编写 MBR 从硬盘启动 linux0.11

作者: dswei@ustc.edu **日期:** 2005.4.3

计算机启动时 BIOS 会把启动盘第一个扇区的数据读入内存 0x7C00 开始处，然后跳到这里继续执行。从硬盘启动和从软盘启动唯一的区别就是映象文件存储方式的不同：

1. 对于从软盘启动的方式，映象文件连续地存放在软盘开始的位置处。放在第一个扇区的 bootsect.s 被 BIOS 读入内存后，就会把余下的映象文件读入内存，然后继续执行
2. 对于从硬盘启动的方式，映象文件存放在 Minix 格式的硬盘分区里，MBR(硬盘第一个扇区)中的程序需要根据硬盘的参数和 Minix 文件系统的存储格式读出它

本文描述了如何编写 MBR 中的程序，把存放在硬盘第一个分区根目录下的 linux0.11 映像文件 Image 读入内存。

文章中小字体的括号里的数字表示参考文献的编号(见后面的参考文献列表)。

本文分 5 个部分：
1、硬盘简介
2、Minix 1.0 文件系统简介
3、程序流程
4、上机实验
5、程序代码(以附件形式给出)

一．硬盘简介(1)

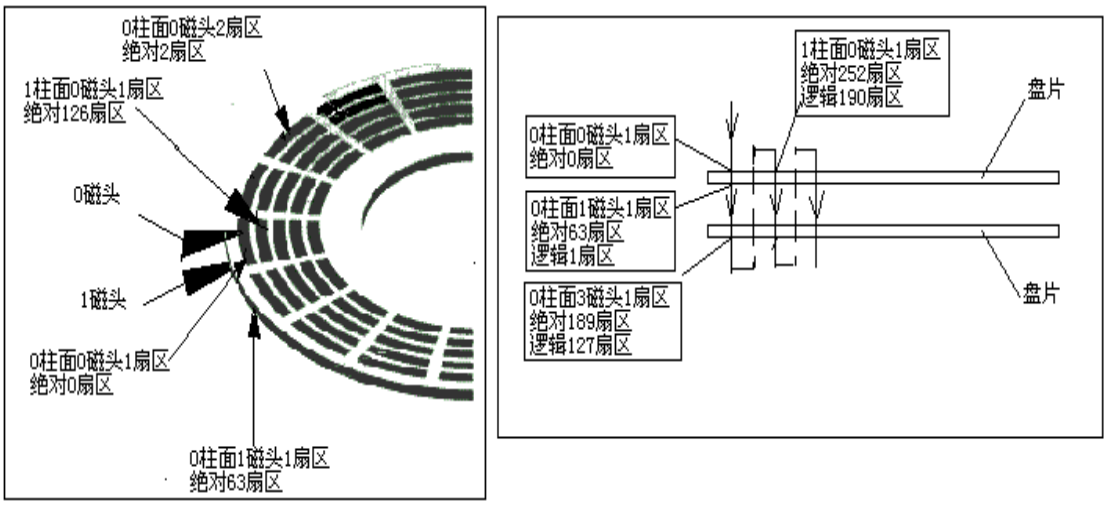

图 1．硬盘扇区编号(此图来源文献 1)

我们需要了解的是物理扇区编号方式和绝对扇区编号方式。物理扇区号直接按柱面、磁头、扇区 3 者的组合来定位某个扇区。对于硬盘的第一个扇区，其编号为“0 柱面 0 磁头 1 扇区”。我们假设硬盘磁头数为 16，每磁道扇区数为 63，下面描述遍历整个硬盘时柱面号、磁头号、扇区号的变化规律(以(x,y,z)表示 x 柱面 y 磁头 z 扇区)： (0,0,1)、(0,0,2)、……、(0,0,63)、

(0,1,1)、(0,1,2)、……、(0,1,63)、
(0,2,1)、(0,2,2)、……、(0,2,63)、
……
(0,15,1)、(0,15,2)、……、(0,15,63)、
(1,0,1)、(1,0,2)、……、(1,0,63)、
(1,1,1)、(1,1,2)、……、(1,1,63)、

