

## 25 | 用户态内存映射：如何找到正确的会议室？

2019-05-24 刘超

趣谈Linux操作系统

[进入课程 >](#)



讲述：刘超

时长 23:01 大小 18.45M



前面几节，我们既看了虚拟内存空间如何组织的，也看了物理页面如何管理的。现在我们需要一些数据结构，将二者关联起来。

### mmap 的原理

在虚拟地址空间那一节，我们知道，每一个进程都有一个列表 `vm_area_struct`，指向虚拟地址空间的不同的内存块，这个变量的名字叫 **mmap**。

复制代码

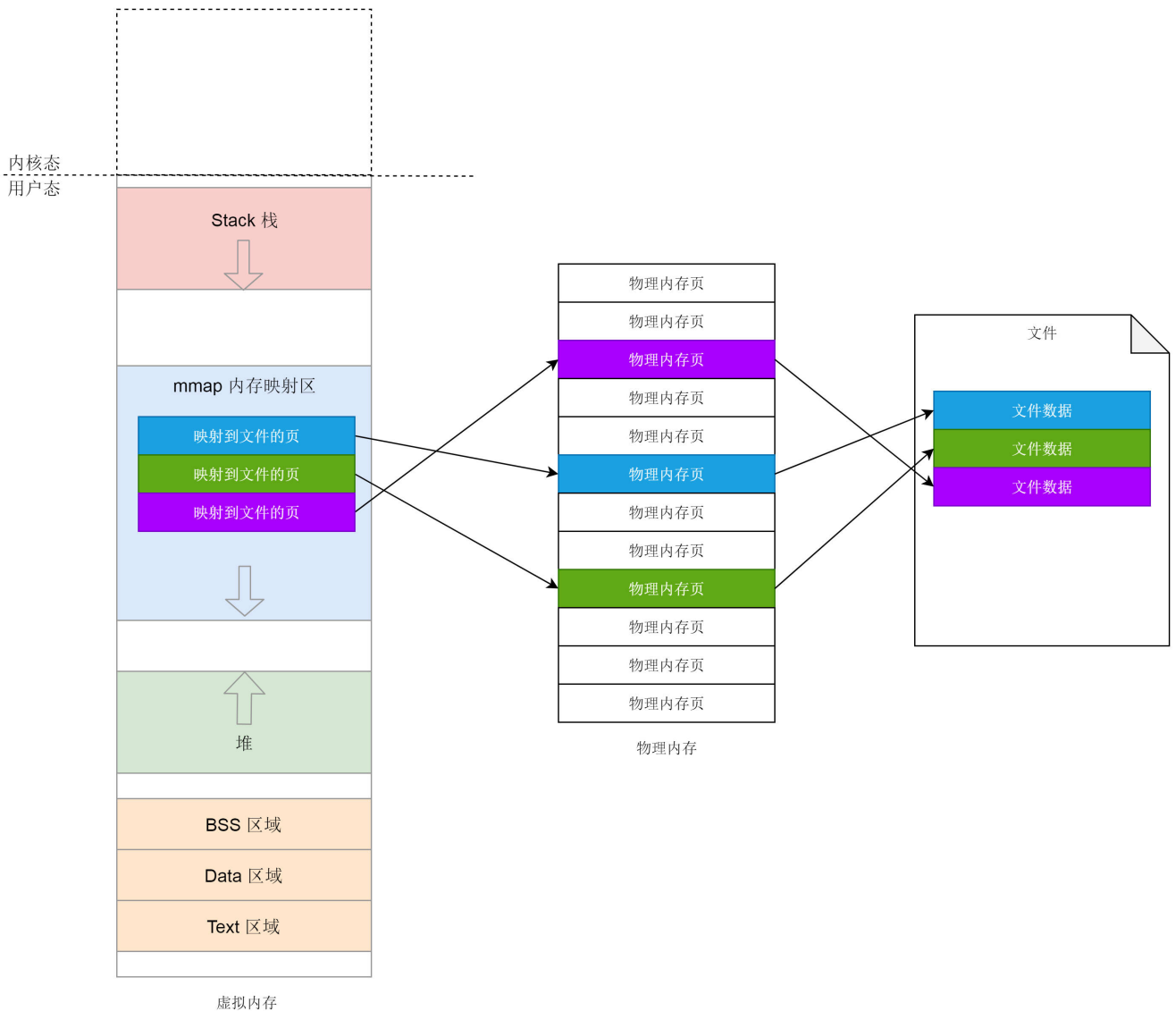
```
1 struct mm_struct {  
2     struct vm_area_struct *mmap;           /* list of VMAs */  
3     .....  
4 }  
5
```

```

6
7 struct vm_area_struct {
8     /*
9      * For areas with an address space and backing store,
10     * linkage into the address_space->i_mmap interval tree.
11     */
12     struct {
13         struct rb_node rb;
14         unsigned long rb_subtree_last;
15     } shared;
16
17
18
19
20     /*
21     * A file's MAP_PRIVATE vma can be in both i_mmap tree and anon_vma
22     * list, after a COW of one of the file pages. A MAP_SHARED vma
23     * can only be in the i_mmap tree. An anonymous MAP_PRIVATE, stack
24     * or brk vma (with NULL file) can only be in an anon_vma list.
25     */
26     struct list_head anon_vma_chain; /* Serialized by mmap_sem &
27                                     * page_table_lock */
28     struct anon_vma *anon_vma;      /* Serialized by page_table_lock */
29
30
31
32
33     /* Function pointers to deal with this struct. */
34     const struct vm_operations_struct *vm_ops;
35     /* Information about our backing store: */
36     unsigned long vm_pgoff;          /* Offset (within vm_file) in PAGE_SIZE
37                                     units */
38     struct file * vm_file;           /* File we map to (can be NULL). */
39     void * vm_private_data;          /* was vm_pte (shared mem) */

