Национальный исследовательский университет
компьютерных технологий, механики и оптики

Факультет ПИиКТ

Операционные системы
Лабораторная работа №2

Вариант: ioctl(memblock, pci_dev)

Работу выполнил: Кулаков Н. В.
Группа: P33312
Преподаватель: Осипов С. В.

Санкт-Петербург
2022

1 Текст задания

Разработать комплекс программ на пользовательском уровне и уровне ядра, 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:

- ioctl - передача параметров через управляющий вызов к файлу/устройству.

2 Выполнение

2.1 Листинг модуля

```c
#ifndef __IOCTL_H
#define __IOCTL_H

#include <linux/ioctl.h>
#include <linux/types.h>


struct user_pci_dev {
  unsigned int  devfn;    /* Encoded device & function index */
  unsigned short  vendor;
  unsigned short  device;
  unsigned short  subsystem_vendor;
  unsigned short  subsystem_device;
};

struct user_memblock {
  _Bool bottom_up;  /* is bottom up direction? */
  uint64_t current_limit;
  struct memblock_type *memory;
  struct memblock_type *reserved;
};

#define IOCTL_BASE 'i'

/* NOTE: _IOW means userland is writing and kernel is reading. _IOR*/
#define IOCTL_TEST _IO(IOCTL_BASE, 1)
#define IOCTL_READ_MEMBLOCK _IOR(IOCTL_BASE, 2, struct user_memblock*)
#define IOCTL_READ_PCIDEV _IOR(IOCTL_BASE, 3, struct user_pci_dev*)

#endif // !__IOCTL_H
```

Листинг 1: ioctl.h

```c
#ifndef __IOCTL_DEV
#define __IOCTL_DEV

struct device_interface {
  atomic_t available;
  struct cdev cdev;
};

int device_open(struct inode *inode, struct file *file);
int device_release(struct inode *inode, struct file *file);
long device_ioctl(struct file *file, unsigned int cmd, unsigned long arg);

#endif // !__IOCTL_DEV
```

