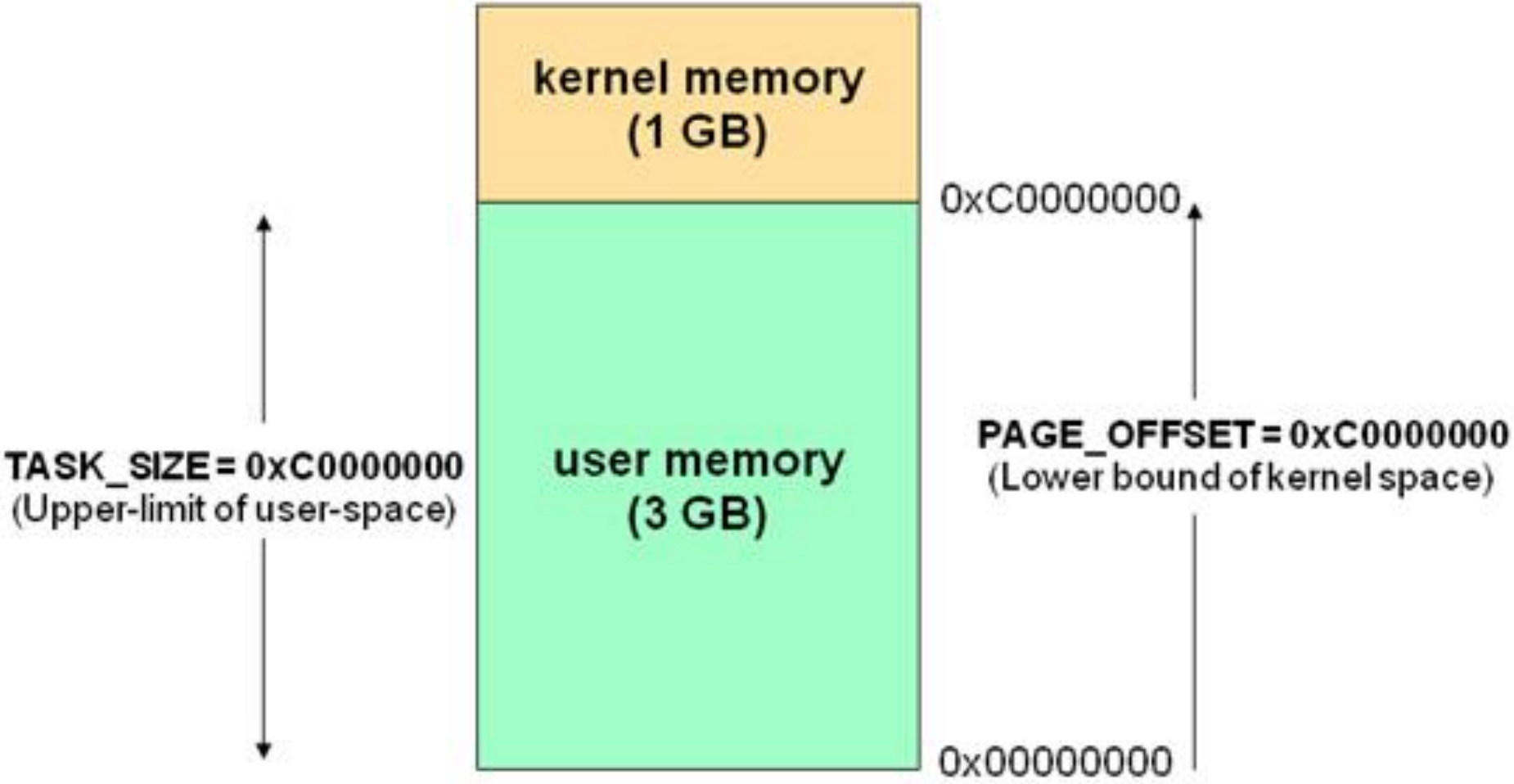


现代采用虚拟内存的操作系统通常都使用平坦地址空间，对于32位的操作系统而言，每个进程的虚拟地址空间都是0x00000000~0xC0000000,合计3G大小。

进程的3G的虚拟地址空间只有映射为物理地址空间，才能够被使用!

