



下载APP

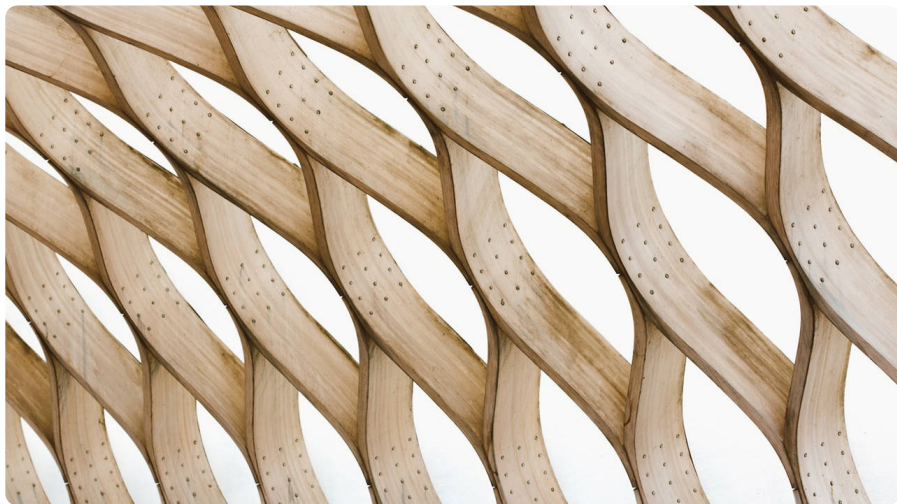


41 | IPC（中）：不同项目组之间抢资源，如何协调？

2019-07-01 刘超

趣谈Linux操作系统

[进入课程 >](#)



讲述：刘超

时长 17:32 大小 16.06M




了解了如何使用共享内存和信号量集合之后，今天我们来解析一下，内核里面都做了什么。

不知道你有没有注意到，咱们讲消息队列、共享内存、信号量的机制的时候，我们其实能够从中看到一些统一的规律：**它们在使用之前都要生成 key，然后通过 key 得到唯一的 id，并且都是通过 xxxget 函数。**

在内核里面，这三种进程间通信机制是使用统一的机制管理起来的，都叫 ipcxxx。

为了维护这三种进程间通信进制，在内核里面，我们声明了一个有三项的数组。

我们通过这段代码，来具体看一看。

 复制代码


```
1 struct ipc_namespace {
2     .....
3     struct ipc_ids  ids[3];
4     .....
5 }
6
7 #define IPC_SEM_IDS      0
8 #define IPC_MSG_IDS     1
9 #define IPC_SHM_IDS     2
10
11 #define sem_ids(ns)      ((ns)->ids[IPC_SEM_IDS])
12 #define msg_ids(ns)     ((ns)->ids[IPC_MSG_IDS])
13 #define shm_ids(ns)     ((ns)->ids[IPC_SHM_IDS])
```

根据代码中的定义，第 0 项用于信号量，第 1 项用于消息队列，第 2 项用于共享内存，分别可以通过 `sem_ids`、`msg_ids`、`shm_ids` 来访问。

这段代码里面有 `ns`，全称叫 `namespace`。可能不容易理解，你现在可以将它认为是将一台 Linux 服务器逻辑的隔离为多台 Linux 服务器的机制，它背后的原理是一个相当大的话题，我们需要在容器那一章详细讲述。现在，你就可以简单的认为没有 `namespace`，整个 Linux 在一个 `namespace` 下面，那这些 `ids` 也是整个 Linux 只有一份。

接下来，我们再来看 `struct ipc_ids` 里面保存了什么。

首先，`in_use` 表示当前有多少个 `ipc`；其次，`seq` 和 `next_id` 用于一起生成 `ipc` 唯一的 `id`，因为信号量，共享内存，消息队列，它们三个的 `id` 也不能重复；`ipcs_idr` 是一棵基数树，我们又碰到它了，一旦涉及从一个整数查找一个对象，它都是最好的选择。


