一张图深度解析 Linux 共享内存的内核实现

扬帆 sailing_9806#163.com

http://blog.csdn.net/sailor_8318/article/details/39484747

【摘要】本文首先介绍了众所周知的共享内存用户态 API，然后介绍了相关的内核主要数据结构，并逐一分析了 shmget、shmat、数据访问、shmdt 的内核实现及数据结构之间的动态关系，从数据的关联图即可一窥共享内存的实现机制。

【关键字】共享内存，shmat， smget， mmap，shmid_kernel

1 功能

System V 共享内存作为多进程间通信的最高效手段，是因为：

1、 其将物理内存直接映射为虚拟地址，通过虚拟地址即可直接访问数据，避免了 rd/wr 等系统调用的开销
2、 其避免了 msg 及 socket 通信方式的数据拷贝过程

基本原理介绍可参考“Linux 环境进程间通信(五): 共享内存(下)”

2 示例代码

```
/***********************************************************
*实验要求：	创建两个进程，通过共享内存进行通讯。
*功能描述：	本程序申请了上一段程序相同的共享内存块，然后循环向共享中
*		写数据，直至写入“end”。
*日　　期：	2010-9-17
*作　　者：	国嵌
**********************************************************/
#include <unistd.h>
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/shm.h>
#include "shm_com.h"

/*
 * 程序入口
 * */
int main(void)
{
	int running=1;
	void *shared_memory=(void *)0;
	struct shared_use_st *shared_stuff;
	char buffer[BUFSIZ];
```

```
int shmid;
/*创建共享内存*/
shmid=shmget((key_t)1234,sizeof(struct shared_use_st),0666|IPC_CREAT);
if(shmid==-1)
{
	fprintf(stderr,"shmget failed\n");
	exit(EXIT_FAILURE);
}

/*映射共享内存*/
shared_memory=shmat(shmid,(void *)0,0);
if(shared_memory==(void *)-1)
{
	fprintf(stderr,"shmat failed\n");
	exit(EXIT_FAILURE);
}
printf("Memory attached at %X\n",(int)shared_memory);

/*让结构体指针指向这块共享内存*/
shared_stuff=(struct shared_use_st *)shared_memory;
/*循环的向共享内存中写数据，直到写入的为“end”为止*/
while(running)
{
	while(shared_stuff->written_by_you==1)
	{
		sleep(1);//等到读进程读完之后再写
		printf("waiting for client...\n");
	}
	printf("Ener some text:");
	fgets(buffer,BUFSIZ,stdin);
	strncpy(shared_stuff->some_text,buffer,TEXT_SZ);
	shared_stuff->written_by_you=1;
	if(strncmp(buffer,"end",3)==0)
	{
		running=0;	//结束循环
	}
}
/*detach 共享内存*/
if(shmdt(shared_memory)==-1)
{
	fprintf(stderr,"shmdt failed\n");
	exit(EXIT_FAILURE);
}
exit(EXIT_SUCCESS);
```

```
}

/*******************************************************
*实验要求：	创建两个进程，通过共享内存进行通讯。
*功能描述：	本程序申请和分配共享内存，然后轮训并读取共享中的数据，直至
*		读到“end”。
*日	期：	2010-9-17
*作	者：	国嵌
*******************************************************/
#include <unistd.h>
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/shm.h>
#include "shm_com.h"

/*
 * 程序入口
 * */
int main(void)
{
	int running=1;
	void *shared_memory=(void *)0;
	struct shared_use_st *shared_stuff;
	int shmid;
	/*创建共享内存*/
	shmid=shmget((key_t)1234,sizeof(struct shared_use_st),0666|IPC_CREAT);
	if(shmid==-1)
	{
		fprintf(stderr,"shmget failed\n");
		exit(EXIT_FAILURE);
	}

	/*映射共享内存*/
	shared_memory=shmat(shmid,(void *)0,0);
	if(shared_memory==(void *)-1)
	{
		fprintf(stderr,"shmat failed\n");
		exit(EXIT_FAILURE);
	}
	printf("Memory attached at %X\n",(int)shared_memory);
```

```
    /*让结构体指针指向这块共享内存*/
    shared_stuff=(struct shared_use_st *)shared_memory;

    /*控制读写顺序*/
    shared_stuff->written_by_you=0;
    /*循环的从共享内存中读数据，直到读到“end”为止*/
    while(running)
    {
        if(shared_stuff->written_by_you)
        {
            printf("You wrote:%s",shared_stuff->some_text);
            sleep(1);   //读进程睡一秒，同时会导致写进程睡一秒，这样做到读了之后再写
            shared_stuff->written_by_you=0;
            if(strncmp(shared_stuff->some_text,"end",3)==0)
            {
                running=0; //结束循环
            }
        }
    }
    /*删除共享内存*/
    if(shmdt(shared_memory)==-1)
    {
        fprintf(stderr,"shmdt failed\n");
        exit(EXIT_FAILURE);
    }
        exit(EXIT_SUCCESS);
}
```

