一、简单介绍下早期的内存实现：（可略过）

        1、在早期的计算机中，运行一个程序的特点是：

            （1）会把这些程序全都装入内存，

            （2）程序都是直接运行在内存上的，也就是说程序中访问的内存地址都是实际的物理内存地址。

        2、在早起的内存实现方式中出现的问题：

            （1）当计算 机同时运行多个程序时，必须保证这些程序用到的内存总量要小于计算机实际物理内存的大小。

            （2）进程地址空间不隔离。进程间可以相互修改数据

            （3）内存使用率低。运行一个进程，必须在内存中为它分配实际进程大小的空间（不管当前这些空间是否都会用到）

            （4）程序运行的地址不确定。分配内存时，只是单纯的从内存空间找一个足够满足进程要求的空间，所以进程运行的地址具有随机性。

二、虚拟地址技术

      为了解决上面的问题，一些大牛们就提出了传说中的虚地址技术。即：在用户进程和实际的物理内存之间加一个中间层（虚拟地址），让用户进程只能访问虚拟地址，并且把虚拟地址物理地址转换的实现交给[**操作系统**](http://lib.csdn.net/base/operatingsystem)。

      采用虚拟地址的特点：

        1、当创建进程时，

            （1）OS会为每个进程分配一个4G大小的虚拟地址空间，且每个进程都有3G的用户空间和1G的内核空间。但最后这1G的内核空间中的内容对于不同进程来说是一样的（之所以是4G大小，是因为在32位操作系统中，一个指针的大小是4btyes，所以能访问的地址空间就是0x00000000~0xFFFFFFFF == 4G）。

            （2）这4G的虚拟地址空间是操作系统虚拟出来的，并不是真实存在的。

            （3）每个进程只能访问自己的虚拟地址，无法访问别的进程的虚拟地址。

            （4）虽然每个进程的虚拟地址都有4G大小，但并不是这4G的地址都可以由用户任意使用。在**[Linux](http://lib.csdn.net/base/linux" \o "Linux知识库" \t "_blank)**中，系统把虚拟地址划分成2个部分（如下图所示）：

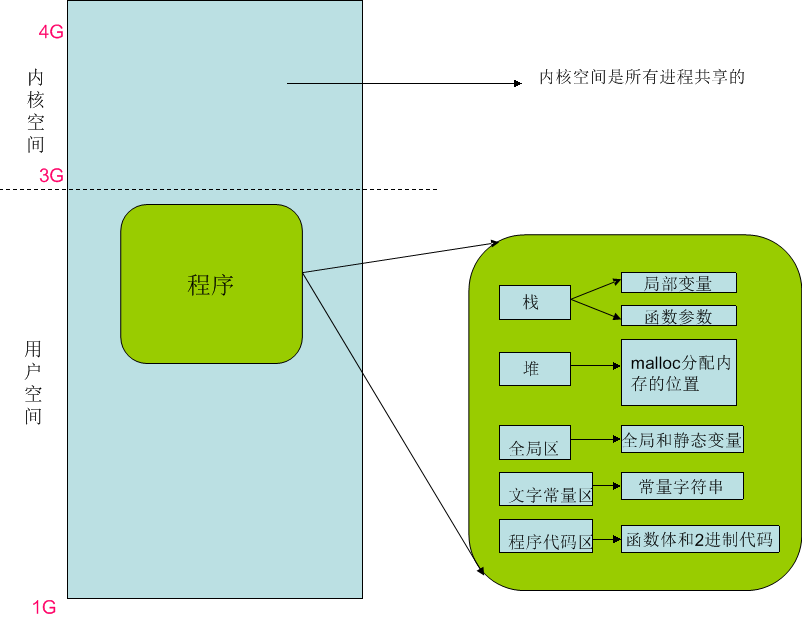
注：1、任意一个时刻，在一个CPU上只有一个进程在运行。所以对于此CPU来讲，在这一时刻，整个系统只存在一个4GB的虚拟地址空间，这个虚拟地址空间是面向此进程的。

       当进程发生切换的时候，虚拟地址空间也随着切换。每个进程只有在运行的时候，其虚拟地址空间才被运行它的CPU所知。在其它时刻，其虚拟地址空间对于CPU来说，是不可知的。所以尽管每个进程都可以有4 GB的虚拟地址空间，但在CPU眼中，只有一个虚拟地址空间存在。虚拟地址空间的变化，随着进程切换而变化。

