操作系统引导探究 (Version 0.02)

哈尔滨工业大学 计算机体系结构实验室 谢煜波

(xieyubo@126.com)

Version 0.02 修改记录:

对与 GDT 有关的段描述符方面的描述进行了修订，更正了上一个版本中出现的一些错误，增加了一些描述，使其更完善。

与上个版本中不同的地方均用红色标记。

前言

本篇文章并不旨在完整的讨论一个多引导系统程序怎样去引导不同的操作系统，而只打算从编写操作系统的角度出发，谈谈计算机怎样从加电开始，从无到有，将操作系统运行起来，在其中将尽量详尽的描述从实模式到保护模式的过渡，目的只在于能将所学与广大爱好者共享，为希望开发操作系统的朋友留下一点资料，也为自己留下一点心得。

本篇文章将以开发中的 pyos 系统引导程序为例，pyos 是一个正在开发中的实验型操作系统，它并不打算以目前任何一种运行中的操作系统为模式，而只想通过自己编写一个从头到尾的操作系统来学习知识，积累技术，如果你有兴趣，非常欢迎你的加入！

本篇纯属学习过程中的一点心得体会，如果你发现其中有错误或不当之处，非常希望你来信指教。

一、计算机从加电开始都做了什么？

当机算机的电源键被按下时，同这个键相联的电信号线就会送出一个电信号给主板，主板将此电信号传给供电系统，供电系统开始工作，为整个系统供电，并送出一个电信号给 BIOS，通知 BIOS 供电系统已经准备完毕。随后 BIOS 启动一个程序，进行主机自检，主机自检的主要工作是确保系统的每一个部分都得到了电源支持，内存储器、主板上的其它芯片、键盘、鼠标、磁盘控制器及一些 I/O 端口正常可用，此后，自检程序将控制权还给 BIOS。接下来 BIOS 读取 BIOS 中的相关设置，得到引导驱动器的顺序，然后依次检查，直到找到可以用来引导的驱动器（或说可以用来引导的磁盘，包括软盘、硬盘、光盘等），然后调用这个驱动器上磁盘的引导扇区进行引导。BIOS 是怎么知道或说分辨哪一个磁盘可以用来引导的呢？

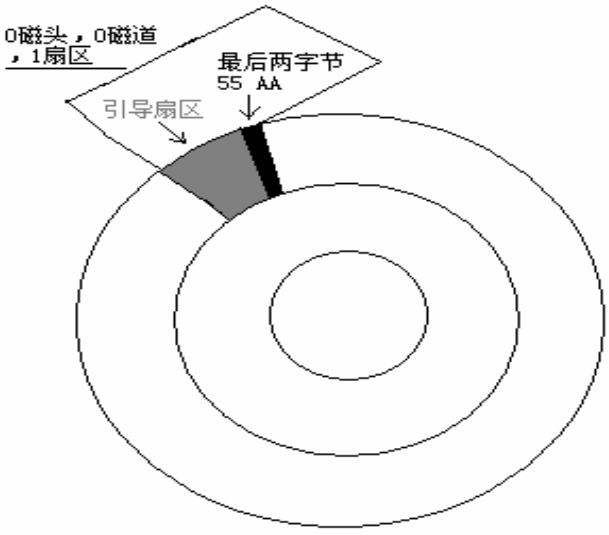
二、认识引导程序

BIOS 将磁盘的第一个扇区（磁盘最开始的 512 字节）载入内存，放在 0x0000:0x7c00 处（见图三），如果这个扇区的最后两个字节是“55 AA”，那么这就是一个引导扇区，这个磁盘也就是一块可引导盘。通常这个大小为 512B 的程序就称为引导程序（boot）。如果最后两个字节不是“55 AA”，那么 BIOS 就检查下一个磁盘驱动器。

通过上面的表述我们可以总结出如下三点引导程序所具有的特点：

1. 它的大小是 512B，不能多一字节也不能少一字节，因为 BIOS 只读 512B 到内存中去。
2. 它的结尾两字节必须是“55 AA”，这是引导扇区的标志。
3. 它总是放在磁盘的第一个扇区上（0 磁头，0 磁道，1 扇区），因为 BIOS 只读

第一个扇区。



（图一）

因此，在我们编写引导程序的时候，我们也必须注意上面的三点原则，符合上面三点原则的程序都可以看作是引导程序，至少 BIOS 是这样认为的，虽然它也许可能是你随意写的一段并没有什么实际意义的代码。

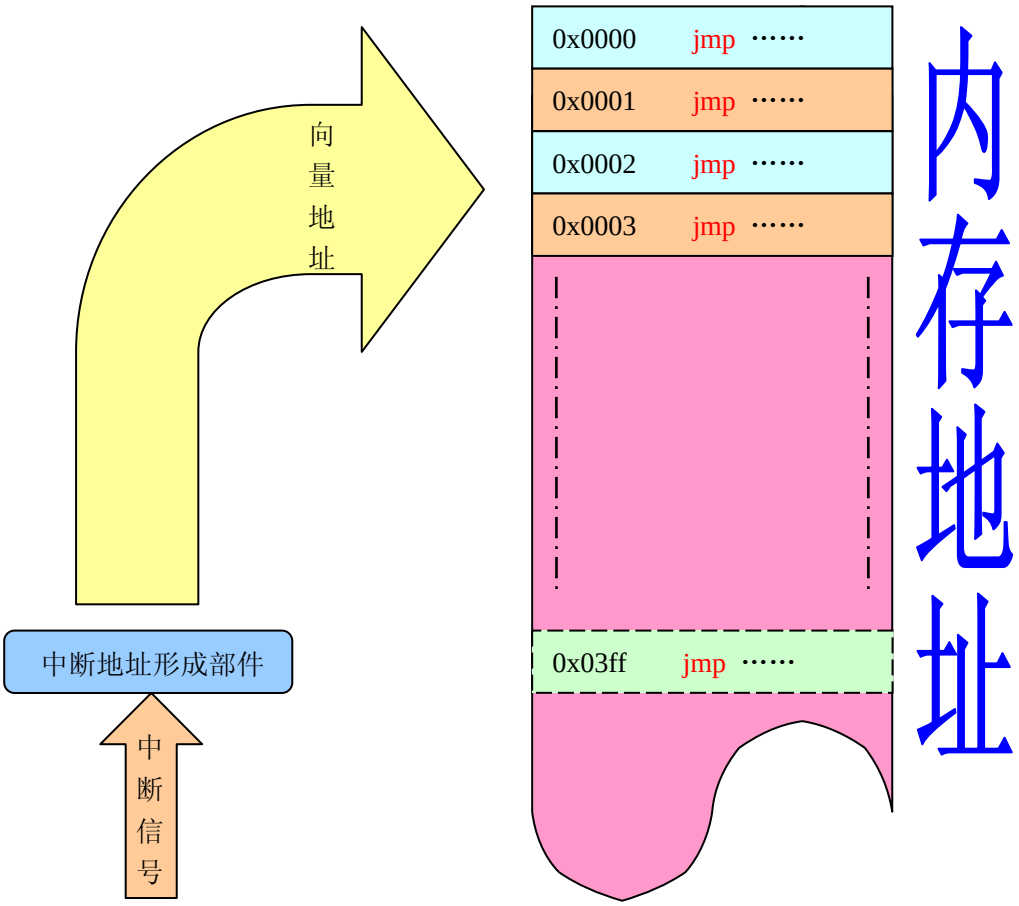
因为 BIOS 一次只读一个扇区也即 512 字节的数据到内存中，这显然是不够的，现在操作系统都比较庞大，因此我们必须在引导扇区里，将存在磁盘上的操作系统的核心部分读进内存，然后再跳转到操作系统的核心部分去执行。

