> 作 者：道哥，10+年嵌入式开发老兵，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(PDF 格式)。

【IOT物联网小镇】

目录

分段存储的坏处
物理内存的管理
映射表
一个线性地址的寻址过程

终于开始介绍分页机制了，作为一名 Linuxer，大名鼎鼎的分页机制必须要彻底搞懂！

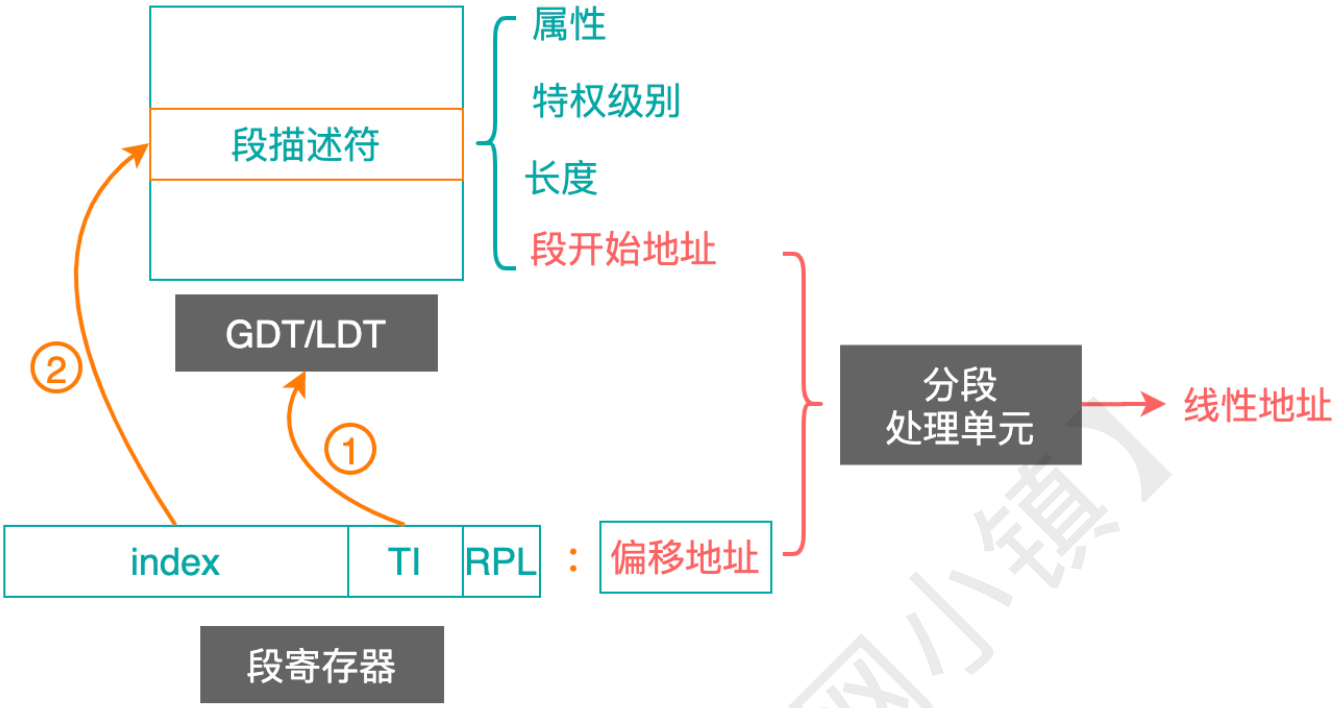
我就尽自己的最大努力，正确把我理解的分页机制，用图文形式彻底分解，希望对您有所帮助！

> 一共分 3 篇文章:
>
> > 这篇文章主要介绍单映射表;
> >
> > 下一篇介绍两级映射(页目录和页表);
> >
> > 最后一篇介绍对映射表自身的操作。

分段存储的坏处

在之前的文章中，我们多次描写了一个段描述符的结构，其中就包括段的开始地址、界限和各种段的属性。

经过分段处理单元的权限检查和计算，这个开始地址加上偏移量，就是一个线性地址，如下图所示:



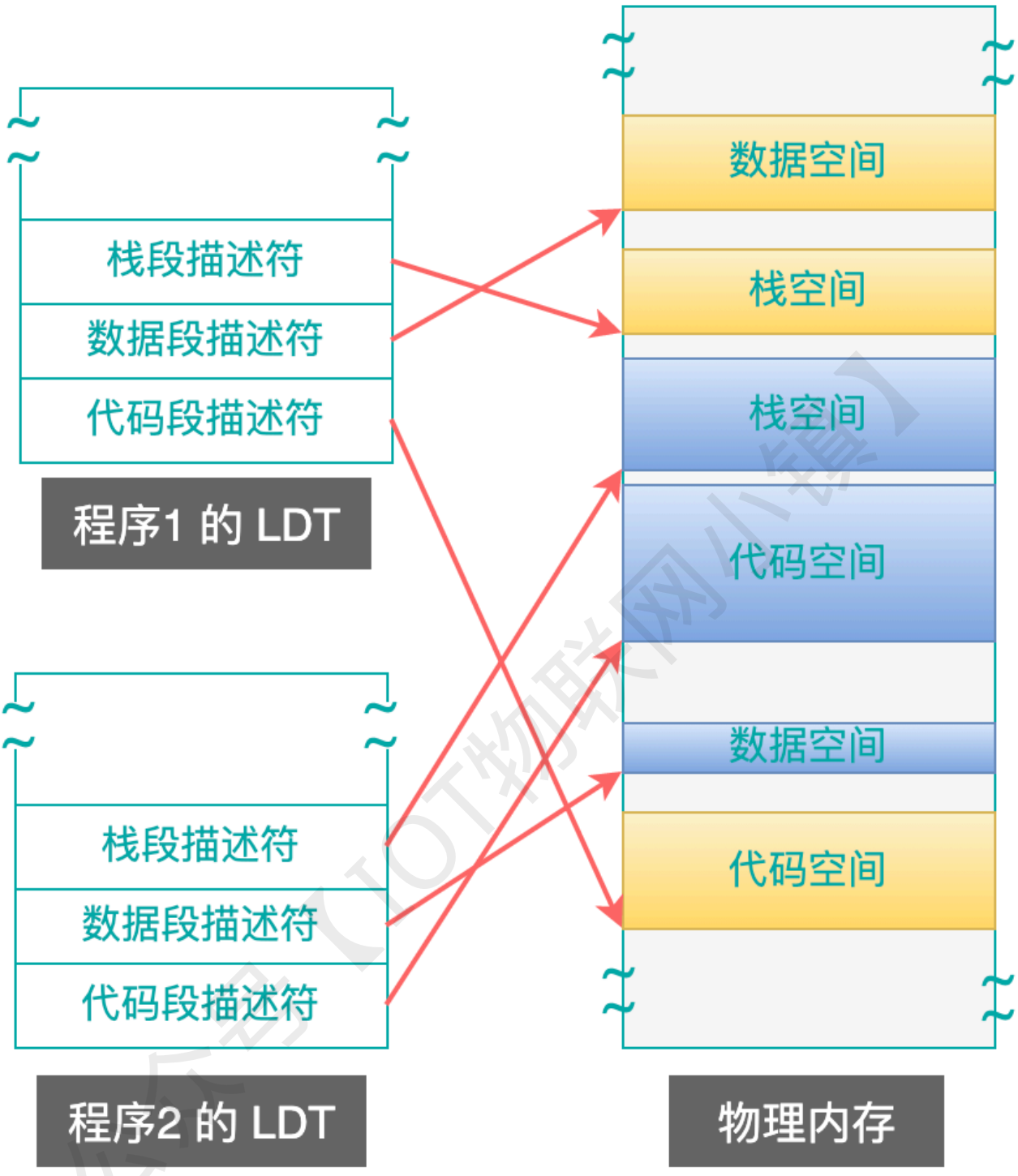
在 x86 系统中，分段机制是固有的，必须经过这个环节才能得到一个线性地址。

> 所以 Linux 系统中，为了“不使用”分段机制，但是又无法绕过，只好定义了“平坦”的分段模型。

在没有开启分页机制的情况下，分段单元输出的线性地址就等于物理地址。

这里就存在着一个重要的问题：从段的开始地址，一直到段空间的最后地址，这是一块连续的空间！

在这样的情况下，每一个用户程序中，包含的所有段，在物理内存上所对应的空间也必须是连续的，如下图：



