

# Linux Kernel Project 3: Memory Management

薛春宇 518021910698

## 1 实验要求

写一个模块 `mtest`，当模块被加载时，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 `Error occurred when finding page based on vma...`
- `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。

## 2 实验环境

- Linux OS 版本：Ubuntu 18.04
- Kernel 版本：Linux-5.5.11
- Gcc 版本：7.4.0

## 3 实验内容

本节中，将分别介绍 `listvma`、`findpage` 和 `writeval` 三个模块的具体实现思路，即涉及到的相关理论知识。此外，我们还将介绍根据用户输入统一调配三个模块的函数 `mtest_proc_write()`，其同时也是用户空间与 `proc` 的接口函数。

### 3.1 模块一： `listvma` 虚拟内存地址打印

内存描述符 `mm_struct`

在 Linux 中，每个进程都有自己的进程描述符 `task_struct`，而当前进程的进程描述符可以从 `current` 变量获得，该变量定义在 `asm/current.h` 头文件中，而 `task_struct` 中的 `mm_struct mm` 则指向了当前进程的内存描述符。`mm_struct` 在 Linux 源码中 `/include/linux/mm_types.h` 头文件的第 370 行定义，其作用是描述当前进程在 Linux 管理视角下虚拟地址空间的所有信息。以下给出 `mm_struct` 的部分字段：

```

1 struct mm_struct {
2     struct {
3         struct vm_area_struct *mmap;    /* list of VMAs */
4         struct rb_root mm_rb;
5         // ...
6         pgd_t * pgd;
7         // ...
8         int map_count;    /* number of VMAs */
9         // ...
10        spinlock_t page_table_lock; /* Protects page tables and some counters */
11        struct rw_semaphore mmap_sem;
12        // ...
13    };

```

各字段的含义分别为：

- `vm_area_struct *mmap`：结构体指针，指向存储虚拟地址空间 VMA 双向链表的节点（将在下一节中详细介绍）
- `struct rb_root mm_rb`：VMA 红黑树根节点，Linux 使用红黑树存储每个进程的全部 VMA，以提高查找效率
- `pgd_t *pgd`：页全局目录 *Page Global Directory*
- `int map_count`：当前进程对应的 VMA 数目
- `spinlock_t page_table_lock`：保护虚拟内存中页表和一些计数变量的自旋锁
- `struct rw_semaphore mmap_sem`：读写信号量。由于虚拟内存区域属于系统临界区，在执行读操作时需要加锁。

### VMA 双向链表节点 `vm_area_struct`

`vm_area_struct` 定义在 `include/linux/mm_types.h` 头文件的第 292 行，其作用是将进程的虚拟内存地址 VMA 组织成一个双向链表。双向链表中的每个节点均包含该段地址的起止地址 `vm_start` 和 `vm_end`，分别指向前后节点的指针 `*vm_next` 和 `*vm_prev`，当前进程的内存描述符 `*vm_mm`，和指示虚拟地址读写权限的 `vm_flags`。在我们实现 `listvma` 模块时，只需要顺序遍历该链表，读取起止地址和读写权限并打印即可。

```

1 struct vm_area_struct {
2     unsigned long vm_start;    /* Our start address within vm_mm. */
3     unsigned long vm_end;    /* The first byte after our end address within vm_mm. */
4     /* linked list of VM areas per task, sorted by address */
5     struct vm_area_struct *vm_next, *vm_prev;
6     struct rb_node vm_rb;
7     // ...
8     struct mm_struct *vm_mm;    /* The address space we belong to. */
9     unsigned long vm_flags;    /* Flags, see mm.h. */
10 };

```

根据 `/include/linux/mm.h` 中的内容，`vm_flags` 的低四位表示其读写权限，即 `0x1` 表示可读，`0x2` 表示可写，`0x4` 为可执行，`0x8` 为可共享。其中，VMA 中与读写权限相关的宏定义在 `include/linux/mm.h` 的 249 ~ 252 行，通过比较 `vm_flags` 和这些宏的值即可判断读写权限。

```

1 #define VM_READ    0x00000001  /* currently active flags */
2 #define VM_WRITE   0x00000002
3 #define VM_EXEC    0x00000004
4 #define VM_SHARED   0x00000008

