

操作系统实验报告

Lab2

姓名：刘博

学号：141220065

计算机科学与技术系

2016. 4. 4

邮箱：1610266604@qq.com

1. 实验目的:

1. 从实模式进入保护模式
2. 加载内核到内存某地址并跳转运行
3. 初始化中断向量表
4. 初始化 GDT 表
5. 配置 TSS 段
6. 从磁盘加载用户程序到内存相应地址, 并修改用户程序的各个 GDT 表项
7. 进入用户空间前的相关配置
8. 正式进入用户空间
9. 调用库函数 printf (这里我是在显存进行输出)

2. 实验过程:

1. 首先将原 lab1 中的 bootloader 内的内容拷贝到 lab2 的 bootloader 文件夹下。完成对 kernel 文件的加载;
2. 然后对中断向量表进行初始化:

```
set_trap(idt + 0, SEG_KCODE, (uint32_t)vec0, DPL_KERN);
set_trap(idt + 1, SEG_KCODE, (uint32_t)vec1, DPL_KERN);
set_trap(idt + 2, SEG_KCODE, (uint32_t)vec2, DPL_KERN);
set_trap(idt + 3, SEG_KCODE, (uint32_t)vec3, DPL_KERN);
set_trap(idt + 4, SEG_KCODE, (uint32_t)vec4, DPL_KERN);
set_trap(idt + 5, SEG_KCODE, (uint32_t)vec5, DPL_KERN);
set_trap(idt + 6, SEG_KCODE, (uint32_t)vec6, DPL_KERN);
set_trap(idt + 7, SEG_KCODE, (uint32_t)vec7, DPL_KERN);
set_trap(idt + 8, SEG_KCODE, (uint32_t)vec8, DPL_KERN);
set_trap(idt + 9, SEG_KCODE, (uint32_t)vec9, DPL_KERN);
set_trap(idt + 10, SEG_KCODE, (uint32_t)vec10, DPL_KERN);
set_trap(idt + 11, SEG_KCODE, (uint32_t)vec11, DPL_KERN);
set_trap(idt + 12, SEG_KCODE, (uint32_t)vec12, DPL_KERN);
set_trap(idt + 13, SEG_KCODE, (uint32_t)vec13, DPL_KERN);
set_trap(idt + 14, SEG_KCODE, (uint32_t)vec14, DPL_KERN);

set_trap(idt + 0x80, SEG_KCODE, (uint32_t)vecsys, DPL_USER);

set_intr(idt + 32, SEG_KCODE, (uint32_t)irq0, DPL_KERN);
set_intr(idt + 32 + 1, SEG_KCODE, (uint32_t)irq1, DPL_KERN);
set_intr(idt + 32 + 14, SEG_KCODE, (uint32_t)irq14, DPL_KERN);
```

这里对 14 个系统陷阱进行初始化, 并且加入系统调用;

然后再 do_irq.s 进行函数定义:

```
.globl vec0; vec0: pushl $0; jmp asm_do_irq
.globl vec1; vec1: pushl $1; jmp asm_do_irq
.globl vec2; vec2: pushl $2; jmp asm_do_irq
.globl vec3; vec3: pushl $3; jmp asm_do_irq
.globl vec4; vec4: pushl $4; jmp asm_do_irq
.globl vec5; vec5: pushl $5; jmp asm_do_irq
.globl vec6; vec6: pushl $6; jmp asm_do_irq
.globl vec7; vec7: pushl $7; jmp asm_do_irq
.globl vec8; vec8: pushl $8; jmp asm_do_irq
.globl vec9; vec9: pushl $9; jmp asm_do_irq
.globl vec10; vec10: pushl $10; jmp asm_do_irq
.globl vec11; vec11: pushl $11; jmp asm_do_irq
.globl vec12; vec12: pushl $12; jmp asm_do_irq
.globl vec13; vec13: pushl $13; jmp asm_do_irq
.globl vec14; vec14: pushl $14; jmp asm_do_irq

.globl vecsys; vecsys: pushl $0x80; jmp asm_do_irq

.globl irq0; irq0: pushl $1000; jmp asm_do_irq
.globl irq1; irq1: pushl $1001; jmp asm_do_irq
.globl irq14; irq14: pushl $1014; jmp asm_do_irq

.globl irq_empty; irq_empty: pushl $-1; jmp asm_do_irq
```

