学习小组-Binder中mmap原理分享

一. 前提知识介绍

1. 进程隔离

进程隔离：在操作系统中，进程与进程间的**内存都是不共享的**。
好处：
① 用户程序可以访问任意内存，寻址内存的每个字节，这样就很容易破坏操作系统，造成操作系统崩溃。
② 如果需要同时运行多个程序特别困难，比如你想同时运行一个微信和一个 QQ 音乐都不行.(因为如果他们都使用到同一个内存区域导致数据覆盖)

2. 虚拟内存地址

假设计算机是**32位 ，**我们都知道代表了32位地址总线，代表这个计算机可以寻址的范围是0-0xFFFFFFFF(4G地址空间)，但是我配了**256M 内存条，** 计算机怎么处理?
虚拟内存地址大小就是这里的 4G,与计算机的位数有关系

(1) 内存分页

虚拟内存空间按照一定大小进行等量的划分成“**页**” **（linux 设置为4kB）**
物理内存按照按照一定大小进行等量划分为“**页帧**”
一般虚拟内存>物理内存

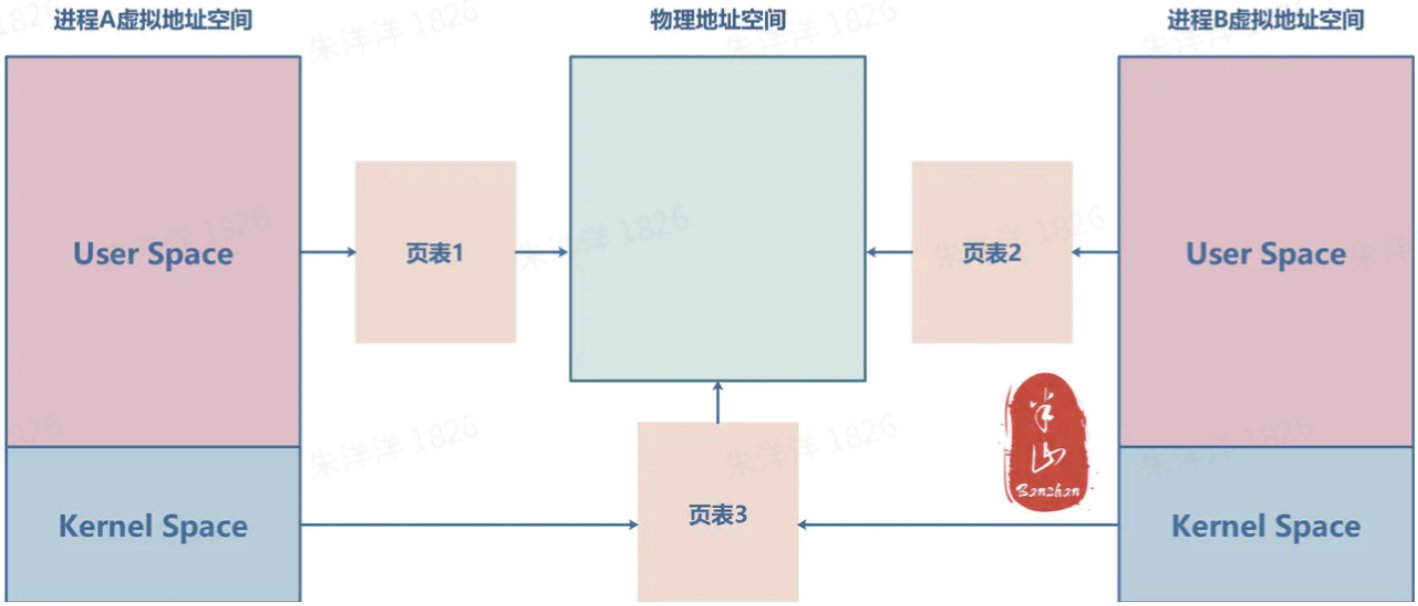
(2)虚拟内存地址与物理内存的映射方式

页面失效(Page Fatal)：操作系统找到一个**最近最少使用的帧，让其失效，并写入到磁盘上**，随后把需要**访问的页**放到**页帧**上面，**修改页表的映射关系**，从而保证所有的页都有调度的可能。
虚拟内存地址 = 页号+偏移量
页号：通过页表映射到页帧
偏移量：页大小(就是前面4kb)
举个寻址例子：一个虚拟地址页号是 8 ，偏移量是4kb，

1. 通过页表找到对应的页帧号，如果该页不再内存中则页面失效
2. 将页帧号和偏移量交给MMU(**Memory Management Unit**), 就可以组成一个实际上的物理内存地址，就可以访问物理内存数据

页表的大致数据结构

虚拟页号	物理页号	状态位（是否调入内存）	访问字段（用于记录最近被访问的次数或者上次访问的时间。用于置换算法）	修改位（页面被调用内存之后是否被修改，防止多次写入外存）	外存地址（页面外存的位置，常用于外面的硬盘）
0	a	0	0	0	x
1	b	1	2	1	y
2	c	1	3	1	z