```

其实内存映射不仅仅是物理内存和虚拟内存之间的映射，还包括将文件中的内容映射到虚拟内存空间。这个时候，访问内存空间就能够访问到文件里面的数据。而仅有物理内存和虚拟内存的映射，是一种特殊情况。



前面咱们讲堆的时候讲过，如果我们要申请小块内存，就用 `brk`。`brk` 函数之前已经解析过了，这里就不多说了。如果申请一大块内存，就要用 `mmap`。对于堆的申请来讲，`mmap` 是映射内存空间到物理内存。

另外，如果一个进程想映射一个文件到自己的虚拟内存空间，也要通过 `mmap` 系统调用。这个时候 `mmap` 是映射内存空间到物理内存再到文件。可见 `mmap` 这个系统调用是核心，我们现在来看 `mmap` 这个系统调用。

[复制代码](#)

```
1 SYSCALL_DEFINE6(mmap, unsigned long, addr, unsigned long, len,
2                 unsigned long, prot, unsigned long, flags,
3                 unsigned long, fd, unsigned long, off)
4 {
5     .....
6     error = sys_mmap_pgoff(addr, len, prot, flags, fd, off >> PAGE_SHIFT);
7     .....
8 }
```

```

9
10
11 SYSCALL_DEFINE6(mmap_pgoff, unsigned long, addr, unsigned long, len,
12                 unsigned long, prot, unsigned long, flags,
13                 unsigned long, fd, unsigned long, pgoff)
14 {
15     struct file *file = NULL;
16     .....
17     file = fget(fd);
18     .....
19     retval = vm_mmap_pgoff(file, addr, len, prot, flags, pgoff);
20     return retval;
21 }

```


如果要映射到文件，fd 会传进来一个文件描述符，并且 mmap\_pgoff 里面通过 fget 函数，根据文件描述符获得 struct file。struct file 表示打开的一个文件。

接下来的调用链是 vm\_mmap\_pgoff->do\_mmap\_pgoff->do\_mmap。这里面主要干了两件事情：

调用 get\_unmapped\_area 找到一个没有映射的区域；

调用 mmap\_region 映射这个区域。

我们先来看 get\_unmapped\_area 函数。

 复制代码


```

1 unsigned long
2 get_unmapped_area(struct file *file, unsigned long addr, unsigned long len,
3                 unsigned long pgoff, unsigned long flags)
4 {
5     unsigned long (*get_area)(struct file *, unsigned long,
6                             unsigned long, unsigned long, unsigned long);
7     .....
8     get_area = current->mm->get_unmapped_area;
9     if (file) {
10         if (file->f_op->get_unmapped_area)
11             get_area = file->f_op->get_unmapped_area;
12     }
13     .....
14 }

```


这里面如果是匿名映射，则调用 `mm_struct` 里面的 `get_unmapped_area` 函数。这个函数其实是 `arch_get_unmapped_area`。它会调用 `find_vma_prev`，在表示虚拟内存区域的 `vm_area_struct` 红黑树上找到相应的位置。之所以叫 `prev`，是说这个时候虚拟内存区域还没有建立，找到前一个 `vm_area_struct`。

如果不是匿名映射，而是映射到一个文件，这样在 Linux 里面，每个打开的文件都有一个 `struct file` 结构，里面有一个 `file_operations`，用来表示和这个文件相关的操作。如果是我们熟知的 `ext4` 文件系统，调用的是 `thp_get_unmapped_area`。如果我们仔细看这个函数，最终还是调用 `mm_struct` 里面的 `get_unmapped_area` 函数。殊途同归。

 复制代码

```
1 const struct file_operations ext4_file_operations = {
2     .....
3     .mmap          = ext4_file_mmap
4     .get_unmapped_area = thp_get_unmapped_area,
5 };
6
7
8 unsigned long __thp_get_unmapped_area(struct file *filp, unsigned long len,
9     loff_t off, unsigned long flags, unsigned long size)
10 {
11     unsigned long addr;
12     loff_t off_end = off + len;
13     loff_t off_align = round_up(off, size);
14     unsigned long len_pad;
15     len_pad = len + size;
16     .....
17     addr = current->mm->get_unmapped_area(filp, 0, len_pad,
18     off >> PAGE_SHIFT, flags);
19     addr += (off - addr) & (size - 1);
20     return addr;
21 }
```

我们再来看 `mmap_region`，看它如何映射这个虚拟内存区域。

 复制代码

```
1 unsigned long mmap_region(struct file *file, unsigned long addr,
2     unsigned long len, vm_flags_t vm_flags, unsigned long pgoff,
3     struct list_head *uf)
4 {
5     struct mm_struct *mm = current->mm;
6     struct vm_area_struct *vma, *prev;
```


```

7      struct rb_node **rb_link, *rb_parent;
8
9
10     /*
11      * Can we just expand an old mapping?
12      */
13     vma = vma_merge(mm, prev, addr, addr + len, vm_flags,
14                    NULL, file, pgoff, NULL, NULL_VM_UFFD_CTX);
15     if (vma)
16         goto out;
17
18
19     /*
20      * Determine the object being mapped and call the appropriate
21      * specific mapper. the address has already been validated, but
22      * not unmapped, but the maps are removed from the list.
23      */
24     vma = kmem_cache_zalloc(vm_area_cachep, GFP_KERNEL);
25     if (!vma) {
26         error = -ENOMEM;
27         goto unacct_error;
28     }
29
30
31     vma->vm_mm = mm;
32     vma->vm_start = addr;
33     vma->vm_end = addr + len;
34     vma->vm_flags = vm_flags;
35     vma->vm_page_prot = vm_get_page_prot(vm_flags);
36     vma->vm_pgoff = pgoff;
37     INIT_LIST_HEAD(&vma->anon_vma_chain);
38
39
40     if (file) {
41         vma->vm_file = get_file(file);
42         error = call_mmap(file, vma);
43         addr = vma->vm_start;
44         vm_flags = vma->vm_flags;
45     }
46     .....
47     vma_link(mm, vma, prev, rb_link, rb_parent);
48     return addr;
49     .....

