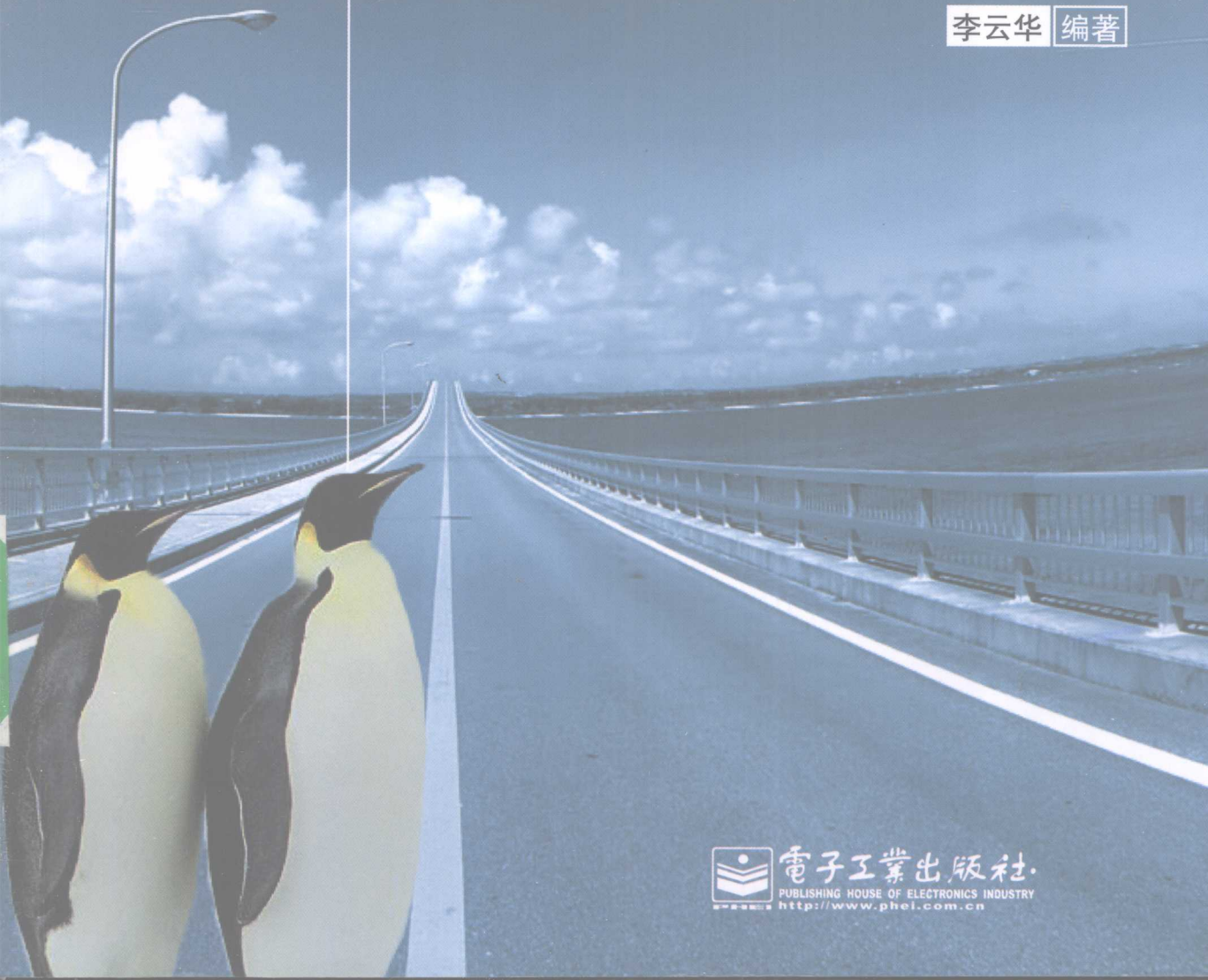
Broadview
www.broadview.com.cn

独辟蹊径品内核：Linux内核源代码导读

李云华 编著



電子工業出版社
PUBLISHING HOUSE OF ELECTRONICS INDUSTRY
http://www.phei.com.cn

独辟蹊径品内核：Linux内核源代码导读

李云华 编著

電子工業出版社
Publishing House of Electronics Industry
北京•BEIJING

内 容 简 介

本书根据最新的 2.6.24 内核为基础。在讲述方式上，本书注重实例分析，尽量在讨论“如何做”的基础上，深入讨论为什么要这么做，从而实现本书的写作宗旨：“授人以渔”。在内容安排上，本书包含以下章节 x86 硬件基础；基础知识；Linux 内核 Makefile 分析；Linux 内核启动；内存管理；中断和异常处理；系统调用；信号机制在类 UNIX 系统中；时钟机制；进程管理；调度器；文件系统；常用内核分析方法。