 复制代码

```
1 struct ipc_ids {
2     int in_use;
3     unsigned short seq;
4     struct rw_semaphore rwsem;
```

```
5         struct idr ipcs_idr;
6         int next_id;
7     };
8
9     struct idr {
10         struct radix_tree_root   idr_rt;
11         unsigned int              idr_next;
12     };
```


也就是说，对于 `sem_ids`、`msg_ids`、`shm_ids` 各有一棵基数树。那这棵树里面究竟存放了什么，能够统一管理这三类 `ipc` 对象呢？

通过下面这个函数 `ipc_obtain_object_idr`，我们可以看出端倪。这个函数根据 `id`，在基数树里面找出来的是 `struct kern_ipc_perm`。

 复制代码

```
1 struct kern_ipc_perm *ipc_obtain_object_idr(struct ipc_
2 {
3     struct kern_ipc_perm *out;
4     int lid = ipcid_to_idx(id);
5     out = idr_find(&ids->ipcs_idr, lid);
6     return out;
7 }
```

如果我们看用于表示信号量、消息队列、共享内存的结构，就会发现，这三个结构的第一项都是 struct kern_ipc_perm。

 复制代码


```
1 struct sem_array {
2     struct kern_ipc_perm    sem_perm;        /* perm
3     time_t                  sem_ctime;        /* crea
4     struct list_head        pending_alter;    /* penc
5
6     struct list_head        pending_const;    /* penc
7                                           /* that
8     struct list_head        list_id;          /* undc
9     int                      sem_nsems;       /* no.
10    int                      complex_count;    /* penc
11    unsigned int              use_global_lock; /* >0:
12
13    struct sem                sems[];
14 } __randomize_layout;
15
16 struct msg_queue {
17     struct kern_ipc_perm q_perm;
18     time_t q_stime;        /* last msgsnd
19     time_t q_rtime;        /* last msgrcv
20     time_t q_ctime;        /* last change
21     unsigned long q_cbytes; /* current numb
22     unsigned long q_qnum;   /* number of me
23     unsigned long q_qbytes; /* max number c
24     pid_t q_lspid;          /* pid of last
25     pid_t q_lrpid;          /* last receive
26
27     struct list_head q_messages;
28     struct list_head q_receivers;
```

```

29         struct list_head q_senders;
30     } __randomize_layout;
31
32     struct shmid_kernel /* private to the kernel */
33     {
34         struct kern_ipc_perm    shm_perm;
35         struct file             *shm_file;
36         unsigned long           shm_nattch;
37         unsigned long           shm_segsz;
38         time_t                  shm_atim;
39         time_t                  shm_dtim;
40         time_t                  shm_ctim;
41         pid_t                   shm_cprid;
42         pid_t                   shm_lprid;
43         struct user_struct      *mlock_user;
44
45         /* The task created the shm object.  NULL if th
46         struct task_struct      *shm_creator;
47         struct list_head        shm_clist;          /* list
48     } __randomize_layout;

