

# 走进程序世界的田园（上）

作者：[于渊](#)

(最初发表于《程序员》2004年第7、8、10月号)

## 引子

不知道你有没有思考过这样的问题，这个世界上很少有人是数学家、物理学家和化学家，那么，为什么我们要从小学一直到初中高中花费累计十年以上的时间来学习数学、物理和化学课程？你可能有时感到在学校里学到的东西在工作中并没有得到很好的应用，那么我们为什么还要寒窗苦读？为什么我们还要主张尽可能普及大学教育？

其实很显然一件事有没有用不是那么直接的，学校教育教会我们的，不仅仅是圆周率、牛顿定律和分子式，更重要的，我们在获得了这个世界各种常识的同时，学会了如何思考，用逻辑的、形象的、收敛的和发散的等等各种思维方式来思考和解决问题。

是的，我们学会了两样东西：常识，以及如何思考。

我相信这世间的道理是彼此相通的，比如，你是否也思考过这样的问题，大学课程中为什么要教授操作系统？

大学中有很多专业开设操作系统课程，可事实是，即便是计算机系的学生也只是有很少一部分毕业后真正从事操作系统的开发。而且实际上，即便不知道进程管理磁盘调度好像也并不能影响一个人从事编写应用程序的工作。

那么我们是不是就因此可以把操作系统这门课程从课程表中删除呢？或者由必修改成选修？答案无疑是否定的。这就跟我们不能将物理课从中学课程表中删除是一样的道理。其实在计算机世界中操作系统课程教会我们的，也是这两样东西：常识和如何思考。这两样东西是如此重要，以至于我们要以最大的热情去掌握它们。

如果你从来没接触过进程管理，你可能回答不了为什么一颗 CPU 可以允许同时运行多个程序；如果你不知道什么叫做段页式存储，你可能怎么也不理解为什么不同的程序中相同的内存地址中会是不同的内容，而这些内容居然不会相互影响；如果你不明白 I/O 系统的原理，你可能不知道为什么在 Linux 下用汇编语言编程显示字符串不能再使用你熟悉的 BIOS 中断 int 10h。这些是常识的缺乏，通常情况下缺乏常识可能不会有严重后果，但是当你需要开发驱动程序时，当你的程序中涉及到大量内存操作时，你会发现再也无法忽视操作系统的理论及实现，常识此时变得不仅仅是重要，甚至是攸关生死。就好像你可以不知道为什么推上电闸电灯会亮，却不能不知道绝对不能用手去推电闸上那两片金属。

可能有一天你写程序时需要考虑共享资源的分配，以及若干过程的调度，你冥思苦想得不出一个好的方法，但实际上这些问题在操作系统理论体系中已经有了很成形的解决方案，你只需要拿来主义就够了。不仅如此，其实很多开发中遇到的问题都能在操作系统理论中找到它们的影子，这就是我们上面提到过的，操作系统教我们如何思考。

我们所能见到的任何一个 32 位操作系统都是庞大且繁杂的，这是一个系统工程，它把进程管理，存储管理，I/O 控制，磁盘管理等等许许多多模块整合在一起，使它们协同工作，并且向下能控制最底层的硬件，向上又能提供应用程序的接口，严谨又不失灵活。宏观上，这是一套生动的设计模

式教材；微观上，你又能在其中找到各种各样的算法和数据结构。通过深入地研究操作系统，无疑能让我们学会思考从宏观到微观的各种问题。

如果你是一个应用程序开发者，对操作系统的学习就好比苦练内功，从影响的深远程度来看，它远比招式重要。只有对操作系统理论有一定的了解，你才能够有信心面对开发中遇到的各种问题，因为你了解本质，了解计算机的各个部分“到底”是怎么运行的，所以你才能在处理各种情况时得心应手。

如果你是一个痴迷于逻辑思维的思考者，操作系统中用到的各种模式和算法都堪称经典，学习它们就好像学习下棋时研究前人的经典棋谱。无论是具体实现还是其中的思想，都值得去仔细研究和借鉴。

如果你仅仅是个孜孜以求的操作系统爱好者，可能从引导扇区到进程调度，每一个细节都能让你兴奋不已。从最最细节的汇编代码，到各种各样的调度算法，可能都会让你热情高涨。

是啊，操作系统是这样一座激动人心的宝藏，等待我们去挖掘。但可惜的是，我们对它的关注和研究却始终稍嫌不够，可能只有少数对操作系统怀有极大兴趣的学习者，才会有耐心通过参考理论书籍和阅读开放源代码的方式来对它进行研究。大部分人，可能开始就被那些纯粹理论的大学教材吓怕了。

本着共同研究，开发宝藏的原则，我想在这里跟大家一道，回顾和学习一下操作系统中一些具体的内容。所谓具体者，是不想空谈理论，而是从一些具体细节出发，希望能引起更多人对它的兴趣。因为我一直觉得，从感性认识中获得的体验要比纯粹理论深刻地多。而从实际操作中获得的成就感，更是能够大大增强学习者的信心和兴致，从而让学习过程变得轻松而有效。

我将从最简单的引导扇区的编写开始，尽量消除操作系统与读者之间的距离感，然后我们来共同研究一下保护模式这一 x86 系统中最基础的技术，再然后在进程管理，I/O 控制等方面做一些探讨。

写程序有时候看上去很容易，大家完全可以借助最新的技术，最先进的开发工具，用比较简单的方法和比较少的时间，实现功能强大的应用，但同时，这些技术却让开发者离操作系统技术渐行渐远。然而一个优秀的开发者应该具备各个方面的素质，尤其是扎实的内功，毕竟，程序最终还是通过操作系统由硬件来运行的。我在今后的文章中，也会尽量多地跟大家讨论操作系统中跟应用软件开发结合点，从而尽量让从事各方面开发的读者都能够学有所用。

同时，我也非常欢迎大家参与到这个讨论中来，在相互学习的同时，希望能够引起大家对于操作系统技术更多的关注。

## 引导扇区释疑

什么？又是酱鸡翅？！你眉头皱起——公司的工作餐怎么就不能出点新的花样？吹着空调还会出汗的大热的天却还要忍受这份油腻，此刻的你显然更需要西芹百合！这就如同你曾深深痴迷的这份工作，其实你是多么怀念在黑色背景上敲出一行命令之后屏幕哗哗卷动的景象，多么自豪于曾经用汇编在屏幕上绘出的彩色的图案，而如今，虚拟机，API 等等新鲜玩意让你总感觉仿佛想亲吻女友的手时却亲到了手套，没有亲密感，也许，你不仅仅需要 Java、.NET 这样的高楼大厦，你同样需要清新的田园，比如你每天都在使用却可能一知半解的——计算机如何引导。

那么，现在就让我们一起走进这田园，看看计算机究竟是如何引导的。

## 计算机的启动过程

如果你买来新的计算机，硬盘上还是一片空白的时候，你按下 power 键，仍然可以看到许许多多字符和图案，这显然不是操作系统的一部分，而是 BIOS 的程序在工作。当然，最后你能看到一行字，提示你插入引导盘。是的，BIOS 在寻找一个可供引导的磁盘，找到之后，便会加载盘上的引导模块，并交出控制权，将接力棒传递给操作系统。

