Linux内核 Lab03 - 内存管理

实验内容

写一个模块mtest，当模块加载时，创建一个proc文件 /proc/mtest，该文件接收三种类型的参数，具体如下：

- `listvma` 打印当前进程的所有虚拟内存地址，打印格式为`start-addr end-addr permission`
- `findpage addr` 把当前进程的虚拟地址转化为物理地址并打印，如果不存在这样的翻译，则输出`translation not found`
- `writeval addr val` 向当前地址的指定虚拟地址中写入一个值。

注：所有输出可以用`printk`来完成，通过`dmesg`命令查看即可

实验思路与实现过程

0. 模块框架

在实现三个功能前，先准备一个模块框架处理用户输入，然后再研究如何实现目标功能；

0.0. 创建 /proc/文件

方式和lab01完全一样，略过，重点是修改写入函数的具体操作:;

0.1. 处理命令

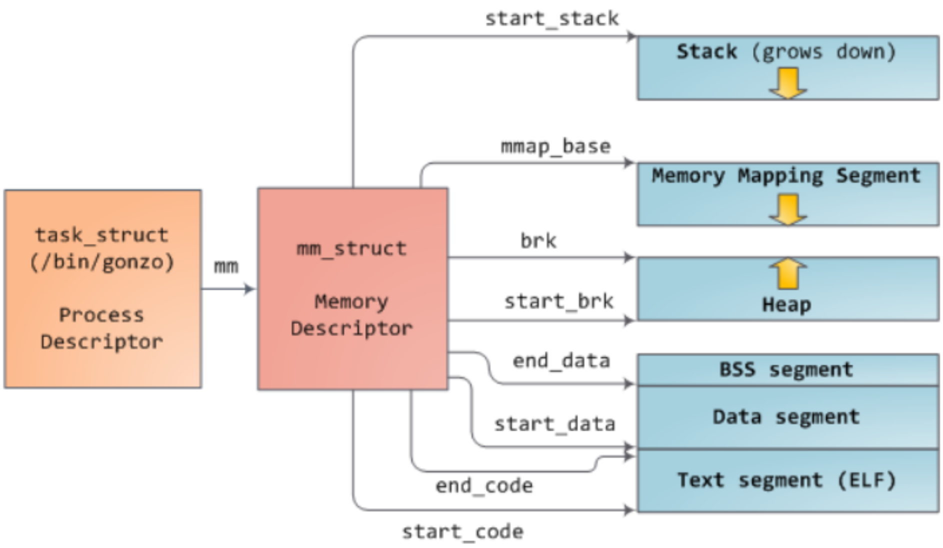
对命令进行简单的字符处理、将其转换为对应的数据类型，并根据命令调用三个希望实现的函数，与实验目的关联不大，略过，详见源代码；

1. 查找当前进程的所有虚拟内存地址

在描述进程的task_struct (位于<linux/sched.h>)数据结构中成员中有一个指向进程内存管理的数据结构的指针：

```
struct mm_struct        *mm;
```

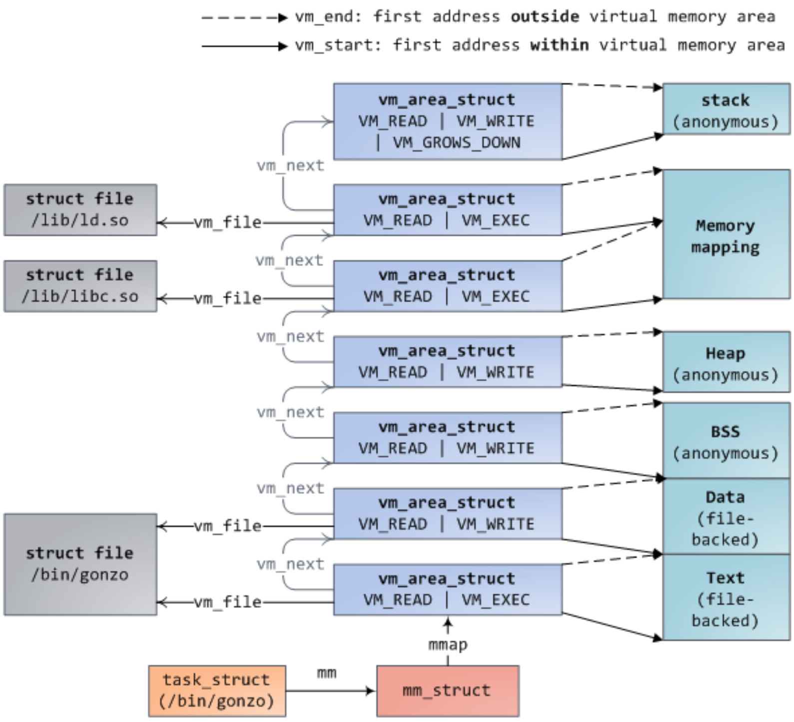
mm_struct 描述了对应进程内存管理的状态, 如图[1]:



