Linux内存地址映射

http://www.ilinuxkernel.com

目　　录

1　概述 3
2　IA-32体系结构内存地址映射 4
2.1　CPU相关寄存器 4
2.1.1　系统寄存器 5
2.1.2　内存管理寄存器 5
2.2　保护模式的内存管理 7
2.3　32位时页面机制地址映射 9
2.3.1　逻辑地址到线性地址的映射 9
2.3.2　线性地址到物理地址的映射 11
2.4　PAE页面机制地址映射过程 12
2.4.1　PDPTE寄存器 12
2.4.2　逻辑地址到线性地址的映射 13
2.4.3　线性地址到物理地址的映射 13
3　Linux内核的地址映射过程 15
3.1　段式映射过程 15
3.2　页式映射过程 17
4　Linux地址映射实验 19
4.1　gdtr和cr3寄存器值的获取 20
4.2　读取物理内存上的数据 21
4.3　地址映射过程实验 21
4.3.1　段式映射过程 23
4.3.2　页式映射过程 24
4.4　PAE机制下地址映射过程实验 26
4.4.1　段式映射过程 26
4.4.2　页式映射过程 27
5　常见问题及解答 28

1 概述

在分析Linux内存地址映射前，先来看一段简单的C程序代码在32位和64位系统上运行结果：

```
#include <stdio.h>

int main()
{
    unsigned long tmp;
    tmp = 0x12345678;

    printf("tmp variable address:0x%08lX\n", &tmp);

    return 0;
}

[root@RHEL6 C]# ./addr
tmp variable address:0xBFF42DEC
[root@RHEL6 C]#

[root@RHEL6-x64-1 C]# ./addr
tmp variable address:0x7FFF138E0888
```

上面的程序功能非常简单，就是打印出临时变量tmp的地址。程序运行后在32位系统中tmp地址为0xBFF42DEC，在64位系统中打印地址为0x7FFF138E0888（注意：每次运行的结果都可能不同）。那么这个地址对是逻辑地址，还是线性地址？物理地址？临时变量tmp到底存放在物理内存的哪个位置上？

在Intel体系结构的CPU中，现代操作系统如Linux都采用内存保护模式来管理内存。我们看Linux内核中的内存管理相关内容时，会遇到一个基本问题：普通用户程序中的地址是如何转换到内存上的物理地址的？IA-32架构的CPU规定地址映射过程是逻辑地址 ⟹ 线性地址 ⟹ 物理地址。Linux既然能在Intel架构的CPU上运行，就要遵守这个规定，那么Linux又是如何进行地址映射的？

本文以RHEL5.8 i686内核源码版本2.6.18-308（源码下载地址

ftp://ftp.redhat.com/pub/redhat/linux/enterprise/5Server/en/os/SRPMS/），在IA-32架构CPU为基础，分析CPU架构下的地址映射过程及Linux内核中的实现，并给出一些工具（附源码）来验证整个地址映射过程（包括PAE机制下的映射），方便我们更直观和深入理解Linux在x86 CPU地址映射。对于Linux内核在x64_64 CPU中的地址映射过程，会在另外一篇文章中单独介绍。

2 IA-32体系结构内存地址映射

2.1 CPU相关寄存器

IA-32架构中提供了10个32位和6个16位的寄存器。这些寄存器分为三大类：

- 通用寄存器
- 控制寄存器
- 段寄存器

通用寄存器进而分为数据、指针和索引寄存器。通用寄存器包括EAX、EBX、ECX、EDX、ESI、EDI、EBP、ESP。

数据寄存器有EAX、EBX、ECX、EDX，索引寄存器是两个：ESI（Source Index）、EDI（Destination Index），这两个寄存器都和字符串处理指令相关。指针寄存器也有两个：ESP（Stack Pointer）和EBP（Base Pointer），这两个寄存器主要来维护栈。我们知道程序运行和函数调用时，都用到栈，不同的程序以及程序中不同的函数，都有各自的栈空间，那么怎么来维护不同的栈？寄存器ESP就是指向当前栈的栈顶，而寄存器EBP指向当前栈的栈底。注意程序运行及函数调用时，不同的栈，EBP和ESP的值都不同。

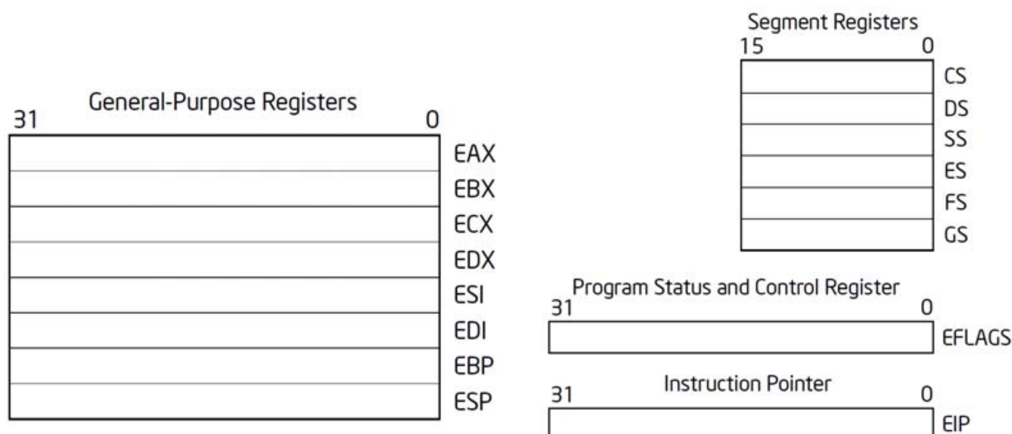


图1 通用寄存器、段寄存器和控制寄存器

控制寄存器有2个：EIP和EFLAGS。处理器使用EIP来跟踪下一条要执行的指令，也称为程序计数寄存器。

段寄存器有6个：CS、ES、DS、FS、GS、SS。

2.1.1 系统寄存器

为了初始化CPU和控制系统操作，IA-32提供了EFLAGS寄存器和几个系统寄存器。EFLAGS寄存器用来保存系统的一些状态标志。

（1）EFLAGS寄存器中的IOPL域，控制任务和模式的切换、中断处理、指令跟踪和访问权限；

（2）控制寄存器（CR0、CR2、CR3和CR4）包含用来控制系统级别操作的数据和信息；其他一些标志来只是CPU的特殊功能；CR3寄存器，对本文很重要，后面会详细介绍它的功能以及如何获取CR3积存器的值。

（3）调试寄存器允许设置程序的断点，以调试程序和系统软件；

（4）GDTR、LDTR和IDTR寄存器

（5）任务寄存器

（6）型号相关的寄存器

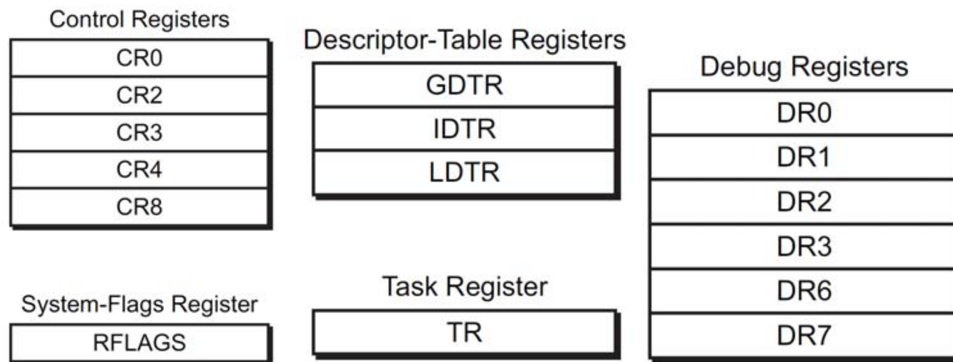


图2 系统寄存器

2.1.2 内存管理寄存器

处理器提供了4个内存管理寄存器：GDTR、LDTR、IDTR和TR。这4个寄存器在前面一小节（系统寄存器）中提到，我们关心地址映射（内存管理）相关的寄存器，故这里详细介绍一下。

