操作系统原理实验报告

实验六：二状态进程模型

数据科学与计算机学院　17大数据与人工智能

17341015　陈鸿峥

一、 实验目的

-

二、 实验要求

保留原型原有特征的基础上，设计满足下列要求的新原型操作系统：

- 在C程序中定义进程表，进程数量为4个。
- 内核一次性加载4个用户程序运行时，采用时间片轮转调度进程运行，用户程序的输出各占1/4屏幕区域，信息输出有动感，以便观察程序是否在执行。
- 在原型中保证原有的系统调用服务可用。再编写1个用户程序，展示系统调用服务还能工作。

三、 实验环境

具体环境选择原因已在实验一报告中说明。

- Windows 10系统 + Ubuntu 18.04(LTS)子系统
- gcc 7.3.0 + nasm 2.13.02 + GNU ld (Binutils) 2.3.0
- GNU Make 4.1
- Oracle VM VirtualBox 6.0.6
- Bochs 2.6.9
- Sublime Text 3 + Visual Studio Code 1.33.1

虚拟机配置：内存4M，1.44M虚拟软盘引导，1.44M虚拟硬盘。

四、 实验方案

本次实验进入保护模式。

由于保护模式涵盖的内容非常多，下面将叙述在本次实验中完成的部分。

1. 引导程序—由实模式到保护模式

本部分对应着`bootloader.asm`文件。

在最开始，操作系统只能进入16位的实模式，引导程序的加载与原来实模式的加载相同。

- 处理器会搜索可用的存储媒介，如软盘、硬盘、CD等

- 检查每个引导盘的有效性：最后一个字必须为0x55aa才为有效的引导盘
- 如果有效，则第一个扇区，即主引导程序(master boot record, MBR)，将会被加载入RAM地址为0000:7C00的地方（也就是物理地址0x07C00，此过程也被称为自举bootstrap）
- 通过BIOS跳转到0x7C00继续执行

在引导程序中调用写好的汇编例程load_kernel（调用BIOS int 13h中断读软盘），将内核程序加载到0x7E00的地方。

接下来我们就需要从**实模式切换到保护模式**(switch_to_pm)。

- 关中断cli
- 通过lgdt指令加载全局描述表（其含义可见4.1.1节）
- 将控制寄存器cr0的第0位设置为1（其含义可见4.1.2节）
- 通过far jump跳转入32位保护模式，此时处在内核代码段CODE_SEG

通过上面操作即可成功切换到保护模式。然后重新将所有段地址设置为内核数据段地址DATA_SEG，同时设置好栈基址ebp和栈指针esp。

引导程序的最后就可以直接call我们的内核入口地址了！即kernel_entry.asm中的_start入口。

(i) 寻址方式

保护模式下的寻址方式与实模式有很大的不同。

- 实模式：16位的段寄存器内容左移4位(10h)作为**段基地址**，加上16位**段内偏移**，形成20位的**物理地址**，实现最大寻址空间$1MB = 2^{20}B$。如

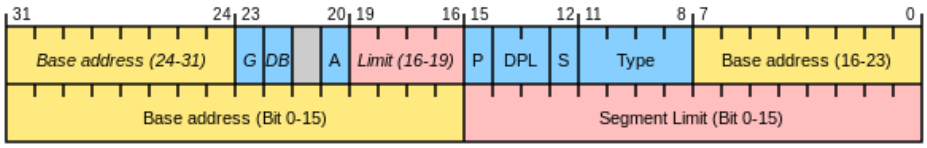
$$\text{segment:offset} = 1234h : 5678h = 12340h + 5678h = 18018h$$

- 保护模式：16位的段寄存器存储的是**段选择符**(selector)，通过访问全局描述符表(global descriptor table, GDT)中对应段描述符的内容，可以得到32位**段基地址**，再与**段内偏移**相加形成32位**线性地址**[1]

$$\text{descriptor:offset} = 10h : 5678h = gdt(10h) + 2567h$$

全局描述符表作为保护模式的核心，关系着内存地址的访问与保护，需要十分熟悉。每个描述符共64位(8B)，如下图所示

[1]由于在本实验中还未开启分页模式，故线性地址即为物理地址。



其中

- 0-15、48-51：段界限
- 16-39、56-63：基地址
- 40-43：描述符类型
  - 40：访问位（虚存使用）
  - 41：0只读（数据段）/只可执行（代码段）；1可读可写/可读可执行
  - 42：扩展方向
  - 43：0数据段，1代码段
- 44：0系统描述符，1代码或数据描述符
- 45-46：特权级(ring)，0内核态，3用户态，1-2系统态
- 47：段是否在内存中（虚存使用）
- 52：保留位(OS)
- 53：保留位(0)
- 54：段类型，0为16位，1为32位
- 55：粒度，0无，1界限为4K的倍数

我在`gdt.inc`中定义了初始进入保护模式时的三个描述符，按照上面每个位的含义填写并加载

- `gdt_null`(0x00)：全零描述符
- `gdt_code`(0x08)：代码段描述符
- `gdt_data`(0x10)：数据段描述符

(ii) 控制寄存器

x86处理器除了状态寄存器`eflags`、`eip`，还有4个32位的控制寄存器，它们都保存着全局与任务无关的机器状态。

- `cr0`：包含6个预定义标志，其中**第0位**是我们需要关注的

标志位	含义
0(PE)	**保护模式**
1(MP)	监控协处理器
2(EM)	仿真协处理器
3(TS)	任务切换
4(ET)	协处理器类型(80287/80387)
5	未用
6(PG)	内存分页

- `cr1`：未定义的控制寄存器，未来使用
- `cr2`：页故障线性地址寄存器，保存最后一次出现页故障的全32位线性地址
- `cr3`：页目录基址寄存器，保存页目录表的物理地址