我们考虑最简单最易学习的情况，那就是软盘。

如果你将一张非引导磁盘插入软驱的话，BIOS 仍然会报错，提示你插入一张系统盘，这说明 BIOS 并非来者不拒全部试图加载执行的，那么它选择的标准是什么呢？实际上很简单，它会去检查软盘的 0 面 0 磁道 1 扇区（大小为 512 字节），如果发现它以 55AA 结束，则 BIOS 认为它是一个引导扇区，也就是我们说的 Boot Sector。当然，一个正确的 Boot Sector 只有 55AA 这个结束标志是没有意义的，它还应该包含一段少于 512 字节的执行码，以便能被放在一个扇区内并正确运行。

一旦 BIOS 发现了 Boot Sector，它就会将这 512 字节的整个扇区内容装载到内存的 0:7c00 处，然后跳转到 0:7c00 处将控制权彻底交给这段引导代码。到此为止，计算机不再有 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。

## 马上实践——一个最小的引导扇区

### 准备工作：

#### 硬件

- 一台计算机（Windows 操作系统）
- 一张空白软盘

#### 软件

- 汇编编译器 NASM。最新版本可以在此链接处获得：<http://sourceforge.net/projects/nasm>。（此刻你可能会有疑问，为什么是 NASM，而不是 MASM 或者 TASM，对于这一点我稍候再来解释）
- 软盘绝对扇区读写工具。其实你完全可以自己写一个，用 CreateFile 和 WriteFile 这两个 API 就搞定了，非常容易。我就是这样做的，省去了在网上寻找工具的时间。

- 最好有一个虚拟机比如 VirtualPC，可以在试验的时候不必重启自己的计算机。

## 代码

我们来看这一段代码：

```
org 07c00h          ; 告诉编译器程序加载到 7c00 处
jmp $              ; 无限循环
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
dw 0aa55h          ; 引导扇区需要以 55AA 结束
```

在对程序进行解释之前，为了尽快看到效果有初步的感性认识，请先随我来做以下操作：  
首先用 NASM 编译一下：

```
nasm boot.asm -o boot.bin
```

我们就得到了一个 512 字节大小的 boot.bin，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，好了，这张软盘已经是一个引导盘了。

然后把它放到你的软驱中重新启动计算机，或者使用 VirtualPC 模拟启动过程，从软盘引导，你看到了什么？

答案是什么令人惊喜的结果也没有出现，这倒容易理解，因为我们的程序第一个语句就是一个死循环。我们除了让程序停滞在那里，其余什么也没做。

这显然并不令人满意，我们得看到些效果才行，让我们将代码稍作修改：

```
org 07c00h          ; 告诉编译器程序加载到 7c00 处
mov ax, 0b800h
mov es, ax          ; 设置 es 以便直接写显存
mov byte [es:0], 'a' ; 在显存第一个字节写入字符 'a'
mov byte [es:1], 0ch ; 在显存第二个字节写入十六进制值 C，表示黑底红字
jmp $              ; 无限循环
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
dw 0aa55h          ; 引导扇区需要以 55AA 结束
```

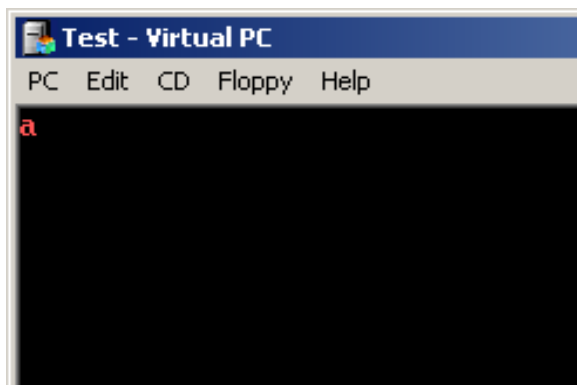
我们在程序无限循环之前插入了四行，目的是让我们的引导程序能显示一个红色的字符 'a'。插入的这四行比较易懂，效果是在 B800:0000 处写入了两个字节：'a' 和 0ch。我们知道，B800:0000 恰好是显存的首地址。

同样的方法编译，写入磁盘并重启，你看到了什么？

你看到红色的字符 a 了！

多么令人激动啊，这表明我们的程序正确运行了，进一步，这说明我们自己编写的引导扇区试验成功了！

如果你用的是 VirtualPC，出现的应该是这样的景象（局部）：



这简直是太妙了，因为有了这样的开头，就意味着你可以在此基础上做出任意的扩展，甚至写出自己的操作系统！这一步迈出来的确并不难，但却具有历史意义！

## 解释

你可能早就已经迫不及待，想要了解上面代码的所有细节，下面我就来详细解释一下。

## 为什么是 NASM

你可能感到很奇怪，为什么居然有人用 NASM 这样东西，而不是你从前使用的 MASM 或者 TASM，实际上这有点涉及到个人喜好，但是事实是，我从开始无意中接触到 NASM 开始，就决定从此彻底抛弃 MASM 了。因为它具备以下几个主要特点：

### 1. 代码清晰，避免了 MASM 中容易混淆的语法。

这项特点在 NASM 的多个细节中都有体现，在这里我仅举两例。

第一，在 NASM 中，任何不被方括号 [] 括起来的标签或变量名都将被认为是地址，访问标签中的内容必须使用 []。所以，

```
mov ax, Message
```

将会把 Message 对应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 的值赋成为 1。

实际上，在 NASM 中，变量和标签是一样的，也就是说，

```
foo dw 1 ≡ foo: dw 1
```

而且你会发现，offset 这个东东在 NASM 也是不需要的。因为不加方括号时表示的就是 offset。

我个人认为这是 NASM 的一大优点，要地址就不加方括号，也不必额外的什么劳什子 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。

第二，既然所有标签都是地址，使得 NASM 具有另外一个特点，就是不记忆变量类型，所以在给变量赋值的时候，必须加上赋值的类型，比如：

```
mov byte [var1], 'a'
```

这个命令中，‘byte’是不能少的。

### 2. 可以在不同平台中使用

如果你想学习一次就可以在不同平台下使用的话，NASM 几乎是唯一的选择。如果你想进行完全的代码移植，NASM 是完美的工具。因为不管在 Dos, Windows 还是 Linux, NASM 都是可用的，而且用法完全相同。

### 3. 免费

可能这项特性已经不足以吸引你的眼球，但却的确是它的一个可爱的特性。

本文不是专门的 NASM 介绍文章，写这么多已经稍嫌啰嗦，但是我认为它的确是一个值得推荐的工具，尤其是，如果你不想仅仅了解引导扇区的写法，而是在此基础上深究下去，进行操作系统的研究的话，我保证你会越来越体会到 NASM 这一工具的优点。

## 关键代码解释

上面两段代码的注释已经写得比较清晰，在这里对几个问题着重强调一下。

### 1. org 的使用

org 的作用是告诉编译器，这个程序将来被加载到内存的哪个位置。我们在稍后的例子中会使

用到常量，编译器就是以 org 指定的这个地址为基准来确定常量的地址。

## 2. 关于\$和\$\$

\$表示当前行被汇编后的地址。这好像不太好理解，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 >> a.asm
```

打开 a.asm，你会发现这样一行：

```
00007C09 EBFEB          jmp short 0x7c09
```

明白了吧，\$在这里的意思原来就是 0x7c09（在加载到内存之后）。

那么\$\$表示什么呢？它表示一个节（section）的开始处被汇编后的地址。在这里，我们的程序只有一个节，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，也即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 55AA 占用的两个字节，恰好是 512 个字节。

## 3. 55AA 还是 AA55

初学者经常被这个问题搞得非常头痛，总也搞不清楚到底谁在前谁在后，其实归根到底还是没把本质弄明白。

IBMPC 的原则是“高位在高字节”。举个例子，如果有一个 DWORD 类型的数 0x12345678 放在内存中，看起来会是这样：

