探索OS的内存管理原理

Linux阅码场 2022-11-23 00:00 发表于广东

以下文章来源于元闰子的邀请，作者元闰子

元闰子的邀请

阅读、写作、生活

前言

内存作为计算机系统的组成部分，跟开发人员的日常开发活动有着密切的联系，我们平时遇到的**Segment Fault**、**OutOfMemory**、**Memory Leak**、**GC**等都与它有关。本文所说的内存，指的是计算机系统中的主存（Main Memory），它位于存储金字塔中CPU缓存和磁盘之间，是程序运行不可或缺的一部分。

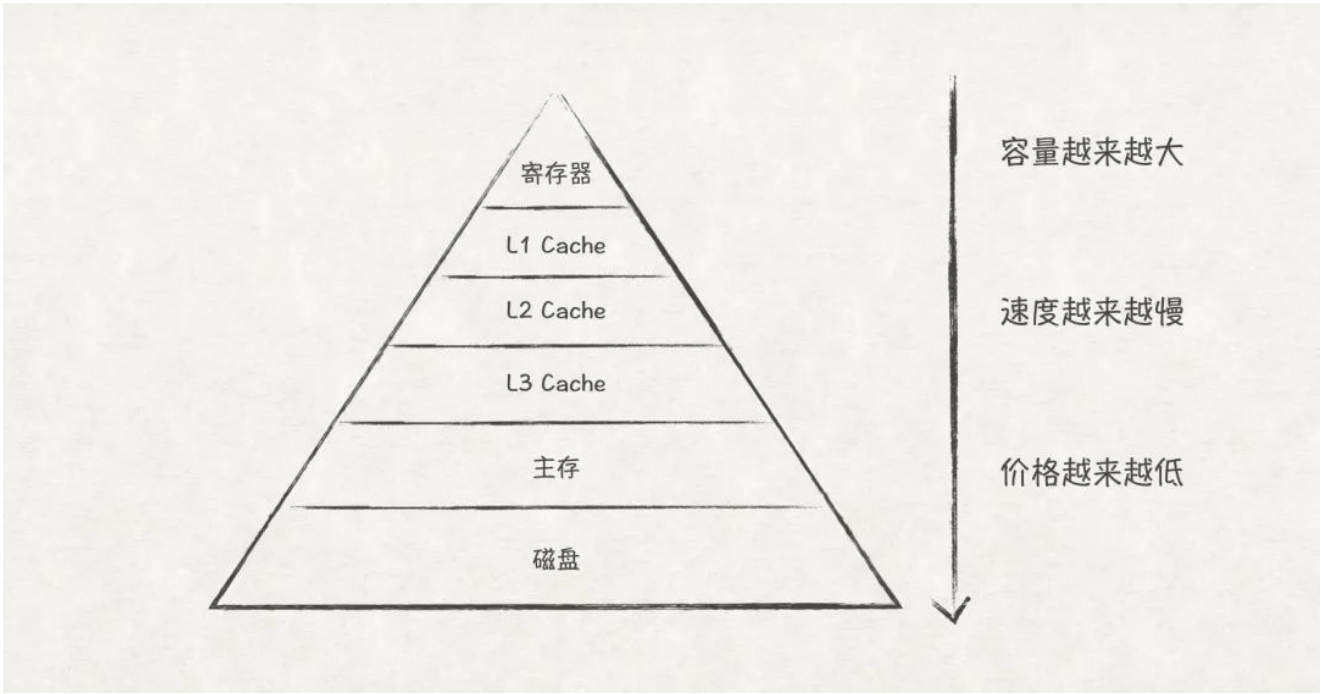


图1

在计算机系统中，**主存通常都是由操作系统（OS）来管理**，而内存管理的细则对开发者来说是无感的。对于一个普通的开发者，他只需懂得如何调用编程语言的接口来进行内存申请和释放，即可写出一个**可用**的应用程序。如果你使用的是带有垃圾回收机制的语

言，如Java和Go，甚至都不用主动释放内存。但如果你想写出**高效**应用程序，熟悉OS的内存管理原理就变得很有必要了。

下面，我们将从最简单的内存管理原理说起，带大家一起窥探OS的内存管理机制，由此熟悉底层的内存管理机制，写出高效的应用程序。

独占式内存管理

早期的单任务系统中，同一时刻只能有一个应用程序独享所有的内存（除开OS占用的内存），因此，内存管理可以很简单，只需在内存上划分两个区域：

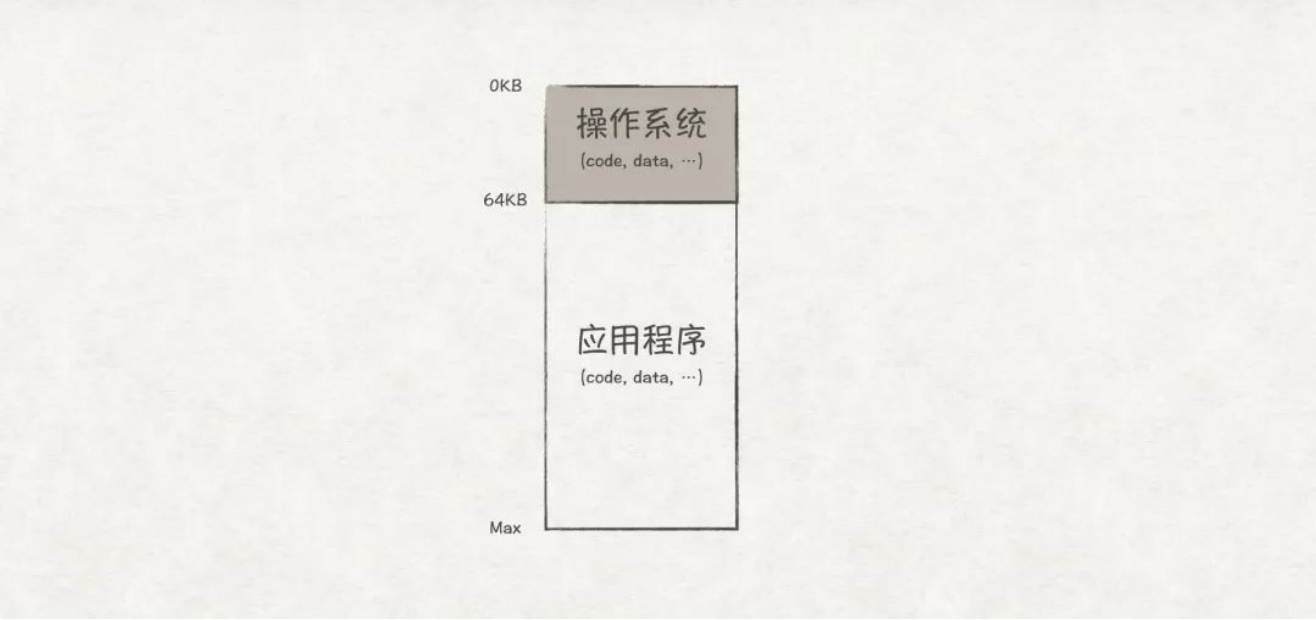
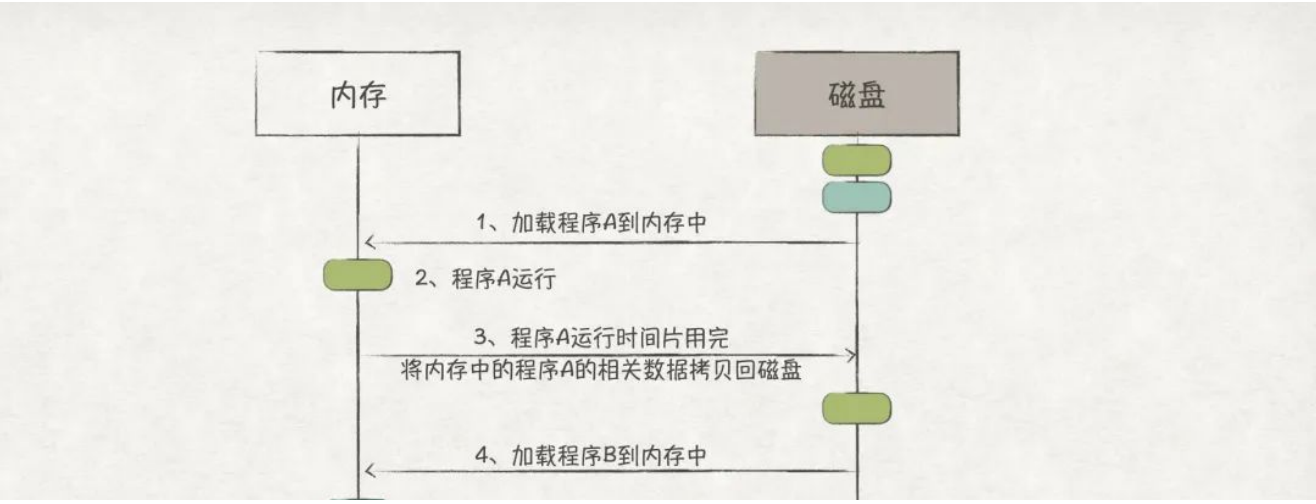


图2

在多任务系统中，计算机系统已经可以做到多个任务并发运行。如果还是按照独占式的管理方式，那么每次任务切换时，都会涉及多次内存和磁盘之间的**数据拷贝**，效率极其低下：



