**Chap0.保护模式背景知识**

* **保护模式面面观**
* **实模式、保护模式概述**

在IA32下，CPU有两种工作模式：实模式和保护模式。当系统初启时，CPU是工作在实模式下的，实模式究竟有什么缺点导致我们必须要弃之而进入保护模式呢？

1. 实模式 寻址方式 只能寻址1M空间，而且这1M空间，还有一部分被系统占用，我们能使用的空间不到1M，相比保护模式提供的GB级的寻址能力，实模式唯有退位让贤了

2. 实模式 提供的保护机制比较薄弱，不过对于我们的简单的OS，这个不在考虑范围之内

之前我们提到了保护模式强大的寻址能力，现在让我们深入比较一下实模式和保护模式的寻址方式，看看保护模式到底强大在哪里

首先是实模式的寻址方式：

**物理地址=段值：偏移=段值\*(2^4)+偏移**

由于段寄存器是16位的，故物理地址的寻址范围只有2^20字节，也就是1MB的大小。

接着是保护模式的寻址方式，我们假设还未启分页机制：（未开启分页机制时，线性地址=物理地址，关于分页机制会在本章节后面会做介绍）

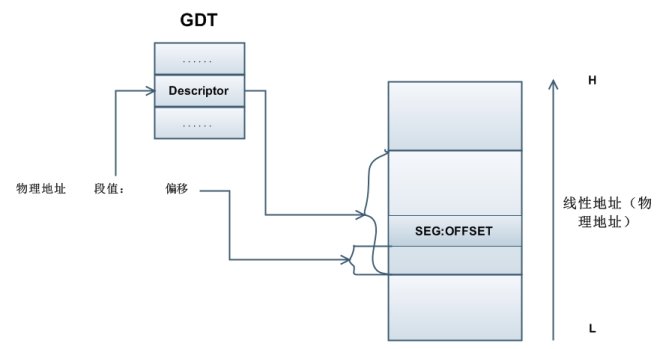


图0.1

我们暂时先不管GDT是什么，只需知道在保护模式下，GDT像一个数组一样记录了一些值

原先段值作为GDT数组的下标（称为段选择子），据此从GDT数组中取出一个32位数作为段基址，再加上相应的偏移，就构成了物理地址，这样物理地址的寻址范围便可扩大为2^32 字节（即4GB），简略的说就是：

**线性地址(物理地址)=GDT[段值] + 偏移**

说到这里，我们对保护模式可谓有了初步的了解，下面我们将从GDT，GDTR，CR0寄存器，段选择子等多个方面来将保护模式一探究竟

现在，相信你一定已经对GDT很好奇，迫不及待了吧。下面我们来看看GDT中到底埋藏着怎样的秘密

* **GDT中的秘密**

之前我们将GDT简化为一个存放段基址的数组，是因为我们极为关注GDT中的段基址，实际上，GDT远不止如此

GDT，全称为全局描述符表（Global Desriptor Table）。GDT在保护模式中的地位极高，没有GDT就没有强大的寻址能力，保护模式的优越性也荡然无存。

