30 日でできる! OS 自作入門

30 天自制操作系统

读书笔记

【日】川合秀实　著

周自恒 李黎明 曾祥江 张文旭　译

前　言

本文档为《30 天自制操作系统》（人民邮电出版社）一书的读书笔记。

文档发布在 Github 上，将随着阅读进度不定期更新。请访问 https://github.com/mengyingchina/osask-notes 获取文档最新的版本。

原书的版权声明一节，有：

> 本书中文简体字版由 Mynavi Corporation 授权人民邮电出版社独家出版。未经出版者书面许可，不得以任何方式复制或抄袭本书内容。

由于部分内容摘自书中，不清楚这样摘录部分内容会不会存在版权问题，如果有版权问题，我会及时删除相关内容，请知情者告知[1]，谢谢！

[1]Email:mail@wanhu.me

目　　录

第 0 天　着手开发之前

1　前言

阅读本书几乎不需要相关储备知识，这一点稍后还会详述。不管是用什么编程语言，只要是曾经写过简单的程序，对编程有一些感觉，就已经足够了（即使没有任何编程经验，应该也能看懂），因为这本书主要就是面向初学者的。书中虽然有很多 C 语言程序，但实际上并没有用到很高深的 C 语言知识，所以就算是曾经因为 C 语言太难而中途放弃的人也不用担心看不懂。当然，如果具备相关知识的话，理解起来会相对容易一些，不过即使没有相关知识也没关系，书中的说明都很仔细，大家可以放心。

本书以 IBM PC/AT 兼容机（也就是所谓的 Windows 个人电脑）为对象进行说明。

2　何谓操作系统

3　开发操作系统的各种方法

4　无知则无畏

当我们打算开发操作系统时，总会有人从旁边跳出来，罗列出一大堆专业术语，问这问那，像内核怎么做啦，外壳怎么做啦，是不是单片啦，是不是微内核啦，等等。虽然有时候提这些问题也是有益的，但一上来就问这些，当然会让人无从回答。

要想给他们一个满意答复，让他们不再从旁指手画脚的话，还真得多学习，拿出点像模像样的见解才行。但我们是初学者，没有必要去学那些麻烦的东西，费时费力且不说，当我们知道现有操作系统在各方面都考虑得如此周密的时候，就会发现自己的想法太过简单而备受打击没了干劲。如果被前人的成果吓倒，只用这些现有的技术来做些拼拼凑凑的工作，岂不是太没意思了。

所以我们这次不去学习那些复杂的东西，直接着手开发。就算知道一大堆专业术语、专业理论，又有什么意思呢？还不如动手去做，就算做出来的东西再简单，起码也是自己的成果。而且自己先实际操作一次，通过实践找到其中的问题，再来看看是不是已经有了这些问题的解决方案，这样下来更能深刻地理解那些复杂理论。不管怎么说，反正目前我们也无法回答那些五花八门的问题，倒不如直接告诉在一旁指手画脚的人们：我们就是想用自己的方法做自己喜欢的事情，如果要讨论高深的问题，就另请高明吧。

作者苦口婆心地说了这么多就是希望如果你想开发个操作系统，就动手去写吧，到底自己重写个操作系统有什么用倒可以先放着。

如果你到现在还对要不要读这本书，或者读这本书的期望的收获有疑问，推荐你阅读豆瓣上本书的一篇评论[1]后再自行决定。

> 这本书对基础知识要求不高，懂点 C 语言和 CPU 基本知识就可以了，适合初学者。要是奔着了解操作系统原理或内核的期望，就不适宜读这本书了。30 天后也许你真的可以向作者那样做出一个基本的系统模型，但这并不意味着你对内存管理、进程管理、设备管理有着怎样高深的认识。读这本书之前先弄清自己的定位吧，毕竟时间宝贵。

5　如何开发操作系统

6　操作系统开发中的困难

7　学习本书时的注意事项（重要!）

8　各章内容摘要

[1] http://book.douban.com/review/5606888/

第 1 天　从计算机结构到汇编程序入门

1　先动手操作

随书附带了光盘[1]，给出了书中的全部示例程序，以及部分用到的工具。

> 打开附带光盘，里面有一个名为 tolset 的文件夹，把这个文件夹复制到硬盘的任意一个位置上。现在里面的东西还不多，只有 3MB 左右，不过以后我们自己开发的软件也都要放到这个文件夹里，所以往后它会越来越大，因此硬盘上最好留出 100MB 左右的剩余空间。工具安装到此结束，我们既不用修改注册表，也不用设定路径参数，就这么简单。而且以后不管什么时候，都可以把这整个文件夹移动到任何其他地方。用这些工具，我们不仅可以开发操作系统，还可以开发简单的 Windows 应用程序或 OSASK 应用程序等。

示例程序在附带光盘中名为 projects 的目录下，只要需要的示例程序目录复制到 tolset 文件夹里，就可以正常运行示例程序了。

考虑到开发中使用真实的软盘很不方便，作者特意准备了一个模拟器。

[1] 下载链接：http://pan.baidu.com/share/link?shareid=541099&uk=3657658273 或自行搜索“30 天自制操作系统.iso”