2. 硬件抽象层

硬件抽象层(Hardware Abstraction Layer, HAL)通过将底层的硬件设施进行封装，而只留下上层的API接口，使得通过C语言也可以很方便地进行底层操作。

通过汇编入口(`kernel_entry.asm`)[2]进入内核后，直接跳转入C语言编写的内核(`call main`)。由于保护模式下BIOS都无法使用，故进入内核后第一件事则是初始化各种硬件抽象。

(i) 全局描述符表(GDT)

由于进入了C的内核，原来在`bootloader.asm`中定义的GDT已经无法定位也很难使用，故需要重新建立GDT并加载。

用C的结构体可以使得GDT的定义比较清晰，如`gdt.h`中所示

```
struct gdt_descriptor {

    // bits 0-15 of segment limit
    uint16_t      limit;

    // bits 0-23 of base address
    uint16_t      baseLo;
    uint8_t       baseMid;

    /*
     * descriptor access flags
     * | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
     * | P | DPL  | S |     TYPE     |
     * P: Present in memory or not
     * DPL(Descriptor Privilege Level): ring 0 - ring 3
```

[2]在该汇编入口程序中，还定义了其他基本的设施，方便后面HAL的调用。

```
     * S: 1 - Data/Code descriptor | 0 - gate descriptor
     * Type: When S = 1
     *    3: Executable
     *    2: Consistent
     *    1: 1 - Read+Write | 0 - Only read
     *    0: Accessed
     */
    uint8_t        flags; // access

    /*
     * grand
     * | 7 | 6  | 5 | 4  | 3 | 2 | 1 | 0 |
     * | G | D/B | 0 | AVL | Limit high |
     * G: 0 - B | 1 - 4KB
     */
    uint8_t        grand; // limit_high, flags

    // bits 24-32 of base address
    uint8_t        baseHi;
} __attribute__((packed));
```

注意最后添加(packed)的编译指令阻止编译器对该结构体进行对齐优化。通过下面的API可以方便地设置GDT。

```
void gdt_set_descriptor(uint32_t i, uint64_t base, uint64_t limit, uint8_t access,
    ↪  uint8_t grand)
```

初始化以下6个段描述符

- SEG_NULL(0x00)：全零描述符
- SEG_KER_CODE(0x08)：内核代码段描述符
- SEG_KER_DATA(0x10)：内核数据段描述符
- SEG_USER_CODE(0x18)：用户代码段描述符（设置为ring3）
- SEG_USER_DATA(0x20)：用户数据段描述符（设置为ring3）
- SEG_TSS(0x28)：TSS段描述符（见4.2.2节）

由于重新加载了GDT，故要重新加载入GDT寄存器，其定义如下

```
struct gdtr {

    // size of gdt
    uint16_t       m_limit;

    // base address of gdt
    uint32_t       m_base;
```

```
} __attribute__((packed));
```

汇编接口load_gdt如下，需要重新设置各个段寄存器指向内核代码段，并进行一个far jump。

```
[ global load_gdt ]
[ extern _gdtr ]
load_gdt:
    lgdt [ _gdtr ]
    mov ax, 0x10              ; kernel data selector
    mov ds, ax
    mov es, ax
    mov fs, ax
    mov gs, ax
    mov ss, ax
    jmp 0x08:load_gdt_ret
load_gdt_ret:
    ret
```

(ii) 任务状态段(TSS)

保护模式之所以能够保护就是因为它设立了4个不同的特权级

- ring0：内核态，权限最高
- ring1：系统态
- ring2：系统态
- ring3：用户态，权限最低

当需要从高的特权级向低的特权级转换时，就需要结合任务状态段(Task State Segment, TSS)进行切换。 TSS包含了寄存器、局部描述符表等信息，如下所示。

```
struct tss_entry {
    uint32_t prevTss;
    uint32_t esp0;
    uint32_t ss0;
    uint32_t esp1;
    uint32_t ss1;
    uint32_t esp2;
    uint32_t ss2;
    uint32_t cr3;
    uint32_t eip;
    uint32_t eflags;
    uint32_t eax;
    uint32_t ecx;
    uint32_t edx;
```

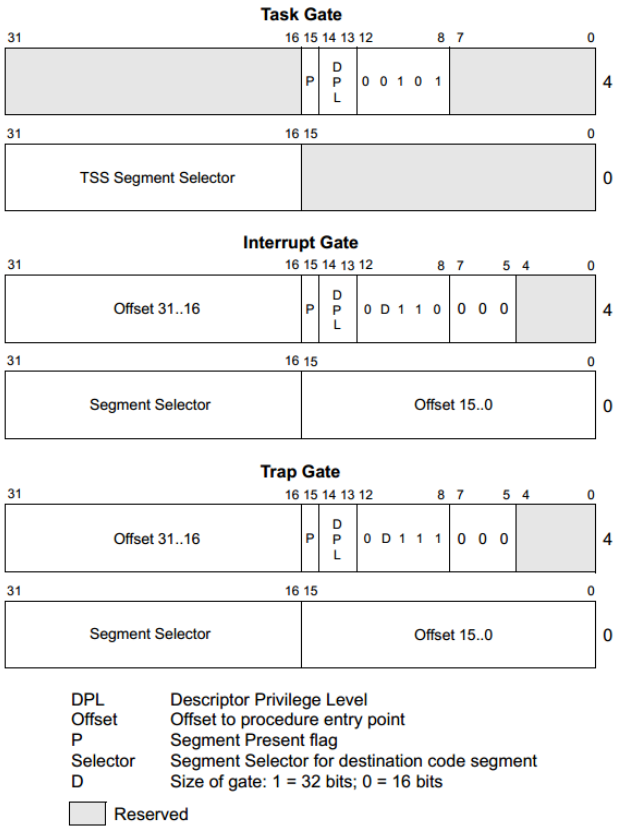
```
    uint32_t ebx;
    uint32_t esp;
    uint32_t ebp;
    uint32_t esi;
    uint32_t edi;
    uint32_t es;
    uint32_t cs;
    uint32_t ss;
    uint32_t ds;
    uint32_t fs;
    uint32_t gs;
    uint32_t ldt;
    uint16_t trap;
    uint16_t iomap;
} __attribute__((packed));
```