三、通过 BIOS 读磁盘扇区

从上面的描述我们可以知道，引导程序需要将存在于磁盘上的操作系统读入内存，因此这里我们不得不再讲一讲，怎样不通过操作系统（因为现在还没有操作系统）去读取磁盘上的内容。一般说来这有两种方法可以实现，一种是直接读写磁盘的 I/O 端口，一种是通过 BIOS 中断实现。前一种方法是最低层的方法（后一种方法也是在它的基础上实现的），具有极高的灵活性，可以将磁盘上的内容读到内存中的任意地方，但编程复杂。第二种方法是前一种方法稍微高层一点的实现，牺牲了一点灵活性，比如，它不能把磁盘上的内容读到 0x0000:0x0000 ~ 0x0000:0x03FF 处。为什么不能读到此处呢？这里我们将不得不描述一下 CPU 在加电后的中断处理机制。

3.1 BIOS 的中断处理

中断是什么？相信学过计算机的人都不会陌生，如果你对中断一点都不了解建议你翻看一下《计算机组成原理》（高等教育出版社　唐朔飞），上面有非常详尽的描述，而一般的汇编教材也多有谈及，因此这里只打算讲讲 BIOS 对中断的处理。



（图二）

由上图（图二）我们可以清楚的看到，当中断信号产生时，中断信号通过“中断地址形成部件”产生一个中断向量地址，此向量地址其实就是指向一个实际内存地址的指针，而这个实际内存地址中往往安排一条跳转指令（jmp）跳转到实际处理此中断的中断服务程序中去执行。这一块专门用于处理中断跳转的内存就被称为中断向量表。在内存中，这块中断向量表被放在什么地方呢？而实际的中断处理程序又在什么地方呢？

3.2 系统的内存安排（1M）

要回答上面的两个问题，我们需要看看系统中内存是怎么安排的。在 CPU 被加电的时候，最初的 1M 的内存，是由 BIOS 为我们安排好了的，每一字节都有特殊的用处。

0x00000~0x003FF：	中断向量表
0x00400~0x004FF：	BIOS 数据区
0x00500~0x07BFF：	自由内存区
0x07C00~0x07DFF：	引导程序加载区
0x07E00~0x9FFFF：	自由内存区
0xA0000~0xBFFFF：	显示内存区
0xC0000~0xFFFFF：	BIOS 中断处理程序区

(图三)

由上图我们现在可以很方便的问答上面提出的两个问题。由于 0x00000~0x003FF 是中断向量表所在，因此不能将磁盘中的操作系统读到此处，因为这样会覆盖中断向量表，就无法再通过 BIOS 中断读取磁盘内容了。你也许会说：我是先调用中断，再读的啊。但事实上 BIOS 在读的过程中自己会多次调用其它中断辅助完成。

3.3 利用 BIOS 13 号中断读取磁盘扇区

有了前面的描述作为基础，下面我们可以正式描述怎样通过 BIOS 中断读取磁盘扇区了。要读取磁盘扇区，我们需要使用 BIOS 的 13 号中断，13 号中断会将几个寄存器的值作为其参数，因此，我们在调用 13 号中断的过程中需要首先设置寄存器。那么当怎样设置寄存器呢？会用到哪些寄存器呢？请往下看：

AH 寄存器：存放功能号，为 2 的时候，表示使用读磁盘功能
DL 寄存器：存驱动器号，表示欲读哪一个驱动器
CH 寄存器：存磁头号，表示欲读哪一个磁头
CL 寄存器：存扇区号，表示欲读的起始扇区
AL 寄存器：存计数值，表示欲读入的扇区数量

在设置了这几个寄存器后，我们就可以使用 int 13 这条指令调用 BIOS 13 号中断读取指定的磁盘扇区，它将磁盘扇区读到 ES:BX 处，因此，在调用它之前，我们实际上还需要

设置 ES 与 BX 寄存器，以指出数据在内存中存放的位置。

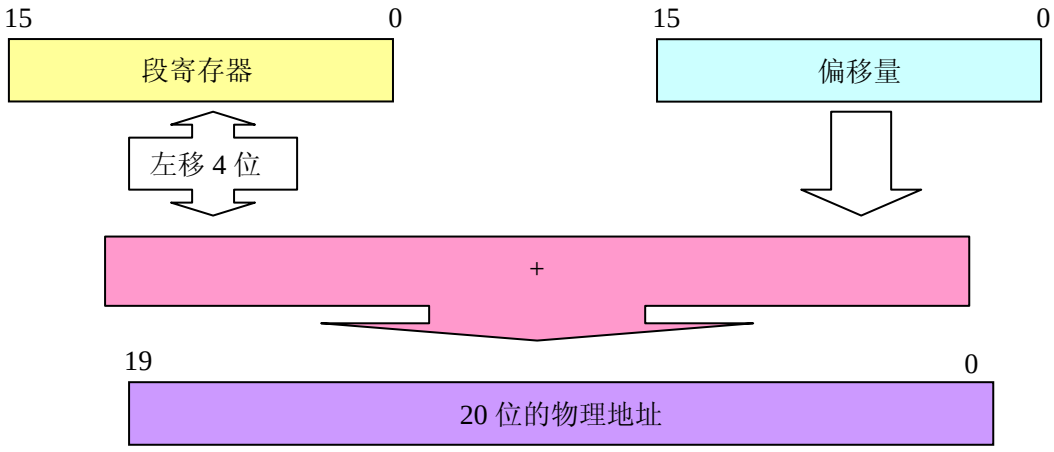
四、保护模式下，段模式内存地址的访问

写程序离不开对内存的访问，然而在保护模式下内存的访问与在实模式下内存的访问完全不同，这里我们将详细描述一下保护模式下内存的访问方法。当然，这里并不打算完整的介绍保护模式下所有的内存访问方法与机制，只介绍从实模式转到保护模式下所需要进行的转换，完整的内存访问请你参见《Intel 用户手册》。当然，随着 pyos 的实验进行，我也会在后面的实验报告与心得体会中渐渐描述。

