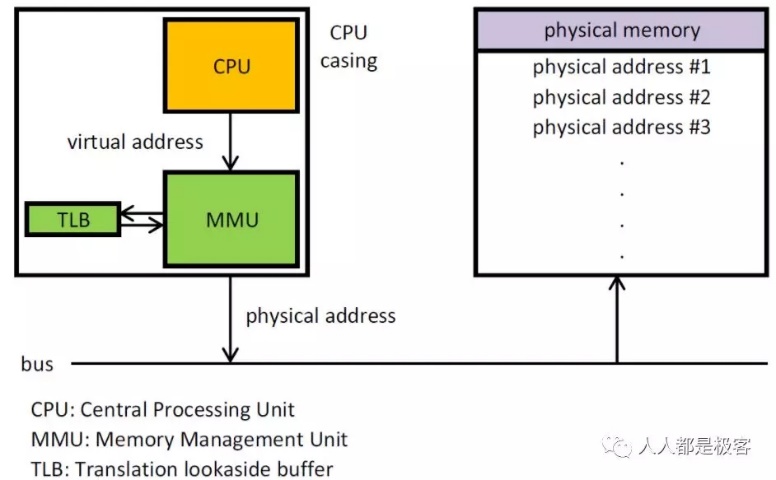
# CPU是如何访问内存的？

内存管理可以说是一个比较难学的模块，之所以比较难学。一是内存管理涉及到硬件的实现原理和软件的复杂算法，二是网上关于内存管理的解释有太多错误的解释。希望可以做个内存管理的系列，从硬件实现到底层内存分配算法，再从内核分配算法到应用程序内存划分，一直到内存和硬盘如何交互等，彻底理解内存管理的整个脉络框架。本节主要讲解硬件原理和分页管理。

## CPU通过MMU访问内存

我们先来看一张图：



从图中可以清晰地看出，CPU、MMU、DDR 这三部分在硬件上是如何分布的。首先 CPU 在访问内存的时候都需要通过 MMU 把虚拟地址转化为物理地址，然后通过总线访问内存。MMU 开启后 CPU 看到的所有地址都是虚拟地址，CPU 把这个虚拟地址发给 MMU 后，MMU 会通过页表在页表里查出这个虚拟地址对应的物理地址是什么，从而去访问外面的 DDR（内存条）。

所以搞懂了 MMU 如何把虚拟地址转化为物理地址也就明白了 CPU 是如何通过 MMU 来访问内存的。

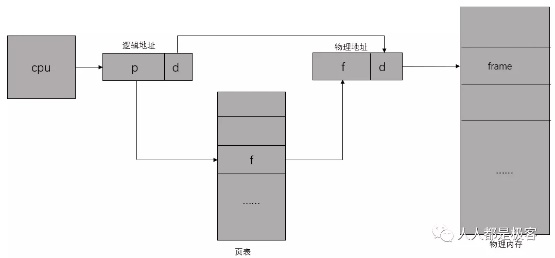
MMU 是通过页表把虚拟地址转换成物理地址，页表是一种特殊的数据结构，放在系统空间的页表区存放逻辑页与物理页帧的对应关系，每一个进程都有一个自己的页表。

CPU 访问的虚拟地址可以分为：p（页号），用来作为页表的索引；d（页偏移），该页内的地址偏移。现在我们假设每一页的大小是 4KB，而且页表只有一级，那么页表长成下面这个样子（页表的每一行是32个 bit，前20 bit 表示页号 p，后面12 bit 表示页偏移 d）：



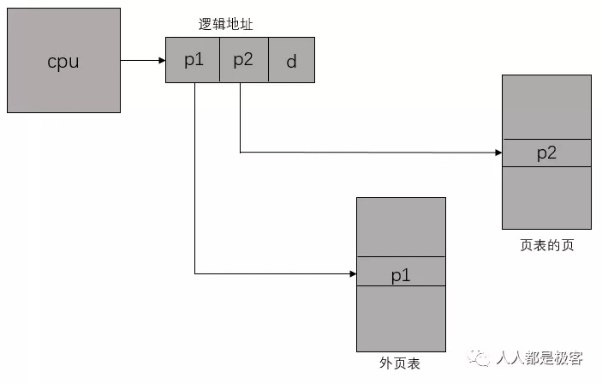
CPU，虚拟地址，页表和物理地址的关系如下图：

页表包含每页所在物理内存的基地址，这些基地址与页偏移的组合形成物理地址，就可送交物理单元。



上面我们发现，如果采用一级页表的话，每个进程都需要1个4MB的页表(假如虚拟地址空间为32位（即4GB）、每个页面映射4KB以及每条页表项占4B，则进程需要1M个页表项（4GB / 4KB = 1M），即页表（每个进程都有一个页表）占用4MB（1M \* 4B = 4MB）的内存空间)。然而对于大多数程序来说，其使用到的空间远未达到4GB，何必去映射不可能用到的空间呢？也就是说，一级页表覆盖了整个4GB虚拟地址空间，但如果某个一级页表的页表项没有被用到，也就不需要创建这个页表项对应的二级页表了，即可以在需要时才创建二级页表。做个简单的计算，假设只有20%的一级页表项被用到了，那么页表占用的内存空间就只有0.804MB（1K \* 4B + 0.2 \* 1K \* 1K \* 4B = 0.804MB）。除了在需要的时候创建二级页表外，还可以通过将此页面从磁盘调入到内存，只有一级页表在内存中，二级页表仅有一个在内存中，其余全在磁盘中（虽然这样效率非常低），则此时页表占用了8KB（1K \* 4B + 1 \* 1K \* 4B = 8KB），对比上一步的0.804MB，占用空间又缩小了好多倍！总而言之，采用多级页表可以节省内存。

二级页表就是将页表再分页。仍以之前的32位系统为例，一个逻辑地址被分为20位的页码和12位的页偏移d。因为要对页表进行再分页，该页号可分为10位的页码p1和10位的页偏移p2。其中p1用来访问外部页表的索引，而p2是是外部页表的页偏移。



