# 逻辑地址、线性地址、物理地址：

1. 逻辑地址：
2. 概念：由程序产生的，与段相关的偏移地址部分
3. 实模式下由“段基地址+段内偏移”组成；保护模式下由“段选择符+段内偏移”组成

// 段内偏移指明了从段开始的地方到实际地址之间的距离

1. 线性地址：
2. 概念：是逻辑地址到物理地址变换之间的中间层。程序代码会产生逻辑地址，或者说是段中的偏移地址，加上相应段的基地址就生成了一个线性地址。如果启用了分页机制，那么线性地址可以再变换产生一个物理地址。如果没有启动分页机制，那么线性地址就是物理地址。
3. 物理地址：
4. 概念：是目前CPU外部地址总线上的寻址物理内存的地址信号，是地址变换的最终结果地址。
5. 转化关系：

内存控制单元（MMU）通过一种称为分段单元的硬件电路把一个逻辑地址转化成一个线性地址，接着，通过叫分页单元的硬件电路把线性地址转换成一个物理地址。

# 硬件中的分段:

## 1、实模式和保护模式：

（1）实模式：处理器的一种操作模式。CPU启动的时候的模式。不能使用多线程，不能实现权限分级。还不能访问20为以上的地址线，也就是说，只能访问1M内存。

（2）保护模式：又称为虚拟地址保护模式。操作系统接管CPU以后，会使CPU进入保护模式，这时候可以发挥80x86的所有威力，包括权限分级、内存分页等功能。

## 2、段选择符和段寄存器：

（1）逻辑地址的组成：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：16位长的字段，称为段选择符。

偏移量：32位长的字段

（2）段寄存器：存放段选择符的寄存器。如cs、ss、ds、es、fs、gs

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段

除了这3个段，其他段寄存器可以指向任意数据段。

// 段寄存器仅仅存放段选择符

（3）段描述符：每一个段由一个8字节的段描述符表示，它描述了段的特征。段描述符存放在全局描述符表（Global Descriptor Table，GDT）或局部描述符表（Local Descriptor Table， LDT）中。 3个字段: 索引号 、 T1（表明指示器）、 RPL（请求者特权级）

// 段描述符在GDT或者LDT内的相对地址是有段选择符的索引号乘以8

（4）GDTR：是一个48位的寄存器，内容是一个32位的基地址和一个16位的段限，其中32位的基址是指GDT在内存中的地址。GDT在主存中的地址和大小存放在GDTR控制寄存器中。

（5）LDTR：由一个可见的16位寄存器和一个不可见的描述符寄存器组成。当前正在被使用的LDT地址和大小LDTR控制寄存器中。

（6）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字段值为2，s标志位置为0.

（7）段选择符的字段：

Index： 指定了放在GDT和LDT中的相应段描述符的入口

T1：指明了段描述符实在GDT中(T1==0)或在LDT中（T1==1）

RTL：请求者特权级。由RTL字段的值指定CPU的当前特权及（CPL）的值。如果RTL字段的值是3，则CPU的当前特权及（CPL）等于3，ds寄存器必须含有用户数据段的段选择符；如果RTL是0，则CPL=0，ds寄存器必须含有内核数据段的段选择符。

## 3、通过分段单元，把逻辑地址转化为线性地址：

（1）先检查段选择符的T1字段，查看段描述符保存在哪一个描述符表中。从相应的寄存器（gdtr寄存器或者ldtr寄存器）中获取描述符表的线性基地址。

（2）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），这个结果和gdtr和ldtr寄存器中的内容相加。（gdtr存放gdt的地址，ldtr同理）

（3）把逻辑地址的偏移量和段描述符Base字段（段首字节的线性地址）相加，就得到了线性地址。

## 4、指令和数据寻址：

（1）用户代码段和用户数据段：运行在用户态的所有linux进程都使用这两个段来对指令和数据寻址。

（2）内核代码段和内核数据段：运行在内核态的所有Linux进程都使用这两个段来对指令和数据寻址。

（3）所有段的线性地址都是从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。线性地址=逻辑地址+0x00000000，也就是相当于逻辑地址等于线性地址了。

另外的解释：段的基地址都为0x00000000，按照逻辑地址转化为线性地址的公式，

线性地址 = 0x00000000 + 段内偏移量，可以得出“在Linux下，逻辑地址和线性地址总是一致的”，即逻辑地址的偏移量字段的值和线性地址的值总是相同的。

