<<独辟蹊径品内核>>

图书基本信息

书名：<<独辟蹊径品内核>>

13位ISBN编号：9787121085154

10位ISBN编号：7121085151

出版时间：2009-08-01

出版时间：电子工业出版社

作者：李云华

页数：482

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，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。

<<独辟蹊径品内核>>

内容概要

《独辟蹊径品内核：Linux内核源代码导读》根据最新的2.6.24内核为基础。
在讲述方式上，《独辟蹊径品内核：Linux内核源代码导读》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《独辟蹊径品内核：Linux内核源代码导读》的写作宗旨：“授人以渔”。
在内容安排上，《独辟蹊径品内核：Linux内核源代码导读》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。
《独辟蹊径品内核：Linux内核源代码导读》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。

作者简介

李云华，是一名内核技术的狂热爱好者，长期从事操作系统内核、计算机网络、设备驱动程序、以及嵌入系统方面的开发和研究。
拥有丰富的设备驱动开发、网络优化、内核及驱动移植、嵌入式系统构建等方面的开发经验。
对Windows内核驱动及Linux内核驱动均有丰富的开发经验及心得体会。

<<独辟蹊径品内核>>

书籍目录

章节摘录

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。
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230×4B（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220个页面，那么映射4GB空间的映射表仅仅需要220×4B（32系统地址为4Bvte）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220个页面，只需要20位就可以表示220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+0X123454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Oxl2345A10中的低12位被用伽页内偏移量，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。

<<独辟蹊径品内核>>

编辑推荐

较新的内核版本：本书使用的内核版本为2.6.24。
独特的写作手法：本书在讨论“ How ”的基础上，力求进一步探究“ Why ”。
“ 授人以渔 ”的写作宗旨：Linux内核处于飞速发展中，任何资料都无法覆盖内核的方方面面。配收以笔者学习过程的疑问和经验为基础，融会贯通于各个章节，毫无保留地就如何学习内核，如何分析内核进行了大量且大胆的探讨，从而力求体现本书的写作宗旨——“ 授人以渔 ”。

版权说明