
LUNAIX
Operating System
从零开始
DEVELOPING YOUR OWN
自制操作系统
OPERATING SYSTEM FROM SCRATCH
理论
Theory
OPERATING SYSTEM DEVELOPMENT
TUTORIAL SERIES
EP 5

顺便提一下

往后的视频中，我将会进行分类

理论
Theory

- 总是纯干货
- 一般篇幅长

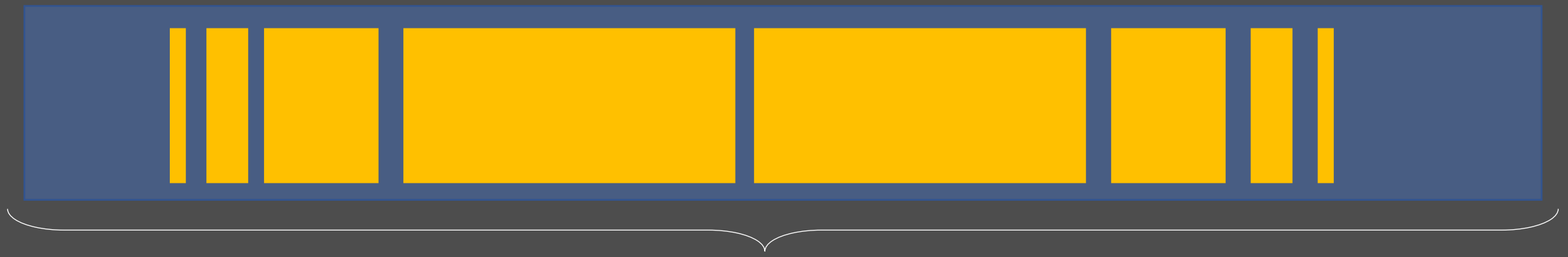
实践
Practice

- 全是代码！
- 有时会穿插一些理论作为补充
- 一般篇幅短

本集内容总结自：
Intel® 64 and IA-32 Architectures Manual (Volume 3A) Chapter 3 *Protected-Mode Memory Management*

保护模式下的内存

- 32 位模式下面内存被分为一个个段（ Segment ）。
- 每个段之间互不干涉，彼此不可见，或许有着不同的权限。
- 段是最小的管理单位（如果我们没有开启分页机制的话）。
- 我们可以把段想象成一个具有特定职能的区域，存放着特定职能的数据。
  （当然，这个职能是有我们开发者自己决定的。）



线性内存空间

注意：段是离散的对象，也就是说，没有必要非得一个紧挨着一个。

保护模式下的寻址

我们如何找到一个段?

- 一个段的起始位置由基地址（Base Address）确定。大小由界限（Limit）决定。
- 所以，通过 **段基地址** + **段内偏移**，我们可以计算出段内任何一个数据的线性地址

和 16 位的寻址：**段地址** ***16+** **偏移地址** 的区别?

- 保护模式下的段允许我们自己定义段的位置和段的长度。
- 因为我们可以使用更大的寄存器了，没有必要在使用 *16 来进行左移四位来争取更大的寻址空间。
- 所以，保护模式下的 **段基地址** + **段内偏移** 的取值可以说是更加的自由。