因为每一个程序的代码、数据长度都是不确定、不一样的，按照这样的映射方式，物理内存将会被分割成各种离散的、大小不同的块。

经过一段运行时间之后，有些程序会退出，那么它们占据的物理内存空间可以被回收，但是，这些物理内存都是以很多碎片的形式存在。

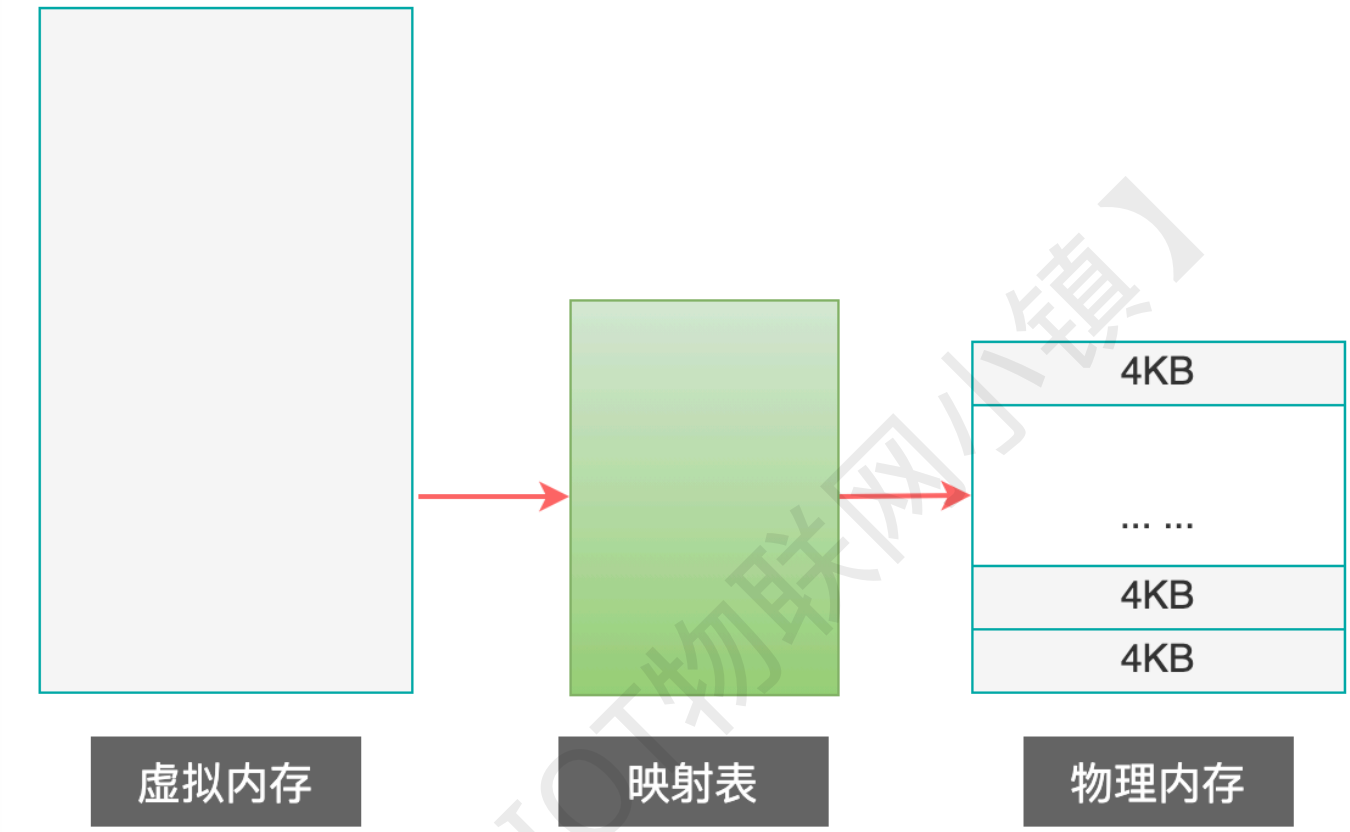
如果这个时候操作系统想分配一块稍微大一些的连续空间，虽然空闲的物理内存空间总数是足够的，但是不连续啊，这就给物理内存带来极大的浪费！

怎么办?

