**Author:Cuiyifeng**

什么是实模式？

简答：实模式是CPU启动的时候的模式，这时候就相当于一个速度超快的8086 ，不能使用多线程，不能实现权限分级，还不能访问20位以上地址线,也就是说只能访问1M内存(!!!)

简单原理：

实模式工作机理：

对于8086/8088来说计算实际地址是用绝对地址对1M求模。8086的地址线的物理结构：20根，也就是它可以物理寻址的内存范围为2^20个字节，即1 M空间，但由于8086/8088所使用的寄存器都是16位，能够表示的地址范围只有0-64K，这和1M地址空间来比较也太小了，所以为了在8086/8088下能够访问1M内存，Intel采取了分段寻址的模式：16位段基地址:16位偏移EA。其绝对地址计算方法为：16位基地址左移4位+16位偏移=20位地址。 比如：DS=1000H EA=FFFFH 那么绝对地址就为：10000H + 0FFFFH = 1FFFFH 地址单元。通过这种方法来实现使用16位寄存器访问1M的地址空间，这种技术是处理器内部实现的，通过上述分段技术模式，能够表示的最大内存为：FFFFh: FFFFh=FFFF0h+FFFFh=10FFEFh=1M+64K-16Bytes（1M多余出来的部分被称做高端内存区HMA）。但8086/8088只有20位地址线，只能够访问1M地址范围的数据，所以如果访问100000h~10FFEFh之间的内存（大于1M空间），则必须有第21根地址线来参与寻址（8086/8088没有）。因此，当程序员给出超过1M（100000H-10FFEFH）的地址时，因为逻辑上正常，系统并不认为其访问越界而产生异常，而是自动从0开始计算，也就是说系统计算实际地址的时候是按照对1M求模的方式进行的，这种技术被称为wrap-around。

**实模式（RM,16位微处理器）**

CPU[复位](http://baike.baidu.com/view/5247748.htm)（reset）或加电（power on）的时候以实模式启动，处理器以实模式工作。在实模式下，内存寻址方式和8086相同，由16位段寄存器的内容乘以16（10H）当做段基地址，加上16位[偏移地址](http://baike.baidu.com/view/883224.htm)形成20位的物理地址，最大寻址空间1MB，最大分段64KB。可以使用32位指令。32位的x86 CPU用做高速的8086。在实模式下，所有的段都是可以读、写和可执行的。

286架构导入[保护模式](http://baike.baidu.com/view/177586.htm)，允许硬件等级的存储器保护。然而要使用这些新的特色，需要额外先前不需要的软件指令。由于x86微处理机主要的设计规格，是能够完全地向前兼容于针对先前所有x86芯片所撰写的软件，因此286芯片的开机是处于'实模式'—也就是关闭新的存储器保护特性的模式，所以可以运行针对旧的微处理器所设计的软件。到现在为止，即使最新的x86 CPU一开始在电源打开处于实模式下，也能够运行针对先前任何芯片所撰写的软件.[1]

基本概念

**保护模式下的实模式**

[保护模式](http://baike.baidu.com/view/177586.htm)：寻址采用32位段和偏移量，最大[寻址空间](http://baike.baidu.com/view/2007755.htm)4GB，最大分段4GB (Pentium Pre及以后为64GB)。在保护模式下CPU可以进入虚拟8086方式，这是在保护模式下的实模式程序运行环境。

运行命令

程序运行的实质是什么?其实很简单,就是指令的执行,显然CPU是指令得以执行的硬件保障,那么CPU如何知道指令在什么地方呢?

80x86系列是使用CS[寄存器](http://baike.baidu.com/view/6159.htm)配合IP寄存器来通知CPU指令在内存中的位置.

程序指令在执行过程中一般还需要有各种数据,80x86系列有DS、ES、FS、GS、SS等用于指示不同用途的数据段在内存中的位置。

程序可能需要调用系统的服务子程序，80x86系列使用中断机制来实现[系统服务](http://baike.baidu.com/view/685551.htm)。

总的来说，这些就是实模式下一个程序运行所需的主要内容（其它如跳转、返回、端口操作等相对来说比较次要。）

保护模式

保护模式---从程序运行说起

无论实模式还是保护模式，根本的问题还是程序如何在其中运行。

因此我们在学习保护模式时应该时刻围绕这个问题来思考。和实模式下一样，保护模式下程序运行的实质仍是“CPU执行指令，操作相关数据”，因此实模式下的各种[代码段](http://baike.baidu.com/view/1315853.htm)、[数据段](http://baike.baidu.com/view/1005328.htm)、[堆栈段](http://baike.baidu.com/view/76043.htm)、[中断服务程序](http://baike.baidu.com/view/600257.htm)仍然存在，且功能、作用不变。

那么保护模式下最大的变化是什么呢？答案可能因人而异，我的答案是“地址转换方式”变化最大。

方式比较

先看一下实模式下的地址转换方式，假设我们在ES中存入0x1000，DI中存入0xFFFF,那么ES:DI=0x1000\*0x10+0xFFFF=0x1FFFF,这就是众所周知的“左移4位加偏移”。

那么如果在保护模式下呢？假设上面的数据不变ES=0x1000,DI=0xFFFF，现在ES:DI等于什么呢？

公式如下：（注：0x1000=1000000000000b= 10 0000 0000 0 00）ES:DI=全局描述符表中第0x200项描述符给出的段基址+0xFFFF

现在比较一下，好象是不一样。再仔细看看，又好象没什么区别！

为什么说没什么区别，因为我的想法是，既然ES中的内容都不是真正的段地址，凭什么实模式下称ES为“段寄存器”，而到了保护模式就说是“选择子”？你觉得呢?

