

26 | 内核态内存映射：如何找到正确的会议室？

2019-05-27 刘超

趣谈Linux操作系统

[进入课程 >](#)



讲述：刘超

时长 15:01 大小 13.77M



前面讲用户态内存映射机制的时候，我们已经多次引申出了内核的映射机制，但是咱们都暂时放了放，这一节我们就来详细解析一下，让你彻底搞懂它。

首先，你要知道，内核态的内存映射机制，主要包含以下几个部分：

内核态内存映射函数 `vmalloc`、`kmap_atomic` 是如何工作的；


内核态页表是放在哪里的，如何工作的？`swapper_pg_dir` 是怎么回事；

出现了内核态缺页异常应该怎么办？

内核页表

和用户态页表不同，在系统初始化的时候，我们就要创建内核页表了。


我们从内核页表的根 `swapper_pg_dir` 开始找线索，在 `arch/x86/include/asm/pgtable_64.h` 中就能找到它的定义。

 复制代码

```
1 extern pud_t level3_kernel_pgt[512];
2 extern pud_t level3_ident_pgt[512];
3 extern pmd_t level2_kernel_pgt[512];
4 extern pmd_t level2_fixmap_pgt[512];
5 extern pmd_t level2_ident_pgt[512];
6 extern pte_t level1_fixmap_pgt[512];
7 extern pgd_t init_top_pgt[];
8
9
10 #define swapper_pg_dir init_top_pgt
```

`swapper_pg_dir` 指向内核最顶级的目录 `pgd`，同时出现的还有几个页表目录。我们可以回忆一下，64 位系统的虚拟地址空间的布局，其中 `XXX_ident_pgt` 对应的是直接映射区，`XXX_kernel_pgt` 对应的是内核代码区，`XXX_fixmap_pgt` 对应的是固定映射区。

它们是在哪里初始化的呢？在汇编语言的文件里面的 `arch\x86\kernel\head_64.S`。这段代码比较难看懂，你只要明白它是干什么的就行了。

 复制代码

```
1 __INITDATA
2
3
4 NEXT_PAGE(init_top_pgt)
5     .quad    level3_ident_pgt - __START_KERNEL_map + _KERNPG_TABLE
6     .org     init_top_pgt + PGD_PAGE_OFFSET*8, 0
7     .quad    level3_ident_pgt - __START_KERNEL_map + _KERNPG_TABLE
8     .org     init_top_pgt + PGD_START_KERNEL*8, 0
9     /* (2^48-(2*1024*1024*1024))/(2^39) = 511 */
10    .quad    level3_kernel_pgt - __START_KERNEL_map + _PAGE_TABLE
11
12
13 NEXT_PAGE(level3_ident_pgt)
14    .quad    level2_ident_pgt - __START_KERNEL_map + _KERNPG_TABLE
15    .fill    511, 8, 0
16 NEXT_PAGE(level2_ident_pgt)
17    /* Since I easily can, map the first 1G.
18     * Don't set NX because code runs from these pages.
19     */
20    PMDS(0, __PAGE_KERNEL_IDENT_LARGE_EXEC, PTRS_PER_PMD)
```

```

21
22
23 NEXT_PAGE(level3_kernel_pgt)
24     .fill    L3_START_KERNEL,8,0
25     /* (2^48-(2*1024*1024*1024)-((2^39)*511))/(2^30) = 510 */
26     .quad    level2_kernel_pgt - __START_KERNEL_map + _KERNPG_TABLE
27     .quad    level2_fixmap_pgt - __START_KERNEL_map + _PAGE_TABLE
28
29
30 NEXT_PAGE(level2_kernel_pgt)
31     /*
32      * 512 MB kernel mapping. We spend a full page on this pagetable
33      * anyway.
34      *
35      * The kernel code+data+bss must not be bigger than that.
36      *
37      * (NOTE: at +512MB starts the module area, see MODULES_VADDR.
38      * If you want to increase this then increase MODULES_VADDR
39      * too.)
40      */
41     PMDS(0, __PAGE_KERNEL_LARGE_EXEC,
42           KERNEL_IMAGE_SIZE/PMD_SIZE)
43
44
45 NEXT_PAGE(level2_fixmap_pgt)
46     .fill    506,8,0
47     .quad    level1_fixmap_pgt - __START_KERNEL_map + _PAGE_TABLE
48     /* 8MB reserved for vsyscalls + a 2MB hole = 4 + 1 entries */
49     .fill    5,8,0
50
51
52 NEXT_PAGE(level1_fixmap_pgt)
53     .fill    51