现在的需求是：操作系统提供给用户的段空间必须是连续的，但是物理内存最好不要连续。

软件领域有一句经典名言：没有什么是不能通过增加一个抽象层解决的！

在内存管理上，新加的这一层就是虚拟内存：把物理内存按照一个固定的单位(4 KB，称作一个物理页)进行分割，然后把连续的虚拟内存，映射到若干个不连续的物理内存页。



图中绿色的的映射表，就是用来把虚拟内存，映射到物理内存。

物理内存的管理

关于映射表的细节，下一个主题再聊，先来看一下操作系统对物理内存的状态管理。

在如今的一台 PC 机上，内存动辄就是是 8G/16G/32G 的配置，好像很充裕、随便用。

但是在 N 年以前，买一个 U 盘都是按照 MB 为单位的，更别说内存了。

因此在那个时代，面对 MB 级别的物理内存，操作系统还能够把它虚拟成 4GB 的内存空间给用户程序使用，也是挺厉害的！

言归正传，在这篇文章中，我们就奢侈一点，假设可用的物理内存有 1GB 的空间。

当系统上电之后，BIOS 会检查系统的各种硬件资源，并告诉操作系统，其中就包括这 1GB 的物理内存。

按照一个物理页的大小 4KB 进行划分，1 GB 的空间就是262144 (1GB / 4K)个物理页。

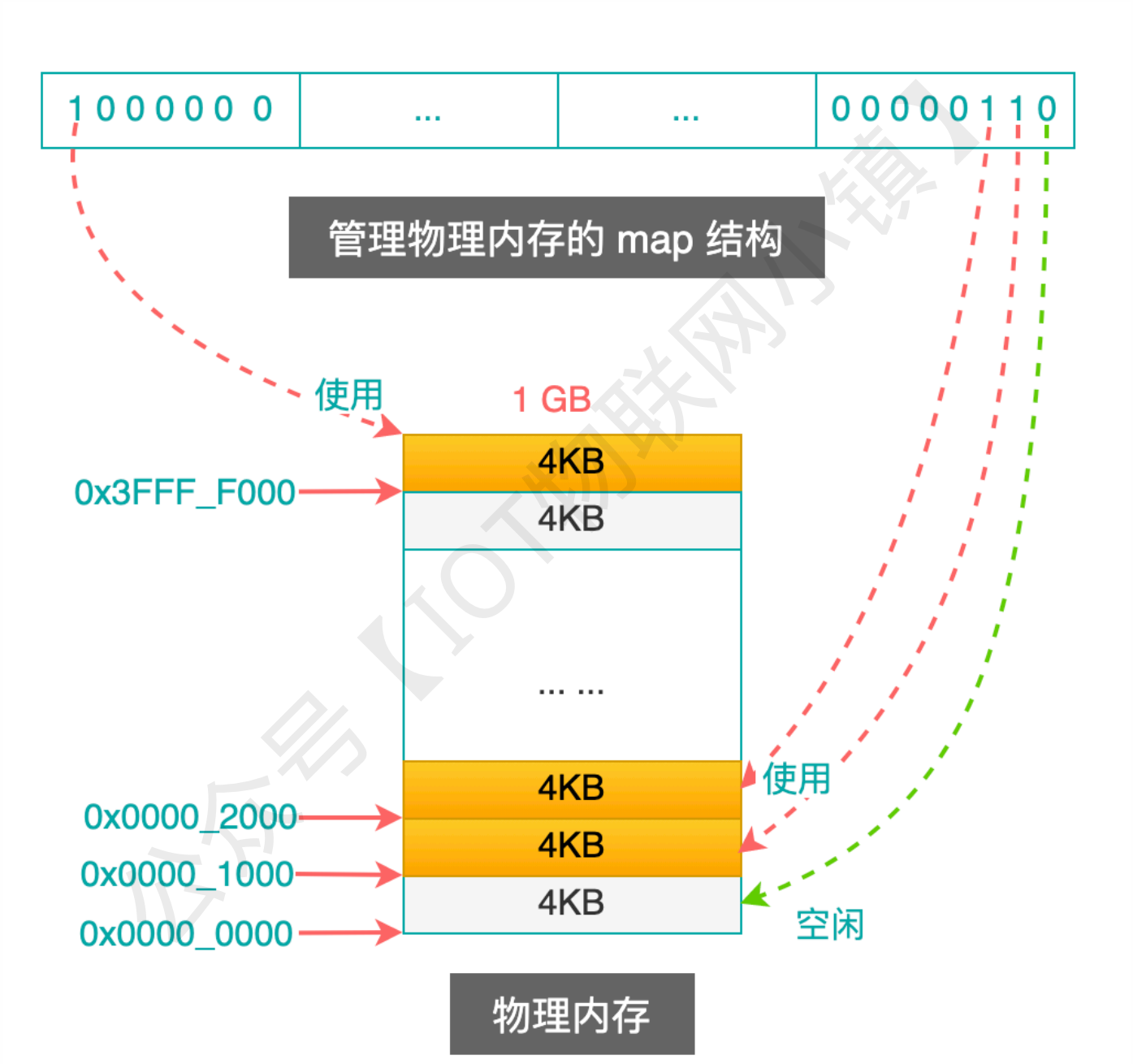
操作系统需要对这些页进行管理，也就是维护它们的状态：哪些页正在被使用，哪些页空闲。