其实它们都是一种映射，只是映射规则不同而已：在实模式下这个“地址转换方式”是“左移4位”；在保护模式下是“查全局/局部描述表”。前者是系统定义的映射方式，后者是用户自定义的转换方式。而它影响的都是“shadow register”从函数的观点来看，前者是表达式函数，后者是列举式函数：

实模式： F(es-->segment)={segment |segment=es\*0x10}

保护模式：F(es-->segment)={segment |(es,segment)∈GDT/LDT}

其中GDT、LDT分别表示全局描述符表和局部描述符表。

组成部分

保护模式最基本的组成部分是围绕着“地址转换方式”的变化增设了相应的机构。

1、数据段

前面说过，实模式下的各种代码段、数据段、堆栈段、中断服务程序仍然存在，我将它们统称为“数据段”，本文从此向下凡提到数据段都是使用这个定义。

2、描述符

保护模式下引入描述符来描述各种数据段，所有的描述符均为8个字节64位（0-7)，由第5个字节说明描述符的类型，类型不同，描述符的结构也有所不同。

若干个描述符集中在一起组成描述符表，而描述符表本身也是一种数据段，也使用描述符进行描述。从现在起，“地址转换”由描述符表来完成，从这个意义上说，描述符表是一张地址转换函数表。

3、选择子(描述符的一部分)

选择子是一个2字节的数，共16位，最低2位表示RPL，第3位表示查表是利用GDT（全局描述符表）还是LDT（局部描述符表）进行，最高13位给出了所需的描述符在描述符表中的地址。（注：13位正好足够寻址8K项）有了以上三个概念之后可以进一步工作了，现在程序的运行与实模式下完全一样！！！各段寄存器仍然给出一个“段值”，只是这个“假段值”到真正的段地址的转换不再是“左移4位”，而是利用描述符表来完成。但现在出现一个新的问题是：

系统如何知道GDT/LDT在内存中的位置呢？

为了解决这个问题，显然需要引入新的寄存器用于指示GDT/LDT在内存中的位置。在80x86系列中引入了两个新寄存器GDTR和LDTR，其中GDTR用于表示GDT在内存中的段地址和段限（就是表的大小），因此GDTR是一个48位的寄存器，其中32位表示段地址，16位表示段限（最大64K，每个描述符8字节，故最多有64K/8=8K个描述符）。LDTR用于表示LDT在内存中的位置，但是因为LDT本身也是一种数据段，它必须有一个描述符，且该描述符必须放在GDT中，因此LDTR使用了与DS、ES、CS等相同的机制，其中只存放一个“选择子”，通过查GDT表获得LDT的真正[内存地址](http://baike.baidu.com/view/404417.htm)。对了，还有中断要考虑，在80x86系列中为中断服务提供中断/陷阱描述符，这些描述符构成中断描述符表（IDT），并引入一个48位的全地址寄存器存放IDT的内存地址。理论上IDT表同样可以有8K项，可是因为80x86只支持256个中断，因此IDT实际上最大只能有256项（2K大小）。

问题总结

基本问题

前面介绍了保护模式的基本问题，也是核心问题，解决了上面的问题，程序就可以在保护模式下运行了。

但众所周知80286以后在保护模式下实现了对多任务的硬件支持。我的第一反应是：为什么不在实模式下支持多任务，是不能还是不愿？

思考之后，我的答案是：实模式下能实现多任务。

因为多任务的关键是有了描述符，可以给出关于数据段的额外描述，如权限等，进而在这些附加信息的基础上进行相应的控制，而实模式下缺乏描述符，但假设我们规定各段的前2个字节或若干字节用于描述段的附加属性，我觉得和使用描述符这样的机制没有本质区别，如果再附加其他机制...基于上述考虑，我更倾向于认为任务是独立于保护模式之外的功能。

下面我们来分析一下任务。任务的实质是什么呢？很简单，就是程序嘛！！

所谓任务的切换其实就是程序的切换！！

现在问题明朗了。实模式下程序一个接一个运行，因此程序运行的“环境”不必保存；保护模式下可能一个程序在运行过程中被暂停，转而执行下一个程序，我们要做什么？很容易想到保存程序运行的环境就行了（想想游戏程序的保存进度功能），比如各寄存器的值等。

显然这些“环境”数据构成了一类新的数据段（即TSS）。延用前面的思路，

给这类数据段设置描述符（TSS描述符），将该类描述符放在GDT中（不能放在LDT中，因为80x86不允许：）），最后再加一个TR寄存器用于查表。TR是一个起“选择子”作用的寄存器，16位。

好了，任务切换的基本工作就是将原任务的“环境”存入TSS数据段，更新TR寄存器，系统将自动查GDT表获得并装载新任务的“环境”，然后转到新任务执行。

附加要求

为什么叫附加要求，因为现在任务还不能很好地工作。前面说过，任务实质上是程序，不同的程序是由不同的用户写的，所有这些程序完全可能使用相同的地址空间，而任务的切换过程一般不会包括内存数据的刷新，不是不可能，而是如果那样做太浪费了。因此必须引入分页机制才可能有效地完成对多任务的支持。

分页的实质就是实现程序内地址到物理地址的映射，这也是一个“地址转换”机制，同样可以使用前面的方案（即类似GDT的做法）：首先建立页表这样一种数据段，在80x86中使用二级页表方案，增设一个CR3寄存器用于存放一级页表（又称为页目录）在内存中的地址，CR3共32位，其低12位总是为零，高20位指示页目录的内存地址，因此页目录总是按页对齐的。CR3作为任务“环境”的一部分在任务切换时被存入TSS数据段中。

当然还得有相应的缺页中断机制及其相关寄存器CR2（页故障线性地址寄存器）。

保护模式下增加了什么？

1、寄存器 GDR LDR IDR TR CR3

2、数据段 描述符表(GDT LDT) 任务数据段(TSS) 页表(页目录 二级页表)

3、机制 权限检测(利用选择子/描述符/页表项的属性位)

