第五讲 分页机制及Linux中的初步表示

分页机制在段机制之后进行，以完成线性一物理地址的转换过程。段机制把逻辑地址转换为线性地址，分页机制进一步把该线性地址再转换为物理地址。

分页机制由CR0中的PG位启用。如PG=1，启用分页机制，并使用本节要描述的机制，把线性地址转换为物理地址。如PG=0，禁用分页机制，直接把段机制产生的线性地址当作物理地址使用。分页机制管理的对象是固定大小的存储块，称之为**页(page)**。分页机制把整个线性地址空间及整个物理地址空间都看成由页组成，在线性地址空间中的任何一页，可以映射为物理地址空间中的任何一页（我们把物理空间中的一页叫做一个**页面或页框(page frame)**）。图 2-20示出分页机制如何把线性地址空间及物理地址空间划分为页，以及如何在这两个地址空间进行映射。图2.10的左边是线性地址空间，并将其视为一个具有一页大小的固定块的序列。图2.20的右边是物理地址空间，也将其视为一个页面的序列。图中，用箭头把线性地址空间中的页，与对应的物理地址空间中的页面联系起来。在这里，线性地址空间中的页，与物理地址空间中的页面是随意地对应起来。

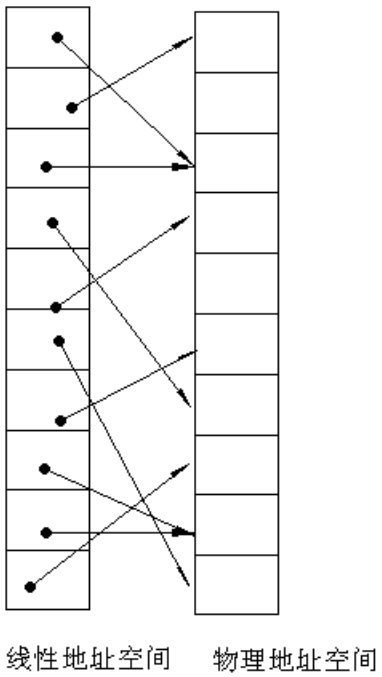


图 2-20 分页机制把线性地址转换位物理地址

IA32使用4K字节大小的页。每一页都有4K字节长，并在4K字节的边界上对齐，即每一页的起始地址都能被4K整除。因此，IA32把4G字节的线性地址空间，划分为1G个页面，每页有4K字节大小。分页机制通过把线性地址空间中的页，重新定位到物理地址空间来进行管理，因为每个页面的整个4K字节作为一个单位进行映射，并且每个页面都对齐4K字节的边界，因此，线性地址的低12位经过分页机制直接地作为物理地址的低12位使用。

线性—物理地址的转换，可将其意义扩展为允许将一个线性地址标记为无效，而不是实际地产生一个物理地址。有两种情况可能使页被标记为无效：其一是线性地址是操作系统不支持的地址；其二是在虚拟存储器系统中，线性地址对应的页存储在磁盘上，而不是存储在物理存储器中。在前一种情况下，程序因产生了无效地址而必须被终止。对于后一种情况，该无效的地址实际上是请求操作系统的虚拟存储管理系统，把存放在磁盘上的页传送到物理存储器中，使该页能被程序所访问。由于无效页通常是与虚拟存储系统相联系的，这样的无效页通常称为未驻留页，并且用页表属性位中叫做存在位的属性位进行标识。未驻留页是程序可访问的页，但它不在主存储器中。对这样的页进行访问，形式上是发生异常，实际上是通过异常进行缺页处理。

一、 分页机构

如前所述，分页是将程序分成若干相同大小的页，每页4K个字节。如果不允许分页(CR0的最高位置0)，那么经过段机制转化而来的32位线性地址就是物理地址。但如果允许分页(CR0的最高位置1)，就要将32位线性地址通过一个两级表格结构转化成物理地址。

1. 两级页表结构

为什么采用两级页表结构呢?