## GDT：

1. 在单处理器系统中只有一个GDT，而在多处理器系统中每一个CPU对应一个GDT。所有的GDT都存放在cpu\_gdt\_table数组中，而所有GDT的地址和它们的大小（当初始化gdtr寄存器时使用）被存放在cpu\_gdt\_descr数组中。
2. 每一个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使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。
3. 每一个GDT中包含的18个段描述符指向下列的段：
4. 用户态和内核态下的代码段和数据段共4个
5. 任务状态段（TSS），每一个处理器有一个。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。所有任务状态段都顺序地存放在init\_tss数组中。第n个TSS描述符的Base字段指向init\_tss数组的第n个元素。

4、组成：

（1）1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。

（2）3个局部线程存储段（Thread-Local Storage、 TLS）段：这种机制允许多线程应用使用最多3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销一个TLS段。

（3）3个与高级电源管理（AMP）相关的段：由于是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。

（4）与支持即插即用（PnP）功能BIOS服务程序相关的5个段：在前一种情况下，就像前述与AMP相关的3个段的情况一样，由于BIOS例程使用段，所以当Linux的PnP设备驱动程序调研BIOS函数来检测PnP设备使用的资源时，就可以使用自定义的代码段和数据段。

（5）被内核用来处理“双重错误”异常的特殊TSS段

## LDT：

1. 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供大多数的进程共享。缺省的局部描述符表放在default\_ldt数组中。它包含5个项，但内核仅仅有效的使用了其中两个项:用于iBCS执行文件的调用门和Solaris/x86可执行文件的调用们。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。
2. 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，该CPU的GDT副本中的LDT表项相应地就被修改了。
3. 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：

1. 页：线性地址被分成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，页既指一组线性地址，又指在这组地址中的数据。
2. 页框：分页单元把所有的RAM分成固定长度的页框（有时叫做物理页），每一个页框包含一个页，也就是说一个页框的长度和一个页的长度一致。页框是主存的一部分，因此也是一个存储区域。页只是一个数据块，可以存放在任何页框或磁盘中。
3. 页表：把线性地址映射到物理地址的数据结构称为页表。页表存在主存中。并在启动分页单元之前必须由内核对页表进行适当的初始化。
4. 常规分页：

（1）32位的线性地址被分成3个域：

Directory（目录） 最高10位

Table（页表） 中间10位

Offset（偏移量） 最低12位

线性地址转化分两步完成，每一步都基于一种转化表，第一种转化表称为页目录表，第二种转化表称为页表。使用二级模式目的在于减少每个进程页所需要RAM的数量。

1. Directory字段决定页目录中的目录项，而目录项指向适当的页表。
2. Table字段决定页表项
3. 表项含有页所在页框的物理地址
4. Offset字段决定页框内的相对位置

// 68 页例题存在问题

## 高速缓存：

1. 通写：高速缓存控制器总是写RAM也写高速缓存。
2. 回写：只更新高速缓存，不改变RAM内容

## 转化后援缓冲器（TLB）：

1. 作用：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表项中，以便以后对同一个线性地址的引用可以快速得到转换。

## Linux中的分页：

1、4级分页：

页全局目录、页上级目录、页中间目录、页表

1. 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。线性地址因此被分为5个部分。

页全局目录PGD、页上级目录PUD、页中间目录PMD、页表PT

// 图 P76

1. 在32位的机器上，32位地址被分成了三部分；从软件的角度，由于引入了两部分，也就是说，共有五部分。——要让二层架构的硬件忽略掉这两部很简单，将页上级目录和页中间目录的长度设置为0就可以了。在64位的机器上，使用四层转化架构的CPU，此时不再把中间的两个设置为0。由此，兼容了32位机器和64位机器。
2. 硬件根本看不到所谓的PUD、PMD，所以本质上要求PGD的项，直接对应于一个PT。而不是再到PUD和PMD去转化。从软件的角度来讲，因为PGD、PUD、PMD中存放的一个项的大小相同。因此，在软件层面，所谓的PGD先到PUD，再到PMD的映射转化，就变成了保持原值不变，一一转手就可以了。这样，就实现了“逻辑上指向一个PUD，再指向一个PMD，但在物理地址上是直接指向相应的PT”
3. 简化页表处理的宏：
4. PAGE\_SHIFT: 指定Offset字段的位数。
5. PMD\_SHIFT: 指定线性地址的Offset字段和Table字段的总位数。
6. PUD\_SHIFT: 确定页上级目录项能映射的区域的大小的对数。
7. PGDIR\_SHIFT: 确定页全局目录项能映射的区域大小的对数
8. 页表处理：
9. pte\_t、pmd\_t、pud\_t、pgd\_t分别描述页表项、页中间目录项、页上级目录项、页全局目录项的格式。
10. 读页标志的函数：

