5.硬盘操作

CPU与外设、存储器的连接和数据交换都需要通过接口设备来实现。

每个连接到I/O总线上的设备都由自己的I/O地址集，即所谓的I/O端口（I/O port）。

每个设备的I/O端口都被组织成一组专用的寄存器，CPU可给控制寄存器发命令对设备进行控制、从状态寄存器读取设备状态、可以向输出寄存器写入数据来把数据输出到设备、可通过读取输入寄存器的内容来从设备取得数据。

总之就是通过读写端口来控制设备。

一个普通的PC主板上通常有两个IDE口，分别对应两个IDE通道:primary和secondary有时也成IDE0和IDE1。

每个IDE通道又能连接两个设备，称为主设备（Master）和从设备（Slave），对不同的IDE通道的访问是通过I/O端口来区分的。

IDE（integrated drive electronics）即电子集成驱动器，主要接硬盘和光驱。

接到主设备上的硬盘称为0号硬盘。

与0号硬盘有关的I/O端口：

1F0H　　 0号硬盘数据寄存器

1F1H　　 0号硬盘错误寄存器（读时）、0号硬盘Features寄存器（写时）

1F2H　　 0号硬盘数据扇区计数

1F3H　　 0号硬盘扇区数

1F4H　　 0号硬盘柱面（低字节）

1F5H　　 0号硬盘柱面（高字节）

1F6H　　 0号硬盘驱动器/磁头寄存器

1F7H　　 0号硬盘状态寄存器（读时）、0号硬盘命令寄存器（写时）

注：下图来自于渊《Orange‘s 一个操作系统的实现》，他使用的是LBA方式，babyos暂时使用CHS方式，故bit0～bit3表示磁头号，bit4为驱动器号为0，bit6为0，表示使用CHS方式。

[cpp] view plain copy print?

/\*

| 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |

+-----+-----+-----+-----+-----+-----+-----+-----+

| 1 | L | 1 | DRV | HS3 | HS2 | HS1 | HS0 |

+-----+-----+-----+-----+-----+-----+-----+-----+

| | \\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_/

| | |

| | `------------ If L=0, Head Select.

| | These four bits select the head number.

| | HS0 is the least significant.

| | If L=1, HS0 through HS3 contain bit 24-27 of the LBA.

| `--------------------------- Drive. When DRV=0, drive 0 (master) is selected.

| When DRV=1, drive 1 (slave) is selected.

`--------------------------------------- LBA mode. This bit selects the mode of operation.

When L=0, addressing is by 'CHS' mode.

When L=1, addressing is by 'LBA' mode.

\*/

即：

可以从端口0x1F0读取数据；

若发生错误可以从0x1F1读取错误；

若要从硬盘读数据可以从0x1F2指定读取的扇区数，0x1F3、0x1F4、0x1F5、0x1F6指定CHS（也可以是LBA，此处暂时不研究）

可以从0x1F7读取硬盘状态或向硬盘发送命令。

接口控制方式：

1）程序查询方式：CPU通过I/O指令询问指定外设当前状态，如果外设准备就绪，进行操作，否则CPU等待，循环查询。

2）中断处理方式：外设数据交换准备就绪，向CPU提出服务请求。

3）DMA（直接存储器存取）：采用一个专门的控制器来控制内存与外设直接的数据交流，无须CPU介入。

下面将要使用的方式是程序查询方式，因为babyos 还没有实现中断。

6.读取若干个扇区

现在只学习最简单的硬盘操作——从硬盘读取若干个扇区，其他功能以后学习文件系统时再研究。

步骤：

1）通过状态寄存器查询硬盘状态，看是否空闲，若忙，则等待

2）把读取扇区的个数、CHS写入相应端口

3）通过命令寄存器向硬盘发送读命令

4）从数据寄存器读取数据

注：

babyos目前只想从硬盘读若干个扇区，不想实现高级的硬盘操作功能和文件系统；

只考虑主硬盘；

暂时使用CHS方式，但会实现一个LBA到CHS的转换方法；

[cpp] view plain copy print?

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

> File: harddisk.c

> Describe: 实现基本硬盘操作功能

> Author: 孤舟钓客

> Mail: guzhoudiaoke@126.com

> Time: 2013年01月06日 星期日 17时40分32秒

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

#include <harddisk.h>

#include <io.h>

#include <font.h>

#include <graphics.h>

BOOL harddisk\_read(u32 lba, u32 sects\_to\_read, u8\* buffer)

{

u32 cylinder\_no, head\_no, sect\_no, temp;

u32 num\_of\_dwords;

cylinder\_no = lba / (HD0\_HEAD\_PER\_CYLINDER \* HD0\_SECT\_PER\_TRACK);

temp = lba % (HD0\_HEAD\_PER\_CYLINDER \* HD0\_SECT\_PER\_TRACK);

head\_no = temp / HD0\_SECT\_PER\_TRACK;

sect\_no = temp % HD0\_SECT\_PER\_TRACK + 1;

/\* 检查硬盘是否忙，忙则等待 \*/

while ((inb(HD\_PORT\_STATUS) & 0x80) != 0)

;