每个中断处理函数都如上图定义。

初始化中断向量表后，继续对全局描述符表进行初始化：

```
gdt[SEG_KCODE] = SEG(STA_X | STA_R, 0, 0xffffffff, DPL_KERN);
gdt[SEG_KDATA] = SEG(STA_W, 0, 0xffffffff, DPL_KERN);
gdt[SEG_UCODE] = SEG(STA_X | STA_R, USER_CS_BASE, 0xffffffff, DPL_USER);
gdt[SEG_UDATA] = SEG(STA_W, USER_SS_BASE, 0xffffffff, DPL_USER);
gdt[SEG_TSS] = SEG16(STS_T32A, &tss, sizeof(TSS) - 1, DPL_KERN);
gdt[SEG_TSS].s = 0;
set_gdt(gdt, sizeof(gdt));
```

这里分成了两类，一类是 kernel 的全局描述符，一类是 user 的全局描述符，分别用 dpl 区分。

初始化全局描述符表后，要对 tss 段进行初始化，由于 tss 是 kernel 才能访问的段，所以将其特权级设为 dpl_kern；

```
tss.esp0 = STACK;
tss.ss0 = KSEL(SEG_KDATA);
ltr(KSEL(SEG_TSS));
```

这里 stack 是一个宏定义，定义如下：

```
#define SECTSIZE 512
#define ELF_OFFSET_DISK 200 * SECTSIZE
#define STACK 0x800000
#define USER_CS_BASE 0x200000
#define USER_SS_BASE 0x200000
```

由于 i386 进行特权级切换时需要进行栈帧切换，而且 i386 进行栈帧切换时不考虑 esp，只

用 ss 段进行切换，所以这里 esp 可以初始化成 0x800000，ss 段为 kernel 的 ss 段（基址为 0），这样 kernel 内核栈的起始位置就是 ss0: esp0 = 0x800000;

初始化 tss 段寄存器后需要用 ltr 指令将其地址加载到 tr 寄存器中;

初始化段寄存器后，就可以加载用户的程序代码了:

```
void
load_umin(void) {
    /*
     * Load your app here
     * 加载用户程序
     */
    struct ProgramHeader *ph, *eph;
    unsigned char * pa, *i;
    elf = (void *)buf;
    readdisk((void *)elf, 4096, ELF_OFFSET_DISK);

    ph = (struct ProgramHeader *)((char *)elf + elf->phoff);
    eph = ph + elf->phnum;
    for(;ph < eph; ph++)
    {
        if(ph->type == 1)
        {
            pa = (unsigned char *)ph->paddr + USER_CS_BASE;
            readdisk(pa, ph->filesz, ph->off + ELF_OFFSET_DISK);
            for(i = pa + ph->filesz; i < pa + ph->memsz; *i++ = 0);
        }
    }
}
```

其中加载用户程序的代码如上图，这里需要注意两点:

1. 由于用户程序的存放位置是磁盘中的第 202 个扇区，所以加载时需要在磁盘偏移量为 512*200 的位置处开始进行加载，这里将它定义为一个 ELF_OFFSET_DISK 宏;
2. 由于 ph->off 是相对于 elf 在磁盘中的偏移量，所以加载时需要在 ph->off 的基础上加上对应的 elf 的偏移量才能对正确的程序代码进行加载;

