

Project 3: 内存管理

task 1

实验要求

1. 创建名为 mtest 的 proc 文件。
2. 从 proc 文件中读取用户指令，指令形式为 r < pid > < address > 或者是 w < pid > < address > < content >。pid 为目标进程的进程号。address 为 16 进制的目标虚拟地址。r 表示读取该进程该地址的一字节内容，w 表示向该进程的该地址写入一字节 content 的内容。
3. 取得进程的 task_struct，并根据其找到目的地址对应的页，并得到页对应的 struct page 结构体。pfn_to_page 宏可以从页框号得到对应的 struct page 结构体。
4. 因为模块代码处于内核内存空间中，所以并不能直接访问该页的内容，还需要将该页映射到内核内存空间中，内核函数 kmap_local_page 可以完成这项工作。
5. 读取或者修改对应的内存地址并存储下来。
6. 若用户指令为读取指令，用户必须通过读取 proc 文件，得到对应的值。

实验过程

- 1、类似于实验一，在模块初始化函数中首先创建名为 mtest 的 proc 文件

```
proc_ent = proc_create("mtest", 0666, NULL, &proc_ops);
```

- 2、在proc_write函数中从 proc 文件中读取用户指令, 拷贝至内核空间;

```
copy_from_user(buf, ubuf, count)
```

- 3、解析用户指令，根据命令格式分别读取对应的cmd('w'/'r'), pid, addr, val, 并用kstrtoul函数将字符转换为unsigned int类型的数据。

```
kstrtoul(data, 16, &pid);
```

- 4、根据pid目标进程的进程号取得进程的 task_struct 结构体，
首先根据pid进程号得到struct pid结构体，再索引至task_struct 结构体。

```
task = get_pid_task(find_get_pid(pid), PIDTYPE_PID);
```

[/kernel/pid.c](#)

```
struct task_struct *get_pid_task(struct pid *pid, enum pid_type type)
{
    struct task_struct *result;
    rcu_read_lock();
    result = pid_task(pid, type);
    if (result)
        get_task_struct(result);
    rcu_read_unlock();
    return result;
}
```

```
EXPORT_SYMBOL_GPL(get_pid_task);

struct pid *find_get_pid(pid_t nr)
{
    struct pid *pid;

    rcu_read_lock();
    pid = get_pid(find_vpid(nr));
    rcu_read_unlock();

    return pid;
}
EXPORT_SYMBOL_GPL(find_get_pid);
```

5、得到页对应的 struct page 结构，可以像实例中的做法，首先根据虚拟地址得到物理地址，再将物理地址偏移PAGE_SHIFT（12）得到page frame number，用pfn_to_page 宏可以从页框号得到对应的 struct page 结构体。

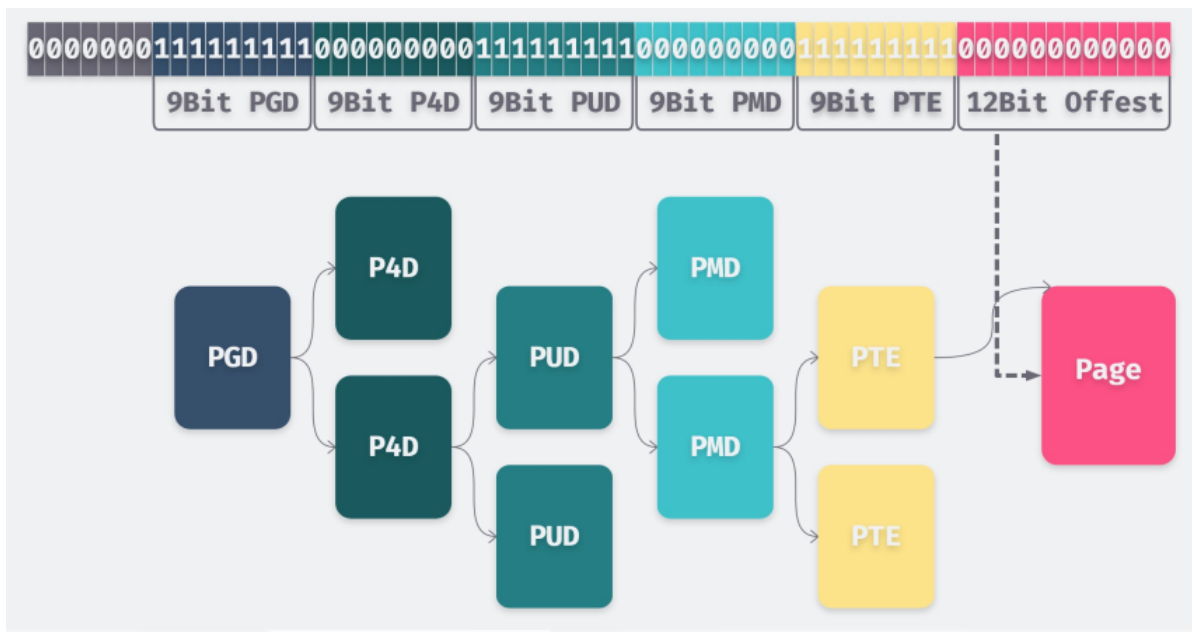
```
page = pfn_to_page(addr >> PAGE_SHIFT);
```

