

# 摩斯电码

操作系统内核

博客园

首页

新随笔

管理

## 内存映射函数remap\_pfn\_range学习——示例分析（1）

### 阅读目录(Content)

- [作者](#)
- [平台](#)
- [参考](#)
- [概述](#)
- [正文](#)
  - [一、驱动程序](#)
  - [二、用户测试程序](#)
  - [三、测试](#)

[回到顶部\(go to top\)](#)

作者

彭东林

QQ 405728433

搜索

找找看

谷歌搜索

我的标签

Android官网文档(192)

内核和驱动(165)

Android(151)

调试(56)

Qemu(48)

TINY4412(45)

[回到顶部\(go to top\)](#)

## 平台

Linux-4.10.17

Qemu-2.8 + vexpress-a9

DDR: 1GB

[回到顶部\(go to top\)](#)

## 参考

[Linux 虚拟内存和物理内存的理解](#)

[Linux进程分配内存的两种方式--brk\(\) 和mmap\(\)](#)

[Linux中的mmap的使用](#)

[程序（进程）内存分布 解析](#)

[回到顶部\(go to top\)](#)

## 概述

Linux内核提供了remap\_pfn\_range函数来实现将内核空间的内存映射到用户空间：



```
1 /**
2  * remap_pfn_range - remap kernel memory to
3  * userspace
4  * @vma: user vma to map to
5  * @addr: target user address to start at
6  * @pfn: physical address of kernel memory
7  * @size: size of map area
8  * @prot: page protection flags for this
9  * mapping
10  *
11  * Note: this is only safe if the mm
12  * semaphore is held when called.
13  */
14 int remap_pfn_range(struct vm_area_struct
15 *vma, unsigned long addr,
16 unsigned long pfn, unsigned long
17 size, pgprot_t prot);
```

TQ2440(34)

Git(28)

ARM架构和指令集(22)

Ubuntu(22)

更多



上面的注释对参数进行了说明。当用户调用mmap时，驱动中的file\_operations->mmap会被调用，可以在mmap中调用remap\_pfn\_range，它的大部分参数的值都由VMA提供。具体可以参考LDD3的P420.

[回到顶部\(go to top\)](#)

## 正文

下面结合一个简单的例子学习一下。在驱动中申请一个32个Page的缓冲区，这里的PAGE\_SIZE是4KB，所以内核中的缓冲区大小是128KB。user\_1和user\_2将前64KB映射到自己的用户空间，其中user\_1向缓冲区中写入字符串，user\_2去读取。user\_3和user\_4将后64KB映射到自己的用户空间，其中user\_3向缓冲区中写入字符串，user\_4读取字符串。user\_5将整个128KB映射到自己的用户空间，然后将缓冲区清零。此外，在驱动中申请缓冲区的方式有多种，可以用kmalloc、也可以用alloc\_pages，当然也可用vmalloc，下面会分别针对这三个接口实现驱动。

涉及到的测试程序和驱动程序可以到下面的链接下载：

[https://github.com/pengdonglin137/remap\\_pfn\\_demo](https://github.com/pengdonglin137/remap_pfn_demo)

## 一、驱动程序

下面先以kzalloc申请缓冲区的方式为例介绍，调用kmalloc申请32个页，我们知道kzalloc返回的虚拟地址的特点是对应的物理地址也是连续的，所以在调用remap\_pfn\_range的时候很方便。首先在驱动init的时候申请128KB的缓冲区：



```
1 static int __init remap_pfn_init(void)
2 {
3     int ret = 0;
4
5     kbuff = kzalloc(BUF_SIZE, GFP_KERNEL);
6     // 这里的BUF_SIZE是128KB
7     if (!kbuff) {
8         ret = -ENOMEM;
9         goto err;
10    }
11    ret = misc_register(&remap_pfn_misc);
12    // 注册一个misc设备
13    if (unlikely(ret)) {
14        pr_err("failed to register misc
15        device!\n");
16        goto err;
17    }
18    return 0;
19 err:
20    return ret;
21 }
```



第11行注册了一个misc设备，相关信息如下：



```
1 static struct miscdevice remap_pfn_misc = {
2     .minor = MISC_DYNAMIC_MINOR,
3     .name = "remap_pfn",
4     .fops = &remap_pfn_fops,
5 };
```



