操作系统实验三　操作系统的引导

一、实验目的

- 熟悉实验环境；
- 建立对操作系统引导过程的深入认识；
- 掌握操作系统的基本开发过程；
- 能对操作系统代码进行简单的控制，揭开操作系统的神秘面纱。

二、实验环境

Vmware 17.5.1，Ubuntu 20.04，Bochs 2.4.6

三、实验内容

- 阅读《Linux 内核完全注释》第 6 章，对计算机和 Linux 0.11 的引导过程进行初步了解；
- 改写 Linux 0.11 的引导程序 bootsect.s，打印指定的提示信息；
- 改写 Linux 0.11 的进入保护模式前的设置程序 setup.s，使得 bootsect.s 能完成 setup.s 的载入，并跳转到 setup.s 开始地址执行。setup.s 获取基本硬件参数，且不再加载 Linux 内核，仅输出提示和硬件信息。

四、实验过程

4.1 改写 bootsect.s

基于 Linux-0.11/boot 目录下的 bootsect.s 进行改写，使其能在屏幕上打印一段提示信息“LiuzikangOS is booting...”。

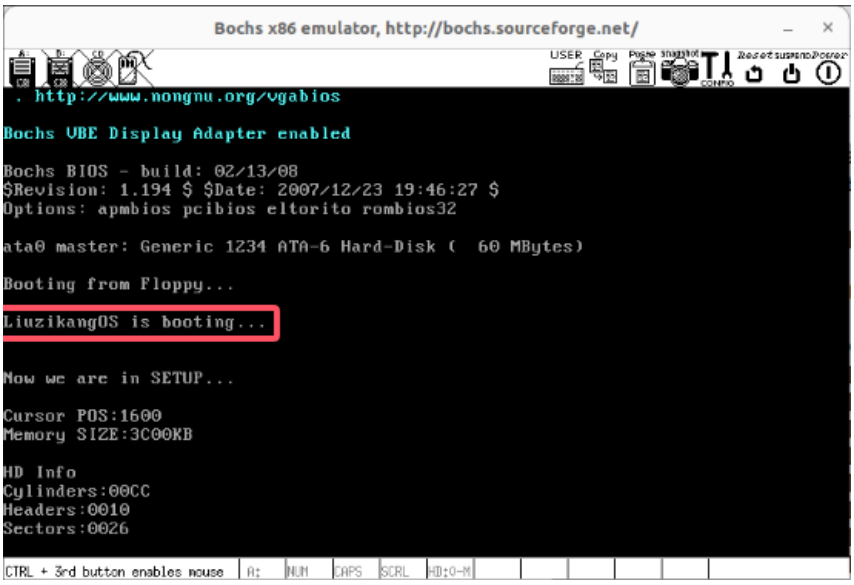
修改打印的内容 msg1 如下：

```
244 msg1:
245         .byte 13,10
246         .ascii "LiuzikangOS is booting..."
247         .byte 13,10,13,10
```

找到屏幕显示部分的代码，其将 msg1 输出到屏幕。修改打印内容的长度为 25+2*3=31（“LiuzikangOS is booting...”25 字节，开头 1 个回车换行，结尾 2 个回车换行）。

```
 92 ! Print some inane message
 93
 94         mov     ah,#0x03                ! read cursor pos
 95         xor     bh,bh
 96         int     0x10
 97
 98         mov     cx,#31
 99         mov     bx,#0x0007              ! page 0, attribute 7 (normal)
100         mov     bp,#msg1
101         mov     ax,#0x1301              ! write string, move cursor
102         int     0x10
```

编译运行：在 linux-0.11/boot/目录下执行“as86 -0 -a -o bootsect.o bootsect.s”和“ld86 -0 -s -o bootsect bootsect.o”命令编译和链接 bootsect.s。执行“dd bs=1 if=bootsect of=Image skip=32”命令，生成映像文件，并去掉 32 字节的 Minix 可执行文件头部。将生成的 Image 文件拷贝到 linux-0.11/目录下，并在 oslab/目录下使用“./run”命令运行程序。



4.2 改写 setup.s

基于 Linux-0.11/boot 目录下的 setup.s 进行改写，使得 bootsect.s 能完成 setup.s 的载入，并跳转到 setup.s 开始地址执行。且 setup.s 向屏幕输出一行“Now we are in SETUP…”；获取基本的硬件参数（如内存参数、显卡参数、硬盘参数等）将其存放在内存的特定地址，并输出到屏幕上；不再加载 Linux 内核，仅需保持上述信息显示在屏幕上。

4.2.1 向屏幕输出指定信息

参照 bootsect.s 中打印信息的代码改写 setup.s，注意在开始时修改 es 的值（段偏移地址）为 cs，打印内容为“Now we are in SETUP...”：

```
181 msg1:
182         .byte 13,10
183         .ascii "Now we are in SETUP..."
184         .byte 13,10,13,10
```