```

## listvma 模块的实现

具体实现细节见 [Appendix A](#) 中的 `mtest_list_vma()` 函数。

## 3.2 模块二：findpage 虚拟地址到物理地址的转换

### Linux 分页机制

Linux 中的内存管理采用了分页机制，这个机制从最开始的二级页表，到四级页表，在最近发布的 Linux 内核版本中，分页机制已经升级到五级页表。虚拟地址需要通过页表的一级一级的转换，找到基址所在页的页号，再根据索引找到相应物理地址页帧的页帧号，与偏移量结合得到最后的物理地址。

为了探寻本实验环境下的多级页表类型，我们需要阅读实现页表类型的相关头文件，华为鲲鹏服务器的 arm 架构下，页表类型定义在 `arch/arm64/include/asm/pgtable_types.h` 内：

```

1 typedef struct { pteval_t pte; } pte_t;
2 #define pte_val(x) ((x).pte)
3 #define __pte(x) ((pte_t) { (x) } )
4
5 #if CONFIG_PGTABLE_LEVELS > 2
6 typedef struct { pmdval_t pmd; } pmd_t;
7 #define pmd_val(x) ((x).pmd)
8 #define __pmd(x) ((pmd_t) { (x) } )
9 #endif
10
11 #if CONFIG_PGTABLE_LEVELS > 3
12 typedef struct { pudval_t pud; } pud_t;
13 #define pud_val(x) ((x).pud)
14 #define __pud(x) ((pud_t) { (x) } )
15 #endif
16
17 typedef struct { pgdval_t pgd; } pgd_t;
18 #define pgd_val(x) ((x).pgd)
19 #define __pgd(x) ((pgd_t) { (x) } )
20

```

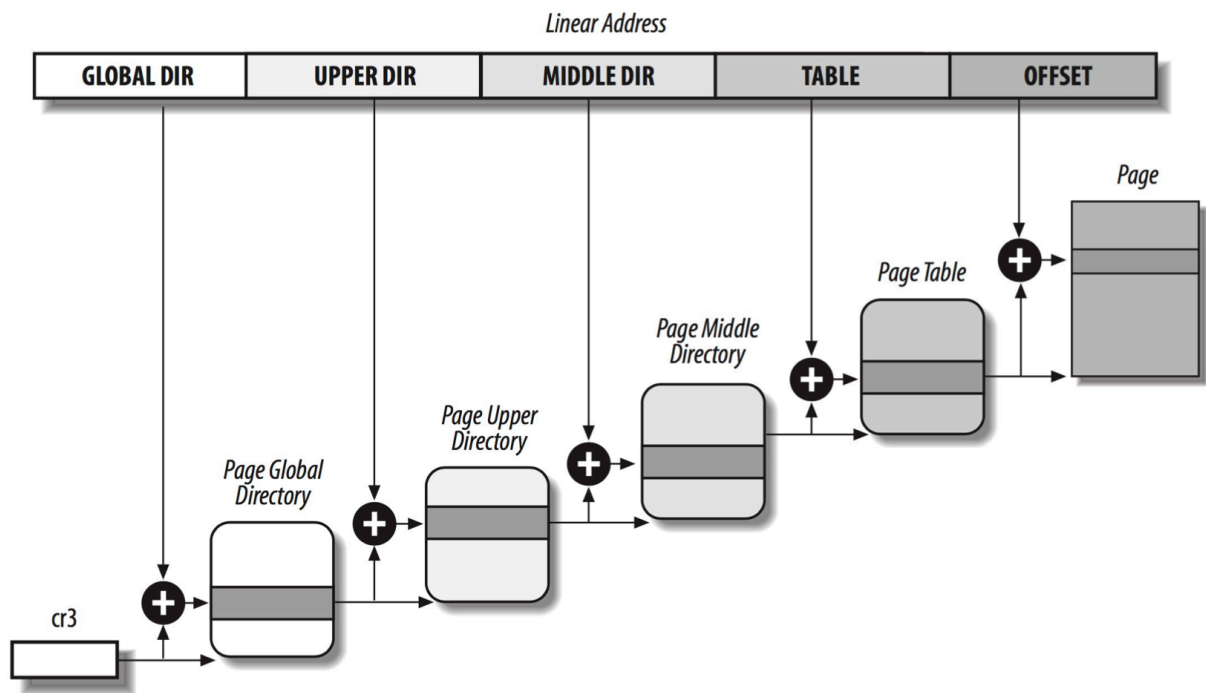
可以看到，当前版本下的 Linux 内核仍采用四级页表。Linux 内存管理四级页表的虚拟地址划分如下：