最简单、直观的方法，就是用一块连续的内存空间来描述每一个物理页的状态，每一个bit位对应一个物理页：

> bit = 1: 表示该物理页被使用;
>
> bit = 0：表示该物理页空闲;

262144 个页需要262144个bit位，也就是32768个字节。

那么对于1 GB大小的物理内存来说，如下图所示：



利用map结构，操作系统就知道当前: 哪些物理页正在被使用，哪些物理页是空闲的。

> 1. 每一个物理页是 4KB，所以地址中最后 12 个 bit 都是 0;
> 2. map 结构本身也需要存储在物理内存中的，因此 32768 个字节，一共需要 8 个物理页来存储(32768 / 4 * 1024 = 8)。

映射表

在32位系统中，虚拟内存的最大空间是4GB，这是每一个用户程序都拥有的虚拟内存空间。

> 1. 实际上，操作系统都会把虚拟内存的高地址部分，用作操作系统，低地址部分留给用户程序使用；
> 2. Linux 系统中，高地址的 1GB 空间是操作系统使用；Windows 系统中，高地址的 2GB 的空间被操作系统使用，但是可以调整；

但是，实际的物理内存只有1GB(假设值)，那么操作系统就要使用自己的腾挪大法，让用户程序认为4GB的内存空间全部可用。

就好比变戏法一样：十个碗，九个盖，谁能玩的溜、不露馅，谁就是高手！

计算一下映射表本身所占据的空间大小：

映射表中的每一个表项，指向一个物理页的开始地址。

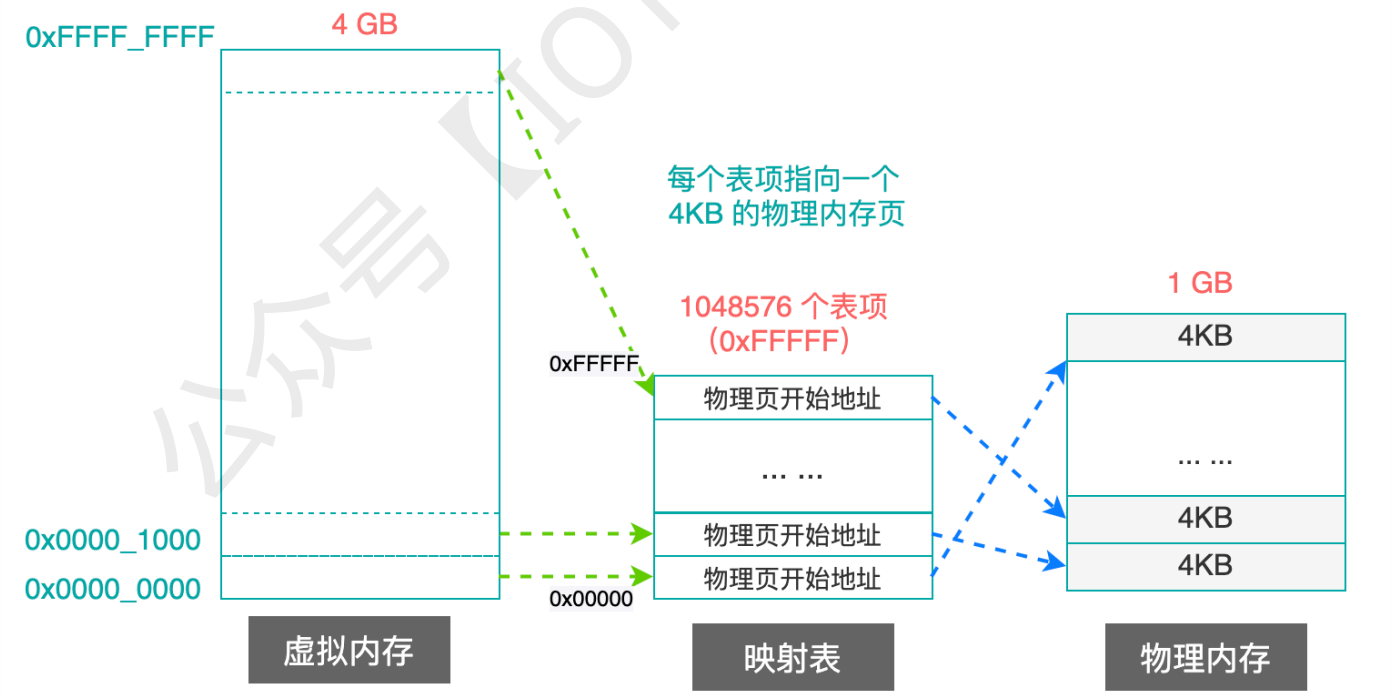
在32位系统中，地址的长度是4个字节，那么映射表中的每一个表项就是占用4个字节。