其成员中有指针

```
struct vm_area_struct *mmap;          /* list of VMAs */
```

指向一张链表 (vm_area_struct 定义于 <linux/mm_types.h>)，对应了每一块连续虚拟地址的起始地址、是否可读、是否可写等信息，如下图[1]:



起始地址为vm_start, 终止地址为vm_end, 是否可读位为VM_READ, 是否可写位为VM_WRITE;

所以要找到一个进程的所有虚拟地址，就需要遍历这个链表;

而我们研究的是当前进程，在linux源码中已经定义了一个宏current，即一个指向当前进程的进程描述符的指针，使用它就能够遍历这个链表;

同时需要注意访问链表的时候要先对其"加锁"，结束之后"解锁"，以防止冲突;

实现如下:

```
/*
```

```
    Print all vma of the current process
*/
static void mtest_list_vma(void)
{
    struct vm_area_struct* p;   //line 292 in mm_type.h

    printk(KERN_INFO"Virtual memory area:\n");
    down_write(&current->mm->mmap_sem);
        for(p = current->mm->mmap;p!=NULL;p=p->vm_next)
        {
            printk(
                KERN_INFO"0x%08lx - 0x%08lx\t%s %s\n",
                p->vm_start, p->vm_end,
                (p->vm_flags & VM_READ)? "readable":"not readable",
                (p->vm_flags & VM_WRITE)? "writable":"not writable"
            );
        }
    up_write(&current->mm->mmap_sem);
}
```

2. 将确定的虚拟内存地址转化为物理内存

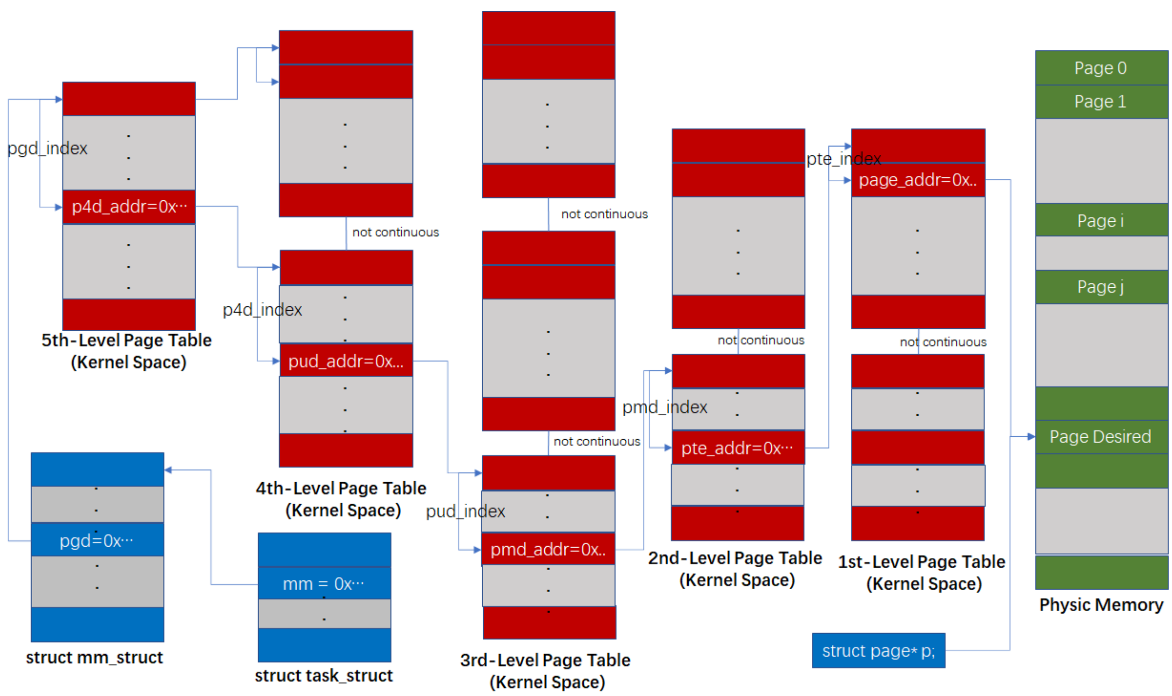
这个转化过程需要通过页表来完成，实验使用的64位linux内核有五级页表，其64位虚拟地址的组成如下:

63 PGD_SHIFT P4D_SHIFT PUD_SHIFT PMD_SHIFT PAGE_SHIFT 0

pgd_index	p4d_index	pud_index	pmd_index	pte_index	page_offset

5-Level Page Table in Linux (64-bit)

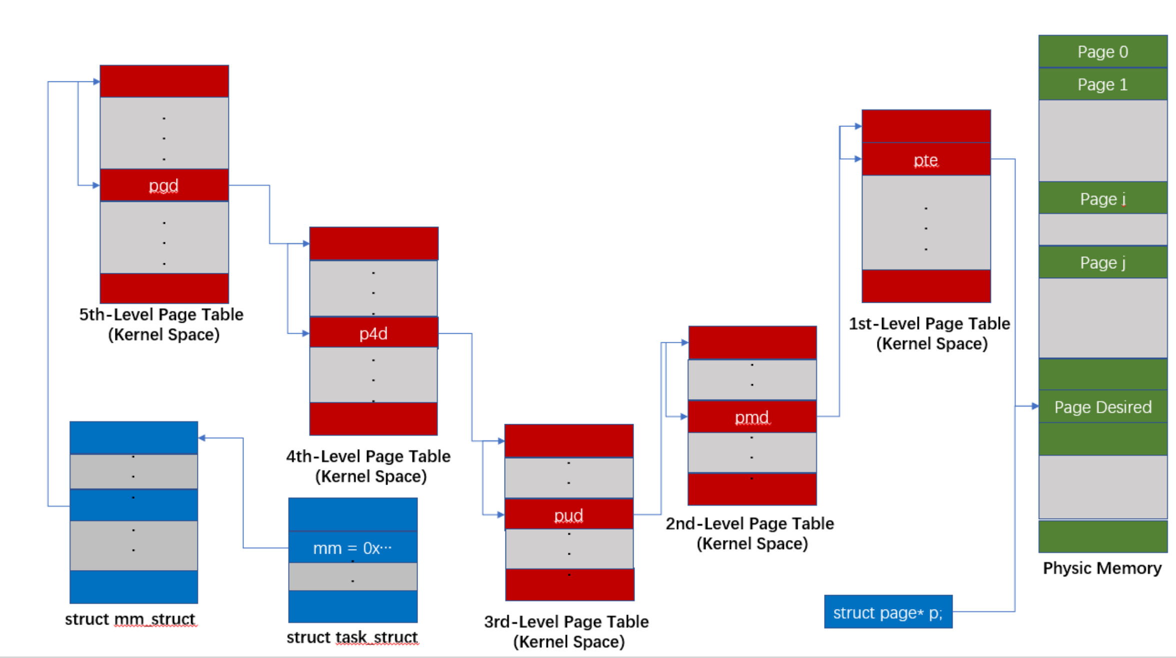
整个5级页表的结构即相关参数可以用下图表来理解:



但事实上，内核源码并非使用unsigned long类型来表示各级页表及其物理地址，而是对每一级页表定义了一个数据结构（尽管这个数据结构可能就是宏定义为unsigned long），并提供了相关宏和函数，来让编程人员更恰当地访问页表:

	数据结构	从vaddr获取表示下一级页表(或物理页)的数据结构(返回指针)	是否合法
	task_struct* p	pgd = pgd_offset(p->mm, addr)	
5	pgd_t* pgd	p4d = p4d_offset(pgd, addr)	pgd_none(*pgd)
4	p4d_t* p4d	pud = pud_offset(p4d, addr);	p4d_none(*p4d)
3	pud_t* pud	pmd =pmd_offset(pud, addr);	pud_none(*pud)
2	pmd_t* pmd	pte = pte_offset_kernel(pmd, addr);	pmd_none(*pmd)
1	pte_t* pte	pte_val(*pte) & PAGE_MASK //物理页地址 (unsigned long)	pte_none(*pte)
		page = pte_page(*pte)	
Ph	page* page		

通过这些接口，我们可以很方便地访问到指向物理页面的数据结构，所以在linux中的实际访问过程应该如下图所示：



根据上面的思路，就可以得到翻译虚拟地址函数的具体的代码实现为：

```c
// 在源文件中还添加有一些辅助理解的输出
static void mtest_find_page(unsigned long addr)
{
    pgd_t* pgd;
    p4d_t* p4d;
    pud_t* pud;
    pmd_t* pmd;
    pte_t* pte;

    pgd = pgd_offset(current->mm, addr);
    if(pgd_none(*pgd))
    {
        printk("wrong pgd\n");
        return- 1;
    }

    p4d = p4d_offset(pgd, addr);
    if(p4d_none(*p4d))
    {
        printk("wrong p4d\n");
        return -1;
    }

    pud = pud_offset(p4d, addr);
    if(pud_none(*pud))
    {
        printk("wrong pud\n");
        return-1;
    }

    pmd = pmd_offset(pud, addr);
    if(pmd_none(*pmd))
    {
        printk("wrong pmd\n");
        return-1;
    }

    pte = pte_offset_kernel(pmd, addr);
    if(pte_none(*pte))
    {
        printk("wrong pte\n");
        return-1;
    }

    /*
        physical address = [page frame address | page offset]
            where
            page frame address = pte_val(*pte) & PAGE_MASK
                and
                page offset = addr & ~ PAGE_MASK
    */
    unsigned long phy_addr = (pte_val(*pte) & PAGE_MASK) | (addr & ~PAGE_MASK);

    printk(KERN_INFO"Virtual address: \t0x%08lx\n", addr);
    printk(KERN_INFO"Physical address:\t0x%08lx\n", phy_addr);
}
```

3. 将一个值写入虚拟地址对应的物理地址

有了2.中的思路，我们就可以封装一个函数，来直接获取需要的物理页的入口:

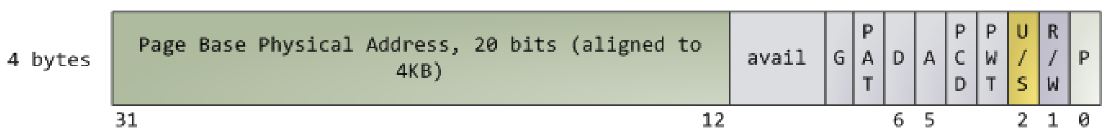
```
pte_t* get_pte_by_vm(unsigned long addr)
{
    pgd_t* pgd;
    p4d_t* p4d;
    pud_t* pud;
    pmd_t* pmd;
    pte_t* pte;

    pgd = pgd_offset(current->mm, addr);
    ...
    p4d = p4d_offset(pgd, addr);
    ...
    pud = pud_offset(p4d, addr);
    ...
    pmd = pmd_offset(pud, addr);
    ...
    pte = pte_offset_kernel(pmd, addr);
    ...
    return pte;
}
```

获取了物理页入口后，我们再通过宏

```
pte_write(*pte)
```

来判断对应物理页是否可写;

pte_t 是一个32位的数据结构，只有31-12位是对应页的物理地址（物理页地址必须是物理页长度4k的整数倍，即后12位必须为0），剩余12位含有一些描述这个物理页的flag, 如下图[1]:



通过宏 pte_page(*pte)来获取对应物理页的描述符;

在实验过程中，如果尝试直接计算物理地址phy_addr, 然后赋值(* phy_addr) = val, 则会直接造成kernel panic, 查资料显示这应该是内核的保护机制造成的;