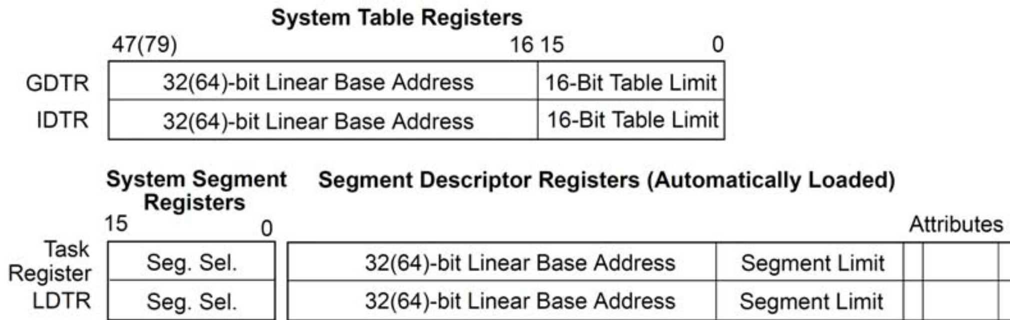


图3 内存管理寄存器

1. Global Descriptor Table Register（GDTR）

GDTR寄存器保存GDT的基址和表限（table limit）。基址是GDT的第一个字节的地址，表限（table limit）给出表的大小。

指令LGDT和SGDT是分别用来设置和保存GDTR寄存器的值。上电或重启处理器后，基址默认的值为0，表限（table limit）的值为0FFFFH。在保护模式运行前，作为处理器初始化的一部分，必须在GDTR寄存器中设置新的基址。

在后面的Linux地址映射过程中，会用到GDTR寄存器。在此顺便介绍一下汇编指令SGDT，即获取GDTR寄存器的值。SGDT指令的操作如下：

Operation

```
IF instruction is IDTR
    THEN
        IF OperandSize    16
            THEN
                DEST[0:15]    IDTR(Limit);
                DEST[16:39]    IDTR(Base); (* 24 bits of base address loaded; *)
                DEST[40:47]    0;
            ELSE (* 32-bit Operand Size *)
                DEST[0:15]    IDTR(Limit);
                DEST[16:47]    IDTR(Base); (* full 32-bit base address loaded *)
        FI;
    ELSE (* instruction is SGDT *) IF
        OperandSize    16
            THEN
                DEST[0:15]    GDTR(Limit);
                DEST[16:39]    GDTR(Base); (* 24 bits of base address loaded; *)
                DEST[40:47]    0;
            ELSE (* 32-bit Operand Size *)
                DEST[0:15]    GDTR(Limit);
                DEST[16:47]    GDTR(Base); (* full 32-bit base address loaded *)
        FI; FI;
```

2. Local Descriptor Table Register（LDTR）

LDTR寄存器的值包括：16位的段选择码、基址、段限（segment limit）和LDT描述符属性。基址是LDT段的起始地址，段限是段的大小。

指令LLDT和SLDT是分别用来设置和保存LDTR寄存器的值。

3. Interrupt Descriptor Table Register（IDTR）

IDTR寄存器的内容是：IDT的基址和大小。

指令LIDT和SIDT是分别用来设置和保存IDTR寄存器的值。

4. Task Register（TR）

TR寄存器的内容包括：16位的段选择码、基址、段限（segment limit）和描述符的属性。由于Linux中没有使用TR寄存器，这里不作详细介绍。

介绍这些CPU寄存器时，很多人一脸茫然。这4个内存管理寄存器的用途是什么？何时会用到？这里只是简单介绍，随着地址映射过程的逐步分析，再回过头来看这部分内容，就会很容易明白其用途和何时用。

2.2 保护模式的内存管理

在IA-32架构中，内存管理分为两块：分段和分页。分段机制是将代码、数据和栈分开，这样处理器上运行多个程序，不会相互影响。分页是操作系统可以实现按需分页和虚拟内存功能。分页也可以用来隔离多个任务。当运行在保护模式时，分段的基址是必须使用的，不能关闭分段功能；然而分页是可选的。在Linux系统中，实际上是没有真正使用IA-32的分段机制，仅使用分页机制。

在分析地址映射过程之前，先描述几个概念：逻辑地址（Logical Addres）、线性地址（Linear Address）和物理地址（Physical Address）。

- 逻辑地址

是在机器语言指令中，来说明操作数和指令的地址；每个逻辑地址包括两部分：段

（Segment）和偏移量（Offset）。

- 线性地址

也通常成为虚拟地址，在32位系统中，它是32位的无符号整型，最大可以达到4G。

- 物理地址

就是真正物理内存上的地址。

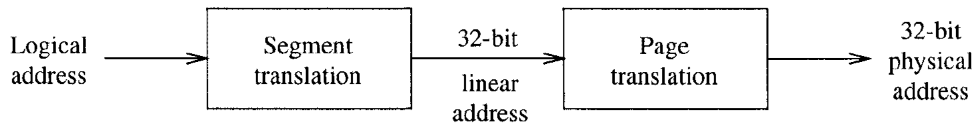


图4 地址映射过程

图5是IA-32中的段页式模型。

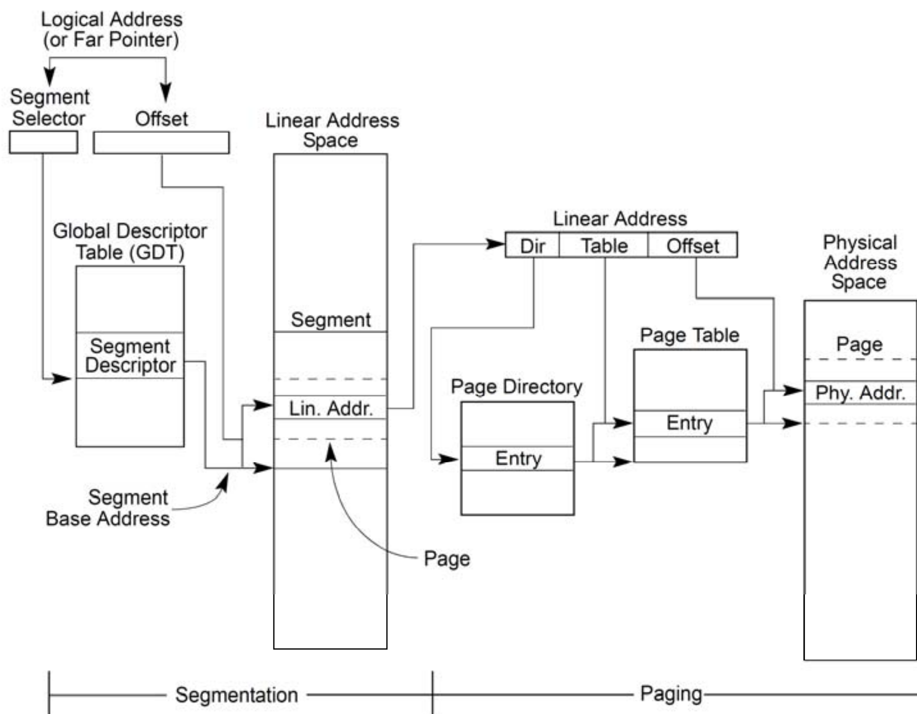


图5 IA-32中的分段和分页

从图5可以看出，地址映射过程是逻辑地址 ⟹线性地址⟹ 物理地址。也就是要经过两个映射过程：第一步是段式映射，第二步是页式映射。

当一条访问内存指令发出一个内存地址时（大家思考一下，此时CPU发出的地址类型是逻辑地址？还是线性地址？还是物理地址？）CPU就按照段式映射和页式映射两个步骤来计算出实际上应该放在数据总线的地址，