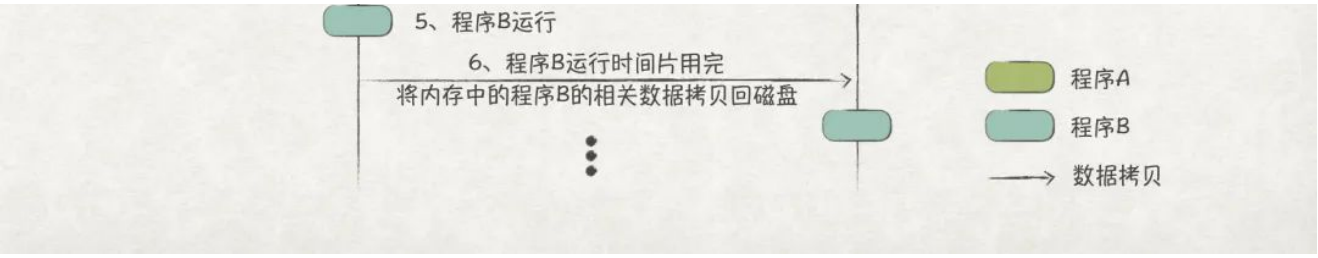


图3

最直观的解决方法就是让所有程序的数据都常驻在内存中（**假设内存足够大**），这样就能避免数据拷贝了：

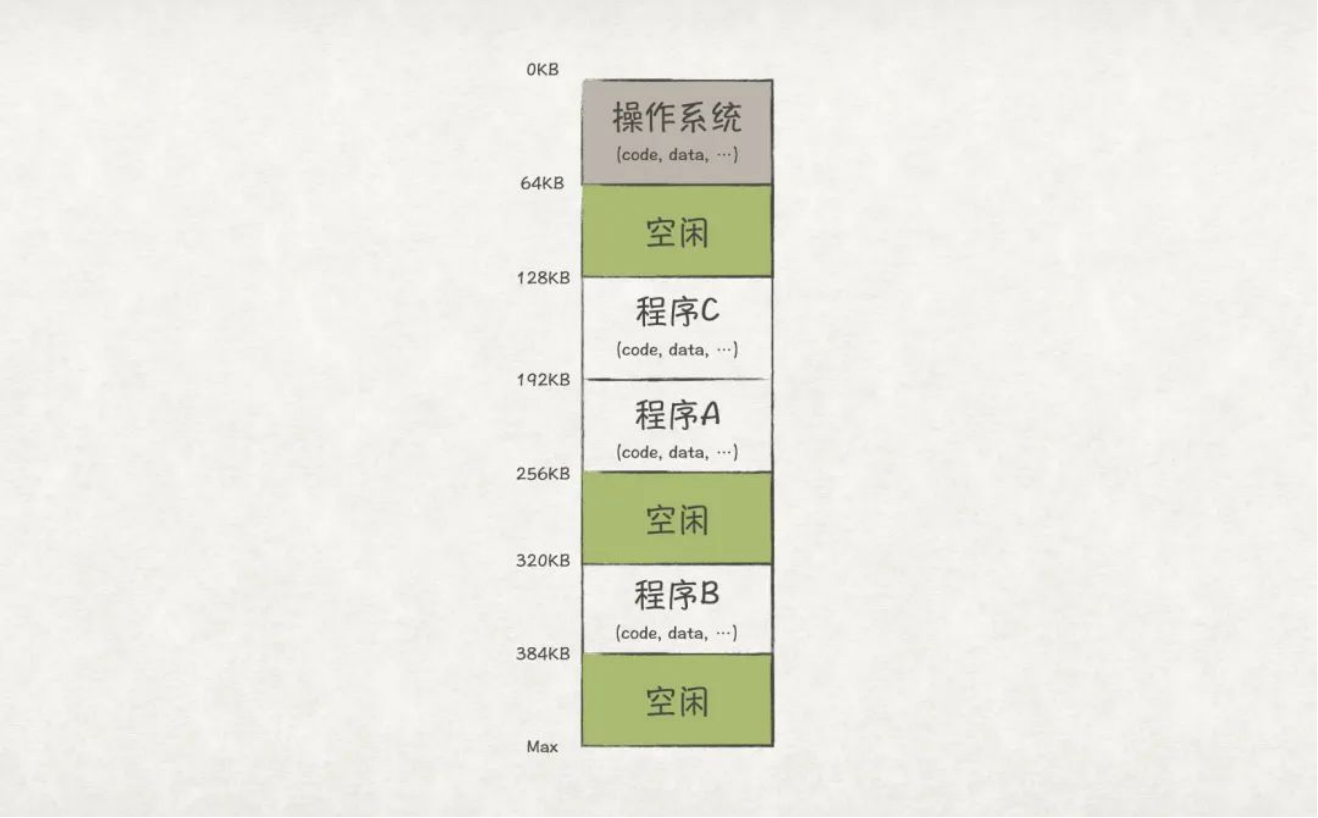


图4

但这样又会带来另一个问题，**程序之间的内存地址空间是没有隔离的**，也就是程序A可以修改程序B的内存数据。这样的一个重大的安全问题是用户所无法接受的，要解决该问题，就要借助**虚拟化**的力量了。

虚拟地址空间

为了实现程序间内存的隔离，OS对物理内存做了一层虚拟化。OS为每个程序都虚拟化出一段内存空间，这段虚拟内存空间会映射到一段物理内存上。但对程序自身而言，它只能看到自己的**虚拟地址空间**，也就有独占整个内存的错觉了。

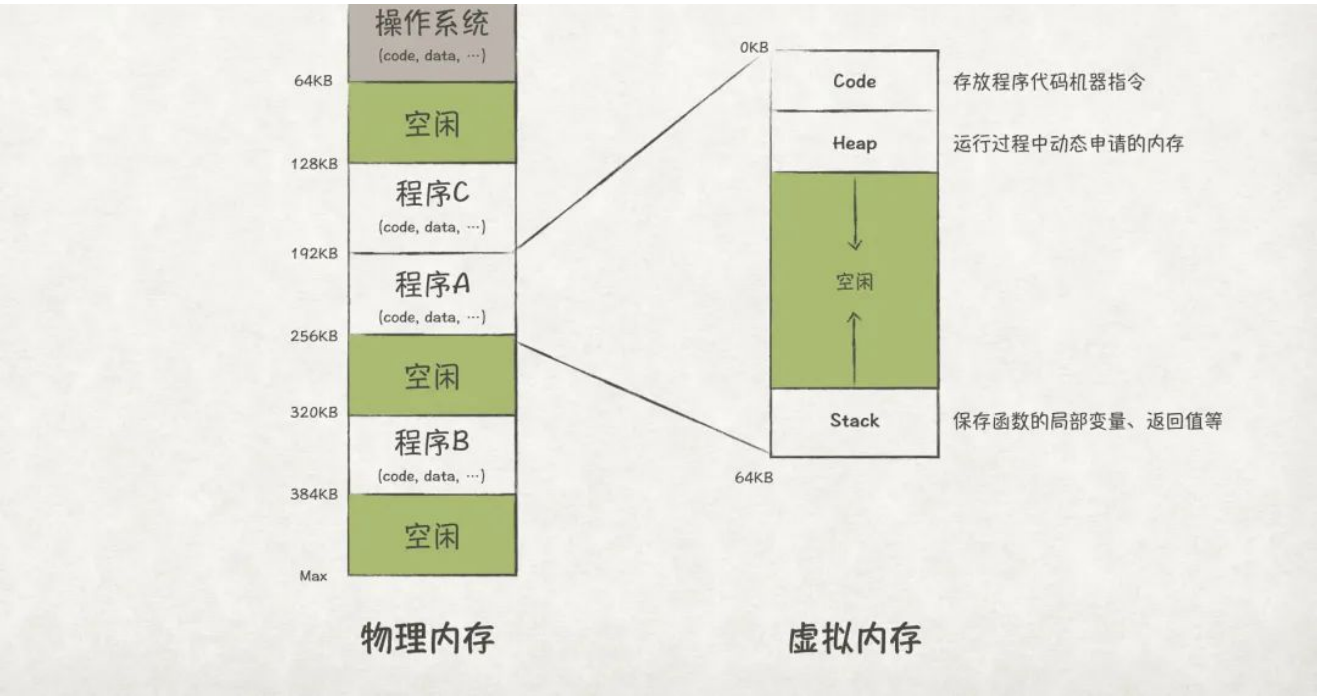


图5

上图中，虚拟内存空间分成了三个区域，其中**Code**区域存储的是程序代码的机器指令；**Heap**区域存储程序运行过程中动态申请的内存；**Stack**区域则是存储函数入参、局部变量、返回值等。Heap和Stack会在程序运行过程中不断增长，分别放置在虚拟内存空间的上方和下方，并往相反方向增长。

从虚拟地址空间到物理地址空间的映射，需要一个转换的过程，完成这个转换运算的部件就是**MMU**（memory management unit），也即**内存管理单元**，它位于CPU芯片之内。

要完成从虚拟地址到物理地址到转换，MMU需要**base**和**bound**两个寄存器。其中**base**寄存器用来存储程序在物理内存上的**基地址**，比如在*图5*中，程序A的基地址就是192KB；**bound**寄存器（有时候也叫**limit**寄存器）则保存虚拟地址空间的Size，主要用来避免越界访问，比如*图5*中程序A的size值为64K。那么，基于这种方式的地址转换公式是这样的：

> **物理地址 = 虚拟地址 + 基地址**

以*图5*中程序A的地址转换为例，当程序A尝试访问超过其bound范围的地址时，物理地址会转换失败：

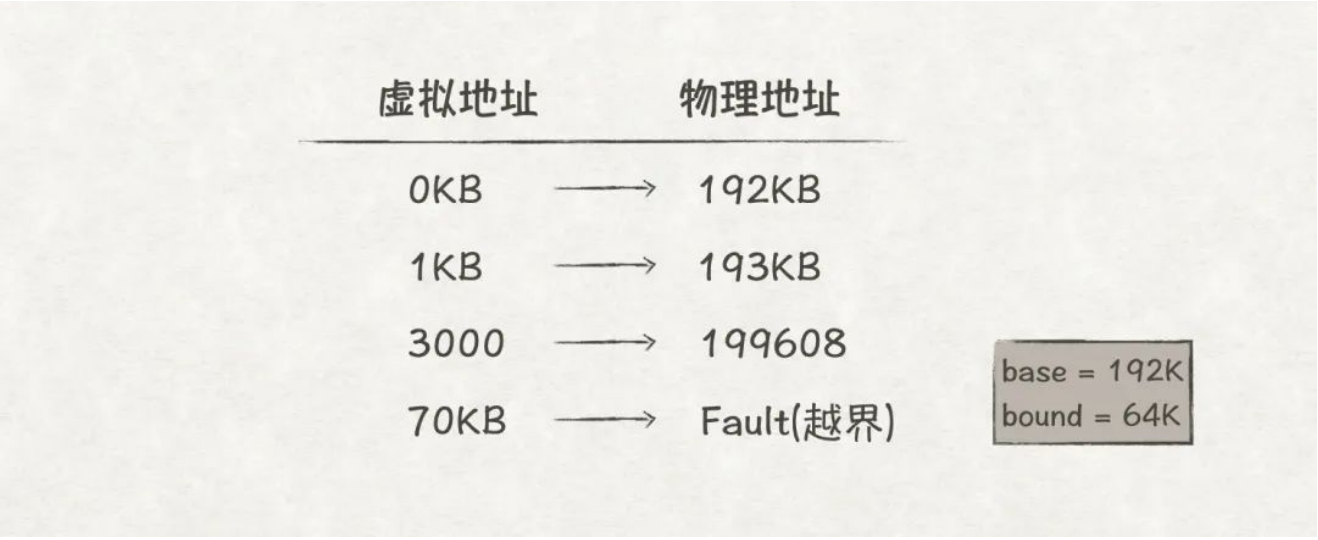


图6

现在，我们再次仔细看下程序A的物理内存分布，如下*图7*所示，中间有很大的一段空闲内存是“**已申请，未使用**”的空闲状态。这也意味着即使这部分是空闲的，也无法再次分配给其他程序使用，这是一种巨大的**空间浪费**！为了解决这个内存利用率低下的问题，我们熟悉的**段式内存管理**出现了。

