# 第二章 内存寻址

本章介绍寻址技术。值得庆幸的是，操作系统自身不必完全了解物理内存；如今的微处理器包含的硬件线路使内存管理既高效又健壮，所以编程错误就不会导致内存不正确的访问。

作为本书的一部分，本章将详细描述 80x86 微处理器怎样进行芯片级的内存寻址，Linux 又是如何利用寻址硬件的。我们希望当你学习内存寻址技术在 Linux 最流行的硬件平台上的详细实现方法时，既能够更好地理解分页单元的一般原理，又能更好地研究内存寻址技术在其他平台上是如何实现的。

关于内存管理有三章，这是其中的第一章；还有第八章，讨论内核怎样给自己分配主存；以及第九章，考虑怎样给进程分配线性地址。

## 内存地址

程序员偶尔会引用内存地址（*memory address*）作为访问内存单元内容的一种方式，但是，当使用 80x86 微处理器时，我们必须区分以下三种不同的地址：

逻辑地址（*logical address*）

> 包含在机器语言指令中用来指定一个操作数或一条指令的地址。这种寻址方式在 80x86 著名的分段结构中表现得尤为具体，它促使 MS-DOS 或 windows 程序员把程序分成若干段。每一个逻辑地址都由一个段*(segment)*和偏移量（*offset* 或 *displacement*）组成，偏移量指明了从段开始的地方到实际地址之间的距离。

线性地址（*linear address*）*(*也称虚拟地址*virtual address)*

> 是一个 32 位无符号整数，可以用来表示高达 4GB 的地址，也就是，高达 4 294 967 296 个存储器单元。线性地址通常用 16 进制数字表示，值的范围从 0x00000000 到 0xffffffff。

物理地址（*physical address*）

> 用于内存芯片级内存单元寻址。它们与从微处理器的地址引脚发送到内存总线上的电信号相对应。物理地址由 32 位或 36 位无符号整数表示。

内存控制单元（MMU）通过一种称为分段单元（*segmentation unit*）的硬件电路把一个逻辑地址转换成线性地址；接着，第二个称为分页单元（*paging unit*）的硬件电路把线性地址转换成一个物理地址（见图2-1）。

图 2-1：逻辑地址转换

在多处理机系统中，所有 CPU 都共享同一内存；这意味着 RAM 芯片可以由独立的 CPU 并发地访问。因为在 RAM 芯片上的读或写操作必须串行地执行，因此一种所谓内存仲裁器（memory arbiter）的硬件电路插入在总线和每个 RAM 芯片之间。其作用是如果某个 RAM 芯片空闲，就准予一个 CPU 访问，如果该芯片忙于为另一个处理器提出的请求服务，就延迟这个 CPU 的访问。即使在单处理器上也使用存储器仲裁器，因为单处理器系统中包含一个叫做 DMA 的特殊处理器，而 DMA 与 CPU 并发操作（参见第十三章“直接存储器访问（DMA）”一节“）。在多处理器系统的情况下，因为仲裁器有多个输入端口，所以其结构更加复杂。例如，双 Pentium 在每个芯片的入口维持一个两端口仲裁器，并在试图使用公用总线前请求两个 CPU 交换同步信息。从编程观点看，因为仲裁器由硬件电路管理，因此它是隐藏的。

## 硬件中的分段

从80286模式开始，Intel微处理器以两种不同的方式执行地址转换，这两种方式分别称为实模式(real mode)和保护模式(protected mode)。我们将从下一节开始描述保护模式下的地址转换。实模式存在的主要原因是要维持处理器与早期模型兼容，并让操作系统自举(参阅附录一中针对实模式的简短描述)。

### 段选择符和段寄存器

一个逻辑地址由两部分组成：一个段标识符和一个指定段内相对地址的偏移量。段标识符是一个16位长的字段，称为段选择符(segment selector 见图2-2)，而偏移量是一个32位长的字段。我们将在本章“快速访问段描述符”一节描述断选择符字段。


图 2-2：段描述符格式

为了快速方便地找到段选择符，处理器提供段寄存器，段寄存器的唯一目的是存放段选择符。这些段寄存器称为 cs， ss， ds， es， fs 和 gs。尽管只有 6 个段寄存器，但程序可以把同一个段寄存器用于不同的目的，方法是先将其值保存在存储器中，用完后再恢复。

6 个寄存器中 3 个有专门的用途：

cs
代码段寄存器，指向包含程序指令的段。

ss
栈段寄存器，指向包含当前程序栈的段。

ds
数据段寄存器，指向包含静态数据或者全局数据的段。

其它三个段寄存器作一般用途，可以指向任意的数据段。

cs 寄存器还有一个很重要的功能：它含有一个两位的字段，用以指明 CPU 的当前特权级（Current Privilege Level，CPL）。值为 0 代表最高优先级，而值为 3 代表最低优先级。Linux 只用 0 级和 3 级，分别称之为内核态和用户态。

## 段描述符

每个段由一个8字节的段描述符(Segment Descriptor)表示（参见图2-2），它描述了段的特征。段描述符放在全局描述符表（Global Descriptor Table, GDT）或局部描述符表（Local Descriptor Table, LDT）中。

通常只定义一个GDT，而每个进程除了存放在GDT中的段之外如果还需要创建附加的段，就可以有自己的LDT。GDT在主存中的地址和大小存放在gdtr处理器寄存器中，当前正被使用的LDT地址和大小放在ldtr处理器寄存器中。

图 2-3 阐明了段描述符的格式；表 2-1 解释了图中各个字段的含义。

表 2-1：段描述符字段

| 字段名 | 描述 |
| --- | --- |
| Base | 包含段的首字节的线性地址 |
| G | 粒度标志 G：如果该位清为 0，段大小以字节为单位，否则以 4096 字节的倍数计。 |
| Limit | 存放段中最后一个内存单元的偏移量，从而决定段的长度。如果 G 被置为 0，段的大小在 1 个字节到 1MB 之间变化；否则，将在 4KB 到 4GB 之间变化。 |
| S | 系统标志 S，如果它被清 0，则这是一个系统段，存储诸如局部描述符表这种关键的数据结构，否则它是一个普通的代码段或数据段。 |
| Type | 描述了段的类型特征和它的存取权限（请看表下面的描述）。 |
| DPL | 描述符特权级（Descriptor Privilege Level）字段：用于限制对这个段的存取。它表示为访问这个段而要求的 CPU 最小的优先级。因此，DPL 设为 0 的段只能当 CPL 为 0 时（即在内核态）才<br>是可访问的，而 DPL 设为 3 的段对任何 CPL 值都是可访问的。 |
| P | Segment-Present 标志：等于 0 表示段当前不在主存中。Linux 总是把这个标志（第 47 位）设为 1，因为它从来不把整个段交换到磁盘上去。 |

D 或 B　　称为 D 或 B 的标志，取决于是代码段还是数据段。D 或 B 的含义在两种情况下稍微有所区别，但是如果段偏移量的地址是 32 位长，就基本上把它置为 1，如果这个偏移量是 16 位长，它被清 0（更详细描述参见 Intel 使用手册）。

AVL 标志　　可以由操作系统使用，但是被 Linux 忽略。

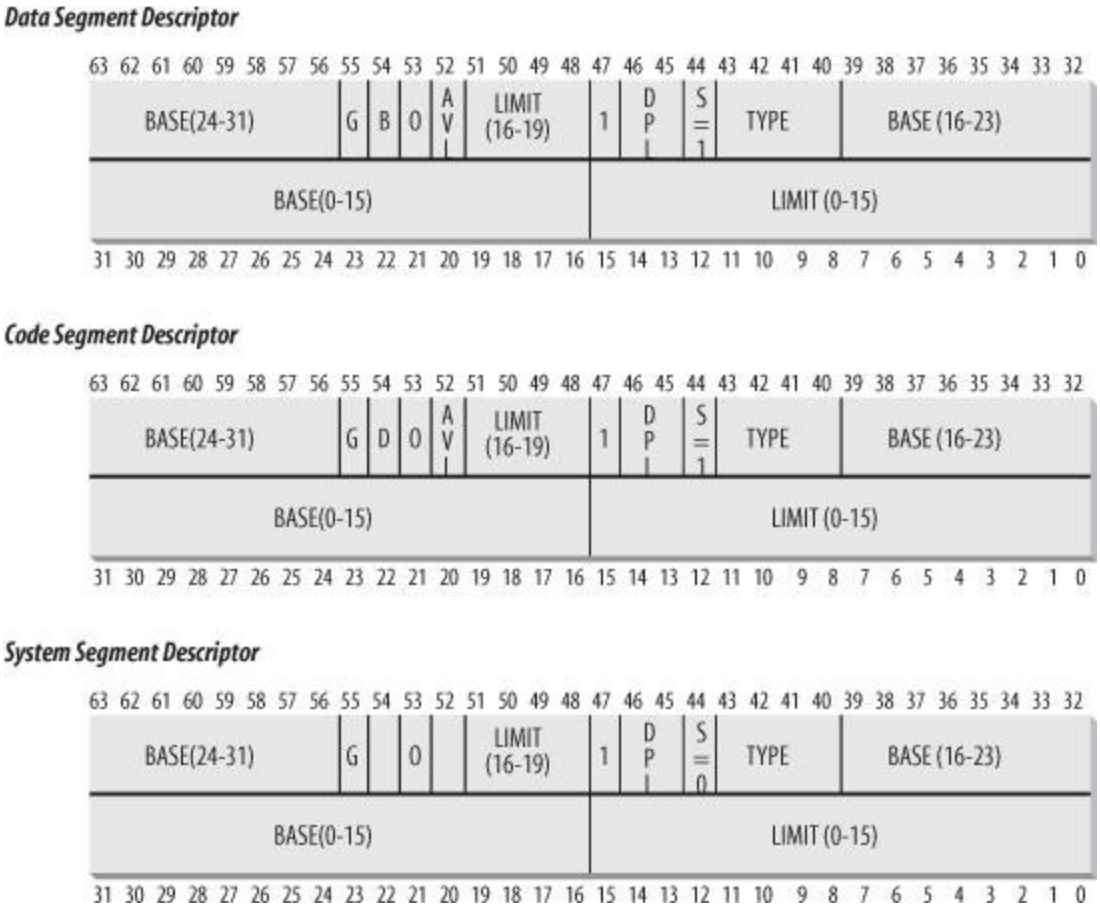



图 2-3：段描述符格式

有几种不同类型的段以及和它们对应的段描述符。下面列出了 Linux 中广泛被采用的类型：

代码段描述符

表示这个段描述符代表一个代码段；它可以放在 GDT 或 LDT 中。该描述符置 S 标志（非系统段）。

数据段描述符

> 表示这个段描述符代表一个数据段，它可以放在GDT或LDT中。该描述符置S标志。栈段是通过通用的数据段实现的。

任务状态段描述符（TSSD）

表示这个段描述符代表一个任务状态段（Task State Segment，TSS），也就是说这个段用于保存处理器寄存器的内容（参见第三章中的“任务状态段”一节）。它只能出现在 GDT 中。根据相应的进程是否正在 CPU 上运行，其 Type 字段的值分别为 11 或 9。这个描述符的 S 标志置为 0。

局部描述符表描述符（LDTD）

表示这个段描述符代表一个包含 LDT 的段，它只出现在 GDT 中，相应的 Type 字段为 2，S 标志为 0。下一节说明 80x86 处理器是如何决定一个段描述符是存放在 GDT 中还是存放在进程的 LDT 中。

## 快速访问段描述符

我们回忆一下逻辑地址由 16 位段选择符和 32 位偏移量组成，段寄存器仅仅存放段选择符。

为了加速逻辑地址到线性地址的转换，80x86 处理器提供一种附加的非编程的寄存器（一个不能被程序员所设置的寄存器），供 6 个可编程的段寄存器使用。每一个非编程的寄存器含有 8 个字节的段描述符（在前一节已讲述），由相应的段寄存器中的段选择符来指定。每当一个段选择符被装入段寄存器时，相应的段描述符就由内存装入到对应的非编程 CPU 寄存器。从那时起，针对那个段的逻辑地址转换就可以不访问主存中的 GDT 或 LDT，处理器只需直接引用存放段描述符的 CPU 寄存器即可。仅当段寄存器的内容改变时，才有必要访问 GDT 或 LDT（参见图 2－4）。

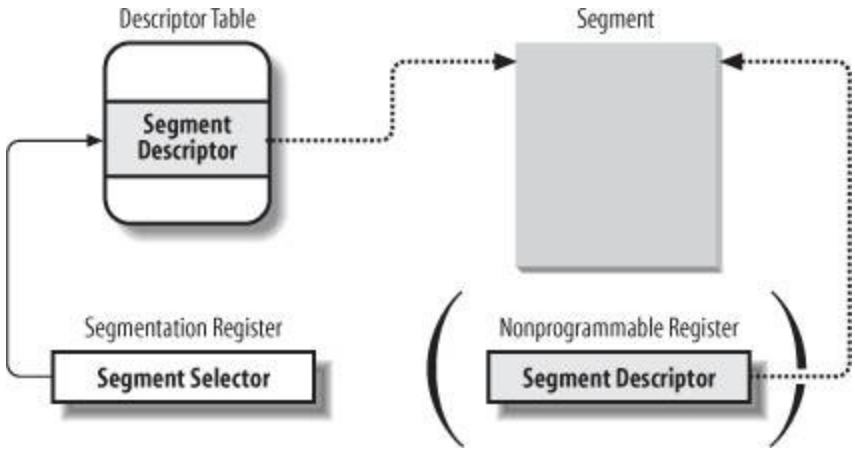



图 2-4：段选择符和段描述符

表 2-2 描述了任意段选择符所包含的三个字段。

表 2-2：段选择符字段

| 字段名 | 描述 |
| --- | --- |
| index | 指定了放在 GDT 或 LDT 中的相应段描述符的入口（在下面将做进一步的讲述）。 |
| TI | TI（(Table Indicator)标志：指明段描述符是在 GDT 中（TI=0）或在 LDT 中（TI=1）。 |

