

# 操作系统 lab2 系统调用

邱梓豪

141130077

1.首先在 bootloader/start.S 中实现从实模式切换到保护模式。

```
.global start
start:
    cli                                #关闭中断

    inb $0x92, %al                    #启动A20总线
    orb $0x02, %al
    outb %al, $0x92

    lgdt gdtDesc                      #加载GDTR

    movl %cr0, %eax                   #启动保护模式
    orl $0x1, %eax
    movl %eax, %cr0

    ljmp $0x08, $start32              #长跳转切换至保护模式
```

```
.code32
start32:

    movw $0x18, %ax
    movw %ax, %gs

    movw $0x10, %ax
    movw %ax, %ds
    movw %ax, %ss
    movw %ax, %es

    movl $0x8000, %esp

    jmp bootMain
```

接着再对 GDT 中的相关段进行设置。这里设置代码段，数据段和  
视频段。

2.之后在 bootloader/boot.c 中将内核代码的 ELF 文件的加载入内存中，然后跳转执行。

```
void bootMain(void) {
    /* 加载内核至内存，并跳转执行 */
    struct ELFHeader* elf;           // start of elf file
    struct ProgramHeader* ph;
    struct ProgramHeader* ph_end;    // start of ProgramHeader
    unsigned char* pa, *pa_end;      // physical addr of a segment

    elf=(struct ELFHeader*)0x8000;    // load elf file to 0x8000 in memory
    readSect((unsigned char*)elf, 1); // load sector 1 on disk to memory

    /* load each program segment */
    ph = (struct ProgramHeader*)((char*)elf + elf->phoff);
    ph_end = ph + elf->phnum;
    while (ph<ph_end)
    {
        pa = (unsigned char*)ph->paddr; // load to this physical addr
        pa_end = pa + ph->filesz;      // end of the physical addr
        int i;
        for (i=ph->off/SECTSIZE+1; pa<pa_end; pa+=SECTSIZE, i++)
            readSect(pa, i);
        readSect(pa, i);
        ph++;
    }

    // execute the kernel
    ((void(*)(void))elf->entry)();
}
```

由于 bootloader 的大小最多为 510 字节，所以这里必须反复调整代码以满足要求。

3.之后在会开始执行内核代码，打开 kernel 下的 main.c，可以看到目前在 kernel 内核中的执行流程如下：

```
void kEntry(void) {
    initSerial(); // initialize serial port
    initIdt();    // initialize idt
    initIntr();   // initialize 8259a
    initSeg();    // initialize gdt, tss
    loadUMain();  // load user program, enter user space

    while(1);
    assert(0);
}
```

初始化串口=》初始化 idt=》初始化 8259a=》初始化段寄存器=》载入用户程序。

在内核部分首先补全的代码在在 kvm.c 中，与初始化段有关。首先是初始化 TSS:

这里 TSS 的 esp0 我初始化成 0xffffffff，这是前面初始化 GDT 各段时用的界限，这里保持一致；ss0 我初始化成内核代码段的段选择子。

```
/*
 * 初始化TSS
 */
tss.esp0=0xffffffff;
tss.ss0=KSEL(SEG_KDATA);
asm ("ltr %%ax:: "a" (KSEL(SEG_TSS)));
```

之后设置正确的段寄存器，我将内核数据段段选择子依次赋给%es,%ss 和%ds:

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

4.然后加载用户程序，由于在生产 kernel 的时候其代码被延伸到了 200 号扇区，所以用户程序代码从第 201 号扇区开始。

```
print STDERR "OK: Kernel is $n bytes - Extended to 200 sectors\n";
```

首先在内存中申请一块 4096 字节的空间用于存放用户程序，之后从 201 号扇区开始连续读取 8 个扇区（一共 4096 字节）进入 buf。用 elf 指向这块空间以便之后进行解析。

```

/*加载用户程序至内存*/
uint8_t buf[4096];
struct ELFHeader* elf;
struct ProgramHeader* ph;
unsigned char* pa, *pa_end;

elf=(struct ELFHeader*)buf;
pa=(unsigned char*)elf;
pa_end=pa+4096;

for (int i=0;i<8;i++){ // read 4096 bytes from disk into buf
    readSect(pa, i+201);
    pa+=SECTSIZE;
}

