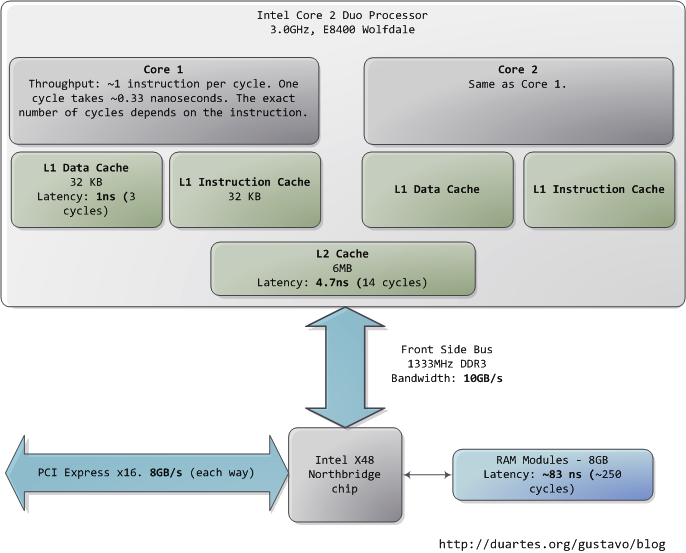
**[CPU与内存的那些事](http://www.cnblogs.com/xkfz007/archive/2012/10/08/2715163.html)**

**CPU的等待有多久?**

   本文以一个现代的、实际的个人电脑为对象，分析其中CPU（Intel Core 2 Duo 3.0GHz）以及各类子系统的运行速度——延迟和数据吞吐量。通过粗略的估算PC各个组件的相对运行速度，希望能给大家留下一个比较直观的印象。本文中的数据来自实际应用，而非理论最大值。时间的单位是纳秒（ns，十亿分之一秒），毫秒（ms，千分之一秒），和秒（s）。吞吐量的单位是兆字节（MB）和千兆字节（GB）。让我们先从CPU和内存开始，下图是北桥部分：



    第一个令人惊叹的事实是：CPU快得离谱。在Core 2 3.0GHz上，大部分简单指令的执行只需要一个时钟周期，也就是**1/3纳秒**。即使是真空中传播的光，在这段时间内也只能走10厘米（约4英寸）。把上述事实记在心中是有好处的。当你要对程序做优化的时候就会想到，执行指令的开销对于当今的CPU而言是多么的微不足道。

当CPU运转起来以后，它便会通过L1 cache和L2 cache对系统中的主存进行读写访问。cache使用的是静态存储器([SRAM](http://en.wikipedia.org/wiki/Static_RAM))。相对于系统主存中使用的动态存储器（DRAM），cache读写速度快得多、造价也高昂得多。cache一般被放置在CPU芯片的内部，加之使用昂贵高速的存储器，使其给CPU带来的延迟非常低。在指令层次上的优化（instruction-level optimization），其效果是与优化后代码的大小息息相关。由于使用了高速缓存技术（caching），那些能够整体放入L1/L2 cache中的代码，和那些在运行时需要不断调入/调出（marshall into/out of）cache的代码，在性能上会产生非常明显的差异。

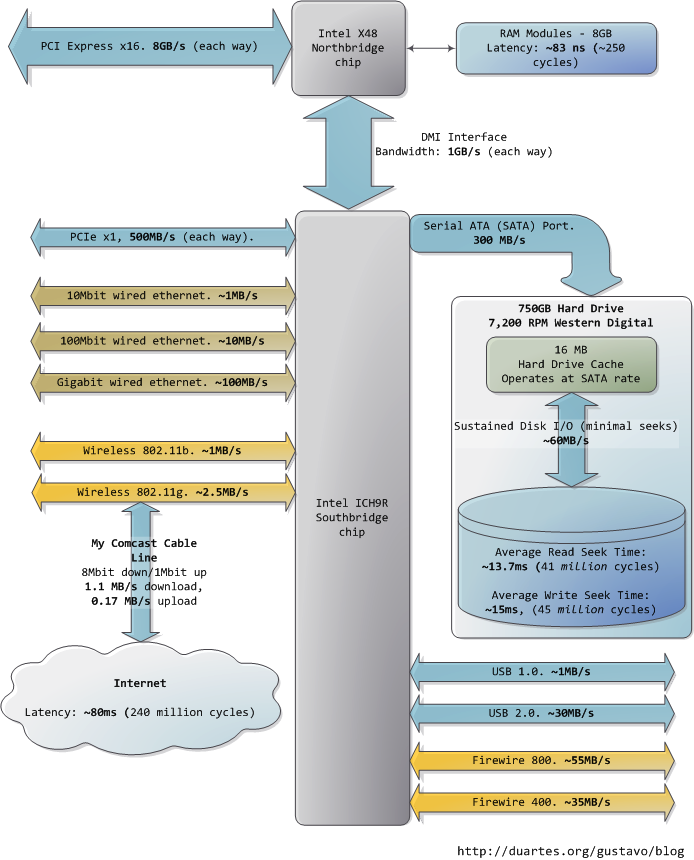
    正常情况下，当CPU操作一块内存区域时，其中的信息要么已经保存在L1/L2 cache，要么就需要将之从系统主存中调入cache，然后再处理。如果是后一种情况，我们就碰到了第一个瓶颈，一个大约250个时钟周期的延迟。在此期间如果CPU没有其他事情要做，则往往是处在停机状态的（stall）。为了给大家一个直观的印象，我们把CPU的一个时钟周期看作一秒。那么，从L1 cache读取信息就好像是拿起桌上的一张草稿纸（3秒）；从L2 cache读取信息则是从身边的书架上取出一本书（14秒）；而从主存中读取信息则相当于走到办公楼下去买个零食（4分钟）。

    主存操作的准确延迟是不固定的，与具体的应用以及其他许多因素有关。比如，它依赖于列选通延迟(CAS)以及内存条的型号，它还依赖于CPU指令预取的成功率。指令预取可以根据当前执行的代码来猜测主存中哪些部分即将被使用，从而提前将这些信息载入cache。

    看看L1/L2 cache的性能，再对比主存，就会发现：配置更大的cache或者编写能更好的利用cache的应用程序，会使系统的性能得到多么显著的提高。如果想进一步了解有关内存的诸多信息，读者可以参阅Ulrich Drepper所写的一篇经典文章《What Every Programmer Should Know About Memory》。

   人们通常把CPU与内存之间的瓶颈叫做[冯·诺依曼瓶颈](http://en.wikipedia.org/wiki/Von_Neumann_bottleneck)（von Neumann bottleneck）。当今系统的前端总线带宽约为10GB/s，看起来很令人满意。在这个速度下，你可以在1秒内从内存中读取8GB的信息，或者10纳秒内读取100字 节。遗憾的是，这个吞吐量只是理论最大值（图中其他数据为实际值），而且是根本不可能达到的，因为主存控制电路会引入延迟。在做内存访问时，会遇到很多零 散的等待周期。比如电平协议要求，在选通一行、选通一列、取到可靠的数据之前，需要有一定的信号稳定时间。由于主存中使用电容来存储信息，为了防止因自然 放电而导致的信息丢失，就需要周期性的刷新它所存储的内容，这也带来额外的等待时间。某些连续的内存访问方式可能会比较高效，但仍然具有延时。而那些随机 的内存访问则消耗更多时间。所以延迟是不可避免的。

图中下方的南桥连接了很多其他总线（如：PCI-E, USB）和外围设备：



   令人沮丧的是，南桥管理了一些反应相当迟钝的设备，比如硬盘。就算是缓慢的系统主存，和硬盘相比也可谓速度如飞了。继续拿办公室做比喻，等待硬盘寻道的时间相当于离开办公大楼并开始长达**一年零三个月**的环球旅行。这就解释了为何电脑的大部分工作都受制于磁盘I/O，以及为何数据库的性能在内存缓冲区被耗尽后会陡然下降。同时也解释了为何充足的RAM（用于缓冲）和高速的磁盘驱动器对系统的整体性能如此重要。

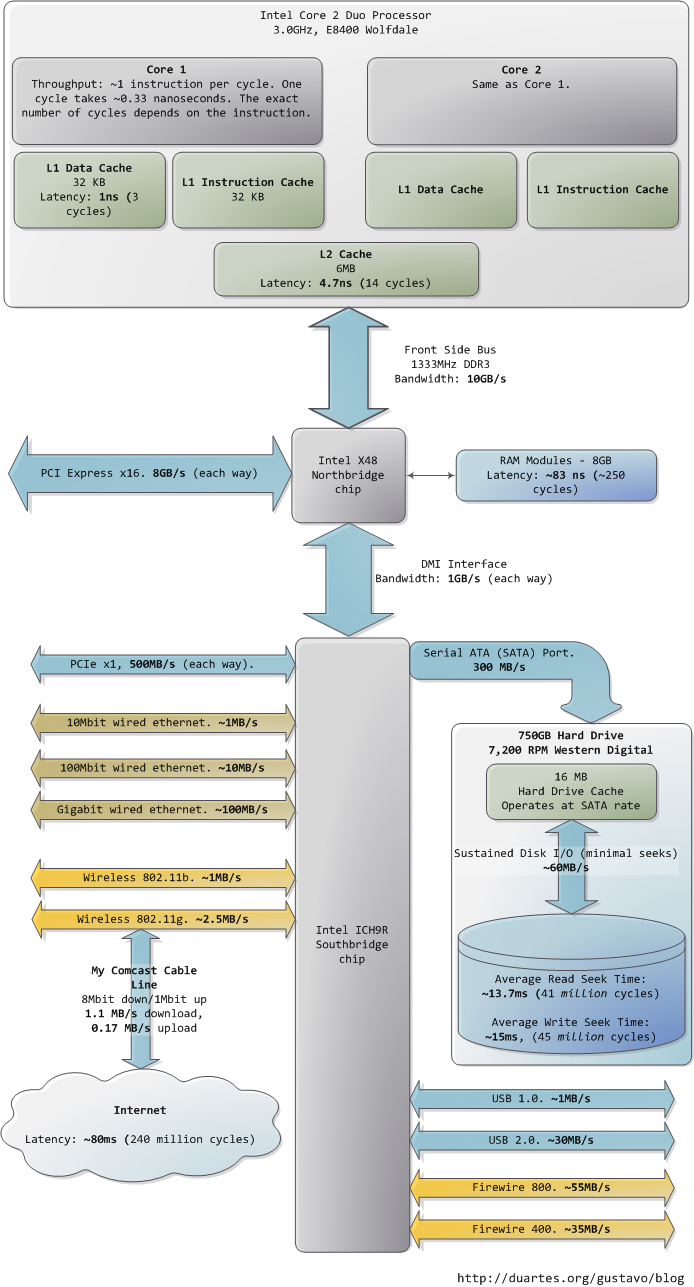
   虽然磁盘的"连续"存取速度确实可以在实际使用中达到，但这并非故事的全部。真正令人头疼的瓶颈在于寻道操作，也就是在磁盘表面移动读写磁头到正确的磁道上，然后再等待磁盘旋转到正确的位置上，以便读取指定扇区内的信息。RPM（每分钟绕转次数）用来指示磁盘的旋转速度：RPM越大，耽误在寻道上的时间就越少，所以越高的RPM意味着越快的磁盘。这里有一篇由两个Stanford的研究生写的很酷的文章，其中讲述了寻道时间对系统性能的影响：《Anatomy of a Large-Scale Hypertextual Web Search Engine》

