**操作系统的引导**

目录

1. [实验目的](https://cms.hit.edu.cn/mod/wiki/view.php?pageid=15#toc-1)[[编辑]](https://cms.hit.edu.cn/mod/wiki/edit.php?pageid=15&section=%E5%AE%9E%E9%AA%8C%E7%9B%AE%E7%9A%84)

2. [实验内容](https://cms.hit.edu.cn/mod/wiki/view.php?pageid=15#toc-2)[[编辑]](https://cms.hit.edu.cn/mod/wiki/edit.php?pageid=15&section=%E5%AE%9E%E9%AA%8C%E5%86%85%E5%AE%B9)

3. [实验报告](https://cms.hit.edu.cn/mod/wiki/view.php?pageid=15#toc-3)[[编辑]](https://cms.hit.edu.cn/mod/wiki/edit.php?pageid=15&section=%E5%AE%9E%E9%AA%8C%E6%8A%A5%E5%91%8A)

4. [评分标准](https://cms.hit.edu.cn/mod/wiki/view.php?pageid=15#toc-4)[[编辑]](https://cms.hit.edu.cn/mod/wiki/edit.php?pageid=15&section=%E8%AF%84%E5%88%86%E6%A0%87%E5%87%86)

5. [实验提示](https://cms.hit.edu.cn/mod/wiki/view.php?pageid=15#toc-5)[[编辑]](https://cms.hit.edu.cn/mod/wiki/edit.php?pageid=15&section=%E5%AE%9E%E9%AA%8C%E6%8F%90%E7%A4%BA)

**实验目的[[编辑]](https://cms.hit.edu.cn/mod/wiki/edit.php?pageid=15&section=%E5%AE%9E%E9%AA%8C%E7%9B%AE%E7%9A%84)**

* 熟悉hit-oslab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。

**实验内容[[编辑]](https://cms.hit.edu.cn/mod/wiki/edit.php?pageid=15&section=%E5%AE%9E%E9%AA%8C%E5%86%85%E5%AE%B9)**

此次实验的基本内容是：

1. 阅读《Linux内核完全注释》的第6章，对计算机和Linux 0.11的引导过程进行初步的了解；
2. 按照下面的要求改写0.11的引导程序bootsect.s
3. 有兴趣同学可以做做进入保护模式前的设置程序setup.s。

改写bootsect.s主要完成如下功能：

1. bootsect.s能在屏幕上打印一段提示信息“XXX is booting...”，其中XXX是你给自己的操作系统起的名字，例如LZJos、Sunix等（可以上论坛上秀秀谁的OS名字最帅，也可以显示一个特色logo，以表示自己操作系统的与众不同。）

改写setup.s主要完成如下功能：

1. bootsect.s能完成setup.s的载入，并跳转到setup.s开始地址执行。而setup.s向屏幕输出一行"Now we are in SETUP"。
2. setup.s能获取至少一个基本的硬件参数（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。
3. setup.s不再加载Linux内核，保持上述信息显示在屏幕上即可。

**实验报告**[**[编辑]**](https://cms.hit.edu.cn/mod/wiki/edit.php?pageid=15&section=%E5%AE%9E%E9%AA%8C%E6%8A%A5%E5%91%8A)

在实验报告中回答如下问题：

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。

**评分标准**[**[编辑]**](https://cms.hit.edu.cn/mod/wiki/edit.php?pageid=15&section=%E8%AF%84%E5%88%86%E6%A0%87%E5%87%86)

* bootsect显示正确，30%
* bootsect正确读入setup，10%
* setup获取硬件参数正确，20%
* setup正确显示硬件参数，20%
* 实验报告，20%

**实验提示[[编辑]](https://cms.hit.edu.cn/mod/wiki/edit.php?pageid=15&section=%E5%AE%9E%E9%AA%8C%E6%8F%90%E7%A4%BA)**

操作系统的boot代码有很多，并且大部分是相似的。本实验仿照Linux-0.11/boot目录下的bootsect.s和setup.s，以剪裁它们为主线。当然，如果能完全从头编写，并实现实验所要求的功能，是再好不过了。

同济大学赵炯博士的《Linux内核0.11完全注释（修正版V3.0）》（以后简称《注释》）的第6章是非常有帮助的参考，实验中可能遇到的各种问题，几乎都能找到答案。可以在“[资料和文件](https://cms.hit.edu.cn/mod/folder/view.php?id=1716)下载”中下载到该书的电子版。同目录中，校友谢煜波撰写的《操作系统引导探究》也是一份很好的参考。

