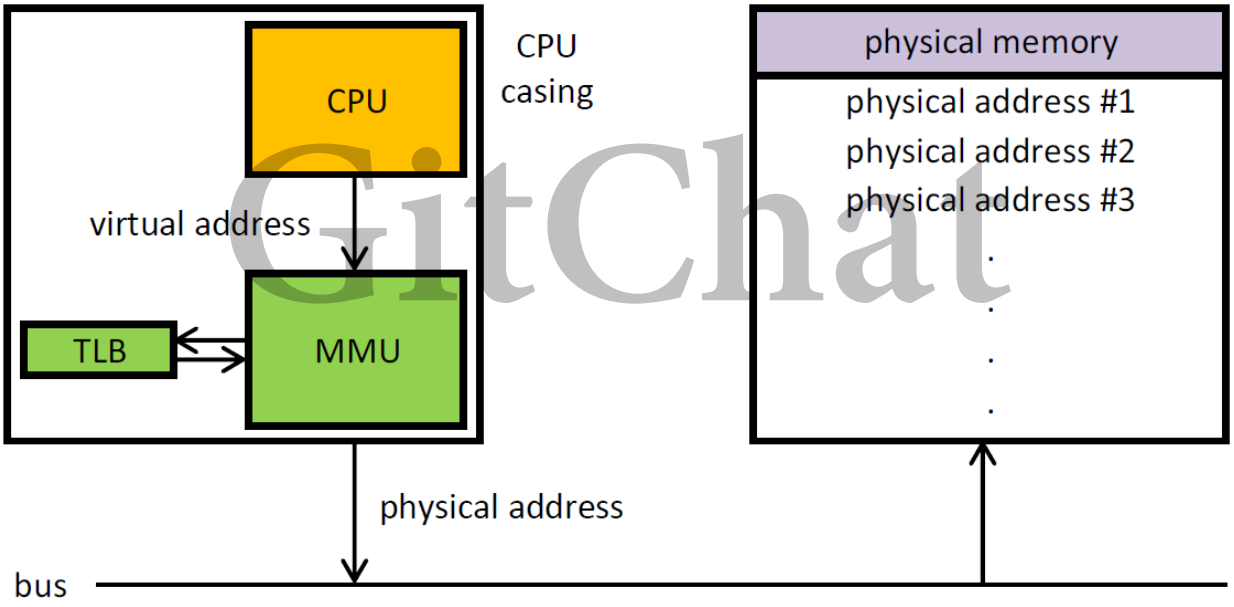
Linux 内存管理之内核态剖析

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。本场 Chat 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：



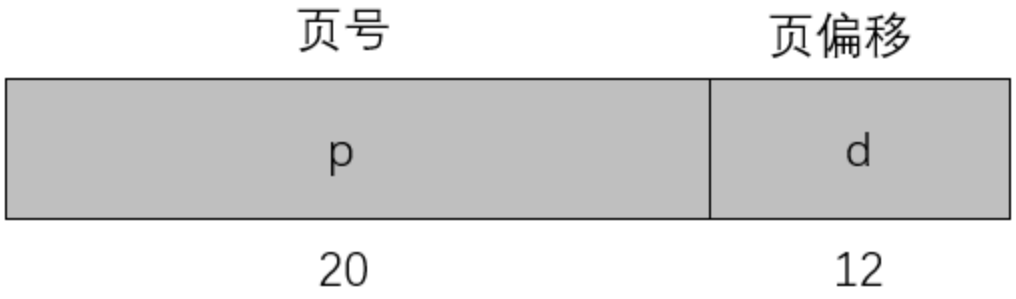
CPU: Central Processing Unit
MMU: Memory Management Unit
TLB: Translation lookaside buffer