本实验采用另一种做法（本质一样）

Linux 中为各种硬件的页式寻址提供了一套通用的抽象模型

支持 5 级页表

- 页全局目录 PGD
- 页四级目录 P4D
- 页上级目录 PUD
- 页中级目录 PMD
- 页表项 PTE



每个进程的进程描述符中的 mm 字段保存有这个进程的内存管理相关信息，其中 mm->pgd 是这个进程的页表地址。

```

struct mm_struct {
    struct {
        struct vm_area_struct *mmap;           /* list of VMAs */
        struct rb_root mm_rb;
        u64 vmacache_seqnum;                   /* per-thread vmacache
        .....
        unsigned long task_size;               /* size of task vm space */
        unsigned long highest_vm_end;          /* highest vma end address */
        pgd_t * pgd;
    }
}

```

pgd基址的计算

```

#define pgd_offset(mm, addr)    (((mm)->pgd) + pgd_index(addr))

```

逐级计算物理地址：

```

pgd = pgd_offset(mm, vaddr); // pgd 基址
p4d = p4d_offset(pgd, vaddr); // pgd 基址 + pgd → p4d 基址
pud = pud_offset(p4d, vaddr); // p4d 基址 + p4d → pud 基址
pmd = pmd_offset(pud, vaddr); // pud 基址 + pud → pmd 基址
pte = pte_offset_kernel(pmd, vaddr); // pmd 基址 + pmd → pte基址

```

再由pte得到page

```

curr_page = pte_page(*pte);

```

pte_page的定义如下，也是用计算pfn得到page的。

```

#define pte_pfn(pte)    (pte_val(pte) >> 32)
#define pte_page(pte)    pfn_to_page(pte_pfn(pte))

```

6、将该页映射到内核内存空间中并访问

```

kernel_addr = kmap(curr_page); // map to kernel vaddr
page_offset = vaddr & (~PAGE_MASK); // offset of vaddr
kernel_addr += page_offset; // base + offset