这样加载驱动后会在/dev下生成一个名为remap\_pfn的节点，用户程序可以通过这个节点跟驱动通信。其中remap\_pfn\_fops的定义如下：



```
1 static const struct file_operations
2 remap_pfn_fops = {
```

```
2     .owner = THIS_MODULE,  
3     .open = remap_pfn_open,  
4     .mmap = remap_pfn_mmap,  
5 };
```



第3行的open函数这里没有做什么实际的工作，只是打印一些log，比如将进程的内存布局信息输出

第4行，负责处理用户的mmap请求，这是需要关心的。

先看一下open函数具体打印了那些内容：



```
1 static int remap_pfn_open(struct inode  
*inode, struct file *file)  
2 {  
3     struct mm_struct *mm = current->mm;  
4  
5     printk("client: %s (%d)\n",  
current->comm, current->pid);  
6     printk("code section: [0x%lx  
0x%lx]\n", mm->start_code, mm->end_code);  
7     printk("data section: [0x%lx  
0x%lx]\n", mm->start_data, mm->end_data);  
8     printk("brk section: s: 0x%lx, c:  
0x%lx\n", mm->start_brk, mm->brk);  
9     printk("mmap section: s: 0x%lx\n",  
mm->mmap_base);  
10    printk("stack section: s: 0x%lx\n",  
mm->start_stack);  
11    printk("arg section: [0x%lx  
0x%lx]\n", mm->arg_start, mm->arg_end);  
12    printk("env section: [0x%lx  
0x%lx]\n", mm->env_start, mm->env_end);  
13  
14    return 0;  
15 }
```



第5行将进程的名字以及pid打印出来

第6行打印进程的代码段的范围

第7行打印进程的data段的范围，其中存放的是已初始化全局变量。而bss段存放的是未初始化全局变量，存放位置紧跟在data段后面，堆区之前

第8行打印进程的堆区的起始地址和当前地址  
第9行打印进程的mmap区的基地址，这里的mmap区是向下增长的。具体mmap区的基地址跟系统允许的当前进程的用户栈的大小有关，用户栈的最大size越大，mmap区的基地址就越小。修改用户栈的最大尺寸需要用到ulimit -s xxx命令，单位是KB，表示用户栈的最大尺寸，用户栈的尺寸可以上G，而内核栈却只有区区的2个页。  
第10行打印进程的用户栈的起始地址，向下增长  
第11行和第12行的暂不关心。

下面是remap\_pfn\_mmap的实现：



```
1 static int remap_pfn_mmap(struct file *file,
2 struct vm_area_struct *vma)
3 {
4     unsigned long offset = vma->vm_pgoff <<
PAGE_SHIFT;
5     unsigned long pfn_start =
(virt_to_phys(kbuff) >> PAGE_SHIFT) +
vma->vm_pgoff;
6     unsigned long virt_start = (unsigned
long)kbuff + offset;
7     unsigned long size = vma->vm_end -
vma->vm_start;
8     int ret = 0;
9     printk("phy: 0x%lx, offset: 0x%lx, size:
0x%lx\n", pfn_start << PAGE_SHIFT, offset,
size);
10
11     ret = remap_pfn_range(vma,
vma->vm_start, pfn_start, size,
vma->vm_page_prot);
12     if (ret)
13         printk("%s: remap_pfn_range failed
at [0x%lx 0x%lx]\n",
14             __func__, vma->vm_start,
vma->vm_end);
15     else
16         printk("%s: map 0x%lx to 0x%lx,
size: 0x%lx\n", __func__, virt_start,
17             vma->vm_start, size);
18
```

```
19     return ret;
20 }
```



第3行的vma\_pgoff表示的是该vma表示的区间在缓冲区中的偏移地址，单位是页。这个值是用户调用mmap时传入的最后一个参数，不过用户空间的offset的单位是字节（当然必须是页对齐），进入内核后，内核会将该值右移

PAGE\_SHIFT（12），也就是转换为以页为单位。因为要在第9行打印这个编译地址，所以这里将其再左移PAGE\_SHIFT，然后赋值给offset。

第4行计算内核缓冲区中将被映射到用户空间的地址对应的物理页帧号。virt\_to\_phys接受的虚拟地址必须在低端内存范围内，用于将虚拟地址转换为物理地址，而vmalloc返回的虚拟地址不在低端内存范围内，所以需要专门的函数。

