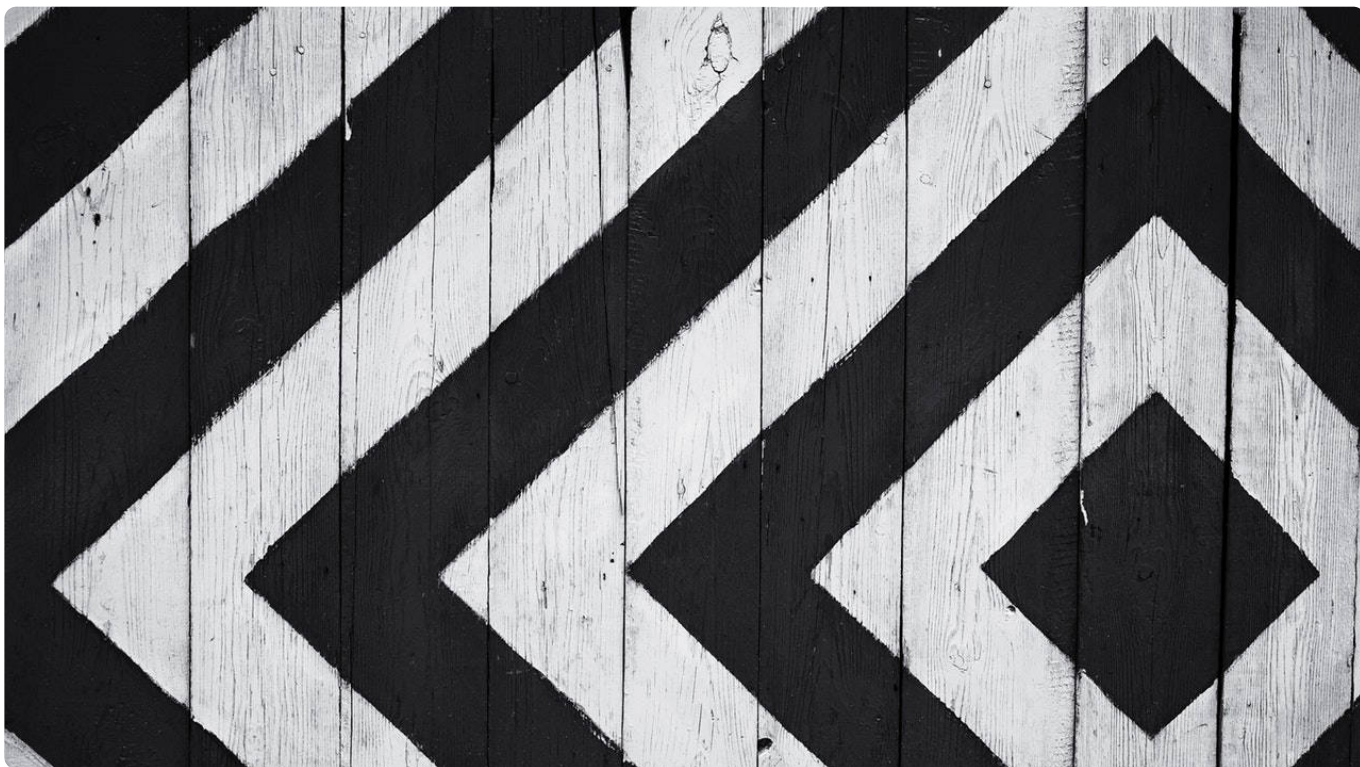


21 | 内存管理（下）：为客户保密，项目组独享会议室封闭开发

2019-05-15 刘超

趣谈Linux操作系统

[进入课程 >](#)