    当磁盘驱动器读取一个大的、连续存储的文件时会达到更高的持续读取速度，因为省去了寻道的时间。文件系统的碎片整理器就是用来把文件信息重组在连续的数据块 中，通过尽可能减少寻道来提高数据吞吐量。然而，说到计算机实际使用时的感受，磁盘的连续存取速度就不那么重要了，反而应该关注驱动器在单位时间内可以完 成的寻道和随机I/O操作的次数。对此，固态硬盘可以成为一个很棒的选择。

    硬盘的cache也有助于改进性能。虽然16MB的cache只能覆盖整个磁盘容量的0.002%，可别看cache只有这么一点大，其效果十分明显。它可以把一组零散的写入操作合成一个，也就是[使磁盘能够控制写入操作的顺序，从而减少寻道的次数](http://lkml.indiana.edu/hypermail/linux/kernel/0110.0/0925.html)。同样的，为了提高效率，一系列读取操作也可以被重组，而且操作系统和驱动器固件(firmware)都会参与到这类优化中来。

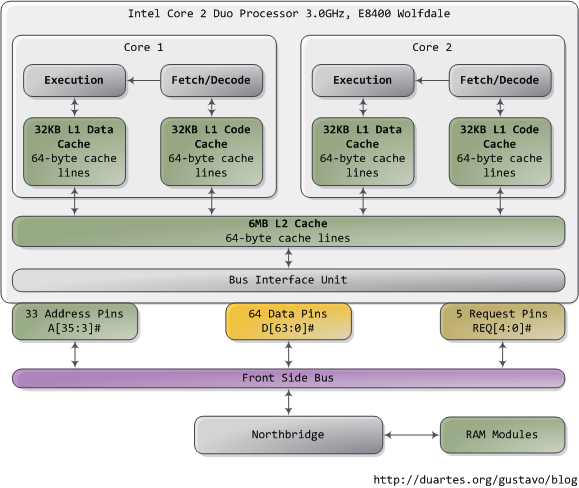
最后，图中还列出了网络和其他总线的实际数据吞吐量。火线(fireware)仅供参考，Intel X48芯片组并不直接支持火线。我们可以把Internet看作是计算机之间的总线。去访问那些速度很快的网站（比如google.com），延迟大约45毫秒，与硬盘驱动器带来的延迟相当。事实上，尽管硬盘比内存慢了5个数量级，它的速度与Internet是在同一数量级上的。目前，一般家用网络的带宽还是要落后于硬盘连续读取速度的，但"网络就是计算机"这句话可谓名符其实。如果将来Internet比硬盘还快了，那会是个什么景象呢？

我希望这些图片能对您有所帮助。当这些数字一起呈现在我面前时，真的很迷人，也让我看到了计算机技术发展到了哪一步。前文分开的两个图片只是为了叙述方便，我把包含南北桥的整张图片也贴出来，供您参考。



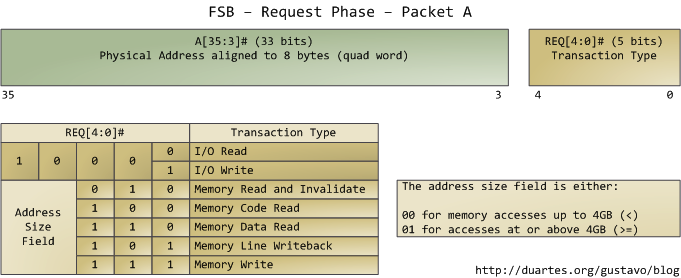
**CPU如何操作内存**

   在你试图理解一个复杂的系统时，如果能揭去表面的抽象并专注于最低级别的概念，往往会有不小的收获。在这个精神的指导下，让我们看看对于内存和I/O端口操作来说最简单、最基础的概念，即CPU与总线之间的接口。其中的细节是很多上层概念的基础，比如线程同步。当然了，既然我是个程序员，就暂且忽略那些只有电子工程师才会去关注的东西吧。下图是我们的老朋友，Core 2：

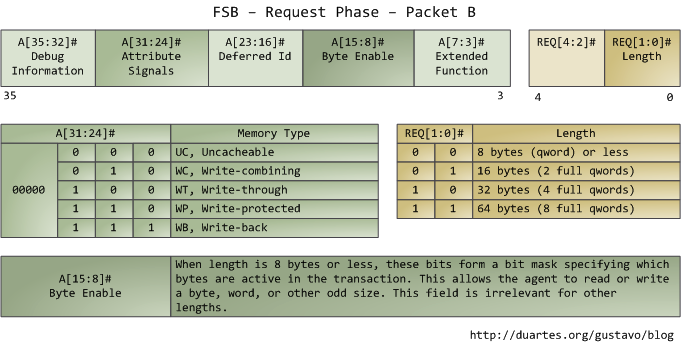


    Core 2 处理器有775个管脚，其中约半数仅仅用于供电而不参与数据传输。当你把这些管脚按照功能分类后，就会发现这个处理器的物理接口惊人的简单。本图展示了参与内存和I/O端口操作的重要管脚：地址线，数据线，请求线。这些操作均发生在前端总线的事务上下文结构(the context of a transaction)中。前端总线事务的执行包含五个阶段：仲裁，请求，侦听，响应，数据操作。在执行事务的过程中，前端总线上的各个部件扮演着不同的角色。这些部件称之为**agent**。通常，agent就是全部的处理器外加北桥。

本文只分析请求阶段。在此阶段中，发出请求的agent往往是一个处理器，它输出两个数据包。下图列出了第一个数据包中最为重要的位，这些数据位通过处理器的地址线和请求线输出：

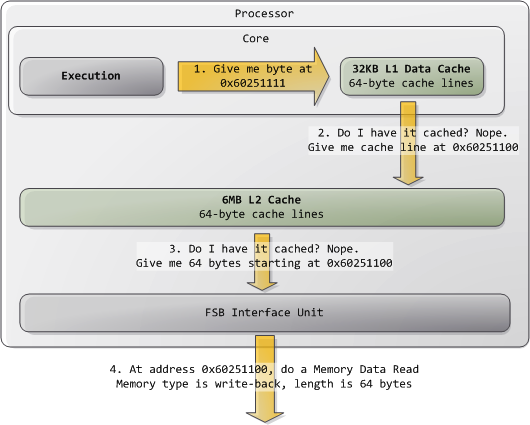


    地址线输出指定了事务发生的物理内存起始地址。我们有33条地址线，他们指定了数据包的第35至第3位，第2至第0位为0。因此，实际上这33条地址线构成了一个36位的、以8字节对齐的地址，正好覆盖64GB的物理内存。这种设定从奔腾Pro就开始了。请求线指定了事务的类型。当事务类型为I/O请求时，地址线指出的是I/O端口地址而不是内存地址。当第一个数据包被发送以后，同样由这组管脚，在下一个总线时钟周期发送第二个数据包：



    属性信号（attribute signal A[31:24]）很有趣，它反映了Intel处理器所支持的5种内存缓冲功能。把这些信息发布到前端总线后，发出请求的agent就可以让其他处理器知道如何根据当前事务处理他们自己的cache，以及让内存控制器（也就是北桥）知道该如何应对。一块指定内存区域的缓存类型由处理器通过查询页表（page table）来决定，页表由OS内核维护。

    典型的情况是，内核把全部内存都视为"回写"类型（write-back），从而获得最好的性能。在回写模式下，内存的最小访问单元为一个[缓存线](http://blog.csdn.net/drshenlei/archive/2009/06/17/4277959.aspx)（cache line），在Core 2中是64字节。当程序想读取内存中的一个字节时，处理器会从L1/L2 cache读取包含此字节的整条缓存线的内容。当程序做写入内存操作时，处理器只是修改cache中的对应缓存线，而不会更新主存中的信息。之后，当真的需要更新主存时，处理器会把那个被修改了的缓存线整体放到总线上，一次性写入内存。所以大部分的请求事务，其数据长度字段都是11（REQ[1:0]），对应64 字节。下图展示了当cache中没有对应数据时，内存读取访问的过程：



在Intel计算机上，有些物理内存范围被[映射为设备地址](http://blog.csdn.net/drshenlei/archive/2009/06/06/4246441.aspx)而不是实际的RAM存储器地址，比如硬盘和网卡。这使得驱动程序可以像读写内存那样，方便的与设备通信。内核会在页表中标记出这类内存映射区域为**不可缓存的**（uncacheable）。对不可缓存的内存区域的访问操作会被总线原封不动的按顺序执行，其操作与应用程序或驱动程序所发出的请求完全一致。因此，这时程序可以精确控制读写单个字节、字、或其它长度的信息。这都是通过设置第二个数据包中的字节使能掩码（byte enable mask A[15:8]）来完成的。

前面讨论的这些基本知识还包含很多关联的内容。比如：

1、  如果应用程序想要尽可能高的运行速度，就应该把会被一起访问的数据尽量组织在同一条缓存线中。一旦这条缓存线被载入，之后的读取操作就会[加快很多](http://blog.csdn.net/drshenlei/archive/2009/06/05/4240703.aspx)，不再需要额外的内存访问了。

2、  对于回写式内存访问，作用于一条缓存线的任何内存操作都一定是原子的（atomic）。这种能力是由处理器的L1 cache提供的，所有数据被同时读写，中途不会被其他处理器或线程打断。特别的，32位和64位的内存操作，只要不跨越缓存线的边界，就都是原子操作。

3、  前端总线是被所有的agent所共享的。这些agent在开启一个事务之前，必须先进行总线使用权的仲裁。而且，每一个agent都需要侦听总线上所有的事务，以便维持cache的一致性。因此，随着部署更多的、多核的处理器到Intel计算机，总线竞争问题会变得越来越严重。为解决这个问题，Core i7将处理器直接连接于内存，并以点对点的方式通信，取代之前的广播方式，从而减少总线竞争。

本文讲述的都是有关物理内存请求的重要内容。当涉及到内存锁定、多线程、缓存一致性的问题时，总线这个角色又将浮出水面。当我第一次看到前端总线数据包的描 述时，会有种恍然大悟的感觉，所以我希望您也能从本文中获益。下一篇文章，我们将从底层爬回到上层去，研究一个抽象概念：虚拟内存。