第5行计算内核缓冲区中将被映射到用户空间的地址对应的虚拟地址

第6行计算该vma表示的内存区间的大小

第11行调用remap\_pfn\_range将物理页帧号pfn\_start对应的物理内存映射到用户空间的vm->vm\_start处，映射长度为该虚拟内存区的长度。由于这里的内核缓冲区是用kzalloc分配的，保证了物理地址的连续性，所以会将物理页帧号从pfn\_start开始的（size >> PAGE\_SHIFT）个连续的物理页帧依次按序映射到用户空间。

将驱动编译成模块后，insmod到内核。

## 二、用户测试程序

这里的五个测试程序都很简单，只是为了证明他们之间确实共享了同一块内存。

user\_1.c:



```
1 #define PAGE_SIZE (4*1024)
2 #define BUF_SIZE (16*PAGE_SIZE)
3 #define OFFSET (0)
4
5 int main(int argc, const char *argv[])
6 {
7     int fd;
8     char *addr = NULL;
9
10    fd = open("/dev/remap_pfn", O_RDWR);
11
12    addr = mmap(NULL, BUF_SIZE, PROT_READ |
13    PROT_WRITE, MAP_SHARED | MAP_LOCKED, fd,
14    OFFSET);
15
16    sprintf(addr, "I am %s\n", argv[0]);
17
18    while(1)
19        sleep(1);
20    return 0;
21 }
```



第10和第12行，打开设备节点，然后从内核空间映射64KB的内存到用户空间，首地址存放在addr中，由于后面既要写入也要共享，所以设置了对应的flags。这里指定的offset是0，即映射前64KB。

第14行输出字符串到addr指向的虚拟地址空间

user\_2.c:



```
1 #define PAGE_SIZE (4*1024)
2 #define BUF_SIZE (16*PAGE_SIZE)
3 #define OFFSET (0)
4
5 int main(int argc, const char *argv[])
6 {
7     int fd;
8     char *addr = NULL;
9
10    fd = open("/dev/remap_pfn", O_RDWR);
11
12    addr = mmap(NULL, BUF_SIZE, PROT_READ |
```



```
PROT_WRITE, MAP_SHARED | MAP_LOCKED, fd,
OFFSET);
13
14     printf("%s", addr);
15
16     while(1)
17         sleep(1);
18
19     return 0;
20 }
```



user\_2跟user\_1实现一般一样，不同之处是将addr指向的虚拟地址空间的内容打印出来。

user\_3.c:



```
1 #define PAGE_SIZE (4*1024)
2 #define BUF_SIZE (16*PAGE_SIZE)
3 #define OFFSET (16*PAGE_SIZE)
4
5 int main(int argc, const char *argv[])
6 {
7     int fd;
8     char *addr = NULL;
9
10    fd = open("/dev/remap_pfn", O_RDWR);
11
12    addr = mmap(NULL, BUF_SIZE, PROT_READ |
PROT_WRITE, MAP_SHARED | MAP_LOCKED, fd,
OFFSET);
13
14    sprintf(addr, "I am %s\n", argv[0]);
15
16    while(1)
17        sleep(1);
18    return 0;
19 }
```



第12行的OFFSET设置的是64KB，表示将内核缓冲区的后64KB映射到用户空间

第14行，向缓冲区中输入字符串

user\_4.c:



```
1 #define PAGE_SIZE (4*1024)
2 #define BUF_SIZE (16*PAGE_SIZE)
3 #define OFFSET (16*PAGE_SIZE)
4
5 int main(int argc, const char *argv[])
6 {
7     int fd;
8     char *addr = NULL;
9
10    fd = open("/dev/remap_pfn", O_RDWR);
11
12    addr = mmap(NULL, BUF_SIZE, PROT_READ |
13    PROT_WRITE, MAP_SHARED | MAP_LOCKED, fd,
14    OFFSET);
15
16    printf("%s", addr);
17
18    while(1)
19        sleep(1);
20    return 0;
21 }
```



