# Linux 内核 --- Project 3

1. **Mtest module**

这次的实验的任务是编写 mtest 模块，使得通过向 proc 文件写入命令的方式来实现一些内存管理方面的操作。在模块编程方面，和之前的实验是一样的，同样是通过 module_init 以及 module_exit 来声明模块的出口以及入口函数。

在 proc 文件系统方面，同样是使用 proc_create 来创建 proc 文件，而这里与之前不同的是文件需要 write 相关的函数接口，而由于实验里没有要求，其实 read 方面应该不需要相应的接口，这里还是使用之前使用的 seq_file 提供的接口，这里最终要的函数即使 hello_write，它接受写入命令并执行相关操作。

```
41 static const struct file_operations hello_proc_fops = {
42   .owner = THIS_MODULE,
43   .open = hello_proc_open,
44   .read = seq_read,
45   .write = hello_write,
46   .llseek = seq_lseek,
47   .release = single_release,
48 };
```

hello_write 这个函数非常简单，它的原型对应着 file_operations 中的 write 的接口原型，这里的操作即使接受写入的字符（最大为 100 个），然后执行相关操作（由 dealing_it 来执行）。

实验要求中有三种命令，dealing_it 函数的操作即使 parse 写入的字符串，检查是否符合三种操作的命令格式，同时检测出命令的相关参数，如果命令合法，则交给 m_list，m_find 以及 m_write 三个函数来执行，否则 prink 错误信息。

```
113 //the dealing function
114 typedef void (*CMD_HANDLE) (int,char**);
115 #define M_DECLARE_HANDLE(name)  \
116     void name(int,char**)
117 M_DECLARE_HANDLE(m_list);
118 M_DECLARE_HANDLE(m_find);
119 M_DECLARE_HANDLE(m_write);
120
121 #define CMD_TOTAL  3
122 const char* CMDs[CMD_TOTAL] = {"listvma","findpage","writeval"};
123 CMD_HANDLE CMD_TODO[CMD_TOTAL] = {m_list,m_find,m_write};
124 #define MAX_PARAM 10
```

```
149     //then deal it
150     if(argc == 0)//nothing
151         return;
152     for(i=0;i<CMD_TOTAL;i++){
153         if(strcmp(CMDs[i],argv[0])==0){
154             (CMD_TODO[i])(argc,argv);
155             return;
156         }
157     }
158     printk(KERN_INFO"Unkown cmd %s\n",argv[0]);
159     return;
160 }
```

2. **Listvma**

listvma 即是打印出当前进程的虚拟内存的信息，将所有的虚拟内存区域都打印出来。这里所要做的即使遍历一遍当前进程的虚拟内存区域的链表即可。

每个进程都有 task_struct，而每个 task_struct 中都有 mm 这一项，它指向的进程的 mm_struct，普通进程都有这一项，而 mm_struct 中的 mmap 即是虚拟内存区域的链表的开端，进程的虚拟内存区域储存在 vm_area_struct 之中，它们连在一个红黑树之中，并且为了方便遍历，它们同样由双向链表链接，这里只需顺着 vm_next 这一项，即可遍历所有区域。实验要求打印虚拟内存区域的开始，结束以及 rwx 模式，这三者存在 vm_area_struct 的 vm_start，vm_end，vm_flags 之中。

最后，通过 echo 命令测试 echo 进程的虚拟内存情况：

```
[zzs@localhost Project3]$ echo 'listvma' > /proc/hello_proc; dmesg | tail -30
[23203.101782] String is listvma; the argc is 1; argv is listvma;
[23203.101796] The vma are:
[23203.101805] 0x08047000-0x0811e000 r-x
[23203.101811] 0x0811e000-0x0811f000 r--
[23203.101816] 0x0811f000-0x08124000 rw-
[23203.101822] 0x08124000-0x08129000 rw-
[23203.101828] 0x08929000-0x08a59000 rw-
[23203.101834] 0x4d5c4000-0x4d5e3000 r-x
[23203.101839] 0x4d5e3000-0x4d5e4000 r--
[23203.101844] 0x4d5e4000-0x4d5e5000 rw-
[23203.101847] 0x4d5e7000-0x4d797000 r-x
[23203.101851] 0x4d797000-0x4d799000 r--
[23203.101854] 0x4d799000-0x4d79a000 rw-
[23203.101858] 0x4d79a000-0x4d79d000 rw-
[23203.101861] 0x4d79f000-0x4d7a2000 r-x
[23203.101865] 0x4d7a2000-0x4d7a3000 r--
[23203.101868] 0x4d7a3000-0x4d7a4000 rw-
[23203.101872] 0x4db9b000-0x4dbba000 r-x
[23203.101875] 0x4dbba000-0x4dbbc000 r--
[23203.101879] 0x4dbbc000-0x4dbbd000 rw-
[23203.101882] 0xb75b0000-0xb75bb000 r-x
[23203.101886] 0xb75bb000-0xb75bc000 r--
[23203.101889] 0xb75bc000-0xb75bd000 rw-
[23203.101893] 0xb75bd000-0xb77bd000 r--
[23203.101896] 0xb77bd000-0xb77bf000 rw-
[23203.101900] 0xb77d9000-0xb77db000 rw-
[23203.101903] 0xb77db000-0xb77e2000 r--
[23203.101907] 0xb77e2000-0xb77e3000 rw-
[23203.101910] 0xb77e3000-0xb77e4000 r-x
[23203.101914] 0xbff08000-0xbff2a000 rw-
```