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）：



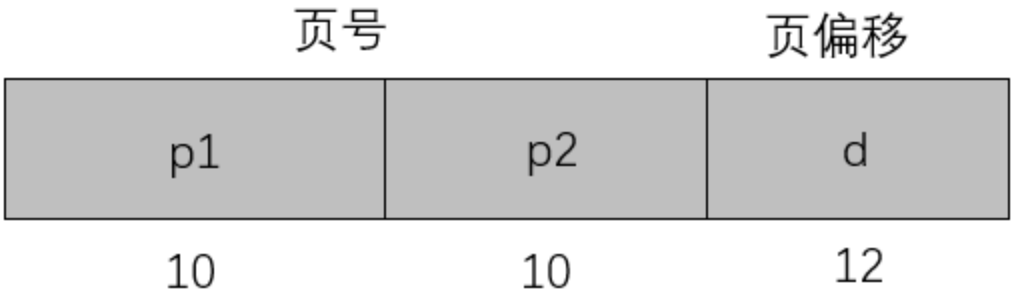
	物理地址	命中否？	RWX 权限	User/Kernel 权限
第0行		否		
第1行	6MB	是	RX	U+K
第2行	8MB	是	RX	U+K
第3行	3MB	是	RW	U+K
	... 此处省略一万字 ...			
第3GB/4KB行	0MB	是	RX	K
第（3GB+4KB）/4KB行	0MB+4KB	是	RX	K

- 当 CPU 访问虚拟地址0的时候，MMU 会去查上面页表的第0行，发现第0行没有命中，于是无论以何种形式访问（R 读，W 写，X 执行），MMU 都会给 CPU 发出 page fault，然后 CPU 自动跳到 fault 的代码去处理 fault。
- 当 CPU 访问虚拟地址 4KB 的时候，MMU 会去查上面页表的第1行（4KB/4KB=1），发现第1行命中，如果这个时候：
  - a）用户是执行读或者执行，则 MMU 去访问内存条的 6MB 这个地址，因为页表里面记录该页的权限是 RX；
  - b）用户是去写 4KB，由于页表里面第1行记录的权限是 RX，没有记录你有写的权限，MMU 会给 CPU 发出 page fault，CPU 自动跳到 fault 的代码去处理 fault。

- 当 CPU 访问虚拟地址 8KB+16 的时候，MMU 会去查上面页表的第2行(8KB/4KB=2)，发现第2行命中了物理地址 8M，如果这个时候，MMU 会访问内存条的 8MB+16 这个物理地址。当然，权限检查也是需要的。
- 当 CPU 访问虚拟地址 3GB 的时候，MMU 会去查上面页表的第 3GB/4KB 行，表中记录命中了，查到虚拟地址 3GB 对应的物理地址是0，于是 MMU 去访问内存条上的地址0。但是，这个访问分成两种情况：
  - CPU 在执行用户态程序的时候，去访问 3GB，由于页表里面记录的 U+K 权限只有 K，所以 U 是没权限的，MMU 会给 CPU 发出 page fault，CPU 自动跳到 fault 的代码去处理 fault；

- CPU 在执行内核态程序的时候，去访问 3GB，由于页表里面记录的 U+K 权限只有 K，所以 K 是有权限的，MMU 不会给 CPU 发出 page fault，程序正常执行。

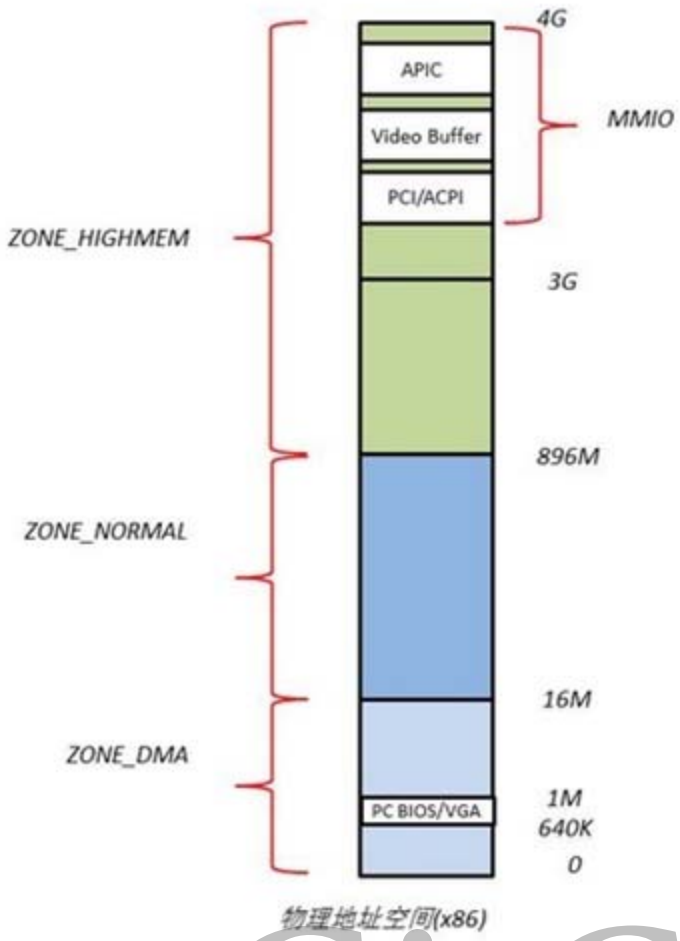
由此可以得知，如果页表只有1级，每 4KB 的虚拟地址空间就需要 32bit 的页表里面的一行，那么 CPU 要覆盖到整个 4GB 的内存，就需要这个页表的大小是：4GB/4KB *4 = 4MB，即每个进程都需要1个 4MB 的页表，这个空间浪费还是很大，于是我们可以采用二级页表。举例如下：