由于 cpu 读取代码时需要加上 cs 的基地址才能读取指令，所以要在加载程序代码之前对所有代码加载的基地址进行偏移（加 0x200000）

正确加载用户代码后，就可以进行向用户空间的跳转了:

```

void
enter_user_space(void) {
    /*
     * Before enter user space
     * you should set the right segment registers here
     * and use 'iret' to jump to ring3
     * 进入用户空间
     */
    asm volatile("movw %%ax,%%es":: "a" (USEL(SEG_UDATA)));
    asm volatile("movw %%ax,%%ds":: "a" (USEL(SEG_UDATA)));

    asm volatile("pushl %0          \n\t"
                  "pushl %1          \n\t"
                  "pushl $0x2        \n\t"
                  "pushl %2          \n\t"
                  "pushl %3          \n\t"
                  "iret              \n\t"
                  :: "i" (USEL(SEG_UDATA)),
                     "i" (STACK),
                     "i" (USEL(SEG_UCODE)),
                     "g" (elf->entry));
}

```

这里进行进入用户空间之前的特权级切换，首先需要将段寄存器更新成用户的段寄存器值，然后将用户的所有栈帧进行更新：

Esp = esp0;（便于栈帧切换）

Ss = usr_ss;

Eflags = 0x2;（eflags 的初始值）

Eip = elf->entry;（将用户用户代码的入口存放在 eip 中，这样 iret 后，程序代码就会自动运行）

跳转进入用户空间后，就可以进行 printf 输出了：

```

uentry(void){
    printf("printf test begin...\n");
    printf("the answer should be:\n");
    printf("#####\n");
    printf("Hello, welcome to OSlab! I'm the body of the game. ");
    printf("Bootblock loads me to the memory position of 0x100000, and Makefile also tells me that I'm at
the location of 0x100000. ");
    printf("~!@#(^&*)_+`1234567890-=..... ");
    printf("Now I will test your printf: ");
    printf("1 + 1 = 2, 123 * 456 = 56088\n0, -1, -2147483648, -1412505855, -32768, 102030\n0, ffffffff,
80000000, abcdef01, ffff8000, 18e8e\n");
    printf("#####\n");
    printf("your answer:\n");
    printf("=====\n");
    printf("%s %s%scome %co%s", "Hello,", "", "wel", 't', " ");
    printf("%c%c%c%c%c! ", '0', 'S', 'l', 'a', 'b');
    printf("I'm the %s of %s. %s 0x%x, %s 0x%x. ", "body", "the game", "Bootblock loads me to the memory
position of", "0x100000, and Makefile also tells me that I'm at the location of", "0x100000");
    printf("~!@#(^&*)_+`1234567890-=..... ");
    printf("Now I will test your printf: ");
    printf("%d + %d = %d, %d * %d = %d\n", 1, 1, 1 + 1, 123, 456, 123 * 456);
    printf("%d, %d, %d, %d, %d, %d\n", 0, 0xffffffff, 0x80000000, 0xabcdef01, -32768, 102030);
    printf("%x, %x, %x, %x, %x\n", 0, 0xffffffff, 0x80000000, 0xabcdef01, -32768, 102030);
    printf("=====\n");
    printf("Test end!!! Good luck!!!\n");
    while(1);
}

```

这里使用了网站上的测试用例；

Printf 函数声明如下：

```
void printf(const char *format,...)
```

这里 `format` 是要输出的字符串，我们需要将其中的所有控制符进行区分，然后每遇到一个控制符便将后面的一个对应的参数输出；

提取参数的方法如下：

```
void **args = (void **) &format + 1;
```

这里利用了二级指针，将 `format` 的地址提取出来，则其上一个地址就是需要输出的字符串；即 `args` 便是指向参数列表的二级指针；

参数记为 `args[0],args[1],args[2]...`；

`Printf` 对所有的控制符进行区分，然后对每个控制符进行转化和整理，最后整理成一个可输出的字符串，并将其传给 `syscall` 函数：

```
static inline int32_t syscall(int num, int check, uint32_t a1,uint32_t a2, uint32_t a3, uint32_t a4, uint32_t a5)
{
    /*Generic system call: pass system call number in AX
    up to five parameters in DX,CX,BX,DI,SI
    Interrupt kernel with T_SYSCALL

    The "volatile" tells the assembler not to optimize
    this instruction away just because we don't use the
    return value

    The last clause tells the assembler that this can potentially
    change the condition and arbitrary memory locations.*/

    /*Lab2 code here 嵌入汇编代码，调用int $0x80*/
    int32_t ret = 0;
    asm volatile("int $0x80": "=a"(ret) : "a"(num), "d"(check), "c"(a1), "b"(a2), "D"(a3), "S"(a4));
    return ret;
}
```

