

Reference

一个操作系统的实现

x86汇编: 从实模式到保护模式

Intro

我选择保护模式下完成实验三, 这四周将"一个操作系统的实现"前九章读了很多遍, 并参考他的代码(准确地说是抄了很多). 为了证明我并没有单纯地抄 代码, 我将附上实验笔记文件.

需要攻克的问题如下

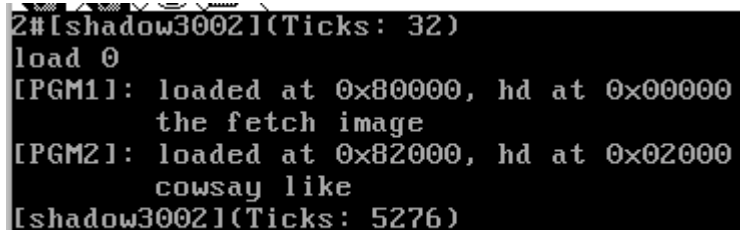
- 进入保护模式
- 将elf的内核读入内存(有坑)
- 准备 GDT, 准备进程表(进程控制块)
- 设置 8259a, 关闭或开启中断, 准备中断handle的表
- 设置时钟中断处理函数, 实现分时多进程
- 实现tty, 以便处理输出和键盘输入
- 实现一个 RingBuffer, 处理键盘输入 backspace
- 实现系统调用
- 实现微内核架构的基础--进程通信
- 基于进程通信实现硬盘驱动

实验需求完成情况

老师需求

- 硬盘上建立用户程序的信息表 => 未实现

- 控制台查看用户程序的信息 => 实现



```
2#[shadow3002](Ticks: 32)
load 0
[PGM1]: loaded at 0x80000, hd at 0x00000
the fetch image
[PGM2]: loaded at 0x82000, hd at 0x02000
cowsay like
[shadow3002](Ticks: 5276)
```

- 设计命令 => 实现

```

2#[shadow30021(Ticks: 32)
load 1
[PGM]: 0

      _\
     .0+\
    '000/
   ' +0000:
  ' +000000:
 - +000000+:
  \/:-!++0000+:
   \ /++++/+++++++:
    \ /+++++++/+++++++:
     \ /+++0000000000000000/
      \ /000SSSS0++0SSSSSS0+
       \ .00SSSSSS0-'\'\ /0SSSSSS+
        -0SSSSSS0.      :SSSSSSSS0.
         :0SSSSSSS/      0SSSS0+++
          \0SSSSSSSS/      +SSSS000/-
           \0SSSSSS0+/:--  -:/+0SSSS0+-
            +SS0+:-'\      \.-/+0S0:
             ++:.\      \-/+/\
              \      \-/+/\
               \      \-/+/\
                \      \-/+/\
                 \      \-/+/\
                  \      \-/+/\
                   \      \-/+/\
                    \      \-/+/\
                     \      \-/+/\
                      \      \-/+/\
                       \      \-/+/\
                        \      \-/+/\
                         \      \-/+/\
                          \      \-/+/\
                           \      \-/+/\
                            \      \-/+/\
                             \      \-/+/\
                              \      \-/+/\
                               \      \-/+/\
                                \      \-/+/\
                                 \      \-/+/\
                                  \      \-/+/\
                                   \      \-/+/\
                                    \      \-/+/\
                                     \      \-/+/\
                                      \      \-/+/\
                                       \      \-/+/\
                                        \      \-/+/\
                                         \      \-/+/\
                                          \      \-/+/\
                                           \      \-/+/\
                                            \      \-/+/\
                                             \      \-/+/\
                                              \      \-/+/\
                                               \      \-/+/\
                                                \      \-/+/\
                                                 \      \-/+/\
                                                  \      \-/+/\
                                                   \      \-/+/\
                                                    \      \-/+/\
                                                     \      \-/+/\
                                                      \      \-/+/\
                                                       \      \-/+/\
                                                        \      \-/+/\
                                                         \      \-/+/\
                                                          \      \-/+/\
                                                           \      \-/+/\
                                                            \      \-/+/\
                                                             \      \-/+/\
                                                              \      \-/+/\
                                                               \      \-/+/\
                                                                \      \-/+/\
                                                                 \      \-/+/\
                                                                  \      \-/+/\
                                                                   \      \-/+/\
                                                                    \      \-/+/\
                                                                     \      \-/+/\
                                                                      \      \-/+/\
                                                                       \      \-/+/\
                                                                        \      \-/+/\
                                                                         \      \-/+/\
                                                                          \      \-/+/\
                                                                           \      \-/+/\
                                                                            \      \-/+/\
                                                                             \      \-/+/\
                                                                              \      \-/+/\
                                                                               \      \-/+/\
                                                                                \      \-/+/\
                                                                                 \      \-/+/\
                                                                                  \      \-/+/\
                                                                                                                                
[shadow30021(Ticks: 12961)
load 2
[PGM]: 1

      ^  ^
      -- --
    (oo)-----
      (__)      )/
          ||----w ||
          ||      ||

```

因为后续肯定会开发文件系统, 所以我觉得建立表没必要

我实现了两个存放在硬盘上的打印字符串程序, 通过一个命令 `load`, 以参数选择用户程序, 并开启一个单独的进程运行, 运行后进程被锁死(调度器不调度), 下次加载时重置控制块保存的 `eip`, `esp`, 再次运行.