```

内核页表的顶级目录 `init_top_pgt`，定义在 `__INITDATA` 里面。咱们讲过 ELF 的格式，也讲过虚拟内存空间的布局。它们都有代码段，还有一些初始化了的全局变量，放在 `.init` 区域。这些说的就是这个区域。可以看到，页表的根其实是全局变量，这就使得我们初始化的时候，甚至内存管理还没有初始化的时候，很容易就可以定位到。


接下来，定义 `init_top_pgt` 包含哪些项，这个汇编代码比较难懂了。你可以简单地认为，`quad` 是声明了一项的内容，`org` 是跳到了某个位置。

所以，`init_top_pgt` 有三项，上来先有一项，指向的是 `level3_ident_pgt`，也即直接映射区页表的三级目录。为什么要减去 `__START_KERNEL_map` 呢？因为 `level3_ident_pgt` 是

定义在内核代码里的，写代码的时候，写的都是虚拟地址，谁写代码的时候也不知道将来加载的物理地址是多少呀，对不对？

因为 `level3_ident_pgt` 是在虚拟地址的内核代码段里的，而 `__START_KERNEL_map` 正是虚拟地址空间的内核代码段的起始地址，这在讲 64 位虚拟地址空间的时候都讲过了，要是想不起来就赶紧去回顾一下。这样，`level3_ident_pgt` 减去 `__START_KERNEL_map` 才是物理地址。

