Binder系列1—Binder Driver

Nov 1, 2015

> 基于Android 6.0的源码剖析，在讲解Binder原理之前，先从kernel的角度来讲解Binder Driver.

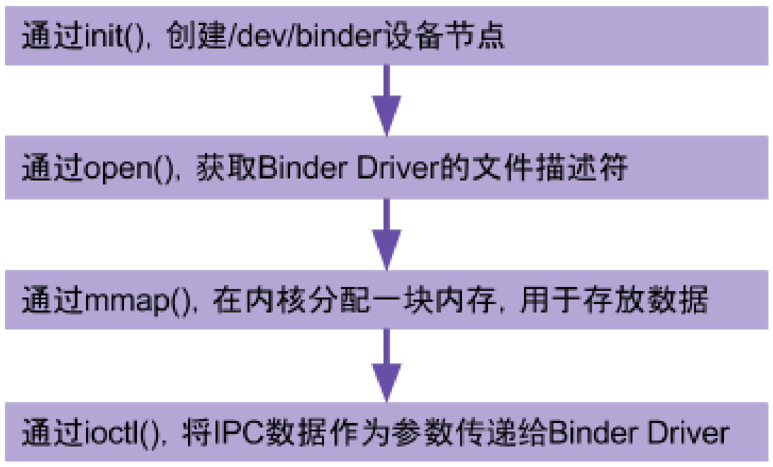
概述

源码路径

Binder Driver的源码路径

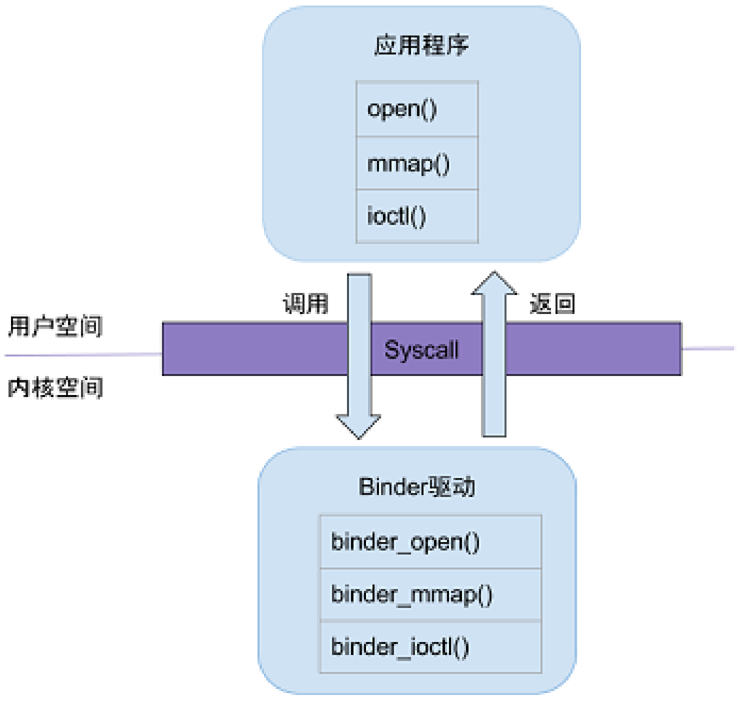
```
/kernel/drivers/android/binder.c
/kernel/include/uapi/linux/android/binder.h
```

Binder驱动是Android专用的，但底层的驱动架构与Linux驱动一样。binder驱动在以misc设备进行注册，作为虚拟设备，没有直接操作硬件，只是对设备内存的处理。主要是驱动设备的初始化(binder_init)，打开 (binder_open)，映射(binder_mmap)，数据操作(binder_ioctl)。


通过init(), 创建/dev/binder设备节点
通过open(), 获取Binder Driver的文件描述符
通过mmap(), 在内核分配一块内存, 用于存放数据
通过ioctl(), 将IPC数据作为参数传递给Binder Driver

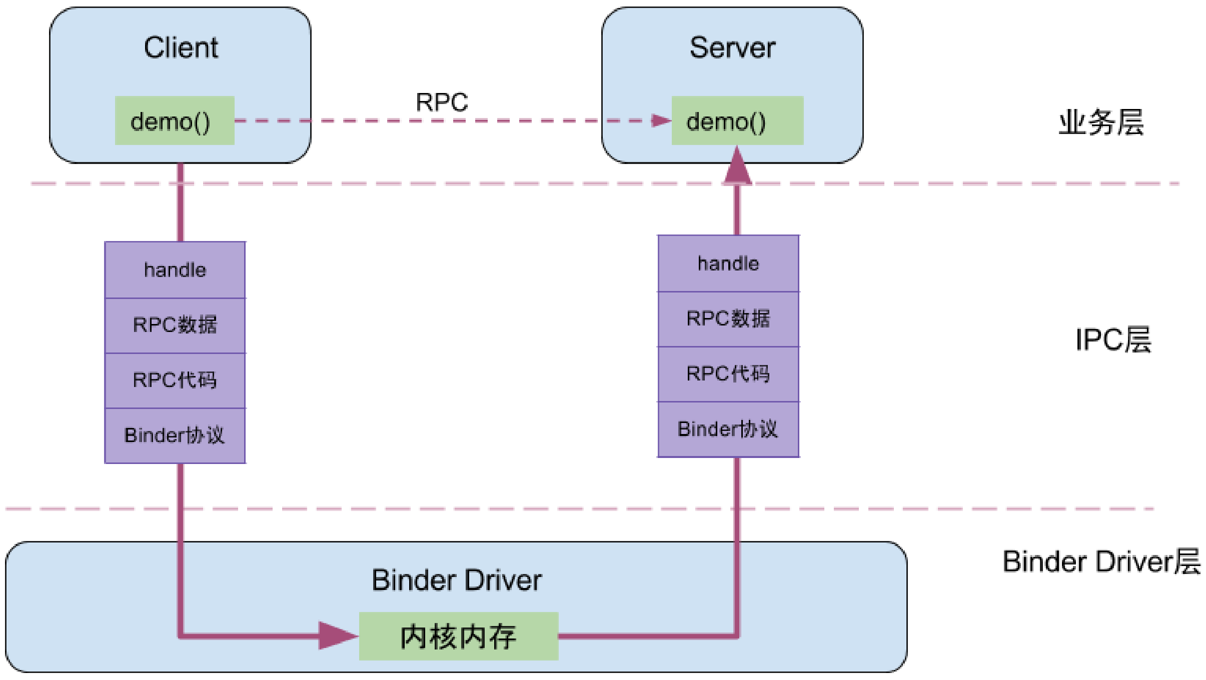
系统调用

用户态的程序调用kernel驱动，需要陷入内核态，进行系统调用(syscall)，比如打开Binder驱动方法的调用链为： open-> __open() -> binder_open()。 open()为用户空间的方法，__open()便是系统调用中相应的处理方法，通过查找，对应调用到内核binder驱动的binder_open()方法，至于其他的从用户态陷入内核态的流程也基本一致。



Binder通信

Client进程通过RPC(Remote Procedure Call Protocol)与Server通信，可以简单地划分为三层，驱动层、IPC层、业务层。 `demo()` 便是Client端和Server共同协商定义好的业务；handle、RPC数据、代码、协议这4项组成了IPC层的数据，通过IPC层进行数据传输；而真正在Client和Server两端建立通信的基础设施便是Binder Driver。



例如，当名为 `BatteryStatsService` 的Client向ServiceManager注册服务的过程中，IPC层的数据组成为：Hanlde=0，RPC代码为 `ADD_SERVICE` ，RPC数据为 `BatteryStatsService` ，Binder协议为 `BC_TRANSACTION` 。

一、 binder_init

主要工作是为了注册misc设备

```
static int __init binder_init(void)
{
        int ret;
        binder_deferred_workqueue = create_singlethread_workqueue("bind
er"); //创建名为binder的workqueue
        if (!binder_deferred_workqueue)
                return -ENOMEM;

        binder_debugfs_dir_entry_root = debugfs_create_dir("binder", NU
LL); //创建debugfs目录
        if (binder_debugfs_dir_entry_root)
                binder_debugfs_dir_entry_proc = debugfs_create_dir("pro
c",
                                                binder_debugfs_dir_ent
ry_root);
        ret = misc_register(&binder_miscdev);    // 注册misc设备
        if (binder_debugfs_dir_entry_root) {
                ... //在debugfs文件系统中创建一系列的文件
        }
        return ret;
}
```