```

解析 elf 文件的过程就是遍历 elf 文件中的每个段，每段的地址可以用 elf 的地址加上段偏移量 elf->phoff 再加上 i\*每段长度得到。先判断每段的类型是否为可载入（PT\_LOAD），如果是，则将该段载入到段中 paddr 指向的内存区域，filesz 和 memsz 之间的区域用 0 填充。之后调用 enterUserSpace 函数，传入用户代码首地址，转入用户空间。

```

for (int i=0;i<elf->phnum;i++)
{
    ph=(struct ProgramHeader*)((char*)elf+elf->phoff+i*(elf->phentsize));
    if (ph->type == 1)
    {
        pa=(unsigned char*)ph->paddr;
        pa_end=pa+ph->filesz;
        int i=ph->off/SECTSIZE+201;
        for (; pa<pa_end; pa+=SECTSIZE, i++)
            readSect(pa, i);
        while(pa_end<pa+ph->memsz)
            *pa_end++=0;
    }
}
enterUserSpace(elf->entry);

```

## 5.转入用户空间

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

asm volatile("pushfl"); // eflags
asm volatile("pushl %%eax"::"a"(KSEL(SEG_KCODE))); // cs
asm volatile("pushl %%eax"::"a"(entry)); // eip

asm volatile("iret");
```

在转入用户空间前，首先要将用户数据段的段选择子赋给%ds和%es，之后依次将 eflags，cs 和 eip 压入栈中，因为 iret 指令会从栈中弹出这三个值并赋给相应的寄存器，为使转入用户空间后直接执行用户程序，这里 eip 就应该是用户程序的入口地址。

6.处理系统调用。现在 lib/syscall.c 中实现 printf 函数的功能，即实现格式化输出。

```
int32_t syscall(int num, uint32_t a1, uint32_t a2,
                uint32_t a3, uint32_t a4, uint32_t a5)
{
    int32_t ret = 0;

    /* 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), "S"(a4), "D"(a5)); // S:%esi D:%edi

    return ret;
}
```

syscall 函数相当于对 int \$0x80 这条指令进行封装，为上层函数提供系统调用的接口，这里将 num，a1，a2，a3，a4，a5 这 6 个可能用到的参数保存入通用寄存器内，然后将返回值放在 ret 中返回给上

层函数。

printf 函数利用一次遍历格式化字符串以实现格式化输出。

```
void printf(const char *format,...){
    char* p;           // point to data
    int num;            // store signed number
    unsigned int u_num; // store unsigned number
    char ch;            // store character
    char* str;          // store string
    int i;
```

首先利用几个参量来存储当前读到的数、字符或字符串

```
if (format == 0)
    return;
else
{
    p = (char*)&format;
    p += 4;
}
```

由于第一个参数格式化字符串在栈中保存的仅是地址，占 4 字节，所以 p 加 4 便指向格式化字符串之后的数据区。

```
while(*format)
{
    if (*format == '%')
    {
        switch(++format) // choose what kind of format to output
        {
            case 'd': num = *(int*)p;
                      if (num < 0){
                          syscall(4, 2, '-', 0, 0, 0);
                          num = -num;
                      }
                      i=0;
                      int dtmp[20];
                      do{
                          dtmp[i++]=num%10;
                          num /= 10;
                      }while(num!=0);
                      for (; i>0; i--)
                          syscall(4, 2, ((dtmp[i-1]>0)?dtmp[i-1]:-dtmp[i-1])+'0', 0, 0, 0);
                      break;
        }
    }
}
```

下面是 printf 的具体格式化流程，测试用例只有%d，%c，%x，%s这四种格式化控制，所以我在这里只对这四种情况进行处理。

对于%d的情况，首先利用指针类型转换将该四字节内容解析为一

个 int 型的数，如果这个数小于 0，则输出负号，并将其变成其相反数；正数不变。之后用一个循环将该数的每一位放进数组 dtmp 中，最后逆序将数组 dtmp 中的各个数输出即可。

这里要注意一种特殊情况，那就是-2147483648 这个数，这个数的相反数还是它自己，所以最后在调用 syscall 时要注意判断数组内数的正负，是负数的话还要将其转为正数。

格式化%x 的方法相似，只不过不用处理负数，在将数转化成字符这一步中利用 switch 语句，小于 10 的数加上 '0' 输出，大于等于 10 的数输出相应字母即可。