地址的高10位作为一级页表的索引，中间10位作为二级页表的索引。

物理地址空间布局

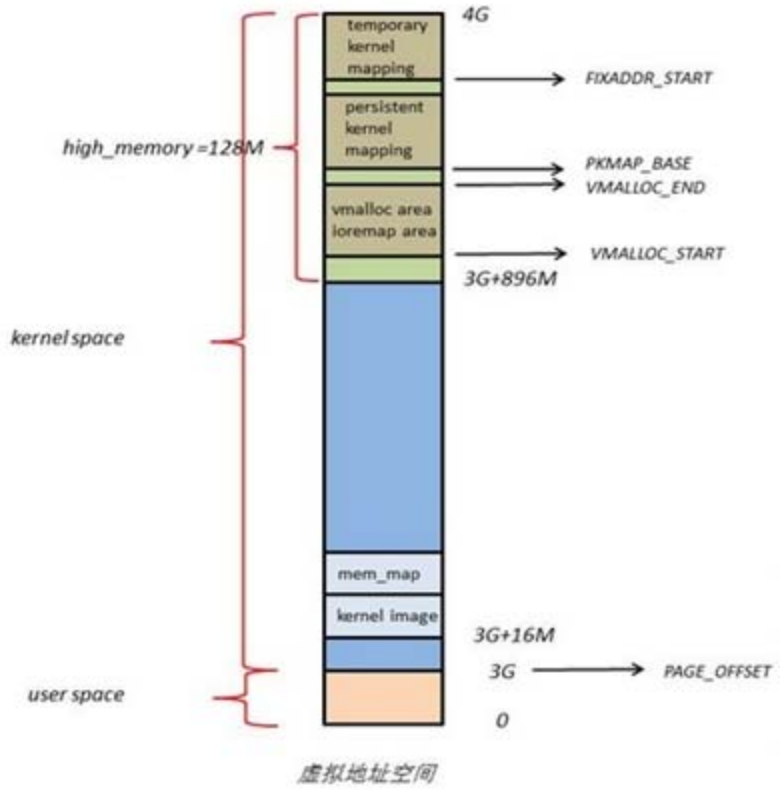
我们先来看一张图：



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三类内存管理区中，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。