在GDT中定义了TSS的入口后，还需通过`ltr`将其加载入寄存器（`tss.h`中定义）。

(iii) 中断描述符表(IDT)

实模式中我们采用中断向量表(Interrupt Vector Table, IVT)给出中断处理程序的入口。而在保护模式中我们则采用中断描述符表(Interrupt Description Table, IDT)给出中断处理程序的入口。

下图从Intel Developer's Manual Volume 3, Chapter 6中截取。



对应的，可以给出C的结构体

```
struct IDT_entry {
    unsigned short int offset_lowerbits;
    unsigned short int selector;
    unsigned char zero;
    unsigned char type_attr;
    unsigned short int offset_higherbits;
};
```

同时给出API接口

```
int install_ir (uint32_t i, uint16_t type_attr, uint16_t selector, uint64_t irq)
void setvect (int intno, uint64_t vect)
```

类似于GDT，IDT也有自己的寄存器，需要在初始化时进行加载。

```
; extern void load_idt(unsigned long *idt_ptr); // C function
[ global load_idt ]
load_idt:
    mov edx, [ esp + 4 ]
    lidt [ edx ]      ; load interrupt description table (IDT)
    sti               ; turn on interrupts
    ret
```

加载完IDT后就可以对一些常用的中断号进行设置。由于是面向x86处理器的操作系统，故采用Intel CPU给出的中断号，如下图所示。

Vector	Mnemonic	Description	Type	Error Code	Source
0	#DE	Divide Error	Fault	No	DIV and IDIV instructions.
1	#DB	Debug Exception	Fault/ Trap	No	Instruction, data, and I/O breakpoints; single-step; and others.
2	—	NMI Interrupt	Interrupt	No	Nonmaskable external interrupt.
3	#BP	Breakpoint	Trap	No	INT3 instruction.
4	#OF	Overflow	Trap	No	INTO instruction.
5	#BR	BOUND Range Exceeded	Fault	No	BOUND instruction.
6	#UD	Invalid Opcode (Undefined Opcode)	Fault	No	UD instruction or reserved opcode.
7	#NM	Device Not Available (No Math Coprocessor)	Fault	No	Floating-point or WAIT/FWAIT instruction.
8	#DF	Double Fault	Abort	Yes (zero)	Any instruction that can generate an exception, an NMI, or an INTR.
9		Coprocessor Segment Overrun (reserved)	Fault	No	Floating-point instruction.[1]
10	#TS	Invalid TSS	Fault	Yes	Task switch or TSS access.
11	#NP	Segment Not Present	Fault	Yes	Loading segment registers or accessing system segments.
12	#SS	Stack-Segment Fault	Fault	Yes	Stack operations and SS register loads.
13	#GP	General Protection	Fault	Yes	Any memory reference and other protection checks.
14	#PF	Page Fault	Fault	Yes	Any memory reference.
15	—	(Intel reserved. Do not use.)		No	
16	#MF	x87 FPU Floating-Point Error (Math Fault)	Fault	No	x87 FPU floating-point or WAIT/FWAIT instruction.
17	#AC	Alignment Check	Fault	Yes (Zero)	Any data reference in memory.[2]
18	#MC	Machine Check	Abort	No	Error codes (if any) and source are model dependent.[3]
19	#XM	SIMD Floating-Point Exception	Fault	No	SSE/SSE2/SSE3 floating-point instructions[4]
20	#VE	Virtualization Exception	Fault	No	EPT violations[5]
21-31	—	Intel reserved. Do not use.			
32-255	—	User Defined (Non-reserved) Interrupts	Interrupt		External interrupt or INT *n* instruction.

在本操作系统中，没有对这些异常进行特别的处理，只是将出错的信息输出，并陷入死循环（见`exception.h`）。对于没有处理的异常，则调用默认中断处理程序，如下。

```
void default_handler () {
    put_error("Error: Unhandled Exception!");
    for(;;);
}
```

同时类似Linux将0x80设置为系统调用，供用户程序调用。

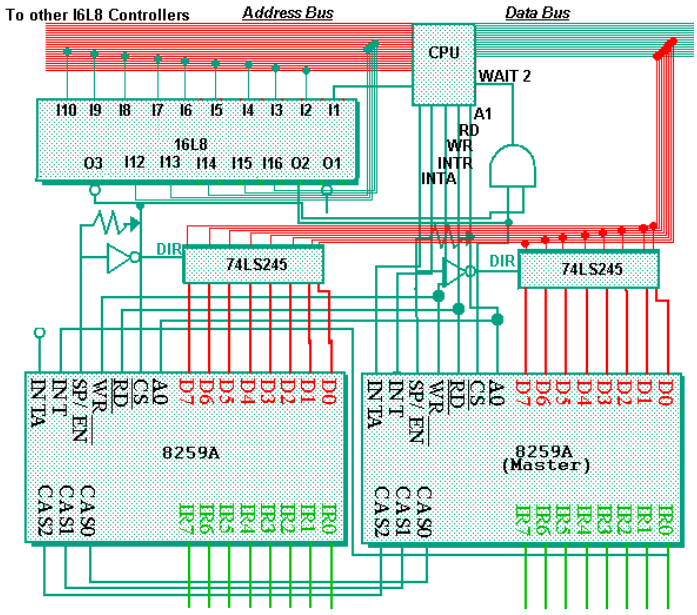
(iv) 可编程中断控制器(PIC)