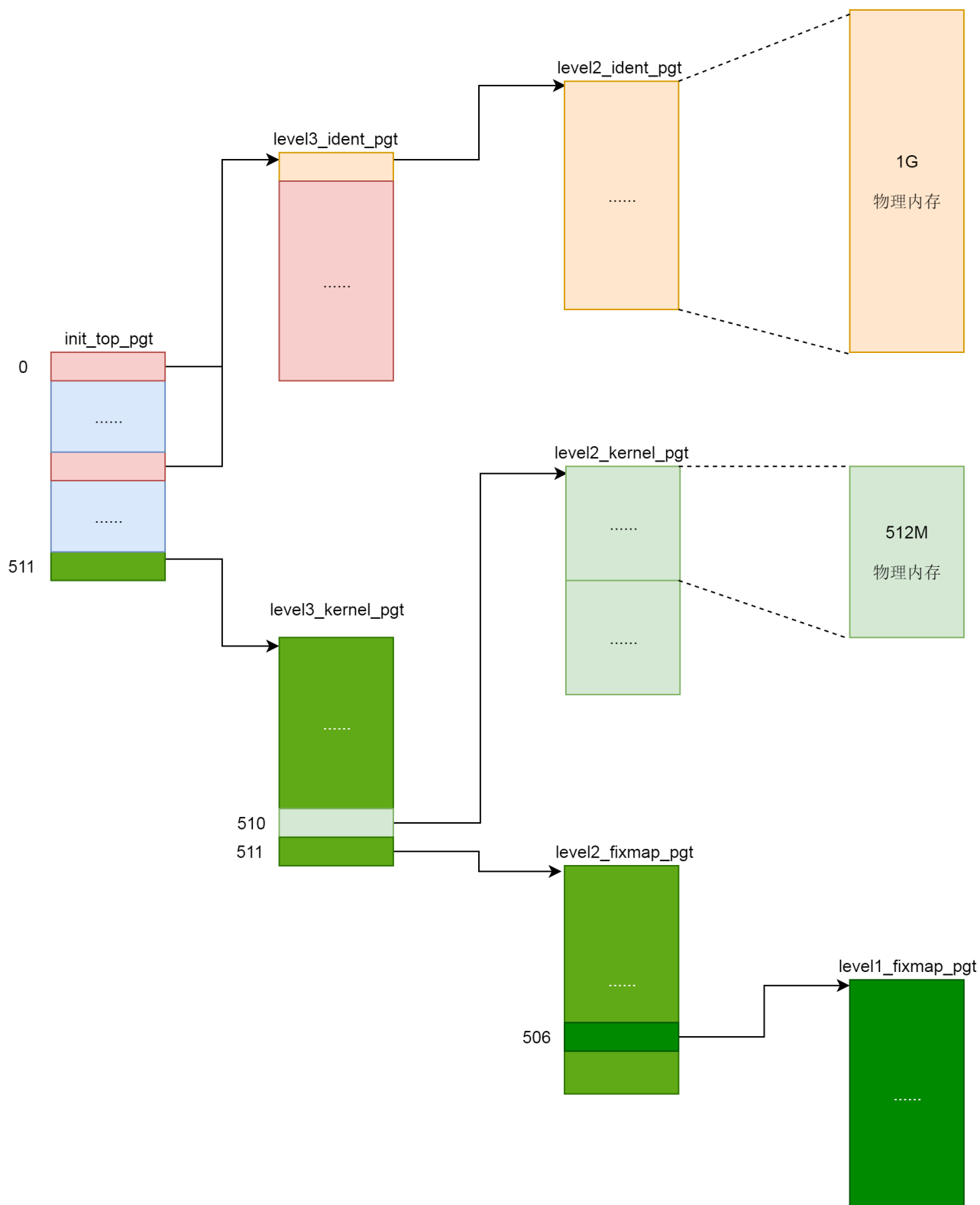
第一项定义完了以后，接下来我们跳到 `PGD_PAGE_OFFSET` 的位置，再定义一项。从定义可以看出，这一项就应该是 `__PAGE_OFFSET_BASE` 对应的。`__PAGE_OFFSET_BASE` 是虚拟地址空间里面内核的起始地址。第二项也指向 `level3_ident_pgt`，直接映射区。

 复制代码

```
1 PGD_PAGE_OFFSET = pgd_index(__PAGE_OFFSET_BASE)
2 PGD_START_KERNEL = pgd_index(__START_KERNEL_map)
3 L3_START_KERNEL = pud_index(__START_KERNEL_map)
```

第二项定义完了以后，接下来跳到 `PGD_START_KERNEL` 的位置，再定义一项。从定义可以看出，这一项应该是 `__START_KERNEL_map` 对应的项，`__START_KERNEL_map` 是虚拟地址空间里面内核代码段的起始地址。第三项指向 `level3_kernel_pgt`，内核代码区。

接下来的代码就很类似了，就是初始化个表项，然后指向下一级目录，最终形成下面这张图。



内核页表定义完了，一开始这里的页表能够覆盖的内存范围比较小。例如，内核代码区 512M，直接映射区 1G。这个时候，其实只要能够映射基本的内核代码和数据结构就可以了。可以看出，里面还空着很多项，可以用于将来映射巨大的内核虚拟地址空间，等用到的时候再进行映射。

如果是用户态进程页表，会有 mm_struct 指向进程顶级目录 pgd，对于内核来讲，也定义了一个 mm_struct，指向 swapper_pg_dir。

 复制代码


```
1 struct mm_struct init_mm = {
2     .mm_rb          = RB_ROOT,
3     .pgd            = swapper_pg_dir,
4     .mm_users       = ATOMIC_INIT(2),
5     .mm_count       = ATOMIC_INIT(1),
6     .mmap_sem       = __RWSEM_INITIALIZER(init_mm.mmap_sem),
7     .page_table_lock = __SPIN_LOCK_UNLOCKED(init_mm.page_table_lock),
8     .mmlist         = LIST_HEAD_INIT(init_mm.mmlist),
9     .user_ns        = &init_user_ns,
10    INIT_MM_CONTEXT(init_mm)
11 };
```