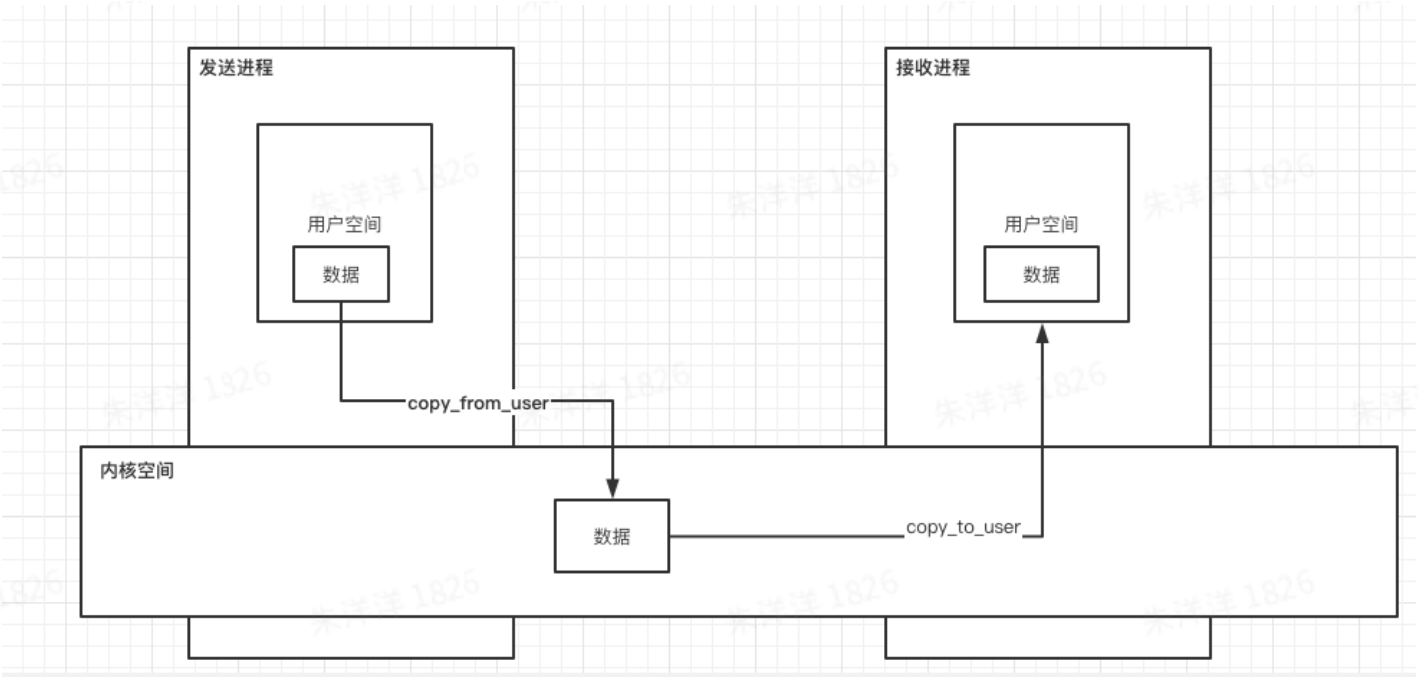


二. mmap的介绍

原理就是：修改用户空间的页表，使得内核空间地址与用户空间地址映射同一个物理的页面

1. Binder数据传输与普通的数据传输差异

由于进程隔离，传统的数据跨进程传输如下



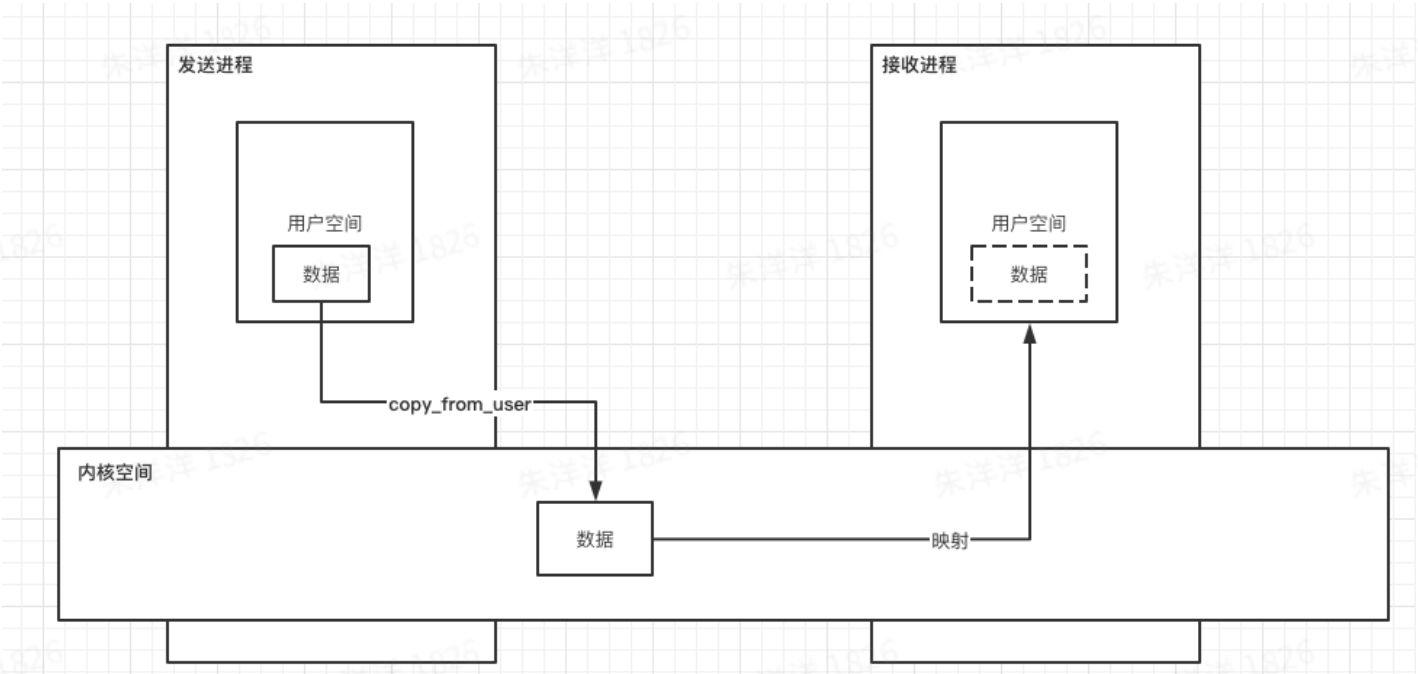
发送进程把自己想要传输的数据拷贝->内核空间缓存区域->内核空间数据拷贝到接收进程的空间