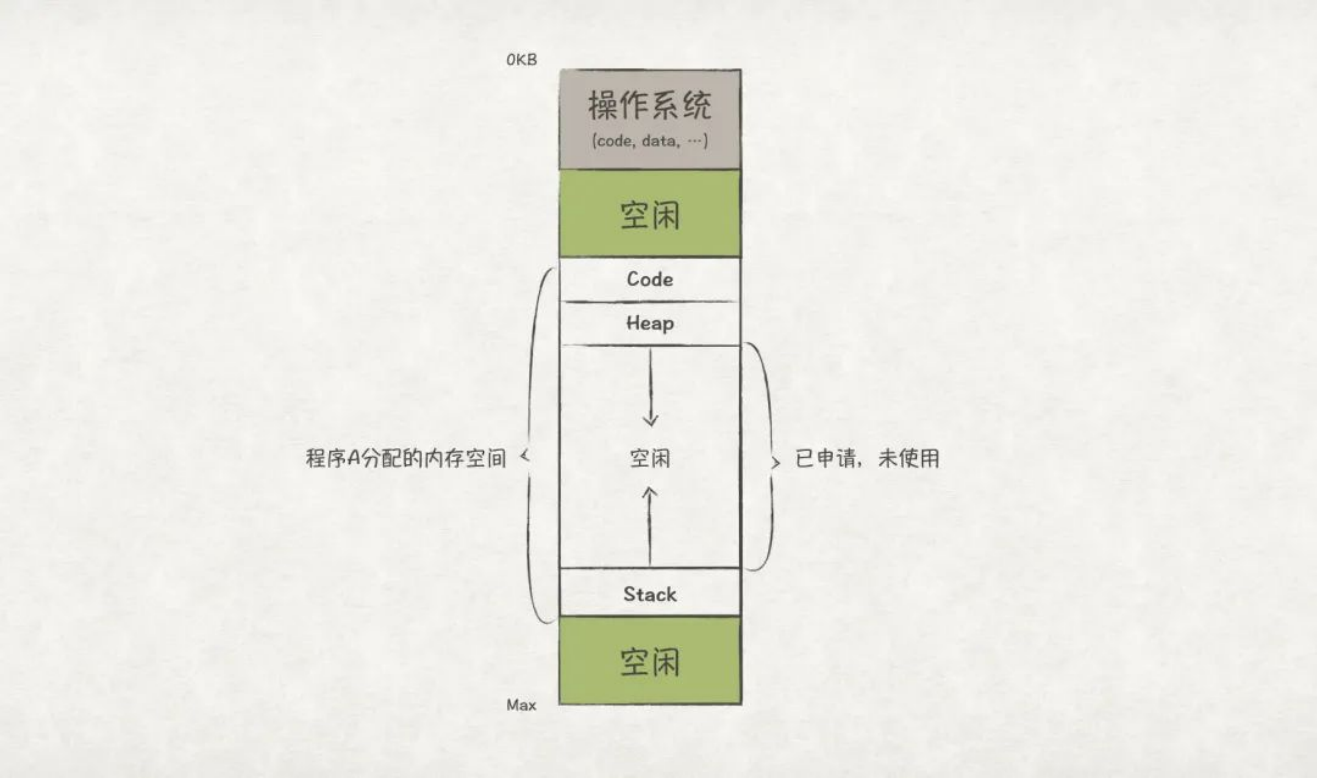


图7

段式内存管理

在上一节中，我们发现如果以程序为单位去做内存管理，会存在内存利用率不足的问题。为了解决该问题，段式内存管理被引入。**段（Segment）是逻辑上的概念，本质**

上是一块连续的、有一定大小限制的内存区域，前文中，我们一共提到过3个段：Code、Heap和Stack。

段式内存管理以段为单位进行管理，它允许OS将每个段灵活地放置在物理内存的空闲位置上，因此也避免了“已申请，未使用”的内存区域出现：

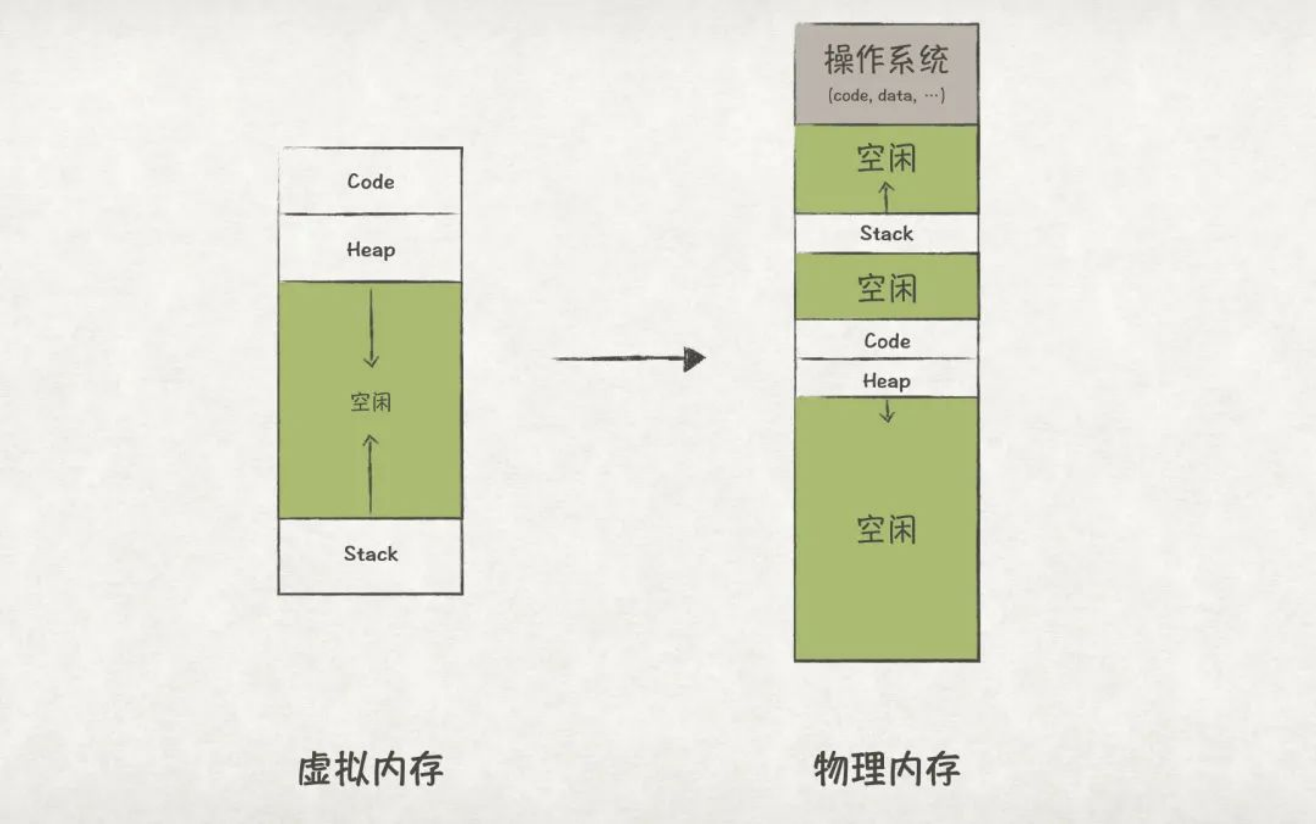


图8

地址转换

从上*图8*可知，段式内存管理中程序的物理内存空间可能不再连续了，因此为了实现从虚拟地址到物理地址到转换，MMU需要为每个段都提供一对base-bound寄存器，比如：

Segment	Base	Bound
Code	32KB	4KB
Heap	36KB	16KB
Stack	20KB	6KB

图9

给一个虚拟地址，MMU是如何知道该地址属于哪个段，从而根据对应的base-bound寄存器转换为对应的物理地址呢？

假设虚拟地址有16位，因为只有3个段，因此，我们可以使用虚拟地址的高2位来作为段标识，比如 00 表示Code段， 01 表示Heap段， 11 表示Stack段。这样MMU就能根据虚拟地址来找到对应段的base-bound寄存器了：

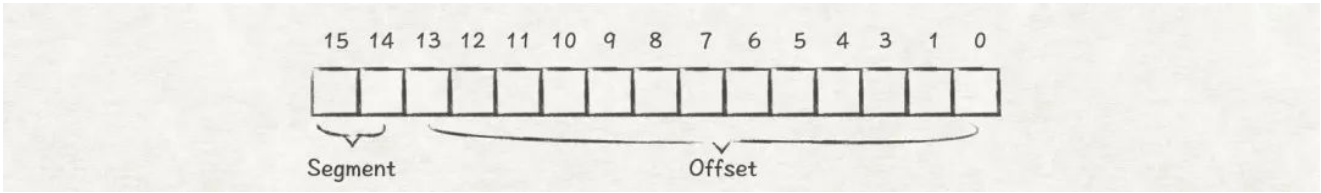


图10

但这样还不是能够顺利的将虚拟地址转换为物理地址，我们忽略了重要的一点：**Heap段和Stack段的增长方向是相反的**，这也意味着两者的地址转换方式是不一样的。因此，我们还必须在虚拟地址中多使用一位来标识段的增长方向，比如 0 表示向上（低地址方向）增长， 1 表示向下（高地址方向）增长：

