02 | X86体系结构中的实模式和保护模式

2021-10-27 海纳

《编程高手必学的内存知识》 课程介绍 >

讲述：海纳

时长 19:58 大小 18.29M

你好，我是海纳。

上一节课我们讲了虚拟内存的概念，分析了线性地址（虚拟地址）是如何映射到物理地址上的。

不过，在 X86 架构诞生之初，其实是没有虚拟内存的概念的。1978 年发行的 8086 芯片是 X86 架构的首款芯片，它在内存管理上使用的是直接访问物理内存的方式，这种工作方式，有一个专门的名称，那就是实模式（Real Mode）。上节课我们也曾简单提到过，直接访问物理内存的工作方式让程序员必须要关心自己使用的内存会不会与其他进程产生冲突，为程序员带来极大的心智负担。

后来，CPU 上就出现虚拟内存的概念，它可以将每个进程的地址空间都隔离开，极大地减轻了程序员的负担，同时由于页表项中有多种权限保护标志，极大地提高了应用程序的数据安全。所以人们把 CPU 的这种工作模式称为保护模式（Protection Mode）。

从实模式演进到保护模式，X86 体系架构的内存管理发生了重大的变化，最大的不同就体现在段式管理和中断的管理上。所以今天这节课，我们会围绕这两个重点，让你彻底理解 X86 体系架构下的内存管理演进。你也能通过这节课的学习，学会阅读 Linux 内核源码的段管理和中断管理的相关部分，还可以增加调试 coredump 文件的能力。

这里我们就按照时间顺序，从 8086 芯片中的实模式开始讲起。

8086 中的实模式

8086 芯片是 Intel 公司在 1978 年推出的 CPU 芯片，它定义的指令集对计算机的发展历程影响十分巨大，之后的 286、386、486、奔腾处理器等等都是在 8086 的基础上演变而来。这一套指令集也被称为 X86 指令集。直到今天，很多大学里的微机原理课和汇编语言课还是使用 8086 进行讲解。

8086 的寄存器只有 16 位，我们也习惯于称 8086 的工作模式是 16 位模式。而且，后面的 CPU 为了保持兼容，在芯片上电了以后，还必须运行在 16 位模式之下，这种模式有个正式的名字，叫做**实模式**（Real Mode）。**在实模式下，程序员是不能通过内存管理单元（Memory Management Unit, MMU）访问地址的，程序必须直接访问物理内存。**

那实模式下，我们是怎么访问存储的物理地址的呢？

8086 的寄存器位宽是 16 位，但地址总线却有 20 位，这意味着 8086 的寻址空间是 1M。但是在写程序的时候，我们是没有办法把一个地址完整地放到一个寄存器里的，因为它的寄存器相比地址少了 4 位。

为了解决这个问题，8086 就引入了**段寄存器**，例如 cs、ds、es、gs、ss 等。段寄存器中记录了一个段基地址，通过计算可以得到我们存储的真实地址，也就是物理地址。物理地址可以使用“段寄存器: 段内偏移”这样的格式来表示，计算的公式是：

物理地址 = 段寄存器 << 4 + 段内偏移

不过，在我们写汇编代码的时候，也不一定就要使用段寄存器来表示段基址，也可以使用“段基址: 段内偏移”这样的立即数的写法，比如你可以看下这个节选自 Linux 的 bootsect 中的代码：

复制代码

```
BOOTSEG   = 0x7c0

_start:
  jmpl $BOOTSEG, $start2

start2:
  movw $BOOTSEG, %ax
  movw %ax, %ds
  ...
```

这块代码里，它跳转的目标地址就是 0x7c0 << 4 + OFFSET(start2)。跳转成功以后，cs 段寄存器中的值就是段基址 0x7c0，start2 的偏移值是 8，所以记录当前执行指令地址的 ip 寄存器中的值就是实际地址 0x7c08。

而且，这块代码里也包含了段基址和段内偏移值这种地址形式，这显然有别于我们上节课所讲的虚拟地址。这种包含了段基址和段内偏移值的地址形式有一个专门的名字，叫做**逻辑地址**。你可以看到，虚拟地址是一个整数，而逻辑地址是一对整数。所以说，在 8086 芯片中，逻辑地址要经过一步计算才可以得到物理地址。

在 8086 中，cs 被用来做为代码段基址寄存器，比如上面示例代码中的 jmp 指令，跳转成功就会把段基址自动存入 cs 寄存器。ds 被用来做为数据段基址寄存器，你可以看看下面这个代码：