大家也许会感到访问内存，计算地址时这么麻烦！在本文最后会给出工具和源码来查看

和验证整个地址映射过程，大家有一个更为直观的认识。

2.3 32位时页面机制地址映射

2.3.1 逻辑地址到线性地址的映射

前面一节介绍了IA-32内存管理模型，及段式映射过程、页式映射过程，但只是一些概要性的介绍。现在我们看一下更为详细的过程。

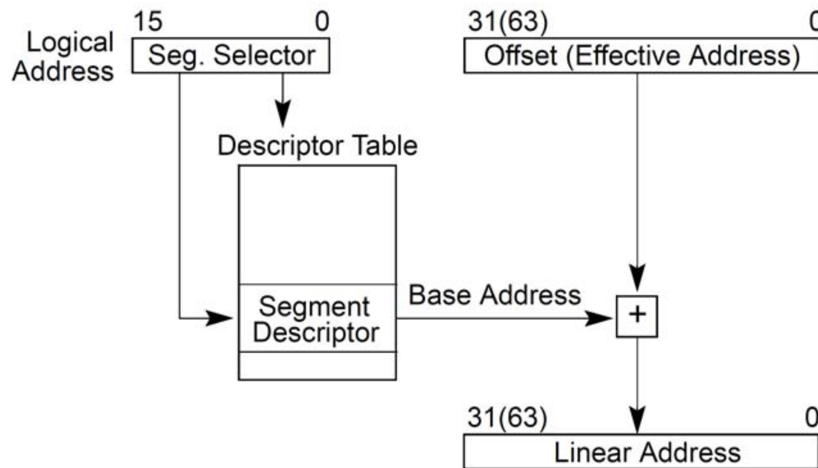


图6 逻辑地址到线性地址映射过程

从逻辑地址到线性地址的映射过程如下：

（1）根据指令的性质来确定应该使用哪一个段寄存器，例如转移指令中的地址在代码段，而取数据指令中的地址在数据段；

（2）根据段存器的内容，找到相应的“地址段描述结构”，段描述结构都放在一个表中（GDT或LDT、TR、IDT），而表的起始地址保存在GDTR、LDTR、IDTR、TR寄存器中。这就是4个内存管理寄存器GDTR、LDTR、IDTR和TR的用途；

（3）从地址段描述结构中找到基地址；

（4）将指令发出的地址作为位移，与段描述结构中规定的段长度相比，看看是否越界；

（5）根据指令的性质和段描述符中的访问权限来确定是否越权；

（6）将指令中发出的地址作为位移，与基地址相加而得出线性地址。

从逻辑地址到线性地址映射过程，有几个细节地方需要关注两个问题：

a）逻辑地址就是CPU指令发出的地址，那么段选择码（Segment Selector）的值在哪里？ b）知道段选择码后，需要从描述符表（Descriptor Table）中找到相应的表项，那怎

么知道描述符表在内存中哪个位置？

第一个问题的答案是段选择码在段寄存器中（回忆一下IA-32中的6个段寄存器），如CS、DS等。第二个问题的答案是描述符表的基址在内存管理寄存器中（GDTR、LDTR、IDTR、TR）。当然每个地址只会对应一个段寄存器和内存管理寄存器。

知道了从哪里取段选择码（Segment Selector），也知道了从内存的哪个位置取段描述符，那么段选择码和段描述符的内容是怎样的？

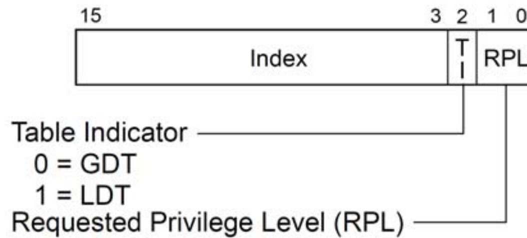
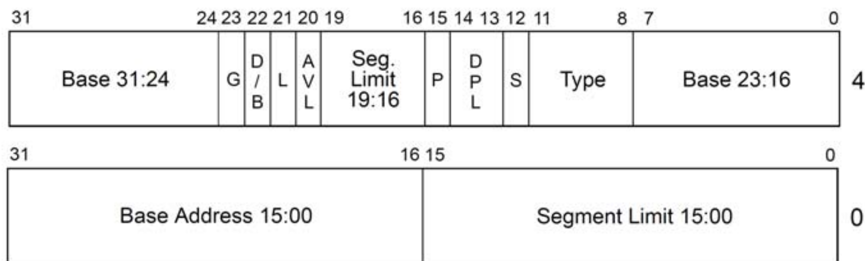


图7 段选择码



L — 64-bit code segment (IA-32e mode only)
AVL — Available for use by system software
BASE — Segment base address
D/B — Default operation size (0 = 16-bit segment; 1 = 32-bit segment)
DPL — Descriptor privilege level
G — Granularity
LIMIT — Segment Limit
P — Segment present
S — Descriptor type (0 = system; 1 = code or data)
TYPE — Segment type

图8 段描述符

图7和图8分别是段选择码和段描述符的结构。段寄存器的高13位（低3位另作他用）用作访问段描述表中具体描述结构的下标（index）。GDTR或LDTR中的段描述表指针和段寄存器中给出的下标结合在一起，才决定了具体的段描述表项在内中的什么地方。

从图8的段描述符结构中，我们注意段基址（Base Address 15:00、Base 23:16、Base 31:24）和段限（Segment Limit 15:00）。

