OSLab1 系统引导

冯诗伟 161220039

2018 年 3 月 23 日

1 实验目的

从实模式切换至保护模式，在保护模式下读取磁盘 1 号扇区中的 Hello World 程序至内存中的相应位置，跳转执行该 Hello World 程序，并在终端中打印“Hello, World!”。

2 实验步骤

2.1 实模式到保护模式的切换

首先关闭中断，打开 A20 地址线，加载 GDTR，设置 CR0 的 PE 位（第 0 位）为 1，通过长跳转设置 CS 进入保护模式。相关代码如下。

```
1     cli                                 #关闭中断
2     inb $0x92, %al                      #启动A20总线
3     orb $0x02, %al
4     outb %al, $0x92
5     data32 addr32 lgdt gdtDesc          #加载GDTR
6     movl %cr0, %eax                     #启动保护模式
7     orb $0x01, %al
8     movl %eax, %cr0
9     data32 ljmp $0x08, $start32         #长跳转切换至保护模式
```

start.S 的第一句就是 cli 指令，直到 IDT 初始化结束才能用 sti 指令开中断。这是因为 IDT 初始化结束前 OS 不能正常的响应中断。

启动 A20 总线是由于在保护模式下 (主要针对 80286 之后的芯片)，如果 A20 被禁止，则第 20 位在做 CPU 地址访问的时候是无效的，永远只能被作为 0。对于 80386 的 32-bit 来说最大寻址为 0xFFFEFFFF，即访问的内存只能是奇数 MB 段，1M、3M、5M……只有开启了 A20 地址线后，才能访问连续的内存地址。

第 5 行用 lgdt 指令加载 GDTR 的 limit 和 base，根据下面 gdtDesc 的设置可以看出，16bit 的长度限制，32bit 的基地址 (即 gdt)。

```
1     gdtDesc:
2         .word (gdtDesc - gdt -1)
3         .long gdt
```

第 6–8 行是将%cr0 的最低位 PE 位设置为 1，开启保护模式。

第 9 行是使用 ljmp 指令装载 cs 段寄存器和 eip。

进入保护模式之后，初始化段寄存器和 esp, 初始化之后就跳转至 bootMain。相关代码如下。

```
1   # Set up the protected-mode data segment registers
2         movw $0x10,%ax
3         movw %ax,%ds
4         movw %ax,%ss
5         movw %ax,%es
6         movw %ax,%fs
7         movw $0x18,%ax
8         movw %ax,%gs
9   # Set up a stack for C code.
10        movl $0, %ebp
11        movl $(128 << 20), %esp
12        sub $16, %esp
13        jmp bootMain
```

不过值得注意的有以下两点：

1、通过 objdump 指令反汇编 start.o 发现这里的 mov 是 8E，不是普通的 mov 指令，将寄存器或内存单元的内容移到段寄存器的指令，所以不能直接将立即数移动到段寄存器中，而是用一个通用寄存器中转一下。

2、在 app.s 中关于通过写显存打印字符的代码中发现%gs 是有关视频段的段寄存器，所以将视频段选择子存入%gs 中。下面来看一下视频段的具体位置。根据这里的 gdt 的信息我们看到，每个表项由 2 个 word 和 4 个 byte 来初始化，共 64bit。视频段的起始位置是 3*8=24bytes, 所以将%gs 赋值为 0x18。

```
1  gdt:
2      .word 0,0                          #GDT第一个表项必须为空
3      .byte 0,0,0,0
4
5      .word 0xffff,0                     #代码段描述符
6      .byte 0,0x9a,0xcf,0
7
8      .word 0xffff,0                     #数据段描述符
9      .byte 0,0x92,0xcf,0
10
11     .word 0xffff,0x8000                #视频段描述符
12     .byte 0x0b,0x92,0xcf,0
```

2.2 加载磁盘中的程序

由于中断关闭，无法通过陷入磁盘中断调用 BIOS 进行磁盘读取，在框架代码中使用代码框架中实现了的 readSec(void *dst, int offset) 这一接口，readSec() 通过读写（in，out 指令）磁盘的相应端口来实现磁盘

特定扇区的读取。

```
void bootMain(void) {
        void (*elf)(void);
        // loading sector 1 to memory
        readSect((void*)0x8c00,1);
        elf=(void*)0x8c00;
        elf();
}
```

通过上述接口读取磁盘 MBR 之后扇区中的程序至内存的特定位置并跳转执行，从下面的 app/Makefile 中可以看出该 Hello World 程序入口地址为 0x8c00。程序加载结束后，把函数指针 elf 也赋值为 0x8c00。

```
app.bin: app.s
        gcc -c -m32 app.s -o app.o
        ld -m elf_i386 -e start -Ttext 0x8c00 app.o -o app.elf
        objcopy -S -j .text -O binary app.elf app.bin
```

2.3 运行 HelloWorld 程序

此时 eip=0x8c00，将通过写显存一个一个地打印“Hello, World!”中的字符。具体实现如下：

```
    movl $((80*5+0)*2), %edi                #在第5行第0列打印
    movb $0x0c, %ah                         #黑底红字
    movb $72, %al                           #72为H的ASCII码
    movw %ax, %gs:(%edi)                    #写显存
```

这样就可以打印出‘H’字符，剩下的 12 个字符大同小异，只需修改第一行汇编代码中的列和第三行汇编代码中的 ASCII 码即可。最后再加上一句‘jmp start’，跳至此段代码的最开始，反复打印，否则会执行大量的未初始化的无用代码，程序会报错。

3 实验收获

1、更加深入地了解了系统启动的基本过程。
2、对段寄存器、段选择子、段描述符表、段描述符等概念有了更加直观和细致的认识。
3、发现自己并不是很了解 C 语言中的函数指针的相关用法。

4 实验中遇到的问题

1、最初没有发现%gs 寄存器和视频段的关系，对%gs 的初始化出了一点问题。
2、对 elf 函数指针进行赋值之后并没有再执行‘elf()’，导致一直没有打印字符串成功。
3、最初并不是很能看懂汇编代码中 gdt 和 gdtDesc 相关的代码，后来查阅资料后发现其实就是按照段描述符的格式来进行初始化。