操作系统的引导-实验报告

2021113117 王宇轩

1. 改写bootsect.s

一开始用新版本实验环境整体Linux 0.11可以正常编译运行，但编译运行bootsect.s时一直出现问题，就想着先用旧版本实验环境试试，以下实验是在旧版本实验环境中进行的。

对bootsect.s，改写`msg1`部分：

```
msg1:
    .byte 13,10
    .ascii "WYX os is booting..."
    .byte 13,10,13,10
```

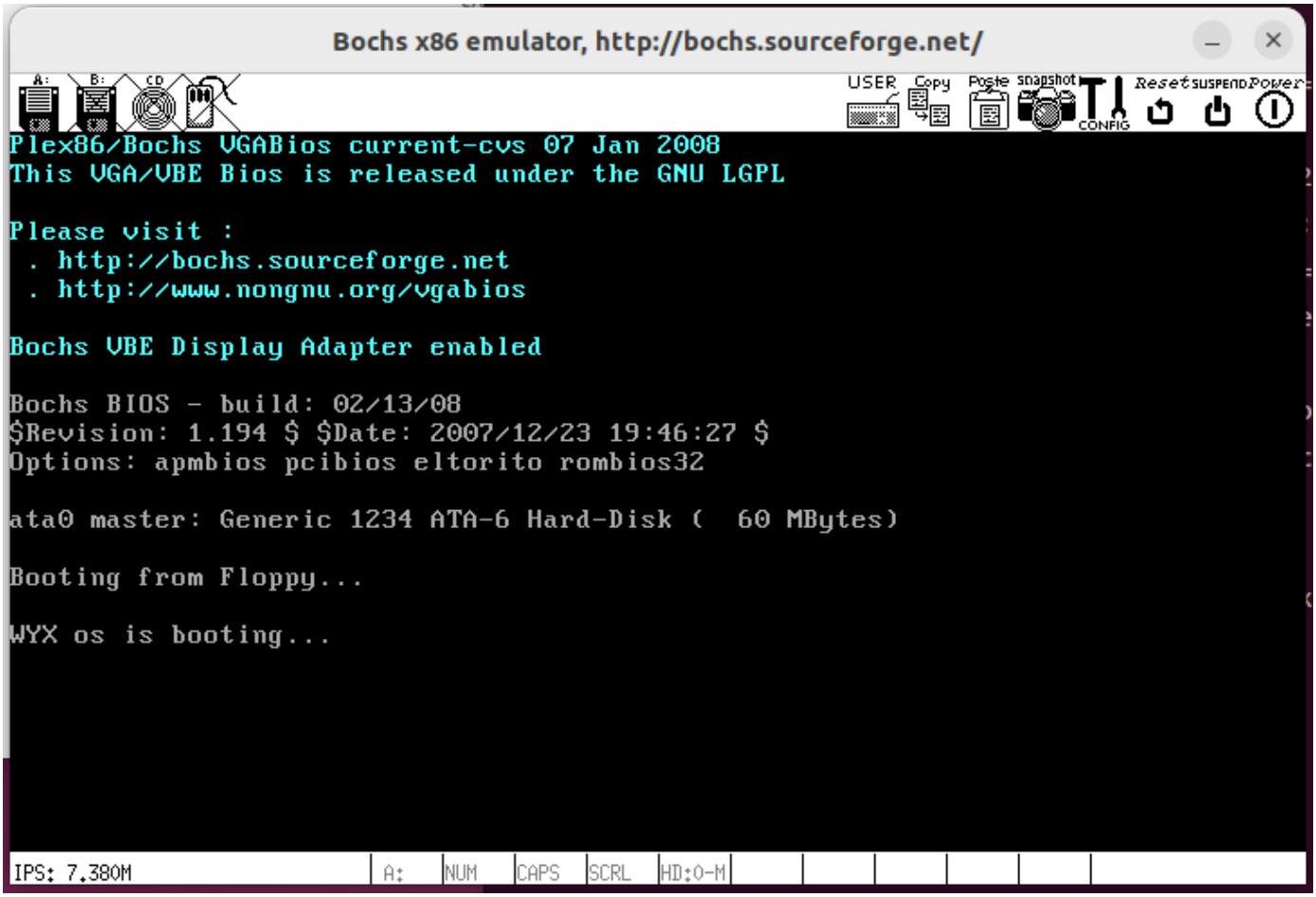
改写字符串长度：

```
! Print some inane message

    mov ah,#0x03        ! read cursor pos
    xor bh,bh
    int 0x10

    mov cx,#26
    mov bx,#0x0007      ! page 0, attribute 7 (normal)
    mov bp,#msg1
    mov ax,#0x1301      ! write string, move cursor
    int 0x10
```