换句话说就是扇区号是从 1 到 63 的 63 进制，磁头号是 0 到 15 的 16 进制，“百位”是柱面号，“十位”是磁头号，“个位”是扇区号。

绝对扇区号从 0 开始，遍历硬盘时依次增 1。

两者的换算关系如下(abs_sector 表示绝对扇区号，cyl 表示柱面号，head 表示磁头号，sector 表示扇区号，nheads 表示磁头数，nspt 表示每磁道扇区数)：

```
sector = abs_sector % nspt + 1
track  = abs_sector / nspt
head   = track % nheads
cyl    = track / nheads
```

如何知道上面说的磁头数 nheads、每磁道扇区数 nspt 呢？启动时 BIOS 会把硬盘参数表放在内存某个位置。对于第一个硬盘，硬盘参数表的首地址放在中断 0x41 处，即内存地址 4*0x41=0x104 开始的 4 个字节表示硬盘参数表的段地址(后面 2 字节)和偏移地址(前面 2 字节)。硬盘参数表的结构如下：

```
dw  cylinders
db  nheads
dw  0
dw  write pre-comp
db  0
db  0 "control byte"
db  0, 0, 0
dw  landing zone
db  nspt(sectors/track)
db  0
```

第二个硬盘的参数表地址放在 BIOS 中断向量 0x46 处。

对于硬盘，我们还需要稍微知道一点分区表的信息。分区表放在硬盘第一个扇区的 0x1be～0x1fd 处(共 64 字节，第一个分区的信息在 0x1be～0x1ce)。我们需要使用的仅仅是存放在 0x1c6 处的“分区起始绝对扇区号”(对应于程序中的 start_sect 变量)

二. Minix 1.0 文件系统简介(2) (3)

关于 Minix 1.0 文件系统的更详细知识请参考文献 2、3，下面仅介绍理解本程序需要了解的知识。

Minix 1.0 格式的分区结构如下：

引导块	超级块	i 点位图块…	逻辑块位图块…	i 节点块…	数据区……

1、对于 Minix 1.0，每个盘块占 1k 字节(即 2 个扇区)。引导块的第一个扇区就是本分区的第一个扇区，扇区号存在 MBR 中 0x1c6 处(就是上文的“分区起始绝对扇区号”)。盘块从 0 开始编号，对于盘块 n，其绝对扇区号 = 2*n +分区起始绝对扇区号。

2、i 点位图块和逻辑块位图块的大小、数据区的起始盘块号等信息存放在超级块中，超级块的数据结构如下：

```
struct d_super_block{
    unsigned short s_ninodes;       //节点数
    unsigned short s_nzones;        //逻辑块数
    unsigned short s_imap_blocks;   //i 节点位图所占用的数据块数
    unsigned short s_zmap_blocks;   //逻辑块位图所占用的数据块数
    unsigned short s_firstdatazone; //第一个数据逻辑块
    unsigned short s_log_zone_size; //log2(数据块数/逻辑块)
    unsigned long  s_max_size;      //文件最大长度
    unsigned short s_magic;}；      //文件系统魔数
```

我们只需要知道开始存放 i 节点的盘块号 inode_start_zone 和第一个数据逻辑块号 firstdatazone：

```
    inode_start_zone = 2 + s_imap_blocks + s_zmap_blocks
    firstdatazone    = s_firstdatazone
```

3、每个 i 节点占 32 字节，其数据结构如下：

```
    struct d_inode{
        unsigned short i_mode;      //文件类型和属性
        unsigned short i_uid;       //用户 id
        unsigned long  i_size;      //文件大小(字节数)
        unsigned long  i_time;      //修改时间(1970.1.1:0 算起，秒)
        unsigned char  i_gid;       //组 id
        unsigned char  i_nlinks;    //链接数
        unsigned short i_zone[9];}; //直接(0～6)、间接(7)、双重间接
                                    //(8)逻辑块号
```

根据其中的 i_size 可以计算出文件占用的盘块数 = (i_size + 1023) / 1024

文件数据的前面 7k 存放在称为直接块的 7 个盘块中，这 7 个盘块的盘块号存在 i_zone[0]～i_zone[6]中。如果文件大于 7k，则需要使用到一次间接块(i_zone[7])，它对应的盘块里存放的不是文件数据，而是 7k 之后的文件数据存放的盘块号。比如一次间接块的第 1、2 字节指出第 8k 文件数据存放的盘块号码。一次间接块可以存放 512 个盘块号。如果文件大于(512+7)k，则需要使用二次间接块 i_zone[8]。图示如下：