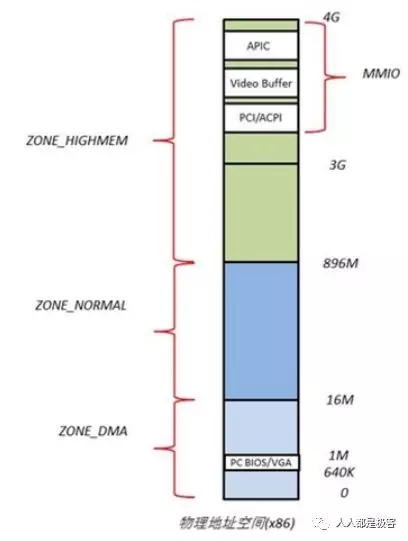
# 物理地址和虚拟地址的分布

[上一节](https://mp.weixin.qq.com/s?__biz=MzIxMjE1MzU4OA==&mid=2648918636&idx=1&sn=8e07e312ee86e5cd39328eb41396bc87&chksm=8f5d8ff7b82a06e1e3dd892f32b84ecb35c4d4bd98520a846873af32bfb477636a73e6dcbb75&token=1659295715&lang=zh_CN&scene=21#wechat_redirect)内容的学习我们知道了CPU是如何访问内存的，CPU拿到内存后就可以向其它人（kernel的其它模块、内核线程、用户空间进程、等等）提供服务，主要包括：

* 以虚拟地址（VA）的形式，为应用程序提供远大于物理内存的虚拟地址空间（Virtual Address Space）
* 每个进程都有独立的虚拟地址空间，不会相互影响，进而可提供非常好的内存保护（memory protection）
* 提供内存映射（Memory Mapping）机制，以便把物理内存、I/O空间、Kernel Image、文件等对象映射到相应进程的地址空间中，方便进程的访问
* 提供公平、高效的物理内存分配（Physical Memory Allocation）算法
* 提供进程间内存共享的方法（以虚拟内存的形式），也称作Shared Virtual Memory
* 在提供这些服务之前需要对内存进行合理的划分和管理，下面让我们看下是如何划分的。

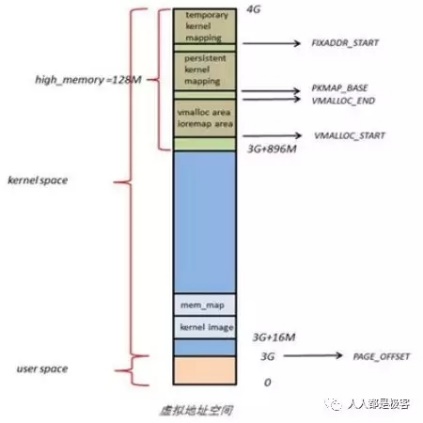
## 物理地址空间布局

Linux系统在初始化时，会根据实际的物理内存的大小，为每个物理页面创建一个page对象，所有的page对象构成一个mem\_map数组。进一步，针对不同的用途，Linux内核将所有的物理页面划分到3类内存管理区中，如图，分别为ZONE\_DMA，ZONE\_NORMAL，ZONE\_HIGHMEM。



* ZONE\_DMA 的范围是 0~16M，该区域的物理页面专门供 I/O 设备的 DMA 使用。之所以需要单独管理 DMA 的物理页面，是因为 DMA 使用物理地址访问内存，不经过 MMU，并且需要连续的缓冲区，所以为了能够提供物理上连续的缓冲区，必须从物理地址空间专门划分一段区域用于 DMA。
* ZONE\_NORMAL 的范围是 16M~896M，该区域的物理页面是内核能够直接使用的。
* ZONE\_HIGHMEM 的范围是 896M~结束，该区域即为高端内存，内核不能直接使用。

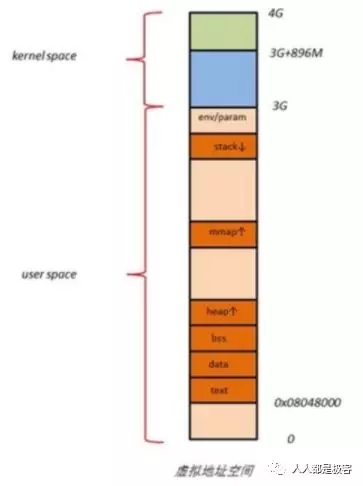
## Linux内核空间虚拟地址分布



在 Kernel Image 下面有 16M 的内核空间用于 DMA 操作。位于内核空间高端的 128M 地址主要由3部分组成，分别为 vmalloc area、持久化内核映射区、临时内核映射区。

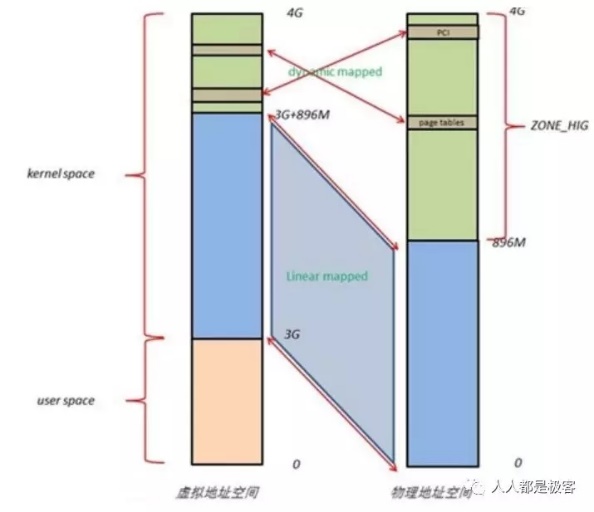
由于 ZONE\_NORMAL 和内核线性空间存在直接映射关系，所以内核会将频繁使用的数据如 Kernel 代码、GDT、IDT、PGD、mem\_map 数组等放在 ZONE\_NORMAL 里。而将用户数据、页表（PT）等不常用数据放在 ZONE\_HIGHMEM 里，只在要访问这些数据时才建立映射关系（kmap()）。比如，当内核要访问 I/O 设备存储空间时，就使用 ioremap() 将位于物理地址高端的 mmio 区内存映射到内核空间的 vmalloc area 中，在使用完之后便断开映射关系。