```

还记得咱们刚找到了虚拟内存区域的前一个 `vm_area_struct`，我们首先要看，是否能够基于它进行扩展，也即调用 `vma_merge`，和前一个 `vm_area_struct` 合并到一起。

如果不能，就需要调用 `kmem_cache_zalloc`，在 Slub 里面创建一个新的 `vm_area_struct` 对象，设置起始和结束位置，将它加入队列。如果是映射到文件，则设置 `vm_file` 为目标文件，调用 `call_mmap`。其实就是调用 `file_operations` 的 `mmap` 函数。对于 ext4 文件系统，调用的是 `ext4_file_mmap`。从这个函数的参数可以看出，这一刻文件和内存开始发生关系了。这里我们将 `vm_area_struct` 的内存操作设置为文件系统操作，也就是说，读写内存其实就是读写文件系统。


 复制代码

```
1 static inline int call_mmap(struct file *file, struct vm_area_struct *vma)
2 {
3     return file->f_op->mmap(file, vma);
4 }
5
6
7 static int ext4_file_mmap(struct file *file, struct vm_area_struct *vma)
8 {
9     .....
10     vma->vm_ops = &ext4_file_vm_ops;
11     .....
12 }
```

我们再回到 `mmap_region` 函数。最终，`vma_link` 函数将新创建的 `vm_area_struct` 挂在了 `mm_struct` 里面的红黑树上。

这个时候，从内存到文件的映射关系，至少要在逻辑层面建立起来。那从文件到内存的映射关系呢？`vma_link` 还做了另外一件事情，就是 `__vma_link_file`。这个东西要用于建立这层映射关系。

对于打开的文件，会有一个结构 `struct file` 来表示。它有个成员指向 `struct address_space` 结构，这里面有棵变量名为 `i_mmap` 的红黑树，`vm_area_struct` 就挂在这棵树上。

 复制代码

```
1 struct address_space {
2     struct inode          *host;          /* owner: inode, block_device */
3     .....
4     struct rb_root        i_mmap;        /* tree of private and shared mappings */
5     .....
6     const struct address_space_operations *a_ops; /* methods */
7 }
```

```

7 .....
8 }
9
10
11 static void __vma_link_file(struct vm_area_struct *vma)
12 {
13     struct file *file;
14
15
16     file = vma->vm_file;
17     if (file) {
18         struct address_space *mapping = file->f_mapping;
19         vma_interval_tree_insert(vma, &mapping->i_mmap);
20     }

```

到这里，内存映射的内容要告一段落了。你可能会困惑，好像还没和物理内存发生任何关系，还是在虚拟内存里面折腾呀？

对的，因为到目前为止，我们还没有开始真正访问内存呀！这个时候，内存管理并不直接分配物理内存，因为物理内存相对于虚拟地址空间太宝贵了，只有等你真正用的那一刻才会开始分配。

## 用户态缺页异常

一旦开始访问虚拟内存的某个地址，如果我们发现，并没有对应的物理页，那就触发缺页中断，调用 `do_page_fault`。

 复制代码

```

1 dotraplinkage void notrace
2 do_page_fault(struct pt_regs *regs, unsigned long error_code)
3 {
4     unsigned long address = read_cr2(); /* Get the faulting address */
5     .....
6     __do_page_fault(regs, error_code, address);
7     .....
8 }
9
10
11 /*
12  * This routine handles page faults. It determines the address,
13  * and the problem, and then passes it off to one of the appropriate
14  * routines.
15  */

```




```

16 static ninline void
17 __do_page_fault(struct pt_regs *regs, unsigned long error_code,
18                unsigned long address)
19 {
20     struct vm_area_struct *vma;
21     struct task_struct *tsk;
22     struct mm_struct *mm;
23     tsk = current;
24     mm = tsk->mm;
25
26
27     if (unlikely(fault_in_kernel_space(address))) {
28         if (vmalloc_fault(address) >= 0)
29             return;
30     }
31     .....
32     vma = find_vma(mm, address);
33     .....
34     fault = handle_mm_fault(vma, address, flags);
35     .....

```

在 `__do_page_fault` 里面，先要判断缺页中断是否发生在内核。如果发生在内核则调用 `vmalloc_fault`，这就和咱们前面学过的虚拟内存的布局对应上了。在内核里面，`vmalloc` 区域需要内核页表映射到物理页。咱们这里把内核的这部分放放，接着看用户空间的部分。