这里产生一个问题：

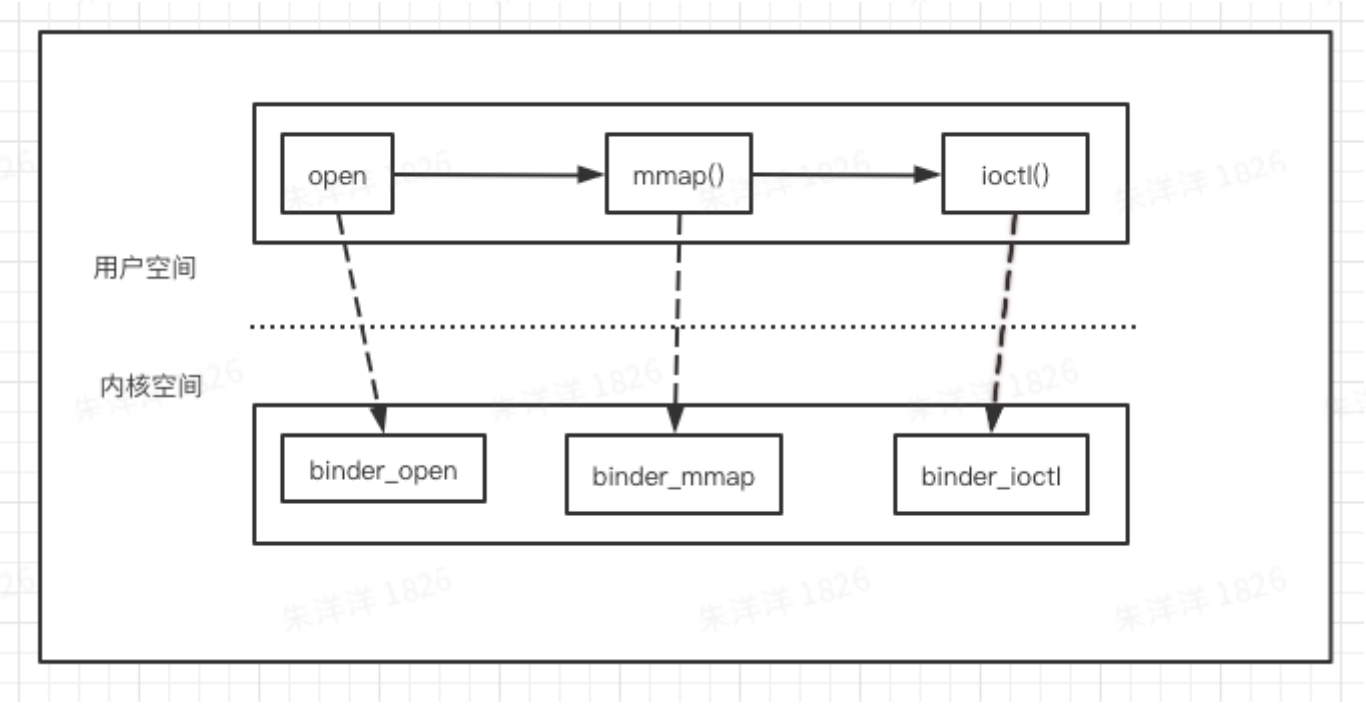
我直接传给内存空间数据地址不就行了，为什么要拷贝一次？

答：**因为每个进程产生是虚拟内存地址(c里面的指针)，必须要搭配到自己进程的页表才能够访问到物理内存，才能获取到数据。内核空间有自己的页表，无法使用别的进程直接传递过来的虚拟内存地址，同一个虚拟内存地址在内核和接收进程通过不同的页表，找到会是不同的物理内存地址。**