在IA32中页表共含1M个表项，每个表项占4个字节。如果把所有的页表项存储在一个表中，则该表最大将占4M字节连续的物理存储空间。为避免使页表占有如此巨额的物理存储器资源，故对页表采用了两级表的结构，而且对线性地址的高20位的线性—物理地址转化也分为两部完成，每一步各使用其中的10位。

两级表结构的第一级称为页目录，存储在一个4K字节的页面中。页目录表共有1K个表项，每个表项为4个字节，并指向第二级表。线性地址的最高10位(即位31~位32)用来产生第一级的索引，由索引得到的表项中，指定并选择了1K个二级表中的一个表。

两级表结构的第二级称为页表，也刚好存储在一个4K字节的页面中，包含1K个字节的表项，每个表项包含一个页的物理基地址。第二级页表由线性地址的中间10位(即位21~位12)进行索引，以获得包含页的物理地址的页表项，这个物理地址的高20位与线性地址的低12位形成了最后的物理地址，也就是页转化过程输出的物理地址，具体转化过程稍后会讲到，如图 2-21 为两级页表结构。

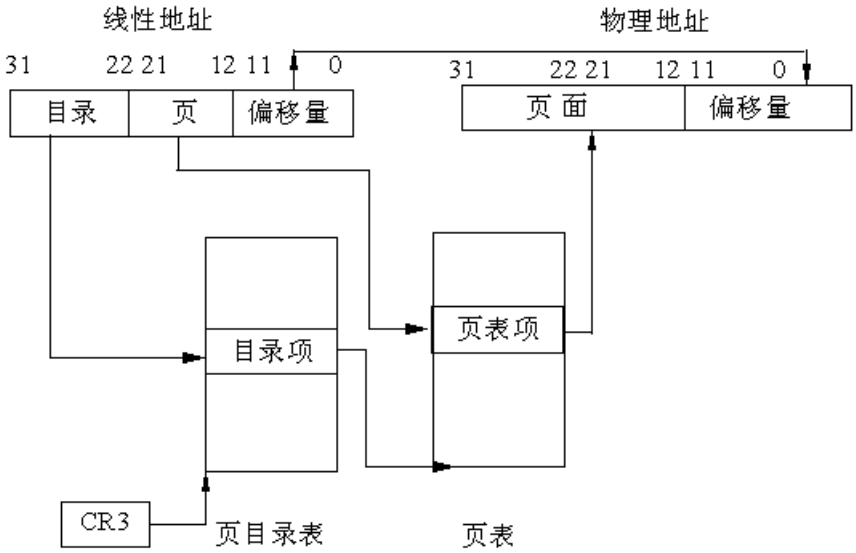


图2.21　两级页表结构

2. 页目录项

图 2-22的页目录表，最多可包含1024个页目录项，每个页目录项为4个字节，结构

	7	6	5	4	3	2	1	0
7~0位	PSE	0	A	PCD	PWT	U/S	R/W	P
15~8位	3~0位页表地址				OS专用			0
23~16位	11~4位页表地址							
31~24位	19~12位页表地址							

如图4.23所示。

图2.22 页目录中的页目录项

- 第31~12位是20位页表地址，由于页表地址的低12位总为0，所以用高20位指出32位页表地址就可以了。因此，一个页目录最多包含1024个页表地址。
- 第0位是存在位，如果P=1，表示页表地址指向的该页在内存中，如果P=0，表示不在内存中。
- 第1位是读/写位，第2位是用户/管理员位，这两位为页目录项提供硬件保护。当特权