既然需要让4GB的虚拟内存全部可用，那么映射表中就需要能够表示这所有的4GB空间，那么就一共需要1048576(4GB / 4KB)个表项。

所以，映射表占据的总空间大小就是：1048576 * 4 = 4 MB 的大小。

也就是说，映射表自己本身，就要占用 1024 个物理页(4MB / 4KB)。

> 正是因为使用一个映射表，需要占用这么大的物理内存空间，所以才有后面的多级分页机制。



虚拟内存看上去被虚线“分割”成4KB的单元，其实并不是分割，虚拟内存仍然是连续的。

这个虚线的单元仅仅表示它与映射表中每一个表项的映射关系，并最终映射到相同大小的一个物理内存页上。

例如：

> 1. 虚拟内存的 0 ~ 4KB 空间，对应映射表第 0 个表项中，其中存储的物理地址是 0x3FFF_F000(最后一个物理页)；
> 2. 虚拟内存的 4KB ~ 8KB 空间，对应映射表第 1 个表项中，其中存储的物理地址是 0x0000_0000(第 0 个物理页)；
> 3. 虚拟内存的最后 4KB 空间，对应映射表最后一个表项中，其中存储的物理地址是 0x0000_1000(第 1 个物理页)；

也就是说：

虚拟内存与映射表之间，是平行的一一对应关系；

映射表中的物理地址，与物理内存之间，是随机的映射关系，哪里可用就指向哪里(物理页)。

以上就是用一个映射表，把物理内存以4KB为一个页进行分配，然后再与虚拟内存对应起来，包装成连续的虚拟内存给用户使用。

虽然最终使用的物理内存是离散的，但是与虚拟内存对应的线性地址是连续的。

处理器在访问数据、获取指令时，使用的都是线性地址，只要它是连续的就可以了，最终都能够通过映射表找到实际的物理地址。

为了有一个更加感性的认识，我们再来看一个稍微具象一点的实例。

一个线性地址的寻址过程

我们假设用户程序中有一个代码段，那么在这个程序的LDT（局部描述符表）中，段描述的结构如下：

假设条件如下：

> 1. 虚拟内存（32位系统）：4GB，实际的物理内存 1GB；
> 2. 代码段的开始地址位于 3 GB 的地方，也就是 0xC000_0000；
> 3. 代码段的长度是 1 MB；