我们有了这个模拟器，不用软盘，也不用终止 Windows，就可以确认所开发的操作系统启动以后的动作，很方便呢。

使用模拟器的方法也非常简单，我们只需要在用!cons_nt.bat[2]（或者是!cons_9x.bat）打开的命令行窗口中输入“run”指令就可以了。然后一个名叫 QEMU 的非常优秀的免费 PC 模拟器就会自动运行。

§

在这一节中，作者使用二进制编辑器（十六进制编辑器）做了一个 helloos.img 文件出来。先输入了一些内容，并把内容保存成软盘映像文件格式，将这个文件写入软盘，并用它来启动电脑。画面上会显示出“hello, world”这个字符串。如果有兴趣，希望自己尝试，请参考书中这一小节的内容。

2　究竟做了些什么

简单解释了为什么上一节可以用二进制来写一个所谓的操作系统（虽然只能显示“hello, world”这个字符串）。

3　初次体验汇编程序

好，现在就让我们马上来写一个汇编程序，用它来生成一个跟刚才完全一样的 helloos.img 吧。我们这次使用的汇编语言编译器是笔者自己开发的，名为“nask”，其中的很多语法都模仿了自由软件里

[2]要根据 Windows 的版本决定用哪一个。后缀为 9x 代表是 Windows 9X 系统，后缀为 nt 的代表使用 NT 架构的 Windows 系统，如 Windows XP 及其以后的版本，以后默认为运行 cons_nt.bat，并在后面将其简写为!cons。

PS：作者开发这个系统是 2000 年左右的事情，写书的时间也比较早，所以考虑了这些问题，可以理解哈！

享有盛名的汇编器“NASM”，不过在“NASM”的基础之上又提高了自动优化能力。

†[3] projects\01_day\helloos1\

helloos.nas

```
1     DB    0xeb, 0x4e, 0x90, 0x48, 0x45, 0x4c, 0x4c, 0x4f
2     DB    0x49, 0x50, 0x4c, 0x00, 0x02, 0x01, 0x01, 0x00
3     DB    0x02, 0xe0, 0x00, 0x40, 0x0b, 0xf0, 0x09, 0x00
4     DB    0x12, 0x00, 0x02, 0x00, 0x00, 0x00, 0x00, 0x00
5     DB    0x40, 0x0b, 0x00, 0x00, 0x00, 0x00, 0x29, 0xff
6     DB    0xff, 0xff, 0xff, 0x48, 0x45, 0x4c, 0x4c, 0x4f
7     DB    0x2d, 0x4f, 0x53, 0x20, 0x20, 0x20, 0x46, 0x41
8     DB    0x54, 0x31, 0x32, 0x20, 0x20, 0x20, 0x00, 0x00
9     RESB    16
10    DB    0xb8, 0x00, 0x00, 0x8e, 0xd0, 0xbc, 0x00, 0x7c
11    DB    0x8e, 0xd8, 0x8e, 0xc0, 0xbe, 0x74, 0x7c, 0x8a
12    DB    0x04, 0x83, 0xc6, 0x01, 0x3c, 0x00, 0x74, 0x09
13    DB    0xb4, 0x0e, 0xbb, 0x0f, 0x00, 0xcd, 0x10, 0xeb
14    DB    0xee, 0xf4, 0xeb, 0xfd, 0x0a, 0x0a, 0x68, 0x65
15    DB    0x6c, 0x6c, 0x6f, 0x2c, 0x20, 0x77, 0x6f, 0x72
16    DB    0x6c, 0x64, 0x0a, 0x00, 0x00, 0x00, 0x00, 0x00
17    RESB    368
```

[3]源代码在随书光盘（见第1节的说明）中的路径; 另外，为了便于查看，给代码中添加了行号，导致复制文中代码不是很方便，请直接使用光盘中的源代码或手动输入。

```
18      DB      0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x55, 0xaa
19      DB      0xf0, 0xff, 0xff, 0x00, 0x00, 0x00, 0x00, 0x00
20      RESB    4600
21      DB      0xf0, 0xff, 0xff, 0x00, 0x00, 0x00, 0x00, 0x00
22      RESB    1469432
```

把 helloos1 文件夹复制粘帖到 tolset 文件夹里，我们只要在用“!cons”打开的命令行窗口里输入“asm”，就可以生成 helloos.img 文件。在用“asm”作成 img 文件后，再执行“run”指令，就可以得到与刚才一样的结果。

§

DB 指令是“data byte”的缩写，也就是往文件里直接写入 1 个字节的指令。

RESB 指令是“reserve byte”的略写，如果想要从现在的地址开始空出 10 个字节来，就可以写成 RESB 10，意思是我们预约了这 10 个字节（大家可以想象成在对号入座的火车里，预订了 10 个连号座位的情形）。而且 nask 不仅仅是把指定的地址空出来，它还会在空出来的地址上自动填入 0x00，所以我们这次用这个指令就可以输出很多的 0x00，省得我们自己去写 18 万行程序了，真是帮了个大忙。