讲述：刘超

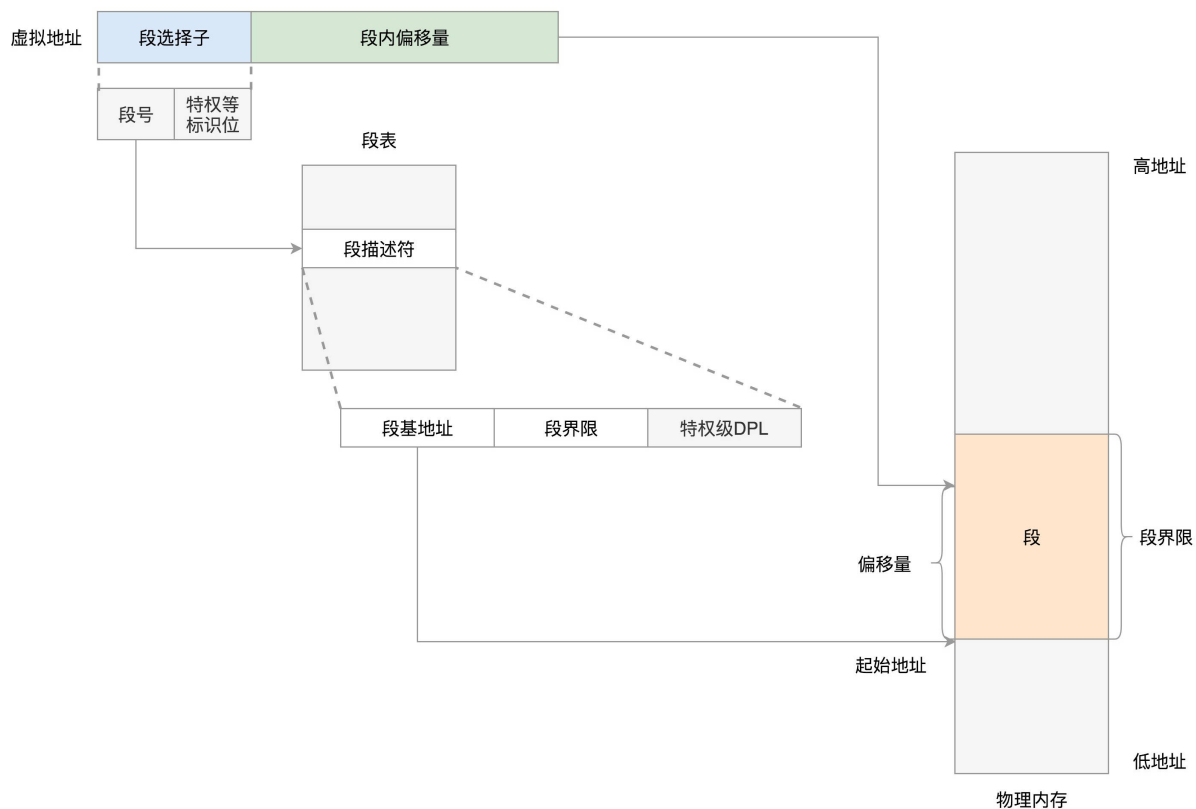
时长 10:52 大小 9.96M



上一节，我们讲了虚拟空间的布局。接下来，我们需要知道，如何将其映射成为物理地址呢？

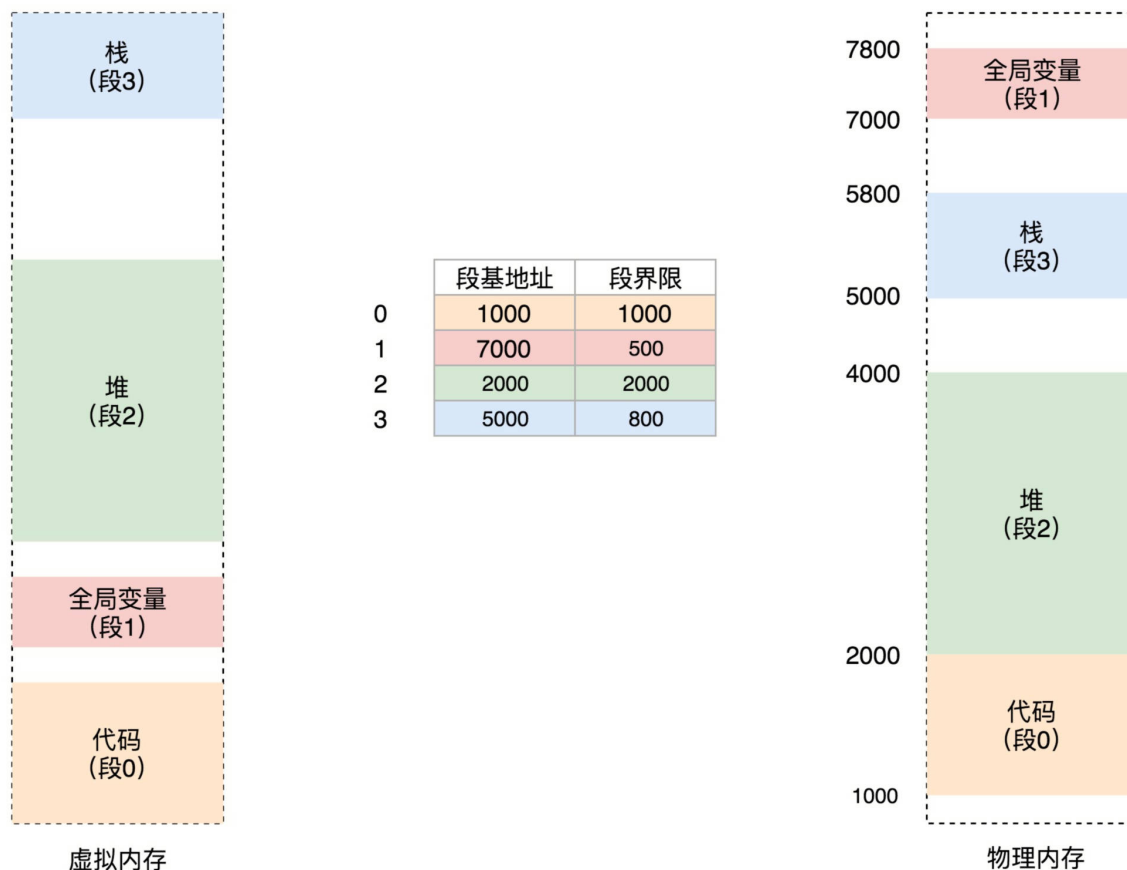
你可能已经想到了，咱们前面讲 x86 CPU 的时候，讲过分段机制，咱们规划虚拟空间的时候，也是将空间分成多个段进行保存。