`debugfs_create_dir` 是指在debugfs文件系统中创建一个目录，返回值是指向dentry的指针。当kernel中禁用debugfs的话，返回值是-%ENODEV。默认是禁用的。如果需要打开，在目录 `/kernel/arch/arm64/configs/` 下找到目标defconfig文件中添加一行 `CONFIG_DEBUG_FS=y` ，再重新编译版本，即可打开debug_fs。

misc_register

注册misc设备， `miscdevice` 结构体，便是前面注册misc设备时传递进去的参数

```
static struct miscdevice binder_miscdev = {
        .minor = MISC_DYNAMIC_MINOR, //次设备号 动态分配
        .name = "binder",     //设备名
        .fops = &binder_fops  //设备的文件操作结构，这是file_operations结
构
};
```

`file_operations` 结构体,指定相应文件操作的方法

```
static const struct file_operations binder_fops = {
        .owner = THIS_MODULE,
        .poll = binder_poll,
        .unlocked_ioctl = binder_ioctl,
        .compat_ioctl = binder_ioctl,
        .mmap = binder_mmap,
        .open = binder_open,
        .flush = binder_flush,
        .release = binder_release,
};
```

二、 binder_open

打开binder驱动设备

```
static int binder_open(struct inode *nodp, struct file *filp)
{
        struct binder_proc *proc; // binder进程 【见附录】

        proc = kzalloc(sizeof(*proc), GFP_KERNEL); // 为binder_proc结构
体在分配kernel内存空间
        if (proc == NULL)
                return -ENOMEM;
        get_task_struct(current);
        proc->tsk = current;   //将当前线程的task保存到binder进程的tsk
        INIT_LIST_HEAD(&proc->todo);
        init_waitqueue_head(&proc->wait); //初始化等待队列
        proc->default_priority = task_nice(current);  //将当前进程的nice
值转换为进程优先级

        binder_lock(__func__);   //同步锁，因为binder支持多线程访问
        binder_stats_created(BINDER_STAT_PROC); //BINDER_PROC对象创建数
加1
        hlist_add_head(&proc->proc_node, &binder_procs);
        proc->pid = current->group_leader->pid;
        INIT_LIST_HEAD(&proc->delivered_death);
        filp->private_data = proc;       //file文件指针的private_data变量
指向binder_proc数据
        binder_unlock(__func__); //释放同步锁

        return 0;
}
```

binder_open: 过程中的filp->private_data = proc 该语句很重要，将通过filp可获取相应的binder_proc保存到filp指针的private_data成员变量，那么之后通过filp可获取相应的binder_proc，而binder_proc里管理IPC所需的各种信息，拥有其他结构体的跟结构体。

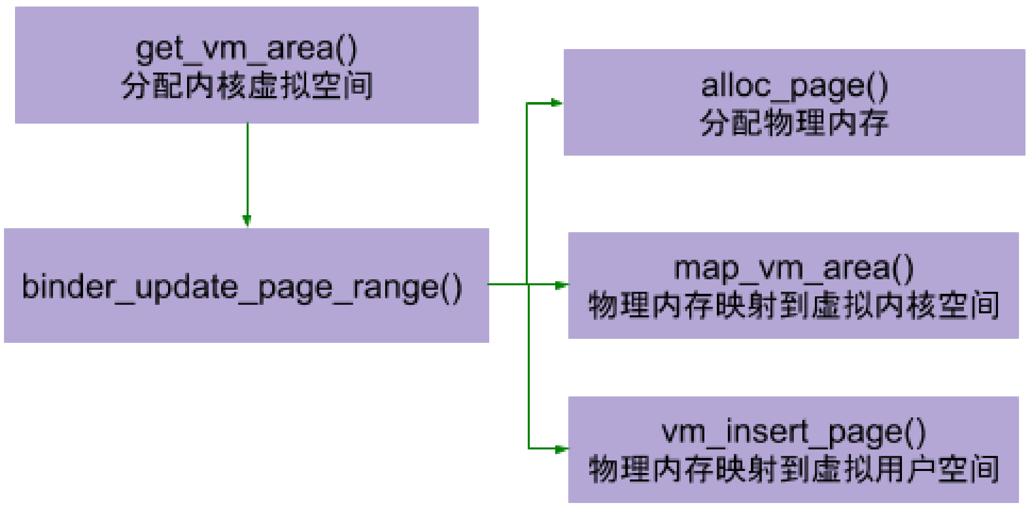
三、 binder_mmap

> binder_mmap(文件描述符，用户虚拟内存空间)

主要功能：首先在内核虚拟地址空间，申请一块与用户虚拟内存相同大小的内存；然后再申请1个page大小的物理内存，再将同一块物理内存分别映射到内核虚拟地址空间和用户虚拟内存空间，从而实现了用户空间的Buffer和内核空间的Buffer同步操作的功能。

```
static int binder_mmap(struct file *filp, struct vm_area_struct *vma)
{
        int ret;
        struct vm_struct *area; //虚拟内核空间
        struct binder_proc *proc = filp->private_data;
        const char *failure_string;
        struct binder_buffer *buffer;  //【见附录】

        if (proc->tsk != current)
                return -EINVAL;

        if ((vma->vm_end - vma->vm_start) > SZ_4M)
                vma->vm_end = vma->vm_start + SZ_4M;  //保证映射内存大小
不超过4M

        mutex_lock(&binder_mmap_lock);  //同步锁
        //分配一个连续的内核虚拟空间，与进程虚拟空间大小一致
        area = get_vm_area(vma->vm_end - vma->vm_start, VM_IOREMAP);
        if (area == NULL) {
                ret = -ENOMEM;
                failure_string = "get_vm_area";
                goto err_get_vm_area_failed;
        }
        proc->buffer = area->addr;
        proc->user_buffer_offset = vma->vm_start - (uintptr_t)proc->buf
fer;
        mutex_unlock(&binder_mmap_lock); //释放锁

        ...
        proc->pages = kzalloc(sizeof(proc->pages[0]) * ((vma->vm_end -
vma->vm_start) / PAGE_SIZE), GFP_KERNEL);//分配一个虚拟用户空间的页数大小的
内存给pages指针
        if (proc->pages == NULL) {
                ret = -ENOMEM;
                failure_string = "alloc page array";
                goto err_alloc_pages_failed;
        }
        proc->buffer_size = vma->vm_end - vma->vm_start;

        vma->vm_ops = &binder_vm_ops;
        vma->vm_private_data = proc;

        //分配物理页面，同时映射到内核空间和进程空间 【见】
        if (binder_update_page_range(proc, 1, proc->buffer, proc->buffe
r + PAGE_SIZE, vma)) {
                ret = -ENOMEM;
                failure_string = "alloc small buf";
                goto err_alloc_small_buf_failed;
        }
        buffer = proc->buffer;
        INIT_LIST_HEAD(&proc->buffers);
        list_add(&buffer->entry, &proc->buffers);
        buffer->free = 1;
        binder_insert_free_buffer(proc, buffer);
        proc->free_async_space = proc->buffer_size / 2;
        barrier();
        proc->files = get_files_struct(current);
        proc->vma = vma;
        proc->vma_vm_mm = vma->vm_mm;
        return 0;

        ...// 错误flags跳转处，free释放内存之类的操作
        return ret;
}
```

binder_mmap的主要工作可用下面的图来表达：



`binder_update_page_range` 主要完成工作：分配物理空间，将物理空间映射到内核空间，将物理空间映射到进程空间。

代码如下：

```
static int binder_update_page_range(struct binder_proc *proc, int alloc
ate,
                                    void *start, void *end,  struct v
m_area_struct *vma)
{
        ...
        for (page_addr = start; page_addr < end; page_addr += PAGE_SIZ
E) {
                int ret;
                struct page **page_array_ptr;
                page = &proc->pages[(page_addr - proc->buffer) / PAGE_S
IZE];
                BUG_ON(*page);
                *page = alloc_page(GFP_KERNEL | __GFP_HIGHMEM | __GFP_Z
ERO);  //分配物理内存
                if (*page == NULL) {
                        goto err_alloc_page_failed;
                }
                tmp_area.addr = page_addr;
                tmp_area.size = PAGE_SIZE + PAGE_SIZE;
                page_array_ptr = page;
                ret = map_vm_area(&tmp_area, PAGE_KERNEL, &page_array_p
tr); //物理空间映射到虚拟内核空间
                if (ret) {
                        goto err_map_kernel_failed;
                }
                user_page_addr = (uintptr_t)page_addr + proc->user_buff
er_offset;
                ret = vm_insert_page(vma, user_page_addr, page[0]);
//物理空间映射到虚拟进程空间
                if (ret) {
                        goto err_vm_insert_page_failed;
                }
        }
        ...
}
```