3 主要数据结构及其关系

通过上面的示例代码我们大概了解了共享内存的用户 API，但其是如何实现的呢，让我们来一探究竟。首先介绍相关的主要数据结构。

3.1 ipc_params

该数据结构为用户空间和内核空间通信的 API，key、flg、size 为创建共享内存的必备参数

```
/*
 * Structure that holds the parameters needed by the ipc operations
 * (see after)
 */
struct ipc_params {
    key_t key;
    int flg;
    union {
```

```
        size_t size;      /* for shared memories */
        int nsems;        /* for semaphores */
    } u;                  /* holds the getnew() specific param */
};
```

3.2 shmid_kernel

shmid_kernel 一个共享内存区在内核态的 ipc 标识

```
struct shmid_kernel /* private to the kernel */
{
        struct kern_ipc_perm    shm_perm;
        struct file             *shm_file;  /* 定位共享内存在 ramfs 中的 inode */
        unsigned long           shm_nattch;  /* 被映射的次数，为 0 时才能删除此共享内存区 */
        unsigned  long          shm_segsz;  /* 为用户态传递下来的共享内存区size*/
        time_t                  shm_atim;
        time_t                  shm_dtim;
        time_t                  shm_ctim;
        pid_t                   shm_cprid;
        pid_t                   shm_lprid;
        struct user_struct      *mlock_user;

        /* The task created the shm object.  NULL if the task is dead. */
        struct task_struct      *shm_creator;
};
```

3.3 kern_ipc_perm

kern_ipc_perm 保存用户态 shm key 值和内核态的 shmid 及其他权限信息

```
/* used by in-kernel data structures */
struct kern_ipc_perm
{
        spinlock_t      lock;
        bool            deleted;
        int             id;     /* shm 的内核标识，同一个 key 多次映射的 shmid 可能不一样*/
        key_t           key;    /* 用户空间用于识别 shm 的 key 标识，该 key 标识可以静态约定或者根据某个值唯一标识，避免冲突*/
        kuid_t          uid;
        kgid_t          gid;
        kuid_t          cuid;
```

```
        kgid_t          cgid;
        umode_t         mode;
        unsigned long   seq;
        void            *security;
};
```

3.4 shm_file_data

当进程 attach 到某个共享内存区时，即建立该数据结构，后续所有操作都通过该数据结构访问到其他所有信息。

```
struct shm_file_data {
        int id;
        struct ipc_namespace *ns;
        struct file *file;
        const struct vm_operations_struct *vm_ops;
};
```

3.5 shm_file_operations

```
static const struct file_operations shm_file_operations = {
        .mmap           = shm_mmap,
        .fsync          = shm_fsync,
        .release        = shm_release,
};
```

3.6 shm_vm_ops

```
static const struct vm_operations_struct shm_vm_ops = {
        .open   = shm_open,   /* callback for a new vm-area open */
        .close  = shm_close,  /* callback for when the vm-area is released */
        .fault  = shm_fault,
};
```