```

也就是说，我们完全可以通过 `struct kern_ipc_perm` 的指针，通过进行强制类型转换后，得到整个结构。做这件事情的函数如下：

 复制代码

```

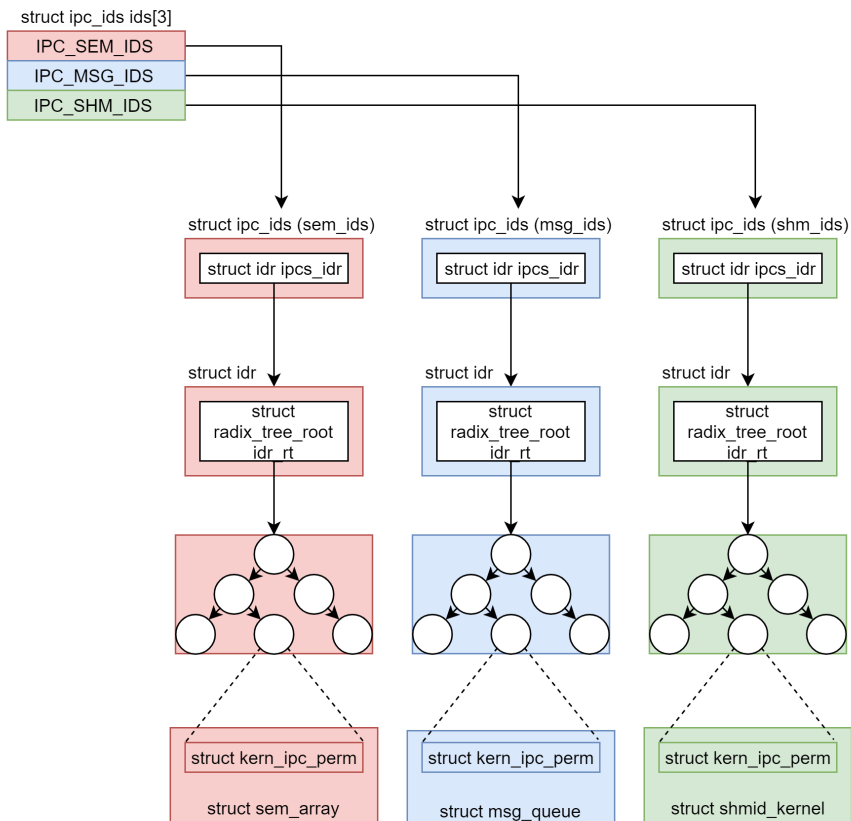
1 static inline struct sem_array *sem_obtain_object(struc
2 {
3     struct kern_ipc_perm *ipcp = ipc_obtain_object_
4     return container_of(ipcp, struct sem_array, sem

```

```
5 }
6
7 static inline struct msg_queue *msq_obtain_object(struct
8 {
9     struct kern_ipc_perm *ipcp = ipc_obtain_object_
10     return container_of(ipcp, struct msg_queue, q_p
11 }
12
13 static inline struct shmid_kernel *shm_obtain_object(st
14 {
15     struct kern_ipc_perm *ipcp = ipc_obtain_object_
16     return container_of(ipcp, struct shmid_kernel,
17 }
```




通过这种机制，我们就可以将信号量、消息队列、共享内存抽象为 ipc 类型进行统一处理。你有没有觉得，这有点儿面向对象编程中抽象类和实现类的意思？没错，如果你试图去了解 C++ 中类的实现机制，其实也是这么干的。



有了抽象类，接下来我们来看共享内存和信号量的具体实现。

如何创建共享内存？

首先，我们来看创建共享内存的系统调用。

 复制代码

```
1 SYSCALL_DEFINE3(shmget, key_t, key, size_t, size, int,
```




```

2 {
3     struct ipc_namespace *ns;
4     static const struct ipc_ops shm_ops = {
5         .getnew = newseg,
6         .associate = shm_security,
7         .more_checks = shm_more_checks,
8     };
9     struct ipc_params shm_params;
10    ns = current->nsproxy->ipc_ns;
11    shm_params.key = key;
12    shm_params.flg = shmflg;
13    shm_params.u.size = size;
14    return ipcget(ns, &shm_ids(ns), &shm_ops, &shm_
15 }

```

这里面调用了抽象的 `ipcget`、参数分别为共享内存对应的 `shm_ids`、对应的操作 `shm_ops` 以及对应的参数 `shm_params`。

如果 `key` 设置为 `IPC_PRIVATE` 则永远创建新的，如果不是的话，就会调用 `ipcget_public`。 `ipcget` 的具体代码如下：

 复制代码

```

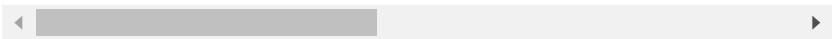
1 int ipcget(struct ipc_namespace *ns, struct ipc_ids *ids
2             const struct ipc_ops *ops, struct ipc_params *params)
3 {
4     if (params->key == IPC_PRIVATE)
5         return ipcget_new(ns, ids, ops, params);
6     else

