操作系统思辨报告1

2019211315班　2019211243　谭恩浩

> 请说明计算机从系统加电，引导扇区执行，内核加载并转到内核入口开始执行的过程，请比较 hariboteOS，XV6，Ucore三种自制系统的过程并体会它们的异同

计算机从系统加电，引导扇区执行，内核加载并转到内核入口开始执行的过程

这里我们讨论的是16位，1M的8086芯片。

系统加电

计算机加电后，CPU硬件的逻辑设计使得加电的时候代码段寄存器`CS`内的值被强制设置为`0xFFFF`，指令指针寄存器`IP`内的值被设置为`0x0000`，`CS:IP`即`CS*16+IP`指向地址`0xFFFF0`，这是BIOS程序在ROM上的入口地址。BIOS是在计算机出厂的时候直接烧录在ROM里面的。当计算机执行了`0xFFFF0`处的第一条指令时，就开始执行BIOS程序了，接下来就由BIOS控制计算机。

🤗 BIOS全程为“Basic Input Output System”，即基本输入输出系统。BIOS是计算机加电后启动的第一个程序。其实，它是一组固化到计算机内主板上一个ROM芯片上的程序，它保存着计算机最重要的基本输入输出的程序、开机后自检程序和系统自启动程序，它可从CMOS中读写系统设置的具体信息。 其主要功能是为计算机提供最底层的、最直接的硬件设置和控制。此外，BIOS还向作业系统提供一些系统参数。系统硬件的变化是由BIOS隐藏，程序使用BIOS功能而不是直接控制硬件。现代作业系统会忽略BIOS提供的抽象层并直接控制硬件组件。

🤗 关于为什么BIOS的地址不是从`0x0`开始，而是从`0xFFFF0`开始，这是因为，不同的计算机厂商生产的计算机所带的外设不一样，因此，这段程序大小也限机型的不同而不一样。若A厂生产的BIOS大小为1K，B厂生产的BIOS大小为2K，且都将其放在`0x0`处，那么用户写的程序就可以从`0x400`处开始，也可以从`0x800`处开始，非常不统一。但若将BIOS放在1M内存的顶部，那么用户写的程序就都可以从`0x0`处开始了。但若将BIOS放在1M内存的顶部，如果其大小为1K，那么应当`0xFFC00`处开始放，但如果大小为2K，那么应当从`0xFF800`开始放。这回CPU不清楚是应当从`0xFFC00`开始执行还是应当从`0xFF800`开始执行了。为了解决这个问题，8086规定，CPU均从`0xFFFF0`处开始执行，而在`0xFFFF0`处，放一条无条件转移指令`JMP`。如果A厂的BIOS是从`0xFFC00`开始放的，那么这条转移指令就跳转到`0xFFC00`处开始执行。如果B厂的BIOS是从`0xFF800`开始放的，那么这条转移指令就跳转到`0xFF800`处开始执行，各个厂 家可以跟据自己所生产的BIOS程序的大小，来决定此转移指令具体跳转到的位置。

BIOS

BIOS的工作相当简单，主要的工作就是执行主机自检（POST），然后将磁盘上的操作系统载入内存的指定位置。首先BIOS对CPU发送`int 0x19`中断，使CPU运行`int 0x19`中断对应的中断服务程序，这个中断服务程序的作用就是把软盘第一个扇区的程序加载到内存的指定位置。BIOS查找磁盘上的第一个扇区（0面0道1扇区），如果该扇区是启动扇区（扇区以`0x55AA`结尾），那么就将它载入`0x7C00`的内存地址，然后跳转到此地址上执行。如果不是，就继续检查下一个扇区，若此磁盘上没有启动扇区，则输出错误信息，结束执行。启动扇区只有512字节大小，是存不下操作系统的，因此它的主要就是储存操作

系统的启动程序，称为bootsect。当BIOS将启动扇区的内容加载到内存并跳转到程序开始的第一个地址后，控制权就转移给了bootsect，此刻BIOS的工作已经结束了。

> 🤗 为什么bootloder加载在`0x7C00`上呢？这是因为86系列操作系统是向前兼容的。当时，搭配的操作系统是86-DOS。这个操作系统需要的内存最少是32KB。我们知道，内存地址从`0x0`开始编号，32KB的内存就是`0x0000~0x7FFF`。8088芯片本身需要占用`0x0000~0x03FF`，用来保存各种中断处理程序的储存位置。（主引导记录本身就是中断信号`int 0x19`的处理程序。）所以，内存只剩下`0x0400~0x7FFF`可以使用。**为了把尽量多的连续内存留给操作系统，主引导记录就被放到了内存地址的尾部。**由于一个扇区是512字节，主引导记录本身也会产生数据，需要另外留出512字节保存。所以，它的预留位置就变成了：`0x7FFF - 512 - 512 + 1 = 0x7C00`

加载内核

现在由操作系统开始全权控制计算机，开始根据自身的需求安排内存。bootsect首先对内存进行规划：包括之后将要加载的程序的扇区数，被加载到的位置，启动扇区被BIOS加载的位置等等，方便之后使用。之后根据操作系统的设计，将磁盘上其他分区的操作系统的完整内容采用某些方法载入到内存中，然后安排一条指令跳转到内核入口，用户就可以使用计算机了。