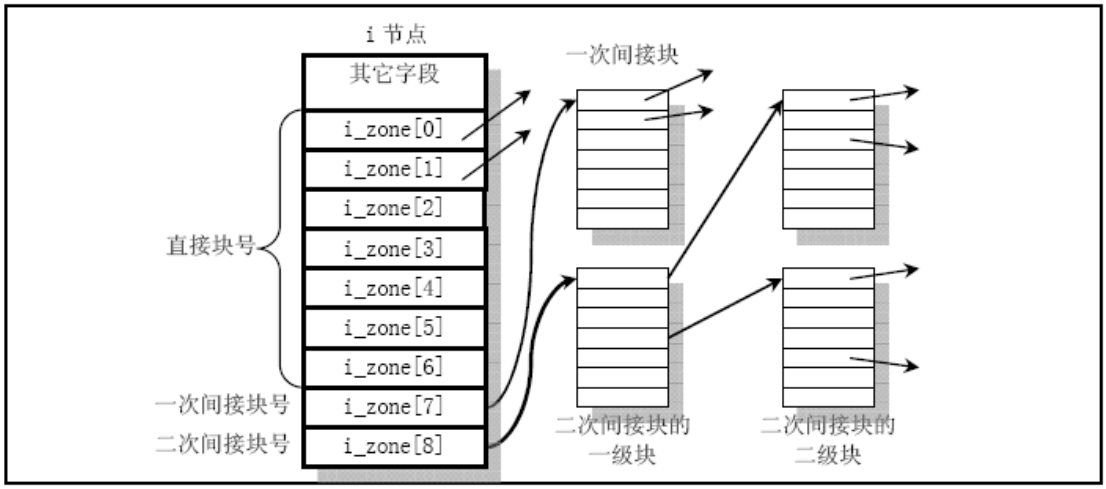

图 2．文件数据存储结构(此图来源文献 3)

根据 i_zone[0]～i_zone[6]，可以直接读出前面 7k 数据；然后读出一次间接块 i_zone[7]，根据其中的盘块号即可读出其余的数据；忽略 i_zone[8]，因为 linux0.11 映像文件没那么大。

图 3 是映像文件 Image 一次间接块的前面部分数据。每两个直接表示一个盘块号。“ 00 00 ”表示这 1k 的数据都是 0(比如红圈里的)，蓝圈里的“ 70 05 ”表示这 1k 数据存放的盘块号是 0x0570。

```
0015be00h: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ; ................
0015be10h: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ; ................
0015be20h: 70 05 00 00 71 05 00 00 72 05 73 05 74 05 75 05 ; p...q...r.s.t.u.
0015be30h: 76 05 77 05 78 05 79 05 7A 05 7B 05 7C 05 7D 05 ; v.w.x.y.z.{.|.}.
0015be40h: 7E 05 7F 05 80 05 81 05 82 05 83 05 84 05 85 05 ; ~. .€.?????
0015be50h: 86 05 87 05 88 05 89 05 8A 05 8B 05 8C 05 8D 05 ; ????????
0015be60h: 8E 05 8F 05 90 05 91 05 92 05 93 05 94 05 95 05 ; ????????
0015be70h: 96 05 97 05 98 05 99 05 9A 05 9B 05 9C 05 9D 05 ; ????????
0015be80h: 9E 05 9F 05 A0 05 A1 05 A2 05 A3 05 A4 05 A5 05 ; ????????
0015be90h: A6 05 A7 05 A8 05 A9 05 AA 05 AB 05 AC 05 AD 05 ; ????????
0015bea0h: AE 05 AF 05 B0 05 B1 05 B2 05 B3 05 B4 05 B5 05 ; ????????
0015beb0h: B6 05 00 00 00 00 B7 05 B8 05 00 00 00 00 B9 05 ; ?....??....?
```

图 3．映像文件 Image 一次间接块的前面部分数据

4、第一个数据块存放的就是根目录文件的前 1k 数据，根目录文件存放的是称为目录项的信息。其数据结构如下：

```
struct dir_entry{
    unsigned short inode;    //i 节点
    char name[14];};         //文件名
```

