binder驱动-------之内存映射篇

标签： android ibinder驱动

2014-06-25 15:52 3240人阅读 评论(2) 收藏 举报

分类：

linux binder

目录(?) [+]

1：binder内存管理概述

binder一次跨进程通讯，只需要一次拷贝(原因后面会解析)，而一般的像socket通讯则需要两次拷贝；参与binder通讯的进程，无论是client还是服务器端，他们都会通过调用ProcessState::self()函数来建立自己的初步映射，为什么说是初步影射呢，因为binder_mmap为我们分配了指定长度的虚拟地址，但却只是建立一个物理页的影射，其他的虚拟地址都还暂未建立虚拟到物理的影射。下面我们的讲述从ProcessState::self()函数开始，该函数使用典型的单列模式：如已创建该实例则直接返回，如果没有创建，则创建返回这个实例，这里需要锁来防止创建两个同类型的实例，该函数还是static类型的，所以可以在系统的任何地方调用。

[cpp]

```cpp
sp<ProcessState> ProcessState::self()
{
    Mutex::Autolock _l(gProcessMutex);
    if (gProcess != NULL) {
        return gProcess;
    }
    gProcess = new ProcessState;
    return gProcess;
}
```

展开构建函数：ProcessState，如下：

[cpp]

```cpp
ProcessState::ProcessState()
    : mDriverFD(open_driver())//打开/dev/binder设备驱动
    , mVMStart(MAP_FAILED)
    , mManagesContexts(false)
    , mBinderContextCheckFunc(NULL)
    , mBinderContextUserData(NULL)
    , mThreadPoolStarted(false)
    , mThreadPoolSeq(1)
{
    if (mDriverFD >= 0) {
        // XXX Ideally, there should be a specific define for whether we
        // have mmap (or whether we could possibly have the kernel module
        // availabla).
#if !defined(HAVE_WIN32_IPC)
        // mmap the binder, providing a chunk of virtual address space to receive transactions.
        mVMStart = mmap(0, BINDER_VM_SIZE, PROT_READ, MAP_PRIVATE | MAP_NORESERVE, mDriverFD, 0);
```

```
建立影射，直接调用到驱动的binder_mmap函数
        if (mVMStart == MAP_FAILED) {
            // *sigh*
            ALOGE("Using /dev/binder failed: unable to mmap transaction memory.\n");
            close(mDriverFD);
            mDriverFD = -1;
        }
#else
        mDriverFD = -1;
#endif
    }

    LOG_ALWAYS_FATAL_IF(mDriverFD < 0, "Binder driver could not be opened.  Terminating.");
}
```

以上函数：打开了/dev/binder设备节点，并且建立了初步的映射，binder driver中的mmap函数具体分析见下一节。

先大概说下，整个函数执行完后，都做了些什么事情：

为进程的用户和内核空间分配了一段虚拟地址，这段虚拟地址将用于binder内存的访问。binder的物理内存页由binder驱动负责分配，然后这些物理页的访问，将分为进程的用户空间和进程内核空间。由于进程的用户空间和内核空间的虚拟地址范围是不一样的，所以这里分配的一段虚拟地址将包括进程的用户空间地址和内核空间地址。

并且在映射建立初始阶段，只是映射了一个物理页，而不是为整个虚拟地址空间都建立相应的物理影射，目的是充分高效的利用内存，只是到需要用到的时候才建立映射，用完后马上释放映射，这样可以充分高效的利用系统的物理内存页。

内核刚开始只是分配了一个物理页，并且分别将这个物理页映射到进程的内核虚拟地址空间V1（修改内核空间的页表映射）和进程的用户虚拟地址空间V2（修改用户空间的页表映射）。在用户空间访问V1和在内核空间访问V2，其实都是访问的是同一个物理内存块，从而实现进程的内核和用户空间共享同一块物理内存的目的。这样binder驱动在内核空间，将一段数据拷贝到这个物理页，则该进程的用户空间则不需要copy_to_user()即可以同步看到内核空间的修改，并能够访问这段物理内存。

每个进程都有属于自己的一块binder内存，这块内存的大小就是在进程调用mmap系统调用时分配的，binder_proc->buffer指向这块内存在内核空间的首地址，这块内存块在进程的用户空间的首地址为：ProcessState::mVMStart,他们的差值存储在binder_proc->user_buffer_offset变量中，这样在已知内核地址情况下，在用户空间只需要加上这个user_buffer_offset偏移值，即可以得到在用户空间访问的地址，从而达到访问同一物理内存地址的目的。