这里还要说一下，数字的前面加上 0x，就成了十六进制数，不加 0x，就是十进制数。这一点跟 C 语言是一样的。

4　加工润色

刚才我们把程序变成了短短的 22 行，这成果令人欣喜。不过还有一点不足就是很难看出这些程序是干什么的，所以我们下面就来稍微改写一下，让别人也能看懂。

†projects\01_day\helloos2

helloos.nas

```
; hello-os
; TAB=4

; 以下这段是标准 FAT12 格式软盘专用的代码

        DB      0xeb, 0x4e, 0x90
        DB      "HELLOIPL"      ; 启动区的名称可以是任意的字符串（8 字节）
        DW      512             ; 每个扇区（sector）的大小（必须为 512 字节）
        DB      1               ; 簇（cluster）的大小（必须为 1 个扇区）
        DW      1           ; FAT 的起始位置（一般从第一个扇区开始）
        DB      2           ; FAT 的个数（必须为 2）
        DW      224         ; 根目录的大小（一般设成 224 项）
        DW      2880        ; 该磁盘的大小（必须是 2880 扇区）
        DB      0xf0        ; 磁盘的种类（必须是 0xf0）
        DW      9           ; FAT 的长度（必须是 9 扇区）
        DW      18          ; 1 个磁道（track）有几个扇区（必须是 18）
        DW      2           ; 磁头数（必须是 2）
        DD      0           ; 不使用分区，必须是 0
        DD      2880        ; 重写一次磁盘大小
        DB      0,0,0x29        ; 意义不明，固定
```

```
21          DD      0xffffffff      ;（可能是）卷标号码
22          DB      "HELLO-OS   "   ; 磁盘的名称（11 字节）
23          DB      "FAT12   "      ; 磁盘格式名称（8 字节）
24          RESB    18              ; 先空出 18 字节
25
26  ; 程序主体
27          DB      0xb8, 0x00, 0x00, 0x8e, 0xd0, 0xbc, 0x00, 0x7c
28          DB      0x8e, 0xd8, 0x8e, 0xc0, 0xbe, 0x74, 0x7c, 0x8a
29          DB      0x04, 0x83, 0xc6, 0x01, 0x3c, 0x00, 0x74, 0x09
30          DB      0xb4, 0x0e, 0xbb, 0x0f, 0x00, 0xcd, 0x10, 0xeb
31          DB      0xee, 0xf4, 0xeb, 0xfd
32
33  ; 信息显示部分
34
35          DB      0x0a, 0x0a      ; 2 个换行
36          DB      "hello, world"
37          DB      0x0a            ; 换行
38          DB      0
39
40          RESB    0x1fe-$         ; 填写 0x00, 直到 0x001fe
41          DB      0x55, 0xaa
42
```

```
43 ; 以下是启动区以外部分的输出
44
45         DB      0xf0, 0xff, 0xff, 0x00, 0x00, 0x00, 0x00, 0x00
46         RESB    4600
47         DB      0xf0, 0xff, 0xff, 0x00, 0x00, 0x00, 0x00, 0x00
48         RESB    1469432
```

首先是“;”命令，这是个注释命令。

其次是 DB 指令的新用法。我们居然可以直接用它写字符串。在写字符串的时候，汇编语言会自动地查找字符串中每一个字符所对应的编码，然后把它们一个字节一个字节地排列起来。这个功能非常方便，也就是说，当我们想要变更输出信息的时候，就再也不用自己去查字符编码表了。

再有就是 DW 指令和 DD 指令，它们分别是“data word”和“data double-word”的缩写，是 DB 指令的“堂兄弟”。word 的本意是“单词”，但在计算机汇编语言的世界里，word 指的是“16 位”的意思，也就是 2 个字节。“double-word”是“32 位”的意思，也就是 4 个字节。对了，差点忘记说 RESB 0x1fe-$ 了。这个美元符号的意思如果不讲，恐怕谁也搞不明白，它是一个变量，可以告诉我们这一行现在的字节数（如果严格来说，有时候它还会有别的意思，关于这一点我们明天再讲）。在这个程序里，我们已经在前面输出了 132 字节，所以这里的 $ 就是 132。因此 nask 先用 0x1fe 减去 132，得出 378 这一结果，然后连续输出 378 个字节的 0x00。

那这里我们为什么不直接写 378，而非要用 $ 呢？这是因为如果将显示信息从“hello, world”变成“this is a pen.”的话，中间要输出 0x00 的字节数也会随之变化。换句话说，我们必须保证软盘的第 510 字节（即第 0x1fe 字节）开始的地方是 55 AA。如果在程序里使用美元符号（$）的话，汇编语言会自动计算需要输出多少个 00，我们也就可以很轻松地改写输出信息了。

第 2 天　汇编语言学习与 Makefile 入门

1　介绍文本编辑器

如中文译者所注，推荐使用 Notepad++ 文本编辑器。

使用这个软件打开光盘中提供的源代码会出现日语注释乱码，选择 格式 –> 编码字符集 –> 日文 –>Shift-JIS 即可正常显示代码中原书作者的日语注释。但是，关闭文件之后重新打开又会恢复原状，解决方法为在选择 Shift-JIS 编码后复制内容到一个新的文件中去，保存替换原先的文件即可，需要提示的是，新建的文件要使用 ANSI 编码格式保存，不能是 UTF-8，否则会导致后面编译时报错。

