> 作 者：道哥，10+年的嵌入式开发老兵，专注于：C/C++、嵌入式、Linux。
>
> 关注下方公众号
>
> > 回复【书籍】，获取 Linux、嵌入式领域经典书籍。
> >
> > 回复【PDF】，获取所有原创文章的 PDF 格式汇总。

目录

在x86系统中，为了能够更加充分、灵活的使用物理内存，把物理内存按照4KB的单位进行分页。

然后通过中间的映射表，把连续的虚拟内存空间，映射到离散的物理内存空间。映射表中的每一个表项，都指向一个物理页的开始地址。

但是这样的映射表有一个明显的缺点：映射表自身也是需保存在物理内存中的，它使用了多达4MB的物理内存空间(每个表项4个字节，一共有4G/4K个表项)。

为了解决这个问题，x86处理器使用了两级转换：页目录和页表。

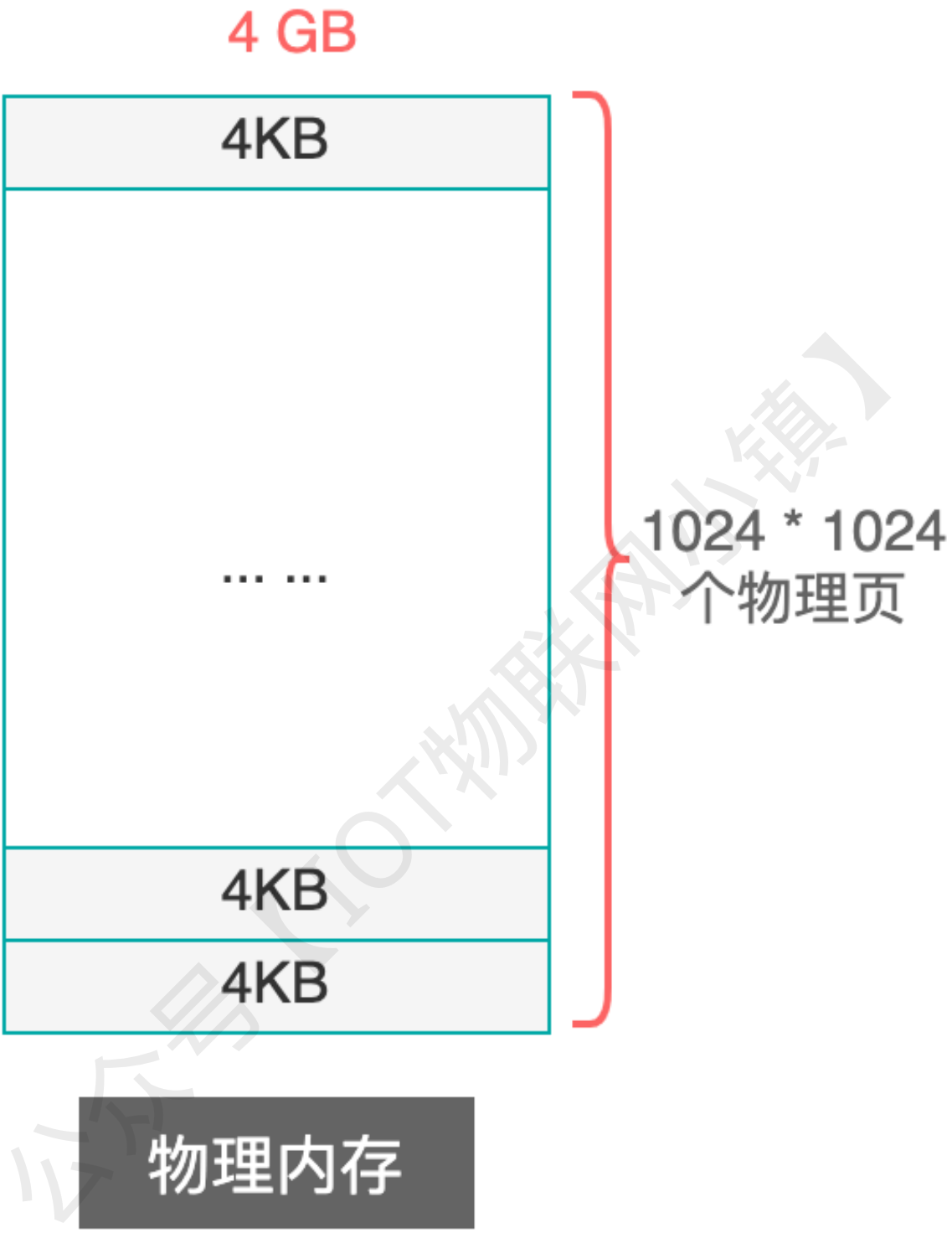
这篇文章，我们就把这个最重要的内存管理机制搞定。

页表

在一个32位的系统中，物理内存的最大可表示空间就是0xFFFF_FFFF，也就是4GB。

虽然实际安装的物理内存可能远远没有这么大，但是在设计内存管理机制的时候，还是需要按照最大的可寻址范围来进行设计的。

按照一个物理页4KB的单位来划分，4GB空间可以分割为1024 * 1024个物理页:



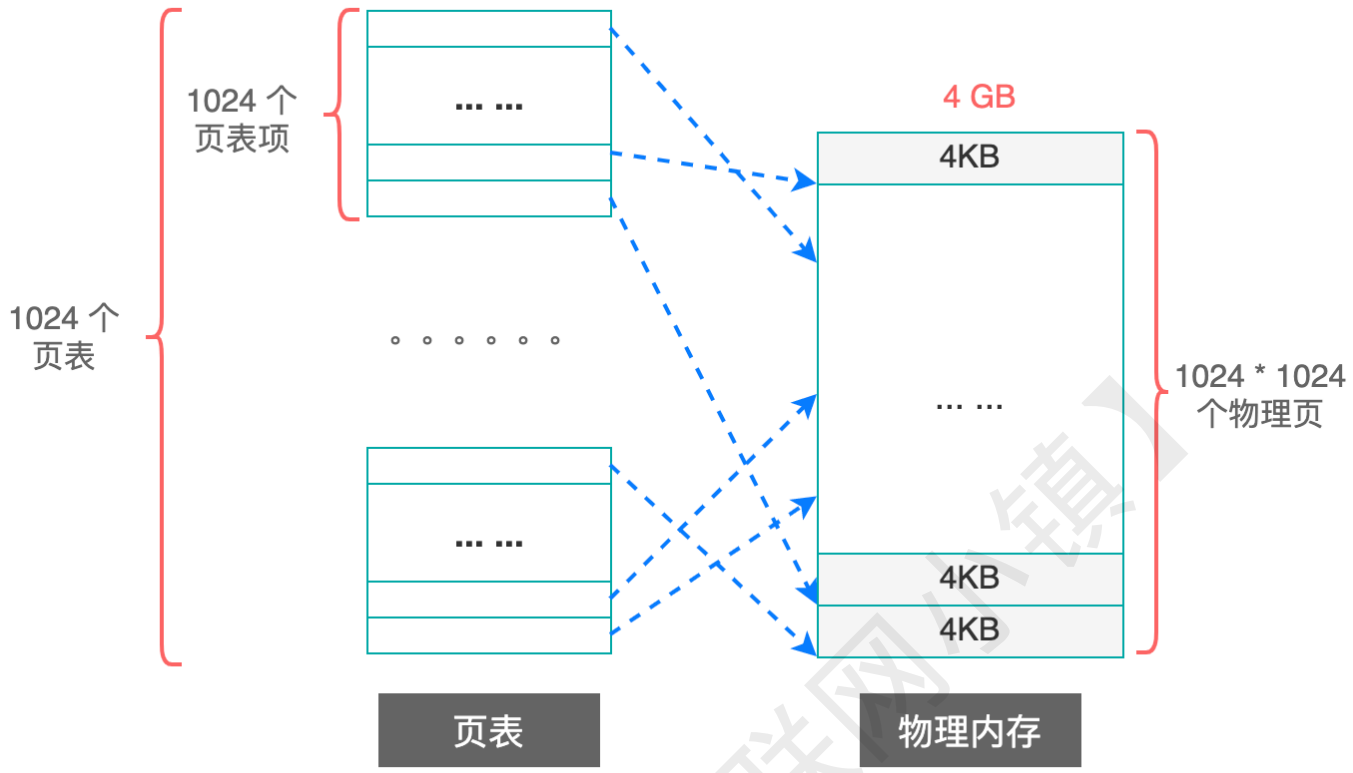
在上一篇文章中，使用单一的映射表来指向这些物理页，导致了映射表自身占据了太多的物理内存空间。

> 一个用户程序中定义的几个段，可能实际上只使用了很小的空间，完全用不到 4 GB。但是仍然需要为它分配多达 4GB 的物理内存空间来保存这个映射表，很浪费。

为了解决这个问题，可以把这个单一映射表进行拆分成1024个：

> 1. 每一个映射表中，只有 1024 个表项，每一个表项仍然指向一个物理页的起始地址；

> 2. 一共使用 1024 个这样的映射表;



这样一来，1024(每个表中的表项个数) * 1024(表的个数)，仍然可以覆盖4GB的物理内存空间。

这里的每一个表，就称作页表，所以一共有1024个页表。

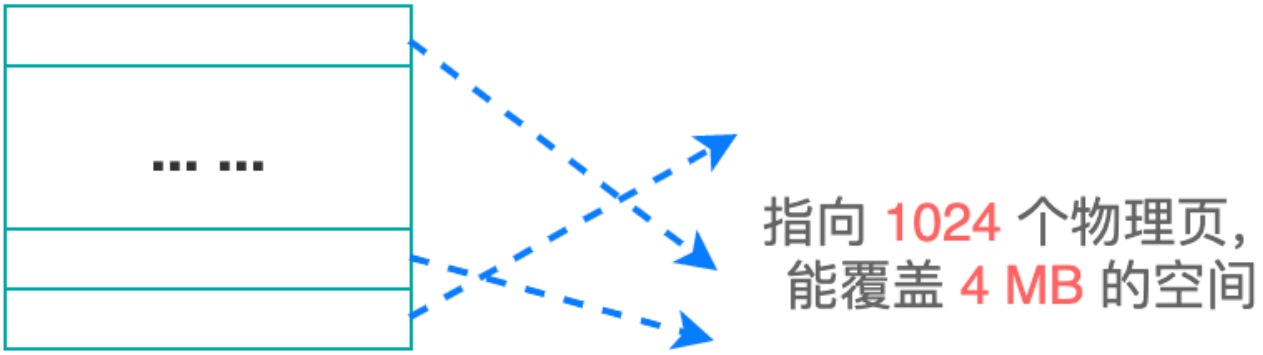
每一个页表项占用4个字节，一个页表中1024个表项，就占用4KB的物理内存空间，正好是一个物理页的大小。

也许有的小伙伴就开始算账了：一个页表自身占用4KB，那么1024个页表一共就占用了4MB的物理内存空间，仍然是很多啊?

是的，从总数上看是这样，但是：一个程序是不可能完全使用全部的 4GB 空间的，也许只要几十个页表就可以了。

例如：一个用户程序的代码段、数据段、栈段，一共就需要10 MB的空间，那么使用3个页表就足够了，加上页目录，一共需要16 KB的空间。

> 计算过程:
>
> 每一个页表项指向一个 4KB 的物理页，那么一个页表中 1024 个页表项，一共能覆盖 4MB 的物理内存;
>
> 那么 10MB 的程序，向上取整之后(4MB 的倍数，就是 12 MB)，就需要 3 个页表就可以了。



一个页表

记住上图中的一句话：一个页表，可以覆盖 4MB 的物理内存空间(1024 * 4 KB)。

页表中，每一个表项的格式如下：

31 ... 12	11 10 9	8	7	6	5	4	3	2	1	0
页的物理地址 (bit[31:12])	AVL	G	pat	D	A	pcd	pwt	US	RW	P

页表项格式

注意下面的这几个属性：

> P(Present): 存在位。1 - 物理页存在; 0 - 物理页不存在;
>
> RW(Read/Write): 读/写位。1 - 这个物理页可读可写; 0 - 这个物理页只可读;
>
> D(Dirty): 脏位。表示这个物理页中的数据是否被写过;