// P79

## 物理内存布局：

1. 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哪些不可用。

## 进程页表：

1. 进程的线性地址空间分为两部分：
2. 从0x00000000到0xbfffffff的线性地址，无论进程运行在用户态还是内核态都可以寻址。
3. 从0xc0000000到0xffffffff的线性地址，只有内核态的进程才能寻址。
4. 当用户运行在用户态时，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，所产生的地址大小大于等于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。
5. 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。

## 内核页表：

1. 内核维持着一组自己使用的页表，驻留在所谓的主内核页全局目录中，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，主内核全局目录的最高目录项部分作为参考模型，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。
2. 内核如何初始化自己的页表？

第一个阶段：内核创建一个有限的地址空间，包括内核代码段和数据段、初始化页表和用于存放动态数据结构的共128KB大小的空间。这个最小的限度的地址空间仅够将内核装入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RAM并适当地建立分页表。

## 虚拟内存、内核空间、用户空间：

1. Linux虚拟内存的大小为2^32（在32位的x86机器上），内核将这4G字节的空间分为两部分。最高的1G字节（从虚拟地址0xC0000000到0xFFFFFFFF）供内核使用，称为“内核空间”。而较低的3G字节（从虚拟地址0x00000000到0xBFFFFFFF），供各个进程使用，称为“用户空间”。因为每个进程可以通过系统调用进入内核空间，因此，Linux内核空间由系统内的所有进程共享。于是，从具体进程的角度来看，每个进程可以拥有4G字节的虚拟地址空间（也叫虚拟内存）。
2. 每个进程有各自的私有用户空间（0~3G），这个空间对系统中的其他进程是不可见的。最高的1GB内核空间则为所有进程以及内核所共享。另外，进程的“用户空间”也叫“地址空间”。
3. 用户空间不是进程共享的，而是进程隔离的。每个进程最大都可以有3GB的用户空间。一个进程对其中的一个地址的访问，与其他进程对于同一个地址的访问，绝对不冲突。比如，一个进程从其用户空间地址0x1234ABCD处可以读出整数8，而另外一个进程从其用户空间的地址0x1234ABCD处可以读出整数20，这取决于进程自身的逻辑。
4. 任意一个时刻，在一个CPU上只有一个进程在运行。所以对于此CPU来讲，在这一个时刻，整个系统只存在一个4GB的虚拟地址空间，这个虚拟地址空间是面向此进程的。当进程发生切换的时候，虚拟地址空间也随着切换。由此可以看出，每个进程都有自己的虚拟地址空间，只有此进程运行的时候，其虚拟地址空间才被运行它的CPU所知。在其他时刻，其虚拟地址空间对于CPU来讲，是不可知的。所以尽管每个进程都可以有4GB的虚拟地址空间，但在CPU眼中，只有一个虚拟地址空间。虚拟地址空间的变化，随着进程切换而变化。
5. 如果给出的页表不同，那么CPU将某一虚拟地址空间中的地址转化成的物理地址就会不同。所以系统分别为每一个进程都建立页表。将每个进程的虚拟地址空间根据自己的需要映射到物理地址空间上。既然某一个时刻在某一CPU上只能有一个进程在运行，那么当进程发生切换的时候，将页表更换为相应进程的页表，这就可以实现每个进程都有自己的虚拟地址空间而互不影响。所以，在任意时刻，对于一个CPU来说，只需要有当前进程的页表，就可以实现其虚拟地址到物理地址的转化。

