

# H1 CS353 Linux内核 Project3 报告

517030910214 刘宏洲

## H2 0. 简介

在本Project中，我们将探索有关Linux内存管理方面的内容。

本次Project的实验平台为华为云ECS服务器，它采用双核鲲鹏处理器，系统环境配置如下：

- Ubuntu 18.04.3 LTS (GNU/Linux 5.6.6 aarch64)
- GNU Make 4.1
- gcc 7.4.0

## H2 1. listvma

### H3 1.1 实现

这一部分需要实现打印当前进程的所有虚拟内存地址的功能。创建proc文件和初始化模块的部分已经很熟悉了。我们还需要编写一个处理输入参数的函数。而本次Project中我们只对proc文件写，因此需要在创建proc\_entry的时候传入一个自定义的proc\_ops结构体（以前是file\_operations），其.proc\_write字段为我们的自定义写函数。在这个写函数中，我们处理参数并调用相应函数。具体代码便不再赘述。

现在我们实现listvma功能。首先需要知道当前进程的虚拟地址在哪里，是如何被组织起来的。我们知道，每个进程都有自己的进程描述符task\_struct，而当前进程的task\_struct指针可以从current变量中获得，这个变量被定义在asm/current.h中。而task\_struct中的mm指向了当前进程的内存描述符。观察linux/mm\_types.h中定义的mm\_struct，我们发现，其中定义了名为mmap的vm\_area\_struct结构体指针，其注释显示，这个指针指向虚拟内存区域构成的链表。

同样，在linux/mm\_types.h中，我们可以找到vm\_area\_struct结构体的定义。

```
1 struct vm_area_struct {
2     /* The first cache line has the info for VMA tree walking. */
3
4     unsigned long vm_start; /* Our start address within vm_mm. */
5     unsigned long vm_end; /* The first byte after our end address
6                             within vm_mm. */
7
8     /* linked list of VM areas per task, sorted by address */
9     struct vm_area_struct *vm_next, *vm_prev;
10    .....
11    struct mm_struct *vm_mm; /* The address space we belong to. */
12    pgprot_t vm_page_prot; /* Access permissions of this VMA. */
13    unsigned long vm_flags; /* Flags, see mm.h. */
14    .....
15 } __randomize_layout;
```

可以看到，一个进程的虚拟内存区间组成了一个双向链表，每个节点中保存了这段虚拟内存的起始地址与结束地址，这两个地址表示这个区间为[vm\_start, vm\_end)。而根据mm.h中的内容，vm\_flags的低四位表示其读写权限，即0x1为可读，0x2为可写，0x4为可执行，0x8为可共享。至此，我们可以轻松完成这一部分的代码。

```

1  static void mtest_list_vma(void)
2  {
3      struct vm_area_struct *cur = current->mm->mmap;
4      while (cur)
5      {
6          char perm[5] = "----";
7          if(cur->vm_flags & VM_READ)
8          {
9              perm[0] = 'r';
10         }
11         if(cur->vm_flags & VM_WRITE)
12         {
13             perm[1] = 'w';
14         }
15         if(cur->vm_flags & VM_EXEC)
16         {
17             perm[2] = 'x';
18         }
19         if(cur->vm_flags & VM_SHARED)
20         {
21             perm[3] = 's';
22         }
23         else
24         {
25             perm[3] = 'p';
26         }
27
28         printk(KERN_INFO "0x%lx\t0x%lx\t%s\n", cur->vm_start, cur->vm_end,
29             perm);
30         cur = cur->vm_next;
31     }