4.1 实模式下的内存访问

计算机在加电时，处于“实模式”，在计算机中有一个 CR0 寄存器，又称为 0 号控制寄存器，在这个寄存器中，最低位也即第 0 位，被称为 PM（Protected Modle：保护模式）位，当它被清零的时候表示 CPU 在“实模式”下工作，当它被置位的时候表示 CPU 在“保护模式”下工作。在计算机加电的时候，它是被清零的，所在这个时候的计算机，处于“实模式”。

“实模式”下的内存访问通过段寄存器与偏移量构成，比如前面描述中常常出现的 0x:0000:0x0001 就是一个实模式下的内存地址。分号前面的值表示段寄存器中的值，分号后面的值表示偏移量，实际物理地址的形成如下图所示：



（图四）

然而在保护模式下，内存地址却不是如上图所示的方法形成的。那么它又是怎样形成的呢？

4.2 保护模式下的内存地址形成

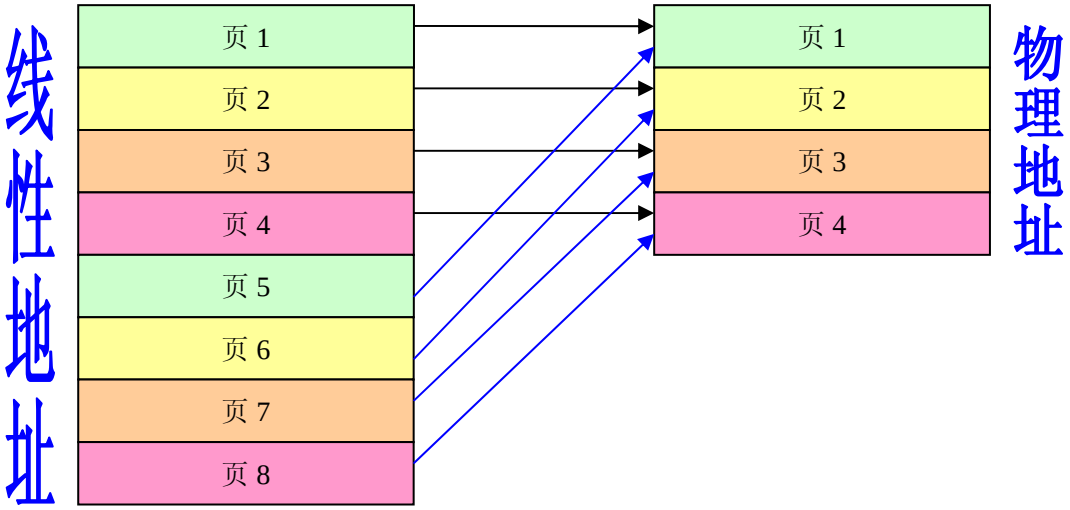
保护模式下内存地址就复杂多了， 我们首先要分清三个概念：逻辑地址、线性地址与物理地址。物理地址很好理解，逻辑地址也好理解，就是程序所使用的地址。那么什么是线性地址呢？

其实如果不使用分页机制的话，线性地址就是物理地址，它与物理地址是一一对应的，线性地址 0，也就是物理地址 0。但我们知道，32 位的 CPU 拥有 32 根地址线，也就是可以访问：

$$2^{32} = 4GB$$

的内存空间，这实在是一个太大的空间了！现在很少有机器的物理内存能有这么大。那怎么

在有限的物理空间中使用 4GB 的空间呢？人们把物理内存分成许多页，同样也把整个 4GB 的线性地址空间分成大小相同的许多页。在线性地址空间中，当某些页被使用的时候，某些页可能没有被使用，操作系统可以让 CPU 将没有被使用的页调出物理内存（存放在磁盘的某个地方，以备需要的时候再次调入），而把需要使用的页调入，这样，虽然物理内存空间有限，但也几乎可以使用所有的线性地址空间了。这就称为从线性地址到物理地址的映射，这是一个多对一的映射，也就是说多个线性空间中的页对应一个物理空间中的页，希望下面一幅图能有助于你理解这样的分页机制。