```
L —————> H  
78 56 34 12
```

因为 78 处在数字的低位，于是也会被放到内存的低位。

这里有一点需要思考一下，就是计算机只知道数字，不知道类型，所以，从内存的某个地址取出一个数，必须在指明类型的情况下才是有意义的，比如已知有这样的内存映像：

```
L —————> H  
78 56 34 12
```

若想取出一个 BYTE，你会得到 0x78；若想取出一个 WORD，你会得到 0x5678；若想取出一个 DWORD，你会得到 0x12345678。

回头看看我们的代码：

```
dw 0aa55h
```

我们指定把 0aa55h 这个 WORD 类型数字放在引导扇区最末端，aa 处在数字的高位，会被放到内存的高位，于是它在内存中的映像应该是：

```
L —————> H  
55 aa
```

很简单，也很明了不是吗？

## 再作扩充——一个变一行

只显示一个字符显然是不够的，我们想要更进一步的成就感，比如显示一个字符串。可是如果显示每一个字符都要两行代码来实现的话，难免显得笨拙而低效。是的，你一定想到了，我们可以使用 BIOS 中断。

请看代码：

---

```
org 07c00h          ; 程序会被加载到 7c00 处，所以需要这一句  
mov ax, cs
```

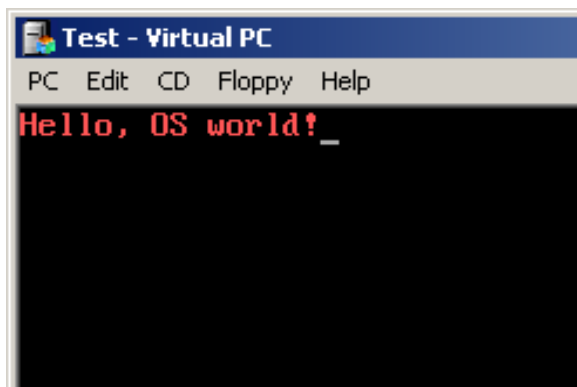
```