复制代码

```
INITSEG = 0x9000
    ....
    movw $INITSEG, %ax
    movw %ax, %ds
    movb $0x03, %ah
    xor  %bh,  %bh
    int  $0x10
    movw %dx, (0)
    movb $0x88, %ah
    int  $0x15
    movw %ax, (2)
```

上述代码的第 7 行执行 0x10 号 BIOS 中断，它的结果存放在 dx 寄存器中，然后第 8 行，将结果存入内存 0x90000，9 至 11 行再把 0x15 号 BIOS 中断的结果存到 0x90002 处。

在寻址时，我们并没有明确地声明数据段基址存储在段寄存器 ds 中，但是 CPU 在执行时会默认使用 ds 做为数据段寄存器。类似的还有 ss，它是做为栈基址寄存器，当我们在使用 push 指令的时候，要保存的数据会放在 ss:(sp) 的位置。

CPU 没有强制规定代码段和数据段分离，也就意味着，你使用 ds 段寄存器去访问指令，CPU 也是允许的。但在实际编程时，我们还是会把数据和代码分到不同的段里，并且将数据段的起始地址放到 ds 寄存器，把代码段的地址放到 cs 寄存器。这种按功能分段的管理内存方式就是段式管理。关于段式管理和页式管理的对比，我们稍后会加以介绍。

到这里 8086 的实模式，我们已经基本讲完了。8086 是最古老的 X86 芯片，在实模式下，它只能直接操作物理内存，非常不便于编程，这一点，我们在上节课也提到了。接下来，我们把目光转向 X86 体系架构中的保护模式，它是实模式的进一步发展。

i386 中的保护模式

经过十年的发展，X86 CPU 迎来了历史上使用最广泛、影响力最大的 32 位 CPU，这就是 i386 芯片。i386 与 8086 的一个很大的不同，就是它采用了全新的保护模式。这个体现在，i386 中的段式管理机制，相比 8086 发生了重大变化；同时，i386 芯片在段式管理的基础上，还引入了页式管理。

i386 在完成各种初始化动作以后，就会开启页表，从此程序员就不必再直接操作物理内存的地址空间了，代替它的是线性地址空间。而且由于段和页都能提供对内存的保护，安全性也得到了提升，所以这种工作模式被称为保护模式（Protection Mode）。i386 的保护模式是一种段式管理和页式管理混合使用的模式。

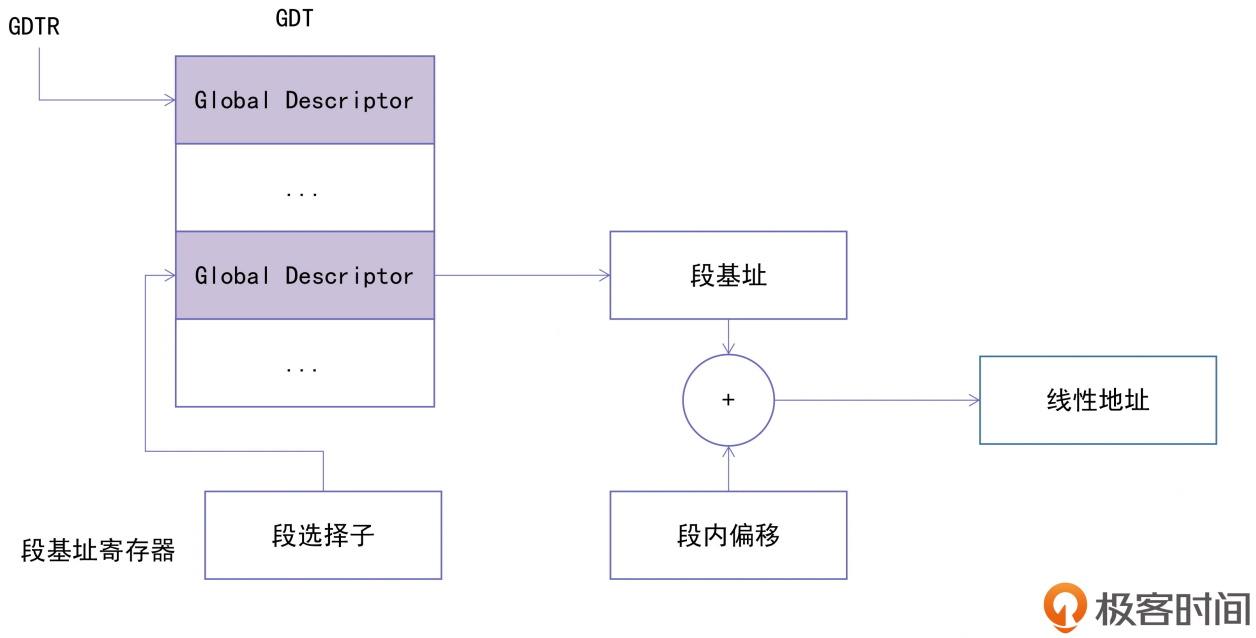
至于页式管理，我们上一节课已经讲过了，所以这里我们就来看一下相比 8086，段式管理在 i386 上有了哪些变化。

