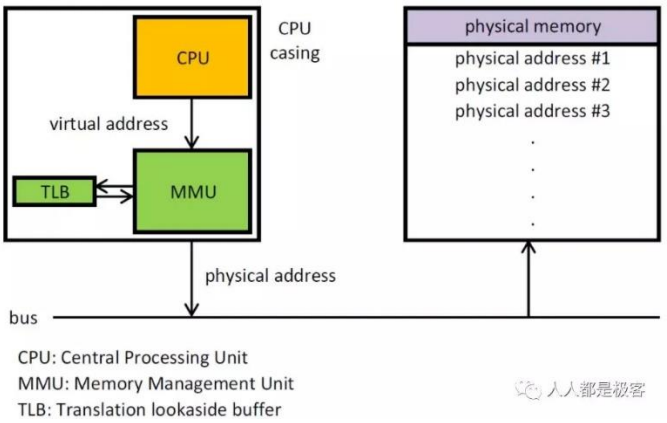
1 CPU 是如何访问内存的？

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。

1.1 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）。

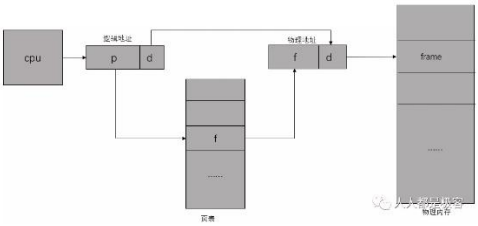
所以搞懂了 MMU 如何把虚拟地址转化为物理地址也就明白了 CPU 是如何通过 MMU 来访问内存的。

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）：

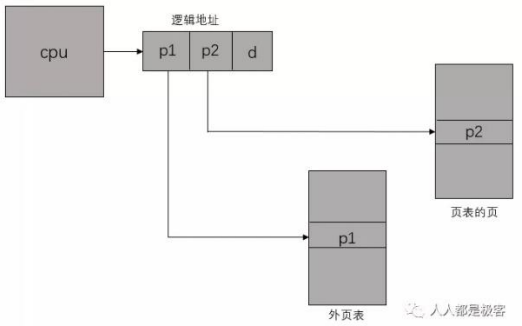
CPU，虚拟地址，页表和物理地址的关系如下图：

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。



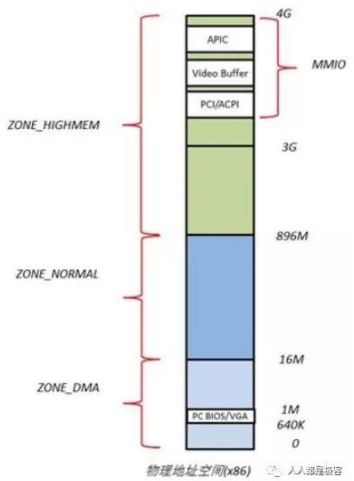
2 物理地址和虚拟地址的分布

上一节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。

2.1 物理地址空间布局

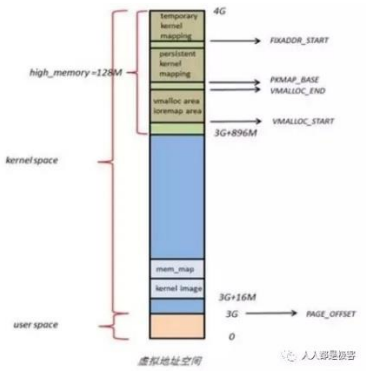
Linux 系统在初始化时，会根据实际的物理内存的大小，为每个物理页面创建一个 page 对象，所有的 page 对象构成一个 mem_map 数组。进一步，针对不同的用途，Linux 内核将所有的物理页面划分到 3 类内存管理区中，如图，分别为 ZONE_DMA，ZONE_NORMAL，ZONE_HIGHMEM。



