操作系统实验报告

Lab1

姓名：刘博

学号：141220065

计算机科学与技术系

2016.3.12

邮箱：1610266604@qq.com

1. 实验目的：

1. 实现一个简单的引导程序，并且通过该引导程序实现简单的输出程序；
2. 了解实模式和保护模式之间的转换，了解 bios 中断并会简单使用；
3. 了解操作系统的加载过程，熟悉 linux 环境。

2. 实验过程：

•第一部分：实现简单的引导程序，并且能够简单的进行输出。（要求用汇编代码实现，不使用 bios 中断）

•目的：了解操作系统引导程序的加载过程，学习写显存输出，了解实模式和保护模式的区别和转化。

•操作方法：

1. 首先要熟悉框架代码，注意到引导程序的代码主要在 start.s 文件中，所以我们先来看 start.s 文件

```
#include "asm.h"
# 从此开始是16位代码
.code16
.globl start
start:
        cli

        xorw %ax, %ax
        movw %ax, %ds
        movw %ax, %es
        movw %ax, %ss

        movw $0x2401, %ax
        int $0x15

        lgdt gdtdesc

        movl %cr0, %eax
        orl $0x1, %eax
        movl %eax, %cr0

        ljmp $GDT_ENTRY(1), $start32
```

内容：

start.s 文件内容如上，可以看到 start 函数中主要是实模式下的代码，对部分段寄存器进行初始化，并且完成实模式到保护模式的跳转，下面我们分块进行解读：

1. cli：这是用来进行关中断的指令，通过改变 IF 的值来进行关中断操作，这在 PA 当中已经很常见。

2. 初始化段寄存器：

```
xorw    %ax, %ax
movw    %ax, %ds
movw    %ax, %es
movw    %ax, %ss
```

这四句话是在对段寄存器初始化（赋值为 0），由

于在实模式中，cs 寄存器是用来和 ip 寄存器合并存放代码指针的，所以对于 cs 寄存器我们不作处理。

3. 打开 A20 地址线：

```
movw    $0x2401, %ax
int     $0x15
```

所谓打开 A20 地址线，实际上就是扩大内存的寻址空间，当 A20 地址线关闭时，内存最大寻址空间为 FFFF：FFFF，对于大于该地址的寻址地址默认环绕回 1M 地址范围，对于 24 根地址线的 80286 和 80386，cpu 通过 A20 地址线开关来控制内存的寻址范围，从而实现实模式到保护模式的跳转。所以这段代码通过 0x15 号 bios 中断来打开 A20 地址线。

4. 加载 GDT 表的基地址：

```
lgdt    gdtdesc
```

lgdt 指令可以将 gdt 表的基地址加载到 cpu 的 gdtr 寄存器中，gdtdesc 则是 gdt 表的基地址。

5. 设置 cr0 的 PE 位，打开保护模式开关：

```
movl    %cr0, %eax
orl     $0x1, %eax
movl    %eax, %cr0
```

这一段指令是用来将 cr0（控制寄存器）的 PE（protect enable）位置为 1。因为在进入保护模式时操作系统会检查 PE 位，如果为 1，则进入保护模式。

6. 长跳转进入保护模式：

```
ljmp    $GDT_ENTRY(1), $start32
```

通过 ljmp 指令,将 cs 寄存器赋值,同时将 start32 代码的首地址赋值给 ip，从而跳转到保护模式代码。

2. 下面便进入保护模式代码：

```
.code32
start32:
        movw $GDT_ENTRY(3), %ax
        movw %ax, %gs

        movw $GDT_ENTRY(2), %ax
        movw %ax, %ds
        movw %ax, %es
        movw %ax, %ss

        movb $0x0c, %ah
        movl $((80 * 5 + 0) * 2), %edi

        movw $Hello, %si
        movl $0x12, %ecx
        call print

.loop32:
        movl $0x8000, %esp

        call bootmain
```