变化一：段选择子和全局描述符表

在 i386 上，地址总线是 32 位的，通用寄存器也变成 32 位的，这就意味着因为寄存器位数不够而产生的段基址寄存器已经失去了作用。

但是 i386 没有直接放弃掉段寄存器，而是将它进化成了新的段式内存管理。段寄存器仍然是 16 位寄存器，但是其中存的不再是段基址，而是被称为段选择子的东西。

相比 8086 芯片，i386 中多了一个叫全局描述符表（Global Descriptor Table, GDT）的结构。它本质上是一个数组，其中的每一项都是一个全局描述符，32 位的段基址就存储在这个描述符里。段选择子本质上就是这个数组的下标。具体你可以看看下面这张图：



GDT 的地址也要保存在寄存器里，这个寄存器就是 GDTR，这个做法和上一节课中我们讲到的 CR3 寄存器的做法十分相似。

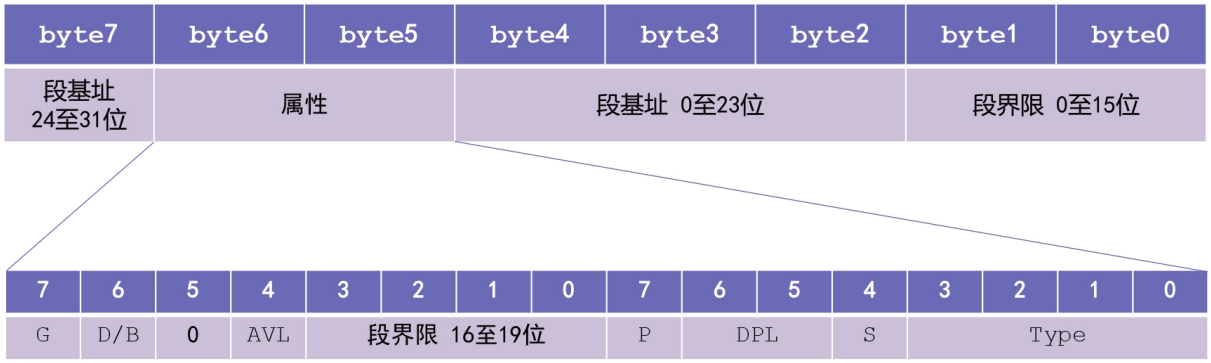
在上面这张图中，CPU 在处理一个逻辑地址“cs:offset”的时候，就会将 GDTR 中的基址加上 cs 中的下标值来得到一个段描述符，再从这个段描述符中取出段基址，最后将段基址与偏移值相加，这样就可以得到线性地址了。这个线性地址就是我们上节课中所讲的虚拟地址。

得到线性地址以后，剩下的工作我们就非常熟悉了：**由 CPU 的 MMU 将线性地址映射为物理地址，然后就可以交给地址总线去进行读写了。**

变化二：段寄存器对段的保护能力增强

在 8086 中，段寄存器只起到了段基址的作用，对于段的各种属性并没有加以定义。例如，在实模式下，任何指令都可以对代码段进行随意地更改。

但在 i386 中，对段的保护能力加强了，我们先来看一下 i386 中段描述符（也就是 GDT 中的每一项）的结构。



极客时间

你会看到，描述符中除了记录了段基址之外，还记录了段的长度，以及定义了一些与段相关的属性，其中比较重要的属性有 P 位、DPL、S 位、G 位和 Type。我们接下来一个个来分析。

P 位是一个比特，指示了段在内存中是否存在，1 表示段在内存中存在，0 则表示不存在。

DPL，占据了两个比特，指的是描述符特权级，英文是 Descriptor Privilege Level。Intel 规定了 CPU 工作的 4 个特权级，分别是 0、1、2、3，数字越小，权限越高。