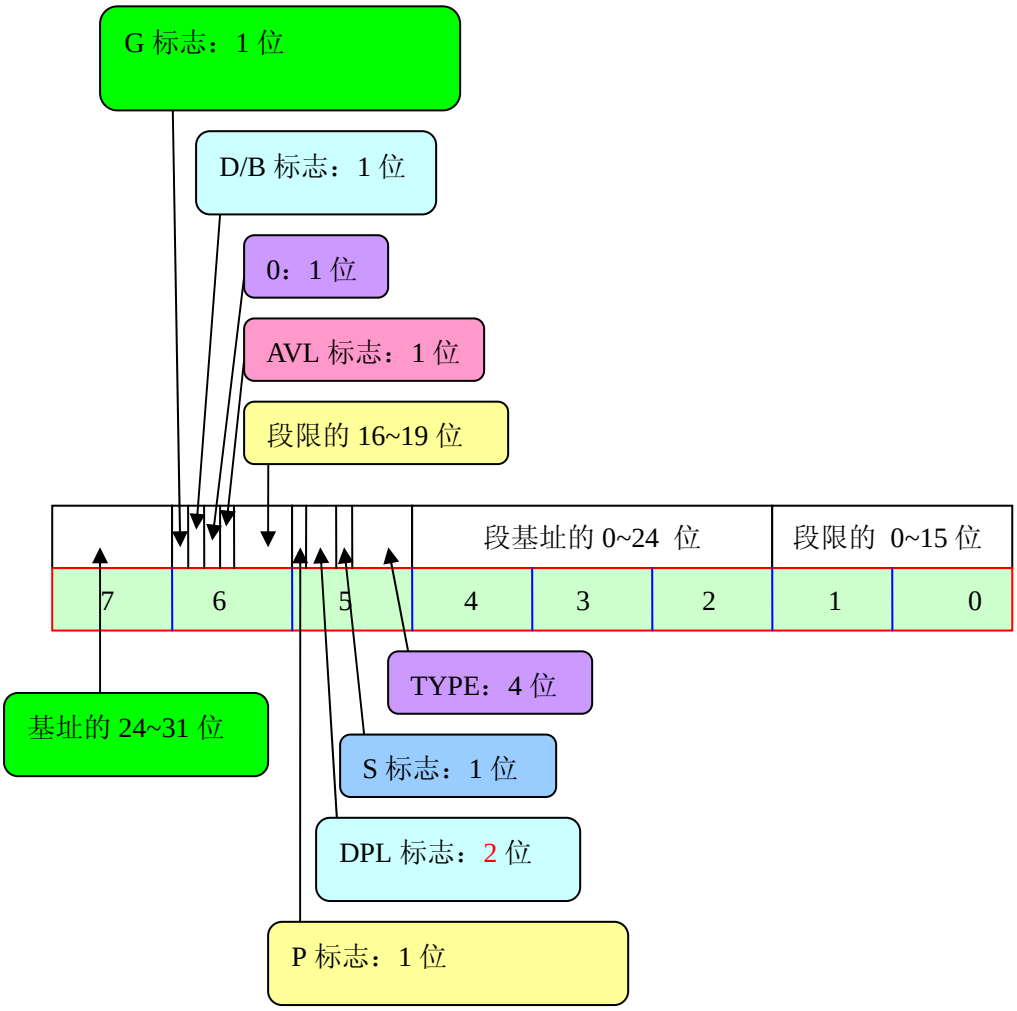


（图五）

上面是一种最简单的映射方式，术语称作“直接相连”映射，它大约只能用来说明问题，而在一个实际的操作系统中通常是“全相联相连”映射，也就是说线性地址中的页可以是映射到物理地址中的任何一个页中，只要那块物理地址空间现在是空闲的。不过，通过上图也能说明问题，当线性地址中的页 5 需要被访问时，CPU 通过地址映射机制将其转换到物理地址，发现其对应物理地址中的页 1。于是 CPU 会产生一个所谓的缺页中断来通知操作系统进行处理，操作系统相应这个中断，并在中断服务程序中将物理地址页 1 中的内容放到磁盘上的一个地方（虚拟内存），然后将线性地址中的页 5 载入物理内存页 1 中。这里就当可以比较明显的区别什么是线性地址，什么是物理地址了。

然而，当不使用分页机制的时候，线性地址就会被 CPU 当做物理地址来使用，线性地址会被直接放在 CPU 的地址信号线上。不过，在编写应用程序的时候，我们通常使用的却是另一种地址——逻辑地址，从逻辑地址到线性地址也存在着与上述机制类似的一种映射机制，不过这个机制常常称为“段模式”，它是由操作系统与 CPU 硬件共同完成的。操作系统的任务就是分配映射表，而 CPU 硬件的任务就是按着映射表进行映射。而这样的映射表在操作系统编写中又称之为“描述符表”，有两种重要的描述符表，一种是“全局描述符表（GDT）”，另一种是“局部描述符表（LDT）”，这两种表的用途不同，但它们的用法却是近似的，下面我们就来描述一下全局描述符表。

说到表，学过数据结构的人都知道，其实它就是一种数据结构，全局描述符表也是一种数据结构，当这种结构放在一块连续的内存之中就称之为表了。表由表项组成，全局描述符表由它的表项“全局描述符”组成，其实单纯的术语就叫“描述符”，只因为它放在全局描述符表中就成了全局描述符了。这个描述符由 8 个字节组成，下面我们就来看看它的结构：



(图六)

TYPE： 表明此段的类型，4 位中的最高位被清 0 的时候表是它是数据段，相应的余下的三位，从左到右依次为 E、W、A，即数据段的 TYPE 为：0EWA。其中 E 表示向下增长位，置 1 时表示向下增长（这主要是在大小需要动态改变的堆栈段中使用，如果是向下增长的段，动态的改变段的大小限制，会让堆栈空间加到堆栈的底部。如果堆栈的大小不需要改变，那么这个段既可以是向下增长的段，也可以是非向下增长的段）；W 表示可写位，置 1 表示可写；A 表示被访问位（如果 CPU 访问了它，此位将会被置 1）。

4 位中的最高位被置 1 时表示它是代码段，相应的余下的三位，从左到右依次为 C、R、A，即代码段的 TYPE 为：1CRA。其中 C 是表明此代码段是否是一致代码段，如果 C 被置 1，表明此是一致代码段。一致代码段主要是用于特权级访问控制，这在以后的实验报告中会详细论述；R 表明此段是否可读，置 1 表示可读；A 表示被访问位，这与前述一样。

S：为 1 时表示这是一个代码段或数据段描述符，为 0 时表示这是一个系统段描述符。系统段描述符又称为特殊段描述符，包括：局部描述符表（LDT）描述符，任务状态段（TSS）描述符，调用门描述符，中断门描述符，陷阱门及任务门描述符等。

DPL：表示特权级，从 00~11，共 0，1，2，3 四个特权级

P：为 0 是表示此描述符无效，不能被使用

AVL：留给程序员随便用的

D/B：为 0 的时候表示它是一个 16 位的段，为 1 时表示它是一个 32 位的段

G：为 0 时，表示段限的单位是 1 字节，为 1 时表示段限的单位是 4KB，并且段偏移量的最低 12 位将不被检测是否在段限之中。（这一点现在可能不好理解，但我下面马上会解释）。

这里面有两个部份比较有意思，一个是“基址”，一个是“段限”。基址应当比较好理解，它给出的是一个段在线性内存中的起始地址，对于“段限”，顾名思义，就是段大小的限制。不过它有点特别，对于一个段的最大可访问的地址 CPU 是通过下面的公式计算得到的：