本书适合初、中级 Linux 用户、从事内核相关开发的从业人员，也可以作为各类院校相关专业的教材及 Linux 培训班的教材，也可作为 Linux 内核学习的专业参考书。

读者可登录 www.broadview.com.cn，下载本书源代码。

图书在版编目（CIP）数据

独辟蹊径品内核：Linux 内核源代码导读 / 李云华编著. —北京：电子工业出版社，2009.8
ISBN 978-7-121-08515-4

I. 独… II. 李… III. Linux 操作系统 IV. TP316.89

中国版本图书馆 CIP 数据核字（2009）第 038405 号

责任编辑：高洪霞
印　　刷：北京东光印刷厂
装　　订：三河市皇庄路通装订厂
出版发行：电子工业出版社
　　北京市海淀区万寿路 173 信箱　邮编 100036
开　　本：787×1092　1/16　印张：31　字数：804 千字
印　　次：2009 年 8 月第 1 次印刷
印　　数：4000 册　定价：65.00 元

凡所购买电子工业出版社图书有缺损问题，请向购买书店调换。若书店售缺，请与本社发行部联系，联系及邮购电话：（010）88254888。

质量投诉请发邮件至 zlts@phei.com.cn，盗版侵权举报请发邮件至 dbqq@phei.com.cn。

服务热线：（010）88258888。

前 言

几乎每一个操作系统内核的学习者在初学阶段都会感觉到难以入门。这是由于内核涉及到知识面非常广泛，需要学习者从根本上掌握大量的知识，这包括：程序编译，链接，装载的细节，操作系统理论，计算机系统体系结构，数据结构与算法，深厚的C/汇编语言编程功底。如此相对较高的门槛常常令很大一部分初学者望而却步。那么是不是一定要先学好以上的各门知识后才能学习内核呢？事实上大部分学习者在学习以上各门知识都会遇到同样的问题，因为知识是一个网状结构。所以重要的不是先去学会什么知识，而是学会如何学习，学会在自己掌握的知识体系上提出问题，学会思考，进而坚持不懈的解决心中的疑问。笔者从学完C/C++语言开始，由于C/C++的示例程序都是在命令行下的，于是常常想如何才能编写出视窗程序，学习了MFC，但是同样想不通诸如WM_CHAR，WM_LBUTTONDOWN的消息从何而来，带着MFC中诸多疑问，笔者开始学习Windows SDK程序开发，在这个学习过程中感觉对MFC的认识更加深入了，但同时又有新的问题想不通，于是进而学习Windows DDK，之后开始学习操作系统内核。在这个过程中，笔者也遇到过数不尽的疑问，但是都是需要的时候再补充相关知识。因此初学者要明白，学习并不需要等到“万事具备”了才可以开始。需要的是保持好奇心，养成思考的习惯，树立解决问题的决心。很多读者渴望寻找好的入门教材，也常常有人问看什么书才能进步的快，但是当他们看了别人推荐的书却没有取得同样的收获，这是为什么呢？笔者认为，读书有以下几种境界：

1. 面对书上讲到的某个知识点，不能接合自己掌握的知识提出疑问[1]，仅仅知识死记书本上的东西。这种状态就算学到最高境界，也仅仅只是能把书本上的知识点完好的记下来在脑海中形成孤立的知识节点。

2. 面对书本上讲到的某个知识点，能接合自己掌握的知识提出疑问，但是大多数时候没有探索精神，仅仅局限于到其他书籍或者请教别人来排除心中的疑问。脑海中的知识形成了简单网状结构，但由于探索能力长期得不到锻炼，综合自己的知识去分析和解决问题的能力十分有限。