以 Linux 为例，Linux 只使用了 0 和 3 两个特权级，并且规定 0 是内核态，3 是用户态。特权级的切换是比较复杂的一种机制，但 Linux 只使用了中断这一种，后面我们会再讲到中断。

接下来我们再看 S 位，S 为 1 代表该描述符是数据段 / 代码段描述符，为 0 则代表系统段 / 门描述符。门是 i386 提供的用于切换特权级的机制，有调用门、陷阱门、中断门、任务门等。在 Linux 系统中，只使用了中断门描述符。

然后是 G 位，它指的是定义段颗粒度（Granularity），它的值为 0 时，段界限的单位是字节，为 1 时段界限以 4KB 为单位，也就是一页。

我们也可以从图中看出定义段长度的“段界限”字段并不是连续的，它一共有 20 位，分散在两个地方。当 G=1 时，段界限的最大值是 2^20 * 4K = 4G，这是 i386 一个段的最大长度。

最后是 Type 属性，它定义了描述符类型，我把比较重要的类型用表列在了下面，你可以看看。

Type值	数据段和代码段描述符	门描述符
0x0	只读	undefined
0xA	执行/读	undefined
0xE	执行/读，一致代码段	386中断门

到这里，我们已经解释清楚了，i386 中保护模式相比 8086 实模式在段式管理上的升级。那么在现代的 CPU 和操作系统中，段式管理和页式管理又是怎样的关系呢？要讲清楚这一点就要先对比这两种内存管理方式的优缺点。

段式管理对比页式管理

段式管理会按功能把内存空间分割成不同段，有代码段、数据段、只读数据段、堆栈段，等等，为不同的段赋予了不同的读写权限和特权级。通过段式管理，操作系统可以进一步区分内核数据段、内核代码段、用户态数据段、用户态代码段等，为系统提供了更好的安全性。

但是段的长度往往是不能固定的，例如不同的应用程序中，代码段的长度各不相同。如果以段为单位进行内存的分配和回收的话，数据结构非常难于设计，而且难免会造成各种内存空间的浪费。页式管理则不按照功能区分，而是按照固定大小将内存分割成很多大小相同的页面，不管是存放数据，还是存放代码，都要先分配一个页，再将内容存进页里。

所以，你可以看到，相比页式管理，段式管理的优点是提供更好的安全性，按照内存的用途进行划分更符合人的直观思维。它的缺点就是由于不定长，难于进行分配、回收调度。

而页式管理的优点是大小固定，分配回收都比较容易。而且段式管理所能提供的安全性，在现代 CPU 上也可以被页表项中的属性替代，所以现在段式管理已经变得越来越不重要了。像 64 位 Linux 系统，它把所有段的基地址都设成了从 0 开始，段长度设置为最大。这样段式管理的重要性就大大下降了。

但是，如果我们以 X86 的历史演进来看，你会发现段式管理其实是最早出现的（8086 芯片），然后才出现了页式管理（i386 芯片）。而且，我们现代的 X86 架构的 CPU，也同时兼容段式管理和页式管理，我们可以认为是一种混合的段页式管理（当然，并不是所有人都认可这种命名方式）。

总的来说，现代的操作系统都是采用段式管理来做基本的权限管理，而对于内存的分配、回收、调度都是依赖页式管理。

到这里，我们就讲清楚了 8086 实模式到 i386 保护模式下段式管理的演进，并且进一步分析了段式管理和页式管理的对比和现状。

保护模式相比实模式，发生重大变化的不止是内存管理，同时还有中断管理。因为管理中断的结构与段式管理的全局描述符表的结构非常相似，所以我们在讲保护模式时也一起讲一下。你可以将中断机制与段管理机制比较着一起学习。

中断描述符表

中断描述符表（Interruption Description Table, IDT），是 i386 中一个非常重要的描述符表，它也是保护模式对比实模式的另一大不同。你在后面学习 fork、execve 的实现时，涉及到的写保护中断，缺页中断等机制都要依赖它。

CPU 与外设之间的协同工作是以中断机制来进行的。例如，我们敲击键盘的时候，键盘的控制器就会向 CPU 发起一个中断请求。CPU 在接到请求以后，就会停下正在做的工作，把当前的寄存器状态全部保存好，然后去调用中断服务程序。当然，这个过程中有一些是 CPU 的工作，有一些是操作系统的工作，但因为我们关注的重点是内存，所以就没必要计较这里面细微的差别了。