### 3. **Findpage**

findpage 的要求即使将虚拟地址转换为物理地址，当然，很有可能虚拟地址没有对应的物理地址（虚拟地址不合法或是其页表被 swap 或是没有使用），查找了许久并没有内存中有直接的函数可以使用，因此参考一种 page_walking 的方法通过页表来寻找物理地址。

Linux 的页表逻辑上分为 4 层，对于 x86-64 系统来说，其 mmu 的却需要 4 层，而对于 x86-32（没有 pae）来说，实际的页表只有 2 层，但当前的 linux 统一使用了 4 层的结构，只是 32 位中有几层并没用到，无论如何，这里考虑了四层的页表，即 pgd，pud，pmd，pte，通过 4 层循环来查找虚拟地址，如果其中有一层发现为空，则虚拟地址没有对应的物理地址；如果最终找到了页表项，而则直接读取其中的内容可以得到页表的物理地址，再加上页内偏移即可。

这里主要用到的函数是 xxx_offset，xxx_none，xxx_bad（xxx 可以替换为 pgd，pud 以及 pmd），pte_offset_map，pte_pfn。其中，pte_pfn 是通过页表内容得到 page frame 的序号，而该页的物理地址即使序号左移 PAGE_SHIFT 位。

这里同样是通过 echo 程序写入 findpage 命令来测试，第一次测试的是非法的虚拟地址，第二次测试的是程序的代码段的虚拟地址：

```
[zzs@localhost Project3]$ echo 'findpage 0x00001234' > /proc/hello_proc; dmesg | tail -5
[24157.103874] 0xb77e3000-0xb77e4000 r-x
[24157.103877] 0xbff08000-0xbff2a000 rw-
[24174.321092] String is findpage; the argc is 2; argv is findpage;0x00001234;
[24174.321107] Command ok: findpage 00001234
[24174.321112] Translation not find for address 0x00001234
```

```
zzs@localhost Project3]$ echo 'findpage 0x08052345' > /proc/hello_proc; dmesg | tail -3
24380.740442] Command ok: findpage 08052345
24380.740449] Page table entry is 2037043237
24380.740453] Translation for address 0x08052345 is 0x796ad000 with offset 345
```

4. **Writeal**

最后一个内容是修改虚拟内存地址的值，由于修改的是当前进程的内存空间，因此直接使用 copy_to_user 即可，这个函数在内核中非常常用，通过<linux/uaccess.h>头文件可以包含这个函数，而 writeval 则通过这个函数可以直接完成。

测试程序即使通过写入文件来修改自身的内存空间的一个变量的值（修改一个局部变量），结果为其被写入的操作所改动：

```c
int w = 100;
printf("Before the value is %d\n",w);
void * p = &w;
char buf[50];
sprintf(buf,"writeval %p %d",p,480);
int fd = open("/proc/hello_proc",O_RDWR);
write(fd,buf,strlen(buf));
close(fd);
printf("After the value is %d\n",w);
return ;
```

```
[zzs@localhost Project3]$ ./write
Before the value is 100
After the value is 480
```

5. **总结**

这次实验是有关内存管理的内容，这一部分十分复杂，同时与 architecture 十分相关，因此实验的完成比起之前费劲不少，当然完成过程中也简化了一些内容（比如访问进程内存管理相关结构的时候没有考虑并发操作，即没有使用这些结构中的锁，这样可能会有一些 sync 方面的问题）。