线性地址到物理地址的映射

常用名词

前面内容中出现过的不再解释。

1、RPL 选择子当中的权限位确定的权限

2、CPL 特指CS中的选择子当中的权限位确定的权限

3、EPL EPL=Max(RPL,CPL),即RPL和CPL中数值较大的，或说权限等级较小的

4、DPL 描述符中的权限位确定的权限

5、PL 泛指以上4种特权级

6、任务特权 =CPL

7、I/O特权 由EFLAGS寄存器的位13、14确定的权限

8、一致代码段 一种特殊的代码段，它在CPL>=DPL时允许访问

正常的代码段在CPL=DPL RPL<=DPL时才允许访问

**补充：CPU是如何处理物理地址的？**

在实模式下用段寄存器左移4位与偏移量相加。在保护模式下用段描述符中的基址加偏移量。这两者其实是一样的。

当Intel把80286推出时，其地址空间变成了24位，则从8086的20位到24位，十分自然地要加大段寄存器才行。实际上，段寄存器和指针都被加大了，只是由于保护的原因，加大的部分没有被程序看见，到了80386之后，地址又从24位加大到32位。

在8086中，CPU只有“看得见部分”，但在80286之后，在“看不见部分”中已经包含了地址值。“看得见部分”就退化为只是一个标号，再也不用参与地址形成运算了。地址的形成总是从“不可看见部分”取出基址值与偏移相加形成地址。

也就是说，在实模式下，当一个段寄存器被装入一个值时，“看不见部分”的段限界被设成FFFFH，基址部分将装入值左移4位，段属性部分设成16位0特权级。这个过程与保护模式时装入一个段寄存器是同理的，只是保护模式的“不可见部分”是从描述表中取值，而实模式是一套固定的过程。

对于CPU在翻译地址时，是没有实模式与保护模式之分的，它只管用基址（“不可见部分”）去加上偏移量。

实模式下的段寄存器装入有固定的形成办法，从而也就不需要保护模式的“描述符”了，因此，保持了与8086/8088的兼容性。而“描述符”也只是为了装入段寄存器的“不可见部分”而设的。（保护模式。还为了存储段属性等实现保护机制的数据结构）

CPU的地址形成与“看得见部分”的当前值毫无关系。这也就解释了为什么刚进入保护模式时，后面的代码依然被正确地运行，而这时代码段寄存器CS的值还是进入保护模式前的实模式值，或者从保护模式回到实模式时，代码段CS被改变之前程序是正常地工作，而不会“突变”到CS左移4位的地址上去。

比如：在保护模式时，CS是08H的选择子，到了实模式时，CS还是08H，但地址不会突然变成80H加上偏移量。因为地址的形成不理会段寄存器“看得见部分”的当前值，这一值只是在被装入时对CPU有用。（实模式与保护模式只是在向段寄存器加载值时处理不同，之后的处理是相同的）

地址的形成与CPU的工作模式无关，也就是说，实模式与0特权级保护模式不分页时是一模一样的。明白了这一机理后，在实模式下一样可以处理通常被认为只有在保护模式才能做的事，比如访问整个机器的内存。不必理会保护模式下的众多术语或许会更易于理解，如选择子就是“看得见部分”，描述符是为了装入“不可见部分”而设的。

**保护模式(PM, 32位微处理器)**

**什么是保护模式？简答**

保护模式，是一种80286系列和之后的x86兼容[CPU](http://baike.baidu.com/view/2089.htm)操作模式。保护模式有一些新的特色，设计用来增强[多工](http://baike.baidu.com/view/560598.htm)和系统稳定度，像是内存保护，[分页](http://baike.baidu.com/view/159980.htm) 系统，以及硬件支援的虚拟内存。大部分的现今x86 [操作系统](http://baike.baidu.com/view/880.htm) 都在保护模式下运行，包含Linux、FreeBSD、以及 [微软](http://baike.baidu.com/view/2353.htm) Windows 2.0和之后版本。保护模式与[实模式](http://baike.baidu.com/view/404433.htm)相对应。在80286以前，CPU只有实时模式，地址总线有20位，而内存地址是16位，也就是最多能够访问2^20=1M的内存空间。在80286及以后，内存地址改为16位或32位，至少可以访问到2^32=4G的内存空间。但为了保证后续的CPU能够运行旧的CPU，只能保持向下兼容。因此，80286及以后的CPU首先进入实模式，然后通过切换机制再进入到保护模式。

**注意，在32位模式下，段寄存器仍然是16位**

80386寄存器的宽度大多是32位，可分为如下几组：

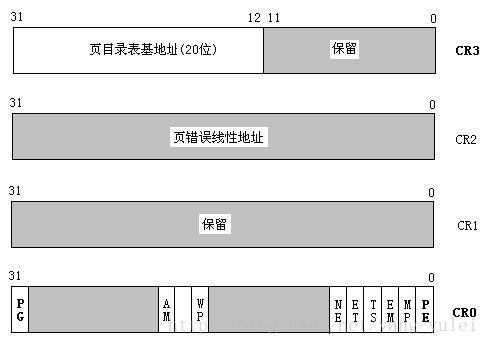
通用寄存器、段寄存器、指令指针

及标志寄存器、系统地址寄存器、调试寄存器、控制寄存器和测试寄存器。

应用程序主要使用前面三组寄存器，只有系统才会使用其他寄存器。

**控制寄存器：**

控制寄存器（CR0~CR3）用于控制和确定处理器的操作模式以及当前执行任务的特性。4个控制寄存器都是32位的。



**CR0**：含有控制CPU操作模式和状态的标识

CR1：保留不用

CR2：存储导致页错误的线性地址

**CR3**：含有页目录表的物理内存基址

CR0中的保护控制位：

**PE**：CR0的位0是启用保护（Protection Enable）标志。（CR0的最低位）

当设置该位时即**开启了保护模式**，当复位时即进入实地址模式。这个标志仅开启**段级保护**，而没有启用分页机制。若要启用分页机制，那么PE和PG都要置位。

**PG**：CR0的位31是分页（Paging）标志。（CR0的最高位）

当设置该位时即**开启了分页机制**，当复位时则禁止分页机制，此时所有线性地址等同于物理地址。

注意，在开启这个标志之前必须已经开启PE标志，否则CPU会产生一个一般保护性异常。

改变PG位的代码必须在线性地址空间和物理地址空间中具有相同地址，这部分具有相同地址的代码在分页和未分页世界之间起着桥梁的作用。

如果PE=0、PG=0，处理器工作在实地址模式下。（兼容早期的实模式操作系统）

如果PE=1、PG=0，处理器工作在无分页机制的段保护模式下（兼容段式管理的操作系统）

如果PE=1、PG=1，处理器工作在段页式保护模式下

在系统刚上电时，处理器被复位成PE=0和PG=0（即实模式状态），以允许引导代码在启用分段和分页机制之前能够初始化这些寄存器和[**数据结构**](http://lib.csdn.net/base/31)。