```

7、对该位置进行读 or 写操作

```

*kernel_addr = val; // write val

```

```

return *kernel_addr; // return val

```

实验步骤

切换至该目录下，编译

```

make

```

安装模块

```
sudo insmod mtest.ko
```

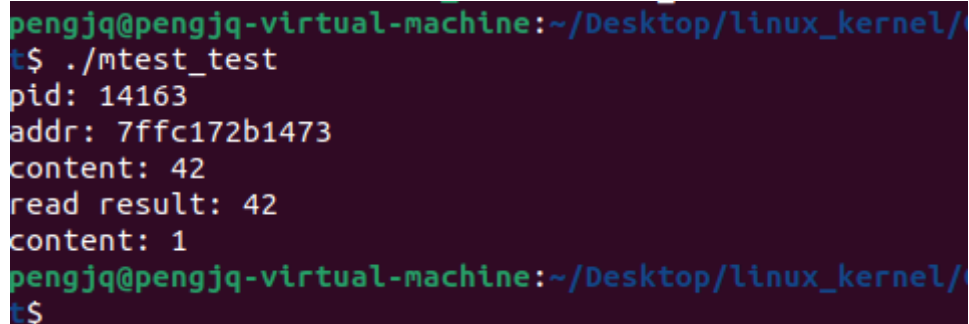
运行程序

```
./mtest_test
```

校验结果后卸载模块

```
sudo rmmod mtest.ko
```

实验结果



```
pengjq@pengjq-virtual-machine:~/Desktop/linux_kernel/
t$ ./mtest_test
pid: 14163
addr: 7ffc172b1473
content: 42
read result: 42
content: 1
pengjq@pengjq-virtual-machine:~/Desktop/linux_kernel/
t$
```

实验心得

本实验基于内存管理的相关理论，读写任意进程的内存位置。通过此次实验我深入理解了linux系统中内存是如何组织的，明白了分页机制以及虚拟地址到物理地址是如何映射的。在动手时按照实验提示一步步做，但是有些函数还是得自己寻找，比如pfn_to_page 宏必须先得到pfn才行，之前对页框号的由来不太理解，通过不断地搜寻资料，查看源码，并向助教确认自己的方法才明白其含义。经过这次实验，让我再一次意识到linux内核代码的庞大，仅仅是对页表的处理，就定义了如此多的API，有些函数功能或者变量定义非常相近，看得我眼花缭乱，不知道用哪个好，通过加深对内存管理机制的理解，对源码也逐渐清晰起来，学会调用正确的函数实现其功能。在此次实验中出现过许多次内核模块安装不上，读写时系统崩溃的情况，只能重启和重装系统继续实验，但是我仍收获了不少经验和理解。

task2

实验要求

编写一个内核模块，创建一个 proc 文件，在用户使用 mmap 系统调用时，为用户设置好对应的映射。具体来说，该模块需要完成以下操作：

1. 创建 proc 文件，同时设置好该文件被调用 mmap 系统调用时的回调函数。这个可以通过设置 struct proc_ops 结构体的 proc_mmap 成员实现。
2. 通过 alloc_page 函数分配一个物理页，并向物理页中写入一些特殊内容，方便和其他页做区分。在写入内容之前，和第一部分的模块一样，也要将该物理页映射到内核内存空间中。
3. 当 proc 文件的 proc_mmap 回调函数被调用时，利用 remap_pfn_range 函数将之前所分配的页与用户内存空间对应起来。
4. 用户可以直接访问 mmap 映射得到的内存空间并读到写入的特殊内容。

实验过程

1、创建 proc 文件

```
proc_ent = proc_create("maptest", 0666, NULL, &proc_ops);
```

2、设置该文件被调用 mmap 系统调用时的回调函数，proc_mmap在/include/linux/proc_fs.h#L43中定义，根据其声明，设置回调的方法。

```
int (*proc_mmap)(struct file *, struct vm_area_struct *);
```

proc_mmap中所需实现的功能是利用 remap_pfn_range 函数将之前所分配的页与用户内存空间对应起来。查询remap_pfn_range 函数，定义在/mm/memory.c#L2452中

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

    err = track_pfn_remap(vma, &prot, pfn, addr, PAGE_ALIGN(size));
    if (err)
        return -EINVAL;

    err = remap_pfn_range_notrack(vma, addr, pfn, size, prot);
    if (err)
        untrack_pfn(vma, pfn, PAGE_ALIGN(size));
    return err;
}
EXPORT_SYMBOL(remap_pfn_range);
```

vm_area_struct用于描述虚拟内存空间，相关定义如下

```
struct vm_area_struct {
    /* The first cache line has the info for VMA tree walking. */

    unsigned long vm_start;    /* Our start address within vm_mm. */
    unsigned long vm_end;      /* The first byte after our end address
                                within vm_mm. */
    /* linked list of VM areas per task, sorted by address */
    struct vm_area_struct *vm_next, *vm_prev;
    struct rb_node vm_rb;
    unsigned long rb_subtree_gap;
    struct mm_struct *vm_mm;    /* The address space we belong to. */

    pgprot_t vm_page_prot;
    unsigned long vm_flags;     /* Flags, see mm.h. */
    .....
}
```

可以根据vma对象来得到remap_pfn_range的相关参数，其中vm_page_prot是控制访问权限的 " 保护位 "。

