上篇

本书分上下两篇。上篇的内容与本书第一版（也即《自己动手写操作系统》）基本一致，共分七章。

第 1 章是个简单的开头，是我们操作系统之旅的第一步。虽然简单，但它的意义却非同小可，希望读者读完之后能够明白，一项看似繁重的工作，完全可以分解成相对容易的若干小步，而一旦你开始了，那么剩下的只不过是一点耐心而已。

第 2 章和第 3 章是准备工作，这是理论型的书籍最容易忽视的部分，然而它的重要性却不容忽视。再高明的厨师也需要有张案板，有个锅灶。编写自己操作系统的热情，决不应该被准备阶段的困难所浇灭。所以笔者用了不少的篇幅来介绍如何搭建自己的工作环境，以及讲述 Intel CPU 保护模式的基本概念和原理。当然，如果读者已经熟悉其中的内容，可以有选择性地跳过。

第 4 章和第 5 章介绍了如何写一个可用的引导扇区和用以加载内核的 Loader。这些内容也是传统操作系统书籍容易忽视的，因为引导扇区和内核加载器（Loader）严格来讲并不能算是操作系统的一部分。然而一个火箭不能没有发射架，要完成自己的操作系统，这两章内容必不可少。希望在读完这两章之后，读者能够彻底弄明白一个静态的内核是如何转变成一个运转中的操作系统的。

第 6 章介绍进程，这算得上是操作系统中最重要的概念。在这一章中我们将共同实现一个进程，接着是多个进程，并且让它们同时运行。在这一章中还引入了系统调用的概念，并实现了简单的进程调度。

第 7 章介绍输入/输出系统，引入了控制台的概念，主要涉及的是键盘和显示器的读写。通过这一章，读者可以了解操作系统与外部设备的通信方法。

总体来说，上篇的主要侧重点在于帮助读者开始一段旅程。如果读者跟随本书一起实践的话，那么到本篇结尾处，你应该已经初窥门径，并且具备进一步探索的信心了。

Verum ipsum factum.

—— Giambattista Vico

1 马上动手写一个最小的“操作系统”

虽说万事开头难，但有时也未必。比如说，写一个有实用价值的操作系统是一项艰巨的工作，但一个最小的操作系统或许很容易就实现了。现在我们就来实现一个小得无法再小的“操作系统”，建议你跟随书中的介绍一起动手来做，你会发现不但很容易，而且很有趣。

1.1 准备工作

对于写程序，准备工作无非就是硬件和软件两方面，我们来看一下：

1. 硬件
   - 一台计算机（Linux 操作系统[1]或 Windows 操作系统均可）
   - 一张空白软盘
2. 软件
   - 汇编编译器 NASM

     NASM 最新版本可以从其官方网站获得[2]。此刻你可能会有疑问：这么多汇编编译器中，为什么选择 NASM？对于这一点本书后面会有解释。
   - 软盘绝对扇区读写工具

     在 Linux 下可使用 dd 命令，在 Windows 下则需要额外下载一个工具比如 rawrite[3]或者图形界面的 rawwritewin[4]。当然如果你愿意，也可以自己动手写一个“能用就好”的工具，并不是很复杂[5]。

[1]实际上 Linux 并非一种操作系统，而仅仅是操作系统的内核。在这里比较准确的说法是 GNU/Linux，本书提到的 Linux 泛指所有以 Linux 为内核的 GNU/Linux 操作系统。GNU 经常被人们遗忘，但它的贡献无论怎样夸大都不过分，这不仅仅是因为在你所使用的 GNU/Linux 中，GNU 软件的代码量是 Linux 内核的十倍，更加因为，如果没有以 Richard Stallman 为首的 GNU 项目倡导自由软件文化并为之付出艰辛努力，如今你我可能根本没有自由软件可用。本书将 GNU/Linux 简化为 Linux，仅仅是为表达方便，绝不是因为 GNU 这一字眼可有可无。