每个进程的binder内存，组织在以下三个列表中：

- struct list_head buffers;//这个列表连接所有的内存块，以地址的大小为顺序，各内存块首尾相连
- struct rb_root free_buffers;//连接所有的已建立映射的虚拟内存块，以内存的大小为index组织在以该节点为根

的红黑树下

- struct rb_root allocated_buffers;//连接所有已经分配的虚拟内存块，以内存块的开始地址为index组织在以该节点为根的红黑树下

如果在进程的binder事物传递过程中，需要为target 进程分配内存，则调用binder_alloc_buf(struct binder_proc *proc, size_t data_size,
size_t offsets_size, int is_async)函数为target进程分配需要的内存。

binder_alloc_buf函数首先从target进程的binder_proc->free_buffers树中查找size（=data_size+offsets_size）大小的内存块，如果能够精确匹配到，则返回该内存块，并将该内存从binder_proc->free_buffers树中删除，并将它插入到binder_proc->allocated_buffers树中

如果不能找到精确的匹配的大小块，则从binder_proc->free_buffers树中查找最接近size，但又大于size的内存，找到后返回该内存块，并将该内存从binder_proc->free_buffers树中删除，并将它插入到binder_proc->allocated_buffers树中

如果在binder_proc->free_buffers树中找不任何一块比size大的空闲内存块，则通过binder_update_page_range函数分配更多地物理内存，并建立相应的虚拟到物理地址的映射，由于映射的最小单元是一个物理页，如果所需的size又比一个page size小，则将这块内存分为两部分，前一部分就是大小为size大小的内存块，返回该该内存块，并将它插入到binder_proc->allocated_buffers树中，而剩余的内存块则直接插入到binder_proc->free_buffers树中

binder_free_buf函数：（to be continued）

2：binder的mmap

用户空间的ProcessState函数中的mmap系统调用会调用到binder驱动中的binder_mmap函数。先看下系统调用mmap的原型：

void *mmap(void *addr, size_t length int " prot ", int " flags , int fd, off_t offset);

mmap() creates a new mapping in the virtual address space of the calling process. The starting address for the new mapping is specified in*addr*. The *length* argument specifies the length of the mapping. If *addr* is NULL, then the kernel chooses the address at which to create the mapping; this is the most portable method of creating a new mapping. If*addr* is not NULL, then the kernel takes it as a hint about where to place the mapping; on Linux, the mapping will be created at a nearby page boundary. The address of the new mapping is returned as the result of the call.

我们看servicemanager进程的maps图

```
/system/etc/bluetooth # cat /proc/74/maps
40040000-40042000 r-xp 00000000 1f:09 1046       /system/bin/servicemanager
40042000-40043000 r--p 00001000 1f:09 1046       /system/bin/servicemanager
40043000-40044000 rw-p 00002000 1f:09 1046       /system/bin/servicemanager
40044000-40052000 r-xp 00000000 1f:09 960        /system/bin/linker
40052000-40053000 r--p 00000000 00:00 0
40053000-40054000 r--p 0000e000 1f:09 960        /system/bin/linker
40054000-40055000 rw-p 0000f000 1f:09 960        /system/bin/linker
40055000-4005f000 rw-p 00000000 00:00 0
4005f000-40062000 r-xp 00000000 1f:09 783        /system/lib/liblog.so
40062000-40063000 r--p 00002000 1f:09 783        /system/lib/liblog.so
40063000-40064000 rw-p 00003000 1f:09 783        /system/lib/liblog.so
40064000-400a9000 r-xp 00000000 1f:09 758        /system/lib/libc.so
400a9000-400aa000 ---p 00000000 00:00 0
400aa000-400ac000 r--p 00045000 1f:09 758        /system/lib/libc.so
400ac000-400ae000 rw-p 00047000 1f:09 758        /system/lib/libc.so
400ae000-400b9000 rw-p 00000000 00:00 0
400b9000-400ba000 r-xp 00000000 1f:09 690        /system/lib/libstdc++.so
400ba000-400bb000 r--p 00000000 1f:09 690        /system/lib/libstdc++.so
400bb000-400bc000 rw-p 00001000 1f:09 690        /system/lib/libstdc++.so
400bc000-400d1000 r-xp 00000000 1f:09 661        /system/lib/libm.so
400d1000-400d2000 r--p 00014000 1f:09 661        /system/lib/libm.so
400d2000-400d3000 rw-p 00015000 1f:09 661        /system/lib/libm.so
400d3000-400db000 r--s 00000000 00:0d 1258       /dev/__properties__ (deleted)
400db000-400fb000 r--p 00000000 00:0d 844        /dev/binder
40489000-4048a000 rw-p 00000000 00:00 0          [heap]
bea56000-bea77000 rw-p 00000000 00:00 0          [stack]
ffff0000-ffff1000 r-xp 00000000 00:00 0          [vectors]
```