图 4 就是根目录文件第一个盘块的部分信息，图中红圈里面的数据表示本本目录项的 i 节点，蓝圈里面的数据表示该目录形的名称是 Image。从图上我们可以看到，前面两个目录项分别是目录“ . ”和目录“ .. ”，紧随其后的是 bin、dev、etc、mnt、root、tmp、usr 等目录，再后面的是 hello.c、Image、image 文件(至于是目录还是文件，需要找出相应的 i 节点根据其 i_mode 项来判断)。

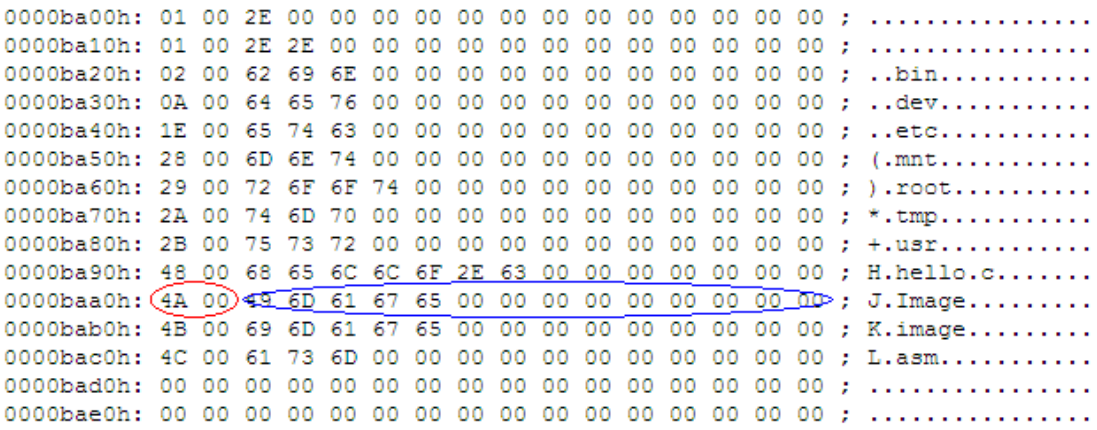

图 4．根目录文件第一个数据块部分信息

三．程序流程

现在可以描述查找映像文件并读出它的过程了：

a．获取硬盘参数——磁头数、每磁道扇区数(以便计算绝对扇区号对应的柱面号、磁头号、扇区号)，获取分区参数——第一个分区起始绝对扇区号(以定位 Minix 文件系统的逻辑块)

b．读出超级块，计算出 i 节点开始存放的盘块号 inode_start_zone 和第一个数据逻辑块号 firstdatazone

c．读出根目录文件(本程序只读出了它前面的 1k 数据——即第一个数据逻辑块)

d．根据映像文件名 Image 找到对应的目录项，取得其 i 节点编号(0x004A，图 3 红圈部分)

e．根据映像文件 i 节点编号和 inode_start_zone 读出 i 节点

f．根据 i 节点的 i_size 信息确定要读取的盘块树，根据 i_zone[0]～i_zone[6]读出前面 7k 数据

g．根据 i_zone[7]读出一次间接块的数据

h．根据一次间接块确定的盘块号，读出

linux0.11 映像文件由 4 部分组成：1、bootsect.s，2、setup.s，3、head.s，4、系统模块。从软盘启动时，bootsect.s 生成的可执行代码(512 字节)被读入 0x7C00 处，它执行时先把自己移到 0x90000 处，然后把 setup.s 生成的可执行代码(大小为 2k)从软盘读入到内存 0x90200 处，并把映像文件余下部分读到内存 0x10000 处。本 MBR 程序把映像文件前面 2.5k 数据(对应 512 字节的 bootsect.s 可执行代码和 2k 的 setup.s 可执行代码)读到 0x90000 处，把映像文件余下部分读到内存 0x10000 处，然后跳到 0x90200 执行 setup.s 代码。可见，本程序的功能只是与 bootsect.s 相似，bootsect.s 从软盘读出映像文件，而本程序从硬盘中读出。