接下来在用户空间里面，找到你访问的那个地址所在的区域 `vm_area_struct`，然后调用 `handle_mm_fault` 来映射这个区域。

 复制代码

```

1 static int __handle_mm_fault(struct vm_area_struct *vma, unsigned long address,
2                             unsigned int flags)
3 {
4     struct vm_fault vmf = {
5         .vma = vma,
6         .address = address & PAGE_MASK,
7         .flags = flags,
8         .pgoff = linear_page_index(vma, address),
9         .gfp_mask = __get_fault_gfp_mask(vma),
10    };
11    struct mm_struct *mm = vma->vm_mm;
12    pgd_t *pgd;
13    p4d_t *p4d;
14    int ret;
15
16

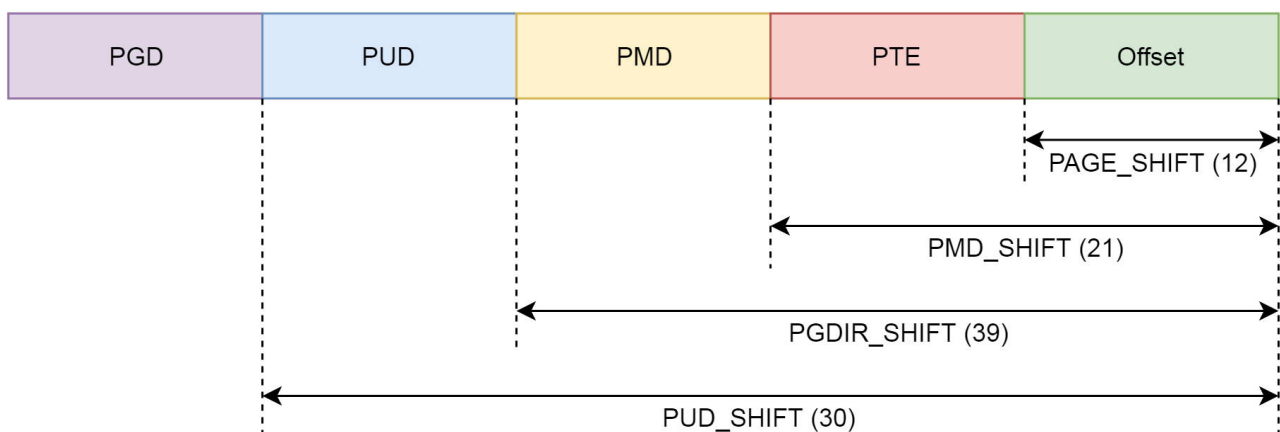
```

```

17     pgd = pgd_offset(mm, address);
18     p4d = p4d_alloc(mm, pgd, address);
19     .....
20     vmf.pud = pud_alloc(mm, p4d, address);
21     .....
22     vmf.pmd = pmd_alloc(mm, vmf.pud, address);
23     .....
24     return handle_pte_fault(&vmf);
25 }

```


到这里，终于看到了我们熟悉的 PGD、P4G、PUD、PMD、PTE，这就是前面讲页表的时候，讲述的四级页表的概念，因为暂且不考虑五级页表，我们暂时忽略 P4G。



pgd\_t 用于全局页目录项，pud\_t 用于上层页目录项，pmd\_t 用于中间页目录项，pte\_t 用于直接页表项。

每个进程都有独立的地址空间，为了这个进程独立完成映射，每个进程都有独立的进程页表，这个页表的最顶级的 pgd 存放在 task\_struct 中的 mm\_struct 的 pgd 变量里面。

在一个进程新创建的时候，会调用 fork，对于内存的部分会调用 copy\_mm，里面调用 dup\_mm。

 复制代码

```

1  /*
2   * Allocate a new mm structure and copy contents from the
3   * mm structure of the passed in task structure.
4   */
5  static struct mm_struct *dup_mm(struct task_struct *tsk)
6  {


```

```

7      struct mm_struct *mm, *oldmm = current->mm;
8      mm = allocate_mm();
9      memcpy(mm, oldmm, sizeof(*mm));
10     if (!mm_init(mm, tsk, mm->user_ns))
11         goto fail_nomem;
12     err = dup_mmap(mm, oldmm);
13     return mm;
14 }

```

在这里，除了创建一个新的 `mm_struct`，并且通过 `memcpy` 将它和父进程的弄成一模一样之外，我们还需要调用 `mm_init` 进行初始化。接下来，`mm_init` 调用 `mm_alloc_pgd`，分配全局页目录项，赋值给 `mm_struct` 的 `pgd` 成员变量。


 复制代码

```

1 static inline int mm_alloc_pgd(struct mm_struct *mm)
2 {
3     mm->pgd = pgd_alloc(mm);
4     return 0;
5 }

```

`pgd_alloc` 里面除了分配 PDG 之外，还做了很重要的一个事情，就是调用 `pgd_ctor`。

 复制代码

```

1 static void pgd_ctor(struct mm_struct *mm, pgd_t *pgd)
2 {
3     /* If the pgd points to a shared pagetable level (either the
4      * ptes in non-PAE, or shared PMD in PAE), then just copy the
5      * references from swapper_pg_dir. */
6     if (CONFIG_PGTABLE_LEVELS == 2 ||
7         (CONFIG_PGTABLE_LEVELS == 3 && SHARED_KERNEL_PMD) ||
8         CONFIG_PGTABLE_LEVELS >= 4) {
9         clone_pgd_range(pgd + KERNEL_PGD_BOUNDARY,
10                        swapper_pg_dir + KERNEL_PGD_BOUNDARY,
11                        KERNEL_PGD_PTRS);
12     }
13     .....
14 }

```

pgd\_ctor 干了什么事情呢？我们注意看里面的注释，它拷贝了对于 swapper\_pg\_dir 的引用。swapper\_pg\_dir 是内核页表的最顶级的全局页目录。

一个进程的虚拟地址空间包含用户态和内核态两部分。为了从虚拟地址空间映射到物理页面，页表也分为用户地址空间的页表和内核页表，这就和上面遇到的 vmalloc 有关系了。在内核里面，映射靠内核页表，这里内核页表会拷贝一份到进程的页表。至于 swapper\_pg\_dir 是什么，怎么初始化的，怎么工作的，我们还是先放一放，放到下一节统一讨论。

至此，一个进程 fork 完毕之后，有了内核页表，有了自己顶级的 pgd，但是对于用户地址空间来讲，还完全没有映射过。这需要等到这个进程在某个 CPU 上运行，并且对内存访问的那一时刻了。

当这个进程被调度到某个 CPU 上运行的时候，咱们在[调度](#)那一节讲过，要调用 context\_switch 进行上下文切换。对于内存方面的切换会调用 switch\_mm\_irqs\_off，这里面会调用 load\_new\_mm\_cr3。


cr3 是 CPU 的一个寄存器，它会指向当前进程的顶级 pgd。如果 CPU 的指令要访问进程的虚拟内存，它就会自动从 cr3 里面得到 pgd 在物理内存的地址，然后根据里面的页表解析虚拟内存的地址为物理内存，从而访问真正的物理内存上的数据。