RPL　　请求者特权级：当相应的段选择符装入到 cs 寄存器中时指示出 CPU 当前的特权级；它还可以用于在访问数据段时有选择地削弱处理器的特权级（详情请参见 Intel 文档）。

由于一个段描述符是 8 字节长，它在 GDT 或 LDT 内的相对地址是由段选择符的最高 13 位的值乘以 8 得到的。例如：如果 GDT 在 0x00020000（这个值保存在 gdtr 寄存器中），且由段选择符所指定的索引号为 2，那么相应的段描述符地址是 0x00020000 + (2 ×8), 或 0x00020010。

GDT 的第一项总是设为 0。这就确保空段选择符的逻辑地址会被认为是无效的，因此引起一个处理器异常。能够保存在 GDT 中的段描述符的最大数目是 8191，即 $2^{13}$-1。

## 分段单元

图 2-5 详细显示了一个逻辑地址是怎样转换成相应的线性地址的。分段单元（segmentation unit）执行以下操作：

- 先检查段选择符的 TI 字段，以决定段描述符保存在哪一个描述符表中。TI 字段指明描述符是在 GDT 中（在这种情况下，分段单元从 gdtr 寄存器中得到 GDT 的线性基地址）还是在激活的 LDT 中（在这种情况下，分段单元从 ldtr 寄存器中得到 LDT 的线性基地址）。

- 从段选择符的 index 字段计算段描述符的地址，index 字段的值乘以 8（一个段描述符的大小），这个结果与 gdtr 或 ldtr 寄存器中的内容相加。

- 把逻辑地址的偏移量与段描述符 Base 字段的值相加就得到了线性地址。

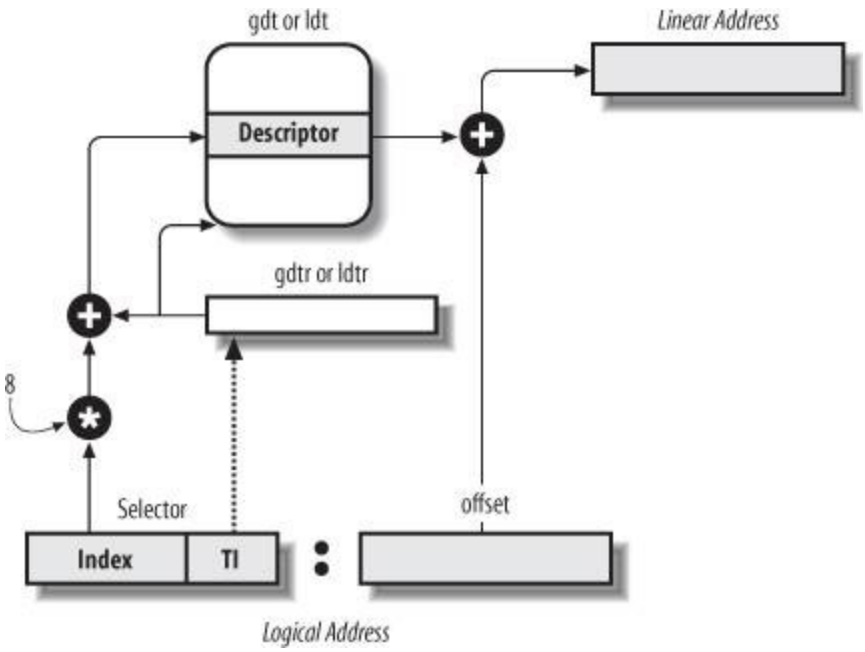


图 2-5：逻辑地址的转换

请注意，多亏了与段寄存器相关的不可编程寄存器，只有当段寄存器的内容被改变时才需要执行前两个操作。

## Linux 中的分段

80x86 微处理器中的分段鼓励程序员把他们的程序化分成逻辑上相关的实体，例如子程序或者全局与局部数据区。然而，Linux 以非常有限的方式使用分段。实际上，分段和分页在某种程度上有点多余，因为它们都可以划分进程的物理地址空间：分段可以给每一个进程分配不同的线性地址空间，而分页可以把同一线性地址空间映射到不同的物理空间。与分段相比，Linux 更喜欢使用分页方式，因为：

- 当所有进程使用相同的段寄存器值时，内存管理变得更简单，也就是说它们能共享同样的一组线性地址。
- Linux 设计目标之一是可以把它移植到绝大多数流行的处理器平台上。然而， RISC 体系结构对分段的支持很有限。

2.6 版的 Linux 只有在 80x86 结构下才需要使用分段。

运行在用户态的所有 Linux 进程都使用一对相同的段来对指令和数据寻址。这两个段就是所谓的用户代码段和用户数据段。类似地，运行在内核态的所有 Linux 进程都使用一对相同的段对指令和数据寻址：它们分别叫做内核代码段和内核数据段。表 2-3 显示了这四个重要段的段描述符字段的值。

表 2-3：

四个主要的 Linux 段的段描述符字段的值

| 段 | Base | G | Limit | S | Type | DPL | D/B | P |
|---|---|---|---|---|---|---|---|---|
| 用户代码段 | 0x00000000 | 1 | 0xfffff | 1 | 10 | 3 | 1 | 1 |
| 用户数据段 | 0x00000000 | 1 | 0xfffff | 1 | 2 | 3 | 1 | 1 |
| 内核代码段 | 0x00000000 | 1 | 0xfffff | 1 | 10 | 0 | 1 | 1 |
| 内核数据段 | 0x00000000 | 1 | 0xfffff | 1 | 2 | 0 | 1 | 1 |

相应的段描述符由宏__USER_CS，__USER_DS，__KERNEL_CS，和__KERNEL_DS 分别定义。例如，为了对内核代码段寻址，内核只需要把这个宏产生的值装进 cs 段寄存器即可。

注意，与段相关的线性地址从 0 开始，达到 $2^{32}$-1 的寻址限长。这就意味着在用户态或内核态下的所有进程可以使用相同的逻辑地址。

所有段都从 0x00000000 开始，这可以得出另一个重要结论，那就是在 Linux 下逻辑地址与线性地址是一致的，即逻辑地址的偏移量字段的值与相应的线性地址的值总是一致的。

如前所述，CPU 的当前特权级（CPL）反映了进程是在用户态还是内核态，并由存放在 cs 寄存器中的段选择符的 RPL 字段指定。只要当前特权级被改变，一些段寄存器必须相应地更新。例如，当 CPL=3 时（用户态），ds 寄存器必须含有用户数据段的段选择符，而当 CPL=0 时，ds 寄存器必须含有内核数据段的段选择符。

类似的情况也出现在 ss 寄存器中。当 CPL 为 3 时，它必须指向一个用户数据段中的用户栈，而当 CPL 为 0 时，它必须指向内核数据段中的一个内核栈。当从用户态切换到内核态时，Linux 总是确保 ss 寄存器装有内核数据段的段选择符。

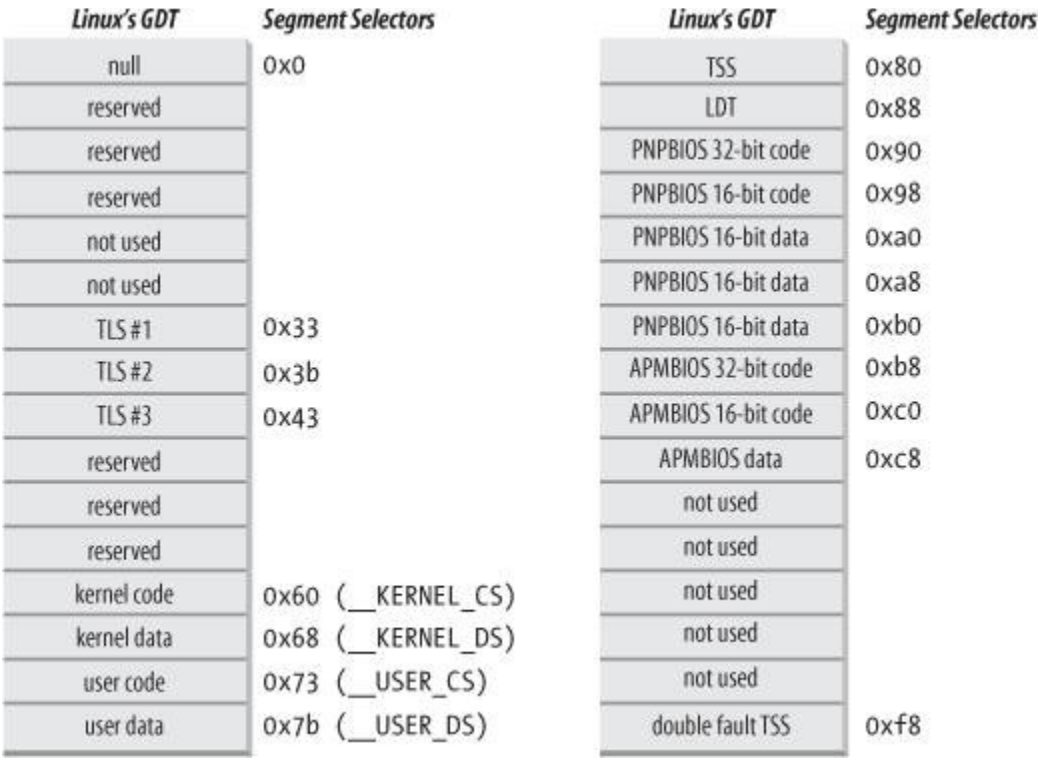
当对指向指令或者数据结构的指针进行保存时，内核根本不需要为其设置逻辑地址的段选择符，因为 cs 寄存器就含有当前的段选择符。例如，当内核调用一个函数时，它执行一条 call 汇编语言指令，该指令仅指定它逻辑地址的偏移量部分，而段选择符不用设置，其隐含在 cs 寄存器中了。因为“在内核态执行” 的段只有一种，叫做代码段，由宏__KERNEL_CS 定义，所以只要当 CPU 切换入内核态时足可以将__KERNEL_CS 装载入 cs。同样的道理也适用于指向内核数据结构的指针（隐含地使用 ds 寄存器）以及指向用户数据结构的指针（内核显式地使用 es 寄存器）。

除了刚才描述的 4 个段以外，Linux 还使用了其它几个专门的段。我们将在下一节讲述 Linux GDT 的时候介绍它们。

**Linux GDT**

在单处理器系统中只有一个 GDT，而在多处理器系统中每个 CPU 对应一个 GDT。所有的 GDT 都存放在 cpu_gdt_table 数组中，而所有 GDT（当初始化 gdtr 寄存器时使用）的地址和它们的大小存放在 cpu_gdt_descr 数组中。如果你到源代码索引中查看，可以看到这些符号都在文件 arch/i386/kernel/head.S 中被定义。本书中的每一个宏、函数和其它符号都列在源代码索引中，所以你能在源代码中很方便地找到它们。

图 2-6 是 GDT 的布局示意图。每个 GDT 包含 18 个段描述符和 14 个空的，未使用的，或保留的项。插入未使用的项的目的是为了以便经常一起访问的描述符能够处于同一个 32 字节的硬件高速缓存行中（参见本章后面“硬件高速缓存“一节）。



| Linux's GDT | Segment Selectors |
|---|---|
| null | 0x0 |
| reserved | |
| reserved | |
| reserved | |
| not used | |
| not used | |
| TLS #1 | 0x33 |
| TLS #2 | 0x3b |
| TLS #3 | 0x43 |
| reserved | |
| reserved | |
| reserved | |
| kernel code | 0x60 (__KERNEL_CS) |
| kernel data | 0x68 (__KERNEL_DS) |
| user code | 0x73 (__USER_CS) |
| user data | 0x7b (__USER_DS) |

| Linux's GDT | Segment Selectors |
|---|---|
| TSS | 0x80 |
| LDT | 0x88 |
| PNPBIOS 32-bit code | 0x90 |
| PNPBIOS 16-bit code | 0x98 |
| PNPBIOS 16-bit data | 0xa0 |
| PNPBIOS 16-bit data | 0xa8 |
| PNPBIOS 16-bit data | 0xb0 |
| APMBIOS 32-bit code | 0xb8 |
| APMBIOS 16-bit code | 0xc0 |
| APMBIOS data | 0xc8 |
| not used | |
| not used | |
| not used | |
| not used | |
| not used | |
| double fault TSS | 0xf8 |

图 2-6：全局描述符表

每一个 GDT 中包含的 18 个段描述符指向下列的段：

- 用户和内核态下的代码和数据段共四个（参见前面一节）。

- 任务状态段（TSS），每个处理器有一个。每个 TSS 相应的线性地址空间都是内核数据段相应线性地址空间的一个小子集。所有的任务状态段都顺序地存放在 init_tss 数组中；值得特别说明的是，第 n 个 CPU 的 TSS 描述符的 Base 字段指向 init_tss 数组的第 n 个元素。G（粒度）标志被清 0，而 Limit 字段置为 0xeb，因为 TSS 段是 236 字节长。Type 字段置为 9 或 11（可用的 32 位 TSS），且 DPL 置为 0，因为不允许用户态下的进程访问 TSS 段。在第三章“任务状态段”一节你可以找到 Linux 是如何使用 TSS 的细节。

- 一个包括缺省局部描述符表的段，这个段通常是被所有进程共享的段（参见下一节）。

- 三个局部线程存储段（TLS）：这中机制允许多线程应用程序使用最多三个局部于线程的数据段。系统调用 set_thread_area( )和 get_thread_area( ) 分别为正在执行的进程创建和撤消一个 TLS 段。

- 与高级电源管理（AMP）相关的三个段：由于 BIOS 代码使用段，所以当 Linux APM 驱动程序调用 BIOS 函数来获取或者设置 APM 设备的状态时，就可以使用自定义的代码段和数据段。

- 与支持即插即用（PnP）功能的 BIOS 服务程序相关的 5 个段：在前一种情况下，就像前述与高级电源管理（AMP）相关的三个段的情况一样，由于 BIOS 例程使用段，所以当 Linux 的即插即用设备驱动程序调用 BIOS 函数来检测即插即用设备使用的资源时，就可以使用自定义的代码段和数据段。

