Author: wzt
EMail: wzt@xsec.org
Site: http://www.xsec.org & http://hi.baidu.com/wzt85
Date: 2008-6-13

一. 绪 论

我们经常在程序的反汇编代码中看到一些类似 0x32118965 这样的地址，操作系统中称为线性地址，或虚拟地址。虚拟地址有什么用？虚拟地址
又是如何转换为物理内存地址的呢？本章将对此作一个简要阐述。

1.1 Linux 内存寻址概述

现代意义上的操作系统都处于 32 位保护模式下。每个进程一般都能寻址 4G 的物理空间。但是我们的物理内存一般都是几百 M，进程怎么能获得 4G
的物理空间呢？这就是使用了虚拟地址的好处，通常我们使用一种叫做虚拟内存的技术来实现，因为可以使用硬盘中的一部分来当作内存使用
。例外一点现在操作系统都划分为系统空间和用户空间，使用虚拟地址可以很好的保护内核空间被用户空间破坏。

对于虚拟地址如何转为物理地址,这个转换过程有操作系统和 CPU 共同完成. 操作系统为 CPU 设置好页表。CPU 通过 MMU 单元进行地址转换。

1.2 浏览内核代码的工具

现在的内核都很大， 因此我们需要某种工具来阅读庞大的源代码体系，现在的内核开发工具都选用 vim+ctag+cscope 浏览内核代码，网上已有
现成的 makefile 文件用来生成 ctags/cscope/etags。

一、用法：

找一个空目录，把附件 Makefile 拷贝进去。然后在该目录中选择性地运行如下 make 命令：

$ make

将处理/usr/src/linux 下的源文件，在当前目录生成 ctags, cscope

注：SRCDIR 用来指定内核源代码目录，如果没有指定，则缺省为/usr/src/linux/

1) 只创建 ctags

$ make SRCDIR=/usr/src/linux-2.6.12/ tags

2) 只创建 cscope

$ make SRCDIR=/usr/src/linux-2.6.12/ cscope

3) 创建 ctags 和 cscope

$ make SRCDIR=/usr/src/linux-2.6.12/

4) 只创建 etags

$ make SRCDIR=/usr/src/linux-2.6.12/ TAGS

二、处理时包括的内核源文件：

1) 不包括 drivers，sound 目录

2) 不包括无关的体系结构目录

3) fs 目录只包括顶层目录和 ext2，proc 目录
三、最简单的 ctags 命令
1) 进入
进入 vim 后，用
:tag func_name
跳到函数 func_name
2) 看函数（identifier）
想进入光标所在的函数，用
CTRL +]
3) 回退
回退用　　CTRL + T
1.3　内核版本的选取
本次论文分析， 我选取的是 linux-2.6.10 版本的内核。最新的内核代码为 2.6.25。但是现在主流的服务器都使用的是 RedHat AS4 的机器，它使
用 2.6.9 的内核。我选取 2.6.10 是因为它很接近 2.6.9，现在红帽企业 Linux 4 以 Linux2.6.9 内核为基础，是最稳定、最强大的商业产品。在 2004
年期间，Fedora 等开源项目为 Linux 2.6 内核技术的更加成熟提供了一个环境，这使得红帽企业 Linux v.4 内核可以提供比以前版本更多更好的
功能和算法，具体包括:
? 通用的逻辑 CPU 调度程序：处理多内核和超线程 CPU。
? 基于对象的逆向映射虚拟内存：提高了内存受限系统的性能。
? 读复制更新：针对操作系统数据结构的 SMP 算法优化。
? 多 I/O 调度程序：可根据应用环境进行选择。
? 增强的 SMP 和 NUMA 支持：提高了大型服务器的性能和可扩展性。
? 网络中断缓和（NAPI）：提高了大流量网络的性能。
Linux 2.6 内核使用了许多技术来改进对大量内存的使用，使得 Linux 比以往任何时候都更适用于企业。包括反向映射（reverse mapping）
、使用更大的内存页、页表条目存储在高端内存中，以及更稳定的管理器。因此，我选取 linux-2.6.10 内核版本作为分析对象。

