一、逻辑地址转线性地址

机器语言指令中出现的内存地址，都是逻辑地址，需要转换成线性地址，再经过 MMU(CPU 中的内存管理单元)转换成物理地址才能够被访问到。

我们写个最简单的 hello world 程序，用 gccs 编译，再反编译后会看到以下指令：

```
mov        0x80495b0, %eax
```

这里的内存地址 0x80495b0 就是一个逻辑地址，必须加上隐含的 DS 数据段的基地址，才能构成线性地址。也就是说 0x80495b0 是当前任务的 DS 数据段内的偏移。

在 x86 保护模式下，段的信息（段基线性地址、长度、权限等）即**段描述符**占 8 个字节，段信息无法直接存放在段寄存器中（段寄存器只有 2 字节）。Intel 的设计是段描述符集中存放在 GDT 或 LDT 中，而**段寄存器存放的是段描述符在 GDT 或 LDT 内的索引值**(index)。

Linux 中逻辑地址等于线性地址。为什么这么说呢？因为 Linux 所有的段（用户代码段、用户数据段、内核代码段、内核数据段）的线性地址都是从 0x00000000 开始，长度 4G，这样 线性地址=逻辑地址+ 0x00000000，也就是说逻辑地址等于线性地址了。

这样的情况下 **Linux 只用到了 GDT**，不论是用户任务还是内核任务，都没有用到 LDT。GDT 的第 12 和 13 项段描述符是 __KERNEL_CS 和__KERNEL_DS，第 14 和 15 项段描述符是 __USER_CS 和__USER_DS。内核任务使用__KERNEL_CS 和__KERNEL_DS，所有的用户任务共用__USER_CS 和__USER_DS，也就是说不需要给每个任务再单独分配段描述符。内核段描述符和用户段描述符虽然起始线性地址和长度都一样，但 DPL(描述符特权级)是不一样的。__KERNEL_CS 和__KERNEL_DS 的 DPL 值为 0（最高特权），__USER_CS 和__USER_DS 的 DPL 值为 3。

用 gdb 调试程序的时候，用 info reg 显示当前寄存器的值：

```
cs                    0x73          115

ss                    0x7b          123

ds                    0x7b          123

es                    0x7b          123
```

可以看到 ds 值为 0x7b，转换成二进制为 00000000 01111011，TI 字段值为 0，表示使用 GDT，GDT 索引值为 01111，即十进制 15，对应的就是 GDT 内的 __USER_DATA 用户数据段描述符。

