Lab 3 : Memory Management

实 验 报 告

刘子涵　518021910690

【1】实验要求

写一个模块 mtest，当模块加载时，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 `translation not found`
- `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

- 实验平台：华为云弹性云服务器（2vCPUs | 4GiB | kc1.large.2 | Ubuntu 18.04 server 64bit with ARM）
- Linux 内核版本：5.5.11
- GCC 版本：7.4.0

【3】实验思路及具体实现

1. Linux 内存描述符

一个进程拥有的内存描述符 `mm_struct` 在 `include/linux/mm_types.h` 第 370 行中定义，它抽象并描述了 Linux 视角下管理进程虚拟地址空间的所有信息。以下列举一些字段：

```
struct mm_struct {
    struct vm_area_struct *mmap; // 虚拟区间（VMA）有序链表，按照区间起始地址递增方式组织
    struct rb_root mm_rb; // VMA 红黑树根节点，将进程所有的 VMA 记录到红黑树中，以提高查找效率
    pgd_t *pgd; // 页全局目录
    int map_count; // VMA 数量
    struct rw_semaphore mmap_sem; // 读写信号量
    // ...
};
```

2. Linux 虚拟内存区间组织与读写权限（`listvma` 实现）

由上文可知，Linux 虚拟内存区间（VMA）按照区间起始地址递增方式组织成有序链表的形式，指向该链表头结点的指针是 `struct vm_area_struct *mmap`。找到 `vm_area_struct` 的定义，在同一文件的第 292 行。以下列举一些字段：

```
struct vm_area_struct {
    unsigned long vm_start; // VMA 起始地址
    unsigned long vm_end; // VMA 终止地址（不包含本身）
    struct vm_area_struct *vm_next, *vm_prev; // 链表的后一个/前一个结点
    struct rb_node vm_rb; // 对应红黑树节点
    unsigned long vm_flags; // 权限，在 mm.h 中定义
    // ...
};
```

可见，实现 `listvma` 功能非常简单，只需要遍历一次这个链表即可。由于虚拟内存区间属于系统临界区，在执行读操作时需要对临界区加读锁，这一操作可以利用 `mm_struct` 中定义的读写信号量来实现。另外，虚拟内存区间的读写权限在 `include/linux/mm.h` 中第 249 行定义。通过检查 `vm_flags` 中的低 3 位即可判断区间的读、写、执行权限。

```
#define VM_READ     0x00000001
#define VM_WRITE    0x00000002
#define VM_EXEC     0x00000004
```

`listvma` 代码实现细节见【附录 A】。

3. Linux 分页机制、虚拟地址到物理地址的转换（`findpage` 实现）

Linux 将线性地址分为：

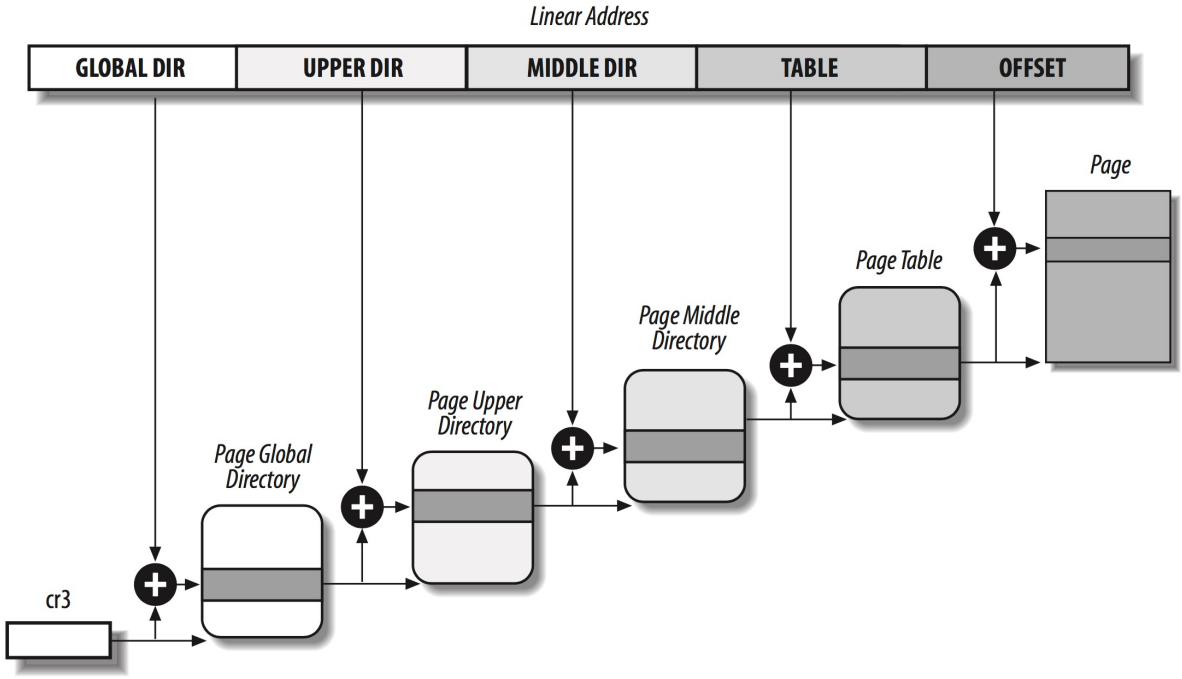
页全局目录 pgd（9）	页上级目录 pud（9）	页中级目录 pmd（9）	页表 pte（9）	偏移量 offset（12）