       2、内核空间与物理内存空间的映射和用户空间与物理内存空间的映射方式是不同：

            虽然内核空间占据了每个虚拟空间中的最高1GB字节，但映射到物理内存却总是从最低地址（0x00000000）开始的，之所以这么规定，是为了在内核空间与物理内存之间建立简单的线性映射关系。其中，3GB（0xC0000000）就是物理地址与虚拟地址之间的位移量，在[**linux**](http://lib.csdn.net/base/linux)代码中就叫做PAGE\_OFFSET。  
                  
我们来看一下在include/asm/i386/page.h头文件中对内核空间中地址映射的说明及定义：  
  
#define \_\_PAGE\_OFFSET           (0xC0000000)  
……  
#define PAGE\_OFFSET             ((unsigned long)\_\_PAGE\_OFFSET)  
#define \_\_pa(x)                 ((unsigned long)(x)-PAGE\_OFFSET)  
#define \_\_va(x)                 ((void \*)((unsigned long)(x)+PAGE\_OFFSET))  
对于内核空间而言，给定一个虚地址x，其物理地址为“x- PAGE\_OFFSET”，给定一个物理地址x，其虚地址为“x+ PAGE\_OFFSET”。  
  
这里再次说明，宏\_\_pa()仅仅把一个内核空间的虚地址映射到物理地址，

             这种方法决不适用于用户空间，用户空间的地址映射要复杂得多，它通过分页机制完成。



（图一）4G虚拟地址空间的整体结构



（图二）内核空间分配图



（图三，地址分布图）

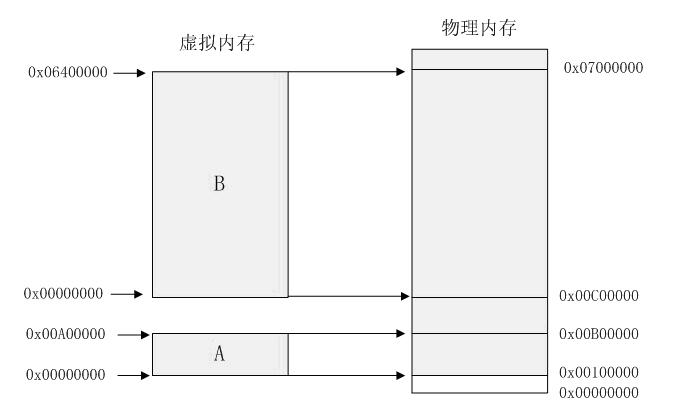
        2、与虚拟地址相对的就是我们内存条实际的大小了。以本人PC为例：我用的是2G内存，那么实际的物理地址空间就是：0x00000000~0x1FFFFFFF

三、分段和分页

          1、采用虚拟地址技术后出现的新问题是：虚拟地址和物理地址之间的映射方式。大牛们提出的解决方式就是分段、分页和段页式了。具体实现我们用一个例子来演示:

        eg:假设有两个进程 A 和 B ，进程 A 所需内存大小为 10M ，其虚拟地址空间分布在 0x00000000 到 0x00A00000 ，进程 B 所需内存为 100M ，其虚拟地址空间分布为 0x00000000 到 0x06400000 。

       （1）采用分段的方式



       这种分段的映射方法虽然解决了上述中的问题（2）和问题（4），但并没能解决问题（1）（3），即内存的使用效率问题。在分段的映射方法中，每次换入换出内存的都是整个程序， 这样会造成大量的磁盘访问操作，导致效率低下。所以这种映射方法还是稍显粗糙，粒度比较大

