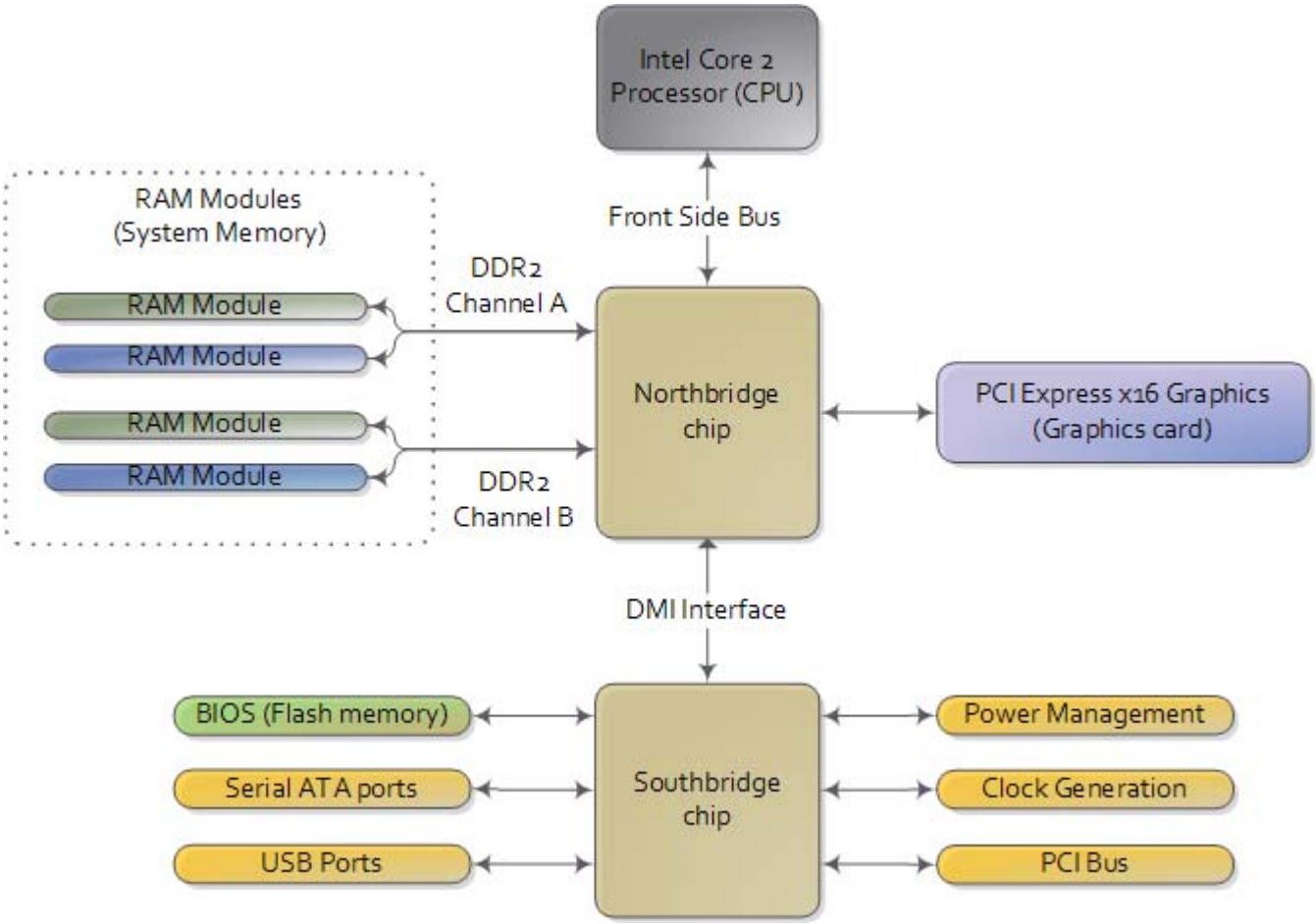
主板芯片组与内存映射

（Motherboard Chipsets and the Memory Map）

一组讲述计算机内幕的文章，旨在揭示现代操作系统内核的工作原理。我希望这些文章能对电脑爱好者和程序员有所帮助，特别是对这类话题感兴趣但没有相关知识的人们。讨论的焦点是 Linux，Windows，和 Intel 处理器。钻研系统内幕是我的一个爱好。我曾经编写过不少内核模式的代码，只是最近一段时间不再写了。这第一篇文章讲述了现代 Intel 主板的布局，CPU 如何访问内存，以及系统的内存映射。

作为开始，让我们看看当今的 Intel 计算机是如何连接各个组件的吧。下图展示了主板上的主要组件：



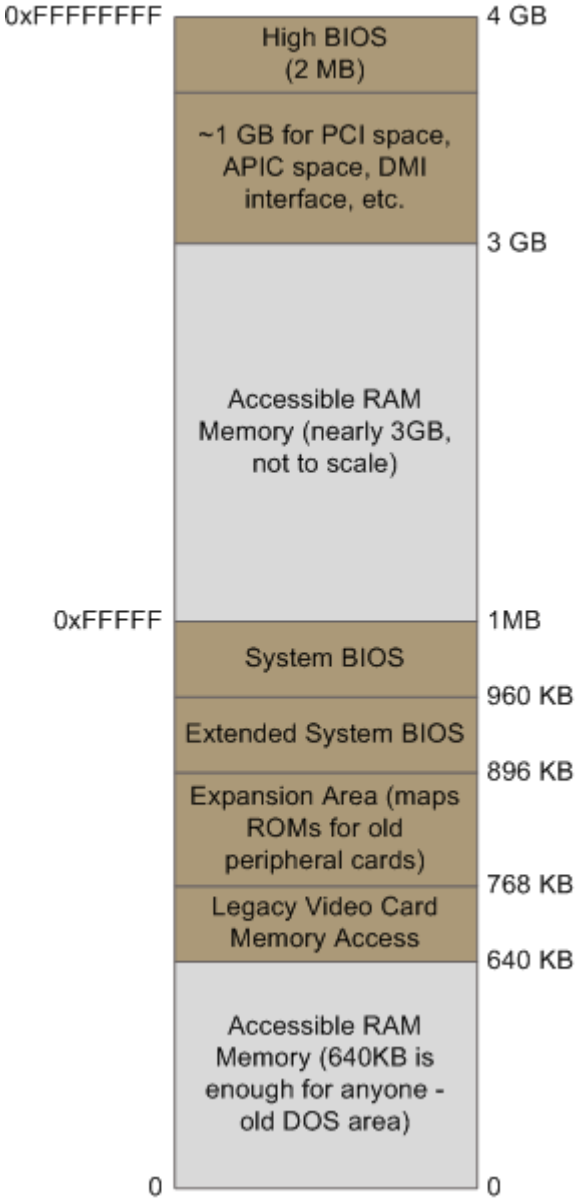
现代主板的示意图，北桥和南桥构成了芯片组。

当你看图时，请牢记一个至关重要的事实：CPU 一点也不知道它连接了什么东西。CPU 仅仅通过一组针脚与外界交互，它并不关心外界到底有什么。可能是一个电脑主板，但也可能是烤面包机，网络路由器，植入脑内的设备，或 CPU 测试工作台。CPU 主要通过 3 种方式与外界交互：内存地址空间，I/O 地址空间，还有中断。

眼下，我们只关心主板和内存。安装在主板上的 CPU 与外界沟通的门户是前端总线（front-side bus），前端总线把 CPU 与北桥连接起来。每当 CPU 需要读写内存时，都会使用这条总线。CPU 通过一部分管脚来传输想要读写的物理内存地址，同时另一些管脚用于发送将被写入或接收被读出的数据。一个 Intel Core 2 QX6600 有 33 个针脚用于传输物理内存地址（可以表示 2^{33} 个地址位置），64 个针脚用于接收/发送数据（所以数据在 64 位通道中传输，也就是 8 字节的数据块）。这使得 CPU 可以控制 64GB 的物理内存（2^{33} 个地址乘以 8 字节），尽管大多数的芯片组只能支持 8GB 的 RAM。

现在到了最难理解的部分。我们可能曾经认为内存指的就是 RAM，被各式各样的程序读写着。的确，大部分 CPU 发出的内存请求都被北桥转送给了 RAM 管理器，但并非全部如此。物理内存地址还可能被用于主板上各种设备间的通信，这种通信方式叫做内存映射 I/O。这类设备包括显卡，大多数的 PCI 卡（比如扫描仪或 SCSI 卡），以及 BIOS 中的 flash 存储器等。

当北桥接收到一个物理内存访问请求时，它需要决定把这个请求转发到哪里：是发给 RAM？抑或是显卡？具体发给谁是由内存地址映射表来决定的。映射表知道每一个物理内存地址区域所对应的设备。绝大部分的地址被映射到了 RAM，其余地址由映射表来通知芯片组该由哪个设备来响应此地址的访问请求。这些被映射为设备的内存地址形成了一个经典的空洞，位于 PC 内存的 640KB 到 1MB 之间。当内存地址被保留用于显卡和 PCI 设备时，就会形成更大的空洞。这就是为什么 32 位的操作系统无法使用全部的 4GB RAM。Linux 中，/proc/iomem 这个文件简明的列举了这些空洞的地址范围。下图展示了 Intel PC 低端 4GB 物理内存地址形成的一个典型的内存映射：



Intel 系统中，低端 4GB 内存地址空间的布局。

实际的地址和范围依赖于特定的主板和电脑中接入的设备，但是对于大多数 Core 2 系统，情形都跟上图非常接近。所有棕色的区域都被设备地址映射走了。记住，这些在主板总线上使用的都是物理地址。在 CPU *内部*（比如我们正在编写和运行的程序），使用的是逻辑地址，必须先由 CPU 翻译成物理地址以后，才能发布到总线上去访问内存。

这个把逻辑地址翻译成物理地址的规则比较复杂，而且还依赖于当时 CPU 的运行模式（实模式，32 位保护模式，64 位保护模式）。不管采用哪种翻译机制，CPU 的运行模式决定了有多少物理内存可以被访问。比如，当 CPU 工作于 32 位保护模式时，它只可以寻址 4GB 物理地址空间（当然，也有个例外叫做物理地址扩展，但暂且忽略这个技术吧）。由于顶部的大约 1GB 物理地址被映射到了主板上的设备，CPU 实际能够使用的也就只有大约 3GB 的 RAM（有时甚至更少，我曾用过一台安装了 Vista 的电脑，它只有 2.4GB 可用）。如果 CPU 工作于实模式，那么

它将只能寻址 1MB 的物理地址空间（这是早期的 Intel 处理器所支持的唯一模式）。如果 CPU 工作于 64 位保护模式，则可以寻址 64GB 的地址空间（虽然很少有芯片组支持这么大的 RAM）。处于 64 位保护模式时，CPU 就有可能访问到 RAM 空间中被主板上的设备映射走了的区域了（即访问空洞下的 RAM）。要达到这种效果，就需要使用比系统中所装载的 RAM 地址区域更高的地址。这种技术叫做回收(reclaiming)，而且还需要芯片组的配合。