正确的访问地址方法应该是建立一个(临时的)映射关系, 即使用kmap()函数:

```
pte_t* pte = get_pte_by_vm(addr);
...
page = pte_page(*pte);

/*
```

```c
    Build a temporary map
    from the virtual address to the physical page.
    Now we can access this page through the porinter vaddr.
*/
void *vaddr = kmap(page);
((unsigned long*)vaddr)[addr & ~PAGE_MASK] = val;

/*
    Test if we write to the write place
*/
printk("Test: value  0x%08lx is written\n",
            ((unsigned long*)vaddr)[addr & ~PAGE_MASK]);
/*
    Unmap this page.
*/
kunmap(vaddr);
```

由上述建立映射的操作，访问指针vaddr的时候实际上访问的是对应物理页；

实验效果

1. echo listvma > /proc/mtest

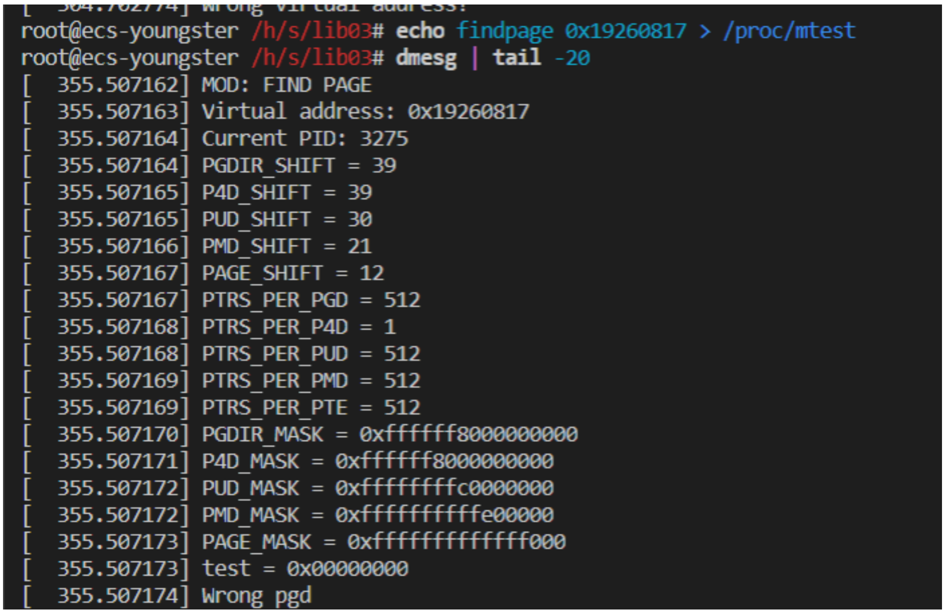
```
root@ecs-youngster /h/s/lib03# echo listvma> /proc/mtest
root@ecs-youngster /h/s/lib03# dmesg | tail -80
[  285.317926] MOD: LIST VMA
[  285.317927] Virtual memory area:
[  285.317928] 0xaaaae8317000 - 0xaaaae8424000  readable not writable
[  285.317929] 0xaaaae8434000 - 0xaaaae8439000  readable not writable
[  285.317930] 0xaaaae8439000 - 0xaaaae843a000  readable writable
[  285.317931] 0xaaaae843a000 - 0xaaaae843b000  readable writable
[  285.317931] 0xaaab23370000 - 0xaaab23475000  readable writable
[  285.317932] 0xffff94000000 - 0xffff94021000  readable writable
[  285.317933] 0xffff94021000 - 0xffff98000000  not readable not writable
[  285.317933] 0xffff9c000000 - 0xffff9c021000  readable writable
[  285.317934] 0xffff9c021000 - 0xffffa0000000  not readable not writable
[  285.317935] 0xffffa3dbc000 - 0xffffa3dbd000  not readable not writable
[  285.317936] 0xffffa3dbd000 - 0xffffa45bd000  readable writable
[  285.317936] 0xffffa45bd000 - 0xffffa45be000  not readable not writable
[  285.317937] 0xffffa45be000 - 0xffffa4dbe000  readable writable
[  285.317938] 0xffffa4dbe000 - 0xffffa4de6000  readable not writable