额外完成

我还实现了一个和操作系统一起编译的用户态(ring3)的进程 `shell`, 并且有一系列实用工具, 比如 `proc` 打印进程表, `fetch` 查看系统状态, `bsd bse` 进行base64加解码等等, 但代码并没有在硬盘上.

打印进程表, 查看进程运行状态

```

[shadow30021(Ticks: 5276)
proc
sys_task pid: 0  state: 1
tty pid: 1  state: 0
hard disk driver pid: 2  state: 2
file system pid: 3  state: 0
TestA pid: 4  state: 0
TestB pid: 5  state: 1
shell pid: 6  state: 0
user pgm pid: 7  state: 3
[shadow30021(Ticks: 39223)
_

```

base64编码

```
[shadow3002](Ticks: 39223)
bse 123
[base64-encode]encoding (123)
MTIz
```

fetch查看ticks和进程数

```
[shadow3002](Ticks: 98800)
fetch

      -`
     .0+`
    `000/
   `+0000:
  `+000000:
 -+000000+:
  `/:-:++0000+:
   `/+++/+++++++:
    `/+++++++:
     `/++0000000000000000/`
      ./000SSSSSO++0SSSSSSSO+`
       .00SSSSSSO-`"/0SSSSSS+`
        -0SSSSSSSO.      :SSSSSSSO.
         :0SSSSSSSS/      0SSSSSO+++.
          /0SSSSSSSS/      +SSSS000/-
           \0SSSSSSSO+/:--  -:/+0SSSSSO+-
            +SSO+:--`      `.-/+0SO:
             ++:.      `.-/+`
              .      `.-/+`

shadow3002@bochs
-----
OS: S30S
Host: Bochs x86 Emulator 2.6.9
Kernel: 3.0.0
Ticks: 130166
Processes: 8
Shell: void
DE: void
WM: void
WM Theme: void
Theme: void
Icons: void
CPU: I don't know
GPU: void
GPU: void
Memory: 32MB
```

简易计算器

```
2#[shadow3002](Ticks: 32)
cc 123+4123
4246
[shadow3002](Ticks: 10156)
cc 61234/134
456
[shadow3002](Ticks: 18444)
cc 87*99
8613
```

部分笔记及代码

用户程序Makefile

```
#####
PGM_ENTRYPOINT = 0x82000

UPG_ASMKFLAGS = -f elf32
UPG_CFLAGS = -m32 -I user_pgm/ -Og -fno-pic -c -fno-builtin -fno-stack-protector
UPG_LDFLAGS = -m elf_i386 -Ttext $(PGM_ENTRYPOINT) --oformat binary

# This Program
PGM = user_pgm.bin
UPG_OBJS = user_pgm/main.o user_pgm/cstart.o
```

```

pgm: $(PGM)
    dd if=user_pgm.bin seek=16 of=80m.img bs=512 count=16 conv=notrunc

$(PGM) : $(UPG_OBJS)
    $(LD) $(UPG_LDFLAGS) -o $(PGM) $(UPG_OBJS)

user_pgm/main.o: user_pgm/main.cpp
    $(CC) $(UPG_CFLAGS) -o $@ $<

user_pgm/cstart.o : user_pgm/cstart.asm
    $(ASM) $(ASM_KFLAGS) -o $@ $<