从上面可以看到，Linux 在 x86 的分段机制上运行，却通过一个巧妙的方式绕开了分段。

Linux 主要以分页的方式实现内存管理。

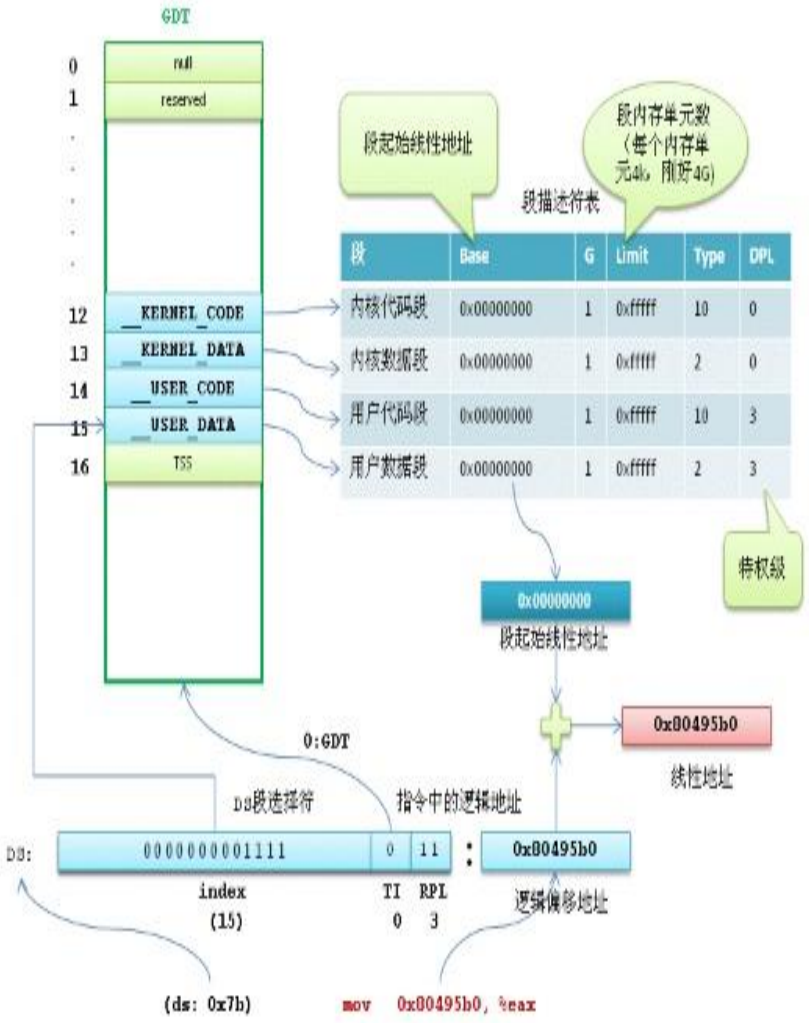

图1 逻辑地址转线性地址

二、线性地址转物理地址

前面说了 Linux 中逻辑地址等于线性地址，那么线性地址怎么对应到物理地址呢？这个大家都知道，那就是通过分页机制，具体的说，就是通过页表查找来对应物理地址。

准确的说分页是 CPU 提供的一种机制，Linux 只是根据这种机制的规则，利用它实现了内存管理。

在保护模式下，控制寄存器 CR0 的最高位 PG 位控制着分页管理机制是否生效，如果 PG=1，分页机制生效，需通过页表查找才能把线性地址转换物理地址。如果 PG=0，则分页机制无效，线性地址就直接做为物理地址。

分页的基本原理是把内存划分成大小固定的若干单元，每个单元称为一页（page），每页包含 4k 字节的地址空间（为简化分析，我们不考虑扩展分页的情况）。这样每一页的起始地址都是 4k 字节对齐的。为了能转换成物理地址，我们需要给 CPU 提供**当前任务的**线性地址转物理地址的查找表，即页表(page table)。注意，**为了实现每个任务的平坦的虚拟内存，每个任务都有自己的页目录表和页表**。

为了节约页表占用的内存空间，x86 将线性地址通过页目录表和页表两级查找转换成物理地址。

32 位的线性地址被分成 3 个部分：

最高 10 位 Directory 页目录表偏移量，中间 10 位 Table 是页表偏移量，最低 12 位 Offset 是物理页内的字节偏移量。

页目录表的大小为 4k（刚好是一个页的大小），包含 1024 项，每个项 4 字节（32 位），项目里存储的内容就是**页表的物理地址**。如果页目录表中的页表尚未分配，则物理地址填 0。

页表的大小也是 4k，同样包含 1024 项，每个项 4 字节，内容为最终物理页的物理内存起始地址。

每个活动的任务，必须要先分配给它一个页目录表，并把页目录表的物理地址存入 cr3 寄存器。页表可以提前分配好，也可以在用到的时候再分配。

还是以 mov 0x80495b0, %eax 中的地址为例分析一下线性地址转物理地址的过程。

前面说到 Linux 中逻辑地址等于线性地址，那么我们要转换的线性地址就是 0x80495b0。转换的过程是由 CPU 自动完成的，Linux 所要做的就是准备好转换所需的页目录表和页表（假设已经准备好，给页目录表和页表分配物理内存的过程很复杂，后面再分析）。

内核先将当前任务的页目录表的物理地址填入 cr3 寄存器。

线性地址 0x80495b0 转换成二进制后是 0000 1000 0000 0100 1001 0101 1011 0000，最高 10 位 0000 1000 00 的十进制是 32，CPU 查看页目录表第 32 项，里面存放的是页表的物理地址。线性地址中间 10 位 00 0100 1001 的十进制是 73，页表的第 73 项存储的是最终物理页的物理起始地址。物理页基地址加上线性地址中最低 12 位的偏移量，CPU 就找到了线性地址最终对应的物理内存单元。