`Syscall` 函数将所用到的所有参数分别赋值给相应的寄存器，然后使用 `int` 指令进入中断：

参数说明：`eax`：系统调用号（返回值）

`Edx`：文件描述符

`EcX`：字符串的地址

`Ebx`：字符串输出的长度

`Edi`：输出的位置（行）

`Esi`：输出的位置（列）

调用 `int` 指令后，系统会自动保存当前所有寄存器的值，然后陷入内核中断：

```
void
irq_handle(struct TrapFrame *tf) {
    /*
     * 中断处理程序
     */

    asm volatile("movw %%ax,%%es": "a" (KSEL(SEG_KDATA)));
    asm volatile("movw %%ax,%%ds": "a" (KSEL(SEG_KDATA)));
    switch(tf->irq) {
        case 0x80: do_syscall(tf);break;
        case 3:write(2,"HIT BREAKPOINT EXCEPTION(13)",28,8,0);while(1);break;
        case 13:write(2,"HIT PROTECTION EXCEPTION(13)",28,8,0);while(1);break;
        case 1000:assert(0);break;
        case 1001:assert(0);break;
        case 1014:assert(0);break;
        default:assert(0);
    }
    asm volatile("movw %%ax,%%es": "a" (USEL(SEG_UDATA)));
    asm volatile("movw %%ax,%%ds": "a" (USEL(SEG_UDATA)));
}
```

这里需要注意先将 **ds**, **es** 段寄存器更新（手动），因为 **int** 指令只会切换栈帧和 **cs**，如果不进行手动切换，则系统判断 **eax** 的值会出错（变为 0）；

进入中断后，根据 **eax** 的值判断属于哪种中断，根据 **0x80** 判断为系统调用，于是进入 **do_syscall** 函数中进行系统调用；

```
void
do_syscall(struct TrapFrame *tf) {
    switch(tf->eax)
    {
        case SYS_write: tf->eax = write(tf->edx, (void *)tf->ecx, tf->ebx, tf->edi, tf->esi); break;
        default: assert(0);
    }
}
```

Do_syscall 函数根据 **eax** 寄存器值判断是系统调用中的哪种函数，根据 **eax = 4** 判断属于 **write** 函数的系统调用，于是调用 **write** 函数，并且将文件描述符，字符串起始地址，字符串长度，字符串打印位置传给 **write** 函数；