mov ds, ax
mov es, ax
Call DispStr ; 调用显示字符串例程
jmp $ ; 无限循环
DispStr:
mov ax, BootMessage
mov bp, ax ; ES:BP = 串地址
mov cx, 16 ; CX = 串长度
mov ax, 01301h ; AH = 13, AL = 01h
mov bx, 000ch ; 页号为 0 (BH = 0) 黑底红字 (BL = 0Ch, 高亮)
mov dl, 0
int 10h ; int 10h
ret
BootMessage: db " Hello, OS world!"
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
dw 0aa55h ; 引导扇区需要以 55AA 结束

```

这段代码看上去长了许多,但实际上主体框架只有 5 行(从第 2 行到第 6 行),其中调用了一个显示字符串的子程序。程序的第 2、3、4 行是三个 mov 指令,使 ds 和 es 两个段寄存器指向与 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。

我们来试验一下,编译,写入磁盘,启动:



成功!

来来来,下面泡一杯咖啡,然后靠在椅背上静静欣赏一下自己的成果吧,让你的屏幕暂时停在这一刻。这是一件多么有趣的作品!虽然我们的代码很短,却已经涉及到了如此多的技术细节,我们甚至使用了 BIOS 中断,在中断例程的帮助下,我们几乎是无所不能的,想象一下吧,最振奋人心的一点是,你可以进行磁盘操作,将更多的程序加载到内存中并且执行,这意味着你真的已经可以在这个小东西的基础上一点点扩充,甚至建造操作系统的大厦!

## 之所以这是田园,因为这里离泥土最近

你可能很久都没有过如此透彻地了解一件事,就好像你又看到泥土中生长出绿色的植物。这是一种回归自然的感觉,那么,就请尽情享受这一刻爽快的感受吧,忘掉 Java, .NET, 还有那讨厌的酱鸡翅。

## 段页式存储亲密接触

今年的雨水好像格外的充足，城市在这样的考验下变得手忙脚乱。该死的交通！你骂道。是啊，有谁面对满城排不走的水会不着急上火呢？城市啊城市，你到底该让我爱呢，还是让我恨？

那么朋友，让我们再次回到我们的田园吧，在这里，松软的泥土，DIY的排水沟渠，让你的田地足以经受再大的风雨。饿了吃瓜，渴了痛饮一瓢井水，累了休息，最惬意的，外面雨下得酣的时候，点一盏油灯，捧一本红楼，坐在窗前，细细读来，夫复何求！

是啊，田园永远这样让人陶醉，并且远离烦恼。那么今天的田园从哪里说起呢？就从邻家小朋友的中学课本开始吧。

## 两小儿辩日

有时候，谁都可能不经意间被一些已经习以为常的小事难住，即便已经成为高人也是这样。就好像你今天无意中翻到邻家小朋友课本中的《两小儿辩日》，及至最后“孔子不能决”，圣贤尚且会被小儿的问题迷惑。

实际上可能这些貌似渺小的问题背后恰恰隐藏着复杂的原理。比如一个非常简单的例子，如果你写两个一模一样的程序，然后同时打开两个VC，同时调试，你会发现，从变量地址到寄存器的值，几乎全部都是一样的！而这些“一样的”地址之间完全不会混淆起来，而是各自完成着自己的职责。试图思考这个问题时你的脑海可能浮现出诸如“内存管理”，“映射”等等这样的关键词，好像朦胧中明白是怎么回事，但深究起来，这并不是个一两句话可以说清的问题。

再比如一个稍稍复杂点的例子。DLL你一定不会陌生。它不仅实现了代码的动态加载，更是可以让同一个模块在多个进程间共享。我们知道，如果我们想通过DLL在多个进程间共享数据的话，单纯在DLL中声明全局变量是不够的，在其中一个进程中对DLL中的全局变量的修改并不能在其它进程中看到。于是乎要实现数据的共享，我们必须定义一个共享数据段。那么在共享数据段中定义的全局变量跟一个普通的全局变量之间，到底有什么不同？

其实隐藏在问题背后的，就是我们耳熟能详却很少深究的段页式存储管理。今天，就让我们来一次亲密接触，来看一下它到底是怎么回事。

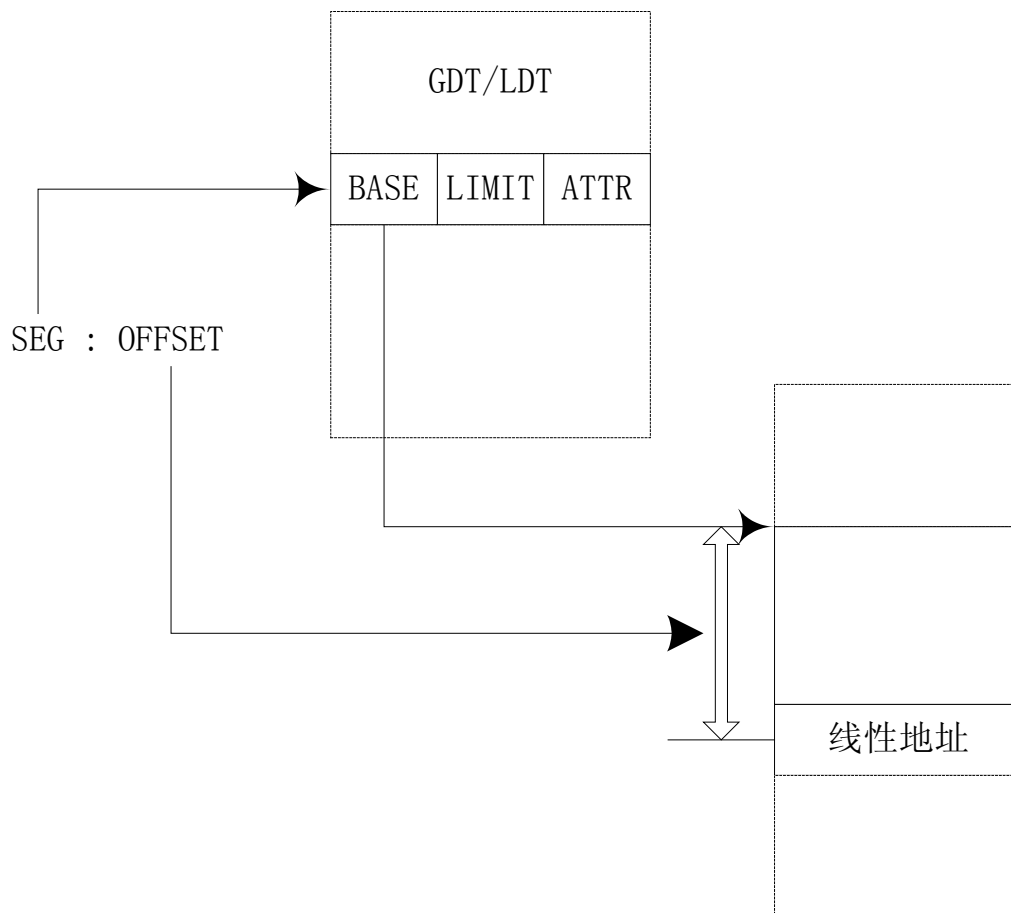
## 段页式存储管理概述

### 分段管理机制

在8086/8088汇编中，一个地址是由段和偏移两部分组成的。物理地址遵循这样的计算公式：  
物理地址 = 段值 × 16 + 偏移

而对于80386以及之后的32位CPU，在保护模式下，地址仍然可以用“SEG:OFFSET”这样的形式来表示，只不过“段”的概念发生了根本的变化。实模式下段值还是可以看作是地址的一部分的，段值为XXXXh表示以XXXX0h开始的一段内存。而保护模式下，虽然段值仍然由原来16位的CS、DS等寄存器表示，但此时它仅仅变成了一个索引，这个索引指向GDT或LDT的一个表项（实际上这样的表项有个专门的名称叫做描述符），这个表项是个数据结构，在这个数据结构中，详细定义了段的起始地址，界限，属性等等内容。将段的起始地址与偏移相加，就得到了线性地址。

下面这个示意图大致表示出了简化后的由段值和偏移得到线性地址的过程：



实际情况要复杂一些，比如进行权限、段界限检验等等，但大致的原理是这样的。

有一点比想象中麻烦的是，我们由此得到的仍然不是物理地址，而是所谓“线性地址”，在分页管理机制不生效的情况下，线性地址就等同于物理地址，但大多数情况下分页管理机制是生效的，这时要得到物理地址还要经过一步转换。

说到这里我们不妨回头想一下，我们做试验的两个相同的程序中，寄存器值是完全相同的，这就意味着，对于同样的变量或函数入口，段值和偏移地址是完全一样的，也就是说，线性地址也是完全一样的。一样的线性地址中放着不同的内容，可以猜到，一定是分页管理机制在同时起作用。

好，那么我们就来看一下分页管理机制是怎么回事。

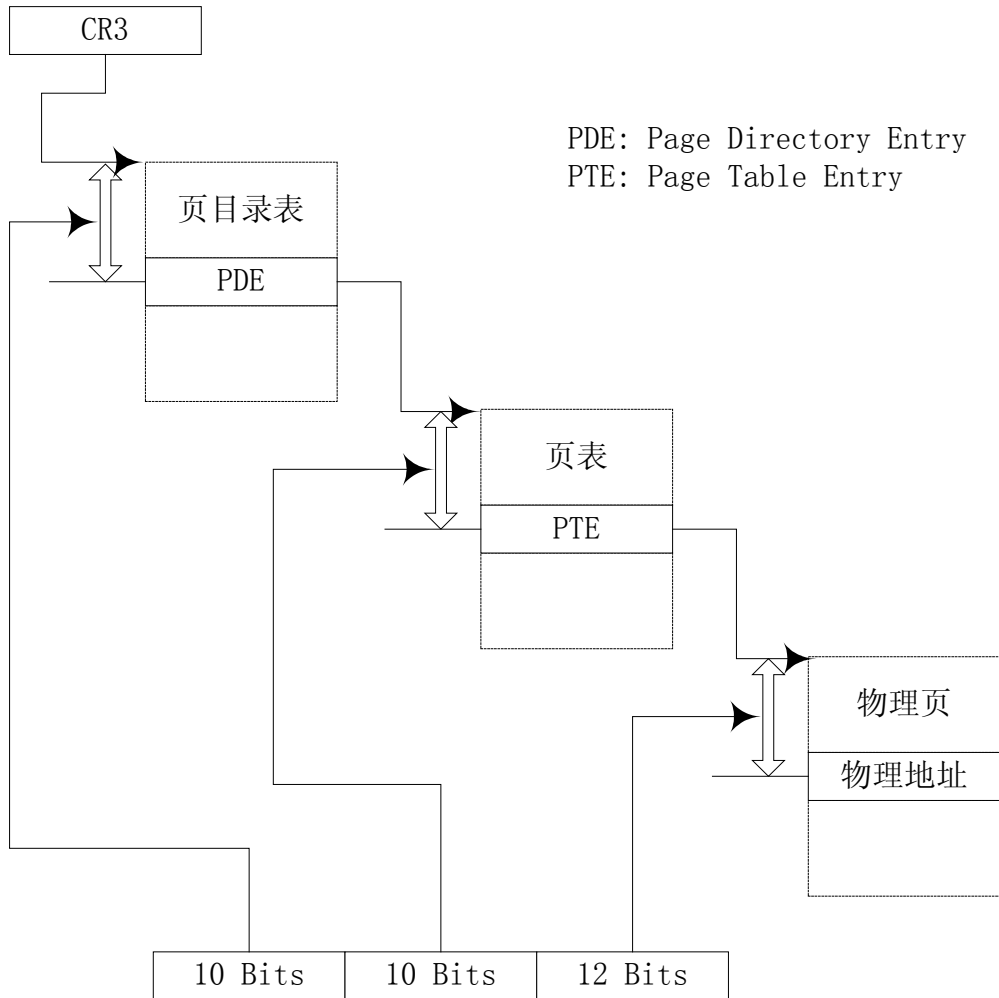
## 分页管理机制

你可能会问，分段管理机制看上去已经很像样了，为什么还要加上分页管理机制呢？其实它的主要目的在于实现虚拟存储器。因为稍候你可以看到，线性地址中任意一个页都能映射到物理地址中的任何一个页，这无疑使得内存管理变得相当灵活。

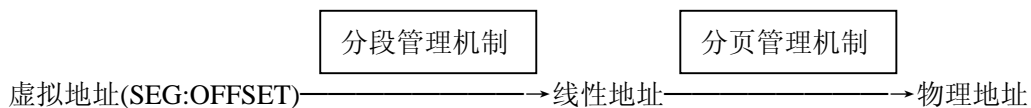
在 80386 中，每个页大小固定，为 4096 字节（4K）。使用两级页表，第一级叫做页目录，大小为 4K，存储在一个物理页中，每个表项 4 字节长，共有 1K 个表项。每个表项对应第二级的一个页表，第二级的每一个页表也有 1K 个表项，每一个表项对应一个物理页。

线性地址向物理地址转换时，先是从由 CR3 指定的页目录中根据线性地址的高 10 位得到页表地址，然后在页表中根据线性地址的第 12~21 位得到物理页首地址，将这个首地址加上线性地址低 12 位，便得到了物理地址。

下面是转换过程示意图：



到目前为止，段页式存储管理的基本原理想必你已经有了一个大致轮廓，总结起来是这样的：



现在再回想一下我们的例子，你应该已经明白了，一定是相同的线性地址通过不同的页目录和页表被映射到了不同的物理地址。为了证明我们的猜测以及亲自看一下由线性地址到物理地址这一映射过程，我们下面就亲自动手来看一下。

## 让变量现出原形

现在我们知道，只要可以查看 GDT 和 CR3，并且可以访问物理内存，随便给我们一个虚拟地址，我们自己就得出相应的物理地址。而通过 SoftIce 这一切都能办到，那么，我们就顺着我们的思路走下去，试试我们的想法是否正确，检验一下我们对于段页式存储的理解是不是已经足够深刻。

我们还是用刚才那两个几乎完全相同的程序，假设它们都是 VC 的 MFC AppWizard 生成的基于对话框的程序，名字叫做 SimpleA 和 SimpleB。在两个对话框上各有一个按钮，按钮的单击事件是这样的：

### SimpleA