ARM64 架构下的页表类型定义在文件 `arch/arm64/include/asm/pgtable_types.h` 中：

```
typedef u64 pteval_t;
typedef u64 pmdval_t;
typedef u64 pudval_t;
typedef u64 pgdval_t;
// ...
typedef struct { pteval_t pte; } pte_t;
#define pte_val(x) ((x).pte)
// ...
```

可见，该版本下的 Linux 分页机制仍为 4 级（不存在 p4d）。如下图，根据该地址结构容易计算出某一虚拟地址对应的物理地址，计算过程即：

- 当前进程的 `mm_struct` 中的 `pgd` 字段记录了 pgd 基址
- pgd 基址 + pgd → pud 基址
- pud 基址 + pud → pmd 基址
- pmd 基址 + pmd → pte基址
- pte 基址 + pte → page 基址
- page 基址 + offset → 物理地址



而计算偏移的过程，可以直接利用 `arch/arm64/include/asm/pgtable.h` 定义的宏来实现，比如通过 pgd 基址和 pgd 计算 pud 基址的过程可以利用宏 `pgd_offset(mm, addr)` 实现，返回 `pgd_t*` 类型的指针。该宏的调用过程如下，可见其计算过程是将 pgd 基址加上 `pgd_index(addr)`，后者即偏移量，是通过将虚拟地址右移 39 位，然后取低 9 位。

```
// pgd_offset(mm, addr)
#define pgd_index(addr)     (((addr) >> PGDIR_SHIFT) & (PTRS_PER_PGD - 1))
#define pgd_offset_raw(pgd, addr)   ((pgd) + pgd_index(addr))
#define pgd_offset(mm, addr)    (pgd_offset_raw((mm)->pgd, (addr)))
```

其它几步的计算类似，利用如下的宏可以实现：

```
// pud_offset(dir, addr)
#define pud_index(addr)     (((addr) >> PUD_SHIFT) & (PTRS_PER_PUD - 1))
#define pud_offset_phys(dir, addr)  (pgd_page_paddr(READ_ONCE(*(dir))) +
pud_index(addr) * sizeof(pud_t))
#define pud_offset(dir, addr)   ((pud_t *)__va(pud_offset_phys((dir), (addr))))

// pmd_offset(dir, addr)
#define pmd_index(addr)     (((addr) >> PMD_SHIFT) & (PTRS_PER_PMD - 1))
#define pmd_offset_phys(dir, addr)  (pud_page_paddr(READ_ONCE(*(dir))) +
pmd_index(addr) * sizeof(pmd_t))
#define pmd_offset(dir, addr)   ((pmd_t *)__va(pmd_offset_phys((dir), (addr))))

// pte_offset_kernel(dir, addr)
#define pte_index(addr)     (((addr) >> PAGE_SHIFT) & (PTRS_PER_PTE - 1))
#define pte_offset_phys(dir,addr)   (pmd_page_paddr(READ_ONCE(*(dir))) +
pte_index(addr) * sizeof(pte_t))
#define pte_offset_kernel(dir,addr) ((pte_t *)__va(pte_offset_phys((dir),
(addr))))
```