页全局目录 pgd (9)	页上级目录 pud (9)	页中级目录 pmd (9)	页表 pte (9)	偏移量 offset (12)
------------------	------------------	------------------	---------------	--------------------

Linux 从虚拟地址到物理地址的转换

四级页表的物理地址转换过程如下：

- 从当前进程的进程描述符 `task_struct` 中的 `pgd` 字段获取页全局目录 `pgd` 的基址，可通过 `pgd_offset(mm, addr)` 函数实现；
- `pgd` 基址 + 虚拟地址中的 `pgd` 字段 = `pud` 基址，可通过 `pud_offset(pgd, addr)` 函数实现；
- `pud` 基址 + 虚拟地址中的 `pud` 字段 = `pmd` 基址，可通过 `pmd_offset(pud, addr)` 函数实现；
- `pmd` 基址 + 虚拟地址中的 `pmd` 字段 = `pte` 基址，可通过 `pte_offset_map(pmd, addr)` 函数实现；
- `pte` 基址 + 虚拟地址中的 `pte` 字段 = `page` 基址，可通过 `pte_page(*pte)` 函数实现；
- `page` 基址 + 虚拟地址中的 `offset` 字段 = 物理地址，可通过 `page_to_phys(page)` 函数实现。



findpage 模块的实现

在上述地址转换过程中，`pgd_offset`、`pud_offset` 等函数，包括 `pte_offset_map` 和 `pte_page`，均在 `arch/arm64/include/asm/pgtable.h` 通过宏的形式定义和实现，以 `pgd` 为例，计算时将 `pgd` 基址与偏移量 `pgd_index(addr)` 相加，函数返回 `pgd_t *` 类型的指针。

```
1  /* to find an entry in a page-table-directory */
2  #define pgd_index(addr)    (((addr) >> PGDIR_SHIFT) & (PTRS_PER_PGD - 1))
3  #define pgd_offset_raw(pgd, addr) ((pgd) + pgd_index(addr))
4  #define pgd_offset(mm, addr) (pgd_offset_raw((mm)->pgd, (addr)))
```

`pte_page(*pte)` 函数的返回值是页描述符 `struct *page page`，到此步为模块二的第一部分，我们将这部分单独封装为一个函数 `find_page_based_on_vma(vma, addr)`，同时供模块三 `writeval` 的调用。注意到并非所有的虚拟地址都会对应 `pgd` 等页表层级，因此需要检查上述函数返回的指针是否有效，并利用 `pgtable.h` 中提供的 `pgd_none()` 和 `pgd_bad()` 等函数进行指针的检查。

模块二的第二部分被实现为 `mtest_find_page(addr)`，依次实现以下内容：

- 调用 `find_vma(current->mm, addr)` 函数，利用当前进程的内存描述符和输入的 `addr`，查找存储相应虚拟地址的双向链表结构体 `vm_area_struct`，该函数在 `/mm/mmap.c` 文件中实现，遍历当前虚拟内存所在的