什么是保护模式？

286处理器体系结构引入了地址保护模式的概念，处理器能够对内存及一些其他外围设备做硬件级的保护设置（保护设置实质上就是屏蔽一些地址的访问）。

对于80286或以上的CPU通过A20 GATE来控制A20地址线。 技术发展到了80286，虽然系统的地址总线由原来的20根发展为24根，这样能够访问的内存可以达到2^24=16M,但是Intel在设计80286时提出的目标是向下兼容,所以在实模式下，系统所表现的行为应该和8086/8088所表现的完全一样，也就是说，在实模式下，80386以及后续系列应该和8086/8088完全兼容仍然使用A20地址线。所以说80286芯片存在一个BUG：它开设A20地址线。如果程序员访问100000H-10FFEFH之间的内存，系统将实际访问这块内存（没有wrap-around技术），而不是象8086/8088一样从0开始。

 为了解决上述兼容性问题，IBM使用键盘控制器上剩余的一些输出线来管理第21根地址线（从0开始数是第20根） 的有效性，被称为A20 Gate：

      1> 如果A20 Gate被打开，则当程序员给出100000H-10FFEFH之间的地址的时候，系统将真正访问这块内存区域；

     2 如果A20 Gate被禁止，则当程序员给出100000H-10FFEFH之间的地址的时候，系统仍然使用8086/8088的方式即取模方式（8086仿真）。绝大多数IBM PC兼容机默认的A20 Gate是被禁止的。现在许多新型PC上存在直接通过BIOS功能调用来控制A20 Gate的功能。

 上面所述的内存访问模式都是实模式，在80286以及更高系列的PC中，即使A20 Gate被打开，在实模式下所能够访问的内存最大也只能为10FFEFH，尽管它们的地址总线所能够访问的能力都大大超过这个限制。为了能够访问10FFEFH以上的内存，则必须进入保护模式。

**保护模式：**经常缩写为p-mode,在Intel iAPX 286程序员参考手册中（iAPX 286是Intel 80286的另一种叫法）它又被称作为虚拟地址保护模式。经管在Intel 80286手册中已经提出了虚地址保护模式，但实际上它只是一个指引，真正的32位地址出现在Intel 80386上。保护模式本身是80286及以后兼容处理器序列之后产成的一种操作模式，它具有许多特性设计为提高系统的多道任务和系统的稳定性。例如内存的保护，分页机制和硬件虚拟存储的支持。现代多数的x86处理器操作系统都运行在保护模式下，包括Linux, Free BSD, 和Windows 3.0（它也运行在实模式下，为了和Windows 2.x应用程序兼容）及以后的版本。

80286及以后的处理器另一种工作模式是实模式（仅当系统启动的一瞬间），本着向下兼容的原则屏蔽保护模式特性，从而容许老的软件能够运行在新的芯片上。作为一个设计规范，所有的x86系列处理器，除嵌入式Intel80387之外，都是系统启动工作在实模式下，确保遗留下的操作系统向下兼容。它们都必须被启动程序（操作系统程序最初运行代码）重新设置而相应进入保护模式的，在这之前任何的保护模式特性都是无效的。在现代计算机中，这种匹配进入保护模式是操作系统启动时最前沿的动作之一。

在被调停的多道任务程序中，它可以从新工作在实模式下是相当可能的。保护模式的特性是阻止被其他任务或系统内核破坏已经不健全的程序的运行，保护模式也有对硬件的支持，例如中断运行程序，移动运行进程文档到另一个进程和置空多任务的保护功能。 386及以后系列处理器不仅具有保护模式又具有32位寄存器，结果导致了处理功能的混乱，因为80286虽然支持保护模式，但是它的寄存器都是16位的，它是通过自身程序设定而模拟出的32位，并非32位寄存器处理。归咎于这种混乱现象，它促使Windows/386及以后的版本彻底抛弃80286的虚拟保护模式，以后保护模式的操作系统都是运行在80386以上，不再运行在80286（尽管80286模式支持保护模式），所以说80286是一个过渡芯片，它是一个过渡产品。

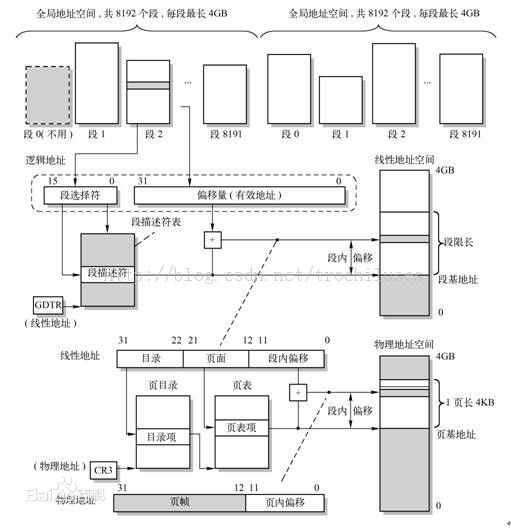
尽管286和386处理器能够实现保护模式和兼容以前的版本，但是内存的1M以上空间还是不易存取，由于内存地址的回绕，IBM PC XT （现以废弃）设计一种模拟系统，它能过欺骗手段访问到1M以上的地址空间，就是开通了A20地址线。在保护模式里，前32个中断为处理器异常预留，例如，中断0D（十进制13）常规保护故障和中断00是除数为零异常。