随后编译运行。在`./run`的时候，bochs遇到了问题，是和`display_library: sdl`，相关的。查阅2020新版本实验环境的linux-0.11.bxrc，发现其中`display_library: x`，照此修改对应的bxrc文件后，恢复正常。运行显示如下：



2. 改写setup.s

这里想说明一下，我的Ubuntu虚拟机里面没装输入法，因此我写的代码注释也是英文的。

首先，去掉setup.s中所有我们不需要的部分，仅保留读取设备信息部分代码，且读硬盘参数按照实验提示上的对应代码进行替换：

```
!从0x41处拷贝16个字节（磁盘参数表）
    mov    ax,#0x0000
    mov    ds,ax
    lds    si,[4*0x41]
    mov    ax,#INITSEG
    mov    es,ax
    mov    di,#0x0004
    mov    cx,#0x10
    rep             !重复16次
    movsb
```

然后在`start`处添加显示信息代码，与我们在bootsect.s中所作的工作类似：

```
    mov ah,#0x03        ! read cursor pos
    xor bh,bh
    int 0x10
    mov cx,#28
```

```
    mov bx,#0x0006
    mov bp,#msg2
    mov ax,cs
    mov es,ax
    mov ax,#0x1301
    int 0x10
```

其中，`msg2`和其他一众提示信息如下：

```
msg2:
    .byte 13,10
    .ascii "Now we are in SETUP..."
    .byte 13,10,13,10
msg_cursor:
    .byte 13, 10
    .ascii "Cursor Position:"
msg_mem:
    .byte 13,10
    .ascii "Memory Size:"
msg_mem2:
    .ascii "KB"
msg_vc:
    .byte 13,10
    .ascii "Display Page:"
msg_vc2:
    .byte 13,10
    .ascii "Video Mode and Window Width:"
msg_cy:
    .byte 13,10
    .ascii "Cylinders:"
msg_hd:
    .byte 13,10
    .ascii "Headers:"
msg_sec:
    .byte 13,10
    .ascii "Sectors:"
```

当设备信息读取完后，我们开始显示。提示信息（如`"Cursor Position:"`）的显示方式与之前类似，对于16进制数值的显示调用实验手册上的代码，如下。此处删去了`mov dx,(bp)`一句，因为我们可以调用`print_hex`前直接把内存中的数取至`dx`。

```
!以16进制方式打印栈顶的16位数
print_hex:
        mov     cx,#4           ! 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
! we have directly put the values in dx so we don't need this
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
```

以显示光标位置信息为例，代码如下。其他的设备信息都可以类似的方式进行显示。

```
! print cursur position
    mov ah,#0x03        ! read cursor pos
    xor bh,bh
    int 0x10
    mov cx,#18
    mov bx,#0x0006
    mov bp,#msg_cursor
    mov ax,#0x1301
    int 0x10
    mov dx,[0]      ! cursor position
    call    print_hex
```