Листинг 2: ioctl_dev.h

```c
#include <linux/cdev.h>
#include <linux/export.h>
```

```c
#include <linux/fs.h>
#include <linux/init.h>
#include <linux/kernel.h>
#include <linux/memblock.h>
#include <linux/module.h>
#include <linux/pci.h>

#include "ioctl.h"
#include "ioctl_dev.h"

MODULE_LICENSE("GPL");
MODULE_AUTHOR("Nikita Kulakov");

#define DEVICE_CLASS "devc_ioctl"
#define DEVICE_NAME "dev_ioctl"
char *ioctl_dev_name = DEVICE_NAME;

struct class *dev_class;
dev_t devno = 0;

struct ioctl_message {
  struct memblock memblock;
  struct pci_dev pci_dev;
} ioctl_message;

struct device_interface device_interface;

struct file_operations file_operations = {
    .owner = THIS_MODULE,
    .read = NULL,
    .write = NULL,
    .open = device_open,
    .release = device_release,
    .unlocked_ioctl = device_ioctl,
};

// Functions for manipulating device_interface
static void ioctl_init_device_interface(struct device_interface *interface) {
  memset(interface, 0, sizeof(struct device_interface));
  atomic_set(&interface->available, 1);
}

static void ioctl_del_device_interface(struct device_interface *interface) {
  memset(interface, 0, sizeof(struct device_interface));
}

static int ioctl_setup_cdev(struct device_interface *interface) {
  cdev_init(&interface->cdev, &file_operations);
  return cdev_add(&interface->cdev, devno, 1);
}

// On module load
static void __exit ioctl_mod_exit(void) {
  if (dev_class) {
    device_destroy(dev_class, devno);
    class_destroy(dev_class);
    printk(KERN_INFO "ioctl_mod: dev=%s, class=%s destroyed", DEVICE_NAME,
           DEVICE_CLASS);
  }
  cdev_del(&device_interface.cdev);
  unregister_chrdev_region(devno, 1);
  ioctl_del_device_interface(&device_interface);
  printk(KERN_INFO "ioctl_mod: interface unloaded\n");
}

// On module exit
static int __init ioctl_mod_init(void) {
  int result = 0;

  ioctl_init_device_interface(&device_interface);

  result = alloc_chrdev_region(&devno, 0, 1, DEVICE_NAME);
  if (result < 0) {
    printk(KERN_WARNING "ioctl_mod: can't get major number %d\n", MAJOR(devno));
    goto fail;
  }
```

```
  result = ioctl_setup_cdev(&device_interface);
  if (result < 0) {
    printk(KERN_WARNING "ioctl_mod: error %d setuping cdev\n", result);
    goto fail;
  }
  printk(KERN_INFO "ioctl_mod: interface loaded\n");

  // create file in /dev
  dev_class = class_create(THIS_MODULE, DEVICE_CLASS);
  if (dev_class == NULL) {
    printk(KERN_WARNING "ioctl_mod: can't create device class");
    goto fail;
  }
  if (device_create(dev_class, NULL, devno, NULL, DEVICE_NAME) == NULL) {
    printk(KERN_WARNING "ioctl_mod: can't create device file");
    goto fail;
  }
  printk(KERN_INFO "ioctl_mod: dev=%s, class=%s created", DEVICE_NAME,
         DEVICE_CLASS);
  return 0;

fail:
  ioctl_mod_exit();
  return -1;
}

// public API
int device_open(struct inode *inode, struct file *file) {
  // inode->i_cdev is offset pointed to device_interface
  struct device_interface *dev_interface;
  dev_interface = container_of(inode->i_cdev, struct device_interface, cdev);
  printk(KERN_INFO "ioctl_mod: OPENED");
  if (!atomic_dec_and_test(&dev_interface->available)) {
    atomic_inc(&dev_interface->available);
    printk(KERN_ALERT "ioctl_mod: interface is busy, unable to open\n");
    return -EBUSY;
  }
  return 0;
}

int device_release(struct inode *inode, struct file *file) {
  struct device_interface *dev_interface;
  dev_interface = container_of(inode->i_cdev, struct device_interface, cdev);
  printk(KERN_INFO "ioctl_mod: RELEASED");
  atomic_inc(&dev_interface->available);
  return 0;
}

long device_ioctl(struct file *file, unsigned int cmd, unsigned long arg) {
  switch (cmd) {
  case IOCTL_TEST: {
    uint32_t value;
    printk(KERN_INFO "ioctl_mod: ioctl(IOCTL_TEST)");
    if (copy_from_user(&value, (uint32_t *)arg, sizeof(value))) {
      return -EFAULT;
    }
    printk(KERN_INFO "\treceived=%u\n", value);

    value = 0x12345678;
    if (copy_to_user((uint32_t *)arg, &value, sizeof(value))) {
      return -EFAULT;
    }
    printk(KERN_INFO "\tsent=%u\n", value);

    break;
  }
  case IOCTL_READ_MEMBLOCK: { // convert memblock to user_memblock
    struct user_memblock u_memblock =
        (struct user_memblock){.bottom_up = memblock.bottom_up,
                               .current_limit = memblock.current_limit,
                               .memory = &memblock.memory,
                               .reserved = &memblock.reserved};
    printk(KERN_INFO "ioctl_mod: ioctl(IOCTL_READ_MEMBLOCK)");
    if (copy_to_user((struct user_memblock *)arg, &u_memblock,
                     sizeof(u_memblock))) {
      return -EFAULT;
```

```
    }
    break;
  }
  case IOCTL_READ_PCIDEV: {
    struct pci_dev *dev;
    struct user_pci_dev u_dev;
    printk(KERN_INFO "ioctl_mod: ioctl(IOCTL_READ_PCIDEV)");
    dev = pci_get_device(PCI_ANY_ID, PCI_ANY_ID, NULL);
    u_dev = (struct user_pci_dev){.device = dev->device,
                                  .subsystem_device = dev->subsystem_device,
                                  .subsystem_vendor = dev->subsystem_vendor,
                                  .vendor = dev->vendor,
                                  .devfn = dev->devfn};
    if (copy_to_user((struct user_pci_dev *)arg, &u_dev, sizeof(u_dev))) {
      return -EFAULT;
    }
    break;
  }
  }
  return 0;
}

module_init(ioctl_mod_init);
module_exit(ioctl_mod_exit);
```