## Linux用户空间虚拟地址分布



用户进程的代码区一般从虚拟地址空间的 0x08048000 开始，这是为了便于检查空指针。代码区之上便是数据区，未初始化数据区，堆区，栈区，以及参数、全局环境变量。

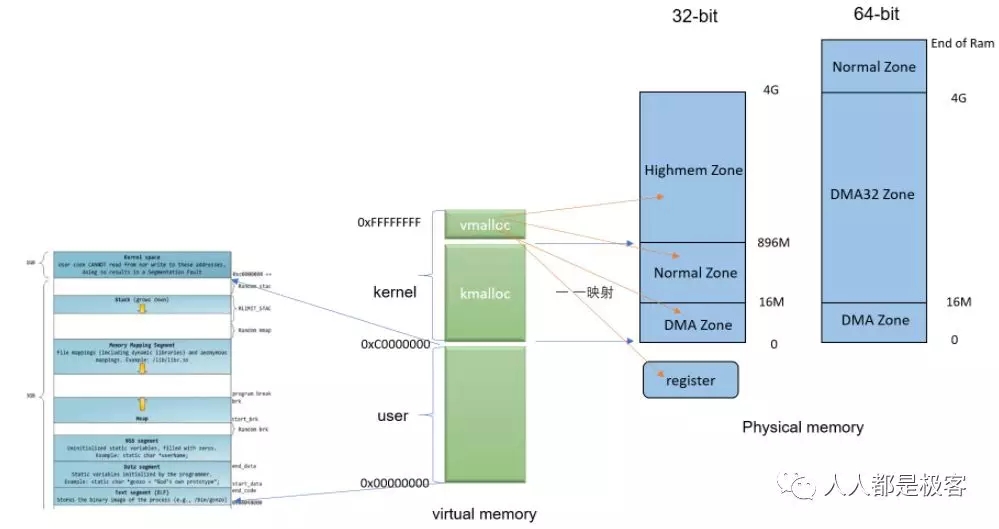
## Linux物理地址和虚拟地址的关系



Linux 将 4G 的线性地址空间分为2部分，0~3G 为 user space，3G~4G 为 kernel space。

由于开启了分页机制，内核想要访问物理地址空间的话，必须先建立映射关系，然后通过虚拟地址来访问。为了能够访问所有的物理地址空间，就要将全部物理地址空间映射到 1G 的内核线性空间中，这显然不可能。于是，内核将 0~896M 的物理地址空间一对一映射到自己的线性地址空间中，这样它便可以随时访问 ZONE\_DMA 和 ZONE\_NORMAL 里的物理页面；此时内核剩下的 128M 线性地址空间不足以完全映射所有的 ZONE\_HIGHMEM，Linux 采取了动态映射的方法，即按需的将 ZONE\_HIGHMEM 里的物理页面映射到 kernel space 的最后 128M 线性地址空间里，使用完之后释放映射关系，以供其它物理页面映射。虽然这样存在效率的问题，但是内核毕竟可以正常的访问所有的物理地址空间了。

到这里我们应该知道了 Linux 是如何用虚拟地址来映射物理地址的，最后我们用一张图来总结一下：



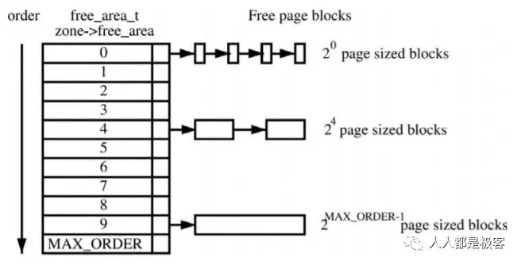
# Linux内核内存管理算法Buddy和Slab

有了前两节的学习相信读者已经知道CPU所有的操作都是建立在虚拟地址上处理(这里的虚拟地址分为内核态虚拟地址和用户态虚拟地址)，CPU看到的内存管理都是对page的管理，接下来我们看一下用来管理page的经典算法--Buddy。

## Buddy分配算法

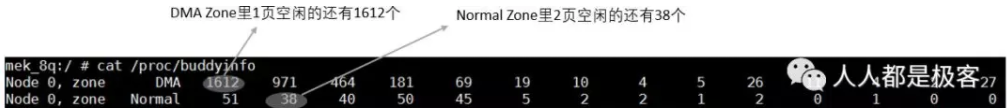


假设这是一段连续的页框，阴影部分表示已经被使用的页框，现在需要申请一个连续的5个页框。这个时候，在这段内存上不能找到连续的5个空闲的页框，就会去另一段内存上去寻找5个连续的页框，这样子，久而久之就形成了页框的浪费。为了避免出现这种情况，Linux内核中引入了伙伴系统算法(Buddy system)。把所有的空闲页框分组为11个块链表，每个块链表分别包含大小为1，2，4，8，16，32，64，128，256，512和1024个连续页框的页框块。最大可以申请1024个连续页框，对应4MB大小的连续内存。每个页框块的第一个页框的物理地址是该块大小的整数倍，如图：