可以看到mmap的长度是128k。我们通过如下代码：frameworks/base/cmds/servicemanager/service_manager.c

```cpp
int main(int argc, char **argv)
{
    struct binder_state *bs;
    void *svcmgr = BINDER_SERVICE_MANAGER;

    bs = binder_open(128*1024);

    if (binder_become_context_manager(bs)) {
        ALOGE("cannot become context manager (%s)\n", strerror(errno));
        return -1;
    }

    svcmgr_handle = svcmgr;
    binder_loop(bs, svcmgr_handler);
    return 0;
}
```

可以看到他mmap系统调用规定的地址长度就是128K

而我们看蓝牙apk的进程中的maps：

```
/system/etc/bluetooth # cat /proc/7529/maps | grep "binder"
4013d000-4015b000 r-xp 00000000 1f:09 773        /system/lib/libbinder.so
4015b000-40160000 r--p 0001d000 1f:09 773        /system/lib/libbinder.so
40160000-40161000 rw-p 00022000 1f:09 773        /system/lib/libbinder.so
5c525000-5c623000 r--p 00000000 00:0d 844        /dev/binder
/system/etc/bluetooth #
```

他映射的虚拟地址长度为：0xFE000，即为frameworks/native/libs/binder/processState.cpp中的#define

BINDER_VM_SIZE ((1*1024*1024) - (4096 *2))的大小。从上面两个列子我们知道，进程所属的binder内存，也就是系统调用mmap所分配的地址空间，并可以通过cat proc/pid/maps命令查看到。

我们再看/drivers/staging/android/binder.c驱动中static int binder_mmap(struct file *filp, struct vm_area_struct *vma)函数的含义：

vma->vm_start和vma->vm_end即为此次映射内核为我们分配的开始地址和结束地址，他们差值就是系统调用mmap中的length的值。而vma->vm_start的则是系统调用mmap调用的返回值。需要注意的是vma->vm_start和vma->vm_end都是调用进程的用户空间的虚拟地址，他们地址范围可以通过如下命令：cat /proc/pid/maps | grep "/dev/binder"看到，如上面两个图，针对bluetooth.apk进程，他们vma->vm_start和vma->vm_end的值分别为：0x5c525000和0x5c623000