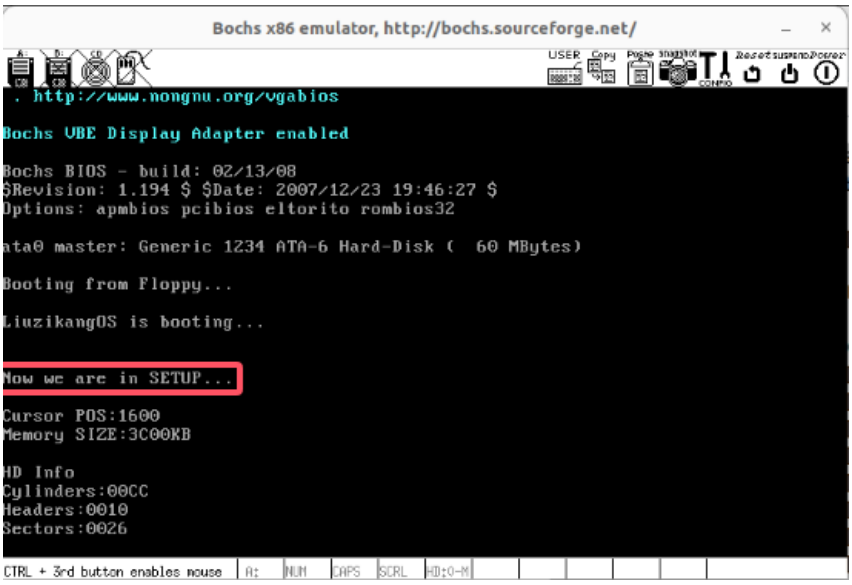
修改长度为 22+2*3=28：

```
32          mov     ah,#0x03                ! read cursor pos
33          xor     bh,bh
34          int     0x10
35
36          mov     cx,#28
37          mov     bp,#msg1
38          call    print_msg
```

build.c 从命令行参数得到 bootsect、setup 和 system 内核的文件名，将三者做简单的整理后一起写入 Image 文件。为了能借助 Makefile 编译 bootsect.s 和 setup.s，需要修改 linux-0.11/tools/目录下的 build.c，将 178 行处代码修改如下，使得 argv[3]为“none”时忽略 system 相关程序，只写入 bootsect 和 setup。

```
178         if (strcmp(argv[3], "none") == 0)
179                 return 0;
```

编译运行：在 linux-0.11/目录下执行“make BootImage”生成映像文件，并在 oslab/目录下使用“./run”命令运行程序。



4.2.2 获取基本硬件参数

用 ah=#0x03 调用 0x10 中断可以读出光标的位置，用 ah=#0x88 调用 0x15 中断可以读出内存的大小。中断向量表中 int 0x41 的中断向量位置(4*0x41 = 0x0000:0x0104)存放的并不是中断程序的地址，而是第一个硬盘的基本参数表。每个硬盘参数表有 16 个字节大小。

根据实验手册，修改 setup.s，获取基本硬件参数（光标位置、内存大小和硬盘参数等）并存放在 0x90000 处。

```
40 ! 获取基本硬件信息
41 ! ok, the read went well so we get current cursor position and save it for posterity.
42         mov     ax,#INITSEG
43         mov     ds,ax           ! set ds=0x9000
44         mov     ah,#0x03        ! read cursor pos
45         xor     bh,bh
46         int     0x10
47         mov     [0],dx
48
49 ! Get memory size (extended mem, kB)
50         mov     ah,#0x88
51         int     0x15
52         mov     [2],ax
53
54 ! Get hd0 data
55         mov     ax,#0x0000
56         mov     ds,ax
57         lds     si,[4*0x41]     ! 中断向量偏移
58         mov     ax,#INITSEG
59         mov     es,ax           ! 目标地址0x9000
60         mov     di,#0x0004
61         mov     cx,#0x10        ! 重复16次
62         rep
63         movsb
```