这些关于内存的知识将为下一篇文章做好铺垫。下次我们会探讨机器的启动过程：从上电开始，直到 boot loader 准备跳转执行操作系统内核为止。如果你想更深入的学习这些东西，我强烈推荐 Intel 手册。虽然我列出的都是第一手资料，但 Intel 手册写得很好很准确。这是一些资料：

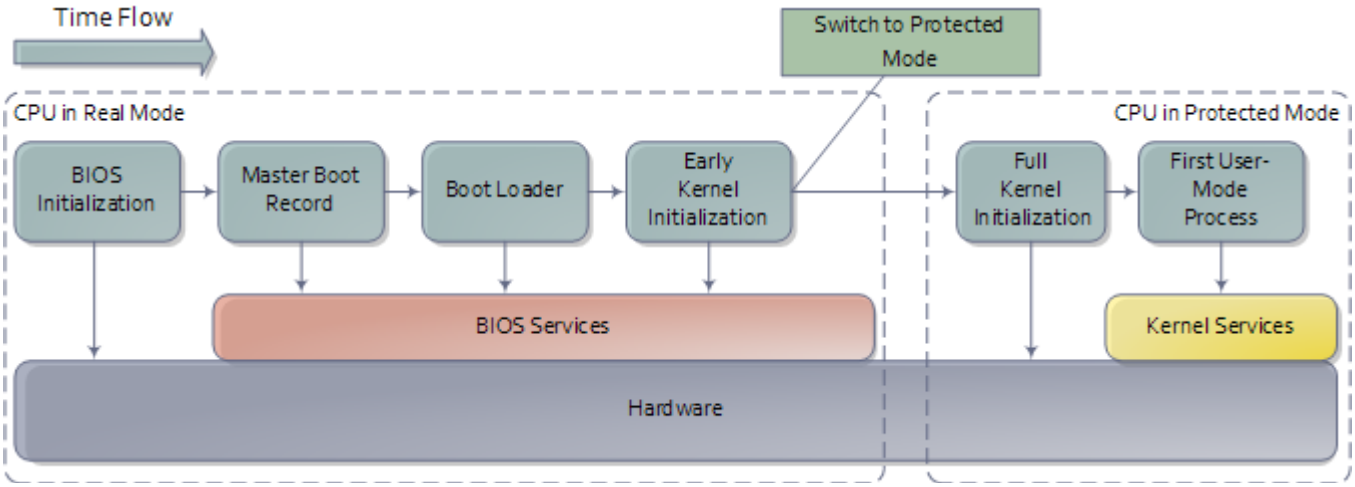
- 《Datasheet for Intel G35 Chipset》描述了一个支持 Core 2 处理器的有代表性的芯片组。这也是本文的主要信息来源。
- 《Datasheet for Intel Core 2 Quad-Core Q6000 Sequence》是一个处理器数据手册。它记载了处理器上每一个管脚的作用（当你把管脚按功能分组后，其实并不算多）。很棒的资料，虽然对有些位的描述比较含糊。
- 《Intel Software Developer's Manuals》是杰出的文档。它优美的解释了体系结构的各个部分，一点也不会让人感到含糊不清。第一卷和第三卷 A 部很值得一读（别被"卷"字吓倒，每卷都不长，而且您可以选择性的阅读）。

计算机的引导过程

（How Computers Boot Up）

前一篇文章介绍了 Intel 计算机的主板与内存映射，从而为本文设定了一个系统引导阶段的场景。引导（Booting）是一个复杂的，充满技巧的，涉及多个阶段，又十分有趣的过程。

下图列出了此过程的概要：



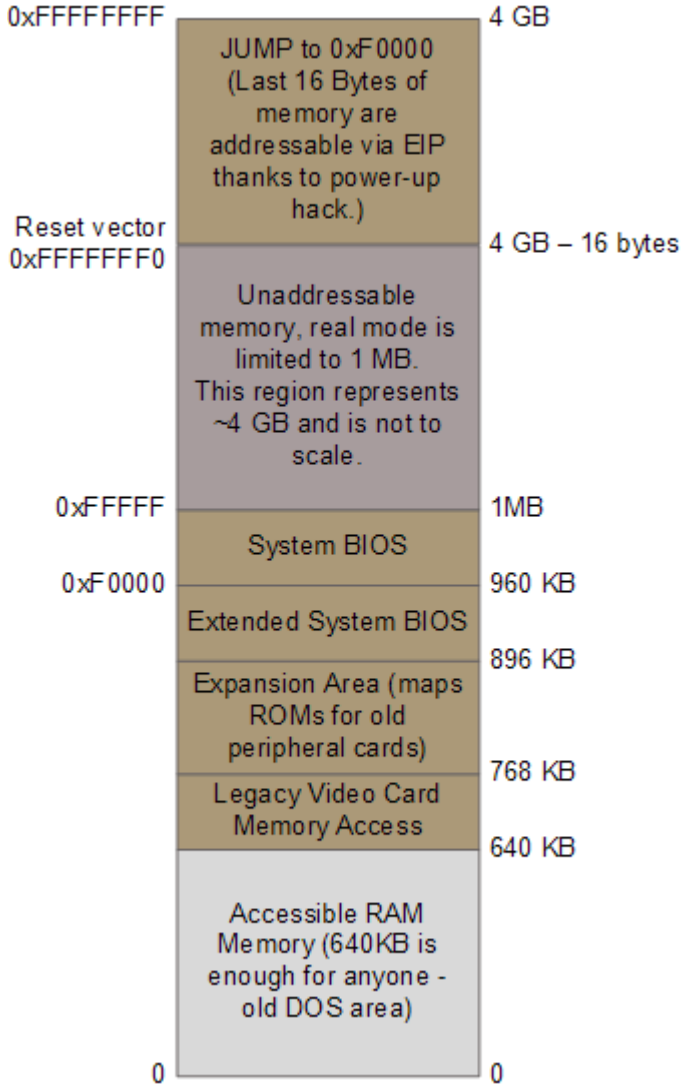
引导过程概要

当你按下计算机的电源键后（现在别按！），机器就开始运转了。一旦主板上电，它就会初始化自身的固件(firmware)——芯片组和其他零零碎碎的东西——并尝试启动 CPU。如果此时出了什么问题（比如 CPU 坏了或根本没装），那么很可能出现的情况是电脑没有任何动静，除了风扇在转。一些主板会在 CPU 故障或缺失时发出鸣音提示，但以我的经验，此时大多数机器都会处于僵死状态。一些 USB 或其他设备也可能导致机器启动时僵死。对于那些以前工作正常，突然出现这种症状的电脑，一个可能的解决办法是拔除所有不必要的设备。你也可以一次只断开一个设备，从而发现哪个是罪魁祸首。

如果一切正常，CPU 就开始运行了。在一个多处理器或多核处理器的系统中，会有一个 CPU 被动态的指派为引导处理器（bootstrap processor 简写 BSP），用于执行全部的 BIOS 和内核初始化代码。其余的处理器，此时被称为应用处理器（application processor 简写 AP），一直保持停机状态直到内核明确激活他们为止。虽然 Intel CPU 经历了很多年的发展，但他们一直保持着完全的向后兼容性，所以现代的 CPU 可以表现得跟原先 1978 年的 Intel 8086 完全一样。其实，当 CPU 上电后，它就是这么做的。在这个基本的上电过程中，处理器工作于实模式，分页功能是无效的。此时的系统环境，就像古老的 MS-DOS 一样，只有 1MB 内存可以寻址，任何代码都可以读写任何地址的内存，这里没有保护或特权级的概念。

CPU 上电后，大部分寄存器的都具有定义良好的初始值，包括指令指针寄存器（EIP），它记录了下一条即将被 CPU 执行的指令所在的内存地址。尽管此时的 Intel CPU 还只能寻址 1MB 的内存，但凭借一个奇特的技巧，一个隐藏的基地址（其实就是个偏移量）会与 EIP 相加，其结果指向第一条将被执行的指令所处的地址 0xFFFFFFF0（长 16 字节，在 4GB 内存空间的尾部，远高于 1MB）。这个特殊的地址叫做复位向量(reset vector)，而且是现代 Intel CPU 的标准。

主板保证在复位向量处的指令是一个跳转，而且是跳转到 BIOS 执行入口点所在的内存映射地址。这个跳转会顺带清除那个隐藏的、上电时的基地址。感谢芯片组提供的内存映射功能，此时的内存地址存放着 CPU 初始化所需的真正内容。这些内容全部是从包含有 BIOS 的闪存映射过来的，而此时的 RAM 模块还只有随机的垃圾数据。下面的图例列出了相关的内存区域：



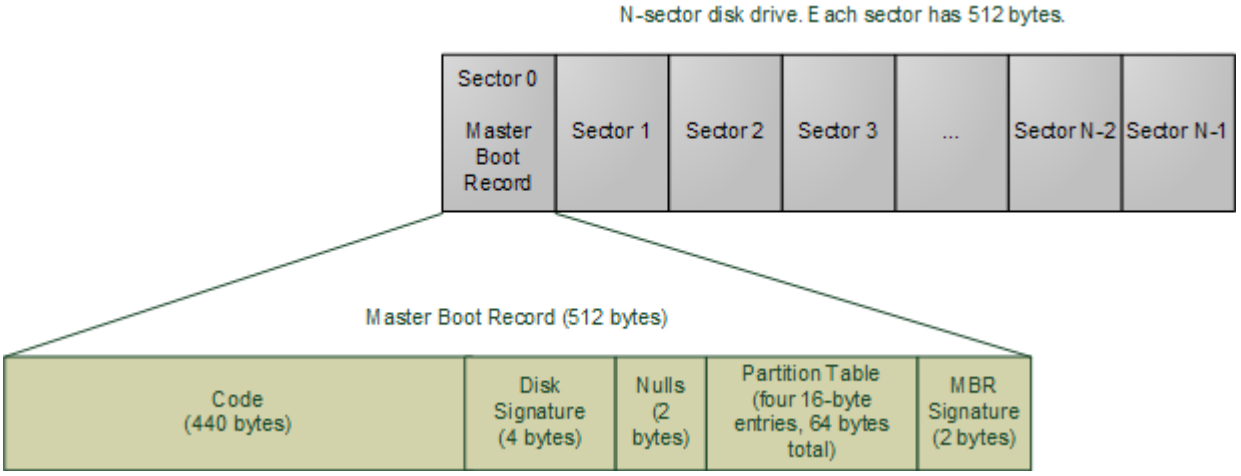
引导时的重要内存区域

随后，CPU 开始执行 BIOS 的代码，初始化机器中的一些硬件。之后 BIOS 开始执行上电自检过程（POST），检测计算机中的各种组件。如果找不到一个可用的显卡，POST 就会失败，导致 BIOS 进入停机状态并发出鸣音提示（因为此时无法在屏幕上输出提示信息）。如果显卡正常，那么电脑看起来就真的运转起来了：显示一个制造商定制的商标，开始内存自检，天使们大声的吹响号角。另有一些 POST 失败的情况，比如缺少键盘，会导致停机，屏幕上显示出错信息。其实 POST 即是检测又是初始化，还要枚举出所有 PCI 设备的资源——中断，内存范围，I/O 端口。现代的 BIOS 会遵循高级配置与电源接口（ACPI）协议，创建一些用于描述设备的数据表，这些表格将来会被操作系统内核用到。