先暂时不用理会这个词汇的意思，当**分页机制**没有启用时，你只需要知道：**线性地址等于物理地址**。
之后我们会详细展开讲解。

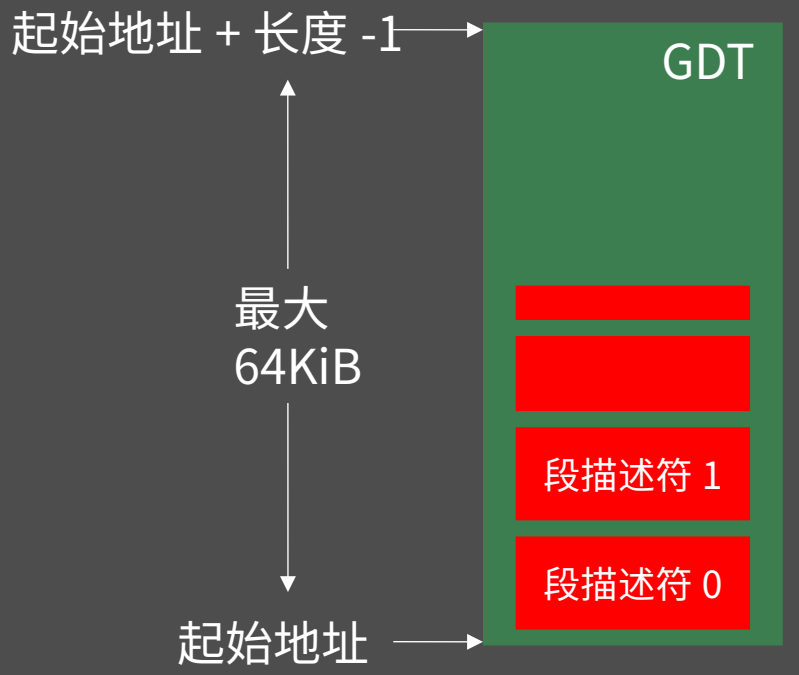
全局描述符表（GDT）

- 全局描述符表是一块内存空间，最大 64KB
- 可以理解为一个数组，里面的每个元素是**段描述符**。
- 全局描述符的地址指针会被存放在一个特殊的寄存器里—— GDTR 。
- GDTR 是一个 48 位大小的寄存器。这是因为全局描述符的地址指针大小也是 48 位。