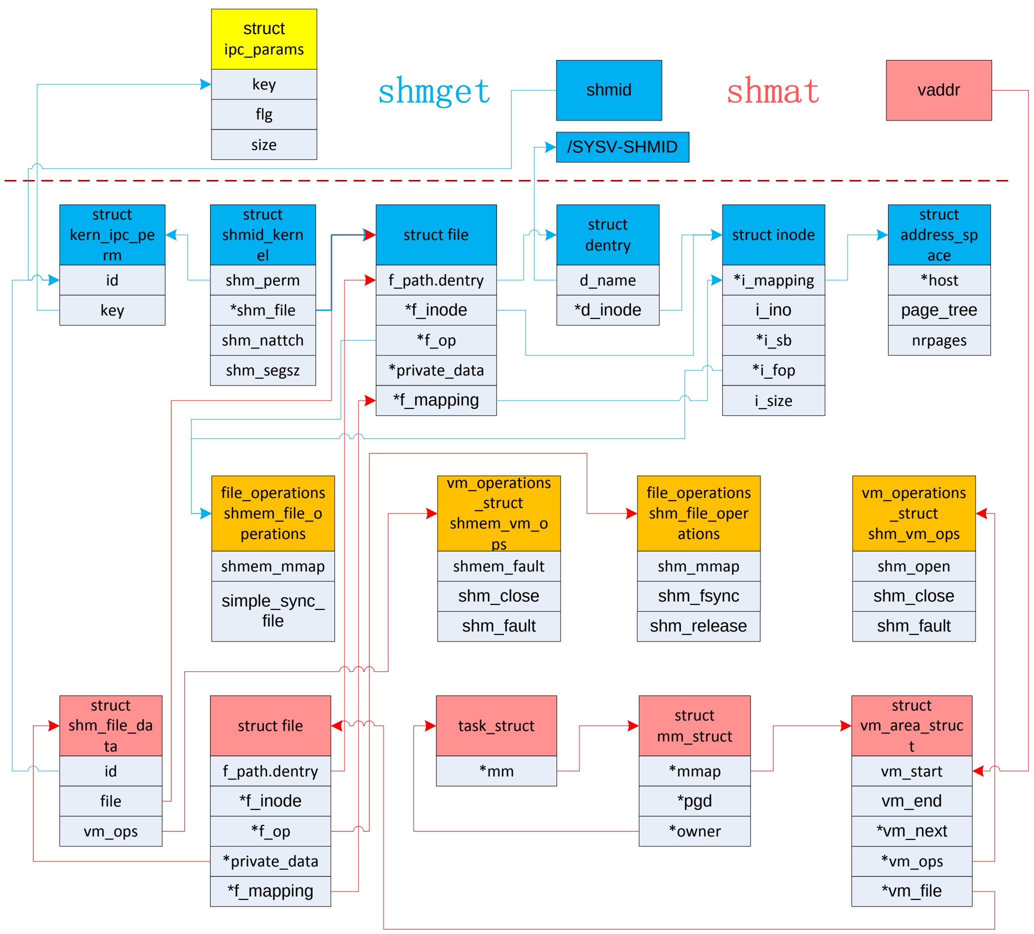
3.7 ipc_ops

```
/*
 * Structure that holds some ipc operations. This structure is used to unify
 * the calls to sys_msgget(), sys_semget(), sys_shmget()
 *      . routine to call to create a new ipc object. Can be one of newque,
 *        newary, newseg
 *      . routine to call to check permissions for a new ipc object.
 *        Can be one of security_msg_associate, security_sem_associate,
```

```
 *              security_shm_associate
 *          . routine to call for an extra check if needed
 */
struct ipc_ops {
        int (*getnew) (struct ipc_namespace *, struct ipc_params *);
        int (*associate) (struct kern_ipc_perm *, int);
        int (*more_checks) (struct kern_ipc_perm *, struct ipc_params *);
};

shm_ops.getnew = newseg;
shm_ops.associate = shm_security;
shm_ops.more_checks = shm_more_checks;
```