那就直接用分段机制呗。我们来看看分段机制的原理。



分段机制下的虚拟地址由两部分组成，**段选择子**和**段内偏移量**。段选择子就保存在咱们前面讲过的段寄存器里面。段选择子里面最重要的是**段号**，用作段表的索引。段表里面保存的是这个段的**基地址**、**段的界限**和**特权等级**等。虚拟地址中的段内偏移量应该位于 0 和段界限之间。如果段内偏移量是合法的，就将段基地址加上段内偏移量得到物理内存地址。

例如，我们将上面的虚拟空间分成以下 4 个段，用 0 ~ 3 来编号。每个段在段表中有一个项，在物理空间中，段的排列如下图的右边所示。



如果要访问段 2 中偏移量 600 的虚拟地址，我们可以计算出物理地址为，段 2 基地址 2000 + 偏移量 600 = 2600。

多好的机制啊！我们来看看 Linux 是如何使用这个机制的。

在 Linux 里面，段表全称**段描述符表**（segment descriptors），放在**全局描述符表 GDT**（Global Descriptor Table）里面，会有下面的宏来初始化段描述符表里面的表项。

复制代码

```
1 #define GDT_ENTRY_INIT(flags, base, limit) { { { \
2     .a = ((limit) & 0xffff) | (((base) & 0xffff) << 16), \
3     .b = (((base) & 0xff0000) >> 16) | (((flags) & 0xf0ff) << 8) | \
4         ((limit) & 0xf0000) | ((base) & 0xff000000), \
5     } } }
```

一个段表项由段基地址 base、段界限 limit，还有一些标识符组成。

复制代码

```
1 DEFINE_PER_CPU_PAGE_ALIGNED(struct gdt_page, gdt_page) = { .gdt = {
```


```

2  #ifdef CONFIG_X86_64
3      [GDT_ENTRY_KERNEL32_CS]      = GDT_ENTRY_INIT(0xc09b, 0, 0xffffffff),
4      [GDT_ENTRY_KERNEL_CS]        = GDT_ENTRY_INIT(0xa09b, 0, 0xffffffff),
5      [GDT_ENTRY_KERNEL_DS]        = GDT_ENTRY_INIT(0xc093, 0, 0xffffffff),
6      [GDT_ENTRY_DEFAULT_USER32_CS] = GDT_ENTRY_INIT(0xc0fb, 0, 0xffffffff),
7      [GDT_ENTRY_DEFAULT_USER_DS]   = GDT_ENTRY_INIT(0xc0f3, 0, 0xffffffff),
8      [GDT_ENTRY_DEFAULT_USER_CS]   = GDT_ENTRY_INIT(0xa0fb, 0, 0xffffffff),
9  #else
10     [GDT_ENTRY_KERNEL_CS]          = GDT_ENTRY_INIT(0xc09a, 0, 0xffffffff),
11     [GDT_ENTRY_KERNEL_DS]          = GDT_ENTRY_INIT(0xc092, 0, 0xffffffff),
12     [GDT_ENTRY_DEFAULT_USER_CS]     = GDT_ENTRY_INIT(0xc0fa, 0, 0xffffffff),
13     [GDT_ENTRY_DEFAULT_USER_DS]     = GDT_ENTRY_INIT(0xc0f2, 0, 0xffffffff),
14     .....
15 #endif
16 } };
17 EXPORT_PER_CPU_SYMBOL_GPL(gdt_page);

```

这里面对于 64 位的和 32 位的，都定义了内核代码段、内核数据段、用户代码段和用户数据段。

另外，还会定义下面四个段选择子，指向上面的段描述符表项。这四个段选择子看着是不是有点眼熟？咱们讲内核初始化的时候，启动第一个用户态的进程，就是将这四个值赋值给段寄存器。

