> 作 者：道哥，10+年的嵌入式开发老兵，专注于：C/C++、嵌入式、Linux。
>
> 关注下方公众号
>
> > 回复【书籍】，获取 Linux、嵌入式领域经典书籍。
> >
> > 回复【PDF】，获取所有原创文章的 PDF 格式汇总。

目录

x86 处理器中，提供了4个特权级别：0，1，2，3。数字越小，特权级别越高！

一般来说，操作系统是的重要性、可靠性是最高的，需要运行在0 特权级；

应用程序工作在最上层，来源广泛、可靠性最低，工作在3 特权级别。

中间的1 和 2两个特权级别，一般很少使用。

理论上来讲，可以把那些可靠性介于操作系统和应用程序之间的程序安排在这两个特权级上。

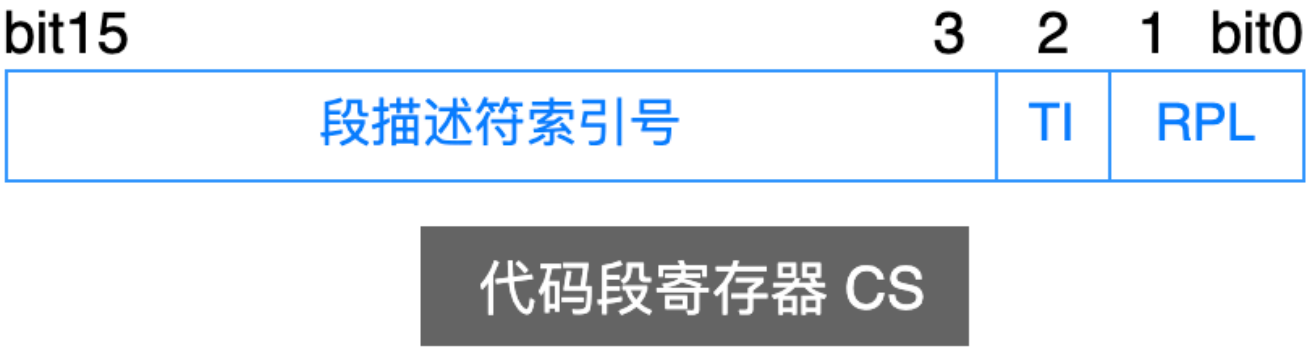
在处理器中，有3个相关的术语与特权级密切相关：

1. CPL: Current Privilege Level 当前特权级;
2. DPL: Descriptor Privilege Level 描述符特权级;
3. RPL: Requestor Privilege Level 请求特权级;

