МИНИСТЕРСТВО ОБРАЗОВАНИЯ И НАУКИ РФ

Федеральное государственное автономное
образовательное учреждение высшего образования
«Национальный исследовательский университет ИТМО»

Факультет программной инженерии и компьютерной техники

ЛАБОРАТОРНАЯ РАБОТА №2

по дисциплине

“Операционные системы”

Вариант: ioctl: page, syscall_info

Студент:

Воробьев К.О.

Группа P33302

Преподаватель:

Осипов С. В.

Санкт-Петербург, 2022

Задание:

Разработать комплекс программ на пользовательском уровне и уровне ярда, который собирает информацию на стороне ядра и передает информацию на уровень пользователя, и выводит ее в удобном для чтения человеком виде. Программа на уровне пользователя получает на вход аргумент(ы) командной строки (не адрес!), позволяющие идентифицировать из системных таблиц необходимый путь до целевой структуры, осуществляет передачу на уровень ядра, получает информацию из данной структуры и распечатывает структуру в стандартный вывод. Загружаемый модуль ядра принимает запрос через указанный в задании интерфейс, определяет путь до целевой структуры по переданному запросу и возвращает результат на уровень пользователя.

Интерфейс передачи между программой пользователя и ядром и целевая структура задается преподавателем. Интерфейс передачи может быть один из следующих:

1. syscall - интерфейс системных вызовов.
2. ioctl - передача параметров через управляющий вызов к файлу/устройству.
3. procfs - файловая система /proc, передача параметров через запись в файл.
4. debugfs - отладочная файловая система /sys/kernel/debug, передача параметров через запись в файл.

Целевая структура может быть задана двумя способами:

1. Именем структуры в заголовочных файлах Linux
2. Файлом в каталоге /proc. В этом случае необходимо определить целевую структуру по пути файла в /proc и выводимым данным.