 复制代码

```

1  #define __KERNEL_CS      (GDT_ENTRY_KERNEL_CS*8)
2  #define __KERNEL_DS      (GDT_ENTRY_KERNEL_DS*8)
3  #define __USER_DS        (GDT_ENTRY_DEFAULT_USER_DS*8 + 3)
4  #define __USER_CS        (GDT_ENTRY_DEFAULT_USER_CS*8 + 3)

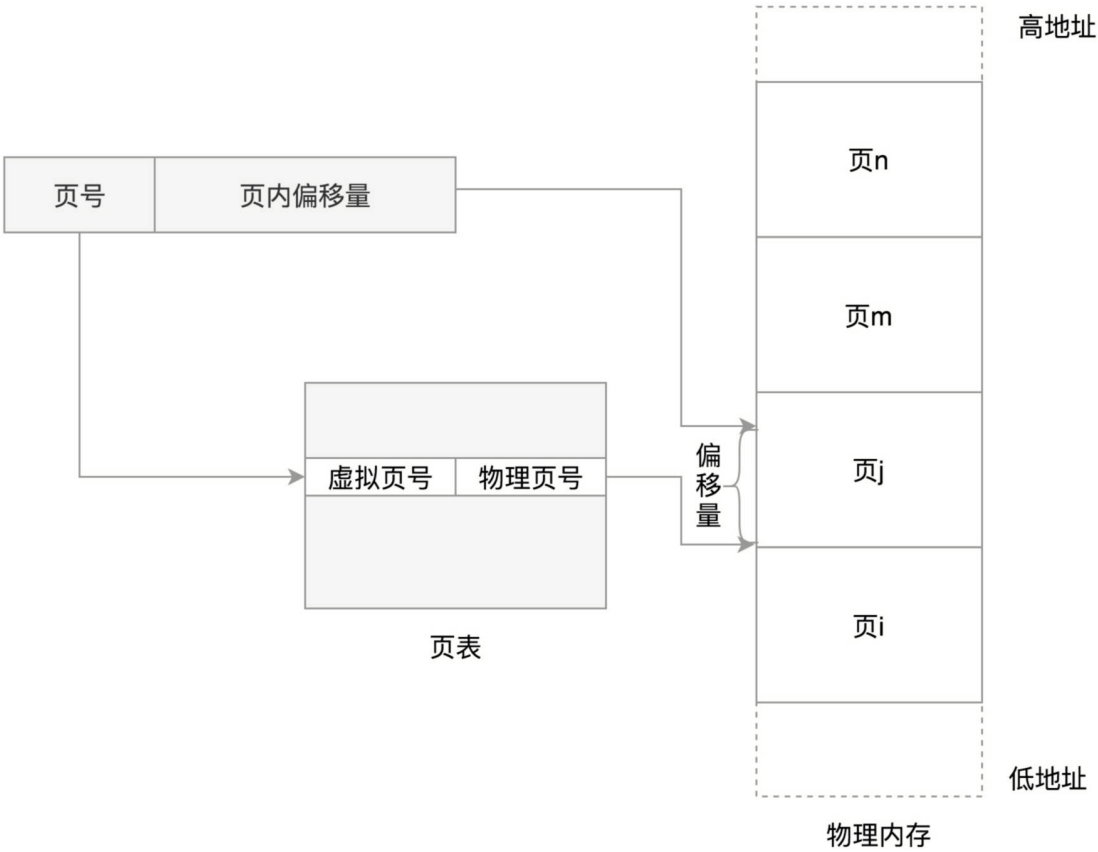
```

通过分析，我们发现，所有的段的起始地址都是一样的，都是 0。这算哪门子分段嘛！所以，在 Linux 操作系统中，并没有使用到全部的分段功能。那分段是不是完全没有用处呢？分段可以做权限审核，例如用户态 DPL 是 3，内核态 DPL 是 0。当用户态试图访问内核态的时候，会因为权限不足而报错。