```

进入保护模式

```

; 加载 GDTR
    lgdt    [GdtPtr]

; 关中断
    cli

; 打开地址线A20
    in     al, 92h
    or     al, 00000010b
    out    92h, al

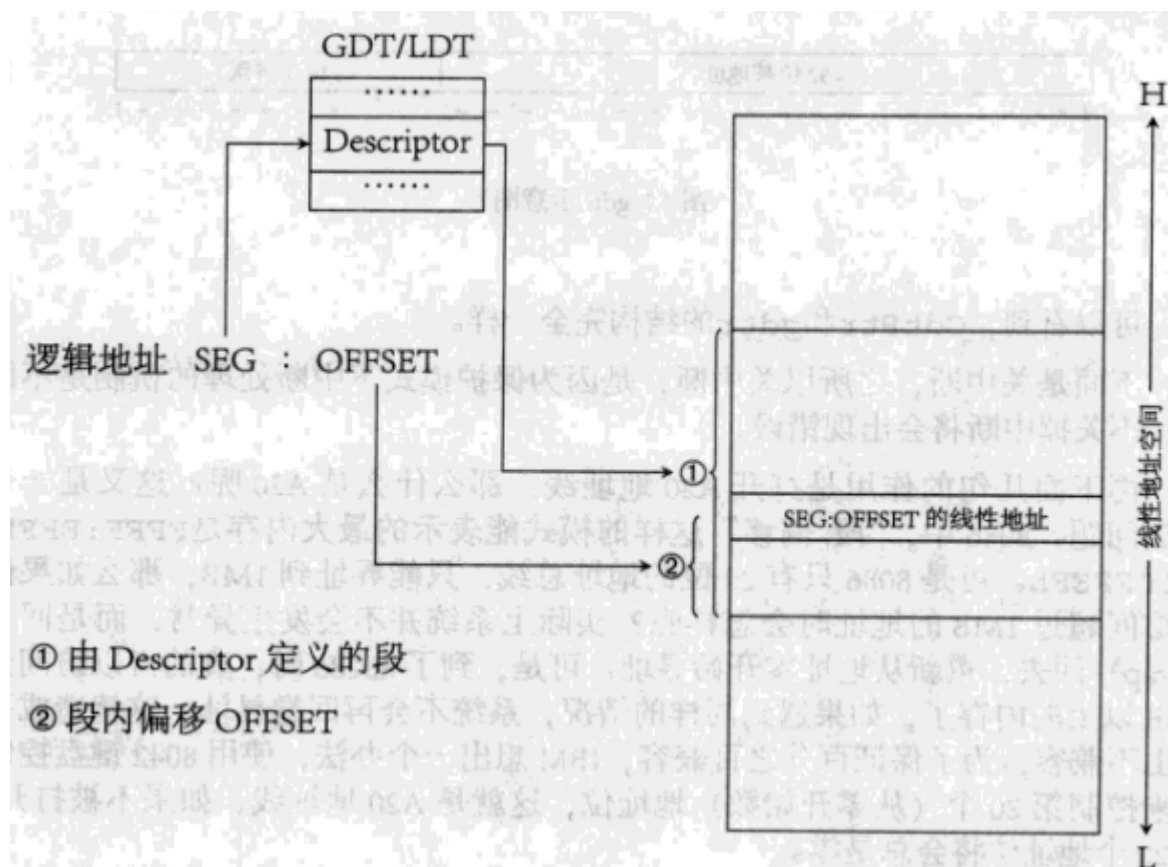
; 准备切换到保护模式
    mov     eax, cr0
    or      eax, 1
    mov     cr0, eax

; 真正进入保护模式
    jmp     dword SelectorFlatC: (BaseOfLoaderPhyAddr+LABEL_PM_START)

```

准备GDT

因为保护模式下对内存的访问是间接地, 保护模式下段寄存器变成了一个索引, 指向一个数据结构的一个表项, 表项中定义了段的起始地址, 界限, 属性等内容.



全局描述符表(Global Descriptor Table,GDT.)

GDT中的第一个描述符必须是空描述符.

GDT以八字节的段描述符(Segment Descriptor)



- 32位的段起始地址
- 20位的段边界
- G位是粒度(Granularity)
- S位用于指定描述符的类型 (Descriptor Type)
- DPL表示描述符的特权级 (Descriptor Privilege Level,DPL)
- P是段存在位(Segment Present)
- X表示是否可以执行行行(executable)
- C位指示段是否为特权级依从的 (Conforming)
- A位是已访问(Accessed)位

初始化GDTR

全局描述符表寄存器(GDTR, 六字节48位, 32位的线性地址和16位的边界。边界数值为全局描述符表的大小减一。

打开A20

这是一个历史遗留问题, 8086最大寻址位数为 20, 如果试图访问超过 1MB 的内存会触发回卷, 重新从地址零开始寻址. 但是到了80286(24 位寻址), 为了保证兼容, intel使用8042键盘控制器控制第 20 个地址位 A20 (从A0开始数), 如果不被打开, 第20个地址位将总为0.

关中断

因为保护模式下中断处理机制与实模式不同(实模式的机制是怎么样的呢?)

设置PE位

将cr0寄存器第0位设置为1, 这将使CPU运行于保护模式.

进入保护模式

为一个jmp命令, 这是为了改变旧的 cs

将elf的内核读入内存(有坑)

`InitKernel`: ; 遍历每一个 Program Header, 根据 Program Header 中的信息来确定把什么放进内存, 放到什么位置, 以及放多少。

```
xor esi, esi
mov cx, word [BaseOfKernelFilePhyAddr + 2Ch];  ecx <- pELFHdr->e_phnum
movzx ecx, cx                                ;
mov esi, [BaseOfKernelFilePhyAddr + 1Ch]      ; esi <- pELFHdr->e_phoff
add esi, BaseOfKernelFilePhyAddr              ; esi <- OffsetOfKernel + pELFHdr->e_phoff

.Begin:
mov eax, [esi + 0]
cmp eax, 0                                    ; PT_NULL
jz .NoAction
push dword [esi + 010h] ; size 1
mov eax, [esi + 04h] ;
add eax, BaseOfKernelFilePhyAddr; | ::memcpy( (void*)(pPHdr->p_vaddr),
push eax ; src | uchCode + pPHdr->p_offset,
push dword [esi + 08h] ; dst | pPHdr->p_filesz;
call MemCpy ;
add esp, 12 ;

.NoAction:
add esi, 020h ; esi += pELFHdr->e_phentsize
dec ecx
cmp ecx, 4
jnz .Begin

ret
```

`cmp ecx, 4`, 就是控制不处理最后四个header

这一部分为16位代码, 我觉得没必要弄特别清楚, 直接抄的源码, 但是遇到了问题, program header中虚拟地址和物理地址为0x08XXXXXX, 这是因为分页机制. 即使在链接时强制指定代码段开始位置 使用 `-Ttext`, 仍然有部分 program header地址错误, 根据观察后我手动控制, 不处理后四个program header(他们内容大小每次编译很一致, 为无用的信息).