第12行的OFFSET设置的是64KB，表示将内核缓冲区的后64KB映射到用户空间

第14行，输出缓冲区中内容

user\_5.c:



```
1 #define PAGE_SIZE (4*1024)
2 #define BUF_SIZE (32*PAGE_SIZE)
3 #define OFFSET (0)
4
5 int main(int argc, const char *argv[])
6 {
7     int fd;
8     char *addr = NULL;
9     int *brk;
10
11    fd = open("/dev/remap_pfn", O_RDWR);
12
```

```
13     addr = mmap(NULL, BUF_SIZE, PROT_READ |  
PROT_WRITE, MAP_SHARED | MAP_LOCKED, fd, 0);  
14     memset(addr, 0x0, BUF_SIZE);  
15  
16     printf("Clear Finished\n");  
17  
18     while(1)  
19         sleep(1);  
20     return 0;  
21 }
```



第13行，将内核缓冲区的整个128KB都映射到用户空间

第14行，清除缓冲区中内容

### 三、测试

#### 1、内核空间的虚拟内存布局

在内核的启动log里可以看到内核空间的虚拟内存布局信息：



```
1 [ 0.000000] Virtual kernel memory layout:  
2 [ 0.000000]     vector  : 0xffff0000 -  
0xffff1000    ( 4 kB)  
3 [ 0.000000]     fixmap  : 0xffc00000 -  
0xfff00000    (3072 kB)  
4 [ 0.000000]     vmalloc : 0xf0800000 -  
0xff800000    ( 240 MB)  
5 [ 0.000000]     lowmem   : 0xc0000000 -  
0xf0000000    ( 768 MB)  
6 [ 0.000000]     pkmap    : 0xbfe00000 -  
0xc0000000    ( 2 MB)  
7 [ 0.000000]     modules : 0xbf000000 -  
0xbfe00000    ( 14 MB)  
8 [ 0.000000]       .text : 0xc0008000 -  
0xc0800000    (8160 kB)  
9 [ 0.000000]       .init : 0xc0b00000 -  
0xc0c00000    (1024 kB)  
10 [ 0.000000]       .data : 0xc0c00000 -  
0xc0c7696c    ( 475 kB)  
11 [ 0.000000]       .bss : 0xc0c78000 -  
0xc0cc9b8c    ( 327 kB)
```

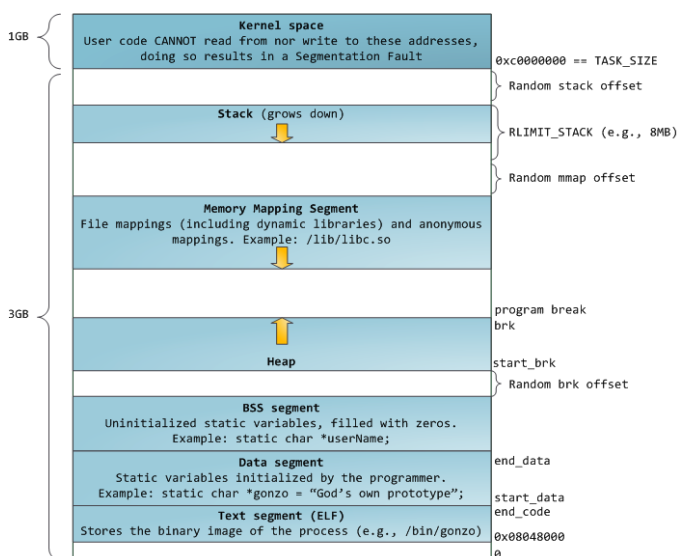


用kzalloc分配的内存会落在第5行表示的虚拟内存范围内

用vmalloc分配的内存会落在第4行表示的虚拟内存范围内

## 2、用户虚拟地址空间的布局

下面是Linux系统下用户的虚拟内存布局大致信息：



这里需要注意的是：

当调用malloc分配内存的时候，如果传给malloc的参数小于128KB时，系统会在heap区分配内存，分配的方式是向高地址调整brk指针的位置。当传给malloc的参数大于128KB时，系统会在mmap区分配，即分配一块新的vma，其中可能会涉及到vma的合并扩展等操作。

可以参考：[Linux进程分配内存的两种方式](#)  
[--brk\(\)](#) 和 [mmap\(\)](#)

