# Linux内存

Linux采用的是虚拟内存管理技术，每个进程都有自己独立的进程内存空间，大小为4G的连续虚拟内存。4G的虚拟内存又被分为两个部分：用户空间（0到3G）和内核空间（3G到4G）。用户能接触到的只是用户空间的虚拟地址。内核空间的虚拟地址只有内核才能访问。

虚拟内存透明的使用磁盘空间，使得程序在运行的时候可以使用比物理能存更大的内存空间。在使用虚拟内存的时候，程序数据被分割成基本单元，这些单元被称为页（page）。程序正在使用的页在内存中，未被使用的页则存放在硬盘。物理内存则被划分成若干个用来保存这些页的区域被称为页框（page frame）。如果被访问的页没有在内存中，则会产生一个缺页中断。此时内核必须选择一个页面将其内容写入磁盘，然后再从磁盘将需要访问的页取出到内存中。

**逻辑地址**（**虚拟地址**）：由一个段标识符和偏移量组成，表示从段开始的地方到实际地址之间的距离。

**线性地址**：又叫虚拟地址，32位无符号整数。最高可以表达4GB的地址。范围0x00000000到0xffffffff。

**物理地址**：内存芯片内存单元寻址，从CPU的地址引脚发送到总线上的电信号相对应。由32位或36位无符号整数表示。

内存控制单元MMU通过分段单元将逻辑地址转换为线性地址，然后通过分页单元将线性地址转换为物理地址。

## 硬件分段

### 段标识符（选择符）

段标识符也叫做段选择符，**用来从段描述符表中选择段描述符**。段标识符长度为16位。段标识符格式：

|  |  |  |
| --- | --- | --- |
| Index（13位） | TI（1位） | RPL（2位） |

index：指定放在GDT或LDT中相应段描述符的入口。

TI：0表示段描述符在GDT中，1表示段描述符在LDT中。

RPL：请求特权级。当段标识符装入cs寄存器时，指示出CPU当前的特权级。还可以在访问数据段时有选择降低CPU的特权级。

### 段寄存器

CPU提供了6个段寄存器cs\ss\ds\es\fs\gs，用来**存放**段标识符。其中三个寄存器有专门用途：

cs：代码段寄存器，指向存续指令的段。寄存器其中两位用以指明CPU当前特权级。值位0代表最高优先级。值位3代表最低优先级。linux用0和3代表内核态与用户态。

ss：栈段寄存器，指向当前程序栈的段。

ds：数据段寄存器，指向静态数据或全局数据段。

ds寄存器指向用户数据段还是内核数据段是由cs寄存器CPL字段决定的，为0则ds指向内核数据段，为3则ds指向用户数据段。

ss寄存器的值也是一样。

### 段描述符

段描述符是放在GDT全局描述符表或局部描述符表LDT中，8字节大小用来存放一个段的信息。包含以下字段：

Base：段首的线性地址。

G：粒度。如果为0，则段大小以字节为单位，否则以4096的倍数计算。

S：系统标志。如果为0，标识一个系统段。否则就是一个普通的代码段或者数据段。

Type：描述段的类型特征和存取权限。

GDT全局段描述符表。在内存中的地址和大小放在CPU的gdtr寄存器。

LDT局部段描述符表。在内存中的地址和大小放在CPU的ldtr寄存器。

代码段描述符和数据段描述符都可以放在GDT或LDT中，描述符S标志位1。

任务状态段描述符（TSSD）：用于保存CPU寄存器的内容，只能放在GDT中。根据其对应的进程是否在CPU上执行，其Type字段的值分别为11或9。S标志位为0。

局部描述符表描述符（LDTD）：只出现在GDT中。Type字段值为2，S标志位为0。

从CPU的段寄存器中得到段标识符。根据其中的索引值在描述符表找到段描述符。再根据段描述符的base地址和逻辑地址的偏移量，就可以找到对应的段。

为了能够快速完成逻辑地址到线性地址的转换。CPU提供了一种**非编程的寄存器**，用来**存放段描述符**。当段标识符放入段寄存器，对应的段描述符也就被放入非编程的寄存器。这样后续就可以直接根据段描述符得到线性地址。减少了对GDT或LDT的访问。