其实 Linux 倾向于另外一种从虚拟地址到物理地址的转换方式，称为**分页**（Paging）。

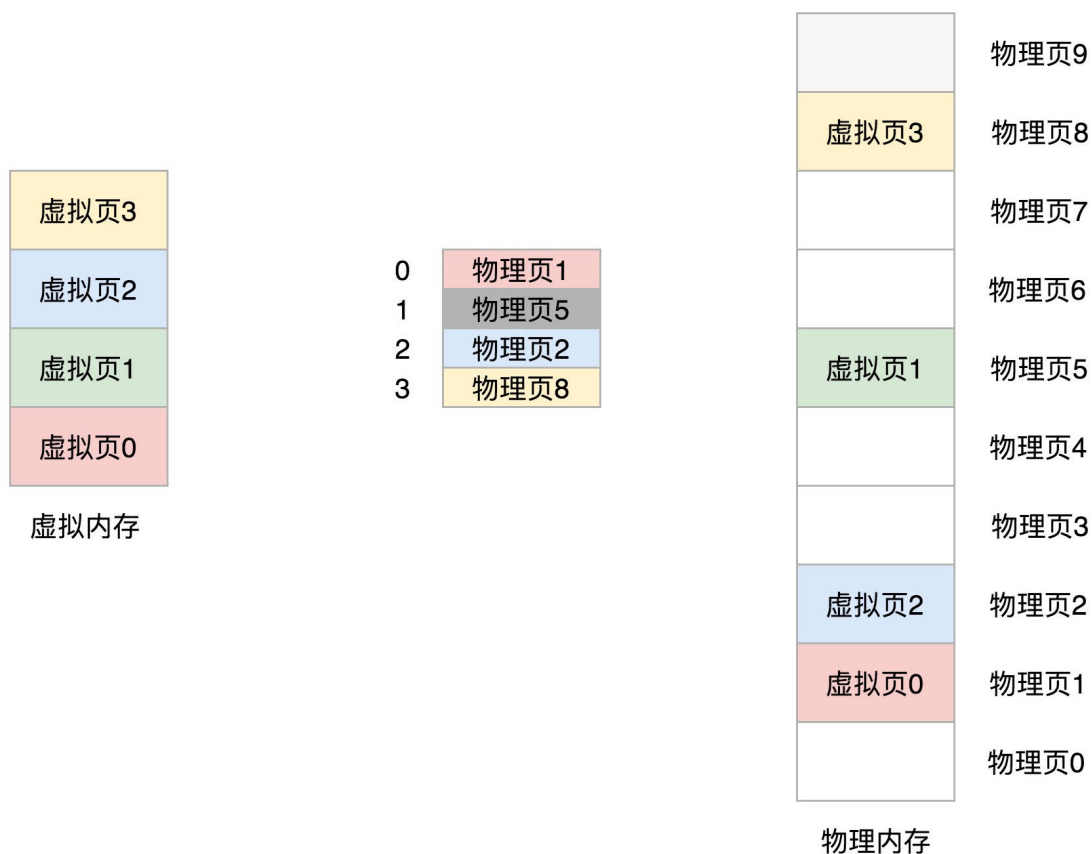
对于物理内存，操作系统把它分成一块一块大小相同的页，这样更方便管理，例如有的内存页面长时间不用了，可以暂时写到硬盘上，称为**换出**。一旦需要的时候，再加载进来，叫作**换入**。这样可以扩大可用物理内存的大小，提高物理内存的利用率。

这个换入和换出都是以页为单位的。页面的大小一般为 4KB。为了能够定位和访问每个页，需要有个页表，保存每个页的起始地址，再加上在页内的偏移量，组成线性地址，就能对于内存中的每个位置进行访问了。