其实GDT并不复杂，它就是一张存放段描述符的表，每个段描述符代表一个物理内存段，占8个字节，其结构如下图所示：

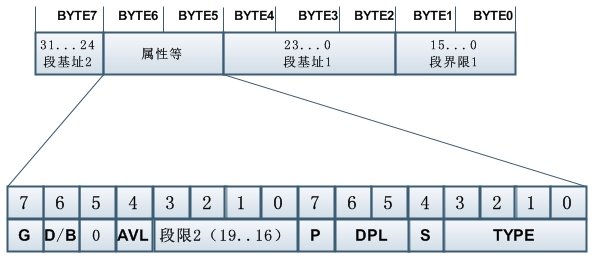


图0.2

关于各属性位的含义：

G：

用于表示段限制域所用的单位

G=0，段限制单位为字节；

G=1，段限制单位为页（4KB）

D/B：

代码段：

D=1，代码段默认的操作和寻址尺寸均为32位

D=0，代表代码段是16位的80286寻址模式，操作和寻址尺寸均为16位

堆栈段：

B=1，隐含的堆栈操作为32位，堆栈指针用ESP

B=0，堆栈操作为16位，堆栈指针用SP

数据段：

B=1，表示数据位32位

B=0，表示数据位16位

AVL：

保留位，可以被系统软件使用

AVL=0，系统软件不可使用本段

AVL=1，系统软件可使用本段

P：

存在位

P=1，该段在内存中

P=0，该段不在内存

DPL：

描述符特权级位，占两位。这两位可表示0~3级特权，表示该段访问的特权级

S：

描述符类型位

S=1，数据/代码段描述符

S=0，系统段/门描述符

TYPE：

段类型位，其具体含义如下图所示：

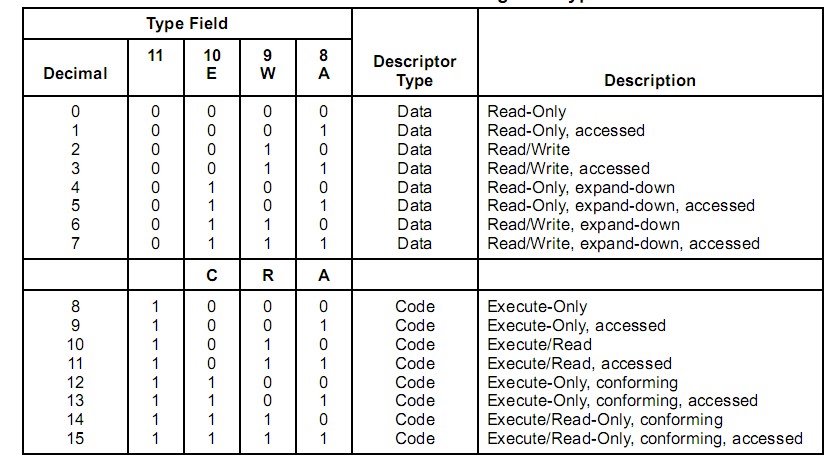


图0.3

说了这么多，其实目前我们最关心的部分只有**段基址**与**段界限。**32位的段基址实现了4G的超大寻址空间，而段界限限制了段的大小，保护了不同的物理地址段之间不能越界访问。这些不正是我们在之前标榜的保护模式的强大功能吗?

对于GDT，我们还有一个问题：

GDT存在于内存之中，系统如何知道内存那一片是GDT？答案是GDTR寄存器。GDTR寄存器的作用是为系统指定GDT表的起始位置与界限，通过lgdt指令可以对GDTR寄存器赋值。GDTR寄存器的结构如下图所示：

https://wave.googleusercontent.com/wave/attachment/%C3%A4%C2%BF%C2%9D%C3%A6%C2%8A%C2%A4%C3%A6%C2%A8%C2%A1%C3%A5%C2%BC%C2%8F3.jpg?id=oOfOZ-Ir2&key=AH0qf5zffOS-ZaUrZIbSgeZWT8fNwaJ37Q

图0.4

* **CR0寄存器**

当我们准备好了GDT表并且加载了GDTR寄存器后，一切已经准备完毕，我们只需拨动开关，就可以摆脱实模式，进入保护模式了。这个开关就在CR0寄存器中，CR0寄存器的结构如下图所示：

https://wave.googleusercontent.com/wave/attachment/CR0.jpg?id=x0NwJHkS2&key=AH0qf5zUJpEjjRT4jlVbIA8pk84GQStZvg