ELF文件由4部分组成, 分别是ELF头 (ELF header)、程序头表 (Program header table)、节 (Section) 和节头表 (Section header table) 。

ELF header

```
#define EI_NIDENT 16
typedef struct{
    unsigned char e_ident[EI_NIDENT];
    Elf32_Half e_type;
    Elf32_Half e_machine;
    Elf32_Word e_version;
    Elf32_Addr e_entry;
    Elf32_Off e_phoff;
    Elf32_Off e_shoff;
    Elf32_Word e_flags;
    Elf32_Half e_ehsize;
    Elf32_Half e_phentsize;
    Elf32_Half e_phnum;
    Elf32_Half e_shentsize;
    Elf32_Half e_shnum;
    Elf32_Half e_shstrndx;
}Elf32_Ehdr;
```

最开头是16个字节的e_ident, 其中包含用以表示ELF文件的字符, 以及其他一些与机器无关的信息。开头的4个字节值固定不变, 为0x7f和ELF三个字符。

e_type 它标识的是该文件的类型。

e_machine 表明运行该程序需要的体系结构。

e_version 表示文件的版本。

e_entry 程序的入口地址。

e_phoff 表示Program header table 在文件中的偏移量 (以字节计数) 。

e_shoff 表示Section header table 在文件中的偏移量 (以字节计数) 。

e_flags 对IA32而言, 此项为0。

e_ehsize 表示ELF header大小 (以字节计数) 。

e_phentsize 表示Program header table中每一个条目的大小。

e_phnum 表示Program header table中有多少个条目。

e_shentsize 表示Section header table中的每一个条目的大小。

e_shnum 表示Section header table中有多少个条目。

e_shstrndx 包含节名称的字符串是第几个节 (从零开始计数) 。

Program header

Program header描述的是一个段在文件中的位置、大小以及它被放进内存后所在的位置和大小。

```
typedef struct {
    Elf32_wordp_type;
    Elf32_offp_offset;
    Elf32_Addr p_vaddr;
    Elf32_Addr p_paddr;
    Elf32_wordp_filesz;
    Elf32_word p_memsz;
    Elf32_word p_flags;
    Elf32_word p_align;
}
```

p_type 当前Program header所描述的段的类型。

p_offset 段的第一个字节在文件中的偏移。

p_vaddr 段的一个字节在内存中的**虚拟地址**

p_paddr 在物理内存定位相关的系统中，此项是为**物理地址**保留。

p_filesz 段在文件中的长度。

p_memsz 段在内存中的长度。

p_flags 与段相关的标志。

p_align 根据此项值来确定段在文件及内存中如何对齐。

准备 GDT , 准备进程表(进程控制块)

设置 8259a , 关闭或开启中断, 准备中断handle的表

```
void init_8259A()
{
    out_byte(INT_M_CTL, 0x11); // Master 8259, ICW1.
    out_byte(INT_S_CTL, 0x11); // Slave 8259, ICW1.
    out_byte(INT_M_CTLMASK, INT_VECTOR_IRQ0); // Master 8259, ICW2. 设置 '主8259' 的中断
    入口地址为 0x20.
    out_byte(INT_S_CTLMASK, INT_VECTOR_IRQ8); // Slave 8259, ICW2. 设置 '从8259' 的中断
    入口地址为 0x28
    out_byte(INT_M_CTLMASK, 0x4); // Master 8259, ICW3. IR2 对应 '从8259'.
    out_byte(INT_S_CTLMASK, 0x2); // Slave 8259, ICW3. 对应 '主8259' 的 IR2.
    out_byte(INT_M_CTLMASK, 0x1); // Master 8259, ICW4.
    out_byte(INT_S_CTLMASK, 0x1); // Slave 8259, ICW4.

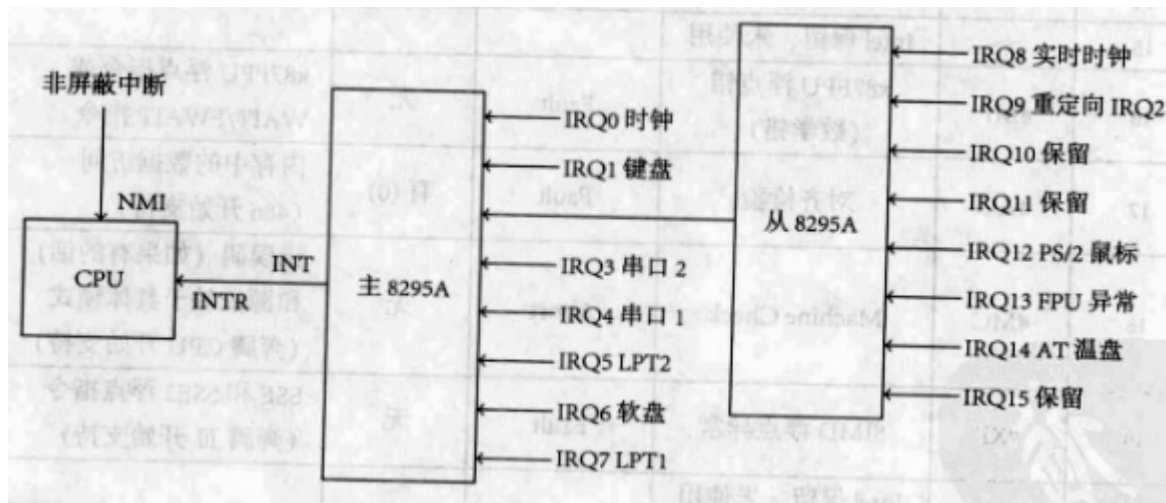
    out_byte(INT_M_CTLMASK, 0xFF); // Master 8259, OCW1.
    out_byte(INT_S_CTLMASK, 0xFF); // Slave 8259, OCW1.

    int i;
    for (i = 0; i < NR_IRQ; i++) {
        irq_table[i] = spurious_irq;
    }
}
```



```
}  
}
```