红黑树，根据节点中的 `vm_start` 和 `vm_end` 字段查找 `vma`：

```
1  /* Look up the first VMA which satisfies addr < vm_end, NULL if none. */
2  struct vm_area_struct *find_vma(struct mm_struct *mm, unsigned long addr) {
3      struct rb_node *rb_node;
4      struct vm_area_struct *vma;
5      // ...
6      while (rb_node) {
7          struct vm_area_struct *tmp;
8          tmp = rb_entry(rb_node, struct vm_area_struct, vm_rb);
9          if (tmp->vm_end > addr) {
10             vma = tmp;
11             if (tmp->vm_start <= addr)
12                 break;
13             rb_node = rb_node->rb_left;
14         } else
15             rb_node = rb_node->rb_right;
16     }
17     // ...
18     return vma;
19 }
```

- 调用部分一实现的 `find_page_based_on_vma(vma, addr)` 函数，利用当前进程的 `vma` 和输入的 `addr`，查找虚拟地址所对应的页，若 `page` 为空则报错并返回；
- 基于上一步中函数返回的页描述符 `page`，调用 `page_to_phys(page)` 函数，将虚拟地址对应的页转化为物理地址对应的页帧，并与使用 `PAGE_MASK` 进行 `addr` 位过滤得到的 `offset` 字段结合，得到最终的物理地址。

具体实现细节见 [Appendix B](#) 中的 `find_page_based_on_vma()` 和 `mtest_find_page()` 函数。

### 3.3 模块三： `writeval` 写入指定虚拟地址

#### 内核态虚拟地址空间

当进程要向指定的地址写数据时，所使用的虚拟地址来自于进程私有的虚拟地址空间。然而，当我们调用内核模块来向指定的地址写入数据时，所使用的虚拟地址则来自于内核态共享的虚拟地址空间，两套空间互相独立。因此，为了使用内核模块修改用户态进程中指定的虚拟地址上的值，我们需要首先将该用户态虚拟地址转化为物理地址，再将该物理地址映射到内核态虚拟地址空间。

#### `writeval` 模块的实现

本模块被实现为 `mtest_write_val(addr, val)` 函数，依次实现以下内容：

- 调用 `find_vma(current->mm, addr)` 函数，利用当前进程的内存描述符和输入的 `addr`，查找存储相应虚拟地址的双向链表节点 `vm_area_struct`。若返回的 `vma` 指针为空或不可写，则报错并返回；
- 调用 3.2 节中实现的 `find_page_based_on_vma(vma, addr)` 函数，利用当前进程的 `vma` 和输入的 `addr`，查找虚拟地址所对应的页，若 `page` 为空则报错并返回；

- 调用 `page_address(page)` 函数，将物理地址对应的 `page` 转化为内核态虚拟地址 `kernel_addr`，注意到与之前地址的声明不同的是，这里的 `kernel_addr` 需要被声明为指针，以供后续值的写入。值的写入可以通过 `*kernel_addr=val;` 实现，完成写入后将相关信息通过 `dmesg` 打印到系统信息中。