定义完了内核页表，接下来是初始化内核页表，在系统启动的时候 start_kernel 会调用 setup_arch。

 复制代码

```
1 void __init setup_arch(char **cmdline_p)
2 {
3     /*
4      * copy kernel address range established so far and switch
5      * to the proper swapper page table
6      */
7     clone_pgd_range(swapper_pg_dir + KERNEL_PGD_BOUNDARY,
8                     initial_page_table + KERNEL_PGD_BOUNDARY,
9                     KERNEL_PGD_PTRS);
10
11
12     load_cr3(swapper_pg_dir);
13     __flush_tlb_all();
14     .....
15     init_mm.start_code = (unsigned long) _text;
16     init_mm.end_code = (unsigned long) _etext;
17     init_mm.end_data = (unsigned long) _edata;
18     init_mm.brk = _brk_end;
19     .....
20     init_mem_mapping();
21     .....
22 }
```

在 setup_arch 中, load_cr3(swapper_pg_dir) 说明内核页表要开始起作用了, 并且刷新了 TLB, 初始化 init_mm 的成员变量, 最重要的就是 init_mem_mapping。最终它会调用 kernel_physical_mapping_init。

 复制代码

```
1  /*
2   * Create page table mapping for the physical memory for specific physical
3   * addresses. The virtual and physical addresses have to be aligned on PMD level
4   * down. It returns the last physical address mapped.
5   */
6  unsigned long __meminit
7  kernel_physical_mapping_init(unsigned long paddr_start,
8                              unsigned long paddr_end,
9                              unsigned long page_size_mask)
10 {
11     unsigned long vaddr, vaddr_start, vaddr_end, vaddr_next, paddr_last;
12
13
14     paddr_last = paddr_end;
15     vaddr = (unsigned long)__va(paddr_start);
16     vaddr_end = (unsigned long)__va(paddr_end);
17     vaddr_start = vaddr;
18
19
20     for (; vaddr < vaddr_end; vaddr = vaddr_next) {
21         pgd_t *pgd = pgd_offset_k(vaddr);
22         p4d_t *p4d;
23
24
25         vaddr_next = (vaddr & PGDIR_MASK) + PGDIR_SIZE;
26
27
28         if (pgd_val(*pgd)) {
29             p4d = (p4d_t *)pgd_page_vaddr(*pgd);
30             paddr_last = phys_p4d_init(p4d, __pa(vaddr),
31                                       __pa(vaddr_end),
32                                       page_size_mask);
33             continue;
34         }
35
36
37         p4d = alloc_low_page();
38         paddr_last = phys_p4d_init(p4d, __pa(vaddr), __pa(vaddr_end),
39                                   page_size_mask);
40
41
42         p4d_populate(&init_mm, p4d_offset(pgd, vaddr), (pud_t *) p4d);
43     }
44     __flush_tlb_all();
```



```
45
46
47         return paddr_1
```

在 `kernel_physical_mapping_init` 里，我们先通过 `__va` 将物理地址转换为虚拟地址，然后在创建虚拟地址和物理地址的映射页表。

你可能会问，怎么这么麻烦啊？既然对于内核来讲，我们可以用 `__va` 和 `__pa` 直接在虚拟地址和物理地址之间直接转来转去，为啥还要辛辛苦苦建立页表呢？因为这是 CPU 和内存的硬件的需求，也就是说，CPU 在保护模式下访问虚拟地址的时候，就会用 CR3 这个寄存器，这个寄存器是 CPU 定义的，作为操作系统，我们是软件，只能按照硬件的要求来。