/\* 设置读取的扇区数和CHS，

HD\_PORT\_DRIVE\_HEAD端口bit7、bit5需要为1，bit6为0时bit0～bit3表示磁头号，

bit4为驱动器号，0 表示HD0，故下面head\_no要或操作10100000即0xa0 \*/

outb(sects\_to\_read, HD\_PORT\_SECT\_COUNT);

outb(sect\_no, HD\_PORT\_SECT\_NO);

outb(cylinder\_no, HD\_PORT\_CYLINDER\_LOW);

cylinder\_no >>= 8;

outb((cylinder\_no), HD\_PORT\_CYLINDER\_HIGH);

head\_no |= 0xa0;

outb((head\_no), HD\_PORT\_DRIVE\_HEAD);

/\* 发送读命令 \*/

outb(HD\_CMD\_READ, HD\_PORT\_COMMAND);

num\_of\_dwords = (sects\_to\_read << 7);

/\* 从HD\_PORT\_DATA读取数据，每个扇区512字节，即sects\_to\_read << 7个双字 \*/

insl(HD\_PORT\_DATA, buffer, num\_of\_dwords);

return TRUE;

}

1. 用户程序的结构：

1) 一般源程序都以段的形式进行组织，这样可以使逻辑更加清晰，在NASM中使用section关键字定义一个段，形式是：section 段名

2) 程序可以用段名来引用段，但是NASM编译器并不关心段的具体用途，或者说是根本不知道段的用途（代码段还是数据段等），同时NASM对段的数量也没有任何限制，如果代码中没有定一段则整个程序自成一段；

3) 定义段的同时可以定义段的一些属性，比如可以使用关键字align来定义段的对其方式，比如：section code align=16，这样就表示该段的其实地址是以16字节对齐的，即段的起始位置必须是16的整数倍；

！注意：该属性只影响段的起始位置的对其但不影响段的末尾对齐方式，事实上NASM也无法判断一个段的末尾，只有当遇到一个新的段的定义的时候才能知道前一个段结束了；

4) 段的起始位置：就是该段中第一行指令的地址（指令可以是普通指令也当然可以是数据定义指令db、dw等等；

5) 和加载程序之间的约定——应用程序头部Header：

i. 在有操作系统的环境下编译完一个程序之后编译器会隐式地、默认地添加一个应用程序头部（位置处于程序的起始位置处）；

ii. 头部包含着加载器该如何加载该程序的一些信息，或者说是加载器和程序之间的某些约定或规范，而加载器通过这些信息将程序正确地加载进内存中；

iii. 在有操作系统的环境下，应用程序头部和加载器都是操作系统负责的，但是在这里我们模拟一下这个操作系统的工作，即手写完成加载器和程序头部来模拟操作系统的这两个功能；

2. 用户程序和加载器的简单实现：

用户程序：默认程序已经正确加载到了内存的空闲位置，并且从定义的标号start处该是执行程序，作用是将两个数据段中的字符串打印到屏幕上，并且处理回车和换行两个控制符

！注意用户程序头部的定义，里面包含了程序大小、程序开始执行的入口、程序中各个段的起始位置等信息；

！其中重定位表的作用就是：编译后各项保存的是各段在源程序中的绝对汇编地址，经加载程序加载后就将各项修改成在物理内存中实际的地址，因此称为重定位表；

app.nas，编译后生成app.bin

[plain] view plain copy

; 应用程序头

; 用于提供加载器相关加载信息

; 是应用程序规范的一部分

section header vstart=0

app\_size dd app\_end ; [APP\_SIZE:0x00] 程序的大小（字节）

app\_entry dw start ; [APP\_ENTRY:0x04] 入口处偏移地址

app\_entry\_seg dd section.code1.start ; [APP\_ENTRY\_SEG:0x06] 入口处段地址

; section.段名.start是NASM提供的伪指令，用于段起始位置在源程序中的绝对汇编地址

; 绝对汇编地址是指相对于整个源程序头的偏移量，而整个程序头的绝对汇编地址是0

; 绝对汇编地址是一个32位无符号数，因此使用dd表示

c\_realloc\_tbl dw (tbl\_end - tbl\_start) / 4 ; [C\_REALLOC\_TBL:0x0A] 重定位表表项数目

tbl\_start: ; [TBL\_START:0x0C]

seg\_addr\_code1 dd section.code1.start

seg\_addr\_code2 dd section.code2.start

seg\_addr\_data1 dd section.data1.start

seg\_addr\_data2 dd section.data2.start

seg\_addr\_stack dd section.stack.start

tbl\_end:

; section header end

;;

;;

section stack align=16 vstart=0

resb 256

stack\_end:

; section stack end

;;

;;

section data1 align=16 vstart=0

msg0 db ' This is NASM - the famous Netwide Assembler. '

db 'Back at SourceForge and in intensive development! '

db 'Get the current versions from http://www.nasm.us/.'

db 0x0d,0x0a,0x0d,0x0a

db ' Example code for calculate 1+2+...+1000:',0x0d,0x0a,0x0d,0x0a

db ' xor dx,dx',0x0d,0x0a

db ' xor ax,ax',0x0d,0x0a

db ' xor cx,cx',0x0d,0x0a

db ' @@:',0x0d,0x0a

db ' inc cx',0x0d,0x0a

db ' add ax,cx',0x0d,0x0a

db ' adc dx,0',0x0d,0x0a

db ' inc cx',0x0d,0x0a

db ' cmp cx,1000',0x0d,0x0a

db ' jle @@',0x0d,0x0a

db ' ... ...(Some other codes)',0x0d,0x0a,0x0d,0x0a

db 0

; section data1 end

;;