U/S	R/W	允许级别3	允许级别0
0	0	无	读/写
0	1	无	读/写
1	0	只读	读/写
1	1	读/写	读/写

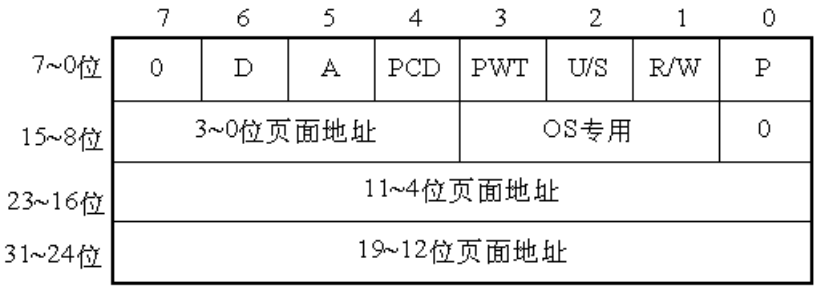
级为3的进程要想访问页面时，需要通过页保护检查，而特权级为0的进程就可以绕过页保护，如图2.23 所示。

图2.23 由U/S和R/W提供的保护

- 第3位是PWT（Page Write-Through）位，表示是否采用写透方式，写透方式就是既写内存（RAM）也写高速缓存,该位为1表示采用写透方式
- 第4位是PCD（Page Cache Disable）位，表示是否启用高速缓存,该位为1表示启用高速缓存。
- 第5位是访问位，当对页目录项进行访问时，A位=1。
- 第7位是Page Size标志，只适用于页目录项。如果置为1，页目录项指的是4MB的页面，请看后面的扩展分页。
- 第9~11位由操作系统专用，Linux也没有做特殊之用。

3. 页面项

IA32的每个页目录项指向一个页表，页表最多含有1024个页面项，每项4个字节，包



含页面的起始地址和有关该页面的信息。页面的起始地址也是4K的整数倍，所以页面的低12位也留作它用，如图2.24所示。

图2.24 页表中的页面项

第31~12位是20位物理页面地址，除第6位外第0～5位及9~11位的用途和页目录项一样，第6位是页面项独有的，当对涉及的页面进行写操作时，D位被置1。

4GB的存储器只有一个页目录，它最多有1024个页目录项，每个页目录项又含有1024个页面项，因此，存储器一共可以分成1024×1024=1M个页面。由于每个页面为4K个字节，所以，存储器的大小正好最多为4GB。