比较 hariboteOS，XV6，Ucore三种自制系统的过程并体会它们的异同

从计算机加电到BIOS扫描扇区中间的操作都是一样的，这里主要比较三种操作系统如何在启动扇区的程序被加载进内存后控制计算机的。

haribote

hariboteOS比较简单，启动区的程序单纯地通过朴素代码设计为把磁盘中剩下扇区的内容全部读入内存。但额外加了一个试错，这是因为hariboteOS是从软盘上读取操作系统，软盘非常不可靠，有时会发生不能读数据的情况，这时就需要重新读一遍，为了避免软盘真的损坏了而操作系统陷入无限重新读软盘的死循环，程序设计为只重读5次，超过5次就放弃，认为此软盘坏掉了。hariboteOS的启动程序被在放在内存地址`0x8000`处，之后读入的软盘数据依序往后储存，这是作者随便选的地址。

为了方便之后的开发，需要把CPU从16位模式转变到32位模式（16位模式下和32位模式下机器语言的命令代码不一样）。另外CPU的自我保护功能（识别出可疑的机器语言并进行屏蔽，以免破坏系统）只在32位下才能使用。注意，如果使用32位模式就不能调用BIOS功能了，这是因为BIOS是用16位机器语言写的。因此作者把需要用到BIOS的事情都放在了转换成32位模式之前。

Ucore

Ucore增加了从实模式到保护模式的转换以及分段储存管理机制。Ucore中从启动扇区载入的程序称为bootloader，bootloader完成的工作包括：

- 切换到保护模式，启用分段机制
- 读磁盘中ELF执行文件格式的Ucore操作系统到内存
- 显示字符串信息
- 把控制权交给Ucore操作系统

下面先简要解释一下新出现的概念：

1. **实模式**下，软件可访问的物理内存空间不能超过1MB，实模式下程序代码和数据位于不同区域，操作系统和用户程序并没有区别对待，而且每一个指针都是指向实际的物理地址。
2. **保护模式**下，80386的全部32根地址线有效，可寻址高达4G字节的线性地址空间和物理地址空间，可访问64TB (有2^14个段，每个段最大空间为2^32字节)的逻辑地址空间，可采用分段存储管理机制和分页存储管理机制。
3. **分段机制**将内存划分成以起始地址和长度限制这两个二维参数表示的内存块，这些内存块就称之为段(Segment)。分段机制涉及5个关键内容:逻辑地址、物理地址、段描述符表、段描述符、段选择子。
4. 保护模式下，系统拥有4个**特权级**。有3种主要的资源受到保护:内存，I/O端口以及执行特殊机器指令的能力。在Ucore中，CPU只用到其中的2个特权级:0(内核态)和3(用户态)。
5. **ELF文件格式**是Linux系统下的一种常用目标文件(object file)格式，有三种主要类型:用于执行的可执行文件，用于连接的可重定位文件和共享目标文件

磁盘中Ucore操作系统是以ELF文件格式存储的，其中，ELF header（`elfhdr`）在文件开始处描述了整个文件的组织。ELF的文件头包含整个执行文件的控制结构，其定义在elf.h中；program header（`proghdr`）描述与程序执行直接相关的目标文件结构信息，用来在

文件中定位各个段的映像，同时包含其他一些用来为程序创建进程映像所必需的信息。根据`elfhdr`和`proghdr`的结构描述，bootloader就可以完成对ELF格式的ucore操作系统的加载过程。

bootloader让CPU进入保护模式后，下一步的工作就是从硬盘上加载并运行OS。考虑到实现的简单性，bootloader的访问硬盘都是LBA模式的PIO(Program IO)方式，即所有的IO操作是通过CPU访问硬盘的IO地址寄存器完成。当bootloader通过读取硬盘扇区把Ucore在系统加载到内存后，就转跳到Ucore操作系统在内存中的入口位置(`kern/init.c`中的`kern_init`函数的起始地址)，这样Ucore就接管了整个控制权。

XV6

XV6系统的加载步骤如下：

1. 首先，XV6的启动程序为`bootasm.S`。在`bootasm.S`中，系统必须初始化CPU的运行状态。具体地说，需要将x86 CPU从启动时默认的Intel 8088 16位实模式切换到80386之后的32位保护模式；然后设置初始的GDT，将虚拟地址直接按值映射到物理地址；最后，调用`bootmain.c`中的`bootmain()`函数。
2. `bootmain()`函数的主要任务是将内核的ELF文件从硬盘中加载进内存，并将控制权转交给内核程序。具体地说，此函数首先将ELF文件的前4096个字节（也就是第一个内存页）从磁盘里加载进来，然后根据ELF文件头里记录的文件大小和不同的程序头信息，将完整的ELF文件加载到内存中。然后根据ELF文件里记录的入口点，将控制权转交给XV6系统。
3. `entry.S`的主要任务是设置页表，让分页硬件能够正常运行，然后跳转到`main.c`的`main()`函数处，开始整个操作系统的运行。
4. `main()`函数首先初始化了与内存管理、进程管理、中断控制、文件管理相关的各种模块，然后启动第一个叫做`initcode`的用户进程。至此，整个XV6系统启动完毕。