需要注意的是，oslab中的汇编代码使用as86编译，语法和汇编课上所授稍有不同。

**下面将给出一些更具体的“提示”。这些提示并不是实验的一步一步的指导，而是罗列了一些实验中可能遇到的困难，并给予相关提示。它们肯定不会涵盖所有问题，也不保证其中的每个字都对完成实验有帮助。所以，它们更适合在你遇到问题时查阅，而不是当作指南一样地亦步亦趋。本书所有实验的提示都是秉承这个思想编写的。**

**Linux 0.11相关代码详解**

boot/bootsect.s、boot/setup.s和tools/build.c是本实验会涉及到的源文件。它们的功能详见《注释》的6.2、6.3节和16章。

如果使用Windows下的环境，那么要注意Windows环境里提供的build.c是一个经过修改过的版本。Linus Torvalds的原版是将0.11内核的最终目标代码输出到标准输出，由make程序将数据重定向到Image文件，这在Linux、Unix和Minix等系统下都是非常有效的。但Windows本身的缺陷（也许是特色）决定了在Windows下不能这么做，所以flyfish修改了build.c，将输出直接写入到Image（flyfish是写入到Boot.img文件，我们为了两个环境的一致，也为了最大化地与原始版本保持统一，将其改为Image）文件中。同时为了适应Windows的一些特殊情况，他还做了其它一些小修改。

**引导程序的运行环境**

引导程序由BIOS加载并运行。它活动时，操作系统还不存在，整台计算机的所有资源都由它掌控，而能利用的功能只有[BIOS中断调用](https://cms.hit.edu.cn/mod/wiki/view.php?pageid=16)。

**完成bootsect.s的屏幕输出功能**

首先来看完成屏幕显示的关键代码如下：

! 首先读入光标位置  
 mov ah,#0x03   
 xor bh,bh  
 int 0x10  
  
 ! 显示字符串“LZJos is running...”  
 mov cx,#25 ! 要显示的字符串长度  
 mov bx,#0x0007 ! page 0, attribute 7 (normal)  
 mov bp,#msg1  
 mov ax,#0x1301 ! write string, move cursor  
 int 0x10  
  
inf\_loop:  
 jmp inf\_loop ! 后面都不是正经代码了，得往回跳呀  
 ! msg1处放置字符串  
  
msg1:  
 .byte 13,10 ! 换行+回车  
 .ascii "LZJos is running..."  
 .byte 13,10,13,10 ! 两对换行+回车  
 !设置引导扇区标记0xAA55  
 .org 510  
boot\_flag:  
 .word 0xAA55 ! 必须有它，才能引导

接下来，将完成屏幕显示的代码在开发环境中编译，并使用linux-0.11/tools/build.c将编译后的目标文件做成Image文件。

**编译和运行**

Ubuntu上先从终端进入~/oslab/linux-0.11/boot/目录。Windows上则先双击快捷方式“MinGW32.bat”，将打开一个命令行窗口，当前目录是oslab，用cd命令进入linux-0.11\boot。无论那种系统，都执行下面两个命令编译和链接bootsect.s：

as86 -0 -a -o bootsect.o bootsect.s  
ld86 -0 -s -o bootsect bootsect.o

其中-0（**注意**：这是数字0，不是字母O）表示生成8086的16位目标程序，-a表示生成与GNU as和ld部分兼容的代码，-s告诉链接器ld86去除最后生成的可执行文件中的符号信息。

如果这两个命令没有任何输出，说明编译与链接都通过了。Ubuntu下用ls -l可列出下面的信息：

-rw--x--x 1 root root 544 Jul 25 15:07 bootsect  
-rw------ 1 root root 257 Jul 25 15:07 bootsect.o  
-rw------ 1 root root 686 Jul 25 14:28 bootsect.s

Windows下用dir可列出下面的信息：

2008-07-28 20:14 544 bootsect  
2008-07-28 20:14 924 bootsect.o  
2008-07-26 20:13 5,059 bootsect.s

其中bootsect.o是中间文件。bootsect是编译、链接后的目标文件。

需要留意的文件是bootsect的文件大小是544字节，而引导程序必须要正好占用一个磁盘扇区，即512个字节。造成多了32个字节的原因是ld86产生的是Minix可执行文件格式，这样的可执行文件处理文本段、数据段等部分以外，还包括一个Minix可执行文件头部，它的结构如下：

struct exec {  
 unsigned char a\_magic[2]; //执行文件魔数  
 unsigned char a\_flags;  
 unsigned char a\_cpu; //CPU标识号  
 unsigned char a\_hdrlen; //头部长度，32字节或48字节  
 unsigned char a\_unused;  
 unsigned short a\_version;  
 long a\_text; long a\_data; long a\_bss; //代码段长度、数据段长度、堆长度  
 long a\_entry; //执行入口地址  
 long a\_total; //分配的内存总量  
 long a\_syms; //符号表大小   
};