```

unsigned long pfn = page_to_pfn(page); // page frame number
unsigned long size = vma->vm_end - vma->vm_start;

// remap kernel memory to userspace
ret = remap_pfn_range(vma, vma->vm_start, pfn, size, vma->vm_page_prot);

```

3、在初始化程序中，通过 `alloc_page` 函数分配一个物理页，为了读写该物理页，使用 `kmap_local_page` 函数将该物理页映射到内核内存空间中。

`alloc_page` 函数由分配 $2^0 = 1$ 页的 `alloc_pages` 函数而来，`gfp_mask` 是内存分配掩码，用于内存分配的设置，`GFP_KERNEL` 就是一种类型标志

```

#define alloc_page(gfp_mask) alloc_pages(gfp_mask, 0)

```

```

#define GFP_KERNEL  (__GFP_RECLAIM | __GFP_IO | __GFP_FS)

```

具体实现如下

```

page = alloc_page(GFP_KERNEL); // order = 0 2^0=1 page
base = kmap_local_page(page);

```

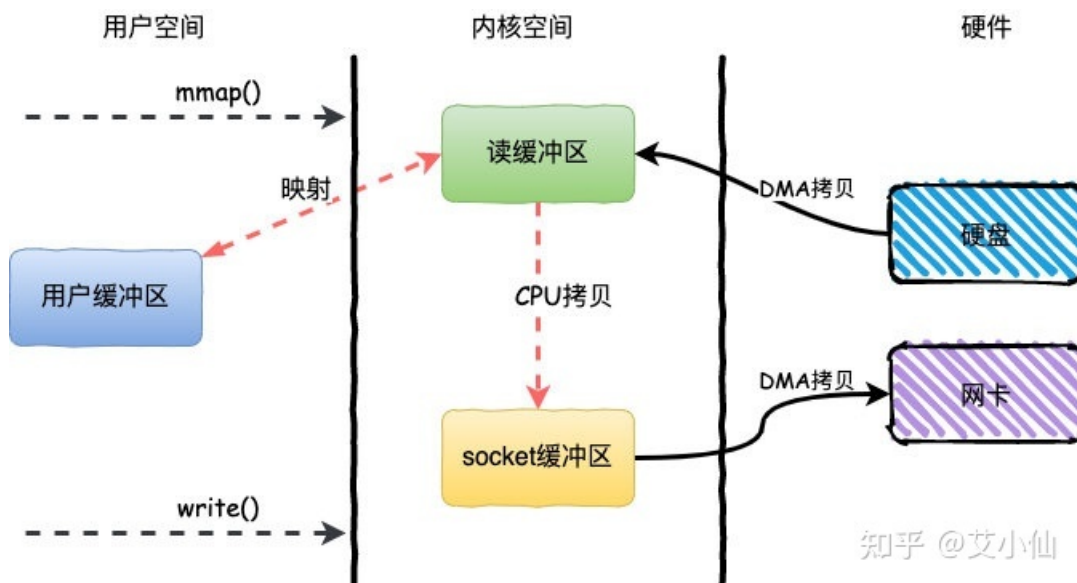
4、向物理页中写入一些设置好的内容，用 `memcpy` 函数复制即可。

```

memcpy(base, content, sizeof(content));
kunmap_local(base); // unmap a page mapped via kmap_local_page()

```

5、当测试程序调用 `mmap` 时，调用回调函数，将设置好的内容复制到物理页中，最后输出。



实验步骤

切换至该目录下，编译