原书作者推荐的文本编辑器 TeraPad 在中文系统中会出现软件本身的文本乱码，如菜单栏工具栏的文字，但是无需设置就可以正常显示代码中的日语注释。

2　继续开发

选讲程序的核心部分，核心程序之前和启动区以外的内容需要具备软盘方面的相关知识，后面讲。

†projects\02_day\helloos3

helloos.nas

```
; hello-os
; TAB=4

        ORG         0x7c00              ; 指明程序的装载地址

; 以下的记述用于标准 FAT12 格式软盘

        JMP         entry
        DB         0x90
---中略---
; 程序核心

entry:
        MOV         AX,0                ; 初始化寄存器
        MOV         SS,AX
        MOV         SP,0x7c00
        MOV         DS,AX
        MOV         ES,AX

        MOV         SI,msg
putloop:
```

```
22          MOV         AL,[SI]
23          ADD         SI,1              ; 给 SI 加 1
24          CMP         AL,0
25          JE          fin
26          MOV         AH,0x0e             ; 显示一个文字
27          MOV         BX,15              ; 指定字符颜色
28          INT         0x10              ; 调用显卡 BIOS
29          JMP         putloop
30  fin:
31          HLT                           ; 让 CPU 停止，等待指令
32          JMP         fin                 ; 无限循环
33
34  msg:
35          DB          0x0a, 0x0a         ; 换行两次
36          DB          "hello, world"
37          DB          0x0a              ; 换行
38          DB          0
```

§

ORG 指令：程序从指定的这个地址开始，也就是把程序装载到内存中的指定地址。这里是 0x7c00。

JMP 指令：无条件跳转。配合下面的标签"entry" 等，可指定跳转的目的地。

“entry”：标签声明，用于指定 JMP 指令的跳转地址。

MOV 指令：赋值。“MOV AX,0”相当于“AX=0”这样一个赋值语句。同样，“MOV SS,AX”相当于“SS=AX”。

§

相关寄存器：

16 位寄存器：AX、CX、DX、BX、SP、BP、SI、DI；

其中，前四个的高低 8 位可当 8 位寄存器用，AL、CL、DL、BL、AH、CH、DH、BH；

32 位寄存器：16 位寄存器可以扩展为 32 位寄存器，EAX、ECX、EDX、EBX、ESP、EBP、ESI、EDI；

段寄存器：ES、CS、SS、DS、FS、GS；

§

“MOV SI,msg”：这里是可以“将标号赋值给寄存器”。在汇编语言中，所有的标号都仅仅是单纯的数字。每个标号对应的数字，是由汇编器根据 ORG 指令计算出来的。编译器计算出的“标号的地方对应的内存地址”就是那个标号的值。这里，msg 的地址是 0x7c74，所以这个指令就是把 0x7c74 带入到 SI 寄存器去。

“MOV AL,[SI]”：MOV 指令的数据传送源和传送目的地不仅可以是寄存器或常数，也可以是内存地址。这个时候，我们使用方括号 ([]) 来表示内存地址。另外，可以用来指定内存地址的寄存器只有 BX、BP、SI、DI 这几个。

可以用下面指令将 SI 地址的 1 字节内容读入到 AL：

```
MOV AL, BYTE [SI]
```

由于 MOV 指令的规则，即源数据和目的数据必须位数相同，也就是向 AL 里代入的只能是 BYTE，这样以来就可以省略 BYTE，即可以写成：

```
MOV AL, [SI]
```

§

ADD 指令是加法指令。若以 C 语言的形式改写“ADD SI,1”的话，就是“SI=SI=1”。

CMP 是比较指令。“CMP AL,0”，是将 AL 中的值与 0 进行比较。

JE 是条件跳转指令之一。所谓的条件跳转指令，就是根据比较的结果决定跳转或者不跳转。就 JE 指令而言，如果比较结果相等，则跳转到指定的地址；而如果比较结果不等，则不跳转，继续执行下一条指令。

```
CMP AL, 0
JE fin
```

这两条指令相当于：

```
if(AL == 0){ goto fin;}
```

§

INT 是软件中断指令。在这里是调用显卡 BIOS，不解释。

§

HLT 指令，让 CPU 停止动作，进入待机状态。

§

用 C 语言改写 helloos.nas 程序节选。

```
1 entry:
2     AX = 0;
3     SS = AX;
```

```
4     SP = 0x7c00;
5     DS = AX;
6     ES = AX;
7     SI = msg;
8  putloop:
9     AL = BYTE [SI];
10    SI = SI+1;
11    if (AL ==0 ){goto fin;}
12    AH = 0x0e;
13    BX = 15;
14    INT 0x10;
15    goto putloop;
16 fin:
17    HLT;
18    goto fin;
```

就是有了这个程序，我们才能把 msg 里写的数据，一个字符一个字符地显示出来，并且数据变成 0 以后，HLT 指令就会让程序进入无限循环，“hello,world”就是这样显示出来的。

§

程序中的 ORG 后地址 0x7c00 是因为目前约定内存的0x00007c00-0x00007dff 地址为启动区内容的装载地址，不能随便改成其它地址。

3　先制作启动区

考虑到以后的开发，不要一下子用 nask 来制作整个磁盘映像，而是先用它来制作 512 字节的启动区，剩下的部分我们用磁盘映像管理工具来做。

先把 helloos.nas 的后半部分截掉，这是因为启动区只需要最后的 512 字节。现在这个程序仅仅用来制作启动区，所以把文件名改为 ipl.nas。

然后改造 asm.bat，将输出的文件名改成 ipl.bin。另外，也顺便输出列表文件 ipl.lst。这是一个文本文件，可以用来简单地确认每个指令是怎么翻译成机器语言的。