虚拟地址分为两部分，**页号**和**页内偏移**。页号作为页表的索引，页表包含物理页每页所在物理内存的基地址。这个基地址与页内偏移的组合就形成了物理内存地址。

下面的图，举了一个简单的页表的例子，虚拟内存中的页通过页表映射为了物理内存中的页。



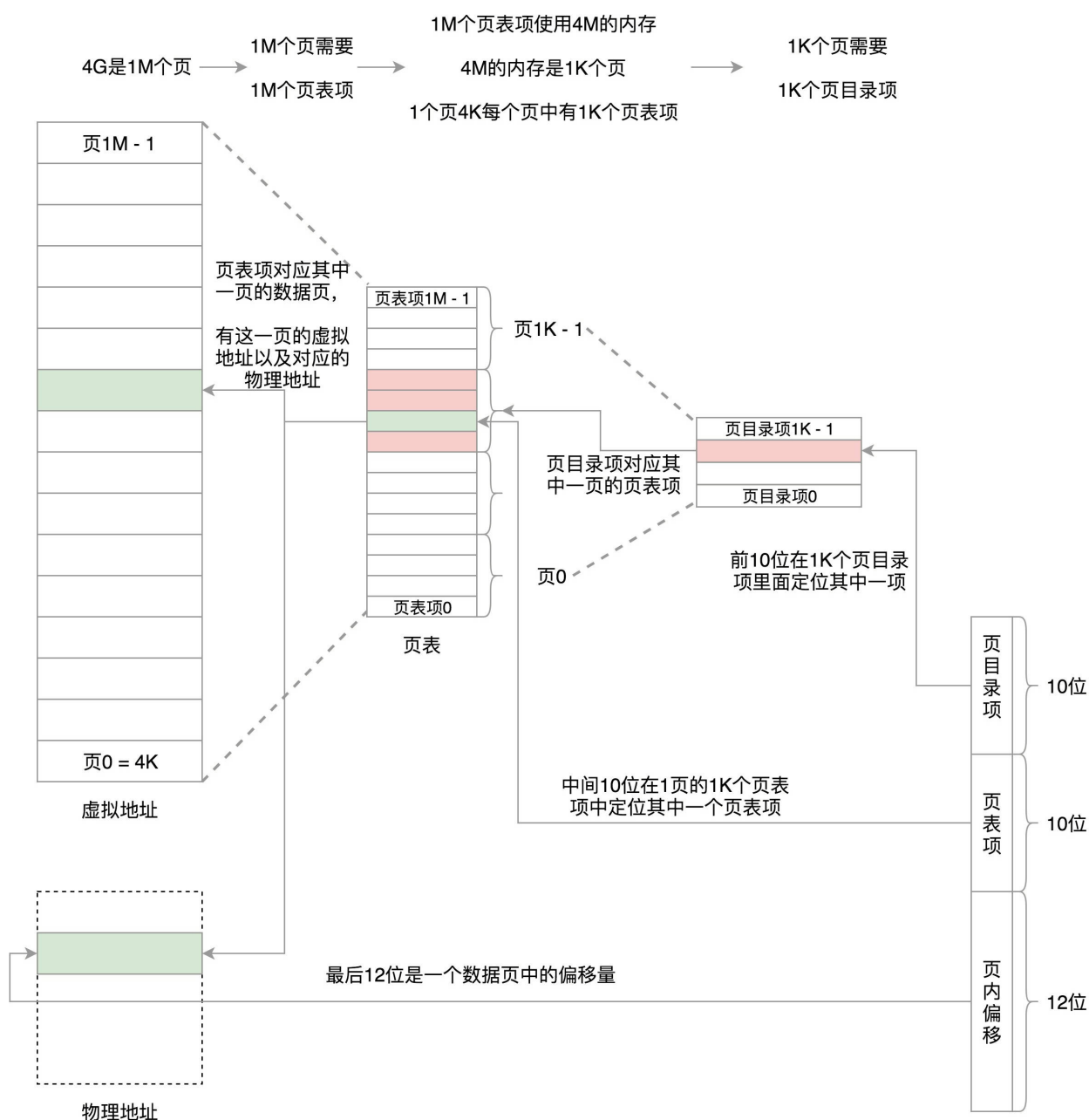
32 位环境下，虚拟地址空间共 4GB。如果分成 4KB 一个页，那就是 1M 个页。每个页表项需要 4 个字节来存储，那么整个 4GB 空间的映射就需要 4MB 的内存来存储映射表。如果每个进程都有自己的映射表，100 个进程就需要 400MB 的内存。对于内核来讲，有点大了。

页表中所有页表项必须提前建好，并且要求是连续的。如果不连续，就没有办法通过虚拟地址里面的页号找到对应的页表项了。

那怎么办呢？我们可以试着将页表再分页，4G 的空间需要 4M 的页表来存储映射。我们把这 4M 分成 1K (1024) 个 4K，每个 4K 又能放在一页里面，这样 1K 个 4K 就是 1K 个页，这 1K 个页也需要一个表进行管理，我们称为页目录表，这个页目录表里面有 1K 项，每项 4 个字节，页目录表大小也是 4K。

页目录有 1K 项，用 10 位就可以表示访问页目录的哪一项。这一项其实对应的是一整页的页表项，也即 4K 的页表项。每个页表项也是 4 个字节，因而一整页的页表项是 1K 个。再用 10 位就可以表示访问页表项的哪一项，页表项中的一项对应的就是一个页，是存放数据的页，这个页的大小是 4K，用 12 位可以定位这个页内的任何一个位置。