图0.5

在这里我们只需要关心CR0寄存器的第0位（即PE位），就是我们所谓的“开关”，当它被置为1时就表明CPU当前是运行于保护模式下了。

* **段选择子**

我们在 第一部分 Version1 中开始提及段选择子（Selector）的概念，由于这个词不太符合汉语的用于规范（至少我个人这么认为），它让人感觉有些玄乎。其实，段选择子就相当于某个段描述符在GDT表中的偏移量，但又比偏移要稍微复杂些，其结构为：

|  |
| --- |
| https://wave.googleusercontent.com/wave/attachment/%C3%A6%C2%AE%C2%B5%C3%A9%C2%80%C2%89%C3%A6%C2%8B%C2%A9%C3%A5%C2%AD%C2%90.jpg?id=VK5Ph8cZ4&key=AH0qf5wsTvcn3PZLf__Fpw1Jczv7tml2PA |
| 图0.6 |

TI：

TI=0，指示从全局描述符表GDT中读取描述符

TI=1，指示从局部描述符表LDT中读取描述符

RPL：

实现特权级比较时需要用到的两位，关于特权级的问题暂时不作解释

举个例子：当selector = 0x8时，selector代表descriptor[1]，其中TI=0 RPL=0

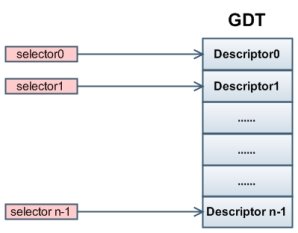


图0.7

* **分页机制**
* **分页机制概述**

相信分页机制这个词大家已经不陌生了，我们在之前多次提到过分页机制这个名词，首先我们需要介绍一下”页“的概念：所谓”页“，就是一块内存，在80386中，页的大小是固定的4K字节（4KB）。

下面就让我们来对对分页机制一探究竟：

在进入保护模式之后，对于地址映射问题我们有两种选择方案：

1. 默认方案 ：将线性地址直接映射为物理地址：**物理地址 = 线性地址 .**

该方案存在一个致命的缺点，物理地址空间一定要与线性地址空间一样大，否则便容易出现寻址越界的错误。例如，对于80386 CPU，由于其线性地址空间大小为4GB，所以物理内存也必须达到4GB。我们都知道，这对当时而言无异于天方夜谭。

2. 分页机制方案：将线性地址间接映射到一块较小的物理内存上（即页目录与页表，详见下文），该物理内存上存储的内容即为真正要寻址的物理地址，简而言之就是：**物理地址 = f（线性地址）**,至于函数f中到底发生了什么，通过下图我们可以略知大概

|  |
| --- |
| https://wave.googleusercontent.com/wave/attachment/%C3%A5%C2%88%C2%86%C3%A9%C2%A1%C2%B5%C3%A6%C2%9C%C2%BA%C3%A5%C2%88%C2%B6.jpg?id=tkSRMokC4&key=AH0qf5yO-ccKMP7rS1-TSGvCl89mbtb3Yg |
| 图0.8 |

对于这个图或许你还不怎么明白，不要紧，随着文档的进行我们会让形势逐步趋于明朗。但至少我们可以知道一点，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，也消除了第一个方案中的致命缺点。

下面我们将具体阐述分页机制进行地址转换的原理，看看线性地址是如何映射为物理地址的：

转换使用两级页表，第一级叫做页目录（Page Directory），大小为4KB，存储在一个物理页中，每个页表4字节长，共有1024个页目录项（Page Directory Entry,简称PDE）。每个页目录项对应第二级的一个大小为4KB的页表（Page Table），第二级的每一个页表也有1024个表项（Page Table Entry ,简称PTE），每一个表项对应一个物理页。

以下为具体转换流程：

1. 由寄存器CR3得到页目录起始地址

2. 将线性地址的高10位作为页目录中的偏移，得到对应页表的起始地址

3. 将线性地址的12-21位作为页表中的偏移，得到物理页的首地址

4. 将这个首地址加上线性地址地址的低12位，便得到了物理地址

至此，地址转换的工作就大功告成了

现在再回过头去看看上图，是否别有一番滋味了？你是否以为自己已经对分页机制了如指掌了？别急，俗话说”细节决定成败“，我们对于PDE,PTE,CR3的细节问题还不甚了解，接下来的工作就是要将这些盲点一一扫除。