```
int gi_var;
```

```

void CSimpleADlg::OnButton1()
{
    gi_var = 0x123;
} //此处设置断点

```

---

## SimpleB

```

int gi_var;
void CSimpleBDlg::OnButton1()
{
    gi_var = 0xABC;
} //此处设置断点

```

---

现在我们打开 SoftICE Symbol Loader，“File-Open”，打开 SimpleA.exe，点击菜单中的“Module-Translate”和“Module-Load”。

用 Ctrl+D 呼出 SoftIce，用“file SimpleADlg.cpp”命令打开源代码，在 CSimpleADlg::OnButton1() 这一函数的结尾处设置断点。F5 退出。

下面再次打开 SoftICE Symbol Loader，对 SimpleB.exe 做同样的操作。

好了，准备就绪，点击 SimpleA 的按钮，SoftIce 被弹出，用命令“d”看一下变量 gi\_var 的虚拟地址：

```
:d gi_var ↓
```

---

将数据窗口的显示格式设成 DWORD，我们看到，gi\_var 的地址是 0023:004168c0，值是 0x00000123。

我们先把 0023:004168c0 这个虚拟地址转成线性地址，所以我们需要先看一下 GDT，看一下 0023 这个选择子对应的是怎样一个段：

```

:gdt ↓
Sel.  Type      Base      Limit      DPL  Attributes
GDTbase=80036000  Limit=03FF
0008  Code32     00000000  FFFFFFFF  0    P  RE
0010  Data32     00000000  FFFFFFFF  0    P  RW
001B  Code32     00000000  FFFFFFFF  0    P  RE
0023  Data32     00000000  FFFFFFFF  0    P  RW
.....

```

---

原来这个段对应的基地址是 0，那么毫无疑问，gi\_var 的线性地址就是 004168c0 了。我们先来把这个地址分解，它的高 10 位，中 10 位和低 12 位的值分别是：

0x1	0x16	0x8c0
-----	------	-------

由于页目录的一个表项是 4 个字节，所以 PDE 的地址可以这样计算：

addr of PDE = CR3 + 1 \* 4

下面我们就来看一下 CR3 这个寄存器的值：

```

:addr ↓
CR3      LDT Base:Limit  KPEB      PID      NAME
.....
1B481000          8142A020  04A8     SimpleA

```

---

1D5C2000                    8144E700    03D8    SimpleB

.....

---

进程 SimpleA 的 CR3 值为 1B481000，所以有

**addr of PDE = 1B481000 + 4 = 1B481004**

通过 peek 命令，我们来看一下 PDE 的值是什么：

---

```
:peek d 1B481004 ↓
0x1197E067  0001000110010111110000001100111  295166055
```

---

表项的低 12 位是属性，故 PTE 的地址可以这样计算：

**addr of PTE = 0x1197E000 + 0x16 \* 4 = 0x1197E058**

进一步看一下 PTE 的内容：

---

```
:peek d 1197E058 ↓
0x1CD29067  00011100110100101001000001100111  483561575
```

---

好，由此我们可以得出 gi\_var 的物理地址了：

**phys. Addr of gi\_var = 0x1CD29000 + 8C0 = 0x1CD298C0**

来看看其中的内容是什么：

---

```
:peek d 1CD298C0 ↓
0x00000123  000000000000000000000000100100011  291
```

---

啊，真的是 0x123！这决不是巧合，我们通过 GDT 和 CR3 手动计算出了变量的物理地址！

为了进一步确认，我们用 phys 命令查看一下所有映射到此物理地址的虚拟地址：

---