二. X86 的硬件寻址方法
请参考 Intel x86 手册^_^

三. 内核对页表的设置
CPU 做出映射的前提是操作系统要为其准备好内核页表，而对于页表的设置，内核在系统启动的初期和系统初始化完成后都分别进行了设置。
3.1 与内存映射相关的几个宏
这几个宏把无符号整数转换成对应的类型

```
#define __pte(x)                ((pte_t) { (x) } )
#define __pmd(x)                ((pmd_t) { (x) } )
#define __pgd(x)                ((pgd_t) { (x) } )
#define __pgprot(x)             ((pgprot_t) { (x) } )
```

根据 x 把它转换成对应的无符号整数

```
#define pte_val(x)              ((x).pte_low)
```

#define pmd_val(x) ((x).pmd)
#define pgd_val(x) ((x).pgd)
#define pgprot_val(x) ((x).pgprot)
把内核空间的线性地址转换为物理地址
#define __pa(x) ((unsigned long)(x)-PAGE_OFFSET)
把物理地址转化为线性地址
#define __va(x) ((void *)((unsigned long)(x)+PAGE_OFFSET))

x 是页表项值， 通过 pte_pfn 得到其对应的物理页框号， 最后通过 pfn_to_page 得到对应的物理页描述符
#define pte_page(x) pfn_to_page(pte_pfn(x))
如果对应的表项值为 0， 返回 1
#define pte_none(x) (!(x).pte_low)
x 是页表项值， 右移 12 位后得到其对应的物理页框号
#define pte_pfn(x) ((unsigned long)(((x).pte_low >> PAGE_SHIFT)))
根据页框号和页表项的属性值合并成一个页表项值
#define pfn_pte(pfn, prot) __pte(((pfn) << PAGE_SHIFT) | pgprot_val(prot))
根据页框号和页表项的属性值合并成一个中间表项值
#define pfn_pmd(pfn, prot) __pmd(((pfn) << PAGE_SHIFT) | pgprot_val(prot))
向一个表项中写入指定的值
#define set_pte(pteptr, pteval) (*(pteptr) = pteval)
#define set_pte_atomic(pteptr, pteval) set_pte(pteptr,pteval)
#define set_pmd(pmdptr, pmdval) (*(pmdptr) = pmdval)
#define set_pgd(pgdptr, pgdval) (*(pgdptr) = pgdval)
根据线性地址得到高 10 位值， 也就是在目录表中的索引
#define pgd_index(address) (((address)>>PGDIR_SHIFT) & (PTRS_PER_PGD-1))
根据页描述符和属性得到一个页表项值
#define mk_pte(page, pgprot) pfn_pte(page_to_pfn(page), (pgprot))

3.2 内核页表的初始化

内核在进入保护模式前，还没有启用分页功能，在这之前内核要先建立一个临时内核页表，因为在进入保护模式后， 内核继续初始化直到建
立完整的内存映射机制之前， 仍然需要用到页表来映射相应的内存地址。 临时页表的初始化是在 arch/i386/kernel/head.S 中进行的：
swapper_pg_dir 是临时页全局目录表， 它是在内核编译过程中静态初始化的.
pg0 是第一个页表开始的地方， 它也是内核编译过程中静态初始化的.
内核通过以下代码建立临时页表：

```
ENTRY(startup_32)
    .....
/* 得到开始目录项的索引，从这可以看出内核是在 swapper_pg_dir 的 768 个表项开始进行建立的， 其对应的线性地址就是 0xc0000000 以上的地
址， 也就是内核在初始化它自己的页表 */
        page_pde_offset = (__PAGE_OFFSET >> 20);
/* pg0 地址在内核编译的时候， 已经是加上 0xc0000000 了， 减去 0xc00000000 得到对应的物理地址 */
```

```
        movl $(pg0 - __PAGE_OFFSET), %edi
/* 将目录表的地址传给 edx， 表明内核也要从 0x00000000 开始建立页表， 这样可以保证从以物理地址取指令到以线性地址在系统空间取指令
的平稳过渡， 下面会详细解释 */
        movl $(swapper_pg_dir - __PAGE_OFFSET), %edx
        movl $0x007, %eax
        leal 0x007(%edi),%ecx
        Movl %ecx,(%edx)
        movl %ecx,page_pde_offset(%edx)
        addl $4,%edx
        movl $1024, %ecx
11:
        stosl   addl $0x1000,%eax
        loop 11b
        /* 内核到底要建立多少页表， 也就是要映射多少内存空间， 取决于这个判断条件。在内核初始化程中内核只要保证能映射到包括内
核的代码段，数据段， 初始页表和用于存放动态数据结构的 128k 大小的空间就行 */
        leal (INIT_MAP_BEYOND_END+0x007)(%edi),%ebp
        cmpl %ebp,%eax
        jb 10b
        movl %edi,(init_pg_tables_end - __PAGE_OFFSET)
```