进程在调用mmap系统调用时，就会调用到binder驱动对应的file_operations->mmap()成员函数，即binder_mmap函数。

```cpp
static int binder_mmap(struct file *filp, struct vm_area_struct *vma)
{
    int ret;
    struct vm_struct *area;
    struct binder_proc *proc = filp->private_data;
    const char *failure_string;
    struct binder_buffer *buffer;//内核进入这个函数时，就已经预先为此次映射分配好了调用进程在用户空间的虚拟地址范围（vma->vm_start，vma->vm_end）

    if ((vma->vm_end - vma->vm_start) > SZ_4M)
        vma->vm_end = vma->vm_start + SZ_4M;

    binder_debug(BINDER_DEBUG_OPEN_CLOSE,
             "binder_mmap: %d %lx-%lx (%ld K) vma %lx pagep %lx\n",
             proc->pid, vma->vm_start, vma->vm_end,
             (vma->vm_end - vma->vm_start) / SZ_1K, vma->vm_flags,
             (unsigned long)pgprot_val(vma->vm_page_prot));

    if (vma->vm_flags & FORBIDDEN_MMAP_FLAGS) {
        ret = -EPERM;
        failure_string = "bad vm_flags";
        goto err_bad_arg;
    }
    vma->vm_flags = (vma->vm_flags | VM_DONTCOPY) & ~VM_MAYWRITE;

    mutex_lock(&binder_mmap_lock);
    if (proc->buffer) {
        ret = -EBUSY;
        failure_string = "already mapped";
        goto err_already_mapped;
    }
        //为进程所在的内核空间申请与用户空间同样长度的虚拟地址空间，这段空间用于内核来访问和管理binder内存区域
    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;
    }//对应内核虚拟地址的开始，即为binder内存的开始地址
    proc->buffer = area->addr;
    proc->user_buffer_offset = vma->vm_start - (uintptr_t)proc->buffer;//同一块物理内存，内核的虚拟地址同应用空间的虚拟地址的差
    mutex_unlock(&binder_mmap_lock);

#ifdef CONFIG_CPU_CACHE_VIPT
    if (cache_is_vipt_aliasing()) {
        while (CACHE_COLOUR((vma->vm_start ^ (uint32_t)proc->buffer))) {
            printk(KERN_INFO "binder_mmap: %d %lx-%lx maps %p bad alignment\n", proc->pid, vma->vm_start, vma->vm_end, proc->buffer);
            vma->vm_start += PAGE_SIZE;
        }
    }
#endif//用于存放内核分配的物理页的页描述指针：struct page，每个物理页对应这样一个struct page结构
    proc->pages = kzalloc(sizeof(proc->pages[0]) * ((vma->vm_end - vma->vm_start) / PAGE_SIZE), GFP_KERNEL);
    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;//映射的长度即为binder内存的大小

    vma->vm_ops = &binder_vm_ops;//设定
    vma->vm_private_data = proc;//为binder内存的最开始的一个页的地址建立虚拟到物理页的映射，其他的虚拟地址都还没有建立相应的映射，没有映射也就还不能够被访问
    if (binder_update_page_range(proc, 1, proc->buffer, proc->buffer + PAGE_SIZE, vma)) {
        ret = -ENOMEM;
        failure_string = "alloc small buf";
        goto err_alloc_small_buf_failed;
    }
    buffer = proc->buffer;
    INIT_LIST_HEAD(&proc->buffers);//proc->buffers用来连接所有已建立映射的binder内存块
    list_add(&buffer->entry, &proc->buffers);
    buffer->free = 1;
    binder_insert_free_buffer(proc, buffer);//将刚分配的一个page大小的binder内存块插入到proc->free_buffers红黑树中
    proc->free_async_space = proc->buffer_size / 2;
    barrier();
    proc->files = get_files_struct(proc->tsk);
    proc->vma = vma;
    proc->vma_vm_mm = vma->vm_mm;

    /*printk(KERN_INFO "binder_mmap: %d %lx-%lx maps %p\n",
         proc->pid, vma->vm_start, vma->vm_end, proc->buffer);*/
    return 0;

err_alloc_small_buf_failed:
    kfree(proc->pages);
    proc->pages = NULL;
err_alloc_pages_failed:
    mutex_lock(&binder_mmap_lock);
    vfree(proc->buffer);
    proc->buffer = NULL;
```

```
err_get_vm_area_failed:
err_already_mapped:
    mutex_unlock(&binder_mmap_lock);
err_bad_arg:
    printk(KERN_ERR "binder_mmap: %d %lx-%lx %s failed %d\n",
           proc->pid, vma->vm_start, vma->vm_end, failure_string, ret);
    return ret;
}
```

上面binder_update_page_range函数就是为进程的内核空间和进程的用户空间针对同一块物理内存建立映射，这样进程的用户空间和内核空间就可以共享该物理内存了。