你可能又会问了，按照咱们将初始化的时候的过程，系统早早就进入了保护模式，到了 `setup_arch` 里面才 `load_cr3`，如果使用 `cr3` 是硬件的要求，那之前是怎么办呢？如果你仔细去看 `arch\x86\kernel\head_64.S`，这里面除了初始化内核页表之外，在这之前，还有另一个页表 `early_top_pgt`。看到关键字 `early` 了嘛？这个页表就是专门用在真正的内核页表初始化之前，为了遵循硬件的要求而设置的。早期页表不是我们这节的重点，这里我就不展开多说了。

vmalloc 和 kmap_atomic 原理

在用户态可以通过 `malloc` 函数分配内存，当然 `malloc` 在分配比较大的内存的时候，底层调用的是 `mmap`，当然也可以直接通过 `mmap` 做内存映射，在内核里面也有相应的函数。

在虚拟地址空间里面，有个 `vmalloc` 区域，从 `VMALLOC_START` 开始到 `VMALLOC_END`，可以用于映射一段物理内存。

 复制代码

```
1 /**
2  *      vmalloc - allocate virtually contiguous memory
3  *      @size:      allocation size
4  *      Allocate enough pages to cover @size from the page level
5  *      allocator and map them into contiguous kernel virtual space.
6  *
7  *      For tight control over page level allocator and protection flags
8  *      use __vmalloc() instead.
9  */
```



```

10 void *vmalloc(unsigned long size)
11 {
12     return __vmalloc_node_flags(size, NUMA_NO_NODE,
13                                 GFP_KERNEL);
14 }
15
16
17 static void *__vmalloc_node(unsigned long size, unsigned long align,
18                             gfp_t gfp_mask, pgprot_t prot,
19                             int node, const void *caller)
20 {
21     return __vmalloc_node_range(size, align, VMALLOC_START, VMALLOC_END,
22                                 gfp_mask, prot, 0, node, caller);
23 }

```

我们再来看内核的临时映射函数 `kmap_atomic` 的实现。从下面的代码我们可以看出，如果是 32 位有高端地址的，就需要调用 `set_pte` 通过内核页表进行临时映射；如果是 64 位没有高端地址的，就调用 `page_address`，里面会调用 `lowmem_page_address`。其实低端内存的映射，会直接使用 `__va` 进行临时映射。

 复制代码

```

1 void *kmap_atomic_prot(struct page *page, pgprot_t prot)
2 {
3     .....
4     if (!PageHighMem(page))
5         return page_address(page);
6     .....
7     vaddr = __fix_to_virt(FIX_KMAP_BEGIN + idx);
8     set_pte(kmap_pte-idx, mk_pte(page, prot));
9     .....
10    return (void *)vaddr;
11 }
12
13
14 void *kmap_atomic(struct page *page)
15 {
16     return kmap_atomic_prot(page, kmap_prot);
17 }
18
19
20 static __always_inline void *lowmem_page_address(const struct page *page)
21 {
22     return page_to_virt(page);
23 }
24
25


```

```
26 #define page_to_virt(x) __va(PFN_PHYS(page_to_pfn(x))
```

内核态缺页异常

可以看出，kmap_atomic 和 vmalloc 不同。kmap_atomic 发现，没有页表的时候，就直接创建页表进行映射了。而 vmalloc 没有，它只分配了内核的虚拟地址。所以，访问它的时候，会产生缺页异常。

内核态的缺页异常还是会调用 do_page_fault，但是会走到咱们上面用户态缺页异常中没有解析的那部分 vmalloc_fault。这个函数并不复杂，主要用于关联内核页表项。