```

```


7             return ipcget_public(ns, ids, ops, para
8 }
9
10 static int ipcget_public(struct ipc_namespace *ns, stru
11 {
12     struct kern_ipc_perm *ipcp;
13     int flg = params->flg;
14     int err;
15     ipcp = ipc_findkey(ids, params->key);
16     if (ipcp == NULL) {
17         if (!(flg & IPC_CREAT))
18             err = -ENOENT;
19         else
20             err = ops->getnew(ns, params);
21     } else {
22         if (flg & IPC_CREAT && flg & IPC_EXCL)
23             err = -EEXIST;
24         else {
25             err = 0;
26             if (ops->more_checks)
27                 err = ops->more_checks(
28 .....
29             }
30         }
31     return err;
32 }

```



在 ipcget_public 中，我们会按照 key，去查找 struct kern_ipc_perm。如果没有找到，那就看是否设置了 IPC_CREAT；如果设置了，就创建一个新的。如果找到了，就将对应的 id 返回。

我们这里重点看，如何按照参数 `shm_ops`，创建新的共享内存，会调用 `newseg`。

 复制代码

```
1 static int newseg(struct ipc_namespace *ns, struct ipc_
2 {
3     key_t key = params->key;
4     int shmflg = params->flg;
5     size_t size = params->u.size;
6     int error;
7     struct shmid_kernel *shp;
8     size_t numpages = (size + PAGE_SIZE - 1) >> PAG
9     struct file *file;
10    char name[13];
11    vm_flags_t acctflag = 0;
12    .....
13    shp = kvmalloc(sizeof(*shp), GFP_KERNEL);
14    .....
15    shp->shm_perm.key = key;
16    shp->shm_perm.mode = (shmflg & S_IRWXUGO);
17    shp->mlock_user = NULL;
18
19    shp->shm_perm.security = NULL;
20    .....
21    file = shmem_kernel_file_setup(name, size, acct
22    .....
23    shp->shm_cprid = task_tgid_vnr(current);
24    shp->shm_lprid = 0;
25    shp->shm_atim = shp->shm_dtim = 0;
26    shp->shm_ctim = get_seconds();
27    shp->shm_segsz = size;
28    shp->shm_nattch = 0;
29    shp->shm_file = file;
```

```
30         shp->shm_creator = current;
31
32         error = ipc_addid(&shm_ids(ns), &shp->shm_perm,
33         .....
34         list_add(&shp->shm_clist, &current->sysvshm.shr
35         .....
36         file_inode(file)->i_ino = shp->shm_perm.id;
37
38         ns->shm_tot += numpages;
39         error = shp->shm_perm.id;
40         .....
41         return error;
42     }
```




newseg 函数的第一步，通过 kvmalloc 在直接映射区分配一个 struct shmid_kernel 结构。这个结构就是用来描述共享内存的。这个结构最开始就是上面说的 struct kern_ipc_perm 结构。接下来就是填充这个 struct shmid_kernel 结构，例如 key、权限等。

newseg 函数的第二步，共享内存需要和文件进行关联。** 为什么要做这个呢？我们在讲内存映射的时候讲过，虚拟地址空间可以和物理内存关联，但是物理内存是某个进程独享的。虚拟地址空间也可以映射到一个文件，文件是可以跨进程共享的。


咱们这里的共享内存需要跨进程共享，也应该借鉴文件映射的思路。只不过不应该映射一个硬盘上的文件，而是映射到一个内存文件系统上的文件。mm/shmem.c 里面就定义了这样一个基于内存的文件系统。这里你一定要注意区分 shmem 和 shm 的区别，前者是一个文件系统，后者是进程通信机制。