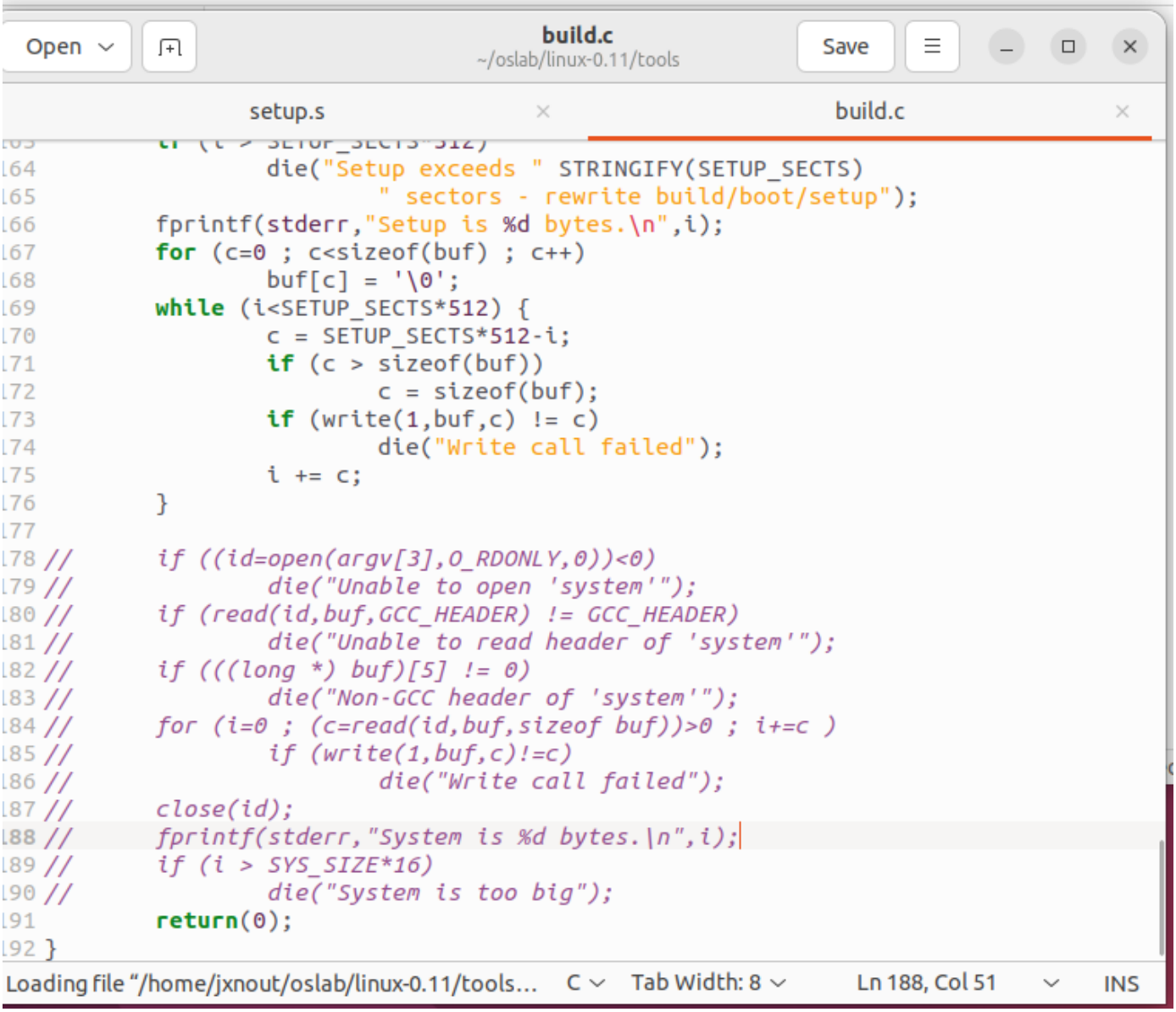
3. 改写bootsect.s装载setup

此处，我们不再需要bootsect在内存中移动、长跳转，可以直接让bootsect把setup载入自己后面，直接执行。由于bootsect大小为512KB，0x7c00处开始执行，所以可把 `SETUPSEG` 设为0x7e00，而把读取的扇区数 `SETUPLEN` 改为2。删去bootsect.s中其他无用代码（如设置GDT、IDT，有关装载system的部分等等）删去。具体请见源代码。