[cpp]

```cpp
static int binder_update_page_range(struct binder_proc *proc, int allocate,
                    void *start, void *end,
                    struct vm_area_struct *vma)
{
    void *page_addr;
    unsigned long user_page_addr;
    struct vm_struct tmp_area;
    struct page **page;
    struct mm_struct *mm;

    binder_debug(BINDER_DEBUG_BUFFER_ALLOC,
             "binder: %d: %s pages %p-%p\n", proc->pid,
             allocate ? "allocate" : "free", start, end);

    if (end <= start)//表示已经建立过映射，不需再映射
        return 0;

    trace_binder_update_page_range(proc, allocate, start, end);

    if (vma)
        mm = NULL;
    else
        mm = get_task_mm(proc->tsk);

    if (mm) {
        down_write(&mm->mmap_sem);
        vma = proc->vma;
        if (vma && mm != proc->vma_vm_mm) {
            pr_err("binder: %d: vma mm and task mm mismatch\n",
                proc->pid);
            vma = NULL;
        }
    }

    if (allocate == 0)//表示拆除已经建立的映射
        goto free_range;

    if (vma == NULL) {
        printk(KERN_ERR "binder: %d: binder_alloc_buf failed to "
               "map pages in userspace, no vma\n", proc->pid);
        goto err_no_vma;
    }
```

```
43.
44.     for (page_addr = start; page_addr < end; page_addr += PAGE_SIZE) {
45.         int ret;
46.         struct page **page_array_ptr;
47.         page = &proc->pages[(page_addr - proc->buffer) / PAGE_SIZE];
48.
49.         BUG_ON(*page);
50.         *page = alloc_page(GFP_KERNEL | __GFP_ZERO);//分配一个物理页，并将该物理页的struct page指针值存放在proc->pages二维数组中
51.         if (*page == NULL) {
52.             printk(KERN_ERR "binder: %d: binder_alloc_buf failed "
53.                    "for page at %p\n", proc->pid, page_addr);
54.             goto err_alloc_page_failed;
55.         }
56.         tmp_area.addr = page_addr;//映射对应的内核虚拟地址
57.         tmp_area.size = PAGE_SIZE + PAGE_SIZE /* guard page? */;//映射对应的大小
58.         page_array_ptr = page;//该物理页对应的struct page结构体，用这个结构体可以完全的描述这个物理页
59.         ret = map_vm_area(&tmp_area, PAGE_KERNEL, &page_array_ptr);//为内核的这段虚拟地址建立虚拟到物理页的映射
60.         if (ret) {
61.             printk(KERN_ERR "binder: %d: binder_alloc_buf failed "
62.                    "to map page at %p in kernel\n",
63.                    proc->pid, page_addr);
64.             goto err_map_kernel_failed;
65.         }
66.         user_page_addr =
67.             (uintptr_t)page_addr + proc->user_buffer_offset;//由内核的虚拟地址得到用户空间的虚拟地址
68.         ret = vm_insert_page(vma, user_page_addr, page[0]);//为应用空间的这段虚拟地址建立虚拟到物理的映射
69.         if (ret) {                      //至此应用空间和内核空间都映射到了同一块物理页内存
70.             printk(KERN_ERR "binder: %d: binder_alloc_buf failed "
71.                    "to map page at %lx in userspace\n",
72.                    proc->pid, user_page_addr);
73.             goto err_vm_insert_page_failed;
74.         }
75.         /* vm_insert_page does not seem to increment the refcount */
76.     }
77.     if (mm) {
78.         up_write(&mm->mmap_sem);
79.         mmput(mm);
80.     }
81.     return 0;
82.
83. free_range://释放影射
84.     for (page_addr = end - PAGE_SIZE; page_addr >= start;
85.          page_addr -= PAGE_SIZE) {
86.         page = &proc->pages[(page_addr - proc->buffer) / PAGE_SIZE];//得到该段虚拟地址对应的struct page结构体
87.         if (vma)
88.             zap_page_range(vma, (uintptr_t)page_addr +//拆除应用空间的映射表
89.                 proc->user_buffer_offset, PAGE_SIZE, NULL);
90. err_vm_insert_page_failed://查出内核空间的映射表
91.         unmap_kernel_range((unsigned long)page_addr, PAGE_SIZE);
92. err_map_kernel_failed://释放对应的物理内存页，这样这块物理内存页又可以被系统其他地方使用了
93.         __free_page(*page);
```

```
94.         *page = NULL;
95. err_alloc_page_failed:
96.         ;
97.     }
98. err_no_vma:
99.     if (mm) {
100.         up_write(&mm->mmap_sem);
101.         mmput(mm);
102.     }
103.     return -ENOMEM;
104. }
```

刚开始binder_mmap只是为进程映射一个物理页，当后续如果映射的内存不够用的时候，根据需要增加建立相应的新的物理页映射，用于满足接下来的物理内存分配需求。而这个动作就是在binder_alloc_buf函数中实现的。就是当系统的物理内存不够用的时候，通过binder_alloc_buf动态的建立需要的物理内存的映射（但最少是一个物理页，因为最少的映射单元就是一个物理页）

在binder_mmap函数中，建立内存分配和映射的操作函数集（binder_vm_ops）时，并没有像通用的mmap驱动那样去设置fault()函数，让系统如果发生缺页异常时，调用这个fault函数来分配物理内存，并且建立相应的虚拟地址到物理地址的映射，而是在binder_alloc_buf函数中去建立动态的物理内存的分配和虚拟地址到物理地址的映射，这样做的好处就是：减少了系统中断的次数，加速binder内存发生缺页时的处理速度。因为fault()函数是在异常处理被调用的，即中断处理中被调用的，过多的这种处理会让系统对用户的输入反应迟钝。