3.8 数据结构之间的关系

随着共享内存的建立、映射、访问等过程，最终会在建立如下的数据信息关联表，通过此表即可完全搞懂共享内存的内部原理。



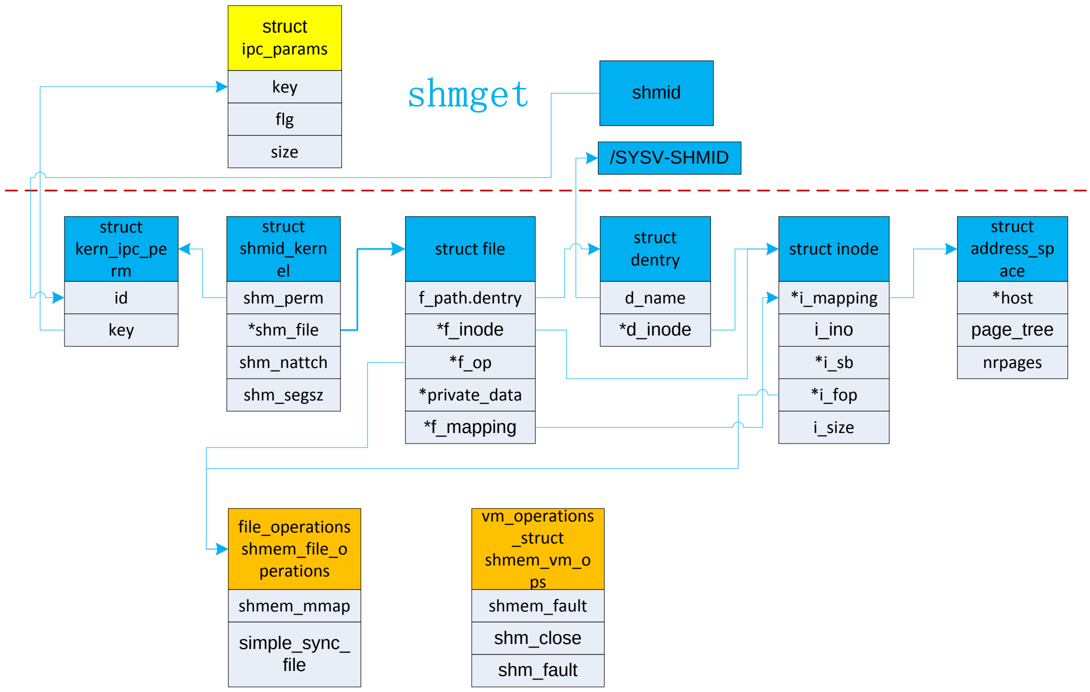
4 创建 or 打开 share memory

4.1主流程

以 key 为关键字获取 shm 信息。若在 ipc 中未创建，则在 shm 文件系统（tempfs）里分配一个 inode，其对应文件为/SYSV-shmid(用户态不可见)，并分配一个 file 文件描述符指向此 inode 的 dentry，并保存在 ipc shm 数据结构 shmid_kernel 里，并返回 shmid。若已经创建，则获取 shmid 即可。

共享内存的物理地址保存在 inode 的 struct address_space *i_mapping 域的 struct radix_tree_root page_tree; /* radix tree of all pages */成员中。共享内存也使用了 page cache 的框架来管理物理页，但并不是通过 read/write 等系统调用方式来访问共享内存“文件”。

在内核态建立的相关数据关联信息如下：



黄色是用户态的参数输入， 蓝色部分是 shmget 过程中动态建立的信息，其中 shmid 为最终返回值。

用 systemtap（可参考文章）监测到的函数调用栈信息如下：

```
shmem_alloc_inode(sb=0xf5c3ac00)
 0xc1153110 : shmem_alloc_inode+0x0/0x30 [kernel]
 0xc11a5a50 : alloc_inode+0x20/0x80 [kernel]
 0xc11a7ba6 : new_inode_pseudo+0x16/0x60 [kernel]
 0xc11a7c07 : new_inode+0x17/0x30 [kernel]
 0xc115409b : shmem_get_inode+0x2b/0x170 [kernel]
```

```
 0xc11545c4 : shmem_file_setup+0xb4/0x1b0 [kernel]
 0xc12915b9 : newseg+0x239/0x2a0 [kernel]
 0xc128dc51 : ipcget+0x111/0x1d0 [kernel]
 0xc1291cf2 : sys_shmget+0x52/0x60 [kernel]
 0xc1292b39 : sys_ipc+0x249/0x280 [kernel]
 0xc161abb4 : syscall_call+0x7/0xb [kernel]
--------------------------------------
```

4.2 Shmget

用户空间以 key 为关键字来区分不同的 share memory

```
SYSCALL_DEFINE3(shmget, key_t, key, size_t, size, int, shmflg)
{
	struct ipc_namespace *ns;
	struct ipc_ops shm_ops;
	struct ipc_params shm_params;

	ns = current->nsproxy->ipc_ns;
	shm_ops.getnew = newseg;

	shm_params.key = key;
	shm_params.flg = shmflg;
	shm_params.u.size = size;

	return ipcget(ns, &shm_ids(ns), &shm_ops, &shm_params);
}
```

4.3 ipcget_public

```
/**
 *	ipcget_public -	get an ipc object or create a new one
 *	@ns: namespace
 *	@ids: IPC identifer set
 *	@ops: the actual creation routine to call
 *	@params: its parameters
 *
 *	This routine is called by sys_msgget, sys_semget() and sys_shmget()
 *	when the key is not IPC_PRIVATE.
 *	It adds a new entry if the key is not found and does some permission
 *		/ security checkings if the key is found.
 *
 *	On success, the ipc id is returned.
 */
```

```
static int ipcget_public(struct ipc_namespace *ns, struct ipc_ids *ids,
		struct ipc_ops *ops, struct ipc_params *params)
{
	ipcp = ipc_findkey(ids, params->key);
	if (ipcp == NULL) {
		/* key not used */
		if (!(flg & IPC_CREAT))
			err = -ENOENT;
		else
			err = ops->getnew(ns, params);
	} else {
			if (ops->more_checks)
				err = ops->more_checks(ipcp, params);
	}
}
```

以 key 为关键字在现有的 share memory 实例中查找，查找失败，则 ops->getnew(ns, params) 创建一个新的 shm 实例；查找成功，做一些必要的安全性检查即可。