最后，利用宏 `pte_page()` 将计算得到的 page 基址转换为页描述符 `struct page *curr_page = pte_page(*pte)`。通过虚拟地址计算页描述符的函数实现细节见【附录 B】，定义该函数是考虑到实现 `writeval` 也需要实现该步骤。注意到，不是所有的虚拟地址都存在对应的 pgd、pud、pmd、pte，所以需要检查计算得到的指针是否有效，同样可以直接借助 `pgtable.h` 定义的宏来实现，比如 `pgd_none()` 和 `pgd_bad()` 宏检查 pgd。

基于上一步函数 `_find_page()` 返回的页描述符，可以方便计算出物理地址，只需要利用宏 `page_to_phys()` 计算出页的物理地址（需要使用 `PAGE_MASK` 过滤相应位）；另外，虚拟地址和物理地址的偏移量相同，直接取虚拟地址的低 12 位（使用 `~PAGE_MASK` 过滤相应位）即可得到页内偏移量。将页起始物理地址和页内偏移合并在一起，即得到最终的物理地址。实现细节见【附录 C】。

4. 在特定虚拟地址上写数据（ `writeval` 实现）

首先，使用 `find_vma()` 函数查找第一个满足 `vaddr<vm_end` 的 VMA，该函数定义在 `mm.h` 中。需要检查该 VMA 是否有效（检查该 VMA 是否为空，检查该 VMA 起始地址是否大于 `vaddr` ），无效则输出 `invalid vma` 并直接返回；还需要检查该页是否可写（检查该 VMA 的 `vm_flags` ），不可写则输出 `unwritable page` 并直接返回。

然后，利用 `_find_page()` 函数得到当前虚拟地址的页描述符，也需要检查该页是否存在。

接着，利用 `page_address()` 函数计算该页的虚拟地址，并将其转换为 `unsigned long*` 以便写入一个 `unsigned long` 类型的值，加上页内偏移量即得到最终的内核虚拟地址，直接写入即可。

函数实现细节见【附录 D】。

5. 模块与 proc 文件接口实现

三种功能由模块统一整合并加载，通过向 proc 文件写入不同的参数以实现不同的功能，将写入的参数转换为特定类型再调用相应的功能函数。实现细节见【附录 E】。

【4】实验测试及效果截图

1. 模块编译

编写 Makefile 对模块进行编译：

```
obj-m:=mtest.o
KDIR:=/lib/modules/$(shell uname -r)/build
PWD:=$(shell pwd)
all:
    make -C $(KDIR) M=$(PWD) modules
clean:
    make -C $(KDIR) M=$(PWD) clean
```

2. 测试命令

```
make && insmod mtest.ko
echo listvma > /proc/mtest
echo findpage 0x... > /proc/mtest
echo writeval 0x... 0x... > /proc/mtest
rmmod mtest
```

3. 测试截图与分析

- **listvma**

  使用 `dmesg` 查看内核打印的当前进程所有 VMA 的地址区间和权限信息如下图。

```
root@ecs-lzh:~/lab3# echo listvma > /proc/mtest
root@ecs-lzh:~/lab3# dmesg | tail -20
[  212.327991] VMA 0xffffaf5c9000 - 0xffffaf5ca000
[  212.327991] r
[  212.327991] -
[  212.327992] x

[  212.327993] VMA 0xffffaf5ca000 - 0xffffaf5cb000
[  212.327994] r
[  212.327994] -
[  212.327994] -

[  212.327996] VMA 0xffffaf5cb000 - 0xffffaf5cd000
[  212.327996] r
[  212.327996] w
[  212.327997] -

[  212.327998] VMA 0xfffffb209000 - 0xfffffb22a000
[  212.327998] r
[  212.327999] w
[  212.327999] -
```

- **findpage**
  - 测试有效的虚拟地址：`0xffffaf5cb000`（上图倒数第二个区间起始地址），得到其物理地址 `0x12ef9a000`；
  - 测试无效的虚拟地址：`0xffffaf5cd0ab`，输出 `translation not found`，原因是页起始地址 `pte` 无效。

```
root@ecs-lzh:~/lab3# echo findpage 0xffffaf5cb000 > /proc/mtest
root@ecs-lzh:~/lab3# dmesg | tail -5
[  212.327998] r
[  212.327999] w
[  212.327999] -

[  361.003515] vma 0xffffaf5cb000 -> pma 0x12ef9a000
root@ecs-lzh:~/lab3#
root@ecs-lzh:~/lab3# echo findpage 0xffffaf5cd0ab > /proc/mtest
root@ecs-lzh:~/lab3# dmesg | tail -5
[  212.327999] -

[  361.003515] vma 0xffffaf5cb000 -> pma 0x12ef9a000
[  410.037626] [pte] not available
[  410.037627] translation not found
```