需要输出的信息：

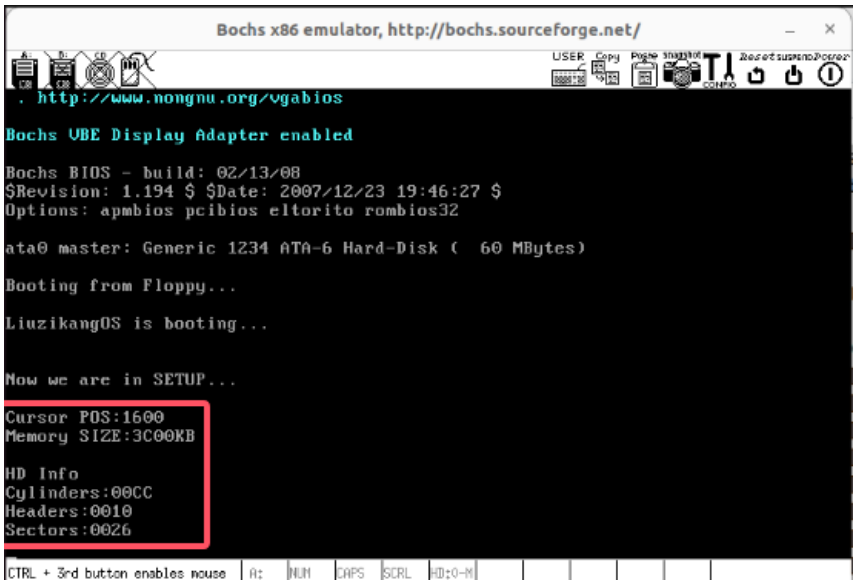
```
msg1:
        .byte 13,10
        .ascii "Now we are in SETUP..."
        .byte 13,10,13,10

msg2:
        .ascii "Cursor POS:"    ! 11

msg3:
        .ascii "Memory SIZE:"   ! 12

msg4:
        .ascii "KB"             ! 2

msg5:
        .ascii "HD Info"        ! 7

msg6:
        .ascii "Cylinders:"     ! 10 柱面数

msg7:
        .ascii "Headers:"       ! 8 磁头数

msg8:
        .ascii "Sectors:"       ! 8 扇区数
```

添加相关 print 代码，将这些信息以 16 进制形式在屏幕上显示出来。

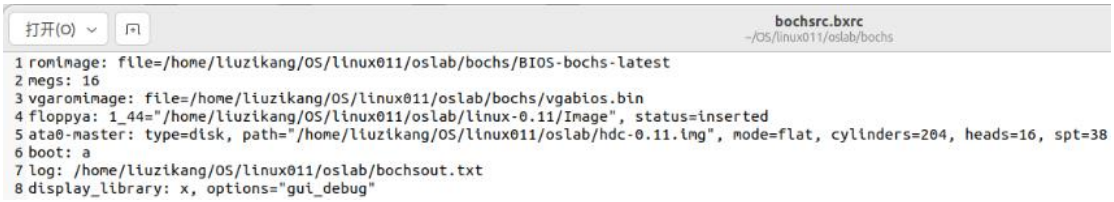
```
! 以16进制方式打印栈顶的16位数
print_hex:
        mov     ax,#INITSEG     ! 初始化es
        mov     es,ax
        mov     cx,#4           ! 计数器，4个十六进制数字
        mov     dx,(bp)         ! 将(bp)所指的值放入dx中，如果bp是指向栈顶的话
print_digit:
        rol     dx,#4           ! 循环以使低4比特用上 !! 取dx的高4比特移到低4比特处。
        mov     ax,#0xe0f       ! ah = 请求的功能值，al = 半字节(4个比特)掩码。
        and     al,dl           ! 取dl的低4比特值。
        add     al,#0x30        ! 给al数字加上十六进制0x30
        cmp     al,#0x3a
        jl      outp            ! 是一个不大于十的数字
        add     al,#0x07        ! 是a～f，要多加7
outp:
        int     0x10
        loop    print_digit
        ret

! 打印回车换行
print_nl:
        mov     ax,#0xe0d       ! CR
        int     0x10
        mov     al,#0xa         ! LF
        int     0x10
        ret

! 将打印也封装成函数，注意调用前需要把cx赋字符串长度，bp赋字符串位置
print_msg:
        mov     ax,#SETUPSEG
        mov     es,ax           ! 初始化es
        mov     bx, #0x0007
        mov     ax, #0x1301
        int     0x10
        ret
```