4.4 newseg

```
/**
 * newseg - Create a new shared memory segment
* @params: ptr to the structure that contains key, size and shmflg
*/

static int newseg(struct ipc_namespace *ns, struct ipc_params *params)
{
	key_t key = params->key;
	int shmflg = params->flg;
	size_t size = params->u.size;
	struct shmid_kernel *shp;
	int numpages = (size + PAGE_SIZE -1) >> PAGE_SHIFT;	/*  计算 shm 文件大小*/
	struct file * file;

	shp = ipc_rcu_alloc(sizeof(*shp));
	shp->shm_perm.key = key;
	shp->shm_perm.mode = (shmflg & S_IRWXUGO);

	sprintf (name, "SYSV%08x", key);	/* shm 文件名称，包含 keyid */
	file = shmem_file_setup(name, size, acctflag);	/*  在 shm 的 tempfs 中创建一个文件 inode
节点，并返回一个文件描述符，文件存在哪个路径了呢？？  是个隐藏文件，用户空间看不
到!!  */

	id = ipc_addid(&shm_ids(ns), &shp->shm_perm, ns->shm_ctlmni);
```

```
	shp->shm_segsz = size;
	shp->shm_nattch = 0;
	shp->shm_file = file;    /* 将 file 指针保存在 ipc shmid_kernel 中 shp->shm_file 中以备后用 */
	/*
	 * shmid gets reported as "inode#" in /proc/pid/maps.
	 * proc-ps tools use this. Changing this will break them.
	 */
	file->f_dentry->d_inode->i_ino = shp->shm_perm.id;    /* shm ID 作为 inode number */

	error = shp->shm_perm.id;
	return error;
}
```

4.5 shmem_file_setup

```
/**
 * shmem_file_setup - get an unlinked file living in tmpfs
 * @name: name for dentry (to be seen in /proc/<pid>/maps
 * @size: size to be set for the file
*/
struct file *shmem_file_setup(const char *name, loff_t size, unsigned long flags)
{
	int error;
	struct file *file;
	struct inode *inode;
	struct path path;
	struct dentry *root;

	error = -ENOMEM;
	this.name = name;
	this.len = strlen(name);
	root = shm_mnt->mnt_root;
	path.dentry = d_alloc(root, &this);    /* 在 shm 所 mount 文件系统根目录下创建 dentry 节点 */
	path.mnt = mntget(shm_mnt);

	inode = shmem_get_inode(root->d_sb, S_IFREG | S_IRWXUGO, 0, flags); /* 创建 inode 节点 */

	d_instantiate(path.dentry, inode); /* 将 dentry 和 inode 节点关联起来 */
	inode->i_size = size;

	file = alloc_file(&path, FMODE_WRITE | FMODE_READ,
```

```
            &shmem_file_operations);   /* 分配一个 file 文件描述符指向该 inode 节点 ，并指定该文件操作指针为 shmem_file_operations    */

        return file;

}
EXPORT_SYMBOL_GPL(shmem_file_setup);
```

4.6 alloc_file

分配一个 file 描述符，并指向参数中的 dentry 和 inode，并初始化 file operations 指针

http://lxr.free-electrons.com/source/fs/file_table.c#L166

```
/**
  * alloc_file - allocate and initialize a 'struct file'
  * @mnt: the vfsmount on which the file will reside
  * @dentry: the dentry representing the new file
  * @mode: the mode with which the new file will be opened
  * @fop: the 'struct file_operations' for the new file
*/
struct file *alloc_file(struct path *path, fmode_t mode,
                const struct file_operations *fop)
{
        struct file *file;

        file = get_empty_filp();

        file->f_path = *path;
        file->f_mapping = path->dentry->d_inode->i_mapping;
        file->f_mode = mode;
        file->f_op = fop;
}
EXPORT_SYMBOL(alloc_file);
```

4.7 用户态信息

```
drq@ubuntu:/mnt/hgfs/systemtap$ ipcs -m
------ Shared Memory Segments --------
key           shmid        owner        perms        bytes        nattch        status
0x000004d2 32768         drq          666          2052         0

drq@ubuntu:/mnt/hgfs/systemtap/share-m$ cat /proc/sysvipc/shm
       key        shmid  perms          size    cpid   lpid  nattch     uid     gid    cuid    cgid
```