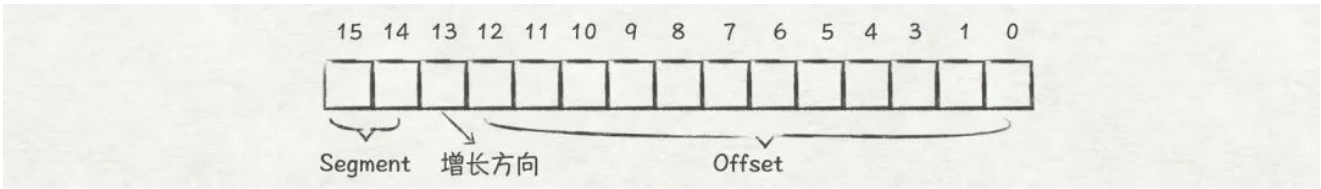
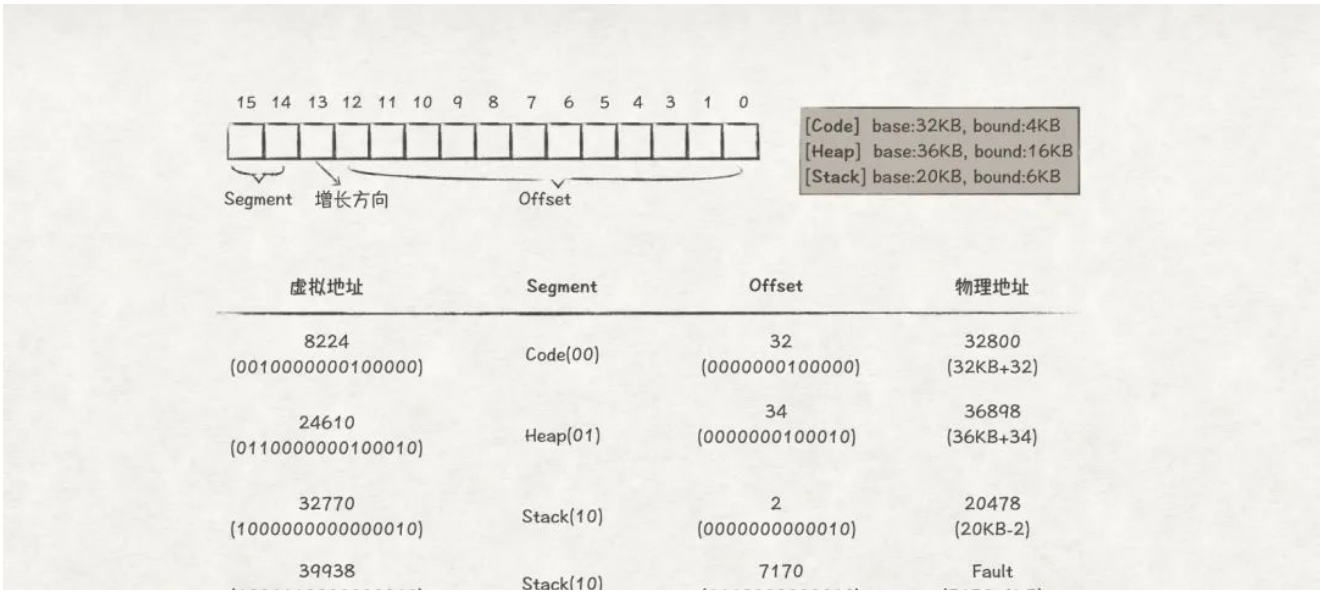


图11

下面，看一组段式内存管理地址转换的例子：



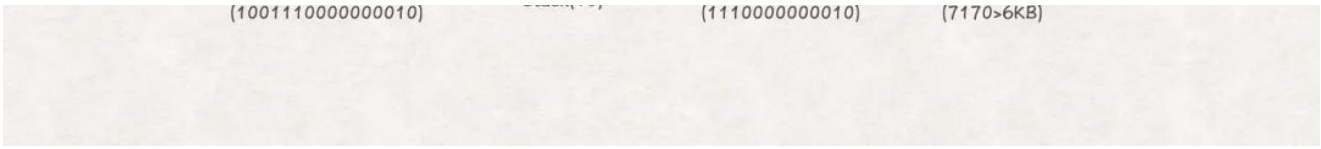


图12

那么，总结段式内存管理的地址转换算法如下：

```
// 获取当前虚拟地址属于哪个段
Segment = (VirtualAddress & SEG_MASK) >> SEG_SHIFT
// 得到段内偏移量
Offset  = VirtualAddress & OFFSET_MASK
// 获得内存增长的方向
GrowsDirection = VirtualAddress & GROWS_DIRECTION_MASK
// 有效性校验
if (Offset >= Bounds[Segment])
    RaiseException(PROTECTION_FAULT)
else
    if (GrowsDirection == 0) {
      PhysAddr = Base[Segment] + Offset
    } else {
      PhysAddr = Base[Segment] - Offset
    }
```

内存共享和保护

段式内存管理还可以很方便地**支持内存共享**，从而达到节省内存的目的。比如，如果存在多个程序都是同一个可执行文件运行起来的，那么这些程序是可以共享Code段的。为了实现这个功能，我们可以在虚拟地址上设置保护位，当保护位为只读时，表示该段可以共享。另外，如果程序修改了只读的段，则转换地址失败，因此也可以达到**内存保护**的目的。

Segment	Base	Bound	Grows Direction	Protection
Code	32KB	4KB	0	Read-Execute
Heap	36KB	16KB	0	Read-Write
Stack	20KB	6KB	1	Read-Write

图13

内存碎片

段式内存管理的最明显的缺点就是**容易产生内存碎片**，这是因为在系统上运行的程序的各个段的大小往往都不是固定的，而且段的分布也不是连续的。当系统的内存碎片很多时，内存的利用率也会急剧下降，对外表现就是**虽然系统看起来还有很多内存，却无法再运行起一个程序**。

解决内存碎片的方法之一是**定时进行碎片整理**：

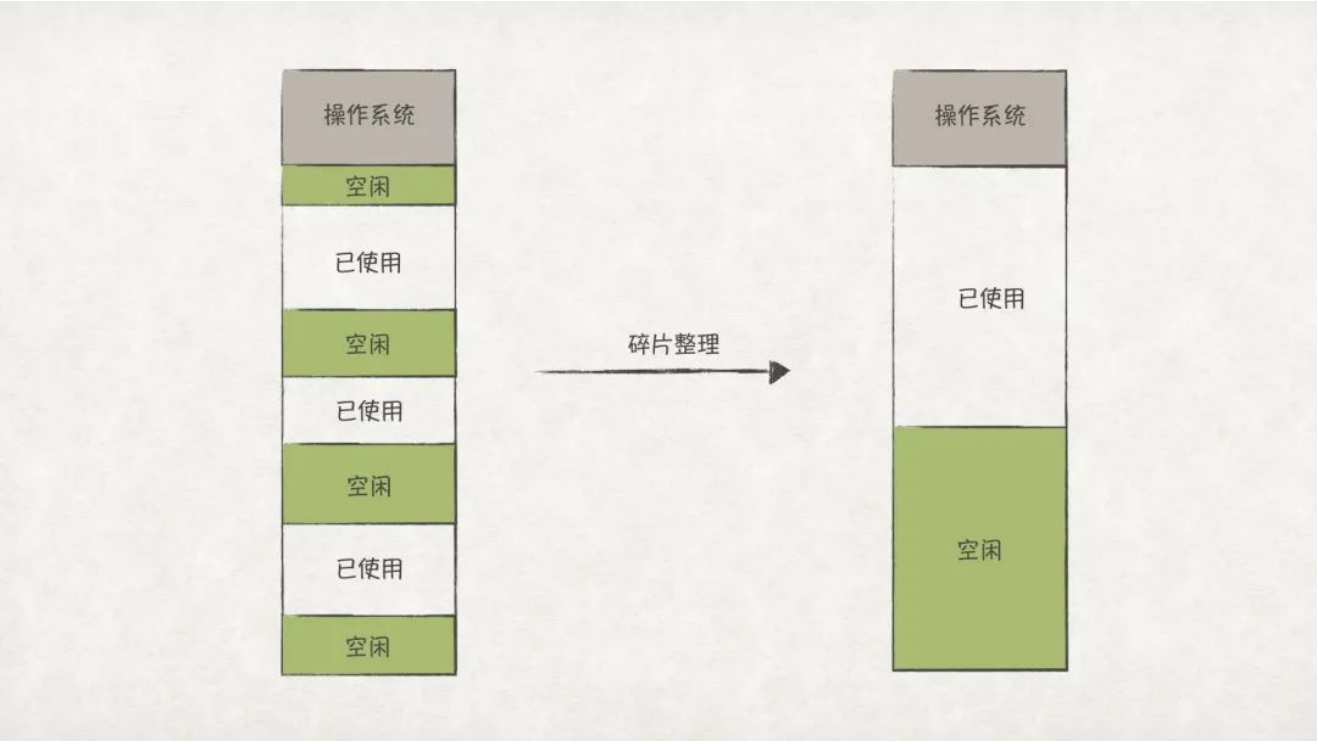


图14

但是碎片整理的代价极大，一方面需要进行多次内存拷贝；另一方面，在拷贝过程中，正在运行的程序必须停止，这对于一些以人机交互任务为主的应用程序，将会极大影响用户体验。

另一个解决方法就是接下来要介绍的**页式内存管理**。

页式内存管理

页式内存管理的思路，是将虚拟内存和物理内存都划分为多个**固定大小的区域**，这些区域我们称之为**页（Page）**。页是内存的最小分配单位，一个应用程序的虚拟页可以存放在任意一个空闲的物理页中。