在上述代码中， 内核为什么要把用户空间和内核空间的前几个目录项映射到相同的页表中去呢，虽然在 head.S 中内核已经进入保护模式，但是
内核现在是处于保护模式的段式寻址方式下，因为内核还没有启用分页映射机制，现在都是以物理地址来取指令， 如果代码中遇到了符号地址
，只能减去 0xc0000000 才行， 当开启了映射机制后就不用了现在 cpu 中的取指令指针 eip 仍指向低区，如果只建立内核空间中的映射， 那么当
内核开启映射机制后， 低区中的地址就没办法寻址了，应为没有对应的页表， 除非遇到某个符号地址作为绝对转移或调用子程序为止。因此
要尽快开启 CPU 的页式映射机制.

```
movl $swapper_pg_dir-__PAGE_OFFSET,%eax
        movl %eax,%cr3              /* cr3 控制寄存器保存的是目录表地址 */
        movl %cr0,%eax              /* 向 cr0 的最高位置 1 来开启映射机制 */
        orl $0x80000000,%eax
        movl %eax,%cr0
        ljmp $__BOOT_CS,$1f         /* Clear prefetch and normalize %eip */
1:
        lss stack_start,%esp
```

通过 ljmp $__BOOT_CS,$1f 这条指令使 CPU 进入了系统空间继续执行 因为__BOOT_CS 是个符号地址，地址在 0xc0000000 以上。
在 head.S 完成了内核临时页表的建立后，它继续进行初始化，包括初始化 INIT_TASK，也就是系统开启后的第一个进程;建立完整的中断处理程
序，然后重新加载 GDT 描述符，最后跳转到 init/main.c 中的 start_kernel 函数继续初始化.

3.3 内核页表的完整建立

内核在 start_kernel()中继续做第二阶段的初始化，因为在这个阶段中， 内核已经处于保护模式下，前面只是简单的设置了内核页表， 内核
必须首先要建立一个完整的页表才能继续运行，因为内存寻址是内核继续运行的前提。
pagetable_init()的代码在 mm/init.c 中：
[start_kernel()>setup_arch()>paging_init()>pagetable_init()]
为了简单起见， 我忽略了对 PAE 选项的支持。

```
static void __init pagetable_init (void)
{
        ......
          pgd_t *pgd_base = swapper_pg_dir;
          ......
          kernel_physical_mapping_init(pgd_base);
          ......
}
```