POST 完毕后，BIOS 就准备引导操作系统了，它必须存在于某个地方：硬盘，光驱，软盘等。BIOS 搜索引导设备的实际顺序是用户可定制的。如果找不到合适的引导设备，BIOS 会显示出错信息并停机，比如"Non-System

Disk or Disk Error"没有系统盘或驱动器故障。一个坏了的硬盘可能导致此症状。幸运的是，在这篇文章中，BIOS 成功的找到了一个可以正常引导的驱动器。

现在，BIOS 会读取硬盘的第一个扇区（0 扇区），内含 512 个字节。这些数据叫做主引导记录（Master Boot Record 简称 MBR）。一般说来，它包含两个极其重要的部分：一个是位于 MBR 开头的操作系统相关的引导程序，另一个是紧跟其后的磁盘分区表。BIOS 丝毫不关心这些事情 :它只是简单的加载 MBR 的内容到内存地址 0x7C00 处，并跳转到此处开始执行，不管 MBR 里的代码是什么。



主引导记录

这段在 MBR 内的特殊代码可能是 Windows 引导装载程序，Linux 引导装载程序（比如 LILO 或 GRUB），甚至可能是病毒。与此不同，分区表则是标准化的：它是一个 64 字节的区块，包含 4 个 16 字节的记录项，描述磁盘是如何被分割的（所以你可以在一个磁盘上安装多个操作系统或拥有多个独立的卷）。传统上，Microsoft 的 MBR 代码会查看分区表 ,找到一个(唯一的)标记为活动(active)的分区 ,加载那个分区的引导扇区(boot sector)，并执行其中的代码。引导扇区是一个分区的第一个扇区，而不是整个磁盘的第一个扇区。如果此时出了什么问题，你可能会收到如下错误信息："Invalid Partition Table"无效分区表或"Missing Operating System"操作系统缺失。这条信息不是来自 BIOS 的，而是由从磁盘加载的 MBR 程序所给出的。因此这些信息依赖于 MBR 的内容。

随着时间的推移，引导装载过程已经发展得越来越复杂，越来越灵活。Linux 的引导装载程序 Lilo 和 GRUB 可以处理很多种类的操作系统，文件系统，以及引导配置信息。他们的 MBR 代码不再需要效仿上述"从活动分区来引导"的方法。但是从功能上讲，这个过程大致如下：

MBR 本身包含有第一阶段的引导装载程序。GRUB 称之为阶段一。

由于 MBR 很小，其中的代码仅仅用于从磁盘加载另一个含有额外的引导代码的扇区。此扇区可能是某个分区的引导扇区，但也可能是一个被硬编码到 MBR 中的扇区位置。

MBR 配合第 2 步所加载的代码去读取一个文件，其中包含了下一阶段所需的引导程序。这在 GRUB 中是"阶段二"引导程序，在 Windows Server 中是 C:\NTLDR。如果第 2 步失败了，在 Windows 中你会收到错误信息，比如"NTLDR is missing"NTLDR 缺失。阶段二的代码进一步读取一个引导配置文件（比如在 GRUB 中是 grub.conf，在 Windows 中是 boot.ini）。之后要么给用户显示一些引导选项，要么直接去引导系统。

此时，引导装载程序需要启动操作系统核心。它必须拥有足够的关于文件系统的信息，以便从引导分区中读取内核。在 Linux 中，这意味着读取一个名字类似"vmlinuz-2.6.22-14-server"的含有内核镜像的文件，将之加载到内存并跳转去执行内核引导代码。在 Windows Server 2003 中，一部份内核启动代码是与内核镜像本身分离的，事实上是嵌入到了 NTLDR 当中。在完成一些初始化工作以后 ,NTDLR 从"c:\Windows\System32\ntoskrnl.exe"文件加载内核镜像，就像 GRUB 所做的那样，跳转到内核的入口点去执行。

这里还有一个复杂的地方值得一提（这也是我说引导富于技巧性的原因）。当前 Linux 内核的镜像就算被压缩了，在实模式下，也没法塞进 640KB 的可用 RAM 里。我的 vanilla Ubuntu 内核压缩后有 1.7MB。然而，引导装载程序必须运行于实模式，以便调用 BIOS 代码去读取磁盘，所以此时内核肯定是没法用的。解决之道是使用一种

倍受推崇的"虚模式"。它并非一个真正的处理器运行模式（希望 Intel 的工程师允许我以此作乐），而是一个特殊技巧。程序不断的在实模式和保护模式之间切换，以便访问高于 1MB 的内存同时还能使用 BIOS。如果你阅读了 GRUB 的源代码，你就会发现这些切换到处都是（看看 stage2/目录下的程序，对 real_to_prot 和 prot_to_real 函数的调用）。在这个棘手的过程结束时，装载程序终于千方百计的把整个内核都塞到内存里了，但在这后，处理器仍保持在实模式运行。

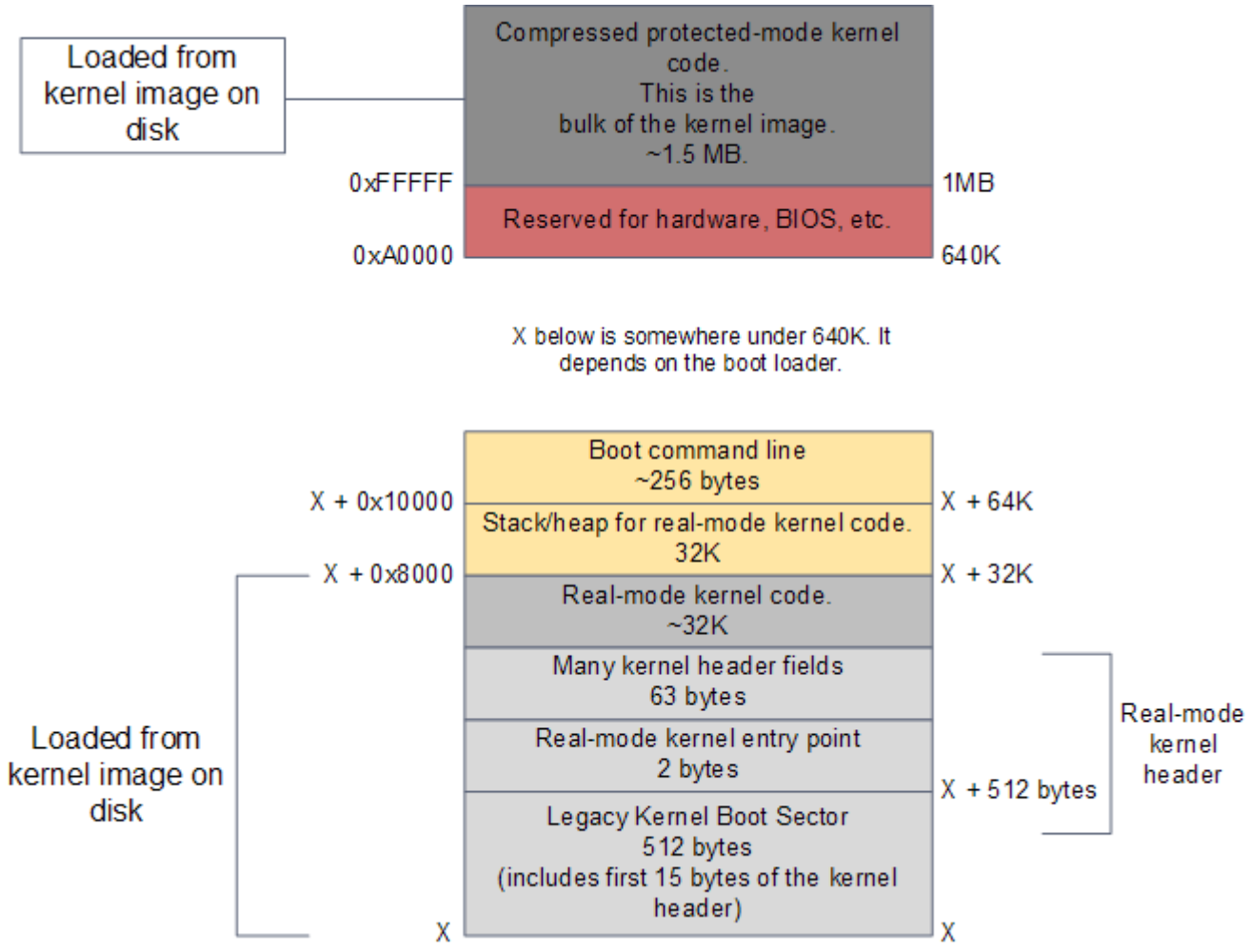
至此，我们来到了从"引导装载"跳转到"早期的内核初始化"的时刻，就像第一张图中所指示的那样。在系统做完热身运动后，内核会展开并让系统开始运转。下一篇文章将带大家一步步深入 Linux 内核的初始化过程，读者还可以参考 Linux Cross reference 的资源。我没办法对 Windows 也这么做，但我会把要点指出来。

内核引导过程

（The Kernel Boot Process）

上一篇文章解释了计算机的引导过程，正好讲到引导装载程序把系统内核镜像塞进内存，准备跳转到内核入口点去执行的时刻。作为引导启动系列文章的最后一篇，就让我们深入内核，去看看操作系统是怎么启动的吧。由于我习惯以事实为依据讨论问题，所以文中会出现大量的链接引用 Linux 内核 2.6.25.6 版的源代码（源自 Linux Cross Reference）。如果你熟悉 C 的语法，这些代码就会非常容易读懂；即使你忽略一些细节，仍能大致明白程序都干了些什么。最主要的障碍在于对一些代码的理解需要相关的背景知识，比如机器的底层特性或什么时候、为什么它会运行。我希望能尽量给读者提供一些背景知识。为了保持简洁，许多有趣的东西，比如中断和内存，文中只能点到为止了。在本文的最后列出了 Windows 的引导过程的要点。