因为这样， CPU 才能找的到我们这个“段登记表”在内存的哪个旮旯里。

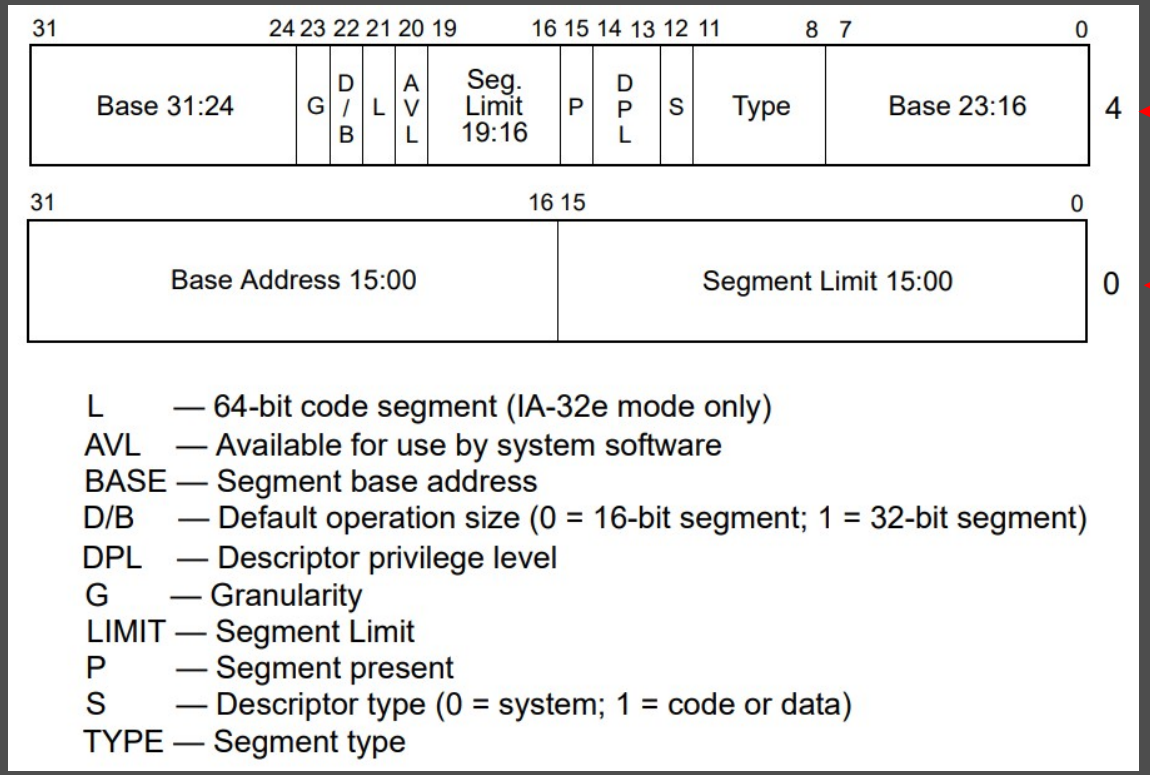
全局描述符的地址指针的格式如下：

这就是为什么，全局描述符表的限制是大小是 64KiB



段描述符（Segment Descriptor）

- 段描述符描述了这个段的特性
- 每个段描述符为 8 个字节，即两个 DWORD
- 以下是段描述符的布局。选自： Intel® 64 and IA-32 Architectures Manual (Volume 3A), Chapter 3.4.5, Figure 3-8



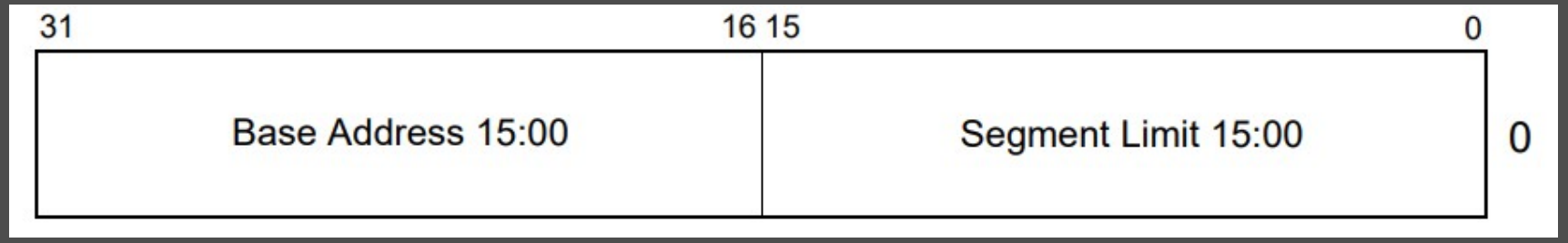
高 32 位

低 32 位

BYTE	字节	8 位
WORD	字	2 BYTEs， 16 位
DWORD	双字	2 WORDs， 4 BYTEs， 32 位
QWORD	四字	2 DWORDs， 8 BYTEs， 64 位