如果要访问更多的内存，则必须进入保护模式，那么，在保护模式下，A20 Gate对于内存访问有什么影响呢？

为了搞清楚这一点，我们先来看一看A20的工作原理。A20，从它的名字就可以看出来，其实它就是对于A20（从0开始数）的特殊处理(也就是对第21根地址线的处理)。如果A20 Gate被禁止，对于80286来说，其地址为24根地址线，其地址表示为EFFFFF；对于80386极其随后的32根地址线芯片来说，其地址表示为FFEFFFFF。这种表示的意思是：

**1>** 如果A20 Gate被禁止。则其第A20在CPU做地址访问的时候是无效的，永远只能被作为0。所以，在保护模式下，如果A20 Gate被禁止，则可以访问的内存只能是奇数1M段，即1M,3M,5M…，也就是00000-FFFFF, 200000-2FFFFF,300000-3FFFFF…

      2如果A20 Gate被打开。则其第20-bit是有效的，其值既可以是0，又可以是1。那么就可以使A20线传递实际的地址信号。如果A20 Gate被打开，则可以访问的内存则是连续的。



段页式寻址方式

**RM到PM的切换过程及原理**

**GDT**

全局描述表（GDT Global Descriptor Table）：在保护模式下，一个重要的必不可少的[数据结构](http://baike.baidu.com/view/9900.htm)就是它。

为什么要有GDT？我们首先考虑一下在实时模式下的编程模型：

在实时模式下，我们对一个[内存地址](http://baike.baidu.com/view/404417.htm)的访问是通过Segment:Offset的方式来进行的，其中Segment是一个段的Base Address，一个Segment的最大长度是64 KB，这是16-bit系统所能表示的最大长度。而Offset则是相对于此Segment Base Address的[偏移量](http://baike.baidu.com/view/1254177.htm)。Base Address+Offset就是一个内存绝对地址。由此，我们可以看出，一个段具备两个因素：Base Address和Limit（段的最大长度），而对一个内存地址的访问，则是需要指出：使用哪个段？以及相对于这个段Base Address的Offset，这个Offset应该小于此段的Limit。当然对于16-bit系统，Limit不要指定，默认为最大长度64KB，而 16-bit的Offset也永远不可能大于此Limit。我们在实际编程的时候，使用16-bit段寄存器CS（Code Segment），DS（Data Segment），SS（Stack Segment）来指定Segment，CPU将段寄存器中的数值向左偏移4-bit，放到20-bit的地址线上就成为20-bit的Base Address。

到了保护模式，内存的管理模式分为两种，段模式和页模式，其中页模式也是基于段模式的。也就是说，保护模式的内存管理模式事实上是：纯段模式和段页式。进一步说，段模式是必不可少的，而页模式则是可选的——如果使用页模式，则是段页式；否则这是纯段模式。

既然是这样，我们就先不去考虑页模式。对于段模式来讲，访问一个内存地址仍然使用Segment:Offset的方式，这是很自然的。由于保护模式运行在32位系统上，那么Segment的两个因素：Base Address和Limit也都是32位的。IA-32允许将一个段的Base Address设为32-bit所能表示的任何值（Limit则可以被设为32-bit所能表示的，以2^12为倍数的任何值），而不象实时模式下，一个段的Base Address只能是16的倍数（因为其低4-bit是通过左移运算得来的，只能为0，从而达到使用16-bit段寄存器表示20-bit Base Address的目的），而一个段的Limit只能为固定值64 KB。另外，保护模式，顾名思义，又为段模式提供了保护机制，也就说一个段的描述符需要规定对自身的访问权限（Access）。所以，在保护模式下，对一个段的描述则包括3方面因素：[Base Address, Limit, Access]，它们加在一起被放在一个64-bit长的数据结构中，被称为段描述符。这种情况下，如果我们直接通过一个64-bit段描述符来引用一个段的时候，就必须使用一个64-bit长的段寄存器装入这个段描述符。但Intel为了保持向后兼容，将段寄存器仍然规定为16-bit（尽管每个段寄存器事实上有一个64-bit长的不可见部分，但对于程序员来说，段寄存器就是16-bit的），那么很明显，我们无法通过16-bit长度的段寄存器来直接引用64-bit的段描述符。怎么办？

解决的方法就是把这些长度为64-bit的段描述符放入一个[数组](http://baike.baidu.com/view/209670.htm)中，而将段寄存器中的值作为下标索引来间接引用（事实上，是将段寄存器中的高13-bit的内容作为索引）。这个全局的数组就是GDT。事实上，在GDT中存放的不仅仅是段描述符，还有其它描述符，它们都是64-bit长，我们随后再讨论。

GDT可以被放在内存的任何位置，那么当程序员通过段寄存器来引用一个段描述符时，CPU必须知道GDT的入口，也就是基地址放在哪里，所以Intel的设计者门提供了一个寄存器GDTR用来存放GDT的入口地址，程序员将GDT设定在内存中某个位置之后，可以通过LGDT指令将GDT的入口地址装入此寄存器，从此以后，CPU就根据此寄存器中的内容作为GDT的入口来访问GDT了。

GDT是保护模式所必须的数据结构，也是唯一的——不应该，也不可能有多个。另外，正象它的名字（Global Descriptor Table）所揭示的，它是全局可见的，对任何一个任务而言都是这样。

除了GDT之外，IA-32还允许程序员构建与GDT类似的数据结构，它们被称作LDT（Local Descriptor Table），但与GDT不同的是，LDT在系统中可以存在多个，并且从LDT的名字可以得知，LDT不是全局可见的，它们只对引用它们的任务可见，每个任务最多可以拥有一个LDT。另外，每一个LDT自身作为一个段存在，它们的段描述符被放在GDT中。

IA-32为LDT的入口地址也提供了一个寄存器LDTR，因为在任何时刻只能有一个任务在运行，所以LDT寄存器全局也只需要有一个。如果一个任务拥有自身的LDT，那么当它需要引用自身的LDT时，它需要通过LLDT将其LDT的段描述符装入此寄存器。LLDT指令与LGDT指令不同的是，LGDT指令的操作数是一个32-bit的内存地址，这个内存地址处存放的是一个32-bit GDT的入口地址，以及16-bit的GDT Limit。而LLDT指令的操作数是一个16-bit的选择子，这个选择子主要内容是：被装入的LDT的段描述符在GDT中的索引值——这一点和刚才所讨论的通过段寄存器引用段的模式是一样的。