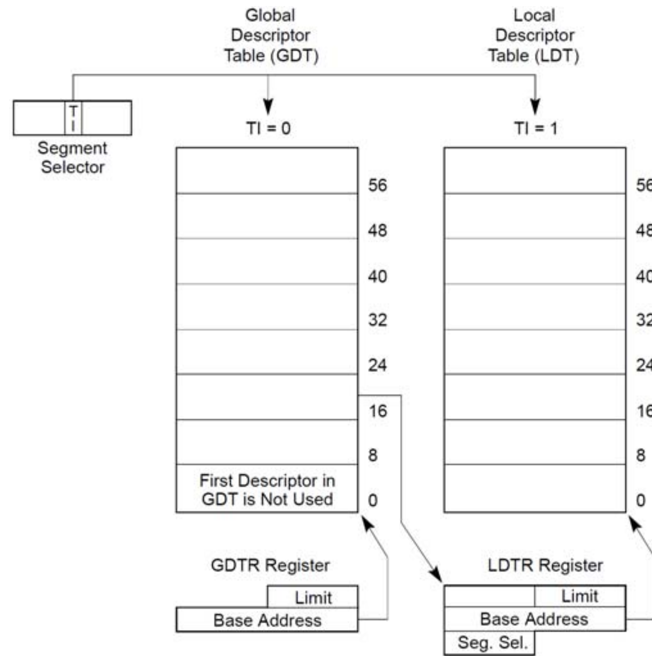


图9 全局和局部描述符表

2.3.2 线性地址到物理地址的映射

上面是将逻辑地址映射为线性地址的过程，此时还要将线性地址转换为物理地址，才是真正要访问的内存地址。

从线性地址到物理地址的映射过程为：

（1）从CR3寄存器中获取页面目录（Page Directory）的基地址；

（2）以线性地址的dir位段为下标，在目录中取得相应页面表（Page Table）的基地址；

（3）以线性地址中的page位段为下标，在所得到的页面表中获得相应的页面描述项；

（4）将页面描述项中给出的页面基地址与线性地址中的offset位段相加得到物理地址。

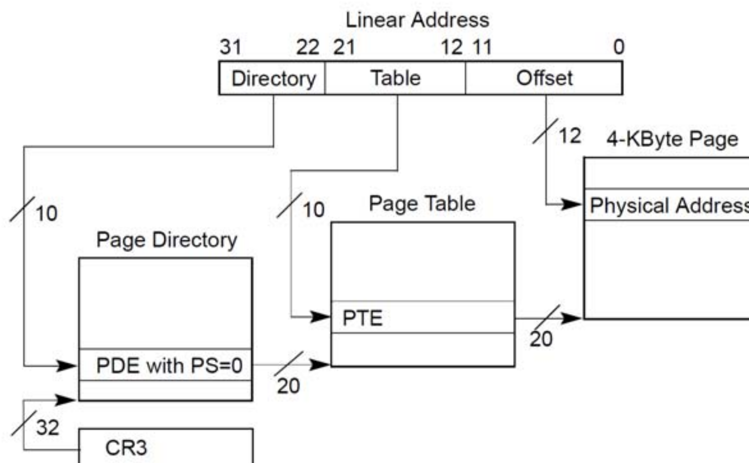


图10 线性地址到物理地址的映射过程

31:12 (fields)														
Address of page directory[1]	Ignored	PCD	PWT	Ignored	CR3									
Bits 31:22 of address of 2MB page frame	Reserved (must be 0)	Bits 39:32 of address[2]	PAT	Ignored	G	1	D	A	PCD	PWT	U/S	R/W	1	PDE: 4MB page
Address of page table	Ignored	0	Ign	A	PCD	PWT	U/S	R/W	1	PDE: page table				
Ignored	0	PDE: not present												
Address of 4KB page frame	Ignored	G	PAT	D	A	PCD	PWT	U/S	R/W	1	PTE: 4KB page			
Ignored	0	PTE: not present												

图11 32bit页面机制时CR3寄存器和页面项

对上面的步骤做一下补充说明。CR3寄存器的值从哪里来的？每个进程都会有自己的地址空间，页面目录也在内存不同的位置上，这样不同进程就有不同的CR3寄存器的值。CR3寄存器的值一般都保存在进程控制块中，如linux中的task_struct数据结构中。

2.4 PAE页面机制地址映射过程

PAE（Physical Address Extension）是IA-32架构中物理地址扩展机制。在正常情况下，32位系统只能使用4G物理内存，但使能PAE机制后使地址总线达到36根，最大可以访问64G物理内存

必须设置CR0寄存器的第31bit为0（CR0.PG=1），打开页式映射机制；设置CR4寄存器第5bit为1（CR4.PAE=1）和IA32_EFER..LMW=0，使能PAE机制。当逻辑处理器使用PAE页面机制时，会将32位线性地址转换为52位物理地址（《Intel 64 and IA-32 Architectures Software Developer's Manual Volume 3A System Programming Guide, Part 1 May 2012》章节4.4 PAE Paging）。

2.4.1 PDPTE寄存器

当使用PAE机制时，CR3寄存器的值是指向32字节对齐页面目录指针表（Page Directory Pointer Table）。

表1 PAE机制下CR3寄存器的使用

Bit Position(s)	Contents
4:0	Ignored
31:5	Physical address of the 32-Byte aligned page-directory-pointer table used for linear-address translation
63:32	Ignored (these bits exist only on processors supporting the Intel-64 architecture)

页面目录指针表中共有4个64bit项，称为PDPTEs。每个PDPTE控制1G线性地址空间的访问。对应PDPTE，逻辑处理器维护4个架构无关的内部PDPTE寄存器，分别为PDPTE0、PDPTE1、PDPTE2、PDPTE3。

表2 PAE机制下PDPTE格式

Bit Position(s)	Contents
0 (P)	Present; must be 1 to reference a page directory
2:1	Reserved (must be 0)
3 (PWT)	Page-level write-through; indirectly determines the memory type used to access the page directory referenced by this entry (see Section 4.9)
4 (PCD)	Page-level cache disable; indirectly determines the memory type used to access the page directory referenced by this entry (see Section 4.9)
8:5	Reserved (must be 0)
11:9	Ignored
(M-1):12	Physical address of 4-KByte aligned page directory referenced by this entry[1]
63:M	Reserved (must be 0)

2.4.2 逻辑地址到线性地址的映射

PAE机制仅与页面映射有关，因此逻辑地址到线性地址映射过程一样。详细步骤请参考前面非PAE机制下映射过程。

2.4.3 线性地址到物理地址的映射

PAE机制可以映射4K页面或者2MB页面，在此我们仅考虑4K页面的映射过程，如下图所示。

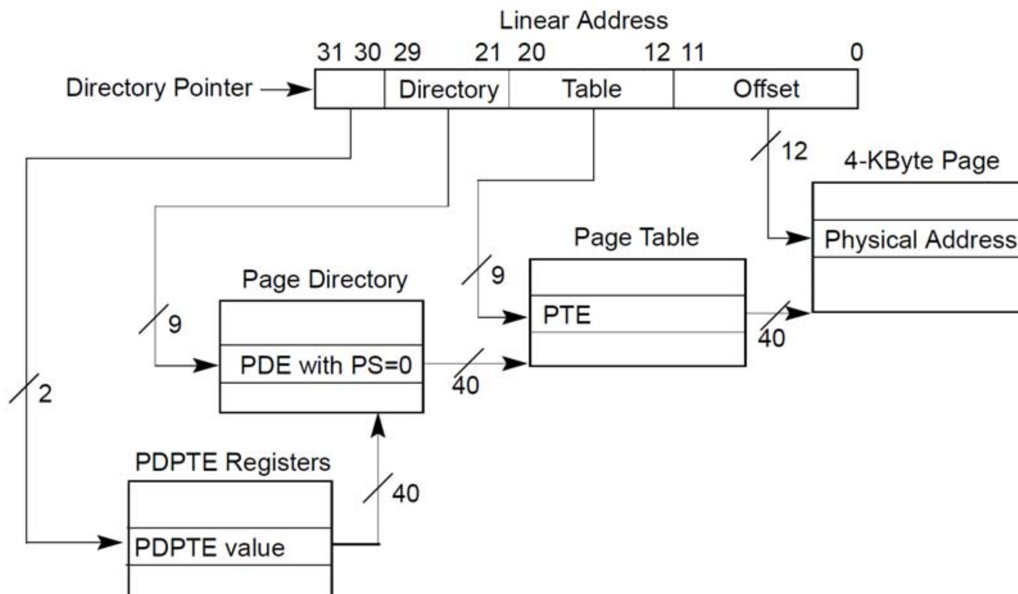


图12 PAE机制下线性地址到物理地址的映射过程

从线性地址到物理地址的映射过程为：

（1）PDPTE值的来源

线性地址的30~31bit用来选择PDPTE寄存器，即PDPTEi，i的值就是31:30的值。因为线性

地址的31:30两个bit用来选择PDPTE寄存器，因此只能控制访问1GB的线性地址空间。当PDPTEi寄存器的P标志位（bit 0）值为0时，CPU就忽略63:1，表示该寄存器控制的1GB线性地址空间没有对应的映射（即这段线性地址还没有分配相应的物理内存），当访问这段线性地址时，就会产生页面错误异常。

（2）页面目录表项

当PDPTEi寄存器的P标志位为1时，页面目录表（Page Directory Table）的地址在PDPTEi寄存器的51:12（相当于非PAE机制下CR3寄存器的值）。页面目录表由512个64bit页面目录项（Page Directory Entry）组成。具体目录项（PDE）物理地址计算方法：

- Bits 51:12来自PDPTEi （页面目录表的地址）
- Bits 11:3是线性地址的bit29:21 （页面目录表内偏移）
- Bits 2:0为0

（3）页面表项

PDE（页面目录项） 的51:12是页面表的起始物理地址。具体的页面表（PTE）物理地址计算方法：

- Bits 51:12来自PDE（页面表的起始地址）
- Bits 11:3是线性地址的bit20:12 （页面表内偏移）
- Bits 2:0为0

（4）最终物理地址的计算方法

- Bits 51:12来自PTE（页面的起始地址）
- Bits 11:0是线性地址的bit20:12 （页面内偏移）

Bits (63 … 51 … M[1] M-1 … 32 … 0) → fields, left to right	Entry
Ignored[2] | Address of page-directory-pointer table | Ignored	CR3
Reserved[3] | Address of page directory | Ign. | Rsvd. | PCD | PWT | Rsvd | 1	PDPTE: present
Ignored | 0	PDTPE: not present
XD[4] | Reserved | Address of 2MB page frame | Reserved | PAT | Ign. | G | 1 | D | A | PCD | PWT | U/S | R/W | 1	PDE: 2MB page
XD | Reserved | Address of page table | Ign. | 0 | Ign | A | PCD | PWT | U/S | R/W | 1	PDE: page table