在这个函数中 pgd_base 变量指向了 swapper_pg_dir， 这正是内核目录表的开始地址，
pagetable_init()函数在通过
kernel_physical_mapping_init()函数完成内核页表的完整建立。
kernel_physical_mapping_init 函数同样在 mm/init.c 中， 我略去了与 PAE 模式相关的代码：

```
static void __init kernel_physical_mapping_init(pgd_t *pgd_base)
{
          unsigned long pfn;
          pgd_t *pgd;
          pmd_t *pmd;
          pte_t *pte;
          int pgd_idx, pmd_idx, pte_ofs;
          pgd_idx = pgd_index(PAGE_OFFSET);
          pgd = pgd_base + pgd_idx;
          pfn = 0;
          for (; pgd_idx < PTRS_PER_PGD; pgd++, pgd_idx++) {
                    pmd = one_md_table_init(pgd);
                    if (pfn >= max_low_pfn)
                              continue;
                    for (pmd_idx = 0; pmd_idx < PTRS_PER_PMD && pfn < max_low_pfn; pmd++,
pmd_idx++) {
                    unsigned int address = pfn * PAGE_SIZE + PAGE_OFFSET;
                    ......
                    pte = one_page_table_init(pmd);
                    for (pte_ofs = 0; pte_ofs < PTRS_PER_PTE && pfn < max_low_pfn; pte++,
pfn++, pte_ofs++) {
                              if (is_kernel_text(address))
                                        set_pte(pte, pfn_pte(pfn, PAGE_KERNEL_EXEC));
                              else
                                        set_pte(pte, pfn_pte(pfn, PAGE_KERNEL));
                    ......
```

```
            }
}
```
通过作者的注释， 可以了解到这个函数的作用是把整个物理内存地址都映射到从内核空间的开始地址，即从 0xc0000000 的整个内核空间中，
直到物理内存映射完毕为止。这个函数比较长， 而且用到很多关于内存管理方面的宏定义，理解了这个函数， 就能大概理解内核是如何建立
页表的，将这个抽象的模型完全的理解。 下面将详细分析这个函数:
函数开始定义了 4 个变量 pgd_t *pgd， pmd_t *pmd， pte_t *pte， pfn;
pgd 指向一个目录项开始的地址，pmd 指向一个中间目录开始的地址，pte 指向一个页表开始的地址 pfn 是页框号被初始为 0. pgd_idx 根据
pgd_index 宏计算结果为 768，也是内核要从目录表中第 768 个表项开始进行设置。 从 768 到 1024 这个 256 个表项被 linux 内核设置成内核目录项，
低 768 个目录项被用户空间使用. pgd = pgd_base + pgd_idx; pgd 便指向了第 768 个表项。
然后函数开始一个循环即开始填充从 768 到 1024 这 256 个目录项的内容。
one_md_table_init()函数根据 pgd 找到指向的 pmd 表。
它同样在 mm/init.c 中定义:
```
static pmd_t * __init one_md_table_init(pgd_t *pgd)
{
        pmd_t *pmd_table;
#ifdef CONFIG_X86_PAE
        pmd_table = (pmd_t *) alloc_bootmem_low_pages(PAGE_SIZE);
        set_pgd(pgd, __pgd(__pa(pmd_table) | _PAGE_PRESENT));
        if (pmd_table != pmd_offset(pgd, 0))
                BUG();
#else
        pmd_table = pmd_offset(pgd, 0);
#endif
        return pmd_table;
}
```
可以看出， 如果内核不启用 PAE 选项， 函数将通过 pmd_offset 返回 pgd 的地址。因为 linux 的二级映射模型，本来就是忽略 pmd 中间目录表的。
接着又个判断语句：
```
>> if (pfn >= max_low_pfn)
>>          continue;
```
这个很关键，max_low_pfn 代表着整个物理内存一共有多少页框。当 pfn 大于 max_low_pfn 的时候，表明内核已经把整个物理内存都映射到了系
统空间中， 所以剩下有没被填充的表项就直接忽略了。因为内核已经可以映射整个物理空间了， 没必要继续填充剩下的表项。
紧接着的第 2 个 for 循环，在 linux 的 3 级映射模型中，是要设置 pmd 表的， 但在 2 级映射中忽略， 只循环一次，直接进行页表 pte 的设置。
```
>> address = pfn * PAGE_SIZE + PAGE_OFFSET;
```
address 是个线性地址， 根据上面的语句可以看出 address 是从 0xc000000 开始的，也就是从内核空间开始，后面在设置页表项属性的时候会用
到它.

>> pte = one_page_table_init(pmd);
根据 pmd 分配一个页表，代码同样在 mm/init.c 中:

```
static pte_t * __init one_page_table_init(pmd_t *pmd)
{
        if (pmd_none(*pmd)) {
        pte_t *page_table = (pte_t *) alloc_bootmem_low_pages(PAGE_SIZE);
        set_pmd(pmd, __pmd(__pa(page_table) | _PAGE_TABLE));
        if (page_table != pte_offset_kernel(pmd, 0))
                BUG();
        return page_table;
        }
        return pte_offset_kernel(pmd, 0);
}
```

pmd_none 宏判断 pmd 表是否为空，如果为空则要利用 alloc_bootmem_low_pages 分配一个 4k 大小的物理页面。 然后通过 set_pmd(pmd, __pmd (__pa(page_table) | _PAGE_TABLE));来设置 pmd 表项。page_table 显然属于线性地址，先通过 __pa 宏转化为物理地址,在与上_PAGE_TABLE 宏，
此时它们还是无符号整数，在通过__pmd 把无符号整数转化为 pmd 类型，经过这些转换，就得到了一个具有属性的表项， 然后通过 set_pmd 宏设
置 pmd 表项.
接着又是一个循环，设置 1024 个页表项。
is_kernel_text 函数根据前面提到的 address 来判断 address 线性地址是否属于内核代码段,它同样在 mm/init.c 中定义:

```
static inline int is_kernel_text(unsigned long addr)
{
        if (addr >= (unsigned long)_stext && addr <= (unsigned long)__init_end)
                return 1;
        return 0;
}
```