**LDT**

LDT只是一个可选的数据结构，你完全可以不用它。使用它或许可以带来一些方便性，但同时也带来复杂性，如果你想让你的OS内核保持简洁性，以及可移植性，则最好不要使用它。

引用GDT和LDT中的段描述符所描述的段，是通过一个16-bit的数据结构来实现的，这个数据结构叫做Segment Selector——段选择子。它的高13位作为被引用的段描述符在GDT/LDT中的下标索引，bit 2用来指定被引用段描述符被放在GDT中还是到LDT中，bit 0和bit 1是RPL——请求特权等级，被用来做保护目的，我们这里不详细讨论它。

前面所讨论的装入[段寄存器](http://baike.baidu.com/view/364403.htm)中作为GDT/LDT索引的就是Segment Selector，当需要引用一个内存地址时，使用的仍然是Segment:Offset模式，具体操作是：在相应的段寄存器装入Segment Selector，按照这个Segment Selector可以到GDT或LDT中找到相应的Segment Descriptor，这个Segment Descriptor中记录了此段的Base Address，然后加上Offset，就得到了最后的内存地址。

**进入过程：**

**1.设置GDT**

由上一节的讨论得知，GDT是保护模式所必须的[数据结构](http://baike.baidu.com/subview/9900/5066576.htm)，那么我们在进入保护模式之前，必须设定好GDT，并通过LGDT将其装入相应的寄存器。

尽管GDT允许被放在内存的任何位置，但由于GDT中的元素——描述符——都是64-bit长，也就是说都是8个字节，所以为了让CPU对GDT的访问速度达到最快，我们应该将GDT的入口地址放在以8个字节对齐，也就是说是8的倍数的地址位置。

GDT中第一个描述符必须是一个空描述符，也就是它的内容应该全部为0。如果引用这个描述符进行内存访问，则是产生General Protection异常。

如果一个OS不使用虚拟内存，段模式会是一个不错的选择。但现代OS没有不使用虚拟内存的，而实现虚拟内存的比较方便和有效的[内存管理](http://baike.baidu.com/view/4541016.htm)方式是页式管理。但是在IA-32上如果我们想使用页式管理，我们只能使用段页式——没有方法可以完全禁止段模式。但我们可以尽力让段的效果降低的最小。

IA-32提供了一种被称作“Basic Flat Model”的分段模式可以达到这种效果。这种模式要求在GDT中至少要定义两个段描述符，一个用来引用Data Segment，另一个用来引用Code Segment。这2个Segment都包含整个线性空间，即Segment Limit = 4 GB，即使实际的物理内存远没有那么多，但这个空间定义是为了将来由页式管理来实现虚拟内存。

在这里，我们只是处于启动阶段，所以我们只需要初步设置一下GDT，等真正进入保护模式，启动了OS Kernel之后，具体OS打算如何设置GDT，使用何种内存管理模式，由Kernel自身来设置，启动只需要给Kernel的数据段和代码段设置全部线性空间就可以了。

段描述符的格式如下图所示：

具体到[代码段](http://baike.baidu.com/view/1315853.htm)和[数据段](http://baike.baidu.com/view/1005328.htm)，它们的格式如下图所示：

下面就是在启动阶段为进入保护模式而设置的临时的gdt。这里定义了3个段描述符：第一个是系统规定的空描述符，第2个是引用4 GB线性空间的代码段，第3个是引用4 GB线性空间的数据段。这是"Basic Flat Model"所要求的最下GDT设置，但就启动阶段，只是为了进入保护模式，并为内核提供一个连续的，最大的线性空间这个目的而言，已经足够了。

# Descriptor tables

gdt:

.word 0, 0, 0, 0 # dummy

.word 0xFFFF # 4Gb - (0x100000\*0x1000 = 4Gb)

.word 0 # base address = 0

.word 0x9A00 # code read/exec

.word 0x00CF # granularity = 4096, 386

# (+5th nibble of limit)

.word 0xFFFF # 4Gb - (0x100000\*0x1000 = 4Gb)

.word 0 # base address = 0

.word 0x9200 # data read/write

.word 0x00CF # granularity = 4096, 386

# (+5th nibble of limit)

**2.将GDT表入口和大小装入GDTR寄存器**

设置好GDT之后，我们需要通过LGDT指令将设定的gdt的入口地址和gdt表的大小装入GDTR寄存器。

GDTR寄存器包括两部分：32-bit的线性基地址，以及16-bit的GDT大小（以字节为单位）。需要注意的是，对于32-bit线性基地址，必须是32-bit绝对物理地址，而不是相对于某个段的偏移量。而我们在启动阶段，在进入保护模式之前，我们CS和DS设置很可能不是0，所以我们必须计算出gdt的绝对物理地址。

为了执行LGDT指令，你需要把这两部分内容放在内存的某个位置，然后将这个位置的内存地址作为操作数传递给LGDT指令。然后LGDT指令会自动将保存在这个位置的这两部分值装入GDTR寄存器。

# 这是存放GDTR所需的两部分内容的位置

gdt\_48:

.word 0x8000 # gdt limit=2048,

# 256 GDT entries

.word 0, 0 # gdt base (filled in later)

# 下面这段代码用来计算GDT的32-bit线性地址，并将其装入GDTR寄存器。

xorl %eax, %eax # Compute gdt\_base

movw %ds, %ax # (Convert %ds:gdt to a linear ptr)

shll 4, %eax

addl $gdt, %eax

movl %eax, (gdt\_48+2)

lgdt gdt\_48 # load gdt with whatever is appropriate

**3.其他必要操作**

在进入保护模式之前，除了需要设置和装入GDT之外，还需要做如下一些事情：

屏蔽所有可屏蔽中断；

装入IDTR；中文全称：中断描述表寄存器

所有协处理器被正确的复位。

由于在实时模式和保护模式下的中断处理机制有一些不同，所以在进入保护模式之前，务必禁止所有可屏蔽中断，这可以通过下面两种方法之一：

使用CLI指令；

对8259A[可编程中断控制器](http://baike.baidu.com/view/2348508.htm)编程以屏蔽所有中断。

即使当我们进入保护模式之后，也不能马上将中断打开，这时因为我们必须在OS Kernel中对相关的保护模式中断处理所需的数据结构正确的初始化之后，才能打开中断，否则会产生处理器异常。