- **writeval**
  - 向一个有效虚拟地址 `0xffffaf5cb0ab` 写入 `0x123`，得到写入结果 `written 0x123 to address 0xffff0000eef9a558`；
  - 向一个不可写页中的虚拟地址 `0xffffaf5ca0ab` 写入 `0x123`，得到结果 `unwritable page`；
  - 向一个本身无效的虚拟地址 `0xffffaf5cd0ab` 写入 `0x123`，得到结果 `unexisted page`，原因是页起始地址 `pte` 无效。

```
root@ecs-lzh:~/lab3# echo writeval 0xffffaf5cb0ab 0x123 > /proc/mtest
root@ecs-lzh:~/lab3# dmesg | tail -5

[  361.003515] vma 0xffffaf5cb000 -> pma 0x12ef9a000
[  410.037626] [pte] not available
[  410.037627] translation not found
[  602.049146] written 0x123 to address 0xffff0000eef9a558
root@ecs-lzh:~/lab3#
root@ecs-lzh:~/lab3# echo writeval 0xffffaf5ca0ab 0x123 > /proc/mtest
root@ecs-lzh:~/lab3# dmesg | tail -5
[  361.003515] vma 0xffffaf5cb000 -> pma 0x12ef9a000
[  410.037626] [pte] not available
[  410.037627] translation not found
[  602.049146] written 0x123 to address 0xffff0000eef9a558
[  651.218295] unwritable page
root@ecs-lzh:~/lab3#
root@ecs-lzh:~/lab3# echo writeval 0xffffaf5cd0ab 0x123 > /proc/mtest
root@ecs-lzh:~/lab3# dmesg | tail -5
[  410.037627] translation not found
[  602.049146] written 0x123 to address 0xffff0000eef9a558
[  651.218295] unwritable page
[  683.334740] [pte] not available
[  683.334742] unexisted page
```

【5】实验心得

本次实验基于 Linux 内核内存管理的理论课程，通过实践深入理解了 Linux 的分页机制和寻址过程，提高了我阅读分析大规模系统软件源码的能力和调试能力。本次实验难度较大，主要是需要在实验过程中熟悉很多页表机制的 API，在 `pgtable.h`、`mm.h` 等中定义了许多功能相似的宏和函数，如何找到合适的函数为自己所用是一大难点。在这一过程中我花费了大量时间去尝试各种 API，期间出现多次写入后系统直接崩溃，终端窗口关闭的情况，只有重新开机继续做。阅读源码过程中，我会着重看注释和 API 的名称，这有助于我找到适合的 API，也提醒我自己在编程的时候需要注意命令和注释等命名规范的问题。总之，本次实验收获不少！感谢老师和助教的指导！

【附 录】

代码 A：`mtest_list_vma()` 函数

```c
/* Print all vma of the current process */
static void mtest_list_vma(void) {
    struct mm_struct *mm = current->mm;
    struct vm_area_struct *vma = mm->mmap;

    down_read(&(mm->mmap_sem)); // lock the read critical section

    // traverse list of VMAs
    while (vma) {
        printk("VMA 0x%lx - 0x%lx\n", vma->vm_start, vma->vm_end);
        // permission flags in `mm.h`
        (vma->vm_flags & VM_READ) ? printk("r\n") : printk("-\n");
        (vma->vm_flags & VM_WRITE) ? printk("w\n") : printk("-\n");
        (vma->vm_flags & VM_EXEC) ? printk("x\n") : printk("-\n");
        printk("\n");
        vma = vma->vm_next;
    }
```

```
    up_read(&(mm->mmap_sem)); // unlock the read critical section
}
```

代码 B：`_find_page()` 函数

```
/* find page of va */
static struct page *_find_page(unsigned long vaddr) {
    struct mm_struct *mm = current->mm;
    struct page *curr_page;

    pgd_t *pgd;
    pud_t *pud;
    pmd_t *pmd;
    pte_t *pte;

    // walk the page table
    // 1. get [page global directory, pgd]
    pgd = pgd_offset(mm, vaddr);
        // printk("pgd: %llx\n", pgd_val(*pgd));
    if (pgd_none(*pgd) || pgd_bad(*pgd)) {
        printk("[pgd] not available\n");
        return NULL;
    }
    // 2. get [page upper directory, pud]
    pud = pud_offset(pgd, vaddr);
        // printk("pud: %llx\n", pud_val(*pud));
    if (pud_none(*pud) || pud_bad(*pud)) {
        printk("[pud] not available\n");
        return NULL;
    }
    // 3. get [page middle directory, pmd]
    pmd = pmd_offset(pud, vaddr);
        // printk("pmd: %llx\n", pmd_val(*pmd));
    if (pmd_none(*pmd) || pmd_bad(*pmd)) {
        printk("[pmd] not available\n");
        return NULL;
    }
    // 4. get [page table entry, pte]
    pte = pte_offset_kernel(pmd, vaddr);
        // printk("pte: %llx\n", pte_val(*pte));
    if (pte_none(*pte)) {
        printk("[pte] not available\n");
        return NULL;
    }

    curr_page = pte_page(*pte);
    return curr_page;
}
```