后续参照“Now we are in SETUP…”的输出方式，调用相关 print 函数打印各硬件信息。

编译运行：在 linux-0.11/目录下执行“make BootImage”生成映像文件，并在 oslab/目录下使用“./run”命令运行程序。



将结果与 Bochs 配置文件对比，硬件信息一致。



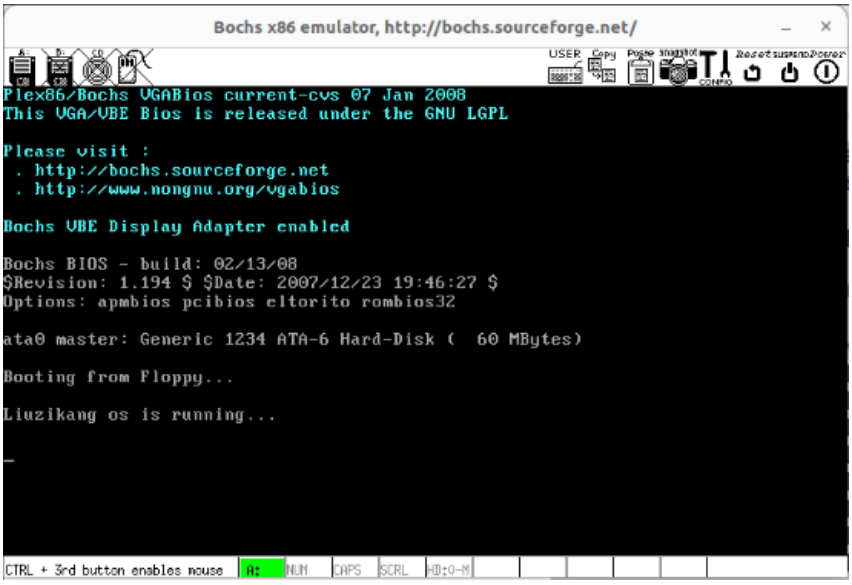
4.2.3 不再加载 Linux 内核

在输出提示信息和硬件信息后，添加如下代码，使得程序进入死循环，setup.s 不再加载 Linux 内核。

```
141 ! 进入死循环,不再加载Linux内核
142 pause:
143         jmp     pause
```

五、实验结果

此次实验通过修改 Linux 0.11 的 bootsect.s、setup.s 代码，控制操作系统在引导程序过程实现指定输出，提高了汇编代码能力，对操作系统的引导过程有了更深刻的理解。



有时，继承传统意味着别手蹩脚。 x86 计算机为了向下兼容，导致启动过程比较复杂。请找出 x86 计算机启动过程中，被硬件强制，软件必须遵守的两个“多此一举”的步骤（多找几个也无妨），说说它们为什么多此一举，并设计更简洁的替代方案。

1.POST（Power-On Self-Test）

在计算机启动时，BIOS 会执行一个自检过程，检查系统硬件（如 CPU、内存、显卡、硬盘等）是否正常。这是为了确保硬件在启动时没有问题，通常通过一系列“滴滴”声（或不发声）来反馈硬件状态。

• 为什么多此一举：

在现代计算机中，大部分硬件（特别是内存和硬盘）早已可以通过更高效的硬件检测工具来检查，而无需依赖 BIOS 执行硬件自检。并且 ST 会拖慢启动时间，尤其是当自检过程中出现故障时，需要更长时间来进行排查和恢复。

• 简洁替代方案：

可以通过硬件直接提供自检信息，或直接将自检过程与操作系统的启动整合。例如，在主板上增加一个硬件级别的诊断模块，可以在系统开机后直接向操作系统报告硬件健康状况，而不是由 BIOS 手动执行这个过程。对于较为复杂的硬件，可以在操作系统加载后由专门的驱动程序进行检测，BIOS 则不再承担硬件检查任务。

2. MBR（Master Boot Record）引导方式

传统的 x86 计算机使用 MBR 作为启动引导方式。MBR 是存储在硬盘的第一个扇区（0x1F1）中的一个小程序，它包含操作系统的启动加载器。启动时，BIOS 会通过读取 MBR 中的引导代码来启动操作系统。

• 为什么多此一举：

MBR 在多分区硬盘和大容量存储设备上存在许多局限性（如最大支持 2TB 的硬盘，分区数量有限等）。此外，它的引导过程非常简单，通常需要多个阶段，涉及从 MBR 加载到操作系统引导加载器的额外步骤，比较低效。

BIOS 和 MBR 的组合是上世纪 80 年代末的设计，当时的硬件和需求完全不同。如今的 UEFI（统一可扩展固件接口）和 GPT（GUID 分区表）已经可以提供更强大的功能和更快的启动速度。

• 简洁替代方案：

UEFI 是一个现代的固件接口，替代了传统的 BIOS，支持更高效的启动方式，并且能够支持大容量硬盘（超过 2TB）和更多的分区。因此可以让操作系统的引导过程直接通过 UEFI 协议，而不需要通过 MBR 和传统引导加载器的多次跳转。

3. BIOS 中硬件的配置管理

在传统的 BIOS 中，用户可以通过 BIOS 设置界面手动配置硬件的启动顺序、启用或禁用某些硬件组件（如 USB、硬盘等）。

• 为什么多此一举：

现代操作系统可以更高效地自动识别硬件并根据需求进行配置，而不需要在每次启动时进入 BIOS 进行手动设置。并且许多硬件和固件现在支持即插即用和热插拔，不需要每次启动时都手动配置硬件。

• 简洁替代方案：

在硬件级别实现自动配置，让操作系统在启动过程中自动识别和配置硬件，而不是依赖传统的 BIOS 设置界面。例如，可以通过固件与操作系统之间的标准化协议（如 ACPI）自动配置硬件。