图13 PAE页面机制时CR3寄存器和页面项

3 Linux内核的地址映射过程

Linux内核要在IA-32架构上运行，就要进行前面介绍的地址映射过程。很多人看地址映射内容，感觉是云里雾里，本身内容也比较抽象。现在开始结合Linux内核中的源码来分析我们写的普通应用程序的地址映射过程。

在概述中，我们给出了一个小程序，程序运行后打印临时变量tmp的地址为0xBFF42DEC。也许大家会有个疑问，这个地址到底是逻辑地址？线性地址？物理地址？事实上，打印的结果只是逻辑地址，而不是真正的物理地址。

既然打印的结果0xBFF42DEC是逻辑地址，那么我们就可以以此作为开始，来逐步解析整个地址映射过程。

3.1 段式映射过程

段式映射过程实际上就是从逻辑地址到线性地址的映射过程。

临时变量tmp的逻辑地址为0xBFF42DEC，那么在段式映射过程中（见图6），它就是偏移量（offset）。现在已经知道了偏移量，但还不知道段选择码。临时变量tmp存放在栈中，IA-32提供了SS（Stack Segment）寄存器，那就从SS寄存器中读取段选择码。

内核在建立一个进程时都要将其段寄存器设置好，有关代码在include/asm/processor.h。（注意：内核源码版本为2.6.18-308.el5）

```
00531:
00532: #define start_thread(regs, new_eip, new_esp) do {          \
00533:         __asm__("movl %0,%%fs ; movl %0,%%gs": :"r" (0));    \
00534:         set_fs(USER_DS);                                 \
00535:         regs->xds = __USER_DS;                           \
00536:         regs->xes = __USER_DS;                           \
00537:         regs->xss = __USER_DS;                           \
00538:         regs->xcs = __USER_CS;                           \
00539:         regs->eip = new_eip;                             \
00540:         regs->esp = new_esp;                             \
00541:         preempt_disable();                               \
00542:         load_user_cs_desc(smp_processor_id(), current->mm);     \
00543:         preempt_enable();                                \
00544: } while (0)
```

IA-32中有6个段寄存器。Linux内核建立一个进程时把DS、ES、SS寄存器的值都设为__USER_DS，CS寄存器的值设为__USER_CS，而另外两个段寄存器FS和GS都设为0。这样Linux中事实上只有只使用了两个段：代码段（CS）和数据段（DS）。而且每个进程

的6个段寄存器值都相同，只有EIP和ESP值不同。

虽然Linux中进程只分两个段（代码段和数据段），但IA-32的处理器并不知道操作系统把程序分为多少个段。接着前面临时变量tmp的逻辑地址转换到线性地址过程，偏移量已经知道为0xBFF42DEC，因为tmp在栈中，那么就要从SS寄存器中读取段选择码。而SS寄存器的值为__USER_DS，其值定义在文件include/asm/segment.h中。

```
00053:
00054: #define GDT_ENTRY_DEFAULT_USER_CS 14
00055: #define __USER_CS (GDT_ENTRY_DEFAULT_USER_CS * 8 + 3)
00056:
00057: #define GDT_ENTRY_DEFAULT_USER_DS 15
00058: #define __USER_DS (GDT_ENTRY_DEFAULT_USER_DS * 8 + 3)
```

__USER_CS（14*8 +3 = 115）的值展开二进制的结果为：

0000000001110　011

__USER_DS（15*8 + 3 =123）的值展开二进制的结果为：

0000000001111　011

高13位为index，而第三个bit都为0，表示仅使用GDT（Global Descriptor Table），而没有使用LDT（Local Descriptor Table）。事实上，Linux确实没有使用LDT。LDT只是在VM86模式中运行wine以及其他在Linux上模拟运行Windows软件或DOS软件的程序中才使用。

现在确定了偏移量(offset)和段选择码中的index，也确定了是从GDT表中，以15(index)为下标，找到相应的段描述符（segment descriptor），从段描述符中找到段的基址是多少，段限又是多少?

GDT表在内存中的位置，是由GDTR寄存器保存的。IDT表中的值是在操作系统启动时设置好的，在本文最后的实验内容中再给出如何读GDTR的值及段描述符。事实上，全局描述表中的表项值在文件arch/i386/kernel/head.S中，先将该文件中的一段代码截取出来。

```
00486: /*
00487:  * The Global Descriptor Table contains 28 quadwords, per-CPU.
00488:  */
00489:        .align L1_CACHE_BYTES
00490: ENTRY(cpu_gdt_table)
00491:        .quad 0x0000000000000000   /* NULL descriptor */
00492:        .quad 0x0000000000000000   /* 0x0b reserved */
00493:        .quad 0x0000000000000000   /* 0x13 reserved */
00494:        .quad 0x0000000000000000   /* 0x1b reserved */
00495:        .quad 0x0000000000000000   /* 0x20 unused */
00496:        .quad 0x0000000000000000   /* 0x28 unused */
```

```
00497:     .quad 0x0000000000000000    /* 0x33 TLS entry 1 */
00498:     .quad 0x0000000000000000    /* 0x3b TLS entry 2 */
00499:     .quad 0x0000000000000000    /* 0x43 TLS entry 3 */
00500:     .quad 0x0000000000000000    /* 0x4b reserved */
00501:     .quad 0x0000000000000000    /* 0x53 reserved */
00502:     .quad 0x0000000000000000    /* 0x5b reserved */
00503:
00504:     .quad 0x00cf9a000000ffff    /* 0x60 kernel 4GB code at
0x00000000 */
00505:     .quad 0x00cf92000000ffff    /* 0x68 kernel 4GB data at
0x00000000 */
00506:     .quad 0x00cffa000000ffff    /* 0x73 user 4GB code at
0x00000000 */
00507:     .quad 0x00cff2000000ffff    /* 0x7b user 4GB data at
0x00000000 */
00508:
00509:     .quad 0x0000000000000000    /* 0x80 TSS descriptor */
00510:     .quad 0x0000000000000000    /* 0x88 LDT descriptor */
00511:
```

因为我们得到的index为15（二进制为1111），那么就对应下面一项。

.quad 0x00cff2000000ffff   /* 0x7b user 4GB data at 0x00000000 */

我们再来对照图7来确定段的基址和段限（segment limit）。

0～15，48～51bits为Segment Limit。而段的基址是16～31，32～39，56～63bits组成。所以段限的值为0xffff，而段的基址值为0（16～31，32～39，56～63bits全为0）。而G为都是1，段长为4KB，而上限为0xffff，这样数据段就是从0地址开始的整个4G虚存空间，逻辑地址到线性地址的映射保持不变（因为段的基址为0）。事实上，代码段也是如此，大家有兴趣可以自行分析一下。

我们终于完成了逻辑地址到线性地址映射的过程，段式映射结束后，发现逻辑地址和虚拟地址是一样的，没有变化。临时变量tmp的逻辑地址为0xBFF42DEC，段式映射后的线性地址也为0xBFF42DEC。

3.2 页式映射过程

页式映射过程就是将线性地址映射到物理地址的过程。

临时变量tmp的逻辑地址为0xBFF42DEC，段式映射后的线性地址也0xBFF42DEC，现在将线性地址0xBFF42DEC映射到真正的物理地址，就是放在地址总线上的地址。