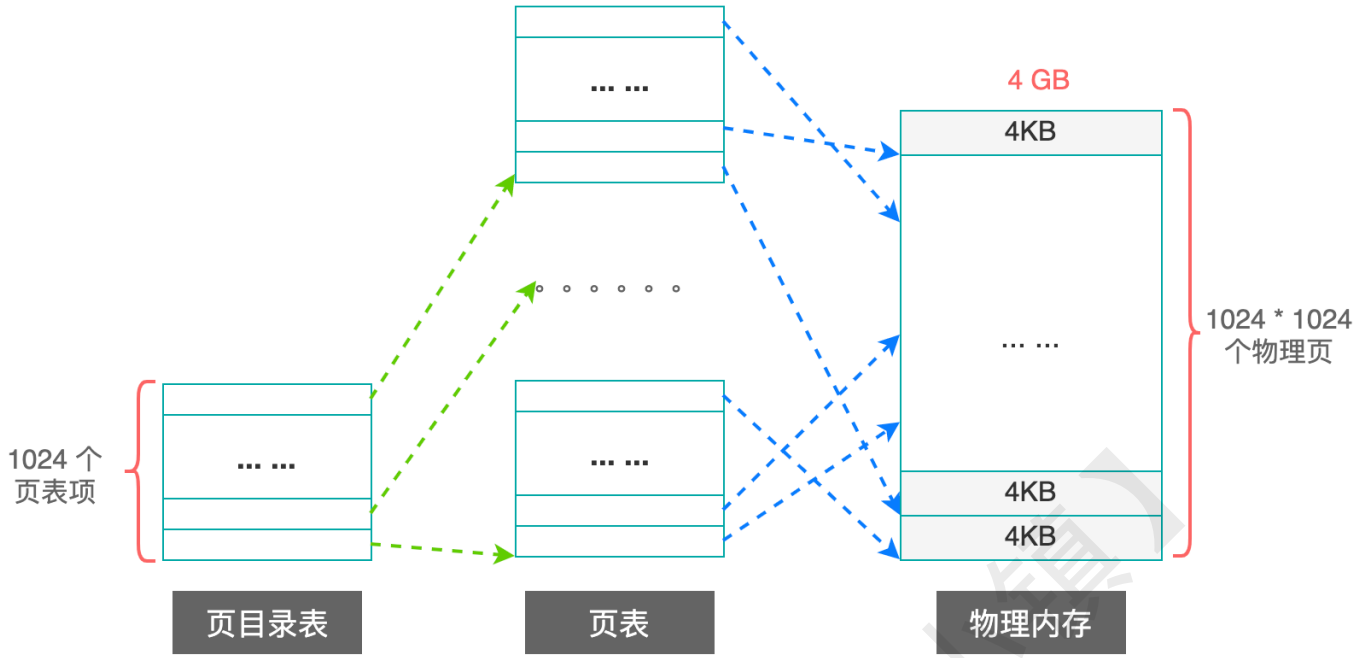
页目录

现在，每一个物理页，都被一个页表中的表项来指向了，那么这1024个页表的地址，应该由谁来指向呢?

这就是页目录表！

在页目录中，每一个表项指向一个页表的开始地址(物理地址)。

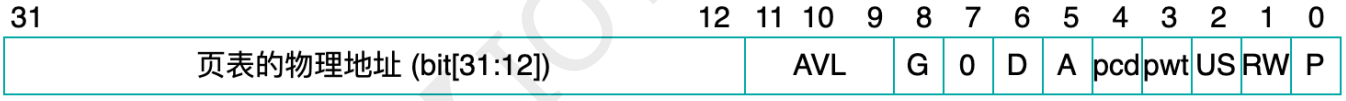
> 操作系统在加载用户程序的时候，不仅仅需要分配物理内存，来存放程序的内容; 而且还需要分配物理内存，用作程序的页目录和页表。



再来算算账：

上文说过：每一个页表覆盖4MB的内存空间，那么页目录中一共有1024个表项，指向1024个页表的物理地址，那么页目录能覆盖的内存空间就是1024 * 4MB，也就是4GB，正好是32位地址的最大寻址范围。

页目录中，每一个表项的格式如下：



页目录表项格式

其中的属性字段，与页表中的属性类似，只不过它的描述对象是页表。

还有一点：每一个用户程序都有自己的页目录和页表！下文有详细说明。

相关寄存器

现在，所有页表的物理地址被页目录表项指向了，那么页目录的物理地址，处理器是怎么知道的呢?

答案就是：CR3 寄存器，也叫做: PDBR(Page Table Base Register)。

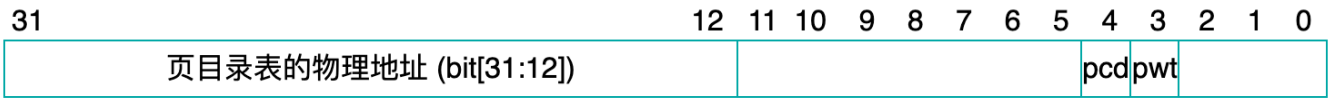
当操作系统调度任务的时候，每个任务(程序)都有自己的页目录和页表，并且CR3寄存器被记录在任务的TSS段中。