[2]NASM 的官方网站位于 http://sourceforge.net/projects/nasm。

[3]rawrite 可以在许多地方找到，比如http://ftp.debian.org/debian/tools/。

[4]下载地址为http://www.chrysocome.net/rawwrite。

[5]我们不需要太强大的软盘读写工具，只要能将 512 字节的数据写入软盘的第一个扇区就足够了。

1.2 十分钟完成的操作系统

你相不相信，一个“操作系统”的代码可以只有不到20行？请看代码1.1。

代码 1.1 chapter1/a/boot.asm

```
1         org     07c00h                  ; 告诉编译器程序加载到7c00处
2         mov     ax, cs
3         mov     ds, ax
4         mov     es, ax
5         call    DispStr                 ; 调用显示字符串例程
6         jmp     $                       ; 无限循环
7   DispStr:
8         mov     ax, BootMessage
9         mov     bp, ax                  ; ES:BP = 串地址
10        mov     cx, 16                  ; CX = 串长度
11        mov     ax, 01301h              ; AH = 13,  AL = 01h
12        mov     bx, 000ch               ; 页号为0(BH = 0) 黑底红字(BL = 0Ch,高亮)
13        mov     dl, 0
14        int     10h                     ; 10h 号中断
15        ret
16  BootMessage:            db      "Hello, OS world!"
17  times   510-($-$$)      db      0       ; 填充剩下的空间，使生成的二进制代码恰好为512字节
18  dw      0xaa55                          ; 结束标志
```

把这段代码用NASM编译一下：

```
▷ nasm boot.asm -o boot.bin
```

我们就得到了一个512字节的boot.bin，让我们使用软盘绝对扇区读写工具将这个文件写到一张空白软盘的第一个扇区。在Linux下可以这样做[6]：

```
▷ dd if=boot.bin of=/dev/fd0 bs=512 count=1
```

在Windows下可以这样做[7]：

```
▷ rawrite2.exe -f boot.bin -d A
```

好了，你的第一个“操作系统”就已经完成了。这张软盘已经是一张引导盘了。

把它放到你的软驱中重新启动计算机，从软盘引导，你看到了什么？

计算机显示出你的字符串了！红色的“Hello, OS world!”，多么奇妙啊，你的“操作系统”在运行了！

如果使用虚拟机比如Bochs的话（下文中将会有关于Bochs的详细介绍），你应该能看到如图1.1所示的画面[8]。

这真的是太棒了，虽然你知道它有多么简陋，但是，毕竟你已经制作了一个可以引导的软盘了，而且所有工作都是你亲手独立完成的！

[6]取决于硬件环境和具体的Linux发行版，此命令可能稍有不同。
[7]rawrite有多个版本，此处选用的是2.0版。
[8]画面看上去有点乱，因为我们打印字符前并未进行任何的清屏操作。

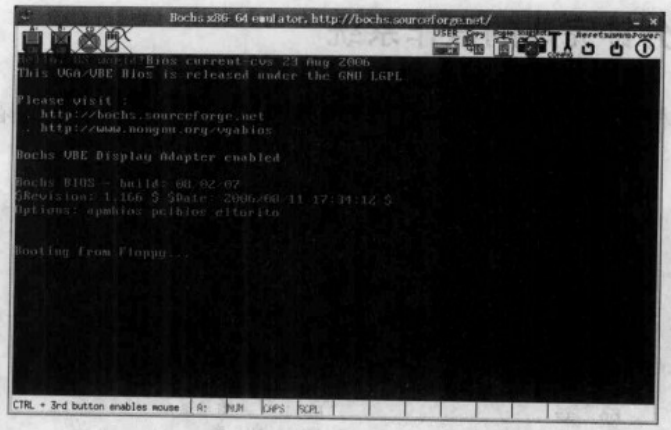

图 1.1 最小的“操作系统”

1.3 引导扇区

你可能还没有从刚刚的兴奋中走出来，可是我不得不告诉你，实际上，你刚刚所完成的并不是一个完整的操作系统，而仅仅是一个最最简单的引导扇区（Boot Sector）。然而不管我们完成的是什么，至少，它是直接在裸机上运行的，不依赖于任何其他软件，所以，这和我们平时所编写的应用软件有本质的区别。它不是操作系统，但已经具备了操作系统的一个特性。