3. 面对书上讲到的某个知识点，能接合自己掌握的知识提出疑问，并且能根据问题补充

[1]比如：如何才能根据自己掌握的知识来解释或者推理出新看到的知识点，或者找到要解释推理出新知识还欠缺的知识点在哪方面。

相关必要的知识，不断综合分析各知识点的关系，提出各种假设和验证排除的方法并亲自验证，解决不了问题决不罢休。如能经过长期锻炼，其脑海中的知识点形成复杂的网状结构，综合分析能力必将加强。

4. 根据自己掌握的知识，提出全新的问题，并始终坚持找到答案为止。这种境界需要渊博的知识作为基础。

因此，不要还没学内核就被吓倒，说了这么多看似和内核无关的东西，就是要从先排除读者的心理担忧，树立正确的态度，重要的不是学会什么，而是学会学习。确定自己处于哪一种学习境界，然后通过学习某项具体的知识把自己提升到更高的境界。在现实生活中我们不难发现，能力强的学什么都又快又好。其根本原因在于他们处于更高的学习境界，并形成了良性循环！

有很多的人都渴望学习操作系统内核，但是内核涉及到的知识非常广泛，因此很多人半途而废，许多人往往抱怨没有好的书籍，教材。实际上，对于同一本书籍，不同的读者收获也是不同的，这取决于他们的态度和学习方法。笔者建议，在读书的时候，一定要以自己心中的疑问作为主线，而不要没有任何疑问就死记书本上的知识。

如何使用本书

笔者认为对于任何知识的学习，首先是以自我为中心，任何书籍资料都是用来解答读者心中的疑问的，因此在你阅读一本书时，首先要明确自己的疑问是什么？这可以是一个非常粳概的问题，例如：“Linux 内核是什么？”；也可以是一个非常细节的问题，例如：“按下键盘上的 A，到屏幕上显示出字符 A 的内部原理”。当你有了来自内心深处经过独立思考的疑问后，阅读对你来说是一种享受，一种乐趣。来自内心的疑问，经过不断的综合分析，缜密的推理，坚持不懈的查阅和求索，之后拨开迷雾见天日喜悦只有经过才能体会。虽然本书是一本很厚的书，但是这不是畏惧的理由，也不要因为它厚，就给自己下一个决心，制定一个阅读计划，几个月要读完本书。学习是主动探求的过程，而不是被动接受，在这个过程中，有太多的东西，不是谁可以计划出来的。例如：在笔者学习内核之初，看到大量的传言，读完《Understanding the linux Kernel》，读完《Linux 内核情景分析》... 就可以成为“高手”了。于是笔者常常捧着厚厚的书，寻思着自己什么时候可以读完，然而有时好几天也前进不了几页，免不了感慨自己今生将与“高手”无缘，但是又心有不甘，于是囫囵吞枣的“快速”前进，但是越前进，就越感觉到艰难。“欲速则不达”这个道理人人都懂，但是在切身体会之前，人人都会犯这个错误。在经历了很长一段曲折和郁闷之后，笔者摆脱了“书”的束缚，完全以自己的疑问为中心，例如在读到中断处理时，由于知识不够全面，于是丢开内核的书籍，阅读了大量的计算机体系结构方面的资料，同样计算机体系结构的书籍也很厚，但是我也没有想过要把它们读完，这时只捡中断相关的读，之后再来读内核的书籍，发现自己原理懂了，但是具体到理解代码时，就迷糊了，于是有补充 GCC 内嵌汇编，C 代码编译到汇编代码的相关知识，反复试验等等。这个过程很慢，但是积累到最后，笔者发现自己读的非常快，甚至可以不读了，因为很多地方，只要读到前面的，就领悟了作者后面想要说什么了。

至今，我仍然没有完成当初为了成为“高手”而制定下的“宏伟”目标，因为我没有完整的读完《Understanding the Linux Kernel》、《Linux 内核情景分析》或《Linux 内核完全剖析》等等这类传说中“惊世骇俗”之作中的任何一本。但是笔者却从这些著作中受益菲浅。

现在，你应该知道要如何使用本书了吧？那就是不要拘泥如任何教条。虽然本书经笔者从初学到现在的心得体会以及相关笔记和资料整理而成，初学者的大量疑问都能在本书本书中找到答案。但是每个人都是独一无二的，笔者希望任何一个读者能综合利用本书和其它相关资料寻找你自己的答案。多问一点为什么，多一点假设，多一点思考，多一点推