> 物理内存中的页，我们通常称之为页帧（Page Frame）

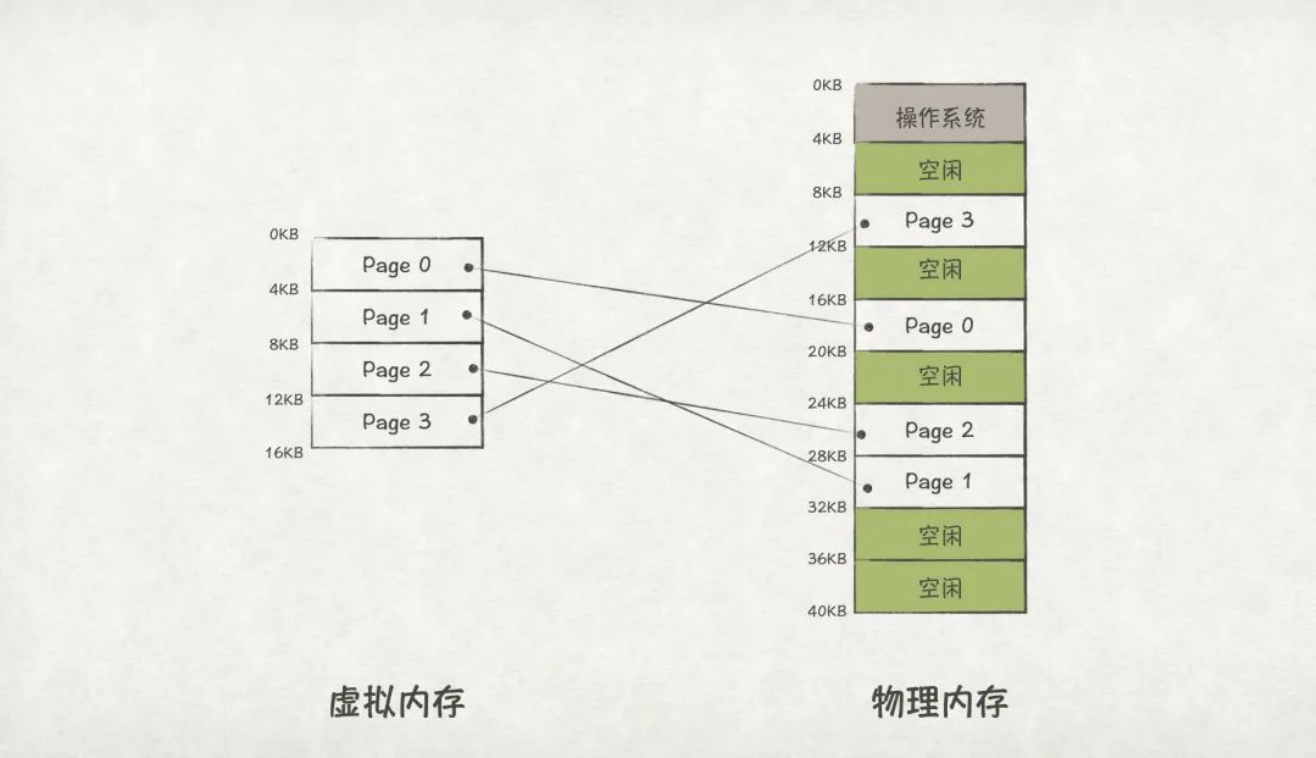


图15

因为页的大小是固定的，而且作为最小的分配单位，这样就可以解决段式内存管理中内存碎片的问题了。

> 但页内仍然有可能存在内存碎片。

地址转换

页式内存管理使用**页表（Page Table）**来进行虚拟地址到物理地址到转换，地址转换的关键步骤如下：

1）根据虚拟页找到对应的物理页帧

每个虚拟页都有一个编号，叫做**VPN（Virtual Page Number）**；相应的，每个物理页帧也有一个编号，叫做**PFN（Physical Frame Number）**。页表存储的就是VPN到PFN的映射关系。

2）找到地址在物理页帧内的偏移（Offset）

地址在物理页帧内的偏移与在虚拟页内的偏移保持一致。

我们可以将虚拟地址划分成两部分，分别存储VPN和Offset，这样就能通过VPN找到PFN，从而得到PFN+Offset的实际物理地址了。

比如，假设虚拟内存空间大小为64Byte（6位地址），页的大小为16Byte，那么整个虚拟内存空间一共有4个页。因此我们可以使用高2位来存储VPN，低4位存储Offset：

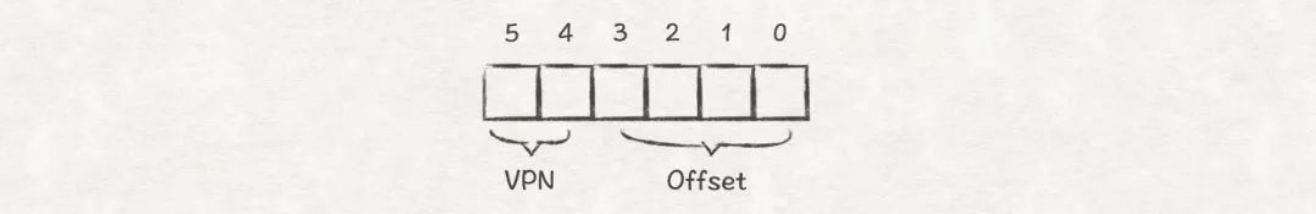


图16

下面看一个转换例子，VPN（01）通过页表找到对应的PFN（111），虚拟地址和物理地址的页内偏移都是0100，那么虚拟地址010100对应的物理地址就是1110100了。

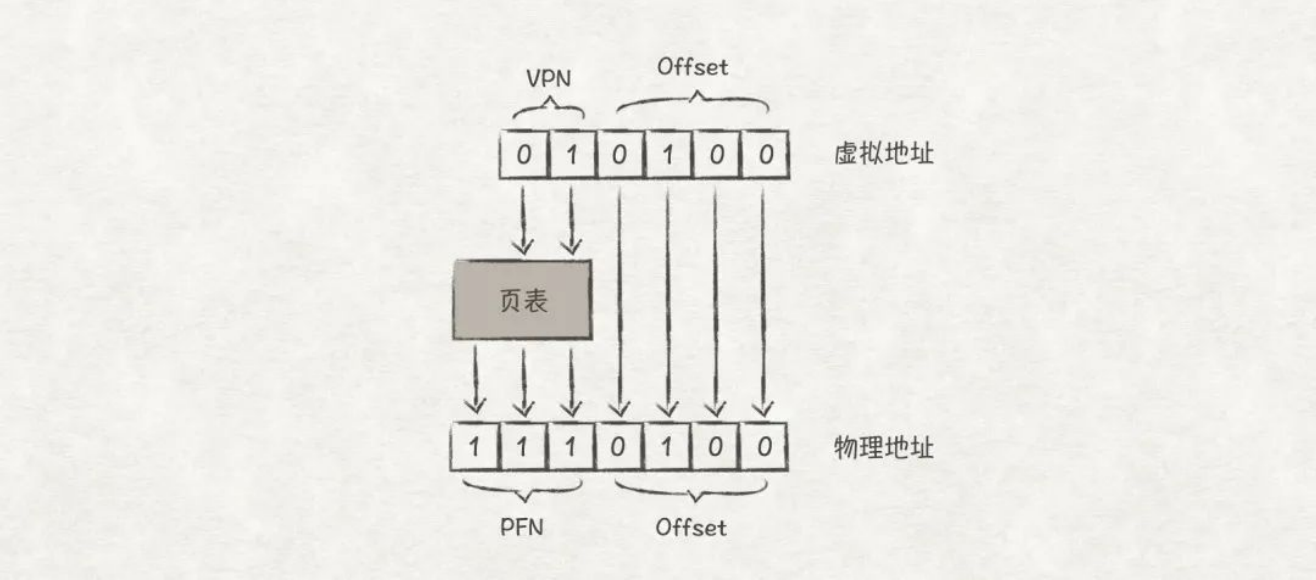


图17

页表和页表项

OS为每个程序都分配了一个页表，存储在内存当中，页表里由多个**页表项（PTE，**Page Table Entry）组成。我们可以把页表看成是一个数组，数组的元素为PTE：

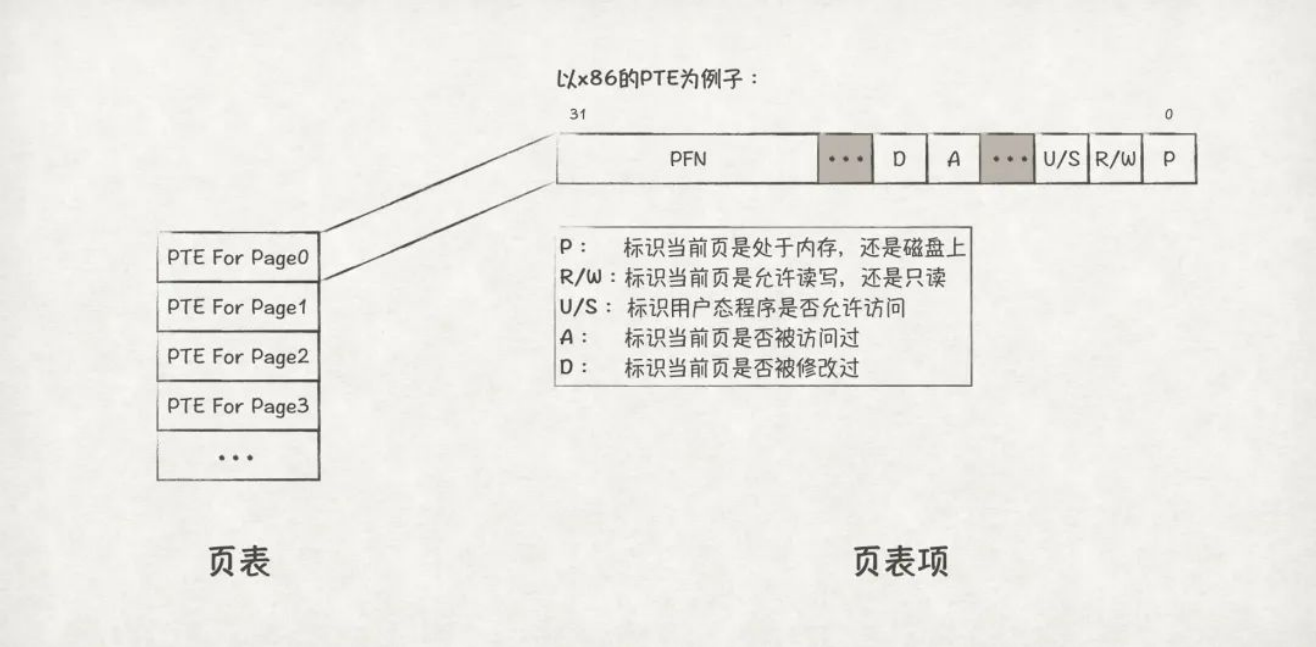


图18

以x86系统下的PTE组成为例，PTE一共占32位，除了PFN之外，还有一些比较重要的信息，比如**P**（Present）标识当前页是否位于内存上（或是磁盘上）；**R/W**（Read/Write）标识当前页是否允许读写（或是只读）；**U/S**（User/Supervisor）标识当前页是否允许用户态访问；**A**（Access）标识当前页是否被访问过，在判断当前页是否为热点数据、页换出时特别有用；**D**（Dirty）标识当前页是否被修改过。