The struct page is defined in include/linux/mm_types.h as:

```
struct page {
        unsigned long flags;            /* Atomic flags, some possibly
                                         * updated asynchronously */
        /*
         * Five words (20/40 bytes) are available in this union.
         * WARNING: bit 0 of the first word is used for PageTail(). That
         * means the other users of this union MUST NOT use the bit to
         * avoid collision and false-positive PageTail().
         */
        union {
                struct {        /* Page cache and anonymous pages */
                        /**
                         * @lru: Pageout list, eg. active_list protected by
                         * lruvec->lru_lock.  Sometimes used as a generic list
                         * by the page owner.
                         */
                        union {
                                struct list_head lru;

                                /* Or, for the Unevictable "LRU list" slot */
                                struct {
                                        /* Always even, to negate PageTail */
                                        void *__filler;
                                        /* Count page's or folio's mlocks */
                                        unsigned int mlock_count;
                                };

                                /* Or, free page */
                                struct list_head buddy_list;
                                struct list_head pcp_list;
                        };
                        /* See page-flags.h for PAGE_MAPPING_FLAGS */
                        struct address_space *mapping;
                        pgoff_t index;          /* Our offset within mapping. */
                        /**
                         * @private: Mapping-private opaque data.
                         * Usually used for buffer_heads if PagePrivate.
                         * Used for swp_entry_t if PageSwapCache.
                         * Indicates order in the buddy system if PageBuddy.
                         */
                        unsigned long private;
```

```
                };
                struct {        /* page_pool used by netstack */
                        /**
                         * @pp_magic: magic value to avoid recycling non
                         * page_pool allocated pages.
                         */
                        unsigned long pp_magic;
                        struct page_pool *pp;
                        unsigned long _pp_mapping_pad;
                        unsigned long dma_addr;
                        union {
                                /**
                                 * dma_addr_upper: might require a 64-bit
                                 * value on 32-bit architectures.
                                 */
                                unsigned long dma_addr_upper;
                                /**
                                 * For frag page support, not supported in
                                 * 32-bit architectures with 64-bit DMA.
                                 */
                                atomic_long_t pp_frag_count;
                        };
                };
                struct {        /* Tail pages of compound page */
                        unsigned long compound_head;    /* Bit zero is set */

                        /* First tail page only */
                        unsigned char compound_dtor;
                        unsigned char compound_order;
                        atomic_t compound_mapcount;
                        atomic_t compound_pincount;
#ifdef CONFIG_64BIT
                        unsigned int compound_nr; /* 1 << compound_order */
#endif
                };
                struct {        /* Second tail page of compound page */
                        unsigned long _compound_pad_1;  /* compound_head */
                        unsigned long _compound_pad_2;
                        /* For both global and memcg */
                        struct list_head deferred_list;
                };
                struct {        /* Page table pages */
                        unsigned long _pt_pad_1;        /* compound_head */
                        pgtable_t pmd_huge_pte; /* protected by page->ptl */
                        unsigned long _pt_pad_2;        /* mapping */
                        union {
                                struct mm_struct *pt_mm; /* x86 pgds only */
                                atomic_t pt_frag_refcount; /* powerpc */
                        };
#if ALLOC_SPLIT_PTLOCKS
                        spinlock_t *ptl;
#else
                        spinlock_t ptl;
#endif
                };
                struct {        /* ZONE_DEVICE pages */
                        /** @pgmap: Points to the hosting device page map. */
                        struct dev_pagemap *pgmap;
                        void *zone_device_data;
                        /*
                         * ZONE_DEVICE private pages are counted as being
                         * mapped so the next 3 words hold the mapping, index,
                         * and private fields from the source anonymous or
                         * page cache page while the page is migrated to device
                         * private memory.
                         * ZONE_DEVICE MEMORY_DEVICE_FS_DAX pages also
                         * use the mapping, index, and private fields when
                         * pmem backed DAX files are mapped.
                         */
                };

                /** @rcu_head: You can use this to free a page by RCU. */
                struct rcu_head rcu_head;
        };

        union {         /* This union is 4 bytes in size. */
                /*
                 * If the page can be mapped to userspace, encodes the number
                 * of times this page is referenced by a page table.
                 */
                atomic_t _mapcount;
```

```
                /*
                 * If the page is neither PageSlab nor mappable to userspace,
                 * the value stored here may help determine what this page
                 * is used for.  See page-flags.h for a list of page types
                 * which are currently stored here.
                 */
                unsigned int page_type;
        };

        /* Usage count. *DO NOT USE DIRECTLY*. See page_ref.h */
        atomic_t _refcount;

#ifdef CONFIG_MEMCG
        unsigned long memcg_data;
#endif

        /*
         * On machines where all RAM is mapped into kernel address space,
         * we can simply calculate the virtual address. On machines with
         * highmem some memory is mapped into kernel virtual memory
         * dynamically, so we need a place to store that address.
         * Note that this field could be 16 bits on x86 ... ;)
         *
         * Architectures with slow multiplication can define
         * WANT_PAGE_VIRTUAL in asm/page.h
         */
#if defined(WANT_PAGE_VIRTUAL)
        void *virtual;                  /* Kernel virtual address (NULL if
                                           not kmapped, ie. highmem) */
#endif /* WANT_PAGE_VIRTUAL */

#ifdef LAST_CPUPID_NOT_IN_PAGE_FLAGS
        int _last_cpupid;
#endif
} _struct_page_alignment;
```

The struct syscall_info is defined in include/linux/ptrace.h as:

```
struct syscall_info {
        __u64                   sp;
        struct seccomp_data     data;
};
```

Выполнение:

/cmd_mode.h

```
#define WR_PID _IOW('a','a', int32_t*)
#define RD_MY_PAGE _IOR('a','c', struct my_page*)
#define RD_MY_SYSCALL_INFO _IOR('a','d', struct my_syscall_info*)

struct my_seccomp_data {
   int nr;
   __u32 arch;
   __u64 instruction_pointer;
   __u64 args[6];
};

struct my_syscall_info {
   __u64        sp;
   struct my_seccomp_data data;
};

struct my_page {
   unsigned long flags;
   void* mapping;
   long vm_address;
};
```