而Android中的binder使用了mmap机制，**解决了内核进程与用户态的产生的虚拟内存地址都指向了同一个物理内存地址**



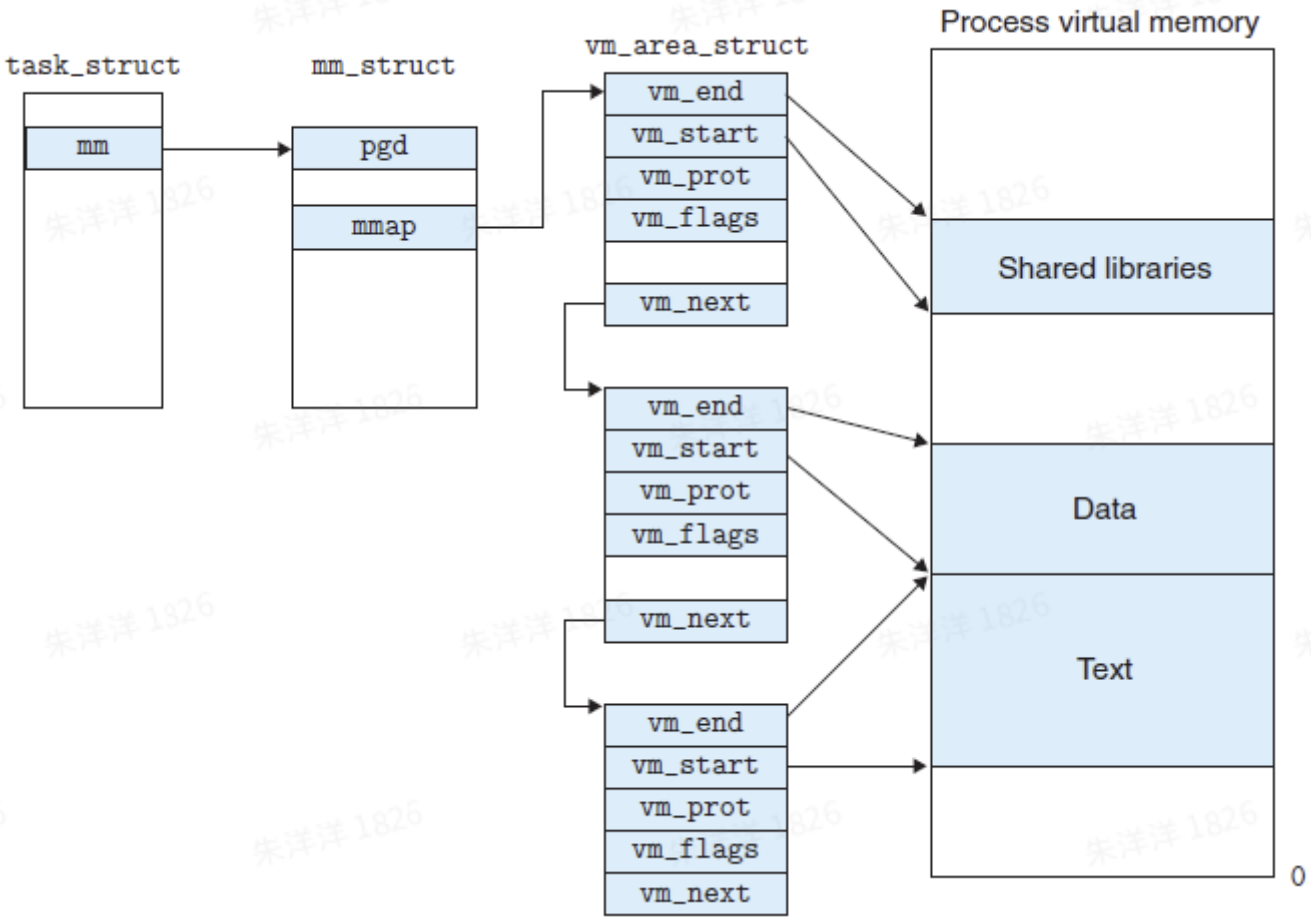
Android 手机启动流程：init进程-> zygote进程 -> ServiceManager进程 -> system_server进程 -> system_server启动AMS, WMS等framework代码 xxxxx启动的地方



需要使用Binder的进程，都是需要先通过binder_open打开Binder设备，然后通过binder_mmap进行内存的映射，再之后就是通过binder_ioctl进行数据传输等操作

2. 基本结构体的了解

linux内核使用vm_area_struct结构来表示一个独立的虚拟内存区域，由于每个不同的虚拟内存区域功能和内部机制都不同，因此一个进程使用多个vm_area_struct结构来分别表示不同类型的虚拟内存区域。各个vm_area_struct结构使用链表或者树形结构链接，方便进程快速访问，如下图所示：



vm_area_struct结构中包含区域起始和终止地址以及其他相关信息.