这里需要注意两点。第一点，cr3 里面存放当前进程的顶级 pgd，这个是硬件的要求。cr3 里面需要存放 pgd 在物理内存的地址，不能是虚拟地址。因而 load\_new\_mm\_cr3 里面会使用 \_\_pa，将 mm\_struct 里面的成员变量 pdg（mm\_struct 里面存的都是虚拟地址）变为物理地址，才能加载到 cr3 里面去。

第二点，用户进程在运行的过程中，访问虚拟内存中的数据，会被 cr3 里面指向的页表转换为物理地址后，才在物理内存中访问数据，这个过程都是在用户态运行的，地址转换的过程无需进入内核态。

只有访问虚拟内存的时候，发现没有映射多物理内存，页表也没有创建过，才触发缺页异常。进入内核调用 do\_page\_fault，一直调用到 \_\_handle\_mm\_fault，这才有了上面解析到这个函数的时候，我们看到的代码。既然原来没有创建过页表，那只好补上这一课。于是，\_\_handle\_mm\_fault 调用 pud\_alloc 和 pmd\_alloc，来创建相应的页目录项，最后调用 handle\_pte\_fault 来创建页表项。

绕了一大圈，终于将页表整个机制的各个部分串了起来。但是咱们的故事还没讲完，物理的内存还没找到。我们还得接着分析 `handle_pte_fault` 的实现。

 复制代码

```
1 static int handle_pte_fault(struct vm_fault *vmf)
2 {
3     pte_t entry;
4     .....
5     vmf->pte = pte_offset_map(vmf->pmd, vmf->address);
6     vmf->orig_pte = *vmf->pte;
7     .....
8     if (!vmf->pte) {
9         if (vma_is_anonymous(vmf->vma))
10             return do_anonymous_page(vmf);
11         else
12             return do_fault(vmf);
13     }
14
15
16     if (!pte_present(vmf->orig_pte))
17         return do_swap_page(vmf);
18     .....
19 }
```

这里面总的来说分了三情况。如果 PTE，也就是页表项，从来没有出现过，那就是新映射的页。如果是匿名页，就是第一种情况，应该映射到一个物理内存页，在这里调用的是 `do_anonymous_page`。如果是映射到文件，调用的就是 `do_fault`，这是第二种情况。如果 PTE 原来出现过，说明原来页面在物理内存中，后来换出到硬盘了，现在应该换回来，调用的是 `do_swap_page`。

我们来看第一种情况，`do_anonymous_page`。对于匿名页的映射，我们需要先通过 `pte_alloc` 分配一个页表项，然后通过 `alloc_zeroed_user_highpage_movable` 分配一个页。之后它会调用 `alloc_pages_vma`，并最终调用 `__alloc_pages_nodemask`。

这个函数你还记得吗？就是咱们伙伴系统的核心函数，专门用来分配物理页面的。`do_anonymous_page` 接下来要调用 `mk_pte`，将页表项指向新分配的物理页，`set_pte_at` 会将页表项塞到页表里面。

 复制代码


```
1 static int do_anonymous_page(struct vm_fault *vmf)
```

```

2 {
3     struct vm_area_struct *vma = vmf->vma;
4     struct mem_cgroup *memcg;
5     struct page *page;
6     int ret = 0;
7     pte_t entry;
8     .....
9     if (pte_alloc(vma->vm_mm, vmf->pmd, vmf->address))
10         return VM_FAULT_OOM;
11     .....
12     page = alloc_zeroed_user_highpage_movable(vma, vmf->address);
13     .....
14     entry = mk_pte(page, vma->vm_page_prot);
15     if (vma->vm_flags & VM_WRITE)
16         entry = pte_mkwrite(pte_mkdirty(entry));
17
18
19     vmf->pte = pte_offset_map_lock(vma->vm_mm, vmf->pmd, vmf->address,
20                                   &vmf->ptl);
21     .....
22     set_pte_at(vma->vm_mm, vmf->address, vmf->pte, entry);
23     .....
24 }

```

第二种情况映射到文件 `do_fault`，最终我们会调用 `__do_fault`。

 复制代码

```

1 static int __do_fault(struct vm_fault *vmf)
2 {
3     struct vm_area_struct *vma = vmf->vma;
4     int ret;
5     .....
6     ret = vma->vm_ops->fault(vmf);
7     .....
8     return ret;
9 }
10

```

这里调用了 `struct vm_operations_struct vm_ops` 的 `fault` 函数。还记得咱们上面用 `mmap` 映射文件的时候，对于 `ext4` 文件系统，`vm_ops` 指向了 `ext4_file_vm_ops`，也就是调用了 `ext4_filemap_fault`。

```

1 static const struct vm_operations_struct ext4_file_vm_ops = {
2     .fault          = ext4_filemap_fault,
3     .map_pages      = filemap_map_pages,
4     .page_mkwrite   = ext4_page_mkwrite,
5 };
6
7
8 int ext4_filemap_fault(struct vm_fault *vmf)
9 {
10     struct inode *inode = file_inode(vmf->vma->vm_file);
11     .....
12     err = filemap_fault(vmf);
13     .....
14     return err;
15 }

```


ext4\_filemap\_fault 里面的逻辑我们很容易就能读懂。vm\_file 就是咱们当时 mmap 的时候映射的那个文件，然后我们需要调用 filemap\_fault。对于文件映射来说，一般这个文件会在物理内存里面有页面作为它的缓存，find\_get\_page 就是找那个页。如果找到了，就调用 do\_async\_mmap\_readahead，预读一些数据到内存里面；如果没有，就跳到 no\_cached\_page。

```

1 int filemap_fault(struct vm_fault *vmf)
2 {
3     int error;
4     struct file *file = vmf->vma->vm_file;
5     struct address_space *mapping = file->f_mapping;
6     struct inode *inode = mapping->host;
7     pgoff_t offset = vmf->pgoff;
8     struct page *page;
9     int ret = 0;
10    .....
11    page = find_get_page(mapping, offset);
12    if (likely(page) && !(vmf->flags & FAULT_FLAG_TRIED)) {
13        do_async_mmap_readahead(vmf->vma, ra, file, page, offset);
14    } else if (!page) {
15        goto no_cached_page;
16    }
17    .....
18    vmf->page = page;
19    return ret | VM_FAULT_LOCKED;
20 no_cached_page:
21    error = page_cache_read(file, offset, vmf->gfp_mask);
22    .....