- 被内核用来处理“双重错误”（译注：处理一个异常时可能会引发另一个异常。在这种情况下产生双重错误）异常的特殊 TSS 段（参见第四章的“异常”一节）。

如前所述，系统中每个处理器都拥有一个 GDT 副本。除少数几种情况以外，所有 GDT 的副本都存放相同的表项。首先，每个处理器都有它自己的 TSS 段，因此其对应的 GDT 项不同。此外，GDT 中只有少数项可能依赖于 CPU 正在执行的进程（LDT 和 TLS 段描述符）。最后，在某些情况下，处理器可能临时修改 GDT 副本里的某个项；例如，当调用 APM 的 BIOS 例程时就会发生这种情况。

**Linux** 局部描述符表

大多数用户态下的 Linux 程序不使用局部描述符表，这样内核就定义了一个缺省的 LDT 供大多数进程共享。缺省的局部描述符表存放在 default_ldt 数组中。它包含 5 个项，但内核仅仅有效地使用了其中的两个：用于 iBCS 可执行文件的调用门和 Solaris/x86 可执行文件的调用门（参见第二十章的“执行域”一节）。调用门是 80x86 微处理器提供的一种机制，用于在调用预定义函数时改变 CPU 的特权级，由于我们不会再深入地讨论它们，所以请参考 Intel 文档以获取更多详情。

在某些情况下，进程仍然需要创建自己的局部描述符表。这对有些应用程序很有用，像 Wine 那样的程序，它们执行面向段的微软 Windows 应用程序。modify_ldt( )系统调用允许进程创建自己的局部描述符表。

任何被 modify_ldt( )创建的自定义局部描述符表仍然需要它自己的段。当处理器开始执行拥有自定义局部描述符表的进程时，那么该 CPU 的 GDT 副本中的局部描述符表项也就相应地被修改了。

用户态下的程序同样也利用 modify_ldt( )来分配新的段，但内核却从不使用这些段，它也不需要了解相应的段描述符，因为这些段描述符被包含在进程自定义的局部描述符表中了。

## 硬件中的分页

分页单元（paging unit）把线性地址转换成物理地址。其中的一个关键任务是把所请求的访问类型与线性地址的访问权限相比较，如果这次内存访问是无效的，就产生一个缺页异常（参见第四章和第八章）。

为了效率起见，线性地址被分成以固定长度为单位的组，称为页（page）。页内部连续的线性地址被映射到连续的物理地址中。这样，内核可以指定一个页的物理地址和其存取权限，而不用指定页所包含的全部线性地址的存取权限。我们遵循通常习惯，使用术语“页”既指一组线性地址，又指包含在这组地址中的数据。

分页单元把所有的RAM分成固定长度的页框（page frame）（有时叫做物理页）。每一个页框包含一个页

（page），也就是说一个页框的长度与一个页的长度一致。页框是主存的一部分，因此也是一个存储区域。区分一页和一个页框是很重要的，前者只是一个数据块，可以存放在任何页框或磁盘中。

把线性地址映射到物理地址的数据结构称为页表（page table）。页表存放在主存中，并在启用分页单元之前必须由内核对页表进行适当的初始化。

从 80386 开始，所有的 80x86 处理器都支持分页，它通过设置 cr0 寄存器的 PG 标志启用。当 PG=0 时，线性地址就被解释成物理地址。

## 常规分页

从 80386 起，Intel 处理器的分页单元处理 4KB 的页。

32 位的线性地址被分成 3 个字段：
目录（Directory）
　　最高 10 位
页表（Table）
　中间 10 位
偏移量（Offset）
　最低 12 位

线性地址的转换分两步完成，每一步都基于一种转换表，第一种转换表称为页目录表（page directory），第二种转换表称为页表（page table）。

使用这种二级模式的目的在于减少每个进程页表所需 RAM 的数量。如果使用简单的一级页表，那将需要高达 $2^{20}$ 个表项（也就是，在每项 4 个字节时，需要 4M RAM）来表示每个进程的页表（如果进程使用全部 4GB 线性地址空间），即使一个进程并不使用那个范围内的所有地址。二级模式通过只为进程实际使用的那些虚拟内存区请求页表来减少内存。

每个活动进程必须有一个分配给它的页目录。不过，没有必要马上为进程的所有页表都分配内存。当进程实际需要一个页表时才给该页表分配RAM会更为有效率。

正在使用的页目录的物理地址存放在控制寄存器 cr3 中。线性地址内的 Directory 字段决定页目录中的目录项，而目录项指向适当的页表。地址的 Table 字段依次又决定页表中的表项，而表项含有页所在页框的物理地址。Offset 字段决定页框内的相对位置（见图 2-7）。由于它是~~一个~~ 12 位长~~的字段~~，故每一页含有 4096 字节的数据。

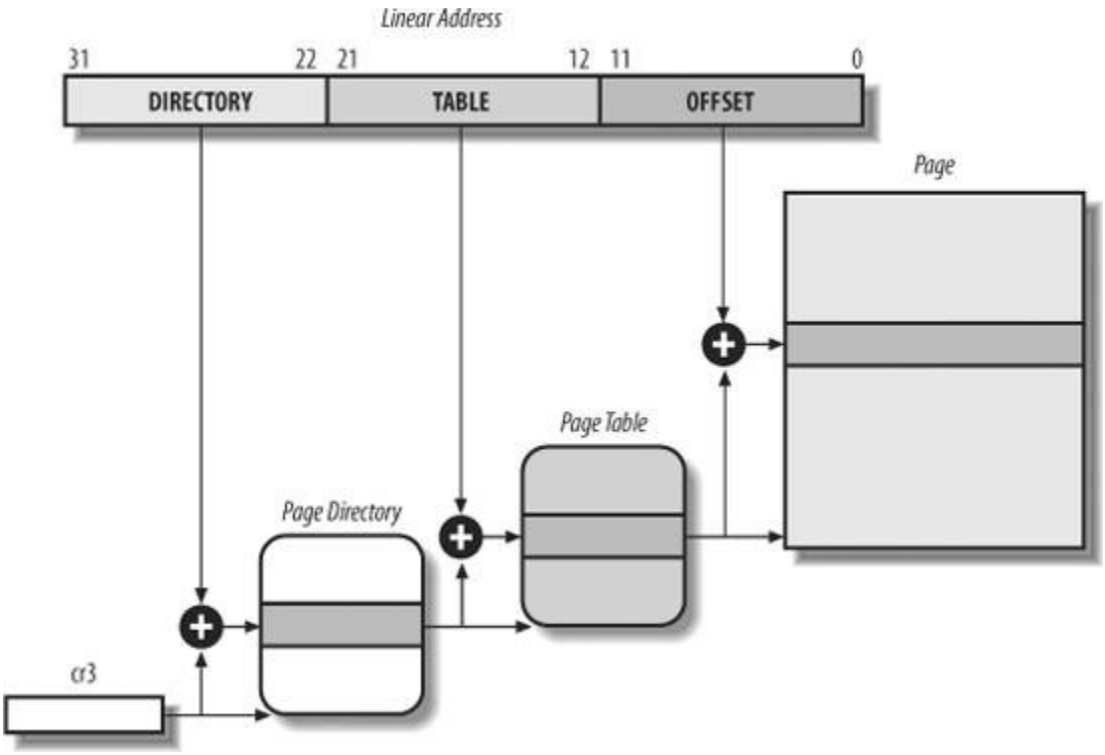



图 2-7：80x86 处理器的分页

Directory字段和Table字段都是10位长，因此页目录和页表都可以多达1024项。那么一个页目录可以寻址到高达1024×1024×4096=$2^{32}$个存储单元，这和你对32位地址所期望的~~寻址范围~~一样。

页目录项和页表项有同样的结构，每项都包含下面的字段：

Present标志

> 如果被置为 1，所指的页（或页表）就在主存中；如果该标志为 0，则这一页不在主存中，此时这个表项剩余的位可由操作系统用于自己的目的。如果执行一个地址转换所需的页表项或页目录项中 Present 标志被清 0，那么分页单元就把该线性地址存放在控制寄存器 cr2 中，并产生 14 号异常：缺页异常。（我们将在第十七章中看到 Linux 如何使用这个字段）。

Field包含页框物理地址最高*20*位的字段

> 由于每一个页框有 4KB 的容量，它的物理地址必须是 4096 的倍数，因此物理地址的最低 12 位总是为 0。如果这个字段指向一个页目录，相应的页框就含有一个页表，如果它指向一个页表，相应页框就含有一页数据。

Accessed 标志

> 每当分页单元对相应页框进行寻址时就设置这个标志。当选中的页被交换出去时，这一标志就可以由操作系统使用。分页单元从来不重置这个标志；而是必须由操作系统去做。

Dirty 标志

只应用于页表项中。每当对一个页框进行写操作时就设置这个标志。与 Accessed 标志一样，当选中的页被交换出去时，这一标志就可以由操作系统使用。分页单元从来不重置这个标志，而是必须由操作系统去做。

Read/Write 标志

含有页或页表的存取权限（Read/Write 或 Read）（参阅本章后面“硬件保护方案”一节）。

User/Supervisor 标志

含有访问页或页表所需的特权级（参见后面的“硬件保护方案”一节）。

PCD 和PWT标志

硬件高速缓存处理页或页表的控制方式（参见本章后面“硬件高速缓存”一节）。

Page Size 标志

只应用于页目录项。如果设置为 1，页目录项指的是 2MB 或 4MB 的页框（参见下一节）。

Global 标志

只应用于页表项。这个标志是在 Pentium Pro 引入的，用来防止常用页从 TLB（译注 1）高速缓存中刷新出去（参阅本章后面“翻译后备缓冲器（TLB）“一节）。只有在 cr4 寄存器的页全局启用（Page Global Enable，PGE）标志置位时这个标志才起作用。

## 扩展分页

从 Pentium 模式开始， 80x86 微处理器引入了扩展分页（extended paging），它允许页框大小为 4MB 而不是 4KB（参见图 2-8）。扩展分页用于把大段连续的线性地址转换成相应的物理地址，在这些情况下，内核可以不用中间页表进行地址转换，从而节省内存并保留 TLB 项（参阅“翻译后备缓冲器“一节）。

---

译注 1：TLB 的全称为 Translation Lookaside Buffer，这是 IBM 的叫法，有时也叫联想存储器（Associative Memory），俗称“快表”。

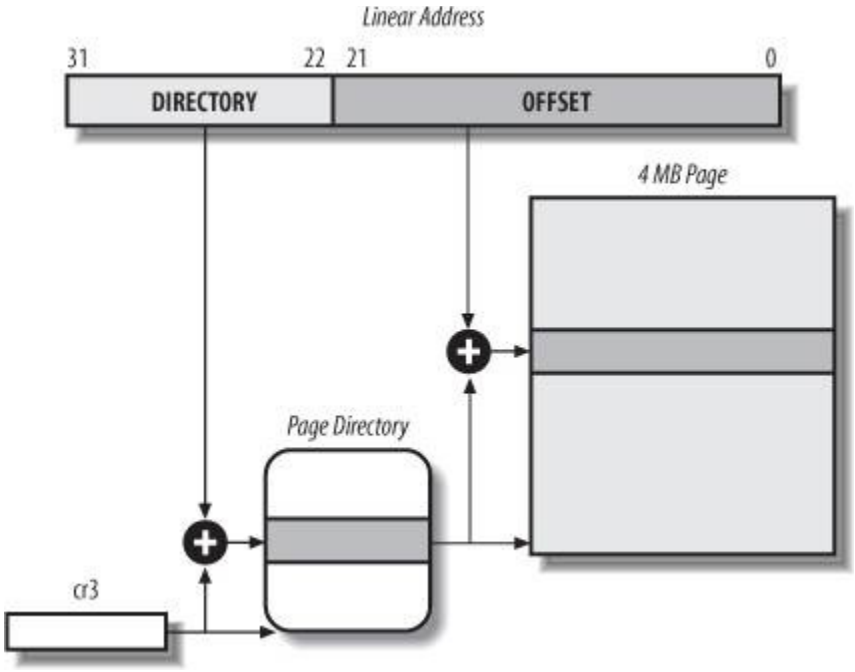



图 2-8：扩展分页

正如前面所述，通过设置页目录项的 Page Size 标志启用扩展分页功能。在这种情况下，分页单元把 32 位线性地址分成两个字段：

Directory
最高 10 位
Offset
其余 22 位

扩展分页和正常分页的页目录项基本相同，除了：

- Page Size 标志必须设置。
- 20 位物理地址字段只有最高 10 位是有意义的。这是因为每一个物理地址都是在以 4MB 为边界的地方开始的，故这个地址的最低 22 位为 0。

通过设置 cr4 处理器寄存器的 PSE 标志能使扩展分页与常规分页共存。

## 硬件保护方案

分页单元和分段单元的保护方案不同。尽管 80x86 处理器允许一个段使用四种可能的特权级别，但与页和页表相关的特权级只有两个，因为特权由前面“常规分页”一节中所提到的 User/Supervisor 标志所控制。若这个标志为 0，只有当 CPL 小于 3（这意味着对于 Linux 而言，处理器处于内核态）时才能对页寻址；若该标志

为 1，则总能对页寻址。

此外，与段的三种存取权限（读，写，执行）不同的是，页的存取权限只有两种(读，写)。如果页目录项或页表项的 Read/Write 标志等于 0，说明相应的页表或页是只读的，否则是可读写的（注1）。

## 常规分页举例

这个简单的例子将有助于阐明常规分页是如何工作的。我们假定内核已给一个正在运行的进程分配的线性地址空间范围是 0x20000000 到 0x2003ffff（注2）。这个空间正好由 64 页组成。我们不必关心包含这些页的页框的物理地址，事实上，其中的一些页甚至可能不在主存中。我们只关注页表项中剩余的字段。