理，多一点试验，多一点坚持。最后，你会感慨原来传说中的任何“秘籍”都是“浪得虚名”，因为读完它，你不一定能成为“高手”，而“高手”却不需要读完它。能否成为“高手”的决定性因素取决于你的学习方法和学习态度，而好的“秘籍”仅仅只是催化剂。

由于笔者水平有限，纰漏之处在所难免，因此希望读者能够指正。笔者的联系方式是：Addylee2004@163.com. 另外对本书有任何问题，意见或建议，勘误等等，欢迎前来 http://www.osplay.org 与笔者进行讨论。

目 录

第1章　x86 硬件基础

如果你是一个 Linux 内核初学者，你一定常常遇到：保护模式，分段机制，分页机制，段地址，线性地址，中断门，调用门，局部描述符，全局描述符，等等这样的名词。这些概念常常把初学者弄得“云里来，雾里去”。你会常常感慨，Intel 为什么要设计这么复杂的概念呢？仅仅是一个地址机制，就常常让初学者打开书本，发现自己会算了，可是一旦关上书本，换个例子，又迷惑了。

事实上这些机制的背后都有它的缘由，任何一个复杂的设计都是由一个简单的设计发展起来的，当简单的设计满足不了实际需要时，就会一步一步地革新，直到问题被圆满地解决。因此理解一个“复杂”的东西的最好方式不是去记住它，而是要从最简单的地方入手，一步一步地推敲，简单的设计在现实中会遇到什么问题？又该如何解决这个问题？再联系到你现在要理解的复杂的例子，慢慢地建立起一条完整的线索，这样知识不会出现断层，你也不需要一次跨越一个鸿沟，一切都水到渠成。例如：在学习保护模式中的地址机制时，你一定会感觉为什么要这么复杂呢？实模式不是很简单吗？既然实模式简单，那么不妨想想，如果使用实模式会遇到什么问题呢？把这些问题都一一列出来，再结合现有机制，你就会很自然的理解为什么需要这么做了。本章将试图让读者看到这些概念背后的“为什么”。

1.1　保护模式

1.1.1　分页机制

内存按字节编址，每个地址对应一个字节的存储单元，早期的程序直接使用物理地址。在单任务操作系统时代，物理内存被划分为两部分，一部分地址空间由操作系统使用，另外一部分由应用程序使用。到了多任务时代，由于程序中的全局变量，起始加载地址是在链接期决定的[1]。如果直接使用物理地址，则很可能有多个起始地址一致的应用程序需要同时被加载运行，这就需要把冲突的程序加载到另外的地址上去，然后重新修正程序中的所有相关的全局符号的地址。然而在早期的计算机系统上内存容量十分有限，即便是通过重定

[1]不能理解这句话的读者需要补充程序编译、链接以及加载方面的知识，可以通过学习 Windows 平台下的 PE 文件格式，或者 Linux 下的 ELF 文件格式，来补充这部分知识，《Windows 环境下 32 位汇编语言程序设计》以及《Linkers & Loaders》是两本不错的教材。

位解决了加载地址冲突的问题，由于内存大小的限制，能够同时加载运行的程序仍然十分有限。而在多任务系统上，某些进程在部分时间内处于等待状态，于是人们很自然地想到，当内存不够的时候把处于等待状态的进程换入磁盘，腾出一些内存空间来加载新程序。这又带来了新的问题：每次腾出来的空间地址可不是固定不变的，这就意味着把磁盘上的内容加载进来的时候，又要重新修正程序中的相关地址，在每次换入换出的过程中要不断地修正相关程序的地址。而且还有一个更为严重的问题：假设进程A出现一个错误，对某个物理地址进行了写入操作，恰好这个地址又属于进程B，当进程B被调度运行的时候，必然会出现错误。很难想象一个软件产品的BUG却导致用户抱怨另外一个软件产品。于是虚拟内存技术发展起来。在虚拟内存中，程序代码中访问的不再是物理地址，而是虚拟地址。