/core_mod.c

```
#include <linux/init.h>
#include <linux/module.h>
#include <linux/kernel.h>
#include <linux/ioctl.h>
#include <linux/ptrace.h>
#include <linux/pid.h>
```

```c
#include <linux/netdevice.h>
#include <asm/syscall.h>
#include <linux/sched.h>
#include <linux/namei.h>
#include <linux/mm_types.h>
#include <asm/page.h>

#include "cmd_mode.h"

MODULE_LICENSE("GPL");
MODULE_AUTHOR("Kirill Vorobyev");
MODULE_DESCRIPTION("My ioctl driver");
MODULE_VERSION("1.0");

const int MYMAJOR = 28;

int pid = 0;

static void print_my_page(const struct my_page* mp) {
    printk("\nPage:\n");
    printk("flags: %lu\n", mp->flags);
    printk("Virtual address: %ld, Page address: %p\n", mp->vm_address, mp-
>mapping);
}

static void print_my_syscall_info(const struct my_syscall_info* msi) {
    printk("\nSyscall_info:\n");
    printk("sp %lld\n", msi->sp);
    printk("nr %d\n", msi->data.nr);
    printk("instruction_pointer %lld\n", msi->data.instruction_pointer);
    printk("arg %d  0x%08X\n", 0, (unsigned int) msi->data.args[0]);
    printk("arg %d  0x%08X\n", 1, (unsigned int) msi->data.args[1]);
    printk("arg %d  0x%08X\n", 2, (unsigned int) msi->data.args[2]);
    printk("arg %d  0x%08X\n", 3, (unsigned int) msi->data.args[3]);
    printk("arg %d  0x%08X\n", 4, (unsigned int) msi->data.args[4]);
    printk("arg %d  0x%08X\n", 5, (unsigned int) msi->data.args[5]);
}

static struct page* get_page_by_mm_and_address(struct mm_struct* mm, long
address) {
    pgd_t* pgd;
    p4d_t* p4d;
    pud_t* pud;
    pmd_t* pmd;
    pte_t* pte;
    struct page* page;
    pgd = pgd_offset(mm, address);
    if (!pgd_present(*pgd)) return NULL;
    p4d = p4d_offset(pgd, address);
    if (!p4d_present(*p4d)) return NULL;
    pud = pud_offset(p4d, address);
    if (!pud_present(*pud)) return NULL;
    pmd = pmd_offset(pud, address);
    if (!pmd_present(*pmd)) return NULL;
    pte = pte_offset_map(pmd, address);
    if (!pte_present(*pte)) return NULL;
    page = pte_page(*pte);
    return page;
}

// This function will be called when we write IOCTL on the Device file
static long driver_ioctl(struct file *file, unsigned int cmd, unsigned long
arg) {
    struct my_page mp;
```

```
    struct my_syscall_info msi;
    struct page* page;
    struct task_struct* task;
    struct mm_struct* mm;
    struct vm_area_struct* vm;
    switch(cmd) {
        case WR_PID:
            if(copy_from_user(&pid, (int*) arg, sizeof(pid))) pr_err("Data write
error!\n");
            pr_info("Pid = %d\n", pid);
            break;
        case RD_MY_PAGE:
            task = get_pid_task(find_get_pid(pid), PIDTYPE_PID);
            mm = task->mm;
            vm = mm->mmap;
            if (mm == NULL) {
                printk(KERN_INFO "Task struct hasn't mm\n");
            }
            long address = vm->vm_start;
            long end = vm->vm_end;
            while (address <= end) {
                page = get_page_by_mm_and_address(mm, address);
                if (page != NULL) {
                    mp.flags = page->flags;
                    mp.mapping = (void*)page->mapping;
                    mp.vm_address = address;
                    break;
                }
                address += PAGE_SIZE;
            }
            print_my_page(&mp);
            if(copy_to_user((struct my_page*) arg, &mp, sizeof(struct my_page)))
{
                printk(KERN_INFO "Data read error!\n");
            }
            break;
        case RD_MY_SYSCALL_INFO:
            task = get_pid_task(find_get_pid(pid), PIDTYPE_PID);
            unsigned long args[6] = {};
            struct pt_regs* regs = task_pt_regs(task);
            msi.sp = user_stack_pointer(regs);
            msi.data.instruction_pointer = instruction_pointer(regs);
            msi.data.nr = syscall_get_nr(task, regs);
            if (msi.data.nr != -1L) syscall_get_arguments(task, regs, args);
            print_my_syscall_info(&msi);
            if(copy_to_user((struct my_syscall_info*) arg, &msi, sizeof(struct
my_syscall_info))) printk(KERN_INFO "Data read error!\n");
            break;
        default:
            pr_info("Command not found!");
            break;
    }
    return 0;
}

static struct file_operations fops = {
    .owner          = THIS_MODULE,
    .unlocked_ioctl = driver_ioctl,
};

static int __init ioctl_core_init(void) {
    printk(KERN_INFO "Core mode is started, hello, world!\n");
    int retval;
    retval = register_chrdev(MYMAJOR, "my_driver", &fops);
```

```
    if (0 == retval) {
        printk("my_driver device number Major:%d , Minor:%d\n", MYMAJOR, 0);
    } else if (retval > 0) {
        printk("my_driver device number Major:%d , Minor:%d\n", retval >> 20,
retval & 0xffff);
    } else {
        printk("Couldn't register device number!\n");
        return -1;
    }
    return 0;
}

static void __exit ioctl_core_exit(void) {
    unregister_chrdev(MYMAJOR, "my_driver");
    printk(KERN_INFO "Core mode is finished, goodbye!\n");
}

module_init(ioctl_core_init);
module_exit(ioctl_core_exit);
```