算一算：6 char(6字节)+1 short(2字节)+6 long(24字节)=32，正好是32个字节，去掉这32个字节后就可以放入引导扇区了（这是tools/build.c的用途之一）。

对于上面的Minix可执行文件，其a\_magic[0]=0x01，a\_magic[1]=0x03，a\_flags=0x10（可执行文件），a\_cpu=0x04（表示Intel i8086/8088，如果是0x17则表示Sun公司的SPARC），所以bootsect文件的头几个字节应该是01 03 10 04。为了验证一下，Ubuntu下用命令“hexdump -C bootsect”可以看到：

00000000 01 03 10 04 20 00 00 00 00 02 00 00 00 00 00 00 |.... ...........|  
00000010 00 00 00 00 00 00 00 00 00 82 00 00 00 00 00 00 |................|  
00000020 b8 c0 07 8e d8 8e c0 b4 03 30 ff cd 10 b9 17 00 |.........0......|  
00000030 bb 07 00 bd 3f 00 b8 01 13 cd 10 b8 00 90 8e c0 |....?...........|  
00000040 ba 00 00 b9 02 00 bb 00 02 b8 04 02 cd 13 73 0a |..............s.|  
00000050 ba 00 00 b8 00 00 cd 13 eb e1 ea 00 00 20 90 0d |............. ..|  
00000060 0a 53 75 6e 69 78 20 69 73 20 72 75 6e 6e 69 6e |.Sunix is runnin|  
00000070 67 21 0d 0a 0d 0a 00 00 00 00 00 00 00 00 00 00 |g!..............|  
00000080 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 |................|  
\*  
00000210 00 00 00 00 00 00 00 00 00 00 00 00 00 00 55 aa |..............U.|  
00000220

Windows下用UltraEdit把该文件打开，果然如此。

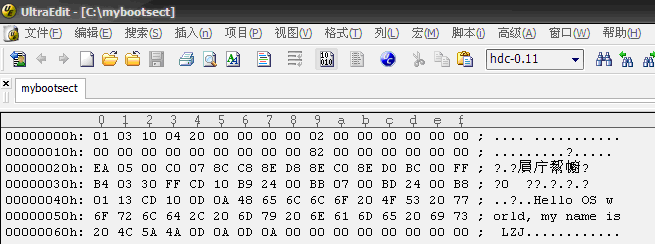


图1 用UltraEdit打开文件bootsect

 接下来干什么呢？是的，要去掉这32个字节的文件头部（tools/build.c的功能之一就是这个）！随手编个小的文件读写程序都可以去掉它。不过，懒且聪明的人会在Ubuntu下用命令：

$ dd bs=1 if=bootsect of=Image skip=32

生成的Image就是去掉文件头的bootsect。

Windows下可以用UltraEdit直接删除（选中这32个字节，然后按Ctrl+X）。

去掉这32个字节后，将生成的文件拷贝到linux-0.11目录下，并一定要命名为“Image”（注意大小写）。然后就“run”吧！

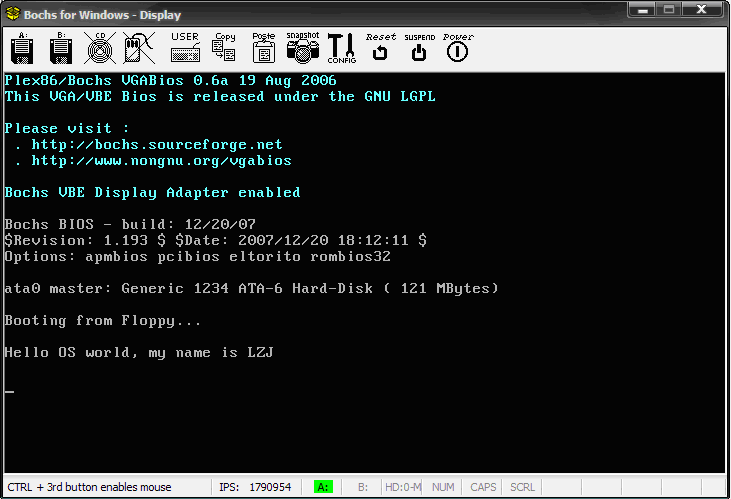


图2 bootsect引导后的系统启动情况

**bootsect.s读入setup.s**

首先编写一个setup.s，该setup.s可以就直接拷贝前面的bootsect.s（可能还需要简单的调整），然后将其中的显示的信息改为：“Now we are in SETUP”。

