Project 3: Memory Management

Yesheng Ma

May 17, 2017

Abstract

In this project, we are going to look into the Linux memory management system and try to write a kernel module that deals with virtual memory address translation.

1 Introduction

In this project, we will mainly deal with memory management and we will interact with Linux kernel by a Linux kernel model. This module will create a file `/proc/mtest` and should meet the following requirements:

- listvma: list all the virtual memory area of this process.
- findpage vm: print the physical address related to the input virtual address.
- writeval vm val: write a unsigned long integer to a virtual memory address.

2 Create proc File

As what we have done in previous projects, we should first create a file under proc file system. The core kernel API we use is `proc_create` where a `file_operations` should be specified.

The module initialization part is listed as follows:

```
1  static int __init mtest_init(void)
2  {
3      struct proc_dir_entry *entry;
4
5      printk(KERN_INFO "mtest installed\n");
6      entry = proc_create("mtest", 0666, NULL, &mtest_fops);
7      if (!entry)
8          return -1;
9      return 0;
10 }
```

where the most important part is the file operations struct `mtest_init`:

```
static struct file_operations mtest_fops =
{
    .owner     = THIS_MODULE,
    .open      = mtest_open,
    .read      = seq_read,
    .write     = mtest_write,
    .llseek    = seq_lseek,
    .release   = single_release,
};
```

3 Core Implementation

The core implementation of this project is within the function `mtest_write`. This function mainly deals with three kinds of inputs and further use some helper functions to interact to kernel. The code is shown as follows:

```
if (memcmp(cmd, "findpage", 8)) {
    list_vma();
} else if (memcmp(cmd, "findpage", 8) == 0) {
    if (sscanf(cmd+8, "%lx", &v1) == 1)
        find_page(v1);
} else if (memcmp(cmd, "writeval", 8) == 0) {
    if (sscanf(cmd+8, "%lx %lx", &v1, &v2) == 2)
        write_val(v1, v2);
} else {
    printk(KERN_INFO "Invalid input\n");
}
```

And next we will analyze how to implement the core functions `list_vma`, `find_page`, and `write_val`.

3.1 List VMA

List virtual memory area is quite easy. The task struct of current process can be easily access by `current` macro and we can easily get the memory map by `current->mm->mmap`, the code is shown as follows:

```
down_read(&mm->mmap_sem);
for (vma = mm->mmap; vma; vma = vma->vm_next) {
    if (vma->vm_flags & VM_READ) a[0] = 'r';
    else a[0] = '-';
    if (vma->vm_flags & VM_WRITE) a[1] = 'w';
    else a[1] = '-';
    if (vma->vm_flags & VM_EXEC) a[2] = 'x';
    else a[2] = '-';
```

```
    printk("0x%lx 0x%lx %c%c%c\n", vma->vm_start, vma->vm_end, a[0],
        a[1], a[2]);
}
```

Do remember to hold the lock for memory map here.

3.2 Find page

To find the physical address of a virtual address, we first need to get the translation of the page of the virtual address. First, we can find the virtual memory area of this virtual address by `find_vma(vma, addr)` and we will get a `vm_area_struct` and the page directory is stored in `vma->mm`.

Next we will walk into this page table to figure out the translated address. The first level is page global directory(pgd), the second level is page upper directory, the third level is page middle directory and the last level is page table entry. After we figure out the page table entry, i.e. page frame number, we can get the page related to it using `pfn_to_page`. Then we can get the page address of this page using `page_address` and by concatnating its latter bits, we can get the page's physical address. The code is shown as follows:

```
// seek page table entry
pgd = pgd_offset(mm, addr);
pud = pud_offset(pgd, addr);
pmd = pmd_offset(pud, addr);
pte = pte_offset_map(mm, pmd, addr, &ptl);
page = pfn_to_page(pte_pfn(*pte));
get_page(page);

// get physical address from page info
kern_addr = (unsigned long) page_address(page);
kern_addr += (addr & ~PAGE_MASK);
```

3.3 Write Value

With the help of last question, this one is easier to solve: to directly write to that memory location if that address is writable. The code is shown as follows:

```
// check for permission
if (!(vma->vm_flags & VM_WRITE))
    ...

// write to that location
kern_addr = (unsigned long) page_address(page);
kern_addr += addr & ~PAGE_MASK;
*(unsigned long *) kern_addr = val;
```

4 Experiment

I tested the kernel model on Arch Linux virtual machine and here are the results.

```
[root@kernel mm]# echo listvma > /proc/mtest
[root@kernel mm]# dmesg | tail -10
[48965.750572] 41 0x7f750e212000 0x7f750e214000 rw-
[48965.750575] 42 0x7f750e214000 0x7f750e237000 r-x
[48965.750579] 43 0x7f750e293000 0x7f750e42b000 r--
[48965.750583] 44 0x7f750e42b000 0x7f750e42f000 rw-
[48965.750586] 45 0x7f750e436000 0x7f750e437000 r--
[48965.750590] 46 0x7f750e437000 0x7f750e438000 rw-
[48965.750593] 47 0x7f750e438000 0x7f750e439000 rw-
[48965.750597] 48 0x7ffd13461000 0x7ffd13483000 rw-
[48965.750600] 49 0x7ffd13487000 0x7ffd13489000 r--
[48965.750604] 50 0x7ffd13489000 0x7ffd1348b000 r-x
```

Figure 1: List virtual memory area

```
[root@kernel mm]# dmesg  | tail -1
[49144.824177] Translate 0x7ffd13489000 to kernel address 0xffff88000197f000
[root@kernel mm]#
```

Figure 2: Find virtual memory page

```
[root@kernel mm]# echo "writeval 0x7f750ddbe000 0x123" > /proc/mtest
[root@kernel mm]# dmesg | tail -1
[49689.129428] 0x123 written to address 0xffff880016248000
[root@kernel mm]#
```

Figure 3: Write to virtual memory address

5 Conclusion

In this project, I really leaned a lot about Linux kernel memory management subsystem and a lot about Linux kernel programming APIs. I find memory management is extremely important for model operating systems and I will learn more about this in my free time.

Acknowledgement

Thanks Prof. Chen for guidance on Linux kernel and TAs for their hard work.