以32位系统为例，每一个进程有4GB的虚拟地址空间，每个进程中有一个表，它记录着每个虚拟地址对应的物理地址是多少，这样当程序加载的时候，可以先分配好物理内存，然后把物理内存的地址填入这个表里面，这样进程之间互不影响。假设程序A和B都是要求在地址BASE处加载(程序中使用的都是虚拟地址。)，由于每个进程都有4GB的私有虚拟地址空间，因此两个进程没有加载冲突。操作系统分配的物理地址分别是A1和B1，然后A1和B1起始的物理内存地址分别被填入两个进程虚拟地址映射表中，从而建立虚拟地址和物理地址的一一映射关系。当进程A访问虚拟地址BASE+X的时候，由于MMU的硬件支持，硬件自动查找进程A的地址映射表，从而访问到物理地址为A1+X的内存单元。同理，当进程B访问虚拟地址为BASE+X的时候，MMU自动查找进程B的地址映射表，从而访问到B1+X的内存单元。当然，实际上的虚拟地址机制比这个复杂得多，但是在对它又了总体认识之后，再来学习一个实际的例子就要简单得多了。接下来就以32位的X86系统为例，进一步介绍虚拟内存机制。

每个进程拥有4GB的虚拟地址空间，每个字节的虚拟地址可以通过地址映射表映射到一个字节的物理地址上面去。因此这个映射表本身必然要占据很大的内存空间，如何设计映射表成为问题的关键。如果在虚拟地址映射表中为每一个字节建立映射关系，那么映射4GB的虚拟地址需要$2^{30} \times 4B$(32位系统地址为4Byte)的内存。可见简单的一一填表映射是不能满足现实要求的。为了要减少虚拟地址映射表项占用的内存空间，所有操作系统都采用了页式管理。把物理内存划分为4KB，8KB或者16KB大小的页，这样每个页面在虚拟地址映射表中仅仅占用4Byte的内存。以4KB的页大小为例，4GB的虚拟地址空间有2^{20}个页面，那么映射4GB空间的映射表仅仅需要$2^{20} \times 4B$(32系统地址为4Byte)的内存。

其映射原理如图1.1所示：程序要访问的地址是0x12345A10，CPU中的MMU首先找到这个进程的虚拟地址映射表，其起始物理地址为0x10000000。在4KB页大小的情况下，4GB虚拟地址空间含有2^{20}个页面，只需要20位就可以表示2^{20}的大小了，所以虚拟地址的高20位0x12345作为虚地址映射表中的索引，在32位系统上虚地址映射表中的每一项是4个字节，所以MMU根据地址0x10000000 + 0x12345*4取得虚拟地址0x12345A10对应的物理页面起始地址为0x54321000，该地址的低12位总是为0，这是由于每一个4KB大小的物理页面总是在4KB的边界上对齐的。而虚地址0x12345A10中的低12位被用做

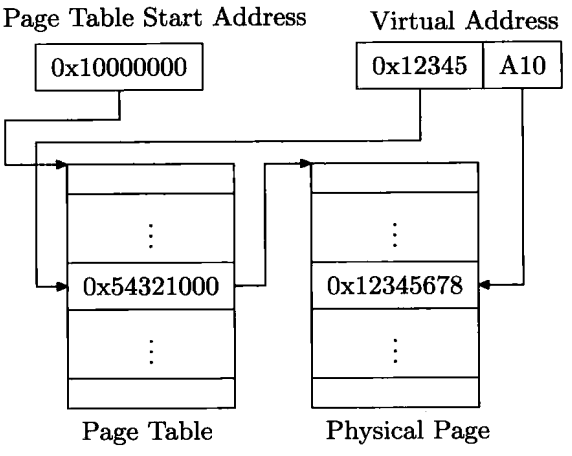


图 1.1 虚拟地址映射

页内偏移量，最终虚地址 0x12345A10 对应的物理地址为 0x54321000+A10，而 CPU 访问到的内容是 0x12345678。

由于页大小为 4KB，虚地址表中的表项低 12 位总是为 0，因此可以把低 12 位用来做标识位。例如把第 0 位用做存在位，当第 0 位为 1 时表示该页面在物理内存中，反之表示该页面不在物理内存中。假设一个进程要占用 10MB 的内存空间，在进程初始化的时候，虚地址映射表初始化为 0，在内存不足的情况下，系统只分配了 5MB 的内存，这 5MB 内存的物理地址被填入到映射表中，同时表项中最低位被设置成 1，当进程访问到另外 5MB 的虚地址的时候，MMU 在查表时发现最低位为 0，于是触发一个缺页中断，这个时候，系统缺页中断处理例程再分配内存页面，同时更新相应的映射表项，之后程序就可以正常运行了。

