Lab2 系统调用实验报告

151220098　汤聪

一.　实验目的：

本实验通过实现一个简单的应用程序，并在其中调用一个自定义实现的系统调用，介绍基于中断实现系统调用的全过程。

二.　实验过程：

1. Bootloader 从实模式进入保护模式，加载内核至内存，并跳转执行

(1) 从实模式进入保护模式

在 lab1 中已经进行过该操作，即在 start.s 中实现，将 ebp 初始化为 0，esp 初始化为 0xfff0

```
start:
        cli
        inb $0x92,%al
        orb $0x02,%al
        outb %al,$0x92
        data32 addr32 lgdt gdtDesc
        movl %cr0,%eax
        orb $0x01,%al
        movl %eax,%cr0
        data32 ljmp $0x08,$start32
.code32
start32:
        movw $0x10,%ax
        movw %ax,%ds
        movw %ax,%es
        movw %ax,%ss
        movl $0xfff0,%esp
        movl $0,%ebp
        jmp bootMain
```

(2) 加载内核至内存

先在 boot.c 中定义一个 readseg()函数，参数有代码加载的内存位置，代码长度和相对于头文件的偏移量，定义上界和下界，调用 readsect()函数每次读一个扇区的数据；

```
static void readseg(unsigned char *pa,unsigned int offset,unsigned int len)
{
        unsigned char *epa;
        epa = pa + len;
        pa -= offset % SECTSIZE;
        offset = (offset/SECTSIZE)+1;
        for(;pa<epa;pa+=SECTSIZE,offset++)
                readSect(pa,offset);
}
```

然后在 bootmain 函数中定义一个 elf 文件头指针，先将内核中的数据读入 elf 文件中，然后加载 elf 头表到内存中去，根据 elf->phnum 知道程序段的数量，根据 elf->phoff 知道偏移量，最后根据 elf->entry 跳转到 kernel 中去。

```
void bootMain(void) {
        /* 加载内核至内存，并跳转执行 */
        struct ELFHeader *elf;
        struct ProgramHeader *ph,*eph;
        elf=(struct ELFHeader*)0x8000;
        readseg((unsigned char *)elf,0,4096);
        ph=(struct ProgramHeader*)((char *)elf+elf->phoff);
        eph = ph + elf->phnum;
        for(;ph<eph;ph++)
                readseg((unsigned char *)(ph->paddr),ph->off,ph->filesz);
        ((void (*)(void))(elf->entry))();
        while(1);
}
```

2. 内核初始化 IDT，初始化 GDT，初始化 TSS

（1） 初始化 IDT

在 kernel/kernel/idt.c 中已经初始化完毕，无需改动；

（2） 初始化 GDT

在 kernel/kernel/kvm.c 中的 initSeg()中已经初始化完成；

（3） 初始化 TSS

在 kernel/kernel/kvm.c 中的 initSeg()初始化 tss.esp0 和 tss.ss0

```
ess */
  asm volatile("movl %%esp,%0":"=m"(tss.esp0));
  tss.ss0=KSEL(SEG_KDATA);
  asm volatile("ltr %%ax":: "a" (KSEL(SEG_TSS)));
```

(4) 设置正确的段寄存器

在 kernel/kernel/kvm.c 中的 initSeg()中初始化三个段寄存器

```
/*设置正确的段寄存器*/
asm volatile("movw %%ax,%%es"::"a"(KSEL(SEG_KDATA)));
asm volatile("movw %%ax,%%ds"::"a"(KSEL(SEG_KDATA)));
asm volatile("movw %%ax,%%ss"::"a"(KSEL(SEG_KDATA)));
```

3. 内核加载用户程序至内存，对内核堆栈进行设置，通过 iret 切换至用户空间，执行用户程序

(1)加载用户程序至内存

和第一部分加载内核至内存相似，先将用户程序加载到 elf 中，再将 elf 读入内存中；但是扇区的位置为 201，而不是 1

(2)用 iret 切换至用户态

将段寄存器加载到用户段，对内核堆栈进行设置，ss,cs,eip 都设置为 0，esp 设置为 0x300000，es,ds 读入 ax 的值，iret 之后，寄存器的值会更新。

```
asm volatile("movw %%ax,%%es"::"a"(USEL(SEG_UDATA)));
asm volatile("movw %%ax,%%ds"::"a"(USEL(SEG_UDATA)));
asm volatile("pushl %0"::"a"(USEL(SEG_UDATA)));
asm volatile("pushl $0x300000");
asm volatile("pushl $0x202");
asm volatile("pushl %0"::"a"(USEL(SEG_UCODE)));
asm volatile("pushl %0"::"a"(entry));
asm volatile("iret");
```