_stext, __init_end 是个内核符号， 在内核链接的时候生成的， 分别表示内核代码段的开始和终止地址.
如果 address 属于内核代码段，那么在设置页表项的时候就要加个 PAGE_KERNEL_EXEC 属性，如果不是，则加个 PAGE_KERNEL 属性.

```
#define _PAGE_KERNEL_EXEC \
        (_PAGE_PRESENT | _PAGE_RW | _PAGE_DIRTY | _PAGE_ACCESSED)

#define _PAGE_KERNEL \
        (_PAGE_PRESENT | _PAGE_RW | _PAGE_DIRTY | _PAGE_ACCESSED | _PAGE_NX)
```

最后通过 set_pte(pte, pfn_pte(pfn, PAGE_KERNEL));来设置页表项， 先通过 pfn_pte 宏根据页框号和页表项的属性值合并成一个页表项值，
然户在用 set_pte 宏把页表项值写到页表项里。
当 pagetable_init() 函数返回后，内核已经设置好了内核页表，紧着调用 load_cr3(swapper_pg_dir);

```
#define load_cr3(pgdir) \
        asm volatile("movl %0,%%cr3": :"r" (__pa(pgdir)))
```

将控制 swapper_pg_dir 送入控制寄存器 cr3. 每当重新设置 cr3 时， CPU 就会将页面映射目录所在的页面装入 CPU 内部高速缓存中的 TLB 部分. 现
在内存中(实际上是高速缓存中）的映射目录变了，就要再让 CPU 装入一次。由于页面映射机制本来就是开启着的， 所以从这条指令以后就扩大
了系统空间中有映射区域的大小, 使整个映射覆盖到整个物理内存(高端内存)除外. 实际上此时 swapper_pg_dir 中已经改变的目录项很可能还
在高速缓存中， 所以还要通过__flush_tlb_all()将高速缓存中的内容冲刷到内存中，这样才能保证内存中映射目录内容的一致性。

3.4 对如何构建页表的总结

通过上述对 pagetable_init()的剖析， 我们可以清晰的看到， 构建内核页表， 无非就是向相应的表项写入下一级地址和属性。 在内核空间
保留着一部分内存专门用来存放内核页表.当 cpu 要进行寻址的时候，无论在内核空间，还是在用户空间， 都会通过这个页表来进行映射。对于
这个函数， 内核把整个物理内存空间都映射完了， 当用户空间的进程要使用物理内存时，岂不是不能做相应的映射了？ 其实不会的， 内核
只是做了映射， 映射不代表使用， 这样做是内核为了方便管理内存而已。

四. 实例分析映射机制

4.1 示例代码

通过前面的理论分析，我们通过编写一个简单的程序， 来分析内核是如何把线性地址映射到物理地址的。

```
[root@localhost temp]# cat test.c
#include <stdio.h>
void test(void)
{
        printf("hello, world.\n");
}
int main(void)
{
        test();
}
```