具体实现细节见 [Appendix C](#) 中的 `mtest_write_val()` 函数。

### 3.4 `proc` 接口函数的实现

在 `proc` 接口函数 `mtest_proc_write(*file, *buffer, count, *data)` 内，我们首先调用函数 `copy_from_user(proc_buf, buffer, count)`，将储存在 `buffer` 内的用户空间内的输入 `copy` 到内核空间里的全局 `buffer` `proc_buf` 内，`count` 是输入总字符数，且若函数返回值非 0，则报错并返回。之后，我们调用 `/include/linux/string.h` 头文件内定义的 `strcmp()` 字符串比较函数，判断输入需要执行哪一个模块，并分别调用相应的函数即可。

```
1 // Implement the interface of proc file, different echo -> different operations
2 static ssize_t mtest_proc_write(struct file *file, const char __user * buffer,
3 size_t count, loff_t * data) {
4     // Address
5     unsigned long int addr;
6     // Value for write
7     unsigned long int value;
8
9     // Copy from user buffer
10    int copy_return_msg = copy_from_user(proc_buf, buffer, count);
11    if(copy_return_msg != 0) {
12        printk(KERN_ERR "Error occurred when copying from user...\n");
13        return -EFAULT;
14    }
15    if (strcmp(proc_buf, "listvma", 7) == 0) {
16        printk(KERN_INFO "Operation: listvma\n");
17        mtest_list_vma();
18    }
19    else if(strcmp(proc_buf, "findpage", 8) == 0) {
20        // Addr input
21        sscanf(proc_buf + 9, "%lx", &addr);
22
23        printk(KERN_INFO "Operation: findpage 0x%lx\n", addr);
24        mtest_find_page(addr);
25    }
26    else if(strcmp(proc_buf, "writeval", 8) == 0) {
27        // Addr and value input
28        sscanf(proc_buf + 9, "%lx %ld", &addr, &value);
29        printk(KERN_INFO "Operation: writeval 0x%lx with %ld\n", addr, value);
30        mtest_write_val(addr, value);
31    }
32    else {
33        printk(KERN_ERR "Error occurred when getting input...\n");
34    }
```

```
34     return count;
35 }
```

## 4 实验结果及分析

### 4.1 Makefile 及编译、内核载入

编写如下 `Makefile`，并使用 `make` 指令对 `mtest.c` 文件进行编译：

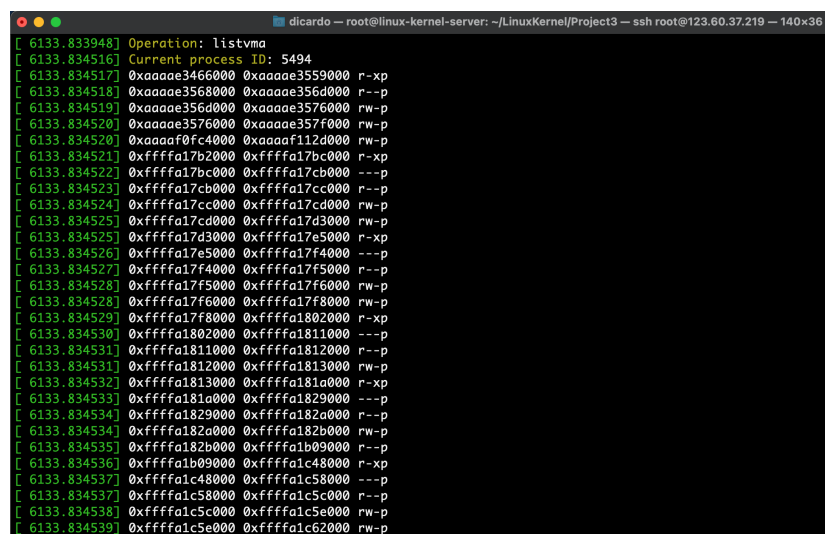
```
1  obj-m := mtest.o
2  KDIR := /lib/modules/$(shell uname -r)/build
3  PWD := $(shell pwd)
4
5  all:
6      make -C $(KDIR) M=$(PWD) modules
7  clean:
8      rm *.o *.ko *.mod.c *.mod Module.symvers modules.order -f
```