程序代码见第 5 部分的 bootload.s，图 5 是流程图。

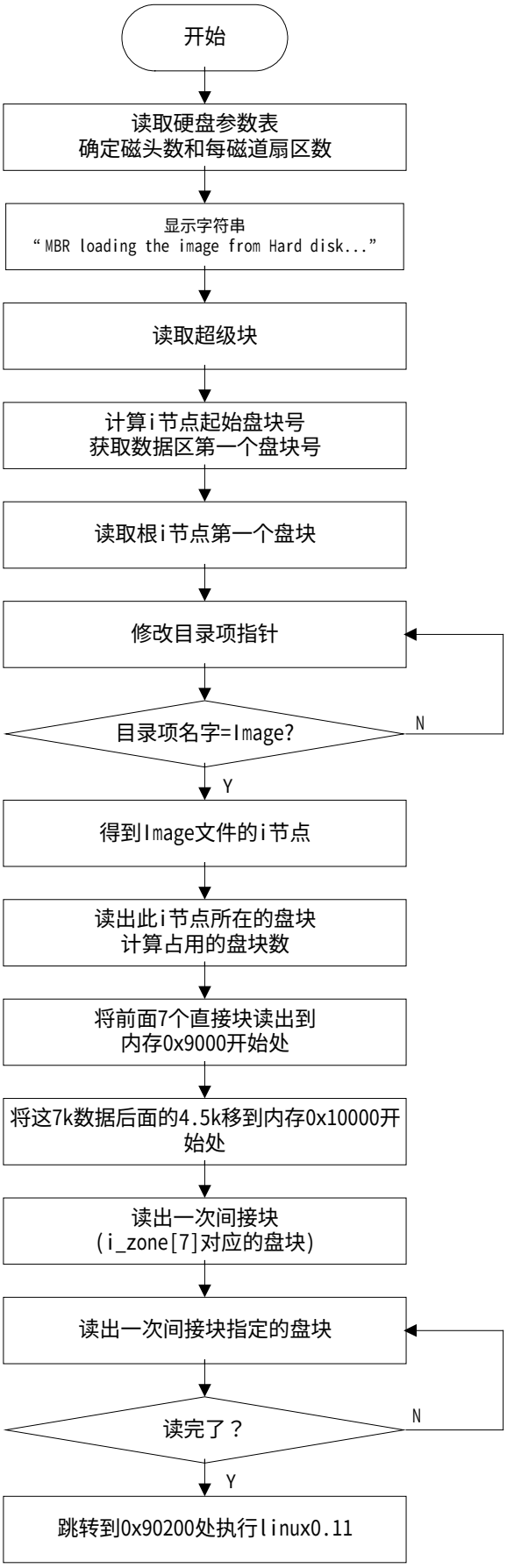
开始
读取硬盘参数表
确定磁头数和每磁道扇区数
显示字符串
“MBR loading the image from Hard disk...”
读取超级块
计算i节点起始盘块号
获取数据区第一个盘块号
读取根i节点第一个盘块
修改目录项指针
目录项名字=Image?
N
Y
得到Image文件的i节点
读出此i节点所在的盘块
计算占用的盘块数
将前面7个直接块读出到
内存0x9000开始处
将这7k数据后面的4.5k移到内存0x10000开始处
读出一次间接块
(i_zone[7]对应的盘块)
读出一次间接块指定的盘块
读完了？
N
Y
跳转到0x90200处执行linux0.11

图 5．MBR 程序流程图

四．上机实验

实验工具：

1、Bochs PC 仿真软件上运行的 SLS-Linux

下载地址：

http://oldlinux.org/Linux.old/bochs/sls-1.0.zip

2、linux-0.11-devel-040809.zip 压缩包(里面有 Bochs 安装程序)

下载地址：

http://oldlinux.org/Linux.old/bochs/linux-0.11-devel-040809.zip

3、DOS 格式映像文件工具 WinImage

下载地址：

http://ourworld.compuserve.com/homepages/gvollant/winima70.exe

实验步骤：

hdc-0.11.img 硬盘映像文件上有 linux0.11 根文件系统，但现在它不能引导 linux0.11：把 bochsrc-hd.bxrc 文件中的“ boot:a ”改为“ boot:c ”，运行 bochsrc-hd.bxrc 将导致引导失败。下面修改其 MBR 使得可以引导：