为了使Linux能在32位和64位CPU上运行，就要采用统一的页面地址模型。从2.6.11内核开始，页面地址模型采用了4级页面，如图14所示。

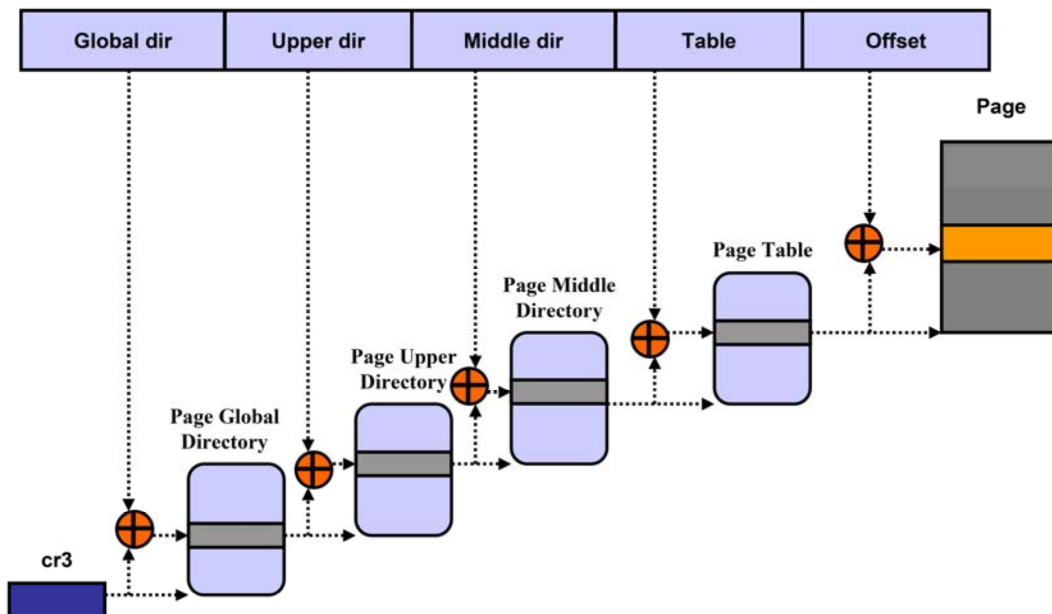


图14 Linux页式地址模型

在第二部分内容中，我们描述了IA-32的页式映射过程。完成线性地址到物理地址的映射过程，有一个寄存器很重要，就是CR3寄存器。知道CR3寄存器的值，我们就知道该进程的页面目录在内存中的位置，根据dir位段，找到相应的页面表，再根据Table位段，就可以找到tmp所在页面的起始地址。

CR3寄存器的值是从哪里设置的？内核在创建进程时，会分配页面目录，页面目录的地址就保存在task_struct结构中，task_struct结构中有个mm_struct结构类型的指针mm，mm_struct结构中有个字段pgd（见下面代码429行）就是用来保存该进程的CR3寄存器的值。下面一段代码来自kernel/fork.c，创建进程时，需要分配一个页面目录。

```
00427: static inline int mm_alloc_pgd(struct mm_struct * mm)
00428: {
00429:         mm->pgd = pgd_alloc(mm);
00430:         if (unlikely(!mm->pgd))
00431:                 return -ENOMEM;
00432:         return 0;
00433: }
```

下面一段代码是切换进程时，地址空间的切换过程。注意41行就是加载即将运行进程

的页面目录地址到CR3寄存器中。下面一段代码摘自include\asm\mmu_context.h

```
00025: static inline void switch_mm(struct mm_struct *prev,
00026:                              struct mm_struct *next,
00027:                              struct task_struct *tsk)
00028: {
00029:         int cpu = smp_processor_id();
00030:
00031:         if (likely(prev != next)) {
00032:                 /* stop flush ipis for the previous mm */
00033:                 cpu_clear(cpu, prev->cpu_vm_mask);
00034: #ifdef CONFIG_SMP
00035:                 per_cpu(cpu_tlbstate, cpu).state = TLBSTATE_OK;
00036:                 per_cpu(cpu_tlbstate, cpu).active_mm = next;
00037: #endif
00038:                 cpu_set(cpu, next->cpu_vm_mask);
00039:
00040:                 /* Re-load page tables */
00041:                 load_cr3(next->pgd);
00042:
00043:                 /*
00044:                  * load the LDT, if the LDT is different:
00045:                  */
00046:                 if (unlikely(prev->context.ldt != next->context.ldt))
00047:                         load_LDT_nolock(&next->context, cpu);
00048:         }
00049: #ifdef CONFIG_SMP
00050:         else {
00051:                 per_cpu(cpu_tlbstate, cpu).state = TLBSTATE_OK;
00052:                 BUG_ON(per_cpu(cpu_tlbstate, cpu).active_mm != next);
00053:
00054:                 if (!cpu_test_and_set(cpu, next->cpu_vm_mask)) {
00055:                         /* We were in lazy tlb mode and leave_mm disabled
00056:                          * tlb flush IPI delivery. We must reload %cr3.
00057:                          */
00058:                         load_cr3(next->pgd);
00059:                         load_LDT_nolock(&next->context, cpu);
00060:                 }
00061:         }
00062: #endif
00063: } ?     end switch_mm ?
```

后面我们会在实验中实际观察和验证地址映射过程。

4 Linux地址映射实验

理论和架构的介绍是枯燥的，现在我们可以做一些实验，以更直观验证地址映射过程。要想验证一下这个过程，需要首先解决两个问题：

（1）需要知道GDTR和CR3寄存器的值；

验证x86段式映射时，需要知道Global Descriptor Table的地址。而该表的地址保存在GDTR寄存器中。页式映射时，需要知道页面目录表和PDPTE的物理地址，这两个地址与控制寄存器CR3密切相关。

（2）要能查看所有物理内存的内容。

验证地址映射，必然涉及到物理内存。那么我们要解决如何访问指定物理地址存储的数据问题。

这两点普通用户程序肯定是没法做到的，就需要内核编程，然后结合用户编程。这里说明一下，以下编程环境和实验都在操作系统为32位RHEL5.8，内核版本为2.6.18-308.el5和2.6.18-308.el5PAE。在使用以下代码前，请确认环境，64位系统中和其他内核版本（如2.6.32系列）会有所差别。

4.1 gdtr和cr3寄存器值的获取

在用户空间的程序中，是无法直接获取gdtr和cr3寄存器的值，因为访问这两个寄存器的值需要特权指令。不过我们可以绕道达到获取这两个寄存器的值，就是在/proc文件系统下创建一个文件，通过该文件来获取用户程序的cr3寄存器和gdtr寄存器的值。这需要内核编程。

读取cr3寄存器的值，可以借助内核asm/system.h中提供的API来实现。gdtr寄存器的值，可以通过sgdt指令来读取。示例代码如下：

```
struct gdtr_struct{
    short limit;
    unsigned long address __attribute__((packed));
};

static unsigned int cr0,cr3,cr4;
static struct gdtr_struct gdtr;
    cr0 = read_cr0();
```

```
cr3 = read_cr3();

cr4 = read_cr4();

asm(" sgdt gdtr");
```

代码编译后，就可以通过命令insmod加载模块。

```
[root@localhost Programming]# insmod sys_reg.ko
[root@localhost Programming]# cat /proc/sys_reg
cr4=000006D0  PSE=1  PAE=0
cr3=35A9D000 cr0=8005003B
pgd:0xF5A9D000
gdtr address:C2011000, limit:FF
[root@localhost Programming]#
```

可以通过/proc文件系统来查看我们感兴趣的寄存器值，那么在用户程序中就可以方便的获取当前进程的相关寄存器的值了，读/proc/sys_reg文件即可。

4.2 读取物理内存上的数据

前面一节可以获取到gdtr和cr3两个寄存器的值，但页面目录、页面表都保存在物理内存中。有些数据在内核空间中，如GDT；但有些在用户空间中，如PGD和PTE。用户空间是不能访问所有内存空间的，但内核进程可以访问所有的物理内存。显然，我们仍可以通过内核编程和用户编程相结合查看所有的内存数据。