 参考: <http://blog.csdn.net/drshenlei/article/details/4243733>

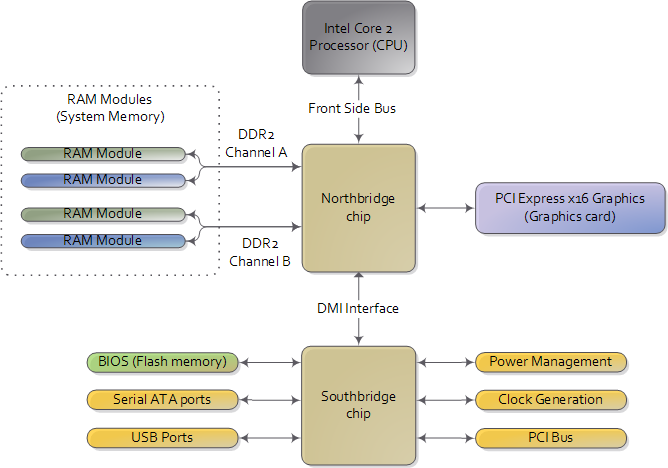
**主板芯片组与内存映射**

原文标题：Motherboard Chipsets and the Memory Map

原文地址：http://duartes.org/gustavo/blog/

   我打算写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：



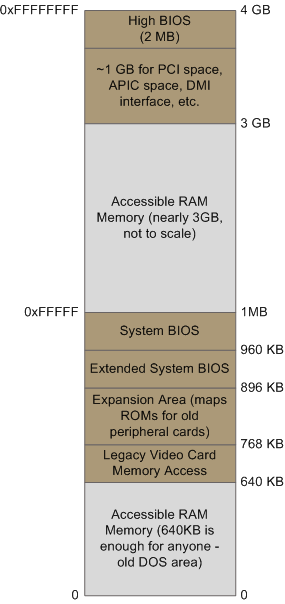
现代主板的示意图，北桥和南桥构成了芯片组。

     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](http://en.wikipedia.org/wiki/Image:Intel_80486DX2_bottom.jpg)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233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（233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](http://en.wikipedia.org/wiki/Memory-mapped_IO)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](http://support.microsoft.com/kb/929605)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](http://en.wikipedia.org/wiki/Physical_address_extension)，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](http://en.wikipedia.org/wiki/Real_mode)，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：

λ         《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。

λ         《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。

λ         《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）。

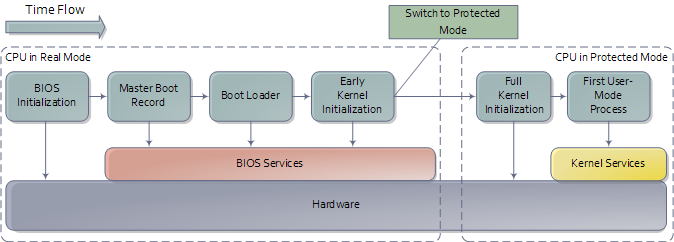
λ         Pádraig Brady建议我链接到Ulrich Drepper的一篇[关于内存的优秀文章](http://people.redhat.com/drepper/cpumemory.pdf)。确实是个好东西。我本打算把这个链接放到讨论存储器的文章中的，但此处列出的越多越好啦。

参考: <http://blog.csdn.net/drshenlei/article/details/4246441>

**计算机的引导过程**

原文标题：How Computers Boot Up

   前一篇文章介绍了Intel计算机的[主板与内存映射](http://blog.csdn.net/drshenlei/archive/2009/06/06/4246441.aspx)，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：



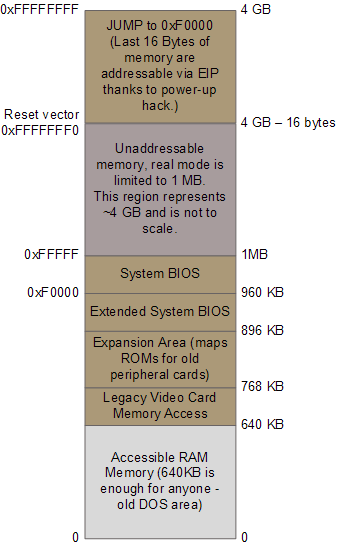
引导过程概要

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](http://en.wikipedia.org/wiki/Real_mode)，[分页](http://en.wikipedia.org/wiki/Paging)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](http://en.wikipedia.org/wiki/Reset_vector)(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](http://blog.csdn.net/drshenlei/archive/2009/06/06/4246441.aspx)。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：

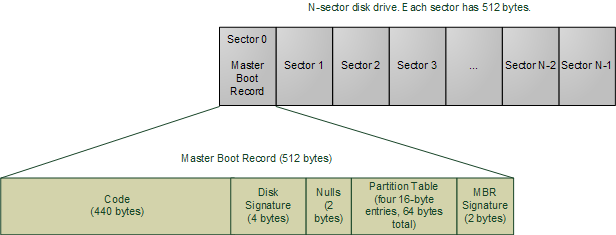


引导时的重要内存区域

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](http://en.wikipedia.org/wiki/Power_on_self_test)（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](http://en.wikipedia.org/wiki/ACPI)（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](http://en.wikipedia.org/wiki/Disk_sector)（0扇区），内含512个字节。这些数据叫做[主引导记录](http://en.wikipedia.org/wiki/Master_boot_record)（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：

1、  MBR本身包含有第一阶段的引导装载程序。GRUB称之为阶段一。

2、  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。

3、  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。

4、  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](http://en.wikipedia.org/wiki/Unreal_mode)"。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。

参考:

<http://blog.csdn.net/drshenlei/article/details/4250306>

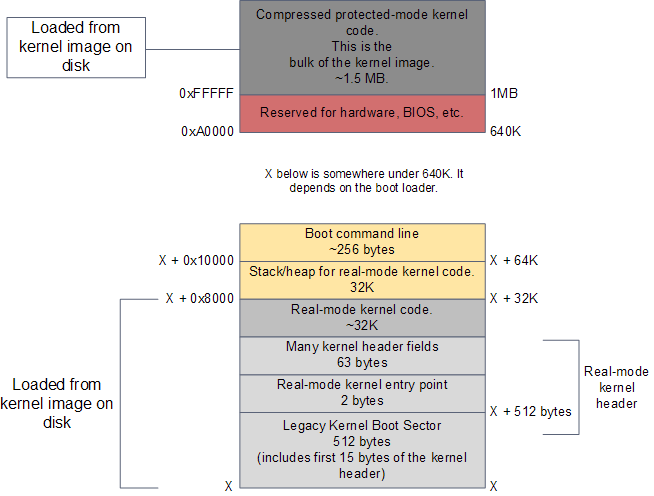
**内核引导过程**

原文标题：The Kernel Boot Process

原文地址：http://duartes.org/gustavo/blog/

   上一篇文章解释了计算机的[引导过程](http://blog.csdn.net/drshenlei/archive/2009/06/07/4250306.aspx)，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。

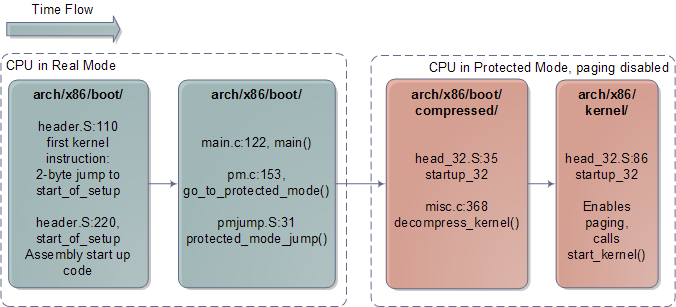
   当Intel x86的引导程序运行到此刻时，处理器处于实模式（可以寻址1MB的内存），（针对现代的Linux系统）RAM的内容大致如下：



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](http://en.wikipedia.org/wiki/Interrupt_vector_table)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](http://en.wikipedia.org/wiki/Global_descriptor_table)的地址。所以go\_to\_protected\_mode()调用了setup\_idt() 和 [setup\_gdt()](http://lxr.linux.no/linux+v2.6.25.6/arch/x86/boot/pm.c)，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](http://en.wikipedia.org/wiki/Paging)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](http://lxr.linux.no/linux+v2.6.25.6/arch/x86/kernel/head_32.S)。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：

1、  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。）

2、  为内存建立最终的全局描述符表。

3、  建立页表以便可以开启分页功能。

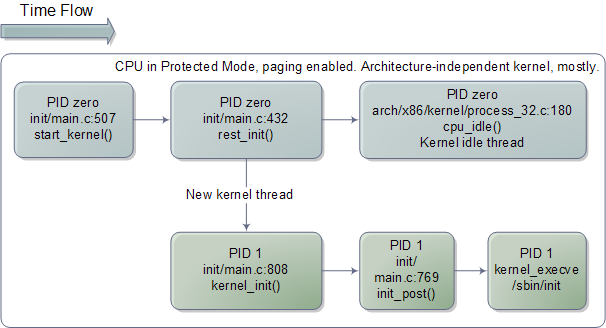
4、  使能分页功能。

5、  初始化栈空间。

6、  创建最终的中断描述符表。

7、  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：



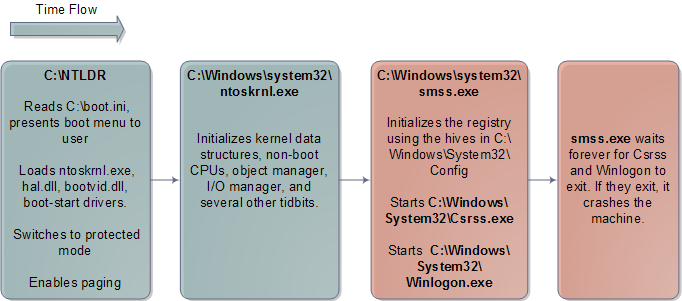
与体系结构无关的Linux内核初始化过程

    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：

λ         最好也最重要的资料是实际的内核代码，Linux或BSD的都成。

λ         Intel出版的杰出的[软件开发人员手册](http://www.intel.com/products/processor/manuals/index.htm)，你可以免费下载到。

λ         《[理解](http://www.amazon.com/exec/obidos/ASIN/0596005652/gustduar-20)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。

λ         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。

参考:

<http://blog.csdn.net/drshenlei/article/details/4253179>

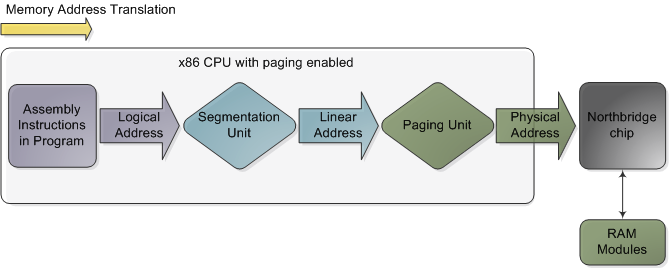
**内存地址转换与分段**

原文标题：Memory Translation and Segmentation

原文地址：http://duartes.org/gustavo/blog/

   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](http://blog.csdn.net/drshenlei/archive/2009/06/08/4253179.aspx)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](http://blog.csdn.net/drshenlei/archive/2009/06/06/4246441.aspx)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：