在实时模式下，中断处理使用IVT(Interrupt Vector Table)，在保护模式下，中断处理使用IDT（Interrupt Descriptor Table），所以，我们必须在进保护模式之前设置IDTR。

IDTR的格式和GDTR相同，IDTR的装入方式和GDTR也相同。由于IDT中相关的中断处理程序需要让OS Kernel来设定，所以在启动阶段，我们只需要将IDTR中IDT的基地址和Size都设为0就可以了，随后，等进入保护模式之后，由OS Kernel来真正设置它。

关于中断机制和中断处理，请参考 Interrupt & Exception ，这里不再赘述。

#

# 这是存放IDTR所需的两部分内容的位置

#

idt\_48:

.word 0 # idt limit = 0

.word 0, 0 # idt base = 0L

# 对于IDTR的处理，只需要这一条指令即可

lidt idt\_48 # load idt with 0,0

#

# 通过设置8259A PIC，屏蔽所有可屏蔽中断

#

movb xFF, %al # mask all interrupts for now

outb %al, xA1

call delay

movb xFB, %al # mask all irq's but irq2 which

outb %al, x21 # is cascaded

# 保证所有的协处理都被正确的Reset

xorw %ax, %ax

outb %al, xf0

call delay

outb %al, xf1

call delay

# Delay is needed after doing I/O

delay:

outb %al,x80

ret

好，一切准备就绪

进入保护模式，还是进入实时模式，完全靠CR0寄存器的PE标志位来控制：如果PE=1，则CPU切换到PM，否则，则进入RM。

设置CR0-PE位的方法有两种：

第一种

第一种是80286所使用的LMSW指令，后来的80386及更高型号的CPU为了保持向后兼容，都保留了这个指令。这个指令只能影响最低的4 bit，即PE，MP，EM和TS，对其它的没有影响。

#

#通过LMSW指令进入保护模式

#

movw $0x0001, %ax # protected mode (PE) bit

lmsw %ax # This is it!

第二种

第二种是Intel所建议的在80386以后的CPU上使用的进入PM的方式，即通过移动MOV指令。MOV指令可以设置CR0寄存器的所有域的值。

#

#通过MOV指令进入保护模式

#

movl %cr0, %eax

xorb $0x01, %al # set PE = 1

movl %eax, %cr0 # go!!

现在已经进入保护模式了。

**4.启动内核**

我们已经从实时模式进入保护模式，现在我们马上就要启动OS Kernel了。

OS Kernel运行在32-bit段模式，而当前我们却仍然处于16-bit段模式。这是怎么回事？为了了解这个问题，我们需要仔细探讨一下IA-32的段模式的实现方法。

IA-32共提供了6个16-bit段寄存器：CS，DS，SS，ES，FS，GS。但事实上，这16-bit只是对程序员可见的部分，但每个寄存器仍然包括64-bit的不可见部分。

可见部分是为了供程序员装载段寄存器，但一旦装载完成，CPU真正使用的就只是不可见部分，可见部分就完全没有用了。

不可见部分存放的内容是什么？具体格式我没有看到相关资料，但可以确定的是隐藏部分的内容和段描述符的内容是一致的（请参考段描述的格式），只不过格式可能不完全相同。但格式对我们理解这一点并不重要，因为程序员不可能能够直接操作它。

我们以CS寄存器为例，对于其它寄存器也是一样的：

在实时模式下，当我们执行一个装载CS寄存器的指令的时候（jmp，call，ret等），相关的值会被装入CS寄存器的可见部分，但同时CPU也会根据可见部分的内容来设置不可见部分。比如我们执行"ljmp x1234, $go "之后，CS寄存器的可见部分的内容就是1234h，同时，不可见部分的32-bit Base Address域被设置为00001234h，20-bit的Limit域被设置为固定值10000h，也就是64 KB，Access Information部分的其它值我们不去考虑，只考虑其D/B位，由于执行此指令时处于Real Mode模式，所以D/B被设置为0，表示此段是一个16-bit段。当对CS寄存器的可见部分和不可见部分的内容都被设置之后，CS寄存器的装载工作完成。随后当CPU需要通过CS的内容进行地址运算的时候，则仅仅引用不可见部分。

在保护模式下，当我们执行一个装载CS寄存器的指令的时候，段选择子（Segment Selector）被装入CS寄存器的可见部分，同时CPU根据此选择子到相应的描述符表中（GDT或LDT）找到相应的段描述符并将其内容装载入CS寄存器的不可见部分。随后CPU当需要通过CS的内容进行地址运算的时候，也仅仅引用不可见部分。

从上面的描述可以看出，事实上CPU在引用段寄存器的内容进行地址运算时，实时模式和保护模式是一致的。另外，也明白了为什么我们在实时模式下设置的段寄存器的内容到了保护模式下仍然引用的是16-bit段。

那么我们如何将CS设置为引用32-bit段？方法就像我们前面所讨论的，使用jmp或[call指令](http://baike.baidu.com/view/3127923.htm)，引用一个段选择子，到GDT中装载一个引用32-bit段的段描述符。

需要注意的是，如果CS寄存器的内容指出当前是一个16-bit段，那么当前的地址模式也就是16-bit地址模式，这与你当前是出于实时模式还是保护模式无关。而我们装载32-bit段的jmp指令或call指令必须使用的是32-bit地址模式。而我们当前的boot部分代码是16-bit代码，所以我们必须在此jmp/call指令前加上地址转换前缀代码66h。

下面的例子就是使用jmp指令装入32-bit段。Jmpi指令的含义是段间跳转，其Opcode为Eah，其格式为：jmpi Offset, Segment Selector。