[  285.317939] 0xffffa4de6000 - 0xffffa4df0000  readable not writable
[  285.317939] 0xffffa4df0000 - 0xffffa4dff000  not readable not writable
[  285.317940] 0xffffa4dff000 - 0xffffa4e00000  readable not writable
[  285.317941] 0xffffa4e00000 - 0xffffa4e01000  readable writable
[  285.317942] 0xffffa4e01000 - 0xffffa4e07000  readable writable
[  285.317942] 0xffffa4e07000 - 0xffffa4e19000  readable not writable
[  285.317943] 0xffffa4e19000 - 0xffffa4e28000  not readable not writable
[  285.317944] 0xffffa4e28000 - 0xffffa4e29000  readable not writable
[  285.317945] 0xffffa4e29000 - 0xffffa4e2a000  readable writable
[  285.317945] 0xffffa4e2a000 - 0xffffa4e2c000  readable writable
[  285.317946] 0xffffa4e2c000 - 0xffffa4e36000  readable not writable
[  285.317947] 0xffffa4e36000 - 0xffffa4e45000  not readable not writable
[  285.317948] 0xffffa4e45000 - 0xffffa4e46000  readable not writable
[  285.317948] 0xffffa4e46000 - 0xffffa4e47000  readable writable
[  285.317949] 0xffffa4e47000 - 0xffffa4e4e000  readable not writable
[  285.317950] 0xffffa4e4e000 - 0xffffa4e5d000  not readable not writable
[  285.317950] 0xffffa4e5d000 - 0xffffa4e5e000  readable not writable
[  285.317951] 0xffffa4e5e000 - 0xffffa4e5f000  readable writable
[  285.317952] 0xffffa4e68000 - 0xffffa5146000  readable not writable
[  285.317953] 0xffffa5146000 - 0xffffa514a000  readable writable
[  285.317953] 0xffffa514a000 - 0xffffa51f3000  readable not writable
[  285.317954] 0xffffa51f3000 - 0xffffa5202000  not readable not writable
[  285.317955] 0xffffa5202000 - 0xffffa5203000  readable not writable
[  285.317956] 0xffffa5203000 - 0xffffa5204000  readable writable
[  285.317956] 0xffffa5204000 - 0xffffa5344000  readable not writable
[  285.317957] 0xffffa5344000 - 0xffffa5353000  not readable not writable
```