这段代码很简单， 我们故意要 main 调用 test 函数， 就是想看下 test 函数的虚拟地址是如何映射成物理地址的。

4.2 段式映射分析

我们先编译， 在反汇编下 test 文件

```
[root@localhost temp]# gcc -o test test.c
[root@localhost temp]# objdump -d test
08048368 <test>:
8048368:        55                      push   %ebp
8048369:        89 e5                   mov    %esp,%ebp
804836b:        83 ec 08                sub    $0x8,%esp
804836e:        83 ec 0c                sub    $0xc,%esp
```

```
8048371:        68 84 84 04 08          push   $0x8048484
8048376:        e8 35 ff ff ff          call   80482b0 <printf@plt>
804837b:        83 c4 10                add    $0x10,%esp
804837e:        c9                      leave
804837f:        c3                      ret
08048380 <main>:
8048380:        55                      push   %ebp
8048381:        89 e5                   mov    %esp,%ebp
8048383:        83 ec 08                sub    $0x8,%esp
8048386:        83 e4 f0                and    $0xfffffff0,%esp
8048389:        b8 00 00 00 00          mov    $0x0,%eax
804838e:        83 c0 0f                add    $0xf,%eax
8048391:        83 c0 0f                add    $0xf,%eax
8048394:        c1 e8 04                shr    $0x4,%eax
8048397:        c1 e0 04                shl    $0x4,%eax
804839a:        29 c4                   sub    %eax,%esp
804839c:        e8 c7 ff ff ff          call   8048368 <test>
80483a1:        c9                      leave
80483a2:        c3                      ret
80483a3:        90                      nop
```

从上述结果可以看到, ld 给 test()函数分配的地址为 0x08048368.在 elf 格式的可执行文件代码中，ld 的实际位置总是从 0x8000000 开始安排程序
的代码段， 对每个程序都是这样。至于程序在执行时在物理内存中的实际位置就要由内核在为其建立内存映射时临时做出安排， 具体地址则
取决于当时所分配到的物理内存页面。假设该程序已经运行， 整个映射机制都已经建立好，并且 CPU 正在执行 main()中的 call 8048368 这条指
令， 要转移到虚拟地址 0x08048368 去运行. 下面将详细介绍这个虚拟地址转换为物理地址的映射过程.
首先是段式映射阶段。由于 0x08048368 是一个程序的入口，更重要的是在执行的过程中是由 CPU 中的指令计数器 EIP 所指向的， 所以在代码段中
。 因此， i386CPU 使用代码段寄存器 CS 的当前值作为段式映射的选择子， 也就是用它作为在段描述表的下标.那么 CS 的值是多少呢？
用 GDB 调试下 test：

```
(gdb) info reg
eax            0x10         16
ecx            0x1          1
edx            0x9d915c 10326364
ebx            0x9d6ff4 10317812
esp            0xbfedb480       0xbfedb480
ebp            0xbfedb488       0xbfedb488
esi            0xbfedb534       -1074940620
edi            0xbfedb4c0       -1074940736
eip            0x804836e        0x804836e
eflags         0x282        642
```

cs 0x73 115
ss 0x7b 123
ds 0x7b 123
es 0x7b 123
fs 0x0 0
gs 0x33 51

可以看到 CS 的值为 0x73, 我们把它分解成二进制:

0000 0000 0111 0011

最低 2 位为 3， 说明 RPL 的值为 3， 应为我们这个程序本省就是在用户空间，RPL 的值自然为 3.

第 3 位为 0 表示这个下标在 GDT 中。

高 13 位为 14， 所以段描述符在 GDT 表的第 14 个表项中， 我们可以到内核代码中去验证下:

在 i386/asm/segment.h 中:

```
#define GDT_ENTRY_DEFAULT_USER_CS            14
#define __USER_CS (GDT_ENTRY_DEFAULT_USER_CS * 8 + 3)
```