代码 C：`mtest_find_page()` 函数

```
/* Find va->pa translation */
static void mtest_find_page(unsigned long vaddr) {
    unsigned long paddr;
    unsigned long page_addr;
```

```
    unsigned long page_offset;

    // get current page of vaddr
    struct page *curr_page = _find_page(vaddr);

    if (!curr_page) {
        printk("translation not found\n");
        return;
    }

    page_addr = page_to_phys(curr_page) & PAGE_MASK;
    page_offset = vaddr & (~PAGE_MASK);
    paddr = page_addr | page_offset;

    printk("vma 0x%lx -> pma 0x%lx\n", vaddr, paddr);
}
```

代码 D：`mtest_write_val()` 函数

```
/* Write val to the specified address */
static void mtest_write_val(unsigned long vaddr, unsigned long val) {
    // look up the first VMA which statisfies vaddr < vm_end, NULL if none
    struct vm_area_struct *vma = find_vma(current->mm, vaddr);
    // get current page of vaddr
    struct page *curr_page = _find_page(vaddr);

    // whether the page is existed
    if (!curr_page) {
        printk("unexisted page\n");
        return;
    }

    // whether the vma is valid
    if (!vma || vma->vm_start > vaddr) {
        printk("invalid vma\n");
        return;
    }

    // whether the page is writable
    if (!(vma->vm_flags & VM_WRITE)) {
        printk("unwritable page\n");
        return;
    }

    // write value
    unsigned long *kernel_addr;
    kernel_addr = (unsigned long*)page_address(curr_page);
    kernel_addr += vaddr & (~PAGE_MASK);
    *kernel_addr = val;
    printk("written 0x%lx to address 0x%lx\n", val, (unsigned long)kernel_addr);
}
```

代码 E：模块与 proc 文件接口

```c
/* proc write interface */
static ssize_t mtest_proc_write(struct file *file,
                                const char __user *ubuf,
                                size_t count,
                                loff_t *ppos) {
    char buf[BUFSIZE];
    char data[BUFSIZE];
    unsigned long addr, val;
    unsigned short offset;

    if (*ppos > 0 || count > BUFSIZE)
        return -EFAULT;
    if (copy_from_user(buf, ubuf, count))
        return -EFAULT;
    sscanf(buf, "%s", data);

    if (!strcmp(data, "listvma")) {
        /* listvma */
        mtest_list_vma();
    } else if (!strcmp(data, "findpage")) {
        /* findpage */
        offset = 9;
        sscanf(buf + offset, "%s", data);
        kstrtoul(data, 16, &addr);
        mtest_find_page(addr);
    } else if (!strcmp(data, "writeval")) {
        /* writeval */
        offset = 9;
        sscanf(buf + offset, "%s", data);
        kstrtoul(data, 16, &addr);
        while (*(buf + offset) != ' ') offset ++;
        offset ++;
        sscanf(buf + offset, "%s", data);
        kstrtoul(data, 16, &val);
        mtest_write_val(addr, val);
    }

    *ppos = strlen(buf);
    return *ppos;
}


/* proc file_operations struct */
static struct file_operations proc_mtest_operations = {
    .owner = THIS_MODULE,
    .write = mtest_proc_write
};


static int __init mtest_init(void) {
    mtest_proc_entry = proc_create("mtest", 0666, NULL, &proc_mtest_operations);
    return 0;
}
```

```c
static void __exit mtest_exit(void) {
    proc_remove(mtest_proc_entry);
}

MODULE_LICENSE("GPL");
MODULE_DESCRIPTION("Memory Management Lab Test Module");
MODULE_AUTHOR("Zihan Liu");

module_init(mtest_init);
module_exit(mtest_exit);
```