思考题

> 指导：**己思考、练习，不需要提交**

1，为什么计算机启动最开始的时候执行的是 **BIOS代码** 而不是操作系统自身的代码?

> 计算机启动的时候，内存里还没有操作系统，也没有任何其他的任何可执行代码、数据，我们需要从外存（硬盘）里把指令、数据调入内存，然而该调用指令本身也需要从某些存储器里调进内存，于是只好设置一个用硬件实现的可读存储器，在加电之后把其中固化的指令自动调入内存，进而根据这些指令进一步地从外存里调用数据。这种固化的指令就是 BIOS。

2，为什么 BIOS 只加载了一个扇区，后续扇区却是由 `bootsect` 代码加载? 为什么 BIOS 没有把所有需要加载的扇区都加载?

> 这与 BIOS 有关。BIOS 接到启动操作系统的命令之后，约定只从启动扇区把代码加载到 `0x07c00`，至于这个首扇区中是不是系统的启动程序，BIOS 不会关心。这保证了 BIOS 可以启动任意的操作系统，前提是这些系统都把自己的引导信息放到启动磁盘的首个扇区。如果 BIOS 要将所有需要的扇区都加载进去，那么很显然会需要 BIOS 判断要加载几个扇区、所启动的是什么系统，这无疑会大大加深工作难度，而且没有必要。

3，为什么 BIOS 把 `bootsect` 加载到 `0x07c00` ，而不是 `0x00000` ? 加载后又马上挪到 `0x90000` 处，是何道理? 为什么不一次加载到位?

> 问题 1：BIOS 先前已经在内存在开始的地方，用 1KB 的空间构建了中断向量表（`0x000` - `0x3FF`），如果把 `bootsect` 加载到内存开始的地方将会覆盖 BIOS 中断向量表；该问题在教材的第 9 页下方的点评处有说明。

> 问题 2：BIOS 与 Linux本质上是两套程序，所以不一定有什么必然关联性；进一步地讲，BIOS 的行为很机械化，操作系统应该去适配 BIOS。BIOS 先把启动磁盘的首个扇区内容加载到 `0x7c00`，然后把执行权交给 `0x07c00` 之后的那些代码（即 `CS:IP` 指向了 `ox07c00`），这之后的操作与 BIOS 无关，是 Linux 的决定。

4，`bootsect`、`setup`、`head` 程序之间是怎么衔接的? 给出代码证据。

> `bootsect` 是被 BIOS 调入内存的。`setup` 程序位于启动磁盘的2、3、4、5 号扇区，这些扇区是被 `bootsect` 程序调用 BIOS 中断向量表（`0x000` - `0x3FF`，占用 1KB）的 `0x13` 中断调入内存的。

> `bootsect` 可以调用 BIOS 中的中断向量，并可以通过写入寄存器的方式，给中断服务器程序传递参数。`bootsect` 通过语句 `jmpi 0,SETUPSEG` 将控制权转移给 `setup` 之后，继续进行磁盘文件载入工作，它借助 `0x13` 中断，继续将 6号扇区及其之后的 239 个扇区调入内存，这一长达 240 个磁盘扇区的模块是 `system` 模块。完成这部分调用，并再次确定设备号之后，`bootsect` 程序完成使命。

```
// boot/bootsect.s

SETUPLEN = 4            // bootsect.s 里设置了该变量，意思是setup要占用 4 个扇区
SETUPSEG = 0x9020       // 由于自我复制之后，bootsect 程序
                        // 存在于 0x9000之后的 512B 空间里，
                        // 所以复制 setup 程序且紧挨着 bootsect 的话，
                        // 就需要从 0x90200 位置开始写。
```

```
...

load_setup:
    mov dx,#0x0000      ! drive 0, head 0
    mov cx,#0x0002      ! sector 2, track 0
    mov bx,#0x0200      ! address = 512, in INITSEG
    mov ax,#0x0200+SETUPLEN ! service 2, nr of sectors
    int 0x13            ! read it       // 调用 0x13 中断
    jnc ok_load_setup   ! ok - continue
    mov dx,#0x0000
    mov ax,#0x0000      ! reset the diskette
    int 0x13
    j   load_setup
...
! after that (everyting loaded), we jump to
! the setup-routine loaded directly after
! the bootblock:

    jmpi    0,SETUPSEG  // 这句话说明开始跳转到 setup 的地方,
                        // 即 0x90200。在此之后, bootsect 将会失去作用,
                        // 等待它的是被覆盖。
```

setup 接手后，首先提取机器系统数据，然后把这部分数据写入 0x90000，以备 main 函数使用，同时这部分内存原先被 bootsect 占用，现在 bootsect 没用了，可覆盖腾出空间。

> setup 将机器系统数据写入 bootsect 的位置之后，开始关闭系统的中断响应，并将 system 模块覆盖到内存的首部。由于 system 模块里面有系统代码，也有 head 程序，而且后者处于 system 的前端，所以在此之后，内存首部就是 head 程序。

> 设置初始化 IDT 和 GDT 之后，setup 打开 A20 地址线。之后为了开启保护模式，setup 重写可编程中断控制器 8259A，然后将第 0 号 32 位寄存器 CR0 的 PE 位（即第 0 位，可称之为 CR0 的 0 位）设置为 1 并通过语句 jmpi 0,8 这句跳转，指向了保护模式某个中断，让系统开始执行内存头部的 head 程序。

5，setup 程序里的 cli 是为了什么？

> cli 即禁止响应系统中的任何中断，实际操作是将 CPU 的 **标志寄存器**（EFLAGS，32位）中的中断允许标志（IF，第 9 位）设置为 0。**此处即将进行实模式下的中断向量表与保护模式下的中断描述符表交接，如果此时仍可以响应中断，那么中断的到来就会不得不面临实模式中断机制废除、保护模式中断机制尚未打开的尴尬局面，最终导致宕机。**

6，setup 程序的最后是 jmpi 0,8 为什么这个 8 不能简单的当作阿拉伯数字8看待？

> **底层源代码的一个特点就是数据的每一个二进制位都有自己明确的意义。** 如果把 8 当作一个数字来看，是很难解读的。如果把它看成是二进制的 1000，那么就可以了。1000 的右边 1、2 位 00 表示**特权级别**，00 代表内核态，11 代表用户态。右边数第 3 位的 0 代表所选择的表是 GDT，如果是 1 则代表 LDT。最左边的 1 表示所选择表的第几项。通过查看 setup 初始化的 GDT 可知，该命令指向的是内存首部。（GDT 项目号排序 0、1、2，那 2 是否需要两位来表示呢？果不其然，参见 28 页的上数第一段 10000）

7，打开 A20 和打开 PE 究竟是什么关系，保护模式不就是 32 位的吗？为什么还要打开 A20？有必要吗？

> 打开 A20 之后，将可以使用 32 位内存寻址模式，将 CR0 的 PE 位设置为 1，可以使得 CPU 工作在保护模式下，即采用 GDT 进行寻址。两者对于打开保护模式缺一不可。

8，在 setup 程序里曾经设置过一次 gdt，为什么在 head 程序中将其废弃，又重新设置了一个？为什么折腾两次，而不是一次搞好？

9，Linux 是用 C 语言写的，为什么没有从 main 开始，而是先运行 3 个汇编程序，道理何在？

10，为什么不用 call，而是用 ret “调用” main 函数？画出调用路线图，给出代码证据。

> **下面两题参考 IA-32-3 中文版.pdf**

11，保护模式的“保护”体现在哪里？

12，特权级的目的和意义是什么？为什么特权级是基于段的？