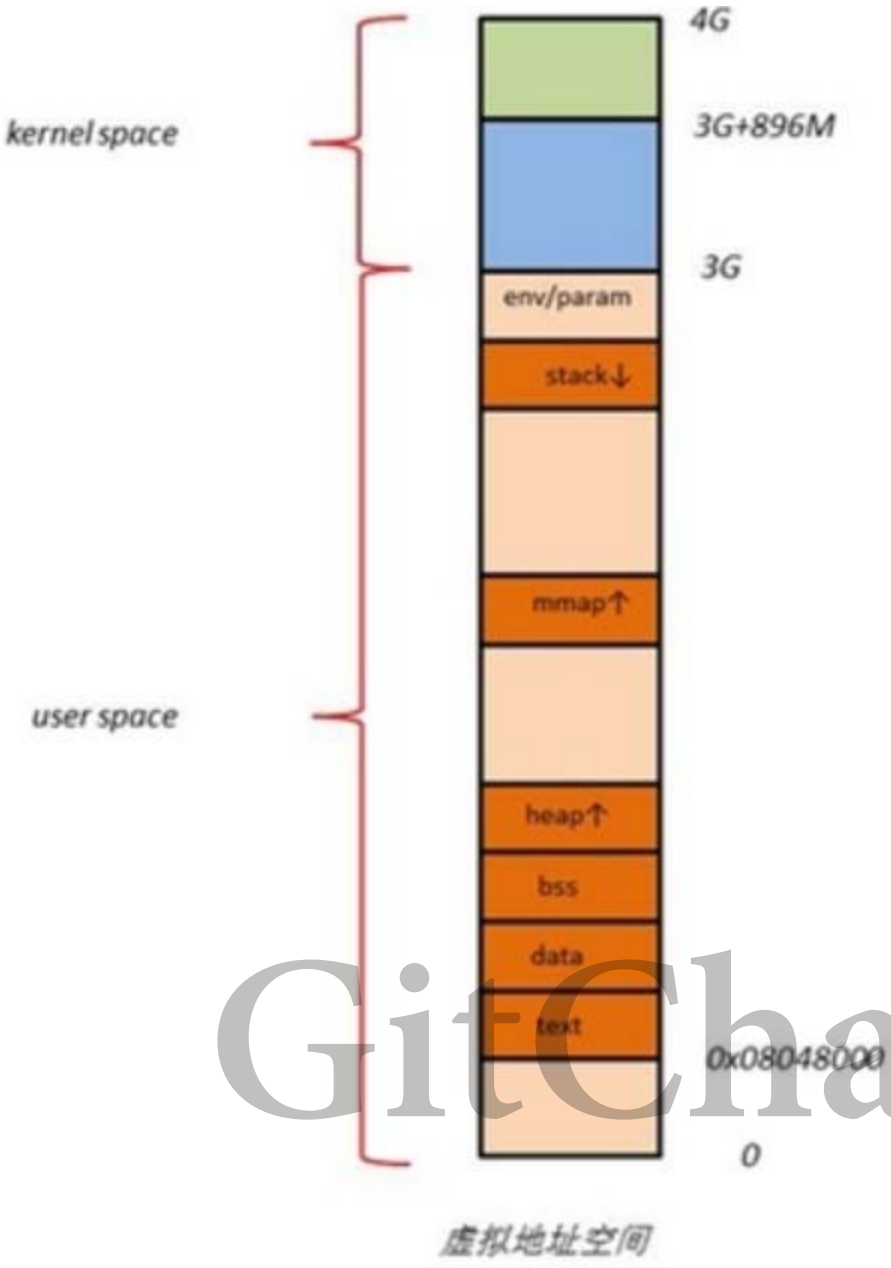
Linux虚拟地址内核空间分布



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。

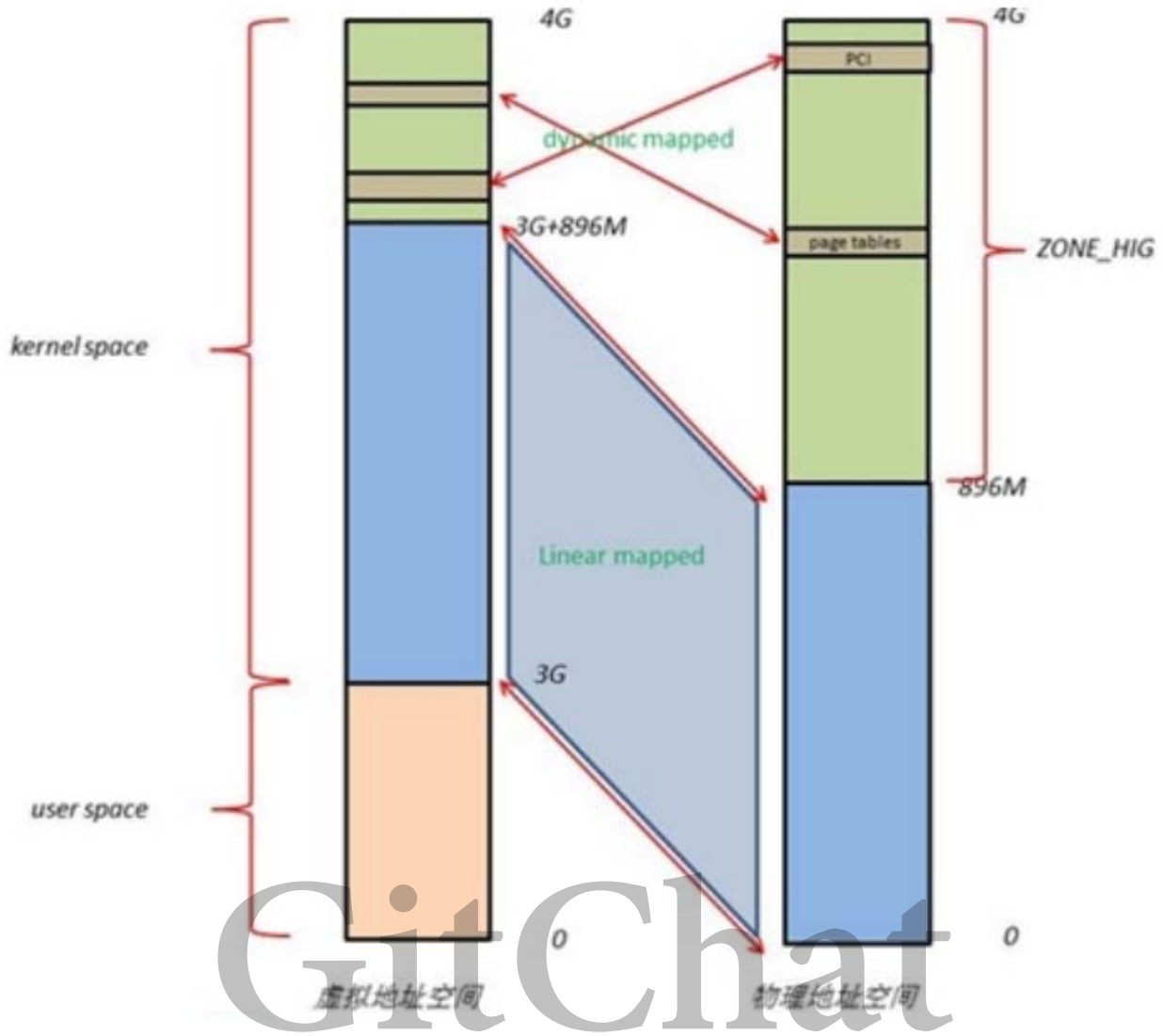
由于 ZONE_NORMAL 和内核线性空间存在直接映射关系，所以内核会将频繁使用的数据如 Kernel 代码、GDT、IDT、PGD、mem_map 数组等放在 ZONE_NORMAL 里。而将用户数据、页表（PT）等不常用数据放在 ZONE_ HIGHMEM 里，只在要访问这些数据时才建立映射关系（kmap()）。比如，当内核要访问 I/O 设备存储空间时，就使用 ioremap() 将位于物理地址高端的 mmio 区内存映射到内核空间的 vmalloc area 中，在使用完之后便断开映射关系。