- 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。

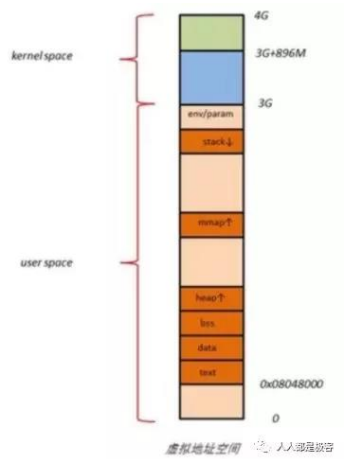
2.2 Linux 内核空间虚拟地址分布



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。

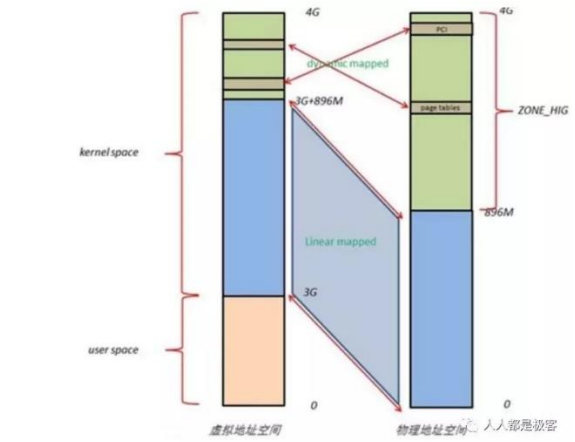
由于 ZONE_NORMAL 和内核线性空间存在直接映射关系，所以内核会将频繁使用的数据如 Kernel 代码、GDT、IDT、PGD、mem_map 数组等放在 ZONE_NORMAL 里。而将用户数据、页表（PT）等不常用数据放在 ZONE_HIGHMEM 里，只在要访问这些数据时才建立映射关系（kmap()）。比如，当内核要访问 I/O 设备存储空间时，就使用 ioremap() 将位于物理地址高端的 mmio 区内存映射到内核空间的 vmalloc area 中，在使用完之后便断开映射关系。

2.3 Linux 用户空间虚拟地址分布



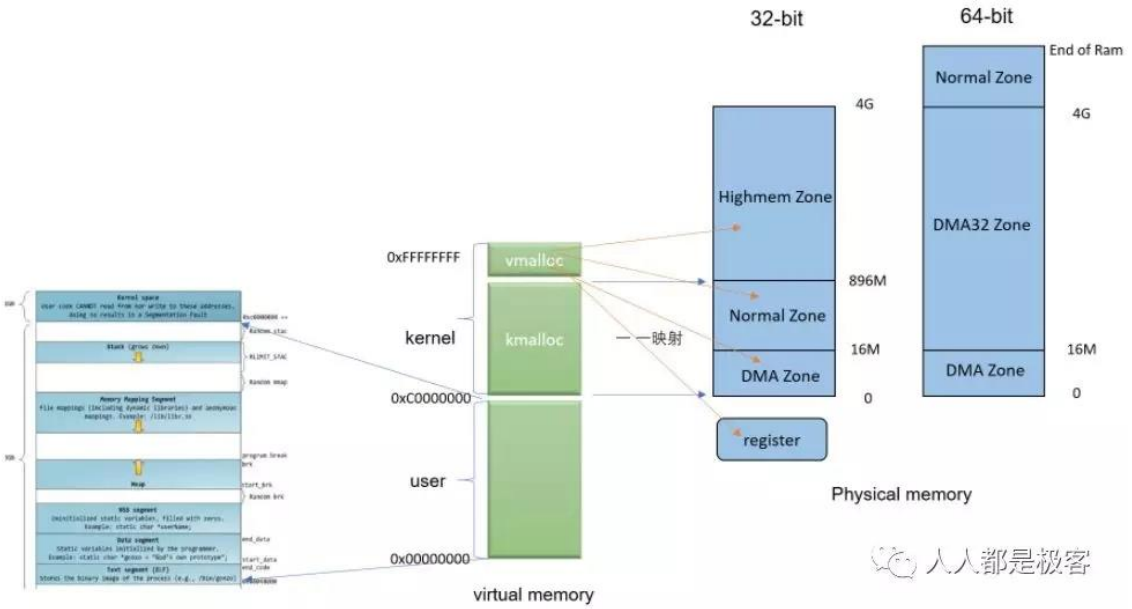
用户进程的代码区一般从虚拟地址空间的 0x08048000 开始，这是为了便于检查空指针。代码区之上便是数据区，未初始化数据区，堆区，栈区，以及参数、全局环境变量。

2.4 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。

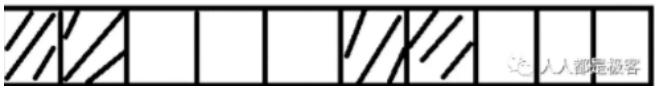
到这里我们应该知道了 Linux 是如何用虚拟地址来映射物理地址的，最后我们用一张图来总结一下：