处理器就会把CR3寄存器更新为新任务的页目录开始地址，这样就相当于每个任务都是在自己的地址空间中来执行了。

当处理器在获取指令、操作数据时，操作的是线性地址。页处理单元就会从CR3 寄存器中开始查表，把这个线性地址最终转换成物理地址。

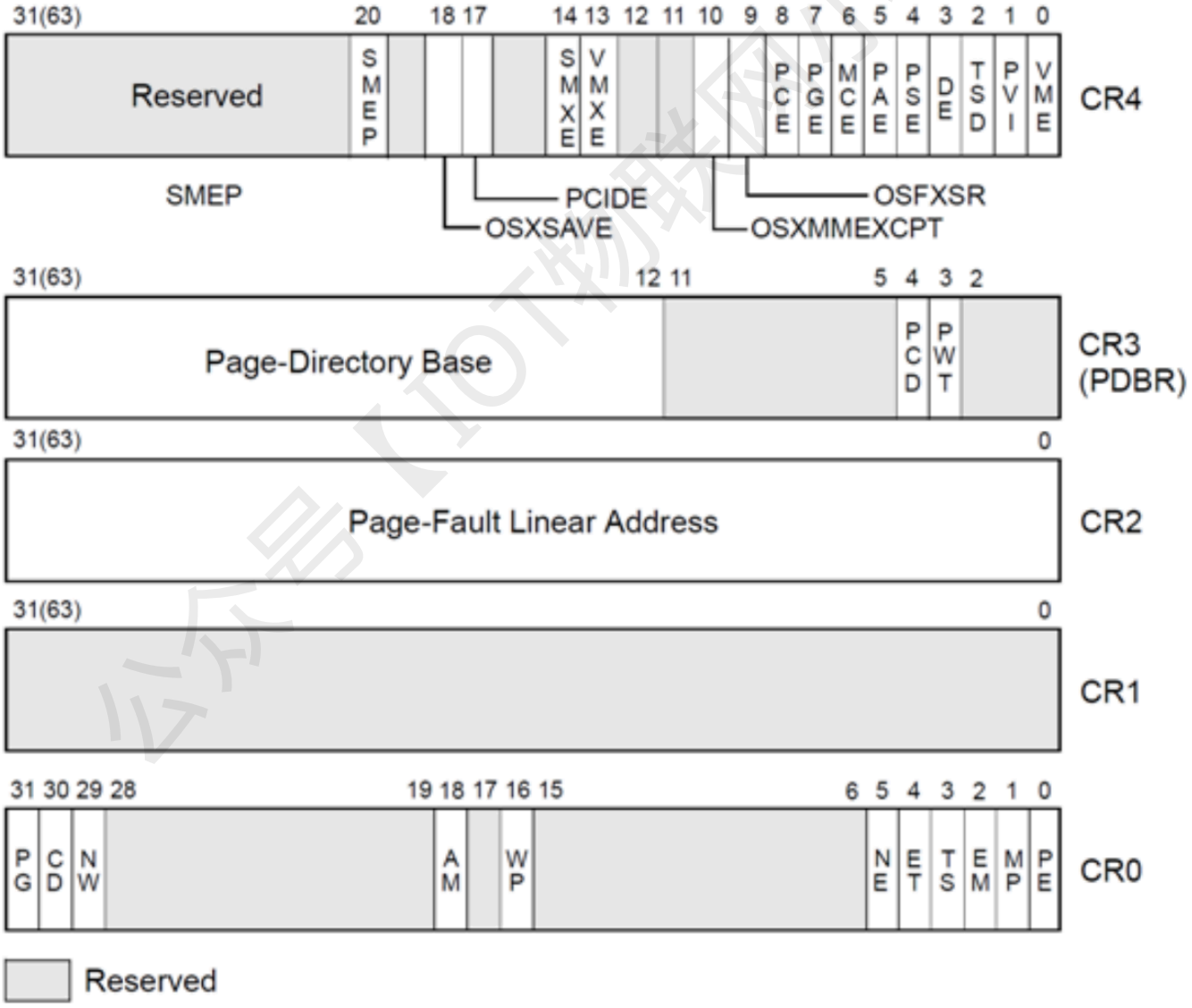
> 当然，处理器中还有一个快表，用来加快从线性地址到物理地址的转换过程。

CR3寄存器的格式如下：



CR3 寄存器

顺便把官网上的其他几个控制寄存器都贴出来：



其中，CR0寄存器的最高位PG，就是开启页处理单元的开关。

也即是说：

当系统上电之后，刚开始的地址寻址方式一直是 [段:偏移地址] 的方式。

当启动代码准备好页目录和页表之后，就可以设置 CR0.PG = 1。

此时，处理器中的页处理单元就开始工作了：面对任何一个线性地址，都要经过页处理单元之后，才得到一个物理地址。

加载用户程序时: 物理页分配过程

在之前的文章中，介绍过一个用户程序被操作系统加载的全过程，简述如下：

1. 读取程序 header 信息，解析出程序的总长度，从任务自己的虚拟内存中分配一块足够的连续空间;
2. 分配一个空闲物理页，用作程序的页目录，页目录的地址会记录在稍后创建的 TSS 段中;
3. 使用虚拟内存中的线性地址，分配一个物理页(4 KB)，登记到页目录和页表中;
4. 从硬盘上读取 8 个扇区的数据(每个扇区 512 字节)，存放到刚才分配的物理页中;
5. 检查程序内容是否读取完毕：是-进入第 6 步；否-返回到第 3 步;
6. 为用户程序创建一些必要的内核数据结构，比如：TSS、TCB/PCB 等等;
7. 为用户程序创建 LDT，并且在其中创建每一个段描述符;
8. 把操作系统的页目录中高端地址部分的表项，复制给用户程序的页目录表。

> 这样的话，所有用户程序的页目录中，高端地址的表项都指向相同的页表地址，就达到了共享“操作系统空间”的目的。

这里主要聊一下第3步，假设用户程序文件在硬盘上的长度是`20 MB`，实际的物理内存是`1 GB`。

> 可以先计算一下：页目录中，每一个表项覆盖的空间是 4 MB，那么 20 MB的数据，需要 5 个表项就可以了。

在初始状态，页目录中的所有表项都是空的，其中的P位都是为`0`，表示页表都不存在。

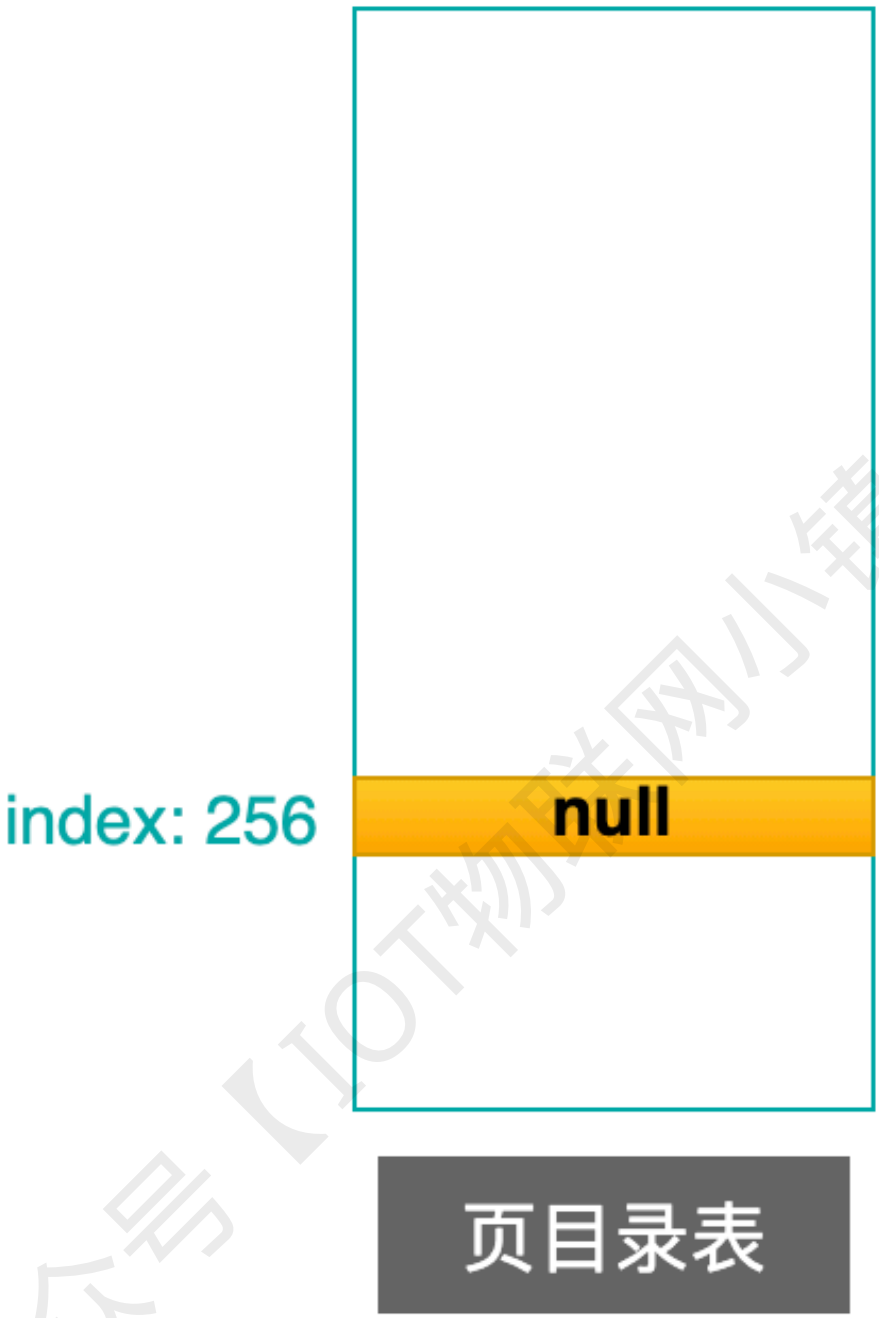
操作系统首先从虚拟内存中，分配一块`20 MB`的空间，假设从`1 GB (0x4000_0000)`的地址处开始吧，这个地址是线性地址。