让我们从分配给进程的线性地址的最高 10 位开始。这两个地址都以 2 开头后面跟着 0，因此高 10 位有相同的值，即 0x080 或十进制的 128。因此，这两个地址的 Directory 字段都指向进程页目录的第 129 项。相应的目录项中必须包含分配给该进程的页表的物理地址（参见图 2-9）。如果没有给这个进程分配其它的线性地址，页目录的其余 1023 项都填为 0。

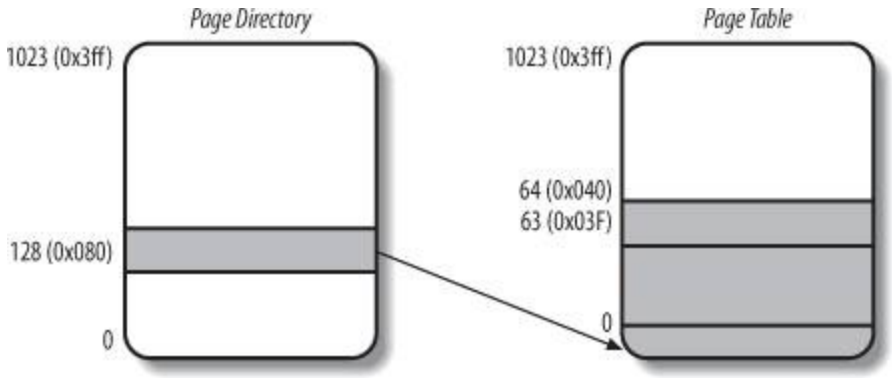



图 2-9：分页的例子

中间 10 位的值（即 Table 字段的值）范围从 0 到 0x03f，或十进制的从 0 到 63。因而只有页表的前 64 个表项是有意义的，其余 960 表项都填 0。

假设进程需要读线性地址 0x20021406 中的字节。这个地址由分页单元按下面的方法处理:

1. Directory 字段的 0x80 用于选择页目录的第 0x80 目录项，此目录项指向和该进程的页相关的页表。
2. Table 字段 0x21 用于选择页表的第 0x21 表项，此表项指向包含所需页的页框。
3. 最后，Offet 字段 0x406 用于在目标页框中读偏移量为 0x406 中的字节。

---

注1：新的 Intel Pentium 4 处理器在每个 64 位页表项中增加了一个 NX（No eXecute）标志（必须激活 PAE，参见本章后面的“物理地址扩展（PAE）”一节）。Linux 2.6.11 支持这个硬件特性。

注2：正如我们在后面章节所看到的那样，3GB 线性地址空间是一个上限，但是用户态进程只允许引用其中的一个子集。

如果页表第 0x21 表项的 Present 标志为 0，则此页就不在主存中；在这种情况下，分页单元在线性地址转换的同时产生一个缺页异常。无论何时，当进程试图访问限定在 0x20000000 到 0x2003ffff 范围之外的线性地址时，都将产生一个缺页异常，因为这些页表项都填充了 0，尤其是它们的 Present 标志都被清 0。

## 物理地址扩展（**PAE**）分页机制

处理器所支持的 RAM 容量受连接到地址总线上的地址管脚数限制。早期 Intel 处理器从 80386 到 Pentium 使用 32 位物理地址。从理论上讲，这样的系统上可以安装高达 4GB 的 RAM；而实际上，由于用户进程线性地址空间的需要，内核不能直接对 1GB 以上的 RAM 进行寻址，我们将会在后面“Linux 中的分页”一节中看到这一点。

然而，大型服务器需要大于 4GB 的 RAM 来同时运行数以千计的进程，近几年这对 Intel 造成了压力，所以必须扩展 32 位 80386 结构所支持的 RAM 容量。

Intel 通过在它的处理器上把管脚数从 32 增加到 36 已经满足了这些需求。从 Pentium Pro 开始，Intel 所有处理器现在寻址能力达 $2^{36}$ = 64GB。不过，只有引入一种新的分页机制把 32 位线性地址转换为 36 位物理地址才能使用所增加的物理地址。

从 Pentium Pro 处理器开始，Intel 引入一种叫做物理地址扩展（Physical Address Extension，PAE）的机制。另外一种叫做页大小扩展（Page Size Extension (PSE-36)）的机制在 Pentium III 处理器中引入，但是 Linux 并没有采用这种机制，因而我们在本书中不做进一步讨论。

激活 PAE 是通过设置 cr4 控制寄存器中的物理地址扩展（PAE）标志。页目录项中的页大小标志 PS 启用大尺寸页（在 PAE 启用时为 2MB）。

Intel 为了支持 PAE 已经改变了分页机制。

- 64GB 的 RAM 被分为 $2^{24}$ 个页框，页表项的物理地址字段从 20 位扩展到了 24 位。因为 PAE 页表项必须包含 12 个标志位（在前面“常规分页”一节已描述）和 24 个物理地址位，总数之和为 36，页表项大小从 32 位变为 64 位增加了一倍。结果，一个 4KB 的页表包含 512 个表项而不是 1024 个表项。

- 引入一个叫做页目录指针表（Page Directory Pointer Table，PDPT）的页表新级别，它由 4 个 64 位表项组成。

- cr3 控制寄存器包含一个 27 位的页目录指针表（PDPT）基地址字段。因为 PDPT 存放在 RAM 的前 4GB 中，并在 32 字节（$2^5$）的倍数上对齐，因此 27 位足以表示这种表的基地址。

- 当把线性地址映射到 4 Kb 的页时（页目录项中的 PS 标志清 0），32 位线性地址按下列方式解释：

cr3
: 指向一个PDPT

位 *31-30*
: 指向 PDPT 中 4 个项中的一个

位 *29-21*
: 指向页目录中 512 个项中的一个

位 *20-12*
: 指向页表中 512 项中的一个

位 *11-0*
: 4KB 页中的偏移量

- 当把线性地址映射到 2MB 的页时（页目录项中的 PS 标志清 0），32 位线性地址按下列方式解释：

cr3
: 指向一个PDPT

位 *31-30*
: 指向 PDPT 中 4 个项中的一个

位 *29-21*
: 指向页目录中 512 个项中的一个

位 20-0
: 2MB 页中的偏移量

总之，一旦 cr3 被设置，就可能寻址高达 4GB RAM。如果我们希望对更多的 RAM 寻址，就必须在 cr3 中放置一个新值，并改变 PDPT 的内容。然而，使用 PAE 的主要问题是线性地址仍然是 32 位长。这就迫使编程人员用同一线性地址映射不同的 RAM 区。在后面的“当 RAM 大于 4096MB 时的最终内核页表”一节中，我们将描述启用 PAE 时 Linux 如何初始化页表。很明显，PAE 并没有扩大进程的线性地址空间，因为它只处理物理地址。此外，只有内核能够修改进程的页表，所以在用户态下运行的进程不能使用大于 4GB 的物理地址空间。另一方面值得注意的是，PAE 允许内核使用容量高达 64GB 的 RAM，从而显著增加了系统中的进程数量。

## 64 位系统中的分页

我们在前面几节已经看到，32 位微处理器普遍采用两级分页（[注3]）。然而两级分页并不适用于采用 64 位系统的计算机。让我们用一种思维实验来解释为什么：

[注3]：80x86 处理器中引入的第三级分页在当 PAE 激活时仅仅只是将页目录项和页表项的数目从 1024 个减少到了 512 个。这样就将页表项从 32 位扩大到了 64 位，于是它们能够存放物理地址的最高 24 位。

首先假设一个大小为 4KB 的标准页。因为 1KB 覆盖 $2^{10}$ 个地址的范围，4KB 覆盖 $2^{12}$ 个地址，所以偏移量字段是 12 位。这样线性地址就剩下 52 位分配给页表和页目录字段。如果我们现在决定仅仅使用 64 位中的 48 位来寻址（这个限制仍然使我们自在地拥有 256TB 的寻址空间！），剩下的 48-12 = 36 位将被分配给页表和页目录字段。如果我们现在决定为两个字段各预留 18 位，每个进程的页目录和页表都含有 $2^{18}$ 个项，即超过 256,000 个项。

由于这个原因，所有 64 位处理器的硬件分页系统都使用了额外的分页级别。使用的级别数量取决于处理器的类型。表 2-4 总结了一些 Linux 所支持 64 位平台使用的硬件分页系统的主要特征。对于与平台名称相关的硬件的简要描述请参见第一章的“硬件的依赖性”一节。

表 2-4：一些 64 位系统的分页级别

| 平台名称 | 页大小 | 寻址使用的位数 | 分页级别数 | 线性地址分级 |
|---|---|---|---|---|
| alpha | 8 KB[a] | 43 | 3 | 10 + 10 + 10 + 13 |
| ia64 | 4 KB[a] | 39 | 3 | 9 + 9 + 9 + 12 |
| ppc64 | 4 KB | 41 | 3 | 10 + 10 + 9 + 12 |
| sh64 | 4 KB | 41 | 3 | 10 + 10 + 9 + 12 |
| x86_64 | 4 KB | 48 | 4 | 9 + 9 + 9 + 9 + 12 |

[a] 该体系结构支持不同的页大小；我们选择了一种 Linux 支持的典型页大小。

稍后我们将会在本章的“Linux 中的分页”一节看到，Linux 成功地提供了一种通用分页模型，它适合于绝大多数所支持的硬件分页系统。

## 硬件高速缓存（**Hardware Cache**）

当今的微处理器时钟频率接近几个 GHZ，而动态 RAM（DRAM）芯片的存取时间是时钟周期的数百倍。这意味着，当从 RAM 中取操作数或向 RAM 中存放结果这样的指令执行时，CPU 可能等待很长时间。

为了缩小 CPU 和 RAM 之间的速度不匹配，引入了硬件高速缓存存储器（Hardware cache memory）。硬件高速缓存基于著名的局部性原理（*locality principle*），该原理既适用程序结构和也适用数据结构。这表明由于程序的循环结构及把相关数组可以组织成线性数组，最近最常用的相邻地址在最近的将来又被用到的可能性极大。因此，引入小而快的存储器来存放最近最常使用的代码和数据变得很有意义。为此，80x86 体系结构中引入了一个叫行（*line*）的新单位。行由几十个连续的字节组成，它们被以脉冲突发模式（burst mode）在慢速 DRAM 和快速的片上静态 RAM（SRAM）之间传送，用来实现高速缓存。

高速缓存再被细分为行的子集。在一种极端的情况下，高速缓存可以是直接映射的（direct mapped），这时主存中的一个行总是存放在高速缓存中完全相同的位置。在另一种极端情况下，高速缓存是充分关联的（fully associative），这意味着主存中的任意一个行可以存放在高速缓存中的任意位置。但是大多数高速缓存在某种程度上是 N-路组关联的（N-way set associative），意味着主存中的任意一个行可以存放在高速缓存 N 行中的任意一行中。例如，内存中的一个行可以存放到一个 2 路组关联高速缓存两个不同的行中。

如图 2-10 所示，高速缓存单元插在分页单元和主存储器之间。它包含一个硬件高速缓存存储器（hardware

cache memory）和一个高速缓存控制器（cache controller）。高速缓存存储器存放内存中真正的行。高速缓存控制器存放一个表项数组，每个表项对应高速缓存存储器中的一个行。每个表项有一个标签（tag）和描述高速缓存行状态的几个标志(flag)。这个标签(tag)由一些位组成，这些位让高速缓存控制器能够辨别由这个行当前所映射的内存单元。这种内存物理地址通常分为三组：最高几位对应标签，中间几位对应高速缓存控制器的子集索引，最低几位对应行内的偏移量。

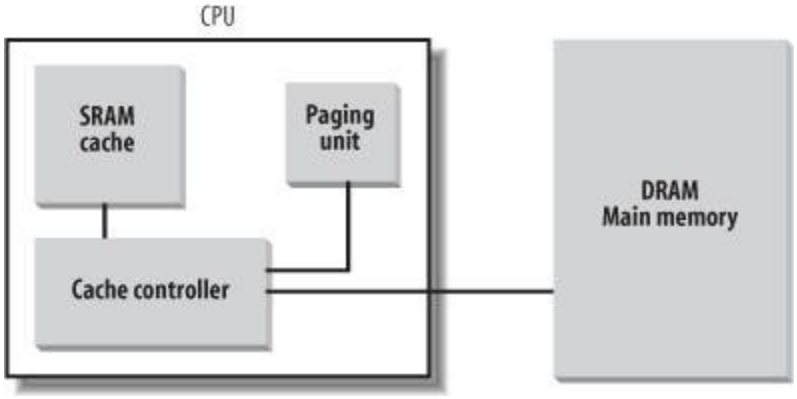



图 2-10：处理器硬件高速缓存

当访问一个 RAM 存储单元时，CPU 从物理地址中提取出子集的索引号并把子集中所有行的标签与物理地址的高几位相比较。如果发现某一个行的标签与这个物理地址的高位相同，则 CPU 命中一个高速缓存（cache hit）；否则，高速缓存没有命中（cache miss）。

当命中一个高速缓存时，高速缓存控制器的操作不同，具体取决于存取类型。对于~~一个~~读操作，控制器从高速缓存行中选择数据并送到 CPU 寄存器；RAM 不被访问且节约了 CPU 时间，因此，高速缓存系统起到了其应有的作用。对于~~一个~~写操作，控制器可能采用以下两个基本策略之一，分别称之为通写（write-through）和回写（write-back）。在通写中，控制器总是既写 RAM 也写高速缓存行，为了提高写操作的效率关闭高速缓存。回写方式只更新高速缓存行，不改变 RAM 的内容，提供了更快的功效。当然，回写结束以后，RAM 最终必须被更新。只有当 CPU 执行一条要求刷新高速缓存表项的指令时，或者当一个 FLUSH 硬件信号产生时（通常在高速缓存不命中~~发生~~之后），高速缓存控制器才把高速缓存行写回到 RAM 中。