;;

section data2 align=16 vstart=0

msg1 db ' Welcome and enjoy NASM! '

db '2015-01-05'

db 0

; section data2 end

;;

;;

section code1 align=16 vstart=0

start:

mov ax, [seg\_addr\_stack]

mov ss, ax

mov sp, stack\_end

mov ax, [seg\_addr\_data1]

mov ds, ax

mov bx, msg0

call put\_string ; 显示第一段信息

; 在加载程序中将es指向header了

push word [es:seg\_addr\_code2] ; 先将code2的偏移地址和段地址入栈

mov ax, \_start.begin

push ax

retf ; 利用retf修改cs:ip使其跳转至code2

.continue:

mov ax, [es:seg\_addr\_data2]

mov ds, ax

mov bx, msg1

call put\_string ; 使ds:bx指向msg1并输出

jmp $

; end start

; 字符串控制宏以及显卡光标端口宏

CHAR\_TRAIL equ 0x00 ; 字符串结束符

CHAR\_RET equ 0x0D ; 回车符

CHAR\_NL equ 0x0A ; 换行符

DCHAR\_NONE equ 0x0720 ; 显存中显示空的字

PORT\_CHOOSE equ 0x3D4 ; 索引端口，用于选择子端口（8位）

SUBPORT\_HIGH equ 0x0E ; 子端口号

SUBPORT\_LOW equ 0x0F ; 这两个子端口分别存放光标位置的高位和低位

PORT\_DATA equ 0x3D5 ; 数据端口，存放选定的端口中的数据（8位）

VIDEO\_SEG\_BEGIN equ 0xB800 ; 显卡区域起始段地址

; func put\_string

; <- [ds:bx]:msg0

; colision register: es

; 将msg0打印至屏幕

put\_string:

push es

; 获取当前光标位置保存在ax中

mov dx, PORT\_CHOOSE

mov al, SUBPORT\_HIGH

out dx, al ; 选择一个子端口

mov dx, PORT\_DATA

in al, dx

mov ah, al ; 从子端口中读取光标高位保存在ah中

mov dx, PORT\_CHOOSE

mov al, SUBPORT\_LOW

out dx, al

mov dx, PORT\_DATA

in al, dx ; 同理从子端口中读取光标低位保存在al中

; 最终将整个结果保存在ax中

; 目前ax存放着光标的位置

.lp: mov cl, [bx] ; 读取一个字符保存在cl中

cmp cl, CHAR\_TRAIL ; 判断该字符是否是结束符

je .ret

call put\_char ; 不是结束符就打印该字符

inc bx ; 继续读取下一个字符

jmp .lp

.ret: pop es

ret

; func put\_char

; <- cl:当前读取的一个字符

; colision register: ds, bx

put\_char:

push ds

push bx ; 备份

; ds和es都指向显卡

mov bx, VIDEO\_SEG\_BEGIN

mov ds, bx

mov es, bx

; 目前ax存放着光标的位置

cmp cl, CHAR\_RET ; 判断字符是否是回车

jne .next0 ; 不是回车则继续接下来的步骤

.deal\_ret: ; 是回车则处理回车

mov bl, 80

div bl

mul bl ; 除去光标位置中80的余数即可

; ax中得到的是回车后光标的位置

jmp .set\_cursor

.next0: cmp cl, CHAR\_NL ; 判断是否是换行符

jne .next1 ; 如果不是换行符则继续接下来的代码

.deal\_nl: ; 处理换行的情形

add ax, 80 ; 换行很简单，只要加80即可

jmp .deal\_roll\_screen ; 换行可能会造成屏幕滚动，因此需要处理

.next1: ; 结束、回车、换行都不是那就是普通字符了，因此需要打印出来，并且光标后移一位

mov bx, ax ; 先将ax复制到bx中

shl bx, 1 ; 显卡区域每个字符占两个字节（还有一个属性字节）

mov [bx], cl

inc ax ; 光标后移一位

; jmp .deal\_roll\_screen ; 光标后移也可能会造成滚屏

.deal\_roll\_screen:

cmp ax, 2000

jl .set\_cursor ; 检查光标是否越界，如果越界则需要滚屏，否则可以直接设置光标

.roll\_screen: ; 滚屏处理

mov si, 80 \* 2

mov di, 0

mov cx, 2000 - 80

cld

rep movsw

.clear\_bottom\_line: ; 滚屏后需要清除最后一行

mov bx, (2000 - 80) \* 2

mov cx, 80

.cls:

mov word [bx], DCHAR\_NONE

add bx, 2

loop .cls

mov ax, 2000 - 80 ; 滚屏后光标位置设置成最后一行起始

; jmp .set\_cursor ; 滚屏完成后方可显示新的光标的位置了

.set\_cursor:

mov bx, ax ; 将光标位置备份到bx中，因为访问端口会用到ax

mov dx, PORT\_CHOOSE

mov al, SUBPORT\_HIGH

out dx, al

mov dx, PORT\_DATA

mov al, bh

out dx, al

mov dx, PORT\_CHOOSE

mov al, SUBPORT\_LOW

out dx, al

mov dx, PORT\_DATA

mov al, bl

out dx, al

pop bx

pop ds

ret

; section code1 end

;;

;;

section code2 align=16 vstart=0

\_start:

.begin: push word [es:seg\_addr\_code1] ; code2没做什么实事就是再跳回code1的continue继续执行

mov ax, start.continue

push ax

retf

; section code2 end

;;