页式内存管理的缺点

地址转换效率低

根据前文介绍，我们可以总结页式内存管理机制下地址转换的算法如下：

```
// 从虚拟地址上得到VPN
VPN = (VirtualAddress & VPN_MASK) >> SHIFT
// 找到VPN对应的PTE的内存地址
PTEAddr = PTBR + (VPN * sizeof(PTE))
// 访问主存，获取PTE
PTE = AccessMemory(PTEAddr)
// 有效性校验
if (PTE.Valid == False)
```

```
    RaiseException(INVALID_ACCESS)
else
    // 获取页内偏移量
    offset = VirtualAddress & OFFSET_MASK
    // 计算得出物理地址
    PhysAddr = (PTE.PFN << PFN_SHIFT) | offset
```

我们发现，每次地址转换都会**访问一次主存来获取页表**，比段式内存管理（无主存访问）低效很多。

占用空间大

假设地址空间为32-bit，页的大小固定为4KB，那么整个地址空间一共有$2^{32}/4KB = 2^{20}$个页表，也即页表一共有2^{20}个PTE。现假设每个PTE大小为4-byte，那么每个页表占用4MB的内存。如果整个系统中有100个程序在运行，那么光是页表就占用了400MB的内存，这同样是用户无法接受的。

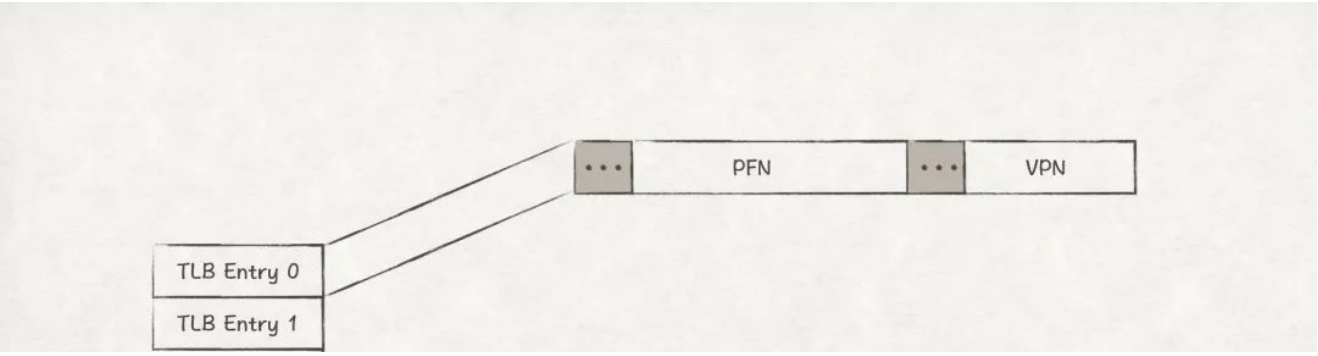
接下来，我们将介绍如何去优化页式内存管理的这两个显著缺点。

让页式管理的地址转换更快

TLB：Translation-Lookaside Buffer

根据前文所述，页式内存管理地址转换因为多了一次主存访问，导致效率很低。如果能够避免或者减少对主存的访问，那么就能让地址转换更快了。

很多人应该都可以想到通过**增加缓存**的方式提升效率，比如为避免频繁查询磁盘，我们一般在内存中增加一层缓存来提升数据访问效率。**那么为了提升访问主存中数据的效率，自然应该在离CPU更近的地方增加一层缓存**。这个离CPU更近的地方，就是前文提到的位于CPU芯片之内的MMU。而这个高速缓存，就是**TLB（Translation-Lookaside Buffer）**，它缓存了VPN到PFN到映射关系，类似于这样：



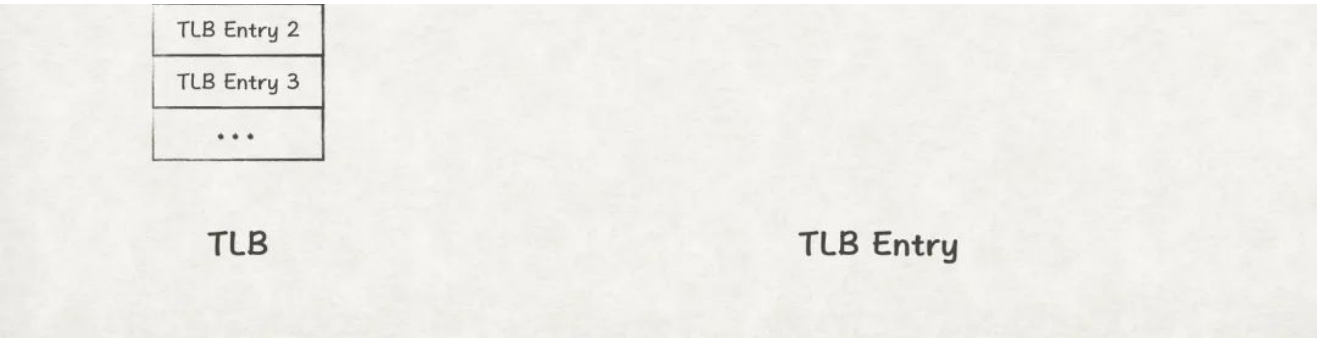


图19

增加TLB之后，地址转换的算法如下：

```
VPN = (VirtualAddress & VPN_MASK) >> SHIFT
(Success, TlbEntry) = TLB_Lookup(VPN)
if (Success == True)   // TLB Hit
    if (CanAccess(TlbEntry.ProtectBits) == True)
        Offset   = VirtualAddress & OFFSET_MASK
        PhysAddr = (TlbEntry.PFN << SHIFT) | Offset
    else
        RaiseException(PROTECTION_FAULT)
else                   // TLB Miss
    PTEAddr = PTBR + (VPN * sizeof(PTE))
    PTE = AccessMemory(PTEAddr)
    if (PTE.Valid == False)
        RaiseException(SEGMENTATION_FAULT)
    else if (CanAccess(PTE.ProtectBits) == False)
        RaiseException(PROTECTION_FAULT)
    else
        TLB_Insert(VPN, PTE.PFN, PTE.ProtectBits)
        RetryInstruction()
```

从上述算法可以发现，在TLB缓存命中（TLB Hit）时，能够避免直接访问主存，从而提升了地址转换的效率；但是在TLB缓存不命中（TLB Miss）时，仍然需要访问一次主存，而且还要往TLB中插入从主存中查询到的PFN，效率变得更低了。因此，**我们必须尽量避免TLB Miss的出现**。

更好地利用TLB

下面，我们通过一个数组遍历的例子来介绍如何更好地利用TLB。

假设我们要进行如下的一次数组遍历：

```
int sum = 0;
for (i = 0; i < 10; i++) {
    sum += a[i];
}
```

数组的内存的分布如下：

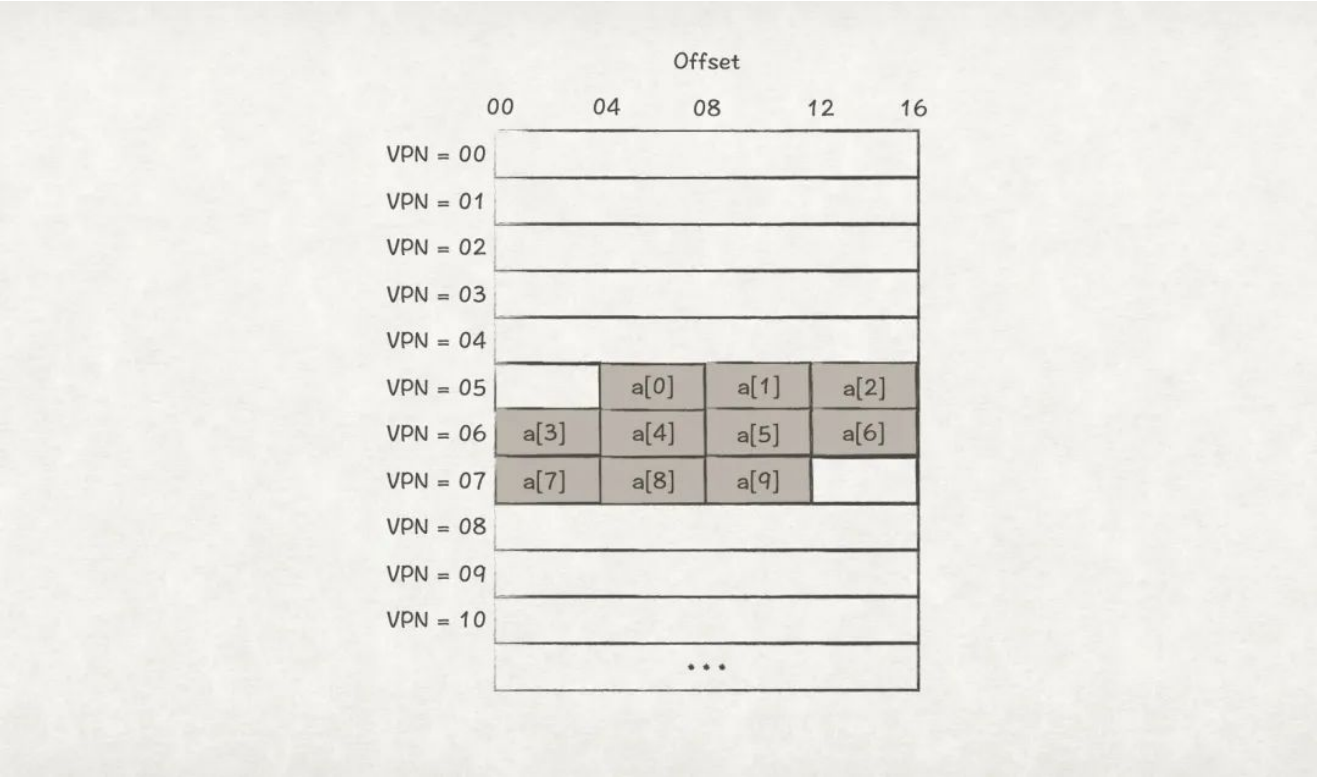


图20

a[0]~a[2]位于Page 5上，a[3]~a[6]位于Page 6上，a[7]~a[8]位于Page 7上。当我们首先访问a[0]时，由于Page 5并未在TLB缓存里，所以会先出现一次TLB Miss，接下来的a[1]和a[2]都是TLB Hit；同理，访问a[3]和a[7]时都是TLB Miss，a[4]~a[6]和a[8]~a[9]都是TLB Hit。所以，整个数组遍历下来，TLB的缓存命中情况为：`Miss, Hit, Hit, Miss, Hit, Hit, Hit, Miss, Hit, Hit`，TLB缓存命中率为70%。我们发现，访问同一页上的数据TLB的缓存更易命中，这就是**空间局部性**的原理。