> 注意：在“平坦”型分段模型下，线性地址等于虚拟地址。

`0x4000_0000 = 0100_0000_0000_0000___0000_0000_0000_0000`

前`10`位表示该线性地址在页目录中的索引，中间`10`位表示页表中的索引，最后`12`位表示物理页中的偏移地址。

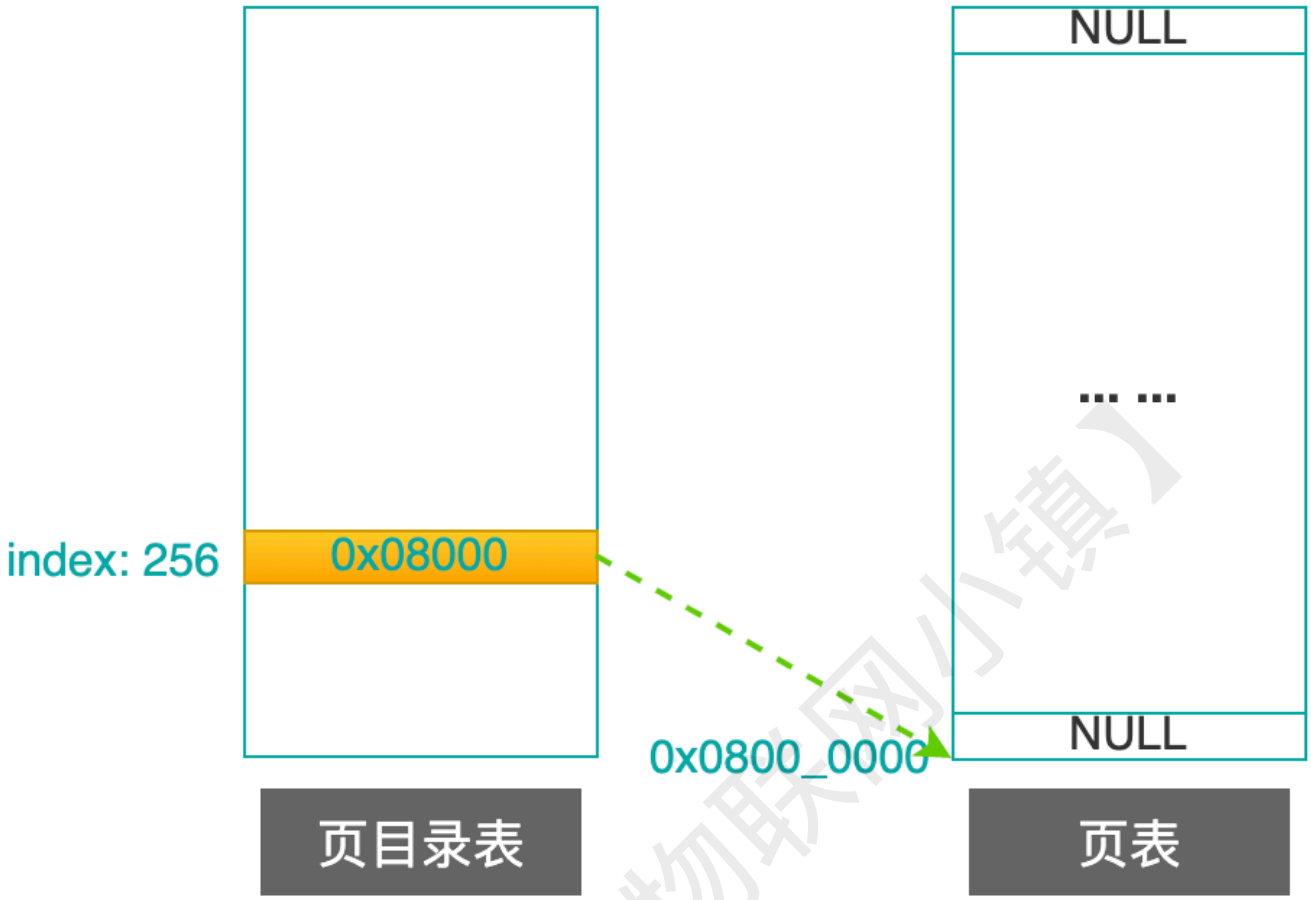
因此，前`10`位就是`0100_0000_00`，表示这个线性地址位于页目录中的第`256`个表项：