;;

section trail align=16

app\_end:

; section trail end

！resb指令就是reserve byte的缩写，即保留一定数量的字节的意思，因此必然还有resw、resd，表示保留一定数量的字和双字的意思，既然是保留就不对定义的数据进行初始化，因此该指令就是定义一段连续的未初始化的数据；

！关于汇编地址的介绍以及硬盘访问的端口都在源代码中详细介绍，所以这里就不累述了；

加载器的实现：作为主引导扇区程序

loader.nas，编译后生成loader.mbr

[plain] view plain copy

; 主引导扇区程序作为应用程序加载器

; 虽然就只有一个段但是也需要定义

; 最主要是为了使用段属性vstart=0x7C00

; 这样就可以使得段内的所有汇编地址都是相对0x7C00开始的

; 因为MBR加载在0x0000:0x7C00处，因此IP初始化为0x7C00

; 而所有偏移地址都是相对0x7C00的

; 有了这一步程序中的所有标号都能真正代表偏移地址了

section loader align=16 vstart=0x7C00

jmp near start

LBA\_APP\_START equ 100 ; 应用程序所在硬盘的起始逻辑扇区号，这里是人为规定的

ADDR\_20\_LOAD\_START dd 0x10000 ; 内存中加载的起始20位绝对物理地址

; 应用程序头中信息的偏移地址

APP\_SIZE\_LOW equ 0x00

APP\_SIZE\_HIGH equ 0x02

APP\_ENTRY equ 0x04

APP\_ENTRY\_SEG equ 0x06

APP\_ENTRY\_SEG\_LOW equ 0x06

APP\_ENTRY\_SEG\_HIGH equ 0x08

C\_REALLOC\_TBL equ 0x0A

TBL\_START equ 0x0C

; 从0x0FFFF往下（即地址减小）的一段区域一般都作为MBR的栈！

; 因此ss:sp指向0x0000:0x0000

; 这样在push的时候sp能回到0xFFFF

start: mov ax, 0

mov ss, ax

mov sp, ax

; ds -> 内存中加载的起始位置段地址

mov ax, [cs:ADDR\_20\_LOAD\_START]

mov dx, [cs:ADDR\_20\_LOAD\_START+2]

mov bx, 16

div bx

mov ds, ax

mov es, ax ; 留给用户程序时使ds和es都指向加载位置首部

; 先读取一个扇区，即应用程序头所在的扇区

xor di, di

mov si, LBA\_APP\_START ; [di:si]全局保存当前读取的逻辑扇区号

mov cx, 1 ; 读取一个扇区

call read\_lba

; 读取完毕，ds:0指向程序的第一扇区中的内容

mov dx, [APP\_SIZE\_HIGH]

mov ax, [APP\_SIZE\_LOW]

mov bx, 512

div bx

cmp dx, 0

jne .deal\_left ; 有余数，可以将已经读取的那个扇区看做余数的扇区

dec ax ; 无余数则需要减去已经读取的那个扇区

.deal\_left:

cmp ax, 0

je redirect\_entry ; 如果没有剩余扇区要读则直接去重定位程序入口点

push ds ; 备份并改变其指向

mov cx, ax ; 剩余要读的扇区数量

mov ax, ds

add ax, 0x20 ; 使其指向下一个512字节起始处（必然是16位对齐的）

mov ds, ax

inc si ; 指向下一个要读的扇区

call read\_lba

pop ds ; 恢复ds使其指向加载的程序的开始处

; 到此为止程序彻底加载完毕

; 接下来的工作是将程序头中的入口地址，以及重定位表中的地址

; 修改成实际的物理地址

; 这里所重定位的地址都是段地址

; 将程序中段的绝对汇编地址更新成加载在内存中的实际物理段地址

; 公式是：16位物理段地址 = (整个程序起始位置的20位物理 + 段的32位绝对汇编地址) >> 4

redirect\_entry: ; 重定位入口处地址

mov dx, [APP\_ENTRY\_SEG\_HIGH] ; [dx:ax]中保存入口处的绝对汇编地址

mov ax, [APP\_ENTRY\_SEG\_LOW]

call calc\_seg\_phy\_addr\_16 ; 计算段的16位段地址（即物理段地址），结果保存在ax中

mov [APP\_ENTRY\_SEG], ax ; 更新

; 处理重定位表

mov cx, [C\_REALLOC\_TBL]

mov bx, TBL\_START

.realloc:

mov dx, [bx + 2]

mov ax, [bx]

call calc\_seg\_phy\_addr\_16

mov [bx], ax

add bx, 4

loop .realloc

jmp far [APP\_ENTRY] ; 控制权交给应用程序

; func read\_lba

; <- [di:si]:读取的逻辑扇区号

; <- cx:读取的扇区数量

; <- ds:目的区域段地址

; 将cx个扇区的内容读取到ds:0所指向的内存空间中

read\_lba:

PORT\_DATA equ 0x1F0 ; 数据端口（16位）

PORT\_ERRNO equ 0x1F1 ; 错误端口（8位）保存最后一次执行命令后的状态（错误原因）

PORT\_CLBA equ 0x1F2 ; 计数端口（8位）保存读写的扇区数量

PORT\_LBA\_START equ 0x1F3 ; 逻辑扇区号端口（32位共4个8位口）

; 低28位确定待操作的起始扇区号

; 最高的4位指定扇区寻址模式以及类型选择符）

PORT\_CTRL equ 0x1F7 ; 控制端口（8位）下读写命令同时又能反映硬盘工作状态

CTRL\_READ equ 0x20 ; 读命令，向控制端口发送

BIT\_MASK equ 10001000B ; 位掩码，取控制端口的第7位和第3位

; 第7位表示硬盘是否忙，1表示忙

; 第3位表示硬盘是否就绪，1表示就绪

STATUS\_READY equ 00001000B ; 彻底就绪时第7位是0，第3位是1，用于检测硬盘是否就绪

; 指定读取的扇区数量

mov dx, PORT\_CLBA

mov al, cl

out dx, al

; 向LBA地址口写入28位逻辑扇区号

mov dx, PORT\_LBA\_START ; 0~7位

mov ax, si

out dx, al

inc dx ; 8~15位

mov al, ah

out dx, al

inc dx ; 16~23位

mov ax, di

out dx, al

inc dx ; 24~27位

mov al, 111\_0\_0000B ; ah保存24~27位，al中保存扇区寻址模式以及类型选择符

; 其中最高位的111表示采用28位逻辑扇区号模式

; 后面一位的0表示是主盘，1表示从盘，即盘片类型选择符

or al, ah

out dx, al

; 发出读命令

mov dx, PORT\_CTRL

mov al, CTRL\_READ

out dx, al

.waits: ; 检测硬盘是否就绪，没就绪就一直等待就绪

in al, dx

and al, BIT\_MASK

cmp al, STATUS\_READY

jne .waits

; 准备就绪就开始读取

shl cx, 8 ; 一个扇区512B，即256个字

; cx记录剩余多少字未读完，而cx原本存放剩余扇区数

; 因此cx要乘以256，即左移8位

mov dx, PORT\_DATA

xor bx, bx

.readw: ; 循环读取程序，将其加载至ds:0处

in ax, dx

mov [bx], ax

add bx, 2

loop .readw

ret

; func calc\_seg\_phy\_addr\_16

; <- [dx:ax]:段32位绝对汇编地址

; -> ax:16位物理段地址

calc\_seg\_phy\_addr\_16:

; 这里的20位起始加载地址使用32位保存的

; 因此可以通过带进位的加法得到段起始位置的实际的20位物理地址

add ax, [cs:ADDR\_20\_LOAD\_START]

adc dx, [cs:ADDR\_20\_LOAD\_START+2]

; 现在将绝对的20位物理地址右移4位就能得到16位的物理段地址了

; 必须dx和ax同时右移

; 方法是ax右移4位即可

; 而dx采用循环右移4位，应该移到ax高4位的那4位重新回到dx高4位

; 然后用位掩码去的dx高4位

; 再利用or将这4位写入ax的高4位即可

shr ax, 4 ; 低16位右移4位

ror dx, 4

and dx, 0xF000 ; 位掩

or ax, dx ; 写入

ret

times 510-($-$$) db 0

dw 0xAA55

！注意：程序开始处的一大堆宏定义就是用于和应用程序头部进行通讯的，即那些表项等在头部中的偏移位置，这样就可以轻松访问这些表项了；

3. 查看程序运行结果：将loader.mbr写进虚拟硬盘的0号扇区（总共一个扇区），将app.bin写进100号扇区（总共两个扇区），然后将虚拟盘作为虚拟机的启动盘放在VirtualBox中运行中即可；

第四节 引导扇区

引导扇区是硬盘、软盘或类似的数据存储设备的一个扇区，内含负责引导（booting）“存放在盘片（disk）的其他部分的程序（通常，但不必然是操作系统）”的机器码。

引导扇区有两种：

Volume Boot Record 是磁盘未被分区的第一个扇区，或已分区的分区的第一个扇区，包含了加载与唤起操作系统(放在这个分区之内或放在这个磁盘上)的码。

Master Boot Record 是磁盘已被分区的第一个扇区，它包含定位活动分区与唤起它的VBR的码。

IBM PC兼容机上，BIOS不在意VBR与MBR的不同，甚至分区。固件只是加载并运行磁盘的第一个扇区(sector)，在MBR里的码，才知道磁盘分区消息；且是负责加载引导活动分区的VBR的地方。

如果你从“一个没有安装操作系统的磁盘”引导，屏幕会显示"Please Insert a bootable disc and press a key"；这是开机扇区(boot sector)显示的，而不是机器的固件。

主引导记录（Master Boot Record，缩写：MBR），又叫做主引导扇区，是电脑开机后访问硬盘时所必须要读取的首个扇区，主引导扇区记录着硬盘本身的相关消息以及硬盘各个分区的大小及位置消息，是数据消息的重要入口。如果它受到破坏，硬盘上的基本数据结构消息将会丢失，需要用繁琐的方式试探性的重建数据结构消息后才可能重新访问原先的数据，对于那些扇区为512位组的磁盘，MBR分区表不支持容量大于2.2TB（2.2×1012字节）的分区。

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 标准MBR结构 | | | | | |
| **地址** | | | **描述** | | **长度** |
| Hex | Oct | Dec | （字节） |
| 0 | 0 | 0 | 代码区 | | **440** |
| （最大446） |
| 01B8 | 670 | 440 | 选用磁盘标志 | | 4 |
| 01BC | 674 | 444 | 一般为空值; 0x0000 | | 2 |
| 01BE | 676 | 446 | **标准MBR分区表规划** | | **64** |
| （四个16 byte的主分区表入口） | |
| 01FE | 776 | 510 | 55h | MBR有效标志： | **2** |
| 01FF | 777 | 511 | AAh | 0x55AA |
| **MBR，总大小：446 + 64 + 2 =** | | | | | **512** |