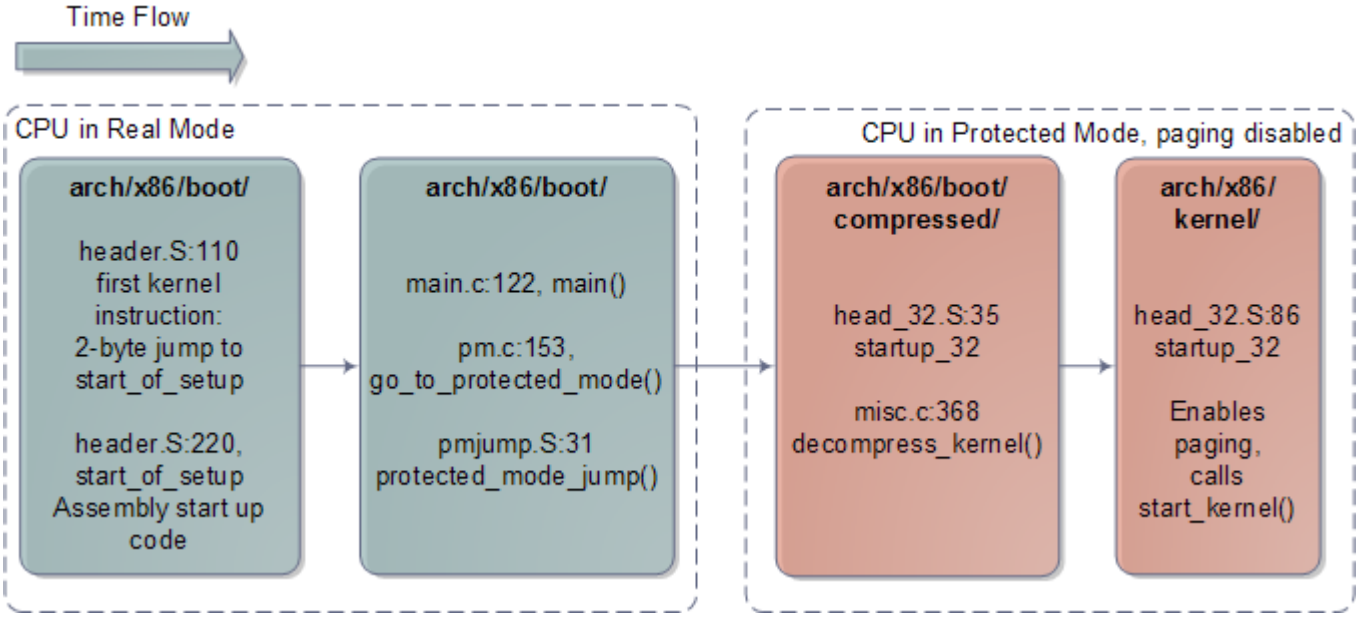
当 Intel x86 的引导程序运行到此刻时，处理器处于实模式（可以寻址 1MB 的内存），（针对现代的 Linux 系统）RAM 的内容大致如下：



引导装载完成后的 RAM 内容

引导装载程序通过 BIOS 的磁盘 I/O 服务，已经把内核镜像加载到内存当中。这个镜像只是硬盘中内核文件（比如/boot/vmlinuz-2.6.22-14-server）的一份完全相同的拷贝。镜像分为两个部分：一个较小的部分，包含实模式的内核代码，被加载到 640KB 内存边界以下；另一部分是一大块内核，运行在保护模式，被加载到低端 1MB 内存地址以上。

如上图所示，之后的事情发生在实模式内核的头部（kernel header）。这段内存区域用于实现引导装载程序与内核之间的 Linux 引导协议。此处的一些数据会被引导装载程序读取。这些数据包括一些令人愉快的信息，比如包含内核版本号的可读字符串，也包括一些关键信息，比如实模式内核代码的大小。引导装载程序还会向这个区域写入数据，比如用户选中的引导菜单项对应的命令行参数所在的内存地址。之后就到了跳转到内核入口点的时刻。下图显示了内核初始化代码的执行顺序，包括源代码的目录、文件和行号：



与体系结构相关的 Linux 内核初始化过程

对于 Intel 体系结构，内核启动前期会执行 arch/x86/boot/header.S 文件中的程序。它是用汇编语言书写的。一般说来汇编代码在内核中很少出现，但常见于引导代码。这个文件的开头实际上包含了引导扇区代码。早期的 Linux 不需要引导装载程序就可以工作，这段代码是从那个时候留传下来的。现今，如果这个引导扇区被执行，它仅仅给用户输出一个"bugger_off_msg"之后就会重启系统。现代的引导装载程序会忽略这段遗留代码。在引导扇区代码之后，我们会看到实模式内核头部（kernel header）最开始的 15 字节；这两部分合起来是 512 字节，正好是 Intel 硬件平台上一个典型的磁盘扇区的大小。

在这 512 字节之后，偏移量 0x200 处，我们会发现 Linux 内核的第一条指令，也就是实模式内核的入口点。具体的说，它在 header.S:110，是一个 2 字节的跳转指令，直接写成了机器码的形式 0x3AEB。你可以通过对内核镜像运行 hexdump，并查看偏移量 0x200 处的内容来验证这一点——这仅仅是一个对神志清醒程度的检查，以确保这一切并不是在做梦。引导装载程序运行完毕时就会跳转执行这个位置的指令，进而跳转到 header.S:229 执行一个普通的用汇编写成的子程序，叫做 start_of_setup。这个短小的子程序初始化栈空间（stack），把实模式内核的 bss 段清零（这个区域包含静态变量，所以用 0 来初始化它们），之后跳转执行一段又老又好的 C 语言程序：arch/x86/boot/main.c:122。

main()会处理一些登记工作（比如检测内存布局），设置显示模式等。然后它会调用 go_to_protected_mode()。然而，在把 CPU 置于保护模式之前，还有一些工作必须完成。有两个主要问题：中断和内存。在实模式中，处理器的中断向量表总是从内存的 0 地址开始的，然而在保护模式中，这个中断向量表的位置是保存在一个叫 IDTR 的 CPU 寄存器当中的。与此同时，从逻辑内存地址（在程序中使用）到线性内存地址（一个从 0 连续编号到内存顶端的数值）的翻译方法在实模式和保护模式中是不同的。保护模式需要一个叫做 GDTR 的寄存器来存放内存全局描述

符表的地址。所以 go_to_protected_mode()调用了 setup_idt()和 setup_gdt()，用于装载临时的中断描述符表和全局描述符表。

现在我们可以转入保护模式啦，这是由另一段汇编子程序 protected_mode_jump 来完成的。这个子程序通过设定 CPU 的 CR0 寄存器的 PE 位来使能保护模式。此时，分页功能还处于关闭状态；分页是处理器的一个可选的功能，即使运行于保护模式也并非必要。真正重要的是，我们不再受制于 640K 的内存边界，现在可以寻址高达 4GB 的 RAM 了。这个子程序进而调用压缩状态内核的 32 位内核入口点 startup_32。startup32 会做一些简单的寄存器初始化工作，并调用一个 C 语言编写的函数 decompress_kernel()，用于实际的解压缩工作。

decompress_kernel()会打印一条大家熟悉的信息"Decompressing Linux..."（正在解压缩 Linux）。解压缩过程是原地进行的，一旦完成内核镜像的解压缩，第一张图中所示的压缩内核镜像就会被覆盖掉。因此解压后的内核也是从 1MB 位置开始的。之后，decompress_kernel()会显示"done"（完成）和令人振奋的"Booting the kernel"（正在引导内核）。这里"Booting"的意思是跳转到整个故事的最后一个入口点，也是保护模式内核的入口点，位于 RAM 的第二个 1MB 开始处（偏移量 0x100000，此值是由芬兰 Halti 山巅之上的神灵授意给 Linus 的）。在这个神圣的位置含有一个子程序调用，名叫...呃...startup_32。但你会发现*这一位*是在另一个目录中的。这位 startup_32 的第二个化身也是一个汇编子程序，但它包含了 32 位模式的初始化过程：