段基址 + 段限值 * 段限单位 = 此段最大可访问地址

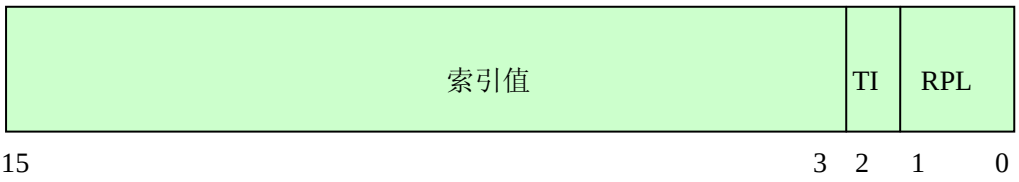
如果一个偏移地址大于了此段最大可访问地址的话，CPU 就将产生一个错误中断，这样一来就可以防止一个程序非法访问另一个程序的内存空间，这对内存起到了保护作用，“保护模式”由此得名。

所以，如果段限是 0，那么此段最大可访问地址就是段的基址，因此，当段限单位为一字节时，此段的段大小就是 1 字节；当段限单位为 4KB 时，因为 CPU 将不检测偏移量的最低 12 位，而这 12 位最大可能为 0xFFF，因此，这时此段的可访问范围就为 4KB，所以：

（段限值 + 1）* 段限单位 = 此段大小

现在我们可以正式开始描述在保护模式下段模式是怎样访问内存的了。这里之所以要强调“段模式”是因为在保护模式下还有一种前面叙述过的内存访问模式——页模式，它负责将线性地址再按某种映射转换为物理地址。“页模式”也是基于段模式的，在不使用它的情况下，线性地址会被直接放到地址线上当做物理地址使用。“段模式”是不可避免的，所谓的“纯页模式”只是将整个线性地址当作一个整段，没有什么方法可以真正绕过“段模式”，因为这是由 CPU 内存访问机制所规定的。本篇只描述段模式。

我们已经知道从程序使用的逻辑地址到线性地址的映射是通过“描述符”来完成的，而“描述符”又是放在描述符表中的，那么，一个描述表中有许多描述符，到底选用哪一个描述符呢？这就由一个索引来决定，这个索引将指出是表中的第几个描述符，这个索引有一个专门的术语来描述，常常称它为“段选择子”。“段选择子”由 2 个字节共 16 位组成，下面，我们就来看看它提供了哪些信息：



（图七）

其中：

RPL：指示出特权级，00~11，共 0、1、2、3 四个特权级，与前述一样。

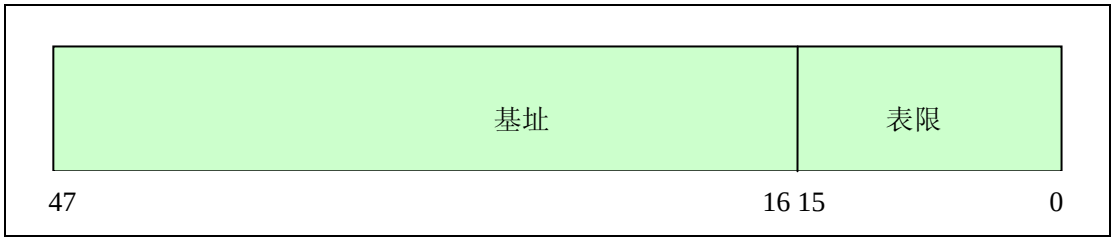
TI：为 0 时表明这是个用于全局描述符表的选择子，为 1 时表明用于局部描述符表。

索引值：用来指示表中第几个描述符。索引值共有 13 位，因此，每张描述符表共可有 8K 个表项，而一个表项如前所述，占 8 个字节，因此一张描述符表最大可达 64K。

不知道大家是否注意到这样一个事实，如果将“段选择子”的最后 3 位置 0，这整个段选择子其实就是一个描述符在描述符表中的偏移量！这里我们可以发现 Intel 的工程师在设计的时候真的是非常精巧，如此的安排，可以使选取一个描述符的速度极大加快，因为将一个段选择子最后 3 位清零后与描述符表的基址相加，就立即可以得到一个描述符的物理地

址，通过这个地址就可以直接得到一个描述符。那么这个描述符表的基址又是放在哪儿的呢？

所为描述符表的基址也就是此描述符表在内存中的起始地址，也即表中第一个描述符所在的内存地址，系统中用两个特殊的寄存器来存放，一个用于存放全局描述符表的基址，称之为“全局描述符表寄存器（GDTR）”，另一个用来存放局部描述符表的基址，称之为“局部描述符表寄存器（LDTR）”，它们的结构如下图所示：



（图八）

其中表限也即表的大小限制，它的使用与前面所描述的段限是类似的，因此，这里就不在描述了。

在保护模式下，以前实模式下的段寄存器还是有用的，不过它不再用来存放段的基址，而是用来存放“段选择子”，它的名字也变成了“段选择子寄存器”，在访问内存的时候，我们需要给出的是“段选择子”，而不是段基址了。