我们知道 Linux 中用户进程线性地址能寻址的范围是 0 — 3G，那么是不是需要提前先把这 3G 虚拟内存的页表都建立好呢？一般情况下，物理内存是远远小于 3G 的，加上同时有很多进程都在运行，根本无法给每个进程提前建立 3G 的线性地址页表。Linux 利用 CPU 的一个机制解决了这个问题。进程创建后我们可以给页目录表的表项值都填 0，CPU 在查找页表时，如果表项的内容为 0,则会引发一个缺页异常，进程暂停执行，Linux 内核这时候可以通过一系列复杂的算法给分配一个物理页，并把物理页的地址填入表项中，进程再恢复执行。当然进程在这个过程中是被蒙蔽的，它自己的感觉还是正常访问到了物理内存。

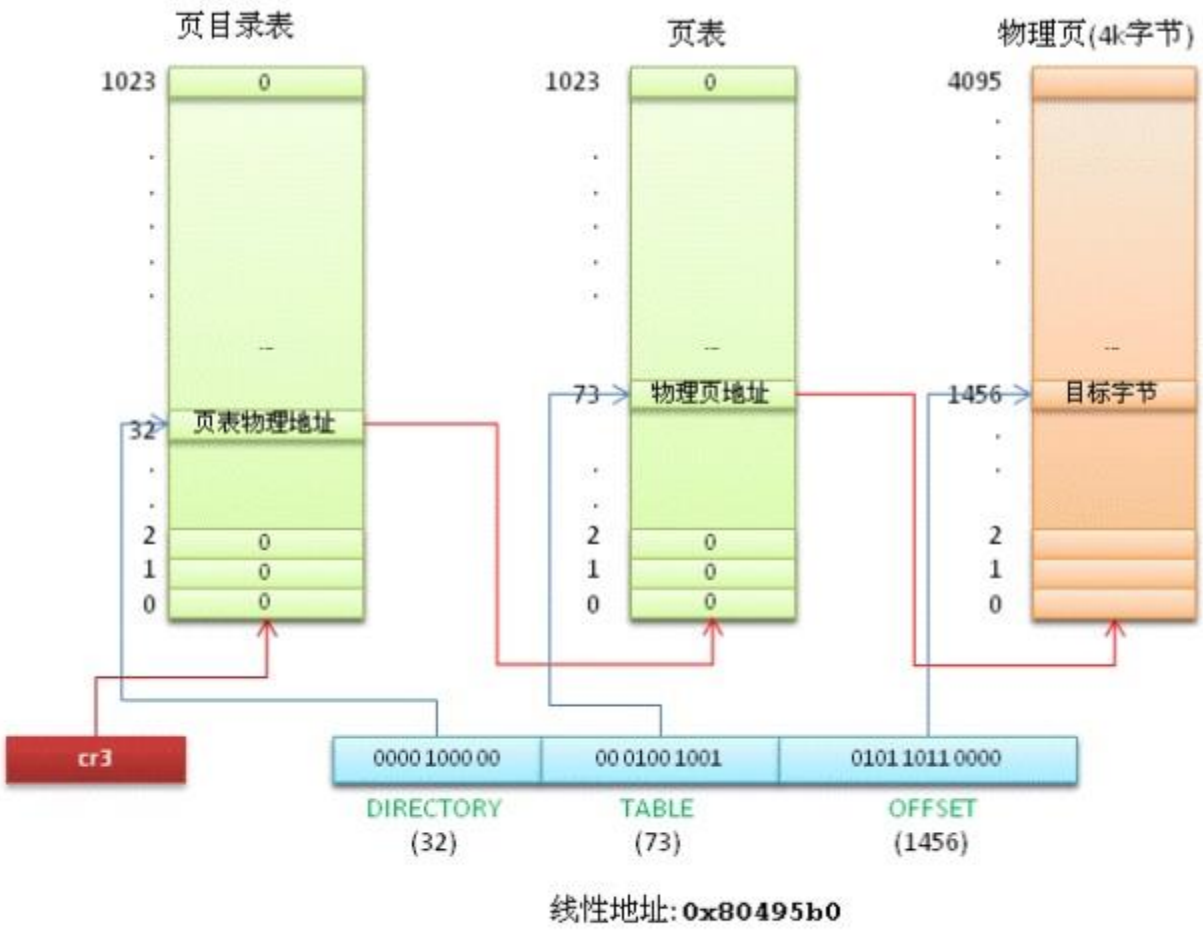

图2 线性地址转物理地址