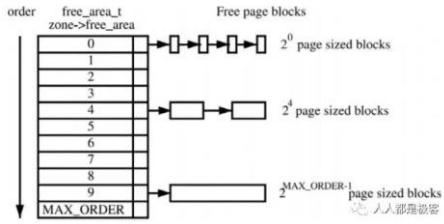
3 Linux 内核内存管理算法 Buddy 和 Slab

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。

3.1 Buddy 分配算法



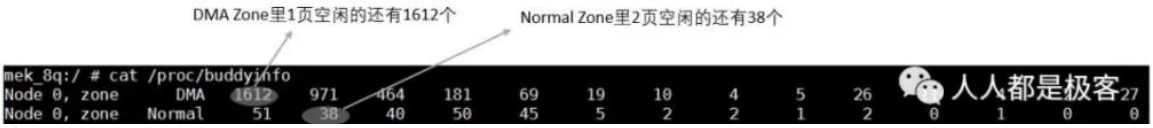
假设这是一段连续的页框，阴影部分表示已经被使用的页框，现在需要申请一个连续的 5 个页框。这个时候，在这段内存上不能找到连续的 5 个空闲的页框，就会去另一段内存上去寻找 5 个连续的页框，这样子，久而久之就形成了页框的浪费。为了避免出现这种情况，Linux 内核中引入了伙伴系统算法(Buddy system)。把所有的空闲页框分组为 11 个块链表，每个块链表分别包含大小为 1，2，4，8，16，32，64，128，256，512 和 1024 个连续页框的页框块。最大可以申请 1024 个连续页框，对应 4MB 大小的连续内存。每个页框块的第一个页框的物理地址是该块大小的整数倍，如图：



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：

```
mek_8q:/ # echo m > /proc/sysrq-trigger
[ 6473.020026] sysrq: SysRq : Show Memory
[ 6473.023876] Mem-Info:
[ 6473.026163] active_anon:167383 inactive_anon:325 isolated_anon:0
[ 6473.026163]  active_file:63004 inactive_file:126125 isolated_file:0
[ 6473.026163]  unevictable:64 dirty:0 writeback:0 unstable:0
[ 6473.026163]  slab_reclaimable:7560 slab_unreclaimable:11013
[ 6473.026163]  mapped:78779 shmem:409 pagetables:3766 bounce:0
[ 6473.026163]  free:279472 free_pcp:392 free_cma:115749
[ 6473.061186] Node 0 active_anon:669532kB inactive_anon:1300kB active_file:252016kB inactive_file:504500kB unevictable:256kB isolated(anon):0kB isolated(file):0kB mapped:315116kB dirty:0kB writeback:0kB shmem:1636kB shmem_thp: 0kB shmem_pmdmapped: 0kB anon_thp: 262144kB writeback_tmp:0kB unstable:0kB pages_scanned:0 all_unreclaimable? no
[ 6473.090984] DMA free:1098324kB min:29392kB low:36740kB high:44088kB active_anon:363380kB inactive_anon:1036kB active_file:53524kB inactive_file:123388kB unevictable:0kB writepending:0kB present:2058240kB managed:1865676kB mlocked:0kB slab_reclaimable:5432kB slab_unreclaimable:12540kB kernel_stack:4224kB pagetables:7980kB bounce:0kB free_pcp:348kB local_pcp:12kB free_cma:462996kB
[ 6473.124581] lowmem_reserve[]: 0 968 968
[ 6473.128491] Normal free:19564kB min:15660kB low:19572kB high:23484kB active_anon:306152kB inactive_anon:264kB active_file:198492kB inactive_file:381112kB unevictable:256kB writepending:0kB present:1048576kB managed:992088kB mlocked:256kB slab_reclaimable:24888kB slab_unreclaimable:31512kB kernel_stack:6336kB pagetables:7084kB bounce:0kB free_pcp:1220kB local_pcp:112kB free_cma:0kB
[ 6473.162263] lowmem_reserve[]: 0 0 0
[ 6473.165799] DMA: 1519*4kB (UMEC) 971*8kB (UMEC) 464*16kB (UMEC) 180*32kB (UMEC) 69*64kB (UMEC) 19*128kB (UMC) 10*256kB (UMEC) 4*512kB (UMC) 5*1024kB (UC) 29*2048kB (UMEC) 11*4096kB (UMC) 4*8192kB (UME) 2*16384kB (EC) 27*32768kB (MC) = 1098324kB
[ 6473.187751] Normal: 43*4kB (UE) 46*8kB (UE) 69*16kB (UME) 50*32kB (UME) 45*64kB (UME) 5*128kB (UME) 2*256kB (UE) 2*512kB (UM) 1*1024kB (E) 1*2048kB (E) 0*4096kB 1*8192kB (M) 0*16384kB 0*32768kB = 19564kB
[ 6473.206149] Node 0 hugepages_total=0 hugepages_free=0 hugepages_surp=0 hugepages_size=2048kB
[ 6473.214613] 189534 total pagecache pages
[ 6473.218548] 0 pages in swap cache
[ 6473.221879] Swap cache stats: add 0, delete 0, find 0/0
[ 6473.227117] Free swap  = 0kB
[ 6473.230004] Total swap = 0kB
[ 6473.232895] 776704 pages RAM
[ 6473.235814] 0 pages HighMem/MovableOnly
[ 6473.239643] 62265 pages reserved
[ 6473.242887] 163840 pages cma reserved
```