该芯片处理外部中断



BIOS初始化时, IRQ0-7被设置为08h到0Fh, 我们需要重新设置.

我通过向端口写入特点ICW来设置.

初始化8259a

1. 往端口 20h (主片) 或 A0h (从片) 写入 ICW1。
2. 往端口 21h (主片) 或 A1h (从片) 写入 ICW2。
3. 往端口 21h (主片) 或 A1h (从片) 写入 ICW3。
4. 往端口 21h (主片) 或 A1h (从片) 写入 ICW4。

ICW格式

ICW1 (对应端口 20h 和 A0h)	7	对 PC 系统必须为 0	ICW2 (对应端口 21h 和 A1h)	7	80x86 中断向量
	6			6	
	5			5	
	4	对 ICW1 必须为 1 (端口必须为 20h 或 A0h)		4	
	3			3	000:80x86 系统
	2	1=level triggered 模式, 0=edge triggered 模式		2	
	1	1=4 字节中断向量, 0=8 字节中断向量		1	
	0	1= 单个 8259, 0= 级联 8259		0	
主片 ICW3 (对应端口 21h)	7	1=IR7 级联从片, 0= 无从片	从片 ICW3 (对应端口 A1h)	7	必须为 0
	6	1=IR6 级联从片, 0= 无从片		6	
	5	1=IR5 级联从片, 0= 无从片		5	
	4	1=IR4 级联从片, 0= 无从片		4	
	3	1=IR3 级联从片, 0= 无从片		3	从片连的主片的 IR 号
	2	1=IR2 级联从片, 0= 无从片		2	
	1	1=IR1 级联从片, 0= 无从片		1	
	0	1=IR0 级联从片, 0= 无从片		0	
ICW4 (对应端口 21h 和 A1h)	7	未使用 (设为 0)			
	6				
	5				
	4	1=SFNM 模式, 0=sequential 模式			
	3	主 / 从缓冲模式			
	2				
	1	1= 自动 EOI, 0= 正常 EOI			
	0	1=80x86 模式, 0=MCS 80/85			

通过OCW屏蔽外部中断或发送EOI

OCW1 (对应端口 21h 和 A1h)	7	0=IRQ7 打开, 1= 关闭
	6	0=IRQ6 打开, 1= 关闭
	5	0=IRQ5 打开, 1= 关闭
	4	0=IRQ4 打开, 1= 关闭
	3	0=IRQ3 打开, 1= 关闭
	2	0=IRQ2 打开, 1= 关闭
	1	0=IRQ1 打开, 1= 关闭
	0	0=IRQ0 打开, 1= 关闭

EOI: 结束中断处理, 实模式下每一次中断处理结束都需要发送一个EOI给8259a

设置时钟中断处理函数, 实现分时多进程

开启时钟中断, 设置频率

```
void init_clock()
{
    /* 初始化 8253 PIT */
    out_byte(TIMER_MODE, RATE_GENERATOR);
    out_byte(TIMER0, (u8) (TIMER_FREQ/HZ) );
    out_byte(TIMER0, (u8) ((TIMER_FREQ/HZ) >> 8));

    put_irq_handler(CLOCK_IRQ, clock_handler); /* 设定时钟中断处理程序 */
    enable_irq(CLOCK_IRQ);                    /* 让8259A可以接收时钟中断 */
}
```

时钟中断处理函数

```
void clock_handler(int irq)
{
    ticks++;
    p_proc_ready->ticks--;

    if (k_reenter != 0) {
        return;
    }

    if (p_proc_ready->ticks > 0) {
        return;
    }

    schedule(); // 进程调度
}
```