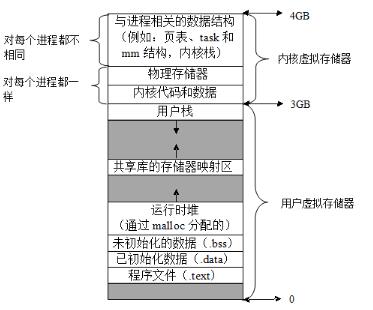
## 内核空间到物理内存的映射：

1. 内核空间对所有的进程都是共享的，其中存放的是内核的代码和数据，而进程空间中存放的是用户程序的代码和数据，不管是内核程序还是用户程序，它们被编译和连接以后，所形成的指令和符号都是虚拟地址，而不是物理内存中的物理地址。
2. 虽然内核空间占据了每个虚拟空间中的最高1GB字节，但映射到物理内存却总是从最低地址（0x00000000）开始的，之所以这么规定，是为了在内核空间与物理内存之间建立简单的线性映射关系。其中，3GB（0xC0000000）就是物理地址与虚拟地址之间的位移量，在Linux代码中就叫做PACE\_OFFSET。
3. 对于内核空间而言，给定一个虚拟地址x，其物理地址为“x – PACE\_OFFSET”，给定一个物理地址x，其虚拟地址为“x + PACE\_OFFSET”
4. 宏\_\_pa()仅仅把一个内核空间的虚拟地址映射带物理地址，而绝不适用于用户空间，用户空间的地址映射要复杂得多，它通过分页机制完成。

# 补充：

## Linux的内存管理：

1. Linux的进程的虚拟空间及其划分

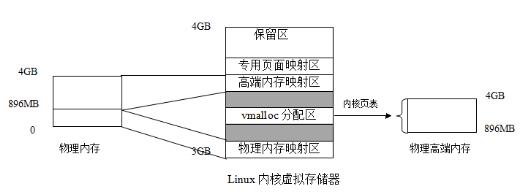


一个Linux进程的虚拟存储器

1. Linux内核虚拟存储器：

Linux中1GB的内核虚拟映射存储空间又被划分为物理映射区、虚拟内存分配区、高端页面映射区、专用页面映射区和系统保留映射区这几个区域。

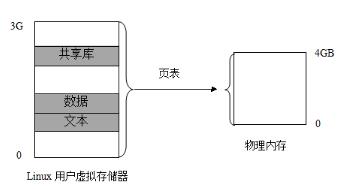
一般情况下，物理内存映射区最大长度为896MB，系统的物理内存被顺序映射到物理内存映射区中。当系统物理内存大于896MB时，超过系统物理内存那部分内存被称为高端内存（小于896MB的系统物理内存被称为常规内存），内核在存取高端内存时，必须将它们映射到高端内存映射区中。下图反映出Linux内核虚拟存储器与物理内存之间的关系。



注意：物理内存中0~896MB区域通常由内核使用，当然内核不用时用户程序可以使用；896MB以上的区域通常由用户程序来使用。

1. 用户虚拟存储器：

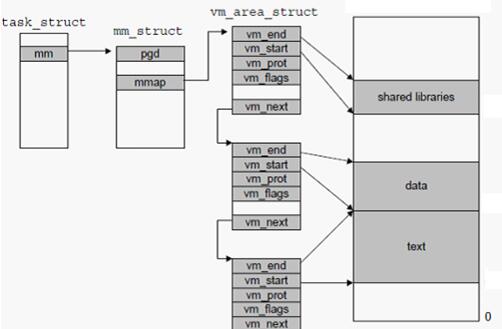
Linux用户虚拟存储器总是通过页表访问内存，绝不会直接访问。



1. 进程空间的描述：

内核为系统中每个进程维护一个单独的任务结构task\_struct。任务结构中的元素包含指向内核运行该进程所需要的所有信息（例如，PID、指向用户栈的指针、可执行目标文件的名字及程序计数器）。

task\_struct中一条字段mm（拥有的内存描述符）指向mm\_struct，它描述进程使用的地址空间.pgd指向第一级页表（页全局目录）的基址（第一个存储单元的起始地址），而mmap指向一个vm\_area\_struct（线性区）的链表。



## Linux的内存映射：

当可执行文件准备运行时，可执行文件的内容仅仅映射到了对应进程的虚拟地址空间中，而没有调入物理内存。当程序开始运行并使用到这部分时，Linux才通过缺页中断把它们从磁盘上调入内存。这种将文件连接到进程虚拟地址空间的过程称为内存映射。