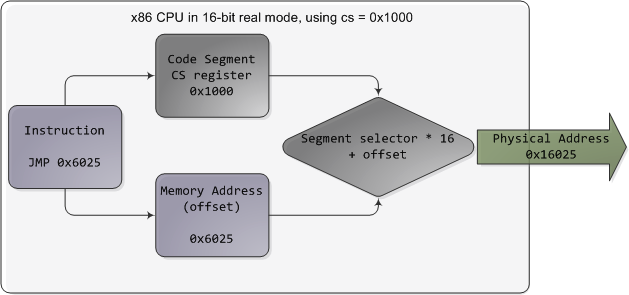
x86 CPU开启分页功能后的内存地址转换过程

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。

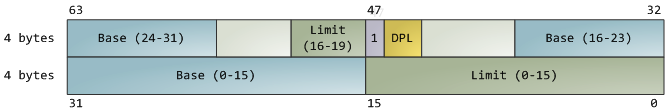
    在实模式下，比如在[引导启动的初期](http://blog.csdn.net/drshenlei/archive/2009/06/07/4250306.aspx)，段选择符是一个16位的数值，指示出一个段的开始处的物理内存地址。这个数值必须被以某种方式放大，否则它也会受限于64K当中，分段就没有意义了。比如，CPU可能会把这个段选择符当作物理内存地址的高16位（只需将之左移16位，也就是乘以216）。这个简单的规则使得：可以按64K的段为单位，一块块的将4GB的内存都寻址到。遗憾的是，Intel做了一个很诡异的设计，让段选择符仅仅乘以24（或16），一举将寻址范围限制在了1MB，还引入了过度复杂的转换过程。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](http://mirror.href.com/thestarman/asm/debug/Segments.html)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](http://en.wikipedia.org/wiki/C_memory_model)（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

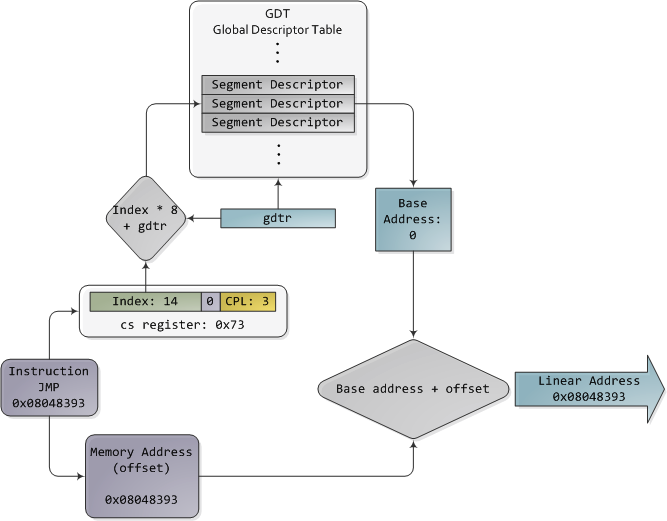
有三种类型的段：代码，数据，系统。为了简洁明了，只有描述符的共有特征被绘制出来。**基地址**（base address）是一个32位的线性地址，指向段的开始；**段界限**（limit）指出这个段有多大。将基地址加到逻辑地址上就形成了线性地址。DPL是描述符的特权级（privilege level），其值从0（最高特权，内核模式）到3（最低特权，用户模式），用于控制对段的访问。

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：

https://images.cnblogs.com/cnblogs_com/xkfz007/201210/201210081409201263.png

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](http://lxr.linux.no/linux+v2.6.25.6/arch/x86/kernel/cpu/common.c)。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](http://en.wikipedia.org/wiki/Von_Neumann_bottleneck)。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。

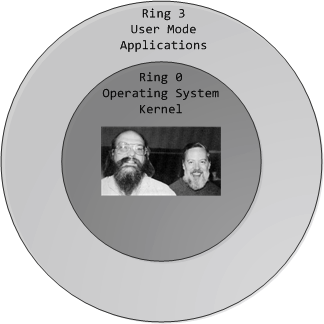
参考: <http://blog.csdn.net/drshenlei/article/details/4261909>

**CPU的运行环, 特权级与保护**

原文标题：CPU Rings, Privilege, and Protection

原文地址：http://duartes.org/gustavo/blog/

   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](http://blog.csdn.net/drshenlei/archive/2009/06/11/4261909.aspx)（segment selector）有关。如下所示：



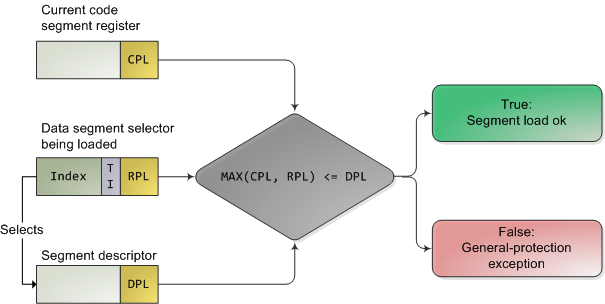
数据段和代码段的段选择符

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](http://blog.csdn.net/drshenlei/archive/2009/06/11/4261909.aspx)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：



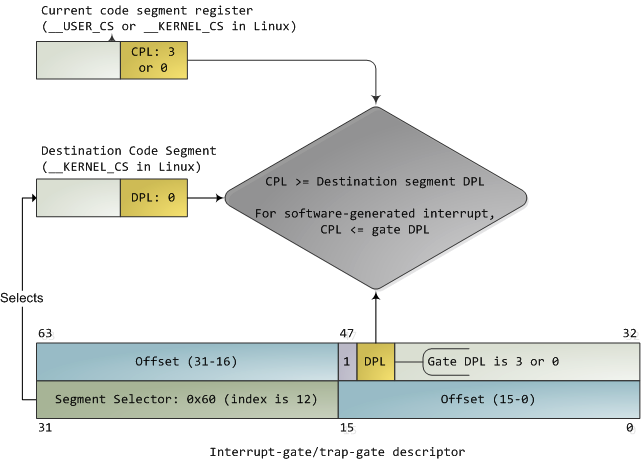
x86的分段保护

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 J。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。

参考:

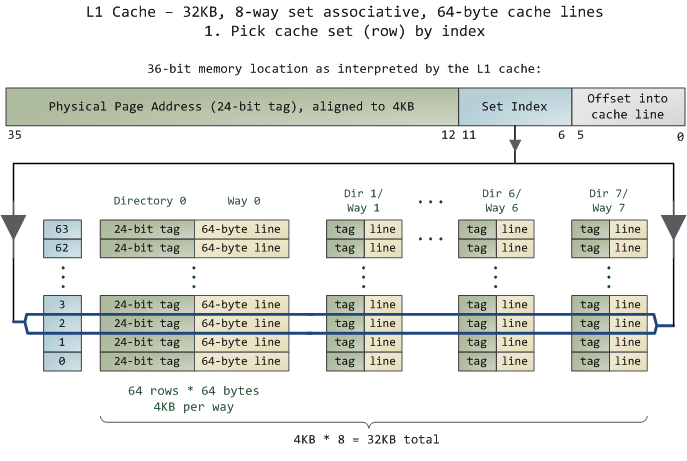
<http://blog.csdn.net/drshenlei/article/details/4265101>

**Cache: 一个隐藏并保存数据的场所**

原文标题：Cache: a place for concealment and safekeeping

原文地址：http://duartes.org/gustavo/blog/

   本文简要的展示了现代Intel处理器的CPU cache是如何组织的。有关cache的讨论往往缺乏具体的实例，使得一些简单的概念变得扑朔迷离。也许是我可爱的小脑瓜有点迟钝吧，但不管怎样，至少下面讲述了故事的前一半，即Core 2的 L1 cache是如何被访问的：



L1 cache – 32KB，8路组相联，64字节缓存线

1.       由索引拣选缓存组（行）

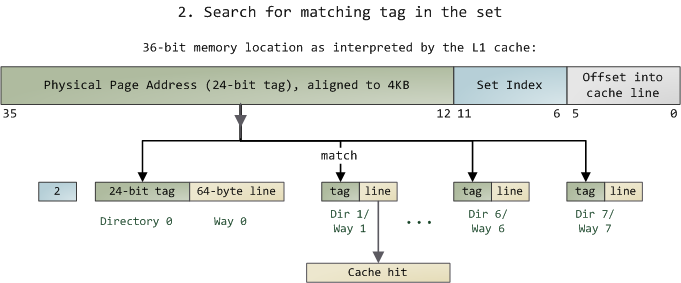
   在cache中的数据是以**缓存线**（line）为单位组织的，一条缓存线对应于内存中一个连续的字节块。这个cache使用了64字节的缓存线。这些线被保存在cache bank中，也叫**路**（way）。每一路都有一个专门的**目录**（directory）用来保存一些登记信息。你可以把每一路连同它的目录想象成电子表格中的一列，而表的一行构成了cache的一**组**（set）。列中的每一个单元（cell）都含有一条缓存线，由与之对应的目录单元跟踪管理。图中的cache有64 组、每组8路，因此有512个含有缓存线的单元，合计32KB的存储空间。

   在cache眼中，物理内存被分割成了许多4KB大小的物理内存页（page）。每一页都含有4KB / 64 bytes == 64条缓存线。在一个4KB的页中，第0到63字节是第一条缓存线，第64到127字节是第二条缓存线，以此类推。每一页都重复着这种划分，所以第0页第3条缓存线与第1页第3条缓存线是不同的。

   在**全相联缓存**（fully associative cache）中，内存中的任意一条缓存线都可以被存储到任意的缓存单元中。这种存储方式十分灵活，但也使得要访问它们时，检索缓存单元的工作变得复杂、昂贵。由于L1和L2 cache工作在很强的约束之下，包括功耗，芯片物理空间，存取速度等，所以在多数情况下，使用全相联缓存并不是一个很好的折中。

   取而代之的是图中的**组相联缓存**（set associative cache）。意思是，内存中一条给定的缓存线只能被保存在一个特定的组（或行）中。所以，*任意*物理内存页的第0条缓存线（页内第0到63字节）**必须**存储到第0组，第1条缓存线存储到第1组，以此类推。每一组有8个单元可用于存储它所关联的缓存线（译注：就是那些需要存储到这一组的缓存线），从而形成一个8路关联的组（8-way associative set）。当访问一个内存地址时，地址的第6到11位（译注：组索引）指出了在4KB内存页中缓存线的编号，从而决定了即将使用的缓存组。举例来说，物理地址0x800010a0的组索引是000010，所以此地址的内容一定是在第2组中缓存的。

   但是还有一个问题，就是要找出一组中*哪个*单元包含了想要的信息，如果有的话。这就到了缓存目录登场的时刻。每一个缓存线都被其对应的目录单元做了*标记*（tag）；这个标记就是一个简单的内存页编号，指出缓存线来自于哪一页。由于处理器可以寻址64GB的物理RAM，所以总共有64GB / 4KB == 224个内存页，需要24位来保存标记。前例中的物理地址0x800010a0对应的页号为524,289。下面是故事的后一半：