```

如果没有物理内存中的缓存页，那我们就调用 `page_cache_read`。在这里显示分配一个缓存页，将这一页加到 lru 表里面，然后在 `address_space` 中调用 `address_space_operations` 的 `readpage` 函数，将文件内容读到内存中。`address_space` 的作用咱们上面也介绍过了。

 复制代码

```

1 static int page_cache_read(struct file *file, pgoff_t offset, gfp_t gfp_mask)
2 {
3     struct address_space *mapping = file->f_mapping;
4     struct page *page;
5     .....
6     page = __page_cache_alloc(gfp_mask|__GFP_COLD);
7     .....
8     ret = add_to_page_cache_lru(page, mapping, offset, gfp_mask & GFP_KERNEL);
9     .....
10    ret = mapping->a_ops->readpage(file, page);
11    .....
12 }
```

`struct address_space_operations` 对于 ext4 文件系统的定义如下所示。这么说来，上面的 `readpage` 调用的其实是 `ext4_readpage`。因为我们还没讲到文件系统，这里我们不详细介绍 `ext4_readpage` 具体干了什么。你只要知道，最后会调用 `ext4_read_inline_page`，这里面有部分逻辑和内存映射有关就行了。

 复制代码

```

1 static const struct address_space_operations ext4_aops = {
2     .readpage          = ext4_readpage,
3     .readpages         = ext4_readpages,
4     .....
5 };
6
7
8 static int ext4_read_inline_page(struct inode *inode, struct page *page)
9 {
10     void *kaddr;
11     .....
12     kaddr = kmap_atomic(page);
13     ret = ext4_read_inline_data(inode, kaddr, len, &iloc);
```



```


14         flush_dcache_page(page);
15         kunmap_atomic(kaddr);
16     .....
17 }

```

在 `ext4_read_inline_page` 函数里，我们需要先调用 `kmap_atomic`，将物理内存映射到内核的虚拟地址空间，得到内核中的地址 `kaddr`。我们在前面提到过 `kmap_atomic`，它是用来做临时内核映射的。本来把物理内存映射到用户虚拟地址空间，不需要在内核里面映射一把。但是，现在因为要从文件里面读取数据并写入这个物理页面，又不能使用物理地址，我们只能使用虚拟地址，这就需要在内核里面临时映射一把。临时映射后，`ext4_read_inline_data` 读取文件到这个虚拟地址。读取完毕后，我们取消这个临时映射 `kunmap_atomic` 就行了。

至于 `kmap_atomic` 的具体实现，我们还是放到内核映射部分再讲。

我们再来看第三种情况，`do_swap_page`。之前我们讲过物理内存管理，你这里可以回忆一下。如果长时间不用，就要换出到硬盘，也就是 `swap`，现在这部分数据又要访问了，我们还得想办法再次读到内存中来。

 复制代码

```

1 int do_swap_page(struct vm_fault *vmf)
2 {
3     struct vm_area_struct *vma = vmf->vma;
4     struct page *page, *swapcache;
5     struct mem_cgroup *memcg;
6     swp_entry_t entry;
7     pte_t pte;
8     .....
9     entry = pte_to_swp_entry(vmf->orig_pte);
10    .....
11    page = lookup_swap_cache(entry);
12    if (!page) {
13        page = swapin_readahead(entry, GFP_HIGHUSER_MOVABLE, vma,
14                                vmf->address);
15    .....
16    }
17    .....
18    swapcache = page;
19    .....
20    pte = mk_pte(page, vma->vm_page_prot);
21    .....
22    set_pte_at(vma->vm_mm, vmf->address, vmf->pte, pte);

```


```

23         vmf->orig_pte = pte;
24         .....
25         swap_free(entry);
26         .....
27     }

```

do\_swap\_page 函数会先查找 swap 文件有没有缓存页。如果没有，就调用 swapin\_readahead，将 swap 文件读到内存中来，形成内存页，并通过 mk\_pte 生成页表项。set\_pte\_at 将页表项插入页表，swap\_free 将 swap 文件清理。因为重新加载回内存了，不再需要 swap 文件了。

swapin\_readahead 会最终调用 swap\_readpage，在这里，我们看到了熟悉的 readpage 函数，也就是说读取普通文件和读取 swap 文件，过程是一样的，同样需要用 kmap\_atomic 做临时映射。