在系统初始化的时候，shmem_init 注册了 shmem 文件系统 shmem_fs_type，并且挂在到了 shm_mnt 下面。

 复制代码

```
1 int __init shmem_init(void)
2 {
3     int error;
4     error = shmem_init_inodecache();
5     error = register_filesystem(&shmem_fs_type);
6     shm_mnt = kern_mount(&shmem_fs_type);
7     .....
8     return 0;
9 }
10
11 static struct file_system_type shmem_fs_type = {
12     .owner          = THIS_MODULE,
13     .name           = "tmpfs",
14     .mount          = shmem_mount,
15     .kill_sb        = kill_litter_super,
16     .fs_flags       = FS_USERSNS_MOUNT,
17 };
```

接下来, `newseg` 函数会调用 `shmem_kernel_file_setup`, 其实就是在 `shmem` 文件系统里面创建一个文件。

 复制代码


```
1  /**
2   * shmem_kernel_file_setup - get an unlinked file livir
3   * @name: name for dentry (to be seen in /proc/<pid>/ma
4   * @size: size to be set for the file
5   * @flags: VM_NORESERVE suppresses pre-accounting of th
6   struct file *shmem_kernel_file_setup(const char *name,
7   {
8       return __shmem_file_setup(name, size, flags, S_
9   }
10
11 static struct file *__shmem_file_setup(const char *name
12                                     unsigned long fl
13 {
14     struct file *res;
15     struct inode *inode;
16     struct path path;
17     struct super_block *sb;
18     struct qstr this;
19     .....
20     this.name = name;
21     this.len = strlen(name);
22     this.hash = 0; /* will go */
23     sb = shm_mnt->mnt_sb;
24     path.mnt = mntget(shm_mnt);
25     path.dentry = d_alloc_pseudo(sb, &this);
26     d_set_d_op(path.dentry, &anon_ops);
27     .....
28     inode = shmem_get_inode(sb, NULL, S_IFREG | S_I
29     inode->i_flags |= i_flags;
```

```

30         d_instantiate(path.dentry, inode);
31         inode->i_size = size;
32         .....
33         res = alloc_file(&path, FMODE_WRITE | FMODE_READ,
34                         &shmem_file_operations);
35         return res;
36     }

```

`__shmem_file_setup` 会创建新的 `shmem` 文件对应的 `dentry` 和 `inode`，并将它们两个关联起来，然后分配一个 `struct file` 结构，来表示新的 `shmem` 文件，并且指向独特的 `shmem_file_operations`。

 复制代码

```

1 static const struct file_operations shmem_file_operations = {
2     .mmap                = shmem_mmap,
3     .get_unmapped_area   = shmem_get_unmapped_area,
4 #ifdef CONFIG_TMPFS
5     .llseek              = shmem_file_llseek,
6     .read_iter           = shmem_file_read_iter,
7     .write_iter          = generic_file_write_iter,
8     .fsync               = noop_fsync,
9     .splice_read         = generic_file_splice_read,
10    .splice_write        = iter_file_splice_write,
11    .fallocate            = shmem_fallocate,
12 #endif
13 };

```


newseg 函数的第三步，通过 ipc_addid 将新创建的 struct shmid_kernel 结构挂到 shm_ids 里面的基数树上，并返回相应的 id，并且将 struct shmid_kernel 挂到当前进程的 sysvshm 队列中。

至此，共享内存的创建就完成了。

如何将共享内存映射到虚拟地址空间？

从上面的代码解析中，我们知道，共享内存的数据结构 struct shmid_kernel，是通过它的成员 struct file *shm_file，来管理内存文件系统 shmem 上的内存文件的。无论这个共享内存是否被映射，shm_file 都是存在的。