 复制代码

```
1  /*
2   * 32-bit:
3   *
4   *   Handle a fault on the vmalloc or module mapping area
5   */
6  static noinline int vmalloc_fault(unsigned long address)
7  {
8      unsigned long pgd_paddr;
9      pmd_t *pmd_k;
10     pte_t *pte_k;
11
12
13     /* Make sure we are in vmalloc area: */
14     if (!(address >= VMALLOC_START && address < VMALLOC_END))
15         return -1;
16
17
18     /*
19      * Synchronize this task's top level page-table
20      * with the 'reference' page table.
21      *
22      * Do _not_ use "current" here. We might be inside
23      * an interrupt in the middle of a task switch..
24      */
25     pgd_paddr = read_cr3_pa();
26     pmd_k = vmalloc_sync_one(__va(pgd_paddr), address);
27     if (!pmd_k)
28         return -1;
29
30
31     pte_k = pte_offset_kernel(pmd_k, address);
32     if (!pte_present(*pte_k))
33         return -1;
```

```
34
35
36         return 0
```

总结时刻

至此，内核态的内存映射也讲完了。这下，我们可以将整个内存管理的体系串起来了。

物理内存根据 NUMA 架构分节点。每个节点里面再分区域。每个区域里面再分页。

物理页面通过伙伴系统进行分配。分配的物理页面要变成虚拟地址让上层可以访问，kswapd 可以根据物理页面的使用情况对页面进行换入换出。

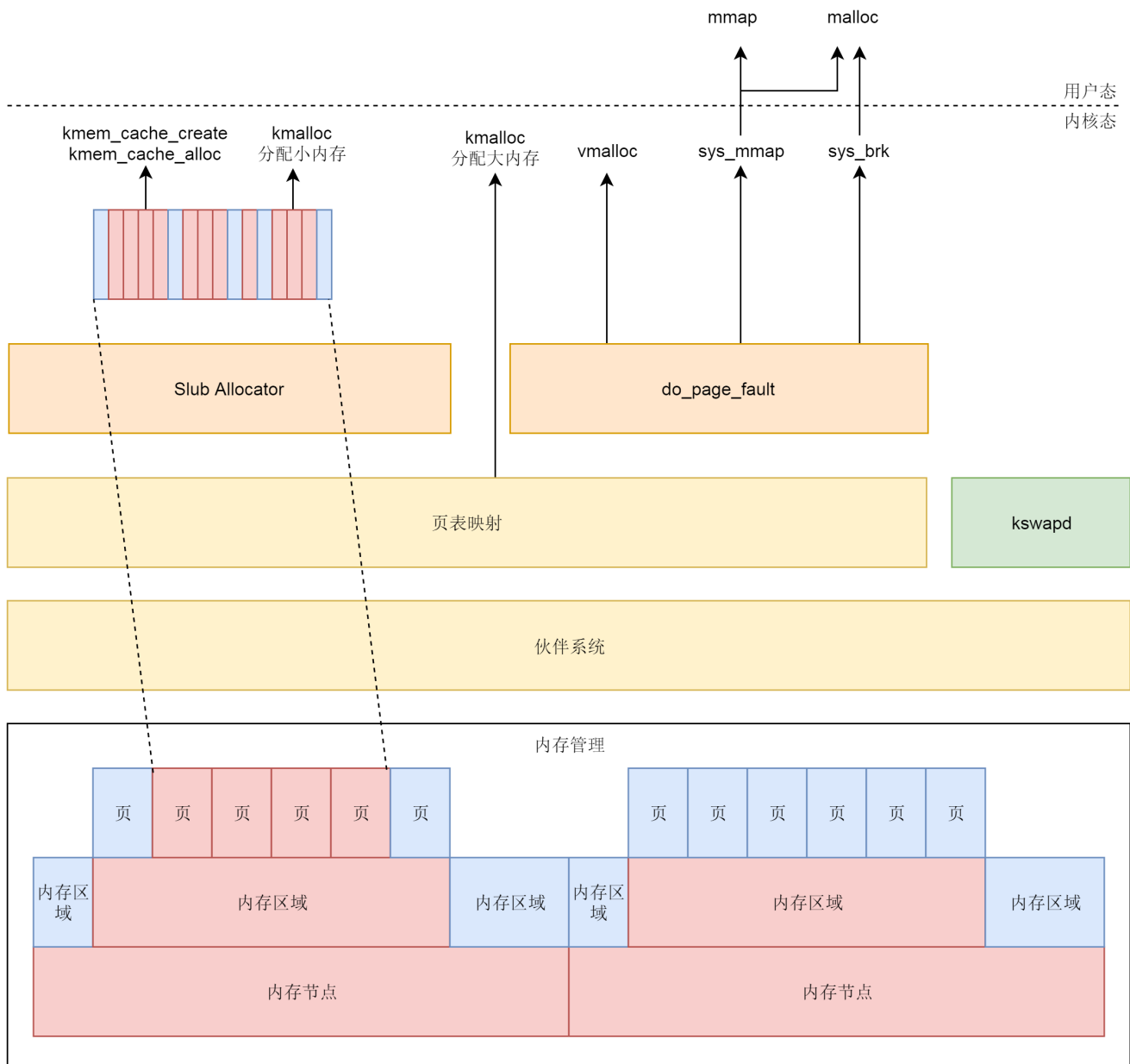
对于内存的分配需求，可能来自内核态，也可能来自用户态。