在组中搜索匹配标记

   由于我们只需要去查看某一组中的8路，所以查找匹配标记是非常迅速的；事实上，从电学角度讲，所有的标记是同时进行比对的，我用箭头来表示这一点。如果此时正好有一条具有匹配标签的有效缓存线，我们就获得一次缓存命中（cache hit）。否则，这个请求就会被转发的L2 cache，如果还没匹配上就再转发给主系统内存。通过应用各种调节尺寸和容量的技术，Intel给CPU配置了较大的L2 cache，但其基本的设计都是相同的。比如，你可以将原先的缓存增加8路而获得一个64KB的缓存；再将组数增加到4096，每路可以存储256KB。经过这两次修改，就得到了一个4MB的L2 cache。在此情况下，需要18位来保存标记，12位保存组索引；缓存所使用的物理内存页的大小与其一路的大小相等。（译注：有4096组，就需要lg(4096)==12位的组索引，缓存线依然是64字节，所以一路有4096\*64B==256KB字节；在L2 cache眼中，内存被分割为许多256KB的块，所以需要lg(64GB/256KB)==18位来保存标记。）

   如果有一组已经被放满了，那么在另一条缓存线被存储进来之前，已有的某一条则必须被腾空（evict）。为了避免这种情况，对运算速度要求较高的程序就要尝试仔细组织它的数据，使得内存访问均匀的分布在已有的缓存线上。举例来说，假设程序中有一个数组，元素的大小是512字节，其中一些对象在内存中相距4KB。这些对象的各个字段都落在同一缓存线上，并竞争同一缓存组。如果程序频繁的访问一个给定的字段（比如，通过[虚函数表](http://en.wikipedia.org/wiki/Vtable)vtable调用虚函数），那么这个组看起来就好像一直是被填满的，缓存开始变得毫无意义，因为缓存线一直在重复着腾空与重新载入的步骤。在我们的例子中，由于组数的限制，L1 cache仅能保存8个这类对象的虚函数表。这就是组相联策略的折中所付出的代价：即使在整体缓存的使用率并不高的情况下，由于组冲突，我们还是会遇到缓存缺失的情况。然而，鉴于计算机中各个存储层次的[相对速度](http://blog.csdn.net/drshenlei/archive/2009/06/05/4240703.aspx)，不管怎么说，大部分的应用程序并不必为此而担心。

一个内存访问经常由一个线性（或虚拟）地址发起，所以L1 cache需要依赖分页单元（paging unit）来求出物理内存页的地址，以便用于缓存标记。与此相反，组索引来自于线性地址的低位，所以不需要转换就可以使用了（在我们的例子中为第6到11位）。因此L1 cache是物理标记但虚拟索引的（**physically tagged**but **virtually indexed**），从而帮助CPU进行并行的查找操作。因为L1 cache的一路绝不会比MMU的一页还大，所以可以保证一个给定的物理地址位置总是关联到同一组，即使组索引是虚拟的。在另一方面L2 cache必须是物理标记和物理索引的，因为它的一路比MMU的一页要大。但是，当一个请求到达L2 cache时，物理地址已经被L1 cache准备（resolved）完毕了，所以L2 cache会工作得很好。

最后，目录单元还存储了对应缓存线的*状态*（state）。在L1代码缓存中的一条缓存线要么是无效的（invalid）要么是共享的（shared，意思是有效的，真的J）。在L1数据缓存和L2缓存中，一条缓存线可以为4个MESI状态之一：被修改的（modified），独占的（exclusive），共享的（shared），无效的（invalid）。Intel缓存是包容式的（**inclusive**）：L1缓存的内容会被复制到L2缓存中。在下一篇讨论线程（threading），锁定（locking）等内容的文章中，这些缓存线状态将发挥作用。下一次，我们将看看前端总线以及内存访问*到底*是怎么工作的。这将成为一个内存研讨周。

（在回复中Dave提到了直接映射缓存（direct-mapped cache）。它们基本上是一种特殊的组相联缓存，只是只有一路而已。在各种折中方案中，它与全相联缓存正好相反：访问非常快捷，但因组冲突而导致的缓存缺失也非常多。）

小结：

1.         内存层次结构的意义在于利用引用的空间局部性和时间局部性原理，将经常被访问的数据放到快速的存储器中，而将不经常访问的数据留在较慢的存储器中。

2.         一般情况下，除了寄存器和L1缓存可以操作指定字长的数据，下层的内存子系统就不会再使用这么小的单位了，而是直接移动数据块，比如以缓存线为单位访问数据。

3.         对于组冲突，可以这么理解：与上文相似，假设一个缓存，由512条缓存线组成，每条线64字节，容量32KB。

a)         假如它是直接映射缓存，由于它往往使用地址的低位直接映射缓存线编号，所以所有的32K倍数的地址（32K，64K，96K等）都会映射到同一条线上（即第0线）。假如程序的内存组织不当，交替的去访问布置在这些地址的数据，则会导致冲突。从外表看来就好像缓存只有1条线了，尽管其他缓存线一直是空闲着的。

b)        如果是全相联缓存，那么每条缓存线都是独立的，可以对应于内存中的任意缓存线。只有当所有的512条缓存线都被占满后才会出现冲突。

c)        组相联是前两者的折中，每一路中的缓存线采用直接映射方式，而在路与路之间，缓存控制器使用全相联映射算法，决定选择一组中的哪一条线。

d)        如果是2路组相联缓存，那么这512条缓存线就被分为了2路，每路256条线，一路16KB。此时所有为16K整数倍的地址（16K，32K，48K等）都会映射到第0线，但由于2路是关联的，所以可以同时有2个这种地址的内容被缓存，不会发生冲突。当然了，如果要访问第三个这种地址，还是要先腾空已有的一条才行。所以极端情况下，从外表看来就好像缓存只有2条线了，尽管其他缓存线一直是空闲着的。

e)         如果是8路组相联缓存（与文中示例相同），那么这512条缓存线就被分为了8路，每路64条线，一路4KB。所以如果数组中元素地址是4K对齐的，并且程序交替的访问这些元素，就会出现组冲突。从外表看来就好像缓存只有8条线了，尽管其他缓存线一直是空闲着的。

]

参考: <http://blog.csdn.net/drshenlei/article/details/4277959>

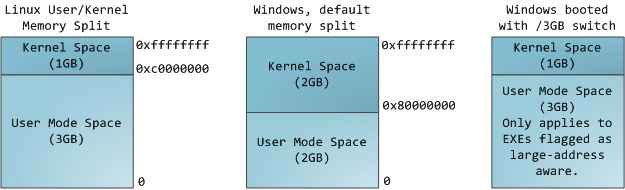
**剖析程序的内存布局**

原文标题：Anatomy of a Program in Memory

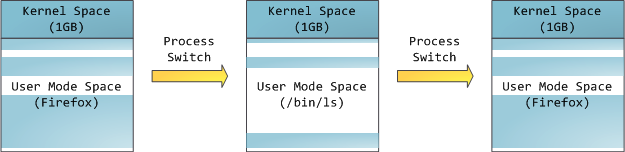
原文地址：http://duartes.org/gustavo/blog/

       内存管理模块是操作系统的心脏；它对应用程序和系统管理非常重要。今后的几篇文章中，我将着眼于实际的内存问题，但也不避讳其中的技术内幕。由于不少概念是通用的，所以文中大部分例子取自32位x86平台的Linux和Windows系统。本系列第一篇文章讲述应用程序的内存布局。

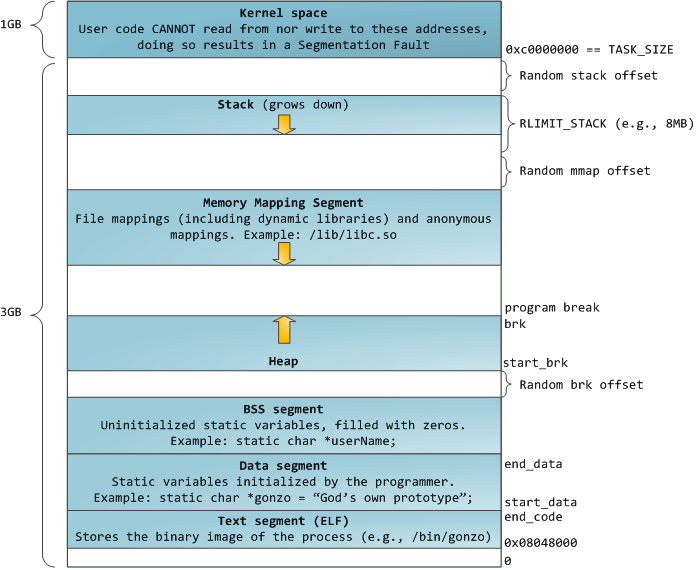
       在多任务操作系统中的每一个进程都运行在一个属于它自己的内存沙盘中。这个沙盘就是**虚拟地址空间**（virtual address space），在32位模式下它**总是一个4GB的内存地址块**。这些虚拟地址通过**页表**（page table）映射到物理内存，页表由操作系统维护并被处理器引用。每一个进程拥有一套属于它自己的页表，但是还有一个隐情。只要虚拟地址被使能，那么它就会作用于这台机器上运行的*所有软件*，包括内核本身。因此一部分虚拟地址必须保留给内核使用：



    这并不意味着内核使用了那么多的物理内存，仅表示它可支配这么大的地址空间，可根据内核需要，将其映射到物理内存。内核空间在页表中拥有较高的[特权级](http://blog.csdn.net/drshenlei/archive/2009/06/12/4265101.aspx)（ring 2或以下），因此只要用户态的程序试图访问这些页，就会导致一个页错误（page fault）。在Linux中，内核空间是持续存在的，并且在所有进程中都映射到同样的物理内存。内核代码和数据总是可寻址的，随时准备处理中断和系统调用。与此相反，用户模式地址空间的映射随进程切换的发生而不断变化：



    蓝色区域表示映射到物理内存的虚拟地址，而白色区域表示未映射的部分。在上面的例子中，Firefox使用了相当多的虚拟地址空间，因为它是传说中的吃内存大户。地址空间中的各个条带对应于不同的内存段（memory segment），如：堆、栈之类的。记住，这些段只是简单的内存地址范围，与Intel处理器的段*没有*关系。不管怎样，下面是一个Linux进程的标准的内存段布局：



    当计算机开心、安全、可爱、正常的运转时，几乎每一个进程的各个段的起始虚拟地址都与上图**完全一致**，这也给远程发掘程序安全漏洞打开了方便之门。一个发掘过程往往需要引用绝对内存地址：栈地址，库函数地址等。远程攻击者必须依赖地址空间布局的一致性，摸索着选择这些地址。如果让他们猜个正着，有人就会被整了。因此，地址空间的随机排布方式逐渐流行起来。Linux通过对[栈](http://lxr.linux.no/linux+v2.6.28.1/fs/binfmt_elf.c)、[内存映射段](http://lxr.linux.no/linux+v2.6.28.1/arch/x86/mm/mmap.c)、[堆](http://lxr.linux.no/linux+v2.6.28.1/arch/x86/kernel/process_32.c)的起始地址加上随机的偏移量来打乱布局。不幸的是，32位地址空间相当紧凑，给随机化所留下的空当不大，[削弱了这种技巧的效果](http://www.stanford.edu/~blp/papers/asrandom.pdf)。