# 由于当前的代码是16-bit代码，而我们要执行32-bit地址模式的指令，指令前

# 需要有地址模式切换前缀66h，如果我们直接写jmp指令，由[编译器](http://baike.baidu.com/view/487018.htm)来生成代码

# 的话，是无法作到这一点的，所以我们直接写相关数据。

.byte 0x66, 0xea # prefix + jmpi-opcode

.long 0x1000 # Offset

.word \_\_KERNEL\_CS　# CS segment selector

上面的代码相当于32-bit指令：

jmpi 0x1000,\_\_KERNEL\_CS

如果\_\_KERNEL\_CS段选择子所引用的段描述符设置的段空间为线形地址[0，4 GB]，而我们将OS Kernel放在物理地址1000h，那么此jmpi指令就跳转到OS Kernel的入口处，并开始执行它。

此时，启动阶段结束，OS正式开始运行！

**补充A20Gate知识：**

**A20地址线并不是打开保护模式的关键，只是在保护模式下，不打开A20地址线，你将无法访问到所有的内存（具体参考下面的第5点）**

1. 用于80286与8086兼容
2. 用于80286处于实模式下时，防止用户程序访问到100000h~10FFEFh之间的内存（高端内存）
3. 8086模式，A20关闭的情况下，访问超过1MB内存时，会自动回卷
4. 8086模式下，A20打开的情况下，访问超过1MB内存，就真实的访问
5. 保护模式下，A20关闭（始终为0），则用户的地址只能是：0 - (1MB-1), 2 - (3MB-1), 4 - (5MB-1)，我们可以这样设想，A20为个位数（以1MB为单位），如果它始终为0，你永远不可能让这个数变成奇数。
6. 保护模式下，A20开启，则可以访问全地址，没有奇偶MB的问题。

**调用BIOS中断就可以实现A20 Gate的控制功能。**

这个BIOS中断为 INT 15h,AX=2401h。被称为Fast A20。

**1. A20 Gate in Protected Mode**

从80286开始，系统出现了一种新的机制，被称为保护模式。到了80386，保护模式得到了进一步的完善和发展，并且对于80386以后的芯片，保护模式的变化就非常小了。

我们在上一节已经谈到，如果要访问更多的内存，则必须进入保护模式，那么，在保护模式下，A20Gate对于内存访问有什么影响呢？

为了搞清楚这一点，我们先来看一看A20的工作原理。A20，从它的名字就可以看出来，其实它就是对于20-bit（从0开始数）的特殊处理(也就是对第21根地址线的处理)。如果A20Gate被禁止，对于80286来说，其地址为24bit，其地址表示为EFFFFF；对于80386极其随后的32-bit芯片来说，其地址表示为FFEFFFFF。这种表示的意思是如果A20Gate被禁止，则其第20-bit在CPU做地址访问的时候是无效的，永远只能被作为0；如果A20 Gate被打开，则其第20-bit是有效的，其值既可以是0，又可以是1。

所以，在保护模式下，如果A20Gate被禁止，则可以访问的内存只能是奇数1M段，即1M,3M,5M…，也就是00000-FFFFF,200000-2FFFFF,300000-3FFFFF…。如果A20 Gate被打开，则可以访问的内存则是连续的。

**2. How to Enable A20Gate**

多数PC都使用键盘控制器（8042芯片）来处理A20Gate。

从理论上讲，打开A20Gate的方法是通过设置8042芯片输出端口（64h）的2nd-bit，但事实上，当你向8042芯片输出端口进行写操作的时候，在键盘缓冲区中，或许还有别的数据尚未处理，因此你必须首先处理这些数据。

流程如下：

　1. 禁止中断；

　2. 等待，直到8042 Inputbuffer为空为止；

　3. 发送禁止键盘操作命令到8042Input buffer；

　4. 等待，直到8042 Inputbuffer为空为止；

　5. 发送读取8042 OutputPort命令；

　6. 等待，直到8042 Outputbuffer有数据为止；

　7. 读取8042 Outputbuffer，并保存得到的字节；

　8. 等待，直到8042 Inputbuffer为空为止；

　9. 发送Write 8042Output Port命令到8042 Input buffer；

　10. 等待，直到8042 Inputbuffer为空为止；

　11. 将从8042 OutputPort得到的字节的第2位置1（OR 2），然后写入8042 Input buffer；

　12. 等待，直到8042 Inputbuffer为空为止；

　13. 发送允许键盘操作命令到8042Input buffer；

　14. 打开中断。

下面代码是一个相关实现：

enable\_a20:

sti

call wait\_input\_empty

movb $0xAD, %al

outb $0x64 #disableKeyboard

callwait\_input\_empty

movb $0xD0, %al

outb $0x64 #command-read 8042 output port

callwait\_output\_full

inb $0x60 # got thevalue of 8042 output port and save it

pushb %al

callwait\_input\_empty

movb $0xD1, %al

outb $0x64 #command-write 8042 output port

callwait\_input\_empty

popb %al

orb $0x02, %al #enable A20 Gate

outb $0x60

callwait\_input\_empty

movb $0xAE, %al

outb $0x64 #enableKeyboard

cli

ret

wait\_input\_empty:

rp1: inb $0x64

testb %al, 0x02

jnz rp1

ret

wait\_output\_full:

rp2: inb $0x64

testb %al, 0x01

jz rp2

ret

以上描述的是一种和IBMPC完全兼容的，通过键盘控制器控制A20 Gate的方法。但是，正象我们在前面所提到的，A20 Gate与键盘操作完全没有关系，IBM之所以将A20Gate的功能控制放在键盘控制器上，完全是一种为了弥补Intel 80286与Intel8086/8088不完全兼容的缺陷，而采取的Hacker行为，所以在许多新型PC上存在着一种通过芯片来直接控制A20 Gate的BIOS功能，我们在RealMode下只需要**调用BIOS中断就可以实现A20 Gate的控制功能。**

**这个BIOS中断为 INT 15h, AX=2401h。被称为Fast A20。**

movw $0x2401, %ax

int $0x15