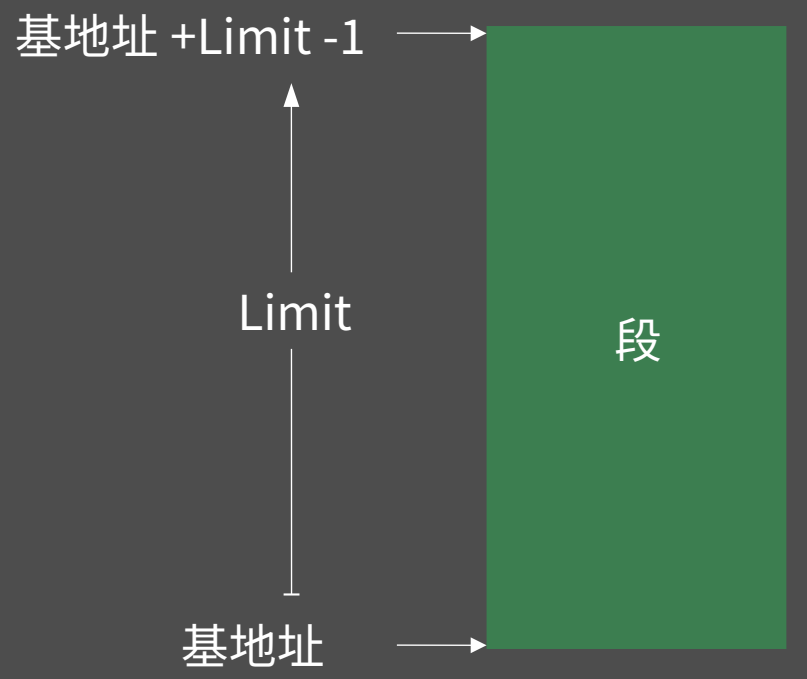
* 字的大小与架构有关，这里只是针对 x86 的情况

段描述符（Segment Descriptor）

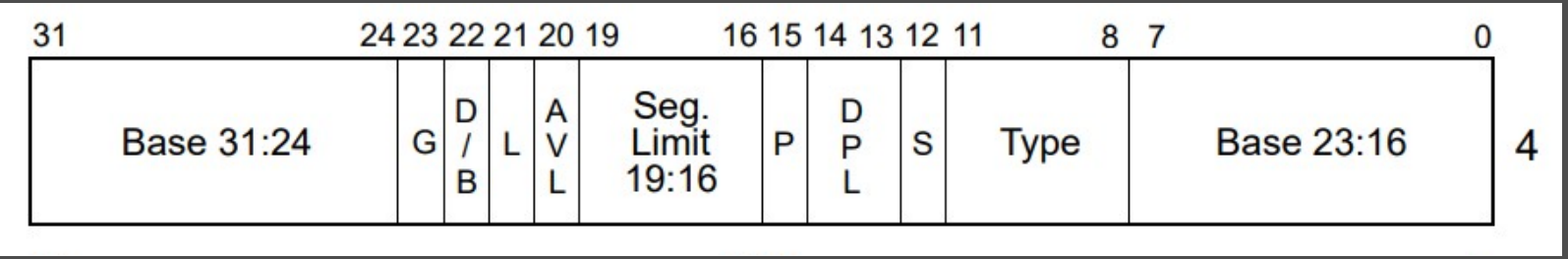


- 第 0~15 位（16 位）：段大小，或者叫做段边界（Limit）
- 第 16~31 位（16 位）：段基地址（线性地址）（Base）

16 位的地址？我们不是在 32 位下面吗?



段描述符（Segment Descriptor）



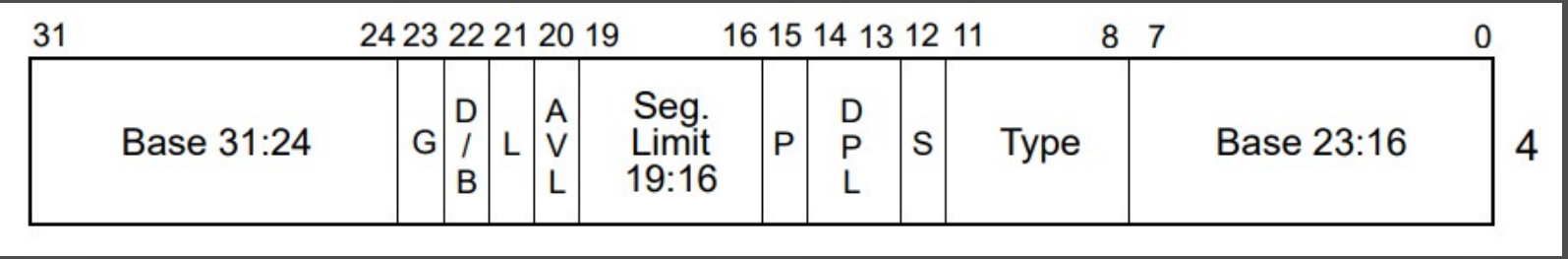
高 32 位

- 第 0~7 位：段基地址的 第 15~23 位。
- 第 24~31 位：段基地址的 第 24~31 位。
- 第 16~19 位：段大小的 第 16~19 位。

可以看到，段的基地址和范围被拆开来存放，这点我们到时在创建自己的 SD 时需要特别注意！

- 第 23 位： G，粒度（Granularity），表明了段大小的单位。
  - G=0： 单位为**一字节**，段大小最大可以为 1MB
  - G=1： 单位为 **4KB**，段大小最大可以为 4GB
- 第 12 位： S，段的职能（段的类型）
  - S=0： 这个段是一个系统段（暂时用不到）。
  - S=1： 这个段是一个代码或数据段。
- 第 15 位： P，指示段是否在内存中，这个涉及到虚拟内存。
  - P=0： 这个段目前在内存中。
  - P=1： 这个段不在在内存中。