它清理了保护模式内核的 bss 段。（这回是*真正*的内核了，它会一直运行，直到机器重启或关机。）

1) 为内存建立最终的全局描述符表。

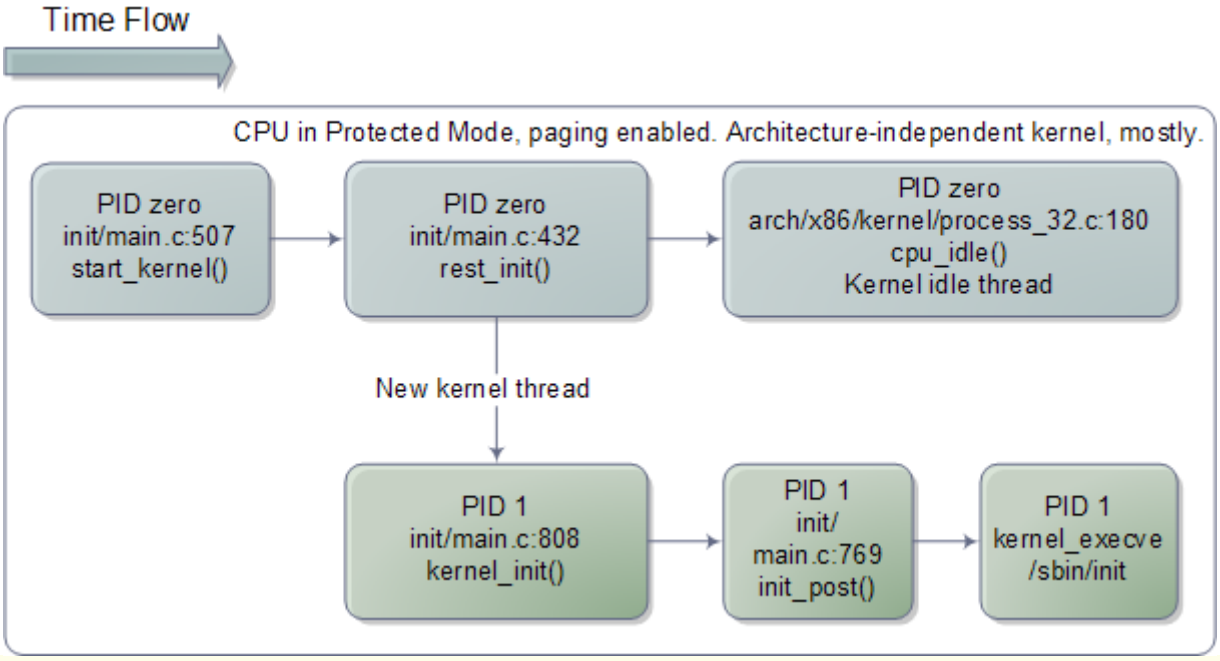
2)建立页表以便可以开启分页功能使能分页功能。

3)初始化栈空间。

4)创建最终的中断描述符表。

5)最后，跳转执行一个体系结构无关的内核启动函数：start_kernel()。

下图显示了引导最后一步的代码执行流程：



与体系结构无关的 Linux 内核初始化过程

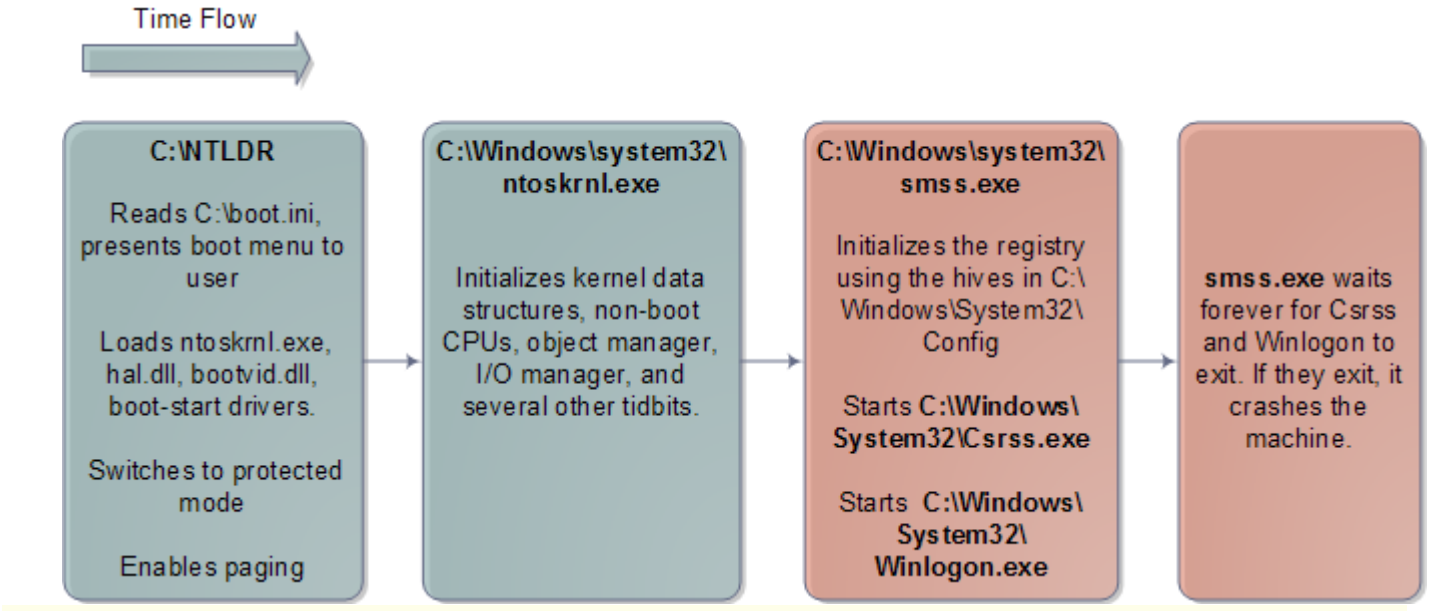
start_kernel()看起来更像典型的内核代码，几乎全用 C 语言编写而且与特定机器无关。这个函数调用了一长串的函数，用来初始化各个内核子系统和数据结构，包括调度器（scheduler），内存分区（memory zones），计时器（time keeping）等等。之后，start_kernel()调用 rest_init()，此时几乎所有的东西都可以工作了。rest_init()会创建一个内核线程，并以另一个函数 kernel_init()作为此线程的入口点。之后，rest_init()会调用 schedule()来激活任务调度功能，然后调用 cpu_idle()使自己进入睡眠（sleep）状态，成为 Linux 内核中的一个空闲线程（idle

thread）。cpu_idle()会在 0 号进程（process zero）中永远的运行下去。一旦有什么事情可做，比如有了一个活动就绪的进程(runnable process),0 号进程就会激活 CPU 去执行这个任务,直到没有活动就绪的进程后才返回。

但是,还有一个小麻烦需要处理。我们跟随引导过程一路走下来,这个漫长的线程以一个空闲循环(idle loop)作为结尾。处理器上电执行第一条跳转指令以后，一路运行，最终会到达此处。从复位向量（reset vector）->BIOS->MBR->引导装载程序->实模式内核->保护模式内核，跳转跳转再跳转，经过所有这些杂七杂八的步骤，最后来到引导处理器（boot processor）中的空闲循环 cpu_idle()。看起来真的很酷。然而，这并非故事的全部，否则计算机就不会工作。

在这个时候，前面启动的那个内核线程已经准备就绪，可以取代 0 号进程和它的空闲线程了。事实也是如此，就发生在 kernel_init()开始运行的时刻（此函数之前被作为线程的入口点）。kernel_init()的职责是初始化系统中其余的 CPU，这些 CPU 从引导过程开始到现在，还一直处于停机状态。之前我们看过的所有代码都是在一个单独的 CPU 上运行的,它叫做引导处理器(boot processor)。当其他 CPU——称作应用处理器(application processor)——启动以后,它们是处于实模式的,必须通过一些初始化步骤才能进入保护模式。大部分的代码过程都是相同的，你可以参考 startup_32，但对于应用处理器，还是有些细微的不同。最终，kernel_init()会调用 init_post()，后者会尝试启动一个用户模式（user-mode）的进程，尝试的顺序为：/sbin/init，/etc/init，/bin/init，/bin/sh。如果都不行，内核就会报错。幸运的是 init 经常就在这些地方的，于是 1 号进程（PID 1）就开始运行了。它会根据对应的配置文件来决定启动哪些进程，这可能包括 X11 Windows，控制台登陆程序，网络后台程序等。从而结束了引导进程,同时另一个 Linux 程序开始在某处运行。至此,让我祝福您的电脑可以一直正常运行下去,不出毛病。

在同样的体系结构下，Windows 的启动过程与 Linux 有很多相似之处。它也面临同样的问题，也必须完成类似的初始化过程。当引导过程开始后，一个最大的不同是，Windows 把全部的实模式内核代码以及一部分初始的保护模式代码都打包到了引导加载程序(C：\NTLDR)当中。因此，Windows 使用的二进制镜像文件就不一样了，内核镜像中没有包含两个部分的代码。另外，Linux 把引导装载程序与内核完全分离，在某种程度上自动的形成不同的开源项目。下图显示了 Windows 内核主要的启动过程：



Windows 内核初始化过程

自然而然的，Windows 用户模式的启动就非常不同了。没有/sbin/init 程序，而是运行 Csrss.exe 和 Winlogon.exe。Winlogon 会启动 Services.exe（它会启动所有的 Windows 服务程序）、Lsass.exe 和本地安全认证子系统。经典的 Windows 登陆对话框就是运行在 Winlogon 的上下文中的。

本文是引导启动系列话题的最后一篇。感谢每一位读者，感谢你们的反馈。我很抱歉，有些内容只能点到为止；我打算把它们留在其他文章中深入讨论，并尽量保持文章的长度适合 blog 的风格。下次我打算定期的撰写关于“Software Illustrated”的文章，就像本系列一样。最后，给大家一些参考资料：

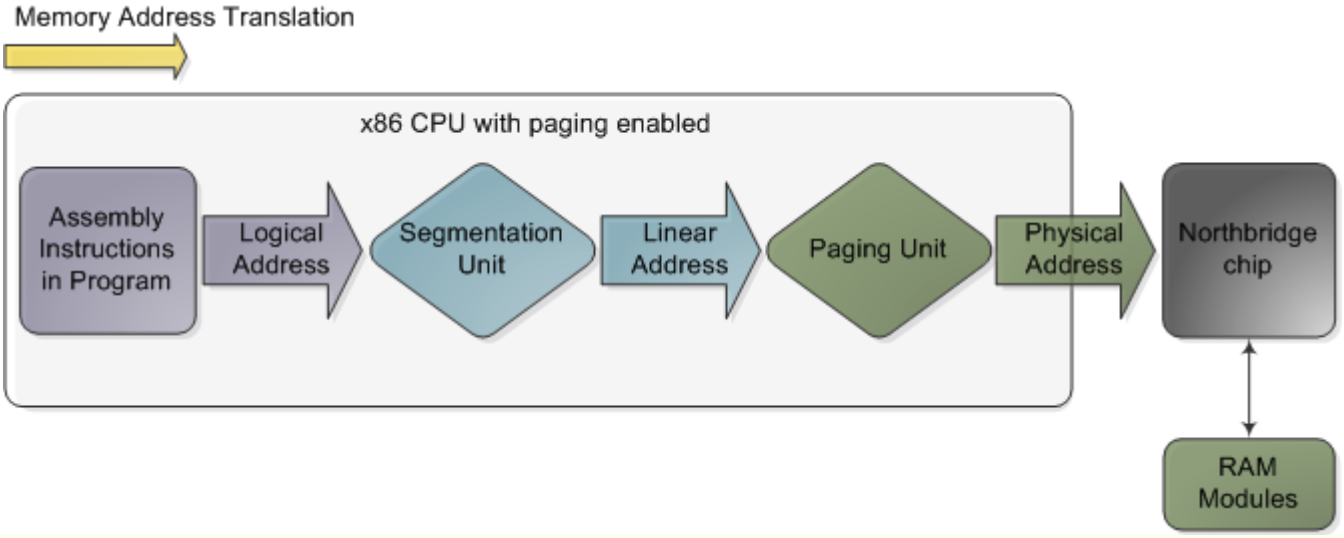
- 最好也最重要的资料是实际的内核代码，Linux 或 BSD 的都成。
- Intel 出版的杰出的软件开发人员手册，你可以免费下载到。
- 《理解 Linux 内核》是本好书，其中讨论了大量的 Linux 内核代码。这书也许有点过时有点枯燥，但我还是将它推荐给那些想要与内核心意相通的人们。《Linux 设备驱动程序》读起来会有趣得多，讲的也不错，但是涉及的内容有些局限性。最后，网友 Patrick Moroney 推荐 Robert Love 所写的《Linux 内核开发》，我曾听过一些对此书的正面评价，所以还是值得列出来的。
- 对于 Windows，目前最好的参考书是《Windows Internals》，作者是 David Solomon 和 Mark Russinovich，后者是 Sysinternals 的知名专家。这是本特棒的书，写的很好而且讲解全面。主要的缺点是缺少源代码的支持。

第四章 内存地址转换与分段

（Memory Translation and Segmentation）

本文是 Intel 兼容计算机（x86）的内存与保护系列文章的第一篇，延续了启动引导系列文章的主题，进一步分析操作系统内核的工作流程。与以前一样，我将引用 Linux 内核的源代码，但对 Windows 只给出示例（抱歉，我忽略了 BSD，Mac 等系统，但大部分的讨论对它们一样适用）。文中如果有错误，请不吝赐教。