另外还增加了一个 makeimg.bat 文件。它是以 ipl.bin 为基础，制作磁盘映像文件 helloos.img 的批处理文件。利用作者开发的磁盘映像管理工具 edimg.exe，先读入一个空白的磁盘映像文件，然后在开头写入 ipl.bin，最后输出名为 helloos.img 的磁盘映像文件。

这样，从编译到测试的步骤为双击!cons，然后在命令行窗口中按顺序输入 asm –>makeimg –>run 这 3 个命令。

下一节有更简单的编译方式。

4　Makefile 入门

作者编写了一个 Makefile 文件，这样就可以方便的生成所需要的文件。

†projects\02_day\helloos5

Makefile

```
1
2 # デフォルト動作
```

```
3
4 default :
5     ../z_tools/make.exe img
6
7 # ファイル生成規則
8
9 ipl.bin : ipl.nas Makefile
10    ../z_tools/nask.exe ipl.nas ipl.bin ipl.lst
11
12 helloos.img : ipl.bin Makefile
13    ../z_tools/edimg.exe   imgin:../z_tools/fdimg0at.tek \
14        wbinimg src:ipl.bin len:512 from:0 to:0   imgout:helloos.img
15
16 # コマンド
17
18 asm :
19    ../z_tools/make.exe -r ipl.bin
20
21 img :
22    ../z_tools/make.exe -r helloos.img
23
24 run :
```

```
25      ../z_tools/make.exe img
26      copy helloos.img ..\z_tools\qemu\fdimage0.bin
27      ../z_tools/make.exe -C ../z_tools/qemu
28
29  install :
30      ../z_tools/make.exe img
31      ../z_tools/imgtol.com w a: helloos.img
32
33  clean :
34      -del ipl.bin
35      -del ipl.lst
36
37  src_only :
38      ../z_tools/make.exe clean
39      -del helloos.img
```

使用方法为：用!cons 打开命令行窗口，然后就可以通过输入 `make img` 来生成映像文件，输入 `make run` 来运行，等等。可以使用的参数在 Makefile 文件中写明了。另外，当执行不带参数的 make 命令时，相当于执行 `make img`。

第 3 天　进入 32 位模式并导入 C 语言

作者给开发的操作系统起名字叫 纸娃娃操作系统——haribote os。

1　制作真正的 IPL

制作一个可以称为真正的 IPL（启动程序装载器），让启动区真正的开始装载程序。

§

因为磁盘最初的 512 字节是启动区，所以要装载下一个 512 字节的内容。程序是在上一天的基础上修改的，添加了以下内容：

†projects\03_day\harib00a

ipl.nas 本次添加的部分

```
1        MOV     AX,0x0820
2        MOV     ES,AX
3        MOV     CH,0            ; シリンダ 0
```

```
4       MOV     DH,0            ; ヘッド 0
5       MOV     CL,2            ; セクタ 2
6
7       MOV     AH,0x02           ; AH=0x02 : ディスク読み込み
8       MOV     AL,1            ; 1 セクタ
9       MOV     BX,0
10      MOV     DL,0x00           ; A ドライブ
11      INT     0x13            ; ディスク BIOS 呼び出し
12      JC      error
```

INT 0x13 是调用 BIOS 的 0x13 号函数。
下面是 BIOS 13 中断的简单说明（功能有磁盘的读、写、扇区校验、寻道）

- AH=0x02 读盘/0x03 写盘/0x04 校验/0x0c 寻道
- AL= 处理连续扇区数
- CH= 柱面号 &0xff
- CL= 扇区号（0∼5 位）|（柱面号 &0x300)≫2
- DH= 磁头号
- DL= 驱动器号
- ES:BX= 缓冲地址