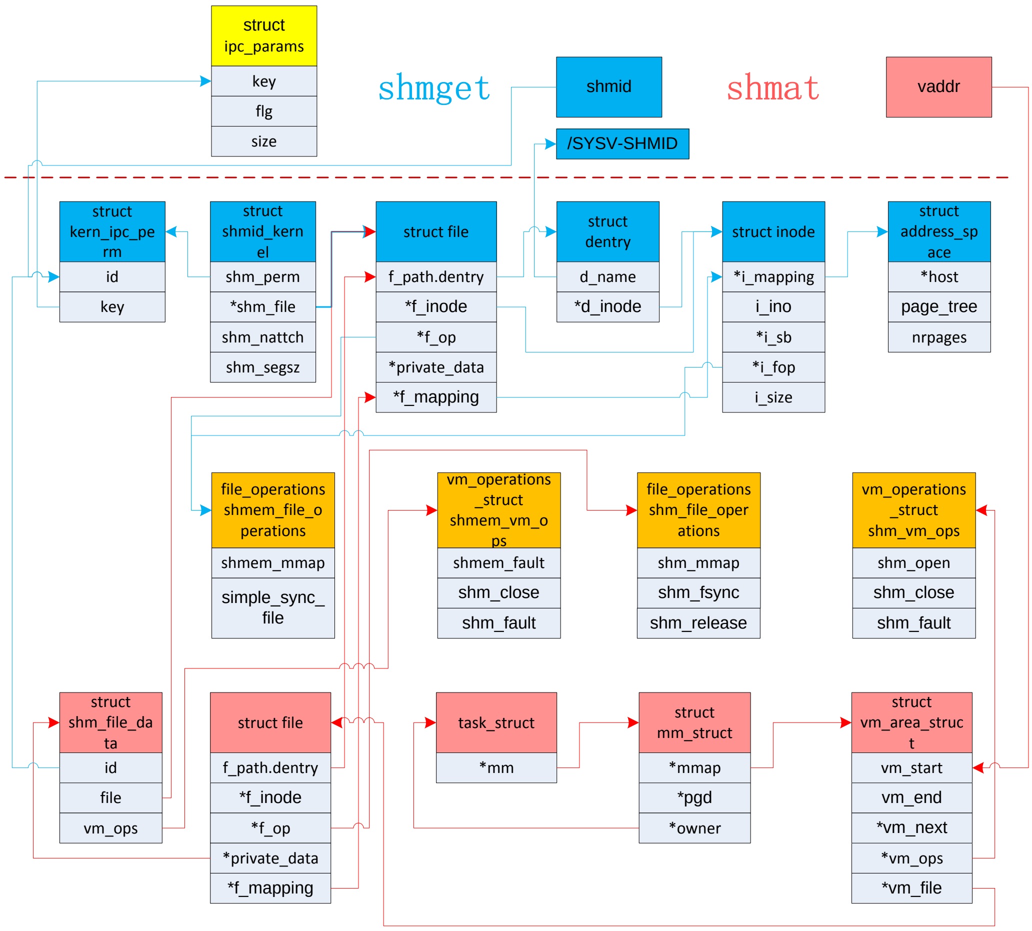
```
atime          dtime          ctime          rss          swap
        1234          65536     666          2052    6924    6924          1    1000    1000    1000
1000 1411221835               0 1411221835          4096               0
drq@ubuntu:/mnt/hgfs/systemtap/share-m$

drq@ubuntu:/mnt/hgfs/systemtap/share-m$ cat /proc/meminfo | grep Shmem
Shmem:                    144 kB

drq@ubuntu:/mnt/hgfs/systemtap/share-m$ mount
/dev/sda1 on / type ext4 (rw,errors=remount-ro)
tmpfs on /run type tmpfs (rw,noexec,nosuid,size=10%,mode=0755)
none on /run/shm type tmpfs (rw,nosuid,nodev)

drq@ubuntu:/mnt/hgfs/systemtap/share-m$ df -h
Filesystem          Size    Used Avail Use% Mounted on
/dev/sda1            39G     17G     20G   47% /
udev                494M    4.0K   494M     1% /dev
tmpfs               201M    812K   200M     1% /run
none                5.0M       0    5.0M     0% /run/lock
none                501M    152K   501M     1% /run/shm
```

5 attach 到 share memory

5.1 主流程

以 shmid attach 到 shm 上，最终在进程空间分配一块内存区域 vm_area_struct 指向 shm 文件的物理页，加入进程的内存描述符 current->mm，此 vm_area_struct 可通过 cat /proc/$pid/maps 查看。

在内核态建立的数据关联信息如下：



红色部分为 shmat 期间在内核新建立的数据信息，并最终返回 vm_start 即用户可直接访问的用户态地址。

用 systemtap 监测到的函数调用栈信息如下：

```
-------------------------------------
shmem_mmap(file=0xc4b42e40 vma=0xddacb000)
 0xc11544e0 : shmem_mmap+0x0/0x30 [kernel]
 0xc12918d2 : shm_mmap+0x22/0x60 [kernel]
 0xc1169380 : mmap_region+0x3d0/0x590 [kernel]
 0xc1169726 : do_mmap_pgoff+0x1e6/0x2d0 [kernel]
 0xc12925af : do_shmat+0x30f/0x3c0 [kernel]
 0xc1292af2 : sys_ipc+0x202/0x280 [kernel]
 0xc161abb4 : syscall_call+0x7/0xb [kernel]
-------------------------------------
-------------------------------------
shm_open(vma=0xddacb000)
 0xc1291850 : shm_open+0x0/0x60 [kernel]
 0xc12918f3 : shm_mmap+0x43/0x60 [kernel]
 0xc1169380 : mmap_region+0x3d0/0x590 [kernel]
```

```
 0xc1169726 : do_mmap_pgoff+0x1e6/0x2d0 [kernel]
 0xc12925af : do_shmat+0x30f/0x3c0 [kernel]
 0xc1292af2 : sys_ipc+0x202/0x280 [kernel]
 0xc161abb4 : syscall_call+0x7/0xb [kernel]
--------------------------------------
```

