内核中的物理内存分配函数kernelapi PDF转换可能丢失图片或格式，建议阅读原文 https://www.100test.com/kao_ti2020/213/2021_2022__E5_86_85_E6_A0_B8_E4_B8_AD_E7_c103_213944.htm 在网上查资料时看到几篇介绍 linux driver 编写的文章，其中提到 kmalloc()与__get_free_page()返回地址的问题，我们都知道 kmalloc() 与__get_free_page() 分配的是物理内存，但它返回的到底是什么？那几篇关于驱动编写的文章中提到申请的是物理地址，返回的依然是物理地址。但有一篇文章中，作者 对此提出了质疑，但没有给出答案。这也就是我写这篇笔记的原因。在找答案的同时也将 linux kernel 分配物理内存的流程做了下分析。这仅是篇笔记，写的比较乱。自己能看懂就行了。 这里只分析分配连续物理地址的函数。对于 vmalloc() 这种分配非连续物理地址的函数不在本记录范围之内。 1、kmalloc() 分配连续的物理地址，用于小内存分配。 2、__get_free_page() 分配连续的物理地址，用于整页分配。 至于为什么说以上函数分配的是连续的物理地址和返回的到底是物理地址还是虚拟地址，下面的记录会做出解释。 kmalloc() 函数本身是基于 slab 实现的。slab 是为分配小内存提供的一种高效机制。但 slab 这种分配机制又不是独立的，它本身也是在页分配器的基础上来划分更细粒度的内存供调用者使用。也就是说系统先用页分配器分配以页为最小单位的连续物理地址，然后 kmalloc() 再在这上面根据调用者的需要进行切分。 关于以上论述，我们可以查看 kmalloc() 的实现，kmalloc()函数的实现是在 __do_kmalloc() 中，可以看到在 __do_kmalloc()代码里最终调用了 __cache_alloc() 来分配一个 slab，其实

kmem_cache_alloc() 等函数的实现也是调用了这个函数来分配新的 slab。我们按照 __cache_alloc()函数的调用路径一直跟踪下去会发现在 cache_grow() 函数中使用了 kmem_getpages()函数来分配一个物理页面，kmem_getpages() 函数中调用的alloc_pages_node() 最终是使用 __alloc_pages() 来返回一个struct page 结构，而这个结构正是系统用来描述物理页面的。这样也就证实了上面所说的，slab 是在物理页面基础上实现的。kmalloc() 分配的是物理地址。__get_free_page() 是页面分配器提供给调用者的最底层的内存分配函数。它分配连续的物理内存。__get_free_page() 函数本身是基于 buddy 实现的。在使用 buddy 实现的物理内存管理中最小分配粒度是以页为单位的。关于以上论述，我们可以查看__get_free_page()的实现，可以看到__get_free_page()函数只是一个非常简单的封状，它的整个函数实现就是无条件的调用 __alloc_pages() 函数来分配物理内存，上面记录 kmalloc()实现时也提到过是在调用 __alloc_pages() 函数来分配物理页面的前提下进行的 slab 管理。那么这个函数是如何分配到物理页面又是在什么区域中进行分配的？回答这个问题只能看下相关的实现。可以看到在 __alloc_pages() 函数中，多次尝试调用get_page_from_freelist() 函数从 zonelist 中取得相关 zone，并从其中返回一个可用的 struct page 页面（这里的有些调用分支是因为标志不同）。至此，可以知道一个物理页面的分配是从 zonelist（一个 zone 的结构数组）中的 zone 返回的。那么 zonelist/zone 是如何与物理页面关联，又是如何初始化的呢？继续来看 free_area_init_nodes() 函数，此函数在系统初始化时由 zone_sizes_init() 函数间接调用的，zone_sizes_init()函数填