4. 线性地址到物理地址的转换

当访问一个操作单元时，如何由分段结构确定的32位线性地址通过分页操作转化成32位物理地址呢？过程如图2.25所示。

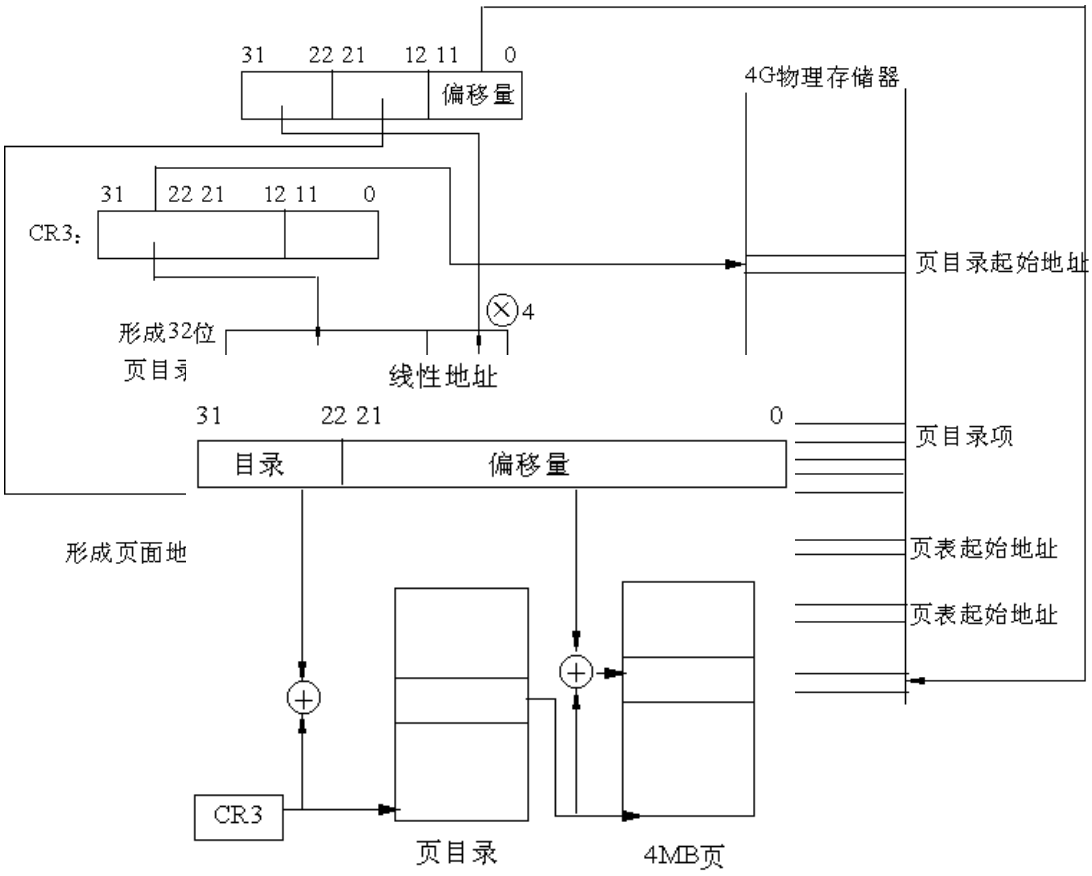


图2.25　　32位线性地址到物理地址的转换

第一步，CR3包含着页目录的起始地址，用32位线性地址的最高10位A31~A22作为页目录的页目录项的索引，将它乘以4，与CR3中的页目录的起始地址相加，形成相应页表的地址。

第二步，从指定的地址中取出32位页目录项，它的低12位为0，这32位是页表的起始地址。用32位线性地址中的A21~A12位作为页表中的页面的索引，将它乘以4，与页表的起始地址相加，形成32位页面地址。

第三步，将A11~A0作为相对于页面地址的偏移量，与32位页面地址相加，形成32位物理地址。

5.扩展分页

从奔腾处理器开始,Intel微处理器引进了扩展分页,它允许页的大小为4MB,

在扩展分页的情况下,分页机制把32位线性地址分成两个域：最高10位的目录域和其余22位的偏移量。

二、 Linux中的分页机制

如前所述，Linux主要采用分页机制来实现虚拟存储器管理。这是因为：

- Linux的分段机制使得所有的进程都使用相同的段寄存器值，这就使得内存管理变得简单，也就是说，所有的进程都使用同样的线性地址空间（0～4G）。
- Linux设计目标之一就是能够把自己移植到绝大多数流行的处理器平台。但是，许多RISC处理器支持的段功能非常有限。

为了保持可移植性，Linux采用三级分页模式而不是两级，这是因为许多处理器（如康柏的Alpha，Sun的UltraSPARC，Intel的Itanium）都采用64位结构的处理器，在这种情况下，两级分页就不适合了，必须采用三级分页。图2.28为三级分页模式，为此，Linux定义了三种类型的页表:

- 总目录PGD（Page Global Directory）
- 中间目录PMD（Page Middle Derectory）
- 页表PT（Page Table）