5.2 do_shmat

建立 share memory 后，以 shmid 进行后续访问操作

```
SYSCALL_DEFINE3(shmat, int, shmid, char __user *, shmaddr, int, shmflg)
{
	err = do_shmat(shmid, shmaddr, shmflg, &ret);
	return (long)ret;
}

/*
 * Fix shmaddr, allocate descriptor, map shm, add attach descriptor to lists.
*/
long do_shmat(int shmid, char __user *shmaddr, int shmflg, ulong *raddr)
{
	struct shmid_kernel *shp;
	unsigned long addr;
	unsigned long size;
	struct file * file;
	struct path path;

	ns = current->nsproxy->ipc_ns;
	shp = shm_lock_check(ns, shmid);	/* 通过 shmid 找到 ipc 数据结构 shmid_kernel */

	path = shp->shm_file->f_path;	/* 获得共享文件的路径 */
	path_get(&path);
	shp->shm_nattch++;
	size = i_size_read(path.dentry->d_inode);	/* 根据 dentry 找到 inode，获取文件大小 */

	sfd = kzalloc(sizeof(*sfd), GFP_KERNEL);	/*每个进程自身维护的信息*/

	file = alloc_file(&path, f_mode,
			is_file_hugepages(shp->shm_file) ?
			  &shm_file_operations_huge :
			  &shm_file_operations);	/* 分配一个新文件描述符指向共享文件，文件访问指针为 shm_file_operations)*/

	file->private_data = sfd;
```

```
	file->f_mapping = shp->shm_file->f_mapping;    /*  指向共享文件的 address_space */
	sfd->id = shp->shm_perm.id;    /*  保存 shmid*/
	sfd->ns = get_ipc_ns(ns);
	sfd->file = shp->shm_file; /*  指向共享文件的 file 描述符  */
	sfd->vm_ops = NULL;

	user_addr = do_mmap (file, addr, size, prot, flags, 0);
	*raddr = user_addr;      /*  返回在进程空间分配的虚拟地址空间指针*/
}
```

5.3 shm_mmap

do_mmap 最终调用 shm_file_operations 的 shm_mmap

```
static int shm_mmap(struct file * file, struct vm_area_struct * vma)
{
	struct shm_file_data *sfd = shm_file_data(file);
	int ret;

	ret = sfd->file->f_op->mmap(sfd->file, vma); /*  最终调用 shmem_file_setup 阶段创建的 shm 里的 file 文件的 f_op 指针 shmem_file_operations 中的 mmap  实现 shmem_mmap*/

	sfd->vm_ops = vma->vm_ops;    /* shmem_vm_ops    */
	vma->vm_ops = &shm_vm_ops;   /* 将 shmem_vm_ops 替换为 shm_vm_ops ，  以便 vm_ops 的其他地方可以进行额外封装处理如 shm_open */
	shm_open(vma);

	return ret;
}
```

5.4 shmem_mmap

```
static int shmem_mmap(struct file *file, struct vm_area_struct *vma)
{
	file_accessed(file);
	vma->vm_ops = &shmem_vm_ops;
	vma->vm_flags |= VM_CAN_NONLINEAR;
	return 0;
}
```

5.5 shm_open

进程 attache 到 shm 后，更新相关访问信息如时间，attach 的个数

```
/* This is called by fork, once for every shm attach. */
static void shm_open(struct vm_area_struct *vma)
{
	struct file *file = vma->vm_file;
	struct shm_file_data *sfd = shm_file_data(file);
	struct shmid_kernel *shp;

	shp = shm_lock(sfd->ns, sfd->id);
	BUG_ON(IS_ERR(shp));
	shp->shm_atim = get_seconds();
	shp->shm_lprid = task_tgid_vnr(current);
	shp->shm_nattch++;
	shm_unlock(shp);
}
```

5.6 用户态信息

进程 attach 到 shm 后，其 nattch 会增加

```
drq@ubuntu:/mnt/hgfs/systemtap$ ipcs -m

------ Shared Memory Segments --------
key        shmid      owner      perms      bytes      nattch     status
0x000004d2 262144     drq        666        2052       1
```

可以从进程 mm 中看到映射的虚拟地址空间

```
drq@ubuntu:/mnt/hgfs/systemtap/share-m$ ps -ef | grep sh-read
drq      11803  5829 99 02:00 pts/7    00:00:17 ./sh-read
```

b76f0000-b76f1000 为 shm 映射后的虚拟地址空间，/SYSV000004d2 为 shm 的虚拟文件

```
drq@ubuntu:/mnt/hgfs/systemtap/share-m$ cat /proc/11803/maps | grep SYS
b76f0000-b76f1000 rw-s 00000000 00:04 262144     /SYSV000004d2 (deleted)
```