四、 binder_ioctl

binder_ioctl()函数负责在两个进程间收发IPC数据和IPC reply数据。

> ioctl(文件描述符，ioctl命令，数据类型)

(1) 文件描述符，是通过open()方法打开Binder Driver后返回值；

(2) ioctl命令和数据类型是一体的，不同的命令对应不同的数据类型

ioctl命令	数据类型	操作
BINDER_WRITE_READ	struct binder_write_read	收发Binder IPC数据
BINDER_SET_MAX_THREADS	__u32	设置Binder线程最大个数
BINDER_SET_CONTEXT_MGR	__s32	设置Service Manager节点
BINDER_THREAD_EXIT	__s32	释放Binder线程
BINDER_VERSION	struct binder_version	获取Binder版本信息
BINDER_SET_IDLE_TIMEOUT	__s64	没有使用
BINDER_SET_IDLE_PRIORITY	__s32	没有使用

上述命令中 `BINDER_WRITE_READ` 命令使用率最为频繁，也是ioctl的核心命令。

源码

```
static long binder_ioctl(struct file *filp, unsigned int cmd, unsigned long arg)
{
        int ret;
        struct binder_proc *proc = filp->private_data;
        struct binder_thread *thread;  // binder线程
        unsigned int size = _IOC_SIZE(cmd);
        void __user *ubuf = (void __user *)arg;
        //进入休眠状态，直到中断唤醒
        ret = wait_event_interruptible(binder_user_error_wait, binder_stop_on_user_error < 2);
        if (ret)
                goto err_unlocked;

        binder_lock(__func__);
        thread = binder_get_thread(proc); //【获取binder_thread 见小节4.1】
        if (thread == NULL) {
                ret = -ENOMEM;
                goto err;
        }

        switch (cmd) {
        case BINDER_WRITE_READ:  //进行binder的读写操作
                ret = binder_ioctl_write_read(filp, cmd, arg, thread); //读写操作【见小节4.2】
                if (ret)
                        goto err;
                break;
        case BINDER_SET_MAX_THREADS: //设置binder最大支持的线程数
                if (copy_from_user(&proc->max_threads, ubuf, sizeof(proc->max_threads))) {
                        ret = -EINVAL;
                        goto err;
                }
                break;
        case BINDER_SET_CONTEXT_MGR: //成为binder的上下文管理者，也就是ServiceManager成为守护进程
                ret = binder_ioctl_set_ctx_mgr(filp);
                if (ret)
                        goto err;
                break;
        case BINDER_THREAD_EXIT:   //当binder线程退出，释放binder线程
                binder_free_thread(proc, thread);
                thread = NULL;
                break;
        case BINDER_VERSION: {  //获取binder的版本号
                struct binder_version __user *ver = ubuf;
                if (size != sizeof(struct binder_version)) {
                        ret = -EINVAL;
                        goto err;
                }
                if (put_user(BINDER_CURRENT_PROTOCOL_VERSION,
                             &ver->protocol_version)) {
                        ret = -EINVAL;
                        goto err;
                }
                break;
        }
        default:
                ret = -EINVAL;
                goto err;
        }
        ret = 0;
err:
        if (thread)
                thread->looper &= ~BINDER_LOOPER_STATE_NEED_RETURN;
        binder_unlock(__func__);
        wait_event_interruptible(binder_user_error_wait, binder_stop_o
n_user_error < 2);

err_unlocked:
        trace_binder_ioctl_done(ret);
        return ret;
}
```

4.1 binder_get_thread()

从binder_proc中查找binder_thread,如果存在则直接返回，如果不存在则新建一个，并添加到当前的proc

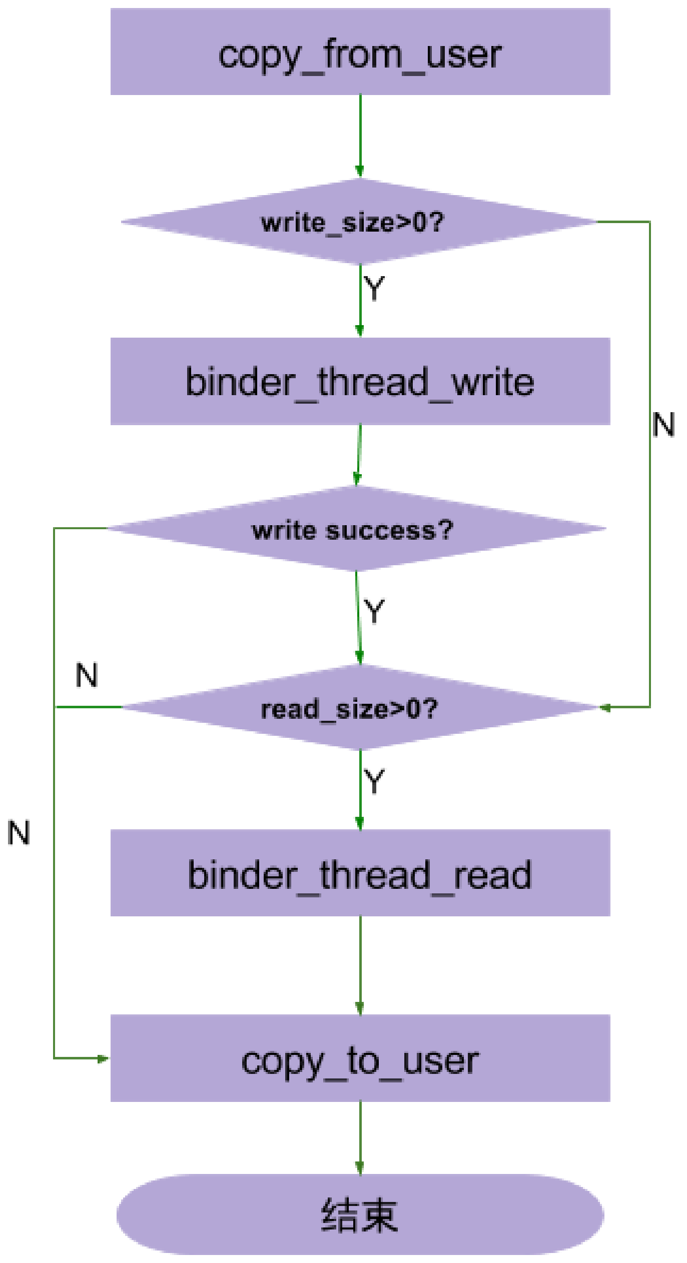
```
static struct binder_thread *binder_get_thread(struct binder_proc *pro
c)
{
        struct binder_thread *thread = NULL;
        struct rb_node *parent = NULL;
        struct rb_node **p = &proc->threads.rb_node;
        while (*p) {  //根据当前进程的pid，从binder_proc中查找相应的binde
r_thread
                parent = *p;
                thread = rb_entry(parent, struct binder_thread, rb_nod
e);
                if (current->pid < thread->pid)
                        p = &(*p)->rb_left;
                else if (current->pid > thread->pid)
                        p = &(*p)->rb_right;
                else
                        break;
        }
        if (*p == NULL) {
                thread = kzalloc(sizeof(*thread), GFP_KERNEL); //新建bi
nder_thread结构体
                if (thread == NULL)
                        return NULL;
                binder_stats_created(BINDER_STAT_THREAD);
                thread->proc = proc;
                thread->pid = current->pid;  //保存当前进程(线程)的pid
                init_waitqueue_head(&thread->wait);
                INIT_LIST_HEAD(&thread->todo);
                rb_link_node(&thread->rb_node, parent, p);
                rb_insert_color(&thread->rb_node, &proc->threads);
                thread->looper |= BINDER_LOOPER_STATE_NEED_RETURN;
                thread->return_error = BR_OK;
                thread->return_error2 = BR_OK;
        }
        return thread;
}
```

4.2 binder_ioctl_write_read()