Linux 虚拟地址用户空间分布



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。

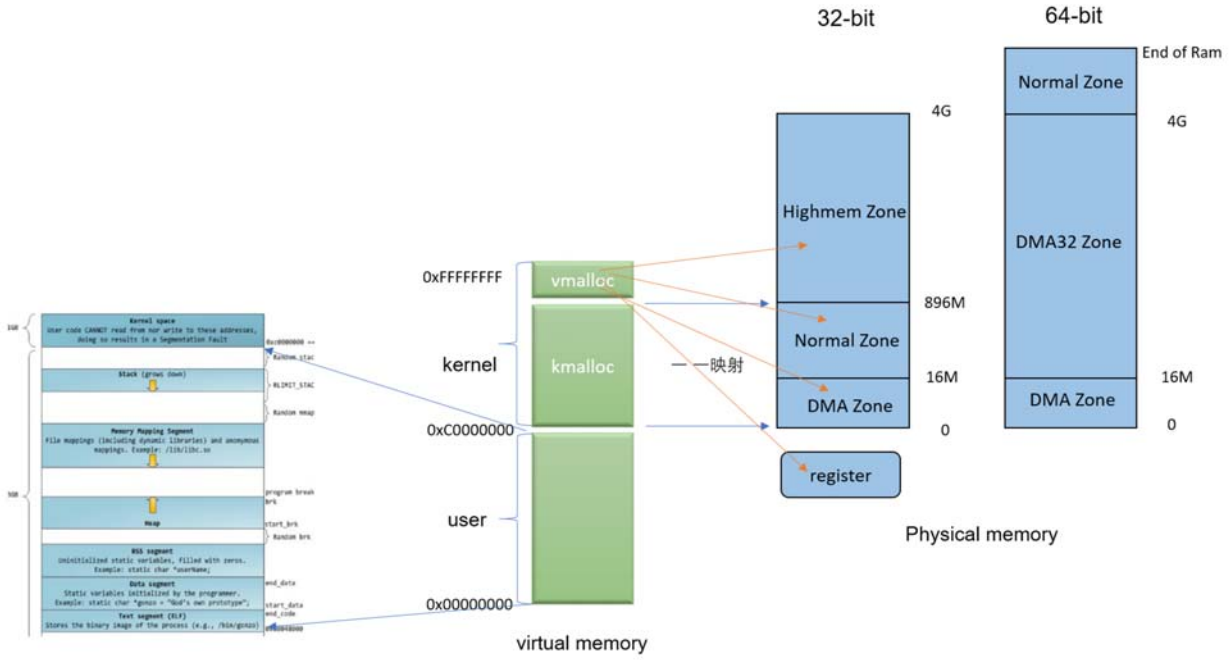
Linux 虚拟地址与物理地址映射的关系



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，下面我们用一张图来总结一下：



防止内外碎片的方法

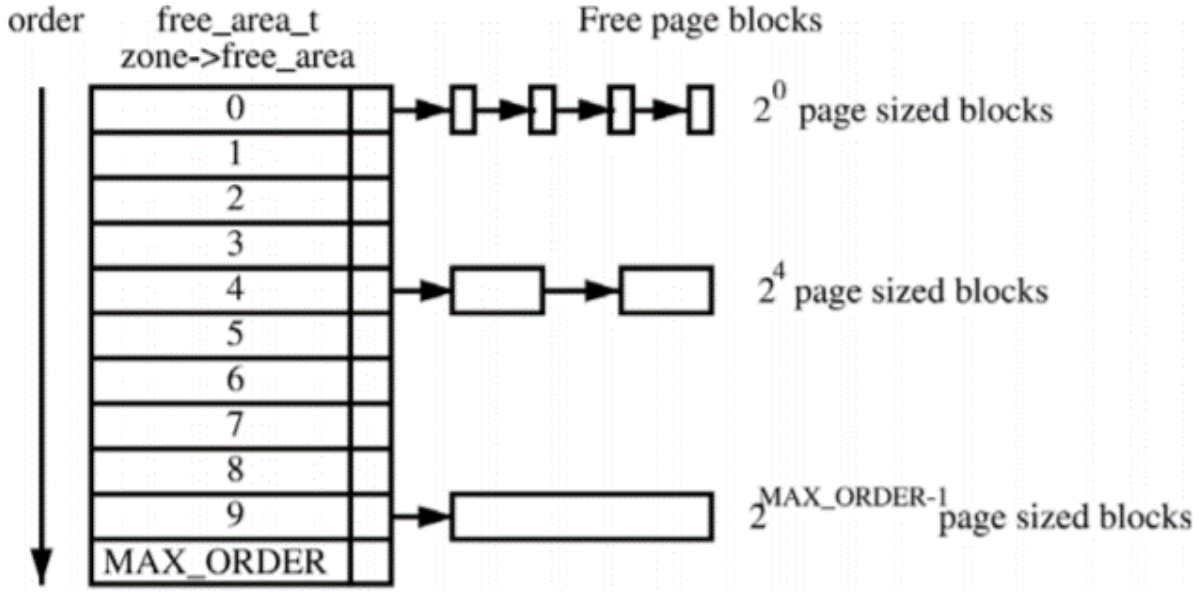
我们先来搞清楚什么是外部碎片。

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。称为**外部碎片**。面对这种外部碎片造成的浪费，内核使用 Buddy 算法来解决。

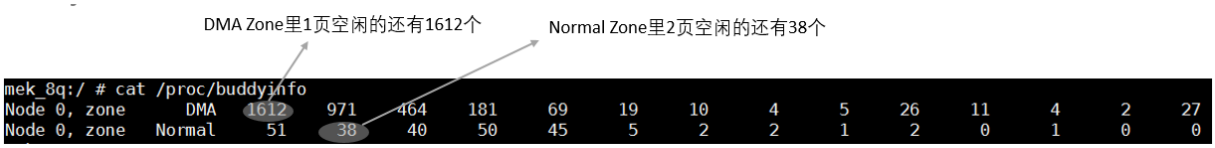
当我们申请几十个字节的时候，内核也是给我们分配一个页，这样在每个页中就形成了很大的浪费。称之为**内部碎片**。面对这种内部碎片造成的浪费，内核使用 Slab 机制来解决。