```

可见，我们只需要遍历这个 `vm_area_struct` 结构体构成的链表，就可以获得所有的虚拟内存区间。

### H3 1.2 结果

我们直接在命令行中使用以下命令测试：

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

这样，使用 `dmesg` 就可以看到当前进程（这里是zsh）的所有虚拟内存区间。其结果如下：

```

[ 2437.492636] /proc/mtest successfully created
[ 2444.161862] listvma
[ 2444.161866] 0xaaaab89f4000 0xaaaab8a9e000 r-xp
[ 2444.161867] 0xaaaab8aae000 0xaaaab8ab0000 r--p
[ 2444.161868] 0xaaaab8ab0000 0xaaaab8ab6000 rw-p
[ 2444.161868] 0xaaaab8ab6000 0xaaaab8aca000 rw-p
[ 2444.161869] 0xaaaaf6d7000 0xaaaaf816000 rw-p
[ 2444.161870] 0xfffffa5617000 0xfffffa5619000 r-xp
[ 2444.161871] 0xfffffa5619000 0xfffffa5628000 ---p
[ 2444.161871] 0xfffffa5628000 0xfffffa5629000 r--p
[ 2444.161872] 0xfffffa5629000 0xfffffa562a000 rw-p
[ 2444.161873] 0xfffffa562a000 0xfffffa562d000 r-xp
[ 2444.161874] 0xfffffa562d000 0xfffffa563c000 ---p
[ 2444.161874] 0xfffffa563c000 0xfffffa563d000 r--p
[ 2444.161875] 0xfffffa563d000 0xfffffa563e000 rw-p
[ 2444.161876] 0xfffffa564c000 0xfffffa5659000 r-xp
[ 2444.161876] 0xfffffa5659000 0xfffffa5668000 ---p
[ 2444.161877] 0xfffffa5668000 0xfffffa5669000 r--p
[ 2444.161878] 0xfffffa5669000 0xfffffa566a000 rw-p
[ 2444.161879] 0xfffffa566a000 0xfffffa5673000 r-xp
[ 2444.161879] 0xfffffa5673000 0xfffffa5682000 ---p
[ 2444.161880] 0xfffffa5682000 0xfffffa5683000 r--p
[ 2444.161881] 0xfffffa5683000 0xfffffa5684000 rw-p
[ 2444.161881] 0xfffffa5684000 0xfffffa568b000 r-xp
[ 2444.161882] 0xfffffa568b000 0xfffffa569a000 ---p
[ 2444.161883] 0xfffffa569a000 0xfffffa569b000 r--p
[ 2444.161883] 0xfffffa569b000 0xfffffa569c000 rw-p
[ 2444.161884] 0xfffffa569c000 0xfffffa56bb000 r-xp
[ 2444.161885] 0xfffffa56bb000 0xfffffa56cb000 ---p
[ 2444.161886] 0xfffffa56cb000 0xfffffa56cc000 r--p
[ 2444.161887] 0xfffffa56cc000 0xfffffa56cd000 rw-p
[ 2444.161887] 0xfffffa56cd000 0xfffffa5712000 r-xp
[ 2444.161888] 0xfffffa5712000 0xfffffa5722000 ---p
[ 2444.161889] 0xfffffa5722000 0xfffffa5724000 r--p
[ 2444.161889] 0xfffffa5724000 0xfffffa572b000 rw-p
[ 2444.161890] 0xfffffa572b000 0xfffffa572c000 rw-p
[ 2444.161891] 0xfffffa5730000 0xfffffa5732000 r-xp
[ 2444.161891] 0xfffffa5732000 0xfffffa5741000 ---p
[ 2444.161892] 0xfffffa5741000 0xfffffa5742000 r--p
[ 2444.161893] 0xfffffa5742000 0xfffffa5743000 rw-p
[ 2444.161893] 0xfffffa5743000 0xfffffa574a000 r--p
[ 2444.161894] 0xfffffa574a000 0xfffffa5754000 r-xp