进程如何管理和分配它的3G的虚拟地址空间呢?

分治思想

按照不同的访问属性和功能划分为不同的内存区域，我们也叫虚拟内存区域（VMA）

hello world

- 代码段：可执行文件的内存映射
- 数据段：可执行文件的已初始化全局变量和静态局部变量的内存映射；
- bss段：未初始化的或者值为0的变量的内存映射；
- lib库的代码段；（多个）
- lib库的数据段；（多个）
- lib库的bss段；（多个）
- 任何内存映射文件（有名mmap建立）；
- 任何共享内存段（匿名mmap建立）；
- 进程栈stack；
- 进程堆heap；

```c
#include <stdio.h>
#include <unistd.h>>
void main()
{
        printf( "PID = %d\n", getpid());
        while(1) {
                sleep(2);
        }
        return;
}
```

```
root@ubuntu:/home/jinxin/linux-4.9.229# cat /proc/10738/maps
00400000-00401000 r-xp 00000000 08:01 938729                             /home/jinxin/app/main
00600000-00601000 r--p 00000000 08:01 938729                             /home/jinxin/app/main
00601000-00602000 rw-p 00001000 08:01 938729                             /home/jinxin/app/main
7f939873e000-7f93988fc000 r-xp 00000000 08:01 1462783                    /lib/x86_64-linux-gnu/libc-2.19.so
7f93988fc000-7f9398afc000 ---p 001be000 08:01 1462783                    /lib/x86_64-linux-gnu/libc-2.19.so
7f9398afc000-7f9398b00000 r--p 001be000 08:01 1462783                    /lib/x86_64-linux-gnu/libc-2.19.so
7f9398b00000-7f9398b02000 rw-p 001c2000 08:01 1462783                    /lib/x86_64-linux-gnu/libc-2.19.so
7f9398b02000-7f9398b07000 rw-p 00000000 00:00 0
7f9398b07000-7f9398b2a000 r-xp 00000000 08:01 1462780                    /lib/x86_64-linux-gnu/ld-2.19.so
7f9398d0e000-7f9398d11000 rw-p 00000000 00:00 0
7f9398d28000-7f9398d29000 rw-p 00000000 00:00 0
7f9398d29000-7f9398d2a000 r--p 00022000 08:01 1462780                    /lib/x86_64-linux-gnu/ld-2.19.so
7f9398d2a000-7f9398d2b000 rw-p 00023000 08:01 1462780                    /lib/x86_64-linux-gnu/ld-2.19.so
7f9398d2b000-7f9398d2c000 rw-p 00000000 00:00 0
7ffcc4ded000-7ffcc4e0e000 rw-p 00000000 00:00 0                          [stack]
7ffcc4f01000-7ffcc4f03000 r--p 00000000 00:00 0                          [vvar]
7ffcc4f03000-7ffcc4f05000 r-xp 00000000 00:00 0                          [vdso]
ffffffffff600000-ffffffffff601000 r-xp 00000000 00:00 0                  [vsyscall]
```