 复制代码

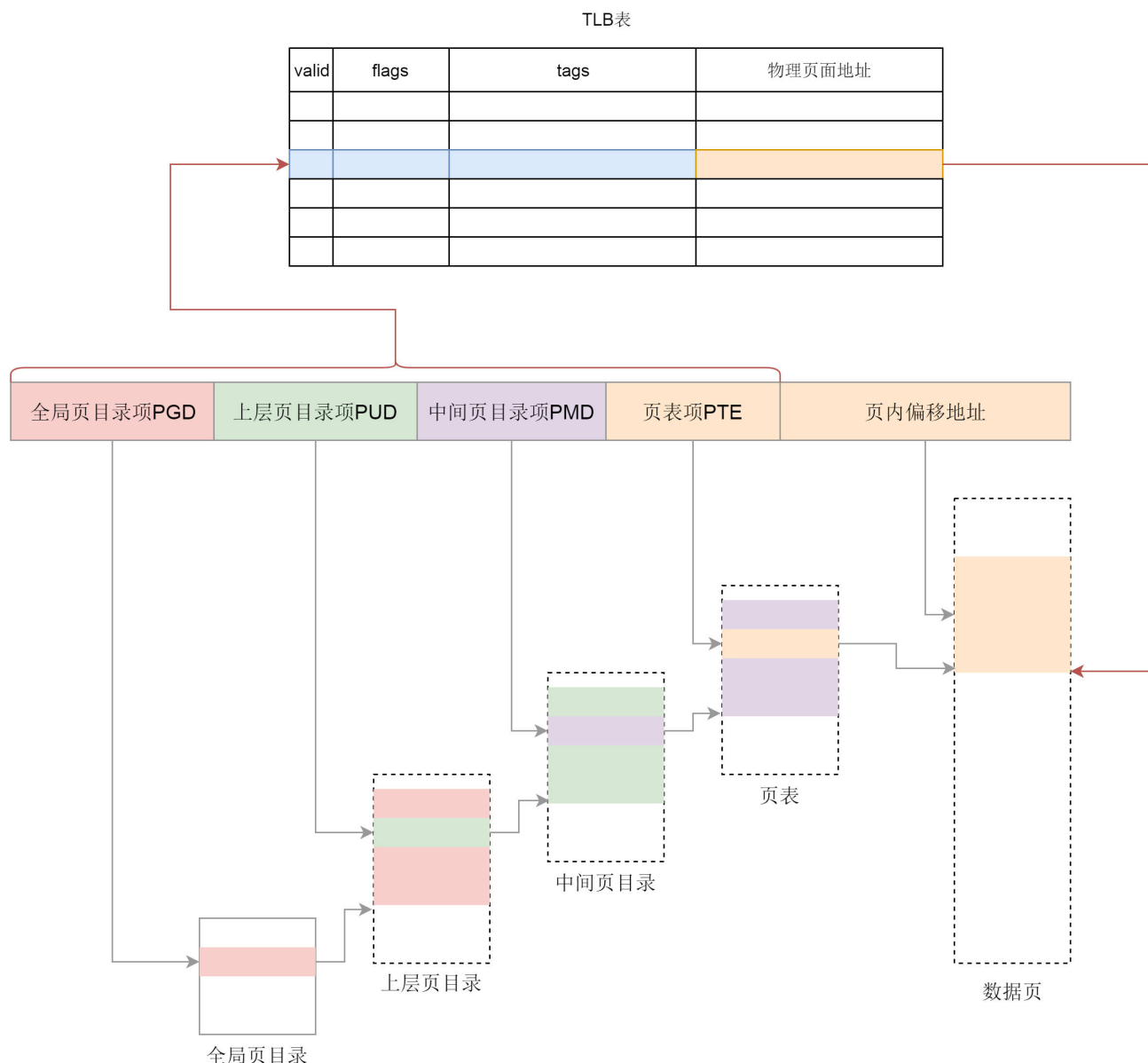
```

1 int swap_readpage(struct page *page, bool do_poll)
2 {
3     struct bio *bio;
4     int ret = 0;
5     struct swap_info_struct *sis = page_swap_info(page);
6     blk_qc_t qc;
7     struct block_device *bdev;
8     .....
9     if (sis->flags & SWP_FILE) {
10         struct file *swap_file = sis->swap_file;
11         struct address_space *mapping = swap_file->f_mapping;
12         ret = mapping->a_ops->readpage(swap_file, page);
13         return ret;
14     }
15     .....
16 }

```

通过上面复杂的过程，用户态缺页异常处理完毕了。物理内存中有了页面，页表也建立好了映射。接下来，用户程序在虚拟内存空间里面，可以通过虚拟地址顺利经过页表映射的访问物理页面上的数据了。

为了加快映射速度，我们不需要每次从虚拟地址到物理地址的转换都走一遍页表。



页表一般都很大，只能存放在内存中。操作系统每次访问内存都要折腾两步，先通过查询页表得到物理地址，然后访问该物理地址读取指令、数据。

为了提高映射速度，我们引入了**TLB**（Translation Lookaside Buffer），我们经常称为**快表**，专门用来做地址映射的硬件设备。它不在内存中，可存储的数据比较少，但是比内存要快。所以，我们可以想象，TLB 就是页表的 Cache，其中存储了当前最可能被访问到的页表项，其内容是部分页表项的一个副本。