对于ioctl()方法中，传递进来的命令是cmd = `BINDER_WRITE_READ` 时执行该方法，arg是一个 `binder_write_read` 结构体

```
static int binder_ioctl_write_read(struct file *filp,
                                unsigned int cmd, unsigned long arg,
                                struct binder_thread *thread)
{
        int ret = 0;
        struct binder_proc *proc = filp->private_data;
        unsigned int size = _IOC_SIZE(cmd);
        void __user *ubuf = (void __user *)arg;
        struct binder_write_read bwr;

        if (size != sizeof(struct binder_write_read)) {
                ret = -EINVAL;
                goto out;
        }
        if (copy_from_user(&bwr, ubuf, sizeof(bwr))) { //把用户空间数据ub
uf拷贝到bwr
                ret = -EFAULT;
                goto out;
        }

        if (bwr.write_size > 0) {
                //当写缓存中有数据，则执行binder写操作【见4.3】
                ret = binder_thread_write(proc, thread,
                                          bwr.write_buffer,
                                          bwr.write_size,
                                          &bwr.write_consumed);
                trace_binder_write_done(ret);
                if (ret < 0) { //当写失败，再将bwr数据写回用户空间，并返回
                        bwr.read_consumed = 0;
                        if (copy_to_user(ubuf, &bwr, sizeof(bwr)))
                                ret = -EFAULT;
                        goto out;
                }
        }
        if (bwr.read_size > 0) {
                //当读缓存中有数据，则执行binder读操作，【见4.4】
                ret = binder_thread_read(proc, thread, bwr.read_buffer,
                                         bwr.read_size,
                                         &bwr.read_consumed,
                                         filp->f_flags & O_NONBLOCK);
                trace_binder_read_done(ret);
                if (!list_empty(&proc->todo))
                        wake_up_interruptible(&proc->wait); //进入休
眠，等待中断唤醒
                if (ret < 0) { //当读失败，再将bwr数据写回用户空间，并返回
                        if (copy_to_user(ubuf, &bwr, sizeof(bwr)))
                                ret = -EFAULT;
                        goto out;
                }
        }

        if (copy_to_user(ubuf, &bwr, sizeof(bwr))) { //将内核数据bwr拷贝到用户空间ubuf
                ret = -EFAULT;
                goto out;
        }
out:
        return ret;
}
```

对于 `binder_ioctl_write_read` 的流程图，如下：



流程：

- 首先把用户空间数据拷贝到内核空间bwr；
- 当bwr写缓存中有数据，则执行binder写操作；当写失败，再将bwr数据写回

用户空间，并退出；

- 当bwr读缓存中有数据，则执行binder读操作；当读失败，再将bwr数据写回用户空间，并退出；
- 最后把内核数据bwr拷贝到用户空间。

4.3 binder_thread_write()

binder线程的写操作，下面只列举部分cmd命令相应的操作方法

```
int binder_thread_write(struct binder_proc *proc,
                        struct binder_thread *thread,
                        binder_uintptr_t binder_buffer, size_t size,
                        binder_size_t *consumed)
{
        uint32_t cmd;
        void __user *buffer = (void __user *)(uintptr_t)binder_buffer;
        void __user *ptr = buffer + *consumed;
        void __user *end = buffer + size;

        while (ptr < end && thread->return_error == BR_OK) {
                if (get_user(cmd, (uint32_t __user *)ptr))
                        return -EFAULT;
                ptr += sizeof(uint32_t);
                trace_binder_command(cmd);
                if (_IOC_NR(cmd) < ARRAY_SIZE(binder_stats.bc)) {
                        binder_stats.bc[_IOC_NR(cmd)]++;
                        proc->stats.bc[_IOC_NR(cmd)]++;
                        thread->stats.bc[_IOC_NR(cmd)]++;
                }
                switch (cmd) {
                        case BC_ACQUIRE:
                        case BC_RELEASE:
                        case BC_DECREFS: {
                                uint32_t target;
                                struct binder_ref *ref;
                                const char *debug_string;

                                if (get_user(target, (uint32_t __user
*)ptr))
                                        return -EFAULT;
                                ptr += sizeof(uint32_t);
                                if (target == 0 && binder_context_mgr_n
ode &&
                                    (cmd == BC_INCREFS || cmd == BC_ACQ
UIRE)) {
                                        ref = binder_get_ref_for_node(p
roc,
                                                               binder_context_m
gr_node);
                                } else
                                        ref = binder_get_ref(proc, targ
et);
                                if (ref == NULL) {
                                        break;
                                }
                                switch (cmd) {
                                case BC_INCREFS:
                                        debug_string = "IncRefs";
                                        binder_inc_ref(ref, 0, NULL);
                                        break;
                                case BC_ACQUIRE:
                                        debug_string = "Acquire";
                                        binder_inc_ref(ref, 1, NULL);
                                        break;
                                case BC_RELEASE:
                                        debug_string = "Release";
                                        binder_dec_ref(&ref, 1);
                                        break;
                                case BC_DECREFS:
                                default:
                                        debug_string = "DecRefs";
                                        binder_dec_ref(&ref, 0);
                                        break;
                                }
                          break;
                        }


                        case BC_TRANSACTION:
                        case BC_REPLY: {
                                struct binder_transaction_data tr;

                                if (copy_from_user(&tr, ptr, sizeof(t
r)))
                                        return -EFAULT;
                                ptr += sizeof(tr);
                                binder_transaction(proc, thread, &tr, c
md == BC_REPLY);
                                break;
                        }

                        case BC_REGISTER_LOOPER: //注册Looper
                                if (thread->looper & BINDER_LOOPER_STAT
E_ENTERED) {
                                        thread->looper |= BINDER_LOOPE
R_STATE_INVALID;
                                        binder_user_error("%d:%d ERROR:
BC_REGISTER_LOOPER called after BC_ENTER_LOOPER\n",
                                                proc->pid, thread->pi
d);
                                } else if (proc->requested_threads ==
0) {
                                        thread->looper |= BINDER_LOOPE
R_STATE_INVALID;
                                        binder_user_error("%d:%d ERROR:
BC_REGISTER_LOOPER called without request\n",
                                                proc->pid, thread->pi
d);
                                } else {
                                        proc->requested_threads--;
                                        proc->requested_threads_starte
d++;
                                }
                                thread->looper |= BINDER_LOOPER_STATE_R
EGISTERED;

                                break;

                        case BC_ENTER_LOOPER:  //进入Looper
                                if (thread->looper & BINDER_LOOPER_STAT
E_REGISTERED) {

                                        thread->looper |= BINDER_LOOPE
R_STATE_INVALID;

                                }
                                thread->looper |= BINDER_LOOPER_STATE_E
NTERED;

                                break;
                        case BC_EXIT_LOOPER:  //退出Looper
                                thread->looper |= BINDER_LOOPER_STATE_E
XITED;

                                break;

                        case BC_REQUEST_DEATH_NOTIFICATION: //处理死亡通
知请求

                        case BC_CLEAR_DEATH_NOTIFICATION: {
                                uint32_t target;
                                binder_uintptr_t cookie;
                                struct binder_ref *ref;
                                struct binder_ref_death *death;

                                if (get_user(target, (uint32_t __user
*)ptr))

                                        return -EFAULT;
                                ptr += sizeof(uint32_t);
                                if (get_user(cookie, (binder_uintptr_t
__user *)ptr))

                                        return -EFAULT;
                                ptr += sizeof(binder_uintptr_t);
                                ref = binder_get_ref(proc, target);
                                if (ref == NULL) {
                                        break;
                                }
                                if (cmd == BC_REQUEST_DEATH_NOTIFICATIO
N) {

                                        if (ref->death) {
                                                break;
                                        }
                                        death = kzalloc(sizeof(*death),
GFP_KERNEL);
                                        if (death == NULL) {
                                                thread->return_error =
BR_ERROR;
                                                break;
                                        }
                                        binder_stats_created(BINDER_STA
T_DEATH);
                                        INIT_LIST_HEAD(&death->work.ent
ry);
                                        death->cookie = cookie;
                                        ref->death = death;
                                        if (ref->node->proc == NULL) {
                                                ref->death->work.type =
BINDER_WORK_DEAD_BINDER;
                                                if (thread->looper & (B
INDER_LOOPER_STATE_REGISTERED | BINDER_LOOPER_STATE_ENTERED)) {
                                                        list_add_tai
l(&ref->death->work.entry, &thread->todo);
                                                } else {
                                                        list_add_tai
l(&ref->death->work.entry, &proc->todo);
                                                        wake_up_interru
ptible(&proc->wait);
                                                }
                                        }
                                } else {
                                        if (ref->death == NULL) {
                                                break;
                                        }
                                        death = ref->death;
                                        if (death->cookie != cookie) {
                                                break;
                                        }
                                        ref->death = NULL;
                                        if (list_empty(&death->work.ent
ry)) {
                                                death->work.type = BIND
ER_WORK_CLEAR_DEATH_NOTIFICATION;
                                                if (thread->looper & (B
INDER_LOOPER_STATE_REGISTERED | BINDER_LOOPER_STATE_ENTERED)) {
                                                        list_add_tai
l(&death->work.entry, &thread->todo);
                                                } else {
                                                        list_add_tai
l(&death->work.entry, &proc->todo);
                                                        wake_up_interru
ptible(&proc->wait);
                                                }
                                        } else {
                                                BUG_ON(death->work.type
!= BINDER_WORK_DEAD_BINDER);
                                                death->work.type = BIND
ER_WORK_DEAD_BINDER_AND_CLEAR;
                                        }
                                }
                        } break;
                        ...
                }

                *consumed = ptr - buffer;
        }
        return 0;
}
```