当高速缓存没有命中时，高速缓存行被写回到内存中，如~~果~~有必要~~的话~~，把正确的行从 RAM 中取出放到高速缓存的表项中。

多处理器系统的每一个处理器都有一个单独的硬件高速缓存，因此它们需要额外的硬件电路用于保持高速缓存内容的同步。如图 2-10 所示，每个 CPU 都有自己的本地硬件高速缓存。但是，现在更新变得更耗时：只要一个 CPU 修改了它的硬件高速缓存，它就必须检查同样的数据是否包含在其它的硬件高速缓存中；如果是，它必须通知其它 CPU 用适当的值对其更新。常把这种活动叫做高速缓存侦听（cache snooping）。庆幸的是，所有这一切都在硬件级处理，内核无需关心。

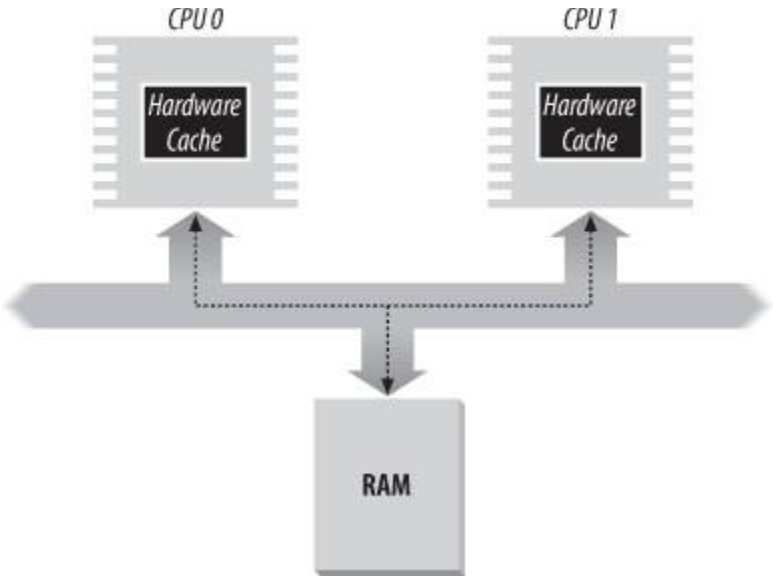



图 2-11：双处理器中的高速缓存

高速缓存技术正在快速向前发展。例如，第一代Pentium芯片包含一颗称为L1-cache的片上高速缓存。近期的芯片又包含另外~~一个~~的容量更大，速度较慢，称之为L2-cache，L3-cache的片上高速缓存。多级高速缓存之间的一致性是由硬件实现的。Linux忽略这些硬件细节并假定只有一个单独的高速缓存。

处理器的 cr0 寄存器的 CD 标志位用来启用或禁用高速缓存电路。这个寄存器中的 NW 标志指明高速缓存是使用通写还是回写策略。

Pentium 处理器高速缓存的另一个有趣的特点是，让操作系统把不同的高速缓存管理策略与每一个页框相关联。为此，每一个页目录项和每一个页表项都包含两个标志：PCD 标志指明当访问包含在这个页框中的数据时，高速缓存功能必须被启用还是禁用。PWT 标志指明当把数据写到页框时，必须使用的策略是回写策略还是通写策略。Linux 清除了所有页目录项和页表项中的 PCD 和 PWT 标志；结果是，对于所有的页框都启用高速缓存，对于写操作总是采用回写策略。

### 翻译后备缓冲器（TLB）

除了通用硬件高速缓存之外， 80x86 处理器还包含了另~~外~~一个称~~之~~为翻译后备缓冲器或 TLB（Translation Lookaside Buffer）的高速缓存用于加快线性地址的转换。当一个线性地址被第一次使用时，通过慢速访问 RAM 中的页表计算出相应的物理地址。同时，物理地址被存放在一个 TLB 表项（TLB entry）中，以便以后对同一个线性地址的引用就可以快速地得到转换。

在多处理系统中，每个 CPU 都有自己的 TLB，这叫做该 CPU 的本地 TLB。与硬件高速缓存相反，TLB 中的对应项不必同步，这是因为运行在现有 CPU 上的进程可以使同一线性地址与不同的物理地址发生联系。

当 CPU 的 cr3 控制寄存器被修改时，硬件自动使本地 TLB 中的所有项都无效，这是因为新的一组页表被启用而 TLB 指向的是旧数据。

## Linux 的分页

Linux 采用了一种同时适用于 32 位和 64 位系统的普通分页模型。正像前面“64 位系统中的分页”一节所解释的那样，两级页表对 32 位系统来说已经足够了，但 64 位系统需要更多数量的分页级别。直到 2.6.10 版本，Linux 采用三级分页的模型。从 2.6.11 版本开始，采用了四级分页模型（注4）。图 2-12 中展示的 4 种页表分别被称作：

- 页全局目录（Page Global Directory）
- 页上级目录（Page Upper Directory）
- 页中间目录（Page Middle Directory）
- 页表（Page Table）

页全局目录包含若干页上级目录的地址，页上级目录又依次包含若干页中间目录的地址，而页中间目录又包含若干页表的地址。每一个页表项指向一个页框。线性地址因此被分成五个部分。图 2-12 没有显示位数，因为每一部分的大小与具体的计算机体系结构有关。

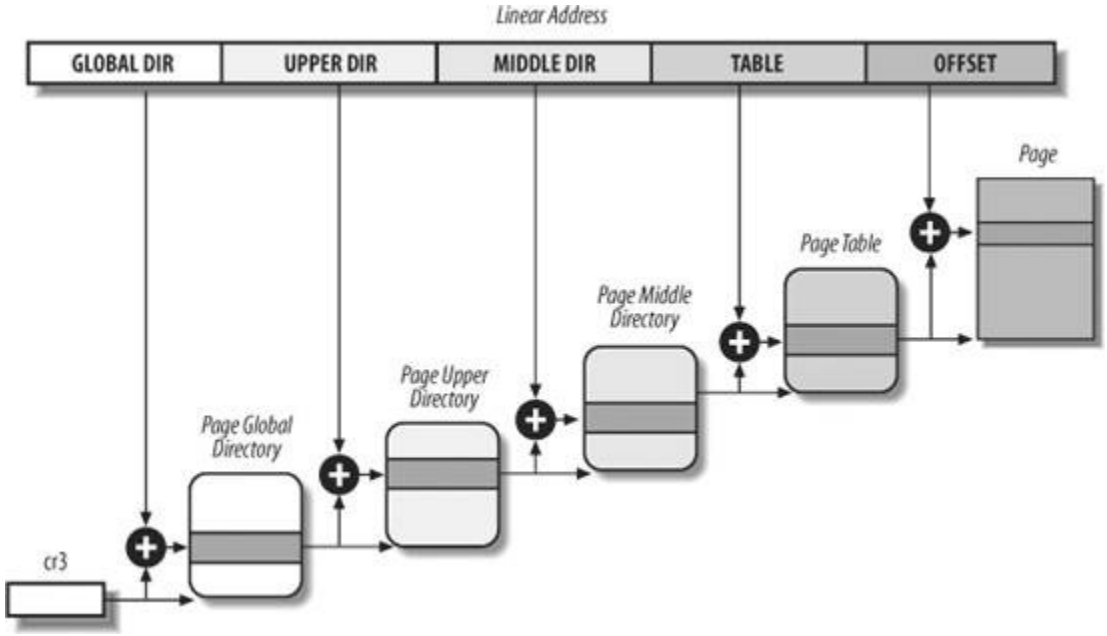



图 212：Linux 分页模型

对于没有启用物理地址扩展的32位系统，两级页表已经足够了。从本质上说Linux通过使“页上级目录”位和“页中间目录”位全为0，彻底取消了页上级目录和页中间目录字段。不过，页上级目录和页中间目录在指针序

注4：这个变化用来全力支持 x86_64 平台使用的对线性地址的位的划分（参见表 2－4）。

列中的位置被保留，以便同样的代码在32位系统和64位系统下都能使用。内核为页上级目录和页中间目录保留了一个位置，这是通过把它们的页目录项数设置为1，并把这两个目录项映射到页全局目录的一个合适的目录项而实现的。

启用了物理地址扩展的 32 位系统使用了三级页表。Linux 的页全局目录对应 80x86 的页目录指针表（PDPT），取消了页上级目录，页中间目录对应 80x86 的页目录，Linux 的页表对应 80x86 的页表。

最终，64 位系统使用三级还是四级分页取决于硬件对线性地址的位的划分（参见表 2-2）。

Linux 的进程处理很大程度上依赖于分页。事实上，线性地址到物理地址的自动转换使下面的设计目标变得可行：

- 给每一个进程分配一块不同的物理地址空间，这确保了可以有效地防止寻址错误。

- 区别页（即一组数据）和页框（即主存中的物理地址）之不同。这就允许存放在某个页框中的一个页，然后保存到磁盘上，以后重新装入这同一页时又被装在不同的页框中。这就是虚拟内存机制的基本要素（参见第十七章）。

在本章剩余的部分，为了具体起见，我们将涉及 80x86 处理器使用的分页机制。

我们将在第九章看到，每一个进程有它自己的页全局目录和自己的页表集。当发生进程切换时（参见第三章“进程切换”一节），Linux 把 cr3 控制寄存器的内容保存在前一个执行进程的描述符中，然后把下一个要执行进程的描述符的值装入 cr3 寄存器中。因此，当新进程重新开始在 CPU 上执行时，分页单元指向一组正确的页表。

把线性地址映射到物理地址虽然有点复杂，但现在已经成了一种机械式的任务。本章下面的几节中列举了一些比较单调乏味的函数和宏，它们检索内核为了查找地址和管理表格所需的信息；其中大多数函数只有一两行。也许现在你就想跳过这部分，但是知道这些函数和宏的功能是非常有用的，因为在贯穿本书的讨论中你会经常看到它们。

## 线性地址字段

下列宏简化了页表处理：

PAGE_SHIFT
: 指定 Offset 字段的位数；当用于 80x86 处理器时，它产生的值为 12。由于页内所有地址都必须放在 Offset 字段， 80x86 系统的页的大小是 $2^{12}$=4096 字节。 PAGE_SHIFT 的值为 12 可以看作以 2 为底的页大小的对数。这个宏由 PAGE_SIZE 使用以返回页的大小。最后， PAGE_MASK 宏产生的值为 0xfffff000，用以屏蔽 Offset 字段的所有位。

PMD_SHIFT
: 指定线性地址的 Offset 和 Table 字段的总位数；换句话说，是页中间目录项可以映射的区域大小的对数。PMD_SIZE 宏用于计算由页中间目录的一个单独表项所映射的区域大小，也就是一个页表的大小。PMD_MASK 宏用于屏蔽 Offset 字段与 Table 字段的所有位。

  当 PAE 被禁用时，PMD_SHIFT 产生的值为 22（来自 Offset 的 12 位加上来自 Table 的 10 位），PMD_SIZE 产生的值为 $2^{22}$ 或 4 MB， PMD_MASK 产生的值为 0xffc00000。相反，当 PAE 被激活时，PMD_SHIFT 产生的值为 21 (来自 Offset 的 12 位加上来自 Table 的 9 位)， PMD_SIZE 产生的值为 $2^{21}$ 或 2 MB 以及 PMD_MASK 产生的值为 0xffe00000。

  大型页不使用最后一级页表，所以产生大型页尺寸的 LARGE_PAGE_SIZE 宏等于 PMD_SIZE (2PMD_SHIFT)，而在大型页地址中用于屏蔽 Offset 字段和 Table 字段的所有位的 LARGE_PAGE_MASK 宏，就等于 PMD_MASK。

PUD_SHIFT
: 确定页上级目录项能映射的区域大小的对数。PUD_SIZE 宏用于计算页全局目录中的一个单独表项所能映射的区域大小。PUD_MASK 宏用于屏蔽 Offset 字段，Table 字段，Middle Air 字段和 Upper Air 字段的所有位。

  在 80x86 处理器上，PUD_SHIFT 总是等价于 PMD_SHIFT，而 PUD_SIZE 则等于 4MB 或 2MB。

PGDIR_SHIFT
: 确定页全局页目录项能映射的区域大小的对数。 PGDIR_SIZE 宏用于计算页全局目录中一个单独表项所能映射区域的大小。PGDIR_MASK 宏用于屏蔽 Offset, Table，Middle Air 及 Upper Air 的所有位。

  当 PAE 被禁止时，PGDIR_SHIFT 产生的值为 22（与 PMD_SHIFT 和 PUD_SHIFT 产生的值相同），PGDIR_SIZE 产生的值为 $2^{22}$ 或 4 MB，以及 PGDIR_MASK 产生的值为 0xffc00000。相反，当 PAE 被激活时，PGDIR_SHIFT 产生的值为 30 (12 位 Offset 加 9 位 Table 再加 9 位 Middle Air)，PGDIR_SIZE 产生的值为 $2^{30}$ 或 1 GB 以及 PGDIR_MASK 产生的值为 0xc0000000。

PTRS_PER_PTE, PTRS_PER_PMD, PTRS_PER_PUD 以及 PTRS_PER_PGD
: 用于计算页表、页中间目录、页上级目录和页全局目录表中表项的个数。当 PAE 被禁止时，它们产生的值分别为 1024，1，1 和 1024。当 PAE 被激活时，产生的值分别为 512，512，1 和 4。

## 页表处理

pte_t、pmd_t、pud_t 和 pgd_t 分别描述页表项、页中间目录项、页上级目录和页全局目录项的格式。当 PAE 被激活时它们都是 64 位的数据类型，否则都是 32 位数据类型。pgprot_t 是另一个 64 位（PAE 激活时）或 32 位（PAE 禁用时）的数据类型，它表示与一个单独表项相关的保护标志。

五个类型转换宏（__ pte、__ pmd、__ pud、__ pgd 和__ pgprot）把一个无符号整数转换成所需的类型。另外的五个类型转换宏（pte_val，pmd_val， pud_val， pgd_val 和 pgprot_val）执行相反的转换，即把上面提到的四种特殊的类型转换成一个无符号整数。

内核还提供了许多宏和函数用于读或修改页表表项：

- 如果相应的表项值为 0，那么，宏 pte_none、pmd_none、pud_none 和 pgd_none 产生的值为 1，否则产生的值为 0。

- 宏 pte_clear、pmd_clear、pud_clear 和 pgd_clear 清除相应页表的一个表项，由此禁止进程使用由该页表项映射的线性地址。ptep_get_and_clear( )函数清除一个页表项并返回前一个值。