段描述符（Segment Descriptor）



高 32 位

- 第 13-14 位（2 位）： DPL 位，指示了段的权限级别
  - DPL=00： Ring 0 级别，最高权限。
  - DPL=01： Ring 1 级别，
  - DPL=10： Ring 2 级别，
  - DPL=11： Ring 3 级别，最底权限。

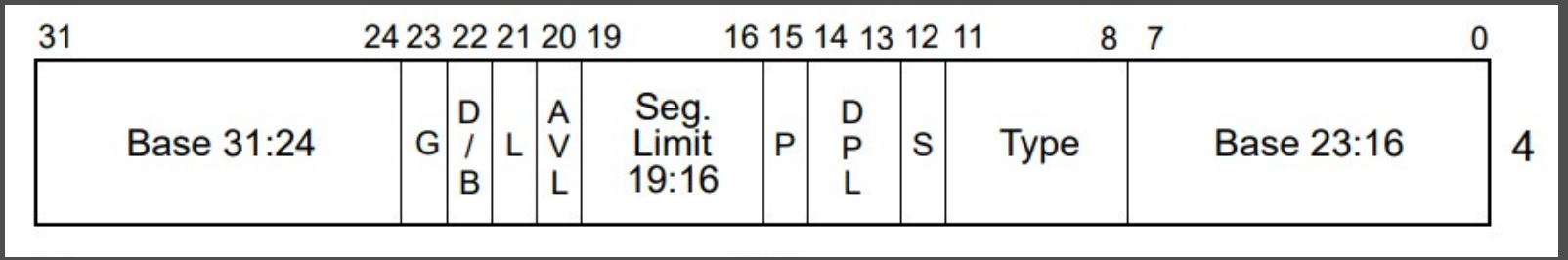
P.s. 处理器会根据这个数值来控制段的访问（从高 DPL 访问低 DPL 的段将会被阻止，除非一些特殊的情况，我们很快会说到）。

- 第 21 位： L，表明这个段是否包含了 64 位的代码。
  - L=0： 包含。
  - L=1： 未包含。

P.s. 由于我们目前主要针对的是 32 位操作系统，所以我们一般是令 L=0 。

- 第 20 位： AVL，软件保留位。没什么特别的，你可以往这里随便填数字。

段描述符（Segment Descriptor）



高 32 位

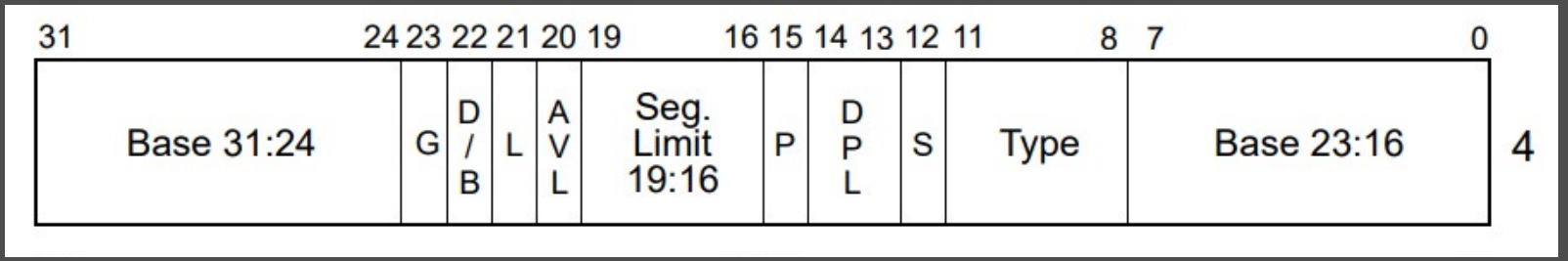
- 第 22 位： D/B ，默认操作数大小，默认栈指针大小或者默认上部边界

  P.s. 关于这一个标志位主要是用于向下兼容 16 位程序的，这允许 16 位程序可以在 32 位的处理器上运行。

  关于这一位的具体信息，大家有兴趣的话，可以自行阅读：
  Intel® 64 and IA-32 Architectures Manual (Volume 3A), Chapter 3.4.5