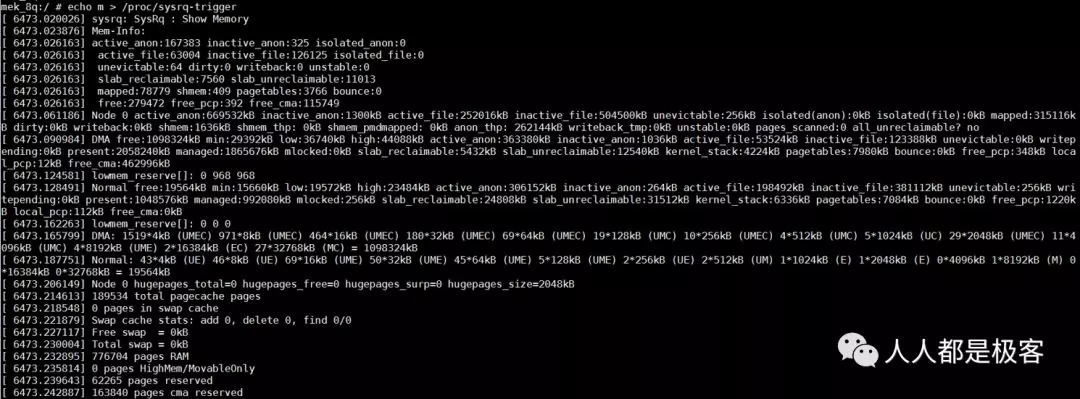


假设要申请一个256个页框的块，先从256个页框的链表中查找空闲块，如果没有，就去512个页框的链表中找，找到了则将页框块分为2个256个页框的块，一个分配给应用，另外一个移到256个页框的链表中。如果512个页框的链表中仍没有空闲块，继续向1024个页框的链表查找，如果仍然没有，则返回错误。页框块在释放时，会主动将两个连续的页框块合并为一个较大的页框块。

从上面可以知道Buddy算法一直在对页框做拆开合并拆开合并的动作。Buddy算法牛逼就牛逼在运用了世界上任何正整数都可以由2^n的和组成。这也是Buddy算法管理空闲页表的本质。  
空闲内存的信息我们可以通过以下命令获取：



也可以通过echo m > /proc/sysrq-trigger来观察buddy状态，与/proc/buddyinfo的信息是一致的：



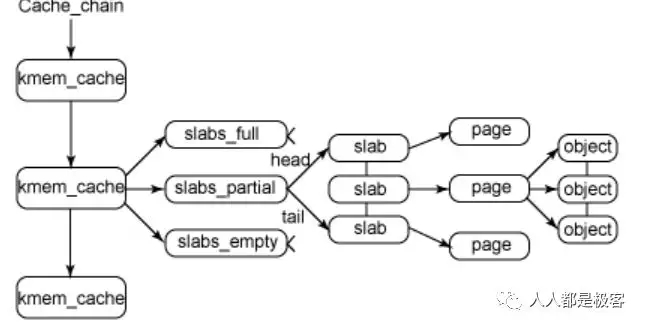
## CMA

细心的读者或许会发现当Buddy算法对内存拆拆合合的过程中会造成碎片化的现象，以至于内存后来没有了大块的连续内存，全是小块内存。当然这对应用程序是不影响的(前面我们讲过用页表可以把不连续的物理地址在虚拟地址上连续起来)，但是内核态就没有办法获取大块连续的内存（比如DMA, Camera, GPU都需要大块物理地址连续的内存）。

在嵌入式设备中一般用CMA来解决上述的问题。CMA的全称是contiguous memory allocator， 其工作原理是：预留一段的内存给驱动使用，但当驱动不用的时候，CMA区域可以分配给用户进程用作匿名内存或者页缓存。而当驱动需要使用时，就将进程占用的内存通过回收或者迁移的方式将之前占用的预留内存腾出来，供驱动使用。

## Slab

在Linux中，伙伴系统（buddy system）是以页为单位管理和分配内存。但是现实的需求却以字节为单位，假如我们需要申请20Bytes，总不能分配一页吧！那岂不是严重浪费内存。那么该如何分配呢？slab分配器就应运而生了，专为小内存分配而生。slab分配器分配内存以Byte为单位。但是slab分配器并没有脱离伙伴系统，而是基于伙伴系统分配的大内存进一步细分成小内存分配。我们先来看一张图：



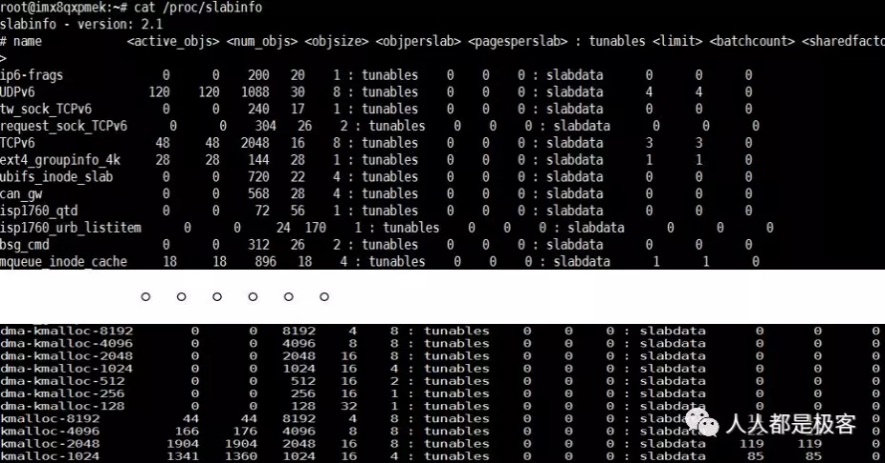
kmem\_cache是一个cache\_chain的链表，描述了一个高速缓存，每个高速缓存包含了一个slabs的列表，这通常是一段连续的内存块。存在3种slab：

* slabs\_full(完全分配的slab)
* slabs\_partial(部分分配的slab)
* slabs\_empty(空slab,或者没有对象被分配)。

slab是slab分配器的最小单位，在实现上一个slab有一个货多个连续的物理页组成（通常只有一页）。单个slab可以在slab链表之间移动，例如如果一个半满slab被分配了对象后变满了，就要从slabs\_partial中被删除，同时插入到slabs\_full中去。