我们的目标是：查找线性地址`0xC000_2020`所对应的物理地址。

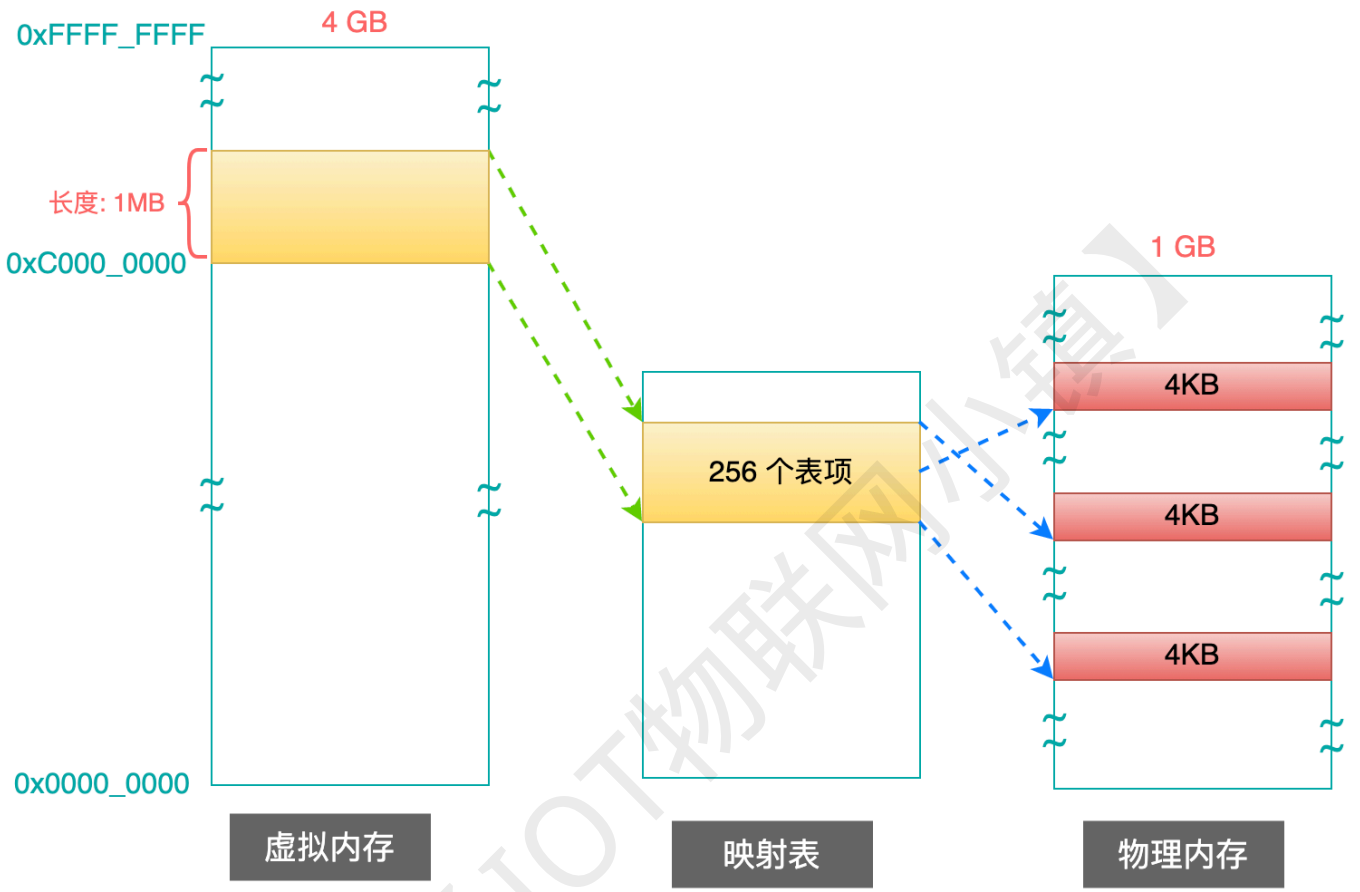
根据描述符的结构，其中的段基地址是`0xC000_0000`，界限是`0x00100`，段描述符中，其它的字段暂时不用关心。