```
:phys 1CD298C0 ↓
004168C0
9CD298C0
```

---

我们看到 004168C0 这个地址了，这再一次证明我们的过程是正确的！

按 F5 退出 SoftIce，用同样地办法再来试一下 SimpleB。你会发现，从 CR3 到页目录到页表，已经统统跟 SimpleA 不一样了。这也正契合了我们之前的猜测。

## 云开雾散，水落石出

如果你也跟我一道将这个过程走过一遍的话，我想你一定已经了解了整个事情的原委，关于段页式存储管理的整套机制。此时此刻回头想想，你可能发现原先一直不求甚解的东西现在真的变得豁然开朗。就比如文章开头提过的那个 DLL 的例子，如果你用同样的方法跟踪一遍的话，最后你会发现不同进程中共享数据段内的变量最终指向同一块内存地址，而一般的全局变量就会像我们刚刚分析过的 gi\_var 一样，在不同的进程中被映射到了不同的物理页面。

可能你会有“这理所当然”的想法，但实际是，你自己亲眼所见这样激动人心的方式见证了存储管理的原理，当再一次你需要通过 DLL 的方式在不同进程中共享数据时，你不会再忘掉把它放在共享数据段中，因为一个思想已经深深的植根于你的脑海。当你遇到其它类似较为底层的内存相关问题时，你将因为了解它的具体实现方式而变得从容而游刃有余。

## 进一步思考：段页式存储为什么

小的时候，老师告诉我，爱问为什么的孩子才是好孩子。牛顿看到苹果落地，瓦特看到水开了蒸汽顶起壶盖儿，便引起了思考，因为思考，平凡变成了伟大。我们有理由相信，在牛顿和瓦特之前，曾经有无数的人也看到过苹果落地壶盖跳动，但是他们习以为常，认为那理所应当，于是失去了变成伟大的机会。

此时的你，可能不是小孩子很久了，那么，你是否还记得老师的教导？

如果你还记得，那么你有没有思考过，为什么会有段页式存储这样的东西？

可能由于教材太过枯燥，你还没来得及弄明白是怎么回事一节课便告结束，可能内容太过繁杂，你还没来得及好好消化就被推向了考场，可能 CS 太过好玩，你还没来得及抽出时间去想这复杂的问题一整个学期就已过去。那么现在机会来了，今天，你已经用眼睛亲眼见证了段页式存储的整个运作过程，来思考这个问题恰是时候。

### 页式存储

为什么要有分页机制这个问题我们在上文中已经稍有涉及，我们提到，“它的主要目的在于实现虚拟存储器”，但实际上这个答案仍不能让我们完全满意，因为我们还可以进一步提问：为什么要实现虚拟存储器？实现虚拟存储器为什么就需要分页？这是它的主要目的，那么有没有次要目的？

首先需要虚拟存储器的根本原因在于内存并不总是够用的。Mr. Gates 曾经有一个著名的预言，他说 640K 内存对任何人都足够了，但事实上现在很多计算机的内存已经多于 640M 了，可有时候还是捉襟见肘。软件的发展速度总是这么快。这就使得，作为一个操作系统，必须考虑到内存不够用的情况，也就是实现虚拟存储器。

实现了虚拟存储器使得程序可以使用的空间比实际存储器大得多。因为它利用了磁盘作为内存的扩展。那么我们来想象一下这样一个简单的过程：在必要时，操作系统将进程 A 换出到磁盘，再将进程 B 换入到内存，在另外的某时刻，操作系统再将进程 C 换出到磁盘，将进程 A 换入内存。这里的一个问题是，进程 A 在换出后又被换入时，是不是一定要再次被加载到内存的同样的位置？

如果没有任何存储管理机制直接使用物理内存的话，毫无疑问程序一定要被放置到与先前相同的位置，不然就会导致寻址相关的一系列问题。但这太不灵活了，得想个办法才行。这个办法应该达到这样的效果：

程序中相同的地址（虚拟地址）可以对应不同的物理地址，以便在被换入内存时可以根据情况的不同被放置在不同的位置，而进程本身对此可能毫无所知。

能有一个方便的函数  $f$ ，改变其中一个参数（假设为  $x$ ），就能让同一个虚拟地址映射到不同的物理地址。

分页机制恰好就是这两个条件的满足者，那个方便的参数  $x$ ，在 80386 中，就是寄存器 CR3。

说起来这只是使用分页机制比较直观的一个原因，其实若想深究其由来的话，免不了要从操作系统的发展史来看，从 IBM 的 OS/MFT 到 OS/MVT 一直到现在我们看到的基于 Intel IA32 架构的成熟的操作系统，存储管理其实是从最简单的固定分区慢慢一步步发展到今天的。这让我想起了大学时我的一位老师的话：“科学总是这样，后人踩着前人的肩膀，不断前行。”

### 段式存储

我想理解分段机制应该是比理解分页机制要容易一些的，因为毕竟分页机制对于程序员是透明的，而分段对于程序员来说就是可见的了。事实情况也是，分页机制要更接近底层，而分段机制则比较高级，它更接近于人的思维方式。

比如，在一个进程中，我们完全可以把代码和数据放在不同的段中，每个段中的只包含一个类

型的对象，于是这个段就可以有针对这种特定类型的合适的保护。

其实分段机制的目的也正在于此，正如《操作系统：设计与实现》一书中所提到的：分段机制“允许程序和数据能够划分为逻辑独立的地址空间以帮助实现共享和保护”。

## 总结：形而上者谓之道

由于分页机制的存在，程序可以不必关心实际上有多少物理内存，好像提供了一个不依赖于硬件（物理内存）的平台，而在此基础之上，我们通过分段机制，从逻辑上对存储器进行管理，比如以权限为工具，进行共享和保护。不知道你发现没有，从思想层面上来看，其实有很多事情都具有相似的两层机构，比如操作系统和应用程序——操作系统为应用程序构造了一个设备无关的平台，让应用程序在它基础上尽情地发挥；再比如低级语言与高级语言——高级语言把程序员从与硬件相关的指令操作中解脱出来，从而可以更加关注逻辑；再再比如 Java 虚拟机和 Java 语言——前者为后者提供了一个操作系统无关的虚拟机使得后者实现了跨平台。这些类似的两层逻辑都是先将底层的细节屏蔽掉，以便第二层能够更方便地关注问题的重点。

我很欣赏凤凰卫视阮次山的一句话，“许多看来不相关的事，其实都是相互有关联的。”我们的类比见证了这一点。那么引申开来，这样的思维方式一旦被我们融会贯通，就完全能在更多方面得到我们的应用。形而上者谓之道，不要被缤纷的表象所迷惑，其实，道理往往都是相同的。

## 零距离进程调度

### 真理的后院

真理看起来总是很简单，许多世界上最伟大的真理，比如勾股定理或者牛顿定律，在我们上中学的时候就已经了解。然而之所以成为伟大的真理，当然不是因为它们简单，事实情况是，这些读起来简单的道理，发现它们是多么的难。

就好比你今天读到了进程调度算法，无论是哲学家进餐问题，还是理发师睡觉问题，读起来都很容易，它们之所以经典，也恰恰是用简单易懂的类比解决了复杂的问题。但是我们相信，发现这些解决方案，无疑是程序世界中的经典。

那么我们应该仔细思考的是，这些解决方案是怎么被想出来的？我们能否因这些智慧之光的照耀而得到启示？我觉得想到这些解决方法需要有两个要素，一是对发生的事情足够地熟悉，熟悉到每个细节都有很好的了解；二是触类旁通的能力，从细节中抓住本质，并从生活经验中得到启示。相对于看上去简单的真理，做到这两点无疑非常地难，我习惯于把这两点称为“真理的后院”，不易看到，却广阔得多。

在学习这些方法时，很多时候我们一扫而过，我们好像已经读懂又好像什么也没有得到，我们浮在表面，像是雾里看花。

一切源于我们对它们没有感性的认识，我们甚至不知道一个进程在操作系统中是怎样的表现形式。前人讲，勿在浮沙筑高台，我想我们只有认真回顾前辈们走过的路，仔细体会他们当时遇到的困难和思路的历程，才有可能获得他们当初的一些灵感。因为他们当初也是在不断实践的过程中不断完善原有的思想。

在我试图了解进程调度时，我首先想到的是看一下前人是“怎么做”的，他们用怎样的方式实现了进程以及进程调度，亲眼见到之后的感觉是不一样的。那么今天，让我们从最底层开始，对进程作一次零距离的接触。

### 今天不提微软

不知何时起，鄙视微软变成了一种时尚，即便你从来没离开过 Windows，即便你哪怕偷偷动用本来打算给女朋友买裙子的钱还是会去买那只昂贵的微软闪灵鲨，嘴里面还是喜欢说上几句，“该死的微软”，虽然心里面此时可能并没有想到微软是多么可恶。

不过无论微软是否可恶，当你想近距离接触操作系统，你想到的一定首先是 Linux，没错，看不到源代码的 Windows，如今终于有了让你觉得可恶的理由，那好，今天，我们彻底地时尚一次，今天，我们不提微软。

### Linux 0.01

Linux 内核的版本已经升级到了 2.6.x，但那个版本研究起来实在太复杂了，它并不适合我们学习，我们应该先从简单的做起，循序渐进，那么最简单的操作系统内核是什么呢？我想至少有两种选择，一个是 Minix，另一个便是 Linux 0.01，也即 Linus 最初形成的那个版本。说起来 Minix 是 Linux 的

师父，但是由于徒弟 Linux 源代码更容易得到，而且由于网上交叉参考（下文中会有介绍）提供的便利，阅读起来也更方便，我们就以它为例来进行我们的探索。

在本文中，由于篇幅所限，很多代码不能列在其中，如果你感兴趣的话，可以到网上将代码找来，非常容易。在这里，我也顺便给初读 Linux 源代码的读者介绍一点阅读技巧。

## 代码如何阅读

工欲善其事，必先利其器。拿过上万行的代码，打开哪个文件都是复杂的逻辑和不知道哪里来的变量，千万不要害怕，这里介绍两种方法，可以方便地阅读源代码：

### Source Insight

这是一个好用的专门阅读源代码的工具，安装之后新建一个项目，然后将源代码导入，就可以开始阅读了。你可以方便地查找变量的声明，函数的定义等等各种内容，还可以很快找到一个符号都在哪里使用过，这些通过好用的菜单、快捷键等方式就可以实现。

### Cross-Referencing Linux

在线的交叉引用使你可以在不必下载源代码的情况下方便地阅读它们，你不但可以在不同的文件中来回浏览，而且可以方便地比较同一个文件不同版本的区别，另外，它还支持多种方式的搜索。

## Linux 启动过程概述

引导扇区我们已经讨论过了，对于计算机的启动我们不再感到陌生，Linux 的 Boot Sector 代码是 boot/boot.s，它做了几次代码的移动，还是蛮有趣的。

开始执行后，它首先把自己（512 字节的引导扇区代码）移动到 0x9000:0000 处，然后把系统代码放到 0x1000:0000 处，再然后把系统代码移动到 0x0000:0000 处，由于每次移动 64K，所以源和目的不会重叠。经历了三次搬家，终于安顿好了，这时跳到保护模式，到 0x0000:0000 处开始执行系统代码。

啊，原来是这么回事，你终于明白了，一个小的引导扇区加载一个大的内核，然后跳转到内核中继续执行，原来就这么简单，是的，就这么简单。现在我们就到 0x0000:0000 处看看 Linux 继续在做些什么。

现在内存 0x0000:0000 处对应的代码，实际上就是 head.s，它更新了 IDT 和 GDT，建立了分页机制，然后就跳转到 main() 函数了。

函数 main() 在 init/main.c 中，啊，感觉重见天日，呵呵，熟悉的 C，此时显得尤其可爱。

main() 很短，看上去结构明了，先是做了若干初始化的工作，然后开中断，执行一个函数：move\_to\_user\_mode()。看名字就大体知道这一句做什么了，它执行完之后程序便完成 ring0 到 ring3 的层级转移，进入用户模式执行。

好了，最激动人心的时刻终于来到了，因为第一个进程随着 fork 系统调用的执行而产生，它便是 init 进程。自此之后，Linux 的车轮便开始转动不停了。

## 进程调度——从中断开始

好，让我们闭上眼睛想象一下进程调度的过程。现在有许多许多进程正在运行着，但在单 CPU

系统中，在同一时刻只有一个进程有 CPU 的使用权而其它处于待命状态。我们假设正在执行着的是进程 A，这时中断发生了，可以预见，进程 A 的状态先被保存起来，然后系统决定那个进程应该被恢复，比如说是 B 吧，然后将进程 B 恢复运行。

这样看来应该挺简单的，那么我们就来一步一步地分析一下。

首先来看时钟中断处理程序：

---