## 3、user\_1和user\_2

运行user1：

```
[root@vexpress mnt]# ./user_1
```

可以看到如下内核log:



```
1 [ 2494.835749] client: user_1 (870)
2 [ 2494.835918] code section: [0x8000
0x87f4]
3 [ 2494.836047] data section: [0x107f4
0x1092c]
4 [ 2494.836165] brk section: s: 0x11000, c:
0x11000
5 [ 2494.836307] mmap section: s: 0xb6f17000
6 [ 2494.836441] stack section: s: 0xbe909e20
7 [ 2494.836569] arg section: [0xbe909f23
0xbe909f2c]
8 [ 2494.836689] env section: [0xbe909f2c
0xbe909ff3]
9 [ 2494.836943] phy: 0x8eb60000, offset: 0x0,
size: 0x10000
10 [ 2494.837176] remap_pfn_mmap: map
0xeeb60000 to 0xb6d75000, size: 0x10000
```



进程号是870，可以分别用下面的查看一下该进程的地址空间的map信息:



```
1 [root@vexpress mnt]# cat /proc/870/maps
2 00008000-00009000 r-xp 00000000 00:12
1179664 /mnt/user_1
3 00010000-00011000 rw-p 00000000 00:12
1179664 /mnt/user_1
4 b6d75000-b6d85000 rw-s 00000000 00:10 8765
/dev/remap_pfn
5 b6d85000-b6eb8000 r-xp 00000000 b3:01 143
/lib/libc-2.18.so
6 b6eb8000-b6ebf000 ---p 00133000 b3:01 143
/lib/libc-2.18.so
7 b6ebf000-b6ec1000 r--p 00132000 b3:01 143
/lib/libc-2.18.so
8 b6ec1000-b6ec2000 rw-p 00134000 b3:01 143
/lib/libc-2.18.so
9 b6ec2000-b6ec5000 rw-p 00000000 00:00 0
```

```

10 b6ec5000-b6ee6000 r-xp 00000000 b3:01 188
/lib/libgcc_s.so.1
11 b6ee6000-b6eed000 ---p 00021000 b3:01 188
/lib/libgcc_s.so.1
12 b6eed000-b6eee000 rw-p 00020000 b3:01 188
/lib/libgcc_s.so.1
13 b6eee000-b6f0e000 r-xp 00000000 b3:01 165
/lib/ld-2.18.so
14 b6f13000-b6f15000 rw-p 00000000 00:00 0
15 b6f15000-b6f16000 r--p 0001f000 b3:01 165
/lib/ld-2.18.so
16 b6f16000-b6f17000 rw-p 00020000 b3:01 165
/lib/ld-2.18.so
17 be8e9000-be90a000 rw-p 00000000 00:00 0
[stack]
18 bed1c000-bed1d000 r-xp 00000000 00:00 0
[sigpage]
19 bed1d000-bed1e000 r--p 00000000 00:00 0
[vvar]
20 bed1e000-bed1f000 r-xp 00000000 00:00 0
[vdso]
21 ffff0000-ffff1000 r-xp 00000000 00:00 0
[vectors]

```



上面的每一行都可以表示一个vma的映射信息，其中第4行是需要关心的：

```

1 b6d75000-b6d85000 rw-s 00000000 00:10 8765
/dev/remap_pfn

```

含义：

"b6d75000"是vma->vm\_start的值，"b6d85000"是vma->vm\_end的值，b6d85000减b6d75000是64KB，即给vma表示的虚拟内存区域的大小。