在支持 Intel 的主板芯片组上，CPU 对内存的访问是通过连接着 CPU 和北桥芯片的前端总线来完成的。在前端总线上传输的内存地址都是物理内存地址，编号从 0 开始一直到可用物理内存的最高端。这些数字被北桥映射到实际的内存条上。物理地址是明确的、最终用在总线上的编号，不必转换，不必分页，也没有特权级检查。然而，在 CPU 内部，程序所使用的是逻辑内存地址，它必须被转换成物理地址后，才能用于实际内存访问。从概念上讲，地址转换的过程如下图所示：



x86 CPU 开启分页功能后的内存地址转换过程

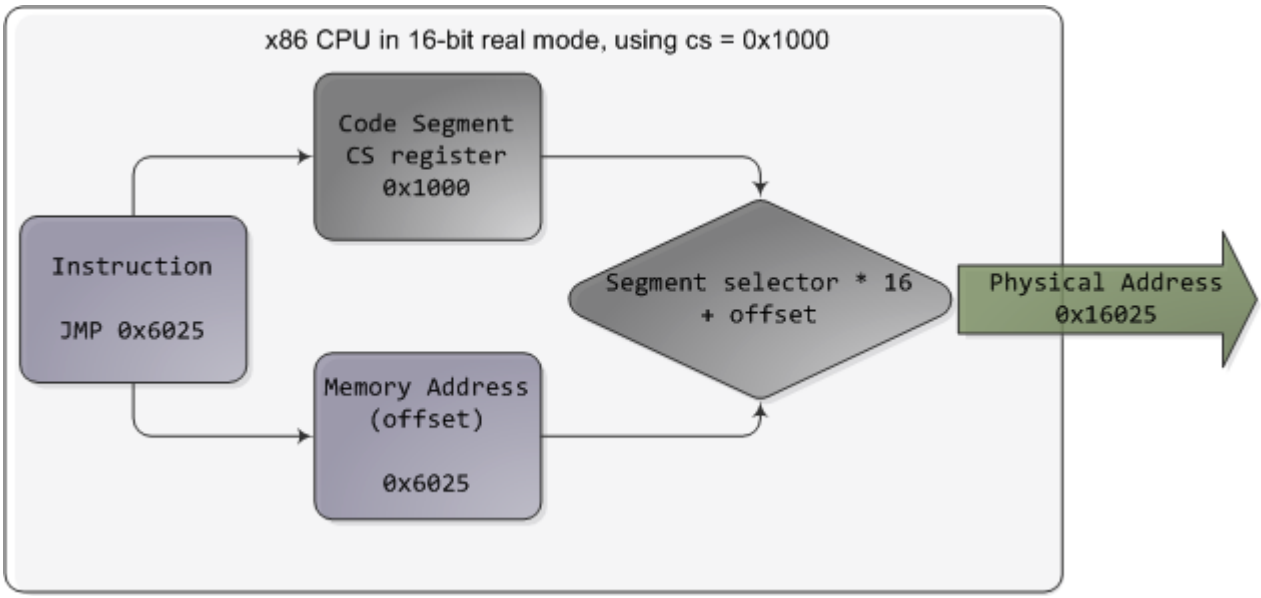
此图并未指出详实的转换方式，它仅仅描述了在 CPU 的分页功能开启的情况下内存地址的转换过程。如果 CPU 关闭了分页功能，或运行于 16 位实模式，那么从分段单元（segmentation unit）输出的就是最终的物理地址了。当 CPU 要执行一条引用了内存地址的指令时，转换过程就开始了。第一步是把逻辑地址转换成线性地址。但是，为什么不跳过这一步，而让软件直接使用线性地址（或物理地址呢？）其理由与："人类为何要长有阑尾？它的主要作用仅仅是被感染发炎而已"大致相同。这是进化过程中产生的奇特构造。要真正理解 x86 分段功能的设计，我们就必须回溯到 1978 年。

最初的 8086 处理器的寄存器是 16 位的，其指令集大多使用 8 位或 16 位的操作数。这使得代码可以控制 216 个字节（或 64KB）的内存。然而 Intel 的工程师们想要让 CPU 可以使用更多的内存，而又不用扩展寄存器和指令的位宽。于是他们引入了段寄存器（segment register），用来告诉 CPU 一条程序指令将操作*哪一个* 64K 的内存区块。一个合理的解决方案是：你先加载段寄存器，相当于说"这儿！我打算操作开始于 X 处的内存区块"；之后，再用 16 位的内存地址来表示相对于那个内存区块（或段）的偏移量。总共有 4 个段寄存器：一个用于栈（ss），

一个用于程序代码（cs），两个用于数据（ds，es）。在那个年代，大部分程序的栈、代码、数据都可以塞进对应的段中，每段 64KB 长，所以分段功能经常是透明的。

现今，分段功能依然存在，一直被 x86 处理器所使用着。每一条会访问内存的指令都隐式的使用了段寄存器。比如，一条跳转指令会用到代码段寄存器（cs），一条压栈指令（stack push instruction）会使用到堆栈段寄存器（ss）。在大部分情况下你可以使用指令明确的改写段寄存器的值。段寄存器存储了一个 16 位的段选择符（segment selector）；它们可以经由机器指令（比如 MOV）被直接加载。唯一的例外是代码段寄存器（cs），它只能被影响程序执行顺序的指令所改变，比如 CALL 或 JMP 指令。虽然分段功能一直是开启的，但其在实模式与保护模式下的运作方式并不相同的。

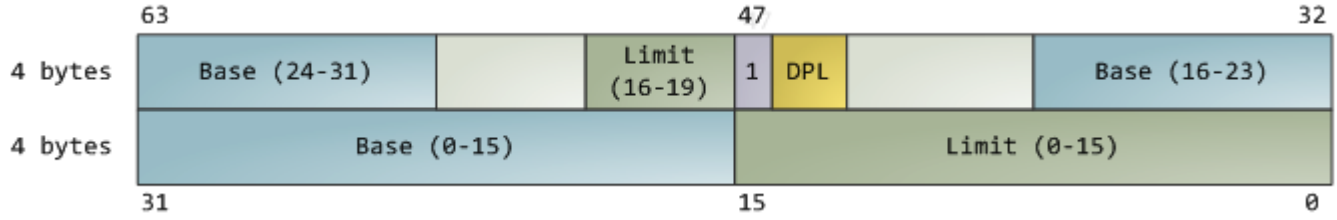
在实模式下，比如在引导启动的初期，段选择符是一个 16 位的数值，指示出一个段的开始处的物理内存地址。这个数值必须被以某种方式放大，否则它也会受限于 64K 当中，分段就没有意义了。比如，CPU 可能会把这个段选择符当作物理内存地址的高 16 位（只需将之左移 16 位，也就是乘以 2^16）。这个简单的规则使得：可以按 64K 的段为单位，一块块的将 4GB 的内存都寻址到。遗憾的是，Intel 做了一个很诡异的设计，让段选择符仅仅乘以 24（或 16），即左移 4bit（总计 20 bit Addr =1MB Space），从而一举将寻址范围限制在了 1MB，还引入了过度复杂的转换过程。下述图例显示了一条跳转指令，cs 的值是 0x1000：



实模式分段功能

实模式的段地址以 16 个字节为步长，从 0 开始编号一直到 0xFFFF0（即 1MB）。你可以将一个从 0 到 0xFFFF 的 16 位偏移量（逻辑地址）加在段地址上。在这个规则下，对于同一个内存地址，会有多个段地址/偏移量的组合与之对应，而且物理地址可以超过 1MB 的边界，只要你的段地址足够高（参见臭名昭著的 A20 线）。同样的，在实模式的 C 语言代码中，一个远指针（far pointer）既包含了段选择符又包含了逻辑地址，用于寻址 1MB 的内存范围。真够"远"的啊。随着程序变得越来越大，超出了 64K 的段，分段功能以及它古怪的处理方式，使得 x86 平台的软件开发变得非常复杂。这种设定可能听起来有些诡异，但它却把当时的程序员推进了令人崩溃的深渊。

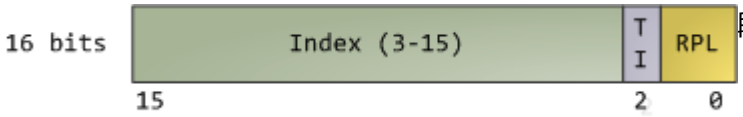
在 32 位保护模式下，段选择符不再是一个单纯的数值，取而代之的是一个索引编号，用于引用段描述符表中的表项。这个表为一个简单的数组，元素长度为 8 字节，每个元素描述一个段。看起来如下：



段描述符

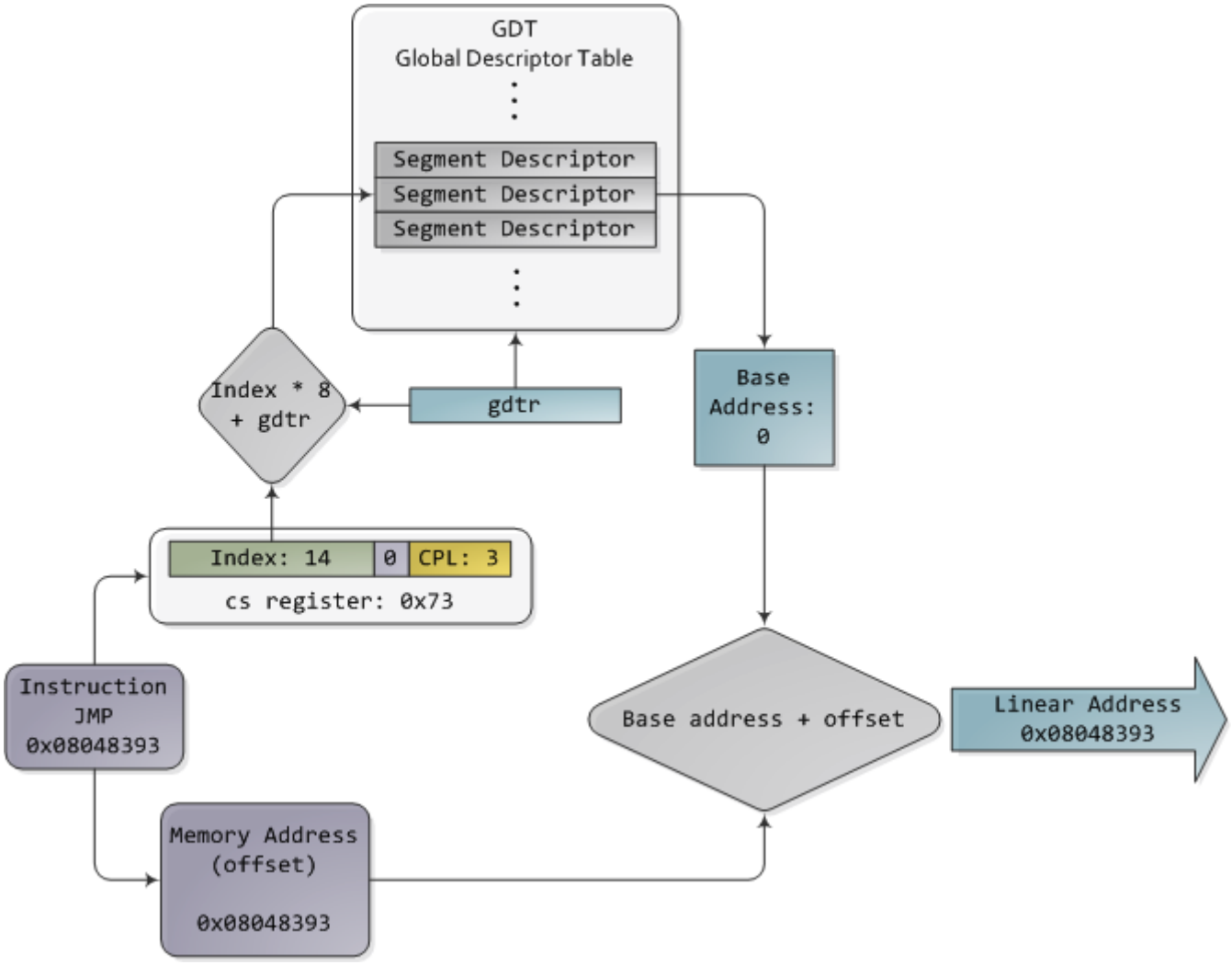
有三种类型的段 :代码 ,数据 ,系统。为了简洁明了 ,只有描述符的共有特征被绘制出来。基地址(base address ）是一个 32 位的线性地址，指向段的开始；段界限（ limit ）指出这个段有多大。将基地址加到逻辑地址上就形成了线性地址。DPL 是描述符的特权级（ privilege level ），其值从 0（最高特权，内核模式）到 3（最低特权，用户模式），用于控制对段的访问。