> 界限一共有 20 位，假设粒度是 4KB，那么 1 MB 的长度除以 4KB，结果就是 0x00100。

代码段的开始地址(线性地址)0xC000_0000，位于虚拟内存靠近高端四分之一的位置，那么映射表中对应的表项，也是位于高端的四分之一的位置。

映射表中每一个表项指向一个4KB大小的是物理页，那么长度为1MB的代码段，就需要256个表项。

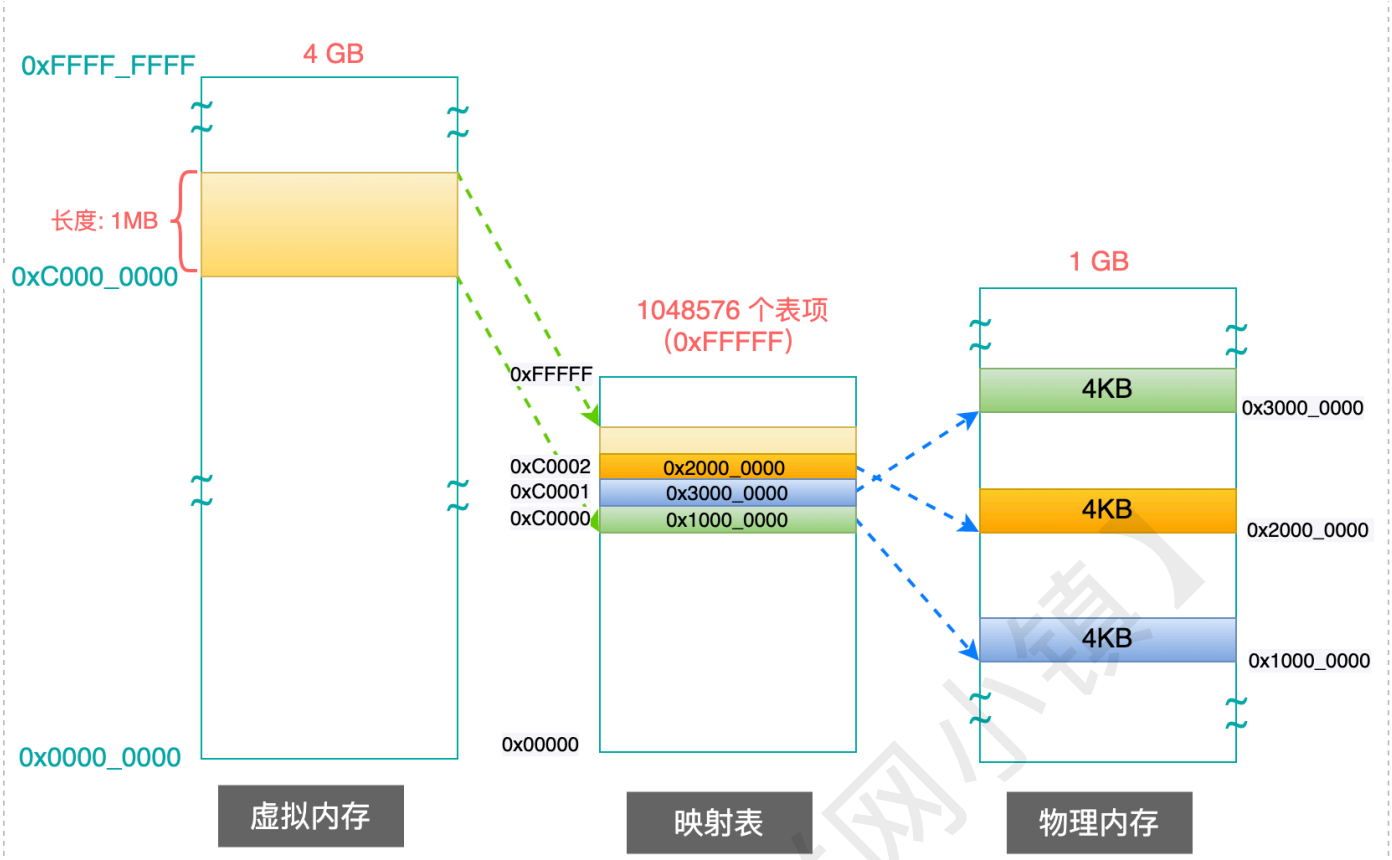
也就是说映射表中有 256个表项，指向256个物理页：



对于我们要查找的线性地址0xC000_2020，首先把它拆解成两部分：

> 高 20 位 0xC0002: 是映射表索引;
>
> 低 12 位 0x020: 是物理页内的偏移地址;

索引值0xC002，对应于下图中从3GB开始的第2个表项：



在上面这个示意图中，代码段的开始地址 0xC000_0000，对应于映射表中索引为0xC0000这个表项，这个表项中记录的物理内存页开始地址是 0x1000_0000（距离开始地址256 MB）。

代码段的长度是1 MB，一共需要256个表项，那么最后这个表项的索引就应该是0xC00FF。

那么对于我们要寻找的线性地址 0xC000_2020，对应的表项索引号是0xC0002，这个表项中记录的物理内存页的开始地址是 0x2000_0000（距离开始地址512 MB）。

找到了物理内存的起始地址，再加上偏移量0x020，那么最终的物理地址就是：0x2000_0020。

以上就是通过映射表，从线性地址到物理地址的页转换过程。

对于使用二级页表的转换机制来说，原理都是一样的。无非是把高20位的索引拆开(10 位 + 10 位)，使用两个表来转换，这个问题下一篇文章会详细聊。

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

本文描述了：通过一个映射表，把连续的虚拟内存，映射到离散的物理页，极大的利用了物理内存。

当操作系统需要分配一大块、连续的内存空间给用户程序时，映射表中的表项可以指向多个不连续的物理页，反正用户程序接触不到这一层(用户程序只与虚拟内存打交道)。

这样利用物理内存的效率就极大的提高了。

再加上换出和换入机制(把硬盘当做物理内存来用)，让用户程序以为有用不完的物理内存。

同时，我们也讨论了这个单一映射表的坏处，那就是映射表本身也占用了4MB的物理内存空间。

为了解决这个问题，伟大的先驱者们又引入了多级映射表，那就是页目录表和页表，我们下一篇文章再见！

如果这篇文章对您有小小的帮助，请转发给身边的小伙伴，让我们一起进步！

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

星标公众号，能更快找到我！

C/C++、物联网、嵌入式、Lua语言

Linux 操作系统、应用程序开发设计

扫码关注公众号

道哥 个人微信

喜欢请分享，满意点个赞，最后点在看。