  同样的，对段描述符每一个字段的解释也都可以在那上面找到。（十分的详细！）

段描述符（Segment Descriptor）



高 32 位

- 第 8~11 位（4 位）： Type，描述符的类型（子类型）
  - Type 的这四位分别是： X,E,W,A 或者 X,C,R,A 。
    - **X**=0，该段是数据段，此时：
      - **E**： 拓展方向 (**E**xpansion-direction)，用于栈段（也算是一种数据段）
        - E=0： 向上拓展。
        - E=1： 向下拓展。
      - **W**： 该段是否可写？ (**W**rite-enable)
        - W=0： 只读。
        - W=1： 读 + 写
    - **X**=1： 该段是代码段，此时：
      - **C**： 是否顺从权限级别。（**C**onforming）
        - C=0： 只允许被处在相同特权级的段里的代码调用。
        - C=1： 允许低特权级的段调用，调用时，调用者的 **CPL** 不变。
      - **R**： 是否可读。（**R**ead-enable）
      - A： 是否被访问（**A**ccessed）
        - 这个是由 CPU 动态的设置的，每当这个段被访问后，这个位就会被拉高（A=1）。

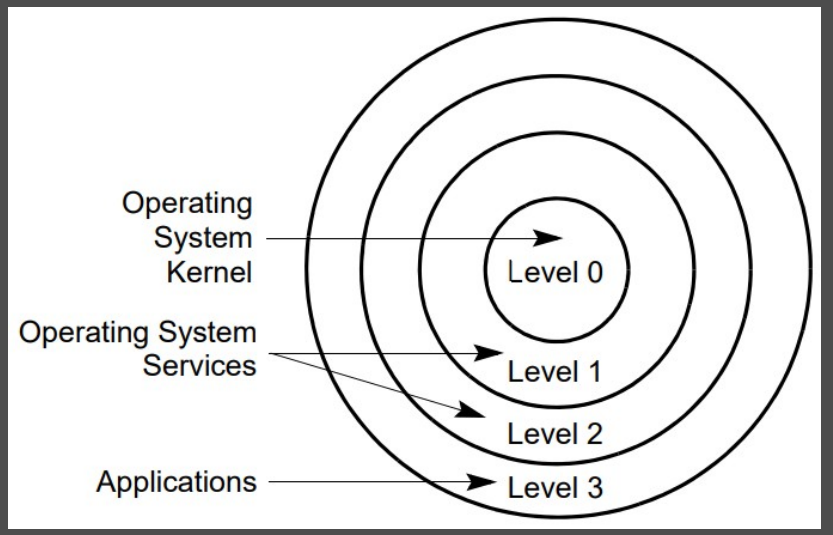
Type			
位 11	位 10	位 9	位 8
X	E/C	W/R	A

特权级

特权级有： 0 ， 1 ， 2 ， 3 这几个等级。

我们可以使用一个同心圆去理解。

- 0 级，最中心的圆，所以权限最高。
- 往外特权级依次**递增**，权限**递减**



用户程序在 3 级，我们的内核会运行在 0 级。中间的等级则留给其他驱动程序或者一些服务。

CPU 如何追踪程序的特权级?