[ 2444.161895] 0xfffffa5754000 0xfffffa5763000 ---p
[ 2444.161895] 0xfffffa5763000 0xfffffa5764000 r--p
[ 2444.161896] 0xfffffa5764000 0xfffffa5765000 rw-p
[ 2444.161897] 0xfffffa5765000 0xfffffa576b000 rw-p
[ 2444.161898] 0xfffffa576b000 0xfffffa577d000 r-xp
[ 2444.161898] 0xfffffa577d000 0xfffffa578c000 ---p
[ 2444.161899] 0xfffffa578c000 0xfffffa578d000 r--p
[ 2444.161900] 0xfffffa578d000 0xfffffa578e000 rw-p
[ 2444.161900] 0xfffffa578e000 0xfffffa5790000 rw-p
[ 2444.161901] 0xfffffa5790000 0xfffffa579a000 r-xp
[ 2444.161902] 0xfffffa579a000 0xfffffa5799000 ---p
[ 2444.161902] 0xfffffa5799000 0xfffffa57aa000 r--p
[ 2444.161903] 0xfffffa57aa000 0xfffffa57ab000 rw-p
[ 2444.161904] 0xfffffa57ab000 0xfffffa57b2000 r-xp
[ 2444.161905] 0xfffffa57b2000 0xfffffa57c1000 ---p
[ 2444.161905] 0xfffffa57c1000 0xfffffa57c2000 r--p
[ 2444.161906] 0xfffffa57c2000 0xfffffa57c3000 rw-p
[ 2444.161907] 0xfffffa57c3000 0xfffffa57b2000 r--p
[ 2444.161907] 0xfffffa57b2000 0xfffffa57c3000 r-xp
[ 2444.161908] 0xfffffa57c3000 0xfffffa57c2000 ---p
[ 2444.161909] 0xfffffa57c2000 0xfffffa57c3000 rw-p
[ 2444.161909] 0xfffffa57c3000 0xfffffa57c4000 rw-p
[ 2444.161910] 0xfffffa57c4000 0xfffffa57c5000 rw-p
[ 2444.161911] 0xfffffa57c5000 0xfffffa57c6000 r-xp
[ 2444.161911] 0xfffffa57c6000 0xfffffa57c7000 ---p
[ 2444.161912] 0xfffffa57c7000 0xfffffa57c8000 r--p
[ 2444.161913] 0xfffffa57c8000 0xfffffa57c9000 rw-p
[ 2444.161914] 0xfffffa57c9000 0xfffffa57ca000 r-xp
[ 2444.161915] 0xfffffa57ca000 0xfffffa57cb000 r--p
[ 2444.161915] 0xfffffa57cb000 0xfffffa57cc000 rw-p
[ 2444.161916] 0xfffffa57cc000 0xfffffa57cd000 r-xp
[ 2444.161917] 0xfffffa57cd000 0xfffffa57ce000 ---p
[ 2444.161917] 0xfffffa57ce000 0xfffffa57cf000 r--p
[ 2444.161918] 0xfffffa57cf000 0xfffffa57d0000 rw-p
[ 2444.161919] 0xfffffa57d0000 0xfffffa57d1000 r-xp
[ 2444.161920] 0xfffffa57d1000 0xfffffa57d2000 ---p
[ 2444.161920] 0xfffffa57d2000 0xfffffa57d3000 r--p
[ 2444.161921] 0xfffffa57d3000 0xfffffa57d4000 rw-p
[ 2444.161921] 0xfffffa57d4000 0xfffffa57d5000 r-xp
[ 2444.161922] 0xfffffa57d5000 0xfffffa57d6000 rw-p
[ 2444.161923] 0xfffffa57d6000 0xfffffa57d7000 rw-p
[ 2444.161923] 0xfffffa57d7000 0xfffffa57d8000 r--p
[ 2444.161924] 0xfffffa57d8000 0xfffffa57d9000 r-xp
[ 2444.161925] 0xfffffa57d9000 0xfffffa57da000 r--p
[ 2444.161925] 0xfffffa57da000 0xfffffa57db000 rw-p
[ 2444.161926] 0xfffffa57db000 0xfffffa57dc000 rw-p