1、把 sls-1.0.zip 和 linux-0.11-devel-040923.zip 解压到同一个目录下，安装 Bochs2.1.1。

2、用文本编辑器打开 bochsrc.bxrc，把 69 行“ floppyb:1_44=tmp1.imz, status=inserted ”中的 tmp1.imz 改为 tmp.imz(原文有误)

3、在 bochsrc.bxrc 中添加一行，挂接 hdc-0.11.img：

“ ata0-slave: type=disk, path="hdc-0.11.img", mode=flat, cylinders=121, heads=16, spt=63 ”

4、使用 WimImage 打开 tmp.imz，把我们所写的 MBR 程序 bootload.s 和 makefile 拖入，保存

5、双击运行 bochsrc.bxrc，按空格键进入，使用 root 登录

6、运行如下命令将 linux0.11 映像文件复制到 hdb1 的根目录下：

```
mount /dev/hdb1 /mnt
cp /mnt/usr/src/linux/Image /mnt/Image
```

7、下面把存在 b 盘(即 tmp.imz 软盘映像)中的 bootload.s 和 makefile 复制出来：

```
mkdir asm
cd asm
mcopy b:\makefile makefile
mcopy b:\bootload.s bootload.s
```

8、汇编、链接 bootload.s，把可执行代码写到 hdb1 的 MBR 中去：

```
make    //参考附录 makefile 文件的说明
```