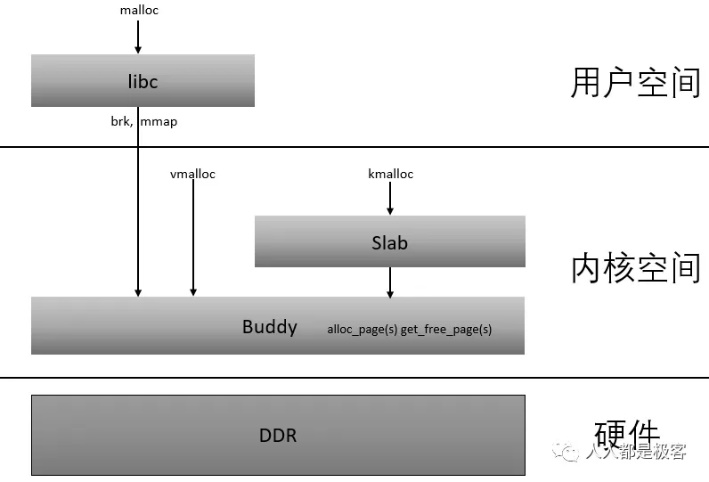
为了进一步解释，这里举个例子来说明，用struct kmem\_cache结构描述的一段内存就称作一个slab缓存池。一个slab缓存池就像是一箱牛奶，一箱牛奶中有很多瓶牛奶，每瓶牛奶就是一个object。分配内存的时候，就相当于从牛奶箱中拿一瓶。总有拿完的一天。当箱子空的时候，你就需要去超市再买一箱回来。超市就相当于partial链表，超市存储着很多箱牛奶。如果超市也卖完了，自然就要从厂家进货，然后出售给你。厂家就相当于伙伴系统。

可以通过下面命令查看slab缓存的信息：



## 总结

从内存DDR分为不同的ZONE，到CPU访问的Page通过页表来映射ZONE，再到通过Buddy算法和Slab算法对这些Page进行管理，我们应该可以从感官的角度理解了下图：



# Linux用户态进程的内存管理

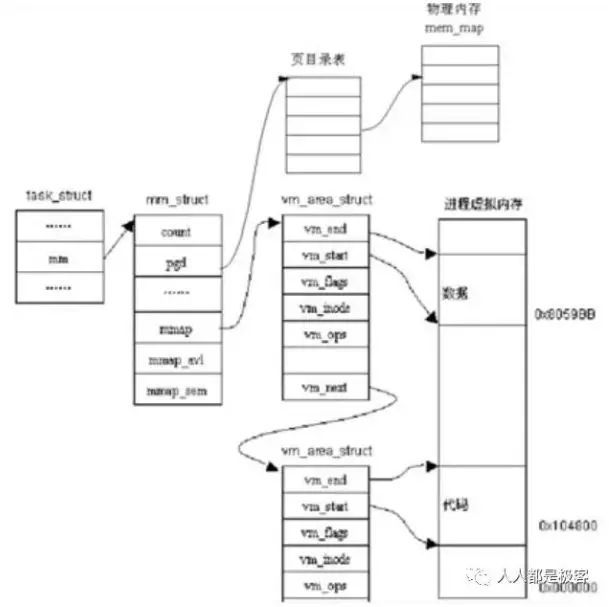
我们了解了内存在内核态是如何管理的，本篇文章我们一起来看下内存在用户态的使用情况，如果上一篇文章说是内核驱动工程师经常面对的内存管理问题，那本篇就是应用工程师常面对的问题。

相信大家都知道对用户态的内存消耗对象是进程，应用开发者面对的所有代码操作最后的落脚点都是进程，这也是说为什么内存和进程两个知识点的重要性，理解了内存和进程两大法宝，对所有软件开发的理解都会有了全局观（关于进程的知识以后再整理和大家分享）。

下面闲话少说，开始本篇的内容——进程的内存消耗和泄漏

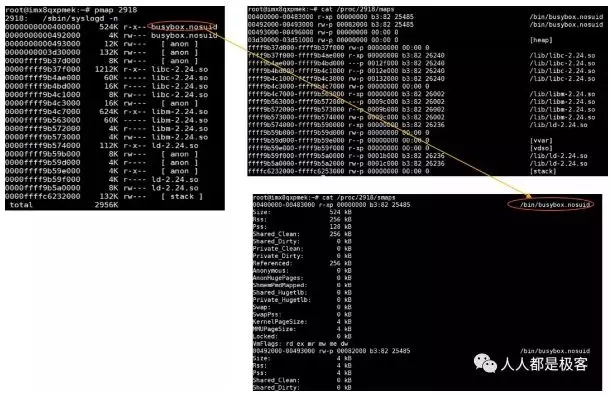
## 进程的虚拟地址空间VMA（Virtual Memory Area）

在linux操作系统中，每个进程都通过一个task\_struct的结构体描叙，每个进程的地址空间都通过一个mm\_struct描叙，c语言中的每个段空间都通过vm\_area\_struct表示，他们关系如下 :

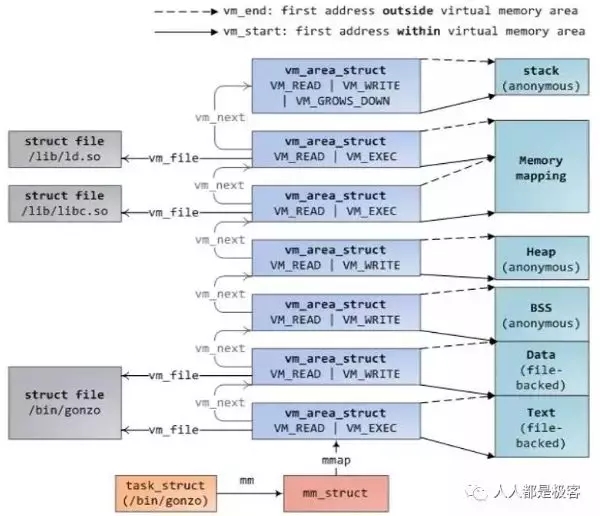


上图中，task\_struct中的mm\_struct就代表进程的整个内存资源，mm\_struct中的pgd为页表，mmap指针指向的vm\_area\_struct链表的每一个节点就代表进程的一个虚拟地址空间，即一个VMA。一个VMA最终可能对应ELF可执行程序的数据段、代码段、堆、栈、或者动态链接库的某个部分。