    进程地址空间中最顶部的段是栈，大多数编程语言将之用于存储局部变量和函数参数。调用一个方法或函数会将一个新的**栈桢**（stack frame）压入栈中。栈桢在函数返回时被清理。也许是因为数据严格的遵从LIFO的顺序，这个简单的设计意味着不必使用复杂的数据结构来追踪栈的内容，只需要一个简单的指针指向栈的顶端即可。因此压栈（pushing）和退栈（popping）过程非常迅速、准确。另外，持续的重用栈空间有助于使活跃的栈内存保持在CPU缓存中，从而加速访问。进程中的每一个线程都有属于自己的栈。

       通过不断向栈中压入的数据，超出其容量就有会耗尽栈所对应的内存区域。这将触发一个页故障（page fault），并被Linux的expand\_stack()处理，它会调用acct\_stack\_growth()来检查是否还有合适的地方用于栈的增长。如果栈的大小低于RLIMIT\_STACK（通常是8MB），那么一般情况下栈会被加长，程序继续愉快的运行，感觉不到发生了什么事情。这是一种将栈扩展至所需大小的常规机制。然而，如果达到了最大的栈空间大小，就会栈溢出（stack overflow），程序收到一个段错误（Segmentation Fault）。当映射了的栈区域扩展到所需的大小后，它就不会再收缩回去，即使栈不那么满了。这就好比联邦预算，它总是在增长的。

       动态栈增长是唯一一种访问未映射内存区域（图中白色区域）而被允许的情形。其它任何对未映射内存区域的访问都会触发页故障，从而导致段错误。一些被映射的区域是只读的，因此企图写这些区域也会导致段错误。

       在栈的下方，是我们的内存映射段。此处，内核将文件的内容直接映射到内存。任何应用程序都可以通过Linux的mmap()系统调用（[实现](http://lxr.linux.no/linux+v2.6.28.1/arch/x86/kernel/sys_i386_32.c)）或Windows的CreateFileMapping() / [MapViewOfFile()](http://msdn.microsoft.com/en-us/library/aa366761%28VS.85%29.aspx)请求这种映射。内存映射是一种方便高效的文件I/O方式，所以它被用于加载动态库。创建一个不对应于任何文件的匿名内存映射也是可能的，此方法用于存放程序的数据。在Linux中，如果你通过malloc()请求一大块内存，C运行库将会创建这样一个匿名映射而不是使用堆内存。'大块'意味着比MMAP\_THRESHOLD还大，缺省是128KB，可以通过mallopt()调整。

       说到堆，它是接下来的一块地址空间。与栈一样，堆用于运行时内存分配；但不同点是，堆用于存储那些生存期与函数调用无关的数据。大部分语言都提供了堆管理功能。因此，满足内存请求就成了语言运行时库及内核共同的任务。在C语言中，堆分配的接口是malloc()系列函数，而在具有垃圾收集功能的语言（如C#）中，此接口是new关键字。

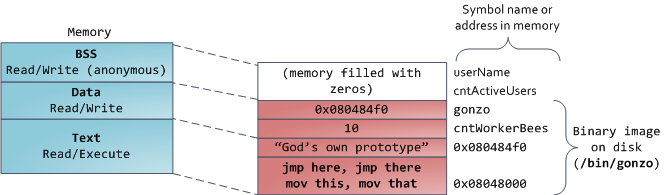
       如果堆中有足够的空间来满足内存请求，它就可以被语言运行时库处理而不需要内核参与。否则，堆会被扩大，通过brk()系统调用（[实现](http://lxr.linux.no/linux+v2.6.28.1/mm/mmap.c)）来分配请求所需的内存块。堆管理是很[复杂](http://g.oswego.edu/dl/html/malloc.html)的，需要精细的算法，应付我们程序中杂乱的分配模式，优化速度和内存使用效率。处理一个堆请求所需的时间会大幅度的变动。实时系统通过[特殊目的分配器](http://rtportal.upv.es/rtmalloc/)来解决这个问题。堆也可能会变得零零碎碎，如下图所示：



    最后，我们来看看最底部的内存段：BSS，数据段，代码段。在C语言中，BSS和数据段保存的都是静态（全局）变量的内容。区别在于BSS保存的是未被初始化的静态变量内容，它们的值不是直接在程序的源代码中设定的。BSS内存区域是匿名的：它不映射到任何文件。如果你写static int cntActiveUsers，则cntActiveUsers的内容就会保存在BSS中。

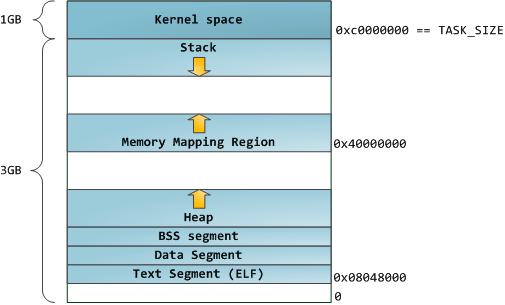
    另一方面，数据段保存在源代码中已经初始化了的静态变量内容。这个内存区域不是匿名的。它映射了一部分的程序二进制镜像，也就是源代码中指定了初始值的静态变量。所以，如果你写static int cntWorkerBees = 10，则cntWorkerBees的内容就保存在数据段中了，而且初始值为10。尽管数据段映射了一个文件，但它是一个私有内存映射，这意味着更改此处的内存不会影响到被映射的文件。也必须如此，否则给全局变量赋值将会改动你硬盘上的二进制镜像，这是不可想象的。

    下图中数据段的例子更加复杂，因为它用了一个指针。在此情况下，指针gonzo（4字节内存地址）*本身的值*保 存在数据段中。而它所指向的实际字符串则不在这里。这个字符串保存在代码段中，代码段是只读的，保存了你全部的代码外加零零碎碎的东西，比如字符串字面 值。代码段将你的二进制文件也映射到了内存中，但对此区域的写操作都会使你的程序收到段错误。这有助于防范指针错误，虽然不像在C语言编程时就注意防范来得那么有效。下图展示了这些段以及我们例子中的变量：



    你可以通过阅读文件/proc/pid\_of\_process/maps来检验一个Linux进程中的内存区域。记住一个段可能包含许多区域。比如，每个内存映射文件在mmap段中都有属于自己的区域，动态库拥有类似BSS和数据段的额外区域。下一篇文章讲说明这些"区域"（area）的真正含义。有时人们提到"数据段"，指的就是全部的数据段 + BSS + 堆。

    你可以通过nm和objdump命令来察看二进制镜像，打印其中的符号，它们的地址，段等信息。最后需要指出的是，前文描述的虚拟地址布局在Linux中是一种"灵活布局"（flexible layout），而且以此作为默认方式已经有些年头了。它假设我们有值RLIMIT\_STACK。当情况不是这样时，Linux退回使用"经典布局"（classic layout），如下图所示：



       对虚拟地址空间的布局就讲这些吧。下一篇文章将讨论内核是如何跟踪这些内存区域的。我们会分析内存映射，看看文件的读写操作是如何与之关联的，以及内存使用概况的含义。

参考:

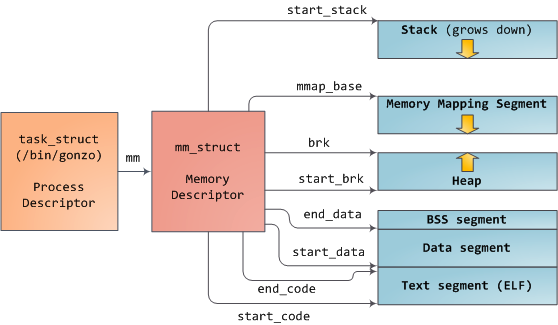
<http://blog.csdn.net/drshenlei/article/details/4339110>

**内核是如何管理内存的**

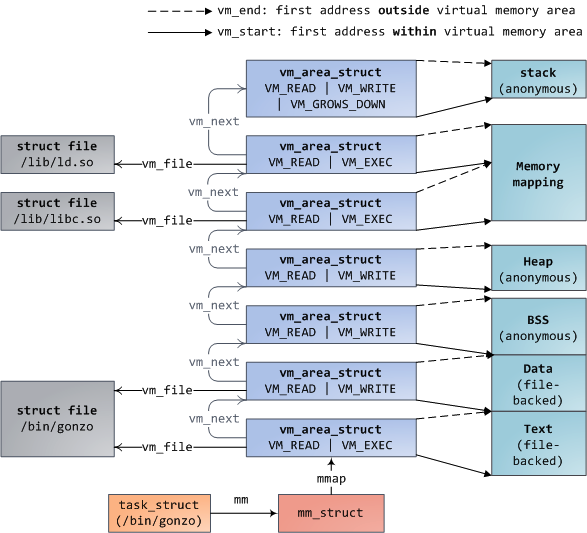
原文标题：How The Kernel Manages Your Memory

原文地址：http://duartes.org/gustavo/blog/

       在仔细审视了进程的[虚拟地址布局](http://blog.csdn.net/drshenlei/archive/2009/07/11/4339110.aspx)之后，让我们把目光转向内核以及其管理用户内存的机制。再次从gonzo图示开始：



    Linux进程在内核中是由task\_struct的实例来表示的，即进程描述符。task\_struct的mm字段指向**内存描述符**（memory descriptor），即mm\_struct，一个程序的内存的执行期摘要。它存储了上图所示的内存段的起止位置，进程所使用的物理内存页的[数量](http://lxr.linux.no/linux+v2.6.28.1/include/linux/mm_types.h)（**rss**表示Resident Set Size），虚拟内存空间的[使用量](http://lxr.linux.no/linux+v2.6.28.1/include/linux/mm_types.h)，以及其他信息。我们还可以在内存描述符中找到用于管理程序内存的两个重要结构：**虚拟内存区域**集合（the set of virtual memory areas）及**页表**（page table）。Gonzo的内存区域如下图所示：

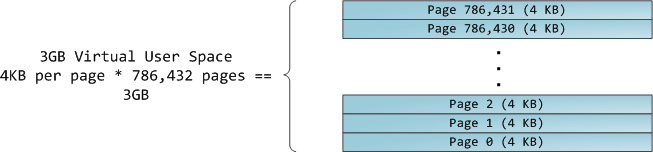


    每一个虚拟内存区域（简称VMA）是一个连续的虚拟地址范围；这些区域不会交叠。一个vm\_area\_struct的实例完备的描述了一个内存区域，包括它的起止地址，决定访问权限和行为的[标志位](http://lxr.linux.no/linux+v2.6.28/include/linux/mm.h)，还有vm\_file字段，用于指出被映射的文件（如果有的话）。一个VMA如果没有映射到文件，则是**匿名的**（anonymous）。除memory mapping 段以外，上图中的每一个内存段（如：堆，栈）都对应于一个单独的VMA。这并不是强制要求，但在x86机器上经常如此。VMA并不关心它在哪一个段。