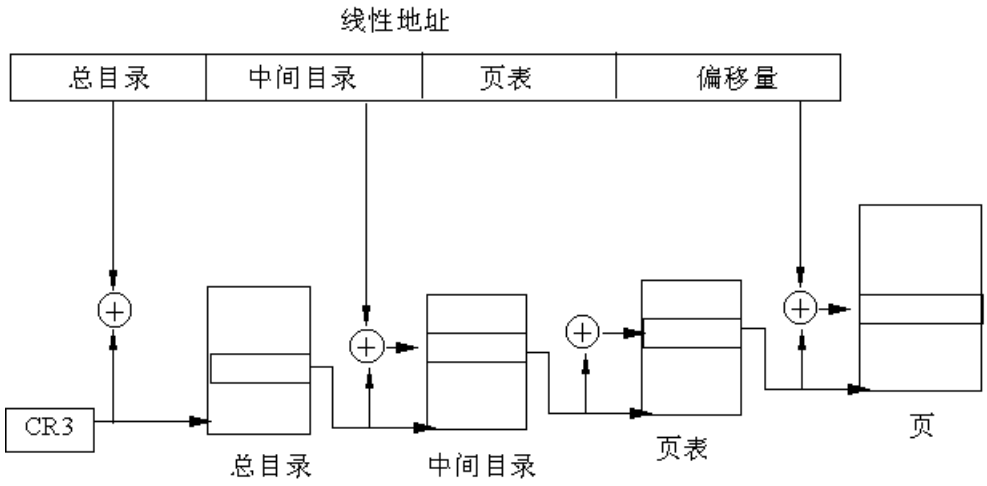


图2.28 Liunx的三级分页

尽管Linux采用的是三级分页模式，但我们的讨论还是以Intel奔腾处理器的两级分页模式为主，因此，Linux忽略中间目录层，以后，我们把总目录就叫页目录。

（一） 与页相关的数据结构及宏的定义

上一节讨论的分页机制是硬件对分页的支持，这是虚拟内存管理的硬件基础。要想使这种硬件机制充分发挥其功能，必须有相应软件的支持，我们来看一下Linux所定义的一些主要数据结构，其分布在include/asm-i386/目录下的page.h，pgtable.h及pgtable-2level.h三个文件中。

1. 表项的定义

如上所述，PGD、PMD及PT表的表项都占4个字节，因此，把它们定义为无符号长整数，分别叫做pgd_t、pmd_t及pte_t(pte 即Page table Entry)，在page.h中定义如下：

```
typedef struct { unsigned long pte_low; } pte_t;
typedef struct { unsigned long pmd; } pmd_t;
typedef struct { unsigned long pgd; } pgd_t;
typedef struct { unsigned long pgprot; } pgprot_t;
```

可以看出，Linux没有把这几个类型直接定义长整数而是定义为一个结构，这是为了

让gcc在编译时进行更严格的类型检查。另外，还定义了几个宏来访问这些结构的成分，这也是一种面向对象思想的体现：

```
#define pte_val(x)      ((x).pte_low)
#define pmd_val(x)      ((x).pmd)
#define pgd_val(x)      ((x).pgd)
```

从图2.22和图2.24 可以看出，对这些表项应该定义成位段，但内核并没有这样定义，而是定义了一个页面保护结构pgprot_t和一些宏：

```
typedef struct { unsigned long pgprot; } pgprot_t;
#define pgprot_val(x)   ((x).pgprot)
```

字段pgprot的值与图2.24页面项的低12位相对应，其中的9位对应0～9位，在pgtalbe.h中定义了对应的宏：

```
#define _PAGE_PRESENT   0x001
#define _PAGE_RW        0x002
#define _PAGE_USER      0x004
#define _PAGE_PWT       0x008
#define _PAGE_PCD       0x010
#define _PAGE_ACCESSED  0x020
#define _PAGE_DIRTY     0x040
#define _PAGE_PSE       0x080   /* 4 MB (or 2MB) page, Pentium+, if present.. */
#define _PAGE_GLOBAL    0x100   /* Global TLB entry PPro+ */
```

在你阅读源代码的过程中你能体会到，把标志位定义为宏而不是位段更有利于编码。

另外，页目录表及页表在 pgtable.h 中定义如下：

```
extern pgd_t swapper_pg_dir[1024];
extern unsigned long pg0[1024];
```

swapper_pg_dir 为页目录表，pg0 为一临时页表，每个表最多都有 1024 项。

2．线性地址域的定义

Intel 线性地址的结构如图 2.29 所示：

图 2.29　32 位的线性地址结构

（1） 偏移量的位数

```
#define PAGE_SHIFT      12
#define PAGE_SIZE       (1UL << PAGE_SHIFT)
#define PTRS_PER_PTE    1024
#define PAGE_MASK       (~(PAGE_SIZE-1))
```