完成编译后，执行 `insmod mtest.ko` 指令将内核模块插入到内核态中，并使用 `lsmod | grep mtest` 查看模块是否已经加载成功。

### 4.2 模块一： `listvma`

使用如下指令进行测试：

```
1  echo listvma > /proc/mtest
```



```
[ 6133.833948] Operation: listvma
[ 6133.834516] Current process ID: 5494
[ 6133.834517] 0xaaaae346000 0xaaaae3559000 r-xp
[ 6133.834518] 0xaaaae3568000 0xaaaae356d000 r--p
[ 6133.834519] 0xaaaae356d000 0xaaaae3576000 rw-p
[ 6133.834520] 0xaaaae3576000 0xaaaae357f000 rw-p
[ 6133.834520] 0xaaaae357f000 0xaaaae3580000 r-wp
[ 6133.834521] 0xfffffa17b2000 0xfffffa17bc000 r-xp
[ 6133.834522] 0xfffffa17bc000 0xfffffa17cb000 ---p
[ 6133.834523] 0xfffffa17cb000 0xfffffa17cc000 r--p
[ 6133.834524] 0xfffffa17cc000 0xfffffa17cd000 rw-p
[ 6133.834525] 0xfffffa17cd000 0xfffffa17d3000 rw-p
[ 6133.834525] 0xfffffa17d3000 0xfffffa17e5000 r-xp
[ 6133.834526] 0xfffffa17e5000 0xfffffa17f4000 ---p
[ 6133.834527] 0xfffffa17f4000 0xfffffa17f5000 r--p
[ 6133.834528] 0xfffffa17f5000 0xfffffa17f6000 rw-p
[ 6133.834528] 0xfffffa17f6000 0xfffffa17f8000 r-xp
[ 6133.834529] 0xfffffa17f8000 0xfffffa1802000 r--p
[ 6133.834530] 0xfffffa1802000 0xfffffa1811000 ---p
[ 6133.834531] 0xfffffa1811000 0xfffffa1812000 r--p
[ 6133.834531] 0xfffffa1812000 0xfffffa1813000 rw-p
[ 6133.834532] 0xfffffa1813000 0xfffffa181a000 r-xp
[ 6133.834533] 0xfffffa181a000 0xfffffa1829000 ---p
[ 6133.834534] 0xfffffa1829000 0xfffffa182a000 r--p
[ 6133.834534] 0xfffffa182a000 0xfffffa182b000 rw-p
[ 6133.834535] 0xfffffa182b000 0xfffffa1b09000 r--p
[ 6133.834536] 0xfffffa1b09000 0xfffffa1c48000 r-xp
[ 6133.834537] 0xfffffa1c48000 0xfffffa1c58000 ---p
[ 6133.834537] 0xfffffa1c58000 0xfffffa1c5c000 r--p
[ 6133.834538] 0xfffffa1c5c000 0xfffffa1c5e000 rw-p
[ 6133.834539] 0xfffffa1c5e000 0xfffffa1c62000 rw-p
```

相应的输出如上图。可以发现，有大量虚拟地址位于 `0xfffff80000000` 之后，这段区域是 `64bit Linux` 的内核虚拟地址空间，进程有大量虚拟地址位于该空间内。



## 4.2 模块二：findpage

从模块一的输出结果中选取一个 `vma` 作为模块二测试的输入，使用如下指令进行测试：

```
1 | echo findpage 0xfffffa1829000 > /proc/mtest
```

从下图中的输出可知，模块成功找到该虚拟地址对应的页，虚拟地址 `0xfffffa1829000` 对应的物理地址为 `0x5ad4c000`：

```
[ 6214.117618] Operation: findpage 0xfffffa1829000
[ 6214.117621] Virtual Address: 0xfffffa1829000 -> Physical Address: 0x5ad4c000
```

当随机查找一个未被使用的虚拟地址时，模块则会输出 `Error` 信息，原因是页起始地址 `pte` 无效：

```
[ 6265.958543] Operation: findpage 0xfffffa2729000
[ 6265.958545] Error occurred when finding page based on vma...
```

## 4.3 模块三：writeval

仍然从模块一的输出结果中选取一个可写的 `vma` 作为模块三输入的地址，并选择值 `666`，使用如下指令进行模块三测试：

```
1 | echo writeval 0xfffffa1812000 666 > /proc/mtest
```

从下图的输出可知，模块成功将值 `666` 写入虚拟地址 `0xfffffa1812000` 对应的存储空间内：

```
[ 6528.816132] Operation: writeval 0xfffffa1812000 with 666
[ 6528.816135] Successfully write 666 into virtual address: 0xfffffa1812000
```