比如，我现在想使用全局描述符表中第二个表项，即其中的第二个“段描述符”，这个“段选择子”就需按如下的方式构成：

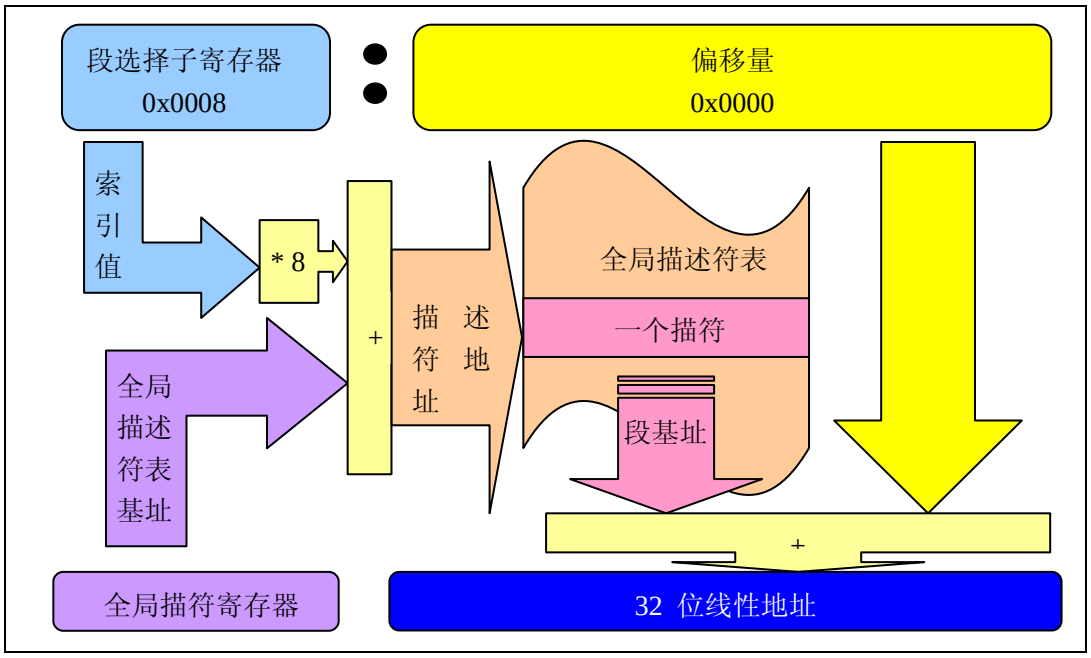
RPL：00，因为我们现在是在写操作系统，工作在 0 特权级

TI：0，我们使用全局描述符

索引值：1，我们使用第二个全局描述符，第一个全局描述符编号为 0，第二个为 1

因此，我们的“段选择子”为：0000 0000 0000 10000，也即 0x0008，因此，对于 0x0008:0x0000 这样一个逻辑地址，在保护模式下就应看成是使用全局描述符段中第二个描述符所描述的段，并且偏移量为 0 的内存地址。

这个逻辑地址的线性地址是怎样形成的呢？请看如下的图示：

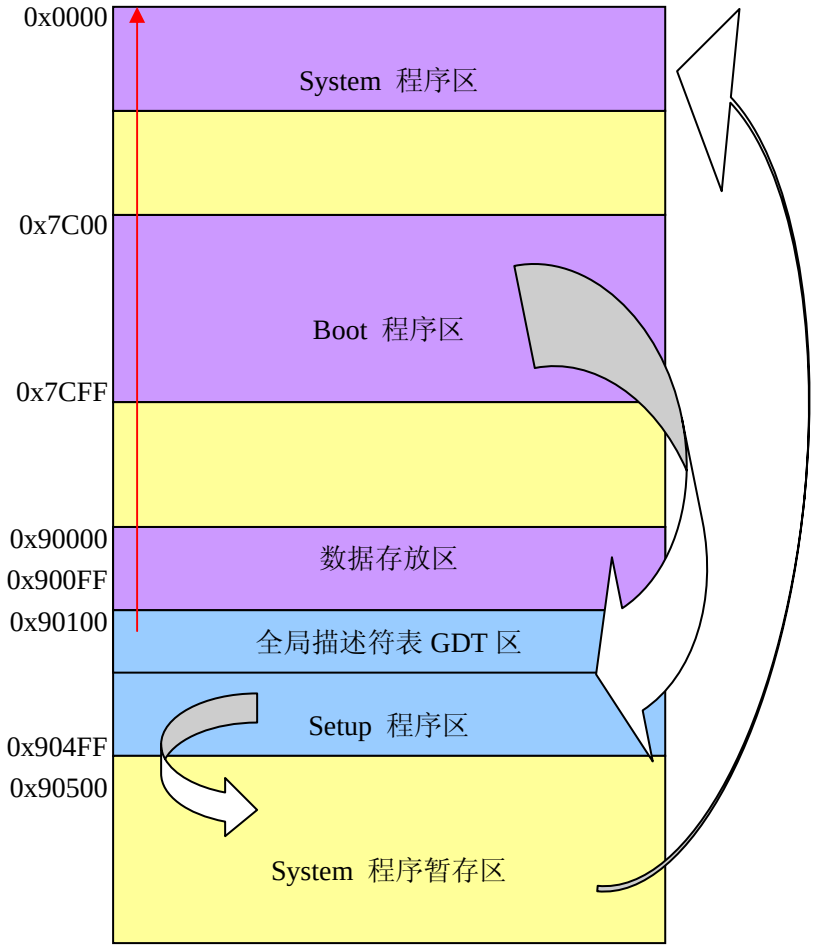


（图九）

相信，从上图中你可以清楚的看出一个逻辑地址是怎样转换为一个 32 位的线性地址的。

五、pyos 引导程序编写

pyos 是一个正在编写中的操作系统，是一个实验中的项目，关于编写的目的与动机我已在前言中谈论过了，这里，仅就此篇所讲述的内容，谈谈 pyos 引导程序的编写，在编写期间参考了 linux 0.11 内核引导程序的编写，不过 pyos 并不是基于 linux 的，就它们的引导程序之间也有许多不一样的地方。下面我们先来看看 pyos 的整个引导区的内存安排：



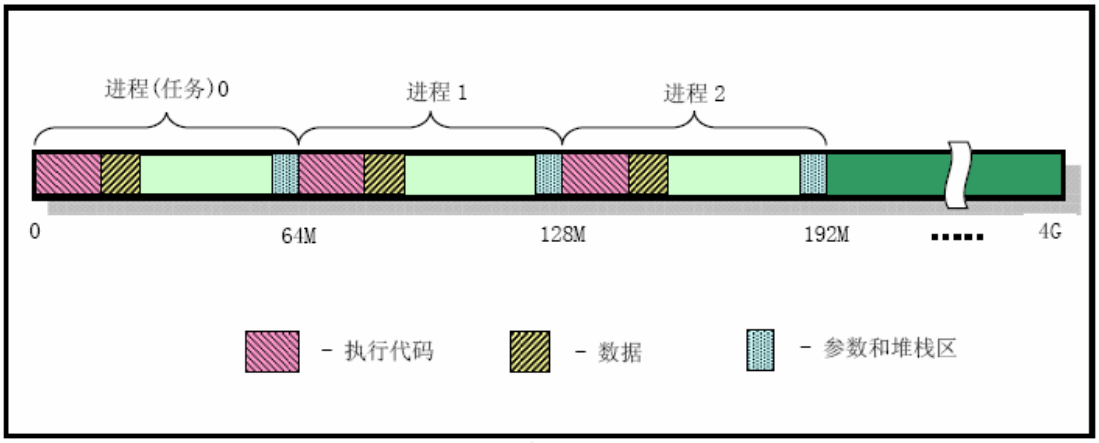
（图十）

上面一幅图就是 pyos 的内存安排图，也是引导程序流程图，pyos 是两级引导系统，首先是 boot 被 BIOS 读入，随后 boot 读入 setup，setup 读入 system 程序到暂存区，然后把 system 程序搬到内存顶部，并建立指向 system 程序所在段的段描述符及建立 GDT，然后切换 CPU 到保护模式，最后跳转到 system 程序中执行，至此 pyos 系统引导完毕，system 程序将是 pyos 真正的系统内核。图中的数据存放区用来存发在 boot、setup、pyos 三者程序间需要传递的参数。