进程调度

```
void schedule()
{
    Process* p;
    int greatest_ticks = 0;

    while (!greatest_ticks) {
        for (p = proc_table; p < proc_table + proc_num; p++) {
            if (p->state == READY) {
                if (p->ticks > greatest_ticks) {
                    greatest_ticks = p->ticks;
                    p_proc_ready = p;
                }
            }
        }
    }
}
```

```

    if (!greatest_ticks)
        for (p = proc_table; p < proc_table + proc_num; p++)
            if (p->state == READY)
                p->ticks = p->priority;
    }
}

```

中断与特权级转换

通过jmp或call进行的转移

如果目标为非一致代码段, 要求CPL等于目标的DPL, 同时要求RPL小于等于DPL

如果目标为一致代码段, 要求CPL大于或等于目标段的DPL

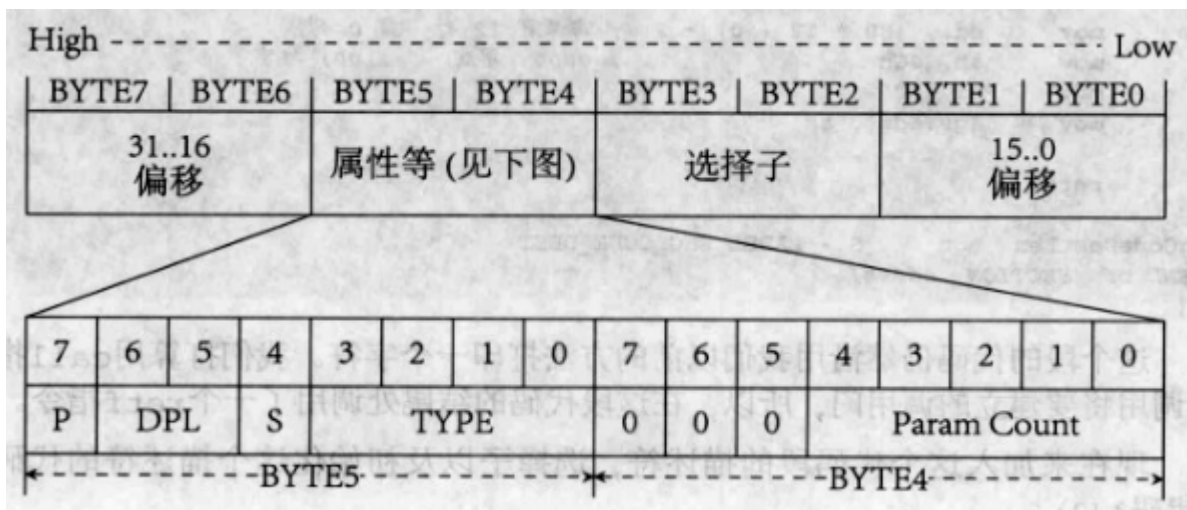
换言之, 非一致代码段, 只能在相同特权级代码转移

一致代码段, 只能从低到高

通过调用门

门描述符

- 调用门
- 中断门
- 陷阱门
- 任务门



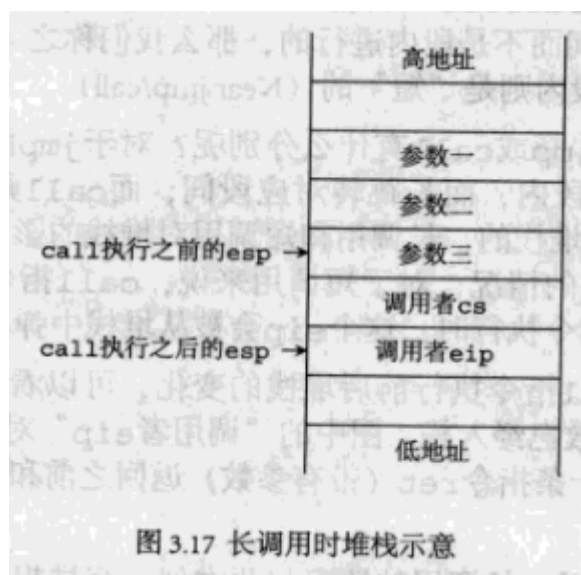
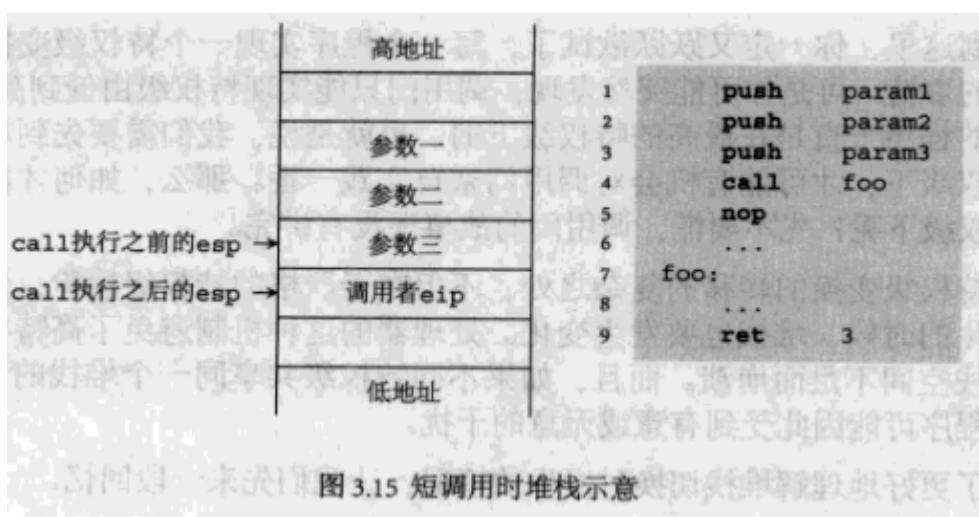
调用门使用的特权检验为