2. echo findpage xxx > /proc/mtest

2.1.输入一个合法的地址（输出中保留了一些实验过程的辅助理解的变量值，可以看到实验所用的华为云系统的页表分级的具体状态）：

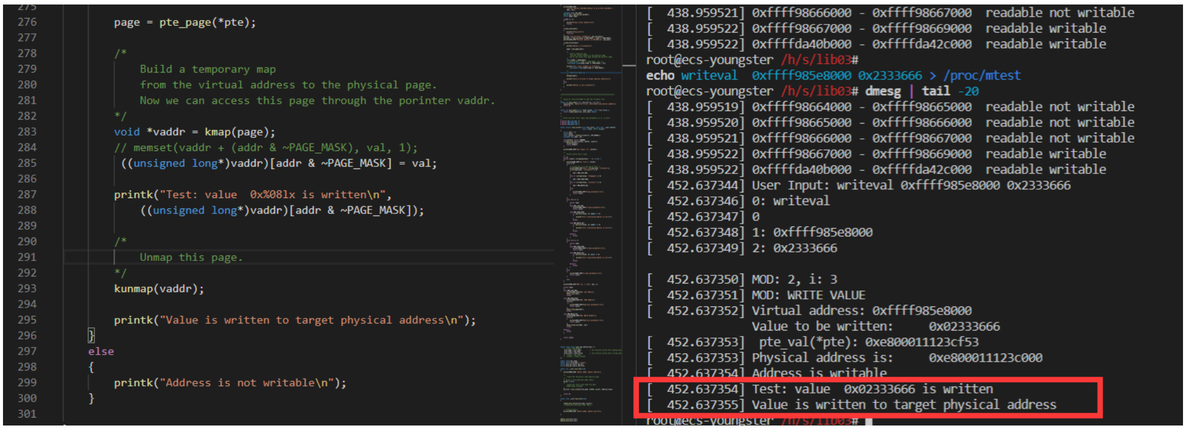
```
root@ecs-youngster /h/s/lib03# echo findpage 0xffffa5619000 > /proc/mtest
root@ecs-youngster /h/s/lib03# dmesg | tail -40
[  370.589808] MOD: FIND PAGE
[  370.589808] Virtual address: 0xffffa5619000
[  370.589809] Current PID: 3118
[  370.589810] PGDIR_SHIFT = 39
[  370.589810] P4D_SHIFT = 39
[  370.589811] PUD_SHIFT = 30
[  370.589811] PMD_SHIFT = 21
[  370.589812] PAGE_SHIFT = 12
[  370.589813] PTRS_PER_PGD = 512
[  370.589813] PTRS_PER_P4D = 1
[  370.589814] PTRS_PER_PUD = 512
[  370.589814] PTRS_PER_PMD = 512
[  370.589815] PTRS_PER_PTE = 512
[  370.589816] PGDIR_MASK = 0xffffff8000000000
[  370.589816] P4D_MASK = 0xffffff8000000000
[  370.589817] PUD_MASK = 0xffffffffc0000000
[  370.589817] PMD_MASK = 0xffffffffffe00000
[  370.589818] PAGE_MASK = 0xfffffffffffff000
[  370.589819] test = 0x00000000
[  370.589819] current->mm->pgd = 0xffff0000ef166000
[  370.589820] pgd = 0xffff0000ef166ff8
[  370.589821] p4d = 0xffff0000ef166ff8
[  370.589821] pud = 0xffff0000f07eeff0
[  370.589822] pmd = 0xffff0000f058c958
[  370.589822] pte = 0xffff0000f1faa0c8
[  370.589823]  pgd_index: 0x000001ff
[  370.589823]  pgd_index(addr): 0x000001ff
[  370.589824]  pud_index(addr): 0x000001fe
[  370.589825]  pmd_index(addr): 0x0000012b
[  370.589825]  pte_index(addr): 0x00000019
[  370.589826]  pgd[pgd_index(addr)]: 0x00000000
[  370.589827]  pud[pud_index(addr)]: 0x00000000
[  370.589827]  pte[pte_index(addr)]: 0x00000000
[  370.589828]  pgd_val(*pgd): 0x1307ee003
[  370.589828]  p4d_val(*p4d): 0x1307ee003
[  370.589829]  pud_val(*pud): 0x13058c003
[  370.589830]  pmd_val(*pmd): 0x131faa003
[  370.589830]  pte_val(*pte): 0xe0000113cc4bd3
[  370.589831] Virtual address:         0xffffa5619000
[  370.589832] Physical address:        0xe0000113cc4000
root@ecs-youngster /h/s/lib03# 
```

2.2. 输入一个非法的地址：



3. echo writeval xxx xxx > /proc/mtest

3.1. 向一个可写的地址写入值 0x2333666



3.2. 向一个不可写的地址写入值，输出不可写

```
echo writeval  0xffff98666000 0x2333666 > /proc/mtest
root@ecs-youngster /h/s/lib03# dmesg | tail -20
[  452.637352] Virtual address: 0xffff985e8000
               Value to be written:     0x02333666
[  452.637353]  pte_val(*pte): 0xe800011123cf53
[  452.637353] Physical address is:     0xe800011123c000
[  452.637354] Address is writable
[  452.637354] Test: value  0x02333666 is written
[  452.637355] Value is written to target physical address
[  568.549540] User Input: writeval 0xffff98666000 0x2333666
[  568.549542] 0: writeval
[  568.549543] 0
[  568.549544] 1: 0xffff98666000
[  568.549545] 2: 0x2333666

[  568.549545] MOD: 2, i: 3
[  568.549546] MOD: WRITE VALUE
[  568.549547] Virtual address: 0xffff98666000
               Value to be written:     0x02333666
[  568.549548]  pte_val(*pte): 0xe0000112f4bfd3
[  568.549549] Physical address is:     0xe0000112f4b000
[  568.549549] Address is not writable
```

实验总结

0. 利用google搜索可以快速找到理论学习中的一些数据结构即函数的名称
1. 在bootlin[3] 中可以快速找到某个名称的数据结构的定义以及实现文件

参考资料

[1]How The Kernel Manages Your Memory

https://manybutfinite.com/post/how-the-kernel-manages-your-memory/

[2]Linux中的kmap

https://zhuanlan.zhihu.com/p/69329911

[3] bootlin

https://elixir.bootlin.com/linux/latest/source