内核每进程的vm_area_struct项	/proc/pid/maps中的项	含义
vm_start	“-”前一列，如00377000	此段虚拟地址空间起始地址
vm_end	“-”后一列，如00390000	此段虚拟地址空间结束地址
vm_flags	第三列，如r-xp	此段虚拟地址空间的属性。每种属性用一个字段表示，r表示可读，w表示可写，x表示可执行，p和s共用一个字段，互斥关系，p表示私有段，s表示共享段，如果没有相应权限，则用'-'代替
vm_pgoff	第四列，如00000000	对有名映射，表示此段虚拟内存起始地址在文件中以页为单位的偏移。对匿名映射，它等于0或者vm_start/PAGE_SIZE
vm_file->f_dentry->d_inode->i_sb->s_dev	第五列，如fd:00	映射文件所属设备号。对匿名映射来说，因为没有文件在磁盘上，所以没有设备号，始终为00:00。对有名映射来说，是映射的文件所在设备的设备号
vm_file->f_dentry->d_inode->i_ino	第六列，如9176473	映射文件所属节点号。对匿名映射来说，因为没有文件在磁盘上，所以没有节点号，始终为00:00。对有名映射来说，是映射的文件的节点号
	第七列，如/lib/ld-2.5.so	对有名来说，是映射的文件名。对匿名映射来说，是此段虚拟内存在进程中的角色。[stack]表示在进程中作为栈使用，[heap]表示堆。其余情况则无显示

```c
#include <stdio.h>
#include <unistd.h>>
void main()
{
        char *buffer = NULL;
        buffer = (char *)malloc(2 * sizeof(char));
        printf( "PID = %d\n", getpid());
        while(1) {
                sleep(2);
        }
```

```
root@ubuntu:/home/jinxin/linux-4.9.229# cat /proc/11356/maps
00400000-00401000 r-xp 00000000 08:01 938729                             /home/jinxin/app/main
00600000-00601000 r--p 00000000 08:01 938729                             /home/jinxin/app/main
00601000-00602000 rw-p 00001000 08:01 938729                             /home/jinxin/app/main
01cdd000-01cfe000 rw-p 00000000 00:00 0                                  [heap]
7fcc1261e000-7fcc127dc000 r-xp 00000000 08:01 1462783                    /lib/x86_64-linux-gnu/libc-2.19.so
7fcc127dc000-7fcc129dc000 ---p 001be000 08:01 1462783                    /lib/x86_64-linux-gnu/libc-2.19.so
7fcc129dc000-7fcc129e0000 r--p 001be000 08:01 1462783                    /lib/x86_64-linux-gnu/libc-2.19.so
7fcc129e0000-7fcc129e2000 rw-p 001c2000 08:01 1462783                    /lib/x86_64-linux-gnu/libc-2.19.so
7fcc129e2000-7fcc129e7000 rw-p 00000000 00:00 0
7fcc129e7000-7fcc12a0a000 r-xp 00000000 08:01 1462780                    /lib/x86_64-linux-gnu/ld-2.19.so
7fcc12bee000-7fcc12bf1000 rw-p 00000000 00:00 0
7fcc12c08000-7fcc12c09000 rw-p 00000000 00:00 0
7fcc12c09000-7fcc12c0a000 r--p 00022000 08:01 1462780                    /lib/x86_64-linux-gnu/ld-2.19.so
7fcc12c0a000-7fcc12c0b000 rw-p 00023000 08:01 1462780                    /lib/x86_64-linux-gnu/ld-2.19.so
7fcc12c0b000-7fcc12c0c000 rw-p 00000000 00:00 0
7ffdf8474000-7ffdf8495000 rw-p 00000000 00:00 0                          [stack]
7ffdf854e000-7ffdf8550000 r--p 00000000 00:00 0                          [vvar]
7ffdf8550000-7ffdf8552000 r-xp 00000000 00:00 0                          [vdso]
ffffffffff600000-ffffffffff601000 r-xp 00000000 00:00 0                  [vsyscall]
```

进程的虚拟地址空间

内核使用mm_struct来描述一个进程的虚拟地址空间，用vm_area_struct来描述一个虚拟内存区域（VMA），进程的虚拟地址空间由多个VMA组成。

```
struct mm_struct {
        struct vm_area_struct * mmap;           /* 指向虚拟内存区域（VMA）链表 */
        struct rb_root mm_rb;                   /* 指向red_black树*/
        struct vm_area_struct * mmap_cache;     /* 指向最近找到的虚拟内存区域*/
        pgd_t * pgd;                            /* 指向该进程的页目录表*/
        atomic_t mm_users;                      /* 用户空间中的有多少用户*/
        atomic_t mm_count;                      /* 对"struct mm_struct"有多少引用*/
        int map_count;                          /* 虚拟内存区域的个数*/
        struct list_head mmlist;                /* 所有活动（active）mm的链表 */
        unsigned long start_code, end_code, start_data, end_data;
        unsigned long start_brk, brk, start_stack;//堆的首地址、堆的尾地址、栈的首地址
        unsigned long arg_start, arg_end;       // 命令行参数首地址、尾地址
        unsigned long env_start, env_end;       // 环境变量的首地址、尾地址
};
```