Листинг 3: ioctlmod.c

2.2 Листинг пользовательской программы

```
#include <linux/pci.h>

#include <errno.h>
#include <fcntl.h>
#include <inttypes.h>
#include <stdint.h>
#include <stdio.h>
#include <stdlib.h>
#include <sys/ioctl.h>
#include <unistd.h>

#include "../mod/ioctl.h"

#define DEVICE_PATH "/dev/dev_ioctl"

int device_open(const char *device_name);
void device_close(const char *device_name, int fd_device);

int device_open(const char *device_name) {
  printf("[INFO]: Open device\n");
  int fd_device = open(device_name, O_RDWR);
  if (fd_device == -1) {
    printf("[ERROR]: Can't open device with name '%s'\n", device_name);
    exit(EXIT_FAILURE);
  }
  return fd_device;
}

void device_close(const char *device_name, int fd_device) {
  printf("[INFO]: Close device\n");
  int result = close(fd_device);
  if (result == -1) {
    printf("[ERROR]: Can't close device with name '%s'\n", device_name);
    exit(EXIT_FAILURE);
  }
}

// Pretty printing
#define STRUCT(type, pInstance, ...)                                          \
  {                                                                           \
    printf("%s=%p: {\n", #type, &pInstance);                                  \
    type *pStr = &pInstance;                                                  \
    __VA_ARGS__                                                               \
    printf("}\n");                                                            \
  }
#define FIELD(pPat, pInstance)                                                \
  { printf("  %s=%" #pPat "\n", #pInstance, pStr->pInstance); }
// for inttypes...
```

```c
#define PFIELD(pPat, pInstance)                                                 \
  { printf("  %s=%" pPat "\n", #pInstance, pStr->pInstance); }

int main() {
  int fd = device_open(DEVICE_PATH);
  { // test
    uint32_t value = 0x87654321;
    if (ioctl(fd, IOCTL_TEST, &value) < 0) {
      perror("[ERROR]: ioctl");
      exit(EXIT_FAILURE);
    }
    printf("%s: received value(HEX)=%x\n", "IOCTL_TEST", value);
  }
  { // read memblock
    struct user_memblock value;
    if (ioctl(fd, IOCTL_READ_MEMBLOCK, &value) < 0) {
      perror("[ERROR]: ioctl");
      exit(EXIT_FAILURE);
    }
    printf("%s: received ", "IOCTL_READ_MEMBLOCK");
    STRUCT(struct user_memblock, value,
           FIELD(u, bottom_up) PFIELD(PRIu64, current_limit) FIELD(p, memory)
               FIELD(p, reserved))
  }
  { // read pci_dev
    struct user_pci_dev value;
    if (ioctl(fd, IOCTL_READ_PCIDEV, &value) < 0) {
      perror("[ERROR]: ioctl");
      exit(EXIT_FAILURE);
    }
    printf("%s: received ", "IOCTL_READ_PCIDEV");
    STRUCT(struct user_pci_dev, value,
           FIELD(u, devfn) FIELD(u, vendor) FIELD(u, device)
               FIELD(u, subsystem_vendor) FIELD(u, subsystem_device))
  }
  device_close(DEVICE_PATH, fd);
  return EXIT_SUCCESS;
}
```