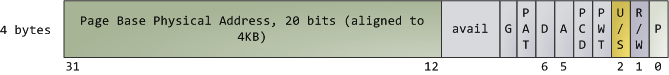
       一个程序的VMA同时以两种形式存储在它的内存描述符中：一个是按起始虚拟地址排列的链表，保存在mmap字段；另一个是[红黑树](http://en.wikipedia.org/wiki/Red_black_tree)，根节点保存在mm\_rb字段。红黑树使得内核可以快速的查找出给定虚拟地址所属的内存区域。当你读取文件/proc/pid\_of\_process/maps时，内核只须简单的遍历指定进程的VMA链表，并[打印出每一项](http://lxr.linux.no/linux+v2.6.28.1/fs/proc/task_mmu.c)来即可。

    在Windows中，EPROCESS块可以粗略的看成是task\_struct和mm\_struct的组合。VMA在Windows中的对应物时虚拟地址描述符（Virtual Address Descriptor），或简称VAD；它们保存在平衡树中（AVL tree）。你知道Windows和Linux最有趣的地方是什么吗？就是这些细小的不同点。

           4GB虚拟地址空间被分割为许多**页**（page）。x86处理器在32位模式下所支持的页面大小为4KB，2MB和4MB。Linux和Windows都使用4KB大小的页面来映射用户部分的虚拟地址空间。第0-4095字节在第0页，第4096-8191字节在第1页，以此类推。VMA的大小必须是*页面大小的整数倍*。下图是以4KB分页的3GB用户空间：



       处理器会依照**页表**（page table）来将虚拟地址转换到物理内存地址。每个进程都有属于自己的一套页表；一旦进程发生了切换，用户空间的页表也会随之切换。Linux在内存描述符的pgd字段保存了一个指向进程页表的指针。每一个虚拟内存页在页表中都有一个与之对应的**页表项**（page table entry），简称PTE。它在普通的x86分页机制下，是一个简单的4字节记录，如下图所示：

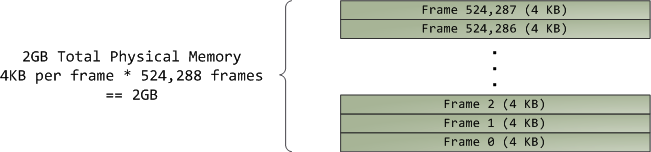


        Linux有一些函数可以用于[读取](http://lxr.linux.no/linux+v2.6.28.1/arch/x86/include/asm/pgtable.h)或[设置](http://lxr.linux.no/linux+v2.6.28.1/arch/x86/include/asm/pgtable.h)PTE中的每一个标志。P位告诉处理器虚拟页面是否**存在**于（present）物理内存中。如果是0，访问这个页将触发页故障（page fault）。记住，当这个位是0时，内核可以根据喜好，随意的使用其余的字段。R/W标志表示读/写；如果是0，页面就是只读的。U/S标志表示用户/管理员；如果是0，则这个页面只能被内核访问。这些标志用于实现只读内存和保护内核空间。

        D位和A位表示数据脏（**dirty**）和访问过（**accessed**）。脏表示页面被执行过写操作，访问过表示页面被读或被写过。这两个标志都是粘滞的：处理器只会将它们置位，之后必须由内核来清除。最后，PTE还保存了对应该页的起始物理内存地址，对齐于4KB边界。PTE中的其他字段我们改日再谈，比如物理地址扩展（Physical Address Extension）。

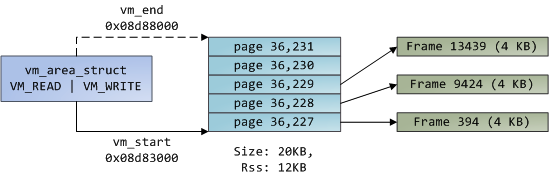
    虚拟页面是内存保护的最小单元，因为页内的所有字节都共享U/S和R/W标志。然而，同样的物理内存可以被映射到不同的页面，甚至可以拥有不同的保护标志。值得注意的是，在PTE中没有对执行许可（execute permission）的设定。这就是为什么经典的x86分页可以执行位于stack上的代码，从而为黑客利用堆栈溢出提供了便利（使用return-to-libc和其他技术，甚至可以利用不可执行的堆栈）。PTE缺少不可执行（no-execute）标志引出了一个影响更广泛的事实：VMA中的各种许可标志可能会也可能不会被明确的转换为硬件保护。对此，内核可以尽力而为，但始终受到架构的限制。

       虚拟内存并不存储任何东西，它只是将程序地址空间映射到底层的物理内存上，后者被处理器视为一整块来访问，称作**物理地址空间**（physical address space）。对物理内存的操作还与总线[有点联系](http://blog.csdn.net/drshenlei/archive/2009/06/05/4243733.aspx)，好在我们可以暂且忽略这些并假设物理地址范围以字节为单位递增，从０到最大可用内存数。这个物理地址空间被内核分割为一个个**页帧**（page frame）。处理器并不知道也不关心这些帧，然而它们对内核至关重要，因为**页帧是物理内存管理的最小单元**。Linux和Windows在32位模式下，都使用4KB大小的页帧；以一个拥有2GB RAM的机器为例：



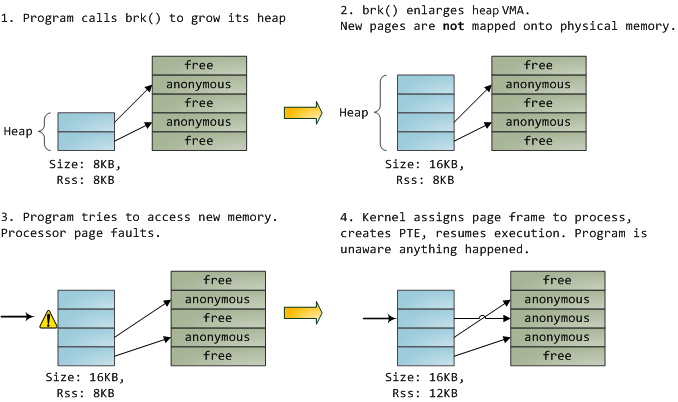
    在Linux中，每一个页帧都由一个[描述符](http://lxr.linux.no/linux+v2.6.28/include/linux/mm_types.h)和[一些标志](http://lxr.linux.no/linux+v2.6.28/include/linux/page-flags.h)所跟踪。这些描述符合在一起，记录了计算机内的全部物理内存；可以随时知道每一个页帧的准确状态。物理内存是用buddy memory allocation技术来管理的，因此如果一个页帧可被buddy 系统分配，则它就是**可用的**（free）。一个被分配了的页帧可能是**匿名的**（anonymous），保存着程序数据；也可能是**页缓冲的**（page cache），保存着一个文件或块设备的数据。还有其他一些古怪的页帧使用形式，但现在先不必考虑它们。Windows使用一个类似的页帧编号（Page Frame Number简称PFN）数据库来跟踪物理内存。

       让我们把虚拟地址区域，页表项，页帧放到一起，看看它们到底是怎么工作的。下图是一个用户堆的例子：



       蓝色矩形表示VMA范围内的页，箭头表示页表项将页映射到页帧上。一些虚拟页并没有箭头；这意味着它们对应的PTE的**存在位**（Present flag）为0。形成这种情况的原因可能是这些页还没有被访问过，或者它们的内容被系统换出了（swap out）。无论那种情况，对这些页的访问都会导致页故障（page fault），即使它们处在VMA之内。VMA和页表的不一致看起来令人奇怪，但实际经常如此。

       一个VMA就像是你的程序和内核之间的契约。你请求去做一些事情（如：内存分配，文件映射等），内核说"行"，并创建或更新适当的VMA。但它并非立刻就去完成请求，而是一直等到出现了页故障才会真正去做。内核就是一个懒惰，骗人的败类；这是虚拟内存管理的基本原则。它对大多数情况都适用，有些比较熟悉，有些令人惊讶，但这个规则就是这样：VMA记录了双方*商定做什么*，而PTE反映出懒惰的内核*实际做了什么*。这两个数据结构共同管理程序的内存；都扮演着解决页故障，释放内存，换出内存（swapping memory out）等等角色。让我们看一个简单的内存分配的例子：



       当程序通过brk()系统调用请求更多的内存时，内核只是简单的[更新](http://lxr.linux.no/linux+v2.6.28.1/mm/mmap.c)堆的VMA，然后说搞好啦。其实此时并没有页帧被分配，新的页也并没有出现于物理内存中。一旦程序试图访问这些页，处理器就会报告页故障，并调用do\_page\_fault()。它会通过调用find\_vma()去[搜索](http://lxr.linux.no/linux+v2.6.28/arch/x86/mm/fault.c)哪一个VMA含盖了产生故障的虚拟地址。如果找到了，还会根据VMA上的访问许可来比对检查访问请求（读或写）。如果没有合适的VMA，也就是说内存访问请求没有与之对应的合同，进程就会被处以段错误（Segmentation Fault）的罚单。

    当一个VMA被[找到](http://lxr.linux.no/linux+v2.6.28/arch/x86/mm/fault.c)后，内核必须[处理](http://lxr.linux.no/linux+v2.6.28/mm/memory.c)这个故障，方式是察看PTE的内容以及VMA的类型。在我们的例子中，PTE显示了该页并[不存在](http://lxr.linux.no/linux+v2.6.28/mm/memory.c)。事实上，我们的PTE是完全空白的（全为0），在Linux中意味着虚拟页还没有被映射。既然这是一个匿名的VMA，我们面对的就是一个纯粹的RAM事务，必须由do\_anonymous\_page()处理，它会分配一个页帧并生成一个PTE，将出故障的虚拟页映射到那个刚刚分配的页帧上。

       事情还可能有些不同。被换出的页所对应的PTE，例如，它的Present标志是0但并不是空白的。相反，它记录了页面内容在交换系统中的位置，这些内容必须从磁盘读取出来并通过do\_swap\_page()加载到一个页帧当中，这就是所谓的major fault。

       至此我们走完了"内核的用户内存管理"之旅的前半程。在下一篇文章中，我们将把文件的概念也混进来，从而建立一个内存基础知识的完成画面，并了解其对系统性能的影响。

参考:

<http://blog.csdn.net/drshenlei/article/details/4350928>

**转: 页面缓存-内存与文件的那些事**

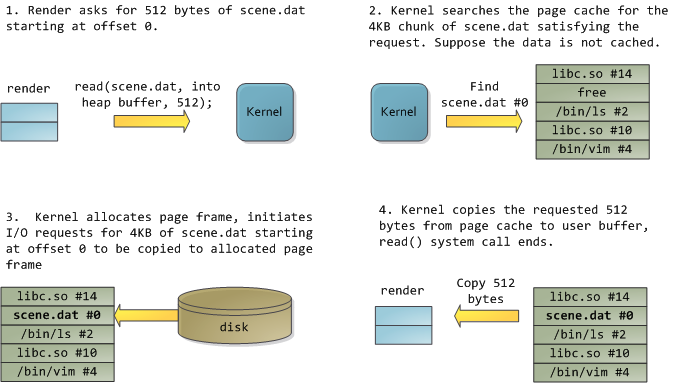
原文标题：Page Cache, the Affair Between Memory and Files