4.4 binder_thread_read()

binder线程的读操作，下面只列举部分cmd命令相应的操作方法

```
static int binder_thread_read(struct binder_proc *proc,
                              struct binder_thread *thread,
                              binder_uintptr_t binder_buffer, size_t si
ze,
                              binder_size_t *consumed, int non_block)
{
        void __user *buffer = (void __user *)(uintptr_t)binder_buffer;
        void __user *ptr = buffer + *consumed;
        void __user *end = buffer + size;

        int ret = 0;
        int wait_for_proc_work;

        if (*consumed == 0) { //写入BR_NOOP到ptr指向的缓冲区
                if (put_user(BR_NOOP, (uint32_t __user *)ptr))
                        return -EFAULT;
                ptr += sizeof(uint32_t);
        }

retry:
        wait_for_proc_work = thread->transaction_stack == NULL &&
                                list_empty(&thread->todo);

        if (thread->return_error != BR_OK && ptr < end) { //将error传回p
tr，跳转done
                if (thread->return_error2 != BR_OK) {
                        if (put_user(thread->return_error2, (uint32_t
__user *)ptr))
                                return -EFAULT;
                        ptr += sizeof(uint32_t);
                        binder_stat_br(proc, thread, thread->return_err
or2);
                        if (ptr == end)
                                goto done;
                        thread->return_error2 = BR_OK;
                }
                if (put_user(thread->return_error, (uint32_t __user *)p
tr))
                        return -EFAULT;
                ptr += sizeof(uint32_t);
                binder_stat_br(proc, thread, thread->return_error);
                thread->return_error = BR_OK;
                goto done;
        }


        thread->looper |= BINDER_LOOPER_STATE_WAITING;
        if (wait_for_proc_work) //当前线程没有实物需要处理
```