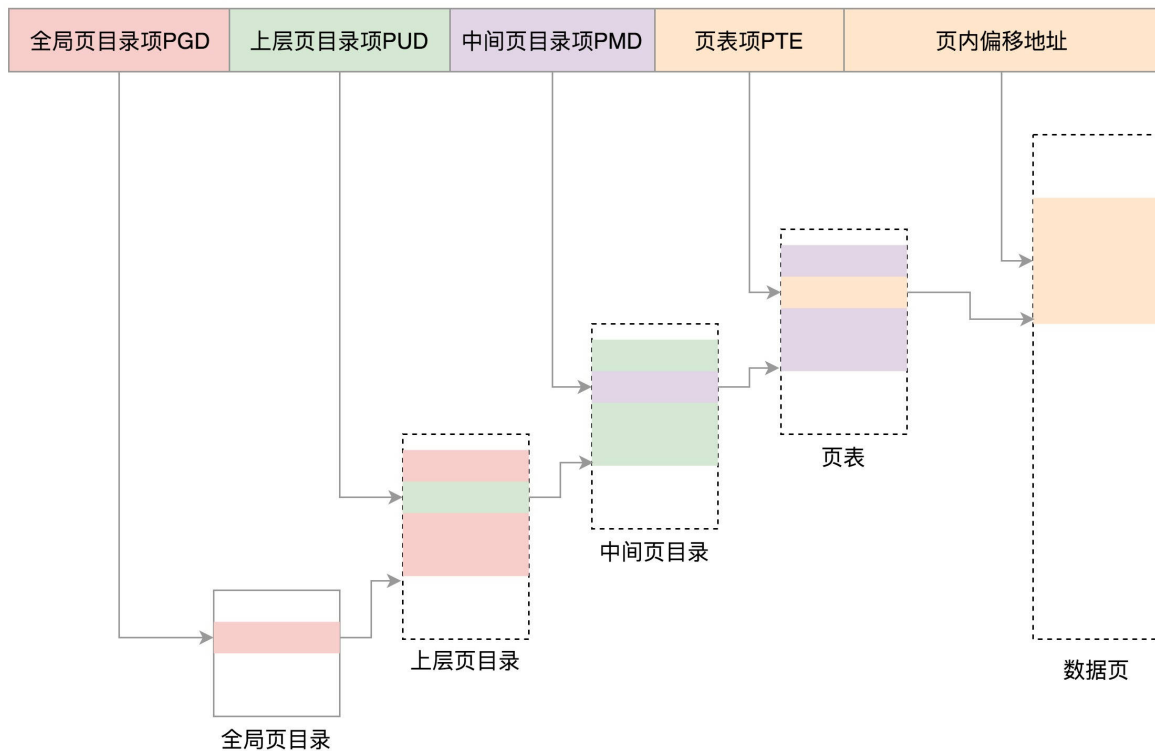
这样加起来正好 32 位，也就是用前 10 位定位到页目录表中的一项。将这一项对应的页表取出来共 1k 项，再用中间 10 位定位到页表中的一项，将这一项对应的存放数据的页取出来，再用最后 12 位定位到页中的具体位置访问数据。



你可能会问，如果这样的话，映射 4GB 地址空间就需要 4MB+4KB 的内存，这样不是更大了吗？当然如果页是满的，当时是更大了，但是，我们往往不会为一个进程分配那么多内存。

比如说，上面图中，我们假设只给这个进程分配了一个数据页。如果只使用页表，也需要完整的 1M 个页表项共 4M 的内存，但是如果使用了页目录，页目录需要 1K 个全部分配，占用内存 4K，但是里面只有一项使用了。到了页表项，只需要分配能够管理那个数据页的页表项页就可以了，也就是说，最多 4K，这样内存就节省多了。

当然对于 64 位的系统，两级肯定不够了，就变成了四级目录，分别是全局页目录项 PGD（Page Global Directory）、上层页目录项 PUD（Page Upper Directory）、中间页目录项 PMD（Page Middle Directory）和页表项 PTE（Page Table Entry）。