返回值：FLAGS.CF=0，没有错误 AH=0；FLAGS=1 有错误，AH 保存错误码

这里，JC（jump if carry）是条件跳转指令，如果进位标志为 1，就跳转。跳转条件看调用函数的返回值 FLAG.CF。

对照程序和 BIOS 函数参数说明，可以知道我们这次使用的是读盘，柱面号是 0，磁头号是 0，扇区号是 2，磁盘号是 0。

§

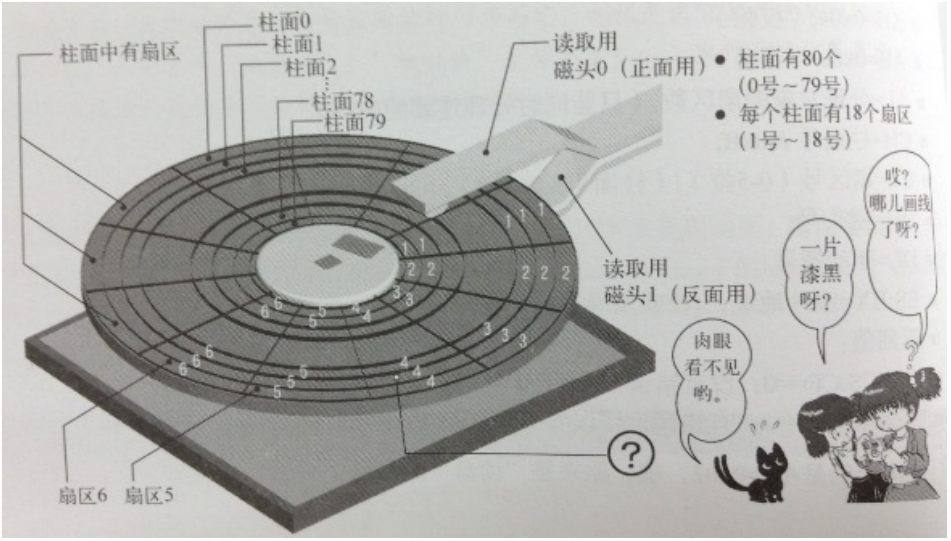

图 3.1: 软盘的结构

一张软盘有 80 个柱面，2 个磁头，18 个扇区，且一个扇区有 512 字节。所以一张软盘的容量是 $80 \times 2 \times 18 \times 512 = 1474560Byte = 1440KB$。

含有 IPL 的启动区，位于 C0-H0-S1（柱面 0，磁头 0，扇区 1），下一个扇区是 C0-H0-S2，这次我们要装载的扇区就是这个。

ES:BX= 缓冲地址 是个内存地址，表明要把软盘上读出的数据装载到内存的哪个位置。由于一个 BX 只能表示 0∼0xffff 的值，即 64K，太小了，使用段寄存器以 ES:BX 这种方式来表示地址，写成“MOV AL,[ES:BX]”，代表“ES×16+BX”的内存地址。程序中指定了 ES=0x0820，BX=0，所以软盘的数据会被装载到 0x8200∼0x83ff 的位置。

§

作者使用变量的方式改写了 Makefile 文件，可以看一下。

2　试错

鉴于软盘的不可靠性，有时候需要在软盘读不出来的时候多读几次，这里重读 5 次。

†projects\03_day\harib00b

ipl.nas 本次添加的部分

```
; ディスクを読む

        MOV     AX,0x0820
        MOV     ES,AX
        MOV     CH,0            ; シリンダ 0
        MOV     DH,0            ; ヘッド 0
        MOV     CL,2            ; セクタ 2
```

```
8
9           MOV         SI,0            ; 失敗回数を数えるレジスタ
10 retry:
11          MOV         AH,0x02           ; AH=0x02 : ディスク読み込み
12          MOV         AL,1            ; 1 セクタ
13          MOV         BX,0
14          MOV         DL,0x00           ; A ドライブ
15          INT         0x13            ; ディスク BIOS 呼び出し
16          JNC         fin               ; エラーがおきなければ fin へ
17          ADD         SI,1            ; SI に 1 を足す
18          CMP         SI,5            ; SI と 5 を比較
19          JAE         error            ; SI >= 5 だったら error へ
20          MOV         AH,0x00
21          MOV         DL,0x00           ; A ドライブ
22          INT         0x13            ; ドライブのリセット
23          JMP         retry
```

其中，JNC（jump if not carry），进位标志为 0 的话跳转；JAE（jump if above or equal），大于或等于时跳转。

在重新读盘之前，做了如下处理，AH=0x00，DL=0x00，INT=0x13，即完成系统复位。

3　读到 18 扇区

往后多读几个扇区，读完柱面 0 的 18 个扇区。

†projects\03_day\harib00c

ipl.nas 本次添加的部分

```
; ディスクを読む

        MOV     AX,0x0820
        MOV     ES,AX
        MOV     CH,0            ; シリンダ 0
        MOV     DH,0            ; ヘッド 0
        MOV     CL,2            ; セクタ 2
readloop:
        MOV     SI,0            ; 失敗回数を数えるレジスタ
retry:
        MOV     AH,0x02           ; AH=0x02 : ディスク読み込み
        MOV     AL,1            ; 1 セクタ
        MOV     BX,0
        MOV     DL,0x00           ; A ドライブ
        INT     0x13            ; ディスク BIOS 呼び出し
        JNC     next            ; エラーがおきなければ next へ
        ADD     SI,1            ; SI に 1 を足す
```

```
18          CMP         SI,5            ; SI と 5 を比較
19          JAE         error           ; SI >= 5 だったら error へ
20          MOV         AH,0x00
21          MOV         DL,0x00           ; A ドライブ
22          INT         0x13            ; ドライブのリセット
23          JMP         retry
24  next:
25          MOV         AX,ES           ; アドレスを 0x200 進める
26          ADD         AX,0x0020
27          MOV         ES,AX           ; ADD ES,0x020 という命令がないのでこうしている
28          ADD         CL,1            ; CL に 1 を足す
29          CMP         CL,18           ; CL と 18 を比較
30          JBE         readloop        ; CL <= 18 だったら readloop へ
```

JBE(jump if below or equal)，小于等于则跳转。

要读入下一个扇区只需给 CL 加 1，给 ES 加上 0x20（512/16）。CL 是扇区号，ES 指定读入地址。

这里使用循环的方式读入各个扇区，而不是在开始的时候设置读入的扇区数 AL=17，因为磁盘 BIOS 读盘函数有一些“补充说明”：

> 指定处理的扇区数，范围在 0x01 0xff（指定 0x02 以上的数值时，要特别注意能够连续处理多个扇区的条件。如果是 FD 的话，似乎不能跨越多个磁道，也不能超过 64KB 的界限。）