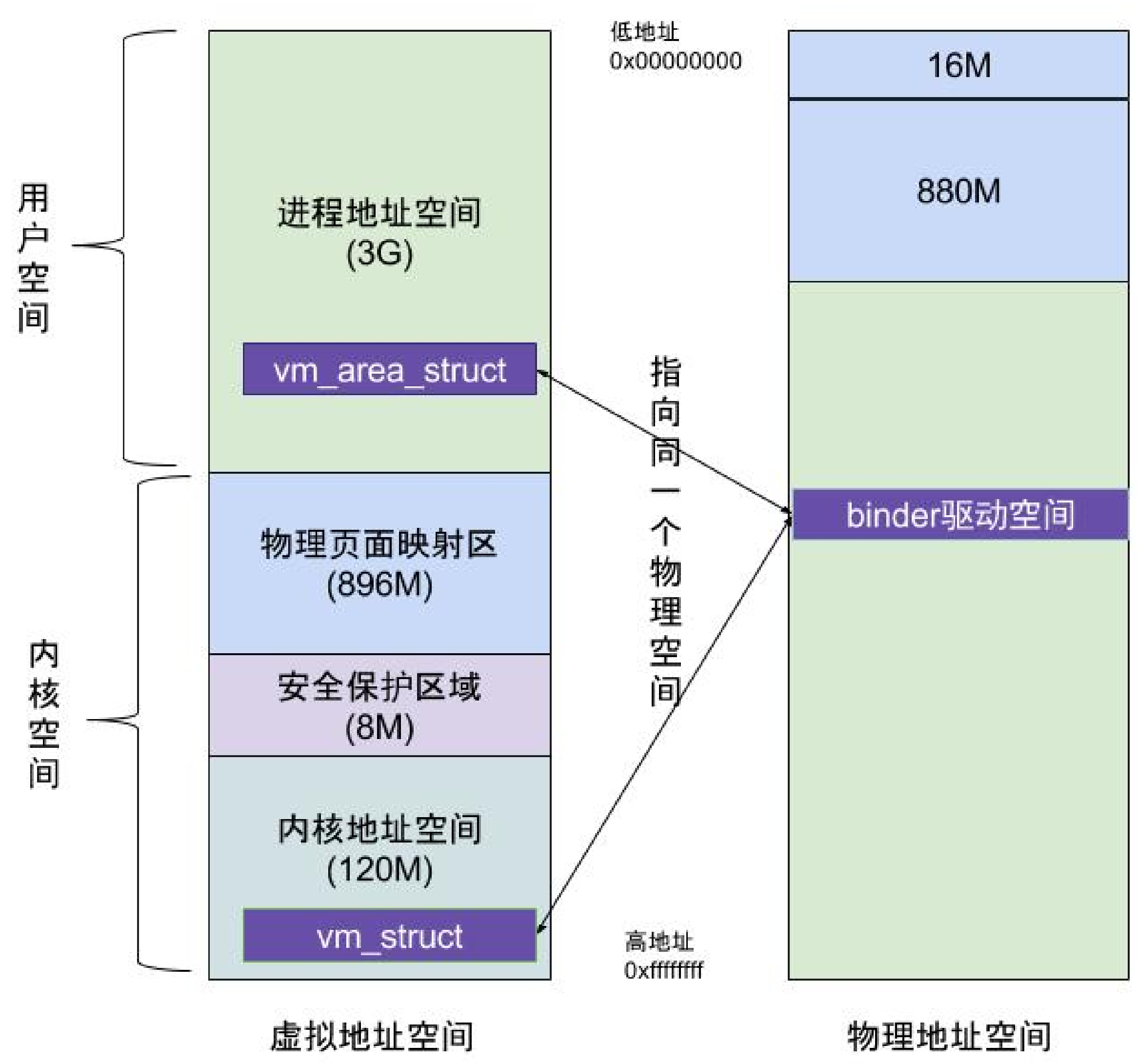
```
                proc->ready_threads++;

        binder_unlock(__func__);

        if (wait_for_proc_work) { ///当前线程没有实物需要处理
                if (!(thread->looper & (BINDER_LOOPER_STATE_REGISTERED
|
                                        BINDER_LOOPER_STATE_ENTERED)))
{
                        wait_event_interruptible(binder_user_error_wai
t,
                                                 binder_stop_on_user_er
ror < 2);
                }
                binder_set_nice(proc->default_priority);  //设置优先级
                if (non_block) { //没阻塞模式，直接返回
                        if (!binder_has_proc_work(proc, thread))
                                ret = -EAGAIN;
                } else //阻塞模式，让当前线程进入休眠状态，等待请求来唤醒
                        ret = wait_event_freezable_exclusive(proc->wai
t, binder_has_proc_work(proc, thread));
        } else {
                if (non_block) {
                        if (!binder_has_thread_work(thread))
                                ret = -EAGAIN;
                } else
                        ret = wait_event_freezable(thread->wait, binde
r_has_thread_work(thread));
        }

        binder_lock(__func__);

        if (wait_for_proc_work)
                proc->ready_threads--;
        thread->looper &= ~BINDER_LOOPER_STATE_WAITING;

        if (ret)
                return ret;

        while (1) {
                uint32_t cmd;
                struct binder_transaction_data tr;
                struct binder_work *w;
                struct binder_transaction *t = NULL;

                if (!list_empty(&thread->todo)) {
                        w = list_first_entry(&thread->todo, struct bind
er_work,
                                             entry);
                } else if (!list_empty(&proc->todo) && wait_for_proc_wo
rk) {
                        w = list_first_entry(&proc->todo, struct binde
r_work,
                                             entry);
                } else {
                        /* no data added */
                        if (ptr - buffer == 4 &&
                            !(thread->looper & BINDER_LOOPER_STATE_NEE
D_RETURN))
                                goto retry;
                        break;
                }

                if (end - ptr < sizeof(tr) + 4)
                        break;

                switch (w->type) {
                case BINDER_WORK_TRANSACTION: {
                        t = container_of(w, struct binder_transaction,
work);
                } break;
                case BINDER_WORK_TRANSACTION_COMPLETE: {
                        cmd = BR_TRANSACTION_COMPLETE;
                        if (put_user(cmd, (uint32_t __user *)ptr))
                                return -EFAULT;
                        ptr += sizeof(uint32_t);

                        binder_stat_br(proc, thread, cmd);
                        binder_debug(BINDER_DEBUG_TRANSACTION_COMPLETE,
                                     "%d:%d BR_TRANSACTION_COMPLETE\n",
                                     proc->pid, thread->pid);

                        list_del(&w->entry);
                        kfree(w);
                        binder_stats_deleted(BINDER_STAT_TRANSACTION_CO
MPLETE);
                } break;
                case BINDER_WORK_NODE: {
                        struct binder_node *node = container_of(w, stru
ct binder_node, work);
                        uint32_t cmd = BR_NOOP;
                        const char *cmd_name;
                        int strong = node->internal_strong_refs || nod
e->local_strong_refs;
                        int weak = !hlist_empty(&node->refs) || node->l
ocal_weak_refs || strong;

                        if (weak && !node->has_weak_ref) {
                                cmd = BR_INCREFS;
                                cmd_name = "BR_INCREFS";
                                node->has_weak_ref = 1;
                                node->pending_weak_ref = 1;
                                node->local_weak_refs++;
                        } else if (strong && !node->has_strong_ref) {
                                cmd = BR_ACQUIRE;
                                cmd_name = "BR_ACQUIRE";
                                node->has_strong_ref = 1;
                                node->pending_strong_ref = 1;
                                node->local_strong_refs++;
                        } else if (!strong && node->has_strong_ref) {
                                cmd = BR_RELEASE;
                                cmd_name = "BR_RELEASE";
                                node->has_strong_ref = 0;
                        } else if (!weak && node->has_weak_ref) {
                                cmd = BR_DECREFS;
                                cmd_name = "BR_DECREFS";
                                node->has_weak_ref = 0;
                        }
                        if (cmd != BR_NOOP) {
                                if (put_user(cmd, (uint32_t __user *)pt
r))
                                        return -EFAULT;
                                ptr += sizeof(uint32_t);
                                if (put_user(node->ptr,
                                             (binder_uintptr_t __user
*)ptr))
                                        return -EFAULT;
                                ptr += sizeof(binder_uintptr_t);
                                if (put_user(node->cookie,
                                             (binder_uintptr_t __user
*)ptr))
                                        return -EFAULT;
                                ptr += sizeof(binder_uintptr_t);

                                binder_stat_br(proc, thread, cmd);
                                binder_debug(BINDER_DEBUG_USER_REFS,
                                             "%d:%d %s %d u%016llx c%01
6llx\n",
                                             proc->pid, thread->pid, cm
d_name,
                                             node->debug_id,
                                             (u64)node->ptr, (u64)node-
>cookie);
                        } else {
                                list_del_init(&w->entry);
                                if (!weak && !strong) {
                                        binder_debug(BINDER_DEBUG_INTER
NAL_REFS,
                                                        "%d:%d node %d u%0
16llx c%016llx deleted\n",
                                                        proc->pid, thread-
>pid,
                                                        node->debug_id,
                                                        (u64)node->ptr,
                                                        (u64)node->cooki
e);
                                        rb_erase(&node->rb_node, &proc-
>nodes);
                                        kfree(node);
                                        binder_stats_deleted(BINDER_STA
T_NODE);
                                } else {
                                        binder_debug(BINDER_DEBUG_INTER
NAL_REFS,
                                                        "%d:%d node %d u%0
16llx c%016llx state unchanged\n",
                                                        proc->pid, thread-
>pid,
                                                        node->debug_id,
                                                        (u64)node->ptr,
                                                        (u64)node->cooki
e);
                                }
                        }
                } break;
                case BINDER_WORK_DEAD_BINDER:
                case BINDER_WORK_DEAD_BINDER_AND_CLEAR:
                case BINDER_WORK_CLEAR_DEATH_NOTIFICATION: {
                        struct binder_ref_death *death;
                        uint32_t cmd;

                        death = container_of(w, struct binder_ref_deat
h, work);
                        if (w->type == BINDER_WORK_CLEAR_DEATH_NOTIFICA
TION)
                                cmd = BR_CLEAR_DEATH_NOTIFICATION_DONE;
                        else
                                cmd = BR_DEAD_BINDER;
                        if (put_user(cmd, (uint32_t __user *)ptr))
                                return -EFAULT;
                        ptr += sizeof(uint32_t);
                        if (put_user(death->cookie,
                                     (binder_uintptr_t __user *)ptr))
                                return -EFAULT;
                        ptr += sizeof(binder_uintptr_t);
                        binder_stat_br(proc, thread, cmd);
                        if (w->type == BINDER_WORK_CLEAR_DEATH_NOTIFICA
TION) {
                                list_del(&w->entry);
                                kfree(death);
                                binder_stats_deleted(BINDER_STAT_DEAT
H);
                        } else
                                list_move(&w->entry, &proc->delivered_d
eath);
                        if (cmd == BR_DEAD_BINDER)
                                goto done; /* DEAD_BINDER notifications
can cause transactions */
                } break;
                }

                if (!t)
                        continue;

                BUG_ON(t->buffer == NULL);
                if (t->buffer->target_node) {
                        struct binder_node *target_node = t->buffer->ta
rget_node;

                        tr.target.ptr = target_node->ptr;
                        tr.cookie =  target_node->cookie;
                        t->saved_priority = task_nice(current);
                        if (t->priority < target_node->min_priority &&
                            !(t->flags & TF_ONE_WAY))
                                binder_set_nice(t->priority);
                        else if (!(t->flags & TF_ONE_WAY) ||
                                 t->saved_priority > target_node->min_p
riority)
                                binder_set_nice(target_node->min_priori
ty);
                        cmd = BR_TRANSACTION;
                } else {
                        tr.target.ptr = 0;
                        tr.cookie = 0;
                        cmd = BR_REPLY;
                }
                tr.code = t->code;
                tr.flags = t->flags;
                tr.sender_euid = from_kuid(current_user_ns(), t->sende
r_euid);

                if (t->from) {
                        struct task_struct *sender = t->from->proc->ts
k;
```

```
                        tr.sender_pid = task_tgid_nr_ns(sender,
                                                        task_active_pi
d_ns(current));
                } else {
                        tr.sender_pid = 0;
                }

                tr.data_size = t->buffer->data_size;
                tr.offsets_size = t->buffer->offsets_size;
                tr.data.ptr.buffer = (binder_uintptr_t)(
                                        (uintptr_t)t->buffer->data +
                                        proc->user_buffer_offset);
                tr.data.ptr.offsets = tr.data.ptr.buffer +
                                        ALIGN(t->buffer->data_size,
                                            sizeof(void *));

                if (put_user(cmd, (uint32_t __user *)ptr))
                        return -EFAULT;
                ptr += sizeof(uint32_t);
                if (copy_to_user(ptr, &tr, sizeof(tr)))
                        return -EFAULT;
                ptr += sizeof(tr);

                trace_binder_transaction_received(t);
                binder_stat_br(proc, thread, cmd);

                list_del(&t->work.entry);
                t->buffer->allow_user_free = 1;
                if (cmd == BR_TRANSACTION && !(t->flags & TF_ONE_WAY))
{
                        t->to_parent = thread->transaction_stack;
                        t->to_thread = thread;
                        thread->transaction_stack = t;
                } else {
                        t->buffer->transaction = NULL;
                        kfree(t);
                        binder_stats_deleted(BINDER_STAT_TRANSACTION);
                }
                break;
        }

done:

        *consumed = ptr - buffer;
        if (proc->requested_threads + proc->ready_threads == 0 &&
            proc->requested_threads_started < proc->max_threads &&
            (thread->looper & (BINDER_LOOPER_STATE_REGISTERED |
             BINDER_LOOPER_STATE_ENTERED)) /* the user-space code fails
```

```
to */
                /*spawn a new thread if we leave this out */) {
                    proc->requested_threads++;

                    if (put_user(BR_SPAWN_LOOPER, (uint32_t __user *)buffe
r))
                              return -EFAULT;
                    binder_stat_br(proc, thread, BR_SPAWN_LOOPER);
          }
          return 0;
}
```

