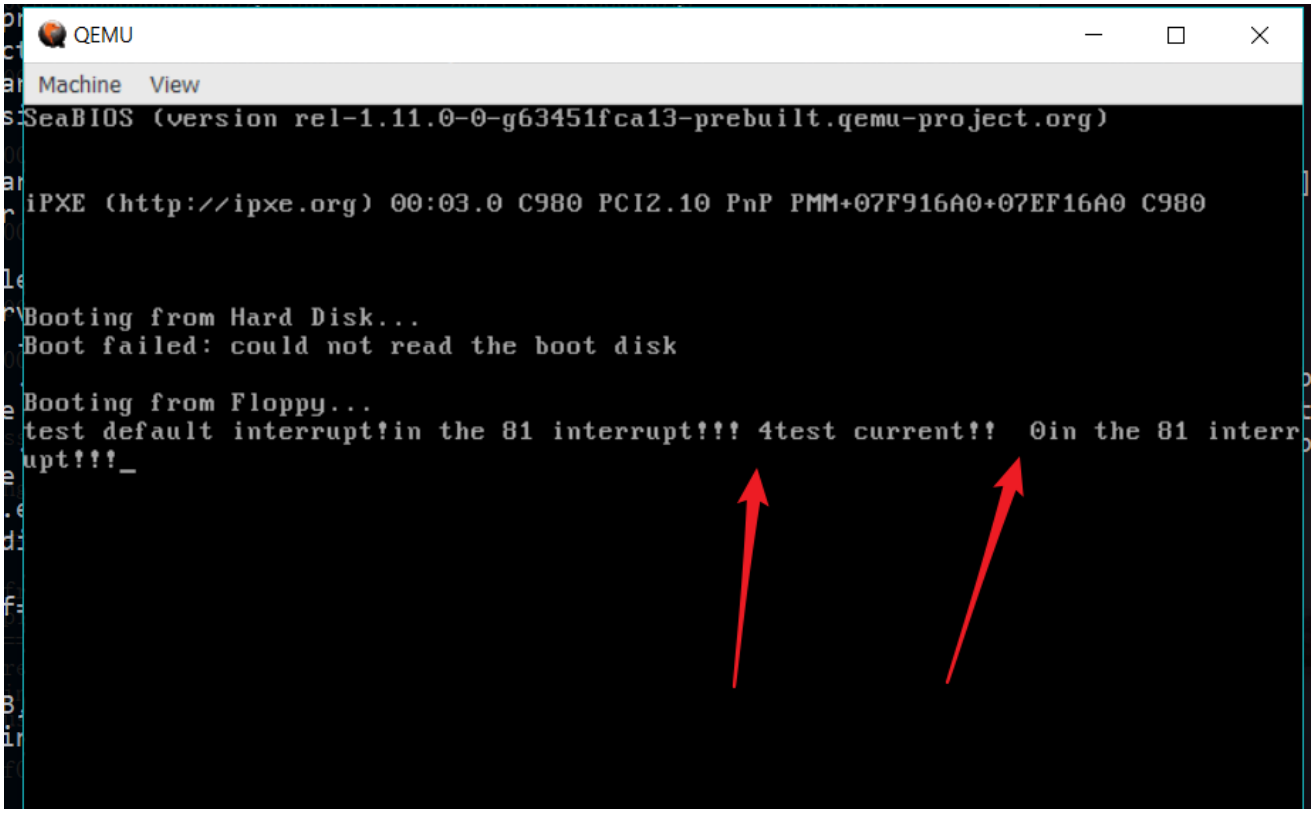
```
void proc_schedule(){
    int num = g_cur_proc - g_pcb_table;
    // FIXME:硬编码了当前进程数量
    int next_num = (num + 1)  % 2;
    g_cur_proc = &g_pcb_table[next_num];
    g_cur_proc_context_stack = g_cur_proc->kernel_stack;
}
```

遇到的问题记录

这一个是我坑的比较惨的一个坑，C语言中，传参的指针会失去大小信息

```
void update_current_process_context(PT_REGS * regs){
    ...
    memcpyk((char *)regs, (char *)&current->regs, sizeof(regs));
    ...
}
```

我有一个函数一开始是这么写的，但是一直没有成功，后来打印了 `sizeof(regs)` 发现结果是4



终于才明白过来，不能够这么写，必须使用类型名。

```
void update_current_process_context(PT_REGS * regs){
    ...
    memcpyk((char *)regs, (char *)&current->regs, sizeof(PT_REGS));
    ...
}
```