有了 TLB 之后，地址映射的过程就像图中画的。我们先查块表，块表中有映射关系，然后直接转换为物理地址。如果在 TLB 查不到映射关系时，才会到内存中查询页表。

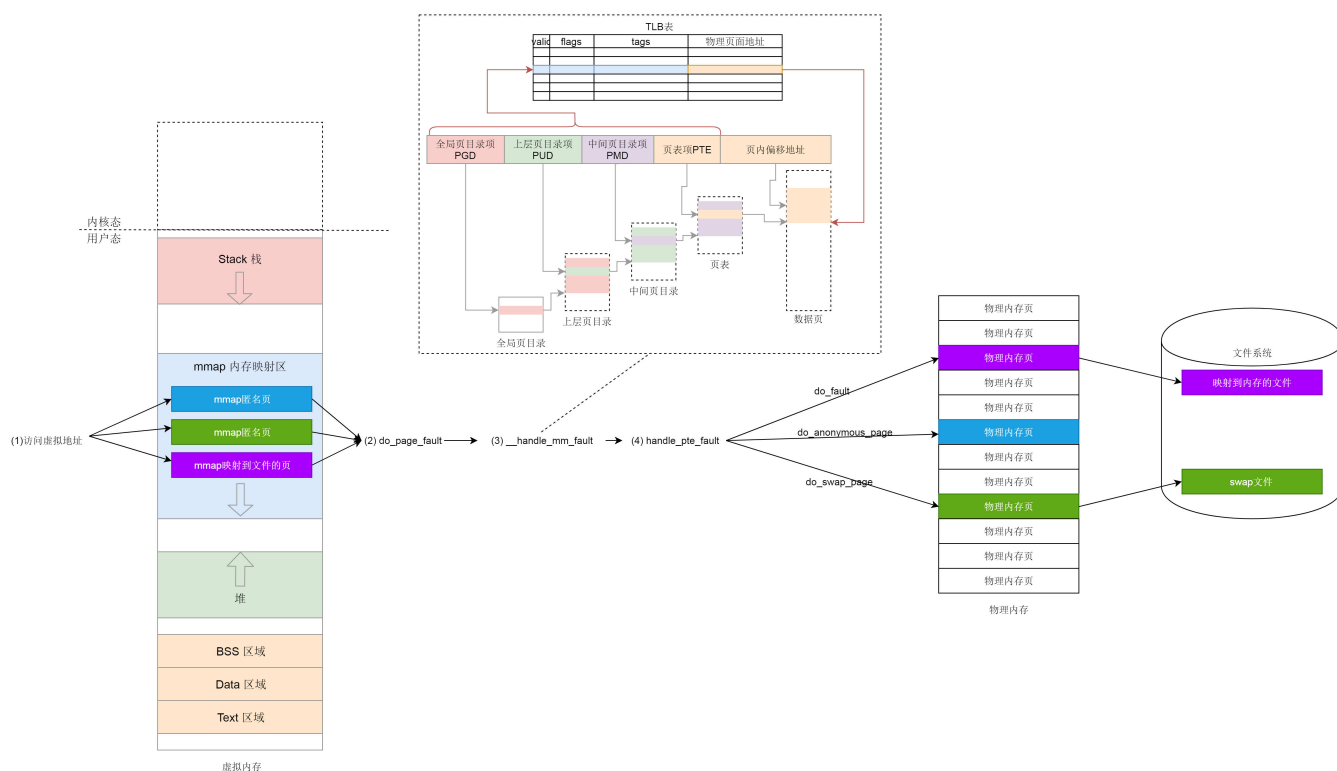
## 总结时刻

用户态的内存映射机制，我们解析的差不多了，我们来总结一下，用户态的内存映射机制包含以下几个部分。

用户态内存映射函数 `mmap`，包括用它来做匿名映射和文件映射。

用户态的页表结构，存储位置在 mm struct 中。

在用户态访问没有映射的内存会引发缺页异常，分配物理页表、补齐页表。如果是匿名映射则分配物理内存；如果是 swap，则将 swap 文件读入；如果是文件映射，则将文件读入。



## 课堂练习

你可以试着用 `mmap` 系统调用，写一个程序来映射一个文件，并读取文件的内容。

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 24 | 物理内存管理（下）：会议室管理员如何分配会议室？

下一篇 26 | 内核态内存映射：如何找到正确的会议室？

## 精选留言 (11)

写留言



why

2019-05-25

9

- 申请小块内存用 brk; 申请大块内存或文件映射用 mmap
- mmap 映射文件, 由 fd 得到 struct file
  - 调用 ...->do\_mmap
    - 调用 get\_unmapped\_area 找到一个可以进行映射的 vm\_area\_struct
    - 调用 mmap\_region 进行映射...

展开 ▾



zzuse

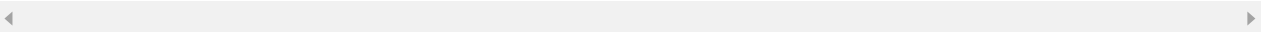
2019-05-24

6

我感觉学得很吃力，调用链太长了

展开 ▾

作者回复: 忽略调用链, 记住重点节点, 调用链就是为了证明的确这样过去的



**活的潇洒**

2019-05-29

👍 1

比起《深入浅出计算机组成原理》和《Linux性能优化实战》的篇幅  
本节花了三天, 每天不少于2小时, 才把笔记做完, 估计老师也花费不少时间  
day25笔记: <https://www.cnblogs.com/luoahong/p/10916458.html>

展开 ▾



**安排**

2019-05-24

👍 1

打卡, 通俗易懂

展开 ▾



**Geek\_49fbe...**

2019-06-01

👍

老师, 我们平时说的pss应该是指已经分配给进程的物理页面大小的总和吧? 那如果运行中有部分页面被swap到了硬盘, 此时的pss还把这部分大小算进去吗?

展开 ▾



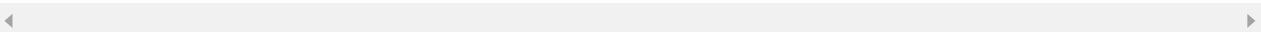
**玉剑冰锋**

2019-05-28

👍

分配全局页目录项, 赋值给mm\_struct的pdg成员变量。这里应该是pgd吧老师?

作者回复: Page Global Directory, PGD, 是的, 老是倒



**LDxy**

2019-05-26

👍

请问老师, 内核里面这些复杂的机制的实现, 在当初软件开发开始前有详细的设计文档的

吗？分布在全球各地的开发者是如何能达成这种复杂设计的共识的呢？这些内核里的函数相互依赖又和底层硬件相关，是如何进行单元测试的呢？

作者回复: 可以参考一下开源软件的运作模式，要写设计，大牛review，通过后写代码，大牛组成委员会，看够不够资格合并进去，要合并进去就要有相应的测试用例，覆盖率等，有邮件列表，实时对话工具



**kdb\_reboot**

2019-05-26



又开始跟这个专栏了 因为感觉内容还是有料的；一个建议：在讲解每一章的时候 可否列出参考资料 或者推荐资料 或者推荐阅读的章节？有证可查 也可以互相参考

展开 ∨



**微秒**

2019-05-26



老师，我觉得你这里说了好多地方出现了没有修饰的内存字眼，麻烦你写具体的物理内存或者虚拟内存，不然看得云里雾里的

作者回复: 在行文中，会强调的



**卫江**

2019-05-24



老师，想问一下，中断和异常有什么区别

展开 ∨

作者回复: 有的异常会产生中断，有的异常是应用层的，可以不产生中断



**一笔一画**

2019-05-24



请教下老师，内核线程的task struct上的mm为什么为空？另外看代码还有个active\_mm，这个设计上有什么考虑吗？