主引导记录（Master Boot Record，缩写：MBR），又叫做主引导扇区，是计算机开机后访问硬盘时所必须要读取的首个扇区，它在硬盘上的三维地址为（柱面，磁头，扇区）＝（0，0，1）。在深入讨论主引导扇区内部结构的时候，有时也将其开头的446字节内容特指为“主引导记录”（MBR），其后是4个16字节的“磁盘分区表”（DPT），以及2字节的结束标志（55AA）。因此，在使用“主引导记录”（MBR）这个术语的时候，需要根据具体情况判断其到底是指整个主引导扇区，还是主引导扇区的前446字节。

主引导扇区记录着硬盘本身的相关信息以及硬盘各个分区的大小及位置信息，是数据信息的重要入口。如果它受到破坏，硬盘上的基本数据结构信息将会丢失，需要用繁琐的方式试探性的重建数据结构信息后才可能重新访问原先的数据。主引导扇区内的信息可以通过任何一种基于某种操作系统的分区工具软件写入，但和某种操作系统没有特定的关系，即只要创建了有效的主引导记录就可以引导任意一种操作系统（操作系统是创建在高级格式化的硬盘分区之上，是和一定的文件系统相联系的）。

对于硬盘而言，一个扇区可能的字节数为128×2n（n=0,1,2,3）。大多情况下，取n=2，即一个扇区（sector）的大小为512字节。

主引导记录最开头是第一阶段引导代码。其中的硬盘引导程序的主要作用是检查分区表是否正确并且在系统硬件完成自检以后将控制权交给硬盘上的引导程序（如GNU GRUB）。它不依赖任何操作系统，而且启动代码也是可以改变的，从而能够实现多系统引导。

第五节 硬盘分区表

硬盘分区表占据主引导扇区的64个字节（偏移01BEH--偏移01FDH），可以对四个分区的信息进行描述，其中每个分区的信息占据16个字节。具体每个字节的定义可以参见硬盘分区结构信息。

|  |  |  |
| --- | --- | --- |
| 硬盘分区结构信息 | | |
| **偏移** | **长度 （字节）** | **意义** |
| 00H | 1 | 分区状态：00-->非活动分区；80-->活动分区； |
| 其它数值没有意义 |
| 01H | 1 | 分区起始磁头号（HEAD），用到全部8位 |
| 02H | 2 | 分区起始扇区号（SECTOR），占据02H的位0－5； |
| 该分区的起始磁柱号（CYLINDER），占据 |
| 02H的位6－7和03H的全部8位 |
| 04H | 1 | 文件系统标志位 |
| 05H | 1 | 分区结束磁头号（HEAD），用到全部8位 |
| 06H | 2 | 分区结束扇区号（SECTOR），占据06H的位0－5； |
| 该分区的结束磁柱号（CYLINDER），占据 |
| 06H的位6－7和07H的全部8位 |
| 08H | 4 | 分区起始相对扇区号 |
| 0CH | 4 | 分区总的扇区数 |

下面是一个例子：

如果某一分区在硬盘分区表的信息如下

80 01 01 00 0B FE BF FC 3F 00 00 00 7E 86 BB 00

则我们可以看到，最前面的"80"是一个分区的激活标志，表示系统可引导；"01 01 00"表示分区开始的磁头号为1，开始的扇区号为1，开始的柱面号为0；"0B"表示分区的系统类型是FAT32，其他比较常用的有04（FAT16）、07（NTFS）；"FE BF FC"表示分区结束的磁头号为254，分区结束的扇区号为63、分区结束的柱面号为764；"3F 00 00 00"表示首扇区的相对扇区号为63（小端序）；"7E 86 BB 00"表示总扇区数为12289662（小端序）。

对于大于8.4G的现代硬盘，CHS已经无法表示, BIOS使用LBA模式，对于超出的部分，CHS值通常设为0xFEFFFF，并加以忽略，直接使用Offset 0x08-0x0c的4字节相对值，再进行内部转换。

结束标志字55，AA（偏移1FEH－偏移1FFH）最后两个字节，是检验主引导记录是否有效的标志。

主引导记录与硬盘分区

从主引导记录的结构可以知道，它仅仅包含一个64个字节的硬盘分区表。由于每个分区信息需要16个字节，所以对于采用MBR型分区结构的硬盘，最多只能识别4个主要分区（Primary partition）。所以对于一个采用此种分区结构的硬盘来说，想要得到4个以上的主要分区是不可能的。这里就需要引出扩展分区了。扩展分区也是主要分区的一种，但它与主分区的不同在于理论上可以划分为无数个逻辑分区。

扩展分区中逻辑驱动器的引导记录是链式的。每一个逻辑分区都有一个和MBR结构类似的扩展引导记录（EBR），其分区表的第一项指向该逻辑分区本身的引导扇区，第二项指向下一个逻辑驱动器的EBR，分区表第三、第四项没有用到。