```
145 _timer_interrupt:
146     push %ds           # save ds,es and put kernel data space
147     push %es           # into them. %fs is used by _system_call
148     push %fs
149     pushl %edx          # we save %eax,%ecx,%edx as gcc doesn't
150     pushl %ecx          # save those across function calls. %ebx
151     pushl %ebx          # is saved as we use that in ret_sys_call
152     pushl %eax
153     movl $0x10,%eax
154     mov %ax,%ds
155     mov %ax,%es
156     movl $0x17,%eax
157     mov %ax,%fs
158     incl _jiffies
159     movb $0x20,%al     # EOI to interrupt controller #1
160     outb %al,$0x20
161     movl CS(%esp),%eax
162     andl $3,%eax       # %eax is CPL (0 or 3, 0=supervisor)
163     pushl %eax
164     call _do_timer     # 'do_timer(long CPL)' does everything from
165     addl $4,%esp       # task switching to accounting ...
166     jmp ret_from_sys_call
```

---

表一（节自 kernel/system\_call.s）

看起来中断发生后系统一开始实在忙着保存各个寄存器的内容。在重新设置了几个段寄存器值并置中断命令控制器的 EOI 位后，系统调用了 do\_timer 函数，参数是进程 A 的 CPL。让我们来看一下 do\_timer()：

---

```
159 void do_timer(long cpl)
160 {
161     if (cpl)
162         current->utime++;
163     else
164         current->stime++;
165     if ((--current->counter)>0) return;
166     current->counter=0;
167     if (!cpl) return;
168     schedule();
169 }
```

---

表二（节自 kernel/sched.c）

函数中出现的 `current` 是指向当前进程数据结构的指针，里面的成员 `utime` 和 `stime` 分别用来表示用户态和内核态运行的时间，`counter` 用来描述进程运行时间片，它是递减的。函数根据 `CPL` 的值来决定应该增加 `utime` 还是 `stime`，并且决定是否进行进程调度。如果中断是在内核态发生，调度函数 `schedule()` 是不予执行的。

现在的中断是在进程 A 运行时发生，`CPL` 为 3，`schedule()` 被执行。说起这个 `schedule()` 函数，且不说它在整个 Linux 中有多重要，单是写在函数上面的注释就很有趣：

'`schedule()`' is the scheduler function. This is GOOD CODE! There probably won't be any reason to change this, as it should work well in all circumstances (ie gives IO-bound processes good response etc). The one thing you might take a look at is the signal-handler code here.

Linus 当初如此的自信！记得第一次看它的时候我的眼球一下子被吸引住，虽然后来的版本中此函数并没有像 Linus 说的那样无需修改，但每次看到这里我还是会唏嘘一声。

好了让我们看看这个函数的内容：

---

```
68 void schedule(void)
69 {
70     int i,next,c;
71     struct task_struct ** p;

    /* 处理信号（代码略去） */

    ⋮

    /* 选择下一个应运行的进程（代码略去），
    这里我们假设是进程 B */

    ⋮

104     switch_to(next);
105 }
```

---

表三（节自 `kernel/sched.c`）

这个函数主要分为三个部分，首先处理信号，然后选择下一个应运行的进程，再然后调用 `switch_to()`，对信号的处理我们先不管它，选择下一个进程的算法我们稍后再作讨论，现在我们先要把进程切换的线索理顺，看一下 `switch_to(next)` 是怎么回事：

---

```
168 #define switch_to(n) {\
169 struct {long a,b;} __tmp;\
170 __asm__( "cmpl %%ecx,_current\n\t" \
171          "je 1f\n\t" \
172          "xchgl %%ecx,_current\n\t" \
173          "movw %%dx,%1\n\t" \
174          "ljmp %0\n\t" \
175          "cmpl %%ecx,%2\n\t" \
176          "jne 1f\n\t" \
177          "clts\n\t" \
```