关于push 的字节数

在用于保存寄存器映像的结构体中，我一开始使用以下的语句，总是不能够正确的访问到相应的值。

如，以下语句

```
pt_regs->orig_eax
```

我会得到其他的值，而不是我想要的orig_eax。

后来经过调试，发现在进行push的时候，无论是16位的寄存器还是32位的寄存器，都会往栈中push32位的数据，当然可能会补0，而当时我的结构体如下所示。部分 `uint16_t` 是罪魁祸首，它们其实在栈中也是32位的。后来我把下面的16位的数据都改成了32位，问题就解决了。

```
/* 用来保存CPU的上下文信息 */
// NOTICE:在push的时候，无论是普通寄存器还是段寄存器，push都是32位
typedef struct pt_regs{
    uint32_t ebx; // 0x00
    uint32_t ecx; // 0x04
    uint32_t edx; // 0x08
    uint32_t esi; // 0x0c
    uint32_t edi; // 0x10
```

```
    uint32_t ebp; // 0x14
    uint32_t eax; // 0x18
    uint16_t xds; // 0x1a
    uint16_t xes; // 0x1c
    uint16_t xfs; // 0x1e
    // 中断号
    uint32_t orig_eax; // 0x22
    // 返回地址信息
    uint32_t eip;
    uint16_t xcs;
    uint32_t eflags;
    uint16_t esp;
    uint32_t xss;
}proc_regs_t;
```

实验亮点

在这次实验中，我认为我有以下亮点。

1. 参考了很多书籍中相关的实现，并进行了自己的思考，并没有盲目的抄，而是选择其方法，思路的精妙之处为自己所用，并且抛弃了一些原本源码中编写得很不好的地方。
    1. 如字符设备的实现，公共中断例程实现中的技巧。可见前面模块说明处。
2. 操作系统虽小，不过每一个模块结构清晰，接口明确，耦合度低，注释详细，可扩展性强。在如今我已经重构的操作系统框架上，我可以很容易地进一步的利用在书本上学习到的知识，在我自己的操作系统上亲自实现并测试。
3. 进入了保护模式，并在保护模式下我成功使用了CLion+qemu+gdb进行C代码级别的调试（感谢韦博耀提供的资料），对操作系统开发的效率有着很大的提高。
    1. 

```
/*********************************************/
/* 测试内核进程的初始化和切换 */
/* 使用时钟中断中断作为进程切换的测试中断 */
/*********************************************/

void _test_clock_proc1() {
    while (1) {
        for (int i = 0; i < 10000000; i++);
        com_print("pid : 1");
    }
}

void _test_clock_proc2() {
    while (1) {
        for (int i = 0; i < 10000000; i++);
        com_printk("pid : 2");
    }
}


void _test5() {
    _init_a_process(0, "test1", 1, _test_clock_proc1, (proc_regs_t *)0x20000);
    _init_a_process(1, "test2", 2, _test_clock_proc2, (proc_regs_t *)0x30000);
```

实验感想

这一次实验过程蛮曲折的，本来实验四/五那一部分已经写好了保护现场和进程切换，并且这个框架也搭建的差不多，感觉后面的实验再做难度也不会太大，但是总觉得缺了一些什么东西。在之前写操作系统的时候，很多时候遇到了问题，并没有去进一步的思考系统的可读性，扩展性，写到进程切换的时候，甚至萌生了一些不敢调试的想法（毕竟是自己写的代码，有点可怕)。看着这个框架，有种想重构的想法，又想到进保护模式，在32位的代码中可以使用C代码级别的调试，下了决心要进入保护模式，尝试一下。也可以趁此机会，将操作系统写得更优雅一点，于是抛弃了原有的框架，重新写了一份。

除此之外，趁着这一次学习操作系统的过程，我也有着阅读linux内核源码的念头，想着以后可能再难有这样的“DDL催促着自己写操作系统”的状态，于是便在这次实验的过程中，也开始去图书馆借了不少关于linux内核的书籍，边阅读linux相关的书籍，边动手去实现。原本想着可能复制代码可能也很简单，但是直到自己开始尝试将linux的代码使用进自己的操作系统，才觉得他们代码之间的耦合关系过强（虽然这是难以避免的），并且在linus的注释中，还充斥着各种“hope there was no bug","seems to work"等字眼，看得我贼难受。

在阅读书籍的基础上吧，我进一步去思考操作系统模块之间的耦合性强的原因。可以理解的是，linux的开发过程中，遗留下来的一些历史问题，影响了整体代码的风格。不过说实话，在2.6版本的内核中，linux 的代码比linux-0.11的代码优雅很多，在看《独辟蹊径品内核：linux内核源代码导读》这本书中，我从这本书中获得了很大的启发，这对我之后内核的实现提供的很大的帮助，特别是中断系统这一部分。

此次实现暂时未碰触到保护模式的核心：特权级之间跳转与访问的限制，同时，在初始化新进程的时候，使用了硬编码的方式为新内核进程分配栈空间，因此不需要对cr3寄存器进行修改就可以实现进程的切换。这也是目前系统模块间耦合度的原因之一。在之后，也会考虑继续增加更多的模块，让我的操作系统具有更加完善的功能。

这一次操作系统，以及以前的操作系统，开发的过程都公开在了github仓库中，希望自己这个学期完成的一份简洁而不简单的操作系统能够与同学们相互交流，找到更好的实现，也让之后的学弟学妹们进一步参考与学习。

[github链接] (https://github.com/WalkerYF/my_operating_system)

参考资料

1. 《orange's 一个操作系统的实现》
2.