### 分段单元

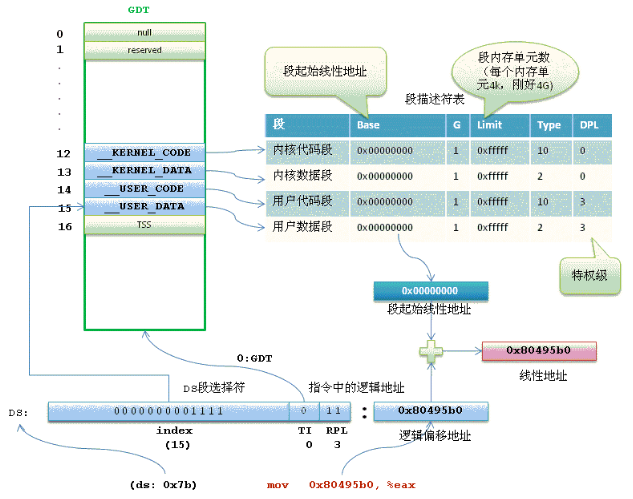
如何根据逻辑地址得到线性地址

1.检查段标识符的TI字段，判定段描述符是在GTD还是在LDT。如果在GTD，从gdtr寄存器得到GDT线性地址。在LDT，则从ldtr寄存器得到LDT的线性地址。

2.根据段标识符的index字段找到段描述符。段描述符线性地址 = GTD/LTD线性地址 + index \* 8。

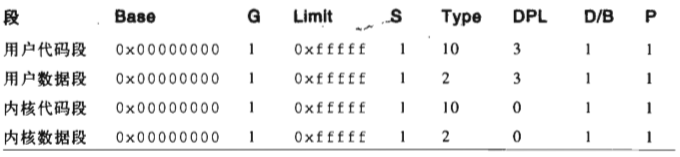
3.段描述符的Base字段值 + 逻辑地址的偏移量就是逻辑地址对应的线性地址。

有了不可编程寄存器之后，可以省下前两个步骤。



## linux分段

linux只有在80x86结构下使用分段，分为用户代码段、用户数据段、内核代码段、内核数据段。



从上面的表可以看出，由于各段base地址都是0，因此逻辑地址与线性地址总是一致的，即逻辑地址中的偏移量数值上与线性地址值相同。

多个CPU每个都会对应一个GDT。GDT存放在cpu\_gdt\_table数组中，GDT的地址和大小存放在cpu\_gdt\_descr数组中。

每个GDT包含18个段描述符和14个空的（保留的）项。这样保证所有的描述符都能处在同一个32字节的硬件高速缓存中。

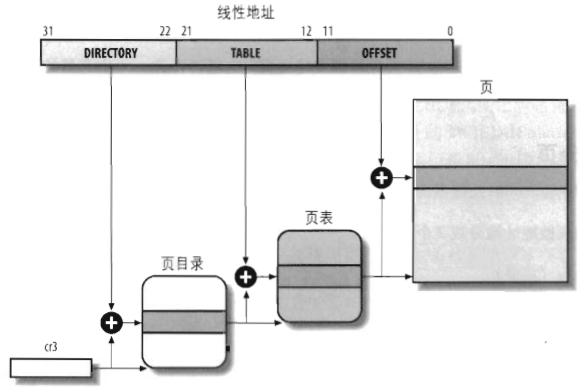
## 硬件分页

分页单元将线性地址转换为物理地址。会将请求访问的类型与线性地址的访问类型比较，如果不一致，则产生缺页异常。

线性地址被分成以固定长度为单位的组，称为页。页内部连续的线性地址被映射到连续的物理地址。分页单元把所有的RAM分成固定长度的页框。每个页框的大小和页的大小是一样的。页和页框的区别在于，页是一个数据块，可以放在任何页框中。页框则是主存的一部分，是一个存储区域。把线性地址映射到物理地址的数据结构叫页表。页表也存放在主存中。

80x86通过设置cr0寄存器的PG标志启用分页，当PG=0，线性地址被转换成物理地址。

**常规分页**中，线性地址的转换分为两步，第一步根据页目录表进行转换，第二部根据页表进行转换。offset是12位，因此每个页有4KB数据。



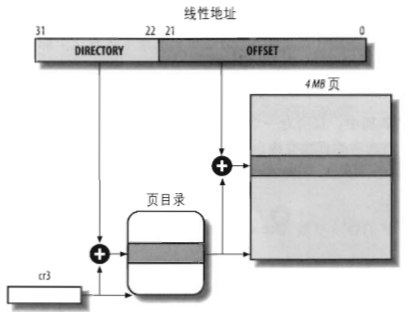
寄存器cr3中存放的是正在使用的页目录物理地址。根据线性地址的directory域找到线性地址对应的页目录项。再根据线性地址的table域找到页表项，页表项就是页所在页框的物理地址。最后offset则对应页框内的页位置。