3：共享内存在binder通讯中的使用

3.1 共享内存的分配

在通过进行binder事物的传递时，如果一个binder事物（用struct binder_transaction结构体表示）需要使用到内存，就会调用binder_alloc_buf函数分配此次binder事物需要的内存空间。需要注意的是：从目标进程所在binder内存空间分配所需的内存：（drivers/staging/android/binder.c）

[cpp]

```
01.     t->sender_euid = proc->tsk->cred->euid;
02.     t->to_proc = target_proc;//事物的目标进程
03.     t->to_thread = target_thread;//事物的目标线程
04.     t->code = tr->code;//事物代码
05.     t->flags = tr->flags;
06.     t->priority = task_nice(current);//线程的优先级的迁移
07.
08.     trace_binder_transaction(reply, t, target_node);
09.
10.     t->buffer = binder_alloc_buf(target_proc, tr->data_size,//从target进程的binder内存空间分配所需的内存大小（tr->data_size+tr->offsets_size）
11.         tr->offsets_size, !reply && (t->flags & TF_ONE_WAY));
12.     if (t->buffer == NULL) {
13.         return_error = BR_FAILED_REPLY;
14.         goto err_binder_alloc_buf_failed;
15.     }
16.     t->buffer->allow_user_free = 0;
```

```
17.     t->buffer->debug_id = t->debug_id;
18.     t->buffer->transaction = t;//该binder_buffer对应的事务
19. t->buffer->target_node = target_node;//该事物对应的目标binder实体
```

至此已经为事务传输分配了所需的内存，分配的binder内存的开始地址存放在binder_buffer->data域中，binder_buffer->data_size存放此次事务对应的数据大小，binder_buffer->offsets_size表示数据中包含的binder对象的数目。该事务初始化好，会被放到target线程所在的todo列表中，被唤醒target线程。

3.2 共享内存的访问

阻塞在binder_thread_read函数中的thread->wait等待队列上的目标线程，在被唤醒后，从自己的thread->todo双向列表中取出在binder_transaction（）函数放入到该队列的事务（struct binder_transaction），并用这个结构体来初始化struct binder_transaction_data结构体：

[cpp]

```
01. tr.code = t->code;
02. tr.flags = t->flags;
03. tr.sender_euid = t->sender_euid;
04.
05. if (t->from) {
06.     struct task_struct *sender = t->from->proc->tsk;
07.     tr.sender_pid = task_tgid_nr_ns(sender,
08.                     current->nsproxy->pid_ns);
09. } else {
10.     tr.sender_pid = 0;
11. }
12.
13. tr.data_size = t->buffer->data_size;//事务对应的数据的大小
14. tr.offsets_size = t->buffer->offsets_size;
15. tr.data.ptr.buffer = (void *)t->buffer->data +//得到事物对应的内核数据在用户空间的访问地址
16.             proc->user_buffer_offset;
17. tr.data.ptr.offsets = tr.data.ptr.buffer +//得到事务对应的数据在用户空间的访问地址
18.             ALIGN(t->buffer->data_size,
19.                 sizeof(void *));
20.
21.     if (put_user(cmd, (uint32_t __user *)ptr))//拷贝返回命令
22.     return -EFAULT;
23. ptr += sizeof(uint32_t);
24. if (copy_to_user(ptr, &tr, sizeof(tr)))//注意：只是拷贝了命令对应的固定大小的tr参数，并没有拷贝
    tr.data.ptr.buffer指向的内容            return -EFAULT;
25. ptr += sizeof(tr);
```

最终struct binder_transaction_data结构体会通过ioctl系统调用以BR_xx命令+参数（binder_transaction_data结构体）的形式返回给target进程的用户空间，并被用户空间所处理。target进程的用户空间直接使用tr.data.ptr.buffer的值就可以访问到远端传递过来的数据，而不需要做mem copy的动作将内核的数据通过copy_to_user来拷贝到用户空间；同样用户空间直接使用tr.data.ptr.offsets的值就可以直接访问binder对象的偏移数组

下一篇 蓝牙协议-----之pan profile on bluedroid

顶
0