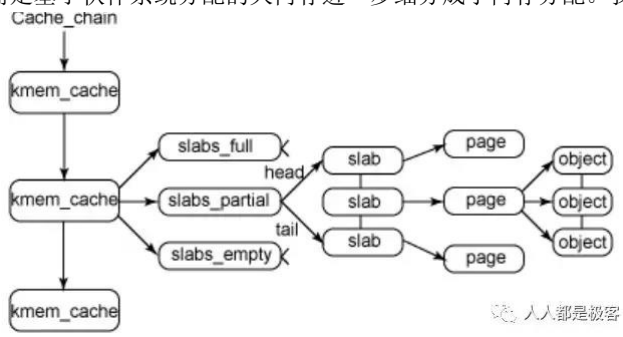

3.2 CMA

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。

3.3 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：

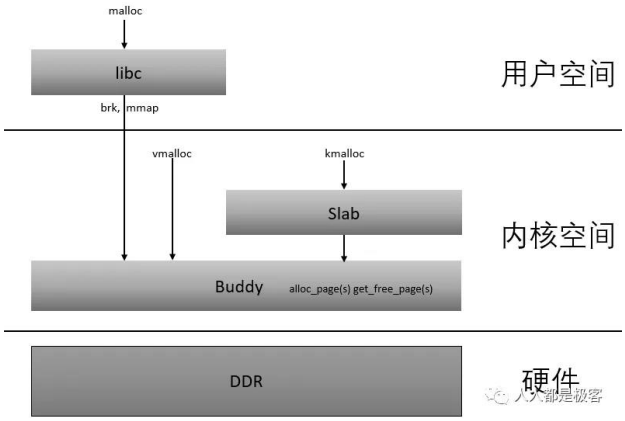
```
root@imx8qxpmek:~# cat /proc/slabinfo
slabinfo - version: 2.1
# name            <active_objs> <num_objs> <objsize> <objperslab> <pagesperslab> : tunables <limit> <batchcount> <sharedfacto
>
ip6-frags              0      0    200   20    1 : tunables    0    0    0 : slabdata      0      0      0
UDPv6                120    120   1088   30    8 : tunables    0    0    0 : slabdata      4      4      0
tw_sock_TCPv6          0      0    240   17    1 : tunables    0    0    0 : slabdata      0      0      0
request_sock_TCPv6     0      0    304   26    2 : tunables    0    0    0 : slabdata      0      0      0
TCPv6                 48     48   2048   16    8 : tunables    0    0    0 : slabdata      3      3      0
ext4_groupinfo_4k     28     28    144   28    1 : tunables    0    0    0 : slabdata      1      1      0
ubifs_inode_slab       0      0    720   22    4 : tunables    0    0    0 : slabdata      0      0      0
can_gw                 0      0    568   28    4 : tunables    0    0    0 : slabdata      0      0      0
isp1760_qtd            0      0     72   56    1 : tunables    0    0    0 : slabdata      0      0      0
isp1760_urb_listitem      0      0     24  170    1 : tunables    0    0    0 : slabdata      0      0      0
bsg_cmd                0      0    312   26    2 : tunables    0    0    0 : slabdata      0      0      0
mqueue_inode_cache    18     18    896   18    4 : tunables    0    0    0 : slabdata      1      1      0
```