"rw-s"表示的是vma->vm\_flags，其中's'表示share，'p'表示private

"00000000"表示偏移量，也就是vma->vm\_pgoff的值

"00:10"表示该设备节点的主次设备号

"8765"表示该设备节点的inode值

"/dev/remap\_pfn"表示设备节点的名字。

也可以用pmap查看该进程的虚拟地址空间映射信息：



```
1 [root@vexpress mnt]# pmap -x 870
2 870: {no such process} ./user_1
3 Address      Kbytes      PSS    Dirty    Swap
Mode Mapping
4 00008000      4         4        0        0
r-xp  /mnt/user_1
5 00010000      4         4        4        0
rw-p  /mnt/user_1
6 b6d75000     64         0        0        0
rw-s  /dev/remap_pfn
7 b6d85000    1228       424        0        0
r-xp  /lib/libc-2.18.so
8 b6eb8000     28         0        0        0
---p  /lib/libc-2.18.so
9 b6ebf000      8         8        8        0
r--p  /lib/libc-2.18.so
10 b6ec1000      4         4        4        0
rw-p  /lib/libc-2.18.so
11 b6ec2000     12         8        8        0
rw-p  [ anon ]
12 b6ec5000    132        64        0        0
r-xp  /lib/libgcc_s.so.1
13 b6ee6000     28         0        0        0
---p  /lib/libgcc_s.so.1
14 b6eed000      4         4        4        0
rw-p  /lib/libgcc_s.so.1
15 b6eee000    128       122        0        0
r-xp  /lib/ld-2.18.so
16 b6f13000      8         8        8        0
rw-p  [ anon ]
17 b6f15000      4         4        4        0
r--p  /lib/ld-2.18.so
18 b6f16000      4         4        4        0
rw-p  /lib/ld-2.18.so
19 be8e9000    132         4        4        0
rw-p  [stack]
20 bed1c000      4         0        0        0
r-xp  [sigpage]
21 bed1d000      4         0        0        0
r--p  [vvar]
22 bed1e000      4         0        0        0
r-xp  [vdso]
23 ffff0000      4         0        0        0
r-xp  [vectors]
```


```
24 -----
25 total          1808      662      48      0
```



然后运行user\_2:

```
1 [root@vexpress mnt]# ./user_2
2 I am ./user_1
```

可以看到user\_1写入的信息，下面是内核log以及虚拟地址空间映射信息：



```
1 [ 2545.832903] client: user_2 (873)
2 [ 2545.833087] code  section: [0x8000
0x87e0]
3 [ 2545.833178] data  section: [0x107e0
0x10918]
4 [ 2545.833262] brk   section: s: 0x11000, c:
0x11000
5 [ 2545.833346] mmap  section: s: 0xb6fb5000
6 [ 2545.833423] stack section: s: 0xbea0ee20
7 [ 2545.833499] arg   section: [0xbea0ef23
0xbea0ef2c]
8 [ 2545.833590] env   section: [0xbea0ef2c
0xbea0eff3]
9 [ 2545.833761] phy: 0x8eb60000, offset: 0x0,
size: 0x10000
10 [ 2545.833900] remap_pfn_mmap: map
0xeeb60000 to 0xb6e13000, size: 0x10000
11
12 [root@vexpress mnt]# cat /proc/873/maps
13 00008000-00009000 r-xp 00000000 00:12
1179665      /mnt/user_2
14 00010000-00011000 rw-p 00000000 00:12
1179665      /mnt/user_2
15 b6e13000-b6e23000 rw-s 00000000 00:10 8765
/dev/remap_pfn
16 b6e23000-b6f56000 r-xp 00000000 b3:01 143
/lib/libc-2.18.so
17 b6f56000-b6f5d000 ---p 00133000 b3:01 143
/lib/libc-2.18.so
18 b6f5d000-b6f5f000 r--p 00132000 b3:01 143
/lib/libc-2.18.so
19 b6f5f000-b6f60000 rw-p 00134000 b3:01 143
```



```

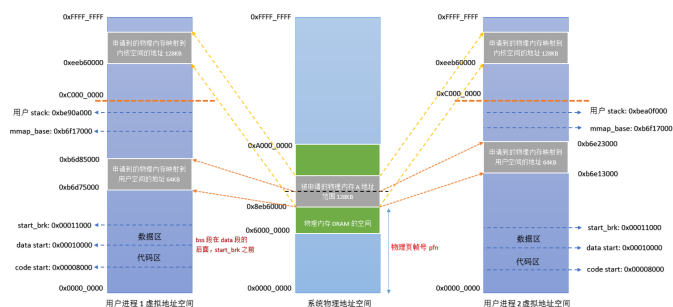
/lib/libc-2.18.so
20 b6f60000-b6f63000 rw-p 00000000 00:00 0
21 b6f63000-b6f64000 r-xp 00000000 b3:01 188
/lib/libgcc_s.so.1
22 b6f84000-b6f8b000 ---p 00021000 b3:01 188
/lib/libgcc_s.so.1
23 b6f8b000-b6f8c000 rw-p 00020000 b3:01 188
/lib/libgcc_s.so.1
24 b6f8c000-b6fac000 r-xp 00000000 b3:01 165
/lib/ld-2.18.so
25 b6fb0000-b6fb3000 rw-p 00000000 00:00 0
26 b6fb3000-b6fb4000 r--p 0001f000 b3:01 165
/lib/ld-2.18.so
27 b6fb4000-b6fb5000 rw-p 00020000 b3:01 165
/lib/ld-2.18.so
28 be9ee000-bea0f000 rw-p 00000000 00:00 0
[stack]
29 beedf000-beee0000 r-xp 00000000 00:00 0
[sigpage]
30 beee0000-beee1000 r--p 00000000 00:00 0
[vvar]
31 beee1000-beee2000 r-xp 00000000 00:00 0
[vdso]
32 ffff0000-ffff1000 r-xp 00000000 00:00 0
[vectors]