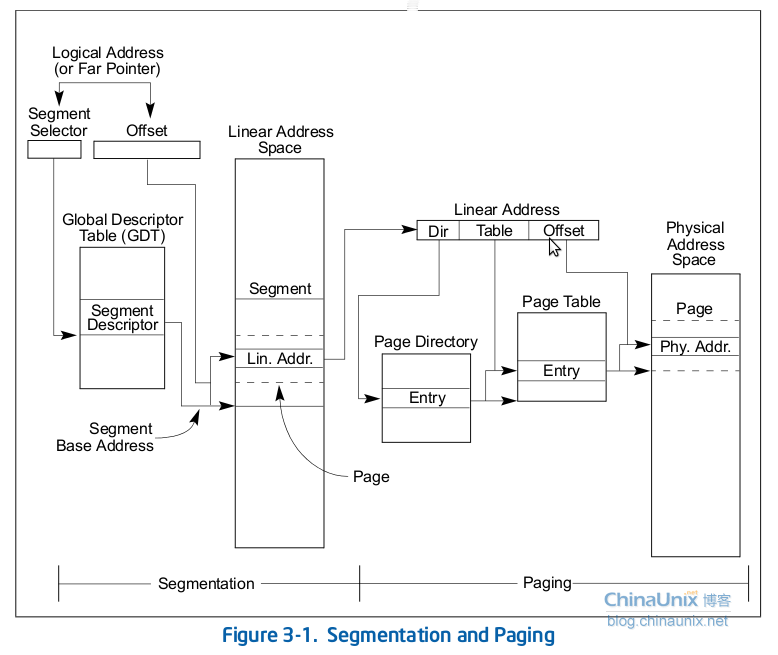
       （2）采用分页式

              分页的基本方法是，将地址空间分成许多的页。每页的大小由 CPU 决定，然后由操作系统选择页的大小。目前 Inter 系列的 CPU 支持 4KB 或 4MB 的页大小，而 PC 上目前都选择使用 4KB 。按这种选择， 4GB 虚拟地址空间共可以分成 1048576 个页， 512M 的物理内存可以分为 131072 个页。

              分页的思想是程序运行时用到哪页就为哪页分配内存，没用到的页暂时保留在 硬盘上。

四、这里对分段、分页只做了一个简单笼统的介绍，如果想更深入的理解底层的实现机制，请看下一篇

Linux中的虚拟地址、物理地址和内存管理方式（二）

我们都知道，动态共享库里面的函数的共享的，这也是动态库的优势所在，就是节省内存。C 编译出来的可执行文件几乎都会用到libc的库，假如没有这个共享的技术，每个可执行文件都要占一份libc库的内存，这将是极大的内存浪费。 可是一直没搞明白，怎么样才能证明共享库里面函数的地址在物理内存层面是同一份。其实，这个问题的本质是程序里面的逻辑地址和物理内存地址之间是怎样映射的，说的再赤裸裸一点，就是我给你个逻辑地址，请你在物理内存中找到对应的地址，或者我给你个物理地址，请你把这个物理内存里面存的东西告诉我。  
  
    最近两天，发现了一篇很牛的博文，这个博文彻底解决了逻辑地址 线性地址 物理地址的内存映射问题，作者的功力特别身后，他十分kind的提供了一篇29页的pdf文档，此文章一出，就彻底终结这个问题了。那我为什么还要写这篇博文呢。作者以2.6.18内核为例，提供了两个内核模块和两个应用层的程序，我在自己的Ubuntu 12.04上花了时间完整的验证了文档里面PAE（Physical Address Extension）模式的地址映射，发现代码里面存在一些兼容性的问题，导致编译不过，主要是内核版本不同和gcc带来的一些小问题。所以我花了4个多小时才把这个实验完整的做下来。如果想通过做实验来加深理解的筒子可以参考我修改后的程序。我无意抄袭，还是那句话，光荣属于前辈。  
  
    下面的图来自Intel的手册64-ia-32-architectures-software-developer-vol-3a-part-1-manual ，很好的解释的逻辑地址到物理地址的映射。所谓逻辑地址，就是我们C 语言中取地址符后，看到的地址。  
    [](http://blog.chinaunix.net/attachment/201212/1/24774106_1354368480Oo5W.png)  
    采用原文的函数

1. #include <stdio.h>
2. int main()
3. {
4. unsigned long tmp;
5. tmp = 0x12345678;
6. printf("tmp address:0x%08lX\n", &tmp);
7. return 0;
8. }
9. tmp address:0xBF86D16C

    输出的地址为0xBF86D16C,这个就是官方手册上说的逻辑地址。首先需要将逻辑地址转化成线性地址。然后将线性地址转化成物理地址。将逻辑地址转化成线性地址，就是江湖传说的分段机制，也就是上图下面的segmentation。  
  
    1 段式映射  
    临时变量tmp的逻辑地址0xBF86D16C就是偏移量，因为tmp位于栈中，IA-32提供了SS（Stack Segment）寄存器。

1. //arch/x86/kernel/process\_32.c
2. //-------------------------------------------
3. void
4. start\_thread(struct pt\_regs \*regs, unsigned long new\_ip, unsigned long new\_sp)
5. {
6. set\_user\_gs(regs, 0);
7. regs->fs = 0;
8. regs->ds = \_\_USER\_DS;
9. regs->es = \_\_USER\_DS;
10. regs->ss = \_\_USER\_DS;
11. regs->cs = \_\_USER\_CS;
12. regs->ip = new\_ip;
13. regs->sp = new\_sp;
14. /\*
15. \* Free the old FP and other extended state
16. \*/
17. free\_thread\_xstate(current);
18. }

    实际上有6个段寄存器，但是DS,ES ，SS的值是一样的，FS和GS都是0，这样其实6个段寄存器本质是两个：CS和DS。每个进程的6个寄存器是一样的，不同的是EIP和ESP。从上面的代码中也可以看到。