保护模式的主要代码在两个矩形框中：

橙色的矩形框内主要是根据 gdt 表的第 2 项来

对数据段寄存器进行更新。

蓝色矩形框内是设置栈帧地址(0x8000 为默认)然后跳出 start.s 进入 bootmain 函数。

3. elf 文件的解析以及程序代码的加载：

elf 文件存在于磁盘的 1 号扇区，是用来加载可执行文件程序代码，这里我们进入 bootmain 函数来进行 elf 文件的解析和加载。

```
void bootmain(void)
{
        struct ELFHeader *elf;

        /* 这里是加载磁盘程序的代码 */
        struct ProgramHeader *ph, *eph;
        unsigned char * pa, *i;
        elf = (struct ELFHeader *)0x8000;
        readdisk((void *)elf, 4096, 0);

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

上图为 bootmain 的函数代码：

1. 首先将 elf 文件定位在内存 0x8000 处，为了与 esp 栈帧对应。
2. 然后通过读取磁盘文件的函数将 elf 文件从磁盘中加载到内存的对应位置。(elf 文件默认大小为 4096 个字节)

3. 解析 elf 文件，将其中的 program header（程序头表）内容加载到内存对应的位置上。
4. 根据 elf->entry 的值跳转进入 main 函数。(引导程序任务完成)。

Extra：实现在引导程序（保护模式）下的输出（Boot:Hello,world!）

操作方法：在 start.s 文件中，我们可以通过汇编指令更改显存，从而打印我们需要输出的字符串，操作代码如下：

1.

```
movw $GDT_ENTRY(3), %ax
movw %ax, %gs
```

这段代码首先将 gdt 表的第三项（视频段寄存器描述符）的内容加载到 gs 寄存器中，从而使 gs 寄存器为视频段选择子。

2.

```
movb $0x0c, %ah
movl $((80 * 5 + 0) * 2), %edi

movw $Hello, %si
movl $0x12, %ecx
call print
```

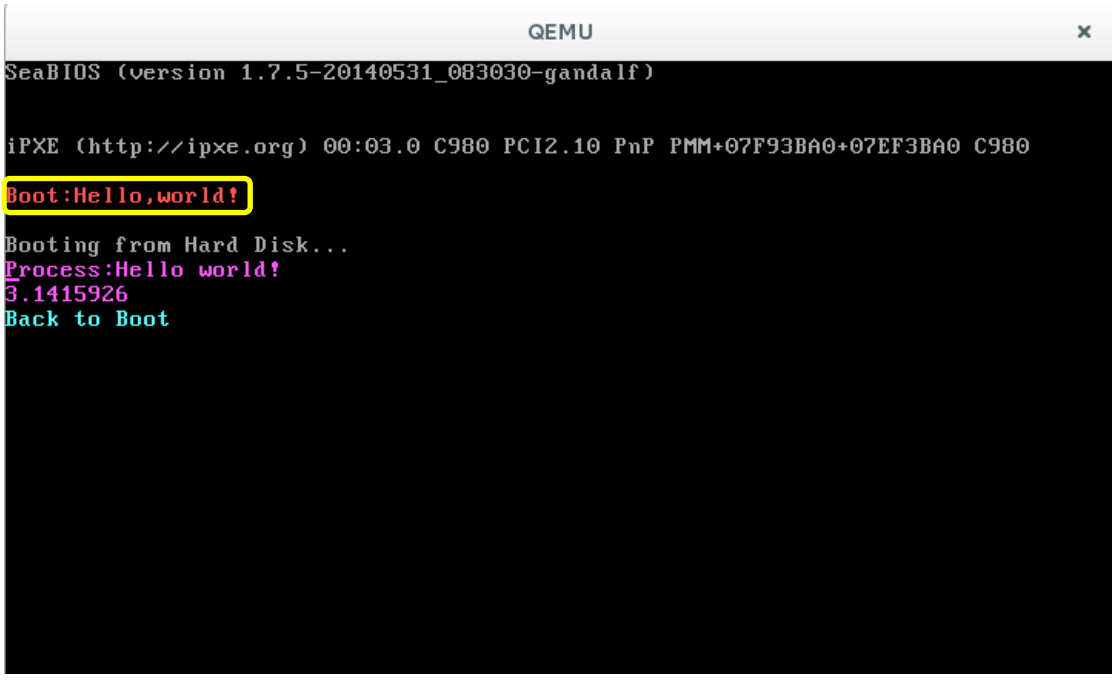
然后将 ah 寄存器赋值（黑底红字），edi 寄存器为显存的行列数（第 5 行第 0 列），ecx 寄存器为所需要输出的字符串字符数，然后将 Hello 字符串通过 print 函数进行输出。

3.