理解了这3个特权级的保护规则，就理解了操作系统保护系统的终极密码！

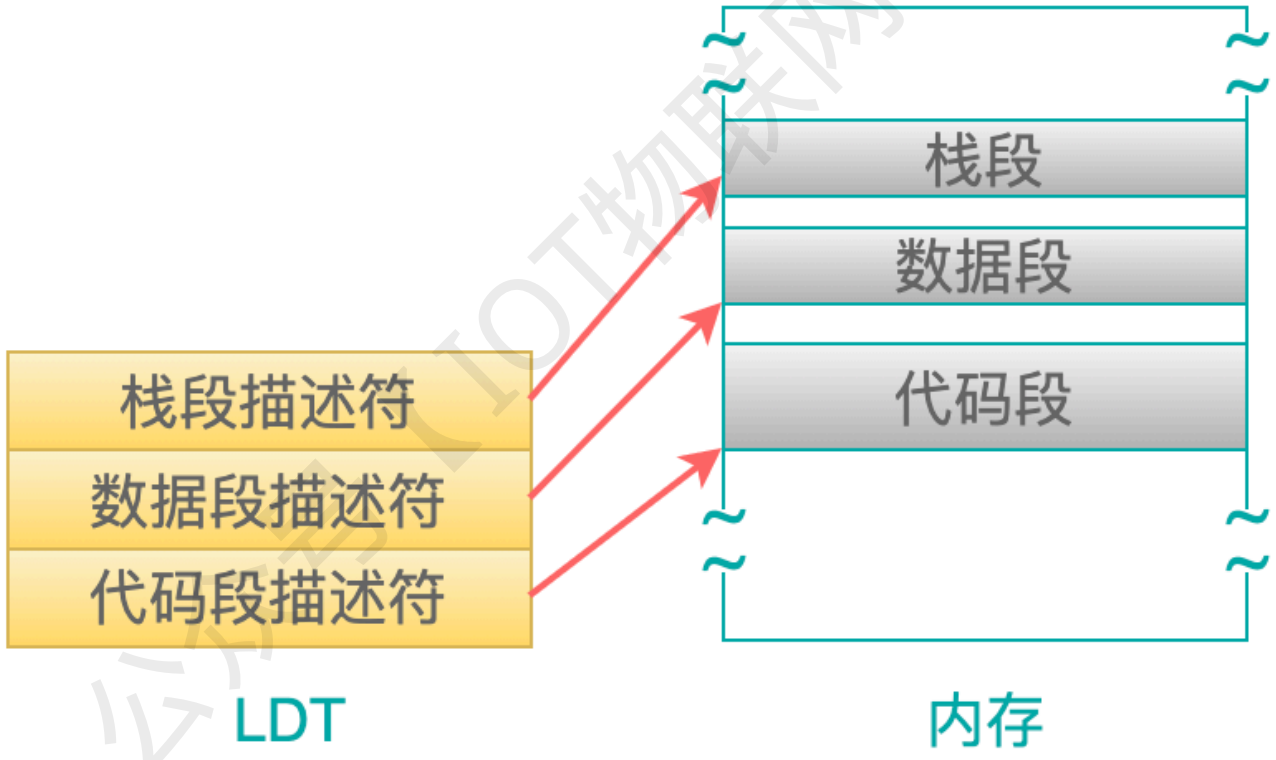
CPL：当前特权级

当前特权级，是指当前正在执行的代码的特权级别，它由当前正在执行的代码段寄存器cs中的bit[1 ~ 0]来决定：



cs寄存器中的最低两位怎么写的是RPL?

原因是这样的：我们在执行一段代码之前，这段代码位于内存中的一段空间中，而它的代码段描述符位于LDT局部描述符表中，如下图所示：



假设现在想进入这个代码段中执行，那么我们就需要给代码段寄存器 cs赋值为：0x0007 (0000_0000_0000_0111)。

此时cs寄存器中当前内容的低两位就称作：当前优先级，而准备赋值给cs的值是0x0007，这个0x0007就称作选择子。

按照选择子的结构来解析：

> 1. RPL: bit[1 ~ 0] = 11B，十进制就是 3，就表示这个选择子的请求特权级别是 3;