五、Binder内存

Binder进程间通信效率高的核心机制，如下图：

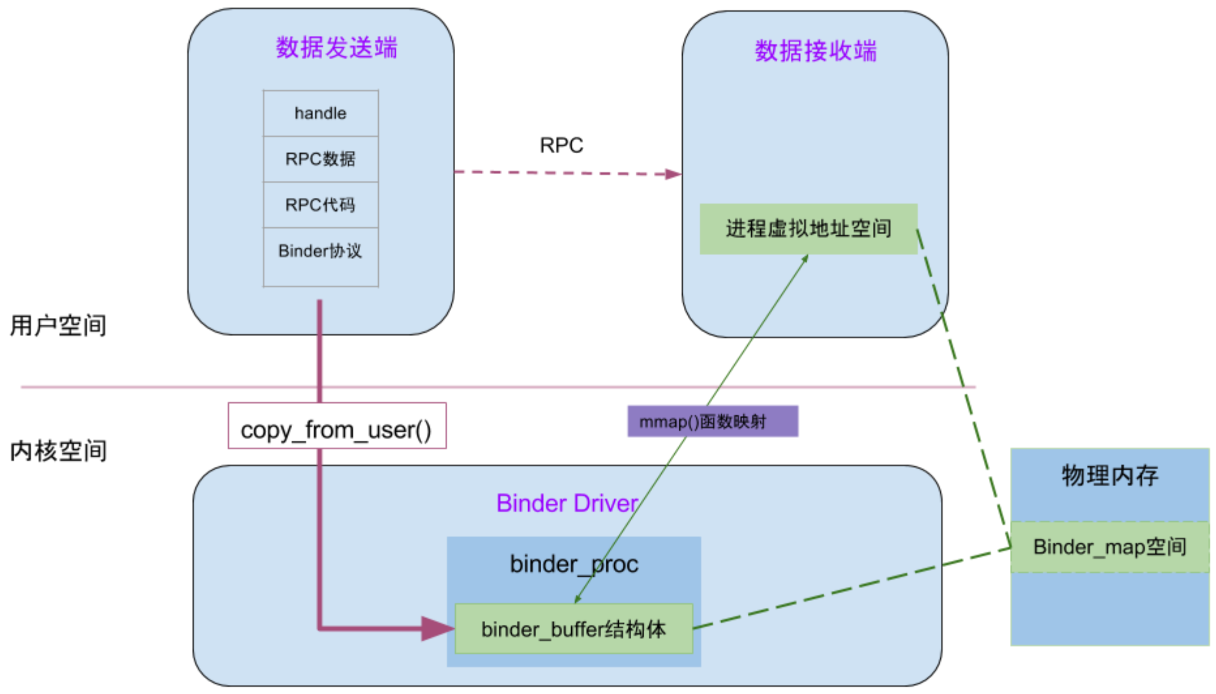


虚拟进程地址空间(vm_area_struct)和虚拟内核地址空间(vm_struct)都映射到同一块物理内存空间。当Client端与Server端发送数据时，Client（作为数据发送端）先从自己的进程空间把IPC通信数据 `copy_from_user` 拷贝到内核空间，而Server端（作为数据接收端）与内核共享数据，不再需要拷贝数据，而是通过内存地址空间

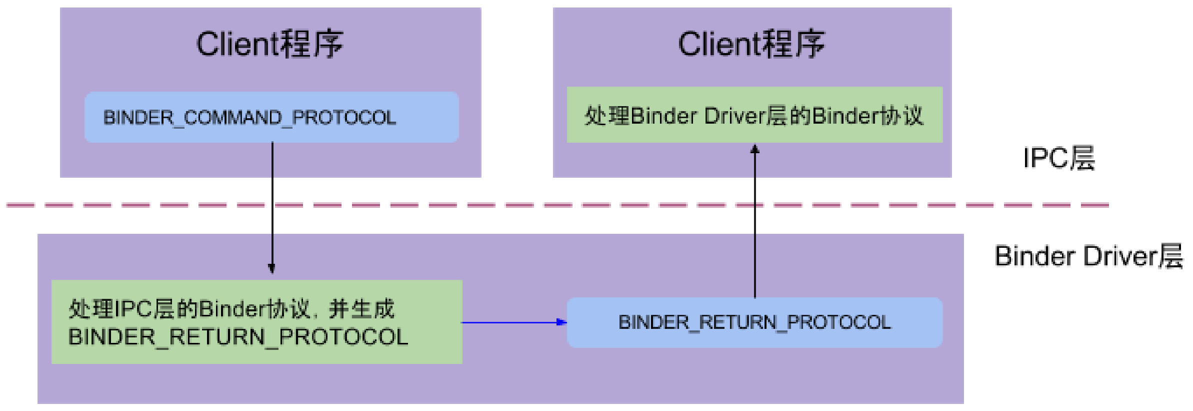
的偏移量，即可获悉内存地址，整个过程只发生一次内存拷贝。一般地做法，需要Client端进程空间拷贝到内核空间，再由内核空间拷贝到Server进程空间，会发生两次拷贝。

对于进程和内核虚拟地址映射到同一个物理内存的操作是发生在数据接收端，而数据发送端还是需要将用户态的数据复制到内核态。到此，可能有读者会好奇，为何不直接让发送端和接收端直接映射到同一个物理空间，那样就连一次复制的操作都不需要了，0次复制操作那就与Linux标准内核的共享内存的IPC机制没有区别了，对于共享内存虽然效率高，但是对于多进程的同步问题比较复杂，而管道/消息队列等IPC需要复制2两次，效率较低。这里就不先展开讨论Linux现有的各种IPC机制跟Binder的详细对比，总之Android选择Binder的基于速度和安全性的考虑。

下面这图是从Binder在进程间数据通信的流程图，从图中更能明了Binder的内存转移关系。



六、Binder协议指令



Binder协议包含在IPC数据中，分为两类:

1. `BINDER_COMMAND_PROTOCOL` ：binder请求码，以”BC_“开头，用于从IPC层传递到Binder Driver层；
2. `BINDER_RETURN_PROTOCOL` ：binder响应码，以”BR_“开头，用于从Binder Driver层传递到IPC层；

binder请求码

binder请求码，是用 `enum binder_driver_command_protocol` 来定义的，是用于应用程序向binder驱动设备发送请求消息，以BC_开头，总17条；(-代表目前不支持的请求码)

请求码	**参数类型**	**解释**
BC_TRANSACTION	binder_transaction_data	已发送的事务数据
BC_REPLY	binder_transaction_data	已发送的事务数据
BC_ACQUIRE_RESULT	-	-
BC_FREE_BUFFER	binder_uintptr_t(指针)	参数是指向接收的事务数据
BC_INCREFS	__u32	参数是descriptor
BC_ACQUIRE	__u32	参数是descriptor
BC_RELEASE	__u32	参数是descriptor
BC_DECREFS	__u32	参数是descriptor
BC_INCREFS_DONE	binder_ptr_cookie	binder的指针
BC_ACQUIRE_DONE	binder_ptr_cookie	binder的cookie
BC_ATTEMPT_ACQUIRE	-	-
BC_REGISTER_LOOPE	无参数	注册一个spawned looper线程
BC_ENTER_LOOPER	无参数	应用级线

		程进入binder loop
BC_EXIT_LOOPER	无参数	应用级线程退出binder loop
BC_REQUEST_DEATH_NOTIFICATION	binder_handle_cookie	请求死亡通知
BC_CLEAR_DEATH_NOTIFICATION	binder_handle_cookie	清除死亡通知
BC_DEAD_BINDER_DONE	binder_uintptr_t(指针)	死亡binder完成

binder响应码

binder响应码，是用 `enum binder_driver_return_protocol` 来定义的，是binder设备向应用程序回复的消息，以BR_开头，总18条；

响应码	**参数类型**	**解释**
BR_ERROR	__s32	错误码
BR_OK	无参数	ok
BR_TRANSACTION	binder_transaction_data	已接收的事务数据
BR_REPLY	binder_transaction_data	已接收的事务数据
BR_ACQUIRE_RESULT	-	-
BR_DEAD_REPLY	无参数	死亡回复
BR_TRANSACTION_COMPLETE	无参数	事务完成
BR_INCREFS	binder_ptr_cookie	binder的指针或cookie
BR_ACQUIRE	binder_ptr_cookie	binder的指针或cookie
BR_RELEASE	binder_ptr_cookie	binder的指针或cookie
BR_DECREFS	binder_ptr_cookie	binder的指针或cookie
BR_ATTEMPT_ACQUIRE	-	-

BR_NOOP	无参数	不做任何事，检验下一条命令
BR_SPAWN_LOOPER	无参数	创建新的服务线程
BR_FINISHED	-	-
BR_DEAD_BINDER	binder_uintptr_t(指针)	参数代表cookie
BR_CLEAR_DEATH_NOTIFICATION_DON	binder_uintptr_t(指针)	清除死亡通知，参数代表cookie
BR_FAILED_REPLY	无参数	最后一条transaction失败