6 数据访问

用户空间经过 shmat 后，得到用于访问共享内存的虚拟地址，即可以通过该地址直接访问共享的物理内存。但因为页表尚未建立起来，因此触发 page fault，然后建立页表。

```
-------------------------------------
shmem_fault(vma=0xddacb000 vmf=0xc25cbe7c)
 0xc1155eb0 : shmem_fault+0x0/0x90 [kernel]
 0xc12911a4 : shm_fault+0x14/0x20 [kernel]
 0xc11606ce : __do_fault+0x6e/0x550 [kernel]
```

```
0xc11631cf : handle_pte_fault+0x8f/0xaf0 [kernel]
0xc1164d4d : handle_mm_fault+0x1dd/0x280 [kernel]
0xc161ddea : do_page_fault+0x15a/0x4b0 [kernel]
0xc161b2a3 : error_code+0x67/0x6c [kernel]
```

6.1 shm_fault

在 shm_mmap 的最后将 vm_operations 的操作指针更新为了 shm_vm_ops，其 page fault 处理函数为 shm_fault。其最终仍然调用的是 shmem_vm_ops 的 shmem_fault

```
static int shm_fault(struct vm_area_struct *vma, struct vm_fault *vmf)
{
	struct file *file = vma->vm_file;
	struct shm_file_data *sfd = shm_file_data(file);

	return sfd->vm_ops->fault(vma, vmf);
}
```

6.2 shmem_fault

shmem_fault 根据产生缺页异常的线性地址找到对应的物理页（vma->vm_file->f_path.dentry->d_inode），并将这个物理页加入页表，之后用户就可以像访问本地数据一样直接访问共享内存

```
static int shmem_fault(struct vm_area_struct *vma, struct vm_fault *vmf)
{
	struct inode *inode = vma->vm_file->f_path.dentry->d_inode;
	int error;
	int ret;

	if (((loff_t)vmf->pgoff << PAGE_CACHE_SHIFT) >= i_size_read(inode))
		return VM_FAULT_SIGBUS;

	error = shmem_getpage(inode, vmf->pgoff, &vmf->page, SGP_CACHE, &ret);
	if (error)
		return ((error == -ENOMEM) ? VM_FAULT_OOM : VM_FAULT_SIGBUS);

	return ret | VM_FAULT_LOCKED;
}
```

7 Detach shm

Detach shm 时只会将进城对应的 mm_struct 信息 release，但不会删除 shm 自身。其中 shm_nattch--。

```
shm_close(vma=0xddadf8f0)
  0xc1291910 : shm_close+0x0/0xb0 [kernel]
  0xc1167086 : remove_vma+0x26/0x60 [kernel]
  0xc1168a5c : do_munmap+0x21c/0x2e0 [kernel]
  0xc129272b : sys_shmdt+0x9b/0x140 [kernel]
  0xc1292b1b : sys_ipc+0x22b/0x280 [kernel]
  0xc161abb4 : syscall_call+0x7/0xb [kernel]
-------------------------------------
-------------------------------------
shm_release(ino=0xf69f9e50 file=0xddbdb540)
  0xc1291330 : shm_release+0x0/0x40 [kernel]
  0xc1190ab6 : fput+0xe6/0x210 [kernel]
  0xc1167092 : remove_vma+0x32/0x60 [kernel]
  0xc1168a5c : do_munmap+0x21c/0x2e0 [kernel]
  0xc129272b : sys_shmdt+0x9b/0x140 [kernel]
  0xc1292b1b : sys_ipc+0x22b/0x280 [kernel]
  0xc161abb4 : syscall_call+0x7/0xb [kernel]
-------------------------------------
```

8 删除 share memory

相关命令如下：

```
drq@ubuntu:/mnt/hgfs/systemtap$ ipcs -m

------ Shared Memory Segments --------
key        shmid      owner      perms      bytes      nattch     status
0x00000000 262144     drq        666        2052       1          dest

drq@ubuntu:/mnt/hgfs/systemtap$ ipcrm -m 262144
drq@ubuntu:/mnt/hgfs/systemtap$ ipcs -m

------ Shared Memory Segments --------
key        shmid      owner      perms      bytes      nattch     status
```

程序可以通过 shmctl IO 调用删除 shm。

9 参考文档

共享内存代码示例
http://blog.csdn.net/cschengvdn/article/details/21086711

Linux 环境进程间通信(五)：共享内存(下)
http://www.ibm.com/developerworks/cn/linux/l-ipc/part5/index2.html