同理，还可以把一位划分出来作为读写位，如果对一个地址进行写入操作，MMU 在查表的时候会根据其读写位判断是否允许写入。几乎每一个程序员都知道访问 NULL 指针时，一定会出错，那么操作系统是如何捕捉到这个错误的呢？地址 0(NULL)实实在在地对应了内存中的一个物理内存地址，如何根据一个地址来判断指针是不是合法的呢？各个操作系统都保证在一个进程中虚拟地址从 0 开始的某一段区域是不映射的，其页表项为 0。例如 Windows 中把 0~64K 的地址区域划分到 NULL 指针区而不被映射。因此访问这部分地址的时候必然会触发缺页中断，这个时候操作系统就可以判断出这个地址是否落在 NULL 指针区内。否则无论像 malloc 这一类的函数返回的指针是 0 还是其他的值，都无法判断分配成功或是失败。

另外 Windows，Linux 等操作系统都有一个文件映射技术，它可以把一个大小远超出系统物理内存的文件映射到进程的虚地址空间 X 起始处，之后程序访问通过数组的方式，X[*0*]，X[*1*]....X[*n*]来访问文件的第*0*到*n*个字节。由于物理内存小于文件大小，所以内核只读入一部分的文件内容，并建立映射，当访问到没有被映射的部分 X[*m*]的时候就会触发缺

页中断，这个时候中断处理例程会再分配一定的物理内存页面，然后把它映射到 X[*m*]上，同时从磁盘读入文件对应的内容到该处。这个过程对程序来说是透明的，程序根本不用关心系统物理内存到底有多大。还可以通过把同一物理页面映射到不同的进程的虚拟地址空间中去来实现内存共享，无论各个进程映射的虚拟地址是相同还是不同的，访问到的都是同一个物理页面。在操作系统中有一个重要的 Copy-On-Write 机制，多个进程共享同一片内存，这片内存的读写位被设置成 0，当某个进程对其写入的时候，触发中断，在中断处理程序中，把要写入的相关页面复制一份，之后该进程单独使用这些内存，而进程间仍然共享其他的内存。通过这些例子读者可以看到虚拟内存、虚拟地址、进程的虚地址空间及 MMU 之间的区别和联系。

虚地址到物理的转换过程是由硬件自动完成的。由于虚地址映射表是进程私有的，因此各个进程的虚地址映射表被放在不同的物理内存中，而且每个进程都必须把这个表的起始物理地址告诉 MMU，这就是 Page Table Start Address 的作用，很容易想到这个起始地址必须是物理地址。前面说过虚地址映射表大小为 4MB，而虚地址的高 20 位是这个表中的索引，这意味着这 4MB 空间必须作为进程的必备资源在启动的时候一次分配，而且这 4MB 的内存必须在物理地址上是连续的。这可不是一个好消息，想想虚拟内存的设计理念就是要在一个小内存系统上运行尽可能多的程序，而这个限制将导致一个配备 64M[2]内存的系统也运行不了几个进程。考虑到一个进程不需要同时访问到 4GB 的内存，因此映射表也可以被分散开来，像物理页面那样在需要的时候再分配映射。于是两级，三级甚至四级[3]页表的概念被提出来。

图1.2是 32 位 X86 系统的两级页表结构。地址映射表分为两级，第一级被称为页目录表，第二级为页表，每个进程的页目录表起始物理地址由 CPU 中的寄存器 CR3 指定，而虚拟地址的最高 10 位将作为页目录的索引，CR3+(页目录索引 × 4)就可以得到 PDE[4] 。PDE 的高 20 位指定了一个页表的起始地址，低 12 位是一些标致位，同时虚拟地址的中间 10 位被用做页表的索引，从而得到 PTE[5]。PTE 的高 20 位指定了一个 4KB 页面的起始地址，而虚拟地址的最低 12 位被用做页内偏移量，从而访问到虚拟地址指定的内存单元。最高 10 位用做页目录的索引，每一项 4 个字节，所以页目录大小正好为 4KB，1024 项，每一项指向一个页表，每个页表又有 1024 项指向对应的 4KB 页，总共可以映射的内存就是 1024 × 1024 × 4KB，也就是 4GB。这样的话，一个进程的页表可以被分散开来，在页目录索引时如果发现页表没有被映射，就可以通过缺页中断来分配并建立新的映射。