4. 用户程序调用自定义实现的库函数 printf 打印字符串

（1） 在 lib/syscall.c 中实现 printf 函数

分别要实现打印字符，打印字符串，打印十进制，打印十六进制
系统调用号为 2，
所以打印字符用 syscall(2,0,ch,0,0,0)就可以实现；

```
void printch(char ch)
{
        syscall(2,0,ch,0,0,0);
}
```

打印字符串只要循环输出每个字符

```
void printstr(char* str)
{
        int i=0;
        for(;str[i]!='\0';i++)
        {
                syscall(2,0,str[i],0,0,0);
        }
}
```

打印十进制数就是将一个 int 型按字符形式输出，但是要考虑 0 和负数的情况，先将数每次除 10，每次取余存入数组中，再将数组元素倒序输出

```
if(dec==0)
{
        syscall(2,0,'0',0,0,0);
        return;
}
if(dec<0)
{
        syscall(2,0,'-',0,0,0);
        dec=-dec;
}
int temp=dec;
char t[20];
int count=0;
for(;temp>0;temp=temp/10)
{
        t[count]=(char)(temp%10)+'0';
        count++;
}
int i=0;
for(;i<=count-1;i++)
        syscall(2,0,t[count-1-i],0,0,0);
```

打印十六进制于十进制很类似，还需要考虑余数大于 9，则要先减 10，加 a

```
if(hex==0)
{
        syscall(2,0,'0',0,0,0);
        return;
}
char t[20];
int count=0;
unsigned temp=hex;
for(;temp>0;temp=temp/16)
{
        if(temp%16<10)
                t[count]=(char)(temp%16)+'0';
        else
                t[count]=(char)((temp%16)-10)+'a';
        count++;
}
int i=0;
for(;i<=count-1;i++)
        syscall(2,0,t[count-1-i],0,0,0);
```

然后根据 printf 中参数的地址逐个往后推移，逐个打印。

(2)syscall()函数实现将一些参数保存至寄存器中，然后调用 int 0x80 中断

```
/* 内嵌汇编 保存 num, a1, a2, a3, a4, a5 至通用寄存器*/
asm volatile("int $0x80" : "=a"(ret): "a"(num), "b"(a1), "c"(a2), "d"(a3

return ret;
```

5. Printf 基于中断陷入内核，由内核完成在视频映射的显存地址中写入内容，完成字符串的打印

在 kernel/kernel/ireHandle.c 函数中，当陷入 0x80 中断时，执行 syscallHandle()函数，当系统调用号为 2 时，实现写显存功能，这与 lab1 中打印 hello,world 相类似，但是无法确定下一打印字符的位置，需要添加一个变量 location 表示光标的位置，每次打印 location 往后移两个位置，当遇到换行符时，需要换行，location=location+160-location%160，更新 location 的值，移动光标的位置，实现换行。

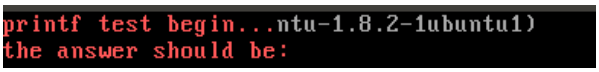
```
if(tf->eax==2)
{
        if(tf->ecx=='\n')
        {
                location=location+(160-location%160);
                return;
        }
        asm volatile("movw %%ax,%%gs"::"a"(KSEL(SEG_VEDIO)));
        asm volatile("movb $0x0c,%ah");
        asm volatile("movb %0,%%al"::"m"(tf->ecx));
        asm volatile("movl %0,%%edi"::"m"(location));
        asm volatile("movw %ax,%gs:(%edi)");
        location+=2;
}
```

三. 执行结果：

```
printf test begin...ntu-1.8.2-1ubuntu1)
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, -./,),(-*,(, -1412505855, -32768, 102030
0, ffffffff, 80000000, abcedf01, ffff8000, 18e8e
-----------------------------------------------------------
Test end!!! Good luck!!!
```

四. 感想与反思：

1. 加载内核到至内存那部分和 pa 中很类似
2. 写显存覆盖了 qemu 中原本的内容，



3. 理解整个过程不是很清楚。