我们知道，当计算机电源被打开时，它会先进行加电自检（POST），然后寻找启动盘，如果是选择从软盘启动，计算机就会检查软盘的 0 面 0 磁道 1 扇区，如果发现它以 0xAA55[9]结束，则 BIOS 认为它是一个引导扇区。当然，一个正确的引导扇区除了以 0xAA55 结束之外，还应该包含一段少于 512 字节的执行码。

好了，一旦 BIOS 发现了引导扇区，就会将这 512 字节的内容装载到内存地址 0000:7c00 处，然后跳转到 0000:7c00 处将控制权彻底交给这段引导代码。到此为止，计算机不再由 BIOS 中固有的程序来控制，而变成由操作系统的一部分来控制。

现在，你可能明白了为什么在那段代码的第一行会出现“`org 07c00`”这样的代码。没错，这行代码就是告诉编译器，将来我们的这段程序要被加载到内存偏移地址 0x7c00 处。好了，下面将对代码的其他部分进行详细解释。

1.4 代码解释

其实程序的主体框架只是第2行到第6行这么一点点而已，其中调用了一个显示字符串的子程序。程序的第2、3、4行是 3 个 mov 指令，使 ds 和 es 两个段

[9]假如把此扇区看做一个字符数组`sector[]`，那么此结束标志相当于`sector[510]==0x55`，且`sector[511]==0xAA`。

寄存器指向与 cs 相同的段，以便在以后进行数据操作的时候能定位到正确的位置。第5行调用子程序显示字符串，然后`jmp $`让程序无限循环下去。

可能有很多人开始学汇编时用的都是 MASM，其实 NASM 的格式跟 MASM 总体上是差不多的，在这段程序中，值得说明的地方有以下几点：

1. 方括号 [] 的使用

   在 NASM 中，任何不被方括号 [] 括起来的标签或变量名都被认为是地址，访问标签中的内容必须使用 []。所以，

   ```
   mov ax, BootMessage
   ```

   将会把“Hello, OS world!”这个字符串的首地址传给寄存器 ax。又比如，如果有：

   ```
   foo dw 1
   ```

   则`mov ax, foo`将把`foo`的地址传给`ax`，而 `mov bx, [foo]`将把`bx`的值赋为 1。

   实际上，在 NASM 中，变量和标签是一样的，也就是说：

   ```
   foo dw 1 等价于 foo: dw 1
   ```

   而且你会发现，Offset 这个关键字在 NASM 也是不需要的。因为不加方括号时表示的就是 Offset。

   笔者认为这是 NASM 的一大优点，要地址就不加方括号，也不必额外地用什么 Offset，想要访问地址中的内容就必须加上方括号。代码规则非常鲜明，一目了然。

2. 关于 `$` 和 `$$`

   `$` 表示当前行被汇编后的地址。这好像不太容易理解，不要紧，我们把刚刚生成的二进制代码文件反汇编来看看：

   ```
   ▷ ndisasmw -o 0x7c00 boot.bin >> disboot.asm
   ```

   打开 disboot.asm，你会发现这样一行：

   ```
   00007C09  EBFE                jmp short 0x7c09
   ```

   明白了吧，`$` 在这里的意思原来就是 0x7c09。

   那么 `$$` 表示什么呢？它表示一个节（section[10]）的开始处被汇编后的地址。在这里，我们的程序只有 1 个节，所以，`$$` 实际上就表示程序被编译后的开始地址，也就是 0x7c00。

在写程序的过程中，`$-$$`可能会被经常用到，它表示本行距离程序开始处的相对距离。现在，你应该明白`510-($-$$)`表示什么意思了吧？

`times 510-($-$$) db 0`表示将 0 这个字节重复`510-($-$$)`遍，也就是在剩下的空间中不停地填充 0，直到程序有 510 字节为止。这样，加上结束标志 0xAA55 占用的 2 字节，恰好是 512 字节。

[10]注意：这里的 section 属于 NASM 规范的一部分，表示一段代码，关于它和 `$$` 更详细的注解请参考 NASM 联机技术文档。

1.5 水面下的冰山

即便是非常袖珍的程序，也有可能遇到不能正确运行的情况，对此你一定并不惊讶，谁都可能少写一个标点，或者在一个小小的逻辑问题上犯迷糊。好在我们可以调试，通过调试，可以发现错误，让程序日臻完美。但是对于操作系统这样的特殊程序，我们没有办法用普通的调试工具来调试。可是，哪怕一个小小的引导扇区，我们也没有十足的把握一次就写好，那么，遇到不能正确运行的时候该怎么办呢？在屏幕上没有看到我们所要的东西，甚至于机器一下子重启了，你该如何是好呢？