	call	jmp
目标是一致代码段	$CPL \leq DPL_G$, $RPL \leq DPL_G$, $DPL_B \leq CPL$	
目标是非一致代码段	$CPL \leq DPL_G$, $RPL \leq DPL_G$, $DPL_B \leq CPL$	$CPL \leq DPL_G$, $RPL \leq DPL_G$, $DPL_B = CPL$

简单地说, 通过调用门可以实现低特权级向高特权级的转移

call的堆栈变化

call在短调用时只压 eip, 在长调用时多压 cs



但是, 问题在于长调用时堆栈会切换, 比如用户程序调用中断, 用户程序使用自己的栈, 中断处理程序使用系统栈.



TSS

TSS就是解决从ring3->ring0堆栈切换的问题的,

转移流程

1. 根据目标DPL, 从TSS选择ss和esp
2. 从TSS读取新ss, esp
3. 对ss检验
4. 暂时保存当前ss, esp
5. 加载新ss, esp
6. 将旧ss, esp压栈

实现tty, 以便处理输出和键盘输入

这一部分直接借鉴别人的代码, 没有深究

实现三级缓冲和 RingBuffer, 处理键盘输入 backspace

因为如果键盘输入的是回退键, buffer就会变得很复杂, 所以实现三个缓存, 第一个缓存记录键盘的make和break, 第二个缓存记录当前输入缓冲区记录的字符, 如果不为 \b, 则加入ring buffer, 如果为 \b, 则pop ring buffer, 如果为 \n, 就清空缓冲区, 放入第三个缓冲, 等待中断读取字符.

一级缓冲, 位于键盘驱动

```
static KB_INPUT kb_in;

void keyboard_handler(int irq)
{
    u8 scan_code = in_byte(KB_DATA);

    if (kb_in.count < KB_IN_BYTES) {
        *(kb_in.p_head) = scan_code;
    }
}
```

```

        kb_in.p_head++;
        if (kb_in.p_head == kb_in.buf + KB_IN_BYTES) {
            kb_in.p_head = kb_in.buf;
        }
        kb_in.count++;
    }
}

```

二级缓冲, 位于tty, 为RingBuffer, 每次键盘驱动发送信息给tty, tty会执行两种工作

1. 处理显示
2. 将字符放入ringbuffer, 或从ringbuffer中删除字符

```

struct RingBuffer
{
    u8 buffer[RingBuffersize];

    u32 head;
    u32 tail;
};

void in_process(Tty* p_tty, u32 key)
{
    char output[2] = {'\0', '\0'};

    if (!(key & FLAG_EXT)) {
        put_key(p_tty, key);
        push_key(&p_tty->rb, key);
    }
    else {
        ...
        case BACKSPACE:
            put_key(p_tty, '\b');
            back_key(&p_tty->rb);
            break;
        ...
    }
}

```

三级缓冲, 也是一个ringbuffer, 当二级缓冲接收到一个 `\n`, 二级缓冲将全部字符发送到三级缓冲, 等待 `sys_getch()`

```

switch(raw_code) {
    case ENTER:
        put_key(p_tty, '\n');
        push_key(&p_tty->rb, '\n');
        while (ring_length(&p_tty->rb)) push_key(&p_tty->gb, pop_key(&p_tty->rb));
        break;
}

```

实现系统调用

其实就是自定义中断, 并设置几个参数, 我以getch系统调用过程举例

1. getch()调用汇编代码

```
getch:
    mov     eax, _NR_getch
    int     INT_VECTOR_SYS_CALL
    ret
```

2. CPU调用汇编代码 `sys_call` , 注意到不仅推入三个参数, 还推入了当前进程的指针

```
sys_call:
    call    save

    sti
    push    esi

    push    dword [p_proc_ready]
    push    edx
    push    ecx
    push    ebx
    call    [sys_call_table + eax * 4]
    add esp, 4 * 4

    pop esi
    mov     [esi + EAXREG - P_STACKBASE], eax
    cli

    ret
```

3. 调用 `sys_call_table` 数组中的系统调用函数

```
u8 sys_getchar(u32 u1, u32 u2, u32 u3, Process* p_proc)
{
    if (ring_length(&tty_table[p_proc->nr_tty].gb)) {return
    pop_key(&tty_table[p_proc->nr_tty].gb);}
    return '\0';
}
```