Листинг 4: main.c

2.3 Скрипты

```
KDIR=/home/nikit/data/kernel/linux-6.0
BUILD_KDIR=$(KDIR)/build
BUILD_DIR=$(PWD)/build

MODNAME=ioctlmod

obj-m += $(MODNAME).o

default:
  make -C $(KDIR) O=$(BUILD_KDIR) M=$(BUILD_DIR) src=$(PWD) modules

clean:
  make -C $(KDIR) O=$(BUILD_KDIR) M=$(BUILD_DIR) src=$(PWD) clean
  rm -rf $(BUILD_DIR)

mv_ko: default
  mv $(BUILD_DIR)/$(MODNAME).ko ./

# export:
#   scp -P ${VM_PORT} $(BUILD_DIR)/$(MODNAME).ko ${VM_ADDRESS}:${VM_PATH}
#   # also copy whole directory for buiding app
#   $(eval PREV_DIR := ${PWD}/)
#   cd ..
#   scp -r -P ${VM_PORT} $(PREV_DIR) ${VM_ADDRESS}:${VM_PATH}
```

Листинг 5: Makefile модуля

```
#!/bin/bash

set -u

BASE_DIR=`pwd`
SCRIPT_DIR=$BASE_DIR/`dirname $0`
```

```
LINUX_DIR=$1
LIST_REQUIRED="build/ Makefile include/ scripts/ arch/"
SCRIPTS="install-kernel.sh test-module.sh"

cd $LINUX_DIR
make O=build ARCH=x86 -j8 vmlinux modules
for script in $SCRIPTS; do cp $SCRIPT_DIR/$script .; done

tar cf $BASE_DIR/export-$LINUX_DIR.tar $LIST_REQUIRED $SCRIPTS
# tar czf $BASE_DIR/export-$LINUX_DIR.tar.gz .
rm $SCRIPTS
```

Листинг 6: build-archive.sh

```
#!/bin/bash

set -u

MODULE=$1
BASE_NAME=`basename $MODULE`
MODULE_NAME=${BASE_NAME%.*}

sudo dmesg -C
sudo insmod $MODULE
sudo rmmod $MODULE_NAME
dmesg
```

Листинг 7: test-module.sh

```
#!/bin/bash

sudo make O=build ARCH=x86 modules_install
sudo make O=build ARCH=x86 install
```

Листинг 8: install-kernel.sh

2.4 Результаты работы программы

```
nikit@vmk:~/data/linux$ make -C lab/app run
make: Entering directory '/home/nikit/data/linux/lab/app'
sudo ./build/app
[INFO]: Open device
IOCTL_TEST: received value(HEX)=12345678
IOCTL_READ_MEMBLOCK: received value struct user_memblock=0x7ffd1b2b1890: {
  bottom_up=0
  current_limit=4831838208
  memory=0xffffffffaba70d90
  reserved=0xffffffffaba70db8
}
IOCTL_READ_PCIDEV: received value struct user_pci_dev=0x7ffd1b2b18b8: {
  devfn=0
  vendor=32902
  device=4663
  subsystem_vendor=0
  subsystem_device=0
}
[INFO]: Close device
make: Leaving directory '/home/nikit/data/linux/lab/app'

nikit@vmk:~/data/linux$ dmesg
[   66.059766] ioctlmod: loading out-of-tree module taints kernel.
[   66.060171] ioctl_mod: interface loaded
[   66.063085] ioctl_mod: dev=dev_ioctl, class=devc_ioctl created
[  126.195155] ioctl_mod: OPENED
[  126.195163] ioctl_mod: ioctl(IOCTL_TEST)
[  126.195164]  received=2271560481
[  126.195166]  sent=305419896
[  126.195950] ioctl_mod: ioctl(IOCTL_READ_MEMBLOCK)
[  126.197450] ioctl_mod: ioctl(IOCTL_READ_PCIDEV)
```

Листинг 9: Output

3 Вывод

В ходе выполнения данной лабораторной работы были выполнены исследования в области того, каким образом можно экспортировать функции ядра, если они изначально недоступны для модулей. Также

было понято, что драйверы - тоже модули ядра, соответственно по заданию требовалось написать свой драйвер. Кроме того, было изучено, как писать свой драйвер ioctl, syscall, собирать ядро и модули с основной машины для виртуальной для более быстрой компиляции.