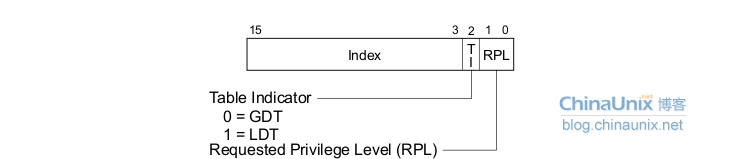
1. arch/x86/include/asm/segment.h
2. ------------------------------------------
3. #define GDT\_ENTRY\_DEFAULT\_USER\_CS    14
4. #define GDT\_ENTRY\_DEFAULT\_USER\_DS    15
5. #define GDT\_ENTRY\_KERNEL\_BASE        (12)
6. #define GDT\_ENTRY\_KERNEL\_CS        (GDT\_ENTRY\_KERNEL\_BASE+0)
7. #define GDT\_ENTRY\_KERNEL\_DS        (GDT\_ENTRY\_KERNEL\_BASE+1)
8. #define \_\_KERNEL\_CS    (GDT\_ENTRY\_KERNEL\_CS\*8)
9. #define \_\_KERNEL\_DS    (GDT\_ENTRY\_KERNEL\_DS\*8)
10. #define \_\_USER\_DS    (GDT\_ENTRY\_DEFAULT\_USER\_DS\*8+3)
11. #define \_\_USER\_CS    (GDT\_ENTRY\_DEFAULT\_USER\_CS\*8+3)

    \_\_USER\_CS(14\*8 +3 = 115)的值展开二进制的结果为:

1. 0000000001110 011

    \_\_USER\_DS(15\*8 + 3 =123)的值展开二进制的结果为:

1. 0000000001111 011

   上面的两组数字就是段选择符，段选择符有16位，其中含义如下图：  
  
[](http://blog.chinaunix.net/attachment/201212/1/24774106_1354371395j6xe.png)  
  
  
    TI表示我要选择的段描述符是存在GDT中还是LDT中。GDT和LDT可以简单理解成两个表，每个表里面都存放这一组地址。  
  
    我们的CS和DS对应的TI位都是0，换句话说，我们要着的段描述符在GDT中。实际上，我们的Linux程序里用的段描述符总是选择GDT，几乎没有选择LDT的。毛德操老爷子说，只有像wine这种进程才会用到LDT这样的东西。  
  
    RPL表示特权等级，0表示最高权限，3表示无特权。之所以在

1. #define \_\_USER\_CS (GDT\_ENTRY\_DEFAULT\_USER\_CS\*8+3)

   有个+3，就是表示，我的段无特权，同时我的段描述符存在GDT这张表里面。前面的13位表示是GDT表的index，或者说是第几项。   
  
    接下来就是去GDT这张表，去找到我们要的段描述符。等等，我们一直很爽的叫着GDT，知道我们的DS段描述符是在index =15的位置，可是从来没有人告诉我们GDT这张表放在哪里。  
  
    GDTR横空出世了，GDT的地址就存放在GDTR这个寄存器里面。问题是怎么读出啦GDTR寄存器的值？  
  
    前面提到的博文作者写了一个内核模块，来提取GDTR，CR0 CR3  等的值，主干代码在下面：

1. static int my\_get\_info( char \*buf, char \*\*start, off\_t off, int count )
2. {
3. int len = 0;
4. struct mm\_struct \*mm;
5. mm = current->active\_mm;
6. cr0 = read\_cr0();
7. cr3 = read\_cr3();
8. cr4 = read\_cr4();
9. //asm(" sgdt gdtr");
10. asm("sgdt %0":"=m"(gdtr));
11. len += sprintf( buf+len, "cr4=%08X ", cr4 );
12. len += sprintf( buf+len, "PSE=%X ", (cr4>>4)&1 );
13. len += sprintf( buf+len, "PAE=%X ", (cr4>>5)&1 );
14. len += sprintf( buf+len, "\n" );
15. len += sprintf( buf+len, "cr3=%08X cr0=%08X\n",cr3,cr0);
16. len += sprintf( buf+len, "pgd:0x%08X\n",(unsigned int)mm->pgd);
17. len += sprintf( buf+len, "gdtr address:%lX, limit:%X\n", gdtr.address,gdtr.limit);
18. // len += sprintf( buf+len, "cpu\_gdt\_table address:0x%08lX\n", cpu\_gdt\_table);
19. return len;
20. }