之所以做成两级引导主要是考虑到以后扩展时的方便，各程序间都差不多是独立的，以后可以重写 boot 或者 setup 以提供更多可选择的引导方式。System 程序暂存区是因为如

前所述，不能直接将数据读到中断向量表中覆盖原中断向量表，当数据读完之后，不再调用中断了，才将程序搬到内存顶部覆盖原中断向量表，对于保护模式下的中断向量表，将由 system 程序负责建立，交给 system 使用的是一块完整而干净的内存。

对于 pyos 进程内存安排，准备参照 Linux 0.11 进行，内存安排如下：



（图十一，来源《Linux 0.11 内核代码完全注释》）

一个进程享有 64M 空间，4GB / 64M = 64，也即系统最大进程数为 64。因此一个段的段限为：64MB，每个进程占用全局描术符表中两个描述符，一个为数据段描述符，一个为代码段描述符，段限均为 64MB。

六、pyos 引导程序源代码

下面将提供 pyos 引导程序的全部源代码，因为 system 还未完全完成，因此这里只是让它简单的打印一个字符以示引导工作完成，代码中已有较为详尽的注释，如果仍有不太清楚的地方，可以去 http://purec.binghua.com（纯 C 论坛）操作系统实验专区，查看 pyos 以前的实验报告，上面有非常详尽的注释及相关原理说明，并详细描述了怎样编译及实验。

```
;文件名: boot.asm
;作　者: 谢煜波
;Emailv: xieyubo@126.com
;
;内存分配如下
;内存起始地址为 0x90000
;最大结束地址为 0x9ffff
;最大共 64KB
;所有启动代码在一个段内，方便调用
;启动代码共分两部分，一是 boot，一是 setup，这点照搬 linux 0.11 的设计
;但与其不同的是，boot 不会将自己搬到 0x90000 处，而直接跳到 0x90100 处运行
;0x90000~0x900ff (256B) 系统保留来存放一些从 BIOS 中取出的关键数据
;0x90100~0x904ff (1KB)：此处开始存放 setup，setup 大小为 1KB

[BITS 16]                                                    ;编译成 16 位的指令
[ORG 0x7C00]
;-------------------------------------------------------------------------------
```

```
jmp  Main
;-----------------------------------------------------------------------------------------
;数据定义
MSG                db              "Loading pyos ..."  ;输出信息
                   db              13 , 10 , 0         ;13 表示回车，10 表示换行，
                                                       ;0 表示字符串结束
BOOTSEG            equ             0x0000              ;boot 所在的段基址
SETUPSEG           equ             0x9000              ;setup 所在的段基址
SETUPOFFSET        equ             0x0100              ;setup 所在的偏移量
SETUPSIZE          equ             1024                ;setup 的大小，必须是 512 的倍数
BOOTDRIVER         db              0                   ;保存启动的驱动器号
;-----------------------------------------------------------------------------------------
ShowMessage:
  ;以下程序行为显示输出信息
  mov              ah , 0x0e                          ;设置显示模式
  mov              bh , 0x00                          ;设置页码
  mov              bl , 0x07                          ;设置字体属性

  .nextchar:
  lodsb
  or               al , al
  jz               .return
  int              0x10
  jmp              .nextchar

  .return:
  ret
;-----------------------------------------------------------------------------------------
Main:
  mov              [BOOTDRIVER] , dl                  ;得到启动的驱动器号

  ;以下程序设置数据段
  mov              ax , BOOTSEG
  mov              ds , ax
  mov              si , MSG

  call             ShowMessage                        ;显示信息

  ;读入 setup
  ;从磁盘的第二个扇区读到 0x90100 处
  .readfloopy:
  mov              ax , SETUPSEG
  mov              es , ax
  mov              bx , SETUPOFFSET
```

```
  mov                ah , 2
  mov                dl , [BOOTDRIVER]
  mov                ch , 0
  mov                cl , 2
  mov                al , SETUPSIZE / 512                ;读入扇区数( 2 个共 1KB )
  int                0x13
  jc                 .readfloopy

  ;把启动驱动器号保存在 0x90000 处
  mov                 al , [BOOTDRIVER]
  mov                 [0] , al

  ;跳转
  jmp                 SETUPSEG : SETUPOFFSET
;-----------------------------------------------------------------------------
times 510-($-$$) db 0
db 0x55
db 0xAA
```

```
;文件名: setup.asm
;作   者; 谢煜波
;Emailv: xieyubo@126.com

;此 setup 程序完成 boot 未完成的启动工作，
;包括从 BIOS 中读出系统信息存放在指定位置
;初始化 GDT,LDT 表，完成从保护模式到实模式的转换
;实模式的代码也由此程序读入

[BITS 16]
[ORG 0x0100]
;-------------------------------------------------------------------------------
jmp                 Main
;-------------------------------------------------------------------------------
SETUPSEG            equ                 0x9000
SETUPOFFSET         equ                 0x0100
SETUPSIZE           equ                 1024            ;setup 的大小 1KB,必须是 512 的倍数
SYSTEMSEG           equ                 0x0000
SYSTEMOFFSET        equ                 0x0000
SYSTEMSIZE          equ                 1024            ;SYSTEM 的大小 1KB,此值必须是 512 的倍
数
                                                        ;实际值可以不符

;下面定义临时 GDT 表的描述符
;总共定义三个段，一个空段由 intel 保留，一个代码段，一个数据段
```

```
gdt_addr:
                    dw                  0x7fff          ;GDT 表的大小
                    dw                  gdt             ;GDT 表的位置
                    dw                  0x0009
gdt:
  gdt_null:
                    dw                  0x0000
                    dw                  0x0000
                    dw                  0x0000
                    dw                  0x0000
  gdt_system_code:
                    dw                  0x3fff          ;段限(0x3fff+1)*4KB=64KB
                    dw                  0x0000
                    dw                  0x9a00
                    dw                  0x00c0
  gdt_system_data:
                    dw                  0x3fff
                    dw                  0x0000
                    dw                  0x9200
                    dw                  0x00c0
;-------------------------------------------------------------------------------
;等待键盘控制器空闲的子程序
Empty_8042:
  in                al , 0x64
  test              al , 0x2
  jnz               Empty_8042
  ret
;-------------------------------------------------------------------------------
Main:
  ;初始化寄存器，因为 Bios 中断及 call 会用到堆栈或 ss 寄存器
  ;在 CPU 启动或复位时是由 BIOS 初始化的，而现在进行了段转移，需要我们重新设置
  mov               ax , SETUPSEG
  mov               ds , ax
  mov               es , ax
  mov               ss , ax
  mov               sp , 0xffff

  ;-----------------------------------------------------------------------
  ;从 BIOS 中到底应读出哪些有用信息，现在还不确定，因此暂时跳过此功能块
  ;-----------------------------------------
  ;0x90000 (1B): 保存启动驱动器号，由 boot 程序存入
  ;-----------------------------------------------------------------------

  ;下面读入 system 到 setup 程序的后面
```

```
;因为 0x00000 现在是放 BIOS 中断的地方，因此还不能直接将 system 读到 0x00000 处，
;否则将无法调用 BIOS 中断读入磁盘
.readfloopy:
mov                 ax , SETUPSEG
mov                 es , ax
mov                 bx , SETUPOFFSET + SETUPSIZE
mov                 ah , 2
mov                 dl , [0]
mov                 ch , 0
mov                 cl , 1 + 1 + SETUPSIZE / 512        ;system 所在的启始扇区
                                                        ;第一个 1 是指从 1 开始记数
                                                        ;第二个 1 是 boot 所占扇区数
mov                 al , SYSTEMSIZE / 512               ;读入扇区数( 2 个扇区共 1KB )
int                 0x13
jc                  .readfloopy

;下面将读入的 system 搬移到 0x00000 位置
cld
mov                 si , SETUPOFFSET + SETUPSIZE
mov                 ax , SYSTEMSEG
mov                 es , ax
mov                 di , SYSTEMOFFSET
mov                 cx , SYSTEMSIZE / 4
rep movsd

;下面开始为进入保护模式而进行初始化工作
cli                                                     ;关中断
lgdt                [gdt_addr]                          ;载入 gdt 的描述符

;下面打开 A20 地址线
call                Empty_8042
mov                 al , 0xd1
out                 0x64 , al
call                Empty_8042
mov                 al , 0xdf
out                 0x60 , al
call                Empty_8042

;下面设置进入 32 位保护模式运行
mov                 eax , cr0
or                  eax , 1
mov                 cr0 , eax

jmp dword           0x8:0x0
```

```
;-------------------------------------------------------------------------------
times 1024-($-$$) db 0
```