下面就是虚拟地址与物理地址绑定的逻辑

```
int handle_mm_fault(struct mm_struct *mm, struct vm_area_struct *vma,
                    unsigned long address, unsigned int flags)
 {
   pgd_t *pgd;  // 页全局目录项
   pud_t *pud;  // 页上级目录项
   pmd_t *pmd;  // 页中间目录项
   pte_t *pte;  // 页表项
   ...
   pgd = pgd_offset(mm, address);          // 获取虚拟内存地址对应的页全局目录项
   pud = pud_alloc(mm, pgd, address);      // 获取虚拟内存地址对应的页上级目录项
  ...
   pmd = pmd_alloc(mm, pud, address);      // 获取虚拟内存地址对应的页中间目录项
  ...
   pte = pte_alloc_map(mm, pmd, address); // 获取虚拟内存地址对应的页表项
  ...
   // 对页表项进行映射
   return handle_pte_fault(mm, vma, address, pte, pmd, flags);
}
```

3. binder_mmap函数调用 以Android R 最新代码为例

看代码之前了解一下 vm_area_struct 结构体

https://opengrok.pt.xiaomi.com/opengrok3/xref/miui-r-aquila-dev/kernel/msm-4.14/drivers/android/binder.c#5246

```
1  //vm_area_struc描述用户态空间虚拟地址的结构体
2  static int binder_mmap(struct file *filp, struct vm_area_struct *vma)
3  {
4          int ret;
5          //获取当前进程的binder_proc结构体，在前面binder_open的时候，已经将结构体保存在
   filp的private_data变量中
6          struct binder_proc *proc = filp->private_data;
7          const char *failure_string;
8
9          if (proc->tsk != current->group_leader)
10                 return -EINVAL;
11
12         //映射的内存不能大于4M
13         if ((vma->vm_end - vma->vm_start) > SZ_4M)
14                 vma->vm_end = vma->vm_start + SZ_4M;
15
16         binder_debug(BINDER_DEBUG_OPEN_CLOSE,
17                      "%s: %d %lx-%lx (%ld K) vma %lx pagep %lx\n",
18                      __func__, proc->pid, vma->vm_start, vma->vm_end,
19                      (vma->vm_end - vma->vm_start) / SZ_1K, vma->vm_flags,
20                      (unsigned long)pgprot_val(vma->vm_page_prot));
21
22         //检查用户空间地址是否有可写权限
23         if (vma->vm_flags & FORBIDDEN_MMAP_FLAGS) {
24                 ret = -EPERM;
25                 failure_string = "bad vm_flags";
26                 goto err_bad_arg;
27         }
28         vma->vm_flags |= VM_DONTCOPY | VM_MIXEDMAP;
29         vma->vm_flags &= ~VM_MAYWRITE;
30
31         vma->vm_ops = &binder_vm_ops;
32         vma->vm_private_data = proc;
33
34         //关键地方
35         ret = binder_alloc_mmap_handler(&proc->alloc, vma);
36         if (ret)
37                 return ret;
38         mutex_lock(&proc->files_lock);
```

```
        proc->files = get_files_struct(current);
        mutex_unlock(&proc->files_lock);
        return 0;

err_bad_arg:
        pr_err("%s: %d %lx-%lx %s failed %d\n", __func__,
               proc->pid, vma->vm_start, vma->vm_end, failure_string, ret);
        return ret;
}
```