○ ○ ○ ○ ○ ○

```
dma-kmalloc-8192       0      0   8192    4    8 : tunables    0    0    0 : slabdata      0      0      0
dma-kmalloc-4096       0      0   4096    8    8 : tunables    0    0    0 : slabdata      0      0      0
dma-kmalloc-2048       0      0   2048   16    8 : tunables    0    0    0 : slabdata      0      0      0
dma-kmalloc-1024       0      0   1024   16    4 : tunables    0    0    0 : slabdata      0      0      0
dma-kmalloc-512        0      0    512   16    2 : tunables    0    0    0 : slabdata      0      0      0
dma-kmalloc-256        0      0    256   16    1 : tunables    0    0    0 : slabdata      0      0      0
dma-kmalloc-128        0      0    128   32    1 : tunables    0    0    0 : slabdata      0      0      0
kmalloc-8192          44     44   8192    4    8 : tunables    0    0    0 : slabdata [illegible] [illegible]      0
kmalloc-4096         166    176   4096    8    8 : tunables    0    0    0 : slabdata [illegible] [illegible]      0
kmalloc-2048        1904   1904   2048   16    8 : tunables    0    0    0 : slabdata    119    119      0
kmalloc-1024        1341   1360   1024   16    4 : tunables    0    0    0 : slabdata     85     85      0
```


3.4 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：



4 Linux 用户态进程的内存管理

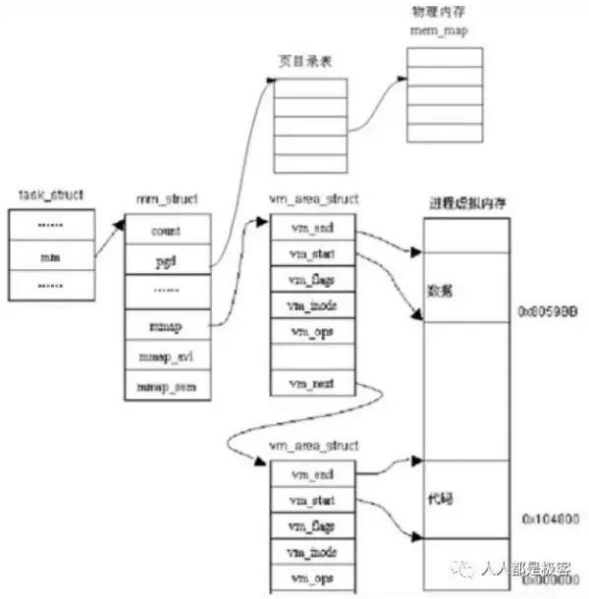
我们了解了内存在内核态是如何管理的，本篇文章我们一起来看下内存在用户态的使用情况，如果上一篇文章说是内核驱动工程师经常面对的内存管理问题，那本篇就是应用工程师常面对的问题。