CPL （Current Privilege Level）：当前特权级，记录了当前正在执行的程序的特权级。

注意： CPL 不总是等于 DPL

- 因为正在运行的程序的特权级有可能发生变动。举个不是很恰当的例子： Windows 里面的 “以管理员身份运行”。

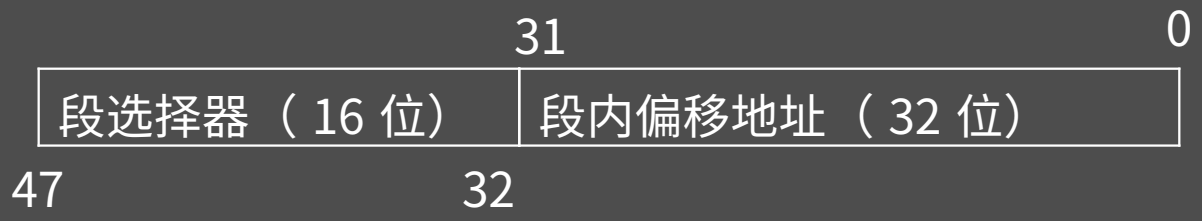
再论寻址 - 逻辑地址

段基地址 + **段内偏移** 可以表示某个段内任意一个字节的线性地址

段的基地址得要通过查询 GDT 才能知道，我们需要另一种表述方式！

保护模式下的逻辑地址：

逻辑地址是一个长度为 48 位的数字，由一个 16 位的段选择器，Segment Selector ，以及 32 位段内偏移组成。



*** 注意，段选择器并不是 GDT 内的偏移！**

再论寻址 - 段选择器

注意

GDT 的第一个 SD （索引 =0 ），是一个**空 SD** ，（所有的位都是 0 ）。如果使用这个 SD 进行寻址的话， CPU 会产生一个 **GP （ General Protection ）异常**，这会直接导致 **Triple Fault !**

- 3~15 位， **Index** ： 段的索引，第几个 SD （就像是操作数组一样）
  - 范围： 0~8191 ，代表着 GDT 中 8192 个 SD
- 2 位， **TI** ： 表指示器（ Table Indicator ）
  - TI=0 ： 这个段描述符在 **GDT** 里
  - TI=1 ： 这个段描述符在 **LDT** 里
- 0~1 位， **RPL** ： 请求权限级。段选择器的权限（一般是构建此选择器的程序的权限）
  - RPL=00 ： Ring 0
  - RPL=01 ： Ring 1
  - RPL=10 ： Ring 2
  - RPL=11 ： Ring 3

$$64 \times 1024 \div 8 = 8192$$

RPL=CPL

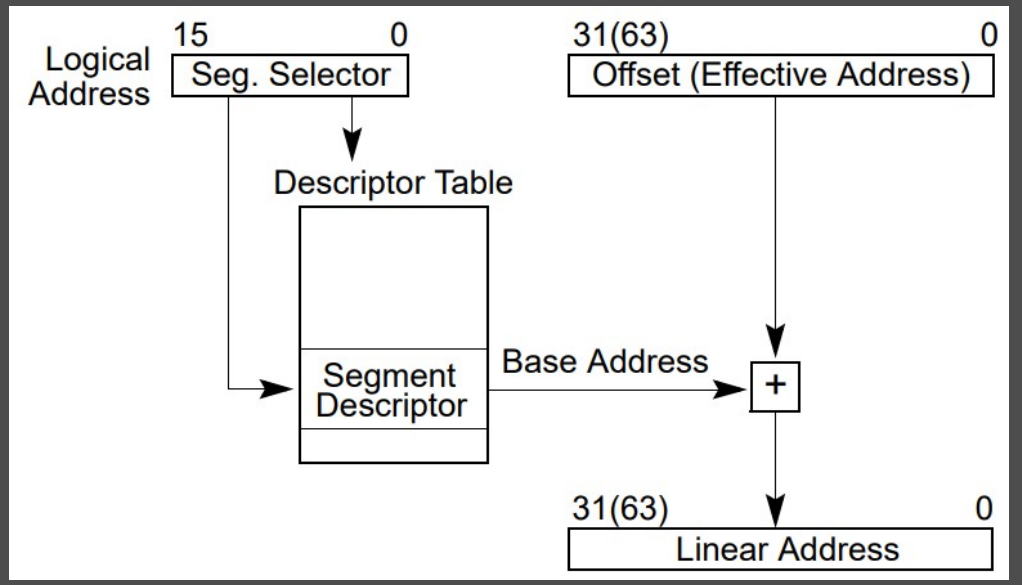
再论寻址 - 逻辑地址

寻址流程（伪代码）：

```
GetLinearAddr(logic_addr):
    selector, offset = logic_addr
    index, ti, rpl = selector
    table = []
    if ti is set:
        table = ldt
    else:
        table = gdt
    sd = table[index * 8]

    if sd == null or not 使用 rpl 去
    判断是否有权限可以访问 sd:
        throw GP

    return sd.base + offset
```