这些段描述符被保存在两个表中：全局描述符表（ GDT ）和局部描述符表（ LDT ）。电脑中的每一个 CPU（或一个处理核心)都含有一个叫做 gdtr 的寄存器 ,用于保存 GDT 的首个字节所在的线性内存地址。为了选出一个段，你必须向段寄存器加载符合以下格式的段选择符：



段选择符

对 GDT，TI 位为 0；对 LDT，TI 位为 1；index 指出想要表中哪一个段描述符（译注：原文是段选择符，应该是笔误）。对于 RPL，请求特权级（ Requested Privilege Level ），以后我们还会详细讨论。现在，需要好好想想了。当 CPU 运行于 32 位模式时，不管怎样，寄存器和指令都可以寻址整个线性地址空间，所以根本就不需要再去使用基地址或其他什么鬼东西。那为什么不干脆将基地址设成 0，好让逻辑地址与线性地址一致呢？Intel 的文档将之称为"扁平模型"（ flat model ），而且在现代的 x86 系统内核中就是这么做的（特别指出，它们使用的是基本扁平模型）。基本扁平模型（ basic flat model ）等价于在转换地址时关闭了分段功能。如此一来多么美好啊。就让我们来看看 32 位保护模式下执行一个跳转指令的例子 其中的数值来自一个实际的 Linux 用户模式应用程序：



保护模式的分段

段描述符的内容一旦被访问，就会被 cache（缓存），所以在随后的访问中，就不再需要去实际读取 GDT 了，否则会有损性能。每个段寄存器都有一个隐藏部分用于缓存段选择符所对应的那个段描述符。如果你想了解更多细

节，包括关于 LDT 的更多信息，请参阅《Intel System Programming Guide》3A 卷的第三章。2A 和 2B 卷讲述了每一个 x86 指令，同时也指明了 x86 寻址时所使用的各种类型的操作数：16 位，16 位加段描述符（可被用于实现远指针），32 位，等等。

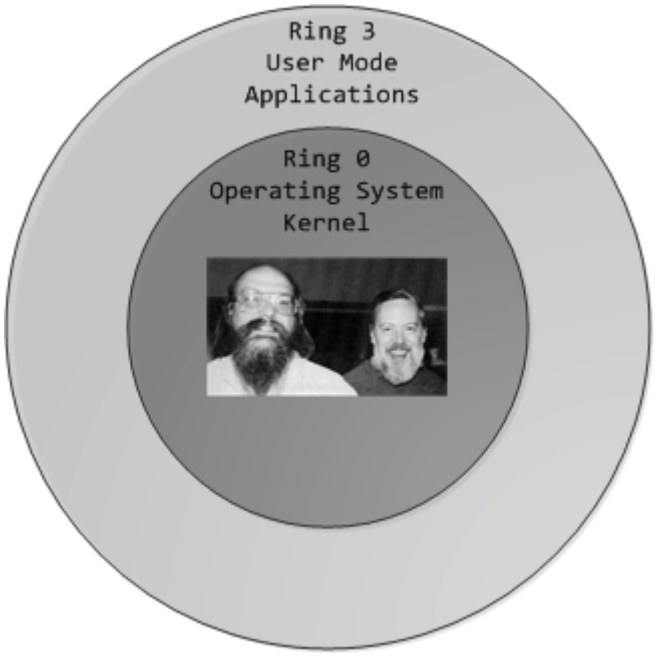
在 Linux 上，只有 3 个段描述符在引导启动过程被使用。他们使用 GDT_ENTRY 宏来定义并存储在 boot_gdt 数组中。其中两个段是扁平的，可对整个 32 位空间寻址：一个是代码段，加载到 cs 中，一个是数据段，加载到其他段寄存器中。第三个段是系统段，称为任务状态段（Task State Segment）。在完成引导启动以后，每一个 CPU 都拥有一份属于自己的 GDT。其中大部分内容是相同的，只有少数表项依赖于正在运行的进程。你可以从 segment.h 看到 Linux GDT 的布局以及其实际的样子。这里有 4 个主要的 GDT 表项：2 个是扁平的，用于内核模式的代码和数据，另两个用于用户模式。在看这个 Linux GDT 时，请留意那些用于确保数据与 CPU 缓存线对齐的填充字节——目的是克服冯·诺依曼瓶颈。最后要说说，那个经典的 Unix 错误信息"Segmentation fault"（分段错误）并不是由 x86 风格的段所引起的，而是由于分页单元检测到了非法的内存地址。唉呀，下次再讨论这个话题吧。

Intel 巧妙的绕过了他们原先设计的那个拼拼凑凑的分段方法，而是提供了一种富于弹性的方式来让我们选择是使用段还是使用扁平模型。由于很容易将逻辑地址与线性地址合二为一，于是这成为了标准，比如现在在 64 位模式中就强制使用扁平的线性地址空间了。但是即使是在扁平模型中，段对于 x86 的保护机制也十分重要。保护机制用于抵御用户模式进程对系统内核的非法内存访问，或各个进程之间的非法内存访问，否则系统将会进入一个狗咬狗的世界！在下一篇文章中，我们将窥视保护级别以及如何用段来实现这些保护功能。

第五章 CPU 的运行环、特权级与保护

（CPU Rings, Privilege, and Protection）

可能你凭借直觉就知道应用程序的功能受到了 Intel x86 计算机的某种限制，有些特定的任务只有操作系统的代码才可以完成，但是你知道这到底是怎么一回事吗？在这篇文章里，我们会接触到 x86 的特权级（privilege level），看看操作系统和 CPU 是怎么一起合谋来限制用户模式的应用程序的。特权级总共有 4 个，编号从 0（最高特权）到 3（最低特权）。有 3 种主要的资源受到保护：内存，I/O 端口以及执行特殊机器指令的能力。在任一时刻，x86 CPU 都是在一个特定的特权级下运行的，从而决定了代码可以做什么，不可以做什么。这些特权级经常被描述为保护环（protection ring），最内的环对应于最高特权。即使是最新的 x86 内核也只用到其中的 2 个特权级：0 和 3。



x86 的保护环

在诸多机器指令中，只有大约 15 条指令被 CPU 限制只能在 ring 0 执行（其余那么多指令的操作数都受到一定的限制）。这些指令如果被用户模式的程序所使用，就会颠覆保护机制或引起混乱，所以它们被保留给内核使用。

如果企图在 ring 0 以外运行这些指令，就会导致一个一般保护错（general-protection exception），就像一个程序使用了非法的内存地址一样。类似的，对内存和 I/O 端口的访问也受特权级的限制。但是，在我们分析保护机制之前，先让我们看看 CPU 是怎么记录当前特权级的吧，这与前篇文章中提到的段选择符（segment selector）有关。如下所示：

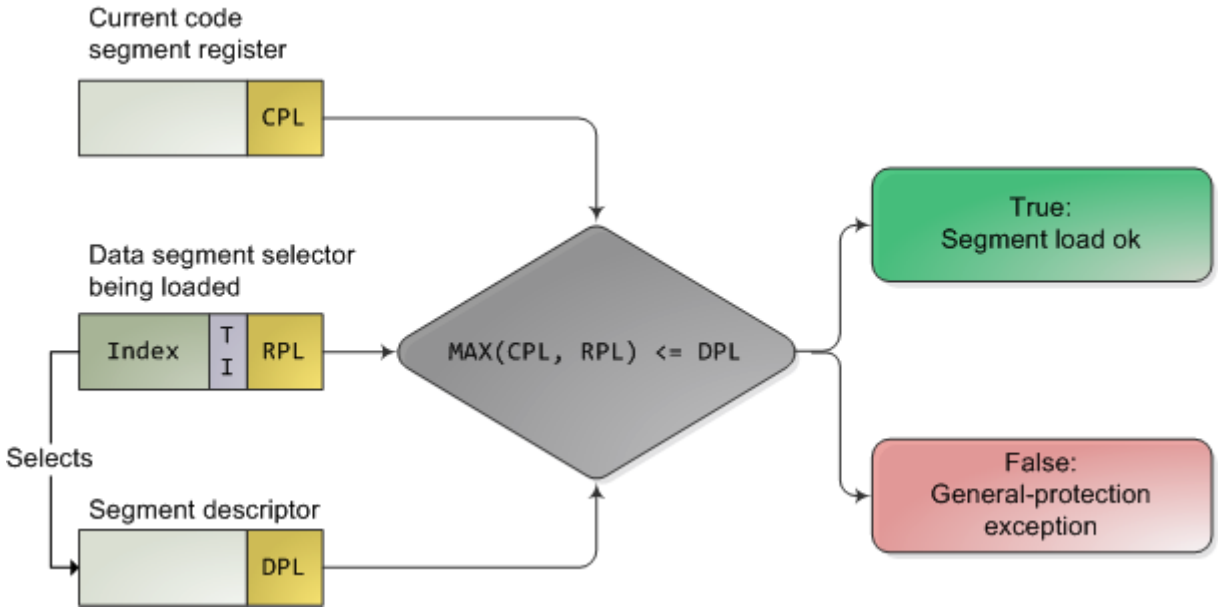
数据段和代码段的段选择符

数据段选择符的整个内容可由程序直接加载到各个段寄存器当中，比如 ss（堆栈段寄存器）和 ds（数据段寄存器）。这些内容里包含了请求特权级（Requested Privilege Level，简称 RPL）字段，其含义过会儿再说。然而，代码段寄存器（cs）就比较特别了。首先，它的内容不能由装载指令（如 MOV）直接设置，而只能被那些会改变程序执行顺序的指令（如 CALL）间接的设置。而且，不像那个可以被代码设置的 RPL 字段，cs 拥有一个由 CPU 自己维护的当前特权级字段（Current Privilege Level，简称 CPL），这点对我们来说非常重要。这个代码段寄存器中的 2 位宽的 CPL 字段的值总是等于 CPU 的当前特权级。Intel 的文档并未明确指出此事实，而且有时在线文档也对此含糊其辞，但这的确是个硬性规定。在任何时候，不管 CPU 内部正在发生什么，只要看一眼 cs 中的 CPL，你就可以知道此刻的特权级了。

记住，CPU 特权级并不会对操作系统的用户造成什么影响，*不管*你是根用户，管理员，访客还是一般用户。所有的用户代码都在 ring 3 上执行，所有的内核代码都在 ring 0 上执行，跟是以哪个 OS 用户的身份执行无关。有时一些内核任务可以被放到用户模式中执行，比如 Windows Vista 上的用户模式驱动程序，但是它们只是替内核执行任务的特殊进程而已，而且往往可以被直接删除而不会引起严重后果。