- set_pte，set_pmd，set_pud 和 set_pgd 向一个页表项中写入指定的值。set_pte_atomic 与 set_pte 作用相同，但是当 PAE 被激活时它同样能保证 64 位的值能被原子地写入。

- 如果 a 和 b 两个页表项指向同一页并且指定相同的访问优先级，pte_same(a,b)返回 1，否则返回 0。

- 如果页中间目录项指向一个大型页（2MB 或 4MB），pmd_large(e)返回 1，否则返回 0。

宏 pmd_bad 由函数使用并通过输入参数传递来检查页中间目录项。如果目录项指向一个不能使用的页表，也就是说，如果至少出现以下条件中的一个，则这个宏产生的值为 1：

- 页不在主存中（Present 标志被清除）。

- 页只允许读访问(Read/Write 标志被清除)。

- Acessed 或者 Dirty 位被清除（对于每个现有的页表，Linux 总是强制设置这些标志）。

pud_bad 宏和 pgd_bad 宏总是产生 0。没有定义 pte_bad 宏，因为页表项引用一个不在主存中的页，一个不可写的页或一个根本无法访问的页都是合法的。

如果一个页表项的 Present 标志或者 Page Size 标志等于 1，则 pte_present 宏产生的值为 1，否则为 0。前面讲过页表项的 Page Size 标志对微处理器的分页部件来讲没有意义，然而，对于当前在主存中却又没有读、写或执行权限的页，内核将其 Present 和 Page Size 分别标记为 0 和 1。这样，任何试图对此类页的访问都会引起一个缺页异常，因为页的 Present 标志被清 0，而内核可以通过检查 Page Size 的值来检测到产生异常并不是因为缺页。

如果相应表项的 Present 标志等于 1，也就是说，如果对应的页或页表被装载入主存，pmd_present 宏产生的值为 1。pud_present 宏和 pgd_present 宏产生的值总是 1。

表 2-5 中列出的函数用来查询页表项中任意一个标志的当前值；除了 pte_file()外，其他函数只有在 pte_present 返回 1 的时候，才能正常返回页表项中任意一个标志。

表 2-5：读页标志的函数

| 函数名称 | 说明 |
| --- | --- |
| pte_user( ) | 读 User/Supervisor 标志。 |
| pte_read( ) | 读 User/Supervisor 标志（表示 80x86 处理器上的页不受读的保护）。 |
| pte_write( ) | 读 Read/Write 标志。 |
| pte_exec( ) | 读 User/Supervisor 标志（80x86 处理器上的页不受代码执行的保护）。 |
| pte_dirty( ) | 读 Dirty 标志。 |
| pte_young( ) | 读 Accessed 标志。 |
| pte_file( ) | 读 Dirty 标志（当 Present 标志被清除而 Dirty 标志被设置时，页属于一个非线性磁盘文件映射，参考第十六章）。 |

表 2-6 列出的另一组函数用于设置页表项中各标志的值。

表 2-6：设置页标志的函数

| 函数名称 | 说明 |
| --- | --- |
| mk_pte_huge( ) | 设置页表项中的 Page Size 和 Present 标志。 |
| pte_wrprotect( ) | 清除 Read/Write 标志。 |
| pte_rdprotect( ) | 清除 User/Supervisor 标志。 |
| pte_exprotect( ) | 清除 User/Supervisor 标志。 |
| pte_mkwrite( ) | 设置 Read/Write 标志。 |
| pte_mkread( ) | 设置 User/Supervisor 标志。 |
| pte_mkexec( ) | 设置 User/Supervisor 标志。 |
| pte_mkclean( ) | 清除 Dirty 标志。 |

pte_mkdirty( ) 设置 Dirty 标志。

pte_mkold( ) 清除 Accessed 标志（把此页标记为未访问）。

pte_mkyoung( ) 设置 Accessed 标志（把此页标记为访问过）。

pte_modify(p,v) 把页表项 p 的所有访问权限设置为指定的值 v。

ptep_set_wrprotect( ) 与 pte_wrprotect( )类似，但作用于指向页表项的指针。

ptep_set_access_flags( ) 如果 Dirty 标志被设置为 1 则将页的访问权设置为指定的值，并调用 flush_tlb_page()函数（参见本章稍后的“翻译后备缓冲器”一节）。

ptep_mkdirty( ) 与 pte_mkdirty( )类似，但作用于指向页表项的指针。

ptep_test_and_clear_dirty( ) 与 pte_mkclean( )类似，但作用于指向页表项的指针并返回 Dirty 标志的旧值。

ptep_test_and_clear_young( ) 与 pte_mkold( )类似，但作用于指向页表项的指针并返回 Accessed 标志的旧值。

现在，我们来讨论表 2-7 中列出的宏，它们把一个页地址和一组保护标志组合成页表项，或者执行相反的操作，从一个页表项中提取出页地址。请注意这其中的一些宏对页的引用是通过 “页描述符”的线性地址（参见第 8 章“页描述符”一节），而不是通过该页本身的线性地址。

表 2-7:对页表项操作的宏

| 宏名称 | 说明 |
| --- | --- |
| pgd_index(addr) | 找到线性地址 addr 对应的的目录项在页全局目录中的索引（相对位置）。 |
| pgd_offset(mm, addr) | 接收内存描述符地址 mm（参见第九章）和线性地址 addr 作为参数。这个宏产生地址 addr 在页全局目录中相应表项的线性地址；通过内存描述符 mm 内的一个指针可以找到这个页全局目录。 |
| pgd_offset_k(addr) | 产生主内核页全局目录中的某个项的线性地址，该项对应于地址 addr（参见稍后“内核页表”一节）。 |
| pgd_page(pgd) | 通过页全局目录项 pgd 产生页上级目录所在页框的页描述符地址。在两级或三级分页系统中，该宏等价于 pud_page()，后者应用于页上级目录项。 |

pud_offset(pgd, addr)　　参数为指向页全局目录项的指针 pgd 和线性地址 addr。这个宏产生页上级目录中目录项 addr 对应的线性地址。在两级或三级分页系统中，该宏产生 pgd，即一个页全局目录项的地址。

pud_page(pud)　　通过页上级目录项 pud 产生相应的页中间目录的线性地址。在两级分页系统中，该宏等价于 pmd_page()，后者应用于页中间目录项。

pmd_index(addr)　　产生线性地址 addr 在页中间目录中所对应目录项的索引（相对位置）。

pmd_offset(pud, addr)　　接收指向页上级目录项的指针 pud 和线性地址 addr 作为参数。这个宏产生目录项 addr 在页中间目录中的偏移地址。在两级或三级分页系统中，它产生 pud，即页全局目录项的地址。

pmd_page(pmd)　　通过页中间目录项 pmd 产生相应页表的页描述符地址。在两级或三级分页系统中，pmd 实际上是页全局目录中的一项。

mk_pte(p,prot)　　接收页描述符地址 p 和一组访问权限 prot 作为参数，并创建相应的页表项。

pte_index(addr)　　产生线性地址 addr 对应的表项在页表中的索引（相对位置）。

pte_offset_kernel(dir, addr)　　线性地址 addr 在页中间目录 dir 中有一个对应的项，该宏就产生这个对应项，即页表的线性地址。另外，该宏只在主内核页表上使用（参见稍后“内核页表”一节）。

pte_offset_map(dir, addr)　　接收指向一个页中间目录项的指针 dir 和线性地址 addr 作为参数，它产生与线性地址 addr 相对应的页表项的线性地址。如果页表被保存在高端存储器中，那么内核建立一个临时内核映射（参见第八章“高端内存页框的内核映射”一节），并用 pte_unmap 对它进行释放。pte_offset_map_nested 宏和 pte_unmap_nested 宏是相同的，但它们使用不同的临时内核映射。

pte_page( x )　　返回页表项 x 所引用页的描述符地址。

pte_to_pgoff( pte )　　从一个页表项的 pte 字段内容中提取出文件偏移量，这个偏移量对应着一个非线性文件内存映射所在的页（参见第十六章“非线性存储器映射”一节）。

pgoff_to_pte(offset )　　为非线性文件内存映射所在的页创建对应页表项的内容。

这里罗列最后一组函数来简化页表项的创建和撤消。

当使用两级页表时，创建或删除一个页中间目录项是不重要的。如本节前部分所述，页中间目录仅含有一个指向下属页表的目录项。所以，页中间目录项只是页全局目录中的一项而已。然而当处理页表时，创建一个页表项可能很复杂，因为包含页表项的那个页表可能就不存在。在这样的情况下，有必要分配一个新页框，把它填写为 0，并把这个表项加入。

如果 PAE 被激活，内核使用三级页表。当内核创建一个新的页全局目录时，同时也分配四个相应的页中间目录；只有当父页全局目录被释放时，这四个页中间目录才得以释放。

当使用两级或三级分页时，页上级目录项总是被映射为页全局目录中的一个单独项。

与以往一样，表 2-8 中列出的函数描述是针对 80x86 构架的。

表 2-8：页分配函数

| 函数名称 | 说明 |
| --- | --- |
| pgd_alloc( mm ) | 分配一个新的页全局目录。如果 PAE 被激活，它还分配三个对应用户态线性地址的子页中间目录。参数 mm(内存描述符的地址)在 80x86 构架上被忽略。 |
| pgd_free( pgd) | 释放页全局目录中地址为 pgd 的项。如果 PAE 被激活，它还将释放用户态线性地址对应的三个页中间目录。 |
| pud_alloc(mm, pgd, addr) | 在两级或三级分页系统下，这个函数什么也不做：它仅仅返回页全局目录项 pgd 的线性地址。 |
| pud_free(x) | 在两级或三级分页系统下，这个宏什么也不做。 |
| pmd_alloc(mm, pud, addr) | 定义这个函数以使普通三级分页系统可以为线性地址 addr 分配一个新的页中间目录。如果 PAE 未被激活，这个函数只是返回输入参数 pud 的值，也就是说，返回页全局目录中目录项的地址。如果 PAE 被激活，该函数返回线性地址 addr 对应的页中间目录项的线性地址。参数 mm 被忽略。 |
| pmd_free(x ) | 该函数什么也不做，因为页中间目录的分配和释放是随同它们的父全局目录一同进行的。 |
| pte_alloc_map(mm, pmd, addr) | 接收页中间目录项的地址 pmd 和线性地址 addr 作为参数，并返回与 addr 对应的页表项的地址。如果页中间目录项为空，该函数通过调用函数 pte_alloc_one( )分配一个新页表。如果分配了一个新页表，addr 对应的项就被创建，同时 User/Supervisor 标志被设置为 1。如 |

果页表被保存在高端内存，则内核建立一个临时内核映射（参见第八章“高端内存页框的内核映射”一节），并用 pte_unmap 对它进行释放。

| | |
|---|---|
| pte_alloc_kernel(mm, pmd, addr) | 如果与地址 addr 相关的页中间目录项 pmd 为空，该函数分配一个新页表。然后返回与 addr 相关的页表项的线性地址。该函数仅被主内核页表使用（参见稍后“内核页表”一节）。 |
| pte_free( pte) | 释放与页描述符指针 pte 相关的页表。 |
| pte_free_kernel(pte) | 等价于 pte_free( )，但由主内核页表使用。 |
| clear_page_range(mmu, start,end) | 从线性地址 start 到 end 通过反复释放页表和清除页中间目录项来清除进程页表的内容。 |

## 物理内存布局

在初始化阶段，内核必须建立一个物理地址映射来指定哪些物理地址范围对内核可用而哪些不可用（或者因为它们映射硬件设备 I/O 的共享内存，或者因为相应的页框含有 BIOS 数据）。

内核将下列页框记为保留：

- 在不可用的物理地址范围内的页框
- 含有内核代码和已初始化的数据结构的页框

保留页框中的页绝不能被动态分配或交换到磁盘上。

一般来说，Linux内核安装在RAM中从物理地址0x00100000开始的地方，也就是说，从第二个MB开始。所需页框总数依赖于内核的配置方案：典型的配置所得到的内核可以被安装在小于3MB的RAM中.