其中 PAGE_SHIFT 宏定义了偏移量的位数为 12，因此页大小 PAGE_SIZE 为 2^{12} = 4096 字节；PTRS_PER_PTE 为页表的项数；最后 PAGE_MASK 值定义为 0xfffff000，用以屏蔽掉偏移量域的所有位（12 位）。

（2） PGDIR_SHIFT

```
#define PGDIR_SHIFT     22
#define PTRS_PER_PGD    1024
#define PGDIR_SIZE      (1UL << PGDIR_SHIFT)
```

```
#define PGDIR_MASK        (~(PGDIR_SIZE-1))
```

PGDIR_SHIFT 是页表所能映射区域线性地址的位数，它的值为 22（12 位的偏移量加上 10 位的页表）；PTRS_PER_PGD 为页目录目录项数；PGDIR_SIZE 为页目录的大小,为 2^{22}，即 4MB；PGDIR_MASK 为 0xffc00000，用于屏蔽偏移量位与页表域的所有位。

（3）PMD_SHIFT

```
#define PMD_SHIFT        22
#define PTRS_PER_PMD     1
```

PMD_SHIFT 为中间目录表映射的地址位数，其值也为 22，但是因为 Linux 在 386 中只用了两级页表结构，因此，让其目录项个数为 1，这就使得中间目录在指针序列中的位置被保存，以便同样的代码在 32 位系统和 64 位系统下都能使用。后面的讨论我们不再提及中间目录。

（二） 对页目录及页表的处理

在 page.h，pgtable.h 及 pgtable-2level.h 三个文件中还定义有大量的宏，用以对页目录、页表及表项的处理，我们在此介绍一些主要的宏和函数。

1.表项值的确定

```
static inline int pgd_none(pgd_t pgd)            { return 0; }
static inline int pgd_present(pgd_t pgd)         { return 1; }

#define pte_present(x) ((x).pte_low & (_PAGE_PRESENT | _PAGE_PROTNONE))
```

pgd_none（）函数直接返回 0，表示尚未为这个页目录建立映射，所以页目录项为空。pgd_present（）函数直接返回 1，表示映射虽然还没有建立，但页目录所映射的页表

肯定存在于内存（即页表必须一直在内存）。

pte_present 宏的值为 1 或 0，表示 P 标志位。如果页表项不为 0，但标志位为 0，则表示映射已经建立，但所映射的物理页面不在内存。

2. 清相应表的表项:

```
#define pgd_clear(xp)                                    do { } while (0)

#define pte_clear(xp)    do { set_pte(xp, __pte(0)); } while (0)
```

pgd_clear 宏实际上什么也不做，定义它可能是为了保持编程风格的一致。pte_clear 就是把 0 写到页表表项中。

3.对页表表项标志值进行操作的宏。

这些宏的代码在 pgtable.h 文件中，表 2.1 给出宏名及其功能。

表 2.1 对页表表项标志值进行操作的宏及其功能

宏名	功能
Set_pte()	把一个具体的值写入表项
Pte_read()	返回 User/Supervisor 标志值（由此可以得知是否可以在用户态下访问此页）
Pte _write()	如果 Present 标志和 Read/Write 标志都为 1，则返回 1（此页是否存在并可写）
Pte _exec()	返回 User/Supervisor 标志值
Pte _dirty()	返回 Dirty 标志的值（说明此页是否被修改过）
Pte _young()	返回 Accessed 标志的值（说明此页是否被存取过）
Pte _wrprotect()	清除 Read/Write 标志
Pte _rdprotect()	清除 User/Supervisor 标志
Pte _mkwrite	设置 Read/Write 标志
Pte _mkread	设置 User/Supervisor 标志
Pte _mkdirty()	把 Dirty 标志置 1
Pte _mkclean()	把 Dirty 标志置 0
Pte _mkyoung	把 Accessed 标志置 1
Pte _mkold()	把 Accessed 标志置 0
Pte _modify(p,v)	把页表表项 p 的所有存取权限设置为指定的值 v
Mk_pte()	把一个线性地址和一组存取权限合并来创建一个 32 位的页表表项
Pte _pte_phys()	把一个物理地址与存取权限合并来创建一个页表表项
Pte _page()	从页表表项返回页的线性地址