```

make

```

安装模块

```
sudo insmod maptest.ko
```

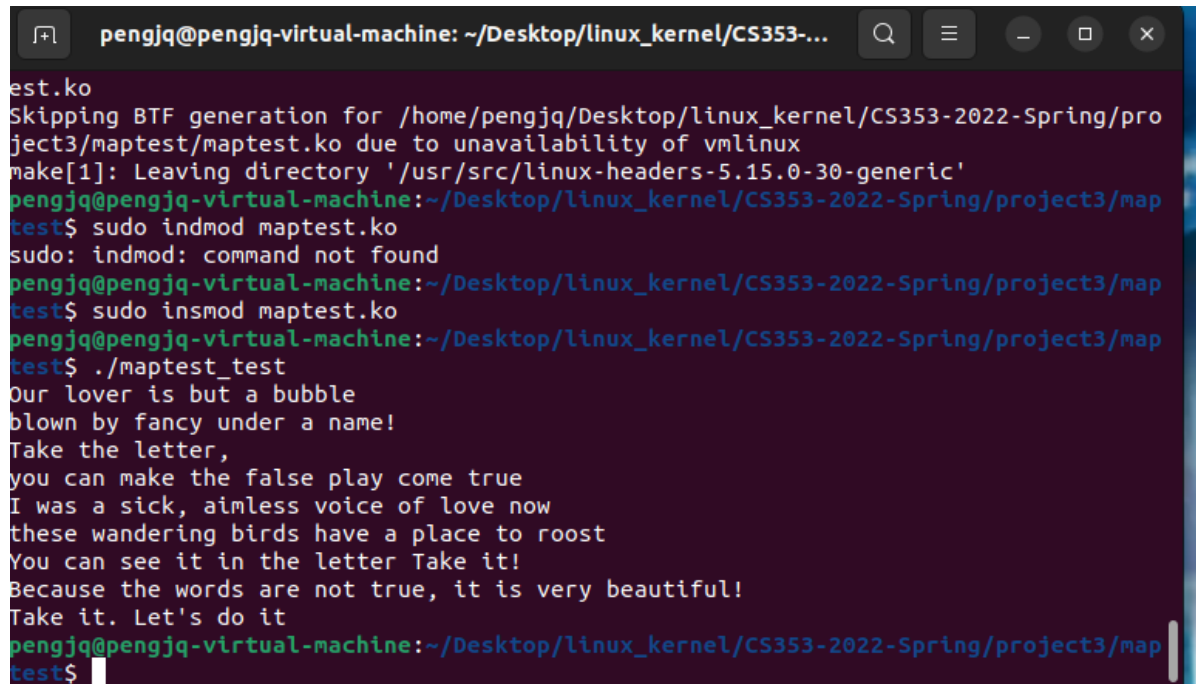
运行程序

```
./maptest_test
```

校验结果后卸载模块

```
sudo rmmod maptest.ko
```

实验结果



```
pengjq@pengjq-virtual-machine: ~/Desktop/linux_kernel/CS353-...
est.ko
Skipping BTF generation for /home/pengjq/Desktop/linux_kernel/CS353-2022-Spring/pro
ject3/maptest/maptest.ko due to unavailability of vmlinux
make[1]: Leaving directory '/usr/src/linux-headers-5.15.0-30-generic'
pengjq@pengjq-virtual-machine:~/Desktop/linux_kernel/CS353-2022-Spring/project3/map
test$ sudo indmod maptest.ko
sudo: indmod: command not found
pengjq@pengjq-virtual-machine:~/Desktop/linux_kernel/CS353-2022-Spring/project3/map
test$ sudo insmod maptest.ko
pengjq@pengjq-virtual-machine:~/Desktop/linux_kernel/CS353-2022-Spring/project3/map
test$ ./maptest_test
Our lover is but a bubble
blown by fancy under a name!
Take the letter,
you can make the false play come true
I was a sick, aimless voice of love now
these wandering birds have a place to roost
You can see it in the letter Take it!
Because the words are not true, it is very beautiful!
Take it. Let's do it
pengjq@pengjq-virtual-machine:~/Desktop/linux_kernel/CS353-2022-Spring/project3/map
test$
```

实验心得

本实验编写一个模块，使得 proc 文件可以通过 mmap 的方式被用户读取。由于实验提示较为详尽，根据实验提示，定义好 proc_mmap 成员函数，分配空间并写入物理页。通过此次实验，让我深入理解了 mmap 机制，并从底层理解了 mmap 机制存在的好处，缩短了 CPU 的拷贝时间，节省内存空间，