Buddy 算法

Buddy 把所有的空闲页框分为11个块链表，每块链表中分布包含特定的连续页框地址空间，比如第0个块链表包含大小为2^0个连续的页框，第1个块链表中，每个链表元素包含2个页框大小的连续地址空间，……，第10个块链表中，每个链表元素代表 4M 的连续地址空间。

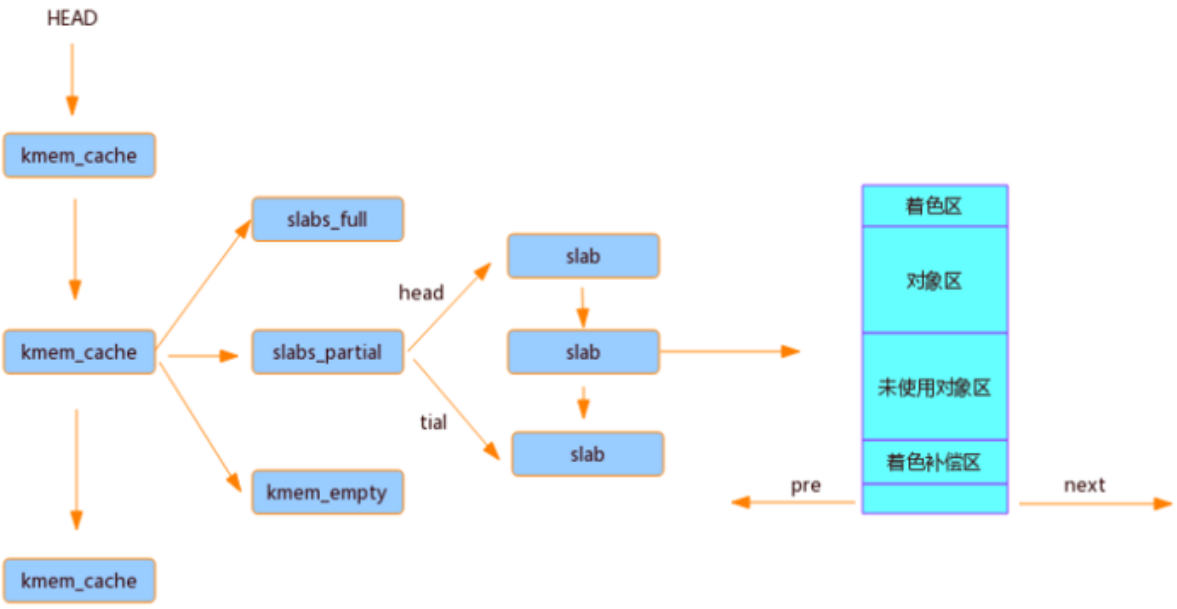


通过以下命令可以看出 Buddy 的信息：



Slab 机制

先来看一下 Slab 分配器的主要结构：



每个缓存都包含了一个 slabs 列表，这是一段连续的内存块（通常都是页面）。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）

注意 slabs_empty 列表中的 slab 是进行回收（reaping）的主要备选对象。正是通过此过程，slab 所使用的内存被返回给操作系统供其他用户使用。

slab 列表中的每个 slab 都是一个连续的内存块（一个或多个连续页），它们被划分成一个个对象。这些对象是从特定缓存中进行分配和释放的基本元素。注意 slab 是 slab 分配器进行操作的最小分配单位，因此如果需要对 slab 进行扩展，这也就是所扩展的最小值。通常来说，每个 slab 被分配为多个对象。

由于对象是从 slab 中进行分配和释放的，因此单个 slab 可以在 slab 列表之间进行移动。例如，当一个 slab 中的所有对象都被使用完时，就从 slabs_partial 列表中移动到 slabs_full 列表中。当一个 slab 完全被分配并且有对象被释放后，就从 slabs_full 列表中移动到 slabs_partial 列表中。当所有对象都被释放之后，就从 slabs_partial 列表移动到 slabs_empty 列表中。

可以通过以下命令查看 Slab 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
osg_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
```