接下来需要编写bootsect.s中载入setup.s的关键代码。原版bootsect.s中下面的代码就是做这个的。

load\_setup:  
 mov dx,#0x0000 !设置驱动器和磁头(drive 0, head 0): 软盘0磁头  
 mov cx,#0x0002 !设置扇区号和磁道(sector 2, track 0):0磁头、0磁道、2扇区  
 mov bx,#0x0200 !设置读入的内存地址：BOOTSEG+address = 512，偏移512字节  
 mov ax,#0x0200+SETUPLEN !设置读入的扇区个数(service 2, nr of sectors)，  
 !SETUPLEN是读入的扇区个数，Linux 0.11设置的是4，  
 !我们不需要那么多，我们设置为2  
 int 0x13 !应用0x13号BIOS中断读入2个setup.s扇区  
 jnc ok\_load\_setup !读入成功，跳转到ok\_load\_setup: ok - continue  
 mov dx,#0x0000 !软驱、软盘有问题才会执行到这里。我们的镜像文件比它们可靠多了  
 mov ax,#0x0000 !否则复位软驱 reset the diskette  
 int 0x13  
 jmp load\_setup !重新循环，再次尝试读取  
ok\_load\_setup:  
 ！接下来要干什么？当然是跳到setup执行。

所有需要的功能在原版bootsect.s中都是存在的，我们要做的仅仅是删除那些对我们无用的代码。

**再次编译**

现在有两个文件都要编译、链接。一个个手工编译，效率低下，所以借助Makefile是最佳方式。

在Ubuntu下，进入linux-0.11目录后，使用下面命令（注意大小写）：

$ make BootImage

Windows下，在命令行方式，进入Linux-0.11目录后，使用同样的命令（不需注意大小写）：

make BootImage

无论哪种系统，都会看到：

Unable to open 'system'  
make: \*\*\* [BootImage] Error 1

有Error！这是因为make根据Makefile的指引执行了tools/build.c，它是为生成整个内核的镜像文件而设计的，没考虑我们只需要bootsect.s和setup.s的情况。它在向我们要“系统”的核心代码。为完成实验，接下来给它打个小补丁。

**修改build.c**

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。其中system是第三个参数（argv[3]）。当“make all”或者“makeall”的时候，这个参数传过来的是正确的文件名，build.c会打开它，将内容写入Image。而“make BootImage”时，传过来的是字符串"none"。所以，改造build.c的思路就是当argv[3]是"none"的时候，只写bootsect和setup，忽略所有与system有关的工作，或者在该写system的位置都写上“0”。

修改工作主要集中在build.c的尾部，请斟酌。

当按照前一节所讲的编译方法编译成功后，run，就得到了如图3所示的运行结果，和我们想得到的结果完全一样。

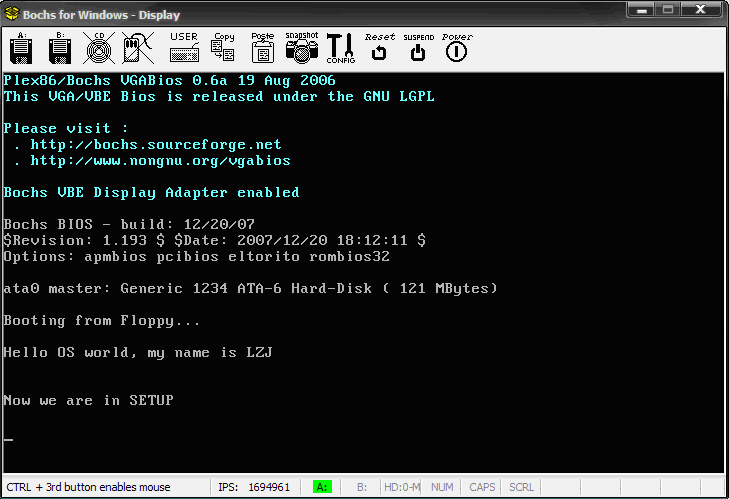


图3 用修改后的bootsect.s和setup.s进行引导的结果

**setup.s获取基本硬件参数**

setup.s将获得硬件参数放在内存的0x90000处。原版setup.s中已经完成了光标位置、内存大小、显存大小、显卡参数、第一和第二硬盘参数的保存。

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。有些硬件参数的获取要稍微复杂一些，如磁盘参数表。在PC机中BIOS设定的中断向量表中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。第二个硬盘的基本参数表入口地址存于int 0x46中断向量位置处。每个硬盘参数表有16个字节大小。下表给出了硬盘基本参数表的内容：