VMA的分布情况可以有通过pmap命令，及maps，smaps文件查看，如下图：

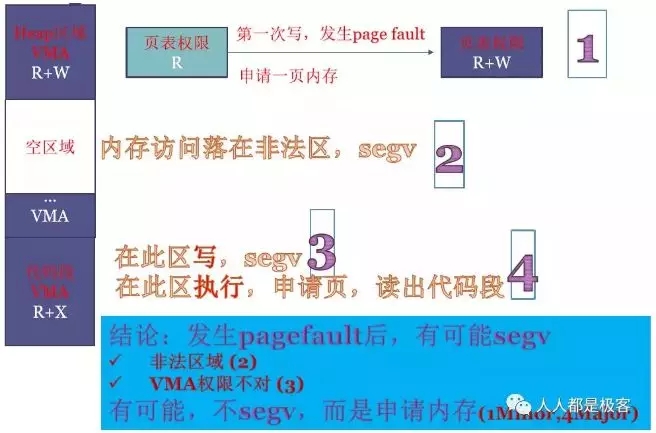


另，VMA的具体内容可参考下图。



## page fault的几种可能性

我们先来看张图：



（此图来源于宋宝华老师）

如，调用malloc申请100M内存，IA32下在0~3G虚拟地址中立刻就会占用到大小为100M的VMA，且符合堆的定义，这一段VMA的权限是R+W的。但由于Lazy机制，这100M其实并没有获得，这100M全部映射到一个物理地址相同的零页，且在页表中记录的权限为只读的。当100M中任何一页发生写操作时，MMU会给CPU发page fault（MMU可以从寄存器读出发生page fault的地址；MMU可以读出发生page fault的原因），Linux内核收到缺页中断，在缺页中断的处理程序中读出虚拟地址和原因，去VMA中查，发现是用户程序在写malloc的合法区域且有写权限，Linux内核就真正的申请内存，页表中对应一页的权限也修改为R+W。

如，程序中有野指针飞到了此程序运行时进程的VMA以外的非法区域，硬件就会收到page fault，进程会收到SIGSEGV信号报段错误并终止。如，程序中有野指针飞到了此程序运行时进程的VMA以外的非法区域，硬件就会收到page fault，进程会收到SIGSEGV信号报段错误并终止。

如，代码段在VMA中权限为R+X，如果程序中有野指针飞到此区域去写，则也会发生段错误。（另，malloc堆区在VMA中权限为R+W，如果程序的PC指针飞到此区域去执行，同样发生段错误。）

如，执行代码段时会发生缺页，Linux申请1页内存，并从硬盘读取出代码段，此时产生了IO操作，为major主缺页。如，执行代码段时会发生缺页，Linux申请1页内存，并从硬盘读取出代码段，此时产生了IO操作，为major主缺页。

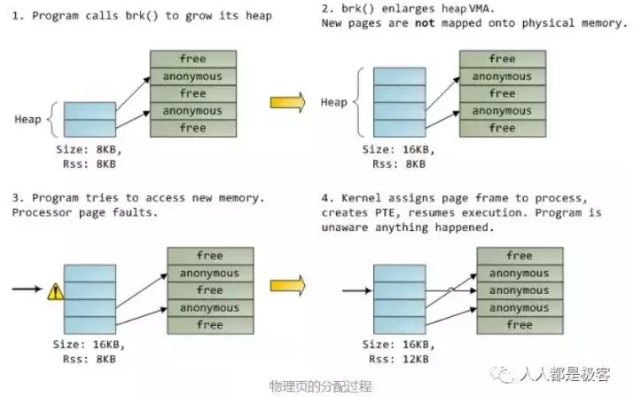


（此图来源于宋宝华老师）

综上，page fault后，Linux会查VMA，也会比对VMA中和页表中的权限，体现出VMA的重要作用。

## malloc分配的原理

malloc的过程其实就是把VMA分配到各种段当中，这时候是没有真正分配物理地址的。malloc 调用后，只是分配了内存的逻辑地址，在内核的mm\_struct 链表中插入vm\_area\_struct结构体，没有分配实际的内存。当分配的区域写入数据是，引发页中断，建立物理页和逻辑地址的映射。下图表示了这个过程。



从操作系统角度来看，进程分配内存有两种方式，分别由两个系统调用完成：brk和mmap（不考虑共享内存）。

* malloc小于128k的内存，使用brk分配内存，将\_edata往高地址推(只分配虚拟空间，不对应物理内存(因此没有初始化)，第一次读/写数据时，引起内核缺页中断，内核才分配对应的物理内存，然后虚拟地址空间建立映射关系)
* malloc大于128k的内存，使用mmap分配内存，在堆和栈之间找一块空闲内存分配(对应独立内存，而且初始化为0)

## 内存的消耗VSS RSS PSS USS

首先，我们评估一个进程的内存消耗都是指用户空间的内存，不包括内核空间的内存消耗 。这里我们用工具 procrank先来看下Linux进程的内存占用量 。

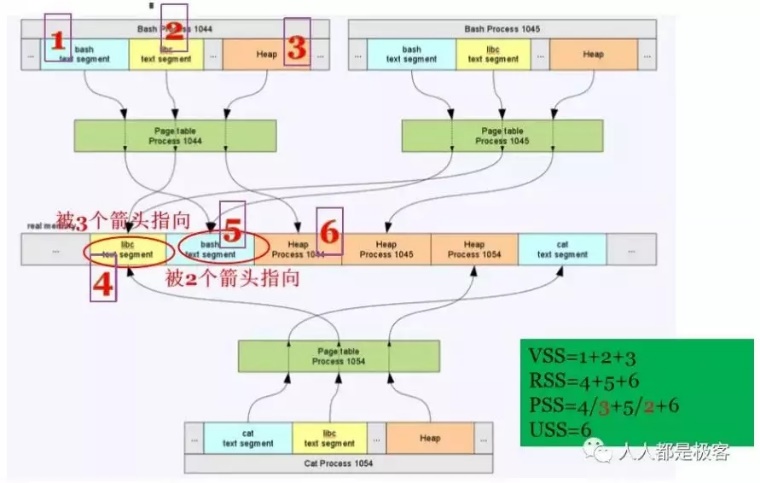


* VSS -Virtual Set Size 虚拟耗用内存（包含共享库占用的内存）
* RSS -Resident Set Size 实际使用物理内存（包含共享库占用的内存）
* PSS -Proportional Set Size 实际使用的物理内存（比例分配共享库占用的内存）
* USS -Unique Set Size 进程独自占用的物理内存（不包含共享库占用的内存）