https://opengrok.pt.xiaomi.com/opengrok3/xref/miui-r-aquila-dev/kernel/msm-4.14/drivers/android/binder_alloc.c#738

```
/**
 * binder_alloc_mmap_handler() - map virtual address space for proc
 * @alloc:        alloc structure for this proc
 * @vma:        vma passed to mmap()
 *
 * Called by binder_mmap() to initialize the space specified in
 * vma for allocating binder buffers
 *
 * Return:
 *      0 = success
 *      -EBUSY = address space already mapped
 *      -ENOMEM = failed to map memory to given address space
 */
int binder_alloc_mmap_handler(struct binder_alloc *alloc,
                              struct vm_area_struct *vma)
{
        int ret;
        const char *failure_string;
        struct binder_buffer *buffer;

        mutex_lock(&binder_alloc_mmap_lock);
        //判断是否已经映射了空间，避免重复映射
        if (alloc->buffer) {
                ret = -EBUSY;
                failure_string = "already mapped";
                goto err_already_mapped;
        }
        //将用户内存虚拟空间地址赋值给binder_proc的buffer变量？？？
        alloc->buffer = (void __user *)vma->vm_start;
        mutex_unlock(&binder_alloc_mmap_lock);

        ////用于存放内核分配的物理页的页描述指针
```

```
33          alloc->pages = kzalloc(sizeof(alloc->pages[0]) *
34                                     ((vma->vm_end - vma->vm_start) / PAGE_SIZE),
35                                 GFP_KERNEL);
36          if (alloc->pages == NULL) {
37                  ret = -ENOMEM;
38                  failure_string = "alloc page array";
39                  goto err_alloc_pages_failed;
40          }
41          //设置buffer_size  为用户进程空间虚拟地址内存大小
42          alloc->buffer_size = vma->vm_end - vma->vm_start;
43
44          //申请buffer的结构体
45          buffer = kzalloc(sizeof(*buffer), GFP_KERNEL);
46          if (!buffer) {
47                  ret = -ENOMEM;
48                  failure_string = "alloc buffer struct";
49                  goto err_alloc_buf_struct_failed;
50          }
51
52          //buffer中保存了用户空间的buffer指针
53          buffer->user_data = alloc->buffer;
54          //将该buffer指针放到 alloc->buffers的后面
55          list_add(&buffer->entry, &alloc->buffers);
56          buffer->free = 1;
57          //将分配好的buffer插入到对应的表中(空闲内存表中)
58          binder_insert_free_buffer(alloc, buffer);
59          alloc->free_async_space = alloc->buffer_size / 2;
60          //将用户空间地址信息保存到binder_proc中
61          binder_alloc_set_vma(alloc, vma);
62          mmgrab(alloc->vma_vm_mm);
63          return 0;
64
65  err_alloc_buf_struct_failed:
66          kfree(alloc->pages);
67          alloc->pages = NULL;
68  err_alloc_pages_failed:
69          mutex_lock(&binder_alloc_mmap_lock);
70          alloc->buffer = NULL;
71  err_already_mapped:
72          mutex_unlock(&binder_alloc_mmap_lock);
73          pr_err("%s: %d %lx-%lx %s failed %d\n", __func__,
74                 alloc->pid, vma->vm_start, vma->vm_end, failure_string, ret);
75          return ret;
76  }
```

```c
static int binder_update_page_range(struct binder_alloc *alloc, int allocate,
                                    void __user *start, void __user *end)
{
        void __user *page_addr;
        unsigned long user_page_addr;
        struct binder_lru_page *page;
        struct vm_area_struct *vma = NULL;
        struct mm_struct *mm = NULL;
        bool need_mm = false;

        binder_alloc_debug(BINDER_DEBUG_BUFFER_ALLOC,
                           "%d: %s pages %pK-%pK\n", alloc->pid,
                           allocate ? "allocate" : "free", start, end);

        if (end <= start)
                return 0;

        trace_binder_update_page_range(alloc, allocate, start, end);

        if (allocate == 0)
                goto free_range;

        //先尝试分配一page
        for (page_addr = start; page_addr < end; page_addr += PAGE_SIZE) {
                page = &alloc->pages[(page_addr - alloc->buffer) / PAGE_SIZE];
                if (!page->page_ptr) {
                        need_mm = true;
                        break;
                }
        }

        if (need_mm && mmget_not_zero(alloc->vma_vm_mm))
                mm = alloc->vma_vm_mm;

        if (mm) {
                down_read(&mm->mmap_sem);
                if (!mmget_still_valid(mm)) {
                        if (allocate == 0)
                                goto free_range;
                        goto err_no_vma;
                }
```

```
42                 vma = alloc->vma;
43         }
44
45         if (!vma && need_mm) {
46                 pr_err("%d: binder_alloc_buf failed to map pages in userspace, no
   vma\n",
47                         alloc->pid);
48                 goto err_no_vma;
49         }
50
51         for (page_addr = start; page_addr < end; page_addr += PAGE_SIZE) {
52                 int ret;
53                 bool on_lru;
54                 size_t index;
55
56                 index = (page_addr - alloc->buffer) / PAGE_SIZE;
57                 page = &alloc->pages[index];
58
59                 if (page->page_ptr) {
60                         trace_binder_alloc_lru_start(alloc, index);
61
62                         on_lru = list_lru_del(&binder_alloc_lru, &page->lru);
63                         WARN_ON(!on_lru);
64
65                         trace_binder_alloc_lru_end(alloc, index);
66                         continue;
67                 }
68
69                 if (WARN_ON(!vma))
70                         goto err_page_ptr_cleared;
71
72                 trace_binder_alloc_page_start(alloc, index);
73                 //分配一个物理页，并将该物理页的struct page指针值存放在proc-> page_ptr
   二维数组中
74                 page->page_ptr = alloc_page(GFP_KERNEL |
75                                             __GFP_HIGHMEM |
76                                             __GFP_ZERO);
77                 if (!page->page_ptr) {
78                         pr_err("%d: binder_alloc_buf failed for page at %pK\n",
79                                 alloc->pid, page_addr);
80                         goto err_alloc_page_failed;
81                 }
82                 page->alloc = alloc;
83                 INIT_LIST_HEAD(&page->lru);
```

```
84
85                  //由内核的虚拟地址得到用户空间的虚拟地址
86                  user_page_addr = (uintptr_t)page_addr;
87                  //为应用空间的这段虚拟地址建立虚拟到物理的映射
88                  ret = vm_insert_page(vma, user_page_addr, page[0].page_ptr);
89                  if (ret) {
90                          pr_err("%d: binder_alloc_buf failed to map page at %lx in
    userspace\n",
91                                 alloc->pid, user_page_addr);
92                          goto err_vm_insert_page_failed;
93                  }
94
95                  if (index + 1 > alloc->pages_high)
96                          alloc->pages_high = index + 1;
97
98                  trace_binder_alloc_page_end(alloc, index);
99                  /* vm_insert_page does not seem to increment the refcount */
100         }
101         if (mm) {
102                 up_read(&mm->mmap_sem);
103
104                 mmput(mm);
105         }
106         return 0;
107
108 free_range:
109         for (page_addr = end - PAGE_SIZE; 1; page_addr -= PAGE_SIZE) {
110                 bool ret;
111                 size_t index;
112
113                 index = (page_addr - alloc->buffer) / PAGE_SIZE;
114                 page = &alloc->pages[index];
115
116                 trace_binder_free_lru_start(alloc, index);
117
118                 ret = list_lru_add(&binder_alloc_lru, &page->lru);
119                 WARN_ON(!ret);
120
121                 trace_binder_free_lru_end(alloc, index);
122                 if (page_addr == start)
123                         break;
124                 continue;
125
126 err_vm_insert_page_failed:
```

```
                __free_page(page->page_ptr);
                page->page_ptr = NULL;
err_alloc_page_failed:
err_page_ptr_cleared:
                if (page_addr == start)
                        break;
        }
err_no_vma:
        if (mm) {
                up_read(&mm->mmap_sem);
                mmput(mm);
        }
        return vma ? -ENOMEM : -ESRCH;
}
```