    asm那句代码在我的gcc下编译不过，我修改了下。感兴趣的同学可以考虑下为啥编译不过。  
      
    总之我们有办法取GDTR寄存器的值，从而找到了GDT这张表，然后从这张表里面着第16项（index=15），我们就能找到我们的DS段描述符。

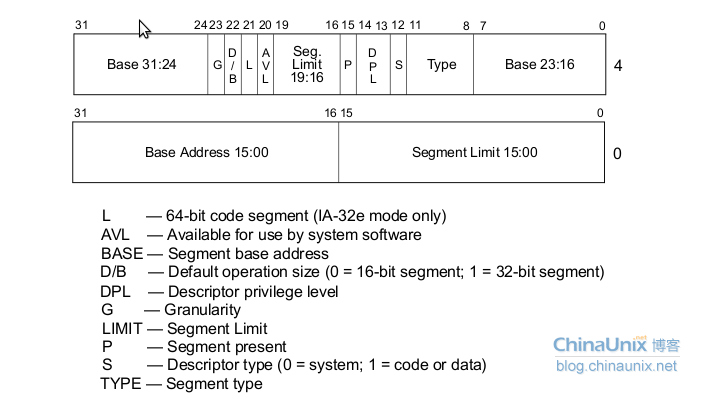
1. root@manu:~/code/c/self/mm\_addr# ./mem\_map
2. %ebp:0xBF86D178
3. tmp address:0xBF86D16C
4. cr4=000006F0 PSE=1 PAE=1
5. cr3=06E3C000 cr0=8005003B
6. pgd:0xC6E3C000
7. gdtr address:F7BB9000, limit:FF

    国外大牛提供了一个叫做dram的内核模块，还有一个fileview的tool，这个tool+模块相互配合，能够读到物理地址里面对应的内容。这个内核模块是大杀器啊，我解决共享库迷惑就全靠在这个内核模块上了。作者是低于2.6.32的内核，我们是高于2.6.32的内核，所以稍加修改，就能用在我的Ubuntu上了。  
  
    可以算出GDT的地址为F7BB9000 - c0000000，然后用作者提供的工具fileview去看下内存内容

1. -----------------------------------------------------------
2. gdtr : f7bb9000 - c0000000 = 37bb9000
3. 0000037BB9000 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ................
4. 0000037BB9010 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ................
5. 0000037BB9020 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ................
6. 0000037BB9030 FF FF 00 B9 61 F3 DF B7 00 00 00 00 00 00 00 00 ....a...........
7. 0000037BB9040 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ................
8. 0000037BB9050 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ................
9. 0000037BB9060 FF FF 00 00 00 9B CF 00 FF FF 00 00 00 93 CF 00 ................
10. 0000037BB9070 FF FF 00 00 00 FB CF 00 FF FF 00 00 00 F3 CF 00 ................
11. 0000037BB9080 6B 20 C0 EA BB 8B 00 F7 00 00 00 00 00 00 00 00 k ..............
12. 0000037BB9090 FF FF 00 00 00 9A 40 00 FF FF 00 00 00 9A 00 00 ......@.........
13. 0000037BB90A0 FF FF 00 00 00 92 00 00 00 00 00 00 00 92 00 00 ................
14. 0000037BB90B0 00 00 00 00 00 92 00 00 FF FF 00 00 00 9A 40 00 ..............@.
15. 0000037BB90C0 FF FF 00 00 00 9A 00 00 FF FF 00 00 00 92 40 00 ..............@.
16. 0000037BB90D0 FF FF 00 00 00 92 CF 00 FF FF 00 40 29 93 8F 36 ...........@)..6
17. 0000037BB90E0 18 00 80 0C BC 91 40 F7 00 00 00 00 00 00 00 00 ......@.........
18. 0000037BB90F0 00 00 00 00 00 00 00 00 6B 20 00 48 80 89 00 C1 ........k .H....

OK ，我们取到了我们的DS段描述符：

1. FF FF 00 00 00 F3 CF 00 = 00cff300 0000ffff

[](http://blog.chinaunix.net/attachment/201212/1/24774106_1354373240az9W.png)  
  
    自己对照就能的出，BASE=0x00000000，费了半点的劲，最后的得出：  
    分段机制是fake的，虚拟地址总是能线性地址。（本来分段机制下虚拟地址表示方式是：段的基址+段内偏移量，现在段的基地址永远是0，所以虚拟地址就永远等于段内偏移量）  
      