中断根据中断来源的不同，又可以细分为 Fault、Trap、Abort 以及普通中断。我们这门课对它们也不加区分，例如执行除法的时候除数为 0 的情况、访问数据时权限不足引发的保护错误、由用户使用 int 指令产生的中断等，虽然中断源不同，它们的类型也不相同，但我们统一称它们为中断。

硬件负责产生中断，CPU 会响应中断，但是中断来了以后要做什么事情是由操作系统定义的。操作系统要通过设置某个中断号的中断描述符，来指定中断到达以后要调用的函数。**中断描述符表（IDT）的作用就体现在这了，它的本质就是中断描述符的数组。**

IDT 的基地址存储在 idtr 寄存器中，这和 GDTR 的设计如出一辙。每个中断都有一个编号与其对应，我们称之为**中断向量号**。中断向量号是 CPU 提前分配好的，我也把比较重要的中断向量号放在了下表里，你可以看看。

向量号	助记符	描述	源
0	#DE	除法错	除法指令遇到除数为0
10	#TS	无效TSS	访问TSS时，例如任务切换，特权级切换等
11	#NP	段不存在	加载段寄存器时
12	#SS	栈错误	对栈进行操作时，例如push/pop
13	#GP	保护错误	访问数据或代码时权限不足
14	#PF	页错误	在开启分页的情况下访问内存
32~255	——	用户定义中断	int 指令，Linux主要使用128号中断来进行系统调用

极客时间

在这个表里，我们没有看到前边所提到的键盘中断，这是因为键盘中断都是由一个名为 8259A 的芯片在管理。

两片级联的 8259A 芯片可以管理 16 个中断，其中包括了时钟中断、键盘中断，还有软盘、硬盘、鼠标的中断等等。这些中断的中断向量号是可以通过对 8259A 编程进行设置的。虽然 8259A 的编程比较繁琐，但好在只需要操作系统开机引导时设置一次。

你也可以看到，Linux 系统把中断向量表的 32 号中断（用户自定义中断的第一位）设置成 8259A 的 0 号中断，也就是说 IDT 的 32 号至 47 号都分配给了 8259A 所管理的中断。键盘、软盘、硬盘、鼠标的中断服务程序就设置在这里。

关于中断，我们掌握这么多就已经足够了，更多的知识我们会在后面的课程按需讲解。

现在，我们可以通过一个例子，体验一下中断的使用。在 Linux 系统上，我们把下面这个代码保存到文件 hello.c 中，并且使用"gcc -o hello hello.c"编译，得到可执行程序 hello。再运行它，你就可以看到屏幕上打印出一行"hello"。

复制代码

```
// compile command : gcc -o hello hello.c
void sayHello() {
    const char* s = "hello\n";
    __asm__("int $0x80\n\r"
            ::"a"(4), "b"(1), "c"(s), "d"(6):);
}

int main() {
    sayHello();
    return 0;
}
```

相比于使用 printf 进行打印，需要引入头文件"stdio.h"，我们这段代码里没有使用任何头文件，但一样可以在控制台上进行打印。

这是因为，我们使用了 0x80 号中断进行了 Linux 系统调用。系统调用号在 eax 中，也就是 4，代表 write 这个调用。第一个参数在 ebx 中，其值为 1，代表控制台的标准输出；第二个参数是字符串"hello"的地址，在 rcx 中；第三个参数是字符串的长度，也就是 6，存储在 edx 中。

这样，我们就通过中断，就不必再使用 C 语言的 printf 进行输出，这就绕过了 C 语言的基础库，完成了向控制台打印的功能。

课程小结

今天我们拆解了 X86 体系架构下的实模式和保护模式，也认识了两个 X86 演进史上非常重要的 CPU。

8086 是 16 位的 CPU，我们称 8086 的工作模式为实模式，它的特点是直接操作物理内存，内存管理容易出错，要十分小心，代码编写和调试都很困难。

之后出现的 i386，则采用了和实模式不同的保护模式。相比实模式，i386 中的保护模式，采用了页式管理，但它没有彻底放弃 8086 的段式管理，而是将段寄存器中的值由段基址变成了段选择子。段选择子本质是 GDT 表的下标值，段基址都转移到 GDT 中去了。