```
case 'x':    u_num = *(unsigned int*)p;
            i=0;
            int xtmp[20];
            do{
                xtmp[i++]=u_num%16;
                u_num /= 16;
            }while(u_num!=0);
            while (i>0){
                switch(xtmp[i-1]){
                    case 10: syscall(4, 2, 'a', 0, 0, 0); break;
                    case 11: syscall(4, 2, 'b', 0, 0, 0); break;
                    case 12: syscall(4, 2, 'c', 0, 0, 0); break;
                    case 13: syscall(4, 2, 'd', 0, 0, 0); break;
                    case 14: syscall(4, 2, 'e', 0, 0, 0); break;
                    case 15: syscall(4, 2, 'f', 0, 0, 0); break;
                    default: syscall(4, 2, xtmp[i-1]+'0', 0, 0, 0); break;
                }
                i--;
            }
            break;
```

```
break;
case 's':    str = (char*)(*((char**)p));
            while(*str != '\0'){
                syscall(4, 2, *str++, 0, 0, 0);
            }
            break;
case 'c':    ch = *(char*)p;
            syscall(4, 2, ch, 0, 0, 0);
            break;
```

格式化%s 和%c 都是直接将内存所指向的内存单元的内容取出输

出即可。

```
    }  
    p+=4;  
}  
else        // output the string directly  
    syscall(4, 2, *format, 0, 0, 0);  
    format++;
```

每输出一个格式化数后 `p` 要加上 4 以指向下个待输出的数。如果 `*format!='%'` 时，则直接输出该字符。

最后 `format++`，指向格式化字符串的下一个字符，开始新一轮循环。

## 7.int 之后的过程。

在 `printf` 调用 `int 0x80` 陷入内核之后，`int` 会去根据中断号 `0x80` 查 IDT 表，在 `idt.c` 中可以看到该中断的处理方式：

```
setIntr(idt + 0x80, SEG_KCODE, (uint32_t)irqSyscall, DPL_USER);
```



```

.global irqSyscall
irqSyscall:
    pushl $0 // push dummy error code
    pushl $0x80 // push interrupt vector into kernel stack
    jmp asmDoIrq

.global asmDoIrq
asmDoIrq:
    pushal // push process state into kernel stack

    pushl %esp
    call irqHandle
    addl $4, %esp

    popal
    addl $4, %esp //interrupt vector is on top of kernel stack
    addl $4, %esp //error code is on top of kernel stack
    iret

```

之后进入 irqHandle，执行相关处理程序。

```

Void irqHandle(struct TrapFrame *tf) {
    /*
     * 中断处理程序
     */
    /* Reassign segment register */
    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)));

    switch(tf->irq) {
        case -1:
            break;
        case 0xd:
            GProtectFaultHandle(tf);
            break;
        case 0x80:
            syscallHandle(tf);
            break;
        default:assert(0);
    }
}

```

在中断处理程序中，首先将各数据段切换到内核数据段，然后根据陷阱帧中的 irq 来确定处理程序。

```

void syscallHandle(struct TrapFrame *tf) {
    /* 实现系统调用*/
    switch (tf->eax){
        case 4: putchar((char)tf->ecx);           // write
                tf->eax=1;
                break;
        default: assert(0); break;
    }
}

```

在处理程序中，我设 write 这一系统调用的编号为 4，利用 putchar 实现的接口将字符（tf->ecx）写入显存中，将返回值（tf->eax）设为 1，表明操作成功。

## 8.写显存。

写显存之前应该在 GDT 中为显存注册，在/include/x86/memory.h 中我定义视频段的下标为 6.