```

图1. listvma结果

这里可以发现，有大量的区间处于0xffff80000000以后，这一区域是64bit Linux的内核虚拟地址空间。进程在这里有大量虚拟地址区间存在。

## H2 2. findpage

### H3 2.1 实现

这一部分需要实现虚拟地址向物理地址的转换。我们知道，Linux的内存管理采用了分页机制。而这个机制也从一开始的采用二级页表到四级页表，现在的Linux已经可以支持五级页表了。虚拟地址需要通过页表的一层层转换，找到所在的页，从而找到处于物理内存中的页帧。

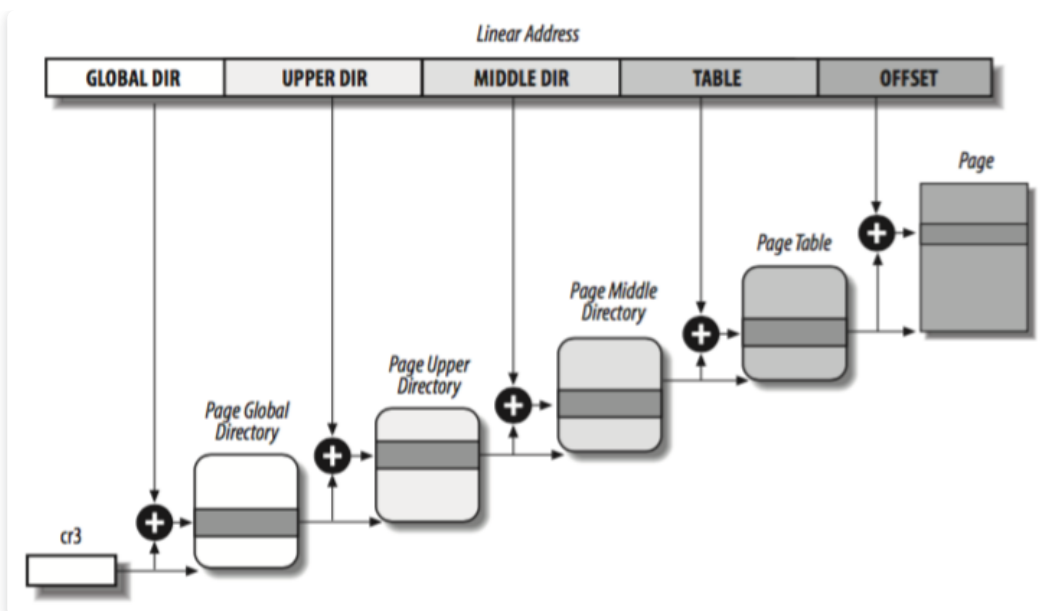


图2. 四级页表

上图是传统的四级页表，而五级页表则是在page global directory (pgd) 后面又增加了一级 page 4 directory (p4d)。而有关的结构体可以在 `asm/pgtable_types.h` (x86)，`asm-generic/5level-fixup.h` 等头文件中找到。

要将虚拟地址翻译为物理地址，只需要不断地取出虚拟地址内对应页表的内容作为偏移量索引页表，最终获得页帧。将最低的12位偏移量与页帧的基址相加，即可获得物理地址。相应代码如下：

```

1  static struct page* _find_page(struct vm_area_struct *vma, unsigned long
   addr)
2  {
3      struct mm_struct *mm = vma->vm_mm;
4      pgd_t *pgd = pgd_offset(mm, addr);
5      p4d_t *p4d = NULL;
6      pud_t *pud = NULL;
7      pmd_t *pmd = NULL;
8      pte_t *pte = NULL;
9      struct page *page = NULL;
10
11     if(pgd_none(*pgd) || pgd_bad(*pgd))
12         return NULL;
13     p4d = p4d_offset(pgd, addr);
14     if(p4d_none(*p4d) || p4d_bad(*p4d))
15         return NULL;
16     pud = pud_offset(p4d, addr);
17     if(pud_none(*pud) || pud_bad(*pud))
18         return NULL;
19     pmd = pmd_offset(pud, addr);
20     if(pmd_none(*pmd) || pmd_bad(*pmd))
21         return NULL;
22     pte = pte_offset_map(pmd, addr);
23     if(pte_none(*pte) || !pte_present(*pte))
24         return NULL;
25     page = pte_page(*pte);
26     if(!page)
27         return NULL;

```

```

28     pte_unmap(pte);
29     return page;
30 }

```

这里大量用到了定义在 `asm/pgtable.h` 中的函数，后缀为 `offset` 的函数用于根据虚拟地址 `addr` 取出对于层级页表的表项。而后缀为 `none`、`bad`、`present` 等的函数，则被用于检测当前虚拟地址是否能被转换为物理地址（是否在各级页表中有表项）。值得注意的是，第一级页表索引时，需要提供当前虚拟地址区间的内存描述符。通过 `vm_area_struct` 结构体定义包含的 `vm_mm` 项可以找到内存描述符。最后，如果找到了虚拟地址所在页，这个函数将返回指向页描述符的指针。

```

1  static void mtest_find_page(unsigned long addr)
2  {
3      struct vm_area_struct *vma = find_vma(current->mm, addr);
4      struct page *page = NULL;
5      unsigned long phy_addr;
6      if (!vma)
7      {
8          printk(KERN_ERR "Translation not found\n");
9          return;
10     }
11     page = _find_page(vma, addr);
12     if (!page)
13     {
14         printk(KERN_ERR "Translation not found\n");
15         return;
16     }
17     else
18     {
19         phy_addr = page_to_phys(page) | (addr & ~PAGE_MASK);
20         printk(KERN_INFO "VA:0x%lx\tPA:0x%lx\n", addr, phy_addr);
21     }
22 }

```

在以上函数中，首先利用 `find_vma` 获得虚拟地址对应的 `vma` 结构体，也可以判断当前虚拟地址是否在这个进程的虚拟地址区间中。随后我们从 `_find_page` 获得页描述符，判断对于这个虚拟地址，是否存在翻译。如果一切正常，我们将页的基址用 `page_to_phys` 转换为页帧的基址，并将低12位设置为虚拟地址的低12位就可以获得物理地址。

### H3 2.2 结果

我们采用以下两条命令来测试

```

1  echo findpage 0xffffa5cff123 > /proc/mtest
2  echo findpage 0xffffa5d39000 > /proc/mtest