/user_mode.c

```
#include <sys/ioctl.h>
#include <stdio.h>
#include <stdint.h>
#include <stdlib.h>
#include <fcntl.h>
#include <malloc.h>
#include <unistd.h>
#include <linux/ptrace.h>

#include "cmd_mode.h"

static void print_line(void) {
    printf("_____________________\n");
}

static void print_my_syscall_info(const struct my_syscall_info* si) {
    printf("\nSyscall_info:\n\n");
    printf("sp %lld\n", si->sp);
    printf("nr %d\n", si->data.nr);
    printf("instruction_pointer %lld\n", si->data.instruction_pointer);
    for (int i = 0; i < 6; i++) printf("arg %d  0x%08X\n", i, (unsigned int)
si->data.args[i]);
}

static void print_page(const struct my_page* mp) {
    printf("\nPage:\n\n");
    printf("flags: %ld\n", mp->flags);
    printf("Virtual address: %ld\n", mp->vm_address);
    printf("Page address: %p\n", mp->mapping);
}

int main(int argc, char *argv[]) {
    int fd;
    if(argc < 2) {
        printf("The program needs an argument - a pid!\n");
        return -1;
    }
    int32_t pid = atoi(argv[1]);
    if (pid < 1) {
        printf("Pid can be only greater than 0\n");
        return -1;
    }
```

```c
    printf("\nOpening a driver...\n");
    fd = open("/dev/etc_device", O_WRONLY);
    if(fd < 0) {
        printf("Cannot open device file:(\n");
        return 0;
    }
    struct my_page mp;
    struct my_syscall_info msi;
    // Writing a pid to driver
    ioctl(fd, WR_PID, (int32_t*) &pid);
    // Reading my_page from driver
    ioctl(fd, RD_MY_PAGE, &mp);
    print_line();
    print_page(&mp);
    // Reading my_syscall_info from driver
    ioctl(fd, RD_MY_SYSCALL_INFO, &msi);
    print_line();
    print_my_syscall_info(&msi);
    print_line();
    printf("Closing a driver...\n");
    close(fd);
}
```

```
kirill@kirill-VirtualBox:~/src/ioctl_lab$ sudo ./user_mode 3069

Opening a driver...
________________________

Page:

flags: 4503598553759798
Virtual address: 94138079924224
Page address: 0xffff8efe3141b5c0
________________________

Syscall_info:

sp 140726827939040
nr -237
instruction_pointer 94138079925758
arg 0  0xC4D8BE40
arg 1  0x834E7768
arg 2  0x7DA2FFB8
arg 3  0x35232F80
arg 4  0x1C4F3200
arg 5  0x7DA2FE80
________________________
Closing a driver...
```

Вывод:

Во время выполнения лабораторной работы я углубился в работу ядра linux. Написал собственный модуль ядра и клиентское приложение, для работы с этим модулем, реализовав общение между ними с помощью ioctl.