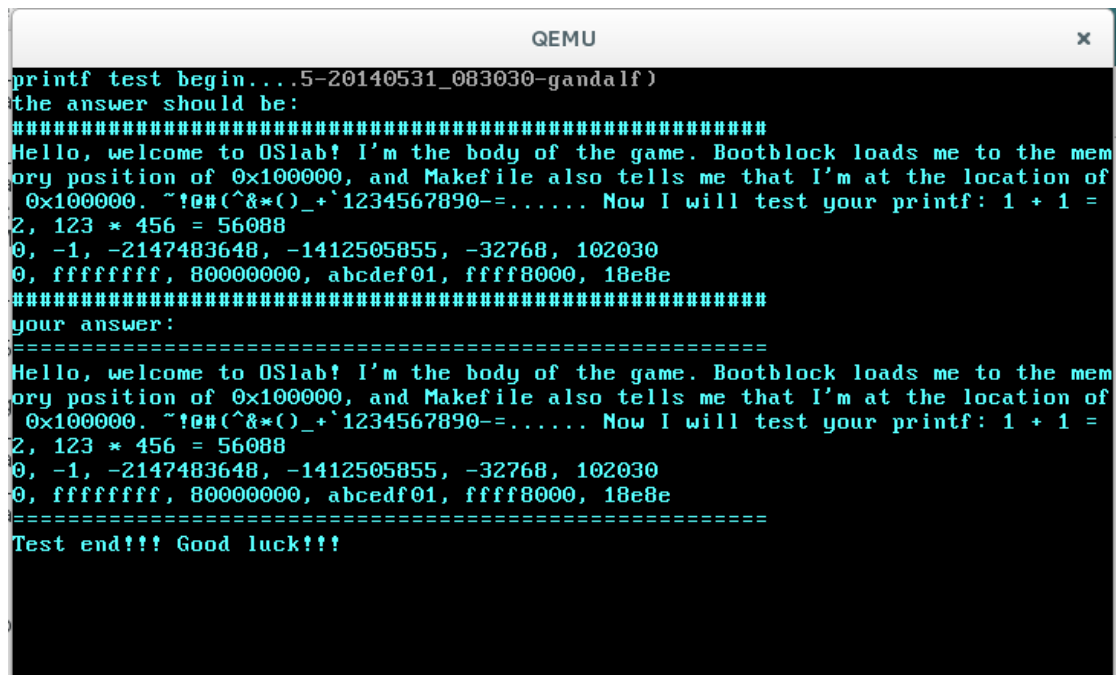
```
int
write(int fd, char *buf, int len, int line, int row) {
    if(fd == 1 || fd == 2)
    {
        uint16_t *gs = (void *)0xb8000;
        gs = gs + line * 80 + row;
        union character c;
        int i;
        for(i = 0; i < len; i++)
        {
            c.ch = (buf + 0x200000)[i];
            c.color = 0x0b;
            gs[i] = c.val;
        }
        return len;
    }
    assert(0);
    return -1;
}
```

Write 函数进行输出，这里可以有两种输出方式，一种是使用定义好的 **putchar** 函数进行串口的输出，另外一种是直接写显存进行屏幕输出，这里选择的是显存的输出（串口输出更为简单）

由于显存的起始地址是 **0xb8000**，所以我们用 **gs** 指向该地址，然后将 **buf** 中的字符值输出到 **gs** 中，操作方法类似于 **boot**；

注意这里 **buf** 的地址并不是实际上字符串的地址，而是需要在其基础地址上加上 **cs** 的偏移量才可以，否则无法读出正确的字符串；

然后就可以进行输出测试了，输出的结果如图：



```

QEMU
printf test begin...5-20140531_083030-gandalf)
the answer should be:
#####
Hello, welcome to OSlab! I'm the body of the game. Bootblock loads me to the mem
ory position of 0x100000, and Makefile also tells me that I'm at the location of
0x100000. ~!@#(^&*()_+'1234567890-=..... Now I will test your printf: 1 + 1 =
2, 123 * 456 = 56088
0, -1, -2147483648, -1412505855, -32768, 102030
0, ffffffff, 80000000, abcdef01, ffff8000, 18e8e
#####
your answer:
=====
Hello, welcome to OSlab! I'm the body of the game. Bootblock loads me to the mem
ory position of 0x100000, and Makefile also tells me that I'm at the location of
0x100000. ~!@#(^&*()_+'1234567890-=..... Now I will test your printf: 1 + 1 =
2, 123 * 456 = 56088
0, -1, -2147483648, -1412505855, -32768, 102030
0, ffffffff, 80000000, abcdef01, ffff8000, 18e8e
=====
Test end!!! Good luck!!!
  
```

至此 lab2 的所有操作完成;

3. 总结与感想:

本次实验让我了解了操作系统内部真正的特权转换和段切换,从而更好的为后面的实验打下基础,不过还是希望助教能够更加清楚地进行讲解,我们也会积极提问;