2. TI: bit[2] = 1B，表示到 LDT 中查找段描述符;
3. 索引号：bit[15 ~ 3] 的索引值为 0，表示到 LDT 中偏移量为 0 (0 = 0 * 4, 每个描述符占据 4 个字节) 的位置获取段描述符;

当处理器进行一系列权限检查之后，允许进入这段代码中去执行，那么就设置`cs = 0x0007`。

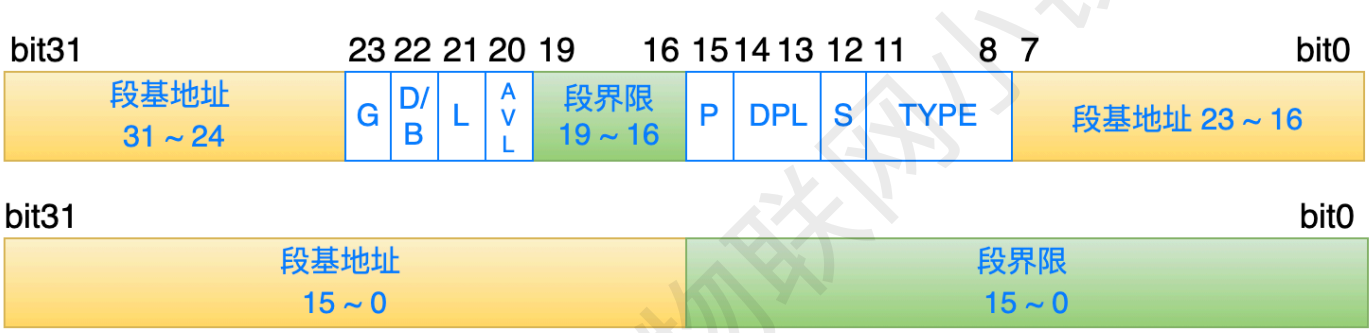
此时cs寄存器中的最低2位就等于选择子中的RPL，也就是3。

在一般情况下，CPL都是等于RPL的。

DPL：描述符特权级

DPL指的是一个段描述符中，用来指定这个描述符所代表的段，具有什么样的特权级别。

关于描述符的结构，如下图所示：



`bit[14 ~ 13]`就表示这个段描述符的特权级别。

当请求访问一个段时(不论是数据段，还是代码段)，处理器在GDT或者LDT中找到段描述符之后，就会把CPL、RPL与描述符中的DPL 进行比较。只有满足一定的规则，才允许访问这个描述符所指向的那个段。

> 具体的比较规则，下文有描述。

RPL：请求者特权级

刚才的CPL内容中，已经描述了RPL是什么东西，它俩是密切相关的。

但是，有时候 CPL 与 RPL 并不相同。

比如：

一个用户程序，想通过操作系统提供的系统函数，去访问内存中的一块用户程序自己的内存空间(数据段)。

用户程序需要告诉操作系统：访问哪一个数据段，偏移量是多少。

这些信息需要把一个选择子通过操作系统来赋值给数据段寄存器 ds。

假设选择子是`0x000F`(二进制：`0000_0000_0000_1111`):

> 索引号：1;

> TI: 使用 LDT;
>
> RPL: 3;

也就是说：当操作系统接受用户程序的请求之后，开始执行系统函数时，此时的CPL是操作系统的特权级别0。

此时操作系统需要把一个选择子赋值给数据段寄存器 ds，而这个选择子是由用户程序作为参数传递给操作系统的。

在这个场景中：CPL = 0, RPL = 3，它俩就不相等。

操作系统用这个选择子0x000F到用户程序的LDT中，根据索引号1找到数据段描述符之后，把CPL(0)、RPL(3)与描述符中的DPL 进行比较，来判断是否有权限访问这个数据段。

> 1. 用户程序的数据段 DPL 一定是 3，这是由操作系统在加载程序之初就决定好的;
> 2. 根据下文的特权级检查规则，这样的访问是允许的;

其实这里有一个隐患：

假如用户程序是一个恶意程序，它想破坏操作系统的数据，于是就传入一个指向操作系统数段的选择子：0x0010(二进制：0000_0000_0001_0000)：

> 索引号：2(假设通过其它途径，知道操作系统的某个数据段位于 GDT 的第 2 个表项);
>
> TI: 使用 GDT;
>
> RPL: 0;

此时，如果操作系统很无脑的就原样接收了用户程序的调用请求，就会通过GDT找到属于操作系统的数据段进行破坏性操作。

操作系统不会这么傻的，它在接收用户程序请求的时候，会严格检查用户程序传入的参数。

如果它发现运行在3 特权级的用户程序，传入一个0 特权级的 RPL，就会警觉：请求特权级竟然比你自己的运行特权级还高，你想干什么?

于是，操作系统就会把选择子中的RPL修改为用户程序的当前特权级CPL。

就好比：一个村长去找市长办事，诉求是：想在自己村的集体土地上盖一座厂房。市长认为：这是你们村自己的土地，你可以随便折腾，准许。

但是，如果村长的诉求是：想在市民广场的旁边盖一座厂房。此时市长就会呵斥：这个地方不是你们村的一亩三分地，想干啥就干啥，滚开！

特权级检查规则

代码段的特权级检查

一般情况下，只允许两个特权级相同的代码段进行转移。

例如：

> 1. 从用户程序的一个代码段(CPL = 3)，跳转到另一个 DPL = 3 的代码段;