接下来，我们要将共享内存映射到虚拟地址空间中。调用的是 shmat，对应的系统调用如下：

 复制代码

```
1 SYSCALL_DEFINE3(shmat, int, shmid, char __user *, shmac
2 {
3     unsigned long ret;
4     long err;
5     err = do_shmat(shmid, shmaddr, shmflg, &ret, SHMLBA
6     force_successful_syscall_return();
7     return (long)ret;
8 }
9
```



```

10 long do_shmat(int shmid, char __user *shmaddr, int shmf
11               ulong *raddr, unsigned long shmlba)
12 {
13     struct shmid_kernel *shp;
14     unsigned long addr = (unsigned long)shmaddr;
15     unsigned long size;
16     struct file *file;
17     int err;
18     unsigned long flags = MAP_SHARED;
19     unsigned long prot;
20     int acc_mode;
21     struct ipc_namespace *ns;
22     struct shm_file_data *sfd;
23     struct path path;
24     fmode_t f_mode;
25     unsigned long populate = 0;
26     .....
27     prot = PROT_READ | PROT_WRITE;
28     acc_mode = S_IRUGO | S_IWUGO;
29     f_mode = FMODE_READ | FMODE_WRITE;
30     .....
31     ns = current->nsproxy->ipc_ns;
32     shp = shm_obtain_object_check(ns, shmid);
33     .....
34     path = shp->shm_file->f_path;
35     path_get(&path);
36     shp->shm_nattch++;
37     size = i_size_read(d_inode(path.dentry));
38     .....
39     sfd = kzalloc(sizeof(*sfd), GFP_KERNEL);
40     .....
41     file = alloc_file(&path, f_mode,
42                      is_file_hugepages(shp->shm_fi
43                      &shm_file_operations_hu
44                      &shm_file_operations));

```

```
45 .....
46     file->private_data = sfd;
47     file->f_mapping = shp->shm_file->f_mapping;
48     sfd->id = shp->shm_perm.id;
49     sfd->ns = get_ipc_ns(ns);
50     sfd->file = shp->shm_file;
51     sfd->vm_ops = NULL;
52 .....
53     addr = do_mmap_pgoff(file, addr, size, prot, fl
54     *raddr = addr;
55     err = 0;
56 .....
57     return err;
58 }
```



在这个函数里面，`shm_obtain_object_check` 会通过共享内存的 `id`，在基数树中找到对应的 `struct shmid_kernel` 结构，通过它找到 `shmem` 上的内存文件。


接下来，我们要分配一个 `struct shm_file_data`，来表示这个内存文件。将 `shmem` 中指向内存文件的 `shm_file` 赋值给 `struct shm_file_data` 中的 `file` 成员。

然后，我们创建了一个 `struct file`，指向的也是 `shmem` 中的内存文件。

为什么要再创建一个呢？这两个的功能不同，shmem 中 shm_file 用于管理内存文件，是一个中立的，独立于任何一个进程的角色。而新创建的 struct file 是专门用于做内存映射的，就像咱们在讲内存映射那一节讲过的，一个硬盘上的文件要映射到虚拟地址空间中的时候，需要在 vm_area_struct 里面有一个 struct file *vm_file 指向硬盘上的文件，现在变成内存文件了，但是这个结构还是不能少。


新创建的 struct file 的 private_data，指向 struct shm_file_data，这样内存映射那部分的数据结构，就能够通过它来访问内存文件了。

新创建的 struct file 的 file_operations 也发生了变化，变成了 shm_file_operations。

 复制代码


```
1 static const struct file_operations shm_file_operations
2     .mmap          = shm_mmap,
3     .fsync         = shm_fsync,
4     .release       = shm_release,
5     .get_unmapped_area = shm_get_unmapped_area
6     .llseek        = noop_llseek,
7     .fallocate     = shm_fallocate,
8 };
```

接下来，do_mmap_pgoff 函数我们遇到过，原来映射硬盘上的文件的时候，也是调用它。这里我们不再详细解析了。它会分配一个 vm_area_struct 指向虚拟地址空间中没有分配的区域，它的 vm_file 指向这个内存文件，然后它会调用 shm_file_operations 的 mmap 函数，也即 shm_mmap 进行映射。