再论寻址 - 段寄存器

四个在 32 位模式下依然是 16 位的寄存器：

不是设计失误！

实模式寻址下他们存放的是段地址。
保护模式寻址下他们存放的是段选择器（ GDT 内的偏移）

实模式下：

段地址 : 偏移地址

保护模式下：

段选择器 : 偏移地址

保护模式还免费赠送了两个段寄存器

注意： **CS 和 SS ， 建议** 指向正确类型的段！

注意：其他四个随便安排。

小提示

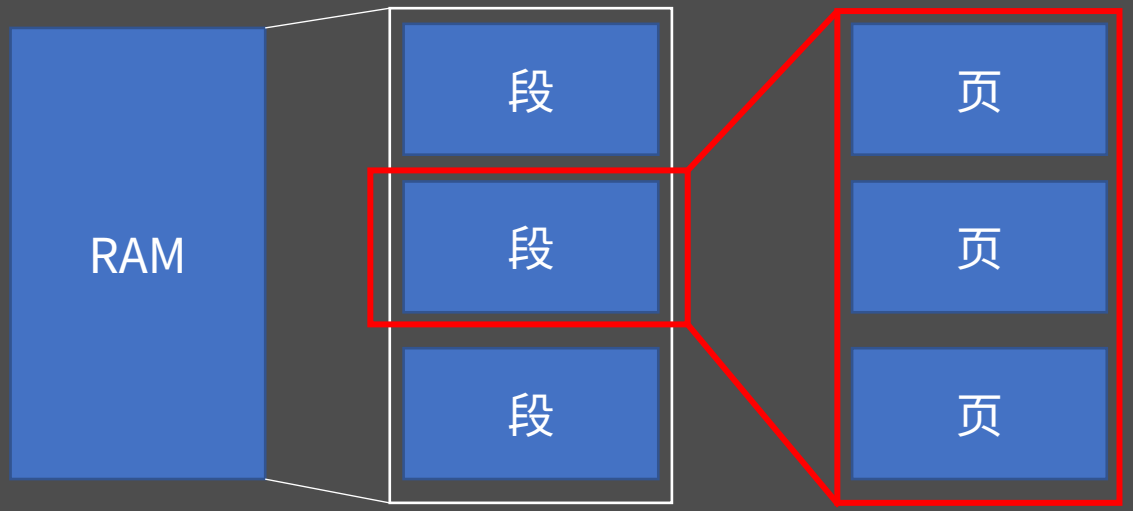
由于 x86 架构采用 **段描述符缓存** 的技术。为了获得更好的性能，这边建议在编写代码时，尽量把常用的段选择器存放在这 6 个寄存器中。

That’s it.

分页（Paging）

分页机制是对段的进一步划分。

内存被进一步划分为更小且更加离散的区域，以实现更加灵活的内存管理。同时减少了内存碎片的产生。



应用场景

- 内存管理的 LRU 算法。
- 虚拟内存。

分页（Paging）

- 分页只有在保护模式下才可被开启。
- 分页默认是不启用的。
- 通过设置 CR0 寄存器的 PG 位可以开启分页。
- 有三种分页模式，可通过设置不同寄存器的指定位来开启：
  - 32 位分页
  - PAE 分页
  - 4 层分页（用于 IA-32e 模式）
- 我们主要着重讨论 32 位分页模式。
- 在这个模式下：
  - 32 位的线性地址
  - 40 位的物理地址
  - 每页大小可以是 4KB 或者 4MB 。

40 位地址？我们不是 32 位的吗？

这个和 PSE 有关系（Page Size Extension），PSE 的特点就是将页的大小从传统的 4KB 拓展到 4MB （两者可以共存）。