    我们还可以得到其他有用的信息：

1. S=1 非系统段
2. G=1 以4096为单位
3. DPL=0x11，内核态用户态均可访问

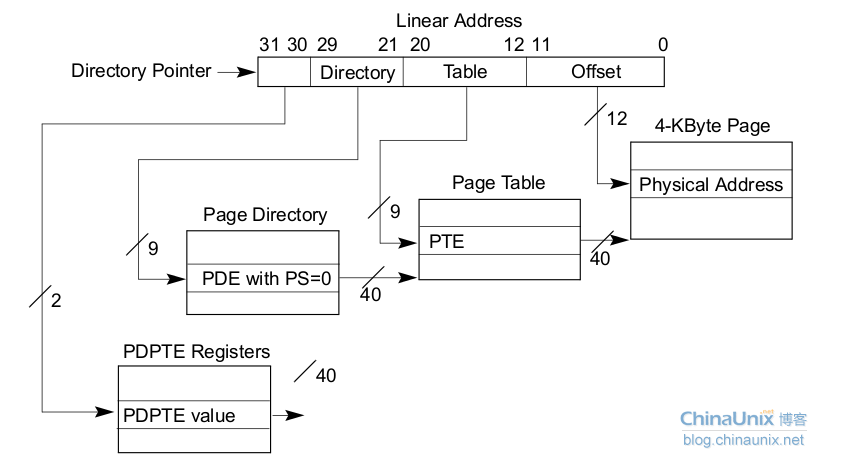
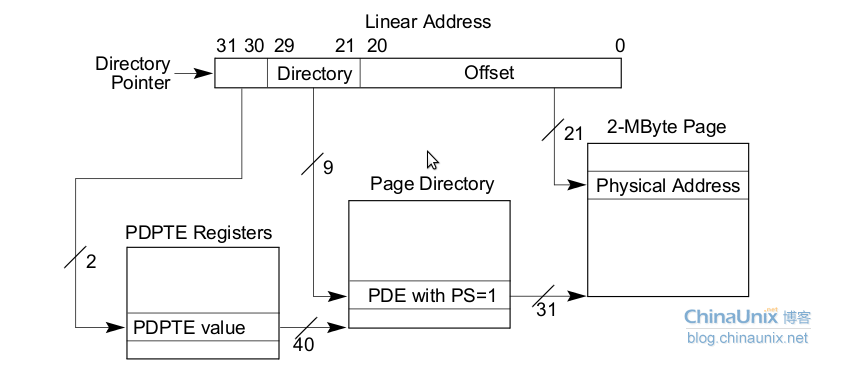
    所以经过这么一番折腾，最终的结果是，虚拟地址总是等于线性地址。以后就不要再折腾了。  
  
2  页式映射。  
   有了线性地址，下一步就是获取物理地址了。  
  
    我的电脑采用了PAE，物理地址扩展分页机制，看下我的uname -ar

1. uname -r
2. 3.2.0-29-generic-pae

    上面程序也正PAE=1也证明了我的的确确的采用了PAE. PAE要比常规分页稍稍复杂一点。   
  
    先讲讲啥是PAE。 目前的服务器基本都突破了4G的内存，很多PC都已经突破4G 了，我有同事就有16G 内存的PC，让我羡慕的直流口水。Intel通过把管脚从32增加到36，可以支持64G内存，但是，必须引入一种新的分页机制，把32位的线性地址转化成36位的物理地址，才能充分利用这64G的内存。  
  
    这个机制就是PAE ：   
    1 引入一个页目录指针表PDPT，有4个64位的item组成。  
    2 cr3寄存器中27位用来表示 页目录指针表PDPT的地址(32字节对齐，所以不需要32来表示)。 

    3 线性地址的高2位决定4个PDPT item的的哪一个。

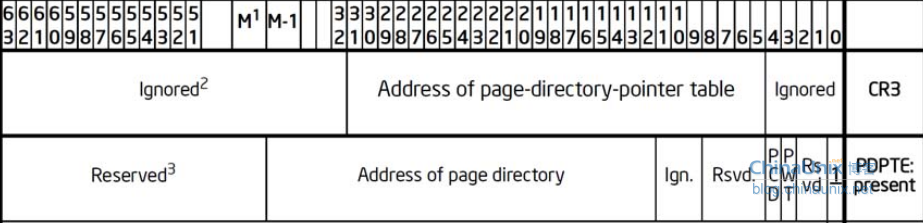
PS. 解释下CR0~CR4寄存器：  
控制寄存器（CR0～CR4）用于控制和确定处理器的操作模式以及当前执行任务的特性，如图4-3所示。CR0中含有控制处理器操作模式和状态的系统控制标志；CR1保留不用；CR2含有导致页错误的线性地址；CR3中含有页目录表物理内存基地址，因此该寄存器也被称为页目录基地址寄存器PDBR（Page-Directory Base address Register）。

[](http://blog.chinaunix.net/attachment/201212/2/24774106_1354416328uPM2.png)      
    上图完整的描述了PAE模式下线性地址到物理地址的映射。稍微不好懂的就是40这个数字的含义：  
     Intel手册里面有下面的句子：  
  