> 2. 从操作系统的一个代码段(CPL = 0)，跳转到另一个 DPL = 0 的代码段;

但是处理器也提供了一些特殊途径，让低特权级的代码可以转移到高特权级的代码中去执行：

> 1. 如果在高特权级代码段描述中的 TYPE 字段中，C = 1，就允许低特权级的代码转移进来;
> 2. 通过调用门，低特权级代码也可以转移到高特权级的代码段;

这里主要描述第一种情况，也就是当目标代码段描述符的TYPE字段中C = 1，也就是所谓的依从代码，或者一致性代码。

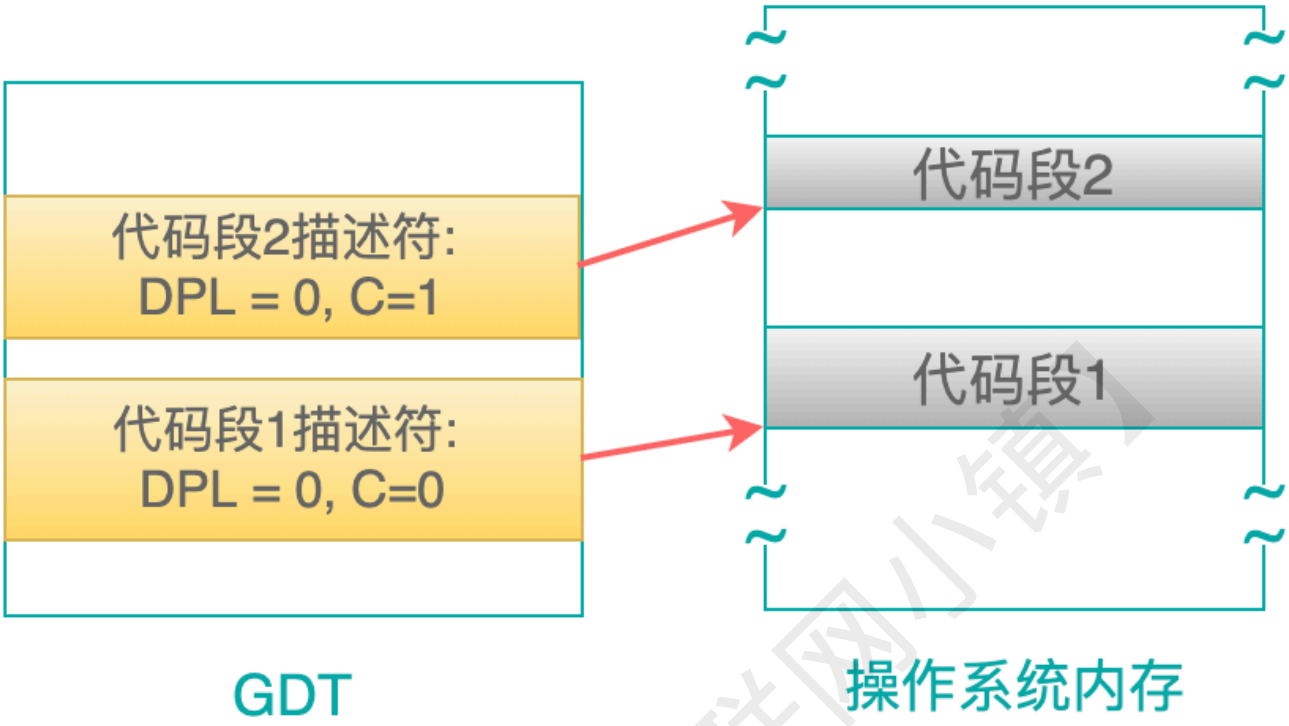
段的类型 TYPE				
	代码段	X	0	XXX
			1	可执行
		C	0	不允许从低特权级直接进入到高特权级
			1	允许从低特权级直接进入到高特权级
		R	0	段内容不可以被读出
			1	段内容可以被读出
		A	0	段最近没有被访问过
			1	段最近被访问过
	数据段	X	0	不可执行
			1	XXX
		E	0	向高地址方向扩展，普通数据段
			1	向低地址方向扩展，栈段
		W	0	段不允许写入
			1	段允许写入
		A	0	段最近没有被访问过
			1	段最近被访问过