接下来，从模块一的输出中选取一个读写权限为不可写的 `vma` `0xfffffa1829000` 作为输入的地址，则会得到 `Error occurred when writing to an unwritable page...` 的报错：

```
[ 6340.146198] Operation: writeval 0xfffffa1829000 with 666
[ 6340.146200] Error occurred when writing to an unwritable page...
```

最后，当我们尝试写入一个未被使用的虚拟地址 `0xfffffa17f6000` 时，模块得到报错 `Error occurred when finding page based on vma...`：

```
[ 6464.114053] Operation: writeval 0xfffffa17f6000 with 666
[ 6464.114556] Error occurred when finding page based on vma...
```

## 5 实验心得

本次实验是 *Linux* 内核课程的第三次正式 *project*，旨在掌握面向 *Linux* 进程的内存管理，包括虚拟内存的打印、转换为物理地址以及特定地址的值写入。在模块一 `listvma` 的实现过程中，我通过阅读源码的方式深入分析了 *Linux* 内存描述符的相关字段，并通过阅读 `vm_area_struct` 结构体的具体实现了解了虚拟内存区间双向链表的组织形式，以及 `vm_flags` 所表示的虚拟地址区间的读写权限；同时，在实现 `findpage` 和 `writeval` 模块的过程中，我初步理解了 *Linux* 内核的分页机制和虚拟地址到物理地址的转换机制，并掌握了对特定地址进行值修改



的方法。

在实验过程中，我首先根据实验指导书上的提示，通过关键词匹配的方式分别找到了几个关键结构体的源码实现，并依次进行调用函数的深入搜索、研究，最终细化到底层实现，力求完全掌握该结构体的功能及所调用函数的意义。由于目前网上针对 *Linux* 内核管理模块的教程大多质量参差不齐，且内核版本也浮动较大，因此我必须不断进行试错，不断修改报错的 *API* 或结构体类名。例如，在本机上的 *Ubuntu 20.04* 中，内核的文件结构体对应的类名是 `proc_ops`，但当我从虚拟机上将代码移植到华为鲲鹏服务器上时，`/proc` 接口处却报错 `proc_ops` 不含有成员 `.proc_write`，在经过一番搜索后仍未找到解决方案，最后尝试将文件结构体修改为 `file_operations`，成功跑通，当前版本的 *Kernel* 中 `linux/proc_fs.h` 支持的文件结构体类型时 `file_operations` 而非 `proc_ops`。

在服务器上进行实验的时候，`dmesg` 时总会看见系统报出大量的报错，在华为云的开发者论坛中搜索后得知，这是当前华为鲲鹏服务器升级到 `Ubuntu20.04` 后存在的一个已知的内核问题，暂时无法解决。[Ref: 鲲鹏服务器一直系统报错](#)

```
1 [drm:virtio_gpu_dequeue_ctrl_func [virtio_gpu]] *ERROR* response 0x1202 (command 0x103)
```

本次实验中，我不仅更深一步地掌握了 *Linux* 源码阅读的方法，增强了自身阅读源码的能力，还提高了发现问题、解决问题的能力。希望能在下面的实验中也能收获满满！

## Appendix A: *listvma* 代码实现

```
1  /*Print all vma of the current process*/
2  static void mtest_list_vma(void) {
3      // Pointer to the linked list in virtual memory
4      struct vm_area_struct *cur = current->mm->mmap;
5      printk(KERN_INFO "Current process ID: %d", current->pid);
6
7      while(cur) {
8          // Permission
9          char permission[5] = "----";
10
11         // Read
12         if(cur->vm_flags & VM_READ) {
13             permission[0] = 'r';
14         }
15         // Write
16         if(cur->vm_flags & VM_WRITE) {
17             permission[1] = 'w';
18         }
19         // Execute
20         if(cur->vm_flags & VM_EXEC) {
21             permission[2] = 'x';
22         }
```

```

23     // Shared or not
24     if(cur->vm_flags & VM_SHARED) {
25         permission[3] = 's';
26     } else {
27         permission[3] = 'p';
28     }
29
30     printk(KERN_INFO "0x%lx 0x%lx %s\n", cur->vm_start, cur->vm_end, permission);
31     cur = cur->vm_next;
32 }
33 }