下面再用一张图来更好的解释VSS,RSS,PSS,USS之间的区别：



有了对VSS,RSS,PSS,USS的了解，我们趁热打铁来看下内存在进程中是如何被瓜分的：



（此图来源于宋宝华老师）

1044，1045，1054三个进程，每个进程都有一个页表，对应其虚拟地址如何向real memory上去转换。

process 1044的1，2，3都在虚拟地址空间，所以其VSS=1+2+3。

process 1044的4，5，6都在real memory上，所以其RSS=4+5+6。

分析real memory的具体瓜分情况：

4 libc代码段，1044，1045，1054三个进程都使用了libc的代码段，被三个进程分享。

5 bash shell的代码段，1044，1045都是bash shell，被两个进程分享。

6 1044独占

所以，上图中4+5+6并不全是1044进程消耗的内存，因为4明显被3个进程指向，5明显被2个进程指向，衍生出了PSS（按比例计算的驻留内存）的概念。进程1044的PSS为4/3 +5/2 +6。

最后，进程1044独占且驻留的内存USS为 6。

一般来说内存占用大小有如下规律：VSS >= RSS >= PSS >= USS

# vmalloc与mmap

mmap()系统调用是在用户进程与内核之间共享内存区域的常用方法。我们最近有个程序，需要应用进程能够读取内核驱动获取的数据，经过简单的调研，决定采用mmap方式。实现起来不难，在驱动中注册一个字符设备，实现该设备的mmap()方法即可。但这其中有一点小曲折。

在实现设备的mmap()方法时，需要将物理内存映射到应用程序通过mmap()系统调用传下来的vma中。vma代表的是进程的一段虚拟地址空间。在第一版里，考虑的不全面，利用alloc\_pages()将整个内存段申请为一段连续的物理地址空间。然后通过remap\_pfn\_range()函数将这段连续的物理内存映射到vma中。经过长时间的测试，没有发现问题。直到今天，在部署一个老集群时，遇到了问题。这个集群中有很多老机器，内存只有十多个G，而且长时间运行后产生了大量的内存碎片。从而导致，我们无法获得足够的连续物理内存。没办法，只好重新调整驱动中分配内存的方式，改用vmalloc获取地址空间。

在kernel里，通常有3种申请内存的方式：vmalloc, kmalloc, alloc\_pages。kmalloc与alloc\_pages类似，均是申请连续的地址空间。而vmalloc则可以申请一段不连续的物理地址空间，并将其映射到连续的线性地址上。每次vmalloc之后，内核会创建一个**vm\_struct**，用以映射分配到的不连续的内存区域。vm\_struct类似**vma**，但是又不是一回事。vma是将物理内存映射到进程的虚拟地址空间。而vm\_struct是将物理内存映射到内核的线性地址空间。

既然vmalloc拿到的不是连续的物理内存，那么将这些内存映射到vma时，就不能直接利用remap\_pfn\_range()了。

此时可以采用两种方法，一种是实现vm\_operations\_struct的fault()方法，用以在缺页时再映射需要的页。此方法操作起来较为麻烦。

另一种方法是直接使用remap\_vmalloc\_range()函数。该函数的原型为：

int remap\_vmalloc\_range(struct vm\_area\_struct \*vma, void \*addr, unsigned long pgoff)

其中参数vma是mmap使用调用传下来的，addr即为vmalloc()所分配内存的起始地址。而pgoff则为mmap()系统调用里的偏移参数，可以通过vma->vm\_pgoff获得。该函数成功执行后，返回值为0。如果返回值为负数，则说明出错了。通常是由于所传的参数不正确。

需要注意的是，需要映射到用户空间的内存段，不能直接利用vmalloc()分配，而应该使用**vmalloc\_user()**函数。该函数除了分配内存之外，还会将相应的vm\_struct结构标记为VM\_USERMAP。否则，remap\_vmalloc\_range将返回错误。

在这个项目中碰到的教训是，永远不要假设系统中一定会有超过一个页的连续物理内存。

不过较新的内核具有compact机制，可以整理内存碎片。但是，目前至少有一大部分机器不支持，或未开启此机制。

# linux内存映射mmap原理分析

一直都对内存映射文件这个概念很模糊，不知道它和虚拟内存有什么区别，而且映射这个词也很让人迷茫，今天终于搞清楚了。。。下面，我先解释一下我对映射这个词的理解，再区分一下几个容易混淆的概念，之后，什么是内存映射就很明朗了。

## 原理

首先，“映射”这个词，就和数学课上说的“一一映射”是一个意思，就是建立一种一一对应关系，在这里主要是只 硬盘上文件 的位置与进程 逻辑地址空间 中一块大小相同的区域之间的一一对应，如图1中过程1所示。这种对应关系纯属是逻辑上的概念，物理上是不存在的，原因是进程的逻辑地址空间本身就是不存在的。在内存映射的过程中，并没有实际的数据拷贝，文件没有被载入内存，只是逻辑上被放入了内存，具体到代码，就是建立并初始化了相关的数据结构（struct address\_space），这个过程有系统调用mmap()实现，所以建立内存映射的效率很高。