两级页表虽然能解决进程一次要连续分配 4MB 页表的问题，但是每次访问内存都需要两次查表才得到物理地址最后访问到指定的内存，这样一来降低了系统内存的访问速度，为此 CPU 内部设置了最近存取的页面的缓存，被称为 TLB，程序给出虚拟地址后 CPU 先到 TLB 中查找，如果 TLB 没有命中再访问两级页表，从而极大地提高了访问速度。

[2]在早期配备 64MB 内存已经算是大内存系统了。
[3]三级，四级页表一般被应用在 64 位系统上。
[4]即 Page Directory Entry,页目录项。
[5]即 Page Table Entry,页表项。

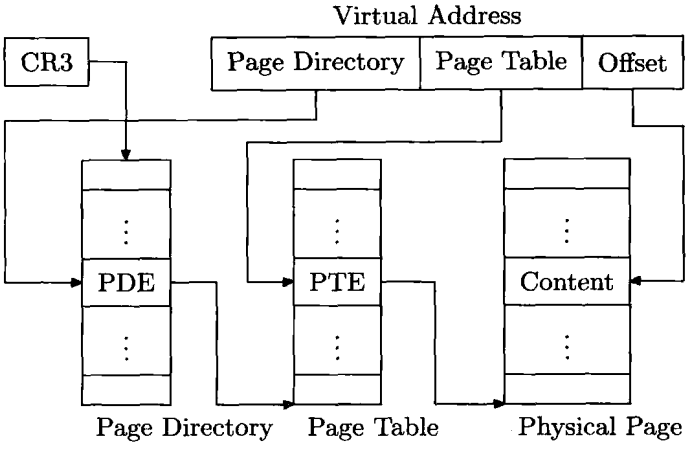


图 1.2 32 位 X86 系统两级页表结构

通过前面的讨论可以看到，每一个进程都有独立的页表，进程总是通过查本进程页表获取物理地址的，因此无法直接修改其他进程的内存。这样一来，进程就被保护起来而不受其他进程的影响了。这是保护模式的一个重要特征，但是这样的保护还是不够的。我们知道几乎每一个进程都要通过操作系统提供的接口执行一些操作系统内核提供的代码。例如进程通过 read 系统调用读取文件内容，而具体读取文件的代码则是操作系统内核提供的，同时内核的这些代码也需要使用一些数据，这部分代码和数据必须映射到每一个进程的地址空间中，但是如果任何一个进程有意或无意地修改了这部分代码或数据的话，那么其后果是严重的。

一种由硬件支持的进程权限级别就是用来解决这个问题的，多数构架的 CPU 提供 4 个级别，0 代表最高级别，3 则是最低级别。然而多数操作系统的设计只使用了其中两个级别，内核运行在级别 0 上，而应用程序运行在级别 3 上，人们通常把它们称做 ring0 和 ring3，当 ring3 的代码试图访问 ring0 的代码或者数据的时候，硬件自动进行权限检查，只有通过检查的才能访问成功。于是进程有两个执行环境，应用层和内核层，应用层程序在 ring3 执行程序自己的代码，而只能通过特殊的途径[6]进入 ring0 执行内核代码。有了这个保护，ring3 的代码无法直接修改 ring0 的代码和数据，而进程进入 ring0 时执行的那些代码都是内核提供的，内核保证这部分代码不会有错误，也不会恶意修改数据或代码，从而保证了系统的健壮性。除此之外，CPU 的指令也被分类，某些特权指令只能在 ring0 的级别上执行。这样保证某些重要的资源不能被应用程序随意修改，例如中断向量表的起始地址。分页保护和权限保护机制被称为保护模式，而之前的 8086 不具备这两种机制，被称为实模式。

图1.3是 X86 页表，页目录中的一些常用标志位，操作系统通过设置这些标志位来控制硬件实现前面介绍的保护模式的功能。其中页表项和页目录项大致相同。更加详细的信息请参看《Intel Architecture Software Developer's Manual Volume 3: System Programming》。

[6]如系统调用。