```

178     "l:"\
179     ::"m" (*&__tmp.a),"m" (*&__tmp.b),\
180     "m" (last_task_used_math),"d" _TSS(n),"c" ((long) task[n]);\
181 }

```

表四（节自 include/linux/sched.h）

找到声明你才发现，`switch_to` 原来是个宏，而且是用 AT&T 汇编编写的宏，看上去是有一点晦涩难懂啊，不要紧我们一点一点看。“d” `_TSS(n)`把 `_TSS(n)`赋给 `edx`，“c” `((long) task[n])`把 `&task[n]`赋给 `ecx`。`_TSS(n)`又是一个宏，执行结果是进程 B 对应的 TSS 的选择子。

开头就比较 `current` 和 `&task[n]`，如果相等直接退出，这好理解，如果要恢复的进程就是当前进程的话，不用进行切换。下面紧接着交换 `ecx` 和 `current`，然后让 `__tmp.b` 等于 `_TSS(n)`，再然后来了一个长跳转。`ljmp (*&__tmp.a)`？你可能被弄糊涂了，是啊，这个地方有点奇怪，但正是这条奇怪的指令完成了任务切换，详细情况是这样的：

在通过 TSS 进行任务切换时，执行的是类似这样的指令：

```
jmp TSS_SELECTOR:OFFSET
```

但是由于 TSS 中已经通过 `cs` 和 `eip` 指定了入口点，所以 `OFFSET` 变得多余，会被丢弃掉。结合这里的代码，我们可以想到，`__tmp` 其实就是 `TSS_SELECTOR:OFFSET`，`__tmp.a` 是 `OFFSET`，`__tmp.b` 是 `SELECTOR`，由于 `OFFSET` 是被丢弃的，所以 `__tmp.a` 值是什么无所谓，而 `__tmp.b` 已经事先被我们赋值为进程 n 对应的 TSS 的选择子，所以这一奇怪的语句原来是通过 TSS 的跳转，执行之后，进程 `task[next]`（进程 B）就被恢复了。在跳到进程 B 之前，当前各个寄存器的值会被保存到当前的 TSS 中。

说到这里你可能有点奇怪，啊？没了？跳到进程 B 了，那下面的代码还写了干嘛？其实我们可以想象一下，当进程 B 运行时发生了时钟中断，`schedule()`函数选择了进程 A 作为下一个被恢复的进程，在执行同一个 `ljmp` 的时候，不就跳回到这里来了么？说到底，内核态的这一段程序本质上还是进程 A 的一部分。当然，它同时也是其它各个进程的一部分。

好的，我们再来看一看从别的进程跳回到进程 A 之后的情况。程序继续执行，我们看到程序下一步跳到了 `ret_from_sys_call`。

```

82 ret_from_sys_call:
   /* 进行若干判断以及处理过信号，此处略去 */
126 3:    popl %eax
127      popl %ebx
128      popl %ecx
129      popl %edx
130      pop %fs
131      pop %es
132      pop %ds
133      iret

```

表五（节自 kernel/system\_call.s）

在做完一系列信号处理等操作之后，终于要回到用户态中了，弹出已保存的寄存器，执行指令 `iret`，正式回到用户态继续执行进程 A。

其实在后面的版本中，基于对运行效率的考虑，Linux 背叛了 Intel，进程的切换机制发生了很大的变化，进程的状态不再保存在 TSS 中，切换时也不再通过 TSS 来进行，TSS 届时除了用于保存内存 `ss` 和 `esp` 之外再无其它用处，而且到时候 TSS 只有一个，而不再是像这里一样每个任务都有一个。

这实际上与 Intel CPU 的设计初衷是相违背的，因为设计者当初一定是希望每个任务有单独的 TSS，进程的描述尽在其中。

## Minix 与 Linux: 师父与徒弟

你应该知道，当初 Linus 开始写 Linux 的时候，用的参考书便是 Andrew S. Tanenbaum 所著的《操作系统：设计与实现》，学习的源代码就是书中的 Minix。

如果你看过 Minix 的源代码，你会发现其实很有趣，因为在 Minix 中，进程切换也不是通过 TSS 这种硬件方式来进行，而是直接将各寄存器的值 push 到进程表中，等到恢复运行时再 pop 出来，最后用一个 iretd 指令彻底回到用户态恢复进程运行。

为什么 Linus 当时没有学习师父的做法呢？我想可能 Linus 最初也是想顺应 Intel 设计的初衷吧。但最终在这一点上，Linux 还是跟师父又走到了一起，使用了相同的切换机制。

我开始还怀疑是 Minix 后来的版本发生了改变，但当我找到 Minix1.1，我发现 Minix 随着版本的演变任务切换的方式却没有变过。

殊途同归，这很有趣。

## 调度算法

刚才被我们略过去的调度算法，现在回头来看一下：

```
...
68 void schedule(void)
69 {
85 /* this is the scheduler proper: */
86
87     while (1) {
88         c = -1;
89         next = 0;
90         i = NR_TASKS;
91         p = &task[NR_TASKS];
92         while (--i) {
93             if (!*--p)
94                 continue;
95             if ((*p)->state == TASK_RUNNING && (*p)->counter > c)
96                 c = (*p)->counter, next = i;
97         }
98         if (c) break;
99         for(p = &LAST_TASK ; p > &FIRST_TASK ; --p)
100             if (*p)
101                 (*p)->counter = ((*p)->counter >> 1) +
102                     (*p)->priority;
103     }
```

```
104     switch_to(next);
105 }
...
```

表六（节自 kernel/sched.c）

这段让 Linus 非常自信的程序，用的算法其实很简单，它先是比较每个进程的 counter，找出最大的一个（假设为进程 X），只要 X 的 counter 大于零，则下一个被恢复的进程就确定为 X。不然则每一个进程的 counter 被重置，计算公式为： $\text{counter} = \text{counter} / 2 + \text{priority}$ 。看得出，priority 越大，进程被分到的执行时间就越多，这也正是实际情况中所需要的。

## 小结

好了，到此，对于 Linux0.01 的进程调度你一定有了一个大致轮廓，总结一下是这样的：

1. 进程状态保存在 TSS 中
2. 进程的切换通过 TSS 进行
3. 进程的调度使用简单的基于优先级的算法

Tanenbaum 一直坚持不对 Minix 的代码做扩展，就是为了保持其简洁的特性以利于学习。如今的 Linux 内核已经庞大得不利于学习了，好在我们还能翻出这些旧帐来慢慢研究，以便让入门显得不那么困难。随着版本的升高，Linux 进程调度变得越来越复杂，但有了对这个框架的了解，运用循序渐进的方法，我想再复杂的机制也不是问题。

## 勿以善小而不为

进程调度听起来是很深奥的，可到这里你可能已经发现，涉及到的代码并不是很多，至少主干部分的流程还是很明晰的，进一步我们可以想到，又有多少事情是真正困难的呢？可能我们缺少的仅仅是一个突破口，或者，仅仅是那么一点点的实践精神。

今天我们就以进程调度作为突破口，窥视了一下 Linux 进程的真实面目，其实如果深究下去，你更可以发现进程管理中的同步机制等内容在实践中有着广泛的应用，而且进程管理本身对于我们深刻地理解线程这个概念也有着很好的指导作用。这一切，只需要稍稍花费一点时间阅读一下随处可见的源代码就能做到。