Windows系统默认情况下，一般都是只划分一个主分区给系统，剩余的部分全部划入扩展分区。这里有下面几点需要注意：

在MBR分区表中最多4个主分区或者3个主分区＋1个扩展分区，也就是说扩展分区只能有一个，然后可以再细分为多个逻辑分区。

在Linux系统中，硬盘分区命名为sda1－sda4或者hda1－hda4（其中a表示硬盘编号可能是a、b、c等等）。在MBR硬盘中，分区号1－4是主分区（或者扩展分区），逻辑分区号只能从5开始。

在MBR分区表中，一个分区最大的容量为2T，且每个分区的起始柱面必须在这个disk的前2T内。你有一个3T的硬盘，根据要求你至少要把它划分为2个分区，且最后一个分区的起始扇区要位于硬盘的前2T空间内。[3]如果硬盘太大则必须改用GPT。

主引导扇区的读取流程

系统开机或者重启。

BIOS加电自检（Power On Self Test -- POST）。BIOS执行内存地址为FFFF:0000H处的跳转指令，跳转到固化在ROM中的自检程序处，对系统硬件（包括内存）进行检查。

读取主引导记录（MBR）。当BIOS检查到硬件正常并与CMOS中的设置相符后，按照CMOS中对启动设备的设置顺序检测可用的启动设备。BIOS将相应启动设备的第一个扇区（也就是MBR扇区）读入内存地址为0000:7C00H处。

检查0000:7CFEH-0000:7CFFH（MBR的结束标志位）是否等于55AAH，若不等于则转去尝试其他启动设备，如果没有启动设备满足要求则显示"NO ROM BASIC"然后死机。

当检测到有启动设备满足要求后，BIOS将控制权交给相应启动设备。启动设备的MBR将自己复制到0000:0600H处，然后继续执行。

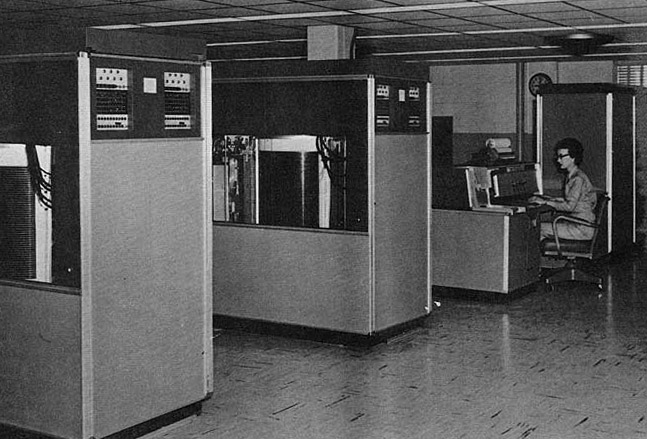
根据MBR中的引导代码启动引导程序。

事实上，BIOS不仅检查0000:7CFEH-0000:7CFFH（MBR的结束标志位）是否等于55AAH，往往还对磁盘是否有写保护、主引导扇区中是否存在活动分区等进行检查。如果发现磁盘有写保护，则显示磁盘写保护出错信息；如果发现磁盘中不存在活动分区，则显示类似如下的信息“Remove disk or other media Press any key to restart”。

全局唯一标识分区表（GUID Partition Table，缩写：GPT）是一个实体硬盘的分区表的结构布局的标准。它是可扩展固件接口（EFI）标准（被Intel用于替代个人计算机的BIOS）的一部分。GPT分配64bits给逻辑块地址，因而使得最大分区大小在264-1个扇区成为了可能。对于每个扇区大小为512字节的磁盘，那意味着可以有9.4ZB（9.4 x 1021字节）或8 ZiB-512字节（9,444,732,965,739,290,426,880字节或 18,446,744,073,709,551,615（264-1）个扇区x 512（29）字节每扇区）。

与支持最大卷为2 TB（Terabytes）并且每个磁盘最多有4个主分区（或3个主分区，1个扩展分区和无限制的逻辑驱动器）的MBR磁盘分区的样式相比，GPT磁盘分区样式支持最大卷为128 EB（Exabytes）并且每磁盘的分区数没有上限，只受到操作系统限制（由于分区表本身需要占用一定空间，最初规划硬盘分区时，留给分区表的空间决定了最多可以有多少个分区，IA-64版Windows限制最多有128个分区，这也是EFI标准规定的分区表的最小尺寸）。与MBR分区的磁盘不同，至关重要的平台操作数据位于分区，而不是位于非分区或隐藏扇区。另外，GPT分区磁盘有备份分区表来提高分区数据结构的完整性。

在20世纪50年代中期，虽然之前人们已经在使用打孔卡和磁带进行数据的存储，但是要想在上述存储介质上查找某个数据却非常困难，往往需要数小时的时间，就是因为这些存储产品采用的是顺序存取技术。而那些被昵称为“造反派”、“牛仔”的IBM实 验室的技术人员一个十分单纯的想法就是，找到一种随机存取的方法，加快数据的存取速度。Al Hoagland是当时18个参与此项产品研发的人之一，他当时还是加州大学伯克利分校的研究生。他回忆说：“当时的气氛真有点像火箭发射，在达到最后目的地之前，谁也不知道能否成功，新的产品又会是什么样。1956年9月14日，他们终于宣布开发出了一种将对全球计算领域产生重大影响的产品，那就是统计控制随机存取法（Random Access Method for Accounting Control，RAMAC）”。这款商用磁盘存储系统就是RAMAC 305，有两个冰箱那么宽，内部安装了50个直径两英尺的磁盘，重量约1吨，当时可以存储“惊人”的500万个字符(5MB)。