页目录项和页表项有相同的结构，包含如下字段：

User/Supervisor：页和页表的特权级。如果为0，只有CPL小于3的时才能对页寻址。如果为1，则总能对页寻址。

Read/Write：页和页表的存取权限。只有两种，为0表示只读，否则表示可读写。

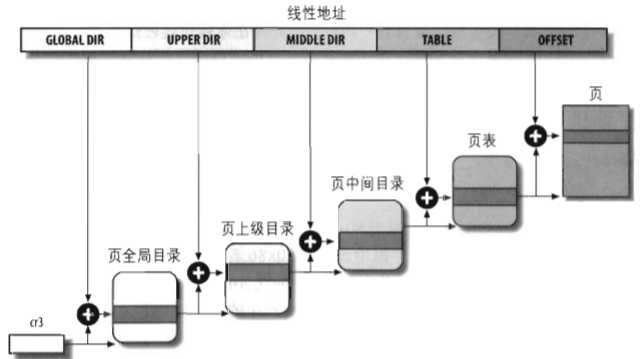
**扩展分页**则允许每个页有4MB数据。这种比常规分页少了页表。通过设置cr4寄存器的PSE标志，能使扩展页和常规页共存。



对于64位的x86平台，会采用4级分页的方式，页大小还是4KB。寻址使用48位，地址分级9+9+9+9+12。

## linux分页

2.6.11版本的linux采用四级分页模型。之前的linux采用的是三级分页模型。四级分页为：页全局目录、页上级目录、页中间目录、页表。



内核中使用struct page来表示物理页，注意是物理页而不是虚拟页。

### 页结构

struct page {

unsigned long flags;

atomic\_t \_count;

atomic\_t \_mapcount;

unsigned long private;

void \*virtual;

};

flag：用来存放页状态，比如页是否为脏，是否锁定。每个位表示一个状态，一共可以表示32种状态。

\_count：保存页的引用计数。如果为0表示页空闲。如果为正数表示页正在被使用。

virtual：保存页的虚拟地址。

内核把具有相似特性的页进行分组，划分为不同的**区**。linux系统中有三个内存管理区：

ZONE\_DMA：其中的页用来执行DMA操作。x86上对应的物理内存<16MB。

ZONE\_NORMAL：包含正常映射的页。x86上对应的物理内存16-896MB。

ZONE\_HIGHMEM：包含高端内存，页不能永久映射到内核地址空间。x86上对应的物理内存>896MB。

### 区结构

**区的作用**就是内核根据内存的用途进行逻辑上的管理，需要什么样的内存，就从对应的区中取出所需数量的页。

在64位x86的架构下，因为可以映射64位的内存空间，所以没有ZONE\_HIGHMEM区。

struct zone {

spinlock\_t lock;

unsigned long watermark[NR\_WMARK];

};

lock：自旋锁，防止并发访问。但只是保护区的数据结构，对于区中管理的页不保护。

watermark：当前区的最小值、最低水位和最高水位。通过水位来控制区中内存消耗的基准。

## slab层

slab是linux操作系统的一种内存分配机制。目的是针对频繁分配和释放的对象进行管理。避免频繁分配和释放带来的性能开销，同时也避免小块内存的分配和释放带来大量的内存碎片。

slab层根据不同的对象管理了多个高速缓存，每个高速缓存对应一种对象。比如进程描述符task\_struct高速缓存。高速缓存又划分为多个slab，每个slab由一个或者多个物理上连续的页组成。一般情况下一个slab由一个页组成。slab有专用的（如task\_struct）和通用的（如32、64、128等大小的）高速缓存。

从高速缓存中获取内存的时候，先从部分满的slab中分配。没有部分满的slab，就从空闲slab分配。如果所有slab都满，则创建新的slab。

### 高速缓存描述符

struct kmem\_cache\_s {

struct kmem\_list3 lists;

unsigned int objsize;

unsigned int num;

const char\* name;

};

lists：包含三个链表头，分别对应slab的三种状态：部分满、完全满和空闲。

objsize：缓存的对象大小，单位字节。

num：每个slab中包含的对象内存数量。

name：高速缓存的名称。

### slab描述符

struct slab\_s {

struct list\_head list;

unsigned long colouroff;

void\* s\_mem;

unsigned int inuse;

kmem\_bufctl\_t free;

};

list：指向高速缓存描述符的三个双向循环链表中的一个（已满，未满和空闲链表）。

clouroff：slab中第一个对象的便宜。

s\_mem：指向slab中的第一个对象（可能已使用也可能空闲）。

inuse：当前slab中已分配的对象个数。

### 普通和专用高速缓存

普通高速缓存：

专用高速缓存：