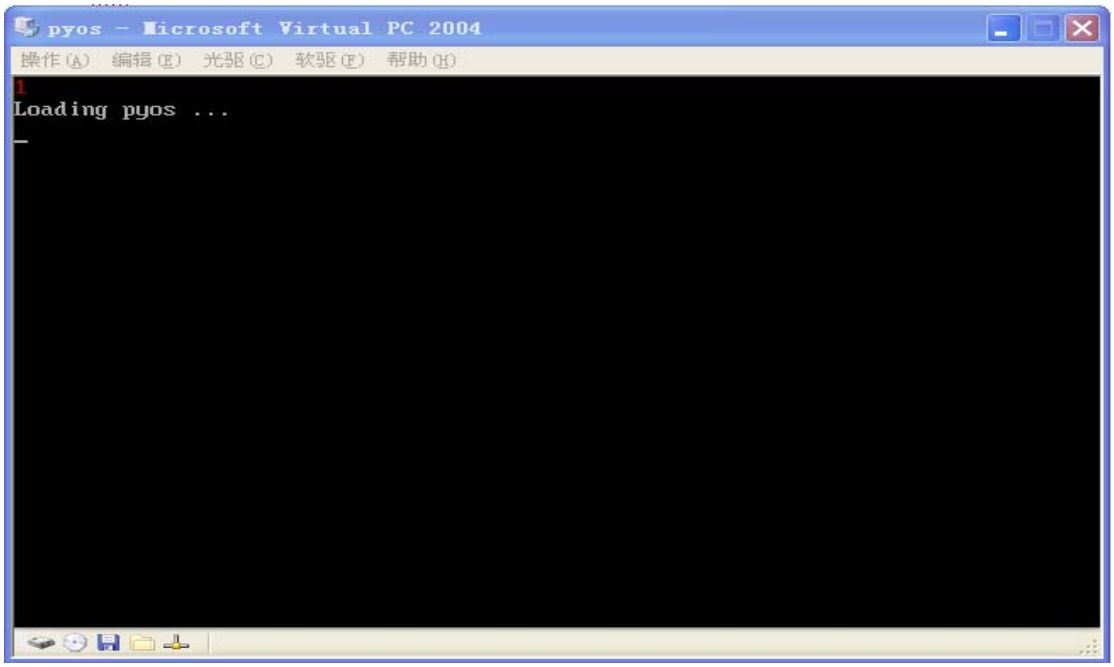
```
;文件名: kernel.asm
;作　者: 谢煜波
;Emailv: xieyubo@126.com

[BITS 32]
[ORG 0x0]
;-------------------------------------------------------------------------------
jmp  Main
;-------------------------------------------------------------------------------
Main:
  ;设置寄存器
  mov                 ax , 0x10
  mov                 ds , ax

  mov                 cl , '1'
  mov                 [0xb8000] , cl
  mov                 cl , 0x04
  mov                 [0xb8001] , cl
  jmp $
```

以上程序中有一个地方本篇及以前的实验报告中也未提到，这就是 A20 地址线的问题，对于有关 A20 地址线的问题，在《Linux 0.11 内核源代码完全注释》中有非常详细的描述，作者还列举了其它几种打开 A20 地址线的方法，并分析了可能存在的问题。这是一本非常好的书，推荐大家阅读。纯 C 论坛上（http://purec.binghua.com）可以下载本书的电子版（PDF 格式），也可以在上面找到另外一些相关资源。

下面就是运行时的截图，现在它只能引导，什么也干不了，希望下次它能多干一点~~



参考资料

1. 《IA-32 Intel® Architecture Software Developer’s Manual Volume 3:System Programming Guide》 （Intel 2001）
2. 《Linux 内核 0.11 完全注释》（赵炯，2003）

《计算机组成原理》（唐朔飞，高等教育出版社）