虚拟内存区域

struct vm_area_struct：
用来描述一个虚拟内存区域（VMA）。
内核将每个内存区域作为一个单独的内存对象管理，每个内存区域都有一致的属性，比如权限等。
所以我们程序的代码段、数据段和bss段在内核里都分别有一个struct vm_area_struct 结构体来描述

```
struct vm_area_struct {
        struct mm_struct * vm_mm;    /* 虚拟内存区域所在的虚拟地址空间*/
        unsigned long vm_start;      /* 在进程虚拟地址空间中的起始地址*/
        unsigned long vm_end;        /* 在进程虚拟内存中的结束地址 */
        struct vm_area_struct *vm_next, *vm_prev;
        pgprot_t vm_page_prot;       /* 对这个虚拟内存区域的存取权限 */
        unsigned long vm_flags;      /* 虚拟内存区域的标志. */
        /*对这个虚拟内存区域进行操作的函数 */
        struct vm_operations_struct * vm_ops;
        struct file * vm_file;          /* File we map to (can be NULL). */
              unsigned long vm_pgoff;       /* 文件中的偏移量 */

 };
```

vm_flags的取值：

VM_READ：

　此虚拟内存区域可读

VM_WRITE：

　此虚拟内存区域可写

VM_EXEC：

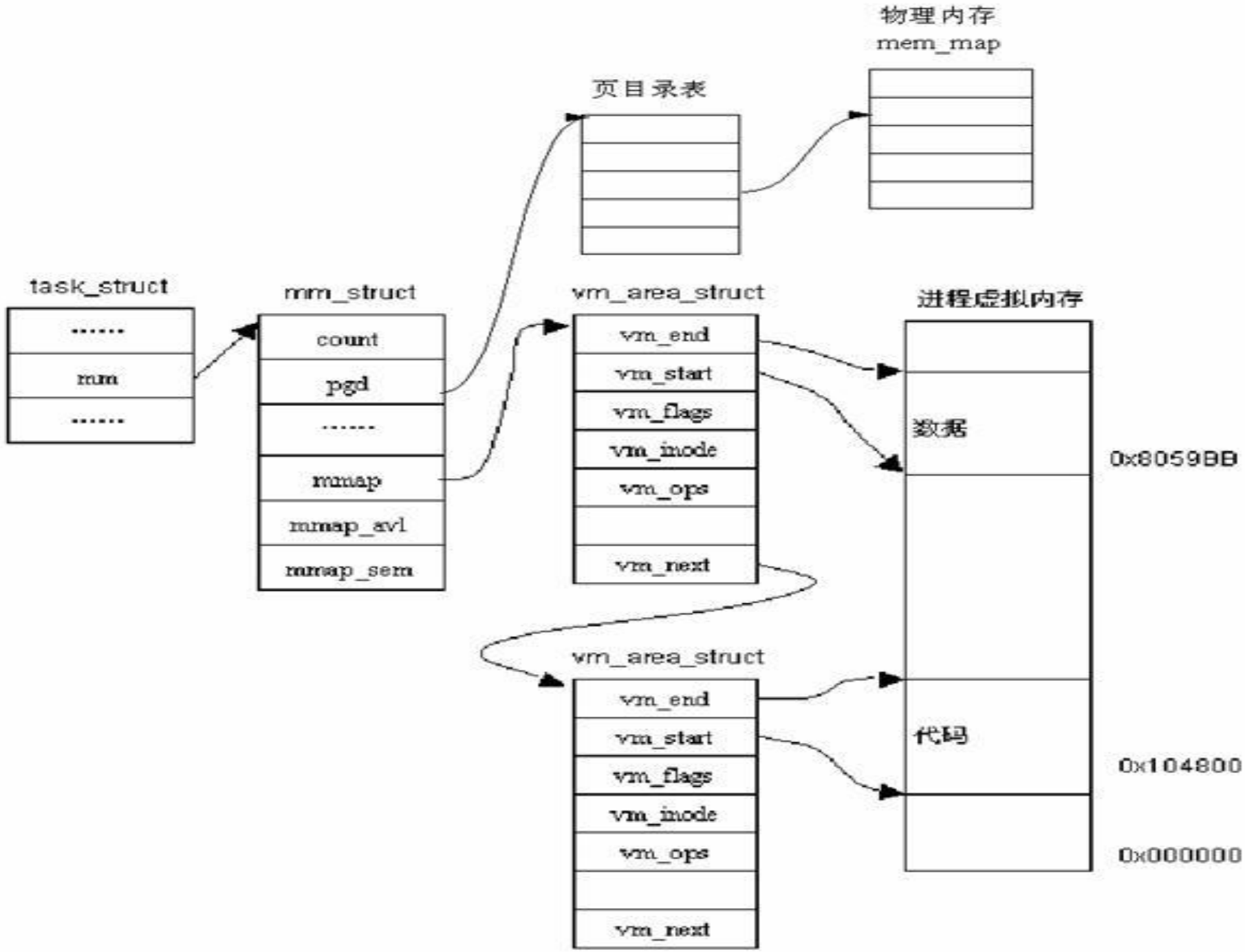
　此虚拟内存区域可执行

VM_SHARED：

　此虚拟内存区域可在多个进程之间共享

VM_IO：

　此虚拟内存区域映射设备I/O空间


物理内存
mem_map
页目录表
task_struct
.....
mm
.....
mm_struct
count
pgd
.....
mmap
mmap_avl
mmap_sem
vm_area_struct
vm_end
vm_start
vm_flags
vm_inode
vm_ops
vm_next
进程虚拟内存
数据
0x8059BB
vm_area_struct
vm_end