 复制代码

```
1 static int shm_mmap(struct file *file, struct vm_area_s
2 {
3     struct shm_file_data *sfd = shm_file_data(file)
4     int ret;
5     ret = __shm_open(vma);
6     ret = call_mmap(sfd->file, vma);
7     sfd->vm_ops = vma->vm_ops;
8     vma->vm_ops = &shm_vm_ops;
9     return 0;
10 }
```

shm_mmap 中调用了 shm_file_data 中的 file 的 mmap 函数，这次调用的是 shmem_file_operations 的 mmap，也即 shmem_mmap。


 复制代码

```
1 static int shmem_mmap(struct file *file, struct vm_area
2 {
```

```
3         file_accessed(file);
4         vma->vm_ops = &shmem_vm_ops;
5         return 0;
6     }
```

这里面，`vm_area_struct` 的 `vm_ops` 指向 `shmem_vm_ops`。等从 `call_mmap` 中返回之后，`shm_file_data` 的 `vm_ops` 指向了 `shmem_vm_ops`，而 `vm_area_struct` 的 `vm_ops` 改为指向 `shm_vm_ops`。


我们来看一下，`shm_vm_ops` 和 `shmem_vm_ops` 的定义。

 复制代码

```
1 static const struct vm_operations_struct shm_vm_ops = {
2     .open    = shm_open,      /* callback for a new v
3     .close   = shm_close,     /* callback for when th
4     .fault   = shm_fault,
5 };
6
7 static const struct vm_operations_struct shmem_vm_ops =
8     .fault           = shmem_fault,
9     .map_pages       = filemap_map_pages,
10 };
```


它们里面最关键的就是 `fault` 函数，也即访问虚拟内存的时候，访问不到应该怎么办。

当访问不到的时候，先调用 `vm_area_struct` 的 `vm_ops`，也即 `shm_vm_ops` 的 `fault` 函数 `shm_fault`。然后它会转而调用 `shm_file_data` 的 `vm_ops`，也即 `shmem_vm_ops` 的 `fault` 函数 `shmem_fault`。

 复制代码

```
1 static int shm_fault(struct vm_fault *vmf)
2 {
3     struct file *file = vmf->vma->vm_file;
4     struct shm_file_data *sfd = shm_file_data(file)
5     return sfd->vm_ops->fault(vmf);
6 }
```

虽然基于内存的文件系统，已经为这个内存文件分配了 `inode`，但是内存也却是一点儿都没分配，只有在发生缺页异常的时候才进行分配。

 复制代码

```
1 static int shmem_fault(struct vm_fault *vmf)
2 {
3     struct vm_area_struct *vma = vmf->vma;
4     struct inode *inode = file_inode(vma->vm_file);
5     gfp_t gfp = mapping_gfp_mask(inode->i_mapping);
```

```

6 .....
7         error = shmem_getpage_gfp(inode, vmf->pgoff, &v
8                                     gfp, vma, vmf, &ret);
9 .....
10 }
11
12 /*
13  * shmem_getpage_gfp - find page in cache, or get from
14  *
15  * If we allocate a new one we do not mark it dirty. Th
16  * vm. If we swap it in we mark it dirty since we also
17  * entry since a page cannot live in both the swap and
18  *
19  * fault_mm and fault_type are only supplied by shmem_f
20  * otherwise they are NULL.
21  */
22 static int shmem_getpage_gfp(struct inode *inode, pgoff
23                             struct page **pagep, enum sgp_type sgp, gfp_t g
24                             struct vm_area_struct *vma, struct vm_fault *vm
25 {
26 .....
27     page = shmem_alloc_and_acct_page(gfp, info, sbinfo,
28                                     index, false);
29 .....
30 }

```

shmem_fault 会调用 shmem_getpage_gfp 在 page cache 和 swap 中找一个空闲页，如果找不到就通过 shmem_alloc_and_acct_page 分配一个新的页，他最终会