可以看到段描述符的确就是 GDT 表的第 14 个表项中。

我们去 GDT 表看看具体的表项值是什么， GDT 的内容在 arch/i386/kernel/head.S 中定义:

```
ENTRY(cpu_gdt_table)
        .quad 0x0000000000000000        /* NULL descriptor */
        .quad 0x0000000000000000        /* 0x0b reserved */
        .quad 0x0000000000000000        /* 0x13 reserved */
        .quad 0x0000000000000000        /* 0x1b reserved */
        .quad 0x0000000000000000        /* 0x20 unused */
        .quad 0x0000000000000000        /* 0x28 unused */
        .quad 0x0000000000000000        /* 0x33 TLS entry 1 */
        .quad 0x0000000000000000        /* 0x3b TLS entry 2 */
        .quad 0x0000000000000000        /* 0x43 TLS entry 3 */
        .quad 0x0000000000000000        /* 0x4b reserved */
        .quad 0x0000000000000000        /* 0x53 reserved */
        .quad 0x0000000000000000        /* 0x5b reserved */
        .quad 0x00cf9a000000ffff         /* 0x60 kernel 4GB code at 0x00000000 */
        .quad 0x00cf92000000ffff         /* 0x68 kernel 4GB data at 0x00000000 */
        .quad 0x00cffa000000ffff        /* 0x73 user 4GB code at 0x00000000 */
        .quad 0x00cff2000000ffff        /* 0x7b user 4GB data at 0x00000000 */
        .quad 0x0000000000000000        /* 0x80 TSS descriptor */
        .quad 0x0000000000000000        /* 0x88 LDT descriptor */
         /* Segments used for calling PnP BIOS */
        .quad 0x00c09a0000000000        /* 0x90 32-bit code */
        .quad 0x00809a0000000000        /* 0x98 16-bit code */
        .quad 0x0080920000000000        /* 0xa0 16-bit data */
        .quad 0x0080920000000000        /* 0xa8 16-bit data */
        .quad 0x0080920000000000        /* 0xb0 16-bit data */
         /*
```

```
         * The APM segments have byte granularity and their bases
         * and limits are set at run time.
         */
        .quad 0x00409a0000000000        /* 0xb8 APM CS    code */
        .quad 0x00009a0000000000        /* 0xc0 APM CS 16 code (16 bit) */
        .quad 0x0040920000000000        /* 0xc8 APM DS    data */
        .quad 0x0000000000000000        /* 0xd0 - unused */
        .quad 0x0000000000000000        /* 0xd8 - unused */
        .quad 0x0000000000000000        /* 0xe0 - unused */
        .quad 0x0000000000000000        /* 0xe8 - unused */
        .quad 0x0000000000000000        /* 0xf0 - unused */
        .quad 0x0000000000000000    /* 0xf8 - GDT entry 31: double-fault TSS */
.quad 0x00cffa000000ffff        /* 0x73 user 4GB code at 0x00000000 */
```

我们把这个值展开成二进制:

0000 0000 1100 1111 1111 1010 0000 0000 0000 0000 0000 0000 1111 1111 1111 1111

根据上述对段描述符表项值的描述， 可以得出如下结论：

B0-B15, B16-B31 是 0， 表示基地址全为 0.

L0-L15, L16-L19 是 1， 表示段的上限全是 0xffff.

G 位是 1　　表示段长度单位均为 4KB。

D 位是 1　　表示对段的访问都是 32 位指令

P 位是 1　　表示段在内存中。

DPL 是 3　　表示特权级是 3 级

S 位是 1　　表示为代码段或数据段

type 为 1010　　表示代码段， 可读， 可执行， 尚未收到访问

这个描述符指示了段从 0 地址开始的整个 4G 虚存空间，逻辑地址直接转换为线性地址。所以在经过段式映射后就把逻辑地址转换成了线性地址， 这也是在 linux 中， 为什么逻辑地址等同于线性地址的原因了。

4.3 页式映射分析

现在进入页式映射的过程了， Linux 系统中的每个进程都有其自身的页面目录 PGD, 指向这个目录的指针保存在每个进程的 mm_struct 数据结构

中。每当调度一个进程进入运行的时候,内核都要为即将运行的进程设置好控制寄存器 cr3，而 MMU 的硬件则总是从 cr3 中取得指向当前页面目

录的指针。当我们在程序中要转移到地址 0x08048368 去的时候， 进程正在运行，cr3 早以设置好，指向我们这个进程的页面目录了。 先将线性

地址 0x08048368 展开成二进制：

0000 1000 0000 0100 1000 0011 0110 1000

对照线性地址的格式，可见最高 10 位为二进制的 0000 1000 00, 也就是十进制的 32，所以 MMU 就以 32 为下标在其页面目录中找到其目录项。这个

目录项的高 20 位指向一个页面表，CPU 在这 20 位后添上 12 个 0 就得到页面表的指针。找到页面表以后， CPU 再来看线性地址中的中间 10 位，

0001001000，即十进制的 72.于是 CPU 就以此为下标在页表中找相应的表项。表项值的高 20 位指向一个物理内存页面，在后边添上 12 个 0 就得到物

理页面的开始地址。假设物理地址在 0x620000 的,线性地址的最低 12 位为 0x368. 那么 test() 函数的入口地址就为 0x620000+0x368 = 0x620368