操作系统发现这个表项中为空，没有指向任何一个页表。

于是就从物理内存中，找一个空闲的物理页，用作页目录中第256个表项指向的页表。

假设在物理内存上128 MB (0x0800_0000)的地址处，找到一个空闲的物理页，用作这个页表。



把页表中的1024个表项全部清空，并且把页表的物理地址0x0800_0000，登记在页目录中的第256个表项中：0x08000。

因为页表的物理地址一定是4KB对齐的(最后的12位全部为0)，所以页目录的表项中只需要记录页表地址的高 20 位即可。

现在，页表也有了，下面就是分配一个物理页来存储程序的内容。

假设在刚才那个物理页(用作页表的那个)的上面，又找到一个空闲的物理页，地址是：0x0800_1000。

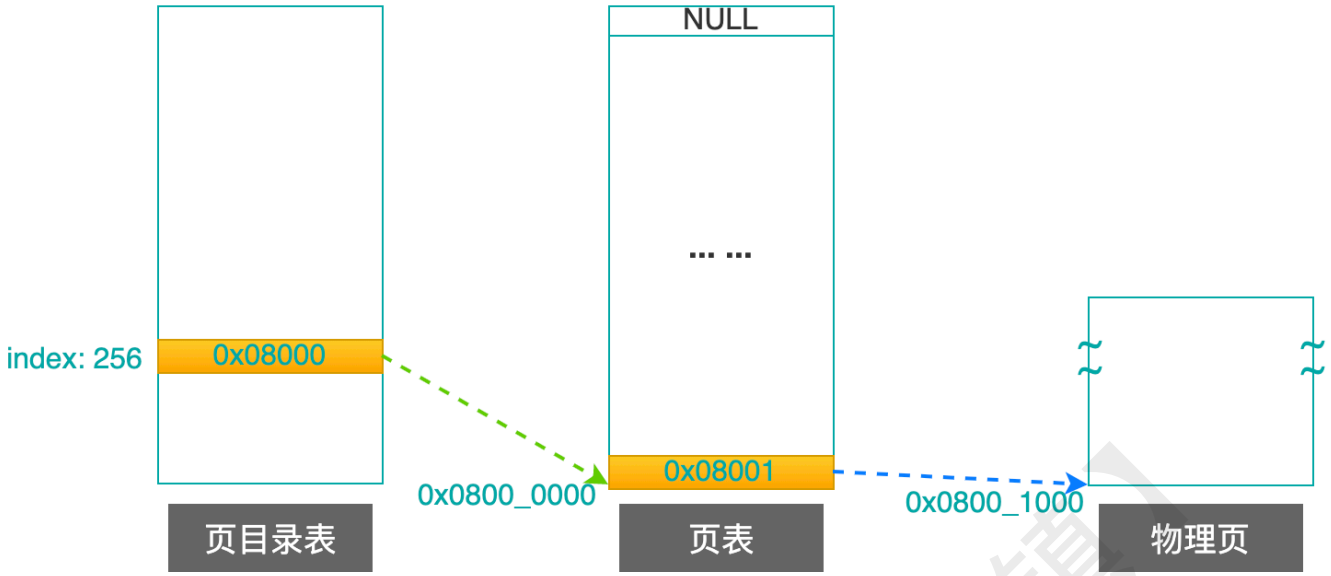
此时，这个用于存放程序内容的物理页的地址，就需要记录在页表的一个表项中了。

那么应该放在哪个索引位置呢?

需要根据线性地址的中间 10 位来确定：

0x4000_0000 = 0100_0000_0000_0000___0000_0000_0000_0000

中间10位的全部是0，说明索引值就是0，也就是说页表中的第0个表项，保存这个物理页的地址：



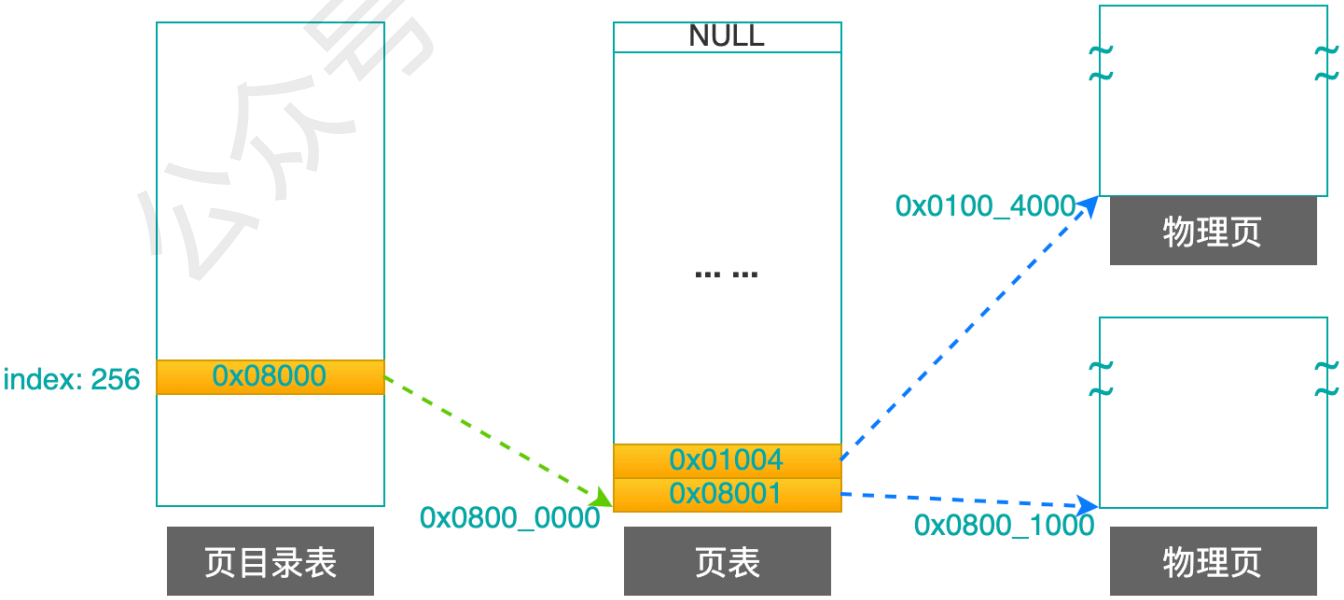
一个物理页的大小是4 KB，硬盘上一个扇区的大小是512 B，那么从硬盘上连续读取8个扇区的数据就可以把一个物理页写满。

当读取了一个物理页的内容后，通过计算发现用户程序内容还没有读取完，于是继续重复以上流程。

1. 线性地址增加 4KB：0x4000_1000 = 0100_0000_0000_0000___0001_0000_0000_0000;
2. 前 10 位没有变，仍然是页目录中的第 256 个表项，发现这个表项指向的页表已经存在了，于是就不用再分配物理页用作页表了;
3. 分配一个空闲物理页，用于存放程序内容，假设在 0x0100_4000处找到一个，把这个地址登记在页表中;

   > 此时，线性地址的中间 10 位的索引值是 1，表示页表中的第 1 个表项。

4. 从硬盘上读取 8 个扇区的数据，写入这个物理页;



因为页目录中一个表项所覆盖的范围是4 MB(也就是一个页表中1024个表项所指向的物理页总和)，所以当读取了4 MB的程序内容之后，这个页表中的所有表项就被填满了。