```
print:
        pushl %ebp
        movl %esp, %ebp
l1:
        lodsb
        movw %ax, %gs:(%edi)
        addl $0x2, %edi
        decl %ecx
        cmpl $0, %ecx
        jne l1
        leave
        ret
```

Print 函数如上图所示：其中 lods 指令用来将 si 寄存器中字符串的地址所对应的字符加载到 al 寄存器中，然后将 ax（ah+al）寄存器的内容加载到显存中，对于每个字符进行循环，即可输出整个字符串。

*显示结果如下：



黄色框内即是该字符串的输出。

P.S：

1.对于单个字符的输出，我们仅仅需要将对应字符的ASCII码赋值给al寄存器，然后将ax寄存器的内容加载到gs:（edi）的位置即可，但是对于字符串我们需要对每个字符的ASCII码进行赋值，这无疑加大了代码的冗余量，而且也会不符合MBR小于512字节的要求，所以在这里我们需要将字符串的输出封装成为一个固定的函数（汇编语句组），从而实现代码的重复利用。

2.对于字符串ASCII码的存取：

在汇编代码中，由于我们能力有限，不能像C语言中定义字符串数组并逐个遍历，所以我们可以借鉴讲义中对Hello字符串的定义：

Hello:　.string”Boot:Hello,world!”

注意这里Hello是该字符串的首地址，所以我们需要lods指令，将字符串逐一加载到al寄存器中，从而实现对字符串的存取。

3.MBR大小问题：

在操作系统中，引导程序大小不能超过512字节，所以对于字符串输出这种空间占用较大的程序代码大小有严格限制。在我们这个实验中，根据我的尝试，引导程序最多只能输出hello和back to boot两句话（顶多再加上loading），对于其他字符串的输出则无能为力，所以，

这里无法进行更多字符串的输出。如果有方法能够进行输出，希望助教或者老师能够在实验课上不吝赐教。

- **第二部分：将磁盘中的程序写入到内存的相应位置并运行。**

- **目的：将程序代码加入内存后，对 main 函数里面的 print 函数进行实现，能够返回并无限循环。**

- **操作方法：**

1. 对于程序代码的加载已经在第一部分进行了描述，这里叙述一个重要的问题：

   对磁盘的读写：在 boot.c 文件中我们有了现成的对磁盘扇区进行读写的函数，但是这个并不够，我们需要做到对磁盘任意位置，任意字节进行读写，所以我们需要对这个函数进行封装，封装函数如下：

```
void readdisk(unsigned char *elf, int count, int offset) {
        unsigned char * eelf;
        eelf = elf + count;
        elf -= offset % SECTSIZE;
        offset = offset / SECTSIZE + 1;
        for(; elf < eelf; elf += SECTSIZE, offset++)
                readsect(elf,offset);
}
```

   参数：elf（pa）目的内存的物理地址；

count：所需要读取的磁盘字节数；

offset：源文件的磁盘偏移量；

首先通过 offset 我们可以计算出扇区号，通过 count 可以计算出所需要读取的扇区数，然后我们就可以通过循环调用 readsect 函数来实现对任意字节的读写。

2. 实现完成程序代码的加载之后，我们就会跳转进入 main 函数，这里 main 函数通过调用 print 函数进入 print 函数中。

```
#include "lib.h"

void main()
{
        print("Process:Hello world!",8,0);
        print("3.1415926",9,0);
}
```

3. 重点：实现 lib.h 中的 print 函数：

对于 print 函数我们已经在 start.s 中实现了一次，不过那里我们可以通过汇编语言实现对寄存器和显存的更改。在这里我们仍然需要对显存进行更改，所以我们需要内联汇编（asm）的帮助，程序代码如下：

```
void print(char *str, int line, int row)
{
        asm volatile("movw $0x18, %ax   \n\t"
                     "movw %ax, %gs     \n\t");

        asm volatile("movb $0x0d, %%ch  \n\t"
                     "movl %0, %%edi" : : "g"((80 * line + row) * 2) : "%cx", "%edi")