8259A微控制器(microcontroller)/可编程中断控制器(Programmable Interrupt Controller, PIC)是CPU与外围设备交互的一个界面。通过提供中断处理例程(Interrupt Routine, IR)，我们就可以处理8259A发出的中断请求(Interrupt Request, IRQ)。

通过`int`指令调用的称为软件中断，而8259A产生的则是硬件中断，如下表所示。

x86 Hardware Interrupts		
8259A Input pin	Interrupt Number	Description
IRQ0	0x08	Timer
IRQ1	0x09	Keyboard
IRQ2	0x0A	Cascade for 8259A Slave controller
IRQ3	0x0B	Serial port 2
IRQ4	0x0C	Serial port 1
IRQ5	0x0D	AT systems: Parallel Port 2. PS/2 systems: reserved
IRQ6	0x0E	Diskette drive
IRQ7	0x0F	Parallel Port 1
IRQ8/IRQ0	0x70	CMOS Real time clock
IRQ9/IRQ1	0x71	CGA vertical retrace
IRQ10/IRQ2	0x72	Reserved
IRQ11/IRQ3	0x73	Reserved
IRQ12/IRQ4	0x74	AT systems: reserved. PS/2: auxiliary device
IRQ13/IRQ5	0x75	FPU
IRQ14/IRQ6	0x76	Hard disk controller
IRQ15/IRQ7	0x77	Reserved

处理器都会有自己内部的PIC微处理器，而每一个PIC只支持8个IRQ，因此大多主板(motherboard)都会包含一个二级(secondary)/从(slave)PIC微控制器。故现在大多计算机都有2个PIC，其连接方式如下图所示。



在pic.h中，我们可以将这些端口号进行抽象，更加方便编写程序。

```
// The following devices use PIC 1 to generate interrupts
#define     PIC_IRQ_TIMER         0 // IMPORTANT!
#define     PIC_IRQ_KEYBOARD      1 // IMPORTANT!
#define     PIC_IRQ_SERIAL2       3
#define     PIC_IRQ_SERIAL1       4
#define     PIC_IRQ_PARALLEL2     5
#define     PIC_IRQ_DISKETTE      6 // IMPORTANT!
#define     PIC_IRQ_PARALLEL1     7
```

```
// The following devices use PIC 2 to generate interrupts
#define    PIC_IRQ_CMOSTIMER    0
#define    PIC_IRQ_CGARETRACE   1
#define    PIC_IRQ_AUXILIARY    4
#define    PIC_IRQ_FPU          5
#define    PIC_IRQ_HDC          6
```

通过I/O端口编程，可以实现CPU和8259A的交互。我们同样可以提供一组I/O指令方便C程序调用，定义在io.h中。

```
static inline unsigned char port_byte_in(unsigned short port){
    unsigned char result;
    __asm__ volatile ("in al, dx"
                    :"=a"(result)
                    :"d"(port)
                    );
    return result;
}

static inline void port_byte_out(unsigned short port, unsigned char data){
    __asm__ volatile ("out dx, al"
                    :
                    :"a"(data), "d"(port)
                    );
}
```

而初始化的过程通过一系列的操作命令(operation command)实现。操作命令包括初始化命令字(Initialization Command Words, ICW)和操作命令字(Operation Command Words, OCW)。

初始化过程需要向8259A发送4组ICW，将重要的端口号（如时钟、键盘等）打开，如下所示。

```
void pic_init () {

    uint8_t    icw = 0;

    // disable hardware interrupts
    disable ();

    // ICW1: Begin initialization of PIC
    icw = (icw & ~PIC_ICW1_MASK_INIT) | PIC_ICW1_INIT_YES; // 00010000
    icw = (icw & ~PIC_ICW1_MASK_IC4) | PIC_ICW1_IC4_EXPECT; // 00010001

    pic_send_command (icw, 0); // 0x11
```

```
    pic_send_command (icw, 1); // 0x11

    // ICW2: remap offset address of IDT
    pic_send_data (0x20, 0); // master
    pic_send_data (0x28, 1); // slave

    // ICW3 for master PIC is the IR that connects to secondary pic in binary
        ↪ format
    // ICW3 for secondary PIC is the IR that connects to master pic in decimal
        ↪ format
    pic_send_data (0x04, 0);
    pic_send_data (0x02, 1);

    // ICW4: Enables i86 mode
    icw = (icw & ~PIC_ICW4_MASK_UPM) | PIC_ICW4_UPM_86MODE; // 00010001
    pic_send_data (icw, 0);
    pic_send_data (icw, 1);

    enable();
}
```

(v) 键盘中断

由于在PIC初始化中我们已经将主控制器的端口号映射到0x20开始的8个中断，故键盘将对应着0x21号中断，我们只需要提供键盘中断处理程序keyboard_handler_main即可。而通过I/O端口读入的为按键的扫描码（定义在scancode.h），需要将扫描码转化为ASCII码才能为我们使用。

```
void keyboard_handler_main(void)
{
    unsigned char status;
    char keycode;

    /* write End of Interrupt (EOI) */
    port_byte_out(0x20, 0x20);

    status = port_byte_in(KEYBOARD_STATUS_PORT);
    /* Lowest bit of status will be set if buffer is not empty */
    if (status & 0x01) {
        keycode = port_byte_in(KEYBOARD_DATA_PORT);
        if (keycode < 0)
            return;

        char ascii = asccode[(unsigned char) keycode][0];
        kb_char = ascii;
```

```
    }
}
```