每一个问题都是一把锁，你要相信，世界上一定存在一把钥匙可以打开这把锁。你也一定能找到这把钥匙。

一个引导扇区代码可能只有 20 行，如果 Copy&Paste 的话，10 秒钟就搞定了，即便自己敲键盘抄一遍下来，也用不了 10 分钟。可是，在遇到一个问题时，如果不小心犯了小错，自己到运行时才发现，你可能不得不花费 10 个 10 分钟甚至更长时间来解决它。笔者把这 20 行的程序称做水面以上的冰山，而把你花了数小时的时间做的工作称做水面下的冰山。

古人云：“授之以鱼，不如授之以渔。”本书将努力将冰山下的部分展示给读者。这些都是笔者经历了痛苦的摸索后的一些心得，这些方法可能不是最好的，但至少可以给你提供一个参考。

好了，就以我们刚刚完成的引导扇区为例，你可以想像得到，将来我们一定会对这 20 行进行扩充，最后得到 200 行甚至更多的代码，我们总得想一个办法，让它调试起来容易一些。

其实很容易，因为有 Bochs，我们前面提到的虚拟机，它本身就可以作为调试器使用[11]。请允许我再次卖一个关子，留待下文分解。但是请相信，调试一个引导扇区——甚至是一个操作系统，在工具的帮助下都不是很困难的事情。只不过跟调试一个普通的应用程序相比，细节上难免有一些不同。

如果你使用的是 Windows，还有个可选的方法能够帮助调试，做法也很简单，就是把“`org 07c00h`”这一行改成“`org 0100h`”就可以编译成一个.COM 文件让它在 DOS 下运行了。我们来试一试，首先把`07c00h`改成`0100h`，编译：

```
▷ nasm boot.asm -o boot.com
```

好了，一个易于执行和调试的引导扇区就制作完毕了。如果你是以 DOS 或者 Windows 为平台学习的编程，那么调试.COM 文件一定是你拿手的工作，比如使用 Turbo Debugger。

调试完之后要放到软盘上进行试验，我们需要再把`0100h`改成`07c00h`，这样改来改去比较麻烦，好在强大的 NASM 给我们提供了预编译宏，把原来的“`org 07c00h`”一行变成许多行即可：

代码 1.2 chapter1/b/boot.asm

```
1 ;%define _BOOT_DEBUG_            ; 制作 Boot Sector 时一定将此行注释掉！
2                                  ; 去掉此行注释后可做成 .COM文件易于调试：
```

[11]如果你实在性急，请翻到第12页第2.1.4节看一下具体怎么调试。

```
3                                   ;   nasm Boot.asm -o Boot.com
4
5   %ifdef  _BOOT_DEBUG_
6           org  0100h              ; 调试状态，做成 .COM 文件，可调试
7   %else
8           org  07c00h             ; BIOS 将把 Boot Sector 加载到 0:7C00 处
9   %endif
```

这样一来，如果我们想要调试，就让第一行有效，想要做引导扇区时，将它注释掉就可以了。

这里的预编译命令跟C语言差不多，就不用多解释了。

至此，你不但已经学会了如何写一个简单的引导扇区，更知道了如何进行调试。这就好比从石器时代走到了铁器时代，宽阔的道路展现在眼前，运用工具，我们有信心将引导扇区不断扩充，让它变成一个真正的操作系统的一部分。

1.6 回顾

让我们再回过头看看刚才那段代码吧，大部分代码你一定已经读懂了。如果你还是一个NASM新手，可能并不是对所有的细节都那么清晰。但是，毕竟你已经发现，原来可以如此容易地迈出写操作系统的第一步。

是啊，这是个并不十分困难的开头，如果你也这样认为，就请带上百倍的信心，以及一直以来想要探索OS奥秘的热情，随我一起出发吧！