BRL61-IBM 305 RAMAC

时间转移到1962年，这时已经是14英寸的硬盘占据市场的统治地位，直到70年代中期，14英寸硬盘几乎占据了全部硬盘市场，几乎所有这些设备都出售给大型计算机制造商。那个时候那种大型硬盘也并不是普通用户能消费的起的，而且容量不过百兆左右。

到了70年代末期，8英寸硬盘就已经诞生，体积也相应减小了不少。

1978-1980年，更小的8英寸驱动器被开发出来，其中包括Shugart Assaciates、Micropolis、priam和昆腾这些老牌硬盘厂商，不过容量仅为10M、20M、30M以及40M，相比14英寸硬盘而言要小了很多，所以这种型号受到了当时只需要大容量硬盘的大型计算机制造商的冷落，因此这些8英寸型号的市场新入者将他们的创新性硬盘投入新的应用：小型计算机。

8英寸产品在对于成熟的小型计算机制造商来说重要的性能标准方面，容量、单位存储成本和存取速度是非常优越的，随后几年，随着8英寸硬盘制造商通过积极地采用技改性改进，已极快的速度扩大产品的容量，而且由于单位容量极大增长，使得8英寸硬盘单位存储成本跌至14英寸硬盘以下，很快成为了市场新的霸主。

时间转到了1980年，硬盘的体积终于又出现了变化，下面这款就是世界第一台5.25英寸硬 盘驱动器ST-506，作为首款真正面向台式机的硬盘，5.25寸的出现势必具有其特殊的意义，对于许多80后的电脑玩家来说，所接触到的第一块电脑硬盘 大部分是5.25英寸开始的，虽然它的容量仅有5MB，但它的出现却带动了一个时代。

时隔四年，到了1984年，一家苏格兰企业Rodime首先开发出了3.5寸硬盘，相信当时谁也没有料到，这个尺寸现在会成就如此伟大的时代，因为就如同8英寸硬盘出世之初一样，3.5英寸结构硬盘在一开始推出时根本不被重视，其原因也是因为成本高而容量太小，难以满足人们的需求，到了88年为止，也仅有不到一半的硬盘厂商开始生产3.5英寸硬盘。

直到90年代，3.5寸硬盘才开始真正走向辉煌，而5.25英寸走向了暮年，不仅因为5.25英寸盘的转速最终不能提的太高，影响寻道时间，而且在可靠性和成本等方面也存在诸多问题，因此后来厂商都放弃了5.25英寸盘的设计思路，转向了3.5寸和2.5寸。

转眼就到2.5寸硬盘的出现了，1989年科罗拉多州朗蒙特的一家市场新入者Prairietek宣布推出一种2.5英寸硬盘，也因为它成了硬盘行业的焦点，到90年以后各硬盘厂商也都纷纷推出了自己的2.5英寸硬盘。

这一次的硬盘厂商在态度上明显有了区别，在14寸转8寸与5.25寸转3.5寸时，硬盘厂商都纷纷推迟，而这一次，却一反常态，表现十分积极，这就要归功于当时的笔记本制造商，2.5英寸硬盘轻便、省电、体积小的特点很让当时的笔记本制造商看 好，纷纷订购，所以才有了2.5英寸硬盘如此快速的在硬盘市场站稳脚跟，也让这个尺寸一直沿用至今，演变为如今的局面。

到了1992年，更加MINI的1.8寸硬盘诞生了，1.8英寸硬盘的大小已经到达和名片一 般，小巧便携与省电的优势更加的明显，不过另一方面，价格较2.5英寸要更加昂贵，以至于普通的笔记本也不可以使用，一般是放在超便携设备，当然部分 MINI笔记本也会使用1.8寸盘。

硬盘的体积越来越小，也让我们看到了技术的一步步进步，现在越小巧的硬盘越用在更高端的产品上，就如同1.8英寸硬盘推出时最大的市场不是在电脑中，而是在便携式的心脏监控设备中！

随着笔记本电脑市场的不断增长和数码相机、数码摄相机、个人PDA、MP3播放器和高端手机 等手持移动数码设备的迅速升级换代，人们对移动存储设备的要求也越来越高，大容量小体积成为了移动存储设备的发展趋势。闪存盘虽然体积小但容量也太小,而传统的移动硬盘虽然容量大但体积又太大，所以这个时候微硬盘的概念诞生了，而1英寸硬盘也应运而生。

东芝发布的0.85英寸微型硬盘比IBM发明的1.0英寸微型硬盘还要小巧，该微型硬盘相信 可以用于更小型的移动设备中，包括MP3音频播放器、手机、PDA、GPS全球定位仪、数码摄像机、多媒体播放器等手持掌上设备。随着0.85英寸硬盘容 量的提升和未来几年更有竞争力的价格到来，相信能为微型硬盘市场再创辉煌。

　　1、1991年，40 MB的硬盘花了37秒才可以读完一片有26 MB容量的磁盘(platter)。

　　2、1998年，3.2 GB的硬盘花了3分31秒才可以读完一片有1.6 GB容量的磁盘。

　　3、1999年，10 GB的硬盘花了5分37秒才可以读完一片有3.2 GB容量的磁盘。

　　4、2004年，60 GB的硬盘花了18分34秒才可以读完一片有40 GB容量的磁盘。

　　5、2006年，750 GB的硬盘花了52分才可以读完一片有200 GB容量的磁盘。