每次检测到按键就将当前按键的扫描码转化为ASCII码，然后存入缓冲区kb_char中，结束中断。同时，我们可以提供一个API，即getchar()，检测缓冲区的内容是否合法，若合法则将其内容取出来返回，如下所示。

```
char getchar()
{
    char c = INVALID_KB_CHAR;
    kb_char = INVALID_KB_CHAR;
    while (c == INVALID_KB_CHAR)
        c = kb_char;
    kb_char = INVALID_KB_CHAR;
    return c;
}
```

由于中断处理需要采用iret返回，故实现时都提供了汇编入口，然后调用C的处理程序进行中断处理。注意在中断处理时应先关中断，然后再打开，如下面的例子所示。

```
; setvect(0x21,(unsigned long)keyboard_handler); // keyboard uses 33 interrupt
keyboard_handler:
    cli
    call    keyboard_handler_main ; C function
    sti
    iretd                         ; 32-bit return
```

有getchar()函数后，就可以实现C的标准库stdio.h。常用的函数均已实现，如下

- getchar、getline
- scanf、sscanf
- printf
- put_error、put_info
- clear_screen

其他更细的函数请直接查看源程序。光标的处理与前几次实验相同，在此不再赘述。

(vi) 可编程区间计时器(PIT)

可编程区间计时器(Programmable Interval Timer, PIT)是处理器内置的时间计数器，每进行一次计数就会触发中断，需要处理器进行处理。通过下面的API可以创建一个计时器，同时将频率设置为100Hz。

```
void pit_start_counter (uint32_t freq, uint8_t counter, uint8_t mode)
```

有了PIC的铺垫，PIT的编程就方便很多。类似于键盘的初始化，只需设置中断号和中断处理程序即可。

```
setvect (32, (unsigned long)pit_handler); // pit uses 32 interrupt
```

由于时间中断处理对于分时操作系统有着至关重要的作用，其涉及到进程的切换，故在4.4节才进行阐述。

(vii) 磁盘控制

由于BIOS被禁用了，故从磁盘读取数据依然要自己进行编程。本来想通过直接编写软盘驱动来读取数据，但尝试了多次都无法成功。后来迫不得已使用集成驱动电子设备(Integrated Drive Electronics, IDE)进行读取，缺点是需要新建一个虚拟硬盘并挂载入虚拟机，但好处是编程十分方便。

如下所示，直接传入要读的磁盘扇区号，即可通过以下几条指令对IDE/ATA端口进行编程，一次性将扇区读取出来。

```
/* readsect - read a single sector at @secno into @dst */
static void readsect(uintptr_t dst, uint32_t secno) {

    port_byte_out(0x1F2, 1);                    // count = 1
    port_byte_out(0x1F3, secno & 0xFF);
    port_byte_out(0x1F4, (secno >> 8) & 0xFF);
    port_byte_out(0x1F5, (secno >> 16) & 0xFF);
    port_byte_out(0x1F6, ((secno >> 24) & 0xF) | 0xE0);
    port_byte_out(0x1F7, 0x20);                 // cmd 0x20 - read sectors
    // wait for disk to be ready
    waitdisk();

    // read a sector
    insw(0x1F0, dst, SECTSIZE / 2);
}
```

其中，`insw`函数对应着x86汇编的`rep insw`指令，可以一次性将字符串从I/O端口缓冲区中读入对应内存地址。

这里类似于1.44M软盘的格式创建了一个同样是1.44M的虚拟硬盘（2磁头、80磁道、每道18扇区）。

(viii) 其他API

硬件抽象层还提供了其他API，如下

- `generate_interrupt`：生成中断
- `sleep`：使用PIT的计数器计数，超过指定时间才退出，进而实现等待的功能

3. 堆栈

由于堆栈在操作系统实验中非常重要，故单独开一节阐述。

(i) 函数调用

由于操作系统内核采用C和汇编混编，故函数调用有两种情况，重点需要考虑如何利用堆栈传递函数参数。

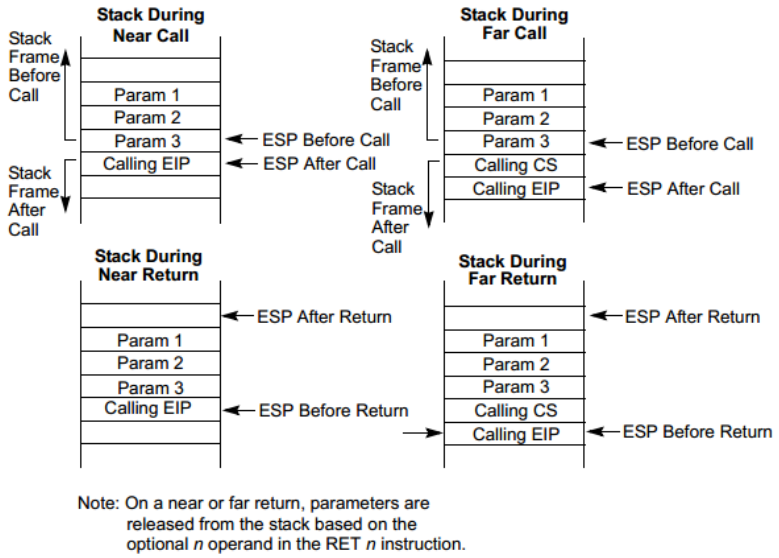
1. 汇编调用C函数：通过push将参数依次进栈，然后直接call调用C中对应函数。而C函数的参数列表从左到右依次从栈顶中读出。样例如下

```
[ extern test_function ]
push arg2
push arg1
call test_function
; void test_function(int arg1, int arg2); // C function
```