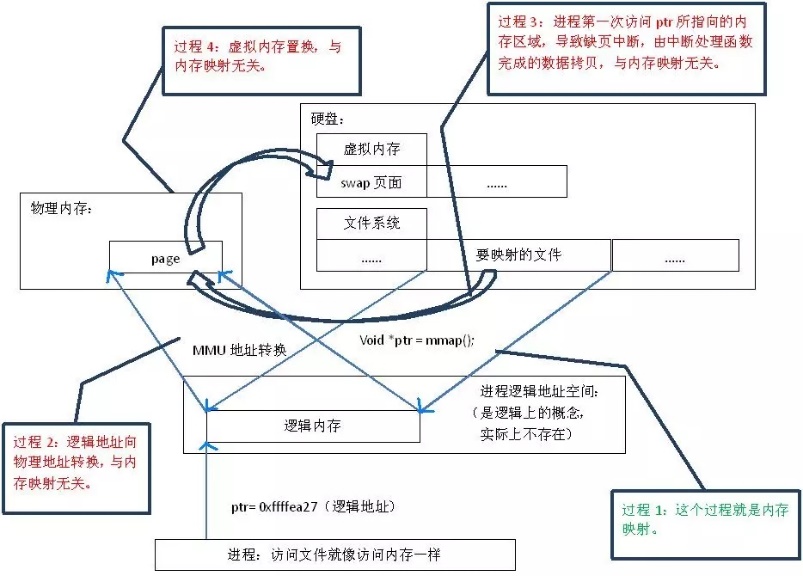


图1.内存映射原理

既然建立内存映射没有进行实际的数据拷贝，那么进程又怎么能最终直接通过内存操作访问到硬盘上的文件呢？那就要看内存映射之后的几个相关的过程了。

mmap()会返回一个指针ptr，它指向进程逻辑地址空间中的一个地址，这样以后，进程无需再调用read或write对文件进行读写，而只需要通过ptr就能够操作文件。但是ptr所指向的是一个逻辑地址，要操作其中的数据，必须通过MMU将逻辑地址转换成物理地址，如图1中过程2所示。这个过程与内存映射无关。

前面讲过，建立内存映射并没有实际拷贝数据，这时，MMU在地址映射表中是无法找到与ptr相对应的物理地址的，也就是MMU失败，将产生一个缺页中断，缺页中断的中断响应函数会在swap中寻找相对应的页面，如果找不到（也就是该文件从来没有被读入内存的情况），则会通过mmap()建立的映射关系，从硬盘上将文件读取到物理内存中，如图1中过程3所示。这个过程与内存映射无关。

如果在拷贝数据时，发现物理内存不够用，则会通过虚拟内存机制（swap）将暂时不用的物理页面交换到硬盘上，如图1中过程4所示。这个过程也与内存映射无关。

## 效率

从代码层面上看，从硬盘上将文件读入内存，都要经过文件系统进行数据拷贝，并且数据拷贝操作是由文件系统和硬件驱动实现的，理论上来说，拷贝数据的效率是一样的。但是通过内存映射的方法访问硬盘上的文件，效率要比read和write系统调用高，这是为什么呢？原因是read()是系统调用，其中进行了数据拷贝，它首先将文件内容从硬盘拷贝到内核空间的一个缓冲区，如图2中过程1，然后再将这些数据拷贝到用户空间，如图2中过程2，在这个过程中，实际上完成了两次数据拷贝；而mmap()也是系统调用，如前所述，mmap()中没有进行数据拷贝，真正的数据拷贝是在缺页中断处理时进行的，由于mmap()将文件直接映射到用户空间，所以中断处理函数根据这个映射关系，直接将文件从硬盘拷贝到用户空间，只进行了一次数据拷贝。因此，内存映射的效率要比read/write效率高。

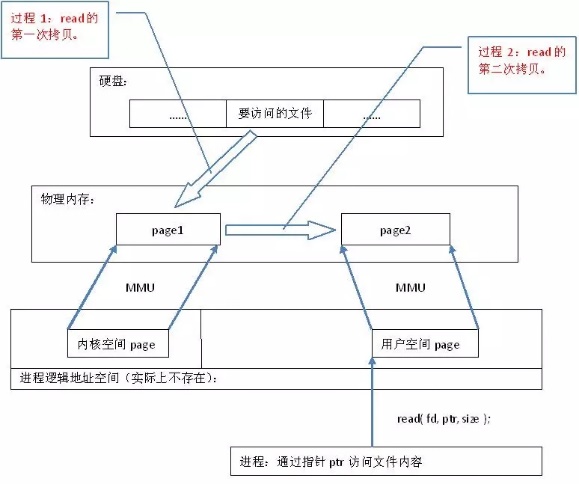


图2.read系统调用原理

下面这个程序，通过read和mmap两种方法分别对硬盘上一个名为“mmap\_test”的文件进行操作，文件中存有10000个整数，程序两次使用不同的方法将它们读出，加1，再写回硬盘。通过对比可以看出，read消耗的时间将近是mmap的两到三倍。

|  |
| --- |
| #include<unistd.h>  #include<stdio.h>  #include<stdlib.h>  #include<string.h>  #include<sys/types.h>  #include<sys/stat.h>  #include<sys/time.h>  #include<fcntl.h>  #include<sys/mman.h>    #define MAX 10000    int main()  {  int i=0;  int count=0, fd=0;  struct timeval tv1, tv2;  int \*array = (int \*)malloc( sizeof(int)\*MAX );    /\*read\*/  gettimeofday( &tv1, NULL );  fd = open( "mmap\_test", O\_RDWR );  if( sizeof(int)\*MAX != read( fd, (void \*)array, sizeof(int)\*MAX ) )  {  printf( "Reading data failed.../n" );  return -1;  }    for( i=0; i<MAX; ++i )  ++array[ i ];    if( sizeof(int)\*MAX != write( fd, (void \*)array, sizeof(int)\*MAX ) )  {  printf( "Writing data failed.../n" );  return -1;  }  free( array );  close(fd);  gettimeofday( &tv2, NULL );  printf( "Time of read/write: %dms/n", tv2.tv\_usec-tv1.tv\_usec );    /\*mmap\*/  gettimeofday( &tv1, NULL );  fd = open( "mmap\_test", O\_RDWR );  array = mmap( NULL, sizeof(int)\*MAX, PROT\_READ|PROT\_WRITE, MAP\_SHARED, fd, 0 );  for( i=0; i<MAX; ++i )  ++array[ i ];  munmap( array, sizeof(int)\*MAX );    msync( array, sizeof(int)\*MAX, MS\_SYNC );  free( array );  close( fd );  gettimeofday( &tv2, NULL );  printf( "Time of mmap: %dms/n", tv2.tv\_usec-tv1.tv\_usec );  return 0;  } |

输出结果：

Time of read/write: 154ms

Time of mmap: 68ms