https://opengrok.pt.xiaomi.com/opengrok3/xref/miui-r-aquila-dev/kernel/msm-4.14/mm/memory.c#1791

```
int vm_insert_page(struct vm_area_struct *vma, unsigned long addr,
                        struct page *page)
{
        if (addr < vma->vm_start || addr >= vma->vm_end)
                return -EFAULT;
        if (!page_count(page))
                return -EINVAL;
        if (!(vma->vm_flags & VM_MIXEDMAP)) {
                BUG_ON(down_read_trylock(&vma->vm_mm->mmap_sem));
                BUG_ON(vma->vm_flags & VM_PFNMAP);
                vma->vm_flags |= VM_MIXEDMAP;
        }
        return insert_page(vma, addr, page, vma->vm_page_prot);
}
```

```
static int insert_page(struct vm_area_struct *vma, unsigned long addr,
                        struct page *page, pgprot_t prot)
{
        struct mm_struct *mm = vma->vm_mm;
        int retval;
        pte_t *pte;
        spinlock_t *ptl;

        retval = -EINVAL;
        if (PageAnon(page))
```

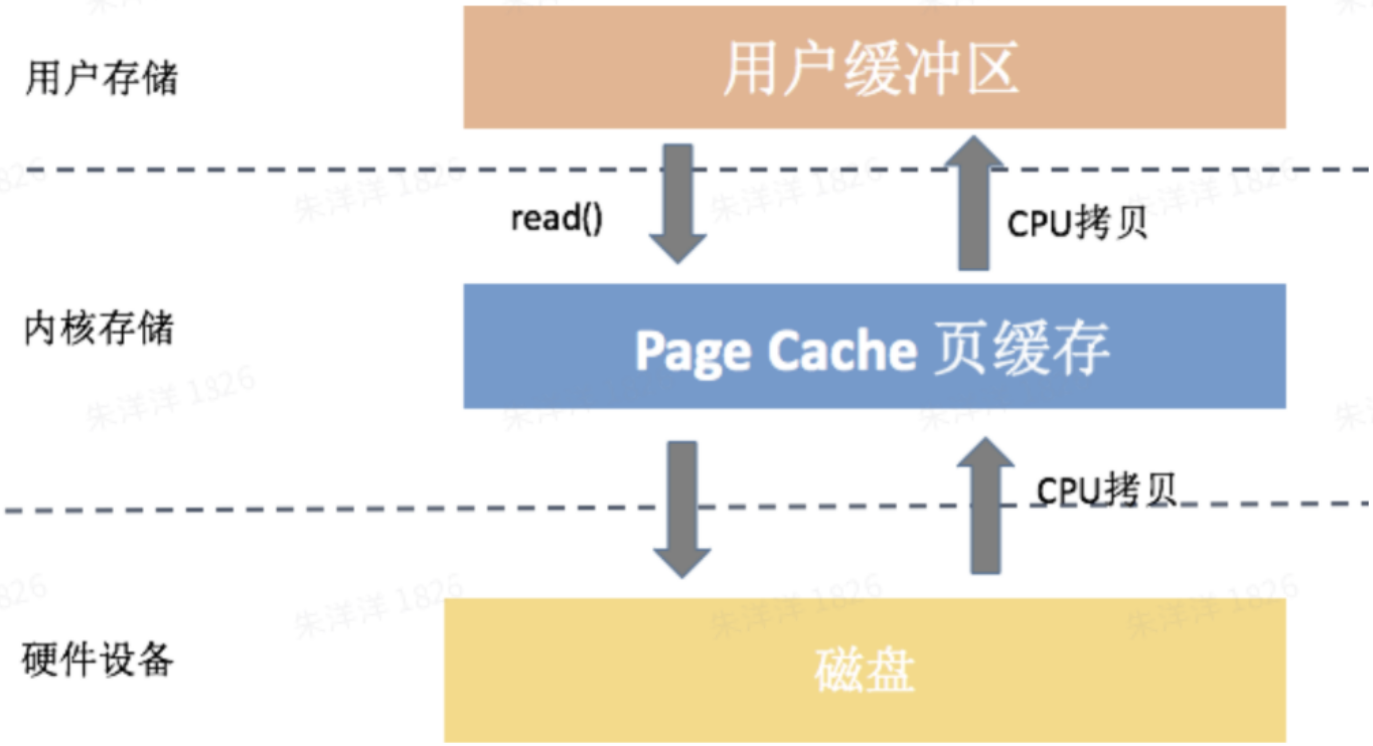
```
11                  goto out;
12          retval = -ENOMEM;
13          flush_dcache_page(page);
14          pte = get_locked_pte(mm, addr, &ptl);
15          if (!pte)
16                  goto out;
17          retval = -EBUSY;
18          if (!pte_none(*pte))
19                  goto out_unlock;
20
21          /* Ok, finally just insert the thing.. */
22          get_page(page);
23          inc_mm_counter_fast(mm, mm_counter_file(page));
24          page_add_file_rmap(page, false);
25          //修改页表
26          set_pte_at(mm, addr, pte, mk_pte(page, prot));
27
28          retval = 0;
29          pte_unmap_unlock(pte, ptl);
30          return retval;
31  out_unlock:
32          pte_unmap_unlock(pte, ptl);
33  out:
34          return retval;
35  }
```