此时，读取的程序内容所占用的【线性地址】空间是：0x4000_0000 ~ 0x403F_FFFF。

下面再继续读取新内容时，就从 0x4040_0000 这个线性地址处开始存放，读取过程与上面都是一样的：

1. 确定页目录表项：

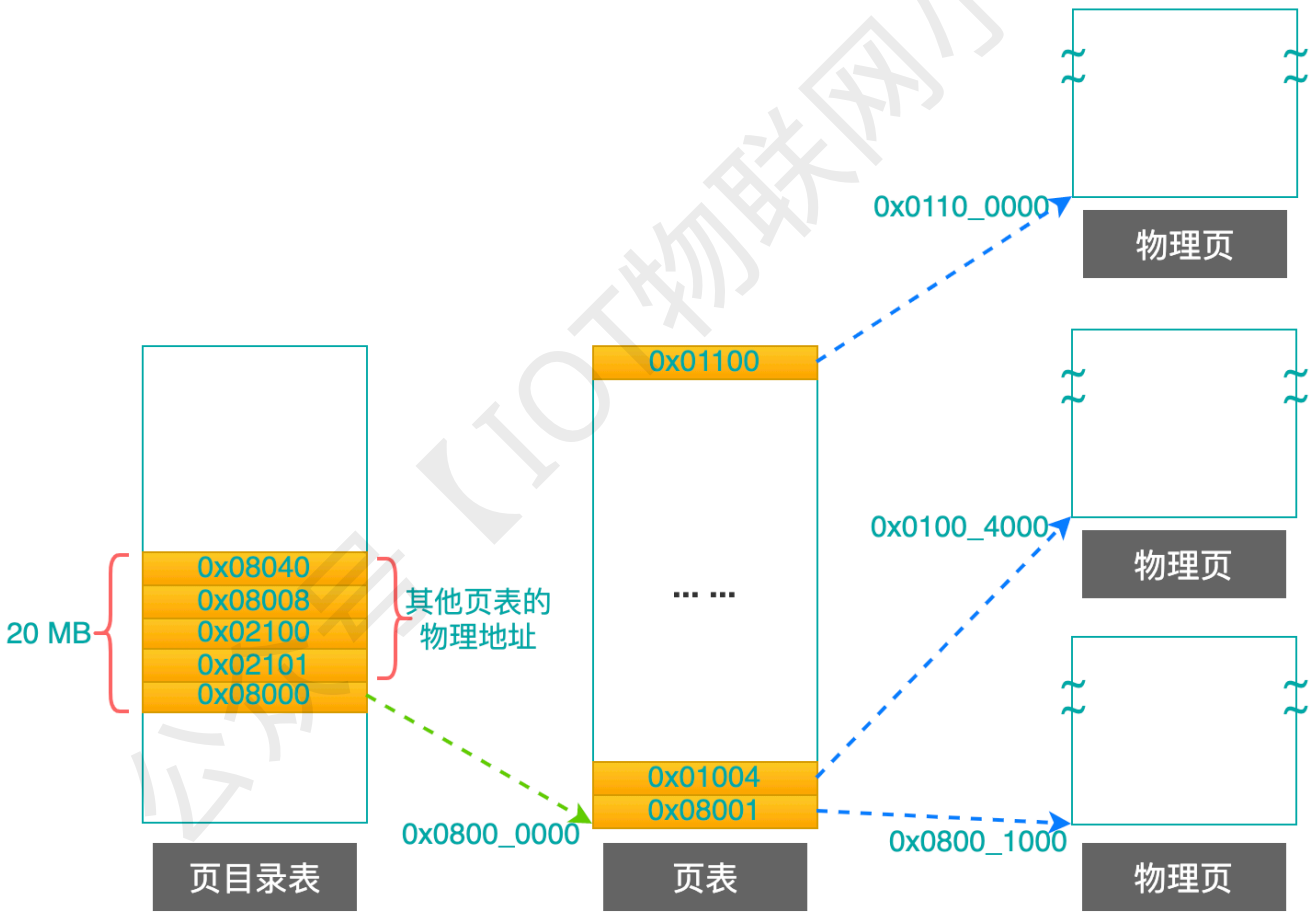
   > 0x4040_0000 = 0100_0000_0100_0000___0000_0000_0000_0000，前 10 位的索引值是 257;

2. 发现 257 这个表项为空，于是分配一个空闲的物理页，用作它的页表;
3. 分配一个物理页，用作存储程序内容，并把这个物理页的地址记录在页表中;

   > 线性地址 0x4040_0000 的中间 10 位的索引值是 0，所以放在页表的第一个表项中;