```



上面的log信息可以查看：[https://github.com/pengdonglin137/remap\\_pfn\\_demo/blob/master/log/user2](https://github.com/pengdonglin137/remap_pfn_demo/blob/master/log/user2)

根据上面的空间映射信息可以得到下面的示意图：



#### 4、user\_3和user\_4

相关的log信息可以查看：

<https://github.com/pengdonglin137>

[/remap\\_pfn\\_demo/blob/master/log/user3](https://github.com/pengdonglin137/remap_pfn_demo/blob/master/log/user3)

<https://github.com/pengdonglin137>

[/remap\\_pfn\\_demo/blob/master/log/user4](https://github.com/pengdonglin137/remap_pfn_demo/blob/master/log/user4)

下面是运行user3的log和映射信息：



```
1 [ 4938.000918] client: user_3 (884)
2 [ 4938.001117] code section: [0x8000
0x87f4]
3 [ 4938.001205] data section: [0x107f4
0x1092c]
4 [ 4938.001281] brk section: s: 0x11000, c:
0x11000
5 [ 4938.001410] mmap section: s: 0xb6ff1000
6 [ 4938.001485] stack section: s: 0xbea10e20
7 [ 4938.001549] arg section: [0xbea10f23
0xbea10f2c]
8 [ 4938.001606] env section: [0xbea10f2c
0xbea10ff3]
9 [ 4938.001793] phy: 0x8eb70000, offset:
0x10000, size: 0x10000
10 [ 4938.001996] remap_pfn_mmap: map
0xeeb70000 to 0xb6e4f000, size: 0x10000
11
12 [root@vexpress mnt]#
13 [root@vexpress mnt]# cat /proc/884/maps
14 00008000-00009000 r-xp 00000000 00:12
1179666 /mnt/user_3
15 00010000-00011000 rw-p 00000000 00:12
1179666 /mnt/user_3
16 b6e4f000-b6e5f000 rw-s 00010000 00:10 8765
/dev/remap_pfn
17 b6e5f000-b6f92000 r-xp 00000000 b3:01 143
/lib/libc-2.18.so
18 b6f92000-b6f99000 ---p 00133000 b3:01 143
/lib/libc-2.18.so
19 b6f99000-b6f9b000 r--p 00132000 b3:01 143
/lib/libc-2.18.so
20 b6f9b000-b6f9c000 rw-p 00134000 b3:01 143
/lib/libc-2.18.so
21 b6f9c000-b6f9f000 rw-p 00000000 00:00 0
22 b6f9f000-b6fc0000 r-xp 00000000 b3:01 188
/lib/libgcc_s.so.1
```

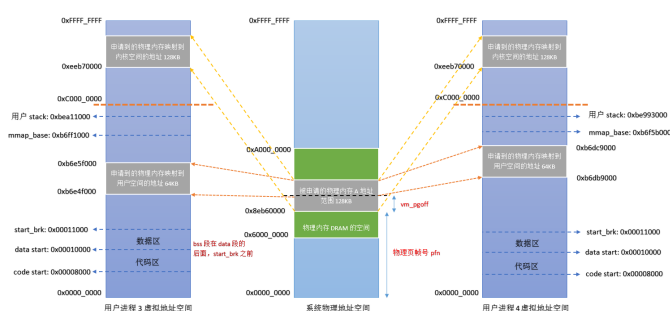
```

23 b6fc0000-b6fc7000 ---p 00021000 b3:01 188
/lib/libgcc_s.so.1
24 b6fc7000-b6fc8000 rw-p 00020000 b3:01 188
/lib/libgcc_s.so.1
25 b6fc8000-b6fe8000 r-xp 00000000 b3:01 165
/lib/ld-2.18.so
26 b6fed000-b6fef000 rw-p 00000000 00:00 0
27 b6fef000-b6ff0000 r--p 0001f000 b3:01 165
/lib/ld-2.18.so
28 b6ff0000-b6ff1000 rw-p 00020000 b3:01 165
/lib/ld-2.18.so
29 be9f0000-bea11000 rw-p 00000000 00:00 0
[stack]
30 bebe9000-bebea000 r-xp 00000000 00:00 0
[sigpage]
31 bebea000-bebeb000 r--p 00000000 00:00 0
[vvar]
32 bebeb000-bebec000 r-xp 00000000 00:00 0
[vdso]
33 ffff0000-ffff1000 r-xp 00000000 00:00 0
[vectors]
```