三. mmap的使用途径以及优缺点

好处

- 减少系统调用。我们只需要一次 mmap() 系统调用，后续所有的调用像操作内存一样，而不会出现大量的 read/write 系统调用。
- 减少数据拷贝。普通的 read() 调用，数据需要经过两次拷贝；**而 mmap 只需要从磁盘拷贝一次就可以了**，并且由于做过内存映射，也不需要再拷贝回用户空间。
- 可靠性高。mmap 把数据写入页缓存后，跟缓存 I/O 的延迟写机制一样，可以依靠内核线程定期写回磁盘。**但是需要提的是，mmap 在内核崩溃、突然断电的情况下也一样有可能引起内容丢失，当然我们也可以使用 msync 来强制同步写**。

常见IO的缺点



写操作来说，应用程序也会将数据先写到页缓存中去，数据是否被立即写到磁盘上去取决于应用程序所采用写操作的机制。**默认系统采用的是延迟写机制，应用程序只需要将数据写到页缓存中去就可以**了，完全不需要等数据全部被写回到磁盘，系统会负责定期地将放在页缓存中的数据刷到磁盘上。

Page Cache 中被修改的内存称为“脏页”，内核通过 flush 线程定期将数据写入磁盘。具体写入的条件我们可以通过 /proc/sys/vm 文件或者 sysctl -a | grep vm 命令得到。

```
vm.dirty_writeback_centisecs = 500    // flush每隔5秒执行一次
vm.dirty_expire_centisecs = 200       //脏数据超时2秒触发一次flush到磁盘中
```

缺点：

内核空间的虚拟地址空间有限，大量使用导致内核空间的虚拟地址空间不够用(目前大部分的应用还没支持 64 位，除去内核使用的地址空间，一般我们可以使用的虚拟内存空间只有 3GB 左右)

四. mmap的实战

1. mmap原型函数介绍

https://opengrok.pt.xiaomi.com/opengrok3/xref/miui-r-umi-dev/bionic/libc/bionic/mmap.cpp

```
void *mmap(void *addr,size_t length,int prot,int flags,int fd,off_t offsize);
```

- 参数addr：指向欲映射的内存起始地址，通常设为 NULL，代表让系统自动选定地址，映射成功后返回该地址。
- 参数length：代表将文件中多大的部分映射到内存。
- 参数prot：映射区域的保护方式。可以为以下几种方式的组合：

`PROT_NONE` 无权限，基本没有用
`PROT_READ` 读权限
`PROT_WRITE` 写权限
`PROT_EXEC` 执行权限

- 参数flags: 描述了映射的类型。

`MAP_FIXED` 开启这个选项，则 addr 参数指定的地址是作为必须而不是建议。如果由于空间不足等问题无法映射则调用失败。不建议使用。
`MAP_PRIVATE` 表明这个映射不是共享的。文件使用 copy on write 机制映射，任何内存中的改动并不反映到文件之中。也不反映到其他映射了这个文件的进程之中。如果只需要读取某个文件而不改变文件内容，可以使用这种模式。
`MAP_SHARED` 和其他进程共享这个文件。往内存中写入相当于往文件中写入。会影响映射了这个文件的其他进程。与 MAP_PRIVATE冲突。

- 参数fd: 文件描述符。进行 map 之后，文件的引用计数会增加。因此，我们可以在 map 结束后关闭 fd，进程仍然可以访问它。当我们 unmap 或者结束进程，引用计数会减少。
- 参数offset: 文件偏移，从文件起始算起。

2. 项目工程代码：

https://git.n.xiaomi.com/zhuyangyang/mmap

五. 构建一套高性能高可用性的日志框架

这个目前我们使用系统的Slog的调用处
https://opengrok.pt.xiaomi.com/opengrok3/xref/miui-r-umi-dev/system/core/liblog/pmsg_writer.cpp#142
UNIX提供了另外两个函数—readv()和writev()，它们只需一次系统调用就可以实现在文件和进程的多个缓冲区之间传送数据，免除了多次系统调用或复制数据的开销。readv()称为散布读，即将文件中若干连续的数据块读入内存分散的缓冲区中。**writev()称为聚集写，即收集内存中分散的若干缓冲区中的数据写至文件的连续区域中。**

业界的主要应用：

1. MMKV 高性能key-value的存储架构，用于替代shareprefrefence https://github.com/Tencent/MMKV
2. 微信开源的mars 基础组件中的xlog https://github.com/Tencent/mars#mars_cn