* **PDE，PTE，CR3结构剖析**
* **PDE（Page Directory Entry），PTE（Page Table Entry）结构**

|  |
| --- |
| https://wave.googleusercontent.com/wave/attachment/PDE.jpg?id=yqefloTX2&key=AH0qf5ykECKt-I79CkLzSIr9r3nfd0dTLA |
| 图0.9 |

|  |
| --- |
| https://wave.googleusercontent.com/wave/attachment/PTE.jpg?id=mkOCkLUa2&key=AH0qf5wOu2W_r_ZO9qmMb8VYz8Vn9qLfMg |
| 图0.10 |

图中各位解释如下：

Avail:

指示页或页表能否为系统软件所用，我们可以暂时忽略

P：

存在位，表示当前条目所指向的页或页表是否在物理内存中

P=0，表示页不在内存中，如果处理器试图访问此页，将会产生页异常

P=1，表示页在内存中

R/W：

指定一个页或者一组页的读写权限

R/W=0，表示只读

R/W=1，表示可读可写

U/S:

指定一个页的特权级

U/S=0，表示系统级别

U/S=1，表示拥护级别

PWT：

用于控制对单个页或者页表的缓冲策略

PWT=0，使用Write-back缓冲策略

PWT=1，使用Wirte-through缓冲策略

PCD:

用于控制对单个页或者页表的缓冲

PCD=0，页或者页表可以被缓冲

PCD=1，页或者页表不可以被缓冲

A：

指示页或者页表是否被访问，此位往往在页或页表刚刚被加载到屋里内存中时被内存管理程序清零，处理器会在第一次访问此页或页面时设置此位。而且，处理器并不会自动清除此位，只有软件能清除它。

D：（仅存在于PTE中）

指示页或者页表是否被写入，此位往往在页或页表刚刚被加载到屋里内存中时被内存管理程序清零，处理器会在第一次访问此页或页面时设置此位。而且，处理器并不会自动清除此位，只有软件能清除它。

PS：（仅存在于PDE中）

决定页大小

PS=0，页大小为4KB，PDE指向页表

PAT：（仅存在于PTE中）

在80386中该位为保留位，衡为0

G：

指示全局页。由于该位涉及到的背景知识比较复杂，在UnixV6++中并未用到，故此处不予深入解释

* **CR3寄存器结构**

|  |
| --- |
| https://wave.googleusercontent.com/wave/attachment/CR3.jpg?id=mkOCkLUa5&key=AH0qf5zQMQyEWvnzXbXRoujRt-6v1xadQg |
|  |

图0.11

CR3又叫做PDBR（Page-Directory Base Register）。它的高20位将是页目录表的首地址的高20位，页目录表首地址的低12位会是零，也就是说，页目录表会是4KB对齐的。类似地，PDE中的页表基地址（Page-Table Base Address）以及PTE中的页基地址(Page Base Address)也是用高20位来表示4KB对齐的页表和页。至于第3位和第4位的两个标志，我们暂时予以忽略。

## 开启分页机制

事情到了这一步，我们基本上已经将分页机制原原本本的和盘托出，想必你也已经觉得万事俱备了。那我们就来尝试一下开启分页机制吧。

想要开启分页机制，我们必须经过如下步骤：

1.给CR3寄存器赋值，设定页目录的起始地址

2.初始化页目录项与页表项

3.打开分页机制的开关，正式跨入分页机制的门槛

什么？开关在哪？好吧，是我忘了告诉你了，开启分页机制用的与开启保护模式的是同一个“遥控器”，即CR0寄存器。只要将CR0的最高位PG位（见图 0.1）置为1，分页机制就正式开启了。