调用内存管理系统的 `alloc_page_vma` 在物理内存中分配一个页。

至此，共享内存才真的映射到了虚拟地址空间中，进程可以像访问本地内存一样访问共享内存。

总结时刻

我们来总结一下共享内存的创建和映射过程。

1. 调用 `shmget` 创建共享内存。
2. 先通过 `ipc_findkey` 在基数树中查找 `key` 对应的共享内存对象 `shmid_kernel` 是否已经被创建过，如果已经被创建，就会被查询出来，例如 `producer` 创建过，在 `consumer` 中就会查询出来。
3. 如果共享内存没有被创建过，则调用 `shm_ops` 的 `newseg` 方法，创建一个共享内存对象 `shmid_kernel`。例如，在 `producer` 中就会新建。
4. 在 `shmem` 文件系统里面创建一个文件，共享内存对象 `shmid_kernel` 指向这个文件，这个文件用 `struct file` 表示，我们姑且称它为 `file1`。
5. 调用 `shmat`，将共享内存映射到虚拟地址空间。
6. `shm_obtain_object_check` 先从基数树里面找到 `shmid_kernel` 对象。

7. 创建用于内存映射到文件的 file 和 shm_file_data，这里的 struct file 我们姑且称为 file2。

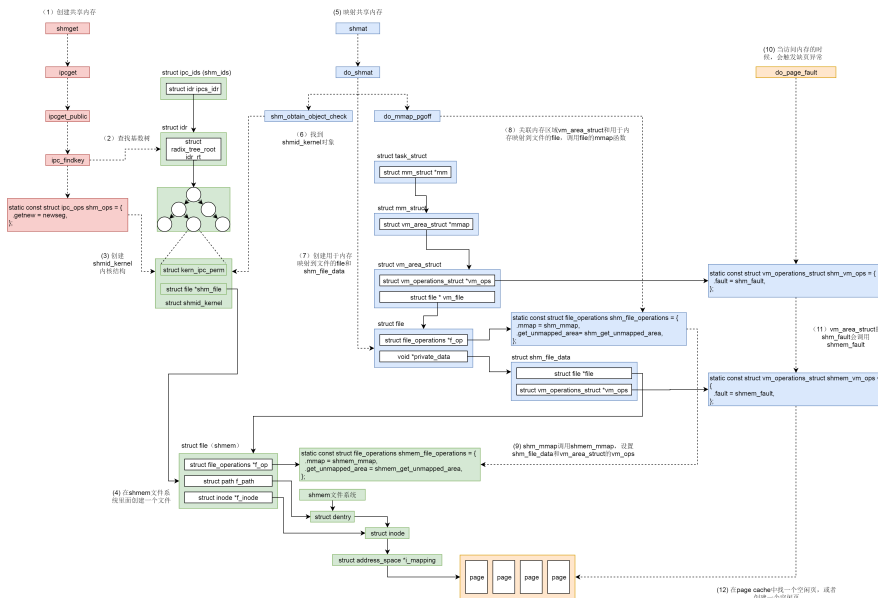
8. 关联内存区域 vm_area_struct 和用于内存映射到文件的 file，也即 file2，调用 file2 的 mmap 函数。

9. file2 的 mmap 函数 shm_mmap，会调用 file1 的 mmap 函数 shmem_mmap，设置 shm_file_data 和 vm_area_struct 的 vm_ops。

10. 内存映射完毕之后，其实并没有真的分配物理内存，当访问内存的时候，会触发缺页异常 do_page_fault。

11. vm_area_struct 的 vm_ops 的 shm_fault 会调用 shm_file_data 的 vm_ops 的 shmem_fault。

12. 在 page cache 中找一个空闲页，或者创建一个空闲页。



课堂练习

在这里，我们只分析了 `shm_ids` 的结构，消息队列的程序我们写过了，但是 `msg_ids` 的结构没有解析，你可以试着解析一下。

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

 极客时间

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 40 | IPC（上）：不同项目组之间抢资源，如何协调？

精选留言 (2)

写留言



Amark

2019-07-02

老师有没有什么通俗易懂的资料，您将的太专业了

展开



不一样的烟火

2019-07-01

听完了 快点更新

展开

作者回复: 牛