        int i;
        for(i = 0; str[i] != '\0'; i++)
        {
                asm volatile("movb %0, %%cl             \n\t"
                             "movw %%cx, %%gs:(%%edi)   \n\t"
                             "addl $0x2, %%edi"
                                 : : "g"(str[i]));
        }
}
```

首先我们仍然需要将 gs 设置为视频段寄存器的选择子，所以我们将 gdt 描述符加载到 gs 中（GDT_ENTRY（3） == $0x18）；

然后我们需要设置输出格式：即颜色，位置；

然后我们通过函数参数 str 确定输出字符的 ASCII 码,将其赋值给 al,通过循环进行输出，至此，print 函数执行完毕。

P.S:

1. 内联汇编的书写方式：

通过查询资料我们可以得到基本的内联汇编的书写方式，这里需要说明几点：

1. Volatile限定符为修饰，在内联汇编中可以不使用，使用它的目的是为了让gcc编译时不对代码进行优化，免得代码意思产生偏差。

2. Asm(汇编语言模板:输出:输入:修饰限定)；语言格式中，三个冒号为分割符，将语句分成三个区域：

   A. 汇编语言模板：这里是汇编语言书写的地方，对于简单的内联汇编语句可以只有这一部分的存在，在这里需要严格按照汇编语言的格式进行书写，不能有任何错误。

   B. 输出模块：这里是对汇编语言中产生的输出进行赋值的部分，可以在这里将输出值取出，并在其他代码中使用。

   C. 输入模块：这里可以将C语言中的表达式，常量带入到汇编语言中进行操作，在我们的内联汇编中多次使用这个方式。

D. 修饰限定模块：这里是对汇编语言的描述和补充，不影响语言含义，只是对 gcc 编译器的通知和约束（如对寄存器和内存的改动）。

基本条件确定后：我们可以解读上述 print 函数的代码，我们可以以其中一句话为例进行解读：

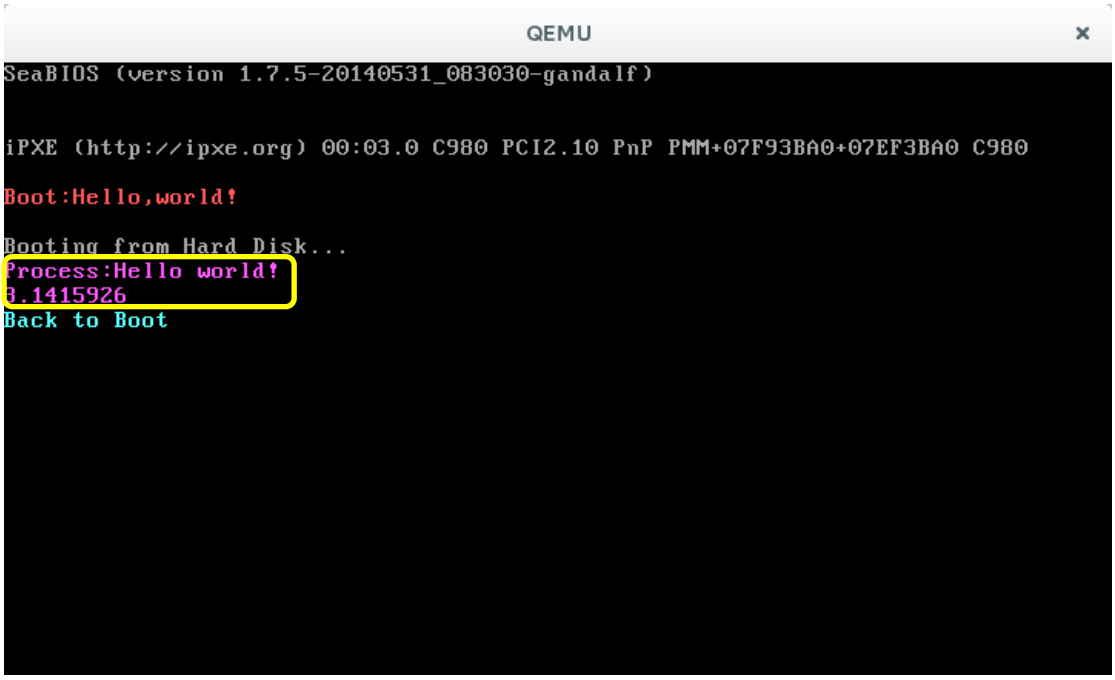
```
asm volatile("movb %0, %%cl                \n\t"
             "movw %%cx, %%gs:(%%edi)      \n\t"
             "addl $0x2, %%edi"
                 : : "g"(str[i]));
```

这是一句完整的内联汇编语句，其中任意两句汇编语言之间用‘\n’或者‘\t’或者‘；’隔开；‘%0’是占位符，它会被后面的输入或者输出语句替换，0 表示对应关系，最大可以到 9；”g”(str[i])是输入，”g”是限制符，它用来决定括号内变量的存储方式，‘i’表示立即数，‘r’表示寄存器，‘m’表示内存，‘g’表示不进行特殊限定，编译器可以根据需要选择存储方式，括号里面是需要替换的表达式（变量），因此这句话是用来将 str[i]的数

值存于 cl 寄存器中。注意在有占位符的语句中，为了对占位符和寄存器进行区分，所有的寄存器前面都要打上%%。

PS：当我将上述代码中的 cx 替换成为 ax 时，这段代码就不能正确的输出了，原因未解。

*输出显示如下：



4. 输出结束后，函数返回到 bootmain，bootmain 返回到 start 中，通过 loop: jmp loop 实现无限循环。至此，lab1 结束。

5. 感想与总结：通过这次 lab，我学到了引导程序的启动过程，以及实模式和保护模式之间的转化，通过这次 lab 也了解了如何对显存进行操作。收获很多，同时希望老师和助教能够进一步对汇编以及输出的过程做进一步的讲解。谢谢老师。