实际上页表的处理是一个复杂的过程，在这里我们仅仅让读者对软硬件如何结合起来有一个初步的认识，有关页表更多的内容我们将在内存管理部分接着讨论。

三、Linux 系统地址映射举例

Linux 采用分页存储管理。虚拟地址空间划分成固定大小的“页”，由 MMU 在运行时将虚拟地址映射（变换）成某个物理页面中的地址。从 IA32 系列的历史演变过程可知，分段管理在分页管理之前出现，因此，IA32 的 MMU 对程序中的虚拟地址先进行段式映射（虚拟地址转换为线性地址），然后才能进行页式映射（线性地址转换为物理地址）。既然硬件结构是这样设计的，Linux 内核在设计时只好服从这种选择，只不过，Linux 巧妙地使段式映射实际上不起什么作用。本节通过一个程序的执行来说明地址的映射过程。

假定我们有一个简单的 C 程序 Hello.c

```
# include <stdio.h>
greeting ( )
{
      printf( “Hello,world!\n” );
}
main()
 {
       greeting();
 }
```

之所以把这样简单的程序写成两个函数，是为了说明指令的转移过程。我们用 gcc 和 ld 对其进行编译和连接，得到可执行代码 hello。然后，用 Linux 的实用程序 ob,jdump 对其进行反汇编:

```
% objdump —d hello
```

得到的主要片段为:

```
08048568 <greeting>:
8048568:      push1  %ebp
8048569:      mov1  %esp, %ebp
804856b:      push1  $0x809404
8048570:      call1    8048474  <_init+0x84>
8048575:      add1   $0x4, %esp
8048578:      leave
8048579:      ret
804857a:      mov1  %esi, %esi
0804857c <main>:
804857c:      push1  %ebp
804857d:      mov1  %esp, %ebp
804857f:      call1    8048568  <greeting>
8048584:      leave
8048585:      ret
8048586:      nop
8048587:      nop
```

最左边的数字是连接程序 ld 分配给每条指令或标识符的虚拟地址,其中分配给 greeting()这个函数的起始地址为 0x08048568。Linux 最常见的可执行文件格式为 elf(Executable and Linkable Format)。在 elf 格式的可执行代码中，ld 总是从 0x8000000 开始安排程序的“代码段”，对每个程序都是这样。至于程序执行时在物理内

存中的实际地址，则由内核为其建立内存映射时临时分配，具体地址取决于当时所分配的物理内存页面。

假定该程序已经开始运行，整个映射机制都已经建立好，并且 CPU 正在执行 main()中的“call 08048568”这条指令，于是转移到虚地址 0x08048568。Linux 内核设计的段式映射机制把这个地址原封不动地映射为线性地址，接着就进入页式映射过程。

每当调度程序选择一个进程运行时，内核就要为即将运行的进程设置好控制寄存器 CR3，而 MMU 的硬件总是从 CR3 中取得指向当前页目录的指针。

当我们的程序转移到地址 0x08048568 的时候，进程正在运行中，CR3 指向我们这个进程的页目录。根据线性地址 0x08048568 最高 10 位，就可以找到相应的目录项。把 08048568 按二进制展开：

0000 1000 0000 0100 1000 0101 0110 1000

最高 10 位为 0000 1000 00，即十进制 32，这样以 32 为下标在页目录中找到其目录项。这个目录项中的高 20 位指向一个页表，CPU 在这 20 位后填 12 个 0 就得到该页表的物理地址。

找到页表之后，CPU 再来找线性地址的中间 10 位，为 0001001000，即十进制 72，于是 CPU 就以此为下标在页表中找到相应的页表项，取出其高 20 位，假定为 0x840，然后与线性地址的最低 12 位 0x568 拼接起来，就得到 greeting()函数的入口物理地址为 0x840568，greeting()的执行代码就存储在这里。