2. C函数调用汇编：函数参数从右往左入栈，在汇编中则由右往左弹出。通过加__attribute__((__cdecl__))编译指令，强制进栈次序，并且可以实现变参数传递。

```
// extern void test_function(int arg1, int arg2); // assembly
test_function(arg1, arg2); // directly call
// pop eax ; arg1
// pop ebx ; arg2
```

在保护模式中，由于存在不同的段选择子，故call和ret都有远(far)和近(near)之分。在相同段内的调用/返回则是近的，在不同段间的调用/返回则是远的。其堆栈内容如下图[3]所示。

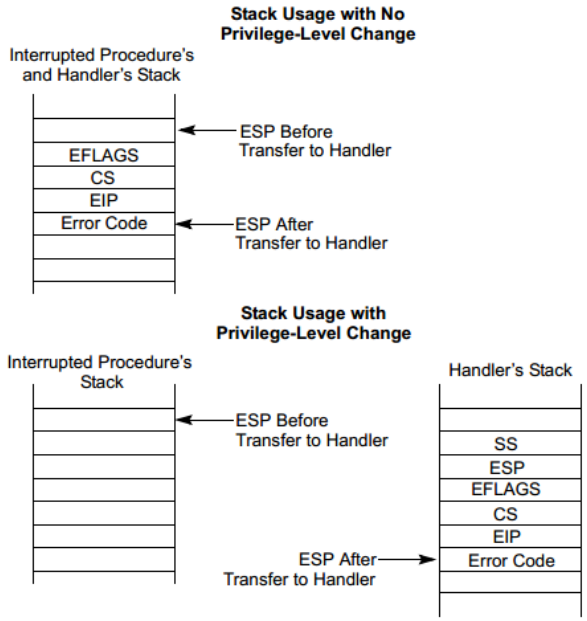


[3]Intel Manual Volume 1, Chapter 6, Procedure Calls, Interrupts, and Exceptions

(ii) 硬件中断与返回

这一节的内容同样很关键，是后面做进程切换的核心。

当产生中断/异常时，硬件会自动将`eflags`、`cs`和`eip`进行堆栈，至于有没有`error code`则要看情况。但对于保护模式来说，由于涉及特权级的切换，所以问题会多很多。如下图[4]所示，当中断处理程序和原程序特权级相同时，使用同一个堆栈。而特权级不同时，则会发生堆栈切换，参数都会被堆到新的栈中。同时`ss`和`esp`会作为最开始的元素进入栈中，方便中断处理后的返回。

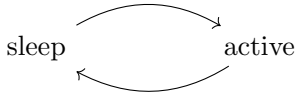


`iret`可谓是保护模式中最重要的一条指令了，它的功能如下

- 自动从当前栈顶弹出`cs:eip`，作为返回地址
- 设置新的标志位为`eflags`
- 如果发生特权级切换，还需弹出`ss:eip`，恢复原来的栈

4. 二状态进程模型

这是本次实验的核心内容。只用两个状态描述进程，即为执行和等待。在实施中我采用了active和sleep表达同样的意思。每个进程都在这两个状态间不断变换，如下图所示。



我采用了枚举体结构以确保可读性和可扩展性。

[4]Intel Manual Volume 3, Chapter 6, Interrupt and Exception Handling

```
enum PROC_STATUS
{
    PROC_SLEEP = 0,
    PROC_ACTIVE = 1
};
```

(i) 进程控制块

进程控制块(Process Control Block, PCB)是描述进程最为关键的结构，它保存了每个进程的状态。在C内核中，我采用如下方法定义。

```
struct regs {
    // segment register
    uint32_t gs; // 16-bit
    uint32_t fs; // 16-bit
    uint32_t es; // 16-bit
    uint32_t ds; // 16-bit

    // general registers, saved by pusha
    // the order should be the same
    uint32_t edi;
    uint32_t esi;
    uint32_t ebp;
    uint32_t esp; // stack
    uint32_t ebx;
    uint32_t edx;
    uint32_t ecx;
    uint32_t eax;

    // saved by int (interrupt)
    uint32_t eip;
    uint32_t cs; // 16-bit
    uint32_t eflags;
    uint32_t user_esp;
    uint32_t ss; // 16-bit
} __attribute__((packed));
typedef struct regs regs;

typedef struct process {
    regs   regImg;
    int    pid;
    int    priority;
    int    status;
    int    tick;
} process;
```

一部分是寄存器镜像regImg，另一部分则是进程的信息，如

- pid：进程的序号，唯一标识
- priority：进程优先级，用于进程调度
- status：进程状态，执行或等待
- tick：进程剩余工作时长，初始化为MAX_TICK，到时间会被自动切换

由于目前操作系统的功能还比较少，故PCB中的信息也较少，之后会逐步往里面添加东西。

同时还定义了一些全局变量：

- process proc_list[MAX_PROCESS]：进程队列，用于管理、分配、调度进程
- process* curr_proc：当前进程指针，初始化为NULL
- uint32_t curr_pid：当前的进程数，初始化为0

(ii) 进程初始化

进程初始化通过task.h中的proc_init()函数完成。设置好pid和priority后，将所有进程状态设为等待即可。

(iii) 进程创建

创建一个进程需要知道它的代码段cs、数据段ds、以及它存放的地址addr。然后分别给PCB的内容初始化。注意标志位eflags需要读出来后对0x200进行置位，以确保进入用户态后中断可用。

```
process* proc_create(uint32_t cs, uint32_t ds, uintptr_t addr)
{
    disable(); // disable interrupts
    process* pp = proc_alloc();
    pp->regImg.cs = cs;
    pp->regImg.ds
        = pp->regImg.es
        = pp->regImg.fs
        = pp->regImg.gs
        = pp->regImg.ss
        = ds;
    uint32_t flag;
    asm volatile ("pushf\n\t"
                  "pop eax\n\t"
                  :"=a"(flag)
                  :
                 );
    pp->regImg.eflags = flag | 0x200; // interrupt
    pp->regImg.eip = addr;
```

```
    pp->regImg.esp
        = pp->regImg.ebp
        = pp->regImg.user_esp
        = addr + PROC_SIZE; // reset
    reset_time(pp);
    return pp;
}
```