经过上面这些读盘处理，已经把磁盘上 C0-H0-S2 到 C0-H0-S18 的 $512 \times 17 = 8704$ 个字节的内容，装载到了内存的 0x8200∼0xa3ff。

4　读入 10 个柱面

C0-H0-S18 扇区的下一个扇区是磁盘反面的 C0-H1-S1，按顺序读到 C0-H1-S18，接着读 C1-H0-S1，最后一直读到 C9-H1-S18。

†projects\03_day\harib00d

```
; ディスクを読む

        MOV     AX,0x0820
        MOV     ES,AX
        MOV     CH,0            ; シリンダ 0
        MOV     DH,0            ; ヘッド 0
        MOV     CL,2            ; セクタ 2
readloop:
        MOV     SI,0            ; 失敗回数を数えるレジスタ
retry:
        MOV     AH,0x02           ; AH=0x02 : ディスク読み込み
        MOV     AL,1            ; 1 セクタ
        MOV     BX,0
        MOV     DL,0x00           ; A ドライブ
        INT     0x13            ; ディスク BIOS 呼び出し
        JNC     next            ; エラーがおきなければ next へ
```

```
17          ADD         SI,1                ; SI に 1 を足す
18          CMP         SI,5                ; SI と 5 を比較
19          JAE         error                ; SI >= 5 だったら error へ
20          MOV         AH,0x00
21          MOV         DL,0x00               ; A ドライブ
22          INT         0x13                ; ドライブのリセット
23          JMP         retry
24  next:
25          MOV         AX,ES                ; アドレスを 0x200 進める
26          ADD         AX,0x0020
27          MOV         ES,AX                ; ADD ES,0x020 という命令がないのでこうしている
28          ADD         CL,1                ; CL に 1 を足す
29          CMP         CL,18                ; CL と 18 を比較
30          JBE         readloop            ; CL <= 18 だったら readloop へ
31          MOV         CL,1
32          ADD         DH,1
33          CMP         DH,2
34          JB          readloop            ; DH < 2 だったら readloop へ
35          MOV         DH,0
36          ADD         CH,1
37          CMP         CH,CYLS
38          JB          readloop            ; CH < CYLS だったら readloop へ
```

JB(jump if below), 如果小于就跳转。

在程序开头使用了 EQU 指令来声明常数，即`CYLS EQU 10`。现在已经能够把软盘最初的 $10 \times 2 \times 18 \times 512 = 184320byte = 180KB$ 的内容完整的装载到内存中了。

5　着手开发操作系统

编写一个短小的程序，只让它 HLT。

`projects\03_day\harib00e`

haribote.nas

```
fin:
    HLT
    JMP fin
```

使用 nask 编译，输出成 harbote.sys。用“make img”指令来生成映像文件。

使用作者最开始提到的二进制编辑器查看 haribote.img 和 haribote.sys 内容，可以发现 0x002600 附近保存着文件名，0x004200 位置保存着文件的内容，这里分别是 haribotesys 和编译后的 harbote.sys 里面的内容“`F4 EB FD`”。

这样，要做的工作就是将操作系统的本身的内容写到名为 harbote.sys 的问卷中，再把它保存到磁盘映像里，然后从启动区执行这个 harbote.sys 就行了。

6　从启动区执行操作系统

现在的程序是从启动区开始，把磁盘上的内容装载到内存 0x8000 号地址，所以磁盘映像上位于 0x004200 号地址的程序位于内存的 0x8000+0x4200=0xc200 号地址。

修改 haribote.nas，加上 ORG 0xc200，然后在 ipl.nas 处理的最后加上 JMP 0xc200 这个指令。详细程序见“projects/03_day/harib00f”。

通过运行“make run”来运行程序。

7　确认操作系统的执行情况

通过切换以下画面模式，让画面变成一片漆黑，证明程序正常运行。

†projects/03_day/harib00g

```
1 ; haribote-os
2 ; TAB=4
3
4         ORG         0xc200              ; このプログラムがどこに読み込まれるのか
5
6         MOV         AL,0x13              ; VGA グラフィックス、320x200x8bit カラー
7         MOV         AH,0x00
8         INT         0x10
9 fin:
```

```
10          HLT
11          JMP         fin
```

设置显卡模式：

- AH=0x00
- AL= 模式：
  - 0x03：16 色字符模式，80×25
  - 0x12：VGA 图形模式，640×480×4 位彩色模式，独特的 4 面存储模式
  - 0x13：VGA 图形模式，320×200×8 位彩色模式，调色板模式
  - 扩展 VGA 图形模式，800×600×4 位彩色模式，独特的 4 面存储模式
- 返回值：无

程序的变动：将 ipl.nas 改名为 ipl10.nas，提醒这个程序只能读入 10 个柱面。

想把磁盘装载内容的结束地址告诉给 haribote.sys，在 ipl10.nas 文件中“JMP 0xc200”之前加入了一行代码，将 CYLS 的值写到内存地址 0x0ff0 中。