接下来，我们再次重新遍历一次数组，由于经过上一次之后Page 5 ～ Page 7的转换信息已经在TLB缓存里里，所以第二次遍历的TLB命中情况为：`Hit, Hit, Hit, Hi`

t，Hit，Hit，Hit，Hit，Hit，Hit ，TLB缓存命中率为100%！这就是**时间局部性**的原理，短时间内访问同一内存数据也能够提升TLB缓存命中率。

TLB的上下文切换

因为TLB缓存的是**当前正在运行**程序的上下文信息，当出现程序切换时，TLB里面的上下文信息也必须更新，否则地址转换就会异常。解决方法主要有2种：

- 方法1：每次程序切换都清空TLB缓存（Flush TLB），让程序在运行过程中重新建立缓存。
- 方法2：允许TLB缓存多个程序的上下文信息，并通过**ASID（address space identifier**，地址空间标识符，可以理解为程序的PID）做区分。

方法1实现简单，但是每次程序切换都需要重新预热一遍缓存，效率较低，主流的做法是采用方法2。

> 需要注意的是TLB是嵌入到CPU芯片之内的，对于多核系统而言，如果程序在CPU之间来回切换，也是需要重新建立TLB缓存！因此，把一个程序绑定在一个固定的核上有助于提升性能。

让页表更小

大页内存

降低页表大小最简单的方法就是**让页更大**。前文的例子中，地址空间为32-bit，页的大小为4KB，PTE的大小为4-byte，那么每个页表需要4MB的内存空间。现在，我们把页的大小增加到16KB，其他保持不变，那么每个页表只需要$2^{32}/16KB = 2^{18}$个PTE，也即1MB内存，内存占用降低了4倍。

大页内存对TLB的使用也有优化效果，因TLB能够缓存的上下文数量是固定的，当页的数量更少时，上下文换出的频率会降低，TLB的缓存命中率也就增加了，从而让地址转换的效率更高。

段页式内存管理

根据前文所述，程序的地址空间中，堆与栈之间的空间很多时候都是处于未使用状态。对应到页表里，就是有很大一部分的PTE是invalid状态。但因为**页表要涵盖整个地址空间的范围**，这部分invalid的PTE只能留在页表中，从而造成了很大的空间浪费。

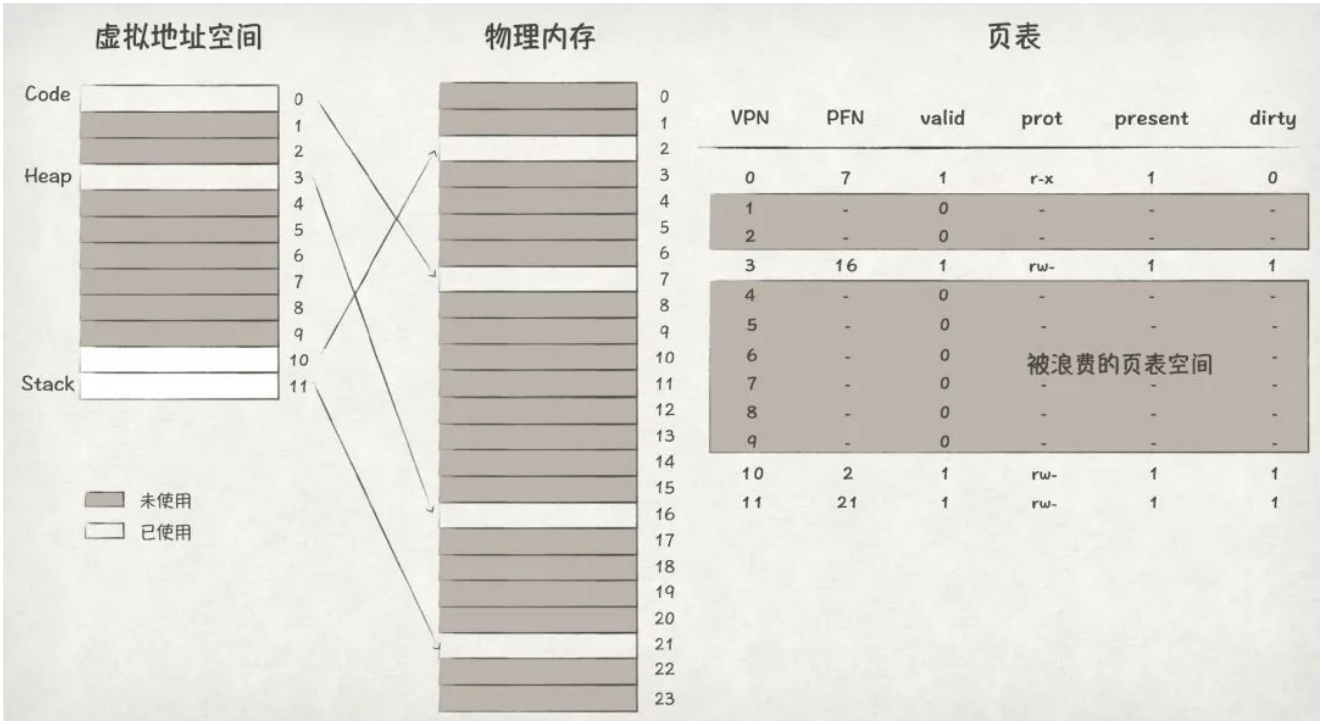


图21

前文中，我们通过段式内存管理解决了堆与栈之间内存空间的浪费问题。对应到页表中，我们也可以为页式内存管理引入段式管理的方式，也即**段页式内存管理**，解决页表空间浪费的问题。

具体的方法是，为程序的地址空间划分出多个段，比如Code、Heap、Stack等。然后，在每个段内单独进行页式管理，也即为每个段引入一个页表：

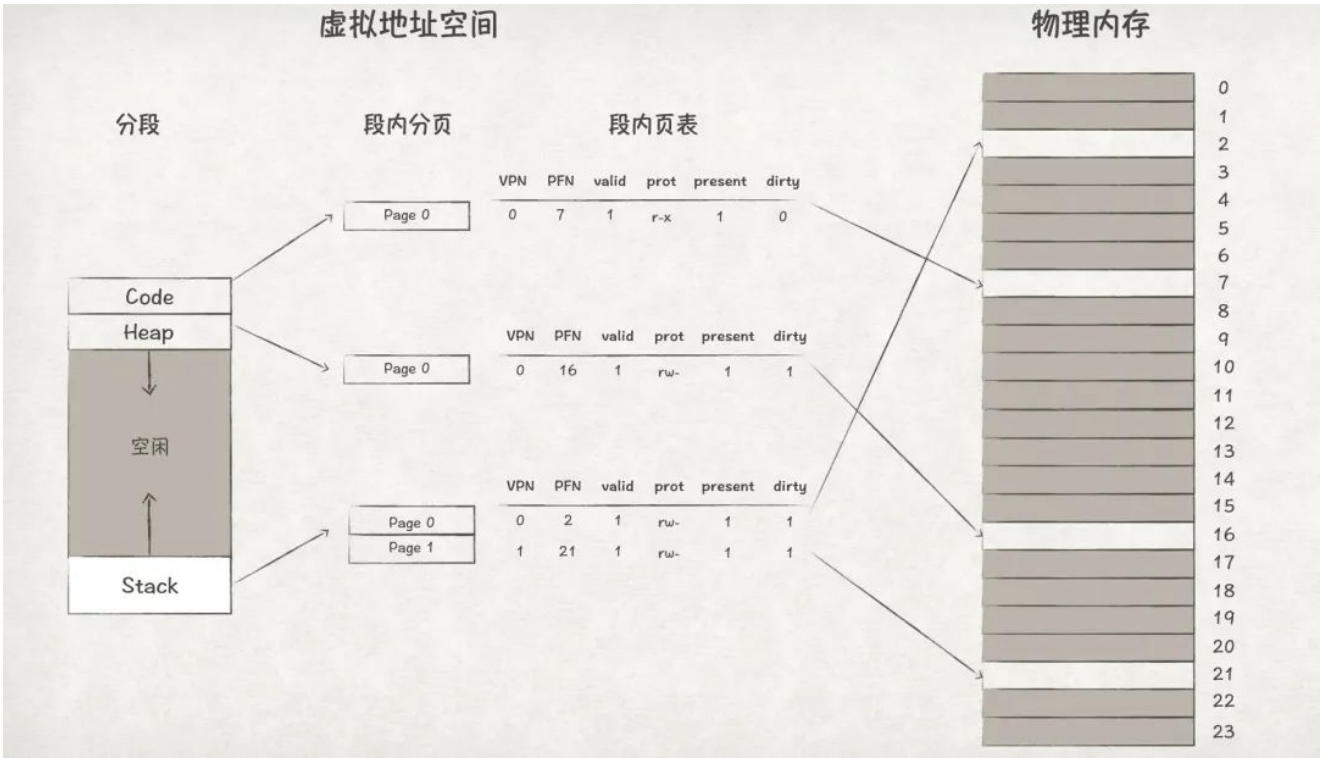


图22

从上图可知，将页表分段之后，页表不再需要记录那些处于空闲状态的页的PTE，从而节省了内存空间的消耗。

多级页表

降低页表大小另一个常见的方法就是**多级页表**（Multi-level Page Table），多级页表的思路也是减少处于空闲状态的页的PTE数量，但方法与段页式内存管理不同。**如果说段页式内存管理可以看成是将页表分段，那么多级页表则可以看成是将页表分页**。