表1 磁盘基本参数表

|  |  |  |
| --- | --- | --- |
| 位移 | 大小 | 说明 |
| 0x00 | 字 | 柱面数 |
| 0x02 | 字节 | 磁头数 |
| … | … | … |
| 0x0E | 字节 | 每磁道扇区数 |
| 0x0F | 字节 | 保留 |

所以获得磁盘参数的方法就是复制数据。

下面是将硬件参数取出来放在内存0x90000的关键代码。

mov ax,#INITSEG   
mov ds,ax !设置ds=0x9000  
mov ah,#0x03 !读入光标位置  
xor bh,bh  
int 0x10 !调用0x10中断  
mov [0],dx !将光标位置写入0x90000.  
  
!读入内存大小位置  
mov ah,#0x88  
int 0x15  
mov [2],ax  
  
!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

现在已经将硬件参数（只包括光标位置、内存大小和硬盘参数，其他硬件参数取出的方法基本相同，此处略去）取出来放在了0x90000处，接下来的工作是将这些参数显示在屏幕上。这些参数都是一些无符号整数，所以需要做的主要工作是用汇编程序在屏幕上将这些整数显示出来。

以十六进制方式显示比较简单。这是因为十六进制与二进制有很好的对应关系（每4位二进制数和1位十六进制数存在一一对应关系），显示时只需将原二进制数每4位划成一组，按组求对应的ASCII码送显示器即可。ASCII码与十六进制数字的对应关系为：0x30～0x39对应数字0～9，0x41～0x46对应数字a～f。从数字9到a，其ASCII码间隔了7h，这一点在转换时要特别注意。为使一个十六进制数能按高位到低位依次显示，实际编程中，需对bx中的数每次循环左移一组（4位二进制），然后屏蔽掉当前高12位，对当前余下的4位（即1位十六进制数）求其ASCII码，要判断它是0～9还是a～f，是前者则加0x30得对应的ASCII码，后者则要加0x37才行，最后送显示器输出。以上步骤重复4次，就可以完成bx中数以4位十六进制的形式显示出来。

下面是完成显示16进制数的汇编语言程序的关键代码，其中用到的BIOS中断为INT 0x10，功能号0x0E（显示一个字符），即AH=0x0E，AL=要显示字符的ASCII码。

!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  
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

这里用到了一个loop指令，每次执行loop指令，cx减1，然后判断cx是否等于0。如果不为0则转移到loop指令后的标号处，实现循环；如果为0顺序执行。另外还有一个非常相似的指令：rep指令，每次执行rep指令，cx减1，然后判断cx是否等于0，如果不为0则继续执行rep指令后的串操作指令，直到cx为0，实现重复。

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

只要在适当的位置调用print\_bx和print\_nl（注意，一定要设置好栈，才能进行函数调用）就能将获得硬件参数打印到屏幕上，完成此次实验的任务。但事情往往并不总是顺利的，前面的两个实验大多数实验者可能一次就编译调试通过了（这里要提醒大家：编写操作系统的代码一定要认真，因为要调试操作系统并不是一件很方便的事）。但在这个实验中会出现运行结果不对的情况（为什么呢？因为我们给的代码并不是100%好用的）。所以接下来要复习一下汇编，并阅读《[Bochs使用手册](https://cms.hit.edu.cn/mod/wiki/view.php?pageid=22)》，学学在Bochs中如何调试操作系统代码。

我想经过漫长而痛苦的调试后，大家一定能兴奋地得到下面的运行结果：

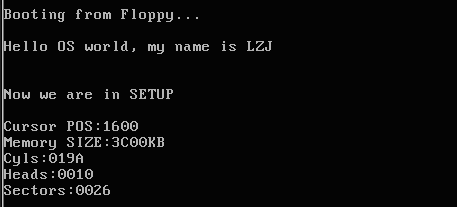


图4 用可以打印硬件参数的setup.s进行引导的结果

 Memory Size是0x3C00KB，算一算刚好是15MB（扩展内存），加上1MB正好是16MB，看看Bochs配置文件bochs/bochsrc.bxrc：

……  
megs: 16  
……  
ata0-master: type=disk, mode=flat, cylinders=410, heads=16, spt=38  
……

这些都和上面打出的参数吻合，表示此次实验是成功的。