Linux用户程序如何访问物理内存，详细内容看参考http://ilinuxkernel.com/?p=1248 。

物理内存访问模块编译并加载到内核后，在通过命令mknod在/dev目录下建议一个设备文件。

#mknod /dev/phy_mem c 85 0

用户程序访问设备phy_mem，就可以访问内存上所有数据了。这和用户程序获得一些寄存器值的方式一样。

4.3 地址映射过程实验

为了方便验证地址映射，我们编写一个简单的应用程序。该程序始终不退出（因为一旦退出，所对应的物理内存就会变化），且可以读取自身进程的CR3、GDTR寄存器的值，代码如下：

```
#define REGISTERINFO "/proc/sys_reg"
#define BUFSIZE  4096
```

```
static char buf[BUFSIZE];
static unsigned long addr;

#define FILE_TO_BUF(filename, fd) do{        \
      static int local_n;     \
      if (fd == - 1 && (fd = open(filename, O_RDONLY)) == - 1) { \
      fprintf(stderr, "Open /proc/register file failed! \n");  \
      fflush(NULL);  \
      _exit(102);        \
      }       \
      lseek(fd, 0L, SEEK_SET); \
      if ((local_n = read(fd, buf , sizeof buf -    1)) < 0) {   \
      perror(filename);      \
      fflush(NULL);  \
      _exit(103);        \
      }       \
      buf [local_n] = 0;     \
}while(0)

int main()
{
      unsigned long tmp;
      tmp = 0x12345678;
      static int cr_fd = - 1;

      asm("movl %ebp, %ebx\n movl %ebx, addr");
      printf("\n%%ebp:0x%08lX\n", addr);
      printf("tmp address:0x%08lX\n", &tmp);
      FILE_TO_BUF(REGISTERINFO, cr_fd);
      printf("%s", buf );

      while(1);
      return 0;
}
```

将上面源码编译并执行。执行步骤：

（1）加载dram.ko驱动；（以访问所有物理内存）

（2）加载sys_reg.ko驱动；（读取系统寄存器）

（3）运行./mem_map程序

（4）使用fileview查看物理内存内容。

```
[root@localhost Programming]# ./mem_map

%ebp:0xBFD8E9A8
tmp address:0xBFD8E9A0
cr4=000006D0  PSE=1  PAE=0
cr3=35B0F000 cr0=8005003B
pgd:0xF5B0F000
gdtr address:C2011000, limit:FF
```

我们得到gdtr寄存器的值中地址是0xC2011000，至于gdtr寄存器的数据格式，请参照第二部分内容。gdtr寄存器给出的值，仍是线性地址。其物理地址是0x C2011000 - PAGE_OFFSET，而- PAGE_OFFSET的值是0xC0000000，所以GDT（Global Descriptor Table）所在内存的物理地址是0x2011000。

4.3.1 段式映射过程

临时变量tmp的地址为0xBFD8E9A0，这是个逻辑地址，要将映射为物理地址，首先是段式映射。gdtr寄存器的值已经知道为0x2011000，通过fileview程序查看该地址处的内容。

```
                          FILEVIEW

                        '/dev/phy_mem'

0000002011000    0000000000000000    0000000000000000
0000002011010    0000000000000000    0000000000000000
0000002011020    0000000000000000    0000000000000000
0000002011030    B7DFF2F94AD0FFFF    0000000000000000
0000002011040    0000000000000000    0000000000000000
0000002011050    0000000000000000    0000000000000000
0000002011060    00CF9A000000FFFF    00CF92000000FFFF
0000002011070    00C0FB0000008049    00CFF2000000FFFF
0000002011080    C2008B0098002073    C000827510200027
0000002011090    00409A000000FFFF    00009A000000FFFF
00000020110A0    000092000000FFFF    0000920000000000
00000020110B0    0000920000000000    00409A000000FFFF
```

大家看到GDT中index为15的值确实为

.quad 0x00cff2000000ffff   /* 0x7b user 4GB data at 0x00000000 */

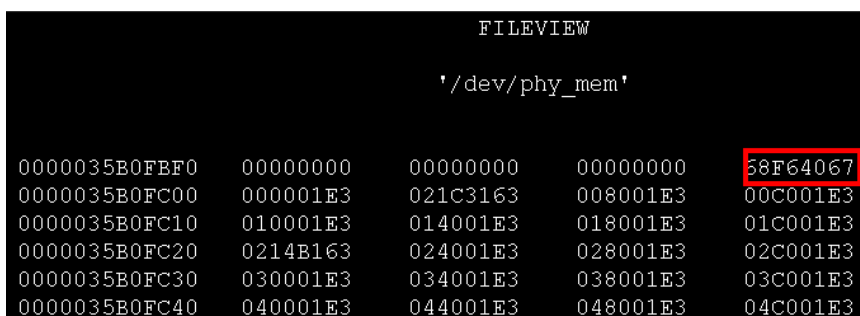
对照段式映射过程，段式映射的结果是逻辑地址和线性地址是一样的，即线性地址也是0xBFD8E9A0。

4.3.2 页式映射过程

程序给出临时变量tmp的地址是0xBFD8E9A0，将其以二进制表示为（高10bit、中间10bit、最后12bit）

1011111111　0110001110　100110100000

也就是dir（高10位）为0x2ff，由于每个数据项为4字节，而cr3寄存器的值为0x35B0F000，所以该页面表的起始地址（即Page Directory Entry）为0x35B0F000+0x2FF*4＝0x35B0FBFC。我们来看物理地址0x35B0FBFC地址中的内容。

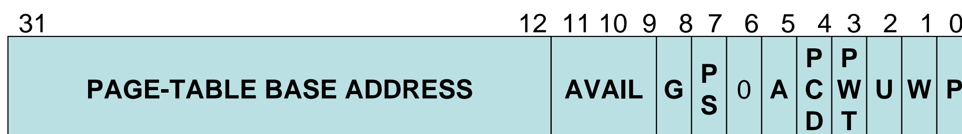


FILEVIEW

'/dev/phy_mem'

0000035B0FBF0	00000000	00000000	00000000	68F64067
0000035B0FC00	000001E3	021C3163	008001E3	00C001E3
0000035B0FC10	010001E3	014001E3	018001E3	01C001E3
0000035B0FC20	0214B163	024001E3	028001E3	02C001E3
0000035B0FC30	030001E3	034001E3	038001E3	03C001E3
0000035B0FC40	040001E3	044001E3	048001E3	04C001E3

得到的数据内容为0x68F64067，由于页面目录中的每一项都指向一个页面表，页面表的起始地址肯定是页面大小的整数倍，所以页面目录中每个数据项的低12位可作它用。见图15。所以指向的页面表起始地址为0x68F64000。



LEGEND
P = Present (1=yes, 0=no)
W = Writable (1 = yes, 0 = no)
U = User (1 = yes, 0 = no)
A = Accessed (1 = yes, 0 = no)
G = Global (1 = yes, 0 = no)
PS = Page-Size (0=4KB, 1 = 4MB)

PWT = Page Write-Through (1=yes, 0 = no)
PCD = Page Cache-Disable (1 = yes, 0 = no)

图15 页面目录数据项格式

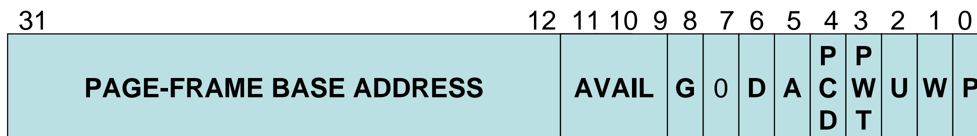
内存0x68F64000处的是页面表的起始地址，页面表内的偏移为0110001110b，即0x18E，所以临时变量tmp所在的页面起始地址保存在0x68F64000+0x18E*4= 0x68F64638处，我们来看看物理地址0x68F64638处内存的数据，该处的数据就是tmp变量所在的物理页面起始地址。

```
                         FILEVIEW

                       '/dev/phy_mem'

0000068F64610  00000000  00000000  00000000  00000000
0000068F64620  00000000  00000000  00000000  00000000
0000068F64630  00000000  68F66067  699D7067  00000000
0000068F64640  00000000  00000000  00000000  00000000
0000068F64650  00000000  00000000  00000000  00000000
0000068F64660  00000000  00000000  00000000  00000000
```