1）A PDE is selected using the physical address defined as follows:  
— Bits 51:12 are from PDPTEi.  
— Bits 11:3 are bits 29:21 of the linear address.  
— Bits 2:0 are 0.  
2）PDE的bit7（PS位）决定了采用4K大小的页还是2M 大小的页。如果是2M 大小的页，上面的图针对的是4K 大小的页。2M大小的页采用这种模式：  
  
[](http://blog.chinaunix.net/attachment/201212/2/24774106_1354417033LCws.png)  
    对于我们而言，我们采用的不是2M 大小的页，后面实验中我们可以看下PS位。所以这种2M的页的模式，后面我们就不讲了。  
  
3）A PTE is selected using the physical address defined as follows:  
— Bits 51:12 are from the PDE.        
— Bits 11:3 are bits 20:12 of the linear address.  
— Bits 2:0 are 0.  
4）获取最后的物理地址  
— Bits 51:12 are from the PTE.  
— Bits 11:0 are from the original linear address.  
  
  
    OK 回到我们的例子：  
    

1. 线性地址：
2. 0xBF86D16C
3. 0x 10          111111 100          0 0110 1101           0001 0110 1100

 高2位是10，表示选择index =2 的那个PDPT item

1. root@manu:~/code/c/self/mm\_addr# ./mem\_map
2. %ebp:0xBF86D178
3. tmp address:0xBF86D16C
4. cr4=000006F0 PSE=1 PAE=1
5. cr3=06E3C000 cr0=8005003B
6. pgd:0xC6E3C000
7. gdtr address:F7BB9000, limit:FF

CR3的值是0x06E3C000 ，注意下图，后5位ignore，所以真正的地址还是0x6E3C000.[](http://blog.chinaunix.net/attachment/201212/1/24774106_1354375698WmBb.png)  
  
 我们看下0x6E3C000地址下存放的啥东西，再次祭出我们的dram神器：

1. 0000006E3C000 01 B0 E3 06 00 00 00 00 01 60 3C 08 00 00 00 00 .........`<.....
2. 0000006E3C010 01 50 3C 08 00 00 00 00 01 40 93 01 00 00 00 00 .P<......@......
3. 0000006E3C020 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ................
4. 0000006E3C030 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ................
5. 0000006E3C040 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ................
6. 0000006E3C050 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ................
7. 0000006E3C060 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ................
8. 0000006E3C070 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ................
9. 0000006E3C080 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ................
10. 0000006E3C090 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ................
11. 0000006E3C0A0 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ................
12. 0000006E3C0B0 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ................
13. 0000006E3C0C0 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ................
14. 0000006E3C0D0 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ................
15. 0000006E3C0E0 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ................
16. 0000006E3C0F0 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ................

    蓝色的地址就是我们要找的：

1. 01 50 3C 08 00 00 00 00 = 0x083c5001,

其实这是一个64位的地址，12~51位是页目录表项的基地址。  
其中bit 0表示的是present，表示该64位地址是有效的。  
其中bit7（PS位）没有置位，表明采用的页是4K 大小的页，而不是2M大小的页。  
可以算出表项的基地址为：0x83c5000。

1. 线性地址：
2. 0xBF86D16C
3. 0x 10          111111 100          0 0110 1101           0001 0110 1100

   0x83c5000+(111111100)b \*8= 0x83c5fe0。  
  
看下这个地址下的内容：

1. 00000083C5FB0 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ................
2. 00000083C5FC0 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ................
3. 00000083C5FD0 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ................
4. 00000083C5FE0 67 70 D8 06 00 00 00 00 00 00 00 00 00 00 00 00 gp..............
5. 00000083C5FF0 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ................
6. 00000083C6000 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ................
7. 00000083C6010 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ................
8. 00000083C6020 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ................
9. 00000083C6030 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ................
10. 00000083C6040 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ................
11. 00000083C6050 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ................
12. 00000083C6060 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ................
13. 00000083C6070 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ................
14. 00000083C6080 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ................
15. 00000083C6090 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ................
16. 00000083C60A0 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 000

 取到的地址为 

1. 67 70 D8 06 00 00 00 00 = 0x6d87067

考虑到4096对齐和12~51位是有效地址， 页面目录表的地址为6d87000

1. 线性地址：
2. 0xBF86D16C
3. 0x 10         111111 100        0 0110 1101            0001 0110 1100

0x6d87000 + (001101101)b \*8  = 0x6d87368，  
看下这个地址下的内容

2. 0000006D87360 47 40 65 07 00 00 00 80 47 A0 94 0D 00 00 00 80 G@e.....G.......
3. 0000006D87370 47 B0 BD 09 00 00 00 80 00 00 00 00 00 00 00 00 G...............
4. 0000006D87380 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ................
5. 0000006D87390 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ................
6. 0000006D873A0 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ................
7. 0000006D873B0 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ................
8. 0000006D873C0 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ................
9. 0000006D873D0 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ................
10. 0000006D873E0 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ................
11. 0000006D873F0 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ................
12. 0000006D87400 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ................
13. 0000006D87410 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ................
14. 0000006D87420 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ................
15. 0000006D87430 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ................
16. 0000006D87440 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ................
17. 0000006D87450 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ................