实现微内核架构的基础--进程通信

这一部分也是借鉴地比较多, 代码主要实现了 `send_recv` , 实现进程间的通信, 核心为一个 `Message` 的结构体作为通信交换的标准. 比较 `tricky` 的是避免依赖成环, 这部分代码看不太懂

```
int sys_sendrec(int function, int src_dest, Message* m, Process* p)
{
    assert(k_reenter == 0);

    int ret = 0;
    int caller = proc2pid(p);
    Message* m1a = (Message*)va21a(caller, m);
    m1a->source = caller;
```



```

assert(m1a->source != src_dest);

/**
 * Actually we have the third message type: BOTH. However, it is not
 * allowed to be passed to the kernel directly. Kernel doesn't know
 * it at all. It is transformed into a SEND followed by a RECEIVE
 * by `send_recv()'.
 */
if (function == SEND) {
    ret = msg_send(p, src_dest, m);
    if (ret != 0)
        return ret;
}
else if (function == RECEIVE) {
    ret = msg_receive(p, src_dest, m);
    if (ret != 0)
        return ret;
}
else {
    panic("{sys_sendrec} invalid function: "
          "%d (SEND:%d, RECEIVE:%d).", function, SEND, RECEIVE);
}

return 0;
}

```

基于进程通信实现硬盘驱动

参数设置参考orange

注意点有几个, 虚拟地址和线性地址的转换, 线性地址和物理地址的转换, 当然我的分页映射就是 $f(x)=x$, 所以直接使用线性地址就可以了

初始化, 设置处理函数

```

static void init_hd()
{
    /* Get the number of drives from the BIOS data area */
    u8 * pNrDrives = (u8*)(0x475);
    printf("NrDrives:%d.\n", *pNrDrives);
    assert(*pNrDrives);

    put_irq_handler(AT_WINI_IRQ, hd_handler);
    enable_irq(CASCADE_IRQ);
    enable_irq(AT_WINI_IRQ);
}

```

硬盘驱动

```

void task_hd()
{

```

```

Message msg;

init_hd();

while (1) {
    send_recv(RECEIVE, ANY, &msg);

    int src = msg.source;

    switch (msg.type) {
    case DEV_OPEN:
        hd_identify(0);
        break;
    case DEV_READ:
        hd_read(&msg);
        break;
    default:
        assert("HD driver::unknown msg");
        break;
    }

    send_recv(SEND, src, &msg);
}
}

```

硬盘读

```

static void hd_read(Message* p)
{
    hd_cmd cmd;

    u32 sect_nr = (u32)p->POSITION >> 9;

    cmd.features = 0;
    cmd.count = (p->CNT + SECTOR_SIZE - 1) / SECTOR_SIZE;
    cmd.lba_low = sect_nr & 0xFF;
    cmd.lba_mid = (sect_nr >> 8) & 0xFF;
    cmd.lba_high = (sect_nr >> 16) & 0xFF;
    cmd.device = MAKE_DEVICE_REG(1, 0, (sect_nr >> 24) & 0xF);
    cmd.command = ATA_READ;

    hd_cmd_out(&cmd);

    int bytes_left = p->CNT;
    char* la = (char*)va2la(p->PROC_NR, p->BUF);

    while (bytes_left > 0) {
        int bytes = SECTOR_SIZE < bytes_left ? SECTOR_SIZE : bytes_left;

        interrupt_wait();
        port_read(REG_DATA, hdbuf, SECTOR_SIZE);
        memcpy(la, (void*)va2la(2, hdbuf), bytes);
    }
}

```

```

        bytes_left -= SECTOR_SIZE;
        la += SECTOR_SIZE;
    }
    printf("[hd]end\n");
}

```

表 9.1 硬盘 I/O 端口及寄存器

组别	I/O 端口		读时	写时
	Primary	Secondary		
Command Block Registers	1F0h	170h	Data	Data
	1F1h	171h	Error	Features
	1F2h	172h	Sector Count	Sector Count
	1F3h	173h	LBA Low	LBA Low
	1F4h	174h	LBA Mid	LBA Mid
	1F5h	175h	LBA High	LBA High
	1F6h	176h	Device	Device
	1F7h	177h	Status	Command
Control Block Register	3F6h	376h	Alternate Status	Device Control

硬盘读向cmd block reg写入值, 再等待硬盘中断发送数据, 一次读 512 字节.

心得体会

Makefile描述了文件的依赖关系

我的环境为 `archlinux + vs code + clang++ + make`, Make帮我节省了很多时间, 只要规定好文件的依赖关系和生成命令, 就可以一键生成.

切换到保护模式的繁琐

真的十分繁琐, 我想到了一句话, 工程是妥协的产物

宏内核与微内核之争

注意到 `send_recv` 内部, 假如进程a向发送数据给进程b, `send_recv` 需要将a和b的buf地址转换为物理地址, 然后做一次 `memcpy`.

我们看到微内核进程间通信的开销是很大的, 而宏内核直接传buf的指针就行. 当然微内核很美.

这也说明了cpp中 `new` 的开销, 需要不断中断来获取堆的空间, 所以游戏引擎中都使用页机制, 一次申请大量内存, 自己管理.

硬盘中断频繁

注意到驱动程序需要每 512 字节就中断CPU一次, 所以后面产生了新技术, 硬盘将读到的数据传到内存, 等读完才产生一次中断.

其他

心得太多,一时想不起来,就先这样吧