BR_SPAWN_LOOPER：binder驱动已经检测到进程中没有线程等待即将到来的事务。那么当一个进程接收到这条命令时，该进程必须创建一条新的服务线程并注册该线程，该操作通过 `BC_ENTER_LOOPER`

七、 结构体附录

列举Binder相关的核心结构体

binder_write_read

用户空间程序和Binder驱动程序交互基本都是通过BINDER_WRITE_READ命令，来进行数据的读写操作。

类型	成员变量	解释
binder_size_t	rite_size	write_buffer的字节数
binder_size_t	write_consumed	已处理的write字节数
binder_uintptr_t	write_buffer	指向write数据区
binder_size_t	read_size	read_buffer的字节数
binder_size_t	read_consumed	已处理的read字节数
binder_uintptr_t	read_buffer	指向read数据区

- write_buffer变量：用于发送IPC(或IPC reply)数据，即传递经由Binder Driver的数据时使用。
- read_buffer 变量：用于接收来自Binder Driver的数据，即Binder Driver在接收IPC(或IPC reply)数据后，保存到read_buffer，再传递到用户空间；

write_buffer和read_buffer都是包含Binder协议命令和binder_transaction_data结构体。

- copy_from_user()将用户空间IPC数据拷贝到内核态binder_write_read结构体；
- copy_to_user()将用内核态binder_write_read结构体数据拷贝到用户空间；

binder_proc

binder_proc结构体：用于管理IPC所需的各种信息，拥有其他结构体的跟结构体。

类型	成员变量	解释
struct hlist_node	proc_node	进程节点
struct rb_root	threads	保存binder_thread结构体的红黑树的跟节点
struct rb_root	nodes	保存binder_node结构体的红黑树的根节点
struct rb_root	refs_by_desc	保存binder_ref实体的引用(以handle为key)
struct rb_root	refs_by_node	binder实体的引用（以ptr为key）
int	pid	创建binder_proc结构体的进程id
struct vm_area_struct *	vma	指向进程虚拟地址空间的指针
struct mm_struct *	vma_vm_mm;	
struct task_struct *	tsk	创建binder_proc结构体的进程结构体
struct files_struct *	files	
	struct hlist_node	deferred_work_node
int	deferred_work	
void *	buffer	接收IPC数据的内核地址空间指针(binder_buffer)
ptrdiff_t	user_buffer_offset	内核空间与用户空间的地址偏移量
struct	list_head buffers	
struct	rb_root free_buffers	空闲buffer
struct	rb_root allocated_buffers	已分配buffer
size_t	free_async_space	

struct page **	pages	描述物理内存页的数据结构
size_t	buffer_size	接收IPC数据的内核地址空间大小
uint32_t	buffer_free	
struct list_head	todo	进程将要做的事
wait_queue_head_t	wait	等待队列
struct binder_stats	stats	
struct list_head	delivered_death	
int	max_threads	最大线程数
int	requested_threads	
int	requested_threads_started	
int	ready_threads	
long	default_priority	默认优先级
struct dentry *	debugfs_entry	

- 其中rb_root是一种红黑树结构，红黑树作为自平衡的二叉树树便于查询和维护。
  - `refs_by_desc` 记录的是 `binder_transaction_data` 结构体中target的 `handle`；
  - `refs_by_node` 记录的是 `binder_transaction_data` 结构体中target的 `ptr`；
- `user_buffer_offset` 是虚拟进程地址与虚拟内核地址的差值，也就是说同一物理地址，当内核地址为kernel_addr，则进程地址为proc_addr = kernel_addr + user_buffer_offset。

binder_thread

binder_thread结构体代表当前binder操作所在的线程

类型	**成员变量**	**解释**
struct binder_proc *	proc	线程所属的进程
struct rb_node	rb_node	
int	pid	线程pid
int	looper	
struct binder_transaction *	transaction_stack	正在处理的事务
struct list_head	todo	将要处理的数据列表

uint32_t	return_error	write失败后，返回的错误码
uint32_t	return_error2	write失败后，返回的错误码
wait_queue_head_t	wait	等待队列的队头
struct binder_stats	stats	binder线程的统计信息

binder_buffer

类型	成员变量	解释
struct list_head	entry	空闲和已分配实体的地址
struct rb_node	rb_node	空闲实体大小或已分配实体地址
unsigned	free	标记是否是空闲buffer，占位1bit
unsigned	allow_user_free	占位1bit
unsigned	async_transaction	占位1bit
unsigned	debug_id	占位29bit
struct binder_transaction *	transaction	
struct binder_node *	target_node	Binder实体
size_t	data_size	
size_t	offsets_size	
uint8_t	data[0]	

每一个binder_buffer分为空闲和已分配的，通过free标记来区分。空闲和已分配的binder_buffer通过各自的成员变量rb_node分别连入binder_proc的free_buffers(红黑树)和allocated_buffers(红黑树)。

binder_node

binder_node代表一个binder实体

类型	成员变量	解释
int	debug_id	
struct binder_work	work	
struct rb_node	rb_node	binder正常使用，union
struct hlist_node	dead_node	binder进程已销毁，union
struct binder_proc	proc	binder所在的进程

*		
struct hlist_head	refs	
int	internal_strong_refs	
int	local_weak_refs	
int	local_strong_refs	
binder_uintptr_t	ptr	Binder实体所在用户空间的地址
binder_uintptr_t	cookie	附件数据
unsigned	has_strong_ref	占位1bit
unsigned	pending_strong_ref	占位1bit
unsigned	has_weak_ref	占位1bit
unsigned	pending_weak_ref	占位1bit
unsigned	has_async_transaction	占位1bit
unsigned	accept_fds	占位1bit
unsigned	min_priority	占位8bit
struct list_head	async_todo	

binder_state

类型	成员变量	解释
int	fd	文件描述符
void *	mapped	内存映射地址
size_t	mapsize	内存映射大小

flat_binder_object

flat_binder_object结构体代表Binder对象在两个进程间传递的扁平结构。

类型	成员变量	解释
__u32	type	
__u32	flags	
binder_uintptr_t	binder	local对象，union
__u32	handle	remote对象，union
binder_uintptr_t	cookie	local对象相关的额外数据

binder_transaction

```
struct binder_transaction {
        int debug_id;
        struct binder_work work;
        struct binder_thread *from;
        struct binder_transaction *from_parent;
        struct binder_proc *to_proc;
        struct binder_thread *to_thread;
        struct binder_transaction *to_parent;
        unsigned need_reply:1;
        struct binder_buffer *buffer;
        unsigned int    code;
        unsigned int    flags;
        long    priority;
        long    saved_priority;
        kuid_t  sender_euid;
};
```

binder_transaction_data

当BINDER_WRITE_READ命令的目标是本地Binder node时，target使用ptr，否则使用handle。只有当这是Binder node时，cookie才有意义，表示附加数据，由进程自己解释。

```
struct binder_transaction_data {
        union {
                __u32   handle;    //binder实体的引用
                binder_uintptr_t ptr;    //Binder实体在当前进程的地址
        } target;  //RPC目标
        binder_uintptr_t        cookie;
        __u32           code;           //RPC代码

        __u32           flags;
        pid_t           sender_pid;  //发送者进程的pid
        uid_t           sender_euid; //发送者进程的euid
        binder_size_t   data_size;
        binder_size_t   offsets_size;

        union {
                struct {
                        binder_uintptr_t        buffer;
                        binder_uintptr_t        offsets;
                } ptr;
                __u8    buf[8];
        } data;   //RPC数据
};
```

喜欢

1 条评论

最新 最早 最热

嘿嘿参北斗哇

(http://yhh.duoshuo.com/user-url/?user_id=6239442989723158000)
(http://yhh.duoshuo.com/user-url/?user_id=6239442989723158000)

刚刚 回复 顶 转发

嘿嘿参北斗哇 (http://www.baidu.com/p/嘿嘿参北斗哇) 帐号管理

(http://duoshuo.com/settings/avatar/)

说点什么吧…

分享到: 发布

多说 (http://duoshuo.com)

✉ gityuan@gmail.com (mailto:gityuan@gmail.com) · Github (https://github.com/yuanhuihui) · 天道酬勤 · © 2015 Yuanhh · Jekyll (https://github.com/jekyll/jekyll) theme by HyG (https://github.com/Gaohaoyang)