vm_start
vm_flags
vm_inode
vm_ops
vm_next
代码
0x104800
0x000000

线程之间共享内存地址的实现机制：

在linux中，如果clone()时设置CLONE_VM标志，我们把这样的进程称作为线程。线程之间共享同样的虚拟内存空间。
fork()函数利用copy_mm()函数复制父进程的mm_struct,也就是current->mm域给其子进程。

kernel/fork.c

```
static int copy_mm(unsigned long clone_flags, struct task_struct *tsk){

}
```

用户空间的mmap

```
#include <sys/mman.h>

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

测试:
把一个文件用mmap()映射到进程地址空间里

用户空间的mmap的内核实现

用户空间的mmap()会通过系统调用调用到内核的do_mmap()函数。
do_mmap()函数会:
1.首先创建一个新的VMA并初始化，然后加入进程的虚拟地址空间里。
2.然后调用底层的mmap函数建立VMA和实际物理地址的联系（建立页表）

驱动的mmap实现

设备驱动的mmap实现主要是将这个物理设备的可操作区域映射到一个进程的虚拟地址空间。这样用户空间就可以直接采用指针的方式访问设备的可操作区域。在驱动中的mmap实现主要是完成一件事，就是建立设备的可操作区域到进程虚拟空间地址的映射过程。同时也需要保证这段映射的虚拟存储器区域不会被进程当做一般的空间使用，因此需要添加一系列的保护方式。

驱动的mmap建立虚拟地址和物理地址的映射

建立vma和物理地址的映射的工作由remap_pfn_range来完成， 原型如下：

```
int remap_pfn_range(struct vm_area_struct *vma, unsigned long virt_addr, unsigned long
    pfn, unsigned long size, pgprot_t prot);
```

vma

需要建立映射的VMA

virt_addr

需要建立映射的VMA的起始地址

pfn

页帧号, 对应虚拟地址应当被映射的物理地址. 这个页帧号简单地是物理地址右移 PAGE_SHIFT 位

size

需要建立映射的VMA的大小, 以字节.

prot

使用在 vma->vm_page_prot 中找到的值.