```

根据第一部分的结果，第一条命令中的虚拟地址是存在于进程的虚拟地址区间中的。而第二条命令中的虚拟地址不在任何区间中。因此可以看到如下结果：

```
[ 2589.508790] findpage 0xfffffa5cff123
[ 2589.508792] VA:0xfffffa5cff123      PA:0x11add5123
[ 2630.820244] findpage 0xfffffa5d39000
[ 2630.820246] Translation not found
```

图3. findpage结果

## H2 3. writeval

### H3 3.1 实现

在第二部分的基础上，我们可以很轻松地实现writeval这一功能。值得注意的是，在一个进程中对指定地址写数据时，这个地址是进程的虚拟地址。然而模块的特殊之处在于，它运行在内核态，因此虚拟地址空间是共享的内核态虚拟地址空间。而在64位系统中，内核态的低端虚拟地址空间线性地映射了所有物理地址空间（与32位系统不同）。因此，为了在模块中修改一个运行于用户态的进程的虚拟地址的值，我们需要首先将这个地址转化为物理地址，然后再转换到内核的虚拟地址空间中。代码如下：

```
1 static void mtest_write_val(unsigned long addr, unsigned long val)
2 {
3     struct vm_area_struct *vma = find_vma(current->mm, addr);
4     struct page *page = NULL;
5     unsigned long *kernel_addr;
6     if (!vma)
7     {
8         printk(KERN_ERR "VMA not found\n");
9         return;
10    }
11    if (!(vma->vm_flags & VM_WRITE))
12    {
13        printk(KERN_ERR "Writing permission denied\n");
14        return;
15    }
16    page = _find_page(vma, addr);
17    if (!page)
18    {
19        printk(KERN_ERR "Translation not found\n");
20        return;
21    }
22    else
23    {
24        kernel_addr = (unsigned long *)page_address(page);
25        *kernel_addr = val;
26        printk(KERN_INFO "Write %ld into VA:0x%lx\n", val, addr);
27    }
28 }
```

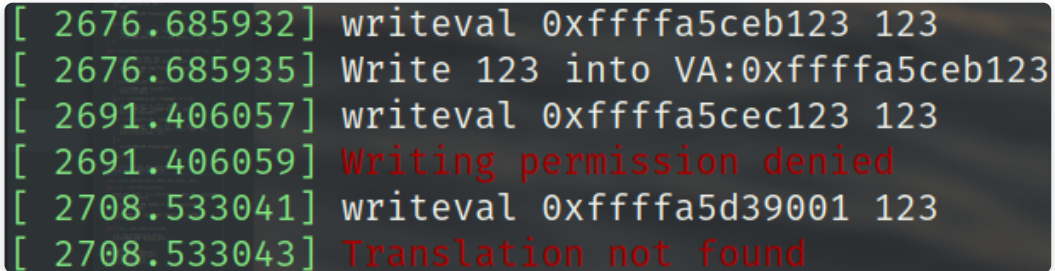
可见，我们首先是找vma，然后判断addr是否存在。然后我们还要根据vm\_area\_struct的vm\_flags成员判断这个地址是否可写。随后调用\_find\_page获得地址所在页。我们还需要调用page\_address()函数找出页对应的内核虚拟地址，然后向其中写入即可。还有另一种方法，则是再进一步将页转化为物理地址，然后调用phys\_to\_virt()函数（定义在asm/memory.h）获得对应的内核虚拟地址再写入。

### H3 3.2 结果

我们采用以下命令来测试

```
1 echo writeval 0xfffffa5ceb123 123 > /proc/mtest  
2 echo writeval 0xfffffa5cec123 123 > /proc/mtest  
3 echo writeval 0xfffffa5d39001 123 > /proc/mtest
```

使用 `dmesg` 命令即可查看结果。参照第一部分的结果，第一条命令的地址存在于一个可写的虚拟地址区间内，因此写入成功。第二条命令的地址处于不可写的区间内，因此不能写入。而第三条命令的地址是不存在任何区间内的，因此也无法写入。



```
[ 2676.685932] writeval 0xfffffa5ceb123 123  
[ 2676.685935] Write 123 into VA:0xfffffa5ceb123  
[ 2691.406057] writeval 0xfffffa5cec123 123  
[ 2691.406059] Writing permission denied  
[ 2708.533041] writeval 0xfffffa5d39001 123  
[ 2708.533043] Translation not found
```

图4. writeval结果

## H2 4. 总结

在这一个Project中，我对Linux内核的内存管理机制有了更加深入的理解，包括页表、虚拟地址、物理地址、内核虚拟地址等等。还对Linux内核代码中的各种数据结构有了一定的了解，这对未来的学习和实践有着巨大的意义。