由于还未涉及内存分配，故这里采用了比较简单的内存管理，即直接在addr+PROC_SIZE的地方建立堆栈。

(iv) 进程切换

进程切换是进程管理的重中之重，涉及到保护现场save和恢复现场restart两个步骤。 **本次进程切换保护现场和恢复现场的代码均为自己实现，未按照老师课件上的方法。**

由于目前的操作系统采用分时(time-sharing)方式，故进程切换总发生在时钟中断的时候。

流程图如下图所示，下面将逐步进行解释。

当时钟中断到来时，会通过IDT查找到中断处理程序pit_handler。这是一个汇编入口，如下所示。

```
pit_handler:          ; handle process switching
    cli
    jmp save_proc_entry ; DO NOT USE CALL!!! WILL DESTROY STACK
save_proc_entry_ret:
    sti
    iretd
```

注意进程切换过程中一定要小心翼翼，如何多余的操作都会导致全局崩溃。故一进入中断处理程序，先关中断，然后立即跳到save_proc_entry保存现场。之所以不用call也是防止堆栈被破坏。

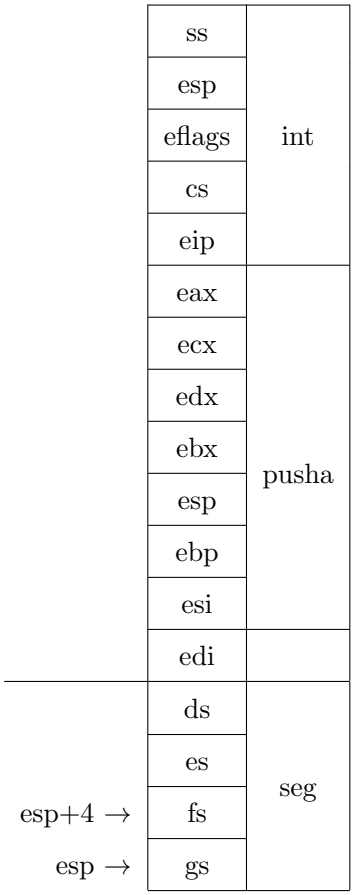
由前面硬件中断和返回一节已知，从用户程序（用户态）转到中断处理程序（内核态）发生了堆栈切换，此时栈顶应该还包含原用户程序栈的信息，这点可以加以利用。故保护现场的过程如下

```
save_proc_entry:
    pusha ; ax,cx,dx,bx,sp,bp,si,di
    push ds
    push es
    push fs
    push gs

    call pit_handler_main
```

```
    pop gs
    pop fs
    pop es
    pop ds
    popa
    jmp save_proc_entry_ret
```

经过一系列push操作后，栈的内容如下



由call pit_handler_main跳入C函数进行状态保存。

```
void pit_handler_main(
    uint32_t gs,uint32_t fs,uint32_t es,uint32_t ds,
    uint32_t di,uint32_t si,uint32_t bp,uint32_t sp,
    uint32_t bx,uint32_t dx,uint32_t cx,uint32_t ax,
    uint32_t ip,uint32_t cs,uintptr_t flags,
    uint32_t user_esp, uint32_t ss)
{
    // increment tick count
    pit_ticks++;

    if (curr_proc == NULL){
        if (curr_pid != 0)
```

```
            schedule_proc(); // no need to save
        interruptdone(0);
        return;
    }

    if (curr_proc->tick > 0){
        curr_proc->tick--;
        interruptdone(0);
        return;
    }

    curr_proc->regImg.eip = ip;
    curr_proc->regImg.cs = cs;
    curr_proc->regImg.eflags = flags;

    curr_proc->regImg.eax = ax;
    curr_proc->regImg.ecx = cx;
    curr_proc->regImg.edx = dx;
    curr_proc->regImg.ebx = bx;

    curr_proc->regImg.esp = sp;
    curr_proc->regImg.ebp = bp;
    curr_proc->regImg.esi = si;
    curr_proc->regImg.edi = di;

    curr_proc->regImg.ds = ds & 0xFFFF;
    curr_proc->regImg.es = es & 0xFFFF;
    curr_proc->regImg.fs = fs & 0xFFFF;
    curr_proc->regImg.gs = gs & 0xFFFF;

    curr_proc->regImg.ss = ss;
    curr_proc->regImg.user_esp = user_esp;

    schedule_proc(); // change another process

    // tell hal we are done
    interruptdone(0);
}
```

注意只有当有进程且要进行进程切换时才进行状态保存，否则都将直接跳过。

这里运用了**不同特权态下中断处理的堆栈机理**来实现进程的保存，代码十分简洁和清晰。

当一个进程的时间片用完时，就会通过schedule_proc函数进行进程调度。

```
void schedule_proc()
{
```

```
    process* pp = proc_pick(); // round-robin
    if (pp == NULL) return;
    proc_switch(pp);
}
```