后面的过程就不再唠叨了，一样一样的~~

最终的页目录和页表的布局，类似下面这张图：



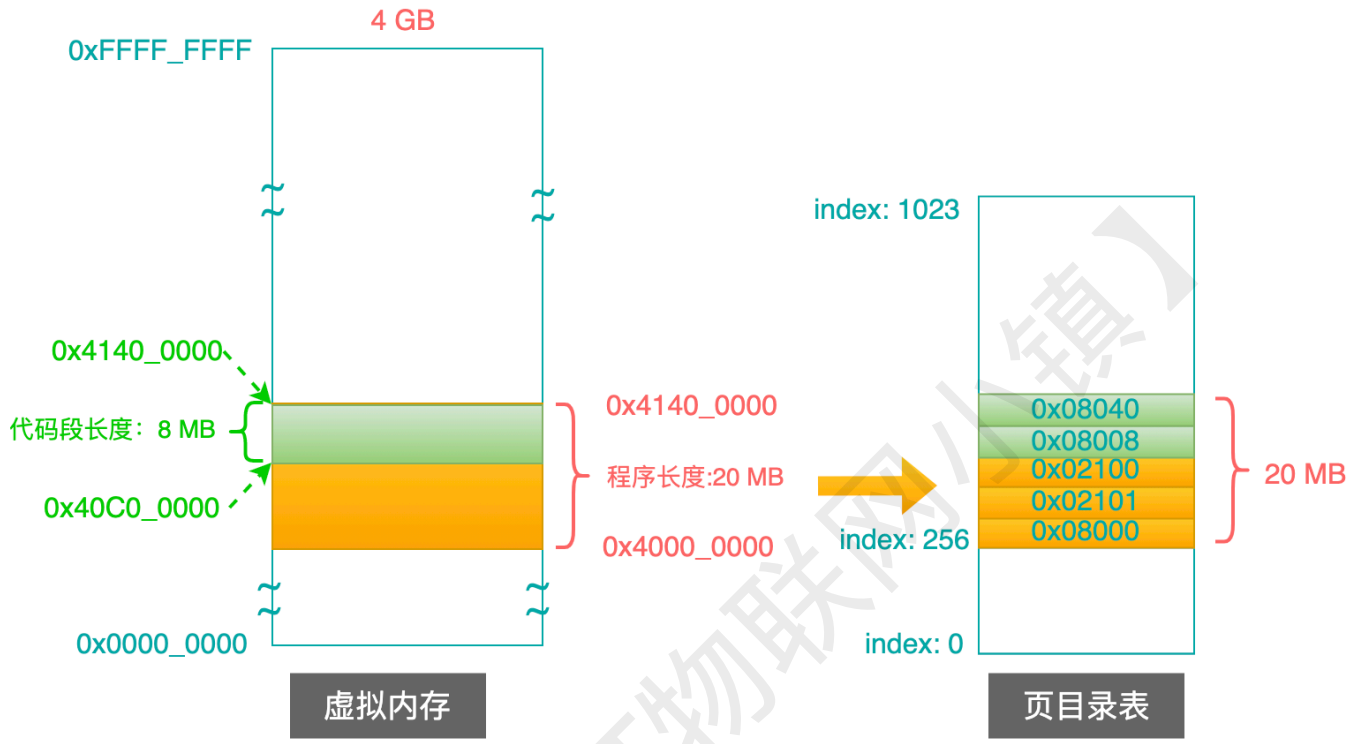
线性地址到物理地址的变换过程

如果理解了上一个主题的内容，那么部分应该就可以不用再看了，因为它俩是相反的过程，而且查找过程更简单一些。

仍然继续我们的假设：

> 1. 用户程序的长度是 20 MB，存放在虚拟内存 0x4000_0000 ~ 0x4140_0000 (线性地址)这段空间内；
> 2. 代码段的长度是 8 MB，从虚拟内存的 0x40C0_0000 处开始存放；

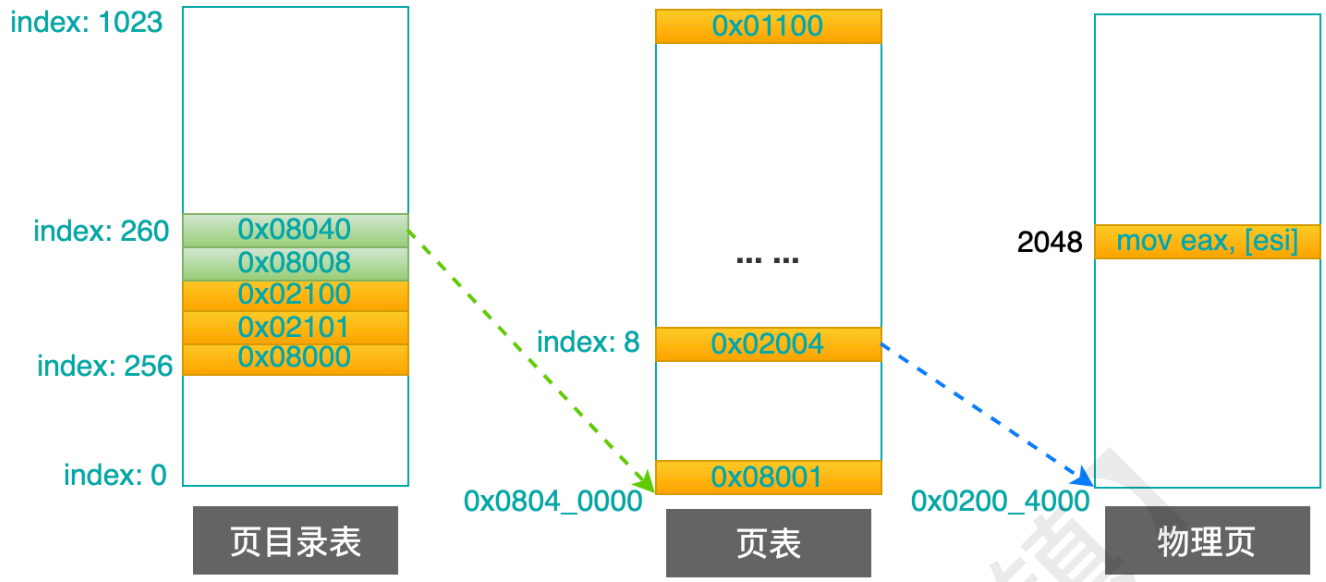
也就是如下图所示：



现在，用户程序的内容已经全部读取到内存中了，页目录、页表全部都安排妥当了。

在页目录表中，一共有 5 个表项，正好表示这20MB的地址空间，其中，8 MB 的代码空间对应于索引号为 259 和 260 这两个表项。

目标：处理器在执行代码时，遇到一个线性地址`0x4100_8800`，页处理单元需要把它转换得到物理地址。



`0x4100_8800 = 0100_0001_0000_0000___1000_1000_0000_0000`

首先，根据线性地址的前 10 位(`0100_0001_00`)，得到它在页目录中的索引值为`260`，这个个表项中记录的页表地址为 `0x08040`，因为页表地址的低12位一定是`0`，没有记录在表项中，因此这个页表的地址就是`0x0804_0000`。

> 页目录表的开始地址，肯定是从 CR3 寄存器获取的;

其次，根据线性地址的中间 10 位(`00_0000___1000`)，得到页表中的索引值为`8`，这个表项中记录的物理页地址为`0x02004`，补上低位的12个`0`，就得到物理页的开始地址是`0x0200_4000`。

最后，根据线性地址的最后 12 位(`1000_0000_0000`)，得到它在物理页的偏移量 `2048`，也就是说：从物理页的开始地址(`0x0200_4000`)，偏移`2048`个字节的地方，就是这个线性地址(`0x4100_8080`)对应的物理地址(`0x0200_4800`)。

大功告成！

------ End ------

关于虚拟地址到物理地址的转换、页目录和页表的查找过程，基本就讨论结束了。

不知道客官您是否已经酒足饭饱？如果还满意的话，请您鼓励一下，给我点个赞，转发给朋友圈中的技术小伙伴，非常感谢！

推荐阅读

【1】C语言指针-从底层原理到花式技巧，用图文和代码帮你讲解透彻
【2】一步步分析-如何用C实现面向对象编程
【3】原来gdb的底层调试原理这么简单
【4】内联汇编很可怕吗？看完这篇文章，终结它！

其他系列专辑：精选文章、C语言、Linux操作系统、应用程序设计、物联网