为什么内核没有安装在 RAM 第一个 MB 开始的地方？因为 PC 体系结构有几个独特的地方必须考虑到。例如：

- 页框 0 由 BIOS 使用，存放加电自检（Power-On Self-Test ，POST）期间检查到的硬件配置。因此，很多膝上型电脑的 BIOS 甚至在系统初始化后还将数据写到该页框。

- 物理地址从 0x000a0000 到 0x000fffff 的范围通常留给 BIOS 例程，并且映射 ISA 图形卡上的内部存储器。这个区域就是所有 IBM 兼容 PC 上从 640KB 到 1MB 之间著名的洞：物理地址存在但被保留，对应的页框不能由操作系统使用。

- 前 1MB 内的其他页框可能由特定计算机模型保留。例如，IBM 笔记本电脑把 0x0a 页框映射到 0x9f 页框。

在启动过程的早期阶段（参看附录一），内核询问 BIOS 并了解物理内存的大小。在新近的计算机中，内核也调用 BIOS 过程建立一组物理地址范围和其对应的内存类型。

随后，内核执行 machine_specific_memory_setup( ) 函数，该函数建立物理地址映射（参见表 2-9 为例）。当然，如果这张表是可获取的，那是内核在 BIOS 列表的基础上构建的；否则，内核按保守的缺省设置构建这张表。从 0x9f (LOWMEMSIZE( )) 到 0x100 (HIGH_MEMORY)号的所有页框都标记为保留。

表 2-9：BIOS 提供的物理地址映射举例

| 开始 | 结束 | 类型 |
|---|---|---|
| 0x00000000 | 0x0009ffff | Usable |
| 0x000f0000 | 0x000fffff | Reserved |
| 0x00100000 | 0x07feffff | Usable |
| 0x07ff0000 | 0x07ff2fff | ACPI data |
| 0x07ff3000 | 0x07ffffff | ACPI NVS |
| 0xffff0000 | 0xffffffff | Reserved |

表 2-9 显示了具有 128MB RAM 计算机的典型配置。从 0x07ff0000 到 0x07ff2fff 的物理地址范围中存有加电自测（POST）阶段由 BIOS 写入的系统硬件设备信息；在初始化阶段，内核把这些信息拷贝到一个合适的内核数据结构中，然后认为这些页框是可用的。相反，从 0x07ff3000 到 0x07ffffff 的物理地址范围被映射到硬件设备的 ROM 芯片。从 0xffff0000 开始的物理地址范围标记为保留，因为它由硬件映射到 BIOS 的 ROM 芯片（参见附录一）。注意 BIOS 也许并不提供一些物理地址范围的信息（在上述表中，范围是 0x000a0000 到 0x000effff）。为安全可靠起见，Linux 假定这样的范围是不可用的。

内核可能不会见到 BIOS 报告的所有物理内存：例如，如果未使用 PAE 支持来编译，即使有更大的物理内存可供使用，内核也只能寻址 4GB 大小的 RAM。setup_memory( ) 函数在 machine_specific_memory_setup( ) 执行后被调用：它分析物理内存区域表并初始化一些变量来描述内核的物理内存布局，这些变量如表 2-10 所示。

表 2-10：描述内核物理内存布局的变量

| 变量名称 | 说明 |
|---|---|
| num_physpages | 最高可用页框的页框号 |
| totalram_pages | 可用页框的总数量 |

| | |
|---|---|
| min_low_pfn | RAM 中在内核映像后第一个可用页框的页框号 |
| max_pfn | 最后一个可用页框的页框号 |
| max_low_pfn | 被内核直接映射的最后一个页框的页框号（低地址内存） |
| totalhigh_pages | 内核非直接映射的页框的总数（高地址内存） |
| highstart_pfn | 内核非直接映射的第一个页框的页框号 |
| highend_pfn | 内核非直接映射的最后一个页框的页框号 |

为了避免把内核装入一组不连续的页框里，Linux 更愿跳过第 1MB 的 RAM。更明确地说，Linux 用 PC 体系结构未保留的页框来动态存放所分配的页。

图 2-13 显示 Linux 怎样填充前 3MB 的 RAM。我们假设内核需要小于 3MB 的 RAM。

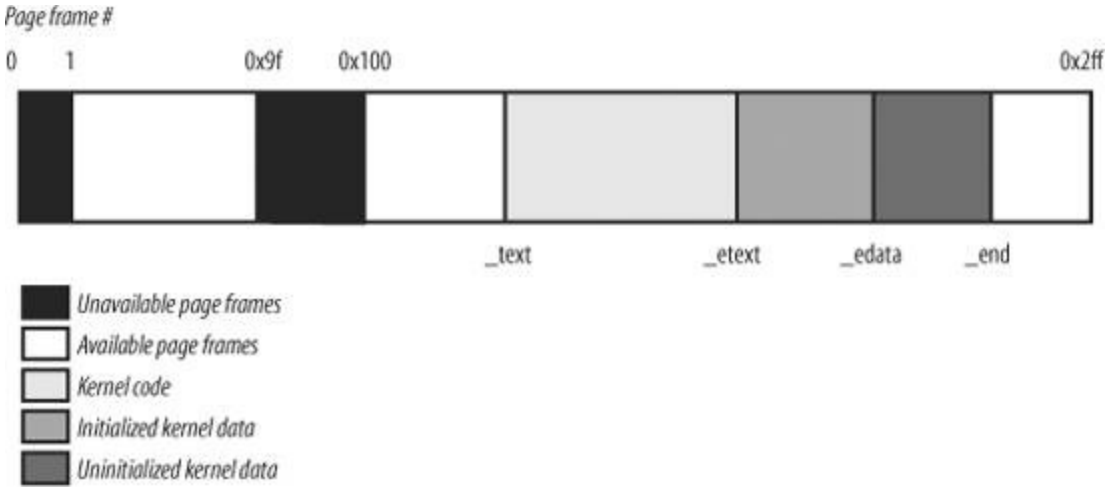



图 2-13：Linux2.6 的前 768 个页框（3MB）

符号_text 对应于物理地址 0x00100000，表示内核代码第一个字节的地址。内核代码的结束位置由另外一个类似的符号_etext 表示。内核数据分为两组：初始化过的数据的和没有初始化的数据。初始化的数据在_etext 后开始，在_edata 处结束。紧接着是未初始化的数据并以_end 结束。

图中出现的符号并没有在源代码中定义，它们是编译内核时产生的（注5）。

## 进程页表

注5：你可以在 *System.map* 文件中找到这些符号的线性地址，*System.map* 是编译内核以后所创建的。

进程的线性地址空间分成两部分：

- 从 0x00000000 到 0xbfffffff 的线性地址，无论用户态还是内核态的进程都可以寻址。

- 从 0xc0000000 到 0xffffffff 的线性地址，只有内核态的进程才能寻址。

当进程运行在用户态时，它产生的线性地址小于 0xc0000000；当进程运行在内核态时，它执行内核代码，所产生的地址大于等于 0xc0000000。但是，在某些情况下，内核为了检索或存放数据必须访问用户态线性地址空间。

宏 PAGE_OFFSET 产生的值是 0xc0000000，这就是进程在线性地址空间中的偏移量，也是内核生存空间的开始之处。在本书中，我们常常直接引用 0xc0000000 这个数。

页全局目录的第一部分表项映射的线性地址小于 0xc0000000(在 PAE 未启用时是前 768 项，PAE 启动时是前 3 项)，具体大小依赖特定进程。相反，剩余的表项对所有进程来说都应该是相同的，它们等于主内核页全局目录（master kernel Page Global Directory）的相应表项 (参见下一节)。

## 内核页表

内核维持着一组自己使用的页表，驻留在所谓主内核页全局目录（master kernel Page Global Directory）中。系统初始化后，这组页表还从未被任何进程或任何内核线程直接使用；更确切地说，主内核页全局目录的最高目录项部分，为系统中每个普通进程对应的页全局目录项提供参考模型。

我们在第八章“非连续内存区的线性地址“一节将会解释，内核如何确保对主内核页全局目录的修改能传递到由进程实际使用的页全局目录中。

我们现在描述内核如何初始化自己的页表。这个过程分为两个阶段。事实上，内核映像刚刚被装入内存后，CPU 仍然运行于实模式，所以分页功能没有被启用。

第一个阶段，内核创建一个有限的地址空间，包括内核的代码段和数据段、初始页表和用于存放动态数据结构的共 128KB 大小的空间。这个最小限度的地址空间仅足够将内核装入 RAM 和对其初始化的核心数据结构。

第二个阶段，内核充分利用剩余的 RAM 并适当地建立分页表。下一节解释这个方案是怎样实施的。

### 临时内核页表

临时页全局目录是在内核编译过程中静态地初始化的，而临时页表是由 startup_32( )汇编语言函数（定义于 arch/i386/kernel/head.S）初始化的。我们不再过多提及页中间目录，因为它们相当于页全局目录项。在这个阶段 PAE 支持并未被激活。

临时页全局目录放在 swapper_pg_dir 变量中。临时页表在 pg0 变量处开始存放，紧接在内核未初始化的数据段（图 2-13 中的_end 符号）后面。为简单起见，我们假设内核使用的段、临时页表和 128KB 的内存范围能容

纳于 RAM 前 8MB 空间里。为了映射 RAM 前 8MB 的空间，需要使用到两个页表。

分页第一个阶段的目标是允许既在实模式下也在和保护模式下都能很容易地对这 8MB 寻址。因此，内核必须创建一个映射，把从 0x00000000 到 0x007fffff 的线性地址和从 0xc0000000 到 0xc07fffff 的线性地址映射到从 0x00000000 到 0x007fffff 的物理地址。换句话说，内核在初始化的第一阶段，可以通过与物理地址相同的线性地址或者通过从 0xc0000000 开始的 8MB 线性地址对 RAM 的前 8MB 进行寻址。

内核通过把 swapper_pg_dir 所有项都填充为 0 来创建期望的映射，不过，0、1、0x300（十进制 768）和 0x301(十进制 769)这四项除外；后两项包含了从 0xc0000000 到 0xc07fffff 间的所有线性地址。0、1、0x300 和 0x301 按以下方式初始化：

- 0 项和 0x300 项的地址字段置为 pg0 的物理地址，而 1 项和 0x301 项的地址字段置为紧随 pg0 后的页框的物理地址。

- 把这四个项中的 Present,、Read/Write 和 User/Supervisor 标志置位。

- 把这四个项中的 Accessed 、Dirty、PCD、PWD 和 Page Size 标志清 0。

汇编语言函数startup_32( )也启用分页单元。通过向cr3控制寄存器装入swapper_pg_dir的地址及设置cr0控制寄存器的PG标志来达到这一目的。下面是等价的代码片段：

```
movl $swapper_pg_dir-0xc0000000,%eax
movl %eax,%cr3       /*设置页表指针*/…
movl %cr0,%eax
orl $0x80000000,%eax
movl %eax,%cr0       /*……设置分页（PG）位*/
```

**RAM** 小于 **896MB** 时的最终内核页表

由内核页表所提供的最终映射必须把从 0xc0000000 开始的线性地址转化为从 0 开始的物理地址。

宏__pa 用于把从 PAGE_OFFSET 开始的线性地址转换成相应的物理地址，而宏__va 做相反的转化。

主内核页全局目录（master kernel Page Global Directory）仍然保存在 swapper_pg_dir 变量中。它由 paging_init( ) 函数初始化。该函数进行如下操作：

1. 调用 pagetable_init( )适当地建立页表项。
2. 把 swapper_pg_dir 的物理地址写入 cr3 控制寄存器中。
3. 如果 CPU 支持 PAE 并且如果内核编译时支持 PAE，将 cr4 控制寄存器 的 PAE 标志置位。
4. 调用 flush_tlb_all( )使 TLB 的所有项无效。

1.
pagetable_init( )执行的操作既依赖于现有 RAM 的容量，也依赖于 CPU 模式。让我们从最简单的情况开始。我们的计算机有小于 896MB（注6）的 RAM，32 位物理地址足以对所有可用 RAM 进行寻址，因而没有必要激活 PAE 机制。（参见前面“物理地址扩展（PAE）分页机制”一节）。

swapper_pg_dir 页全局目录由如下等价的循环重新初始化：

```
pgd = swapper_pg_dir + pgd_index(PAGE_OFFSET); /* 768 */
phys_addr = 0x00000000;
while (phys_addr < (max_low_pfn * PAGE_SIZE)) {
    pmd = one_md_table_init(pgd); /*返回 pgd*/
set_pmd(pmd, _ _pmd(phys_addr | pgprot_val(_ _pgprot(0x1e3))));
/* 0x1e3 == Present, Accessed, Dirty, Read/Write,
        Page Size, Global */
phys_addr += PTRS_PER_PTE * PAGE_SIZE; /* 0x400000 */
++pgd;
}
```

我们假定 CPU 是支持 4MB 页和“全局（global）”TLB 表项的最新 80x86 微处理器。注意如果页全局目录项对应的是 0xc0000000 之上的线性地址，则把所有这些项的 User/Supervisor 标志清 0，由此拒绝用户态进程访问内核地址空间。还要注意 Page Size 被置位使得内核可以通过使用大型页来对 RAM 进行寻址（参见本章先前的“扩展分页”一节）。

由 startup_32( )函数创建的物理内存前 8MB 的恒等映射用来完成内核的初始化阶段。当这种映射不再必要时，内核调用 zap_low_mappings( )函数清除对应的页表项。

实际上，这种描述并未说明全部事实。我们将在后面“固定映射的线性地址”一节看到，内核也调整与“固定映射的线性地址”对应的页表项。

当 **RAM** 大小在 **896MB** 和 **4096MB** 之间时的最终内核页表

在这种情况下，并不把 RAM 全部映射到内核地址空间。Linux 在初始化阶段可以做的最好的事是把一个具有 896MB 的 RAM 窗口（window）映射到内核线性地址空间。如果一个程序需要对现有 RAM 的其余部分寻址，那就必须把某些其他的线性地址间隔映射到所需的 RAM。这意味着修改某些页表项的值。我们将在第八章讨论这种动态重映射是如何进行的。

内核使用与前一种情况相同的代码来初始化页全局目录。

当 **RAM** 大于 **4096MB** 时的最终内核页表

现在让我们考虑大于 4GB 计算机的内核页表初始化；更确切地说，我们处理以下发生的情况：

---

注6：线性地址的最高 128MB 留给几种映射去用（参见本章后面“固定映射的线性地址”一节和第七章“非连续存储器区管理”一节）。因此映射 RAM 所剩空间为 1GB−128MB＝896MB。

- CPU 模式支持物理地址扩展（PAE）

- RAM 容量大于 4GB

- 内核以 PAE 支持来编译

尽管 PAE 处理 36 位物理地址，但是线性地址依然是 32 位地址。如前所述，Linux 映射一个 896 MB 的 RAM 窗口到内核地址空间；剩余 RAM 留着不映射，并由动态重映射来处理，第八章将对此描述。与前一种情况的主要差异是使用三级分页模型，因此页全局目录按以下循环代码来初始化：

```
pgd_idx = pgd_index(PAGE_OFFSET); /* 3 */
for (i=0; i<pgd_idx; i++)
set_pgd(swapper_pg_dir + i, _ _pgd(_ _pa(empty_zero_page) + 0x001));
/* 0x001 == Present */
pgd = swapper_pg_dir + pgd_idx;
phys_addr = 0x00000000;
for (; i<PTRS_PER_PGD; ++i, ++pgd) {
pmd = (pmd_t *) alloc_bootmem_low_pages(PAGE_SIZE);
set_pgd(pgd, _ _pgd(_ _pa(pmd) | 0x001)); /* 0x001 == Present */
if (phys_addr < max_low_pfn * PAGE_SIZE)
    for (j=0; j < PTRS_PER_PMD /* 512 */
         && phys_addr < max_low_pfn*PAGE_SIZE; ++j) {
    set_pmd(pmd, _ _pmd(phys_addr |
                pgprot_val(_ _pgprot(0x1e3))));
    /* 0x1e3 == Present, Accessed, Dirty, Read/Write,
              Page Size, Global */
    phys_addr += PTRS_PER_PTE * PAGE_SIZE; /* 0x200000 */
    }
}
swapper_pg_dir[0] = swapper_pg_dir[pgd_idx];
```