这里的proc_pick采用最简单的轮盘转(round-robin)方式进行进程的挑选/调度，然后将新的进程送到proc_switch中进行进程切换。

```
void proc_switch(process* pp)
{
    if (!(curr_proc == NULL && curr_pid != 0)){
        curr_proc->status = PROC_SLEEP;
    }
    reset_time(pp);

    pp->status = PROC_ACTIVE;
    curr_proc = pp;

    // set up kernel stack
    int stack = 0;
    __asm__ volatile ("mov eax, esp":"=a"(stack)::);
    tss_set_stack(KERNEL_DS,stack);

    interruptdone(0); // IMPORTANT!!!
    restart_proc(
        pp->regImg.gs, pp->regImg.fs, pp->regImg.es, pp->regImg.ds,
        pp->regImg.edi, pp->regImg.esi, pp->regImg.ebp, pp->regImg.esp,
        pp->regImg.ebx, pp->regImg.edx, pp->regImg.ecx, pp->regImg.eax,
        pp->regImg.eip,
        pp->regImg.cs,
        pp->regImg.eflags,
        pp->regImg.user_esp,
        pp->regImg.ss
        );
}
```

首先重置新进程的时间片，将其设为活跃进程。

然后设置好TSS段的内核栈—这是保护模式的方便之处，在进程切换过程中可以使用内核栈进行辅助操作，避免复杂的数据移动、指针操作。

在开始恢复现场之前一定要注意告知PIT中断处理已经结束，否则在切入用户态后即使设置了eflags，也无法使用中断（因为当前中断未结束）。

最后将保存的PCB信息传入restart_proc，进行现场恢复。

```
restart_proc:
```

```
cli
add esp, 4  ; skip return address
pop gs
pop fs
pop es
pop ds
popa
iretd       ; flush cs:eip
```

理解了堆栈的内容和原理后，其实会发现很简单。先将栈顶restart_proc的返回地址跳过，然后剩下的内容就是我们要恢复的状态了。

由于目前中断处理程序在内核态，而cs处在用户态，故iret指令[5]在执行时先会从TSS中读取内核栈地址ss0:esp0，作为切换进程的过渡。

跳转到cs:eip后，恢复eflags，然后会将ss:esp弹出作为用户栈，最终完成进程切换。

整个过程非常简明、流畅，没有多余的东西，既完美地保存了现场，又完美地恢复了现场，成功做到进程切换的无缝连接！

(v) 多用户程序执行

在create_user_proc中，我创建了四个用户进程，每次创建前都先用IDE将对应硬盘上的程序读入内存，再进行进程创建。下面是一个例子。

```
read_sectors(ADDR_USER_START,0,2); // addr, sect, cnt
proc_create(USER_CS,USER_DS,ADDR_USER_START);
```

注意这里创建的进程都是用户态。以cs为例，除了声明用户态代码段选择子0x18外，还需声明请求特权态(requested privilege level, RPL)为3，即ring 3。将这两者相或得到0x1b，即为真正的USER_CS。这会在iret执行时，实施内核态到用户态的迁移。

完整流程如下图所示。

- Shell中解析到exec指令后，调用create_user_proc函数
- 关中断，创建四个用户进程，然后开中断
- 下一个时钟中断来临时，调用schedule_proc进行调度（注意现在还在内核态，第一次执行时不会进行进程保存）
- 通过proc_switch中的restart_proc函数，启动第一个进程，由**内核态切入用户态**，且保持中断开启
- 在第一个进程执行过程中，遇到时钟中断，**由用户态切回内核态**进行中断处理
- 判断时间片用完，则先进行进程状态保存，然后调用schedule_proc切换进程，如此往复

[5]iretd是32位的返回指令

五、 实验结果

初始化界面

多进程执行（2图）

六、 实验总结

耗费了大量精力，完全重构，也没有了之前的bug

- 任务切换堆栈
- 不同任务地址相同都不行
- controller not ready for writing
- 开中断搞了两天 interrupt(0)
- 又忘记改org!!!

截图网页历史记录

Github代码提交

七、 参考资料

1. OS Development Series, http://www.brokenthorn.com/Resources/OSDevIndex.html
2. Roll your own toy UNIX-clone OS, http://www.jamesmolloy.co.uk/tutorial_html/
3. The little book about OS development, http://littleosbook.github.io/
4. Writing a Simple Operating System from Scratch, http://www.cs.bham.ac.uk/~exr/lectures/opsys/10_11/lectures/os-dev.pdf
5. Intel® 64 and IA-32 Architectures Software Developer's Manual
6. UCore OS Lab, https://github.com/chyyuu/ucore_os_lab
7. CMU CS 15-410, Operating System Design and Implementation, https://www.cs.cmu.edu/~410/
8. 李忠，王晓波，余洁，《x86汇编语言-从实模式到保护模式》，电子工业出版社，2013

附录 A. 程序清单

由于程序太多，请直接见压缩文件。

附录 B. 附件文件说明

1. 内核核心代码

序号	文件	描述
1	bootloader.asm	主引导程序
2	kernel_entry.asm	内核汇编入口程序
3	kernel.c	内核C入口程序
4	Makefile	自动编译指令文件
5	bootflpy.img	引导程序/内核软盘
6	mydisk.hdd	虚拟硬盘
7	bochsrc.bxrc	Bochs配置文件

2. 内核头文件

序号	文件	描述
1	disk_load.inc	BIOS读取磁盘
2	show.inc	常用汇编字符显示
3	gdt.inc	汇编全局描述符表
4	gdt.h	C全局描述符表
5	idt.h	中断描述符表
6	hal.h	硬件抽象层
6.1	pic.h	可编程中断控制器
6.2	pit.h	可编程区间计时器
6.3	keyboard.h	键盘处理
6.4	tss.h	任务状态段
6.5	ide.h	硬盘读取
7	io.h	I/O编程
8	exception.h	异常处理
9	task.h	多进程设施
10	user.h	用户程序处理
11	terminal.h	Shell
12	scancode.h	扫描码
13	stdio.h	标准输入输出
14	string.h	字符串处理

3. 用户程序

用户程序都放置在usr文件夹中。

序号	文件	描述
1-4	prgX.asm	飞翔字符用户程序
5	box.asm	画框用户程序