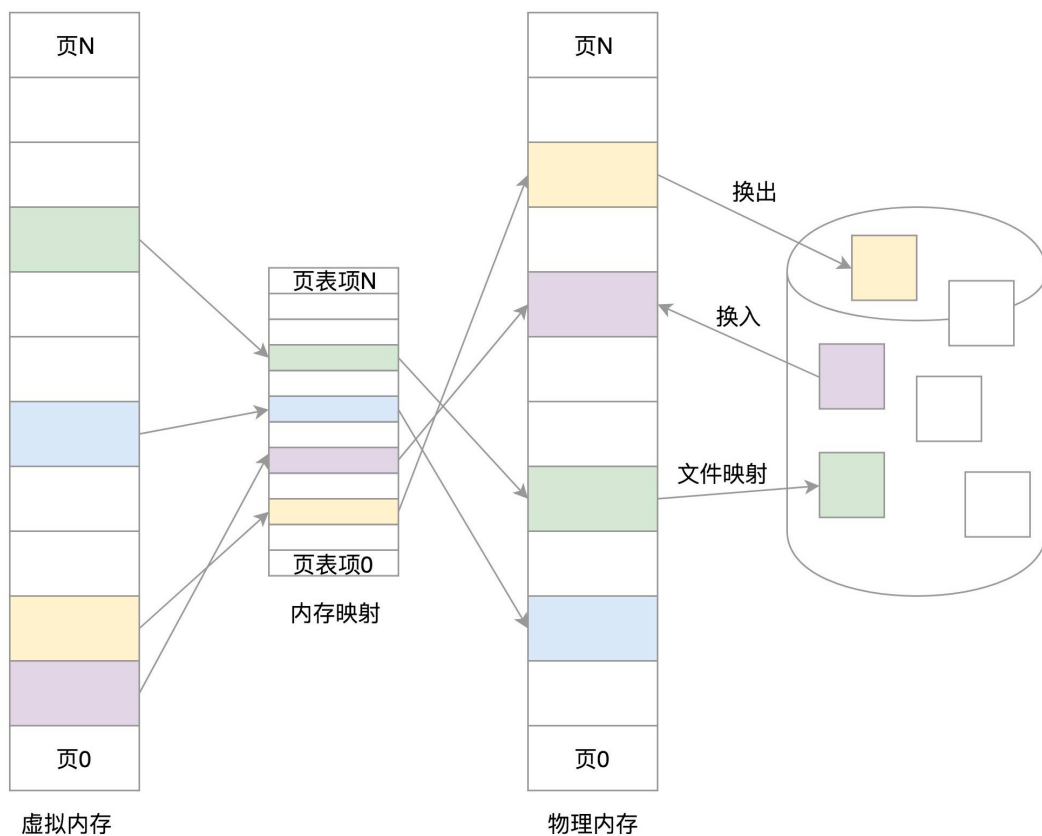
总结时刻

这一节我们讲了分段机制、分页机制以及从虚拟地址到物理地址的映射方式。总结一下这两节，我们可以把内存管理系统精细化为下面三件事情：

第一，虚拟内存空间的管理，将虚拟内存分成大小相等的页；

第二，物理内存的管理，将物理内存分成大小相等的页；

第三，内存映射，将虚拟内存也和物理内存也映射起来，并且在内存紧张的时候可以换出到硬盘中。



课堂练习

这一节我们说一个页的大小为 4K，有时候我们需要为应用配置大页（HugePage）。请你查一下大页的大小及配置方法，咱们后面会用到。

欢迎留言和我分享你的疑惑和见解，也欢迎你收藏本节内容，反复研读。你也可以把今天的内容分享给你的朋友，和他一起学习、进步。

趣谈 Linux 操作系统

像故事一样的操作系统入门课

刘超

网易杭州研究院

云计算技术部首席架构师



新版升级：点击「 请朋友读」，10位好友免费读，邀请订阅更有**现金**奖励。

© 版权归极客邦科技所有，未经许可不得传播售卖。页面已增加防盗追踪，如有侵权极客邦将依法追究其法律责任。

上一篇 20 | 内存管理（上）：为客户保密，规划进程内存空间布局

下一篇 22 | 进程空间管理：项目组还可以自行布置会议室

精选留言 (10)

写留言



why

2019-05-16

5

- 内存管理(下)
- 虚拟内存地址到物理内存地址的映射
- 分段
 - 虚拟地址 = 段选择子(段寄存器) + 段内偏移量
 - 段选择子 = 段号(段表索引) + 标识位...

展开



Leon

2019-05-15

3

分页机制本质上来说就是类似于linux文件系统的目录管理一样，页目录项和页表项相当于

根目录和上级目录，页内便宜量就是相对路径，绝对路径就是整个32位地址，分布式存储系统也是采用的类似的机制，先用元数据存储前面的路径，再用块内偏移定位到具体文件，感觉道理都差不多



Helios

2019-05-15

👍 1

请问老师为什么一个表项用4个字节去存储呢

展开 ▾

作者回复: 规定，可以去查一下表项的结构，太细节了，所以这里没有提



深海极光

2019-06-03

👍

老师请问下，不同进程的虚拟地址会出现映射到同一个物理地址即相同的page，如果会是把这个page换出还是怎么处理的，如果不回映射到同一个又是怎么保证的呢，谢谢

展开 ▾



小松松

2019-05-20

👍

请问一下，Linux在哪些管理上使用的分段，哪些情况使用的是分页呢？还是说现代操作系统都已经倾向于使用分页来管理了。



一笔一画

2019-05-17

👍

hugepage好像dpdk用过，这个应该对应的页目录项也会相应变少吧？另外，换页不是很惨吗？



Sharry

2019-05-15

👍

Nice, 终于看到最想了解的虚拟空间与物理页面的映射了

展开 ▾

Earl

2019-05-15



页的大小必须是2的n次方，而且与TLB的结构有关

展开 ▾



崔伟协

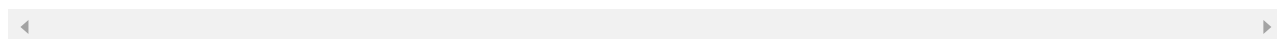
2019-05-15



分页，分段机制的优劣在于哪儿呢，为什么有分页分段

展开 ▾

作者回复: 都是硬件的机制，操作系统作为软件要用硬件机制。文章里面写了优劣势了。分段容易碎片，不容易换出。



有铭

2019-05-15



为什么页的默认大小是4KB，这是以什么理由定下来的，为什么不是2KB或者8KB呢