具体的做法是将页表按照一定大小分成多个部分（**页目录**，Page Directory，PD），如果某个页目录下所有的页都是处于空闲状态，则无须为该页目录下的页申请PTE。

以二级页表为例，下图对比了普通页表和多级页表的构成差异：

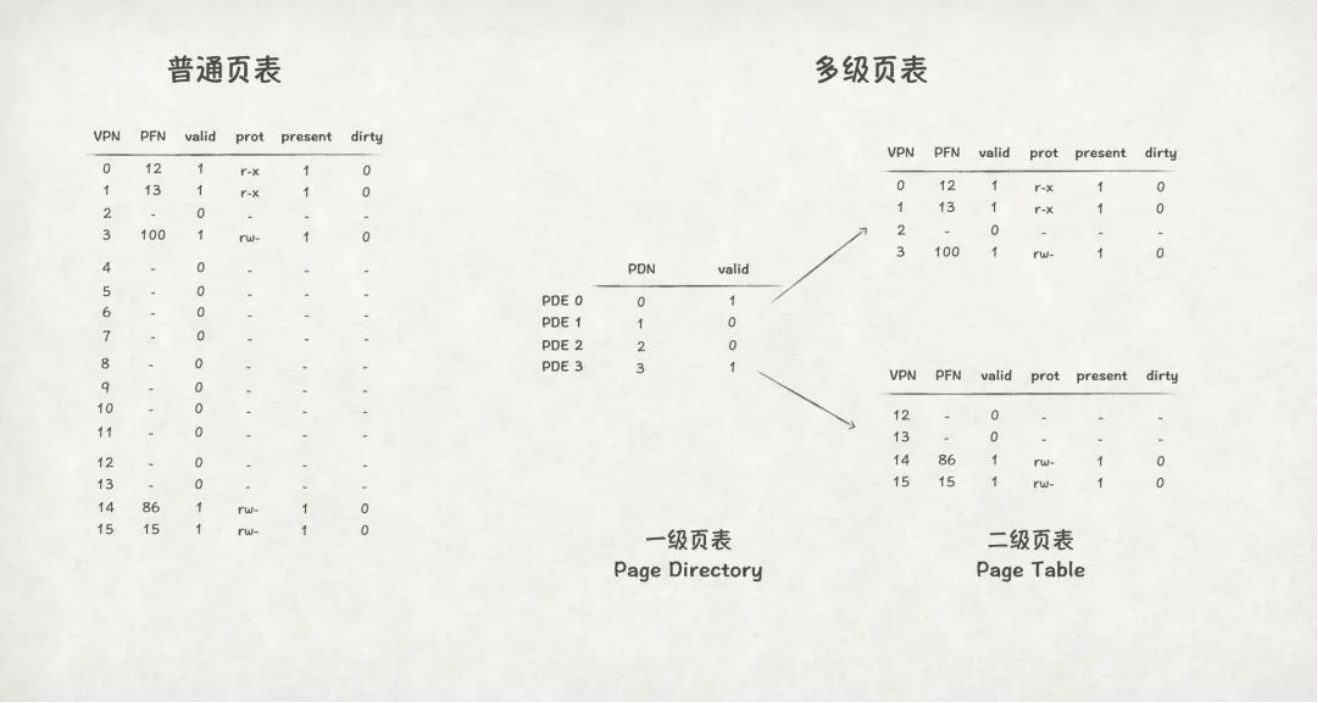


图23

下面，我们再对比一下普通页表和多级页表的空间消耗。还是假设地址空间为32-bit，页的大小为4KB，PTE的大小为4-byte，一共有2^{20}个页，那么普通页表需要**4MB**的内存空间。现在，我们将2^{20}个页切分为2^{10}份，也即有2^{10}个页目录，每个页目录下管理2^{10}个页，也即有2^{10}个**PDE**（Page Directory Entry）。假设PDE也占4-byte内存，且根据**20/80定律**假设有80%的页处于空闲状态，那么二级页表只需要**0.804MB**！（$2^{10} * 4 + 2^{20} * 4 * (1 - 80\%)$）

由此可见，多级页表能够有效降低页表的内存消耗。多级页表在实际运用中还是较为常见的，比如Linux系统采用的就是4级页表的结构。

Swap Sapce: 磁盘交换区

到目前为止，我们都是假设物理内存足够大，可以容纳所有程序的虚拟内存空间。然而，这往往是不切实际的，以32-bit地址空间为例，一个程序的虚拟内存为4G，假设有100个程序，那么一共需要400G的物理内存（忽略共享部分）！另外，程序运行过程中，并不是一直都需要所有的页，很多时候只需要其中的一小部分。

因此，**如果我们可以先把那些暂时用不到的页先存在磁盘上，等需要用到时再加载到内存上，那么就可以节省很多物理内存**。磁盘中用来存放这些页的区域，被称作**Swap Sapce**，也即交换区。

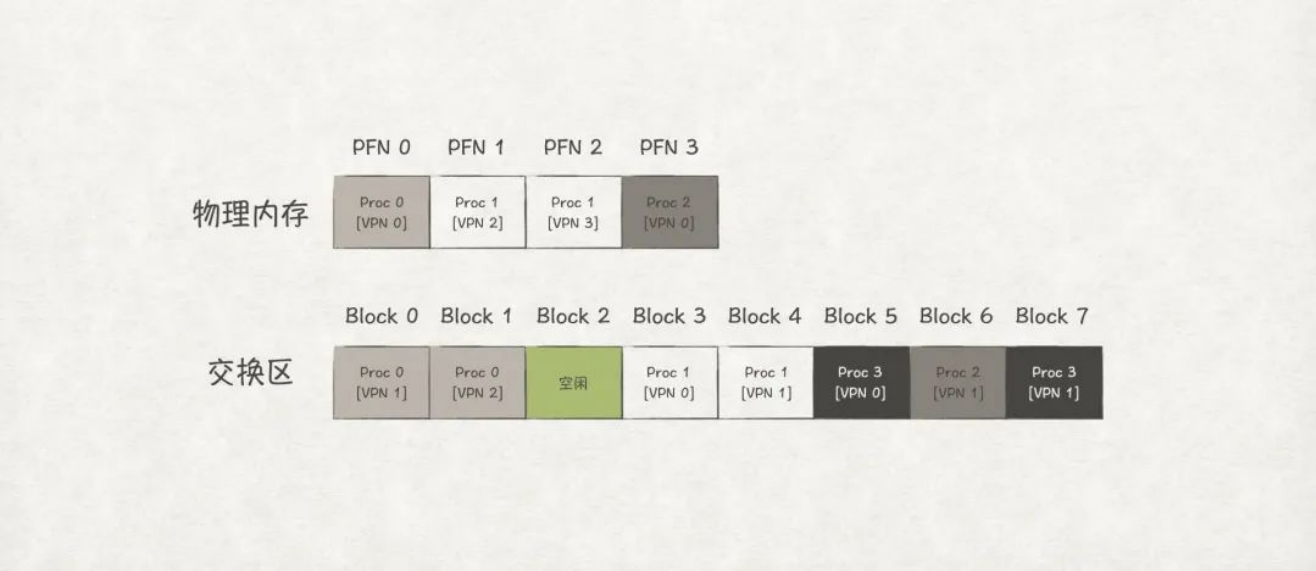


图24

在这种机制之下，当程序访问某一个地址，而这个地址所在的页又不在内存上时，就会触发**缺页（Page Fault）中断**。就像TLB缓存不命中时会带来额外的开销一样，**缺页也会导致内存的访问效率降低**。因为在处理缺页中断时，OS必须从磁盘交换区上把数据加载到内存上；而且当空闲内存不足时，OS还必须将内存上的某些页换出到交换区中。这一进一出的磁盘IO访问也直接导致缺页发生时，内存访问的效率下降许多。

因此，在空闲内存不足时，页的换出策略显得极为重要。如果把一个即将要被访问的页换出到交换区上，就会带来本可避免的无谓消耗。页的换出策略很多，常见的有**FIFO**（先进先出）、**Random**（随机）、**LRU**（最近最少使用）、**LFU**（最近最不经常使用）等。在常见的工作负载下，FIFO和Random算法的效果较差，实际用的不多；LRU

和LFU算法都是建立在历史内存访问统计的基础上，因此表现较前两者好些，实际应用也多一些。目前很多主流的操作系统的页换出算法都是在LRU或LFU的基础上进行优化改进的结果。

最后

本文主要介绍了OS内存管理的一些基本原理，从独占式内存管理，到页式内存管理，这过程中经历了许多次优化。这其中的每一种优化手段，都朝着如下3个目标前进：

1、**透明化**（transparency）。内存管理的细节对程序不可见，换句话说，程序可以自认为独占整个内存空间。

2、**效率**（efficiency ）。地址转换和内存访问的效率要高，不能让程序运行太慢；空间利用效率也要高，不能占用太多空闲内存。

3、**保护**（protection）。保证OS自身不受应用程序的影响；应用程序之间也不能相互影响。

当然，目前主流的操作系统（如Linux、MacOS等）的内存管理机制要比本文介绍的原理复杂许多，但本质原理依然离不开本文所描述的几种基础的内存管理原理。

> 参考
>
> 1、Operating Systems: Three Easy Pieces, Remzi H Arpaci-Dusseau / Andrea C Arpaci-Dusseau
>
> 2、为什么 HugePages 可以提升数据库性能 ，面向信仰编程
>
> 3、**探索CPU的调度原理**，元闰子的邀请
>
> 更多文章请关注微信公众号：**元闰子的邀请**

JusterZhu

绕过CDN获取服务器真实IP地址

菜鸟学信安

10个实用 Linux Shell 脚本案例

咸鱼爱搞机