原文地址：http://duartes.org/gustavo/blog/

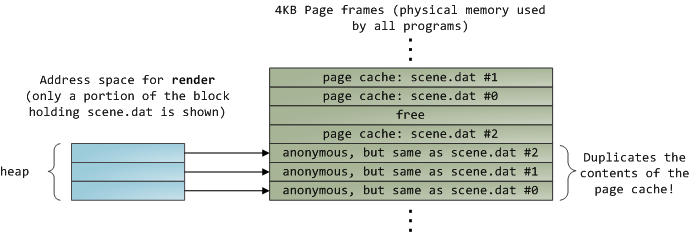
   上次我们考察了内核如何为一个用户进程[管理虚拟内存](http://blog.csdn.net/drshenlei/archive/2009/07/15/4350928.aspx)，但是没有涉及文件及I/O。这次我们的讨论将涵盖非常重要且常被误解的文件与内存间关系的问题，以及它对系统性能的影响。

提到文件，操作系统必须解决两个重要的问题。首先是硬盘驱动器的存取速度缓慢得令人头疼（相对于内存而言），尤其是[磁盘的寻道性能](http://blog.csdn.net/drshenlei/archive/2009/06/05/4240703.aspx)。第二个是要满足'一次性加载文件内容到物理内存并在程序间*共享*'的需求。如果你使用[进程浏览器](http://technet.microsoft.com/en-us/sysinternals/bb896653.aspx)翻看Windows进程，就会发现大约15MB的共享DLL被加载进了每一个进程。我目前的Windows系统就运行了100个进程，如果没有共享机制，那将消耗大约1.5GB的物理内存*仅仅用于*存放公用DLL。这可不怎么好。同样的，几乎所有的Linux程序都需要ld.so和libc，以及其它的公用函数库。

令人愉快的是，这两个问题可以被一石二鸟的解决：页面缓存（page cache），内核用它保存与页面同等大小的文件数据块。为了展示页面缓存，我需要祭出一个名叫render的Linux程序，它会打开一个scene.dat文件，每次读取其中的512字节，并将这些内容保存到一个建立在堆上的内存块中。首次的读取是这样的：

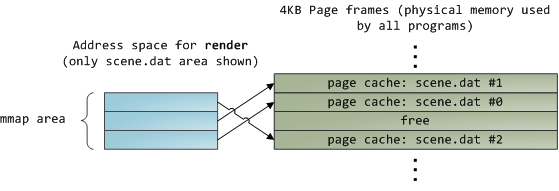


在读取了12KB以后，render的堆以及相关的页帧情况如下：



这看起来很简单，但还有很多事情会发生。首先，即使这个程序只调用了常规的read函数，此时也会有三个 4KB的页帧存储在页面缓存当中，它们持有scene.dat的一部分数据。尽管有时这令人惊讶，但的确**所有的常规文件I/O都是通过页面缓存来进行的**。在x86 Linux里，内核将文件看作是4KB大小的数据块的序列。即使你只从文件读取一个字节，包含此字节的整个4KB数据块都会被读取，并放入到页面缓存当中。这样做是有道理的，因为磁盘的持续性数据吞吐量很不错，而且一般说来，程序对于文件中某区域的读取都不止几个字节。页面缓存知道每一个4KB数据块在文件中的对应位置，如上图所示的#0, #1等等。与Linux的页面缓存类似，Windows使用256KB的views。

不幸的是，在一个普通的文件读取操作中，内核必须复制页面缓存的内容到一个用户缓冲区中，这不仅消耗CPU时间，伤害了CPU cache的性能，还**因为存储了重复信息而浪费物理内存**。如上面每张图所示，scene.dat的内容被保存了两遍，而且程序的每个实例都会保存一份。至此，我们缓和了磁盘延迟的问题，但却在其余的每个问题上惨败。**内存映射文件**（memory-mapped files）将引领我们走出混乱：



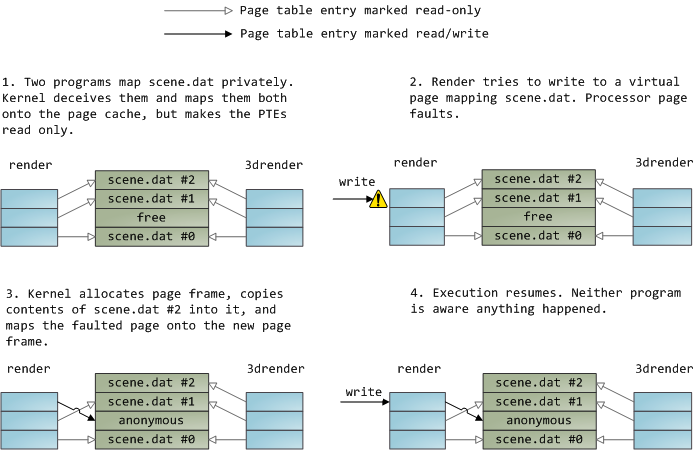
当你使用文件映射的时候，内核将你的程序的虚拟内存页直接映射到页面缓存上。这将导致一个显著的性能提升：[《](http://www.amazon.com/Windows-Programming-Addison-Wesley-Microsoft-Technology/dp/0321256190/)Windows系统编程》指出常规的文件读取操作运行时性能改善30%以上；[《](http://www.amazon.com/Programming-Environment-Addison-Wesley-Professional-Computing/dp/0321525949/)Unix环境高级编程》指出类似的情况也发生在Linux和Solaris系统上。你还可能因此而节省下大量的物理内存，这依赖于你的程序的具体情况。

和以前一样，提到性能，实际测量才是王道，但是内存映射的确值得被程序员们放入工具箱。相关的API也很漂亮，它提供了像访问内存中的字节一样的方式来访问一个文件，不需要你多操心，也不牺牲代码的可读性。回忆一下[地址空间](http://blog.csdn.net/drshenlei/archive/2009/07/11/4339110.aspx)、还有那个在Unix类系统上关于mmap的实验，Windows下的CreateFileMapping及其在高级语言中的各种可用封装。当你映射一个文件时，它的内容并不是立刻就被全部放入内存的，而是依赖[页故障](http://lxr.linux.no/linux+v2.6.28/mm/memory.c)（page fault）按需读取。在[获取](http://lxr.linux.no/linux+v2.6.28/mm/filemap.c)了一个包含所需的文件数据的页帧后，对应的故障处理函数会[将你的虚拟内存页映射到页面缓存上](http://lxr.linux.no/linux+v2.6.28/mm/memory.c)。如果所需内容不在缓存当中，此过程还将包含磁盘I/O操作。

现在给你出一个流行的测试题。想象一下，在最后一个render程序的实例退出之时，那些保存了scene.dat的页面缓存会被立刻清理吗？人们通常会这样认为，但这是个坏主意。如果你仔细想想，我们经常会在一个程序中创建一个文件，退出，紧接着在第二个程序中使用这个文件。页面缓存必须能处理此类情况。如果你再多想想，内核何必总是要舍弃页面缓存中的内容呢？记住，磁盘比RAM慢5个数量级，因此一个页面缓存的命中（hit）就意味着巨大的胜利。只要还有足够的空闲物理内存，缓存就应该尽可能保持满状态。所以它与特定的进程并不相关，而是一个系统级的资源。如果你一周前运行过render，而此时scene.dat还在缓存当中，那真令人高兴。这就是为什么内核缓存的大小会稳步增加，直到缓存上限。这并非因为操作系统是破烂货，吞噬你的RAM，事实上这是种好的行为，反而释放物理内存才是一种浪费。缓存要利用得越充分越好。

由于使用了页面缓存体系结构，当一个程序调用write()时，相关的字节被简单的复制到页面缓存中，并且将页面标记为脏的（dirty）。磁盘I/O一般**不会**立刻发生，因此你的程序的执行不会被打断去等待磁盘设备。这样做的缺点是，如果此时计算机死机，那么你写入的数据将不会被记录下来。因此重要的文件，比如数据库事务记录必须被fsync() （但是还要小心磁盘控制器的缓存）。另一方面，读取操作一般会打断你的程序直到准备好所需的数据。内核通常采用积极加载（eager loading）的方式来缓解这个问题。以**提前读取**（read ahead）为例，内核会预先加载一些页到页面缓存，并期待你的读取操作。通过提示系统即将对文件进行的是顺序还是随机读取操作（参看madvise(), [readahead()](http://www.kernel.org/doc/man-pages/online/pages/man2/readahead.2.html), [Windows](http://msdn.microsoft.com/en-us/library/aa363858%28VS.85%29.aspx)缓存提示），你可以帮助内核调整它的积极加载行为。Linux的确会对内存映射文件进行预取，但我不太确定Windows是否也如此。最后需要一提的是，你还可以通过在Linux中使用O\_DIRECT或在Windows中使用NO\_BUFFERING来绕过页面缓存，有些数据库软件就是这么做的。

   一个文件映射可以是**私有的**（private）或**共享的**（shared）。这里的区别只有在**更改**（update）内存中的内容时才会显现出来：在私有映射中，更改并不会被提交到磁盘或对其他进程可见，而这在共享的映射中就会发生。内核使用**写时拷贝**（copy on write）技术，通过页表项（page table entries），实现私有映射。在下面的例子中，render和另一个叫render3d的程序（我是不是很有创意？）同时私有映射了scene.dat。随后render改写了映射到此文件的虚拟内存区域：



上图所示的只读的页表项*并不*意 味着映射是只读的，它们只是内核耍的小把戏，用于共享物理内存直到可能的最后一刻。你会发现'私有'一词是多么的不恰当，你只需记住它只在数据发生更改时 起作用。此设计所带来的一个结果就是，一个以私有方式映射文件的虚拟内存页可以观察到其他进程对此文件的改动，只要之前对这个内存页进行的都是读取操作。 一旦发生过写时拷贝，就不会再观察到其他进程对此文件的改动了。此行为不是内核提供的，而是在x86系统上就会如此。另外，从API的角度来说，这也是合理的。与此相反，共享映射只是简单的映射到页面缓存，仅此而已。对页面的所有更改操作对其他进程都可见，而且最终会执行磁盘操作。最后，如果此共享映射是只读的，那么页故障将触发段错误（segmentation fault）而不是写时拷贝。

   被动态加载的函数库通过文件映射机制放入到你的程序的地址空间中。这里没有任何特别之处，同样是采用私有文件映射，跟提供给你调用的常规API别无二致。下面的例子展示了两个运行中的render程序的一部分地址空间，还有物理内存。它将我们之前看到的概念都联系在了一起。