需要关注的是第16行，其中的"00010000"表示 offset，大小是64KB，也就是vma->vm\_pgoff的值。

下面是user 3和user 4的共享内存的示意图:



## 5、 user\_5

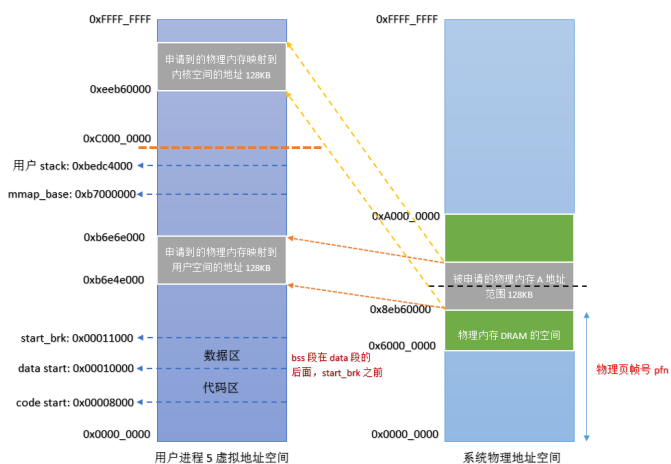
user\_5负责将128KB的内核缓冲区映射到自己的用户空间，并清除其中的内容。

log信息可以查看：<https://github.com>

/pengdonglin137/remap\_pfn\_demo/blob/master

```
/log/user5
```

下面是映射示意图:



未完待续...

标签: 内核和驱动, 内存管理

好文要顶

关注我

收藏该文



摩斯电码

关注 - 38

粉丝 - 242

+加关注

1

0

« 上一篇: 在busybox里使用ulimit命令

» 下一篇: 内存映射函数remap\_pfn\_range学习——示例分析 (2)

posted @ 2017-12-30 14:56 摩斯电码 阅读(13270) 评论(1) 编辑 收藏 举报

刷新评论 刷新页面 返回顶部

登录后才能查看或发表评论, 立即 [登录](#) 或者 [逛逛](#) 博客园首页

【活动】博客园&顺顺智慧联合沙龙：研发进度太慢，开发人员如何破局？

【推荐】阿里云云大使特惠：新用户购ECS服务器1核2G最低价87元/年

【推荐】大型组态、工控、仿真、CAD\GIS 50万行VC++源码免费下载!

【推荐】百度智能云超值优惠：新用户首购云服务器1核1G低至69元/年

【推荐】和开发者在一起：华为开发者社区，入驻博客园科技品牌专区

【推广】园子与爱卡汽车爱宝险合作，随手就可以买一份的百万医疗保险



#### 编辑推荐：

- 浅谈 C# 更改令牌 ChangeToken
- CNN卷积神经网络详解
- 记一次 .NET 某流媒体独角兽 API 句柄泄漏分析
- 流量录制与回放技术实践
- 熟悉而陌生的新朋友——IAsyncDisposable

#### 最新新闻：

- 旷视科技更新招股书：上半年营收6.7亿，新增5.0 on Kubernetes上会 (2021-09-03 23:34)
- 突发！中芯国际董事长辞职，去年年薪近700万

Copyright © 2021 摩斯电码

Powered by .NET 5.0 on Kubernetes

元 (2021-09-03 23:28)

· 雷军卖出3亿股小米股票？真相是：这是他此前捐赠的股票 (2021-09-03 23:26)

· 我们的互联网，正在变成一台「上瘾机器」 (2021-09-03 23:20)

· 或将成为国内首款3A的《黑神话：悟空》，距离独占有多近？ (2021-09-03 18:42)

» 更多新闻...