我们可以看到0x68F64638处内存的数据为0x699D7067。图16是页面表项数据格式，。低12bit是一些属性。因此tmp变量所在的物理页面起始地址为0x699D7000。



LEGEND
P = Present (1=yes, 0=no)
W = Writable (1 = yes, 0 = no)
U = User (1 = yes, 0 = no)
A = Accessed (1 = yes, 0 = no)
D = Dirty (1 = yes, 0 = no)
G = Global (1 = yes, 0 = no)
PWT = Page Write-Through (1=yes, 0 = no)
PCD = Page Cache-Disable (1 = yes, 0 = no)

图16 页面表数据项格式

tmp变量所在页面偏移量为100110100000b，即0x9A0。因此tmp变量所对应的物理地址为：0x699D7000 + 0x9A0 = 0x699D79A0。我们来看一下物理地址0x699D79A0存储的数据是什么。

```
                         FILEVIEW

                       '/dev/phy_mem'

00000699D79A0  12345678  BFD8E9C0  BFD8EA18  003CFE9C
00000699D79B0  003B1CA0  08048670  BFD8EA18  003CFE9C
00000699D79C0  00000001  BFD8EA44  BFD8EA4C  003B2828
00000699D79D0  00000000  00000001  00000001  00000000
00000699D79E0  0050FFF4  003B1CA0  00000000  BFD8EA18
00000699D79F0  954F6497  2AAB7336  00000000  00000000
```

通过我们的查看物理内存程序，看到物理地址0x699D79A0存储的数据确实为0x12345678。

终于找到程序中临时变量tmp在物理内存上的位置了，即tmp address:0xBFD8E9A0对应的物理地址为0x699D79A0。为此我们完整地验证了非PAE机制下地址映射的全过程。

4.4 PAE机制下地址映射过程实验

在前面一节，我们的实验是验证最简单地址映射过程。但没有打开PAE时，32位Linux内核只能使用4G物理内存。打开PAE机制后，最大可以使用64G物理内存，PAE机制下的地址映射稍有不同。本小节我们来验证PAE机制下地址映射过程。

下图是在PAE机制下，运行结果。

```
[root@localhost Programming]# ./mem_map

%ebp:0xBF820B98
tmp address:0xBF820B90
cr4=000006F0  PSE=1  PAE=1
cr3=021C6580 cr0=8005003B
pgd:0xC21C6580
gdtr address:C2011000, limit:FF
```

4.4.1 段式映射过程

临时变量tmp的地址为0xBF820B90，这是个逻辑地址，要将映射为物理地址，首先是段式映射。gdtr寄存器的值已经知道为0x2011000，通过fileview程序查看该地址处的内容。

```
                         FILEVIEW

                       '/dev/phy_mem'


0000002011000     0000000000000000     0000000000000000
0000002011010     0000000000000000     0000000000000000
0000002011020     0000000000000000     0000000000000000
0000002011030     B7DFF2F94AD0FFFF     0000000000000000
0000002011040     0000000000000000     0000000000000000
0000002011050     0000000000000000     0000000000000000
0000002011060     00CF9A000000FFFF     00CF92000000FFFF
0000002011070     00C0FB0000008049     00CFF2000000FFFF
0000002011080     C2008B0098002073     C000827510200027
0000002011090     00409A000000FFFF     00009A000000FFFF
00000020110A0     000092000000FFFF     0000920000000000
00000020110B0     0000920000000000     00409A000000FFFF
```

大家看到GDT中index为15的值确实为

.quad 0x00cff2000000ffff   /* 0x7b user 4GB data at 0x00000000 */

对照段式映射过程，段式映射的结果是逻辑地址和线性地址是一样的，即线性地址也是0xBF820B90。

4.4.2 页式映射过程

程序给出临时变量tmp的地址是0xBF820B90，将其以二进制表示为（高2bit、中间9bit、中间9bit、最后12bit）

10 111111100 000100000 101110010000

高2bit用来选择PDPTEi寄存器，PDPTEi寄存器的内容是存放在内存中的。共有4个PDPTE寄存器，组成PDPTE表，物理起始地址在CR3寄存器中。CR3寄存器值为0x021C6580。

1. PDPTEi寄存器的值

```
                         FILEVIEW

                       '/dev/phy_mem'

00000021C6580      0000000035878001       0000000035AEC001
00000021C6590      0000000035AED001       0000000000012001
00000021C65A0      0000000037A87001       0000000037A86001
00000021C65B0      0000000037A85001       0000000000012001
00000021C65C0      0000000037965001       0000000037966001
00000021C65D0      0000000037967001       0000000000012001
```

因此PDPTE2的值为0x0000000035AED001。

2. 页面目录表的起始地址

PDPTE2的值为0x0000000035AED001，因此页面目录表的起始物理物理为0x35AED000。

3. 页面表的地址

0x35AED000 + 111111100b*8 = 0x35AEDFE0，即页面表的物理地址存放在0x35AEDFE0地址处。页面表项的物理地址为0x6716B000。

```
                         FILEVIEW

                       '/dev/phy_mem'

0000035AEDFE0      000000006716B067       0000000000000000
0000035AEDFF0      0000000000000000       0000000000000000
0000035AEE000      0200000098393BC0       00000000062F0000
0000035AEE010      0000000000000000       0000000000000000
0000035AEE020      0000000000000000       0000000000000000
```

4. 页面表项值

页面表项的值就是tmp变量所在物理页面的起始地址。我们来看一下页面表项的值。

0x6716B000 +　000100000b*8 = 0x6716B100。该物理地址处的内容为

0x800000007C9EC067。

```
                         FILEVIEW

                      '/dev/phy_mem'

000006716B100    800000007C9EC067    0000000000000000
000006716B110    80000000674FE067    0000000000000000
000006716B120    0000000000000000    0000000000000000
000006716B130    0000000000000000    0000000000000000
```

5. 物理页面地址和tmp变量物理地址计算

tmp变量所在物理页面起始地址为0x7C9EC000。因此tmp变量所在物理地址为：

0x7C9EC000 + 101110010000b = 0x7C9ECB90。我们来验证一下物理地址

0x7C9ECB90存放的数据是什么。

```
                         FILEVIEW

                      '/dev/phy_mem'

000007C9ECB90   12345678   BF820BB0   BF820C08   003CFE9C
000007C9ECBA0   003B1CA0   08048670   BF820C08   003CFE9C
000007C9ECBB0   00000001   BF820C34   BF820C3C   003B2828
000007C9ECBC0   00000000   00000001   00000001   00000000
000007C9ECBD0   0050FFF4   003B1CA0   00000000   BF820C08
000007C9ECBE0   B4E6ECAF   0B58197E   00000000   00000000
```

物理地址0x7C9ECB90存放的数据确实是0x12345678，和我们代码中赋值一致。

至此，我们完整地验证了32位系统中PAE机制下的地址映射过程。

5　常见问题及解答

1、每个用户现在进行地址转换的的时候,查询的那个页目录表和页表,是存在内核空间还是用户空间?这些表是所有用户进程共享的,还是独立的?

答：页面目录表和页表都是存放在物理内存中的，在内核空间才能访问，不在用户空间。用户进程的页面表是独立的。

2、页面表的映射关系,是在装载器loader文件到内存的时候动态分配的吗?

答：页面表是进程创建时，内核就会分配相应的页面表。当进程访问物理内存时，内核填充相应的页面表项。

3、页面映射关系表 只有用户线程才有吧，内核线程也有这个表吗,还是直接-pageoffset就算出来了?

答：不管是内核还是用户态，都要必须遵守CPU架构的地址映射模型。内核进程寻址也是完整地进行了段页式地址转换的,不过最终转换的结果刚好就是-page_offset。

附实验源码下载地址：

http://www.ilinuxkernel.com/files/Memory_Address_Mapping.tar.bz2