对于内核态，kmalloc 在分配大内存的时候，以及 vmalloc 分配不连续物理页的时候，直接使用伙伴系统，分配后转换为虚拟地址，访问的时候需要通过内核页表进行映射。

对于 kmem_cache 以及 kmalloc 分配小内存，则使用 slub 分配器，将伙伴系统分配出来的大块内存切成一小块一小块进行分配。

kmem_cache 和 kmalloc 的部分不会被换出，因为用这两个函数分配的内存多用于保持内核关键的数据结构。内核态中 vmalloc 分配的部分会被换出，因而当访问的时候，发现不在，就会调用 do_page_fault。

对于用户态的内存分配，或者直接调用 mmap 系统调用分配，或者调用 malloc。调用 malloc 的时候，如果分配小的内存，就用 sys_brk 系统调用；如果分配大的内存，还是用 sys_mmap 系统调用。正常情况下，用户态的内存都是可以换出的，因而一旦发现内存中不存在，就会调用 do_page_fault。



课堂练习

伙伴系统分配好了物理页面之后，如何转换成为虚拟地址呢？请研究一下 `page_address` 函数的实现。

欢迎留言和我分享你的疑惑和见解，也欢迎你收藏本节内容，反复研读。你也可以把今天的内容分享给你的朋友，和他一起学习、进步。

趣谈 Linux 操作系统

像故事一样的操作系统入门课

刘超

网易杭州研究院

云计算技术部首席架构师



新版升级：点击「👤 请朋友读」，10位好友免费读，邀请订阅更有**现金**奖励。

© 版权归极客邦科技所有，未经许可不得传播售卖。页面已增加防盗追踪，如有侵权极客邦将依法追究其法律责任。

上一篇 25 | 用户态内存映射：如何找到正确的会议室？

下一篇 27 | 文件系统：项目成果要归档，我们就需要档案库

精选留言 (3)

写留言



Sentry

2019-05-27

1

内核能用c语言编写，是不是意味着用c可以直接操作物理内存，另外linux上的c语言编译器是用什么语言开发的，c语言实现了自举吗，c语言跨平台底层原理是什么，请老师答疑解惑。

展开

作者回复: 是因为C语言编译完了就直接是硬件能够识别的二进制，不像java，需要jvm才能运行。C语言不用自举，除了第一个开发C语言的，需要用汇编来做，后面都可以用锤子造锤子



LDxy
2019-06-02



操作系统是如何知道计算机上有多少物理内存的呢?

展开 ▾



活的潇洒
2019-05-29



决心从头把计算机所有的基础课程全部补上，夯实基础，一定要坚持到最后
day26笔记: <https://www.cnblogs.com/luoahong/p/10931320.html>