我还顺便修改了一下显示 `WYX os is booting...` 信息的属性设置（ `bl` 寄存器值），添加了高亮、颜色、闪烁效果。

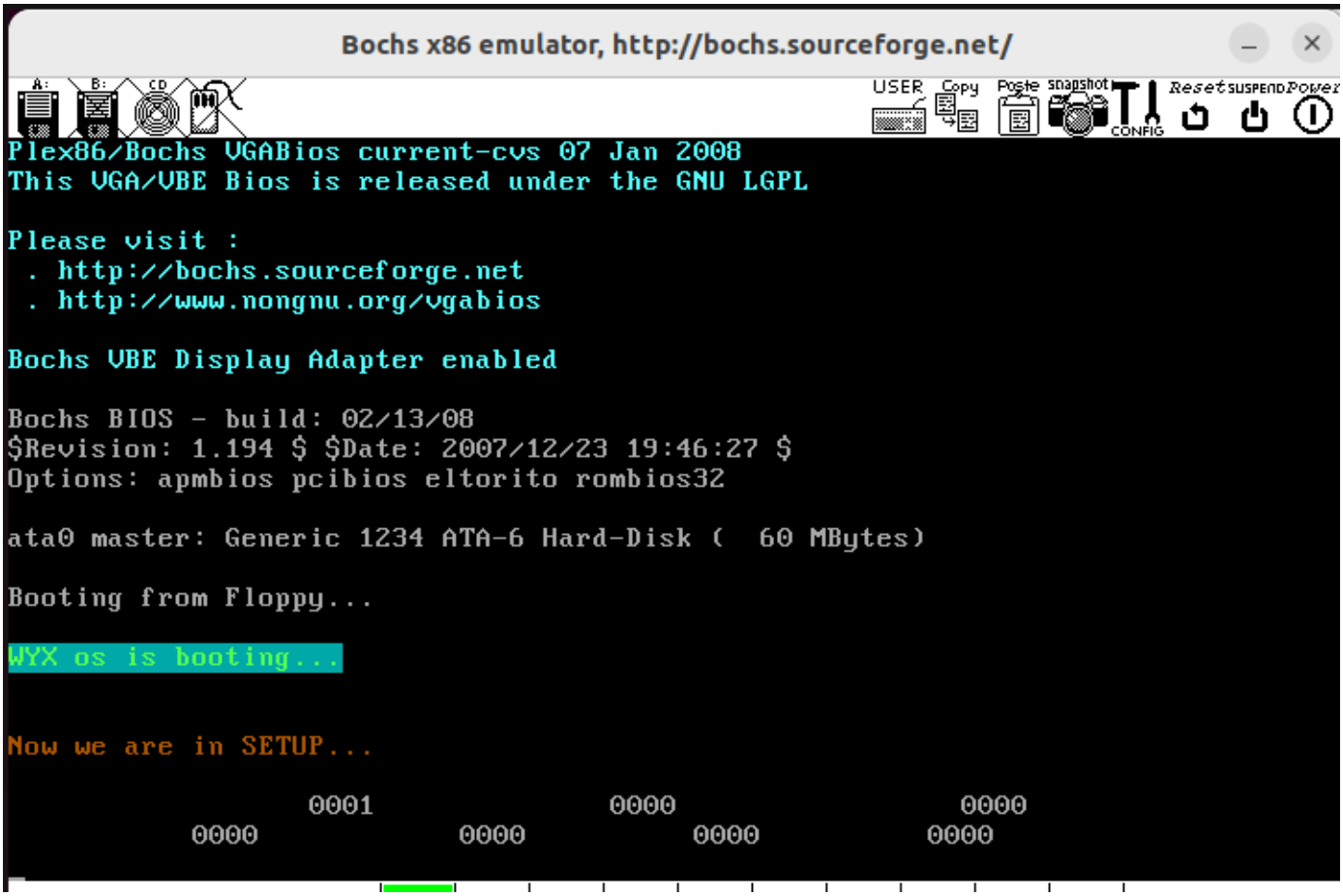
4. 改写build.c

直接把第三个参数相关代码注释掉即可，如下。



4. 试运行

编译，调试运行，输出结果如下。



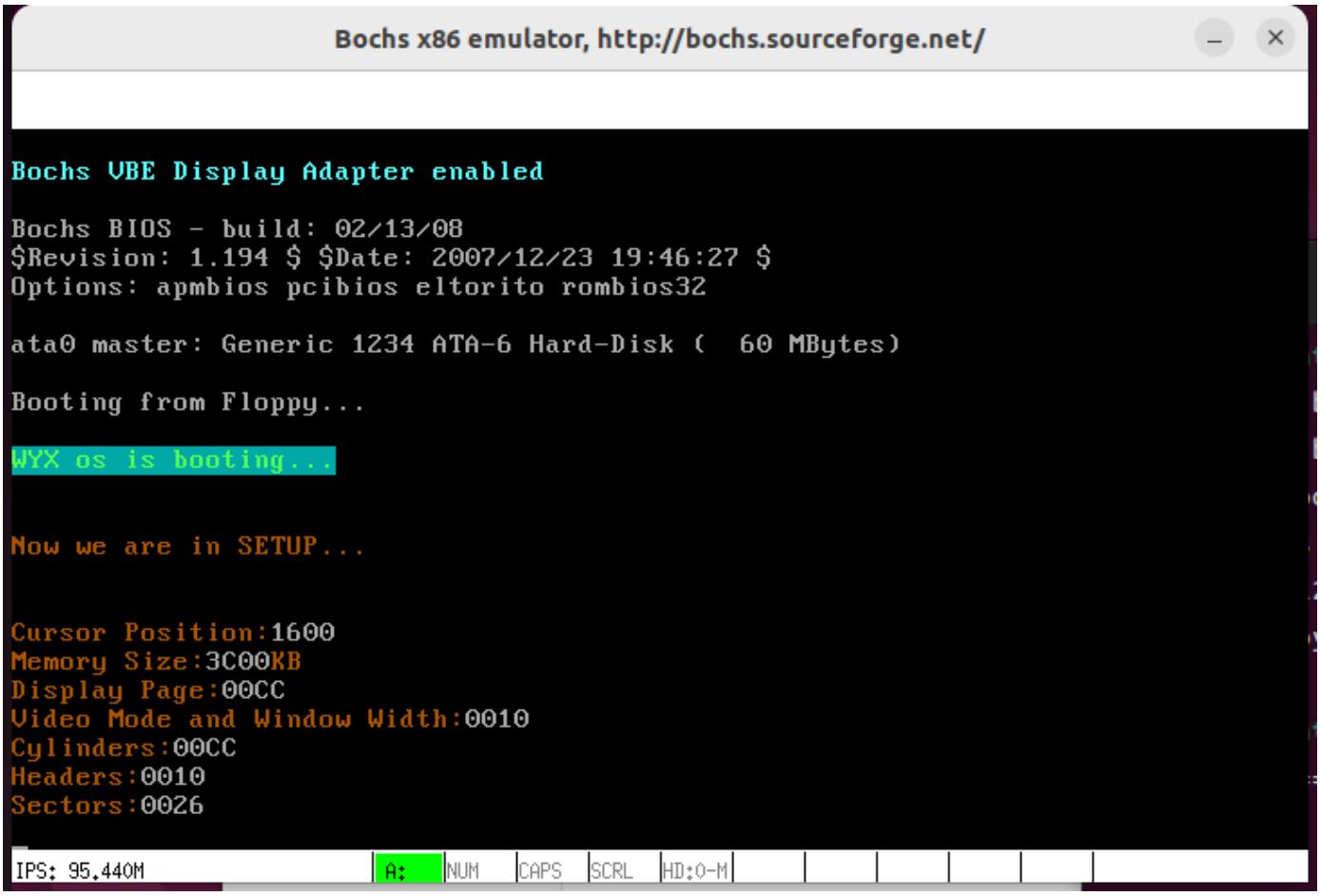
为什么会这样呢?

5. 修正setup.s

经过一段时间的调试和思考，是寻址出了问题，具体而言，是读磁盘参数时，段寄存器被修改了（`0x0000`），之后在定位显示内容的时候自然会出错。我们把它改回来，重新指向`INITSEG`就好了。

```
! Reset rigisters
        mov ax, cs
        mov es, ax
        mov ax, #INITSEG
        mov ds, ax
        mov ss, ax
```

显示结果如下，是正确无误的。



6. 问题回答

1. 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。
    i. bootsect.s先载入内存0x7c00处，随后又移动到0x90000处继续执行： **原因**：因为当时 system 模块的长度不会超过 0x80000 字节大小（即 512KB），所以 bootsect 程序把 system模块读入物理地址 0x10000 开始位置处时并不会覆盖在 0x90000（576KB）处开始的 bootsect 和 setup 模块上。 **替代方案**：可以把bootsect直接加载到0x90000或更后部执行，从而不用跳转。
    ii. 整个系统从地址 0x10000 移至 0x0000 处而不直接在内存起始处加载system： **原因**：这是因为随后执行的 setup 开始部分的代码还需要利用 ROM BIOS 提供的中断调用功能来获取有关机器配置的一些参数（例如显示卡模式、硬盘参数表等）。而当 BIOS 初始化时会在物理内存开始处放置一个大小为 0x400 字节(1KB)的中断向量表，直接把系统模块放在物理内存开始处将导致该中断向量表被覆盖掉。因此引导程序需要在使用完 BIOS 的中断调用后才能将这个区域覆盖掉。 **替代方案**：将中断向量表放置在其他区域，直接将系统加载入内存开始处。