```

## Appendix B: findpage 代码实现

```

1  /* Find the corresponding page based on the vma */
2  static struct page* find_page_based_on_vma(struct vm_area_struct* vma, unsigned
long addr) {
3      // Memory descriptor of current process
4      struct mm_struct *mm = vma->vm_mm;
5      // Page global directory
6      pgd_t *pgd = pgd_offset(mm, addr);
7      // // Page 4 directory (new to 5 levels page table in Linux)
8      // p4d_t *p4d = NULL;
9      // Page upper directory
10     pud_t *pud = NULL;
11     // Page middle directory
12     pmd_t *pmd = NULL;
13     // Page table
14     pte_t *pte = NULL;
15     // Page
16     struct page *page = NULL;
17
18     // Check and assignment
19     if(pgd_none(*pgd) || pgd_bad(*pgd)) { return NULL; }
20     // // Assign to p4d
21     // p4d = p4d_offset(pgd, addr);
22
23     // if(p4d_none(*p4d) || p4d_bad(*p4d)) { return NULL; }
24     // Assign to pud
25     pud = pud_offset(pgd, addr);
26
27     if(pud_none(*pud) || pud_bad(*pud)) { return NULL; }
28     // Assign to pmd
29     pmd = pmd_offset(pud, addr);
30
31     if(pmd_none(*pmd) || pmd_bad(*pmd)) { return NULL; }
32     // Assign to pte

```

```

33     pte = pte_offset_map(pmd, addr);
34
35     if(pte_none(*pte) || !pte_present(*pte)) { return NULL; }
36     // Page
37     page = pte_page(*pte);
38     if(!page) { return NULL; }
39     pte_unmap(pte);
40     return page;
41 }
42
43
44 /*Find va->pa translation */
45 static void mtest_find_page(unsigned long addr) {
46     // Page
47     struct page *page = NULL;
48     // Physical address
49     unsigned long physical_addr;
50
51     // Find vma
52     struct vm_area_struct *vma = find_vma(current->mm, addr);
53     if(!vma) {
54         printk(KERN_ERR "Error occurred when finding vma...\n");
55         return;
56     }
57
58     page = find_page_based_on_vma(vma, addr);
59     if(!page) {
60         printk(KERN_ERR "Error occurred when finding page based on vma...\n");
61         return;
62     }
63
64     physical_addr = page_to_phys(page) | (addr & ~PAGE_MASK);
65     printk(KERN_INFO "Virtual Address: 0x%lx -> Physical Address: 0x%lx\n", addr,
66     physical_addr);
67 }

```

## Appendix C: writeval 代码实现

```

1  /*Write val to the specified address */
2  static void mtest_write_val(unsigned long addr, unsigned long val) {
3      // Page
4      struct page *page = NULL;
5      // Virtual addr in kernel space
6      unsigned long *kernel_addr;
7
8      // Find vma
9      struct vm_area_struct *vma = find_vma(current->mm, addr);

```

```
10  if(!vma) {
11      printk(KERN_ERR "Error occurred when finding vma...\n");
12      return;
13  }
14
15  // If vm cannot be written
16  if(!(vma->vm_flags & VM_WRITE)) {
17      printk(KERN_ERR "Error occurred when writing to an unwritable page...");
18      return;
19  }
20
21  page = find_page_based_on_vma(vma, addr);
22  if(!page) {
23      printk(KERN_ERR "Error occurred when finding page based on vma...\n");
24      return;
25  } else {
26      kernel_addr = (unsigned long *)page_address(page);
27      // Modify the value
28      *kernel_addr = val;
29      printk(KERN_INFO "Successfully write %ld into virtual address: 0x%lx\n", val,
30      addr);
31  }
```