也即是说：如果TYPE.C = 1，那么处理器就允许：比这个描述符的 DPL 更低特权级的代码，转移到这个代码中来执行。

在数值上就是：（特权级越低，数值越大）

> CPL >= DPL
>
> RPL >= DPL

例如：操作系统中有2个代码段，它们的描述符中的C标志位不同：



此刻正在执行一个用户程序: CPL = 3。

那么用户程序就可以转移到代码段 2中去执行，不可以转移到代码 1中。

并且，转移到操作系统的代码段2 之后，当前特权级CPL保持不变，仍然为3。

有两个类比：

1. 类似于 Linux 中的 sudo 指令

如果一条指令需要root权限，我们可以使用su -指令，把身份转换到root，然后再去执行。

此时所有的身份、环境变量等信息，都是root用户的。

我们还可以直接使用sudo指令，这时就相当于是临时提升了用户的权限，但是那些环境变量等信息，依然是当前用户的，而不是 root 用户的。

2. 村长找市长办贷款

村长去市里的银行申请贷款，但是自己的权力不够，银行不鸟他，于是村长就去找市长帮忙。

于是，市长就给村长一个亲笔介绍信，村长带着这封信到银行之后，银行一看：有市长大人的背书，于是就给村长办理贷款手续了。

但是，在办理手续的过程中，所有需要签字的地方，只能写村长自己(特权级不变)，而不能写市长的名字。

另外，对于上图中的代码段1，由于其C标志位是0，只能允许相同特权级的程序转移进来，从数值上表示就是：

> CPL == DPL

> RPL == DPL

最后还有一点需要记住：高特权级的代码，永远都不能转移到低特权级的代码。就好比：市长永远都不会以村长的身份去办事。

数据段的特权级检查

数据段的特权级检查规则比较简单：高特权级的程序，可以访问低特权级的数据，反之不可以。

从数值上表示就是：

> CPL <= DPL
>
> RPL <= DPL

栈段的特权级检查

栈段的特权级检查规则，也比较简单，x86 处理器要求当前特权级 CPL 必须与目标栈段的 DPL 相同。

从数值上表示就是：

> CPL == DPL
>
> RPL == DPL

为了满足这个要求，当从用户程序(CPL = 3)转移到操作系统(DPL = 0)时，如果是通过依存(一致性)代码段转移进去，当前特权级是不变的，此时使用的栈仍然是用户程序的栈空间。

如果是通过其他途径转移进去(eg: 调用门)，当前特权级发生了变化(CPL = 0)，此时使用的栈就必须是 0 特权级下的栈空间了。

因此，操作系统在加载这个用户程序的时候，就需要提前申请一块栈空间，以准备在以上这样的场景中使用。

在Linux系统中，只用了0 和 3这两个特权级，因此每一个用户程序只需要提前准备好0特权级下使用的栈就可以了。

如果一个操作系统使用了0 ~ 3所有的四个特权级，那么操作系统就必须为：运行在3特权级下的用户程序准备3个栈空间，用于该用户程序转移到特权级0、1、2 下作为栈空间来使用。

------ End ------

这篇文章主要从特权级的角度，来理解操作系统对系统的保护。

在这样的机制下，操作系统很好的保护了系统不被恶意程序破坏，同时也为用户程序的执行提供了一些通道，来调用更底层的功能。

推荐阅读

【1】C语言指针-从底层原理到花式技巧，用图文和代码帮你讲解透彻
【2】一步步分析-如何用C实现面向对象编程
【3】原来gdb的底层调试原理这么简单
【4】内联汇编很可怕吗？看完这篇文章，终结它！

其他系列专辑：精选文章、C语言、Linux操作系统、应用程序设计、物联网

星标公众号，能更快找到我！

C/C++、物联网、嵌入式、Lua语言

Linux 操作系统、应用程序开发设计

扫码关注公众号

道哥 个人微信

喜欢请分享，满意点个赞，最后点在看。