充了三个区域：ZONE_DMA，ZONE_NORMAL，ZONE_HIGHMEM。并把他们作为参数调用 free_area_init_nodes()，在这个函数中会分配一个 pglist_data 结构，此结构中包含了 zonelist/zone结构和一个 struct page 的物理页结构，在函数最后用此结构作为参数调用了 free_area_init_node() 函数，在这个函数中首先使用 calculate_node_totalpages() 函数标记 pglist_data 相关区域，然后调用 alloc_node_mem_map() 函数初始化 pglist_data结构中的 struct page 物理页。最后使用 free_area_init_core()函数关联 pglist_data 与 zonelist。现在通以上分析已经明确了__get_free_page() 函数分配物理内存的流程。但这里又引出了几个新问题，那就是此函数分配的物理页面是如何映射的？映射到了什么位置？到这里不得不去看下与 VMM 相关的引导代码。 在看 VMM 相关的引导代码前，先来看一下 virt_to_phys() 与phys_to_virt 这两个函数。顾名思义，即是虚拟地址到物理地址和物理地址到虚拟地址的转换。函数实现十分简单，前者调用了__pa(address) 转换虚拟地址到物理地址，后者调用 __va(addrress) 将物理地址转换为虚拟地址。再看下 __pa __va 这两个宏到底做了什么。 #define __pa(x) ((unsigned long)(x)-PAGE_OFFSET) #define __va(x) ((void *)((unsigned long)(x) PAGE_OFFSET)) 通过上面可以看到仅仅是把地址加上或减去 PAGE_OFFSET，而PAGE_OFFSET 在 x86 下定义为 0xC0000000。这里又引出了疑问，在 linux 下写过 driver 的人都知道，在使用 kmalloc() 与 __get_free_page() 分配完物理地址后，如果想得到正确的物理地址需要使用 virt_to_phys() 进行转换。那么为什么要有这一步呢？我们不

分配的不就是物理地址么？怎么分配完成还需要转换？如果返回的是虚拟地址，那么根据如上对 virt_to_phys() 的分析，为什么仅仅对 PAGE_OFFSET 操作就能实现地址转换呢？虚拟地址与物理地址之间的转换不需要查页表么？代着以上诸多疑问来看 VMM 相关的引导代码。 直接从 start_kernel() 内核引导部分来查找 VMM 相关内容。可以看到第一个应该关注的函数是 setup_arch()，在这个函数当中使用paging_init() 函数来初始化和映射硬件页表（在初始化前已有 8M 内存被映射，在这里不做记录），而 paging_init() 则是调用的pagetable_init() 来完成内核物理地址的映射以及相关内存的初始化。在 pagetable_init() 函数中，首先是一些 PAE/PSE/PGE 相关判断 与设置，然后使用 kernel_physical_mapping_init() 函数来实现内核物理内存的映射。在这个函数中可以很清楚的看到，pgd_idx 是以PAGE_OFFSET 为启始地址进行映射的，也就是说循环初始化所有物理地址是以 PAGE_OFFSET 为起点的。继续观察我们可以看到在 PMD 被初始化后，所有地址计算均是以 PAGE_OFFSET 作为标记来递增的。分析到这里已经很明显的可以看出，物理地址被映射到以 PAGE_OFFSET 开始的虚拟地址空间。这样以上所有疑问就都有了答案。kmalloc() 与__get_free_page() 所分配的物理页面被映射到了 PAGE_OFFSET 开始的虚拟地址，也就是说实际物理地址与虚拟地址有一组一一对应的关系， 正是因为有了这种映射关系，对内核以 PAGE_OFFSET 启始的虚拟地址的分配也就是对物理地址的分配（当然这有一定的范围，应该在 PAGE_OFFSET与 VMALLOC_START 之间，后者为 vmalloc()

函数分配内存的启始地址）。这也就解释了为什么 virt_to_phys() 与 phys_to_virt() 函数的实现仅仅是加/减 PAGE_OFFSET 即可在虚拟地址与物理地址之间转换，正是因为了有了这种映射，且固定不变，所以才不用去查页表进行转换。这也同样回答了开始的问题，即 kmalloc() / __get_free_page() 分配的是物理地址，而返回的则是虚拟地址（虽然这听上去有些别扭）。正是因为有了这种映射关系，所以需要将它们的返回地址减去 PAGE_OFFSET 才可以得到真正的物理地址。