页全局目录中的前三项与用户线性地址空间相对应，内核用一个空页（empty_zero_page）的地址对这三项进行初始化。第四项用页中间目录 (pmd) 的地址初始化，该页中间目录是通过调用 alloc_bootmem_low_pages( )分配的。页中间目录中的前 448 项（有 512 项，但后 64 项留给非连续内存分配；参见第八章的“非连续内存区域管理”一节）用 RAM 前 896MB 的物理地址填充。

注意，支持 PAE 的所有 CPU 模式也支持大型 2MB 页和全局页。正如前一种情况一样，只要可能，Linux 使用大页来减少页表数。

然后页全局目录的第四项被拷贝到第一项中，这样好为线性地址空间的前 896MB 中的低物理内存映射做镜像。为了完成对 SMP 系统的初始化，这个映射是必需的：当这个映射不再必要时，内核通过调用

zap_low_mappings( )函数来清除对应的页表项，正如先前的情况一样。

## 固定映射的线性地址（**fix-mapped linear addresses**）

我们看到内核线性地址第四个 GB 的初始部分映射系统的物理内存。但是，至少 128MB 的线性地址总是留作他用，因为内核使用这些线性地址实现非连续内存分配和固定映射的线性地址。

非连续内存分配仅仅是动态分配和释放内存页的一种特殊方式，将在第八章“非连续内存区管理”一节描述。本节我们集中讨论固定映射的线性地址。

固定映射的线性地址基本上是一种类似于 0xffffc000 这样的常量线性地址，其对应的物理地址不必等于线性地址减去 0xc000000，而是可以以任意方式建立。因此，每个固定映射的线性地址都映射一个物理内存的页框。我们将会在后面的章节看到，内核使用固定映射的线性地址来代替指针变量，因为这些指针变量的值从不改变。

固定映射的线性地址概念上类似于对 RAM 前 896MB 映射的线性地址。不过，固定映射的线性地址可以映射任何物理地址，而由第 4GB 初始部分的线性地址所建立的映射是线性的（线性地址 X 映射物理地址 X-PAGE_OFFSET）。

就指针变量而言，固定映射的线性地址更有效。事实上，间接引用一个指针变量比间接引用一个立即常量地址要多一次内存访问。此外，在间接引用一个指针变量之前对其值进行检查是一个良好的编程习惯；相反，对一个常量线性地址的检查则是没有必要的。

每个固定映射的线性地址都由定义于 enum fixed_addresses 数据结构中的整型索引来表示：

```
enum fixed_addresses {
FIX_HOLE,
FIX_VSYSCALL,
FIX_APIC_BASE,
FIX_IO_APIC_BASE_0,
[...]
 __end_of_fixed_addresses
};
```

每个固定映射的线性地址都存放在线性地址第四个 GB 的末端。fix_to_virt( )函数计算从给定索引开始的常量线性地址：

```
inline unsigned long fix_to_virt(const unsigned int idx)
{
if (idx >= _ _end_of_fixed_addresses)
      __this_fixmap_does_not_exist( );
return (0xfffff000UL - (idx << PAGE_SHIFT));
}
```

让我们假定某个内核函数调用 fix_to_virt(FIX_IOAPIC_BASE_0)。因为该函数声明为“inline”，所以 C 编译程序不调用 fix_to_virt( )，而是仅仅把它的代码插入到调用函数中。此外，运行时从不对这个索引值执行检查。事实上，FIX_IOAPIC_BASE_0 是个等于 3 的常量，因此编译程序可以去掉 if 语句，因为它的条件在编译时为假。相反，如果条件为真，或者 fix_to_virt( )的参数不是一个常量，则编译程序在连接阶段产生一个错误，因为符号 __this_fixmap_does_not_exist 在别处没有定义。最后，编译程序计算 0xffffe000-(3<<PAGE_SHIFT)，并用常量线性地址 0xffffc000 代替 fix_to_virt( )函数调用。

为了把一个物理地址与固定映射的线性地址关联起来，内核使用 set_fixmap(idx, phys) 和 set_fixmap_nocache(idx, phys)宏。这两个函数都把 fix_to_virt(idx)线性地址对应的一个页表项初始化为物理地址 phys；不过，第二个函数也把页表项的 PCD 标志置位，因此，当访问这个页框中的数据时禁用硬件高速缓存（参见本章前面“硬件高速缓存”一节）。反过来，clear_fixmap(idx)用来撤消固定映射线性地址 idx 和物理地址之间的连接。

## 处理硬件高速缓存和 TLB

存储器寻址的最后一个主题是关于内核如何使用硬件高速缓存来达到最佳效果。硬件高速缓存和翻译后援缓冲器（TLB）在提高现代计算机体系结构的性能上扮演着重要角色。内核开发者采用一些技术来减少高速缓存和 TLB 的未命中次数。

处理硬件高速缓存

如前所述，硬件高速缓存是通过高速缓存行（cache line）寻址的。L1_CACHE_BYTES 宏产生以字节为单位的高速缓存行的大小。在早于 Pentium 4 的 Intel 模型中，这个宏产生的值为 32；在 Pentium 4 上，它产生的值为 128。

为了使高速缓存的命中率达到最优化，内核在下列决策中考虑构架：

- 一个数据结构中最常使用的字段放在该数据结构内的低偏移部分，以便它们能够处于高速缓存的同一行中。
- 当为一大组数据结构分配空间时，内核试图把它们都存放在内存，以便所有高速缓存行按同一方式使用。

80x86 微处理器自动处理高速缓存的同步，所以应用于这种处理器的 Linux 内核并不处理任何硬件高速缓存的刷新。不过内核却为不能同步高速缓存的处理器提供了高速缓存刷新接口。

处理 TLB

处理器不能自动同步它们自己的 TLB 高速缓存，因为决定线性地址和物理地址之间映射何时不再有效的是内核，而不是硬件。

Linux 2.6 提供了几种在合适时机应当运用的刷新方法，这取决于页表更换的类型（参见表 2-11）。

表 2-11：独立于系统的使 TLB 表项无效的方法

| 方法名称 | 说明 | 典型的应用时机 |
| --- | --- | --- |
| flush_tlb_all | 刷新的 TLB 表项（包括那些全局页对应的 TLB 表项，即那些 Global 标志被置位的页） | 改变内核页表项时 |
| flush_tlb_kernel_range | 刷新给定线性地址范围内的所有 TLB 表项（包括那些全局页对应的 TLB 表项） | 更换一个范围内的内核页表项时 |
| flush_tlb | 刷新当前进程拥有的非全局页相关的所有 TLB 表项 | 执行进程切换时 |
| flush_tlb_mm | 刷新指定进程拥有的非全局页相关的所有 TLB 表项 | 创建一个新的子进程时 |
| flush_tlb_range | 刷新指定进程的线性地址间隔对应的 TLB 表项 | 释放某个进程的线性地址间隔时 |
| flush_tlb_pgtables | 刷新指定进程中特定的相临页表集相关的 TLB 表项刷新指定进程中特定的相临页表集相关的 TLB 表项 | 释放进程的一些页表时 |
| flush_tlb_page | 刷新指定进程中单个页表项相关的 TLB 表项 | 处理缺页异常时 |

尽管普通 Linux 内核提供了丰富的 TLB 方法，但通常每个微处理器都提供了更受限制的一组使 TLB 无效的汇编语言指令。在这个方面，一个更为灵活的硬件平台就是 Sun 的 UltraSPARC。与之相比，Intel 微处理器只提供了两种使 TLB 无效的技术：

- 在向 cr3 寄存器写入值时所有 Pentium 处理器自动刷新相对于非全局页的 TLB 表项。

- 在 Pentium Pro 及以后的处理器中，invlpg 汇编语言指令使映射指定线性地址的单个 TLB 表项无效。

表 2-12 列出了采用这种硬件技术的 Linux 宏；这些宏是实现独立于系统的方法的基本要素。

表 2-12：Intel Pentium Pro 及以后的处理器上使用的使 TLB 无效的宏

| 宏名称 | 描述 | 使用对象 |
| --- | --- | --- |
| __flush_tlb( ) | 将 cr3 寄存器的当前值重新写回 cr3 | flush_tlb_m，flush_tlb_range |
| __flush_tlb_global( ) | 通过清除 cr4 的 PGE 标志禁用全局页，将 cr3 寄存器的当前值重新写回 cr3，并再次设置 PGE 标志 | flush_tlb_all，flush_tlb_kernel_range |
| __flush_tlb_single(addr) | 接收参数 addr 执行 invlpg 汇编语言指令 | flush_tlb_page |

注意表 2-12 中没有 flush_tlb_pgtables 方法：在 80x86 系统中，当页表与父页表解除链接时什么也不需要做，所以实现这个方法的函数为空。

独立于构架的使 TLB 无效的方法非常简单地扩展到了多处理器系统上。在一个 CPU 上运行的函数发送一个处理器间中断（参见第四章的“处理器间中断处理”）给其它的 CPU 来强制它们执行适当的函数使 TLB 无效。

一般来说，任何进程切换都会暗示着更换活动页表集。相对于过期页表，本地 TLB 表项必须被刷新；这个过程在内核把新的页全局目录的地址写入 cr3 控制寄存器时会自动完成。不过内核在下列情况下将避免 TLB 被刷新：

- 当两个使用相同页表集的普通进程之间执行进程切换时（参见第七章的“schedule( )函数”一节）。

- 当在一个普通进程和一个内核线程间执行进程切换时。事实上，我们将在第九章的“内核线程的内存描述符”一节看到，内核线程并不拥有自己的页表集；更确切地说，它们使用刚在 CPU 上执行过的普通进程的页表集。

除了进程调度以外，还有其他几种情况下内核需要刷新 TLB 中的一些表项。例如，当内核为某个用户态进程分配页框并将它的物理地址存入页表项时，它必须刷新与相应线性地址对应的任何本地 TLB 表项。在多处理器系统中，如果有多个 CPU 在使用相同的页表集，那么内核还必须刷新这些 CPU 上使用相同页表集的 TLB 表项。

为了避免多处理机系统上无用的 TLB 刷新，内核使用一种叫做懒惰 TLB（lazy TLB）模式的技术。其基本思想是，如果几个 CPU 正在使用相同的页表，而且必须对这些 CPU 上的一个 TLB 表项刷新，那么，在某些情况下，正在运行内核线程的那些 CPU 上的刷新就可以延迟。

事实上，每个内核线程并不拥有自己的页表集；更确切地说，它使用一个普通进程的页表集。不过，没有必要使一个用户态线性地址对应的 TLB 表项无效，因为内核线程不访问内核态地址空间（注7）。

当某个 CPU 开始运行一个内核线程时，内核把它置为懒惰 TLB 模式。当发出清除 TLB 表项的请求时，处于懒惰 TLB 模式的每个 CPU 都不刷新相应的表项；但是，CPU 记住它的当前进程正运行在一组页表上，而这组页表的 TLB 表项对用户态地址是无效的。只要处于懒惰 TLB 模式的 CPU 用一个不同的页表集切换到一个普通进程，硬件就自动刷新 TLB 表项，同时内核把 CPU 设置为非懒惰 TLB 模式。然而，如果处于懒惰 TLB 模式的 CPU 切换到的进程与刚才运行的内核线程拥有相同的页表集，那么，任何使 TLB 无效的延迟操作必须由内核有效地实施；这种使 TLB 无效的“懒惰”操作可以通过刷新 CPU 的所有非全局 TLB 项来有效地获取。

注7：顺便说一句，flush_tlb_all 方法并不使用懒惰 TLB 模式机制；通常只有在内核修改与内核地址空间相关的一个页表项时才调用这个方法。

为了实现懒惰 TLB 模式，需要一些额外的数据结构。cpu_tlbstate 变量是一个具有 NR_CPUS 个结构的静态数组（这个宏的默认值是 32；它代表了系统中 CPU 的最大数量），这个结构有两个字段，一个是指向当前进程内存描述符的 active_mm 字段（参见第八章），一个是具有两个状态值的 state 字段：TLBSTATE_OK（非懒惰模式）或 TLBSTATE_LAZY（懒惰模式）。此外，每个内存描述符中包含一个 cpu_vm_mask 字段，该字段存放的是 CPU（这些 CPU 将要接收与 TLB 刷新相关的处理器间中断）下标；只有当内存描述符属于当前运行的一个进程时这个字段才有意义。

当一个 CPU 开始执行内核线程时，内核把该 CPU 的 cpu_tlbstate 元素的 state 字段置为 TLBSTATE_LAZY；此外，活动（active）内存描述符的 cpu_vm_mask 字段存放系统中所有 CPU（包括进入懒惰 TLB 模式的 CPU）的下标。对于与给定页表集相关的所有 CPU 的 TLB 表项，当另外一个 CPU 想使这些表项无效时，该 CPU 就把一个处理器间中断发送给下标处于对应内存描述符的 cpu_vm_mask 字段中的那些 CPU。

当 CPU 接受到一个与 TLB 刷新相关的处理器间中断，并验证它影响了它当前进程的页表集时，它就检查它的 cpu_tlbstate 元素的 state 字段是否等于 TLBSTATE_LAZY；如果等于，内核就拒绝使 TLB 表项无效，并从内存描述符的 cpu_vm_mask 字段删除该 CPU 下标。这有两种结果：

- 只要 CPU 还处于懒惰 TLB 模式，它将不接受其他与 TLB 刷新相关的处理器间中断。

- 如果 CPU 切换到另一个进程，而这个进程与刚被替换的内核线程使用相同的页表集，那么内核调用 __flush_tlb()使该 CPU 的所有非全局 TLB 表项无效。