9、退出：

```
umount /mnt
```

关闭 Bochs

步骤 6～9 的运行界面如图 6

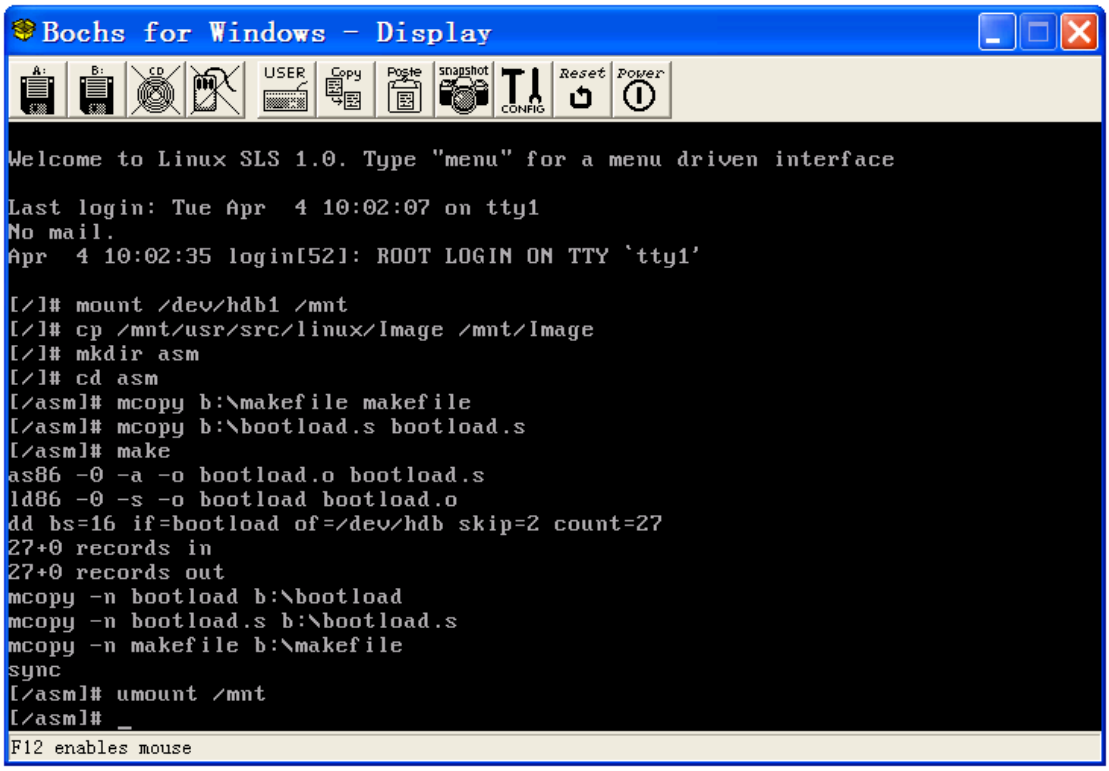

图 6. 修改硬盘映像文件 MBR

10、修改 bochsrc-hd.bxrc，把 boot:a 改为 boot:c，双击运行。成功！界面如下：

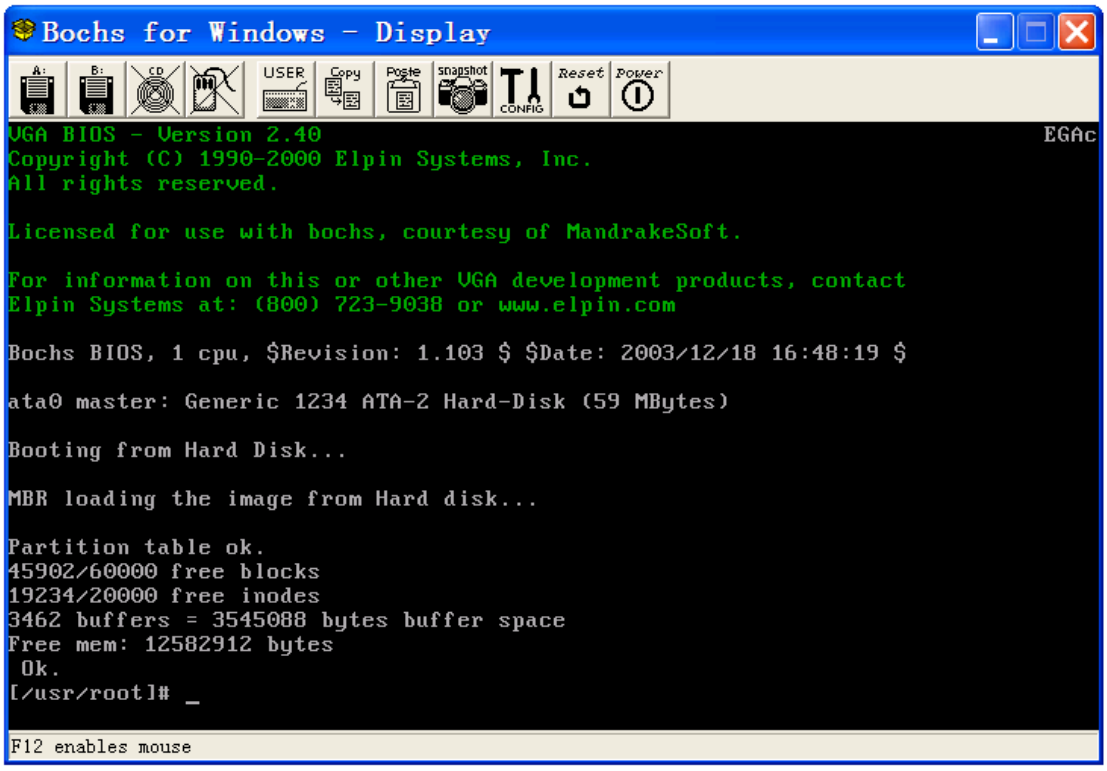

图 7. 从硬盘启动 linux0.11

五．程序代码

见随本文件发布的附件(一个是 MBR 程序 bootload.s，一个是 makefile 文件)

参考文献：

1、《带您深入了解硬盘分区表与逻辑锁》
作者：迈克尔
来源：家缘硬件网
日期：2005.1.9
链接：
http://www.pcnethome.com/Article/NEWS/skill/200501/1037.html

2、《Linux0.11 学习体会》
作者：chenfangjian
来源：http://www.oldlinux.org/论坛子论坛“Linux 内核完全注释”
链接：
http://www.oldlinux.org/cgi-bin/LB5000XP/topic.cgi?forum=1&topic=1507

3、《Linux 内核完全注释》
作者：赵炯
机械工业出版社

4、http://www.oldlinux.org/论坛子论坛“Linux 0.11 系统的建立和实验”上 linuxlala 回复的一篇帖子，他详细描述了编写 linux0.11 引导程序的方法。
链接：
http://www.oldlinux.org/cgi-bin/LB5000XP/topic.cgi?forum=4&topic=226