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

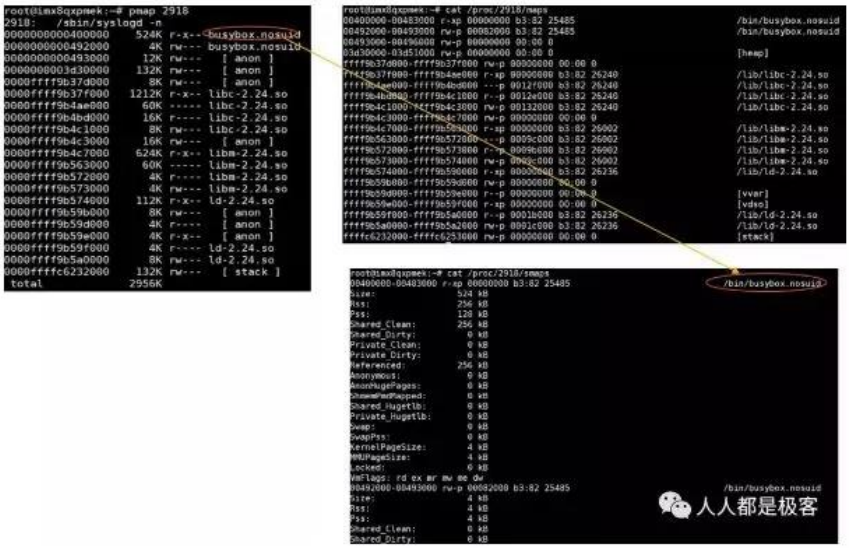
4.1 进程的虚拟地址空间 VMA（Virtual Memory Area）

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 ：

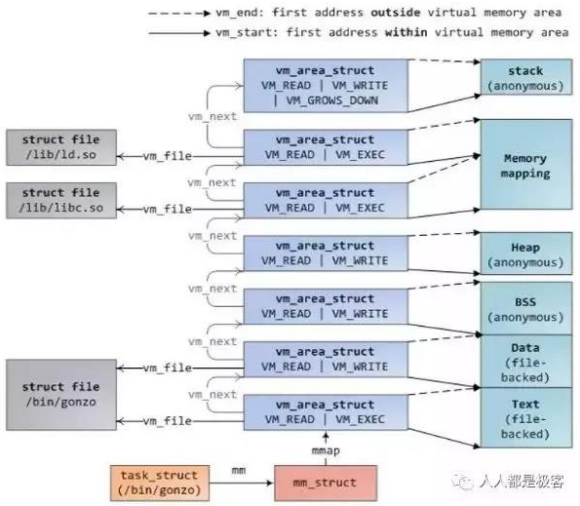


上图中，task_struct 中的 mm_struct 就代表进程的整个内存资源，mm_struct 中的 pgd 为页表，mmap 指针指向的 vm_area_struct 链表的每一个节点就代表进程的一个虚拟地址空间，即一个 VMA。一个 VMA 最终可能对应 ELF 可执行程序的数据段、代码段、堆、栈、或者动态链接库的某个部分。

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。



4.2 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。

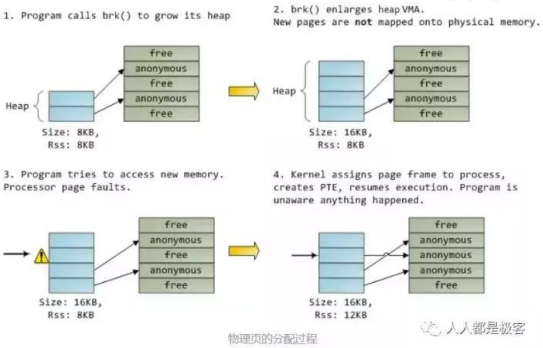
缺页	
major主缺页	minior次缺页
发生缺页必须读硬盘 如：上4） 有IO行为，处理时间长	发生缺页直接申请到内存 如：malloc内存，写时发生 无IO行为，处理时间相对major短

（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。

4.3 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）

4.4 内存的消耗 VSS RSS PSS USS

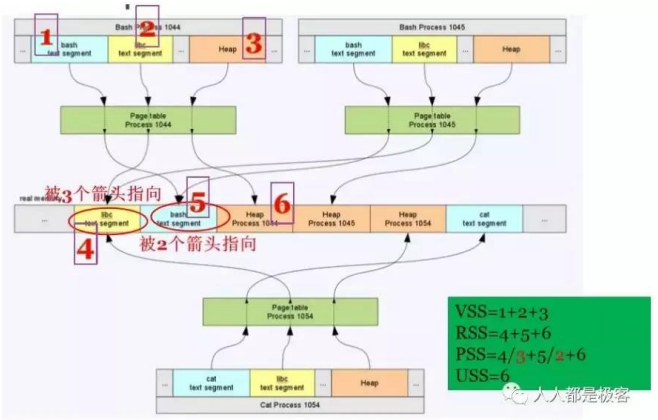
首先，我们评估一个进程的内存消耗都是指用户空间的内存，不包括内核空间的内存消耗 。这里我们用工具 procrank 先来看下 Linux 进程的内存占用量 。

```
# procrank
procrank
PID    Vss      Rss      Pss      Uss cmdline
481  31536K   30936K   14337K    9956K system_server
475  26128K   26128K   10046K    5992K zygote
526  25108K   25108K    9225K    5384K android.process.acore
523  22388K   22388K    7166K    3432K com.android.phone
574  21632K   21632K    6109K    2468K com.android.settings
521  20816K   20816K    6050K    2776K jp.co.omronsoft.openwnn
474   3304K    3304K    1097K     624K /system/bin/mediaserver
37    304K     304K     289K     288K /sbin/adbd
29    720K     720K     261K     212K /system/bin/rild
601    412K     412K     225K     216K procrank
  1    204K     204K     185K     184K /init
35    388K     388K     182K     172K /system/bin/qemud
284   384K     384K     160K     148K top
27    376K     376K     148K     136K /system/bin/vold
261   332K     332K     123K     112K logcat
33    396K     396K     105K      80K /system/bin/keystore
32    316K     316K     100K      88K /system/bin/installd
269   328K     328K      95K      72K /system/bin/sh
26    280K     280K      93K      84K /system/bin/servicemanager
45    304K     304K      91K      80K /system/bin/qemu-props
34    324K     324K      91K      68K /system/bin/sh
260   324K     324K      91K      68K /system/bin/sh
600   324K     324K      91K      68K /system/bin/sh
25    308K     308K      88K      68K /system/bin/sh
28    232K     232K      67K      60K /system/bin/debuggerd
```

- 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