运行“make run”查看效果，应该是一片漆黑画面。

8　32 位模式前期准备[1]

考虑到系统以后会支持各种不同的画面模式，就需要把现在的设置信息（BOOT_INFO）保存起来以备后用。

†projects\03_day\harib00h

haribote.nas

```
; haribote-os
; TAB=4

; BOOT_INFO 関係
CYLS    EQU     0x0ff0          ; ブートセクタが設定する
LEDS    EQU     0x0ff1
VMODE    EQU     0x0ff2          ; 色数に関する情報。何ビットカラーか?
SCRNX    EQU     0x0ff4          ; 解像度の X
SCRNY    EQU     0x0ff6          ; 解像度の Y
VRAM    EQU     0x0ff8          ; グラフィックバッファの開始番地

        ORG     0xc200          ; このプログラムがどこに読み込まれるのか

        MOV     AL,0x13          ; VGA グラフィックス、320x200x8bit カラー
        MOV     AH,0x00
```

[1]书中作者先讲述了为什么用 32 位模式，自己看下书吧。

```
16          INT         0x10
17          MOV         BYTE [VMODE],8      ; 画面モードをメモする
18          MOV         WORD [SCRNX],320
19          MOV         WORD [SCRNY],200
20          MOV         DWORD [VRAM],0x000a0000
21
22  ; キーボードの LED 状態を BIOS に教えてもらう
23
24          MOV         AH,0x02
25          INT         0x16                ; keyboard BIOS
26          MOV         [LEDS],AL
27
28  fin:
29          HLT
30          JMP         fin
```

[VRAM] 里保存的是 0xa0000。VRAM 是显卡内存，它的各个地址对应画面上的像素。不同的画面模式对应不同的 VRAM，因此这里将使用的 VRAM 地址保存在 BOOT_INFO 里。这种画面模式下 VRAM 是“0xa000~0xaffff 的 64KB”。画面的像素数、颜色数以及从 BIOS 取得的键盘信息都保存在内存 0x0ff0 位置附近。

9　开始导入 C 语言

现在，直接切换到 32 位模式，然后运行 C 语言写程序。

程序做了很大的改动，haribote.sys 的前半部分使用汇编语言编写的，后半部分是用 C 语言编写的，所以将 haribote.nas 改成了 asmhead.nas，并且，为了调用 C 语言写的程序，添加了 100 行左右的汇编代码。这些汇编代码作者在后面再讲解，这里直接跳过，分析 C 语言部分。

C 语言部分写在 bootpack.c 文件中。

†projects\03_day\harib00i

bootpack.c

```
1 void HariMain(void)
2 {
3
4 fin:
5     /* ここに HLT を入れたいのだが、C 言語では HLT が使えない！ */
6     goto fin;
7
8 }
```

§

bootpack.c 变成机器语言的过程：

1. 使用 ccl.exe 从 bootpack.c 生成 bootpack.gas；

2. 使用 gas2nask.exe 从 bootpack.gas 生成 bootpack.nas；
3. 使用 nask.exe 从 bootpack.nas 生成 bootpack.obj；
4. 使用 obj2bim.exe 从 bootpack.obj 生成 bootpack.bim；
5. 使用 bim2hrb.exe 从 bootpack.bim 生成 bootpack.hrb；
6. 使用 copy 指令将 asmhead.bin 与 bootpack.hrb 单纯结合起来就生成了 haribote.sys。

10　实现 HLT（harib00j）

†projects\03_day\harib00j

naskfun.nas

```
1  ; naskfunc
2  ; TAB=4
3
4  [FORMAT "WCOFF"]                ; オブジェクトファイルを作るモード
5  [BITS 32]                        ; 32 ビットモード用の機械語を作らせる
6
7
8  ; オブジェクトファイルのための情報
9
10 [FILE "naskfunc.nas"]            ; ソースファイル名情報
```

```
11
12          GLOBAL    _io_hlt             ; このプログラムに含まれる関数名
13
14
15 ; 以下は実際の関数
16
17 [SECTION .text]          ; オブジェクトファイルではこれを書いてからプログラムを書く
18
19 _io_hlt:     ; void io_hlt(void);
20          HLT
21          RET
```

使用汇编语言编写了一个函数，io_hlt。将输出设置为 WCOFF 模式，可以编译成目标文件，与 bootpack.obj 链接。

在 nask 目标文件的模式下，必须设定文件名信息，然后再写明下面程序的函数名。需要先在函数名前面加上“_”，否则不能很好地与 C 语言函数链接。需要链接的函数名都需要 GLOBAL 指令声明。

下面写一个实际的函数。先写一个与用 GLOBAL 声明的函数名相同的标号，从此处开始写代码就可以了。

†projects\03_day\harib00j

bootpack.c

```
1 /* 他のファイルで作った関数がありますと C コンパイラに教える */
2
3 void io_hlt(void);
4
```

```
5 /* 関数宣言なのに、{} がなくていきなり; を書くと、
6     他のファイルにあるからよろしくね、という意味になるのです。 */
7
8 void HariMain(void)
9 {
10
11 fin:
12     io_hlt(); /* これで naskfunc.nas の _io_hlt が実行されます */
13     goto fin;
14
15 }
```

“make run”运行程序。