我们终于到了最后一个页表了：

1. 47 A0 94 0D 00 00 00 80 = 0x80000000 0d94a047

最终物理地址计算  
1）  12~51位来自 0x80000000 0d94a047  
      换句话说就是：0d94a000        
2） 0 ~11来自线性地址的最后12位

1. 线性地址：
2. 0xBF86D16C
3. 0x 10         111111 100                0 0110 1101          0001 0110 1100
4. 0xd94a000 + （0001 0110 1100）b = 0x0d94a16c

OK ，最后总算得到了物理地址 0x0d94a16c.  
用我们的神器看下物理地址的内容是不是0x12345678

1. 000000D94A160 70 A2 7A B7 00 00 00 00 A9 86 04 08 78 56 34 12 p.z.........xV4.
2. 000000D94A170 A0 86 04 08 00 00 00 00 00 00 00 00 D3 14 5F B7 ..............\_.
3. 000000D94A180 01 00 00 00 14 D2 86 BF 1C D2 86 BF 58 98 79 B7 ............X.y.
4. 000000D94A190 00 00 00 00 1C D2 86 BF 1C D2 86 BF 00 00 00 00 ................
5. 000000D94A1A0 A0 82 04 08 F4 DF 77 B7 00 00 00 00 00 00 00 00 ......w.........
6. 000000D94A1B0 00 00 00 00 A9 68 DD 32 B8 4C 57 81 00 00 00 00 .....h.2.LW.....
7. 000000D94A1C0 00 00 00 00 00 00 00 00 01 00 00 00 A0 84 04 08 ................
8. 000000D94A1D0 00 00 00 00 A0 F6 7A B7 E9 13 5F B7 F4 BF 7B B7 ......z...\_...{.
9. 000000D94A1E0 01 00 00 00 A0 84 04 08 00 00 00 00 C1 84 04 08 ................
10. 000000D94A1F0 54 85 04 08 01 00 00 00 14 D2 86 BF A0 86 04 08 T...............
11. 000000D94A200 10 87 04 08 70 A2 7A B7 0C D2 86 BF 18 C9 7B B7 ....p.z.......{.
12. 000000D94A210 01 00 00 00 DF E8 86 BF 00 00 00 00 E9 E8 86 BF ................
13. 000000D94A220 F9 E8 86 BF 04 E9 86 BF 0E E9 86 BF 2F EE 86 BF ............/...
14. 000000D94A230 3E EE 86 BF 4C EE 86 BF 5A EE 86 BF 6E EE 86 BF >...L...Z...n...
15. 000000D94A240 B0 EE 86 BF D3 EE 86 BF E4 EE 86 BF EC EE 86 BF ................
16. 000000D94A250 03 EF 86 BF 13 EF 86 BF 25 EF 86 BF 32 EF 86 BF ........%...2...

看下右上角的蓝色0x12345678,那就是我们存储的tmp的值。。  
  
  
    再次感谢ilinuxkernel博主写的文档，让我解决了这个彻底解决了这个虚拟地址到物理地址的转换，我喜欢这样的文章，他让我更深刻的理解计算机的原理，这片博文绝大部分的贡献都是这位kind的博主，光荣属于前辈。  
  
    为了方便感兴趣的筒子顺利的做这个实验，我将这个修改后的代码放在github上。没有窃取原博主劳动成果的意思。  
地址为：https://github.com/manuscola/mm\_addr  
  
plus：  
fileview工具提供了按照字节，双字节 ，4字节，8字节的方式来展示内存内容，可惜我昨晚实验的时候，没好好看fileview的源代码，所以都是按照BYTE的方式展现物理内存的内容。后面有感兴趣的筒子想做实验的话，可以好好看下fileview的source code。  
  
  
参考文献：  
1[Linux内存地址映射](http://ilinuxkernel.com/?p=1276)  
2 深入理解计算机系统  
3 深入理解linux内核  
4 [Linux用户程序如何访问物理内存](http://ilinuxkernel.com/?p=1248)  
5 http://cs.usfca.edu/~cruse/cs635/

6 intel 官方手册64-ia-32-architectures-software-developer-vol-3a-part-1-manual.pdf