段式管理负责将逻辑地址转换为线性地址，或者称为虚拟地址，页式管理负责将线性地址映射到物理地址。i386 的保护模式采用了段页式混合管理的模式，兼具了段式管理和页式管理的优点。

除了段页式内存管理这个不同之外，保护模式和实模式的区别还体现在中断描述符表（IDT）上。IDT 是保护模式的一个重要组成部分，它保存着 i386 中断服务程序的入口地址。

8086 和 i386 对 X86 架构的 CPU 影响巨大。直到今天，X86 架构的 CPU 在上电以后，为了与 8086 保持兼容，还是运行在 16 位实模式下，也就是说所有访存指令访问的都是物理内存地址。在启动操作系统后，才会切换到保护模式下进行工作。

练习题

我们今天这节课只讲了 16 位 CPU 和 32 位 CPU，并没有讲 64 位 CPU 的段式管理是怎么做的。实际上 64 位 CPU 的段式管理和 32 位的结构非常相似，惟一的区别是段描述符的段基址和段长度字段都被废弃了，也就是说不管你将段基址设置成什么，都会被 CPU 自动识别为 0。

那么请你思考，CPU 为什么要这么设计呢？一方面，它还保留了段寄存器，另一方面，它又不再起到逻辑地址转换线性地址的作用，这不是很奇怪吗？请你站在 CPU 架构师的角度思考一下原因。

欢迎把这节课分享给更多对计算机内存感兴趣的朋友。我是海纳，我们下节课见。

吊打面试官

- 段式管理和页式管理会出现内存碎片吗?

段式管理可以做到段根据实际需求分配空间，所以有多少需求就分配多大的段。那么在段的内部就不会产生空间浪费，也就没有碎片，但是由于每个段的长度不固定，所以多个段未必能恰好使用所有的内存空间，也就是说段与段之间还是会产生碎片。而页则是固定大小的，通常是4K，假设我们要为代码准备空间，即使这一段机器码不足4K，我们也要为它分配4K，因为一个页的大小就是4K，我们最少只能分配一个页。所以页式管理，页与页之间紧密排列，但是页内会出现内存浪费，也就是内存碎片。

在服务端岗位面试中

上一篇 01 | 为什么可用内存会远超物理内存？

精选留言 (4)

写留言

Geek_24df3c
2021-10-27

老师给的例子在x86上可以跑通，arm上不行

展开

作者回复: 你说得非常对！看了我们的前导课就会明白，x86和arm的寄存器都不一样，所以这个内嵌汇编只能在x86上运行。汇编是不能跨平台的。不过你倒是可以尝试修改一下哦，这就是跨架构移植了：）

1

慢动作
2021-10-27

没有段以后，代码权限是以什么为单位管理的？GDT是每个进程单独一份，IDT是系统独一份？

作者回复: 1. 第一个问题，linux内核引入了vm_area_struct结构，通过软件的办法做了很多权限管理的事情，这个可以代替段的权限管理的部分工作；另外，每个页也有读写权限管理，所以硬件本身的页管理机制也代替了一些段的权限管理机制。
2. IDT是全局的，你已经理解了。GDT也是全局的。linux会使用GDT来区分内核代码段和内核数据段。每个进程单独的确实也有这种结构，叫做局部（local）表述符表，LDT才是和单个进程相

关的表，其中的描述符的结构与全局描述符是完全一样的。由于它的结构和GDT非常像，我故意略去了。

3. 段选择子我没有展开讲它的结构，实际上，段选择子的第三位为0就表示要在GDT中找描述符，为1就在LDT中查找。所以你只要理解GDT就足够了。但既然你考虑到这里了，我就单独回答一下这个问题。

你思考得很深入，点赞！

1

郑童文

2021-10-27

请问老师，IDT是储存在操作系统的内核内存空间的吗？GDT是存在进程的用户内存空间还是内核内存空间呢？ 谢谢！

展开

作者回复: 好问题。gdt和idt都是由操作系统设置的。所以我们可以理解成是在内核空间里。但这仍然不准确。最准确的说法是由于gdtr和idtr里存的是物理地址，相当于操作系统从物理地址里扣了一块给gdt和idt。这块物理地址以后就不参与分配了。

送过快递的码农

2021-10-27

保留段寄存器是不是为了向下兼容实模式啊？因为64位地址总线够大了，不需要段了。但是为了保证向下兼容性，段不做删除？纯属瞎猜

展开

作者回复: Good，这是很重要的一方面。是x86的设计哲学，但也给x86架构带来了巨大的包袱。