由于限制了对内存和 I/O 端口的访问，用户模式代码在不调用系统内核的情况下，几乎*不能*与外部世界交互。它不能打开文件，发送网络数据包，向屏幕打印信息或分配内存。用户模式进程的执行被严格限制在一个由 ring 0 之神所设定的沙盘之中。这就是为什么从设计上就决定了：一个进程所泄漏的内存会在进程结束后被统统回收，之前打开的文件也会被自动关闭。所有的控制着内存或打开的文件等的数据结构全都不能被用户代码直接使用；一旦进程结束了，这个沙盘就会被内核拆毁。这就是为什么我们的服务器只要硬件和内核不出毛病，就可以连续正常运行 600 天，甚至一直运行下去。这也解释了为什么 Windows 95/98 那么容易死机：这并非因为微软差劲，而是因为系统中的一些重要数据结构，出于兼容的目的被设计成可以由用户直接访问了。这在当时可能是一个很好的折中，当然代价也很大。

CPU 会在两个关键点上保护内存：当一个段选择符被加载时，以及，当通过线形地址访问一个内存页时。因此，保护也反映在内存地址转换的过程之中，既包括分段又包括分页。当一个数据段选择符被加载时，就会发生下述的检测过程：



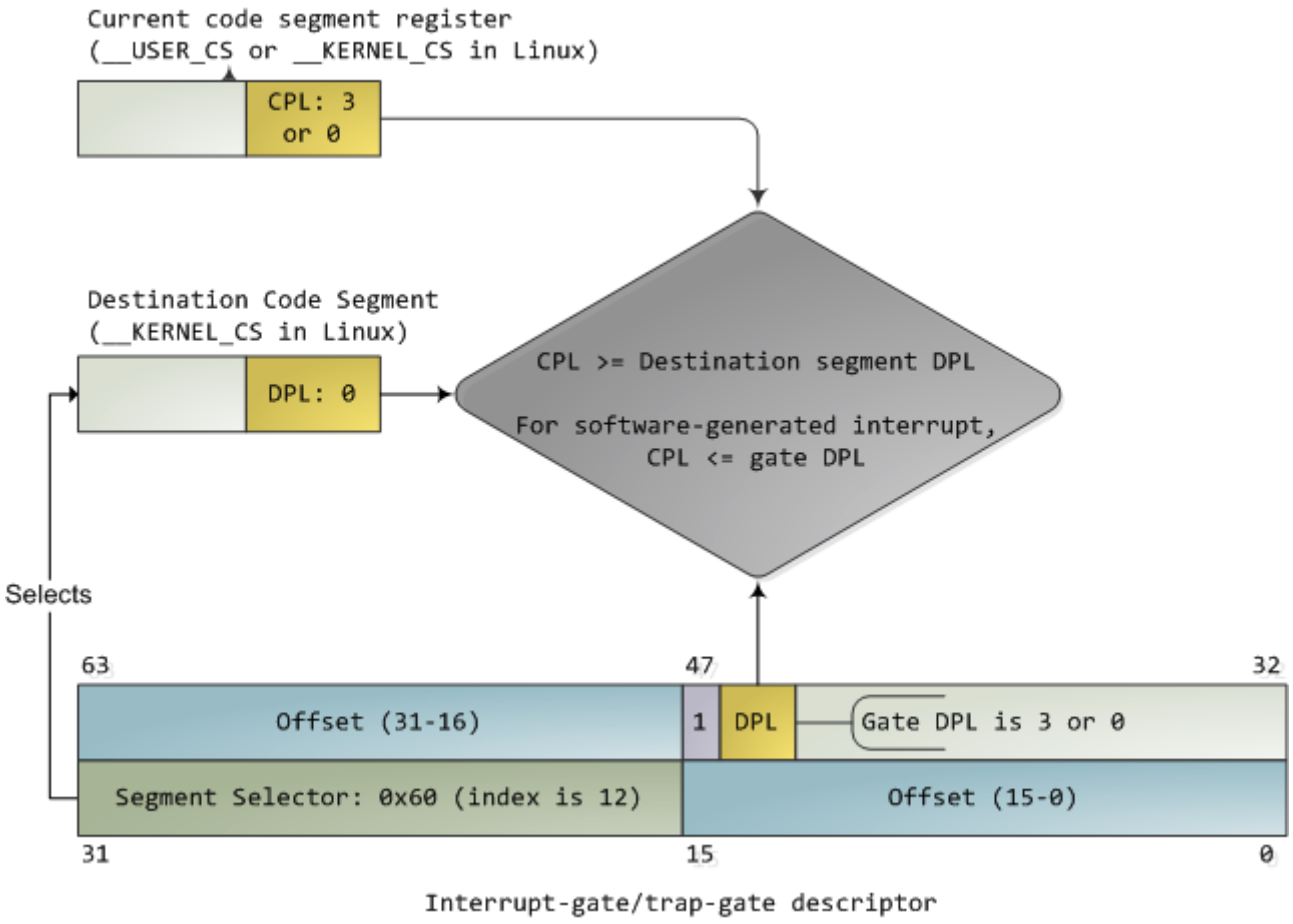
x86 的分段保护

因为越高的数值代表越低的特权，上图中的 MAX()用于挑出 CPL 和 RPL 中特权最低的一个，并与描述符特权级（descriptor privilege level，简称 DPL）比较。如果 DPL 的值大于等于它，那么这个访问就获得许可了。RPL 背后的设计思想是：允许内核代码加载特权较低的段。比如，你可以使用 RPL=3 的段描述符来确保给定的操作所使用的段可以在用户模式中访问。但堆栈段寄存器是个例外，它要求 CPL，RPL 和 DPL 这 3 个值必须完全一致，才可以被加载。

事实上，段保护功能几乎没什么用，因为现代的内核使用扁平的地址空间。在那里，用户模式的段可以访问整个线形地址空间。真正有用的内存保护发生在分页单元中，即从线形地址转化为物理地址的时候。一个内存页就是由一个页表项（page table entry）所描述的字节块。页表项包含两个与保护有关的字段：一个超级用户标志（supervisor flag），一个读写标志（read/write flag）。超级用户标志是内核所使用的重要的 x86 内存保护机制。当它开启时，内存页就不能被 ring 3 访问了。尽管读写标志对于实施特权控制并不像前者那么重要，但它依然十分有用。当一个进程被加载后，那些存储了二进制镜像（即代码）的内存页就被标记为只读了，从而可以捕获一些指针错误，比如程序企图通过此指针来写这些内存页。这个标志还被用于在调用 fork 创建 Unix 子进程时，实现写时拷贝功能（copy on write）。

最后，我们需要一种方式来让 CPU 切换它的特权级。如果 ring 3 的程序可以随意的将控制转移到（即跳转到）内核的任意位置，那么一个错误的跳转就会轻易的把操作系统毁掉了。但控制的转移是必须的。这项工作是通过门描述符（gate descriptor）和 sysenter 指令来完成的。一个门描述符就是一个系统类型的段描述符，分为了 4 个子类型：调用门描述符（call-gate descriptor），中断门描述符（interrupt-gate descriptor），陷阱门描述符（trap-gate descriptor）和任务门描述符（task-gate descriptor）。调用门提供了一个可以用于通常的 CALL 和 JMP 指令的内核入口点，但是由于调用门用得不多，我就忽略不提了。任务门也不怎么热门（在 Linux 上，它们只在处理内核或硬件问题引起的双重故障时才被用到）。

剩下两个有趣的：中断门和陷阱门，它们用来处理硬件中断（如键盘，计时器，磁盘）和异常（如缺页异常，0 除数异常）。我将不再区分中断和异常，在文中统一用“中断”一词表示。这些门描述符被存储在中断描述符表（Interrupt Descriptor Table，简称 IDT）当中。每一个中断都被赋予一个从 0 到 255 的编号，叫做中断向量。处理器把中断向量作为 IDT 表项的索引，用来指出当中断发生时使用哪一个门描述符来处理中断。中断门和陷阱门几乎是一样的。下图给出了它们的格式。以及当中断发生时实施特权检查的过程。我在其中填入了一些 Linux 内核的典型数值，以便让事情更加清晰具体。



伴随特权检查的中断描述符

门中的 DPL 和段选择符一起控制着访问，同时，段选择符结合偏移量（Offset）指出了中断处理代码的入口点。内核一般在门描述符中填入内核代码段的段选择符。一个中断永远不会将控制从高特权环转向低特权环。特权级必须要么保持不变（当内核自己被中断的时候），或被提升（当用户模式的代码被中断的时候）。无论哪一种情况，作为结果的 CPL 必须等于目的代码段的 DPL。如果 CPL 发生了改变，一个堆栈切换操作就会发生。如果中断是被程序中的指令所触发的（比如 INT n），还会增加一个额外的检查：门的 DPL 必须具有与 CPL 相同或更低的特权。这就防止了用户代码随意触发中断。如果这些检查失败，正如你所猜测的，会产生一个一般保护错（general-protection exception）。所有的 Linux 中断处理器都以 ring 0 特权退出。

在初始化阶段，Linux 内核首先在 setup_idt()中建立 IDT，并忽略全部中断。之后它使用 include/asm-x86/desc.h 的函数来填充普通的 IDT 表项（参见 arch/x86/kernel/traps_32.c）。在 Linux 代码中，名字中包含"system"字样的门描述符是可以从用户模式中访问的，而且其设置函数使用 DPL 3。"system gate"是 Intel 的陷阱门，也可以从用户模式访问。除此之外，术语名词都与本文对得上号。然而，硬件中断门并不是在这里设置的，而是由适当的驱动程序来完成。

有三个门可以被用户模式访问：中断向量 3 和 4 分别用于调试和检查数值运算溢出。剩下的是一个系统门，被设置为 SYSCALL_VECTOR。对于 x86 体系结构，它等于 0x80。它曾被作为一种*机制*，用于将进程的控制转移到内核，进行一个*系统调用*（system call），然后再跳转回来。在那个时代，我需要去申请"INT 0x80"这个没用的牌照

。从奔腾 Pro 开始，引入了 sysenter 指令，从此可以用这种更快捷的方式来启动系统调用了。它依赖于 CPU 上的特殊目的寄存器，这些寄存器存储着代码段、入口点及内核系统调用处理器所需的其他零散信息。在 sysenter 执行后，CPU 不再进行特权检查，而是直接进入 CPL 0，并将新值加载到与代码和堆栈有关的寄存器当中（cs，eip，ss 和 esp）。只有 ring 0 的代码 enable_sep_cpu()可以加载 sysenter 设置寄存器。

最后，当需要跳转回 ring 3 时，内核发出一个 iret 或 sysexit 指令，分别用于从中断和系统调用中返回，从而离开 ring 0 并恢复 CPL=3 的用户代码的执行。噢！Vim 提示我已经接近 1,900 字了，所以 I/O 端口的保护只能下次再谈了。这样我们就结束了 x86 的运行环与保护之旅。感谢您的耐心阅读。