```

#define SEG_VIDEO    6           // video memory

```

然后在 initSeg()这个函数中设置该段的 base 为 0xb8000，这样就可以通过将视频段的偏移量赋给%gs 来实现写显存。

```

gdt[SEG_VIDEO] = SEG(STA_W,           0xb8000, 0xffffffff, DPL_USER);

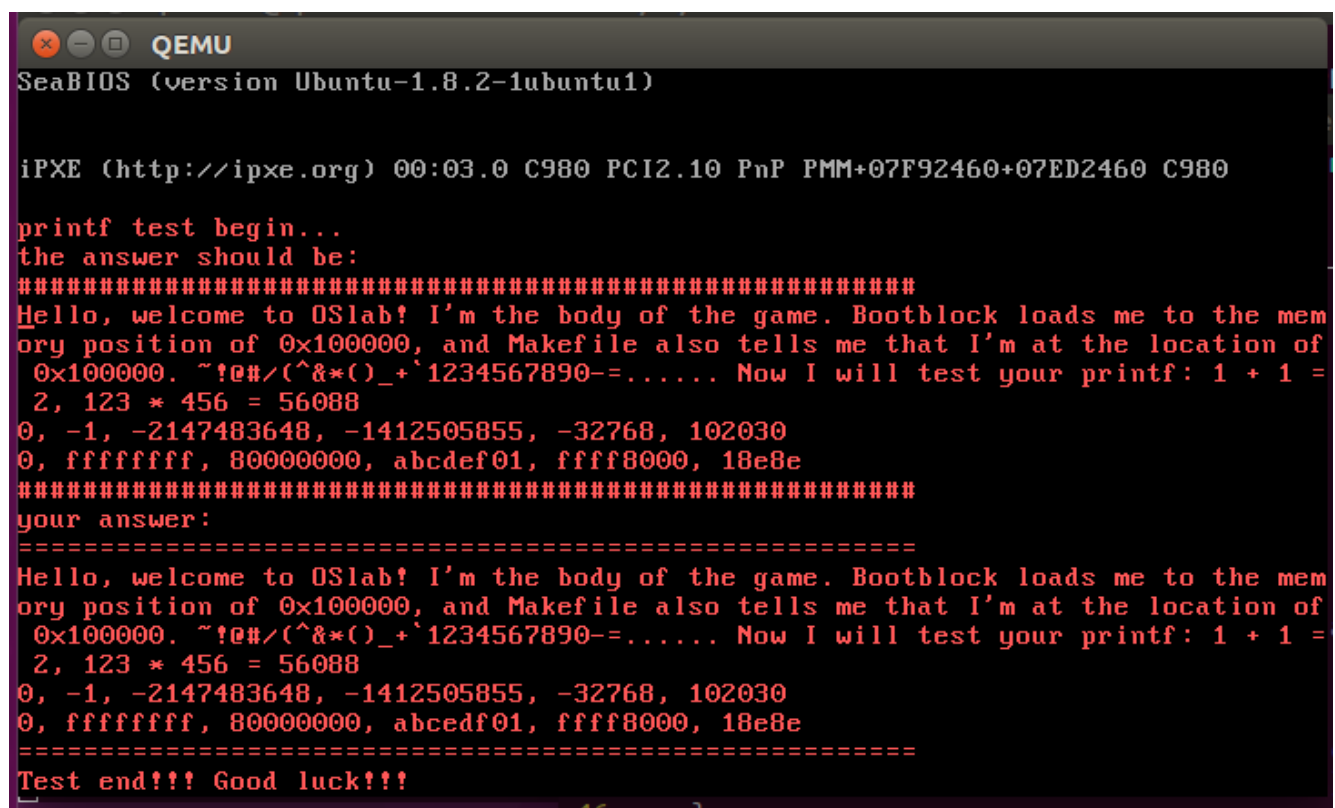
```

具体写显存的代码如下我用 line 记录当前写的行，col 记录当前写的列，先计算出要写入的位置赋给%edi，再将要写入的字符的 ascii 码赋给%al，便可以完成写入。

```
// The following code writes char to video memory
if (col==79){
    line++;
    col=-1;
}
int edi=0;
if (ch == '\n'){
    line++;
    col=-1;
    return;
}
else{
    col++;
    edi=(80*line+col)*2;

    asm volatile("movl %%eax, %%edi"::"a"(edi));
    asm volatile("movw $0x30, %ax");
    asm volatile("movw %ax, %gs");
    asm volatile("movb $0x0c, %ah");
    asm volatile("movb %%bl, %%al"::"b"(ch));
    asm volatile("movw %ax, %gs:(%edi)");
}
}
```

最后的效果如下：



```
QEMU
SeaBIOS (version Ubuntu-1.8.2-1ubuntu1)

iPXE (http://ipxe.org) 00:03.0 C980 PCI2.10 PnP PMM+07F92460+07ED2460 C980

printf test begin...
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!!!
```